学　士　論　文

題　目　Goにおけるアクターモデルの実現に向けたライブラリの設計と実装

京都大学 総合人間学部 認知情報学系

氏 名　中田健誠

Goにおけるアクターモデルの実現に向けたライブラリの設計と実装

中田健誠

Abstract

この論文ではアクターモデルをプログラミング言語 Go において実現するための新たなライブラリの提案を行う。アクターモデルは 1973 年に考案されたプログラミングモデルでありながら、昨今においても Swift の言語仕様に追加されるなど、その強みは現代においても評価されている。Go は 2009 年に発表された比較的新しい言語でありながら、規模の大きなオープンソースのプロジェクトに複数採用されているなど存在感のあるプログラミング言語である。アクターモデルの利点は並行プログラミングにおけるデッドロックやレースコンディションなどの問題を回避できる点であり、Goroutine をはじめとする並行プログラミングのサポートに注力している Go と相性が良いと考えられる。この論文ではマクロが存在しない Go において一般的な手法となっているコード生成をベースとしたアクターモデルのライブラリを提案し、実装を行う。また、Goroutine を活用し、アクターのメッセージごとに Goroutine を実行することで、アクターモデルを実現しており、スレッドが軽量な Go の特性を活用した実装になっていると言える。このライブラリは Go における既存のライブラリと比較して、メッセージングに型が付く点や、オブジェクト指向の思考でプログラムをかけるデザインになっている点、リエントランシーのサポートによりデッドロックを防いでいる点などで異なる。また、複数のホストにスケールさせた状態でアプリケーションを動作させる構成が組まれることが多い現代において有効と思われる、他のホストにおけるアクターとの通信やデータベースにおけるアクターの管理を中心としたアクターの宣言的管理や、その応用によるアクター単位でのリリースなどの手法を考案する。

目 次

第1章 はじめに

本論文では、プログラミング言語 Go [1] に向けたアクターモデルを実現するためのライブラリを提案する。

アクターモデルと呼ばれるアクターというオブジェクトを中心とするプログラミングモデルが存在する。このモデルは並行プログラミングへの強みがあり、直近においても Swift の言語機能に採用されるなど [2]、その強みは評価され続けている。アクターはそれぞれ完全に独立しており、メッセージパッシングのみでやりとりを行う。これにより、複数のスレッドが動作するような並行プログラミングにおいても、データの安全性を保ちやすいという利点がある。

Go は Google が開発し、2009 年に発表された言語であり、現在はオープンソースでの開発が続けられている [3]。Go は開発の主体となっている Google をはじめとして、多くの企業で実際にアプリケーション開発に採用されており、また、Kubernetes [4] などの規模の大きいオープンソースのプロジェクトにおける使用例もある。Goroutine と呼ばれる Go のランタイムで管理される言語レベルの軽量スレッドをサポートしており、並行プログラミングに強みを持つ。しかし、言語レベルや標準ライブラリなどにおいて、アクターモデルをベースとしたような考え方の概念は存在せず、ユーザーは適切な箇所で排他制御などを各自で実装する必要がある。とはいえ、一般的に排他制御などによる問題の解決は単純ではなく、複数のロックを管理し、複数の関数に異なる排他制御をかける必要があったり、排他制御を多くの場所で行いすぎると、デッドロックに気をつける必要が出てきたりすることが多々ある。ここでの問題点はこのような並行プログラミングにおける問題の対策をユーザーが講じているかどうかを静的に見つけられない点である。そのため、並行プログラミングにおけるこの種の問題が対策されるかどうかは完全にユーザーに委ねられていることとなる。アクターモデルの利点は、正しく使用されることによって上記の問題が解決される点であり、本論文で提案するライブラリを用いることで Go においてもユーザーが負担なくアクターモデルを使用できることを目指す。

本論文ではマクロが存在しない Go において一般的となっているコード生成によるアクターのライブラリを提案する。Goroutine を活用し、アクターのメッセー

ジごとに Goroutine を実行することで、アクターモデルを実現しており、スレッドが軽量な Go の特性を活用した実装になっていると言える。 このライブラリは Go における既存のライブラリと比較して、メッセージングに型が付く点や、リエントランシーのサポートにより並列プログラミングでない場合と類似したプログラムを書いていてもデッドロックを防げている点などで異なる。また、ユーザーが定義した interface からアクターを生成する手法を取っている。これにより、オブジェクト指向プログラミングのような使用感でアクターを扱うことができるアクティブオブジェクト指向 [5] のアクターの表現を実現することができており、この点でも既存のライブラリとはデザインが大きく異なっている。これにより、オブジェクト指向プログラミングに慣れている開発者にとって簡単にアクターモデルを用いたプログラミングをすることができ、interface などの言語機能も活用することができる。

また、ランタイムで生成されたアクターをデータベース等を通してランタイムの外から管理できるように拡張することで、アクターの宣言的な管理を実現し、その応用としてアクターレベルでのリリースなどの可能性を示す。

第2章 Go に関して

Go[1] は Google が開発し、2009 年に発表された言語であり、現在はオープンソースでの開発が続けられている [3]。強い静的型付けを特徴としている。 Goroutines と呼ばれる Go のランタイムで管理される言語レベルの軽量スレッドをサポートしており [6] 、Goroutines をうまく扱うための標準パッケージも多く提供されるなど並行プログラミングに強みを持つ。

Go の使用されるケースとしては企業におけるサーバーサイドアプリケーションの開発に使用されることが多い。また、Kubernetes[4] などの規模の大きいオープンソースのプロジェクトにおける使用例もある。

ここから先で、本論文の設計に大きく関わる Go の言語機能を大まかに説明する。以降で説明する言語機能は大きな断りがない限り、Go の v1.17 における説明であるとする。

2.1 Go におけるメソッド

本ライブラリの設計に大きく関わる “メソッド”[7] について紹介する。

Go では以下のように同一のパッケージ内で定義された任意の型にメソッドを定義することができる。(筆者書き下し)

```
1 type MyFloat float64
2
3 func (f MyFloat) Abs() float64 {
4     if f < 0 {
5         return float64(-f)
6     }
7     return float64(f)
8 }
```

この Abs() メソッドを使用するには、以下のように記述する。(筆者書き下し)

```
1 f := MyFloat(1.1)
2 result := f.Abs() // 実行
```

```
3 fmt.Println(result) // 1.1 と出力される
```

2.2 Go における interface

他の言語と同様に interface の機能が存在する [8]。以下のように定義する。(筆者書き下し)

```
1 type User interface {
2     SetAge(ctx context.Context, age int)
3     GetAge(ctx context.Context) int
4 }
```

これにより、この SetAge メソッドと GetAge メソッドを持つ全ての型はこの interface を満たしていることになる。

2.3 Goroutines とチャネル

Goroutines[6] は前述のように Go のランタイムで管理される言語レベルの軽量スレッドである。以下の構文で使用することができる。(筆者書き下し)

```
1 go f(x, y, z)
```

これによって、別の goroutine が生成され、そちらで f(x, y, z) が実行される。

また、チャネル [9] を使用して goroutine 同士で通信を行うことができる。(筆者書き下し)

```
1  func main() {
2      // 文字列を送ることができるチャネルを作成
3      ch := make(chan string)
4
5      go sendping(ch)
6
7      // チャネルを通して送られてくるメッセージを待つ。
8      msg := <-ch
9
10     fmt.Println(msg)
```

```
11 }
12
13 func sendping(ch chan string) {
14     // チャネルに"pingを送る"
15     ch <- "ping"
16 }
```

コメントにあるように、チャネルからの値の受信の際、メッセージが送られてくるまでそこでその goroutine の実行がストップする。

また、チャネルには固定長のバッファーを設けることができ、メッセージを貯めることができる。上記の例ではバッファーを設けていないため、送信の際にそのチャネルの受信待ちをしている goroutine が存在しない場合は送信の箇所で実行がストップする。

2.4 Go におけるモジュールとパッケージ

Go にはモジュールとパッケージという概念が存在している。

同じディレクトリに存在するソースコード群は基本的に同じパッケージに属する。他のパッケージの関数や構造体などを使用したい場合は import 文を使用してインポートすることになる。

仮に、以下の helloworld パッケージを ~/project/helloworld ディレクトリに作成したとする。(筆者書き下し)

```
1 package helloworld
2
3 import "fmt"
4
5 func HelloWorld() {
6     fmt.Println("hello world")
7 }
```

ここでは標準出力のために fmt パッケージという標準パッケージをインポートしている。

対して、モジュールとは一つ以上のパッケージの集まりである。例として先程の helloworld パッケージを全て GitHub でホストすることを前提に一つのモジ

ュールにしてみる。

~/project ディレクトリで以下のコマンドを実行する。

```
go mod init github.com/sanposhiho/example
go mod tidy
```

これによって、go.mod と go.sum というファイルが~/project ディレクトリに生成される。init ではモジュールの初期化が行われ、tidy ではモジュール内のパッケージを走査し、依存している外部のモジュールから自動で go.mod と go.sum の編集を行なっている。 go.mod と go.sum はモジュールの依存関係の管理などのために使用されるファイルである。

go.mod は以下のようになっている。

```
module github.com/sanposhiho/example

go 1.17
```

では、先程の helloworld パッケージを別のモジュールから呼んでみることにする。

Go では実行可能なファイルは main パッケージに記述する必要があり、main パッケージ内の main 関数が実行されることになる。 以下のような main パッケージを~/project2 ディレクトリに作成したとする。(筆者書き下し)

```
package main

import github.com/sanposhiho/example/helloworld

func main() {
    helloworld.HelloWorld()
}
```

ここでは先程の helloworld パッケージの HelloWorld 関数を呼び出している。import にはモジュール名とモジュール内のどのディレクトリにインポートしたいパッケージが存在しているかを繋げて記載している。これにより、インポートは成功するが、この main パッケージが他モジュールを読み込むためには同様にこの main パッケージもモジュールとし、依存関係を明示する必要がある。

同様に以下のコマンドを~/project2 ディレクトリで実行する。

```
go mod init github.com/sanposhiho/example2
go mod tidy
```

生成される go.mod は以下のようになっている。

```
module github.com/sanposhiho/example2

go 1.18

require (
    github.com/sanposhiho/example XXXXXXXXXXXX
)
```

helloworld パッケージが属するモジュールの情報が加わっていることが見てわかる。XXXXXX の箇所にはコミットハッシュやバージョンなどが記載される。

このようにして Go はモジュールとパッケージという仕組みによって外部のライブラリなどのインポートなどを行う。

第3章 アクターモデルに関する概要と先行研究、種類

この章ではまずアクターモデルに関する概要と先行研究について触れ、その後、広く使われているアクターモデルの実装例である Erlang と Swift のアクターに関してその基本的な使用方法を紹介し、既存のアクターのデザインを確認する。また、Go にすでに存在するアクターモデルのライブラリである protoactor-go [10]、ergo [11] や gosiris [12] について概要を説明する。

