#16 TECHARTIKEL

EINFÜHRUNG IN GOLANG

17.02.2025

Marcel Rieger

Inhalt

Einführung in Golang

Heutzutage gibt es eine riesige Auswahl an Programmiersprachen für die Softwareentwicklung. Zu den bekanntesten zählen C++, C#, Java, JavaScript, PHP, Python, R, Swift und Go. Jede dieser Sprachen wurde für spezifische Anwendungsgebiete entwickelt, sei es für Webanwendungen, mobile Apps oder Desktopprogramme. Aber warum sollte man sich gerade für Go entscheiden? Dieser Artikel bietet einen hilfreichen Anhaltspunkt, um diese Entscheidung zu erleichtern.

1. Die Geburt der Programmiersprache Go/Golang

Was ist Go und wie ist diese Programmiersprache eigentlich entstanden? Um das zu verstehen, müsste man sich in die Gedanken von drei Männern im Jahr 2007 versetzen, die bei Google beschlossen, eine neue Idee in die Tat umzusetzen. Sie machten sich Gedanken, welche Themen in den zukünftigen Jahren dominieren würden. Dabei waren in ihren Augen unterschiedliche Auswahlkriterien, wie eine effiziente Kompilierung, effiziente Ausführung (Parallelität) und eine sichere automatische Speicherverwaltung erforderlich. Innerhalb weniger Tage setzten sie sich Ziele und skizzierten, wie ihre zukünftige Programmiersprache aussehen sollte.

Unbezahlt und neben ihrer regulären Arbeit begannen sie mit der Entwicklung von Go. Bis Januar 2008 hatte Ken einen Compiler entwickelt, der ihm C-Code als Ausgabe generierte. Im Mai 2008 begann Ian Taylor unabhängig von den anderen, an einem GCC-Frontend für Go auf Basis der Entwurfsspezifikation zu arbeiten. Schließlich half Russ Cox ab Ende 2008 dabei, die Sprache und Bibliotheken von einem Prototypen in die Realität umzusetzen. Go wurde geboren.

Daraus folgte, dass Go am 10. November 2009 zu einem öffentlichen Open-Source-Projekt wurde. Dadurch konnte die Community zahlreiche Ideen, Diskussionen und Code beitragen. Bis heute ist daraus eine weltweite Gemeinschaft von Millionen Go-Programmierern entstanden – **die Gophers**. [1]

Gopher - das Maskottchen

Kann eine Programmiersprache heute ohne ein Maskottchen auskommen? Natürlich nicht! Daher wurde auch das einzigartige Maskottchen erfunden, der Gopher. Übrigens wurde die erste Version des Maskottchens von Renee French schon lange vor Go's Geburt designet. Das Maskottchen wurde später für das Go-Projekt angepasst. Mehr über die Entwicklung des Go-Gophers kannst du hier [2] nachlesen.

Ziele von Go

Was waren die Designziele mit Go? Dazu müssen einige Kriterien von Go betrachtet werden: Einfachheit, Skalierbarkeit, Unterstützung für moderne Hardware, Effizienz beim Entwickeln, Typsicherheit, Laufzeit und eine verbesserte Paketstruktur. Diese waren die Hauptkriterien, die Go in seiner Sprache verbinden und im Gegensatz zu anderen Sprachen verbessern wollte. [3]

2. Schlüsselfunktionen und Vorteile

Einfachheit und Lesbarkeit

Go ist eine Sprache mit einer sehr einfachen Syntax. Die Entwickler von Go - Rob Pike, Robert Griesmer und Ken Thompson - hatten eine Abneigung gegen C++. Daher wollten sie eine Sprache entwickeln, die C sehr nahe kommt, aber mit moderneren Funktionen und besserer Sicherheit ausgestattet ist. Daher haben sie versucht die Syntax so einfach und übersichtlich zu halten wie nur möglich. Und das merkt man auch, wenn man mit Go programmiert.

