Ruby on Rails 検証報告書

日本 OSS 推進フォーラム アプリケーション部会
Ruby アプリケーションタスクフォース

2009.5.12

目　　次

図表目次

1. はじめに

1.1 検証背景

近年の企業情報システムは Web アプリケーション技術を用いて構築されることが一般的となっている。Web アプリケーション構築のために数多くの種類の技術が提案・利用されているが、企業情報システムを受託開発するような企業(システム・インテグレータ:SIer)により採用される技術は、規模の大小やシステムの特性にもよるが Java・.NET および PHP を利用した技術が採用されることが多い。それら技術は多くの企業や技術者によって、開発や運用のためのノウハウが長年の経験から蓄積されており、安定した技術として Web アプリケーション開発の主流となっている。反面、Java・.NET および PHP の問題点も明らかにされつつあり、歴史の長さに伴う技術範囲の肥大化・複雑化による技術者の負担や開発コストの増大なども問題視されはじめた。2000 年代前半にはこれまでの延長線上ではない、さまざまな技術的アプローチが提案されつつある状況であった。

そういった状況の中で、生産性の高さやアジャイルな開発に適した Web フレームワークとして Ruby on Rails(以下 Rails)が 2004 年に発表された。その名の通り、Ruby on Rails は Ruby と呼ばれるプログラミング言語で作成されている。

Rails のリリース以来米国を中心に海外では Rails の採用が広まりつつあり、日本国内においても Web による B2C を生業とする企業や、小規模な企業情報システムの開発ツールとして徐々に広まりを見せている。国内での認知度が高まるにつれて、日本各地でコミュニティ主催の勉強会や企業による情報交換会なども頻繁に催されるようになってきており、もうひとつの主流技術として注目度が高まりつつある状況にある。また、Ruby は日本人のまつもとゆきひろ氏により作成された言語であり、日本発の技術が利用されているという点においても国内の注目度が高まる要因の一つとなっている。

しかしながら、Ruby は誕生から約 16 年経過[1]しているとはいえ注目されだしたのは Rails 誕生後のここ数年であるため、企業情報を構築するために必要な基礎的な情報すら各企業や技術者に蓄積されていない。Web や書籍という形でプログラミングテクニックなどの情報は流通し始めてはいるが、運用やノウハウなど実践的な情報や経験は、先駆的な特定の企業や技術者が保有している段階であり、多くの一般的な企業や技術者には広まっていない。多くの情報システムを扱う企業は、基礎的な技術情報から蓄積しなければならない状況である。

1.2 検証目的

企業情報システムを構築するにあたり、次の 2 点について検討するための基礎的な情報を収集することを目的とした。

- Rails は情報システム構築のツールとして利用することが出来るのか
- どのようにすれば有効に活用することができるか

前節でも述べたように、多くの企業において実践的な Rails の技術的な情報やノウハウは蓄積されていない。

[1]Ruby は 1993 年 2 月 24 日に生まれまたとされている

プログラミング上のテクニックや導入手段などは書籍や Web 上の情報を参照することで解決できることが多いが、いざ利用するとなるとこれまでの Web アプリケーション構築の経験から得られている「運用上つまずきそうなポイント」や「他の技術(Java や.NET ベースのフレームワーク)ではありがちな設計の Rails の適応性」に対する不安は払拭しきれない。また、Rails を採用するならば、Rails の最大限に有効活用できるような設計やプロジェクト運営なども検討する必要があり、そういった方法論も同時に考えることができるような基礎的な情報が必要になる。

1.3 Ruby および Ruby on Rails の概要

Ruby とは

Ruby は、まつもと ゆきひろ氏により開発されたオブジェクト指向言語である。スクリプト言語であり、まつもと氏による実装では[2]インタプリタとして実装されている。言語機能としてクラスの定義や継承、クロージャ、Mixin などの機能を備えている。情報システムの構築で用いられる多くの言語である C や C#, Java は静的な型付け言語であるのに対して、Ruby は動的な型付け言語である。

Ruby の公式サイトでは、次のように Ruby を紹介している。

「Ruby とは」http://www.ruby-lang.org/ja/about/

> **Ruby** は、手軽なオブジェクト指向プログラミングを実現するための種々の機能を持つオブジェクト指向スクリプト言語です。本格的なオブジェクト指向言語である **Smalltalk**、**Eiffel** や **C++**などでは大げさに思われるような領域でのオブジェクト指向プログラミングを支援することを目的としています。もちろん通常の手続き型のプログラミングも可能です。
>
> **Ruby** はテキスト処理関係の能力などに優れ、**Perl** と同じくらい強力です。さらにシンプルな文法と、例外処理やイテレータなどの機構によって、より分かりやすいプログラミングが出来ます。
>
> まあ、簡単にいえば **Perl** のような手軽さで「楽しく」オブジェクト指向しようという言語です。どうぞ使ってみてください。
>
> **Ruby** はまつもと ゆきひろが個人で開発しているフリーソフトウェアです。

[2] Ruby 1.8 のリファレンス実装のこと。その他には Java による実装の JRuby や.NET による実装の IronRuby、Ruby 1.9 のリファレンス実装などが存在する。

Rails とは

Rails は、Web アプリケーションのためのフレームワークであり、デンマークの David Heinemeier Hanssons 氏により作成された。Ruby on Rails の名の通り Ruby により作られている。アーキテクチャとして MVC が採用されていいる。その他の同種のフレームワークと比べて少ないコーディング量で開発できることや、アジャイルな開発形態を想定した設計であると謳われている。

設計思想

Rails の特徴として「DRY」「設定より規約」と呼ばれる設計思想が挙げられる。それぞれ、Rails 作者らの著作より引用[3]する。

「DRY(Don't Repeat Yourself):繰り返しを避けよ」

> **DRY** は **Don't Repeat Yourself**(繰り返しを避けよ)の省略です。これは、システムのあらゆるピースは、**1** ヵ所にのみ記述されなければならないということです。**Rails** では **Ruby** のパワーを利用してこれを実践しています。**Rails** アプリケーションでは重複はごくわずかしかありません。記述すべき箇所は **1** ヵ所だけでいいのです。その箇所は、たいてい **MVC** アーキテクチャの規約が提案してくれます。あとはそこから先に進むだけです。簡単なスキーマ変更に、最低でも **5**〜**6** ヵ所のコード変更が必要だった他の **Web** フレームワークプログラマにとっては、まさに天啓でした。

「CoC (Convention over Configuration):設定より規約」

> **Convention over Configuration**(設定より規約)も重要な概念です。これはつまり、あなたが自分のアプリケーションを組み立てていくとき、たいていの場面では **Rails** がよろしくやってくれるという意味です。規約に従っていくだけで、**Rails** アプリケーションを書くことができるのです。典型的な **Java** アプリケーションで **XML** の設定をするコードより少ないコードで済んでしまいます。規約をいじる必要がある場合にも **Rails** で簡単にオーバーライドできます。

[3] Dave Thomas, David Heinemeier Hanssons (2006). Agile Web Development With Rails. Pragmatic Bookshelf.
(デビッド・トーマス, デビッド・ハイネマイヤー・ハンソン, 前田 修吾(訳)
Rails によるアジャイル Web アプリケーション開発 第 2 版(2007) オーム社 P.2)

アーキテクチャ

Rails の Model/View/Controller(MVC)アーキテクチャに基づいてアプリケーションを構築する。

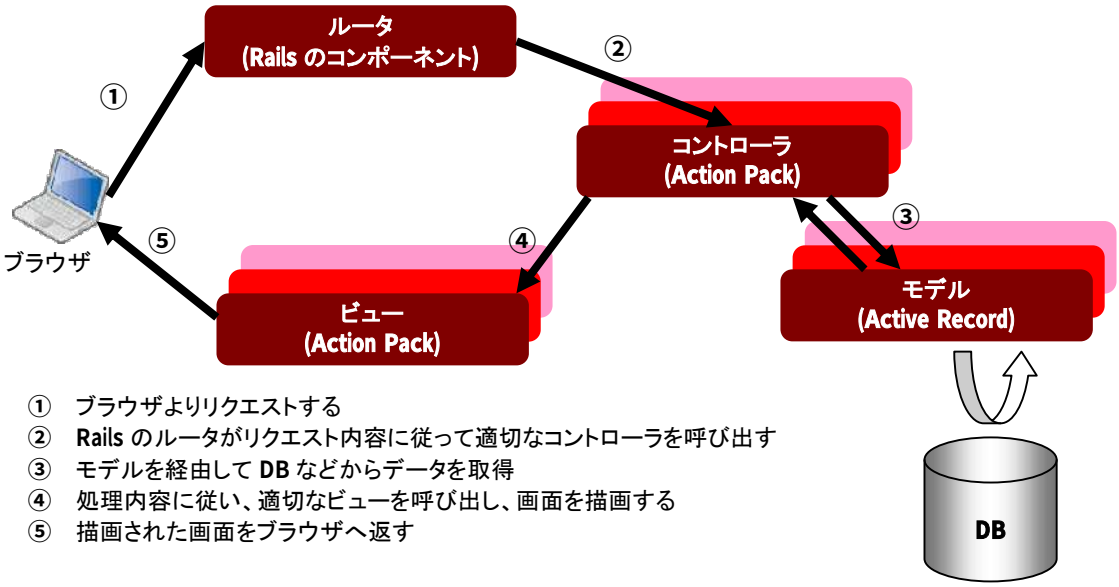


図 1-1 Ruby on Rails のアプリケーション・フレームワーク概要図

コントローラ

クライアント(図 1-1のブラウザ)からのリクエストを解釈し、応答データを作成する。実装に従って、リクエストを解釈し、必要に応じて後述のモデルを利用してデータベースからデータの取得もしくは格納を行う。リクエストに従って、適切なレスポンスするために後述するビューに処理を遷移させる。

ルータにより、ブラウザからのリクエストは、コントローラのメソッド呼び出しのように処理される。アプリケーション開発者はコントローラにメソッドを追記するだけで処理内容をコーディングすることができる。

Rails ではコントローラの役割として Action Pack と呼ばれるコンポーネントを同梱している。

ビュー

ビューではコントローラによって処理された結果をもとに、テンプレートエンジン等で描画処理をおこなう。

基本的には Web ブラウザに対する応答として HTML を描画する。ただし、ビューで生成されるものは HTML とは限らず、RSS リーダに返される XML データや、Ajax のための JavaScript コードなども生成することが出来る。

HTML を描画するために、ビューには erb と呼ばれるテンプレートエンジンが採用されている。erb は Java における JSP、.NET における aspx のように、HTML の間にプログラム処理を埋め込めるような構文形態が採用されている。

Rails ではビューの役割としてコントローラと同じ Action Pack と呼ばれるコンポーネントを同梱している。

モデル

データの取り扱いを担当する。データは Ruby のオブジェクトとして表現される。アプリケーションが利用するデータを、DB などの格納先から取り出し、もしくは、参照するなどの処理を実行する。

多くの Web アプリケーションでは、データの格納にリレーショナル・データベース(RDB)が利用されている。RDB とオブジェクト指向では、データの取り扱いに関する思想に大きな違いがある。思想の違いからデータを

相互に交換するためには多くのソースコードを書く必要があり「面倒」である。これはインピーダンス・ミスマッチと呼ばれる問題であり、このミスマッチを解消するためのコンポーネントは O/R マッパーと呼ばれる。Rails では O/R マッパーとして ActiveRecord と呼ばれるコンポーネントが同梱されており、Rails のモデル層は ActiveRecord を中心に構成されている。ActiveRecord は Ruby のもつ強力な動的言語の特性を生かして作成されており、アプリケーションと RDB のデータのコミュニケーションをコーディングの面から強力にサポートする。

ジェネレータ

プロジェクトを作成、もしくは、これまでに挙げたコントローラやモデル、ビューなどの各部品の「ひな形」となるソースコードを生成する機能である。ジェネレータにより生成されたひな形を元に、開発者はアプリケーションを実装する。

ジェネレータという名前から何かのフォーマットの入力データを元にソースコードを出力するイメージではあるが、Rails におけるジェネレータは「ひな形生成ツール」のような位置づけで用意されている。コマンドライン上から引数によってジェネレータに対して情報が引き渡され、ひな形が作成される。引数以上の表現力が必要な場合は、ジェネレータにより生成されたソースコードや設定ファイルなどに対して開発者が追記することとなる。ジェネレータは開発の初期段階で何かのひな形を生成したりするために利用されるものであり、常にジェネレータを動かし続けて開発をするような用途ではない。

ジェネレータにより生成されるものの例を、表 1-1に示す。ジェネレータにより生成できるものは表 1-1にあげられたもの以外にも存在する。

表 1-1 Ruby on Rails のジェネレータの例

対象	説明
プロジェクト	構築するアプリケーション全体のフォルダ構成や設定ファイル等を生成する。
scaffold	DB に対するデータの CRUD 処理をひな形として生成する。生成されるものはコントローラ・モデル・ビュー・自動テストのひな形である。
コントローラ	画面動作を記述する為のスケルトンファイル（コントローラと、ビュー）を生成する。生成されたファイルに開発者がコードを記述する。
モデル	指定のモデルクラスを定義したファイルを生成する。モデルクラスは生成時より動作できる。
ビュー	指定したビューの為のファイル構成を生成する。画面を追加する際に指定する。
テスト	自動テストのためのひな形を生成する。多くは付随的に生成され、例えばコントローラを生成すると、自動的にコントローラ用のテストも同時に生成される。

2. 検証計画

2.1 検証項目選定方針

Web アプリケーションを構築・運用するにあたり、必要であると思われる技術や経験的に問題となるであろう技術的要素を検証項目として洗い出し、ドキュメントや実機により検証することとした。検証項目を選定するにあたっては、書籍や Web 上で明らかにされているプログラミング上のテクニックなどは最小限にとどめ、Web アプリケーションを作成した場合に問題となるような項目を優先的に選定することとした。

なお、本検証プロジェクトの期間や予算上、あらゆる検証項目に対して調査することは出来ない為、検証項目の重要度によって優先度をつけ検証を実施することとした。

2.2 検証環境

機器構成

機器構成は図 2-1のような構成とした。後述されるように今回の検証ではサーバ OS として Windows を使用した。

情報システムの受託開発現場では Windows サーバも多く見られるためである。また、Linux での動作実績は多く Web 上などで報告されているが、Windows での実績報告はあまり見ることができないためあえて検証環境とすることとした。DBMS は、Windows とセットで採用されることが多い SQL Server 2005 を採用した。但し、DBMS の種類が関係ない検証項目では一部 MySQL も利用した。

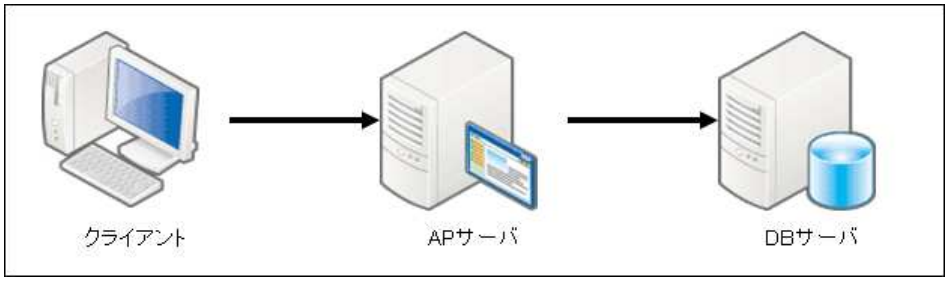


図 2-1 検証に使用した機器概要図

クライアント

種別	内容
OS	Windows XP Professional SP3(32bit)
ブラウザ 1	Microsoft Internet Explorer 6
ブラウザ 2	Mozilla Firefox 3

AP サーバ(VM-Ware 上に構築)

種別	内容
OS	Windows Server 2003 R2, Standard Edition
フレームワーク	Ruby on Rails 2.2
CPU	インテル Core2 Quad Q6600
メモリ	1GB

DB サーバ

種別	内容
OS	Windows XP Professional SP3(32bit)
DB	SQL Server 2005 SP2(32bit) MySQL 5
CPU	インテル Core2 Quad Q6600
メモリ	4GB

検証用 Rails で利用した Ruby コンポーネント

Rails の基本パッケージやそのバージョンは、Ruby の標準的なパッケージ管理ソフトである RubyGems で管理されている。検証のために利用したバージョンを表 2-1に列挙する。使用した RubyGems のバージョンは 1.3.1 である。

表 2-1 パッケージ一覧

パッケージ名	バージョン	パッケージ名	バージョン
Actionmailer	2.2.2	ruby-net-ldap	0.0.4
Actionpack	2.2.2	rubygems-update	1.3.1
Activerecord	2.2.2	sources	0.0.1
activerecord-sqlserver-adapter	1.0.0.9250	will_paginate	2.2.2
Activeresource	2.2.2	win32-api	1.0.4
Activesupport	2.2.2	win32-clipboard	0.4.3
cgi_multipart_eof_fix	2.5.0	win32-dir	0.3.2
Fxri	0.3.6	win32-eventlog	0.4.6
Fxruby	1.6.12	win32-file	0.5.4
gem_plugin	0.2.3	win32-file-stat	1.2.7
Hpricot	0.6	win32-process	0.5.3
log4r	1.0.5	win32-sapi	0.1.4
Mongrel	1.1.5	win32-sound	0.4.1
Rails	2.2.2	windows-api	0.2.0
Rake	0.8.3	windows-pr	0.7.2
ruby-debug-base	0.10.0	actionmailer	2.2.2
ruby-debug-ide	0.1.10	-	-

