Four Days on Rails

Compiled by John McCreesh

日本語訳 板垣 正敏

(Rails1.1.6 対応版修正 2)

目次

はじめに

これまで Rails について何度も大げさな主張がされてきた。たとえば、OnLAMP.com[1]の記事は「Rails を使うと一般的な Java フレームワークに比較して最低でも 10 倍早く Web アプリケーションが開発できる」と公言している。この記事ではさらに PC に Ruby と Rails をインストールし、実質上 1 行もコードを書くことなく、動く scaffold アプリケーションを生成する方法を示している。

この記事は印象が強烈だが、「本当の」Web アプリ開発者は、これが手品であることを知っている。「本当の」Web アプリケーションはこんなに単純ではない。Rails の舞台裏では何が行われているのだろう。さらに進んで「本当の」アプリケーションを開発するのはどれぐらい大変なのだろうか。

この辺から問題は少しややこしくなる。Rails には充実したオンラインドキュメントが完備している。実際、リファレンスマニュアルの形で 30,000 語を超えるオンラインドキュメントは、初心者には多すぎるくらいだ。欠けているのは、Rails で開発を始め、進めていくために必要なページを示してくれる鉄道路線図である。

このドキュメントは、このギャップを埋めるために用意したものだ。ここでは、PC に Ruby on Rails を設定し、動くところまで実行済みであることを想定している。(まだインストールしていないのであれば、Curt の記事の通りにすること)ここまで終わっていれば、1 日目の終わりまで進んだことになる。

「Rails の 2 日目」は、手品の舞台裏を知ることから始まる。“scaffold”コードを順に見ていることになる。新しい機能は太字で表示され、解説があり、その後に、Rails あるいは Ruby のドキュメントへの参照を示す。それを読むことで理解を深めることができる。
「Rails の 3 日目」は、“scaffold”コードを足場に、「本当の」アプリケーションに見えるようなものを構築する。一日をかけて Rails を使い、あなた自身の工具箱を作り上げていくことになる。何よりも、オンラインドキュメントの参照が楽にできるようになり、自分で答えを見つけることができるようになるだろう。

「Rails の 4 日目」には、もうひとつテーブルを追加し、参照一貫性を維持するという複雑な仕事をこなすことになる。最後には、使えるアプリケーションと、自分で開発を始めるのに十分な道具、そして、更なる情報をどこで得ればよいかという知識が身についているはずだ。

本当に 10 倍速いかって？4 日間 Rails を使った後で、ご自分で判断してほしい。

ドキュメンテーション:このドキュメントにはハイライトされた下記への参照が含まれる:
ドキュメンテーション —Rails のドキュメンテーション http: //api.rubyonrails.com (このドキュメントは、gems でのインストールの際に PC の次の場所にもインストールされる。`C:\Program Files\ruby\lib\ruby\gems\n.n\doc\actionpack-n.n.n\rdoc\index.html`)
Ruby のドキュメンテーション—“Programming Ruby — The Pragmatic Programmers Guide”は次の場所でオンラインで利用可能であり、あるいはダウンロードできる。http://www.ruby-doc.org/docs/ruby-doc-bundle/ProgrammingRuby/index.html
謝辞: IRC チャネル[2]とメーリングリスト[3]上の親切な方々に感謝する。オンラインアーカイブは、私が Rails や Ruby のラーニングカーブを上っていく際に、それらの人々にいかに助けられたかの証である。
バージョン: 2.3 は Rails の 0.12.1 を使用している。最新版と ToDo のプログラムコードをダウンロードするには http://rails.homelinux.org を参照のこと。ドキュメントは OpenOffice.org の Writer で作成し、pdf ファイルを出力した。(訳注:日本語版の改訂にあたり、日本語版独自の Rails 1.1.6 での作動確認と加筆修正を行った)

---

1 *Rolling with Ruby on Rails,* Curt Hibbs 20-Jan2005 http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html
2 irc://irc.freenode.org/rubyonrails
3 http://lists.rubyonrails.org/mailman/listinfo/rails

翻訳について：翻訳およびRails1.1.6対応版改訂については、板垣 正敏 masatoshi@rails.toが全ての責任を負っている。翻訳版についても、*Creative Commons Attribution-NonCommercial-ShareAlike License*が適用される。誤訳等の指摘や感想などについては、上記のメールアドレスに送信いただきたい。

Rails 1日目

“To Do リスト”アプリケーション

このドキュメントでは、簡単な To Do リストアプリケーションの構築をなぞってゆく。このアプリケーションは、PDAにあるように、アイテムのリストが、カテゴリーによりグループ化され、オプションのノートがつけられる。（およその外観は 29 ページの図5「To Do リスト」の画面を参照のこと）

Rails のスクリプトを実行する

これは私のPCでの例である。私の Web 関連ファイルは、c:\www\webroot にある。私は次のようにコマンドを発行して、これを W:ドライブとしてマッピングした。

```
C:\> subst w: c:\www\webroot
C:\>W:
W:\> rails ToDo
W:\> cd ToDo
W:\ToDo>
```

rails ToDo を実行することで新しく ToDo ディレクトリが作成され、その下には一連のファイルとサブディレクトリが生成される。中でも重要なのは下記のディレクトリである。

app
アプリケーションの核になる部分を含む。その中は、model, view, controller, および helper の各サブディレクトリに分かれている。

config
アプリケーションで使用するデータベースの詳細を定義した database.yml ファイルを含む。

log
アプリケーション特有のログを含む。注:development.log は、エラー追跡に便利な Rails でのアクションを逐次記録したトレースを含むが、定期的に削除する必要がある。

public
Apache 用のディレクトリで、images, javascripts, および stylesheets の各サブディレクトリを含む。

アプリケーションを Web サーバに追加する

私は、あらゆるもの（Apache2, MySQL, 他）を 1 台の開発用 PC で動かしているので、次の 2 ステップでアプリケーションを Web ブラウザで開くときにわかりやすい名前をつけている。（訳注:実運用を前提としないのであれば、Rails 付属のスクリプトにより、Webrick で動作確認をしながら開発を進めることができる）

Hosts ファイルでアプリケーションを定義する

C:\winnt\system32\drivers\etc\hosts （抜粋）

```
127.0.0.1    todo
```

Apache の設定ファイルでアプリケーションを定義する

Apache2\conf\httpd.conf

```
<VirtualHost *>
   ServerName todo
   DocumentRoot /www/webroot/ToDo/public

   <Directory /www/webroot/ToDo/public/>
      Options ExecCGI FollowSymLinks
      AllowOverride all
```

```
      Allow from all
      Order allow,deny
   </Directory>
</VirtualHost>
```

Fastcgi への切替

あなたが忍耐強く(あるいは強力な PC をもってい)なければ、このアプリケーションで fastcgi を有効化したほうが良い。

```
public\.htaccess
# パフォーマンスをあげるために、ディスパッチャーを fastcgi 対応版に変更する
 RewriteRule ^(.*)$ dispatch.fcgi [QSA,L]
```

Rails が稼動していることを確認する

これまでの作業で、ブラウザで http://todo/ が参照できるはずである。(Congratulations, you've put Ruby on Rails!というページが参照できるはずだ)(訳注:`ruby script/server` でポート番号のオプションなしに Webrick サーバを起動した場合、`http://localhost:3000/`を参照すること)

Rails のバージョン

あなたがこのドキュメントを読むまでに、Rails のバージョンはいくつか新しくなっているだろう。このドキュメントの手順をなぞりたいのであれば、インストール済みのバージョンを確認すること。(訳注:この日本語改訂版では、Rails 1.1.6 で動作確認を行っているため、下記の設定は不要である)

```
w:\ToDo>gem list --local
```

下記に列挙したバージョンと異なる場合には、このドキュメントで使用されているバージョンをインストールすることを強く推奨する。つまり、

```
W:\ToDo>gem install rails --version 0.12.1
```

これによって何かがおかしくなることは無い。Ruby の gems ライブラリは、複数のバージョンを取り扱えるように設計されている。だから、次のようにして、Rails に対して To Do リストアプリケーションでは、"Four Days"用のバージョンを使うように強制できる。

```
config\environment.rb (抜粋)
# Require Rails libraries.
require 'rubygems'
require_gem 'activesupport', '= 1.0.4'
require_gem 'activerecord', '= 1.10.1'
require_gem 'actionpack', '= 1.8.1'
require_gem 'actionmailer', '= 0.9.1'
require_gem 'actionwebservice', '= 0.7.1'
require_gem 'rails', '= 0.12.1'
```

同じバージョンを使用する理由はまったく単純である。"Four Days"は Rails によって生成されたコードを多用する。Rails の開発が進むにつれて、生成されるコードも困ったことに進化する。このドキュメントは進化しない。(私が新しいバージョンを書かないかぎりは)だから、あなた自身が楽をするためには、"Four Days"と同じバージョンを使用すること。この"Four Days"が終わったら、最新のそして最高の Rails のバージョンを使えばよい。そうすれば、Rails の開発者がどのような改善を行ったかを理解できるだろう。

データベースを設定する

私は MySQL に、"`todos_development`","`todos_test`","`todos_production`"という新しいデータ

ベースを設定した。

```
Your MySQL connection id is 3 to server version: 5.0.24a-community-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database todos_development default character set utf8;
Query OK, 1 row affected (0.02 sec)

mysql> create database todos_test default character set utf8;
Query OK, 1 row affected (0.07 sec)

mysql> create database todos_production default character set utf8;
Query OK, 1 row affected (0.01 sec)

mysql> grant all on todos_development.* to foo@localhost identified by 'bar';
Query OK, 0 rows affected (0.09 sec)

mysql> grant all on todos_test.* to foo@localhost;
Query OK, 0 rows affected (0.00 sec)

mysql> grant all on todos_production.* to foo@localhost;
Query OK, 0 rows affected (0.00 sec)

mysql> \q
```

このデータベースへの接続は config\database.yml ファイルに指定する。

config\database.yml (抜粋)

```
development:
  adapter: mysql
  database: todos_development
  host: localhost
  username: foo
  password: bar
test:
  adapter: mysql
  database: todos_test
  host: localhost
  username: foo
  password: bar
production:
  adapter: mysql
  database: todos_production
  host: localhost
  username: foo
  password: bar
```

マイグレーションによりカテゴリテーブルを作成する

Rails1.0 以降は、マイグレーション機能によって、SQL スクリプトを書かずにテーブルの定義が可能である。この日本語改訂版では、この機能を使用してカテゴリテーブルを定義する。categories テーブルは To Do リスト中のアイテムをグループ分けする際に使われる 単純なカテゴリのリストである。

モデルの生成

データモデルを生成すると、自動的にマイグレーションファイルのテンプレートも生成される。モデル名は 1 文字目が大文字で、単数形を指定する。Rails がテーブル名を生成する際には、小文字の複数形に変換されることに注目しよう。

```
W:\ToDo>ruby script/generate model Category
      exists  app/models/
      exists  test/unit/
```

```
      exists  test/fixtures/
      create  app/models/category.rb
      create  test/unit/category_test.rb
      create  test/fixtures/categories.yml
      create  db/migrate
      create  db/migrate/001_create_categories.rb
w:\ToDo>
```

参考までに、Rails におけるデータベースの命名規則などに関する規約とヒントを示す。

テーブルおよび列の命名に関するヒントとポイント:

- 列名に含まれるアンダースコアは Rails が人に読みやすい名前を生成する際には取り除かれる。
- 列名に大文字小文字を混ぜて使用するときは注意すること。Rails の一部は大文字小文字を区別する。
- 全てのテーブルには、'id'という名前の主キーがあること。MySQL では、numeric auto_increment を使えば簡単に実現できる。
- 他のテーブルへのリンクは同様に'_id'という命名規則に従うべきである。
- Rails は created_at / created_on および updated_at / updated_on という名前の列の更新を自動的に行う。したがって、これらの列を追加するのはいい考えである。

Documentation: ActiveRecord::Timestamp

- 有用なヒント:マルチユーザーシステムを構築しているのであれば(いまは関係ないが)、もしテーブルに lock_version (integer default 0)という列があると、Rails は自動的に楽観的ロックを実装してくれる。しなければならないのは、更新フォームに rock_version を隠しフィールドとして含めることだけである。

