

Ractor の機能強化、 2024 年

笹田 耕一

株式会社ストアーズ



今日のトピック

- Ractors の重要な機能をサポート • “require” • “timeout”
- Ractor のメモリ管理の問題 • 将来の機能強化計画 • GC 戦略 • 提案された API



笹田 耕一

- 株式会社ストアーズに在籍（2023年～）@mametterと共にRubyインタプリタ開発者• YARV（Ruby 1.9～）
 - 世代別/増分GC（Ruby 2.1～）
 - Ractor（Ruby 3.0～）
 - debug.gem（Ruby 3.1～）
 - M:N スレッドスケジューラ（Ruby 3.3～）
 - ...
- Ruby協会理事（2012年～）



STORES

今年も来れて嬉しいです！

「ラクター」は

- Ruby 3.0 から導入されました。 •


マルチコアでのパフォーマンス

😊😊向上のために Ruby 上で並列コンピューティングを可能にするように設計されています。 • より高速なアプリケーションを作成できます。 • 堅牢な並行プログラミング
😊😊オブジェクト共有によるバグはありません。

より安全な並行プログラミングを実現する 厳格な Ractor ルール

- 🙄 Ractor間のオブジェクト共有機能を制限する
 - 共有不可能なオブジェクトと共有可能なオブジェクト
 - 共有不可能なオブジェクト – ほとんどのオブジェクト
 - 共有可能なオブジェクト – 一部の特別なオブジェクト
 - 不変オブジェクト
 - いくつかの特殊オブジェクト
 - クラス/モジュール
 - Ractor オブジェクト
 - ...
- 定数 (など) は、子Ractor (メイン以外の Ractor) によって共有不可能なオブジェクトを取得/設定することはできません。
- グローバル変数は子 Ractor からはアクセスできません。
- ...

ラクターズに関する問題

-  重要な機能の欠如 • 子 Ractor の「require」 • 子 Ractor の「timeout」

- ...

- ...

-  パフォーマンスの低下 • メモリ管理について

- ...

- ...

「要求」の問題

子Ractorは「require」を呼び出すことができない

- 一部のコードは子 Ractor のメソッドでライブラリを必要とします• “autoload” の場合• `def foo = (require “foo”; Foo.foo)`

- “pp” の場合 – 最初の“pp”が呼び出されたときに“pp”を要求します。• そのため、子Ractorで“require”を許可する必要があります。•

“require”は子Ractorでは禁止されています。

- `$LOAD_PATH`、`$LOADED_FEATURES`などにアクセスします。• ロードされたコードは共有できないオブジェクトを持つ定数を定義できます。

`STR = “str” # 共有できないオブジェクトを設定する`

- RubyGemsの複雑なロジック• “require”はメイ

ンRactorで実行する必要がある

"必要とする"

解決策: メインRactorの「require」

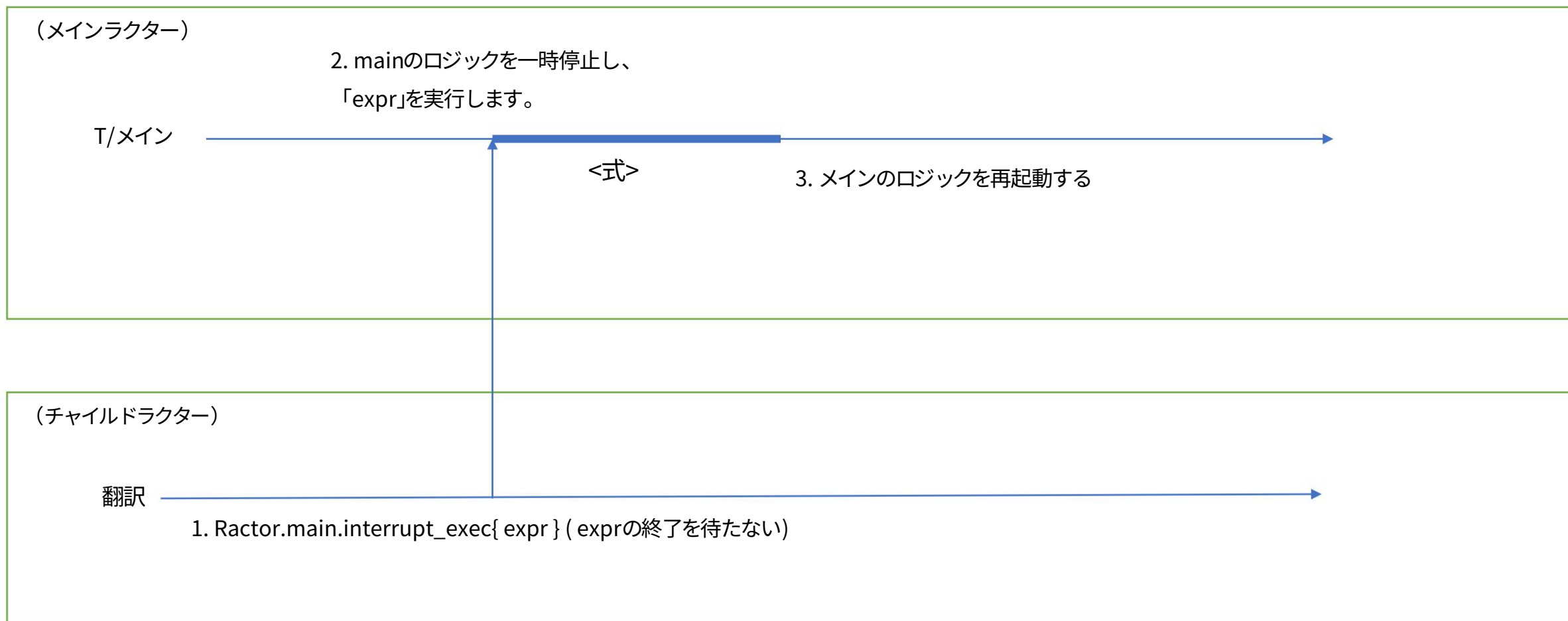
- 新しいAPI 「Ractor#interrupt_exec{ expr }」を導入
 - レシーバーRactorのメインスレッドでexprを非同期的に実行する
 - ブロックは共有可能なProcに変換されます（外部スコープにアクセスできません）
 - exprの戻り値は無視されるので明示的に送信する必要があります
 - ハンドラーを捕まえたり、信号を送るのが好きです（そのため危険でもあります）
 - メインスレッドはIOブロッキングなどのメソッド（シグナル処理など）によって中断されます。 • Ractor.main.interrupt_exec{ \$g=1 }

は“\$g=1”を実行します。

メインRactorのメインスレッド

- メインRactorに限定されたリソースにアクセスするのに便利です
- メインスレッドを中断するにはオーバーヘッドが必要

「require」を実装するには ラクター#interrupt_exec



Ractor.require(feature) で 「require」を実装する

クラス Ractor

```
def self.require(機能)
```

```
  c = Ractor::Channel.new
```

```
  Ractor.main.interrupt_execは  
  スレッド.newdo
```

```
    c << 必要(機能)
```

```
    救助例外 => e
```

```
    c << e
```

```
  終わり
```

```
  終わり
```

```
  c.テイク
```