Die Syntax leicht zu verstehen und beruht auf sehr rudimentären Funktionen. Zusätzlich zu der einfachen Syntax bringt Go auch einige Funktionen mit, die das Leben des Entwicklers erleichtern sollen, wie zum Beispiel den "Garbage Collector" und die Typinferenz. Darüber hinaus wurden in der Version Go 1.18 auch "Generics" in die Sprache eingeführt. [4]

Standardbibliothek

Wenn man bereits Zeit im Golang-Subreddit verbracht hat, weiß man, dass Go-Entwickler oft empfehlen, die Funktionen der Standardbibliothek zu verwenden - und das aus gutem Grund! Sie enthält sehr robuste Funktionen, darunter Pakete zum Codieren und Decodieren von JSON, CSV, Binär- und anderen Datenformaten, zur Durchführung von HTML-Templatefunktionen, zum Starten eines voll funktionsfähigen HTTP-Servers und Routers, für zeitbasierte Operationen und Manipulationen, reguläre Ausdrücke, kryptografische Funktionen wie Hashing, Bildkodierung und -decodierung für PNGs, JPEGs und GIFs, Kompressions- und Dekompressionsfunktionen für Gzip und andere Formate sowie viele weitere Pakete. Aufgrund dessen können wahrscheinlich die meisten Anwendungsfälle allein durch die Verwendung der Standardbibliothek abgedeckt werden. Zusätzlich können mit Go Mod alle Packages von GitHub heruntergeladen oder eigene Bibliotheken erstellt und hochgeladen werden. [4]

3. Grundlegende Syntax und Konzepte

Die Syntax von Go wurde so konzipiert, dass sie einfach zu lesen ist. Dazu kommen nun ein paar Beispiele.

Packages und Imports

Go-Programme sind so aufgebaut, dass sie in "Packages" unterteilt werden können. Der Grundgedanke eines Packages ist eine Ansammlung von mehreren Go-Dateien. Diese Dateien erhalten dann alle in der ersten Zeile den Zusatz „package“.

```
package main

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

Hier ist zu erkennen, dass die Datei dem Package "main" zugeordnet ist. Typischerweise werden nach der Deklaration des Packages alle weiteren benötigten Imports für das Go-Programm hinzugefügt. Es ist wichtig zu wissen, dass die Imports nur in der aktuellen Datei geladen werden und nur hier genutzt werden können. [5]

Datentypen und Variablen

Variablen können in Go auf unterschiedliche Weise deklariert werden. Es ist möglich, das Schlüsselwort **"var"** vor den Variablennamen zu schreiben, gefolgt vom **Typ** und **optional einem initialisierten Wert**. Falls die Variable direkt initialisiert wird, ist die Angabe des Typs nicht zwingend notwendig. In Go kann auch eine **Kurzschreibweise** gewählt werden, bei der ":=" direkt sowohl den Typ als auch den initialen Wert in die Variable schreibt. Konstanten hingegen verwenden das Schlüsselwort **"const"** und müssen somit zur Compile-Zeit bekannt sein. Sie können während der Programmausführung jedoch nicht geändert werden.

```
var i int = 20
var i = 20 // Die Variable i wurde direkt als Integer initialisiert.

// Kurzschreibweise
```

```
i := 20

// Konstanten Deklaration
const defaultPort = 8080
const (
   defaultPort = 8080
   localPort   = 8181
   service = "user"
)
```

Verwendung von Strings

Ein String in Golang ist ein **lesbares Slice aus Bytes**, also eine Gruppierung von mehreren Bytes. Ein Beispiel wäre vorname := "Test Name". Zusätzlich kann ein String auch spezielle Zeichenkombinationen wie zum Beispiel \n oder \t enthalten. Dadurch sind auch folgende Befehle möglich: adresse := "Deutsches Museum \t Museumsinsel". Die Ausgabe mit der Println()-Funktion wäre dann wie folgt:

Deutsches Museum
Museumsinsel

Strukturen (Structs)

Unter einer **Struktur (struct)** in Go versteht man **zusammengefasste Variablen** unter einem **zusammengesetzten Datentyp**. Diese Variablen, auch Felder genannt, können unterschiedlichste Typen aufweisen und ähneln den klassischen Klassen aus anderen objektorientierten Sprachen. Im Gegensatz zu Klassen besitzen Structs **jedoch keine Vererbung**. Im Vergleich zu den normalen Basistypen wie Zeichenfolgen (Strings) oder Ganzzahlen (Ints) können Structs **komplexe Entitäten** darstellen, wie beispielsweise eine Person mit einem Namen, Alter und Adresse, eine Bestellung mit Namen, Kosten und Vorrat oder einen Punkt im 2D-Raum mit X- und Y-Koordinaten. [6]

In Go werden Strukturen durch das Schlüsselwort **"type"** definiert. Um zum Beispiel eine Person zu modellieren, kann man folgende Struktur erstellen und auf unterschiedliche Weise initialisieren.