3. 検証作業の概略

3.1 検証項目

検証項目の選定方針は「2.1 検証項目選定方針（P.14）」にて述べた通りである。

環境

評価項目	評価内容
SQL Server 2005 への接続	Rails から SQL Server 2005 を利用することが出来るかについて検証する。

機能

評価項目	評価内容
画面作成支援	HTML フォームの部品化など、画面作成を容易にするための補完機能があるか。
DB 処理	DB にデータの登録／参照／更新／削除の方法。
HTTP セッション管理	セキュリティを確保した安全なセッション管理の方法。
GUI によるデバッガ	Rails のアプリケーション開発では、GUI によるデバッガが提供されているか調査する。
バッチ処理	業務アプリケーションでは、所定の時間に DB に対してバッチ処理を行うケースも考えられる。Rails でバッチ処理を行う方法を調査する。
規約に従っていない DB スキーマを採用した場合の制限	「規約」に従わない DB スキーマを採用した場合に、生産性の著しい低下や、ソースコードの可読性がていかしないかについて検証する。
Web サービスによるシステム連携	Web サービス(SOAP)によるシステム連携をすることができるのかについて検証する。
HTML の文字コード	HTML(ビュー)にて採用することが出来る文字コードを調査する。
SQL 文を明示的に発行する	SQL 文の発行に制約がないかについて調査する。

運用

評価項目	評価内容
出力ログ	Rails で提供されているログ機能やその使用方法について調査する。
運用監視項目	Rails のアプリケーションを本番稼働させるときに、運用監視ツールに対して何を監視項目とさせるか検討する。
ライセンス上の制約	ライセンス上の制約により、一般的な受託開発のビジネスモデルに影響が生じないことを確認する

セキュリティ

評価項目	評価内容
SSL への対応	SSL 利用における Web サーバと Rails の連携について検証する。

開発支援

評価項目	評価内容
テストツールの調査	単体テストの自動化ツールや、カバレッジ測定ツールについて調査する。

4. 検証結果

4.1 画面作成支援

目的

入力用の画面部品として入力用タグを、Rails の標準的な支援のみで使えることを基礎的なレベルで検証する。

検証方法

実際にコーディングをしてみることで機能を利用することが出来るかを検証した。

検証結果

Rails での画面コーディング

Rails のビューは HTML に留まらず XML や Ajax のための JavaScript を返すことが出来るが、ここでは HTML について記述する。

Rails では HTML の描画のために eRuby と呼ばれるテンプレートの仕様が利用されている。Java であれば JSP、.NET であれば aspx と同じような書式である。

Rails の標準では eRuby の実装として ERB が採用されている。テンプレートファイルの拡張子は erb である。図 4-1中では<%から%>まで囲まれた部分に Ruby 言語を埋め込むことが出来る。<%=と記述すると、タグ内での処理結果が文字列化されて埋め込まれる。

```
<p>
  <b>Title:</b>
  <span id="title"><%=h @entry.title %></span>
</p>

<p>
  <b>Body:</b>
  <span id="body"><%=h @entry.body %></span>
</p>

<%= link_to 'Edit', edit_entry_path(@entry) %> ¦
<%= link_to 'Back', entries_path %>
```

図 4-1 erb ファイルの例

ヘルパーメソッドによる入力部品の支援

モデル層やコントロール層との連携部分のコーディングを支援するために、画面部品を作成するためのヘルパーメソッドが用意されている。ヘルパーメソッドには Rails により提供されているものと、開発者により作成されるものがある。Rails には入力用部品毎(input タグや textarea タグなど)に専用のヘルパーメソッドが用意されており、開発者を支援する。

ヘルパーメソッドは「便利なメソッド」というような位置づけで提供され、「本質的ではないが書かねばならない処理」を少ないコーディング量で済むようにうまく設計されている。

例えば、既存データの編集画面では DB から取得したデータを、テキストフィールドなどにセットしておかなければいけない。また、入力データのバリデーションで失敗した場合は、テキストフィールドに赤印を入れて再入力を促すなどの処理も必要である。多くのアプリケーションで必要となる処理を Rails では、規約としてとりあつえるように、Railsよりヘルパーメソッドが提供されている。ヘルパーメソッドを活用することにより、開発が強力にアシストされる。

図 4-2はヘルパーメソッドを活用した例である。6 行目のヘルパーメソッド(text_field)には DB のカラム名を引数として渡している。これは、規約により、DB の title カラムを対象としたテキストボックスであることが定義づけられ、Rails により title カラムに関するバリデーション処理や表示処理、更新処理が支援される。

ここで挙げたもの以外の入力部品(ドロップダウンやラジオ、チェックボックスなど)についても、ヘルパーメソッドは用意されている。

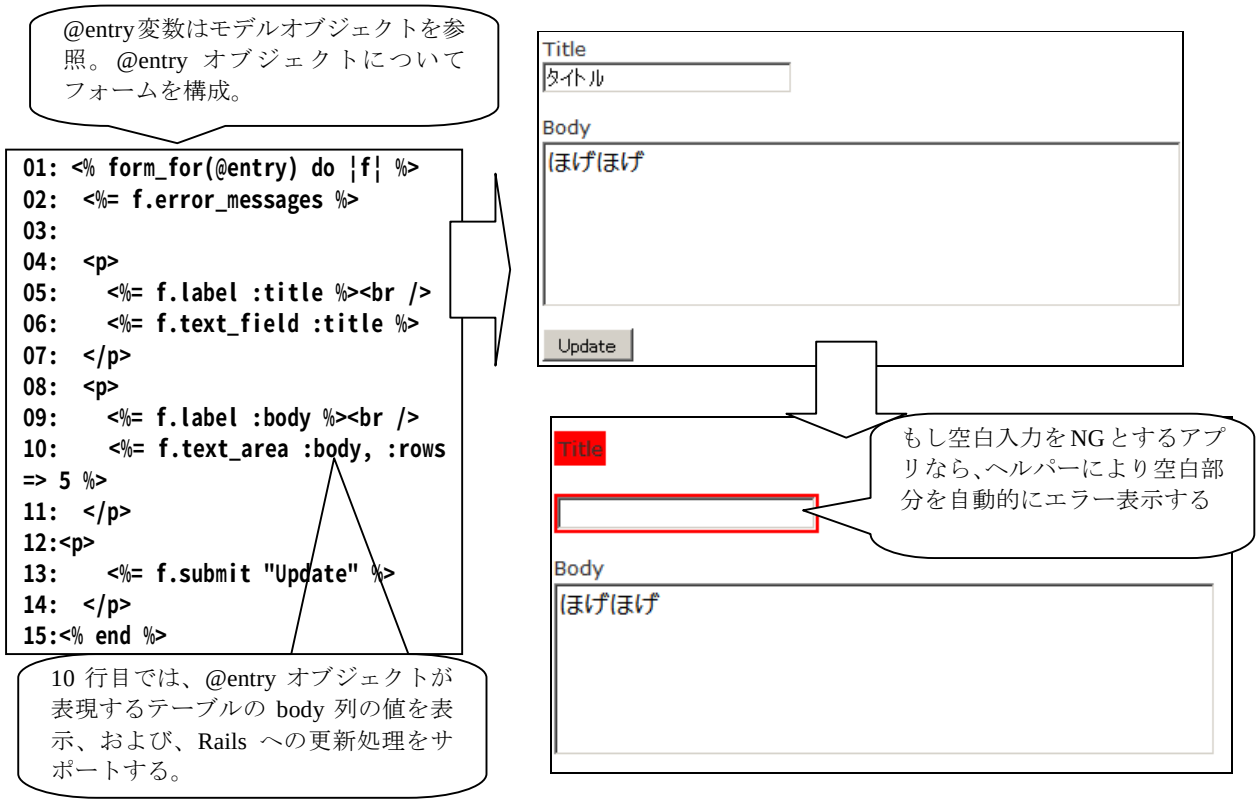


図 4-2 ヘルパーメソッドの利用例

画面のパーツ化

一つの erb ファイルで一つの画面を生成することも出来るが、部分テンプレート呼ばれる機能を使えば、画面の一部分だけを erb で記述することができる。部分テンプレートを他の erb ファイルから集めて一つの画面にすることも出来る。部分テンプレートを利用することで、画面に統一感をだしたり、同じような画面をいくつもコーディングしたりする必要がなくなり便利である。

Rails ではレイアウトと呼ばれる機能も存在しており、画面の全体的なレイアウトを erb で用意しておき、リクエストに応じてレイアウトの中の表示を切り替えることが出来る。レイアウトを用いることで、画面における見た目のヘッダやフッタのコーディングの負荷が低減される(図 4-3)。

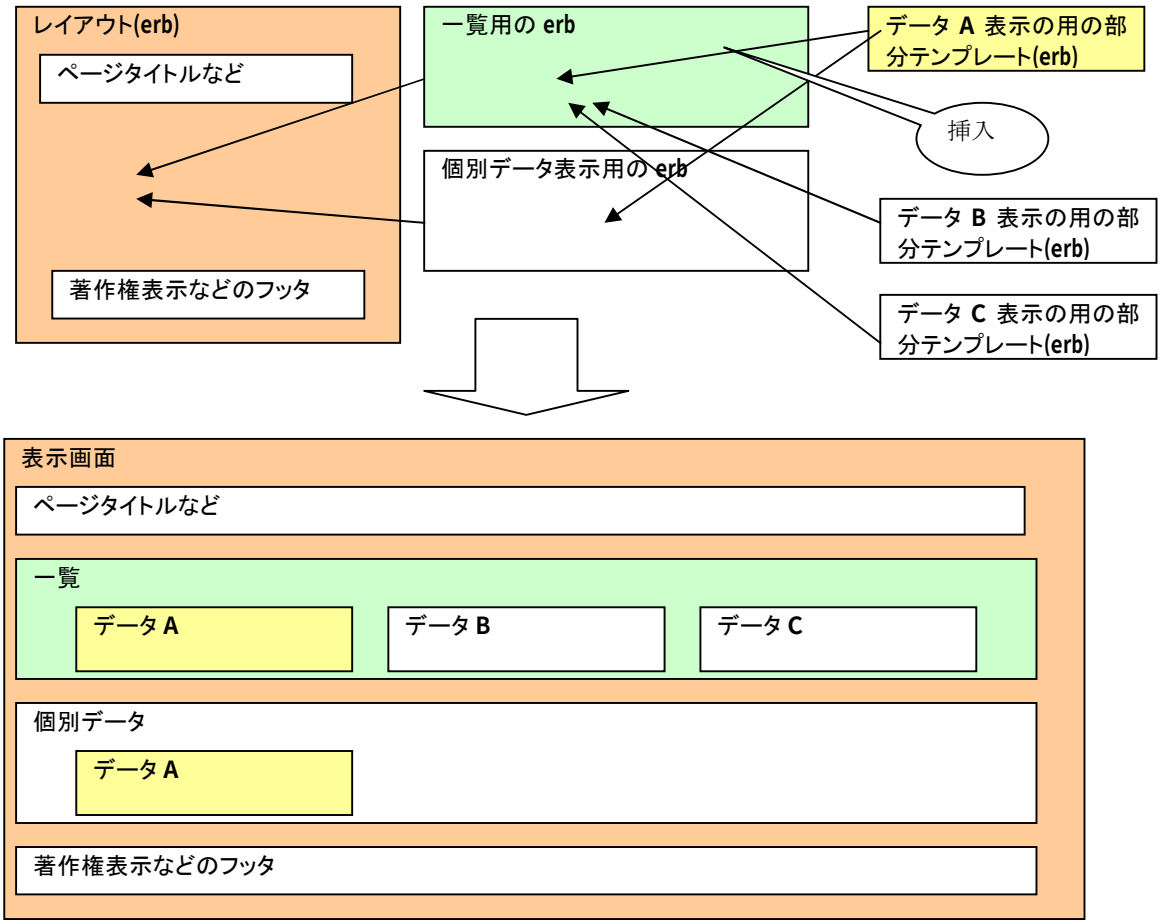


図 4-3 テンプレートの埋め込み

これらの埋め込みは、レイアウトと呼ばれる仕組みと rednder と呼ばれるヘルパーメソッドで実現されており、引数に部分テンプレート名を渡すことで実現できる。

ここでは、UserController と User モデルを例に簡単に説明する。

レイアウトは、処理コントローラの名前と同じ名前の erb ファイルが基本的には選択される。

app/views/layouts/user.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>Users: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>
<p style="color: green"><%= flash[:notice] %></p>
<%= yield  %>
</body>
</html>
```

例えば、UserController の show メソッドが実行されるようなリクエストがあった場合、、app/views/users/show.html.erb の描画結果が yield 部分に挿入される。

```
<h1>User info</h1>
<%= render :partial => 'user_info' %>
```

この例では部分テンプレートを指定している。実行時、render メソッドにより部分テンプレートが評価・挿入される。部分テンプレートのファイル名はアンダースコアで始まる erb である。先ほどの例では次のようなファイル名となる。

app/views/users/_user_info.html.erb

```
<p>Name: <%=h @user.name %></p>
<p>Age: <%=h @user.age %></p>
```

評価結果は次のように、一つの HTML として描画される。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>Users: show</title>
  <link href="/stylesheets/scaffold.css?1237132223" media="screen" rel="stylesheet"
type="text/css" />
</head>
<body>

<p style="color: green"></p>

<h1>User info</h1> <!-- app/views/users/show.html.erb 部分 -->
<p>Name: なまえ</p> <!-- app/views/users/_user_info.html.erb 部分 -->
<p>Age: 16</p> <!-- app/views/users/_user_info.html.erb 部分 -->

</body>
</html>
```

ここで示したものは一例である。この他にも様々なパラメータや利用方法が用意されている。

ヘルパーメソッドによるクロスサイト・スクリプティング対策

クロスサイト・スクリプティングは、Web アプリケーションが HTML の制御文字(< や > など)をエスケープせずに出力してしまうことにより、開発者の意図しない挙動を Web ブラウザに許してしまうことによりおこる。

悪意のあるユーザが、HTML として解釈できるタグ共に、ユーザの識別情報を悪意のあるユーザに送信するような JavaScript などを、Web アプリケーションへ入力する。その入力値を、何の対策も施されていない Web アプリケーションによって表示してまうと、JavaScript が実行されてしまい、ユーザの識別情報が悪意あるユーザに送信されてしまうなどの被害が発生する。クロスサイト・スクリプティングの詳しい仕組みや想定される被害についての詳述は Web 上での議論などに譲る。

図 4-4に erb の例を再び掲載する。

```
01:<p>
02:  <b>Title:</b>
03:  <span id="title"><%=h @entry.title %></span>
04:</p>
05:
06:<p>
07:  <b>Body:</b>
08:  <span id="body"><%=h @entry.body %></span>
09:</p>
10:
11:<%= link_to 'Edit', edit_entry_path(@entry) %> ¦
12:<%= link_to 'Back', entries_path %>
```

図 4-4 erb の例再び

erb では<%= %>で Ruby の処理結果を HTML に埋め込むことが出来るが、そののままでは HTML として解釈してしまう<などの文字をそのまま出力できてしまう。外部からの入力を表示しないと分かっている項目ならばそれでも良いが、外部からの入力も出力する(例えば掲示板アプリケーションにおけるコメント文など)箇所で、対策を施していない場合、悪意あるユーザにスクリプトを埋め込まれてしまう危険性が生じてしまう。

図 4-4は既に対策済みである。例えば 3 行目には<%=に h が付加されている(<%=h のようになっている)。h はヘルパーメソッドである。h メソッドは入力された文字列を、HTML エスケープして出力するメソッドである。h メソッドによりエスケープさせることで、クロスサイト・スクリプティングなどのクラック行為を防止することが出来る。

Rails でアプリケーションを作成する場合は、入力値を出力する際には必ず h メソッドを介するように習慣づけたい。

4.2 DB 処理

目的

ActiveRecord による DB 処理について次の点を検証する。

- 登録/参照/更新/削除(CRUD)が実現できること
- トランザクション制御ができること

検証内容

ActiveRecord における DB 処理について単純な CRUD 処理を検証・調査する。

検証結果

ActiveRecord の規約

「設定より規約」「DRY」[4]などの設計思想から、ActiveRecord には様々な規約が設定されている。規約の対象は、テーブル名や列名に対する命名規則であったり、列の型であったり、モデルクラスの命名規則に関わるものである。ActiveRecod の規約についてここで簡単に紹介する。

