Special Contents

『現場で使える Ruby on Rails 5 速習実践ガイド』特典PDF

Special 1

トランザクション処理

本書Chapter4の「4-4-3」として掲載している内容をより詳しく解説したものです。

アプリケーションを開発していく際に、複数の処理をまとめて一つの処理として扱いたいケースを取り扱うことがあります。例えば銀行の入出金処理です。Aさんの口座からBさんの口座にお金を移す、という処理を行いたい場合、下記の2つの処理を行うことになります。

1. Aさんの口座からお金を引き出す
2. Bさんの口座に引き出したお金を入金する

さて、1の処理は成功して、2の処理がエラーで失敗してしまった場合、Aさんの口座からはお金が引かれたのに、Bさんの口座には入金されず、引かれた分のお金が失われることになてしまいます。これを防ぐには、1,2 の処理を「すべての処理が成功」か「一つでも失敗した処理があればすべて失敗」のどちらかとして扱う必要があります。このように複数の処理を一つの処理にまとめて行う方法をトランザクション処理といいます。

RailsではActiveRecord::Base.transactionを使うことでトランザクション処理を実現できます。下記は例として、Model1とModel2というモデルのレコードをトランザクション処理で保存するコードです。

```
model1 = Model1.new
model2 = Model2.new

ActiveRecord::Base.transaction do
  model1.save!
  model2.save!
end
```

モデルでは、クラス・インスタンスのどちらからもtransactionメソッドを利用できます。1つのモデルクラスに関するTransaction処理を行うのであれば、この方法を利用することが多いでしょう。

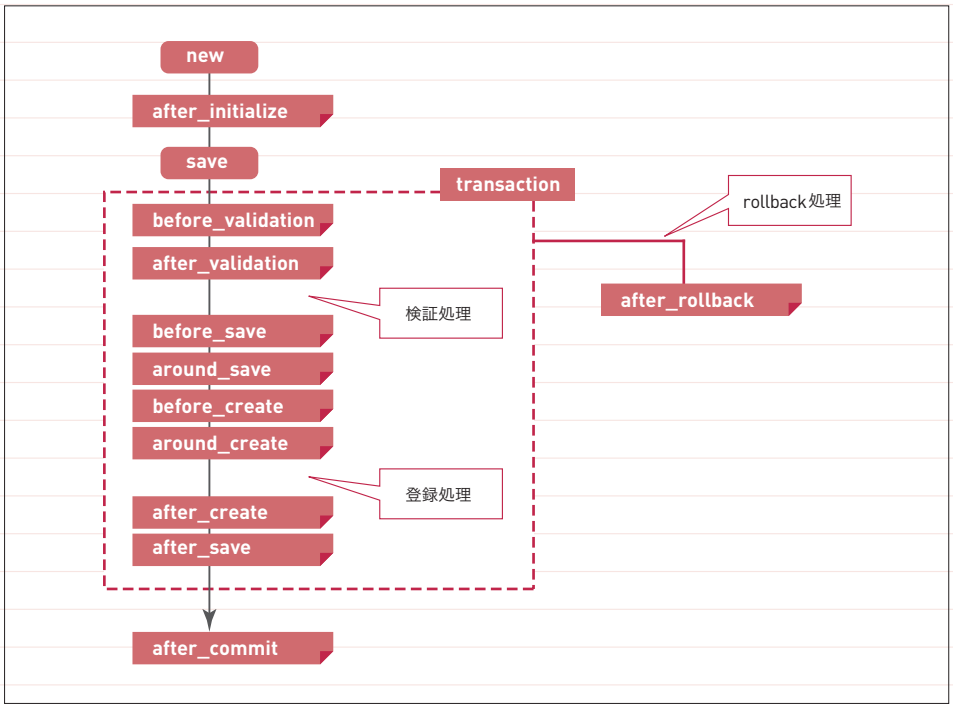
```
Model1.transaction do
  model1.save!
  model2.save!
end

model1.transaction do
  model1.save!
  model2.save!
end
```

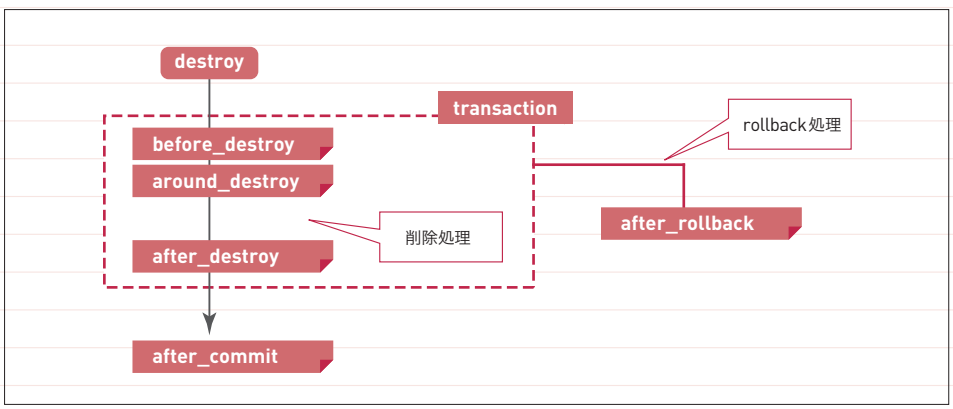
transaction内では、例外が発生するとそれ以前の処理がロールバック（取消）されます。上記のコード例だと、model2.save!で例外が発生した場合、model1.save!で保存された内容はロールバックされ、save前の状態に戻ります。

