

Introducing **CRYSTAL** Programming Language

Crystal-JP

MakeNowJust 5t111111 at_grandpa AKJ msky

Introducing Crystal Programming Language

Crystal-JP (5t111111, AKJ, at grandpa, MakeNowJust, msky)

Table of Contents

| | |
|-------------------------|----|
| 1.はじめに | 1 |
| 1.1.コンパイラ言語 Crystal | 2 |
| 1.2. Crystal の型システム | 2 |
| 1.3. Null 安全性 | 3 |
| 1.4. マクロ | 4 |
| 1.5. Crystal の並行処理 | 4 |
| 1.6. C バインディング | 4 |
| 1.7. パッケージ管理 | 4 |
| 1.8. Crystal のコミュニティ | 4 |
| 1.9.まとめ | 5 |
| 2. Getting Started | 7 |
| 2.1. Crystal のインストール | 7 |
| 2.2. Hello Crystal | 10 |
| 2.3. Crystal を書くためのエディタ | 11 |
| 2.4. Crystal Playground | 13 |
| 2.5.まとめ | 14 |
| 3. 構文 | 15 |
| 3.1. Crystal と Ruby の違い | 15 |
| 3.2. 型システム | 16 |
| 3.3. typeof | 17 |
| 3.4. リテラル | 17 |
| 3.5. 変数 | 18 |
| 3.6. クラス・メソッド | 19 |
| 3.7. enum | 22 |
| 3.8. メソッド呼び出し | 23 |
| 3.9. 条件分岐・繰り返し | 24 |
| 3.10.まとめ | 24 |
| 4. マacro | 26 |
| 4.1. マacroとは | 26 |
| 4.2. マacroの利点 | 27 |
| 4.3. マacroの展開 | 28 |
| 4.4. 標準搭載のマacro | 29 |
| 4.5. マacroの文法 | 32 |
| 4.6.まとめ | 45 |
| 5. Shards | 47 |
| 5.1. shard を使う | 47 |
| 5.2. shard を作る | 54 |
| 5.3. テスト | 65 |

| | |
|------------------------------|-----|
| 5.4.まとめ | 74 |
| 6. Web開発 | 76 |
| 6.1. CrystalでのWeb開発の前提知識 | 76 |
| 6.2. KemalによるWeb開発 | 78 |
| 6.3.補注 | 90 |
| 7. CLI開発 | 92 |
| 7.1. CrystalでCLIツール | 92 |
| 7.2.自作のechoコマンド「myecho」を作る | 92 |
| 7.3.バージョン表示のオプション-vを実装する | 96 |
| 7.4.ヘルプ表示のオプション-h--helpを実装する | 100 |
| 7.5.prefixを付ける--prefixを実装する | 105 |
| 7.6.サードパーティ製のライブラリを使う | 109 |
| 7.7.まとめ | 115 |
| 8.著者紹介 | 116 |
| 8.1.5t111111 | 116 |
| 8.2.AKJ | 116 |
| 8.3.at_grandpa | 116 |
| 8.4.MakeNowJust | 116 |
| 8.5.msky | 117 |

1. はじめに

著者: 5t111111

この本はプログラミング言語 Crystal の入門書です。おそらくまとめた一冊の書籍としては世界初の初心者向け Crystal 本になっています。

Crystal が登場したのは2014年のことで、それからずっと「Fast as C, Slick as Ruby」というコンセプトで作られています。

つまり Crystal は、

- C 言語の速さ
- Ruby の書き味

を同時に持つ実用的なプログラミング言語を目指して開発されています。元々は Manas Technology Solutions 社によって2011年から開発されていましたが、現在はオープンソースのプロジェクトとして公開され有志の手によって精力的に開発が進んでいます。そして、とうとうバージョン1.0のリリースが見えてくるところまで完成度が高くなりました。

このように Crystal は新しいプログラミング言語ですので、本書を手に取っていただいた方の中には Crystal というプログラミング言語について初めて知ったという方もいるかもしれません。そこで、このはじまりの章では Crystal の特徴について概要を簡単に紹介していきます。

ただ、どのようなプログラミング言語かということに関しては、言葉で説明するよりもまず実際のプログラムを見てイメージをつかんでもらう方が早いですよね。次のプログラムは、Crystal でごくシンプルな HTTP サーバを実装したプログラムです。

```
require "http/server"

server = HTTP::Server.new do |context|
  context.response.content_type = "text/plain"
  context.response.print "Hello world, got #{context.request.path}!"
end

puts "Listening on http://127.0.0.1:8080"
server.listen(8080)
```

いかがでしょうか。プログラムの細かいところはまだわからなくて問題ありませんが、Ruby を使ったことがある方であれば Crystal のプログラムが一見してとても Ruby に似ていることに気づいたと思います。自ら「Slick as Ruby」と称していることからもわかるように、Crystal は Ruby にとてもよく似たシンタックスを持ったプログラミング言語です。しかし、Crystal は Ruby とは異なる独自の特徴をいくつも持っており、それによって「Fast as C」の高速性や型安全なプログラミングを実現しています。

ここから、その Crystal の特徴を簡単にお伝えしていきます。

1.1. コンパイラ言語 Crystal

Crystal の処理系はコンパイラとして提供されます。このことはつまり、Crystal のプログラムを実行するには、Crystal コンパイラでソースコードをコンパイルし実行ファイルを生成する必要があることを示しています。これは類似したシンタックスを持つ Ruby がコンパイル不要なインタプリタ型言語として作られていることとは対照的です。

NOTE

より正確に言うと、Ruby も実行する際に内部的にコンパイル処理を行なっているのですが、ここでは話を単純にするために実質的にインタプリタ型言語と分類しています

Crystal はコンパイラ型の言語となっていることで、実行時速度の高速化や次項でお伝えする静的な型システムを実現しています。

また、コンパイルすることで直接実行できるファイルが生成されるので、実行する環境に処理系を用意する必要がなくアプリケーションのデプロイが容易なことも利点の1つです。なお、本書では詳細には触れませんが、Crystal はコンパイラ基盤に LLVM を利用しており、各プラットフォームに最適化した実行ファイルの生成が可能になっています。

1.2. Crystal の型システム

Crystal は静的型付けのプログラミング言語です。これも、動的型付け言語の Ruby とは異なった Crystal の大きな特徴です。

プログラムの中でデータの型は静的に解析されチェックが行われます。このことによって、型の不一致などはコンパイル時に補足され、実行時にエラーになってしまふことを防止できます。

ただ、Crystal は強力な型推論の仕組みを備えているため、多くの場合はプログラマが自分で面倒な型指定をしなくてもよいようになっています。

例えば、次の簡単なプログラムを見てください。

```
def shout(x)
  # Int32 と String はどちらも 'to_s' メソッドを持っている
  x.to_s.upcase
end

foo = ENV["FOO"]? || 10

typeof(foo)      # => (Int32 | String)
typeof(shout(foo)) # => String
```

このプログラムは、変数 `foo` に、

- ・環境変数 `FOO` が存在すればその値の文字列 (`String` 型)
- ・存在しなければ `10` という数値 (`Int32` 型)

を代入し、`foo` の型と `shout(foo)` の型を `typeof` によって調査するものです。`shout` メソッドの中では、引数として渡されたデータを `to_s` と `upcase` というメソッドによって大文字の文字列に変換しています。

このとき、`String` も `Int32` もどちらも `to_s` という文字列変換メソッドを持っていることがコンパイラによって正しく推論されるため、型を指定していくなくてもエラーにはなりません。続いて `typeof` の結果を見てみましょう。`foo` の型は実行時まで `String` か `Int32` かが判定できないために両方の複合型（ユニオン型）になっています。一方、`shout(foo)` は戻り値の型から `String` だということが保証されるため、その型も `String` として正しく推論されていることがわかります。

1.3. Null 安全性

「Null 参照によるエラーの損害は10億ドル相当だ」という有名な言葉を聞いたことがある人は多いと思います。

NOTE [Null References: The Billion Dollar Mistake - Tony Hoare](#)

実際にプログラミングをしていても、おそらくほとんどの人が Null に対して何か操作しようとしてエラーになってつらい経験をしたことがあるのではないでしょうか。Java の `NullPointerException` などはよく話のネタになりますよね。

そこで、最近のモダンなプログラミング言語は Null 安全性を取り入れているものが多くなってきており、Crystal も例外ではありません。

次のプログラムを見てください。

`hello_world.cr`

```
if rand(2) > 0
    my_string = "hello world"
end

puts my_string.upcase
```

これは、`rand` メソッドによって `0 ~ 2` を生成し、その結果によって `if` 条件を判定しているものです。しかし、実行するまでその結果が「真」になるかどうかはわかりません。したがって、`my_string` が Null でないことをコンパイル時に保証することはできません。この場合は `upcase` メソッドが呼び出せるかどうかがわからないため、コンパイルは以下ののようなエラーになります。

```
$ crystal hello_world.cr
Error in hello_world.cr:5: undefined method 'upcase' for Nil (compile-time type is
(String | Nil))

puts my_string.upcase
          ^~~~~~
```

このようにして、Crystal では Null 安全性を保証しており、また、Null でない場合だけに実行する処理の記述なども書きやすい設計となっています。

1.4. マクロ

先述したように、Crystal は静的型付けのコンパイラ言語であり、それによって安全なプログラムの構築が可能になっています。その一方で Ruby のような動的型付け言語が得意な「実行時にプログラムの動作を変える」こと、いわゆるメタプログラミングや実行時リフレクションといったテクニックは制限されます。

しかし Crystal には強力なマクロの仕組みが搭載されており、それを活用することでかなり自由度の高いプログラムを書くことが可能です。

マクロについては後の章で詳しく触れます。

1.5. Crystal の並行処理

現在のプログラミングにおける大きな変化として、並行処理や並列処理が高い注目を集めようになったことがあげられるでしょう。進化した並列処理の機構を特徴とする Rust や Go 言語、そして非同期処理によってシェアを伸ばした Node.js を見ても、プログラミング言語に並行処理は欠かせないものとなっています。

本書では大きく触れませんが、Crystal では、ノン・プリエンプティブな軽量スレッドを使った並行処理が可能になっており、それをファイバーと呼んでいます。ファイバー間におけるデータのやり取りには Go 言語や Clojure のようにチャンネルを利用し、共有メモリやロックを意識しなくても使いやすいように設計されています。

1.6. C バインディング

本書の範囲を超てしまうので詳細は触れませんが、Crystal は C 言語ライブラリのバインディングを書きやすいので、C 言語の資産を Crystal から利用することも難しくありません。

1.7. パッケージ管理

現代的なプログラミングでは OSS などで提供されるライブラリをいくつも利用することがごく一般的ですが、数が多くなってくるとその依存関係を管理するのがとても大変になります。したがって、パッケージングのシステムがあるかどうかはプログラミング言語の利用しやすさにおいて大きな位置を占めています。

Crystal には Shards というパッケージ管理のシステムがあります。この Shards についても後の章で詳しく触れます。

1.8. Crystal のコミュニティ

プログラミング言語を使うときには、その言語自体の仕様はもちろんですが、その言語コミュニティが充実していることも同じくらい大事ですよね。ここでは、Crystal のコミュニティや公式の一次情報を得るためにのサイトを紹介します。

Crystal 公式サイト（英語）

<https://crystal-lang.org>

Crystal の公式サイトです。リリースノートやブログ、そして API ドキュメントがあります。英語ですが、基本的には一次情報はこのサイトを中心として見ていれば得ることができます。

Crystal の Google グループ（英語）

<https://groups.google.com/forum/#!forum/crystal-lang>

Crystal に関することで何かあれば、まずこの Google グループに書き込むことが推奨されています。例えば Crystal に関する質問や Crystal を使ったプロジェクトの紹介など、Crystal に関することであれば何でも OK です。活発にやりとりが行われています。

Gitter の Crystal 公式チャットルーム（英語）

<https://gitter.im/crystal-lang/crystal>

Crystal に関する Gitter の公式チャットルームです。チャットなので Google グループより気軽に質問などができる、レスポンスも早いことが期待できます。こちらもとても活発で、IRC との連携もされています。

Crystal の GitHub リポジトリ

<https://github.com/crystal-lang/crystal>

Crystal の開発は GitHub 上で行われています。Crystal の不具合を見つけたときに issues に報告したり、何か素晴らしい新機能や修正について直接 pull request を送ることもできます。

Crystal-JP の Slack（日本語）

<http://crystal.pine.moe>

日本の Crystal ユーザコミュニティ「Crystal-JP」の Slack への招待ページです。Crystal-JP の Slack では日本語で Crystal の情報交換を行うことができます。お気軽に参加ください。また、Crystal-JP では不定期に（今のところ東京のみですが）イベントを開催しており、そのお知らせなども主にこの Slack で告知されます。

また、Crystal のライブラリなどを探したい場合には [Awesome Crystal](#) や [CrystalShards](#) が参考になるでしょう。

1.9. まとめ

本章では、Crystal の特徴を簡単に説明しました。Crystal がどのようにして「 C 言語のように速く、Ruby の書き味を持つ」ということを実現しているかの概要だけでも把握していただき、Crystal の魅力を感じていただけたら幸いです。

次の章からはより実践的な Crystal のプログラミングについて説明していきます。

最後になりますが、本書で利用している Crystal のバージョンは執筆時点で最新の [0.26.1](#) です。異なるバージョンの Crystal では掲載されているプログラムが期待通りに動作しない可能性もあります。予めご了承ください。

2. Getting Started

著者: 5t111111

「はじめに」の章で説明したように、Crystal はコンパイラ言語なので、プログラムのソースコードをコンパイルするためのコンパイラが必要になります。

この章ではまず、Crystal を実際に使っていくための様々なプラットフォームでのインストール方法を紹介します。

そして、プログラムを書くために大事なエディタのサポート状況や、Web ブラウザ上で Crystal プログラムを実行することのできる Playground というツールについても紹介します。

2.1. Crystal のインストール

Crystal のインストール方法は OS ごとに異なります。ここでは、主要な Linux ディストリビューションや macOS、そして Windows へのインストール方法を紹介します。

2.1.1. Debian や Ubuntu

Debian 系のディストリビューションでは、Crystal の公式リポジトリを利用することができます。

そのために、まず APT の構成にリポジトリを追加します。以下のコマンドを実行してください。

```
$ curl -sSL https://dist.crystal-lang.org/apt/setup.sh | sudo bash
```

これによりリポジトリが構成されキーが登録されます。もし上記を自分で設定するのがお好みの場合は、以下を実行しても同じことができます。

```
$ curl -sL "https://keybase.io/crystal/pgp_keys.asc" | sudo apt-key add -
$ echo "deb https://dist.crystal-lang.org/apt crystal main" | sudo tee
/etc/apt/sources.list.d/crystal.list
$ sudo apt-get update
```

リポジトリを構成したら以下で Crystal がインストールできます。

```
$ sudo apt install crystal
```

以下に記載するパッケージは必須ではありませんが、標準ライブラリの機能を使う上で必要となるためインストールすることを推奨します。

```
$ sudo apt install libssl-dev      # OpenSSL に必要です  
$ sudo apt install libxml2-dev    # XML に必要です  
$ sudo apt install libyaml-dev    # YAML に必要です  
$ sudo apt install libgmp-dev     # Big numbers に必要です  
$ sudo apt install libreadline-dev # Readline に必要です
```

なお、新しいバージョンの Crystal がリリースされた場合は以下でアップグレードすることができます。

```
$ sudo apt update  
$ sudo apt install crystal
```

2.1.2. Red Hat や CentOS

Red Hat 系のディストリビューションでも、Crystal の公式リポジトリを利用することができます。

まず YUM の構成にリポジトリを追加します。以下のコマンドを実行してください。

```
$ curl https://dist.crystal-lang.org/rpm/setup.sh | sudo bash
```

これでリポジトリが構成されキーが登録されますが、以下のように、同様のことを手動で行うことも可能です。

```
$ rpm --import https://dist.crystal-lang.org/rpm/RPM-GPG-KEY  
  
$ cat > /etc/yum.repos.d/crystal.repo <<END  
$ [crystal]  
$ name = Crystal  
$ baseurl = https://dist.crystal-lang.org/rpm/  
$ END
```

リポジトリを構成したら以下で Crystal がインストールできます。

```
$ sudo yum install crystal
```

なお、新しいバージョンの Crystal がリリースされた場合は以下でアップグレードすることができます。

```
$ sudo yum update crystal
```

2.1.3. Arch Linux

Arch Linux では、ユーザリポジトリから Crystal をインストールできます。パッケージの依存関係を管理するための `shards` も必要になるので同様にインストールしてください。

```
$ sudo pacman -S crystal shards
```

2.1.4. Gentoo Linux

Gentoo Linux では、メインの overlay に Crystal が含まれています。

以下で構成のフラグを確認することができます。

```
# equery u dev-lang/crystal
[ Legend : U - final flag setting for installation]
[       : I - package is installed with flag      ]
[ Colors : set, unset                         ]
* Found these USE flags for dev-lang/crystal-0.18.7:
U I
-- doc      : Add extra documentation (API, Javadoc, etc). It is recommended to
enable per package instead of globally
-- examples : Install examples, usually source code
++ xml      : Use the dev-libs/libxml2 library to enable Crystal xml module
+- yaml     : Use the dev-libs/libyaml library to enable Crystal yaml module
```

以下を実行してインストールしてください。

```
$ su -
# emerge -a dev-lang/crystal
```

2.1.5. macOS

macOS では、[Homebrew](#) を利用して簡単に Crystal をインストールすることができます。

```
$ brew update
$ brew install crystal
```

なお、macOS で Crystal を使うときに以下のエラーに遭遇することがあります。

```
ld: library not found for -levent
```

その場合は、以下のようにコマンドラインツールを再インストールした上で、デフォルトのツールチェインを選択し直す必要があります。

```
$ xcode-select --install
$ xcode-select --switch /Library/Developer/CommandLineTools
```

2.1.6. Linuxbrew

[Linuxbrew](#) を使って Linux に Crystal をインストールすることもできます。

```
$ brew update  
$ brew install crystal-lang
```

もし Crystal の言語自体を開発することにも興味があるのであれば、同時に LLVM もインストールしておくと良いでしょう。その場合は上記の2行目を以下に変更します。

```
$ brew install crystal-lang --with-llvm
```

2.1.7. Windows

残念ながら、まだ Crystal は Windows での実行をサポートしていません。ですが、[Windows Subsystem for Linux](#) を利用することで Windows 10 上で Crystal を使うことが可能です。

WSL 上での Crystal のインストール方法は、それぞれの Linux ディストリビューションにおけるインストール方法と同様です。例えば、もし WSL に Ubuntu を導入したのであれば、前掲の「 Debian や Ubuntu 」でのインストール方法を参照してください。

2.1.8. その他のインストール方法

ここまで、プラットフォームごとのインストール方法を説明しましたが、自分の使っている環境が対応していない場合や、より最新のバージョンを使いたい場合のインストール方法を紹介します。入門書の範囲を超えてしまったため、詳細についてはリンク先をご覧ください。

- [tar ボールからのバイナリインストール](#)
- [ソースコードからのビルド](#)

2.2. Hello Crystal

2.2.1. インストールの確認

Crystal のインストールができたら、以下のコマンドを実行してみましょう。これで、Crystal がインストールされて利用できる状態かどうかが確認できます。

`hello.cr`

```
$ crystal -v
```

正常にインストールされている場合、

```
Crystal 0.26.1 (2018-08-27)
```

といったバージョン情報が表示されます。もし「コマンドが見つからない」といったエラーが表示された場合は、

- Crystal のインストールでエラーが発生していないか
- Crystal をインストールした場所にパスが通っているか

を確認してください。

2.2.2. はじめての Crystal プログラム

無事にインストールして使える状態になっていたら、はじめての Crystal のプログラムを書いてみましょう。ご多分に漏れず、ここでも最初のプログラムは「Hello world」とします。

好きなエディタで、以下のプログラムを書いて `hello.cr` として保存してください。Crystal の拡張子は `.cr` です。

```
puts "Hello world!"
```

それではプログラムを実行してみましょう。

```
$ crystal hello.cr
```

実行して、

```
Hello world!
```

と表示されたら成功です！

Crystal はコンパイラ言語ですが、このように1つのコマンドでコンパイルと実行を同時にを行うことができます。

```
$ crystal run hello.cr
```

と実行しても同様です。

2.3. Crystal を書くためのエディタ

さて、これで Crystal を書くための準備が整ったので、次の章からは文法など実際にプログラムを書くための内容に入っていきます。ですが、プログラムを快適に書くためにはエディタのサポートも欠かせません。

この章では最後に、以下の代表的なオープンソースエディタの Crystal サポートの状況について簡単に紹介します。

- Visual Studio Code

- Atom
- Vim
- Emacs

2.3.1. Visual Studio Code

Visual Studio Code、通称 VSCode は機能の豊富さとシンプルさ軽さを両立したエディタとしてとても人気のエディタです。

VSCode で Crystal プログラミングをサポートするエクステンションにはいくつか種類がありますが、現在最も活発に開発されているのは Crystal Language というものです。GitHub 上では [crystal-lang-tools/vscode-crystal-lang](#) というリポジトリで開発されています。