```
type Person struct {
   Name    string
   Alter   int
```

```
    Adresse string
}

func (p Person) Greet() string {
    return "Hallo, mein Name ist " + p.Name
}

func main() {

    var p0 Person    // Zero-value Initializierung
    fmt.Println(p0)  // Output: { 0 }

    p1 := Person{"Alice", 30, "Mustermannstr. 1"} // Initialisierung per Reihenfolge
    p2 := Person{
        Name:  "Alice",
        Alter:   30,
        Adresse: "Mustermannstr. 1",
    } // Initialisierung durch Variablen

    p3 := new(Person) // Strichwort 'new' und  Speicher wird zugeteilt und mit Nullw
erten befuellt
    fmt.Println(p3) // Output: &{ 0 }
    p3.Name = "Alice"
    p3.Alter = 30
    p3.Adresse = "Mustermannstr. 1"
    fmt.Println(p3) // Output: &{Alice 30 Mustermannstr. 1}
    fmt.Ptintln(p3.Greet()) // Output: Hallo, mein Name ist Alice
}
```

Zeiger (Pointer)

Zeiger in Go sind eine der grundlegendsten Funktionen, da Entwickler durch Zeiger direkte Speicheroperationen durchführen können. Dadurch können Datenstrukturen dynamisch angelegt und die Effizienz des Programms enorm verbessert werden. Es macht einen großen Unterschied, ob man nur **eine Speicheradresse an Funktionen übergibt oder das komplette Objekt mit allen initialisierten Werten**. In Go werden Zeiger mit dem Symbol ***** dargestellt, und mithilfe des &-Operators kann auf die Speicheradresse zugegriffen werden. Im Folgenden finden Sie ein kleines Beispiel zur Erstellung eines Zeigers.

```
var x int = 10
var ptr *int = &x
```

```
fmt.Println(x)   // Ausgabe: 10
fmt.Println(ptr) // Ausgabe: Speicheradresse von x
fmt.Println(*ptr) // Ausgabe: 10 (Wert von x)
```

Standardmäßig wird in Go ein Parameter **'by value'**, also durch eine Kopie übergeben. Mit Zeigern kann man dies jedoch als **'by reference'** übergeben und das **Objekt in der Funktion direkt manipulieren**. Es können also direkt die Originalwerte in der Funktion überschrieben werden, wie im Folgenden zu erkennen:

```
func plus(x *int) {
  *x++
}

func main() {
  x := 10
  plus(&x)
  fmt.Println(x) // Ausgabe: 11
}
```

Wann sollte man aber Zeiger am besten Nutzen? Zeiger sind vor allem gut, um Arrays oder größere Datenstrukturen effizient zwischen Funktionen zu übergeben bzw. zu transportieren. Außerdem eignen sie sich, um nicht-initialisierte-Objekte oder freien Speicher zu erkennen. Denn Zeiger werden standardmäßig mit einem **nil Wert** initialisiert. Und man kann Zeiger dazu nutzen, um den Speicherverbrauch zu reduzieren, besonders bei großen Datenstrukturen. [7]

4. Array vs Slice

Neulinge in der Golang-Programmierung sind oft verwirrt, da **zwei unterschiedliche Arten von Arrays** in Go existieren. Beide speichern Objekte in einer bestimmten Reihenfolge, erfüllen aber einen unterschiedlichen Zweck. [8]

Arrays

in Array in Go ist eine nummerierte Sequenz von verschiedenen Elementen oder Objekten. Die Elemente in einem Array müssen denselben Typ aufweisen, und **nachdem ein Array erstellt wurde, kann seine Länge nicht mehr verändert** werden. Die Vorlage für viele Strukturen war damals die Programmiersprache C. Um ein Array zu deklarieren, wird die folgende Syntax verwendet:

var arrayname [arraylänge] datentyp.

Im folgenden Beispiel werden **6 Elemente vom Typ int** in das Array initialisiert.

