Railsの気持ちを考えながらコントローラとビューを整頓する

RailsTokyo#3

2026-02-19 諸橋恭介 @moro

Sponsor RubyStackNews

Reach senior Ruby & Rails engineers worldwide

Put your brand in front of experienced developers, tech leads, and decision makers across the global Ruby ecosystem.

Become a Sponsor

Keynote: dynamic!

Speaker

MOROHASHI Kyosuke

日々Railsアプリケーションを書いているプログラマ。Ruby と Rails、TDD が好き。 著書に「Rails 3レシピブック(共著)」「はじめる! Cucumber」など。好きなメソッドの変遷は、Array#each から Object#extend を経て Hash#compact。

dynamic!

Kaigi on Rails 2025 Keynote:

2025-09-26 諸橋恭介 @moro

あなたのRailsの話をしよう

- これまでの経験を活かして新しいことをすると題材は出てくると思う
- ミートアップに必要なのはあなたのトーク
  - 「偏った」分野でも誰かに届けばいいと思って話していた
  - 複数のスピーカーによる異なるトピックがあることの重要性
- 書いたコードや話した内容を元にコミュニティで知られていく経験はよいものです
- RailsTokyoがこれからも気軽に登壇できる場所であってほしい

https://speakerdeck.com/yahonda/railsnohua-wo

Rails の気持ちを考えながら
コントローラとビューを整頓する

Rails の気持ちを考えながら

コントローラとビューを整頓する

Rails の気持ちに沿うと、シンプルに実装できる

- イベントエンティティを見出して RDB に表現し、 has_many :through な AR モデルとして、一級に扱う
- イベントエンティティに代表される「コト」を REST リソースとして捉え やりたいことをその CRUD 操作で表現する
- Rails が提供するインターフェースを思い描き、 AR や AMo 、 Ruby の柔軟性を活かして継続的に開発・リファクタリングしていく

私が思う Rails の気持ち

- 昔から変わらずある基本 API は、 Rails も使って欲しかろう
- こうできたらなと思ったら、 Rails もそのとおりに発展した API も気持ちに沿うだろう
- 基本設計を維持したうえで洗練され続けている API も、きっとどんどん活用して欲しいのではないか

昔からある基本 API を使う

- REST であり、アプリケーション機能をリソースの CRUD で表現する
  - 「 CRUD で済むアプリケーションに向いてる」というのとはちょっと筋が逆
- フレームワークとの結合には、 XXX::Base の継承を用いる
  - それ以外でいっぱい継承しろ、とは言ってない。するなとも言ってないけど
  - アプリケーションコードでの共通化は Concern を使う、とかも

こうしたいと思ってたら、そう発展した API を使う

- AR の DB 非依存の機能を使いたいなーと思っていたら ActiveModel ができた
- アプリケーションでの共通規定クラスが欲しいと思ってたら ApplicationRecord が定義されるようになった
- params.require(:q).permit(:a, :b, :c) がめんどくさいと思ってたら params.expect が追加された

基本設計を維持、洗練され続けている API を使う

- ActiveRecord はほんとうにすごい。 DB のテーブル / カラム名を
  クラス / 属性名にマッピングするという基本機能は変わらず便利なうえで、

- ActiveRecord::Relation と、その基盤の Arel::Relation 、それを使った
  AssociationProxy が本当にすごいので、これを使い倒したい

  - has_many ~ :through をさらに :through できて、まともなクエリになるんですよ !!

Rails の気持ちを考えながら

コントローラとビューを

整頓する

コントローラとビュー

- Fat Controller よくないよね、くらいしか語られていない
  - 無軌道に Fat なコントローラはよくない
    - (ネガティブ表現) な (対象) はよくない。それはそう
- コントローラの責務だけやるコントローラならばよいのでは？
  - (ポジティブ表現) な (対象) ならよいのでは？ それはそう
- あんまり語られてこなかったところなので考えてみたい

