

Pony concurrency built into the type system

@matsu_chara

2016/3/20 kbkz_tech#9

スライドが81ページあるので

爆速でやります。

今日のスライド

https://www.slideshare.net/matsu_chara/pony-concurrency-built-into-the-type-system-59778750

タイトルはパクリ

**Debasish Ghosh**
@debasishg

⚙️ フォロー中

Yes! .. "concurrency built into the type system
[..] that statically ensures freedom from data
races" **#ponylang**

Markus Fix @lispmeister
@debasishg @Amsterdandy This: blog.acolyer.org/2016/02/17/den...

🌐 翻訳を表示

3
リツイート

12
いいね



16:50 - 2016年2月22日

<https://twitter.com/debasishg/status/701675442793615360>

自己紹介



- @matsu_chara
- ドワンゴ一年目
- 好きな言語: PHP
- ブログ: <http://matsu-chara.hatenablog.com/>
- Kafkaを使ったマイクロサービス基盤作り

今日は

最強言語PHPの話

ではなく

超高速で型安全な
アクターモデル言語Ponyの話をします

Ponyとは

- 2015年4月に0.1 がリリースされた言語
- Causalityというロンドンの会社が開発。
- 元はImperial Collegeで研究されていた。
 - 論文 A String of Ponies Transparent Distributed Programming with Actors
s.blessing 2013

商用サポートあり

Value Proposition

... OR HOW TO BRING BACK THE FREE LUNCH.



Pony Language

[read more](#)



Pony Programming Trainings

[book](#)



Pony Commercial Support

[get in touch](#)

<http://www.causality.io/>

気合入ってる

で、一体どんな言語なんだ・・・

Ponyとは

- アクターモデルに特化
- 速さと(型による)安全性を追求
 - pony philosophy
 - capability-secure

さらに

distributed pony

- 真の狙いは並行処理ではなく分散システムの構築にある
- まだ実装は未公開(論文はある)

論文での提案手法リスト <http://www.doc.ic.ac.uk/teaching/distinguished-projects/2013/s.blessing.pdf>

This thesis provides the following contributions:

- An algorithm with constant space complexity for hierarchical work stealing in tree networks.
- A joining algorithm to add new slave nodes to a cluster of *Ponies* at runtime.
- Deferred reference counting in distributed systems.
- Distributed hierarchical garbage collection of actors.
- Causal message delivery in distributed systems with no software overhead.
- Termination of actor applications based on distributed quiescence.

まだ出てないので今日は
concurrent ponyの話に限定

Ponyとは

- アクターモデルに特化
 - 速さと(型による)安全性を追求
- pony philosophy
- capability-secure

Pony哲学(抜粋)

<https://github.com/ponylang/ponyc/wiki/Philosophy>

- Fully type safe. There is no "trust me, I know what I'm doing" coercion.
 - 型安全!
- Fully memory safe. There is no "this random number is really a pointer, honest."
 - 危険なポインタ操作とかは無し!
- No crashes. A program that compiles should never crash (although it may hang or do something unintended).
 - コンパイル通ったらクラッシュもハングもしない!

コンパイルが通ったら絶対に
クラッシュさせないという
熱い気持ち

Ponyとは(再掲)

- アクターモデルに特化
- 速さと(型による)安全性を追求
 - pony philosophy
 - capability-secure

capability secureの前に
アクターモデルについて

アクターモデル(かなり省略版)