```
package main
import "fmt"

func main() {
    var numbers [6]int
    fmt.Println(numbers)  // [0 0 0 0 0 0]
}
```

Im vorherigen Beispiel werden sechs Integer in einem Array gespeichert. Es ist wichtig anzumerken, dass in **Golang Arrays als eigenständige Werte definiert** sind. Wenn ein Array kopiert oder weitergegeben wird, wird immer eine Kopie der Daten anstelle des Arrays weitergeleitet. [8]

Slices

Slices hingegen sind **dynamisch anpassbare Arrays**. Ihre Länge kann flexibel verändert werden. Einfach gesagt, ist ein Slice eine Referenz auf ein zugrundeliegendes Array. In Go werden Slices häufiger genutzt, einfach weil sie vielseitiger eingesetzt werden können.

Hier ist ein Beispiel, um ein Slice zu initialisieren.

```
package main
import "fmt"

func main() {
```

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

Bei Slices muss die Länge nicht bei der Initialisierung angegeben werden. Der Go-Compiler kann aus der Anzahl der Objekte auf die Länge des Slices schließen. Es gibt jedoch noch weitere Unterschiede zu Arrays. Wenn **ein Slice in eine Funktion übergeben** wird, wird **eine Referenz des original Slices übergeben**, nicht eine Kopie, wie bei Arrays. Slices haben eine Kapazität und eine Länge. Die Kapazität ist die Anzahl der Elemente, die das zugrundeliegende Array enthalten kann (startend beim ersten Element des Slices). Die Länge ist die Anzahl der Elemente, die das Slice derzeit besitzt. Sie sind dynamisch in der Länge anpassbar. Dafür kann die in Golang direkt eingebundene 'append' Funktion genutzt werden, wodurch das Slice automatisch wachsen kann. [8]

```
package main

func (s *service) GetVersions(features ...string) ([]*Version, error) {
   result := make([]*Version, 0)
   for _, feature := range features {
      versions, err := findVersionsByFeatures(feature) // Beispiel Funktion
      if err != nil {
         return err
      }

      result = append(result, versions...)
   }
   return result, nil
}
```

Hier ist ein Beispiel, wie die **'append' Funktion** genutzt werden kann. In der obigen Funktion werden durch einen variadischen Parameter eine beliebige Anzahl an Strings an die **GetVersions Funktion** übergeben. Am Anfang der Funktion wird ein **'result' als Slice** mit der **Länge 0** initialisiert. Dann werden die übergebenen Parameter in einer Schleife durchlaufen, und die Ergebnisse automatisch an das Slice hinzugefügt. Dabei wird die Länge automatisch angepasst und muss nicht von Anfang an angegeben werden. Schlussendlich wird das Ergebnis mit den zusammengefassten Versionen zurückgegeben. [8]

Wann sollte ein Array genutzt werden?

Ein paar Richtlinien, wann es Sinn macht, ein Array zu nutzen:

- Wenn zur Laufzeit die Anzahl der Elemente bereits bekannt ist und sich nicht ändern wird.
- Wenn man die Speicherplatzverwaltung verbessern möchte, da Arrays eine höhere Speicherplatzeffizienz als Slices haben.
- Oder wenn man sich keine Sorgen um Änderungen am ursprünglichen Array machen möchte, da immer eine Kopie weitergereicht wird.

Ein gutes Beispiel für die Nutzung eines Arrays sind feste Daten wie z. B. die Wochentage.

```
var daysOfWeek [7]string = [7]string{"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"}
```

Wann werden Slices genutzt?

- Wenn das Array dynamisch zur Laufzeit angepasst werden muss.
- Wenn mit Objekten gearbeitet werden muss, die zur Laufzeit noch nicht feststehen oder noch nicht absehbar ist, wie viele Objekte benötigt oder gespeichert werden müssen.
- Oder wenn man gerne die eingebauten Funktionen wie 'append', 'copy' oder 'len' aus der Standardbibliothek benutzen möchte.

Ein Slice kann mit der Standardfunktion **'make' initialisiert** werden. Ein Beispiel dafür ist das Lesen von Datenpaketen, beidem die Größe im Vorhinein nicht bekannt ist. [8]

