

本書は、小社刊の以下の刊行物をもとに、大幅に加筆と修正を行い書籍化したものです。

・[WEB+DB PRESS]Vol.127 特集2「作って学ぶPhoenix——Elixirによる高速なWeb開発！」

本書で利用しているソフトウェアは、執筆時点の最新である次のバージョンで動作確認を行っています。

- ・ Elixir 1.15.7-otp-26
- ・ Erlang/OTP 26.1.2
- ・ Phoenix 1.7.10
- ・ Ecto 3.11.1
- ・ Phoenix LiveView 0.20.1
- ・ Nx 0.6.4
- ・ Axon 0.6.0
- ・ Nerves 1.10.4

環境や時期により、手順・画面・動作結果などが異なる可能性があります。

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた運用は、必ずお客様自身の責任と判断によって行ってください。これらの情報の運用結果について、技術評論社および著者はいかなる責任も負いません。

本書の情報は第1刷発行時点のものを記載していますので、ご利用時には変更されている場合もあります。

本書に記載されている会社名・製品名は、一般に各社の登録商標または商標です。本書中では、™、©、®マークなどは表示していません。

上記をご承諾いただいたうえで、本書をご利用願います。これらの注意事項をお読みいただくずにお問い合わせいただいても、著者・出版社は対処しかねます。あらかじめ、ご承知おきください。

はじめに

本書は、プログラミング言語 Elixir について、基本的な文法や機能についてひととおり解説するとともに、Web アプリケーションはもとより、機械学習や IoT といった領域における実践的な応用についても解説します。そのため、Elixir のことをあまりよく知らない方々にはもちろんのこと、すでに Elixir に馴染みのある方々も含む幅広い読者の皆様にお楽しみいただける書籍になっていると自負します。

プログラミング言語 Elixir

本書が紹介するプログラミング言語 Elixir^{注1} は、José Valim (ジョゼ・ヴァリム) 氏を中心とするコミュニティによって活発に開発されている関数型言語です。2011 年に開発が開始され、2012 年に 0.5.0 として公開された^{注2}後、2014 年にバージョン 1.0 がリリースされた^{注3}比較的若い言語です。本書の執筆時点では、バージョン 1.15.7 が最新の安定版としてリリースされています^{注4}。

Elixir は、低遅延で高い可用性を要求される分散システムの構築と運用を目的とする Erlang VM (Virtual Machine、仮想マシン) 上で動作する言語です。その文法は Ruby から大きく影響を受けており、多くの Web 開発者にとって馴染みやすいでしょう^{注5}。

Elixir の評判

プログラミング言語としての Elixir そのものについては、本書の第 1 章から第 6 章までで解説します。ここでは、Elixir を利用している人々の観点から、現状を概観してみましよう。

■ 百花繚乱のプログラミング言語

ご存じのとおり、世の中にはたくさんのプログラミング言語が存在します。Web アプリケーション開発においては、Ruby や PHP、Java といった言語が広く使われてきました。昨今では、Web フロントエンドのみならずバックエンド

注 1 <https://elixir-lang.org/>

注 2 <https://elixir-lang.org/blog/2012/05/25/elixir-v0-5-0-released/>

注 3 <https://elixir-lang.org/blog/2014/09/18/elixir-v1-0-0-released/>

注 4 本書で説明する内容も、バージョン 1.15.7 を前提にしています。

注 5 作者の José Valim 氏は、Elixir を作るうえで影響を受けた言語のトップ 3 は Erlang、Ruby、Clojure であると述べています (<https://cognitect.com/cognicast/120>)。Elixir は、堅牢な Erlang VM の上で動く、Ruby のように親しみやすい、JVM (*Java Virtual Machine*、Java 仮想マシン) に対する Clojure のような言語であると言えるでしょう。

の開発においても、TypeScriptの躍進が目立つところです。また、機械学習・AI領域においては、関連ライブラリやエコシステムが充実しているPythonが広く使われています。さらに、アプリケーションやシステムの複雑化に伴い静的型付けの意義があらためて見直され、GoやRustといった言語の人気が増えます高まっています。

■ 書籍に見る Elixir の人気度

プログラミング言語の人気度をはかるのは難しいですが、どれほど多くの書籍が出版されているかを指標の一つとしてとらえることができるでしょう。TypeScript、Python、Go、Rustについて、ここ数年、とりわけ多くの書籍が刊行されているように思われます。Elixirについてはどうでしょうか？ 2020年以降、日本語の書籍としては『プログラミング Elixir (第2版)』^{注6}、『Elixir実践ガイド』^{注7}が刊行されました。刊行点数としてははっけして多いわけではありませんが、着実に裾野を広げつつあることは確かでしょう。本書が、そうした流れを後押しすることを、筆者一同願ってやみません。

■ 国外における Elixir の状況

国外の状況はどうでしょうか？ 前述の『プログラミング Elixir (第2版)』の著者Dave Thomas氏が運営するPragmatic Bookshelf^{注8}を中心に、Elixirという名前を冠する書籍が多数刊行されています^{注9}。また、書籍ではありませんが、プログラミングに関する質問・回答を通じてナレッジを共有するWebサイトStack Overflow^{注10}が2022年に開発者に対して実施した調査では、Elixirが「最も愛されている (most loved)」言語として2位にランクイン^{注11}したことが話題となりました。Elixirが、国外の開発者の間で一定の地歩を占めているのは確かであるようです。

■ 国内の盛り上がりを支えるコミュニティ

国内においてはまだまだメジャーな言語とはいえないElixirですが、コミュニティの盛り上がりはすでにメジャー級であると言えます。その証拠に、

注6 Dave Thomas 著、笹田耕一・鳥井雪訳『プログラミング Elixir (第2版)』オーム社、2020年

注7 黒田努著『Elixir 実践ガイド』インプレス、2021年

注8 <https://pragprog.com/>

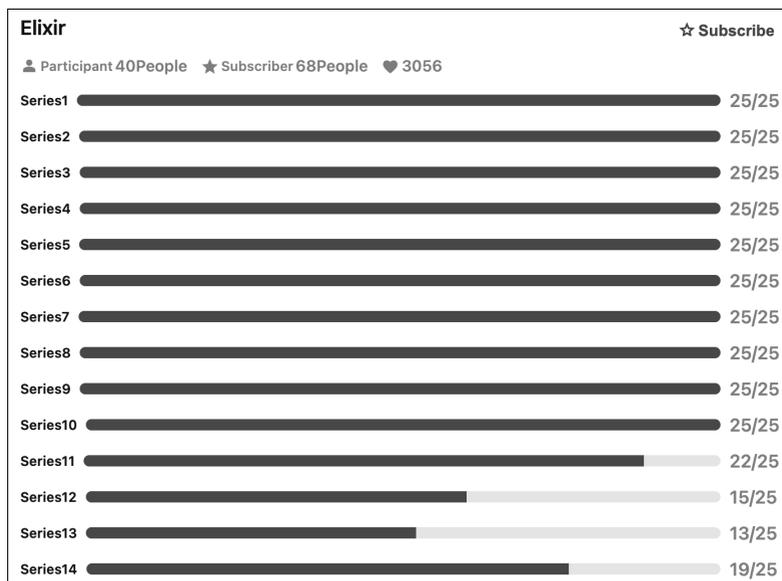
注9 Pragmatic BookshelfのWebサイトから「Elixir」で検索してみてください。
<https://pragprog.com/search/?q=Elixir>

注10 <https://ja.stackoverflow.com/>

注11 <https://survey.stackoverflow.co/2022/#technology-most-loved-dreaded-and-wanted>

本書の執筆陣も多く参加したElixir Advent Calendar 2023^{注12}では、他言語と比較しても圧倒的なボリュームの記事数が投稿されました(図1)。

● 図1 Qiita Advent Calendar 2023のプログラミング言語カテゴリ^{注13}におけるElixirに関する記事の投稿数



そうした精力的な活動を支えているのが、国内に30ほど存在するとされるElixirコミュニティです^{注14}。冒頭で述べたとおり、Elixir自体がコミュニティによって開発されている言語であるのと同様に、利用者もたくさんのコミュニティを通じてつながり、活発に情報交換しています。読者の皆様にも、ぜひご参加することをお勧めします。

■ Elixir のポテンシャル

なぜElixirがそんなにも熱狂的な盛り上がりを見せているのでしょうか？ その一つとして、Elixirやその基盤であるErlang/OTPの持つ大きなポテンシャルが、人々を強く惹きつけているからという理由が挙げられるでしょう。

注 12 <https://qiita.com/advent-calendar/2023/elixir>

注 13 https://qiita.com/advent-calendar/2023/categories/programming_languages

注 14 どんなコミュニティがあるかについては、「Elixir コミュニティの歩き方～国内オンライン編～」(<https://speakerdeck.com/elijo/elixirkomiyunitei-falsebu-kifang-guo-nei-onrainbian>)をご覧ください。

■ Elixir の利用事例

Elixir や Erlang/OTP の近年における大規模な利用事例としては、国内ではゲーム業界における事例が目立つようです^{注15}。また、チャットプラットフォームの Discord でも、Elixir が大規模に利用されているようです^{注16}。大量の接続とトラフィックを扱う必要のあるシステムを構築するのに、Erlang VM が向いているからでしょう。また、本書でも詳しく解説する Web アプリケーションフレームワーク Phoenix Framework^{注17} が早くから登場したことから、Web アプリケーション領域においても利用が進んでいます。

