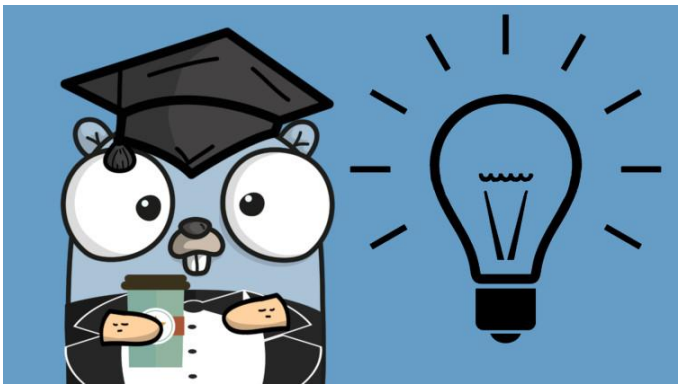
Effektive Go

Die Bibel der Go (Golang) Entwickler



Einleitung

Über Effective Go

Das Dokument Effective Go beschreibt wie die Programmiersprache Go anzuwenden ist. Das Ziel ist es Go so verwenden, wie es von den Machern gedacht ist. Diese Art wird auch als “idiomatischen Code” verstanden.

Codebeispiele sind aus der original Version übernommen:

- https://golang.org/doc/effective_go.html

Effective Go setzt Grundlagen in Go voraus. Es wird empfohlen zuvor folgende Dokumente zuvor zu lesen.

- language specification
- Tour of Go
- How to Write Go Code

Für alle Dokumente ist auch eine deutsche Übersetzung von Hans-Werner Heinzen vorhanden:

- Mit Go arbeiten
- Effektiv Go
- Go-Sprachbeschreibung

In den deutschen Versionen werden wirklich alle Elemente übersetzt, da die Dokumentation von Paketen auf Englisch ist, empfehle ich jedoch so oft wie möglich die Englischen Versionen.

Kursunterlagen

- Diese Gliederung: https://goo.gl/mkXWiX
- Effective Go
- Effektiv Go programmieren (deutsche Übersetzung)

Tipps für den Kurs

- Die Videos können schneller abgespielt werden.
    - Aufnahmefähigkeit ist schneller als meine Erzählgeschwindigkeit.
    - Diese Methode ist nicht für jeden etwas, deshalb einfach mal ausprobieren
- Beim Ansehen des Video
    - “b” drücken, um ein Lesezeichen zu setzen
- Feedback geben

- Bei Problemen einfach eine Frage hier im Kurs stellen

Formatierung

- Einigung über Formatierungsregeln kosten Zeit und Energie
- In Go verwenden wir **gofmt**
    - Teil des go Toolings
    - Editoren verwenden goimports

Aus folgender Definition

```
 type T struct {
    name string // name of the object
    value int // its value
}
```

werden die einzelnen Spalten korrekt ausgerichtet

```
type T struct {
    name    string // name of the object
    value   int    // its value
}
```

- Fehlende Details
    - **Einrückungen**: Default werden Tabs gesetzt. Leerzeichen erlaubt, sollten jedoch nur wenn unbedingt notwendig angewendet werden.
    - **Zeilenlänge**: Keine Einschränkung. Zeilen können umgebrochen werden und danach wird ein extra Tab gesetzt.
        - Wenn Aufzählungen, welche durch “,” getrennt werden umgebrochen werden, dann ist am Zeilenende immer ein “,”

```
s := []string{
    "eins",
    "zwei",
}
```

- Beispiel im Go Playground: https://play.golang.org/p/3RIbEHC2O0v
- Beispiel VSCode

Video: 264

Kommentare

- Block-Kommentare /* */ C-Stil

    - Für Paket-Beschreibungen
    - Deaktivieren von Code
- Zeilenkommentare // C++ Stil
    - Standardfall
    - Kommentare immer mit Leerzeichen
      Falsch: //mein Kommentar
      Richtig: // mein Kommentar
- Programm godoc (auch Webserver)
- Paketkommentar

```
/*
Package regexp implements a simple library for regular expressions.

The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
        '$'
        '.'
        character
        '[' [ '^' ] character-ranges ']'
        '(' regexp ')'
*/
package regexp
```

    - vor der Paket Deklaration
    - bei mehreren Dateien kann dies in einer beliebigen Datei erfolgen
        - es kann auch eine extra Datei erstellt werden: doc.go
    - Soll das Paket vorstellen
    - Wird von godoc als erstes generiert
    - I.d.R. als Blockkommentar bei einfachen Paketen auch einfacher möglich
- Werden in godoc als Text interpretiert
    - Keine Formatierung oder HTML in die Kommentare schreiben
- Doc Kommentar

```
// Compile parses a regular expression and returns, if successful,
// a Regexp that can be used to match against text.
func Compile(str string) (*Regexp, error) {
```

    - Vor der Deklaration
    - Als ganzer Satz erstes Wort soll der Name des jeweiligen Elements sein

- Kommentare bei Variablen Gruppen
    - Einfache Formulierung, da für die ganze Gruppe gültig

```
// Error codes returned by failures to parse an expression.
var (
    ErrInternal      = errors.New("regexp: internal error")
    ErrUnmatchedLpar = errors.New("regexp: unmatched '('")
    ErrUnmatchedRpar = errors.New("regexp: unmatched ')'")
    ...
)
```

- Exkurs: Gruppierung von Variablen kann auch Beziehungen zwischen Variablen darstellen

```
var (
    countLock   sync.Mutex
    inputCount  uint32
    outputCount uint32
    errorCount  uint32
)
```

Video: 265

Namen

- Exportierte Elemente beginnen immer mit einem Großbuchstaben.
- In anderen Sprachen gibt es das Keyword public
- In Go: “exported” und “unexported”
- Analog zu objektorientierten Sprachen “public” und “private”

Video: 266

Paketnamen

- Einfache kurze Namen
- Kleinschreibung
- Keine Angst vor Namenskonflikten, nur weil ein anderes Paket bereits den geeignetsten Namen verwendet
    - Namespace für das Paket kann überschrieben werden
- der Paketname wird über das Quellverzeichnis definiert
- Exportierte Elemente eines Pakets sollen immer in Verbindung mit dem Paketnamen betrachtet werden
    - die Verwendung ist: bufio.Reader

        - Deshalb ist z.B. BufReader nicht gewünscht
- Kurze Funktionsnamen

Video: 267

Getter

- Go hat keine automatische Unterstützung für Getter und Setter
- Get nicht im Funktionsnamen
    - Idiomatisch: obj.Owner()
    - Nicht idiomatisch: obj.GetOwner()

```
type MyType struct {
    owner string
}
owner := obj.Owner()
if owner != user {
    obj.SetOwner(user)
}
```

- Setter dürfen Set im Namen haben
    - obj.SetOwner()

Video: 268

Interface-Namen

- Ein-Methoden-Interface
    - -er Suffix wird an den Funktionsnamen gehängt
    - Read() -> Reader
    - Write() -> Writer
- Funktionen mit bereits anerkannten Signaturen
    - Read, Write, Close, Flush, String
    - Bei verwendung dieser Namen sollen die üblichen Signaturen eingehalten werden
    - Bei der Implementierung soll auch der übliche Name verwendet werden
        - String Converter (Stringer Interface)
            - idiomatisch: String()
            - nicht idiomatisch: ToString()

Video: 269

MixedCaps

Variablen oder Funktionen aus längeren Wörtern werden mit MixedCaps geschrieben

- idiomatisch: ReadWriter, notExported
- nicht idiomatisch: Read_Writer

Video: 270

Semikolons ;

- die formale Go Grammatik verwendet Semikolons
- sind im Code nicht sichtbar
- werden durch den Lexer automatisch hinzugefügt
- Im Code nur bei for loops
- öffnende geschweifte Klammern dürfen nicht in einer neuen Zeile stehen, da sonst das Semikolon falsch gesetzt wird.

