iX DEVELOPER

Winter 2019/20

12,90 €
Österreich 14,20 €
Schweiz 25.80 CHF
Luxemburg 14,80 €
Italien, Spanien 16,80 €
www.ix.de

code(), build(), deploy()

Moderne Softwareentwicklung

Cloud-native & Serverless

Das Jahr der Service-Meshes

FaaS und Kubeless auf Kubernetes

Chaos Engineering

Besser programmieren

Mehr Sicherheit dank DevSecOps

Git-Funktionen ausreizen

Interviews: Martin Fowler und Uncle Bob Martin

Machine Learning

Deep-Learning-Modelle kompatibel portieren

TensorFlow 2 und Machine Learning im Browser

Sprachenvielfalt

Rust, Elixir und Elm:
skalierbar, robust und sicher

Koroutinen in Kotlin und Go

Rheinwerk Konferenz für

KOTLIN

16. – 17. März 2020
im KOMED / Köln

Zwei Tage Kotlin, zwei Tage volles Programm!

Von Java zu Kotlin, Sprachfeatures, Mobile und Backend: Die KKON ist die Konferenz, um intensiv in Kotlin einzutauchen. **Ganztägige Workshops** und **Fachvorträge** von den führenden Kotlin-Experten machen Sie fit in Kotlin. Tauschen Sie sich mit den Profis aus und holen Sie sich wertvolles Wissen für Ihre Projekte.

Mit den internationalen Kotlin-Koryphäen **Annyce Davis**, **Shelby Cohen** und **Katie Levy** sowie **Eugenio Marletti** und **Sebastian Aigner** von JetBrains.

Sechs Gründe, dabei zu sein!

- Ihre zwei Tage mit den Besten der Branche – inkl. JetBrains Kotlin Clinic
- Vorträge zu allen wichtigen Kotlin-Themen
- Drei Intensivseminare für Softwareentwickler
- Alle Sessions auf Video, alle Folien der Speaker
- 100 Prozent Rheinwerk
- Leckeres Essen, tolle Location

Tickets und alle Infos: **www.rheinwerk-kkon.de**

Unsere Sponsoren und Partner:

MATHEMA

JET BRAINS

heise Developer

Im Geflecht der Dienste

Softwareentwicklung unterliegt einem steten Wandel. Wichtige aktuelle Trends wie Microservices und ML gehören ebenso dazu wie fast zeitlose Themen rund um sichere Programmierung und die Suche nach der geeigneten Programmiersprache.

Zeitgemäße Softwareentwicklung ist ohne das rasant wachsende Cloud-native-Ökosystem – also Anwendungen und Infrastruktur, die originär für den Betrieb in der Cloud geschaffen sind, – kaum noch denkbar. Rund um die zentrale Orchestrierungsplattform Kubernetes lassen sich Cloud-native Applikationen in flexibel maßgeschneiderte Microservices aufteilen und in Container verpackt dynamisch skalierbar in die Cloud ausrollen und ausführen.

Die plattformübergreifende Flexibilität und Skalierbarkeit kommt aber nicht ohne eine neue Komplexität daher: Die Partitur an Microservices flüssig und im Takt zu orchestrieren, verlangt nicht nur optimales Konfigurieren, sondern auch ein gehöriges Maß an Know-how. Sonst gerät die Continuous Delivery Pipeline schnell ins Stocken, wenn die Infrastruktur nicht mitspielt.

Als Rundum-sorglos-Paket für Microservices stellen Service-Meshes Entwicklern eine weitere Kontroll- und Monitoringebene zur Verfügung, die wichtige Aspekte wie Observability, Routing, Resilienz und Sicherheit leichter beherrschbar macht. Der rasche Aufstieg von Istio, Linkerd und vergleichbaren Werkzeugen hat 2019 zum Jahr der Service-Meshes avancieren lassen.

Wem die Last der Infrastrukturverwaltung indes zu schwer wiegt, der geht einen Schritt weiter und gibt die Verantwortung an den Plattformbetreiber für Serverless Computing ab. Als ereignisgesteuerte Architektur verspricht Function as a Service (FaaS) eine dynamisch skalierende, selbstorganisierende Komposition kurzlebiger Funktionsaufrufe anstelle der eher statischen Orchestrierung klassischer, als PaaS ausgelegter Microservices. Im Zweifelsfall lässt sich aber auch mit Tools wie Kubeless eine Brücke zwischen diesen Welten schlagen.

Vor der Qual der Wahl, das am besten geeignete Werkzeug für die angepeilte Anwendung zu finden, stehen Entwickler allerdings nicht nur im Hinblick auf Plattform, Architektur und Infrastruktur. Auch die Programmiersprache sollte dem Zweck angemessen sein. Zwischen universell tauglichen und hochspezialisierten Sprachen finden sich zahlreiche Abstufungen. Während sich C/C++ und Java in der klassischen Anwendungsentwicklung über Jahrzehnte einen festen Platz erobert haben, legen die ebenfalls vielfältig einsetzbaren Sprachen Python und Go in der Entwicklergunst deutlich zu. Beide folgen einem ähnlich pragmatischen Ansatz: Sie sind leicht zu erlernen und versprechen eine schnelle Umsetzung – ohne komplexe Probleme wie Nebenläufigkeit auszuklammern.

Geht es konkret darum, verteilte (Web-)Anwendungen robust und skalierbar zu gestalten, empfehlen sich eher spezialisierte Sprachen wie Elm und Elixir – oder auch Rust, das grundlegend auf Sicherheit ausgelegt ist. Denn insbesondere Security rückt auch immer stärker in den Fokus und spielt mittlerweile eine zentrale Rolle im gesamten Lebenszyklus der Anwendungsentwicklung, wie der Begriff DevSecOps eindrücklich deutlich macht. Ein ganzheitlicher Ansatz muss vom sauberen Code über die geeigneten Tools und Frameworks bis zum sicheren Betrieb der Anwendung reichen – und auch Prozesse sowie organisatorische Strukturen im wachsenden Geflecht der Dienste stets neu hinterfragen.

Beim Hypethema Machine Learning, das von Fortschritten bei Frameworks wie TensorFlow 2 oder auch neuen Möglichkeiten des ML-Einsatzes im Browser befeuert wird, kommen neben der Sicherheit auch zunehmend Fragen des Datenschutzes und der Ethik in die Diskussion. Die moderne Softwareentwicklung ist also auf allen Ebenen nicht nur von spannenden Trends geprägt, sondern fordert alle Beteiligten mit ihrem teilweise rasanten Wandel heraus, sich neuen Herausforderungen zu stellen.

Mit diesem Sonderheft möchten wir Ihnen die Orientierung im Dschungel der Softwareentwicklung vereinfachen. Wir hoffen, dass Ihnen der breite Themenmix und die umfangreichen Praxisartikel einen besseren Überblick verschaffen und vielfältige Anregungen liefern. *iX* und *heise Developer* wünschen Ihnen viel Spaß beim Lesen!

