Ruby っぽいコード入門

@kappaz 編　著

2019-09-22 版　　かっぱ書房 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。

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。

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

また以下のような方々を想定読者としています。

- 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 Ruby 初級者に入った位の方です。

「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。

目次

第 1 章

「Ruby っぽい」について

1.1 「Ruby っぽい」とは？

まず「Ruby っぽい」とは何でしょうか？ これは実務経験半年の私が論ずるにはあまりにも危険な話題で、実際ビクビクしながら書いています。しかしあえて主張させていただきます。「Ruby っぽいとはこれだ！」という概念は…特にありません。

決して逃げたわけではありません。きちんと根拠があります。Ruby はそもそも色々な書き方ができる言語です。「同じ目的であっても様々な選択肢があり、開発者によって自分の好きなやり方を選べることで、楽しくプログラミングを行おう」といった方針を持っています。そのため特定の書き方を「Ruby っぽい」と公式に推奨するようなことはありません。

ただそれでも、現実的には「Ruby っぽい」という言葉は存在しています。実際「Ruby っぽい」や「Ruby らしい」と行った表現はインターネット上の技術記事に散見されます。それらは大きな意味での「概念」や「哲学」といったものではなく、個々の状況において多くの Ruby 技術者が支持する、メソッドの使い方やコーディングスタイルを示しているようです。それを須藤 功平さんが抽象的にわかりやすく表現した内容を、RubyMagazine の記事 (https://magazine.rubyist.net/articles/0043/0043-BeALibraryDeveloper.html) から引用します。

> 「Ruby らしい」とはどういうことでしょうか。「○○らしい」とは「他と似ている」ということです。「Ruby らしい」書き方だとまわりのコードと似たような記述になります。つまり、まわりのコードと統一感がでるということです。

この表現が現状最も適切に「Ruby っぽい」を表現していると、私は感じています。

1.2 本書における「Ruby っぽい」の定義

前節の内容を踏まえ、本書では以下を「Ruby っぽい」コードの定義とします。

- Ruby エンジニアにとって馴染みのある表現を用いたコード

前述の通りここで言う「Ruby っぽい」というのは、個々の状況におけるベストプラクティスです。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。そこで本書では、成果が得られやすい (= 効率が良い) 内容をピックアップして、効率的にある程度「Ruby っぽい」コードを書くための方法を紹介します。

1.3 「Ruby っぽい」コードを書くメリット

「Ruby っぽい」が具体的に何を表すか？ は説明しました。では「Ruby っぽい」コードにするとどんなメリットがあるのでしょうか？

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。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 Ruby 初級者目線でまとめてここで紹介します。

1.3.1 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重要です。自分が書いたコードは、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 エンジニアから見て馴染みのある＝読みやすいコードになります。

1.3.2 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、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で

す。余りにも多くの指摘を受け、お互い悪気はないのに雰囲気が悪くなる、そういった事は頻繁に起こり得ます。「Ruby っぽい」コードを書ければ、プルリクエスト等のコードレビュー時に上司や先輩から細かい指摘を受けにくくなり、少数のより重要なやりとりに集中できるでしょう。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。

1.3.3 他の Ruby エンジニア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。

Ruby 初級者が読むコードは、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、自分より経験のある Ruby エンジニアが書いたものである事が多いのではないでしょうか？ そういったコードは、Ruby でよ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コードである事が多く、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を見て、パッと理解できることが増えるでしょう。

1.3.4 プログラミング自体が楽しくなってくる

私の経験では「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、それはそれで導入方法の確認やメンテナンス状況、他のライブラリとの相性問題など、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。[*1]

[*1] 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。(https://github.com/rubocop-hq/rubocop)

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(Ruby のライブラリ) 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。コードの品質向上をサポートしてくれるとても頼もしいやつです。

Rubocop は「Ruby エンジニアであれば知らない人はいない」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 エンジニアにとって見慣れたものである可能性が高く、本書の目的である「Ruby っぽい」コードを目指す上で、最初に導入する内容としてもってこいです。まずは Rubocop のアドバイスに従うことで「Ruby っぽい」コードを目指しましょう。

ここからは Rubocop の導入に関して以下の流れで段階的に説明していきます。

1. Rubocop によるコード修正の例
2. Rubocop 導入前の注意点
3. Rubocop のインストール
4. Rubocop の実施
5. Rubocop 指摘の修正方法
6. Rubocop のカスタマイズ
7. 既存プロジェクトへ Rubocop を適用する際の注意事項
8. Rubocop を部分的に無効化する

2.1 Rubocop によるコード修正の例

まず Rubocop を使うと具体的にはどのようにコードを「Ruby」っぽく出来るか見てみましょう。修正対象のサンプルコードとしては以下を用います。単純なコードですが、Ruby っぽくない点が多数存在しています。

▼リスト 2.1 target.rb: Ruby っぽくないコード

```
a=1
b = 3
c = [1, 2, 3, 4, 5]
d = []

if a > 10 then
  result = 1
else
  result = 2
end

if result == 1
 p result
end

for num in c do
  d << "#{num*10}"
end

p d
```

Rubocop の指摘を元にコードを修正すると以下のようになります。(Rubocop の設定はデフォルトです)

▼リスト 2.2 target_after_fix.rb: Rubocop によって修正したコード

```
# frozen_string_literal: true

a = 1
c = [1, 2, 3, 4, 5]
d = []

result = if a > 10
           1
         else
           2
         end

p result if result == 1

c.each do |num|
  d << (num * 10).to_s
```

```
end

p d
```

まだまだ修正の余地はありますが、大分「Ruby っぽいコード」に近づいたと思います。どのような手順で修正を行うか？ は章の後半で紹介します。

2.2 Rubocop 導入前に意識しておきたい事

Rubocop 導入前に 2 点ほど意識していただきたいことがあります。

2.2.1 Rubocop の本来の用途

本書では Rubocop を、自分が書いたコードを「Ruby」っぽくするという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ける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。[*1]