Video: 271

Kontrollstrukturen

- Verwandt mit C
    - kein do oder while
    - erweiterte if und switch Anweisungen
        - optionale Initialisierung (Scope liegt dann innerhalb der Anweisung)
    - break und continue akzeptieren optional ein Label

Video: 272

If

```
if x > 0 {
    return y
}
```

- benötigt geschweifte Klammern
- guter Stil, wenn if über mehrere Zeilen läuft
- Initialisierung von Variablen. Achtung Scope ist nur innerhalb des if Statements!

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

- idiomatic: else wird weggelassen, wenn break, continue, goto oder return folgt
    - gilt insbesondere auch für Fehlerbehandlung

```
f, err := os.Open(name)
if err != nil {
```

```
    return err
}
codeUsing(f)
```

Fehlerbehandlung

```
f, err := os.Open(name)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    f.Close()
    return err
}
codeUsing(f, d)
```

Redeklaration und mehrmalige Zuweisung

- Go Funktionen können mehrere Rückgabewerte besitzen
- Kurzdeklaration definiert beide Variablen

```
f, err := os.Open(name)

// nicht erlaubt
err := myFunc()
// erlaubt
d, err := f.Stat()
```

- mehrmalige Deklarationen sind erlaubt wenn:
    - die Variable im gleichen Scope ist (ansonsten würde eine neuer Speicherbereich/Variable dafür angelegt werden)
    - der Wert der neuen Deklaration kann dem vorhandenen Typ zugewiesen werden
    - mindestens eine neue Variable wird durch die Kurzdeklaration neu definiert
- Grund: für die Fehlerbehandlung soll immer einheitlich err verwendet werden.

Video: 274

Schleifen bzw. for

In Go gibt es nur for für die Definition von Schleifen. while oder do-while sind hier nicht vorhanden.

- Arten von for

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```

- Definition der Index Variablen über Kurzdeklaration

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

- Loops über array, slice, string, map oder das Auslesen eines channels werden mit range abgebildet

```
for key, value := range oldMap {
    newMap[key] = value
}
```

Wenn nur der erste Wert gebraucht wird:

```
for key := range m {
    if key.expired() {
        delete(m, key)
    }
}
```

Wenn nur der zweite Wert gebraucht wird

```
sum := 0
for _, value := range array {
    sum += value
}
```

- Range über Text
    - Ein Buchstabe kann in UTF-8 mehr als ein byte benötigen
        - https://www.youtube.com/watch?v=MijmeoH9LT4 (Computerphile)
    - In Go wir ein Buchstabe als rune abgebildet

```
for pos, char := range "日本\x80語" { // \x80 is an illegal UTF-8 encoding
    fmt.Printf("character %#U starts at byte position %d\n", char, pos)
```

```
}
```

Ausgabe:

```
character U+65E5 '日' starts at byte position 0
character U+672C '本' starts at byte position 3
character U+FFFD ''' starts at byte position 6
character U+8A9E '語' starts at byte position 7
```

Abbildung mehrerer Index Variablen

- multiple Zuweisung von Go kann verwendet werden

```
// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```

Video: 275

Switch

- Allgemeiner als in C
- break muss nicht verwendet werden. Wenn ein case zutrifft, dann wird switch auch beendet.
    - möchte man trotzdem weitere Fälle prüfen verwendet man **fallthrough**
- switch ohne Bedingung
    - der Ausdruck unter case wird interpretiert

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

- switch mit Bedingung

```
switch user.Role {
case "admin":
      ...
}
```

Die unterschiedlichen Fälle können auch kommasepariert in einem case angegeben werden

```
func shouldEscape(c byte) bool {
    switch c {
    case ' ', '?', '&', '=', '#', '+', '%':
        return true
    }
    return false
}
```

Break kann auch innerhalb von switch verwendet werden, wenn bestimmte Fälle frühzeitig abgebrochen werden müssen.
Zur break anweisung kann zusätzlich ein Label angegeben werden, das break bezieht sich dann auf den Loop des Labels.
Folgendes Beispiel beinhaltet verschiedene Fälle

```
Loop:
      for n := 0; n < len(src); n += size {
            switch {
            case src[n] < sizeOne:
                  if validateOnly {
                        break
                  }
                  size = 1
                  update(src[n])

            case src[n] < sizeTwo:
                  if n+1 >= len(src) {
                        err = errShortInput
                        break Loop
                  }
                  if validateOnly {
                        break
                  }
                  size = 2
                  update(src[n] + src[n+1]<<shift)
            }
      }
```

Type-Switch

- Der Type Switch kann verwendet werden, um den Typ eines Interfaces zu bestimmen.
  https://play.golang.org/p/xjLhnEXj_V0
- Gilt nicht nur für leere Interfaces:
  https://play.golang.org/p/HF6XGpDDHCS
- switch v.(type) { … }

```
var t interface{}
t = functionOfSomeType()
switch t := t.(type) {
default:
    fmt.Printf("unexpected type %T\n", t)     // %T prints whatever type t
has
case bool:
    fmt.Printf("boolean %t\n", t)             // t has type bool
case int:
    fmt.Printf("integer %d\n", t)             // t has type int
case *bool:
    fmt.Printf("pointer to boolean %t\n", *t) // t has type *bool
case *int:
    fmt.Printf("pointer to integer %d\n", *t) // t has type *int
}
```

Video: 277

Funktionen

Mehrere Rückgabewerte

- eine Funktion kann mehrere Werte zurückgeben
  - Ziel klare Trennung zwischen Wert und Fehler bzw. Erfolgsmeldung
  - `func (file *File) Write(b []byte) (n int, err error)`
- Weitere Anwendung, wenn Zahl und ein Pointer zurückgeliefert wird

```
func nextInt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {
```

```
    }
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x*10 + int(b[i]) - '0'
    }
    return x, i
}
```

Die Anwendung der Funktion wäre dann wie folgt

```
    for i := 0; i < len(b); {
        x, i = nextInt(b, i)
        fmt.Println(x)
    }
```

- https://play.golang.org/p/TNpG4x_NloR

Video: 278

Benannter Rückgabewert

- Rückgabewerte/Ergebnisvariablen können analog wie die Eingangswerte auch mit einem Namen versehen werden
- In dem Fall wird am Anfang eine Variable mit Null Wert initialisiert
- Danach wird nur das Keyword return verwendet (analog VBA)

```
func ReadFull(r Reader, buf []byte) (n int, err error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:]
    }
    return
}
```

- https://play.golang.org/p/omBLyrGtETC
- https://www.calhoun.io/using-named-return-variables-to-capture-panics-in-go/

Video: 279

Defer

- Defer wird am Ende einer Funktion ausgeführt
- Bei mehreren “LIFO” last in first out
    - https://play.golang.org/p/zyXqG8Al2-2
- Anwendung:
    - Freigeben von gebundenen Ressourcen z.B. Close()
    - Aufruf von Funktionen am Ende einer Funktion
        - Beispiel trace und untrace: https://play.golang.org/p/wHGsA5ogYcp
    - recover bei einer panic()