Actorを用意



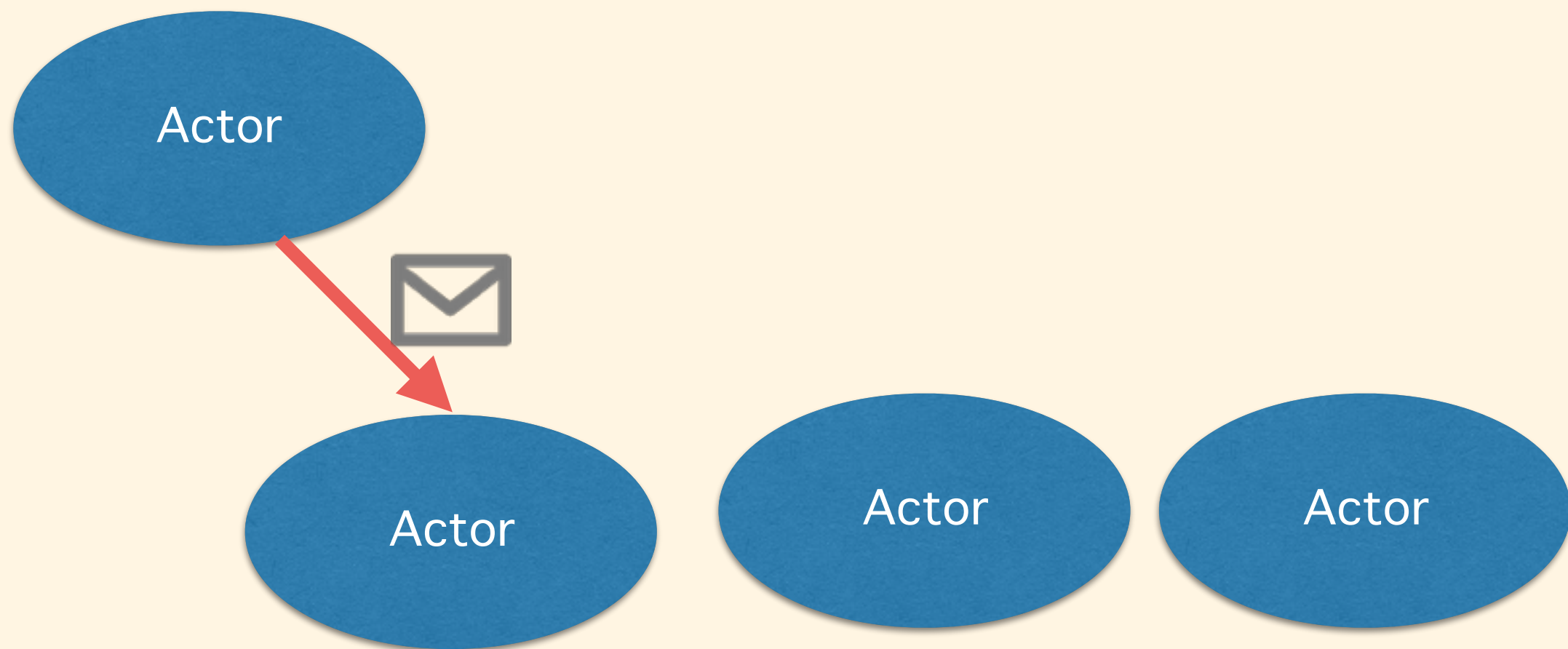
Actor

Actor

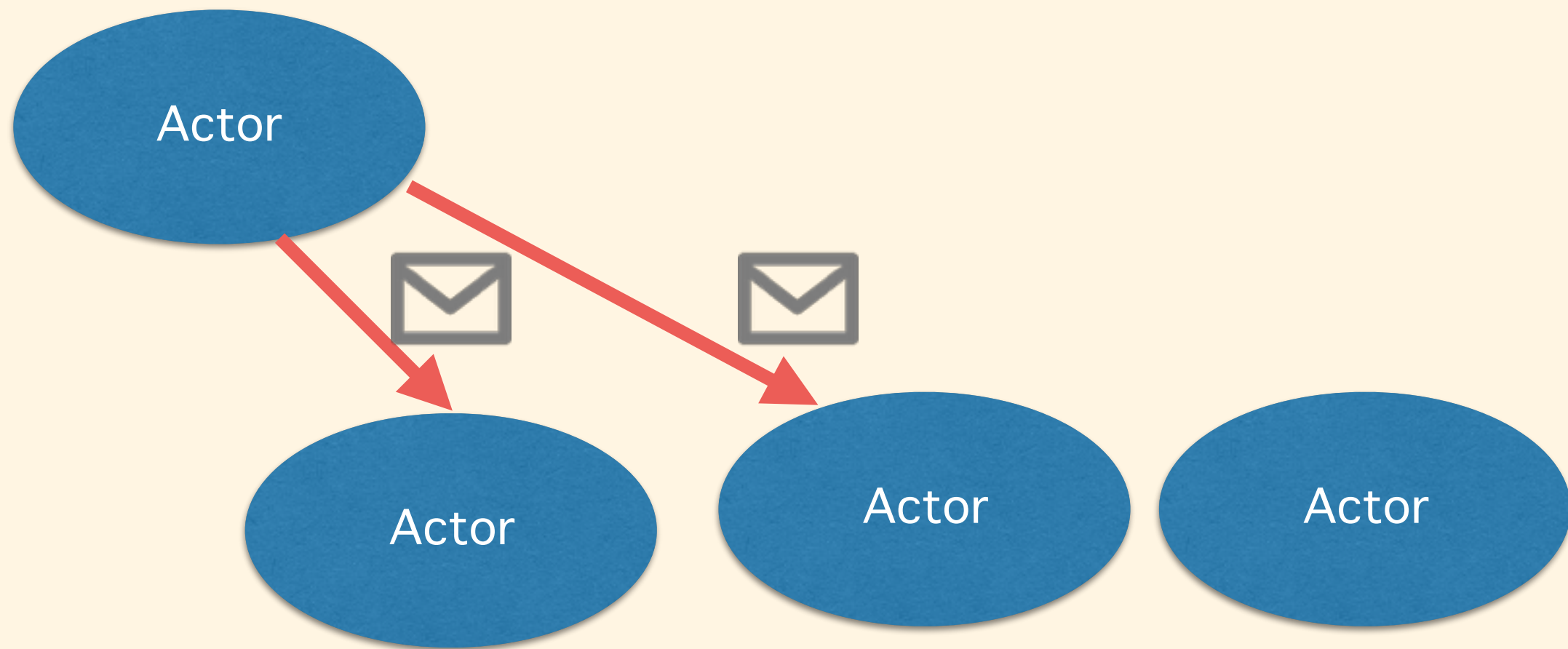
Actor

Actor

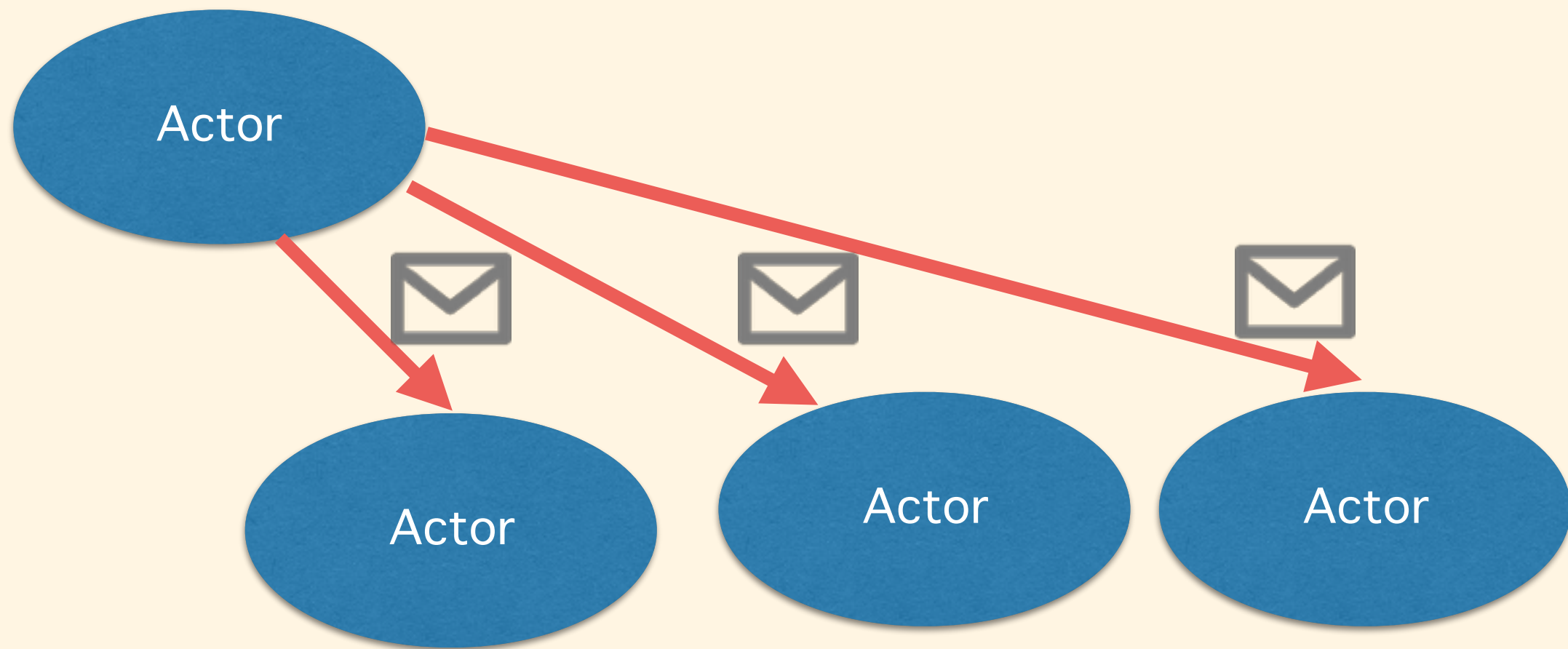
各アクターにメッセージを配信



各アクターにメッセージを配信

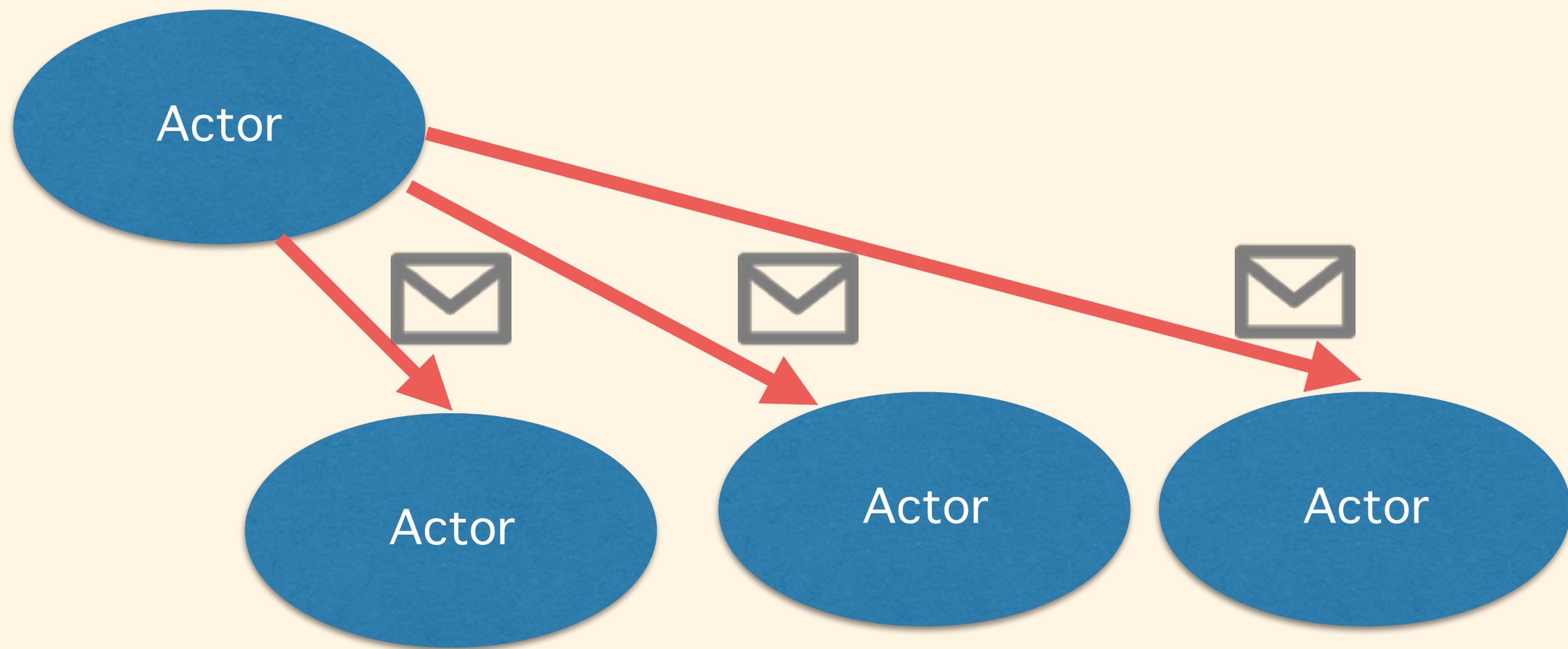


各アクターにメッセージを配信



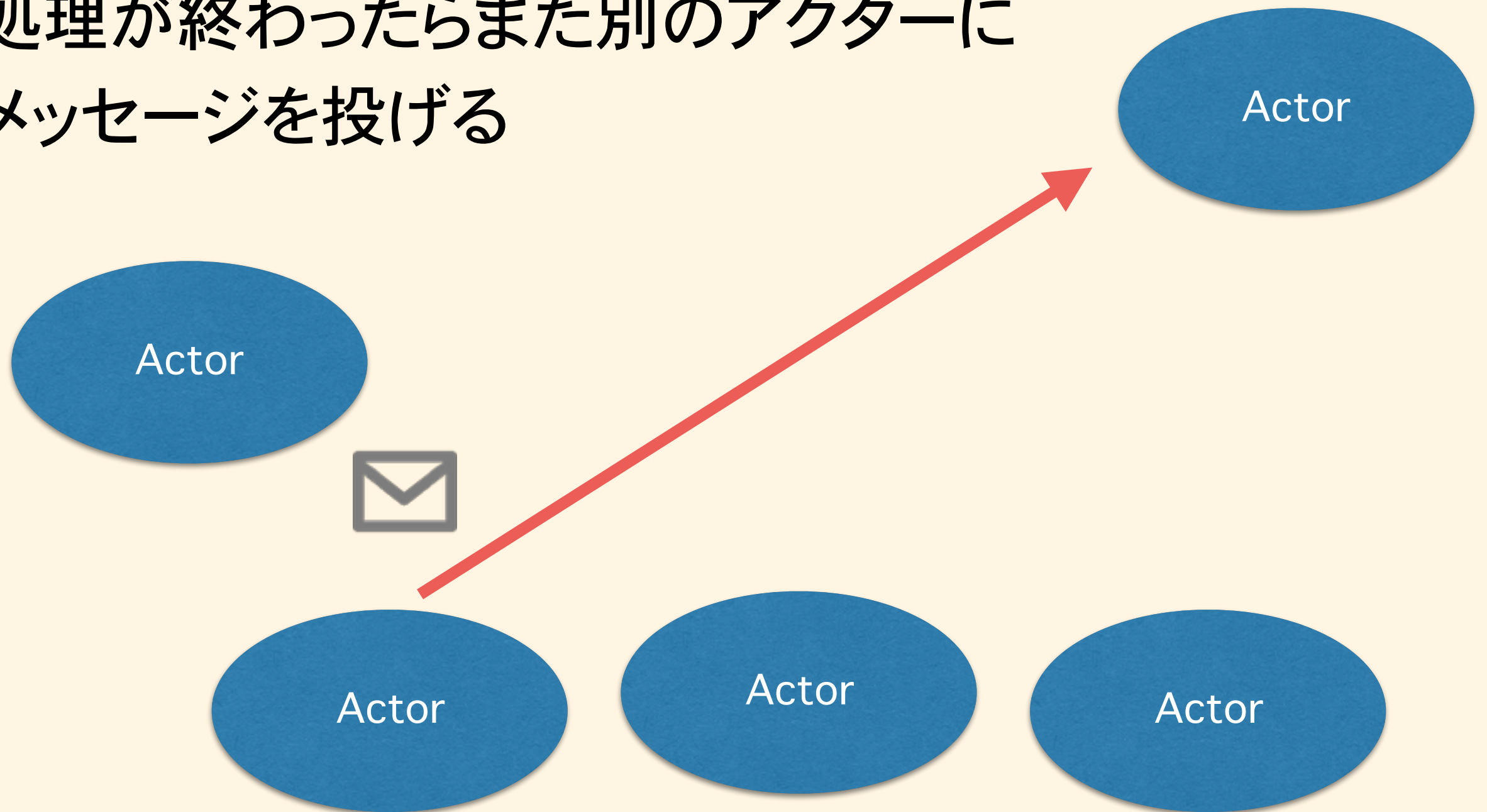
メッセージを処理

一つのアクターはシングルスレッドで
メッセージを1つずつ処理



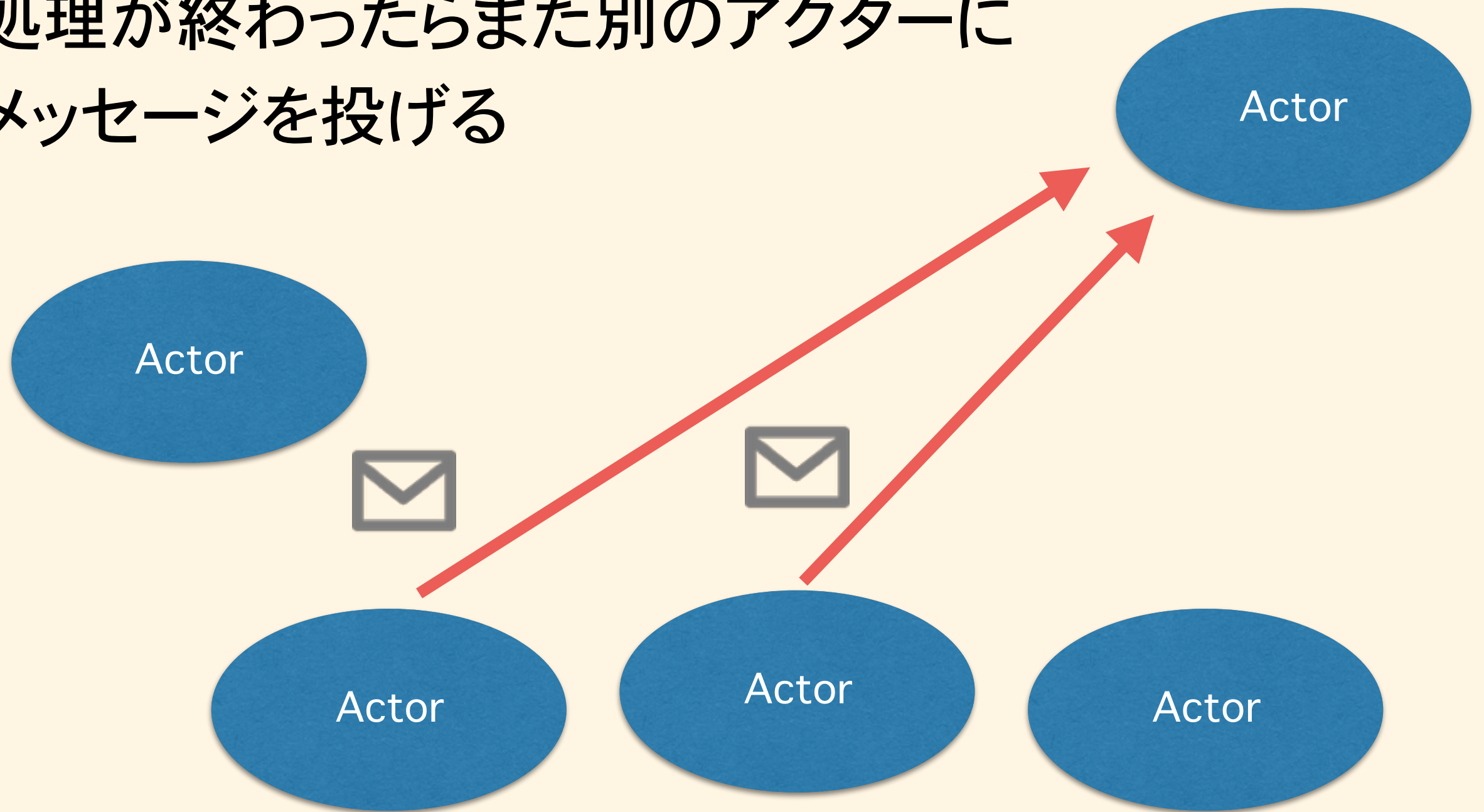
メッセージドリブン

処理が終わったらまた別のアクターに
メッセージを投げる



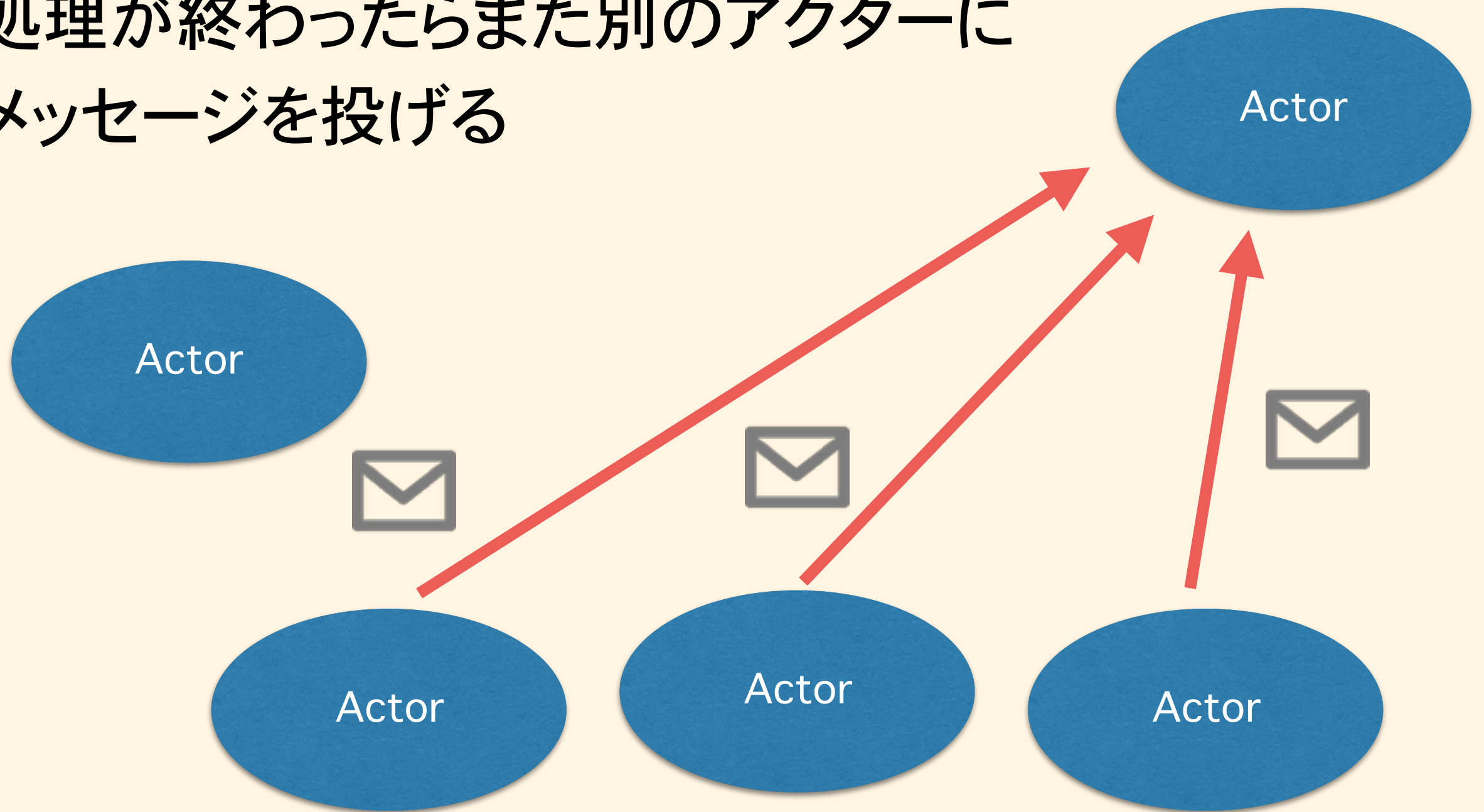
