

2024 年度 卒業論文

Ractor を用いた Ruby の並列処理性能評価と  
Rubocop による並列コード記述支援

Parallel Processing Performance Evaluation of Ruby Using  
Ractor and Parallel Code Writing Support Using Rubocop

成蹊大学理工学部情報科学科

ソフトウェア研究室

S202148 柳澤 快

## 目次

第 1 章 序論 .....	1
1.1. はじめに .....	1
第 2 章 Ractor の概要.....	3
2.1. Ractor について .....	3
2.2. Actor model .....	3
2.3. Ractor の仕様.....	5
2.3.1. インタープリタプロセス内の Ractor .....	5
2.3.2. Ractor におけるオブジェクト共有 .....	5
2.3.2.1. 不変オブジェクト (Immutable Objects) .....	5
2.3.2.2. クラス・モジュールオブジェクト (Class/Module Objects) .....	6
2.3.2.3. 特別な共有可能オブジェクト (Special Shareable Objects) .....	6
2.3.3. Ractor 間の通信プロトコル .....	6
2.3.3.1. プッシュ型通信 (Push Type Communication) .....	6
2.3.3.2. プル型通信 (Pull Type Communication) .....	7
第 3 章 実験 .....	8
3.1. 概要 .....	8
3.2. 実験環境 .....	8
3.3. 利用するプログラム .....	8
3.4. 本プログラムの流れ .....	8
3.5. 結果 .....	9
3.6. JIT コンパイラ .....	11
3.6.1. YJIT(Yet Another Ruby JIT) .....	11
3.6.2. RJIT .....	12
3.6.3 実験・結果 .....	12
第 4 章 Rubocop 概要 .....	15
4.1. Rubocop とは.....	15
4.2. 機能 .....	15
4.2.1. コードスタイルの検証 .....	16
4.2.2. コード品質のチェック .....	16
4.2.3. 自動修正 .....	16
4.3. rubocop-ast.....	16

4.4.	カスタムルール .....	17
第 5 章 Rubocop によるコード支援の実装 .....		20
5.1.	方針 .....	20
5.2.	実装 .....	21
5.2.1.	プッシュ型通信 .....	21
5.2.2.	オブジェクト共有 .....	24
5.2.3.	プル型通信 .....	26
5.2.4.	Ractor#take .....	28
第 6 章 結論 .....		31
参考文献 .....		33
謝辞	34	
付録	35	

# 第1章 序論

## 1.1. はじめに

Ruby は 1995 年に一般公開されたオブジェクト指向スクリプト言語である。Ruby は他の言語(Perl, Smalltalk, Eiffel, Ada, Lisp)の影響を受けて設計されている[1]。Ruby は多くの支持を集めているが、こうした成長は Ruby の Web アプリケーションフレームワーク Ruby on Rails の人気に起因している。Ruby on Rails は、開発工数を抑えられ、比較的短期間で目的のサービスやアプリ開発の実現を最大の特徴としていることから、Ruby は SaaS 企業やスタートアップ企業をはじめとして、数多くの採用例がある。Ruby にはさまざまな処理系があり、公式の処理系として MRI(Matze' Ruby Implementation)がある。これは、C 言語で実装された Ruby の公式処理系であり、プラットフォームを問わず、動作可能である。まつもとゆきひろ氏により開発され始め、最も広く使用されている。MRI 以外にも、JRuby, IronRuby, MacRuby, Rubinius などさまざまな Ruby 処理系が存在する。

近年、コンピュータの性能はマルチコアプロセッサの普及により向上している。それに伴い、ソフトウェアの並列処理能力が重要となっている。並列計算機上で複数の処理を同時に実行するために、多くのプログラミング言語では複数スレッドを同時実行させることができる。しかし、Ruby はグローバルインタープリターロック (Global Interpreter Lock: GIL) によって同時に実行可能なスレッドは 1 つのみとなっている。また、Ruby では GIL のことを一貫して GVL(Global VM Lock)と呼ばれている。これにより、スレッドを使う限り、通常の方法では並列プログラムを Ruby で記述することはできない。

Ractor の登場により異なる Ractor 間でスレッドを並列に実行することが可能になった。Ractor とは、スレッドの安全性を気にせずに Ruby の並列実行機能を提供するように設計された機構である。Ruby のフレームワークである Ruby on Rails のアプリケーションサーバーにおいても、リクエストを並列に処理することで更なるパフォーマンス向上が期待されている。また、Ruby コミュニティでは、コード品質を維持し、可読性や保守性を向上させるためのツールとして、静的解析ツール Rubocop が広く利用されている。Rubocop は、Ruby コミュニティで採用されているコーディングスタイルガイド (Ruby Style Guide) に基づき、コードのフォーマットや構文の一貫性を保つための自動チェックを提供する。これにより、開発者はスタイルの統一を図り、バグの発生を未然に防ぐことが可能となる。

現在、Rubocop は主に Ruby on Rails を含むアプリケーション開発において広く活用されているが、並列処理を対象とした支援には十分な機能が備わっているとは言い難い。特に、Ractor を活用した並列処理の記述においては、新しい並列モデルに適したコーディングス

タイルの確立が求められている。しかし、現状の Rubocop は、Ractor の使用に特化したルールはなく、開発者が適切な記述方法を模索しながら開発を行う必要がある。

本研究では、Ractor で記述された Ruby の並列処理の性能評価を行うことにより、Ractor のより良い記述方法を明らかにし、静的解析ツール Rubocop による Ractor を使用した記述への支援を目指す。さらに、Ractor を使用した記述時の注意点を Rubocop によって警告として実装することで、デバッグ時間の短縮や、Ractor の仕様や制約を開発者にとってより分かりやすくすることを目指す。

## 第2章 Ractor の概要

### 2.1. Ractor について

Ruby 3.0 において導入された Ractor は、並行および並列処理を安全に実現するための機構であり、Ruby における Actor Model の抽象化として設計されている。従来の Ruby 環境では、GVL により並列実行が制限されていたが、Ractor の導入により、異なる Ractor 間で GVL の制約を受けずに並列処理を実行することが可能となった。

Ractor の設計は、スレッドセーフな並列処理を提供することを目的としており、異なる Ractor 間でのデータ共有を禁止し、明示的なメッセージパッシングを用いた通信機構を採用している。このアプローチにより、従来のスレッドプログラミングにおいて問題となる競合状態 (race condition) やデッドロック (deadlock) を回避し、スレッド安全性を確保する。

Ractor の開発は当初 Guild という名称のもとで進められていたが、2020 年に正式に Ractor へと名称が変更された。Ruby の処理系である MRI が起動されると、デフォルトで 1 つの Ractor が作成され、これをメイン Ractor と呼ぶ[2]。メイン Ractor は Ruby プログラムのエントリーポイントとして機能し、他の Ractor を生成・管理する役割を担う。各 Ractor は最低 1 つのスレッドを持ち、同一 Ractor 内の複数スレッドは GVL により同時実行されることはない。

Ractor の導入により、マルチコア環境における Ruby アプリケーションのパフォーマンス向上が期待される一方で、従来のスレッドを表すクラス Thread とは異なるプログラミングモデルを採用する必要がある。これにより、並行・並列処理の設計に関する新たなベストプラクティスの確立が求められている。

### 2.2. Actor model

Actor model は、1973 年にマサチューセッツ工科大学の Carl Hewitt 氏によって発表された並列計算の数学的モデルの一種である。このモデルは、独立したアクターと呼ばれる単位が、メッセージのやり取りを通じてそれぞれのアクターが並列に処理を行うという特性を持つ。アクター同士は互いの状態を共有せず、メッセージパッシングによってのみ通信を行うため、競合状態の発生を回避しつつ、高い並列性とスケーラビリティを実現できる。特に、スケールアウトを得意とするクラウドインフラと相性がよく、分散システムや大規模データ処理の分野において、近年再び脚光を浴びるようになった。

このようなスケールアウト型の設計が注目を集める背景には、ムーアの法則の限界が挙げられる。ムーアの法則 (Moore's Law) は、1965 年にインテル社の共同創設者である Gordon Moore 氏によって提唱された経験則であり、「半導体の集積度は 2 年ごとに 2 倍になる」と

いうものである。この法則に基づき、半導体技術の進化により、コンピュータの処理能力は指数関数的に向上し、コストの削減も同時に進められてきた。ムーアの法則は、長年にわたりプロセッサの性能向上を支える基本的な指針となってきたが、近年ではその持続可能性が疑問視されている。

スケールアウトとは、単一のプロセッサの性能向上に依存するのではなく、複数のプロセッサやノードを並列に動作させることで、全体としての処理能力を向上させる手法である。特に、クラウドコンピューティングの普及により、動的なリソースの追加・削除が容易になったことで、スケールアウトを前提としたアーキテクチャの重要性が増している。この流れを受け、ソフトウェア開発においても、並列性と分散処理を前提とした設計が求められており、Actor model はその有力な解決策として再評価されている。

Actor model に基づいたプログラムは、各アクターが独立した状態を持ち、メッセージを通じて非同期に動作するため、インフラのスケールアウトに応じた拡張が容易である。これにより、リソースの追加による処理能力の向上が可能となり、システム全体の効率的な並列処理が実現される。また、アクターは状態をカプセル化し、直接的な共有を避けるため、ロック機構に依存することなく安全な並行実行が可能となる。この特性は、近年のマルチコアプロセッサの普及やクラウドネイティブな環境において、大きな利点となっている。

Actor model の構成は図 1 に示す。

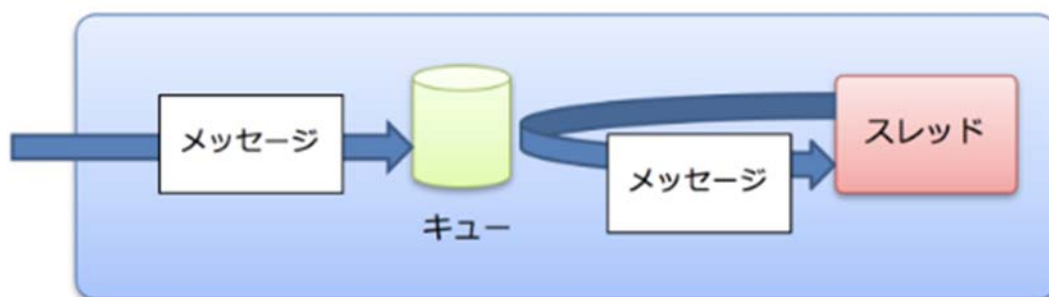


図 1 : Actor model の構成[3]