```
// Contents returns the file's contents as a string.
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close()  // f.Close will run when we're finished.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append is discussed
later.
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err  // f will be closed if we return here.
        }
    }
    return string(result), nil // f will be closed if we return here.
}
```

Daten

Speicher Bereitstellung mit new

- zwei Arten für die Bereitstellung von Arbeitsspeicher new und make
- new

    - liefert einen Pointer auf eine Variable mit dem Zero-Wert
    - https://play.golang.org/p/g6CV_LKyfMx

```
type SyncedBuffer struct {
    lock    sync.Mutex
    buffer  bytes.Buffer
}
```

```
p := new(SyncedBuffer)  // type *SyncedBuffer
var v SyncedBuffer      // type  SyncedBuffer
```

- Achtung bei maps oder channels, diese müssen weiterhin mit make() erzeugt werden
    - https://play.golang.org/p/FvUfG7GwttR

Konstruktoren und Composite Literals

New ist nicht immer hilfreich. Im folgenden Beispiel ist der Code unnötig lang

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
    return f
}
```

Ein Composite Literal erzeugt eine Instanz des jeweiligen Typs, innerhalb der geschweiften Klammern können dann die initialen Werte zugewiesen werden.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0}
    return &f
}
```

Am obigen Beispiel liefert die Funktion einen Pointer auf f zurück. In Go bleibt diese Variable auch nach dem Ausführen der Funktion bestehen.
Der Code kann an der Stelle sogar noch weiter verkürzt werden.

```
return &File{fd, name, nil, 0}
```

Damit das obige Beispiel funktioniert müssen alle Parameter in der korrekten Reihenfolge gesetzt werden. Wenn nur bestimmte Parameter gesetzt werden sollen, dann sieht die Anweisung wie folgt aus:

```
return &File{fd: fd, name: name}
```

Können auch für arrays, slices und maps verwendet werden:

```
a := [...]string   {Enone: "no error", Einval: "invalid argument"}
s := []string      {Enone: "no error", Einval: "invalid argument"}
m := map[int]string{Enone: "no error", Einval: "invalid argument"}
```

https://play.golang.org/p/mUB_dMLruTN

Speicher Bereitstellung mit make

- für zusammengesetzte Typen besitzen “unter der Haube” mehrere andere Datentypen
    - slices
    - maps
        - Muss mit make erzeugt werden, da es sonst zu einer Panic kommt
        - https://play.golang.org/p/L6dFdmbdZBf
    - channels

Unterschied zwischen make und new

```
var p *[]int = new([]int)       // allocates slice structure; *p == nil; rarely useful
var v  []int = make([]int, 100) // the slice v now refers to a new array of 100 ints

// Unnecessarily complex:
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// Idiomatic:
v := make([]int, 100)
```

Arrays

- Arrays werden in Go benötigt, um Slices abzubilden. Es wird empfohlen arrays nicht zu verwenden.
- Die Verwendung von Arrays ist nicht idiomatischer Go Code
- Arrays in Go sind anders als in C
    - Arrays sind Werte
    - Bei einer Zuweisung werden immer alle Elemente kopiert
    - Die größe eines Arrays ist Teil des Typs
        - D.h. [10]int ist ein anderer Typ wie [20]int

```
func Sum(a *[3]float64) (sum float64) {
    for _, v := range *a {
        sum += v
    }
    return
}

array := [...]float64{7.0, 8.5, 9.1}
x := Sum(&array)  // Note the explicit address-of operator
```

Slices

- sind allgemeiner wie ein Array.
- es wird empfohlen wenn möglich Slices zu verwenden
- Slices haben keine feste Länge
- Slices besitzen eine Referenz auf das darunterliegende Array. Dadurch werden bei einer Zuweisung nur die Referenzen kopiert.
- Deshalb kann anstatt eines Pointers ein Slice an eine Funktion übergeben werden.

```
func (f *File) Read(buf []byte) (n int, err error)
```

- https://play.golang.org/p/omBLyrGtETC

```
func main() {
    buf := make([]int, 2)
    a(buf)
    fmt.Println(buf)
}
func a(b []int) {
    b[0] = 1
```

```
    b[1] = 2
}
```

https://play.golang.org/p/YpKgm70bZPy

- Die Länge von Slices darf geändert werden
- cap() gibt die Kapazität des darunterliegenden Arrays aus
- Wenn das Array voll ist, dann wird alles in ein neues größeres Array kopiert

```
func Append(slice, data []byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) {  // reallocate
        // Allocate double what's needed, for future growth.
        newSlice := make([]byte, (l+len(data))*2)
        // The copy function is predeclared and works for any slice
type.
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:l+len(data)]
    copy(slice[l:], data)
    return slice
}
```

Die Append Funktion ist so nützlich, dass diese in die Sprache aufgenommen wurde.

Zweidimensionale Slices

Arrays und Slices sind eindimensional. Um zweidimensionale Arrays oder Slices herzustellen wird ein Array eines Arrays bzw. ein Slice eines Slice definiert:

```
type Transform [3][3]float64  // A 3x3 array, really an array of arrays.
type LinesOfText [][]byte     // A slice of byte slices.
```

Da die Länge des jeweiligen Slices variabel ist, können unterschiedlich lange Einträge generiert werden:

```
text := LinesOfText{
      []byte("Es ist an der Zeit,"),
      []byte("dass die coolen Ziesel"),
      []byte("Schwung in die Party bringen."),
}
```

In manchen Fällen kann es notwendig sein für 2D Slices den kompletten erwarteten Speicher bereitzustellen. Zum Beispiel bei der Verarbeitung eines Bildes. Hier gibt es zwei Möglichkeiten:

- jede Zeile einzeln anzulegen
- Speichern aller Elemente in einem Slice und dann zeilenweise eine Referenz darauf herzustellen.

```
// Allocate the top-level slice.
picture := make([][]uint8, YSize) // One row per unit of y.
// Loop over the rows, allocating the slice for each row.
for i := range picture {
      picture[i] = make([]uint8, XSize)
}
```

```
// Allocate the top-level slice, the same as before.
picture := make([][]uint8, YSize) // One row per unit of y.
// Allocate one large slice to hold all the pixels.
pixels := make([]uint8, XSize*YSize) // Has type []uint8 even though
picture is [][]uint8.
// Loop over the rows, slicing each row from the front of the remaining
pixels slice.
for i := range picture {
      picture[i], pixels = pixels[:XSize], pixels[XSize:]
}
```

Maps

- Verknüpfen die Werte eines Typs (dem Key) mit dem eines anderen Typs (dem Wert)
- Zulässige Keys: Integers, Fließkommazahlen, komplexe Zahlen, Strings, Pointers, Interfaces (solange eine Gleichheitsprüfung vorgenommen werden kann), Structs und Arrays
- Slices dürfen nicht als Schlüssel verwendet werden
- Beinhalten eine Referenz auf die Values (analog Slices)
- Maps können über Composite Literals erzeugt werden

```
var timeZone = map[string]int{
    "UTC":  0*60*60,
    "EST": -5*60*60,
    "CST": -6*60*60,
    "MST": -7*60*60,
    "PST": -8*60*60,
```

```
}
```

- Verwendung der Map ist analog wie bei Arrays und Slices, nur das der Index kein Integer sein muss
- Wenn ein zu einem nicht existierenden Schlüssel ein Wert abgefragt wird, so wird ein Zero-Value des Typs zurückgegeben.

Ein Set kann als Map mit einem bool angelegt werden. Wenn nun der Eintrag auf true gesetzt wird, dann kann direkt darauf geprüft werden.