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、適切な文法などが挙げられます。仕事として行うプログラミングはたいてチームで行います。チームメンバ間でなるべく書き方を統一し、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統一の目的であり、Rubocop はそのために用いられるのが一般的な用途です。

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、本来の用途は頭に入れておくと良いです。

2.2.2 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異なるのが普通です。Rubocop を使わずに、チーム独自のコーディングスタイルが規定されているような場合もあると思いますが、その場合はチームの方針を優先しましょう。

Rubocop を利用してより「Ruby っぽい」コードを目指すことにはもちろん価値があると思いますが、前節において説明したような、「チーム開発においてコードの品質を統一すること」の方が優先されるべきです。

もちろんチームの方針に反しない範囲で Rubocop を導入するのも良いと思います。む

[*1] こういったツールは一般的に linter と呼ばれ大抵の言語に存在しています。

しろ周囲が Rubocop を使っていないなら、相対的により「Ruby っぽいコード」を書ける可能性が高く、周囲に差がつけられます。あなたがもし Rubocop を用いていない環境にいるならば、こっそり自分だけ使って見るのは良いアイデアかもしれません。

2.3 Rubocop のインストール

ここから Rubocop を本格的に導入していきます。まずは Rubocop のインストールに関して説明します。やり方は以下の 2 パターンです。

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 に記載するのが良いと思います。ただチーム開発の環境で自分だけで Rubocop を使いたいような場合には、(勝手に Gemfile に追加できないと思いますので)gem install を用いるのも良いでしょう。もしくは両方やっても問題ありません。

2.3.1 Gemfile に記載 (bundler を利用)

Gemfile に以下のように記載した上で、bundle install を実行するだけインストールが完了します。とても簡単です。

```
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。後々 bundle install によって意図せずに Rubocop のバージョンが変わってしまうことを避けるため、Gemfile には指定バージョンも記載した方が良いでしょう。

Rails を用いたプロジェクトの Gemfile に記載する場合は、開発環境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。このように設定することで、本番環境で Rails のアプリケーションを動かした際、使用しない Rubocop が組み込まれることが無くなります。

2.3.2 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。

2.4 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。

2.4.1 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確認してみると違いが分かります。

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

2.4.2 Rubocop を実際に使ってみる

では実際にサンプルコードを用いて、Rubocop でコードのチェックを試してみます。サンプルコードはリスト 2.1 で示したコードを用います。以下のような実行結果が表示されれば成功です。

```
$ 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 opera
tor =.
a=1
 ^
target.rb:2:1: W: Lint/UselessAssignment: Useless assignment to variable - b.
b = 3
^
target.rb:6:1: C: Style/ConditionalAssignment: Use the return of the conditional f
or variable assignment and comparison.
if a > 10 then ...
^^^^^^^^^^^^^^
target.rb:6:11: C: Style/MultilineIfThen: Do not use then for multi-line if.
if a > 10 then
          ^^^^
target.rb:12:1: C: Style/IfUnlessModifier: Favor modifier if usage when having a s
ingle-line body. Another good alternative is the usage of control flow &&/||.
if result == 1
^^
target.rb:13:1: C: Layout/IndentationWidth: Use 2 (not 1) spaces for indentation.
 p result
^
target.rb:16:1: C: Style/For: Prefer each over for.
for num in c do ...
^^^^^^^^^^^^^^^
target.rb:17:8: C: Style/UnneededInterpolation: Prefer to_s over string interpolat
ion.
  d << "#{num*10}"
       ^^^^^^^^^^
target.rb:17:14: C: Layout/SpaceAroundOperators: Surrounding space missing for ope
rator *.
  d << "#{num*10}"
             ^
target.rb:20:4: C: Layout/TrailingBlankLines: Final newline missing.
p d
```

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

2.5 Rubocop 指摘の修正方法

前節で確認した Rubocop の実行結果を参考にサンプルコードを修正して、「Ruby っぽい」コードにしてみましょう。

2.5.1 Rubocop 実行結果の見方

まず実行結果の見方について説明していきます。

実行結果の概要

一番最後の行に Rubocop 実行結果の概要が記載されています。

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

これは 1 つのファイルを Rubocop でチェックし、11 個のルール違反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」という意味がありますので、Rubocop(ロボコップ) つながりで 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 つの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。

offense の詳細

次に offense の詳細を見て行きたいと思います。実行結果の「Offenses:」と書かれた行から下にある 11 個のまとまり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。指摘からは以下 3 点が読み取れれば OK です。

- 指摘箇所はどこか？
- 指摘の深刻度はどの程度か？
- 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 opera
tor =.
a=1
 ^
```

「target.rb:1:2」は指摘箇所はどこか？ を示しています。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。さらに詳しい情報として、「a=1」のような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。

「C: 」は指摘の深刻さを表しています。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。あまりにも指摘が多い場合は「深刻度が高い指摘のみ対応する」という方針もあります。

「Layout/SpaceAroundOperators:」は Rubocop のチェックルール名 (Cop) は何か？ 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と修正方法の調査に必要なので、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 の概要を説明してくれています。簡単な Cop であればここを見るだけで直せることも多いです。

2.5.2 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 operat
or =.
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, 10 offenses detected
```

さて今見てきた指摘はとても簡単な内容でした。では次の指摘はどうでしょうか？

```
target.rb:6:1: C: Style/ConditionalAssignment: Use the return of the conditional f
or variable assignment and comparison.
if a > 10 then ...
^^^^^^^^^^^^^^
```


まずは概要を見てみます。

