# Rails開発パターン

## Rails Development Patterns

吉岡ひろき
株式会社クレイ
irohiroki@gmail.com

## 概要

Ruby on Rails（以下「Rails」と呼ぶ）を採用したウェブアプリケーション開発には、プログラミング言語Rubyの特性や開発者およびライブラリの豊富さから、他のウェブアプリケーションフレームワークとは異なるノウハウがある。

本稿ではRailsアプリケーション開発の現場から見出された6つのパターンを紹介する。

## はじめに

Railsは2004年にDavid Heinemeier Hansson氏によって生み出され、現在も13人の中心的な開発者と3,000人以上の貢献者によって開発が続いているウェブアプリケーションフレームワークである。2006年からは開発者と利用者のための国際的なカンファンレンス「RailsConf」が毎年開催され、実際のビジネスでもクックパッド [1] やGitHub [2] といった数百万人規模のユーザを持つサービスで採用されている。

このように、Railsは世界規模で利用されているにも関わらず、その開発ノウハウはRails Best Practice [3] やRails3レシピブック [4] のようにベストプラクティスやレシピとして公開されるに留まり、パターン・ランゲージの形をとったものはなかった。

そこで、著者は7人の現役Railsアプリケーション開発者の協力を得て2014年9月からパターン・ランゲージ作成に取り組んでいる。本稿は、そのパターン・ランゲージの中から早期に抽出された6つのパターンを抜粋して紹介するものである。

## パターンの抜粋

### 各パターンの位置付け

パターンはそれぞれ主に適用する時期とレイヤが異なる。パターンと時期、レイヤの関係を図1に示す。

<公式情報>と<Rails Way>は開発の序盤から終盤まで、全レイヤに対して適用できる。<公然のSQL>は全期間でモデル層に適用できる。<レスポンシブテスト>は、中盤以降の全レイヤを対象とする。<コードの節制>は中盤以降に主にコントローラとモデルに対して適用できる。最後に、<レール圏>は開発の終盤に顕在化する問題を扱う。