Documentation: ActiveRecord::Locking

マイグレーションファイルの編集

前記のモデルとともに生成されたマイグレーションファイル db/migrate/001_create_categories.rb を編集する。

db/migrate/001_create_categories.rb

```
class CreateCategories < ActiveRecord::Migration
  def self.up
    create_table :categories  do |t|
      t.column :category, :string, :null => false
      t.column :created_on, :datetime, :null => false
      t.column :updated_on, :datetime, :null => false
    end
  end

  def self.down
    drop_table :categories
  end
end
```

Documentation: ActiveRecord::Migration

マイグレーションを実行しテーブルを生成する

rake によりマイグレーションタスクを実行し、テーブルを生成する。

```
W:\ToDo>rake db:migrate
(in W:/ToDo)
== CreateCategories: migrating ================================================
-- create_table(:categories)
  -> 0.1700s
== CreateCategories: migrated (0.1700s) =======================================


W:\ToDo>
```

Scaffold

コントローラは Rails アプリケーションの心臓部である。

コントローラ生成スクリプトを実行する

```
W:\ToDo>ruby script/generate controller category
exists app/controllers/
exists app/helpers/
create app/views/category
exists test/functional/
create app/controllers/category_controller.rb
create test/functional/category_controller_test.rb
create app/helpers/category_helper.rb
W:\ToDo>
```

これにより、2 つのファイルと 1 つの空のディレクトリが生成される。

app¥controllers¥category_controller.rb
app¥helpers¥category_helper.rb
app¥views¥categories
app¥views¥layouts

もし、Rolling with Ruby on Rails のような初心者向けチュートリアルにある、model/scaffold のトリックをまだ見たことが無ければ、ここで試すと良い。たった一行だけで、Web アプリケーション全体ができるのは驚きだ。

app\controllers\category_controller.rb

```
class CategoryController < ApplicationController
   scaffold :category
End
```

Documentation: ActionController::Scaffolding::ClassMethods

ブラウザで、http://todo/category を参照すれば、Rails がいかに利口かびっくりするだろう。:-)

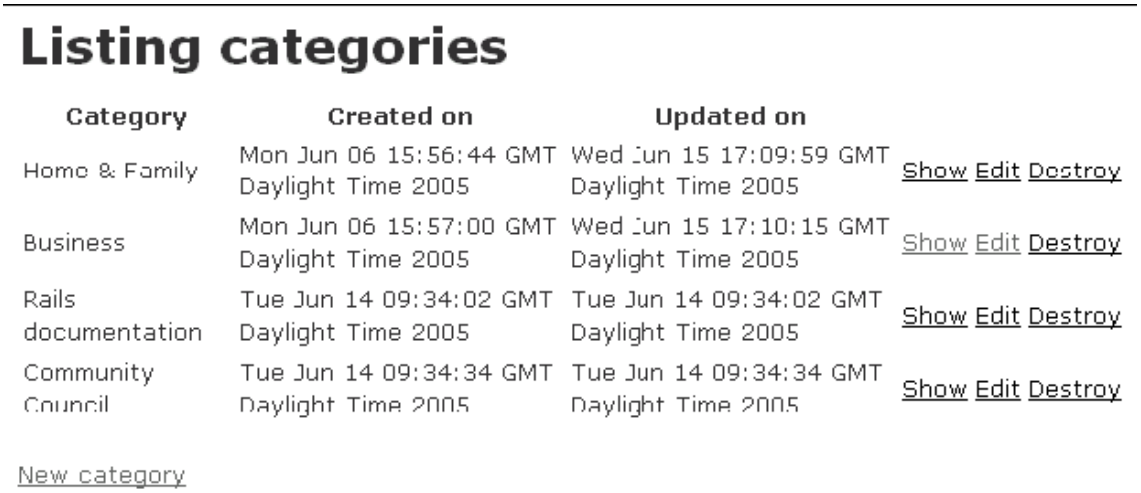

図1 Scaffold‘List’画面

Rails がそれほど利口でないことを理解するため、まったく同じカテゴリを 2 度追加してみよう。Rails は“ActiveRecord::StatementInvalid in Category#create”というわけのわからないメッセージを吐いて終了してしまう。これは、モデルに検証機能を追加することで直すことができる。

モデルを拡張する

モデルには、データに関係するルールが保存される。その中にはデータの検証や参照整合性が含まれる。つまり、ルールを一度書くだけで、Rails がデータのアクセスされる場所では必ずそのルールを自動的に適用してくれるということだ。

データ検証ルールを作成する

Rails はたくさんのエラー処理を(ほとんど)タダでやってくれる。これをデモするために、空の category モデルに、いくつか検証ルールを加えてみよう:

```
app\models\category.rb
class Category < ActiveRecord::Base
validates_length_of :category, :within => 1..20
validates_uniqueness_of :category, :message => "already exists"
end
```

これにより次のチェックが自動的に行われる:

- `validates_length_of`: 列が空ではなく、また長すぎないこと
- `validates_uniqueness_of`: 重複値が捕らえられる。私は、Rails の既定エラーメッセージ`'xxx has already been taken'`が嫌いなので、自前のものを使用している。これは、Rails の一般的な機能であり、まず既定値を試してみて、気に入らなければ上書きをすればよい。

Documentation: ActiveRecord::Validation::ClassMethods

この機能を試すために、重複するレコードを再び入力してみよう。今回は、Rails がクラッシュするのではなく、エラーを処理してくれる。このスタイルは気に入らないだろう—けしてユーザーインターフェイスを考慮した繊細なものではない。だが、タダで手に入るのだからこんなものだろう。

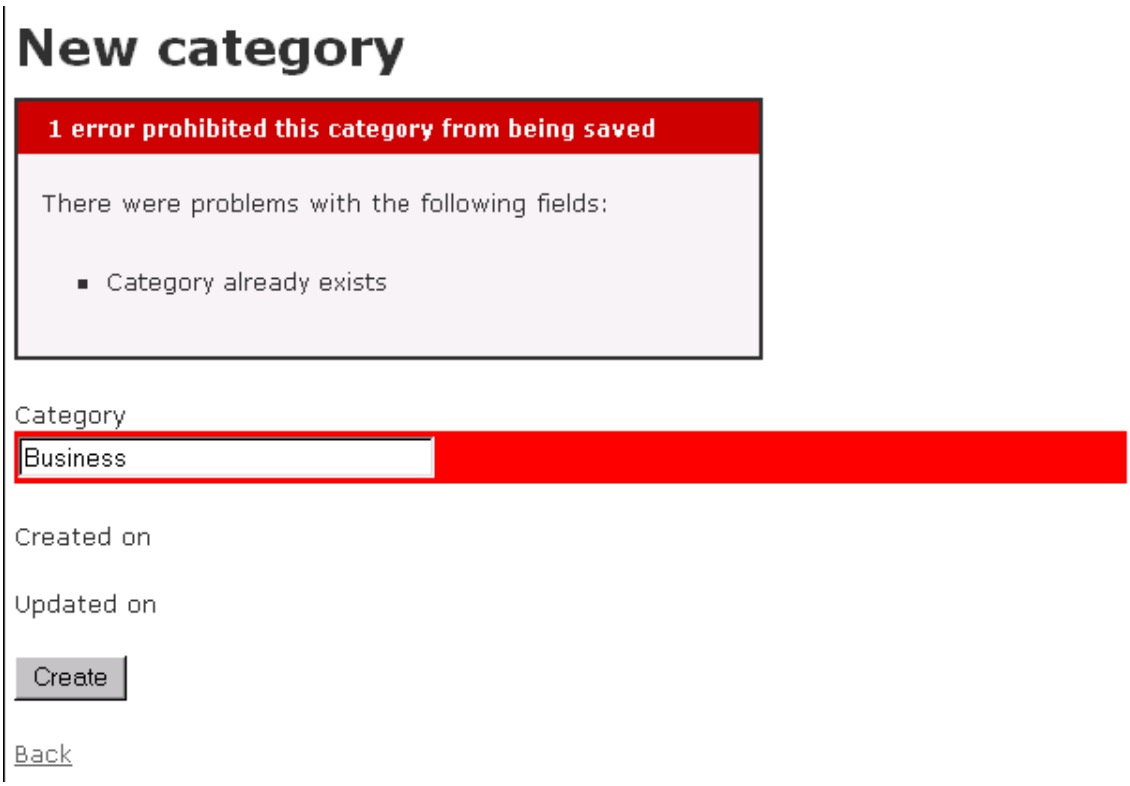

図 2: データエラーを捕捉する

Rails 2 日目

ここから先に進むためには、舞台裏で何が行われているかを知る必要がある。2 日目を通して、Rails が生成する Scaffold コードを体系的に検証し、そのコードの意味を解き明かしてゆきたい。Scaffold を*アクション*として実行すると、Rails は必要なコードを全て動的に生成する。Scaffold を*スクリプト*として実行すると、コードはすべてディスクに書き込まれるので、私たちはコードを調べ、そして、自分の要求に合うようにカスタマイズすることができるようになる。

Running the generate scaffold script

```
W:\ToDo>ruby script/generate scaffold category
dependency model
exists app/models/
exists test/unit/
exists test/fixtures/
skip app/models/category.rb
skip test/unit/category_test.rb
skip test/fixtures/categories.yml
exists app/controllers/
exists app/helpers/
create app/views/categories
exists test/functional/
create app/controllers/categories_controller.rb
create test/functional/categories_controller_test.rb
create app/helpers/categories_helper.rb
create app/views/layouts/categories.rhtml
create public/stylesheets/scaffold.css
create app/views/categories/list.rhtml
create app/views/categories/show.rhtml
create app/views/categories/new.rhtml
create app/views/categories/edit.rhtml
create app/views/categories/_form.rhtml
W:\ToDo>
```

このスクリプトは、完全なアプリケーションを構成するのに必要な、たくさんのファイルを生成する。ファイルには、コントローラ、ビュー、レイアウトそして、スタイルシートまでもが含まれる。ちょっと変わった命名規則によって、単数形から複数形に変わっている点に注意すること。だから、新しいコードを参照するためには、ブラウザで http://todo/categories (WEBrick で hosts ファイルの設定をしていない場合には http://localhost:3000/categories/)を参照するようにしなければならない。実際、混乱を避ける意味からは、`app\controllers\category_controller.rb` といったファイルを誤って実行しないように、削除しておくのが最も望ましい。

生成された Scaffold コード

コントローラ

コントローラの背後のコードを見てみよう。コントローラにはアプリケーションのプログラミングロジックが格納される。コントローラはビューを通じてユーザーとやり取りを行い、モデルを通してデータベースとやり取りする。コントローラのコードを読んで、アプリケーションがどのように成り立っているかを理解できるようになってほしい。Generate scaffold スクリプトによって生成されるコントローラを下記に示す。

\app\controllers\categories_controller.rb

```
class CategoriesController < ApplicationController
  def index
    list
    render :action => 'list'
  end
```

```
  # GETs should be safe (see http://www.w3.org/2001/tag/doc/whenToUseGet.html)
  verify :method => :post, :only => [ :destroy, :create, :update ],
         :redirect_to => { :action => :list }

  def list
    @category_pages, @categories = paginate :categories, :per_page => 10
  end

  def show
    @category = Category.find(params[:id])
  end

  def new
    @category = Category.new
  end

  def create
    @category = Category.new(params[:category])
    if @category.save
      flash[:notice] = 'Category was successfully created.'
      redirect_to :action => 'list'
    else
      render :action => 'new'
    end
  end

  def edit
    @category = Category.find(params[:id])
  end

  def update
    @category = Category.find(params[:id])
    if @category.update_attributes(params[:category])
      flash[:notice] = 'Category was successfully updated.'
      redirect_to :action => 'show', :id => @category
    else
      render :action => 'edit'
    end
  end

  def destroy
    Category.find(params[:id]).destroy
    redirect_to :action => 'list'
  end
end
```

Rails アプリケーションのユーザーが、アクション、たとえば Show のようなものを選択すると、コントローラは、'def show'で示される適切なセクションのコードを実行し、それから、既定では同じ名前のテンプレート'show.rhtml'をレンダリングする。この既定の動作は書き換え可能である。

