再考 アクターモデル

吉祥寺.pm36

yuuki takezawa / ytake

Profile

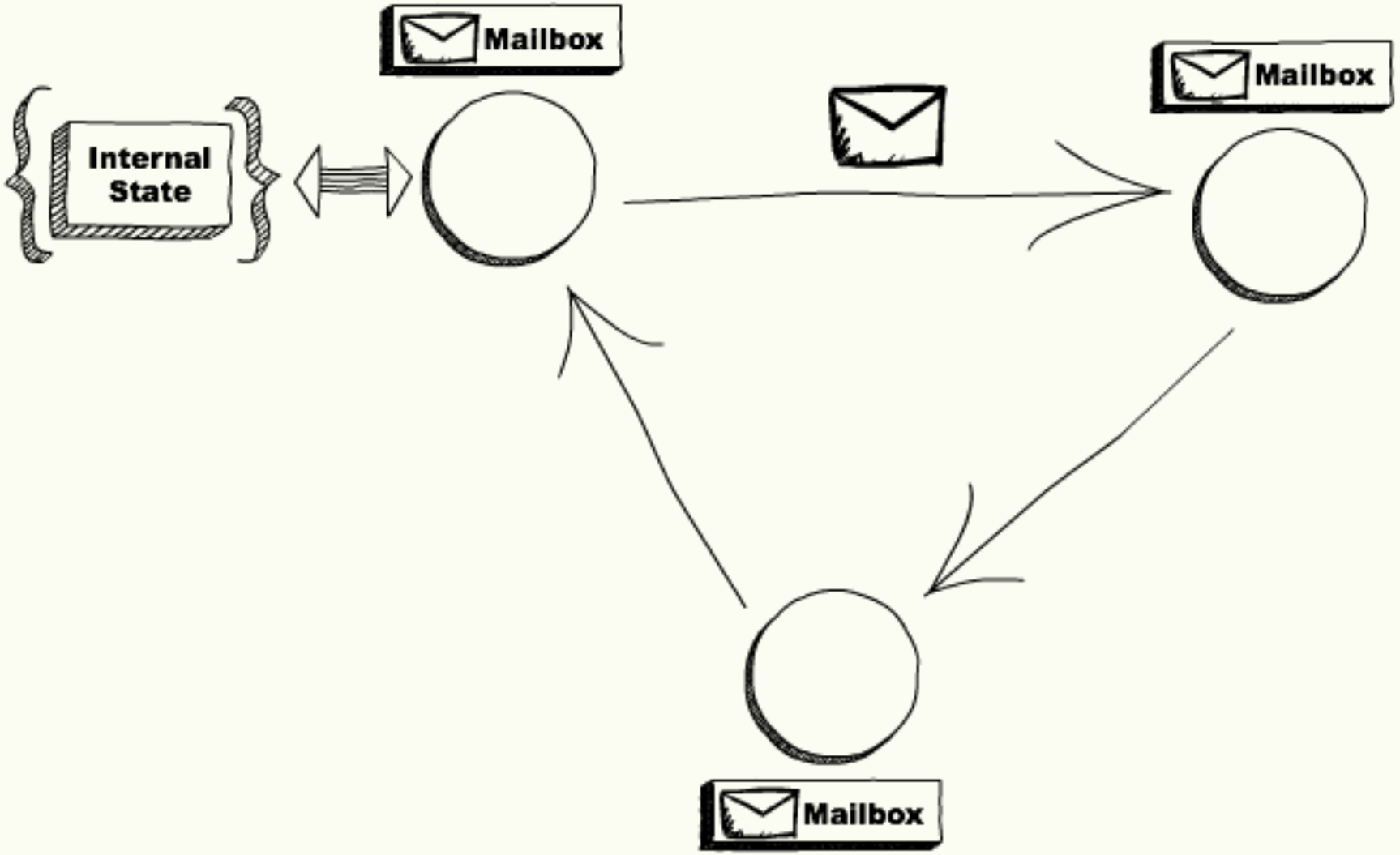
- 竹澤 有貴 a.k.a ytake
- 千株式会社 CTO / ほか技術顧問（ネットプロテクションズなど）
- Go / Scala / Kotlin
- アクターモデル大好き

アクターモデルって？

- 1973年に発表された並行計算の数学的モデルの一種
- 新しいものではありません
- ErlangやScalaなどでおなじみ
- メモリ共有を行わず、ノンブロッキングで
  独立したアクターが状態を持つ

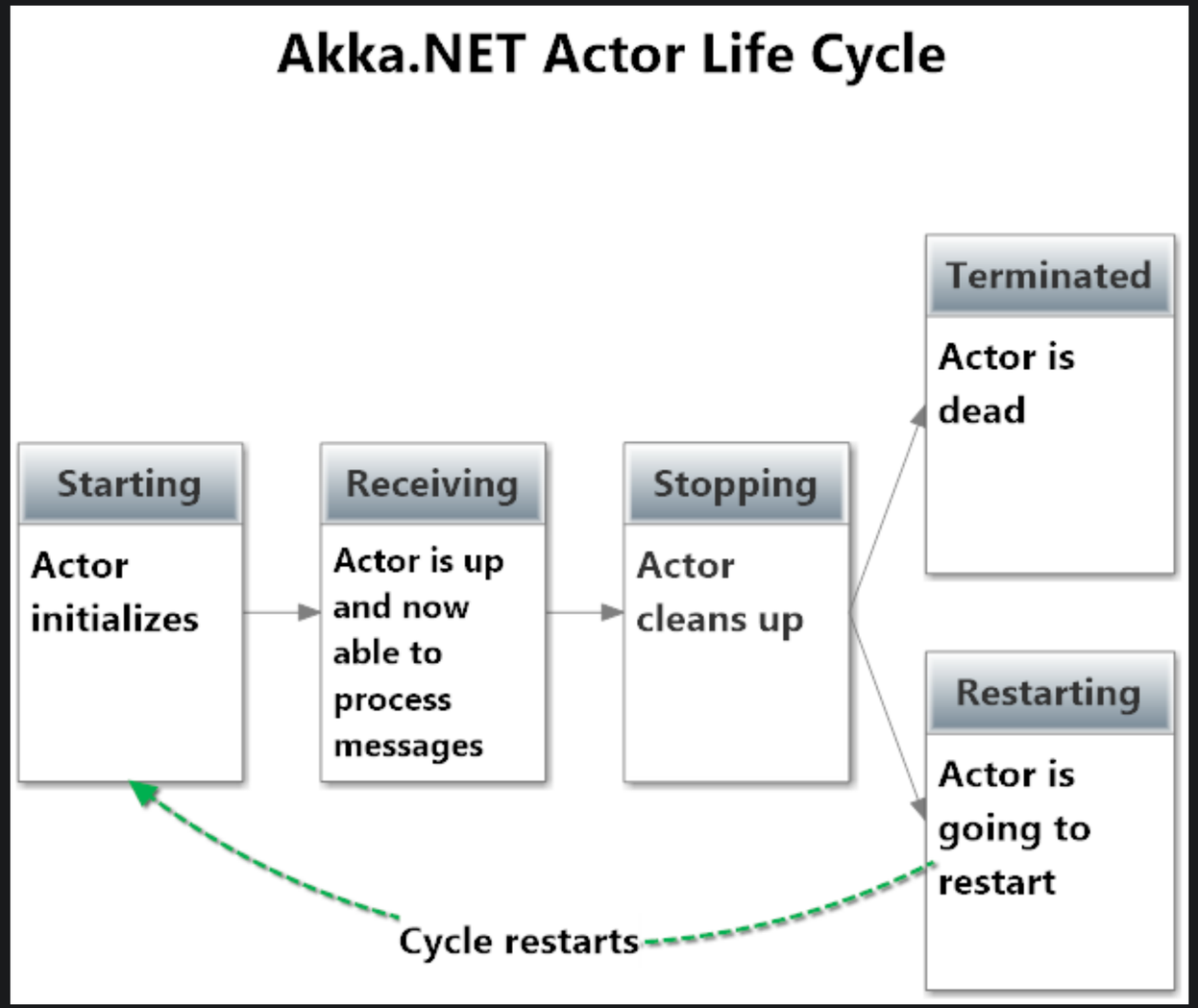
アクターの責務

- アクターのメッセージを送信する
- アクターを生成する
- メッセージに適用する動作を行う


Mailbox
Internal State
Mailbox
Mailbox

アクターモデルって？

- アクターは独立して動作し、それぞれがライフサイクルを持つ
- メモリの共有などは一切行わない
- アクターを直接操作することは不可能でイミュータブル
- メッセージのみでやり取りを行う
- 位置透過性