この Crystal Language エクステンションをインストールするだけで、

- シンタックスハイライト
- 自動インデント
- 自動フォーマット
- エラー検知
- 定義ジャンプ

などの機能がすぐに利用できるようになります。

vscode-crystal-lang の GitHub 上の Wiki にはより詳しい設定などの情報も記載されています。例えば、エラーの検知レベルをカスタマイズしたり、[scry](#) という Crystal の Language Server との連携などにより便利に使うための設定も書かれていますので合わせてご覧ください。

2.3.2. Atom

GitHub 社製のエディタ Atom では、以下のパッケージを導入することで快適に Crystal プログラミングができます。

- [crystal-lang-tools/language-crystal](#)
- [crystal-lang-tools/atom-ide-crystal](#)

language-crystal は Atom のパッケージとしては language-crystal-actual という名前で提供されているので注意が必要です。これをインストールすると Crystal のシンタックスハイライトや自動インデントがサポートされます。

atom-ide-crystal は、前節でも触れた scry という Language Server と連携して IDE のような機能を追加するものです。まだ機能的には充実していない面もありますが、エラー検知をサポートするなど少しづつ開発が進んでいます。

2.3.3. Vim

言わずと知れた Vim でも、プラグインを使うことで Crystal プログラミングにエディタのサポートを受けることができます。

[rhysd/vim-crystal](#) をインストールすると Crystal の filetype が追加され、シンタックスハイライトや自動インデントが有効になります。

また、vim-crystal には Crystal の組み込みツール `crystal tool` や Spec との連携などの便利な機能があり、シームレスにプログラミングしやすくなっています。

エラー検知をしたい場合には、[vim-syntastic/syntastic](#) などの統合解析プラグインを導入が必要です。前述の vim-crystal にはこの syntastic 用のチェックもバンドルされているため、特別な設定をせずとも利用することができます。

2.3.4. Emacs

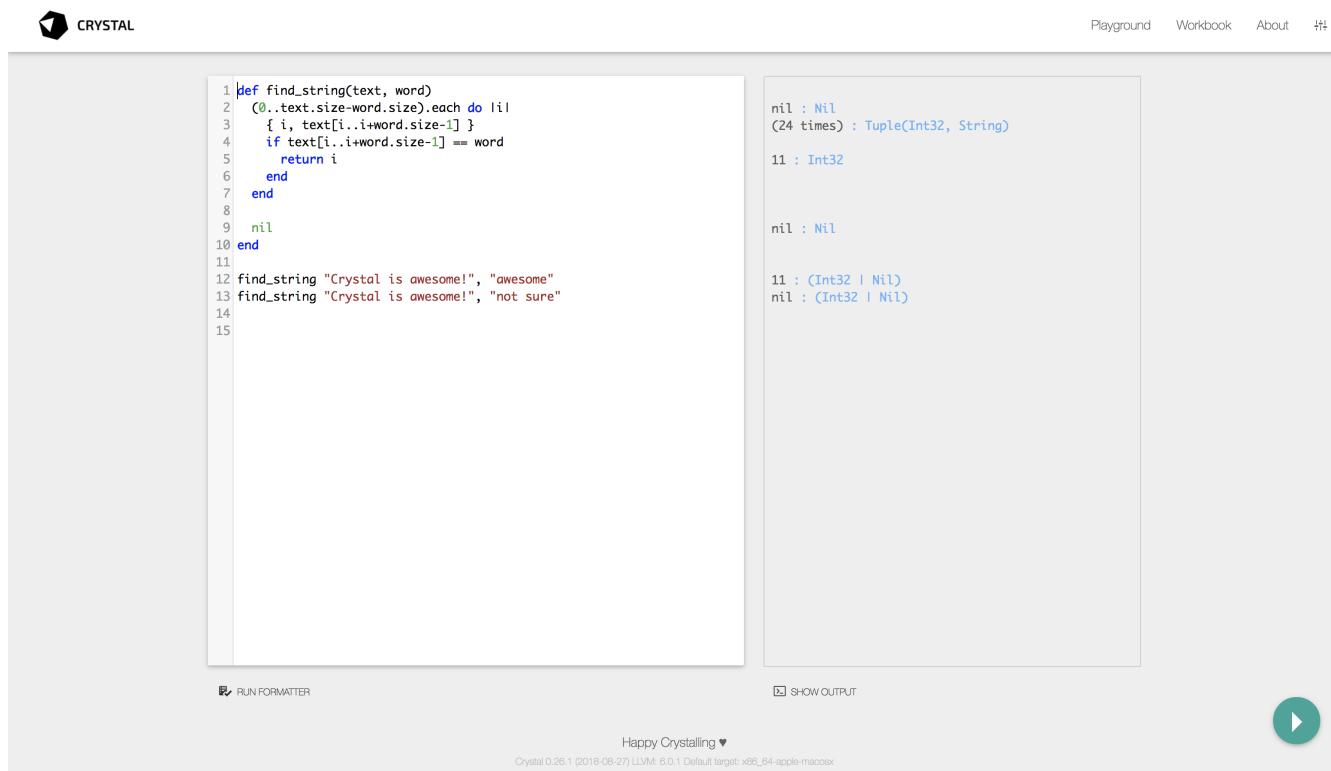
Emacs で Crystal のプログラムを書く場合は [crystal-lang-tools/emacs-crystal-mode](#) を使うのがよいでしょう。

このパッケージを導入するとシンタックスハイライトや自動インデントがサポートされ、定番のチェックツール flycheck にも Crystal のサポートが追加されます。

emacs-crystal-mode パッケージは [MELPA](#) に登録されているので導入も簡単です。

2.4. Crystal Playground

Crystal Playground を使うと Web ブラウザ上で Crystal のコードを実行することができます。



この画像を見るとわかるように、Playground では、

- 実行結果
- 型情報

などを同時に表示しながら手軽に Crystal プログラミングを楽しむことができます。

Playground は Crystal に同梱されているため、Crystal がインストールされていれば、

```
$ crystal play
```

と実行するだけで Playground サーバが立ち上がり、すぐにアクセスして使うことができます。デフォルトではポート 8080 で起動するので、お気に入りの Web ブラウザで <http://localhost:8080> にアクセスしてください。ポートは `-p` オプションでカスタマイズ可能です。

Playground を立ち上げておけば、本書を読み進めるときに、記載されているコードをその場で実行して試したいときにも便利に活用できることでしょう。

2.5. まとめ

この章では、Crystal のインストール方法とエディタのサポート状況、そして Playground について紹介しました。まだ発展途上であり入門書の範囲を超てしまうのでここでは記載しませんが、もし Crystal に慣れてきたらエディタと連携したデバッグ方法なども調べてみるとよいでしょう。

次の章では Crystal の構文について説明します。

3. 構文

著者: MakeNowJust

この章では Crystal の構文について説明します。

Crystal の構文を全て説明すると膨大になってしまふので、ここでは Ruby との違いに焦点を置いて解説したいと思います。

というのも、Crystal の構文は Ruby の影響を強く受けており、多くの場合 Ruby のように書くことで Crystal のプログラムを書くことができます。

しかし、やはり Crystal と Ruby は別のプログラミング言語であり、構文の異なる部分もいくつか存在します。

既に知識があるのであれば、その違いを抑えていくのが Crystal の構文を理解する手助けになるでしょう。

Ruby の構文については次のサイトを参考にしてください。

オブジェクト指向スクリプト言語 Ruby リファレンスマニュアル

<https://docs.ruby-lang.org/ja/latest/doc/index.html>

また、Crystal の完全な構文は、公式サイトにある次のドキュメントを参考にしてください。

Syntax and Semantics

https://crystal-lang.org/docs/syntax_and_semantics/

3.1. Crystal と Ruby の違い

Crystal と Ruby は構文こそよく似ていますが、言語としては次のような大きな違いがあります。

1. Ruby はインタープリタで実行されるが、Crystal はコンパイルして実行する。
2. Ruby には変数に型が無いが、Crystal には型がある。

そして、この2が Crystal と Ruby の構文に違いをもたらしています。

前者を実感する例としては、こんなものがあります。

これは有効な Ruby のプログラムですが、Crystal ではコンパイルエラーになります。

Crystal は `require` で読み込むファイルまで含めてコンパイルしなければなりません。

なので `require` は Ruby のようなメソッドではなく、構文として提供されていて、引数は固定の文字列ではなければいけないです。

```
# Ruby のプログラム！

Dir["foo/*.rb"].each do |file|
  require file
end
```

後者を実感する例としては、こんなものがあります。
これは Crystal のプログラムです。

メソッドの引数に型を指定しているところに注目してください。
引数の型でメソッドをオーバーロードすることができます。
これは Ruby ではできません。

```
# Crystal のプログラム！

# `foo` の引数が `String` の場合のメソッド
def foo(x : String)
  p :string
end

# `foo` の引数が `Int32` （数値）の場合のメソッド
def foo(x : Int32)
  p :int32
end

# 呼び出し時に引数の型によって適切なものが選択される
foo "string"
foo 42

# output:
# :string
# :int32
```

それでは、Crystal と Ruby で構文の異なる部分を説明していきます。

これらの違いを意識しながら読み進めていってください。

3.2. 型システム

はじめに Crystal の型システムについて簡単に説明しておきます。

Crystal の型には次のようなものがあります。

- 通常の型
 - `Int32`（整数）や `String`（文字列）、`Nil`、`Bool`など
- ジェネリックス
 - `Array(Int32)`（要素の型が `Int32` の配列）

- `Array(String)` (要素の型が `String` の配列)
- `Hash(String, Int32)` (キーの型が `String` で値の型が `Int32` のハッシュ)
- ユニオン
 - `Int32 | String` (`Int32` か `String` 型)
 - `Int32?` (`Int32 | Nil` の糖衣構文)
- Proc
 - `Int32 → String` (引数に `Int32` を受け取って `String` を返す Proc)

こんなものがあるんだな、となんとなく覚えておいてください。

3.3. `typeof`

Ruby には無い構文として `typeof` というものが Crystal にはあります。

これは引数として与えられた式の結果についていた型を返す構文です。

引数の式はコンパイル時にのみ利用され、実行時には利用されないように注意してください。

```
# 文字列リテラルの型は `String`  
typeof("foo") # => String  
  
# 引数の式は実行されない  
typeof(puts "hello") # => Nil  
# hello とは出力されない
```

3.4. リテラル

リテラル関連で Ruby と大きく異なるところは、次のものが挙げられます。

- 数値リテラルの型指定
- 空の配列と空のハッシュに対する型指定
- タプルと名前付きタプル

逆に、これ以外は一部の例外を除いて Ruby と同じように書くことができます。

一部の例外としては次のものが挙げられます。

NOTE

- いくつかの `%` 形式のリテラル (`%s` や `%W` など) が存在しない。
- ハッシュの `{foo: bar}` のような形式は名前付きタプルの構文となっている。
- 正規表現の構文がPCREになっている。

3.4.1. 数値リテラルの型指定

Ruby の数値は基本的には `Integer` と `Float` だけです。

しかし、Crystal の数値はその大きさや符号の有無によって `Int32` · `Int64` · `UInt32` · `Float32` · `Float64` などが存在します。

`Int32` は32ビットの符号付き整数型で、`UInt32` は32ビットの符号無し整数型、`Float64` は64ビットの浮動小数点型です。

そして、数値リテラルの末尾に `i32` · `i64` · `f32` · `f64` などと付けることによって、値の型を指定できます。

もちろん数値リテラルの末尾に何も指定しないことも可能で、その場合は整数なら `Int32` 型、小数なら `Float64` 型になります。

```
# 何も指定していない数値は `Int32`、 `Float64` 型
typeof(123) # => Int32
typeof(3.14) # => Float64

# 符号付き整数型
[typeof(123i8), typeof(123i16), typeof(123i32), typeof(123i64), typeof(123i128)]
# => [Int8, Int16, Int32, Int64, Int128]

# 符号無し整数型
[typeof(123u8), typeof(123u16), typeof(123u32), typeof(123u64), typeof(123u128)]
# => [UInt8, UInt16, UInt32, UInt64, UInt128]

# 浮動小数点型
[typeof(3.14f32), typeof(3.14f64)] # => [Float32, Float64]
```

3.4.2. 空の配列と空のハッシュに対する型指定

空の配列は `[]` 、空のハッシュは `{}` のように書けますが、これだと要素やキーの型が分からぬいためコンパイルすることができません。

そこで Crystal では、空のリテラルに `of` 型 と続けることで型を指定します。

ハッシュの場合は `of キーの型 => 値の型` になります。

```
# 要素の型が `Int32` の空の配列を作成
typeof([] of Int32) # => Array(Int32)

# キーの型が `String` 、要素の型が `Int32` の空のハッシュを作成
typeof({} of String => Int32) # => Hash(String, Int32)
```

3.4.3. タプルと名前付きタプル

これは可変長引数、名前付き引数と関連の深い概念なので、そこで説明します。

3.5. 変数

変数名は小文字から始めなければならず、定数は大文字から始めなければいけません。

インスタンス変数は `@` から、クラス変数は `@@` から始めなければいけません。

これらは Ruby と同様です。

しかし Crystal にグローバル変数はありません。
代わりにクラス変数や定数を使ってください。

3.6. クラス・メソッド

クラス・メソッド関連で Ruby と大きく異なるところは、次のものが挙げられます。

- メソッドの型指定・オーバーロード・`previous_def`
- 名前付き引数
- 可変長引数の扱い
- インスタンス変数の型
- `struct`

また、Crystal ではコンパイル時に全てのメソッドが定義されていなければいけません。
なので Ruby の `define_method` のようなことはできません。

3.6.1. メソッドの型指定・オーバーロード・`previous_def`

メソッドの型指定・オーバーロードは前述しましたが、異なる引数の型を持った同名のメソッドを定義すると、呼び出し時に適切なものが選択される、という機能です。

また、このときに引数の型が一致するメソッドが見つからなかった場合、コンパイルエラーになります。

`previous_def` は反対に、一致するメソッドが複数見つかってしまった場合に使う機能です。

この場合は、まず一番最後に定義されたものが呼び出されます。

そして、その中で `previous_def` を使うと、次に定義されたものが呼び出されるのです。

ちなみに、引数の型は指定しないこともできます。

その場合は任意の型を受け取ることになります。

ですが、実際に呼び出された引数が持っていないメソッドを呼び出していた場合は、コンパイルエラーになります。

```
# 最初に定義したもの
def foo(x : Int32)
  1 + x
end

# 次に定義したもの
# こちらが呼ばれる
def foo(x : Int32)
  previous_def + 2 # ここの `previous_def` で上の `foo` が呼ばれる
end

foo(3) # => 6
```

3.6.2. 名前付き引数

名前付き引数とは、Rubyではキーワード引数と呼ばれるものです。

Crystalでは、全ての引数を名前付き引数として呼び出すことができます。

他にも、名前付き引数として指定するための名前と、実際に引数として受け取る変数の名前を分けることができます。

これは、名前付き引数の名前として予約語を使いたいときに便利です。

次のようなメソッドを定義した場合、

```
# 割り算をするメソッド
def div(left, right)
    left / right
end

# 引数名を空白で二つ並べると、
# 一つ目が名前付き引数用の名前、二つ目が内部で使う引数の名前になる
def even(if cond)
    cond
end
```

このように名前付き引数を使ってメソッドを呼び出せます。

```
# 名前付き引数を指定
div(left: 1.0, right: 2.0) # => 0.5
# 順番を逆にして指定
div(right: 2.0, left: 1.0) # => 0.5

# 'cond' ではなく 'if' を名前付き引数に指定する
even if: true # => true
```

3.6.3. 可変長引数

Ruby同様、引数名の前に`*`を置くと可変長引数を受け取るものとして、`**`を置くと名前付き引数の余った引数を受け取るものとしてマークされます。

Rubyでは可変長引数は配列を、キーワード引数の余りはハッシュを受け取ります。

ですが、Crystalでは可変長引数ではタプル（`Tuple`）に、名前付き引数の余りは名前付きタプル（`NamedTuple`）になります。

タプルは配列に似ていますが、固定長・変更不可であり、各要素の型を保持しているのが特徴です。また名前付きタプルもキーがシンボルのハッシュに似た型で、タプルと同様に変更不可で各シンボルのキーに対応する型を保持しているのが特徴です。

splat 展開の際にも、これらを渡します。

splat 展開も Rubyと同様の構文で、メソッド呼び出しで引数の前に`*`を置いたものが可変長引数のsplat

展開となります。また、引数の前に `**` を置いたものは名前付き引数の splat 展開になります。

実際に可変長引数を受け取るメソッドの例です。

```
# 可変長引数・名前付き引数の余りを受け取ってその型を返すメソッド
def vararg_type(*tuple, **named_tuple)
  [typeof(tuple), typeof(named_tuple)]
end
```

この実行結果は次のようになります。

```
# 可変長引数の場合
vararg_type(42, "str")
# => [Tuple(Int32, String), NamedTuple()]

# 名前付き引数の場合
vararg_type(foo: 42, bar: "str")
# => [Tuple(), NamedTuple(foo: Int32, bar: String)]

# splat 展開
tuple = {42, "str"} # タプルのリテラルは配列の '{ }' 版
# 名前付きタプルはハッシュの `:` 区切り版
named_tuple = {
  foo: 42,
  bar: "str",
}
vararg_type(*tuple, **named_tuple)
# => [Tuple(Int32, String), NamedTuple(foo: Int32, bar: String)]
```

3.6.4. インスタンス変数の型

Crystal ではインスタンス変数・クラス変数の型がコンパイル時に決定できなければいけません。

`initialize` メソッドの中でインスタンス変数に代入している場合などは気を利かせて型を推論してくれます。

しかし、そうでない場合は型が分からずにコンパイルエラーになることがあります。

その場合は明示的にインスタンス変数の型を指定してください。

また、メソッドの引数名としてインスタンス変数を指定すると、メソッドの呼び出しと同時に、そのインスタンス変数に代入することができます。

```

class Foo
  # 引数にインスタンス変数を指定すると、その変数に代入される。
  # 加えて、引数の型やデフォルト引数を指定すると、インスタンス変数の型が
  # その型に推論される。
  def initialize(@foo : Int32, @bar = "bar")
    # こういう単純な代入の場合もインスタンス変数の型は推論される。
    @baz = Array(Int32).new
  end

  # インスタンス変数に対するゲッターメソッドを定義
  getter foo, bar, baz
end

foo = Foo.new 42

typeof(foo.foo) # => Int32
typeof(foo.bar) # => String
typeof(foo.baz) # => Array(Int32)

```

3.6.5. struct

`class` とよく似たものとして `struct` があります。

`struct` も `class` とほぼ同等の機能を持っていますが、メモリ確保に違いがあります。
`class` で定義した型のインスタンスはヒープに置かれますが、`struct` はスタックに置かれます。
このため `struct` の方が高速にインスタンスを作ることができます。

しかし、`struct` は自分自身をインスタンス変数を持つことができないという制約があります。

3.7. enum

Crystal には `enum` があります。

これは連番の数値型に分かりやすい名前を付けたもので、さらにメソッドを定義することもできます。

そして、`@[Flags]` 属性を付けると、単なる連番ではなく値はビットフラグになります。

また、`enum` にはオートキャストという機能もあります。引数の型制約に `enum` を指定して、シンボルが渡されたときに、自動でシンボルから `enum` へ変換するというものです。シンボルが `enum` として有効な値かどうかはコンパイル時にチェックされるので、普通にシンボルを使う場合よりも安全です。

```

# 色を表す `enum` を定義
enum Color
  Red
  Green
  Blue

# 値を直接指定することもできる
Yellow = 100
end

def color_name(color : Color)
  color
end

# ファイルの開く際にモードを表す `enum` を定義
@[Flags]
enum FileMode
  Read
  Write
end

# `enum` は定数と同じように `::` で参照する。
Color::Red # => Red

# オートキャストが発動して、`:green` が `Color::Green` に変換される。
color_name(:green) # => Green

# `@[Flags]` のついた `enum` は `|` で複数指定できる。
FileMode::Read | FileMode::Write # => Read | Write

```

3.8. メソッド呼び出し

メソッド呼び出しの構文はほとんど Ruby と同じですが、1つだけ異なる点があります。

Crystal には一引数ブロックの省略記法というものがあります。

これは ブロックの第一引数に対してメソッドを呼び出す場合に **`&.メソッド名`** のように書けるという構文です。

さらに、そこからメソッドチェインを始めることができるため、場合によってはとても便利です。

```

# '42' を `yield` するメソッド
def yield_42
  yield 42
end

yield_42 { |x| x.to_s.size } # => 2
# ↑を次のように書き直せる
yield_42 &.to_s.size # => 2

```

3.9. 条件分岐・繰り返し

条件分岐・繰り返しの構文は Ruby とほとんど同じです。

ただし `redo` はありません。

条件分岐の条件に変数が対象になっている場合、その変数の型がフィルタされます。

例えば、次のコードを考えてみましょう。

```
foo = rand > 0.5 ? "foo" : nil  
  
①  
if foo  
②  
else  
③  
end
```

- ① `foo` の型は `String | Nil`
- ② この位置に来たら `foo` は確実に `String` 型
- ③ この位置に来たら `foo` は確実に `Nil` 型

