Rails のテストの仕組み

y-yagi　著

2019-04-01 版　　発行

はじめに

本書の内容

本書は、Ruby on Rails(以降 Rails) が提供しているテストの仕組みについて説明した本です。

Rails は Web アプリケーションを作る為のフレームワークですが、Rails によって作られたアプリケーションをテストする為の仕組みも合わせて提供しています。本書では、Rails がどのような仕組みを提供しているかについて説明します。

本書は、Rails が提供しているテストの仕組みについて記載しており、その仕組みを使用してテストを書く方法についての詳細については記載していません。予めご了承ください。

なお、特に注記が無い場合は、Rails は執筆時点で最新のバージョン (6.0.0) を対象にしています。

対象読者

本書は、既に Ruby、Rails を使った事がある人を対象としています。Ruby や Rails 自体の説明や、Ruby に関連するツールについての説明は端折っています。予めご了承ください。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責

任も負いません。また、本書に記載されいてる情報は、執筆時点でのものです。時間経過によって情報が古くなっている可能性がありますのであらかじめご了承ください。

第 1 章

テストの概要

まずはじめに、Rails が提供しているテストの為の仕組みの概要について説明します。

1.1 テストフレームワーク

Ruby でテストフレームワークといえば、test-unit、minitest、RSpec の名前が挙がることが多いと思います。Rails は、かつて test-unit を使っていて、現在は minitest を使用しています。Rails 自体のテストも、Rails を使用するユーザに向けたテストの仕組みも、どちらも minitest を使用しています。

日本で Rails アプリケーションの開発をされている場合、RSpec をお使いのケースが多いのではないかと思います。しかし RSpec は Rails がデフォルトでサポートしているテストフレームワークではなく、Rails 自体に RSpec の為の機能は全くありません。

1.2 minitest と Rails

先に述べた通り、Rails では minitest を使用しています。が、minitest をそのまま使用している訳ではありません。色々と機能拡張を行っています。

例えば、minitest では一致しない事をテストしたい場合、`refute` というメソッド、及び、`refute` で始まるメソッドを使用します。例えば値が一致しない事を確認したい場合、`refute_equal` というメソッドを使用します。

リスト 1.2: refute_equal

```
refute_equal 5, User.count
```

しかし Rails では、refute を使うことを推奨していません。代わりに、assert_not というメソッド、及び、assert_not で始まるメソッドが提供されており、そちらを使用することを推奨しています。先の例だと、assert_not_equal を使用する必要があります。

リスト 1.2: assert_not_equal

```
assert_not_equal 5, User.count
```

元々 test-unit には assert_not があり、test-unit から minitest に移行する際に互換性の為にこれらのメソッドが追加されました。

その後、refute を使用するようにする？ という提案もあったようなのですが、それは進まず、assert_not を使う形のままで落ち着いています[*1]。

なお、refute ではなく assert_not を使用する事をチェックする為の RuboCop の cop[*2]があります。Rails のリポジトリではこの cop が有効化されており、refute は一切使われないようになっています。

他にも、Rails ではテストを書く際、test というメソッドを使用してテストを定義します。

リスト 1.3: test メソッド

```
test "should get index" do
  get users_url
  assert_response :success
end
```

これも Rails が提供しているメソッドです。minitest を使用している場合この

[*1] 当時の議論が全ては見つからなかったので推測混じりなのですが、refute という名前をあまり好ましく思わない人がいたため、移行は行われなかったようです。

[*2] https://www.rubydoc.info/gems/rubocop/RuboCop/Cop/Rails/RefuteMethods

メソッドは使用する事は出来ません。

そのため、「Rails は mintiest を使用している」というより、「Rails は minitest を拡張した独自の仕組みを使用している」というのが正確な状態になっています。

1.3 テスト用のクラス

Rails は機能ごとにライブラリがわかれています。例えば O/R マッパーの Active Record、メール送信の Action Mailer、バックグランドジョブの Active Job、という具合です。

Rails では、各ライブラリ毎に、そのライブラリの機能を提供する為のクラスを提供しています。クラス図は次のようになっています。

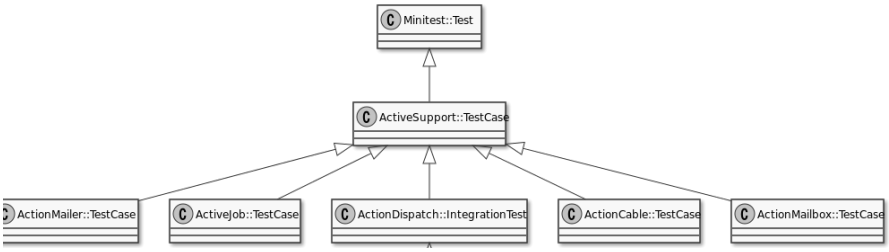

図 1.1: クラス図

紙面の都合上大分クラスを割愛しています。minitest のテスト用のクラスである Minitest::Test を継承した ActiveSupport::TestCase というクラスがあり、各ライブラリのテストクラスはその ActiveSupport::TestCase を継承している、という点だけ覚えておいて下さい。各クラスの詳細については次章で説明します。