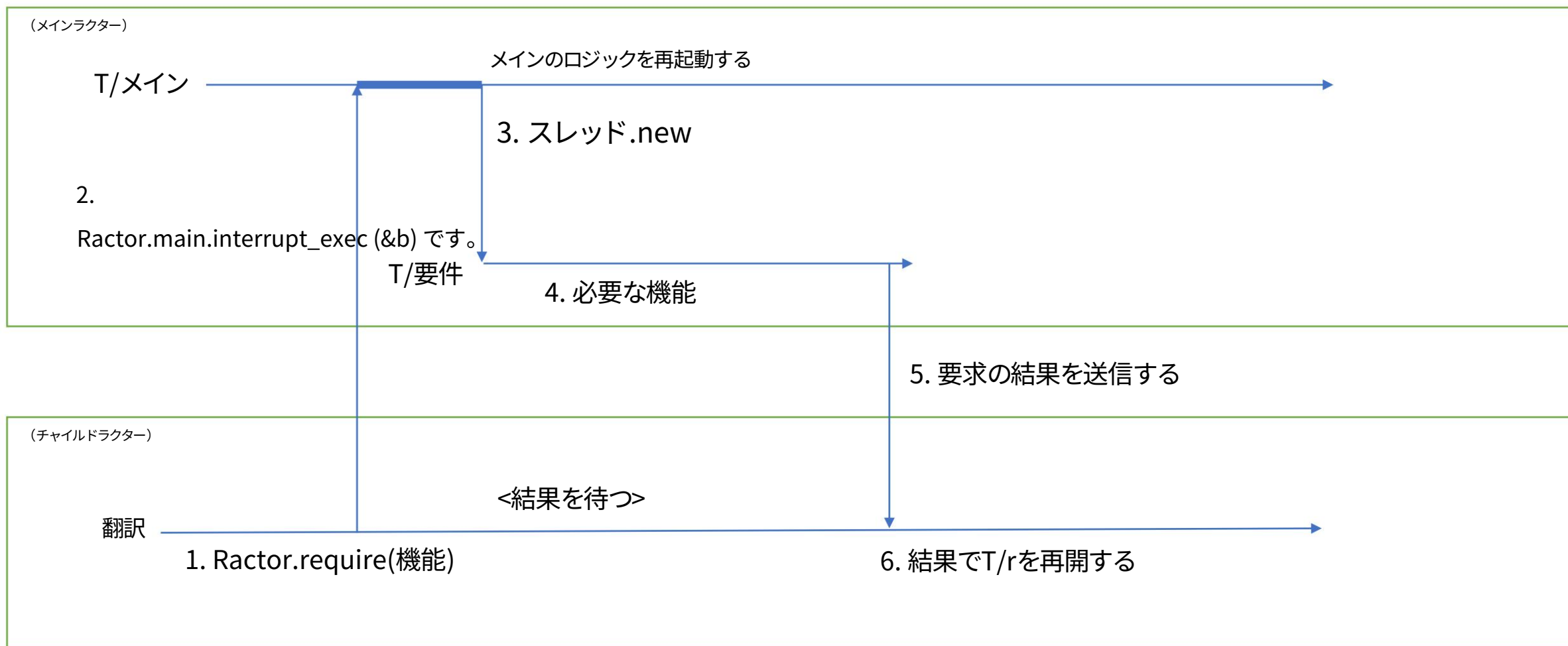
```
  終わり
```

```
def Ractor::Channel.new =
```

```
  Ractor.new{loop{ Ractor.yield Ractor.receive }}
```

- メインRactorに要求する
- 別の「require」を呼び出す
再帰ロック（デッドロック）によるスレッド
- 呼び出し側のRactorは「require」の結果を待つ必要があります

Ractor.require(feature) で “require” を実装する



Ractor.requireで「Kernel#require」を実装する

モジュールカーネル

```
def require(機能)
```

Ractor.main でない限り、 Ractor.require(feature) を返します。

```
# メイン Ractor の元の require
```

終わり

終わり

```
def Ractor.main? = Ractor.current == Ractor.main
```

Ractor がサポートする「require」の問題

- いくつかのライブラリは「Kernel#require()」をオーバーライドします。 •

Rubygems •

Bundler

• ...

- 全員、メイン以外のRactorガードを挿入する必要があります

def require(feature) は Ractor.main でない限

り Ractor.require(feature) を返します。

...

- すべてのオーバーライドの先頭にこの行を追加するように依頼できますか？
定義は？

Ractor がサポートする「require」の問題

先頭に付加するモジュールを提供しますか？

モジュールをカーネルの先頭に追加すれば解決できる

モジュールRactorAwareRequire

```
def require(機能) =
```

```
  Ractor.main? ? Ractor.require(feature) : super
```

終わり

モジュールカーネル

RactorAwareRequire を先頭に追加

終わり

ただし、クラスのすべての祖先にはそれが含まれています

p ".クラス.祖先

=> [文字列、比較可能、オブジェクト、 RactorAwareRequire、カーネル、BasicObject]

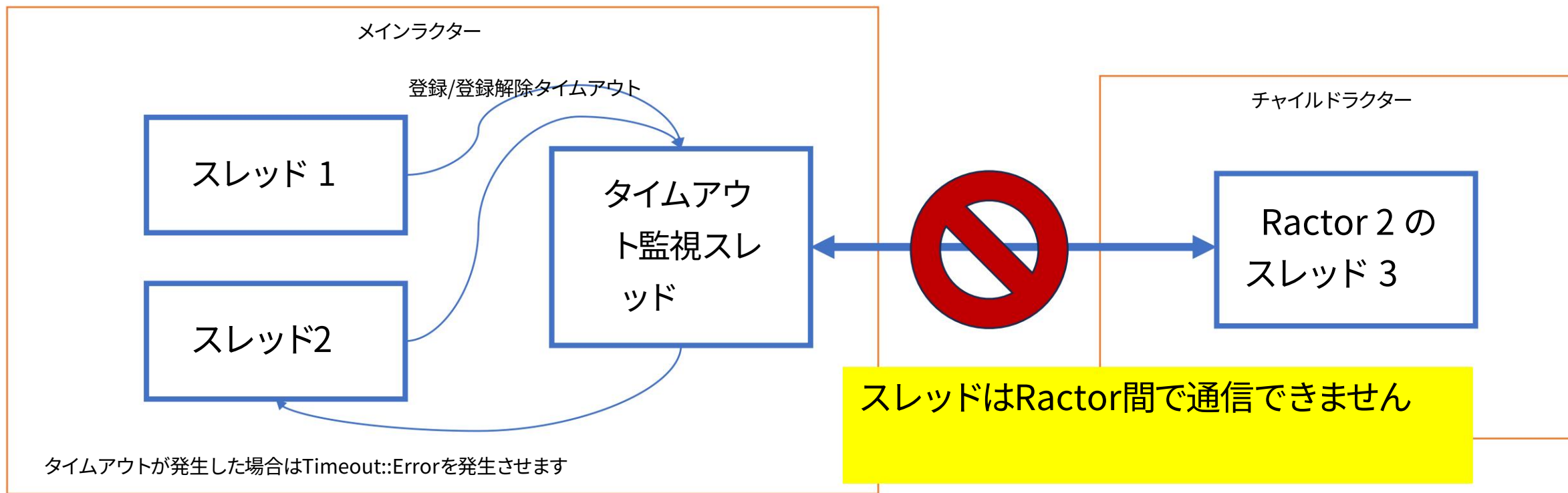
オフトピック

Ractor/スレッド#interrupt_exec

- この機能はデバッガーがすべてのスレッドを停止するのに便利です
 - 現在のデバッガーはRactorをサポートしていません → その主な機能
 - 現在のデバッガー実装では、すべてのスレッドを停止するために「line」トレースポイントを使用していますが、「ブロッキング操作」(I/O 待機など)を実行しているスレッドを停止することはできず、スレッド情報にアクセスすることもできません。
- この機能は、トラップハンドラのように、クリーンアップコードなどのコードを中断する可能性があるため危険です。
 - 大いなる力には大いなる責任が伴う
 - Rubyの機能を導入するのは難しいですか?(C-API?)