ここで、save!でなくsaveを使っていた場合は失敗時に例外が出ないため、戻り値を自分でチェックして例外を出すといったことをしないとロールバックがされないことに注意してください。transaction内ではsave!やcreate!などの例外が発生するメソッドを使うのが便利です。

本書「4-4-1」「4-4-2」で紹介したコールバックもトランザクションと深い関わりがあります。before_saveなどのコールバックを定義した場合、Railsがsave時に自動的にトランザクション処理に含んで実行してくれます。以下に登録時と削除時のトランザクションの範囲とコールバックを示します。



図S-1-1 登録処理



図S-1-2 削除処理

トランザクション内で実行されるコールバックで例外が発生した場合は、ロールバック処理が実行され、after_rollbackが呼ばれます※1。その場合、例外が発生した以降のコールバックは実行されないことに注意しましょう。

また、トランザクション内で実行されるコールバック内で、データベースとは関係のない外部のシステムとのやり取り（ファイルの作成や削除など）を行いたい場合も注意が必要です。トランザクション処理が失敗してロールバックされた場合、外部システムで実行された内容についてはロールバックされないため、モデルの一貫性が損なわれてしまう可能性があります。このような場合はデータベースにCOMMITされた後（＝トランザクション処理が成功している）のコールバックであるafter_commitを使うようにしましょう。

トランザクション内で実行されるコールバックで外部システムを呼ぶ際に気をつけたい点がもう一つあります。それは、外部システムの不調によって長時間待たされてしまう可能性の有無です。たとえば、呼び出し先の外部システムがWebアプリケーションであるような場合には、長時間待たされる可能性を考慮すべきです。トランザクション内部で長時間の待ちが発生すると、データベースが重くなるなどの障害につながる恐れがあります。このような場合はTimeout.timeoutを使って、一定以上の時間がかかったら失敗になるような処理を仕込んでおくことをおすすめします。

このように、モデルの登録・更新・削除はコールバックを含めてトランザクション処理がかけられています。そこで、何らかのトランザクション処理を行いたい場合には、まず適切なコールバックに処理を書くことで自動的にトランザクション処理を実現できないか検討してみましょう。そして、それが難しい場合にだけ、前述のtransactionメソッドを使って自分で処理を書くようにすると良いでしょう。

※1　throw :abortで明示的にコールバックを中止した場合も同様です

Special 2

論理削除にする

本書Chapter7の「7-6 CSV形式のファイルのインポート/エクスポート」の後に入る予定だった内容です。

S-2-1 論理削除とは

今まで一口に「削除」と書いてきましたが、データの削除に関しては「物理削除」と「論理削除」の2つの概念があります。それぞれ以下のような違いがあります。

- 物理削除 …… 実際にDELETE文を発行して、データベースからレコードを削除する（Railsの基本の削除処理）
- 論理削除 …… テーブルに削除用フラグを追加して、そのフラグにより画面上で表示・非表示を切り替える

論理削除は、一度削除したデータを復元可能にしたり、後々分析などに利用したいといったニーズがある場合に用いられます。ただし、扱いが煩雑になったりデータが肥大化するデメリットがあるため、安易に導入せず、本当に必要な場合にだけ用いるようにしましょう。

本節では、タスクの削除を論理削除に変更するやり方を紹介します。

S-2-2 paranoia

Railsでは、たとえばparanoiaというgemを使って論理削除が実現できます。以下に利用方法を示します。

S-2-2-1 paranoiaのインストール

Gemfileに以下の一文を追加してbundleコマンドを実行してください。

Gemfile（アプリケーションフォルダ直下）

```
gem 'paranoia', '~> 2.2'
```

```
$ bundle
```

S-2-2-1 paranoiaのインストール

論理削除では、あるレコードが削除されているかの判断をするためのカラムが必要です。paranoiaでは、削除日時を示すdeleted_atというカラム（カラム名は設定で変更可能）が対象モデルにあることを想定し、ここに実際に日時が入っているかどうかで削除済みかどうかを判断します。

そこで、tasksテーブルにdeleted_atカラムを追加しましょう。次のようにコマンドでマイグレーションファイルを作成し、マイグレーションを実行してください。

```
$ bin/rails g migration AddDeletedAtToTasks deleted_at:datetime:index
Running via Spring preloader in process 16634
      invoke active_record
      create db/migrate/XXXXXXXXXXXX_add_deleted_at_to_tasks.rb
```

```
$ bin/rails db:migrate
```

続いてapp/models/task.rbを以下のように修正し(❶)、論理削除に対応させます。

app/models/task.rb

```
class Task < ApplicationRecord
  acts_as_paranoid —❶

  ...
end
```