### 1. 公式情報

#### 状況

Railsの新機能や、よく知らない機能を使ってアプリケーションを実装している。

#### 問題

ウェブを検索して見つけた実装方法を真似ても、期待通りに動作しない。動作させるために更に検索したり試行錯誤したりして、時間を浪費する。

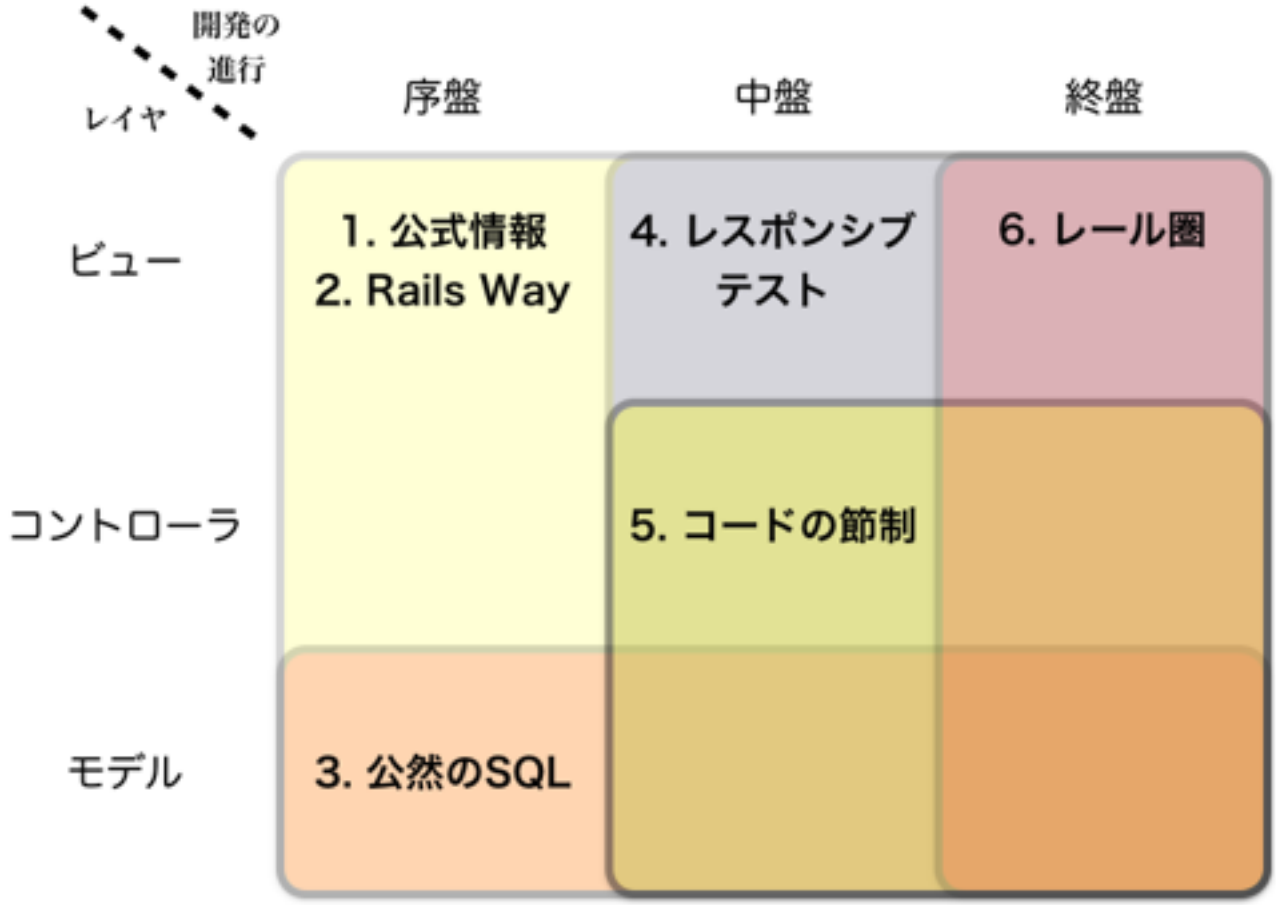


図1. パターンを適用する主な時期とレイヤ

## 問題の原因

Q&AサイトのStack Overflow [5] には「ruby-on-rails」のタグがついた投稿が約18万件あり、PHPのウェブアプリケーションフレームワーク「cakephp」の約2万件、Javaのウェブアプリケーションフレームワーク「struts」の約3千件と比べると、Railsに関する情報がウェブ上に比較的豊富にあることが分かる。

一方、Railsは2008年4月から2014年12月までの間に4つのメジャーバージョンと9つのマイナーバージョンをリリースしており、その度に大小の非互換を生んでいる。

以上のことから、Railsに関してはウェブ検索によって情報は得られるが、開発者のバージョンにおいては必ずしも正しくない状況が発生する。バージョンが一致していても、投稿者の誤解などによって間違った情報が掲載されていることもある。

## フォース

- Railsは互換性を犠牲にすることで積極的に新機能を導入したりコードの保守性を保ったりしている。
- 第三者による情報発信を制限したり、常に正しい情報を求めたりはできない。

## 解決方法

Railsの機能について調べるときは、必ず公式の情報をあたる。

## 適用例

1. 公式ガイド）Rails 3のセキュリティ機能の一つであるattribute_accesibleはRails 4でStrong Parametersに置き換えられたが、ウェブ検索すると2015年1月時点でもattribute_accesibleの使用方法を解説するブログ記事が見つかる。

このように意図せず古い情報を参照しないようにするには、最初に公式のガイド [6] をあたればよい。

2. リファレンス・マニュアル）Railsのリファレンス・マニュアルはウェブ上で公開されている [7] 。また、「http://api.rubyonrails.org/v4.1.0/」のようにURLにバージョン番号を付ければ、指定したバージョンのマニュアルを参照することもできる。

