www.javaspektrum.de

Ausgabe 5/2018
D € 7,90 · A € 8,90 · SFR 14,30

4 194156 107907 05

# JavaSPEKTRUM

Magazin für professionelle Entwicklung und Integration von Enterprise-Systemen

# Javas Zukunft – Java 9 versus Kotlin & Co.


## Alternativen für die Cloud

## Java und Go im Vergleich

Johannes Weigend, Johannes Siedersleben



Javas Zukunft – Java 9 versus Kotlin & Co.

G30325

## Alternativen für die Cloud

# Java und Go im Vergleich

**Johannes Weigend, Johannes Siedersleben**

**Java ist nach dem Tiobe Index 2018 unangefochten auf Platz 1 bei den weltweit eingesetzten Programmiersprachen. Die Sprache ist ausgereift, stabil und verfügt über ein immenses Open-Source-Ökosystem. Was will man mehr? Obwohl Java gerade für die Backend-Entwicklung attraktiv ist, hat Google 2009 eine eigene Programmiersprache Open Source gestellt: Golang oder kurz Go. Dieser Artikel beleuchtet die Stärken und Schwächen von Go gegenüber Java, gibt Hinweise, für welche Projekte Go eine gute Alternative ist, und wie ein „Best of Breed"-Ansatz aussehen kann.**

Interessant an Go ist, dass die Grundbausteine von Cloud-Plattformen (OpenShift oder die Google-Container-Plattform) mit Go erstellt wurden. Docker, Kubernetes, Helm, Grafana oder Prometheus – alles ist mit Go programmiert. Die Fragen aus der Sicht von Java-Experten sind:

- Was macht Go für die Cloud so interessant?
- Gibt es Funktionen, die Java-Programmierer kennen sollten, und wenn ja, welche?

## Über Go

Go wurde seit 2007 von Google entwickelt und am 10.11.2009 Open Source gestellt [Go]. Die Erfinder sind prominent:

- Ken Thompson: Erfinder von Unix und der Programmiersprache B, dem Vorläufer von C,
- Rob Pike: Entwickler von Unix und UTF-8 (zusammen mit Ken Thompson) sowie
- Robert Griesemer: Entwickler der Java Hotspot VM bei Sun Microsystems.

Das wichtigste Ziel der Go-Designer war es, eine einfache Sprache für Cloud, Cluster und Netzwerk-Computing sowie Systemprogrammierung zu entwickeln als Ersatz für C/C++. Wichtige Aspekte waren Skalierbarkeit auf Mehrkernmaschinen und schnelle Compilierung. Der C++-Code von Google leidet an hoher Komplexität und langen Compilierzeiten. Ein weiteres Ziel: Programmieren mit Go soll Spaß machen! Dazu gehören ein schneller Compiler und Sprachfeatures, die man von dynamischen Skriptsprachen wie Python, Ruby oder JavaScript kennt. Syntaktischer Zucker, wie die statische Initialisierung von Maps oder Arrays, ist in Go umgesetzt. Trotzdem ist Go eine minimalistische Sprache mit weniger Schlüsselwörtern als ANSI C (Go 25 vs. C 32).

Go basiert auf zwei unabhängigen Entwicklungssträngen mit der gemeinsamen Wurzel Algol 60: Der eine Strang ist der europäische mit Algol 68, Pascal, Modula und Oberon, die von Nicolas Wirth entwickelt wurden. Der andere basiert auf C, Objective-C, C++ und Java. Go führt beide Stränge nach 50 Jahren wieder zusammen [Grie15].

Go ist heute bei vielen Firmen in der Backend-Entwicklung angekommen (Google, Red Hat, Docker, Cloud Foundry u.v.m.). Fast alle Bausteine der Cloud Native Computing Foundation (Teil der Linux Foundation) wurden in Golang programmiert. Gleiches gilt für den populären Cloud-Scheduler Kubernetes (k8s) sowie für Docker – Grund genug, sich Go im Detail anzusehen.

## Wo unterscheiden sich Java und Go?

Tabelle 1 zeigt, wo sich die beiden Sprachen unterscheiden und wo sie gleich sind. Beide Sprachen werden kompiliert, verfügen über Garbage-Collection und ein umfangreiches Tooling.

Programmiersprachen beurteilt man am besten, indem man Quellcode betrachtet. Fangen wir an.