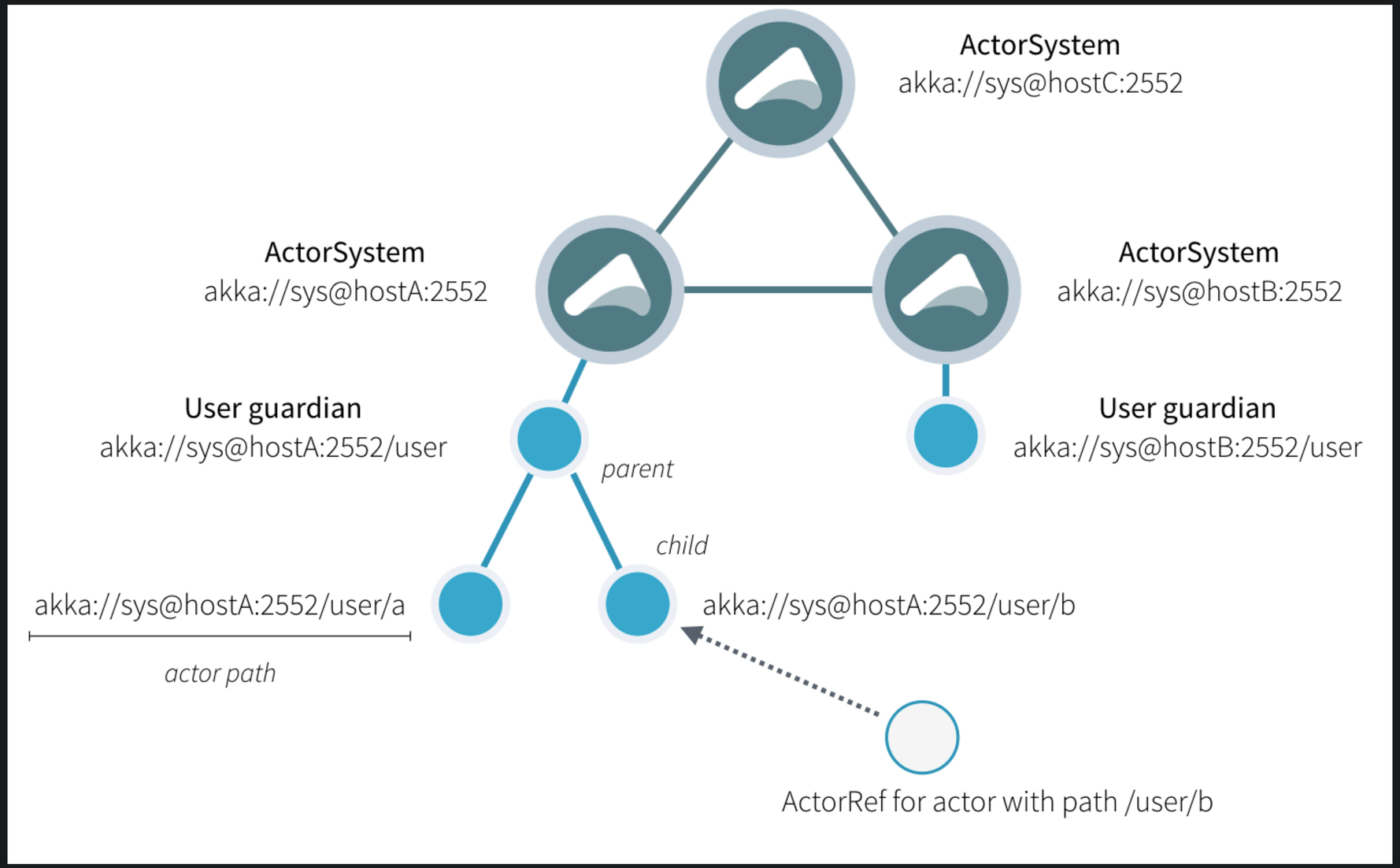
出典 https://petabridge.com/blog/akkadotnet-what-is-an-actor/