```
Use the return of the conditional for variable assignment and comparison.
```

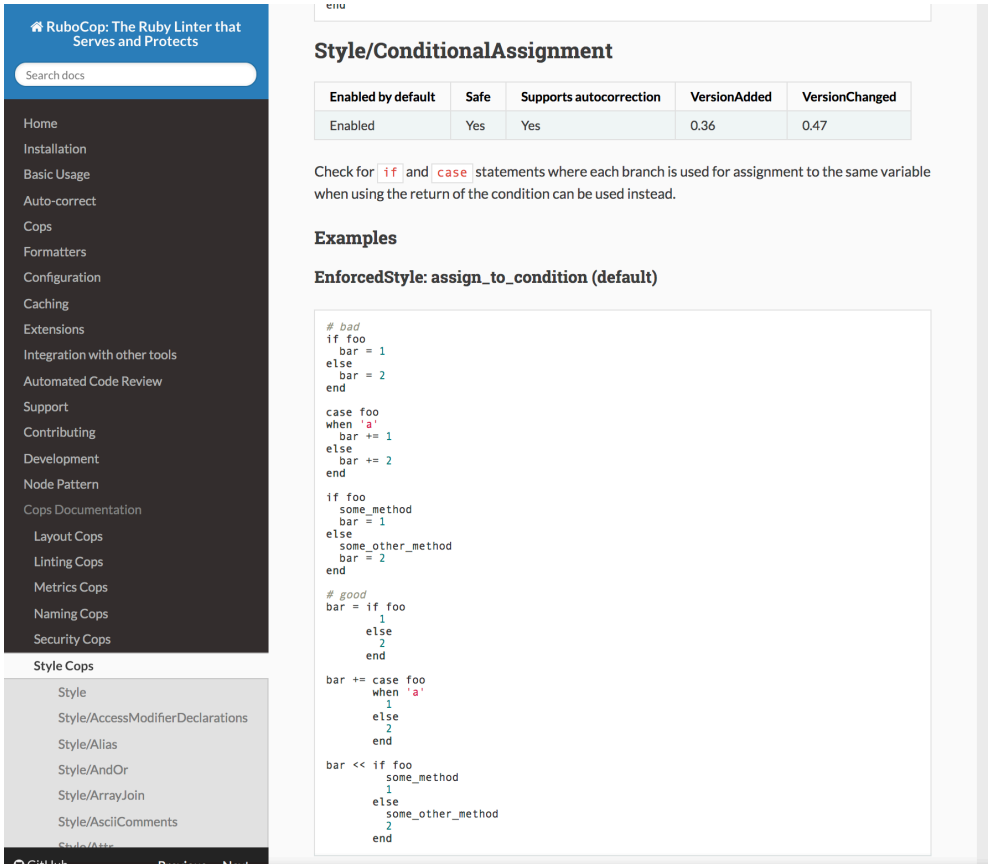
直訳すると「変数 (variable) の代入 (assignment) と比較 (comparison) には条件文 (conditional) の戻り値 (return) を使え」となるでしょうか。

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直せ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 を調べてみましょう。この場合の Cop 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。

すると次のページに辿り着きます。



▲図 2.1　Style/ConditionalAssignment の解説

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ます。

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。

```
result = if a > 10
           1
         else
           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。

Ruby の if は式なので値が戻ってくるのでこのような書き方が可能になります。Ruby の特徴を押さえた「Ruby っぽい」コードになりました。

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

(中略)

1 file inspected, 10 offenses detected
```

残りの 10 個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。(リスト 2.2 の再掲です)

▼リスト 2.3　target_after_fix.rb: Rubocop によって修正したコード (再掲)

```
# frozen_string_literal: true

a = 1
c = [1, 2, 3, 4, 5]
d = []

result = if a > 10
           1
         else
           2
         end

p result if result == 1

c.each do |num|
  d << (num * 10).to_s
end

p d
```

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、自力で修正できる指摘内容に対してのみ使用するべきです。「修正内容がどんな Cop に則った修正か分からない」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、慣れてきた頃に活用すると良いと思います。

```
$ rubocop -a target.rb
Inspecting 1 file
W

Offenses:

target.rb:1:1: C: [Corrected] Style/FrozenStringLiteralComment: Missing magic comm
ent # frozen_string_literal: true.
a=1
^
target.rb:1:2: C: [Corrected] Layout/SpaceAroundOperators: Surrounding space missi
ng for operator =.
a=1
 ^
target.rb:4:1: W: Lint/UselessAssignment: Useless assignment to variable - b.
b = 3
^
target.rb:6:1: C: [Corrected] Style/ConditionalAssignment: Use the return of the c
onditional for variable assignment and comparison.
if a > 10 then ...
^^^^^^^^^^^^^^
target.rb:6:11: C: [Corrected] Style/MultilineIfThen: Do not use then for multi-li
ne if.
if a > 10 then
          ^^^^
target.rb:9:3: C: [Corrected] Layout/IndentationWidth: Use 2 (not -7) spaces for i
ndentation.
  1 ...
  ^
target.rb:10:1: C: [Corrected] Layout/ElseAlignment: Align else with if.
else
^^^^
target.rb:11:3: C: [Corrected] Layout/IndentationWidth: Use 2 (not -7) spaces for
indentation.
  2 ...
  ^
target.rb:12:1: C: [Corrected] Style/IfUnlessModifier: Favor modifier if usage whe
n having a single-line body. Another good alternative is the usage of control flow
 &&/||.
if result == 1
^^
target.rb:13:1: C: [Corrected] Layout/IndentationWidth: Use 2 (not 1) spaces for i
ndentation.
 p result
```

```
^
target.rb:14:1: C: [Corrected] Style/IfUnlessModifier: Favor modifier if usage whe
n having a single-line body. Another good alternative is the usage of control flow
 &&/||.
if result == 1
^^
target.rb:16:1: C: [Corrected] Style/For: Prefer each over for.
for num in c do ...
^^^^^^^^^^^^^^^
target.rb:17:8: C: [Corrected] Style/UnneededInterpolation: Prefer to_s over strin
g interpolation.
  d << "#{num*10}"
       ^^^^^^^^^^
target.rb:17:14: C: [Corrected] Layout/SpaceAroundOperators: Surrounding space mis
sing for operator *.
  d << "#{num*10}"
             ^
target.rb:20:4: C: [Corrected] Layout/TrailingBlankLines: Final newline missing.
p d


1 file inspected, 15 offenses detected, 14 offenses corrected
```

ただし--auto-correct オプションで offense を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 offense が残ります。

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

Offenses:

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

1 file inspected, 1 offense detected
```