Rails の気持ちを考えながら
コントローラとビューを
整頓する

“

整頓　　。整頓可愛

小　　、誰嫌。

Kent Beck 「Tidy First?」
第 1 部 整頓

可愛くてふわふわした小さなサブセット

- 全面リニューアルや、ナントカアーキテクチャの導入ではない
- 日々の開発をしながらコードを少しずつ設計して変更する
  - dynamic! の話です

Rails の気持ちを考えながら
コントローラとビューを整頓する

- ActiveRecord::Relation を活用する
- コントローラのインスタンス変数を減らす
- 一部の includes はビューで指定する

- ActiveRecord::Relation を活用する
- コントローラのインスタンス変数を減らす
- 一部の includes はビューで指定する

基本設計を維持、洗練され続けている API を使う

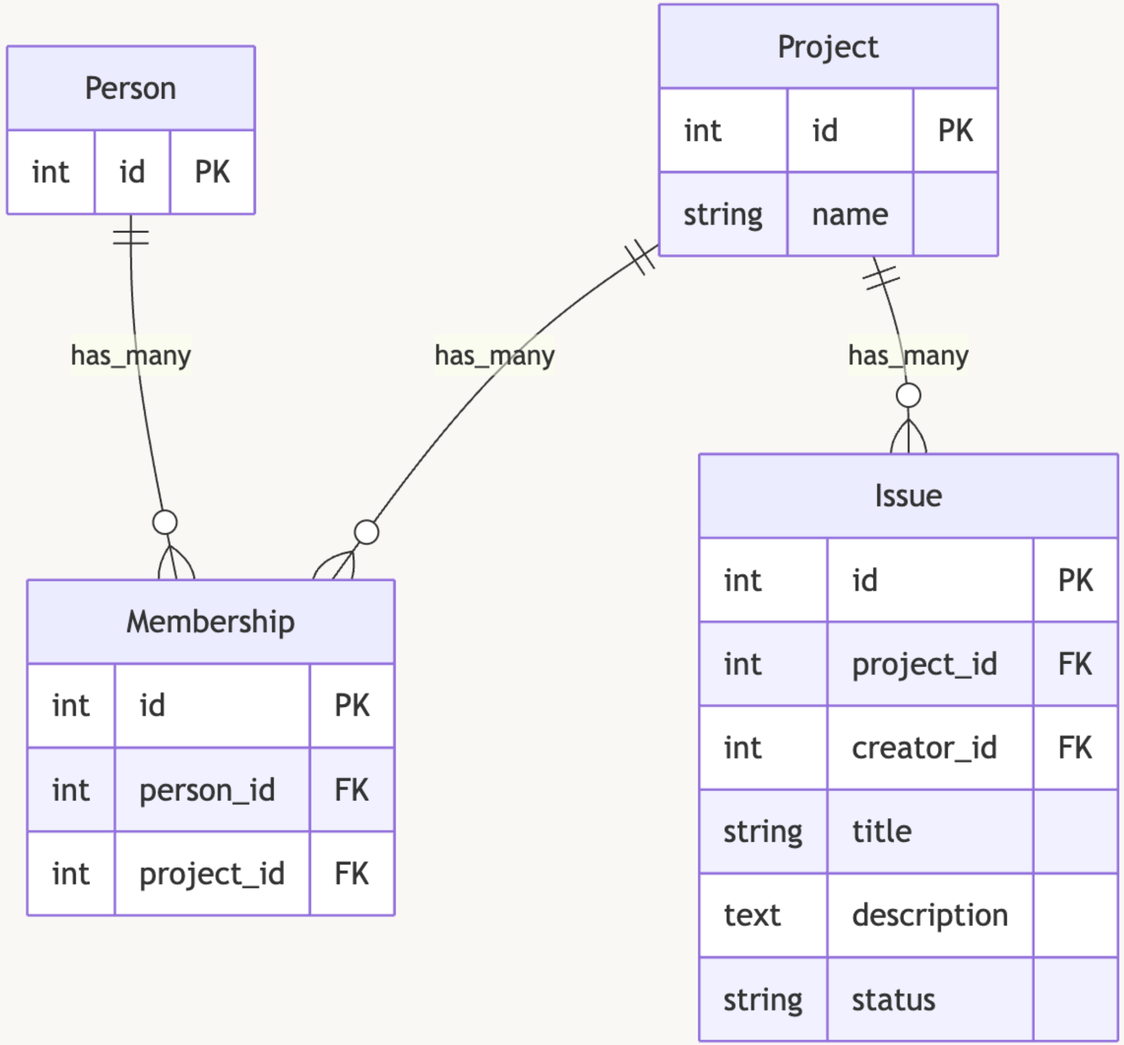
- ActiveRecord はほんとうにすごい。 DB のテーブル / カラム名を
  クラス / 属性名にマッピングするという基本機能は変わらず便利なうえで、
- ActiveRecord::Relation と、その基盤の Arel::Relation 、それを使った
  AssociationProxy が本当にすごいので、これを使い倒したい
  - has_many ~ :through をさらに :through できて、まともなクエリになるんですよ !!