以上で実装は完了です。

Railsコンソールを立ち上げ、動作確認してみましょう。

```
$ bin/rails c
```

まず、.allメソッドを実行してみましょう(結果を見やすくするために、.pluckメソッドを使ってidとdeleted_atのみ表示させています)。

Railsコンソール

```
> Task.all.pluck(:id, :deleted_at)
   (0.5ms) SELECT "tasks"."id", "tasks"."deleted_at" FROM "tasks" WHERE "tasks"."deleted_at" IS NULL
=> [[14, nil], [15, nil], [16, nil], [21, nil], [25, nil], [26, nil]]
```

発行されるSQLに「WHERE "tasks"."deleted_at" IS NULL」という条件があることに着目してください。論理削除に対応したモデルにはデフォルトで「論理削除されていない」という検索条件が適用されるようになるのです。.allのほか.findなどにも同様の条件がつき、通常の使い方においては論理削除されたものは実際に存在しないかのように扱えるようになります。

続いて論理削除の動作を確認しましょう。まずデータベースから1つTaskオブジェクトを取得して、taskという変数に代入します。次に、task.destroyを呼び出して削除を行います。destroyメソッドは、以前は物理削除として動作していました。しかし今は論理削除として動作するように変わっています。

Railsコンソール

```
> task = Task.first
  Task Load (0.3ms) SELECT "tasks".* FROM "tasks" WHERE "tasks"."deleted_at" IS NULL ORDER BY ➡
"tasks"."id" ASC LIMIT $1 [["LIMIT", 1]]
=> #<Task id: 14, name: "最初のタスク", description: "最初のタスクを行う", created_at: "2018-06-21 ➡
13:59:20", updated_at: "2018-06-21 13:59:20", user_id: 3, deleted_at: nil>

> task.destroy
   (0.3ms) BEGIN
  Task Update (0.6ms) UPDATE "tasks" SET "deleted_at" = $1, "updated_at" = $2 WHERE "tasks"."id" = ➡
$3 [["deleted_at", "2018-07-29 12:38:14.865274"], ["updated_at","2018-07-29
12:38:14.865441"], ["id", 14]]

   (1.3ms) COMMIT
=> #<Task id: 14, name: "最初のタスク", description: "最初のタスクを行う", created_at: "2018-06-21 ➡
13:59:20", updated_at: "2018-07-29 12:38:14", user_id: 3, deleted_at: "2018-07-29 12:38:14">
> task.deleted_at
=> Sun, 29 Jul 2018 21:38:14 JST +09:00

> task.deleted?
=> true
```

上記のように、destroyメソッドを実行するとdeleted_atに現在時刻が格納されることを確認できます。また、ここでは最後にdeleted?メソッドを呼んでいますが、このメソッドを使うとオブジェクトが論理削除されているかどうかを調べることができます。

さきほど.allメソッドでは論理削除されたオブジェクトを取得できないと書きましたが、それでは、もしも論理削除されたオブジェクトを含めたすべてのオブジェクトを取得したい場合にはどうすればよいのでしょうか？ それには.with_deletedを利用します。ほかにも、論理削除されたオブジェクトだけを取得する.only_deletedや、論理削除されたオブジェクトが含まれないことを明示的に示す.without_deletedも用意されています。それぞれの動作は以下を確認してください。

Railsコンソール

```
> Task.all.pluck(:id, :deleted_at)
   (0.5ms) SELECT "tasks"."id", "tasks"."deleted_at" FROM "tasks" WHERE "tasks"."deleted_at" IS NULL
=> [[15, nil], [16, nil], [21, nil], [25, nil], [26, nil]]

> Task.with_deleted.pluck(:id, :deleted_at)
   (0.4ms) SELECT "tasks"."id", "tasks"."deleted_at" FROM "tasks"
=> [[15, nil], [16, nil], [21, nil], [25, nil], [26, nil], [14, Sun, 29Jul 2018 21:38:14 JST
+09:00]]

> Task.without_deleted.pluck(:id, :deleted_at)
   (0.5ms) SELECT "tasks"."id", "tasks"."deleted_at" FROM "tasks" WHERE "tasks"."deleted_at" IS ➡
NULL AND "tasks"."deleted_at" IS NULL
=> [[15, nil], [16, nil], [21, nil], [25, nil], [26, nil]]

> Task.only_deleted.pluck(:id, :deleted_at)
   (0.5ms) SELECT "tasks"."id", "tasks"."deleted_at" FROM "tasks" WHERE ("tasks"."deleted_at" IS ➡
NOT NULL)
=> [[14, Sun, 29 Jul 2018 21:38:14 JST +09:00]]
```

paranoiaにはリストア（復元）機能も用意されています。次のように .restore メソッドに復元したいレコードのIDを渡すことで、論理削除済みのレコードを論理削除されていない状態に戻すことができます。