3.1 Actor model の概要と起源

アクターモデルとは 1973 年に C.Hewitt らによって書かれた論文 “A Universal Modular ACTOR Formalism for Artificial Intelligence”[13] で考案された考え方である。このアーキテクチャはアクターというオブジェクトを中心とした考え方である。論文自体はアクターを中心として、人工知能のためのモジュール式のアクターアーキテクチャと定義方法を提案するものになっている。アクターは定義された動作に従って、役割を遂行する能動的なオブジェクトであり、アクターに関係する操作は全て「アクターへのメッセージの送信」という一種類の動作で定義することとするというのがこのアーキテクチャの考え方である。

アクターにはそれぞれにキューが存在しており、送られてきたメッセージはそこに貯められていく。アクターはキューから一つずつメッセージを取り出し、処理を行っていく。これによって、一つのアクターに対して複数の動作が同時に実行されることがなく、複数のスレッドが動作するような並行プログラミングにおいても、データの安全性を保つことができるのがアクターモデルの考え方の大きな利点である。

例えば、アクターモデルを使用していない通常のプログラミングにおける場合を考える。ユーザーのオブジェクトに対して以下のような更新のメソッドが定義されている。(Go による筆者書き下しの実装例)

```
1 func (u *User) IncrementAge() {
```

```
    // 現在の歳を取得
    currentAge := u.Age

    // +1 する
    newAge := currentAge + 1

    // 新たなに更新するage
    u.Age = newAge
}
```

このメソッドは一つのスレッドから呼び出された場合は正しく動作するが、二つのスレッドから同時に呼び出された場合は規定通りに動作しない場合がある。例えばスレッド A が現在の歳を取得した時に同時にスレッド B が現在の歳を取得してしまった場合である。本来は、メソッドを二度呼び出しているため、ユーザーの歳は 2 増加する必要があるが、この場合、ユーザーの歳は二つのスレッドがアクセスする前と比べて 1 しか増加しない。

このような問題は lost-update 問題と一般的に呼ばれるものである。

このような問題を解決する方法はいくつか存在する。ここでは排他制御をかける方法を実装する。(筆者書き下し)

```
var mutex sync.Mutex
func (u *User) IncrementAge() {
    // ロックをかけることで、同時に一つのスレッドからのアクセスのみが行われるようにする
    mutex.Lock()

    // 現在の歳を取得
    currentAge := u.Age

    // +1 する
    newAge := currentAge + 1

    // 新たなに更新するage
    u.Age = newAge

    // ロックを解除する。
    mutex.Unlock()
```

```
17 }
```

この方法で lost-update 問題は確かに解決する。しかし、上記の例はものすごく単純な例であり、例えばIncrementAge 以外からもユーザーの歳が更新される場合があったりする場合は、その関数にも同様の排他制御をかける必要があったり、排他制御を多くの場所で行いすぎると、デッドロックに気をつける必要が出てきたりと、この種の問題を解決するのは実際には簡単ではない [14]。

ここでの問題点はこのような並行プログラミングにおけるバグを静的に見つけられない点である。そのため、並行プログラミングにおけるこの種の問題が対策されるかどうかはユーザーに委ねられていることとなる。

アクターモデルの利点はアクターモデルに沿ったプログラミングを行うことで上記の問題が発生しないことが保証されている点である。

アクターモデルはライブラリとして提供されている場合や言語自体に実装されている場合などがあり、それらにおいても、コンパイルや静的解析ツールなどによって、静的に開発者が正しいプログラミングをしていることを保証することで、上記の問題が発生しないことを保証することができる。

3.2 Erlang

この章では Erlang [15] がどのようにアクターモデルの考え方を導入しているのかを説明する。

Erlang は 1998 年にオープンソースとして公開された関数型言語である。Erlang は言語自体がアクターモデルをベースとした設計となっており、Erlang におけるアクターはプロセスと呼ばれる。 このプロセスは OS レベルのものとは完全に異なるものであり、Erlang のランタイムにおいて管理されている。Erlang におけるプロセスはアクターの考え方を継承しており、それぞれが完全に独立しており、メモリを共有していない。

以下に公式ドキュメント [16] より引用した Erlang におけるプログラムの例を示す。

```
1 -module(tut14).
2
3 -export([start/0, say_something/2]).
4
```

```
5  say_something(What, 0) ->
6      done;
7  say_something(What, Times) ->
8      io:format("~p~n", [What]),
9      say_something(What, Times - 1).
10
11 start() ->
12     spawn(tut14, say_something, [hello, 3]),
13     spawn(tut14, say_something, [goodbye, 3]).
```

Erlang のビルドイン関数である spawn は新しいプロセスの作成に使用される。say_something は再帰的に指定された回数だけ、指定された文言を出力する関数である。この関数を start 関数内の spawn でプロセスとして開始しており、これにより、start 関数は hello を 3 回、goodbye を 3 回出力することとなる。

次に以下の例に移る。 同様に公式ドキュメント [16] より引用している。

```
1  -module(tut15).
2
3  -export([start/0, ping/2, pong/0]).
4
5  ping(0, Pong_PID) ->
6      Pong_PID ! finished,
7      io:format("ping finished~n", []);
8
9  ping(N, Pong_PID) ->
10     Pong_PID ! {ping, self()},
11     receive
12         pong ->
13             io:format("Ping received pong~n", [])
14     end,
15     ping(N - 1, Pong_PID).
16
17 pong() ->
18     receive
19         finished ->
20             io:format("Pong finished~n", []);
21         {ping, Ping_PID} ->
```

```
22              io:format("Pong received ping~n", []),
23              Ping_PID ! pong,
24              pong()
25      end.
26
27  start() ->
28      Pong_PID = spawn(tut15, pong, []),
29      spawn(tut15, ping, [3, Pong_PID]).
```

spawn 関数はプロセスを一意に識別する識別子である PID を返却する。この PID を利用することで、!を使用してそのプロセスに対してメッセージを送ることができる。

ping 関数は指定された回数だけ送られてきた Pong_PID に対して、{ping, self()} というメッセージを送る再帰的な関数である。 指定された回数分の ping のメッセージを送り終えると、最後に Pong_PID に対して、finished というメッセージを送り、実行を終了する。

対して、pong 関数は、送られてくるメッセージによって動作が異なる。{ping, Ping_PID} という形式のメッセージが送られてきた場合は、Pong received ping という文言を出力したのちに、Ping_PID に対して、pong メッセージを送り、再帰的に pong 関数をもう一度実行する。 finished が送られてきた場合は、Pong finished という文言を出力したのちに実行を終了する。

これにより、全体的な動作としては

1 ping が Pong_PID の pong に対して ping のメッセージを送る

2 pong はそれを受け取ると、Pong received ping という文言を出力する

3 pong はその後 ping を送ってきた Ping_PID の ping に対して pong のメッセージを送りかえす

4 指定回数だけ 1-3 を繰り返す

5 ping が Pong_PID の pong に対して finished のメッセージを送る

6 ping が ping finished を出力し、終了する

となる。

3.3 Swift

ここでは Swift 5.5 で導入されたアクター [17] に関して紹介する。

Swift[2] は Apple が主体となり開発を行なっている、iOS や Mac 向けのアプリケーションを開発することを主に目的としている言語である。オープンソースとして公開されている。iOS や iPadOS、macOS で動作するアプリケーションの重要な開発言語として広く用いられている。

Swift 5.5 で Swift Concurrency[18] と呼ばれる並行処理の機能が多く追加され、アクターはその内の一つの機能として追加されるものである。

クラスを使用する際にはデータレースを避けるためにロックによる排他制御を行う必要がある場合がある。こういった同時実行におけるバグを避けるために Swift に導入された機能がアクターの機能である。 Swift のアクターは後述のリエントランシーなどの特徴により、開発者が Swift のクラスなどとほぼ差異がない使用感で使用することができるようにデザインされていることが特徴である。

このようにアクターモデルの考えとオブジェクト指向の考えを融合したものを Active Object Language(もしくは Active Object 指向)[5] と呼ぶ。

以下に Actor の提案書 [17] より引用した Swift のアクターを用いたプログラムの例を示す。

```
1  actor BankAccount {
2    let accountNumber: Int
3    var balance: Double
4
5    init(accountNumber: Int, initialDeposit: Double) {
6      self.accountNumber = accountNumber
7      self.balance = initialDeposit
8    }
```

```
1  extension BankAccount {
2    func deposit(amount: Double) async {
3      assert(amount >= 0)
4      balance = balance + amount
5    }
6  }
```

accountNumber と balance はプロパティであり、accountNumber は定数、balance は変数となっている。また、init 関数はイニシャライザである。これらはクラスと同様の記法となっている。

これらの記法はクラスと似ている部分がほとんどである。大きな違いは、アクターはその名の通り、actor model の考えをベースとしているものであり、データレースからその内部の状態を保護することである。複数のスレッドが同時に特定のアクターにリクエストしている場合も、単一のスレッドのみが一度にアクセスすることを保証する。Swift ではこの保護を総称して actor isolation と呼んでいる。

actor isolation により、クラスのオブジェクトなどとは異なり、デフォルトでは他のアクターからは同期的にプロパティの参照やメソッド呼び出しを行うことができず、両方プロパティの参照、メソッド呼び出し等は非同期に行う必要がある。ただし、アクター自身が自身のプロパティやメソッドを呼び出す際は同期的に行うことができる。

実際にアクターのメソッドの呼び出し例を以下に挙げる。ユーザーは Erlang のようにメッセージを明示的に送ってアクターにリクエストするのではなく、クラスのメソッド呼び出しと同じような感覚でアクターにリクエストするプログラムを書くことができる [17]。

```
1 await otherBankAccount.deposit(amount: amount)
```

await というキーワードが呼び出しについてはいるが、それ以外は otherBankAccount というクラスのオブジェクトに対して deposit というメソッドを呼び出す場合の文法と全く同じである。

await は非同期に呼び出しを行い、その結果を待っていることを意味しており、アクターの deposit の呼び出しが非同期に行われていることを表している。同期的に実行されるクラスのオブジェクトのメソッドの呼び出しと異なる点である。これは Swift のアクターの仕様が単一のスレッドのみが一つのアクターに同時にアクセスすることを保証しているためであり、呼び出した瞬間に同期的に処理できるとは限らないためである。

3.3.1 Sendable プロトコルに関して

Swift にはプロトコルと言う形で型のインターフェースを定義することができる機能が存在する。また、プロトコルの定義に沿ったインターフェースを型が満たすことを “準拠” という。以下に公式ドキュメント [19] のプロトコルの説明を引用する。

> A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to conform to that protocol.
>
> In addition to specifying requirements that conforming types must implement, you can extend a protocol to implement some of these requirements or to implement additional functionality that conforming types can take advantage of.