ということが分かると思います。

このように、条件分岐の条件によって、ブロック内で変数の型がいい感じに変化するのです。

型のフィルタに使える特殊な構文には次のものがあります。

`is_a?`

`foo.is_a?(String)` は変数 `foo` が `String` 型のとき `true` になります。

`nil?`

`is_a?(Nil)` の省略形です。

`responds_to?`

`foo.responds_to?(:size)` は変数 `foo` がメソッド `size` を持っているときに `true` になります。

ちなみにこれらの構文はメソッドのような見た目ですがメソッドではないため、オーバライドなどはできないことに注意してください。

また、これらを `&&` や `||` や `!` で組み合わせたものも動作します。

3.10. まとめ

説明しなかった構文はいくつもありますが、この辺りの構文を覚えておけば Crystal のソースコードがそれなりに読めるようになるはずです。

説明しなかったのは、

- ジェネリックスの構文
- C言語と連携のための構文
- マクロ
- アノテーション

などです。

これらは高度な機能なので、当分は知らなくても問題がないでしょう。

ちなみに、マクロについては次の章で詳しく解説されるはずです。

また、分からぬ構文はあれば最初に挙げた Crystal のドキュメントを確認してみるといいでしょう。
これよりも詳細に書かれているはずです。

Syntax and Semantics

https://crystal-lang.org/docs/syntax_and_semantics/

加えて、標準ライブラリの API ドキュメントは以下にあります。

知らない型やメソッドが出てきたときに確認してみてください。

API ドキュメント

<https://crystal-lang.org/api/>

4. マクロ

著者: at_grandpa

この章では、Crystal のマクロについて説明します。

4.1. マクロとは

Crystal のマクロは次のようなものです。

- ・「Crystal のコードを書く」コード
- ・コンパイルフェーズで実行され、Crystal のコードに展開される
- ・全マクロが展開されたあとの Crystal コードが実際にコンパイルされる

これだけではイメージが湧きづらいので、マクロがどのようなものかを実際に見てみましょう。次のコードを見てください。

```
macro my_macro(method_name, content)
```

```
  def {{method_name}} ①
```

```
    {{content}}
```

```
  end
```

```
end
```

```
my_macro(my_method, "hoge") ②
```

```
my_method ③
```

```
# => "hoge"
```

1. `macro` を用いてマクロを定義します

2. 定義したマクロを呼び出します

a. 引数 `my_method, "hoge"` がマクロに渡されます

b. 引数をもとに処理が行われ、呼び出し箇所に Crystal コードが展開されます

3. Crystal コードに展開された後、通常のコンパイルが行われます

つまり、マクロ展開後は次のようになります。

```
def my_method
```

```
  "hoge"
```

```
end
```

```
my_method # => "hoge"
```

単純なメソッド定義とメソッド呼び出しに展開されています。その後、実際のコンパイルが行われます。マクロのイメージが湧きましたでしょうか。

4.2. マクロの利点

マクロを利用することで、コードの重複を排除できます。次のコードを見てください。

```
class User
  def initialize(@name : String, @age : Int32)
  end

  def name
    @name
  end

  def age
    @age
  end
end

user = User.new("Taro", 30)
user.name # => "Taro"
user.age # => 30
```

典型的な getter メソッドです。`name` と `age` が似たようなメソッドになっています。マクロでこの重複を除去しましょう。

```
class User
  def initialize(@name : String, @age : Int32)
  end

  # macroの定義
  macro my_getter(*names)
    {%
      for name in names %
        def {{name.id}}
          @{{name.id}}
        end
      {%
        end %
    end

  # macroの呼び出し
  my_getter name, age
end

user = User.new("Taro", 30)
user.name # => "Taro"
user.age # => 30
```

マクロを定義し、そのマクロを呼び出しました。一見、元のコードよりも複雑になったように見えます。しかし、今後インスタンス変数が増えたとしても、マクロの呼び出し引数にその名前を渡すだけでよくなります。重複を排除できました。

実は、今回のような `getter` のマクロは、標準すでに搭載されています。よって、上記のコードは次のように書くことができます。

```
class User
  def initialize(@name : String, @age : Int32)
  end

  getter name, age
end

user = User.new("Taro", 30)
user.name # => "Taro"
user.age # => 30
```

かなりすっきりしました。このように、マクロを利用することですっきりとしたコードを書くことができます。

4.3. マクロの展開

重複の除去によって、マクロ呼び出しのコードはすっきりしました。しかし、マクロ定義のコードはどうしても複雑になってしまいます。マクロの理解に加え、展開後の Crystal コードも理解しなければならないからです。

Crystal のバージョン `0.20.4` 以前は、マクロ展開後のコードを知るすべはありませんでした。唯一のヒントは、エラーメッセージだけでした。しかし、Crystal のバージョン `0.20.5` から `crystal tool expand` コマンドが追加されました。

```
$ crystal tool expand --help
Usage: crystal tool expand [options] [programfile] [--] [arguments]

Options:
  -D FLAG, --define FLAG           Define a compile-time flag
  -c LOC, --cursor LOC            Cursor location with LOC as
path/to/file.cr:line:column
  -f text|json, --format text|json Output format text (default) or json
  --error-trace                   Show full error trace
  -h, --help                      Show this message
  --no-color                      Disable colored output
  --prelude                       Use given file as prelude
  -s, --stats                     Enable statistics output
  -p, --progress                  Enable progress output
  -t, --time                       Enable execution time output
  --stdin-filename                Source file name to be read from STDIN
```

`--cursor` オプションでカーソル位置を指定すると、カーソル上のマクロを展開した結果を表示することができます。先程の `getter` で試してみましょう。

```
$ crystal tool expand --cursor /path/to/getter.cr:5:3 /path/to/getter.cr
1 expansion found
expansion 1:
getter(name, age)

# expand macro 'getter' (/path/to/crystal-lang/src/object.cr:230:3)
~> def name
  @name
end
def age
  @age
end
```

マクロが展開されました。意図していた定義です。このコマンドはエディタから実行できるようにすると便利です。設定方法は、エディタそれぞれの方法を参照してください。この `crystal tool expand` のおかげで、マクロのデバッグが格段にしやすくなりました。マクロを記述する際はぜひ活用してみてください。

4.4. 標準搭載のマクロ

Crystal には、標準で搭載されているマクロがあります。便利なものが多いのでいくつかご紹介します。`crystal tool expand` を用いれば内容を把握できます。また、公式の API ドキュメントの各マクロの説明には、そのマクロの定義へのリンクがあるので、興味のある方は確認してみてください。

4.4.1. `def_equals`

オブジェクトの同値性比較を行う `==` メソッドを定義します。同値性比較を行う場合、複数あるインスタンス変数の比較を行います。通常の場合、コードは次のようにになります。

```
struct User
  def initialize(@name : String, @age : Int32)
  end

  # 同値性比較のメソッドを定義
  def ==(other : self)
    return false unless @name == other.@name
    return false unless @age == other.@age
    true
  end
end

user1 = User.new "Taro", 30
user2 = User.new "Taro", 30
user1 == user2 # => true
```

このコードを、`def_equals` を使って書くと次のようにになります。

```

struct User
  def initialize(@name : String, @age : Int32)
 $\langle$ end

# 同値性比較のメソッドを定義
def equals @name, @age
end

user1 = User.new "Taro", 30
user2 = User.new "Taro", 30
user1 == user2 # => true

```

とてもすっきりしました。マクロがいかに強力かがわかります。

4.4.2. record

record は Struct を簡単に定義できるマクロです。通常、Struct の定義は次のように行います。

```

# Structの定義
struct User
  property name : String
  property age : Int32

  def initialize(@name, @age)
 $\langle$ end
end

user1 = User.new "Taro", 30
user1      # => User(@name="Taro", @age=30)
user1.name # => "Taro"
user1.age  # => 30

```

このコードを、**record** を使って書くと次のようになります。

```

# Structの定義
record User, name : String, age : Int32

user1 = User.new "Taro", 30
user1      # => User(@name="Taro", @age=30)
user1.name # => "Taro"
user1.age  # => 30

```

1行で定義が書けてしまいました。**record** は、この他に

- ブロックを渡すことでメソッドを定義できる
- 初期値を与えることができる

- 初期値から型推論できる

という機能もあります。気になる方は `record` のマニュアルを読んでみてください。

4.4.3. parallel

引数に与えた処理を並行処理します。各処理の返り値も受け取ることができます。次のコードは、`job1` `job2` `job3` を並行に実行し、返り値を受け取るコードです。

```
io = IO::Memory.new

# 1秒後に、文字列を出力し文字列を返すlambda
job1 = ->{
  sleep 1
  io.puts "job1 : #{Time.now.to_s("%F %T")}"
  "return job1"
}

# 2秒後に、文字列を出力し文字列を返すlambda
job2 = ->{
  sleep 2
  io.puts "job2 : #{Time.now.to_s("%F %T")}"
  "return job2"
}

# 3秒後に、文字列を出力し文字列を返すlambda
job3 = ->{
  sleep 3
  io.puts "job3 : #{Time.now.to_s("%F %T")}"
  "return job3"
}

# 動かしてみる
io.puts "start : #{Time.now.to_s("%F %T")}"
ret1, ret2, ret3 = parallel job1.call, job2.call, job3.call
```

これを実行すると、返り値、出力結果は次のようにになります。

```
io.to_s
# =>
# start : 2018-01-01 00:00:00
# job1 : 2018-01-01 00:00:01
# job2 : 2018-01-01 00:00:02
# job3 : 2018-01-01 00:00:03

ret1 # => return job1
ret2 # => return job2
ret3 # => return job3
```

実行された時刻を見ると、並行に実行されていることがわかります。また、返り値も適切に受け取ってい
ることがわかります。このように、`parallel` を使えば簡単に並行処理を記述できます。

p!, pp!

`p` や `pp` と同じく、引数として渡された値を標準出力に出力しますが、渡された式自身も表示してくれま
す。デバッグ時に重宝します。

```
arr = [1, 2, 3]

pp! arr.map(&.* 1000)
# output:
# arr.map(&.*(1000)) # => [1000, 2000, 3000]
```

いかがでしたでしょうか。いくつかのマクロを紹介しましたが、この他にも標準のマクロは存在します。
興味のある方は探してみてください。今までのコードがずっとすっきりするはずです。

4.5. マクロの文法

マクロの文法は [公式マニュアル](#) に記載されています。この章では公式マニュアルを基本とし、より詳し
く解説していきます。

4.5.1. マクロのおさらい

マクロの基本的な使い方をおさらいしましょう。次のコードを見てください。この章の冒頭で出たコード
です。

```
macro my_macro(method_name, content)
  def {{method_name}} ①
    {{content}}
  end
end

my_macro my_method, "hoge" ②
my_method # => "hoge"
```

上記の（1）の部分では、`macro` を用いてマクロの定義を書いています。（2）の部分では、定義され
たマクロの呼び出しを行っています。このコードを `crystal run` すると、次のような流れで処理されま
す。

1. マクロ呼び出し時の引数が `my_macro` に渡される
2. 引数展開や条件分岐等の処理をし、Crystal コードが生成される
3. 生成された Crystal コードを、マクロ呼び出し部分に展開する
4. すべてのマクロを展開し終えたら、Crystal コードのコンパイルをする

5. Crystal コードのコンパイルが終わったら実行する

この流れを頭の中に入れつつ、次のステップに進みましょう。

4.5.2. マクロと抽象構文木

Crystal のコードはパーサによって、抽象構文木（Abstract Syntax Tree）にパースされます。抽象構文木を構成する木構造の各要素を AST node と言います。つまり、Crystal のコードは各 AST node で構成されています。

ここでマクロに話を戻します。マクロは Crystal のコードを組み立てるものでした。言い換えると「マクロは AST node を操作して Crystal コードを組み立てるもの」ということになります。実際、マクロが引数として受け取るのは AST node です。そのことを確かめてみましょう。

4.5.3. AST node

マクロが受け取る AST node の型を見てみましょう。

```
# 引数のclass_nameを表示させるmacro
macro ast_node_class_name(ast_node)
  {{ ast_node.class_name }}
end

ast_node_class_name(1)          # => "NumberLiteral"
ast_node_class_name("string")   # => "StringLiteral"
ast_node_class_name(ast)        # => "Call"
ast_node_class_name(["a", "b"])  # => "ArrayLiteral"
ast_node_class_name({key: "value"}) # => "NamedTupleLiteral"
```

NumberLiteral や ArrayLiteral などが表示されました。これらの class は Crystal::Macros::NumberLiteral や Crystal::Macros::ArrayLiteral として定義されています。そして、全 AST node は Crystal::Macros::ASTNode を継承しています。Crystal::Macros::ASTNode の幾つかのメソッドを紹介します。

#line_number は、AST node が書かれている行数を返します。

```
macro ast_node_line_number(ast_node)
  {{ ast_node.line_number }}
end

# ... 他のコードが並ぶ

ast_node_line_number 1      # => 12
ast_node_line_number "string" # => 13
```

#stringify は、AST node の文字列表現を返します。

```

macro ast_node_stringify(ast_node)
  {{ ast_node.stringify }}
end

ast_node_stringify 1          # => "1"
ast_node_stringify "string"   # => "\"string\""
ast_node_stringify ["a", "b"] # => "[\"a\", \"b\"]"
ast_node_stringify CONST      # => "CONST"

```

このように、`Crystal::Macros::ASTNode` class には AST node を操作するためのメソッドが定義されています。そして、それらを継承している class（`Crystal::Macros::ArrayLiteral` など）は、AST node のメソッドに加え、それぞれの便利なメソッドが定義されています。例えば、

`Crystal::Macros::ArrayLiteral` には `Array` に似たメソッドが定義されています。

```

macro ast_node_array_literal(ast_node)
  {{ ast_node.map(&.capitalize).join("::") }}
end

ast_node_array_literal ["apple", "banana"] # => "Apple::Banana"

```

通常の Crystal コードと似たような操作感で書くことができます。AST node を操作しているのか、Crystal コードを操作しているのかをしっかりと意識してプログラミングしましょう。

次からは、実際の文法を具体的に見ていきましょう。

4.5.4. スコープ

マクロにもスコープがあります。

トップレベルに定義した場合は、通常のメソッドと同じようにどこからでも呼び出せるようになります。

トップレベルに定義した場合

```

# トップレベルで定義されたマクロは、どこからでも参照可能
macro my_macro
  "expanded my macro!!!"
end

my_macro # => "expanded my macro!!!"

```

また、トップレベルにマクロを定義する際に `private` 修飾子を付けると、そのファイル内からのみ呼び出せるようになります。

`private` を付けてトップレベルに定義した場合

```
# private を付けた場合は、同一ファイル内でのみ参照可能
private macro my_macro
  "expanded my macro!!!"
end

my_macro # => "expanded my macro!!!"
```

`class` 内にマクロを定義した場合は、インスタンスマソッドではなくクラスマソッドと似たような扱いになることに注意してください。

また、`module` や `struct` でも同様に、クラスマソッドのような扱いになります。

`class` 内に定義した場合

```
class MacroScope
  macro my_macro
    "expanded my macro!!!"
  end

  # class 内で参照可能
  CONSTANT = my_macro

  # インスタンスマソッド内で参照可能
  def instance_method
    my_macro
  end

  # クラスマソッド内で参照可能
  def self.class_method
    my_macro
  end
end

# クラスマソッドと同じ syntax で参照可能
MacroScope.my_macro # => "expanded my macro!!!"

# インスタンスマソッドとしては参照できない
# MacroScope.new.my_macro は undefined method になる

MacroScope::CONSTANT          # => "expanded my macro!!!"
MacroScope.new.instance_method # => "expanded my macro!!!"
MacroScope.class_method        # => "expanded my macro!!!"
```

「マクロが呼び出せない」という問題に陥った場合は、こちらの例を思い出してください。

4.5.5. if

マクロでの条件分岐は `if` を使います。次のコードを見てください。

```
macro conditionals(content)
  {%
    if content == 1 %
      "one"
    %else %
      {{ content }}
    %end %
  end
```

```
conditionals 1 # => "one"
conditionals 2 # => 2
```

if での true/false の扱いは次のようにになっています。

- **false** として扱われるもの
 - Nop
 - NilLiteral
 - BoolLiteral の false
- **true** として扱われるもの
 - 上記以外

また、**if** は **macro** の外でも利用できます。

```
{% if env("DEBUG") %}
  puts "===== DEBUG MODE ====="
{%
  end %}
```

これでちょっとしたマクロを素早く書くことができます。

4.5.6. for

マクロでのループは **for** を使います。次のコードを見てください。

```
macro iteration(names)
  {%
    for name, index in names %
      def {{name}}
        {{index}}
      end
    %end %
  end
```

```
iteration [foo, bar, baz]
```

```
foo # => 0
bar # => 1
baz # => 2
```

`ArrayLiteral` を渡すと `for` 文が回り、メソッドを定義します。この `for` は、`HashLiteral` にも対応しています。

```
macro iteration(names)
  {%
    for key, value in names %
      def {{key.id}}
        {{value}}
      end
    {%
      end %
    }
  end

iteration({foo: "FOO", bar: "BAR", baz: "BAZ"})

foo # => "FOO"
bar # => "BAR"
baz # => "BAZ"
```

`for` も `if` と同様、`macro` の外でも利用できます。

```
{% for name, index in ["foo", "bar", "baz"] %
  def {{name.id}}
    {{index}}
  end
{%
  end %

}

foo # => 0
bar # => 1
baz # => 2
```

4.5.7. 可変長引数

通常の Crystal コードの感覚で可変長引数を扱うことができます。引数の定義に `*` を付けるだけです。受け取った引数は `TupleLiteral` になります。

```
macro variadic_arguments(*names)
  {%
    for name, index in names %
      def {{name.id}}
        {{index}}
      end
    {%
      end %
    }
  end

variadic_arguments foo, bar, baz

foo # => 0
bar # => 1
baz # => 2
```

4.5.8. splat 展開

* は、`ArrayLiteral` と `TupleLiteral` の splat 展開にも使用できます。また、** は `HashLiteral` と `NamedTupleLiteral` の splat 展開に使用できます。次のコードを見てください。

```
macro splat
  p "Splatting a array"
  {%
    array = [1, 2, 3] %
  }
  p {*array}

  p "Splatting a tuple"
  {%
    tuple = {4, 5, 6} %
  }
  p {*tuple}

  # p "Double Splatting a hash"
  # {%
  #   hash = {"a" => 1, "b" => 2} %
  # }
  # p {**hash}
  #
  # -----
  # Syntax error in expanded macro: splat:11: unexpected token: =>
  #
  # p "a" => 1, "b" => 2
  #           ^
  #

  p "Double Splatting a named tuple"
  {%
    named_tuple = {"c": 3, "d": 4} %
  }
  p {**named_tuple}
end

splat
# output:
# "Splatting a array"
# 1
# 2
# 3
# "Splatting a tuple"
# 4
# 5
# 6
# "Double Splatting a named tuple"
# {c: 3, d: 4}
```

展開は、各要素をカンマで区切った形になります。`HashLiteral` もそのままカンマ区切りで出力されますが、使い所によっては上記のように `Syntax error` になります。

4.5.9. 定数

マクロは定数にアクセスできます。次のコードを見てください。

```
CONSTANTS = ["foo", "bar", "baz"]
```

```
{% for value in CONSTANTS %}  
  puts {{value}}  
{% end %}  
# output:  
# foo  
# bar  
# baz
```

一見、マクロ以外の部分をマクロが参照しているので違和感があります。

Crystal は定数の再代入は認めていません。再代入がある場合は、 `already initialized constant XXX` というエラーでコンパイルに失敗します。つまり、定数の値は不变なのでマクロ解析のフェーズでも扱えるというわけです。

ネストしたマクロ

ネストしたマクロも書くことができます。つまり、「マクロ定義を生成するマクロ」です。

ネストしたマクロは、外側から順に内側に向かって展開されます。その際、内側のマクロは外側のマクロで展開されないように \ でエスケープする必要があります。公式マニュアルの例がわかりやすいので引用します。次のコードを見てください。

```

macro define_macros(*names)
  % for name in names %
    macro greeting_for_{name.id}(greeting)
      \{%
        if greeting == "hola" %
          "¡hola {{name.id}}!"
        \{%
          else %
            "\{{greeting.id}\} {{name.id}}"
        \{%
          end %
        end
      \{%
        end %
      end
    \{%
      end %
    end

# This generates:
#
#   macro greeting_for_alice(greeting)
#     \{%
#       if greeting == "hola" %
#         "¡hola alice!"
#       \{%
#         else %
#           "\{{greeting.id}\} alice"
#       \{%
#         end %
#       end
#     \{%
#       macro greeting_for_bob(greeting)
#         \{%
#           if greeting == "hola" %
#             "¡hola bob!"
#           \{%
#             else %
#               "\{{greeting.id}\} bob"
#           \{%
#             end %
#           end
#         \{%
#           define_macros alice, bob

greeting_for_alice "hello" # => "hello alice"
greeting_for_bob "hallo"   # => "hallo bob"
greeting_for_alice "hej"   # => "hej alice"
greeting_for_bob "hola"    # => "¡hola bob!"

