「技術負債にならない・間違えない」権限管理の設計と実装

Kaigi on Rails 2025

@naro143 （ Yusuke Ishimi ）

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

Yusuke Ishimi

株式会社プレックス
テックリード

@naro143　　@naro143

持ち帰っていただきたいこと

- 権限管理の重要性とアンチパターンの理解
- 権限管理の要素を適切に分割することで技術負債と間違いが減らせる
- 具体的な実装例（今後の議論のきっかけになれば幸いです）

目次

- 権限管理の重要性
- よくある実装とアンチパターンと対処法
- 権限管理の要素の整理（ Pundit を例に）
- 要素を適切に分割した Module の実装の解説
- 改善後の事業とサービスへの影響

Module のサンプルは GitHub で公開しています

権限管理

権限管理の例

- 運営アカウントだけ特殊なボタンが表示される
- 管理者アカウントだけメンバーを招待できる
- 追加プランの契約ユーザーだけ機能が増える

業務支援 SaaS だと ...

- 管理者アカウントだけお金の情報（契約金額や給与）が見れる
- 外部アカウント（業務委託）は担当のプロジェクトの情報だけ見れる（取引先は見れない）

権限管理の例

- 運営アカウントだけ特殊なボタンが表示される
- 管理者アカウントだけメンバーを招待できる
- 追加プランの契約コードだけ機能が増える

権限管理のミス

業務支援 SaaS だと ...

- 管理者アカウントだけお金の情報（契約金額や給与）が見れる
- 外部アカウント（業務委託）は担当のプロジェクトの情報だけ見れる

権限管理の例

- 運営アカウントだけ特殊なボタンが表示される
- 管理者アカウントだけメンバーを招待できる
- 追加プランの契約ユーザーだけ機能が増える

業務支援 SaaS だと …

- 管理者アカウントだけお金の情報（契約金額や給与）が見れる
- 外部アカウント（業務委託）は担当のプロジェクトの情報だけ見れる

給与を公開しちゃった
取引先を公開しちゃった

権限管理の例

- 運営アカウントだけ特殊なボタンが表示される
- 管理者アカウントだけメンバーを招待できる
- 追加プランの契約ユーザーだけ機能が増える

業務支援 SaaS だと…

- 管理者アカウントだけお金の情報（契約金額や給与）が見れる
- 外部アカウント（業務委託）は担当のプロジェクトの情報だけ見れる

サービスへの信頼が下がる
事業への損失が大きい

権限管理の例

- 運営アカウントだけ特殊なボタンが表示される
- 管理者アカウントだけメンバーを招待できる
- 追加プランの契約ユーザーだけ機能が増える

権限管理のミスは許されない

業務支援 SaaS だと ...

- 管理者アカウントだけお金の情報（契約金額や給与）が見れる
- 外部アカウント（業務委託）は担当のプロジェクトの情報だけ見れる

レビューする気持ちで見てください

よくある実装①

```ruby
class ProjectsController < ApplicationController
  def create
    unless current_user.admin?
      redirect_to root_path, alert: "権限が必要です"
      return
    end

    ...
  end
end
```

よくある実装②

```ruby
class ProjectsController < ApplicationController
  def create
    unless current_user.manager?
      redirect_to root_path, alert: "権限が必要です"
      return
    end

    ...
  end
end
```

よくある実装③

```ruby
class ProjectsController < ApplicationController
  def create
    unless current_user.super_admin?
      redirect_to root_path, alert: "権限が必要です"
      return
    end

    ...
  end
end
```

よくある実装③

```
1  class ProjectsController < ApplicationController
2    def create
3      unless current_user.super_admin?
4        redirect_to root_path, alert: '権限が必要です'
5        return
6      end
7
8      ...
9    end
10 end
```

なんだよ super_admin って ...

よくある実装③

```
1  class ProjectsController < ApplicationController
2    def create
3      unless current_user.super_admin?
4        redirect_to root_path, alert: "権限が必要です"
5        return
6      end
7
8      ...
9    end
10 end
```

そもそも admin? の判定は
アンチパターン

役割は変わり行く

事業やサービスの変化

事業

- ターゲットとする市場（業種）の変化
- ターゲットとする企業規模の変化
- ターゲットとするユーザーの変化

サービス

- 提供する機能の変化

事業やサービスの変化

事業

- ターゲットとする市場（業種）の変化
- ターゲットとする企業規模の変化
- ターゲットとするユーザーセットの変化

サービス

- 提供する機能の変化

建設業と運送業の
管理者は同じ業務と責務?

事業やサービスの変化

事業

- ターゲットとする市場（業種）の変化
- ターゲットとする企業規模の変化
- ターゲットとするユーザーの変化

サービス

- 提供する機能の変化

10 名と 100 名の企業の管理者は同じ業務と責務?

事業やサービスの変化

事業

- ターゲットとする市場（業種）の変化
- ターゲットとする企業規模の変化
- ターゲットとするユーザーの変化

同じ admin?

サービス

- 提供する機能の変化

事業やサービスの変化

事業

- ターゲットとする市場（業種）の変化
- ターゲットとする企業規模の変化
- ターゲットとするユーザーの変化

サービス

- 提供する機能の変化

役割に依存した判定は
アンチパターン

事業やサービスの変化

事業

- ターゲットとする市場・業種の変化
- ターゲットとする企業規模の変化
- ターゲットとするユーザーの変化

サービス

- 提供する機能の変化

役割の種類も
提供する機能も
増えていく

事業やサービスの変化

事業

- ターゲットとする市場（業種）の変化
- ターゲットとする企業規模の変化
- ターゲットとするユーザーの変化

サービス

- 提供する機能の変化

だから、権限管理は複雑になる
いつか、技術負債になる

リファクタリング

よくある実装①

```ruby
class ProjectsController < ApplicationController
  def create
    unless current_user.admin?
      redirect_to root_path, alert: "権限が必要です"
      return
    end

    ...
  end
end
```

よくある実装①

```
1  class ProjectsController < Ap
2    def create
3      unless current_user.admin?
4        redirect_to root_path, alert: "権限が必要です"
5        return
6      end
7
8      ...
9    end
10 end
```

プロジェクトを作成できるのか?

よくある実装①

```
1  class ProjectsController < Ap
2    def create
3      unless current_user.admin?
4        redirect_to root_path, alert: "権限が必要です"
5        return
6      end
7
8      ...
9    end
10 end
```

プロジェクトを作成できるのか?

わからない…

admin? からの卒業

- 役割に依存した実装は、権限が暗黙的になる
  - レビュワーは、常に役割の権限を知らないといけない
  - 判定箇所が散らばり、どのような権限が定義されているのかわからない
- 役割に依存した実装は、変化に弱い
  - 役割の権限が変化した際に、多くの判定箇所を見ないといけない
  - 権限の整合性を確認するために、多くの判定箇所を見ないといけない

役割でなく、権限に依存する

```
1  class ProjectsController < ApplicationController
2    def create
3      unless current_user.can_create_project?
4        redirect_to root_path, alert: "権限が必要です"
5        return
6      end
7
8      ...
9    end
10 end
```

役割でなく、権限に依存する

```
1  class ProjectsController < Ap
2    def create
3      unless current_user.can_create_project?
4        redirect_to root_path, alert: "権限が必要です"
5        return
6      end
7
8      ...
9    end
10 end
```

プロジェクトを作成できるのか?

役割でなく、権限に依存する

```ruby
class User < ApplicationRecord
  ...

  enum :role, [:admin, :manager, :normal, :external]

  def can_create_project?
    admin? || manager?
  end
end
```

役割でなく、権限に依存する

```
1  class User < ApplicationRecord
2    ...
3
4    def can_create_project?
5      admin? || manager?
6    end
7  end
```

これだけでも、だいぶ良くなる

役割でなく、権限に依存する

役割でなく、権限に依存する

```
1  class User < ApplicationRecord
2    ...
3
4    def can_create_project?
5      admin? || manager?
6    end
7  end
```

より深みへ

権限管理で大事なこと

間違えない

1. 実装で間違えない
    a. 追加、変更をするとき
2. 利用で間違えない
    a. 処理の中で判定をするとき
3. 理解で間違えない
    a. コードリーディングのとき
    b. お問い合わせの回答のとき

間違えない

1. 実装で間違えない
    a. 追加、変更をするとき
2. 利用で間違えない
    a. 処理の中で判定をするとき
3. 理解で間違えない
    a. コードリーディングのとき
    b. お問い合わせの回答のとき

権限はお問い合わせが多い

RBAC と ABAC

RBAC と ABAC

RBAC (Role-Based Access Control)

- 運営アカウント
- 管理者アカウント
- 外部アカウント

ABAC (Attribute-Based Access Control)

- 作成者
- 担当者

RBAC と ABAC

RBAC (Role-Based Access Control)

- 運営アカウント
- 管理者アカウント
- 外部アカウント

ABAC (Attribute-Based Access Control)

- 作成者
- 担当者

権限管理を整理する

具体例で整理

- 「プロジェクトの更新は、管理者か担当者ならできる」

具体例で整理

- 「プロジェクトの更新は、管理者か担当者ならできる」

- 「対象の、操作は、役割か条件ならできる」

具体例で整理

- 「プロジェクトの更新は、管理者か担当者ならできる」
- 「対象の、操作は、役割か条件ならできる」
- 「 Model の、 CRUD は、 Role か Scope ならできる」

Pundit で実装

Pundit

```
1  class ProjectPolicy < ApplicationPolicy
2    def update?
3      user.admin? || user.assignee_projects.exists?(project: record)
4    end
5
6    ...
7  end
```

Pundit

```
1  class ProjectPolicy < ApplicationPolicy
2    def update?
3      user.admin? || user.assignee_projects.exists?(project: record)
4    end
5
6    ...
7  end
```

まだパッとわかる

Pundit　プロジェクトの更新は、管理者かマネージャーかつ担当者か作成者ならできる

外部アカウントはどんな場合もできない

Pundit

```ruby
class ProjectPolicy < ApplicationPolicy
  def update?
    return false if user.external?

    user.admin? ¦¦ (user.manager? && (user.assignee_projects.exists?(project: record) ¦¦ user.author_projects.exists?(project: record)))
  end

  ...
end
```

Pundit

```ruby
class ProjectPolicy < ApplicationPolicy
  def update?
    return false if user.external?

    user.admin? || (user.manager? && (user.assignee_projects.exists?(project: record) || user.author_projects.exists?(project: record)))
  end

  ...
end
```

「通常ユーザーは、プロジェクトの更新ができますか？」と
聞かれたら回答するのに何秒必要ですか?

Pundit

```
1  class ProjectPolicy < ApplicationPolicy
2    def update?
3      return false if user.external?
4
5      user.admin? || (user.manager? && (user.assigned_projects.exists?(project: record) || user.author_projects.exists?(project: record)))
6    end
7
8    ...
9  end
```

全部読みましたね?
5~10 秒くらいかな

「通常ユーザーは、プロジェクトの更新ができますか? 」と
聞かれたら回答するのに何秒必要ですか?

Pundit

```
1  class ProjectPolicy < ApplicationPolicy
2    def update?
3      return false if user.external?
4
5      user.admin? || user.manager? && ... || user.authored_projects.exists?(project: record)
6    end
7
8    ...
9  end
```

実際のサービスでは
より判定は複雑になる

「通常ユーザーは、プロジェクトの更新ができますか？」と
聞かれたら回答するのに何秒必要ですか?

Pundit


どうしてパッとわからないのか
user.admin? || (user.manager? && (user.assignee_projects.exists?(project: record) || user.author_projects.exists?(project: record)))
end
...
end
「通常ユーザーは、プロジェクトの更新ができますか？」と
聞かれたら回答するのに何秒必要ですか?