1.4 その他テストの為の機能たち

テスト用のクラス以外に、テストデータを作成する為の Fixtures や、テストを実行する為の Test Runner、並列にテストを実行する為の Parallel Testing という機能があります。これらについては 3 章で説明します。

1.5 まとめ

本章では、Rails が提供しているテストの概要について説明しました。Rails は minitest を使用している、しかしそのまま使用しているのではなく、minitest を拡張して使用している、という点だけご理解頂けると幸いです。

第 2 章

テストの為のクラスたち

本章では、Rails が提供しているテスト用のクラスについて説明します。

2.1 クラスの一覧

Rails では、テスト用のクラスとして次のクラスを提供しています。

表 2.1: Rails が提供しているテスト用のクラス

クラス名
ActiveSupport::TestCase
ActionMailer::TestCase
ActionView::TestCase
ActionController::TestCase
ActionDispatch::IntegrationTest
ActionDispatch::SystemTestCase
ActiveJob::TestCase
ActionCable::TestCase
ActionCable::Connection::TestCase
ActionCable::Channel::TestCase
ActionMailbox::TestCase
Rails::Generators::TestCase

基本的には各ライブラリ毎にクラスが提供されています。そのため、ライブラリを使用した機能のテストをしたい場合、それぞれに提供されているテストを使

えば良いようになっています。

なお、Active Record や Active Model などの一部ライブラリでは専用のクラスは提供されていません。これは ActiveSupport::TestCase を使用すれば十分で、特別な機能を提供する必要が無い、と判断されている為です。専用のクラスが無い場合は、ActiveSupport::TestCase を使用するようにして下さい。

2.2 ActiveSupport::TestCase

前章でも述べた通り、Rails が提供しているテストクラスのベースになっているクラスです。各テストクラス共通で使用したい機能がある場合、このクラスに追加するようになっています。

例えば、`assert_not` 等の `assert_not_x` で始まるメソッドは ActiveSupport::TestCase に定義されており、全てのテスト用のクラスで使用出来るようになっています。

他にも、`travel`、`travel_to` という時間に関する処理を行う為のヘルパーメソッドが実装されている ActiveSupport::Testing::TimeHelpers というモジュールがあります。このモジュールは ActiveSupport::TestCase で include されており、全てのテストクラスで使用出来るようになっています。

リスト 2.1: travel_to

```
Time.current # => Wed, 27 Mar 2019 11:56:31 JST +09:00

# 1 日前に移動
travel 1.day do
  Time.current # => Tue, 26 Mar 2019 11:56:35 JST +09:00
end

Time.current # => Wed, 27 Mar 2019 11:56:38 JST +09:00
```

ActiveSupport::TestCase は Rails 自体のテストでも使用されています。そのため、Rails 自体のテストをする為に必要なメソッドもこのクラスで使用出来るよう対応されています。

例えば、何か既存の機能を deprecate にした場合に、その機能が deprecate になっている事を確認する為の `assert_deprecated` というアサーションがあり

ます。

リスト 2.2: assert_deprecated

```
# update_attributes メソッドを使用すると deprecate メッセージが出る
topic = Topic.find(1)
assert_deprecated do
  topic.update_attributes("title" => "The First Topic Updated")
end
```

他にも、標準入出力を抑止したり、モック処理用のメソッドが定義されています。

クラスについてのより詳細は、https://edgeapi.rubyonrails.org/classes/ActiveSupport/TestCase.html をご参照下さい。

2.3 ActionMailer::TestCase

メール送信機能のテスト用クラスです。

当然のことですが、テストで実際にメールを送信するわけにはいきません。ActionMailer::TestCase では、メール送信処理が実行されても実際のメールの送信は行わず、代わりに送信処理が実行されたメールを内部で保持し参照出来るようにしています。合わせて、送信処理が呼ばれた (または呼ばれていない) 事を確認する為のアサーション等を提供しています。

リスト 2.3: ActionMailer::TestCase

```
class UserMailerTest < ActionMailer::TestCase
  test "invite friend" do
    # invite_friend を実行したら招待用のメールが送信される
    assert_emails 1 do
      User.invite(email: 'friend@example.com')
    end
  end
end
```

assert_emails は同期処理 (deliver_now)、非同期処理 (deliver_later) どちらで送信されたメールもチェックの対象になります。非同期で送信された