MATTHIAS PARBEL

Sprachen

Entwickler haben bei der Auswahl an Programmiersprachen die Qual der Wahl. Von einfach bis komplex, von universell bis hochspezialisiert reicht das Spektrum. Für moderne verteilte Anwendungen sollten sie außerdem sicher, skalierbar und robust sein.

ab Seite 7

Cloud-native

Mit Kubernetes, Service-Meshes und anderen hilfreichen Tools sind Entwickler auch im Cloud-nativen Zeitalter bestens gerüstet, um altbewährte Prinzipien der Continuous Delivery noch besser einzusetzen und sich von Chaos Engineering inspirieren zu lassen.

ab Seite 49

Sprachen

Beliebtheitssieger
Einstieg in Googles Programmiersprache Go 8

C++-Alternative
Programmieren in D 16

Rust
Sichere Programmiersprache für systemnahe und parallele Software 20

Koroutinen mit Kotlin
Verteilte Systeme: weniger warten, asynchron arbeiten 26

Elixir, Elm und Phoenix
Webanwendungen sicher entwickeln und ohne Laufzeitfehler ausführen 30

Umstieg organisieren
End of Life für Python 2 40

Objektorientierte Programmierung
Smalltalk: Mehr als eine Programmiersprache 44

Cloud-native

Architektur
Continuous Delivery – regelmäßige Releases, höhere Produktivität 50

Developer Tools
Cloud-nativ entwickeln mit Kubernetes 54

Service-Meshes
Brauchen asynchrone Microservices und Self-Contained Systems ein Service-Mesh? 58

Istio: Das Service-Mesh für verteilte Systeme und Microservices 62

Linkerd: Ein Service-Mesh für Kubernetes 66

Function as a Service
Serverless Computing – Theorie und Praxis 72

Kubeless
FaaS auf Kubernetes betreiben 80

Qualitätssicherung
Grundlagen des Chaos Engineering 84

Praxis

Die Anforderungen zeitgemäßer Softwareentwicklung sind geprägt von Schlagwörtern wie DevSecOps, Microservices und GraphQL. Aber auch die stete Auseinandersetzung mit neuen Programmiersprachen sowie die Qualitätssicherung ihres Codes sollten Entwicklungsteams nicht aus den Augen verlieren.

ab Seite 91

Machine Learning

Zur rasanten Entwicklung im Bereich Machine Learning trägt nicht nur das vollständig überarbeitete Framework TensorFlow 2 bei, auch im Browser laufen ML-Anwendungen bereits brauchbar. Auf dem Weg vom Prototyp zur Produktion hilft zudem das Austauschformat ONNX, während Datenschutzanforderungen neue Hürden aufbauen.

ab Seite 135

Praxis

Experten im Gespräch I
Refactoring: Interview mit Martin Fowler 92

Sicherheit
DevSecOps – der nächste Hype nach DevOps und Containern? 94

Moderne Architekturen
Microservices mit Kotlin, Vert.x und OpenAPI 104

Abfragesprache
Eine GraphQL-Applikation mit Java umsetzen 112

Migration
Wie Wire von JavaScript zu TypeScript wechselte 123

Versionsverwaltung
Die vielfältigen und unbekannten Fähigkeiten von Git 126

Experten im Gespräch II
Clean Code: Interview mit Uncle Bob Martin 132

Machine Learning

ML-Framework
TensorFlow 2: Aufgeräumte APIs und neue Namespaces 136

JavaScript
Machine Learning im Browser mit TensorFlow.js 140

Austauschformat
Portabilität für Deep-Learning-Modelle mit ONNX 146

KI und Datenschutz
DSGVO für ML-Anwendungen 151

Sonstiges

Editorial 3
Impressum 145

iX Workshop Workshops 2020

Webanwendungen entwickeln mit React

14. – 17. Januar 2020
in Hannover

Künstliche Intelligenz mit Deep Learning

2-Tage-Kurs zum schnellen Einstieg

27. – 28. Januar 2020
in Hannover

Docker-Container: Administration und Orchestrierung

21. – 24. Januar 2020
in Hannover

Sichere Cloud-Nutzung

10. – 12. Februar 2020
in Hannover

Ihr Weg in die Cloud

03. – 05. Februar 2020
in Hannover

Weitere Infos unter:
http://www.heise-events.de/workshops

Sprachen

Das Spektrum an Programmiersprachen ist so vielfältig wie die Einsatzgebiete, für die sie optimiert sind. Durch die Synthese von Objektprogrammierung, Funktionen und Typen ist Smalltalk schon in den 1970er Jahren angetreten, die komplexe Softwareentwicklung zu vereinfachen. Ein vergleichbares Ziel verfolgten die Python-Macher rund um Guido van Rossum. Sprachen wie Rust, Elixir oder Elm hingegen zielen eher darauf ab, auch verteilte Anwendungen möglichst sicher, skalierbar und robust zu gestalten. Während D den Spagat einer Multiparadigmensprache wagt, verfolgt Googles Go einen pragmatischen Ansatz: die Sprache bleibt kompakt und einfach, macht aber auch Komplexität wie Nebenläufigkeit und Multi-Core-Programmierung beherrschbar.

Einstieg in Googles Programmiersprache Go	**8**
Programmieren in D	**16**
Rust als sichere Programmiersprache für systemnahe und parallele Software	**20**
Koroutinen: Weniger warten, asynchron arbeiten	**26**
Skalierbare, robuste Webanwendungen mit Elixir, Elm und Phoenix	**30**
End of Life für Python 2	**40**
Smalltalk: Mehr als eine Programmiersprache	**44**

Einstieg in Googles Programmiersprache

Auf die Plätze, fertig, Go!

Christoph Iserlohn, Daniel Bornkessel

Go ist eine junge Prorgrammiersprache. Trotzdem ist sie bereits in zahlreichen Projekten im Einsatz, die die Eckpfeiler zeitgemäßer Infrastrukturen bilden. Nicht nur deshalb ist es lohnenswert, sich mit Go auseinanderzusetzen.

Go ist für eine Programmiersprache noch recht jung. Die Veröffentlichung der ersten stabilen Version 1.0 ist etwas mehr als sechs Jahre her. Seitdem hat Go rasant an Popularität gewonnen. Zweimal war sie „Programming Language of the Year“ bei TIOBE – 2009 und 2016. Vor allem bei systemnaher Infrastruktursoftware ist Go nicht mehr wegzudenken: Docker, Kubernetes, etcd, Consul, NATS, Packer, Prometheus, Traefik und Vault sind alle in Go geschrieben. Alles prominente Projekte, die heutzutage für den Betrieb einer zeitgemäßen (Cloud-)Infrastruktur unverzichtbar sind.

Auf den ersten Blick erscheint der Erfolg etwas merkwürdig. Denn Go sticht nicht gerade durch Innovationen hervor. Im Gegenteil: Go lässt viele Dinge vermissen, die bei einer Programmiersprache selbstverständlich scheinen: Exceptions, ein Typsystem mit Typhierarchie und generischen Typen, kovariante Funktions-, Methoden- und Rückgabeparameter oder einen Option-Typ zur Vermeidung des „billion-dollar mistake“ (Null-Reference). Trotz und manchmal auch gerade wegen der Einschränkungen lassen sich mit Go etliche Aufgaben des Programmieralltags gut und schnell umsetzen.

Die Ursprünge von Go

Go entstand 2007 ursprünglich bei Google, ist aber seit 2009 – weit vor Version 1.0 – ein Open-Source-Projekt auf GitHub. Gründungsväter sind unter anderem die Bell-Labs- / UNIX- / C-Berühmtheiten Rob Pike und Ken Thompson. Der Google-Ursprung ist klar erkennbar. Go soll Herausforderungen stemmen, die vor allem Google hat: Skalierung auf allen Ebenen, auf viele Prozessoren beziehungsweise Cores, auf große Codebasen, auf große Entwicklerteams sowie auf viele und schnelle Deployments.

Die Handschrift der Gründungsväter ist ebenfalls unverkennbar: Große Ähnlichkeiten mit C, nicht nur auf der Syntaxebene, sondern auch beim Abstraktionsniveau und den Designzielen. Ebenso die zahlreichen Anleihen bei Plan9, einem Betriebssystem, das Pike und Thompson bei Bell Labs entwickelt hatten. Der Artikel beschreibt im Folgenden einige der hervorstechenden Merkmale von Go, die trotz des Verzichts auf Innovation Go von anderen Programmiersprachen abhebt.