■ 広がりを見せる適用領域

深層学習の登場以降、機械学習・AI の著しい発展が世の中を騒がせ続けています。2021 年 2 月にリリースされた Nx^{注18} を皮切りに、それらの領域における Elixir の適用可能性が切り開かれてきました。Nx は、機械学習に必須となる高速な数値計算のためのライブラリです。今では、ニューラルネットワークを簡潔に表現できる Axon^{注19} や、事前学習済みモデルを簡単に利用できる Bumblebee^{注20} といったライブラリもあります。また、Elixir のコードをインタラクティブに記述・実行できる Livebook^{注21} も活発に開発されています。

主にサーバ上で動作する上述のような領域以外でも、Internet of Things (IoT) での Elixir の利用も進んでいます。Elixir による IoT デバイス向けのプラットフォームの Nerves^{注22} を用いると、Raspberry Pi シリーズや BeagleBone シリーズによって堅牢かつ高機能な IoT デバイスを開発できます。また、Elixir Desktop^{注23} や LiveView Native^{注24} のように、PC やモバイル端末上で動作する

注 15 ・「ロマサガ RS における Elixir サーバー開発実践～生産性を上げてゲームの面白さに注力～」
<https://speakerdeck.com/elixirfest/romasakars-niokeru-elixir-sahakai-fa-shi-jian-sheng-chan-xing-woshang-ketekemufalsemian-bai-sanizhu-li>
・「Erlang/OTP と ejabberd を活用した Nintendo Switch(TM) 向けプッシュ通知システム「NPNS」の開発事例」
<https://speakerdeck.com/elixirfest/otp-to-ejabberd-wohuo-yong-sita-nintendo-switch-tm-xiang-ke-hutusiyutong-zhi-sisutemu-npns-false-kai-fa-shi-li>

注 16 「Real time communication at scale with Elixir at Discord」
<https://elixir-lang.org/blog/2020/10/08/real-time-communication-at-scale-with-elixir-at-discord/>

注 17 <https://www.phoenixframework.org/>

注 18 <https://github.com/elixir-nx/nx>

注 19 <https://github.com/elixir-nx/axon>

注 20 <https://github.com/elixir-nx/bumblebee>

注 21 <https://livebook.dev/>

注 22 <https://nerves-project.org/>

注 23 <https://github.com/elixir-desktop/desktop>

注 24 <https://native.live/>

アプリケーションをElixirによって開発できるようになってきています。さらには、すでに触れたElixirのネットワークやWebアプリケーションにおける強みを活かして、デバイスからクラウドまでを一気通貫にElixirを用いて開発する取り組みも進んでいます^{注25}。

本書の構成

Elixirの多岐にわたる魅力をできるだけ幅広く紹介するために、本書の構成もこだわり抜きました。

まず、第1章から第6章にかけて、Elixirを楽しむためには最低限これだけは知っておいてほしいという内容にしばって、簡潔に文法や機能を紹介します。次に、第7章から第11章にかけて、ElixirでWebアプリケーションを開発してみることでさっそくElixirに「実践入門」していきます。その過程を通じて、ElixirによるWebアプリケーション開発に欠かせない、WebアプリケーションフレームワークのPhoenix Framework（と、目玉機能のLiveView）、RDBMS（Relational DataBase Management System）を扱うライブラリのEctoの使い方を学びます。さらに、第12章から第19章にかけて、機械学習・AI領域およびIoT領域において、具体的なアプリケーションを作っていきます。

このプロセスを通じて、熱狂的な盛り上がりを生み出しているElixirの大きなポテンシャルを、きっと感じていただけることと思います。読者の皆様がElixirの持つ広大な可能性に触れる一助に本書なることを、筆者一同願っています。

2024年1月

筆者を代表して 栗林 健太郎

● サポートページ

本書のサンプルコードは、下記サポートサイトからダウンロードできます。正誤情報なども掲載します。

<https://gihyo.jp/book/2024/978-4-297-14014-4/support>

注 25 たとえば、「Embedded and cloud Elixir for grid-management at Sparkmeter」(<https://elixir-lang.org/blog/2023/03/09/embedded-and-cloud-elixir-at-sparkmeter/>) のような事例があります。

謝 辞

本書の執筆にあたって、森正和さん (@piacere_ex)、衣川亮太さん (@pojiro3) (括弧内はXアカウント名) にレビューをご担当いただきました。単なる用語や解説のレビューにとどまらず、読者にとってわかりにくいところや改善案などを忌憚なくコメントいただき、本書をより良い内容にすることができました。ありがとうございます。また、本書の編集は稲尾尚徳さん、池田大樹さんに担当いただきました。7名という筆者の原稿や意見をまとめていくのは非常に大変だったと思います。根気よく付き合ってくださいまして、誠にありがとうございます。ただし、本書に記載した内容について誤りがある場合、その文責は執筆を担当した筆者らにあります。

やはり本書は、プログラミング言語 Elixir に関わる多くの開発者たちがいてこそ完成しました。本書で取り上げるフレームワークやライブラリのそれぞれの代表者として、Elixir の作者である José Valim 氏、Phoenix の Chris McCord 氏、Nx と Axon の Sean Moriarity 氏、Nerves の Frank Hunleth 氏に深く感謝申し上げます。彼らの愛と情熱に溢れたフレンドリーなリーダーシップがあったからこそ、比較的若いにもかかわらず爆発的に成長を遂げている Elixir の現在のエコシステムがあります。そして、Elixir はやはりプログラミング言語 Erlang なくしてはこの世に生まれることはなかったでしょう。Erlang および Erlang VM の共同設計者であり長らく開発を牽引していた Joe Armstrong 氏は、残念ながら 2019 年にこの世を去っています。この場をお借りして同氏への哀悼の意を示すとともに、すばらしいプログラミング言語を遺してくれたことに深く御礼申し上げます。

国内の盛り上がりを支えるコミュニティを牽引するそれぞれのオーガナイザー、そしてそれぞれのイベントを盛り上げてくれている参加者の皆さまにも感謝いたします。第 19 章で取り上げている IoT アプリケーションのサンプルは、NervesJP で実施したハンズオンイベントの内容をおおいに参考にしています。この教材を作成されたのは、著者らのほかに、衣川亮太さん (@pojiro3)、三宅泰宏さん (@etcinitd)、菊池豊さん (@kikyuta)、西内一馬さん (@nishuichikazuma)、大崎充博さん (@rigutter) の協力によります。ありがとうございます。最後に、Elixir の成長に貢献している世界中のすべてのアルケミスト^{注1}に深く感謝申し上げます。

注 1 Elixir とは、錬金術で用いられる不老不死をもたらしてくれる霊薬を意味します。そのため、プログラミング言語 Elixir の使い手はアルケミスト（錬金術師）と呼ばれることがあります。

執筆者と担当章

章	タイトル	著者名
はじめに		栗林 健太郎
第 1 章	Elixir 小史	大原 常德
第 2 章	Elixir の基礎	大聖寺谷 一樹
第 3 章	基本的な型とパターンマッチ	大聖寺谷 一樹
第 4 章	モジュール	山内 修
第 5 章	Mix を使った Elixir プロジェクトの開発	山内 修、大原 常德（一部）
第 6 章	並行プログラミング	大原 常德
第 7 章	Phoenix の概要	齋藤 和也
第 8 章	Ecto によるデータベース操作	山内 修
第 9 章	phx.gen.auth による認証	齋藤 和也
第 10 章	LiveView によるフロントエンドの開発	齋藤 和也
第 11 章	実践的な Web アプリケーションの開発	齋藤 和也
第 12 章	行列演算ライブラリ Nx の概要	隆藤 唯章
第 13 章	Axon の概要と機械学習システム開発の進め方	隆藤 唯章
第 14 章	機械学習向けのライブラリ	隆藤 唯章
第 15 章	実践的な Axon アプリケーションの開発	隆藤 唯章
第 16 章	Nerves の概要	高瀬 英希
第 17 章	Nerves での開発の進め方	高瀬 英希
第 18 章	Elixir Circuits によるモジュールの制御	高瀬 英希
第 19 章	実践的な IoT アプリケーションの開発	高瀬 英希
あとがき		栗林 健太郎

目次 ■ Elixir 実践入門——基本文法、Web 開発、機械学習、IoT

はじめに.....	iii
謝辞.....	viii
執筆者と担当章.....	ix
目次.....	x

第1章

Elixir 小史	1
Elixir 言語の特徴	2
モダンな関数型言語.....	3
アクターモデルによる並行処理と耐障害性.....	3
大規模システムでの利用.....	4
Erlang/OTP —— Elixir の実行基盤	4
Erlang とはどんな言語か.....	5
Erlang が得意なこと・苦手なこと.....	5
関数型言語としての Erlang.....	6
コンパイラ言語としての Erlang.....	6
プロセスとアクターモデル.....	6
Erlang におけるプロセス.....	7
アクターモデルによるプログラムデザイン.....	7
OTP による Erlang の拡張.....	8
OTP —— Open Telecom Platform.....	8
ビヘイビアによる設計の抽象化.....	8
Elixir の誕生	10
Elixir が作られた背景.....	10
並行処理へのアプローチ.....	10
BEAM 言語としての Elixir.....	11
Elixir の特徴的なシンタックス.....	11
Elixir と Erlang/OTP との関係.....	11
Elixir のプログラム実行.....	12
OTP から見た Elixir アプリケーション.....	12
Elixir の発展.....	12
Elixir のエコシステム.....	12
Elixir のコミュニティ.....	13
Elixir の持つポテンシャル	13
ネットワークアプリケーション.....	13
ネットワークアプリケーションの構造.....	13
Elixir によるネットワークアプリケーション実装のメリット.....	14
Elixir で実装されたネットワークアプリケーション.....	14
Web アプリケーション.....	14
Web アプリケーションの構造.....	14
Web アプリケーションフレームワーク.....	15
Phoenix による Web アプリケーション開発.....	16
機械学習.....	16
機械学習アプリケーションの構造.....	16
Elixir による機械学習.....	17

