第 15 章

[Rails & New Relic] コントローラの filter(before_action など) の実行時間を計測する仕組みを作った

こんにちは！　2020 年 10 月に Wantedly にサーバーサイドエンジニアとしてジョインした工藤 (@kudojp) と申します。

この記事は、僕が入社前の 2020 年 8 月に参加した弊社のサマーインターン[*1]中の取り組みをまとめたものです。

具体的な内容としては、New Relic を使った Rails アプリケーションのパフォーマンス測定をより便利にするツールをその実装手順とともに紹介します。このツールを使うことで、Rails アプリケーションにおいてコントローラで指定された filter の実行時間が計測され、New Relic One 上で監視することが可能になります。(この記事は Rails の version が 6-0-stable でのお話です。)

ちなみに、New Relic は、アプリケーションのパフォーマンス監視プラットフォームであり、filter とは以下のようにコントローラにおいて before_action や around_action で定義される callback 関数のことです。

sample_controller.rb

```
SampleController < ApplicationController
  before_action set_current_user ### これがfilter

  def index
    ...
  end
  ...
end
```

以降、次の構成でお話を進めていきます。

1. 開発に至る経緯

[*1] https://www.wantedly.com/projects/436019

2. 下準備 1<newrelic_rpm の紹介>
3. 下準備 2<binding.pry でコントローラが filter を保持している様を観察する>
4. 実装 1<メソッド指定の filter の実行時間を測定する>
5. 実装 2<ブロック指定の filter の実行時間を測定する>
6. 完成したツールの使用法
7. 後書き

なお、実装 1 終了時のスクリプトでも、十分に有用なツールとなります。また、実装 2 ではメタプログラミングを多用しますので、読み物としてお楽しみください。

15.1 開発に至る経緯

Wantedly のメインサーバは現在 Rails で動いており、僕はインターン中にこの Rails サーバーのパフォーマンスチューニングに取り組みました。

パフォーマンスチューニングといえば「クエリを見直すことで N+1 問題を解消して線形時間かかっていたパフォーマンスを定数時間まで縮めることに成功しました！」というハッピーエンドを思い描きがちですが、自分の場合はそんなに単純にはいきませんでした。自分がチューニングを行った API エンドポイントは、ユーザー認証の部分でかなり時間を食っており、これを改善するためには devise, warden の内部実装を深くまで読み込んで理由を精査していくほかないことが判明しました。(この修正は結局インターン中には行いませんでした。) この「ユーザーの認証の部分でかなりの処理時間を食っている」ことを特定するまでに以下の過程を辿りました。

1. コントローラの該当 action のメソッド内の実装や、ActiveRecord のログとして吐き出されるクエリを眺める。また実行に時間がかかっていそうな行に目星をつけ、`rack-lineprof` を使い、実行時間を調べる。
2. どうやら action のメソッド自体に時間がかかっているのではなく、そのメソッドの前後に挟み込まれた filter に時間がかかっていそうだということに気が付く。
3. そのコントローラと、それが継承している親コントローラで定義されている十数個の filter に関して、その filter メソッド内の実装をまるっと `t = Time.current` と `puts Time.current - t` で挟み込み、実行時間を計測することで、実行に時間がかかっていそうな filter を特定する。(`rack-lineprof` を使っても良かったでしょう)

3 の作業をしながら、「Rails のコントローラの filter の実行時間を楽に計測でき、なおかつそれを New Relic One で継続的に観察できる仕組みがあれば」との思いが溢れ出しました。

その理由としては、(1) この作業は大変に泥臭いですし、時間計測したい filter のコードに変更を加えなければなりません。(2) また、今後 Rails のコントローラの filter の実行時間を測定したい状況が再び発生する可能性は十分にあると考えられます。(3) なおかつ、filter をあるコントローラで設定するとそれを継承するコントローラでもそれが使用される (詳細は後述) ため、それぞれの

filter の実行時間を継続的に計測して監視することは、重い処理が複数のコントローラで共有されてアプリケーションの多くのエンドポイントのレスポンスを遅らせていた、という状況に気づけることがあるかもしれないと考えられます。

ちなみに、New Relic が Rails アプリケーションに提供しているベーシックな計測方法を使えば、Rails の各コントローラの各アクションの処理にどれだけの処理時間がかかるか、の計測は比較的簡単にできます。しかしながら、その処理時間の中で、filter のそれぞれの処理にどれだけの時間がかかったか、というところまでブレイクダウンしてデータを見ることはできません。

それではまず、New Relic に関して、この記事を読む上で最低限必要なことを簡単に紹介します。

15.2 下準備 1<newrelic_rpm の紹介>

New Relic はアプリケーションのパフォーマンス監視等を可能とする監視プラットフォームです。提供されるライブラリをアプリケーションに導入することで、可視化されたデータを New Relic One で確認することができます。

今回の対象は Rails アプリケーションであるため、New Relic Ruby エージェントの公式 gem である `newrelic_rpm` を導入します。この gem をインストールして、New Relic One で発行される `new_relic.yml` を Rails レポジトリの config/配下に置くことで、Rails の各コントローラの各アクションの処理にかかる時間のデータを New Relic One 上で表示できるようになります。