**Johannes Weigend** ist technischer Geschäftsführer bei der QAware GmbH, München. Er entwirft und entwickelt seit mehr als 20 Jahren datenintensive Systeme mit Java und C++. Johannes Weigend ist Java Rockstar und Experte für Diagnose von Cloud-nativen Anwendungen. E-Mail: johannes.weigend@qaware.de

**Prof. Dr. Johannes Siedersleben** ist Beirat der QAware GmbH, München. Er hat mit seinen Publikationen das Thema Softwarearchitektur in Deutschland und darüber hinaus beeinflusst. Er berät die QAware in allen Gebieten des Software-Engineering. E-Mail: johannes.siedersleben@qaware.de

*Tabelle 1: Java und Go im Vergleich*

| Feature | Java | Go |
|---|---|---|
| **Allgemein** | | |
| Dateiformat | beliebig | UTF-8 |
| Mehrfache Rückgabewerte | nein | ja |
| Zeiger | nein | ja |
| Ausnahmebehandlung | ja | panic/recover |
| Generics | ja | nein |
| Garbage-Collection | ja | ja |
| Statischer Linker | nein | ja |
| Reflection | ja | ja |
| Syntax für die Initialisierung von Maps | nein | ja |
| **Nebenläufige Programmierung** | | |
| Nebenläufige Abläufe | Threads | Go-Routinen |
| Synchronisation | Monitore + Locks | Channels + Locks |
| **Objektorientierte Programmierung** | | |
| Klassen kapseln Daten und Funktionen | ja | getrennt |
| Generische Programmierung | Basisklasse Object | interface{} |
| Vererbung | Ja (einfach) | Embedding (mehrfach) |
| Polymorphismus | Klassen + Interfaces | nur Interfaces |
| **Funktionale Programmierung** | | |
| Anonyme Funktionen | Lambda | Function Literals |
| Methoden Referenzen | ja | ja |

## Wert und Referenz

Go kennt wie C den Unterschied zwischen Wert und Referenz: `a int` definiert eine Integervariable, `pa *int` eine Referenz auf Integer. Nach `pa = &a` zeigt `pa` auf `a`, `*pa` liefert den referenzierten Wert, also wieder `a`. C-Programmierer werden sich freuen. Die Go-Syntax wurde gegenüber C etwas eingeschränkt: `&&ppa` ist nicht erlaubt, aber `&pa` und `***pppa` sind möglich.

Es gibt keine Zeigerarithmetik. Im Gegensatz zu C oder C++ werden Zeiger mit `nil` initialisiert. Die Garbage-Collection von Go verhindert Zeiger, die ins Leere zeigen. Parameterübergabe ist by value. Die swap-Funktion hat daher die Signatur `swap(pa, pb *int)`: Die Zeiger `pa`, `pb` werden by value übergeben und nicht verändert, der referenzierte Inhalt wird getauscht.

Java unterscheidet primitive Datentypen, Objekte und, ab Java 10, Value Types. Die Entscheidung *Wert oder Referenz* trifft Java: Objekte sieht der Programmierer per Referenz, primitive Datentypen und Value Types als Wert. In Go entscheidet der Programmierer: Zu jedem Datentyp gibt es den entsprechenden Referenztyp; der Programmierer hat die Wahl und damit die Kontrolle über Speicherlayout und Allokationsstrategie. Flach gespeicherte Strukturen und Arrays vereinfachen die Allokation (1-mal statt n-mal) und liegen benachbart, sodass Prozessor-Caches ideal greifen.

## Wo sind die Lambdas?

Keine Sorge, die gibt es. Funktionen sind Go-Datentypen wie andere auch; die Signatur ist Bestandteil des Datentyps. Das Codefragment in Listing 1 zeigt einige Elemente funktionaler Programmierung in Go: Die Funktion `f` liefert bei n Aufrufen die ersten n Quadratzahlen, `Exec` führt die übergebene Funktion aus und `faculty` liefert eine parameterlose Funktion zurück, die bei n Aufrufen die n ersten Fakultäten ausgibt. Go hat keine eingebauten Iteratoren wie Java.

```
package functional
var i int // an int variable initialized to 0

// i part of closure of f
func squares() int {
  i++; return i*i
}
g := f // f now accessible as g

// Exec accepts a function as parameter
func Exec(func g() int) int {
  return g()
}
// an anonymous function
func faculty() func() int {
  current, factor := 1, 0
  return func() int {
    factor++
    current *= factor
    return current
  }
}
```

*Listing 1: Funktionale Programmierung mit Go*

Damit sind die wesentlichen Elemente der funktionalen Programmierung vorhanden, mit einer wichtigen Einschränkung: Es gibt keine Generics. Generische Programmierung läuft so ähnlich wie im alten Java mit `Object` als generischem Typ und jeder Menge Downcasts.

