☑ 家事の効率を上げよう

8章 パフォーマンス

古い中国のことわざに「象は見る人の数だけ見え方も異なる」というものがあるそうです。Railsについてもまったく同様のことが言えます。開発者にとって、Railsは微笑みや陶酔感を与えてくれる存在です。しかし、いざアプリケーションをデプロイしスケーラビリティを向上させようという段階になると、大量のユーザを次々にさばかなければならないシステム管理者は金切り声を上げて部屋から飛び出したくなるほどの苦痛を受けることになるかもしれません。このように、Railsには明らかなトレードオフの関係が存在します。Rubyという高級言語によって美しいドメイン固有言語を定義し、Active Recordや完璧な動的プログラミングを実現できますが、処理にはそれなりの時間がかかるものです。水面下で行われ、アプリケーション開発者を安楽へと導く魔法のような機能も、デプロイ時にはその処理コストが目立ってしまいます。

ただ、Railsアプリケーションをスケーラブルなものにする方法は確かに存在します。shared-nothing（何も共有しない）という方針に基づいたRailsフレームワークは、ハードウェアの追加投入によって問題解決への道を開いてくれます（「6章　スケールアウト」参照）。開発者はアプリケーションの処理性能を向上させようという努力を後回しにしがちです。中途半端な最適化は望ましくないこともありますが、処理性能やスケーラビリティの向上を考慮しなくてよいということは決してありません。この章では、デプロイされたアプリケーションがどの程度のトラフィックを処理できるかを理解するための手法をいくつか紹介し、少しの努力で処理性能を劇的に向上させるテクニックの基礎についても解説します。

8.1 背景

ベンチマーキングを行って処理性能を測定するという作業は、系統立ったアプローチを忍耐強く続ける開発者と、苦痛をいとわない開発者にしか向いていないのではないかと思えてしまいます。自分から進んで苦痛を受けたいと願う開発者は少ないはずなので、アプリケーションをデプロイする際には可能な限り慎重に計画を立てて実行するようにしましょう。そのためのステップを図解したのが**図 8-1** です。

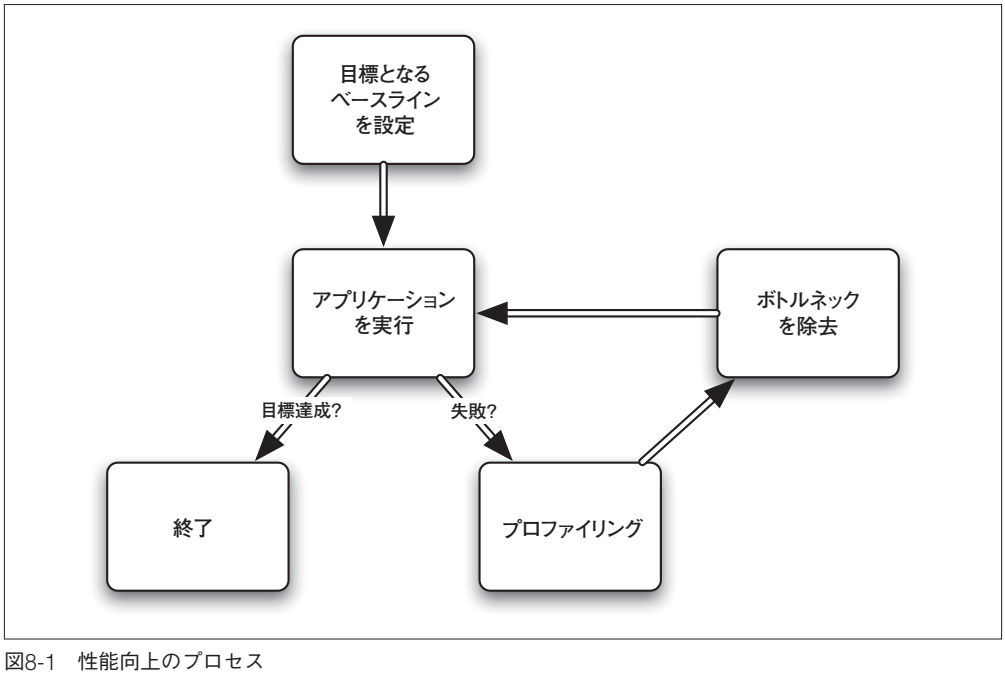

図8-1 性能向上のプロセス

誤ったアプローチ

アプリケーションの性能を向上させるためのプランについて紹介する前に、期待どおりの効果を得られないことが分かっているアプローチをいくつか明らかにしておきたいと思います。