- render_template によって、異なるテンプレートをレンダリングさせることができる。たとえば、index アクションは、'list'すなわち'def list'のコードを実行し、index.rhtml(これは存在しないのだが)の代わりに list.rhtml をレンダリングできる。
- redirect_to はもう一歩進んで、外部の'302 Found'という HTTP レスポンスを利用し、コントローラに戻ってこさせる。たとえば、destroy アクションは、テンプレートをレンダリングする必要が無い。その主な目的(カテゴリーを破棄すること)を実行した後、destroy は単純にユーザーを list アクションに誘導する。
-

Documentation: ActionController::Base

コントローラは、find, new, save, update_attribute そして destroy という Active Record のメソッドを使って、データベーステーブルとの間でデータをやり取りする。いかなる SQL 文も書く必要が無い点に注目し

てほしい。ただ、Rails がどのような SQL を使っているか知りたければ、development.log ファイルに全て記録されている。

Documentation: ActiveRecord::Base

ユーザーの視点から見ると 1 つの論理的アクションを実行する際に、コントローラを 2 回通らなければならないことに注意してほしい。たとえば、テーブルの中のレコードを更新するような場合である。ユーザーが'Edit'を選択すると、コントローラはモデルの中からユーザーが編集したいレコードを抽出し、それから edit ビューをレンダリングする。ユーザーが編集を終えると、edit ビューは、モデルを更新し、show アクションを起動する update アクションを起動する。
(訳注:Rails 1.1.6 では、Scaffold を生成すると、destroy, create, update のアクションについては、POST のみを受け付け、GET によるアクセスを list にリダイレクトするフィルタが自動的に挿入されている。これは、W3C のガイドラインでは、GET によるアクセスは「安全」で無ければならない、つまり、データの変更や削除を招くようなアクセスは POST で無ければならないと定められているため、これに準拠させるためである。)

ビュー

ビューはユーザーインターフェイスが定義される部分である。Rails は次の 3 つの部品から最終的な HTML を生成することができる。

レイアウト(Layout)	テンプレート(Template)	パーシャル(Partial)
app\views\layouts\の下 デフォルト: application.rhtml 又は <controller>.rhtml	app\views\<controller>\ の下 デフォルト: <action>.rhtml	app\views\<controller>\ の下 デフォルト: <partial>.rhtml

- レイアウトは、全てのアクションに共通なコードを提供する。典型的なものとしてブラウザに送られる HTML の始めと終わりがある。
- テンプレートは、アクションに特有なコードを提供する。たとえば、'List'のコードや、'Edit'のコードなどである。
- パーシャルは、複数のアクションで利用可能な、共通なコードの‘サブルーチン’を提供する。たとえば、フォームのための表をレイアウトするためのコードなどである。

レイアウト

Rails の命名規則: app\views\layouts\ディレクトリの中に、コントローラと同じ名前のレイアウトがあれば、特に明示的に異なる指定がされていない限り、そのコントローラのレイアウトとして使用される。
コントローラと同じ名前のレイアウトが存在せず、明示的に指定されたレイアウトが無い場合、アプリケーション名.rhtml あるいはアプリケーション名.xml という名前のファイルがレイアウトとして使用される。
Scaffold スクリプトが生成するレイアウトは、下記のようなものである。

app\vies\layouts\categories.rhtml

```
<html>
<head>
  <title>Categories: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

<p style="color: green"><%= flash[:notice] %></p>

<%= yield  %>

</body>
</html>
```

これは、ほとんどがHTMLだが、中に少しだけ<% %>で囲まれたRubyのコードが含まれている。このレイアウトは、どのようなアクションが実行されるかにかかわらず、レンダリングプロセスから呼び出される。このレイアウトには、<html><head>.. .</head><body>.. .</body></html>という標準的なHTMLタグが含まれている。これは全てのページに表れるものだ。Railsのレンダリングプロセスにより、太字で示したRubyのコードが次のようにしてHTMLに変換される:

- action_nameはActionControllerのメソッドで、コントローラが実行中のアクション(たとえば'List')の名前を返す。これにより実行されたアクションに応じたタイトルがページ上に表示される。

Documentation: ActionController::Base

- stylesheet_link_tag はrailsのHelper、すなわち楽をしてコードを生成する方法である。Railsにはたくさんの"Helper"がある。このHelperは単純に次のHTMLを生成する:
  `<link href="/stylesheets/scaffold.css" media="screen" rel="Stylesheet" type="text/css" />`

Documentation: ActionView::Helpers::AssetTagHelper

yieldは次に何が起こるかのカギである。これによって、たった一つのレイアウトの中に、実行されるアクション(たとえば'edit', 'new', 'list')に応じて、レンダリング時に動的なコンテントを挿入することができるのだ。この動的なコンテントは同名のテンプレートからくるものだ。下記を参照すること。(訳注:このチュートリアルの2.3では、@content_for_layoutという特殊な変数がこの場所で使われていた。現在ではこれは廃止される方向であり、より一般化されたyieldが使用される)

Documentation: ActionController::Layout::ClassMethods.

テンプレート

Railsの命名規則: テンプレートは次の場所にあるapp\views\categories\'action'.rhtml.

Scaffoldスクリプトによって作成されたnew.rhtmlは下記の通り:

app\views\categories\new.rhtml

```
<h1>New category</h1>

<%= start_form_tag :action => 'create' %>
  <%= render :partial => 'form' %>
  <%= submit_tag "Create" %>
<%= end_form_tag %>

<%= link_to 'Back', :action => 'list' %>
```

- start_form_tag はHTMLのフォームを開始するRailsヘルパーで次のタグを生成する。
  `<form action="/categories/create" method="post">`
- submit_tag はそれ自体
  `<input name="submit" type="submit" value="Save changes" />`
  というタグを生成するが、"Create"パラメータにより、既定の"Save changes"が"Create"に書き換えられる。
- end_form_tag はただ、</form>タグを出力するだけである。これはこれまで書かれた中でもっとも有用なRailsヘルパーではないが:-) ブロックの終了を示してくれる。

Documentation: ActionView::Helpers::FormTagHelper

- render_partial はパーシャルな_form.rhtmlを起動する – 次のセクションを参照。

Documentation: ActionView::Partials

-
- link_to は単純にリンク、HTMLの最も基本的な部分を生成する。
  Back

Documentation: ActionView::Helpers::UrlHelper

パーシャル

Rails の命名規則:“foo”というパーシャルは、app\views\'action'_foo.rhtml というファイルに存在する。(最初のアンダースコアに注意)

Scaffold は、‘edit’アクションと‘new’アクションで同じコードを使用する。このため、scaffold はそのコードを render_partial メソッドで起動されるパーシャルに格納する。

app\views\categories_form.rhtml

```
<%= error_messages_for 'category' %>

<!--[form:category]-->
<p><label for="category_category">Category</label><br/>
<%= text_field 'category', 'category'  %></p>

<p><label for="category_created_on">Created on</label><br/>
<%= datetime_select 'category', 'created_on'  %></p>

<p><label for="category_updated_on">Updated on</label><br/>
<%= datetime_select 'category', 'updated_on'  %></p>
<!--[eoform:category]-->
```

- **error_messages_for** は、フォームをサブミットする際に発生したエラーメッセージをマークアップしたテキストを返す。ひとつ以上のエラーが検出された場合、当該の HTML は次のようになる:

-

```
<div class="errorExplanation" id="errorExplanation">
<h2>n errors prohibited this xxx from being saved</h2>
<p>There were problems with the following fields:</p>
<ul>
<li>field_1 error_message_1</li>
<li>... ...</li>
<li>field_n error_message_n</li>
</ul>
</div>
```

これの実例を 1 日目のページ 9 の「図 2:データエラーを捕捉する」で目にしている。CSS のタグが、scaffold スクリプトで生成されたスタイルシートの文に対応していることに注意。

Documentation: ActionView::Helpers::ActiveRecordHelper

- text_field はつぎのHTMLを生成するRailsヘルパーである:
  <input id="category_category" name="category[category]" size="30"
  type="text" value="" />。
  最初のパラメータがテーブル名、2番目がフィールド名である。

Documentation: ActionView::Helpers::FormHelper

Rails では、予約済みの列、created_on と updated_on に対応するラベル及び入力フィールドが生成されてしまうことに注意。

'New'アクション用のレンダーされたビュー

さて、いよいよ'New'アクションの結果として、ブラウザに返されるコードと、それがどこから来たかを見ることにしよう。レイアウトからの出力は太字、テンプレートからの出力は普通のフォントで、パーシャルからの出力はイタリックで示してある:

```
<html>
<head>
<title>Categories: new</title>
<link href="/stylesheets/scaffold.css" media="screen" rel="Stylesheet"
type="text/css" />
</head>
<body>
<h1>New category</h1>
<form action="/categories/create" method="post">
<!--[form:category]-->
<p><label for="category_category">Category</label><br/>
<input id="category_category" name="category[category]" size="30" type="text"
value=""/></p>
<p><label for="category_created_on">Created on</label><br/></p>
<p><label for="category_updated_on">Updated on</label><br/></p>
<!--[eoform:category]-->
<input name="submit" type="submit" value="Create" />
</form>
<a href="/categories/list">Back</a>
</body>
</html>
```

'List'アクションのためのビューの分析

'Edit'と'Show'のビューは、'New'ビューと同様である。'List' にはいくつかの新しい仕掛けが含まれている。コントローラが、'List'テンプレートをレンダリングする前に、次のコードをどのように処理するか覚えておいてほしい:

```
@category_pages, @categories = paginate :category, :per_page => 10
```

paginate は、インスタンス変数である@categories に、Categories テーブルのレコードを並べ替え、一度に :per_page 行分だけ、前頁・次頁などのロジックを含んだ形で展開する。**@category_pages** は Paginator のインスタンスである。これらがテンプレート内でどのように使われているかは、次のセクションの終わりで説明する。

Documentation: ActionController::Pagination

テンプレートは次の通り:

app\views\categories\list.rhtml

```
<h1>Listing categories</h1>

<table>
  <tr>
  <% for column in Category.content_columns %>
    <th><%= column.human_name %></th>
  <% end %>
  </tr>

<% for category in @categories %>
  <tr>
```

```
  <% for column in Category.content_columns %>
    <td><%=h category.send(column.name) %></td>
  <% end %>
    <td><%= link_to 'Show', :action => 'show', :id => category %></td>
    <td><%= link_to 'Edit', :action => 'edit', :id => category %></td>
    <td><%= link_to 'Destroy', { :action => 'destroy', :id => category }, :confirm =>
'Are you sure?', :post => true %></td>
  </tr>
<% end %>
</table>

<%= link_to 'Previous page', { :page => @category_pages.current.previous } if
@category_pages.current.previous %>
<%= link_to 'Next page', { :page => @category_pages.current.next } if
@category_pages.current.next %>

<br />

<%= link_to 'New category', :action => 'new' %>
```

- content_columns は、特別な列(主キーの列、_id や_count で終わる列、単一テーブル継承で使われる列)を除いた列の配列オブジェクトを返す。

Documentation: ActionController::Base

- human_name は、human_attribute_name の同義語であり、属性のキー名をより人間が読みやすい名前に変換する。たとえば、'first_name'ではなく、'First name'のようにである。

Documentation: ActiveRecord::Base

- h は自動的に'HTML'コードをエスケープする。ユーザーが入力し、それが表示される場合の問題のひとつは、ユーザーが偶然(あるいは悪意を持って)表示されたときにシステムを破壊するコードを入力できることにある[4]。この問題に対する防御策として、ユーザーが入力したデータについてHTMLをエスケープするのは良い習慣である。たとえば、</table>は、< /table&gt のようにレンダリングされるが、これは無害である。Rails ではこれは非常に簡単であり、ここに示したとおり、'h'をつけるだけでよい。
- confirm は、link_to ヘルパーの有用なオプションであり、リンク先を実行する前に、ユーザーに対して削除の確認を求める Javascript のポップアップボックスを表示する。

