# 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。

普段 Rails を使用してアプリケーションを作られている方は、一緒にテストも書かれているかと思います。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、その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。

そのため、Rails を使用したアプリケーションでのテストについての話ではなく、Rails が提供しているテストについての話です。仕組みの使い方について詳細には触れていません。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.beta3, commit: 6a5c8b9) 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。そのため、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

# Rails のテスト

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 アプリケーションを開発されている場合、RSpec をお使いの方が多いのではないかと思います。しかし実は 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 RSepc 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。色々と機能拡張をおこなっています。

例えば、minitest では否定の assert を行うのに、`refute` というメソッド、及び、`refute` で始まる各種 assert 用の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 で始まる各種 assert 用が提供されており、そちらを使用することを推奨しています。先の例だと、代わりに assert_not_equal を使用する必要があります。

リスト 1.2: refute_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に追加されました。その後、refute に移行する？というような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 の設定[*2]もあります。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

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

[*1] 当時のちゃんとした議論が見つからなかったので推測混じりなのですが、どうも 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。恐らくレーサー。

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

これも Rails が提供しているメソッドで、minitest だけを使用している場合、この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」を正確ではなく、「Rails は minitest を拡張した独自の仕組みを使用している」が正確な表現になります。

## 1.3　テスト用のクラス

Rails というフレームワーク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、template の表示を行う Action View、メール送信の為の Action Mailer、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。

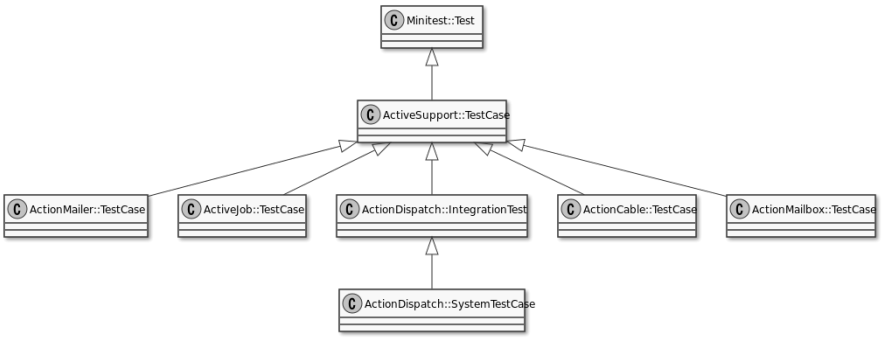


図 1.1: クラス図

紙面の都合上大分割愛していますが、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 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

# テスト用のクラスたち

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、次のような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 |

名前から何となく何の為のクラスが想像出来るのではないかと思います。基本的には各ライブラリ毎にクラスが提供されているので、そのライブラリを使用し

た機能のテストをしたい場合、それぞれ向け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 のような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、特別な処理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った場合、このクラスに機能を追加する、このクラスで module を include する、等を行う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、ActiveSupport::Testing::TimeHelpers という時間を処理する為の `travel`、`travel_to` 等のメソッドが定義されている module が include されています。

リスト 2.1: travel_to

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

# 1 日前の移動
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
```

また、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になっています。

例えば、何か既存の機能を deprecate にしたい場合に、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。これは Rails 全てのライブラリで使用したいアサーションなので、ActiveSupport::TestCase で必要な module が incldue されるようになっています。

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
```

クラスやアサーションについての詳細は、https://edgeapi.rubyonrails.org/classes/ActiveSupport/TestCase.html を参照してください。

## 2.3 ActionMailer::TestCase

メール (送信) のテストの為のクラスです。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配列で管理す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を提供しています。

リスト 2.3: assert_emails

```
test "invite friend" do
  # invite_friend_url に POST したら招待用のメールが送信される
  assert_emails 1 do
    post invite_friend_url, params: { email: 'friend@example.com' }
  end
end
```

assert_emails は同期処理 (@code{deliver_now})、非同期処理 (@code{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。

アサーションについての詳細は、https://edgeapi.rubyonrails.org/classes/ActionMailer/TestHelper.html を参照してください。

## 2.4 ActiveJob::TestCase

ジョブのテストの為のクラスです。メールと事なり、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。当然、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: assert_enqueued_jobs

```
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
```

どのような引数が指定されたかチェックしたり、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。アサーションについての詳細は、https://edgeapi.rubyonrails.org/classes/ActiveJob/TestHelper.html を参照してください。

## 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 の helper のテスト用のクラスです。ActionView::TestCase では heler を使用する為に必要な controller や view の生成処理を行ってくれます。

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 より helper のテストはそもそも生成されなくなり[*1]、このクラスが使用される事は無くなりました。また、helper 単体でのテストはあまり意味が無い[*2]のでは、という声もあり、このクラスを使用する事は基本的に無いかと思います。

## 2.6 ActionController::TestCase

コントローラーのテストの為のクラスです。特定のコントローラー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

```
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 } })
```

[*1] https://github.com/rails/rails/commit/a34b6649d061977026db7124d834faccdf5bd8ef

[*2] Integration テストや System テストビューのテストと合わせてやるのが良いのでは、という意見が多いです。

```
  end

  assert_redirected_to user_url(User.last)
end
```

しかし同様にコントローラー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[*3]、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、コントローラー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大幅に改善され、実行速度の差は大分縮まりました。結果、コントローラーのテストでも 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ーのテストでも 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。

## 2.7 ActionDispatch::IntegrationTest

コントローラー (とルーティングテスト) の為のクラスです。基本的には ActionController::TestCase と同じ目的です。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

[*3] ActionController::TestCase は送信先にコントローラー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。

```
  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
```

他にも、ActionDispatch::IntegrationTest ではルーティングを確認する為のアサーションが使えるようになっています。

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"

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

クラスやアサーションについての詳細は、https://edgeapi.rubyonrails.org/classes/ActionDispatch/IntegrationTest.html を参照してください。

## 2.8 ActionDispatch::SystemTestCase

ActionDispatch::SystemTestCase はシステムテストの為のクラスです。Capybara[*4]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: システムテスト

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

```
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
```

ActionDispatch::SystemTestCase は ActionDispatch::IntegrationTest を継承しており、ActionDispatch::IntegrationTest + Capybara のラッパー的な機能を提供しています。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: ブラウザを指定

```
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。

## 2.9 ActionCable::TestCase

Rails 6.0 から追加された Action Cable のテストの為のクラスです。Action Cable が追加された当初、ユニットテスト用のクラスは 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、ユニット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、ユニットテスト用の仕組みがあった方が良いよね、という声が強まり、はれて Rails 本体に機能が追加されました。

なお、元々 action-cable-testing[*5]という gem があり、その機能を Rails 本体にインポート[*6]した形になります。その為、Rails 6.0 より前で Action Cable のユニット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

```
assert_broadcasts('messages', 1) do
  ActionCable.server.broadcast 'messages', { text: 'hello' }
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。

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

[*6] インポート処理を行ったのも action-cable-testing の作者です。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。

## 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

```
# 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
```

## 2.11 ActionCable::Channel::TestCase

Action Cable のチャンネルに関するテストの為のクラスです。

## 2.12 ActionMailbox::TestCase

Rails 6.0 で追加された Action Mailbox のテスト

## 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、下記のような"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 2.13: FormGenerator

```
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記載する事ができます。

リスト 2.14: FormGenerator

```
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@<code{destination}で指定出来ます。tmp のような、ゴミ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。

# Railsのテストの仕組み

2019年4月1日　初版第1刷　発行

著　者　y-yagi