メールだけチェックしたい場合は、assert_enqueued_emails を使用する必要があります。

クラスについてのより詳細は、https://edgeapi.rubyonrails.org/classes/ActionMailer/TestCase.html をご参照下さい。

2.4 ActiveJob::TestCase

バックグランドジョブ機能のテスト用クラスです。

メールと異なり、バックグランドジョブはテストで実際に実行しても問題無い事が多いでしょう。しかし例えば、「1 時間後に実行されるジョブ」があった場合、テストで実際に 1 時間待つ訳にはいきません。ActiveJob::TestCase では、ジョブの登録処理が行われたらそのジョブを内部で保持し、どのようなジョブが登録されたかを確認出来るようにしています。その登録された内容を確認する為のアサーションも提供されています。

リスト 2.4: ActiveJob::TestCase

```
class LoggingJobTest < ActiveJob::TestCase
  test "withdrawal" do
    user = User.last
    # ユーザが退会したら LoggingJob が登録される
    assert_enqueued_with(job: LoggingJob) do
      user.withdrawal
    end
  end
end
```

アサーションではジョブに指定された引数をチェックしたり、登録されたジョブを実行したりする事も出来るようになっています

クラスについてのより詳細は、https://edgeapi.rubyonrails.org/classes/ActiveJob/TestCase.html をご参照下さい。

2.5 ActionView::TestCase

名前から Action View のテンプレートに関するテスト用のクラスかと推測されるかと思うのですが、実際はちょっと異なり view helper のテスト用クラス

です。

ActionView::TestCase では view helper を使用する為に必要な controller や view の生成処理を行い、テスト単体で view helper メソッドが使用出来るようにしています。

リスト 2.5: ActionView::TestCase

```
module UsersHelper
  def link_to_user(user)
    link_to "#{user.first_name} #{user.last_name}", user
  end
end

class UsersHelperTest < ActionView::TestCase
  test "link_to_user returns link with user name" do
    user = User.find_by(first_name: "Yamada", last_name: "Taro")
    assert_dom_equal %{<a href="/user/#{user.id}">Yamada Taro</a>},
      link_to_user(user)
  end
end
```

しかし、Rails 4.2 から helper のテストはそもそも生成されなくなり、このクラスが使用される事は無くなりました。これは view helper 単体でのテストはあまり意味が無いのでは、という意見によるものです[*1]。そのため、クラス自体は残ったはいますが、目にする事はあまりないかと思います。

2.6 ActionController::TestCase

コントローラのテスト用クラスです。特定のコントローラのメソッドに対して、HTTP リクエストの送信及びレスポンスの確認ができるようになっています。

リスト 2.6: ActionController::TestCase

```
class UsersControllerTest < ActionController::TestCase
```

[*1] インテグレーションテストやシステムテスト等でビューのテストとしてやるのが良いのでは、という意見が多いです。

```
  test "should get index" do
    get :index
    assert_response :success
  end

  test "should create user" do
    assert_difference('User.count') do
      post(:create, params:
        { user: { email: @user.email, name: @user.name } })
    end

    assert_redirected_to user_url(User.last)
  end
end
```

同様にコントローラのテストを行う為のクラスとして ActionDispatch::IntegrationTest があります。

ActionDispatch::IntegrationTest だとルーティングもセットでテストが出来る[*2]、HTTP リクエストがより実際のリクエストに近い形で送信される等のメリットがあるのですが、実行は ActionController::TestCase の方が高速だった為、コントローラのテストには長らく ActionController::TestCase が使われるようになっていました。

しかし、Rails 5.0 で ActionDispatch::IntegrationTest のパフォーマンスが改善され、実行速度の差は大分縮まりました。それにより、コントローラのテストでも ActionDispatch::IntegrationTest が使用される事が推奨されるようになり、scaffold で生成するコントローラのテストでも ActionDispatch::IntegrationTest が使用されるようになりました。

因みに、その対応の際に ActionController::TestCase は gem に切り出して Rails 本体から削除するという話があったのですが、何だかんだまだコード Rails 本体に残ったままになっています。とはいえ、機能追加等が行われる事は (恐らく) 無いので、新規に追加するテストについては ActionDispatch::IntegrationTest を使用する事をおすすめします。

[*2] ActionController::TestCase は送信先にコントローラのアクション名を指定する為、ルーティングのテストは出来なかったのでした。

2.7 ActionDispatch::IntegrationTest

コントローラとルーティングのテスト用クラスです。ActionController::TestCase と異なり、HTTP リクエスト先に任意のパスを指定出来し、ルーティングについても確認出来るようになっています。