図 3: Javascript ポップアップ

(訳注:Rails 1.1.6 では、destroy アクションを呼び出す際に、GET ではなく POST を使用するように controller が変更された。このため、link_to にも post メソッドを指定する:post => true オプションが加えられている)

Documentation: ActionView::Helpers::UrlHelper

ページングのロジックはちょっとわかりにくい。Ruby では、if をモディファイアとして使用できる:*式 if 論理式*で

4 たとえば、ユーザーが Category に</table>と入力した場合を考えていただきたい。

は、論理値が真の場合のみ、式が評価される。@category_pages.current は、ページネータの現在のページを示すページオブジェクトを返す。

ActionController::Pagination::Paginator

そして、@category_pages.current.previous は、このページの直前のページを示すページオブジェクト、又は、このページが最初のページの場合には、nil を返す。

ActionController::Pagination::Paginator::Page

したがって、もし、ナビゲートすべき前のページがあれば、この仕組みはリンクを表示するし、無ければリンクは出力されない。
ページ n に出力されるコードは次のようになる:

```
<a href="/categories/list?page=[n-1]">Previous page</a>
<a href="/categories/list?page=[n+1]">Next page</a>
```

生成された Scaffold コードを仕上げる

Scaffold スクリプトが生成したコードは、そのままでも完璧に機能するし、データモデルに十分な検証機能を付け加えれば、強固になる。しかしながら、それが Rails アプリケーション開発でしなければならないことの全てであれば、プログラマは職を失うだろう。それは明らかにいいことではない。:-) だから、仕上げをしよう。

コントローラ

'List'ビューでは、レコードがアルファベット順に並んでいることを期待するだろう。これには、コントローラにほんの少しだけ変更が必要だ:

app\controllers\categories_controller.rb (抜粋)

```
def list
@category_pages, @categories = paginate :category, :per_page => 10, :order_by =>
'category'
end
```

Documentation: ActionController::Pagination

このアプリケーションでは、Show 画面は必要ない—全ての列が一覧表の画面にきちんと収まる。だから、show の定義は無くても良く、'Edit'の後は、直接一覧表示に戻ることにしよう:

app\controllers\categories_controller.rb (抜粋)

```
 def update
    @category = Category.find(params[:id])
    if @category.update_attributes(params[:category])
      flash[:notice] = 'Category was successfully updated.'
      redirect_to :action => 'list'
    else
      render :action => 'edit'
    end
 end
```

フラッシュメッセージは次の画面でピックアップされて表示される。この場合には list 画面である。Scaffold スクリプトでは、既定ではフラッシュメッセージは表示されないが、まもなく変更してみることにする。下記を参照のこと。

ビュー

フラッシュメッセージを表示する

Rails は、ユーザに対してフラッシュメッセージを返すテクニックを提供している。たとえば、‘正常に更新されました’などというメッセージが、次の画面に表示され、次には消えるというものだ。これらのメッセージは、レイアウトにちょっとした変更を加えることで取り出すことができる。(レイアウトに変更を加えるということは、全ての画面に表示されるということである):

app\views\layouts\categories.rhtml

```
<html>
<head>
  <title>Categories: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>

<body>
<h1><%=@heading %></h1>
<% if @flash[:notice] %>
<span class="notice">
<%=h flash[:notice] %>
</span>
<% end %>

<%= yield  %>

</body>
</html>
```

Documentation: ActionController::Flash

スタイルシートに簡単な追加を行うことで、フラッシュメッセージを目立たせることができる:

public\stylesheets\scaffold.css (抜粋)

```
.notice {
color: red;
}
```

テンプレートとレイアウトの間で変数を共有する

<h1>...</h1> というヘッダテキストが、フラッシュメッセージの上に表示されるようにするために、これをテンプレートからレイアウトに移したことに注意してほしい。テンプレートそれぞれには異なったヘッダがあるので、テンプレートの中で、@heading という変数に値をセットした。Rails では、これは何の問題もない—テンプレート変数は、レンダリング時にレイアウトから参照可能である。
この変更と、他にいくつかのフォーマット変更をしたら、このテンプレートは完成だ。

```
<% @heading = "Category" %>

<table>
  <tr>
    <th>Category</th>
    <th>Created</th>
    <th>Updated</th>
  </tr>

<% for category in @categories %>
  <tr>
    <td><%=h category["category"] %></td>
    <td><%= category["created_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= category["updated_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= link_to 'Edit', :action => 'edit', :id => category %></td>
    <td><%= link_to 'Delete', { :action => 'destroy', :id => category }, :confirm =>
```

```
'Are you sure you want to delete this category?', :post => true %></td>
  </tr>
<% end %>
</table>
<br />
<%= link_to 'New category', :action => 'new' %>
<% if @category_pages.page_count > 1 %>
<hr />
Page: <%= pagination_links @category_pages %>
<hr />
<% end %>
```

- 既定の日付フォーマットは好きではないので、strftime()というRubyメソッドを使用して、好きなように日付と時刻をフォーマットした。

Ruby Documentation: class Time

- pagination_links は、ページネータのための基本的なHTMLリンクを生成する。

ActionView::Helpers::PaginationHelper

Edit画面とNew画面の仕上げ

NewとEditで使われるパーシャルへのいくつかの変更:レイアウト改善のためのテーブルの使用;必要の無いcreated_on/updated_onの削除;ユーザーがCategory列に長すぎる文字列を入力することを防止するなどである。

app\views\categories_form.rhtml
```
<%= error_messages_for 'category' %>
<table>
<tr>
  <td><b><label for="category_category">Category:</label></b></td>
  <td><%= text_field 'category', 'category', 'size' => 20, 'maxlength' => 20 %></td>
</tr>
</table>
```

二つのテンプレートへのわずかばかりの変更(とくに@headingの使用に注目):

app\views\categories\edit.rhtml
```
<% @heading = 'Edit category' %>
<%= start_form_tag :action => 'update', :id => @category %>
<%= render_partial 'form' %>
<hr />
<%= submit_tag 'Save' %>
<%= end_form_tag %>
<%= link_to 'Back', :action => 'list' %>
```

app\views\categories\new.rhtml
```
<% @heading = 'New Category' %>
<%= start_form_tag :action => 'create' %>
<%= render_partial 'form' %>
<hr />
<%= submit_tag 'Save' %>
<%= end_form_tag %>
<%= link_to('Back', :action => 'list') %>
```

これで、Railsの旅二日目は終わりである。Categoriesテーブルの保守をするための実用的なシステムが出来上がった。Railsが生成したScaffoldコードの扱い方を習得し始めたわけだ。

Rails 3 日目

さて、アプリケーションの中核部分の作業に入ろう。Item テーブルは、‘ToDo’のリストを格納する。それぞれの Item は 2 日目に作成したカテゴリのひとつに属する。Item にはオプションで、別のテーブルに Note を持つことがある。これは明日見るとしよう。それぞれのテーブルには’id’という主キーがあり、これを使って複数のテーブルの間のリンクを構成する。

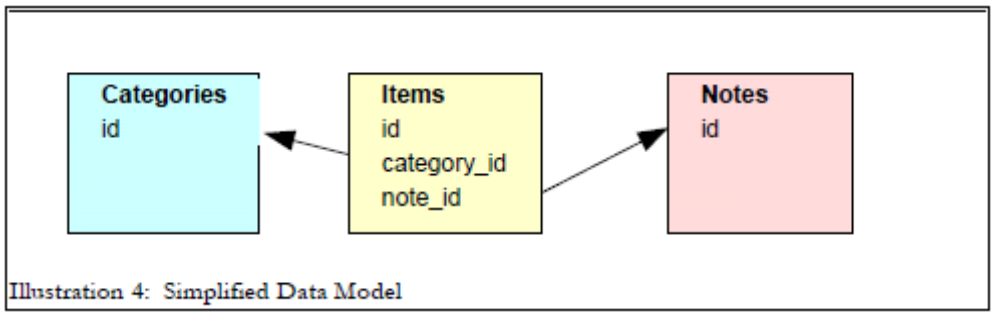

図 4: 単純化したデータモデル

アイテムテーブル

モデルの生成

Category と同様、スクリプトを使用して Item モデルを生成する。

```
W:\ToDo>ruby script/generate model Item
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/item.rb
      create  test/unit/item_test.rb
      create  test/fixtures/items.yml
      exists  db/migrate
      create  db/migrate/002_create_items.rb

W:\ToDo>
```

マイグレーションファイルを編集する

Item テーブルの列は次の通り:

- done — true は To Do アイテムが完了したことを示す[5]
- priority —1 (重要度高) から 5 (重要度低)
- description — 何をしなければならないかを記述した自由文
- due_date — いつまでに行われなければならないかの日付
- category_id — このアイテムが属するカテゴリーへのリンク(Category テーブルの‘id’列)
- note_id — アイテムを説明するオプションのノートへのリンク(Notes テーブルの‘id’列)
- private — true は ToDo アイテムがプライベートであることを示す。

db\migrate\002_create_items.rb

```
class CreateItems < ActiveRecord::Migration
  def self.up
    create_table :items do |t|
      t.column :done, :boolean, :null => false, :default => false
      t.column :priority, :integer, :null => false, :default => 3
      t.column :description, :string, :null => false, :default => ''
      t.column :due_date, :date, :default => nil
      t.column :category_id, :integer, :null => false, :default => 0
      t.column :note_id, :integer, :default => nil
      t.column :private, :boolean, :null => false, :default => false
```

5 MySQL には’boolean’型はないので、0/1 を使わねばならないが、マイグレーションは自動的に boolean と 0/1 のマッピングを行う。

```
      t.column :created_on, :datetime, :null => false
      t.column :updated_on, :datetime, :null => false
    end
  end

  def self.down
    drop_table :items
  end
end
```

マイグレーションの実行

Rake スクリプトでマイグレーションを実行する。

```
W:\ToDo>rake db:migrate
(in W:/ToDo)
== CreateItems: migrating ====================================================
-- create_table(:items)
   -> 0.8320s
== CreateItems: migrated (0.8320s) ===========================================


W:\ToDo>
```

モデルの編集

マイグレーションの後、モデルファイルを次のように編集する:

```
app\models\item.rb
class Item < ActiveRecord::Base
  belongs_to :category
  validates_associated :category
  validates_format_of :done_before_type_cast, :with => /[true|false]/, :message=>"must
be true or false"
  validates_inclusion_of :priority, :in=>1..5, :message => 'must be between 1 (high) and
5 (low)'
  validates_presence_of :description
  validates_length_of :description, :maximum=>40
  validates_format_of :private_before_type_cast, :with => /[true|false]/,
:message=>"must be true  or false"
end
```

テーブル間のリンクを検証する

- `belongs_to` と `validates_associated` を使って、Item テーブルの `category_id` を Category テーブルの `id` とリンクする。

Documentation: ActiveRecord:Associations::ClassMethods

ユーザー入力の検証

- `validates_presence_of` は、'NOT NULL'列に対してユーザー入力がないという事態を防止する。
- `validates_format_of` はユーザー入力のフォーマットを正規表現と比較する。
- ユーザーが数値列に対して入力すると、Rails は常にそれを数値に変換し、もし変換に失敗した場合には 0 として扱う。ユーザーが実際に入力したものが数値かどうかをチェックしたい場合には、入力を `before_type_cast` を使って検証する必要がある。これによって「生の」入力にアクセスできる。[6]

6 もっと明白に使えそうな代替案: `validates_inclusion_of :done_before_type_cast,:in=>"0".."1", :message=>"must be between 0 and 1"`は、もし入力フィールドがブランクのままだとうまく動作しない。(訳注:0/1 を使用する場合の参考として)

- validates_inclusion_of ユーザー入力を許されている値の範囲と照合する。
- validates_length_of ユーザーが保存されるときに切り捨てられてしまうようなデータを入力することを防止する。[7]

ActiveRecord::Validations::ClassMethods

'Notes'テーブル

このテーブルは、特定の ToDo アイテムに関する詳細な情報を保持する、1 個の自由文入力列を持つ。この情報は、Item テーブルの列にあっても良い情報であるが、こうしたほうが Rails について、より多くのことを知ることができる。:-)

モデルを生成する

```
W:\ToDo>ruby script/generate model Note
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/note.rb
      create  test/unit/note_test.rb
      create  test/fixtures/notes.yml
      exists  db/migrate
      create  db/migrate/003_create_notes.rb

W:\ToDo>
```