「タイムアウト」の問題

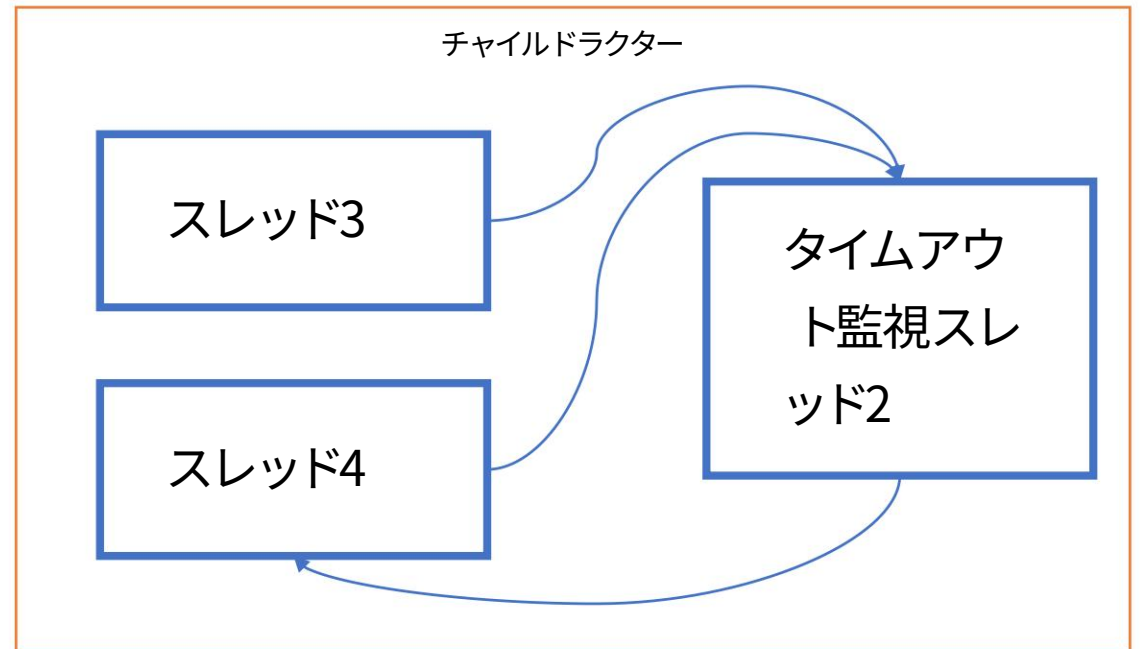
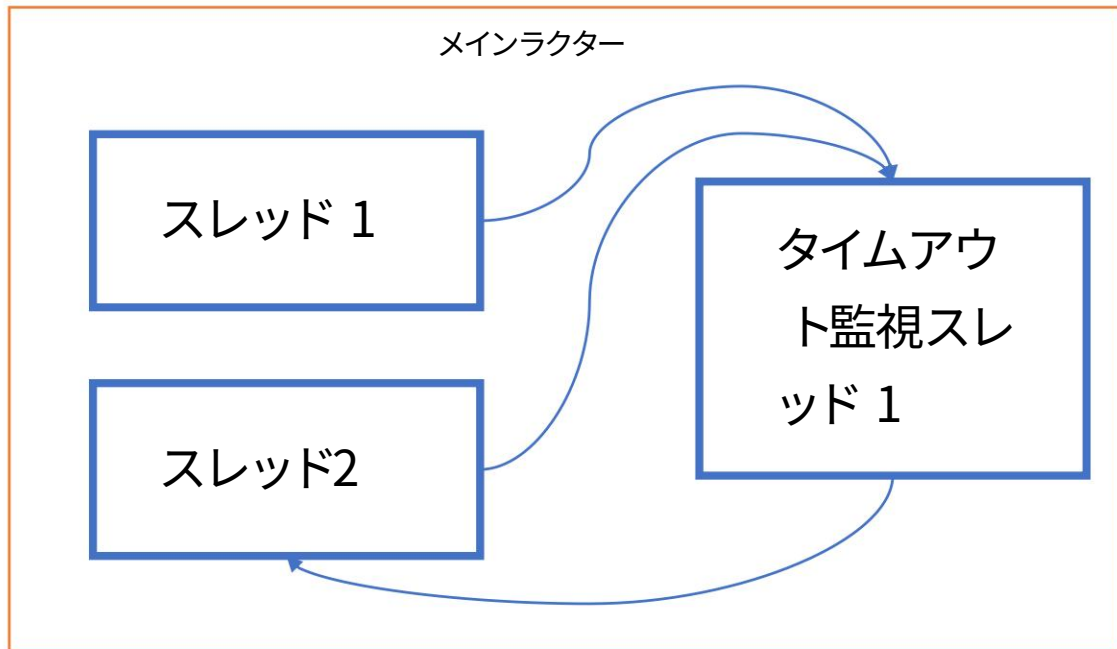
- “timeout” ライブラリはThreadを使用して非同期例外を送信します
タイムアウトスレッドへ
- Ractor間で通信できない



"タイムアウト"

解決策1: Ractorごとにモニターを用意する

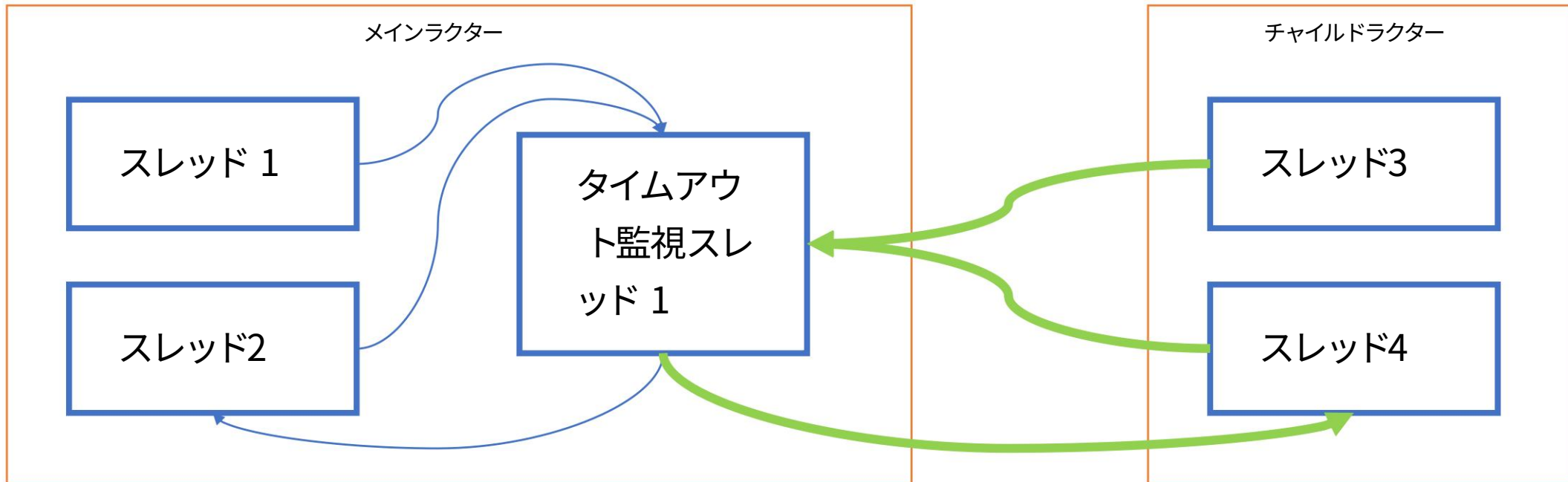
- Ractorごとにモニタースレッドを提供する
- 🤖 Ractor ローカル変数を使用した簡単な実装 (30 分)
- 🤖 モニタースレッドが必要