```

外側のマクロで展開しない部分だけエスケープしていることに注目してください。特に、

```
"\{{greeting.id}\} {{name.id}}"
```

の部分では、外側のマクロで `\{{name.id}\}` の部分は展開されますが、`\{{greeting.id}\}` の部分は展開されません。`\{{greeting.id}\}` の部分は内側のマクロで展開されます。Nested macros は、マクロの記述で重複が多い場合に有効です。しかし、可読性が損なわれやすいので注意が必要です。

4.5.10. 生成コードの注意点

マクロで生成するコードは、それ単体で Crystal のコードとして完結していなければなりません。言い替えれば、生成されたコードを別のファイルに書き出して正しくパースされるようなコードでなければなりません。この制約は忘れてしまいがちなので気をつけましょう。次の例を見てください。

```

ret = ""
var = "pitfalls"

ret = case var
  {% for klass in [Int32, String] %}
    when {{ klass.id }} then "#{var} is {{ klass }}"
  {% end %}
end

```

一見、マクロが展開されたら正しい Crystal のコードが生成されるように見えます。しかし、マクロで生成されるコードは `when` から始まる部分だけなので、Crystal のコードとしては不完全で、エラーとなります。

この場合は、`{% begin %} … {% end %}` でコードを括りましょう。

```

ret = ""

{% begin %}
var = "pitfalls"

ret = case var
  {% for klass in [Int32, String] %}
    when {{ klass.id }} then "#{var} is {{ klass }}"
  {% end %}
end
{% end %}

ret # => "pitfalls is String"

```

こうすることで、マクロが生成するコードが正しい Crystal のコードとなるため、コンパイルが通るようになります。陥りやすい間違いなので気をつけてください。

4.5.11. 型の情報にアクセスできる `@type`

マクロには特別なインスタンス変数 `@type` が用意されています。これを使うと、コンパイル時の型情報にアクセスできます。実際どんなメソッドが存在しているかを見たほうがわかりやすいので、いくつかご紹介します。`@type` は `Crystal::Macros::TypeNode` クラスです。

`TypeNode#instance_vars`

型に定義されているインスタンス変数を返します。返り値は `Crystal::Macros::MetaVar` クラスの配列です。`MetaVar` クラスは、変数やインスタンス変数を表す型で、名前（`MetaVar#name`）と型（`MetaVar#type`）を持っています。

```

class MyClass
  def initialize(@name : String, @age : Int32)
  end

  def instance_variables_name
    # インスタンス変数の名前を配列で返す
    {{ @type.instance_vars.map(&.name.stringify) }}
  end

  def instance_variables_type
    # インスタンス変数の型を配列で返す
    {{ @type.instance_vars.map(&.type) }}
  end
end

my_class = MyClass.new("Taro", 30)
my_class.instance_variables_name # => ["name", "age"]
my_class.instance_variables_type # => [String, Int32]

```

TypeNode#methods

型に定義されているメソッドの情報を返します。返り値は `Crystal::Macros::Def` クラスの配列です。`Def` クラスは、`def` 文を表す型で、メソッド定義に関するさまざまな情報を持っています。例えば、`Def#args` は引数の情報、`Def#return_type` はメソッドの返り値の型を表します。

```

class MyClass
  def m1
  end

  protected def m2
  end

  private def m3
  end

  def methods_name
    # 定義されているメソッドの名前を返す
    {{ @type.methods.map(&.name.stringify) }}
  end

  def methods_visibility
    # 定義されているメソッドのアクセス修飾子を返す
    {{ @type.methods.map(&.visibility.stringify) }}
  end
end

my_class = MyClass.new

my_class.methods_name
# => ["m1", "m2", "m3", "methods_name", "methods_visibility"]

my_class.methods_visibility
# => [":public", ":protected", ":private", ":public", ":public"]

```

これらの他にもメソッドはたくさんあります。私の調べた限りでは、組み合わせればやりたいことはできるという、必要最低限なメソッドはそろっていました。興味のある方はぜひ調べてみてください。

4.5.12. Hooks

一部の特別な名前を持ったマクロは hooks と呼ばれ、特定のタイミングでコンパイル時に実行されます。

テーブル 1. Hooks

| マクロ | 効果 |
|----------------------------|----------------------------------|
| macro inherited … end | サブクラスが定義されたときに実行されるマクロ |
| macro included … end | モジュールが include されたときに実行されるマクロ |
| macro extended … end | モジュールが extend されたときに実行されるマクロ |
| macro method_added … end | メソッドが追加されたときに実行されるマクロ |
| macro method_missing … end | 呼び出そうとしたメソッドが定義されていない場合に実行されるマクロ |
| macro finished … end | インスタンス変数の型が決定したあとに呼び出されるマクロ |

inherited の例を見てみましょう。

```

class SuperClass
macro inherited
  def type_name
    {{ @type.name.stringify }}
  end
end
end

class SubClass < SuperClass
end

# SuperClass.new.type_name は undefined method 'type_name' for SuperClass となる
SubClass.new.type_name # => "SubClass"

```

継承した場合のみ実行されるので、`SuperClass` には `#type_name` が存在していないことがわかります。

`method_missing` の例も見てみましょう。

```

macro method_missing(call)
  puts "name: {{call.name.id}}"
  puts "args: {{call.args.size}} arguments"
end

foo
# output:
# name: foo
# args: 0 arguments

bar 1, 'a'
# output:
# name: bar
# args: 2 arguments

```

`method_missing` の引数は `Crystal::Macros::Call` です。これはメソッドの呼び出しを表すクラスです。`#args` や `#receiver` などがあります。

4.5.13. Fresh variables

マクロが展開されると、マクロ内で定義した変数もそのまま展開され、Crystal コードとして解釈されます。次の例を見てください。

```
macro update_x
  x = 1
end

x = 0
update_x

x # => 1
```

これは、ローカル変数を上書きして重複を排除する際には有効です。しかし、ライブラリで提供するマクロなどでは、意図しない形で上書きされてしまう可能性があります。そのため、**fresh variables** という構文が用意されています。次の例を見てください。

```
macro dont_update_x
  %x = 1
  puts %x
end

x = 0
dont_update_x # outputs 1

x # => 0
```

%変数名 とすることで、そのマクロのコンテキスト内で唯一の変数として扱われます。仕組みは簡単です。上記のコードで `crystal tool expand` をしてみましょう。

```
$ crystal tool expand -c /path/to/fresh_variables_example.cr:6:1
/path/to/fresh_variables_example.cr
1 expansion found
expansion 1:
  dont_update_x

# expand macro 'dont_update_x' (/path/to/fresh_variables_example.cr:2:1)
~> __temp_20 = 1
  puts(__temp_20)
```

`__temp_20` のような変数に置き換わっています。このように、マクロの実行フェーズで変数名を置き換えています。

4.6. まとめ

以上で、マクロとはどういうものか、マクロの文法はどうなっているのかなどの説明を終わります。マクロのおおまかなイメージは湧きましたでしょうか。自分はライブラリを書く際にマクロを使いますが、やはり重複排除の効果はすごいと思います。また、DSL の提供も比較的簡単にできるのではないかでしょうか。マクロに慣れてくると、Crystal 本来のコードを書くよりもマクロを書いている比率が多くなる印象が強いです。今回のこの章を読み、「マクロが読めるようになった」「マクロが書けるようになった」と

いう方が一人でも増えて頂けると幸いです。

5. Shards

著者: AKJ

Ruby に RubyGems があるように、Crystal にも Shards というパッケージ管理の仕組みが存在しています。Shards は Crystal 標準のパッケージマネージャとして採用されており、Shards 関連の操作に使用する `shards` コマンドも Crystal コンパイラに同梱されています。ですので、Shards を利用するための特別な準備は必要ありません。

Shards における個々のパッケージは `shard`（シャード）と呼ばれ、これは鉱物などの「欠片」や「破片」といった意味の単語です。Ruby や Python といった先行する言語と比べると充分ではないかもしれません、それでも GitHub などを利用して 2000 を超える shard が公開されています。

RubyGems と Shards の大きな違いは、パッケージのインストール先です。RubyGems は基本的にグローバルな Ruby の実行環境に対してパッケージをインストールする仕組みで、それをプロジェクト単位で個別管理するために別途 Bundler が登場しました。一方、Shards では導入パッケージをプロジェクト単位で管理することが前提となっています。

また、他言語のパッケージ管理手法と異なり、Shards には集中管理された公式のリポジトリが存在しません。ほとんどの shard は、git リポジトリとして（多くの場合 GitHub 上で）公開されています。このことは、`shard` を公開することへのハードルを大きく下げてくれていますが、一方で公開されている shard のクオリティ確保や、検索性の面では足かせにもなっています。

ただし、この原稿を執筆している2018年9月時点で Crystal がバージョン `1.0` に至っていないよう、Shards の開発は現在も継続中です。現時点での課題となっている点も、今後のバージョンアップで改善されることを期待しましょう。

この章では、`shard` の使い方と作り方、および `shard` 作成において使用することになるテストの実行方法について簡単に説明します。

5.1. `shard` を使う

まずはインターネット上に公開されている shard を自分のプログラムから利用するための手順を見てみましょう。

5.1.1. プロジェクトの機能と名前を決める

前述の通り、Shards では導入パッケージの管理をプロジェクト単位でおこなうため、なにはともあれそのプロジェクトで実現したい機能とプロジェクトの名前を決めなければいけません。とはいえ、通常はやりたいことがあってプログラムを書き始めるはずですので、この辺りでつまづくこともそうないとは思います。

というわけで、実用的かどうかはさておき、ページタイトルと本文をパラメータとして受け取って最低限の HTML ソースを出力する、というだけのシンプルなコマンドラインツールを作つることにしましょう。

プロジェクト名は、`shard` として公開するのであればいくつかるルール（後述）もありますが、そうでない場合にはそこまで深く考える必要はありません。今回はシンプル（simple）な HTML を返すアプリなので `simple_html` というプロジェクト名にすることにします。

simple_html コマンド利用イメージ

```
$ simple_html "Page Title" "Page body."  
<html><head><title>Page Title</title></head><body><h1>Page Title</h1><p>Page  
body.</p></body></html>
```

5.1.2. プロジェクトひな形を作る

crystal コマンドにはソースコードのコンパイル以外にもいくつか便利な機能が実装されており、その中の1つにプロジェクトひな形の生成機能（`crystal init` コマンド）があります。

`crystal init` コマンドを実行する際には、プロジェクトの種類（`app` もしくは `lib`）とプロジェクト名を指定します。例えば、`simple_html` をアプリケーションとして実装する場合のひな形生成コマンドは `crystal init app simple_html` です。このコマンドを実行するとカレントディレクトリの直下にプロジェクト名と同じ名前の `simple_html` ディレクトリができ、その中に各種ひな形が自動生成されます。

`simple_html` アプリケーション用プロジェクトのひな形生成コマンド

```
$ crystal init app simple_html
```

プロジェクトひな形として生成されるファイル/ディレクトリ

.editorconfig

文字コードや改行コード、インデントなどの設定を異なるエディタ間で共通利用するための設定ファイル。詳細を知りたければ EditorConfig で検索。

.git/ (ディレクトリ)

このプロジェクト用の初期化済み git リポジトリ。

.gitignore

git 管理除外ファイルの指定用。

.travis.yml

Travis CI を使用してテストやビルドを自動化するための設定ファイル。使いこなせば強力なツール。ただ、最初のうちはあまり気にしなくても良い。

LICENSE

このプロジェクトのライセンスを明示するためのテキストファイル。自分で使うだけのアプリケーションなら気にする必要はない。標準では MIT ライセンス前提の内容になっているので、必要に応じて適宜変更する。

README.md

Markdown フォーマットで書かれたこのプロジェクトの README ファイル。GitHub などを通じて公開する予定ならちゃんと書いた方が良い。

shard.yml

プロジェクトに関連したメタデータ（名前やバージョン、作者など）やプロジェクトが利用する shard などの情報を記述するための設定ファイル。

src/ (ディレクトリ)

プログラムのソースディレクトリ。ソースコードはすべてこの配下に置く。

src/[プロジェクト名].cr

プロジェクトのメインソースファイル。プログラムのエントリーポイントはここ。

spec/ (ディレクトリ)

テスト用ディレクトリ。テスト関連のファイルはここへ置く。

spec/spec_helper.cr

テスト関連ファイルその1。あまり触らない方。

spec/[プロジェクト名]_spec.cr

テスト関連ファイルその2 (spec ファイル)。実際のテスト内容を記述する方。

5.1.3. shard を探す

`simple_html` を実装するにあたり HTML をすべて手打ちすることができますが、それはいささか面倒なので HTML を出力するのに便利な shard を探してみることにします。

Shards には公式のリポジトリがなく、shard の検索機能が提供されていません。そのため、標準ライブラリで提供されていない機能を使いたい場合に、その機能を提供する shard が存在するかどうかを調べるのが最初のハードルになります。

今のところ、この問題を解決する決定打は出てきていませんが、以下の2サイトは必要な shard を探す際の助けになってくれるはずです。

Awesome Crystal

<https://github.com/veelenga/awesome-crystal>

一定の条件を満たした自薦他薦の shard をカテゴリ分類してリスト化。リストの更新も人の手で管理されているため、限定的ではあるもののある程度の基準をクリアした shard が見つかる。

CrystalShards.xyz

<http://crystalshards.xyz/>

GitHub で公開されている shard のキーワード検索機能を提供。GitHub で公開されている shard を自動的に収集しているため玉石混こうではあるものの、Awesome Crystal より網羅的にマイナな shard も見つけられる。

試しに Awesome Crystal 内を「html」で検索してみると、HTML 構築専用の構文を提供してくれる `html_builder` という shard が見つかりました。早速これを使ってみることにしましょう。

crystal-lang/html_builder

https://github.com/crystal-lang/html_builder

5.1.4. 使用する shard を指定する

プロジェクトディレクトリ内にある `shard.yaml` ファイルは、shard 関係の設定を記述するファイルです。

アプリケーションとして初期化された `shard.yaml` は以下のような内容になっています。

ひな形生成時の `shard.yaml`

```
name: simple_html
version: 0.1.0

authors:
- Your Git Name <your_mail@example.org>

targets:
simple_html:
  main: src/simple_html.cr

crystal: 0.26.1

license: MIT
```

ここへ `html_builder` を指定するための設定を追加しましょう。大抵の場合 shard の README にはその shard を使用するために `shard.yaml` へ追加すべき内容が紹介されています。

README に従って `dependencies` を追加した `shard.yaml` はこのようになりました。

依存ライブラリの設定追加後

```
name: simple_html
version: 0.1.0

authors:
- Your Git Name <your_mail@example.org>

targets:
simple_html:
  main: src/simple_html.cr

crystal: 0.26.1

license: MIT

dependencies:
  html_builder:
    github: crystal-lang/html_builder
```

使いたい shard のバージョン指定する

2018年9月時点の `html_builder` はバージョン `0.2.2` が最新です。そのため、バージョン指定をせず `shard` をインストールした場合には最新のバージョン `0.2.2` がインストールされます。しかし、`shard.yml` では、インストールする `shard` のバージョンを指定することもできます。例えば、`shard.yml` での記述を以下のようにすると、旧バージョンの `0.2.1` をインストールすることができます。

バージョン指定付きの `shard` 設定

```
dependencies:  
  html_builder:  
    github: crystal-lang/html_builder  
    version: 0.2.1
```

ここで指定する `version` には、特定のバージョンを直接記述する以外に、バージョンと `>/</>=/<=/~>` 演算子を組み合わせて使用することもできます。

例

`>= 0.2.0` でバージョン0.2.0以上。

これら演算子のうち、`>/</>=/<=` 辺りはどのバージョンが対象となるのか直感的に理解できると思いますが、`~>` は以下ののようなやや特殊な挙動をします。

- `~> 2.0.3` は、`>= 2.0.3` かつ `< 2.1` と同じ。
- `~> 2.1` は、`>= 2.1` かつ `< 3.0` と同じ。

GitHub 以外で公開されている `shard`

`html_builder` は GitHub で公開されていましたが、`shard.yml` ファイルには、GitHub だけでなく、以下のような方法で `shard` を指定可能です。

`shard` の取得先

`github: user/repository`

GitHub リポジトリを指定する。

`gitlab: user/repository`

GitLab リポジトリを指定する。

`bitbucket: user/repository`

Bitbucket リポジトリを指定する。

`git: git://git.example.org/repository.git`

任意の git リポジトリを指定する。

`path: ../path/to/shard`

ファイルシステム上のディレクトリパスで指定する。

5.1.5. shard をインストールする

`shard.yaml` で指定した shard をプロジェクト環境にインストールするコマンドは `shards install` です。プロジェクトディレクトリの直下でこのコマンドを実行すると、プロジェクトディレクトリの下に `lib` ディレクトリが作られ、その中に必要なファイル一式が展開されます。

`shard` のインストール

```
$ shards install
Fetching https://github.com/crystal-lang/html_builder.git
Installing html_builder (0.2.2)
```

これだけで `shard` のインストールは完了です。

また、インストール済みの `shard` は `shards list` コマンドで確認できます。

```
$ shards list
Shards installed:
* html_builder (0.2.2)
```

5.1.6. ソースコード内で shard を使う

ひな形生成直後のソースファイル `src/simple_html.cr` には、プロジェクト名 `simple_html` をキャメルケースに変換した `SimpleHtml` モジュールと定義されています。この `SimpleHtml` モジュールはこのプロジェクトの名前空間として使用するためものです。ですので、特に理由がない限りプロジェクト固有の機能は `SimpleHtml` モジュールか、もしくはその中で定義した型に対して実装することになります。ただし、この時点ではまだバージョンを表す `VERSION` 定数が定義されているだけの状態です。

ひな形生成時のメインソースコード

```
# TODO: Write documentation for `SimpleHtml`
module SimpleHtml
  VERSION = "0.1.0"

  # TODO: Put your code here
end
```

`shard` として公開されている機能をプログラム中で利用するためには `require` 文を使用してその旨を明示する必要があります。大抵の場合 `shard` の `README` にはその `shard` の使用法も記載されており `require` 文の書き方もそこで確認できます。

指定された `require` 文と、HTML を生成する `SimpleHtml.build` メソッドを実装し、`simple_html` コマンドのソースコードが完成しました。

完成した `simple_html` コマンドのソースコード

```
require "html_builder"

# Build simple HTML source.
module SimpleHtml
  VERSION = "0.1.0"

  def self.build(args : Array(String))
    title = args[0]
    body = args[1]
    HTML.build {
      html {
        head {
          title { text title }
        }
        body {
          h1 { text title }
          p { text body }
        }
      }
    }
  end
end

if File.basename(PROGRAM_NAME) == File.basename(__FILE__, ".cr")
  puts SimpleHtml.build(ARGV)
end
```

5.1.7. 実行してみる

ビルド後に試しに実行してみると `html_builder` の機能を使っていることが確認できました。

`simple_html` コマンドのコパイルと実行結果

```
$ crystal build src/simple_html.cr
$ ./simple_html "Page Title" "Page body."
<html><head><title>Page Title</title></head><body><h1>Page Title</h1><p>Page
body.</p></body></html>
```

動作に問題がないようであれば、最適化フラグ `--release` を付けた状態で再度ビルドしておきましょう。

最適化フラグを付けたビルド

```
$ crystal build --release src/simple_html.cr
```

以上がインターネット上で公開されている shard を利用する際のざっくりとした手順になります。

NOTE

説明をシンプルにするため、今回の `simple_html` コマンドにはあえて省いている機能が多くあります。例えば、現状ではコマンドラインパラメータが足りなかったり多すぎたりする状況が全く考慮されていません。コマンドラインパラメータが足りなければ実行時例外 (`IndexError`) を吐いて異常終了してしまいますし、多すぎた場合には3つめ以降が単に無視されてしまいます。実際に使用するコマンドラインツールを作成する場合には、コマンドラインパラメータのバリデーションやエラー処理などが必要になります。

5.1.8. 補足：shards コマンドのもつ機能

`shard` のインストールに使用した `shards` コマンドには、インストール以外にも `shard` を管理するための各種機能が用意されています。

例えば、Crystal 本体のバージョンを上げたら `shard` が動かなくなった場合など、インストール済みの `shard` をアップデートしなければならない場面に遭遇することがあります。しかし、`shard` のインストール時に使用した `shards install` コマンドは、すでにインストールされている `shard` については処理を行いません。そのため、`shards install` を再度実行したとしてもインストール済みの `shard` はバージョンアップできません。これは、たとえ `shard.yml` で新しいバージョンを明示的に指定したとしても同様です。

こうした場面では、替わりに `shards update` コマンドを使用します。このコマンドはインストール済みの `shard` が `shard.yml` で指定したバージョンと異なっていれば再取得してくれます。また、`shard.yml` でバージョンが指定されていない場合は、該当の `shard` を最新の状態に更新します。

この他にも、インストール済みの `shard` を一覧表示してくれる `shards list` などいくつかの機能が用意されており、使用可能なコマンドやオプションは `shards help` で確認できます。

5.2. shard を作る

Crystal が標準で提供していない様々な機能を使えるようになる、というだけでも Shards が有用な仕組みだということはご理解いただけだと思います。では、自作のライブラリを `shard` として公開するモチベーションとは何でしょう？

「世界中の Crystal 使いの人々に便利な機能を提供したい」という理由も当然アリです。ただ、そこまで肩肘を張らず「自分が便利だから」というだけでも、自作ライブラリを `shard` として公開する理由としては十分です。実際、複数のプロジェクトで共通使用する汎用部品は、仮に自分だけしか使わなさそうであっても `shard` として公開しておくことでデプロイ時の手間やその後のメンテナンスをかなり省力化できます。

例えば、ライブラリを開発環境とは別の本番環境で使用したい場合、プロジェクト全体を `tar` で固めて `scp` で転送、なんてことをしがちです。こうした作業はただでさえ手間がかかりますが、使用するライブラリのバージョンが異なる複数のプロジェクトを維持していくうと思うと気が遠くなります。しかし、もしそのライブラリが `shard` として公開されていれば、`shard.yml` に2~3行追加してから `shards install` を実行するだけですみます。さらに、必要であれば個々のプロジェクトごとに過去のバージョンを指定してインストールすることもできます。

これだけでも、自作ライブラリを `shard` 化しておくメリットはあるのではないでしょうか。

「プログラミング言語の拡張ライブラリを公開する」というと敷居が高く感じられるかもしれません。が、Crystal の `shard` を公開する手順は比較的シンプルです。実際、`crystal init` コマンドで作成したひ

な形をベースにコードを書いて、特に何も考えずにそのまま GitHub へ公開するだけでも、とりあえず shard として使えてしまったりします。

とはいって、ある程度は shard の体裁というものもありますので、ここからは公開リポジトリで shard を公開する際に必要な最低限の手順について説明したいと思います。

NOTE 今回は作成した shard を GitHub 上で公開することを想定しています。GitHub や git の操作方法についてはすでにご存知だということを前提としていますので、これらの使い方については各種 Web サイトや書籍などをご参照ください。

5.2.1. shard の名前

アプリケーションを作成する場合と同様、何はともあれ shard として再利用したいライブラリに持たせる機能と、その名前を決める必要があります。機能については既にやりたいことがあるはずですが、一般に公開する shard の名前は意外と悩みどころです。

shard.yml ファイルの記述方法を定義した「 shard.yml specification. 」には、shard の名前として以下のような命名規則が規定されています。

- 他の shard と重複しないこと
- 50 文字以下
- 英文字は小文字（**a-z**）を使用すべき
- 名前の一部に「 crystal 」を含むべきではない
- 数字（**0-9**）を含んでも良いが、先頭に置いてはならない
- アンダースコア（**_**）やダッシュ（**-**）を含んでも良いが、先頭や末尾に置いてはならない
- アンダースコアやダッシュが連続してはならない

shard.yml specification.

