Go

Nachfolger von C++?

UNIX-Stammtisch Dresden, 4.1.12

Reinhard Wobst, *r.wobst@gmx.de*

Inhaltsverzeichnis

1. Warum eine neue Sprache?

1.1. Kritik an bisherigen compilierten Sprachen:

- "too much bookkeeping"

```
dynamic_cast<MyData*>funky_iterator<MyData &const*>
    (foo::iterator_type<MyData>(obj))
```

- lange Compilezeiten (25K Zeilen in 131 Files bei `#include <iostream>`)
- anstrengend zu lesender Code
- Fehlermeldungen oft nicht mehr zu verstehen (z.B. QTEST-Makro in Qt)
- Java? Jörg Walter (http://www.syntax-k.de/projekte/go-review) schreibt:

"Now, Java is almost the solution. Almost, were it not for reality."

- In den letzten 10 Jahren soll bei compilierten Sprachen nichts Revolutionäres passiert sein. Die Schmerzen sind aber groß.

1.2. Ziele

"Speed, reliability, simplicity - pick two"

Versuch, alle drei Ziele zu erreichen. Geht das? Ja!

- Go ist klein - typsicher - compiliert - nur 10-20% langsamer als C
- deutlich weniger Schreibarbeit als bei C und vor allem bei C++
- Konzepte aus dynamischen Sprachen wie Python (viele Pythonisten unter den Entwicklern!)

2. Entstehung

Robert Griesemer, Rob Pike and Ken Thompson started sketching the goals for a new language on the white board on September 21, 2007

Wichtige Entwickler - gesponsert von Google:

- **Ken Thompson:** C, UNIX, Plan 9
- **Rob Pike:** UNIX "inventor", Plan 9, Entstehung X-Windows, UTF-8
- **Rob Griesemer:** JavaHotSpot Compiler V8, JS machine of Chrome
- **Ian Taylor**
- **Russ Cox**

(mehr dazu in http://golang.org/doc/go_faq.html)
1 Jahr lang Diskussionen über das Konzept, Sprache am 11.11.2009 vorgestellt.

3. Grober Überblick

3.1. Einige Gimmicks

- C-like
- Funktionalität in Paketen (kleine Sprache, wie bei Python)
- keine Typhierarchie!
- Objektorientierung, aber **keine** Klassen, **keine** Exceptions! s.u.
- "duck typing" und trotzdem typstreng:

```
var n int
var n = 9  // no "int" needed
var n int64 = 1.9
n := 9     // no "int" needed
```

- tuple assignment
- array + slices, boundary check

- maps
- methods on **all** own types
- interfaces (Methodensammlungen)
- Channels (schnelle Kanäle, auch zum Synchronisieren
- Goroutinen (sparsame Threads)!

3.2. Datentypen

3.2.1. Grundtypen:

bool
int, int8, int16, int32, int64
uint
uint8 = byte, uint16, uint32, uint64, float32, float64
complex64, complex128
uintptr

string (ohne abschließendes '\0')

- alle Typen unterscheiden sich (auch bei gleicher Darstellung)
- keine automatische Konvertierung! ("don't panic")
- Konstanten sind "ideal", keine Größe (compilerintern)

3.2.2. Abgeleitete Typen

- **Pointer** (keine Pointerarithmetik!) wie in C: `var p *byte`
- **struct:** `var str struct {m,n int, f float32}`
`var STR str{m:1, n:5, f:3.14}`
- **array:** `var ar [5]int` - wie in C, wird aber selten verwendet; wird als Wert übergeben!
- **slice:** Ausschnitt aus einem Array, z.B.

```
var a []int
a = ar[1:4]   // wie in Python: ar[1], ar[2], ar[3]
s := make([]int, 3)   // slice, array implicit
```

slices sind Referenzen, also werden wie Pointer behandelt!
Darstellung: Tripel aus (Zeiger auf Beginn, Länge, max.Länge)
Interne Funktionen: **`len(slice)`** = Länge, **`cap(slice)`** = max. Länge.
→ Index check! Slices können vergrößert werden, cap() beschränkt die Länge.

- **map:** **`var m map[string] float64`** - Referenz!
- auch **Funktionen** sind ein Typ (Vorgriff auf *function literals*):

```
package main
import "fmt"

func doadd(a, b int, f func(x,y int) int) {
    fmt.Printf("%d\n", f(a,b))
}

func main() {
```

```
    doadd(1,3,
         doadd(1,3, func(a,b int) int {return a+b})
         )
}
```

- Strukturen können um **anonyme Felder** erweitert werden ("Vererbung"):

```
type A struct {
  ax, ay int
}

type B struct {
  A
  bx, by float64
}
```

3.3. Operatoren

Wie C, aber

- ohne "="
- ^ statt ~ als unary operator (Bitkomplement)
- &^ als "bit clear": `5^1 = 4`
- weniger Prioritäten als in C
- ++/-- ist ein Statement, kein Postinkrement: `*p++ == (*p)++`, nicht `*(p++)`
- Tuple assigment, sehr schön bei multi-value return:

```
i,j = j,i
i,j = f(31)
```

