Go

Verteilte Softwaresysteme

Prof. Dr. Oliver Braun

Letzte Änderung: 27.04.2020 17:48

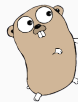
Aus den FAQ von golang.org (1/2)

Go was born out of frustration with existing languages and environments for systems programming. Programming had become too difficult and the choice of languages was partly to blame. One had to choose either ***efficient compilation****,* ***efficient execution****, or* ***ease of programming****; all three were not available in the same mainstream language. Programmers who could were choosing ease over safety and efficiency by moving to dynamically typed languages such as Python and JavaScript rather than C++ or, to a lesser extent, Java.*

Aus den FAQ von golang.org (2/2)

Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. It also aims to be modern, with support for networked and multicore computing. Finally, working with Go is intended to be fast: it should take at most a few seconds to build a large executable on a single computer. To meet these goals required addressing a number of linguistic issues: an expressive but lightweight type system; concurrency and garbage collection; rigid dependency specification; and so on. These cannot be addressed well by libraries or tools; a new language was called for.

Wer begann damit?



Start 2007 durch

- Robert Griesemer
  - Mitarbeit an Java HotSpot Compiler und V8 JavaScript Engine
- Ken Thompson
  - erstes Unix, Programmiersprache B, Plan 9 (verteiltes OS), UTF-8
- Rob Pike
  - Plan 9, UTF-8

Ab 2008 mit

- Ian Lance Taylor
  - UUCP (Unix to Unix Copy Protocol), Arbeit an GCC, GNU Binutils, ...
- Russ Cox
  - Plan 9, Chord lookup service and the cooperative file system (CFS)

Warum Go entstand?

- Rob Pike, 30.10.2009:
  Go fast! Make programming fun again.
- viele Änderungen in den letzten Jahren (vor 2007)
  - ausufernde Library- und Dependency-Ketten
  - Dominanz des Networking
  - Fokus auf Client/Server
  - riesige Cluster
  - Multicore-Systeme
- die verbreiteten Sprachen für Systemprogrammierung wurden nicht mit Augenmerk auf diese Faktoren designed

Konstruktionsgeschwindigkeit

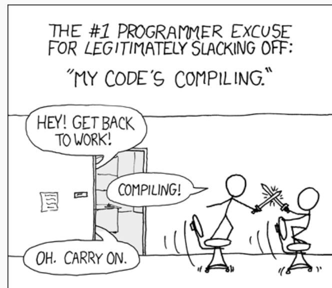
- es dauert zu lange um Software zu bauen
- die Tools sind langsam und werden immer langsamer (mächtiger!)
- Abhängigkeiten sind nicht unter Kontrolle
- Rechner werden nicht mehr schneller
- Software wächst und wächst

Typsystem-Tyrannei

- Robert Griesemer:
  Clumsy type systems drive people to dynamically typed languages.
- Ungeschickte Typsysteme
  - gute Ideen aber schlechte Umsetzung
  - Beispiel: `const` in C++
- Vererbungs-Hierarchien sind zu strikt
  - Typen in großen Programmen fallen nicht einfach in Hierarchien
  - Programmierer verschwenden viel zu viel Zeit über Vererbungshierarchien nachzudenken und sie zu um zu arrangieren
- **entweder** produktiv **oder** sicher (safe)

Ziele von Go

- Effizienz einer statisch-typisierten compilierten Sprache mit der einfachen Programmierung einer dynamischen Sprache verbinden
- Typ-Sicherheit und Speicher-Sicherheit
- gute Unterstützung für Nebenläufigkeit und Kommunikation
- effiziente und latenzfreie Garbage Collection
- high-speed Compilation



Quelle: https://xkcd.com/303/

Design Prinzipien

- orthogonale Konzepte (nicht überlappend)
- reguläre und einfache Grammatik
    - wenige Schlüsselwörter
    - ohne Symboltabelle zu parsen
    - Spezifikation https://golang.org/ref/spec
- reduzierte Notwendigkeit Typen angeben zu müssen
    - aber trotzdem Typsicherheit
- keine Typhierarchien!
    - OOP trotzdem möglich, aber ohne Vererbungshierarchien