Actor model において、アクターはクラスから生成されたインスタンスであり、そのインスタンスにはスレッドとキューが関連づけられている。アクターモデルを利用する場合、キュー内のメッセージを短時間で大量に処理したいという要求が生じた場合、スケーラビリティを向上させるためにアクターを増加させることが可能である。このようなスケールアウトは非常にシンプルであり、アクターを1つ追加するだけで達成することが可能である。これは、アクター自体が内蔵するキューを持っているため、排他制御を意識する必要がないという利点がある。キューに格納されたメッセージを順次、処理する。

Actor model を採用している有名なプログラミング言語として、Erlang や Elixir がある。また、Scala や Java のフレームワークである Akka も Actor model を採用した有名な例であ

る。

## 2.3. Ractor の仕様

以下、Ractor の仕様について一部を紹介する[4]。

### 2.3.1. インタプリタプロセス内の Ractor

Ruby では、1 つのインタプリタプロセス内で複数の Ractor を生成し、それらを並列に実行することが可能である。

`Ractor.new{ 式 }` を使用すると、新しい Ractor が作成され、その内部で指定された式が並列実行される。通常、Ruby インタプリタはメイン Ractor と呼ばれる最初の Ractor を起動し、プログラムの実行が進行する。

メイン Ractor が終了すると、スレッドと同様に、すべての Ractor に終了リクエストが送信される。これは、メインスレッド（最初に起動されたスレッド）が終了すると、Ruby インタプリタがすべての実行中スレッドに終了指示を出す仕組みと同様である。

各 Ractor は 1 つ以上のスレッドを内部に持つ。ただし、Ractor 内部のスレッドは、MRI における GVL の影響を受けるため、GVL を C レベルで明示的に解放しない限り、並列実行はできない。一方、異なる Ractor に属するスレッド同士は並列に実行可能である。

Ractor の生成コストは、スレッドを 1 つ作成する際のオーバーヘッドと同程度であるとされる。

### 2.3.2. Ractor におけるオブジェクト共有

Ruby の Ractor は、従来のスレッドとは異なり、すべてのオブジェクトを共有するわけではない。この設計により、オブジェクトの共有に起因するスレッドセーフティの問題を最小限に抑えることが可能となっている。

Ractor では、オブジェクトは共有可能 (Shareable) と共有不可 (Unshareable) の 2 種類に分類される。通常、ほとんどのオブジェクトは共有不可とされており、その結果、異なる Ractor 間でデータの不整合が発生するリスクが軽減される。一方で、特定の条件を満たしたオブジェクトは共有可能となり、異なる Ractor 間でのやり取りが可能となる。

共有可能なオブジェクトとして、以下の 3 種類が挙げられる。

#### 2.3.2.1. 不変オブジェクト (Immutable Objects)

不変オブジェクトとは、変更が加えられないことが保証されたオブジェクトである。具体的には、オブジェクトが `freeze` されており、かつ内部に共有不可オブジェクトを保持してい



ない場合、これを不変オブジェクトとみなすことができる。

例えば、以下のオブジェクトは不変とみなされる：

- `i = 123`（数値は変更不可能なため、不変オブジェクト）。
- `s = "str".freeze`（文字列を明示的に freeze することで不変化）。

一方で、以下のようなオブジェクトは不変とはならない：

- `a = [1, [2], 3].freeze`（配列全体を freeze しても、内部に変更可能な要素 [2] を含むため不変オブジェクトではない）。
- `h = {c: Object}.freeze`（ハッシュ全体を freeze しても、内部に `Symbol:c` やクラス `Object` が含まれている場合は不変とみなされる）。

#### 2.3.2.2. クラス・モジュールオブジェクト (Class/Module Objects)

Ruby のクラスやモジュールは本質的に共有可能なオブジェクトとみなされ、Ractor 間で安全に利用できる。これにより、複数の Ractor が同一のクラスやモジュールを参照し、メソッドの実行が可能となる。

#### 2.3.2.3. 特別な共有可能オブジェクト (Special Shareable Objects)

Ractor オブジェクト自体や、一部の組み込みオブジェクトは、特別に設計されており、異なる Ractor 間で共有することが許可されている。これにより、必要最小限の情報共有を安全に行うことができる。

### 2.3.3. Ractor 間の通信プロトコル

Ractor は、メッセージ交換による通信を通じて相互作用し、実行を同期する仕組みを持つ。この通信には 2 種類のプロトコルが存在し、それぞれプッシュ型 (push type) とプル型 (pull type) と呼ばれる。これらの通信方法を活用することで、柔軟かつ効率的な並列処理を実現している。

#### 2.3.3.1. プッシュ型通信 (Push Type Communication)

プッシュ型通信は、メッセージの送信と受信が非同期に行われるプロトコルである。この方法では、送信側の Ractor が特定の受信側 Ractor に対してメッセージを送信し、受信側がメッセージを受け取る仕組みとなっている。

- 送信と受信の仕組み

送信側は `Ractor#send(obj)` を用いて対象 Ractor にメッセージを送信し、受信側は `Ractor.receive` を用いてメッセージを受け取る。送信側は対象の Ractor (宛先) を特定するが、受信側は送信元を特定せず、どの Ractor からのメッセージも受け入れる仕様

となっている。

- キューを用いた非同期性

受信側 Ractor は無限のキューを持ち、送信側はこのキューにメッセージを非同期に追加する。この仕組みにより、送信側はブロックされことなくメッセージを送信できる。なお、`Ractor.receive_if{ 条件式 }` を使用することで、特定の条件に一致するメッセージを選択的に受け取ることも可能である。

### 2.3.3.2. プル型通信 (Pull Type Communication)

プル型通信は、送信側と受信側が明確に対となるプロトコルである。この方法では、送信側がデータを提供する準備が整った際に受信側がデータを取得し、通信が完了する。

- 送信と受信の仕組み

送信側は `Ractor.yield(obj)` を用いてデータを提供し、受信側は `Ractor#take` を用いてデータを取得する。プッシュ型とは異なり、送信側は宛先 Ractor を知らず、受信側が送信元の Ractor を特定する形となる。

- 同期的な動作

プル型通信では、送信側または受信側のいずれか一方が通信相手を待機していない場合、通信がブロックされる。これにより、明確なデータフロー制御が可能となる。

## 第3章 実験

### 3.1. 概要

バブルソートを用いて Ractor の実験を行った。筆者は、C 言語と MPI(Message Passing Interface)を用いて、バブルソートの並列化を行なった経験があり、要素数  $N$  を 2 並列でソートさせたときにほぼ  $1/4$  の実行時間で並列ソートを完了させることが出来た経験があるため、Ractor でもこれを用いることにした。

### 3.2. 実験環境

OS	MacOS Sonoma 14.5
CPU	Apple M1 8-Core
Ruby Version	3.3.4

### 3.3. 利用するプログラム

今回利用するプログラムは付属プログラム `bubble_sort_ractor.rb` を利用する。

### 3.4. 本プログラムの流れ

また、本プログラムの流れを図 2 に示す。

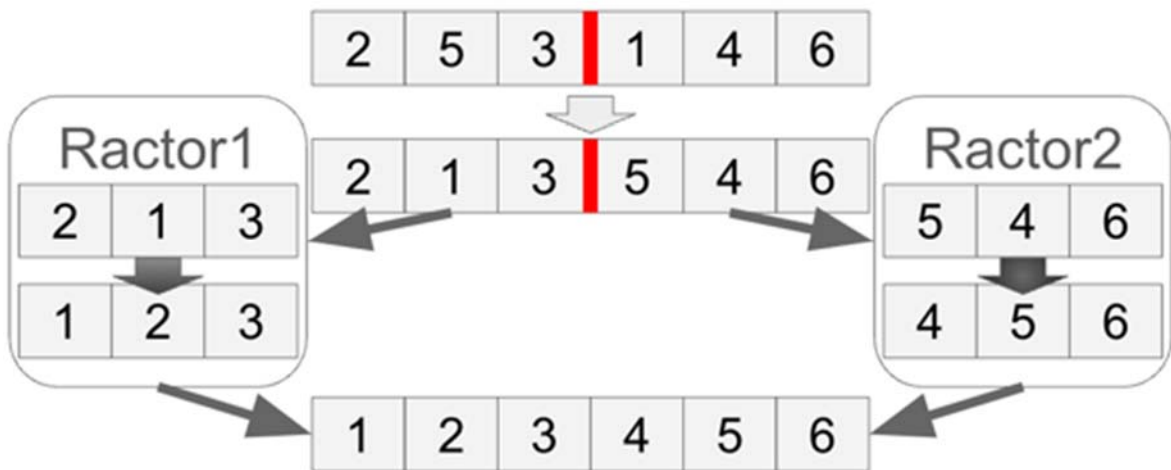


図 2：バブルソートの並列化

1. 配列要素の中央値を基準に次の手順に従い 2 つに分割する。
2. 配列の左端から右に向かって、中央値以上の値を探し、右端から左に向かって中央値未満の値を探索する。
3. 見つけた 2 つの値を交換し、これにより、中央値未満の値が左側に、中央値以上の値が右側に移動する。
4. この操作を左右からの探索が衝突するまで繰り返す。
5. 2 つの Ractor を生成し、各 Ractor に分割された部分配列を送信する。
6. その後、各 Ractor で昇順にバブルソートを実施する。
7. 各 Ractor からソートされた配列を受け取り、それらを結合する。

バブルソート（Bubble Sort）は、隣接する要素を比較しながら並び替えることでリストをソートする基本的なソートアルゴリズムである。以下に、バブルソートの基本的な手順を示す。

1. リストの先頭から末尾に向かって、隣り合う要素を比較する。
2. 比較した要素がソート条件を満たしていない場合（例：昇順ソートにおいて前の要素が後の要素より大きい場合）、これらの要素を交換する。
3. リストの末尾まで到達したら、末尾の要素は確定するため、それを除いた範囲で再び 1 から 2 の操作を繰り返す。
4. この操作をリスト全体がソート済みになるまで続ける。

### 3.5. 結果

実行時間を測定した結果について、表 1 に示す。Ruby のみの記述による逐次実行、Ractor を 1 つ生成し逐次実行行った場合、そして、Ractor を 2 つ生成し、前項の方法で並列実行

