Ractor
どうなった？

笹田 耕一
STORES, Inc.

Koichi Sasada

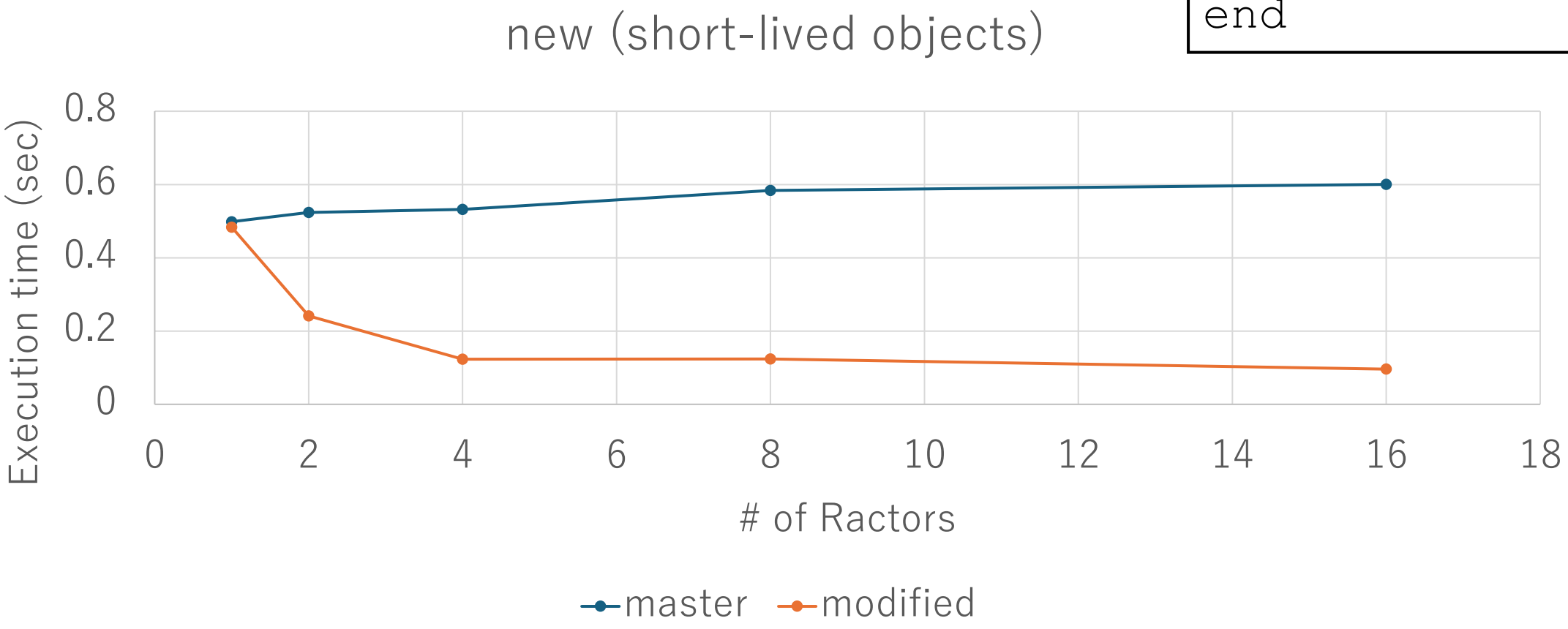
- Ruby interpreter developer employed by **STORES, Inc.** (2023~) with @mametter
  - YARV (Ruby 1.9~)
  - Generational/Incremental GC (Ruby 2.1~)
  - Ractor (Ruby 3.0~)
  - debug.gem (Ruby 3.1~)
  - M:N Thread scheduler (Ruby 3.3~)
  - …
- Ruby Association Director (2012~)
- 今日の受付係＆本屋

1枚で RubyKaigi 2025 の発表のまとめ

- Ractor ごとに GC を独立して実行したい（Ractor Local GC）
  - 今は全部止めて逐次実行
  - それぞれ、Ractor ごと（local）に並列に GC したら速いんじゃない？
    - ほとんどのオブジェクトは、Ractor の中で生まれ、死んでいくので…
    - 自然に並列 GC できて凄い速そう！
    - 子 Ractor はオブジェクトの絶対数も少ないし mark & sweep に有利
  - 共有可能オブジェクトが問題
    - ほかの Ractor が指してる**かも**！（共有してないかも）
    - 間違って回収したら大惨事
  - 解決策：保守的に、共有可能オブジェクトは local GC しない
    - **共有可能オブジェクトの GC は Global GC に任せる**
    - GC サイクル
      - たくさんの local GC
      - 時々 Global GC（全部とめて実行）
    - 細かい話は RubyKaigi 2025 の発表を見てね

Short lived objects
Execution time (sec)

```
def task
  TN.times{
    ''
  }
end
```



Quoted from my talk on RubyKaigi 2025

現状

- Ractor Local GC を入れたい
  - まだちゃんと最新版にキャッチアップできていない
  - （最近 Ractor 周りにいっぱい手が入ってて…）
  - でも 9 月中にはいれたいな、頑張ります

その他の Ractor のはなし

- Ractor::Port
- Ractor.sharable_proc / Ractor.sharable_lambda
- 開発メンバーの話

Ractor::Port

- Ractor 間でメッセージをやりとりする仕組み
- 以前は2通りあった
  - Ractor#send, Ractor.receive（非同期投げっぱなし）
  - Ractor.yield, Ractor#take（ランデブー型）
- Ractor::Port 一つに統一された
  - 非同期投げっぱなし、だけになった（より、Actor モデルっぽい）
  - Ractor.yield, Ractor#take は消されました！（#take はもうすぐ）
- 紹介記事: **`Ractor::Port` — Ractor の API を一新した話**

