Parallelprogrammierung in Go

OpenRheinRuhr 2016
Oberhausen
5. November 2016

Harald Weidner
hweidner@gmx.net

Die Programmiersprache Go

- Freie Programmiersprache (BSD-Lizenz)
  - Entwicklung maßgeblich von Google getragen
  - Entwickelt seit 2007, erste Veröffentlichung 2009
  - Go 1.0 im Frühjahr 2012, aktuell Go 1.7.3
  - Sprache, Compiler, Standardbibliothek, Toolchain
- Ziele
  - Sichere und schnelle Sprache für große Projekte
  - Nutzung der Möglichkeiten moderner Rechner
- Eingebaute Sprachmittel für nebenläufige und parallele Programmierung

Nebenläufig und Parallel

- Nebenläufig (*concurrent*)
  - Mehrere Programmteile lassen sich unabhängig voneinander, in beliebiger Reihenfolge „nebeneinander“ ausführen
- Parallel (*parallel*)
  - Gleichzeitige Ausführung von Programmcode auf separaten Rechenwerken (z.B. CPU Cores)
  - Im Deutschen oft als Oberbegriff für *concurrent* und *parallel* verwendet

Nebenläufigkeit in Go

- **Communicating Sequential Processes (CSP)**
  - Tony Hoare (University of Oxford), 1978
  - Prozessalgebra zur Beschreibung von Interaktionen zwischen unabhängigen Prozessen
- Goroutine
  - Funktion, die nebenläufig ausgeführt wird
  - Syntax: `go f()` oder als Closure: `go func() { … }`
  - Implizite Verwendung, z.B. durch Bibliothek net/http
- Channel
  - Typisierter Kanal (Kommunikation, Synchronisation)

Beispiel: Fibonacci-Zahlen (1)

```
func fib(c chan int) {
   x, y := 1, 1
   for {
      c <- x
      x, y = y, x+y
   }
}

func main() {
   c := make(chan int)
   go fib(c)

   for i := 0; i < 30; i++ {
      fmt.Println(<-c)
   }
}
```

Beispiel: Fibonacci-Zahlen (2)

- Goroutine zur Fibonacci-Berechnung
  - Goroutine nach kurzer Zeit nutzlos, da Fibonacci-Zahlen den Wertebereich von int übersteigen
  - Goroutine verbleibt im Speicher, auch wenn sie nicht mehr gebraucht wird
- Deswegen
  - Bei jedem Start einer Goroutine festlegen, wann und wie sie beendet wird!
  - Z.B. Endliche Schleife oder Quit-Channel

Goroutine als endliche Schleife

```
func fib(c chan int, n int) {
   x, y := 1, 1
   for i := 0; i < n; i++ {
      c <- x
      x, y = y, x+y
   }
   close(c)
}

func main() {
   c := make(chan int)
   go fib(c, 30)

   for f := range c {
      fmt.Println(f)
   }
}
```

Goroutine mit Quit-Channel

```
func fib(c chan int, q chan int) {
    x, y := 1, 1
    for {
        select {
        case c <- x:   x, y = y, x+y
        case <-q:      return
        }
    }
}

func main() {
    c := make(chan int)
    q := make(chan int)
    go fib(c, q)

    for i := 0; i < 30; i++ {
        fmt.Println(<-c)
    }
    q <- 0
}
```

Goroutine als Closure

```
func fib(n int) chan int {
   c := make(chan int)
   go func() {
      x, y := 1, 1
      for i := 0; i < n; i++ {
         c <- x
         x, y = y, x+y
      }
      close(c)
   }()
   return c
}

func main() {
   for i := range fib(30) {
      fmt.Println(i)
   }
}
```

Mehr über Goroutinen (1)

- Goroutinen sind wesentlich leichtgewichtiger als Prozesse / Threads
  - 2 kB initialer Stack (1 Mio. Goroutinen = 2 GB RAM)
  - Scheduler in Go-Runtime verteilt Goroutinen auf Threads des Betriebssystems
- Parallelität durch OS-Threads
  - Gesteuert durch Env.-variable GOMAXPROCS
  - Default (seit Go 1.5): Anzahl CPUs
  - Oder explizit im Programm:
    `runtime.GOMAXPROCS(16)`

