令和 5 年度

修士学位論文

Rust を用いて OTP アプリケーションを構築するためのフレームワークの開発

Development of a Framework for Implementing OTP Applications with Rust

1220310　　岡本 怜士

指導教員　　高田 喜朗

2024 年 2 月 28 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

要 旨

Rust を用いて OTP アプリケーションを構築するためのフレームワークの開発

岡本 怜士

プログラミング言語 Rust ではグリーンスレッドを用いた並行プログラミングのサポートが行われている．しかし，エラーハンドリングに失敗すると他のタスクにも悪影響を及ぼす可能性がある点やタスクをどのスケジューラが実行するかを決める必要がある点などの理由から，グリーンスレッドによるプログラミングはコストが高い．そこで，本研究ではアクターモデルによる並行プログラミングをサポートする Erlang/OTP を用いて耐障害性の高い，かつ大量のプロセスが並行動作するアプリケーションを容易に開発できるフレームワークを提案する．

Rust のグリーンスレッドを用いてアクターモデルの並行プログラミングを可能にする Actix と比較して，4 つのベンチマークプログラムを用いた性能評価実験を行った結果，1 つのベンチマークで大幅に性能が低下したが，2 つのベンチマークプログラムでは Actix と大きな差はなく，OS スレッドを用いて実装した場合と比べて大幅に優れた性能を記録した．また，残り 1 つのベンチマークでは，Actix よりも優れた性能を記録した．さらに，提案フレームワークを用いると OTP が提供する多機能なスーパバイザプロセスを実装できるため耐障害性の高いシステムを低コストで開発でき，タスクのスケジューリングも Erlang VM が行うため，その分コストが低下することが期待される．

キーワード　Erlang/OTP, Rust, 並列処理, 耐障害性

Abstract

Development of a Framework for Implementing OTP Applications with Rust

Okamoto Reiji

In the programming language Rust, support for concurrent programming using green threads is provided. However, failure in error handling can negatively affect other tasks, and a user need to decide which scheduler will execute the task, and for those reasons programming with green threads is costly. Therefore, in this study, we propose a framework that can easily develop applications with high fault tolerance and a large number of processes running concurrently using Erlang/OTP, which supports concurrent programming with the actor model.

Compared to Actix, which enables concurrent programming of the actor model using Rust's green threads, we conducted a performance evaluation experiment using four benchmark programs. Although the performance significantly decreased in one benchmark, there was not much difference in two benchmark programs compared to Actix, and it recorded significantly superior performance compared to when implemented using OS threads. In addition, in one remaining benchmark, it recorded better performance than Actix. Furthermore, since the proposed framework can implement the multifunctional supervisor process provided by OTP, a highly fault-tolerant system can be developed at a low cost. The Erlang VM also performs task scheduling, which is expected to lower the cost of the system.

key words Erlang/OTP, Rust, Parallel Processing, Fault Tolerance

目次

図目次

第 1 章

初めに

プログラミング言語 Rust[1] は，ソフトウェアの信頼性と実行効率の高さの両立を目指すプログラミング言語である．Bugden らの調査 [2] によると，多くの研究論文によって Rust が評価対象となっており，主に次の点で高い評価を得ていると報告されている．

- 高度な型システムによって静的にメモリ安全性保証する高水準言語と，データレイアウトやメモリ管理に対する細かい制御を提供する低水準な言語とのギャップを埋めることに成功している点.
- Java や Go など主要な言語を上回るパフォーマンスを持ち，C/C++に対抗できる点.
- 並行環境におけるデータ競合をコンパイル時に検出できる点.

本研究では，Rust の並行プログラミングに焦点を当てる．Rust における並行プログラミングの実現は，主に OS スレッドを用いる方法と，グリーンスレッドを用いた非同期プログラミングの 2 通りがサポートされている．ただし，グリーンスレッドを用いる方法は今のところその言語機能がサポートされているだけで，グリーンスレッドの実装に対する標準のサポートはない．サードパーティ性のランタイムシステムが複数開発されているが，現在は Tokio[3] がデファクトスタンダードとなっており，Tokio 上で動作するアクターフレームワークの Actix[4] も有名である．

OS スレッドを用いる方法では，大量の I/O タスクを処理しなければならない場合に CPU，およびメモリオーバヘッドが大きくなりすぎるため，グリーンスレッドを用いることでそのオーバヘッドを小さく抑えることが可能になる．

しかし，グリーンスレッドによる非同期プログラミングは逐次コードとは異なる構文を用いる必要があり，非同期関数の定義，その呼び出し方，main 関数の定義などほぼ全ての要素が変更される．また，Tokio が提供する API は低水準であり，その複雑な機能を組み合わせてアプリケーションを構築するのはコストが高い．近年は Tokio の成熟や，Actix のようなグリーンスレッドの機能の抽象化をはかる非標準フレームワークが開発されるなど，この問題は大幅に緩和されている．しかし，各タスクをどのスケジューラが管理するかをプログラマが決定する必要があり，あるタスクで発生したパニックがスケジューラまで伝播すると他のタスクを道連れにしてしまうといった，多くの課題が残っている．

つまり，Rust の並行プログラミングは性能と実装の容易さについてトレードオフがあると言える．本研究ではこの課題に対して,「アプリケーションの設計や実装も容易にでき，耐障害性を満たし，大量のプロセスを並行動作させられる」ことを目指す．

我々は，この問題に対処するために並行指向の関数型プログラミング言語，およびその標準ライブラリやミドルウェアのセットである Erlang/OTP に注目した．Erlang/OTP は OTP の設計原則を提唱しており，耐障害性を満たし，かつ大量のプロセスが並行動作するアプリケーションを容易に構築できるようになっている．また，Erlang から Rust などで実装された機械語関数を呼び出すための機構が標準でサポートされている．これらの機能や考え方を応用できれば Rust を用いて耐障害性を満たし，かつ大量のプロセスが並行動作するアプリケーションを容易に構築できる．

本論文で示す主な貢献は次の 3 点である．

- Erlang/OTP が提供する機能を応用し，Rust だけで OTP アプリケーションを構築できるようにするためのフレームワークを開発した．このフレームワークにはアプリケーションのビルドツールや Rust コンパイラがある程度エラーを検知できるようにするためのライブラリが含まれる．
- 提案フレームワーク，Actix，OS スレッドを使用する場合の性能を比較するためにベンチマークプログラムを作成し，性能評価実験を行った．

- 提案フレームワークを用いた場合の構築容易性や耐障害性を，Actix と比較した場合について検討した．

本論文の構成は以下のようになっている．

まず，第 2 章では提案フレームワークのデザインを述べるにあたって必要となった背景技術をいくつか解説する．続いて第 3 章では，第 2 章で解説した内容を踏まえて研究目的を詳細に述べる．第 4 章では提案フレームワークのデザインについて述べ，第 5 章ではベンチマークプログラムを用いた性能評価実験について述べる．第 6 章では耐障害性と構築容易性に焦点を当てて，提案フレームワークについて検討する．第 7 章では，本研究の内容をまとめ，今後の課題について述べる．

第 2 章

背景技術

研究の動機や目的，および提案手法を述べる前に，背景技術について解説する．主に Rust と Erlang/OTP の言語機能や，グリーンスレッドのスケジューリング戦略，耐障害性の考え方などに焦点を当てる．

2.1 Rust について

現在は大小問わず様々な企業が，多様なプロダクトで Rust を採用している事例が増えた．以下はその一例である．

Firefox [*1] Web ブラウザ

Dropbox [*2] オンラインストレージサービス

Firecracker [*3] micro VM (軽量仮想マシン，AWS Lambda などの基盤技術)

Discord [*4] オンラインチャット・通話システム

本節では，本研究の提案手法を解説するために必要な Rust の基礎について述べる．

まずは Rust が標準でサポートしているビルドツール Cargo について，その後はトレイトやマクロといった高度な言語機能について，最後に Rust による並行プログラミングについて簡単に解説する．

[*1] https://www.mozilla.org/ja/firefox/

[*2] https://www.dropbox.com/

[*3] https://firecracker-microvm.github.io/

[*4] https://discord.com/

2.1.1 Cargo ビルドツール

Rust は Cargo というビルドツールを標準でサポートしている．複数のソースファイルからなるアプリケーションをビルド・実行したり，ソースファイルに埋め込まれたユニットテストを実行したりと，様々な機能が組み込まれている．

Cargo のコンパイル単位はクレート（crate）と呼ばれており，実行可能形式かライブラリ形式のどちらかである．ある機能群を提供する 1 つ以上のクレートのことをパッケージと呼ぶ．

コマンドラインインタフェースは `cargo` コマンドの第 1 引数にサブコマンドを指定し，第 2 引数以降にオプションを指定するように設計されている．例えば，標準で組み込まれている `build` サブコマンドを用いてパッケージをリリースビルドする際はシェル上で次のように実行する．ただし，シェルのプロンプトを記号 `$` で表している．

```
$ cargo build --release
```

プログラム 2.1　cargo build コマンドの実行例

Cargo の特徴の一つとして，このサブコマンドを第三者が拡張しやすいように作られている点が挙げられる．`xxx` という名前の拡張サブコマンドを使用したい場合は，`cargo-xxx` という名前のパッケージを自身の環境にインストールすればよい．

2.1.2 トレイト

Rust におけるトレイト（trait）とは，ある型に対して，特定のメソッドを持っていることを保証するためのものである．OCaml や Haskell における型クラスや，Java におけるインタフェースに似る．

頻繁に利用するトレイトは，デバッグプリントに用いる `Debug` トレイトや，値を複製できることを示す `Clone` トレイトである．詳細を除けば， `Debug` トレイトは Rust の標準ライブラリにおいて次のように定義されている．

```
trait Debug {
```

```
    fn fmt(&self, f: &mut Formatter<'_>) -> Result;
}
```

プログラム 2.2 Debug トレイトの定義

この Debug トレイトの定義によると， Debug トレイトを実装する型は fmt メソッドを持つとわかる．ここでは fmt メソッドの型シグネチャが定義されているだけだが，ここにデフォルト実装を含めることも可能である．デフォルト実装が含まれないメソッドについては，ある型にトレイトの実装を宣言する際に，その具体的な実装を定義する必要がある．以下の例は，ユーザ定義の型 Shape に Debug トレイトを実装する例である (具体的な実装については省略してある)．

```
impl Debug for Shape {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result {
        // ...
    }
}
```

プログラム 2.3 Shape 構造体に対する Debug トレイトの実装

これにより， println! マクロを使用して Shape 型の値をデバッグプリントできるようになる． Clone トレイトも同様にして Shape 型に実装すると，明示的に値を複製する clone メソッドを使用できるようになる．

なお，外部クレートで定義されたトレイトを外部クレートで定義された構造体に実装することは認められていない．これは孤児ルール（orphan rule）と呼ばれており，これによってあるトレイトの実装が各型につき高々一つしかないことが保証される．

2.1.3 マクロ

Rust におけるマクロはいくつか種類があるが，ここでは関数風のマクロのみを取り扱う．

マクロは macro_rules! というマクロを用いて宣言できる．通常の Rust 関数とは異なり，マクロは可変個の引数を取ることができる．引数についてパターンマッチができるため，再

帰的にマクロ呼び出しをするなどして可変個の引数を処理できる．

Rust の書籍 [5, 6] でも紹介されているマクロの例として，ベクタをリテラル風に定義できる vec! マクロの簡易版を紹介する．

```
macro_rules! vec {
    ( $( $x:expr ), *) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    }
}
```

プログラム 2.4 vec!マクロの定義（簡潔版）

マクロの中身は Rust の match 式に類似している．ここでは一つだけパターンが定義されていて，$(...) という表記はカッコ内でパターンにマッチする値をキャプチャする．このカッコの中には $x:expr という表記があり，これは任意の Rust 式にマッチしてその式に $x という名前をつける．その後ろにあるカンマは，実際にマクロの引数に渡される引数がカンマ区切りで与えられることを示すものであり，アスタリスク * は一つ前に記述されたパターン 0 個以上にマッチすることを意味する．例えば vec![1, 2, 3] というコードを書くと，Rust コンパイラは次のようなコードに展開する．

```
{
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
}
```

プログラム 2.5 プログラム 2.4 を用いて vec![1, 2, 3]を展開した結果

{} で囲まれた範囲はブロック式を表しており，ブロック式の値は最後に評価された式の値である．空のベクタを用意し，そのベクタに対してマクロの引数の先頭から順に `push` していくようなコードとなっていることがわかる．