1）主キー列の名前は「id」を前提とする

ActiveRecord は id という名前の列をテーブルの主キー列として自動的に認識する。この機能により、列の追加や検索、テーブルの関連などに関わる定型的な処理の記述量を削減できる。また、主キーの名前はいずれのテーブルも同じ名前になり、コード全体の統一性や可読性が向上される。

2） 主キー列の型は整数型を前提とする

ActiveRecord では主キー列がインクリメンタルな整数型であることを規約として定められている。この規約によりテーブル定義の作成や、テーブルの結合やリレーションの参照などが簡便化されている。

3）テーブル名はモデルクラス名を複数形としたもの

ActiveRecord では、モデルのクラス名を複数形とした名前をテーブル名としている。これは、一つのモデルオブジェクトが 1 レコードを表し、テーブルはレコードの集合であることを(英語圏の人にとって)自然な形で表現する為である。例えば、Book クラスは Books テーブルを表現している。

4）外部参照列名は「参照先テーブル名_id」

「参照先テーブル名_id」とすることで、ActiveRecordは列名から外部参照であることを自動的に認識でき、ある程度は自動的に結合やリレーション先のテーブルの参照を実施することが出来る。また、この規約を利用し、アプリケーション作成者は少ないコーディング量で「1 対 1」や「1 対多」、「多対多」などのテーブル関係を表現することができる。外部参照していることが一目で分かるため、ソースコードの可読性も向上する。

[4] 「1.3 Ruby および Ruby on Rails の概要」の「設計思想(P.11)」参照

DB へのアクセス

ActiveRecord クラスを継承することで、DB テーブルを表現するモデルクラスを作成することができる。例えば、表 4-1のようなテーブルにアクセスする場合、USERS テーブルに対応するモデルクラス User クラスを作成する。

表 4-1 USERS テーブル

列名	型	備考
id	integer	主キー
username	varchar	

【User モデルクラス（user.rb）】

```
01: class User < ActiveRecord::Base
02: end
```

ActiveRecord クラスには様々な機能が実装されている。ActiveRecord クラスを継承することで、モデルクラスは DB アクセスのためのメソッドや、入力されたデータに対するバリデーション機能、リレーション機能など多数の機能を簡単に利用することが出来る。

表 4-2に SQL と対応するメソッドの例を挙げる。表 4-2に挙げたメソッドは一例であり、様々な用途に合わせてメソッドが用意されている。

表 4-2 SQL と Rails のメソッド対応例

SQL の種類	Rails のメソッド名	備考
SELECT	find	クラスメソッド、パラメータで検索条件を記述する
INSERT	save	インスタンスを新規作成した場合は INSERT 文が実行される。
UPDATE	update_attributes	インスタンスが DB のレコードから取得されていると、自動的に UPDATE 文が実行される。
DELTETE	destroy	インスタンスの ID 情報を使用して、指定レコードを削除する。

モデルクラスの使い方サンプル

【サンプルコード】

```
01: #  検索の場合
02: @user = User.find(params[:id])
03:
04: #  登録の場合
05: @user = User.new(params[:user])
06: @user.save
07:
08: #  更新の場合
09: @user = User.find(params[:id])
10: @user.update_attributes(params[:user])
11:
12: #  削除の場合
13: @user = User.find(params[:id])
14: @user.destroy
```

params が指しているものは、Web ブラウザから送信された form 内の入力データ。

1）検索

クラスメソッドの”find”を実行する。このとき画面データの id 情報を使用して該当オブジェクトを取得し、user インスタンス変数にデータを格納している。

この例では 1 行のみが検索されるが、複数行を検索する場合は

```
@users = User.find(:all, :conditions => [“age > ?”, params[:age]])
```

とする。

2）登録

クラスメソッドの”new”を使用して、画面データから User クラスのインスタンスを構成し、”@user”変数にデータを格納する。この時点では DB 上にデータは登録されない。そして、”save”メソッドを実行することで DB への INSERT 文が実行され、コミットされる。

3）更新

検索時に使用した”find”メソッドによって一旦データを取得する。そして、その変数を画面情報を元に変更を行う。例えば、10 行目の params[:user]の内容は次の状態である。

```
params[:user]={“username” => “me”}
```

この状態を、”username”列の値を”me”に変更したい。という状況であるとRails は判断し、10 行目でこの UPDATE 文が実行される。

4）削除

14 行目で”destroy”メソッドが実行されると、DB へ DELETE 文が実行される。

SQL インジェクション対策

検索用メソッドなどでは SQL 文の WHERE 句を自由に記述することが出来るが、WHERE 句を書く場合に気をつけることは SQL インジェクション対策である。Rails の場合は、次の例のように条件値の部分に?と入れ、配列に実際にセットしたい値を並べることで、自動的に SQL 構文にかかる文字をエスケープする。

```
@users = User.find(:all, :conditions => [“age > ?”, params[:age]])
```

特定の値の一括更新を防ぐ(意図しないデータ改ざんの防止)

モデルオブジェクトの値の更新は Hash オブジェクトを利用して一括で更新することが出来る。

```
params[:user] # => { :name => “new name”, :age => “28” }
@user = User.find(params[:id])
@user.attributes = params[:user] # @user の name と age 項目が上書きされる
```

しかしながら、悪意あるユーザなどがブラウザからのリクエスト内容を改ざんした場合、意図しないデータを上書きしてしまうことが可能になってしまう。次の例の場合、あるユーザのアクセス権限テーブルへの参照が変わってしまう。

```
params[:user] # => { :name => “new name”, :age => “28”, :acl_id => “20” } # acl_id を追加したリクエスト
@user = User.find(params[:id])
@user.attributes = params[:user] # @user の name と age と acl_id 項目が上書きされてしまう
```

こういった、意図しない値の一括更新を保護するための仕組みとして、ActiveRecord には attr_accessible メソッドと、attr_protected メソッドがある。

attr_accessible メソッドは一括更新してもよい項目を指定する。

```
class User < ActiveRecord::Base
  attr_accessible :name, :age
end
```

一括更新の対象から外された項目は、意図的に値をセットする必要がある。

```
@user.acl_id = params[:user][:acl_id] #  意図的に上書き
```

指定されていない項目は一括更新の対象からは除外される。attr_protected メソッドは逆に、保護対象としたい項目を指定する。

attr_protected メソッドでは保護項目の指定し忘れや漏れが生まれやすいため、安全性を考慮するのであれば、attr_accessible メソッドを利用したい。また、一括更新の有無にかかわらずミスを防ぐために attr_accessible メソッドは必ず利用した方が良いと考えられる。

トランザクション処理

Rails では DB より提供されているトランザクションを利用できる。

```
01: user = User.new(:username => "me" )
02: blog = Blog.new(:user => user, :name => "My new blog")
03:
04: begin
05:     # user と blog をトランザクション内で保存する
06:     ActiveRecord::Base.transaction do
07:         user.save
08:         blog.save
09:     end
10: rescue Exception => e
11: #  ロールバックされるとこのブロックが実行される
12: end
```

transaction スコープは、06 行目の”transaction do~end”で囲まれた部分(transaction ブロック)内がトランザクションとして処理される。

2 フェーズコミット

2 フェーズコミットはサポートされていない。

4.3 HTTP セッション管理

目的

Rails における HTTP セッションの管理方法を調査し、業務で使用する際に懸念事項ないかどうかを確認する。

検証方法

HTTP セッション管理について使用方法について公式ドキュメントや書籍にて調査した。

検証結果

表 4-3に挙げたような HTTP セッションの管理方法がある。システム開発者はいずれかの格納方法を選択する。いずれの場合もセッションに格納できるデータはシリアライズ可能な Ruby オブジェクトのみである。

デフォルトの格納方法は cookie_store であるが、cookie_store には後述する問題点があるため、アプリケーションの要件などで別途ファイルストア(p_store)や DB ストア(drb_store)を採用する。

今回は、多くの Web アプリケーションで使用されている、cookie_store とファイルへのセッション情報格納(p_store)について調査した。

表 4-3 セッション情報格納方法

名前	保存場所	備考
cookie_store	Cookie	何も指定しない場合は cookie_store となる
p_store	ファイル	
drb_store	データベース	
mem_cache_store	メモリ	かつてはメンテナンスされていたが現時点では非推奨
mem_store	メモリ	かつてはメンテナンスされていたが現時点では非推奨
―	―	ユーザによるカスタムな管理方法

cookie_store 以外の場合のセッション ID の受け渡し

Cookie を格納場所としない場合、ブラウザと Rails 間でのセッション ID の受け渡しは Cookie を使って図 4-5 のように動作している。

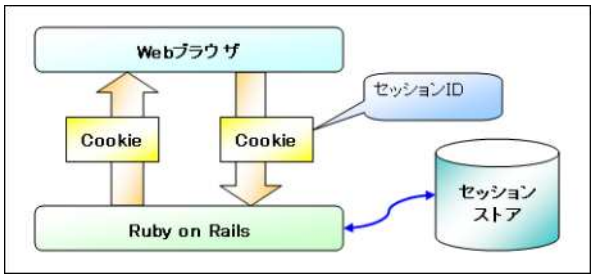


図 4-5 セッション管理の概念図

ネットワーク上に流れる情報はセッション ID のみである。これは、多くの AP ミドルウェア(Tomcat や ASP.NET、PHP など)で採用されている方式である。

クッキーに保存(cookie_store)を選択した場合の注意事項

Cookie をセッションの格納場所とした場合(cookie_store)、HTTP セッションのデータの保存場所は Cookie となる。つまり、データの保管はサーバサイドではなく、Web ブラウザとなる。

保存する情報は、ハッシュ値による改ざん対策は行われているが、盗聴対策（暗号化等）はされていないため、盗聴されて困るデータを格納してはならない。

以下に、Cookie にセッション ID を格納する処理を示す。

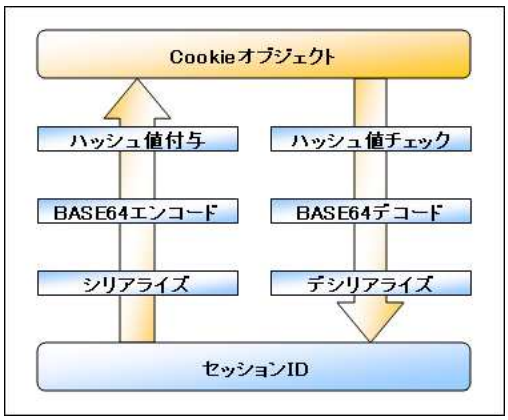


図 4-6 Cookie への入出力概念図

1）セッション情報をマーシャル(シリアライズ)する。
2）Base64 でエンコードする。
3）config.action_controller.session で設定したキーを元に OpenSSL の SHA1 を利用しメッセージダイジェストからハッシュ値を取得する。
4）Base64 化されたデータとハッシュ値を"--"で結合する。
5）4 の結果を Cookie に格納する。

Rails はハッシュ値を確認することで改ざんの有無を確認する。

Cookie へ格納できる情報は4KB までの制限がある。

ファイルや DB にセッションデータを保存する場合の注意点

Rails は、セッションタイムアウトの機能を持たないため、保存されたセッション情報は削除されない。定期的にセッションファイル(./tmp/sessions)やレコードの更新日を確認して、削除するバッチなどを構成する必要がある。

4.4 GUI によるデバッグ

目的

.NET や Java アプリケーションの開発では、統合開発環境に搭載されているデバッガを使用することが日常的に行われている。デバッガ上では、ブレークポイントを使って確認したい箇所付近で処理を一時停止させ、変数がとる値を確認しながらステップ実行をする。Rails での開発環境においても、同様の機能が利用できるかどうか検証した。

検証方法

NetBeans での基本的なデバッガ操作について確認。

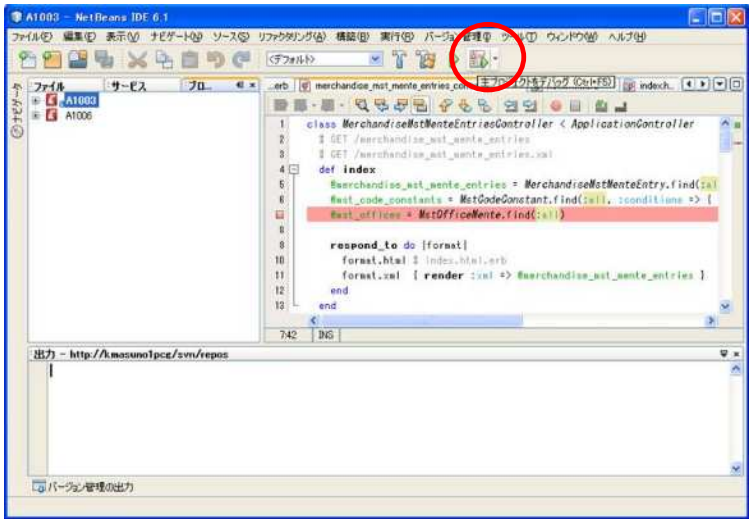
検証結果

NetBeans を利用すると、GUI 上でブレークポイント、ステップ実行共に実現可能である。

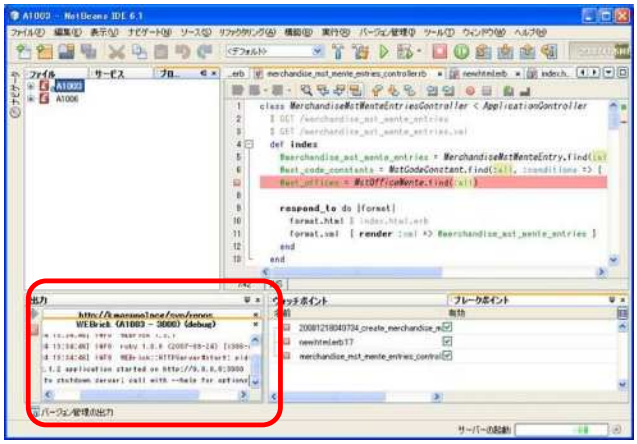
次の手順で、ブレークポイントを設定し、各変数の値を確認することができる。

【主プロジェクトをデバッグ】

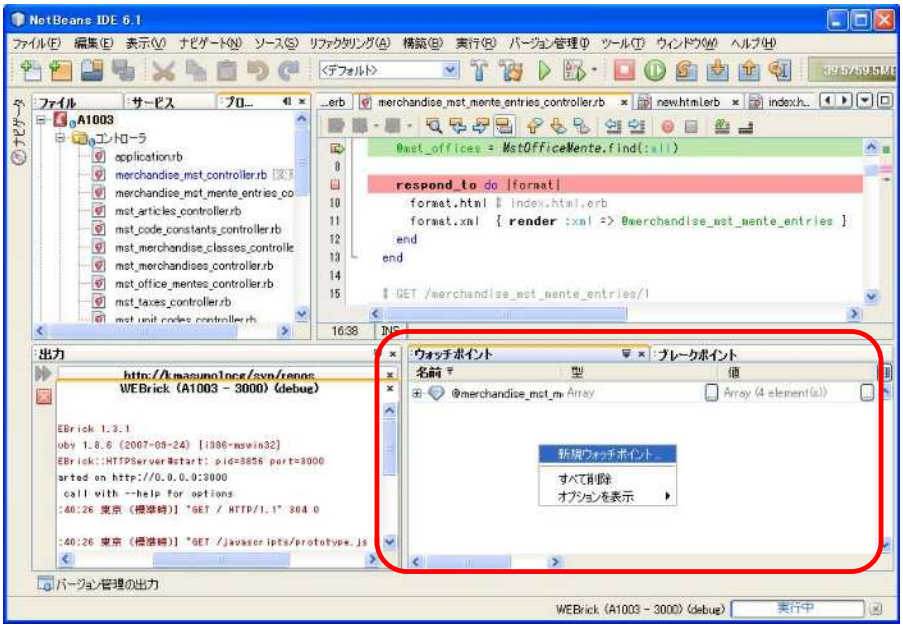
ボタンをクリックする。



Web サーバ(WEBrick や Mongrel)がデバッグモードで起動していることを確認する。

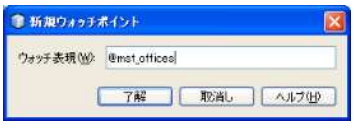


ブラウザを起動し、デバックを行うアプリケーションを開く。



赤色の枠の中に変数の型と、値が表示される。

その際に配列や、ハッシュの場合は、ツリーをドリルダウンして詳細に確認できる。さらに、変数名を指定して、確認したい場合には、『ウォッチポイント』ウインド内で「右クリック」→【新規ウォッチポイント】をクリック。



変数名を入力して【了解】をクリックすると、指定の変数名も表示される。

4.5　バッチ処理

目的

業務アプリケーションでは、夜間にバッチ処理を行うケースも多い。バッチ処理を行う際に Web アプリケーションで作成したコード部品を流用できれば、同じ処理を 2 回書くことがなくなり、生産性／保守性が向上する。