メッセージドリブン

処理が終わったらまた別のアクターに
メッセージを投げる



メッセージドリブン

処理が終わったらまた別のアクターに
メッセージを投げる



shared nothing is 神

- 各アクターでデータを共有しないからデータ競合しない！
- ロックも必要ないからデッドロックのリスクもない！

アクターモデルで書ける言語

- 色々ある
- Erlang/Elixir, Scala, C++,...
- Erlangは神
 - 軽量スレッドに最適化されたVM
 - ノンブロッキングな標準ライブラリ群

一方で

shared nothing の課題

- Erlangなどの言語ではメッセージをコピーして渡す必要がある。
- コピーしなくても良い物も含まれるので当然オーバーヘッドがある。

shared nothing の課題

- Scala/akkaではアクターで書きつつshared memoryっぽくデータを共有できる
- しかしデータを共有したが最後、データ競合・デッドロックのリスクが発生してしまう
- アクターモデルとは何だったのか・・・

全部コンパイル時に保証して欲しい・・・

- 「shared memory方式の効率の良さ」と「shared nothingの安全性」を両立したい。
- data-raceとdead-lockが二大天敵
- shared memoryなんだけどコンパイル時にdata-raceとdead-lockが無いことを保証してくれる言語があればできそう・・・

というわけで
Ponyの話に戻ります

Ponyとは(再掲)

- アクターモデルに特化
- 速さと(型による)安全性を追求
 - pony philosophy
 - capability-secure

capabilities-secure

以下の要素を満たす(という定義)<http://tutorial.ponylang.org/>

- 型安全
- メモリー安全
- 例外安全
- デッドロックなし
- データ競合なし

capabilities-secure

以下の要素を満たす(という定義)<http://tutorial.ponylang.org/>

- 型安全
- メモリー安全
- 例外安全
- デッドロックなし
- データ競合なし ←コンパイル時に保障

データ競合させないための
型修飾子

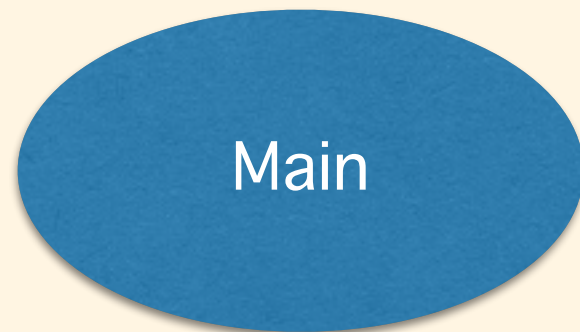
object capabilities

- オブジェクトへのアクセス権を細かく分割して型で表現
- 全6種
- 例) 複数アクターから同時にWriteできる参照があったらコンパイルエラー

論文 <http://www.ponylang.org/papers/fast-cheap.pdf>

object capabilities

例



object capabilities

例

状態を初期化



Main

A blue oval with a subtle drop shadow, containing the word "Main" in white text.