具体例で整理

- 「プロジェクトの更新は、管理者か担当者ならできる」
- 「対象の、操作は、役割か条件ならできる」
- 「 Model の、 CRUD は、 Role か Scope ならできる」

Pundit


対象
操作
class ProjectPolicy < ApplicationPolicy
def update?
return false if user.external?
user.admin? || (user.manager? && (user.assignee_projects.exists?(project: record) || user.author_projects.exists?(project: record)))
end
...
end
役割
条件

RBAC と ABAC

RBAC (Role-Based Access Control)

- 運営アカウント
- 管理者アカウント
- 外部アカウント

ABAC (Attribute-Based Access Control)

- 作成者
- 担当者

Pundit


対象
操作
1 class ProjectPolicy < ApplicationPolicy
2 def update?
3 return false if user.external?
4
5 user.admin? || (user.manager? && (user.assignee_projects.exists?(project: record) || user.author_projects.exists?(project: record)))
6 end
7
8 ...
9 end
役割
条件

Pundit


対象
操作
1 class ProjectPolicy < ApplicationPolicy
2 def update?
3 return false if user.external?
4
5 user.admin? || (user.manager? && (user.assignee_projects.exists?(project: record) || user.author_projects.exists?(project: record)))
6 end
7
8 ...
9 end
役割
条件

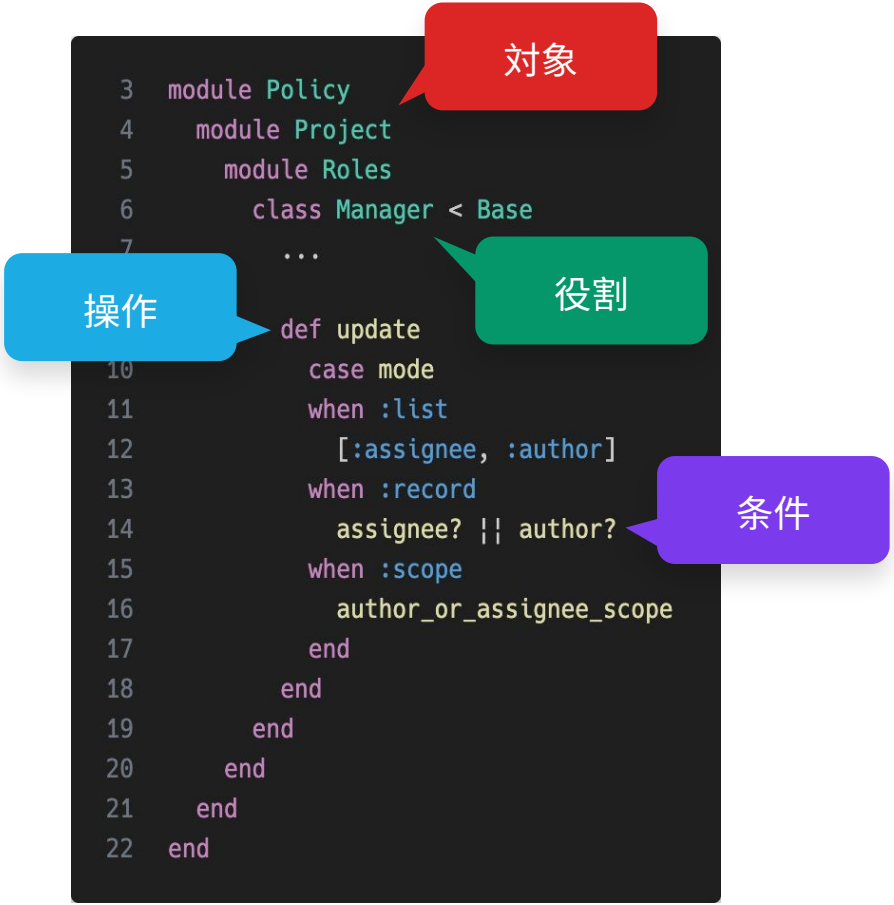
Pundit

Pundit


対象
操作
1 class ProjectPolicy < ApplicationPolicy
2 def update?
3 return false if user.external?
4
5 user.admin? || (user.manager? && (user.assignee_projects.exists?(project: record) || user.author_projects.exists?(project: record)))
6 end
7
8 ...
9 end
つくりました
役割
条件

Module の設計と実装

```
3  module Policy
4    module Project
5      module Roles
6        class Manager < Base
7          ...
8
9          def update
10           case mode
11           when :list
12             [:assignee, :author]
13           when :record
14             assignee? || author?
15           when :scope
16             author_or_assignee_scope
17           end
18         end
19       end
20     end
21   end
22 end
```


対象
module Policy
module Project
module Roles
class Manager < Base
...
役割
操作
def update
case mode
when :list
[:assignee, :author]
when :record
assignee? || author?
条件
when :scope
author_or_assignee_scope
end
end
end
end
end

Project / Manager

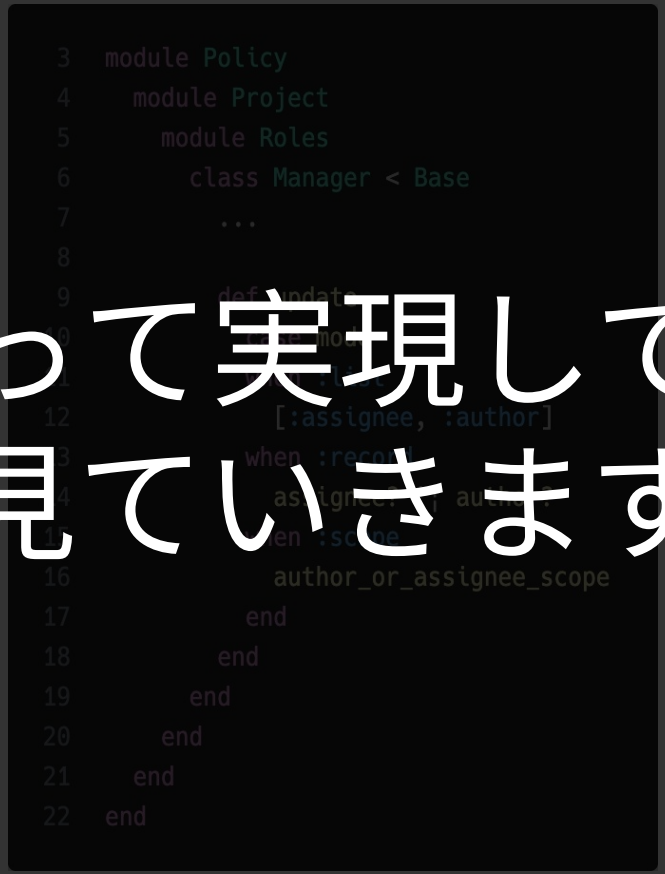
```
def update
```

```
assignee? || author?
```

```
module Policy
  module Project
    module Roles
      class Manager < Base
        ...

        def update
          case mode
          when :list
            [assignee?, author?]
          when :read
            assignee? || author?
          when :scope
            author_or_assignee_scope
          end
        end
      end
    end
  end
end
```