した 3 種類の実行時間の結果である。

表 1 : 3 種類の実行時間の結果

data size	50	500	5,000	50,000	100,000
sequential [sec]	0.000186	0.0213	1.8004	187.6308	761.7486
ractor1 [sec]	0.000885	0.0199	1.8283	187.2024	747.6704
ractor2 [sec]	0.000802	0.0077	0.6294	65.2751	286.7740

バブルソートの計算量は $O(n^2)$ である。並列化した際に最速になる場合を考えると、要素数が半分の $\frac{n}{2}$ であるため、計算量は $O\left(\frac{n^2}{4}\right)$ となる。つまり、逐次実行時よりも 4 倍の速度向上が予測される。しかし、結果としてはデータ数が 50,000 の時の約 2.87 倍が並列化効率の最もよい場合であった。データ数が少ない場合では、Ractor のオーバーヘッドやデータ分割処理により、実行時間が逐次実行に比べて並列実行は 4.3 倍ほど遅くなった。

次に、Ractor のどの箇所でオーバーヘッドがあるのかを Ractor の各処理における実行時間を測定することで調べた。前実験のデータ数 50,000 の並列化について各 Ractor の処理の実行時間を計測した結果を表 2 に示す。

表 2 : Ractor の各処理における実行時間

データ数: 50,000, 分岐点: 24,883, 単位: [sec]

divide 0.00304	new	send	receive	bubble sort	concat 0.00007	総時間 65.27517
	ractor1					
	0.00086	0.00011	0.00017	64.62844		
	ractor2					
	0.00002	0.00011	0.00025	65.27062		

結果として、実行時間の約 99%をバブルソートに費やしている。さらに、ractor1 の生成時間が明らかに ractor2 よりもかかっていることがわかる。さらに Ractor を 2 つ生成し、計 4 つの各 Ractor の生成時間を計測したが、ユーザーが生成する 1 つ目に生成される Ractor は他の Ractor の生成に比べて生成時間がかかることがわかった。

また、データ数 50,000 の並列化が最速になる場合、各 Ractor に 25,000 のデータが送られバブルソートされることになる。試しに、データ数 25,000 で Ractor が 1 つのときの逐次実行を計測した結果、約 46[sec]になることがわかった。さらに、2 つの Ractor による並列実行について、一方の Ractor のバブルソート処理を行わないようにすると約 46[sec]の実行時間が得られた。このことから、複数の Ractor 間で配列の各要素に対してスワップ処理を行う部分では、各 Ractor で処理時間が遅くなることがわかる。

この要因としては次の可能性が考えられるが引き続き調査が必要である。Ruby の配列は可変長配列であるため、ソートする際のスワップ処理を大量に繰り返すことでメモリの再配置が頻繁に発生する。ガベージコレクションが頻発することによる速度低下である。これについてはガベージコレクションを停止して実行しても効果は得られなかった。さらに、Ractor はスレッド安全を保証するために、共通の配列要素へのアクセス時に排他制御が働き、結果として並列に実行されない可能性も考えられる。この場合、排他制御のオーバーヘッドが発生し、並列処理の性能向上が妨げられている可能性がある。

## 3.6. JIT コンパイラ

Ruby は JIT(Just-In-Time)コンパイラの機能を備えている。有効化することにより、実行時に機械語を生成し、最適化が行われ、実行が高速になる [5]。Ruby3.3 には YJIT と RJIT の 2 つの JIT コンパイラがあるが、どちらもデフォルトでは無効になっている。そのため、JIT コンパイラを有効化して再度同様の実験を行った。

### 3.6.1. YJIT(Yet Another Ruby JIT)

特に Ruby の実行速度を大幅に向上させることを目指して開発された。YJIT は、Rust で書かれているため、起動時に Rust コンパイラが存在していないと有効化することはできない。アプリケーションのコードに変更を加える必要がないまま、コンパイラの最適化技術を用いてパフォーマンスを向上させるため、多くの開発現場で注目されている。特に、Web アプリケーション開発で広く使われる Ruby on Rails においては、YJIT を利用することで、実行速度が最大 40%向上するケースも報告されている。

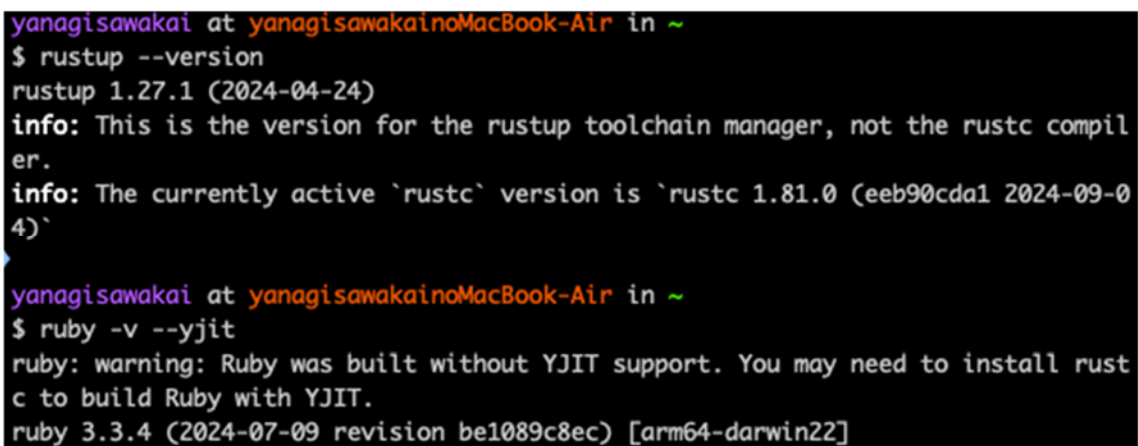
### 3.6.2 RJIT

RJIT は、JIT コンパイラの最適化実験を支援する環境を提供することを主な目的としている。そのため、本番環境では主に YJIT の導入が推奨されている。Ruby で書かれた JIT コンパイラを提供している。Ruby 3.2 では、MJIT という別の JIT コンパイラが存在していたが、これに置き換わる形で RJIT は Ruby 3.3 に導入された。MJIT は Ruby 2.0 に比べて 3 倍の高速化を達成したが、その性能を圧倒する YJIT の登場により導入例が減少してしまった。しかし、MJIT は Ruby で書かれていたため記述が容易であり、YJIT に比べて実装コストが低く、モンキーパッチすることにより、独自の JIT コンパイラを実装することができるという利点があった。しかし、実行時に C コンパイラを起動する仕様であり、その仕様から特殊な実装がされていたため、保守が難化してしまう問題があった。その問題を解決し、MJIT の利点を受け継ぐ RJIT が YJIT 開発を助けることを目的に開発された。

### 3.6.3 実験・結果

上記で述べた通り、YJIT は本番環境に推奨されており、より性能が高いことから、今回は YJIT を使用して実験を行った。

まず、実験では、YJIT を有効化するために Rust 環境を構築した。Rust の環境構築を完了し、YJIT が有効化されているかの確認を行った。確認の結果を図 3 に示す。Rust 環境が正常に導入されているが、Ruby 側で Rust コンパイラの認識ができていなかった。



```
yanagisawakai at yanagisawakainoMacBook-Air in ~  
$ rustup --version  
rustup 1.27.1 (2024-04-24)  
info: This is the version for the rustup toolchain manager, not the rustc compiler.  
info: The currently active `rustc` version is `rustc 1.81.0 (eeb90cda1 2024-09-04)`  
  
yanagisawakai at yanagisawakainoMacBook-Air in ~  
$ ruby -v --yjit  
ruby: warning: Ruby was built without YJIT support. You may need to install rustc to build Ruby with YJIT.  
ruby 3.3.4 (2024-07-09 revision be1089c8ec) [arm64-darwin22]
```

図 3: Rust version と YJIT の有効化確認

Ruby は主に、rbenv によって管理されている。rbenv とは、Ruby のバージョン管理ツールである。プロジェクトごとに異なるバージョンを簡単に切り替えられ、シンプルで軽量な

設計であることから、広く使用されている。この rbenv を使用し、Ruby 3.3.4 の再インストールを行うことにより、YJIT の有効化を確認した。YJIT の有効化の確認結果を図 4 に示す。

```
$ ruby -v --yjit
ruby 3.3.4 (2024-07-09 revision be1089c8ec) +YJIT [arm64-darwin23]
```

図 4：YJIT の有効化確認

YJIT の有効化したため、これを使用して実験を行った。実験は、バブルソートを逐次実行と 2 つの Ractor による並列実行するプログラムを使用し、YJIT を無効化した場合と有効化した場合のそれぞれ 4 通り行った。実験の結果を表 3 に示す。

表 3：YJIT の有無によるバブルソート実行時間

data size	50	500	5,000	50,000	100,000	500,000
sequential [sec]	0.000186	0.0213	1.800	187	761	18800
sequential jit [sec]	0.000893	0.0107	0.823	83.9	282	6320
ractor2 [sec]	0.000802	0.0077	0.629	65.3	266	8360
ractor2 jit [sec]	0.00207	0.0054	0.282	33.8	155	3740

データ数が 50 個の場合は、YJIT が無効化されていた方が実行速度が速いことが確認された。これは、プログラムの実行時にバイトコードをネイティブコードに変換し、このプロセスがオーバーヘッドとなっているため、データ数が小さい場合は、Ruby インタプリタをそのまま実行する方が YJIT を挟むよりも効率的であると考察される。データ数が 500 個以上の場合は、YJIT を有効化した方が、高速であることが確認された。また、YJIT が有効になっている場合、無効化されている場合に比べて、データ数が多くなるにしたがい、速度性能の向上倍率が大きいことがわかった。さらに、YJIT の有無による並列実行と逐次実行の性能倍率について図 5 に示す。