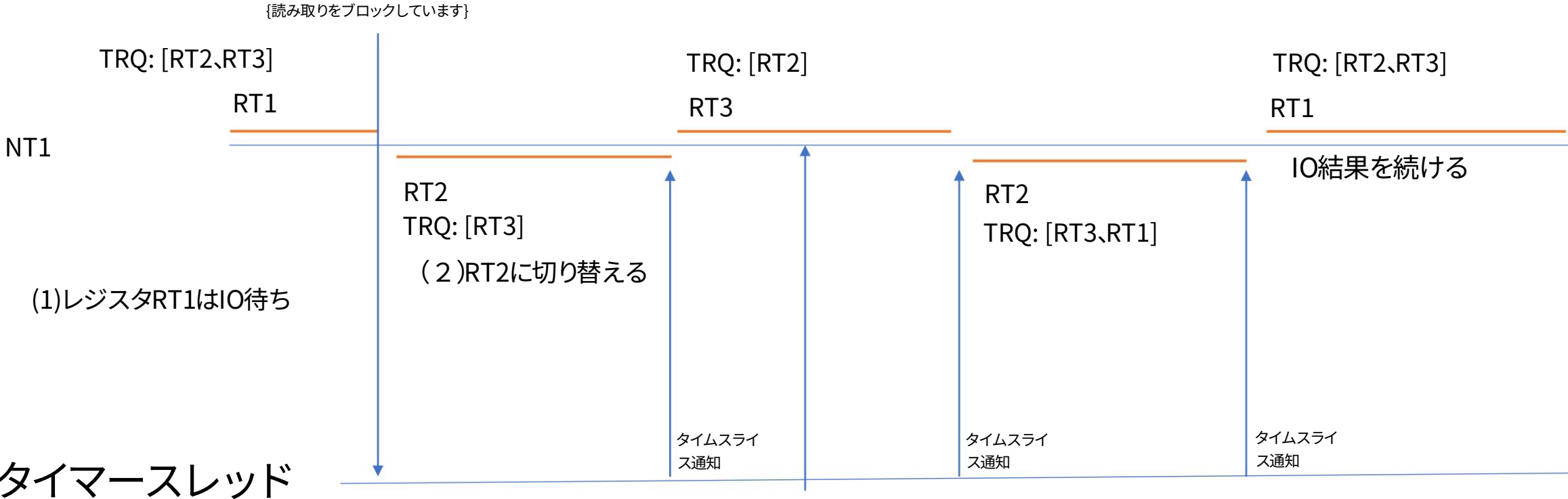
"タイムアウト"

解決策2: 新しい通信パス

- Ractor間の新しい通信パスを使用する
- 🤪 Ruby で 1 つのプロセスを監視 (大規模な Ractor では困難)
- 🤔 難しいAPI設計



管理されたブロッキング操作を処理する



開始ステータス:
RubyスレッドRT1、RT2、RT3がある
TRQ (スレッドレディキュー)は[RT2、RT2]です

(3) RT1をレディキューに追加 →
TRQ: [RT2, RT1]

? RT1は早めにスケジュールできる

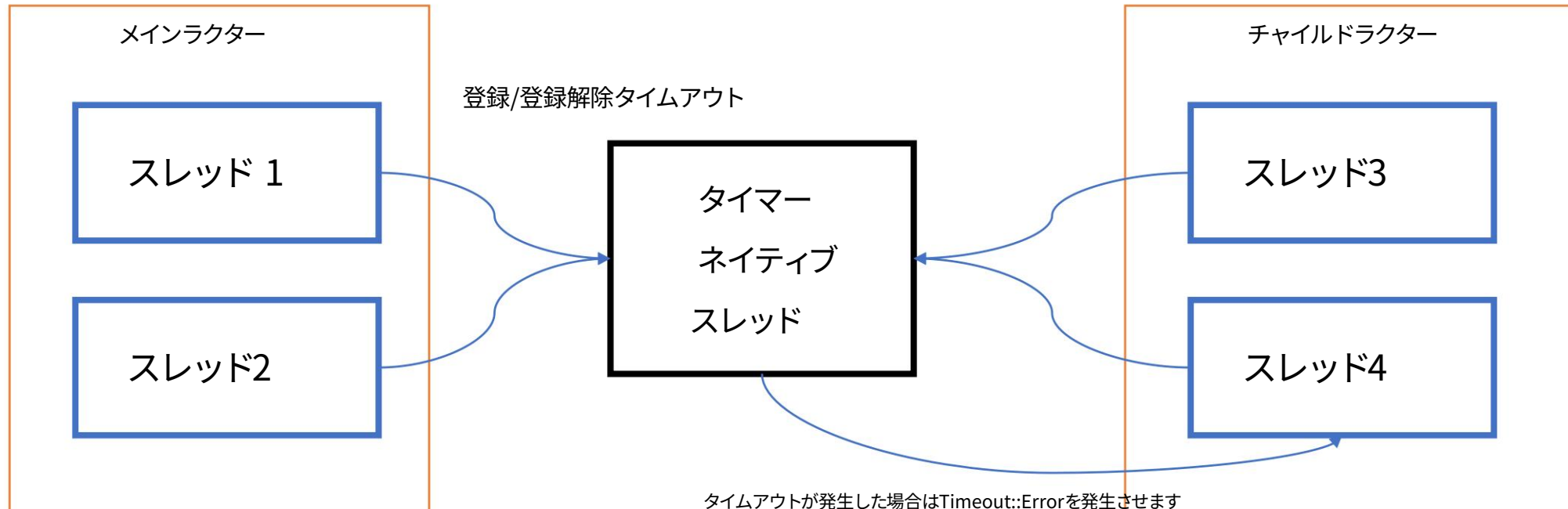
"タイムアウト"

解決策3: ネイティブタイマースレッドを使用する

- M:Nスレッドスケジューラにタイマースレッドを使用する

- タイマースレッドは既にスリープなどのタイムアウトを管理しています。

す。😄 Rubyのタイマースレッドは不要です。C実装のためパフォーマンスが向上します。 • M:Nをサポートしていないプラットフォームをサポートする必要がある



"タイムアウト"

解決策3: ネイティブタイマースレッドを使用する

モジュールタイムアウト

簡易版

```
def timeout(sec, exc = Timeout::Error, msg = "...")
```

```
  RubyVM.timeout_exec(
```

```
    sec, proc{Thread.current.raise exc, msg})を実行する
```

収率

終わり

終わり

- RubyVM.timeout_exec は、Ractor#interrupt_execと同じメカニズムでタイムアウトすると、指定された Proc を呼び出します。
- Thread.timeout_execのような汎用 API を導入できますか？