なんとなくわかった人🙋‍♂️

英語と論理演算がわかれば🤦

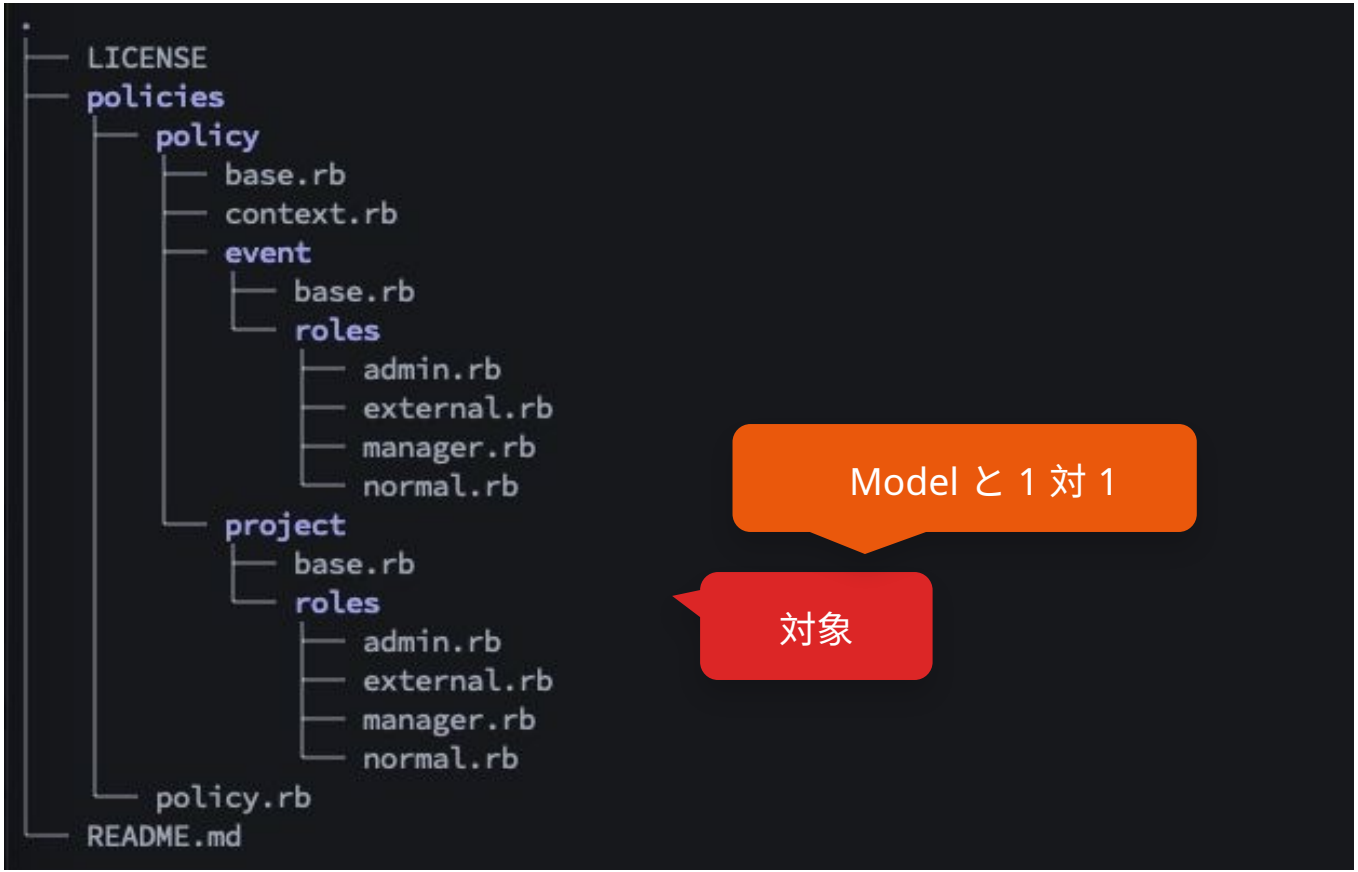


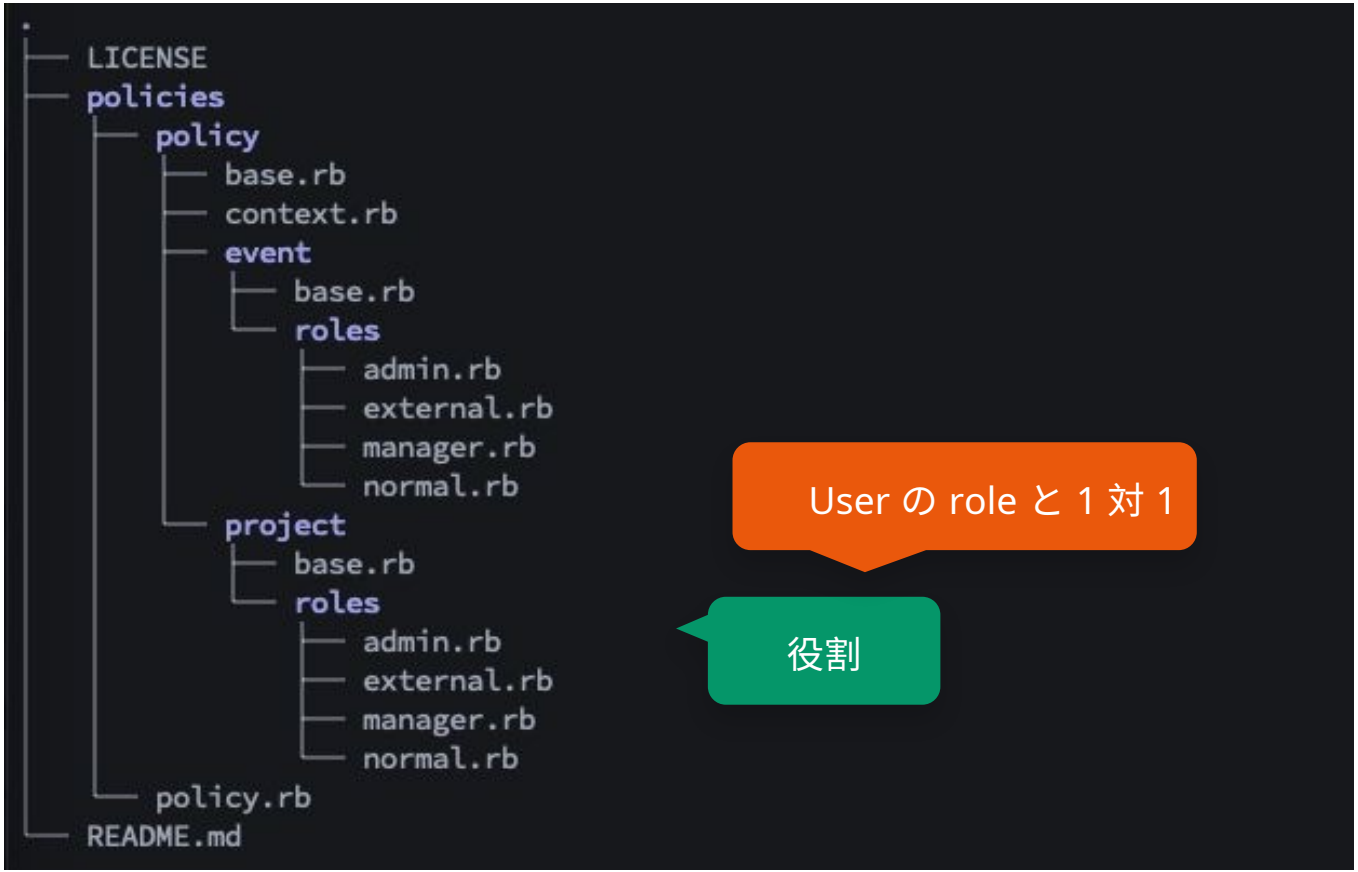
どうやって実現しているか見ていきます

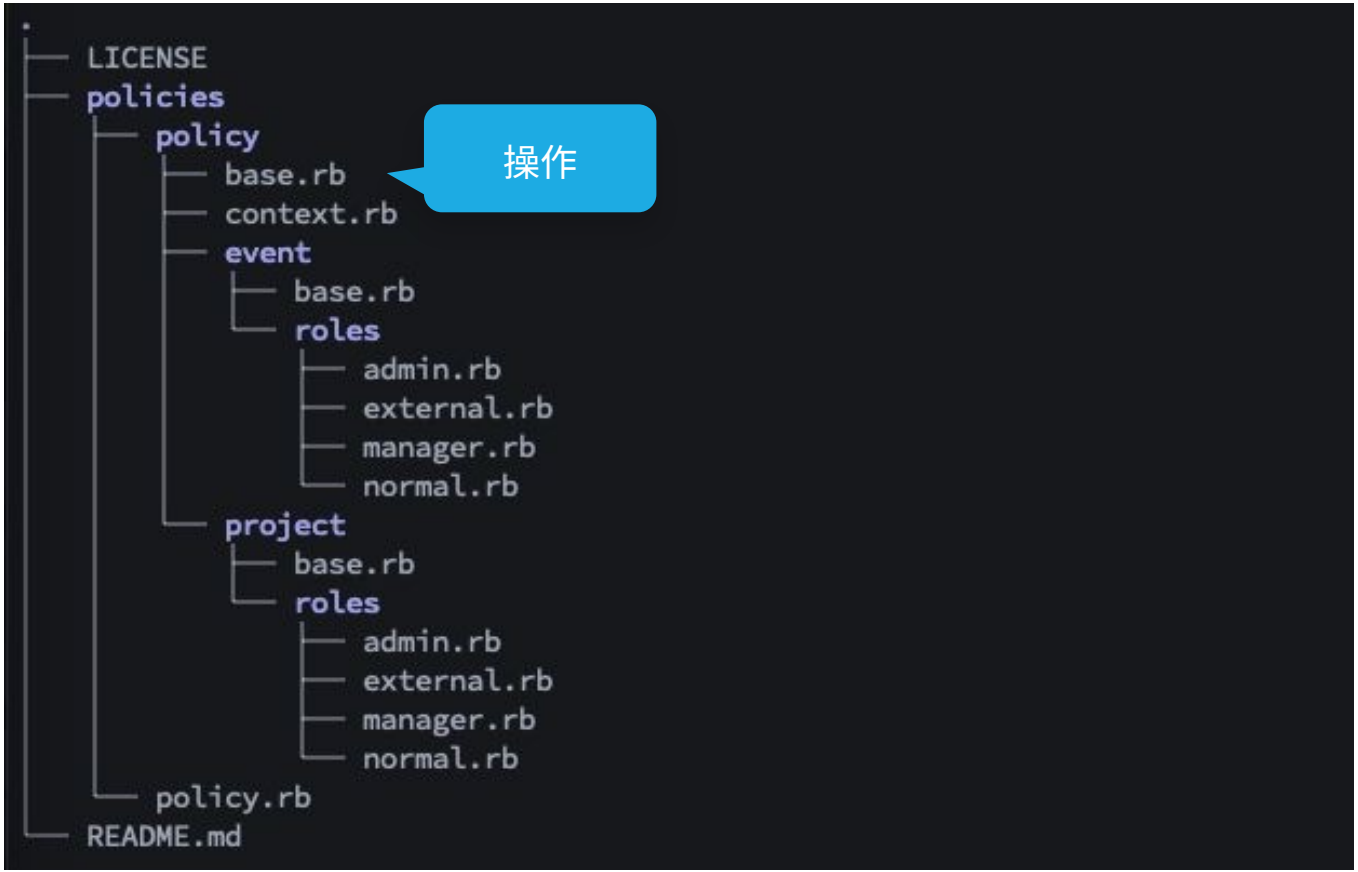
ざっくり概要

1. 対象と役割から権限のクラスを特定
2. 判定モードの指定
3. 操作名の関数を実行
4. 条件の判定

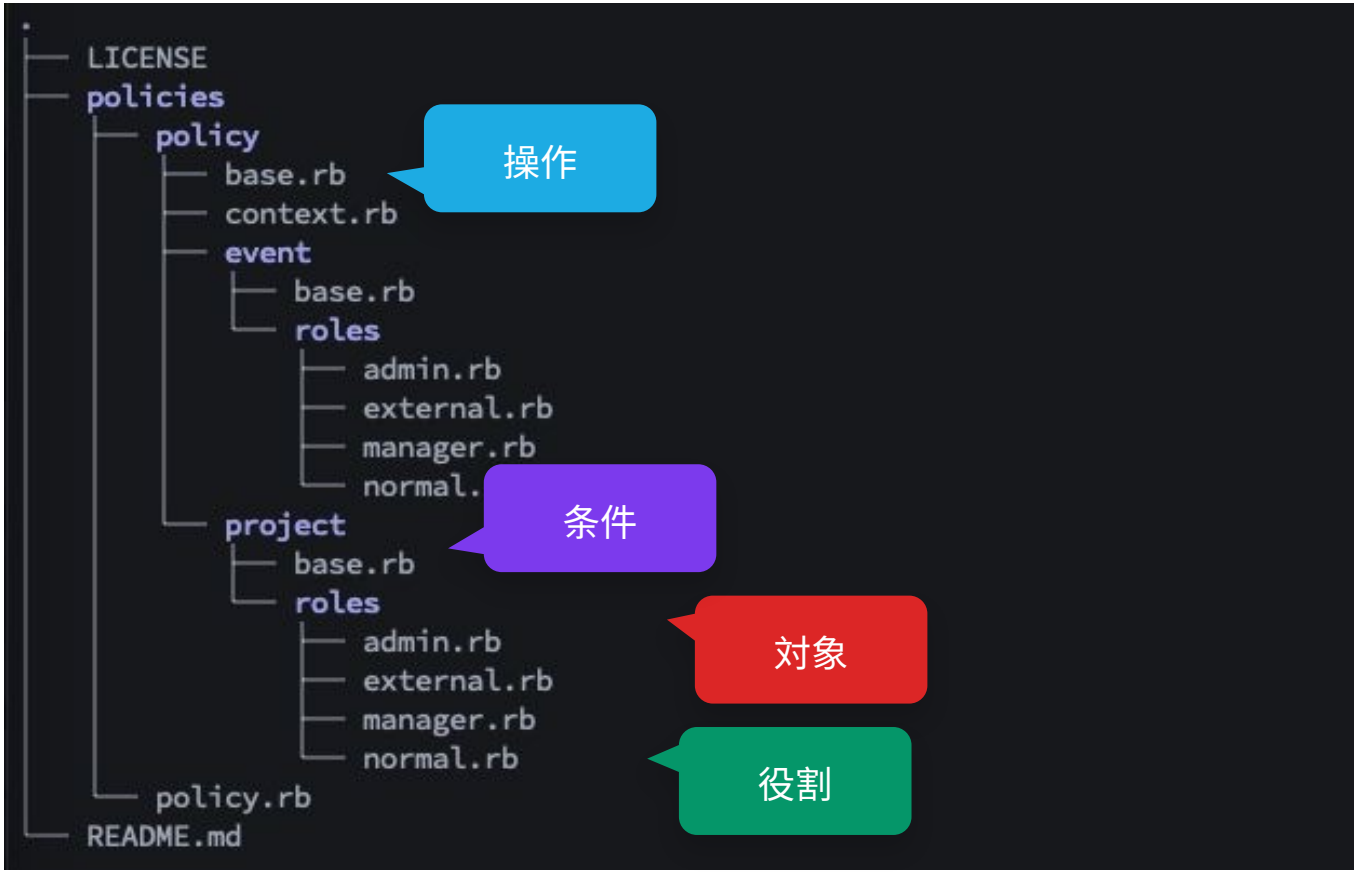

.
LICENSE
policies
policy
base.rb
context.rb
event
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
project
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
policy.rb
README.md


.
LICENSE
policies
policy
base.rb
context.rb
event
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
project
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
policy.rb
README.md
Model と 1 対 1
対象


.
LICENSE
policies
policy
base.rb
context.rb
event
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
project
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
policy.rb
README.md
User の role と 1 対 1
役割

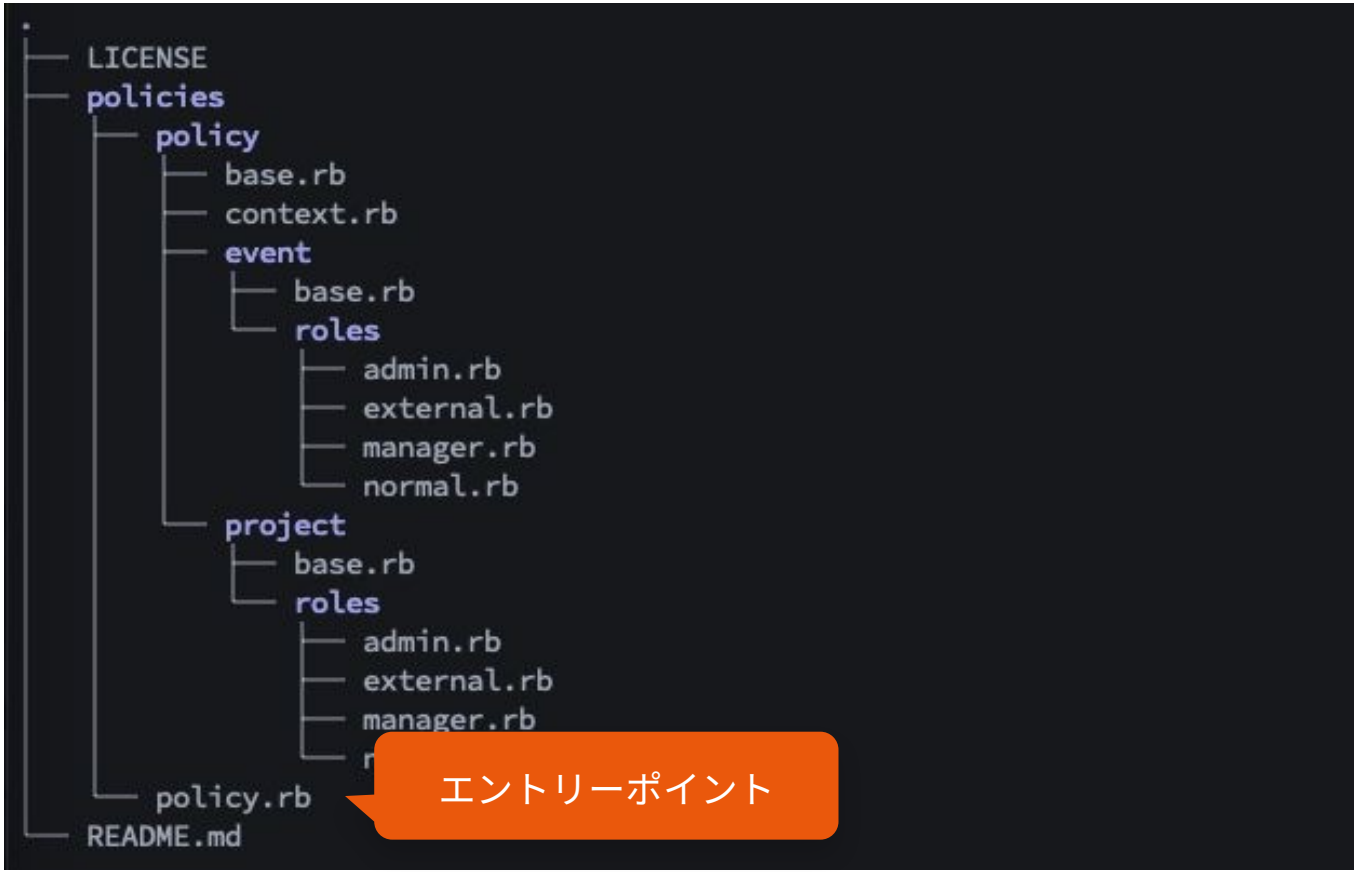

LICENSE
policies
policy
base.rb
context.rb
event
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
project
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
policy.rb
README.md
操作


.
LICENSE
policies
policy
base.rb
context.rb
event
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
project
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
policy.rb
README.md
条件


.
LICENSE
policies
policy
base.rb
context.rb
event
base.rb
roles
admin.rb
external.rb
manager.rb
normal.
project
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
policy.rb
README.md
操作
条件
対象
役割

ここまでの整理

- 対象ごとにディレクトリを作成、役割ごとにファイルを作成
  - 役割と条件を分ける
- 権限のクラスの特定でメタプログラミングを活用
  - 権限の変更が容易
- 対象の基底クラスで CRUD 以外の操作を追加
  - 対象ごとに異なる操作の拡張に対応
- 対象の基底クラスで条件を定義、条件を論理演算で使用
  - 判定は英語と論理演算がわかれば十分
  - 対象によって異なる条件の判定に対応


.
LICENSE
policies
policy
base.rb
context.rb
event
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
project
base.rb
roles
admin.rb
external.rb
manager.rb
policy.rb
README.md
エントリーポイント

```ruby
module Policy
  ...

  def self.authorize(user, record, action)
    context = Context.new(user:)
    context.authorize(record, action)
  end

  def self.authorize_scope(user, scope, action)
    context = Context.new(user:)
    context.authorize_scope(scope, action)
  end

  def self.permissions(user)
    context = Context.new(user:)
    context.permissions
  end
end
```

使い方

1. レコードに対して権限があるかを判定する（ record モード）

```
1  project = Project.find(1)
2
3  readable_project = Policy.authorize(current_user, project, :read)
```

対象と操作が明示されている👍

使い方

2. 権限があるレコードのみに絞り込む（ scope モード）

```
1  projects = Project.all
2
3  readable_projects = Policy.authorize_scope(current_user, projects, :read)
```