これだけでもかなり有用ですが、以下のように、よりカスタマイズした使い方もできます。

- 連続する行を指定して、実行時間を計測する
- 特定のインスタンスメソッドを指定して、実行時間を測定する

どれも今回の記事を理解していただく上で避けては通れないので、駆け足で紹介します。より詳細を知りたい方は、New Relic の Ruby custom instrumentation の公式ドキュメント[*2]をご覧ください。

15.2.1 連続する行を指定して、実行時間を計測する

コードの連続する行の実行時間をするには、`Tracer.in_transaction` メソッドの引数に、コードをラップしたブロックを渡せば良いです。以下の例では、some_code1 と some_code2 の実行時間の合計が測定されます。

sample.rb

```
require 'new_relic/agent/tracer'


Tracer.in_transaction(partial_name: 'Complex/process', category: :task) do
  some_code1
```

[*2] https://docs.newrelic.com/docs/agents/ruby-agent/api-guides/ruby-custom-instrumentation

```
  some_code2
end
```

15.2.2 特定のインスタンスメソッドを指定して、実行時間を測定する

あるクラスのインスタンスメソッドの実行時間を計測したい場合には、そのクラスの定義式内で計測したいインスタンスメソッド名のシンボルを引数にして add_method_tracer を実行すれば良いです。以下の例では、Foo#hoge_method が実行されるたびに、その実行時間が計測されます。

sample.rb

```
require 'new_relic/agent/method_tracer'

class Foo
  include ::NewRelic::Agent::MethodTracer

  def hoge_method
    ...
  end

  add_method_tracer :hoge_method, 'Custom/hoge_method'
end
```

さて、それではこれらを使ってコントローラの filer の実行時間を測定する仕組みを作っていくわけですが、その前に、「彼を知り、己を知れば百戦危うからず」ということで、Rails のコントローラがどのように filter を保持しているのかを確認しましょう。

15.3 下準備 2<binding.pry でコントローラが filter を保持している様を観察する>

適当なコントローラ TestController を実装して、ActionController がどのように before_action で指定された callback 関数を保持しているのかを確認してみましょう。

Rails の ActionController は、before_action の指定方法は、メソッド指定とブロック指定の 2 つがあることを念頭に入れて読み進めてください。

/app/controllers/test_controller.rb

```
class  TestController < ApplicationController
    before_action  :my_first_before_callback  # メソッド指定
    before_action  do                         # ブロック指定
        puts  'my second before callback'
        sleep  1
    end

    def  my_first_before_callback
        puts  'my first before callback'
        sleep  1
    end

    def  test_action
        puts  'request dispatched!'
        sleep  1
    end
```

```
    binding.pry
end
```

ルーティングもしておきます。

/config/routes.rb

```
Rails.application.routes.draw  do
  ...(中略)...
  get  'test/test_action', :to => "test#test_action"
end
```

$bundle exec rails s してみます。ザクっと調べたところ、Rails のコントローラは filter を callback 関数の配列として保持しており、self._process_action_callbacks() でこれを呼び出すことができるようです。

callbacks を見る

```
  [1] pry(main)> .

  From: /Users/user1/repos2/wantedly/app/controllers/test_controller.rb:12 :

       7:          def  my_first_before_callback
       8:                  puts  'my first before callback'
       9:    end
      10:
      11:    binding.pry
   => 12: end

  [2] pry(TestController)> self._process_action_callbacks()
  => #<ActiveSupport::Callbacks::CallbackChain:0x00007fb2c2bdce88
   @callbacks=nil,
   @chain=
    [#<ActiveSupport::Callbacks::Callback:0x00007fb2c735deb0
      @chain_config=
       {:scope=>[:kind],
        :terminator=>#<Proc:0x00007fb2b28506f8 /Users/user1/.rbenv/gems/2.7.0/gems/actionpack
-6.0.3.4/lib/abstract_controller/callbacks.rb:34 (lambda)>,
        :skip_after_callbacks_if_terminated=>true},
      @filter=:set_controller_action,
      @if=[],
      @key=:set_controller_action,
      @kind=:around,
      @name=:process_action,
      @unless=[]>,

    ...(非常に長いので中略)...

     #<ActiveSupport::Callbacks::Callback:0x00007fb2c2bdd4a0
      @chain_config=
       {:scope=>[:kind],
        :terminator=>#<Proc:0x00007fb2b28506f8 /Users/user1/.rbenv/gems/2.7.0/gems/actionpack
-6.0.3.4/lib/abstract_controller/callbacks.rb:34 (lambda)>,
        :skip_after_callbacks_if_terminated=>true},
      @filter=:my_first_before_callback,
      @if=[],
      @key=:my_first_before_callback,
      @kind=:before,
      @name=:process_action,
      @unless=[]>,

     #<ActiveSupport::Callbacks::Callback:0x00007fb2c2bdcfa0
      @chain_config=
       {:scope=>[:kind],
        :terminator=>#<Proc:0x00007fb2b28506f8 /Users/user1/.rbenv/gems/2.7.0/gems/actionpack
-6.0.3.4/lib/abstract_controller/callbacks.rb:34 (lambda)>,
        :skip_after_callbacks_if_terminated=>true},
```

```
      @filter=#<Proc:0x00007fb2c2bdd108 /Users/user1/repos2/wantedly/app/controllers/
test_controller.rb:3>,
      @if=[],
      @key=99960,
      @kind=:before,
      @name=:process_action,
      @unless=[]>],

   @config=
    {:scope=>[:kind],
     :terminator=>#<Proc:0x00007fb2b28506f8 /Users/user1/.rbenv/gems/2.7.0/gems/actionpack
-6.0.3.4/lib/abstract_controller/callbacks.rb:34 (lambda)>,
     :skip_after_callbacks_if_terminated=>true},
   @mutex=#<Thread::Mutex:0x00007fb2c2bdcde8>,
   @name=:process_action>
```