初期段階での最適化

実際に高い処理性能が必要なのかどうかにかかわらず、処理性能を最も重視して開発を行えば、高速でスムーズなアプリケーションができあがるのはある意味当然です。しかし、最適化によってコードは複雑化して読みにくくなり、メンテナンスが困難になるという代償が伴います。最適化の作業にはある程度の時間が必要であり、そのためにプロジェクトの進捗に影響が生じるというリスクも考慮しなければなりません。そのため、処理性能のテストはプロジェクトの中で継続的に行う必要があります。求められている処理性能の水準が満たされているかどうか随時チェックし、重大なボトルネックが発見された場合のみ必要に応じて最適化を行うようにしましょう。

憶測に基づく最適化

アプリケーションが期待に反して低速な場合、問題の所在を推測してみたくなることでしょう。そして問題を解決しようとして、さまざまな最適化のテクニックを試してみたくなることもあるかと思います。しかし、推測は当たることもありますが、本当は解決する必要のない問題の解決に数時間を費やすことになってしまう場合もあります。プロファイ

リングツールや負荷テスト、モニタリングツールなどを使い、何が問題なのかを特定してから解決策を適用しましょう。

キャッシュの乱用

Rails ではキャッシュをとても簡単に利用できるため、キャッシュが最後の砦として安易にとらえられがちです。しかし、キャッシュを利用するというのは見かけよりも難しいことがほとんどです。キャッシュの適用によって魔法のように性能が改善したとしても、テスト時には見られなかった新たなバグが実運用環境に入り込む可能性が生まれてしまいます。キャッシュを伴う機能のテストは困難であるためです。特に、プロジェクトの最終段階で安易にキャッシュを有効化し、性能の向上を願うというのはとても危険です。キャッシュに頼らず、早期にボトルネックとなる可能性がある個所（例えばホームページや最新ニュースのフィードなど）を把握するよう心がけましょう。そして、アプリケーションの機能はキャッシュの利用を前提として設計し、ステージング環境からキャッシュを有効化しておくことによって実運用環境と同等のテストを行えるようにしましょう。

フレームワークの改変

Rails は慣習を重視するフレームワークであり、アプリケーション開発の際にはさまざまな事柄が前提とされています。この理念は、フレームワークの範疇で開発を行う場合にはとてもよく機能します。一方、時には過剰な創造性を発揮し、既存の慣習を破って Rails の本体に手を入れたり、前提条件を回避したりする衝動に駆られることがあるかもしれません。このような場合、Rails フレームワークの開発者が想定していなかった方法で Rails が使われることになります。Rails を新しいバージョンへと更新したり、Rails が正しい方法で使われていることを前提としたプラグインを利用したりしている場合、重大な問題が発生してしまいます。このようなことはせずに、自分が選んだフレームワークの制約の中で最善を尽くすようにしましょう。不明な点があれば Rails コミュニティに質問し、エキスパートたちが自分と同じような問題をどのように解決したのかたずねるべきです。明文化されているか否かにかかわらず、ルールを破るのはよくありません。

1. **最善のシナリオの設定（目標ベースライン）**

   まずしなければならないのは、目標の設定です。これから紹介する手順に従えば、必ず具体的な目標値を得ることができます。ここで定義された最善のベースラインが、処理性能の上限となります。目標を設定するためにはいったん Rails から離れて、実運用環境のサーバがどれだけ多くの HTTP リクエストを処理できるかをまず知る必要があります。実際のアプリケーションの処理性能がここでの値を超えることはありません。次に、Rails アプリケーションに対して可能な限りシンプルなリクエストを送信して測定を行い、Rails アプリケーションとしての性能の上限値を決定します。この時点で 2 つの値が大きくかけ離れてしまっている場合は、プロキシや FastCGI あるいは Mongrel へのチューニングが必要かもしれません。そしてこの値が目標のベースラインになります。

