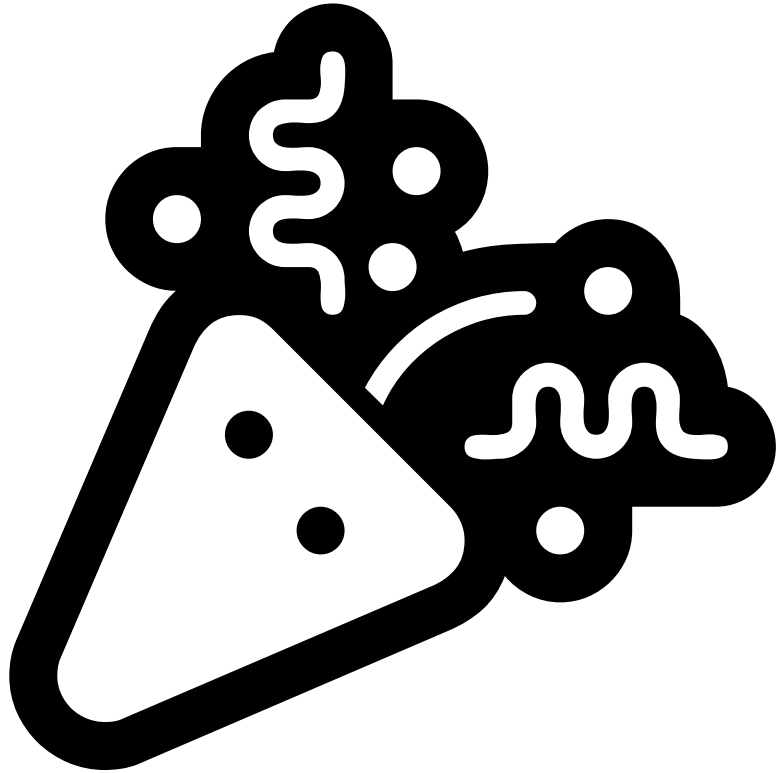
# Ruby 3.0 の Ractor について

笹田耕一
ko1@cookpad.com
クックパッド株式会社

# Ruby 3.0.0 released!



2020/12/25

https://www.ruby-lang.org/ja/news/2020/12/25/ruby-3-0-0-released/
より引用

Ruby 3 では以下の目標を達成しました。

- パフォーマンスの改善
  - MJIT
- 並行処理
  - Ractor
  - Fiber Scheduler
- 静的解析
  - RBS
  - TypeProf


https://gihyo.jp/magazine/wdpress/archive/2021/vol121

特集1

# 詳解 Ruby 3

さらに速く!
さらに書きやすく!