アクターのメソッドの引数と結果の型は Sendable プロトコル [20] に準拠していなければならないという規定が存在する。

Sendable プロトコルに準拠した型の値は、同時に実行されるコード間で共有してもレースコンディションなどが発生せず安全であることを意味しており、Int や String のような単純な値を意味する型などが含まれる。また、アクターは同時に実行されるコード間で安全に共有できるため、全てのアクターは Sendable プロトコルに準拠していることになる。

以下は提案書 [17] より引用した、外部から直接アクターの内部の値を変更することを試みているプログラムの例であり、実際はコンパイルに失敗する。

```
1 class Person {
2   var name: String
3   let birthDate: Date
4 }
5
6 actor BankAccount {
7   // ...
8   var owners: [Person]
```

```
9
10    func primaryOwner() -> Person? { return owners.first }
11  }
```

```
1  if let primary = await account.primaryOwner() {
2    primary.name = "The Honorable " + primary.name //
         problem: concurrent mutation of actor-isolated
         state
3  }
```

この例では Sendable ではない Person というクラスのオブジェクトを primariOwner メソッドが返している。この場合、コメントの箇所で直接アクターの中の状態を変更できてしまっていることがわかる。

アクターのメソッドの引数と結果の型が Sendable である必要があるという規定により、上記のような外部から直接アクターの内部の値の変更をすることなどを防ぐことが出来る。

3.3.2 Swift におけるリエントランシーに関して

また、Swift のアクターにはリエントランシーという性質がある。

リエントランシーは Actor-isolated 関数が他のアクターへのアクセスなどで、その返り値を待っている間は、他のメッセージの処理をアクター上で実行することができるという性質である。アクターとしてとしてはサスペンドせずに次のメッセージ処理を行う。これにより、複数のアクターがメッセージを送り合うようなプログラムにおけるデッドロックを防いでいる。

3.4 Go における既存のアクターモデルのライブラリ

ここでは Go における既存のアクターモデルのライブラリを紹介する。どれもアクターのメッセージを受信した際に型がついておらず、リエントランシーにも対応していないものである。

3.4.1 asynkron/protoactor-go

asynkron/protoactor-go[10] と呼ばれるアクターモデルを実現するための Go のライブラリがすでに存在する。ここではそのライブラリの基本的なシンタックス等を紹介する。(筆者書き下し)

```
type Hello struct{ Who string }
type HelloActor struct{}

func (state *HelloActor) Receive(context actor.Context) {
    // convert received messages
    switch msg := context.Message().(type) {
    case Hello:
        fmt.Printf("Hello %v\n", msg.Who)
    }
}

func main() {
    // Actor の作成System
    sys := actor.NewActorSystem()
    props := actor.PropsFromProducer(func() actor.Actor
        { return &HelloActor{} })
    // Actor の作成
    pid := sys.Root.Spawn(props)

    // メッセージを送る
    sys.Root.Send(pid, Hello{Who: "Taro"})

    time.Sleep(1 * time.Second)
}
```

HelloActor というアクターを前半で定義している。protoactor-go では Receive というメソッドを定義することでその中でメッセージを受け取る。メッセージを受け取ったのちに、Actor はメッセージを変換し、受け取ったメッセージの型によって振る舞いを変更することになる。HelloActor では Hello メッセージを受け取った際に、そのメッセージ内に指定された Who を取得し、“Hello {Who}” といった文章を標準出力する。

そして後半の main 関数では HelloActor を実際に実行し、動作を確認している。まず、ActorSystem というものを作成する必要がある。ActorSystem は一種のスコープであり、actor.Remote という別の仕組みを使うことで別の ActorSystem 間での通信が可能になる。同一の ActorSystem 内にいるアクターは同じ Go のプロセス内に存在することが保証されるが、別の ActorSystem 内のアクターは別の Go のプロセス内に存在する可能性がある。

ActorSystem の Root というフィールドが持つ Spawn メソッドを使用することでアクターを実際に作成することができ、帰り値としてアクターにメッセージを送る際に使用する PID を取得する。

その PID を用いてその後に Hello{Who: “Taro”} というメッセージを送信しており、このメッセージを受け取った HelloActor は “Hello Taro” という風な文章を標準出力し、このプログラムは実行を終了する。

また、ライブラリの特徴として、C# や Java/Kotlin などの実装があり、異なる言語間で実装されたアクターの通信も行うことができる。

3.4.2 ergo-services/ergo

ergo-services/ergo[11] は Erlang/OTP のデザインパターンを Go の上で再現するためのライブラリである。

以下は公式の README[11] から引用したコードの例である。

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6
7      "github.com/ergo-services/ergo"
8      "github.com/ergo-services/ergo/etf"
9      "github.com/ergo-services/ergo/gen"
10     "github.com/ergo-services/ergo/node"
11 )
12
13 // simple implementation of Server
14 type simple struct {
```

```
15      gen.Server
16  }
17
18  func (s *simple) HandleInfo(process *gen.ServerProcess,
        message etf.Term) gen.ServerStatus {
19      value := message.(int)
20      fmt.Printf("HandleInfo: %#v \n", message)
21      if value > 104 {
22          return gen.ServerStatusStop
23      }
24      // sending message with delay
25      process.SendAfter(process.Self(), value+1,
            time.Duration(1*time.Second))
26      return gen.ServerStatusOK
27  }
28
29  func main() {
30      // create a new node
31      node, _ := ergo.StartNode("node@localhost",
            "cookies", node.Options{})
32
33      // spawn a new process of gen.Server
34      process, _ := node.Spawn("gs1",
            gen.ProcessOptions{}, &simple{})
35
36      // send a message to itself
37      process.Send(process.Self(), 100)
38
39      // wait for the process termination.
40      process.Wait()
41      fmt.Println("exited")
42      node.Stop()
43  }
```

アクターはそれぞれ Erlang と同様にプロセスと呼ばれる。最初の ergo.StartNode で node を作成しており、その node から Spawn メソッドを使用してプロセスを

作っている。プロセスに対してメッセージを送るときは Send メソッドを使用しており、メッセージを受け取る simple 構造体はメッセージを受け取るとそのメッセージの型変換を行い、処理を行う。

また、特徴として Erlang の node に直接接続できるという機能がある

3.4.3 teivah/gosiris

teivah/gosiris[12] はとても基本的なアクターモデルのライブラリである。

以下は公式の README[12] から引用したコードの例である。

```
1  package main
2
3  import (
4      "gosiris/gosiris"
5  )
6
7  func main() {
8      //Init a local actor system
9      gosiris.InitActorSystem(gosiris.SystemOptions{
10         ActorSystemName: "ActorSystem",
11     })
12
13     //Create an actor
14     parentActor := gosiris.Actor{}
15     //Close an actor
16     defer parentActor.Close()
17
18     //Create an actor
19     childActor := gosiris.Actor{}
20     //Close an actor
21     defer childActor.Close()
22     //Register a reaction to event types ("message" in
           this case)
23     childActor.React("message", func(context
           gosiris.Context) {
24         context.Self.LogInfo(context, "Received %v\n",
               context.Data)
```

```
25      })
26
27      //Register an actor to the system
28      gosiris.ActorSystem().RegisterActor("parentActor",
            &parentActor, nil)
29      //Register an actor by spawning it
30      gosiris.ActorSystem().SpawnActor(&parentActor,
            "childActor", &childActor, nil)
31
32      //Retrieve actor references
33      parentActorRef, _ :=
            gosiris.ActorSystem().ActorOf("parentActor")
34      childActorRef, _ :=
            gosiris.ActorSystem().ActorOf("childActor")
35
36      //Send a message from one actor to another (from
            parentActor to childActor)
37      childActorRef.Tell(gosiris.EmptyContext, "message",
            "Hi! How are you?", parentActorRef)
38  }
```

protoactor-go や ergo のようにアクターとして構造体を定義するのではなく、gosiris.Actor という構造体に対して、React というメソッドを呼び出して、メッセージを受け取った際の振る舞いを逐次的に追加していくような使用方法になっている。

しかし最後のコミットが 2018 年であり、リポジトリがアーカイブされているため、これ以上の開発がなされる可能性は低いと見られる。

3.5 その他言語におけるアクターモデル

ここでは、その他の言語におけるアクターモデルに関して簡潔に説明する。

3.5.1 Akka

Akka[21] と呼ばれる Java、Scala 向けのアクターモデルのライブラリが存在する。以下の公式ドキュメント [22] より引用した下の例のように型付きでメッセー

ジパッシングを行うことができる点が特徴である。

```
object HelloWorld {
  final case class Greet(whom: String, replyTo:
      ActorRef[Greeted])
  final case class Greeted(whom: String, from:
      ActorRef[Greet])

  def apply(): Behavior[Greet] = Behaviors.receive {
      (context, message) =>
    context.log.info("Hello {}!", message.whom)
    message.replyTo ! Greeted(message.whom, context.self)
    Behaviors.same
  }
}
```

3.5.2 Java の synchronized によるアクターの表現

また、Java には synchronized という言語機能が存在する。こちらはクリティカルセクションであるメソッドに対して複数のスレッドからの処理を禁止し、一つのスレッドのみがそのメソッドを実行していることを保証するための機能である。(著者書き下し)

```
class HogeClass {
  synchronized void method1() {
      ...
  }

  synchronized void method2() {
      ...
  }
```

上記の例のように synchronized をのキーワードをつけたメソッドに適応される。すなわち上記の method1 と method2 はそれぞれ一つのスレッドからしか同時に呼びだされず、method1 と method2 が同時に呼び出されることもない。この synchronized を全てのメソッドに使用し、また外部から状態を変更されないように、インスタンス変数全てを private にすることでクラスをアクターかのように振る

舞わせることができる。同時に複数のスレッドから synchoronized のついたメソッドが呼び出された場合は、一つ一つ非同期に実行されていくため、そこで待ちが発生する可能性がある。これをアクターモデル的に解釈すると、メソッド呼び出しにて返信を同時に受け取っていると見なすことができる。

第4章 ライブラリの設計

本論文が提案するライブラリ Molizen は GitHub 上でオープンソースとして Apache License 2.0 の元で公開されている [23]。 本論文では v0.1.5 を参照することとする。ライブラリの規模としては v0.1.5 時点で 1500 行ほどのものとなっている。

また、バージョニングには Go が推奨するセマンティック バージョニング 2.0.0[24] を採用している。

Go のライブラリには、ツールを通してユーザーに Go のコード生成を要求するものが幾つか存在している。これによってライブラリはユーザーごとに最適な機能を柔軟に提供することが出来る。以下にその例を示す。

- golang/mock[25]: モック[1]のためのフレームワーク
- ent/ent[26]: エンティティフレームワーク (ORM)[2]
- google/wire[27]: 依存性注入 (DI, Dependency injection)[3]のためのライブラリ

Go の公式のリポジトリである golang/mock もそのような方法をとっていることからも、その手法の一般性が見て取れる。

本論文でも同様に、コード生成によりアクターの機能をユーザーに提供するコードが、最終的に生成されるライブラリの構築を行うこととする。ユーザーは interface を定義し、その後ライブラリが提供するツールからコード生成を実行する。本ライブラリが提供するツールによるコード生成により、ユーザーが定義した interface から、アクター として振る舞う構造体' が定義された Go のファイルが生成される。

ユーザーは アクターとして構造体を使用したい場合は生成されたコードの構造体を元の interface を満たす構造体の代わりに使用する。 生成されたコードには “interface の全てのメソッド” に似ているメソッドを持つ アクターの構造体が新

[1] モックとはテスト時に外部の構造体等のオブジェクトの代わりとして用いられるオブジェクトのことで、テスト対象のプログラムが外部のオブジェクトを期待する引数で期待する回数呼び出しているか等を確認するために用いられる。

[2] データモデルを管理するためのフレームワークであり、データベーススキーマをグラフ化することや、スキーマを Go のコードから定義することができるなどの機能を持つ。

[3] オブジェクトの間の依存関係の定義をそのオブジェクトの外から行うこと。

たに宣言されている。アクターのメソッドは同期的に結果を返すのではなく、同期的に Future を返し、非同期に処理を行ってその結果を Future につめるという点で interface のメソッドと異なる。 例えば以下のメソッド (著者書き下し) は