2. **現状の処理性能を知る（アプリケーションのベースライン）**
   アプリケーションの処理性能を向上させたいなら、現状把握は必須です。これによって目標との差も明らかになります。まずは何も最適化を行わずに、簡単な性能測定を行いましょう。これがアプリケーションにとってのベースラインになります。
3. **プロファイリングによるボトルネックの把握**
   ベースラインを定義できたら、システムに対してプロファイリングを行い、ボトルネックを特定しましょう。ボトルネックが存在すると、システム内の他の要素に関係なく、アプリケーションが処理を行える速度が制限されてしまいます。これはあたかもエンジンに対する変速機のようです。体系的にボトルネックを除去することによって、処理性能は向上してゆきます。
4. **ボトルネックの除去**
   処理性能を改善するための最適化はここで行われます。プロファイリングやベンチマーキングを行った時点で、どのような修正を行えば最大の効果を得られるか理解できているはずです。問題点を1つだけ選び、それに集中して修正を行います。これによって、再びプロファイリングと改善のプロセスを開始できるようになります。
5. **繰り返し**
   プロファイリングとベンチマーキングを行い、その結果を元に少しずつコードに変更を加えてゆくという基本的なアプローチの繰り返しによって、どの変更が処理性能に対して好影響あるいは悪影響を与えたかを正確に把握できます。修正は1か所ずつ行い、改善のプロセスをシンプルなものにしましょう。小さな修正を行うたびに測定を行うというアプローチは、長期的には作業の手間を軽減してくれます。

ベンチマーキングの背景となる事柄について理解できたところで、ここからは性能向上のプロセスを構成する要素のいくつかについて解説し、よくあるボトルネックの例とその解決策も紹介することにします。まずはベンチマーキングを通じてベースラインを定義してみましょう。

8.2 初回のベンチマーキング、Mongrelインスタンスの個数に関する検討

アプリケーションに対して初めて行うベンチマーキングによって、アプリケーション全体としての処理性能が明らかになります。ここではHTTPベースの負荷テストツールを使い、ネットワーク経由でテストが行われます。テストの結果に満足できるなら、以降の最適化の作業は不要です。しかし多くの場合、数か所あるいはそれ以上の問題点が発見されるはずです。

ここで、何度も繰り返されてきた「アプリケーションには何個のMongrelインスタンスが必要か」という問いに答える必要があります。筆者の経験では、最高の処理性能を発揮するためのインスタンスの個数は過大に見積もられることが多いようです。自分のWeb 2.0サイトが脚光を浴びてGoogleあたりに買収されるようになりたいと願うのもかまいませんが、現実ではほとんどのRailsアプリケーションが集めるアクセス数は1日あたり10万ページビュー以下で、この程度ならプロ

キシの背後に2つか3つのMongrelインスタンスを配置するだけでも問題なく処理しきれます。このことを確認するための作業がベンチマーキングです。

筆者のテスト環境ではMongrelを使っていますが、別のサーバを使っている場合でも、そのサーバが複数のプロセスを使ってRailsアプリケーションを実行しているなら同じ問題が当てはまります。この問題を解くには、現在用意されているハードウェア環境で考えられる処理性能の上限（ベースライン）を知る必要があります。ab（Apache Bench）あるいはhttperfを使い、Railsを呼び出す最もシンプルなリクエストを送信する場合の1秒当たりの処理件数を調べます。Railsアプリケーションを新規に生成し、コントローラを1つ用意してその中で“Hello!”と出力するだけのアクションを定義します。このアプリケーションに対して測定を行えば、同じハードウェア上でRailsアプリケーションを実行する場合の最速となるベースラインを知ることができます。まず、次のコマンドを実行してください。

```
ezra$ rails benchmark_app
ezra$ cd benchmark_app
ezra$ script/generate controller Bench hello
```

生成されたクラスBenchController（RAILS_ROOT/app/controllers/bench_controller.rbに記述されています）をエディタで開き、次のように記述します。

```
class BenchController < ApplicationController
  def hello
    render :text => "Hello!"
  end
end
```

このアクションhelloはとても小さく、Railsスタックに含まれる機能（セッション管理など）を呼び出した上での最速のリクエストです。ここではデータベースへの接続やテンプレートの挿入などは行われていません。リクエストを受け取ると、アプリケーションはルーティングを行って呼び出し先のコントローラをインスタンス化します。その際に、セッションのためのフィルタやその他のさまざまな処理も実行されています。コードやテンプレート、データベースの呼び出しなどが追加されてゆくにつれて、このアクションの処理速度は低下してゆくため、現時点のコードの速度が上限値になります。