マイグレーションファイルの編集

db\migrate\003_create_notes.rb

```
class CreateNotes < ActiveRecord::Migration
  def self.up
    create_table :notes do |t|
      t.column :more_notes, :text, :null => false
      t.column :created_on, :datetime, :null => false
      t.column :updated_on, :datetime, :null => false
    end
  end

  def self.down
    drop_table :notes
  end
end
```

マイグレーションの実行

```
W:\ToDo>rake db:migrate
(in C:/work/ToDo)
== CreateNotes: migrating =====================================================
-- create_table(:notes)
   -> 0.6510s
== CreateNotes: migrated (0.6510s) ============================================


W:\ToDo>
```

モデルファイルの編集

モデルファイルを編集する。今のところ新しいものは何も無い:

7 Description 列用の 2 つの検証をひとつにまとめることもできる: validates_length_of :description, :within => 1..40

```
app\models\note.rb
class Note < ActiveRecord::Base
  validates_presence_of :more_notes
end
```

しかし、Items モデルに次のリンクをくわえておくことは忘れることができない:

```
app\models\item.rb (抜粋)
class Item < ActiveRecord::Base
  belongs_to :note
```

参照整合性維持のためにモデルを利用する

これから開発するコードでは、ユーザーは任意の Item に対して 1 件の Note を加えられるようにする。しかし、ユーザーが Note を加えた後に Item を削除ししたらどうなるだろう？明らかに、われわれは Note レコードも削除する方法を見つける必要がある。さもなければ、「孤児となった」Note レコードが残ってしまう。

モデル／ビュー／コントローラという設計手法では、このコードはモデルに帰属する。なぜか？後ほど、われわれは、ToDo 画面において、ゴミ箱アイコンをクリックすることで Item を削除できるようになるが、完了した Item を”Purge”をクリックすることでも削除できるようになる。このコードをモデルに入れておけば、削除アクションがどこで行われても、実行されることになる。

```
app\models\item.rb (抜粋)
def before_destroy
  unless note_id.nil?
    Note.find(note_id).destroy
  end
end
```

これはこういう意味である:Item レコードを削除する前に、削除しようとしている Item レコードの Note_id と同じ id を持つ Note レコードを探し出し、それを先に削除する。それが無い場合を除き: :-)

同様に、Notes テーブルからレコードが削除される際には、Items テーブル側でそのレコードへの参照を削除する必要がある:

```
app\models\note.rb (抜粋)
  def before_destroy
    Item.find_by_note_id(id).update_attribute('note_id',nil)
  end
end
```

Documentation: ActiveRecord::Callbacks

もっと Scaffold する

もう少し Scaffold コードを生成しよう。Items テーブルと Notes テーブルについて scaffold する。まだ、Notes については手をつけられないが、ここで scaffold をしておくということは、今日行うコーディングで、Notes テーブルを参照しても、たくさんエラーを出さずにすむということだ。ちょうど家を建てるときに、足場を組んでおけば、壁を 1 枚ずつ立てても、ガラガラガッシャーンなんてことにならないで済むのと同じだ。

```
W:\ToDo>ruby script/generate scaffold Item
[中略]
W:\ToDo>ruby script/generate scaffold Note
[中略]
W:\ToDo>
```

注意:昨日、スタイルシートをカスタマイズしているので、
`overwrite public/stylesheets/scaffold.css? [Ynaq]`
という質問には’n’と答えること。

もう少しビューについて

アプリケーション用のレイアウトを作る

ここまでで、全てのテンプレートに数行の同じコードが入っていることが明らかになっている。したがって、この共通コードをアプリケーション全体のレイアウトに移すことには意味がある。app\views\layouts*.rhtml ファイルを全て削除し、共通の application.rhtml で置き換える。

app\views\layouts\application.rhtml

```
<html>
<head>
<title><%= @heading %></title>
<%= stylesheet_link_tag 'todo' %>
<script language="JavaScript">
<!-- Begin
function setFocus() {
  if (document.forms.length > 0) {
    var field = document.forms[0];
    for (i = 0; i < field.length; i++) {
      if ((field.elements[i].type == "text") || (field.elements[i].type == "textarea")||
(field.elements[i].type.toString().charAt(0) == "s")) {
        document.forms[0].elements[i].focus();
        break;
      }
    }
  }
}
// End -->
</script>
</head>
<body OnLoad="setFocus()">
<h1><%=@heading %></h1>
<% if @flash[:notice] %>
<span class="notice">
<%=h @flash[:notice] %>
</span>
<% end %>
<%= yield %>
</body>
</html>
```

Template で設定された@heading は、今回は<h1>だけでなく<title>でも使われている。

public/stylesheets/scaffold.css はキチンとするために todo.css という名前に変え、レイアウトを美しくするために色、テーブルの罫線に手を加えてある。また、簡単な Javascript を加え、ブラウザで、ユーザーがすぐに入力できるように、最初の入力フィールドにカーソルが来るようにした。

'To Do List' 画面

目指しているのは、PalmPilot や似たような PDA の画面である。最終製品は、図 5 の'To Do List'画面 8 に示す。[8]

ポイント:

- 列見出しの前のチェックマークボタンをクリックすると、完了済み(チェックがついたもの)のアイテム全てが削除される。
- 表示画面は、'Pri', 'Description', 'Due Date', そして'Category'の各列見出しをクリックすることで並べ替えができる。
- 'Done'列の true/false 値は、小さなチェックマークアイコンに変換される
- 期日が過ぎたアイテムは赤字の太字で表示される
- 付随する note がある場合には note アイコンで表示される
- 'Private'列の true/false 値は南京錠のアイコンで表示される
- それぞれのアイテムは右端のアイコンをクリックすることで、編集や削除ができる
- ディスプレイはきれいな縞模様効果で表示される

8 スタイルシートの数行と、アイコンがいくつかあるだけで、画面の感じがずいぶん違うものだ…

- 新しいアイテムは画面下部の‘New To Do...’ボタンをクリックすることで追加できる
- 2 日目に作成した Category へのボタンリンクもある

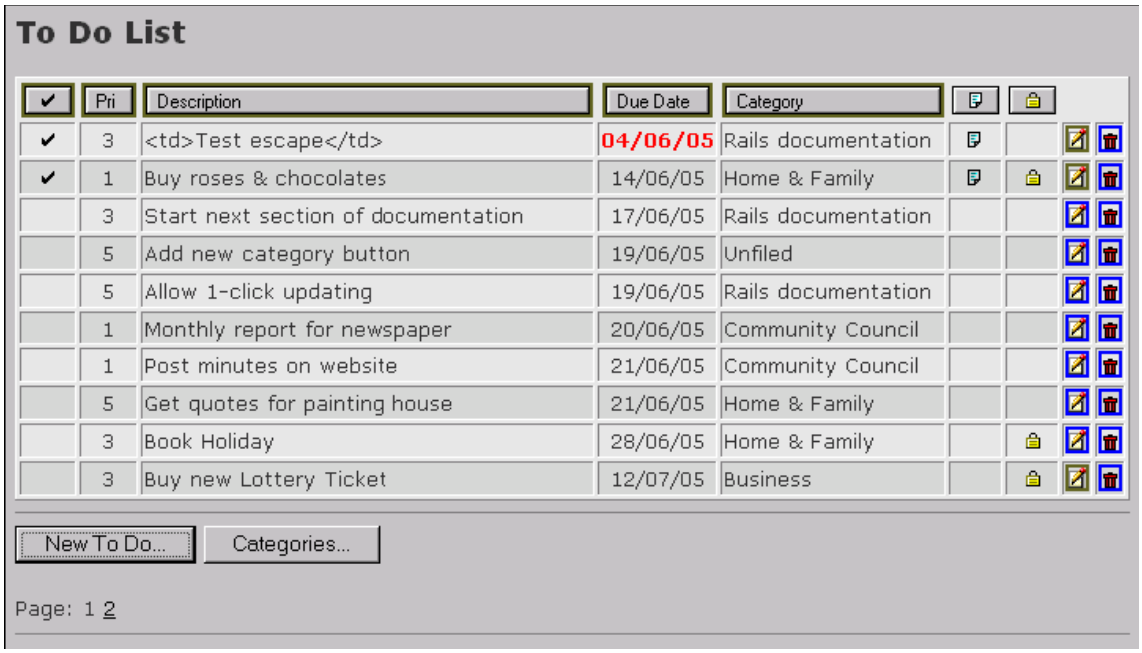

図 5: ‘To Do List’画面

この画面を構成するテンプレートは次の通り:

app\views\items\list.rhtml

```
<% @heading = "To Do List" %>
<%= start_form_tag :action => 'new' %>
<table>
  <tr>
    <th><%= link_to(image_tag('done'), {:action => 'purge_completed'}, :confirm => 'Are
you sure you want to permanently delete all completed To Dos?', :post => true) %></th>
    <th><%= link_to(image_tag('priority', 'alt' => 'Sort by Priority'),{:action =>
'list_by_priority'}) %></th>
    <th><%= link_to(image_tag('description', 'alt' => 'Sort by Description'),{:action =>
'list_by_description'}) %></th>
    <th><%= link_to(image_tag('due_date', 'alt' => 'Sort by Due Date'), {:action =>
'list'}) %></th>
    <th><%= link_to(image_tag('category', 'alt' => 'Sort by Category'), {:action =>
'list_by_category'}) %></th>
    <th><%= show_image 'note' %></th>
    <th><%= show_image 'private' %></th>
    <th> </th>
    <th> </th>
  </tr>
<%= render_collection_of_partials 'list_stripes', @items %>
</table>
<hr />
<%= submit_tag "New To Do..." %>
<%= submit_tag "Categories...", {:type => 'button', :onClick => "parent.location='" +
url_for( :controller => 'categories', :action => 'list' ) + "'" } %>
<%= end_form_tag %>
<%= "Page: " + pagination_links(@item_pages, :params => { :action => @params["action"]
|| "index" }) + "<hr />" if @item_pages.page_count > 1 %>
```

アイコンをクリックすることで完了した ‘To Dos’を削除する

クリッカブルイメージは、link_to と image_tag の 2 つのヘルパーメソッドを組み合わせて使用する。image_tag ヘルパーでは、拡張子の省略されたファイル名を使用可能である。これはデフォルトでは、イメー

ジが pub/images の下にあり、拡張子が .png であることを想定している。イメージをクリックすると、指定されたメソッドが実行される。(訳注:John McCreesh 氏のテキストでは、link_to_image ヘルパーが使用されていたが、これは現在は非推奨となっており、上記のように link_to と image_tag の組み合わせが推奨されている)
:confirm パラメータを指定することで、Javascript のポップアップダイアログが表示されることは前にも説明した。

Documentation: ActionView::Helpers::UrlHelper
Documentation: ActionView::Helpers::AssetTagHelper

'OK'をクリックすると purge_completed メソッドが呼び出される。この新しい purge_completed メソッドはコントローラの中で定義する必要がある:

app\controllers\items_controller.rb (抜粋)
```
 def purge_completed
    Item.destroy_all "done = 1"
    redirect_to :action => 'list'
  end
```

Item.destroy_all は、Items テーブルで、done 列が 1 の全ての行を削除し、もう一度 list アクションを実行する。(destroy_all メソッドの引数は、SQL 文の WHERE 節になるため、データベース管理システムに依存する。MySQL の場合、真偽値を 1,0 で表現しているため、"done = 1"となる)

Documentation: ActiveRecord::Base

列見出しをクリックしてソート順を変える

Pri アイコンをクリックすると list_by_priority メソッドが実行される。この新しい list_by_priority メソッドはコントローラで定義する必要がある:

app\controllers\items_controller.rb (抜粋)
```
  def list
    @item_pages, @items = paginate :item, :per_page => 10, :order_by =>
'due_date,priority'
  end

  def list_by_priority
    @item_pages, @items = paginate :item, :per_page => 10, :order_by =>
'priority,due_date'
    render_action 'list'
  end
```