2.1.4 並行プログラミング：OS スレッド

Rust における並行性実現の手段として，OS スレッドを直接用いる方法がある．OS スレッドを直接扱うための API は `std::thread` モジュールで提供されており，標準ライブラリに同梱されている．単に OS スレッドを 1 つ生成するだけならば， `std::thread::spawn` 関数を用いればよい．引数としてクロージャを受け取り，そのクロージャを実行する OS スレッドを新しく生成する．返り値は `JoinHandle` という構造体で，生成された OS スレッドが終了するのを待機するために用いるものである．

また，二つの OS スレッドが通信するための機能の一つとしてチャネルをサポートしている．こちらも標準ライブラリにて提供されており，`std::sync::mpsc::channel` 関数を使用して作成できるようになっている．ただし，このチャネルは一方通行なものであるため，二つのスレッドで双方向に通信したい場合は，明示的に二つのチャネルを作成する必要がある．

OS スレッドは，例えば並列サーバを実装する際に用いられる手法の一つだが，パフォーマンスに関して問題がある．スレッドの生成やコンテキストスイッチにかかるコストが高かったり，アイドル状態のスレッドでさえシステムリソースを消費してしまったりといった要因で CPU やメモリのオーバヘッドが大きい．したがって，大量のタスクを一度に処理しなければならないような場合に OS スレッドを用いると，リソースを大量に消費してしまう．その対策の一つとして，スレッドを補完するためのタスクの分割手法としてグリーンスレッドを用いて実行する非同期プログラミングという手法をサポートしている．

2.1.5 並行プログラミング：グリーンスレッド

Rust でグリーンスレッドを用いた非同期プログラミングを行う際は，`async/await` 構文を用いて非同期タスクを実装する．非同期タスクは関数定義と似た構文で定義できるが，通常の関数とは内部実装も異なるものであるため，関数として定義すべきか非同期タスクとして定義すべきかは事前に決めなければならない．それぞれの構文は以下の通りである．ただし，どちらも引数は 64 ビット符号付き整数型 `i64` ，返り値は文字列型 `String` 型とした場合である．

```
1 // 関数
2 fn f(arg: i64) -> String {
3     // ...
4 }
5
6 // 非同期タスク
7 async fn a(arg: usize) -> String {
8     // ...
9 }
```

プログラム 2.6　関数，および非同期タスクを定義する構文

関数と非同期タスクは呼び出し側から見ても異なるものである．どちらも呼び出すための構文は同じであるが，関数から非同期タスクを呼び出すことはできない．`main` 関数も関数であるため，通常通り `main` 関数を定義して，その中で非同期タスクを呼び出すことはできないということになる．`main` 関数では，ランタイムシステムを起動し，必要に応じて複数のスケジューラを起動し，そのランタイムに非同期タスクを実行させるという処理を書かなければならない．Tokio を用いる場合は `main` 関数に `tokio::main` 属性をつけることによってある程度の設定は行えるが，後述する Actix を用いる場合は，プログラマが全て明示的に記述する必要がある．

なお，非同期タスク呼び出しの返り値は，その非同期タスクの型シグネチャにある返り値ではない．実際には，その非同期タスクを実行するための状態マシンを返す．上記のプログラム 2.6 で定義した関数，および非同期タスクを呼び出す場合は次のプログラム 2.8 のよう

になる．

```
// 関数呼び出し
let s: String = f(10);

// 非同期タスク呼び出し
let s: impl Future<Output = String> = a(10);
```

プログラム 2.7 関数，および非同期タスクを呼び出す構文

「`let s: T = ...`」 という構文は，型 `T` の変数 `s` の宣言と初期化を行う構文である．「`impl Future<Output = String>`」という型は次のような意味になる．

- 返り値の型は `Future` トレイトを実装している
- その `Future` トレイトが将来的に決める値は `String` 型である

つまり，`a(10)` を評価した段階では，実際に非同期タスク `a(10)` が実行されるわけではなく，その非同期タスクをスケジューリングするために必要な情報やメソッドを含むデータ構造が返される．これを `String` 型の値に解決する際は， `await` 構文を用いて次のようにする．

```
// 非同期タスク呼び出し
let s: String = a(10).await;
```

プログラム 2.8 `await`の使用例

関数のみからなる逐次コードを非同期タスクを用いたコードに単純に移植する場合は，次のような変更が必要になるため，移植コストが高い．

- 非同期タスクに変更したい関数を，すべて `async fn` に変更する．
- その関数を呼び出している部分について，すべて `await` するように変更する．
- `main` 関数を非同期タスクを実行するためのものに変更する．

2.2 Actix

Actix は Tokio 上で動作する Rust のアクターフレームワークである [*5]．本節では Actix の基本的な概念を解説する．

2.2.1 アクターの実装

Actix におけるアクターは，ある動作とその状態をカプセル化するオブジェクトである．アクターを実装するためのコアとなる機能は `Actor` トレイトと，そのアクターが受信したメッセージを処理するための `Handler<T>` トレイトである．Actix では各アクターをイベント駆動型のプログラミングモデルによって定義する．つまり，プロセスの起動やメッセージの受信などをきっかけとして，ユーザが実装したメソッドがコールバックされる．

Actor トレイトが要求するメソッドは，例えば以下のようなものがある．

start メソッド　アクターを起動する際に呼ばれるメソッドで，起動されたアクターの識別子を返す

started メソッド　アクターが起動された直後に呼ばれるメソッドで，返り値はない

stopping メソッド　アクターの停止要求があった際に呼ばれるメソッドで，停止するか停止要求を無視するかを返り値で指定する

stopped メソッド　実際にアクターが停止される際に呼ばれるメソッドで，返り値はない

`Handler<T>` トレイトは型 `T` のメッセージを受け取った際の振る舞いを定義する．型ごとに定義できるため，複数の型のメッセージを受信するアクターについては，`Handler<T>` の実装を複数記述することになる．

[*5] ここで扱う Actix フレームワークと，Rust の Web フレームワークのデファクトスタンダードである Actix-web は，同じ組織が開発・保守しているフレームワークである．似た名前が用いられているが，実際には 2 つのフレームワークはほとんど無関係であり，Actix-web のランタイムシステムとして Actix が採用されているわけではない．

2.2.2 Arbiter

Arbiter は複数のアクターが実行される環境をホストするものである．すなわち，Arbiter はイベントループを管理しており，生成されたアクターを適切にスケジューリングする．アクターをスポーンする際にはどの Arbiter 上に生成するかは明示しなければならない．各アクターは生成された後に異なる Arbiter に移動することはできないが，異なる Arbiter 上で実行されるアクター同士でメッセージパッシングを行うことは可能である．

1 つの Arbiter は 1 つの OS スレッドを制御する．したがって，複数の Arbiter を作成すると，複数の OS スレッドを用いて複数のイベントループを並行して実行することを意味する．この際各 OS スレッドに均等に負荷分散したい場合，どの Arbiter 上にどのアクターを生成するかを決定するのはプログラマの仕事であり，負荷分散機能は組み込まれていない．

2.3 Erlang/OTP について

本節では，提案フレームワークのロジックの元となる Erlang/OTP[7, 8] について解説する．Erlang/OTP は，主にスケーラブルなソフトリアルタイムシステムの構築を目的として使用されるプラットフォームである．Erlang はそのプラットフォームが提供する関数型プログラミング言語であり，OTP はその標準ライブラリやミドルウェア，設計原則などのセットをいう．バイトコードにコンパイルされ，Erlang VM（もしくは BEAM）と呼ばれる仮想機械を用いて実行する点や，分散処理，耐障害性，Code Replacement のサポートが組み込まれている点が特徴である．Erlang はバイトコードを用いて実行される都合上計算タスクに関する性能が低いが，JIT コンパイラが導入されたためその問題は少し緩和された [9]．OTP 24 にて x86 の 64 ビットプラットフォームに対応し，OTP 25 にて AArch64 アーキテクチャに対応している．

本節では，OTP の設計原則について提案フレームワークのロジックに関わる部分を解説する．

2.3.1 監視ツリー

監視ツリーとは，ワーカ・スーパバイザモデルにおけるプロセスの監視構造の一つである．ワーカプロセスが実際の計算を行うプロセスであるのに対し，スーパバイザプロセスは各ワーカプロセスや，別のスーパバイザプロセスを監視する役割を持つ．スーパバイザプロセスが監視しているプロセスが何らかの理由で終了すると，あらかじめ設定した戦略に従ってそのプロセスを再起動する．このような役割を持つ二種類のプロセスを階層的に配置することで，耐障害性を満たすようなシステムの構築が可能となる．

2.3.2 ビヘイビアとコールバック

監視ツリーを構成する各プロセスは似たような振る舞いを持つ．例えば，スーパバイザは以下のような振る舞いをするプロセスであると一般化できる．

- まず，自分が監視するプロセスを起動する
- その後，プロセスが終了したことを表すシグナルを受け取るまで待機し，必要に応じて再起動する

どのようなプロセスを起動・監視するか，または子プロセスの終了を捕捉した際，再起動するかどうかの具体的な戦略等についてはアプリケーションのデザインに依存する．このように，大枠だけを定義し，具体的な部分を抽象化したモジュールをビヘイビアモジュールと呼ぶ．

プログラマがビヘイビアモジュールを使用する際は，そのビヘイビアモジュールが指定したコールバック関数を実装し，それらの関数をエクスポートするように作成したコールバックモジュールを用意する．以降，スーパバイザを例に用いて具体的なコードを挙げる．ここでは，子プロセスを一つも起動しない `nochild` という名前のスーパバイザプロセスを実装したいとする．OTP には `supervisor` という名前のモジュールが提供されており，これがスーパバイザプロセスを実装するためのビヘイビアモジュールに該当する．`supervisor` モジュー

ルが指定するコールバック関数は， init/1 関数のみである．従って，これをエクスポートするモジュールを用意する．

```
1 -module(nochild).
2 -export([init/1]).
3
4 init(_Args) ->
5     SupFlags = #{strategy => one_for_one, intensity => 1, period => 5},
6     ChildSpecs = [],
7     {ok, {SupFlags, ChildSpecs}}.
```

プログラム 2.9 nochild モジュールの定義

init/1 関数の返り値を見ると，{ ok, { SupFlags, ChildSpecs }} という値を返していることがわかる．{} で囲まれた値はタプルを表しており， ok というのはアトムである．SupFlags という変数は 5 行目で指定されているマップが束縛されており，ChildSpecs という変数は 6 行目で空のリストが束縛されている．

supervisor モジュールは，コールバック関数の定義の他にもスーパバイザを起動するための API を提供している．supervisor:start_link/2 などが該当する．この関数の第 1 引数はモジュール名（アトム）で，第 2 引数はそのモジュールの init/1 関数に渡される引数である．supervisor モジュールは start_link/2 関数が呼ばれると，つまり supervisor:start_link(Mod, Arg) が評価されると，内部で Mod:init(Arg) が評価され，その返り値を見てスーパバイザ自身の仕様や，起動・監視する子プロセスの仕様を決定する．この方法によって起動された子プロセスやスーパバイザプロセスは全て並行，または並列に動作する．

OTP にはスーパバイザの他にも，次のようなビヘイビアモジュールを提供している．

gen_server 一般的なサーバ・クライアントモデルにおけるサーバの振る舞いを抽象化したもの

gen_statem 状態マシンの振る舞いを抽象化したもの

gen_event イベントハンドラの振る舞いを抽象化したもの

OTP の設計原則に関する概説は以上で終了し，以降は Erlang から機械語で実装された関数を呼び出すための機構について解説する．

2.4 NIF

Erlang には Native Implemented Function（NIF）と呼ばれる機構が備わっており，C 言語や Rust で実装した機械語関数を Erlang から呼び出せる．Erlang からは通常の関数呼び出しと全く同じ形で扱え，NIF の呼び出しにはコンテキストスイッチが不要である点が特徴として挙げられる．主に Erlang で実装した場合には性能の問題が発生する場合に使用される方法の一つであり，行列演算を C 言語で NIF として実装した場合，Erlang のみを用いて実装した場合よりも演算処理の性能が大幅に向上したというような，NIF の有用性を示す報告がいくつかある [10, 11].