ActiveSupport::Callbacks::CallbackChain は、ActiveSupport::Callbacks::Callback の配列を、インスタンス変数@chain として保持しています。上記のスニペットでは省略しましたが、**この@chain の配列は、この TestController が継承している親コントローラで定義されたものも含まれています**。また、@chain の配列内には、before_action で指定された callback 関数以外にも、after_action で設定された callback 関数も含まれます。(この場合、Callback は`@kind=:after` となっています)

また、`before_action` の指定方法は、メソッド指定とブロック指定の 2 つがあると先述しましたが、この指定の仕方によって各 ActiveSupport::Callbacks::Callback のインスタンス変数@filter の型が異なることが確認できます。`:my_first_before_callback` は型がシンボルになっており、これはメソッド指定された before_action filter に相当します。一方、`#<Proc:0x00007fb2c2bdd108 /.../app/controllers/test_controller.rb:3>`は型が Proc になっており、これはブロック指定された before_action filter に相当しています。

実は、filter 測定の実装は@filter の型 (要は filter の指定方法) によって、全く異なります。次節から、`Symbol` 型の `Callback` の実行時間の測定の仕組み作りを第 1 部で、`Proc` 型の `Callback` の実行時間の測定の仕組み作りを第 2 部で行っていきます。

15.4 実装 第 1 部<メソッド指定の filter の実行時間を測定する>

15.4.1 実装方針

コントローラのインスタンスメソッドで定義される `callback` 関数を `before_action` での指定された filter の実行時間を計測する方法をざっと考えてみましょう。考えるまでもなく、コントローラの各クラス内で、newrelic_rpm で提供される `add_method_tracer` メソッドにその filter のメソッド名を指定する方法が思い浮かびます。

ただし、測定したい `filter` を含むコントローラクラスのファイルを開き、そのコントローラで測定したい `callback` 関数を `add_method_tracer` で毎回指定するというのは手間がかかります。そこで以下のように、after_initialize のタイミングで既に定義されたコントローラクラスを上書きすることにしました。

1. 環境変数 FILTER_TRACED_CONTROLLERS を新しく定義する。この環境変数には、測定したい filter を定義するメソッド名 (複数指定可能) を全角区切りで設定する。
2. /config/initializers/配下に trace_controller_callbacks.rb ファイルを置き、この内部で上記で指定されたコントローラの filter として設定されているメソッドを trace するように Rails.application.config.after_initialize 内で既に定義されたコントローラクラスを上書きする。

ちなみに、環境変数 FILTER_TRACED_CONTROLLERS に、測定したい callback を含むコントローラ名の指定なんてケチくさいことを言わずに、全てのコントローラの全ての filter の実行時間を計測してやったらいいのでは？ という太っ腹な意見もあるでしょう。実は最初はこの方法を取ろうとしていたのですが、New Relic の newrelic-rpm は内部的にメタプログラミングを多用しており、(メタプロは一般に重いため)Rails サーバの起動時間がかなり遅くなってしまうという理由で却下しました。(具体的にはこの記事の最後に紹介する after_initialize ファイルを使って合計約 480 個のコントローラを上書きしようとしたところ、約 10 秒程度起動時間が長くなりました。)

ではこの実装方針でスクリプトを書いていきましょう。

15.4.2 after_initialize ファイルを書く

一晩頑張って、以下のスクリプトが完成しました。このファイルは filter_traced_controllers.rb と命名し、/config/initialize 配下に置くことにします。

filter_traced_controllers.rb

```
if ENV['FILTER_TRACED_CONTROLLERS'].present?
  Rails.application.config.after_initialize do
    begin
      ENV['FILTER_TRACED_CONTROLLERS'].split do |controller_str|
        begin
          controller_klass = controller_str.constantize
        rescue NameError
          Rails.logger.error "=====␣[Failure]␣#{controller_str}␣in␣FILTER_TRACED_CONTROLLERS␣
has␣not␣been␣found␣====="
          next
        end

        unless controller_klass.method_defined?(:_process_action_callbacks)    # このクラスがコ
ントローラではない場合
          Rails.logger.error "=====␣[Failure]␣#{controller_klass}␣in␣FILTER_TRACED_CONTROLLERS␣
is␣not␣a␣traceable␣controller␣====="
          next
        end

        controller_klass.class_eval do
          self.include ::NewRelic::Agent::MethodTracer
          self._process_action_callbacks().send(:chain).each do |callback|
            case callback.raw_filter    # 補足(1)
            when Symbol                 # 補足(2)
              self.add_method_tracer callback.raw_filter
            end
          end
        end
        Rails.logger.info "=====␣[Success]␣Filters␣of␣actions␣for␣#{controller_klass}␣will␣be␣
reported␣to␣New␣Relic␣====="
      end
    rescue StandardError => e
```

```
      Rails.logger.error "=====␣[Failure]␣Unexpected␣Error␣<#{e}>␣occurred␣when␣adding␣method␣
tracers"
    end
  end
end
```