Railsコンソール

```
> Task.restore(14)
  Task Load (0.3ms) SELECT "tasks".* FROM "tasks" WHERE ("tasks"."deleted_at" IS NOT NULL) AND ➡
"tasks"."id" = $1 LIMIT $2 [["id", 14], ["LIMIT", 1]]
   (0.2ms) BEGIN
  SQL (0.3ms) UPDATE "tasks" SET "deleted_at" = NULL, "updated_at" = '2018-07-29 12:41:55.637487' ➡
WHERE "tasks"."id" = $1 [["id", 14]]
   (1.6ms) COMMIT
=> [#<Task id: 14, name: "最初のタスク", description: "最初のタスクを行う", created_at: "2018-06-21 ➡
13:59:20", updated_at: "2018-07-29 12:41:55", user_id: 3, deleted_at: nil>]

> Task.find(14)
  Task Load (0.3ms) SELECT "tasks".* FROM "tasks" WHERE "tasks"."deleted_at" IS NULL AND "tasks".➡
"id" = $1 LIMIT $2 [["id", 14], ["LIMIT", 1]]
=> #<Task id: 14, name: "最初のタスク", description: "最初のタスクを行う", created_at: "2018-06-21 ➡
13:59:20", updated_at: "2018-07-29 12:41:55", user_id: 3, deleted_at: nil>
```

最後に、論理削除に対応したモデルで物理削除を行いたい場合にはどうしたら良いのでしょうか。destroyメソッドは論理削除に置き換わってしまいましたが、代わりにreally_destroy! というメソッドが用意されています。これを使えば物理削除を行うことができます。

Railsコンソール

```
> Task.find(1).really_destroy!
  Task Load (0.5ms) SELECT "tasks".* FROM "tasks" WHERE "tasks"."deleted_at" IS NULL AND "tasks".➡
"id" = $1 LIMIT $2 [["id", 14], ["LIMIT", 1]]
   (0.2ms) SAVEPOINT active_record_1
  SQL (0.4ms) DELETE FROM "tasks" WHERE "tasks"."id" = $1 [["id", 14]]
   (0.1ms) RELEASE SAVEPOINT active_record_1
=> #<Task id: 14, name: "最初のタスク", description: "最初のタスクを行う", created_at: "2018-06-21 ➡
13:59:20", updated_at: "2018-07-29 12:43:29", user_id: 3, deleted_at: "2018-07-29 12:43:29">
```

Special 3

SJRの向き・不向きを理解しよう

本書Chapter8の「8-2-3 コントローラからJavaScriptを返して実行する（SJR）」の後に入る予定だった内容です。

Ajaxリクエストへのレスポンスに JavaScriptを返し、そのJavaScriptがブラウザで実行されることで画面を動的に更新できるSJRはとても便利です。特に、link_toヘルパーなどにremote: trueオプションをつけるだけでAjaxが実現できることと組み合わせると、クライアントサイドで自分でJavaScriptを書かなくても、サーバサイドでSJRでJavaScriptを送り出すだけで柔軟なAjaxを実現することができます。

しかし、ここには落とし穴があります。筆者の経験では、SJRには向いている処理と向いていない処理があり、向いていない処理をSJRで書いてしまうと、メンテナンスしづらい複雑なコード状態に陥ってしまいがちなのです。

SJRに向いているのは、画面を一度動的に更新すれば完結するような単純な処理です。特に画面全体もしくは一部を、ある程度複雑なHTMLの内容に置き換えたいといった場合には、少ない記述量で簡単に実現できます。この際に、更新したい画面のHTMLをサーバサイドの共通のテンプレートを通じて生成できる点※1、共通のビューヘルパーを活用できる点など、ほかの機能とのビューに関する共通化の恩恵を得られるのも嬉しい点です※2。

一方、向いていないのはクライアント側にある種の「状態」を持たせ、そのSJR以外の場所、すなわちクライアントサイドで実行されるほかのJavaScriptコードや、ほかのSJRからもその「状態」を変更するような処理です。別の言い方をすれば、あるSJRによる画面の更新が、その画面でJavaScriptで管理したい関心事にまつわる唯一の処理ではない場合には注意が必要となります※3。

なぜ「状態」を変更し、ほかの処理とその「状態」を共有するような処理がSJRには向いていないのでしょうか？ それには次の2つの理由があります。

1. SJRは基本的にAjax呼び出しが終わった後に実行される独立的なコードであるため、クライアントサイドに用意している独自のJavaScriptオブジェクトにアクセスするのが難しい。そのため、DOMの内容で「状態」を表すことになりがち。そうなると「DOM上の表現」を状態と読み替えてプログラミングする必要が生じ、コードが複雑になり、ミスが発生しやすくなる。
2. ほかの処理と画面の「状態」を共有し、ともに更新するようなケースでは、その「状態」に関わる操作を一元的に管理し、共通化することが望ましいが、処理の一部がSJRになっているとコードの場所が分散してしまい、共通化できる

※1　render_to_stringなどを利用して生成したHTMLを埋め込むSJRを記述できます

※2　このほかのメリットとして、フラグメントキャッシュの利用による高速化が期待できる点や、ユーザーの利用環境の差異によらず一定のHTMLを表示できる点なども挙げられます

※3　このほかのSJRの苦手領域として、ES2015でコードを書いておいてWebpackで互換性のあるJSに変換することができない点が挙げられます。もしもクライアントサイドでこうした変換を前提にES2015で記述していても、SJRは互換性のあるJSで書く必要があります。このような場面では、混乱を避けるためにSJRで記述するコードの量を絞ったほうが管理しやすいかもしれません