2.6 Rubocop のカスタマイズ

Rubocop 実行時に適用する Cop はカスタマイズできます。'.rubocop.yml'というファイルを用いてカスタマイズし、プロジェクトのルートディレクトリに作成すると良いでしょう。

Rubocop は実行時に「.rubocop.yml の存在確認&記載内容の適用を行い、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。

例えば前節の最後に登場した、Lint/UselessAssignment 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 でカスタマイズを行うことがほとんどです。

もし個人開発等で.rubocop.yml を 0 から作成する場合には、以下のような企業が公開している規約が参考にするのも良いでしょう。

- クックパッド

https://github.com/cookpad/styleguide/blob/master/.rubocop.yml

- airbnb

https://github.com/airbnb/ruby/tree/master/rubocop-airbnb

また本書の目的である「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のもアリかと思います。

2.7 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 target.rb --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, 11 offenses detected
Created .rubocop_todo.yml.
```

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 target.rb
Inspecting 1 file
.

1 file inspected, no offenses detected
```

2.7.1 .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 から説明します。

```
# 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 Cop(チェックルール) の名前です。2 行目 3 行目は '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。

2.7.2 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 から変更できます。

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

2.8 Rubocop を部分的に無効化する

前節では新規プロジェクトに Rubocop を導入する際の便利なオプションを紹介しましたが、本節では、既に Rubocop 導入済みのプロジェクトにおいて、部分的に Rubocop を無効化する方法を説明します。例えば.rubocop.yml では Style/LineLength(1 行あたり何文字まで許容するか) の Cop を 100 文字に設定しているが、ここはどうしても 120 文字位書きたい、といった際に便利です。

やり方はとても簡単です。以下のように無効化したい箇所をコメントで囲うだけです。

```
# rubocop:disable all
***除外したいコード***
# rubocop:disable all
```

試しにリスト 2.1 のサンプルコードを利用して、Rubocop を部分的に無効化してみます。

▼リスト 2.4　部分的に Rubocop を無効化

```
 1: # rubocop:disable all
 2: a=1
 3: b = 3
 4: c = [1, 2, 3, 4, 5]
 5: d = []
 6:
 7: if a > 10 then
 8:   result = 1
 9: else
10:   result = 2
11: end
12: # rubocop:emable all
13:
14: if result == 1
15:  p result
16: end
17:
18: for num in c do
19:   d << "#{num*10}"
20: end
21:
22: p d
```

```
$ bundle exec rubocop target_commentout.rb
Inspecting 1 file
C

Offenses:

target_commentout.rb:14:1: C: Style/IfUnlessModifier: Favor modifier if usage when l
if result == 1
^^
target_commentout.rb:15:1: C: Layout/IndentationWidth: Use 2 (not 1) spaces for inde
 p result
^
target_commentout.rb:18:1: C: Style/For: Prefer each over for.
for num in c do ...
^^^^^^^^^^^^^^^
target_commentout.rb:19:8: C: Style/UnneededInterpolation: Prefer to_s over string i
  d << "#{num*10}"
       ^^^^^^^^^^
target_commentout.rb:19:14: C: Layout/SpaceAroundOperators: Surrounding space missin
  d << "#{num*10}"
            ^
target_commentout.rb:22:4: C: Layout/TrailingBlankLines: Final newline missing.
p d


1 file inspected, 6 offenses detected
```

結果を見ると指摘内容が 14 行目以降のものに限定されてお有り、「2.4.2 Rubocop を実際に使ってみる」で示した実行結果と比較して半分程度になっているのが見て取れます。

また以下のように特定の Cop に絞って無効化することもできます。(Style/IfUnlessModifier を無効化する場合)

```
# rubocop:disable Style/IfUnlessModifier
***除外したいコード***
# rubocop:disable Style/IfUnlessModifier
```

▼リスト 2.5　部分的に特定の Cop を無効化

```
 1: # rubocop:disable all
 2: a=1
 3: b = 3
 4: c = [1, 2, 3, 4, 5]
 5: d = []
 6:
 7: if a > 10 then
 8:   result = 1
 9: else
10:   result = 2
11: end
12: # rubocop:emable all
```

```
13:
14: ### 特定の Cop 無効化ここから
15: # rubocop:disable Style/IfUnlessModifier
16:
17: if result == 1
18:  p result
19: end
20:
21: for num in c do
22:   d << "#{num*10}"
23: end
24:
25: p d
26: # rubocop:enable Style/IfUnlessModifier
```

```
$ bundle exec rubocop target_commentout.rb
Inspecting 1 file
C

Offenses:

target_commentout.rb:16:1: C: Layout/IndentationWidth: Use 2 (not 1) spaces for inde
 p result
^
target_commentout.rb:19:1: C: Style/For: Prefer each over for.
for num in c do ...
^^^^^^^^^^^^^^^
target_commentout.rb:20:8: C: Style/UnneededInterpolation: Prefer to_s over string i
  d << "#{num*10}"
       ^^^^^^^^^^
target_commentout.rb:20:14: C: Layout/SpaceAroundOperators: Surrounding space missi
  d << "#{num*10}"
            ^
target_commentout.rb:23:4: C: Layout/TrailingBlankLines: Final newline missing.
p d


1 file inspected, 5 offenses detected
```

「Style/IfUnlessModifier」に関する指摘だけ無くなったことが分かります。

基本的にあまり利用するべき機能ではありませんが、あまり厳密に Rubocop を運用するのも辛いことが多いです。こういった回避策の存在も知っておくと便利です。

第 3 章

Enumerable のメソッドで繰り返し処理を Ruby っぽくする

本章で提案するのは、繰り返し処理における Enumerable モジュールのメソッドの利用です。どんなプログラムでも繰り返し処理は頻繁に使います。そのため Enumerable のメソッドは利用できる機会が多く、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。

本章は以下の流れで Enumerable のメソッドの利用について説明していきます。

- Enumerable の前提知識となる each メソッドについて説明
- each メソッド利用時の課題
- Enumerable のメソッドによる課題の解決
- よく使う Enumerable メソッドの解説

3.1 each メソッドについて

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、その中で利用頻度が高く、まず押さえておくべきなのは each メソッドです。each は配列やハッシュ等のオブジェクトの要素の数だけブロックの中身をします。まず以下に簡単なサンプルコードを示します。

▼リスト 3.1　each のサンプルコード 1

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

#=> 11
#   12
#   13
#   14
#   15
```

each メソッドでは意識しておいてほしい事が 2 点あります。

ブロック引数とブロック内の処理

ブロック引数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任意の処理を行います。上のコードであればブロック引数は「|item|」です。ブロック内では item に 10 を加えたものを p メソッドで出力しています。