まず、一番肝心な補足 (2) の箇所を最初に説明します。

補足 (2) `case callback.raw_filter; when Symbol`

先述したように、callback.raw_filter の型は Symbol と Proc の 2 種類があります。第一部の実装では、callback がインスタンスメソッドで指定された場合の処理を考えているので、callback.raw_filter の型が Symbol の場合の処理のみを記述しています。

第二部で、`case callback.raw_filter; when Proc` の分岐が加わる予定です。

補足 (1) `callback.raw_filter` **で callback 関数の実態を取得する**

ここでは、先程確認した ActiveSupport::Callbacks::Callback のインスタンス変数@filter を取得しています。ここにはちょっとした落とし穴があり、`callback.filter()` としてしまうと、@filter ではなく、@key というインスタンス変数が返ってきてしまいます。

ActiveSupport::Callbacks::Callback のソースコードを見てみましょう。

> https://github.com/rails/rails/blob/6-0-stable/activesupport/lib/active_support/callbacks.rb#L304-L305

```
class  Callback  #:nodoc:#
  ...(中略)...
  attr_accessor  :kind, :name
  attr_reader  :chain_config

  def  initialize(name, filter, kind, options, chain_config)
    @chain_config = chain_config
    @name = name
    @kind = kind
    @filter = filter
    @key = compute_identifier  filter
    @if = check_conditionals(options[:if])
    @unless = check_conditionals(options[:unless])
  end

  def filter; @key; end
  def raw_filter; @filter; end
  def filter; @key; end
  def raw_filter; @filter; end

  ...(以下略)..
 end
```

- filter メソッドで取得しているのは`@key` attribute
- raw_filter メソッドで取得しているのは`@filter` attribute

になっていることがわかります。

15.4.3 現時点でのスクリプト

filter_traced_controllers.rb

```
Rails.application.config.after_initialize do
  if ENV['FILTER_TRACED_CONTROLLERS'].present?
    traced_controllers = []
    ENV['FILTER_TRACED_CONTROLLERS'].split.each do |controller_str|
      begin
        traced_controllers << controller_str.constantize
      rescue
        puts "=====␣#{controller_str}␣in␣FILTER_TRACED_CONTROLLERS␣has␣not␣been␣found␣====="
      end
    end

    traced_controllers.each do |controller|
      begin
        controller.class_eval do
          self.include ::NewRelic::Agent::MethodTracer
          self._process_action_callbacks().send(:chain).each do |callback|
            case callback.raw_filter
            when Symbol
              self.add_method_tracer callback.raw_filter
            end
          end
        end
        puts "=====␣Filters␣of␣actions␣for␣#{controller}␣will␣be␣reported␣to␣New␣Relic␣====="
      rescue
        puts "=====␣#{controller}␣in␣FILTER_TRACED_CONTROLLERS␣has␣not␣been␣found␣====="
      end
    end
  end
end
```

15.4.4 動作検証してみる

それでは、先程の TestController を使って動作検証してみましょう。1. $export FILTER_TRACED_CONTROLLERS=TestController で環境変数を設定 2. $bundle exec rails s 3. ブラウザで localhost:3000/test/test_action

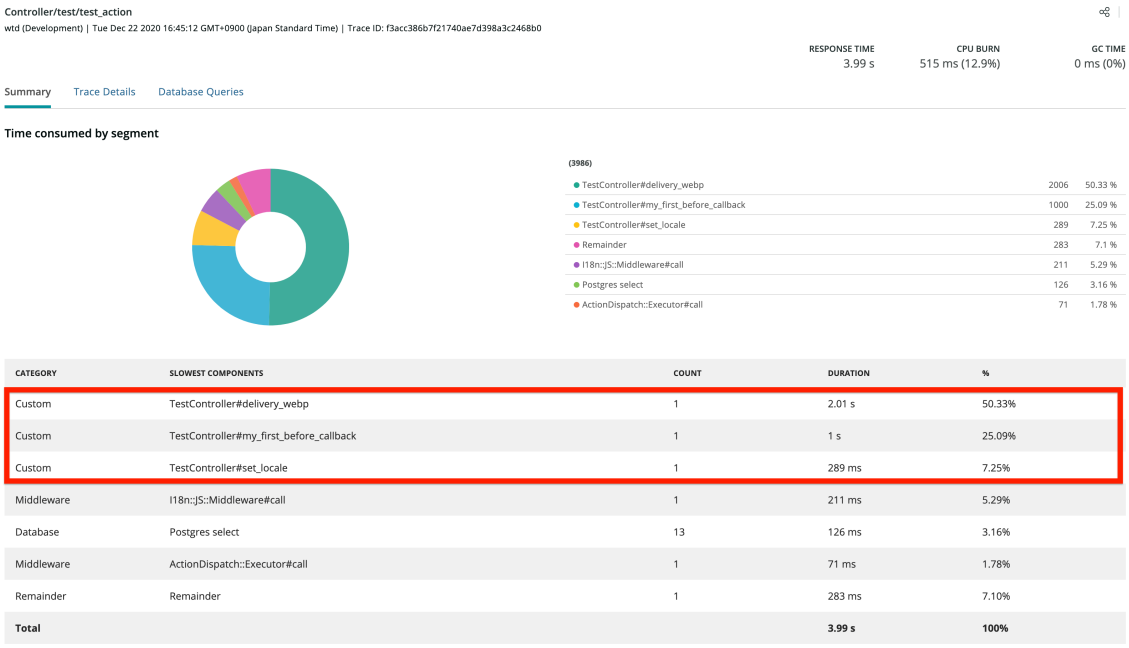

図 15.1　New Relic One 上で計測ができていることの確認

Custom というカテゴリーで `TestControllers#my_first_before_callback` filter の実行時間が取れていることが確認できます。