度合いが制限される。

具体例を見ていきましょう。ある商品に対するレビュー（評価）を投稿する機能を考えてみます。この機能の仕様は次のようになっています。

- レビューは☆による評価とコメントから構成されている。
- 投稿したいレビューの☆の数はボタンを押すことで増やせる（この段階ではサーバには送られない）。
- レビューは一覧になっている。投稿したいレビューの☆とコメントを設定して「レビューを投稿する」ボタンを押すと、データベースに保存され、一覧の先頭に追加される。

この場合、ユーザーの操作は「レビュー投稿欄で☆を増やす」と「レビューを投稿する」の2種類になります。前者をクライアントサイドで、後者をSJRで投稿すると、コードは次に示す4つのファイルのようになります（なお、ここではjQueryを利用しています）。

図S-3-1

app/views/product_reviews/index.html.slim

```
h1 この商品のレビュー

table
  thead
    tr
      th もらった☆
      th コメント
      th 投稿日時
      th
  tbody#product-reviews
    = render @product_reviews※4

div style='border: 1px solid gray; padding: 10px'
  h2 新しくレビューを投稿する
```

※4　renderメソッドにモデルのコレクションを渡すと、それぞれのモデルオブジェクトについて対応するパーシャルを類推しながら描画してくれます。Chapter10-11「ビュー（プレゼンテーション）の共通化」で紹介している:collectionオプションの省略系となります

```
= form_with(model: @new_review, id: 'new-product-review-form') do |form|
  .field
    = form.hidden_field :stars, id: 'new-product-review-stars-field'
    button#new-product-review-stars-increment type='button'
      | ☆をあげる!
    span#new-product-review-stars
  .field
    = form.label :comment, 'コメント'
    = form.text_field :comment, id: 'new-product-review-comment'
    span.actions
      = form.submit 'レビューを投稿する'
```

app/views/product_reviews/_product_review.html.slim

```
tr
  td = '☆' * product_review.stars
  td = product_review.comment
  td = l product_review.created_at, format: :short
  td = link_to 'x', product_review, method: :delete
```

app/assets/javascripts/product_reviews.js

```
$(function() {
  $('#new-product-review-stars-increment').click(function() {
    var $stars = $('#new-product-review-stars');
    var $field = $('#new-product-review-stars-field');
    $stars.html($stars.text() + '☆');
    $field.val(Number($field.val()) + 1); —❶
  });
});
```

app/views/product_reviews/create.js.erb

```
$('#product-reviews').prepend('<%= j render @product_review %>')
$('#new-product-review-stars').html('');
$('#new-product-review-stars-field').val(0); —❷
$('#new-product-review-comment').val('');
```

create.js.erbのSJRでは新しいレビューを一覧の先頭に追加し、投稿フォームをクリアしています。SJRのコード自体は単純でわかりやすい内容ですが、コードの全体像としてはどうでしょうか？

ここでは、まさに先ほど説明したような問題が生じています。

この機能では、クライアントサイドもサーバサイドも、ともに「新しいレビューを投稿するためのフォームにおける現在の☆の数」（$('#new-product-review-stars-field')）を更新します（上記❶❷）。本来ならば、☆の数を表す直接的な変数などをクライアントサイドに用意すべきところですが、SJR側から単純に処理ができるために、投稿欄の中の「☆☆☆」を表示している要素（$('#new-product-review-stars')）を画面の「状態」として利用してしまっているのです。このように表示上の都合が反映されたDOMの内容を「状態」として利用しているコードは複雑で、難解になります。また、表示に関する仕様変更があると処理が正しく動かなくなるなど、壊れやすいコードでもあります。

また、product_reviews.jsとcreate.js.erbは、それぞれ「assets」と「views」の下という物理的に離れた場所にあるため、両者がどちらも特定の「状態」を更新するということを把握しづらく、変更の際の影響範囲も特定しづらいという問題もあります。

このようになってしまったコードは、以下の戦略でリファクタリングすることができます。

1. SJRでは画面の状態を更新しないようにする
   a. SJRを利用せず、素のAJaxにする
   b. 画面の状態を更新する処理をSJRからクライアントサイドに切り出す
2. 状態管理を本来あるべき素直な形にする

上記1-bと2によってリファクタリングした例は次のようになります。

app/assets/javascripts/product_reviews.js

```
$(function() {
  var starsNum = 0;

  var renderStars = function(starsNum) {
    return '☆'.repeat(starsNum);
  }

  var updateView = function() {
    $('#new-product-review-stars').html(renderStars(starsNum));
    $('#new-product-review-stars-field').val(starsNum);
  };

  $('#new-product-review-stars-increment').click(function() {
    starsNum += 1;
    updateView();
  });

  $('#new-product-review-form').on('ajax:success', function(event, data, status, xhr) {
    $('#new-product-review-comment').val('');
    starsNum = 0; —❸
    updateView();
  });

  updateView();
});
```

app/views/product_reviews/create.js.erb

```
$('#product-reviews').prepend('<%= j render @product_review %>')
```

ここでは、以前SJRでやっていた処理のうち、画面の「状態」を更新する処理（すなわちレビュー投稿欄の☆の数をクリアする処理、リファクタリング前のコードの❷）を分離して、rails-ujsのイベントハンドラajax:successを使ってクライアントサイドに記述しています（❸）。SJRでは残った処理、すなわちテンプレートの描画だけを行っています。このようにすれば、☆の数をstarsNumという変数で素直に管理し、画面の表示状態をその状態に基づいて行うという形にコードを整理することができるのです。