abコマンドのヘルプを見てみましょう。

```
ezra$ ab -h
Usage: ab [options] [http://]hostname[:port]/path
Options are:
    -n requests     Number of requests to perform(リクエストの回数)
    -c concurrency  Number of multiple requests to make(同時リクエストの個数)
    -t timelimit    Seconds to max. wait for responses
                    (レスポンスを待機する時間。単位は秒)
    -p postfile     File containing data to POST
```

```
                (POST形式で送信するデータが記述されたファイル)
-T content-type Content-type header for POSTing
                (POST形式で送信する際のContent-typeヘッダ)
-v verbosity    How much troubleshooting info to print
                (トラブルシューティング用に出力される情報の詳細度)
-w              Print out results in HTML tables
                (結果をHTML形式の表として出力)
-i              Use HEAD instead of GET
                (GET形式ではなくHEAD形式のリクエストを実行)
-x attributes   String to insert as table attributes
                (table要素の属性として出力される文字列)
-y attributes   String to insert as tr attributes
                (tr要素の属性として出力される文字列)
-z attributes   String to insert as td or th attributes
                (td要素やth要素の属性として出力される文字列)
-C attribute    Add cookie, eg. 'Apache=1234' (repeatable)
                (送信されるCookie。Apache=1234など。繰り返し指定可)
-H attribute    Add Arbitrary header line, eg. 'Accept-Encoding: zop'
                Inserted after all normal header lines. (repeatable)
                (送信されるヘッダ。Accept-Encoding: zopなど。
                ヘッダの末尾に挿入される。繰り返し指定可)
-A attribute    Add Basic WWW Authentication, the attributes
                are a colon separated username and password.
                (Basic認証を行う。ユーザ名とパスワードをコロンで区切って指定)
-P attribute    Add Basic Proxy Authentication, the attributes
                are a colon separated username and password.
                (プロキシに対してBasic認証を行う。
                ユーザ名とパスワードをコロンで区切って指定)
-X proxy:port   Proxyserver and port number to use
                (プロキシのサーバ名とポート番号)
-V              Print version number and exit
                (バージョン番号を出力して終了)
-k              KeepAlive feature(HTTPのKeep-Alive機能を使用)
-d              Do not show percentiles served table.
                (処理時間ごとのリクエストの分布を表示しない)
-S              Do not show confidence estimators and warnings.
                (信頼度の目安や警告を出力しない)
-g filename     Output collected data to gnuplot format file.
                (結果をgnuplot形式のファイルとして出力)
-e filename     Output CSV file with percentages served
                (処理されたリクエストの割合をCSVファイルとして出力)
-h              Display usage information (this message)(使用法を出力)
```

ここでは、1 人のユーザが 1,000 回のリクエストを送信した場合について測定してみます。

```
ezra$ ab -n 1000 http://localhost:3000/bench/hello
This is ApacheBench, Version 1.3d <$Revision: 1.73 $> apache-1.3
Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Copyright (c) 1998-2002 The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 100 requests
Completed 200 requests
   :       :     :
Completed 900 requests
Finished 1000 requests
Server Software:        Mongrel
Server Hostname:        localhost
Server Port:            3000

Document Path:          /bench/hello
Document Length:        5 bytes

Concurrency Level:      1
Time taken for tests:   9.196 seconds
Complete requests:      1000
Failed requests:        0
Broken pipe errors:     0
Total transferred:      255000 bytes
HTML transferred:       5000 bytes
Requests per second:    108.74 [#/sec] (mean)
Time per request:       9.20 [ms] (mean)
Time per request:       9.20 [ms] (mean, across all concurrent requests)
Transfer rate:          27.73 [KBytes/sec] received

Connnection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0     0    0.0      0     0
Processing:     6     9    6.5      8    83
Waiting:        6     9    6.5      8    83
Total:          6     9    6.5      8    83

Percentage of the requests served within a certain time (ms)
50%        8
66%        8
75%        8
80%        9
90%        9
95%        9
98%       10
99%       32
100%      83 (last request)
```

ここで最も注目すべきなのが、「1秒あたりのリクエスト」の「108.74」という値です。この測定は2ギガヘルツのIntel Core Duoと2ギガバイトのメインメモリを搭載したマシン上で行われました。なおここで使っているような、空のRailsアプリケーションは1つのMongrelインスタンス当たり約45メガバイトのメモリを消費します。繰り返しますが、この測定結果は理論上の上限値を表しており、現実のアプリケーションに即しているわけではありません。

近年のハードウェアでは、単純なアクションなら1秒当たり100リクエスト以上を容易に達成できます。実際の性能は環境ごとに異なりますが、この程度の性能を出せるなら十分でしょう。セッション管理を無効化すれば、さらにベースラインを高い値にできます。

セッションを無効にするだけで、1秒間に400リクエスト以上処理できるようになってしまうこともあります。つまり、セッション管理が必要ない場合は`ActionController::Base`の`session`メソッドを使い、必ずセッションを無効化すべきです。しかし実際のアプリケーションではセッション管理が必要なことがほとんどなので、ここでの108という値が上限値であると考えてよいでしょう。