Auswertung streng von links nach rechts (bei Funktionsrufen). Sehr oft wird genutzt

```
nbytes, err = Write(buf)
if err {
    ...
```

- Strings können addiert werden

3.4. Kontrollstrukturen

3.4.1. if

```
if i < 10 { ...
if v := f(); v < 10 {...
if n, err = fd.Write(buf); err != nil { ... }
```

3.4.2. switch

```
switch a,b = x[i],y[i]; {
    case a < b:  return -1
    case a == b: return 0
    case a > b:  return 1
}
```

mit **`default`** und **`fallthrough`**; auch Typ-Switch:

```
switch x.(type) {
    case int: ...
    case bool, string: ...
    case func(int) float64: ...
    default: ...
}
```

3.4.3. for

```
for ;; { ... } oder
for { ... }

for i := 0; i < 10; i++ { ...
for i,j := 0,N; i < j; i,j = i+1,j-1 {...}

for i < 10 { ... i++ ...}
```

```
for index, val := range slicevar {
for key, val := range mapvar {
for key, rune := range stringvar {
for val := range channelvar {
```

3.4.4. break, continue, goto

```
Loop: for i := 0; i < 10; i++ {
    switch f(i) {
       case 0, 1, 2: break Loop
    }
    g(i)
}
```

(natürlich auch ohne Marke). Und es gibt "**goto**"!

3.5. Funktionen

"Fast" wie in C:

```
func MySqrt(f float64) (v float64, ok bool) {
    if f >= 0 {v,ok = math.Sqrt(f), true}
    else {v,ok = 0,false}
    return v,ok
}
```

Es kann auch nur "return" heißen:

```
func MySqrt(f float64) (v float64, ok bool) {
    if f < 0 { return } // error case: ok=false, v=0
    return math.Sqrt(f),true
}
```

Function Literals (anonyme Funktionen) haben keinen Namen, vgl. Beispiel oben zu doadd().

3.6. Speicherverwaltung

Es gibt **new()**, aber Speicher wird generell per **garbage collection** verwaltet.

```
make([]byte 100)
```

erzeugt implizit ein Bytefeld, das vom GC freigegeben wird. Eigene Speicherverwaltung kann über Slice geschehen ...

3.7. "Objektorientierung"

Go ist nicht objektorientiert - es gibt **keine** Klassen! **Aber:** Auf jedem selbst deklarierten Datentyp kann man beliebig viele Methoden definieren.
Beispiel:

```
type Point struct { x, y float64 }

// A method on *Point
func (p *Point) Abs() float64 {
    return math.Sqrt(p.x*p.x + p.y*p.y) //not: p->x
}

p := &Point{ 3, 4 }
fmt.Print(p.Abs())
```

Auch (modifiziert "receiver" = n nicht):

```
type myint int
func (n myint) Incmod(p int) {return (n+1)%p}
```

- kein private/public, sondern: Beginnt mit Großbuchstaben (i.S.v. UTF-8) = Variable wird aus Package exportiert
- Scope innerhalb Package/Funktion, kein File-Scope!
- init() in einem Package wird beim Import ausgeführt ("Konstruktor")

3.8. Go-Routinen

Entspricht grob dem thread.start_new_thread() von Python: Funktion wird auf eigenem Stack mit gemeinsamen globalen Variablen ausgeführt.

Beispiel:

```
package main

import "fmt"
import "time"

func muh() {
    for i := 0; i < 10; i++ {
        fmt.Printf("muh\n")
        time.Sleep(15.e7)       // nanoseconds
    }
}
```

```
func maeh() {
    for i := 0; i < 10; i++ {
        fmt.Printf("maeh\n")
        time.Sleep(10.e7)
    }
}

func main() {
    go muh()
    go maeh()
    time.Sleep(2.e9)
}
```

Man nutzt *segmented stacks* (split stacks); vgl.a. http://gcc.gnu.org/wiki/SplitStacks (Ian Taylor ist einer der Hauptentwickler von Go); **Ziel**: weniger Speicherverbrauch bei sehr vielen Threads. "Kern-Konzept" von Go.

Problem: Fehlende Synchronisation. Go-Prinzip:

"Don't communicate by sharing memory. Instead, share memory by communicating."

Lösung für obiges Listing: *gochan.go*

Beispiel vom Entwicklerteam:

```
package main
import "fmt"
const N = 100000

func f(left, right chan int) { left <- 1 + <-right }

func main() {
    leftmost := make(chan int)
    var left, right chan int = nil, leftmost
```

```
    for i := 0; i < N; i++ {
        left, right = right, make(chan int)
        go f(left, right)
    }

    right <- 0      // bang!
    x := <-leftmost // wait for completion
    fmt.Println(x)  // N
}
```

Analog in Python: *pychain*
Speicherverbrauch pro Thread: ca. 4KB Go, etwa 8.5MB bei Python??
Channels sind unbuffered (default) oder buffered
select-Anweisung: gleichzeitiges Lauschen an mehreren Channels (timeout implementation)

3.9. Interfaces

Das vielleicht genialste Prinzip von Go.
Python: "file-like object, must have write() and close()"; "has next() =~ Iterator" ... oft in Dokumentation zu lesen.
Go: Interface = Sammlung von Methoden (Signaturen); jeder Datentyp kann beliebig viele Interfaces implementieren.