対象と操作が明示されている👍

使い方

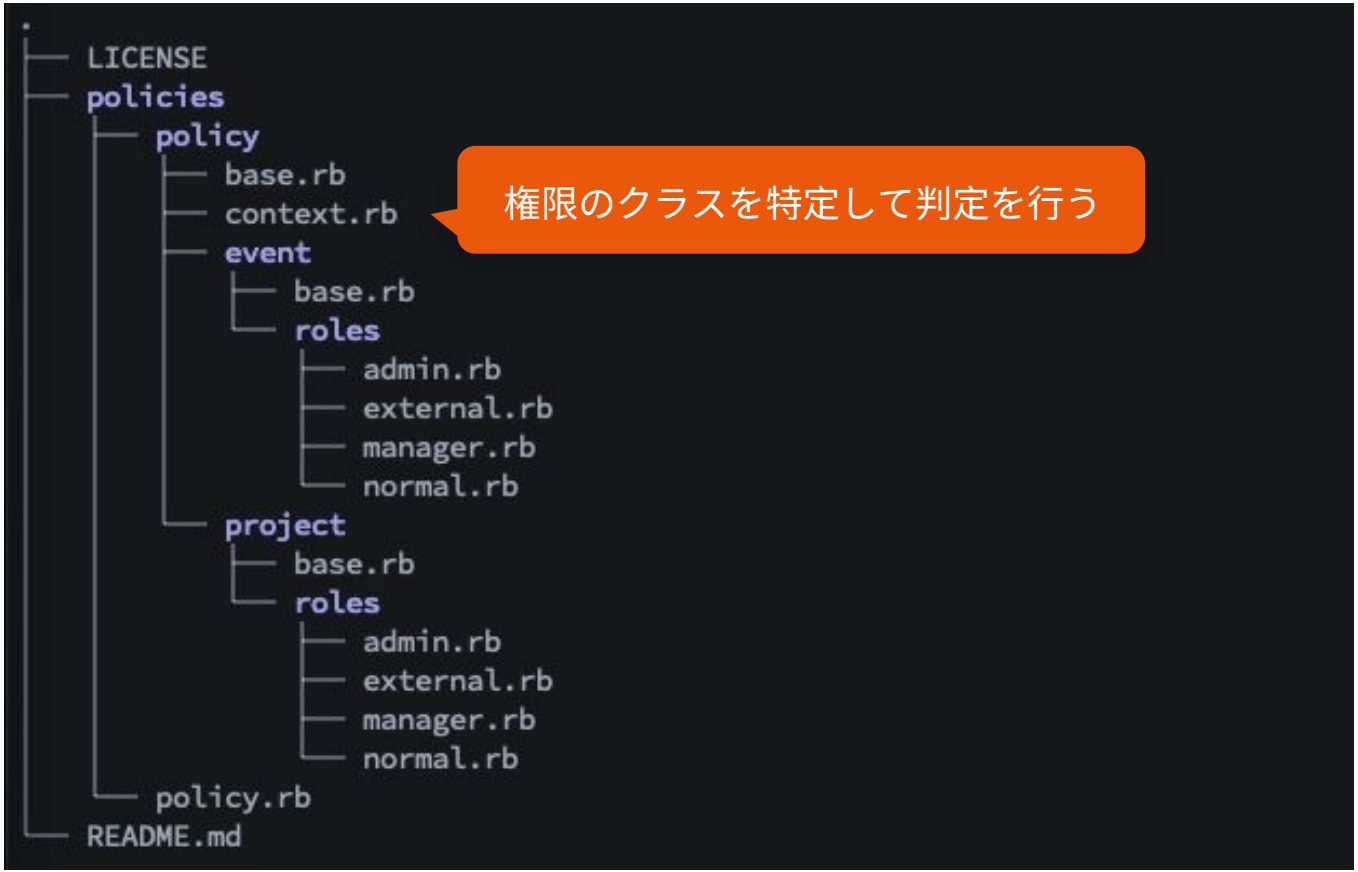
3. 権限の一覧を取得する（ list モード）

```
1  permissions = Policy.permissions(current_user)
2
3  # => JSON
4  # {
5  #   "project": {
6  #     "read": true
7  #     "create": false,
8  #     "update": false,
9  #     "delete": false,
10 #   }
11 # }
```

主にクライアントに渡して利用する

```ruby
module Policy
  ...

  def self.authorize(user, record, act
    context = Context.new(user:)
    context.authorize(record, action)
  end

  def self.authorize_scope(user, scope, action)
    context = Context.new(user:)
    context.authorize_scope(scope, action)
  end

  def self.permissions(user)
    context = Context.new(user:)
    context.permissions
  end
end
```

Context.new して関数を実行


.
LICENSE
policies
policy
base.rb
context.rb
event
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
project
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
policy.rb
README.md
権限のクラスを特定して判定を行う

```ruby
module Policy
  class Context
    ...

    def authorize(record, action)
      raise(ArgumentError, 'record cannot be nil') unless record

      policy = policy_class(user, record.class).new(user:, record:, mode: :record)

      raise(NotAuthorizedError, policy:, action:) unless policy.public_send(action.to_sym)

      policy.record
    end

    def authorize_scope(scope, action)
      raise(ArgumentError, 'scope cannot be nil') unless scope
      raise(ArgumentError, 'scope must be ActiveRecord::Relation') unless scope.is_a?(ActiveRecord::Relation)

      policy = policy_class(user, scope.klass).new(user:, scope:, mode: :scope)

      policy.public_send(action.to_sym)
    end

    def permissions
      list = {}

      policy_constants.each do |constant|
        klass = policy_class(user, constant)
        policy = klass.new(user:, mode: :list)

        constant_result = klass.public_instance_methods(false).each_with_object({}) do |method, result|
          result[method.to_sym] = policy.public_send(method)
        end

        list[constant.to_s.underscore.to_sym] = constant_result
      end

      list
    end

    ...

    def policy_class(user, record_class)
      role = user.role.camelize

      "Policy::#{record_class}::Roles::#{role}".safe_constantize || raise(NotDefinedError, record_class:, role:)
    end
  end
end
```

```ruby
module Policy
  class Context
    ...

    def authorize(record, action)
      raise(ArgumentError, 'record cannot be nil') unless record

      policy = policy_class(user, record.class).new(user:, record:, mode: :record)

      raise(NotAuthorizedError, policy:, action:) unless policy.public_send(action.to_sym)

      policy.record
    end

    def authorize_scope(scope, action)
      raise(ArgumentError, 'scope cannot be nil') unless scope
      raise(ArgumentError, 'scope must be ActiveRecord::Relation') unless scope.is_a?(ActiveRecord::Relation)

      policy = policy_class(user, scope.klass).new(user:, scope:, mode: :scope)

      policy.public_send(action.to_sym)
    end

    def permissions
      list = {}

      policy_constants.each do |constant|
        klass = policy_class(user, constant)
        policy = klass.new(user:, mode: :list)

        constant_result = klass.public_instance_methods(false).each_with_object({}) do |method, result|
          result[method.to_sym] = policy.public_send(method)
        end

        list[constant.to_s.underscore.to_sym] = constant_result
      end

      list
    end

    ...

    def policy_class(user, record_class)
      role = user.role.camelize

      "Policy::#{record_class}::Roles::#{role}".safe_constantize || raise(NotDefinedError, record_class:, role:)
    end
  end
end
```

対象と役割から
権限のクラスを特定する

```
63  def policy_class(user, record_class)
64    role = user.role.camelize
65
66    "Policy::#{record_class}::Roles::#{role}".safe_constantize || raise(NotDefinedError, record_class:, role:)
67  end
```

対象と役割から
権限のクラスを特定する

```ruby
63  def policy_class(user, record_class)
64    role = user.role.camelize
65
66    "Policy::#{record_class}::Roles::#{role}".safe_constantize || raise(NotDefinedError, record_class:, role:)
67  end
```

```ruby
3   module Policy
4     module Project
5       module Roles
6         class Manager < Base
7           ...
8
9           def update
```

```ruby
module Policy
  class Context
    ...

    def authorize(record, action)
      raise(ArgumentError, 'record cannot be nil') unless record

      policy = policy_class(user, record.class).new(user:, record:, mode: :record)

      raise(NotAuthorizedError, policy:, action:) unless policy.public_send(action.to_sym)

      policy.record
    end

    def authorize_scope(scope, action)
      raise(ArgumentError, 'scope cannot be nil') unless scope
      raise(ArgumentError, 'scope must be ActiveRecord::Relation') unless scope.is_a?(ActiveRecord::Relation)

      policy = policy_class(user, scope.klass).new(user:, scope:, mode: :scope)

      policy.public_send(action.to_sym)
    end

    def permissions
      list = {}

      policy_constants.each do |constant|
        klass = policy_class(user, constant)
        policy = klass.new(user:, mode: :list)

        constant_result = klass.public_instance_methods(false).each_with_object({}) do |method, result|
          result[method.to_sym] = policy.public_send(method)
        end

        list[constant.to_s.underscore.to_sym] = constant_result
      end

      list
    end

    ...

    def policy_class(user, record_class)
      role = user.role.camelize

      "Policy::#{record_class}::Roles::#{role}".safe_constantize || raise(NotDefinedError, record_class:, role:)
    end
  end
end
```

使い方

1. レコードに対して権限があるかを判定する（ record モード）

```
1  project = Project.find(1)
2
3  readable_project = Policy.authorize(current_user, project, :read)
```

対象と操作が明示されている👍

```
11  def authorize(record, action)
12    raise(ArgumentError, 'record cannot be nil') unless record
13
14    policy = policy_class(user, record.class).new(user:, record:, mode: :record)
15
16    raise(NotAuthorizedError, policy:, action:) unless policy.public_send(action.to_sym)
17
18    policy.record
19  end
```

権限のクラスを
record モードで new

```
11  def authorize(record, action)
12    raise(ArgumentError, 'record cannot be nil') unless record
13
14    policy = policy_class(user, record.class).new(user:, record:, mode: :record)
15
16    raise(NotAuthorizedError, policy:, action:) unless policy.public_send(action.to_sym)
17
18    policy.record
19  end
```

権限のクラスを
record モードで new

操作名の関数を実行

```ruby
module Policy
  class Context
    ...

    def authorize(record, action)
      raise(ArgumentError, 'record cannot be nil') unless record

      policy = policy_class(user, record.class).new(user:, record:, mode: :record)

      raise(NotAuthorizedError, policy:, action:) unless policy.public_send(action.to_sym)

      policy.record
    end

    def authorize_scope(scope, action)
      raise(ArgumentError, 'scope cannot be nil') unless scope
      raise(ArgumentError, 'scope must be ActiveRecord::Relation') unless scope.is_a?(ActiveRecord::Relation)

      policy = policy_class(user, scope.klass).new(user:, scope:, mode: :scope)

      policy.public_send(action.to_sym)
    end

    def permissions
      list = {}

      policy_constants.each do |constant|
        klass = policy_class(user, constant)
        policy = klass.new(user:, mode: :list)

        constant_result = klass.public_instance_methods(false).each_with_object({}) do |method, result|
          result[method.to_sym] = policy.public_send(method)
        end

        list[constant.to_s.underscore.to_sym] = constant_result
      end

      list
    end

    ...

    def policy_class(user, record_class)
      role = user.role.camelize

      "Policy::#{record_class}::Roles::#{role}".safe_constantize || raise(NotDefinedError, record_class:, role:)
    end
  end
end
```

使い方

2. 権限があるレコードのみに絞り込む（ scope モード）

```
1  projects = Project.all
2
3  readable_projects = Policy.authorize_scope(current_user, projects, :read)
```

対象と操作が明示されている👍

```
21  def authorize_scope(scope, action)
22    raise(ArgumentError, 'scope cannot be nil') unless scope
23    raise(ArgumentError, 'scope must be ActiveRecord::Relation') unless scope.is_a?(ActiveRecord::Relation)
24
25    policy = policy_class(user, scope.klass).new(user:, scope:, mode: :scope)
26
27    policy.public_send(action.to_sym)
28  end
```

権限のクラスを
scope モードで new

```ruby
21  def authorize_scope(scope, action)
22    raise(ArgumentError, 'scope cannot be nil') unless scope
23    raise(ArgumentError, 'scope must be ActiveRecord::Relation') unless scope.is_a?(ActiveRecord::Relation)
24
25    policy = policy_class(user, scope.klass).new(user:, scope:, mode: :scope)
26
27    policy.public_send(action.to_sym)
28  end
```

操作名の関数を実行

権限のクラスを
scope モードで new

```ruby
module Policy
  class Context
    ...

    def authorize(record, action)
      raise(ArgumentError, 'record cannot be nil') unless record

      policy = policy_class(user, record.class).new(user:, record:, mode: :record)

      raise(NotAuthorizedError, policy:, action:) unless policy.public_send(action.to_sym)

      policy.record
    end

    def authorize_scope(scope, action)
      raise(ArgumentError, 'scope cannot be nil') unless scope
      raise(ArgumentError, 'scope must be ActiveRecord::Relation') unless scope.is_a?(ActiveRecord::Relation)

      policy = policy_class(user, scope.klass).new(user:, scope:, mode: :scope)

      policy.public_send(action.to_sym)
    end

    def permissions
      list = {}

      policy_constants.each do |constant|
        klass = policy_class(user, constant)
        policy = klass.new(user:, mode: :list)

        constant_result = klass.public_instance_methods(false).each_with_object({}) do |method, result|
          result[method.to_sym] = policy.public_send(method)
        end

        list[constant.to_s.underscore.to_sym] = constant_result
      end

      list
    end

    ...

    def policy_class(user, record_class)
      role = user.role.camelize

      "Policy::#{record_class}::Roles::#{role}".safe_constantize || raise(NotDefinedError, record_class:, role:)
    end
  end
end
```

使い方

3. 権限の一覧を取得する（ list モード）

```ruby
permissions = Policy.permissions(current_user)

# => JSON
# {
#   "project": {
#     "read": true
#     "create": false,
#     "update": false,
#     "delete": false,
#   }
# }
```

主にクライアントに渡して利用する

```ruby
def permissions
  list = {}

  policy_constants.each do |constant|
    klass = policy_class(user, constant)
    policy = klass.new(user:, mode: :list)

    constant_result = klass.public_instance_methods(false).each_with_object({}) do |method, result|
      result[method.to_sym] = policy.public_send(method)
    end

    list[constant.to_s.underscore.to_sym] = constant_result
  end

  list
end

def policy_constants
  reject_constants = [:Base, :Context, :Error, :NotAuthorizedError, :NotDefinedError]

  Policy.constants.reject { |constant| reject_constants.include?(constant) }
end
```