Nx のエコシステム.....	17
IoT デバイス.....	18
IoT アプリケーションの構造.....	18
IoT と Elixir の親和性.....	18
Nerves のエコシステム.....	19
まとめ.....	19

第2章

Elixir の基礎..... 21

Elixir のインストール.....	22
macOS 上での Elixir のインストール.....	22
パッケージマネージャを利用したインストール.....	22
Windows 上での Elixir のインストール.....	23
バージョン管理ツール asdf を用いた Elixir のインストール.....	23
asdf のインストール.....	23
asdf を利用した Elixir のインストール.....	24
Docker を利用した Elixir の環境構築.....	25
Elixir イメージを利用した環境構築.....	26
Elixir コードの実行方法.....	27
IEx —— REPL 環境を提供する CLI アプリケーション.....	27
Livebook —— Elixir コードを GUI で実行できるアプリケーション.....	27
ファイルからの実行.....	28
コンパイルして実行する場合.....	28
スクリプトとして実行する場合.....	29
基本的な文法.....	29
変数.....	29
代入ではなく束縛.....	30
不変性.....	31
演算子.....	32
算術演算子 —— 数値計算を行う基本的な演算子.....	32
比較演算子 —— 値の比較を行う基本的な演算子.....	33
論理演算子 —— 論理計算を行う基本的な演算子.....	34
パイプ演算子.....	35
マクロ.....	35
制御フロー.....	36
if、unless —— 単独の条件式による条件分岐.....	36
cond —— 条件式による条件分岐.....	37
case —— パターンマッチによる条件分岐.....	38
まとめ.....	39

第3章

基本的な型とパターンマッチ..... 41

基本的な型.....	42
整数 —— 整数を表す型.....	42
浮動小数点 —— 小数点を表す型.....	43

真偽値——真と偽を表す型.....	43
アトム——名前付きの定数を表す型.....	44
文字列——文字列を表す型.....	44
文字リスト——文字コードのリストを表す型.....	45
リスト——連結リストで複数の値をまとめる型.....	46
タプル——連続メモリ空間で複数の値をまとめる型.....	47
マップ——複数のキーと値のペアを表す型.....	48
キーワードリスト——連結リストで複数のキーと値のペアを表す型.....	49
構造体——任意のデータ構造を表す型.....	50
範囲——範囲を表す型.....	51
日付と時間——日付と時間を表す型.....	51
シジル——リテラルを表現する記法.....	52
~C と ~c ——文字リストを表すシジル.....	52
~S と ~s ——文字列を表すシジル.....	53
~W と ~w ——単語のリストを表すシジル.....	53
~R と ~r ——正規表現を表すシジル.....	53
日付や時刻を表すシジル.....	53
パターンマッチ——データとパターンの照合.....	54
パターンマッチとは何か.....	54
さまざまなパターンマッチ.....	55
変数のパターンマッチ.....	55
タプルのパターンマッチ.....	56
リストのパターンマッチ.....	56
マップのパターンマッチ.....	57
文字列のパターンマッチ.....	57
パターンマッチが可能な場所.....	58
関数の引数.....	58
case 式.....	60
ピン演算子——パターンマッチにおける変数の固定.....	60
ガード節——パターンマッチへの条件の追加.....	61
まとめ.....	62

第4章

モジュール..... 63

モジュールと関数によるプログラムの構造化.....	64
モジュールの作成.....	64
無名関数.....	65
別のモジュールで定義した関数の呼び出し.....	66
関数のアリティ表記.....	66
関数キャプチャ.....	67
関数キャプチャを引数に利用.....	67
キャプチャ演算子 & を用いて無名関数を作成.....	68
標準モジュール.....	68
String ——文字列操作を扱う標準モジュール.....	69
length ——長さを取得する関数.....	69
contains? ——指定の文字列を含んでいるかどうかを返す関数.....	69

upcase	— 文字列を大文字にする関数	70
downcase	— 文字列を小文字にする関数	70
capitalize	— 文字列の先頭を大文字にする関数	70
replace	— 文字列を置換する関数	70
reverse	— 文字列を逆順にする関数	71
split	— 文字列を分割する関数	71
split/1		71
split/3		72
slice	— 部分文字列を返す関数	73
slice/2		73
slice/3		73
File	— ファイル操作を行う標準モジュール	73
read	— ファイルを読み込む関数	74
write	— ファイルに書き出す関数	74
IO	— 標準入出力を扱う標準モジュール	75
puts	— 文字列を出力する関数	75
inspect	— 値を文字列に変換して出力する関数	75
read	— 文字列を読み込む関数	76
Enum	— コレクションを「いい感じ」に扱う標準モジュール	77
map	— 各要素を変換する関数	77
filter、reject	— フィルタリングする関数	77
filter		77
reject		78
all?、any?	— 要素を検証する関数	78
all?		78
any?		79
max、min	— 最大値、最小値を取得する関数	79
max		79
min		81
reduce	— 畳み込みを行う関数	81
パイプ演算子との併用		82
Map	— マップを「いい感じ」に扱う標準モジュール	82
new	— マップを生成する関数	82
new/0	— 空のマップを生成する関数	83
new/1	— コレクションからマップを生成する関数	83
new/2	— コレクションと関数からマップを生成する関数	83
has_key?	— マップにキーが含まれているかどうかを返す関数	83
get	— マップからキーを指定して値を取得する関数	84
put	— マップにキーと値を挿入する関数	84
update	— マップを更新する関数	85
merge	— マップをマージする関数	86
Map.merge/2		86
Map.merge/3		86
drop	— マップから指定したキーを削除する関数	87
keys	— マップからすべてのキーを取り出す関数	87
values	— マップからすべての値を取り出す関数	87
to_list	— マップをリストに変換する関数	87
Stream	— コレクションを遅延評価する標準モジュール	88
map	— 各要素を変換する関数	88

filter、reject —— フィルタリングする関数.....	88
filter	88
reject.....	89
iterate —— 無限のストリームを生成する関数.....	89
Enum と Stream のどちらを使うか.....	90
ExUnit —— Elixir の単体テスト標準モジュール	90
テストをすべて実行.....	91
一部のテストのみを実行.....	91
モジュールのテスト.....	92
doctest —— コメントにテストを記述する.....	92
まとめ.....	93

第5章

Mix を使った Elixir プロジェクトの開発..... 95

Mix の基本的な使い方	96
新しいプロジェクトの作成.....	96
mix.exs —— プロジェクトの設定を管理するファイル	96
依存関係の管理	97
Hex —— パッケージマネージャ.....	99
プロジェクト内で IEx の起動.....	99
プロジェクト設定の管理.....	100
Mix タスク —— Elixir 開発を支える便利なコマンド.....	100
mix new —— プロジェクトの作成.....	100
mix deps.get —— 依存パッケージの取得.....	101
mix compile —— プロジェクトのコンパイル	103
mix test —— テストの実行.....	103
mix release —— アプリケーションのリリース.....	104
mix format —— コードフォーマット.....	105
mix escript —— Erlang/Elixir のスクリプト.....	105
mix escript.build —— escript プログラムを作成.....	106
mix escript.install —— escript プログラムをローカルにインストール.....	108
mix help —— Mix タスクの一覧表示とヘルプの表示.....	108
Mix タスクの作成.....	108
開発に便利な Tips	109
observer —— GUI のモニタリングツール.....	109
dbg/2 —— デバッグ用のマクロ.....	112
まとめ.....	113

第6章

並行プログラミング..... 115

プロセスによる並行プログラミングの実現	116
プロセスの生成とメッセージの送受信.....	117

プロセスにメッセージを送信する	117
プロセスがメッセージを受信する	118
プロセスが別のプロセスを生成する	119
プロセスの監視	122
プロセスが別のプロセスをリンクする	122
プロセスが別のプロセスをモニタする	124
高度なプロセス	126
タスク——バックグラウンドで関数を実行するプロセス	126
エージェント——状態を持ったプロセス	127
OTP による並行プログラミングの実現	129
GenServer ビヘイビア——サーバの振る舞いを抽象化したパターン	129
カウンタサーバを実装してみよう	130
カウンタサーバの初期化	130
カウンタサーバの起動	131
同期呼び出しと非同期呼び出し	132
カウンタサーバの同期呼び出し	133
同期呼び出しのカウンタアップ	133
同期呼び出しのカウンタダウン	134
カウンタサーバの非同期呼び出し	136
Supervisor ビヘイビア——プロセスの監視と再起動	138
Supervisor モジュールによる実装	138
再起動設定	139
Counter をスーパーバイザーで監視する	139
まとめ	142

第7章

Phoenix の概要

143

Phoenix とは何か——Elixir 製の Web アプリケーションフレームワーク

144

Phoenix が解決すること——リッチな Web アプリケーション開発の効率化..... 144

Phoenix の特徴——高い生産性とパフォーマンスを兼ね備えたフレームワーク..... 144

Phoenix の基礎知識

145

Phoenix のディレクトリ構成.....