```
GetAge(ctx context.Context) int
```

アクター の構造体では以下のメソッドとして表現される。

```
GetAge(ctx context.Context) *future.Future[GetAgeResult]
```

元のメソッドが直接年齢を同期的に返すものであったのに対して、アクターのメソッドは Future を返却していることが見て取れる。そして、ユーザーはこの Future から後に実行結果を受け取ることができる。受け取りの際にまだ、処理が終わってなかった場合はその時に待ちが発生することとなる。

また、アクターの構造体では元の構造体に存在していた、フィールドにも直接アクセス出来ないようになっている。これはアクター内の情報の意図しない変更が直接行われることを避けるためである。

ユーザーはフィールドにアクセスしたい場合は、そのフィールドを変更するためのメソッドを自分で定義する必要がある。

この章ではここから具体的な使用方法や既存のライブラリとの比較を行っていく。

4.1 使用方法

Molizen は Go1.18 の新しい機能として追加されたジェネリクス [28] を使用しているため、Go 1.18 以上を必要とする。 また、この論文執筆時点では、Go の 1.18 はリリースされておらず、そのため、Go 1.18 の beta1 を使用する必要がある [29]。

Go は各バージョンを公式の Downloads のページ [30] からインストールを行うことができる。

4.1.1 コードの生成を行う

まず、ユーザーはコード生成のためのコマンドをインストールする必要がある。Go がインストールされていれば、コマンドラインから以下を実行することでイン

ストールすることができる。

```
go install
    github.com/sanposhiho/molizen/cmd/molizen@v0.1.5
```

上記によりインストールされる molizen コマンドは以下のように使用する。-source オプションを使用して、アクターの生成の元となる interface が存在するファイルを指定し、-destination オプションを使用して、アクターを生成するファイルを指定する。-destination オプションを指定しなかった場合は、標準出力に結果が出力されることになる。

```
molizen -source /path/to/source -destination
    /path/to/destination
```

例として/user/user.go に以下の interface 定義が存在するとする。(著者書き下し)

```
type User interface {
    SetAge(ctx context.Context, age int)
    GetAge(ctx context.Context) int
}
```

これを元にアクターを/actor/user.go に生成したい場合は以下の molizen コマンドを使用する。

```
molizen -source /user/user.go -destination /actor/user.go
```

そしてこの結果以下のアクターのファイルが生成される。生成されたこのコードの内容の詳細については 4.2 実装の詳細で述べることとする。

```
// Code generated by Molizen. DO NOT EDIT.

// Package actor_user is a generated Molizen package.
package actor_user

import (
    sync "sync"

    context "github.com/sanposhiho/molizen/context"
```

```
10      future "github.com/sanposhiho/molizen/future"
11  )
12
13  // UserActor is a actor of User interface.
14  type UserActor struct {
15      lock     sync.Mutex
16      internal User
17  }
18
19  type User interface {
20      SetAge(ctx context.Context, age int)
21      GetAge(ctx context.Context) int
22  }
23
24  func New(internal User) *UserActor {
25      return &UserActor{
26          internal: internal,
27      }
28  }
29
30  // GetAgeResult is the result type for GetAge.
31  type GetAgeResult struct {
32      Ret0 int
33  }
34
35  // GetAge actor base method.
36  func (a *UserActor) GetAge(ctx context.Context)
        *future.Future[GetAgeResult] {
37      newctx := ctx.NewChildContext(a, a.lock.Lock,
            a.lock.Unlock)
38
39      f := future.New[GetAgeResult](ctx.SenderLocker(),
            ctx.SenderUnlocker())
40      go func() {
41          a.lock.Lock()
42          defer a.lock.Unlock()
```

```
43
44          ret0 := a.internal.GetAge(newctx)
45
46          ret := GetAgeResult{
47              Ret0: ret0,
48          }
49
50          f.Send(ret)
51      }()
52
53      return f
54  }
55
56  // SetAgeResult is the result type for SetAge.
57  type SetAgeResult struct {
58  }
59
60  // SetAge actor base method.
61  func (a *UserActor) SetAge(ctx context.Context, age int)
        *future.Future[SetAgeResult] {
62      newctx := ctx.NewChildContext(a, a.lock.Lock,
            a.lock.Unlock)
63
64      f := future.New[SetAgeResult](ctx.SenderLocker(),
            ctx.SenderUnlocker())
65      go func() {
66          a.lock.Lock()
67          defer a.lock.Unlock()
68
69          a.internal.SetAge(newctx, age)
70
71          ret := SetAgeResult{}
72
73          f.Send(ret)
74      }()
75
```

```
76      return f
77  }
```

4.1.2 生成前の interface に関する制約

生成前の interface には アクター 同士の情報の伝達を行うために本論文の context パッケージ [11] の Context という構造体を第一引数に定義しておくという制約が存在する。この制約を満たす全ての interface を元にアクターを生成することができる。

また、Context は node という構造体から NewContext という関数を使用することで生成される。

node に関しては第 5 章今後の展望で紹介する。

4.1.3 生成されたアクターを使用する

まず、interface を満たす構造体を定義し、アクターを生成する。(著者書き下し)

```
1  actor := actor_user.New(&User{})
```

そして、そのアクターに定義されている元の interface と同様の名前のメソッドを呼ぶことで処理をアクターに依頼することができる。結果は前述のように future を通して取得する。(著者書き下し)

```
1  // 作成したアクターにをに変更するようにメッセージを送る。Age1
2  future := actor.SetAge(ctx, 1)
3  // 処理の終了を待つ。
4  future.Get()
```

以下に単純な使用例 (著者書き下し) を載せる。

```
1  func main() {
2      node := node.NewNode()
3      ctx := node.NewContext()
4
5      // 構造体からアクターを生成する。User
6      actor := actor_user.New(&User{})
7
```

```
    // 作成したアクターにをに変更するようにメッセージを送る。Age1
    future := actor.SetAge(ctx, 1)
    // 処理の終了を待つ。
    future.Get()

    // 現在のアクターのを確認するためにメッセージを送るAge
    future2 := actor.GetAge(ctx)

    // 同様に処理の終了を待つ。から結果を受け取る。future
    age := future2.Get().Ret0

    // 出力する。
    fmt.Println("Result: ", age)
}

// アクターの生成時に指定したUser を満たす構造体を定義する。interface
type User struct {
    name string
    age  int
}

func (u *User) SetAge(ctx context.Context, age int) {
    u.age = age
}

func (u *User) GetAge(ctx context.Context) int {
    return u.age
}
```

この実行結果は以下になる。SetAge の指定通りに Age が 1 に設定され、正しく Age を取得できていることが見て取れる。

```
Result:  1
```

4.2 実装の詳細

ここではライブラリの要点の実装の詳細について紹介する。

4.2.1 interface の静的解析

先程も登場した Go の公式のモックの生成ライブラリである golang/mock[25] の内部のパッケージを使用することで、ユーザーの interface から必要な情報を抽出する。

4.2.2 生成されるアクターの内部構造

アクターは以下のような内部構造を持つ構造体になっている。

```
type UserActor struct {
    lock     sync.Mutex
    internal User
}

type User interface {
    SetAge(ctx context.Context, age int)
    GetAge(ctx context.Context) int
}
```

lock に排他制御のためのロック、internal にユーザーが定義した interface を満たすオブジェクトを格納している。

4.2.3 アクターの生成のための関数

以下のように New 関数が生成されているのでそれを用いてアクターを生成する。前述のように interface を満たすオブジェクトを渡す必要がある。

```
func New(internal User) *UserActor {
    return &UserActor{
        internal: internal,
    }
}
```

4.2.4　生成されるアクターのメソッドの実装について

interface のメソッド一つにつき、アクターのメソッドと結果が格納される構造体が生成される。

```
1  // GetAgeResult is the result type for GetAge.
2  type GetAgeResult struct {
3      Ret0 int
4  }
5
6  // GetAge actor base method.
7  func (a *UserActor) GetAge(ctx context.Context)
       *future.Future[GetAgeResult] {
8      newctx := ctx.NewChildContext(a, a.lock.Lock,
           a.lock.Unlock)
9
10     f := future.New[GetAgeResult](ctx.SenderLocker(),
           ctx.SenderUnlocker())
11     go func() {
12         a.lock.Lock()
13         defer a.lock.Unlock()
14
15         ret0 := a.internal.GetAge(newctx)
16
17         ret := GetAgeResult{
18             Ret0: ret0,
19         }
20
21         f.Send(ret)
22     }()
23
24     return f
25 }
```

メソッドは以下の処理を同期的に行う。

・自身の情報を格納した本論文の context パッケージ [11] の Context を新たに生成する。

・結果を返却するための future を生成し、それを返却する。

そして、以下の処理は Goroutine を用いて非同期的に行う。

・自身の他のメソッドが同時に実行されないようにロックをする。

・内部にもつ初期化時に渡された構造体の GetAge メソッドを呼び出す。

・結果を GetAgeResult に格納する。

・Future の Send 関数を呼び出すことで GetAgeResult を Future に送信する。

アクターは自身のメソッドが同時に実行されることをロックによる排他制御で防いでいる。

4.2.5 リエントランシーについて

Swift と同様に本論文のライブラリでもリエントランシーを採用している。3.3.2 Swift におけるリエントランシーに関してで説明した Swift と同様の理由でデッドロックを防ぐためである。

リエントランシーの必要性の説明のため、以下の例 (著者書き下し) を考える。以下の User interface を元にアクターを生成する。