われわれは、既定の list メソッドに並び順を指定し、新たに list_by_priority メソッド[9]を定義した。Render action を明示的に指定する必要があるということに注意。既定では、Rails は list_by_priority という(存在しない)テンプレートをレンダリングするため。:-)

ヘルパーの追加

Note 列および Private 列の列見出しはイメージだが、クリックはできない。ただイメージを表示するだけの show_image(name) メソッドを定義することにする。

app\helpers\application_helper.rb
```
# Methods added to this helper will be available to all templates in the application.
module ApplicationHelper
  def self.append_features(controller)
    controller.ancestors.include?(ActionController::Base) ?
```

9 list_by_description と list_by_category も同様であり、読者諸氏への練習問題として残しておこう。もし、list_by_category で行き詰まってしまったら、44 ページの「まだやらねばならぬこと」を参照してほしい。

```
    controller.add_template_helper(self) : super
  end

  def show_image(src)
    img_options = { "src" => src.include?('/') ? src : "/images/#{src}" }
    img_options["src"] = img_options["src"] + '.png' unless
    img_options["src"].include?('.')
    img_options["border"] = "0"
    tag('img', img_options)
  end
end
```

このヘルパーは、コントローラでリンクされている:

```
app\controllers\application.rb
class ApplicationController < ActionController::Base
  helper :Application
end
```

このヘルパーはアプリケーション中の全てのテンプレートで使用することができる。

Documentation: ActionView::Helpers

Javascript で作ったナビゲーションボタン

onClick は標準的な Javascript のテクニックであり、新しい Web ページに飛ぶなどのボタンアクションをハンドリングするのに使用される。しかし、Rails は、URL をきれいにすることに長けているため、Rails に正しい URL はどうなっているかをたずねる必要がある。コントローラとアクションを引数で渡すことで、url_for はその URL を返してくれる。

Documentation: ActionController::Base

パーシャルを使ってテーブルをフォーマットする

Item のリストは、きれいなストライプ効果を使って表示したい。パーシャルが答えを提供してくれる。

```
render_partial
<% for item in @items %>
<%= render_partial 'list_stripes', item %>
<% end %>
```

あるいは、もっと経済的な記述方法がある。

```
<%= render :partial => 'list stripes', :collection => @items %>
```

Documentation: ActionView::Partials

Rails は、連番である list_stripes_counter を Partial に渡す。これは、表の中の行を、ライトグレーと、ダークグレーの 2 色でフォーマットするための鍵である。カウンタが奇数か偶数かをテストし、奇数ならライトグレー、偶数ならダークグレーだ。

完成したパーシャルは次の通りだ:

```
app\views\items\_list_stripes.rhtml
<tr class="<%= list_stripes_counter.modulo(2).nonzero? ? "dk_gray" : "lt_gray" %>">
  <td style="text-align: center"><%= list_stripes["done"] ? show_image('done_ico.gif') :
' ' %></td>
  <td style="text-align: center"><%= list_stripes["priority"] %></td>
  <td><%=h list_stripes["description"] %></td>
  <% if list_stripes["due_date"].nil? %>
    <td> </td>
  <% else %>
```

```
    <%= list_stripes["due_date"] < Date.today ? '<td class="past_due" style="textalign:
center">' : '<td style="text-align: center">' %>
    <%=  list_stripes["due_date"].strftime("%d/%m/%y") %></td>
  <% end %>
  <td><%=h list_stripes.category ? list_stripes.category["category"] : 'Unfiled' %></td>
  <td><%= list_stripes["note_id"].nil? ? ' ' : show_image('note_ico.gif')%></td>
  <td><%= list_stripes["private"] ? show_image('private_ico.gif') : ' ' %></td>
  <td><%= link_to(image_tag('edit'), { :controller => 'items', :action => 'edit', :id =>
list_stripes.id }) %></td>
  <td><%= link_to(image_tag('delete'), { :controller => 'items', :action =>
'destroy',:id => list_stripes.id }, :confirm => 'Are you sure you want to delete this
item?', :post => true) %></td>
</tr>
```

Rubyを少しだけ使って、カウンターが奇数か偶数かをテストし、class="dk_gray"または class="lt_gray"をレンダーしている:
`list_stripes_counter.modulo(2).nonzero? ? 'dk_gray' : 'lt_gray'`
最初のクエスチョンマークまでのコードは: *`list_stripes_counter`を2で割ったあまりはゼロ以外かということをたずねている*

Ruby Documentation: class Numeric

行の残りの部分は本当に暗号的な *if then else* 式である。簡潔さのために読みやすさを犠牲にしている:
クエスチョンマークの前の式が真なら、コロンの前の値を返し、そうでなければコロンの後の値を返す。

Ruby Documentation: Expressions

dk_gray と lt_gray という二つのタグは、スタイルシートで定義されている:

```
public\stylesheets\ToDo.css (抜粋)
.lt_gray { background-color: #e7e7e7; }
.dk_gray { background-color: #d6d7d6; }
```

注:同様の *if then else* 構造がチェックマークを表示するためにも使われている。list_stripes["done"]が true であるときはチェックマークを、そうでなければ HTML の空白文字を表示する。
list_stripes[″done″] ? show_image('done_ico') : ' '

データ値に基づくフォーマッティング

特定のアイテム、たとえば期日が過ぎてしまったものをハイライトするのは簡単である。
`list_stripes["due_date"] < Date.today ? '<td class="past_due">' : '<td>'`
繰り返しになるが、このためにはスタイルシートに対応する .past_due が必要である。

参照値が無い場合に対応する。

ユーザーが ToDo アイテムで参照されている Category を削除してしまった場合にも対応できるようにしたい。このような場合には、Category は'Unfiled'として表示されるべきである。
list_stripes.category ? list_stripes.category[″category″] : 'Unfiled'
ここまで演習を進めてくれば、23 ページの図 5「ToDo リスト画面」のような画面になっているはずである。

'New To Do' 画面

次は、New ToDo ボタンが押された場合のことを見てみよう。またしてもここには新しい技が潜んでいる。

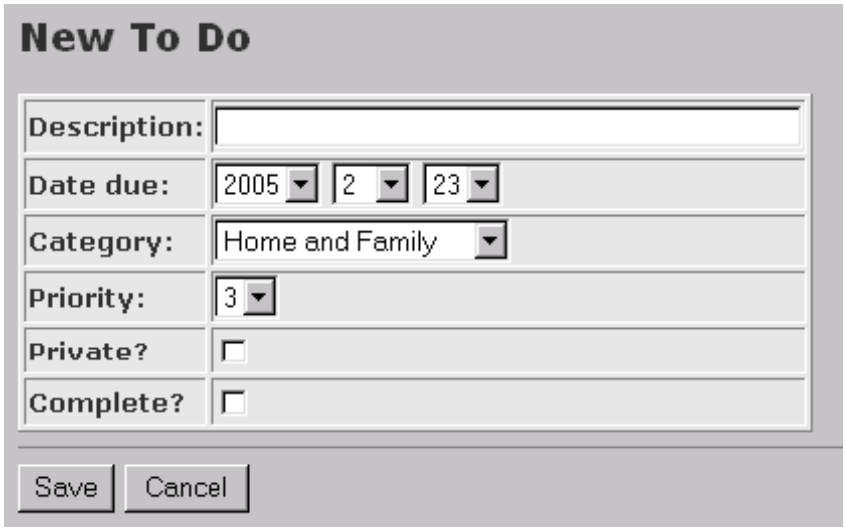

図 6 Net ToDo 画面

テンプレートは最小限である:

app\views\items\new.rhtml
```
<% @heading = "New To Do" %>
<%= error_messages_for 'item' %>
<%= start_form_tag :action => 'create' %>
<table>
 <%= render_partial "form" %>
</table>
<hr />
<%= submit_tag "Save" %>
<%= submit_tag "Cancel", {:type => 'button', :onClick=>"parent.location='" +
url_for(:action => 'list' ) + "'" } %>
<%= end_form_tag %>
```

そして、本当の仕事はパーシャルの中で行われる。これは、Edit アクションと共有できるものだ。

app\views\items_form.rhtml
```
<tr>
  <td><b>Description: </b></td>
  <td><%= text_field 'item', 'description', 'size' => 40, 'maxlength' => 40%></td>
</tr>
<tr>
  <td><b>Date due: </b></td>
  <td><%= date_select 'item', 'due_date', :use_month_numbers => true %></td>
</tr>
<tr>
  <td><b>Category: </b></td>
  <td><select id='item_category_id' name='item[category_id]'>
        <%= options_from_collection_for_select @categories, 'id', 'category',
@item.category_id %>
      </select>
  </td>
</tr>
<tr>
  <td><b>Priority: </b></td>
    <% @item.priority = 3 %>
  <td><%= select 'item','priority',[1,2,3,4,5] %></td>
</tr>
<tr>
  <td><b>Private? </b></td>
  <td><%= check_box 'item','private',{},true,false %></td>
</tr>
<tr>
  <td><b>Complete? </b></td>
```

```
  <td><%= check_box 'item', 'done',{},true,false %></td>
</tr>
```

日付フィールドにドロップダウンリストを作成する
date_select は、日付の入力に初歩的なドロップダウンメニュを生成する。

```
date_select "item","due_date",:use_month_numbers => true
```

Documentation: ActionView::Helpers::DateHelper

Ruby で例外を捕捉する

残念ながら date_select は、2 月 31 日などという日付をいとも素直に受け入れてしまう。Rails はこの値をデータベースに保存しようとして死んでしまう。回避方法のひとつは、Ruby の例外処理メソッドである rescue でこの保存の失敗を捕捉してしまうことである。

```
app\controllers\items_controller.rb (抜粋)
def create
  begin
    @item = Item.new(@params[:item])
    if @item.save
      flash['notice'] = 'Item was successfully created.'
      redirect_to :action => 'list_by_priority'
    else
      @categories = Category.find_all
      render_action 'new'
    end
  rescue
    flash['notice'] = 'Item could not be saved.'
    redirect_to :action => 'new'
  end
end
```

Ruby Documentation: Exceptions, Catch, and Throw

参照テーブルからドロップダウンリストを作成する

これは、Rails がしょっちゅう発生するプログラミングの問題を、非常に簡単に解決するもうひとつの例である。この例では:

```
options_from_collection_for_select @categories, "id", "category",
@item.category_id
```

options_from_collection_for_select は、categories から全てのレコードを読み込み、<option value="*[value of id]*">*[value of category]*</option>の形でレンダリングする。@item_category_id に一致したものが選択されたものとしてタグされる。それだけではなく、データを html_escape してくれる。見事だ。

Documentation: ActionView::Helpers::FormOptionsHelper

データから作成されるドロップダウンリストは、そのデータをどこかから得る必要がある。ということは、コントローラへの追加を意味する:

```
app\controllers\items_controller.rb (抜粋)
def new
  @categories = Category.find_all
  @item = Item.new
end
```

```
def edit
  @categories = Category.find_all
  @item = Item.find(@params[:id])
end
```

コンスタントのリストからドロップダウンを作成する
　これは、前のシナリオの単純なバージョンである。セレクションボックスにリストをハードコーディングするのは、必ずしも良い考えとはいえない。テーブル内のデータを変更するほうが、コード内のデータを変更するよりも簡単だからだ。しかし、ハードコーディングが完全に有効なアプローチである場合も存在する。そのような時は、Railsでは次のように書く:
`select "item","priority",[1,2,3,4,5]`
前行で、どのようにして既定値をセットするのか注意しておくこと。

Documentation: ActionView::Helpers::FormOptionsHelper

チェックボックスを生成する

　もうひとつ常に必要なこと;もうひとつのRailsのヘルパー:
`check_box 'item','private'`

Documentation: ActionView::Helpers::FormHelper

仕上げ

スタイルシートを仕上げる

　ここまでで、'To Do List'画面や、'New To Do'ボタンは機能するようになった。ここで示す画面を構築するために、スタイルシートに次のような変更を追加した:

public\stylesheets\ToDo.css

```
body { background-color: #c6c3c6; color: #333; }
.notice {
color: red;
background-color: white;
}
h1 {
font-family: verdana, arial, helvetica, sans-serif;
font-size: 14pt;
font-weight: bold;
}
table {
background-color:#e7e7e7;
border: outset 1px;
border-collapse: separate;
border-spacing: 1px;
}
td { border: inset 1px; }
.notice {
color: red;
background-color: white;
}
.lt_gray { background-color: #e7e7e7; }
.dk_gray { background-color: #d6d7d6; }
.hightlight_gray { background-color: #4a9284; }
.past_due { color: red }
```