Worker

A blue oval with a subtle drop shadow, containing the word "Worker" in white text.

object capabilities

例

状態を初期化



object capabilities

```
1 actor Main
2   new create(env: Env) =>
3     let s: State iso = State(1)
4     s.update(2)
5
6     // compile error
7     // reference to iso must be unique
8     // let reader: State iso = s
9
10    let w = Worker
11    w.reload(consume s)
12
13    // compile error
14    // can't use a consumed local...
15    // s.update(3)
16
17 actor Worker
18   var state: State iso
19
20   new create() =>
21     state = State(0)
22
23   be reload(state': State iso) =>
24     state = consume state'
25
26 class State
27   var _i: U32
28
29   new iso create(i': U32) =>
30     _i = i'
31
32   fun ref update(i': U32) =>
33     _i = i'
34
```

object capabilities

```
1 actor Main
2   new create(env: Env) =>
3     let s: State iso = State(1)
4     s.update(2)
5
6   // compile error
7   // reference to iso must be unique
8   // let reader: State iso = s
9
10  let w = Worker
11  w.reload(consume s)
12
13  // compile error
14  // can't use a consumed local...
15  // s.update(3)
16
17 actor Worker
18   var state: State iso
19
20   new create() =>
21     state = State(0)
22
23   be reload(state': State iso) =>
24     state = consume state'
25
26 class State
27   var _i: U32
28
29   new iso create(i': U32) =>
30     _i = i'
31
32   fun ref update(i': U32) =>
33     _i = i'
34
```

文字サイズが小さいので拡大

object capabilities

```
26 class State
27   var _i: U32
28
29   new iso create(i': U32) =>
30     _i = i'
31
32   fun ref update(i': U32) =>
33     _i = i'
34
```

new iso createという
メソッド定義により
作成されたインスタンスに
isoという型修飾子が
ついて返ってくる
isoについては後述

object capabilities

```
26 class State
27     var _i: U32
28
29     new iso create(i': U32) =>
30         _i = i'
31
32     fun ref update(i': U32) =>
33         _i = i'
34
```

fun refとすると
状態を変更するメソッドが
定義できる

object capabilities

```
26 class State
27     var _i: U32
28
29     new iso create(i': U32) =>
30         _i = i'
31
32     fun update(i': U32) =>
33         _i = i'
34
```

refを忘れて
状態を変更すると
コンパイルエラー

iso: read/writeユニークな参照

```
1 actor Main
2   new create(env: Env) =>
3     let s: State iso = State(1)
4     s.update(2)
5
6   // compile error
7   // reference to iso must be unique
8   // let reader: State iso = s
9
10  let w = Worker
11  w.reload(consume s)
12
13  // compile error
14  // can't use a consumed local...
15  // s.update(3)
```

```
17 actor Worker
18   var state: State iso
19
20   new create() =>
21     state = State(0)
22
23   be reload(state': State iso) =>
24     state = consume state'
```

```
26 class State
27   var _i: U32
28
29   new iso create(i': U32) =>
30     _i = i'
31
32   fun ref update(i': U32) =>
33     _i = i'
```

```
34
```

iso: read/writeユニークな参照

```
16
17 actor Worker
18   var state: State iso
19
20   new create() =>
21     state = State(0)
22
23   be reload(state': State iso) =>
24     state = consume state'
25
```

isoを受け取るメソッド
(正確にはbehavior)

iso: read/writeユニークな参照

- 他の参照が無いので自由に書き換え可能
- 参照を破棄することで、他アクターに安全にread/write権限を渡すことが出来る

iso: read/writeユニークな参照

```
1 actor Main
2   new create(env: Env) =>
3     let s: State iso = State(1)
4     s.update(2)
5
6     // compile error
7     // reference to iso must be unique
8     // let reader: State iso = s
9
10    let w = Worker
11    w.reload(consume s)
12
13    // compile error
14    // can't use a consumed local...
15    // s.update(3)
```

```
17 actor Worker
18   var state: State iso
19
20   new create() =>
21     state = State(0)
22
23   be reload(state': State iso) =>
24     state = consume state'
25
26 class State
27   var _i: U32
28
29   new iso create(i': U32) =>
30     _i = i'
31
32   fun ref update(i': U32) =>
33     _i = i'
```

34

iso: read/writeユニークな参照

```
1 actor Main
2   new create(env: Env) =>
3     let s: State iso = State(1)
4     s.update(2)
5
6     // compile error
7     // reference to iso must be unique
8     // let reader: State iso = s
9
10    let w = Worker
11    w.reload(consume s)
12
13    // compile error
14    // can't use a consumed local...
15    // s.update(3)
16
```

isoな参照

型は推論されるので
書かなくてもOK

iso: read/writeユニークな参照

```
1 actor Main
2   new create(env: Env) =>
3     let s: State iso = State(1)
4     s.update(2)
5
6     // compile error
7     // reference to iso must be unique
8     // let reader: State iso = s
9
10    let w = Worker
11    w.reload(consume s)
12
13    // compile error
14    // can't use a consumed local...
15    // s.update(3)
16
```

consumeで参照を破棄
&
他アクターに渡す

iso: read/writeユニークな参照

```
1  actor Main
2    new create(env: Env) =>
3      let s: State iso = State(1)
4      s.update(2)
5
6      // compile error
7      // reference to iso must be unique
8      let reader: State iso = s
9
10     let w = Worker
11     w.reload(consume s)
12
13     // compile error
14     // cant't use a consumed local...
15     // s.update(3)
```

isoな参照を
増やそうとすると
コンパイルエラー

iso: read/writeユニークな参照

```
1  actor Main
2    new create(env: Env) =>
3      let s: State iso = State(1)
4      s.update(2)
5
6      // compile error
7      // reference to iso must be unique
8      // let reader: State iso = s
9
10     let w = Worker
11     w.reload(consume s)
12
13     // compile error
14     // can't use a consumed local...
15     s.update(3)
16
```

consume後の
参照は
コンパイルエラー

trn: writeユニーク

- 他アクターからはreadできない
- 自アクターからはreadできる
- 権限を破棄すると、writeがもう存在しなくなることを保証できる。=> (readする参照が別にあっても)他アクターに安全に渡せる