(この他にも、同じく Custom というカテゴリーで `TestController#delivery_webp` と `TestController#set_locale` という行があることに気がつくと思います。これは TestController で継承している親クラスのコントローラである ApplicationController で設定されている filter です。)

15.4.5　まとめ

第 1 部では、メソッドで指定された filter の実行時間を計測する方法を紹介しました。

正直、コントローラの filter はほとんどの場合メソッド定義で指定していることでしょうから、今の段階のスクリプトでも十分に有用なものでしょう。また、もし仮にブロック指定で定義しているものがあったとしても、それが時間を計測したいほど重い処理を含めていることはほぼないでしょう。

ですが、エンジニアという生き物には例えそれが自己満足だと揶揄されても、技術と弄びたくなる夜があるものです。ということで、ブロック指定の filter の実行時間を測定できるように実装したので、紹介させていただきます。読み物としてお楽しみください。

15.5 実装 第 2 部<ブロック指定の filter の実行時間を測定する>

コントローラのブロック指定で定義される callback 関数を before_action での指定する方法を一晩ほど考えた結果、コントローラが Proc として保持している Callback を、「その処理を trace_execution_scoped で挟み、なおかつその処理を包んだ Proc」で置き換えてやるという方法を思いつきました。要は、Proc を、Proc でラッピングしてやるのです。

15.5.1 実装イメージ

Proc を Proc で wrapping する

```
self._process_action_callbacks().send(:chain).each do |callback|
  if callback.raw_filter.class == Proc
    original_filter = callback.raw_filter     # 補足(1)
    wrapped_filter = Proc.new{                # 補足(2)
      self.class.trace_execution_scoped("Custom/proc_filter_of_interest") do
        original_filter.call
      end
    }
    callback.instance_variable_set(:@filter, wrapped_filter)  # 補足(3)
  end
end
```

補足 (1)

元の Callback の、@raw_filter を取得しています。先述したように、@raw_filter attribute を取得するには #filter メソッドを使用しなければなりません。

補足 (2)

元の proc 関数を call で実行する original_filter.call、を trace_execution_scoped で挟み込みむことで、実行時間の計測ができるようになります。この処理を内部に含む Proc を新たに初期化しました。

補足 (3)

元の Callback のインスタンス変数@filter に作成した wrapped_filter を仕込みました。これを after_initialize ファイルに盛り込むと以下のようになりました。

filter_traced_controllers.rb(抜粋)

```
if ENV['FILTER_TRACED_CONTROLLERS'].present?
  Rails.application.config.after_initialize do
    begin
      ENV['FILTER_TRACED_CONTROLLERS'].split do |controller_str|
        ...(中略)...

        controller_klass.class_eval do
          self.include ::NewRelic::Agent::MethodTracer
          self._process_action_callbacks().send(:chain).each do |callback|
```

```
          case callback.raw_filter
            when Symbol
              self.add_method_tracer callback.raw_filter
            when Proc
              original_filter = callback.raw_filter
              # .pry
              wrapped_filter = Proc.new{
                self.class.trace_execution_scoped(
                  "Custom/#{self.class}\#ProcFilter-#{original_filter.source_location[0].split(
"/").last}:#{original_filter.source_location[1]}" # 補足(1)
                ) do
                  original_filter.call
                end
              }
              callback.instance_variable_set(:@filter, wrapped_filter)
              callback.instance_variable_set(:@key, callback.send(:compute_identifier, callback
.raw_filter))  # 補足(2)
            end
          end
        end

        ...(以下略)...
```

補足 (1)

これは連続する行の実行時間の測定をしています。ここでは、metric 名を自ら命名する必要があります。この実装では、例えば test_controller.rb 内でブロック定義された filter であれば、`Custom/TestController#ProcFilter-test_controller.rb:3` という metric 名になり、filter の定義場所の特定が容易にできます。

補足 (2)

ActiveSupport::Callbacks::Callback のインスタンス変数@filter を書き換えたので、同時に@key も書き換えました。ソースコードの内部実装を引用して、この実装の意図を説明します。

https://github.com/rails/rails/blob/6-0-stable/activesupport/lib/active_support/callbacks.rb#L302

```
        def initialize(name, filter, kind, options, chain_config)
          ...
          @key      = compute_identifier filter
          ...
        end
```

となっており、なおかつ、

https://github.com/rails/rails/blob/6-0-stable/activesupport/lib/active_support/callbacks.rb#L367-L374

```
def compute_identifier(filter)
  case filter
  when ::Proc
    filter.object_id
  else
    filter
  end
end
```

となっています。要は、@raw_filter が Proc の場合は、その Proc の object_id を Callback の @key として設定していましたので、これに習いました。

それでは、以下の手順で動作検証してみましょう。

1. $export FILTER_TRACED_CONTROLLERS=TestController で環境変数を設定
2. $bundle exec rails s
3. ブラウザで localhost:3000/test/test_action