Ruby における反復処理はこのように、「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、各要素に対して 1 つずつ適用する形が好まれます[*1]。each はその考えを実現できる最も基本的なメソッドです。

戻り値

each メソッドの戻り値はレシーバであり、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。上のコードであれば each メソッドの戻り値は「list」です。「p item + 10」の結果は each メソッドの戻り値には何も影響を与えません。

言い換えれば、each メソッドのブロック内の処理結果は、そのままだと each メソッド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に値を保存する必要があります。

[*1] この考え方は一般的にはイテレータと呼ばれており他の言語にも存在しますが、Ruby では特に好まれている印象です

▼リスト 3.2　each のサンプルコード 2

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

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

#=> [11, 12, 13]
```

3.1.1　each メソッド利用時の課題

each メソッドは 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が、利用シーンによっては課題もあります。each メソッドでは「オブジェクトの各要素に適用したい処理」と、「処理の結果をどう扱うか？」をブロック内に一緒に記載する必要があります。

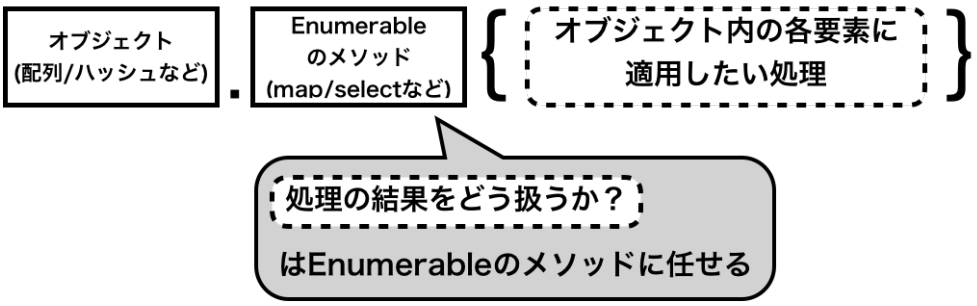
例として リスト 3.2 のコードを用いて説明します。「オブジェクトの各要素に適用したい処理」に該当するコードは `item + 10` で、「結果をどうまとめるかの処理」に該当するコードは `result <<`です。実際の処理は `result << item + 10` であり、ブロック内で一緒に記載されています。

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対して任意の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。加えて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、多くの Ruby エンジニアは避けます。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。

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 のメソッドに任せて、自分は「オブジェクトの各要素に適用したい処理」だけをブロックとしてメソッドに渡します (図 3.1)。



▲図 3.1　Enumerable のメソッドの利用イメージ

3.2.1　each を Enumerable のメソッドに置き換える

実際に リスト 3.2 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 メソッドを使います。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ます。

▼リスト 3.3　map のサンプルコード

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

p list.map { |item| item + 10 }
# => [11, 12, 13]
```

まずブロック内には item + 10 ＝「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、結果をどう扱うか？ は map の中に隠蔽されています。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すことから、処理の結果を格納する変数が必要なくなったためです。each/map のそれぞれのコードを並べて比較してみます。

▼リスト 3.4　each と map の比較

```
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 map を使った処理は、「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、という構造になっていることが分かります。

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 を使って作成できます[*2]。

▼リスト 3.5　each による map メソッドの再実装

```
 1: module MyEnumerable
 2:   def my_map
 3:     array = []
 4:     each do |n|
 5:       array << yield(n)
 6:     end
 7:     array
 8:   end
 9: end
10:
11: class Array
12:   include MyEnumerable
13: end
14:
15: list = (1..3).to_a
16:
17: p(list.my_map { |item| item + 10 })
```


再実装することで「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分かると思います。

「処理の結果をどうまとめるか?」

リスト 3.5 の 1 行目、`MyEnumerable` の `def my_map` に注目してください。リスト 3.2 の構造とほぼ一緒です。異なるのは `def my_map` が受け取ったブロック内の処理を、`yield(n)` を使って展開していることですが、この点が重要になります。`def my_map` から見ると、`yield(n)` が何をしているかは分かりません。`yield(n)` の戻り値を利用しているだけです。行なっているのは

1. `yield(n)` の戻り値 (＝ブロックの戻り値) を array に入れる

[*2] 実際の map とは同じものではありませんが、最低限同じ役割を果たすメソッドとして作成します

2. '1.'をオブジェクトの要素だけ繰り返す
3. 戻り値として array を返す

ですが、この処理は「処理の結果をどうまとめるか？」を実現しています。

「オブジェクト内の各要素に適用したい処理」

リスト 3.5 の 17 行目に注目してください。ここでは my_map を呼び出していますが、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。また「処理の結果をどうまとめるか？」が map の中に隠蔽され、簡潔なコードが書けるようになります。

3.3 良く使う便利な Enumerable のメソッド

これまでの説明の通り、Enumerable を利用することで繰り返し処理を「Ruby っぽい」コードで書けます。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。

そこで Enumerable のメソッドを、個人的に利用頻度が高そうと思った順にいくつか紹介します。

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。(each での再実装は、サンプルコード上で Enumerable のメソッドを置き換える事が可能という範囲で作成しています。実際のメソッドとは異なりますのでご注意ください。気になる人は実際の Ruby のソースコードを確認してみてください。)

3.3.1 map, collect 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。collect メソッドは map メソッドの別名です。別名と言いつつ collect 派の人もたくさんいらっしゃいます。どちらが多数派なのかは不明ですが、私は map 派ですので本書では map で説明をしています。

map は Enumerable のメソッドの中で最もよく使われています。だいたいこれで何とかなります。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。

以下のコードはリスト 3.3 とリスト 3.7 の再掲となります。

▼リスト 3.6　map のサンプルコード (再掲)

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

p list.map { |item| item + 10 }
# => [11, 12, 13]
```