次にMongrelをクラスタ化し、ロードバランサとなる何らかのプロキシの背後に配置します。この構成では複数ユーザによる同時接続を模したテストが可能であり、現時点でのハードウェア上で実行するべきMongrelインスタンスの個数を知ることができます。プロキシのインストールや設定については「6章　スケールアウト」で解説しています。

プロキシの設定が終わったら、まずMongrelインスタンスを2つ起動してテストを行い、その後インスタンスを1つずつ増やしてテストを再実行してゆきます。それぞれのテストの中で、`-c`オプションを使って同時接続のユーザ数を変更できます。筆者は10人、30人そして50人でテストを行うことにしています。これ以上を値を増やすと、`ab`の不具合のため測定結果に誤差が多く含まれるようになってしまいます。ここでは上限値を知ることが目的であり、本格的な負荷テストはアプリケーションが完成して最終的な設定が済んでから行います。実際の利用法に近付けるために、別のサーバからテストを実行したり、複数のサーバから同時にテストを実行したりします。

実際の利用法に近付けるもう1つのテクニックとして、`-C`オプションを使ってCookieの値をセットするというものがあります。先ほどの測定では、1,000回のリクエストすべてに対して新たにセッションが生成されてしまっています。実際には、一度ログインしたユーザは同じセッションを繰り返し使うため、これほど多数のセッションが生成されることはありません。また、サイトの処理性能をテストする際には保護されたページへのアクセスも行うべきであり、このためにもセッションを維持したベンチマーキングが不可欠です。ログイン済みのユーザによるアクセスを再現するために、まずWebブラウザを使ってサイトにログインし、その際Cookieに記録されている`_session_id`の値を調べます。ブラウザの設定画面を探せば、Cookieの内容を参照する何らかの手段が用意されているはずです。

`_session_id`の値が分かったら、それを`ab`コマンドの中で指定します。

例えば次のように一連のコマンドを実行し、アプリケーションに対してベンチマーキングを行います。

```
ezra$ ab -n 5000 -c 10 -C _session_id=ee738c2fcc9e5ab953c35cc13f4fa82d \
        http://localhost:3000/bench/hello
...
ezra$ ab -n 5000 -c 30 -C _session_id=ee738c2fcc9e5ab953c35cc13f4fa82d \
        http://localhost:3000/bench/hello
...
ezra$ ab -n 5000 -c 50 -C _session_id=ee738c2fcc9e5ab953c35cc13f4fa82d \
        http://localhost:3000/bench/hello
...
```

1秒当たりのリクエスト数の増加が止まるまで、1つずつMongrelインスタンスを追加してゆきましょう。これはApache 2.2やnginxなどのソフトウェアを使って負荷分散を行っている場合に特に重要です。なお、Mongrelインスタンスの数を2倍にすれば処理能力も2倍になると考えられがちですが、これは多くの場合誤りです。必要以上にインスタンスを追加すると、ロードバランサの処理が増加してしまうため、かえってレスポンスが悪化してしまうことになります。ハードウェアベースのロードバランサを使えば、インスタンスを倍増させることによって性能もほぼ2倍になるでしょう。しかしこのような装置は、ほとんどのプロジェクトにとって予算オーバーしてしまうほど高価です。

このようにして系統立ったテストを行うことにより、アプリケーション全体の処理性能を予測できるようになります。上限値となるベースラインがないと、どの程度までコードの最適化を続け、どの程度ハードウェアを追加するべきなのか把握することは困難です。Railsアプリケーションは処理の内容もリソースの使い方もさまざまであるため、処理性能の向上につながる一般的なアドバイスを提供するのは困難です。系統だったアプローチに基づき、どのような変更についてもその前後で測定を行うことにより、何が改善に寄与し、何が逆効果をもたらしたか分かるようになります。

8.3 プロファイリングとボトルネック

処理性能の上限値が分かったら、これに向けてアプリケーションを改善してゆきましょう。処理性能の改善には、計画的なアプローチが必要です。筆者は通常、プロファイリングのためのソフトウェア（プロファイラ）を使って処理に時間のかかるコードすなわちボトルネックを検出することにしています。アプリケーションの中でどの部分の処理にどの程度の時間がかかっているのかを正確に理解できたら、自分で修正できるコードのうち最も低速な部分の改善に取り組みます。そして再びベンチマーキングを行い、同じ手順を反復します。

筆者は2種類のプロファイラを利用しています。1つめのプロファイラは、コントローラからデータベースに至るまでのリクエスト全体の処理性能を調べてくれます。もう1つではより詳細なプロファイリングが可能で、Active Recordのモデルに対する測定に特化しています。