3. リリースノート）バージョンアップによって変更された点は、公式ブログのリリースに関する投稿 [8] に書かれている。

4. ソースコード）以上の文書を読んでも不明な点については、GitHubで公開されているRailsのソースコード [9] を読むとよい。

### 結果

初めて使う機能であっても期待通りに動作する。オプションなどの副次的な機能についても完全な情報が得られ、より適切に実装できる。

## 2. Rails Way

### 状況

開発の序盤に様々な設計上の決定をし、実装を開始した。

### 問題

Railsが想定外の処理を行い、解決のために調査したり余分なコードを追加したりして時間を浪費する。

### 問題の原因

Railsは特定の実装には最適な方法があるという仮定に基いて設計されており、その仮定に沿った実装を推奨している [10] 。Convention over Configurationと呼ばれるソフトウェア設計のポリシーもその一部であり、例えばデータベースの「products」テーブルを扱うクラスの名前は「Product」を前提にする。Railsではこのような「Rails Way」と呼ばれる慣例に従うことでコードの量が最小になるように作られている。

Rails Wayの範囲はデータベース層からプレゼンテーション層まで全域に広がっており、開発者が把握していない部分でRails Wayに反した決定をしてしまうと無駄が発生する。

### フォース

- Rails WayはRailsの開発思想そのものであり、変更できない。
- Rails開発の入門者には開発思想が定着していないばかりでなく、開発思想の存在すら知らないことがある。

### 解決方法

他のフレームワークでは自由に選択できる設計や実装であっても、Rails Wayがないか確認し、可能な限り従う。

### 適用例

1. 命名規則）命名規則はRails Wayの代表である。例えば、クラスの定義を含むファイルのパスはクラス名を元に走査される。逆に、クラス名から推測不能なパスにファイルを置き、明示的にパスを指定しなければLoadErrorが発生する。よって、特に理由がなければ命名規則に従うべきである。

2. カラムの予約語）データベースのカラムに予約語がある。「type」というカラムは、一つのテーブルに複数の型を格納する仕組み（Single Table Inheritance）に使われるため、それ以外の用途では避けなければならない。

3. Assets Pipeline）JavaScriptとスタイルシートは、開発段階では複数のファイルに分けておけるが、運用時には転送量削減のためにAssets Pipelineという機構によって自動的に圧縮、連結される。この機構を知らないと、運用時の環境で発生した問題を解決できないことがある。Assets Pipelineはウェブアプリ

ケーションに必須の機構ではなく最適化に含まれるが、Railsは明示的に外さない限り最適化を行う方針を採っている。
4. コールバック）HTTPリクエストやDBアクセスを契機としたトランザクションの前後のコールバックには典型的な使い方がある。例えば、認証にはトランザクションの前のコールバックを利用する。

その他のRails Wayは、Rails Guides [7] やAgile Web Development with Rails 4 [11] に記載されている。

### 結果

最小のコードで最大の生産性を得られる。また、Rails Wayに沿ったコードは書いた本人以外の開発者にも読みやすく、保守性が高まる。

## 3. 公然のSQL

### 状況

ActiveRecordの機能を使ってデータベースを操作している。

### 問題

意図通りのデータベースクエリを発行する方法が分からずに時間を浪費したり、無意識のうちに高負荷のクエリを発行したりする。

### 問題の原因

ActiveRecordを使うと実際のSQLに比べて短いRubyコードでデータベースクエリを生成できる。また、実際のSQLの語順や、データベースの種類に依存せずに記述できるなどのメリットもある。

しかし、より複雑なクエリを生成するためにRubyコードも複雑になることは避けられない。また、ActiveRecordでは特定のデータベースに固有の最適化手法を利用できない場合もある。

### フォース

- コードの読みやすさや互換性のために、ActiveRecordを通じてSQLを構築するのは望ましい。
- 複数のデータベースの全ての機能を単一のライブラリでサポートすることはできない。