JITコンパイラ、並列プログラミング、静的型解析



← 今日の話

第3章

# Ractor

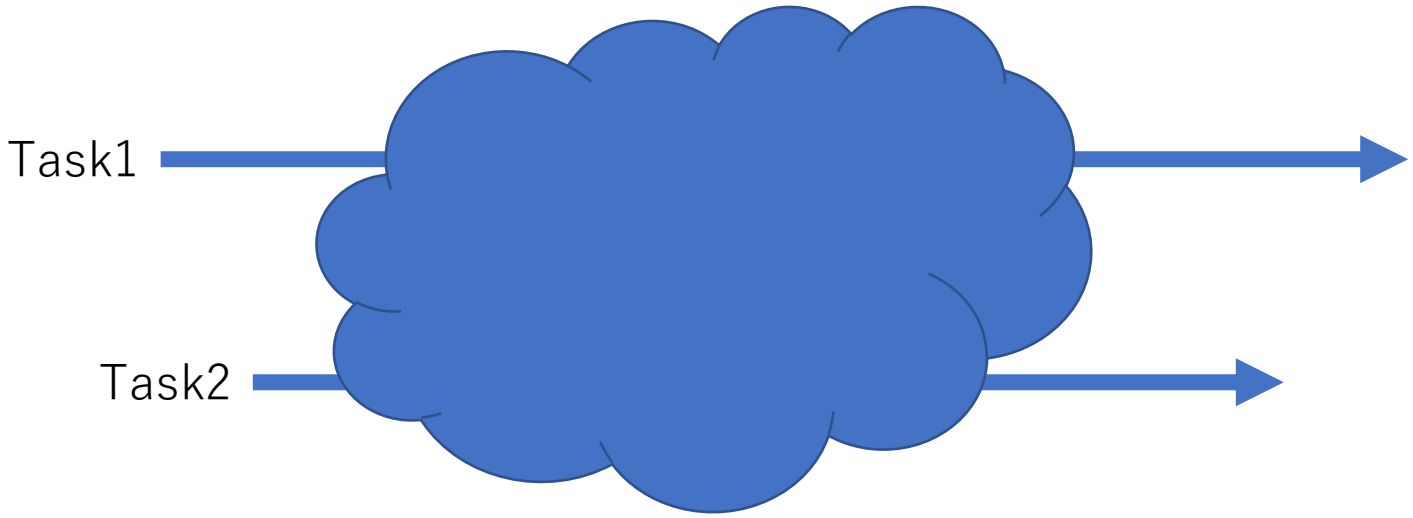
↓???

## スレッド安全を気にせず並行／並列プログラミングが行えるしくみ

笹田 耕一 SASADA Koichi　クックパッド㈱
URL https://www.atdot.net/~ko1/
mail ko1@atdot.net　GitHub ko1　Twitter @koichisasada

# 並行？　並列？

- 何か独立したタスクがあるとき、どっちもいい感じに走らせたい
  - どちらも走らせておくと、いい感じにおわる
  - タスクは、それぞれ逐次的に書くことが多い
    vs. イベントドリブンプログラミング



# 並行？　並列？

- 並行
  - 独立した複数のタスクを同時に実行しているように見せる
  - 論理的な同時
  - 主に、I/O を並行に処理
    - Ruby のスレッドは「並行」をサポート
    - ただし、生成コスト・メモリ消費が重いので、あまり使われない
      （ので、Ruby 3.0 では Fiber scheduler というものが入りました）
- 並列 → 並行処理の実行方法の一つ（vs. 時分割）
  - 独立した複数のタスクを実際に同時に実行する
  - 物理的な同時
  - 主に性能向上が目的
    - Rubyでは、プロセスを複数作って並列をサポート
    - でも、プロセス管理はしんどい・書きづらい（dRuby等でサポート）
  - 並列処理するためには並行処理として記述

# 並行（並列）プログラミングのつらさ

- Thread プログラミングはつらい
  - データを全スレッドが共有するので、同期（ロックなど）を**適切に**しないといけないのでつらい（**人間は間違える生き物なので間違える→ つらい**）
  - CPU 複数あっても（MRIでは）並列処理してくれない
- Process プログラミングはつらい
  - プロセスを管理するのはつらい
  - プロセス間通信をきちんとするのはつらい
  - 生成するのが重い
- Fiber プログラミングはつらい→ Ruby 3.0 から Fiber schedulerは？
  - 同期は結局必要（I/O で切り替わるが、どこでI/Oが起こるかわからない）
  - 生成は軽いので、たくさん作ってもそんなにつらくない（C10K）