```
attended := map[string]bool{
    "Ann": true,
    "Joe": true,
    ...
}

if attended[person] { // will be false if person is not in the map
    fmt.Println(person, "was at the meeting")
}
```

Manchmal muss aber geprüft werden, ob ein Wert in der Map vorhanden ist. Die Abfrage liefert als zweiten Parameter ein bool zurück. Der Wert wird auch “comma ok” genannt.

```
var seconds int
var ok bool
seconds, ok = timeZone[tz]
```

Ein Anwendungsbeispiel einer Funktion mit einer guten Fehlermeldung ist:

```
func offset(tz string) int {
    if seconds, ok := timeZone[tz]; ok {
        return seconds
    }
    log.Println("unknown time zone:", tz)
    return 0
}
```

Mit dem “blank identifier” kann auch nur geprüft werden, ob ein Key existiert:

```
_, present := timeZone[tz]
```

Ein Eintrag (inklusive dem Schlüssel) wird mit delete() gelöscht:

```
delete(timeZone, "PDT")  // Now on Standard Time
```

Video: 289

Printing

- formatierte Ausgabe ist ähnlich wie in C printf
- Teil des fmt Pakets (fmt.Printf, fmt.Sprintf, fmt.Fprintf)
- Weitere Funktionen: fmt.Print und fmt.Println
- fmt.Fprint verwendet das io.Writer interface

```
fmt.Printf("Hello %d\n", 23)
fmt.Fprint(os.Stdout, "Hello ", 23, "\n")
fmt.Println("Hello", 23)
fmt.Println(fmt.Sprint("Hello ", 23))
```

- Die Printroutine entscheidet anhand des Typs, wie diese ausgegeben wird:

```
var x uint64 = 1<<64 - 1
fmt.Printf("%d %x; %d %x\n", x, x, int64(x), int64(x))
```

```
18446744073709551615 ffffffffffffffff; -1 -1
```

- Das “catchall Format” %v kann jeden Typ ausgeben, auch Arrays, Slices, Structs oder Maps

```
fmt.Printf("%v\n", timeZone)  // or just fmt.Println(timeZone)
```

Gibt aus:

```
map[CST:-21600 PST:-28800 EST:-18000 UTC:0 MST:-25200]
```

- %+v gibt bei structs Zusätzlich die Feldnamen aus
- %#v gibt den Wert in Go Syntax aus

```
type T struct {
    a int
    b float64
    c string
}
t := &T{ 7, -2.35, "abc\tdef" }
fmt.Printf("%v\n", t)
fmt.Printf("%+v\n", t)
fmt.Printf("%#v\n", t)
fmt.Printf("%#v\n", timeZone)
```

Ausgabe:

```
&{7 -2.35 abc   def}
&{a:7 b:-2.35 c:abc     def}
```

```
&main.T{a:7, b:-2.35, c:"abc\tdef"}
map[string] int{"CST":-21600, "PST":-28800, "EST":-18000, "UTC":0, "MST":-25200}
```

- %q verwendet Backquotes `wenn möglich`
- %T gibt den Typ eines Wertes aus
- Um die Ausgabe von eigenen Typen anzupassen muss das Stringer Interface implementiert werden.

```
func (t *T) String() string {
    return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)
}
fmt.Printf("%v\n", t)
```

Erzeugt:

```
7/-2.35/"abc\tdef"
```

- Wenn String() des eigenen Typen Sprintf() verwendet, kann es zu einem nicht endenen Loop kommen

```
type MyString string

func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", m) // Error: will recur forever.
}
```

Einfacher Fix durch Typumwandlung

```
type MyString string
func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", string(m)) // OK: note conversion.
}
```

Weitergabe von Argumenten innerhalb von Print Funktionen:

```
// Println prints to the standard logger in the manner of fmt.Println.
func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...))  // Output takes parameters (int,
string)
}
```

- "…" um v (Typ []interface{}) an eine variadic Function zu übergeben.
- weitere Verben und Printfunktionen: https://golang.org/pkg/fmt/#Printf

Exkurs die … Parameter können von jedem beliebigen Typ sein:

```
func Min(a ...int) int {
    min := int(^uint(0) >> 1)  // largest int
```

```
    for _, i := range a {
        if i < min {
            min = i
        }
    }
    return min
}
```

Video: 290

Append

Signatur:

```
func append(slice []T, elements ...T) []T
```

- T ist ein Platzhalter für jeden beliebigen Typ
- So eine Funktion ist in Go selber nicht umsetzbar, da der Compiler hier unterstützen muss. Deshalb ist append() direkt in die Sprache aufgenommen worden
- append() fügt am Ende eines Slice eines oder mehrere Elemente hinzu
- Da das darunterliegende Array sich ändern kann (und somit eine Referenz auf ein neues Array erzeugt wird), muss hier ein Wert zurückgegeben werden.

Anwendung:

```
x := []int{1,2,3}
x = append(x, 4, 5, 6)
fmt.Println(x)
```

- append ist eine variadic function, deshalb können mit … auch Slices zu Slices hinzugefügt werden.

```
x := []int{1,2,3}
y := []int{4,5,6}
x = append(x, y...)
fmt.Println(x)
```

Video: 291

Initialisierung

- Obwohl die Initialisierung ähnlich wie in C oder C++ aussieht, ist Go an der Stelle leistungsfähiger
- Komplexe Strukturen können erzeugt werden
- Go kümmert sich um die richtige Reihenfolge der Initialisierung, selbst zwischen unterschiedlichen Paketen

Video: 292

Konstanten

- Werden fest in den Code compiliert
- Nur Nummern, Runes, Strings oder Boolesche Werte
    - muss durch den Compiler erzeugt werden können
- iota
    - Zähler für die automatische Zuweisung von mehreren Konstanten

```
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
)
```

Wenn man für dieses Beispiel die Stringer Methode implementiert, dann lässt sich hier der Output automatisch anpassen.

```
func (b ByteSize) String() string {
    switch {
    case b >= YB:
        return fmt.Sprintf("%.2fYB", b/YB)
    case b >= ZB:
        return fmt.Sprintf("%.2fZB", b/ZB)
    case b >= EB:
        return fmt.Sprintf("%.2fEB", b/EB)
    case b >= PB:
        return fmt.Sprintf("%.2fPB", b/PB)
    case b >= TB:
        return fmt.Sprintf("%.2fTB", b/TB)
    case b >= GB:
        return fmt.Sprintf("%.2fGB", b/GB)
    case b >= MB:
        return fmt.Sprintf("%.2fMB", b/MB)
    case b >= KB:
        return fmt.Sprintf("%.2fKB", b/KB)
    }
```

```
    return fmt.Sprintf("%.2fB", b)
}
```

An der Stelle kann es nicht zu einer unendlichen Rekursion kommen, da %f verwendet wird, welches nicht die String() Funktion aufruft.
Video: 293

Variablen

Variablen werden ähnlich wie Konstanten initialisiert , jedoch können hier alle Funktionen verwendet werden, da die Zuweisung der Werte auch zur Laufzeit erfolgen kann.

```
var (
    home   = os.Getenv("HOME")
    user   = os.Getenv("USER")
    gopath = os.Getenv("GOPATH")
)
```

Video: 294

Die init Funktion

- jede Datei kann ein oder mehrere init Funktionen besitzen
  https://play.golang.org/p/cpLCXRmVrXe
- Aufruf nachdem die Paket Variablen initialisiert wurden
  - Paket Variablen werden nach den importierten Paketen initialisiert
- Anwendung: Reparieren und Sichern eines korrekten Standes, bevor die Ausführung beginnt.