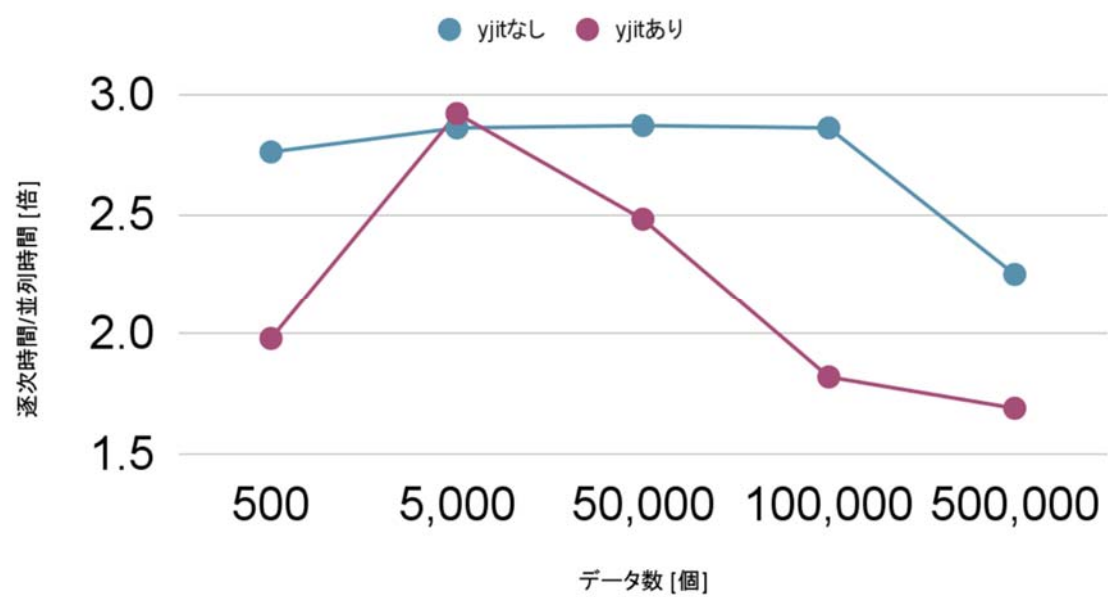


図 5: YJIT の有無による並列実行と逐次実行の性能倍率

## 第4章 Rubocop 概要

### 4.1. Rubocop とは

プログラミングにおけるコード品質の向上は保守性や開発効率を高めるための重要な課題である。特に、動的型付け言語である Ruby は、その柔軟性ゆえにコードスタイルやエラーの検出が開発者間で一貫しない場合が多い。この問題を解決するために登場したのが、Ruby 向けの静的解析ツールである Rubocop である。

Rubocop は、Ruby で記述されたコードの静的解析ツールであり、コードの品質やスタイルガイドの遵守をサポートするためのライブラリである。Ruby コミュニティで広く使用されており、Ruby のコーディング規約である Ruby Style Guide を基に、コードが適切に書かれているかを自動的に解析する。また、コードのフォーマットや構文上の問題を検出・自動修正する機能を提供する。Rubocop は主に、以下の目的で使用される：

- コードスタイルの一貫性の維持：開発チーム間でスタイルの統一を図る。
- 潜在的なバグの発見：コードの非推奨な構文や誤りを事前に検出する。
- コードの可読性向上：規定されたスタイルに従うことで、コードを直感的に理解しやすく、プロジェクト全体で記述を統一する。

また、Rubocop は gem という形式で公開されている。gem は RubyGems と呼ばれる Ruby 専用のパッケージ管理システムによって管理されている。Ruby 1.9 以降に組み込まれている。Ruby のライブラリは主に RubyGems.org に gem として置かれている[6]。そのため、Ruby の開発ではライブラリ指す際に gem と呼ぶことがある。Ruby のエコシステムを支える中核的な存在であり、効率的な開発やコミュニティの活性化を実現する重要な仕組みである。

前述した通り、Ruby にはこの RubyGems を扱うためのライブラリが組み込まれており、これを扱うために gem コマンドが提供されている。Rubocop の導入には、この gem の仕組みを活用する。Ruby 環境があれば、gem install 'gem パッケージ名'を実行することで、すぐに rubocop を使用することが可能である。CLI で rubocop を実行することにより、プロジェクト内のコード解析を行うことが可能である。また、ローカル環境の実行だけでなく、大規模な Ruby プロジェクトにおいて、Rubocop を CI/CD パイプラインに組み込むことで、自動的にコードスタイルをチェックするという使用方法も存在する。

### 4.2. 機能

Rubocop の機能は大きく以下の 3 つに分類される。さらに、その他の解析ルールについ

でも gem として多数、存在している。その代表例として、Web アプリケーションフレームワークである Ruby on Rails に関するルールを Rubocop として提供する rubocop-rails やテストフレームワークである RSpec のルールを提供する rubocop-rspec などがある。

#### 4.2.1. コードスタイルの検証

Rubocop は、コードがスタイルガイドに準拠しているかをチェックするための多くのルールを提供している。簡単な例として、以下のようなルールが含まれる：

- インデント：スペースの数やタブの使用方法の確認。
- 行の長さ：1 行あたりの文字数制限（デフォルトでは 120 文字）。
- 変数名やメソッド名の命名規則：スネークケースやキャメルケースの適用。

#### 4.2.2. コード品質のチェック

Rubocop は、潜在的なバグや非効率なコードパターンを検出する機能も提供する。以下のような例があがる：

- 未使用の変数やメソッド。
- 冗長なコードや非推奨構文の使用。
- 性能に影響を与える可能性があるコード。

#### 4.2.3. 自動修正

Rubocop は検出した問題を自動的に修正する機能を提供する。rubocop コマンドの autocorrect オプションを使用することで使用可能である。この機能は、フォーマットに関連する問題を効率的に解決するため、開発者の負担を大幅に軽減する。

### 4.3. rubocop-ast

rubocop-ast とは Rubocop のバックエンドライブラリであり、gem として提供されている。Ruby コードを抽象構文木(AST: Abstract Syntax Tree)し、その解析を可能にする。AST は、プログラムの構文を木構造で表現したもので、構文解析や静的コード解析の基盤として広く利用されている。Rubocop は rubocop-ast を使用し、効率的にコード解析を行えるようにしている。

また、Ruby コードを解析可能なデータ構造に変換するだけでなく、Rubocop 上での解析を用意するための拡張機能を提供している。通常の AST と比較して、rubocop-ast は次のような追加機能を持っている。

- メソッド群

AST ノードを操作するためのメソッドが豊富に提供されている。メソッドの例を、[図 6](#) に示す。メソッド呼び出しのノードに対して、レシーバ(メソッドの呼び出し元)や引数を簡単に取得することが可能である。

```
node.receiver # メソッド呼び出しのレシーバを取得
node.arguments # メソッド呼び出しの引数を取得
```

図 6: AST ノードの操作メソッド

- パターンマッチング機能

ノードの種類や構造を簡潔に判定するための DSL(ドメイン固有言語)が利用できる。これにより、解析対象を正確に絞り込むことが可能である。パターンマッチングの例を [図 7](#) に示す。

```
node.match?('send nil? :puts ...')
```

図 7: AST のパターンマッチング

`match?` は `rubocop-ast` による AST パターンマッチングの機能である。このメソッドを使用すると、特定の構文パターンにノードが一致するかを確認することが可能である。`'(send nil? :puts ...)'` は、AST ノードのパターンを表現する パターン文字列 である。それぞれの部分は以下のように分解できる。`(send ...)` ノードは `send` ノード(メソッド呼び出し)を表す。Ruby のコードにおけるメソッド呼び出しは AST では `send` ノードとして表現される。`nil?` ではレシーバが `nil` であることを意味する。Ruby では、レシーバが省略されたメソッド呼び出しは内部的にレシーバが `nil` として扱われる。`puts` は呼び出されたメソッド名を示す。この場合は `puts` メソッドが呼び出されていることを意味する。`...` は可変長引数を表す。この部分は「0 個以上の引数」を意味する。

## 4.4. カスタムルール

Rubocop は、特定のルールを担当しているモジュールがある。例として、以下のカテゴリがある。

- Style: スタイルに関するルールを検証。
- Lint: 潜在的なバグや不具合を検出。
- Metrics: 複雑度やコード量を測定。

他にも多くのルールが提供されているが、プロジェクト固有のスタイルや特別な要件を反映するために、既存のルールでは不十分な場合がある。このような場合、Rubocop は独自の

ルールを定義するカスタムルールの作成を可能にしている。これは、Rubocop の既存ルールに加えて、独自のルールを定義するための拡張機能である。これにより、Rubocop は標準ガイドラインに加えて、柔軟にカスタマイズされたコード検証を提供する。

カスタムルールを作成するには、Rubocop の内部構造と rubocop-ast による AST 解析を理解する必要がある。カスタムルールは、Rubocop の内部で用いられる AST を解析し、特定の条件に一致するノードを検出する仕組みで動作する。AST は、Ruby のコードを構造化した木構造として表現したもので、各ノードはプログラムの構文要素であるメソッド呼び出し、条件分岐、変数の代入などを表す。

まず、作成するルールの目的を定義する。どのような問題を検出し、どのようなスタイルを強制するかを明確にする。これにより、ルールの実装方針を定める。

カスタムルールは、RuboCop::Cop::Base を継承したクラスとして実装する。このクラスでは、on\_<node\_type>という形式のメソッドを定義し、メソッド呼び出しや変数定義などの特定のノードタイプを解析する[7]。カスタムルールの実装方法を図 8 に示す。

```
module RuboCop
  module Cop
    module Custom
      class MethodName < Base
        MSG = ' Do not prefix reader method names with number.'

        def on_send(node)
          return unless bad_reader_name?(node)
          add_offense(node, message: MSG)
        end

        private

        def bad_reader_name?(node)
          node.method_name.to_s.start_with?(/¥d/)
        end
      end
    end
  end
end
```

図 8： カスタムルールの実装例

このルールは、数字で始まるメソッド名を検出し、警告を出力するものである。カスタムルールの実装は Rubocop の Cop モジュール内に新しいサブクラスを定義する。この例では、`RuboCop::Cop::Custom::MethodName` として実装している。この構造により、Rubocop の他のルールと一貫性を保ちながら拡張可能な形でカスタムルールを構築することができる。

MSG には違反が検出された際に出力される警告メッセージを指定する。本例では `'Do not prefix reader method names with number.'` がメッセージとして設定されている。これにより、開発者が違反内容を明確に理解することができる。