# Thread つらみ

Thread クイズ

https://gist.github.com/ko1/c0e786825c3b87b2c21a999b68f21c9e

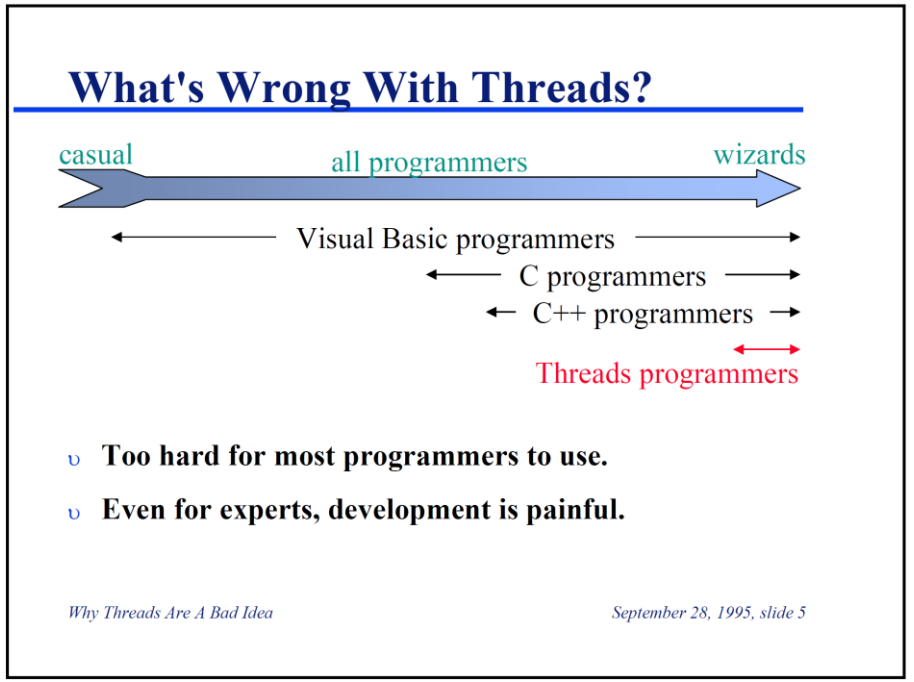
（社内向けにちょっと書いたコードを少しだけ）

cookpad


https://ideas.cookpad.jp/

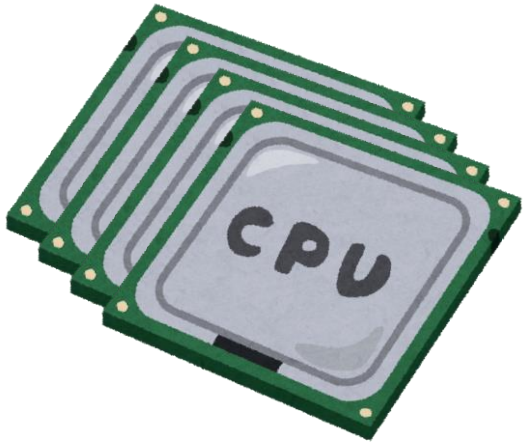
# Rubyでの並列・並行 何が問題か？

- 並行プログラミングは難しい
  - 同期をきちんと書くとかマジ無理（スレッド安全の問題）




https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf

- 気楽に並列処理できない
  - 複数CPUコアを使いきれない
  - プロセスをきちんと使うとかムズイ



第3章

# Ractor

↓これが目標！

**スレッド安全を気にせず並行／並列プログラミングが行えるしくみ**

**笹田 耕一** SASADA Koichi　クックパッド㈱
URL https://www.atdot.net/~ko1/
mail ko1@atdot.net　GitHub ko1　Twitter @koichisasada

# Ractor について（文献）

- https://github.com/ruby/ruby/blob/master/doc/ractor.md
  - 公式ドキュメント　多分、全仕様かいています
  - 英語非ネイティブな私が書いているので、平易な英語です
- **Ruby 3.0 の Ractor を自慢したい (techlife blog)**
  - 日本語で、主に動機について書いています
- WEB+DB Press vol.121 特集1 第3章 Ractor
  - 6ページに頑張って詰め込んでいます
- 発表資料
  - RubyKaigi http://www.atdot.net/~ko1/activities/2020_rubykaigi.pdf
  - Rubyconf http://www.atdot.net/~ko1/activities/2020_rubyconf.pdf