リスト 2.7: ActionDispatch::IntegrationTest

```
class UsersControllerTest < ActionDispatch::IntegrationTest
  test "should get index" do
    get users_url
    assert_response :success
  end

  test "should create user" do
    assert_difference('User.count') do
      post users_url, params:
        { user: { email: @user.email, name: @user.name } }
    end

    assert_redirected_to user_url(User.last)
  end
end
```

ActionDispatch::IntegrationTest ではルーティングを確認する為のアサーションが提供されています。

リスト 2.8: route assertions

```
assert_routing '/home', controller: 'home', action: 'index'
assert_routing 'controller/action/9', {id: "9", item: "square"},
  { controller: "controller", action: "action"}, {}, {item: "square" }

assert_recognizes({controller: 'items', action: 'list'}, 'items/list')
```

また、ActionDispatch::IntegrationTest は Rails 6.0 から自動的に ActionMailer::TestCase、ActiveJob::TestCase で使用されているヘルパーモジュール (ActionMailer::TestHelper、ActiveJob::TestHelper を自動的にインクルードす

るようになりました。これにより、ActionDispatch::IntegrationTest 内で行ったメールの送信、バックグランドジョブの登録についてもテスト出来るようになっています。

クラスについてのより詳細は、https://edgeapi.rubyonrails.org/classes/ActionDispatch/IntegrationTest.html をご参照下さい。

2.8 ActionDispatch::SystemTestCase

ActionDispatch::SystemTestCase はシステムテストの為のクラスです。Capybara[*3]を使用して簡単にブラウザを使用したテストを書けるようになっています。

リスト 2.9: system test

```
class UsersTest < ApplicationSystemTestCase
  test "creating a User" do
    visit users_url
    click_on "New User"

    fill_in "Email", with: @user.email
    fill_in "Name", with: @user.name
    click_on "Create User"

    assert_text "User was successfully created"
    click_on "Back"
  end
end
```

Rails 5.2 までは ActionDispatch::IntegrationTest を継承していましたが、6.0 からは他のクラス同様 ActiveSupport::TestCase を直接継承するようになっています。

Capybara で使用する為のドライバーは ActionDispatch::SystemTestCase で実装されており、ユーザはそのドライバーを指定する為のメソッドを使用すれば、Capybara の設定を意識する事なくブラウザを指定する事ができるようになっています。

リスト 2.10: system test でブラウザを指定

[*3] https://github.com/teamcapybara/capybara

```
class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
  # テストでヘッドレス Chrome を使用する
  driven_by :selenium, using: :headless_chrome, screen_size: [1400, 1400]
end
```

Capybara のラッパー以外の機能としては、スクリーンショットの取得機能があります。任意のタイミングでの取得は勿論、テスト失敗時に自動でスクリーンショットの取得を行ってくれるようになっています。なお、テスト失敗時のスクリーンショットの表示は、Rails 6.0 だとスクリーンショットのファイル名のみです。ターミナル上でスクリーンショットを直接表示したい場合、'RAILS_SYSTEM_TESTING_SCREENSHOT'に適切な値を指定する必要があります。

クラスについてのより詳細は、https://edgeapi.rubyonrails.org/classes/ActionDispatch/SystemTestCase.html をご参照下さい。

2.9 ActionCable::TestCase

Rails 6.0 から追加された Action Cable のテスト用クラスです。Action Cable が追加された当初、Action Cable のテスト用の機能は Rails 本体にありませんでした。これは、Action Cable に関するテストはブラウザを使用して行うテスト (現在のシステムテスト) で確認した方が正確で、単体でテストは行う必要は無いのでは、という意見があった為です。

しかし、API-only アプリケーションでも Action Cable を使う、というケースが出てきました。API-only アプリケーションだとシステムテストは使用出来ない為、専用のテストの仕組みがあった方が良いよね、という声が強まり、晴れて Rails 本体に機能が追加されました。

なお、元々 action-cable-testing[*4]という gem があり、その機能を Rails 本体にインポート[*5]した形になります。その為、Rails 6.0 より前で Action Cable のテストを行いたい場合、action-cable-testing を使用すれば、同等のテストが出

[*4] https://github.com/palkan/action-cable-testing

[*5] Rails 本体へのインポート処理を行ったのは action-cable-testing の作者本人です。元々 Rails 本体に PR を出していたのですが、中々マージされなかった為 gem にされたようです。

来るようになっています。

ActionCable::TestCase ではテスト用の adapter を使用しブロードキャストの管理を行うようになっていて、メッセージが送信された/されてない等をテスト出来るようになっています。