```
type User interface {
    // 自身の名前を返却する
    Name(ctx context.Context) string
    // to に対してを送るPing
    SendPing(ctx context.Context, to
        *actor_user.UserActor)
    // "Hello (from の名前)" を出力し、に対してを送り、が処理されると実行を終了するfromPongPong
    Ping(ctx context.Context, from *actor_user.UserActor)
    // "ponged" と出力する
    Pong(ctx context.Context)
    SetSelf(ctx context.Context, self
        *actor_user.UserActor)
}
```

そのアクターに対して以下のプログラム (著者書き下し) を実行する。

```
func main() {
    node := node.NewNode()
    ctx := node.NewContext()
```

```
4
5       // アクターを二つ生成する。
6       actor1 := actor_user.New(&User{name: "taro"})
7       actor2 := actor_user.New(&User{name: "hanako"})
8
9       future := actor1.SetSelf(ctx, actor1)
10      future.Get()
11      future2 := actor2.SetSelf(ctx, actor2)
12      future2.Get()
13
14      // actor1 に対してにを送るように依頼する。actor2Ping
15      future3 := actor1.SendPing(ctx, actor2)
16
17      future3.Get()
18  }
```

これは以下のように実行されることを期待している。

・SendPing のメッセージを受け取った actor1 は actor2 に対して Ping を送る。

・actor2 は actor1 に対して Name を聞き出し、“Hello (actor1 の名前)” を出力。

・ actor2 は actor1 に対して Pong を送る。

・ Pong を受け取った actor1 は “ponged” を出力。

・ actor2 は actor1 が Pong の処理を終えたことを確認し、Ping の処理を終了する。

・Ping の処理を終えたことを確認した actor1 は SendPing の処理を終了する。

・ future3.Get() の待ちが終了し、プログラムの実行が終了する。

この実行には actor1 と actor2 がお互いにメッセージを送り合うことが必要とされる。

この際、仮にリエントランシーが無効の場合、actor1 は SendPing の処理を全て終えるまで他の処理を実行しない。そのため、actor2 が actor1 の名前を Name メソッドを通して聞き出そうとしたタイミングでデッドロックが発生する。actor1 は actor2 の Ping の処理の終了を待ち続け、actor2 は actor1 の Name の処理を待ち続けるためだ。

リエントランシー が有効にしておくことで、actor1 が actor2 の Ping の処理を待っている間に Name の処理を行えるようになり、この種のデッドロックを防ぐ

ことができる。

このリエントランシーの実現には Future という構造体が大きな役割を果たしている。次節で説明を行う。

4.2.6 Future について

Future は以下のような構造体として定義されている。

```
type Future[T any] struct {
    ch chan T
    result *T
    senderLocker *senderLocker
}

type senderLocker struct {
    mu     sync.Mutex
    locker       func()
    unlocker     func()
    isLockedByUs bool
}
```

内部にはチャネルと呼ばれる型のオブジェクトを持っている。また、senderLocker という構造体に送信者のロックを扱うための関数や状態を管理するフィールドを保持している。

また、初期化のための New 関数は以下のように定義されている。locker と unlocker の引数が渡された際にのみ senderLocker を初期化して、フィールドに格納している。これによって、送信者がアクターではなかった場合、送信者のロックに関する処理をしないようになっている。

```
func New[T any](
    locker func(),
    unlocker func(),
) *Future[T] {
    var sl *senderLocker
    if locker != nil && unlocker != nil {
        sl = &senderLocker{
            locker:       locker,
```

```
            unlocker:     unlocker,
            isLockedByUs: true,
        }
    }
    return &Future[T]{
        ch: make(chan T, 1),
        senderLocker: sl,
    }
}
```

そして、メソッドとして Send と Get が定義されている。

```
func (f *Future[T]) Send(val T) {
    f.ch <- val
}

func (f *Future[T]) Get() T {
    if f.result == nil {
        result := f.get()
        f.result = &result
    }
    return *f.result
}

func (f *Future[T]) get() T {
    for  {
        select {
        case result := <-f.ch:
            f.lockSender()
            return result
        default:
            f.unlockSender()
        }
    }
}

func (f *Future[T]) unlockSender() {
```

```
26      if !f.hasSender() {
27          return
28      }
29
30      f.senderLocker.mu.Lock()
31      defer f.senderLocker.mu.Unlock()
32
33      if f.senderLocker.isLockedByUs {
34          f.senderLocker.unlocker()
35          f.senderLocker.isLockedByUs = false
36          return
37      }
38  }
39
40  func (f *Future[T]) lockSender() {
41      if !f.hasSender() {
42          return
43      }
44
45      f.senderLocker.mu.Lock()
46      defer f.senderLocker.mu.Unlock()
47
48      if !f.senderLocker.isLockedByUs {
49          f.senderLocker.locker()
50          f.senderLocker.isLockedByUs = true
51          return
52      }
53  }
```

Send は内部に持つチャネルに対して、値の送信を行っているのみのシンプルなメソッドである。

Get は大筋の流れとしては、チャネルから結果を所得し、取得できた結果を result という自身のフィールドに格納して、結果を返却するということを行っている。一度受信した結果を保存する処理が入っていることによりこれにより、何度でも Future の Get メソッドから結果を受け取ることができる。

また、Get メソッドには lockSender と unlockSender というメソッドが途中で

実行されている。これらは送信者のロックを操作するためのメソッドであり、この部分が Actor リエントランシーの根幹となる処理である。

4.1.2 生成前の interface に関する制約にて生成前の interface には アクター同士の情報の伝達を行うために本論文の context パッケージ [11] の Context という構造体を第一引数に定義しておくという制約が存在するということを説明した。本論文の context パッケージの Context にはメッセージの送信者の情報が格納されている。アクターのメソッド内で Future の生成される箇所をもう一度見てみる。

```
1 // GetAge actor base method.
2 func (a *UserActor) GetAge(ctx context.Context)
      *future.Future[GetAgeResult] {
3     newctx := ctx.NewChildContext(a, a.lock.Lock,
          a.lock.Unlock)
4
5     f := future.New[GetAgeResult](ctx.SenderLocker(),
          ctx.SenderUnlocker())
6
7 // 以下略...()
```

context から SenderLocker と SenderUnlocker というメソッドが呼び出され、その結果が Future の作成に使用されている。この二つのメソッドはそれぞれリクエストを送ってきたアクターが内部にもつロックをロックする関数とアンロックする関数を返却する。Future は初期化時にこれらの関数を受け取ることで、リクエストを送ってきたアクターが内部に持っているロックの操作を行うことができる。

future の Get というメソッドは結果の受信をチャネルで行うが、まだアクターが処理を終えていなかった場合は処理が終わるまで待ちが発生する。get メソッドでは初めにチャネルから結果を取得できなかった場合に unlockSender を呼び出し送信者のアクターのロックを解除している。

これによって、アクター A がとある処理の途中でアクター B を呼び出す際に、アクター B の処理を future の Get メソッドを用いて待つ間アクター A の他の処理がブロッキングされることを防いでいる。

そして、チャネルから処理の結果を受け取った後に lockSender メソッドを用いて送信者のアクターのロックを再度かけている。

これらのことから、Future はチャネルによって処理の結果が来るのを待っている間、その待ちの間送信者のロックを解除することで送信者に他の処理をすることを許可し、処理の結果が来た際に、再度送信者のロックをすることで、“処理の結果待ちの間だけ送信者に別の処理を行うことを許可する”Actor リエントランシーを実現しているのである。

FutureGroup について

FutureGroup という Future をまとめて扱うためのものを提供している。これによって、以下の例 (著者書き下し) のようにユーザーは全ての future の実行の終了を一箇所で待つことができる。

```
1  g :=
       group.NewFutureGroup[actor_user.IncrementAgeResult]()
2  for i := 0; i < 100; i++ {
3      future := actor.IncrementAge(ctx)
4      // をkey i としてを登録する。future
5      g.Register(future, strconv.Itoa(i))
6  }
7
8  // に入れられた全てのの実行を待つFutureGroupFuture
9  g.Wait()
10
11 // から結果を取り出すFutureGroup
12 // 上記のループでforiの時に格納された=1の結果を取り出すfuture
13 g.Get("1")
```

4.3 既存のアクターモデルのライブラリとの比較

3.4 Go における既存のアクターモデルのライブラリで紹介した、protoactor-go[10] や ergo[11]、gosiris[12] との比較を行う。

ここではそのライブラリと本論文のライブラリの設計や思想の違いを比較し、どのような点が異なり、どのような点で本論文のライブラリが強みを取れるかを述べる。

4.3.1　デザインの方向性について

本論文のライブラリは前の章で挙げた幾つかのアクターモデルの例でいくとSwiftに近く、Active Object 指向といえる。ユーザーは通常通りメソッドを定義し、メッセージパッシングに関しても通常通り構造体のメソッドを呼び出すだけでよく、内部的によしなに処理が行われ、その構造体はアクターとして振る舞うことになる。

しかし protoactor-go や ergo は Erlang に近い。ユーザーはアクターモデルを明確に意識する必要があり、メッセージを明示的に送信する事でアクター同士のコミュニケーションを行う。

まず、protoactor-go や Erlang の採用する手法に関して。こちらのメリットはシンプルさである。ユーザーがアクターモデルを理解している場合、こちらの方が直感的であり、アクターモデルを意識しやすい。

対して、本論文のライブラリや Swift が採用する Active Object 指向について。まずわかりやすいメリットとしては、オブジェクト指向プログラミングに慣れている開発者にとってのハードルの低さである。Swift も本論文のライブラリも既存の機能を拡張してアクターを導入することを目指したものである。

Swift に関しても元々、アクターを使わずに Swift でプログラミングをしていた人たちがいて、アクターモデルに馴染みがない人達にとってのハードルを抑え、いかに使いやすいアクターの形を目指すかを考えられているように感じる。ユーザーはメソッドを通常通り定義し、呼び出しも通常通り行うことでアクターモデルの恩恵を受けることができる。

また、Active Object 指向のメリットはこうした開発者体験だけでなく、interfaceなどをはじめとする既存の言語機能の恩恵を大きく受けることができる、という点がある。

interface はオブジェクト指向においてなくてはならない機能である。アクターにおいても、interface を使用することができることで、同じメソッドを持つアクターを用途に応じて入れ替えることが可能になるなど、interface による恩恵を最大限に受けることができることは、大きな利点となる。

4.3.2　型について

protoactor-go などではメッセージに型がつかないというデメリットがある。以下は protoactor-go を使用したアクターの実装例 (著者書き下し) である。