Rails 部品や作成したクラスを使用して、バッチ処理を作成できるかを確認する。

検証方法

実際にサンプルプログラムを作成して確認した。今回利用した部品はモデルクラスである。

次の手順によってモデル層のオブジェクトを使用できる。処理のイメージは図 4-7のとおりである。

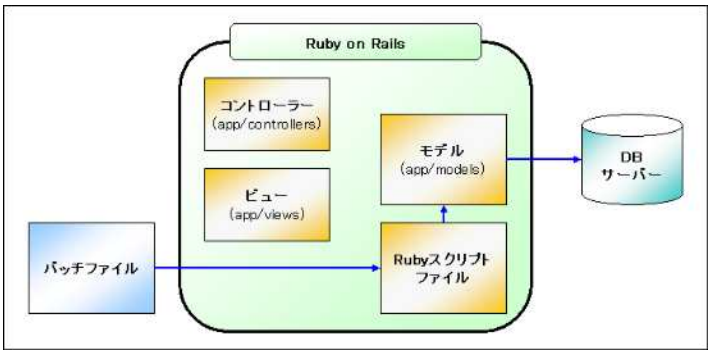


図 4-7　バッチ処理時の動作の流れ

バッチファイルでは Rails の機能のひとつである script/runner を実行する。ここでは Rails に指定した Ruby スクリプトファイルの内容を実行するように処理を runner に渡している。

```
Ruby script/runner 'eval(IO.readlines("app/batch/batch-sample01.rb").join)'
```

Ruby スクリプトファイルでは実際の処理を記述する。

以下の例では、Order クラスを経由してデータを検索と登録している。

```
06: # Order モデルクラスを経由して DB にアクセスする
07: if last_order = Order.find(:first, :order => "id DESC")
08:     an_order0 = Order.new
09:     an_order0.name = last_order.name
10:     an_order0.email = last_order.email
11:     an_order0.address = last_order.address
12:     an_order0.pay_type = "cc"
13:     #  例外処理は適宜実施のこと
14:     an_order0.save
15: end
```

検証結果

本検証によりバッチ処理は動作可能であるが、eval 関数を使用するなど、標準的な方法で実装することができなかった。プラグインとして提供されている非同期ライブラリなどを検証・非同期基盤を実装するなどを検討し、さらなる検証・調査が必要である。

4.6 規約に従っていない DB スキーマを採用した場合の制限

検証目的

Railsには O/R マッパーとして ActiveRecordと呼ばれるコンポーネントが採用されており、モデル層を形成するコンポーネントとなっている。ActiveRecord は Rails のコンポーネントの一部であるため、Rails の設計思想のひとつである「設定より規約[5]」が設計に取り入れられている。

もしも、実案件において ActiveRecord の「規約」に従っていないテーブル定義を採用せざるを得なくなった場合、どの程度アプリケーション構築に対して制約が発生するのかを調査する。

ActiveRecordには、生産性や利便性の為に、テーブル定義やテーブルを表現するモデルクラスに対して設計やコーディングに対して「規約」が定められている。ここでの「規約」とは明確に定義づけられているものではなく、「設定より規約」を哲学に設計されているという意味である。「規約」にそった設計やコーディングをすることにより、ソースコードの可読性やシステムの生産性に寄与すると謳われている。

しかし、「規約」に定められていない振る舞いが必要となった場合に、次のような懸念が考えられるため本検証で確認する。

- 必要な機能が実装できない
- 生産性が著しく低下する
- ソースコードの可読性が著しく低下する

検証方法

検証対象とした規約

ActiveRecord に定められている規約に従わないテーブルを作成し、制約がどの程度現れるかを検証した。検証では次の規約に反したテーブルを作成した。

検証のために作成したテーブル定義とRailsのソースコードは、付録である「A.1 規約に従っていないDBスキーマを採用した場合の制限の検証(P.72)」に記載した。

5）主キー列の名前は「id」を前提とする

ActiveRecord は id という名前の列をテーブルの主キー列として自動的に認識する。この機能により、列の追加や検索、テーブルの関連などに関わる定型的な処理の記述量を削減できる。また、主キーの名前はいずれのテーブルも同じ名前になり、コード全体の統一性や可読性が向上される。

6） 主キー列の型は整数型を前提とする

ActiveRecord では主キー列がインクリメンタルな整数型であることを規約として定められている。この規約によりテーブル定義の作成や、テーブルの結合やリレーションの参照などが簡便化されている。

7）テーブル名はモデルクラスを複数形としたもの

[5] 「1.3Ruby および Ruby on Rails の概要」の「設計思想(P.11)」を参照。

ActiveRecord では、モデルのクラス名を複数形とした名前をテーブル名としている。これは、一つのモデルオブジェクトが 1 レコードを表し、テーブルはレコードの集合であることを(英語圏の人にとって)自然な形で表現する為である。例えば、Book クラスは Books テーブルを表現している。

8）外部参照列名は「参照先テーブル名_id」

「参照先テーブル名_id」とすることで、ActiveRecord は列名から外部参照であることを自動的に認識でき、ある程度は自動的に結合やリレーション先のテーブルの参照を実施することが出来る。また、この規約を利用し、アプリケーション作成者は少ないコーディング量で「1 対 1」や「1 対多」、「多対多」などのテーブル関係を表現することができる。外部参照していることが一目で分かるため、ソースコードの可読性も向上する。

検証した機能

ActiveRecord の持つ次の各機能について、設定機能を利用することで、制約の有無や利用の制限が回避されるかどうかを検証した。

1）レコードの作成

レコードの作成は可能であるか

2）レコードの参照

レコードは検索されるか、検索されたレコードは正しい値を保持しているか

3）レコードの更新

更新された値は正しく DB に反映されるか

4）レコードの削除

削除は問題なく実施されるか

5）外部参照 1 対多

外部参照されているレコードは自動的に参照できるか

6）外部参照 多対多

外部参照されているレコードは自動的に参照できるか

7）複合主キー

複合主キーを利用することが出来るか

検証結果

実案件において、規約に沿っていないテーブル定義を利用せざるをえない場合でも、設定によって ActiveRecord の大半の機能は利用可能である。

ただし、次のような設定のみでは回避できない制約も存在しており、留意しておくべきである。

- 主キーの型を整数型以外にした場合、DB の自動採番機能を利用することが出来ない。レコードを追加する場合、採番の機能をアプリケーション作成者が提供しなければならない。
- 複合主キーは Rails では未対応であり、プラグインなどの外部機能を利用しなければならない。

.

4.7 SQL Server 2005 への接続

検証目的

SQL Server 2005 の利用が可能であるかについて検証する。

Rails を前提にスクラッチで新システムを開発するケースよりも、既存システムに追加する形でのシステム構築が期待されることが予想される。Rails の事例報告では DB として MySQL や PostgreSQL などが紹介される事が多いが、既存の企業情報システムでは Oracle や SQL Server が採用されていることが圧倒的に多い。

今回は、Windows を基盤とした場合に採用されることが多い SQL Server への接続について検証する。

設定手順と検証内容

基本的な CRUD 操作及び、DB の作成支援機能のチェック(DB マイグレーション)を検証した。Rails から DB への接続方法は付録である「A.2 Ruby on Rails から SQL Server2005 への接続設定手順(P.76)」に記載した。

検証結果

Rails は、Ruby DBI と呼ばれる汎用 DB インターフェイスを使い、アプリケーションの設定ファイルに記述された接続先 DB 情報にしたがって対応するドライバを使用して接続する。SQLServerでは、ADO 接続を行うドライバが提供されており、今回の検証では、このドライバを使用した。しかし、最新の Ruby DBI では ADO 接続がサポートされなくなったため、RubyDBI のバージョンを 0.4.0 から 0.2.2 にダウングレードして接続した。

ODBC 接続でのドライバも提供されているが、今回の検証では正常に接続することができなかった。今後、SQLServer に関してはますます利用が難しくなる可能性があり、できるだけ MySQL や PostgreSQL など Rails との接続実績が多い DBMS を採用することが無難であると思われる。

ただし、今後、Rails で受託開発を受注していくとするならば、SQLServer のドライバの需要は高い。したがって、SQLServer のドライバを開発することも検討する必要がある。

4.8 Web サービスによるシステム連携

検証目的

情報系や管理系などのシステムの構築をターゲットに Rails を使う場合、他システムの情報を取得するなどのシステム間の連携が必要となる可能性がある。そのため、システム連携の方法について検証した。

検証対象としたシステム連携技術は、アーキテクチャ間の差異を吸収しやすい Web サービスとした。本検証での Web サービスとは SOAP による通信の事を指す。

検証環境と内容

Web サービス検証のための環境

Rails の検証のために連携対象となるシステムを表 4-4の構成で作成した。

Web サービスが実装されている事例は、経験的に Java で作られたシステムが多いことから、本検証では連携対象のシステムは.NET ではなく Java で作成した。

表 4-4 Web サービス検証環境

項目名	内容
言語	Java 5
アプリケーションサーバ	Tomcat 5.5
Web サービスライブラリ	Axis 1.4

検証内容

Rails と Java それぞれで Web サービスを実装し、お互いから呼び出しあうことでどの程度 Web サービスによるシステム連携が可能であるかを検証した。

Web サービスをお互いのシステムから呼び合うために、WSDL を利用した。WSDL とは、Web サービスを呼び出すために必要な、Web サービスの場所(URL)や、引数の型・戻り値の型などの情報が記載されている XML ドキュメントである。WSDL は Rails, Java 両システムの Web サービス用のフレームワークから自動生成されたものを使った(図 4-8)。

詳細については付録である「A.3 Web サービス検証結果詳説(P.77)」に記載した。

図 4-8 Web サービス検証環境

検証結果

単純なリクエストとレスポンスの取得を検証した結果、検証の範囲内[6]ならば Web サービスによる連携を利用するには特に問題はないことが分かった。

しかし、Rails による Web サービスの実装やサポート状況などについて次のような懸念点が存在することがわかった。実案件にて Web サービスを利用する場合には、案件毎に検討・検証を行う必要がある。

■ 懸念点 1

Rails の Web サービス実装(呼び出され側)の提供が廃止されている。

Rails の Web サービスに対するアプローチが SOAP によるものから、REST 指向に変更された。それに伴い、Rails の公式の SOAP 実装(Active WebService:AWS)がバージョン 1.2 系を最後に廃止された。検証で用いた AWS は、古いバージョンで搭載されていた AWS をバージョン 2.2 系の Rails でも動くようポーティングされたものである。

■ 懸念点 2

WSDL の互換性の検証が本検証範囲内だけでは足りない。複雑な引数や戻り値が定義された WSDL を正確に soap4r が読み取れるかどうかはわからない。Rails から他システムを呼び出すシステムを構築する場合は、事前に検証する必要がある。

[6]詳細については付録である「A.3 Web サービス検証結果詳説(P.77)」に記載した。

4.9 HTML の文字コード

検証目的

様々な事情により UTF-8 を採用できない案件で、Web 画面を UTF-8 以外の文字コードを使ってシステムを構築することができるかどうかを検証する。

検証内容

概要

SQL Server 2003 を使った場合、Web 画面で UTF-8、Shift-JIS(Windows-31J)、EUC-JP、JIS の文字コードを利用できるかどうかを検証した。

詳細

DB 接続層は、Rubyランタイムの利用文字コードに合わせてDBから得られた文字列の文字コードを変換する必要がある。

Windows サーバ上で稼動する Rails から、SQL Server への接続の概要を図 4-9に示した。DB の文字コードの変換は図 4-9中 Ruby Win32OLE 層で実施される。Ruby Win32OLE 層以降、画面のレスポンス処理まで文字コードの変換層が存在しないため、Ruby Win32OLE 層で変換された文字列はそのまま用いられる。そのため、本検証では Ruby Win32OLE 層での変換可能文字列を中心に調べた。

なお、本検証では Windows であるから図 4-9のような構成であるが、利用する OS や DB によって変換する層やコンポーネントは変わる。検証のために Rails に施した設定は、付録である「A.4 画面で UTF-8, Shift-JIS を使うための設定(P.80)」にて記載した。

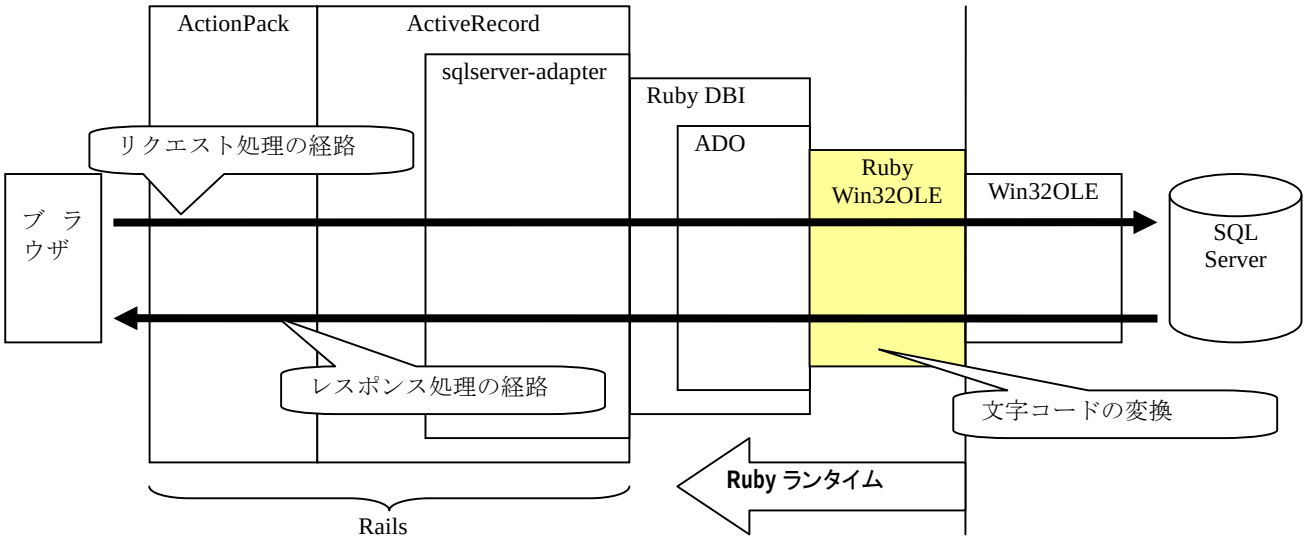


図 4-9 SQL Server への接続経路

検証結果

本検証での環境(Windows + SQL Server 2003)の場合、UTF-8もしくはShift-JISを使うことが出来るが、実用上や未知の問題の発生を避けるためには UTF-8 を利用すべきであると考えられる。

本検証の環境において UTF-8, Shift-JIS 以外を利用できない理由として、Win32OLE を担当している Ruby 層が、UTF-8, Shift-JIS 以外の文字コードに対応していないため、DB からの返された文字列の文字コードを UTF-8, Shift-JIS 以外に変換することが出来ないためである。

さらに、Shift-JIS を使う場合次の問題があることが分かった。

Chars クラスを使う文字列処理の問題

UTF-8 以外の文字列を利用すると、マルチバイトの文字数カウントなどの処理にて問題が出ることが分かった。Rails では、マルチバイト処理のために「ActiveSupport::Multibyte::Chars」というクラス(以下 Chars クラス)が用意されている。この Chars クラスは Unicode のみサポートしているため、文字数のカウントなどマルチバイトを考慮する必要のある処理で問題が出る。

Chars クラスは Rails の提供メソッドである truncate メソッドなどで使われている。truncate とは、表示用に長い文字列を短くするためのメソッドである。設定された長さまで文字を切り取り、末尾に…を付け加える関数である。

検証コードを図 4-10に示す。検証の結果、UTF-8(図 4-11)では正確にマルチバイトを考慮して文字列が省略されているが、Shift-JIS(図 4-12)ではマルチバイトの 2 バイト目を考慮せずに文字列の省略処理が行われてしまい、文字化けのようになってしまっている。なお、切り取り文字数が 3 文字以下は、truncate 関数は対応していないため、3 文字以下の場合の結果は無視する。

```
<% (1..12).each do |i| %>
    <%= i %>: <%= truncate('①２３４５６７８９０', i) %><br />
<% end %>
```

図 4-10 Chars 影響検証コード

```
1: ①２３４５６７８...
2: ①２３４５６７８９...
3: ...
4: ①...
5: ①２...
6: ①２３...
7: ①２３４...
8: ①２３４５...
9: ①２３４５６...
10: ①２３４５６７８９０
11: ①２３４５６７８９０
12: ①２３４５６７８９０
```

図 4-11 UTF-8 の実行結果

```
1: ①２３４５６７８９...
2: ①２３４５６７８９・..
3: ...
4: ・..
5: ①...
6: ①・..
7: ①２...
8: ①２・..
9: ①２３...
10: ①２３・..
11: ①２３４...
12: ①２３４・..
```

図 4-12 Shift-JIS の実行結果

4.10　SSL への対応

検証目的

SSL を使った Web アプリケーションを作成できるかどうかを検証する。

検証内容

次の二点について検証した。

1）Web サーバが SSL で受けた通信を、Rails へ正しく転送できるか

暗号化処理や解読処理および証明書の検証などの SSL 通信に関わる処理自体は、Rails ではなく Web サーバ(Apache など)の役目にあたるため、SSL 利用における Web サーバと Rails の連携(図 4-13参照)について検証した。検証のための Web サーバには Apache2.2 を利用した。