ただし，NIF を使用する際にはいくつか注意点がある．NIF の実行中にプログラムがクラッシュすると，Erlang VM 上で動作する他のすべてのプロセスを道連れにしてしまう．この問題は，Rust を用いてプログラムがクラッシュしないように実装し，エラーが発生しても Erlang VM に影響が出ない形で終了するように NIF を実装することで解決できる．また，後述するように NIF が呼び出されてからリターンするまでに長時間かかるような状況は好ましくないとされている．その場合の対処法はいくつか存在するが，ここでは Dirty Scheduler を扱う方法について解説する．

Rust で NIF を実装するためのライブラリの一つに Rustler[12] というライブラリがある．このライブラリは crates.io[*6] で公開されているため，誰でも使用可能である．本論文では C 言語を用いる方法は扱わず，Rustler ライブラリを用いて NIF を実装する方法のみ取り扱う．

[*6] `https://crates.io`

2.4.1 NIF のロード

NIF は Erlang 関数を機械語によって実装したものであるため，NIF を実装する際は Erlang モジュールの実装も必要となる．例えば，Erlang で `mod` という名前のモジュールに `f/1` という関数を NIF で実装したいとする．この際，まずは Erlang で次のように記述し，`mod` モジュールの `f/1` 関数を実装する．ただし，関数 `f/1` の具体的な実装は割愛している．

```
1 -module(mod).
2
3 f(Args) -> # ...
```

プログラム 2.10 `mod`モジュールの実装例

NIF はモジュールが Erlang VM にロードされた後にリンクされる必要がある．従って，`on_load` ディレクティブを用いてモジュールがリンクされた際に呼び出される関数を指定する．ここで NIF をロードする関数を呼び出せば，NIF がロードされる．

2.4.2 Resource Type

NIF を実装する際，その返り値は Erlang が解釈できるデータ型にエンコードできるものでなければならない．整数や文字列などプリミティブ型についてはエンコードする機能が Rustler ライブラリに組み込まれているが，ユーザ定義の構造体や列挙型についてはエンコード方法をユーザが定義する必要がある．これには Rustler ライブラリが提供する `Encode` トレイトを実装すれば良いが，Rust のトレイトの制約により，例えば外部ライブラリが提供するデータ型に `Encode` トレイトを実装することができない．このようなデータ構造を NIF で扱いたい場合は，NIF が提供する API を通じてメモリ領域を確保し，その領域にデータを格納し，そのポインタを返すように NIF を実装する．このデータ型は Resource Type と呼ばれる．Rustler ライブラリにおいては， `ResourceArc<T>` 型を用いることで Resource Type が容易に扱えるようになっている．`Arc` というのはアトミックな参照カウンタのことを指しており，プロセス間で共有しても安全な参照である．なお，Resource Type として扱う必要のあるデータ型については，`resource` マクロを用いてあらかじめ宣言する必要があ

る．もし宣言していない型に対する `ResourceArc` を作成してもコンパイラがエラーを出して拒否される．この保護機能は，Rustler ライブラリの内部では特定のトレイトを宣言された型に対して実装することで実現しており，そのトレイト自体は外部に公開されていないため，マクロを経由せずに実装することはできないようになっている．

Resource Type は Erlang モジュールごとにローカルな識別子によって管理されている．したがって，ある型 `ResourceArc<T>` をモジュール `M1` の NIF でエンコードした値は異なるモジュール `M2` でデコードしてもうまくデコードできない点には注意が必要である．その場合の対処方法は，Erlang のプリミティブ型でうまく表現する方法をユーザが考える必要がある．

2.4.3 Dirty Scheduler

Erlang VM は，バイトコードを実行する際は通常のスケジューラを用いてスケジューリングを行う．この際はプリエンプティブなスケジューリングが行われるため，実行中 Erlang VM が適切なタイミングでプロセスの実行を中断し，実行待ち状態の他のプロセスにリソースを再割り当てする．しかし，NIF の実行中はスケジューラがブロックされるため，プリエンプティブなスケジューリングが実施できない．したがって，NIF が自発的にリソースを明け渡すのを待たなければならない．現在 NIF では `enif_schedule_nif` という API が提供されており，これを用いて自身をスケジュールし直すことができる．つまり，一度リソースを手放して，他のプロセスが利用できる状態にできる．ただし，2024 年 1 月現在この API のラッパーとなる機能は `rustler` ライブラリでは提供されていない．

このような機能を用いず，呼び出しからリターンするまでに 1 ミリ秒以上時間がかかる NIF を Dirty NIF と呼んでいる．このような NIF を実行する場合で，かつ先ほど述べたように途中で中断し，後ほど再開するような実装が困難な場合の解決策として，Dirty Scheduler というものが提供されている．Dirty Scheduler には CPU バウンドな Dirty NIF をスケジューリングするためのものと，IO バウンドな Dirty NIF をスケジューリングするためのものの二種類が用意されており，どちらのスケジューラを用いるかは，NIF の実装者

が適切に選択し，NIF を実装する際に宣言しておく必要がある．

第 3 章

研究目的

本章では本研究の動機，および目的について解説する．

Rust で（Tokio に代表されるライブラリを用いて）グリーンスレッドによる非同期プログラミングを行う場合，特に複数のタスクが協調動作をする場合において排他制御を行う必要がある．アクターモデル [13] による並行プログラミングは，メッセージパッシングによる協調モデルを特徴とする．このモデルは Erlang/OTP や Akka フレームワーク [14] に採用されており，アクター同士は独立したメモリ空間を持つため排他制御が不要である点がメリットとして挙げられる．また，アクターモデルを導入することによる記述容易性の向上については， [15] において GPU プログラミングに対しても，同期や排他制御のために特別な処理を記述する必要が無くなったと報告されている．

本研究では，グリーンスレッドによる非同期プログラミングをアクターモデルによって実現する方法として新しいフレームワークを提案する．まずは研究の動機として，同じアクターモデルを基とする Erlang と Actix による並列プログラミングについて比較，分析する．

3.1 Erlang VM と Actix の比較

Erlang と Actix はどちらもアクターモデル [16] による並列処理が実現可能である．ここでは，耐障害性とアクターの配置について焦点を当てて比較し，問題提起する．なお，本節では Erlang のプロセス，および Rust の非同期タスクのことをまとめてアクターと呼ぶことにする．

3.1.1　耐障害性

Actix は `Supervised` という名前のトレイトを提供しており，アクターに対して実装することでそのアクターの再起動ができる．ただし，この機能はあくまでもアクターの終了処理後に何かをする（例えば再起動する）ように拡張するだけであり，異常終了を検知してアクターを再起動させるものではない．また，協調動作する複数のプロセスを同時に再起動させるような機能が組み込まれていない．さらに，Actix はあるアクターのクラッシュが他のアクターに影響する可能性がある．以下の図 3.1 は，ランタイムシステムの上に 3 つのスケジューラが起動されており，各スケジューラがいくつかのアクターを制御している様子を表している．図中のバツがついたアクターがパニックを起こしたとする．この時，そのパニッ

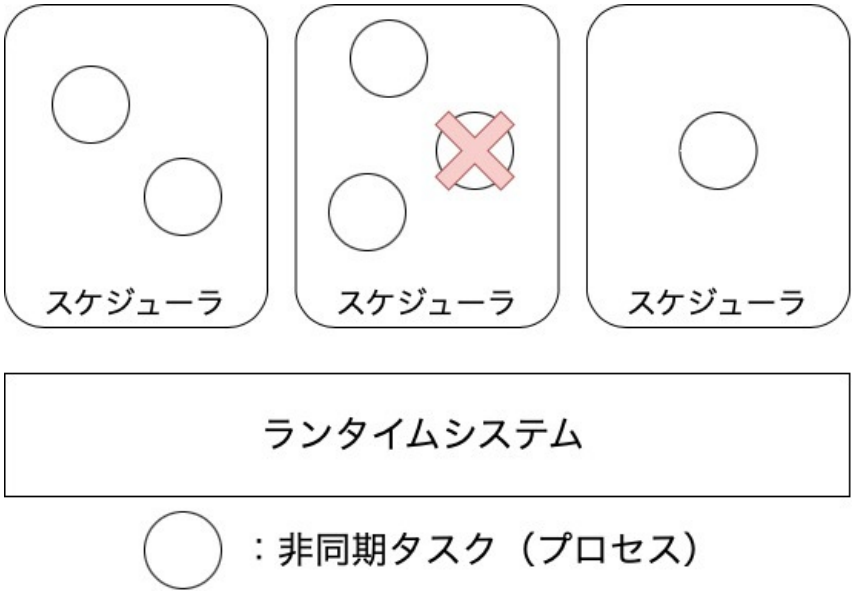

図 3.1　3 つのスケジューラを用いて 6 つのアクターを制御している様子

クが捕捉されずにスケジューラに伝播すると，他のアクターを道連れにして復旧不可能な状態に陥る．この際，道連れにされるアクターは終了処理を行う余地はなく，スーパバイザによる再起動のサポートもない．さらに，このパニックはランタイムシステムにも伝播するため，他のスケジューラも全て道連れにされることになる．Actix のランタイムシステムは，あるスケジューラの異常終了を検知すれば，他のスケジューラを定められた手続きに従って終了させるようになっているため他の二つのスケジューラ（図 3.1 における右と左に配置

されたスケジューラ）上で実行されている非同期タスクは終了処理を行う余地がある．しかし，各タスクは同じスケジューラ上で実行される他のタスクの異常終了を検知する機能が組み込まれておらず，他のスケジューラ上で実行されるタスクが終了した場合も再起動されることなくランタイムシステムが停止するため，一つのタスクの異常でシステム全体が停止することになる．このことをまとめると，次の図 3.2 のようになる．異常が発生する前の状態

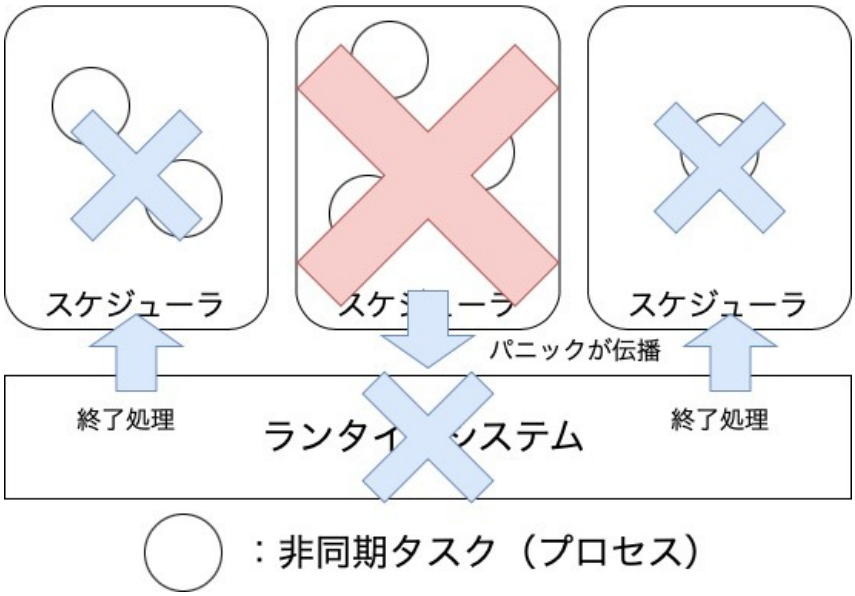

図 3.2　あるタスクがパニックを起こした後の処理

に復旧する機能も組み込まれていないため，現状では障害に対しては脆いフレームワークであると言える．

一方で，Erlang VM は Let it Crash が実現しやすいように設計されている．すなわち，何か異常を検知すればクラッシュし，すぐに再起動することで復旧するという考え方である．Rustler ライブラリを利用することで，Erlang VM に NIF のパニックが伝播することなく捕捉される．従って，NIF の実行中にパニックが発生しても，それが他のアクターに影響を及ぼすことはない．また，Erlang のスーパバイザは，NIF のパニックを含む全ての異常終了を検知し，再起動させることができる．

監視ツリーの概念を省略し，図 3.1 と似た状況を想定した図を図 3.3 に示す．この図では

パニックを起こしたアクターとそれを監視するアクターが同じスケジューラで制御されていると仮定しているが，監視するアクターがどのスケジューラで実行されているかの違いは重要ではない．この状況で，パニックを起こしたアクターは自分だけ終了させ，他のプロセス