Listing 1: Ausführen von Funktionen

```
// Funktionen werden mit dem Schlüsselwort func deklariert
func print(s string) {
  fmt.Println(s)
}

// Die Funktion main ist der Einstiegspunkt in jedes Go-Programm
func main() {
  print("one")
  // print("two") wird in einer Go-Routine ausgeführt
  go print("two")
  print("three")
  time.Sleep(1 * time.Second)
}
```

Nebenläufigkeit in Go

Go hat Nebenläufigkeit beziehungsweise Concurrency auf Sprachebene umgesetzt und sich an Hoares Communicating Sequential Processes orientiert. Die beiden Sprachkonstrukte zur Nutzung von Nebenläufigkeit sind Go-Routinen und Channels. Go-Routinen sind Funktionen oder Methoden, die nebenläufig ausgeführt werden – sie blockieren also nicht die Ausführung des aktuellen Codes so lange, bis die Funktion ihre Berechnungen beendet hat. In Go können Entwickler jede Funktion oder Methode als Go-Routine ausführen, indem sie sie mit dem Schlüsselwort `go` aufrufen (Listing 1).

Der Code aus Listing 1 erzeugt folgende Ausgabe:

```
one
three
two
```

da der zweite Aufruf als Go-Routine nicht blockiert. Der Aufruf von `time.Sleep` am Ende ist notwendig, weil sich das Programm sonst vorm Ausführen beenden würde – ein Go-Programm wartet vor dem Beenden nicht automatisch, bis alle Go-Routinen beendet sind.

Ruft man Funktionen oder Methoden mit Rückgabewerten als Go-Routinen auf, gibt es keine Möglichkeit, an die Rückgabewerte zu gelangen. Aus anderen Sprachen bekannten Konzepte wie Promises oder Futures gibt es in Go nicht.

Listing 2: Ungepufferte Channel

```
func greeting(channel chan string) {
  // schreibe "hello world" in den channel
  channel <- "hello world"
}

func main() {
  // erzeuge einen channel vom Typ string
  channel := make(chan string)
  go greeting(channel)

  var text string = <-channel
  fmt.Println(text)
}
```

Channels

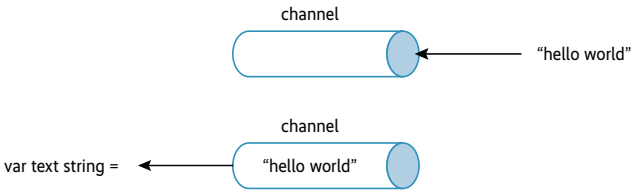
Die Strategie in Go lautet: „Don't communicate by sharing memory; share memory by communicating." Es sollen nicht mehrere nebenläufige Prozesse mit derselben globalen Variablen arbeiten, deren Zugriff zum Beispiel Mutexe oder Semaphoren steuern, sondern Prozesse sollen untereinander durch sogenannte Channels kommunizieren. Channels dienen zum Austausch der Referenzen auf Daten.

Entwickler können Channels wie normale Variablen nutzen und als Funktionsparameter übergeben oder als Rückgabewert zurückliefern. Die einfachste Form eines Channels, der ungepufferte Channel, ist eine Warteschlange nach dem FIFO-Prinzip (First In, First Out). Dabei ist zu beachten, dass das Schreiben in einem ungepufferten Channel so lange blockiert, bis jemand von ihm liest. Umgekehrt blockiert das Lesen eines Channels so lange, bis jemand in den Channel schreibt (Listing 2).

Wie im Beispiel sichtbar, müssen Anwender Channels mit dem Schlüsselwort `make(...)` erstellen. Sie haben außerdem einen Typ, der hier string ist. Die Funktion `greeting` erhält einen Channel, in den sie einen String schreibt. Die Channel-Syntax kann man sich bildlich vorstellen, wenn man einen Channel wie eine Röhre darstellt (Abb. 1).

Möchte man in einen Channel schreiben, zeigt der Pfeil vom Wert auf die Channel-Variable. Möchte man hingegen aus dem Channel lesen, zeigt der Pfeil von der Channel-Variable zu der Variable, die den Wert speichern soll. Da Lese- und Schreiboperationen blockieren, muss das Programm `greeting(channel)` zwingend als Go-Routine aufrufen, damit es nicht in eine Deadlock-Situation läuft. `channel <- "hello world"` würde bei einem synchronen Aufruf so lange blockieren, bis jemand vom Channel liest: Der Lesebefehl wäre nie zu erreichen. Channels verwenden Entwickler deshalb häufig in Kombination mit Go-Routinen. Ein Channel kann mehrere Go-Routinen miteinander verbinden (Listing 3).

Das Beispiel übergibt beiden Go-Routinen denselben Channel, über den sie kommunizieren können. Außerdem schließt der Producer den Channel am Ende, was die Go-Routine `print` durch `s, ok = <- text` abfragen kann: Wenn der zweite Wert `(ok) false` zurückliefert, ist der Channel geschlossen. Die `main`-Methode blockiert so lange, bis man in den `done`-Channel schreibt – er dient



Der Channel als Röhre (Abb. 1)

Listing 3: Verknüpfung von Go-Routinen

```
func producer(text chan string) {
    text <- "one"
    text <- "two"
    text <- "three"
    close(text)
}

func print(text chan string, done chan bool) {
    var s string
    // eine for-Schleife in Go braucht keine Klammern um den
    // Schleifenkopf mit Initialisierung, Test und Fortsetzung
    for ok := true; ok; s, ok = <-text {
        fmt.Println(s)
    }

    done <- true
}

func main() {
    channel := make(chan string)
    done := make(chan bool)

    go producer(channel)
    go print(channel, done)

    <-done
}
```

Listing 4: Parallele Go-Routinen

```
func producer(text chan string, s string) {
  for i := 0; i < 10; i++ {
    text <- s
  }
}

func main() {
  channel := make(chan string)

  go producer(channel, "-")
  go producer(channel, "|")

  for i := 0; i < 20; i++ {
    fmt.Print(<-channel)
  }
}
```

nur zur Synchronisation. Der Wert im Channel interessiert nicht und ist keiner Variable zugeordnet. Channels sind sicher für die parallele Verwendung. Es dürfen mehrere Go-Routinen in denselben Channel schreiben, ohne dass Daten verloren gehen. Das gezeigte Programm in Listing 4 erzeugt Ausgaben wie `||-----|||-----|||||` oder `||-||||||||---------`. Es gehen keine Zeichen verloren, aber die Reihenfolge ist zufällig, je nachdem welche Go-Routine zuerst einen Wert in den Channel schreiben kann.

Mit dem `select`-Statement ist es möglich, auf die erste Antwort von beliebig vielen Channels zu warten (Listing 5). Das Beispiel gibt entweder `message: one` oder `message: two` aus, je nachdem, welcher `print`-Aufruf zuerst seinen Channel beschreibt.

Gepufferte Channels

Die bisher gezeigten Channels hatten keinen Puffer, das heißt Lese- und Schreiboperationen sind genau abzustimmen, damit keine Deadlock-Situation entsteht. Gepufferte Channels können eine bestimmte Anzahl von Werten aufnehmen, bevor Schreiboperationen sie blockieren (Listing 6).

