Das Actor-Modell und Umsetzung in Proto.Actor

Verteilte Softwaresysteme

Prof. Dr. Oliver Braun

Letzte Änderung: 26.11.2018 18:21

Actor-Modell

Nebenläufige Berechnungen mit Actors

- 1973: Carl Hewitt, Peter Bishop und Richard Steiger: "*A Universal Modular ACTOR Formalism for Artificial Intelligence*"
  - Parallelverarbeitung in High Performance Networks
  - gutes Video von 2012 mit Carl Hewitt auf YouTube
- **Actors** sind
  - nebenläufige Einheiten
  - ohne geteilten Speicher
  - die ausschließlich über Nachrichten kommunizieren
- ein **Actor** hat
  - einen Posteingang
  - eine Adresse
  - ein Verhalten

Message Passing statt Locks und Blocks

- Actors können Nachrichten an andere Actors senden
- dies involviert **nie** einen Kontextwechsel zum anderen Thread
- Actor kann nach dem Senden sofort weiter arbeiten
- wesentlicher Unterschied zu Methodenaufrufen:
  - Nachrichten haben kein Ergebnis
  - hat die Ausführung der anderen Actors ein Ergebnis, so sendet dieser einfach eine Antwort
- ein Actor arbeitet immer rein sequentiell auf seinen (gekapselten) Daten
  - d.h. es ist nie ein Lock notwendig, nichts kann blockieren
- ein Actor arbeitet immer nur eine Nachricht ab

Ablauf beim Empfangen einer Nachricht

- der Actor fügt die Nachricht am Ende eine Queue an
- wenn der Actor nicht gerade ausgeführt wurde, wird es als ausführbar markiert
- ein Scheduler startet (irgendwann) den Actor
- der Actor nimmt die erste Nachricht vom Anfang der Queue (Abweichungen je nach Implementierung möglich)
- der Actor reagiert mit folgenden Optionen (auch mehrere)
  - internen Zustand modifizieren
  - Nachrichten zu anderen Actors senden
  - neue Actors erzeugen
  - entscheiden wie er auf die nächste Nachricht reagieren soll
- der Scheduler entzieht dem Actor die CPU

Ein Actor hat...

- eine Mailbox
  - die Nachrichten-Queue
- ein Verhalten
  - Zustand des Actors, interne Variablen, ...
- Nachrichten
  - Daten, die einem Methodenaufruf und Argumenten entspechen
- eine Ausführungsumgebung
  - die Maschinerie, die bei Actors mit Nachrichten den “Nachrichtenbehandlungscode” aufruft
- eine Adresse

Fehlersituationen

- wenn Fehler in einem Task, wie z.B. User ID unbekannt, einfach eine Message mit dem Fehler als Antwort senden
- wenn ein Actor abstürzt?
  - einige Implementierungen nutzen eine baumartige Hierarchie von Actors
  - analog zu Betriebssystemen, die Prozesse hierarchisch organisieren
  - der Vorgänger (hier: **Supervisor**) wird beim Fehler benachrichtigt und kann reagieren

Actor-Implementierungen

- Erlang
  - funktionale Programmiersprache ursprünglich von Ericsson für Telekomsysteme
  - Nebenläufigkeit basiert auf dem Actor-Modell
  - WhatsApp ist in Erlang mit Actors implementiert
- Akka
  - Library für JVM
  - Scala- und Java-API
- Microsoft Orleans
- Proto.Actor
  - Cross-platform actors for .NET, Go, Java and Kotlin

Proto.Actor

Was ist Proto.Actor?

- Proto.Actor bezeichnet sich selbst als
  - Next generation Actor Model framework.

proto actor

- Besonderheit an Proto.Actor: *Actor Standard Protocol*
- damit können Actors in verschiedenen Sprachen geschrieben werden,
  - und trotzdem direkt miteinander kommunizieren
- *We believe that writing correct, concurrent, fault-tolerant and scalable applications is too hard.*
- *Most of the time, that's because we are using the wrong tools and the wrong level of abstraction.*

Ein einzigartiger hybrider Ansatz

- Actors
  - einfache Abstraktion für Nebenläufigkeit und Parallelisierung
  - asynchrones, nicht-blockierendes, performantes event-getriebenes Programmiermodell
  - sehr leicht-gewichtige Prozesse (mehrere Millionen pro GB Heap)
- Virtual Actors
  - einfacher RPC-basierter Ansatz für verteilte Programmierung
- Fault Tolerance
  - Supervisor-Hierachien mit “Let-it-crash”-Semantik
  - Supervisor-Hierarchien können über mehrer Virtuelle Maschinen verteilt sein
  - Self-Healing
- Location Transparency
  - distributed by default