Policy::{ 対象 } で定義されている
対象名を全て取得

```ruby
def permissions
  list = {}

  policy_constants.each do |constant|
    klass = policy_class(user, constant)
    policy = klass.new(user:, mode: :list)

    constant_result = klass.public_instance_methods(false).each_with_object({}) do |method, result|
      result[method.to_sym] = policy.public_send(method)
    end

    list[constant.to_s.underscore.to_sym] = constant_result
  end

  list
end

def policy_constants
  reject_constants = [:Base, :Context, :Error, :NotAuthorizedError, :NotDefinedError]

  Policy.constants.reject { |constant| reject_constants.include?(constant) }
end
```

権限のクラスを
list モードで new

Policy::{ 対象 } で定義されている
対象名を全て取得

```ruby
30  def permissions
31    list = {}
32
33    policy_constants.each do |constant|
34      klass = policy_class(user, constant)
35      policy = klass.new(user:, mode: :list)
36
37      constant_result = klass.public_instance_methods(false).each_with_object({}) do |method, result|
38        result[method.to_sym] = policy.public_send(method)
39      end
40
41      list[constant.to_s.underscore.to_sym] = constant_result
42    end
43
44    list
45  end
46
47  def policy_constants
48    reject_constants = [:Base, :Context, :Error, :NotAuthorizedError, :NotDefinedError]
49
50    Policy.constants.reject { |constant| reject_constants.include?(constant) }
51  end
```

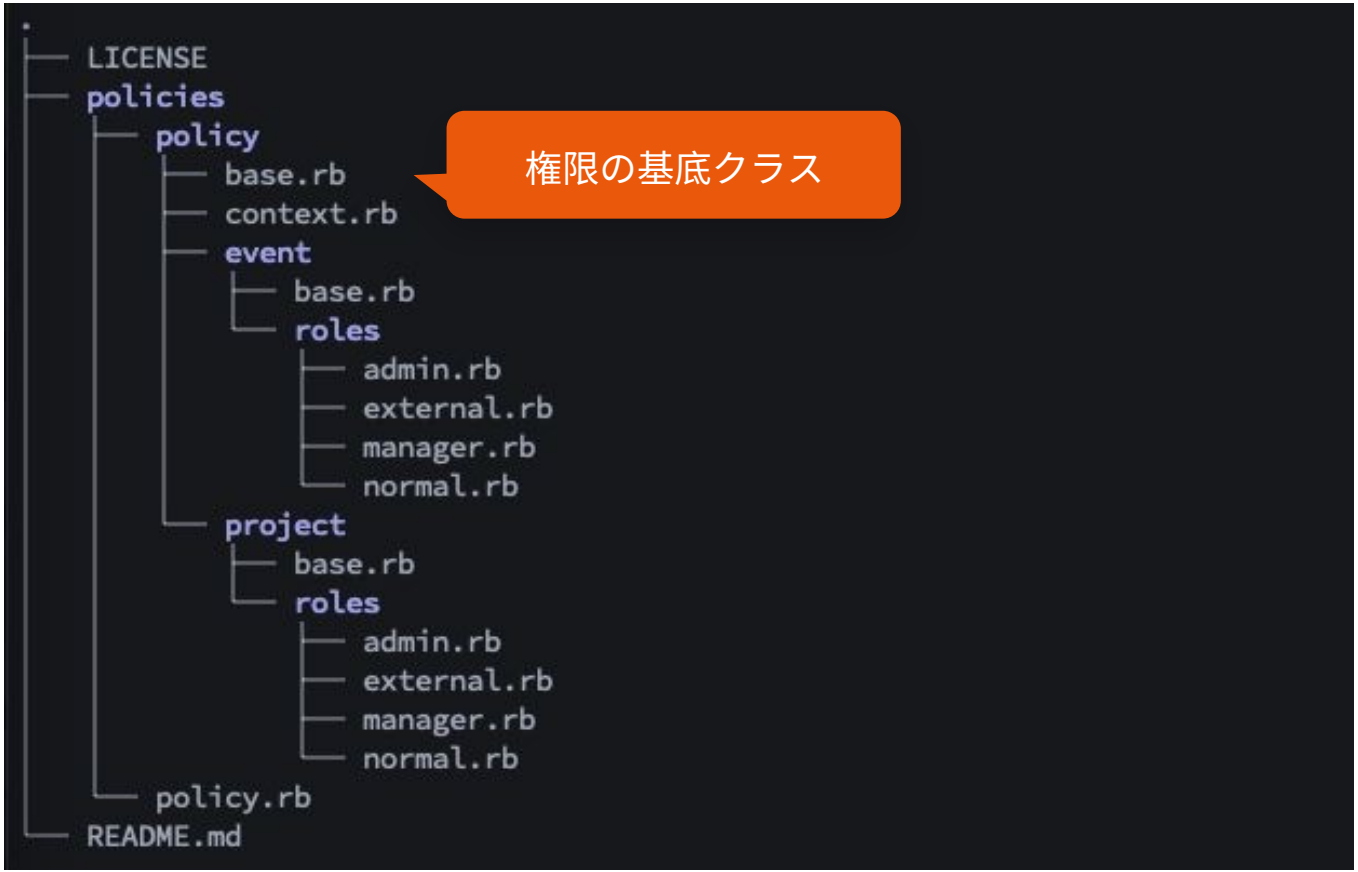
権限のクラスを
list モードで new

権限のクラスで定義されている
全ての public メソッドを実行

Policy::{ 対象 } で定義されている
対象名を全て取得

ここまでの整理

- 対象ごとにディレクトリを作成、役割ごとにファイルを作成
  - 役割と条件を分ける
- メタプログラミングを活用
  - 権限の変更が容易
- 対象の基底クラスで CRUD 以外の操作を追加
  - 対象ごとに異なる操作の拡張に対応
- 対象の基底クラスで条件を定義、条件を論理演算で使用
  - 判定は英語と論理演算がわかれば十分
  - 対象によって異なる条件の判定に対応


.
LICENSE
policies
policy
base.rb
context.rb
event
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
project
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
policy.rb
README.md
権限の基底クラス

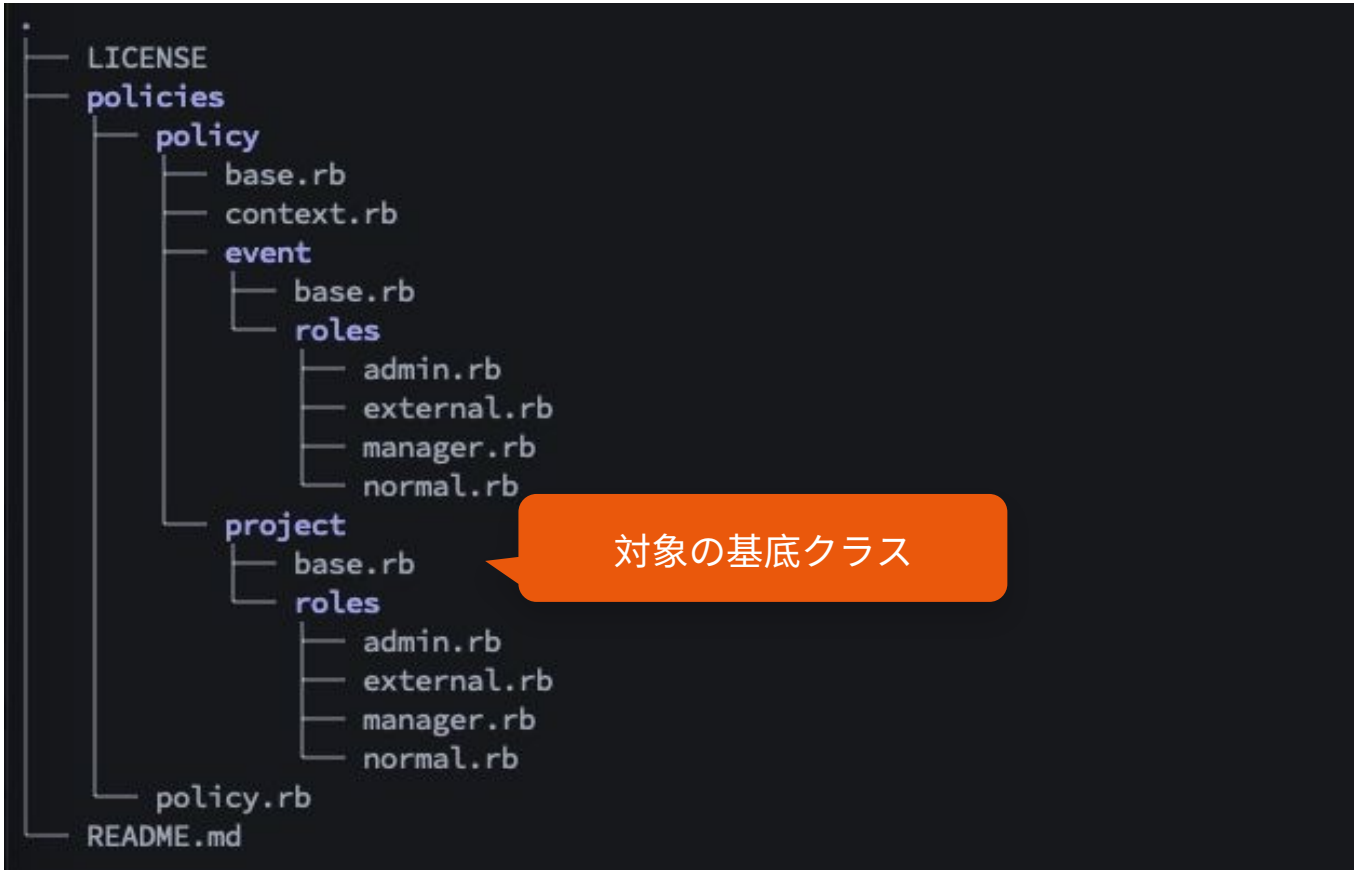
```ruby
module Policy
  class Base
    attr_reader :user, :record, :scope, :mode

    # NOTE: mode: :list, :record, :scope
    def initialize(user:, record: nil, scope: nil, mode: nil)
      @user = user
      @record = record
      @scope = scope
      @mode = mode
    end

    def read
      raise(NotImplementedError)
    end

    def create
      raise(NotImplementedError)
    end

    def update
      raise(NotImplementedError)
    end

    def delete
      raise(NotImplementedError)
    end
  end
end
```

```ruby
module Policy
  class Base
    attr_reader :user, :record, :scope, :mode

    # NOTE: mode: :list, :record, :scope
    def initialize(user:, record: nil, scope: nil, mode: nil)
      @user = user
      @record = record
      @scope = scope
      @mode = mode
    end

    def read
      raise(NotImplementedError)
    end

    def create
      raise(NotImplementedError)
    end

    def update
      raise(NotImplementedError)
    end

    def delete
      raise(NotImplementedError)
    end
  end
end
```

判定に必要な情報
判定のモード

```ruby
module Policy
  class Base
    attr_reader :user, :record, :scope, :mode

    # NOTE: mode: :list, :record, :scope
    def initialize(user:, record: nil, scope: nil, mode: nil)
      @user = user
      @record = record
      @scope = scope
      @mode = mode
    end

    def read
      raise(NotImplementedError)
    end

    def create
      raise(NotImplementedError)
    end

    def update
      raise(NotImplementedError)
    end

    def delete
      raise(NotImplementedError)
    end
  end
end
```

操作のデフォルトとして CRUD を定義


.
LICENSE
policies
policy
base.rb
context.rb
event
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
project
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
policy.rb
README.md
対象の基底クラス

```ruby
module Policy
  module Project
    class Base < Policy::Base
      def author?
        record.author == user
      end

      def assignee?
        record.assignees.exists?(id: user)
      end

      def author_scope
        scope.where(author: user)
      end

      def assignee_scope
        scope.left_joins(:assignees).where(assignees: { id: user }).distinct
      end

      def author_or_assignee_scope
        base_scope = scope.left_joins(:assignees)
        base_scope.where(author: user).or(base_scope.where(assignees: { id: user })).distinct
      end

      def view_settings_page
        raise(NotImplementedError)
      end
    end
  end
end
```

```ruby
module Policy
  module Project
    class Base < Policy::Base
      def author?
        record.author == user
      end

      def assignee?
        record.assignees.exists?(id: user)
      end

      def author_scope
        scope.where(author: user)
      end

      def assignee_scope
        scope.left_joins(:assignees).where(assignees: { id: user }).distinct
      end

      def author_or_assignee_scope
        base_scope = scope.left_joins(:assignees)
        base_scope.where(author: user).or(base_scope.where(assignees: { id: user })).distinct
      end

      def view_settings_page
        raise(NotImplementedError)
      end
    end
  end
end
```