このリファクタリング結果のように、不向きと考えられる「状態管理を一緒に行いたい」操作に対してもSJRを活用することは不可能ではありませんが、次のような難点は残ります。

- Ajax後に実現したい処理をSJRとクライアント側の二箇所に書くことになり、コードの記述量が増える。SJRの利点の1つであるビューテンプレートの活用はできるが、もう1つの利点であるハンドラを自分で書かなくて済む手軽さは失われる。
- SJR処理後に処理を挟むにはrails-ujsのイベントハンドラを利用するしか方法がない。この方法では実DOMのイベントハンドラを扱うため、ReactやVueなどの仮想DOMを使ったライブラリとの相性が悪く、正しく動かすためのコストが大きくなる[※5]。

まとめとしては、これまで解説してきた課題の存在を踏まえて、筆者としてはSJRを次のような感覚で利用することをおすすめします。

- 主にテンプレートを利用したHTMLの組み立てに対して利用する。クライアントサイドの状態を更新するような用途で安易に利用しない。
- 開発初期のちょっとした処理、実験的な機能、とにかく開発を急いでいる場合の解法として利用する。
- 機能が育ち、SJRのコードが複雑・大量になってきたときには、SJRを使わない方法に置き換えていくことも検討する。

ぜひ、本節の内容を参考にして、賢くSJRを活用していただければと思います。

※5　これは、SJRを利用せず、`remote: true`を使ったAjaxにクライアントサイドでハンドラを記述するだけでも生じる問題です。仮想DOM中心でコーディングをしたい場合には`remote: true`自体を利用しないか、クライアントサイドのハンドラなしのごく単純なSJRに限定するのが良いと筆者は考えています

Special 4

プログレスバーのカスタマイズ

本書Chapter8の「8-3-1 Turbolinksの発行するイベント」の後に入る予定だった内容です。

Turbolinksでは画面上部のプログレスバーがデフォルトで有効になっています。プログレスバーとはリクエストの進捗状況を表現するもので、表示の遅延によりユーザーに与えるストレスを軽減することを目的としています。とりわけTurbolinksではリンクをクリックした際の視覚的なレスポンスがないため、プログレスバーによる通知は有効な手段となっています。

このプログレスバーはCSSによりデザインされており、次のように任意のCSSファイル上で.turbolinks-progress-barクラスをオーバーライドすることで見た目を変更することができます。

CSSファイル

```
.turbolinks-progress-bar {
  height: 5px;
  background-color: green;
}
```

なお、プログレスバーは表示の遅延をユーザーに伝える目的の機能なので、画面遷移にかかる時間が短い場合（デフォルトで500ms）には表示されないようになっています。「Turbolinks.setProgressBarDelay(0);」と設定を変更することで、プログレスバーを必ず表示することができます。開発中の動作確認などの際には、一時的に設定を変えると便利でしょう。

Special 5

Tips

本書「Appendix」として掲載予定だった内容です。

S-5-1 byebug でデバッグ

コードが上手く動いていない（バグがある）ときは、バグを探して取り除く作業（デバッグ）をする必要があります。最も手軽にできるデバッグは p メソッドや puts メソッドを利用する「プリントデバッグ」ですが、デバッグを助けてくれるツール「デバッガ」を使ってもっと効率よくデバッグを行うこともできます。Railsではデフォルトのデバッガとしてbyebugというgemを使います（Gemfileにbyebugが記入されています）。

使い方は、デバッグを行いたい箇所に「byebug」と書くだけです。その状態でRailsアプリケーションを実行すると、byebugを書いた行でアプリケーションの実行が停止し、同時に入力プロンプトが立ち上がります。プロンプト上では停止したタイミングにおけるインスタンスや変数の状態などを確認することができます。

実際にtaskleafでbyebugを使ってみましょう。app/controllers/tasks_controller.rb のcreateアクションにbyebug を2箇所書いてください（❶❷）。

app/controllers/tasks_controller.rb

```
...
   def create
    @task = current_user.tasks.new(task_params)
    byebug —❶
    if params[:back].present?
      render :new
      return
    end

    if @task.save
      byebug —❷
      TaskMailer.creation_email(@task).deliver_now
      5.times do
        SampleJob.set(wait: 5.seconds).perform_later
        HighPriorityJob.set(wait: 5.seconds).perform_later
      end
      redirect_to @task, notice: "タスク「#{@task.name}」を登録しました。"
    else
      render :new
    end
  end
...
```

次に、railsサーバを起動し、ブラウザで新規登録画面を開いて「登録する」ボタンを押下してください。

図S-5-1

登録ボタンを押下すると、ブラウザがローディング状態のままになります。その状態でサーバを起動しているターミナルを見ると、以下のように1つ目のbyebugを書いた場所で止まります。

```
   29:   def create
   30:     @task = current_user.tasks.new(task_params)
   31:     byebug
=> 32:     if params[:back].present?
   33:       render :new
   34:       return
   35:     end
   36:
(byebug)
```

