Ruby っぽいコード入門 (仮称)

XXXX 編　著

1

2019-09-22 版　　XXX 発行

前書き

「Ruby っぽい」という言葉に聞き覚えはありませんか？
特に Ruby を始めて間もない方、レビューを受けたり勉強会に行ったりした際に

```
「こうした方が Ruby っぽい」「この書き方は Ruby っぽくない」
「こうすると Ruby らしい」「Ruby ではこういう書き方が好まれる」
```

そんな話を聞いたことはありませんでしょうか？

色々な書き方が出来ることが Ruby の大きな特徴ではありますが、
ある程度のあるべき姿＝「Ruby っぽい」は存在しているようなのです。

「っぽい」ってなんだろう？ なぜそれが「っぽい」のだろうか？
どうすれば「っぽい」になるのだろうか？
Ruby を始めて間もない時期の私はいつもそう感じていました。

が、そんな私も Ruby で実務を約半年経験して、目にしたコードが「Ruby っぽい」かどうか感じ取れるようになってきました。この経験を元に効率良く「Ruby っぽい」コードを書く方法をまとめたら、Ruby に入門したての人に役立つのではないか？ あととにかく技術書典で本を出してみたい！ そんなことを考えて書いたのがこの本です。

「Ruby っぽい」コードが書けるようになると最終的にはプログラミングが楽しくなる！と個人的には思っています。本書によって「Ruby っぽい」コードが書けるようになり、更にプログラムする事自体が楽しくなる、そんな未来の一助に慣れれば幸いです。

またもし Ruby に詳しい方で、この本の内容に疑問や不足を感じるようなことがありま

2

したら是非ご意見ください。(どうかその際はお手柔らかに……)

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

3

目次

4

6

第 1 章

はじめに

1.1　この本のねらい

本書の目的は、効率良く「Ruby っぽい」コードを書くための情報を提供する事です。「効率良く」は以下を含みます。

- (比較的) 簡単に適用できる
- 利用頻度が高い
- 再現性がある

1.2　想定読者

本書の想定読者は以下の通りです。

- Ruby の基本文法等は理解して、ちょっとしたプログラムなら書ける方。
- Ruby の開発者を目指して勉強中の方。
- 他言語経験があり、最近 Ruby を触り始めた方。

一言でいうと、Ruby 入門者はひとまず超えて初級者に入った位の方です。以降本書では対象読者を"Ruby 初級者"と表現します。

1.3　本書で扱わないこと

本書では Ruby on Rails 関連の知識は扱いません。あくまで Ruby にフォーカスした内容となります。ただ Ruby on Rails の環境で Ruby のコードを書く、という状況を想定することはあります。

1.4 本書における「Ruby っぽい」の定義

本書では以下を 2 点を満たしたコードを「Ruby っぽい」コードとします。

1. Ruby 開発者にとって馴染みのある表現
2. Ruby の持つ多様な機能を使いこなしている

これらを十分に実現するのはとても大変で、特に Ruby 初級者には現実的ではありません。なので本書では成果が得られやすい (= 効率が良い) 範囲で、上記 2 点を達成する方法を紹介します。

1.5 「Ruby っぽい」コードを書くメリット

前書きで少し触れましたが、「Ruby っぽい」コードを書けるようになると最終的にはプログラミング自体が楽しくなります。ただそこに行き着くまでにも様々なメリットがあります。それらをまとめてここで紹介します。(Ruby 初級者目線の話になります)

1.5.1 読む側が見やすいコードを書けるようになる

良く言われることではありますが、コードを書く際には可読性を意識するのがとても重要です。自分が書いたコードというのは、プロジェクトが続く限り何らかの形でメンテされ続けます。チーム開発であれば、書いた自分がコードに触れている時間より、他の人が読む時間が長くなります。可読性が高いコードが書けるに越したことはありません。

「Ruby っぽい」コードは前述の通り、他の Ruby 開発者から見て馴染みのある＝読みやすいコードになります。可読性という観点で有効であると言えるでしょう。

1.5.2 周囲の評価が上がりメンタルを守れる (かも)

上の話と関連しますがあなたにも直接的なメリットがあります。読みやすいコードを書く人は、恐らく周囲から良い評価を受けやすくなるはずです。特にコードのレビューをしてくれる人達 (会社の上司や先輩、コミュニティーやスクールのメンター等) からは良い印象を持たれるでしょう。プルリクエストの指摘等も少しずつ減っていくはずです。

そしてこれは精神的な負荷の軽減に繋がります。実務に入りたての駆け出しエンジニアは、慣れない環境や技術力の不足によってメンタルを削られるイベントに遭遇しやすいものです。余りにも多くの指摘を受けて、お互い悪気はないのに雰囲気が悪くなる、そういった事は割と頻繁に起こり得ます。「Ruby っぽい」コードを書ければ、コードに関して

上司や先輩から細かい指摘を受けにくくなり、PR のやりとりも細かいものは減ってくるでしょう。メンタルのケアは本当に重要です。そんな観点でも「Ruby っぽく」書くことは役に立つはずです。

1.5.3 他の Ruby 開発者が書くコードを読めるようになる

「Ruby っぽい」コードが書ければ、読むこともできるはずです。

おそらく Ruby 初級者が読むコードは、自分より経験のある Ruby 開発者が書いたものである事が多いのではないでしょうか？ (会社の上司先輩が書いたコードや、OSS として公開されているものなど) そういったコードは、Ruby で良く使われる表現が多く出てきたり、Ruby の様々な機能を使いこなしたものである事が多く、雑に言えば「Ruby っぽい」コードである可能性が高いでしょう。あなたが「Ruby っぽい」コードを書けるようになれば、そういった他の開発者のコードにおいてをみて、パッと理解できることが増えるでしょう。

1.5.4 プログラミング自体が楽しくなってくる

完全に主観ですが「Ruby っぽい」コードが書けるようになればプログラミング自体が楽しくなってきます。

プログラミング中には、目的とは直接関係無いが必要となるちょっとした処理、というのが割と出現します。自分で作り込まないといけない場合は少し大変です。ライブラリで解決できることも多いですが、それはそれで面倒が増えたりもします。

Ruby はそういった「ちょっとした処理」が標準でたくさん入っています。それらを上手く利用できれば、本来やるべきことに集中でき、心地よく楽しくプログラミングができるようになります。また Ruby 自体がコードを簡潔に書くことを信条としている言語なので、「ちょっとした処理」を利用することも含めて「Ruby っぽい」に近づければコードはシンプルで見た目が良いものになります。

つまり「Ruby っぽい」コードを書けば、本来自分が実現したいことに集中でき、かつコードもシンプルで見た目の良いものになるわけです。その時あなたはこう思うでしょう。「こんな面倒な処理をシンプルに見栄えよくかけちゃう自分って凄くない？」こうなればプログラミングが楽しくないはずがありません。(まぁ実際にはすごいのは Ruby ですけれども、それを使いこなしてるあなたもすごいんです。そう思っておきましょう)