```
func make([]T, len, cap) []T

buffer := make([]byte, 0, 2048) // Bytes haben eine Kapazitaet von 2048, Laenge wird aber mit 0 initialisiert
```

Variadische Parameter

In Golang gibt es eine besondere Signatur, die es uns erlaubt, mehrere Objekte desselben Typs an eine Methode zu übergeben. Das nachfolgende Beispiel zeigt zweimal dieselbe Funktion: einmal mit einem String-Array und einmal mit variadischen Parametern. Die drei Punkte **...** übergeben der Funktion den

variadischen Parameter. Dieser Parameter akzeptiert nun eine **flexible Anzahl von String-Werten und referenziert sie in einem Slice**. [9]

```
package main
import "fmt"

func addUsers(users []string){
  for _,user := range users {
    fmt.Println(user)
  }
}

func main() {
  addUsers([]string{"User A", "User B"})
}

##################################################

func addUsers(users ...string){
  for _,user := range users {
    fmt.Println(user)
  }
}

func main() {
  addUsers("User A", "User B")
}
```

Aber warum sollte man diese variadischen Parameter nutzen? Der große Vorteil besteht darin, dass man **Objekte immer an ein Slice hinzufügen** kann, **ohne** vorher eine **Liste explizit definieren oder initialisieren** zu müssen. Viele Methoden der Standardbibliothek funktionieren auf diese Weise, zum Beispiel auch die 'fmt.Print' Funktion. Im nachfolgenden Beispiel ist es egal, wie viele String-Parameter in die 'addUser' übergeben werden. Zusätzlich kann man auch direkt durch die drei Punkte am Ende der Variable alle Parameter direkt in die nächste Funktion "append" übergeben. Somit benötiget man keine For-Schleife mehr und die User können direkt in der globalen Variable gespeichert werden. [10]

```
package main
import "fmt"
```

```go
var users = []string{}
func addUser(user ...string){
    users = append(users, user...)
}