- ActiveRecord::Relation を活用する
  - 権限チェックをアソシエーションで表現する
  - 複雑な関係を :through の :through で表現する
- コントローラのインスタンス変数を減らす
- 一部の includes はビューで指定する

権限チェックをアソシエーションで表現する

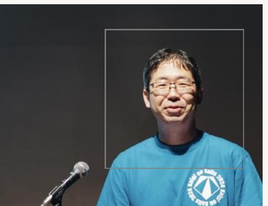
- コントローラのコードが増える理由の一つに、対象リソースへのアクセス可否制御のコードがある
- アソシエーションを活用することである種の権限制御ができる
  - このアソシエーションを支えるのが ActiveRecord::Relation

架空の Issue 管理を例に取ります



Person.hasmany :projects, through: :memberships

- hasmany :through アソシエーション
- ActiveRecord で N:M 関連を表現する定番
  - 交差テーブルもモデルとして扱える
  - その関連でをモデルとして扱えるため、さらに情報を付加できる
    - memberships の role など



hasmany が public メソッドなので rails c での試行錯誤がやりやすいの良いですよね。好き

Rails の気持ちになって考える

- 最初の最初は habtm こと hasandbelongstomany 関連だけがあった
- Rails 2 の段階で、 hasmany :through が追加された
- そのあとも、 hasone :through や belongsto :through など、through 族が隆盛を極めている
  - ドメインの表現として優れている (きょうはこっちの話)
  - 機能的にも through の through などで関連を辿れるのがちょう便利じゃないですか !? AR スゴイ !

改善前：ロードした後に権限をチェックする

- Issue を更新する前に、以下をチェックしたい:
  - アクセス者がプロジェクトのメンバーであること
  - 対象の Issue が本当にそのプロジェクトのものであること

改善前：ロードしたデータの権限をチェックする

```
class IssuesController < ApplicationController
  def update
    @project = Project.find_by(name: params[:project_name])
    member_ids = @project.memberships.pluck(:person_id)

    unless member_ids.include?(current_person.id)
      render plain: ' だめだよー '
      return
    end

    @issue = Issue.find(params[:id])
    unless @issue.project_id == @project.id
      render plain: ' だめだよー '
      return
    end

    @issue.update!(params.expect(issue: %i(title description status)))
    # ...
```

改善後：アクセス可能な集合から対象を探す

- ロードしてから権限をチェックするのでなく
- アクセスできる関係性にある集合から検索する
  - 「アクセスできる関係性」があることは、 memberships から辿れる

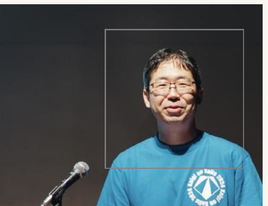
改善後：アクセス可能な集合から対象を探す

```
class IssuesController < ApplicationController
  def update
    @project = current_person.projects.find_by!(name: params[:project_name])
    @issue = @project.issues.find(params[:id])

    @issue.update!(params.expect(issue: %i(title description status)))
    # ...
  rescue ActiveRecord::RecordNotFound
    render plain: ' だめだよー '
  end
end
```

改善後：アクセス可能な集合から対象を探す

- current_person.projects は has_many ~ through: :memberships で結ばれているプロジェクトのみを選択する
- @project.issues はそのプロジェクトのイシューだけを選択する
  - それぞれ、見つからなければ RecordNotFound が発生する
- このクエリも、 ActiveRecord が効率的なやつを作ってくれる

```
rails-tokyo-sample(dev):001> Project.first.issues.class.ancestors
  Project Load (0.1ms)  SELECT "projects".* FROM "projects" ORDER BY
"projects"."id" ASC LIMIT 1 /*application='RailsTokyoSample'*/
=>
[Issue::ActiveRecord_Associations_CollectionProxy,
 Issue::GeneratedRelationMethods,
 ApplicationRecord::GeneratedRelationMethods,
 ActiveRecord::Delegation::ClassSpecificRelation,
 ActiveRecord::Associations::CollectionProxy,
 ActiveRecord::Relation,
 ActiveRecord::SignedId::RelationMethods,
 ActiveRecord::TokenFor::RelationMethods,
 ActiveRecord::FinderMethods,
 ActiveRecord::Calculations,
 ActiveRecord::SpawnMethods,
 ...
```