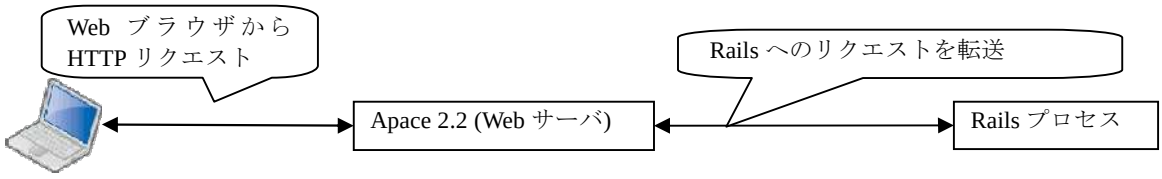


図 4-13　HTTP リクエストを転送

2）自サイト内リンクの URL は https, http と使い分けができるか

SSL を使わない場合、自サイト内のリンクの URL に付加されるプロトコルは http となる。Rails も特に何も設定しないかぎりはページ内リンクを http とする仕様である。

次について検証する。

- 要件の変更などで SSL を採用した場合、簡単にリンク表記を https へ切り替えて利用できるか
- SSL と非 SSL が混在する Web アプリケーションでも問題ないか。

検証結果

Web サーバが SSL で受けた通信を Rails へ正しく転送できるか

次の 2 つの条件を満たすならば転送できる。

Web サーバは Rails へリクエストを平文で転送できる

リクエスト転送時に X_FORWARDED_PROTO ヘッダを付加することができる

■ リダイレクトについて

Rails は、受け取った HTTP リクエストの X_FORWARDED_PROTO ヘッダに'https'という文字列が設定されていると、SSL を処理しているフロントエンドからのリクエストであると判断する(図 4-14参照)。この判断により、自サイトへのリダイレクトを Web ブラウザへ返す際、URL を正しく https と設定してリダイレクトを返す。

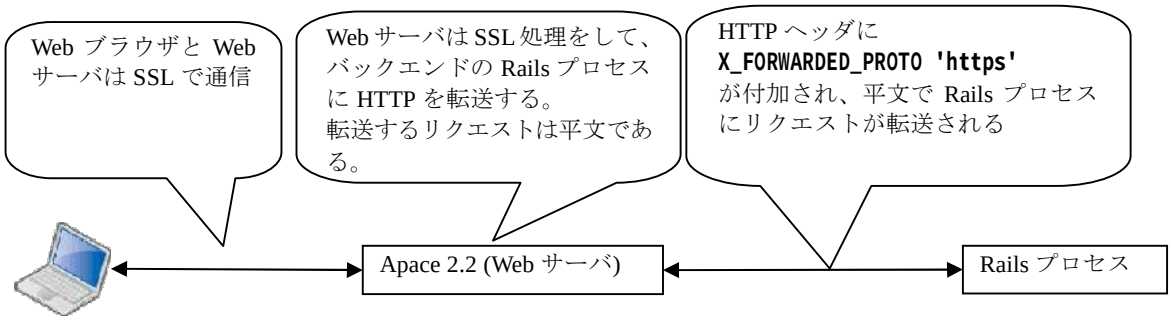


図 4-14 SSL リクエストを Rails に転送

Web アプリケーション内リンクの URL は https, http と使い分けができるか

次のようにすることで、SSL へのリンクは問題なく利用できる。

1）自サイトへのリンクは出来るだけ相対パスで記述する

相対パスが指定されているリンク先は、ブラウザが接続しているプロトコルを継承する。そのため、トップ画面への誘導を https としておくことで、相対パスが指定されているリンク先は自動的に https で接続することが出来る。

url_for, link_to などの Rails のリンク記述に関するヘルパーメソッドを使っている限りは、相対パスとして記述される。

2）部分的に SSL や非 SSL として接続したい場合は、絶対パスで記述しヘルパーメソッドに:protocol パラメータを付記する

次のように、:protocol パラメータを付記することによりプロトコルを切り替えることが出来る。:only_path に false を設定することで絶対パスでリンクが記述される。

```
<%= link_to 'Back', documents_path(:only_path => false, :protocol => 'https') %>
  # => <a href="https://tshinoda1pc/documents">Back</a>
```

検証中に分かった注意点

■ HTTP セッション用の Cookie をセキュア属性にする

SSL で接続する Web の場合、HTTP セッション維持のための Cookie をセキュア属性とする必要がある。設定は対象となるコントローラに対して次のように設定する必要がある。

```
class ApplicationController < ActionController::Base
  session :session_secure => true
end
# =>
# Set-Cookie:
# _TestDB_session=BAh7BiIKZmxhc2hJQzonQWN0aW9uQ29udHJvbGxlcjo6Rmxhc
# 2g6OkZsYXNo%0ASGFzaHsABjoKQHVzZWR7AA%3D%3D--ab12e83fc331
# 51d988b7abad71ad8bd39a85e9ca; path=/; secure
```

■ Cookie をセキュア属性にする

SSL 通信時以外に送信すべきでない Cookie が存在する場合は、セキュア属性を次のように付けることができる。

```
cookies[:key] = {
    :value => 'a yummy cookie',
    :secure => true
 }
```

■ SSL・非 SSL 通信を強制する

Rails 作者の手による「ssl_requirement」Plug-in を利用することで、コントローラ中の機能毎に、SSL 通信や非 SSL 通信を強制することが出来る。Plugin は次のようにコーディングすることで、利用することができる。

app/controllers/application.rb

```
class ApplicationController < ActionController::Base
  include SslRequirement
end
```

app/controllers/documents_controller.rb

```
class DocumentsController < ApplicationController
  ssl_required :show
  ssl_allowed   :new

  def show
    #  必ず SSL でアクセス。http でアクセスすると https にリダイレクトされる
  end

  def new
    # SSL でも http でも可能
  end

  def other
    # http でアクセス。https でアクセスすると http にリダイレクトされる
  end
end
```

4.11　テストツールの調査

調査目的

テストツールの実用性について調査した。

Web アプリケーションシステムを構築するにあたり、テストツールの有無は生産性や品質に直結する。品質に対して高まる要求や、短納期化する現代において、まともなテストツールが存在しない開発環境は選択できない状況である。

また、テストの存在はコードのリファクタリングの前提ともなり、短いイテレーションで改良を繰り返すような開発では必須となる。

調査内容

テストツールについて次の点を調べた。

- 利用できるテストツールの存在について
- テストツールの適用対象(単体テスト・結合テスト・システムテストの何れか)
- コマンド一つでテストが可能か
- テスト用のデータローダは存在するか
- テストできる範囲
- カバレッジ測定ツールの存在

調査結果

利用できるテストツールの存在について

Java 用の単体テスト用フレームワークである JUnit に似たテストツールが Rails 標準としてサポートされている。

Rails では、テストツールについて進化が著しく、RSpec や Cucumber と呼ばれる、画面遷移や画面入力までサポートしたさらに洗練されたテストツールも開発・提案されている。主流がどのテストツールになるか流動的な状況であるが、本検証では Rails 標準添付のテストツールを調査対象とした。

テストツールの適用対象(単体テスト・結合テスト・システムテストの何れか)

単体テストと Rails 内部のみを対象とした結合テストがテストツールの対象範囲である。

コマンド一つでテストが可能か

可能である。特定のテストのみを実行することも可能である。テストに失敗したケースはテスト終了後表示される。

テスト用のデータローダは存在するか

フィクスチャと呼ばれる仕組みが用意されている。テスト対象となるモデル用データを YAML や CSV 形式で記述でき、ロードも簡便な方法が用意されている。

テストできる範囲

Rails によるジェネレータによって、モデル・コントローラ・ビュー(JavaScript などを除いて)などのテストコードが生成される。Rails の標準的なモデル・コントローラ・ビュー以外のコードについても適切にテスト用コードを記述することでテスト対象とすることが出来る。

他のテストツールと比較して、特徴的な部分としてビュー、つまり生成された HTML に対するテストが可能な点にある。HTML 描画結果をテストコードで取得し、CSS セレクタの文法を使ってアサーション対象の文字列を抽出し、HTML が正しく生成されていることをテストすることが出来る。

カバレッジ測定ツールの存在

RCov と呼ばれるツールが存在している。カバレッジ測定での質的指標には、次の指標が多く使われる。

- C0: ステートメントカバレッジ(命令網羅率)
- C1: ブランチカバレッジ(分岐網羅率)
- C2: コンディションカバレッジ(条件網羅率)

RCov で測定できるカバレッジ測定の質的指標は C0 であり、各行が少なくとも 1 回は実行されていることを測定する。

図 4-15は RCov の結果を視覚的に表示させた例である。テストが通過していない行は赤く表示される。

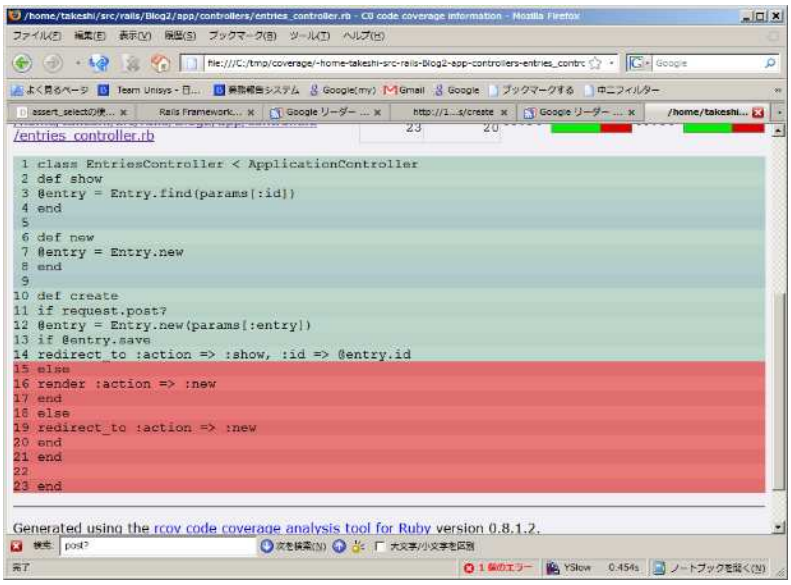


図 4-15 RCov 表示結果

実際のソースコードでは、1 行の間に複数の分岐が存在することは珍しくなく、単体テストの目的からすると「目の粗い」測定基準である。そのため、完全なカバレッジ測定ツールと評価できないが、運用ルールなどを定めることで、一定の生産性や品質保持に貢献することは可能である。

4.12 ライセンス上の制約

調査目的

Ruby 言語および Ruby on Rails のライセンスを調査し、一般的な受託開発のビジネスモデルに制約や問題が生じないことを確認する。

調査内容

Ruby 言語および Rails のライセンスを調査することで、受託開発のビジネスモデルに制約が生じることがないかを調査する。

調査結果

この調査結果は法律の専門家による厳密な法解釈によるものではない。一般的なライセンスの解釈に基づいているので注意すること。

Ruby 言語および Ruby on Rails のライセンスは次の通りである。

■ Ruby 言語

次の何れかのライセンスを選択する。

- GNU 一般公衆利用許諾契約書 バージョン 2 (いわゆる GPL2)
- Ruby 独自ライセンス

いずれにしても、Ruby 自体を改変しない限りは問題がない。ただし、仮に改変した場合はそれぞれのライセンスに従うことになる。GPL2 を適用した場合などは、改変 Ruby を実行する者(例えば顧客。Web 利用者ではない)に対しては、ソースコードを開示する必要がでてくる可能性がある。

■ Ruby on Rails

MIT ライセンスが適用される。

Rails に対して改修や改変などを施したとしても、著作権表示とライセンスを変えない限りにおいて、再配布や実行に関して何らかの義務や制限が付くことはない。

4.13　出力ログ

調査目的

出力されるログのフォーマットや内容、およびログファイルの運用に対して要求されることが多い要件を調べることにより、実運用面で問題が生じ得ないかを検証した。

調査内容

ログの内容やログファイルの運用に対して、一般的に求められるであろう機能を次の 5 点と設定し、各点について調査した(5 は Rails 特有の問題)。

1）ログの出力先の変更
2）ログレベルの種類と出力レベルの設定
3）ログのローテーション機能
4）デフォルトの出力フォーマットとカスタマイズ
5）Rails 固有のログ機能のオフについて(ANSI 制御シーケンスの設定)

調査結果

運用要件次第ではRailsの標準ロガーの運用や出力内容では機能不足である可能性がある。各システムの運用要件などに照らし合わせて、適切なロガーや設定を施したのちに実運用に投入すべきである。

特に次の点が問題であると考えられる。適切な設定・代替手段を採用する必要がある。それぞれの問題の解決法は、個々の調査結果に記した。

- Rails 標準のログフォーマットでは時刻が出力されない
- Rails 標準のログフォーマットではログレベルが分からない
- Rails 標準のロガーではローテーション機能を持っていない

ログの出力先の変更方法

config/environment.rb に対して、config.log_path 変数を変更することで可能。

ログレベルの種類と出力レベルの設定

ログレベルは、開発環境・テスト環境・本番環境などの各環境に設定することが出来る。

ログレベルの一覧と各ログレベルの役割は表 4-5の通りである。

表 4-5 用意されているログレベル

ログレベル	内容
DEBUG	デバッグ用の情報等を出力するレベル。
INFO	アプリケーションの動作報告に利用されるレベル。Rails により、HTTP アクセスや画面描画などの情報は INFO レベルで自動的に出力される。
WARN	警告すべき内容が報告されるレベル。
ERROR	アプリケーションで定義されたエラーが生じた際のログ出力に利用されるレベル。
FATAL	動作環境に処理続行不可能な障害が発生した場合などに使われるログレベル。アプリケーションの開発時では、コーディング中のシンタックスエラーや、DB 停止時などに表示されることが多い。

ログのローテーション機能

Rails 標準のロガーではローテーション機能が搭載されていない。ローテーションを実施する場合は、次の 2 つの手段を代替として検討する必要がある、

1）ロガーとして Log4r などの代替ロガーを利用する
2）logrotate などのログのローテーションをサポートする運用管理ツールの導入を検討する

デフォルトの出力フォーマットとカスタマイズ

図 4-16はログの出力コードとデフォルトの出力フォーマットの例である。標準の状態の場合、ログの出力された時刻とログレベルが表示されない。障害時の問題切り分け作業や、自動監視などの様々な運用に支障をきたすと考えられる。

```
-------------------- 出力のためのコード例 ---------------
logger.debug "debug message"
logger.info "info message"
logger.warn "warn message"
logger.error "error message"
logger.fatal "fatal message"
--------------------- 出力例 --------------------------
debug message
info message
warn message
error message
fatal message
```

図 4-16 Rails 標準ロガーによるログ出力例

実用的なログフォーマットとするためには、出力部分をカスタマイズして必要な情報が同時に出力されるようにするか、別のロガー実装(Log4r など)を利用するなどする。図 4-17は、フォーマットのカスタマイズで対応した場合である。

```
-------config/enviroment.rb、もしくは、
       config/enviroments/*.rb に次を追記 -----------------
config.logger = Logger.new(config.log_path)
class Logger
  def format_message(severity, timestamp, progname, msg)
   "#{severity} #{timestamp.to_s(:db)} (#{$$}) #{msg}¥n"
  end
end
-------------------- 出力のためのコード例 ---------------
logger.debug "debug message"
logger.info "info message"
logger.warn "warn message"
logger.error "error message"
logger.fatal "fatal message"
--------------------- 出力例 --------------------------
DEBUG 2009-02-13 20:01:22 (4853) debug message
INFO 2009-02-13 20:01:22 (4853) info message
WARN 2009-02-13 20:01:22 (4853) warn message
ERROR 2009-02-13 20:01:22 (4853) error message
FATAL 2009-02-13 20:01:22 (4853) fatal message
```

図 4-17 ログのフォーマットをカスタマイズ

Rails 固有のログ機能のオフ(ANSI 制御シーケンスの設定)

Rails ではデバッグ用の SQL 文ログを出力する際、図 4-18のように ANSI 制御シーケンスを使って色づけし、視認性を上げている。

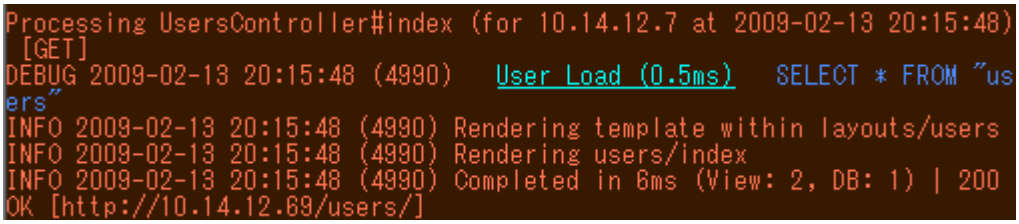


図 4-18 色づけされたログの例

しかし、図 4-19のように ANSI 制御シーケンスに未対応のテキストエディタやターミナルソフトでログファイルを開いた場合は、制御シーケンス部分を解釈することが出来ず視認性や可読性が低下してしまう。また、運用監視ソフトによっては、ログの自動監視に影響を与えてしまうことが懸念される。