# Ractor について

❶複数のRactorは並行／並列に実行される

❷スレッド安全の問題を生じさせないようにRactor間を隔離し、基本的にオブジェクト（データ）を共有しない

- ❷-1：例外的に、いくらか共有可能オブジェクトがある
- ❷-2：共有不可オブジェクトを隔離するため、Rubyの機能に制限がある

❸Ractor間のコミュニケーションは、メッセージの送受信を用いて行う。その際、オブジェクトはコピーか移動を行う

# 複数の Ractor は並列に実行

- Ractor.new{ expr } とすると、expr が**並列**に実行される

```
def tarai(x, y, z) =
  x <= y ? y : tarai(tarai(x-1, y, z),
                     tarai(y-1, z, x),
                     tarai(z-1, x, y))
require 'benchmark'
Benchmark.bm do |x|
  # sequential version
  x.report('seq'){ 4.times{ tarai(14, 7, 0) } }

  # parallel version
  x.report('par'){
    4.times.map do
      Ractor.new { tarai(14, 7, 0) }
    end.each(&:take)
  }
end
```

```
Benchmark result:
          user     system      total        real
seq  64.560736   0.001101  64.561837 ( 64.562194)
par  66.422010   0.015999  66.438009 ( 16.685797)
```

Ubuntu 20.04, Intel(R) Core(TM) i7-6700 (4 cores, 8 hardware threads)

# Ractor 間は「基本的には」隔離される

- ふつうのオブジェクトは、Ractorをまたがない/またげない
  → 同期が不要
  　たのしい並行並列プログラミング
- 共有される特別なオブジェクト
  - 不変オブジェクト
  - **クラス・モジュール**オブジェクト
  - その他特殊なオブジェクト（Ractor オブジェクト自体）

# 言語機能の制限

- グローバル変数、クラス変数の設定／参照
  - まぁ、今どき…
- 定数に**共有不可オブジェクト**の設定／参照
  - 定数は、不変オブジェクトだけにしようという practice
- 共有可能オブジェクトのインスタンス変数の設定／参照

```
class C
  @iv = []
  def self.push(obj) = @iv.push(obj)
end
# これ使ってる人多そう。
# 今これを限定的に許すかどうか議論中です。
```

# 定数サポート

- 共有可能オブジェクトへ変換（基本的に deep-freeze）

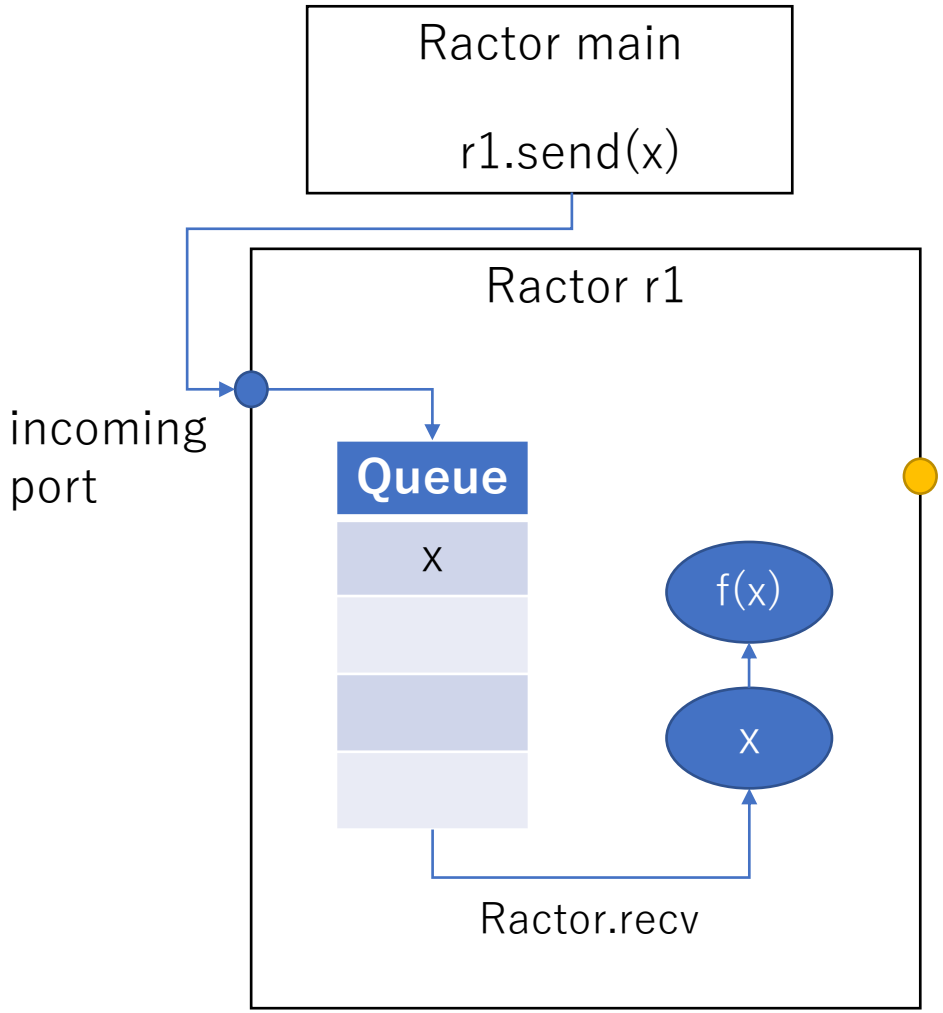
```
Ractor.make_sharable([“a”, “b”, “c”])
#=> [“a”.freeze, “b”.freeze, “c”.freeze].freeze
```