今日はそっち掘りませんが、 Issue::ActiveRecordAssociationsCollectionProxy とかも面白い

- ActiveRecord::Relation を活用する
  - 権限チェックをアソシエーションで表現する
  - 複雑な関係を :through の :through で表現する
- コントローラのインスタンス変数を減らす
- 一部の includes はビューで指定する

複雑な関係を :through の :through で表現でする

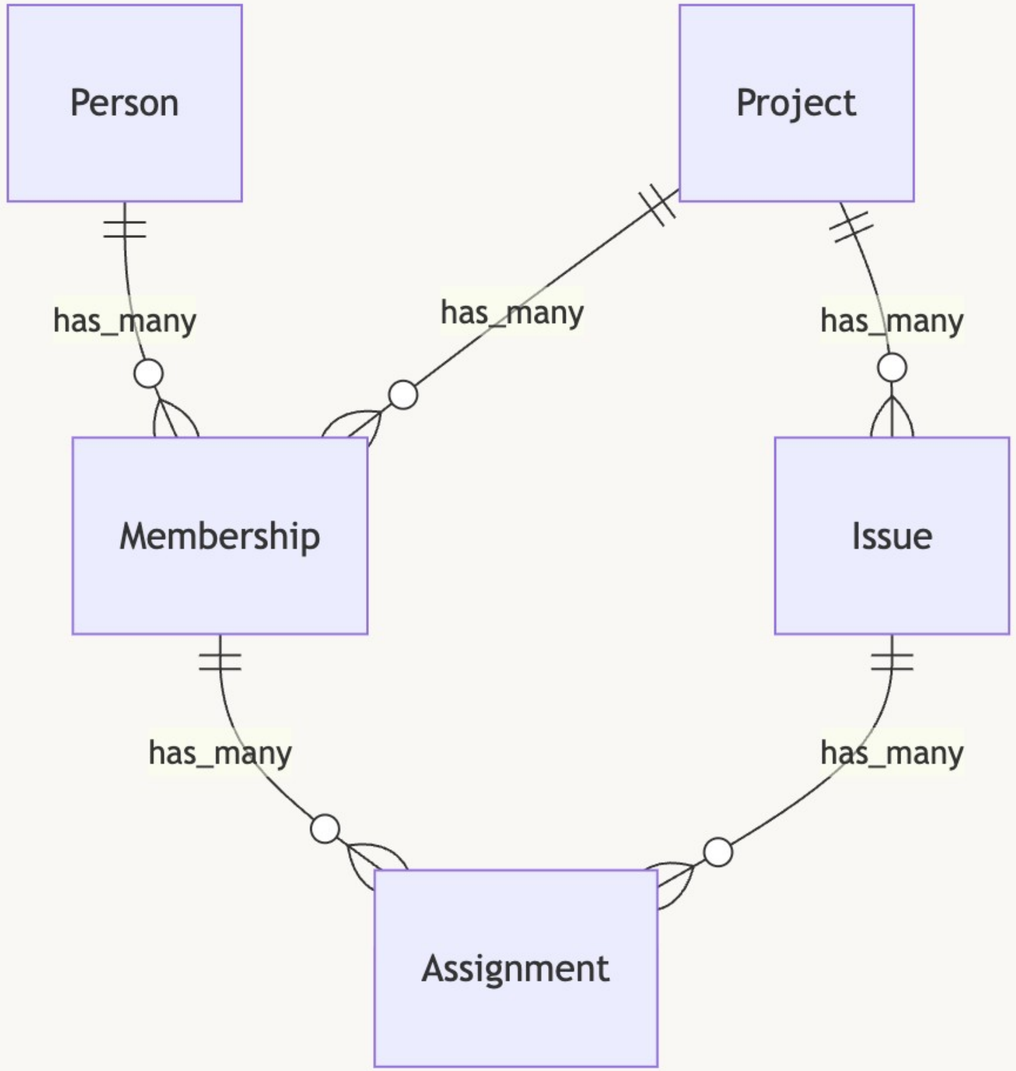
- Issue の担当者を assign するケースを考える
- 素朴に belongsto :person すると良くない
  - person が離任するときにも、 person 自体を消すわけにはいかない
  - すると assignment を個別に消す必要がある？ なんかめんどくさい

```
class Assignment < ApplicationRecord
  belongs_to :issue
  belongs_to :person
end
```