(Proto-)Actors in Go

Quelle:

https://github.com/AsynkronIT/protoactor-go/blob/dev/resources/batman.jpg

Hello-World-Actor (1/3)

- eine Message ist einfach eine Struct, z.B.

```
type hello struct{ Who string }
```

- genauso wie der Actor, z.B.

```
type helloActor struct{}
```

Hello-World-Actor (2/3)

- mit einer `Receive`-Methode, z.B.

```
func (state *helloActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *hello:
        fmt.Printf("Hello %v\n", msg.Who)
    }
}
```

Hello-World-Actor (3/3)

- und die `main`, z.B.

```
func main() {
    props := actor.FromProducer(func() actor.Actor {
        return &helloActor{}
    })
    pid := actor.Spawn(props)
    pid.Tell(&hello{Who: "Roger"})
    console.ReadLine()
}
```

Actors erzeugen

- mit Hilfe von `Props` können Actors erzeugt werden
- `Props` kann einen Actor aus einer Funktion, die die Abarbeitung der Nachrichten implementiert, erzeugen

```
type ActorFunc func(c Context)
```

- alternativ aus einem Typ der das `Actor`-Interface implementiert

```
type Actor interface {
    Receive(c Context)
}
```

... und starten

- nachdem der Actor erzeugt wurde, kann er mit

```
func Spawn(props *Props) *PID
```

  erzeugt werden

- der Actor bekommt dann eine eindeutige ID
- mit `SpawnNamed` kann auch ein eigener Name vergeben werden
- für den Actor wird dann eine Mailbox erzeugt
  - Nachrichten werden an die Mailbox gesendet und an den Actor-Prozess weiter geleitet
- `Receive` wird immer synchron aufgerufen
  - der Zustand des Actors muss daher nicht synchronisiert werden

Kommunikation (1/2)

- die Kommunikation mit einem Actor erfolgt über seine `PID`
- die `Tell`-Methode sendet eine asynchrone Nachricht, z.B.

```
pid.Tell("Hello World")
```

- mit `Request` wird auch eine asynchrone Nachricht gesendet, es wird aber erwartet, dass der Empfänger eine Antwort mit `respondTo` sendet

Kommunikation (2/2)

- synchrone Kommunikation über Futures
- `RequestFuture` sendet eine Nachricht und hat als Ergebnis sofort ein `Future`, z.B.

```
f := actor.RequestFuture(pid,"Hello",
    50 * time.Millisecond)
res, err := f.Result() // waits for pid to reply
```

Lifecycle Events (1/3)

- Nachrichten für Lifecylce-Events als `SystemMessages`

```
type SystemMessage interface {
    SystemMessage()
}

func (*Started) SystemMessage()
func (*Restart) SystemMessage()
func (*Stop) SystemMessage()
func (*Terminated) SystemMessage()
func (*Unwatch) SystemMessage()
func (*Watch) SystemMessage()
func (*Failure) SystemMessage()
```

Lifecycle Events (2/3)

- `AutoReceiveMessage`s werden automatisch behandelt

```
type AutoReceiveMessage interface {
    AutoReceiveMessage()
}
func (*Restarting) AutoReceiveMessage()
func (*Stopped) AutoReceiveMessage()
func (*Stopping) AutoReceiveMessage()
func (*PoisonPill) AutoReceiveMessage()
```

Lifecycle Events (3/3)

- Beispiel

```
func (state *HelloActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *actor.Started:
        fmt.Println("Started, initialize actor here")
    case *actor.Stopping:
        fmt.Println("Stopping, actor is about shut down"))
    case *actor.Restarting:
        fmt.Println("Restarting, actor is about restart")
    case Hello:
        fmt.Printf("Hello %v\n", msg.Who)
    }
}
```

Statemachines / Verhaltensänderung (1/2)

- damit die eine `Receive`-Methode nicht unübersichtlich wird,
  - gibt es die Möglichkeit das Verhalten so zu ändern, das
  - der Actor mit einer anderen Methode Nachrichten bearbeitet
- `SetBehavior(behavior ActorFunc)` ersetzt das Verhalten durch `behavior`
- `PushBehavior(behavior ActorFunc)` ersetzt das Verhalten durch `behavior` und legt das aktuelle Verhalten auf einen Stack
- `PopBehavior()` ersetzt das Verhalten durch das oberste vom Stack (und nimmt es vom Stack)
- sind Methoden des `Context`

Statemachines / Verhaltensänderung (2/2)

