入門からゆっくりとステップアップ……!

改訂4版

基礎 Ruby on Rails

IMPRESS KISO SERIES

黒田努・佐藤和人 共著
株式会社オイアクス 監修

DOWNLOAD

インプレス

はじめに

本書は、『基礎Ruby on Rails』の改訂4版です。約3年ぶりの改訂となります。前の版が対象としていたのは、Ruby on Railsバージョン4.2でした。今回は、バージョン5.2です。この間にRuby on Railsで起きた変化のうち、特筆すべきはActive Storageの導入です。そこで、独立した章（Chapter 13）を設けて詳しく説明することにしました。

今回の改訂では全体の構成を大きく変更しました。最大の変更点は、テストに関する記述をすべて削除したことです。これまで本書ではテスト重視の姿勢を強調してきましたので、この変化については少し説明しておく必要があると思います。

Ruby on Railsは誕生時からテストのしくみをデフォルトで備えていることを「売り」のひとつとしてきました。筆者が十数年前にRuby on Railsに関心を持った理由のひとつでもあります。ウェブアプリケーション開発においてテストが大切であることは、当時も今も変わりはありません。

しかし、筆者自身が本書を用いた講習を実施したり、本書の読者から感想を聞いたりして気付いたことは、初心者の多くが「テスト」の章で脱落してしまうという事実でした。Ruby on Railsにおけるテストとは、人間が実際にソフトウェアを触って目視で動作確認することではありません。「ソフトウェアの動作確認をするソフトウェア」を作って実行し、成功か失敗かの結果を得ることです。この考え方自体が、初心者には難しいようでした。

筆者自身の体験を振り返っても、プログラミング学習の最初の段階からテストのことを意識していたわけではありません。学習が進み、複雑なソフトウェアを書くようになり、無数の不具合に悩まされる経験を経て、テストの重要性に目覚め

たというのが実態です。はじめからテストのことばかり教えられていたら、プログラミングが嫌いになっていたかもしれません。

以上のような理由から、今回の改訂ではテストの話をほぼすべて省きました。また、これまでの3部構成を4部構成に変え、章の数を9から15に増やしました。と言ってもページ数が大幅に増えたわけではありません。旧版の中身を取捨選択しつつ、新たな内容も加えて再構成しました。

さて、今回の改訂で変わらなかった点も列挙してみたいと思います。第1は、Windowsユーザーへの配慮です。Ruby on Railsの業界ではMacユーザーが多いため、書籍でもネットの記事でもmacOS前提で書かれた記事が目立ちます。しかし、プログラミング初心者の大半はWindowsユーザーです。本書は、Windowsで学習を進められるように書かれています。

第2に、Ruby言語の基礎を扱ったChapter 2です。Ruby on Rails解説書の多くはRubyプログラミングの知識を前提としていますが、本書では60ページ足らずの中に最低限の情報がコンパクトにまとめられていますので、初めてRubyに触れる方でも読み始めることができます。

第3に、書籍全体で1つのアプリケーションを作り上げる構成を取っている点も変化していません。最初にモックアップを作り、データベースを導入し、ログイン・ログアウト機能を加え……と進んでいき、最終的にはメンバーのプロフィール画像のアップロード機能や管理ページまでできあがります。読者の皆さんにはテキストエディタを用いて実際にソースコードを打ち込んでみることを強くお勧めします。まずは、Ruby on Railsによるアプリケーション開発の流れをつかむことが大切です。

なお、本書で作成するサンプルアプリケーションasagaoはオープンソース・ソフトウェアとして公開していますので、本書の学習を終えたあと、無償でこのasagaoを利用したり、改造したり、配布したりできます。自分自身で作りたいものを考

えて、あれこれ工夫しながら作っていくのがプログラミング上達の近道ですので、ぜひ挑戦してみてください。

　本書を通じて、読者の皆様にウェブプログラミングの魅力が伝われば幸いです。

2018年7月

黒田努／佐藤和人

本書の読み方

■対象読者

本書は、「RubyとRuby on Railsを初めて学ぶ」人のための本です。PerlやPHP、Javaなどのプログラミング言語をある程度学んだことがある方を読者対象としています。Rubyの知識や本格的なウェブアプリケーション構築の経験は必要ありません。

「プログラミングの経験はあるが、ウェブアプリケーションとはどんなものかさっぱりわからない」という人から「ほかのフレームワークを使っているが、話題のRuby on Railsを試してみたい」という人までを想定して執筆しています。

■本書の構成

Chapter 1　イントロダクション

Ruby on Railsとはどのようなものか紹介し、そのしくみと特徴について簡単に解説します。また、各種のソフトウェアをインストールしてRuby on Railsに必要な環境を構築します。さらに、最初の一歩として実際にRuby on Railsを動かしてみましょう。

Chapter 2　Ruby言語の基礎を学ぼう

Rubyは初めてという方のために、変数の使い方や制御構造、クラスなどRubyの基本的な文法を解説します。Rubyをすでに学んだことのある方は、このChapterは飛ばしてもかまいません。

Chapter 3　コントローラとビュー

Railsアプリケーションを構成する3つの基本要素のうち、コントローラとビューの使い方を解説します。サンプルソースでは、最初のモックアップ（見本サイト）を作成します。

Chapter 4　データベースとモデル

3つの基本要素のうち、データベースを操作するモデルについて解説します。実際にデータベースがないと学習できませんので、データベースの作り方、テーブルの定義、データの投入についても解説します。

Chapter 5　リソースを扱うコントローラ

リソースベースのルーティングと7つの基本的なアクションについて解説します。モデルをリソースとして扱うコントローラの作り方を紹介し、一覧と詳細表示を行うページを作成します。

Chapter 6　リソースの作成と更新

Chapter 5に引き続き、モデルをリソースとして扱うコントローラについて紹介します。フォームを記述し、フォームから送信されたデータを元に、テーブルのレコードを新規作成、更新、削除する方法について解説します。

Chapter 7　バリデーションと国際化

ユーザーがフォームから送信したデータがアプリケーションの仕様に合致しているかどうかの検証（バリデーション）を行う方法と、エラーメッセージを日本語化する方法について学びます。

Chapter 8　単数リソース

ログイン・ログアウト機能、マイアカウント機能、パスワード変更機能などを作りながら「単数リソース」という重要な概念について学習します。

Chapter 9　Active Recordの活用

コールバック、スコープ、ページネーションなどの重要なActive Recordの側面について学習します。

Chapter 10 モデル間の関連付け

モデルとモデルの間に1対多の関連付けを設定する方法を学びます。また、データベースにおける外部キーの役割について学びます。

Chapter 11 セキュリティと例外処理

フォームから送信された情報を保存する際に、モデルの属性を保護する方法を学びます。また、例外が発生したときのエラーページを自作する方法を解説します。

Chapter 12 アセット・パイプライン

アセット・パイプラインを利用してスタイルシート、JavaScriptプログラム、画像ファイルなどを管理する方法を紹介します。

Chapter 13 ファイルのアップロード

Rails 5.2の新機能Active Storageについて解説します。画像ファイルをローカルディスクやクラウドストレージサービスにアップロードする方法を学びます。

Chapter 14 多対多の関連付け

Ruby on Railsでモデルとモデルの間に多対多の関連付けを行う方法について学習します。

Chapter 15 名前空間

複雑なウェブアプリケーションを開発する際に避けて通れない「名前空間」という概念について解説し、その応用例として本書のサンプルサイトに管理ページの機能を追加します。

■動作確認環境

本書の記述やサンプルスクリプトは次の動作環境で確認しています。

- オペレーティングシステム
  - macOS Sierra（v10.12）
  - macOS High Sierra（v10.13）
  - Windows 10
- Rubyのバージョン：2.5.1
- Ruby on Railsのバージョン：5.2.0

なお、Microsoft社のノートパソコンSurface Laptopや教育機関向けのパソコンに搭載されている「Windows 10 S」では、本書で学習を進めるための各種ソフトウェアをインストールすることができません。

■「ディレクトリ」と「フォルダ」

macOSやWindowsではファイルを分類するための容れ物を「フォルダ」と呼びますが、プログラミングの文脈では一般に「ディレクトリ」という用語が使われま

す。本書では原則として「ディレクトリ」を用います。ただし、macOSのFinderやWindowsのエクスプローラ等のGUI環境における操作を説明するときには「フォルダ」という言葉を使います。

■ソースコードの表記

本書では、ソースコードの部分に次のような表記を用いています。

LISTと書かれた薄い灰色の囲みは、サンプルソースに記述されたソースコードを表します。囲みの上のファイル名は、サンプルソースのファイル名です。ソースコードによっては、ファイル全体ではなく、ファイル中の一部を掲載しています。行頭の1～3桁の数字は行番号を示し、太字部分は変更箇所を示します。

LIST chapter05/config/routes.rb

```
1 Rails.application.routes.draw do
  （省略）
5   1.upto(18) do |n|
6     get "lesson/step#{n}(/:name)" => "lesson#step#{n}"
7   end
8
9   resources :members
10 end
```

RESULTと書かれた部分は、LISTのソースコードを実行して、コマンドプロンプトやブラウザで表示したときの結果を表しています。

RESULT

```
こんにちは、Satoさん
```

枠線の囲み内では、文法の基本的な解説や補足のためのソースコード例を紹介したり、サンプルソースの説明のためにコードの一部を引用します。

```
<%= link_to "Home", root_path, class: "menu" %>
```

■コマンド入力の表記

本書では、ターミナルで実行するコマンドに次のような表記を用いています。

```
$ cd asagao
$ bin/rails s
```

「$」は入力待ち状態を示すためにターミナルに表示される記号（プロンプト）を表します。太字部分だけをコマンドとして入力してください。実際に表示される記号は、環境によって異なります。

次のようにコマンド入力行の下に、結果出力を載せることもあります。

```
$ rails -v
Rails 5.2.0
```

ターミナルでirb（Chaper 2）やRailsコンソール（Chapter 4）を使用する場合は、式の入力と結果を次のように表示します。

```
irb(main):004:0> member.errors.messages
=> {:number=>["can't be blank"]}
```

「irb(main):004:0>」がプロンプトで、太字部分が入力すべき式です。「=>」の右に表示されているのは、式を評価した結果です。

■Macユーザーの方へ

MacのJISキーボードで¥を入力するには、option-¥キーを押してください。

■サンプルソース

本書で紹介するサンプルのソースコードは、オイアクス社のサイトからZIPファイルとしてダウンロードできます。次のURLを参照してください。

https://www.oiax.jp/rails5book

本書の中で、LISTに示した「chapter04/app/models/member.rb」のようなファイル名は、ZIPファイルの中身でのファイルの位置を表しています。「ZIPファイル内のchapter04ディレクトリ→appディレクトリ→modelsディレクトリ→member.rbファイル」という意味です。

chapter01からchapter15までのディレクトリに、Chapter 1からChapter 15までの各章終了時のサンプルソースが含まれています。ただし、Chapter 13のサンプルソースは、chapter13のほかにchapter13-aws、chapter13-gcp、chapter13-azureがあります。chapter13には13.6節終了時のサンプルソースが含まれています。それ以外のディレクトリはChapter 13終了時のものですが、