8.3.1 ruby-prof

`ruby-prof`はGemとして提供されており、Rubyのコードに対してかなり詳細にプロファイリングを行えます。Railsアプリケーションや Railsのスタック全体へのプロファイリングを行うため

の便利なプラグインも付属しています。インストールは gem install ruby-prof というコマンドを実行するだけで行えます†。必要なファイルは Ruby のインストールディレクトリの下に置かれます。

筆者の環境では **<Ruby インストールディレクトリ>**/gems/1.8/gems/ruby-prof-0.5.2/rails_plugin に置かれていました。このディレクトリの中に ruby-prof というサブディレクトリがあるので、これを RAILS_ROOT/vendor/plugins にコピーしてください。このサブディレクトリは他の Rails アプリケーションでも使われる可能性があるので、移動ではなくコピーするようにしましょう。アプリケーションに対して変更が必要なのは 1 点だけです。

RAILS_ROOT/config/environments ディレクトリに置かれているそれぞれの環境のための設定（development.rb、test.rb、production.rb など）に、config.cache_classes = (true または false) のような行が記述されているはずです。テスト向けと実運用向けの環境では true と指定されており、開発向け環境では false になっています。クラスの不必要な再読み込みによって測定結果が影響を受けるのを防ぐために、この値は true にする必要があります。プロファイリングのためだけに実運用向け環境を使ったり、開発向け環境の設定を変更するよりは、プロファイリング専用の環境を新規作成するのがよいでしょう。

development.rb の内容をコピーして profiling.rb を作成し、config.cache_classes の値を true にしましょう。そして config/database.yml にも設定を追加します。実運用向け環境のデータベースをそのまま使ってもよいし、プロファイリングのためだけに独立したデータベース環境を用意してもかまいません。プロファイリングの作業を進めてゆくにつれて、このファイルに微調整を加えることがあるかもしれません。例えば、筆者はキャッシュを無効化するようにしています。ここで知りたいのは、コードが実行されない場合ではなく実行される場合の性能だからです。もし必要なら、キャッシュを有効化したプロファイリング向け環境をもう 1 つ用意してもよいでしょう。

これでプロファイリングのための準備は整いました。script/server -e profiling を実行してサーバを起動し、ブラウザからアプリケーションのホームページにアクセスしてみましょう。プロファイラによる計測結果は、サーバを起動したコンソールと log/profiling.log（ここでの profiling は環境の名前を表します）の 2 か所に出力されます。出力例を示します。

```
Completed in 0.00400 (249 reqs/sec) | DB: 0.00200 (49%)
        | 302 Found [http://localhost:3000/profiles/login]
 [http://localhost:3000/profiles/login]

Thread ID: 82213740
Total: 0.005

%self     ...  calls  name
 20.00    ...     18  IO#read
 20.00    ...      2  Mysql#read_rows
```

† 訳注：ここでは ruby-prof のバージョン 0.5.2 を使用して説明を行います。

```
 20.00    ...      33  Hash#default
 20.00    ...       2  Kernel#respond_to_without_attributes?
 20.00    ...       1  ActionController::Benchmarking#perform_act...
```

このプロファイリング結果は、ユーザ名とパスワードを元にユーザの登録情報を検索し、ログインを行うという一般的な処理に対して行ったものです。先頭行は Rails によって出力される典型的なログであり、プロファイラがインストールされていなくても出力されます。以降の行は、処理時間の中でコントローラでの HTML の生成やモデルからデータベースへのアクセスなどが占めている割合を表しています。多くの場合、この出力によって必要な情報をすべて知ることができますが、データベースアクセスのせいで HTML の生成にかかる時間が目立たなくなってしまうこともあります。そのような場合にはより詳細なプロファイリングが必要になります。この出力例にはデータベースに対するネットワーク入出力、MySQL から受け取ったデータの解析や各種のライブラリの呼び出し（読者にとって関連の薄いものが含まれていることもあります）などが含まれています。IO#read や Mysql#read_rows にあまりにも多くの時間がかかっているなら、それはデータベースから取得する行または列の数が多すぎるか、列の中に含まれるデータが大きすぎる（例えば長い文字列など）ということを示しています。ここで出力される情報がすべてではありませんが、手始めとしては非常に優れた情報をここで手に入れることができます。