record モード用の条件を定義

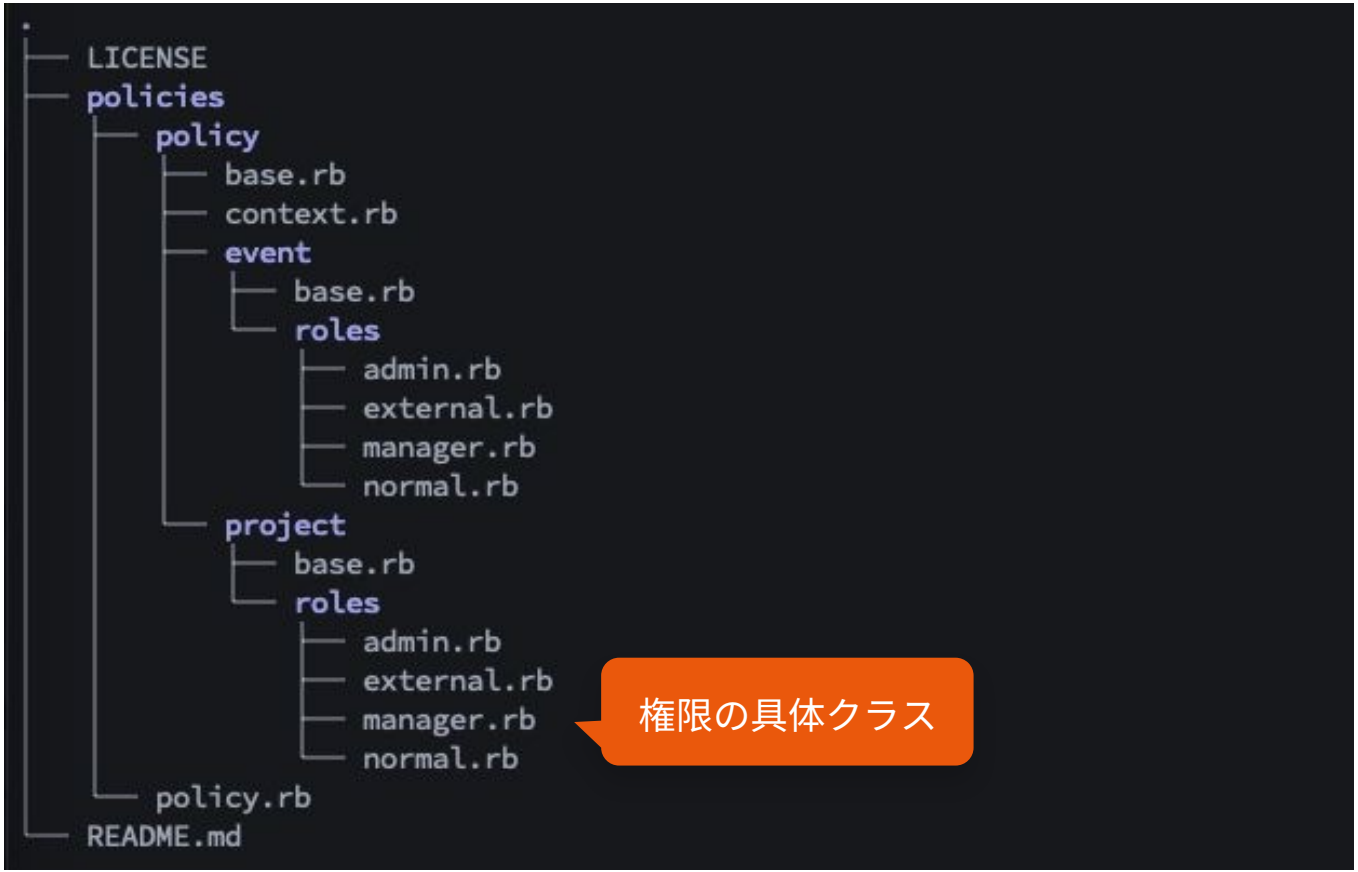
```ruby
module Policy
  module Project
    class Base < Policy::Base
      def author?
        record.author == user
      end

      def assignee?
        record.assignees.exists?(id: user)
      end

      def author_scope
        scope.where(author: user)
      end

      def assignee_scope
        scope.left_joins(:assignees).where(assignees: { id: user }).distinct
      end

      def author_or_assignee_scope
        base_scope = scope.left_joins(:assignees)
        base_scope.where(author: user).or(base_scope.where(assignees: { id: user })).distinct
      end

      def view_settings_page
        raise(NotImplementedError)
      end
    end
  end
end
```

scope モード用の条件を定義


module Policy
module Project
class Base < Policy::Base
def author?
record.author == user
end
def assignee?
record.assignees.exists?(id: user)
end
def author_scope
scope.where(author: user)
end
def assignee_scope
scope.left_joins(:assignees).where(assignees: { id: user }).distinct
end
def author_or_assignee_scope
base_scope = scope.left_joins(:assignees)
base_scope.where(author: user).or(base
end
def view_settings_page
raise(NotImplementedError)
end
end
end
end
CRUD 以外の操作の追加

ここまでの整理

- 対象ごとにディレクトリを作成、役割ごとにファイルを作成
  - 役割と条件を分ける
- 権限のクラスの特定でメタプログラミングを活用
  - 権限の変更が容易
- 対象の基底クラスで CRUD 以外の操作を追加
  - 対象ごとに異なる操作の拡張に対応
- 対象の基底クラスで条件を定義、条件を論理演算で使用
  - 判定は英語と論理演算がわかれば十分
  - 対象によって異なる条件の判定に対応


.
LICENSE
policies
policy
base.rb
context.rb
event
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
project
base.rb
roles
admin.rb
external.rb
manager.rb
normal.rb
policy.rb
README.md
権限の具体クラス

```
module Policy
  module Project
    module Roles
      class Manager < Base
        ...

        def update
          case mode
          when :list
            [:assignee, :author]
          when :record
            assignee? || author?
          when :scope
            author_or_assignee_scope
          end
        end

        def view_settings_page
          case mode
          when :list
            true
          when :record
            true
          when :scope
            scope
          end
        end
      end
    end
  end
end
```

```
3  module Policy
4    module Project
5      module Roles
6        class Manager < Base
7          ...
8
9          def update
10           case mode
11           when :list
12             [:assignee, :author]
13           when :record
14             assignee? || author?
15           when :scope
16             author_or_assignee_scope
17           end
18         end
19
20         def view_settings_page
21           case mode
22           when :list
23             true
24           when :record
25             true
26           when :scope
27             scope
28           end
29         end
30       end
31     end
32   end
33 end
```

対象の基底クラスで定義した条件を使用して判定を表現する

条件がない場合は Boolean を記述する

```
3  module Policy
4    module Project
5      module Roles
6        class Manager < Base
7          ...
8
9          def update
10           case mode
11           when :list
12             [:assignee, :author]
13           when :record
14             assignee? || author?
15           when :scope
16             author_or_assignee_scope
17           end
18         end
19
20         def view_settings_page
21           case mode
22           when :list
23             true
24           when :record
25             true
26           when :scope
27             scope
28           end
29         end
30       end
31     end
32   end
33 end
```

対象の基底クラスで定義した条件を使用して判定を表現する

条件がない場合は
scope か scope.none を記述する

```
3  module Policy
4    module Project
5      module Roles
6        class Manager < Base
7          ...
8
9          def update
10           case mode
11           when :list
12             [:assignee, :author]
13           when :record
14             assignee? || author?
15           when :scope
16             author_or_assignee_scope
17           end
18         end
19
20         def view_settings_page
21           case mode
22           when :list
23             true
24           when :record
25             true
26           when :scope
27             scope
28           end
29         end
30       end
31     end
32   end
33 end
```

具体的なレコードが必要な場合は
or 条件のキーを配列で返す

条件がない場合は
Boolean を記述する

```
3  module Policy
4    module Project
5      module Roles
6        class Manager < Base
7          ...
8
9          def update
10           case mode
11           when :list
12             [:assignee, :author]
13           when :record
14             assignee? || author?
15           when :scope
16             author_or_assignee_scope
17           end
18         end
19
20         def view_settings_page
21           case mode
22           when :list
23             true
24           when :record
25             true
26           when :scope
27             scope
28           end
29         end
30       end
31     end
32   end
33 end
```

追加した操作の条件も
同様に記述

ここまでの整理

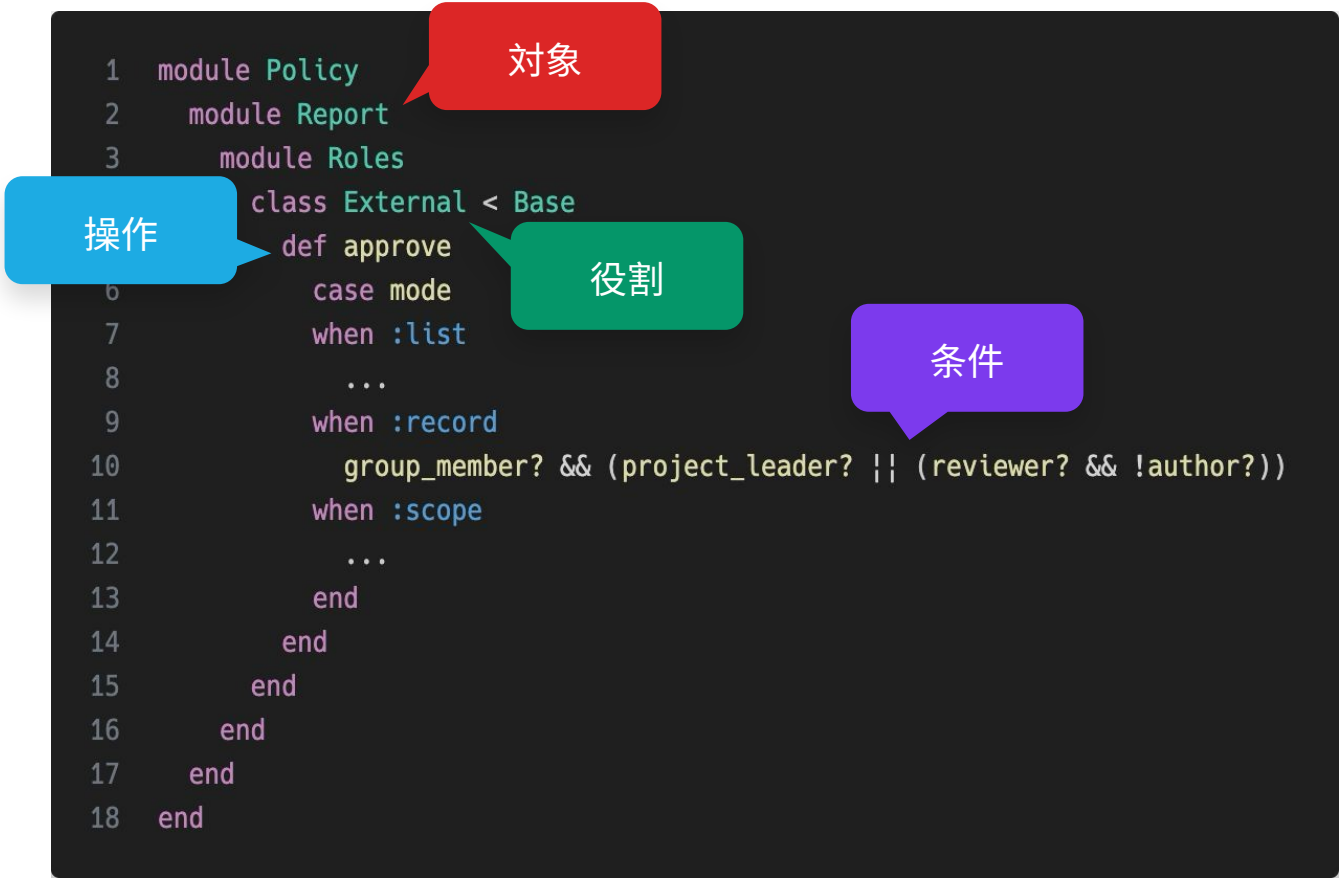
- 対象ごとにディレクトリを作成、役割ごとにファイルを作成
  - 役割と条件を分ける
- 権限のクラスの特定でメタプログラミングを活用
  - 権限の変更が容易
- 対象の基底クラスで CRUD 以外の操作を追加
  - 対象ごとに異なる操作の拡張に対応
- 対象の基底クラスで条件を定義、条件を論理演算で使用
  - 判定は英語と論理演算がわかれば十分
  - 対象によって異なる条件の判定に対応

