

Das Actor-Modell und Umsetzung in Proto.Actor

Verteilte Softwaresysteme

Prof. Dr. Oliver Braun

Letzte Änderung: 14.10.2019 18:03

Actor-Modell

- 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

- 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

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

- 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

- 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.*

- 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-Hierarchien mit “Let-it-crash”-Semantik
 - Supervisor-Hierarchien können über mehrerer Virtuelle Maschinen verteilt sein
 - Self-Healing
- Location Transparency
 - distributed by default



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{}
```

- 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)  
    }  
}
```

- und die main, z.B.

```
func main() {  
    context := actor.EmptyRootContext  
    props := actor.PropsFromProducer(func() actor.Actor {  
        return &helloActor{  
        })  
    pid := context.Spawn(props)  
    context.Send(pid, &hello{who: "Roger"})  
    console.ReadLine()  
}
```

- 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)  
}
```


- nachdem der Actor erzeugt wurde, kann er mit

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

gestartet 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
- **Achtung:** Mischen Sie die Actors nicht mit anderen Nebenläufigkeitskonzepten!
- Actors sind nebenläufige Einheiten
- Innerhalb eines Actors passiert **alles** sequentiell

- die Kommunikation mit einem Actor erfolgt mit Hilfe seiner **PID**
- das Actor-System läuft in einem Context, der das Senden von Nachrichten unterstützt
- die **Send**-Methode des Context sendet eine Nachricht, z.B.
`context.Send(pid, msg)`
 - bei `context.Send` ist der Absender nur dann bekannt, wenn er explizit in die Message geschrieben wird
- alternativ kann die Nachricht mit `context.Request` gesendet werden
 - in dem Fall wird der Absender gesetzt und kann mit `context.Sender` ermittelt werden
 - mit `context.Respond` kann dann auch direkt eine Antwort an den Absender gesendet werden

- Kommunikation ist auch über sog. Futures möglich
 - insbesondere dann wichtig, wenn der Sender kein Actor ist und eine Antwort empfangen möchte
- `context.RequestFuture` sendet eine Nachricht und hat als Ergebnis sofort ein `Future`, z.B.

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

- Nachrichten für Lifecycle-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()
```

- AutoReceiveMessages werden automatisch behandelt

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

- 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)  
    }  
}
```

- damit die eine **Receive**-Methode nicht unübersichtlich wird¹,
 - gibt es die Möglichkeit das Verhalten so zu ändern, das
 - der Actor mit einer anderen Methode Nachrichten bearbeitet
- **Become(behavior ActorFunc)** ersetzt das Verhalten durch **behavior**
- **BecomeStacked(behavior ActorFunc)** ersetzt das Verhalten durch **behavior** und legt das aktuelle Verhalten auf einen Stack
- **UnbecomeStacked()** ersetzt das Verhalten durch das oberste vom Stack (und nimmt es vom Stack)
- bei einem Actor der sein Verhalten ändern kann, muss der Stack explizit gespeichert werden
 - initialisiert wird er mit **actor.NewBehavior**

¹Neben dem ganz normalen Aufteilen des Codes auf mehrere Funktionen.

Beispiel (1/2)

```
type SetBehaviorActor struct {  
    behavior actor.Behavior // Behavior-Stack  
}  
func NewSetBehaviorActor() actor.Actor {  
    act := &SetBehaviorActor{  
        behavior: actor.NewBehavior(), // empty Stack  
    }  
    act.behavior.Become(act.One) // "push"  
    return act  
}  
  
func (state *SetBehaviorActor) Receive(context actor.Context) {  
    state.behavior.Receive(context) // the Stack receives  
}
```

Beispiel (2/2)

```
func (state *SetBehaviorActor) One(context actor.Context) {  
    switch msg := context.Message().(type) {  
    case Hello:  
        fmt.Printf("Hello %v\n", msg.who)  
        state.behavior.Become(state.Other)  
    }  
}
```

```
func (state *SetBehaviorActor) Other(context actor.Context) {  
    switch msg := context.Message().(type) {  
    case Hello:  
        fmt.Printf(  
            "%v, ey we are now handling messages in another behavior",  
            msg.who)  
        }  
    }  
}
```

- 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

- 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

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

Protocol Buffers

- flexibler, 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 Datenzugriffscod ist einfacher zu nutzen

- 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 werden als `messages` definiert
 - Aufzählung von typisierten Datenfeldern
- Beispiel

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

- ... 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;  
}
```

- 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
```

Remote Actors

- 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")  
remote.Register("hello",  
    actor.PropsFromProducer(NewMyActor))
```

Zugriff über die Adresse und den Namen

- mit der Funktion

```
func SpawnNamed(address, name, kind string,  
                timeout time.Duration)  
                (*ActorPidResponse, error)
```

- Beispiel

```
remote.SpawnNamed("localhost:8091", "remote", "hello",  
                 5*time.Second)
```

- die `ActorPidResponse` enthält die `Pid`, an die dann einfach eine Nachricht per `Context.Send` etc. gesendet werden kann
- weitere Beispiele unter <https://github.com/AsynkronIT/protoactor-go/tree/dev/examples>