▼リスト 3.7　each による map メソッドの再実装 (再掲)

```
 1: module MyEnumerable
 2:   def my_map
 3:     array = []
 4:     each do |n|
 5:       array << yield(n)
 6:     end
 7:     array
 8:   end
 9: end
10:
11: class Array
12:   include MyEnumerable
13: end
14:
15: list = (1..3).to_a
16:
17: p(list.my_map { |item| item + 10 })
```


3.3.2　select, find_all, filter 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, filter は find_all の別名です。私は select 派なので以降の説明は select で行います。

このメソッドもよく使う印象です。配列やハッシュから条件に合う要素だけ取り出したい時などに利用できます。

▼リスト 3.8　select, find_all メソッドのサンプルコード

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

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

#=> [2, 4, 6, 8, 10]
```

▼リスト 3.9　each による select メソッドの再実装

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

class Array
  include MyEnumerable
end

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

3.3.3 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。

select, find_all の逆の動作です。if に対する unless と同じような位置付けです。適切に使えるとちょっとだけカッコいい気がします。

▼リスト 3.10　reject メソッドのサンプルコード

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

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

#=> [1, 2, 4, 5, 7, 8, 10]
```

3.3.4 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。

Ruby on Rails の環境では、Rails の find メソッドと被るので detect を利用した方が良いかもしれません。

▼リスト 3.11　find, detect メソッドのサンプルコード

```
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
```

3.3.5 include?メソッド

include?メソッド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。

▼リスト 3.12　include?メソッドのサンプルコード

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

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

#=> true
#   false
```


3.3.6 inject/reduce メソッド

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。reduce メソッは inject メソッドの別名です。inject はパッと見何をしているか分かりにくい面がありますが、使ってみると便利で好きな人は好き！ という感じのメソッドです。

可読性を気にする方もいらっしゃいますので、チーム開発の際には利用していいか確認したほうが良いかもしれません。

動作はコードをみてもらった方が早いです。こちらは each での再実装も記載しましたので、併せて確認して見てください。やっていることは案外単純です。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。

▼リスト 3.13　inject, reduce メソッドのサンプルコード

```
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
```

▼リスト 3.14　each による inject メソッドの再実装

```
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}
```


3.3.7　partition メソッド

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。

たまに使うので覚えておくと便利な印象です。

▼リスト 3.15　partion メソッドのサンプルコード

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

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

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

3.3.8 group_by メソッド

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。each での再実装も記載しました。確認してもらうと何をしているかよく分かると思います。

使わないように見えてたまに欲しくなります。これも覚えているととても便利です。

▼リスト 3.16　group_by のサンプルコード

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

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

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

▼リスト 3.17　each による group_by メソッドの再実装

```
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 }
```

3.3.9 max, min, minmax メソッド

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。

使いそうで案外使わない気がします。

▼リスト 3.18　max, min, minmax メソッドのサンプルコード

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

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

#=> 10
#   1
#   [1, 10]
```

3.3.10 max_by, min_by, minmax_by メソッド

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 メソッド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 メソッドとの違いです。

▼リスト 3.19　max_by, min_by, minmax_by メソッドのサンプルコード

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

p list.max_by { |item| item.size }
p list.min_by { |item| item.size }
p list.minmax_by { |item| item.size }

#=> "three"
#   "one"
#   ["one", "three"]
```


3.3.11 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。

これも使いそうで案外使わない印象です。ただ any?は割と使います。

▼リスト 3.20　all?, any?, none?メソッドのサンプルコード

```
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
```

3.3.12 reverse_each メソッド

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「Ruby っぽい」です。

▼リスト 3.21　reverse_each メソッドのサンプルコード

```
list = (1..5)

list.reverse_each { |item| p item }

#=> 5
#   4
#   3
#   2
#   1
```


3.3.13 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。

▼リスト 3.22　each_with_index のサンプルコード

```
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"
```

3.3.14 sort_by メソッド

sort_by メソッドは、ブロックを使ってオブジェクトの要素をソートした配列を返します。オブジェクトの要素にブロック内の処理を適用し、ブロックの戻り値を<=>演算子で比較して、小さい順に要素を並べます。

ソートの基準をこちらで指定できるので、必要な時はすごく便利です。こういったメソッドが標準で組み込まれているあたりに Ruby の特色が表れています。

▼リスト 3.23　sort_by のサンプルコード

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

list.sort_by { |item| item.size }

# => ["ten", "two", "six", "one", "five", "nine", "four", "seven", "eight", "three"]
```

3.3.15 cycle メソッド

cycle メソッドはオブジェクトの要素の分だけブロックの中身を実施し、それを cycle メソッドに渡した引数の分繰り返します。簡単に言うと、each を複数回繰り返すような感じです。なお引数を省略すると無限回繰り返します。

面白いメソッドなので紹介しましたが、なかなか使いどころが難しいかもしれません。

▼リスト 3.24　cycle のサンプルコード 1

```
list = (1..3).to_a
list.cycle(3) { |item| p item + 10 }

#=> 11
#   12
#   13
#   11
#   12
#   13
#   11
#   12
#   13
```

第 4 章

Ruby Sliver を受けて Ruby の全体像を掴む

前章までは特定のツールやクラスの使い方に関して説明してきましたが、本章では少し視点を変えて、Ruby の言語仕様全体を理解するための手段としての Ruby Silver 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 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

本章はこれまで、Rubocop & Enumerable といった利用頻度の高い実践的な内容を紹介してきましたが、ある意味偏った内容ではありました。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、というのが Ruby Silver をオススメする理由です。「基本的」と言ってももちろん入門書よりは少し上の内容で、組込ライブラリの使い方等が主な出題内容になります。また合格点の明確なテスト形式なので、合格さえすれば一定以上の成果が保証される、というのもミソです。