また、Rubocop では、Ruby コードを AST として解析する。on\_send メソッドは、メソッド呼び出しノードを処理するために利用される。このメソッド内で、検出対象ノードを判別するための条件を設定する。このノードタイプメソッドは、rubocop-ast にて、あらかじめ提供されている[8]。また、on\_send メソッド内の node は解析中の AST ノードを表し、add\_offense メソッドで違反箇所を記録し、警告メッセージを表示するためのメソッドである。

bad\_reader\_name?メソッドは違反条件を定義している。違反条件は Rubocop で特別提供されているものではないため、自ら実装する必要がある。本例では、node.method\_name にてメソッド名を取得し、to\_s\_star\_with? (/¥d/) でメソッド名が数字で始まるかを判定する。return unless bad\_reader\_name?(node) にて、bad\_reader\_name?メソッドの返り値が false の場合、早期にメソッドを終了させる。これを一般にガード節と呼ぶ。

以上のような流れでカスタムルールを作成することが可能である。

## 第5章 Rubocop によるコード支援の実装

### 5.1. 方針

Ruby における並列処理のための新しいモデルとして導入された Ractor は、スレッドセーフな並列処理を実現するための強力な仕組みを提供する。しかし、Ractor は新しい構文や制約を持ち、これを適切に理解する学習コストが必要であり、逐次処理を記述するのに比べて難しい。具体的には、以下のような特徴と課題がある。

- オブジェクト共有の制約  
Ractor は並列処理においてデータの一貫性を維持するため、オブジェクトの共有に厳格な制約を設けている。この制約を守るためのルールは複雑であり、コード記述において頻繁にエラーを引き起こす可能性がある。
- 新しい API と構文の学習コスト  
Ractor は従来の Ruby コードとは異なる API や構文を用いるため、既存の開発者にとっては新しい知識の習得が必要となる。これにより、初学者や既存のプロジェクトへの適用が難しくなることがある。
- 並列処理のバグ発生リスク  
並列処理には競合状態やデッドロックといった問題が起こり得る。Ractor はこれらのリスクを軽減するよう設計されているが、プログラム全体のロジックが複雑になることで、バグのリスクは依然として存在する。

以上の背景を踏まえ、Ractor を用いたコードの記述を支援するための手法として、Rubocop を活用することを提案する。

Ractor の文法規則やベストプラクティスに基づいた静的解析ルールを Rubocop に実装することで、開発者が効率的かつ正確に並列処理を記述できるよう支援する。具体的には以下の3つを目指す。

- 文法エラーの早期検出  
Ractor に固有の文法規則（例：分離性の要件）を RuboCop のカスタムルールとして実装することで、開発者がコード記述時に即座に問題点を把握できるようにする。
- ベストプラクティスの普及  
Ractor の効率的な利用方法や推奨される設計パターンをルール化し、コードレビューを自動化することで、開発者が高品質なコードを書く習慣を身に付けられるよう支援する。
- 学習支援  
開発者が Ractor に関する知識を深められるよう、ルール違反の警告メッセージに適切

な修正方法を提供する。

## 5.2. 実装

### 5.2.1. プッシュ型通信

Ractor は、明示的なメッセージパッシング (send/receive) を用いてデータをやり取りする。この仕組みによりスレッドセーフな並行処理が可能となるが、開発時には以下のような問題が発生しやすい。

- send と receive の対応関係が不明確になる  
Ractor#send が呼び出された箇所と、それに対応する Ractor.receive が記述された箇所の整合性を手動で確認するのは困難である。
- エラーチェックが複雑化する  
対応関係が誤っている場合、プログラムの実行時にエラーや無限に待機してしまう状態が発生する可能性がある。

これらの問題を解決するため、Rubocop のカスタムルールを実装し、send と receive の対応関係を静的解析で検証する仕組みを構築した。

通常の Rubocop の機能では、AST ノード内の Ractor#send と Ractor.receive の対応を確認することは不可能であるが、本実装では RactorChecker クラスを別途作成することで、これを実現した。RactorChecker クラスの実装について、付属プログラム ractor\_checker.rb に示す。具体的には、Ractor 名をキーとしたマッピングを構築し、それに基づいて対応関係を検証する。また、対応関係が正しくない場合には警告を生成するだけでなく、必要に応じてコードを自動修正する機能も提供する。

実装の要点としては、まず RactorChecker クラスがコードファイルを解析して AST を生成し、Ractor.new ブロック内での send および receive の呼び出しを検出する。検出したノードは、それぞれ Ractor.receive または Ractor#send のリストに分類される。次に、これらのリストを比較することで対応関係を検証し、対応する Ractor#send または Ractor.receive が不足している場合には警告を生成する。

このカスタムルールは、以下の 2 つの Cop を中心に構成される。ひとつは Style::RactorSendReceive クラスであり、Ractor.receive が存在するにもかかわらず対応する Ractor#send が見つからない場合に警告を出す。Style::RactorSendReceive クラスの実装については、付属プログラム ractor\_send\_receive.rb に示す。もうひとつは Style::RactorReceiveSend クラスであり、Ractor#send が存在するにもかかわらず対応する Ractor.receive が見つからない場合に警告を出す。Style::RactorReceiveSend クラスの実装については、付属プログラム ractor\_receive\_send.rb に示す。これらの Cop はそれぞれ



RactorChecker を利用して解析を行い、静的解析の結果を基にコードの修正提案を行う。

例えば、Ractor.receive に対応する Ractor#send が不足している場合、Style::RactorSendReceive クラスは警告を生成し、オートコレクト機能によって不足している Ractor#send を自動的に挿入する。同様に、Ractor#send に対応する Ractor.receive が不足している場合には、Style::RactorReceiveSend クラスが警告を生成し、不足している Ractor.receive を自動的に挿入する。このオートコレクト機能により、開発者は手動で修正箇所を特定する必要がなくなり、コードの品質と開発効率が向上する。

このカスタムルールについて図 9 のサンプルコードを用いて、動作を示す。

```
r = Ractor.new do
  msg = Ractor.receive
  msg
end

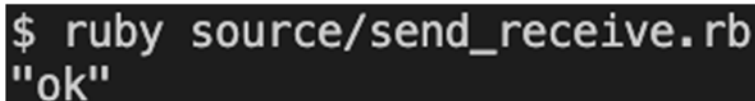
r.send 'ok'

p r.take
```

図 9 : Ractor Send/Receive サンプルコード

使用するサンプルコードの流れについて説明する。まず、新しい Ractor を生成する。この Ractor の中で、Receive 処理を使ってメッセージを受け取り、受け取ったメッセージをそのまま返す。次に、Send 処理を実行して、Ractor に文字列'ok'というメッセージを送信する。最後に、take メソッドを使用して Ractor から返された文字列'ok'を受け取り、標準出力に出力する。

実際にサンプルプログラムの実行結果を図 10 に示す。



```
$ ruby source/send_receive.rb
ok
```

図 10 : send\_receive.rb の実行結果

Send 処理で Ractor に送信した文字列'ok'が標準出力されていることが確認できる。この状態で CLI 上で rubocop ./source/send\_receive.rb を実行した結果を図 11 に示す。

```
Inspecting 1 file
.

1 file inspected, no offenses detected
```

図 11 : rubocop ./source/send\_receive.rb の出力

Receive 処理に対応する Send 処理が存在するため、Rubocop での警告は出力されない。  
次に、サンプルプログラム内の Send 処理(r.send 'ok')をコメントアウトして実行してみる。  
しかし、実行すると、強制終了しない限り、Ractor 内の Receive 処理が送信されるのを無限に待機している状態になる。Rubocop を実行すると、Receive 処理に対応する Send 処理が存在しないため、実装したカスタムルールの警告が出力される。この時の Rubocop 実行結果を図 12 に示す。

```
Inspecting 1 file
C

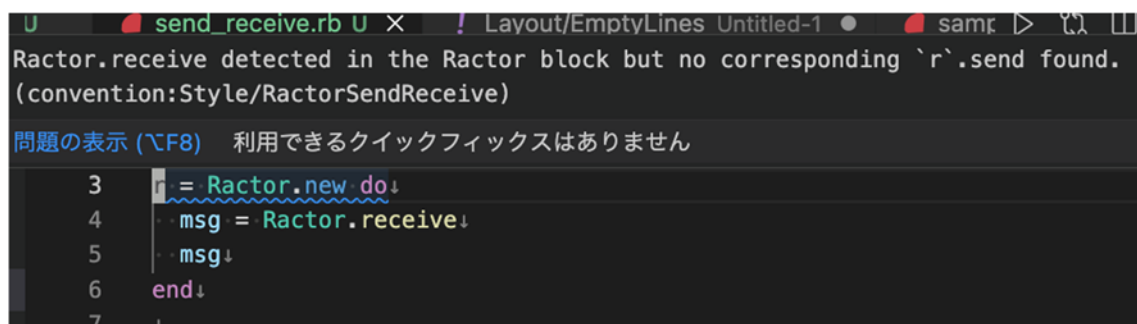
Offenses:

source/send_receive.rb:3:1: C: [Correctable] Style/RactorSendReceive: Ractor.receive
detected in the Ractor block but no corresponding r.send found.
r = Ractor.new do ...
~~~~~

1 file inspected, 1 offense detected, 1 offense autocorrectable
```

図 12 : Send 処理がない場合の rubocop 実行結果

また、VSCode であれば、作成したカスタムルールを読み込み、コード上に警告が出力され、Rubocop を手動で実行する手間を省くことが可能である。この様子を図 13 に示す。



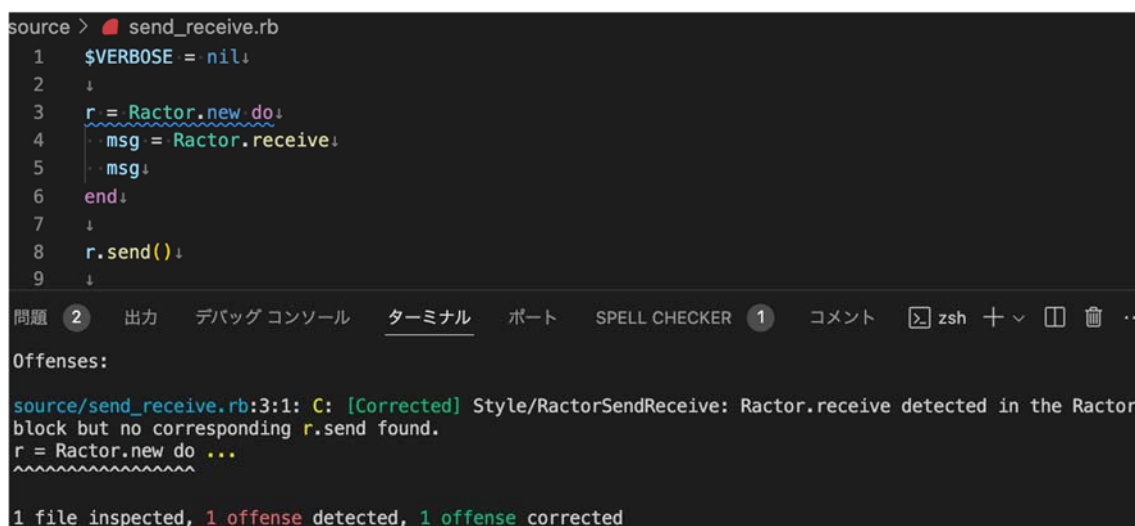
```
U send_receive.rb U X ! Layout/EmptyLines Untitled-1 ● samr > ⇧F8 |||
Ractor.receive detected in the Ractor block but no corresponding `r`.send found.
(convention:Style/RactorSendReceive)
問題の表示 (⇧F8) 利用できるクイックフィックスはありません