Ractor::Port によるプログラミング

```
port = Ractor::Port.new # 長いからRactor.portにしたい
Ractor.new port do |port|
  port.send 42 # 他の Ractor は Port に送れる
  port.receive #=> エラー。他の Ractor は受け取れない
end

# 自 Ractor だけ受け取れる
p port.receive #=> 42
```

Ractor::Port によるプログラミング default_port

```
p Ractor.current.default_port
#=> #<Ractor::Port to:#1 id:0> それぞれ1個もつ

r = Ractor.new do
  param = Ractor.receive
  # Ractor.current.default_port.receive と同じ
  Ractor.main << (task(param); 42)
end
Ractor.receive #=> 42

# Ractor に対して send, receive ができるのは変わらず
```

Rcator::Port によるプログラミング
Thread と組み合わせ可能に！

```
port1 = Ractor::Port.new
port2 = Ractor::Port.new
Ractor.new(port1){|port1| port1 << 1}
Ractor.new(port2){|port1| port1 << 2}
Thread.new{loop{p port1: port1.receive}}
Thread.new{loop{p port2: port2.receive}}
sleep
```

新しい仲間
Ractor#join, Ractor#value

- 終了を待つために導入
  - Ractor#take がなくなったため
  - #join は終了を待つ
  - #value は終了を待ってブロックの結果を返す
  - 覚え方：Thread#join, Thread#value と同じ
  - 中味は Port を作って実行

ちょっとした仕事のオフロードによさそう

```
Ractor.new{ heavy_task() }.value #=> 42
```

新しい仲間
Ractor#monitor, unmonitor

- Ractor の挙動を監視するために入った仕組み
  - Ractor#join, #value を作るため導入
  - イベントがあると、（ほかの Ractor が）登録した Port にメッセージ
  - 発展的機能なので、基本的に気にしないでいいと思う
  - 今は終了イベントしか飛ばさないけど、プロファイリング用途に何か飛ばすといいかな？

```
port = Ractor::Port.new
r = Ractor.new{sleep 1; 42}
r.monitor(port) # 何個でも登録可能
p port.receive #=> :exited
p r.value #=> 42
```

Ractor::Port によるプログラミング
select による複数ポート待ち受け

```
evenp = Ractor::Port.new
oddp = Ractor::port.new
Ractor.new(evenp, oddp){
|evenp, oddp|
  100.times{
    if it.even?
      evenp << it
    else
      oddp << it
    end
  }}
```

```
while true
  port, val =
   Ractor.select(evenp, oddp)
  case port
  when evenp
    p even: val
  when oddp
    p odd: val
  end
end
```

Ractor::Port によるプログラミング
select による終了の待ち受け

```
r1 = Ractor.new{…}
r2 = Ractor.new{…}
r, val = Ractor.select(r1, r2)
#=> r1 か r2 の、先に終了したほうが返る
# (1) r1, r2 の死活監視に使える
# (2) 仕事をさせておいて、速く終わったほうを処理
#       例えば、http request とかね！
```

Ractor::Port 雑感

- なんでやったか
  - いろんな選択肢から、Port を採用（5年かかった）
  - とにかく Ractor.select の yield 待ちが書けなかった
    - 最後まで安定しなかった
    - Ractor::Port は、だいぶ安定しています　コードがすごく単純になった
  - Thread 対応が難しかった
- 著名人からの反応
  - Matz: 「yield と take は変だと思ってたんだよね」
  - m_seki: 「こういうのが要ると思ってたんだよね」
- Go の context みたいなことがどれくらい要るんだろう…？

Ractor.shareable_proc, shareable_lambda

- Ractor に Proc を渡したい！
  - タスクを Proc で書いて woker Ractor に渡すとか
- Ractor 間で共有可能な Proc を作るのはめんどかった
  - self が sharable でないといけないので…
  - `shpr = nil.instance_exec{ Proc.new { … } }`
- Ractor.shareable_proc(self: nil) が加わった（加わりそう）
  - `Shpr = Ractor.shareable_proc{ … }` で良い。わかりやすい！
  - 生成する self を指定可能（デフォルトは nil）
  - `Ractor.make_shareable(proc_obj)` は obsolete だと思う
- まだ入っていないけど、必ず入れたい
  - 外側のローカル変数の扱いで揉めてる

新機能と非互換のまとめ

- 入った（入るだろう）
  - Ractor::Port
  - Ractor.default_port
  - Ractor#join, #value, #monitor, #unmonitor
  - Ractor.sharaeble_proc, sharable_lambda
- 変わった
  - Ractor.select（Port 対応）
- 無くなった（無くなるだろう）
  - Ractor.yield, Ractor#take
  - Ractor.make_sharable(proc_obj) による sharable への変更

開発メンバーの話

- Shopify の方々がすごい力を入れてくれてます
  - 試してくれたり、たくさんパッチ書いてくれたり
  - ありがとうございます
- 毎月~~突き上げを食らう~~ミーティングしてます

まとめ

- Ractor local GC
  - やってます
  - Ruby 3.5/4.0 には入れたい
- 新機能
  - Ractor::Port
  - Ractor.sharable_proc / Ractor.sharable_lambda
- 開発メンバーの話

そろそろ、experimental を外したいなあ