すると、3で以下のようなエラーが発生しました。

```
NameError: undefined local variable or method ‘request’␣for␣ActionController::Base:Class
Did␣you␣mean?␣␣require
from␣/Users/user1/.rbenv/gems/2.7.0/gems/actiontext-6.0.3.4/lib/action_text/engine.rb:54:in␣‘
block␣(3␣levels)␣in␣<class:Engine>’
```

素直に、エラーを発生している action_text/engine.rb の 54 行目付近を見に行きましょう。

https://github.com/rails/rails/blob/6-0-stable/actiontext/lib/action_text/engine.rb#L54

```
    before_action { ActionText::Content.renderer = ApplicationController.renderer.new(request.
env) }
```

before_action に対してブロックを引数で渡しています。こうして before_action filter を定義したは良いものの、いざこれを実行しようすると、このブロック内の request という変数が、実行時の context で見つからなかったということですね。Rails のコントローラでは、request をローカル変数ではなくインスタンス変数として保持しておりますので、request という変数はこの Proc の中に保持されていないことが原因です。

では、そもそも Rails のソースコードではこの問題をどのように解決していたのでしょうか？ (今回のように Proc を Proc で包むという処理を挟まなくたって、request がこの Proc の実行元から参照できないのは一緒じゃないか！ ということです)

ActiveSupport::Callbacks モジュール内で run_callbacks メソッド内に実行の処理が書かれているのではないかと目星をつけて見に行くとまさにそれに関するコメントが書かれていました。

https://github.com/rails/rails/blob/6-0-stable/activesupport/lib/active_support/callbacks.rb#L646-L648

```
# If a proc, lambda, or block is given, its body is evaluated in the context
# of the current object. It can also optionally accept the current object as
# an argument.
```

というわけで、今回はこの proc を TestController の context で実行したいので、#instance_exec メソッドを使用することにします。このメソッドは、公式ドキュメント[*3]では以下のように書かれているため、まさに今回の用途に適しています。

> 与えられたブロックをレシーバのコンテキストで実行します。ブロック実行中は、 self がレシーバのコンテキストになるのでレシーバの持つインスタンス変数にアクセスすることができます。

[*3] https://docs.ruby-lang.org/ja/latest/method/BasicObject/i/instance_exec.html

これを使って、original_filter.call を instance_exec(&original_filter) に書き換えました。

```
          case callback.raw_filter
            when Symbol
              self.add_method_tracer callback.raw_filter
            when Proc
              original_filter = callback.raw_filter
              # .pry
              wrapped_filter = Proc.new{
                self.class.trace_execution_scoped(
                  "Custom/#{self.class}\#ProcFilter-#{original_filter.source_location[0].split(
"/").last}:#{original_filter.source_location[1]}"
                ) do
                  instance_exec(&original_filter)     ### 書き換えた
                end
              }
              callback.instance_variable_set(:@filter, wrapped_filter)
              callback.instance_variable_set(:@key, callback.send(:compute_identifier, callback
.raw_filter))
            end
          end
        end
```

これにて、完成、と言いたいところでしたが、実はまだ問題が残っています。

15.5.2 Proc Callback の Wrapping が何重にも行われるのを避けたい

環境変数 FILTER_TRACED_CONTROLLERS に複数のコントローラを設定するケースを考えてみてください。それらのコントローラで、同じ箇所で定義された filter が指定されている場合、実はそれらは共通の Callback オブジェクトを共有しているのです。

以下で例を示します。ChildController は ParentController を継承しており、ParentController では、ブロック指定の before_action filter が定義されています。

/app/controllers/parent_controller.rb

```
class  ParentController < ApplicationController
    before_action do
      puts  'my␣before␣callback'
    end

    binding.pry
end
```

/app/controllers/child_controller.rb

```
class  ChildController < ParentController
    def  child_action
      puts  'request␣dispatched!'
    end

    binding.pry
end
```

これで$bundle exec rails s してそれぞれの binding.pry の箇所でコントローラが保持している Callback の配列を覗いてみましょう。

```
  From: /Users/user1/repos2/wantedly/app/controllers/parent_controller.rb:11 :
```

```
       6:     def  parent_action
       7:       puts  'request␣dispatched!'
       8:     end
       9:
      10:         binding.pry
   => 11: end

  [1] pry(ParentController)> self._process_action_callbacks().send(:chain).map(&:raw_filter)
  => [#<Proc:0x00007f851a1fc600 /Users/user1/.rbenv/gems/2.7.0/gems/config-2.2.3/lib/config/
integrations/rails/railtie.rb:28>,
   #<Proc:0x00007f851a2eabe8 /Users/user1/.rbenv/gems/2.7.0/gems/actiontext-6.0.3.4/lib/
action_text/engine.rb:54>,
   #<Proc:0x00007f8515a16b88 /Users/user1/repos2/wantedly/app/controllers/parent_controller.rb
:2>,
   ...(以下略)...]
```

```
  From: /Users/user1/repos2/wantedly/app/controllers/child_controller.rb:7 :

      2:     def  child_action
      3:       puts  'request␣dispatched!'
      4:     end
      5:
      6:     binding.pry
   => 7: end

  [1] pry(ChildController)> self._process_action_callbacks().send(:chain).map(&:raw_filter)
  => [#<Proc:0x00007f851a1fc600 /Users/user1/.rbenv/gems/2.7.0/gems/config-2.2.3/lib/config/
integrations/rails/railtie.rb:28>,
   #<Proc:0x00007f851a2eabe8 /Users/user1/.rbenv/gems/2.7.0/gems/actiontext-6.0.3.4/lib/
action_text/engine.rb:54>,
   #<Proc:0x00007f8515a16b88 /Users/user1/repos2/wantedly/app/controllers/parent_controller.rb
:2>
   ...(以下略)...]
```