Gepufferte Channels sind nützlich, um Optimierungen vorzunehmen oder Produzenten und Konsumenten des Channels zeitlich zu entkoppeln. Eine Go-Routine, die einen gepufferten Chan-

Listing 5: Das select-Statement

```
func print(c chan string, msg string) {
    time.Sleep(time.Duration(rand.Intn(50)))
    c <- "message: " + msg
}

func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    go print(c1, "one")
    go print(c2, "two")

    select {
    case s1 := <-c1: fmt.Println(s1)
    case s2 := <-c2: fmt.Println(s2)
    }
}
```

nel befüllt, braucht nicht auf das Lesen eines Ergebnisses zu warten, bevor potenziell langwierige Berechnungen für den nächsten Wert starten können. Das ist für verschiedene Problemstellungen nützlich oder sogar notwendig.

Objektorientierung ohne Klassen und Vererbung

Objektorientierung dient in der Softwareentwicklung häufig zur Strukturierung und Modellierung. Obwohl es in Go weder Klassen noch Vererbung gibt, muss man nicht auf typische objektorientierte Konzepte wie Datenkapselung oder Polymorphie verzichten. Statt Methoden innerhalb von Klassen zu definieren, erfolgt das in Go analog zu Funktionen, aber mit einem zusätzlichen `receiver`-Parameter. Er folgt nach dem Schlüsselwort `func`

Listing 6: Gepuffterte Channels

```
// erzeuge einen channel mit der Puffergröße 1
c := make(chan string, 1)
c <- "text"
fmt.Println(<-c)
funktioniert, da die Schreiboperation nicht blockiert, wohingegen

c := make(chan string, 0) // entspricht "make(chan string)"
c <- "text"
fmt.Println(<-c)         // wird nie erreicht
in eine Deadlock-Situation läuft.
```

Listing 7: Definition einer Methode

```
// Die Deklaration des Package erfolgt immer am Anfang
package staff

type Clerk struct {
    Name string
    Age int
    salary int
}

// Methodendefinition auf dem Typ *Clerk
func (clerk *Clerk) ChangeSalary(amount int) {
    clerk.salary += amount
}

c := Clerk{"Max Mustermann, 30, 40000}

// Mit der Punkt-Notation können dann die Methoden auf den
// jeweiligen Datentypen
// aufgerufen werden, genau wie auf die einzelnen Felder eines
// structs per Punkt-Notation zugegriffen wird.

c.ChangeSalary(100)
```

und gibt den Datentyp an, auf dem die Methode definiert wird. Eine Methode muss zum gleichen Package gehören (Listing 7).

Im Beispiel ist die Methode auf einem Pointer vom Typ Clerk definiert anstatt direkt auf dem Struct `Clerk`. Das ist wichtig, denn sie modizifizert das `clerk`-Struct. Wäre die Methode direkt auf Clerk definiert, hätte sie eine Call-by-Value-Semantik. Sie operiert dann auf einer Kopie des `receiver`-Parameters und `c.ChangeSalary(100)` hätte keinen sichtbaren Effekt, weil nur die Kopie innerhalb der Methode modifiziert ist – `c.salary` wäre nach dem Methodenaufruf immer noch 40.000.

Methoden können Datentypen und die zugehörigen Operationen bündeln und lassen sich wie Objekte benutzen. Zur Datenkapselung gehört noch das Verstecken der Interna, das heißt im obigen Fall sollen die Felder des Clerk Structs nicht direkt zu modifizieren sein, sondern nur über Methoden. Dazu können Entwickler in Go die Sichtbarkeit von Typen, Funktionen, Methoden, Variablen und Konstanten festlegen.

Die Sichtbarkeit bezieht sich immer auf ein Package und ist durch die Groß-/Kleinschreibung festgelegt. Fangen Typen, Funktionen, Methoden, Variablen oder Konstanten mit einem Kleinbuchstaben an, sind sie nur innerhalb des Packages sichtbar. Fangen sie mit einem Großbuchstaben an, sind sie exportiert und außerhalb des Packages sichtbar. Zu welchem Package Typen, Funktionen, Methoden, Variablen und Konstanten gehören, legt die Package-Deklaration am Anfang der Quelltextdatei fest. Ein Package kann aus mehreren Dateien bestehen.

Bei Structs können Anwender die Sichtbarkeit jedes einzelnen Felds kontrollieren. Im Beispiel sind die Felder Name und Age auch außerhalb des Package `staff` sichtbar, das Feld `salary` nicht. Der Zugriff außerhalb von `staff` kann nur über die exportierte Methode `ChangeSalary(amount int)` erfolgen.

Komposition

Vererbung dient zum Wiederverwenden vorhandener Objektdefinitionen, meistens in der Form von Klassen. Entwickler können die Funktionen ergänzen und/oder modifizieren und müssen sie nicht neu implementieren. Als Alternative bietet sich Komposition an, die komplexe Objekte durch eine Kombination von einfachen Objekten bildet. Go unterstützt Komposition über das sogenannte Embedding direkt. Dies bettet einen Typ in einen anderen Typ ein und dessen Methoden sind per Promotion direkt auf dem einbettenden Typ aufrufbar (Listing 8).

Polymorphie

Go unterstützt Polymorphie bei Parametern und Rückgabewerten von Funktionen und Methoden sowie bei eingebetteten Datentypen mit Interfaces. Ein Interface ist eine definierte Menge von Methoden (s. Listing 8).

Im Unterschied zu Programmiersprachen wie Java muss ein Typ nicht explizit deklarieren, dass er ein Interface implementiert. Es reicht, wenn alle Methoden des Interfaces implementiert sind. Im Listing 8 implementiert der Typ `*Clerk` und der Typ `*Manager` das Interface `PaidEmployee`. Interface-Typen entscheiden über die aufzurufende Methode erst zur Laufzeit (dynamic binding):

```
func RaiseSalaries(e []PaidEmployee) {
    for i := 0; i < len(e); i++ {
            // welche ChangeSalary Methode aufgerufen
            // wird, entscheidet sich erst zur Laufzeit
        e.ChangeSalary(-100)
    }
}
```

Listing 8: Komposition via Embedding

```
type Address struct {
    street string
    zipcode string
    city string
}

// der Receiver wird nicht modifiziert,
// daher wird kein Pointer benötigt
func (a Address) humanReadableAddress() string {
    return a.zipcode + a.city + a.street
}

type Clerk struct {
    // ein Feld ohne Bezeichner ist embedded
    Address
    Name string
    Age int
    salary int
}

a := Address{"Karl-Wiechert-Allee 10", "30625 ", "Hannover"}
c := Clerk{a, "Max Mustermann", 30, 4000}

// Die Methoden des eingebetteten Typs sind direkt auf dem
// einbettenden Typ aufrufbar
c.humanReadableAddress()
```

Interface-Typen lassen sich mit Embedding kombinieren, entweder um ein Interface aus mehreren Interface-Typen zusammen zu setzen oder um als Typ in einen anderen eingebettet zu werden (Listing 9).

Eine Besonderheit stellt das leere `interface{}` dar. Es enthält keine Methoden: Jeder Typ implementiert es. Nützlich ist es vor allem, wenn die Struktur eines Typs beim Kompilieren noch nicht bekannt ist, beispielsweise beim Encoding oder Decoding von XML oder JSON. Mit der Kombination von Methoden, Komposition und Interface-Typen lässt sich genausogut objektorientiert programmieren wie mit Klassen und Vererbung. Die Unterschiede sind eher syntaktischer Natur. Methoden müssen im Quelltext nicht mehr zusammen mit ihren Typen definiert werden, wie es bei Klassen der Fall ist. Das hat den Vorteil, dass Entwickler einem Typ nachträglich Methoden hinzufügen können, ohne zwingend den Quelltext des Typs verändern zu müssen.

