Ruby, Railsのバージョンを一気に上げた話

BONDOでのケース

概要

プロジェクトによっては、古いまま使い続けられているRails, Rubyのバージョン。

後回しになりがちだが、**隙を見て一気にやっちゃいましょう！**

という話です。

技術的な部分の話は、プロジェクトによって差が大きいので割愛しますが、

そんなに難しく無いよ！というのが伝われば幸いです。

BONDOについて

- ゲーム好きのためのコミュニティサイト
  - ぷよぷよシリーズ、大戦シリーズなど主にセガ様のゲームコミュニティサイトに採用されています。
  - 登録者数30万人くらい（含クローズ済み）
  - かなり昔から運用されている。

- 使用スタックはほぼ Rails のみ
  - アセットは sprocket を使用せず webpack で管理している。

目次

- 古いバージョンを使い続けるリスク
- 各バージョンごとのEOL
- BONDOでのバージョンアップ実例
- やってみてわかった重要なこと
- 簡単なバージョンアップ手順
- ハマったところ

古いバージョンを使い続けるリスク

- **セキュリティリスクが高まる**
  - 新しいバージョンの対策が使えない。
- **インストールできる gem のバージョンが限られる**
  - こちらもセキュリティリスクあり。
  - 使いたい機能が使えなかったりする。
- **後からまとめてバージョンアップは大変**
  - こうやって後回しになっていく…。

各バージョンごとのEOL

EOL: End Of Life サポート終了

2.4	2020-03-31
2.5	2021-04-05
2.6	2022-04-12
2.7	2023-03-31
3.0	2024-03-31

5.1	2019-08-25
5.2	2022-06-01
6.0	2023-06-01
6.1	2024 ??
7.0	???

BONDOでのバージョンアップ実例

Ruby 2.5

↓

Ruby 2.6

↓

Ruby 2.7

↓

Ruby 3.0

Rails 5.1

↓

Rails 5.2

↓

Rails 6.0

↓

Rails 6.1

バージョンアップの際に重要なこと

- **変更を与えるごとにテストを回す**
  - 不具合が起きた変更を明らかにするため。
  - 後から行うと、何が不具合の原因だったのか分からなくなる事多数。

- **テストカバレッジを上げる**
  - バージョンアップによる文法の変更やメソッドの廃止、動作変更に気づくため。
  - テスト実行時に該当するコードを読み込んでいれば、何かしら違いが出てくるはず。
  - バージョンアップ前にDEPRECATION　WARNINGを検知するにも重要。

- **gem のバージョンアップをできる限り同時に行う**
  - ruby, rails のバージョンに依存している物が多いので、一緒にやっておく。

Rails バージョンアップの手順 その1

1. テストを回して全クリアすることを確認
    a. ここでDEPRECATION WARNINGが出ている部分を全て拾っておく。

2. DEPRECATION WARNINGを潰す
    a. 「次のバージョンではこのメソッド消すよ」等のメッセージが書いてある。
    b. メッセージでググれば大抵出てくるが、よく読む。どこから出ているかも重要。

3. rails 以外の gem のバージョンアップ
    a. 基本的に gem のバージョン指定はせず、最新のものを使う。
    b. $ bundle outdatedで現在入っている gem の最新のバージョンを調べる。
    c. 影響の大きそうなバージョンアップは個別で $ bundle update

Rails バージョンアップの手順 その2

4. rails のバージョンを指定して update
    a. マイナーバージョンごとに上げる。（一気に上げない）
    b. マイナーバージョンないでも最新のパッチバージョンに指定する。
    c. 例：6.0.3 → 6.1.7.4

5. $ rails app:update 実行
    a. config ファイルが新しいものに書き換えられる。
    b. 今まであった設定と、新規に追加された設定をうまいこと MIXする。

6. new_framework_defaults.rb の設定
    a. 新しいバージョンの Railsでの新機能の設定をするファイルが生成される。
    b. 新しい機能はコメントアウトを外すことによって設定できる。
    c. 基本的には全部の機能を使いたい。が、例外もある。

Railsバージョンアップでハマりがちなところ 1

new_framework_defaults.rb の設定

`# This setting is not backwards compatible with earlier Rails versions.`（この設定には後方互換性がありません