第 2 章

Rubocop でコードの見た目を Ruby っぽくする

「Ruby っぽい」コードを書くために最初に提案するのは Rubocop の利用です。

Rubocop は Ruby で最もよく利用されている、静的コード解析を実行する gem(*1) です。この空白はいらないよとか、この場合はこっちのメソッドを使った方がいいよ、といったことをアドバイスしてくれます。コードの可読性向上をサポート上げてくれるとても頼もしいやつです。

Rubocop は Ruby 開発者であれば知らない人はほとんどいない、といっても過言ではない gem です。従って Rubocop に沿ったコードは多くの Ruby 開発者にとって見慣れたもの、もしくは見慣れたものであるべきと思われているコードであり、本書の目的である「Ruby っぽい」コードを目指すにはもってこいなわけです。

まずは Rubocop のアドバイスに従うことで、「Ruby っぽい」コードにしていきましょう。

ここからは Rubocop の導入に関して段階的に説明していきます。具体的には以下の流れです。必要な箇所から読んでいっていただければ幸いです。

- Rubocop 導入前の注意点
- Rubocop のインストール
- Rubocop の実施
- Rubocop 指摘の修正方法
- Rubocop のカスタマイズ
- 既存プロジェクトへ Rubocop を適用する際の注意事項
- Rubocop を CI ツール等と連携する

2.1 Rubocop 導入前に意識しておきたい事

Rubocop 導入前に以下 2 つほど意識しておきたい事があります。

2.1.1 Rubocop の本来の目的

本書では Rubocop を、自分が書いたコードを「Ruby」っぽくする目的で利用しますが、これは一般的な Rubocop の用途からは少しずれています。

Rubocop は基本的にチーム開発においてコーディングスタイルの統一を目的として利用されます。

コーディングスタイルとはプログラムの書き方に関する約束事です。簡単な例で言えば、空白の数や改行、変数の名前、適切な文法の使い方などが挙げられます。仕事として行うプログラミングはチームで行うことがほとんどです。チームのメンバ間で可能な範囲で書き方を統一し、コードの可読性を高めて生産性を上げよう、というのがコーディングスタイルの目的になります。

Rubocop はコーディングスタイルを統一するためのツールです。

もちろん個人のコードをより良くするために Rubocop を利用して問題はありませんが、

1
1

2.1.2 チームの方針を優先すること

前節の話とかぶる部分もありますが、コーディングスタイルはチーム毎にに関して優先する内容があるなら、„, (以下要作成)

2.2 Rubocop のインストール

Rubocop の導入は以下の 2 パターンになると思います。1. Gemfile に記載する (bundler を利用) 2. gem install コマンドを用いる

基本的には Gemfile への記載するやり方を取るのが良いと思います。ただチーム開発の環境であれば、勝手に Gem を追加するわけにいかない事も多いと思いますので、自分だけで Rubocop を使いたいような場合には、gem install を用いるのも良いでしょう。

なお両方やっていただいても問題ありません。

2.2.1 Gemfile に記載 (bundler を利用)

Gemfile に記載するに以下のように記載した上で、bundle install を実行するだけです。

```
gem 'rubocop', '~> 0.70.0', require: false
```

Rubocop はバージョンによってチェックする内容が変わることが多々あります。bundle install によって意図せず Rubocop のバージョンが変わってしまうことを避けるため、バージョンは指定しておいた方が良いです。

Rails プロジェクトの Gemfile に記載する場合は、development 以外で使うことはないと思いますので、

```
group 'development' do
  gem 'rubocop', '~> 0.70.0', require: false
end
```

のような書き方が良いでしょう。

2.2.2 gem install コマンド

ターミナル上で以下のコマンドを実行するだけです。

```
$ gem install 'rubocop'
```

Rubocop の最新バージョンがインストールされます。(本書執筆時点では 0.74.0) 特定のバージョンを使いたいわけでない限り、バージョンを指定する必要はないでしょう。

2.3 Rubocop の実施

インストールはできたので Rubocop が正しく動作するか確認してみましょう。

2.3.1 Rubocop 実施コマンド

インストールの仕方によって、実行する際のコマンドが若干変わります。
「Gemfile に記載」した場合

```
$ bundle exec rubocop target.rb
```

「gem install コマンド」を用いた場合

```
$ rubocop target.rb
```

bundle exec をつける場合とつけない場合では、利用する Rubocop のバージョンが異なる可能性があります。以下のようにそれぞれでバージョンを確認してみると違いがよく分かります。

```
$ bundle exec rubocop -v
0.70.0
$ rubocop -v
0.74.0
```

2.3.2 Rubocop を実際にやってみる

では実際にサンプルコードを用意して、Rubocop でチェックしてみます。

サンプルコードは以下を用います。簡単なコードですが「Ruby っぽい」と言えない点が多数存在しています。

```
# target.rb
a=1
b = 3

if a.nil?
 result = 1
else
 result = 2
end

p result
```

以下のような実行結果が表示されれば成功です。

```
$ bundle exec rubocop target.rb
Inspecting 1 file
W

Offenses:

target.rb:1:1: C: Style/FrozenStringLiteralComment: Missing magic comment # frozen_string_literal: true.
a=1
^
target.rb:1:2: C: Layout/SpaceAroundOperators: Surrounding space missing for operator =.
a=1
 ^
```

```
target.rb:2:1: W: Lint/UselessAssignment: Useless assignment to variable - b.
b = 3
^
target.rb:4:1: C: Style/ConditionalAssignment: Use the return of the conditional for variable assignment and c
if a != nil ...
^^^^^^^^^^^
target.rb:4:6: C: Style/NonNilCheck: Prefer !expression.nil? over expression != nil.
if a != nil
     ^^
target.rb:10:1: C: Style/IfUnlessModifier: Favor modifier if usage when having a single-line body. Another goo
if result == 1
^^
target.rb:11:1: C: Layout/IndentationWidth: Use 2 (not 1) spaces for indentation.
 p result
^

1 file inspected, 7 offenses detected
```

2.4 Rubocop 指摘の修正方法

前節で確認した Rubocop の実行結果を読んで、コードが「Ruby っぽく」なるように修正しましょう。

2.4.1 Rubocop 実行結果の見方

まず一番最後の行を見ると、Rubocop 実行結果の概要が記載されています。

```
1 file inspected, 7 offenses detected
```

これは 1 つのファイルを Rubocop でチェックし、7 個のエラーを見つけたということです。

offense(s) は Rubocop におけるエラーを表しています。offense には「犯罪」という意味もあるので、Rubocop(ロボコップ) が取り締まった「犯罪」という意味で、offense という言葉が使われているのかもしれません。

今回は 1 つファイルを指定して実行しているため、実施結果には「1 file inspected」と記載されています。仮に$ bundle exec rubocop . という形で実行した場合には、カレントディレクトリ以下の全ての rb ファイルが Rubocop のチェック対象となります。その際は、XX files inspected という実行結果が表示されるでしょう。