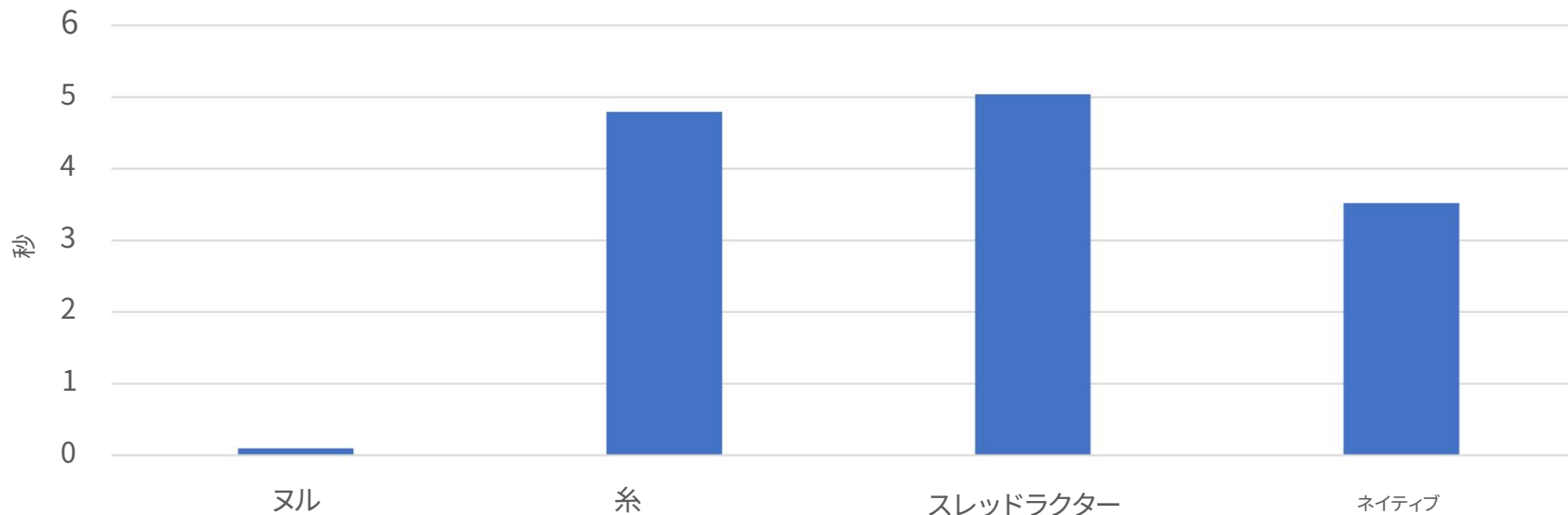
"タイムアウト"

ベンチマーク

- 「タイムアウト」コールの99%はタイムアウトしない

→ 測定: `N.times{timeout(1){null_task}}`

1Mのtimeout()呼び出しの実行時間



- null: タイムアウトなし (== タップ)
- thread: 元のタイムアウト
- thread_ractor: ソリューション 1
- native: ソリューション 3

- 「ネイティブ」は最速ですが、それほど速くはありませんか？

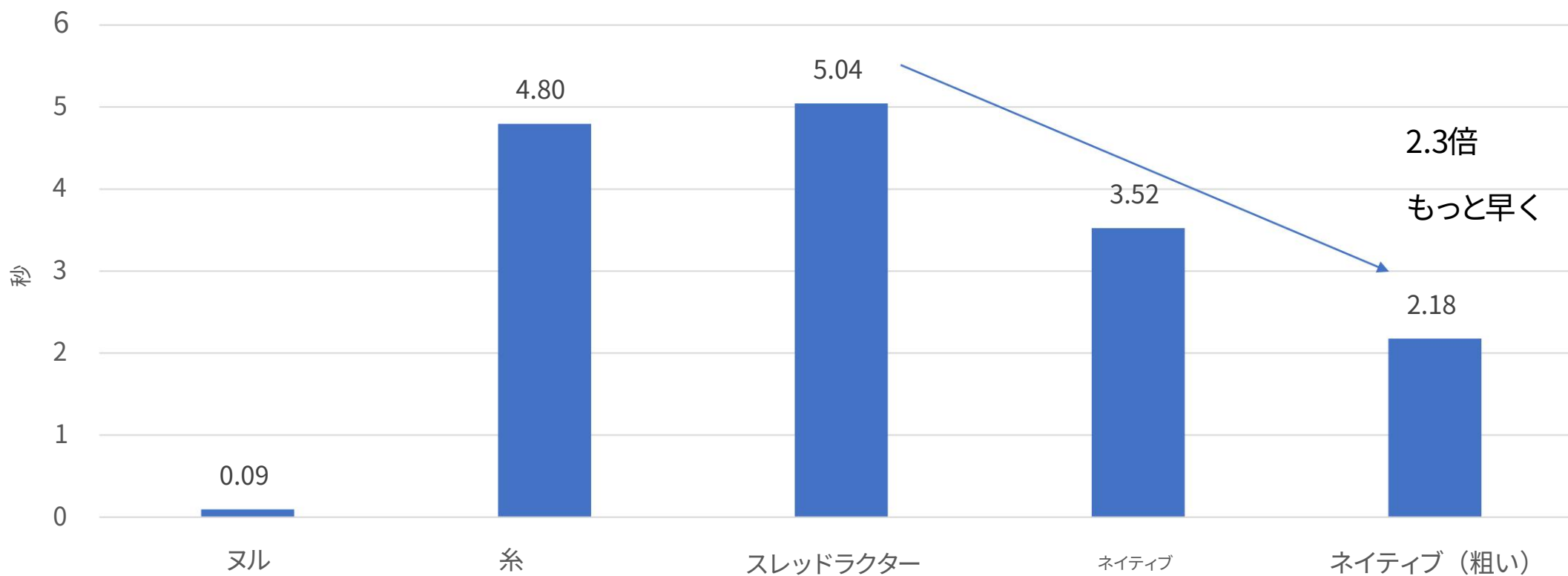
"タイムアウト"

ベンチマーク

- 「perf」はハードウェアタイマーへのアクセスに問題があることを示しています
 - スリープ期間を決定するために、
clock_gettime(CLOCK_MONOTONIC) が使用されます (100 万回)。
- CLOCK_MONOTONIC_CORSE (Linuxの場合)を使用すると、
 - 高速ですが、正確ではありません (Ubuntuでは最大4msの誤差があります) 。
この目的には十分です。