Ähnlich verhält es sich mit Interface-Typen. Nutzer können sie auch nachträglich einführen, ohne den Quelltext des implementierenden Typs verändern zu müssen, weil eine explizite Deklaration nicht nötig ist. Gerade bei größeren Entwicklungsteams und einer sich schnell ändernden Codebasis ist das von Vorteil, da die Vorgehensweise viele Merges und somit Merge-Konflikte vermeiden kann.

Listing 8: Ein Beispiel-Interface

```
type PaidEmployee interface {
    ChangeSalary(amount int)
}

type Manager struct {
    salary int
}

func (m *Manager) ChangeSalary(amount int) {
    if amount < 0 {
            amount *= -1
    }
    m.salary += amount
}
```

Listing 9: Zusammensetzung von Interface-Typen

```
// Interface aus mehreren Interface-Typen zusammengesetzt
type Payroll interface {
    Address
    PaidEmployee
}

type worker struct {
    // Interface als eingebetteter Typ
    PaidEmployee
    Adress
}
```

Orientierung an C

Go orientiert sich sprachlich von der Komplexität eher an C als an C++, ohne jedoch auf die Möglichkeit objektorientierter Programmierung zu verzichten, die bei Go eher an Structs mit Funktions-Pointern als an Klassen mit Konstruktoren und Mehrfachvererbung erinnert. Die Lernkurve ist für Neueinsteiger deshalb wesentlich flacher und die Zeit bis zum ersten produktiv einsetzbaren Code kürzer.

Auf neue Konzepte verzichtet die Programmiersprache, sofern sie nicht absolut nötig sind. Nebenläufigkeit als solch ein kritisches Feature zu sehen ergibt Sinn, da Anwendungen die Fortschritte zeitgemäßer Mehrkern-Prozessoren nur gut nutzen können, wenn Code nebenläufig ausgeführt wird. Go-Routinen und Channels sind als Konzepte einfach verständlich und daher schnell erlernbar. Mit der Strategie, Go schlank zu halten und neue Konzepte konservativ einzuführen, erreichen die Go-Autoren ihr definiertes Ziel ziemlich gut: Go leicht erlernbar und vielseitig einsetzbar machen.

Error-Handling in Go

Produktionsreife Programme bestehen oft zu einem großen Teil aus Fehlerbehandlung, unabhängig von der Programmiersprache. Go nutzt dazu Multivalue Returns, Early Returns und `defer`-Statements. Mit Multivalue Returns ist es leicht, einen Fehler in einer Funktion oder Methode zu signalisieren. Neben den eigentlichen Rückgabewerten gibt Go immer noch einen weiteren Wert zurück, der das Interface error implementiert. Das hatte bis Version 1.13 nur die Methode Error(), die einen Fehler-String zurückgab. Ab dann unterstützt Go außerdem noch das Wrappen des ursprünglichen Fehlers, was die Fehlerbehandlung ein wenig komfortabler gestaltet. Ist er nil, gab es keinen Fehler bei der Ausführung der Funktion beziehungsweise Methode, und Rückgabewerte sind bedenkenlos einsetzbar. Sollte er jedoch nicht nil sein, müssen Entwickler den Fehler beheben und die Rückgabewerte als invalide behandeln.

Operationen auf Dateien sind typischerweise anfällig für allerlei Fehler – Dateien können zum Beispiel nicht vorhanden, zu öffnen oder beschreibbar sein. All das muss man bei der Programmierung berücksichtigen (Listing 10).

Der Code hat noch ein schwerwiegendes Problem. Wenn beim Schreiben in die Datei ein Fehler auftritt, verlässt es die Funktion beziehungsweise Methode vorzeitig im `if`-Block (early return). Die Zeile, die die Datei schließt (`file.Close()`), wird nicht mehr erreicht und es kommt zu einem Ressourcen-Leak. Genau deshalb sind Early Returns in Sprachen ohne automatisches Speichermanagement und try-catch-finally-Konstrukte verpönt. In C nutzt man ein goto, um zum Ende der Funktion zu springen und dort die Ressourcen aufzuräumen und auch nur dort die Funktion zu verlassen (one-point return). In Java wäre die Fehlerbehandlung in einem try-catch-Block angesiedelt, während ein finally-Block die Ressourcen aufräumt. Ein Early Return wäre möglich, aber es ist umstritten, ob das der richtige Stil ist.

Idiomatische Early Returns

In Go sind Early Returns idiomatisch, das heißt, man benötigt einen Weg, um Ressourcen wieder aufzuräumen. Im Beispiel könnte das noch von Hand im `if`-Block passieren, aber sobald mehrere Ressourcen im Spiel sind, wird es schnell unübersichtlich. Deshalb gibt es in Go das `defer`-Statement. Es kann Funktionen oder Methoden nach einem return-Statement ausführen (Listing 11)

Nach dem erfolgreichen Öffnen der Datei merkt `defer file.Close()` das Schließen der Datei vor. `defer` sorgt dafür, dass `file.Close()` beim Verlassen der Funktion mit return ausführt, egal mit welchem return man die Funktion verlässt. Sollten mehrere Aufräumaktionen notwendig sein, können Entwickler weitere `defers` benutzen. Beim Verlassen der Funktion erfolgt die Abarbeitung nach dem Prinzip Last-In-First-Out.

Im Gegensatz zu einer Fehlerbehandlung mit Exceptions ist die Fehlerbehandlung per Rückgabewert repetitiv. Nutzer können aber sofort sehen, wo ein Fehler auftreten kann, und es existiert keine Möglichkeit für einen versteckten Kontrollfluss, wie es bei Exceptions der Fall ist. Mit defer und Early Returns stehen darüber hinaus zusammenhängende Teile immer dicht beisammen und nicht über den Code verteilt. Im Beispiel sind das Öffnen, die eventuell benötigte Fehlerbehandlung und das Schließen der Datei direkt untereinander positioniert. Dadurch ist der Code insgesamt leicht lesbar.

Listing 10: Operationen auf Dateien

```
// versuche eine Datei zu öffnen
// im Erfolgsfall ist file ein Dateihandle und err ist nil
file, err := os.OpenFile(filename, os.O_RDWR, 0755)
// prüfe, ob das Öffnen erfolgreich war
if err != nil {
    // das Öffnen war nicht erfolgreich
    // der normale Programmfluss kann nicht fortgesetzt werden
    return err
}
// das Öffnen war erfolgreich
// file kann genutzt werden, um zum Beispiel in die Datei zu schreiben
n, err := file.WriteString("Hello Go!")
// prüfe, ob das Schreiben erfolgreich war
if err != nil {
    return err
}
fmt.Println("Wrote", n, "bytes to file:", filename)
file.Close()
```

Batterien inklusive

Der Erfolg einer Programmiersprache ist nicht nur von ihren Features abhängig. Maßgeblich beteiligt sind auch die verfügbaren Bibliotheken. Schließlich möchten Entwickler im Programmieralltag nicht jedes Rad neu erfinden. Go verfolgt, ähnlich wie Python, den Kurs der „Batteries included". Das heißt, die Standardbibliothek bringt von Haus aus reichlich Funktionen mit und enthält unter anderem Packages für XML und JSON, HTML- und Text-Templates, über produktionsreife HTTP-Server/Client-Bibliotheken (inklusive HTTP 2.0) bis hin zur Kryptografie.

Gerade Letzteres fehlt bei vielen Programmiersprachen in den Standardbibliotheken, weshalb Anwender häufig auf C-Bibliotheken wie OpenSSL oder libsodium zurückgreifen. Das wiederum verursacht Probleme beim Build und Deployment: Auf dem Build-System braucht es die passende Compiler-Infrastruktur. Bibliotheken und deren Abhängigkeiten müssen in den richtigen Versionen vorhanden sein, sowohl auf dem Build- als auch auf dem Deployment-System.