先の出力結果の中に、HTML を生成する処理が含まれていないことに気付かれたかもしれません。これは、アプリケーションが Redirect After Post パターンを利用しているためです。一般的に、データを送信したユーザを GET 形式で提供されるページへとリダイレクトするというのはよいことです。こうすると、ページの再読み込みや戻るボタンのクリックが発生してもユーザの使い勝手が損なわれることはありません。リダイレクトを行うと、HTML の生成は別のリクエストで行われ、プロファイリングの結果も分けて出力されます。

リダイレクト先のページに対するプロファイリングの結果は次のようになりました。

```
Completed in 0.00700 (142 reqs/sec) | Rendering: 0.00300 (42%)
        | DB: 0.00100 (14%) | 200 OK [http://localhost:3000/profiles/view]
 [http://localhost:3000/profiles/view]

Thread ID: 81957820
Total: 0.007

%self     ...  calls  name
 14.29    ...      6  ActionController::Base#response
 14.29    ...     73  Hash#[]
 14.29    ...     17  Array#each
 14.29    ...     81  String#slice!
 14.29    ...      4  Logger#add
 14.29    ...      1  ActionController::Benchmarking#render
 14.29    ...      1  <Class::ActiveRecord::Base>#method_missing
```

コントローラが実行され、その中でさまざまなメソッドが呼び出されているのが分かります。その中にはアプリケーションのコードが直接呼び出しているものもあれば、Rails が内部的に呼び出し

ているものもあります。この例には突出した値はありませんが、もしあればそれが改善の対象になり得ることを表しています。1行1行を詳細に見る必要はなく、他の行と比べて不自然だったり時間がかかりすぎているものを見つけ、その原因を突き止めることに集中しましょう。プロファイリングとは、改善すべき個所を正確に特定し、推測による作業をできる限り減らすためのものです。

ボトルネックが特定できたら、より詳細なプロファイリングを始めましょう。ただし、難しい操作はまったく必要ありません。その機能はruby-profにすでに用意されており、HTMLやグラフィカルな形式で結果を出力することもできます。RAILS_ROOT/vendor/plugins/ruby-prof/lib/profiling.rbに置かれている、プラグインのファイルを開いてみましょう。その中に、次のようなコードがあるはずです。本文中での説明に合わせて、コメント文には番号を付けてあります。

```
# #1 プレインテキストとして出力します
printer = RubyProf::FlatPrinter.new(result)
printer.print(output, {:min_percent => 2,
                       :print_file => false})
logger.info(output.string)

# #2 HTMLとしてグラフィカルに出力します
# printer = RubyProf::GraphHtmlPrinter.new(result)
# path = File.join(LOG_PATH, 'call_graph.html')
# File.open(path, 'w') do |file|
# printer.print(file, {:min_percent => 2,
#                      :print_file => true})
# end

# #3 KCacheGrindを使った視覚化のためのデータを出力します
# printer = RubyProf::CallTreePrinter.new(result)
# path = File.join(LOG_PATH, 'callgrind.out')
# File.open(path, 'w') do |file|
# printer.print(file, {:min_percent => 1,
#                      :print_file => true})
# end
```

デフォルトでは、プレインテキストとして出力するコード(#1)が有効になっています。これによる出力結果は、今までに見てきたとおりです。必要な情報はすべて含まれていますが、非常に読みにくい出力形式です。他の出力形式を利用するには、該当する行をコメント解除します。

特定の処理について詳細にプロファイリングを行うには、HTML形式での出力(#2)が最も便利です。これを使うと、それぞれのリクエストに対してHTMLファイルが生成されます(すでに存在する場合は上書きされます)。実際にはきわめて大量のデータが出力されるのですが、最小限の情報だけを抜き出すと次のようになります。

```
...     ...  ...                                                  ...
        ...  ActionView::Base#compile_and_render_template        325
        ...  ActionController::Base#perform_action_without_filters  1101
```

```
66.67%  ...  Kernel#send                                    0
        ...  ProfilesController#view                        1101
        ...  ActionController::Base#template_class          1164
...     ...  ...                                            ...
```

出力されたデータはいくつかのセクションに分かれており、それぞれの中で1つずつ太字でメソッド名が記述されています。そのメソッドが、それぞれのセクションの中で注目の対象になっていることを表しています。太字のメソッドよりも上に記述されているのは、該当のメソッドを呼び出したメソッドを表します。逆に、下に記述されているのは太字のメソッドが呼び出したメソッドです。太字の行には、そのメソッドの実行に費やされた時間の割合も表示されています。可能な場合は出力結果の中に行番号も含まれており、ローカルのファイルシステム上にあるRubyのソースコードにリンクされています。Railsのコードにリンクしている場合、そのコードはvendorディレクトリにフリーズされている必要があります。出力されるHTMLはとても長いので、ブラウザのページない検索機能を活用してボトルネックを探しましょう。