Mehr über Goroutinen (2)

- Goroutinen haben keine Identität
  - Können nicht „namentlich“ angesprochen werden
  - Können nicht von außen beendet werden
- Goroutinen, die blockieren, werden in eigenen OS-Thread ausgelagert
  - Blockierende I/O Operationen
  - Netzwerk (Network Poller)
- Preemptiver Scheduler (seit Go 1.2)
  - Preemption Points = Funktionsaufrufe

Mehr über Channels (1)

- First Class
  - Zuweisung an Variablen
  - Argument oder Rückgabewert von Funktionen
- Nebenläufig verwendbar (*thread-safe*)
  - Keine explizite Synchronisation nötig
  - Anders als bei den meisten Variablentypen in Go
- Gepuffert oder ungepuffert

```
c1 := make(chan int)    // ungepuffert
c2 := make(chan int, 10)  // gepuffert
```

Mehr über Channels (2)

- Eigene Typen für send-/receive-only Channels

```
ch := make(chan int, 1)
cs := (chan<- int)(ch)   // send-only
cr := (<-chan int)(ch)   // receive-only

cs <- 42
fmt.Println(<-cr)
close(cr)     // FEHLER: nur der Sender
              // kann den Ch. Schließen!
```

Mehr über Channels (3)

- Kann (vom Sender) explizit geschlossen werden

```
close(ch)
```

- Empfänger liest danach nur Defaultwerte
- Explizite Prüfung auf geschlossenen Channel

```
v, ok := <- ch
```

`ok` ist false, falls `ch` geschlossen wurde, sonst true

- Schreiben in geschlossenen Channel führt zur Runtime Panic!

Globale Variablen

- Goroutinen können auf gemeinsamen Speicher zugreifen (z.B. globale Variablen)
- Führt zu Wettlaufsituationen (Race Conditions)
- Unerwartete / nichtdeterministische Ergebnisse
- Können oft mit dem Race Detector erkannt werden, z.B.

```
go run -race program.go
```

Beispiel: Race Condition (1)

```
var counter = 0
var fin = make(chan int)

func count() {
   for i := 0; i < 10000; i++ { counter++ }
   fin <- 1
}

func main() {
   go count(); go count()
   <-fin; <-fin
   fmt.Println(counter)
}
```

- `counter++` nicht atomar, sondern: lesen-rechnen-schreiben
- Lesen und Schreiben verschiedener Goroutinen potenziell überlappend

Beispiel: Race Condition (2)

```
var a, b int

func f() {
   // mache irgendwas...
   a = 1
   b = 2
}

func main() {
   go f()
   for b == 0 { }
   fmt.Println(a)
}
```

- Die Reihenfolge der Zuweisungen an a und b ist für Betrachter außerhalb der Goroutine nicht deterministisch.
- Mögliche Umsortierung durch
  - Compiler
  - Prozessor
  - CPU-Cache

Mutual Exclusion Lock (Mutex)

```
var counter = 0
var lock sync.Mutex
var quit = make(chan int)

func count() {
   for i := 0; i < 10000; i++ {
      lock.Lock()
      counter++
      lock.Unlock()
   }
   quit <- 1
}

func main() {
   go count(); go count()
   <-quit; <-quit
   fmt.Println(counter)
}
```

- Erstes Lock() wird sofort gewährt.
- Weitere Aufrufe von Lock() werden blockiert, bis Unlock() aufgerufen wurde.