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ました。

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ったりと、ボディーブローのように効いてきた実感があります。

もしかすると実務経験を半年〜1 年以上積んだ方は、既に知っていることが多いためあまりメリットは享受できないかもしれませんが、Ruby 初級者であれば、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 RubySilver の特徴や勉強方法、当日の注意点などを説明します。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

最初に試験要項 (詳細は以下 URL) を参照しながら、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。

本番では間違いなく時間が余ると思いますので、この点を心配する必要はありません。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。そのレベルの問題なら (ちゃんと勉強していれば)20 分程度で解けるでしょう。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 割は問題集とほぼ同じ内容ですが、それだけだと合格にはちょっと届かないかな？ というラインです。学習の際は過去問丸暗記ではなく、理由や背景含めて自分なりに掘り下げていくことが求められます。

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。

試験はパソコン上で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 Ruby2.6.4 が最新ですが、試験の対象バージョンは 2.1.x です。問題集の回答に違和感を感じ、調査してみたら最新バージョンの Ruby では動作が異なった、ということが起こり得ます。

個人的にはこの点で大きく躓く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 (Ruby Association のサイトより)。実際の試験では特に組込ライブラリに関する問題が多いため、重点的に学習する必要があります。学習方法の詳細は後ほど説明します。

4
5

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 学習時間・学習期間の目安

あくまで私の場合ですが、学習時間・学習期間は以下の形でした。

- 学習期間：約 2 週間

- 学習時間：約 30 時間

個人差の大きい部分だと思いますが、長くても 1 ヶ月程度時間を確保できれば十分合格に届くはずです。

なお学習開始時点における私の経歴はおおよそ以下の通りです。ご自身の学習期間を見積もる際の参考にしていただければ幸いです。

- 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。

4.4 学習方法

以降私が実際に Ruby Silver を取得した際に実施した学習方法を紹介します。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ますので、その点ご了承ください。

4.4.1 とりあえず結論

学習方法の概要を列挙すると以下のような形になります。以降それぞれを説明していきます。

- 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 1. 合格記事をさら〜っと眺める

まずは Ruby Silver 自体の大まかな傾向を掴むために「Ruby Silver 勉強法」あたりでググっていくつか記事を眺めてみることをおすすめします。複数の記事を見ていると、勉

強方法やら出題内容等で共通した項目がうっすらと見えてくると思います。

例えば以下のような内容です。

- 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

この段階であまり細かく理解する必要はありません。適当に Google 検索結果の上から 5,6 個記事を直感でピックアップして、流し読みするくらいで良いと思います。

4.4.3 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でしょう。ソシャゲのマラソンみたいなノリでひたすら走るべし。

問題集も 1 つだけではなく、色々な問題集を取っ替え引っ替えするのがオススメです。複数の問題集をこなしているうちに、共通点や傾向が自然と理解できるようになるはずです。私が実際に使った問題集は以下の 3 つです。特におすすめなのはwスマホでも簡単にできる「REx」です。やる気が出ない時も寝っ転がりながらダラダラとこなしたりしていました。

公式模擬問題集

pdf と見せかけてどこかの github リポジトリに飛びます。以下の URL からアクセスできます。

https://www.ruby.or.jp/assets/images/ja/certification/examination/exam_prep_jp.pdf

問題数はテスト 1 回分です。個人的には難易度は一番やさしいと感じました。最初に取り組んでみると良いと思います。

REx

個人的には一番オススメの問題集です。以下の URL からアクセスできます。

https://www.ruby.or.jp/ja/certification/examination/rex

WEB サービスの形をとっておりスマホから簡単に取り組めます。また受ける度に問題の内容や並びが変わるのでなかなか実践的です。おおよそ 8 割が共通の問題で残りの 2 割が入れ替わるような形です。もちろん以前の問題を再び受け直すことも可能です。解説もまぁまぁ書かれています。恐らく一番使いました。ただ github アカウントとの連携が必

要ですので、その点だけご注意ください。

[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(amazon のリンク)

これだけ普通の書籍です。大きな本屋じゃないと入手できないかもしれません。amazon 等を使えば確実に入手できます。最後に紹介していますが、Ruby Silver の対策本といえばまずコレ！ と言われるくらい有名な書籍です。この手の本の常として 3600 円と結構なお値段がしますが、十分その価値はあると思います。でもやっぱり高いので、借りるなり、会社にねだるなり、メルカリを使ってみるなり工夫した方が良いかもしれません。

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、試験範囲の知識に関するリファレンスとしても活用できます。特に 3,4,5 章あたりは、頻出の重要な内容が丁寧に説明されているので、一通り読んでおいても良いと思います。ただいったん問題集を解いた上で、不明な点に絞って読んだ方がより効率的でしょう。

4.4.4 3. 各問題集で 9 割以上正答＆理由も説明できるようにする

次に 2 つ以上の問題集を 9 割以上正当できる＆理由も説明できるようします。その際、理解するための手段は色々な方法を試す方が良いです。

- 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/pry 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、本番で想定外の問題が出ても対応できるようになります。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、といった様にちょっとした寄り道をしてみるのが良いでしょう。

逆に問題集の解答を丸暗記しただけだと、本番で想定外の問題に出くわした時に足元をすくわれる可能性が高くなる、と思います。

4.4.5 4. 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ですが、残りは妙にひねった所のある、いやらしい問題が出てきます。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のでオススメです。

私が実際に直前に確認して助かったのは以下 2 つの記事です。おそらくそのおかげで多分 3 問くらいは得しました。受験前にチラッと目を通しておくことをオススメします。

- 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 その他の注意点

4.5.1 申し込みについて

RubySilver 申し込みの際は以下 2 つを注意してください。

申し込みはプロメトリック社のサイトから

Ruby Silver の申し込みはプロメトリック社のサイトから行います。(http://it.prometric-jp.com/reserve/index.html)