```
type Hello struct{ Who string }
type HelloV2 struct{ Who string }
type HelloActor struct{}

func (state *HelloActor) Receive(context actor.Context) {
    // convert received messages
    switch msg := context.Message().(type) {
    case Hello:
        fmt.Printf("Hello %v\n", msg.Who)
    }
}

func main() {
    // create actor
    sys := actor.NewActorSystem()
    context := actor.NewRootContext(sys, nil,
        opentracing.SenderMiddleware())
    props := actor.PropsFromProducer(func() actor.Actor
        { return &HelloActor{} })
    pid := context.Spawn(props)

    // 正しく処理されるメッセージ
    context.Send(pid, Hello{Who: "Roger"})

    // 正しく処理されないメッセージ
    context.Send(pid, HelloV2{Who: "Roger"})

    time.Sleep(1 * time.Second)
}
```

switch msg := context.Message().(type) {} の部分で context.Message() で取得できるメッセージの型の変換を行っている。context.Message() 自体は interface{} 型を返す。interface{} 型というのはどの型でも当てはまる所謂 any 型のようなものであるために、ユーザーはメッセージが Hello 型か否かを確認し、Hello 型だった場合変換を行う必要がある。

このメッセージの型の変換を行う必要があることは、ユーザーにとって面倒が増えるという点もあるが、ユーザーが正しい型のメッセージを送っていることを静的に確認することができないという点が問題である。現状は SetBehaviourActor に対して Hello 型のみを受け付けているが、ユーザーは Hello 型以外の任意の型を送ることができ、コンパイラはその誤りを検知しない。上記の例でいくと HelloV2 型を送っていることにユーザーは気がつくことができない。同様のメッセージの型の問題は ergo や gosiris においても存在する。

本論文のライブラリはファイルの生成を行うことで、ユーザーの使用する関数に適したアクターを生成する。この手法の大きなメリットは型を含む静的型付けの恩恵を受けることができる点である。その点で、本論文のライブラリは優位であると考えることができる。

4.4 アクターモデルを用いずに記述した場合との比較

ここではアクターモデルを使用せず、ユーザーが自身で排他制御を行う場合と、本論文のライブラリを使用する場合の並行処理の実装の比較を行う。

以下の interface を例に、初めに SetAge で 0 歳に設定したのちに、100 回並行に IncrementAge を呼びだし、最後に GetAge で 100 歳になっていることを確認するプログラムを記述することにする。

```
type User interface {
    SetAge(ctx context.Context, age int)
    IncrementAge(ctx context.Context)
    GetAge(ctx context.Context) int
}
```

アクターを使用しない場合は以下のように記述することになる。

```
func main() {
    node := node.NewNode()
    ctx := node.NewContext()
    user := User{}

    user.SetAge(ctx, 0)

    wg := sync.WaitGroup{}
```

```
9       for i := 0; i < 100; i++ {
10          wg.Add(1)
11          go func() {
12              defer wg.Done()
13              user.IncrementAge(ctx)
14          }()
15      }
16
17      wg.Wait()
18
19      age := user.GetAge(ctx)
20      fmt.Println("[using struct] Result: ", age)
21  }
22
23  // 前述のUser を満たす構造体interface
24  type User struct {
25      name string
26      age  int
27      lock sync.Mutex
28  }
29
30  func (u *User) SetAge(ctx context.Context, age int) {
31      u.age = age
32  }
33
34  // IncrementAge increment user's age.
35  func (u *User) IncrementAge(ctx context.Context) {
36      // 排他制御のためにロックをかける。
37      u.lock.Lock()
38
39      age := u.age
40      u.age = age + 1
41
42      u.lock.Unlock()
43  }
44
```

```
func (u *User) GetAge(ctx context.Context) int {
    return u.age
}
```

アクターを使用する場合は以下のように記述することになる。前述の FutureGroup を使用してまとめて処理を actor に依頼している。

```
func main() {
    node := node.NewNode()
    ctx := node.NewContext()
    actor := actor_user.New(&User{})
    future := actor.SetAge(ctx, 0)
    // wait to set age
    future.Get()

    g := group.NewFutureGroup[actor_user.IncrementAgeResult]()
    for i := 0; i < 100; i++ {
        future := actor.IncrementAge(ctx)
        g.Register(future, strconv.Itoa(i))
    }

    g.Wait()

    future2 := actor.GetAge(ctx)
    fmt.Println("[using actor] Result: ", future2.Get().Ret0)
}

// 前述のUser を満たす構造体interface
type User struct {
    name string
    age  int
}

func (u *User) SetAge(ctx context.Context, age int) {
    u.age = age
```

```
29 }
30
31 // IncrementAge increment user's age.
32 func (u *User) IncrementAge(ctx context.Context) {
33     age := u.age
34     u.age = age + 1
35 }
36
37 func (u *User) GetAge(ctx context.Context) int {
38     return u.age
39 }
```

書き方の違いはあるものの、main 関数にはコード量や複雑さはそれほど違いがないように見える。

着目すべき点は、User 構造体に定義された、IncrementAge メソッドである。

IncrementAge の本質の処理である「年齢を取得して 1 を足したものを登録し直す」という処理がスレッドセーフ[4]ではない。そのため、アクターを使用していない場合、IncrementAge では排他制御のためにロックをかけている。アクターは特性上、同時に IncrementAge の処理が実行されることがないため、排他制御を行う必要がなく、上記の例でも特別に排他制御を行っていないが正しく動作する。

仮に、アクターを使用していない例でロックをかけなかった場合は lost-update 問題によって正しく動作しない場合がある。

このロックの必要、不必要の違いはコード量だけで考えると些細な違いであるように思えるが、アクターを使用しない場合、正しいプログラムを書くにはユーザーは「IncrementAge がスレッドセーフではないため、ロックをかける必要がある」と認識する必要があることが大きく異なる。また、この排他制御が行われていなかったとしても、コンパイラは正常に動作するため、ユーザーは問題を静的に発見することができない。

また、この User 構造体に他にロックをかける必要があるメソッドが追加され、様々な箇所で排他制御が行われることになった場合、デッドロックを避けた正しい排他制御の記述もユーザーに求められる事になる。

このようにこの例に限らず、正しい並行処理のプログラムを書くためには、ユーザーは多くのことを意識する必要があり、また、それらは静的に発見することが

[4] スレッドセーフとは複数のスレッドから呼び出されても安全であることを指す。

難しい特性を持っているのである。

本論文のライブラリを使用する場合は、ライブラリを正しく使用してさえすれば、このような排他制御をユーザーに任せる必要がなく、プログラムがユーザーが期待する通りに、デッドロックやレースコンディションなく動作することが保証される。

・排他制御が必要な箇所で排他制御を行っているか

・排他制御がデッドロックなどを避ける形で正しく行われているか

・そもそもプログラムのどの点で排他制御が必要なのか

は静的に判断することができないため、この並行プログラミングの一般的なミスを人の手以外で発見することは難しいが、

・本論文のライブラリを正しく使用しているかどうか

は静的に判断できることから、本論文のライブラリはユーザーの並行プログラミングの正しい記述に貢献すると言える。

第5章 今後の展望

v0.1.5 時点でサポートしている機能はアクターの基本的な機能ばかりである。この章では今後に追加すべき機能や考えられる展望について述べる。

5.1 Non-reentrant アクター

4.2.5 リエントランシーについてで述べたように現状本論文のライブラリではリエントランシーが有効なアクターが生成される。しかし、ユーザーによってはリエントランシーが不要である、ない方が好ましいと言ったケースも存在すると考えられる。また、このようなリエントランシーを無効にすることに関しては Swift のアクターにも議論があり、提案書にも将来の展望として記載されている。[17]

本論文の場合、現状は一種類のアクターのみがツールから生成されるようになっている。引数などを変更することで、ここに記載している Non-reentrant アクターを含む複数のアクターの種類の中から、ユーザーが自身の目的に則したアクターを生成できるようにすることを考えている。

ただし、Non-reentrant アクターはメッセージを送り合うようなコードを書いた場合にデッドロックを起こす可能性がある点に留意する必要がある。このデッドロックについてはメッセージの処理中にそのメッセージをリクエストしてきたアクターに対してメッセージを送った場合に発生する。例えば、Non-reentrant アクター A, B, C が存在する場合、A が B に、B が C にそして C が A にメッセージを送っていると'non-reentrant アクターであることが原因のデッドロック' を起こすことになる。

どのアクターがどのアクターにメッセージを送信するかというのを静的に決定するのは難しい。しかし、ランタイムで動的にどのアクターがどのアクターにメッセージを送ったかを記録することは可能であるため、ランタイムにより'Non-reentrant アクターであることが原因のデッドロック' が起こったことは見つけることができると考えられる。これによって、ユーザーに'Non-reentrant アクターであることが原因のデッドロック' が起こった際に呼び出す関数などを設定させておくことで、ユーザーはデットロックが発生した場合の動作を自由に設定できるように

なり、一つの助けになると考えられる。例えば、ログを記録している外部のサービスにエラーログを送信する、プログラム全体を終了する、アクターを再起動させるなどの動作をさせることが考えられる。

5.2 障害の伝搬

現状本論文のライブラリにおけるアクターにおいて、例外が発生すると通常の Go アプリケーションと同様に Go プログラムを実行しているプロセス全体が終了する。アクターの初期化時に、例外の処理に関する動作を指定できるオプションの追加を考えている。

また、Erlang などではアクターに別のアクターを監視させるために便利な仕組みが用意されており [31]、それにより、障害が発生した場合の再起動などを任せることができるようになっている。

しかし、Go ではエラーは例外としてではなく関数の返り値として返されることが多く、公式でも多くの場合ではそちらの方法でエラーを伝えることが良いとしているため [32]、例外による障害の監視は難しいと考える。そのため、特定の返り値をエラーとして関数から返すことで、障害が発生したことをライブラリ側から検知できるような仕組みの実装も考えられる。

5.3 内部状態へのアクセスの流出を防ぐ静的解析ツール

現状ではもしユーザーがアクター内部の状態にアクセスできるポインターを返却するメソッドを定義していた場合、それを用いてユーザーはアクターの内部の状態に同期的にアクセスできてしまう。

例えば以下の例の User interface をもとにしてアクターを生成したとする。

```
1 type User interface {
2     FetchNamePointer() *string
3 }
4
5 type userImpl struct {
6     Name string
7 }
8
```