13.7節で選択したクラウドストレージサービスによりサンプルソースが分岐しています。

■練習の仕方

本書では、「Morning Glory」という架空の草野球チームのサイトを作成するためのソースコードを掲載しています。Chapter 1でMorning Gloryサイトのためのディレクトリをパソコン上に作成し、Chapter 3からChapter 15までそのディレクトリのファイルを修正していきます（Chapter 2はRailsではなくRubyの練習です）。

本書を読みながら実際に自分のパソコンのファイルを編集し、アプリケーションを動かせば、Railsの学習を効果的に進められます。ソースコードはかなり長いため、掲載を省略している部分もあります。省略した部分や入力が面倒な部分は、ダウンロードしたサンプルソースからファイルを探し、自分の練習用ディレクトリにコピーしながら進めるとよいでしょう。

■サンプルソースをそのまま動かすには

Chapter 1、およびChapater 3からChapter 15までは、サンプルソースのディレクトリをまるごとローカルディスクにコピーして動かすこともできます。まるごとコピーしたときは、「bundle install」コマンドを実行し、必要なGemパッケージをインストールします。「bin/rails s」コマンドでアプリケーションを起動してください。

```
$ bundle install
$ bin/rails s
```

また、Chapter 4以降では、「bin/rails s」コマンドを実行する前に「bin/rake db:rebuild」コマンドを実行してください。このコマンドの役割については、「4.3　データの保存」のHINT「db:rebuildタスク」を参照してください。

```
$ bundle install
$ bin/rake db:rebuild
$ bin/rails s
```

目次

Part 1

Ruby on Railsの準備とRubyの基礎

このPartでは、Ruby on Railsの学習を始めるための準備を行います。まず、Ruby on Railsの概要について紹介し、必要なソフトウェアをインストールします。次に、Rubyを一度も学んだことのない方のために、その基本的な文法を解説します。

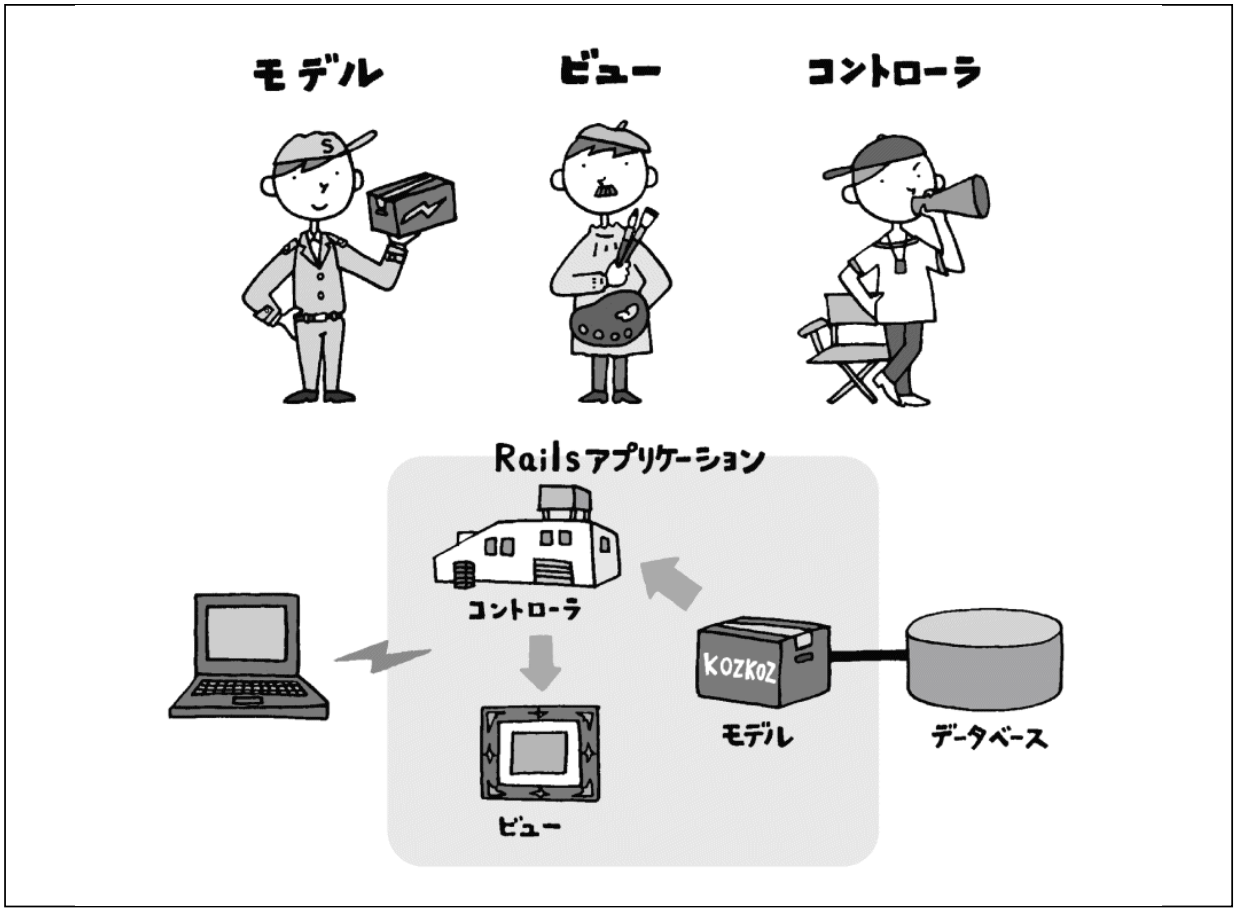
Chapter

1　イントロダクション

　Chapter 1では、Ruby on Railsを使ってウェブアプリケーションを開発するための前提知識と各種ソフトウェアのインストール方法を学びます。

これから学ぶこと

- Ruby on Railsの概要や大まかなしくみ、考え方を理解します。
- プラットフォーム別にRuby on Railsの開発環境を整える手順を学びます。
- Ruby on Railsのアプリケーションを実際に作成し、動かしてみます。



Ruby on Railsは、ウェブアプリケーションを作成するためのフレームワークです。MVCアーキテクチャなどRuby on Railsの基本となるしくみを理解したうえで、アプリケーション作成の準備を行いましょう。

1.1 Ruby on Railsの概要

最初にRuby on Railsとはいったい何なのか、どんな特徴があるのか、といったことを紹介しましょう。

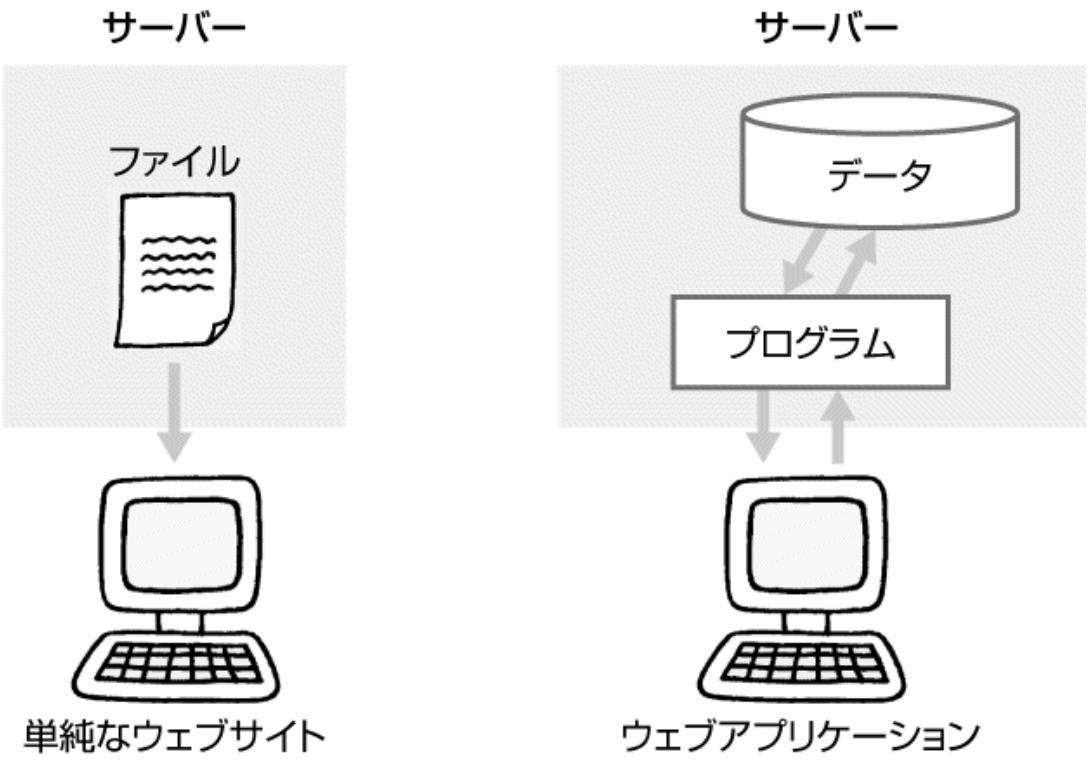
 Ruby on Railsとは

ルビー・オン・レイルズ

Ruby on Railsは、ウェブアプリケーションを開発するためのフレームワークです。本書では略してRailsとも呼びます。

■ウェブアプリケーションとRails

ウェブアプリケーションとは、ウェブ（WWW）を介して何らかのサービスをユーザーに提供するものです。ここで言う「サービス」とは、単にメッセージを表示するだけでなく、ユーザーがメッセージを書き込める掲示板機能や、商品を注文できるショッピング機能などを指します。こうしたサービスを提供するため、ウェブアプリケーションはサーバー上に「データ」を保持し、読み書きしています。サーバー上のファイルをブラウザに送り返すだけの単純なウェブサイトと大きく異なるのはこの点です。