Assignment の belongsto 先は memberships かも

- じつは Assign 対象は「この人がこのプロジェクトに参画しているコト」では？
  - つまり belongsto :memberships
  - 実際のドメインモデルとしても妥当そう。 dependent: :destroy で離任したらアサインが消える
- いっぽう普段は「アサインされている人」くらいの気軽さで参照したい
  - issue.assignments.map { it.membership.person } とか書きたくない
- それが through を through するだけでできる
  - 発行されるクエリもまとも。 join したりまとめてプリロードしたりできる

改善後：through: :memberships



```ruby
class Issue < ApplicationRecord
  has_many :assignments
  has_many :memberships, through: :assignments
  has_many :assignees, through: :memberships, source: :person
end

class Assignment < ApplicationRecord
  belongs_to :issue
  belongs_to :membership
end

class Membership < ApplicationRecord
  belongs_to :person
  belongs_to :project
  has_many :assignments, dependent: :destroy
end
```

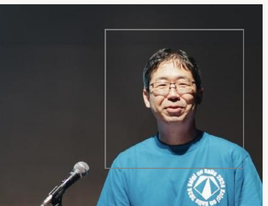
through を through す

```
rails-tokyo-sample(dev):002> project.issues.includes(:assignees)
  .map { [it, it.assignees.size] }

  Issue Load (0.3ms)  SELECT "issues".* FROM "issues" WHERE
"issues"."project_id" = 152639770 /*application='RailsTokyoSample'*/

  Assignment Load (0.1ms)  SELECT "assignments".* FROM "assignments" WHERE
"assignments"."issue_id" IN (77551669, 540637335) /
*application='RailsTokyoSample'*/

  Membership Load (0.1ms)  SELECT "memberships".* FROM "memberships" WHERE
"memberships"."id" IN (1010729577, 996949822, 664416693) /
*application='RailsTokyoSample'*/

  Person Load (0.1ms)  SELECT "people".* FROM "people" WHERE "people"."id" IN
(786122151, 902541635, 663665735) /*application='RailsTokyoSample'*/
```

```
rails-tokyo-sample(dev):003> project.issues.eager_load(:assignees)
  .map { [it, it.assignees.size] }

  Issue Eager Load (0.4ms)  SELECT "issues"."id" AS t0_r0,
"issues"."created_at" AS t0_r1, "issues"."creator_id" AS t0_r2,
"issues"."description" AS t0_r3, "issues"."project_id" AS t0_r4,
"issues"."status" AS t0_r5, "issues"."title" AS t0_r6, "issues"."updated_at"
AS t0_r7, "people"."id" AS t1_r0, "people"."created_at" AS t1_r1,
"people"."updated_at" AS t1_r2
FROM "issues"
LEFT OUTER JOIN "assignments" ON "assignments"."issue_id" = "issues"."id"
LEFT OUTER JOIN "memberships" ON "memberships"."id" =
"assignments"."membership_id"

LEFT OUTER JOIN "people" ON "people"."id" = "memberships"."person_id"

WHERE "issues"."project_id" = 152639770 /*application='RailsTokyoSample'*/
```

このアプローチの Rails の気持ち

- ActiveRecord::Relation は、 RDB の関係代数的な操作
  (選択 /where, 射影 /select, 結合 /join, etc)を Ruby オブジェクトとして扱える
  - 理論そのものではないが、典型的な Web アプリでやりたいことはだいぶできる
  - Relation 同士のチェーンや merge とかできるのすごくないです？
- このミラクル🦄な ActiveRecord を使い倒しましょう
  - 「ちゃんとしたテーブル構造」を定義して、それを辛くなく活用できる



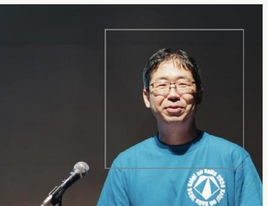
そうはいっても集計クエリよ、、、みたいなのはあるんですが、あれは「アクティブレコードパターン」を
適用しない方が良いフォースがあるわけなので、さらに別にアプローチのほうが良いと思ってます。

- ActiveRecord::Relation を活用する
- コントローラのインスタンス変数を減らす
- 一部の includes はビューで指定する

コントローラのインスタンス変数を減らす

- サイドバーの項目など、アクションの本分ではないデータは「コントローラのインスタンス変数」にはしない。

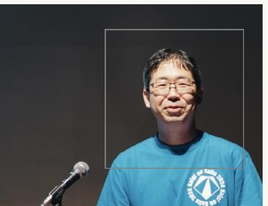
```
class IssuesController < ApplicationController
  def index
    @project = current_person.projects.find_by!(name: params[:pj_name])
    @issues = @project.issues.order(updated_at: :desc)

    #  こういうやつを減らしたい
    @news = @project.news.order(updated_at: :desc).limit(3)
  end
```