Beispiel: *Largest.go*

Interfaces sind abstrakte Datentypen, können als Typ von Funktionsargumenten deklariert werden.

empty Interface: "void *", aber sauber

→ Damit erledigen sich viele Probleme, die C++/Java mühsam und oft starr über Klassenhierarchie/Design Patterns lösen!

3.10. defer, panic, restore = "Ausnahmebehandlung"

3.10.1. defer

Vormerken eines Funktionsrufes beim Verlassen einer Funktion.

Beispiel:

```
func data(fileName string) string {
    f := os.Open(fileName)
    defer f.Close()
    contents := io.ReadAll(f)
    return contents   // f.Close() called here
}
```

`defer f(args...)` entspricht in Python sozusagen `atexit.register(f, (args...))` auf Funktionsebene. Wird in Python über "context manager" ("with"-statement) gelöst, bei weitem weniger flexibel.

3.10.2. panic

= "Ausnahme werfen", *run-time panic* (z.B. Division durch 0 etc.) →

- defer-Funktionen ausführen
- dann Funktion abbrechen
- defer-Funktionen auf nächsthöherer Ebene ausführen
- setzt sich durch Stack fort (sauberer Abbruch der Goroutine).

3.10.3. recover

Beispiel: Abfangen von Panic beim Ruf von Funktion g():

```
func protect(g func()) { // g has "function" type!
 defer func() {
```

```
  log.Println("done")
  // Println executes normally even if there is a panic
   if x := recover(); x != nil {
    log.Printf("run time panic: %v", x)
   }
 }()      // function is anonymous and called!
 log.Println("start")
 g()
}
```

Details vgl. Sprachreferenz.

3.11. Closures

Anonyme Funktion wird innerhalb von Funktion definiert und greift dabei auf Variable der umhüllenden Funktion zu: diese Werte bleiben nach dem **`return`** erhalten:

```
func adder() func(int) int {
    var x int
    return func(delta int) int {
        x += delta
        return x
    }
}

func main() {
    f := adder()
    fmt.Println(f(1))
    fmt.Println(f(20))
    fmt.Println(f(300))
}
```

Ausgabe:

```
1
21
321
```

Unterschied zu Python-Generatoren - dort bleiben alle lokalen Variablen erhalten, keine umhüllende Funktion! (Python kennt aber auch Closures.)

4. Programmieren in Go

- in Go denken, nicht in C/C++/Python/Java ...; slice, tuple assignment, function object, package, interface, channel, goroutine
- im Unterschied zum klassischen OO keine aufwändige Design-Entscheidung vorher - praxisnäher!
- meist kurze, verständliche Fehlermeldungen
- intolerante Compiler: keine nicht verwendete Variable oder import-Anweisungen erlaubt
- vorerst noch mit mitgelieferten Makefiles, Unterstützung durch mitgelieferte Tools:
  - Plan9-Compiler 8g (x86) / 6g (amd64) sowie gccgo (langsamer, bessere Optimierung)
  - gomake, gb, go (build tools)
  - gofmt - automatische Formatierung, auch über vim-Plugin -tabindent=false - wird zu -tabs=false

- godoc - extrahiert Dokumentation aus Programmen und Paketen
- gotest - Unittest-Framework (erinnert etwas an Python/Junit, noch nicht so ausgebaut)

5. Aktueller Stand

5.1.1. Ausblick

- 250 Pakete (150 im Standard-Download derzeit)
- oben erwähnte Tools werden weiterentwickelt
- >200 Contributors
- läuft bisher auf UNIX/Windows
- sehr aktive Entwicklung und Mailinglisten
- Go1 soll Anfang 2012 erscheinen

5.1.2. Nachteile

- Dokumentation ist noch zu schlecht organisiert, auch Sprachreferenz noch unvollständig
- keine Standardargumente, keine benannte Argumente (Diskussion läuft, man ist vorerst dagegen) - es gibt Workarounds (Initialisierung von Strukturen)

- Schreibweise ist ungewohnt
- Toolchain unvollständig (Debugger, nur make) - scons + waf am Horizont
- Referenz / by value irritierend
- Pakete bieten noch nicht den Komfort und die Vollständigkeit, den man z.B. von Python gewohnt ist - in speziellen Fällen kann Python sogar schneller sein! (regexp-Implementierung)
- Format:
  - Gebrauch von Klammern nicht ganz konsistent
  - automatisches Einfügen von Semikolon (Zwang zu K&R Style)
  - raw strings `...`
- noch keine Operatorfunktionen (kann noch kommen)
- noch keine Generics (Diskussion läuft, vielleicht überflüssig)
- Windows Support hinkt hinterher

6. Das Go-Maskottchen

7. Links

Go-Hauptseite: www.golang.org

Ezellentes Go-Buch, noch in Arbeit: http://go-book.appspot.com

Go user group: http://groups.google.com/group/golang-nuts

Tutorials: http://golangtutorials.blogspot.com/2011/05/table-of-contents.html