位置透過性

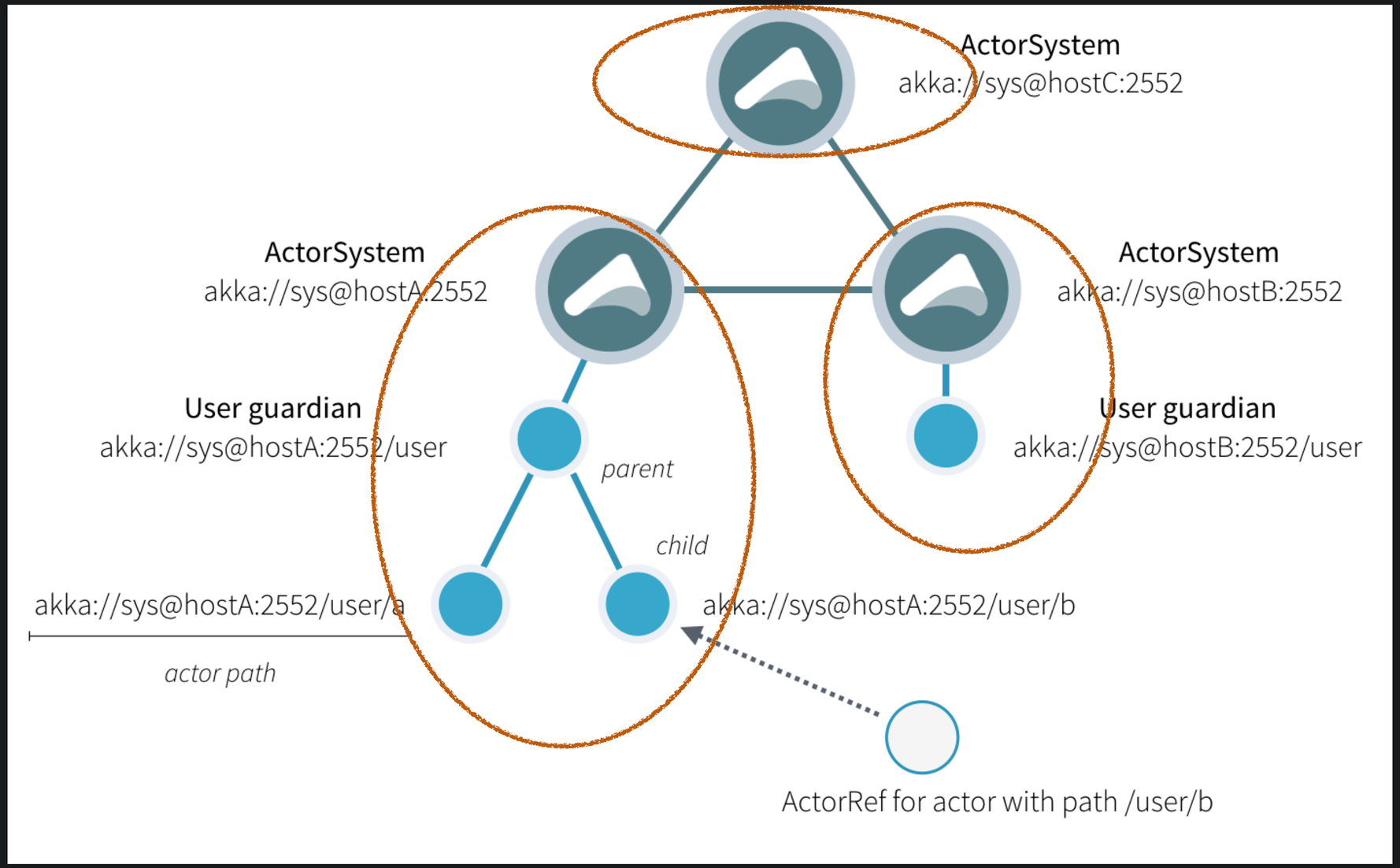
- アクターはローカルやリモート、クラスタで動作できる

- どこにあるのか、どのようにして呼び出すのかは
  コード上で区別することがなくなる

- 境界づけられたコンテキストを基に分割することや、
  スループット向上のため分割することが容易

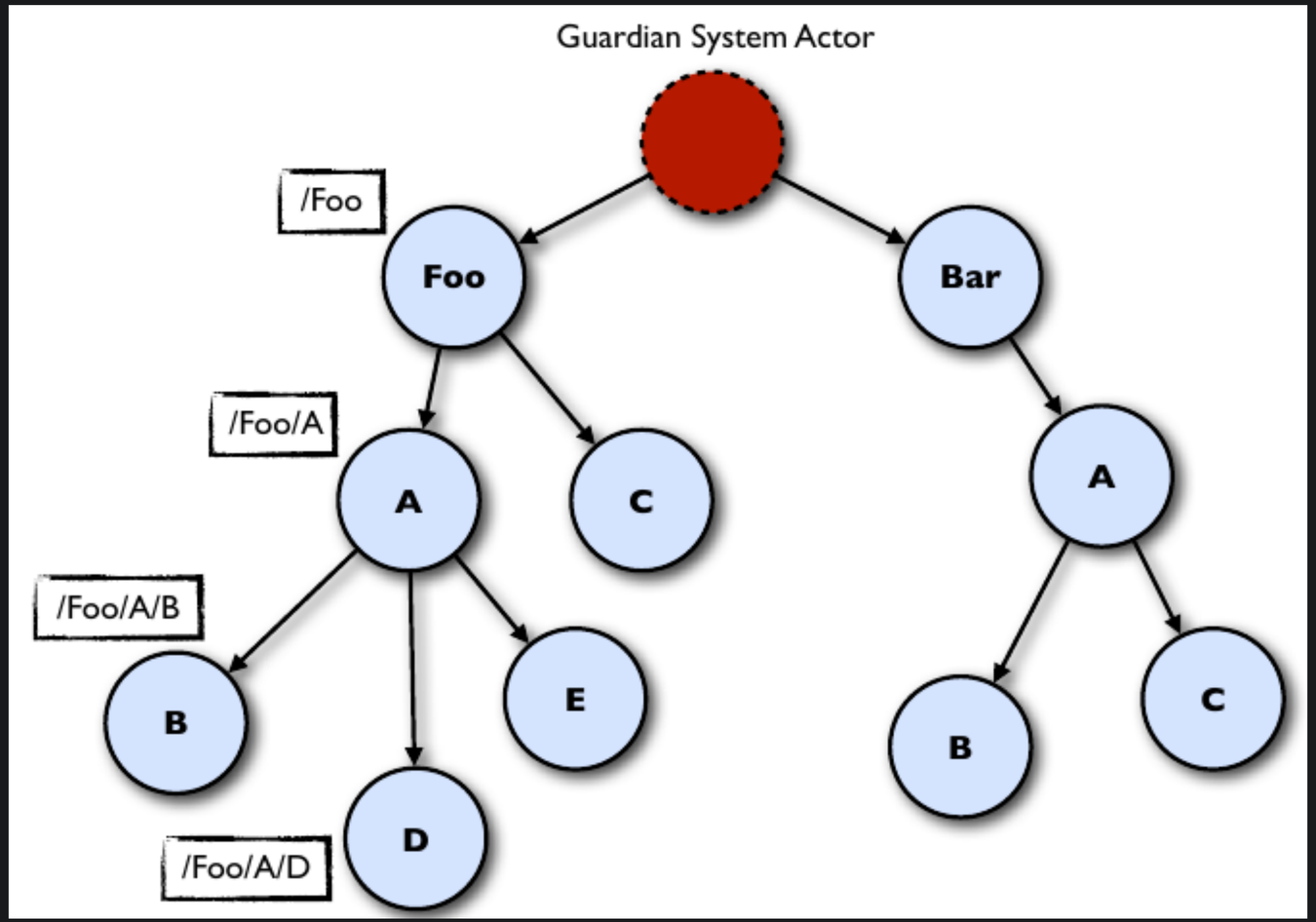


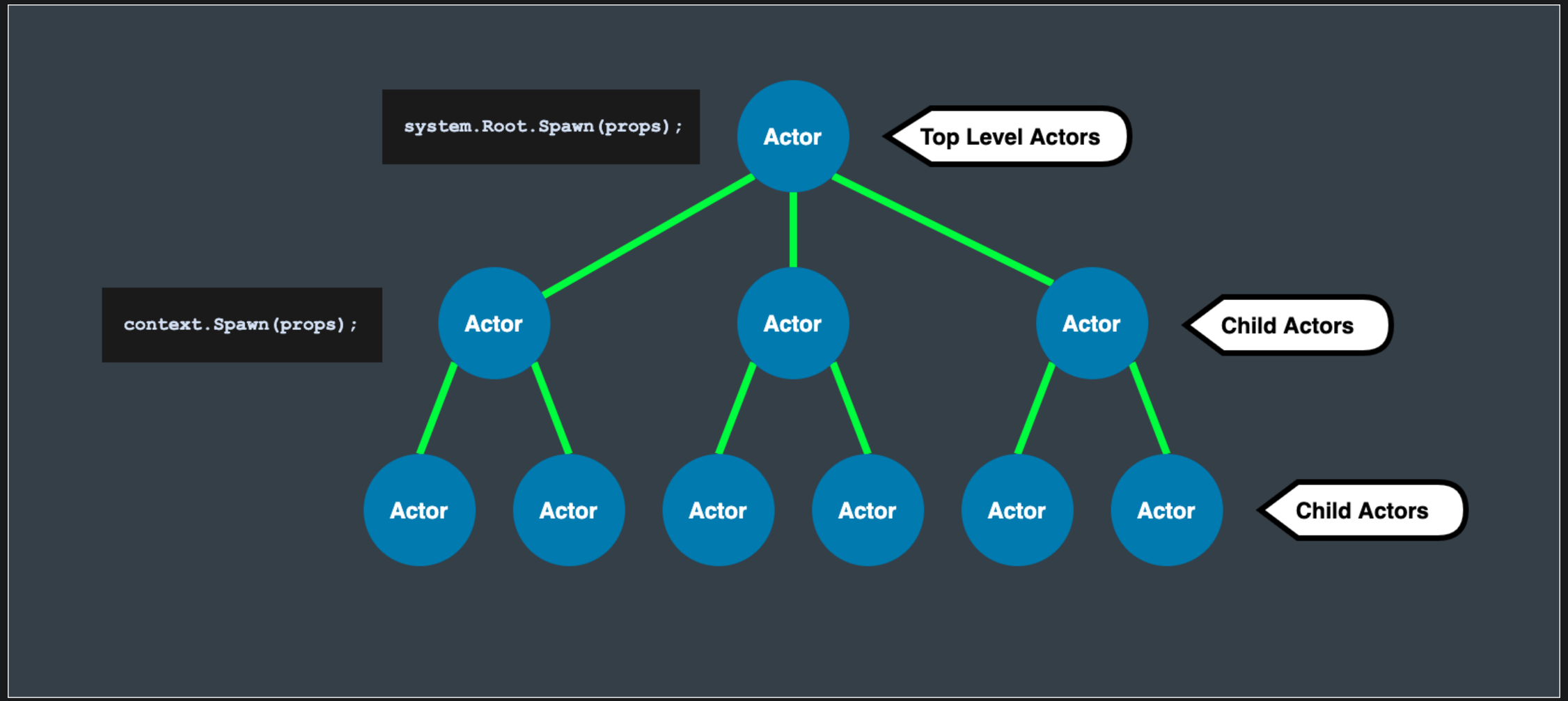
出典 https://pekko.apache.org/docs/pekko/current/general/addressing.html



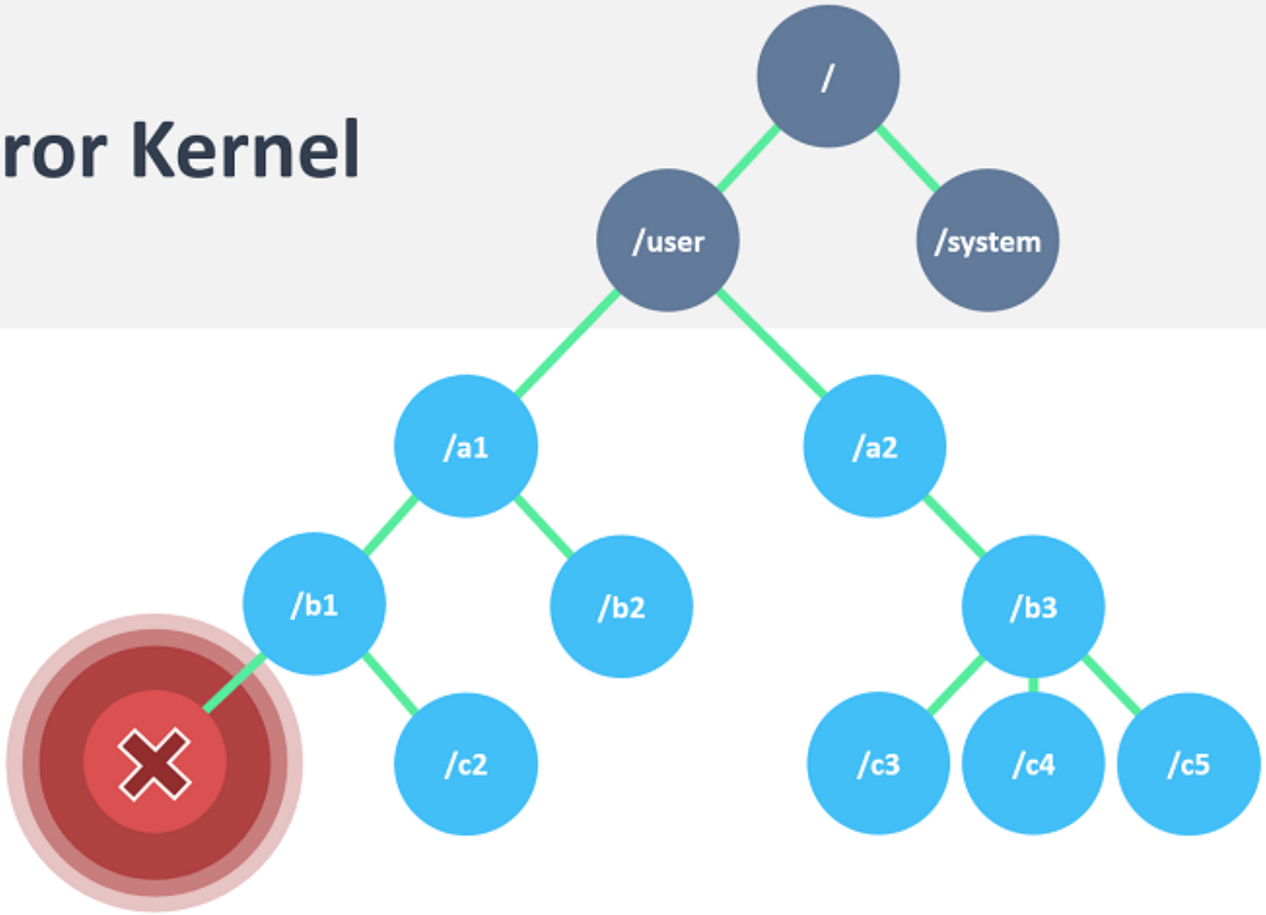
出典 https://pekko.apache.org/docs/pekko/current/general/addressing.html