func main() {
    addUser("User A", "User B", "User C")
    fmt.Println(users)
}
```

5. Concurrency in Go

Parallelität kann ein herausfordernder Aspekt des Programmierens sein. Einige beliebte Sprachen entscheiden sich dafür, herkömmliche Nebenläufigkeit zu umgehen und stattdessen asynchrones I/O und eine Ereignisschleife zu nutzen. Für Sprachen, die Nebenläufigkeit unterstützen, wird das Schreiben von sicherem Code aufgrund verschiedener Hürden wie Deadlocks, gemeinsamen Zustand und Wettlaufsituationen schwierig. Go hat jedoch das Nebenläufigkeitsmodell erfolgreich vereinfacht, indem es **Go-Routinen und Kanäle (Channels)** nutzt.

Go-Routinen, die Threads ähneln, machen es bemerkenswert einfach, Nebenläufigkeit zu erzeugen und zu verwalten. Das Schreiben von nebenläufigem Code in Go beinhaltet lediglich die Verwendung des **"go"**-Schlüsselworts vor einer Funktion.

Kanäle (Channels) hingegen bieten eine elegante Lösung für die **Inter-Thread-Kommunikation**, ermöglichen einen nahtlosen Datenaustausch zwischen Aufgaben oder erlauben Threads, zu blockieren, bis Daten verfügbar sind. Trotz der Einfachheit und Effektivität von Go-Routinen und Kanälen ist es wichtig zu beachten, dass Kanäle ihre eigenen Fallstricke haben. [4]

Go-Routinen und Channels

Go-Routinen bezeichnen im Wesentlichen Threads, die man aus Programmiersprachen wie Java schon kennt, aber dafür **kostengünstiger in Bezug auf Speicherplatz und Startzeit** sind. Die Kosten der Erstellung einer Go-Routine sind sehr niedrig im Vergleich zu Threads. Gerade einmal wenige Kilobyte beträgt die Größe und kann sich je nach Anforderungen auch dynamisch vergrößern oder verkleinern.

Wie werden Go-Routinen gestartet?

Man schreibt vor den Funktionsaufruf das Schlüsselwort "go". Dadurch werden die Prozesse automatisch gestartet. Das bedeutet, dass nicht sequenziell gewartet wird, bis ein Prozess abgeschlossen ist, bevor der nächste Prozess gestartet wird, sondern das Programm springt direkt zum nächsten Aufruf. [11]

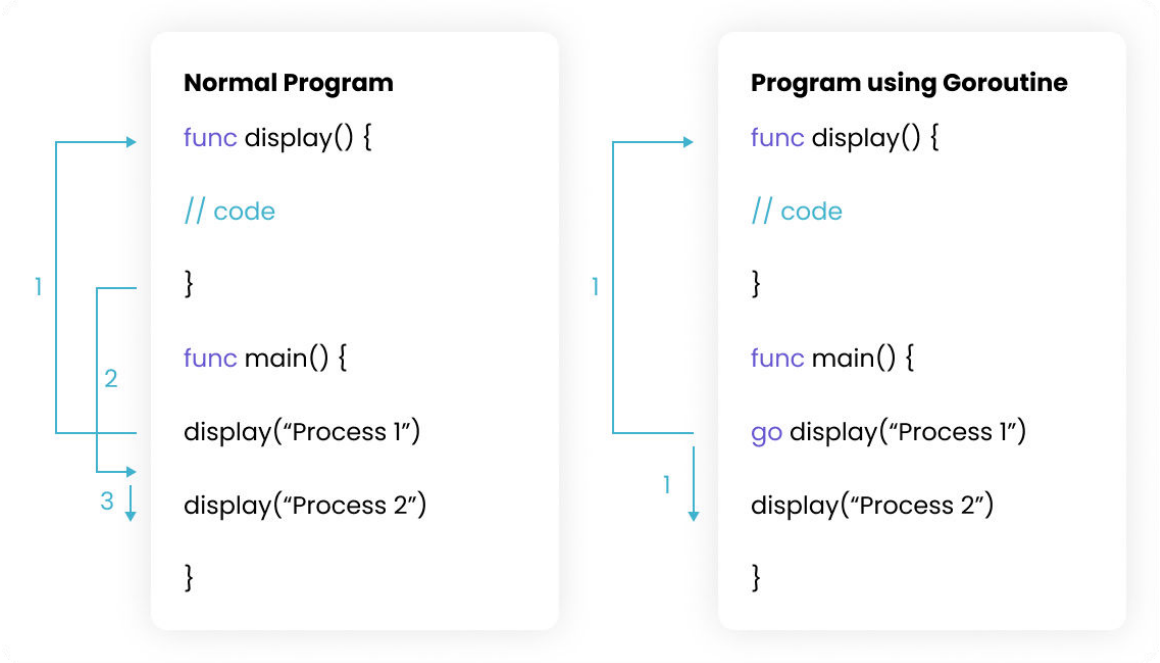

Abbildung 1: Normal vs Go-Routine

Der Start einer neuen Go-Routine:

```
package main

import (
   "fmt"
)

func hello() {
   fmt.Println("Hello world from goroutine")
}
func main() {
   go hello()
   fmt.Println("Hello from the main function")
}
```

Wenn dieses Programm gestartet wird, fällt schnell auf, dass hier etwas Unerwartetes passiert. Es wird in der Konsole nur der Satz "Hello from the main function" ausgegeben. Aber warum? Dazu muss die Go-Routinen genauer betrachtet werden. Wenn in einem Programm eine Go-Routine gestartet wird, erhält die aufrufende Funktion direkt einen Rückgabewert. Somit wartet die Main-Funktion nicht darauf, dass die Go-Routine fertig ausgeführt wurde, und beendet

das Programm, bevor das Ausgabestatement in der Go-Routine ausgeführt werden konnte.

Aus diesem Grund muss das Hauptprogramm mindestens so lange aktiv sein, solange noch Go-Routinen existieren. Andernfalls werden, wie im ersten Beispiel, die Ausgabefunktion nicht ausgeführt, da das Hauptprogramm vorzeitig beendet wurde. Channels können nun dafür sorgen, dass auf die Ausgabe gewartet wird. Im Folgenden wird skizziert, wie der Programmablauf mit Channels funktionieren würde. [12]

```
package main

import (
   "fmt"
)

func hello() {
   fmt.Println("Hello world from goroutine")
        done <- true
}

func main() {

        done := make(chan bool)
   go hello(done)