trn: writeユニーク

```
1 actor Main
2   new create(env: Env) =>
3     let s: State trn = State(1)
4     s.update(2)
5
6     let w = Worker
7     w.reload(consume s)
8
9 actor Worker
10  var state: State val
11
12  new create() =>
13    state = State(0)
14
15  be reload(state': State val) =>
16    state = state'
17
18 class State
19  var _i: U32
20
21  new trn create(i': U32) =>
22    _i = i'
23
24  fun ref update(i': U32) =>
25    _i = i'
26
```

trn: writeユニーク

```
1 actor Main
2   new create(env: Env) =>
3     let s: State trn = State(1)
4     s.update(2)
5
6     let w = Worker
7     w.reload(consume s)
8
9 actor Worker
10  var state: State val
11
12  new create() =>
13    state = State(0)
14
15  be reload(state': State val) =>
16    state = state'
17
18 class State
19   var _i: U32
20
21   new trn create(i': U32) =>
22     _i = i'
23
24   fun ref update(i': U32) =>
25     _i = i'
26
```

new isoからnew trn
に書き換えただけなので略

trn: writeユニーク

```
1 actor Main
2   new create(env: Env) =>
3     let s: State trn = State(1)
4     s.update(2)
5
6     let w = Worker
7     w.reload(consume s)
8
```

```
9 actor Worker
10   var state: State val
11
12   new create() =>
13     state = State(0)
14
15   be reload(state': State val) =>
16     state = state'
17
18 class State
19   var _i: U32
20
21   new trn create(i': U32) =>
22     _i = i'
23
24   fun ref update(i': U32) =>
25     _i = i'
26
```

trn: writeユニーク


```
1 actor Main
2   new create(env: Env) =>
3     let s: State trn = State(1)
4     s.update(2)
5
6     let w = Worker
7     w.reload(consume s)
8
```

trnな変数

trn: writeユニーク

```
1 actor Main
2   new create(env: Env) =>
3     let s: State trn = State(1)
4     s.update(2)
5
6   let w = Worker
7   w.reload(consume s)
8
```

参照を放棄
writeできる参照が
なくなる



trn: writeユニーク

```
1 actor Main
2   new create(env: Env) =>
3     let s: State trn = State(1)
4     s.update(2)
5
6   let w = Worker
7   w.reload(consume s)
```

```
9 actor Worker
10   var state: State val
11
12   new create() =>
13     state = State(0)
14
15   be reload(state': State val) =>
16     state = state'
```

```
18 class State
19   var _i: U32
20
21   new trn create(i': U32) =>
22     _i = i'
23
24   fun ref update(i': U32) =>
25     _i = i'
26
```

trn: writeユニーク

```
9  actor Worker
10    var state: State val
11
12    new create() =>
13      state = State(0)
14
15    be reload(state': State val) =>
16      state = state'
```

← val(定数)になる。

box: read参照

- read専用(not ユニーク)
- 自分のアクターでtrnが書き換えるかも
- 他のアクター内にある定数かも
- とりあえず値が読めれば気にしない

box: read 参照

```
1 actor Main
2   new create(env: Env) =>
3     let s: State trn = State(1)
4     s.update(2)
5
6     let reader: State box = s
7     s.update(3)
8
9     reader.get() // 3
10
11    let w = Worker
12    w.reload(consume s)
13
14    reader.get() // 3
15
16 actor Worker
17   var state: State val
18
19   new create() =>
20     state = State(0)
21
22   be reload(state': State val) =>
23     state = state'
24
25 class State
26   var _i: U32
27
28   new trn create(i': U32) =>
29     _i = i'
30
31   fun ref update(i': U32) =>
32     _i = i'
33
```

box: read参照

```
1 actor Main
2   new create(env: Env) =>
3     let s: State trn = State(1)
4     s.update(2)
5
6     let reader: State box = s
7     s.update(3)
8
9     reader.get() // 3
10
11    let w = Worker
12    w.reload(consume s)
13
14    reader.get() // 3
```

```
16 actor Worker
17   var state: State val
18
19   new create() =>
20     state = State(0)
21
22   be reload(state': State val) =>
23     state = state'
24
25 class State
26   var _i: U32
27
28   new trn create(i': U32) =>
29     _i = i'
30
31   fun ref update(i': U32) =>
32     _i = i'
33
```

trnと全く同じなので略

box: read参照

```
1 actor Main
2   new create(env: Env) =>
3     let s: State trn = State(1)
4     s.update(2)
5
6     let reader: State box = s
7     s.update(3)
8
9     reader.get() // 3
10
11    let w = Worker
12    w.reload(consume s)
13
14    reader.get() // 3
15
```

trnを参照して
読み取りしてOK

box: read参照

```
1 actor Main
2   new create(env: Env) =>
3     let s: State trn = State(1)
4     s.update(2)
5
6     let reader: State box = s
7     s.update(3)
8
9     reader.get() // 3
10
11    let w = Worker
12    w.reload(consume s)
13
14    reader.get() // 3
15
```

trn参照の末路を
気にせず読み取ってOK

val, ref, tag (説明略)

- val: immutable
- ref: mutable
- tag: 参照が同じかどうかの比較しかできない。
(read/writeどちらもNG)

型システムがアクターを前提とするメリット

- data-race freeをコンパイル時に保障できるようになる
 - writeするなら他アクターはreadしちゃダメ。自アクターからはread OK
 - writeしないなら他アクターもreadしてOK。もちろん自分もread OK
 - write uniqueなら書き換え件を放棄したら他のアクターに定数として渡してOK
 - read/write uniqueなら権限放棄しつつ他アクターに渡せば、そっちでまたwriteできる
- 別のアクターに渡す・自分のアクター内で処理するなどの情報が型システムで扱える。