では実際に offense の詳細を見て行きたいと思います。実行結果の上の方に戻りまして、「Offenses:」と書かれた行から下にある 7 つのブロックがそれに該当します。直感的にわかる箇所も多いですが、指摘を 1 つ抜き出して全て説明します。この中からは以下 3

点を読み取れれば OK です。

- 指摘箇所はどこか？
- チェックルール名 (Cop) は何か？
- チェックルールの概要

```
target.rb:1:2: C: Layout/SpaceAroundOperators: Surrounding space missing for operator =.
a=1
 ^
```

"target.rb:1:2"は左からファイル名・行・列を表します。この場合は target.rb というファイルの、1 行目& 2 列目に offense が存在する、という意味になります。大抵の場合は、ファイル名と行が分かれば問題ないでしょう。

"C: "は指摘の重大さを表しています。重大さは convention, warning, error, fatal(この順に深刻度が高くなる) の 4 つに分類され、実施結果にはそれぞれの頭文字が表示されます。この場合は"C: "なので convention となり、重大さは一番低い指摘であることが分かります。とはいえ指摘の重大さに関わらず全て修正する、という方針を取る事が多いかと思いますので、あまり気にしなくていい箇所かもしれません。

"Layout/SpaceAroundOperators:"は Rubocop のチェックルールを示しています。なお Rubocop ではチェックルールのことを「Cop」と読んでいます。チェックルールの判別に必要なので、指摘内容の中で一番重要な情報です。詳細は「Rubocop 指摘の修正方法」を参考にしてください。

"Surrounding space missing for operator =."は Cop(チェックルール) の概要を説明してくれています。簡単な Cop(チェックルール) であればここを見るだけで直せることも多いです。

"a=1"&" ^ "はコード上の該当箇所を示しています。ファイル名・行・列の情報と併せて確認することで、どこを直せば良いかわかります。

2.4.2 Rubocop 指摘の修正方法

では実際に指摘内容を修正していきます。

前節でも説明に使った指摘内容を引き続き用います

```
target.rb:1:2: C: Layout/SpaceAroundOperators: Surrounding space missing for operator =.
a=1
 ^
```

まずは「Cop(チェックルール) の概要」と「コード上の指摘箇所」を確認しましょう。これだけで解決できることも多いです。

- Cop(チェックルール) の概要」

Surrounding space missing for operator =. * コード上の指摘箇所 a=1

直訳すると「演算子"="の周囲にスペースが無い」となります。コードを見るとまさに"a=1"で正に"="の周囲にスペースがありませんね。それを踏まえて以下のように修正すれば OK です。

```
# 修正後
a = 1
```

改めて Rubocop を実行すると、指摘が 1 つ減っていることが確認できます。

```
$ bundle exec rubocop target.rb
Inspecting 1 file

(中略)

1 file inspected, 6 offenses detected
```


さて今見てきた指摘はとても簡単な内容でした。ではこちらの指摘はどうでしょうか？

```
target.rb:4:1: C: Style/ConditionalAssignment: Use the return of the conditional for variable assignment and c
if a != nil ...
^^^^^^^^^^^
```

まずは概要を見てみます。Use the return of the conditional for variable assignment and comparison.

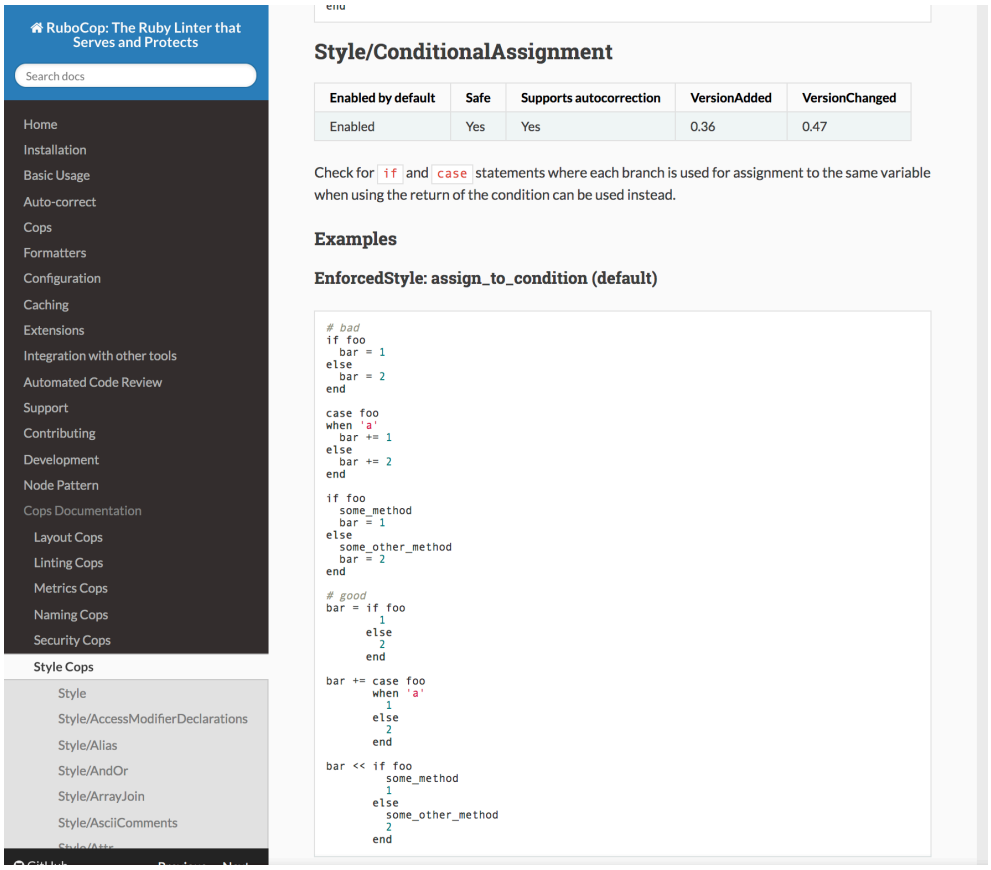
直訳すると「変数 (variable) の代入 (assignment) と比較 (comparison) は条件文 (conditional) の戻り値 (return) を使え」となるでしょうか。

コード上の指摘箇所を踏まえると if の使い方に問題があるように思いますが、これだけだと具体的にどうすれば良いかはちょっと分かりません。

そんな時は公式のドキュメントで Cop(チェックルール) を調べてみましょう。この場合の Cop(チェックルール) は、「Style/ConditionalAssignment」になります。

これを Rubocop の公式ドキュメント https://rubocop.readthedocs.io/en/stable/ 上の検索機能で検索してみます。

すると以下のようなページに辿り着けます。



▲図 2.1　連番つきの図

色々書いてありますが、ひとまずサンプルコードに注目してください。「# bad」に記載されているようなコードを、「# good」の形に直すことが要求されているようです。

ここで改めて target.rb:4:1 を見てみると似たようなコードがあることに気づくと思います。

```
if a.nil?
  result = 1
else
  result = 2
end
```