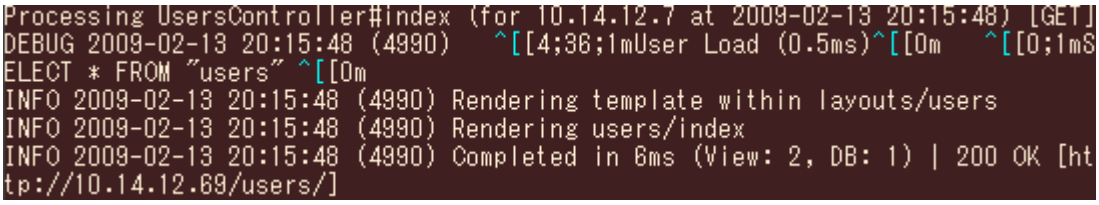


図 4-19 色づけできないエディタで開いた場合

色づけを解除するには config/enviroment.rb に次の設定を記述する。

config.active_record.colorize_logging = false

4.14　運用監視項目

調査目的

アプリケーションの死活監視など、自動監視に関する運用要件に対応できるかどうかを調査する。調査対象は、一般的に基盤と呼ばれる領域で、実行プロセスやリッスンしている TCP ポートなどである。

実装されたアプリケーション固有の内容や、個別の自動監視ツールに特化した項目、サーバマシンのリソース監視などの一般的項目は本調査の対象外としている。

調査内容

サービスの死活確認をするために、監視対象にできる項目を調査した。

調査結果

表 4-6の項目を監視対象とすることで、サービスの自動監視をすることが出来る。ただし、表 4-6にあげた項目は例であり、実運用する場合は運用要件に従ってこれら以外の項目も個別に検討される必要がある。

表 4-6 監視対象

監視対象	監視方法	検知できる事
ruby プロセス	プロセスの有無	サービスの停止
ruby プロセスの仮想メモリサイズ	閾値を設定して監視	メモリリーク
ruby リッスンポート	定期的に HTTP リクエストを送信し、HTTP レスポンスが正常であるかを監視	サービスの停止
ログファイル	FATAL レベルのエラー	DB の停止や想定外のエラーによるサービス停止。プログラムの不具合

4.15　SQL 文を明示的に発行する

調査目的

ここでは、O/R ラッパーの存在により、自由に SQL クエリを発行することができなくなっていないか、SQL クエリを自由に発行することにより生産性が著しく低下することがないかを調査する。

Rails では ActiveRecord と呼ばれる O/R ラッパーにより、よく使われるであろう SQL 文はメソッドなどで巧妙にラッピングされている。しかし、ActiveRecord の表現能力をこえたクエリを DB に投げたい場合は、プログラマにより SQL 文を書く必要がある。

調査内容

SQL 構文を設定できる次のメソッドを調査した。調査対象とした各メソッド名の役割については、次の調査結果にて後述する。

- ActiveRecord::Base.connection.execute()メソッド
- モデル名.find_by_sql()メソッド

調査結果

自由な SQL 文を発行することができ、著しい生産性が劣化したりするなどの弊害は見られなかった。ただし、find_by_sql()メソッド等では SQL 文によりモデルオブジェクトが保持するデータなどに変化が見られるため、あらかじめその挙動について把握しておく必要があることも分かった。

なお、本調査で対象としたメソッド以外にも SQL 文を部分的に挿入できるメソッドは多数存在している。

ActiveRecord::Base.connection.execute()メソッド

ActiveRecord::Base.connection.execute()メソッド(以下 execute メソッド)は、フレームワークが管理している DB コネクションを使って、直接 SQL 構文を実行できる。

DB が解釈できる SQL 文ならば、自由にクエリを投げることが出来る。ただし、execute メソッドを利用した場合の DB からの戻り値は、DB ドライバ固有のオブジェクトがそのまま返ってきており、ActiveRecord の支援を受けることが出来ない。そのため、DB の種類やドライバ固有の処理を、コーディングする必要がある。

モデル名.find_by_sql()メソッド

モデル名.find_by_sql()メソッド(以下、find_by_sql メソッド)は、参照処理(SELECT 文)を直接書くことが出来る。execute メソッドとは違い、slect 文の戻り値は ActiveRecord の支援を受けることができる。

SELECT 文中での列名指定は戻り値に大きな影響を与えるため、テーブルの結合や項目を絞った SELECT 文を find_by_sql メソッドで投げる場合は注意が必要である(図 4-20参照)。

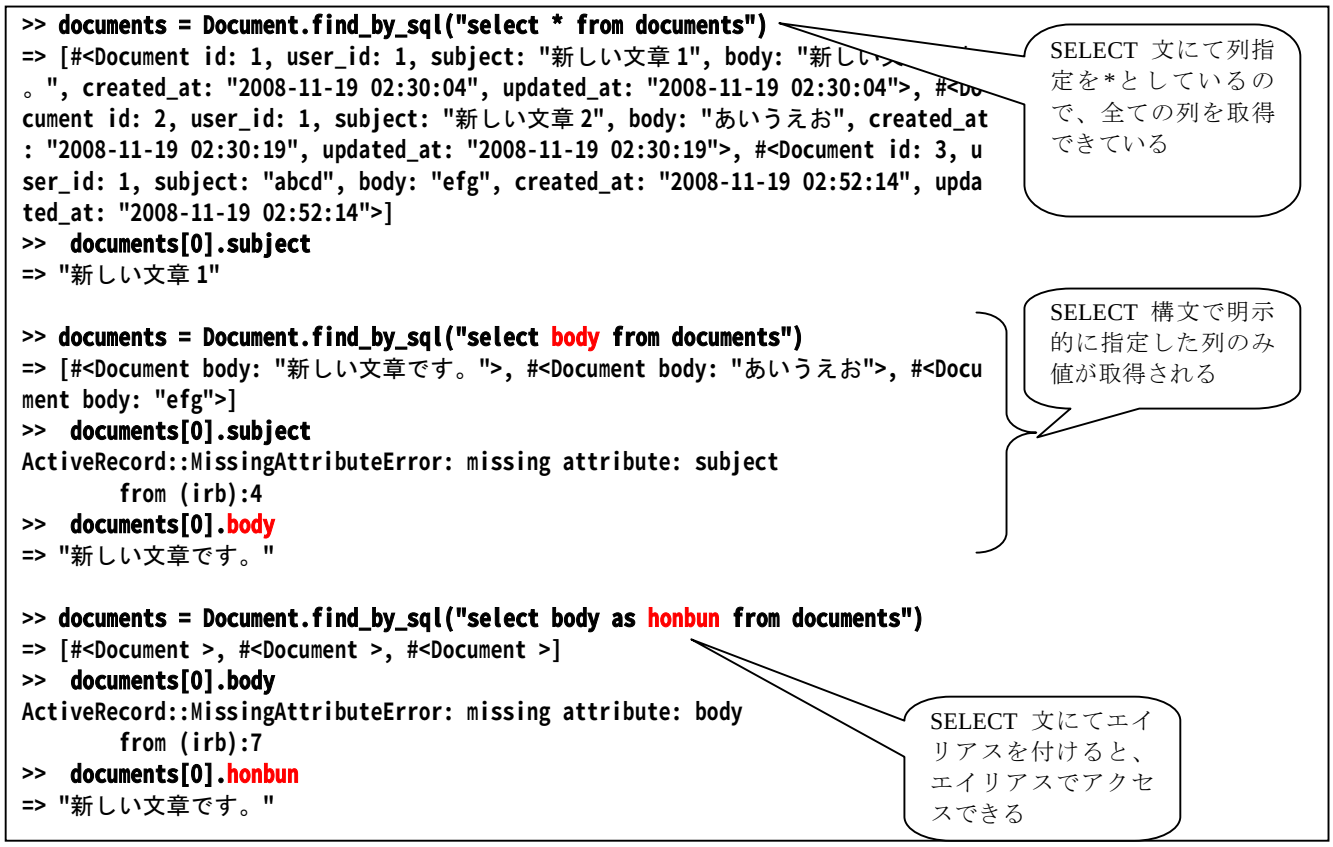


図 4-20 SQL 文を直接書く

4.16　開発に際しての留意事項

情報の収集について

Rails の技術情報について、API についてのドキュメントは存在するが、それらを利用してどのようにしてアプリケーションをコーディングすればよいかという公式のドキュメント[7]は全て英語であり、英語に不慣れな技術者は習得が容易ではない。日本語で技術情報を検索したとしても、記述内容が古いなどして継続的に Rails をウォッチしていない初心者などは混乱する。

Railsの進化速度が速いために書籍が追随できていないという現実もあり、Railsの情報収集については、既存の技術ほどには充実していない。

しかしながら、国内コミュニティによる勉強会の開催や各技術者の努力により日本語化された情報も分散しているとはいえ充実しつつあるため、技術者の努力次第にではあるが情報の収集状況も改善しつつある。

[7] RailsGuides http://guides.rubyonrails.org/

名前の衝突

同一レベルの名前空間内で、モジュールとクラスに同じ名前をつけることはできない。例えば、以下のプログラムでは、class M の定義が「TypeError： M is not a class」のエラーになる。

```
1: module M
2:       .//各種実装
3: end
4: class M
5:       //各種実装
6: end
```

ユーザプログラムで上述のプログラムを作成することはないし、仮にあったとしてもテスト時にエラーを発見できるため取り除くことは容易だ。

ところが、Rails の場合、開発の後半（システムテスト以降）にならないと名称が衝突してしまうことに気付かないケースがある。それは以下3つの機能／状況が重なって引き起こされる。

1. Rails のユーザアプリケーションコードは、必要になったときに自動的にロードされる（オートロード機能）。
2. Rails のユーザアプリケーションは、Mongrel、WEBRick 等のアプリケーションサーバ上のプロセス空間にロードされ実行されるため、アプリケーションサーバやアプリケーションサーバが使用しているパッケージと名称空間を共有する。
3. アプリケーションサーバやパッケージは、他のパッケージやアプリケーションと名称が衝突するのを避けるため、module を使って自身の名称空間を作成する。

以下のサンプルコードを例に、1．の“ユーザアプリケーションコードが必要になったとき”の状況をみてみる。3 行目のコードが実行されたときがそれにあたる。これはユーザアプリケーションのモデルである Service クラスを使用してデータベースから全てのレコードを取り出す処理である。

```
1:class ServicesController < ApplicationController
2:   def index
3:      @services = Service.find(:all)
4:      respond_to do |format|
5:         format.html # index.html.erb
6:         format.xml   { render :xml => @services }
7:      end
8:   end
9:end
```

3 行目が実行されたところで Service という名前の“**実体**”は存在しない。Rails の規約に従い“Service”という名前から service.rb ファイルを検索してロードし、Service モデルの find メソッドが実行される。

もし、このコントローラを実行しているアプリケーションサーバが使用しているパッケージに名称空間として“module Service”を持つものがあったらどうなるだろうか？

同一レベルの名称空間に module と class に同じ名前は付けられないのだから、「Service という名前の**“モジュール”**が存在するため、オートロード機能によるモデルクラスの動的ロードは働かず Service モジュールの find メソッドを実行しようとする」である。

結果、find というメソッドが見つからないというエラーになるか、引数エラーになるか、意図しない戻り値が返ってくるかは、Service モジュールの実装次第である。そして、**パッケージの名称空間とモデルクラス名が衝突している根本原因を示すエラーは示されない。**

【事例】

作成したアプリケーションは使用者に提供するサービスメニューをいくつか持っており、そのサービスを Service というモデルで表現していた。ほぼアプリケーションが完成したのち、ユーザに試用してもらうため本番環境へ移行。本番環境は Windows プラットフォームで、常時サービス提供するため Windows サービスに登録した。サービスへの登録は、mongrel(アプリケーションサーバ)に mongrel_service パッケージを組み込むことで行える。

いざアプリケーションを起動しブラウザからアクセスすると、以下のエラーが出力されプログラムが動かない。

```
undefined method `find' for Service:Module
```

mongrel_service パッケージの名称空間が Service であった！ このため、モデルクラスの Service は名称変更を余儀なくされ、プログラム中に散りばめられたクラス名 Service を全て書き換え、モデルのファイル名の変更、DB のテーブル名の変更を行わなければならなかった。

名前衝突回避のために

名称空間のトップレベルでパッケージのモジュール名（名称空間）と Rails ユーザアプリケーションのモデルクラスのクラス名が衝突するのを検知／回避するための方策を以下に示す。

使用するパッケージは、コーディング着手前に確定し、直接アプリケーションが使用しない（require しない）パッケージであっても開発環境に導入する。

定期的に本番環境で利用する AP サーバ(mongrel など)でテストしておくなど、名前衝突の発生を早い段階で検知するための方策を講じておき、衝突していることが分かったらアプリケーション側の名前を変更するなどの対応を検討する。

Rails2.x から ActionPack のルーティングリソースに namespace の機能が加わっている。モデルやコントロールを module 名（名称空間）付で定義でき、URL にも名称空間を含ませることができる。これにより、先に述べた module 名－class 名の衝突を回避することができる。ただし、URL が変更されるなど根本的な解決策とは言いにくい。

仮に namespace の実装の完成度が高まったとして使用した場合、パッケージの module/class 構成とユーザアプリケーションの module/class 構成が衝突した場合に厄介な問題が発生する。それについては次項「オープンクラスの振舞い」で説明する。

オープンクラスの振舞い

Ruby にはオープンクラスという機能がある。例えば以下のような割り算をしてくれるメソッドクラス定義があったとしよう。

```
class Calculator
  def divide(a,b)
    a/b
  end
end
```

このクラスに対して、掛け算をするメソッドを追加したいとする。その場合は、以下のように Calculator クラス定義をし、掛け算のメソッドを記述する。

```
class Calculator
  def multiply (a,b)
    a*b
  end
end
```

Calclulator クラスは、divide メソッドに加え、新たに multiply メソッドも使える。さらに multiply メソッドは、定義後に作成されたオブジェクトだけでなく、定義前に作成されたオブジェクトでも使うことができる。

interactive ruby を使って一連の動きを見てみよう。

```
irb(main):001:0> class Calculator
irb(main):002:1>   def divide(a,b)
irb(main):003:2>     a/b
irb(main):004:2>   end
irb(main):005:1> end
=> nil
irb(main):006:0> calc = Calculator.new   オブジェクト生成
=> #<Calculator:0x2db6c24>
irb(main):007:0> calc.divide(1,2)
=> 0
irb(main):008:0> calc.divide(4,2)
=> 2
irb(main):009:0> class Calculator
irb(main):010:1>   def multiply(a,b)
irb(main):011:2>     a*b
irb(main):012:2>   end
irb(main):013:1> end
=> nil
irb(main):014:0> new_calc = Calculator.new   multiply メソッド定義後に新たにオブジェクトを生成
=> #<Calculator:0x2da3818>
irb(main):015:0> new_calc.multiply(1,2)
=> 2
irb(main):016:0> new_calc.multiply(2,4)
=> 8
irb(main):017:0> new_calc.divide(4,2)
=> 2
irb(main):018:0> calc.multiply(2,4)   multiply メソッド定義前に生成したオブジェクトも multiply
=> 8                                  メソッドを使用できる。
```

このように、後から機能を追加できるクラスのことを「オープンクラス」という。

オープンクラスでは、前述のようにメソッドを追加するだけでなく、メソッドの書き換えも行える。

例えば、前述の割り算をするメソッドの戻りを商だけでなく余も返すようなメソッドにしたい場合は、以下のように devide メソッドを再定義してやればよい。

```
class Calculator
  def divide(a,b)
    div = a/b
    mod = a%b
    return div, mod
  end
end
```

同様に interactive ruby で動きを見てみる。

```
irb(main):019:0> class Calculator
irb(main):020:1>     def divide(a,b)
irb(main):021:2>         div = a/b
irb(main):022:2>         mod = a%b
irb(main):023:2>         return div, mod
irb(main):024:2>     end
irb(main):025:1> end
=> nil
irb(main):026:0> calc.divide(1,2)
=> [0, 1]
irb(main):027:0> calc.divide(4,2)
=> [2, 0]
irb(main):028:0> new_calc.divide(1,2)
=> [0, 1]
irb(main):029:0> new_calc.diviside(4,2)
=> [2, 0]
```

簡単に「オープンクラス」について説明した。この機能を使って Rails は Web アプリケーションの開発言語として使い勝手の良いように Ruby の機能を拡張しており、Ruby の柔軟性、拡張性の高さを示す機能である。

オープンクラスの仕様上の注意

オープンクラスは意図して使えば柔軟性や拡張性を享受できることは想像できると思う。が、「意図せずオープンクラス機能が使われてしまう」、「意図的にオープンクラスを使った場合でも、複数のライブラリが既存の同じクラスを変更している」など、発見しにくいバグが埋め込まれる可能性もある。