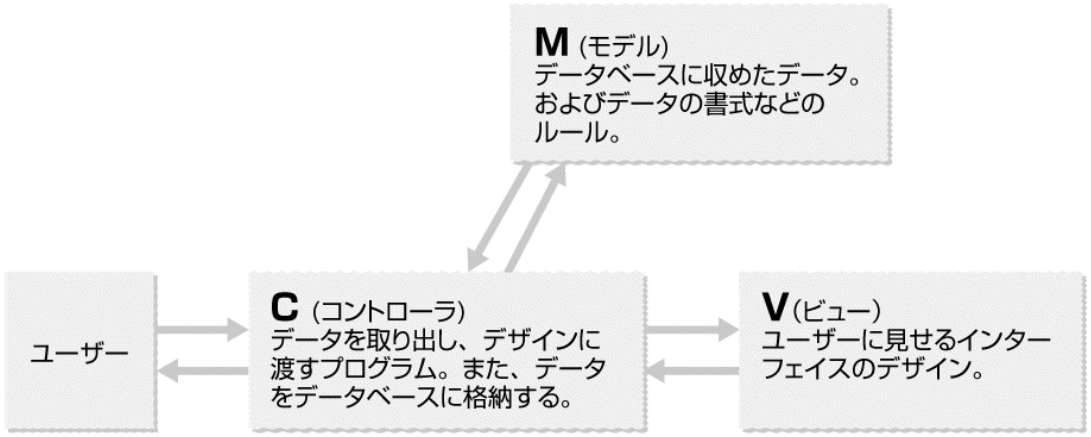


単純なウェブサイトとウェブアプリケーション

ウェブアプリケーションには、3つのものが必要となります。1つ目は、データを保存・操作するしくみで、通常はデータベース管理システム（DBMS）を利用します。2つ目はユーザーに見せるビジュアルなデザインで、HTMLやCSSで記述します。3つ目は、プログラミング言語で書かれたプログラムです。ウェブアプリケーションではJava、PHP、Perlなどさまざまな言語が使われますが、Ruby on Railsで使われる言語はRubyです。

アプリケーションを効率よく開発するためのツール、ライブラリ、設定ファイルなどのセットを**フレームワーク**と呼びます。Railsは、ウェブアプリケーションのためのフレームワークの1つです。ウェブアプリケーションはフレームワークなしでも作れますが、プログラムのコードとSQL文、HTMLがごちゃ混ぜになり、書くのも保守するのも複雑で手間がかかりがちです。

Railsでは、ウェブアプリケーションをすっきりと見通しよく構築できるよう、**MVCアーキテクチャ**と呼ばれる設計法が採用されています。MVCとは、「モデル」、「ビュー」、「コントローラ」の頭文字で、アプリケーションの構成を次の図のように分類することに由来しています。



MVCアーキテクチャ

Railsは、MVCアーキテクチャを備えたウェブアプリケーション開発フレームワークの中で、現在最も有名なものの1つです。

Ruby on Railsのウェブサイト

Railsのウェブサイトは次のURLです。

Ruby on Rails（英語）
https://rubyonrails.org/

Railsに関する最新情報はブログ「Riding Rails」で、入門者のためのガイドは「Ruby on Rails ガイド」で読めます。

Riding Rails（英語）
https://weblog.rubyonrails.org/
Ruby on Rails Guides（英語）
http://guides.rubyonrails.org/
Ruby on Rails ガイド（日本語）
https://railsguides.jp/

クラス別・メソッド別の詳しい情報を調べるには「Ruby on Rails API」が便利です。

Ruby on Rails API（英語）
http://api.rubyonrails.org/

Ruby on Railsの開発者

Railsを作り出したのは、アメリカ在住のデンマーク人プログラマDavid Heinemeier Hansson（デビット・ハイネマイヤ・ハンソン）氏で、よく「DHH」と呼ばれます。彼が37signals社（現Basecamp社）で携わっていたソフトウェア「Basecamp」のフレームワークを取り出して公開し、オープンソースソフトウェアとしたのがRailsです。

DHH氏のサイト

http://david.heinemeierhansson.com/

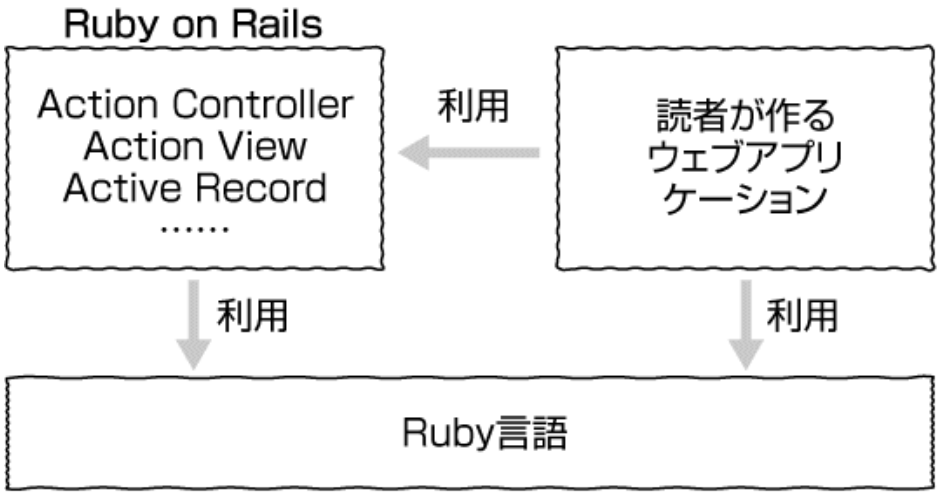
日経BP、ITproによるDHH氏へのインタビュー

https://tech.nikkeibp.co.jp/it/article/NEWS/20060620/241346/

RubyとRails

ルビー

Railsは、プログラミング言語**Ruby**で記述され、RubyのGemパッケージ（後述）として提供されています。また、Railsを使ってウェブアプリケーションを書くときもRubyで記述します。



RubyとRails

Rubyの特徴は、純粋で高機能なオブジェクト指向言語であること、シンプルできれいなコードを書けることです。

Rubyでは、プログラマがクラスを設計できるだけでなく、文字列や数値などもオブジェクトとして実装され、既存のオブジェクトを簡単に拡張できるしくみを備えています。難しいことを考えなくても、オブジェクト指向のプログラムを自然に書けるようになっています。

Rubyは、やさしく学べる言語です。ほかのプログラミング言語を学んだことがある人なら、数日から数週間で自在にRubyプログラムが書けるようになるでしょう。また、Rubyを使うと簡潔で読みやすいコードを書くことができます。Railsの開発者David Heinemeier Hansson氏は、「美しいコードを書けるからRubyを選んだ」と述べています。

Rubyの開発者

プログラミング言語Rubyを作ったのは、日本人のプログラマまつもとゆきひろ氏（通称Matz）です。1995年に初めて公開され、現在では世界中で利用される言語に成長しています。また、世界中の開発者によってオープンソース方式で強化が続けられています。まつもと氏はRubyで最も重視しているのは「プログラミングを楽しむこと」と述べています。

Rubyのバージョン

本書では、Ruby 2.5とRails 5.2を使ってRailsアプリケーションの作成を解説します。RubyとRailsのこれまでのバージョンについて簡単に紹介しましょう。

2018年5月現在、公式にメンテナンスされているRubyのバージョンは次の3つです。毎年12月25日に新しいバージョンのRubyがリリースされる慣例になっています。

- 2.3（2015年12月25日リリース）
- 2.4（2016年12月25日リリース）
- 2.5（2017年12月25日リリース）

Railsのバージョン

Railsが最初に公開されたのは、2004年のバージョン0.5です。2005年には、初のメジャーリリース1.0が公開されました。Railsの最初の大きな飛躍となったのは、「リソース」という概念が導入されたバージョン1.2です（リソースについてはChapter 5を参照）。次の飛躍となった

のはバージョン3で、Rubyで書かれた別のフレームワークMerbとの統合が行われました。これにより、コンポーネントの組み合わせが柔軟になり、パフォーマンスが向上しました。

その後もRuby on Railsはウェブ業界のトレンドを取り入れつつ急速に進化を遂げています。本書の記述は、2018年4月に公開されたバージョン5.2に基づいています。

Railsのバージョン

バージョン番号	日付	バージョン番号	日付
1.0	2005/12/13	3.1	2011/04/31
1.1	2006/03/28	3.2	2012/01/20
1.2	2007/01/19	4.0	2013/06/25
2.0	2007/12/07	4.1	2014/04/08
2.1	2008/06/01	4.2	2014/12/19
2.2	2008/11/21	5.0	2016/06/30
2.3	2009/03/16	5.1	2017/05/10
3.0	2010/08/29	5.2	2018/04/09

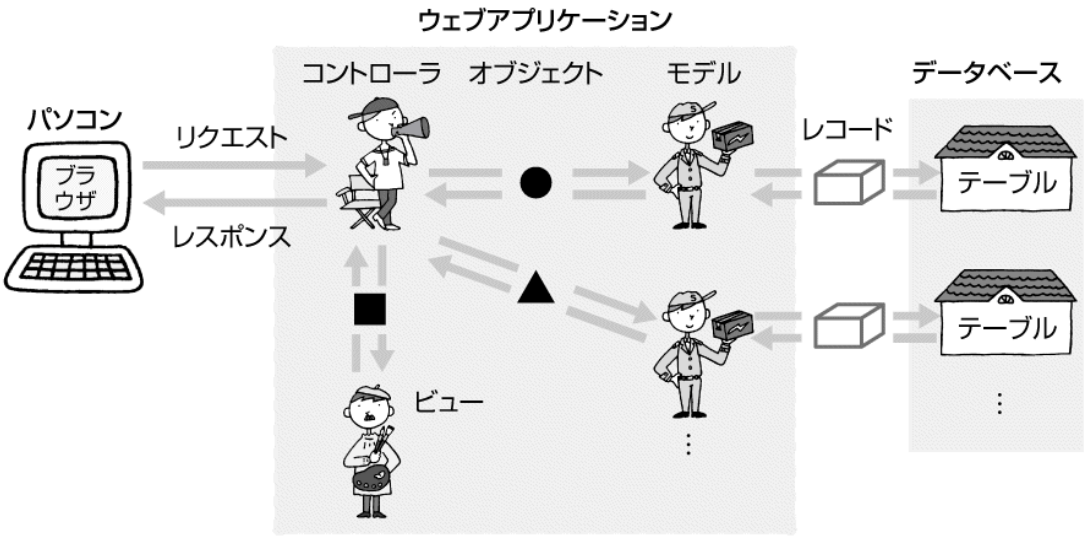
Rails 5.2では、Amazon S3、Google Cloud Storage、Microsoft Azure Storageなどのクラウドストレージサービスにファイルをアップロードするための、Active Storageという新しい機能が導入されました。本書ではChapter 13でこの機能について解説します。

MVCと設計哲学

Railsの基本をなすMVCアーキテクチャと設計哲学について紹介しましょう。

RailsのMVC

Railsでは、データベースのデータを扱うオブジェクトを**モデル**と呼びます。**コントローラ**はモデルからのデータを受け取ってビューに渡します。HTMLにデータを埋め込むためのテンプレートが**ビュー**です。



モデル、ビュー、コントローラの役割

モデルとコントローラの役割分担に注目してください。モデルは倉庫の荷物を出し入れする「倉庫番」のような存在です。ウェブアプリケーションでは、倉庫はデータベースの「テーブル」に、荷物は「レコード」に相当します。また、モデルは倉庫番として不正な荷物が倉庫に紛れ込むのを監視しています。アプリケーションが、不正なレコードをテーブルに挿入しようとしても、モデルはそれを受け付けません。