## Objektorientierte Programmierung

Go trennt Funktionen und Daten genauso wie C. Die klassische OO-Punktnotation wird wie gewohnt unterstützt, indem man das aktuell Objekt (in Java `this`) als Parameter vor den Methodennamen schreibt. Der Name `this` ist bei Go-Programmierern allerdings nicht gerne gesehen; der Go-Syntax-Checker `vet` gibt bei Verwendung von `this` eine Warnung aus. Alle Typen, Variablen und Funktionen, die mit einem Großbuchstaben beginnen, sind implizit `public`. Alle klein geschriebenen Bezeichner sind implizit `package private`. Ein hartes `private`, das die Sichtbarkeit auf die eigene Klasse beschränkt, gibt es in Go nicht – es gibt ja auch keine Klassen im Sinn von Java.

Go kennt keine Vererbung. Die Go-Autoren haben sich von dem OO-Leitsatz „Favour composition over inheritance“ leiten lassen. Tiefe Klassenhierarchien sind schlechtes Design, was jeder bestätigt, der schon einmal in der fünften Ebene einer Klassenhierarchie programmiert hat. Go kann aber Strukturen aus Strukturen zusammensetzen (Embedding) und vermeidet die aufgeblähte Java-Syntax beim Delegieren. Go erlaubt das Überschreiben und Überladen eingebetteter Methoden. Listing 2 zeigt ein einfaches OO-Beispiel.

```
// Rational repräsentiert einen Bruch (numerator/denominator)
type Rational struct {
  numerator int
  denominator int
}
// NewRational Konstruktor als Funktion
func NewRational(numerator int, denominator int) Rational {
  if denominator == 0 {
    panic("division by zero")
  }
  r := Rational{}
  divisor := gcd(numerator, denominator)
  r.numerator = numerator / divisor
  r.denominator = denominator / divisor
  return r
}
// Multipy multipliziert zwei rationale Zahlen und
// gibt das Ergebnis zurück
func (x Rational) Multiply(y Rational) Rational {
  return NewRational(x.numerator * y.numerator,
    x.denominator * y.denominator)
}
var r,s,t Rational
r = NewRational(3,4); s = NewRational(5,6); t = r.Multiply(s)
...
```

*Listing 2: Eigene Datentypen mit Go*

Der Typ `Rational` repräsentiert Brüche, die aus Zähler und Nenner zusammengesetzt sind. Das Beispiel zeigt die Deklaration der Struktur (`struct{}`). Die Funktion `NewRational` ist der zugehörige Konstruktor, `Multiply` multipliziert zwei rationale Zahlen. Konstruktoren folgen der Konvention: `func NewT(p1, ..., pn) T`. Der Konstruktor `NewRational` erzeugt eine rationale Zahl (keine Referenz) und mit teilerfremdem Zähler und Nenner.

Man schreibt Methoden zu einer Struktur, indem man das Empfängerobjekt (hier `x Rational`) vor den Methodennamen setzt. Im Gegensatz zu Java, wo alle Methoden einer Klasse in der Klasse selbst stehen, kann man in Go an beliebiger Stelle neue Methoden hinzufügen.

Der Typ `Rational` arbeitet mit Werten (Call by value). Dadurch sind `Rational`s automatisch unveränderbar (immutable). In Java wird Unveränderbarkeit per Konvention umgesetzt: Methoden geben neue Objekte zurück, die Attribute sind final. Alle Datentypen im JDK sind so gebaut.

## Ausnahmebehandlung

Ähnlich wie mit Java Runtime Exceptions kann man in Go mittels `panic` eine Ausnahme melden. Die Ausnahmebehandlung läuft den Stack zurück bis zur ersten `recover`-Anweisung (dem `catch` in Go) oder bis das Programm endet. Insofern ist es falsch zu behaupten, Go habe keine Ausnahmebehandlung. Beim Aufruf von `panic` kann man beliebige Informationen mitgeben. Normale Fehler meldet man in Go mit Returnwerten, die multipel sein können wie in Python.