階層構造をもつ


Guardian System Actor
/Foo
Foo
Bar
/Foo/A
A
C
A
/Foo/A/B
B
E
B
C
D
/Foo/A/D


system.Root.Spawn(props);
Actor
Top Level Actors
context.Spawn(props);
Actor
Actor
Actor
Child Actors
Actor
Actor
Actor
Actor
Actor
Actor
Child Actors

Error Kernel



フォールトトレランス

- ヒエラルキーを持つということは

  親アクターが子アクターを監督・管理する責務を持つ

  仕事に失敗した場合にどのような戦略を？（スーパーバイザ戦略）

- 親子関係にないアクターも監視することができる

- 失敗時にどのように復旧させるかを指示できる

監視例

```
func (state *parentActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *hello:
        props := actor.PropsFromProducer(newChildActor)
        child := context.Spawn(props)
        context.Send(child, msg)
    case *actor.Terminated:
        fmt.Println("child terminated", msg.Who)
    }
}

func newParentActor() actor.Actor {
    return &parentActor{}
}

type childActor struct{}

func (state *childActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *actor.Restarting:
        fmt.Println("restarting")
    case *hello:
        fmt.Printf("Hello %v\n", msg.Who)
        context.Stop(context.Self())
    }
}

func newChildActor() actor.Actor {
    return &childActor{}
}
```

```go
func (state *parentActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *hello:
        props := actor.PropsFromProducer(newChildActor)
        child := context.Spawn(props)
        context.Send(child, msg)
    case *actor.Terminated:
        fmt.Println("child terminated", msg.Who)
    }
}

func newParentActor() actor.Actor {
    return &parentActor{}
}

type childActor struct{}

func (state *childActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *actor.Restarting:
        fmt.Println("restarting")
    case *hello:
        fmt.Printf("Hello %v\n", msg.Who)
        context.Stop(context.Self())
    }
}

func newChildActor() actor.Actor {
    return &childActor{}
}
```

```go
func (state *parentActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *hello:
        props := actor.PropsFromProducer(newChildActor)
        child := context.Spawn(props)
        context.Send(child, msg)
    case *actor.Terminated:
        fmt.Println("child terminated", msg.Who)
    }
}

func newParentActor() actor.Actor {
    return &parentActor{}
}

type childActor struct{}

func (state *childActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *actor.Restarting:
        fmt.Println("restarting")
    case *hello:
        fmt.Printf("Hello %v\n", msg.Who)
        context.Stop(context.Self())
    }
}

func newChildActor() actor.Actor {
    return &childActor{}
}
```

```
func (state *parentActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *hello:
        props := actor.PropsFromProducer(newChildActor)
        child := context.Spawn(props)
        context.Send(child, msg)
    case *actor.Terminated:
        fmt.Println("child terminated", msg.Who)
    }
}

func newParentActor() actor.Actor {
    return &parentActor{}
}

type childActor struct{}

func (state *childActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *actor.Restarting:
        fmt.Println("restarting")
    case *hello:
        fmt.Printf("Hello %v\n", msg.Who)
        context.Stop(context.Self())
    }
}

func newChildActor() actor.Actor {
    return &childActor{}
}
```

子アクター生成

自動的に階層構造を持つ

子アクターのアドレス

```
func (state *parentActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *hello:
        props := actor.PropsFromProducer(newChildActor)
        child := context.Spawn(props)
        context.Send(child, msg)
    case *actor.Terminated:
        fmt.Println("child terminated", msg.Who)
    }
}

func newParentActor() actor.Actor {
    return &parentActor{}
}

type childActor struct{}

func (state *childActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *actor.Restarting:
        fmt.Println("restarting")
    case *hello:
        fmt.Printf("Hello %v\n", msg.Who)
        context.Stop(context.Self())
    }
}

func newChildActor() actor.Actor {
    return &childActor{}
}
```

```go
func (state *parentActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *hello:
        props := actor.PropsFromProducer(newChildActor)
        child := context.Spawn(props)
        context.Send(child, msg)
    case *actor.Terminated:
        fmt.Println("child terminated", msg.Who)
    }
}

func newParentActor() actor.Actor {
    return &parentActor{}
}

type childActor struct{}

func (state *childActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *actor.Restarting:
        fmt.Println("restarting")
    case *hello:
        fmt.Printf("Hello %v\n", msg.Who)
        context.Stop(context.Self())
    }
}

func newChildActor() actor.Actor {
    return &childActor{}
}
```

子アクターが自分宛てのメッセージを
受け取る
（自動で振り分けられます）

```
func (state *parentActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *hello:
        props := actor.PropsFromProducer(newChildActor)
        child := context.Spawn(props)
        context.Send(child, msg)
    case *actor.Terminated:
        fmt.Println("child terminated", msg.Who)
    }
}

func newParentActor() actor.Actor {
    return &parentActor{}
}

type childActor struct{}

func (state *childActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *actor.Restarting:
        fmt.Println("restarting")
    case *hello:
        fmt.Printf("Hello %v\n", msg.Who)
        context.Stop(context.Self())
    }
}

func newChildActor() actor.Actor {
    return &childActor{}
}
```

子アクターが自分自身を停止した例

```go
func (state *parentActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *hello:
        props := actor.PropsFromProducer(newChildActor)
        child := context.Spawn(props)
        context.Send(child, msg)
    case *actor.Terminated:
        fmt.Println("child terminated", msg.Who)
    }
}

func newParentActor() actor.Actor {
    return &parentActor{}
}

type childActor struct{}

func (state *childActor) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case *actor.Restarting:
        fmt.Println("restarting")
    case *hello:
        fmt.Printf("Hello %v\n", msg.Who)
        context.Stop(context.Self())
    }
}

func newChildActor() actor.Actor {
    return &childActor{}
}
```

親アクターへ停止が通知される

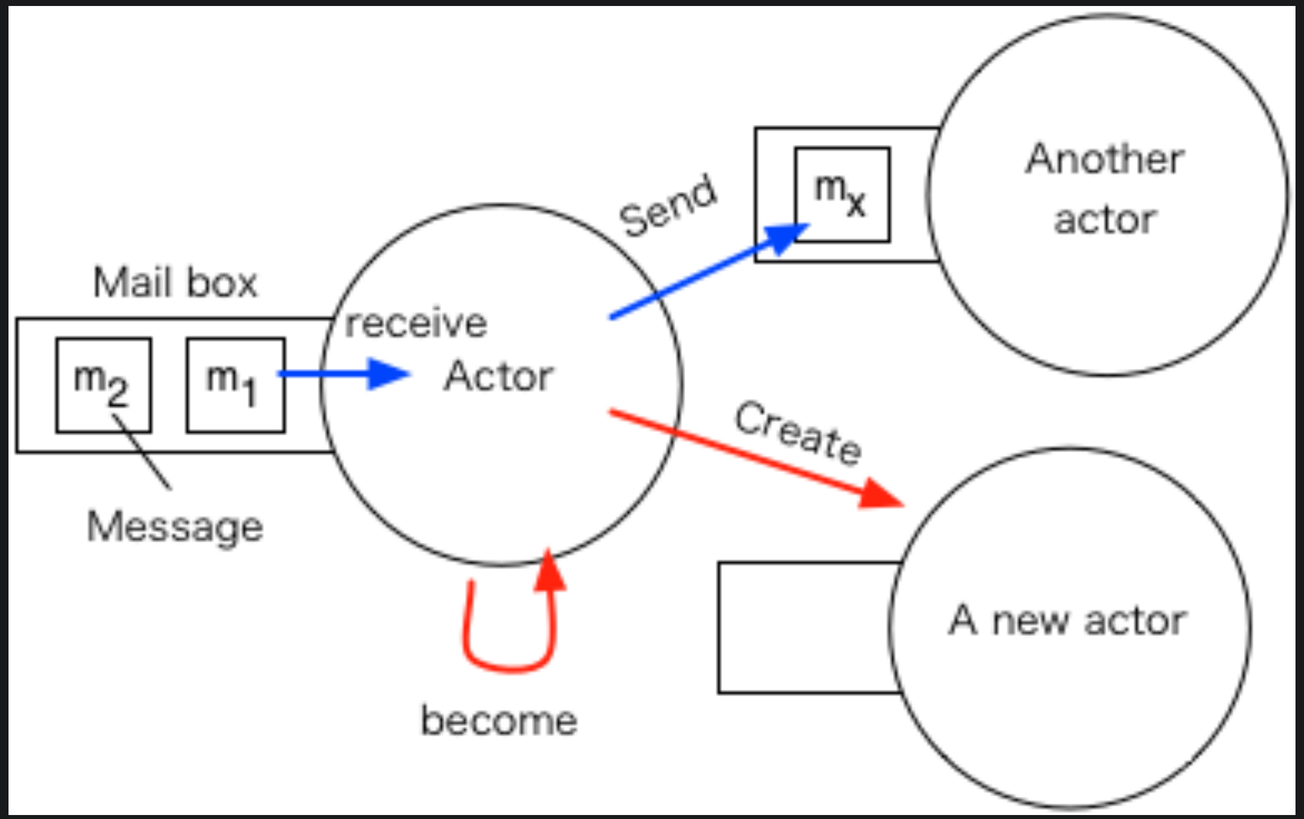
戦略

```
decider := func(reason interface{}) actor.Directive {
     fmt.Println("handling failure for child")
     return actor.StopDirective
 }
 supervisor := actor.NewOneForOneStrategy(10, time.Nanosecond, decider)
 rootContext := system.Root
 pid := rootContext.Spawn(
     actor.PropsFromProducer(newParentActor,
         actor.WithSupervisor(supervisor)))
```

```
decider := func(reason interface{}) actor.Directive {
     fmt.Println("handling failure for child")
     return actor.StopDirective
 }
 supervisor := actor.NewOneForOneStrategy(10, time.Nanosecond, decider)
 rootContext := system.Root
 pid := rootContext.Spawn(
     actor.PropsFromProducer(newParentActor,
         actor.WithSupervisor(supervisor)))
```