<https://github.com/crystal-lang/shards/blob/master/SPEC.md>

基本的にこれらの条件に従う必要がありますが、shard 名の重複についてはそこまで厳密に考えなくても問題になることは少ないでしょう。単機能の shard 名は割とカブりがちですし、その名前を世界中の誰も使っていないことを確認することは現実的には不可能です。とはいって、最低でも標準添付のライブラリとカブるような名前は避ける必要があります。また、余裕があれば CrystalShards.xyz で使いたい名前を検索して、ヒットするかどうか確認くらいはしてみても良いかもしれません。

また、既存 shard の例をいくつかみてみると、shard の命名スタイルには大きく分けて2つのパターンがあるようです。

1. `html_builder` や `mysql` など、機能をそのまま表した名前。単機能とは言わないまでも何か1つの対象に焦点を絞った shard に多い。
2. `kemal` や `topaz` など、機能とは関係なくプロダクトイメージなどからつけられた独自の名前。フレームワークなどある程度の規模を持ち、複数の機能から構成されている shard に多い。

これらを参考にわかり易い、またはカッコいい shard 名を考えてみてください。

ここでは整数 `n` に対して `n` 番目のフィボナッチ数を返してくれる shard を作ってみることにしましょう。单機能のシンプルな shard ですので、shard 名はそのまま `fibonacci` としました。

shard 名はスネークケースで

命名規則上はダッシュ記号も使えことになっていますが、shard 名には基本的には単語の区切りにアンダースコアを使用するスネークケースで命名することをお勧めします。何故かというと、shard 名がソースコードのファイル名にも使用される場合が多いことがその理由です。Crystal 公式ドキュメントの「コーディングスタイル」では、ソースファイルのパスを名前空間に合致させ、ファイル名として型名をスネークケースに変換したものを使用するよう推奨されています。

例

`HTTP::WebSocket` → `http/web_socket.cr`

Crystal公式ドキュメント「コーディングスタイル」

https://crystal-lang.org/docs/conventions/coding_style.html

ちなみに、shard 名をスネークケースで命名してプロジェクト名にも使用すれば、プロジェクトひな形で生成されたソースファイル名をそのまま使用できます。

また、同コーディングスタイルでは、型名（クラス名やモジュール名）として複数の単語を先頭文字のみ大文字にしてそのまま連結したアッパー・キャメルケースを使用することが推奨されています。例えば、`crystal init` コマンドによるプロジェクトひな形の自動生成では、プロジェクト名をアッパー・キャメルケースへ変換した名前で名前空間用のモジュールが定義されます。この点からも shard 名はスネークケースで付けておくと便利です。

5.2.2. プロジェクトひな形の作成

shard を作成する際も、それ用のプロジェクトを立ち上げることになります。プロジェクト名は特にこだわりがなければ shard 名と同じで構いません。というよりむしろ、ひな形で提供される各種ファイルを有効利用するには、プロジェクト名と shard 名をそろえておく方が良いでしょう。

独立したアプリケーションではなく、再利用可能なライブラリを作る場合のプロジェクトひな形生成コマンドは `crystal init lib プロジェクト名` です。

fibonacci shard 用プロジェクトのひな形生成コマンド

```
$ crystal init lib fibonacci
```

なお、ライブラリを作る場合もアプリケーションを作る場合も、`crystal init` コマンドで生成されるファイルやディレクトリの構成に違いはありません。

5.2.3. `shard.yml` の記述確認

`shard` を使用する際にも使用した `shard.yml` ですが、本来このファイルはプロジェクトを `shard` として公開する際のメタデータを記述するためのものです。公開 `shard` にはメタデータとして最低限以下のような情報が必要です。

ひな形に含まれるメタデータ

`name`

`shard` の名前。プロジェクト名がそのまま使用される。

`version`

`shard` のバージョン情報。ひな形生成時は `0.1.0`。

`authors`

`shard` 作成者のリスト（配列）。`git` の設定で `user.name` と `user.mail` が設定されていればそれらが使用される。

`crystal`

`shard` が対応する Crystal のバージョン。ひな形を生成した `crystal` コマンドのバージョンが使用される。

`license`

`shard` 公開時に採用するライセンスの種類。デフォルトでは `MIT` ライセンスが設定されている。

上のリストにある通りこれらの情報はひな形生成時に自動生成されますので、もある程度そのままで利用可能な状態になっています。複数人で開発している場合に `authors` を追加したり、`MIT` 以外のライセンスを `license` に指定したりするなど、必要に応じてこれらの値を変更してください。また、作りたい `shard` に別の `shard` の機能が必要であれば、アプリケーションを作成した際と同様に `dependencies` の設定を追加します。

ひな形生成時の `shard.yml`

```
name: fibonacci
version: 0.1.0

authors:
  - Your Git Name <your_mail@example.org>

crystal: 0.26.1

license: MIT
```

`fibonacci` では他の `shard` を使用することもなく、特に変更すべき内容もありませんのでこのまま進める

ことにします。

shard.yml ファイルのフォーマット

拡張子が `.yml` となっていることからもわかる通り、`shard.yml` ファイルは YAML で書かれています。

「`shard.yml specification.`」には、`shard.yml` 自体のルールも以下のように定義されています。`shard.yml` を修正した場合は、これらの条件から外れないように注意しましょう。このほか、「`shard.yml specification.`」ではひな形に登場しない項目がいくつも説明されています。状況によっては便利な項目もありますので、一度詳しく目を通しておくことをお勧めします。

- YAML ドキュメントとして構文的に正しいこと
- 文字コードは UTF-8 を使用すること
- 空白2文字でインデントすること
- 文字列、配列、ハッシュ以外の YAML の要素を使用しないこと

5.2.4. テストケースを書く

最近ではテスト駆動開発 (test-driven development) や振る舞い駆動開発 (behavior driven development) が流行りのようです。テスト実行機能がコンパイラに標準で用意されてたりすることから、Crystal としてはこうした開発手法が想定されているようです。これらの方では、まず想定されるパブリックメソッドの挙動（テストケース）を定義し、その後にテストケースを満足させるようにコードを実装する、という流れで開発を進めます。

プロジェクトひな形にはあらかじめテスト用の構成が含まれていますので、ソースファイルにコードを書き始める前に、まずは `fibonacci` のテストケースを書いてみましょう。

`fibonacci shard` の使用イメージ

```
require "fibonacci"

Fibonacci.number(3) #=> 2_big_i
```

今回は shard 名（プロジェクト名）が `fibonacci` ですので、名前空間用のモジュールは `Fibonacci` です。提供するメソッドは `n` 番目のフィボナッチ数を返す `Fibonacci.number(n)` のみ。引数 `n` は整数 (`Int` 型のいずれか) でさえあれば良いでしょう。ただし、結果として受け取るフィボナッチ数は `n` が大きくなると急激に大きくなります。`n` が200を超える頃には128ビット整数でも表現しきれないサイズになりますので、こちらは可変長整数 (`BigInt`) 型を使用することにします。また、あまり一般的ではありませんが、フィボナッチ数は `n` が負の場合も定義されています。せっかくですので、`n` が負のフィボナッチ数にも対応させることにしましょう。

プロジェクトひな形にはテストケースを定義するための `spec/fibonacci_spec.cr` が含まれています。ただし、ここで定義されている内容は「`false` は `true` と等しいはず」という内容で、当然ながらこの状態でテストを実行しても必ず不合格になります。

ひな形生成時の fibonacci_spec.cr

```
require "./spec_helper"

describe Fibonacci do
  # TODO: Write tests

  it "works" do
    false.should eq(true)
  end
end
```

fibonacci の挙動に合わせて必要なテストケースを定義すると以下のようになりました。

テストケースを定義した fibonacci_spec.cr

```
require "./spec_helper"

describe Fibonacci do
  describe ".number(n : Int)" do
    it "returns fibonacci numbers correctly." do
      Fibonacci.number(5).should eq BigInt.new(5)
      Fibonacci.number(10).should eq BigInt.new(55)
    end

    it "returns negative fibonacci numbers correctly." do
      Fibonacci.number(-5).should eq BigInt.new(5)
      Fibonacci.number(-10).should eq BigInt.new(-55)
    end

    it "returns very large fibonacci number correctly." do
      Fibonacci.number(200).should eq BigInt.new
      ("280571172992510140037611932413038677189525")
      Fibonacci.number(-200).should eq BigInt.new("-"
      280571172992510140037611932413038677189525")
    end
  end
end
```

テストケースの書き方はこの章の最後で詳しく紹介しますので、ひとまずは「そういうもの」として先へ進んでください。

5.2.5. 機能の実装

必要なテストケースが準備できたら、次はそのテストケースを満足させる機能を実装していきます。テストはパブリックなインターフェース（メソッド）の入出力を定義しているだけですので、メソッドがどう実装されるかは気にしません。

fibonacci の実装方法も何パターンか考えられますが、とりあえず、一度計算した内容のキャッシュくら

いは持てるように実装してみたのが以下のコードです。

`fibonacci.cr` のソースコード

```
require "big"

# Return fibonacci number.
module Fibonacci
  VERSION = "0.1.0"

  @@numbers = [BigInt.new(0), BigInt.new(1)]

  def self.number(n : Int)
    negative = (n < 0)
    even = n.even?
    n = n.abs
    while @@numbers.size < n + 1
      @@numbers << @@numbers[-1] + @@numbers[-2]
    end
    if negative && even
      -@@numbers[n]
    else
      @@numbers[n]
    end
  end
end
```

5.2.6. テストの実行

テストの実行コマンドは `crystal spec` です。プロジェクトディレクトリの直下でこのコマンドを実行すると用意したテストケースが順に評価され、実際の挙動が指定された内容通りかどうかチェックされます。

テスト実行結果

```
$ crystal spec
...
Finished in 203 microseconds
3 examples, 0 failures, 0 errors, 0 pending
```

テスト結果の先頭行には、テストケース (`it` ブロック) ごとの実行結果がそれぞれ1文字で表示されます。表示されたのが `.` であれば、そのテストケースには合格できたことを表しています。もしここに `.` 以外の文字が表示されているようであれば、対応した `it` ブロック内でテストした機能の実装に問題があることになります。先頭行がすべて `.` になるまで、ソースコードの修正とテストを繰り返しましょう。

今回の `spec` ファイルには `it` ブロックが3つあり、上の例では `.` が3つ並んでいます。ということは、実装したソースコードは当初想定した通りの挙動をしているようです。

このように、テストが成功した時点では shard の機能面は完成です。実際、この状態でも GitHub へ公開す

れば、shard として使用できてしまったりします。しかし、曲がりなりにも一般公開する shard には、それなりの体裁といったものもありますので、もうひと手間ふた手間かけてみることにしましょう。

5.2.7. README を書く

ひな形生成時に用意されている `README.md` ファイルは、以下の内容をカバーするように構成されています。

`README.md` ひな形の構成

`# shard 名`

README のタイトルに当たるパート。shard の概要を記載する。

`## Instration`

shard をインストールするための手順を記載するパート。`shard.yml` へ追加する `dependencies` の内容は用意されている。shard 以外の外部ライブラリやコマンドに依存している場合はここで説明しておいた方が良い。

`## Usage`

ソースファイル内での shard の使い方を記載するパート。shard の `require` 方法は用意されている。最低限のサンプルコードくらいは書いておくべき。

`## Development`

開発者向け情報を記載するパート。shard の開発に参加したり、自分で改造したりしたい人向けの情報を記載する。積極的に開発者を募るつもりがなければ項目ごとオミットしても良いかも。

`## Contributing`

shard の開発に参加してくれる人向けの参加手順を記載するパート。積極的に開発者を募るつもりがなければ項目ごとオミットしても良いかも。

`## Contributors`

shard の作成に携わった人のリストを記載するパート。

また、このひな形は GitHub での公開を前提としています。そのため、GitHub で shard を公開する場合には、以下の2点を修正するだけで標準的な README として利用できるようになっています。

1. 何箇所かある `[your-github-name]` を自分の GitHub アカウントに置き換える
2. `TODO:` の部分を埋めていく

なお、ひな形の中で登場する `TODO:` は以下の3箇所です。

`README.md` 中に登場する `TODO:` 項目

`TODO: Write a description here`

その shard の概要。最低限、なにをする shard なのがくらいは書いておいた方が良い。

`TODO: Write usage instructions here`

その shard の使い方。ある程度サンプルコードで代用可能。

`TODO: Write development instructions here`

その shard の開発に関わろうとする人向けの情報。

`fibonacci` はメソッドが1つだけしかなく機能もシンプルなので、`Usage` は最低限のサンプルコードで済ませることにします。また、共同開発者を積極的に募るほどの者でもありませんので、最低限の項目を記載した `README.md` は以下のようになりました。

完成した `README.md`

```
# fibonacci

The fibonacci number calculator for the Crystal programming language.

## Installation

Add this to your application's 'shard.yml':

```yaml
dependencies:
 fibonacci:
 github: github_name/fibonacci
```

## Usage

```crystal
require "fibonacci"

f0 = Fibonacci.number(0) #=> 0_big_i
f1 = Fibonacci.number(1) #=> 1_big_i
f2 = Fibonacci.number(2) #=> 1_big_i
f3 = Fibonacci.number(3) #=> 2_big_i
f4 = Fibonacci.number(4) #=> 3_big_i
```

## Contributors

- [github_name](https://github.com/github\_name) Your Git Name - creator, maintainer
```

5.2.8. ライセンスを選ぶ

プロジェクトのひな形では、`shard` を公開する際のライセンスとして、`MIT` ライセンスが選択されています。

`shard.yml` の `license` には `MIT` が指定されており、プロジェクトディレクトリ内の `LICENSE` ファイルも `MIT` ライセンスの条項が記載されています。`MIT` ライセンスとは、ざっくりいうと「著作権表記とライセンス条項さえ含まれていれば有償無償問わず自由に使って良い、ただし無保証」というものです。ソフトウェアの利用者としてはかなり自由に利用でき、開発者としては免責が明言されているため、比較的使いやすいライセンスの1つです。大した量ではありませんので、詳細については一度 `LICENSE` ファイルの内容を読んでみてください。

もし `MIT` 以外のライセンスを採用したければ、`LICENSE` ファイルの内容を使用したいライセンス条件に書き換えて `shard.yml` の `license` を変更することになります。このとき、`shard.yml` の `license` には OSI

(Open Source Initiative) で定義されたライセンス名か、ライセンスの参照先 URL を指定可能です。

OSI ライセンス名

<https://opensource.org/licenses/alphabetical>

今回は特にライセンスへのこだわりもありませんので、MIT ライセンスのままで行こうと思います。

5.2.9. GitHub へ公開する

ここまで準備が完了したら、プロジェクトを GitHub へ公開しましょう。この辺りの手順は GitHub のドキュメントなどを参照してください。

プロジェクトが GitHub へ公開された結果、別のプロジェクトが `shard.yml` に以下の記述を追加すると `fibonacci` をインストール可能になりました。

```
dependencies:  
  fibonacci:  
    github: github_name/fibonacci
```

ここまで shard を作って公開するまでの手順は完了です。

ただし、この状態では `shards` コマンドが `shard` のバージョンを認識できません。`fibonacci` を使おうとするプロジェクトで `shard.yml` に特定のバージョンを指定していても、常に最新状態（リポジトリの HEAD）の `shard` がインストールされてしまいます。

`shard` を作る手順の最後に、`shard` のバージョン管理方法を紹介しましょう。

5.2.10. `shard` のバージョン管理

毎度おなじみ「`shard.yml specification.`」には、`shard` のバージョンとして以下のようなルールが示されています。

- セマンティックバージョニングに従うことが望ましい
- 数字が含まれていること
- . や - を使用しても良いが、連続しないこと

例

`0.0.1`、`1.2.3`、`2.0.0-rc1` など

強制はされないものの、セマンティックバージョニングのような合理的なバージョン付けを強く推奨。

後方互換が損なわれる大きな変更でも、内部の子細なバグフィックスでもメジャーバージョンが上がるようでは、使う側としてバージョンの変化からバージョンアップのインパクトを測りかねてしまいます。

セマンティックバージョニングではメジャー/マイナ/パッチの3つの数字でバージョンを構成し、パッチよ

りマイナ、マイナよりメジャーバージョンが上がることのインパクトが大きくなるよう定義されています。`shard.yml` 内でバージョンを指定する際の `~>` 演算子は、セマンティックバージョニングのような、先頭に近い数字の変更がインパクトが大きい際に有効に働きます。

セマンティックバージョニング

<https://semver.org/lang/ja/>

3つのバージョン情報

`fibonacci` プロジェクトでは、プロジェクトファイルの2箇所に `shard` のバージョンが登場します。まず `shard.yml` ファイル2行目の `version`、もう1つがソースファイルで定義されている `Fibonacci::VERSION` 定数です。この2つの値は、ひな形生成時にはどちらも `0.1.0` となっており、これがプロジェクト (`shard`) のバージョンになります。

前述の通り、`shard.yml` とソースファイルにバージョンが書かれていても、`shards` コマンドはそのバージョンを認識できません。

このとき使用されるのが第三のバージョン情報が、git リポジトリのバージontタグです。

`shard.yml` の `dependencies` 内で、ある `shard` のバージョンとして `1.0.0` が指定されていたとしましょう。このとき、`shards` コマンドは、その `shard` のリポジトリから `v1.0.0` とタグ付けされたコミットの状態をインストールしようとします。

つまり、先ほどプッシュした `fibonacci` の GitHub リポジトリに `v0.1.0` というタグを付けることで、現状の `fibonacci` がバージョン `0.1.0` だと明示できます。なお、GitHub ではリリース管理機能を使って新しいリリースを追加する際に、Web UI からバージontタグを設定できて便利です。

もし `fibonacci` の機能に修正を加えてバージョンを `0.2.0` へ上げる場合には以下の手順で行うことになるでしょう。こうしておくと、複数のプロジェクトからそれぞれ異なるバージョンの `fibonacci` を利用可能になります。

1. `fibonacci` の機能を修正
2. ソースファイル上の `Fibonacci::VERSION` 定数を `0.2.0` に変更
3. `shard.yml` 2行目の `version` を `0.2.0` に変更
4. 変更をコミットし、GitHub 上の master ブランチへマージ
5. GitHub 上で変更がマージされた master ブランチにバージontタグ `v0.2.0` を追加

というわけで、以上が `shard` を作って公開し、さらに他のプロジェクトから使用してもらうために必要な作業の流れになります。

是非みなさんも自作のライブラリを `shard` として公開してみてください。世界中には自分と同じことで困っていて、自分だけしか使わないだろうと思っていた機能を便利に使ってくれる人が意外といたりしますよ。

5.3. テスト

この章の最後に、shard を作る際にも使用したテストについて簡単にご紹介しましょう。

ここでいうテストはソフトウェアに対するユニットテストの一種で、プログラムが想定した通りの挙動を取るかどうかを調べる機能試験に相当します。これは、あらかじめ「このメソッドにこういった引数を与えると、こんな結果になるはず」というプログラムの挙動（テストケース）を列挙しておき、実際にそうなるかどうかを個々に評価するような仕組みです。

Crystal には、RSpec を参考にしたテスト機能を提供するモジュール `Spec` がコンパイラ自身の標準添付ライブラリとして用意されています。また、標準のプロジェクトひな形にテスト用のファイル一式が含まれているなど、言語自体がテストを強く意識した作りになっています。