3 r = Ractor.new do
4   msg = Ractor.receive
5   msg
6 end
7 ↓
```

図 13 : コード上での Rubocop 警告

さらに、実装したカスタムルールを自動修正機能に対応させた。実際に自動修正機能を実行

してみると Ractor の処理の後に Send 処理が追加される。自動修正機能を使用する場合は rubocop コマンドを -A オプション付きで実行する。開発者は、自動で挿入された Send 処理に引数を記述するのみでよくなる。実際に、自動修正機能を実行した結果を図 14 に示す。8 行目に Send 処理が自動で追加され、図下部の出力では自動修正機能を実行し、修正されたことが出力されている。



```
source > send_receive.rb
1  $VERBOSE = nil
2  ↓
3  r = Ractor.new do
4    msg = Ractor.receive
5    msg
6  end
7  ↓
8  r.send()
9  ↓

問題 2  出力  デバッグ コンソール  ターミナル  ポート  SPELL CHECKER 1  コメント  zsh + ▢ ▢ ...

Offenses:

source/send_receive.rb:3:1: C: [Corrected] Style/RactorSendReceive: Ractor.receive detected in the Ractor
block but no corresponding r.send found.
r = Ractor.new do ...
~~~~~

1 file inspected, 1 offense detected, 1 offense corrected
```

図 14：自動修正機能の実行

### 5.2.2. オブジェクト共有

次のカスタムルールとして、Ractor ブロック外の変数を誤って参照している箇所を指摘するものである。Ractor を活用した並行処理では、Ractor ブロック外の変数を誤って参照しようとするとうエラーが発生する仕様になっている。この問題を防ぐために、Rubocop のカスタムルールを実装し、Ractor ブロック内での外部変数の参照を検出する仕組みを提供した。

本カスタムルールでは、Style::RactorSharingVariables クラスを中心に、Ractor ブロック内で使用される変数がブロック外のスコープに属していないかを検証する。Style::RactorSharingVariables クラスの実装については、付属プログラム ractor\_sharing\_variables.rb に示す。このルールは、AST を解析し、Ractor.new ブロック内での変数参照をチェックすることで動作する。

このルールの実装において重要な役割を果たすのが、RactorExternalReferencesChecker クラスである。RactorExternalReferencesChecker クラスの実装については、付属プログラム ractor\_external\_references\_checker.rb に示す。このクラスは、コード全体の AST を解析し、Ractor.new ブロック内で定義されていない変数が参照されているかどうかを検出する。具体的には、以下のような手順で解析を行う。

1. コードファイルを読み込み、AST を生成する。
2. AST ノードを深さ優先探索し、変数の定義箇所と参照箇所を特定する。
3. `Ractor.new` ブロック内で参照されている変数が、ブロック外で定義されている場合に警告を生成する。

`Style::RactorSharingVariables` クラスは、`Rubocop` の `Cop` を継承し、AST ノード内での変数参照を検出する。具体的には、`on_lvar` メソッドをオーバーライドし、ローカル変数参照ノードを検出した際に、その変数が `Ractor` ブロック外部のスコープに属しているかどうかを `RactorExternalReferencesChecker` を通じて検証する。検証の結果、外部変数が参照されている場合には警告を出力する。

通常の `Rubocop` の機能では `Ractor` ブロック内での外部変数参照を検出することは困難である。しかし、本実装では `RactorExternalReferencesChecker` を用いることで、AST の解析を通じてこれを実現している。また、検出した問題に対して自動修正機能は提供しない設計となっている。これは、外部変数の参照を解消するには通常、コード全体の設計変更が必要となるためである。

このカスタムルールについて図 15 のサンプルコードを用いて、動作を示す。

```
s = 'hello'
r = Ractor.new do
  s << 'world'
end

p r.take
```

図 15 : `RactorSharingVariables` 検証用サンプルコード

使用するサンプルコードについて解説する。このコードは、変数 `s` に文字列 `'hello'` を代入する。その後、`Ractor` を生成し、その中で変数 `s` に格納された `'hello'` に `'world'` を追加する。そして、`Ractor` の結果を `r.take` で取得し、標準出力に出力している。しかし、`Ractor` では安全な並列処理のため、可変なオブジェクトを複数の `Ractor` 間で同時に共有して操作しようとすることはできない仕様になっている。そのため、`Ractor` 内で変数 `s` を操作しようとした時点でエラーになってしまう。

実際に作成したカスタムルールを実行した結果を図 16 に示す。

```
source > sharing_variables_with_main.rb
1 s = 'hello'↓
2 r = Ractor.new do↓
3   s << 'world'↓
4 end↓
5 ↓
6 s << 'kai'↓
7 p r.take↓

問題 5 出力 デバッグ コンソール ターミナル ... zsh + - [ ] [ ] ... ^ x

Offenses:

source/sharing_variables_with_main.rb:3:3: C: Style/RactorSharingVariables: This
may be referencing variables outside of the ractor block, which will result in a
n error.
  s << 'world'
  ^

1 file inspected, 1 offense detected
```

図 16: Rubocop RactorSharingVariables 実行結果

本カスタムルールにより、Ractor を用いた並行処理コードの品質向上が期待できる。このルールを活用することで、開発者はエラーの原因となる外部変数参照を事前に防止できる。

### 5.2.3. プル型通信

次に pull 型の通信についてのルールを実装した。Ractor は Ractor.yield と Ractor#take を用いてデータをやり取りすることが可能である。しかし、Ractor.yield が記述されていても、それに対応する Ractor#take が記述されていない場合、プログラムの挙動が意図しないものになる可能性がある。このような不整合を防ぐため、Rubocop のカスタムルールを実装し、Ractor.yield と Ractor#take の対応関係を静的に検証する仕組みを構築した。

本カスタムルールは、Style::RactorYieldTake クラスを中心に、Ractor.yield が適切に Ractor#take と対応付けられているかを検証する。Style::RactorYieldTake クラスの実装については、付属プログラム ractor\_yield\_take.rb に示す。このルールは、AST を解析し、Ractor.yield の呼び出しが存在する場合に、その Ractor 内で対応する Ractor#take が存在するかを確認する。通常の Rubocop の機能では、このようなペアリングの検証は不可能であるが、本実装では RactorYieldTakeChecker クラスを利用することでこれを実現している。RactorYieldTakeChecker クラスの実装については、付属プログラム ractor\_yield\_take\_checker.rb に示す。

RactorYieldTakeChecker クラスは、コード全体の AST を解析し、Ractor.yield および Ractor#take の呼び出し箇所を特定する。解析の手順は以下の通りである。

1. コードファイルを読み込み、AST を生成する。
2. AST ノードを探索し、`Ractor.yield` と `Ractor#take` の呼び出しを検出する。
3. 検出したノードを `Ractor` 名をキーとしてマッピングし、それに基づいて対応関係を検証する。

`Style::RactorYieldTake` クラスは、`Rubocop` の `Cop` を継承し、`on_send` メソッドをオーバーライドすることで、`Ractor.yield` の呼び出しを検出する。この際、`Ractor.new` ブロック内における `Ractor.yield` の呼び出しを対象とし、対応する `Ractor#take` の有無を `RactorYieldTakeChecker` を通じて検証する。対応する `Ractor#take` がみつからない場合には警告を生成する。

このカスタムルールについて図 17 のサンプルコードを用いて、動作を示す。

```
r = Ractor.new do
  Ractor.yield 'ok'
end

p r.take
```

図 17: `RactorYieldTake` 検証用サンプルコード

使用するサンプルコードについて解説する。まず、`Ractor` を生成し、`Ractor` 内で `Ractor.yield` により `Ractor#take` で値を取得するまで待ち、文字列 `'ok'` を返す。その後、`Ractor#take` で受け取った文字列 `'ok'` を標準出力に出力する。つまり、`Ractor.yield` があり、それに対応する `Ractor#take` がない場合は `Ractor.yield` で返した値を取得する手段がなくなってしまう。

実際に作成したカスタムルールをサンプルコードで実行した結果を図 18 に示す。

```
source > yield_take.rb
1  r = Ractor.new do
2    Ractor.yield 'ok'
3  end
4
5  # p r.take

問題 14 出力 デバッグ コンソール ターミナル ... zsh + - [] ... ^ X

C

Offenses:

source/yield_take.rb:2:3: C: Style/RactorYieldTake: No take found that corresponds to yield
  Ractor.yield 'ok'
  ~~~~~

1 file inspected, 1 offense detected
```

図 18: Rubocop RactorYieldTake 実行結果

このルールを活用することで、開発者は `Ractor.yield` と `Ractor#take` の対応関係を明確にし、意図しない動作を防止できる。

#### 5.2.4. Ractor#take

`Ractor#take` に関してのルールをもう一つ実装した。`Ractor#take` を適切に使用しない場合、期待するデータの受け渡しが行われず、プログラムの動作に問題を引き起こす可能性がある。この課題を解決するために、`Ractor.take` の適切な使用を検証するカスタムルールを実装した。

本カスタムルールは、`Style::RactorTake` クラスを中心に、`Ractor.new` でインスタンスが生成されている際に、`Ractor#take` が使用されているかを検証する。`Style::RactorTake` クラスの実装については、付属プログラム `ractor_take.rb` に示す。このルールは、AST を解析して `Ractor.new` の呼び出しを検出し、対応する `Ractor#take` が存在するかを確認する。これにより、`Ractor` 間でのデータやり取りが正しく行われるように保証する。

`RactorTake` クラスの動作は以下の通りである。

1. AST ノード内で `Ractor.new` ブロックを検出する。
2. `Ractor.new` と対応付いた `Ractor#take` 呼び出しを解析する。
3. 対応する `Ractor#take` が存在しない場合には警告を出力し、問題を開発者に通知する。

このルールの実装には `RactorTakeChecker` クラスが対応する `Ractor#take` を検出する役割を果たしている。`RactorTakeChecker` クラスの実装については、付属プログラム

ractor\_take\_checker.rb に示す。このクラスは、コードファイル全体の AST を解析し、Ractor#take の呼び出し箇所を特定する。また、Ractor インスタンスに対応する Ractor#take が存在するかを判定する。

このカスタムルールについて図 19 のサンプルコードを用いて、動作を示す。