Ruby 作者のまつもとゆきひろ氏は、このデメリットを認識しているが柔軟性／拡張性によるメリットの方が大きいと判断しているようだ。そして、オープンクラスを正しく扱うために以下の三つのルールに従えと言っている。[8]

機能追加に使う

メソッドを追加しても既存プログラムに影響を与えないよう、なるべく機能追加にしておく。そして追加するメソッド名は重複しないように慎重に選ぶ。

機能変更は慎重に

既存メソッドを置き換える場合でも、メソッドのオプション引数を増やし、ある特定の状況下だけ挙動を変更

[8] ITpro 「まつもと直伝プログラミングのおきて　第 21 回オープンクラスと Ruby on Rails」
http://itpro.nikkeibp.co.jp/article/COLUMN/20080619/308756/?ST=oss&P=6

するなど互換性に配慮する。

相互作用に注意

オープンクラスは相互作用を受け易い機能。乱用は禁物。追加したメソッドの名前が衝突した場合、両方を利用しつつ矛盾を解決する手段が無い。

これらは、意図的にオープンクラスを利用する場合のルールである。意図せず使ってしまわないようにするには、これらについて留意しておくことが必要である。

1）利用しているライブラリやアプリケーションサーバを熟知し、名称があたらないように慎重に名称を選ぶ

2）ある程度の人数で開発を行う場合は、まずはネーミングルールを取り決めておくことを忘れずに行う

3）クラスをオープンせずに委譲を検討する

5. 検証総括

5.1 検証結果

実機検証によって得られたアプリケーション開発に関する知見

実機検証のために簡単なアプリケーションを作成したことを元に簡単ではあるが知見を得ることが出来た。

Rails には Web アプリケーションを作成するための機能はほとんどそろっている。Web ブラウザからのリクエストを受け付けてから、DB に対する操作、画面の描画にいたるまで、コーディング量が最小限で済むような工夫が随所でなされており、フレームワークとしての完成度は高い。

Rails には Rails の作法があり、それらに従わない場合は、本来不要であるはずのコードを書いてしまったり、フレームワークが持つ支援を受けることが出来なくなってしまったりしてしまう。どのようなフレームワークを使っていても言えることではあるが、利用する技術の仕様や特性や癖などをしっかり把握して設計をしなければ、Rails を利用したとしても生産性は落ちてしまう。

プログラマ支援

例として DB アクセスの支援機能を紹介する。テーブルのスキーマ定義などは、当然のことながらアプリケーション毎に異なる。例えばテーブル名や列名、列の型など、それぞれについてフレームワークの中で「規約」が設定されており、「アプリケーションにより違う部分だが違うことが普通」である部分のコーディングや設定が極力不要になるよう工夫されている。「規約」に従ったテーブル定義ならば、設定は不要であるか少しの設定で Rails の支援を受けることが出来る。仮に SQL 文が必要になったとしても、DB に格納されているテーブルの定義情報を自動的に読み取り、オブジェクトに値を適切にマッピングする。例として表 5-1をあげる。

表 5-1 DB のテーブル定義と Rails でのアクセス

項目	名称	Ruby 型
テーブル名	USERS	―
モデルクラス名	User	―
検索結果に対応する複数レコード	@users	User の配列
1 レコードのオブジェクト	@user	User
1 レコードの項目を参照	@user.name	規約でマッピングされた型。USRES テーブルの NAME 列の型が Text 型なら String などにマッピングされる。
プライマリー列が 1 のレコードを検索	User.find(1)	User
NAME 列が hoge のレコードを検索	User.find_by_name(‘ hoge ’)	User の配列
NAME 列が hoge ではない全てのレコードを検索	User.find(:all, :conditions => [‘ name <> ? ’ , ‘ hoge ’])	User の配列
画面から入力されたデータ	params[:user]	ハッシュオブジェクト
画面から入力された項目別のデータ	params[:user][:name]	String 型

開発環境

Ruby や Rails は UNIX 文化圏の中で生まれたため、CUI による Rails アプリケーションの開発が一般的であった。しかし、近年は NetBeans などに代表される GUI による統合開発環境(IDE)も整備されつつある。現在の Web アプリケーションの開発現場では IDE での開発が一般的であるため、開発者によっては CUI に慣れておらず IDE が必要な場面も多く見られる。そのため、IDE の存在は重要である。今回の検証では、検証対象として NetBeans を選択し、開発環境としての検証を実施した。

検証の結果、デバッガとしての基本的な機能やコーディング時における補完機能など IDE に求められるであろう一般的な機能は他の開発言語に用意されている IDE 並に充実していることが分かった。CUI が良いか GUI が良いかは開発者により意見が分かれるところではあるが、どちらの選択肢も実用上問題がないために、開発現場や技術者の好みにあわせて開発環境のあり方を決定すればよい。

運用

Rails のデフォルトのログ機能について、いくつかの問題点が見つかった。Rails のデフォルトのログフォーマットには、出力日時やログレベルが併記されず、ログの出た時刻やログの深刻度がわからない。この点については、「4.13 出力ログ(P.50)」にて解決策を提案している。

また、ログが肥大化することを防止するために、多くのシステムではログのローテーションが運用設計として組み込まれるが、Rails はこれについてもデフォルトのロガーでは対応しておらず、同様に対応が必要でもある。

基盤

拡張性・処理性能・安定性

システムの拡張性や処理性能、安定性は、当然のことながらアプリケーションの基盤的設計に大きく左右されることである。しかし、基盤的設計をする前提として Rails の基盤的な仕様や、基盤的設計に対する制約について把握しておく必要がある。

Ruby のリファレンス実装がインタプリタであることから Rails の CPU 利用率は負荷が高いと考えられる。この不利を補うためには利用者数が増える度にサーバの増設が必要となる。Web アプリケーションサーバの処理性能の向上を図る方法としては、スケールアウトが方式を採用する事が多い。一般的には、Web アプリケーションが採用している HTTP セッションの仕組みが分からないとスケールアウト設計は難しい。HTTP セッションの実装方法についての調査したところ、セッションのストア方法にもよるが、サーバサイドにセッション情報を Cookie によって HTTP セッションが維持される仕組みとなっており、負荷分散装置などの支援があればスケールアウトは可能であることが分かった。

Rails のマルチスレッドの問題

Rails はバージョン 2.1 までマルチスレッドは非対応であった。ひとつの OS 上での Rails のスケールアウトは、Ruby プロセスを多重起動し、HTTP リクエストを受ける Web サーバプロセス(例えば Apache HTTP Server など)で負荷分散をするなどして対応してきた歴史がある。また、Ruby 1.8 系列のリファレンス実装におけるスレッドは OS が提供するスレッド機能を利用していない(グリーン・スレッドと呼ばれる)ために、CPU が複数搭載された

マシンの効用を得ることが難しい。

また、Rails 2.2 の正式版リリースは 2008 年 11 月であり、スレッド対応となって日が浅い(本報告書の執筆は 2009 年 2 月)。同期問題などが枯れておらず、スレッド間通信による追求が困難な問題が潜んでいる可能性が高い。

Ruby 1.9 系列のリファレンス実装では OS 提供のスレッドを利用できるようになるため、これまでに挙げた問題は時間と共にコミュニティの努力により解決していくと考えられる。企業活動の一環として Rails を利用する立場としても、積極的にこういった問題解決のための情報を公開していくことが求められる。

2009 年 2 月現在では従来通り、ひとつの OS 上でのスケールアウトは、プロセスの多重起動で対応することを検討した方がよいと考えられる。

SQL Server 2005 を選択することについて

SQL Server 2005 への接続のためのコンポーネントの開発は 2009 年 2 月現在止まっている。実案件において SQL Server 2005 の採用は、システムの寿命やメンテナンスなどを考慮して慎重に選択すべきである。

今後ビジネスを広げていくならば、ドライバの開発も検討するべきである。詳しくは、「4.7 SQL Server 2005 への接続(P.39)」に記載している。

Cookie 管理の HTTP セッション

HTTP セッションの管理方法については「4.3 HTTP セッション管理(P.30)」に詳述しているが、ここで再度、概要ではあるが記述する。

Rails 2.0 以降、HTTP セッションの格納場所のデフォルトはファイルシステムから Cookie に変更された。

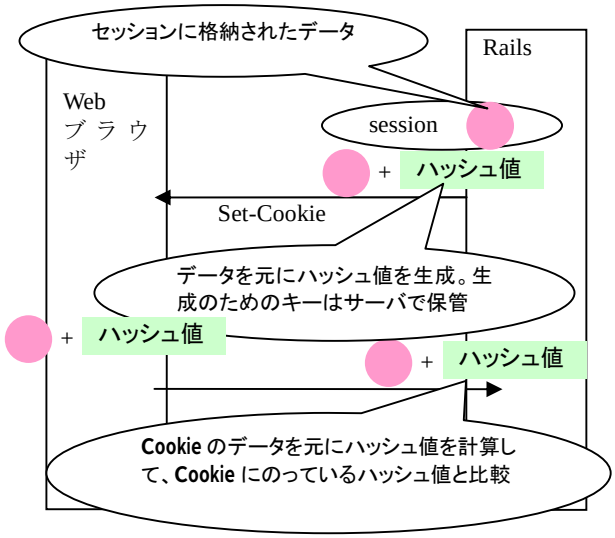


図 5-1 Cookie に格納されたセッション

HTTP セッションの格納先を Cookie として選択すると、セッションオブジェクトに格納されたオブジェクトがマーシャリング(シリアライズ)されて、Cookie に書き込まれる。その時、マーシャリングされたデータと共にデータとサーバサイドで設定されたキーをもとにハッシュ値が生成され、データと同じ Cookie に文字列結合されて格納される。Rails は Web ブラウザから送られたデータが改ざんされていないかを、Cookie のデータを元にハッシュ値を再度生成して比較することで改ざんをチェックする(図 5-1参照)。

ハッシュのキーは OpenSSL の SHA1 実装で生成されており、サーバのキーが漏洩しない限りは騙ることは難しいと思われる。しかし、気をつけなければいけないことは、セッションに格納されたデータそのものは暗号化や難読化といった処理は施されていないことである。外部に公開してはいけない情報をセッションオブジェクトに格納してしまった場合、意図せずして情報が漏洩してしまうことになる。

Java や.NET プログラマにとってこの Rails のデフォルトの HTTP セッションの管理方法は盲点であると思われる。注意されたい。

なお、HTTP セッションの格納先をファイルなどに変更すればこういった懸念は不要である。

Rails の規約と大規模システム構築における標準化による生産性

ここでの大規模とは「アプリケーションの複雑度」が大規模という意味である。

Rails の恩恵を受けるためには「Rails の規約に従う」という前提がある。開発担当者全員が Rails の規約や癖などを把握していれば問題がないが、作業を多人数で分担しなければ手に負えないほどの大規模なシステムになったとしてとしても、Rails 開発に必要なスキルセットを備えた要員をそろえなければならない。一般的な大規模システム開発では、各層(DB スキーマ, DB アクセス層、ロジック層など)で水平分業(図 5-2)が検討されたり、層毎に簡略化された標準化が作成されたりするなどして人数の増加に対処できる。

反面、Rails の規約は垂直方向にも存在するために水平分業や簡略化された標準化などを検討するのは難しい。しかしながら、垂直方向に設計することで層の間の意思の疎通の齟齬や機能の一貫性をもたせることができる。機能×水平の層の数だけ必要だった人数よりも少ない人数での開発も可能である可能性があるが、Rails の利点がどの程度の規模まで生きるのかは、不明であり経験を積む必要がある。

	機能 A	機能 B	機能 C	
画面設計				
ビジネスロジック				
共通部品				
DB スキーマ				
インフラ基盤設計				

図 5-2 水平と垂直

Rails は簡単ではない

開発要員が Rails を使って開発できるまでの期間は長く、負荷も小さくない。今回の調査においても、あるメンバは 1 ヶ月程度「学習」したが、いざコーディングとなったときにジェネレータによるコード生成後に手が止まってしまった。これは個人の学習というよりも、Rails の高生産性だけが取り上げられた結果、書籍でもジェネレータまでは詳しいが、その後の解説が十分ではない可能性がある。書籍の選定について気を使うことはもちろんのこと、規約に従った開発をできるようになる為には、メンバの教育も大切である。

Rails の高生産性は「慣れれば」という前振りが付くことを忘れてはいけない。

パフォーマンスと生産性

Ruby はインタプリタとして実装されているために、パフォーマンス面では負荷が高い傾向にあると言われている。ムーアの法則のとおり CPU の処理性能は年々向上している。反面、性能に必要な費用は年々下がっている。システムの利用者が増加するなどしてパフォーマンスが劣化した場合は、サーバを追加することで対処すればよい。

Rails の生産性とサーバの処理性能を得るためのサーバコストを天秤にかけた場合、Rails に軍配があがることの方が多いのではないだろうか。Rails の生産性を捨て去るほどサーバコストはかからないはずである。

ただし、スケールアウト時における処理性能の向上を確認するための実証や、パフォーマンスチューニングは今回検証できなかった為、追加の検証が必要と考える。

総括

本検証では、入門書籍に書かれているようなコーディング中心の Rails のアプリケーション開発機能よりも、実際の適用現場で問題となるような事を中心に調査した。言語の特性や Rails のアーキテクチャや歴史の浅さからいくつかの懸念点や問題点も明らかになったが、いずれも改善点や避けるべき道筋の提示、解決の提案をすることが出来た。

検証作業を通して浮かび上がったのは、徹底した「DRY」と「設計より規約」[9]の設計思想が貫かれた世界であった。本報告書では「規約」の例として分かりやすい DB スキーマやモデルクラスでの例を挙げたが、その他の機能でも「DRY」や「設定より規約」がかいま見える。そいった設計思想を背景に持つ Rails によって、様々な場面でアプリケーション開発者をアシストしてくれるため、Rails に慣れた場合の生産性は他のフレームワークと比べても飛躍的に高いと感じた。また、「DRY」や「設定より規約」の設計思想で作られた Rails のフレームワークを適正に利用するだけで、アプリケーションの構造にムリやムダ、ムラが生じることを押さえられるようなアーキテクチャであると感じた。

これまでに検証の中で指摘することが出来た留意事項や制約事項を考慮することで、Rails は情報システムを構築するための道具として活用することが出来るといえる。

Rails は機能が強化された 3.0 が近々リリースされる。2.3 からのメジャーバージョンアップである。Rubyもまた、処理性能の大幅な向上を実現し、言語仕様の不備を解決した 1.9 が 2009 年 1 月にリリースされたばかりである。現在のところ Ruby 言語は Rails があってこその勢いではあるが、その他に目を向けてみると Rails 以外にも大規模分散処理の基盤[10]として採用されるなどその応用範囲は広まりつつある。また、Sun Microsystems や Microsoft などの企業によって Java[11]や.NET 基盤[12]に Ruby 処理系が移植されるなど Ruby の実行環境も広まりつつある。

しかしながら、Ruby/Rails を利用した日本国内の受託開発実績はまだまだ少ない。Ruby/Rails の開発実績を積み、事例の情報交換を進めていくことが大切であると考えられる。

[9] 「1.3 Ruby および Ruby on Rails の概要」の「設計思想(P.11)」を参照。
[10] 楽天技術研究所による分散ストレージ ROMA, 分散処理基盤 Fairy プロジェクト。楽天技術研究所にはフェロー研究員としてまつもと氏在籍。
[11] JRuby プロジェクト http://jruby.codehaus.org/
[12] IronRuby プロジェクト http://www.ironruby.net/

6. 本報告書について

6.1 Ruby アプリケーションタスクフォースメンバ

本報告書は下記メンバの協力のもと日本ユニシス株式会社により執筆された。

事務局:大内 一晃　　情報処理推進機構

主査:吉田 正敏　　富士通株式会社

＜メンバ＞(五十音順)

安倍 武志	株式会社グッデイ
伊藤 宣博	キヤノン IT ソリューションズ株式会社
今給黎 道明	キヤノン IT ソリューションズ株式会社
大浦 順史	株式会社富士通ソーシアルサイエンスラボラトリ
大澤 一郎	産業技術総合研究所
木村 守宏	株式会社シーイーシー
篠田 健	日本ユニシス株式会社
鈴木 友峰	株式会社日立製作所
高田 剛	アースインターシステムズ株式会社
多賀野 進	アースインターシステムズ株式会社
筒井 敏人	株式会社富士通ソーシアルサイエンスラボラトリ
藤部 孝志	アースインターシステムズ株式会社
中島 雅彦	株式会社日立製作所
永海 隆俊	株式会社テクノプロジェクト
野山 孝太郎	富士通株式会社
原 嘉彦	株式会社富士通ソーシアルサイエンスラボラトリ
早川 英治	富士通株式会社
藤田 剛	サイオステクノロジー株式会社
藤田 祐治	レッドハット株式会社
前田 青也	株式会社グッデイ
松葉 大造	株式会社シーイーシー