- 定数へ代入するリテラルを共有可能オブジェクトに勝手に

```
# sharable_constant_value: literal
C = [“a”, “b”, “c”]
#=> コンパイル時にコード変換
# C = Ractor.make_sharable([…])
```

# Ractor 間のコミュニケーション

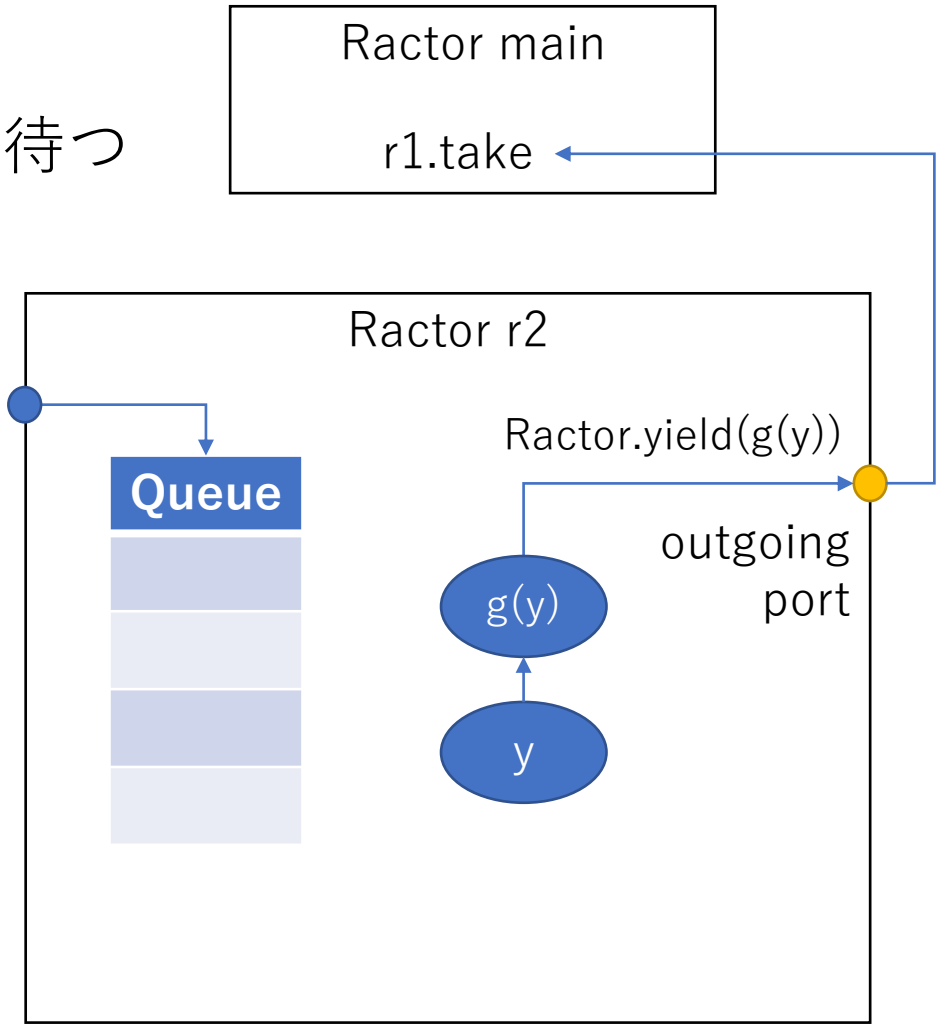
- push型
  - Ractor#send(obj) で対象 Ractor へ送る
    - 対象Ractorのキューへ突っ込む
  - Ractor.receive で受け取る
    - 自キューから取り出す
    - なければブロックして待つ
- よく Actor 型と言われるやつ