## Interfaces

Interfaces sind wie in Java eigene Typen. Go unterstützt die Zusammenführung mehrerer Interfaces durch Einbettung (Embedding). Ein besonderer Typ ist das leere Interface `interface{}`, das ungefähr die Rolle von `Object` in Java übernimmt. Man verwendet zum Beispiel `interface{}` als Typ für generische Container (s. Listing 3).

```
// Stack is a generic LIFO container for untyped objects
type Stack []interface{}

// NewStack constructs an empty stack at the heap
func NewStack() *Stack {
  return new(Stack)
}
// Push pushes a value on the stack
func (s *Stack) Push(value interface{}) {
  *s = append(*s, value)
}
```

*Listing 3: Generische Container am Beispiel Stack*

Der `Stack` basiert auf einem Array von beliebigen Objekten (`interface{}`). Die Klasse `Rational` verwendet als Parameter Werte, also Kopien der Originale. Im Gegensatz dazu ist `Stack` ein Referenztyp, der bei `Push` und `Pop` natürlich modifiziert wird. Der `Stack` kann beliebige Typen beherbergen, aber der Programmierer trägt die Verantwortung für die Typsicherheit. Downcasts schreibt man in der Form `.(T)` an das Ende eines Ausdrucks (wie einen Funktionsaufruf). Bei Go spricht man von Type Assertions, nicht von Downcast (s. Listing 4).

```
s := NewStack()
s.Push(1)
s.Push(2)

sum := 0
for i := 0; i < len(*s); i++ {
  sum += (*s)[i].(int) // type assertion = cast
                       // from interface{} to int
}
```

*Listing 4: Casts in Go: Type Assertions*

## Unittests

Go unterstützt das Bauen und Ausführen von Unittests. Mit `go test` können Tests direkt ausgeführt werden. Ebenso kann man die Test-Coverage ermitteln und in einem Lauf alle Unittests der gesamten Codebase ausführen.

Listing 5 zeigt einen Unittest für den Typ `Stack`. Unittests liegen in gleichnamigen Dateien mit der Endung `T_test.go`. Die Testdateien werden nicht in das finale Executable gepackt und auch bei

Bibliotheken automatisch entfernt. Damit erübrigt sich ein eigener Verzeichnisbaum für Testdateien (wie bei Maven erforderlich). Testressourcen lassen sich in entsprechenden `T_test`-Verzeichnissen ablegen.

```
func TestGenericStack(t *testing.T) {
  s := NewStack()
  s.Push(1)
  s.Push(2)

  res := s.pop()
  if res != 2 {
    t.Error(fmt.Sprintf("Result should be 2 but is %v", ))
  }
}
```

*Listing 5: Unittests mit Go*

Go kann direkt aus den Unittests die Testabdeckung messen und im Browser anzeigen (mit dem Tool `cover`). IDEs wie IntelliJ oder Visual Studio Code verwenden diese Daten ebenfalls für die Anzeige der Testüberdeckung.

## Go-Routinen

Bisher haben wir keine Features gesehen, die Go wirklich von Java unterscheidet. Die folgenden Features machen Go jedoch besonders und sind in Java nicht ohne Weiteres nachzubilden.

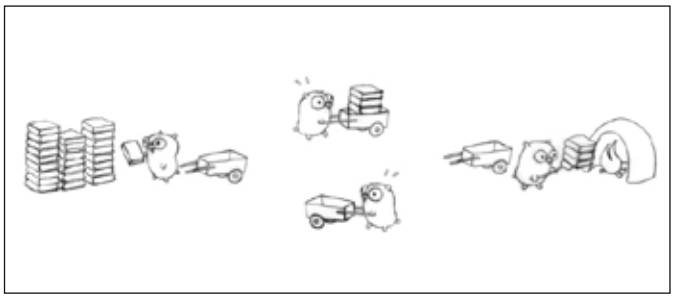

*Abb. 1: Nebenläufigkeit in Go*
*(Quelle: Rob Pike, https://talks.golang.org/2012/waza.slide#14)*

Funktionsaufrufe in Go sind normalerweise synchron. Aber mit dem Schlüsselwort `go` vor dem Aufruf startet die Funktion asynchron (s. Listing 6), der Aufruf kehrt sofort zurück.

```
// synchron
LongRunningFunction()

// asynchron
go LongRunningFunction()
```

*Listing 6: Asynchrone Funktionen*

Die Go-Laufzeitumgebung verwendet einen Thread-Pool und startet nur so viele Threads, wie CPU-Kerne vorhanden sind. Go-Routinen werden auf die vorhandenen Threads verteilt. Jeder Thread führt in der Regel viele Go-Routinen parallel aus. Mit dem Aufruf `go f()` wird die gerufene Funktion in der Regel in einem vorhandenen Thread untergebracht.