この状態で、ターミナルでparamsと打ってEnterを押すとparams（newからcreateに渡されたパラメータ）の内容を見ることができます。

```
(byebug) params
<ActionController::Parameters {"utf8"=>"✓", "authenticity_token"=>"au9hmqLLuLi8Xfd2ENBs2Ycr+oJBneolkVp
uXPyXTS3OV+OdAXeIUeo4ZFTn7xrsMC5WSVL42W2lQcEgPfjoiQ==", "task"=><ActionController::Parameters
{"name"=>"byebugを使ってみる", "description"=>"byebugを書いた場所で止めた時にどうなるか確認したい"}
permitted: false>, "commit"=>"登録する", "controller"=>"tasks", "action"=>"create"} permitted: false>
```

task_paramsと打ってEnterを押すと、task_paramsメソッド（paramsからtaskに関わるパラメータのみフィルタリン

グするメソッド）が実行された結果を見ることができます。

```
(byebug) task_params
<ActionController::Parameters {"name"=>"byebugを使ってみる", "description"=>"byebugを書いた場所で止めた時にどうなるか確認したい"} permitted: true>
```

@task と打ってEnterを押すと、@taskの内容をみることができます。この時点ではcurrent_user.tasks.new(task_params) でnew しただけでsaveされる前の状態なので、id, created_at, updated_atはnilになっていることがわかります。

```
(byebug) @task
#<Task id: nil, name: "beybugを使ってみる", description: "byebugを書いた場所で止めた時にどうなるか確認したい", created_at: nil, updated_at: nil, user_id: 1>
```

次にcキーを打ってEnterを押すと処理が継続され、2つ目のbyebugを書いた位置で止まります。（ちなみにcはcontinueのcです）

```
   15:    def create
   16:      @task = current_user.tasks.new(task_params)
   17:       byebug
   18:       if @task.save
   19:          byebug
=> 20:          redirect_to tasks_url, notice: "タスク「#{@task.name}」を登録しました。"
   21:       else
   22:         render :new
   23:       end
   24: end
(byebug)
```

ここで、save後の@taskの内容を見てみましょう。ターミナルで@taskと打ってEnterを押します。1つ目のbyebugではnilだったid, created_at, updated_at にそれぞれ値が入っていることがわかります。

```
(byebug) @task
#<Task id: 1, name: "beybugを使ってみる", description: "byebugを書いた場所で止めた時にどうなるか確認したい", created_at: "2018-10-08 01:35:31", updated_at: "2018-10-08 01:35:31", user_id: 1>
```

h と打ってEnterを押すと、byebugのヘルプが表示されます。
byebugを終了したいときは、exit! を打ってEnterを押します。

以上のように、byebug を利用すればプリントデバッグに比べて自由かつインタラクティブに調べたい内容を実行することができます。エラーになってしまったりテストが落ちてしまったりと、Railsアプリケーションが期待通り動かないときには、怪しいところを見定めて byebug を書いておき、原因を切り分けて行く方法が有効です。

上記で説明した内容以外にもbyebugは様々な機能を持っています。さらに詳しい説明はRails Guides※1を確認してください。

S-5-2 Bundlerでgemの中身を調べる

Railsアプリケーションではさまざまなgemを使います。gemはブラックボックスではありません。コードを見て、動作を確認することもできますし、コードに手を加えることもできます。

taskleafのルートディレクトリで、bundle show activerecord と打ってみてください。

```
$ bundle show activerecord
path/to/.rbenv/versions/2.5.1/lib/ruby/gems/2.5.0/gems/activerecord-5.2.1
```

Active Record gem が配置されているディレクトリのパスが表示されます。taskleafはRailsサーバ起動時にこれらのファイルを読み込んでいることを示しています。これらのファイルをエディタで開けばコードリーディングをすることができますし、編集することもできます。プリントデバッグやbyebugなどのデバッガによってtaskleafで使っているActive Record gemの挙動を確認しやすくすることもできます。

gemのソースコードをエディタで開くために使える便利なコマンドがあります。ターミナルで「bundle open activerecord」と入力してEnterを押すと、お使いのエディタでgemのディレクトリを開いてくれます（この際使われるエディタは環境変数 EDITOR に指定されたプログラムです。もしも環境変数が指定されておらずエディタで開けない場合は、~/.bashrcなどにexport EDITOR=<editor name>という指定をしてください）。

なお、gemのソースコードに手を入れたあとは、元の状態に戻したくなることがあるかもしれません。そのようなときには「bundle pristine」を利用すると、一括で元の状態に戻すことができます。また、特定のgemのみ戻したいときは「bundle exec gem pristine <gem name>」が利用可能です。

以上のコマンドを活用して、普段の開発の中でもぜひgemの世界に潜り込んでみてください。

S-5-3 Method#source_locationでメソッドの実装箇所を探す

Railsアプリケーションを開発していると「あれ?このメソッドどこで実装されてるんだろう?」ということがよくあります。そんなときはMethod#source_locationを使って実装箇所を確認しましょう。ターミナルでrails cを入力し、Railsコンソールを起動します。

※1 https://guides.rubyonrails.org/debugging_rails_applications.html#debugging-with-the-byebug-gem