Grober Überblick

- Grundsätze
  - saubere, knappe Syntax
  - leichtgewichtiges Typsystem
  - keine impliziten Umwandlungen (*keep things explicit*)
  - nicht typisierte Konstanten ohne feste Größe
  - strikte Teilung von Interface und Implementierung
- Runtime
  - Garbage Collection
  - Strings, Maps, Channels
  - Nebenläufigkeit
- Package model
  - explizite Abhängigkeiten um schneller Builds zu ermöglichen
  - seit Go 1.11 Go Modules

Wer nutzt Go?

- Google — die Go-Entwickler sind Google-Angestellte
    - Go at Google
- Docker etc.
    - Moby (Docker)
    - Kubernetes (automatisiertes Deployment, …)
- Heroku
- …

Ein bisschen was aus den FAQ (1/2)

Quelle: https://golang.org/doc/faq

- keine Generics
  - werden vielleicht noch kommen
  - Komplexität im Typsystem (zu) hoch für geringen Nutzen
- keine Exceptions
  - `try-catch-finally` führt zu “Spaghetti-Code”
  - Go-Funktionen können mehrere Werte zurück geben

Ein bisschen was aus den FAQ (2/2)

- keine Assertions
  - Go-Entwickler haben die Erfahrung gemacht, dass Assertions oft als Krücke verwendet werden um sich keine Gedanken über Fehlerbehandlung und -reporting machen zu müssen
- kein Overloading (Methoden und Operatoren)
  - Auswahl nur nach Namen ist beim Compilieren viel einfacher und schneller
- kein ternärer Operator `?:`
- Pointer, aber keine Pointerarithmetik

Interfaces

- keine Klassen, aber Interfaces
- Duck Typing

  „Wenn ich einen Vogel sehe, der wie eine Ente läuft, wie eine Ente schwimmt und wie eine Ente schnattert, dann nenne ich diesen Vogel eine Ente."

- d.h. ein Datentyp muss einfach nur die Methoden des Interfaces implementieren

Go Command: https://golang.org/cmd/go/

```
bug         start a bug report
build       compile packages and dependencies
clean       remove object files and cached files
doc         show documentation for package or symbol
env         print Go environment information
fix         update packages to use new APIs
fmt         gofmt (reformat) package sources
generate    generate Go files by processing source
get         download and install packages and dependencies
install     compile and install packages and dependencies
list        list packages or modules
mod         module maintenance
run         compile and run Go program
test        test packages
tool        run specified go tool
version     print Go version
vet         report likely mistakes in packages
```

Programmieren in Go

- im Gegensatz zu anderen Sprachen wird der Go-Code üblicherweise in einem einzigen **Workspace**

```
$ go env GOPATH
/Users/obraun/go
```

- innerhalb des Workspace gibt es
    - `src` — für die Sourcen, die üblicherweise auf verschiedene (Git-)Repositories
    - `bin` — für die ausführbaren Dateien
    - `pkg` — für Package Objects
- das ist (zum Teil) obsolet mit Go $\geq$ 1.11
    - vieles ist aber noch so organisiert

Beispiel Workspace

```
bin/
    hello                          # command executable
    outyet                         # command executable
pkg/
    linux_amd64/
        github.com/golang/example/
            stringutil.a           # package object
src/
    github.com/golang/example/
        .git/                      # Git repository metadata
        hello/
            hello.go               # command source
        outyet/
            main.go                # command source
            main_test.go           # test source
```

Import Path

- der Importpfad muss eindeutig sein
- https://golang.org/doc/code.html schlägt als Beispiel vor für eigene Packages den GitHub-Account zu nutzen, also bei mir z.B. `github.com/obcode`
- d.h. ich würde alles was ich in Go programmiere auf meinem Notebook unter `$GOPATH/src/github.com/obcode` speichern