このコードを、「# good」の形に修正すれば良さそうです。直して見ると以下のようになります。

```
if a.nil?
  result = 1
else
  result = 2
end
```

改めて Rubocop を実行すると、指摘が更に 1 つ減っていることが確認できます。

```
$ bundle exec rubocop target.rb
Inspecting 1 file

(中略)

1 file inspected, 5 offenses detected
```

残りの 5 つに関しても同じように指摘内容への修正を実施していけば OK です。最終的には以下のようなコードになります。

```
# frozen_string_literal: true

a = 1

result =
  if !a.nil?
    1
  else
    2
  end

p result if result == 1
~
```

rubocop --auto-correct を利用した修正

これまでは手動での修正を説明してきましたが、自動修正機能を使うこともできます。--auto-correct(もしくは-a) オプションを追加して実行するだけです。ただしこの機能は、手動で修正できる指摘内容に対してのみ使用するべきです。

修正内容に対して、どんな Cop(チェックルール) に則って修正されたか分からない、という状況は望ましくありません。Rubocop の使い始めの時期は利用を控えた方が良いでしょう。

```
$ bundle exec rubocop -a target_mod_auto.rb
Inspecting 1 file
W

Offenses:

target_mod_auto.rb:1:1: C: [Corrected] Style/FrozenStringLiteralComment: Missing magic comment # frozen_string
a=1
^
target_mod_auto.rb:1:2: C: [Corrected] Layout/SpaceAroundOperators: Surrounding space missing for operator =.
a=1
 ^
target_mod_auto.rb:4:1: W: Lint/UselessAssignment: Useless assignment to variable - b.
b = 3
^
target_mod_auto.rb:4:1: C: [Corrected] Style/ConditionalAssignment: Use the return of the conditional for vari
if a != nil ...
^^^^^^^^^^^
target_mod_auto.rb:4:6: C: [Corrected] Style/NonNilCheck: Prefer !expression.nil? over expression != nil.
if a != nil
     ^^
target_mod_auto.rb:7:3: C: [Corrected] Layout/IndentationWidth: Use 2 (not -7) spaces for indentation.
  1 ...
  ^
target_mod_auto.rb:8:1: C: [Corrected] Layout/ElseAlignment: Align else with if.
else
^^^^
target_mod_auto.rb:9:3: C: [Corrected] Layout/IndentationWidth: Use 2 (not -7) spaces for indentation.
  2 ...
  ^
target_mod_auto.rb:10:1: C: [Corrected] Style/IfUnlessModifier: Favor modifier if usage when having a single-l
if result == 1
^^
target_mod_auto.rb:11:1: C: [Corrected] Layout/IndentationWidth: Use 2 (not 1) spaces for indentation.
 p result
^
target_mod_auto.rb:12:1: C: [Corrected] Style/IfUnlessModifier: Favor modifier if usage when having a single-l
if result == 1
^^

1 file inspected, 11 offenses detected, 10 offenses corrected
```

ただし全て修正できるわけではありません。残った修正は手動で行う必要があります。
例えば target.rb の場合は 1 件だけ自動修正できない指摘内容が残ります。

```
$ bundle exec rubocop target_mod_auto.rb
Inspecting 1 file
W

Offenses:

target_mod_auto.rb:4:1: W: Lint/UselessAssignment: Useless assignment to variable - b.
b = 3
^
```

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

2.5 Rubocop のカスタマイズ

Rubocop 実行時に適用する Cop(チェックルール) はカスタマイズすることができます。カスタマイズは'.rubocop.yml'というファイルを用いて行います。プロジェクトのルートディレクトリに作成すると良いしょう。

Rubocop は実行時に.rubocop.yml の存在を確認し、.rubocop.yml に記載された内容があればそちらを適用し、それ以外は Rubocop の default 設定を適用する、という振る舞いをします。(正確にはもう少し複雑ですが、分かりやすさ優先の説明をしています。詳細は公式ドキュメントをご参照ください)

例えば前節の最後に登場した、Lint/UselessAssignment(rubocop --auto-correct で修正できなかった Cop) をチェックの対象外にする場合には、以下のような.rubocop.yml を作成します。

```
Lint/UselessAssignment:
  Enabled: false
```

実際に Rubocop を運用する際には各チームの事情に合わせて.rubocop.yml を作成することがほとんどです。ただ本書の目的である「Ruby っぽい」を目指すという観点では、カスタマイズはなるべく最小限にすることをオススメします。

(余裕があればカスタマイズ関連を記載する)

2.6 既存プロジェクトへ Rubocop を適用 (--auto-config-gen の利用):

既存プロジェクトに Rubocop を導入する際には、指摘が多すぎて一度に対応することが困難、ということが起こるかもしれません。この本の対象読者 (Ruby 実務経験がおおよそ半年以内) の方が自己判断で Rubocop を導入する状況で、数千件の Rubocop 指摘が出ました！ ということはあまりないかもしれません。それでも Rubocop の指摘が 100 件もあったらうんざりしてしまうでしょう。それで Rubocop の導入を諦めてしまったら元も子もありません。

rubocop --auto-config-gen によって.rubocop_todo.yml を作成することで、既に存在するエラーを別ファイルに保存し、関連する Cop を暫定的に無効にすることができます。

説明だけではピンと来ないと思いますので、target.rb を用いて実際にやってみます。

```
$ bundle exec rubocop --auto-gen-config
Added inheritance from `.rubocop_todo.yml` in `.rubocop.yml`.
Phase 1 of 2: run Metrics/LineLength cop
Inspecting 1 file
.

1 file inspected, no offenses detected
Created .rubocop_todo.yml.
Phase 2 of 2: run all cops
Inspecting 1 file
W

1 file inspected, 7 offenses detected


1. rubocop auto-config-gen
```

この結果、.rubocop_todo.yml と.rubocop.yml が作成されました。この後 Rubocop を実施した場合は、Cop を暫定的に無効しているため指摘が 0 件になります。

```
$ bundle exec rubocop
Inspecting 1 file
.

1 file inspected, no offenses detected
```


2.6.1 .rubocop_todo.yml と.rubocop.yml の役割

どのようにして Cop を暫定的に無効化しているのでしょうか？ 作成された.rubocop_todo.yml と.rubocop.yml の中身から説明します。

まずは.rubocop_todo.yml から。

```
# This configuration was generated by
# `rubocop --auto-gen-config`
# on 2019-08-14 20:45:27 +0900 using RuboCop version 0.74.0.
# The point is for the user to remove these configuration records
# one by one as the offenses are removed from the code base.
# Note that changes in the inspected code, or installation of new
# versions of RuboCop, may require this file to be generated again.

# Offense count: 1
# Cop supports --auto-correct.
# Configuration parameters: Width, IgnoredPatterns.
```

```
Layout/IndentationWidth:
  Exclude:
    - 'target.rb'

(以下略)
```

長いので最初の 1 項目だけ抜き出しました。まず"#"で始まっている行はコメントなので無視してください。実際に理解しておくべきなのは以下の 3 行だけです。

```
Layout/IndentationWidth:
  Exclude:
    - 'target.rb'
```