### 解決方法

利用するデータベースの機能を知り、必要に応じて最適なSQLを書けるようにしておく。

### 適用例

1. UNION）ActiveRecordでSQLのUNIONを記述する方法は少なくとも2011年3月から議論されているが、2014年12月の時点でも解決していない [12,13] 。よって、自分で書けるようにしておく必要がある。
2. N+1問題）「N+1問題」は、無駄なデータベースクエリを発生させる典型的な問題である [14] 。例えば、下のようにBookとAuthorと対応するテーブルが定義されているとき、

```
class Author < ActiveRecord::Base
end

class Book < ActiveRecord::Base
  belongs_to :author
end

create_table :authors do |t|
  t.string :name
end

create_table :books do |t|
  t.string :name
  t.references :author, index: true
end
```

次のようなコードは合計11回のクエリを発生する。

```
# コントローラ
@books = Book.last(10)

# ビュー
<% @books.each do |book| %>
  <tr>
    <td><%= book.name %></td>
    <td><%= author.name %></td>
  </tr>
```

```
<% end %>
```

実際に発行されるクエリは下の通り。

```
  SELECT      `books`.*  FROM  `books`
ORDER BY `books`.`id` DESC LIMIT 10
  SELECT   `authors`.* FROM `authors`
WHERE `authors`.`id` = 1 LIMIT 1
  SELECT   `authors`.* FROM `authors`
WHERE `authors`.`id` = 2 LIMIT 1
  …
  SELECT   `authors`.* FROM `authors`
WHERE `authors`.`id` = 9 LIMIT 1
  SELECT   `authors`.* FROM `authors`
WHERE `authors`.`id` = 10 LIMIT 1
```

コントローラのコードを以下のように修正することで、クエリは2回になる。

```
  @books = Book.includes(:author)\
  .last(10)
```

修正後のクエリは以下の通り。

```
  SELECT      `books`.*  FROM  `books`
ORDER BY `books`.`id` DESC LIMIT 10
  SELECT  `authors`.*  FROM  `authors`
WHERE `authors`.`id` IN (10, 9, 8, 7,
6, 5, 4, 3, 2, 1)
```

### 結果

データベースクエリに起因する問題により早く気付けるようになり、リリース後に障害が発生するリスクを下げる。

## 4. レスポンシブテスト

### 状況

アプリケーションの開発が進み、テストコードが増えてきた。

### 問題

テストの実行時間が長くなり、開発者のストレスとなったり、問題の発見が遅れたりする。

### 問題の原因

Railsにはテストコーディングを補助するヘルパーメソッドが用意されていたり、雛形コードのジェネレータによって製品コードと共にテストコードの雛形も生成されるなど、テストコードを書きやすくする仕組みが備わっている。

このように、Railsアプリケーションにおいてテストコードを書くことは推奨されているが、個々のテストケースの価値やテスト実行のコストについてはRails Guidesでも言及されておらず、Rails初心者が実際に問題に遭遇するまでテスト実行時間について検討する機会が提供されていない。

### フォース

- 円滑な開発や問題の早期発見のために、テストコードを書くことは望ましい。
- 全く実行時間のかからないテストコードは存在しない。

### 解決方法

開発の初期からテストコードの費用と効果を評価しながら開発する。

### 適用例

1. インテグレーションテスト）Railsがサポートするテストのうち、ブラウザをエミュレートし、JavaScriptの実行からデータベースアクセスまで全てのレイヤを実行するインテグレーションテストは最も実行時間がかかるため、全ての条件分岐を網羅するテストをインテグレーションテストで書くと、他の種類のテストで書いた場合よりも実行時間が長くなってしまう。インテグレーションテストは、全レイヤを実行しレイヤ間の接続を確認できることを活かし、スモークテストに利用するとよい。
2. ビジネス上の価値の反映）テストケースの価値は、テスト対象のビジネス上の価値に依存する。例えば、ビジネスの成立に必須ではない代替手段のある機能は、カバー率を上げて実行時間をかける価値は低い。逆に、ビジ

ネスの成立に不可欠な機能のカバー率を上げるべきである。
3. モックとスタブの利用）どのレイヤのテストを書く場合でも、テスト対象を明確にし、対象外のコードを実行しないことで実行時間を最短にできる。実行するコードを最小にするためには、モックやスタブの利用を検討する。