Go Modules

- seit Go 1.11 gibt es den Befehl `go mod`
- Initialisieren mit `go mod init <path>`
    - erzeugt die Datei `go.mod`
- weitere Pakete werden dann automatisch vom Go-Tool hinzugefügt

Hello World

- Livecoding-Repo: https://github.com/ob-vss-20ss/ob-vss-20ss
- in `$GOPATH/src/github.com/ob-vss-20ss/ob-vss-20ss/hello/hello.go` bzw. nach einem `go mod init github.com/ob-vss-20ss/ob-vss-20ss`

```go
package main

import "fmt"

func main() {
  fmt.Printf("Hello, world.\n")
}
```

- stolen from https://golang.org/doc/code.html
- in `$GOPATH/src/github.com/ob-vss-20ss/ob-vss-20ss/stringutil/reverse.go`

```go
package stringutil

func Reverse(s string) string {
  r := []rune(s)
  for i, j := 0, len(r)-1; i < len(r)/2;
  i, j = i+1, j-1 {
      r[i], r[j] = r[j], r[i]
  }
  return string(r)
}
```

Und nutzen aus der **main**

```go
package main

import (
	"fmt"
	"github.com/ob-vss-20ss/ob-vss-20ss/stringutil"
)

func main() {
	fmt.Println(stringutil.Reverse("!oG ,olleH"))
}
```

Testing (1/3)

- leichtgewichtiges Testframework enthalten
- in `stringutil/reverse_test.go`

Testing (2/3)

```go
package stringutil

import "testing"

func TestReverse(t *testing.T) {
  cases := []struct {
      in, want string
  }{
      {"Hello, world", "dlrow ,olleH"},
      {"Hello, Go", "oG ,olleH"},
      {"", ""},
  }
```

Testing (3/3)

- und der eigentliche Test:

```
  for _, c := range cases {
      got := Reverse(c.in)
      if got != c.want {
          t.Errorf("Reverse(%q) == %q, want %q",
          c.in, got, c.want)
      }
  }
}
```

- ausführen mit

```
go test
```

Remote Dependencies

- mit dem `go`-Tool können auch Remote Dependencies automatisch installiert werden
- Beispiel

```go
import "github.com/ob-vss-20ss/test/stringutil"
```

  und beim nächsten Kommando mit dem `go`-Tool, z.B. `go build`, steht alles zur Verfügung

Sourcecode-Dokumentation

The Go project takes documentation seriously. Documentation is a huge part of making software accessible and maintainable. Of course it must be well-written and accurate, but it also must be easy to write and to maintain. Ideally, it should be coupled to the code itself so the documentation evolves along with the code. The easier it is for programmers to produce good documentation, the better for everyone.

The comments read by godoc are not language constructs (as with Docstring) nor must they have their own machine-readable syntax (as with Javadoc). Godoc comments are just good comments, the sort you would want to read even if godoc didn't exist.

Quelle: https://blog.golang.org/godoc-documenting-go-code

godoc

- auch bei Dokumentation ist der Ansatz etwas anders (pragmatischer)
- mit dem `godoc`-Tool wird die Dokumentation angezeigt, z.B.

```
$ go doc fmt Printf
func Printf(format string, a ...interface{}) (n int, err error)
    Printf formats according to a format specifier and writes to
    standard output. It returns the number of bytes written and
    any write error encountered.
```

- kann auch als HTML angezeigt werden und hat auch eingebetteten Server

```
godoc -http=:6060
```

- GoDoc-Website kann Dokumentation von GitHub-Projekten hosten (siehe https://godoc.org/-/about)

Wie geht es weiter?

- Selbstständige Einarbeitung in Go im Rahmen der ersten Praktikumstermine
- Unterstützung über Mattermost und Issues, bei Bedarf auch BigBlueButton
- siehe Blatt 0 unter https://ob.cs.hm.edu/lectures/vss