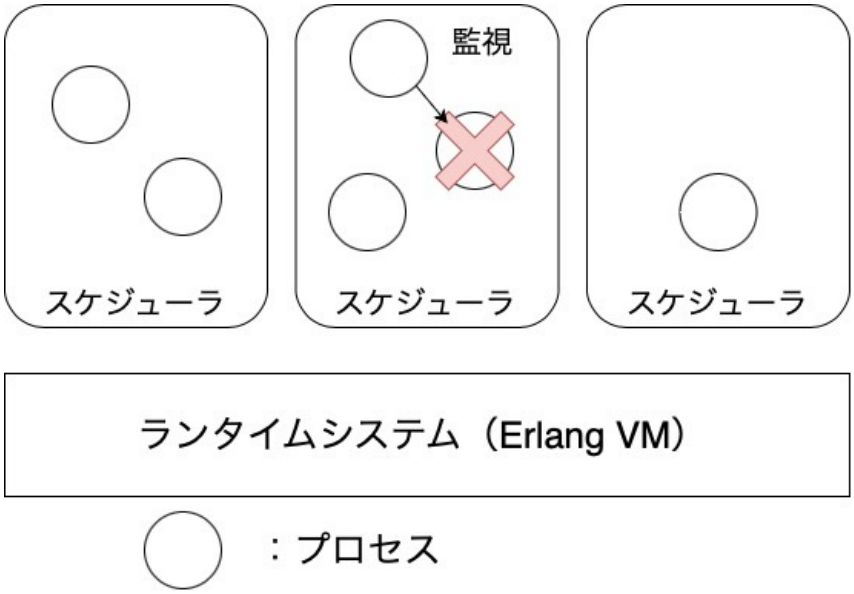

図 3.3　図 3.1 と似た状況について，ランタイムシステムを Erlang VM とした場合の概念図

は巻き込まない．そして，そのアクターを監視していたアクターによって再起動され，初期状態へと戻される．このシナリオを図で表すと，以下の図 3.4 のようになる．終了する前の

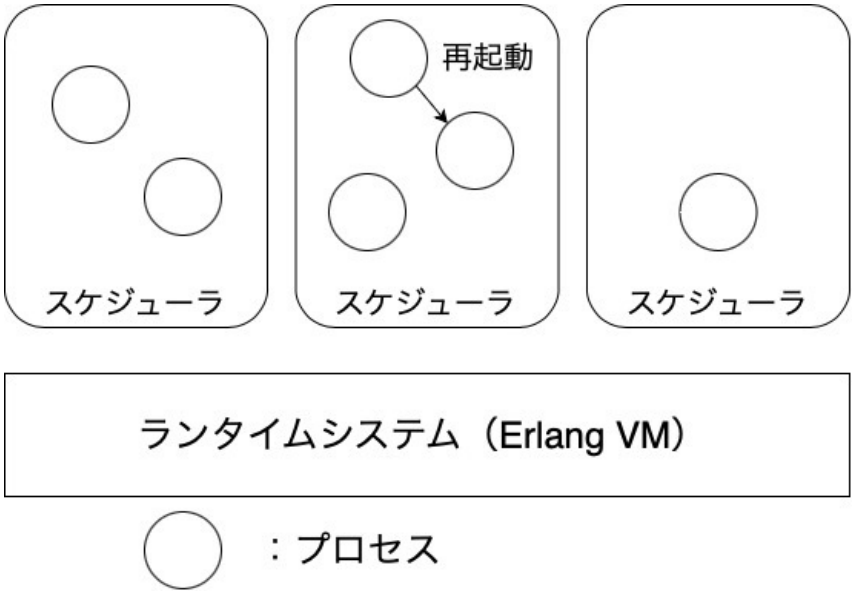

図 3.4　図 3.3 の状況の後，再起動される様子

状態にアクターを復旧する機能は自力で実装する必要があるが，パニックが他のアクターを巻き込まないため，復旧に用いるデータを保持するプロセスを別途定義すれば良い．このアイデアは [17] による．

3.1.2　タスクの配置

複数のアクターを扱う場合において，Actix はどアクターをどのスケジューラに扱わせるかを明示しなければならない．単純なアルゴリズムでも決定できるが，スケジューラごとに仕事量の偏りが生じる可能性があるため慎重に決定する必要がある．

Erlang VM を用いる場合，各アクターをどのスケジューラに扱わせるかは Erlang VM が決定するため，このような手間はかからない．

なお，Tokio に代表される一部のグリーンスレッドのランタイムシステムを提供するライブラリでは，単一のスケジューラ上で実行するか複数のスケジューラ上で実行するかだけを選ぶだけでよく，この場合はタスク [*1] の配置や管理の責任はランタイムシステムにある．ただし Tokio を用いる場合はタスク同士で協調動作をする場合には排他制御が必要になるなど，アクターモデルにはない新たな問題が発生するため，単純化のため本研究では Actix にのみ焦点を当てることとする．

3.2　研究目的

前節での観察を踏まえて，本節では研究目的について述べる．Erlang VM と Actix を比較すると，Erlang VM を用いる利点は以下のようにまとめられる．

- Let it Crash を実現しやすいため，エラーハンドリングの際に他のアクターの影響を考えなくて良い
- 各アクターをどのスケジューラに配置するかは決めなくて良い

[*1] tokio はアクターモデルに基づくものではないため，ここでは Tokio の流儀に合わせてタスクと称する．

したがって，我々は Erlang VM 上で動作するアプリケーションを Rust のみ用いることで開発できれば，耐障害性を満たし，大量のアクターが並行動作するようなシステムをスケジューラのことを考慮する必要なく構築できると考えた．また，ビヘイビアとコールバックの仕組みと NIF を適切に組み合わせ，コールバック関数をすべて Rust で記述すれば，ユーザは Erlang コードを書かなくても良くなるはずである．

しかし，コールバック関数を全て NIF で記述するとしても，Erlang コードを記述する必要がある．Rust だけで OTP の設計原則に則ったアプリケーションを開発できるようにするためには，そのような Erlang コードを隠蔽しなければならない．そこで，本研究ではその役割を担うフレームワークを開発する．

また，フレームワークを選択する際はパフォーマンスも重要な指標と言える．そこで，アクターモデルのためのベンチマークプログラムを Actix を使用する場合と提案フレームワークを使用する場合と，OS スレッドを用いる場合で実装して性能評価実験を行い，結果について考察する．

また，そのフレームワークによって Actix でサポートされていない機能に容易に対応できることを確かめるために，簡単なプログラム例を示して Actix の耐障害性と提案フレームワークの構築容易性や耐障害性について検討する．

第 4 章

フレームワークの設計

本章では，提案フレームワークのデザインについて述べる．

提案フレームワークは二つのツールから成る．一つは，フレームワークのユーザ（以下，単にユーザと称する）が書いた Rust プログラムから OTP の設計原則に則ったアプリケーション（以下，OTP アプリケーションと称する）をビルドし，実行するためのコマンドラインツール．もう一つは，NIF が適切な API を持つよう強制するためのマクロを定義したライブラリである．

まずはフレームワークの全体像を述べた後，各コンポーネントについて詳細に解説する．

4.1 フレームワーク概要

提案フレームワークは Rust を用いて OTP アプリケーションを構築できるようにするためのものである．OTP アプリケーションをプログラミングする際のフローを大まかに記述すると次のようになる．これは NIF を使用する場合も，使用しない場合にも共通する開発フローである．

1. 監視ツリーを設計する
2. 各プロセスについて，適切なビヘイビアモジュールを選択しコールバック関数を実装する

以降，各フェーズについて詳細に述べる．

4.1.1　監視ツリーの設計

まずは OTP アプリケーションそのものの構造を設計する．なお，ここで述べる内容は OTP アプリケーションを開発する前の設計段階のものであり，実際にソースコードとして記述する内容ではない．

監視ツリーを構成する各プロセスのうち葉ではないものは全てスーパバイザプロセスであり，子プロセスを起動・監視する．葉にあたるプロセスはスーパバイザ以外のビヘイビアモジュール（例えば `gen_server` ビヘイビアなど）を用いて実装される．

提案フレームワークでは，監視ツリーの構造は静的に決定されると仮定している．また，他にも監視ツリーについて以下の仮定もしている．

- 連結な木構造である．
  - 監視ツリーの根となるスーパバイザプロセスから全てのプロセスが起動される必要がある．
  - あるプロセスを監視するスーパバイザプロセスは一つしか存在しない．
- あるスーパバイザプロセス，およびその子孫プロセスが自分自身と同じプロセスを起動する際に無限ループに陥らない．
  - 再帰関数に基底条件を設けないと無限に再帰し続けてしまうのと同じ原理．
  - すなわち，再帰的なプロセス生成に基底条件を適切に設けるならば問題は生じない．

例えば，以下の図 4.1 ような構造を持つことになる．

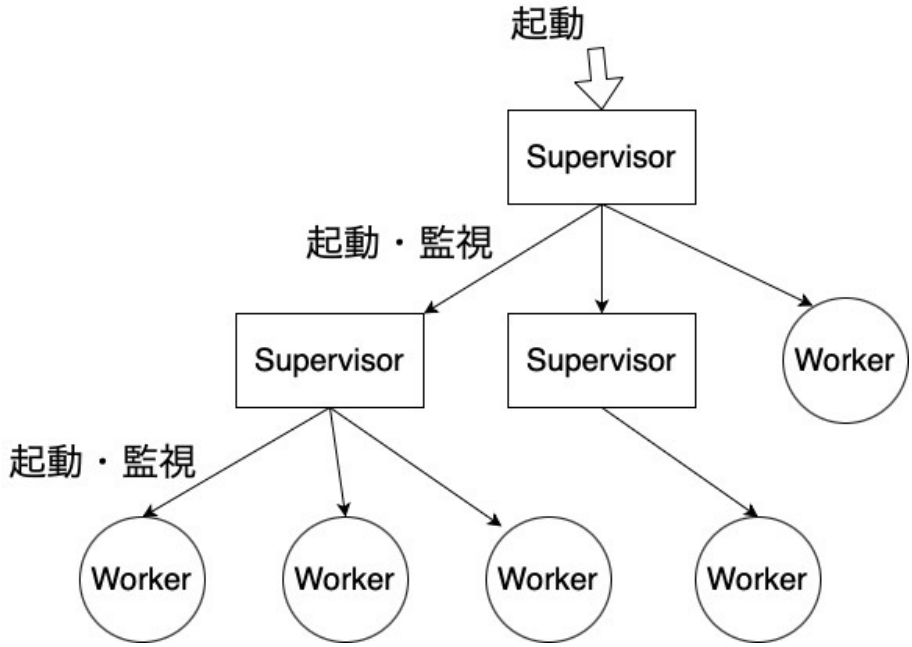

図 4.1　監視ツリーの構造の例

図中の各図形はプロセスを表現しており，四角はスーパバイザプロセス．円はワーカプロセスを表している．また，矢印はスーパバイザが子プロセスを起動・監視する様子を表しており，白抜き矢印は監視ツリーの根に当たるプロセスがコマンドラインインタフェースより起動される様子を表している．

スーパバイザプロセスは，自身が起動された直後に子プロセスを起動する．従って，コマンドラインインタフェースからスーパバイザの根に当たるプロセスを起動すれば，監視ツリーを構成するすべてのプロセスが起動されることになる．

4.1.2　各プロセスの実装

監視ツリーの設計が完了すれば，あとは各プロセスの実装を記述するだけである．まずは各プロセスの役割から，どのビヘイビアモジュールを選択するのが適切かを考え，そのビヘイビアモジュールのコールバック関数について調べる．例えばスーパバイザプロセスであれば，`init/1` 関数を定義すれば良いことがわかる．

OTP が定義した仕様を満たすように各コールバック関数を定義することになる．提案フ

レームワークにおいては，このコールバック関数の定義を全て NIF として Rust で記述することになる．

全プロセスの実装が定義できたあとは，後に述べるビルドツールを用いて OTP アプリケーションをビルド・実行することができる．

4.2　詳細設計

本節では，提案フレームワークの内部構造について詳細に解説する．

4.2.1　提案フレームワークの役割

提案フレームワークは，Erlang VM とユーザが定義した Rust 関数の仲介を担う．以下の図 4.2 は，提案フレームワークの役割を表している．図中の青文字は，誰がその部分の責任を負うのかを表している．