コントローラは、ウェブアプリケーションからモデルとビューを分離した残りの部分であり、いわば「何でも屋」です。その役割は、ブラウザからの入力を受け取る、データをモデルに要求する、データの追加や変更をモデルに指示する、適切なビューを選ぶ、データをビューに渡すなど、さまざまです。コントローラが実行する具体的な仕事のことを「アクション」と呼びます。

■設計哲学

Railsには2つの重要な設計哲学があります。1つは「DRY」、もう1つは「設定より規約」です。Railsの簡潔さと効率のよさを支えているのは、この2つの哲学です。

DRY（Don't Repeat Yourself、繰り返しを避けよ）

DRYは「DRY原則」とも呼ばれます。Railsを使ったウェブアプリケーションでは、同じことを繰り返し記述するのは避けなければなりません。同じことをソースコードや設定ファイルの中で繰り返し記述するのは無駄ですし、仕様変更やバグフィックスのときに一部を変更し忘れる

可能性が高くなります。DRYを意識することで、効率よく品質のよいアプリケーションが作成できます。

設定より規約（Convention over Configuration）

「規約」とは、言い換えれば「デフォルトの設定」です。あらかじめ用意された規約に従ってアプリケーションを開発することで、記述量を大幅に減らせます。たとえば、モデルには命名規約があり、テーブル名をmembersのように複数形にすると、モデルのクラス名は単数形のMember、クラスを記述するファイル名はmember.rbとなります。決まりきった手順に従うことで、余計な設定を記述する必要がなくなり、プログラマはコードに集中できるようになります。こうした効果について、David Heinemeier Hansson氏は「制約が自由をもたらす」と述べています。

Railsの構成と機能

Railsは複数のコンポーネントで構成されています。その機能を少し詳しく見てみましょう。

■Railsのコンポーネント

Railsの実体は、コンポーネント（Rubyで書かれたライブラリ）の集合体です。基本となる3つのコンポーネントの名前を覚えましょう。

- Active Record——モデル
- Action View——ビュー
- Action Controller——コントローラ

Railsのコンポーネントは、Gemパッケージとして提供されています。Gemパッケージとは、Rubyのパッケージ・マネージャである**RubyGems**が管理するライブラリです。RailsのGemパッケージは、次のように構成されています。

- Action Pack

    - Action Controller——コントローラ
    - Action Dispatch——ルーティング
- Action View——ビュー
- Action Mailer——電子メール送信
- Active Model——モデル
- Active Record——データベースと結び付いたモデル
- Active Job——プログラムの非同期実行
- Active Support——共有ライブラリ集
- Active Storage——クラウドストレージサービスへのアップロード
- Railties——railsコマンドなどのユーティリティ

コントローラは、Action Packというパッケージに含まれています。モデルは、抽象的なモデルであるActive Modelと、データベースとやり取りするActive Recordの2つのパッケージからなります。

Active Supportは、Rubyの文字列、数値、時刻などのクラスを拡張するものです。Action MailerとActive Jobについては、本書では扱いません。

■便利な特徴

Railsには、ウェブアプリケーション作成の効率を高める便利な機能がたくさん用意されています。

ルーティング

&や=がたくさん付いた汚いURLではなく、きれいなURLが使えます。たとえば、/members/1のようなわかりやすいパスで、特定の会員の詳細情報ページを呼び出せます。データをリソースとして扱うことで、URLのパスとアプリケーションの関係がさらにすっきりします（解説はChapter 3、Chapter 5）。

テンプレート

Railsでは、ビューをHTMLのテンプレートで作成します。HTML の中に<% ... %>を埋め込んで、Rubyのコードを記述したり、変数の値をそのまま出力したりできます（解説はChapter 3）。

マイグレーション

データベースのテーブルを定義するためにマイグレーション機能が用意されており、SQL文を書かずに済みます。データベースの作成や変更は、railsコマンドで簡単に行えます（解説はChapter 4）。

レコードの操作

データベースのテーブルにbirthdayというカラムがあるとすると、@member.birthdayのような書き方でカラムの値を取り出したり、新しい値を入れたりできます。レコードの取り出し、保存、削除といった操作のためのメソッドも用意されています（解説はChapter 4、5、6）。

バリデーション

テーブルにレコードを保存するときに、空の値を禁止したり、値が決まった書式に従うように指定したりできます。そうした指定を簡単な記述で書けます（解説はChapter 7）。

国際化

ブラウザに表示するテキストを言語別に用意することで、日本語や英語など複数の言語に対応したサイトを作成できます（解説はChapter 7）。

テスト

Railsには、自動テストのしくみがはじめからサポートされています。自然な形でテスト駆動型開発を実践できます（本書では解説しません）。

セッション

セッション機能を使ってサイトにアクセスしたユーザーの状態を保存できます。セッションを利用すれば、サイトにログイン機能を加えるのも簡単です（解説はChapter 8）。

モデル間の関連付け

2つのテーブルを結び付けたときは、自然な形でそれぞれのモデルにアクセスできます。たとえば会員とブログ記事を関連付けたときは、ある会員の投稿した記事を@member.entriesのように表現できます（解説はChapter 10、Chapter 14）。

クラウドストレージサービスとの連携

Amazon S3、Google Cloud Storage、Microsoft Azure Storage等のクラウドストレージサービスへ簡単にファイルをアップロードし、ファイルを配信できます（解説はChapter 13）。

 本書のサンプルアプリケーション

本書では、Chapter 3からChapter 15までで実際にウェブアプリケーションを作りながらRailsの機能を解説していきます。作成するアプリケーションは、草野球チームのサイトMorning Gloryです。Morning Gloryは「朝顔」の意味で、早朝に練習を行うことから名付けたチーム名です。Chapter 15で最終的に完成するMorning Gloryのサイトは次のようになります。

トップページや「ニュース」ページは、チームの活動を外部に向けて紹介します。

Morning Gloryトップページ

ログイン機能があり、会員はログイン名とパスワードを入力してログインすると、会員情報ページなど専用のページを表示できます。

会員情報の詳細ページ

管理者に指定した会員は、管理ページを表示でき、会員や記事のデータの作成や更新ができます。

会員情報の更新ページ

簡単なブログ機能もあり、ログインした会員は自分のブログを作成、編集できます。ブログの記事は、ログインしなくても、誰でも読めるようにしています。

ブログページ

このサンプルアプリケーションのソースコードは、本書のサポートサイトからダウンロードできます。ZIP形式のファイルを展開するとchapter3、chapter4等のディレクトリが現れます。各Chapterの学習を終えた段階のMorning Gloryサイトのソースコードが収められています。chapter15のディレクトリにあるものが、最終的な完成版です。

Morning Gloryサイトのソースコードは、フリーソフトウェアとして無償で公開します。自分のサークルのための会員制サイトや、企業の情報共有システムなどのベースとして活用してください。

1.2 Rails開発環境の構築

自分のパソコンでRailsを学習していくために、必要なソフトウェアをインストールしましょう。手順はmacOSとWindowsとで異なります。また、Windowsの場合にはWindowsの種類により2つの選択肢があります。

環境構築に必要なもの

Railsをパソコン上で動かし、学習していくためには、次のソフトウェアが必要です。必要になるソフトウェアは、どれもインターネット上から無料で入手できます。

■macOS/Windows共通

Ruby

Railsを動かすための、プログラミング言語Rubyの実行環境です。本書では、バージョン2.5を使います。

rbenv/ruby-build

複数のバージョンのRubyをインストールし、必要に応じて切り替えて利用するためのソフトウェアです。

RubyGems

RubyGemsはRuby言語のためのパッケージ・マネージャです。Railsを含め、Ruby関連のソフトウェアのインストールやアップデートに使います。Rubyをインストールすると、RubyGemsも同時にインストールされます。

Ruby on Rails、およびその他のGemパッケージ

Ruby on RailsはGemパッケージとしてインストールします。Railsは複数のGemパッケージから構成されており、それぞれのGemパッケージがまた別のパッケージに依存しています。そうした多数のGemパッケージはまとめてインストールできます。

SQLite3

学習用のデータベース管理システムとして、本書ではSQLite3を利用します。

■macOS

Xcodeとコマンドライン・デベロッパ・ツール

Rubyのインストールにはソースのコンパイルが必要なので、macOSの開発環境であるXcodeとコマンドライン・デベロッパ・ツールを用意します。

Homebrew

HomebrewはmacOS用のパッケージ管理ツールです。

■Windows

Windows Subsytem for Linux（WSL）

Windows Subsytem for Linux（WSL）は、Windows上でLinux向けのソフトウェアを実行するためのしくみです。ただし、64ビット版のWindows 10でしか利用できません。また、教育市場向けの低価格PCや「Surface Laptop」のOSとして使われているWindows 10 Sでは利用できません。

Ubuntu

Ubuntu（ウブントゥ）は、Linuxをベースとしたオペレーティングシステム（OS）です。本書ではWSL環境にUbuntu 18.04をインストールしてRails開発環境を構築します。

MSYS2/MinGW

MSYS2/MinGWは、Windows上でbash、tar、gitなどのUnix向けのツール群を使用するためのソフトウェアパッケージです。Windows Subsytem for Linux（WSL）が利用できない場合は、こちらを利用します。

■サポートサイト

RubyとRailsのインストールや本書のサンプルプログラムに関する情報は、オイアクス社のサポートサイトで公開していきます。ソフトウェアのバージョンアップにより本章のインストール手順が古いものになった場合などは、新しい情報を随時掲載します。次のURLを参照してください。

https://www.oiax.jp/rails5book

macOSにインストールするときの注意

macOSでRubyとRailsのインストール作業を進めるときには、新しいコマンドをインストールするたびに、ターミナルを新しく開き直すのがお勧めです。macOSには最初からRubyとRubyGemsがインストールされており、ターミナルが元からあるコマンドと新しくインストールしたコマンドを取り違えることがあるからです。インストール作業中にうまくいかないことがあったら、ターミナルを新しく開いてやり直してみましょう。

macOSでの開発環境構築

■Xcodeのインストール

本書では、Homebrewを使ってmacOSにRubyをインストールします。Homebrewを利用するにはXcodeとコマンドライン・デベロッパ・ツールが必要になります。まずXcodeをインストールしましょう。

XcodeはAppleのApp Storeでダウンロードできます。App Storeを開いて、右上の検索ボックスで「Xcode」と入力してください。検索結果からXcodeのページを開いて、［入手］