'Edit To Do'画面

　3日目の残りの時間で'Edit To Do'画面の構築を行う。この画面は'New To Do'に非常に良く似ている。学生時代、教科書に「これは読者への練習問題として取っておこう」と書いてあるのにはうんざりさせられたものだ。だ

から、ここで皆さんに同じことができるというのはすばらしいことだ。[10]

それをもって三日目の終わりとしよう。このアプリケーションは、もはやRailsのscaffoldとは似ても似つかぬものになっているが、その裏では、開発を容易にするためのRailsの数々のツールを使っているのだ。

10　しかし、私の大学時代の教科書の著者とは違って、4日目に答えを明かすことにする。:-) 31ページの*app¥views¥items¥edit.rhtml*を参照のこと。

Rails4 日目

'Notes'画面

'Notes'を'Edit To Do'にリンクする

Notes の scaffold コードには、CRUD 機能が全て備わっているが、ユーザーにはどれも直接呼び出してほしくない。その代わりに、Item に付随する Note がない場合、'Edit To Do'画面の Notes のアイコンをクリックすることで、Note を作成できるようにしたい：

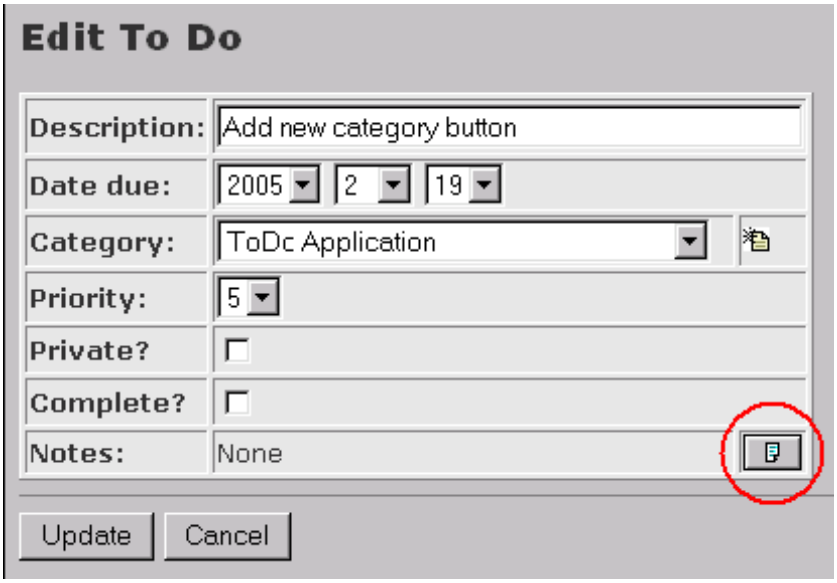

図7：'Edit To Do'画面から新しい Note を作成する

Note が既にある場合には、'Edit To Do'画面の適切なアイコンをクリックすることで、編集や削除を行いたい。：

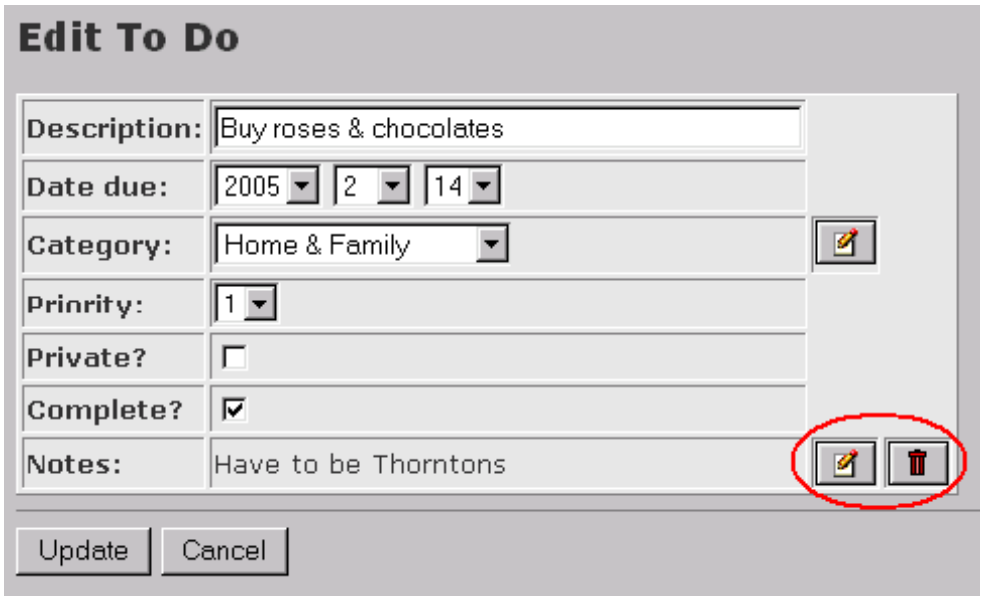

図8：既存の Note の編集または削除

まず最初に、'Edit To Do'画面のコードを見ておこう。ノートボタンが、既存のノートがあるか否かに応じて変わることと、コントロールがどのようにして Notes コントロールに移るかに注意。

app\views\items\edit.rhtml

```
<% @heading = "Edit To Do" %>
<%= error_messages_for 'item' %>
<%= start_form_tag :action => 'update', :id => @item %>
<table>
  <%= render_partial "form" %>
  <tr>
    <td><b>Notes: </b></td>
    <% if @item.note_id.nil? %>
      <td>None</td>
      <td><%= link_to(image_tag('note'), {:controller => 'notes', :action => 'new', :id
=>@item.id}) %></td>
    <% else %>
      <td><%=h @item.note.more_notes %></td>
      <td><%= link_to(image_tag('edit_button'), {:controller => 'notes', :action =>
'edit',:id => @item.note_id}) %></td>
      <td><%= link_to(image_tag('delete_button'), {:controller => 'notes', :action
=>'destroy', :id => @item.note_id }, :confirm => 'Are you sure you want to delete this
note?', :post => true) %></td>
    <% end %>
  </tr>
</table>
<hr />
<%= submit_tag "Save" %>
<%= submit_tag "Cancel", {:type => 'button', :onClick=>"parent.location='" +
url_for(:action => 'list' ) + "'" } %>
<%= end_form_tag %>
```

'Edit Notes' 画面

既存のノートを編集するのは非常にわかりやすい。以下がテンプレートである。

app\views\notes\edit.rhtml

```
<% @heading = "Edit Note" %>
<%= start_form_tag :action => 'update', :id => @note %>
<%= render_partial "form" %>
<%= submit_tag "Save" %>
<%= submit_tag "Cancel", {:type => 'button', :onClick => "parent.location='" + url_for(
:controller => 'items', :action => 'list' ) + "'" } %>
<%= end_form_tag %>
```

それと、対応するパーシャル:

app\views\notes_form.rhtml

```
<table>
  <tr>
    <td><label for="note_more_notes">More notes</label></td>
    <td><%= text_area 'note', 'more_notes' %></td>
  </tr>
</table>
```

一度、ノートの更新あるいは破棄が終われば、'To Do List' 画面に戻りたい。

app\controllers\notes_controller.rb (抜粋)

```
  def update
    @note = Note.find(@params[:id])
    if @note.update_attributes(@params[:note])
      flash['notice'] = 'Note was successfully updated.'
      redirect_to :controller => 'items', :action => 'list'
    else
      render_action 'edit'
    end
  end

  def destroy
    Note.find(@params[:id]).destroy
```

```
    redirect_to :controller => 'items', :action => 'list'
  end
```

我々が作成した参照整合性のルールによって、ノートが削除された場合には、Item 内でそれを指していたいかなる参照も削除されることを思い出していただきたい。(「参照整合性の維持のためにモデルを使用する」を参照すること。

'New Note' 画面

Create はちょっとだけトリッキーだ。我々が行いたいのは、新しい Note を Notes テーブルに保存し、Notes テーブルで新しく作成された id を調べる。そして、それを、Items テーブルの関連する Item の notes_id フィールドに格納する。

セッション変数は、画面間でデータを永続化する有用な方法を提供してくれる。これによって、われわれは Notes テーブルの中のレコードの id を保存することができる。

Documentation: ActionController::Base

セッション変数を使ってデータを保存し取り出す

最初に、まず新しい Notes レコードを作成する。編集する Item の id を渡す:

app\views\items\edit.rhtml (抜粋)

```
<td><%= link_to(image_tag('note'), {:controller => "notes", :action => "new", :id
=>@item.id}) %></td>
```

Notes コントローラの新しいメソッドが、セッション変数にこのように格納する。

app\controllers\notes_controller.rb (抜粋)

```
def new
@session[:item_id] = @params[:id]
@note = Note.new
end
```

'New Notes' テンプレートにはびっくりするような点は無い:

app\views\notes\new.rhtml

```
<% @heading = "New Note" %>
<%= start_form_tag :action => 'create' %>
<%= render_partial "form" %>
<%= submit_tag "Save" %>
<%= submit_tag "Cancel", {:type => 'button', :onClick=>"parent.location='" +
url_for(:controller => 'items', :action => 'list' ) + "'" } %>
<%= end_form_tag %>
```

create メソッドは再びセッション変数を引き出し、Items テーブルから該当するレコードを検索するために使用する。次に create メソッドは、Note テーブルに作成したばかりのレコードの id を使って、Item テーブルの note_id を更新する。そして、再び Items コントローラに戻る:

app\controllers\notes_controller.rb (抜粋)

```
  def create
    @note = Note.new(@params[:note])
    if @note.save
      flash[:notice] = 'Note was successfully created.'
      @item = Item.find(@session[:item_id])
      @item.update_attribute(:note_id, @note.id)
      redirect_to :controller => 'items', :action => 'list'
    else
      render_action 'new'
    end
  end
```

'Categories'画面を変更する

もう、このシステムではたいしてすることは残っていない。始めの方で作成したテンプレートにちょっと手を加え、同じスタイルのナビゲーションボタンを持つようにすることくらいである。:

app\views\categories\list.rhtml

```
<% @heading = "Categories" %>
<form action="/categories/new" method="post">
<table>
  <tr>
    <th>Category</th>
    <th>Created</th>
    <th>Updated</th>
  </tr>
<% for category in @categories %>
  <tr>
    <td><%=h category["category"] %></td>
    <td><%= category["created_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= category["updated_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= link_to(image_tag('edit'), { :action => 'edit', :id => category.id })
%></td>
    <td><%= link_to(image_tag('delete'), { :action => 'destroy', :id => category.id }
,:confirm => 'Are you sure you want to delete this category?', :post => true) %></td>
  </tr>
<% end %>
</table>
<hr />
<%= submit_tag "New Category..." %>
<%= submit_tag "To Dos", {:type => 'button', :onClick => "parent.location='" +
url_for(:controller => 'items', :action => 'list' ) + "'"} %>
</form>
<% if @category_pages.page_count > 1 %>
<hr />
Page: <%= pagination_links @category_pages %>
<hr />
<% end %>
```

app\views\categories\new.rhtml

```
<% @heading = "Add new Category" %>
<%= error_messages_for 'category' %>
<%= start_form_tag :action => 'create' %>
<%= render_partial "form" %>
<hr />
<%= submit_tag "Save" %>
<%= submit_tag "Cancel", {:type => 'button', :onClick => "parent.location='" + url_for(
:action=> 'list' ) + "'"} %>
<%= end_form_tag %>
```

app\views\categories\edit.rhtml

```
<% @heading = "Rename Category" %>
<%= error_messages_for 'category' %>
<%= start_form_tag :action => 'update', :id => @category %>
<%= render_partial "form" %>
<hr />
<%= submit_tag "Update" %>
<%= submit_tag "Cancel", {:type => 'button', :onClick => "parent.location='" + url_for(
:action=> 'list' ) + "'"} %>
<%= end_form_tag %>
```