145

mix.exs ——プロジェクト全体の設定ファイル..... 145

assets ディレクトリ——フロントエンドのリソースを格納するディレクトリ..... 145

config ディレクトリ——アプリケーションの設定情報を格納するディレクトリ..... 146

deps ディレクトリ——Mix のすべての依存関係があるディレクトリ..... 146

lib ディレクトリ——アプリケーションのソースコードを格納するディレクトリ..... 146

priv ディレクトリ——ソースコードには直接含まれないリソースを管理するディレクトリ..... 146

test ディレクトリ——テストコードを格納するディレクトリ..... 147

ルーティング——URL とコントローラの対応

147

リクエストライフサイクル——Phoenix におけるリクエスト処理の流れ..... 147

ルーティングファイル router.ex の書き方..... 148

ルーティングの特殊な関数——resources と live..... 149

検証済みルート..... 149

コントローラ

150

アクションの定義..... 150

レンダリングの指定..... 151

リダイレクトの指定..... 152

フラッシュメッセージの表示..... 152

ビューとテンプレート

152

HTML のレンダリング..... 152

JSON のレンダリング	153
モデル	154
Ecto —— データベースを操作するライブラリ	155
コンテキスト —— 機能のグルーピング	155
Mix タスクによるスキュアフォールディング	156
mix phx.gen.html —— HTML リソースの生成	156
mix phx.gen.json —— JSON リソースの生成	157
mix phx.gen.live —— LiveView リソースの生成	157
mix phx.gen.auth —— ユーザー認証機能の生成	158
そのほかのスキュアフォールディング	158
Phoenix の基本的な使い方	158
PostgreSQL の起動	158
Phoenix プロジェクトジェネレータのインストール	159
Phoenix プロジェクトの作成	159
データベースの初期設定	160
Phoenix の起動	160
まとめ	161

第 8 章

Ecto によるデータベース操作

Ecto とは何か —— Elixir 製のデータベースライブラリ	164
Ecto が解決すること —— データベースへのさまざまな操作をサポート	164
Ecto の特徴 —— データマップおよびクエリビルダ	164
Ecto の基礎知識	164
主要コンポーネント	165
Ecto.Repo —— データベースのラッパ	165
Ecto.Query —— Elixir の文法で書けるクエリビルダ	165
Ecto.Schema —— テーブル構造と Elixir 構造体のマッピング	165
Ecto.Changeset —— データ変更の管理	165
便利な機能	165
データベースの作成、削除	166
マイグレーション —— データベースのスキーマに対する変更	166
トランザクション —— ひと塊の分けることができない操作	166
シードデータの投入	166
データリセット	166
Ecto の基本的な使い方	166
スキーマモジュールとマイグレーションファイルの作成	168
銀行口座スキーマモジュールとマイグレーションファイルの作成	168
取引スキーマモジュールとマイグレーションファイルの作成	169
マイグレーションの実行	170
スキーマモデルどうしの関連付け	170
Transfer と Account の関連付け	170
Account と Transfer の関連付け	171
バリデーション —— 入力データの検証	171
Transfer のバリデーション	172
Account のバリデーション	172
データの操作	173
作成 (Create)	173
読み出し (Read)	174

更新 (Update)	175
トランザクション	175
取引の読み出し	177
削除 (Delete)	178
シードデータの投入	179
まとめ	179

第9章

phx.gen.auth による認証 181

phx.gen.auth とは何か ——ビルトインの Mix タスク	182
phx.gen.auth が解決すること——リッチな認証機能の実装	182
phx.gen.auth の特徴——関連ファイルが生成されることによる処理の追跡とカスタマイズの容易さ	183
phx.gen.auth の基礎知識	183
認証方式	183
認証の基本機能	183
新規登録	183
ログイン/ログアウト	183
認証の補助機能	184
確認メールによる本登録	184
メールアドレスとパスワードの変更	184
パスワードリセット	184
アクセスコントロール	184
pipeline による処理の共通化	184
認証のスコープ	185
phx.gen.auth の基本的な使い方	186
認証機能の追加	186
パスワードバリデーションの変更	187
テストの修正	187
メールの送信方法の変更	188
Amazon SES	188
SendGrid	189
メール本文の設定	189
まとめ	190

第10章

LiveView によるフロントエンドの開発 191

LiveView とは何か ——Elixir で実装するリアルタイム Web	192
LiveView が解決すること——効率的な SPA の開発	192
LiveView の特徴——WebSocket による複数クライアント間でのデータのやりとり ..	192
LiveView の基礎知識	193
LiveView のディレクトリ構成	193
LiveView のライフサイクル	193
状態の管理と更新	194

UIの制御.....	196
HEEx テンプレート—— EEx を拡張した新しいテンプレートエンジン	196
DOM バインディング—— Elixir による DOM イベントのハンドリング	196
ルーティング—— URL と LiveView の対応.....	197
LiveView の実践的な使い方	198
コンポーネント分割.....	198
Phoenix.Component —— ステートレスなコンポーネント	198
Phoenix.LiveComponent —— ステートフルなコンポーネント.....	200
ファイルアップロード.....	201
PubSub によるリアルタイム更新.....	203
JavaScript との連携	203
JS Hooks —— LiveView と JavaScript の間での DOM イベントの送受信	203
JS Module —— LiveView から JavaScript のユーティリティ操作の実行.....	205
Phoenix.LiveViewTest による結合テスト.....	205
まとめ	206

第 11 章

実践的な Web アプリケーションの開発 207

ブログアプリケーション RealWorld の実装	208
RealWorld とは何か.....	208
開発する機能.....	208
記事の CRUD 機能の開発	209
スキャフォールディングの実行	209
マイグレーションファイルの変更	210
ルーティングの設定.....	210
マイグレーションの実行と動作確認.....	210
コメント機能の開発	211
コメントのスキーマモジュールとマイグレーションファイルの作成.....	211
Comment スキーマモジュールの変更.....	212
マイグレーションファイルの変更	212
記事とコメントの関連付け	213
マイグレーションの実行と動作確認.....	213
タグ機能の開発	214
タグの仕様.....	214
記事とタグを関連付けた保存	215
タグのスキーマモジュールとマイグレーションファイルの作成	215
記事とタグを関連付けるスキーマモジュールとマイグレーションファイルの作成	215
Article スキーマモジュールの変更	216
記事とタグを関連付けて保存する関数の追加.....	217
マイグレーションの実行と動作確認	219
タグによる記事検索.....	219
認証機能の開発	220
phx.gen.auth による認証機能の追加	220
記事とユーザーの関連付け	221
マイグレーションファイルの作成	221
Article スキーマモジュールの変更.....	221

コメントとユーザーの関連付け	222
マイグレーションファイルの作成	222
Comment スキーマモジュールの変更	223
アクセスコントロール	224
LiveView による RealWorld の開発	225
画面の実装——記事一覧／作成／編集／削除／詳細	226
記事の一覧	226
記事の作成	227
記事の編集	228
記事の削除	229
記事の詳細	230
タグによる記事検索	233
記事に紐づくタグの保存	233
記事に紐づくタグの表示	235
記事の一覧でのタグによる記事検索	236
タグによる記事検索のテスト	237
デプロイ	237
phx.gen.release によるリリース作成用ファイルの作成	238
Fly.io CLI のインストール	238
macOS でのインストール	238
Windows でのインストール	238
Linux でのインストール	239
Fly.io へのアプリケーションのデプロイ	239
まとめ	241

第 12 章

行列演算ライブラリ Nx の概要

Nx とは何か	244
Nx が解決すること	244
Nx のバックエンド	244
BinaryBackend —— 純 Elixir 製のバックエンド	245
EXLA.Backend —— XLA を使用したバックエンド	245
Torch.Backend —— libTorch を使用したバックエンド	245
Evision.Backend —— OpenCV を使用したバックエンド	246
Nx の基本的な使い方	246
行列の生成	246
Nx.tensor/2 —— さまざまな値からの生成	247
Nx.iota/2 —— 指定した行列の形状で生成	249
Nx.broadcast/3 —— 行列を上書き	250
Nx.Random モジュール	250
行列の情報の取得	252
Nx.shape/1 —— 行列の形状を取得	252
Nx.size/1 —— 行列の要素数を取得	253
Nx.rank/1 —— 行列のネストの深さを取得	253
Nx.axes/1 —— 行列の軸を取得	253
行列の演算	254
四則演算	255
Nx.dot/2 —— 内積の演算	256
比較演算	257
行列の変形	258