→［Appをインストール］ボタンを押せばダウンロードとインストールが始まります（このとき、Apple IDとパスワードを聞かれることがあります）。

Xcodeのダウンロード

インストールが済んだら、LaunchpadでXcodeを開いてライセンスの認証を行ってください。

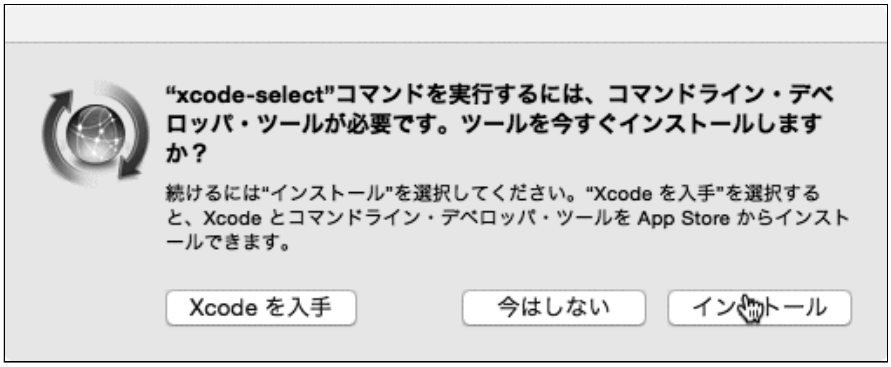
■コマンドライン・デベロッパ・ツールのインストール

RubyとRailsのインストールには、Xcodeだけでなくコマンドライン・デベロッパ・ツールもインストールする必要があります。このツールのインストールは、必ずHomebrewのインストールの前に行ってください。順番を間違えると、Railsに必要なモジュールがコンパイルできなくなります。

macOSのターミナルを開いて、次のコマンドを実行してください。ターミナルは［アプリケーション］フォルダの［ユーティリティ］フォルダの下にあります。これから何度も使うので、ターミナルのアイコンをDockに入れておくとよいでしょう。

```
$ xcode-select --install
```

表示される画面で［インストール］ボタンを押せば、インストールできます。



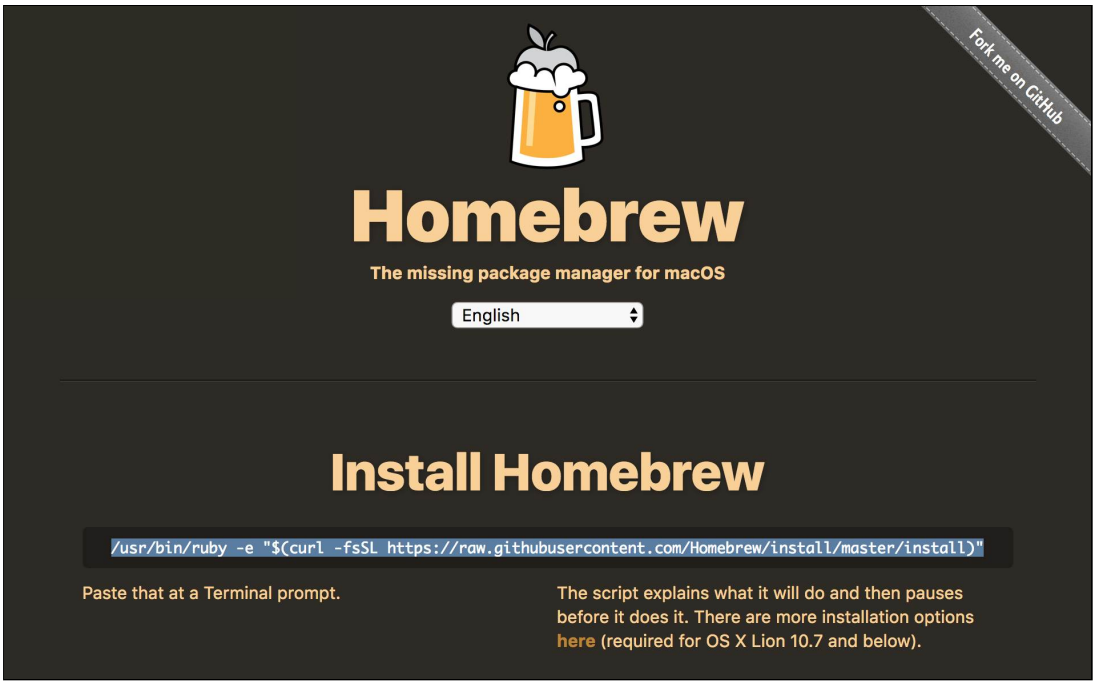
コマンドライン・デベロッパ・ツールのインストール

■Homebrewのインストール

HomebrewでRubyをインストールするには、まずHomebrewのサイトにアクセスしてください。

Homebrew
https://brew.sh/

このページの中で「Install Homebrew」の下に書かれているインストール用のコマンドをコピーしてください。



Homebrewのインストールコマンド

ターミナルを開き、コピーしたコマンドを貼り付けて実行すれば、Homebrewをインストールできます。途中でパスワードを求められるので入力してください。

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

■rbenvとruby-buildのインストール

Homebrewを使ってrbenvをインストールします。rbenvは、複数のバージョンのRubyをインストールして切り替えるためのツールです。また、rbenvに必要なruby-buildもインストールします。

```
$ brew install rbenv ruby-build
```

ターミナルを開くたびにrbenvを初期化するコマンドが自動的に実行されるように、次のコマンドで.bash_profileファイルに「eval "$(rbenv init -)"」を書き加えます。

```
$ echo 'eval "$(rbenv init -)"' >> ~/.bash_profile
```

次のコマンドで、.bash_profileファイルに加えたこのコマンドを実行します（ターミナルを新しく開いても同じ効果を得られます）。

```
$ source ~/.bash_profile
```

■Rubyのインストール

次のコマンドでRuby 2.5.1をインストールします。

```
$ rbenv install 2.5.1
```

「rbenv versions」でインストールされているRubyの一覧を表示できます。

```
$ rbenv versions
* system (set by /Users/taro/.rbenv/version)
2.5.1
```

次のコマンドでRuby 2.5.1が使えるようになります。

```
$ rbenv global 2.5.1
```

ターミナルを開き直してから、Rubyのバージョンを確認してみましょう。

```
$ ruby -v
ruby 2.5.1p57 (2018-03-29 revision 63029) [x86_64-darwin16]
```

本書の刊行後にRuby 2.5.2が出たとしましょう。そのときは次の3つのコマンドを実行すればRuby 2.5.2が使えるようになります。

```
$ brew upgrade ruby-build
$ rbenv install 2.5.2
$ rbenv global 2.5.2
```

■Railsのインストール

RailsおよびRailsに必要なGemパッケージを、RubyGems（gemコマンド）でインストールしましょう。本書のサンプルが動作するように、「--version "5.2.0"」というオプションを付けてバージョン5.2.0がインストールされるようにしてください。

```
$ gem install rails --version "5.2.0" -N
```

なお、-Nオプションを付けるのは、ドキュメントの生成を省略してインストールに要する時間を短縮するためです。

Gemパッケージのインストールが終わると、railsコマンドを使えるようになります。ターミナルを開き直してから、Railsのバージョンを確認してみましょう。

```
$ rails -v
Rails 5.2.0
```

railsディレクトリの作成

ターミナルで次のコマンドを実行し、ホームディレクトリの下にrailsというディレクトリを作成します。チルダ記号（~）はホームディレクトリを示します。

```
$ mkdir ~/rails
```

以上で、macOSにおいてRailsの学習を始める準備が整いました。「テキストエディタの選択」の項に進んでください。

仮想化ソフトウェアの利用という選択肢

macOSに直接Rubyをインストールして Rails開発を行うという方法のほかに、VirtualBoxやDockerなどの仮想化ソフトウェアを利用してUbuntuやCentOSなどのLinux系OSをインストールし、そこにRubyをインストールするという選択肢もあります（本書では解説しません）。やや導入手順が複雑ですが、本番環境とまったく同じOS上で開発を行えるという利点があります。

Windowsでの開発環境構築

■Windowsユーザー名に関する注意点

本題に入る前に、プログラミング初心者が陥りやすい罠について説明します。Windowsのユーザー名（正確には「ユーザーアカウント名」）に漢字、ひらがな、カタカナが含まれている場合、Ruby on Railsの開発環境構築やプログラミングにおいてさまざまな支障が生じます。

もし読者がそのようなユーザー名を使用している場合は、英数字のみを用いたユーザー名に変更するか、英数字のみの名前を持つユーザーを新たに作ってください。なお、本書ではWindowsユーザー名が「oiax」であるとして説明を行っていきます。

■Windowsのバージョン別の選択肢

Windows上にRailsの開発環境を構築する方法として、本書では2つの選択肢を紹介します。

- WSL/Ubuntu
- MSYS2/MinGW

WSL/UbuntuとMSYS2/MinGWを比較すると前者のほうがRails開発に適しているのですが、WSLの動作条件がWindows 10の64ビット版（ただし、Windows 10 Sを除く）であるため、Windows 10 S、Windows 10の32ビット版、Windows 8、Windows 7をお使いの方は必然的にMSYS2/MinGWを選択することになります。

なお、Windows 10 Sとは教育市場向けの低価格PCや「Surface Laptop」のOSとして使われている特別なバージョンのWindows 10です。

Windows 10が64ビット版か32ビット版か調べる方法

Windowsのスタートメニューの「歯車」アイコンをクリックします。「Windowsの設定」が開いたら、［システム］の［バージョン情報］を選択します。

「デバイスの仕様」セクションの「システムの種類」の値が「64」で始まっていれば64ビット版です。そうでなければ32ビット版です。

本書では扱いませんが、WSL/UbuntuとMSYS2/MinGWのほかにも、VirtualBoxやDockerなどの仮想化ソフトウェアを利用してUbuntuやCentOSなどのLinux系OSをインストールするという選択肢もあります。やや導入手順が複雑ですが、本番環境とまったく同じOS上で開発を行えるという利点があります。

■WSL/Ubuntuでの環境構築

WSL/Ubuntuのインストール

WSLとUbuntuをインストールする手順の概要を箇条書きでまとめます。

- コントロールパネルを開き、［プログラム］→［プログラムと機能］→［Windowsの機能の有効化または無効化］に進みます。
- ［Windows Subsystem for Linux］のチェックボックスをオンにして［OK］ボタンをクリックします。
- Windowsを再起動します。
- ［スタート］ボタンを押して、［Microsoft Store］を開きます。

- 検索ボックスに「ubuntu」と入力して現れる選択肢の中から［Ubuntu 18.04］を選びます（注）。
- ［入手］ボタンを押します。
- ［起動］ボタンを押します。
- 黒い背景のウィンドウが開き、「Installing, this may take a few minutes...」というメッセージが表示されます。
- 「Enter new UNIX username:」と表示されたら、Ubuntuで使用するユーザー名を入力します。ただし、ユーザー名は英小文字で始まり、英小文字と数字だけで構成してください。長さは3〜8文字の範囲に収めることをお勧めします。本書では、ユーザー名に「oiax」を選んだものと仮定します。
- 「Enter new UNIX password:」と表示されたら、Ubuntuで使用するパスワードを入力します。
- 「Retype new UNIX password:」と表示されたら、同じパスワードを再入力します。
- 末尾にドル記号（$）のある行が表示されたら、黒い背景のウィンドウを閉じます。