宮本 利明	シーネットネットワークスジャパン株式会社
室脇 俊二	株式会社テクノプロジェクト
森 孝博	富士通株式会社
山﨑 靖之	サイオステクノロジー株式会社
吉野 良成	日本ユニシス株式会社

※所属は 2009 年 4 月当時

6.2 著作権

6.3 商標について

A. 付録

A.1. 規約に従っていない DB スキーマを採用した場合の制限の検証

検証のために作成したテーブル定義

次のような書籍の貸し出しシステムを想定したテーブル定義により検証した。

- 書籍マスターテーブル:書籍の在庫を表すテーブル。

BOOK_MASTER		
isbn	char(13)	PK
title	varchar(500)	
author	varchar(500)	
stock	integer	
note	varchar(1024)	

- 貸し出し名簿テーブル:貸し出し者リストを表す。1 対多を検証するためのテーブル。

LEND_LIST		
lend_id	char(10)	PK
isbn	char(13)	BOOK_MASTER テーブルを参照
user_name	varchar(20)	

- タグマスターテーブル:書籍に付加される分類などの情報を表すタグのマスターテーブル。

TAG_MASTER		
tag_id	int	PK
tag_name	varchar(20)	

- タグテーブル:書籍に付加されたタグを表す。単純な多対多を検証するための結合テーブル。

BOOK_TAG_JOIN_TABLE		
isbn	char(13)	BOOK_MASTER テーブルを参照
tag_id	int	TAG_MASTER テーブルを参照

- タグテーブル: 書籍に付加されたタグを表す。タグにコメントを付記できる。複雑な多対多を検証するためのテーブル。

BOOK_TAG_COMMENT		
book_tag_comment_id	int	PK
isbn	char(13)	BOOK_MASTER テーブルを参照
tag_id	int	TAG_MASTER テーブルを参照
comment	varchar(1024)	

検証のためにモデルクラスに利用した設定

規約に反したテーブルを ActiveRecord で使うために、次の設定を施した。

1）主キーの名前を「id」以外とする

モデルクラスの定義に対して set_primary_key クラスメソッドを追記することにより設定した。BOOK_MASTER テーブルの場合では次のような一行である。

```
set_primary_key 'isbn'
```

2）主キーの型を文字列型(CHAR)とする

モデルクラスへの設定は特に無い。

3）テーブル名をモデルクラスの複数形としない

モデルクラスの定義に対して、set_table_name クラスメソッドを追記することにより設定した。BOOK_MASTER テーブルの場合、モデルクラス名は「BookMaster」として、次のような一行を追加した

```
set_table_name  'book_master'
```

4）外部参照列名を「参照先テーブル名_id」としない

【1 対多の場合】

検証では図 6-1のような 1 対多の関係を BOOK_MASTER テーブルと LEND_LIST テーブルに持たせた。

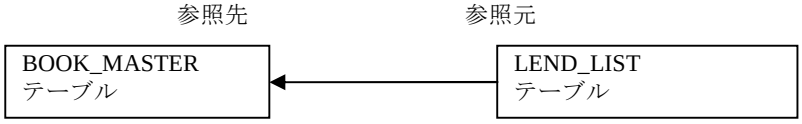


図 6-1 1 対多

LEND_LIST テーブルを表す LendList クラスの定義中に対して、belongs_to クラスメソッドを記載する。belongs_to メソッドには:class_name パラメータと:foreign_key パラメータを設定した。:class_name パラメータには参照先モデルクラス(BookMaster)を設定し、: foreign_keyパラメータには参照先モデルの主キー名を設定した。

```
belongs_to :book,
           :class_name  => 'BookMaster',
           :foreign_key => 'isbn'
```

BOOK_MASTER テーブルを表す BookMaster クラスの定義中に、has_many クラスメソッドを記載した。has_many メソッドには:class_name パラメータ、:foreign_key パラメータを設定した。:class_name パラメータには参照元モデルクラス(LendList)を設定し、:foreign_key パラメータには参照先モデルの主キー名を設定した。

```
has_many :lend_list,
         :class_name => 'LendList',
         :foreign_key => 'isbn'
```

【多対多 – habtm を使った方法】

多対多の結合は、参照されるテーブル間に結合テーブルを挟むことで実現される。ActiveRecord では多対多を表現するために二つの手段が提供されている。

一つは外部参照列の組のみが存在する単純な結合テーブルを利用する方法であり、has_and_belongs_to_many(長いので habtm と略称される)というクラスメソッドを利用する(図 6-2)。

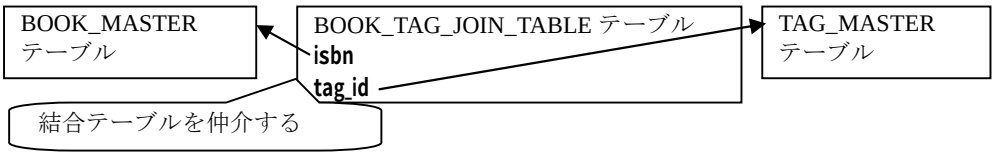


図 6-2 has_one_belongs_to_many を使った多対多

モデルクラスには次のような多対多の設定を記載した。

- BookMaster クラス(BOOK_MASTER テーブルを表す)

```
has_and_belongs_to_many :tags,
                        :class_name => 'TagMaster',
                        :join_table => 'book_tag_join_table',
                        :foreign_key => 'isbn',
                        :association_foreign_key => 'tag_id'
```

- TagMaster クラス(TAG_MASTER テーブルを表す)

```
has_and_belongs_to_many :books,
                        :class_name => 'BookMaster',
                        :join_table => 'book_tag_join_table',
                        :foreign_key => 'tag_id',
                        :association_foreign_key => 'isbn'
```

habtm メソッドに対して、:class_name, :join_table, :foreign_key, :association_foreign_key パラメータを設定することにより、規約に反したテーブル定義でも ActiveRecord を使えるよう設定した。

:class_name は参照先のモデルクラス名、:join_table は結合テーブル名、:foreign_key は結合テーブル内の自分自身を示す列名、:association_foreign_key は結合テーブル内の参照先テーブルを示す列名を記載する。

【多対多 －:through を使った方法】

結合テーブルに対して、柔軟に情報を追加できる方法である。これは、相互に参照し合う二つのモデルクラスに対して指定される **has_many** と結合テーブルの **belongs_to** に対して、**:through** パラメータを指定することにより多対多の関係をモデルクラスで実現する方法である。

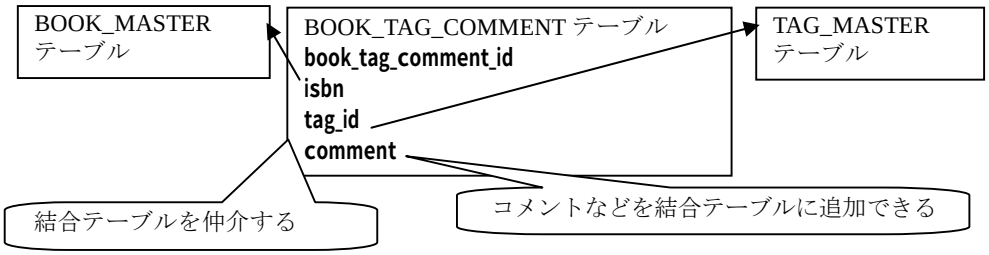


図 6-3 :through パラメータを用いた多対多

モデルクラスには次のような設定で多対多の設定を記載した。

- BookMaster クラス(BOOK_MASTER テーブルを表す)

```
has_many :tag_comments,
              :class_name => 'BookTagComment',
              :foreign_key => 'isbn'

has_many :commented_tags,
              :class_name => 'TagMaster',
              :through => 'tag_comments',
              :source => 'tag'
```

- TagMaster クラス(TAG_MASTER テーブルを表す)

```
has_many :book_comments,
            :class_name => 'BookTagComment',
            :foreign_key => 'tag_id'

has_many :commented_books,
              :class_name => 'BookMaster',
              :through => 'book_comments',
              :source => 'book'
```

- BookTagComment クラス(BOOK_TAG_COMMENT テーブルを表す)

```
belongs_to :book,
                :class_name => 'BookMaster',
                :foreign_key => 'isbn'

belongs_to :tag,
                :class_name => 'TagMaster',
                :foreign_key => 'tag_id'
```

:through を使った多対多の表現は、実際には多対 1 対多というもので、間の結合テーブルを仲介して参照先にアクセスする。:through パラメータと:source パラメータを利用することにより、規約に反したテーブル定義を利用できるようにしている。

A.2. Ruby on Rails から SQL Server2005 への接続設定手順

Windows にて駆動する Rails から SQL Server への接続は、Rails の sqlserver-adapter が Ruby DBI 層を利用し、Ruby DBI の中で Win32OLE を使った ADO 接続をしている。

SQL Server 用のアダプタの設定手順を本項にて述べる。なお、手順は下記公式 wiki に掲載されている。

HowtoConnectToMicrosoftSQLServer

http://wiki.rubyonRails.org/Rails/pages/HowtoConnectToMicrosoftSQLServer

■ 手順 1: SQL Server 用のアダプタをインストール

```
$ gem install ¥
activerecord-sqlserver-adapter --source=http://gems.rubyonRails.org
```

実際は 1 行

■ 手順 2: Ruby DBI ライブラリをインストール

2008/12 現在最新の 0.4.0 ではうまく接続することができないために、モジュールを下記 URL よりダウンロードして、0.2.2 を手動でインストールする。

http://rubyforge.org/frs/download.php/41304/dbi-0.2.2.zip

以下、展開ディレクトリを C:¥tmp¥dbi-0.2.2 とする。

```
C:¥tmp¥dbi-0.2.2>ruby serup.rb config
C:¥tmp¥dbi-0.2.2>ruby serup.rb setup
C:¥tmp¥dbi-0.2.2>ruby setup.rb install
```

■ 手順 3: ADO にて接続するモジュールを所定のディレクトリに配置する

Ruby インストールディレクトリを C:¥ruby1.8 とする。

C:¥tmp¥dbi-0.2.2¥ lib¥dbd ¥ADO.rb を C:¥ruby1.8¥lib¥ruby¥site_ruby¥1.8¥dbd¥ado にコピーする。

■ 手順 4: database.yml を編集する

次のように編集する。host には実際のホスト名ではなく、SQL Server のインスタンス名を設定する。Rails と同じサーバならば.¥[インスタンス名]となる。

```
development:
  adapter: sqlserver
  database: TestDB
  host: .¥SQLEXPRESS
  username: sa
  password: password
```

■ 手順 5:config/environment.rb を編集する

先頭に次の二行を追記した。

```
require 'win32ole'
WIN32OLE.codepage = WIN32OLE::CP_UTF8
```

A.3. Web サービス検証結果詳説

Rails から Java システムの呼び出し

Java 側は Axis を用いて Web サービス化し検証した。

Rails 側(呼び出し側)は、soap4r という SOAP ライブラリの提供ツールに WSDL を読み込ませることにより、SOAP クライアントを生成し、Java システム側の Web サービスを呼び出した。

Web サービス側の処理概要(Java 側)

参照系のサービスを想定して、検証用の Web サービスを作成した。作成した Web サービスは引数としてユーザのプライマリーキーを受け取り、該当するユーザ情報を返す。検索されない場合は、UserNotFound 例外が返される。

メソッドのシグネチャと戻り値の型の概要を図 6-4に示す。HTTP で Web サービスの呼び出しが行われ、Axis により図 6-4に示された findUser メソッドにバインドされる。findUser メソッドは User クラスのオブジェクトを返し、受けたれたオブジェクトがさらに Axis により SOAP レスポンスとして返される。

```
// Web サービス化されたメソッドのシグネチャ
public static User findUser(int id) throws UserNotFound

// 戻り値の型の概要
public class User {
          // 主キー
          public int getID()
          public void setID(int id)
          // 名
          public String getFirstName()
          public void setFirstName(String firstName)
          // 姓
          public String getLastName()
          public void setLastName(String lastName)
          // 年齢
          public int getAge()
          public void setAge(int age)
          // 最終更新日
          public Date getUpdateAt()
          public void setUpdateAt(Date updateAt)
}
```

図 6-4 Java 側の Web サービス・コード概要

Web サービス呼び出し側の処理概要(Rails 側)

Axis により生成された WSDL を元に soap4r 付属のツールを用いて Ruby による SOAP クライアントを生成した。

まず、WSDL の情報を元に生成された戻り値型を図 6-5に示す。Java での User 型の定義が、そのまま WSDL を通じて Ruby で再現できていることが図 6-5のコードから分かる。

```
class User
  attr_accessor :iD
  attr_accessor :age
  attr_accessor :firstName
  attr_accessor :lastName
  attr_accessor :updateAt

  def initialize(iD = nil, age = nil, firstName = nil, lastName = nil, updateAt = nil)
    @iD = iD
    @age = age
    @firstName = firstName
    @lastName = lastName
    @updateAt = updateAt
  end
end
```

図 6-5 WSDL から生成された Ruby コードの一部

Web サービスの呼び出しに成功すると、図 6-5に示した User 型のオブジェクトが soap4r により返される。返される User 型はそのままモデルとして活用した。Web サービスの呼び出しは、User クラスに find メソッドを追加し、その中で実装した(図 6-6)。

図 6-6中には ails の API ではないメソッドやオブジェクトが利用されているが、それらは soap4r により生成されたコードである。

```
require 'to_java/defaultDriver'

class User
<中略>
  def self.find(id)
    user_info = UserInfo.new('http://java_server/SOAPTest/services/UserInfo')

    parameters = FindUser.new(id)
    begin
      ret = user_info.findUser(parameters) # =>  見つからない場合は例外
      user = ret.findUserReturn
    rescue => e
      raise e
    end
    user
  end
end
```

図 6-6 Web サービスの呼び出し

Java システムから Rails の Web サービスの呼び出し

バージョン 2.0 以前の Rails では、Web サービスを提供するためのモジュールとして ActionWebService(AWS)が存在していた。しかし、Ruby on Rails の設計が RESTful 指向へと方針転換した Ruby on Rails 2.0 以降、SOAP による Web サービスは非標準扱いとなってしまった。そのため、2.2.2 には公式の AWS 実装は存在しない。第三者が 2.0 以降のバージョンでも動作するようにポーティングされた実装があるのみである。

ポーティングされた ActionWebService

http://github.com/datanoise/actionwebservice/tree/master

本検証では、ポーティングされた AWS によって検証を実施した。

Web サービス側の処理概要(Rails 側)

Java の時と同様、参照系のサービスを想定して、検証用の Web サービスを作成した。作成した Web サービスは引数としてユーザのプライマリーキーを受け取り、該当するユーザ情報を返す。

AWS では、ActionWebService::API::Base の継承クラスにて Web サービスのシグネチャを定義する(図 6-7 参照)。

```
# app/controllers/user_info_controller.rb
class UserInfoController < ApplicationController
  web_service_api UserInfoApi
  web_service_scaffold :invocation
  wsdl_service_name 'UserInfo'

  def find_user(id)
    User.find(id)
  end
end

# app/services/user_info_api.rb
class UserInfoApi < ActionWebService::API::Base
  api_method :find_user, :expects => [{:id => :int}], :returns => [User]
end
```

図 6-7 Rails による Web サービス実装

Web サービス呼び出し側の処理概要(Java 側)

AWS により出力された WSDL をもとに、Axis 付属のツールを使って Java 側の Web サービスクライアントを生成した。生成された戻り値型とクライアントのシグネチャを図 6-8に示す。

```
// Rails の Web サービスを呼び出すためのメソッド
public ActionWebService.User findUser(int id) throws java.rmi.RemoteException

// 戻り値の型
public class User   implements java.io.Serializable {
      public int getId()
      public void setId(int id)

      public java.lang.String getFirst_name()
      public void setFirst_name(java.lang.String first_name)

      public java.lang.String getLast_name()
      public void setLast_name(java.lang.String last_name)

      public int getAge()
      public void setAge(int age)

      public java.util.Calendar getCreated_at()
      public void setCreated_at(java.util.Calendar created_at)

      public java.util.Calendar getUpdated_at()
      public void setUpdated_at(java.util.Calendar updated_at)
}
```

図 6-8 Axis により生成された Web サービスクライアント

A.4. 画面で UTF-8, Shift-JIS を使うための設定

UTF-8 を利用するための設定

config/environment.rb の先頭に次の二行を追記することで UTF-8 を利用することが出来た。

```
require 'win32ole'
WIN32OLE.codepage = WIN32OLE::CP_UTF8
```

Shift-JIS を利用するための設定

config/environment.rb 及び、各ビューに次のような修正を加えることで Shift-JIS を利用することが出来た。

config/environment.rb 先頭に次を追記。

```
require 'win32ole'
```

config/environment.rb の config 設定箇所に次を追記。

```
config.action_controller.default_charset="Windows-31J"
```

config/environment.rb 末尾に次を追記。

```
$KCODE="s"
```

layout など HTML ヘッダ部分の文字コード設定箇所を修正。

```
<meta http-equiv="content-type" content="text/html;charset=Windows-31J" />
```