ParentController と ChildController の CallbackChain には、同一のメモリアドレスを持つ Proc `#<Proc:0x00007f8515a16b88 /Users/user1/repos2/wantedly/app/controllers/parent_controller.rb:2>` が含まれていることが確認できます。

この結果、例えば `export FILTER_TRACED_CONTROLLERS="ParentController ChildController"` によって、ParentController と ChildController の両方を計測対象として指定してしまうと、Proc を Proc で包む作業が二重で行われることになります。この結果、filter の実行された回数 [Avg calls (per txn)] が 2 倍でカウントされてしまいます。(実行時間の計測に関しては問題ないはずです。)

これを阻止するために、Proc を Proc で包んだ際には、その Proc が**もうこの `Proc` は既に `wrap` 済みだ**ということを示すフラグを持たせましょう。この方法の一つとして、Proc を wrapping し終わった後に、その Proc に対してメタプログラミングを駆使して、インスタンス変数`@_already_wrapped` を新しく定義し、これに true を放り込む、という方法を考案しました。以下が実装のイメージになります。

```
            case callback.raw_filter
            when Proc
              unless callback.instance_variable_get(:@_already_wrapped)  # 補足(1-1)
                callback.instance_variable_set(:@_already_wrapped, true)
                original_filter = callback.raw_filter
                wrapped_filter = Proc.new{
                  self.class.trace_execution_scoped("Custom/ProcFilter/...") do
```

```
                    self.instance_exec(&original_filter)
                  end
                }
                callback.instance_variable_set(:@filter, wrapped_filter)    # 補足(1-2)
                callback.instance_variable_set(:@key, callback.send(:compute_identifier,
callback.raw_filter))
              end
            end
```

補足 (1)

Proc の wrapping の重複を防ぐために、既に wrap された Proc に対しては (2) のように`@_already_wrapped` というインスタンス変数を生やしています。

これで良さそう、と思いきや、最後にもう一つだけ修正しなければいけない箇所があるのです！

15.5.3 環境変数で指定されていないコントローラの発動時に、wrapping された Proc 内で実行時間計測を行わないようにする

先程の ParentController と ChildController の例で最後の問題点を説明します。これらは、共通の Callback で保持される filter を設定しているということを念頭に置いてください。

環境変数に `export FILTER_TRACED_CONTROLLERS="ParentController` を指定したと仮定して、ChildController においてあるリクエストが処理されるケースを考えてみます。この時、ChildController の filter の一つは、ParentController と共通の (Proc で定義される)Callback の filter を保持していて、ParentController は計測対象のコントローラに設定されているのですから、その Callback は、既に計測のための wrapping がなされているはずです。結果として、計測対象ではない ChildController がリクエストを処理した場合でも、filter の実行された回数 [Avg calls (per txn)] がカウントされてしまうということになります。(時間計測に関しては特に問題ないはずです。)

これに対処するために、Proc を Proc で wrapping する際に、以下のような条件分岐を加えるようにしました。

```
                wrapped_filter = Proc.new{
                  if traced_controllers.include? self.class  # 補足(1)
                    self.class.trace_execution_scoped("Custom/ProcFilter/...") do
                      self.instance_exec(&original_filter)
                    end
                  else
                    self.instance_exec(&original_filter)
                  end
                }
```

補足 (1)

traced_controllers はローカル変数で定義されているので、Proc に閉包されるます。このため、先程の request のようにな NameError は発生しません。

これにて完成となります！ 最後に、スクリプトと、使用方法をまとめます。

15.6 完成したツールの使用法

- 以下の filter_traced_controllers.rb を/config/initializers/配下に置く。
- filter の実行時間の計測をしたいエンドポイントに対応するコントローラを調べ、それを環境変数 FILTER_TRACED_CONTROLLERS に指定する (複数指定可能)
  - コントローラは名前空間を含めて指定する
  - 複数のコントローラを指定する場合は、半角スペース区切りで指定する (全角スペースはダメです)
  - (例) $export FILTER_TRACED_CONTROLLERS="TestController MyModule::SampleController"
- Rails サーバを (再) 起動する