この警告がある設定は後回しにして、本番環境のでの安定稼働を確認してから設定を有効にした方がよい。

なぜなら他で不具合が発生した時に、元の Railsバージョンに戻すことができなくなってしまうから。

例えば、cookie署名の暗号化方式の変更設定にこの警告が書いてある。

一度新しい暗号化方式で保存された署名は、古いバージョンでの読み込みに対応できなくなってしまうので、元に戻すことができなくなってしまう。

設定自体は有効にしなければ恩恵を受けられないが、安定稼働してから有効かすることが勧められている。

```
# Make Active Record use stable #cache_key alongside new #cache_version method.
# This is needed for recyclable cache keys.
Rails.application.config.active_record.cache_versioning = true

# Use AES-256-GCM authenticated encryption for encrypted cookies.
# Also, embed cookie expiry in signed or encrypted cookies for increased security.
#
# This option is not backwards compatible with earlier Rails versions.
# It's best enabled when your entire app is migrated and stable on 5.2.
#
# Existing cookies will be converted on read then written with the new scheme.
# Rails.application.config.action_dispatch.use_authenticated_cookie_encryption = true

# Use AES-256-GCM authenticated encryption as default cipher for encrypting messages
# instead of AES-256-CBC, when use_authenticated_message_encryption is set to true.
# Rails.application.config.active_support.use_authenticated_message_encryption = true

# Add default protection from forgery to ActionController::Base instead of in
# ApplicationController.
Rails.application.config.action_controller.default_protect_from_forgery = true

# Use SHA-1 instead of MD5 to generate non-sensitive digests, such as the ETag header.
Rails.application.config.active_support.use_sha1_digests = true

# Make `form_with` generate id attributes for any generated HTML tags.
Rails.application.config.action_view.form_with_generates_ids = true
```

Railsバージョンアップでハマりがちなところ 2

gem のバージョンアップ

bundler がうまいことやってくれる場合も多いが、どうしてもgem 同士のバージョン依存によるエラーは発生する。

エラーのバックトレースから、どのgem でエラーが発生しているかを特定し、該当するgem の CHANGELOG や issue を読む。

なにかしら情報があれば、問題のないバージョンを指定して解消できるか試してみる。

情報が少ない場合は、力技でつづつバージョンを戻す作業もアリかも。。

gem のバージョンアップをどのタイミングで行うか？

rails バージョンアップ前？後？

→ できればどっちも行っておく。

バージョンアップ前に行っておけば、バージョンアップ後はails のバージョンに依存するものだけになるので、そんなに面倒ではないはず。。

Ruby バージョンアップの手順

1. テストを回して全クリアすることを確認
    a. ここでDEPRECATION WARNINGが出ている部分を全て拾っておく。
2. DEPRECATION WARNINGを潰す
    a. Rails の時と同じく出てる。全部潰す！
    b. Rails のバージョンアップ時に全部潰せていれば出てこないかも。
3. rails 以外の gem のバージョンアップ
    a. これもRails の時と同じく。
    b. Rails のバージョンアップと連続で行うときは不要かも。
4. ruby のバージョンアップ
    a. ローカル環境であれば rbenv 等で指定
    b. docker環境であれば、使用する ruby イメージを変更する。

Rubyバージョンアップでハマりがちなところ

バージョンごとに文法の違いがある

基本的には変更前のバージョンでDEPRECATION WARNINGが出ているので、しっかり潰していれば大丈夫なはず…！

3.0から採用された位置引数とキーワード引数の分離は、意外と使用箇所が多く、マイグレーションファイルないにも使用箇所があったりして検知するのに厄介だった。

```ruby
# 3.0以降はこれだとダメ
def hoge(**opt)
  p opt
end

hash = {a: 1, b: 2}
hoge(hash)
```

```ruby
# これはOK
def hoge(**opt)
  p opt
end

hoge(a: 1, b: 2)
```

バージョンアップのまとめ

- **共通の作業が多いので、RailsとRubyのバージョンアップは続けてやってしまった方が楽。**

  → gem のバージョンアップの手間が結構省ける。

- **1つバージョンを上げてしまえば細部は異なるものの、手順は変わらず。**

  → 大規模な変更はあんまり無い。

- **後回しにすると大変になるので、時間が少し空いたらやっておくのが良い。**

  → まとめて行うと、変更点が多くなるのでリスク大。