5.3.1. テストの使い方

細かい説明は後にして、まずはテストの使い方をざっくり見てみましょう。

spec ファイルを作る

まず、テストにはテストケースを定義した spec ファイルが必要です。spec ファイルは、"`spec`" とテスト対象となるコードを `require` した Crystal のソースファイルで、テストケース自体も Crystal のコードとして定義します。

さて、ここでは簡単な例として "がおー" と吠える `Bear` クラスを実装する場合を考えてみましょう。`Bear` クラスが持つインスタンスマソッドは吠え声を文字列で返す `#bark` のみ、ソースファイル名は `bear.cr` です。spec ファイルの名前はテスト対象とするファイルのベースネームに続けて `_spec.cr` を加えたものが使用されることが一般的です。今回もその例に倣い、熊が正しく "がおー" と吠えるかどうかテストする `bear_spec.cr` を、ソースファイルと同じディレクトリに作ってみました。

`bear_spec.cr`

```
require "spec"
require "./bear"

describe Bear do
  describe "#bark" do
    it "returns \"がおー\"." do
      Bear.new.bark.should eq "がおー"
    end
  end
end
```

最初の2行は `Spec` モジュールとソースコードの `require` 文で、4行目以降がテストケースの定義部分になります。

まず登場するのが `describe Bear` で宣言されたブロックです。これは、その内部で行われるテストが `Bear` 型を対象としていることを明示しています。`describe Bear` ブロックの中には、もう1つ `describe "#bark"` ブロックが置かれています。こちらのブロックはその内部で `Bear` 型の中でも `#bark` メソッドに関するテストを行うことを宣言しています。

続いて登場するのがテストケースに相当する `it` ブロックです。`it` ブロックでは、`describe` で宣言された対象となるメソッドの、ある1つの振る舞いに関するテストを行います。また、`it` ブロックには引数としてその内部で行うテスト内容の説明文を与えることができます。この例では、`it (Bear#bark メソッド)` が、正しく "がおー" という文字列を返すかどうかについてテストすることが説明されています。

そして、`it` ブロックの内で実行される `Bear.new.bark.should eq "がおー"` が実際に評価されるテスト内容の記述です。`Bear.new.bark` が "がおー" と等しく (eq) あるべき (should) といったように、実際に評価している内容が（英語として）比較的自然に読める構文になっています。

プログラム本体を実装する

さて、spec ファイルが用意できたので次に `Bear` クラスそのものを `bear.cr` へ実装していくのですが、ついうっかり熊に "わん" と吠えさせてしまいました。

実装ミスを含んだ `bear.cr`

```
class Bear
  def bark
    "わん"
  end
end
```

この状態でテストを実行してみるとどうなるでしょうか。なお、テストの実行コマンドは `crystal spec` です。特定の spec ファイルを対象としたい場合には、その spec ファイル名をパラメータとして与えてください。

テストの実行結果

```
$ crystal spec bear_spec.cr
F

Failures:

1) Bear #bark returns "がおー".
Failure/Error: Bear.new.bark.should eq "がおー"

  Expected: "がおー"
  got: "わん"

# bear_spec.cr:7

Finished in 127 microseconds
1 examples, 1 failures, 0 errors, 0 pending

Failed examples:

crystal spec bear_spec.cr:6 # Bear #bark returns "がおー".
```

テスト結果の先頭行には、テストケース (`it` ブロック) ごとの実行結果サマリが、それぞれ1文字で出力されます。今回はテストケースが1つしかありませんので、表示されているのは `F` が1文字だけです。テス

ト結果 `F` は、実際の挙動がテストケースで指定された条件とは異なる結果を示したため、テストが不合格になった場合に表示されるものです。

このように何らかの理由でテストが不合格になると、`Failures:` に続いて具体的な問題点が出力されます。ここでは、期待された値（`Expected:`）が "がおー" のに対して、実際の結果（`got:`）が "わん" になっていたことが確認できます。このように、テストを行うと、実装されたコードが想定外の挙動を示した際に、どの部分がどのように異なっていたのかを具体的に知ることができます。

テスト結果を元にソースを修正する

テスト結果から問題の箇所は明らかですので、`bear.cr` の実装を修正しました。

修正後の `bear.cr`

```
class Bear
  def bark
    "がおー"
  end
end
```

もう一度テストを実行してみると、今度は `F` ではなく `.` が表示されましたので、実装したコードが当初想定した通りの機能を実現できていることが確認できました。

テストの実行結果（ソース修正後）

```
$ crystal spec bear_spec.cr
.

Finished in 53 microseconds
1 examples, 0 failures, 0 errors, 0 pending
```

このように、まず実装しようとする機能の挙動をテストケースとして定義し、テストケースを満足するように実装を進める開発手法を、テスト駆動開発や振る舞い駆動開発などと呼びます。この手法のメリットとしては、実装コードの品質が担保されるという点はもちろんありますが、テストケース自体がある意味で機能仕様やサンプルコードとして利用できるという点も見逃せません。チームで開発している場合や他の人が書いたコードを引き継ぐ場合などでも、適切なテストケースが用意されていれば最低限の情報はそこから入手可能です。

5.3.2. テスト関連の構文

では次に、`spec` ファイルで使用する各構文についてみてみましょう。

グルーピング用ブロック

グルーピング用ブロックは、1つ以上のテストケースを何らかの基準でまとめるために使用されます。グルーピング用ブロックの内部には次に紹介するテストケース用のブロックだけでなく、別のグルーピング用ブロックをネストさせることも可能です。

先の例で登場した `describe` もグルーピング用ブロックの1種です。`describe` ブロックはテスト対象を元に

したグルーピングを行うもので、テスト対象として型（クラス、モジュールなど）やメソッド名（文字列で指定）を引数として与えることができます。この引数はテスト不合格時の出力メッセージでも利用されるため、テスト対象となるメソッド名に準じた構成をとることが強く推奨されています。例えば、外側の `describe` では型を指定し、内側の `describe` にメソッド名を渡すような形です。またこのときのメソッド名に、クラスメソッドであれば `.` を、インスタンスメソッドであれば `#` を先頭に付けておくと、`Class #method` のように準的なメソッド表記に近い形を再現できます。

Crystal の spec ファイルでは、グルーピング用のブロックとして `describe` 以外にもう1つ `context` が使用可能です。`describe` ブロックがテスト対象をもとにしたグルーピングだとすると、`context` ブロックはその時の状況（条件）によるグルーピングです。例えば、配列が要素を含んでいるか否かで挙動が異なる場合に、それぞれの状況を明示することができます。

`describe` と `context`

```
require "spec"

describe Array do
  describe "#first?" do
    context "self is empty." do
      it "returns nil." do
        Array(Int32).new.first?.should be_nil
      end
    end
  end

  context "self has elements." do
    it "returns receiver's first element." do
      [1, 2, 3].first?.should eq 1
    end
  end
end
end
```

テストケース用ブロック

こちらの例としては `it` がすでに登場しています。`it` はあるメソッドがもつ特定の挙動1つについてのテストケースを定義するためのブロックです。ですので、複数の挙動に対するテスト内容を1つの `it` ブロック内に記述すべきではありません。例えば、オブジェクトの状態にや引数の値によって異なる挙動を示すような場合は、同じメソッドであってもそれぞれに独立した `it` ブロックを設けてテストケースを定義すべきです。

テストケースを書くためのブロックには `it` ともう1つ、`pending` があります。`pending` ブロックは文字通り、一時的にそのテストケースの実行を保留する場合に使用します。例えば、機能仕様としてのテストケースは用意されたものの対象となるメソッドがまだ実装されていないような場合に、`it` の代わりに `pending` ブロックを使用すると良いでしょう。ただし、あくまで一時的に保留しておくだけですので、最終的には `pending` ブロックを `it` ブロックに置き換えてテストに合格できるようにしなければいけません。

it と pending

```
require "spec"
require "./bear"

describe Bear do
  describe "#bark" do
    it "returns \"がおー\"." do
      Bear.new.bark.should eq "がおー"
    end
  end

  describe "#bite" do
    # Bear#bite は未実装
    pending "retuns \"がぶがぶ\"." do
      Bear.new.bite.should eq "がぶがぶ"
    end
  end
end
```

オブジェクトの状態を確認する

`require "spec"` が実行されると、全てのオブジェクトに対して `#should` と `#should_not` という2つのインスタンスマソッドが追加されます。`#should` は、レシーバの状態に対する条件 (`eq "がおー"` など) を引数に取り、実際の値がその条件を満たしていないとテストが不合格になります。一方、`#should_not` は逆に条件を満たしてしまうとテストが不合格になります。

オブジェクトの状態に対する評価条件には以下のような種類があります。一部の評価条件は特定のインスタンスマソッドを持たないオブジェクトに対しては使用できません。例えば、`should be` を使用するには `#same?` が実装されている必要があります。

使用可能なオブジェクトの状態を評価する

`actual.sould eq expected`
actual が expected と等しいかどうか。
条件 : `actual == expected`

`actual.sould be expected`
actual が expected と同値とみなせるかどうか。
条件 : `actual.same?(expected)`

`actual.sould be_a expected`
actual が expected 型の値かどうか。
条件 : `actual.is_a?(expected)`

`actual.sould be_nil`
actual が nil かどうか。
条件 : `actual.nil?`

`actual.should be_true`
actual が true かどうか。

条件 : `actual == true`

`actual.should be_false`

`actual` が `false` かどうか。

条件 : `actual == false`

`actual.should beTruthy`

`actual` が `if` 文で真とみなされるかどうか。

条件 : `actual` が `false`、`nil`、`Pointer.null` のいずれでもない

`actual.should beFalsey`

`actual` が `if` 文で偽とみなされるかどうか。

条件 : `actual` が `false`、`nil`、`Pointer.null` のいずれか

`actual.should be < expected`

`actual` が `expected` より小さいかどうか。

条件 : `actual < expected`

`actual.should be <= expected`

`actual` が `expected` 以下かどうか。

条件 : `actual <= expected`

`actual.should be > expected`

`actual` が `expected` より大きいかどうか。

条件 : `actual > expected`

`actual.should be >= expected`

`actual` が `expected` 以上かどうか。

条件 : `actual >= expected`

`actual.should beClose(expected, delta)`

`actual` と `expexted` の差が `delta` 以下かどうか。

条件 : `(actual - expected).abs <= delta`

`actual.should contain expected`

`actual` が `expexted` を含むかどうか。

条件 : `actual.includes?(expected)`

`actual.should match expected`

`actual` が `expexted` にマッチするかどうか。

条件 : `actual =~ expected`

例外の発生を確認する

テストケース内で `expext_raises` ブロックに引数として例外の型を指定してすると、ブロック内で指定した型の例外が発生することを確認できます。

例外チェック

```
context "self is empty." do
  it "raises an IndexError." do
    expect_raises(IndexError) {
      # IndexError が発生しなければテスト失敗
      Array(Int32).new.first
    }
  end
```

このとき、引数に文字列もしくは正規表現オブジェクトを追加することで、エラーメッセージに対する条件を指定することも可能です。

メッセージ条件付き例外チェック

```
expect_raises(SomeError, "message") { … }
```

エラーメッセージに "message" が含まれる SomeError 型の例外が発生しなければ不合格。

```
expect_raises(SomeError, /pattern/) { … }
```

エラーメッセージが /pattern/ にマッチする SomeError 型の例外が発生しなければ不合格。

なお、発生した例外の状態を他の評価条件でテストしたい場合、`expect_raises` の返り値として例外インスタンスを取得できます。

5.3.3. テスト実行結果の見方

テストの実行結果は大きく分けると以下のパートから構成されます。

1. テストケースの実行結果サマリ
2. 不合格となったテストケースの詳細情報
3. テストの実行に要した時間
4. テスト結果の統計情報
5. 不合格となったテストケースの一覧

このうち、2. および 5. については、全てのテストケースを問題なく通過できた場合には表示されません。

ソース修正前の `bear_spec.cr` の実行結果を例に、それぞれの内容を詳しくみてみましょう。

失敗したテストの実行結果（再掲）

```
$ crystal spec bear_spec.cr
F

Failures:

1) Bear #bark returns "がおー".
Failure/Error: Bear.new.bark.should eq "がおー"
Expected: "がおー"
got: "わん"

# bear_spec.cr:7

Finished in 127 microseconds
1 examples, 1 failures, 0 errors, 0 pending

Failed examples:

crystal spec bear_spec.cr:6 # Bear #bark returns "がおー".
```

テストケースの実行結果サマリ（1行目）

テストケースの実行結果には成功（.）、不合格（F）、エラー（E）、ペンディング（*）の4種類があります。テストの実行結果では、先頭行にテストケースの数だけ実行結果に対応した文字が並んで表示されます。もしテストを実行したのがカラー表示対応のターミナルであれば . は緑、F と E が赤、* が黄色で表示されているかもしれません。

ここで表示される実行結果が全て . になっていれば、テストに合格できたことになります。

実行結果の種類

成功（.）

テストケース内の評価条件を全てパスし、想定外の例外やエラーも発生しなかった場合。

不合格（F）

テストケース内の評価条件を満たさない項目が存在したり、想定される例外が発生しなかった場合。

エラー（E）

テストケース内で `expect_raises` で補足されない想定外の例外が発生した場合

ペンディング（*）

テストケースが `pending` ブロックとして定義されており、テストの実行を保留している場合。

不合格となったテストケースの詳細情報（3~11行目）

Failures: に続いて、不合格となったテストケースの具体的な問題点が出力されます。

不合格テストケースの情報

- 不合格テストケースの通し番号とテストケースの説明（3行目）
- 不合格の原因となったチェック項目（4行目）
- 想定された状態（Expected:）と実際の状態（got:）（6～7行目）
- 該当チェック項目があるファイル名と行番号（9行目）

特に3.で表示される想定値と実際の値との比較は、問題解決の大きな助けになってくれることでしょう。

テストの実行に要した時間（13行目）

テスト全体にかかった時間ですので、あくまで参考までに。

テスト結果の統計情報（14行目）

ここには examples、failures、errors、pending と4種類の数値が出力されます。

examples が spec ファイルに定義されたテストケースの数、後ろ3つは、実行結果が不合格/エラー/ Pending となったテストケースの数です。

ここで後ろ3つの値がゼロになっていれば、テストに合格したことになります。

不合格となったテストケースの再テスト用コマンド一覧（16～18行目）

最後に、不合格となったテストケースについて、個別に再テストする場合に使える crystal spec コマンドが一覧で表示されます。

リスト各行の # から後ろは、どのテストケースについてのものなのかを識別するためのコメントです。

よりも前の部分、この例でいう crystal spec bear_spec.cr:6 を実行することで、問題のあるテストケースだけを対象としてテストを実行できます。

5.3.4. crystal spec コマンドのパラメータ

テストを実行する crystal spec コマンドには、spec ファイル名を以外のパラメータを指定することができます。適切なパラメータを与えることで、ディレクトリ単位や spec ファイル単位、さらには特定のテストケース単位でのテストを実行可能です。

crystal spec コマンドへのパラメータ指定

crystal spec

パラメータを省略した場合、spec/**/*_spec.cr にマッチする全ての spec ファイルを対象とする。

crystal spec dir/

ディレクトリを指定した場合。dir/**/*_spec.cr マッチする全ての spec ファイルを対象とする。

crystal spec spec_file.cr

spec ファイルを指定した場合。spec_file.cr 単体を対象とする。

crystal spec file.cr:10

spec ファイルと行数を指定した場合。spec_file.cr ファイルの10行目から始まるブロックだけを対象とする。

大規模なソースコードをのテストを実施する際には、これらのパラメータを有効活用することでテスト効率を向上できるかもしれません。

なお、`crystal init` コマンドで生成されるプロジェクトひな形ではテスト関連のファイルは `spec` ディレクトリに収められています。そのため、`spec` ファイルを指定しなくても `crystal spec` とだけ打てばテストが実行可能になっています。

5.3.5. `spec` ヘルパファイル

プロジェクトひな形の `spec` ディレクトリには、`spec` ファイル以外に `spec` ヘルパファイル `spec_helper.cr` が用意されています。デフォルトでは、"`spec`" とプロジェクトのソースファイルを `require` するだけの内容になっており、`spec` ファイルから読み込まれています。

`spec` ヘルパは、`spec` ファイルが1つだけの場合それほどありがたみを感じないかもしれません。しかし、プロジェクトが大規模になり、いくつもの `spec` ファイルを駆使するようになると `spec` ヘルパファイルが効果を発揮するようになります。例えば、テスト用のオブジェクト生成やテスト用の特別な環境設定など、複数の `spec` ファイルで共通的に使用するメソッドを `spec` ヘルパファイルに書いておけばコードの重複を防ぐことが可能です。

テストはコードのクオリティを確保する上で重要な工程ですが、慣れないうちは面倒臭く感じることも多いと思います。`spec` ヘルパのような仕組みをうまく活用して、効率よくテストする方法を模索してみてください。

5.4. まとめ

shard を利用する手順

1. shard を利用するプロジェクトを作成する
2. `shard.yml` に使用したい shard の記述を追加する
3. shard をプロジェクトディレクトリにインストールする
4. ソースコードの先頭で shard を `require` する
5. ソースコード内で shard が提供する機能を使用する

shard を作る手順

1. shard 用のプロジェクトを作成する
2. `shard.yml` にメタデータを記述する
3. テストのテストケースを書く
4. テストケースを満足させるようにソースファイルに機能を実装する
5. テストを実行、問題がなくなるまでソースの修正とテストを繰り返す
6. README を書く
7. ライセンスを選ぶ
8. GitHub などのリポジトリへ公開する
9. git リポジトリにバージョンタグを付ける

テストの構文

`describe` ブロック

テスト対象でテストケースをグルーピングする。

`context` ブロック

条件や状況でテストケースをグルーピングする。

`it` ブロック

ある振る舞いに対するテストケースを記述する。

`pending` ブロック

一時的にテストを保留するテストケースに使用する。

`Object#should` メソッド

レシーバがある条件を満たさなければテストに不合格とする。

`Object##should_not` メソッド

レシーバがある条件を満たした場合にテストに不合格とする。

`expect_raises` ブロック

ブロック内で指定された例外が発生しなかったらテストに不合格とする。

6. Web 開発

著者: msky

この章では、Crystal を用いて Web 開発を行う方法について解説します。
本サンプルで使用する Crystal のバージョンは **0.26.0** で確認しています。

6.1. Crystal での Web 開発の前提知識

まず、Crystal では HTTP でのサービスをどのように処理するかについて解説します。
前の章でも既に解説済みですが、改めて Crystal での Web 開発の基本となる部分について解説します。

以下のコードで解説します。

server.cr

```
require "http/server"

server = HTTP::Server.new do |context|
  context.response.content_type = "text/plain"
  context.response.print "Hello world"
end

puts "Listening on http://0.0.0.0:8080"
server.listen(8080)
```

ポート 8080 での HTTP アクセスを Listen する最小のプログラムです。

以下のコードを実行することで簡易 HTTP サーバとして機能します。

localhost:8080 にアクセスすると **Hello world** が表示されます。

```
$ crystal server.cr
Listening on http://0.0.0.0:8080
```

6.1.1. HTTP ハンドラ

基本的に Crystal の HTTP サーバが返却するパラメータは レスポンス本文、ヘッダ、ステータスコードの3種です。

その3種を持っている限りにおいて、処理をスタックすることができます。

この辺りの規約は Ruby でいうところの Rack に似ているといえます。

HTTP ハンドラを実装するには Rack 同様に call メソッドを定義し、引数に **HTTP::Server::Context** 型のオブジェクトを渡すという形式になっています。

以下に例を記載します。

```

require "http/server"

class InterruptTestFirst
  include HTTP::Handler

  def call(context)
    response = call_next(context)
    puts "aa"
    response
  end
end

class InterruptTestSecond
  include HTTP::Handler

  def call(context)
    response = call_next(context)
    puts "bb"
    response
  end
end

server = HTTP::Server.new([InterruptTestFirst.new, InterruptTestSecond.new]) do
|context|
  context.response.content_type = "text/plain"
  context.response.print "Hello world"
end

puts "Listening on http://0.0.0.0:8080"
server.listen(8080)

```

上記の例で言いますと、`InterruptTestFirst` `InterruptTestSecond` の順にスタックに積されます。実行されるのは先入れ後出しなので、後に積まれた `InterruptTestSecond` からになります。

`curl -X GET http://localhost:8080` を実行することで以下の内容がコンソールに出力されます。

```
bb
```

```
aa
```

主に Crystal で Web 開発を行う上での基本的な内容ですが、実際に開発する場合は何らかの Web フレームワークを用いることになるかと思います。
次項から Web フレームワークについて解説します。

6.1.2. Crystal の Web フレームワーク

Crystal で現在最も活発に開発されている Web フレームワークが Kemal です。Ruby でいうところの Sinatra に似た設計思想で作られており、シンプルな実装で Web アプリを作成する

ことが出来ます。

本書では主に Kemal を用いて簡単な Web アプリを作成しながら Web 開発の方法を解説していきます。

Web サイト

<http://kemalcr.com/>

GitHub

<https://github.com/kemalcr/kemal>

また Kemal 以外の主なフレームワークとして以下のものがあります。

Amber

<https://www.amberframework.org>

フルスタックの Web フレームワークです。

様々なコードジェネレータや、 OR/M を備えています。 Rails 的な規約重視のフレームワークです。

6.2. Kemal による Web 開発

それでは本章から Kemal による Web 開発を、サンプルを交えながら説明します。

また本サンプルに使用する Kemal のバージョンは **0.24.0** を想定しています。

6.2.1. Kemal インストール

適当な作業用のディレクトリ以下で、以下のコードを実行してみます。ディレクトリが作成され幾つかのファイルがひな形から作成されます。

本稿ではサンプルアプリを `kemal-sample` として進めます。

```
$ crystal init app kemal-sample  
$ cd kemal-sample
```