```
func init() {
    if user == "" {
        log.Fatal("$USER not set")
    }
    if home == "" {
        home = "/home/" + user
    }
    if gopath == "" {
        gopath = home + "/go"
    }
    // gopath may be overridden by --gopath flag on command line.
    flag.StringVar(&gopath, "gopath", gopath, "override default GOPATH")
}
```

Video: 295

Methoden

Pointers vs. Values

- Methoden können für alle Typen definiert werden
- Beispiel:
    - ByteSlice implementiert die Append Funktion
    - []byte muss zurück geliefert werden, da sich die Referenz zum darunter liegenden Array ändert

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    // Body exactly the same as the Append function defined above.
}
```

Besser Methode auf den Pointer setzen:

```
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // Body as above, without the return.
    *p = slice
}
```

Noch viel besser, wenn wir die Write() Methode implementieren

```
func (p *ByteSlice) Write(data []byte) (n int, err error) {
    slice := *p
    // Again as above.
    *p = slice
    return len(data), nil
}
```

Jetzt implementiert *ByteSlice den io.Writer, was das Zusammenspiel mit vielen Funktionen der Standard Library ermöglicht.

```
   var b ByteSlice
    fmt.Fprintf(&b, "This hour has %d days\n", 7)
```

- &b muss verwendet werden
- nur *ByteSlice (Pointer) implementiert die Write() Funktion
    - Grund: Pointer Methoden können den Wert verändern
- Value Funktionen können auch über Pointer aufgerufen werden

- Für den Aufruf von Methoden kann auch `b.Write()` verwendet werden. Solange `b` Adressierbar ist (d.h. &b ergibt keinen Fehler) korrigiert der Compiler den Ausdruck zu `(&b).Write()`
- Übrigens: Write für ein slice of byte zu Verwenden ist die zentrale Idee hinter dem `bytes.Buffer`

Video: 296

Interfaces und andere Typen

Interfaces

- Interface ist ein Typ, welcher das Verhalten eines Objektes beschreibt
- “Es kann das machen”
- Typisch für Go sind kurze Interfaces (ein bis zwei Methoden)
- Ein Typ kann mehrere Interfaces implementieren

Beispiel: sort.Interface und fmt.Stringer

```
type Sequence []int

// Methods required by sort.Interface.
func (s Sequence) Len() int {
    return len(s)
}
func (s Sequence) Less(i, j int) bool {
    return s[i] < s[j]
}
func (s Sequence) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// Method for printing - sorts the elements before printing.
func (s Sequence) String() string {
    sort.Sort(s)
    str := "["
    for i, elem := range s {
        if i > 0 {
            str += " "
        }
        str += fmt.Sprint(elem)
    }
    return str + "]"
}
```

Video: 297

Umwandlungen

- Um mit unterschiedlichen Typen arbeiten zu können, müssen die Typen auch umgewandelt werden.
- Für das Beispiel Sequence könnte man die String Methode vereinfachen

```
func (s Sequence) String() string {
    sort.Sort(s)
    return fmt.Sprint([]int(s))
}
```

- Die Funktion `[]int()` wandelt `s` temporär um. Dies ist möglich, da `[]int` der zugehörige Basistyp ist.
- Diese Logik ermöglicht es zu sortieren, indem intern eine Typumwandlung erzeugt wird. In dem Fall müsste dann das sort.Interface gar nicht implementiert werden.

```
type Sequence []int
// Method for printing - sorts the elements before printing
func (s Sequence) String() string {
    sort.IntSlice(s).Sort()
    return fmt.Sprint([]int(s))
}
```

Video: 298

Interface Umwandlungen und Typ Zuweisung

- Type switches sind eine Art von Umwandlung für Interfaces
- Ein interface wird als Input verwendet und innerhalb des case wird der entsprechende Typ erzeugt

Vereinfachtes Beispiel wie fmt.Printf arbeitet:

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

var value interface{} // Value provided by caller.
switch str := value.(type) {
case string:
    return str
case Stringer:
    return str.String()
}
```

- Wenn nur ein bestimmter Typ zugewiesen wird, kann auch eine Typ Zuweisung (=type assertion) verwendet werden

```
value.(typeName)
str := value.(string)
```

- Wenn diese Zuweisung nicht möglich ist, dann kommt es zu einen harten Fehler zur Laufzeit
- Prüfung mit dem “comma ok” Ausdruck

```
str, ok := value.(string)
if ok {
    fmt.Printf("string value is: %q\n", str)
} else {
    fmt.Printf("value is not a string\n")
}
```

Der Type Switch von oben kann auch so abgebildet werden:

```
if str, ok := value.(string); ok {
    return str
} else if str, ok := value.(Stringer); ok {
    return str.String()
}
```

Video: 299

Allgemeingültigkeit von Konstruktoren

- Wenn ein Typ nur ein interface implementiert und sonst keine weiteren exportierten Methoden besitzt.
- Der Konstruktor soll dann nur das Interface als Rückgabewert besitzen
- Code wird dadurch flexibler, da nur der Konstruktor Aufruf geändert werden muss.
- Beispiel:
    - cipher.Block interface
    - aes.NewCipher
    - des.NewCipher

```
c := aes.NewCipher(key)
block, err := c.Encrypt(dst, src)
...
// Umstellung auf des
c := des.NewCipher(key)
block, err := c.Encrypt(dst, src)
...
```

Video: 300

Interfaces und Methoden

Ein gutes Beispiel wie interfaces und Methoden gehandhabt werden liefert das http Paket.

Das Handler interface wird wie folgt definiert:

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

- ResponseWriter ist selber auch ein Interface, welches die Write() Methode besitzt.
- D.h. dieser implementiert auch immer den io.Writer
- Siehe hier w kann mit Fprintf verwendet werden

```
// Simple counter server.
type Counter struct {
    n int
}

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ctr.n++
    fmt.Fprintf(w, "counter = %d\n", ctr.n)
}
```

Einbindung des Counters in den URL Baum:

```
import "net/http"
...
ctr := new(Counter)
http.Handle("/counter", ctr)
```

Counter kann noch vereinfacht werden. Ein int reicht in dem Fall:

```
// Simpler counter server.
type Counter int

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    *ctr++
    fmt.Fprintf(w, "counter = %d\n", *ctr)
}
```

Über einen Channel kann der Request weiter geleitet werden. Entkoppelung zwischen HTTP Server Anfragen und dem Hauptprogramm.