Das Scheduling der Go-Routinen eines Threads ist kooperativ (wie bei UI-Technologien). Der Scheduler bekommt immer dann die Kontrolle, wenn eine Go-Routine eine andere Funktion aufruft oder wenn sie wartet (Sleep/IO-Wait). Dieses Modell ist einfach und leichtgewichtig. Es erlaubt, zigtausende Go-Routinen in Sekundenbruchteilen zu starten und abzuarbeiten. Mögliches Problem: Wenn sich eine Funktion unfreundlich verhält (z. B. CPU-intensive Endlosschleife ohne innere Funktionsaufrufe), können andere Go-Routinen verhungern (Starvation). Das kommt allerdings in der Praxis kaum vor (Log-Aufrufe sollten in jeder länger dauernden Aktion vorhanden sein).

## Synchronisierung durch Channels

In Java synchronisiert man Code durch Locks oder mittels `synchronized` (Monitor). In der Go-Bibliothek finden sich ähnliche Locks wie in java.util.concurrent. Go bietet einen weiteren Mechanismus an, nämlich Channels: Der Sender blockiert beim Senden in den Channel solange, bis ein Empfänger daraus liest. Mit diesem Handshake lassen sich Synchronisationsprobleme einfach lösen. *Don't communicate by sharing memory; share memory by communicating* ist das Motto des Erfinders der Channels, Rob Pike. Go-Routinen kommunizieren über Channels einfach und sicher. Locks braucht man nur in Sonderfällen, sie gelten als Low-Level-Programmierung.

```
c := make(chan *int) // Erzeuge Int-Channel
go func() {
  c <- 1 // Send: Aufruf blockiert bis ein Empfänger liest
}()
go func() {
  <-c // Receive: Aufruf blockiert bis ein Sender schreibt
}()
```

*Listing 7: Synchronisation mit Channels*

Das Beispiel in Listing 7 zeigt, wie man einen Channel erzeugt, hineinschreibt und ausliest. Leser und Schreiber sind in der Regel unterschiedliche Go-Routinen.

Mit Channels kann man viele Klassen aus `java.util.concurrent` vereinfachen. Ein Beispiel dafür ist `java.util.concurrent.BlockingQueue`. Es ist möglich, die Java-Klasse weitestgehend identisch nach Go zu portieren (mit Locks, Signals, Wait und Notify). Eine Implementierung, die auf Channels aufbaut, ist aber kürzer und besser lesbar (s. Listing 8).

```
// BlockingQueue is a FIFO container with a fixed capacity
// It blocks a reader when it is empty and a writer when it is full
type BlockingQueue struct {
  channel chan interface{}
}
// NewBlockingQueue constructs a BlockingQuee with a given capacity
func NewBlockingQueue(capacity int) *BlockingQueue {
  q := BlockingQueue{make(chan interface{}, capacity)}
  return &q
}
// Put puts an item in the queue and blocks it the queue is full
func (q *BlockingQueue) Put(item interface{}) {
  q.channel <- item
}
// Take takes an item from the queue and blocks if the queue is empty
func (q *BlockingQueue) Take() interface{} {
  return <-q.channel
}
```

*Listing 8: Blocking Queue*

## Tooling

Für Java-Entwickler ist IntelliJ die populärste IDE. Auch Go-Entwicklung funktioniert mit IntelliJ richtig gut. Dafür gibt es ein Plug-in von JetBrains oder die Goland IDE (eine spezielle Variante von IntelliJ). Eine schöne Alternative ist auch Microsoft Visual Studio Code. Mit den Tools der Go-Installation kommt man sehr weit. Go kann bereits mit der Standard-Installation:

- Compile, Build, Run von Anwendungen aus vielen Packages/Quelldateien
- Cross-Compiling für Linux, OSX und Windows (z. B. `GOOS=Windows; go build` produziert ein Windows .exe auf OSX oder Linux)
- Unit-Testing mit Test-Abdeckung
- Profiling mit pprof (CPU und Memory)
- Formatierung von Quelldateien mit go fmt
- Quellcode-Style-Checker mit go vet
- Slideshows mit ausführbarem Go-Code im Browser
- u.v.m.