`shard.yml` ファイルに以下の内容を追記します。

`shard.yml`

```
dependencies:  
  kemal:  
    github: kemalcr/kemal  
    version: 0.24.0
```

以下のコマンドを実行することで Kemal 本体をインストールすることが出来ます。

```
$ shards install
```

6.2.2. Kemal FirstStep

インストールまで問題なく動かせたら続いて簡単なサンプルを作成してみましょう。
カレントの src ディレクトリ以下の `kemal-sample.cr` に以下の追記を行います。

まず行頭に以下の行を足します。

```
require "kemal"
```

続いて `module Kemal::Sample` 内に以下の内容を追記します。

```
get "/hello" do |env|
  hello = "Hello World"
  render "src/views/hello.ecr"
end
```

末尾の行に以下の内容を追記します。

```
Kemal.run
```

画面側の処理を作成します。

`src` 以下に `view` ディレクトリを作成し、以下の `hello.ecr` ファイルを `view` ディレクトリ内に作成します。

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>hello</title>
</head>
<body>
  <%= hello %>
</body>
</html>
```

Kemal の基本的な使い方として、REST の動詞 GET POST PUT DELETE を以下の形で記述します。

```
get "/path" do |env|
  # 処理
end

post "/path" do |env|
  # 処理
end
```

詳しい内容については後述します。

まずは記述したら、以下のコマンドでビルド、実行します。

```
$ crystal build src/kemal-sample.cr  
$ ./kemal-sample
```

ブラウザで <http://localhost:3000/> でアクセスします。

Hello World と表示されていれば問題ありません。

6.2.3. DB と連動する

通常、Web アプリでは DB が必須です。Kemal で作ったアプリから DB にアクセスすることも可能です。

ライブラリを使用することで PostgreSQL か、もしくは MySQL を使用することが出来ます。

今回は PostgreSQL を使用します。

テーブル 2. テーブル名 *articles*

| カラム名 | 型 |
|---------|--------|
| id | serial |
| title | text |
| content | text |

DB を作成します。

```
$ createdb kemal_sample_development -O your_owner
```

DDL を作成しロードします。

```
create table articles (  
    id      serial primary key,  
    title   text,  
    content text  
);
```

```
$ psql -U your_owner -d kemal_sample_development -f sql/create_articles.sql
```

作成した後、shard.yml ファイルに以下の内容を追記します。

```
dependencies:
  kemal:
    github: kemalcr/kemal
    version: 0.24.0
  db:
    github: crystal-lang/crystal-db
  pg:
    github: will/crystal-pg
```

編集後、以下のコマンドを実行します。

```
$ shards install
```

6.2.4. 投稿一覧ページの編集

これからいよいよ Web アプリらしく記事の一覧ページと詳細ページ、新規投稿ページをそれぞれ作成していきます。

まず全ページで共通で使用するテンプレートは別に作成します。

テンプレートヘッダに新規投稿ページと投稿リストページへのリンクを表示し、ページ内に各ページのコンテンツを表示するように修正していきます。

6.2.5. レイアウトページの作成

まずひな形のページを `application.ecr` という名前で作成します。

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>kemal sample</title>
  <!-- bootstrapを使用する -->
  <link
    rel="stylesheet"
    href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css">
  <link rel="stylesheet" href="/css/custom.css">
</head>
<body>
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <a id="logo">sample app</a>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><a href="/articles">ArticleList</a></li>
        <li><a href="/articles/new">新規投稿</a></li>
      </ul>
    </nav>
  </div>
</header>
<div class="container">
  <%= content %>
</div>
</body>
</html>

```

本サンプルでは BootStrap を使用します。

CDN を使用しますので特にダウンロード不要ですが、ダウンロードする場合は別途 <http://getbootstrap.com/> から必要なファイルをダウンロードし、プロジェクトカレントの `/public` 以下に配置してください。

6.2.6. CSS の作成

ページ修飾用の CSS を作成します。本サンプルでは rails tutorial をそのまま参考にします。

```

body {
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
  resize: vertical;
}

```

```
.center {
  text-align: center;
}

.center h1 {
  margin-bottom: 10px;
}

h1, h2, h3, h4, h5, h6 {
  line-height: 1;
}

h1 {
  font-size: 3em;
  letter-spacing: -2px;
  margin-bottom: 30px;
  text-align: center;
}

h2 {
  font-size: 1.2em;
  letter-spacing: -1px;
  margin-bottom: 30px;
  text-align: center;
  font-weight: normal;
  color: #777;
}

p {
  font-size: 1.1em;
  line-height: 1.7em;
}

#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: #fff;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
}

#logo:hover {
  color: #fff;
  text-decoration: none;
}
```

6.2.7. 一覧ページの作成

記事一覧ページを以下の内容で作成します。

```
<h2>Article List</h2>





```

6.2.8. 新規投稿ページ

記事投稿用のページを新規作成します。

以下のファイルを追加します。

```
<h2>新規投稿</h2>
<form method="post" action="/articles">
<input type="text" name="title" size="10" maxlength="10" />
<br />
<br />
<textarea name="content" cols="40" rows="4"></textarea>
<br />
<br />
<input type="submit" value="post">
</form>
```

6.2.9. 詳細ページ

一覧ページから記事タイトルをクリックした時に遷移する詳細ページを作成します。
データへの更新については後述します。

```
<h2>Article</h2>
<% articles.each do |article| %>
  <h3><%=article["title"] %></h3>
  <p><%=article["content"] %></p>
  <!-- 更新 -->
  <a href="/articles/<%=article["id"] %>/edit" target="_top" class="btn btn-primary">edit</a>
  <br />
<% end %>
```

6.2.10. Kemal プログラム改修

kemal-sample.cr を以下の内容で修正します。

行頭を以下の内容に修正します。

```
require "kemal"
require "db"
require "pg"
```

続いて `module Kemal::Sample` 内に以下の内容を追記します。

```

database_url = "postgres://localhost:5432/kemal_sample_development"
db = DB.open(database_url)

["/", "/articles"].each do |path|
  get path do |env|
    articles = [] of Hash(String, String | Int32)
    db.query("select id, title, content from articles") do |rs|
      rs.each do
        article = {} of String => String | Int32
        article["id"] = rs.read(Int32)
        article["title"] = rs.read(String)
        article["content"] = rs.read(String)
        articles << article
      end
    end
    db.close
    render "src/views/index.epr", "src/views/application.epr"
  end
end

get "/articles/new" do |env|
  render "src/views/articles/new.epr", "src/views/application.epr"
end

post "/articles" do |env|
  # env.params.bodyでformのvalueを取得できます
  title_param = env.params.body["title"]
  content_param = env.params.body["content"]
  params = [] of String
  params << title_param
  params << content_param
  # update, insert, deleteは以下のようにexecでアップデートを実行します
  db.exec("insert into articles(title, content) values($1::text, $2::text)", params)
  db.close
  env.redirect "/"
end

get "/articles/:id" do |env|
  articles = [] of Hash(String, String | Int32)
  article = {} of String => String | Int32
  id = env.params.url["id"].to_i32
  params = [] of Int32
  params << id
  sql = "select id, title, content from articles where id = $1::int8"
  article["id"], article["title"], article["content"] =
    db.query_one(sql, params, as: {Int32, String, String})
  articles << article
  db.close
  render "src/views/articles/show.epr", "src/views/application.epr"
end

```

修正は以上です。次項から処理について解説します。

6.2.11. クエリ

まずクエリは以下のように記述します。

```
db.query("select id, title, content from articles") do |rs|
  rs.each do
    article = {} of String => String | Int32
    article["id"] = rs.read(Int32)
    article["title"] = rs.read(String)
    article["content"] = rs.read(String)
    articles << article
  end
end
```

query メソッドに SQL クエリを記述し、ループ内で結果を格納していきます。

1件だけ取得したい場合は `query_one` もしくは `query_one?` を使います。後者は1件もデータが無い場合がありうる場合に使います。

```
sql = "select id, title, body from articles where id = $1::int8"
article["id"], article["title"], article["body"] =
  db.query_one(sql, params, as: {Int32, String, String})
```

`query_one` の戻り値は、`as` で指定した型の `Touple` になります。

パラメータを SQL 文に渡す場合は上記例で言うと

```
$1::int8
```

と、連番で指定していきます。

これは PostgreSQL の例です。 MySQL の場合は ? で指定します。

指定できる型番は以下の形式です。

- text
- boolean
- int8 int4 int2
- float4 float8
- timestampz date timestamp
- json and jsonb
- uuid
- bytea
- numeric/decimal
- varchar

- regtype
- geo types
- array types: int8 int4 int2 float8 float4 bool text

6.2.12. データの更新

更新系のクエリは以下のような記述で行います。

データの投入は以下のように行います。

```
db.exec("insert into articles(title, content) values($1::text, $2::text)", params)
```

SQL 文に update や delete も設定ができます。

6.2.13. レンダリング

View の構造はテンプレート形式で設定することができます。

```
render "src/views/articles/show.ecr", "src/views/application.ecr"
```

の形式で ecr ファイルをレンダリングすることで、Rails のようにファイルを入れ子の形で描画することができます。

Web アプリ再起動後に <http://localhost:3000/> にアクセスすると、まだ記事が投稿されていないので空欄になっています。

<http://localhost:3000/articles/new> にアクセスすると、タイトルと本文を入力する画面が表示されており、入力後に一覧に遷移すると成功です。

記事タイトルをクリックすると詳細画面に遷移すると成功です。

6.2.14. オブジェクトの CRUD

Kemal は RESTful に対応しており、GET POST 以外にも、PUT DELETE にも対応しております。

記事の編集をサンプルに追加しながら説明します。

新規ページで <src/views/articles/edit.ecr> を追加します。

データの編集画面

```
<h2>投稿編集</h2>
<% articles.each do |article| %>
<form method="post", action="/articles/<%=article["id"] %>">
  <!--
    hiddenフィールドにname="_method"、
    value[put]を設定する。
  -->
  <input type="hidden", name="_method", value="put" />
  <input type="text" name="title" size="10" maxlength="10" value="<%=article["title"] %>" />
  <br />
  <br />
  <textarea name="content" cols="40" rows="4"><%=article["content"] %></textarea>
  <br />
  <br />
  <input type="submit" value="edit" class="btn btn-primary">
</form>
<% end %>
```

ブラウザによっては form が get post 以外には対応していない場合、以下の hidden フィールドの設定で PUT や DELETE で送信することが出来ます。

kemal-sample.cr を以下の内容で修正します。

```

get "/articles/:id/edit" do |env|
  articles = [] of Hash(String, String | Int32)
  article = {} of String => String | Int32
  id = env.params.url["id"].to_i32
  params = [] of Int32
  params << id
  sql = "select id, title, content from articles where id = $1::int8"
  article["id"], article["title"], article["content"] =
    db.query_one(sql, params, as: {Int32, String, String})
  articles << article
  db.close
  render "src/views/articles/edit.epr", "src/views/application.epr"
end

put "/articles/:id" do |env|
  id = env.params.url["id"].to_i32
  title_param = env.params.body["title"]
  content_param = env.params.body["content"]
  params = [] of String | Int32
  params << title_param
  params << content_param
  params << id
  db.exec("update articles set title = $1::text, content = $2::text where id = $3::int8", params)
  db.close
  env.redirect "/articles/#{id}"
end

```

これで、データの追加から一覧表示、編集までの処理が行えるようになります。
 また、DELETE を追加する場合は以下のように行います。

```

delete "/articles/:id" do |env|
  id = env.params.url["id"].to_i32
  params = [] of Int32
  params << id
  db.exec("delete from articles where id = $1::int8", params)
  db.close
  env.redirect "/"
end

```

これでデータの作成、表示、更新、削除の機能を持ったアプリを作ることができます。
 本稿では省略しますが、セッションを用いて認証機能を追加することも出来ます。

6.3. 補注

本稿のコードでは取り上げませんでしたが、Web アプリを書く上で注意すべき事項を以下に記載します。

6.3.1. トランザクション

データの原子性を保つ上で、トランザクション制御は必須です。
以下の設定でトランザクションのコミット、およびロールバックを設定します。

```
database_url = "postgres://localhost:5432/kemal_sample_development"
db = DB.open(database_url)
db.transaction do |tx|
  result = tx.connection.exec("insert into articles(title, content) values('hoge',
'huga')")
  # エラー時
  if !result
    tx.rollback
  else
    tx.commit
  end
end
```

6.3.2. MySQL

MySQL を使用する場合は、以下の設定を行います。
`shard.yml` に以下の記述を行います。

```
dependencies:
  mysql:
    github: crystal-lang/crystal-mysql
```

```
require "db"
require "mysql"

DB.open "mysql://root@localhost/test" do |db|
  # 実行系
  db.exec "insert into contacts values (?, ?)", "John", 30
end
```

7. CLI 開発

著者: at_grandpa

この章では Crystal での CLI 開発について書きます。

7.1. Crystal で CLI ツール

Crystal で CLI ツールの開発をしていきましょう。Crystal で CLI ツールを書くメリットは次のようなものがあります。

- コンパイルしてワンバイナリにできる
- Ruby 風の syntax で雑に書ける
- 実行速度が早い
- コンパイル時に型チェックが入る

CLI ツールは、欲しい時にサッと書いて継続的に使えるようにしたいですね。そういう意味では、Crystal という選択肢は良いのではないかでしょうか。今回初めて Crystal を触るという方も、まずは CLI ツールをサクッと作ってみることをおすすめします。では、早速作っていきましょう。

7.2. 自作の echo コマンド「myecho」を作る

echo コマンドを模倣した myecho コマンドを作っていきましょう。まずは `crystal init app myecho` を実行してひな形を作ります。

```
$ crystal init app myecho
  create  myecho/.gitignore
  create  myecho/.editorconfig
  create  myecho/LICENSE
  create  myecho/README.md
  create  myecho/.travis.yml
  create  myecho/shard.yml
  create  myecho/src/myecho.cr
  create  myecho/src/myecho/version.cr
  create  myecho/spec/spec_helper.cr
  create  myecho/spec/myecho_spec.cr
Initialized empty Git repository in /path/to/myecho/.git/
```

次のようなファイルが作成されました。

```
$ tree
.
├── LICENSE
├── README.md
├── shard.yml
└── spec
    ├── myecho_spec.cr
    └── spec_helper.cr
└── src
    └── myecho
        └── version.cr
            └── myecho.cr
```

3 directories, 7 files

これで準備は整いました。まずはテストを回してみましょう。

```
$ crystal spec
F

Failures:

1) MyEcho works
Failure/Error: false.should eq(true)

Expected: true
got: false

# spec/myecho_spec.cr:7

Finished in 73 microseconds
1 examples, 1 failures, 0 errors, 0 pending

Failed examples:

crystal spec spec/myecho_spec.cr:6 # MyEcho works
```

テストは落ちます。ひな形生成時、`spec/myecho_spec.cr` に失敗するテストが書かれているからです。

spec/myecho_spec.cr

```
require "./spec_helper"
require "../../src/myecho.cr"

describe MyEcho do
  # TODO: Write tests

  it "works" do
    false.should eq(true)
  end
end
```

では、実際のテストから書いていきましょう。`myecho` の基本機能は「与えられた引数の文字列をそのまま出力する」です。

spec/myecho_spec.cr の一部

```
describe MyEcho do
  describe MyEcho::Cli do
    describe "run" do
      it "writes the content of args to specified IO" do
        io = IO::Memory.new          ①
        myecho = MyEcho::Cli.new(io)  ②
        myecho.run(["foo", "bar"])   ③
        io.to_s.should eq "foo bar\n" ④
      end
    end
  end
end
```

- ① 出力する先の `IO` インスタンスを生成します。テスト時は `IO::Memory` に出力します。
- ② `MyEcho::Cli` に `io` を渡し、インスタンスを生成します。
- ③ `MyEcho::Cli#run` に、コマンドライン引数の `ARGV` を模した `["foo", "bar"]` を渡して実行します。
- ④ `io` に出力された文字列を検証します。

まだ実装が終わっていないので、このテストは落ちます。実装側も書きましょう。

src/myecho.cr

```
require "./myecho/*"

module MyEcho
  class Cli
    def initialize(@io : IO = STDOUT) ①
    end

    def run(args)                      ②
      @io.print args.join(" ") + "\n" ③
    end
  end
end
```

① インスタンス変数 `@io` を定義します。初期値は `STDOUT` です。

② `#run` を定義します。

③ `@io` に引数 `args` を出力します。

書けたらテストを回しましょう。

```
$ crystal spec
.

Finished in 66 microseconds
1 examples, 0 failures, 0 errors, 0 pending
```

通りました。これで、受け取った引数をそのまま出力するメソッド `#run` を実装できました。次は build してバイナリを作りましょう。

まずは build 対象のファイルを作ります。`src/myecho.cr` で `#run` を呼び出して直接 build してもよいですが、そうするとテスト時に `#run` が実行されてしまいます。それを避けるために `cli.cr` ファイルを別途作成し、`myecho.cr` を require しましょう。

src/cli.cr

```
require "./myecho"

MyEcho::Cli.new.run(ARGV)
```

これで、モジュールと build ファイルを分離できました。早速 build してみましょう。

```
$ mkdir bin
$ crystal build -o ./bin/myecho ./src/cli.cr
```

`./bin/` ディレクトリを作成し、その中に `myecho` という名前でバイナリを出力しています。`myecho` を実行してみましょう。

```
$ ./bin/myecho Hello!! World!!
Hello!! World!!
```

出力されました！CLI ツールの完成です！いろいろ出力して遊んでみてください。

```
$ ./bin/myecho HAHAHA!
HAHAHA!
```

7.3. バージョン表示のオプション `-v` を実装する

さらに CLI ツールらしくしていきましょう。バージョンを表示させる `-v` オプションを実装します。オプションがあると一気に CLI ツールらしくなりますね。Crystal には `OptionParser` というクラスが用意されています。コマンドラインオプションを扱うのに便利なクラスです。今回は `OptionParser` の使い方も解説しつつ実装していきます。まずはテストから書きましょう。

`spec/myecho_spec.cr` の一部

```
describe "writes the version to specified IO" do
  it "with '-v'" do
    io = IO::Memory.new                      ①
    myecho = MyEcho::Cli.new(io)              ②
    myecho.run(["-v"])                       ③
    io.to_s.should eq MyEcho::VERSION + "\n" ④
  end
end
```

① 出力する先の `IO` インスタンスを生成します。テスト時は `IO::Memory` に出力します。

② `MyEcho::Cli` に `io` を渡し、インスタンスを生成します。

③ `version` を表示するコマンドライン引数 `["-v"]` を渡して実行します。

④ `MyEcho::VERSION` が表示されていることを検証します。

テストを回しましょう。

```
$ crystal spec  
..F  
  
Failures:  
  
1) MyEcho MyEcho::Cli run writes the version to specified IO with '-v'  
Failure/Error: io.to_s.should eq MyEcho::VERSION + "\n"  
  
Expected: "0.1.0\n"  
got: "-v\n"  
  
# spec/myecho_spec.cr:18  
  
Finished in 117 microseconds  
3 examples, 1 failures, 0 errors, 0 pending  
  
Failed examples:  
  
crystal spec spec/myecho_spec.cr:14 # MyEcho MyEcho::Cli run writes the version to  
specified IO with '-v'
```

0.1.0 が期待されていますが -v が出力されていますね。これは期待通りの落ち方です。では実装に入りましょう。単純に「 -v が入力されたら MyEcho::VERSION を出力する」でもよいのですが、先程も宣言した通り OptionParser を導入します。

```

require "./myecho/*"
require "option_parser" ①

module MyEcho
  class Cli
    def initialize(@io : IO = STDOUT)
    end

    class Options
      property display_version : Bool = false
      property args : Array(String) = [] of String
    end

    def run(args)
      # オプション設定を格納する
      options = Options.new

      # OptionParserを用いたオプションの設定
      OptionParser.parse(args) do |parser| ②
        parser.on("-v", "show version") do ③
          options.display_version = true
        end
        parser.unknown_args do |unknown_args| ④
          options.args = unknown_args
        end
      end
    end

    # バージョンの表示
    if options.display_version
      @io.print MyEcho::VERSION + "\n"
      return
    end

    # 引数の表示
    @io.print options.args.join(" ") + "\n"
  end
end

```

① 標準ライブラリの `OptionParser` を `require` します。

② `OptionParser#parse` に `args` を渡し、ブロック内でオプションを定義していきます。

③ `OptionParser#on` の引数に `-v` とその説明文を、ブロックには実行したい処理を書きます。

④ オプション以外の引数は、配列になって `OptionParser#unknown_args` のブロック引数となります。

テストを回しましょう。

```
$ crystal spec
...
Finished in 102 microseconds
3 examples, 0 failures, 0 errors, 0 pending
```

通りました。バイナリも作りましょう。そして、実際にバージョンを表示してみましょう。

```
$ crystal build -o ./bin/myecho ./src/cli.cr
$ ./bin/myecho foo
foo
$ ./bin/myecho -v
0.1.0
```