# Ractor 間のコミュニケーション

- pull型
  - Ractor.yield (obj) で誰かがもっていくのを待つ
    - 持っていくまでブロック
  - Ractor#take で受け取る
    - 対象 Ractor が yield してなければブロック
- ランデブー型同期を実現
- Ractor.new{expr}の返値は yield

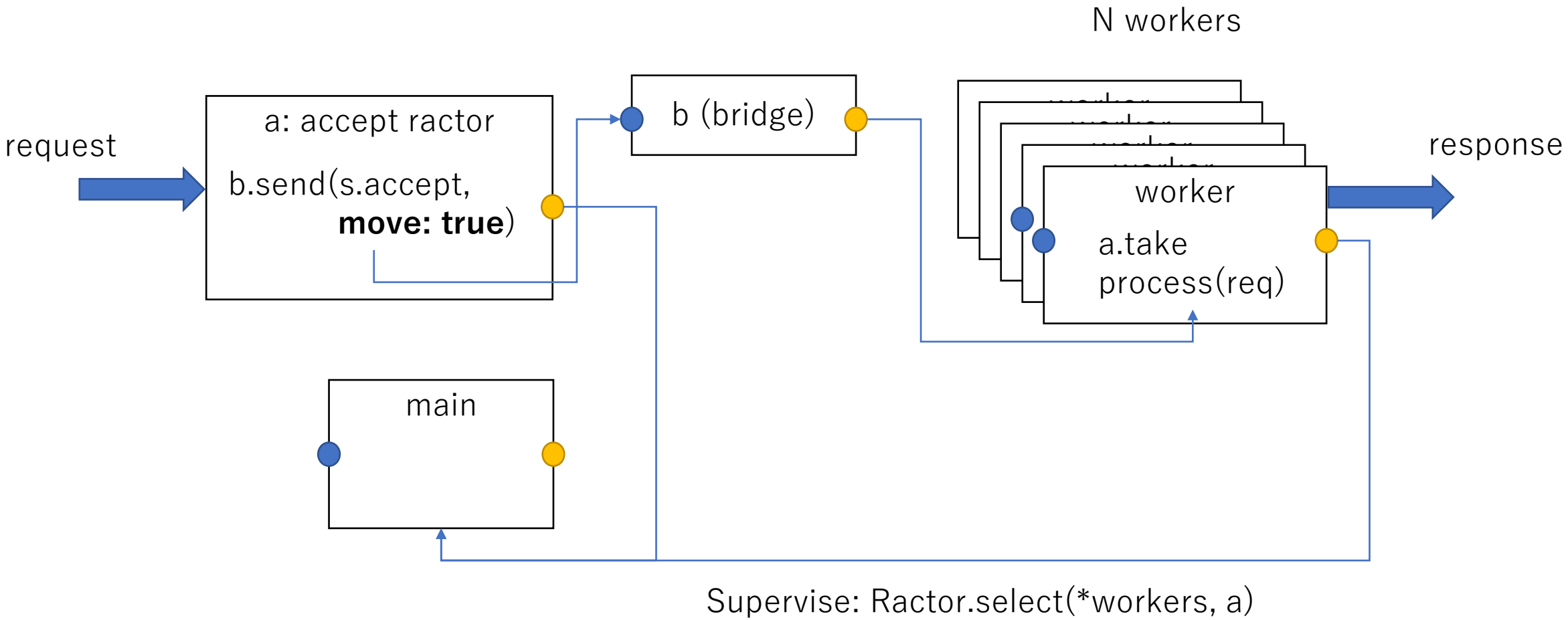
```
# Promise 的使い方
r = Ractor.new{ expr }
… # do something
r.take #=> expr の返値
```