```
// A channel that sends a notification on each visit.
// (Probably want the channel to be buffered.)
type Chan chan *http.Request

func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ch <- req
    fmt.Fprint(w, "notification sent")
}
```

Weiteres Beispiel: **Methoden für Funktionen**
Wir wollen die Argumente ausgeben, mit welchen das Programm gestartet wurde:

```
func ArgServer() {
    fmt.Println(os.Args)
}
```

Wie kann nun dieser ArgsServer für einen HTTP Server verwendet werden?

Schritt 1: Wir definieren einen Typ HandlerFunc
Schritt 2: Für diesen Typ definieren wir die ServeHTTP Methode

```
// The HandlerFunc type is an adapter to allow the use of
// ordinary functions as HTTP handlers.  If f is a function
// with the appropriate signature, HandlerFunc(f) is a
// Handler object that calls f.
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, req).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, req *Request) {
    f(w, req)
}
```

Schritt 3: Wir passen die ArgServer Funktion an, dass die Signatur unserer HandlerFunc entspricht

```
// Argument server.
func ArgServer(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, os.Args)
}
```

Schritt 4: wir verwenden die Typumwandlung, um den ArgServer intern als HandlerFunc zu verwenden

```
http.Handle("/args", http.HandlerFunc(ArgServer))
```

Video: 301

Der blank identifier

- Kann für jeden Wert und Typ stehen
- Lässt den Wert ins Nichts laufen und stellt praktisch einen Platzhalter dar
- Wie in Unix /dev/null

Der blank identifier bei mehreren Rückgabewerten

- die Verwendung bei range ist nur eine Anwendung
- allgemeiner: Verwendung bei mehreren Rückgabewerten
- Alle Variablen in Go müssen verwendet werden
- Wenn eine Funktion mehrere Werte zurück liefert, aber nicht alle verwendet werden

```
if _, err := os.Stat(path); os.IsNotExist(err) {
      fmt.Printf("%s does not exist\n", path)
}
```

ACHTUNG: nicht den Fehler ins Leere laufen lassen!

```
// Bad! This code will crash if path does not exist.
fi, _ := os.Stat(path)
if fi.IsDir() {
    fmt.Printf("%s is a directory\n", path)
}
```

Video: 302

Nicht verwendete Importe und Variablen

- Fehler wenn,
    - ein Paket importiert aber nicht verwendet wird
    - eine Variable deklariert aber nicht verwendet wird

Folgendes Beispiel kompiliert nicht, da die Importe fmt und io und die Variable fd nicht verwendet wird (https://play.golang.org/p/Vs2oQ2XH-NK) :

```
package main

import (
    "fmt"
    "io"
    "log"
```

```
    "os"
)

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: use fd.
}
```

Mit dem _ können die Pakete und Variablen verwendet werden, damit es zu keinen Fehler mehr kommt:

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

var _ = fmt.Printf // For debugging; delete when done.
var _ io.Reader    // For debugging; delete when done.

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: use fd.
    _ = fd
}
```

Wenn Pakete so verwendet werden, sollen die Definitionen gleich nach den Importen folgen, damit diese später einfacher gefunden werden.
Video: 303

Importe für Nebeneffekte

- nicht verwendete Importe wie fmt oder io aus dem letzten Beispiel sollten gelöscht werden.
- Manchmal ist es sinnvoll ein Paket wegen dessen Nebeneffekte zu importieren
    - net/http/pprof registriert in der init() Funktion http Handler
    - Import sieht wie folgt aus:

```
import _ "net/http/pprof"
```

Video: 304

Interface Prüfungen

- In Go muss im Code nicht angegeben werden, welche Interfaces durch einen Typ implementiert werden.
- Die Prüfung wird durch den Compiler durchgeführt

Manche Prüfungen können auch zur Runtime ablaufen. So prüft des JSON Encoder, ob der übergebene Typ den json.Marshaler implementiert. Falls ja, so wird die Methode direkt benutzt. Die Prüfung zur Runtime sieht wie folgt aus:

```
m, ok := val.(json.Marshaler)
```

Wenn nur die Implementierung geprüft werden soll:

```
if _, ok := val.(json.Marshaler); ok {
    fmt.Printf("value %v of type %T implements json.Marshaler\n", val, val)
}
```

Wenn innerhalb eines Paketes geprüft werden soll, ob ein Typ ein Interface implementiert, kann man leere Paketvariablen anlegen. Sobald eine Umwandlung nicht möglich ist kommt es dann zu einem Compiler Error:

```
var _ json.Marshaler = (*RawMessage)(nil)
var _ io.Reader = (*meinTyp)(nil)
```

Video: 305

Embedding - Einbinden

- In Go gibt es keine Unterklassen
- Dafür kann das Embedding verwendet werden
- Ein ReadWriter kann wie folgt definiert werden:

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

type Writer interface {
    Write(p []byte) (n int, err error)
}
// ReadWriter is the interface that combines the Reader and Writer
```

```
interfaces.
type ReadWriter interface {
    Reader
    Writer
}
```

- Nur interfaces können in interfaces eingebunden werden, denn ein interface kann ja kein Feld aufnehmen

Die gleiche Idee funktioniert auch für structs. Beispiel aus bufio Paket

```
// ReadWriter stores pointers to a Reader and a Writer.
// It implements io.ReadWriter.
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}

// NewReadWriter allocates a new ReadWriter that dispatches to r and w.
func NewReadWriter(r *Reader, w *Writer) *ReadWriter {
    return &ReadWriter{r, w}
}
```

Die definition kann auch anders aussehen. Damit dieses Struct auch den io.Reader implementiert muss dann die Methode Read an den Reader durchgereicht werden.

```
type ReadWriter struct {
    reader *Reader
    writer *Writer
}
func (rw *ReadWriter) Read(p []byte) (n int, err error) {
    return rw.reader.Read(p)
}
```

Embedding erlaubt auch einen gewissen Comfort

```
type Job struct {
    Command string
    *log.Logger
}
```

Der Typ Job besitzt nun gleich die Methoden Log und Logf aus dem Logger interface. Man könnte dem Logger auch einen Feldnamen zuweisen, aber das ist an der Stelle gar nicht notwendig. Dadurch kann job ganz einfach Log() verwenden.

```
job.Log("starting now...")
```

Der Logger ist nun auch ein normales Feld. Der Konstruktor bzw. das composite literal können das Feld setzen:

```
func NewJob(command string, logger *log.Logger) *Job {
    return &Job{command, logger}
}
```

Sollte der Logger direkt angesprochen werden, so dient der Name des eingebundenen Typs als Feldname:

```
func (job *Job) Logf(format string, args ...interface{}) {
    job.Logger.Logf("%q: %s", job.Command, fmt.Sprintf(format, args...))
}
```

Namenskonflikte werden dabei wie folgt gelöst:

- Wenn der Typ Job ein Feld oder eine Methode mit dem gleichen Namen des eingebundenen Elementes besitzt, so gewinnt das Element des Typs Job
- Konflikte auf gleicher Ebene ergeben einen Fehler. Wenn der Logger ein Feld oder Methode Logger hat, dann kommt es zu einem Fehler, wenn job.Logger verwendet wird

Video: 306

Nebenläufigkeit - Concurrency

Teile Speicher durch Kommunikation

- Nebenläufige Programmierung ist ein komplexes Thema
- In anderen Sprachen werden Daten zwischen den einzelnen Prozessen durch geteilte Variablen (Referenz auf Arbeitsspeicherbereiche) geteilt
- Go verwendet dafür einen anderen Ansatz. Werte werden durch Channels ausgetauscht.
- Per Definition hat immer nur eine Goroutine zu einem Zeitpunkt Zugriff auf den Wert.
- Data Races können so per Definition nicht passieren
    - Data Race: https://de.wikipedia.org/wiki/Race_Condition

Do not communicate by sharing memory; instead, share memory by communicating.

- Ausprägung des Slogans kann auch zu weit getrieben werden.
- Ein Zähler kann an der Stelle durch ein mutex geschützt werden.
- Jedoch Channels ermöglichen in der Regel eine bessere Umsetzung der Zugriffkontrolle von Goroutinen

Goroutinen

- In Go werden die unabhängig laufenden Threads Goroutinen genannt
- Ist eine Funktion, welche nebenläufig mit anderen Goroutinen im gleichen Namensraum läuft
- Sie sind sehr billig (wenig Speicher, wenig CPU)
- Goroutinen werden auf verschiedene OS Threads aufgeteilt. Der Entwickler muss sich jedoch nicht um die Threads kümmern. Go erledigt das.
- Wird mit dem Befehl go gestartet

```
go list.Sort()  // run list.Sort concurrently; don't wait for it.
```

Hierfür kann auch ein function literal (anonyme Funktion) verwendet werden:

```
func Announce(message string, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }()  // Note the parentheses - must call the function.
}
```

- function literal agiert hier als Closure
- Umgebende Variablen behalten die Gültigkeit solange diese benötigt werden

Video: 308

Channels

- werden analog wie maps mit make erzeugt.
- optional kann ein Integer übergeben werden. Hierdurch wird ein buffered Channel erzeugt

```
ci := make(chan int)            // unbuffered channel of integers
cj := make(chan int, 0)         // unbuffered channel of integers
cs := make(chan *os.File, 100)  // buffered channel of pointers to Files
```

Beispiel: Channel blockiert das Programm, solange kein Signal in den Channel gesteckt wird.