1 行目は見覚えがあると思います。Cop(チェックルール) の名前です。2 行目 3 行目は読んだ通りの内容です。 'target.rb'を除外する (Exclude) という内容です。まとめると、target.rb を「Layout/IndentationWidth:」の確認対象から除外する、という意味になります。その結果 target.rb に既に存在する「Layout/IndentationWidth:」の確認は今後行われなくなります。この設定が 7 つの offense それぞれに対して行われているのが.rubocop_todo.yml です。

.rubocop.yml の中身は以下の通りです。.rubocop_todo.yml の設定を継承する宣言をしています。

```
inherit_from: .rubocop_todo.yml
```

整理すると、.rubocop_todo.yml で Cop 毎に対象外とするファイルを指定.rubocop.yml で.rubocop_todo.yml を継承する設定を記載することによって、「既に存在するエラーを別ファイルに保存し、関連する Cop を暫定的に無効」を実現しています。

2.6.2 注意点

2 点注意してほしいことがあります。

1 点目はファイル単位で Cop が除外されているということです。つまり.ruboco_todo.yml で Exclude として記載されたファイルに関しては、今後新しい修正に対しても Cop は適用されなくなります。例えば上の例であれば、Layout/IndentationWidth:に該当するコードが target.rb に新しく追加されたとしても、Rubocop は検知してくれません。

2 点目は 1 つの Cop に対して、Exclude 対象が 15 ファイル以上になった時には、Cop そのものが無効化されてしまうということです。その場合 Exclude に記載されたファイ

ルかに関わらず、その Cop に対するチェックは行われなくなります。これを避ける方法としては"--exclude-limit"オプションの利用があります。例えば以下のように rubocop --auto-gen-config コマンドを実行することで、Cop そのものが無効化される閾値を、デフォルトの 15 から変更することができます。

```
bundle exec rubocop --auto-gen-config --exclude-limit=99999
```

第 3 章

Enumerable のメソッドで繰り返し処理を Ruby っぽくする

「Ruby っぽい」コードを書くための方法として次に提案するのは、繰り返し処理における Enumerable のメソッドの利用です。有名な所は map や select が該当します。

繰り返し処理はどんなプログラムでも用いられる頻出の処理であるため、Enumerable のメソッドは利用頻度が高く、Ruby エンジニアも好んで用います。使いこなせれば効率的に「Ruby っぽい」コードを書けるようになるはずです。

Enumerable のメソッドを使うなんて当たり前でしょ？ という方はすみません、この章から得られることは無いかもしれません。

ただ、map?select? なにそれ？ or 聞いたことあるけど... という方、map や select は分かるが Enumerable って何よ？ という方は、是非続きを読んでみてください。

3.1 each メソッドについて

Enumerable の話に入る前の前提知識として each メソッドについて説明します。

Ruby で用いられる繰り返し処理の制御構造・メソッドは数多くありますが、その中でまず押さえておくべきなのは each です。each はオブジェクトの要素の数だけブロックの中身を実行するメソッドです。まず以下に簡単なサンプルコードを示します。

```
list = (1..5).to_a
list.each { |item| p item + 10 }
```

```
# 出力結果
11
12
13
14
15
```

each メソッドに関しては意識しておいてほしい事が 2 点あります。

1 点目はブロック引数です。ブロック引数 (上の例では |item|) には、繰り返しごとにオブジェクトの各要素が順に入ります。ブロック内では、ブロック引数に保持されたオブジェクトの各要素を用いて何かしらの処理を行います。上のコードであれば、item に 10 を加えたものを p メソッドで出力する、といった形です。

Ruby における反復処理は、「オブジェクトの各要素に適用したい処理」を用意しておいて、オブジェクトの各要素を取り出しながら 1 つずつ処理を適用する形が好まれます。(この考え方自体は Ruby 固有の話ではなく、オブジェクト指向ベースの考え方をベースにしたもので、他の言語ににも存在します) each はその考えを実現できる最も基本的なメソッドです。

2 点目は戻り値です。each の戻り値はレシーバ (上の例では list) になります。ブロック内でどんな処理をしても戻り値は影響を受けません。

言い換えれば、each のブロック内で行なった処理の結果は、そのままだと each の外で用いることは出来ません。処理結果を外で使いたい場合は、以下のようにブロック外で定義したオブジェクトを渡す必要があります。

```
list = (1..3).to_a

result = []
list.each { |item| result << item + 10 }
p result
```

```
# 出力結果
[11, 12, 13]
```

3.1.1 each 利用時の課題

each は Ruby の反復処理には欠かせないとても重要なメソッドです。しかし前節で説明した、

- Ruby における反復処理は、「オブジェクトの各要素に適用したい処理」を利用する形が好まれる
- each のブロック内の処理結果を外で使いたい場合は、ブロック外で定義したオブジェクトを渡す必要がある

を考慮すると課題となる点もあります。

その課題とは一言で表すと、each メソッドは「オブジェクトの各要素に適用したい処理」と同時に、「処理の結果をどう扱うか？」もブロック内に一緒に記載する必要がある、という点です。

言葉だけだと分かりにくいので、図 XXX のコードをベースに説明していきます。

このコードにおける「オブジェクトの各要素に適用したい処理」は、item + 10 ですが、実際の処理は、result << item + 10 となっており、「result << 」＝「結果をどうまとめるかの処理」もブロック内で一緒に記載している形をとっています。

前述の通り Ruby における反復処理は、オブジェクトの各要素にある処理を適用する形が好まれます。繰り返し処理の中で「結果をどうまとめるかの処理」も同時に書かれているのはあまり好ましくありません。

またこの処理の実現のために、each の外でわざわざ空の result を用意していますが、これも冗長です。

このような書き方は Ruby 的には少しカッコわるく、大抵の Ruby 開発者は避けます。つまり「Ruby っぽい」と言えません。

3.2 Enumerable のメソッド

そこで登場するのが map や select に代表される Enumerable のメソッドです。

Enumerable は Ruby の組込ライブラリの 1 つで、繰り返し処理に関連する便利なメソッドを提供してくれる Mix-in(モジュール) です。主に配列やハッシュなど集合を表すクラスに標準で組み込まれており、数え上げや検索などのメソッドを提供します。ちなみに読み方は「エニュメラブル (発音記号: injú ☒ m ☒ r ☒ bl, "ニュ"のあたりにアクセント)」です。

Enumerable のメソッドを用いると、「オブジェクト内の各要素に適用したい処理」と「処理の結果をどうまとめるか？」の切り離しが、簡潔なコードで実現できます。「処理の結果をどうまとめるか？」を Enumerable のメソッド側に任せて、自分は「オブジェクト内の各要素に適用したい処理」だけをメソッドに渡せば OK というイメージです。