例えば、ルーティングをテストするコードでデータベースにアクセスしてはいけない。永続化層はスタブにし、実際の入出力処理の実行を回避すべきである。

### 結果

アプリケーションを拡張してもテストの実行時間は適切な範囲に保たれる。リグレッションは早期に発見され、開発者は安心して開発を進められる。

## 5. コードの節制

### 状況

ある程度の期間に渡って機能の拡張や改修を繰り返している。

### 問題

モデル層とコントローラ層のクラスが肥大化し、改修や保守が困難になる。

### 問題の原因

Railsにはコードの雛形を生成する機能が備わっており、その機能を使うとモデルとコントローラ、ビューの各層のファイルができる。ビュー層のファイルはHTMLテンプレートなので、トランザクションの処理を書くには適さない。逆に、全てのトランザクションは生成されたモデルとコントローラに書いて実現できる。実際に、David Heinemeier Hansson が著者として参加している入門書Agile Web Development with Rails 4[11]でもモデルとコントローラにコードを書いている。

ところが、現実のソフトウェアはより複雑な場合があり、アプリケーションによって適切なアーキテクチャは異なり [15] 、モデルとコントローラに書くことが最適とは限らない。

### フォース

- 機能の拡張や改修を繰り返すことでコードは増加する。
- 世の中の全てのアーキテクチャをサポートすることはできない。

### 解決方法

モデルとコントローラの責務を明確にし、その範囲を越える処理は、より適切なクラスやレイヤに割り当てる。

### 適用例

1. サービスレイヤ）例えば、複数のモデル（仮にAとBとする）が関与する処理について考える。この処理はAの責務ともBの責務とも言い切れない場合がある。Aの責務としてAからBを参照すれば、AとBが密結合し、保守性が低下するし、逆もまた同様である。だからといって、この処理をコントローラ層に置くと、コントローラ層が肥大化する。このように、複数のモデルが関わる処理はサービスと呼ばれるレイヤに切り出す解決方法がある [16] 。
2. ActiveSupport::Concern）Rails 4.0ではモデルまたはコントローラクラスの機能の一部をモジュールに切り出す仕組みであるActiveSupport::Concernと、専用のディレクトリが導入された。ActiveSupport::Concernを使うことで、モデルまたはコントローラ本来の責務ではない処理や、複数のクラスに共通する処理を独立したモジュールに書ける。
3. Single Responsibility Principle）Single Responsibility Principle（以下SRP）はオブジェク指向プログラミングの原則であり、一

つのクラスが一つの責務を持つことを表す。SRPの侵害はモデルやコントローラの肥大化の原因の一つである。特に、SRPに則ったモデルのリファクタリング方法については、Bryan HelmkampがRubyKaigi 2013で発表している [17] 。

### 結果

アプリケーションの拡張が続いてもクラスの責務は明確で読みやすい。コードは継続的な保守に耐える。

## 6. レール圏

### 状況

計画の大半の機能を実装し終え、ユーザビリティテストや負荷テストを開始した。

### 問題

機能は実現できたが、操作感や表現力などの非機能要件を満たせない。

### 問題の原因

Railsは「Creating a weblog in 15 minutes」という動画 [18] で宣伝するなど、アイデアを素早く形にできることを強みの一つとしている。また、ウェブアプリケーションフレームワークであることにより、ユーザにはブラウザだけで使える利便性を、開発者にはHTMLとCSSだけ知っていればプレゼンテーションできる作りやすさを提供している。このように、Railsには開発者に「簡単にできる」と思わせる要素がある。

しかし、ウェブアプリケーションはユーザインターフェースとデータがネットワークを介して分散しているため、その両方が同一のデバイス上にあるネイティブアプリケーションに比べて応答性能が劣る。また、ブラウザの表現力はHTMLの仕様によって制限されているため、デバイスの性能を最大限に出せるとは限らない。

つまり、Railsで機能を実現できたとしても、アプリケーションに必要な性能が出るとは限らない。また、アプリケーション開発の初期段階では十分な性能が出ていても、機能拡張などの改修を重ねることで性能が出なくなる場合もある。