In Java funktionieren Profiling und das Ermitteln von Coverage-Daten nur mit Dritt-Tools. Alle Java-IDEs können Style-Checks und Formatierung, aber jede macht es anders. Mit Java 9 lassen sich Anwendungen mit Jigsaw zu Installationen packen (inkl. abhängiger Bibliotheken/Module). Neben den JDK-Werkzeugen wie javac, jar, jlink braucht man unbedingt ein Build-Skript oder ein Build-Tool wie Gradle oder Maven. Für Go gibt es das Tool `go` als Einstiegspunkt für alle Aufgaben.

Da man bei größeren Projekten auch bei Go mehrere Build-Schritte benötigt, sind Makefiles beliebt. Sie sind einfach und kurz, da das Go-Tooling fast alle Aufgaben in einem Kommando erledigt. Makefiles bieten sich auch an, wenn wir ein Frontend für eine Menge von Shell-Skripten (z. B. für Docker, Kubernetes usw.) brauchen. Wer Makefiles nicht mag, kann seine Go-Build-Skripte auch mit Gradle bauen. Go verfügt über den eingebauten Profiler pprof, dessen Ausgabe zum Beispiel zur Generierung von Flamegraphs (s. Abb. 2) dient. Flamegraphs sind Darstellungen, die den Callstack der Funktionen in Balken stapeln. Die Balkenbreite zeigt den Anteil der Laufzeit.

Eine gute Einführung in die Welt der Go-Werkzeuge ist [Cam17].

## Microservices mit Go

Go eignet sich gut für die Entwicklung von Microservices mit XML, JSON oder auch Google RPC. Die `ListenAndServe()`-Funktion aus dem Standard-HTTP-Paket stellt einen parallelen Webserver zur Verfügung, der die eingehängte Handler-Funktion in einer eigenen Go-Routine pro Request aufruft.

## Customer-Service

`http.ListenAndServe()` (s. Listing 9) blockiert endlos und kehrt erst im Fehlerfall zurück. In dem Fall wird der Fehler geloggt und das Programm beendet. Die Funktion `getAllCustomers()` transformiert einen Go-Array von Customers in JSON. Das geht in Go mit einem Einzeiler, wie Listing 10 zeigt. Go verwendet Struct-Tags wie Java-Annotationen. Damit wird das JSON-Marshalling gesteuert (s. Listing 11).

Eine gute Einleitung für die Erstellung von Webservices gibt es online [Lan14].

```
func main() {
  http.HandleFunc("/customer", getAllCustomers)
  log.Fatal(http.ListenAndServe(":8080", router))
}
```

*Listing 9: Microservices mit Go - main()*

```
func getAllCustomers(w http.ResponseWriter, r *http.Request) {
  if err := json.NewEncoder(w).Encode(customers); err != nil {
    panic(err)
  }
}
```

*Listing 10: Microservices mit Go - JSON Marshalling*

```
type Customer struct {
  Name string `json:"name"`
  ...
```

*Listing 11: Microservices mit Go-Annotationen*

## Go und Docker

Go und Docker passen gut zusammen. Es gibt Standard-Docker-Images zum Kompilieren von Go-Code. Go-Programme benötigen keine VM, keine eigene Laufzeitumgebung oder sonstige Abhängigkeiten. Da Go direkt eine ausführbare, statisch gebundene Datei erzeugt, bekommt man minimale Container ohne Ballast.

Das Beispiel in Listing 12 verwendet den Apine-Linux-Container einschließlich der Go-Werkzeugpalette. Der fertige Container enthält nur noch das ausführbare Programm und ist damit nur wenige MB groß. Ein Java-Programm benötigt dafür einen Docker-Container mit Dateisystem und JRE und viel Hauptspeicher. 64 MB sind das absolute Minimum. In der Praxis kann ein kleiner Microservice sogar 512 MB Hauptspeicher brauchen. Bei Go kommen wir mit 64 MB sehr weit – für viele Dienste wird das ausreichen.