3つめの出力方法であるKCacheGrindの視覚化用データ形式（#3）では、プロファイリング結果をグラフィカルに表示させるためのデータが出力されます。詳細についてはここでは触れませんが、次に示すサイトに出力サンプルなどの情報が紹介されているのでアクセスしてみてください。1つめのサイトはRubyForgeにあるruby-profのホームページなのですが、Ruby関連の他のドキュメントと同様にとても貧弱です。そこでブログ記事を検索してみたところ、詳細に踏み込んだ優れた記事（2つめのリンク）を発見したのでここで紹介しておきます。

- http://ruby-prof.rubyforge.org/
- http://cfis.savagexi.com/articles/2007/07/10/how-to-profile-your-rails-application

8.3.2 Railsに組み込みのモデル用プロファイラ

処理性能に関する問題はActive Recordのモデル関連であることがほとんどです。例えばモデルの呼び出しが低速すぎたり、ループの中などでファインダを多量に呼び出しているといった問題が考えられます。性能低下の原因がActive Recordのモデルにあると分かったら、Railsに組み込みのプロファイラを利用してみましょう。

このプロファイラはruby-profとほぼ同様に動作しますが、特に何かをインストールする必要はありません。出力されるデータもruby-profと同じくらい大量です。ここでは出力例を紹介するにとどめたいと思います。

```
ezra$ script/performance/profiler 'Profile.find_by_id(1)' 10 graph
Loading Rails...
Using the standard Ruby profiler.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 12.65     0.22      0.22      114     1.90     2.18  Array#select
 10.90     0.40      0.19      498     0.38     0.87  Mysql#get_length
```

```
 9.09     0.56     0.16     1975     0.08     0.11  Kernel.===
 6.29     0.67     0.11      107     1.01     1.30  Mysql::Net#read
 4.60     0.75     0.08       72     1.10    31.40  Integer#times
 4.55     0.83     0.08     3724     0.02     0.03  Fixnum#==
 3.67     0.89     0.06       10     6.30    11.00  ActiveRecord::Base#co...
 3.67     0.95     0.06      109     0.58     0.58  Object#method_added
 3.55     1.01     0.06      137     0.45     5.58  Array#each
 2.74     1.06     0.05       12     3.92     5.17  MonitorMixin.mon_acquire
 2.74     1.11     0.05        8     5.87     5.87  Class#inherited
 2.68     1.15     0.05      963     0.05     0.05  String#slice!
 1.86     1.18     0.03       12     2.67    63.67  Mysql#read_query_result
 1.86     1.22     0.03       56     0.57     0.57  Mysql::Field#initialize
 1.81     1.25     0.03       98     0.32     0.32  Gem::GemPathSearcher#m...
 1.81     1.28     0.03     3445     0.01     0.01  Hash#key?
 1.81     1.31     0.03      736     0.04     0.04  Array#<<
 1.81     1.34     0.03       22     1.41    33.23  Mysql#read_rows
 1.81     1.37     0.03       93     0.33     7.53  Mysql#read_one_row
```

8.4 よくあるボトルネック

ここまではボトルネックを発見する方法について見てきましたが、続いては実際にボトルネックの除去に取り組んでみましょう。多くの専門家がRailsアプリケーションの処理性能に関する問題に取り組んできており、中でもStefan Kaesによる功績は注目に値します。彼はRailsの処理性能に関する権威であり、彼が発見してRailsConf[†]で指摘したボトルネックは各所で引用されています。これらのボトルネックは、発表から数年を経た現在でも依然として重要であり続けています。ここでは次のようなボトルネックが紹介されました。

低速なヘルパメソッド

ビューは大量に繰り返されるループの中でヘルパメソッドを呼び出しているため、ヘルパによる処理が全体の多くを占めるということがしばしばあります。頻繁に利用されるヘルパはシンプルに保ちましょう。

複雑なルーティング

Webサーバが Railsアプリケーションを呼び出すたびに、ルータは`routes.rb`の内容を元にルーティングの処理を行います。ここであまり複雑な処理を行うべきではありません。

アソシエーション

Active Recordを使うと、内部でデータベースの処理を行うコードを簡単に作成できます。しかし簡単に作成できるからといって、アソシエーションの取得が頻繁に発生すると処理性能は低下してしまいます。以降の節で、重大な処理性能の低下につながる問題についていくつか紹介しています。

† http://railsexpress.de/blog/files/slides/railsconf2006.pdf