```
9  func (u *userImpl) FetchNamePointer() *string {
10     return &u.Name
11 }
```

この場合、FetchNamePointer からアクターの内部の構造体である userImpl の Name フィールドへのポインターを得ることができてしまう。このようなコードは静的解析によって発見することが可能であるため、そのような静的解析ツールを提供することで、ユーザーに注意を促すことができる。

5.4 ActorRepo と ActorLet

本論文の context パッケージ [11] の Context はアクターにメッセージを送信するために必要な構造体である。

現状この Context 構造体を生成するためには node という構造体を生成する必要がある。この node は一つの Go アプリケーションの動作するホストの情報を表す構造体として定義されており、これによって、ユーザーは現状どのホストでアプリケーションが動作しているのかをライブラリに知らせることになる。すなわち、一つの Go アプリケーションにつき node は一つ生成されるということとなる。

また、node はそういったホストに関する情報のほか、アクターの起動を管理する ActorLet と、全てのアクターの情報を管理する ActorRepo を内部に持っている。現状では二つとも空の構造体として実装されているため、何も働きはしないが、ここでは将来的にどのような働きを持つのかを説明する。

5.4.1 ActorRepo を用いた複数ホスト上のアクターの管理

まず、将来的にアクターは生成される際に Context を通して、ActorRepo へ登録されるようになる。 この ActorRepo はそのホスト上のアクターのみではなく、別のホスト上で動作するアクターの情報も保持しており、それにより、別ホストに存在するアクターへの通信を行うことができる。

複数のホストを使用する場合は全ての ActorRepo が同じ情報を参照する方法がある。そこで、一つのデータベースを用いて全てのホストのアクターの情報を管理する。それぞれの node の ActorRepo はアクターの情報を常にデータベースに

取得しにいく。また、自身の node でアクターの作成が行われた場合にはその情報をデータベースに登録する。ActorRepo のデータベースに関しては多くの種類をサポートすることでユーザーが自身のニーズやシステム要件にあった ActorRepo を使用できるようになると期待される。

5.4.2 ActorLet による宣言的アクター管理と他のホストに存在するアクターの作成

ActorRepo に登録されるアクターの情報というのはアクターの “理想の状態” を示しており、将来的に ActorLet は常に ActorRepo を監視して、その理想の状態に近づけるような振る舞いをするように実装される。これにより、ActorRepo に新たにアクターを登録することで ActorLet にそのアクターを作成させることができる。

例えば、node1 と node2 が存在するとする。node1 が node2 でアクターを起動したい場合は、ActorRepo に node2 で起動したい Actor を登録する。 ActorLet は定期的に ActorRepo をチェックし、自身の node に割り当てられたアクターの ActorRepo 上での状態を監視している。node2 に存在する ActorLet は ActorRepo に新たに node2 で起動すべきアクターが登録されたことに気がついたタイミングで、そのアクターを node2 で起動する。

このような理想の状態を宣言し、システムはその理想の状態に近づけるように常にループにて状態を制御し続けるという方法は後述の Kubernetes におけるリソースの考え方と同じである [33]。このようなループは Kubernetes では Reconciliation loop と呼ばれる。Reconciliation loop は理想の状態と現在の状態を取得し、その差を調べ、そこから理想の状態に近づくように状態の変更を行う。

例えばとある node にアクターが現状 3 つ存在し、アクターの数を 4 つに変えたい場合を考える。この時、node へ直接新たなアクターを作ることをリクエストしたとすると、そのリクエストが仮にネットワーク障害にて届かなかった場合 node で動作するアクターは 3 つのままである。

しかし、Reconciliation loop を用いた状態管理を採用することで、ActorLet は自律的に理想の状態を取得しに行き、現在のアクターの数が 3 つであるが、理想の状態は 4 つであることに気がつき、新たにアクターを一つ作成することでその差を埋めようと動作することになる。同様に、ネットワークの障害で ActorLet が

理想の状態を取得することに失敗したとする。その場合障害が続く間は node で動作するアクターは 3 つのままとなってしまう。しかし、ネットワーク障害が明けたのちに、ActorLet は理想の状態と現在の状態の差に気がつき、アクターの数を増やす。

これらのことから node へ直接アクターの作成をリクエストする方法と比較して、Reconciliation loop を用いた宣言的な状態管理は障害やその復旧に強いことがわかる。

5.5 他のホストに存在するアクターへのメッセージング

Erlang は他のホストなどに存在するアクターとの通信をサポートしている。ユーザーはどのホストにアクターが存在しているのかを意識せずに、プログラミングをすることができる。しかし、現状本論文のライブラリではそのような他のホストに存在するアクターへのメッセージングはサポートしていない。

現代において、大きなトラフィックを受けるようなアプリケーションは複数のホストにスケーリングされている場合が多い。この場合、アプリケーションの前段に存在するロードバランサーなどにより、設定をもとにして複数のホストにリクエストが転送される。

仮に特定のユーザーの情報を管理している User Actor というアクターが存在するとし、ホスト 1 上で User A の情報を管理するための User Actor が生成されたとする。すると、その後は、他のホストには User A の情報を管理するためのアクターが存在しないため、ホスト 1 で User A の情報を使用するような全てのリクエストが処理される必要がある。 API の設計によってはホスト 1 で User A の情報を使用するような処理を全て行うような設定をロードバランサーにすることは可能かもしれない。しかし、実際には User Actor 以外にも多くの種類のアクターが同時に動作してアプリケーションが動作することが予想される。そのため、全ての種類のアクターのことを考え、特定のホストで特定のアクターが関わる全て処理が実行されるような設定をするというのは現実的ではない。

これらのことから本論文のライブラリでも将来的に同様に他のホスト上のアクターとの透過的な通信をサポートすることが必要である。内部的に、前述の ActorRepo を通して通信したい Actor がどこのホストに存在するかを取得することで、通信を行うことができる。

通信には独自のシリアライズは採用せず、Google がオープンソースで公開しているシリアライズフォーマットである Protocol Buffers[34] を採用する。Protocol Buffers は一般的なシリアライズフォーマットであるため、多くの言語ですでにサポートされており、これを使用することで将来的に異なる言語で実装されたアクター間での通信を行う際にシリアライズの部分の実装を行う必要がなくなるメリットがある。

5.6 Kubernetes 上におけるアクターの状態表現とそれによる外部からのアクターの宣言的管理

ここでは Kubernetes を利用している Go のアプリケーションに本論文のライブラリを使用した場合に、アクターを管理する方法に関して構想を述べる。

Kubernetes[4] とは Google が 2014 年にオープンソースとして公開した、コンテナ化されたワークロードやサービスを宣言的に管理するためのプラットフォームである。

Google Cloud Platform や Amazon Web Service などの大手のクラウドサービス内で、Kubernetes の使用を前提とした専用のサービスが存在する [35][36] など、宣言的なインフラ管理が主流となりつつある、現代においてかなり市民権を得ている。

Kubernetes では多くのインフラリソースが抽象化されている。“Pod” は一つ以上のコンテナとそれらのコンテナの共有リソースを抽象化したリソースであり、“Node” はマシンを抽象化したリソース (仮想マシン、物理マシンのどちらでも良い) を表している。[37]

“ホスト上でコンテナを動作させる” ことは、Kubernetes 上で “Node 上で Pod を動作させる” という風に表現される。

5.6.1 Kubernetes におけるカスタムリソース

Kubernetes には前述のように標準で多くのリソースが登録されている。それに加えて、ユーザーが自身のニーズにあったリソースを登録できる (カスタムリソースと呼ばれる) 機能 [38] が存在する。

アクターをカスタムリソース “Actor” として定義することで、宣言的にアクター

の状態を Kubernetes 上のリソースとして管理することができるようになる。また、アクターが動作するホストを “ActorNode” として定義する (Node という名前のリソースはすでに存在するため “ActorNode” という名前にしている)。

そして、“ActorKind” というアクターごとの種類に関するリソースも追加する。すると、UserActor という “ActorKind” のアクターがある場合に、User A、User B のアクターを生成すると、それぞれが “Actor” リソースとして登録されるというふうに、ActorKind と Actor は 1 対多の関係となる。

それぞれ例えば以下のような定義になると考えられる。

```
kind: Actor
metadata:
    name: "User A"
spec:
    podName: "pod1"
    actorNodeName: "actor-node1"
    actorKind: "UserActor"
    version: "v1.1"
```

```
kind: ActorNode
metadata:
    name: "actor-node1"
spec:
    podName: "pod1"
```

```
kind: ActorKind
metadata:
    name: "UserActor"
spec:
    versions:
        - name: "v1.1"
          image: "hoge:v1.1"
        - name: "v1.0"
          image: "hoge:v1.0"
```

ActorKind はバージョンでアクターのコードの変更を管理する。そのアクターのコードに変更が加えられた際は、バージョンを一つインクリメントする。バー

ジョンごとに、どのコンテナイメージにそのアクターが存在するのかが記載されている。

アクターはアプリケーション内、すなわちコンテナ内で動作するものである。一つの Pod に一つの Actor を含むコンテナしか存在しない場合、一つの Pod に一つの ActorNode と複数の Actor が存在するという形になる。

前述のように ActorRepo が全てのアクターの情報を保持している。

ActorRepo のデータベースとして多くの種類をサポートすることでニーズにあったデータベースをユーザーが使用できるようになるというふうに述べた。ここで述べているカスタムリソースの機能を通したアクターの状態管理は、Kubernetes の api server を ActorRepo として使用するということを示している。

すなわち、Kubernetes の上でアプリケーションが動作している場合、アクターの新規作成が行われた場合には ActorRepo が Kubernetes 上のリソースとしてアクターを登録することとなる。

5.6.2 アクターのホットスワップ

前述のように Kubernetes のリソースとしてアクターを定義することで、Kubernetes の世界からもアクターの管理を行うことができるようになり、大きなメリットが考えられる。

Erlang ではアプリケーション全体を止めずに特定のアクター (Erlang 上のプロセス) を入れ替える機能をサポートしている。 Erlang におけるホットスワップはユーザーが Erlang が提供するシェル内でユーザーが操作することで行う。

本論文のライブラリでもこれをライブラリ単体でサポートできると良いが、Go にはアプリケーション全体を終了せずに、特定の Goroutine やオブジェクトを入れ替えるような機能は存在しない。

現時点でこの機能は既存の他のエコシステムと組み合わせないと実現することは難しいと考えている。そこでここでは前述のようなカスタムリソースを使用する前提のもと Kubernetes の使用時の擬似的なホットスワップ機能のアイデアを説明する。

ユーザーは UserActor のコード上に変更を入れたとし、その変更で User A アクターのみを新たな UserActor に変更したいとする。

まず、ユーザーは変更後のコードからコンテナイメージを作成し、Docker レジストリに格納する。 変更前のコンテナイメージを v1.0、変更後のコンテナイメー

ジを v1.1 とする。

Kubernetes 上では現在 v1.0 のコンテナイメージを使用する Pod が動作している。全ての関連するリソースの状態は以下である。

```
apiVersion: v1
kind: Pod
metadata:
    name: pod1
spec:
    containers:
    - name: app
      image: "hoge:v1.0"
```

```
kind: ActorKind
metadata:
    name: "UserActor"
spec:
    versions:
        - name: "v1.0"
          image: "hoge:v1.0"
```

```
kind: Actor
metadata:
    name: "User A"
spec:
    podName: "pod1"
    actorNodeName: "actor-node1"
    actorKind: "UserActor"
    version: "v1.0"
```