```
r = Ractor.new do
  sleep 10
  p 'a'
end

r.take
```

図 19 : Rubocop RactorTake 検証用サンプルコード

使用するサンプルコードについて解説する。まず、新しい Ractor を生成する。この Ractor の中で、10 秒プログラムの実行を停止する。その後、文字 'a' を標準出力に出力する。最後に、r.take で Ractor の結果を受け取る。この r.take がいない場合、Ractor 内の処理が完了するのを待たずに終了してしまう。それでは意図した動作にはならないため、take メソッドを使用して Ractor の結果を受け取るが必要とされる。

実際に r.take をコメントアウトし、作成したカスタムルールを実行した結果を図 20 に示す。rubocop ./source/take.rb を実行すると take.rb ファイル内の Ractor.new により Ractor が生成されている場合、対応する Ractor#take が存在しなければ、Rubocop による警告が出力される。



```
source > take.rb
1  r = Ractor.new do
2    sleep 10
3    p 'a'
4  end
5
6  # r.take
7  [EOF]
```

問題 23 出力 デバッグ コンソール ターミナル ... zsh

Scanning /Users/yanagisawakai/college\_research/source/take.rb

C

Offenses:

source/take.rb:1:1: C: Style/RactorTake: No r.take found that corresponds to yield  
r = Ractor.new do ...  
~~~~~

1 file inspected, 1 offense detected

図 20: Rubocop RactorTake 実行結果

このように、Ractor インスタンスに対応する Ractor#take を検出することで前述したサンプルコードのような、意図しない挙動を実行せずに検出することができる。

## 第6章 結論

本研究では、Ruby の並列処理機構である Ractor の性能評価を通じて、その利点と課題を明らかにし、静的解析ツール Rubocop を用いたコード支援の可能性について検討した。Ractor は、並列処理における安全性と効率性を両立するために設計された仕組みであり、GVL の制約を超えて、異なる Ractor 間で並列実行が可能であるという特長を持つ。しかし、その利用には特有の制約があり、開発者にとっては新しいプログラミングモデルの理解と適応が求められる。これにより、Ractor を活用する際の学習コストや記述時のミスが課題として浮かび上がる。

本研究で注力したのは、Rubocop を活用してこれらの課題を解決するための方法を模索することであった。Rubocop はコードの静的解析を行い、スタイルガイドに基づいた記述の一貫性を保つための強力なツールであり、本研究ではこれに Ractor 専用のカスタムルールを実装することで、並列処理コードの記述を支援した。その結果、いくつかの効果が確認された。まず、カスタムルールを用いることで、プログラム実行前に潜在的なエラーを検出することが可能となった。これにより、従来は実行中に発覚していた問題を、記述段階で修正することができ、デバッグに要する時間を削減できた。また、Ractor の特有の制約を静的解析によって明示することで、開発者が Ractor を効率的に活用しやすくなり、結果としてコードの可読性や保守性が向上した。

加えて、本研究で実装したカスタムルールは、Ractor の利用に不慣れな開発者にも有用であることが示された。たとえば、Ractor における Send/Receive の対応関係を検証するルールでは、対応するメソッドが不足している場合に警告を表示するだけでなく、必要に応じて自動修正を提供する機能も備えている。このような機能により、開発者は Ractor 特有の仕様を自然に学びながらコードを記述できる環境が整った。また、これらのルールはコードレビューの負担を軽減するだけでなく、ベストプラクティスを普及させる役割も果たす。

一方で、本研究ではいくつかの課題も明らかとなった。第一に、Ractor に関連するさらなるカスタムルールの追加が求められる。現時点では、Ractor の利用に関する基礎的なサポートは実現したが、実用的なシナリオにおいてはより詳細な解析が求められる場合が多い。また、カスタムルールの実行速度に関する問題も課題として浮上した。本研究で実装した解析機能では、Send/Receive の対応関係を検証する際に対象ファイルを 2 回解析する仕組みを採用したが、このプロセスがコード量の増加に比例して性能の低下を引き起こす可能性がある。この問題に対しては、解析アルゴリズムの改善が必要であり、効率的なデータ構造や手法の採用が今後の課題となる。

さらに、Ractor を用いた実験では、Ruby のガベージコレクションが性能に与える影響が一部確認された。Ractor 間でのデータのスワップ処理が頻発する場合、可変長配列を用いることによるメモリの再配置が発生し、これが性能の低下につながることを示唆された。さ

らに、排他制御の影響で並列に実行されない可能性もあることが考察された。

本研究の成果は、Ruby コミュニティにおいて Ractor の普及を促進し、並列処理に関する課題解決の一助となるものである。特に、Rubocop を用いた静的解析ルールの拡張は、Ractor を初めて利用する開発者にとってもわかりやすく、効率的な支援を提供することができる。今後の展望としては、さらなるルールの拡充や、Rubocop における解析機能の最適化を図り、より実用的な解析ツールを開発することが挙げられる。また、本研究で得られた知見をもとに、Rubocop を用いて、Ractor 以外の Ruby プログラムの支援を提供することが期待される。

## 参考文献

- [1] “Ruby とは”. Ruby A PROGRAMMER’S BEST FRIEND.  
<https://www.ruby-lang.org/ja/about/>, (参照 2025-1-21)
- [2] 笹田耕一. Ruby 向け並列化機構 Guild の試作. 情報処理学会プログラミング研究会. 2018
- [3] “「アクターモデル」による並列処理プログラミング入門”. SIOS Tech Lab. <https://tech-lab.sios.jp/archives/8738>, (参照 2025-1-22)
- [4] “Ractor – Ruby’s Actor-like concurrent abstraction”. docs.ruby-lang.org.  
[https://docs.ruby-lang.org/en/master/ractor\\_md.html](https://docs.ruby-lang.org/en/master/ractor_md.html), (参照 2025-1-24)
- [5] 国分崇志. “Ruby 3.3 YJIT のメモリ管理と RJIT”. gikyo.jp. 2024-1-22.  
<https://gihyo.jp/article/2024/01/ruby3.3-jit>, (参照 2025-1-24)
- [6] “ライブラリ”. Ruby A PROGRAMMER’S BEST FRIEND.  
<https://www.ruby-lang.org/ja/libraries/>, (参照 2025-1-24)
- [7] “Development”. docs.rubocop.org.  
<https://docs.rubocop.org/rubocop/development.html>, (参照 2025-1-24)
- [8] “Node Types”. docs.rubocop.org. [https://docs.rubocop.org/rubocop-ast/node\\_types.html](https://docs.rubocop.org/rubocop-ast/node_types.html), (参照 2025-1-25)

## 謝辞

本研究を進めるにあたり、熱心なご指導を賜りました甲斐宗徳教授をはじめ、協力していただいた皆様に感謝いたします。

## 付録

### 1. 付属プログラム bubble\_sort\_ractor.rb

```
require 'benchmark'

def bubble_sort(array)
  size = array.size

  (size - 1).times do
    (size - 1).times do |j|
      array[j], array[j + 1] = array[j + 1], array[j] if array[j] > array[j + 1]
    end
  end
end

def bubble_sort_parallel(array)
  # divide using pivot(MAX/2)
  from_first = 0
  from_last = NUM - 1

  while from_first < from_last
    from_first += 1 while array[from_first] < MAX / 2
    from_last -= 1 while array[from_last] >= MAX / 2

    next unless from_first < from_last

    array_from_first = array[from_first]
    array[from_first] = array[from_last]
    array[from_last] = array_from_first
  end

  branch_point = from_first

  # parallel bubble sort
  Ractor.make_shareable(array)

  r1 = Ractor.new do
    smaller_part = Ractor.receive
    bubble_sort(smaller_part)
```

```

    smaller_part
  end

  r2 = Ractor.new do
    larger_part = Ractor.receive
    bubble_sort(larger_part)
    larger_part
  end

  r1.send(array[...branch_point])
  r2.send(array[branch_point..])

  smaller_part = r1.take
  larger_part = r2.take

  smaller_part.concat(larger_part)
end

def print_array(array, max_display = 10)
  n = array.size
  n.times do |i|
    if i < max_display
      print "#{array[i].to_s.rjust(5)}#{(i + 1) % 15 == 0 ? "\n" : ' '}"
    elsif i == max_display
      puts "\n      *****"
    end

    if i >= n - max_display
      print "#{array[i].to_s.rjust(5)}#{(i + 1) % 15 == 0 ? "\n" : ' '}"
    end
  end
  puts "\n"
end

raise 'Usage: ruby script.rb MAX NUM' if ARGV.size != 2

MAX = ARGV[0].to_i
NUM = ARGV[1].to_i

data = Array.new(NUM) { rand(MAX) }

puts '----- Before sort -----'
print_array(data)

```

```

time = Benchmark.realtime do
  data = bubble_sort_parallel(data)
end

puts "¥n----- After Sort --#{time}sec---"
print_array(data)

```

## 2. 付属プログラム ractor\_checker.rb

```

require 'parser/current'

class RactorChecker
  attr_reader :ractor_receives, :ractor_sends

  def initialize(file_path)
    @file_path = file_path
    @ractor_receives = []
    @ractor_sends = []
  end

  def check
    buffer = Parser::Source::Buffer.new(@file_path)
    buffer.source = File.read(@file_path)

    parser = Parser::CurrentRuby.new
    ast = parser.parse(buffer)

    analyze_ast(ast)
  end

  def receive_paired_with_send?(node)
    exist_receive = false

    @ractor_receives.each do |receive|
      next unless node.children[0] == receive[:ractor]

      exist_receive = true
      @ractor_sends.each do |send|
        return true if receive[:ractor] == send[:ractor]
      end
    end
  end
end