"タイムアウト" ベンチマーク

1Mのtimeout()呼び出しの実行時間

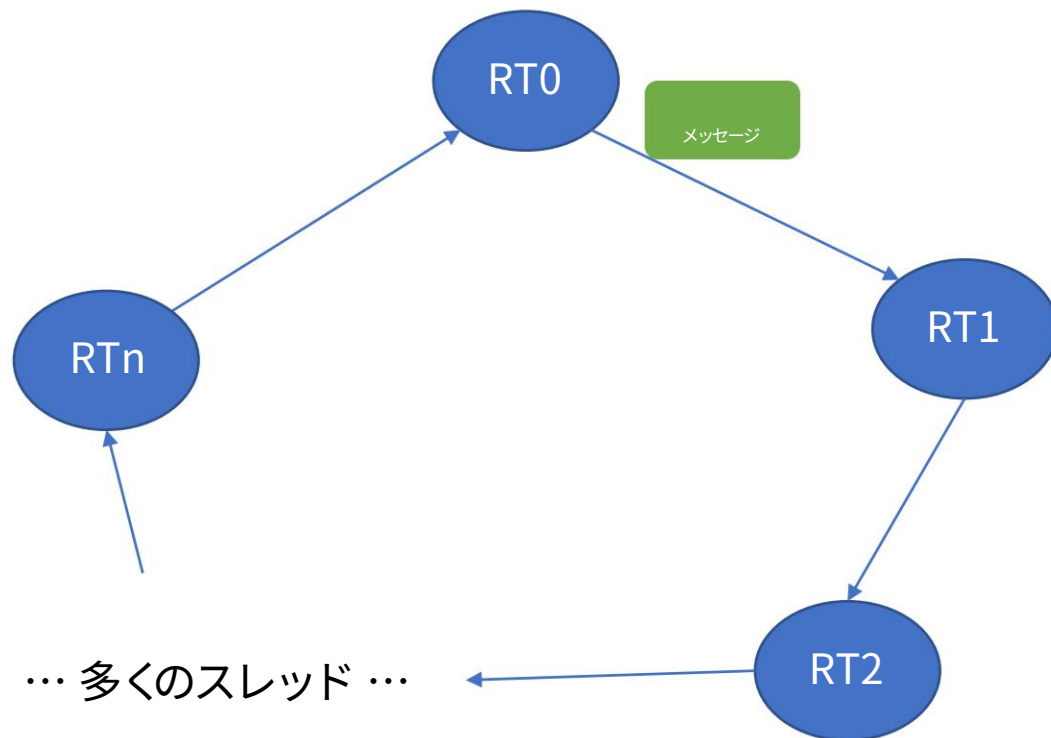


GC パフォーマンスの問題

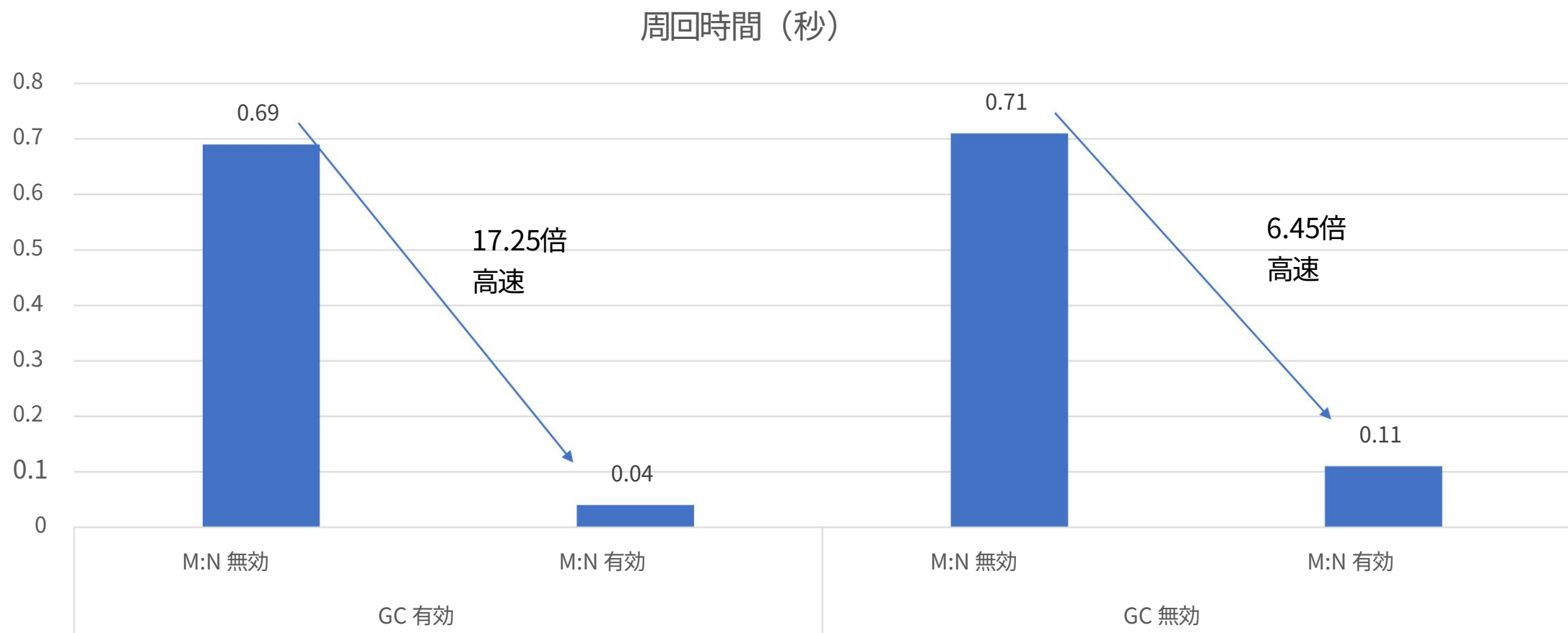
パフォーマンス調査と提案はまだありません

リングの例

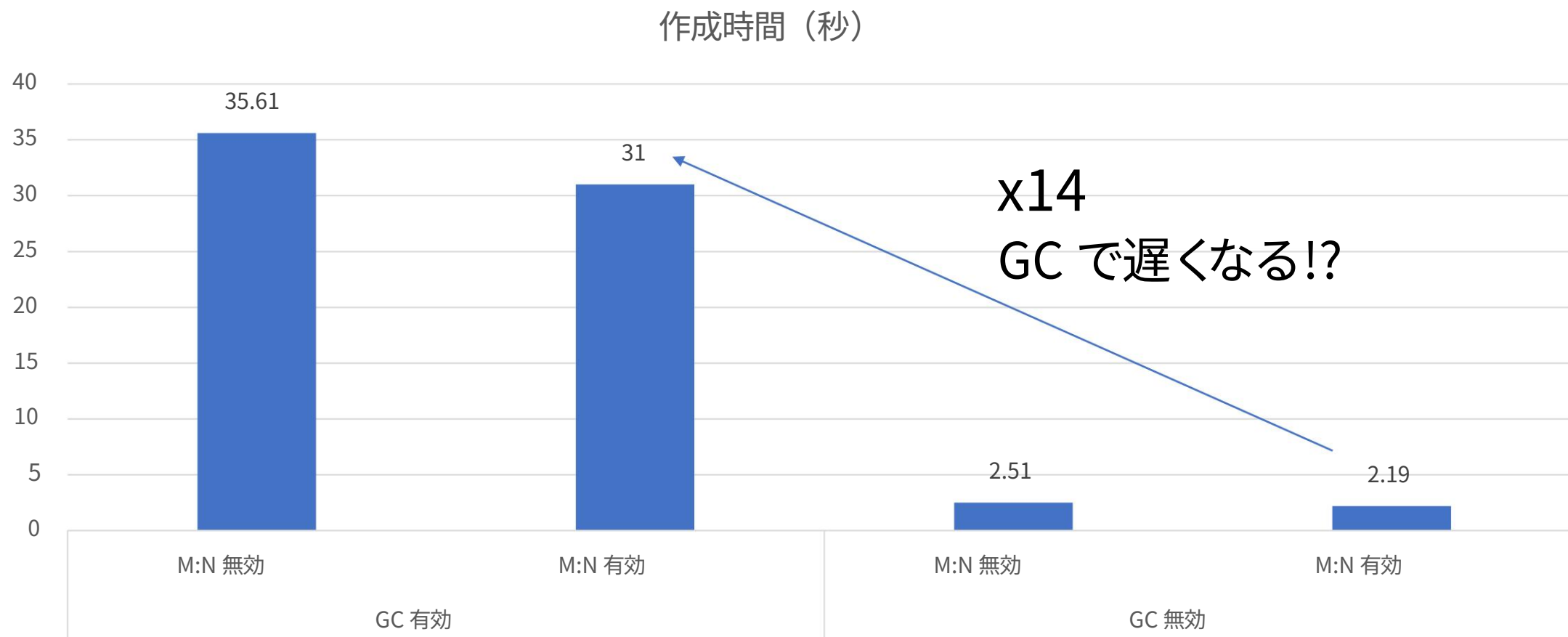
- 50,000個のRactorを作成する
- 次のRactorにメッセージ（オブジェクト）を送信し、一周する時間を測定します