（注）2018年5月17日現在、Microsoft Storeから入手できるUbuntuには「Ubuntu」と「Ubuntu 18.04」があります。前者は2016年4月リリースのUbuntu 16.04を指しています。このバージョンは2021年4月までサポートされます。Microsoft Storeのラインナップは随時更新されるので、読者の皆さんが実際に訪れたときには別のリストが現れるかもしれません。その際は、アプリの説明をよく読んでバージョン番号に「18.04」と書かれているものを選んでください。また、本書のサポートサイトに注意書きが追加されていないかどうかも確認してください。

Ubuntuターミナル

［スタート］ボタンを押して［Ubuntu 18.04］を選択すると、インストール時に現れたのと同様の黒い背景のウィンドウが開きます。これをUbuntuターミナル、あるいは単に「ターミナル」と呼びます。

ターミナルの左端には「oiax@DESKTOP-PKOPPJ5:~$」のような文字列が表示されています。ただし、「DESKTOP-PKOPPJ5」の部分は環境によって異なります。これをプロンプトと呼びます。

Ubuntuの更新

ターミナルで次の2つのコマンドを順に実行し、Ubuntuを更新します。パスワードの入力を求められたら、Ubuntuのインストール時に設定したパスワードを入力してください。

```
$ sudo apt-get update
$ sudo apt-get -y upgrade
```

初回更新時はかなり長い時間がかかるかもしれません。定期的にUbuntuを更新することをお勧めします。

Rubyのインストール

ターミナルで次の4つのコマンドを順に実行し、Ruby 2.5をインストールします。

```
$ sudo add-apt-repository -y ppa:brightbox/ruby-ng
$ sudo apt-get -y install ruby2.5 ruby2.5-dev
$ echo 'export GEM_HOME=~/.gem' >> ~/.bashrc
$ source ~/.bashrc
```

Rubyのバージョンを確認します。

```
$ ruby -v
ruby 2.5.1p57 (2018-03-29 revision 63029) [x86_64-linux]
```

本書の執筆時点では、2.5系の最新版であるRuby 2.5.1がインストールされます。今後、2.5系の新しいバージョンがリリースされれば、バージョン2.5.2以上のRubyがインストールされます。本書の学習に影響はないはずですが、気になる方は本書のサポートサイトを参照してください。

Ubuntuでrbenv/ruby-buildを使うには

Ubuntuでrbenvとruby-buildを使ってRuby 2.5.1をインストールすることもできます。ターミナルで次のコマンド群を順に実行してください。

```
$ sudo apt-get -y install rbenv ruby-build
$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
$ echo 'eval "$(rbenv init -)"' >> ~/.bashrc
$ source ~/.bashrc
$ mkdir -p ~/.rbenv/plugins
$ cd ~/.rbenv/plugins
$ git clone https://github.com/rbenv/ruby-build.git
$ cd
$ rbenv install 2.5.1
$ rbenv global 2.5.1
```

実行するコマンドの数が多く、インストール完了までの時間も大幅に長くなりますが、複数のバージョンのRubyをインストールして切り替えられるようになります。

SQLite3のインストール

データベース管理システムSQLite3をインストールします。ターミナルで次のコマンドを実行してください。

```
$ sudo apt-get install sqlite3 libsqlite3-dev
```

Railsのインストール

ターミナルで次のコマンドを実行します。これはRailsが依存するGemパッケージnokogiriのインストールに必要です。

```
$ sudo apt-get install -y build-essential patch ruby-dev zlib1g-dev liblzma-dev
```

Rails 5.2.0をインストールします。

```
$ gem install rails --version=5.2.0 -N
```

学習用ディレクトリの作成

まず、Windowsのデスクトップにrailsというディレクトリを作成してください。そして、Ubuntuターミナルで次のコマンドを実行します。

```
$ ln -s /mnt/c/Users/oiax/Desktop/rails ~/rails
```

ただし、oiaxの部分は、自分のユーザー名で置き換えてください。コマンドlnによりシンボリックリンクが作られ、その結果Windows側のデスクトップにあるrailsディレクトリがWSL/Ubuntu側のホームディレクトリ直下のrailsディレクトリとして見えるようになりました。チルダ記号（~）はホームディレクトリを示します。

以上で、WSL/Ubuntu環境においてRailsの学習を始める準備が整いました。「テキストエディタの選択」の項に進んでください。

WSL/Ubuntu利用上の重要な注意事項

WSL/Ubuntuを利用するうえで十分に気をつけなければならないのは、Windows側のソフトウェア（たとえば、後述するテキストエディタAtomやVS Code）でWSL/Ubuntu側のディレクトリやファイルを操作すると壊れる危険性がある、ということです。

しかし、本文中で説明した手順でシンボリックリンクを設定した結果、WSL/Ubuntu側のホームディレクトリ直下のrailsディレクトリはWindows側のソフトウェアで操作できるようになっています。逆に言えば、このディレクトリ以外の場所にあるファイルをWindows側のソフトウェアで触ってはいけません。

たとえば、WSL/Ubuntu側のホームディレクトリに.bashrcという設定ファイルがありますが、これをAtomやVS Codeで開いて編集してはなりません。必ず、WSL/Ubuntu側のソフトウェア（後述するnanoやVim等のスクリーンエディタ）で編集してください。

■MSYS2/MinGWでの開発環境構築

MSYS2/MinGWとは

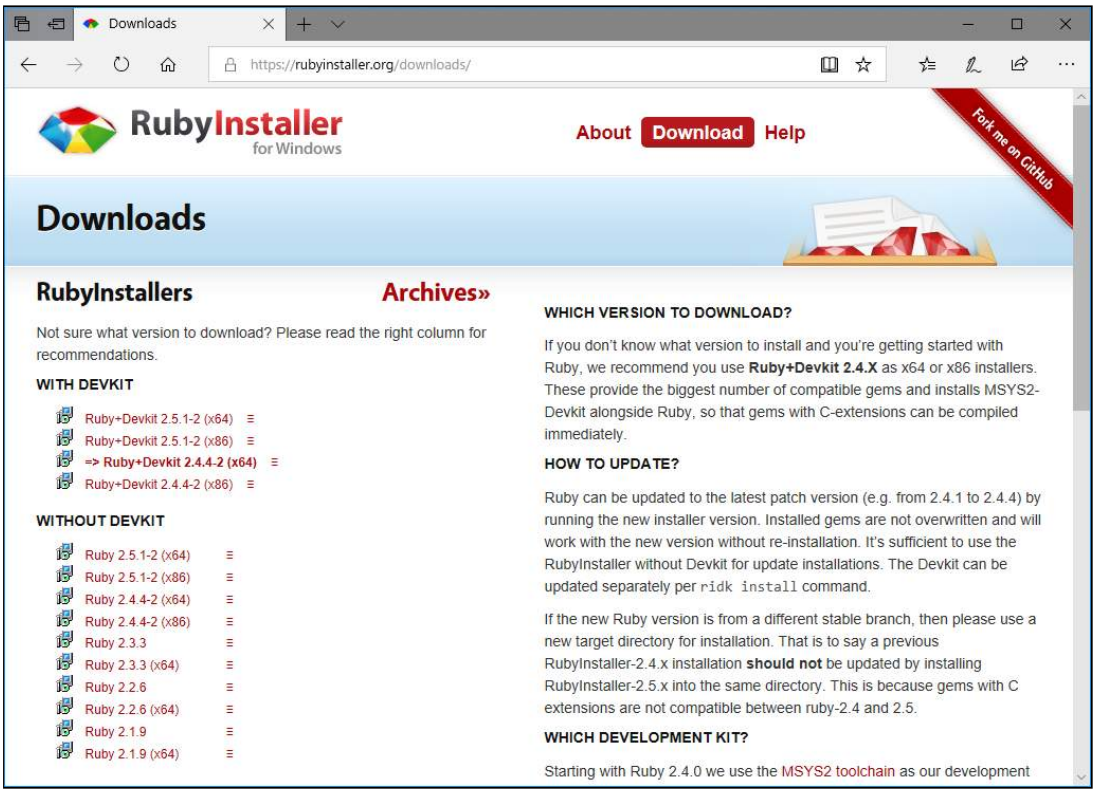
MSYS2（Minimal SYStem 2）とMinGW（Minimalist GNU for Windows）は、Windows上でLinuxの各種ソフトウェア・ライブラリを動作させるためのパッケージです。それぞれ独立したパッケージですが、MSYS2にはMinGWが同梱されます。本書では、両者を合わせてMSYS2/MinGWと呼びます。後述するRubyInstallerでRubyをインストールすると、自動的にMSYS2/MinGWのインストールが始まります。

Rubyのインストール

MSYS2/MinGW環境で動くRubyプログラムは、RubyInstallerでインストールします。ブラウザで次のURLを開いてください。

RubyInstaller for Windows

https://rubyinstaller.org/downloads/



RubyInstallerのダウンロード

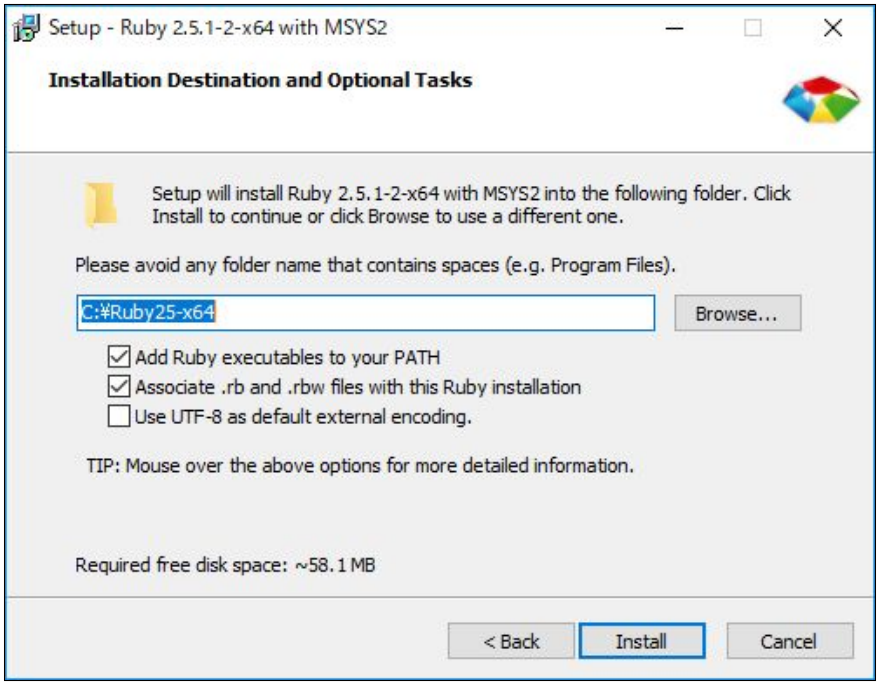
「WITH DEVKIT」セクションから適切なバージョンを選んでダウンロードします。本書執筆時点（2018年7月）では、次の4つの選択肢があります。