User.all のallメソッドがどこで実装されいてるか見てみましょう。

Railsコンソール

```
> User.method(:all).source_location
=> ["path/to/.rbenv/versions/2.5.1/lib/ruby/gems/2.5.0/gems/activerecord-5.2.1/lib/
active_record/scoping/named.rb", 26]
```

ActiveRecord gemの /lib/active_record/scoping/named.rb ファイルの26行目で実装されていることがわかります。methodメソッドは、レシーバにある指定した名前のメソッドをオブジェクトとして返してくれるメソッドです。ここで得たallメソッドのオブジェクトのsource_locationを呼ぶことで、allメソッドが実際に実装されているファイルや、ファイル内の位置を知ることができます。

S-5-4 ActiveRecord::Relation#to_sqlでSQLを確認しよう

ActiveRecordを利用していると、大抵のシーンではSQLを直接書かなくて済みます。ActiveRecordが内部で自動的にSQLを組み立ててくれるからです。しかし、だからといってSQLについてまったく気にしないでも構わないかというと、そうでもありません。どんなSQLが生成されるかを次のような観点で確認することは、よい品質のソフトウェアを開発するために重要です。

- 意図せず冗長なSQLになっていないか。パフォーマンス上問題ないか
- 実装したモデルのスコープが意図通りのSQLを生成するか
- メソッドチェーンで複数のメソッドやスコープをつなぎ合わせたとき、想定したSQLになっているか

SQLの生成結果を確認するためには、ActiveRecord::Relation#to_sqlメソッドを使うことができます。早速使ってみましょう。ターミナルでrails cを入力し、Railsコンソールを起動します。

すべてのユーザを取得するUser.allのSQLを確認する場合は、User.all.to_sqlと打ってEnterを押します。

Railsコンソール

```
> User.all.to_sql
=> "SELECT \"users\".* FROM \"users\""
```

usersテーブルから*をSELECTするSQLが生成されることがわかります。

続いて、tatsuosakuraiという名前のユーザを取得するSQLを見てみましょう。

Railsコンソール

```
> User.where(name: 'tatsuosakurai').to_sql
=> "SELECT \"users\".* FROM \"users\" WHERE \"users\".\"name\" = 'tatsuosakurai'"
```

WHEREを使って条件が絞られていることがわかります。

このようにActiveRecord::Relation#to_sqlメソッドを使うと、どんなSQLが生成されるかを確認できて便利です。「なぜか処理が重い」「思った通りのデータが取得できない」といった場合には、このメソッドを使ってSQLを確認するとよいでしょう。

Special 6

お役立ちリンク集

本書「Appendix」として掲載予定だった内容です。

S-6-1 Ruby

ruby/ruby	https://github.com/ruby/ruby Rubyの本体コード
ruby-lang.org	https://www.ruby-lang.org/ja Rubyの公式サイト
るりまサーチ	https://docs.ruby-lang.org/ja/search/ Rubyのリファレンスマニュアル検索サイト
Rubyアソシエーション	https://www.ruby.or.jp/ja/ Rubyの普及・発展のための組織「一般財団法人 Rubyアソシエーション」のサイト
日本各地域でのRubyコミュニティ	https://github.com/ruby-no-kai/official/wiki/RegionalRubyistMeetUp コミュニティリンク集
るびま	https://magazine.rubyist.net RubyistのRubyist による、Rubyistとそうでない人のためのウェブ雑誌
Ruby Weekly	https://rubyweekly.com Rubyに関する週刊メルマガ
PB memo	http://d.hatena.ne.jp/nagachika Rubyコミッタnagachika さんによるRubyのチェンジログ解説

S-6-2 Rails

rails/rails	https://github.com/rails/rails Rails 本体のコード
Riding Rails	https://weblog.rubyonrails.org Railsの公式ブログ。Railsのバージョンアップ情報や、まだリリースされていない新機能等を紹介
Rails APIリファレンス	https://api.rubyonrails.org/ RailsのAPIリファレンス

Ruby on Rails Guides	https://guides.rubyonrails.org Railsのリファレンスガイド。各機能を体系的に学べる。日本語はこちら https://railsguides.jp
なるようになるブログ	http://y-yagi.hatenablog.com Railsコミッタのy-yagiさんによるコミットログを解説。通称ヤギヌマ新聞
Ruby on Rails Tutorial	https://www.railstutorial.org Railsを使ったWeb開発を学べるチュートリアル。 日本語はこちら https://railstutorial.jp
everyleaf/el-training	https://github.com/everyleaf/el-training 株式会社万葉の提供するRailsの教育カリキュラム

S-6-3 gem

RubyGems.org	https://rubygems.org gemのホスティングサイト
Awesome Ruby	https://awesome-ruby.com gemのリンク集
Awesome Ruby @LibHunt	https://ruby.libhunt.com gemを検索・比較できるサイト
The Ruby Toolbox	https://www.ruby-toolbox.com gemを検索・比較できるサイト

S-6-4 RSpec

RSpec documentation	http://rspec.info/documentation RSpecのAPIリファレンス
RSpec - Relish	https://relishapp.com/rspec RSpecのリファレンス
Better Specs	http://www.betterspecs.org RSpecのガイドライン
rspec-style-guide	https://willnet.gitbooks.io/rspec-style-guide/content/ 可読性の高いテストコードを書くためのお作法集