```
kind: ActorNode
metadata:
    name: "actor-node1"
spec:
    podName: "pod1"
```

ユーザーはそこに新しい v1.1 のコンテナイメージを使用する Pod を追加する。そして、同時に UserActor に対応する ActorKind リソースにバージョン v1.1 を追加する。

```
apiVersion: v1
kind: Pod
metadata:
    name: pod2
spec:
    containers:
    - name: app
      image: "hoge:v1.1"
```

```
kind: ActorKind
metadata:
    name: "UserActor"
spec:
    versions:
        - name: "v1.1"
          image: "hoge:v1.1"
        - name: "v1.0"
          image: "hoge:v1.0"
```

起動した新しい Pod は node 構造体を Go アプリケーション内で作成する。Kubernetes 上で動いている場合はそのタイミングで ActorNode として登録される。

```
kind: ActorNode
metadata:
    name: "actor-node2"
spec:
    podName: "pod2"
```

その後、ユーザーは User A の状態を変更する。

```
kind: Actor
metadata:
    name: "User A"
spec:
```

```
5       podName: "pod2"
6       actorNodeName: "actor-node1"
7       actorKind: "UserActor"
8       version: "v1.1"
```

User A アクターの変更に気がついた actor-node1 の ActorLet は User A のアクターを停止する。そして、同様に変更に気がついた actor-node2 の ActorLet は User A のアクターを作成する。actor-node2 には新しい UserActor がリリースされているため、User A アクターは新たな UserActor として生成される。

User A アクターの移動時に User A への処理が溜まっていた場合を考慮して、そのような移動の前に準備時間を設けるという拡張も考えられる。例えば以下のような API として導入されうるであろう。

```
1  kind: Actor
2  metadata:
3      name: "User A"
4  spec:
5      podName: "pod2"
6      actorNodeName: "actor-node1"
7      actorKind: "UserActor"
8      version: "v1.1"
9      terminationGracePeriodSeconds: 500ms
```

これにより、actor-node1 上の User A アクターは 500ms の既存のメッセージを処理する時間をもらってから、actor-node2 で起動された新しい User A アクターに完全に切り替わることになる。

この例では、User A アクターに関しては一定のダウンタイムが発生する可能性があるものの、システム全体は終了せずにアクターを入れ替えることができることがわかった。この例でユーザーが行う必要がある Actor リソースの状態の変更などはその全てが自動化できるため、ユーザーにとっての負担にはならない。

ここではアクターの入れ替えを例にした。似たような方法で、アクターのカナリアリリース[1]なども実現できると考えられる。現在はコンテナ単位、すなわち Kubernetes のリソース上では Pod 単位のリリースを行うのが普通であったが、この

[1] カナリアリリースとは新機能をリリースする際に、一部のリクエストに対してのみその新機能が有効になるようにリリースし、問題が発生しないかを確かめながら段階的にリリースをしていく手法のことを指す。

拡張を行うことで、ユーザーはアクター単位でのリリースを行うことができるようになり、リリースの安全性やリリース頻度の向上に大きく寄与することが期待される。

また、ここでは Kubernetes の使用を前提とする例を挙げた。同様にして ActorRepo へライブラリ以外から直接アクセスしやすいようなツールを作成することで、Kubernetes を使用せずとも似たような動作が実現できる可能性がある。

第6章 まとめ

本論文ではアクターモデルをベースとしたライブラリの提案とそのベースとなる実装を行い、実現の可能性を実証した。

通常では静的に発見することが難しい並行処理のバグや排他制御に伴うデッドロック等の問題を、アクターを使用できるライブラリを使用することで、ライブラリを正しく使用することによって並行処理によるデッドロックやレースコンディションなどの問題が発生しないことを担保できる。5.3 内部状態へのアクセスの流出を防ぐ静的解析ツールに書いたようにアクターの内部情報へのポインターを流出させるなどのライブラリの使用に際して禁止されているような実装を静的に発見することができる静的解析ツールも開発が可能であるため、静的に並行処理の安全性が担保できるようになったと言える。

実装はコード生成や Goroutine を活用して、アクターモデルを実現しており、スレッドが軽量な Go の特性を活用した実装になっていると言えるであろう。 protoactor-go などの既存のライブラリと比較し、メッセージングに型が付く点は大きな利点となる。また、Swift にも見られるリエントランシーのサポートを行うことで、非常に起こりやすいデッドロックの問題を防ぐことができている。また、アクティブオブジェクト指向よりのデザインによって、オブジェクト指向プログラミングに慣れている開発者にとって簡単にアクターモデルを用いたプログラミングをすることができる点も他のライブラリと比べて異なる点である。

また、実装には至らなかった 5.6 Kubernetes 上におけるアクターの状態表現とそれによる外部からのアクターの宣言的管理にて挙げたような機能の追加を行うことで、アプリケーション内にとどまることなく、アプリケーションの外からもアクターの管理を行うことができるようになり、リリースに関わる安全性の向上にも寄与する可能性を示した。

本論文で提案したライブラリ Molizen は、今後さらに実装を改善していき、ユーザーにこれらの価値を提供できるようにすることを考えている。

第7章 謝辞

本プロジェクトに関して指導をいただいた指導教員の櫻川貴司准教授をはじめ、有益なコメントをいただいた総合人間学部認知情報学系の全ての方に感謝いたします。

参 考 文 献

[1] The go programming language, (https://go.dev/), (Accessed on 2022-01-02).

[2] Swift - apple developer, (https://developer.apple.com/swift/), (Accessed on 2022-01-02).

[3] golang/go: The go programming language, (https://github.com/golang/go), (Accessed on 2022-01-02).

[4] kubernetes/kubernetes: Production-grade container scheduling and management, (https://github.com/kubernetes/kubernetes), (Accessed on 2022-01-02).

[5] Frank De Boer, Vlad Serbanescu, ReinerHähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, Albert Mingkun Yang, A survey of active object languages, *ACM Computing Surveys*, (2018).

[6] A tour of go; goroutines, (https://go.dev/tour/concurrency/1), (Accessed on 2022-01-02).

[7] A tour of go; methods, (https://go.dev/tour/methods/1), (Accessed on 2022-01-02).

[8] A tour of go; interfaces, (https://go.dev/tour/methods/9), (Accessed on 2022-01-02).

[9] A tour of go; channels, (https://go.dev/tour/concurrency/2), (Accessed on 2022-01-02).

[10] asynkron/protoactor-go: Proto actor - ultra fast distributed actors for go, c and java/kotlin, (https://github.com/asynkron/protoactor-go), (Accessed on 2022-01-02).

[11] ergo-services/ergo: a framework for creating microservices using technologies and design patterns of erlang/otp in golang, (https://github.com/ergo-services/ergo), (Accessed on 2022-01-13).

[12] teivah/gosiris: An actor framework for go, (https://github.com/teivah/gosiris), (Accessed on 2022-01-13).

[13] R. Steiger C. Hewitt, P. Bishop, A universal modular actor formalism for artificial intelligence, *IJCAI 20*, (1973).

[14] 土居範久, 相互排除問題——「際どい資源」をいかにプログラムで利用するか, (岩波書店, 2011).

[15] Index - erlang/otp, (https://www.erlang.org/), (Accessed on 2022-01-02).

[16] Erlang – concurrent programming, (https://www.erlang.org/doc/getting_started/conc_prog.html), (Accessed on 2022-01-02).

[17] swift-evolution/0306-actors.md at 23405a18e3ebbe69fcb37b0d316aa4ec5a7b6c46 · apple/swift-evolution, (https://github.com/apple/swift-evolution/blob/23405a18e3ebbe69fcb37b0d316aa4ec5a7b6c46/proposals/0306-actors.md), (Accessed on 2022-01-02).

[18] Concurrency the swift programming language (swift 5.5), (https://docs.swift.org/swift-book/LanguageGuide/Concurrency.html), (Accessed on 2022-01-02).

[19] Protocols the swift programming language (swift 5.5), (https://docs.swift.org/swift-book/LanguageGuide/Protocols.html), (Accessed on 2022-01-02).

[20] swift-evolution/0302-concurrent-value-and-concurrent-closures.md at 23405a18e3ebbe69fcb37b0d316aa4ec5a7b6c46 · apple/swift-evolution, (https://github.com/apple/swift-evolution/blob/23405a18e3ebbe69fcb37b0d316aa4ec5a7b6c46/proposals/0302-concurrent-value-and-concurrent-closures.md), (Accessed on 2022-01-02).

[21] Akka; build concurrent, distributed, and resilient message-driven applications for java and scala — akka, (https://akka.io/), (Accessed on 2022-01-13).

[22] Introduction to actors; akka documentation, (https://doc.akka.io/docs/akka/current/typed/actors.html), (Accessed on 2022-01-13).

[23] sanposhiho/molizen: Molizen is a typed actor framework for go, (https://github.com/sanposhiho/molizen), (Accessed on 2022-01-02).

[24] セマンティック バージョニング 2.0.0 — semantic versioning, (https://semver.org/lang/ja/), (Accessed on 2022-01-02).

[25] golang/mock: Gomock is a mocking framework for the go programming language, (https://github.com/golang/mock), (Accessed on 2022-01-02).

[26] ent/ent: An entity framework for go, (https://github.com/ent/ent), (Accessed on 2022-01-02).

[27] google/wire: Compile-time dependency injection for go, (https://github.com/google/wire), (Accessed on 2022-01-02).

[28] Robert Griesemer Ian Lance Taylor, Type parameters proposal, (https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md, 2022), (Accessed on 2022-01-02).

[29] Russ Cox, Go 1.18 beta 1 is available, with generics - the go programming language, (https://go.dev/blog/go1.18beta1, 2022), (Accessed on 2022-01-02).

[30] Downloads go1.18beta1 - the go programming language, (https://go.dev/dl/#go1.18beta1), (Accessed on 2022-01-02).

[31] Erlang – processes 12.8 error handling, (https://www.erlang.org/doc/reference_manual/processes.html#error-handling), (Accessed on 2022-01-02).

[32] Effective go panic - the go programming language, (https://go.dev/doc/effective_go#panic), (Accessed on 2022-01-02).

[33] Controllers — kubernetes, (https://kubernetes.io/docs/concepts/architecture/controller/), (Accessed on 2022-01-02).

[34] Protocol buffers — google developers, (https://developers.google.com/protocol-buffers), (Accessed on 2022-01-02).

[35] Kubernetes - google kubernetes engine(gke) — google cloud, (https://cloud.google.com/kubernetes-engine), (Accessed on 2022-01-02).

[36] Managed kubernetes service - amazon eks - amazon web services, (https://aws.amazon.com/eks/), (Accessed on 2022-01-02).

[37] Viewing pods and nodes — kubernetes, (https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/), (Accessed on 2022-01-02).

[38] Custom resources — kubernetes, (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/), (Accessed on 2022-01-02).