- Ruby+Devkit 2.5.1-2 (x64)
- Ruby+Devkit 2.5.1-2 (x86)
- Ruby+Devkit 2.4.4-2 (x64)
- Ruby+Devkit 2.4.4-2 (x86)

バージョン2.5系列とバージョン2.4系列の最新版が並んでいます。本書執筆時点ではバージョン2.4系列の利用が推奨されていますが、本書の学習においてはバージョン2.5系列を選んでください。

また、かっこの中に「x64」と書いてあるほうが64ビット版、「x86」と書いてあるほうが32ビット版です。お使いのWindowsの種類に応じて選んでください。

ダウンロードした実行ファイルを起動し、［I accept the License］をチェックして［Next］ボタンを押します。次の画面が表示されたら、そのまま［Install］ボタンを押してください。



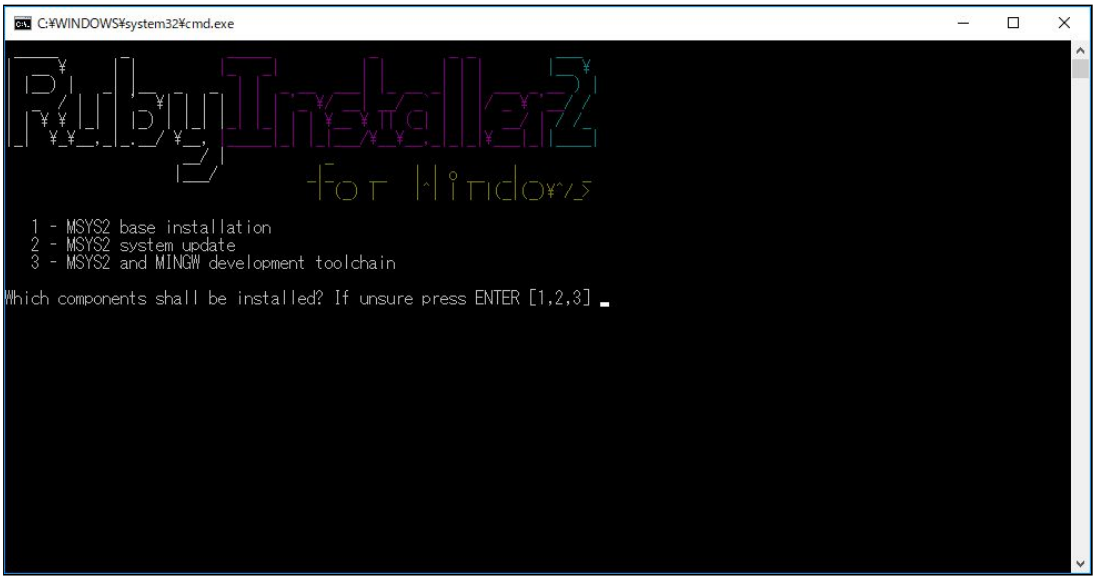
RubyInstallerの設定画面

管理者権限が必要

Windowsでインストール作業を行うには、ユーザーに管理者権限が必要です。現在のアカウントが管理者ではない場合は、管理者アカウントに切り替えるか、管理者である人にインストールしてもらう必要があります。

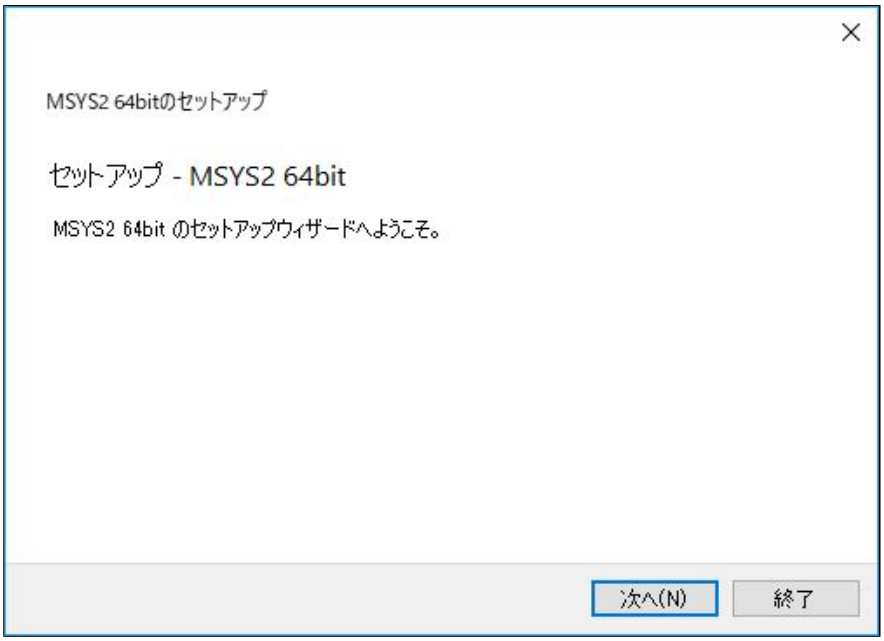
MSYS2/MinGWのインストール

Rubyのインストールが完了したら、そのまま［Finish］ボタンを押してください。すると、コマンドプロンプトが起動して、次のような画面が現れます。



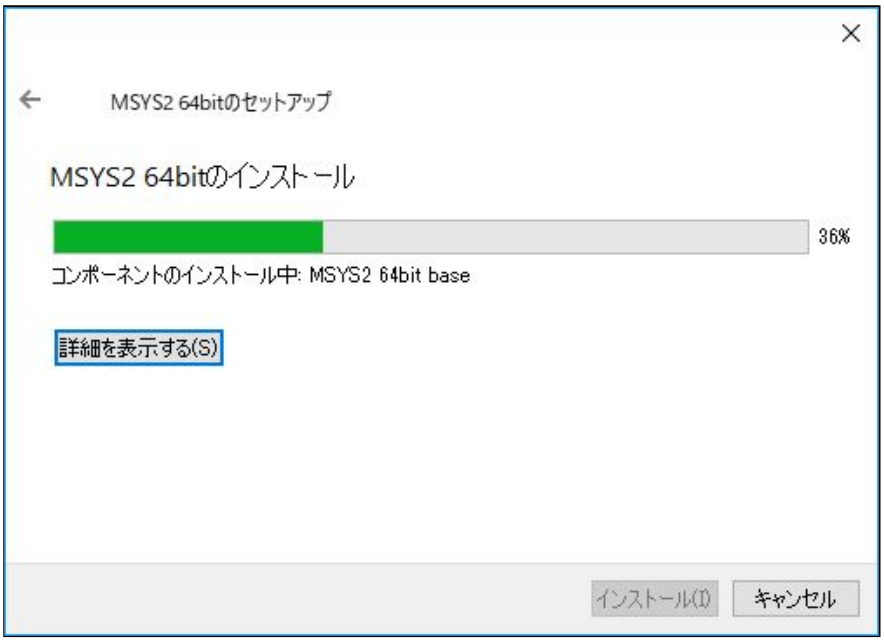
MSYS2/MinGWのインストール画面

キーボードで「1」を入力して Enter キーを押すと、MSYS2のセットアップが始まります。



MSYS2/MinGWのセットアップ開始画面

［次へ］ボタンを2回押してから、［インストール］ボタンを押してください。



MSYS2/MinGWのインストールが進行中

セットアップウィザードが完了したら［完了］ボタンを押してください。MSYS2/MinGWのターミナルが起動して、コマンドプロンプトに似た黒い画面が現れます。ウィンドウ右上の

［X］ボタンを押してMSYS2/MinGWのターミナルを閉じてください。そして、MSYS2/MinGWのインストールのために起動されたコマンドプロンプトも同様に閉じてください。

環境変数の設定

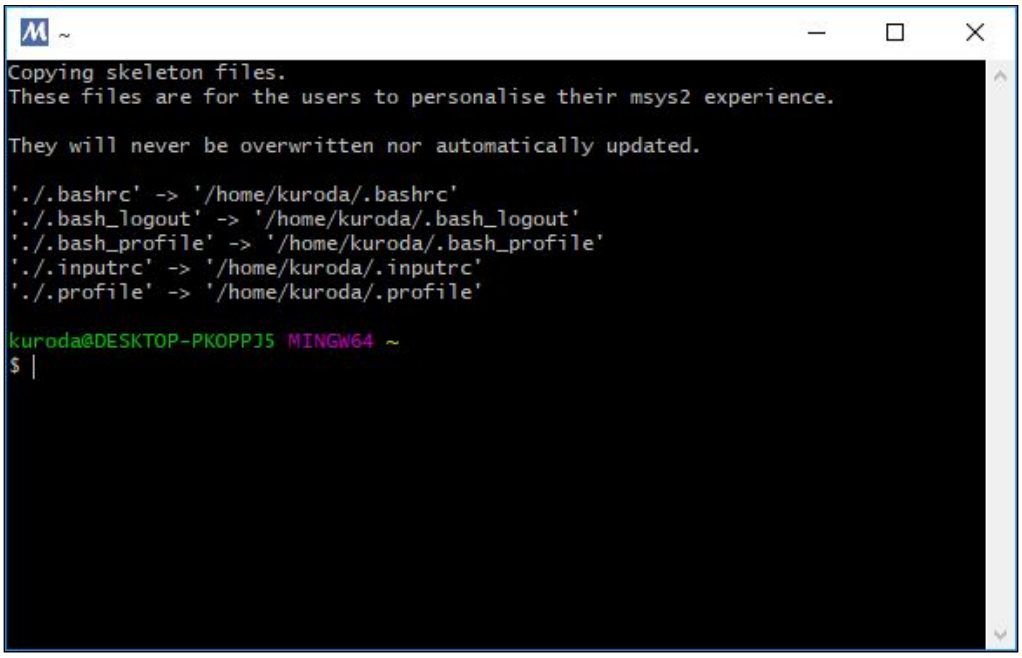
続いて、MSYS2/MinGWを便利に使うため、Windowsの環境変数を設定します。「環境変数」とは、プログラム全般の挙動を制御するための変数名と値（文字列）のペアです。

コントロールパネルを開いて［システム］→［システムの詳細設定］→［環境変数］と進むと環境変数の設定画面が現れます。［ユーザー環境変数］のセクションで環境変数PATHを選んで［編集］ボタンをクリックします。64ビット版のMSYS2/MinGWをお使いの方は、値の先頭にC:¥Ruby25-x64¥msys64を加えてください。32ビット版のMSYS2/MinGWをお使いの方は、値の先頭にC:¥Ruby25¥msys32を加えてください。