バージョン表示ができました！CLI ツールらしくなってきました。ここまで来ると、`--version` を指定してもバージョンを表示させたいですよね。現状だと例外が発生してしまいます。

```
$ ./bin/myecho --version
--version
Invalid option: --version (OptionParser::InvalidOption)

(スタックトレースが続く ... )
```

では対応していきましょう。まずはテストから書きます。

`spec/myecho_spec.cr` の一部

```
describe "writes the version to specified IO" do
  it "with '-v'" do
    io = IO::Memory.new
    myecho = MyEcho::Cli.new(io)
    myecho.run(["-v"])
    io.to_s.should eq MyEcho::VERSION + "\n"
  end

  # 新しいテストケース
  it "with '--version'" do
    io = IO::Memory.new
    myecho = MyEcho::Cli.new(io)
    myecho.run(["--version"]) ①
    io.to_s.should eq MyEcho::VERSION + "\n"
  end
end
```

① `--version` を指定した場合でも、バージョンが表示されることを検証しています。

テストを回すと、例外が発生しテストが落ちます。

```
$ crystal spec
....E

Failures:

1) MyEcho MyEcho::Cli run writes the version to specified IO with '--version'

    Invalid option: --version
Error running at_exit handler: Index out of bounds
```

対応するには、`OptionParser#on` の第二引数に `long_flag` を指定します。

`src/myecho.cr` の一部

```
parser.on("-v", "--version", "show version") do ①
  options.display_version = true
end
```

① `OptionParser#on` の第二引数に `--version` を指定しています。

これでテストが通ります。

```
$ crystal spec
.....
Finished in 189 microseconds
6 examples, 0 failures, 0 errors, 0 pending
```

実際にバージョンを表示してみましょう。

```
$ crystal build -o ./bin/myecho ./src/cli.cr
$ ./bin/myecho foo
foo
$ ./bin/myecho -v
0.1.0
$ ./bin/myecho --version
0.1.0
```

`--version` でもバージョンを表示することができました。このように、`OptionParser` は、コマンドラインオプションを柔軟に扱えます。

7.4. ヘルプ表示のオプション `-h --help` を実装する

次にヘルプを表示してみます。オプションの追加はバージョン表示の時と同じです。まずはテストから書きましょう。

spec/myecho_spec.cr の一部

```
describe "writes the help to specified IO" do
  it "with '-h'" do
    io = IO::Memory.new
    myecho = MyEcho::Cli.new(io)
    myecho.run(["-h"])
    io.to_s.should eq "helpには何が表示される？"
  end
end
```

-h を指定した場合、何が表示されるかはまだわかりません。とりあえずこのまま進みましょう。現状だと、 Invalid option: -h でテストが落ちることは目に見えています。まずは適当に文字列を返しましょう。

src/myecho.cr の一部

```
parser.on("-h", "--help", "show help") do
  options.display_help = true
end

...

# ヘルプの表示
if options.display_help
  @io.print "helpです。" + "\n"
  return
end
```

当然、テストは失敗します。

```

$ crystal spec
.....
Failures:

1) MyEcho MyEcho::Cli run writes the help to specified IO with '-h'
Failure/Error: io.to_s.should eq "helpには何が表示される?"

  Expected: "helpには何が表示される?"
  got: "helpです。\\n"

# spec/myecho_spec.cr:35

Finished in 246 microseconds
10 examples, 1 failures, 0 errors, 0 pending

Failed examples:

crystal spec spec/myecho_spec.cr:31 # MyEcho MyEcho::Cli run writes the help to
specified IO with '-h'

```

ここで、`OptionParser` の便利機能を使います。`OptionParser#to_s` で、ヘルプメッセージを返してくれます。実装してみましょう。

`src/myecho.cr` の一部

```

# OptionParserを用いたオプションの設定
OptionParser.parse(args) do |parser|
  ...
  # ヘルプメッセージの格納
  # すべての設定を行った後で格納する
  options.help_message = parser.to_s + "\\n"
end
...
# ヘルプの表示
if options.display_help
  @io.print options.help_message
  return
end

```

テストを回します。

```

$ crystal spec
...F

Failures:

1) MyEcho MyEcho::Cli run writes the help to specified IO with '-h'
Failure/Error: io.to_s.should eq "helpには何が表示される?"

  Expected: "helpには何が表示される?"
  got: "      -h, --help                                show help\n      -v, --version
show version\n"

# spec/myecho_spec.cr:35

Finished in 174 microseconds
4 examples, 1 failures, 0 errors, 0 pending

Failed examples:

crystal spec spec/myecho_spec.cr:31 # MyEcho MyEcho::Cli run writes the help to
specified IO with '-h'

```

それらしい文字列が返ってきました。テストを修正しましょう。

spec/myecho_spec.cr の一部

```

describe "writes the help to specified IO" do
  it "with '-h'" do
    io = IO::Memory.new
    myecho = MyEcho::Cli.new(io)
    myecho.run(["-h"])
    io.to_s.should eq <<-HELP_MESSAGE
      -h, --help                                show help
      -v, --version                            show version

    HELP_MESSAGE
  end
end

```

今度はテストが回りました。

```

$ crystal spec
.....
Finished in 294 microseconds
10 examples, 0 failures, 0 errors, 0 pending

```

ヘルプが表示されるかを build して確かめます。

```
$ crystal build -o ./bin/myecho ./src/cli.cr
$ ./bin/myecho foo
foo
$ ./bin/myecho -h
-h, --help                                show help
-v, --version                               show version
$ ./bin/myecho --help
-h, --help                                show help
-v, --version                               show version
```

ヘルプが表示されました。しかし、もう少し体裁の整ったヘルプがよいですね。例えば、コマンドの説明や Usage などがあると良さそうです。テストを書きましょう。

spec/myecho_spec.cr の一部

```
describe "writes the help to specified IO" do
  it "with '-h'" do
    io = IO::Memory.new
    myecho = MyEcho::Cli.new(io)
    myecho.run(["-h"])
    io.to_s.should eq <<-HELP_MESSAGE
      My echo.

      Usage: myecho [options] [arguments]

      -h, --help                                show help
      -v, --version                             show version

      HELP_MESSAGE
    end
  end
```

良い感じのヘルプにしてみました。テストが通るように実装しましょう。`OptionParser` には、`#banner=` というメソッドがあり、ヘルプメッセージに加える文字列を定義できます。

src/myecho.cr の一部

```
parser.banner = <<-BANNER
My echo.

Usage: myecho [options] [arguments]

BANNER
```

これでテストを通すことができました。build して動作を確かめましょう。

```
$ crystal build -o ./bin/myecho ./src/cli.cr
$ ./bin/myecho --help

My echo.

Usage: myecho [options] [arguments]

-h, --help           show help
-v, --version        show version
```

より、ヘルプらしくなりました。`OptionParser` を使えば、このように簡単にヘルプを設定できます。

7.5. `prefix` を付ける `--prefix` を実装する

もっと CLI ツールらしくするために、さらにオプションを加えましょう。各コマンド引数に `prefix` を付ける `--prefix` オプションです。期待される動作は次のようになります。

```
$ ./bin/myecho --prefix pre_ foo bar baz
pre_foo pre_bar pre_baz
```

`--prefix PREFIX` を指定することで、各引数の先頭に `PREFIX` を付けます。まずはテストから書きましょう。

`spec/myecho_spec.cr` の一部

```
describe "prefix specified string to each arguments" do
  it "with '--prefix PREFIX'" do
    io = IO::Memory.new
    myecho = MyEcho::Cli.new(io)
    myecho.run(["--prefix", "pre_", "foo", "bar", "baz"])
    io.to_s.should eq "pre_foo pre_bar pre_baz\n"
  end
end
```

このテストを通すように実装しましょう。

src/myecho.cr の一部

```
# OptionParserを用いたオプションの設定
OptionParser.parse(args) do |parser|
  ...
  parser.on("--prefix PREFIX", "prefix to each arguments") do |prefix|
    options.prefix = prefix
  end
  ...
end
...
prefix = options.prefix
unless prefix.nil?
  @io.print options.args.map { |arg| prefix + arg }.join(" ") + "\n"
  return
end
```

OptionParser#on は、`long_flag` の名前だけでも指定可能です。`--prefix PREFIX` のように、オプション引数（PREFIX）を書くと、オプション引数が必須になります。また、今回の `--prefix` 定義のままで、コマンドラインから `--prefix=PREFIX` のように `=` を用いた指定も可能です。いい感じに取り扱ってくれます。

これで `--prefix` の実装も終わりました。build して動作を確認しましょう。

```
$ crystal build -o ./bin/myecho ./src/cli.cr
$ ./bin/myecho --prefix pre_ foo bar baz
pre_foo pre_bar pre_baz
```

うまく動作しているようです。

いかがでしたでしょうか。OptionParser を使っての CLI ツールの作成方法が大体つかめたでしょうか。

以下に、ここまで紹介した `myecho` のコード全てを記載します。

src/cli.cr

```
require "./myecho"

MyEcho::Cli.new.run(ARGV)
```

src/myecho.cr

```

require "./myecho/*"
require "option_parser"

module MyEcho
  class Cli
    def initialize(@io : IO = STDOUT)
    end

    class Options
      property display_version : Bool = false
      property args : Array(String) = [] of String
      property display_help : Bool = false
      property help_message : String = ""
      property prefix : String? = nil
    end

    def run(args)
      # オプション設定を格納する
      options = Options.new

      # OptionParserを用いたオプションの設定
      OptionParser.parse(args) do |parser|
        parser.banner = <<-BANNER
          My echo.

          Usage: myecho [options] [arguments]

          BANNER
        parser.on("--prefix PREFIX", "prefix to each arguments") do |prefix|
          options.prefix = prefix
        end
        parser.on("-h", "--help", "show help") do
          options.display_help = true
        end
        parser.on("-v", "--version", "show version") do
          options.display_version = true
        end
        parser.unknown_args do |unknown_args|
          options.args = unknown_args
        end
        # ヘルプメッセージの格納
        options.help_message = parser.to_s + "\n"
      end

      # ヘルプの表示
      if options.display_help
        @io.print options.help_message
        return
      end
    end
  end
end

```

```

# バージョンの表示
if options.display_version
  @io.print MyEcho::VERSION + "\n"
  return
end

# prefixをつける
prefix = options.prefix
unless prefix.nil?
  @io.print options.args.map { |arg| prefix + arg }.join(" ") + "\n"
  return
end

# 引数の表示
@io.print options.args.join(" ") + "\n"
end
end

```

src/myecho_spec.cr

```

require "./spec_helper"
require "../../src/myecho.cr"

describe MyEcho do
  describe MyEcho::Cli do
    describe "run" do
      it "writes the content of args to specified IO with args" do
        io = IO::Memory.new
        myecho = MyEcho::Cli.new(io)
        myecho.run(["foo", "bar"])
        io.to_s.should eq "foo bar\n"
      end
      describe "writes the version to specified IO" do
        it "with '-v'" do
          io = IO::Memory.new
          myecho = MyEcho::Cli.new(io)
          myecho.run(["-v"])
          io.to_s.should eq MyEcho::VERSION + "\n"
        end
        it "with '--version'" do
          io = IO::Memory.new
          myecho = MyEcho::Cli.new(io)
          myecho.run(["--version"]) ①
          io.to_s.should eq MyEcho::VERSION + "\n"
        end
      end
    end
    describe "writes the help to specified IO" do
      it "with '-h'" do
        io = IO::Memory.new
        myecho = MyEcho::Cli.new(io)
      end
    end
  end
end

```

```

myecho.run(["-h"])
io.to_s.should eq <<-HELP_MESSAGE

My echo.

Usage: myecho [options] [arguments]

--prefix PREFIX          prefix to each arguments
-h, --help                show help
-v, --version             show version

HELP_MESSAGE
end
end
describe "prefix specified string to each arguments" do
  it "with '--prefix PREFIX'" do
    io = IO::Memory.new
    myecho = MyEcho::Cli.new(io)
    myecho.run(["--prefix", "pre_", "foo", "bar", "baz"])
    io.to_s.should eq "pre_foo pre_bar pre_baz\n"
  end
end
end
end

```

次は、もう少しかゆいところに手が届く、サードパーティ製の CLI ビルダーライブラリをいくつかご紹介します。

7.6. サードパーティ製のライブラリを使う

CLI ツールを作成するにあたって、サードパーティ製の CLI ビルダーツールを使うことは得策です。OptionParser よりもリッチな機能を使うことができます。例えば、Ruby だと thor などが有名です。Crystal でも CLI ビルダーツールが作成されています。今回は主観で選択したいいくつかをご紹介します。題材としては、ここまで作ってきた myecho を用います。

7.6.1. mrrooijen/commander

早い時期から作成されていたライブラリです。百聞は一見に如かず、早速 myecho を実装しましょう。

```

require "./myecho_commander/*"
require "commander"

cli = Commander::Command.new do |cmd|
  cmd.use = "myecho"
  cmd.long = "My echo."

  cmd.flags.add do |flag|
    flag.name = "prefix"
    flag.long = "--prefix"
    flag.default = ""
    flag.description = "prefix to each arguments."
  end

  cmd.flags.add do |flag|
    flag.name = "version"
    flag.long = "--version"
    flag.short = "-v"
    flag.default = false
    flag.description = "show version."
  end

  cmd.run do |options, arguments|
    # バージョンの表示
    if options.bool["version"]
      puts MyechoCommander::VERSION
      next
    end

    # prefixをつける
    prefix = options.string["prefix"]
    unless prefix.empty?
      puts arguments.map { |arg| prefix + arg }.join(" ")
      next
    end

    # 引数の表示
    puts arguments.join(" ")
  end
end

Commander.run(cli, ARGV)

```

サードパーティ製のライブラリは、基本的に次の構成になっています。

- コマンドの説明（Description や Usage など）
- Options や Flags の定義

- 実際に実行される箇所

- `run` メソッドや `run` ブロックなど
- ここで `Options` や `Flags` の入力値を受け取れる
- ここで 入力された引数を受け取れる

これらを独自の定義方法で書いていきます。`OptionParser` を用いたときよりも見やすくなっていると思います。また、`help` や `version` など、CLI ツールに必須のオプションはデフォルトでついている場合もあります。`mrooijen/commander` の場合は `help` がデフォルトで設定されているので、コード上には現れていません。また、サブコマンドも実装可能です。詳しくは公式マニュアルを御覧ください。コードを実際に実行すると次のようになります。

```
$ crystal run src/myecho_commander.cr -- foo bar baz
foo bar baz
$ crystal run src/myecho_commander.cr -- --version
0.1.0
$ crystal run src/myecho_commander.cr -- --help
myecho - My echo.

Usage:
  myecho [flags] [arguments]

Commands:
  help [command] # Help about any command.

Flags:
  -h, --help      # Help for this command. default: 'false'.
  --prefix       # prefix to each arguments. default: ''.
  -v, --version   # show version. default: 'false'.
$ crystal run src/myecho_commander.cr -- --prefix pre_ foo bar baz
pre_foo pre_bar pre_baz
```

期待通りの動作をしていることがわかります。

7.6.2. jwaldrip/admiral.cr

最近も開発されているライブラリです。早速コードを見てみましょう。

```

require "./myecho_admiral/*"
require "admiral"

module MyechoAdmiral
  class Cli < Admiral::Command
    define_version MyechoAdmiral::VERSION, short: v
    define_help description: "My echo.", short: h

    define_flag prefix : String?,
      description: "prefix to each arguments.",
      long: prefix

    def run
      # prefixをつける
      prefix = flags.prefix
      unless prefix.nil?
        puts arguments.map { |arg| prefix + arg }.join(" ")
        return
      end

      # 引数の表示
      puts arguments.join(" ")
    end
  end
end

MyechoAdmiral::Cli.run

```

行数が少ないですね。`help` と `version` はマクロで1行で書かれています。こういった DSL を提供しやすいのもマクロの強みです。構造もわかりやすく、読みやすいと思います。もちろん、サブコマンドも対応しています。

実際に実行すると次のようにになります。

```
$ crystal run src/myecho_admiral.cr -- foo bar baz
foo bar baz
$ crystal run src/myecho_admiral.cr -- --version
0.1.0
$ crystal run src/myecho_admiral.cr -- --help
Usage:
myecho [flags...] [arg...]

My echo.

Flags:
--help, -h (default: false)      # Displays help for the current command.
--prefix                         # prefix to each arguments.
--version, -v (default: false)
$ crystal run src/myecho_admiral.cr -- --prefix pre_ foo bar baz
pre_foo pre_bar pre_baz
```

こちらも期待通りの動作をしています。

7.6.3. at-grandpa/clim

これは私自身が作成したライブラリです。記述量の少なさと直感的な記述を目的に作成しています。コードを見てみましょう。

```

require "./myecho_clim/*"
require "clim"

module MyechoClim
  class Cli < Clim
    main_command do
      desc "My echo."
      usage "myecho [options] [arguments] ..."
      option "--prefix PREFIX", type: String, desc: "prefix to each arguments.",
      default: ""
      version MyechoClim::VERSION, short: "-v"
      run do |options, arguments|
        # prefixをつける
        prefix = options.prefix
        unless prefix.empty?
          puts arguments.map { |argument| prefix + argument }.join(" ")
          return
        end

        # 引数の表示
        puts arguments.join(" ")
      end
    end
  end
end

MyechoClim::Cli.start(ARGV)

```

`desc` や `option` などの定義が1行で書けます。`version` マクロも用意しました。コマンドの実行箇所は、`run` ブロックになります。サブコマンドも対応しており、直感的に記述することができます。詳しくはマニュアルを御覧ください。

実際に実行すると次のようにになります。

```
$ crystal run src/myecho_clim.cr -- foo bar baz
foo bar baz
$ crystal run src/myecho_clim.cr -- --version
0.1.0
$ crystal run src/myecho_clim.cr -- --help
```

My echo.

Usage:

```
myecho [options] [arguments] ...
```

Options:

| | |
|---------------------------------|---|
| --prefix PREFIX [default:""] | prefix to each arguments. [type:String] |
| --help | Show this help. |
| -v, --version | Show version. |

```
$ crystal run src/myecho_clim.cr -- --prefix pre_ foo bar baz
pre_foo pre_bar pre_baz
```

こちらも期待通りの動作をしています。

サードパーティ製のライブラリはそれぞれに特徴がありますが、デフォルトの `OptionParser` よりはリッチな機能を提供してくれます。個人で使う小さな CLI ツールだと `OptionParser` で十分かもしれません。しかし、ツールとして公開したりメンテナンスが必要になってくるケースでは、サードパーティ製ライブラリを使うほうが良いかと思います。ぜひ一度試してみてください。

7.7. まとめ

この章では Crystal で CLI ツールの開発を行うことについて説明しました。冒頭に説明した通り、利点は次のようになるでしょう。

- コンパイルしてワンバイナリにできる
- Ruby 風の syntax で雑に書ける
- 実行速度が早い
- コンパイル時に型チェックが入る

ものすごく雑に小さい CLI ツールを作成する場合は、コンパイルがある分、Crystal よりも Ruby の方が速いでしょう。しかし、中規模程度でコード量が多くてメンテナンスが必要になってくるケースだと、Crystal の利点は大きくなってくるのではないかでしょうか。大規模なバッチ処理などでも、処理スピードやメモリ使用量などで Crystal の力が発揮されると思います。

みなさんも一度、敷居の低い CLI ツールを通して Crystal に触れてみてください。

8. 著者紹介

8.1. 5t111111

GitHub のアカウント

[@5t111111](#)

Twitter のアカウント

[@5t111111](#)

自己紹介

Crystal-JP の第一回イベントに参加した流れでドキュメントの日本語化を担当したことからずっと Crystal に関わっています。現在日本語ドキュメントの更新は停止していますが、バージョン 1.0.0 がリリースされた暁にはどんな形であれ再開したいです。

8.2. AKJ

GitHub のアカウント

[@arcage](#)

Twitter のアカウント

[@arcage](#)

自己紹介

地方でシステム業の傍ら、趣味でコードを書いてます。常々型に厳格な Ruby が欲しいと思っていたところに Crystal と出会い、以来 Crystal にどっぷりとハマっています。

8.3. at_grandpa

GitHub のアカウント

[@at-grandpa](#)

Twitter のアカウント

[@at_grandpa](#)

自己紹介

Crystal と出会い、衝撃を受けて好きになり、コードを書いたり活動をしていたところ、このような執筆の機会をいただきました。ありがとうございます。Crystal 言語はとてもおもしろいのでぜひ触れてみてください。Happy crystalling !!

8.4. MakeNowJust

GitHub のアカウント

[@MakeNowJust](#)

Twitter のアカウント

[@make_now_just](#)

自己紹介

仕事で TypeScript を書く傍らで、趣味で Crystal のコンパイラやフォーマッタにコントリビュートしています。Crystal のコンパイラは Crystal 自身で書かれていて、他の C 言語などで書かれたコンパイラよりはコントリビュートの障壁は低いかと思います。Crystal に興味を持ったらコンパイラや標準ライブラリのコードを読んでみると良いかもしれません。

8.5. msky

GitHub のアカウント

[@msky026](#)

Twitter のアカウント

[@msky026](#)

自己紹介

本業はスマホゲーの人。Ruby や Unity など業務や趣味で使ってたりします。Crystal はビルドできる Ruby があったらいいなと思ってた時に存在を知り、そのタイミングで第2回 Crystal 勉強会が開催されてたのでそのままどっぷりと。