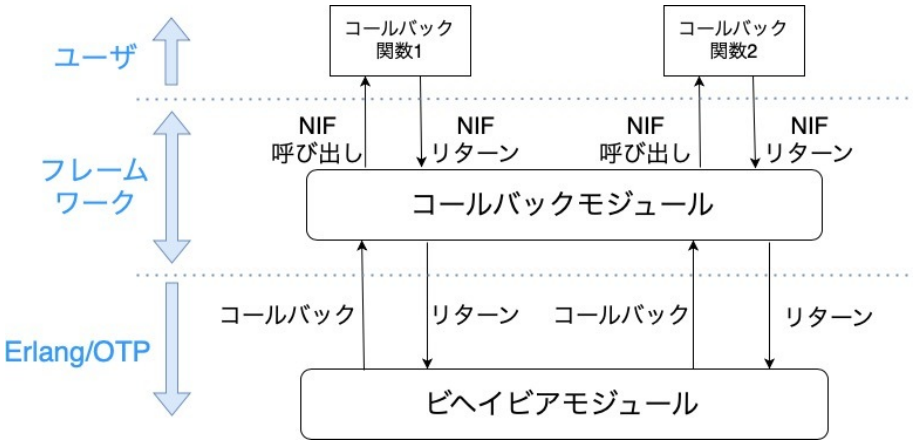

図 4.2　提案フレームワークの役割

まず，Erlang/OTP にビヘイビアモジュールが定義されており，所望のプロセスを実装するために必要なコールバックモジュールを実装する．この時，実際のコールバックモジュールの実装は Erlang モジュールであるため，Rust によるコールバック関数の実装が含まれる動的ライブラリをロードするための Erlang モジュールを自動で展開してリンクするように

する．こうすることで，プロセスの初期化やメッセージの受信などをきっかけに，ビヘイビアモジュールからフレームワークが展開したコールバックモジュールの関数がコールバックされる．この時，フレームワーク内部ではリンクされた NIF が呼び出されるようになっているため，その返り値を受け取り，ビヘイビアモジュールへと返される．なお，NIF の実行中にパニックが起こった際はそのプロセスがクラッシュするが，そのプロセスを監視するスーパバイザによって再起動される．

4.2.2 各プロセスの起動時の引数

まず，すべてのプロセスは生成される際に引数を受け取る．スーパバイザプロセスが子プロセスを起動する際にその引数を与えることになるため，監視ツリーのルートプロセス以外は，ユーザがスーパバイザプロセスの実装にその引数を含められるが，監視ツリーのルートプロセスは，コマンドライン引数を受け取るように設計されている．

4.2.3 ブロッキングしてはいけないコールバック関数の扱い

各プロセスが起動する際は，その初期化処理が必要である．Erlang/OTP が提供している，4 つの主要なビヘイビアモジュール（supervisor, gen_server, gen_statem, gen_event）では `init/1` 関数が初期化処理を行うためのコールバック関数である．特に，supervisor モジュールに定義されている初期化処理は，自身の子プロセスを起動する際にコールバックされる `init/1` 関数が返るまで，スーパバイザを起動する関数（`start_link/2` など）も返らない．つまり，子プロセスの初期化は逐次的である．

以下の図 4.3 は，あるスーパバイザプロセスから二つのワーカプロセスが起動される際の手順を表している．

1. ワーカ 1 を起動するために，ワーカ 1 の初期化関数を呼び出す
2. ワーカ 1 の初期化関数がリターンするのを待つ
3. ワーカ 2 プロセスを起動するために，ワーカ 2 の初期化関数を呼び出す

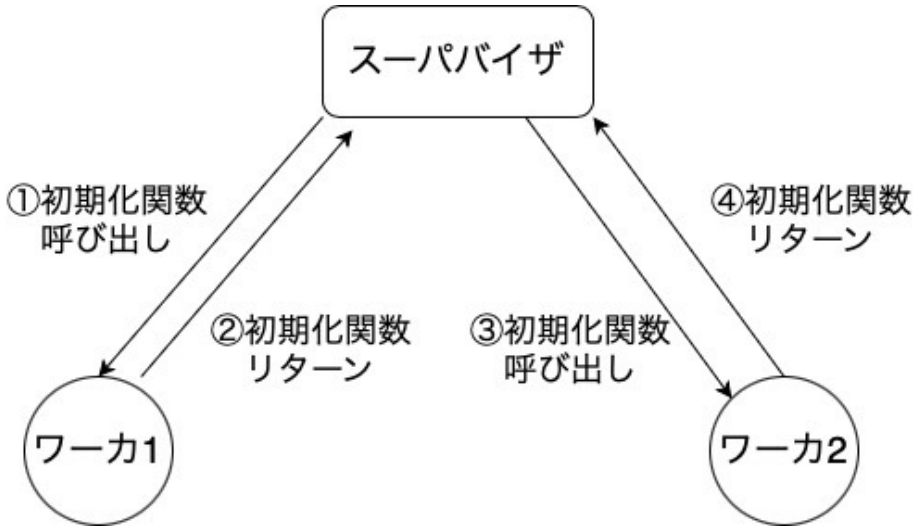

図 4.3　子プロセス初期化の手順

4. ワーカ 2 の初期化関数がリターンするのを待つ

従って，2 の手順でワーカ 1 の初期化関数が（例えばワーカ 2 プロセスの PID を取得しようとするなどして）ブロッキングしてしまうと，デッドロックが発生しアプリケーション全体が停止する．このような問題に対する回避策として，提案フレームワークでは以下の `handle_start/2` コールバック関数を追加し，拡張した．

- 第 1 引数は `init/1` コールバック関数の第 1 引数と同じ値，第 2 引数は `init/1` 関数が返した，プロセスの内部状態の初期値
- 返り値の仕様は `handle_cast/2` コールバック関数と同様

ただし，現時点では gen_server に対してのみサポートしており，他のビヘイビアモジュールについてはサポートしていない（なお，supervisor については子プロセス，およびスーパバイザ自身の仕様を規定するだけであるため，この機能のサポートは不要と考えている）．

4.3　ビルドツール cargo-otp

NIF を用いて OTP アプリケーションを構築するためのビルドツールとして `cargo-otp` というツールを開発した．つまり，`cargo` ビルドツールのサブコマンドとして `otp` というものを実装した．

このサブコマンドはさらにサブコマンドを持っており，以下の 4 つがある．

new　新しい OTP アプリケーションを作成する．指定した名前のディレクトリが生成され，そこが作業用ディレクトリとなる

add　新しいプロセスを実装するためのパッケージを追加する

build　作成した OTP アプリケーションをビルドする

run　ビルドした OTP アプリケーションを実行する

本節の残りの部分では，`cargo-otp` が処理できるディレクトリ構造や，書かなければならない設定ファイル等について解説する．

作業用ディレクトリの構造は次のような形になる．アプリケーション名は `myapp` としている．

```
myapp
├── actor1
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── actor2
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── ...（以降，アクターの種類の数だけ続く）
├── Cargo.lock
├── Cargo.toml
├── stroot
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
└── otp.toml
```

otp.toml というファイルが提案フレームワークに関する設定ファイルで，他のファイルはすべて Cargo が指定するディレクトリ構造となっており，各プロセスの実装（すなわち Erlang モジュール）ひとつにつき 1 つのクレートが対応している．なお， stroot という名前のクレートは，監視ツリーのルートプロセスを実装するためのクレートである．この名前は変更できるが必要不可欠なクレートであるため，フレームワークでデフォルトの名前を指定した．

new コマンドで作成された直後の otp.toml の内容は以下のようになっている．

```
1 [system]
2 stroot = "stroot"
3
4 [behaviours]
5 stroot = "supervisor"
```

プログラム 4.1　otp.tomlの初期内容

Cargo の設定ファイル `Cargo.toml` と同様 TOML 形式で記述する．

`system` テーブルの `stroot` キーは，監視ツリーの根に当たるスーパバイザを指定する．`cargo otp run` が実行されると， `stroot:start_link/3` が評価されるようになっており，この引数にはコマンドライン引数が与えられるようになっている．

`behaviour` テーブルは，各クレートがどのビヘイビアモジュールを使用するかを指定する．これはビルドする際に，すべてのコールバックモジュールの Erlang 実装を与えるために必要な情報である．`build` コマンドを実行すると，NIF の Erlang 実装を自動的に展開し，`erlc` コマンドを呼び出してコンパイルする．

なお， `add` コマンドで新しいクレートを作成する際にこの `otp.toml` ファイルも書き換わるようになっているため，ユーザが直接このファイルを編集する機会はほとんどない．

4.4 otp_wrap ライブラリ

先述した `Supervisor` トレイトなどを含むライブラリ．主な役割は，適切に NIF が実装されていることをできる限り Rust コンパイラが検出できるようにすることである．

しかし，現段階では Erlang のビヘイビアモジュールが要求するコールバック関数をすべて実装していることと，そのコールバック関数のアリティが有効なものであることを検証することができるのみである．

このライブラリは Rust のマクロの機能を用いて，コールバック関数が適切に実装されていることを強制する．Erlang VM から直接呼ばれる NIF は，このマクロが展開する関数であり，この関数がユーザが定義したトレイトメソッドを呼び出すという手法を用いている．

4.5 ユーザの学習コストに関する考察

最後に，Actix を用いる場合と提案フレームワークを用いる場合についてユーザが学習しなければならない事柄についてまとめ，学習コストの差について考察する．

提案フレームワークはアクターモデルに基づくものであるため，アクターモデルに関する

知識を必要とする．しかし，この知識は Actix を用いる場合も同様に必要なものであるため，大きな差は生じないと言える．

提案フレームワークは Rustler ライブラリを使用することが前提となっている．したがって，ユーザは Rustler ライブラリの使用方法を学習するコストがかかることになる．Rustler ライブラリの使用方法を学習するためには，主に Erlang のデータ型に関する知識が必要となるため，最も基本的な知識として Erlang のプリミティブなデータ型に関する知識を必要とする．

また，OTP の設計原則に関する知識も必要とする．監視ツリーやビヘイビアとコールバックに関する知識は必須であり，特にコールバック関数を定義するためには Erlang のオンラインドキュメント [*1] などから定義しなければならないコールバック関数について調べる必要がある．Actix を用いる場合もコールバック関数について調べる必要がある点は同様であるが，提案フレームワークにおいてはコールバック関数の仕様を読み取るためには Erlang のデータ型やその表記方法に関する知識が必要となる．

以上のことから，現状提案フレームワークは Erlang のデータ型に関する知識を前提としており，各ビヘイビアモジュールの機能などを Erlang のドキュメントから読み取る必要がある点が，新たに生じる学習コストであると考えられる．

[*1] `https://www.erlang.org/doc/man/stdlib_app`

第 5 章

性能評価実験

本章では，提案フレームワークを用いて並行システムを構築した際の性能について考察するために，ベンチマークプログラムをいくつか作成し，実験的にその性能を計測する．

そして，その計測結果をもとに，提案フレームワークの性能の特徴について考察する．

本研究では，アクターモデルのベンチマークスイートである Savina[18] からインスピレーションを受けている．本論文では以下の 5 つのベンチマークプログラムを，提案フレームワークを用いた場合と Actix を用いた場合，およびアクター 1 つに付き OS スレッドが一つ生成される場合の 3 種類を実装した．なお，実装はすべて筆者が直接行った．

1. PingPong
2. Thread Ring
3. Forkjoin(throughput)
4. Forkjoin(actor creation)
5. Big

これら 5 つのベンチマークプログラムは Savina ベンチマークスイートではすべてマイクロベンチマークとして設計されている．比較的単純な状況を想定した，メッセージパッシングに関する性能を評価するベンチマークを選択している．

なお，Actix を用いる場合において，どのアクターをどのスケジューラに乗せるかを決定する戦略は，全てのスケジューラに配置されるタスクの数ができるだけ均等になり，かつ可能な限り異なるスケジューラ間で実行されるアクター同士のメッセージパッシングが発生しないと予想される単純なアルゴリズムを設計し，実装している．

5.1　各ベンチマークプログラムの概要

まずは各ベンチマークプログラムの概要について解説する．

5.1.1　PingPong

PingPong ベンチマークは 2 つのプロセス Ping プロセスと Pong プロセスが互いにメッセージを送信し合う．最初に Ping プロセスが Pong プロセスに整数 N を送信すると，Pong プロセスが $N-1$ を送り返す．次に，Ping プロセスが $N-2$ を送信すると，Pong プロセスが $N-3$ を送り返す．以降同様に繰り返し，どちらかが 0 を受信した時点で終了する．