ここら辺から思想が強くなってきます。偏りがある！ 懇親会やインターネットで感想を聞かせてください。

コントローラの ivar は C/V のインターフェース

- 前提として、 Rails コントローラのインスタンス変数は、C で用意したデータを V に渡すインターフェースである
- いわゆる狭義の OOP におけるインスタンス変数とは、とか、MVC(2) のコントローラとは、みたいなことを考えすぎない
  - エントリポイントとなるメソッド (アクション) が違うと、セットされるインスタンス変数の種類自体が変わるとか、 OOP としてはダメな感じがする w
- Rails のことを考えましょう

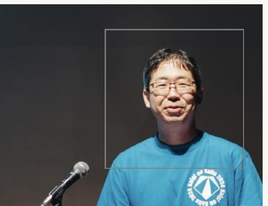


Rails と接続した先の、モデルのロジックを書くときなんかは、もちろん OOP の良い原則を尊重しましょう。
言い換えると、この接面を介して Rails 世界とドメイン界を境界づけるというか

- ActiveRecord::Relation を活用する
- コントローラのインスタンス変数を減らす
  - 特定表現のみで使うデータはビューでロードする
  - partial にコンポーネントを見出す
- 一部の includes はビューで指定する

Rails は単一アクションから複数表現を返せる

- request.format によって、 respondto で分岐したり、 index.html.erb / index.json.jbuilder の選出が変わる
  - 最近も Markdown レンダラーが入ったりしてるので、この思想も現役なはず
- コントローラの責務を最小化すると、 HTML 表現と JSON 表現とで両方で使うデータのみをビューに渡すべき



同じことは Hotwire を考えても言えそう。つまり turbo-stream が読み捨てる箇所をレンダリングするためのデータをコントローラでロードするのは無駄だと思います。後述のようにクエリ自体は発行されないかもですが。

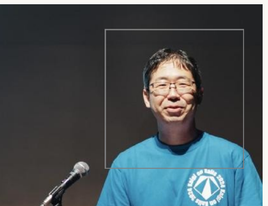
じゃあどうするか？

- 特定テンプレートだけで必要なデータは、 ERB テンプレートや helper でふつうにロードすれば良いのでは
  - つまり、 Ruby コードを書きましょう
  - *.erb だとちょっと窮屈かもしれないけれど、 *.json.jbuilder や、 *.json.ruby などではふつうに Ruby コードが書けるし

ビューでデータをロードするのアリ🐜??

- ビューでデータをロードする≠ビューに SQL を書く
- 生 SQL はさすがに読みづらいからダメだと思います🍐
  - でもあくまでプラクティカルな読み辛さの問題であり、作法の話ではない

```
<% query = 'SELECT * FROM news WHERE ...ORDER BY created_at DESC LIMIT 3' %>
<% ActiveRecord::Base.connection.execute(query).each do |row| %>
  ...
<% end %>
```

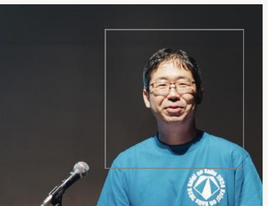


この「ビューに SQL 書くな」(わかる) がだんだんと「ビューで DB にアクセスするな」になったのかなと思ってます。
またはレイヤ分割アーキテクチャの影響か。でも MVC2 一般ではなく、 Rails 個別の話なので、というのが論旨です

ActiveRecord のメソッドチェーンを書くのはアリ

- モデルに published などの scope を定義して使ったり、 .order や .limit など AR のメソッドを呼んだりするのはあり
- チェインが長くて読みづらくなったら、適宜さらに scope やクラスメソッドを定義したり、ヘルパーに抽出したりする

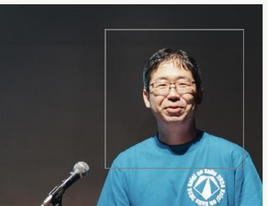
```
<% project.news.published.order(created_at: :desc).limit(3).each do |news| %>
  ...
<% end %>
```



抽出先として、コントローラも Ruby コードを書きやすいのでそこでいいじゃん、というのは一周回ってアリかも。
別メソッドにして helper 宣言で使えるようにするとかは良いと思います。 news でやることじゃないけど。