```
c := make(chan int)  // Allocate a channel.
// Start the sort in a goroutine; when it completes, signal on the channel.
go func() {
    list.Sort()
```

```
    c <- 1  // Send a signal; value does not matter.
}()
doSomethingForAWhile()
<-c   // Wait for sort to finish; discard sent value.
```

- der Channel als Empfänger blockiert die Programmausführung bis Daten geliefert werden.

https://play.golang.org/p/O5LKsb7wf7x

```
func main() {
    c := make(chan bool)
    go func() {
       time.Sleep(time.Second * 3)
       fmt.Println(<-c)
    }()
    c <- true
    fmt.Println("Hello, playground")
}
```

- Bei buffered Channels werden die Daten in einen Puffer geschrieben.
    - Wenn Puffer frei ist blockiert dieser nicht beim Schreiben
    - Wenn Daten im Puffer sind blockiert dieser nicht beim lesen

```
var sem = make(chan int, MaxOutstanding)

func handle(r *Request) {
    sem <- 1    // Wait for active queue to drain.
    process(r)  // May take a long time.
    <-sem       // Done; enable next request to run.
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req)  // Don't wait for handle to finish.
    }
}
```

- sem ist ein Buffered Channel, die Anzahl der gepufferten Elemente entspricht die Anzahl der maximal gewünschten Goroutinen
- mit sem <- 1 wird ein Wert im Buffer belegt und <- sem wieder freigegeben.
- Sobald der Puffer voll ist blockiert sem <- 1 bis dort wieder Platz ist
- Nachteil an dem Beispiel ist, dass Serve() unabhängig MaxOutstanding eine Goroutine erzeugt

Folgende Umstrukturierung löst dieses Problem:

```
func Serve(queue chan *Request) {
    for req := range queue {
        sem <- 1
        go func() {
            process(req) // Buggy; see explanation below.
            <-sem
        }()
    }
}
```

- Der Bug an der Stelle ist, dass req von allen Goroutinen verwendet wird
- Indem wir req als Input des function literals übergeben lösen wird dieses Problem

```
func Serve(queue chan *Request) {
    for req := range queue {
        sem <- 1
        go func(req *Request) {
            process(req)
            <-sem
        }(req)
    }
}
```

Eine andere Lösung ist es eine “neue” Variable zu erzeugen:

```
func Serve(queue chan *Request) {
    for req := range queue {
        req := req // Create new instance of req for the goroutine.
        sem <- 1
        go func() {
            process(req)
            <-sem
        }()
    }
}
```

In jedem Loop wird so eine neue Variable erzeugt, welche dann auch der Goroutine zur Verfügung steht.

Eine alternative Lösung für den Server ist es gleich die maximale Anzahl an Goroutinen zu starten und dann über den Channel die Requests zu übermitteln:

```
func handle(queue chan *Request) {
    for r := range queue {
        process(r)
    }
}
```

```
func Serve(clientRequests chan *Request, quit chan bool) {
    // Start handlers
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
    }
    <-quit  // Wait to be told to exit.
}
```

Video: 310

Channels of Channels

- Channels in Go sind First Class Values, so wie andere Typen auch
- Werden verwendet, um parallele Anweisungen sicher implementieren zu können.

In dem vorherigen Kapitel hatten wir den Typ, welchen der Handler verarbeitet nicht spezifiziert. Wenn wir dort einen Channel für die Ergebnisse hinzufügen, kann jeder Client hierüber auf die jeweilige Antwort warten.
Request kann ungefähr wie folgt aussehen:

```
type Request struct {
    args        []int
    f           func([]int) int
    resultChan  chan int
}
```

- Der Client stellt die Funktion, deren Argumente und einen Channel für das Ergebnis.
- über den Channel clientRequest wird der komplette Request an den Server geschickt

```
func sum(a []int) (s int) {
    for _, v := range a {
        s += v
    }
    return
}

request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// Send request
clientRequests <- request
// Wait for response.
fmt.Printf("answer: %d\n", <-request.resultChan)
```

Die Server Seite sieht dabei wie folgt aus:

```
func handle(queue chan *Request) {
    for req := range queue {
```

```
        req.resultChan <- req.f(req.args)
    }
}
```

An der Stelle ist das Beispiel stark vereinfacht, aber das Prinzip wird dabei klar. Das Beste daran ist, dass das komplette Konstrukt ohne mutex auskommt.
Video: 311

Parallelisierung

- Wenn mehrere CPU vorhanden sind kommt man schnell auf die Idee auch alle CPUs gleichzeitig zu nutzen.
- Wenn die Berechnungen unabhängig voneinander ablaufen können, so können diese auch gleichzeitig bzw. parallel durchgeführt werden.

Angenommen wir müssen aufwändige Berechnungen mit einem Vektor anstellen und das diese Operationen unabhängig voneinander sind

```
type Vector []float64

// Apply the operation to v[i], v[i+1] ... up to v[n-1].
func (v Vector) DoSome(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
    c <- 1    // signal that this piece is done
}
```

Wir lassen die Berechnungen nun über einen Loop unabhängig als Goroutine laufen. Indem wir in den Channel <-c lauschen, warten wir bis die Ausführung fertig ist.

```
const numCPU = 4 // number of CPU cores

func (v Vector) DoAll(u Vector) {
    c := make(chan int, numCPU)  // Buffering optional but sensible.
    for i := 0; i < numCPU; i++ {
        go v.DoSome(i*len(v)/numCPU, (i+1)*len(v)/numCPU, u, c)
    }
    // Drain the channel.
    for i := 0; i < numCPU; i++ {
        <-c    // wait for one task to complete
    }
    // All done.
}
```

Anstatt die CPU Zahl festzulegen, kann auch die Runtime diesen Wert liefern

```
var numCPU = runtime.NumCPU()
```

Alternativ gibt es auch die Funktion `runtime.GOMAXPROCS` über welche der User festlegen kann, wie viele CPUs verwendet werden sollen.

Video: 312

A leaky buffer - Ein Eimer mit Loch

- Die Tools für Concurrency machen auch die herkömmliche Programmierung leichter

Hier ein stark vereinfachtes und nur schematisches Beispiel aus dem RPC Paket:

- Die Client Goroutine loopt über eingehende Daten
- Diese können z.B. über ein Netzwerk kommen
- Um sich nicht um den Speicherbedarf des Buffers kümmern zu müssen wird der Buffered Channel freeList verwendet
- In freeList sind alle freien Buffer gespeichert
- Wenn kein freier Buffer in freeList mehr vorhanden ist, dann wird ein neuer Buffer erzeugt.