Nx.reshape/2 — 行列の変形	258
Nx.transpose/2 — 転置行列に変換	259
Nx.flatten/2 — 行列の平坦化	259
Nx.slice/4 — 行列の切り取り	260
Nx.slice_along_axis/4 — 軸指定可能な行列の切り取り	261
行列から値の取得	262
数値で指定した位置の値を取得	262
範囲で指定した位置の値を取得	263
Nx.argmax/2 — 最大値の位置を返す	266
Nx.reduce_max/2 — 最大値を返す	267
行列データの変換	268
Nx.to_number/1 — 数値型に変換する	268
Nx.to_list/1 — 行列をリストに変換する	269
Nx.to_flat_list/2 — 行列を平坦なリストに変換する	269
Nx.to_batched/3 — 行列をいくつかのまとまりのリストに変換する	269
まとめ	270

第13章

Axon の概要と機械学習システム開発の進め方..... 271

Axon とは何か	272
Axon が解決すること	272
Axon の特徴	272
Axon の基本的な使い方	273
MNIST — 手書き数字の画像データセット	273
推論モデルの構築手順	273
SciData と Nx による学習データの準備	274
SciData によるデータのダウンロード	274
Nx によるデータの整形	275
Axon によるモデルの構築	278
Axon.Loop によるモデルの学習と可視化	280
Axon.Loop.trainer/4 — 損失関数と最適化アルゴリズムの設定	281
Axon.Loop.metric/4 — 算出する評価指標の設定	281
Axon.Loop.kino_vega_lite_plot/4 — 学習過程を描画	282
Axon.Loop.run/4 — 学習の実行	282
Axon.Loop によるモデルの検証	284
Axon.predict/4 — 推論の実行	285
Axon.predict/4 の実行結果	286
Nx.argmax/2 — 結果の絞り込み	286
Nx.to_heatmap/2 を使用した検証	287
まとめ	287

第 14 章

機械学習向けのライブラリ	289
Kino — Livebook 用の UI ライブラリ	290
Kino とは何か	290
Kino の特徴	290
スマートセル	290
Database connection	291
SQL query	292
Chart	293
Data transform	293
Map	294
Neural Network task	295
Slack Message	296
Deploy	297
Stblmage — 軽量画像読み書きライブラリ	298
Stblmage とは何か	298
Stblmage の特徴	298
Stblmage の使い方	299
Stblmage.read_file/2 — 画像の読み込み	299
Stblmage.read_binary/2 — バイナリデータの読み込み	299
Stblmage.resize/3 — 画像のリサイズ	300
Stblmage.write_file/3 — 画像の書き出し	300
Stblmage.to_nx/2 — 画像データの Nx.Tensor への変換	300
Evision — OpenCV ラッパー	301
Evision とは何か	302
Evision の特徴	302
Evision の使い方	302
Evision.imread/1 — 画像の読み込み	302
Evision.imwrite/1 — 画像の書き出し	303
Evision.resize/2 — 画像のリサイズ	303
Evision.rotate/2 — 画像の回転	304
Evision.rectangle/4 — 画像に長方形を描画	304
Evision.putText/6 — 画像に文字を描画	305
Evision.Mat.roi/2 — 画像の切り取り	306
Evision.Mat.to_nx/2 — 画像データの Nx 形式への変換	306
Evision.Mat.from_nx_2d/1 — Nx.Tensor を Evision.Mat に変換	307
Evision.DNN.DetectionModel — 物体検知を行うモジュール	307
Bumblebee — 学習済み Transformer モデル提供ライブラリ	310
Bumblebee とは何か	310
Bumblebee の特徴	310
Bumblebee の使い方	312
利用可能なモデル	314
まとめ	314

第 15 章

実践的な Axon アプリケーションの開発 315

画像分類を行う Web アプリケーションの実装	316
画像分類とは何か	316
実装する機能の説明	316
追加ライブラリのインストール	316
LiveView ページの作成	317
ファイルアップロード機能の実装	318
アップロードの設定	318
フォームの作成	318
サムネイル画像の表示	319
学習済みモデルでの画像分類機能の実装	320
画像データの変換	320
バイナリデータからサムネイル画像の表示	322
Bumblebee で学習済みモデルの読み込み	323
Nx.Serving による推論サーバの起動	324
推論結果の表示	324
まとめ	327

第 16 章

Nerves の概要 329

Nerves とは何か	330
Nerves が解決すること	330
リソース制約の厳しいデバイスへの対応	330
実世界とのインタラクティブなデータ処理	331
詳細なドメイン知識なしでの IoT デバイスの開発	331
さまざまなインターネット通信方式への対応	331
Nerves の特徴	332
IoT システムに適した実行環境	332
潤沢なフレームワーク	333
円滑に開発を進められるツール	334
Nerves の動作する IoT ボード	335
動作要件	335
Buildroot のサポート	335
プロセッサの種類や性能	335
ストレージデバイス	335
公式サポートされているターゲット	336
Raspberry Pi ファミリー	336
BeagleBone ファミリー	336
そのほかのデバイス	336
まとめ	337

第 17 章

Nerves での開発の進め方 339

用意するもの	340
ホスト開発環境の準備	340
推奨される OS 環境	340
Nerves のためのライブラリのインストール	341
Nerves.Bootstrap のインストール——Nerves 専用のジェネレータ	341
fwup のインストール——Nerves ファームウェアの作成ツール	342
SSH 接続のための鍵ペアの作成	342
ターゲットとなるハードウェア	343
Nerves プロジェクトの基本的な開発の進め方	344
プロジェクトの作成とビルド	344
mix nerves.new ——プロジェクトの作成	344
環境変数 MIX_TARGET の設定——ターゲットの設定	345
インターネット接続の設定	346
mix deps.get ——依存ライブラリの取得	347
mix firmware ——ファームウェアのビルド	347
Nerves の動作確認	348
mix burn —— microSD カードへのファームウェアの書き込み	348
ssh nerves.local ——Nerves デバイスへのアクセス	349
Hello Nerves!	350
lEx の終了	351
Nerves ファームウェアのネットワーク越しの更新	351
ソースコードの書き換え	352
mix upload ——SSH 経由での更新	352
更新結果の確認	353
まとめ	353

第 18 章

Elixir Circuits によるモジュールの制御 355

Elixir Circuits とは何か——Elixir/Nerves 向けのモジュール制御ライブラリ ...	356
Elixir Circuits が解決すること	356
Elixir Circuits の特徴	356
モジュールの通信方式	357
GPIO —— 1 ビット単位の制御に向けた通信方式	358
I2C ——バス上のアドレスを指定した通信方式	358
用意するもの	358
Grove Base Hat	359
Grove モジュール	359
Grove モジュールの組み立て	359
Elixir Circuits の使い方	360
ライブラリの追加	361
Circuits.GPIO による LED 点滅の制御	362
Circuits.GPIO.open —— GPIO ピンを設定する関数	362
Circuits.GPIO.write —— GPIO ピンに値を書き出す関数	362

Circuits.GPIO.close —— GPIO ピンのリファレンスを解放する関数	363
LED を制御するドライバ関数のライブラリ化	363
Circuits.GPIO によるボタン入力の検知	364
Circuits.GPIO.read —— GPIO ピンの値を読み込む関数	365
Circuits.GPIO.set_interrupts —— GPIO ピンの入力値の変化を検知する関数	365
ボタンを制御するドライバ関数のライブラリ化	366
Circuits.I2C による温度と湿度の測定	367
Circuits.I2C.detect_devices —— モジュールの I2C アドレスを確認する関数	368
Circuits.I2C.open —— I2C バスを開く関数	368
Circuits.I2C.write —— I2C モジュールに値を書き出す関数	368
Circuits.I2C.read —— I2C モジュールの値を読み込む関数	369
Circuits.I2C.close —— I2C バスのリファレンスを解放する関数	370
温湿度センサを制御するドライバ関数のライブラリ化	370
まとめ	372
第 19 章	
実践的な IoT アプリケーションの開発	373
開発するもの	374
データ受け取りサーバの作成	375
Phoenix プロジェクトの作成	375
IP アドレスの設定	375
mix phx.gen.json による REST API の作成	376
動作の確認	377
データのリアルタイム表示部の作成	377
表示するデータの取得	378
表示ページの作成	378
ルーティングの設定	379
動作の確認	379
データ送信モジュールの作成	380
ライブラリの追加	381
Phoenix サーバへのデータの送信機能の作成	382
Nerves ファームウェアのビルドと更新	383
動作の確認	383
ボタンの押下によるデータ測定	384
ボタンの押下に応じたプロセスの実行	384
Nerves ファームウェアのビルドと更新	385
動作の確認	386
自動実行の設定	386
まとめ	387
あとがき	388
索引	389
著者プロフィール	399

第 1 章

Elixir 小史

本書では、プログラミング言語 Elixir について、幅広い領域における利用法を解説します。すなわち、Elixir の言語としての基礎から始まり、そのエコシステムが提供する独特の Web アプリケーション開発スタイルに加えて、IoT や機械学習開発についても扱います。

Elixir は、なぜそのような幅広い領域において用いられる言語となったのでしょうか。本章では、プログラミング言語 Elixir の特徴を簡単に紹介したあと、以下の内容について解説し、Elixir が持つポテンシャルについての理解を目指します。

- Elixir の実行基盤である Erlang/OTP^{注1} の歴史と特徴
- Erlang/OTP を基盤に Elixir が登場した背景
- Elixir の誕生から現在までの変遷