Es gibt außerdem ein reichhaltiges Ökosystem von Drittanbieter-Bibliotheken, sollte eine Funktion nicht in der Standardbibliothek vorhanden sein. Das Einbinden von C-Bibliotheken ist leicht und über das syscall-Package der Standardbibliothek können Entwickler direkt Systemaufrufe ausführen.

Deployment: schnelle Kompilierung

Laut Rob Pike, einem der Go-Väter, sind die ersten Ideen zu Go bei einem längeren Kompiliervorgang von C++ entstanden. Ziel war es, die Schmerzen mit C++, wie die langen Kompilierzeiten, zu beseitigen. Go kompiliert in der Tat schnell. Auch große Projekte lassen sich meist in wenigen Sekunden kompilieren. Dazu kommt, dass Go-Kompilate statisch gelinkte Binaries sind, die ohne weitere Abhängigkeiten lauffähig sind. Das mühsame und oft fragile Aufsetzen von Abhängigkeiten auf dem Deployment-Ziel entfällt. Ein erfreulicher Nebeneffekt ist das einfache Kompilieren für andere Betriebssysteme und Prozessorarchitekturen. Entwickler müssen nur die passenden Umgebungsvariablen setzen:

```
# Cross-compiling von Windows oder macOS für den Raspberry Pi:
$ GOOS=linux GOARCH=arm GOARM=6 go build -o example
```

Die unschönen Seiten von Go

Neben den vielen Vorteilen von Go gibt es einige Dinge, die nicht gut gelöst sind. Herauszustellen sind vor allem: fehlende generi-

sche Datentypen und das Dependency-Management. Generische Datentypen erlauben parametrische Polymorphie. Das ist gerade bei statisch typisierten Programmiersprachen wünschenswert. Viele Algorithmen und Container-Datentypen müssen Entwickler nur einmal programmieren und dann über den Typ parametrisieren. Ansonsten müsste man sie für jeden Typ ausprogrammieren.

Alle ernstzunehmenden statisch typisierten Programmiersprachen der letzten Jahre unterstützen generische Datentypen – außer Go. Zwar sind die eingebauten Container-Datentypen generisch und über das Verwenden von Interface-Typen lässt sich Codeduplizierung vermeiden, trotzdem fühlt man sich oft in die Zeiten von Java vor Version 1.5 versetzt. Mit dem Unterschied, dass Go das leere Interface statt wie in Java den Typ Object nutzt. In den konkreten Typ wird dann per Cast manuell umgewandelt: ein aufwendiges und fehleranfälliges Verfahren.

Den Kritikpunkt, den viele oft und regelmäßig noch vor Version 1.0 artikulierten, soll Go 2 beheben. Ein Proposal für die Einführung generischer Datentypen liegt vor. Wann Go 2, das nicht mehr mit Go 1.x kompatibel sein wird, erscheint und wie die generischen Datentypen letztlich umgesetzt werden, steht indes noch nicht fest.

Dependency-Management

Der zweite große Kritikpunkt, der ebenfalls von Anfang an besteht, ist das eigenwillige Dependency-Management. Es ist zwar seit jeher möglich gewesen, Dependencies ganz einfach herunterzuladen, auch werden dabei Git und GitHub sowie andere

Listing 11: Das defer-Statement

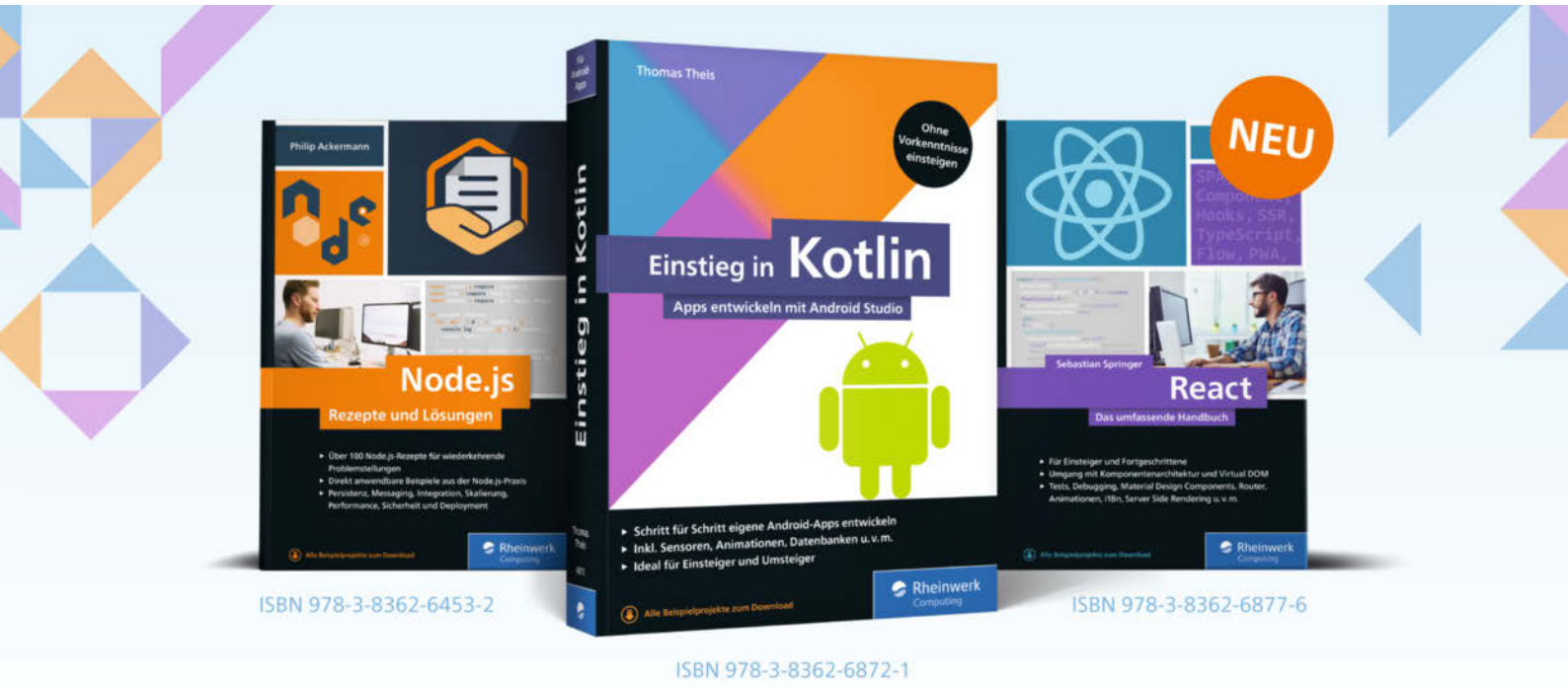
```
// versuche, eine Datei zu öffnen
// im Erfolgsfall ist file ein Dateihandle und err ist nil
file, err := os.OpenFile(filename, os.O_RDWR, 0755)
// prüfe, ob das Öffnen erfolgreich war
if err != nil {
    // das Öffnen war nicht erfolgreich
    // der normale Programmfluss kann nicht fortgesetzt werden
    return
}
// das Öffnen war erfolgreich
// file kann genutzt werden, um zum Beispiel in die Datei zu schreiben
// Schließe die Datei beim Verlassen der Funktion
defer file.Close()
n, err := file.WriteString("Hello Go!")
// prüfe, ob das Schreiben erfolgreich war
if err != nil {
    return
}
fmt.Println("Wrote", n, "bytes to file:", filename)
```

Versionskontrollsysteme und Repository-Hoster unterstützt. Ein Beispiel:

```
$ go get github.com/golang/example/hello
```

Bis Version 1.11 war das Problem allerdings, dass es keine Möglichkeit gibt, eine bestimmte Version einer Abhängigkeit anzugeben. Go lud immer die aktuelle Version. Inkompatibilitäten waren vorprogrammiert und reproduzierbare Builds unmöglich. Das Ergebnis war abhängig davon, wann man die Dependency geladen hat. Bei einem großen Monorepo, das entsprechend gepflegt ist (wie bei Google), war das selten ein Problem. Wollte man jedoch Bibliotheken von Drittanbietern nutzen, konnte jede Änderung dort den eigenen Build kaputt machen.