アクターは 2 つしか存在せず，メッセージ送受信以外の処理は最低限であるため，メッセージの送受信にかかるオーバヘッドが測定される．

5.1.2　Thread Ring

P 個のプロセスがリング状に接続されているとみなして，あるプロセスはある整数 i を受信すると，自身の隣のプロセスに対して $i-1$ を送信する．この初期値を N として，あるプロセスがメッセージが 0 を送信し，その隣のプロセスがその 0 を受信したらプログラムはそれまでにかかった時間を出力して終了する．

概略図を以下の図 5.1 に示す．

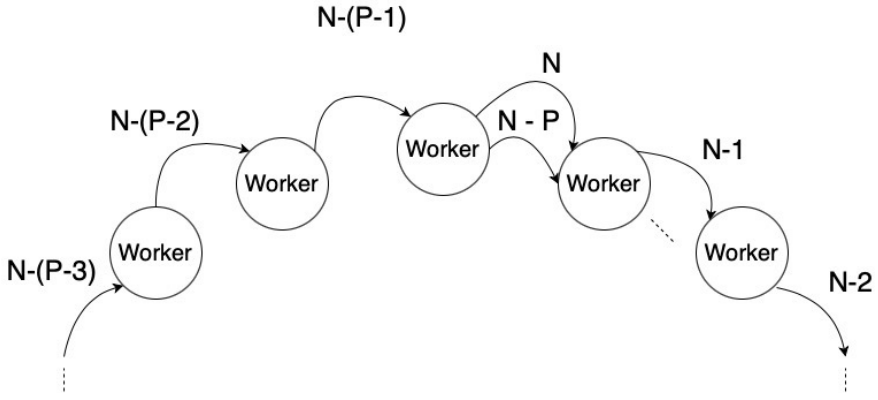

図 5.1　Thread Ring ベンチマークの概略図

中央のワーカプロセスが隣のワーカプロセスに N を送信するところからベンチマークプログラムが開始し，一周するとそのプロセスは $N-(P-1)$ を受信することになる．これを，どれかのプロセスが 0 を受信するまで繰り返す．

各手法の実装に関する仕様は以下のとおりである．

まずは提案フレームワークについて．

- 実際にメッセージの送受信を行うワーカプロセスの他に，ワーカプロセスのうち一つに開始通知を送った後，終了通知を受信するのを待機するプロセスが存在する
- すべてのプロセスを起動し，監視するスーパバイザプロセスが存在する

従って提案フレームワークの場合，実際に定義されたプロセスは三種類ある．

Actix については，各アクターをどの Arbiter 上に生成するかを決定する戦略として以下のようなアルゴリズムを用いた．

1. 各スケジューラが管理すべきタスクの数を求める．
   - スケジューラの数を N_s ，タスクの数を N_t とする
   - このうち， $N_t \mod N_s$ 個のスケジューラは $\lfloor N_s/N_t \rfloor + 1$ 個のタスクを管理し，残りのスケジューラは $\lfloor N_s/N_t \rfloor$ 個のタスクを管理する
2. 各 $i = 1, 2, \ldots, N_t \mod N_s$ について
   - $\lfloor N_s/N_t \rfloor + 1$ 個のワーカプロセスを生成する
3. 各 $i = (N_t \mod N_s) + 1, (N_t \mod N_s) + 2, \ldots, N_s$ について
   - $\lfloor N_s/N_t \rfloor$ 個のワーカプロセスを生成する

このアルゴリズムによって，以下の図 5.2 のような形でワーカプロセスが生成されるようになる．ただし，図中の矢印はメッセージパッシングが発生する流れを表すものである．

PingPong ベンチマークとは異なり，大量のアクターが生成されるため，メッセージの送受信にかかるオーバヘッドに加え，大量のアクターインスタンス間のコンテキストスイッチにかかるオーバヘッドを測定する．

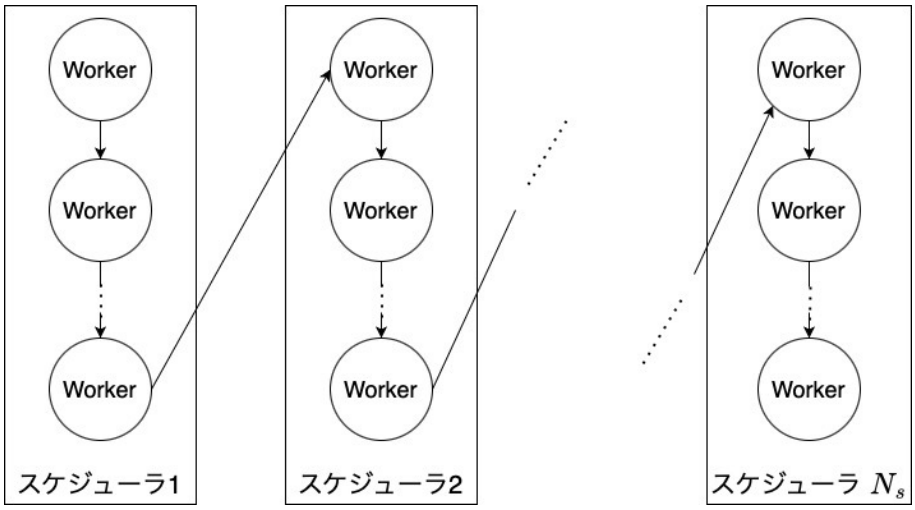

図 5.2　Actix による Thread Ring ベンチマークのタスクの配置戦略

5.1.3　Forkjoin(throughput)

P 個のワーカプロセスと 1 つの送信者プロセスが存在し，送信者プロセスから各ワーカプロセスに対してラウンドロビン形式に各 N 個のメッセージが送信される．ワーカプロセスは内部状態として「残り何度メッセージを受信すべきか」を表す整数値を持つ．送信者プロセスからメッセージを受け取るたびにその整数値をデクリメントし，N 回メッセージを受信すれば，送信者プロセスに終了通知を送る．送信者プロセスがワーカプロセスに対してメッセージを送信し始めてから，すべてのワーカプロセスが終了しそのことを送信者プロセスが確認するまでの時間を計測する．

なお，$P = 1$ の時は Savina ベンチマークスイートにおける Counting Actor と等価であり，$N = 1$ の時は後述する Forkjoin (actor creation) と等価である．

Actix については，各アクターをどの Arbiter 上に生成するかを決定する戦略として以下のようなアルゴリズムを用いた．Arbiter は全部で 16 個生成するとする（実行環境の論理コア数に等しい）．

1. 各ワーカプロセスに対して，0 から順に番号を割り当てる
2. 同様に各 Arbiter に対しても 0 から順に番号を割り当てる

3. 各 $i = 0, 1, \ldots, P-1$ に対して，番号 i を持つワーカプロセスは番号 $i \mod 16$ を持つ Arbiter 上に生成する

複数のアクタが非同期にメッセージを受信するため，メッセージが処理される速度（スループット）を計測するのに適している．

5.1.4 Forkjoin (actor creation)

P 個のプロセスを生成し，それらのプロセスから起動が完了したという旨のメッセージを待機する．先述した Forkjoin (throughput) ベンチマークの $N = 1$ の場合と等価である．

このベンチマークプログラムは 10^5 程度のアクターが生成されるが，リソースの制約上 10^5 個の OS スレッドを立ち上げることが困難だったため，OS スレッドを用いた手法は評価していない．

5.1.5 Big

Big ベンチマークもマイクロベンチマークの一つとして定義されている．P 個のワーカプロセスが生成され，任意のプロセスが他のワーカプロセス全てに対してメッセージを送信する．その後，各ワーカプロセスは $P-1$ 個のメッセージが自身に送信されたことを確認し，終了する．

すべてのプロセスが複数のプロセスからのメッセージを受信するため，メールボックスに対する競合の影響が測定される．

Actix のタスクの配置戦略は，Forkjoin と同じ戦略を用いている．

5.2 実験結果

各ベンチマークプログラムの実験結果を示す．全てのグラフは横軸がパラメータで，縦軸がそのパラメータの時の実行時間（ミリ秒単位）である．なお，各パラメータに対して実行時間を計測するごとに，Erlang VM や Actix のランタイムシステムは再起動されるものと

する．また，各条件ごとに 5 回ずつ実行し，その平均値を記録する．

実験環境は以下の通りである．

CPU　Intel(R) Xeon(R) E-2378 CPU @ 2.60GHz 64bits 8C/16T

メモリ　UDIMM 64GB

OS　Ubuntu 22.04

rustc　1.75.0

Actix　0.13

Erlang/OTP　26

5.2.1　PingPong

PingPong ベンチマークの結果は以下の図 5.3 のようになった．横軸は二つのプロセス間でメッセージパッシングが発生する回数（ラリー回数）を表しており，縦軸は実行時間を表している．同じラリー回数に対して高速に処理できる方が良いため，グラフは下にいくほど良い結果と言える．

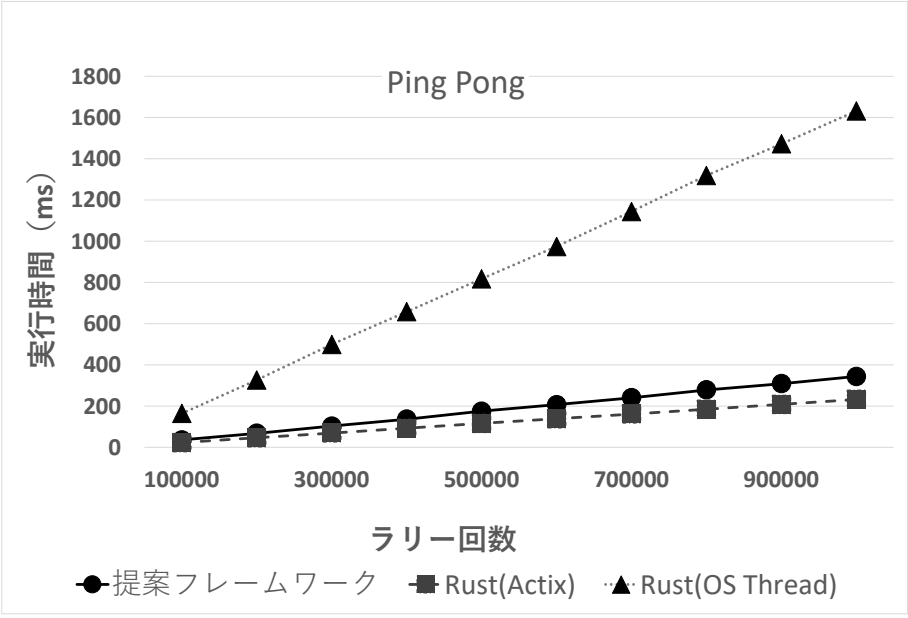

図 5.3　PingPong ベンチマーク実験結果

OS スレッドによる手法が最も遅く，次いで提案フレームワーク，Actix という結果となった．提案フレームワークによる手法は Actix と比較して，ラリー回数が 10^6 の時 100 ミリ秒程度の差が生じる結果となった．

5.2.2　Thread Ring

ThreadRing ベンチマークの結果は以下の図 5.4 のようになった．表の見方は PingPong ベンチマークと同様である．ここではワーカプロセスの数を 1000 とした．