Elixir 言語の特徴

Elixir は、José Valim 氏によって開発されているプログラミング言語です。最大の特徴は、プログラミング言語 Erlang の VM (*Virtual Machine*、仮想マシン) 上で動作し、Erlang のモジュールを直接利用できることです。Erlang VM 上で動作することから、Elixir は以下のような特徴を備えています。

- 耐障害性
- 高可用性
- 分散システム

Erlang は、後述する OTP というさまざまな分散アプリケーションの実装パターンを抽象化したフレームワークを持っています。Erlang はプログラミング言語であり、OTP は Erlang のフレームワークですが、これらは密接に関連していて、一緒に利用されるのが一般的です。そのため Erlang/OTP という表記がよく用いられます。本章では、プログラミング言語の文脈では「Erlang」、プログラミング環境の文脈では「Erlang/OTP」、言語の処理系の文脈では「Erlang VM」という表現を用います。

注1 <https://www.erlang.org/>

したがってElixirは次のErlang/OTPによるプログラミングのパラダイムを受け継いでいます。

- 関数型
- アクターモデル
- プロセス間のメッセージパッシングによる並行処理

本節ではこれらの特徴的な言語機能について解説していきます。

モダンな関数型言語

Elixirは比較的新しい言語で、以下のような現代的なプログラミング言語の要素を備えています。

- ポリモフィズム
- マクロ/メタプログラミング
- パターンマッチ
- テスティング
- ビルドツール
- ライブラリエコシステム

また、プログラミング言語において、生成、代入、引数や戻り値としての受け渡しといった操作を制限なしに行える対象を第一級オブジェクトと言います。Elixirは関数型言語でありElixirの関数は第一級オブジェクトですが、Erlang VM上で生成されるプロセス上で動作することから、ほかの関数型言語とは少し毛色が異なります。

詳細はのちの章で解説しますが、Elixirはこれらの特徴や機能を持つことから、小さなスクリプトから大規模システムのサーバまで無理なく記述できるのです。

アクターモデルによる並行処理と耐障害性

並行処理に対するアプローチは、プログラミング言語によってさまざまです。マルチスレッド、ソフトウェアトランザクショナルメモリ、並行論理プログラミングなど多くの手法があります。Elixirは、並行処理の実現に

アクターモデルを採用しています。

Elixir (の基盤になる Erlang/OTP) はそれぞれ独立して動作するプロセス上で関数が動作し、複数のプロセス間でメッセージをやりとりして処理が実行されます。プロセスは受信したメッセージによって次の動作を決定し、これらの一連の振る舞いは逐次ではなく並行的、非同期的に実行されます。

また Elixir は耐障害性が高い言語として知られています。この耐障害性は次の2つの特徴によるところが大きいです。

- Supervisor によるプロセス監視
- Erlang VM の堅牢性

OTP 中の Supervisor というプロセスの監視パターンを利用することで、プロセスがクラッシュした場合でもプロセスを適切に再起動させることができます。

また、Erlang VM は非常に堅牢で、安定してプログラムを実行できます。筆者は長年プロダクション環境で Elixir/Erlang のサービスを運用していますが、Erlang VM がクラッシュするトラブルに遭遇したことはほとんどありません。

大規模システムでの利用

前節で OTP は分散アプリケーションの実装パターンを集めたフレームワークであると解説しました。この OTP を利用することで、複雑な分散処理を少量のコードで実装することが可能となります。Elixir が大規模で複雑な分散処理を実装するのに適している理由は、この OTP を利用できることにあります。

Erlang/OTP — Elixir の実行基盤

前節では Elixir は Erlang VM 上で動作するプログラミング言語と解説しました。本節では Erlang/OTP の特徴を解説していきます。

Erlang とはどんな言語か

Erlang はエリクソン社が開発した並行処理を簡潔に記述できるプログラミング言語です。もともとは電話交換機のように高い信頼性が要求されるプログラムのために設計されたことから、以下の特徴を持っています。

- 耐障害性
- 分散処理
- 無停止稼働

無停止稼働の特徴により、稼働中のシステムを停止することなくプログラムを変更するホットコードスワップを行うことができます。

これらの機能により、Erlang は高い信頼性が求められるシステムを無停止で稼働させることができます。

さらに先述のとおり、Erlang は以下のプログラミングのパラダイムを持っています。

- 関数型
- アクターモデル
- プロセス間のメッセージパッシングによる並行処理

Erlang は処理をモジュール、関数、引数の組み合わせで記述します。なお、モジュール (Module)、関数 (Function)、引数 (Argument) の頭文字をとって、これらの組み合わせを MFA と呼びます。Erlang VM 上に、これらの関数を処理するプロセスが生成され、複数のプロセスがそれぞれ独立して動作します。

■ Erlang が得意なこと・苦手なこと

Erlang は耐障害性、無停止稼働の特徴を持っていることから、「落ちない」システムを記述するのに優れています。たとえば広告サーバや常時稼働を前提としたミドルウェアなどの、高い SLA (*Service Level Agreement*、サービス水準合意) を求められるシステムを記述することが得意です。また、大量のプロセスを手早く起動できるため、並列処理も得意とされています。

その一方、Erlang は C や C++ に比べると処理速度は速くありません。リストやマップといったデータ構造、浮動小数点数のデータ型を Erlang は持つ

ていますが、CやC++に比較するとオーバーヘッドが大きく高速に処理を行うことはできません。したがって、数値計算や画像処理などを行うのには向いていません。

なお、現在のElixirは、後述する行列演算ライブラリであるNxと、そのバックエンドとしてネイティブな計算処理を賄うライブラリを利用することで、数値計算や画像処理に向かない弱点を克服しています。これらの詳細は第12章～第15章で解説します。

■ 関数型言語としてのErlang

Erlangは関数型言語です。関数を第一級オブジェクトとして扱えるので、関数自身を引数として別の関数に渡したり、値として扱ったりできます。また、Erlangはアクターモデルによる処理を実現できるよう設計されており、関数はアクターであるプロセス上で実行されます。これがほかの関数型言語と大きく異なる点です。

ほかのプログラミング言語とErlangのパラダイムを比較する際、関数型言語かどうかよりも、アクターモデルベースかどうかの違いのほうが大きいと筆者は考えています。

■ コンパイラ言語としてのErlang

Erlangはコンパイル言語です。Erlangのソースコードであるerlファイルがコンパイルされると、モジュールごとにbeamファイルが生成されます。beamファイルはErlang VMのマシン語に変換されたバイナリで、Erlang VM上で実行されます。また、コンパイル時にデバッグ情報を付与することでbeamファイルにデバッグ情報を含めることができます。なお、Erlangのコンパイルや逆コンパイル、逆アセンブルなどの機能は、Erlangにバンドルされた標準ライブラリの関数として提供されます。

プロセスとアクターモデル

本項ではErlangのプロセスとアクターモデルについて解説します。

一般的なプログラミング言語は、プログラムのエン트리ポイントであるmain関数から処理が始まり、分岐や関数呼び出しを繰り返しながら処理を進めていきます。一方Erlangのプログラムは、プログラム起動時に生成さ

れるプロセスが別のプロセスを生成し、さらに別のプロセスを生成すると
いった形で、多くのプロセスがメッセージのやりとりをしながら処理を進
めていきます。

このように一般的なプログラミング言語のプログラムの実行パラダイム
は1次元的なものに対して、Erlangのプログラムの実行パラダイムは多次元
的だと言えるでしょう。

■ Erlang におけるプロセス

紛らわしいことに、ErlangのプロセスとOSのプロセスはまったくの別
物です。Erlangの「プロセス」は、プロセスごとの識別ID（プロセスID、
PID）を持ち、内部に受信したメッセージを保持するキュー（メッセージ
ボックス）を持ちます。プロセスを生成するのに必要なメモリはデフォ
ルトで数百ワード、起動にかかる時間は数マイクロ秒と非常に小さく、大量
のプロセスを極めて短い時間で生成できます。これらのプロセスはそれぞ
れ並列に動作し、同時に別の計算や処理を行うことができます。

■ アクターモデルによるプログラムデザイン

複数のプロセス間では以下のことが可能です。

- プロセスにメッセージを送信する
- プロセスからメッセージを受信する
- プロセスが別のプロセスを生成する
- プロセスが別のプロセスをリンクする
- プロセスが別のプロセスをモニタする

メッセージの送受信やプロセスの生成は簡単にイメージできますが、「リ
ンク」または「モニタ」はイメージがつかみにくいかもしれません。

あるプロセスが別のプロセスと「リンク」されると、プロセスどうしが互
いを監視するようになります。これによって、一方のプロセスが死んだと
き、もう一方のプロセスに対して終了メッセージを送信できます。

「リンク」がプロセス間の相互監視だったのに対して、一方通行の監視を
行うのが「モニタ」です。「リンク」と「モニタ」には相互に監視し合うの
か、片方だけを監視するののかという違いがあります。あるプロセスが別の
プロセスを「モニタ」しているケースを考えてみましょう。監視先のプロセ

スが死んだとき、監視元のプロセスに対して終了メッセージが送信されます。一方、監視元のプロセスが死んだとき、監視先のプロセスに対して終了メッセージは送信されません。つまり、「モニタ」は監視先のプロセスだけを監視するしくみと言えるでしょう^{注2}。