Read / Write Mutex

```
type Pmap struct {        // thread-safe map
    m map[string]string
    l sync.RWMutex
}

func (p *Pmap) Get(k string) (string, bool) {
    p.l.RLock()
    v, ok := p.m[k]
    p.l.RUnlock()
    return v, ok
}
func (p *Pmap) Set(k, v string) {
    p.l.Lock()
    p.m[k] = v
    p.l.Unlock()
}
func (p *Pmap) Delete(k string) {
    p.l.Lock()
    delete(p.m, k)
    p.l.Unlock()
}
```

- RLock() blockiert, wenn andere Lock()s offen
- Lock() blockiert, wenn andere Lock()s oder RLock()s offen
- Nach Lock() kein neues Rlock() mehr möglich

Waitgroup

```
func main() {
   fmt.Println("Hauptprogramm")

   var wg sync.WaitGroup
   wg.Add(4)
   for i := 0; i < 4; i++ {
      go func(n int) {
         fmt.Println("Goroutine", n)
         wg.Done()
      }(i)
   }
   wg.Wait()

   fmt.Println("Hauptprogramm")
}
```

- Warten auf Abschluss zusammen gehörender Events
- Add() vor dem Start einer neuen Goroutine
- Done() am Ende jeder Goroutine
- Wait() blockiert, bis alle Done() da sind

Weitere Sync.-Mechanismen

- Condition Variable
  - Verteilt einen Mutex an wartende Goroutinen
- Once
  - Einmalige Ausführung einer Funktion
- Pool (ab Go 1.3)
  - Verwaltung und Bereitstellung ungenutzter Ressourcen
- Atomare Operationen im Package „sync/atomic“
  - Load, Store, Add, Swap, Compare-and-Swap

Go Memory Model (1)

- Spezifikation der Garantien von Go bezüglich des gemeinsamen Zugriffs auf gemeinsame Variablen
- 11 Regeln mit Erläuterungen und Beispielen
- URL: https://golang.org/ref/mem
- Definiert Ordnungsrelation *happens before* (HB)
  - Mathematisch: Halbordnung (*partial order*)
  - Transitiv: a HB b ∧ b HB c ⇒ a HB c

Go Memory Model (2)

- Allgemeine Regeln
  - Innerhalb einer Goroutine entspricht die HB Ordnung der Reihenfolge der Programmzeilen im Quelltext.
  - Wenn ein Modul p ein Modul q importiert, gilt: Initialisierung von q HB Initialisierung von p.
  - Alle Initialisierungen HB der Hauptfunktion main.main().

Go Memory Model (3)

- Regeln für Goroutinen und Channels
  - Das `go` Statement zum Start einer neuen Goroutine HB der Goroutine selbst.
  - Ein Senden in einen Channel HB dem Empfang des gesendeten Wertes.
  - Das Schließen eines Channels HB dem Empfang des dazugehörigen Defaultwertes.
  - Ein Empfangen von einem ungepufferten Channel HB dem Abschluss des dazugehörigen Sendens.
  - Das k'te Empfangen von einem Channel mit Kapazität C HB dem k+C'ten Senden an diesen Ch.

Go Memory Model (4)

- Regeln für Synchronisierungsoperationen
  - Für eine Variable vom Typ sync.Mutex oder sync.RWMutex l und n<m gilt: der n'te Aufruf von l.Unlock() HB dem Abschluß des m'ten Aufruf von l.Lock()
  - Für jeden Aufruf von l.RLock() bei einem sync.RWMutex l gibt es ein n, so dass
    - n'te Aufruf von l.Unlock() HB l.RLock() und
    - l.RUnlock() HB n+1'te Aufruf von l.Lock()
  - Ein Aufruf von f() aus einem once.Do(f) HB jedem Return von once.Do(f).

Weiterführende Informationen

- Sprache, Spezifikation: http://golang.org/doc
- Vortragsfolien: Go Concurrency Patterns
  https://talks.golang.org/2012/concurrency.slide
- Vortragsvideo: Concurrency is not Parallelism
  https://vimeo.com/49718712
- Blog / Website von Dmitry Vyukov
  http://www.1024cores.net/
- Christian Maurer: Nichtsequentielle Programmierung mit Go 1. Springer, 2. Auflage, 2012
  http://www.springer.com/de/book/9783642299681