そもそも AR::Relation のクエリは lazy に評価される

- project.news.published の時点ではまだ SQL は発行されず、ActiveRecord::Relation が返る
  - .each でループを回す時点で初めて SQL が発行される
- これは、コントローラで ivar に入れても同じこと
- クエリ発行をわざと lazy にする Rails の意図を感じるので、使う側も頑張って気にしないようにしたい

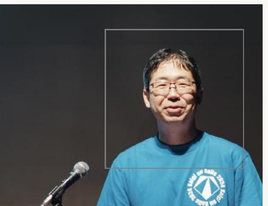


それを避けるにはコントローラで .to_a するとかになるけど、それはやりたくないじゃないですか

- ActiveRecord::Relation を活用する
- コントローラのインスタンス変数を減らす
  - 特定表現のみで使うデータはビューでロードする
  - partial にコンポーネントを見出す
- 一部の includes はビューで指定する

partial にコンポーネントを見出す

- render :partial の分割テンプレートにコンポーネントを見出す
  - 呼び出すタイミングで変数を詰め直せる
    - インスタンス変数を直接参照するのは避けたほうがよいですね
- Strict Locals のことを考えても、 partial にコンポーネントを見出しているように感じている



分割テンプレートの冒頭でフラグメントキャッシュかけると、テンプレートの変更でキャッシュが
自動でパージされる、というのもこの裏付けになっている気がするんですよね。

共通パーツをコンポーネントにする

```
# app/views/projects/_news.html.erb
<%# locals: (project:) %>

<ul>
  <%# projects  のみを入力にして、そこから必要な news を引いてレンダリングする  %>
  <% project.news.published.order(created_at: :desc).limit(3).each do |news| %>
    <li>
      <%= link_to news.title, project_news_path(project, news) %>
    </li>
  <% end %>
</ul>
```

共通パーツをコンポーネントにする

- project を locals で受け取りそこからアソシエーションを辿る
  - AR::Relation なので .published.order(...).limit(...) と続けられる
- これが長いなと思ったら、モデルにメソッドを追加したり、ヘルパーに抽出したりする

このアプローチの Rails の気持ち

- コントローラのインスタンス変数は、ビューとの大事なインターフェースであり、絞るべき
  - HTML でも JSON でも両方で使うようなデータだけをロードする
  - サイドバー用のデータはコントローラではロードしない
- 共通コンポーネントは partial を使ってコンポーネントを作る
  - partial で AR のメソッドチェーンを使ってデータを取得するのもあり
  - AR の lazy 評価により、結果的に SQL 発行タイミングがビューになるのはわざとっぽい

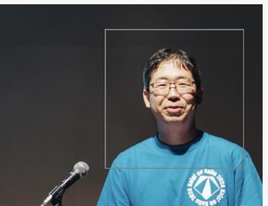
余談：viewcomponent gem

- この方向性とコンフリクトしないので、印象は悪くはない
- 「表示のための Ruby ロジック」の置き場として helper があるけど、もう少し整頓したくなった時に選択肢になりそう
- いっぽうで「分けてみようかな」の段階で導入すると侵襲度が大きい
  - gem も増えるわけだし
  - 1 コンポーネントごとに「専用のディレクトリ +Ruby ソースファイルとテンプレート」 vs. 「 partial テンプレート一個」
- せっかく個別に導入できるわけで、よほど複雑なやつだけとかが良いかな

- ActiveRecord::Relation を活用する
- コントローラのインスタンス変数を減らす
- 一部の includes はビューで指定する

一部の includes はビューで指定する

- 前節は「アクションで表示したい内容の本分じゃないもの」を
  コントローラでロードするのをやめようという主張だった
- 今度は「そのアクションの本分のデータ群でも、特定の表現でのみ
  必要な関連のロードはビューの責務である」という主張です



どんどん思想が強くなるよ！

```ruby
class IssuesController < ApplicationController
  def index
    @project = current_person.projects.find_by!(name: params[:pj_name])
    # HTML 表示用の関連データまでコントローラでロード
    @issues = @project.issues
                      .includes(creator: :avatar)  # HTML 用
                      .order(updated_at: :desc)
  end
end
```

```erb
# app/views/issues/index.html.erb
<% @issues.each do |issue| %>
  <div>
    <%= image_tag issue.creator.avatar.url %>
    <%= issue.title %>
  </div>
<% end %>
```