なお、環境変数で指定されたコントローラそれぞれに関して、(1) 計測対象に加えることが成功した場合は info レベルで、(2) 失敗した場合は error レベルで、Rails ログにその旨が吐き出されます。

filter_traced_controllers.rb

```
# This initializer is to make a configuration for reporting the execution time of filters(
callbacks)
# for all actions in a specified Controller class.
# Controller(s) to be traced can be configurable by environment variable '
FILTER_TRACED_CONTROLLERS'
# Set it this way,,, let's say, 'FILTER_TRACED_CONTROLLERS="TestController MyModule::
SampleController"'

Rails.application.config.after_initialize do
  if ENV['FILTER_TRACED_CONTROLLERS'].present?
    traced_controllers = []
    ENV['FILTER_TRACED_CONTROLLERS'].split.each do |controller_str|
      begin
        traced_controllers << controller_str.constantize
      rescue
        puts "=====␣#{controller_str}␣in␣FILTER_TRACED_CONTROLLERS␣has␣not␣been␣found␣====="
      end
    end

    traced_controllers.each do |controller|
      begin
        controller.class_eval do
          self.include ::NewRelic::Agent::MethodTracer
          self._process_action_callbacks().send(:chain).each do |callback|
            case callback.raw_filter
            when Symbol
              self.add_method_tracer callback.raw_filter
            when Proc
              unless callback.instance_variable_get(:@_already_wrapped)
                callback.instance_variable_set(:@_already_wrapped, true)
                original_filter = callback.raw_filter
                wrapped_filter = Proc.new{
                  if traced_controllers.include? self.class
                    self.class.trace_execution_scoped("Custom/#{self.class}\#ProcFilter-#{
original_filter.source_location[0].split("/").last}:#{original_filter.source_location[1]}") do
                      self.instance_exec(&original_filter)
                    end
                  else
                    self.instance_exec(&original_filter)
```

```
                end
              }
              callback.instance_variable_set(:@filter, wrapped_filter)
              callback.instance_variable_set(:@key, callback.send(:compute_identifier,
callback.raw_filter))
            end
          end
        end
      end
      puts "=====␣Filters␣of␣actions␣for␣#{controller}␣will␣be␣reported␣to␣New␣Relic␣====="
    rescue
      puts "=====␣#{controller}␣in␣FILTER_TRACED_CONTROLLERS␣has␣not␣been␣found␣====="
    end
  end
end
```

15.6.1　動作検証してみる

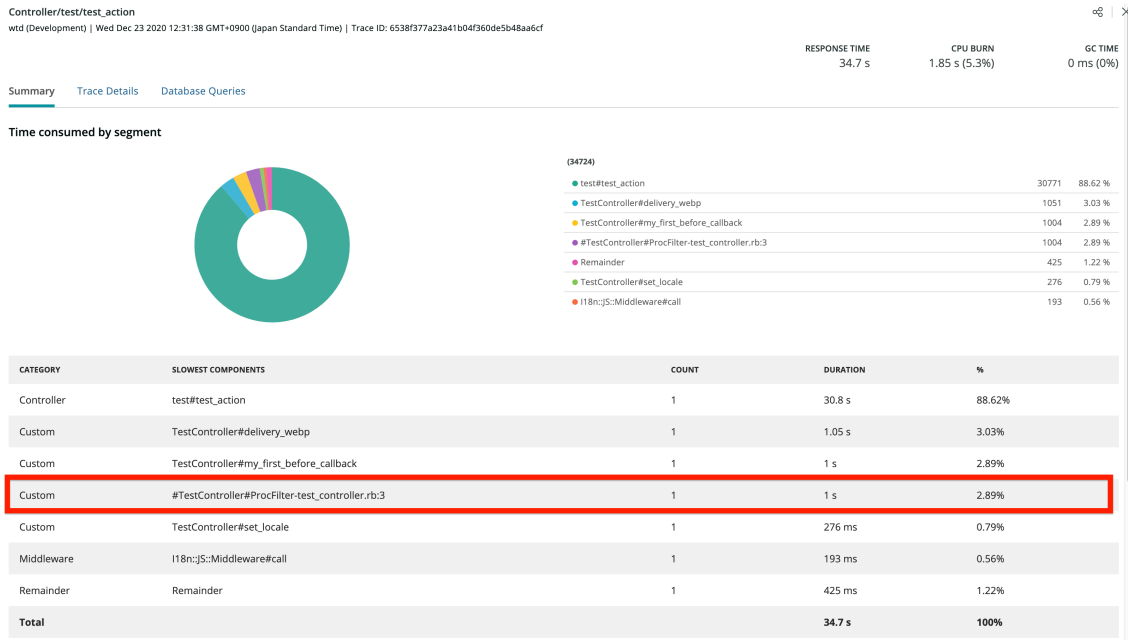

図 15.2　New Relic One 上で計測ができていることの確認

確かに、Custom カテゴリー配下に`#TestController#ProcFilter-test_controller.rb:3` という行が入っているのが確認できます。

15.7　後書き

この章では、

- Callback のソースコードを眺めたり
- メタプロ三昧したり
- Proc の閉包する変数を実行時に参照できなくて怒られたり

本当に長い道のりでした。でも面白かったですね。

余談ですが、僕は今回のように Rails のソースコードをガッツリ追っかけながら読んでいくのが結構好きで、3 ヶ月に 1 回くらい、Rails の内部実装を心のままに読み味わってたまんない気持ちになってます。

余談ですが、ActiveRecord では、before_save や after_save なんてものが登場しますが、これらにも `ActiveSupport::Callbacks::Callback` がガッツリ絡んでいます。

ActiveSupport の Rails Guides[*4]には以下のように記されています。

> Active Support is a collection of utility classes and standard library extensions that were found useful for the Rails framework

この言葉の通り、本当に Rails の色々なところで使われていることがわかります。

といったところで、それではご機嫌よう。バイバイ！

[*4] https://guides.rubyonrails.org/active_support_instrumentation.html