### フォース

- Railsのアーキテクチャを変えることはできない。
- アプリケーションの要件も変えられない。

### 解決方法

ウェブアプリケーションが適したケースにだけRailsを採用する。また、アプリケーションを拡張する際はRailsで実現する範囲を適切に制限し、その範囲を越える部分には他のソフトウェアを利用する。

### 適用例

1. スマートフォン）スマートフォンやタブレットといったモバイルデバイスは同世代のパソコンに比べてJavaScriptやスタイルシートの処理に時間がかかるため、ユーザ体験を損ねる一因になる。特に、スマートフォンは通信環境が不安定なので、サーバからファイルを取り寄せる必要のあるウェブアプリケーションは不利になる。
2. コンシュマー向けアプリケーション）コンシュマー向けのアプリケーションはビジネスなどの特定用途向けのものに比べて高い表現力を求められるケースが多い。

つまり、モバイルデバイスやコンシュマーを対象としたアプリケーションを開発する際には、応答性能や表現力の低さと開発しやすさを検討して採用するフレームワークを決定する。

Railsを採用して十分な操作性や表現力を実現できるケースもあるが、相応のコストを見積もっておくべきである。

### 結果

様々なアプリケーションを開発するなかで、開発しやすさや実装の速さといったRailsのメリットだけを享受し、リリース直前に非機能要件を満たせなくなるなどのトラブルを回避できる。

## まとめ

Railsアプリケーション開発を対象とした作成中のパターン・ランゲージの中から、6つのパターンを紹介した。

今後は、本稿で述べなかったパターンを含めたパターン・ランゲージ全体を完成させ、利用者にとってより使いやすい形式で公開すると共に、より多くのRailsアプリケーション開発者に意見を求めて改善していく予定である。

## 謝辞

パターンの案を提供し、アイデアの本質を探求する議論に参加してくれた赤松祐希氏、浅海拓来氏、朝倉寛史氏、櫻井達生氏、白土慧氏、三村益隆氏、諸橋恭介氏に心から感謝する。また、シェパードとして構成から記法まで丁寧にレビューしてくださった坂本一憲博士にも感謝申し上げる。

## リファレンス

[1] クックパッド株式会社. クックパッド. http://cookpad.com/

[2] GitHub, Inc. GitHub. https://github.com/

[3] Xinmin Labs. Rails Best Practices. http://rails-bestpractices.com/

[4] 高橋征義, 松田明, 諸橋恭介. 2011. Rails3 レシピブック. ソフトバンククリエイティブ.

[5] stack exchange inc. Stack Overflow. http://stackoverflow.com/

[6] Ryan Bigg, 他. Rails Guides. http://guides.rubyonrails.org/

[7] David Heinemeier Hansson, 他. Ruby on Rails API. http://api.rubyonrails.org/

[8] David Heinemeier Hansson, 他. Riding Rails. http://weblog.rubyonrails.org/releases/

[9] David Heinemeier Hansson, 他. rails/rails. https://github.com/rails/rails

[10] Ryan Bigg, 他. Getting Started with Rails. http://guides.rubyonrails.org/getting_started.html

[11] Sam Ruby, Dave Thomas, David Heinemeier Hansson. 2014. Agile Web Development with Rails 4. The Pragmatic Bookshelf.

[12] Lighthouse Import, 他. ActiveRecord UNION left out. https://github.com/rails/rails/issues/939

[13] James Sanders, 他. Make union/except/intersect chainable as SelectManager. https://github.com/rails/arel/pull/320

[14] Ryan Bigg, 他. Eager Loading Associations. http://guides.rubyonrails.org/active_record_querying.html#eager-loading-associations

[15] マーチン・ファウラー. 2005. エンタープライズ アプリケーションアーキテクチャパターン. 翔泳社.

[16] エリック・エヴァンス. 2011. エリック・エヴァンスのドメイン駆動設計. 翔泳社.

[17] Bryan Helmkamp. Refactoring Fat Models with Patterns. http://rubykaigi.org/2013/talk/S32

[18] David Heinemeier Hansson, Creating a weblog in 15 minutes. http://

showmedo.com/videotutorials/video?name=rubyWeblogIn15Mins