リスト 2.11: ActionCable::TestCase

```
class ChartRoomTest < ActionCable::TestCase
  test "broadcast message" do
    assert_broadcasts('messages', 1) do
      ActionCable.server.broadcast 'messages', { text: 'hello' }
    end
  end
end
```

ActionCable::TestCase はブロードキャストに関する処理のみ提供しており、コネクション、チャンネルに関するテストは、後述する ActionCable::Connection::TestCase、ActionCable::Channel::TestCase をそれぞれ使用する必要があります。

クラスについてのより詳細は、https://edgeapi.rubyonrails.org/classes/ActionCable/TestHelper.html をご参照下さい。

2.10 ActionCable::Connection::TestCase

Action Cable のコネクションに関するテストの為のクラスです。接続処理の為のヘルパーメソッド (connect) や、接続に失敗した事を確認する為のアサーション (assert_reject_connection) が提供されています。

リスト 2.12: ActionCable::Connection::TestCase

```
class ApplicationCable::ConnectionTest < ActionCable::Connection::TestCase
  # 適切な cookie が設定されていれば接続出来る
  test "connects with proper cookie" do
    cookies["user_id"] = users(:john).id

    connect

    assert_equal users(:john).id, connection.user.id
```

```
  end

  # 適切な cookie が設定されていない場合接続エラーになる
  test "rejects connection without proper cookie" do
    assert_reject_connection { connect }
  end
end
```

クラスについてのより詳細は、https://edgeapi.rubyonrails.org/classes/ActionCable/Connection/TestCase.html をご参照下さい。

2.11 ActionCable::Channel::TestCase

Action Cable のチャンネルのテストの為のクラスです。チャンネルに対しての subscription の作成処理や、stream が正しく開始されていることを確認する為のアサーションを提供しています。

リスト 2.13: ActionCable::Channel::TestCase

```
class ChatChannelTest < ActionCable::Channel::TestCase
  test "subscribes and stream for room" do
    # "room"に対する subscription を作成
    subscribe room: "15"

    # "room" 15 に対する stream が開始されていること
    assert subscription.confirmed?
    assert_has_stream "chat_15"
  end
end
```

クラスについてのより詳細は、https://edgeapi.rubyonrails.org/classes/ActionCable/Channel/TestCase.html をご参照下さい。

2.12 ActionMailbox::TestCase

Rails 6.0 で追加された Action Mailbox のテストの為のクラスです。Action Mailbox は、メール受信処理の為のライブラリです。メールの受信 -> メールの

内容に応じた各種処理の実施 -> 処理が終わったメールの削除等の機能を提供しています。

「メールの内容に応じた各種処理」は、Mailbox というクラスに定義します。

リスト 2.14: Mailbox

```
class InboxMailbox < ApplicationMailbox
  def process
    # メールの内容を DB に保存
    ReceiveMail.create!(
      from: mail.from.first, to: mail.to.first,
      subject: mail.subject, body: mail.body)

    user = User.find_by(email: mail.to.first)
    # ユーザにメールを通知
    user.notify_email(mail) if user
  end
end
```

この Mailbox クラスに対するテストを行う為のクラスが ActionMailbox::TestCase です。ActionMailbox::TestCase では、受信メールを作成する為のヘルパーメソッドを提供しています。

リスト 2.15: ActionMailbox::TestCase

```
class InboxMailboxTest < ActionMailbox::TestCase
  test "receive mail" do
    receive_inbound_email_from_mail \
      to: '"someone" <someone@example.com>',
      from: '"else" <else@example.com>',
      subject: "Hello world!",
      body: "Hello?"

    mail = ReceiveMail.last
    assert_equal "else@example.com", mail.from
    assert_equal "someone@example.com", mail.to
    assert_equal "Hello world!", mail.subject
    assert_equal "Hello?", mail.body
  end
end
```

他にもソースや fixture から受信メールを作成する為のヘルパーメソッドが提供されています。

クラスについてのより詳細は、https://edgeapi.rubyonrails.org/classes/ActionMailer/TestCase.html をご参照下さい。

2.13 Rails::Generators::TestCase

Rails で提供している generator(ファイルを生成する為の仕組み) は、カスタマイズ可能で、ユーザが任意の generator を追加出来るようになっています。Rails::Generators::TestCase はその generator のテストの為のクラスで、ユーザが追加した generator に対してテストを行えるようになっています。