子アクターが失敗した場合、
同階層にいる子アクターのうち、
対象のものだけ停止させる例

他、状態変化やルーティングなど


Another
actor
Send
m_x
Mail box
receive
Actor
m_2
m_1
Message
Create
A new actor
become

従来の仕組みにどう活かせられるか？

アクターと伝統的な構造

- アクターモデルと従来のアプリケーション構造は共存可能

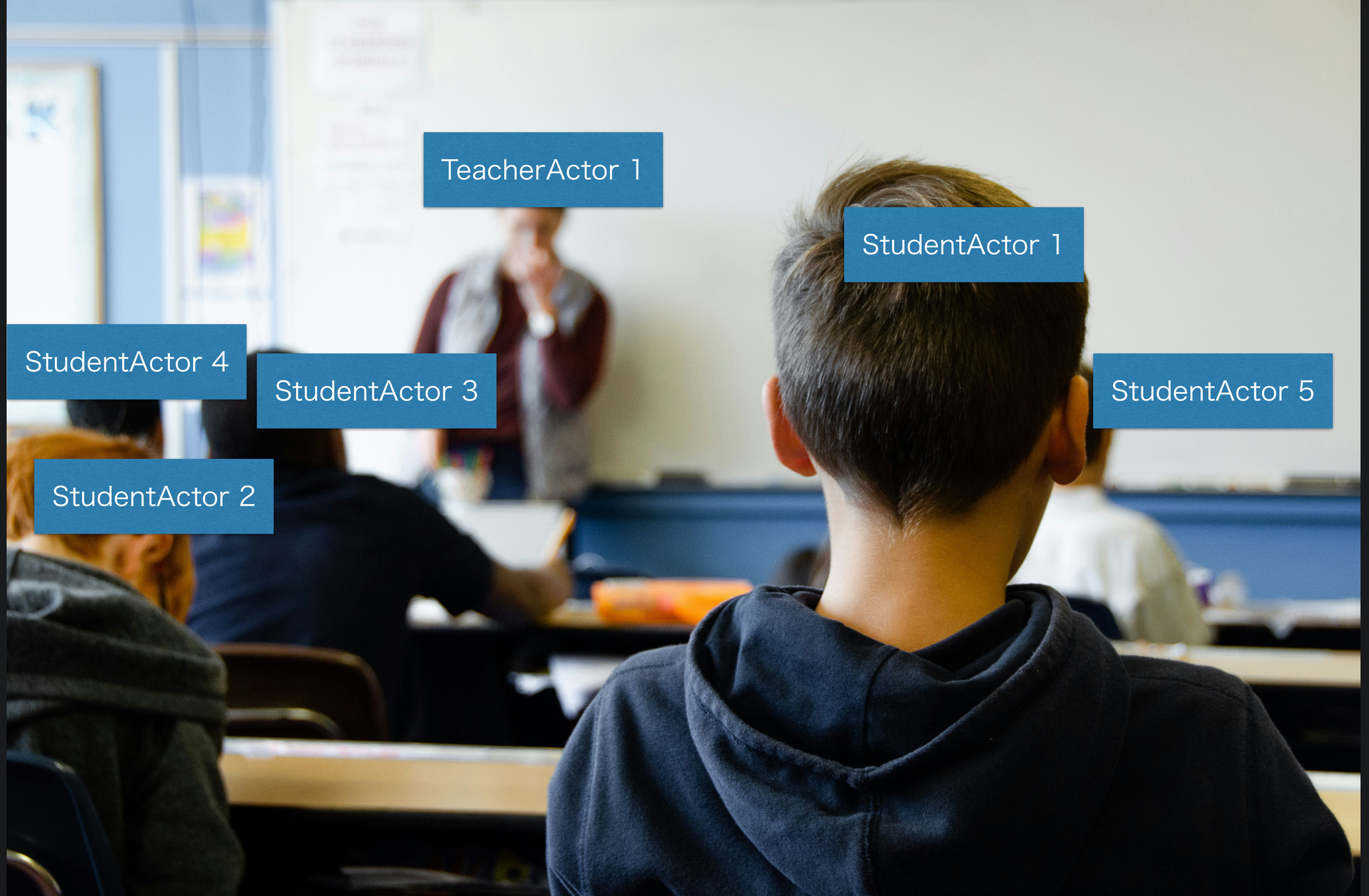
- 戦術パターンなどでは
  アクターがエンティティになり状態をもつ

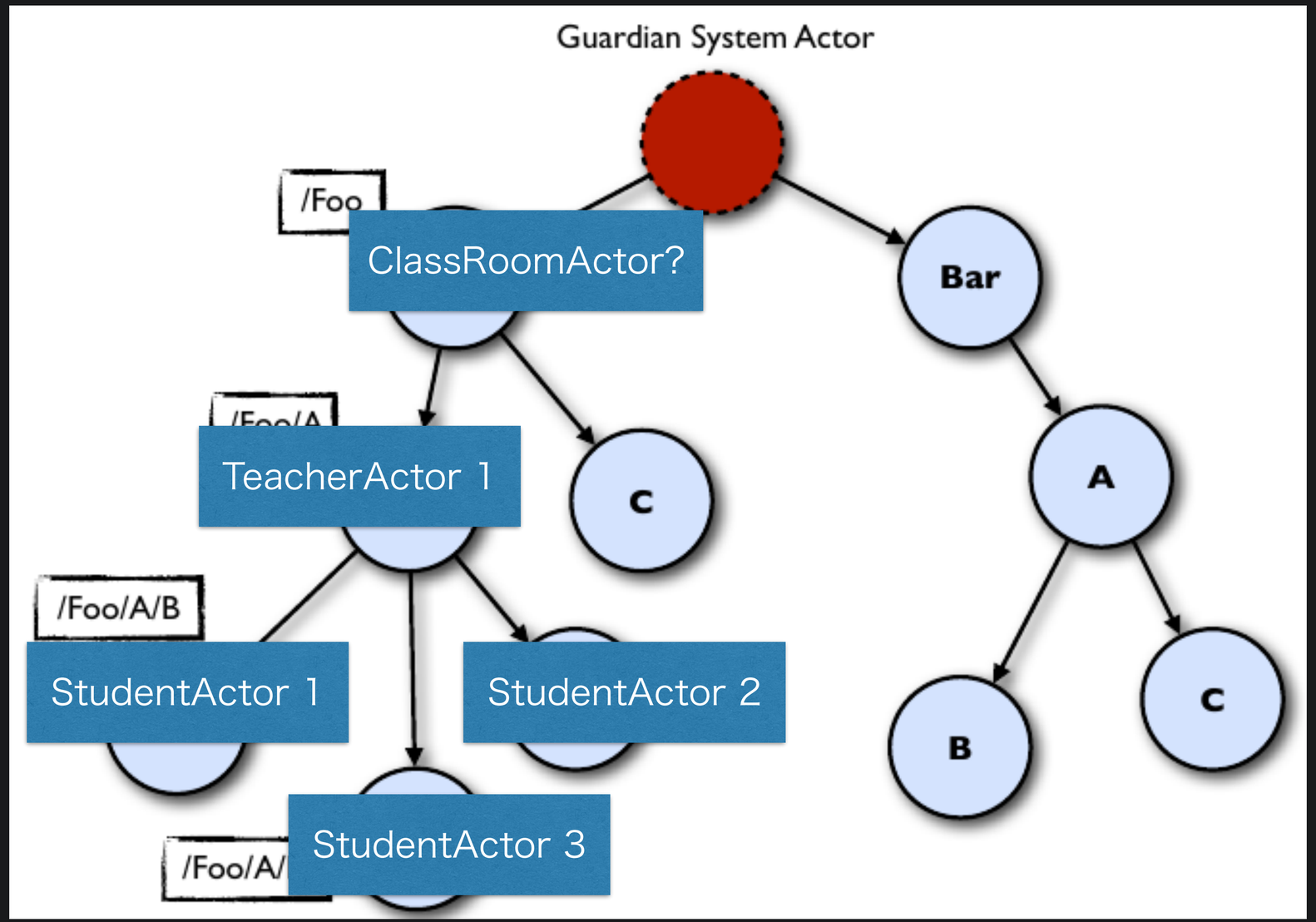
- アクターの状態変化（イベント）を保存することで
  ビューを組み立てることができる（Read Model Update）