3.2.1 each を Enumerable のメソッドに置き換える

実際に図 XXX のコードを Enumerable のメソッドを使って書き直してみます。ここでは map を使ってみましょう。map は「オブジェクトの各要素に適用した処理」の結果を「配列にして」返してくれるメソッドです。

```
list = (1..3).to_a

p list.map { |item| item + 10 }
```

```
# 出力結果
[11, 12, 13]
```

まずブロック内には item + 10 の処理、つまり「オブジェクトの各要素に適用したい処理」しか記載されていません。処理結果はブロックの戻り値として map に渡されます。結果をどう扱うか？ は map の中に隠蔽されている形です。map の場合は結果を配列にまとめています。また result も無くなりました。これは map 自体が戻り値として配列を返してくれるため、処理の結果を格納する変数が必要なくなったためです。

並べて比較してみます。

```
list = (1..3).to_a

# each を使った場合
result = []
list.each { |item| result << item + 10 }
p result

# map を使った場合
p list.map { |item| item + 10 }
```

3 行だったコードが 1 行で表現でき、非常にスッキリしたコードになりました。加えてコードの構造が、「オブジェクトの各要素に適用したい処理」を map に渡しているだけとなっているのがよく分かると思います。

3.2.2 Enumerable のメソッドを each で再実装してみる

理解を深めるために、map を ruby で再実装してみます。Enumerable のメソッドは全て each を用いて定義されていますので、each を使って作成できるはずです。(実際の

map と同じものではありませんが、果たしている役割は同じメソッドを作れるはずです)

```
module MyEnumerable
  def my_map
    array = []
    each do |n|
      array << yield(n)
    end
    array
  end
end

class Array
  include MyEnumerable
end

list = (1..3).to_a

p(list.my_map { |item| item + 10 })
```

「オブジェクト内の各要素に適用したい処理」と「処理の結果をどうまとめるか？」が、どのように切り離されているかがよく分かると思います。

まず module MyEnumerable の def my_map に注目してください。図 XXX の構造とほぼ一緒です。異なるのは yield(n) を使っていることくらいでしょう。しかしこの点が重要です。

def my_map からすると、yield(n) は何をやっているかは知りません。とにかく yield(n) の戻り値 (＝ブロックの戻り値) を array に入れる、オブジェクトの要素だけ繰り返す、戻り値として array を返す、ということだけ行っています。この処理は「処理の結果をどうまとめるか？」に該当します。

逆に my_map 呼び出し時のブロックの中身には、item + 10 ＝「オブジェクト内の各要素に適用したい処理」しか記載されていません。処理結果 (＝ブロックの戻り値) をどう扱うかは完全に my_map にお任せしています。

map を使うことでこのように、「オブジェクト内の各要素に適用したい処理」と「処理の結果をどうまとめるか？」を分離したコードが書けます。さらに「処理の結果をどうまとめるか？」は map の中に隠蔽されているので、見た目上かなり簡潔なコードが書けるようになります。

3.3 良く使う＆知っていると便利な Enumerable のメソッド

さて以上のように、Enumerable を使うと繰り返し処理を「Ruby っぽい」形で書けます。ただ Enumerable のメソッドは「処理の結果をどう扱うか」を含んでいるため、目的に応じて適切なメソッドを使い分ける必要もあります。

ここでは Enumerable のメソッドを利用頻度が高そうな順にいくつか紹介します。「利用頻度が高そう」は完全に私の個人的な経験に基づくものです。

簡単な説明と、利用サンプル、それから一部 each で再実装した際のコードを示します。(なお each での実装は、利用サンプルのコードを置き換える事が可能という範囲で作成しています。実際の Enumerable のメソッドとは異なる可能性がありますので、ご了承ください。)

3.3.1 map メソッド

map メソッドは、ブロックの戻り値を配列にして返します。

map に関してはすでに説明した通りですので説明は割愛します。map は Enumerable のメソッドの中で一番よく使います。each を利用する際には、まずは map で置き換えられないか？ を考えてみると良いです。

3.3.2 select, find_all メソッド

find_all メソッドは、ブロック内に書かれた条件式の結果が true になった要素だけを配列にして返します。

select は find_all の別名です。(実際には select の方が多く見るような？)

コード例:

```
list = (1..10).to_a

p list.select { |item| item.even? }

# 下のようにも書けます
p list.select(&:even?)
```

実行結果:

```
[2, 4, 6, 8, 10]
```

each を用いた select の再実装:

```
module MyEnumerable
  def my_select
    array = []
    each { |n| array << n if yield(n) }
    array
  end
end

# Array への include は省略

list = (1..10).to_a
p list.my_select { |item| item.even? }
```

3.3.3 find, detect メソッド

find メソッドは、ブロック内に書かれた条件式の結果が true になった要素を 1 つ返します。detect は find の別名です。


```
list = (1..10).to_a

p list.find { |item| item.even? }

# 下のようにも書けます
p list.find(&:even?)
```

```
2
```

3.3.4 reject メソッド

reject メソッドは、繰り返し処理 (条件式) の結果に合う要素だけ除外した結果を配列にして返します。find_all, select の逆の動作です。

```
list = (1..10).to_a
```

```
p list.reject { |item| item % 3 == 0 }
```

```
[1, 2, 4, 5, 7, 8, 10]
```

3.3.5 max, min, minmax

max, min メソッドはそれぞれ最大値、最小値を返します。minmax メソッドは、最小値&最大値を配列の形で返します。

```
list = (1..10).to_a

p list.max
p list.min
p list.minmax
```

```
10
1
[1, 10]
```


3.3.6 max_by, min_by, minmax_by

max_by, min_by メソッドはブロックの戻り値が最大, 最小になる要素を返します。minmax_by はブロックの戻り値が最大, 最小になる要素を配列の形で返します。

最大、最小の基準を自分で設定できるところが、max, min, minmax との違いです。

```
list = %w(one two three four five six seven eight nine ten)

p list.max_by(&:size)
p list.min_by(&:size)
p list.minmax_by(&:size)
```

```
"three"
"one"
["one", "three"]
```

3.3.7 include?

include?はオブジェクトの要素に引数と同じものが含まれていれば true を、なければ false を返します。

分かりやすい機能で利用頻度も結構高い印象です。

```
list = (1..10).to_a

list.include?(3)
list.include?(11)
```

```
true
false
```

3.3.8 all?, any?, none?メソッド

all?メソッドはブロックの戻り値が全て真だった場合に true を返します。それ以外は false を返します。any?メソッドはブロックの戻り値が 1 つでも真だった場合に true を返します。それ以外は false を返します。none?メソッドはブロックの戻り値が全て偽だった場合に true を返します。それ以外は false を返します。

特に any?あたりはたまに使う印象です。

```
list = (1..10)

p list.all? { |item| item.class == Integer }
p list.any? { |item| item.to_s.size == 2 }
p list.none? { |item| item < 1 }
```

```
true
true
true
```

3.3.9 reverse_each

名前の通り、逆から実施する each です。繰り返し毎にブロック引数に要素が逆から入ります。

reverse.each と書くとちょっとカッコ悪いので、こちらを使うといいです。

```
list = (1..5)

list.reverse_each { |item| p item }
```

```
5
4
3
2
1
```