例えば、"app/forms"配下にファイルを生成する generator があるとします。

リスト 2.16: form generator

```
class FormGenerator < Rails::Generators::NamedBase
  source_root File.expand_path('templates', __dir__)

  def create_form_file
    template "form.rb",
      File.join("app/forms", class_path, "#{file_name}_form.rb")
  end
end
```

この generator に対して、次のようにテストを行う事が出来るようになっています。

リスト 2.17: form generator test

```
class FormGeneratorTest < Rails::Generators::TestCase
  tests FormGenerator
  destination Rails.root.join('tmp/generators')
  setup :prepare_destination

  test "generator runs" do
    assert_nothing_raised do
      run_generator ["user"]
```

```
    end

    assert_file "app/forms/user_form.rb"
  end
end
```

なお、テストを実行すると実際に generator を実行しファイルの生成を行います。ファイルの生成先は `destination` で指定出来ます。tmp のような、一時ファイルが生成されても良いディレクトリ以外は指定しないよう注意してください。

クラスについてのより詳細は、https://edgeapi.rubyonrails.org/classes/Rails/Generators/TestCase.html をご参照下さい。

第 3 章

テストの為の機能たち

前章では、Rails が提供しているテスト用のクラスについて説明しました。Rails ではそれらのクラス以外にもテスト実行する為の機能が提供されています。本章では、これらテストの為の仕組みについて説明します。

3.1 Fixtures

Fixtures はテスト用のデータを作る為の機能です。基本的にはテストで使用する事を想定しています。

テスト用のデータの作成といえば factory_bot が有名です。factory_bot が独自 DSL でデータを定義するのに対して、Fixtures では YAML でデータの定義を行います。

リスト 3.1: users.yml

```
taro:
  email: taro@example.com
  name: Taro

hanako:
  email: hanako@example.com
  name: Hanako
```

Rails で YAML というと、ERB の記法を使用出来るようにしている事が多いのですが、Fixtures も例に漏れずそうです。YAML としてパース処理を行う前

に、ERB で前処理が行われます。そのため、ファイル内で ERB の記法を使用出来ます。

リスト 3.2: YAML 内で ERB 記法を使用

```
<% 100.times do |n| %>
user_<%= n %>:
  email: <%= "user#{n}@example.com" %>
  name: <%= "user#{n}" %>
<% end %>
```

処理の流れ

Fixtures は下記のような流れで処理を行います。

1. "test/fixtures"配下の YAML ファイルをロード
2. YAML ファイルから対象のテーブル名を取得&データを組立
3. 対象のテーブルから登録されているデータを削除&データの登録

まず"test/fixtures"配下にある、拡張子が"yml"になっているファイルがまとめて読み込まれます。これはサブディレクトリ配下のファイルも全て読み込まれます。

次に、YAML ファイルから対象のテーブル名を取得&データの組立を行います。テーブル名はデフォルトだとファイル名から取得されます。例えば"users.yml"というファイルがあった場合、"users"というテーブルに対して SQL を実行します。

サブディレクトリ配下については、そのサブディレクトリ名が指定された状態でテーブル名が組み立てられます ("admin/users.yml"というファイルがあった場合、テーブル名は"admin_users"になる)。

最後に登録されているデータの削除&データの登録を行います。データの削除を行う事で、仮に既にデータが登録されていても、必ず Fixture に定義されたデータのみが登録されている状態にします。この削除&登録は全てのテーブルに対する処理が 1 つの SQL でまとめて実行されるようになっています。削除&登録も 1 つの SQL にまとめられています。

トランザクショナルテスト

Fixtures では、テスト実行前にトランザクションを開始&テスト終了時にロールバックを実行し、テスト内で作成したデータがデータベースに残らないようにしています。これにより、テスト毎に毎回 Fixture の削除&登録処理をしないでも、同じ状態でテストが実施出来るようにしています。

この挙動は、use_transactional_tests で変更可能で、use_transactional_tests に false を指定すると、トランザクションを使用する代わりに、テスト毎にデータの削除&登録を行うようになります。

これはテストを実行するスレッドが 1 つの場合問題無いのですが、スレッドが複数あって、それぞれ違うデータベースのコネクションを所有した場合問題になります。例えば、Capybara を使用してテストを行う場合、アプリケーションサーバと Capybara が起動するスレッドが別々になる為、この問題が発生します。

Rails では、この問題に対応する為に、異なるスレッドが同じデータベースのコネクションを使用する機能を実装し、Fixtures を使用する場合のみその機能を有効化するようにしています。そのため、Fixtures を使用している場合 Capybara を使用したテストで DB に登録したはずのデータが参照できない、という問題は発生しないようになっています。