システム全体のナビゲーション

アプリケーションを通じた最終的なナビゲーションパスは下図の通りである。冗長な scaffold コード、たとえば `show.rhtml` などは、ただ単純に削除すればよい。これは、scaffold コードの美点である。最初につくるときには何の苦労もないし、いったん目的を果たした後は単純に捨てることができる。

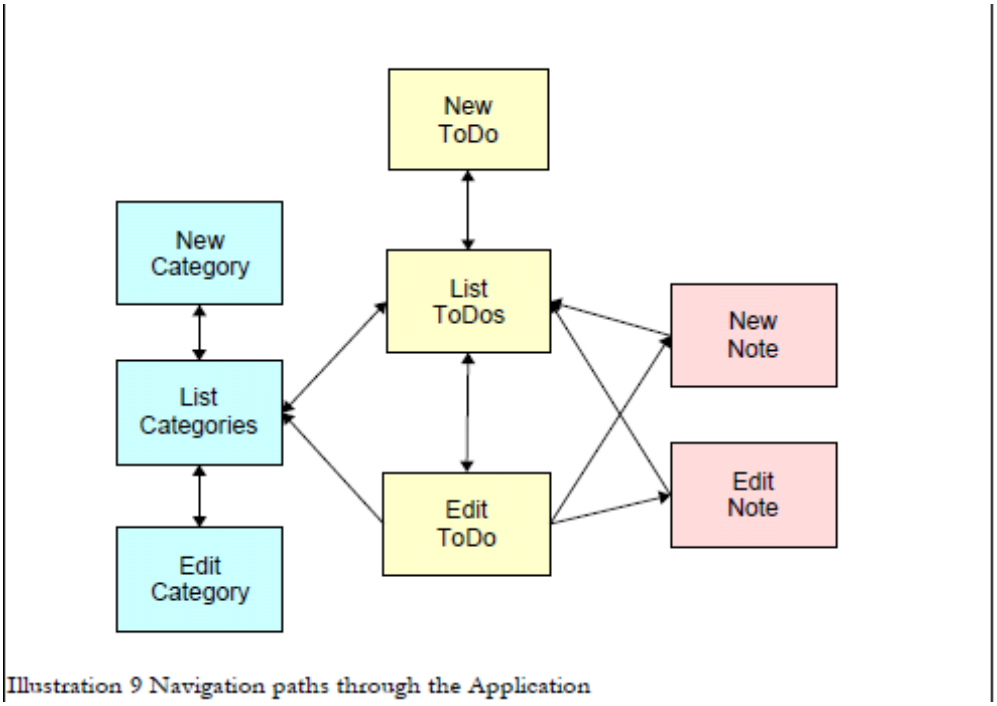

Illustration 9 Navigation paths through the Application

図 9 アプリケーション全体のナビゲーションパス

アプリケーションのホームページを設定する

最後のステップとして、ユーザーがブラウザで `http://todo` を参照した際に表示されるデフォルトの Welcome to Rails?画面を削除しよう。2 つのステップだ:

- Routes ファイルにホームページの定義を加える:

config\routes.rb (抜粋)

```
map.connect '', :controller => 'items'
```

- `public\index.html`のファイル名を `public\index.html.orig`に変更する。

このアプリケーションのコピーをダウンロードする

この‘To Do’アプリケーションを試すためにのコピーをほしければ、http://rails.homelinux.org にリンクがある。

- Rails を使ってディレクトリ構造を作成する必要がある(3 ページの Rails スクリプトを実行するを参照)
- `todo_app.zip` ファイルを新しくできた ToDo ディレクトリにダウンロードする
- ファイルを unzip する `unzip -o todo_app.zip`
- `rename public\index.html public\index.html.orig`
- サンプルデータベースを使いたければ、 `mysql -u root -p < db/ToDo.sql`

そして最後に・・・

このドキュメントがあなたのお役に立てばと思っている ほめ言葉でも悪口でも jpmcc@users.sourceforge.net までフィードバックをいただければうれしい。

さあ、Rails で楽しいコーディングを！

補足:後で考えたこと

‘Four Days’を書いた後、非常にたくさんのフィードバックを戴いた。それらは、このドキュメントの質を向上させるのに非常に役立った。なかでも、繰り返し同じ質問を受けた 「同じ画面で複数のレコードを更新するにはどうすればよいか」というものである。そこで、このもっとも良く聞かれる質問のための補足を加えた。これは、簡単に理解できる Rails の概念ではない。今後もっとたくさんの Helper が出てきてほしい分野である。

複数同時更新

下記の画面では、ユーザーは一番左端のチェックボックスを使って、複数の ToDo 案件にチェックをつけたり外したりできる。その後、‘Save’をクリックして、その結果をデータベースに保存することができる。

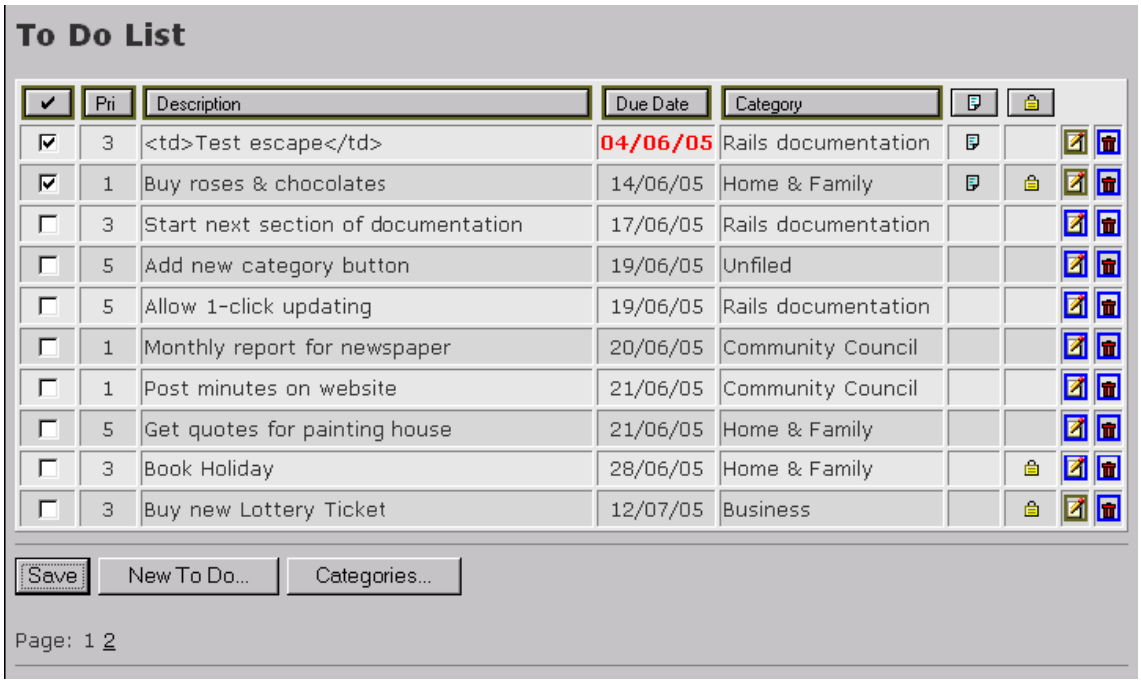

図 10 複数同時更新

ビュー

Rails はもうひとつの命名規約で複数同時更新をサポートする。それは、編集中のレコードの名前の後に角括弧[]で挟まれた id を加えるというものだ。これによって、画面上の複数のレコードから特定のレコードを選び出すことができる。我々が生成しようとしている HTML からさかのぼって見て行こう。これが、id 6のレコードがどのように見えるかだ:

```
<td style="text-align: center">
<input type="checkbox" id="item_done" name="item[6][done]" value="true"
checked />
<input name="item[6][done]" type="hidden" value="false" />
</td>
```

(チェックボックスがチェックされていなければ checked は要らない)

このコードを生成する方法のひとつはこれだ:

app\view\items_list_stripes.rhtm (抜粋)

```
  <td style="text-align: center">
    <%= check_box_tag('item[' + list_stripes.id.to_s +
'][done]','true',list_stripes["done"]) %>
    <%= hidden_field_tag('item[' + list_stripes.id.to_s+ '][done]','false') %>
  </td>
```

check_box_tag のパラメータは、name, value = "true", checked = false, options =

{};
hidden_field_tag の場合は、name, value = nil, options = {} だ

Documentation: ActionView::Helpers::FormTagHelper

そしてもちろん Save ボタンが必要だ:

app\views\items\list.rhtml (抜粋)

```
<% @heading = "To Do List" %>
<%= start_form_tag :action => 'updater' %>
<table>
...
</table>
<hr />
<%= submit_tag "Save" %>
<%= submit_tag "New To Do...", {:type => 'button', :onClick => "parent.location='" +
url_for( :action => 'new' ) + "'" } %>
<%= submit_tag "Categories...", {:type => 'button', :onClick => "parent.location='" +
url_for( :controller => 'categories', :action => 'list' ) + "'" } %>
<%= end_form_tag %>
<%= "Page: " + pagination_links(@item_pages, :params => { :action => @params["action"]
|| "index" }) + "<hr />" if @item_pages.page_count > 1 %>
```

コントローラ

'Save'ボタンをクリックしたときに、コントローラに戻すのは、下記のようなハッシュである:

```
params: {
   :controller=>"items",
   :item=> {
   "6"=>{"done"=>"false"},
   ... etc...
   "5"=>{"done"=>"true"}
},
   :action=>"updater"
}
```

:item ビットに注目してみよう。たとえば、太字の行は、id が 6 のレコードでは、done フィールドの値が false ということを意味している。ここからは、Items テーブルを更新するのは比較的容易である:

app\controller\items_controller (抜粋)

```
  def updater
    @params[:item].each do |item_id, attr|
    item = Item.find(item_id)
    item.update_attribute(:done,attr[:done])
    end
    redirect_to :action => 'list'
  end
```

each によって変数 item_id には 6 が入り、"done" => "false"が attr に入る

Ruby Documentation: class Array

このコードでも動作はするが、development.log で何が起きているかを見ると、Rails が変更されているかいないかにかかわらず、全てのレコードを引き出し、しかも更新していることがわかるだろう。これは、不必要なデータベース更新を生成しているだけではなく、updated_on も更新されることを意味する。これは望ましいことではない。'done'が変わった場合にのみ更新されるほうがうんと良いが、そのためにはちょっとしたコーディングが必要だ。:-(

app\controller\items_controller (抜粋)

```
  def updater
    @params[:item].each do |item_id, attr|
      item = Item.find(item_id)
      if item.done != eval(attr[:done])
        item.update_attribute(:done,attr[:done])
      end
    end
    redirect_to :action => 'list'
  end
```

同類のものを比較するためには、string である done を eval メソッドを使って true/false に変換する必要があることに注意。この手のミスは見逃しやすい。Rails があなたの想定したとおりに動いているかどうかを確認するために、development.log を時々チェックするのは意味があることだ。

ユーザーインターフェイスに関する考察

このコードは機能するが、これを応用してさらに画面上のどの列でも編集できるようにすることもできる。(もうひとつの易しい練習問題だ:-)。ただし、それによってユーザーが何を期待するかという問題が持ち上がってくる。もしもユーザーがチェックボックスのいくつかを変更し、'Save' を押さないまま、'New To Do'をクリックしたり、表示の並べ替えを行ったらどうだろう。システムは常に、別のアクションを実行する前に、'Save' すべきだろうか？これもユーザーへの簡単な練習問題だ・・・

まだやらねばならぬこと

(訳注:Four Days on Rails 2.3 では、Rails 0.12.1 をベースにしていたため、Category で Item をソートするのは、困難であった。原著者の John McCreesh 氏は、find_by_SQL による解決策を提示していたが、Rails 1.1.6 では、外部結合がサポートされているため、下記のようにすっきりと記述することが可能である。)

app\controller\items_controller (抜粋)

```
  def list_by_category
    @item_pages, @items = paginate :items, {:per_page => 10, :include => 'category',
:order_by => 'categories.category,due_date'}
    render_action 'list'
  end
```

Rails で楽しくコーディングしよう！

この文書の Ruby と Rails 用語の索引