3.3.10 each_with_index メソッド

each のブロック内でオブジェクトの各要素だけでなく、繰り返しの回数に対応した index も利用できるようにしたメソッドです。

よく使いそうで意外と使わない、ようでやっぱり使う、そんな感じのメソッドです。

```
list = %w(water coffee tea milk)

list.each_with_index { |item, i| p "No.#{i} #{item}" }
```

```
"No.0 water"
"No.1 coffee"
"No.2 tea"
"No.3 milk"
```

3.3.11 group_by

group_by メソッドは要素をグループ分けします。具体的にはブロックの戻り値をキーとしたハッシュを作成します。使わないように見えてたまに欲しくなります。覚えている

ととても便利です。
再実装してみると何をしているかよく分かります。

```
animals = ["cat", "bat", "bear", "camel", "alpaca"]

p animals.group_by { |item| item[0].chr }
```

```
{"c"=>["cat", "camel"], "b"=>["bat", "bear"], "a"=>["alpaca"]}
```

```
module MyEnumerable
  def my_group_by
    hash = {}
    each do |item|
      key = yield(item)
      hash[key] ||= []
      hash[key] << item
    end
    hash
  end
end

class Array
  include MyEnumerable
end

animals = ["cat", "bat", "bear", "camel", "alpaca"]
p animals.my_group_by { |item| item[0].chr }
```


3.3.12 partion

partition メソッドは、ブロックの戻り値が真の要素と偽の要素でグループ分けをします。結果は [[真の要素...], [偽の要素...]] の形で返します。

```
list = (1..10).to_a

p list.partition { |item| item.odd? }
```

```
[[1, 3, 5, 7, 9], [2, 4, 6, 8, 10]]
```

3.3.13 inject/reduce

inject メソッドは、オブジェクトの要素を 2 つずつ使用して畳み込み計算を行うメソッドです。結構使います。

言葉で説明するよりコードをみてもらった方が早いです。こちらも each での再実装を用意しました。yield の結果を保持しておいて、次のループで yield の引数の 1 つに使うだけです。

```
list = (1..10).to_a

p list.inject { |i, j| i + j}
p list.inject(30) { |i, j| i + j}

# 以下のようにも書けます
# p list.inject(:+)
# p list.inject(30, :+)
```

```
55
85
```



```
module MyEnumerable
  def my_inject(init=nil)
    prev = init
    each { |item|
      if prev.nil?
        prev = item
        next
      end
      prev = yield(prev, item)
    }
    prev
  end
end

class Array
  include MyEnumerable
end

list = (1..10).to_a
p list.my_inject(30) { |i, j| i + j}
```

第 4 章

Ruby Sliver を受けて Ruby の全体像を掴む

本章では Ruby の言語仕様全体を理解するための第 1 歩として、Ruby Silver の取得を紹介します。

RubySilver(正式名称は Ruby Association Certified Ruby Programmer Silver version 2.1：) は、Ruby アソシエーションが運営する Ruby 技術者認定試験のの 1 つです。資格の特徴は以下の通りで、Ruby 初級者が取得するのに非常に向いています。

> Ruby の文法知識、Ruby のクラスとオブジェクト、標準ライブラリの知識について、基本的な技術レベルを持つことを認定します。

4.1 Ruby Silver をオススメする理由

これまではかなり実用的な話をしてきましたので、ここにきていきなり資格の話になったことに違和感を感じる方もいらっしゃるでしょうか。

本章はこれまで、Rubocop & Enumerable といった内容を紹介してきましが、これらは利用頻度の高く、「Ruby っぽい」コードを目指す上で欠かせませんが、ある意味偏った内容ではあります。

Ruby Sliver の取得を通して、Ruby の言語仕様の基本的な部分を満遍なく学ぶことで、よりバランスよく「Ruby っぽい」コードが書けるようになるはずです。

実際私も Ruby Silver を 2019 年 2 月に取得しましたが、「Ruby っぽい」コードに限らず、想像以上に仕事で役に立ったなぁと感じています。

それまで利用したことのない組込ライブラリのメソッドを自然と使いこなせるようになったり、先輩方の書くコードを割とサクッと読めるようになる部分があったりなど、強

い実感ではありませんがボディーブローのように効いてきたように思っています。

おそらく実務経験を半年〜1 年以上積んだ方は得るものが多くないかもと思いますが、Ruby 初級者という範囲に入る方であれば、十分元の取れる投資ではないかな？ と思っています。

ということで本章では以降 RubySilver の特徴や勉強方法、当日の注意点などを記載します。興味のある方はぜひ参考にしてみてください。

4.2 Ruby Sliver の特徴

Ruby アソシエーションに記載された試験要項に沿って、RubySilver の特徴を説明していきます。https://www.ruby.or.jp/ja/certification/examination/

4.2.1 試験時間＆問題数

試験時間は 90 分、問題数は 50 問です。

この点に関しては全く心配する必要はありません。時間は確実に余ります。本番の試験問題は過去問や問題集 (後述) とほぼ同じ内容が結構出ます。50 問中 30〜35 問程度 (6〜7 割) はそんな感じです。ある程度勉強していれば、それらは 20 分もあれば解けるでしょう。そうなると残り 20 問を 70 分で解く形になるわけです。時間は十分余裕があります。焦らずやりましょう。

4.2.2 合格ライン

合格ラインは 75% です。

前述の通り 6〜7 割は過去問や問題集と似たような内容が出ます。ただこの辺りが絶妙で、それだけだと合格できないかなぁというラインです。学習の際は過去問丸暗記ではなく、理由や背景含めて掘り下げるべきです。

4.2.3 試験方法

試験方法はコンピュータ試験（CBT：Computer Based Testing）です。

試験はパソコンを使って回答を選択する形になります。4 つの中から 1 つないし複数の答えを選択する形になります。これは特に注意することはありません。手書きじゃなくて楽だなぁというくらいです。

4.2.4 対象バージョン

対象バージョンは Ruby2.1.x です。

この点は少し注意が必要です。本書執筆時点 (2019 年 9 月) では Ruby2.6.3 が最新ですが、試験の対象バージョンは 2.1.x です。

個人的な経験の範囲ではこの点で大きく悩まされたことはありませんでしたが、過去問や問題集の内容と実際の Ruby の動作が異なる、という事は起こり得ますのでご注意ください。

4.2.5 出題範囲

出題範囲は以下の通りです。

* 文法コメント, リテラル（数値、真偽値、文字列、文字、配列、ハッシュ等）, 変数/定数とスコープ, 演算子, 条件分岐, ループ, 例外処理, メソッド呼び出し, ブロック, メソッド定義, クラス定義, モジュール定義, 多言語対応
* 組み込みライブラリよく使用されるクラス、モジュール（Object、数値クラス、String、Array、Hash、Kernel、Enumerable、Comparable 等）
* オブジェクト指向ポリモルフィズム, 継承, mix-in