File fixtures

新規に Rails アプリケーションを作成すると、"test/fixtures"ディレクトリ配下に"files"というディレクトリが合わせて作成されます。このディレクトリにはテストで使用するファイル (テキストファイル、動画、PDF 等々) を格納する為のディレクトリです。

このディレクトリに格納されたファイルは、file_fixture というメソッドで簡単に参照出来るようになります。

リスト 3.3: file_fixture

```
file_fixture("example.txt") #=> Pathname クラスのインスタンス
file_fixture("example.mp3").size
```

名前が似ているのですが、これはここまで説明してきた Fixtures とは別の、単純にファイルを参照する為の仕組みです。"test/fixtures/files"は YAML ではなくただのファイルを格納する想定のディレクトリである、という事だけご認識頂ければと思います。

3.2 Test Runner

Test Runner はテストを実行する為の機能です。RSpec を使用している場合"rspec"スクリプトを使用してテストを実行しているかと思います。minitest にはこの"rspec"スクリプト相当のものがなく[*1]、ruby か rake を使用してテストを実行していました。しかしこれだと、特定の行のテストを実行する、複数のテストをまとめて実行する、等が実施しづらい、という問題がありました。

それらの問題を解決する為に、Rails では独自にテストを実行する為の仕組みを Rails 5.0 から導入しました。これにより、上記のような任意のテストの実行や、失敗したテストについて再実行するコマンド等の表示が出来るようになりました。これが Test Runner と呼ばれている機能です。Test Runner は、"rails test"コマンドで実行出来ます。

リスト 3.4: rails test

```
$ ./bin/rails test test/models/user_test.rb:5
Run options: --seed 1833

# Running:

.

Finished in 2.243887s, 0.4457 runs/s, 0.4457 assertions/s.
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips

$ ./bin/rails test test/models/receive_mail_test.rb
Run options: --seed 41533

# Running:
```

[*1] minitest 5 系までの話です。6 系だと実行用のスクリプトを追加する予定があるらしいです。

```
F

Failure:
ReceiveMailTest#test_validation [test/models/receive_mail_test.rb:5]:
Expected false to be truthy.


bin/rails test test/models/receive_mail_test.rb:4



Finished in 2.110546s, 0.4738 runs/s, 0.4738 assertions/s.
1 runs, 1 assertions, 1 failures, 0 errors, 0 skips
```

Test Runner は他にもテスト結果に色をつけたり、失敗したら即座にテストを終了する為の機能が組込まれています。

なお、ディレクトリを指定するとそのディレクトリ配下にあるテストを全て実行するようになっているのですが、"test"ディレクトリを指定してもシステムテストは除外されるようになっています。これはシステムテストは処理が重くなる為、通常のテストスイートとは別に実行するべき、という考えによるものです。システムテストを実行したい場合、"test/system"を明示的に指定してテスを実行するか、`test:system` タスクを使用してテストを実行する必要があります。

minitest plugin

minitest は本体の機能が大分少ないのですが、代わりに拡張機能 (plugin) を追加する為の機能がデフォルトで提供されています。

Test Runner は、幾度かの作り直しの結果、現在は minitest の拡張機能として実装されています。Test Runner のコードを見たい方は minitest の拡張機能の使い方を把握しておくと読みやすいと思います。minitest の拡張機能については、http://y-yagi.tumblr.com/post/87286444980/minitest%E3%81%AEextensions%E3%81%AE%E4%BD%9C%E3%82%8A%E6%96%B9 をご参照下さい。

3.3 Parallel Testing

名前の通りテストを並列で実行する為の仕組みです。Rails 6.0 の目玉機能の 1 つ。

parallelize というメソッドが提供されており、ユーザはそのメソッドを使用するだけでテストが並列で実行されるようになります。

リスト 3.6: parallelize

```
class ActiveSupport::TestCase
  parallelize(workers: :number_of_processors)
end
```

テストを実行するワーカーは workers 引数で指定可能です。workers には基本的には数字を指定するのですが、:number_of_processors という値を指定した場合、自動でマシンのコア数の値を指定してくれます。

並列化は、デフォルトではプロセスを fork して行われます。fork が使えない環境では、代わりにスレッドを使用する事が出来ます。スレッドを使用したい場合は、with 引数に、:threads と指定すれば OK です

リスト 3.6: parallelize

```
class ActiveSupport::TestCase
  parallelize(workers: :number_of_processors, with: :threads)
end
```

並列化 (プロセス)

並列化にプロセスを指定した場合と、スレッドを指定した場合とでテスト実行時の挙動が幾つか異なります。

プロセスの場合、プロセスを fork 後データベースを新規に作成、及び、schema のロードを行い、テストではその新規に作成したデータベースを使用するようになります。そのため、プロセス毎に違うデータベースを使用するようになり、プ