ざっくり概要

1. 対象と役割から権限のクラスを特定
2. 判定モードの指定
3. 操作名の関数を実行
4. 条件の判定

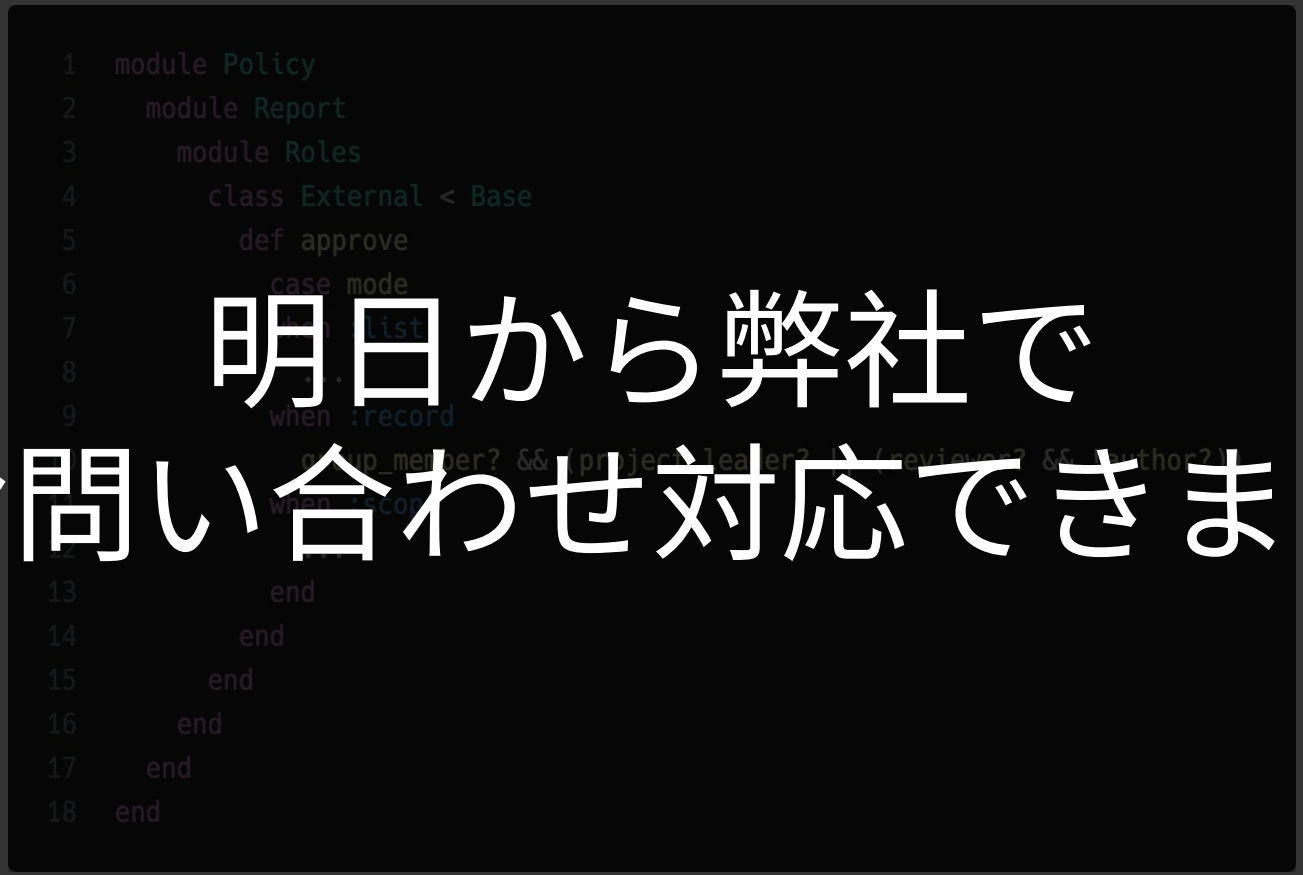
Q. 問題

```
module Policy
  module Report
    module Roles
      class External < Base
        def approve
          case mode
          when :list
            ...
          when :record
            group_member? && (project_leader? || (reviewer? && !author?))
          when :scope
            ...
          end
        end
      end
    end
  end
end
```


対象
操作
役割
条件
1 module Policy
2 module Report
3 module Roles
class External < Base
def approve
case mode
when :list
...
when :record
group_member? && (project_leader? || (reviewer? && !author?))
when :scope
...
end
end
end
end
end
end

```
module Policy
  module Report
    module Roles
      class External < Base
        def approve
          case mode
          when :list
            ...
          when :record
            group_member? && (project_leader? || (reviewer && !author?))
          when :scope
            ...
          end
        end
      end
    end
  end
end
```

なんとなくわかった人🙋



明日から弊社で
お問い合わせ対応できます

使用例

```ruby
class ProjectsController < ApplicationControll
  def index
    projects = Project.all

    @projects = Policy.authorize_scope(current_user, projects, :read)

    ...
  end

  def update
    project = Project.find(params[:id])

    @project = Policy.authorize(current_user, project, :update)

    ...
  end
end
```

read できる projects に絞り込む

```ruby
class ProjectsController < ApplicationControll
  def index
    projects = Project.all

    @projects = Policy.authorize_scope(current_user, projects, :read)

    ...
  end

  def update
    project = Project.find(params[:id])

    @project = Policy.authorize(current_user, project, :update)

    ...
  end
end
```

read できる projects に絞り込む

update できる project か判定

```ruby
class ProjectsController < ApplicationControll
  def index
    projects = Project.all

    @projects = Policy.authorize_scope(current_user, projects, :read)

    ...
  end

  def update
    project = Project.find(params[:id])

    @project = Policy.authorize(current_user, project, :update)

    ...
  end
end
```

read できる projects に絞り込む

update できる project か判定

権限に依存しているので明示的
役割の変化に影響されない

```ruby
module Policy
  module Project
    module Roles
      class Manager < Base
        def read
          case mode
          when :list
            true
          when :record
            true
          when :scope
            scope
          end
        end

        def update
          case mode
          when :list
            [:assignee, :author]
          when :record
            assignee? || author?
          when :scope
            author_or_assignee_scope
          end
        end
      end
    end
  end
end
```

パッとわかる

パッとわかる

```
3  module Policy
4    module Project
5      class Base < Policy::Base
6        def author?
7          record.author == user
8        end
9
10       def assignee?
11         record.assignees.exists?(id: user)
12       end
13
14       def author_scope
15         scope.where(author: user)
16       end
17
18       def assignee_scope
19         scope.left_joins(:assignees).where(assignees: { id: user }).distinct
20       end
21
22       def author_or_assignee_scope
23         base_scope = scope.left_joins(:assignees)
24         base_scope.where(author: user).or(base_scope.where(assignees: { id: user })).distinct
25       end
26
27       def view_settings_page
28         raise(NotImplementedError)
29       end
30     end
31   end
32 end
```

パッとわかる

間違えない

1. 実装で間違えない
    a. 追加、変更をするとき
2. 利用で間違えない
    a. 処理の中で判定をするとき
3. 理解で間違えない
    a. コードリーディングのとき
    b. お問い合わせの回答のとき

間違えない

1. 実装で間違えない
    a. 追加、変更をするとき
2. 利用で間違えない
    a. 処理の中で判断をするとき
3. 理解で間違えない
    a. コードリーディングのとき
    b. お問い合わせの回答のとき

間違えない👏

良い設計の影響

事業への影響

- サービスの信頼性が向上した
  - Module 導入から 1 年が経過したが、不具合の発生件数がゼロ
- 社内からのお問い合わせがゼロになり、開発生産性が向上した
  - CS や PdM が GitHub の 1 次情報を見て理解できるようになった
  - エンジニアは質問されたら、全部調べてしまい 10 分ほど使ってしまう

クライアント（ Next.js ）への影響

- Rails だけでなく Next.js の技術負債も防げている
  - 権限の一覧を渡すため、 Next.js も役割でなく権限に依存する実装に自然となった

```
<>
  <Permission policy="project" method="update">
    <Button>編集する</Button>
  </Permission>
  <Permission policy="project" method="delete">
    <Button>削除する</Button>
  </Permission>
</>
```

クライアント（ Next.js ）への影響

- Rails だけでなく Next.js の技術負債も防げている
  - 権限の一覧を渡すため、 Next.js も役割でなく権限

project の update の権限がない場合は
編集ボタンは表示されない

```
<>
  <Permission policy="project" method="update">
    <Button>編集する</Button>
  </Permission>
  <Permission policy="project" method="delete">
    <Button>削除する</Button>
  </Permission>
</>
```

クライアント（ Next.js ）への影響

- Rails だけでなく Next.js の技術負債も防げている
  - 権限の一覧を渡すため、 Next.js も役割でなく権限

```
<>
  <Permission policy="project" method="update">
    <Button>編集する</Button>
  </Permission>
  <Permission policy="project" method="delete">
    <Button>削除する</Button>
  </Permission>
</>
```

project の update の権限がない場合は
編集ボタンは表示されない

権限による UI の制御を
宣言的に記述できるようにしている

おまけ： Rails から外の世界へ

Next.js から Rails へリクエスト

Next.js から Rails へリクエスト

Module による判定
record と scope モード

NEXT.JS

Next.js から Rails へリクエスト

Next.js から Rails へリクエスト

NEXT.JS

判定済みのデータ

Web Dev Tool でも見れない

Next.js の権限管理

Next.js の権限管理

Next.js の権限管理

Module による判定
list モード

NEXT.JS

Next.js の権限管理

Module による判定
list モード

権限の一覧を渡す

Next.js の権限管理

Module による判定
list モード

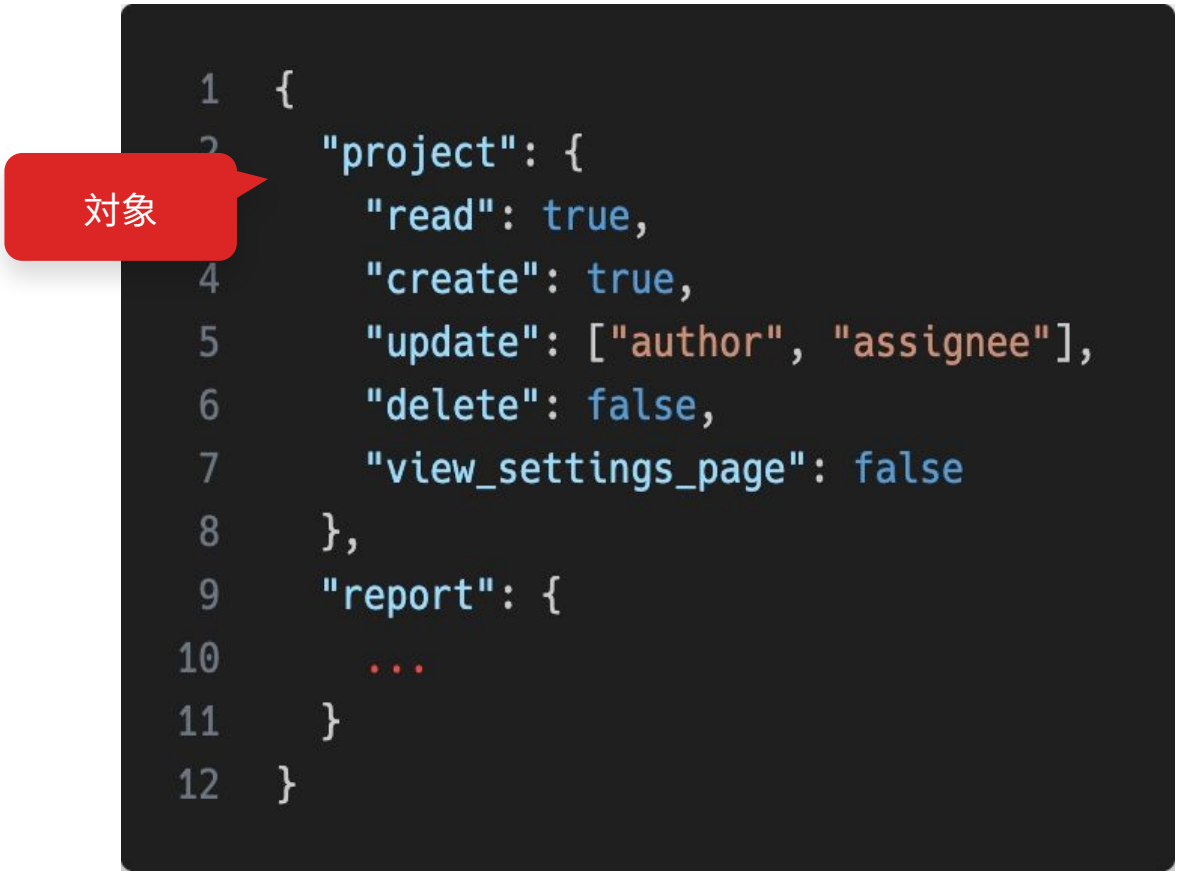
権限の一覧を渡す

権限による UI 制御

NEXT.JS

Next.js での権限管理

```
{
  "project": {
    "read": true,
    "create": true,
    "update": ["author", "assignee"],
    "delete": false,
    "view_settings_page": false
  },
  "report": {
    ...
  }
}
```


对象
{
"project": {
"read": true,
"create": true,
"update": ["author", "assignee"],
"delete": false,
"view_settings_page": false
},
"report": {
...
}
}


操作
{
"project": {
"read": true,
"create": true,
"update": ["author", "assignee"],
"delete": false,
"view_settings_page": false
},
"report": {
...
}
}


操作
具体的なレコードが必要な場合は
or 条件のキーの配列
{
"project": {
"read": true,
"create": true,
"update": ["author", "assignee"],
"delete": false,
"view_settings_page": false
},
"report": {
...
}
}

```
40  const newPolicyMap = Object.entries(permissions).reduce((map, [k]) => {
41    const key = k as keyof Permissions
42    const policy = PolicyFactory.create(key, permissions, { user })
43    // @ts-expect-error PolicyFactoryにないKeyがPermissionsにある場合、インスタンス化時点でTypeErrorとなるためここでは推論を無効化しておく
44    if (policy) map[key] = policy
45    return map
46  }, {} as PolicyMap)
```

```
40  const newPolicyMap = Object.entries(permissions).reduce((map, [k]) =>
41    const key = k as keyof Permissions
42    const policy = PolicyFactory.create(key, permissions, { user })
43    // @ts-expect-error PolicyFactoryにないKeyがPermissionsにある場合、インスタンス化時点でTypeErrorとなるためここでは推論を無効化しておく
44    if (policy) map[key] = policy
45    return map
46  }, {} as PolicyMap)
```

権限のクラスを特定

```
export type PolicyMap = {
  project: ProjectPolicy
  ...
}

export class PolicyFactory {
  static create<K extends keyof PolicyMap>(
    policyKey: K,
    permissions: Permissions,
    context: PolicyContext,
  ): PolicyMap[K] | undefined {
    switch (policyKey) {
      case 'project':
        return new ProjectPolicy(permissions['project'], context) as PolicyMap[K]
      ...
        default: {
        return undefined
      }
    }
  }
}
```

```
7  export type PolicyMap = {
8    project: ProjectPolicy
9    ...
10 }
11
12 export class PolicyFactory {
13   static create<K extends keyof PolicyMap>(
14     policyKey: K,
15     permissions: Permissions,
16     context: PolicyContext,
17   ): PolicyMap[K] | undefined {
18     switch (policyKey) {
19       case 'project':
20         return new ProjectPolicy(permissions['project'], context) as PolicyMap[K]
21       ...
22         default: {
23         return undefined
24       }
25     }
26   }
27 }
```