   <- done
   fmt.Println("Hello from the main function")
}
```

Durch das Erstellen eines „done“-Channels konnte verhindert werden, dass das Programm direkt die Print-Funktion aus der Main-Funktion ausführt. Es wird darauf gewartet, dass ein Wert in den Channel geschrieben wird. Somit kann gewährleistet werden, dass man auf das Beenden der Go-Routine wartet. [16]

6. Golang? Fast zu gut, um wahr zu sein.

Nun wird es Zeit die rosarote Brille abzunehmen und auch die Schwächen der Sprache unter die Lupe zu nehmen.

Einfachheit als Schwäche?

Zwar wurde Go mit dem Ziel der Einfachheit entwickelt, jedoch kann diese Einfachheit für den Entwickler schnell zum Problem werden. Das minimalistische Design von Go hat dafür gesorgt, dass viele **essenzielle Features** aus anderen Sprachen (beispielsweise der Nullish Coalescing-Operator (??) aus C# [14] **nicht in der Standardbibliothek** vorhanden sind.

Einer der größten Kritikpunkte von Go vor der Version 1.18 war, dass keine Generics eingebaut waren. Auf Wunsch der Gophers wurde zwar eine **rudimentäre Unterstützung von Generics eingeführt**, diese ist aber nicht so weit entwickelt, wie es Entwickler aus anderen Programmiersprachen gewohnt sind. Aufgrund fehlender Funktionen in der Standardbibliothek kann es häufig dazu kommen, dass repetitiver Boilerplate-Code für einfache Funktionen geschrieben werden muss.

Angenommen, Sie möchten eine komplexe Datenstruktur aufbauen. Als Entwickler stellt man schnell fest, dass viel repetitiver Code geschrieben werden muss, was in anderen Sprachen eventuell schneller und eleganter gelöst werden kann. Natürlich ist Go-Code einfach zu verstehen. Aber es fühlt sich manchmal so an, als würde man mit Legosteinen versuchen, ein Haus zu bauen. Es ist einfach, jedoch langsam und ineffizient. Zudem müssen für fortgeschrittene Funktionen oft externe Pakete eingebunden werden – z.B. für Datenbankzugriffe oder komplexe HTTP-Clients. [13]

Umständliche Fehlerbehandlung

In Go besteht die Fehlerbehandlung darin, die Fehler als Rückgabewert aus einer Funktion zurückzugeben. Somit müssen explizite Fehlerüberprüfungen nach jedem Funktionsaufruf durchgeführt werden. Auf den ersten Blick sieht es einfach und gut lesbar aus, jedoch kann es schnell zu überfülltem Code und zu hoher Komplexität mit vielen 'if'-Anweisungen führen. [15]

```go
package main

import (
    "errors"
    "fmt"
)

var ErrDivideByZero = errors.New("divide by zero")

func Divide(a, b int) (int, error) {
    if b == 0 {
        return 0, ErrDivideByZero
    }
    return a / b, nil
}

func main() {
    a, b := 10, 0
    resultDivide, err := Divide(a, b)
    if err != nil {
        switch {
        case errors.Is(err, ErrDivideByZero):
            fmt.Println("divide by zero error")
        default:
            fmt.Printf("unexpected division error: %s\n", err)
        }
        return
    }