- Beispiel

```
func (state *SetBehaviorActor) Receive(ctx actor.Context) {
    switch msg := ctx.Message().(type) {
    case Hello:
        fmt.Printf("Hello %v\n", msg.Who)
        ctx.SetBehavior(state.Other)
    }
}

func (state *SetBehaviorActor) Other(ctx actor.Context) {
    switch msg := ctx.Message().(type) {
    case Hello:
        fmt.Printf("%v, in another behavior", msg.Who)
    }
}
```

Supervision — Directives

- Root Actors werden durch die `DefaultSupervisionStrategy()` überwacht
  - löst `RestartDirective` aus, die einen Actor durch eine neue Instanz ersetzt
- andere Direktiven sind
  - `ResumeDirective`: Fortsetzung des fehlerhaften Actors
  - `StopDirective`: Stoppen des fehlerhaften Actors
  - `EscalateDirective`: Eskalieren an den übergeordneten Supervisor
- mit der letzten kann z.B. bei Ausfall eines Actors die gesamte Hierarchie neu gestartet werden

Supervision — Strategies

- startet ein Actor einen neuen Actor, so ist der startende der Parent und der gestartete sein Child
- ein Parent-Actor kann eine eigene `SupervisorStrategy` vorgeben
- die `OneForOneStrategy` wendet eine gegebene Directive auf das fehlerhafte Kind an
- die `AllForOneStrategy` wendet eine gegebene Directive bei einem fehlerhaften Kind auf alle Kinder an

Remote Actors

- in Proto.Actor kommt zum Einsatz:
  - gRPC
    - open-source RPC-Implementierung, sprach- und plattformübergreifend
  - Protocol Buffers

Protocol Buffers

Was ist Protocol Buffers

- felixbler, effizienter und automatisierter Ansatz zum Serialisieren von strukturierten Daten
- *think XML, but smaller, faster, and simpler*
- Warum dann nicht einfach XML?
  - einfacher
  - 3- bis 10-fach kleinere Pakete
  - 20- bis 100-fach schneller
  - weniger uneindeutig
  - generierter Datenzugriffscode ist einfacher zu nutzen

Protokoll-Format

- Spezifikation in einer `.proto`-Datei
- zuerst Protokoll-Format:

```
syntax = "proto3";
```

- aktuell Version 3 (notwendig für Go)
- `package` wird auch als Go-Package verwendet
    - Abweichung über `go_package`

Nachrichten (1/3)

- Nachrichten werden als `message`s definiert
  - Aufzählung von typisierten Datenfeldern
- Beispiel

```
message Person {
  string name = 1;
  int32 id = 2;  // Unique ID number for this person.
  string email = 3;
   ...
}
```

Nachrichten (2/3)

- eigene Aufzählungstypen möglich, z.B.

```
enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
}
message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
}
 ...
```

- Nachricht `Person` enthält die Nachricht `PhoneNumber`

Nachrichten (3/3)

- … sogar mehrfach

```
  repeated PhoneNumber phones = 4;

  google.protobuf.Timestamp last_updated = 5;
}
```

- für letzteres ist ein Import notwendig

```
import "google/protobuf/timestamp.proto";
```

- und ein Adressbuch sind dann viele Personen

```
message AddressBook {
  repeated Person people = 1;
}
```

Codegenerieren mit dem Proto-Buffers-Compiler

- herunterladen und installieren von
  https://github.com/protocolbuffers/protobuf
- für Go-Code muss ein Compiler-Plugin installiert werden
- wir nutzen **abweichend** vom Protobuf-Tutorial `gogoslick`

```
go get github.com/gogo/protobuf/proto
go get github.com/gogo/protobuf/protoc-gen-gogoslick
go get github.com/gogo/protobuf/gogoproto
```

- Go-Code generieren mit

```
protoc –gogoslick_out=. messages.proto
```

- siehe Livecoding-Repo

Remote Actors

Remote zur Verfügung stellen

- Starten eines Actorsystems, das remote erreicht werden kann, mit

```
import "github.com/AsynkronIT/protoactor-go/remote"
func Start(address string, options ...RemotingOption)
```

- Beispiel

```
remote.Start("localhost:8091")
props := actor.FromProducer(
  func() actor.Actor { return &MyActor{} })
actor.SpawnNamed(props, "hello")
```

Zugriff über die Adresse und den Namen

- mit der Funktion

```
func NewPID(address, id string) *PID
```

- Beispiel

```
remote := actor.NewPID("localhost:8091", "hello")
```

- wenn dieser Actor auch per Remote verfügbar ist, kann ein anderer einfach an ihn als `msg.Sender` zurück senden
- weitere Beispiele unter https://github.com/AsynkronIT/protoactor-go/tree/dev/examples