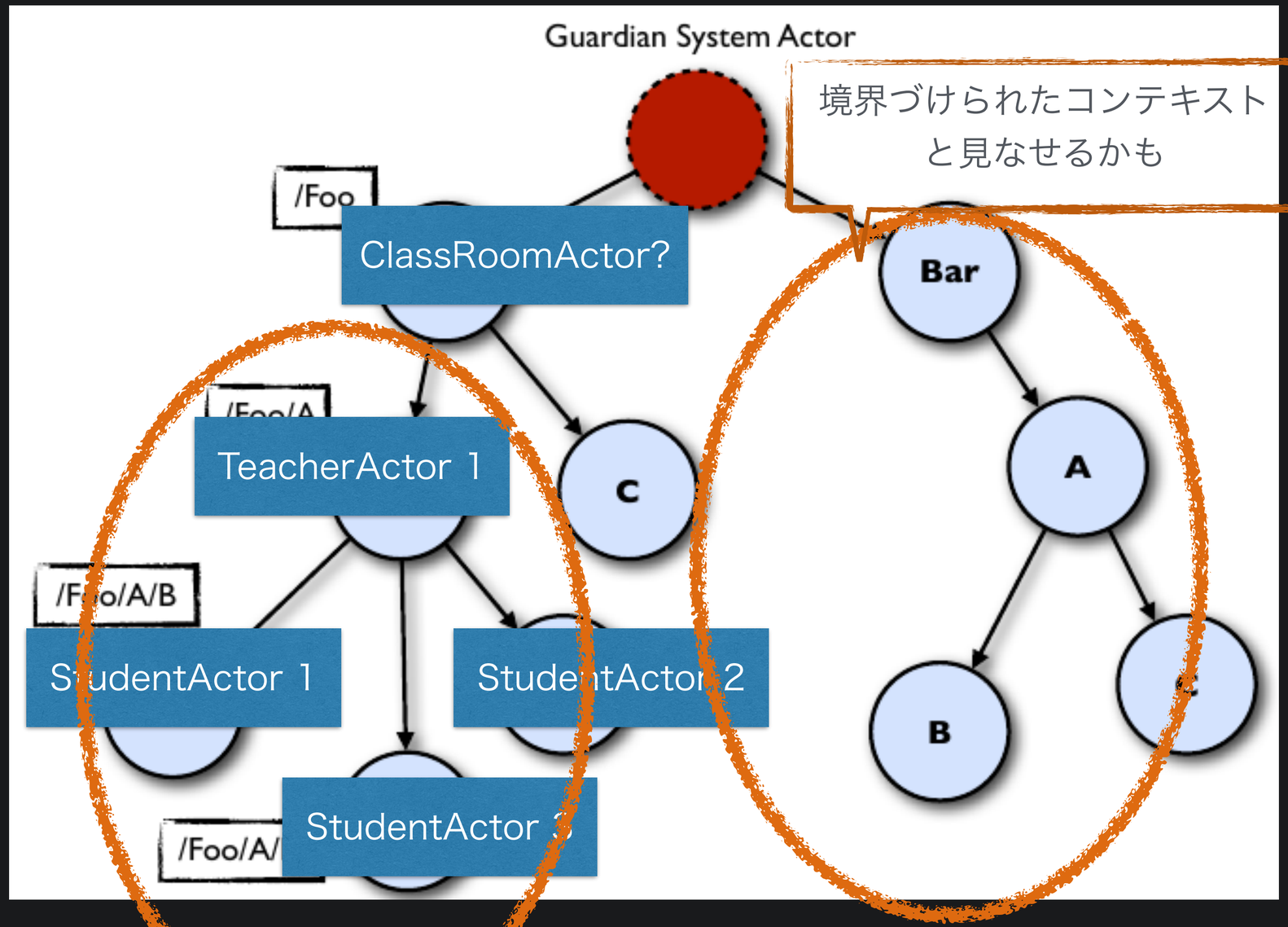
学校のテスト例

- 先生が生徒に対して さんすうのテストを実施
- 生徒はそれぞれ個として問題を解く
- 先生はテスト中に見守るなどの行動
- 全て答えたらテスト用紙を提出する


StudentActor 1
StudentActor 4
StudentActor 3
StudentActor 5
StudentActor 2


TeacherActor 1
StudentActor 1
StudentActor 4
StudentActor 3
StudentActor 5
StudentActor 2


Guardian System Actor
/Foo
ClassRoomActor?
Bar
TeacherActor 1
C
A
/Foo/A/B
StudentActor 1
StudentActor 2
B
C
StudentActor 3
/Foo/A/


Guardian System Actor
境界づけられたコンテキスト
と見なせるかも
/Foo
ClassRoomActor?
Bar
TeacherActor 1
A
C
/Foo/A/B
StudentActor 1
StudentActor 2
B
StudentActor 3
/Foo/A/

全てはアクター

- 先生や生徒もアクター

- 親アクターは集約とみなせる

- １階層上がると抽象度などが変わり、
  ルートが変わるとコンテキストが変わると考えることができる

状態変化

- 生徒の状態が変わった

  先生がどの生徒のテストをうけとった など

- アクターの状態を保管することで事実を残す

- 事実の積み重ねがビューを作る

```
func (u *Cart) Receive(context actor.Context) {
    defer context.Poison(context.Self())
    switch msg := context.Message().(type) {
    case *persistence.RequestSnapshot:
        u.PersistSnapshot(u.state)
    case *persistence.ReplayComplete:
        // リプレイが完了したら内部状態を変更する

        context.Logger
            fmt.Sprintf                                          u.state))
    case *command.AddI
        if u.IsStateEx
            context.Se                                           eady exists"})
            return
        }
        // 略

        u.persistence(context, ev)
        // xxx生成イベントをイベントストリームへ

        context.Send(u.stream, ev)
    case *event.ItemAdded:
        if msg.String() != "" {
            // event がリプレイされた場合は状態を更新する

            u.state = msg
            u.sendToReadModelUpdater(context, msg)
        }
    }
}
```