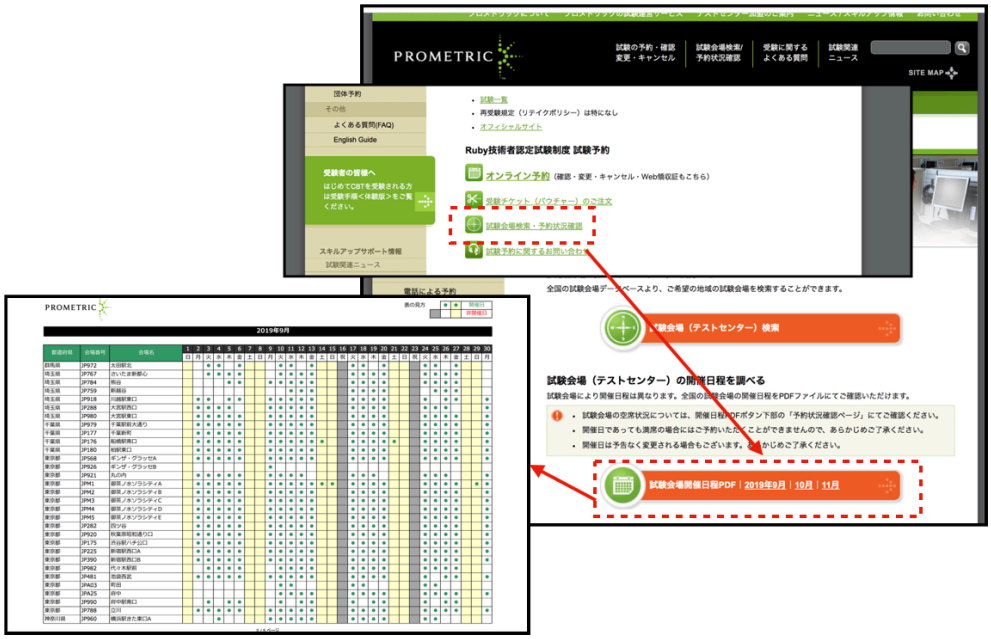
「Ruby Silver 申し込み」あたりでググると、「Ruby 技術者認定試験制度| プロメトリック」のリンクが 2 番目か 3 番目に表示されますのでそこからアクセスするのが簡単です。プロメトリック社のサイトを開いた後は、ページの下側にある「オンライン予約」から申し込みができます。(Ruby Silver/Gold 共通の入口です。手続きを進めていただくと途中で RubSilver を選択する箇所が出てきます)

▲図 4.1　Google 検索結果とプロメトリック社のサイト

日程＆会場は早めにチェックする

いつどの会場で試験を受けるかのチェックは早めに行いましょう。都市部の会場では平日ほぼ毎日試験が開催されていますが、土日の開催はかなり少なく、会場によっては平日を含めても月に数回程度しか開催しない場合もあります。受験日の候補がかなり限定されることも十分に考えられますので、日程＆会場のチェックは早めに行い、ついでに申し込みもしてしまうことをオススメします。

なお会場ごとの試験開催日程は、先ほど紹介した「オンライン予約」のすぐ下にある「試験会場検索・予約状況確認」から参照できます。



▲図 4.2　試験会場検索・予約状況確認

4.5.2　当日の注意点

当日注意して欲しいのは「本人確認書類が 2 つ必要なこと」です。

試験受付の際に 2 種類の本人確認書類を提出する必要があります。「運転免許証」とか「パスポート」とかあればいいんじゃないの？ と思っても、もう 1 種類、本人確認書類が必要です。

これを忘れると本当に試験が受けられません。実際私の知り合いで本人確認書類の準備忘れで試験を受けられなかった人がいます。その場合返金もしてくれません…注意してください。

具体的にどんな種類の本人確認書類が必要か？ はプロメトリック社のサイトをご確認ください。(参考に該当ページの画像を貼っておきます)


IT系資格試験一覧
試験予約前
試験予約前の確認事項
支払方法について
試験会場検索/予約状況確認
試験予約
オンライン予約
電話による予約
特別な受験環境による予約
試験予約後
予約確認 / 変更 / キャンセル
試験当日
必要な本人確認書類
試験後
Web領収証について
法人様向けサービス
受験チケット(バウチャー)
団体予約
その他
よくある質問(FAQ)
English Guide
受験者の皆様へ
はじめてCBTを受験される方は受験手順<体験版>をご覧ください。
スキルアップサポート情報
試験関連ニュース
スクールガイド
スキルアップコラム
本人確認書類について
IT系資格試験の受験に必要な本人確認書類についてご説明します。
受験に必要な本人確認書類
プロメトリック公認テストセンター（APTC）で受験される際は、有効期限内である本人確認書類の原本（コピーおよび電子媒体の本人確認書類は不可）が2点必要です。本人確認書類2点は「グループA」より1点、「グループB1」、または「グループB2」より1点をご提示ください。なお、「予約登録上の氏名」と「本人確認書類の氏名」は必ず同一であることが必要となります。
本人確認書類は2点必要です
グループAより1点＋グループB1またはグループB2より1点
グループA（顔写真付きに限る）より1点
運転免許証
パスポート
社員証
個人番号カード
住民基本台帳カード
グループAに分類されるその他の顔写真付本人確認書類例
※社員証につきましては、注意事項の「社員証については」をご確認ください
両方必要です
グループB1またはグループB2より 1点
グループB1（自署付に限る）
クレジットカード
CREDIT CARD
sample
パスポート
公立図書館・公的な施設利用者カード
グループB1に分類される
その他の顔写真付本人確認書類例
または
グループB2
健康保険証
健康保険カード
住民票（交付日より3ヶ月以内）
グループB2に分類される
その他の顔写真付本人確認書類例

▲図 4.3　本人確認書類

第 5 章

おわりに

著者紹介

第 1 章 ひつじ / @mhidaka

ひつじだよ〜

5
5

Ruby っぽいコード入門

2019 年 4 月 14 日　技術書典 6 版 v1.0.0

著　者　@kappaz 編
編　集　mhidaka
発行所　かっぱ書房

(C) 2017-2019 TechBooster