```
var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)

func client() {
    for {
        var b *Buffer
        // Grab a buffer if available; allocate if not.
        select {
        case b = <-freeList:
            // Got one; nothing more to do.
        default:
            // None free, so allocate a new one.
            b = new(Buffer)
        }
        load(b)              // Read next message from the net.
        serverChan <- b      // Send to server.
    }
}
```

- Der Server empfängt nun einen Buffer mit Daten über serverChan
- Er verarbeitet die Daten
- Wenn in freeList noch Platz ist, dann wird der Buffer in die freeList gesteckt, damit der Client diesen wieder verwenden kann
    - wenn nicht, dann wird der buffer einfach verworfen. Der GarbageCollector kümmert sich dann darum.

- der Default Wert in select wird immer dann verwendet, wenn kein case ausgeführt werden kann.
- Diese selects blockieren deshalb nicht

```
func server() {
    for {
        b := <-serverChan    // Wait for work.
        process(b)
        // Reuse buffer if there's room.
        select {
        case freeList <- b:
            // Buffer on free list; nothing more to do.
        default:
            // Free list full, just carry on.
        }
    }
}
```

Eine Implementierung des leaky buffers analog von diesem Beispiel:
https://github.com/ayanamist/ssr-go/blob/master/common/leakybuf.go
Video: 313

Fehler

- Fehler in Go sind zusätzliche Rückgabewerte von Funktionen
- Sie sind return Werte
- Ein Fehler ist in Go ein interface

```
type error interface {
    Error() string
}
```

- Durch die Implementierung des error Interfaces kann neben dem Fehler auch weiterer Kontext zurück geliefert werden.

```
// PathError records an error and the operation and
// file path that caused it.
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // The associated file.
    Err error    // Returned by the system call.
}

func (e *PathError) Error() string {
```

```
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

Die Ausgabe des Path Errors:

```
open /etc/passwx: no such file or directory
```

- So eine Fehlerausgabe ist um einiges hilfreicher als ein anonymes “there is no such file or directory
- Wenn irgendwie möglich sollte ein Fehler auch seine Herkunft melden.
- Die Caller können dann über Type Switches oder Type Assertions mehr über die Art des Fehlers herausfinden

```
for try := 0; try < 2; try++ {
    file, err = os.Create(filename)
    if err == nil {
        return
    }
    if e, ok := err.(*os.PathError); ok && e.Err == syscall.ENOSPC {
        deleteTempFiles()  // Recover some space.
        continue
    }
    return
}
```

Video: 314

Panic

- Wenn etwas schief läuft soll in der Regel soll ein error gemeldet werden
- jedoch was ist, wenn das Programm dadurch nicht weiter ausgeführt werden darf?
- hierfür kann panic() verwendet werden

```
// A toy implementation of cube root using Newton's method.
func CubeRoot(x float64) float64 {
    z := x/3   // Arbitrary initial value
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if veryClose(z, prevz) {
            return z
        }
    }
    // A million iterations has not converged; something is wrong.
    panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}
```

- In dem Beispiel ist panic() vielleicht in Ordnung.
- In Paketen sollte panic eigentlich nicht angewendet werden
- Einzige vernünftige Anwendung an der Stelle ist während der Initialisierung

```
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

Video: 315

Recover

- Programmabbrüche gibt es
    - bei panic()
    - index out of bounds bei slices
    - nicht funktionierende Type Assertions
- Dabei wird die Funktion gestoppt und die defer Funktionen ausgeführt
- Am Ende folgt der Programmabbruch
- Mit recover kann dieser jedoch gestoppt werden
    - Das panic Argument wird durch recover() zurück geliefert

```
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)
    }
}

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Println("work failed:", err)
        }
    }()
    do(work)
}
```

- Die Logk der Panic ist, dass die defer Funktion unabhängig der panic ausgeführt wird. Also noch bevor das Programm gestoppt wird.
- In der Defer Funktion kann also auch in ein Log geschrieben werden.

```
// Error is the type of a parse error; it satisfies the error interface.
```

```
type Error string
func (e Error) Error() string {
    return string(e)
}

// error is a method of *Regexp that reports parsing errors by
// panicking with an Error.
func (regexp *Regexp) error(err string) {
    panic(Error(err))
}

// Compile returns a parsed representation of the regular expression.
func Compile(str string) (regexp *Regexp, err error) {
    regexp = new(Regexp)
    // doParse will panic if there is a parse error.
    defer func() {
        if e := recover(); e != nil {
            regexp = nil    // Clear return value.
            err = e.(Error) // Will re-panic if not a parse error.
        }
    }()
    return regexp.doParse(str), nil
}
```

- Vernünftiges Beispiel von benannten Rückgabewerten
    - Wenn doParse mit panic abbricht, dann setzt defer den wert regexp auf nil
    - Der Fehler err wird gleich dem Fehler aus panic gesetzt

Diese Beispiel beinhaltet noch ein spannendes Pattern.

- regexp implementiert mit error() das Error interface
- wenn dieser Fehler immer bei parse Fehlern verwendet wird, dann istinnerhalb von defer und recover immer auch dieser Fehler verantwortlich
- So können die Arten von Fehlern getrennt werden
- Jedoch ein Paket sollte selber nie das Hauptprogramm abbrechen können!
    - Immer nur mit recover verwenden!

Video: 316

Ein Web Server

- Beispiel verwendet die Google Chart API
- über addr kann der Port für den Server geändert werden
- templ ist ein HTML Template, welches über die Konstante templateStr gesetzt wird
- über Handle wird der URL Baum registriert
    - Pattern: HandlerFunc Typumwandlung
- mit ListenAndServer wird der Server gestartet

    - Wichtig, da die Funktion einen Fehler zurück geben kann, muss dieser auch im Programm abgefangen werden!
    - Wird häufig vergessen
- QR führt nur das Template aus.
    - der FormValue zu “s” aus dem Request wird in das Template übergeben
    - die Execute Funktion schreibt das ergebnis in den ResponseWriter

```
package main

import (
    "flag"
    "html/template"
    "log"
    "net/http"
)

var addr = flag.String("addr", ":1718", "http service address") // Q=17,
R=18

var templ = template.Must(template.New("qr").Parse(templateStr))

func main() {
    flag.Parse()
    http.Handle("/", http.HandlerFunc(QR))
    err := http.ListenAndServe(*addr, nil)
    if err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}

func QR(w http.ResponseWriter, req *http.Request) {
    templ.Execute(w, req.FormValue("s"))
}

const templateStr = `
<html>
<head>
<title>QR Link Generator</title>
</head>
<body>
{{if .}}
<img
src="http://chart.apis.google.com/chart?chs=300x300&cht=qr&choe=UTF-8&chl={{
.}}" />
<br>
{{.}}
<br>
```

```
<br>
{{end}}
<form action="/" name=f method="GET"><input maxLength=1024 size=70
name=s value="" title="Text to QR Encode"><input type=submit
value="Show QR" name=qr>
</form>
</body>
</html>
`
```

Abschied

- Danke für Deine Aufmerksamkeit
- Anregungen bitte gerne an mich
  Twitter: @lyckade
- Zusätzliche Aktivitäten
  - Erstellen von Übungen bzw. Beispielen
    https://github.com/gokurs/effective_go
  - Übersetzung:
    Pull Requests sind willkommen
    https://github.com/gokurs/effective_go/blob/master/uebersetzung.md
- Anmeldung GopherSlack: https://invite.slack.golangbridge.org/