プロセスはこれらの機能を使って監視し合い、より複雑な並行処理を実現します。

OTP による Erlang の拡張

前項では、Erlang のプロセスによるプログラムデザインについて解説しました。複数のプロセスが協調し合いながら処理を進めることができるので、並行処理を簡単に記述できます。しかし、複雑な並行処理を実装する場合には多くのことを考慮する必要があります。本項で解説する OTP を利用することで、この複雑さをラップしてシンプルに複雑な並行処理を実装できるようになります。

■ OTP — Open Telecom Platform

OTP は Open Telecom Platform の略で、さまざまな並列処理を抽象化したフレームワーク・ライブラリ群です。また、プログラミング言語 Erlang について言及する際、「Erlang」ではなく「Erlang/OTP」と一緒に記載されることが多いです。Erlang のプログラミングにおいて、OTP は非常に重要な位置を占めています。

OTP で実現可能な並列処理のパターンの詳細は第6章で解説しますが、これらのパターンを組み合わせることで、いわゆる「落ちない」システムを簡単に構築できます。

■ ビヘイビアによる設計の抽象化

OTP では、並列処理の定番のパターンを「ビヘイビア」として形式化しています。ビヘイビアは次の2つのモジュールから構成されます。

- ビヘイビアモジュール

OTP がライブラリとして提供。gen_xxx という名前で定義

注2 詳しくは第6章の「プロセスの監視」(122ページ)で解説します。

- コールバックモジュール

ユーザーが定義

ビヘイビアモジュールはOTPが提供するライブラリとして実装されており、プログラマーはこれらのビヘイビアをコールバックモジュール内で宣言します。

クライアント／サーバの処理を提供する `gen_server` ビヘイビアを例に見てみましょう。なお以降のソースコードはErlangのソースコードです。シンタックスが少し特徴的ですが、「こういうものなのだな」と思って読み進めてください。

最初に、コールバックモジュール内でどのビヘイビアの規約に従うかを宣言します。

モジュールとビヘイビアの宣言

```
-module(ch1).
-behaviour(gen_server). %% gen_serverビヘイビアの宣言
```

次に、必要なコールバック関数をエクスポートします。

コールバック関数のエクスポート

```
-export([start_link/0]).
-export([alloc/0, free/1]).
-export([init/1, handle_call/3, handle_cast/2]).
```

最後に、それぞれエクスポートするコールバック関数を実装します。

コールバック関数の実装

```
start_link() ->
  gen_server:start_link({local, ch1}, ch1, [], []).

alloc() ->
  gen_server:call(ch1, alloc).

free(Ch) ->
  gen_server:cast(ch1, {free, Ch}).

init(_Args) ->
  {ok, channels()}.

handle_call(alloc, _From, Chs) ->
  {Ch, Chs2} = alloc(Chs),
  {reply, Ch, Chs2}.

handle_cast({free, Ch}, Chs) ->
  Chs2 = free(Ch, Chs),
  {noreply, Chs2}.
```

このコールバックモジュールをコンパイルして実行すると、`gen_server`の振る舞いをするプログラムが実行可能になります。代表的なビヘイビアとして以下のビヘイビアがよく利用されます。

- `GenServer`
- `Supervisor`

これらのビヘイビアは第6章で解説します。

Elixir の誕生

ElixirのGitHubリポジトリの一番古いコミットは2011年1月9日で、約3年半の開発期間のあと、バージョン1.0が2014年9月にリリースされました。本書の執筆時点ではElixirの最新バージョンは1.15.0になっています。

Elixir が作られた背景

Elixirの開発が始まった2011年は、並行処理の重要性が高くなっていた時期でした。当時よく利用されていたプログラミング言語はオブジェクト指向言語が主流で、並行処理をこれらの言語で実現するには、スレッド間の状態を管理する必要がありました。そのためにロックやミューテックスといった複雑なしくみが必要という問題がありました。

そこで作者のJoséはこれらの課題解決のアプローチとして関数型言語に注目しました。オブジェクトの状態を管理するのではなく、関数を実行して状態が変更された新しいオブジェクトを新たに生成することで、シンプルに並行処理を表現できると考えたのです。

そして、さまざまな関数型言語を研究した結果、Erlang/OTPが採用されました。しかしErlang/OTPではUnicodeの対応がなかったり、メタプログラミングができなかったりなどの制約が多く、のちにElixirとなる新しいプログラミング言語の開発を開始するに至りました。

■ 並行処理へのアプローチ

ErlangにOTPがバンドルされているように、ElixirにもOTP相当の並行処理のパターンを抽象化したビヘイビアモジュールが標準ライブラリとし

てバンドルされています。これらのビヘイビアモジュールを使うことで、Elixir は Erlang/OTP 同様に並行処理をコールバック関数の実装だけで実現できます。

■ BEAM 言語としての Elixir

コンパイル時に Erlang VM の実行バイナリである beam ファイルを生成する言語を BEAM 言語と呼びます。Elixir も BEAM 言語の一つです。

BEAM 言語の種類は多く、Elixir 以外では表 1.1 のような言語があります。

● 表 1.1 主要な BEAM 言語

BEAM 言語	公式サイト
LFE	https://lfe.io/
Efene	http://efene.org/
luerl	https://github.com/rvirding/luerl
clojerl	https://www.clojerl.org/
puerl	https://puerl.fun/
Gleam	https://gleam.run/

■ Elixir の特徴的なシンタックス

Elixir が作られた背景にあるように、Elixir はメタプログラミングが可能で、また Ruby に近い見た目のシンタックスが特徴的で、Erlang/OTP に比べて読みやすいプログラミング言語です。

本章では詳細な説明は割愛しますが、Elixir のシンタックスで特徴的なものに以下があります。

- パイプ演算子 (`|>`)
- パターンマッチ
- メタプログラミング

Elixir と Erlang/OTP との関係

Elixir は BEAM 言語の一種ですので、beam ファイルをコンパイル後に出力して、Erlang VM 上で動作します。また、Elixir のコンパイラなどのコアモジュールは、Erlang/OTP で実装されています。ですので、Elixir 本体は Erlang/OTP を使って実装されていると言えるでしょう。

■ Elixir のプログラム実行

Elixirのソースコード(exファイル)をコンパイルすると、beamファイルが生成されます。これらのbeamファイルは、Erlangでコンパイルしたbeamファイルと同じ形式で、Erlang VM上で同様に実行可能です。そのため、ElixirからErlang/OTPのモジュールや関数をオーバーヘッドなしで利用できます。つまり、Erlang/OTPのライブラリをElixirは再利用できるのです。

■ OTP から見た Elixir アプリケーション

ElixirにはコンパイラやREPL (*Read-Eval-Print Loop*) のインタフェースといったプログラムがバンドルされています。これらのプログラムはErlang/OTPを使って実装されています。Erlang/OTPで実装されているので、当然OTP (の並行処理パターン) を使ってErlang VM上で動作します。実際のところ、ElixirコンパイラやREPLはSupervisorビヘイビアやGenServerビヘイビアなどのさまざまなOTPビヘイビアを使って実装されています。

Elixir の発展

Elixirがリリースされてから2023年で11年になります。リリース当初はライブラリが少なく適用領域も限られていましたが、近年ではIoTや機械学習、SPA (*Single Page Application*) やスマートフォンネイティブアプリ開発などの領域でも採用されるようになってきました。

■ Elixir のエコシステム

Elixirのライブラリは、Hexと呼ばれるパッケージマネージャで管理されています。ライブラリを探す際はhex.pm^{注3}を参照するとよいでしょう。hex.pmではElixirのほかにErlang/OTPで利用できるライブラリもホスティングされています。

また、ElixirにはMixと呼ばれるビルドツールがバンドルされているので、Mixを使うことで簡単にhex.pmからライブラリをインストールできます。

注3 <https://hex.pm/>

■ Elixir のコミュニティ

Elixir のコミュニティは全世界に数多くあります。GitHub で運用されている Elixir の開発リポジトリの Wiki には、主要なミートアップ^{注4}やカンファレンス^{注5}の一覧が掲載されています。

また、日本でも数多くのミートアップやカンファレンスが開催されています。2019 年のコロナパンデミック以降はオンラインでの開催が主流になっていますが、近年ではオフラインイベントが復活してきています。

Elixir の持つポテンシャル

Elixir は並行処理を記述しやすい言語として開発が始まりましたが、近年ではさまざまな分野で利用されるようになってきました。

ネットワークアプリケーション

Elixir の基盤となっている Erlang/OTP は、信頼性の高いシステムを構築する言語として開発されました。信頼性が求められるシステムは多くありますが、その中でも複数のサーバ上で動作するネットワークアプリケーションは高い可用性が求められる分散システムですので、Elixir と相性が良いシステムと言えるでしょう。

■ ネットワークアプリケーションの構造

ネットワークアプリケーションは数多くありますが、その構造としては主に以下の特徴があります。

- ネットワーク上の分散された複数のノード上で動作する
- サーバ/クライアントの構造を持つ

また、ネットワークを通して利用されるため、大量のコネクションやデータを扱える必要があります。

注 4 <https://github.com/elixir-lang/elixir/wiki/Meetups>

注 5 <https://github.com/elixir-lang/elixir/wiki/Conferences>

■ Elixir によるネットワークアプリケーション実装のメリット

ネットワークアプリケーションである Web サーバを考えてみましょう。Web サーバに求められる要件として以下のようなものがあり、これらはまさに Erlang/OTP が得意とするものです。