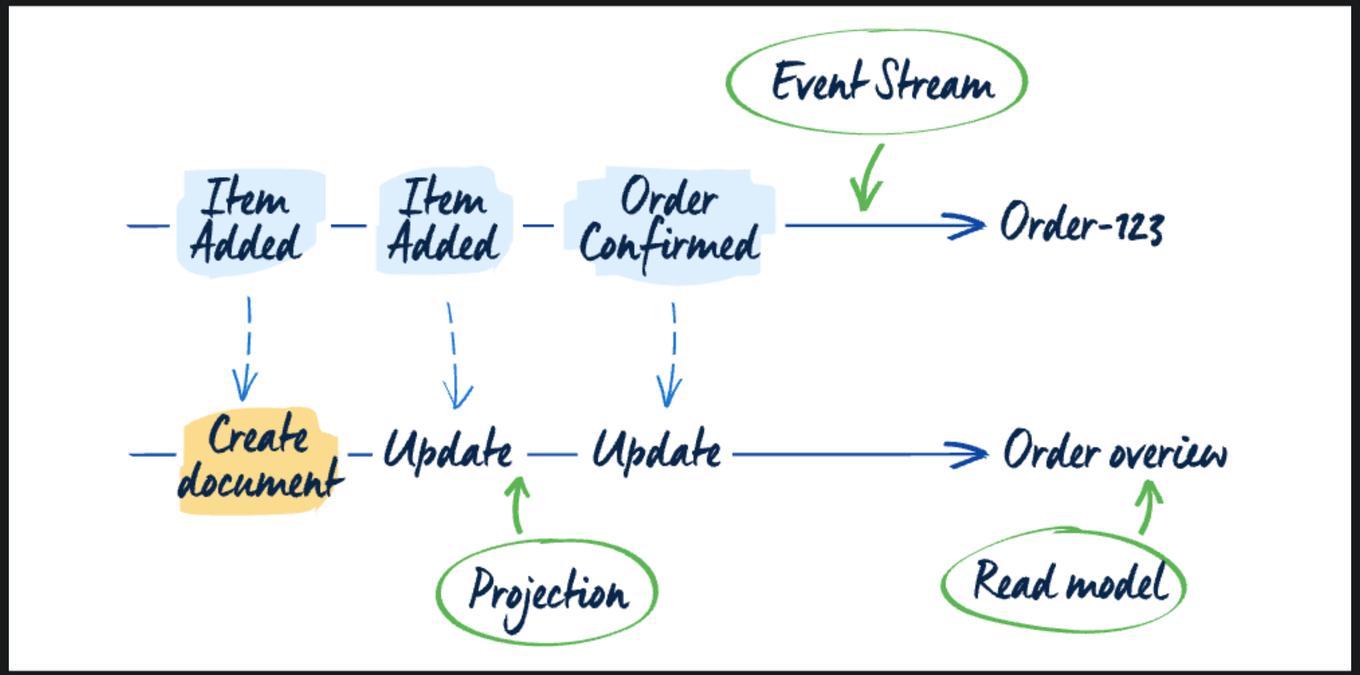
コマンドを受取、
自アクターの状態を変更
状態変更をイベントとして永続化

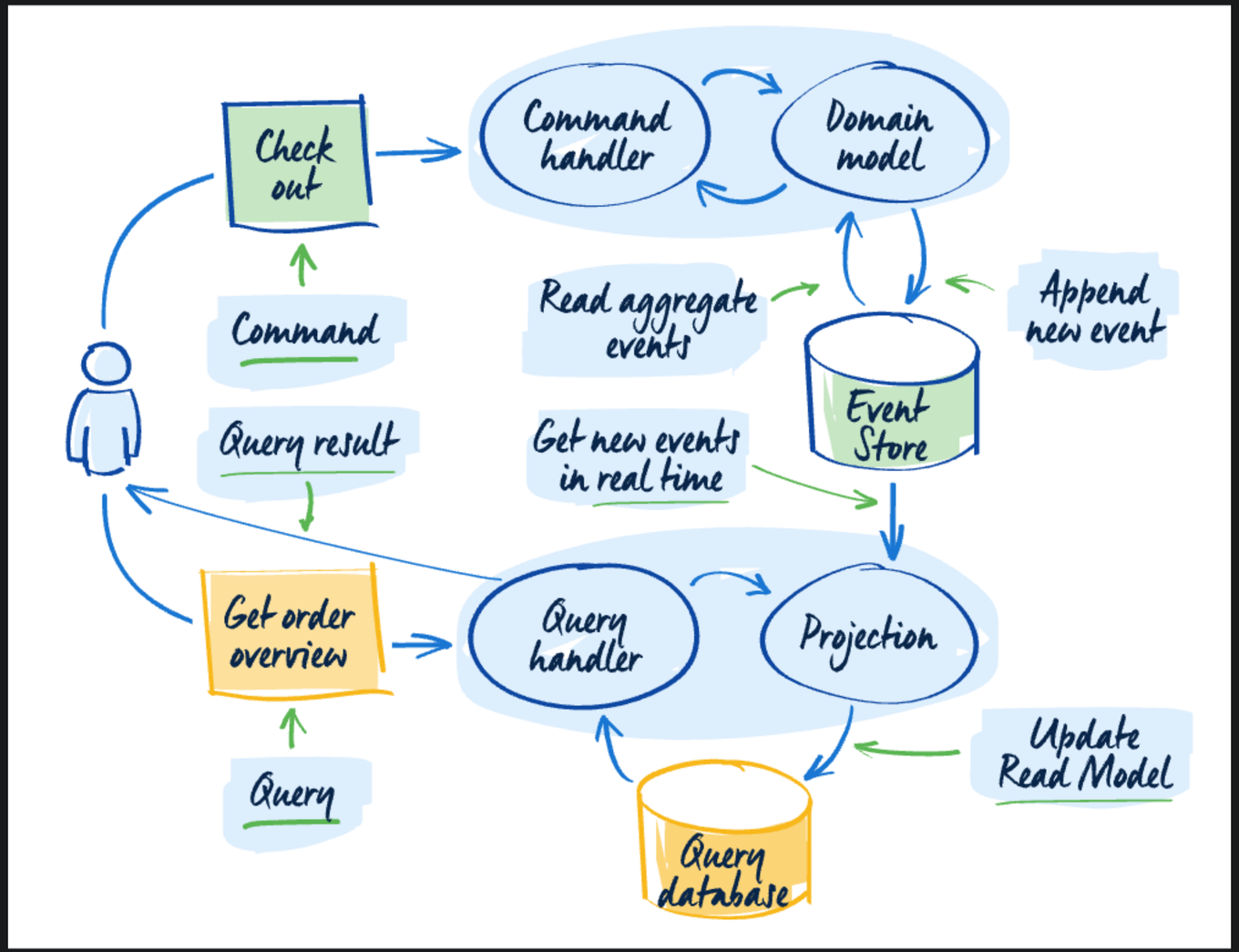
```
func (u *Cart) Receive(context actor.Context) {
    defer context.Poison(context.Self())
    switch msg := context.Message().(type) {
    case *persistence.RequestSnapshot:
        u.PersistSnapshot(u.state)
    case *persistence.ReplayComplete:
        // リプレイが完了したら内部状態を変更する

        context.Logger().Info(
            fmt.Sprintf("replay completed, internal state changed to '%v'", u.state))
    case *command.AddItem:
        if u.IsStateExists(m
            context.Send(u.s                                    eady exists"})
            return
        }
        // 略

        u.persistence(cont    , ev)
        // xxx生成イベントをイベントストリームへ

        context.Send(u.stream, ev)
    case *event.ItemAdded:
        if msg.String() != "" {
            // event がリプレイされた場合は状態を更新する

            u.state = msg
            u.sendToReadModelUpdater(context, msg)
        }
    }
}
```

リードモデル更新ハンドラ（アクター）へ

```go
func (u *Cart) Receive(context actor.Context) {
    defer context.Poison(context.Self())
    switch msg := context.Message().(type) {
    case *persistence.RequestSnapshot:
        u.PersistSnapshot(u.state)
    case *persistence.ReplayComplete:
        // リプレイが完了したら内部状態を変更する

        context.Logger().Info(
            fmt.Sprintf("replay completed, internal state changed to '%v'", u.state))
    case *command.AddItem:
        if u.IsStateExists(msg.Name) {
            context.Send(u.stream, &message.AddItemError{Message: "item already exists"})
            return
        }
        // 略

        u.persistence(context, ev)
        // xxx生成イベントをイベントストリームへ

        context.Send(u.stream, ev)
    case *event.ItemAdded:
        if msg.String() != "" {
            // event がリプレイされた

            u.state = msg
            u.sendToReadModelUpdat
        }
    }
}
```

アクター生成（再生成、リスタートなど）時
過去の状態があれば、
最初のイベントから受信し、状態を復元


Event Stream
Item Added
Item Added
Order Confirmed
Order-123
Create document
Update
Update
Order overiew
Projection
Read model


Check out
Command handler
Domain model
Command
Read aggregate events
Append new event
Event Store
Query result
Get new events in real time
Get order overview
Query handler
Projection
Query
Update Read Model
Query database