# Ractor間のコミュニケーション
# Ractor.select

- 複数の Ractor を同時に take
  - `Ractor.select(r1, r2, …)`

# 組み合わせるといろんなことができそう
# Example: Web application server

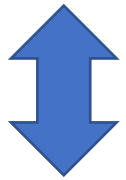


# Ractor の思想

- スレッド：全部共有
  - Good: 簡単に共有
    - 共有しているところを注意して同期する
  - Bad: 簡単に死ぬ
    - 注意し忘れると死ぬ（変な値になったりよくわからないところで死ぬ）
    - どこを共有しているかは、だいぶ集中しないとわからない（ので見逃す）
- Ractor：共有を制限
  - Bad: 共有が手間
    - 共有するところを特別に手間をかけて書く
  - Good: スレッド安全
    - 手間をかけないと単純にエラー
    - エラー箇所を見れば、共有したいことがわかる

多分、共有するところってプログラム全体で見ると、
そんなにないと思うんですよね…

共有のコスト



注意するコスト

# 簡単な評価（2020/9月くらい）
# Create/Invoke/wait time comparison for 10k

| | WSL2 (Ubuntu 20.04) | Ubuntu 18.04 |
|---|---|---|
| process | 9.608186 | 36.939180 |
| ractor | **0.526030** | **0.259494** |
| thread | **0.451909** | **0.137313** |
| fiber | 0.022461 | 0.020944 |
| proc | 0.005264 | 0.003301 |

(sec)

TODO: Make Ractors/threads creation faster as fibers (Ruby 3.1~)

https://gist.github.com/ko1/6257532de84cdb4212581c66415155ed

# で、使えるの？

- 簡単な重いタスクを並列化するのには使えそうです
  - gzip 圧縮したい、とか
  - 簡単なところから試してみてください
- Rails のリクエストを Ractor で並列に～　は**当分無理**
  - ライブラリを対応させなければならない
    - 対応させられるかすらわからない
    - 対応させるためのイディオムが今後育っていくといいな
  - 性能が低い
    - 並列化したほうが遅い、といった問題がまだ残っている
    - バグもまだ残っている

# まとめ

- Ruby 3.0.0 で導入された Ractor についてご紹介
  - モチベーション
  - 簡単な概要
  - 簡単な評価
- Ractor は結構 Ruby のプログラミングスタイルを変更します
  - 私個人としては、今後のために、必要なスタイルの変更だと思っている

  → 新機能の追加ではなく、新たなパラダイムの追加

  例：GCの導入（自分で free できないけど安心を手に入れた！）

よかったら遊んでみてください

**何かあれば ruby-jp slack #concurrency でお気軽に**