リング例のベンチマーク結果



リング例のベンチマーク結果



データは「RubyにおけるM:Nスレッドの実装」、PPL2024より

Ractors の GC パフォーマンスの問題

- (1) ページ数が足りないため GC が多すぎる
- (2) 活動中のラクターの停止

…そしてさらに問題がありますか？

GC パフォーマンスの問題

(1) ページ数が足りないためGCが多すぎる

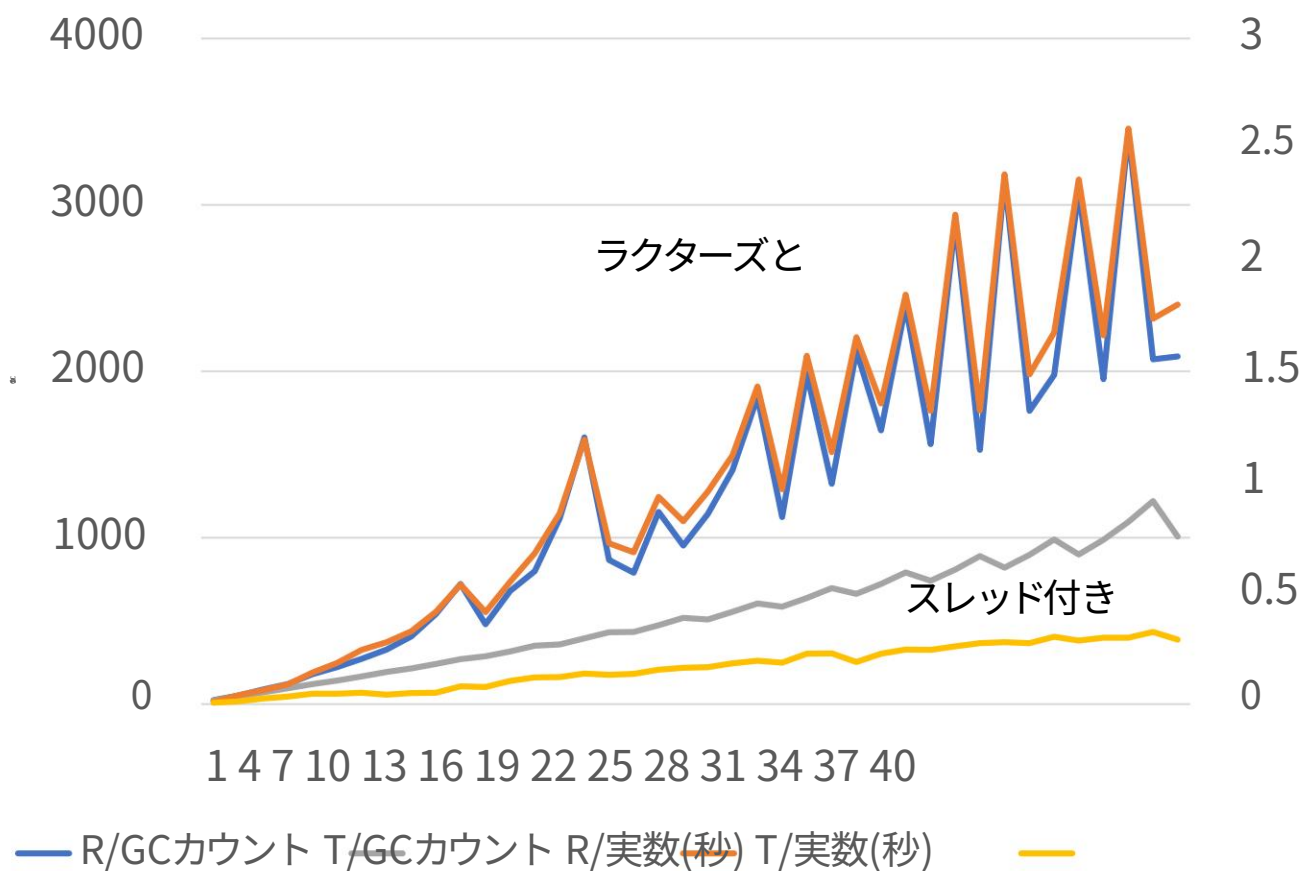
- 基準

- N個のラクタまたはスレッドを作成し、1Mのタスクを実行します。
配列
- `N.times.map{ Ractor.new{ task } }` • GC回数と実行時間は、 }

最悪N

- 並列実行により速度の向上が期待できます

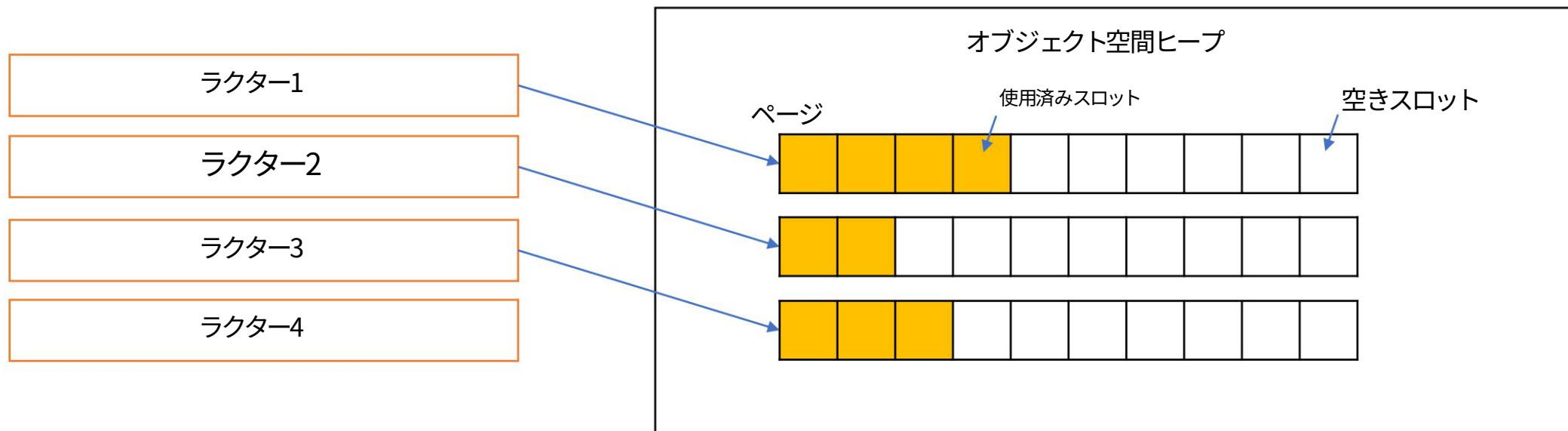
ベンチマーク結果



- 実行時間は GC 数と強く相関している
- ラクタの GC 数は明らかにスレッドよりも大きい
- Ractors の GC パフォーマンスはスレッドよりも遅いようです