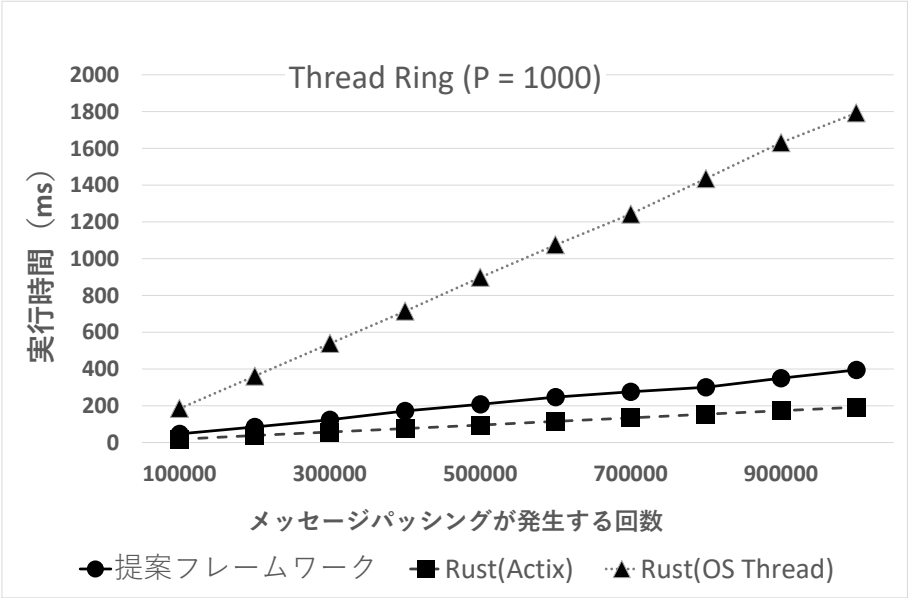

図 5.4　Thread Ring ベンチマーク実験結果

それぞれの手法同士を比較すると PingPong ベンチマークと同様の傾向が見られるが，各手法ごとに PingPong ベンチマークと比較すると，提案フレームワークと OS スレッドによる手法は PingPong ベンチマークと比較して少し実行時間が遅くなり，Actix は PingPong ベンチマークと比べて速くなっている．

5.2.3　Forkjoin (throughput)

Forkjoin (throughput) ベンチマークの結果は以下の図 5.5 のようになった．横軸は各ワーカプロセスが受信するメッセージの数を表しており，縦軸は実行時間（ミリ秒単位）である．Ping Pong ベンチマークや Thread Ring ベンチマークと同様に，高速に動作するほど良いため，グラフは下にいくほど良い．ここではワーカプロセスの数を 1000 とした．

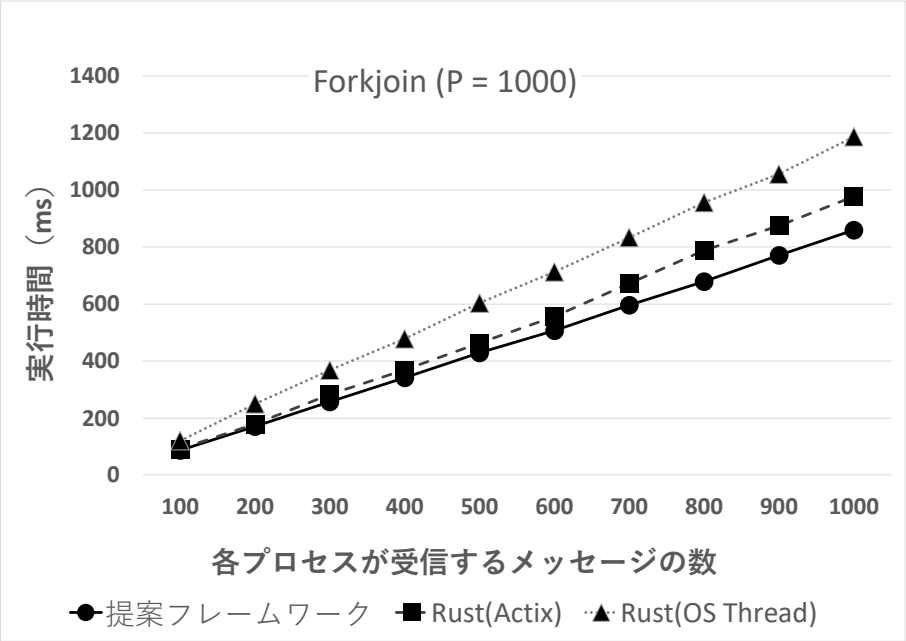

図 5.5　Forkjoin (throughput) ベンチマーク実験結果

各手法同士を比較すると，提案フレームワークが最も速く，次いで Actix，OS スレッドとなった．

5.2.4　Forkjoin (actor creation)

Forkjoin (actor creation) ベンチマークの結果は以下の図 5.6 のようになった.

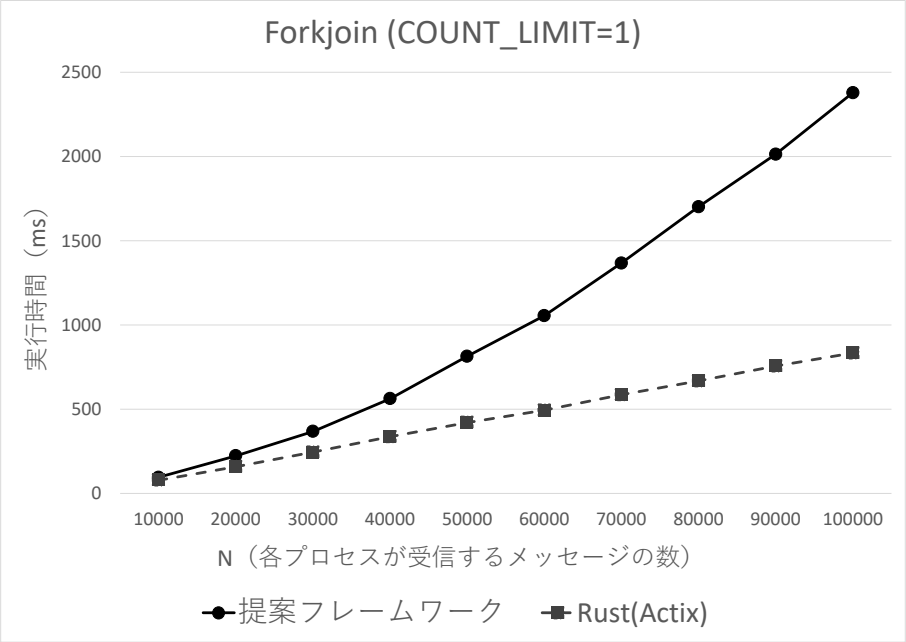

図 5.6　Forkjoin (actor creation) ベンチマーク実験結果

1×10^4 から 2×10^4 程度までは大きな差がないが，アクターが増えると提案フレームワークが Actix よりも遅くなる結果となった.

5.2.5　Big

Big ベンチマークの結果は以下の図 5.7 のようになった．横軸はワーカプロセスの数であり，縦軸は実行時間（ミリ秒単位）である．他のベンチマークと同様に高速に動作するほど良いため，グラフは下にいくほど良い.

実行時間を比較すると，提案フレームワークが最も遅く，Actix が最も速い結果となった.

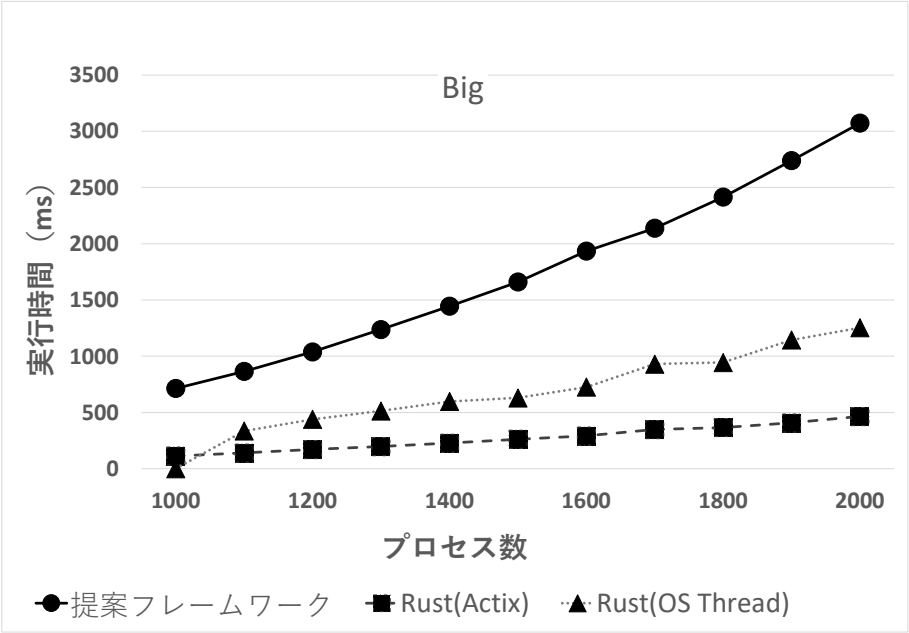

図 5.7　Big ベンチマーク実験結果

5.3　考察

Forkjoin ベンチマークでは他の二つの手法と比べて最も速い結果が出たものの，Big ベンチマークでは最も遅い結果となった．

そのため，メールボックスの競合がなければメッセージパッシングのスループットは他の二つの手法よりも高いパフォーマンスを発揮するが，メールボックスの競合が発生した場合に極端に性能が劣化するものと考えられる．

Erlang を含むいくつかの言語を対象に，メッセージパッシングの性能評価を行った研究 [19] では，Erlang のメッセージパッシングのレイテンシやスループットは高いと評価されており，このような傾向は見られない．しかし，[19] では NIF 呼び出しが伴う場合については考慮されていない．Erlang と Rust の非同期プログラミングに対して評価を行った他の研究も見つけられなかった．従って，Erlang と Rust のパフォーマンスを比較するための性能評価実験や，Erlang が NIF をロードすることや，NIF を実行しているプロセスに対するメールボックスの競合の影響などを詳細に調査するため追加の実験を行う必要がある．

第 6 章

耐障害性，および構築容易性に関する検討

本章ではそれぞれの手法について耐障害性，および構築容易性について検討する．

6.1 耐障害性

Actix は前述の通り，あるタスクがパニックを起こして終了した場合，他のタスクを道連れにしてしまうという問題がある．これを防ぐためには，ユーザが定義する各コールバック関数について，適切なエラー捕捉が必要である．Rust を用いるため，C や C++で問題になりやすいメモリエラー (ダングリングポインタやメモリリークなど) は発生しないものの，エラーが発生する可能性のある処理 (典型的には，配列の範囲外アクセスなど) については，そのエラーが発生しないように注意深くコードを書かなければならない点は C や C++と同様である．

本節では，Erlang/OTP が提供するスーパバイザの機能を Actix と提案フレームワークで取り扱う．主に，次の二つの点に焦点を当てる．

- プロセスの終了を検知して再起動させる
- 再起動戦略 one_for_all

再起動戦略 one_for_all は，スーパバイザプロセスが自身の子プロセスの終了を検知した際に，兄弟プロセスも全て再起動させる戦略を指す．

6.1.1 提案フレームワーク

提案フレームワークで子プロセスの終了を検知し，再起動させる機能を実現する際は，ワーカプロセスを起動する際にスーパバイザプロセスに起動させるだけで良い．提案フレームワークでは，プロセスを起動する方法はスーパバイザから起動することだけであるため，特別なことはしなくて良いということになる．また，再起動戦略を指定する際も，スーパバイザプロセスのコールバック関数を定義する際に，その返り値で示せば良いだけである．

以下に，その例を掲載する．このコードは，one_for_all によるスーパバイザを定義するコード片であり，重要な部分のみを抜粋している．また，参考のため，Thread Ring ベンチマークのスーパバイザプロセスの定義を付録に掲載している．

```
1  mod atoms {
2      rustler::atoms!(
3          // ...
4          one_for_all,
5          any_significant,
6      )
7  }
8
9  fn init<'a>(env: Env<'a>, _args: Term<'a>) -> Term<'a> {
10     let sup_flags = SupFlags {
11         strategy: Some(atoms::one_for_all()),
12         intensity: Some(1),
13         period: Some(5),
14         auto_shutdown: Some(atoms::any_significant()),
15     };
16
17     // ... 以下，子プロセスの定義が続く
18
19     (atoms::ok(), (sup_flags, child_specs)).encode(env)
20 }
```

プログラム 6.1　スーパバイザプロセスの定義例（一部抜粋）

このコードの atoms モジュールは，返り値に用いるアトムをあらかじめ宣言するために

用いる．

11 行目にある `strategy: Some(atoms::one_for_all())` という部分が再起動戦略を指定している部分である．他の戦略を用いたい場合は，atoms モジュールにその戦略名を表すアトム（one_for_one など）を宣言し，11 行目を変更するだけで良い．子プロセスの再起動は OTP によってその処理が定義されているため，提案フレームワークのユーザが追加でコードを書く必要はない．