ロセス間でデータベースへの処理がが競合しないようになっています。プロセス間でのデータのやりとりには、dRuby[*2]が使われるようになっています。なお、作成されたデータベースはテストの後処理で削除されるようになっています。

処理の流れは次のようになっています。

1. dRuby のサーバを起動
2. プロセスを fork
3. テストの前処理を実施
4. テストを実行
5. テスト結果を dRuby のサーバに送信
6. テストの後処理を実施

3〜6 が子プロセスで行われる処理です。3 でデーターベースの作成、6 でその作成したデーターベースの削除を行います。前処理・後処理はユーザが任意の処理を登録する事も可能です。

並列化 (スレッド)

スレッドの場合、(執筆時点では) このデータベースの作成は行われず、全てのスレッドで同じデータベースに対して処理を行うようになっています。テストの内容によってはスレッドの排他制御を意識する必要がありますので注意してください。また、スレッド間でのデータの共有は行われない為、テストとは別にスレッドを起動するシステムテストが正しく動作しない、という制限があります。

なお、minitest には元々 parallel executor という、スレッドを使って並列でテストをする仕組みがあり、Parallel Testing はその仕組みに依存しています。スレッドの場合の処理は minitest の parallel executor をそのまま使用しているだけです。そのため、他のテストフレームワークでそのまま使用出来るような機能ではありませんのでご注意下さい。

[*2] https://docs.ruby-lang.org/ja/latest/library/drb.html

付録 A

ユーザ向け API と内部向け API

ここからはおまけです。どこかで説明を入れたかったが、適切な場所が無かった話を入れています。まずはユーザ向け API と内部向け API についてです。

Rails は、ユーザ向け API(user-facing API) と internal API(内部向け API) が明確に分かれています。

API doc[*1]に記載されている API だけがユーザ向け API で、それ以外の API は全て Rails 内部で使用する内部向け API(internal API) になります。

ユーザ向け API と内部向け API は、API の挙動を変更する際の対応が異なります。ユーザ向け API はリリース間で非互換になる対応は行いません。非互換になる対応が必要な場合、基本的には deprecation サイクル (挙動が変わる旨ユーザに通知してから挙動を変更する) を行うようにしており、バージョンが上がってもいきなり振る舞いが変わる事が無いようにしています。

なお、まれにこの deprecation サイクルを行わない変更があります。その場合、アップグレードガイド[*2]に説明を記載するようになっています。ユーザ向け API の挙動が変わって、かつ、それがアップグレードガイドに説明が無い場合、バグかガイドへの記載漏れです。

逆にいうと、内部向け API はバージョン間で非互換になる対応が行われます。

内部向け API を使うとバージョンアップグレード時の難易度が上がります (前のバージョンで通っていた処理がいきなりエラーで何故か落ちる、特定の引数を渡した時の挙動が変わってる等々) ので、内部向け API は Rails アプリケーションでは使用しない事を推奨します。どうしても使用したい API が内部向け

[*1] https://api.rubyonrails.org/

[*2] https://edgeguides.rubyonrails.org/up_ruby_on_rails.html

になっている場合、その API を公開 API にするよう Rails に PR を送るか、内部向け API を使用している部分を gem にして、Rails アプリケーションとは別に管理しテストしやすくする事をオススメします。

因みに、API doc には、リリース済みの gem に基づいて作成された https://api.rubyonrails.org/ と、GitHub の最新のコミットに基づいて作成された https://edgeapi.rubyonrails.org/ の 2 つがあります。https://api.rubyonrails.org/ は基本的には Rails がリリースされた時にしか更新されないので、最新の内容を確認したい場合、https://edgeapi.rubyonrails.org/ の方でご確認して下さい。

付録 B

rspec-rails

rspec で Rails アプリケーションのテストで使用する場合 rspec-rails gem を使用するかと思います。この gem で提供されている Rails の機能をテストする為の仕組みは、2 章で紹介したクラスたちを使用しているのではなく、それらのクラスで使用しているモジュールを直接使用して実装されています。

そのため、rspec-rails で提供しているテストの為の機能と Rails のテストの為の機能では若干挙動が異なります。また、rspec-rails では Rails 内部向けの API を使用している箇所もあります。そのため、Rails のバージョンをあげたら rspec の挙動がおかしくなった、という事がありえます。ご注意下さい[*1]。

[*1] rspe-rails は開発がアクティブなので Rails の正式リリースより前に対応が行われる事が殆どだと思いますが。

Railsのテストの仕組み

2019年4月1日　初版第1刷　発行
2019年8月26日　初版第2刷　発行
著　者　y-yagi