- 高負荷に強い
- 高い可用性
- 大量のコネクションをさばくことができる

したがって、Elixir を使うことで、Erlang/OTP の特徴を活かしたネットワークアプリケーションを簡単に実装できます。

■ Elixir で実装されたネットワークアプリケーション

Elixir や Erlang/OTP でよく利用される Web サーバとして Cowboy^{注6}があります。Cowboy は Erlang/OTP で実装された Web サーバですが、Elixir でも利用でき、Elixir の Web アプリケーションフレームワークである Phoenix でも採用されています。

そのほかに、Elixir で実装された Web サーバとして Bandit^{注7}があります。Cowboy と比較して性能が高く、注目度が高い Web サーバです。

Web アプリケーション

Elixir で Web アプリケーションを開発する場合、Phoenix を利用することがほとんどです。Phoenix は Rails のようなフルスタック Web アプリケーションフレームワークで、Elixir のキラライブラリと言えるでしょう。

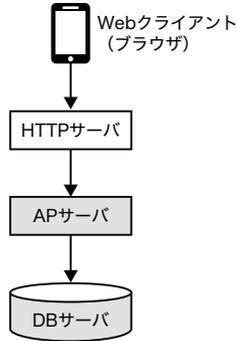
■ Web アプリケーションの構造

一般的な Web アプリケーションの構造は図 1.1 のようになります。

注6 <https://github.com/ninenines/cowboy>

注7 <https://github.com/mtrudel/bandit>

● 図1.1 Webアプリケーションの構造

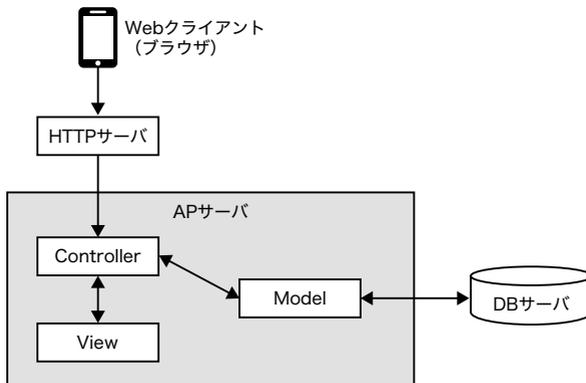


Webクライアントはブラウザで、HTTPサーバはHTTPリクエストを受け取り、APサーバはHTTPサーバからのリクエストを処理し、DBサーバはデータの永続化を行います。

■ Webアプリケーションフレームワーク

Webアプリケーションフレームワークとは、その名のとおりWebアプリケーションを開発するためのフレームワークです。PhoenixはMVCモデルを採用しており、図1.2のような構造を持ちます。

● 図1.2 MVC



MVCはModel、View、Controllerの頭文字を取ったもので、Webアプリケーションの構造を表しています。Controllerはリクエストを受け取り、

ModelとViewを呼び出します。Modelはデータの永続化を行い、ViewはHTMLなどのレスポンスを生成してControllerに返却します。

また、WebSocketを利用したリアルタイム通信をPhoenix Channelsという機能で簡単に実装できます。そのほかにLiveViewという機能を利用することで、JavaScriptを一切書かずにリッチなフロントエンドを実現できるなど、さまざまな機能が用意されています。

■ Phoenix による Web アプリケーション開発

Phoenixを使うことで、高性能なWebアプリケーションを手早く開発できます。また、Phoenix Channelsにより、WebSocketを利用したリアルタイム通信を驚くほど簡単に実装できます。

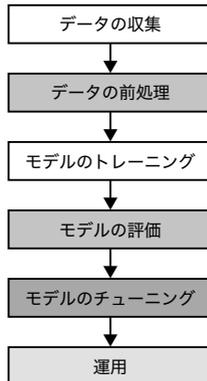
機械学習

第4次AIブームといわれる今日ですが、その基礎技術として機械学習は重要な位置を占めます。機械学習の処理を実装する際によく利用されるプログラミング言語はPythonですが、Elixirでも機械学習の処理を実装できます。

■ 機械学習アプリケーションの構造

機械学習アプリケーションの要素は図1.3のようになります。

● 図1.3 機械学習アプリケーションの要素



「データの収集」でデータ収集システムを構築し、「データの前処理」で収集データを機械学習のモデルに適した形式に変換します。「モデルのトレーニング」では前処理済みのデータを使った機械学習のモデル作成とトレーニング、「モデルの評価」で作成したモデルの性能評価、「モデルのチューニング」でテスト結果をもとにモデルを改善します。最後の「適用」でモデルを実際に利用する運用を行います。

これらの要素の共通点は、データの処理が中心になることです。つまり、Elixir で機械学習を実装する場合、データの処理を効率的に行うことができるライブラリが必要になります。

■ Elixir による機械学習

Nx は、Elixir の作者でもある José によって開発されている機械学習ライブラリです。多次元の配列を効率的に扱うことが可能で、CPU/GPU/TPU のバックエンドで計算を実行できます。Elixir で機械学習を行う場合、Nx は必須のライブラリと言えるでしょう。

また、Livebook という Jupyter Notebook のようなノートブック環境で Elixir を実行できます。Livebook は Elixir のコードを実行しながらその結果を表示でき、機械学習の処理を実装する際に非常に便利です。Elixir で機械学習を実装する場合、Livebook も必須のツールと言えるでしょう。

■ Nx のエコシステム

Nx の周辺には、機械学習の処理を効率的に実装するためのライブラリが多数存在します。Nx のエコシステムを利用することで、Elixir で機械学習を実装する際の生産性を高めることができます。

GitHub の Nx のページには、Nx の関連プロジェクトの一覧^{注8}が掲載されています。これらの中でも特に注目すべきプロジェクトとして Axon^{注9}があります。Axon は Nx を利用したニューラルネットワークライブラリで、数値関数の関数 API や、高レベルなモデル作成 API、トレーニング API を提供しています。

Nx と Axon については第 12～15 章で詳しく解説します。

注8 <https://github.com/elixir-nx>

注9 <https://github.com/elixir-nx/axon>

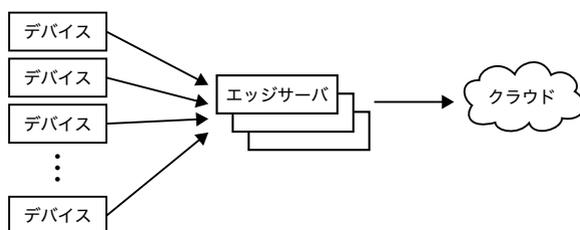
IoT デバイス

IoTはInternet of Thingsの略で、モノがインターネットを経由して通信することを指します。IoTデバイスとは、モノがインターネットを経由して通信するためのデバイスです。Elixirは、IoTデバイスの開発にも利用できます。

IoT アプリケーションの構造

IoTアプリケーションの要素は図1.4のようになります。

● 図1.4 IoTアプリケーションの要素



デバイスはセンサからデータを収集し、エッジサーバに送信します。エッジサーバは、デバイスから受信したデータをクラウドに送信します。クラウド上で、エッジサーバから受信したデータの処理を行います。

IoT と Elixir の親和性

先ほど述べたように、IoTアプリケーションはデバイス、エッジサーバ、クラウドの3つの要素から構成されます。デバイスは後述するNervesを、エッジサーバはPhoenixを、クラウドのデータ処理はNxを利用することで、IoTアプリケーションのほとんどをElixirで実装できます。

また、IoTアプリケーションは大量のデバイスから受信したデータを同時かつリアルタイムに処理し、エラーが発生してもシステム全体が停止することなくエラーを回復できる必要があります。これらの要件はElixirの強みである並行性と耐障害性によって実現できます。

このようにIoTとElixirは親和性が高く、Elixirを使うことでIoTアプリケーションを効率良く実装できます。

■ Nervesのエコシステム

ElixirのWebアプリケーションフレームワークとしてPhoenixがあるように、ElixirのIoTデバイス向けプラットフォームとしてNervesがあります。Nervesの構成要素は以下です。

- Erlang VMが起動できる最小構成のLinuxブートローダ
- Elixirの実行環境
- ビルド管理やファームウェア更新を行うためのコマンドラインツール

Nervesを使って実装したIoTデバイス上で動作するアプリケーションは、Elixirのプロジェクトとして管理されるので、Nerves周辺の機能はElixirのHexパッケージとして組み込むことができます。

たとえば、IoTデバイスでLEDを点灯させたい場合を考えてみます。LEDの操作APIを提供する `nerves_leds`^{注10} パッケージをNervesのプロジェクトに組み込むことで、IoTデバイス上でLEDを点灯させることができます。

Nervesについては第16章～第19章で詳しく解説します。

まとめ

本章ではプログラミング言語Elixirの成り立ちと言語的な特徴について解説しました。また、Webアプリケーション、機械学習、IoT開発におけるElixirの利用状況について紹介しました。

さまざまな分野でElixirが利用されるElixirの汎用性の高さを感じていただけただけでしょうか。

それでは、次章からさっそくElixirのプログラミングについて解説していきましょう。

注10 https://github.com/nerves-project/nerves_leds