6.1.2 Actix

Actix で OTP のスーパバイザと同じ機能を実現したい場合は，自力で実装するか，オープンソースのライブラリ等を探す必要がある．

Actix には Supervised トレイトが提供されており，このトレイトを実装するこ OTP が提供するスーパバイザと類似する機能を実装することは可能である．しかし組み込みの機能のみを用いる場合，各アクターを実装する際はパニックが発生しないように注意深く実装しなければならない．Rust のパニックは， `std::panic::catch_unwind` 関数などを用いて捕捉できるため，`try catch` 構文と同じ要領で，パニックが起こる可能性のある式を `std::panic::catch_unwind` でラップすることになる．

また，OTP のスーパバイザがサポートする再起動戦略も，戦略ごとにユーザが実装する必要がある．スーパバイザを介してアクターを起動する際，スーパバイザは全ての子アクターの識別子を記憶する．なお，各子アクターは自身がパニックで終了する際にも対応できるように，パニックで終了した際は自分でスーパバイザに通知する必要がある点に注意が必要である．

子アクターの終了を捕捉したスーパバイザプロセスは，その子アクター自身と，その子アクターが再起動されることによって，同じく再起動されることになる子アクターも同時に再起動する．どの子アクターを再起動するかは戦略に依存するが， one_for_all の場合は自身が監視する子プロセスを全て再起動する．

他にも，OTP が提供するスーパバイザの機能は，再起動が無限に発生しないように子プ

ロセスの再起動が一定の頻度を超えたら自身も再起動するようにしたり，ある重要な子アクターを指定し，その子アクターが終了した場合は自分も終了したりと，子アクターの監視について豊富な機能を提供している．

これらの機能を Actix で実現したい場合は，2024 年 2 月現在では，その機能がサポートされていないため実装コストが高い．

6.2　タスクの配置

Actix の場合，アクターを実装するためにコールバック関数を定義する必要がある他，そのアクターを適切に，(1 つ，もしくは複数の）スケジューラに配置しなければならない．例えば，スケジューラの個数を，システムで利用可能なコアの数とし，100 個あるタスクをこのスケジューラに均等に配置したい場合，次のようなコード片が必要となる．

なお，以下のコード片は num_cpus クレート [*1] に依存している．

```
// 起動するアクターの数
let n = 100;

// 利用可能なコア数 = c の取得
let c = num_cpus::get();

// スケジューラを c 個起動する．
let mut arbiters = (0..n)
    .map(|_| Arbiter::new())
    .collect::Vec<_>();

for i in 0..n {
    // c 個あるスケジューラのうち， i % c 番目のスケジューラに
    // i 個目のアクターを配置する
    arbiters[i % c].spawn(
        // アクターの起動 ...
    );
```

[*1] https://crates.io/crates/num_cpus

```
18 }
```

プログラム 6.2　タスクの配置を決定する Actix コード片

起動するアクターの数 n が利用可能なコア数 c で割り切れるならば，すべてのスケジューラが同じ数のアクターを管理するようになる．

提案フレームワークを用いる場合は，各アクターを実装するためにコールバック関数を定義したあとは，監視ツリーのルートとなるプロセスを起動するだけでよく，どのアクターがどのスケジューラに配置されるかは Erlang VM が決定する．したがって，上記のようなコードは一切不要である．

6.3　考察

以上のことから，Actix を利用するよりも提案フレームワークを用いる方が，耐障害性の実現やタスクの配置を決定するためのコードを書かなければならない点において，コストが低く抑えられると考える．

ただし，提案フレームワークは Erlang の基本型については知っていることを想定して設計している．Actix は Rust の基本型のみしか使われていないため，提案フレームワークの方が使用するまでの学習コストが高いと言える．また，Erlang のビヘイビアモジュールについてや，そのコールバック関数についても知っていることを想定しているため，NIF の返り値が Erlang が正しく受理することを保証するのも，現状はユーザの責任である．

このギャップを緩和するために，Erlang と Rust 間で型の整合性を保証する型システムの開発などが必要と考えられる．

また，本論文の耐障害性や構築容易性に関する検討は精密な評価ではない．より精密な評価を行うためには，実用的なアプリケーションを構築し，その中で耐障害性を評価したり，そのアプリケーションを構築する際の手間などを評価する必要がある．具体的には，関連研究として Erlang を用いて Let it Crash のパラダイムが安全性を重視するソフトウェアにも適用できるか調査した研究 [20] がある．[20] で使用されている例を用いて，Actix や提案フ

6.3 考察

レームワークを用いた場合を比較するといったものが考えられる.

第 7 章

まとめ

本論文では，Rust を用いて OTP アプリケーションを開発するためのフレームワークを提案し，Actix と OS Thread と比較した性能評価実験と，耐障害性，および構築容易性に関する検討を行った．

提案フレームワークはビルドツールと適切な NIF が実装されていることをある程度保証するためのラッパライブラリを提供する．性能評価実験の結果，Actix よりは劣化する傾向にあり，特定のベンチマークプログラム以外では概ね OS Thread より高速に動作するという結果が得られた．この結果を観察すると，メールボックスに対して競合が起こった際に大きく性能が劣化するものと考えられる．

また，耐障害性については Actix は防御的なプログラミングスタイルにならざるを得ないためエラーハンドリングを適切に行う責任がユーザにある．また，OTP が提供するスーパバイザほど高機能な子プロセス監視用のプロセスを実装するためのライブラリが存在しない．しかし，提案フレームワークを用いる場合は Let it Crash によるプログラミングが可能になるため，防御的なプログラミングが不要になる．

さらに，タスクの配置を考えなければならないといった煩雑な処理も，提案フレームワークでは不要である．

従って，耐障害性や構築容易性については提案フレームワークを用いる方が優れているものと考えられる．ただし，本論文での結論は検討段階であるため，より精密な評価が必要と考えられる．

謝辞

本研究を進めるにあたり，ご多忙の中，指導教員として数多くのご助言をいただいた高知工科大学高田 喜朗先生，及び本論文の副査を引き受けてくださいました，同大学情報学群松崎 公紀先生と同大学情報学群原田 崇司先生に，心より御礼申し上げます．

また，高田研究室所属の学生の皆様や，同期の皆様には様々な交流を通じて多くの知見を得ることができました．さらに，私がここまで健康で不自由なく研究活動に邁進できたのは家族の支えのおかげです．ありがとうございました．

参考文献

[1] Rust programming language [online]. https://www.rust-lang.org. 2024 年 1 月 25 日参照.

[2] William Bugden and Ayman Alahmar. Rust: The programming language for safety and performance. *arXiv preprint arXiv:2206.05503*, 2022.

[3] Tokio - an asynchronous rust runtime [online]. https://tokio.rs. 2024 年 1 月 25 日参照.

[4] actix/actix: Actor framework for rust [online]. https://github.com/actix/actix. 2024 年 1 月 25 日参照.

[5] Jim Blandy, Jason Orendorff, and Leonora F.S. Tindall. *Programming Rust*. O'Reilly Media, Inc., 2021.

[6] The Rust Programming Language. https://doc.rust-lang.org/book/. 2024 年 1 月 26 日参照.

[7] Erlang/otp - index. https://www.erlang.org/. 2024 年 1 月 26 日参照.

[8] Joe Armstrong. Erlang — a Survey of the Language and its Industrial Applications. In *In Proceedings of the Yymposium on Industrial Applications of Prolog (INAP96).*, Vol. 96, pp. 16–18, 1996.

[9] Frej Drejhammar and Lars Rasmusson. Beamjit: a just-in-time compiling runtime for erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pp. 61–72, 2014.

[10] Tanguy Losseau and Peter Van Roy. Numerl: efficient vector and matrix computation for Erlang.

[11] Using Rust to Scale Elixir for 11 Million Concurrent Users [online]. https://discord.com/blog/

using-rust-to-scale-elixir-for-11-million-concurrent-users. 2024年1月27日参照.

[12] rustler — crates.io : Rust package. https://crates.io/crates/rustler. 2024年1月27日参照.

[13] Carl Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[14] Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala — Akka. https://akka.io. 2024年1月30日参照.

[15] 高柳亘, 徐駿剣, 脇田建. Actorモデルにもとづいた非同期並列プログラミング言語actgpuコンパイラの実装とその評価. 情報処理学会研究報告 計算機アーキテクチャ(ARC), Vol. 2012, No. 9, 2012.

[16] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pp. 235–245, 1973.

[17] Dave Thomas. *Programming Elixir*. オーム社, 2018. 笹田 耕一, 鳥井 雪（共訳）.

[18] Shams M. Imam and Vivek Sarkar. Savina — an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, pp. 67–80, 2014.

[19] Valkov, Ivan and Chechina, Natalia and Trinder, Phil. Comparing languages for engineering server software: Erlang, Go, and Scala with Akka. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pp. 218–225, 2018.

[20] Christoph Woskowski, Mikolaj Trzeciecki, and Florian Schwedes. Robust by "let it crash". In *Safecomp 2013 FastAbstract*, p. NC, 2013.

付録 A

Thread Ring ベンチマークのスーパバイザプロセスの実装

Actix を用いる場合と OS Thread を用いる場合はスーパバイザプロセスが存在しないため，提案フレームワークによるもののみを掲載する．

48 行目以降のトレイトの実装が，スーパバイザプロセスを実装する本質的な部分である．

```
1  use otp_wrap::Supervisor;
2  use rustler::{Atom, Encoder, Env, NifMap, Term};
3
4  use crate::atoms::aggregator;
5
6  struct STRoot;
7  otp_wrap::otp_init!(supervisor, mod_name = "stroot", mod_struct = STRoot);
8
9  mod atoms {
10     rustler::atoms! {
11         ok,
12
13         one_for_one,
14         any_significant,
15
16         chain_elem,
17         start_link,
18
19         local,
20
21         aggregator,
```

```
22
23          temporary,
24          brutal_kill,
25          worker,
26          gen_server,
27      }
28  }
29
30  #[derive(Debug, NifMap)]
31  pub struct SupFlags {
32      strategy: Option<Atom>,
33      intensity: Option<usize>,
34      period: Option<usize>,
35      auto_shutdown: Option<Atom>,
36  }
37
38  #[derive(Debug, NifMap)]
39  pub struct ChildSpec<'a> {
40      id: usize,
41      start: Term<'a>,
42      restart: Atom,
43      significant: bool,
44      shutdown: Atom,
45      r#type: Atom,
46  }
47
48  impl Supervisor for STRoot {
49      fn init<'a>(env: Env<'a>, _args: Term<'a>) -> Term<'a> {
50          let sup_flags = SupFlags {
51              strategy: Some(atoms::one_for_one()),
52              intensity: Some(1),
53              period: Some(5),
54              auto_shutdown: Some(atoms::any_significant()),
55          };
56          let mut child_specs = vec![];
57
58          // 環境変数よりパラメータを設定
```

```
59          let chain_len = std::env::var("CHAIN_LENGTH")
60              .expect("please set `CHAIN_LENGTH` environment variable")
61              .parse::<usize>()
62              .expect(Failed to parse `CHAIN_LENGTH` env var to `usize`");
63
64          let count_down = std::env::var("COUNT_DOWN")
65              .expect("please set `COUNT_DOWN` environment variable")
66              .parse::<usize>()
67              .expect("Failed to parse `COUNT_DOWN` env var to `usize`");
68
69          // 終了を捕捉するためのプロセスの起動
70          child_specs.push(ChildSpec {
71              id: std::usize::MAX,
72              start: (
73                  atoms::aggregator(),
74                  atoms::start_link(),
75                  vec![
76                      (atoms::local(), aggregator()).encode(env),
77                      (chain_len, count_down).encode(env),
78                  ]
79              )
80                  .encode(env),
81              restart: atoms::temporary(),
82              significant: true,
83              shutdown: atoms::brutal_kill(),
84              r#type: atoms::worker(),
85          });
86
87          // スレッドリングを構成するプロセスを起動
88          for i in 0..chain_len {
89              let child_name = Atom::from_str(env, &format!("chain_{i}"))
90                  .unwrap();
91              let c = ChildSpec {
92                  id: i,
93                  start: (
94                      atoms::chain_elem(),
95                      atoms::start_link(),
```

```
                    vec![
                        // プロセス名を登録
                        (atoms::local(), child_name).encode(env),
                        (i, chain_len).encode(env),
                    ],
                )
                    .encode(env),
                restart: atoms::temporary(),
                significant: false,
                shutdown: atoms::brutal_kill(),
                r#type: atoms::worker(),
            };
            child_specs.push(c);
        }

        (atoms::ok(), (sup_flags, child_specs)).encode(env)
    }
}
```