権限のクラスを特定

```ts
export type PolicyContext = {
  user: User
}

export abstract class BasePolicy<P extends Permissions[keyof Permissions]> {
  constructor(protected permissions: P, protected context: PolicyContext) {}

  abstract read(...model: unknown[]): boolean
  abstract create(...model: unknown[]): boolean
  abstract update(...model: unknown[]): boolean
  abstract delete(...model: unknown[]): boolean
}
```

```ts
export type PolicyContext = {
  user: User
}

export abstract class BasePolicy<P extends Permissions[keyof Permissions]> {
  constructor(protected permissions: P, protected context: PolicyContext) {}

  abstract read(...model: unknown[]): boolean
  abstract create(...model: unknown[]): boolean
  abstract update(...model: unknown[]): boolean
  abstract delete(...model: unknown[]): boolean
}
```

ユーザーの管理

```
4  export type PolicyContext = {
5    user: User
6  }
7
8  export abstract class BasePolicy<P extends Permissions[keyof Permissions]> {
9    constructor(protected permissions: P, protected context: PolicyContext) {}
10
11   abstract read(...model: unknown[]): boolean
12   abstract create(...model: unknown[]): boolean
13   abstract update(...model: unknown[]): boolean
14   abstract delete(...model: unknown[]): boolean
15 }
```

操作のデフォルトとして
CRUD を定義

```
export class ProjectPolicy extends BasePolicy<Permissions['project']> {
  private author = (project: Project): boolean => {
    return Boolean(project.author && project.author.id === this.context.user.id)
  }

  private assignee = (project: Project): boolean => {
    return extractNodes(project.assignees).some((v) => v.id === this.context.user.id)
  }

  read = (): boolean => {
    return this.permissions['read']
  }

  create = (): boolean => {
    return this.permissions['create']
  }

  update = (): boolean => {
    const permission = this.permissions['update']

    if (typeof permission === 'boolean') {
      return permission
    }

    if (project === undefined) {
      return false
    }

    return permission.some((key) => {
      switch (toCamelCase(key)) {
        case 'author':
          return this.author(project)
        case 'assignee':
          return this.assignee(project)
        default:
          return false
      }
    })
  }

  delete = (): boolean => {
    return this.permissions['delete']
  }

  viewSettingsPage = (): boolean => {
    return this.permissions['viewSettingsPage']
  }
}
```

```
7  export class ProjectPolicy extends BasePolicy<Permissions['project']> {
8    ...
9
10   read = (): boolean => {
11     return this.permissions['read']
12   }
13
14   create = (): boolean => {
15     return this.permissions['create']
16   }
17
18   update = (): boolean => {
19     ...
20   }
21
22   delete = (): boolean => {
23     return this.permissions['delete']
24   }
25
26   viewSettingsPage = (): boolean => {
27     return this.permissions['viewSettingsPage']
28   }
29 }
```

```ts
export class ProjectPolicy extends BasePolicy<Permissions['project']> {
  private author = (project: Project): boolean => {
    return Boolean(project.author && project.author.id === this.context.user.id)
  }

  private assignee = (project: Project): boolean => {
    return extractNodes(project.assignees).some((v) => v.id === this.context.user.id)
  }

  update = (): boolean => {
    const permission = this.permissions['update']

    if (typeof permission === 'boolean') {
      return permission
    }

    if (project === undefined) {
      return false
    }

    return permission.some((key) => {
      switch (toCamelCase(key)) {
        case 'author':
          return this.author(project)
        case 'assignee':
          return this.assignee(project)
        default:
          return false
      }
    })
  }
}
```

```ts
export class ProjectPolicy extends BasePolicy<Permissions['project']> {
  private author = (project: Project): boolean => {
    return Boolean(project.author && project.author.id === this.context.user.id)
  }

  private assignee = (project: Project): boolean => {
    return extractNodes(project.assignees).some((v) => v.id === this.context.user.id)
  }

  update = (): boolean => {
    const permission = this.permissions['update']

    if (typeof permission === 'boolean') {
      return permission
    }

    if (project === undefined) {
      return false
    }

    return permission.some((key) => {
      switch (toCamelCase(key)) {
        case 'author':
          return this.author(project)
        case 'assignee':
          return this.assignee(project)
        default:
          return false
      }
    })
  }
}
```

条件を定義

```
7  export class ProjectPolicy extends BasePolicy<Permissions['project']> {
8    private author = (project: Project): boolean => {
9      return Boolean(project.author && project.author.id === this.context.user.id)
10   }
11
12   private assignee = (project: Project): boolean => {
13     return extractNodes(project.assignees).some((v) => v.id === this.context.user.id)
14   }
15
16   update = (): boolean => {
17     const permission = this.permissions['update']
18
19     if (typeof permission === 'boolean') {
20       return permission
21     }
22
23     if (project === undefined) {
24       return false
25     }
26
27     return permission.some((key) => {
28       switch (toCamelCase(key)) {
29         case 'author':
30           return this.author(project)
31         case 'assignee':
32           return this.assignee(project)
33         default:
34           return false
35       }
36     })
37   }
38 }
```

判定が Boolean の場合

```ts
export class ProjectPolicy extends BasePolicy<Permissions['project']> {
  private author = (project: Project): boolean => {
    return Boolean(project.author && project.author.id === this.context.user.id)
  }

  private assignee = (project: Project): boolean => {
    return extractNodes(project.assignees).some((v) => v.id === this.context.user.id)
  }

  update = (): boolean => {
    const permission = this.permissions['update']

    if (typeof permission === 'boolean') {
      return permission
    }

    if (project === undefined) {
      return false
    }

    return permission.some((key) => {
      switch (toCamelCase(key)) {
        case 'author':
          return this.author(project)
        case 'assignee':
          return this.assignee(project)
        default:
          return false
      }
    })
  }
}
```

具体的なレコードがない場合

```
7  export class ProjectPolicy extends BasePolicy<Permissions['project']> {
8    private author = (project: Project): boolean => {
9      return Boolean(project.author && project.author.id === this.context.user.id)
10   }
11
12   private assignee = (project: Project): boolean => {
13     return extractNodes(project.assignees).some((v) => v.id === this.context.user.id)
14   }
15
16   update = (): boolean => {
17     const permission = this.permissions['update']
18
19     if (typeof permission === 'boolean') {
20       return permission
21     }
22
23     if (project === undefined) {
24       return false
25     }
26
27     return permission.some((key) => {
28       switch (toCamelCase(key)) {
29         case 'author':
30           return this.author(project)
31         case 'assignee':
32           return this.assignee(project)
33         default:
34           return false
35       }
36     })
37   }
38 }
```

or 条件の判定を行う

ここまでの整理

- Rails から渡ってきた権限の一覧の JSON を使って Map をつくっている
- その Map を Next.js で管理することで Next.js の権限管理を実現している
- Next.js でも権限のクラスの定義と特定をしている
- 具体的なレコードが必要な判定は Next.js で行っている

```
91  const checkPermission = useCallback(
92    <P extends PolicyKeys, M extends PolicyMethodKey<P>>(
93      policyKey: P,
94      methodKey: M,
95      model?: ModelType<P, M>,
96    ): boolean => {
97      const validMethod = (policy: PolicyMap[P], method: M) => {
98        return Object.prototype.hasOwnProperty.call(policy, method as string)
99      }
100
101     const policy = policyMap[policyKey]
102     if (!policy) return false
103
104     const method = validMethod(policy, methodKey) ? policy[methodKey] : undefined
105
106     if (typeof method !== 'function') {
107       throw new Error(`Permission method(${methodKey}) is not function in policy(${policyKey})`)
108     }
109
110     if (model) {
111       return method(...[model])
112     }
113
114     return method()
115   },
116   [policyMap],
117 )
```

```
91  const checkPermission = useCallback(
92    <P extends PolicyKeys, M extends PolicyMethodKey<P>>(
93      policyKey: P,
94      methodKey: M,
95      model?: ModelType<P, M>,
96    ): boolean => {
97      const validMethod = (policy: PolicyMap[P], method: M) => {
98        return Object.prototype.hasOwnProperty.call(policy, method as string)
99      }
100
101     const policy = policyMap[policyKey]
102     if (!policy) return false
103
104     const method = validMethod(policy, methodKey) ? policy[methodKey] : undefined
105
106     if (typeof method !== 'function') {
107       throw new Error(`Permission method(${methodKey}) is not function in policy(${policyKey})`)
108     }
109
110     if (model) {
111       return method(...[model])
112     }
113
114     return method()
115   },
116   [policyMap],
117 )
```

対象がない場合

```
91  const checkPermission = useCallback(
92    <P extends PolicyKeys, M extends PolicyMethodKey<P>>(
93      policyKey: P,
94      methodKey: M,
95      model?: ModelType<P, M>,
96    ): boolean => {
97      const validMethod = (policy: PolicyMap[P], method: M) => {
98        return Object.prototype.hasOwnProperty.call(policy, method as string)
99      }
100
101     const policy = policyMap[policyKey]
102     if (!policy) return false
103
104     const method = validMethod(policy, methodKey) ? policy[methodKey] : undefined
105
106     if (typeof method !== 'function') {
107       throw new Error(`Permission method(${methodKey}) is not function in policy(${policyKey})`)
108     }
109
110     if (model) {
111       return method(...[model])
112     }
113
114     return method()
115   },
116   [policyMap],
117 )
```

操作がない場合

```ts
const checkPermission = useCallback(
  <P extends PolicyKeys, M extends PolicyMethodKey<P>>(
    policyKey: P,
    methodKey: M,
    model?: ModelType<P, M>,
  ): boolean => {
    const validMethod = (policy: PolicyMap[P], method: M) => {
      return Object.prototype.hasOwnProperty.call(policy, method as string)
    }

    const policy = policyMap[policyKey]
    if (!policy) return false

    const method = validMethod(policy, methodKey) ? policy[methodKey] : undefined

    if (typeof method !== 'function') {
      throw new Error(`Permission method(${methodKey}) is not function in policy(${policyKey})`)
    }

    if (model) {
      return method(...[model])
    }

    return method()
  },
  [policyMap],
)
```

具体的なレコードが指定されている場合

```
91  const checkPermission = useCallback(
92    <P extends PolicyKeys, M extends PolicyMethodKey<P>>(
93      policyKey: P,
94      methodKey: M,
95      model?: ModelType<P, M>,
96    ): boolean => {
97      const validMethod = (policy: PolicyMap[P], method: M) => {
98        return Object.prototype.hasOwnProperty.call(policy, method as string)
99      }
100
101     const policy = policyMap[policyKey]
102     if (!policy) return false
103
104     const method = validMethod(policy, methodKey) ? policy[methodKey] : undefined
105
106     if (typeof method !== 'function') {
107       throw new Error(`Permission method(${methodKey}) is not function in policy(${policyKey})`)
108     }
109
110     if (model) {
111       return method(...[model])
112     }
113
114     return method()
115   },
116   [policyMap],
117 )
```

具体的なレコードが指定されていない場合

```
1  checkPermission('project', 'update', { model: project})
```

```
1   checkPermission('project', 'update', { model: project})
```

対象と操作が明示されている👍

```tsx
export const Permission = <
  P extends PolicyKeys,
  M extends PolicyMethodKey<P>,
>({
  policy,
  method,
  model,
  fallback,
  children,
}: Props<P, M, Strict>): ReactNode => {
  const result = checkPermission(policy, method, { model: })

  return result ? <>{children}</> : fallback || null
}
```

```tsx
<>
  <Permission policy="project" method="update" model={project} fallback={<PermissionDenied />}>
    <Button>編集する</Button>
  </Permission>
</>
```

権限による UI の制御を
宣言的に記述できるようにしている

project の update の権限がない場合は
編集ボタンは表示されない

```tsx
<>
  <Permission policy="project" method="update" model={project} fallback={<PermissionDenied />}>
    <Button>編集する</Button>
  </Permission>
</>
```

権限による UI の制御を
宣言的に記述できるようにしている

対象と操作が明示されている👍

```tsx
<>
  <Permission policy="project" method="update" model={project} fallback={<PermissionDenied />}>
    <Button>編集する</Button>
  </Permission>
</>
```

権限による UI の制御を
宣言的に記述できるようにしている

具体的なレコードが必要な判定に対応

```tsx
<>
  <Permission policy="project" method="update" model={project} fallback={<PermissionDenied />}>
    <Button>編集する</Button>
  </Permission>
</>
```

権限による UI の制御を
宣言的に記述できるようにしている

権限がない場合の表示

```tsx
<>
  <Permission policy="project" method="update" model={project} fallback={<PermissionDenied />}>
    <Button>編集する</Button>
  </Permission>
</>
```

権限による UI の制御を
宣言的に記述できるようにしている

ここまでの整理

- Rails から渡ってきた権限の一覧の JSON を使って Map をつくっている
- その Map を Next.js で管理することで Next.js の権限管理を実現している
- Next.js でも権限のクラスの定義と特定をしている
- 具体的なレコードが必要な判定は Next.js で行っている
- 権限の Map を活用することで Next.js でも対象と操作が明示された間違えないを実現
- 権限による UI の制御を宣言的に記述できるようにしている

まとめ

- 権限管理を間違えると、事業とサービスへの損失が大きい
- 権限管理が技術負債になるのは、役割が事業とサービスの成長と共に変化することと役割に依存した実装が原因
- 権限の実装は、役割（ admin? ）でなく権限に依存しよう
- 権限管理は、対象と操作と役割と条件の 4 つに分けられる
- 4 つを分割をした Module の設計と実装の解説
- 良い設計の事業とクライアント（ Next.js ）への影響

Module のサンプルは GitHub で公開しています

「技術負債にならない・間違えない」権限管理の設計と実装

Kaigi on Rails 2025

@naro143 （ Yusuke Ishimi ）