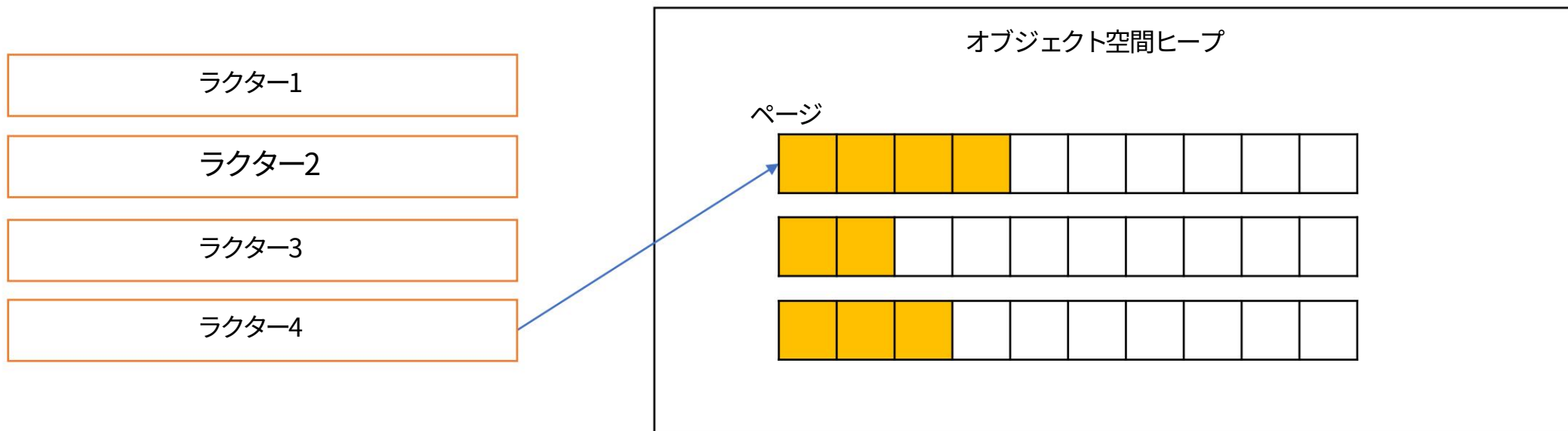
Ractor 上のオブジェクト割り当て

- Ractor上のオブジェクト割り当て時に、Ractorはヒープページを予約しました。 • オブジェクト割り当てごとの追加の同期を削除するため
- 3ページの場合、Ractor 4はページを予約しようとしませんが、ページがありません → 未使用の-slotが多数あってもGCを実行します。



Ractor 上のオブジェクト割り当て

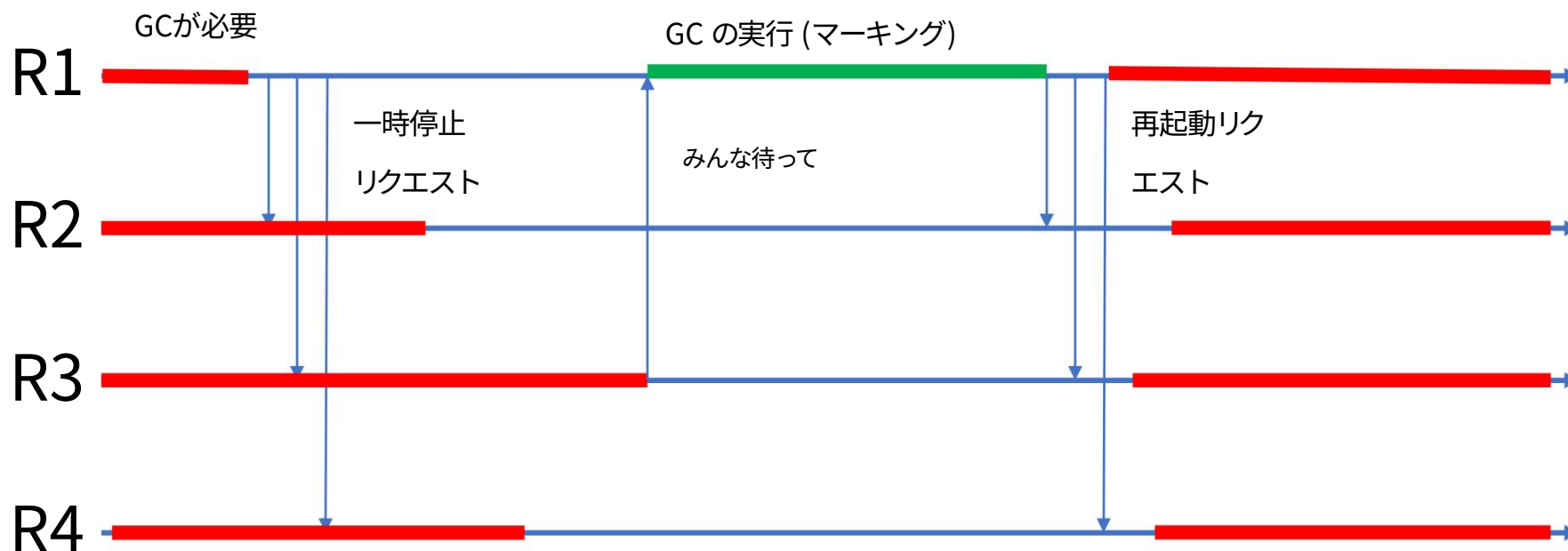
- Ractor 上のオブジェクト割り当て時に、Ractor はヒープページを予約します。
- オブジェクト割り当てごとの追加の同期を削除するため
- 3 ページの場合、Ractor 4 はページを予約しようとしていますが、ページがありません → 未使用の-slotが多
数あってもGC を実行します。



GC パフォーマンスの問題

(2) 活動中のラクターの停止

- ヒープ全体を走査する際に変更が起こらないように、各GCのバリア同期 (マーキング) を行う

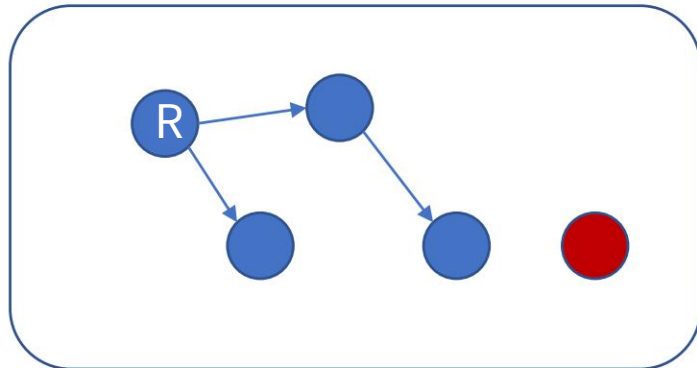


未来

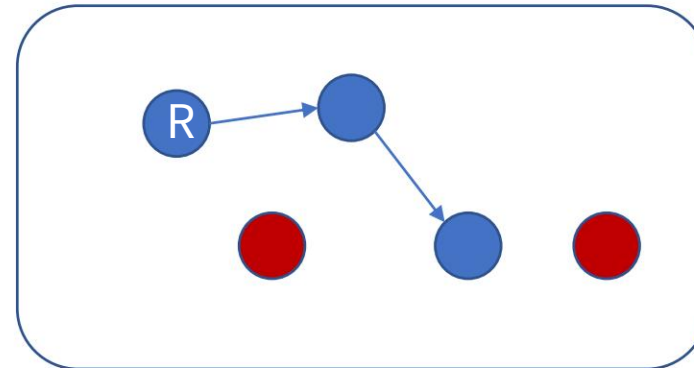
GC チューニング

- Ractorを考慮したGCチューニング
 - Ractorの数に応じて十分なページを用意する
- Ractor ローカル GC
 - 分散 GC 技術が必要• 単一ヒープよりも多くのメモリが必要

R1



R2



未来

この講演で提案された方法

- `Ractor#interrupt_exec` (および `Thread#interrupt_exec`) • `Ractor#main?`
- `Ractor.require(機能)`
- `Ractor::Channel.new`
- `RubyVM.timeout_exec(秒, プロシージャ)`
- もっと？

今日のトピック

- Ractors の重要な機能をサポート • “require” • “timeout”
- Ractor のメモリ管理の問題 • 将来の機能強化計画 • GC 戦略 • 提案された API