Sehr Cool :-)

```
# Dockerfile
FROM golang:alpine as builder
RUN mkdir /build
ADD . /build/
WORKDIR /build
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo
  -ldflags '-extldflags "-static"' -o main .
FROM scratch
COPY --from=builder /build/main /app/
WORKDIR /app
CMD ["./main"]
```

*Listing 12: Dockerfile für minimalen Container*

Der Trick besteht darin, als Ziel-Container `scratch` anzugeben. Das ist ein leerer Container ohne Linux-Dateisystem. Außerdem werden bei `go build` entsprechende Linker-Flags angegeben. Ergebnis ist ein statisch gelinktes Programm ohne externe Abhängigkeiten

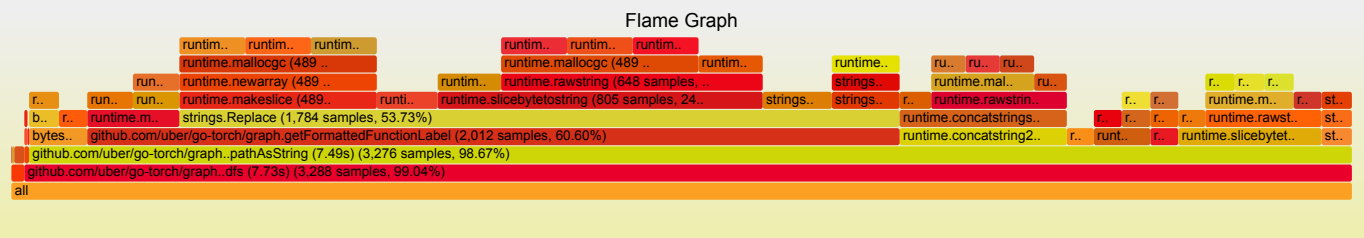


*Abb. 2: Flamegraph (Quelle: Uber Inc., https://github.com/uber/go-torch)*

zu anderen Shared Libraries. Daher ist Go für Docker basierte Laufzeitumgebungen insbesondere für Tools, Sidecar Container in der Cloud besser geeignet als Java.

## Wie kann ein „Best of Breed"-Ansatz aussehen?

Die Beispiele zeigen, wie knapp die Syntax von Go gegenüber Java ist. Für erfahrene Java- oder C++-Programmierer ist Go schnell zu lernen und macht Spaß. Die Vorträge und Präsentationen der Go-Erfinder sind frei verfügbar und geben einen guten Einblick in die Designentscheidungen und Sprachkonzepte. Mit Vorsicht und Erfahrung lassen sich mit Go kompakte und gut lesbare Programme schreiben. Aber man kann viel falsch machen. Go ist keine Sprache für Anfänger. Für systemnahe Anwendungen in der Cloud ist Go sicher das Mittel der Wahl.

Für Java spricht die enorme Reife, die Mächtigkeit der Sprache, die Vielfalt der Tools und der Bibliotheken. Aber das ist auch ein Ballast, den man mitschleppt: Vielleicht ist die Sprache inzwischen zu mächtig, die Tools zu vielfältig. Go ist schlank und schnell; es hat genau die Tools, die man braucht, aber nicht mehr. Ein großer Teil der JDK lässt sich mühelos in Go nachbauen. Hätte jemand Lust? Channels sind einfach, schnell und robust. Asynchrone Programmierung ist immer gefährlich, aber Channels sind ein guter Sicherheitsgurt. Das einzige, was wirklich fehlt, sind Generics. Braucht man so etwas?

## Literatur und Links

**[Cam17]** F. Campoy, Go Tooling in Action, https://github.com/campoy/go-tooling-workshop
**[Grie15]** R. Griesemer, The Evolution of Go, 2015, https://talks.golang.org/2015/gophercon-goevolution.slide#1
**[Go]** https://golang.org/
**[Lan14]** C. Lanou, Making a RESTful JSON API in Go, 2014, https://thenewstack.io/make-a-restful-json-api-go/
**[Pik12]** R. Pike, Go Concurrency Patterns, https://talks.golang.org/2012/concurrency.slide
**[TIOBE]** https://www.tiobe.com/tiobe-index/

# IT-Probleme lösen.
# Digitale Zukunft gestalten.

QAware GmbH München
Aschauer Straße 32
81549 München
Tel.: +49 89 232315-0
Fax: +49 89 232315-129
E-Mail: info@qaware.de

QAware GmbH Mainz
Rheinstraße 4 C
55116 Mainz
Tel.: +49 6131 21569-0
Fax: +49 6131 21569-68
E-Mail: info@qaware.de

QAWARE
SOFTWARE ENGINEERING