```



```

    return false if exist_receive

    true
  end

  def send_paired_with_receive?(node)
    @ractor_sends.each do |send|
      next unless node.children[0].children[0] == send[:ractor]

      @ractor_receives.each do |receive|
        return true if send[:ractor] == receive[:ractor]
      end
    end
  end

  false
end

private

def analyze_ast(node, current_ractor = nil)
  return unless node.is_a?(Parser::AST::Node)

  case node.type
  when :lvasgn
    return unless node.to_json.include?(' (const nil :Ractor) :receive') && node.to_json.include?(' (const
nil :Ractor) :new')

    current_ractor = node.children[0]
    @ractor_receives << { ractor: current_ractor, node: node }
    when :send
      return unless node.children[1] == :send

      @ractor_sends << { ractor: node.to_json.scan(/¥(lvar :¥w+)¥) :send/)[0]&.first&.to_sym, node:
node }
    else
      node.children.each { |child| analyze_ast(child, current_ractor) }
    end
  end
end

```

### 3. 付属プログラム ractor\_send\_receive.rb

```
require 'rubocop'
```

```

require_relative './ractor_checker'

module RuboCop
  module Cop
    module Style
      class RactorSendReceive < Base
        extend AutoCorrector

        MSG = 'Ractor.receive detected in the Ractor block `¥`
              `but no corresponding `%<ractor>s`.send found.'.freeze

        def_node_search :ractor_new?, <<~PATTERN
          (lvasgn $_ractor_name
            (block
              (send (const nil? :Ractor) :new)
              ...
            )
          )
        PATTERN

        def on_lvasgn(node)
          return unless ractor_new?(node)

          file_path = processed_source.file_path
          checker = RactorChecker.new(file_path)
          checker.check

          return if checker.receive_paired_with_send?(node)

          message = message(node)
          add_offense(node, message: message) do |corrector|
            corrector.insert_after(node, "¥n¥n#{node.children[0]}.send()")
          end
        end

        private

        def message(node)
          format(MSG, ractor: node.children[0])
        end
      end
    end
  end
end

```

```
end
```

#### 4. 付属プログラム ractor\_receive\_send.rb

```
require 'rubocop'
require_relative './ractor_checker'

module RuboCop
  module Cop
    module Style
      class RactorReceiveSend < Base
        extend AutoCorrector

        MSG = 'Found `%<ractor>s`.send ' ¥
              'but no corresponding Ractor.receive in ractor block found.'.freeze

        def_node_search :ractor_new_block?, <<~PATTERN
          (lvasgn $_ractor_name
            (block
              (send (const nil? :Ractor) :new)
              ...
            )
          )
        PATTERN

        def_node_search :ractor_send?, <<~PATTERN
          (send (lvar _) :send ...)
        PATTERN

        def on_send(node)
          return unless ractor_send?(node)

          file_path = processed_source.file_path
          checker = RactorChecker.new(file_path)
          checker.check

          return if checker.send_paired_with_receive?(node)

          message = message(node)
          add_offense(node, message: message) do |corrector|
            corrector.insert_before(find_ractor_new_block(node).children[1].children[2],
              "Ractor.receive¥n")
          end
        end
      end
    end
  end
end
```

```

end

private

def find_ractor_new_block(node)
  node.each_ancestor.find { |ancestor| ractor_new_block?(ancestor) }
    .each_child_node.find { |ancestor_node| ractor_new_block?(ancestor_node) }
end

def message(node)
  format(MSG, ractor: node.children[0].children[0])
end
end
end
end
end
end

```

## 5. 付属プログラム ractor\_sharing\_variables.rb

```

require 'rubocop'
require_relative './ractor_external_references_checker'

module RuboCop
  module Cop
    module Style
      class RactorSharingVariables < Base
        MSG = 'This may be referencing variables outside of the ractor block,' ¥
          'which will result in an error.'.freeze

        def_node_matcher :ractor_new?, <<~PATTERN
          (block
            (send (const nil? :Ractor) :new)
            ...
          )
        PATTERN

        def on_lvar(node)
          return unless ractor_new?(node.ancestors[1])

          file_path = processed_source.file_path
          checker = RactorExternalReferencesChecker.new(file_path)
          checker.check
        end
      end
    end
  end
end

```

```

        return unless checker.reference_external_variables?(node.children[0])

        add_offense(node, message: MSG)
      end
    end
  end
end
end

```

## 6. 付属プログラム ractor\_external\_references\_checker.rb

```

require 'parser/current'

class RactorExternalReferencesChecker
  attr_reader :variables_in_ractor, :ractor_external_variables

  def initialize(file_path)
    @file_path = file_path
    @variables_in_ractor = []
    @ractor_external_variables = []
  end

  def check
    buffer = Parser::Source::Buffer.new(@file_path)
    buffer.source = File.read(@file_path)

    parser = Parser::CurrentRuby.new
    ast = parser.parse(buffer)

    analyze_ast(ast)
  end

  def reference_external_variables?(variable_name)
    @ractor_external_variables.any? { |var| var[:name] == variable_name }
  end

  private

  def analyze_ast(node)
    return unless node.is_a?(Parser::AST::Node)

    case node.type
    when :lvasgn

```

```

    if node.to_json.include?(' (const nil :Ractor) :new')
      find_lvar(node)
    else
      @ractor_external_variables << { name: node.children[0] } unless @ractor_external_variables.any?
      { |var| var[:name] == node.children[0] }
    end
  end
else
  node.children.each { |child| analyze_ast(child) }
end
end

def find_lvar(node)
  return unless node.is_a?(Parser::AST::Node)

  case node.type
  when :lvar
    @variables_in_ractor << { name: node.children[0] } unless @variables_in_ractor.any? { |var|
var[:name] == node.children[0] }
  else
    node.children.each { |child| find_lvar(child) }
  end
end
end
end

```

## 7. 付属プログラム ractor\_yield\_take.rb

```

require 'rubocop'
require_relative 'ractor_yield_take_checker'

module RuboCop
  module Cop
    module Style
      class RactorYieldTake < Base
        MSG = 'No take found that corresponds to yield'.freeze

        def_node_search :ractor_yield?, <<~PATTERN
          (send (const nil? :Ractor) :yield ...)
        PATTERN

        def_node_search :ractor_new_block?, <<~PATTERN
          (lvasgn $_ractor_name
            (block
              (send (const nil? :Ractor) :new)

```

```

        ...
    )
)
PATTERN

def on_send(node)
  return unless ractor_yield?(node)

  file_path = processed_source.file_path
  checker = RactorYieldTakeChecker.new(file_path)
  checker.check

  ractor_name = node.each_ancestor.find { |ancestor| ractor_new_block?(ancestor) }.children[0]

  return if checker.yield_paired_with_take?(ractor_name)

  message = message(node)
  add_offense(node, message: message)
end

private

def message(node)
  format(MSG, ractor: node.children[0].children[0])
end
end
end
end
end
end
end

```

## 8. 付属プログラム ractor\_yield\_take\_checker.rb

```

require 'parser/current'

class RactorYieldTakeChecker
  attr_reader :ractor_yields, :ractor_takes

  def initialize(file_path)
    @file_path = file_path
    @ractor_yields = []
    @ractor_takes = []
  end
end

```

```

def check
  buffer = Parser::Source::Buffer.new(@file_path)
  buffer.source = File.read(@file_path)

  parser = Parser::CurrentRuby.new
  ast = parser.parse(buffer)

  analyze_ast(ast)
end

def yield_paired_with_take?(ractor_name)
  @ractor_takes.each { |ractor_take| return true if ractor_take[:ractor] == ractor_name }

  false
end

private

def analyze_ast(node, current_ractor = nil)
  return unless node.is_a?(Parser::AST::Node)

  case node.type
  when :lvasgn
    return unless node.to_json.include?('(:const nil :Ractor) :yield')

    current_ractor = node.children[0]
    @ractor_yields << { ractor: current_ractor, node: node }
  when :send
    return unless node.to_json.scan(/¥(lvar :¥w+)¥) :take¥/).any?

    @ractor_takes << { ractor: node.to_json.scan(/¥(lvar :¥w+)¥) :take¥/)[0]&.first&.to_sym, node:
node }
  else
    node.children.each { |child| analyze_ast(child, current_ractor) }
  end
end
end

```

## 9. 付属プログラム ractor\_take.rb

```

require 'rubocop'
require_relative 'ractor_take_checker'

```



```

module RuboCop
  module Cop
    module Style
      class RactorTake < Base
        MSG = 'No `%<ractor>s`.take found that corresponds to yield'.freeze

        def_node_search :ractor_new_block?, <<~PATTERN
          (lvasgn $_ractor_name
            (block
              (send (const nil? :Ractor) :new)
              ...
            )
          )
        PATTERN

        def on_lvasgn(node)
          return unless ractor_new_block?(node)

          file_path = processed_source.file_path
          checker = RactorYieldTakeChecker.new(file_path)
          checker.check

          return if checker.paired_with_take?(node.children[0])

          message = message(node)
          add_offense(node, message: message)
        end

        private

        def message(node)
          format(MSG, ractor: node.children[0])
        end
      end
    end
  end
end

```

## 10. 付属プログラム ractor\_take\_checker.rb

```

require 'parser/current'

class RactorYieldTakeChecker

```

```

attr_reader :ractor_takes

def initialize(file_path)
  @file_path = file_path
  @ractor_takes = []
end

def check
  buffer = Parser::Source::Buffer.new(@file_path)
  buffer.source = File.read(@file_path)

  parser = Parser::CurrentRuby.new
  ast = parser.parse(buffer)

  analyze_ast(ast)
end

def paired_with_take?(ractor_name)
  return false if @ractor_takes.empty?

  @ractor_takes.each { |ractor_take| return true if ractor_take[:ractor] == ractor_name }

  false
end

private

def analyze_ast(node, current_ractor = nil)
  return unless node.is_a?(Parser::AST::Node)

  case node.type
  when :send
    return unless node.to_json.scan(/%(lvar :(\w+))% :take%)/.any?

    @ractor_takes << { ractor: node.to_json.scan(/%(lvar :(\w+))% :take%)/[0]&.first&.to_sym, node:
node }
  else
    node.children.each { |child| analyze_ast(child, current_ractor) }
  end
end
end

```