Mit Version 1.11 haben die Entwickler Go-Modules offiziell in die Go-Toolchain aufgenommen. Ähnlich wie in anderen Spra-



Profiwissen für Softwareentwickler

Kotlin, React, Node.js – Die neuen Techniken machen moderne Apps und Webanwendungen erst möglich. Mit unseren Büchern bleiben Sie am Puls der Zeit und profitieren von Beispielen aus der Entwicklerpraxis. Professionelles Wissen genau dann, wenn Sie es brauchen!

Alle Bücher und E-Books auch als Bundle:
www.rheinwerk-verlag.de

Rheinwerk

chen ermöglichen sie, Versionen von Abhängigkeiten festzuschreiben und somit reproduzierbare Builds zu erstellen.

Module sind ein oder mehrere Packages, die ein gemeinsames Import-Präfix haben und nach Semantic Versioning versioniert sind. Go-Modules ersetzen zahlreiche Alternativen, die zuvor die Go-Community zur Verfügung

gestellt hat. Mit ihrer Nutzung fällt außerdem die Notwendigkeit,Go-Programme innerhalb des „$GOPATH"-Pfads zu abzulegen, weg - eine weitere willkommene

Verbeserung der Go-Dependency-Toolchain.

Obwohl Go-Modules entwickelt wurden, ohne die Community einzubeziehen, ist ein Ende der Dependency-Management-Problematik somit absehbar. Im Entwicklungsalltag funktionieren Go-Modules nach Erfahrung der Autoren besser als die Community-Alternativen. Ein Umstieg bestehender Projekte ist einfach und schnell.

Go in der Praxis

Die Autoren konnten in den letzten Jahren reichlich Erfahrungen mit Go im professionellen Umfeld sammeln. Trotz – oder zum Teil gerade wegen – fehlender Spracheigenschaften, die zeitgemäße Sprachen wie Scala oder Rust mitbringen, ist Go heute häufig die Sprache der Wahl, um Probleme effizient zu lösen.

Durch die Schlichtheit der Sprache ist es nicht möglich, Probleme so elegant und ausdrucksstark wie in anderen Sprachen zu beseitigen. Stattdessen erinnert der Code Neueinsteiger eher an Spaghetticode. Vor allem die fehlende Fehlerbehandlung und die daraus resultierenden `if err == nil`-Blöcke wirken, vor allem am Anfang, abschreckend. Bei unbekanntem Code kann das jedoch ein großer Vorteil sein, da Go-Code in der Regel einfach nachvollziehbar ist.

Trotz des beschränkten Umfangs der Sprache gibt es eine Sammlung von Best-Practice-Regeln, die allgemein in der Community anerkannt sind. Dieses „Idiomatic Go" prägten maßgeblich die Go-Autoren, und es ist anfangs zum Teil gewöhnungsbedürftig, da sie beispielsweise Map- und Filter-Funktionen als unnütz betrachten. Teil von Idiomatic Go ist auch das Formatierprogramm gofmt, das ein Quasi-Standard bei Go-Programmierern ist und den Code einheitlich formatiert. Diskussionen, ob die geschweifte Klammer in dieselbe oder in die nächste Zeil gehört, oder ob man mit Tabs oder Leerzeichen einrückt (bei Go sind es übrigens Tabs), braucht man somit nicht zu führen.

Die Kombination einer unkomplizierten Sprache mit einer umfangreichen Standardbibliothek erlaubt es, Applikationen oftmals ohne weitere Drittanbieter-Bibliotheken zu erstellen. Werden Bibliotheken eingesetzt, handelt es sich oft nur um API-Wrapper, die REST-Calls abstrahieren (zum Beispiel gegen die API von Kubernetes oder AWS). Bei der Standardbibliothek fällt auf, dass erfahrene Programmierer beteiligt waren, die Probleme pragmatisch lösen möchten.

Einsatz in Infrastrukturprojekten

Der häufige Einsatz von Go in Infrastrukturprojekten hat mehrere gute Gründe. Zunächst sind Nebenläufigkeiten einfach durch Channels und Go-Routinen umsetzbar. Außerdem enthält die Standardbibliothek häufige Abhängigkeiten wie HTTP-Client, HTTP-Server oder JSON En- und Decoding. Da Go die Programme statisch kompiliert, reicht es, das Ergebnis einfach auf den Zielrechner zu kopieren. Installationen von Abhängigkeiten wie dynamische Bibliotheken, Gems oder einer Java-VM, um ein Programm starten zu können, entfallen. Entwickler können statisch kompilierte Go-Programme außerdem einfach als Docker-Images verpacken: Da man keine dynamischen Bibliotheken benötigt, besteht das Docker-Image nur aus dem Go-Binary.

Die Ressourcenanforderungen von Go-Programmen sind gering. In Systemen wie Kubernetes eignen sich Go-Programme daher hervorragend für kleine Sidecar-Umsetzungen, um beispielsweise eine Autorisierung zu implementieren, die mehrere Apps nutzen können. In Projekten, bei denen Wahlfreiheit bezüglich Programmiersprachen bestand, haben die Autoren die Beobachtung gemacht, dass Go-Neulinge nicht müde werden, sich über die sprachlichen Unzulänglichkeiten zu beschweren, das nächste Projekt dann aber trotz allem in Go statt Scala oder Haskell zu implementieren, da es „für das vorliegende Problem gerade gut passt".

Fazit

Go ist keine Programmiersprache, die State-of-the-art-Konzepte implementiert, und begeistert Sprachenthusiasten deshalb weniger. Pragmatiker dafür umso mehr: Features wurden so gewählt, dass die Sprache so klein und hantierbar wie C bleibt und gleichzeitig die Komplexität eines manuellen Speichermanagements unnötig macht. Nebenläufigkeit ist in die Sprache gut und leicht verständlich integriert und effizient implementiert, sodass sie auch den Herausforderungen einer Multi-Core-Welt gewachsen ist. Trotz einiger Beschränkungen und Eigenwilligkeiten schafft es Go oft, seinem Anspruch gerecht zu werden, die Produktivität einer dynamischen Sprache mit der Sicherheit einer statischen Sprache zu verbinden. Anforderungen sind oft effizient umsetzbar.

Eine stetig wachsende Community und die Rückendeckung von Google sichert eine fortlaufende Weiterentwicklung: Sie adressiert ebenfalls die genannten Schwachpunkte. Wobei man sich fragt, warum ein, in typischer Go-Manier, wenig revolutionäres Dependency-Management so lange braucht. Die fehlenden generischen Datentypen sind da kniffliger, da eine Umsetzung konträr zu anderen Zielsetzungen ist: Einfachheit der Sprache, schnelle Kompilierzeiten, geringe Implementationskomplexität sowie Stabilität und Kompatibilität der Sprache. Der aktuelle Vorschlag zu Contracts zeigt, dass das nicht unmöglich ist.