4.3 申し込み方法

4.4 学習方法

以下学習方法を記載する。これは私が Qiita に投稿した記事、「Ruby Silver に合格したので、勉強方法をまとめてみた (2019 年 2 月版)」https://qiita.com/jonakp/items/7f7550eeea78973a0a7f をリファインしたものになっているので、注意されたし

4.4.1 とりあえず結論

* 合格記事をさら〜っと眺める* とにかく色々な問題集を解きまくる* 各問題集で 9 割以上正答＆理由も説明できるようにする* 仕上げに先人が残してくれた要注意情報に目を通す

4.4.2 前提

学習開始時点での私のスペックを記載する。以下勉強方法を読む際の参考に。

* Ruby に関しては 3 ヶ月独学＆研修で 2 週間* Ruby の経験は無くはない、という程度* 他言語の経験あり** 大学時代 C/Java を履修 (?) ** 会社員になってから 5 年程度、たまーに業務でコードに触れる程度* IT 系の知識は薄く広く** 基本情報は持っているとかその程度。* 学習期間 & 学習時間** 学習期間：約 2 週間** 学習時間：約 30 時間

4.4.3 学習方法

1. 合格記事をさら〜っと眺める

"Ruby Silver", "勉強法"あたりでググって記事を眺めてみる。

勉強方法やら出題傾向やらが書いてあるので、なんとなく雰囲気を理解する。

例えば、

* String/Array/Hash クラスのインスタンスメソッドに関する出題が多い* 同じ機能で別の名前を持つメソッドの記憶は必須* 破壊的メソッドとそうでないメソッドの見分けも重要* File/Dir/Time クラスに関しての出題もそこそこ多い

などなど。。。

細かく理解する必要はなく、そんな感じなんだなぁと思うくらいで OK。

適当に Google 検索結果の上から 5,6 個いい感じのを、流し読みするくらい。

2. とにかく問題集を解く

具体的な対策としては、問題集を周回するのが一番手っ取り早い。

ソシャゲのマラソンみたいなノリで、ひたすら走るべし。

(ただし Ruby のことを何も知らない、プログラミングもほとんどやったことない、という人は対象外。Ruby の入門本を 1 冊消化した方が多分良い)

1 つだけじゃ無くて、色々な問題集を取っ替え引っ替えするのがオススメ。やってるうちに、共通点やら傾向が自然と理解できる。

私が実際に使った問題集は、主に以下の 3 つ。

■公式模擬問題集 https://www.ruby.or.jp/assets/images/ja/certification/examination/exam_prep_jp.pdf pdf と見せかけてどこぞの github リポジトリに飛ぶ。無料。一番最初にやって 7 割弱取れた。恐らく一番難易度低め。

REx https://www.ruby.or.jp/ja/certification/examination/rex 受ける度に問題の内容が変わる。8 割共通で残りの 2 割が入れ替わる。もちろん以前の問題を再び受け直すこ

とも可能。解説も割と書いてある。スマホからも受けられて便利。もちろん無料。恐らく一番使った。github アカウントとの連携が必要。

[改訂 2 版］Ruby 技術者認定試験合格教本（Silver/Gold 対応）Ruby 公式資格教科書 https://www.amazon.co.jp/dp/B0756VF9Y3/ref=dp-kindle-redirect?_encoding=UTF8&btkr=1 これだけ普通の書籍。3600 円と結構なお値段がするが、十分その価値はある。問題集だけでなく、出題範囲に関する一通りの知識も記載されているので、リファレンスとしても活用できる。

時間があるなら、3,4,5 章あたりは一通り読んでおいても良い。ただし問題集を解いた上で、不明な点に絞って読んだ方が遥かに効果がある。==== 各問題集で 9 割以上正答＆理由も説明できるようにする上記で挙げたいくつかの種類の問題集で、9 割以上解ける＆理由も説明できるようになるまで、知識の完成度を高めていく。

その際、理解するための手段は色々な方法を試すと良い。

問題集の解答を読む Ruby 技術者認定試験合格教本の該当箇所を読むネットで関連情報をググる (主に Ruby リファレンスなど) irb で実際に動作を確かめるなどなど。。。この辺りを広くやればやるほど、応用の効く知識が身につき、恐らく本番に強くなる。(例えば正答に関係ない選択肢でも、分からなければ積極的に調査する、など）

逆に問題集の解答を丸暗記しただけだと、本番で足元をすくわれる可能性が高くなる、と思う。

仕上げに先人が残してくれた要注意問題に目を通す

本番の試験は、6,7 割は問題集ほぼそのままの内容だったが、残りは妙にひねった所のある、いやらしい内容だった。

試験の前日などに、先人が残したくれた情報に目を通すとその辺りの対策になる。

私が実際に直前に確認して助かったのは以下 2 つの記事。そのおかげで多分 3 問くらいは助かった。

Ruby Silver 試験前に見直すと幸せになれるメモ http://tamata78.hatenablog.com/entry/2015/08/07/200454

Ruby 技術者認定試験 Silver version 2.1 を受けて…https://qiita.com/motty93/items/413485469e4ec665c329

4.5　当日の注意点

- 身分証等は持ち込む- 思いの外時間には余裕がある- 合格証は直接相談すると早く出してくれる、かもしれない。- 盾とかがもらえる- 会場にも依るが物品の持ち込みはかなり厳しく制限される

第 5 章

おわりに

付録 A

Rubocop を CI ツールと連携する

ここまでで Rubocop の導入と利用に関する基本的な流れを説明してきました。一応今までの情報だけでも Rubocop を用いてコードを「Ruby っぽく」することは実現できるでしょう。

しかし 1 つ懸念があります。それは継続して Rubocop を活用し続けられるか？ ということです現状はターミナルから手動で Rubocop を実行することを前提としています。今のままだと Rubocop の実行を忘れてしまったり、その内めんどくさくなって辞めてしまったりすることは十分起こりえるでしょう。せっかく Rubocop を導入したにも関わらず、「Ruby っぽく」するという目標は達成できないかもしれません。

そこで Rubocop を CI ツールと連携させることで、Rubocop を継続的に確実に実施する方法をお伝えします。

ただこの内容はチーム開発をしている環境では自分の一存で決められることではありません。またこの手の強制的なルールの適用を嫌がる人も沢山います。個人開発の環境であれば問題ありませんが、チーム開発の場合はまずメンバに相談してみることを強くお勧めします。

(circle ci も余裕があったらやる感じにする)

付録 B

便利だけどちょっとマニアックな Enumerable のメソッド

B.0.1 flat_map

B.0.2 each_with_object メソッド

each_with_object メソッドは、each での繰り返し処理の中でオブジェクトを扱うことができます。inject メソッドと多少似ていますが、、、

著者紹介

第 1 章 ひつじ / @mhidaka

ひつじだよ〜

Ruby っぽいコード入門 (仮称)

2019 年 4 月 14 日　技術書典 6 版 v1.0.0

著　者　XXXX 編
編　集　mhidaka
発行所　XXX