こういうのを

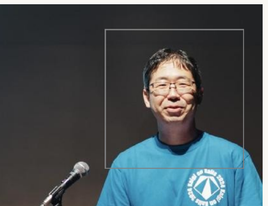
```ruby
class IssuesController < ApplicationController
  def index
    @project = current_person.projects.find_by!(name: params[:pj_name])
    #  コントローラは最小限。 AR::Relation のまま渡す
    @issues = @project.issues.order(updated_at: :desc)
 end
end
```

```ruby
# app/views/issues/index.html.erb
<% @issues.includes(creator: :avatar).each do |issue| %>
  <div>
    <%= image_tag issue.creator.avatar.url %>
    <%= issue.title %>
  </div>
<% end %>
```

こうしたい

表示する関連先を判断するのはビューの責務

- issue の起案者である person のアイコン画像を出すかどうか、を判断するのはビューの責務である
  - HTML 表現においては出すが、 JSON 表現では出さない、など
- @issues.includes(creator: :avatar) をどこで呼び出すか :
  - コントローラで呼ぶと全てのテンプレートで同じ includes が適用されてしまう
  - ビューで呼ぶと、各表現で必要なものを include できる



昔は使っていたけどビューを変えるうちに不要になった include が残っていて、忘れた頃にパフォーマンス悪化の原因になった、みたいな経験ある人も多いかと思います。私もいっぱいあります w

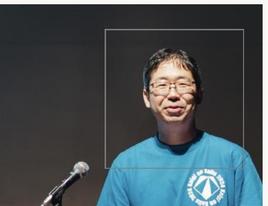
“

設計決定際、共有知識　　、
　　知識移動距離注意払必要。（略）
最終的、結合影響注意。近接
　　互　　影響与。

Vlad Khononov 「ソフトウェア設計の結合バランス」
第 8 章 距離

やはり ActiveRecord::Relation

- AR::Relation のクエリ発行が lazy であるからこそ、ビューに渡したあとで includes を追加指定できる
- やはりこの柔軟性を使い倒すのが Rails の気持ちだろう



とはいえ、すべての includes をビューでやる、というわけじゃないです。モデルのロジックで必要な先読みはモデルでやる。それはそう。

このアプローチの Rails の気持ち

- 前節とほぼ同じ。複数 Representation で使う処理だけにするとコントローラのコードが減る
- 使う場所の近くで `includes` すると、強い結合のあるコードを物理的に近く配置できる
- ミラクル🦄な `ActiveRecord::Relation` を使い倒す

Rails の気持ちを考えながら
コントローラとビューを整頓する

- ActiveRecord::Relation を活用する
- コントローラのインスタンス変数を減らす
- 一部の includes はビューで指定する

整頓後に残るコントローラの責務

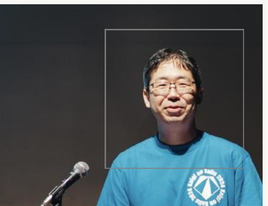
- Web リクエストのコンテキストから入力を取り出すところ
- 取り出した入力をファサードとなるモデルに渡して、ユースケースの処理を呼び出すところ
- その主処理の結果をインスタンス変数に入れて、ビューに渡すところ
- 処理結果に応じて render したり redirectto したりするところ

残った責務ももう少し整頓できる

- セッション情報からアクセス者取得、などには定番のイディオムがある
  - current_person のように ApplicationController で取得・メモ化したり
  - ActiveSupport::CurrentAttribute を使ったり
- もっと基本的にプライベートメソッド抽出してもよい。 Ruby なので
  - ビューでも呼びたければ helper_method 宣言などで拡張もできる

コントローラとビューを整頓する

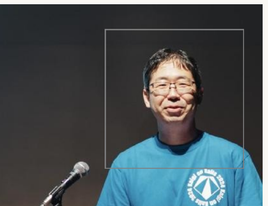
- これらの処理を抽出すると、コントローラのコードを十分に減らせそう
- 新しいライブラリやフレームワーク、アーキテクチャを導入したわけではない



それでも残る責務はあるので、 10 行 / メソッドルールなどの厳守はできないかもですが、見通しは良くなるはず

整頓は可愛くてふわふわした小さなリファクタリング

- この話を気に入ってくれても、一気に全部やろうとしないで⚠️
  - 危ないからね ...
- 1 コントローラの、 1 アクションに、 1tips だけを適用する
- それで、目の前のソフトウェアが少しずつ整頓されていく
- それもまた dynamic! な活動なのだと思います



と、 API などから Rails の気持ちを (勝手に) 読みとりつつ、アプリケーションがいい感じであれるように日々整頓して修復し続けるのが「私の Rails の話」です。地味ですけどもやってみると面白いですよ！