    fmt.Printf("%d / %d = %d\n", a, b, result)
}
```

Verwaltung von Go-Routinen

Wie bereits im vorherigen Kapitel beschrieben, sind Go-Routinen schnelle und einfache Threads und sehr vielseitig einsetzbar. Das Verwalten von Go-Routinen, insbesondere in großen Systemen, kann sehr schnell zu einer komplexen und anspruchsvollen Aufgabe werden. Denn in parallelen Prozessen, Fehler zu finden, ist genauso schwierig wie in anderen Sprachen. Race-Conditions und Deadlocks können eine Folge falscher Anwendung sein.

Stellen wir uns ein hypothetisches Szenario vor, in dem Sie eine Gruppe junger und unerfahrener Go-Entwickler leiten. Diese jungen Entwickler haben gerade Go-

Routinen für sich entdeckt und fangen an, diese überall in das System einzubauen. Man wird schnell feststellen, dass die Codebase mit Channels und Go-Routinen überfüllt ist und es zu ungewollten Race-Conditions und Deadlocks kommt. Die daraus resultierenden Bugs zu beheben, fühlt sich fast so an, als müsste man an Weihnachten noch die verhedderte Weihnachtsbeleuchtung in der Dunkelheit entwirren. Nicht zu empfehlen. [13]

7. Fazit - Für wen ist Go geeignet?

Go ist eine gute Sprache, um **robuste und skalierbare Backend-Server** zu erstellen. Gerade für die **Verarbeitung von vielen Nutzeranfragen in verteilten Systemen** kann Go glänzen. Ebenso eignet es sich gut für **Netzwerk-Server oder Cloud Computing**. Und **DevOps**-Experten können durch die enorme Kompilierungsgeschwindigkeit schnell automatisierte Skripte erstellen.

In einigen Szenarien würde ich persönlich jedoch nicht empfehlen, Go zu benutzen. Wenn ein Projekt umfangreiche Bibliotheken oder Frameworks benötigt, ist Go möglicherweise nicht die richtige Wahl. Es bietet in diesem Sinne kein reifes Ökosystem, und die Entwickler müssen viel Logik selbst entwickeln. Ebenso hat Go kaum Unterstützung im Bereich des maschinellen Lernens. Gerade im Vergleich zu Python oder R kann Go kaum Funktionalität bieten. Leider werden auch keine grafischen Benutzeroberflächen (GUIs) wie in Java, C# oder Python unterstützt. Speziell für die Low-Level-Programmierung gibt es zudem bessere Alternativen wie C oder Rust. Natürlich kann man mit Tiny-Go beispielsweise auch hardwarenah programmieren, aber die Möglichkeiten sind stark eingeschränkt.

8. Quellen

[1] What is the history of the project? url: https://go.dev/doc/faq#history (besucht am 29. 08. 2024).

[2] The Go Gopher. url: https://go.dev/blog/gopher (besucht am 09. 10. 2024).

[3] Go — How It All Began. url: https://medium.com/geekculture/learn-go-part-1-the-beginning-723746f2e8b0 (besucht am 09. 10. 2024).

[4] Should you learn Go in 2023? url: https://youtu.be/U2PpMZ7hWpg?si=RBUDsHhPbJbGiGus (besucht am 22. 10. 2024).

[5] The Basics of Go. url: https://appmaster.io/blog/the-basics-of-go-programming-language (besucht am 23. 10. 2024).

[6] Golang Struct Mastery: Deep Dive for Experienced Developers. url: https://reliasoftware.com/blog/golang-struct (besucht am 23. 01. 2025).

[7] A Comprehensive Guide to Pointers in Go. url: https://medium.com/@jamal.kaksouri/a-comprehensive-guide-to-pointers-in-go-4acc58eb1f4d (besucht am 30. 01. 2025).

[8] Understanding Slices and Arrays in Go. url: https://www.codingexplorations.com/blog/interview-series-understanding-slices-and-arrays-in-go (besucht am 17. 01. 2025).

[9] Variadische Funktion. url: https://de.wikipedia.org/wiki/Variadische_Funktion (besucht am 17. 10. 2024).

[10] Golang Slice Tricks Every Beginner Should Know. url: https://www.youtube.com/watch?v=AL_C9nF_0ss (besucht am 17. 10. 2024).

[11] Go-Routinen. url: https://appmaster.io/de/blog/goroutines-de (besucht am 04. 11. 2024).

[12] Goroutines. url: https://golangbot.com/goroutines/ (besucht am 06. 11. 2024).

[13] Why Go is the worst language you could ever learn. url: https://medium.com/@roma.gordeev/why-go-is-the-worst-language-you-could-ever-learn-4a7be0d37509 (besucht am 17. 01. 2025).

[14] Nullish coalescing-Operator. url: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing (besucht am 17. 01. 2025).

[15] Effective Error Handling in Golang. url: https://earthly.dev/blog/golang-errors/ (besucht am 17. 01. 2025).

[16] Channels, url: https://golangbot.com/channels/ (besucht am 31.01.2025)

[17] Programmieren lernen für Dummies, Buch von Daniel Lorig. url: https://www.google.de/books/edition/Go_programmieren_f%C3%BCr_Dummies/hFlZEAAAQBAJ?hl=de&gbpv=1&printsec=frontcover (besucht am 31.01.2025)