独立して動いている

- クラウドを利用することでシステムは分散

- 分散しているものを同期的に扱うのではなく

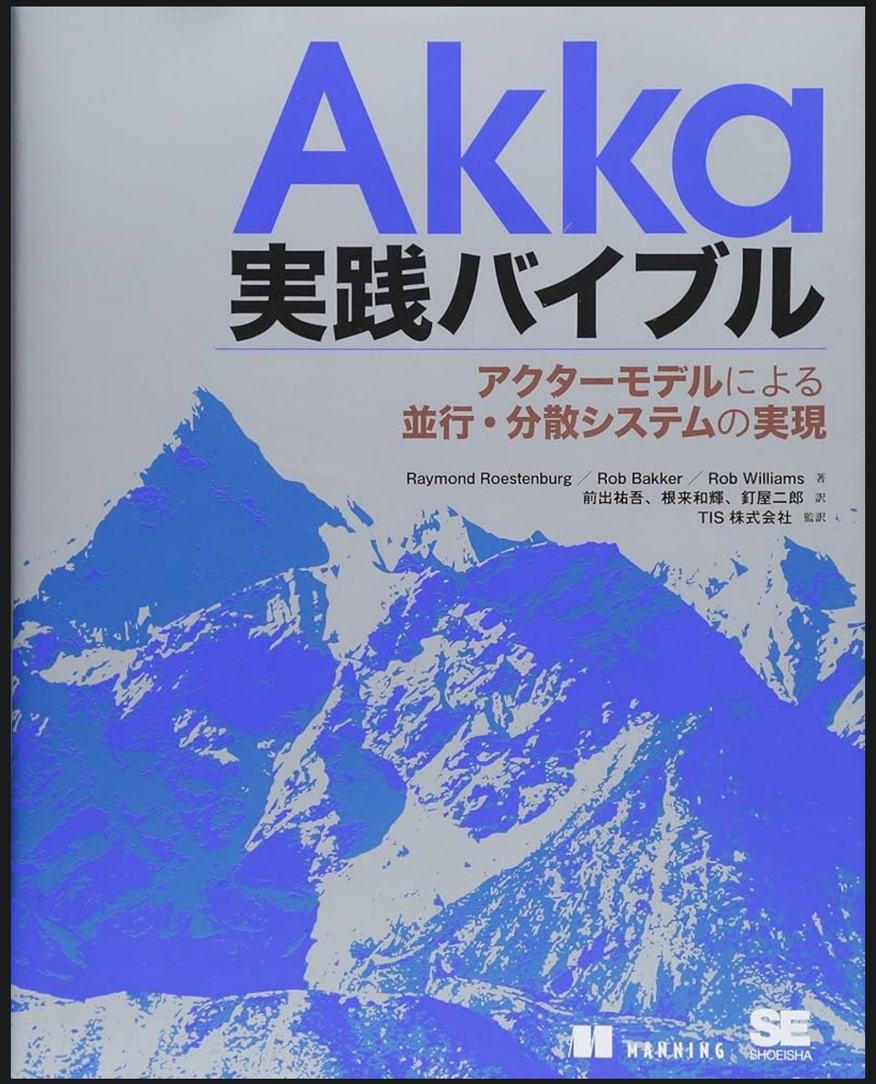
  非同期で並行、すべて個として動作していると捉えると？

- アクターモデルについて再考してみるのはいかがでしょうか？

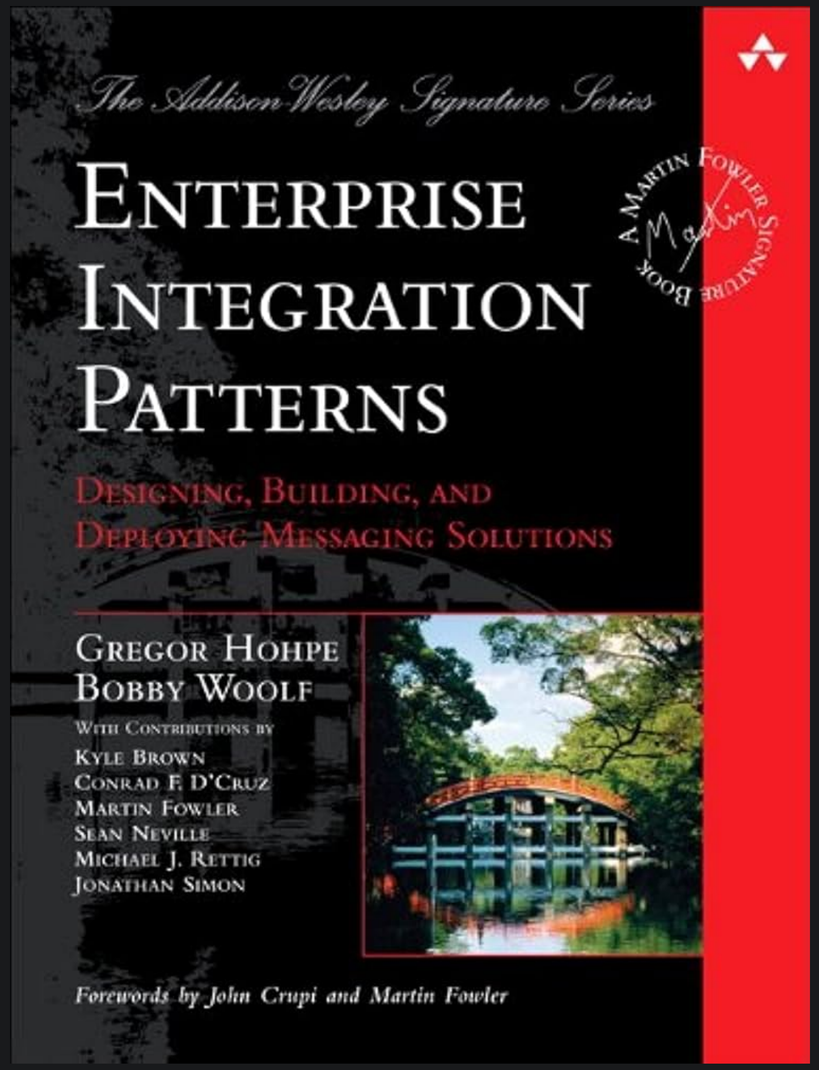
導入できるツールキット

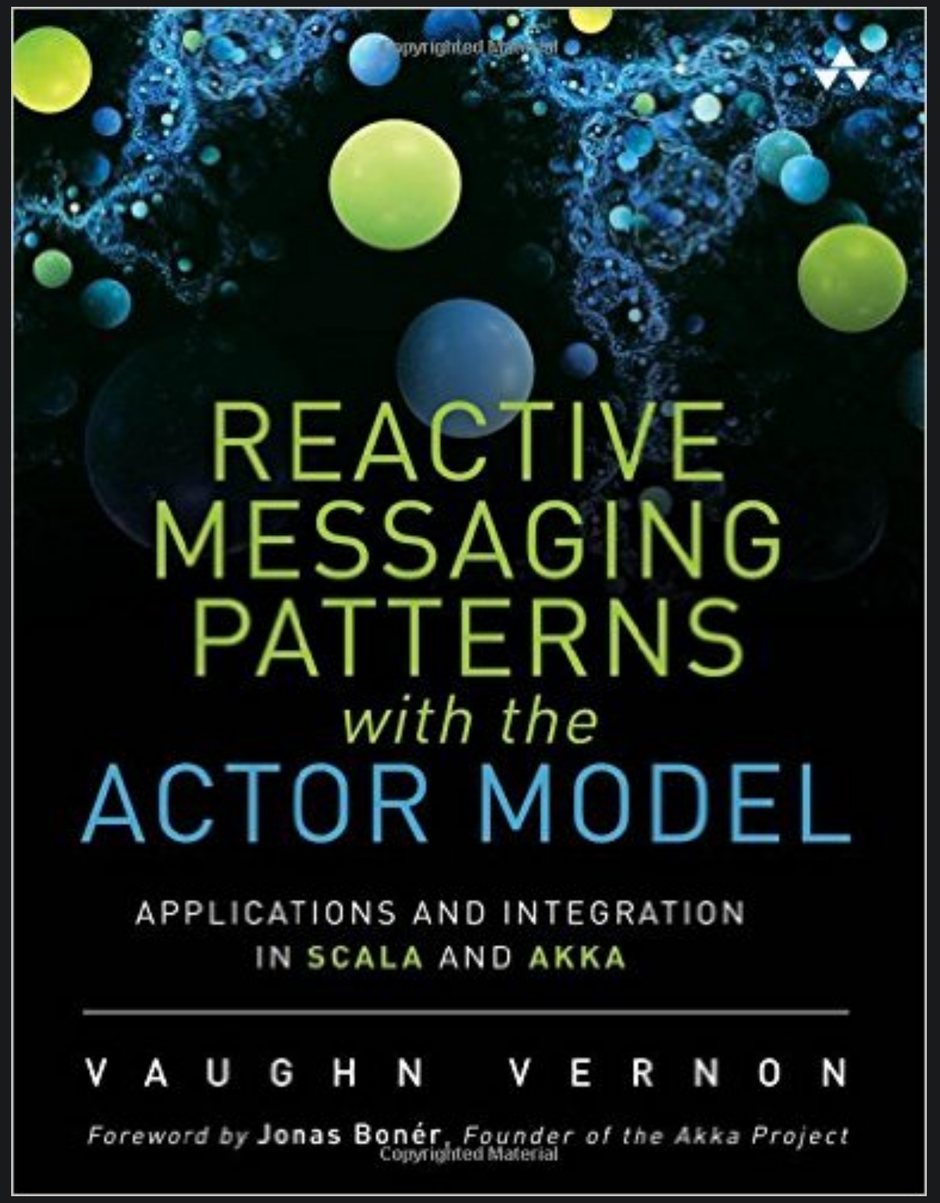
- Akka / Pekko (Java, Scala)
- Erlang OTP
- Proto Actor (Go, .NET)
- ergo (Go)
- Phluxor(PHP) ほか色々

アクターモデルについて理解するために



より理解するために


The Addison Wesley Signature Series
A Martin Fowler Signature Book
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
Gregor Hohpe
Bobby Woolf
With Contributions by
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon
Forewords by John Crupi and Martin Fowler


Copyrighted Material
REACTIVE MESSAGING PATTERNS with the ACTOR MODEL
APPLICATIONS AND INTEGRATION IN SCALA AND AKKA
VAUGHN VERNON
Foreword by Jonas Bonér, Founder of the Akka Project
Copyrighted Material

さいごに

- プログラミングパラダイムがちょっと違うもの
- 聞くだけではなかなか難しいので、ぜひ触ってみてください
- 理解できると普段のプログラミング、モデリングにも活かせる！