型システムがアクターを前提とするメリット

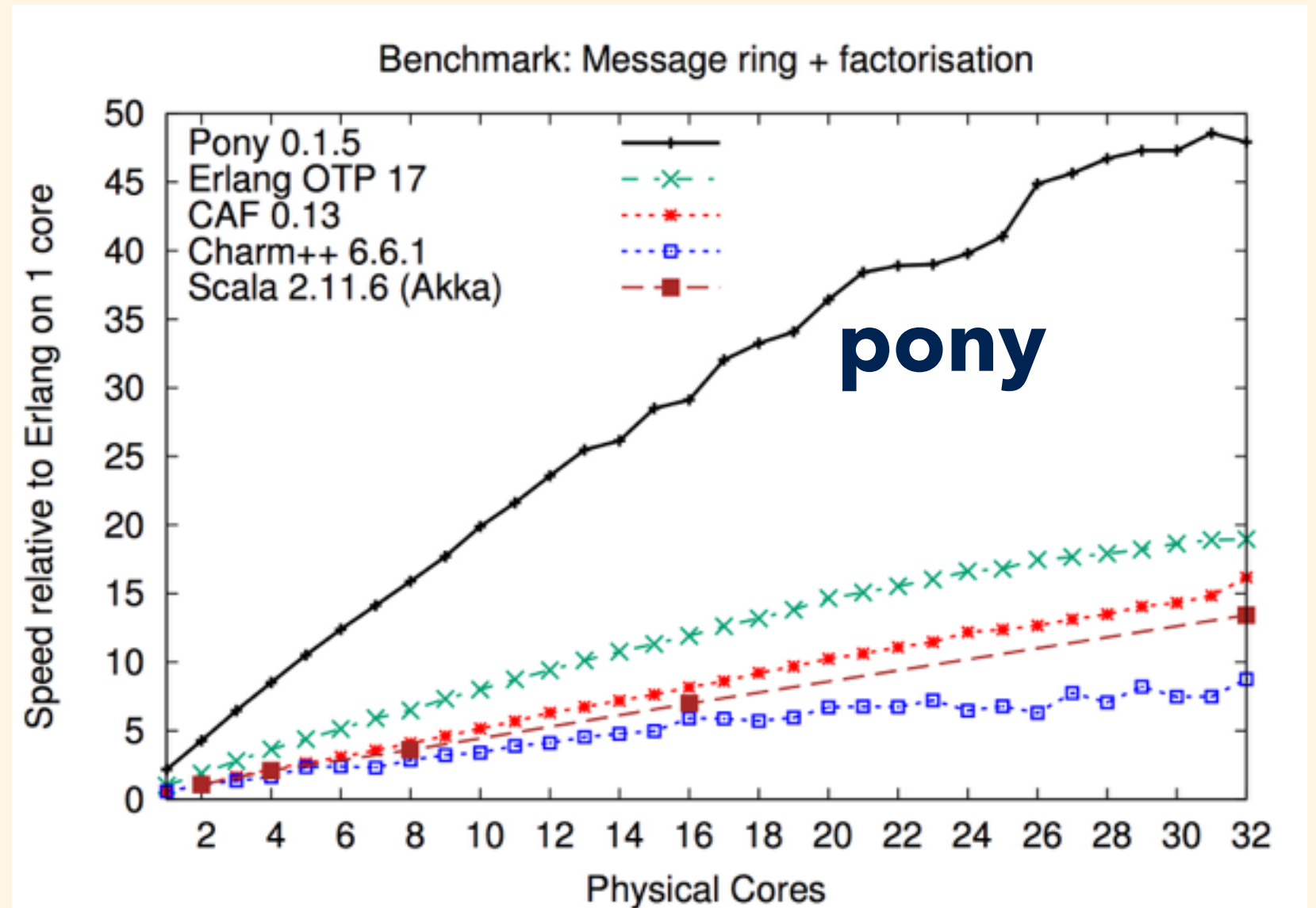
- zero-copyなので大きいデータをメッセージとしてやり取りしてもオーバーヘッドがない
- 標準ライブラリがこの機能をガンガン使っていることによる恩恵も受けられる
- 処理系がデータ競合のケアをしなくて良くなる。GCでの性能改善などに寄与している(らしい) https://www.youtube.com/watch?v=KvLjy8w1G_U

コンパイルが通ったら絶対に
クラッシュさせないという
~~熱い気持ち~~安全な型システム

パフォーマンス (付録)

https://github.com/ponylang/ponylang.github.io/blob/master/benchmarks_all.pdf

- それなりに速そう
- スケジューラーとキューも工夫あり



GCもすごい (付録)

- Pony-ORCA (Ownership and Reference Counting based gc for Actor?)
- stop-the-world lessな並行GC
- one Actor GC/cross Actor GC/Actor GC
- data-race freeを前提に出来る強さ。
- 詳しくは論文(<https://github.com/ponylang/ponylang.github.io/blob/master/papers/OGC.pdf>)

他の言語との比較 (付録)

	<i>Our Work</i>	Gordon	<i>Æminium</i>	DPJ	Kilim	Haller	Scala	Erlang	Rust
Zero-copy	✓	✓	✓	✓	✓	✓	✓		✓
Data-race free	✓	✓	✓	✓	✓ ⁴	✓ ⁵		✓	✓
Statically data-race free	✓	✓	✓	✓	✓	✓			⁶
Non-tree messages	✓	✓				✓	✓	✓	✓
Read unique (<i>iso</i>)	✓	✓	✓		✓	✓			
Write unique (<i>trn</i>)	✓								
Mutability (<i>ref</i>)	✓	✓	✓	✓	✓	✓	✓		✓
Immutability (<i>val</i>)	✓	✓	✓		✓		⁷	✓	✓
Cyclic immutability	✓	✓							
Identity (<i>tag</i>)	✓		⁸						
Destructive read	✓	✓			✓	✓			✓
Recovery	✓	✓							
Using uniques (<i>iso ▷ x</i>)	✓								
Actors	✓				✓	✓	✓	✓	

Table 4. Feature comparison.

⁴ Kilim messages are data-race free but the rest of Java is not.

⁵ The proposed system is data-race free but the rest of Scala is not.

⁶ Rust uses atomic reference counts and reader-writer locks to prevent data races.

⁷ Scala has types that are immutable by design, but cannot annotate references to mutable types as immutable.

⁸ A version of identity, *none*, appears in [27].

Ponyお役立ち資料集 (付録)

- とりあえず公式Tutorial & 発表

- <http://tutorial.ponylang.org/>
- https://www.youtube.com/watch?v=KvLjy8w1G_U

- 論文

- <http://www.doc.ic.ac.uk/teaching/distinguished-projects/2013/s.blessing.pdf>
- <https://github.com/ponylang/ponylang.github.io/tree/master/papers>

まとめ

- shared nothingとshared memoryを統合して高速化を図りつつ安全に並行処理を記述できる
- アクターを前提にした言語&処理系のメリット
 - zero-copy & data-race free & dead-lock free
 - Pony-ORCA GC
- 完全な研究用ではなく実用を目指している。
- 真の狙いは分散システムにある。(distributed ponyに期待)