In einer zeitgemäßen, polyglotten Microservices-Umgebungen ist es unkompliziert, Go einmal auszuprobieren. Die flache Lernkurve und schnellen Turn-around-Zeiten von Go machen es einfach, einen Microservice oder zugehöriges Tooling probehalber mal in Go zu implementieren. (bbo@ix.de)

Christoph Iserlohn

ist Senior Consultant bei INNOQ. Er hat langjährige Erfahrung mit der Entwicklung und Architektur verteilter Systeme. Sein Hauptaugenmerk liegt dabei auf den Themen Skalierbarkeit, Verfügbarkeit und Sicherheit.

Daniel Bornkessel

arbeitet als Senior Consultant bei INNOQ. Seine aktuellen Schwerpunkte sind Continuous Delivery, Platform-as-a-Service-Systeme und Container-Techniken. Er berät Firmen bei der Migration zu einer Microservice-Architektur im Bereich DevOps.

B1 Managed Service & Support

individuell – umfassend – kundenorientiert

Neue oder bestehende Systemlandschaften stellen hohe Anforderungen an Ihr IT-Personal. Mit einem individuellen Support- und Betriebsvertrag von B1 Systems ergänzen Sie Ihr Team um die Erfahrung und das Wissen unserer über 100 festangestellten Linux- und Open-Source-Experten.

Unsere Kernthemen:

Linux Server & Desktop · Private Cloud (OpenStack & Ceph) · Containerization (Docker) · Orchestration (Kubernetes) · Monitoring (Icinga, Nagios & ELK) · Patch Management · Automatisierung (Ansible, Salt, Puppet, Chef)

Wir sind für Sie da – mit qualifizierten Reaktionszeiten ab 10 Minuten und Supportzeiten von 8x5 bis 24x7!

B1 SYSTEMS

B1 Systems GmbH - Ihr Linux-Partner

Linux/Open Source Consulting, Training, Development & Support

ROCKOLDING · KÖLN · BERLIN · DRESDEN

www.b1-systems.de · info@b1-systems.de

Programmieren in D

Mit der Tradition brechen

Robert Schadek

Die Programmiersprache D
hat sich im letzten Jahrzehnt weiterentwickelt.
Der Artikel stellt einige Sprachfeatures
im Einsatz vor.

Die Programmiersprache D ist eine Multiparadigmensprache. Sie vereint imperative, objektorientierte, funktionale und generative Programmierung in einer an C angelehnten Syntax, die zu nativen Binärdateien kompiliert wird. Walter Bright veröffentlichte 2001 die erste Version (D1) des Compilers. 2006 schloss sich Andrei Alexandrescu an und übernahm federführend die Entwicklung der D-Standardlaufzeitbibliothek namens Phobos. Im Jahr 2007 löste D2 diese Version der Sprache ab. Seitdem hat sich die Sprache inkrementell in über hundert Veröffentlichungen weiterentwickelt.

Es gibt verschiedene Compiler: Der Digital Mars Compiler DMD bildet die Referenzimplementierung, der LLVM D Compiler LDC verbindet das Frontend von DMD mit dem LLVM-Backend und schließlich der GNU D Compiler GDC, der ebenfalls eine Version des DMD-Frontends mit dem GCC-Backend verbindet. Mit der Release 9.1 des GCC ist D eine offiziell von GCC unterstützte Sprache. Dieser Artikel geht auf ausgewählte Fähigkeiten und Konstrukte der Sprache ein.

Listing 1 zeigt eine einfache `average`-Funktion. Dabei soll die an C angelehnte Syntax das Verständnis der Funktion vereinfachen. D hat eingebaute Unit-Testing-Fähigkeiten. Der mit dem Schlüsselwort `unittest` beginnende Block testet die `average`-Funktion. Die Unit-Tests werden vor Eintritt in die `main`-Funktion ausgeführt, nachdem der Compiler instruiert wurde, sie mit zu übersetzen.

Mit Scoped Imports arbeiten

Die beiden `import`-Statements in Listing 1 sind sogenannte Scoped Imports (alle Listings unter ix.de/zty4). Anders als in anderen Sprachen erlaubt D es, Imports innerhalb jedes Block-Statements aufzuführen. Die importierten Symbole sind dann nur innerhalb dieses Blocks sichtbar. Die Zahl der möglichen Symbole im globalen Namensraum eines Moduls wird damit eingeschränkt, es kommt selten zu Namenskonflikten. Die Funktion `empty` aus `std.array` ist zum Beispiel nicht im `unittest`-Block zu finden. Sie wird als `member`-Funktion auf dem Array mit dem Namen `arr` aufgerufen, obwohl `empty` als freistehende Funktion implementiert ist. Dies ist durchführbar, weil D ein Feature namens Uniform Function Call Syntax (UFCS) beherrscht. UFCS erlaubt es, normale Funktionen so aufzurufen, als wären sie Funktionen eines Typs, also Member-Funktionen. Das funktioniert, solange sich der Typ der Variablen, auf der die Funktion aufgerufen werden soll, implizit in den Typ des ersten Parameters der Funktion umwandeln lässt. Funktionen lassen sich außerdem ohne Klassenhierarchien wiederverwenden.

Das Schlüsselwort `foreach` leitet eine Schleife ein, die über die Elemente eines Arrays beziehungsweise einer Range iteriert. Eine Range ist ein Konzept, das es erlaubt, Strukturen zu iterieren – vergleichbar mit dem C++-Iteratorkonzept. Die Variable `sum` vom Typ `int` hat keine explizierte Initialisierung, allerdings sind alle Variablen in D mit einem Defaultwert initialisierbar. Für Integer-Typen und Pointer ist dieser Wert 0, für Float-Typen Not a Number (NaN). Das Attribut `@safe` verbietet Zeigerarithmetik und andere Konstrukte, die zu Speicherkorruptionen führen können.

Die vorletzte Zeile im `unittest`-Abschnitt zeigt die CTFE-Fähigkeiten (Compile Time Function Execution) von D. Der Typ der Variablen `ctAvg` ist `enum`, was den Compiler veranlasst, den Aufruf der Funktion `average` zur Kompilierzeit auszuführen. Bis auf einige wenige Ausnahmen lässt sich jede Funktion zur Kompilierzeit ausführen, solange sich ihre Parameter ebenfalls zur Kompilierzeit auswerten lassen. Listing 2 zeigt einige Fähigkeiten und eine vereinfachte Darstellung der grundlegenden Struktur von Arrays in D.

Jedes Array in D kennt seine Länge. Die erste Zeile im `unittest`-Abschnitt erzeugt ein neues Array, der Code in der folgenden Zeile prüft, ob das Array tatsächlich drei Elemente enthält. Ein sogenannter Slice initialisiert das zweite Array namens `copy`. Er enthält das zweite und dritte Element von `arr`. Die obere Grenze eines Slice ist exklusiv, die untere inklusiv. Betrachtet man die vereinfachte Form des Arrays, ist schnell zu erkennen, wie sich das bewerkstelligen lässt. Der Pointer wird um 1 inkrementiert und `length` um 1 reduziert. Die Speicherverwaltung ist vernachlässigbar, da D einen Garbage Collector besitzt.

Als Nächstes bekommt das erste Element von `copy` den Wert 1337 zugewiesen. Da der Pointer von `copy` auf denselben Speicher wie der von `arr` zeigt, sollte sich dies in den Elementen von `arr` widerspiegeln. Die nächste Zeile validiert diese Annahme. Der Speicher des Arrays `staticArray` befindet sich nicht auf dem Heap, sondern auf dem Stack, was beim Beschaffen eines temporären Arrays bekannter Größe hilfreich ist.

Zugriffe auf Arrays

Die Zeile mit der Anweisung `*=` zeigt eine sogenannte Array-Operation: Die Operation `*=` wird auf alle Elemente des Arrays `arr` angewendet. Die Slice-Operation ohne obere und untere Schran-