さらに、［新規］ボタンをクリックして、環境変数MSYS2_PATH_TYPEにinheritという値を設定してください（64ビット版と32ビット版ともに）。

MSYS2/MinGWターミナルの起動

Windowsの［スタート］ボタンを右クリックして［ファイル名を指定して実行］を選択します。64ビット版のMSYS2/MinGWをお使いの方は名前に「mingw64」と入力して［OK］ボタンをクリックします。32ビット版のMSYS2/MinGWをお使いの方は、名前に「mingw32」と入力して［OK］ボタンをクリックします。



MSYS2/MinGWターミナル

以後、このソフトウェアのことを単に「ターミナル」と呼びます。画面左端に表示されているドル記号（$）の右にコマンドを入力することで、さまざまな操作ができます。

たとえば、Rubyのバージョンを確認するには次のコマンドを入力します。

```
$ ruby -v
ruby 2.5.1p57 (2018-03-29 revision 63029) [x64-mingw32]
```

SQLite3のインストール

データベース管理システムSQLite3をインストールします。64ビット版のRuby 2.5をインストールした方は、ターミナルで次のコマンドを実行してください。

```
$ pacman -S mingw-w64-x86_64-sqlite3
```

32ビット版のRuby 2.5をインストールした方は、ターミナルで次のコマンドを実行してください。

```
$ pacman -S mingw-w64-i686-sqlite3
```

winptyのインストール

Rails開発でirb（Chapter 2）とRailsコンソール（Chapter 4）を頻繁に利用することになりますが、MSYS2/MinGWターミナルではうまく動きません。この問題を解決するためにwinptyというソフトウェアをインストールします。

```
$ pacman -S winpty
```

Railsのインストール

Railsのインストールもターミナル上で行います。次のコマンドを実行してください。

```
$ gem install rails --version "5.2.0" -N
```

なお、-Nオプションを付けるのは、ドキュメントの生成を省略してインストールに要する時間を短縮するためです。

セキュリティの警告

gemコマンドや後述の「bin/rails s」コマンドを実行すると、Windowsやセキュリティ対策ソフトの警告が表示されることがあります。Rubyがインターネットに接続しようとするためです。警告が出たら、［アクセスを許可する］などのボタンをクリックしてください。

インストールが済んだら、Railsのバージョンを確認しましょう。

```
$ rails -v
Rails 5.2.0
```

railsディレクトリの作成

まず、Windowsのデスクトップにrailsというフォルダを作成してください。そして、MSYS2/MinGWのターミナルで以下の作業を行います。

まず、次のコマンドを実行し、Windowsの「コマンドプロンプト」のメッセージが英語で表示されるようにします。

```
$ cmd.exe /c "chcp 65001"
Acitve code page: 65001
```

続いて、次のコマンドを実行します。ただし、oiaxの部分は、自分のユーザー名で置き換えてください。

```
$ cmd.exe /c "mklink /j rails C:¥Users¥oiax¥Desktop¥rails"
Junction created for rails <<===>> C:¥Users¥oiax¥Desktop¥rails
```

このコマンドはデスクトップのrailsフォルダへの「ジャンクション」をMSYS2/MinGW側のホームディレクトリに作成します。これによりWindowsの世界とMSYS2/MinGWの世界が連結されます。cd railsコマンドを実行して、railsディレクトリに移動できればOKです。しかし、次のようなエラーメッセージが表示された場合は失敗です。

```
-bash: cd rails: No such file or directory
```

このときは、次のコマンドでジャンクションを削除してから、ジャンクションを作成するコマンドを注意深く入力し直してください。

```
$ unlink rails
```

以上で、MSYS2/MinGW環境においてRailsの学習を始める準備が整いました。

1.3 テキストエディタの選択

Rubyプログラミングを行うには、テキストエディタと呼ばれるソフトウェアが必要です。テキストエディタには多くの種類があり、基本的にはどれを選んでもかまいません。しかし、初心者が迷わないように、筆者の独断と偏見でいくつかのテキストエディタを推薦します。

 ソースコード編集用のテキストエディタ

ソースコードを編集するためのテキストエディタとして筆者が推薦するのは、Visual Studio Code（VS Code）とAtomの2つです。いずれも無償で入手可能であり、macOS、Windows、およびLinux上で動作します。

■Visual Source Code（VS Code）

Visual Source Code（VS Code）はMicrosoft社により開発されたテキストエディタです。次のURLからダウンロードできます。

https://code.visualstudio.com/

macOSでは、ダウンロードしたZIPファイルを展開し、中に含まれている「Visual Source Code」という名前のファイルを「アプリケーション」フォルダに移動してください。

Windowsでは、ダウンロードしたインストーラーを起動し、説明に従ってインストールしてください。

VS Codeをターミナルから起動するには

macOS、WSL/Ubuntu、MSYS2/MinGWのターミナルからVS Codeを起動するには、開きたいディレクトリに移動してから「code .」というコマンドを実行してください。ただし、事前に作業が必要です。

macOSでは、以下の手順で準備してください。

- 「アプリケーション」フォルダからVS Codeを起動する。
- キーボード・ショートカットcommand-shift-Pを押す。
- 「shell install」と入力してEnterキーを押す。

WSL/Ubuntuでは、ターミナルで次のコマンドを実行してください。

```
$ sudo apt-get realpath
```

MSYS2/MinGWでは何もする必要はありません。

Atom

AtomはGitHub社により開発されたテキストエディタです。次のURLからダウンロードできます。

https://atom.io/

macOSでは、ダウンロードしたZIPファイルを展開し、中に含まれている「Atom」という名前のファイルを「アプリケーション」フォルダに移動してください。

Windowsでは、ダウンロードしたインストーラー（AtomSetup.exe）を起動し、説明に従ってインストールしてください。

Atomをターミナルから起動するには

macOS、WSL/Ubuntu、MSYS2/MinGWのターミナルからAtomを起動するには、開きたいディレクトリに移動してから「atom .」というコマンドを実行してください。ただし、macOSでは事前に次の作業が必要です。

- 「アプリケーション」フォルダからAtomを起動する。

- メニューバーの［Atom］メニューから［Install Shell Commands］を選択する。

WSL/UbuntuおよびMSYS2/MinGWでは何もする必要はありません。

「暗号化された資格情報」設定用のテキストエディタ

テキストエディタはソースコードを編集するためだけに使われるわけではありません。Chapter 12と13で「暗号化された資格情報」を設定する際にも必要となります。この目的で利用するには、ターミナル内で動くテキストエディタ（スクリーンエディタ）が適しています。候補はnanoとVimの2つです。

nano

次に紹介するVimと比較すると、テキストエディタnanoは機能的にかなり貧弱です。しかし、「暗号化された資格情報」を設定するのには十分に使えます。操作方法も直感的なので、初心者にはこちらをお勧めします。

とりあえず次の3点だけ覚えておけば、最低限の編集作業を行えます。

- 文字列のコピーと貼り付けは、マウス操作で行うのが簡単。
- 編集内容を保存して終了するには、Ctrl-Xキーを押してから、キーボードで「y」を入力する。
- 編集内容を保存せずに終了するには、Ctrl-Xキーを押してから、キーボードで「n」を入力する。

さて、「暗号化された資格情報」の設定にnanoを利用するためには、プラットフォーム別に準備作業が必要となります。

macOSではターミナルで次のコマンドを実行してください。

```
$ echo 'export EDITOR=nano' >> ~/.bash_profile
```

WSL/Ubuntuではターミナルで次のコマンドを実行してください。

```
$ echo 'export EDITOR=nano' >> ~/.bashrc
```

MSYS2/MinGWではターミナルで次の2つのコマンドを順に実行してください。

```
$ pacman -S nano
$ echo 'export EDITOR=nano' >> ~/.bashrc
```

Vim

Vimは高機能なスクリーンエディタです。ソースコード編集用のテキストエディタとしてもよく使われます。しかし、操作方法に癖がありますので、初心者にはややハードルが高いです。ここでは、操作方法の解説はしません。すでにVimを使っている方のために、「暗号化された資格情報」設定で使うための準備作業手順だけを記載します。

macOSではターミナルで次のコマンドを実行してください。

```
$ echo 'export EDITOR=vim' >> ~/.bash_profile
```

WSL/Ubuntuではターミナルで次のコマンドを実行してください。

```
$ echo 'export EDITOR=vim' >> ~/.bashrc
```

MSYS2/MinGWではターミナルで次の2つのコマンドを順に実行してください。

```
$ pacman -S vim
$ echo 'export EDITOR=vim' >> ~/.bashrc
```

1.4 アプリケーションの新規作成

Rails開発環境の構築が済んだら、アプリケーションの骨格を作成してみましょう。

rails newコマンド

Rails開発環境の構築が済んだら、続いてアプリケーションを1つ作成し、必要なGemパッケージをBundler（後述）で追加します。

まず、ターミナルで次のコマンドを実行し、ホームディレクトリ直下のrailsディレクトリに移動します。チルダ記号（~）はホームディレクトリを示します。

```
$ cd ~/rails
```

現在のディレクトリを確認するには

ターミナルで現在のディレクトリ（カレントディレクトリ）を確認するには、pwdコマンドを使います。いま、自分のホームディレクトリの下のrailsディレクトリにいる場合、macOS環境であれば次のような結果が表示されます。ただし、oiaxの部分はログインしているユーザー名で置き換わります。

```
$ pwd
/Users/oiax/rails
```

WSL/Ubuntu環境あるいはMSYS2/MinGW環境であれば、次のような結果が表示されます。

```
$ pwd
/home/oiax/rails
```

Railsでは、アプリケーションの作成、モデルやコントローラの作成、サーバーの起動などにrailsコマンドを使います。「rails new アプリケーション名」でアプリケーションの骨格を作成します。本書では、野球クラブMorning Gloryのサイトを作成し、そのアプリケーション名を「asagao」とします。

それでは、Railsアプリケーションを作成しましょう。

```
$ rails new asagao -BCMT --skip-coffee -d sqlite3
      create
      create  README.md
      create  Rakefile
      create  .ruby-version
      create  config.ru
      create  .gitignore
      create  Gemfile
 （中略）
      remove  app/assets/javascripts/cable.js
      remove  app/channels
      remove  config/initializers/cors.rb
      remove  config/initializers/new_framework_defaults_5_2.rb
```

「rails new」コマンドはasagaoディレクトリを作成し、Railsアプリケーションに必要なディレクトリやファイル一式を自動的にコピーします。コマンドの末尾に付けたオプション-BCMT --skip-coffee -d sqlite3については後述します。

続いて、asagaoディレクトリに移動します。

```
$ cd asagao
```

bundle lockコマンドの実行（macOSおよびWSL/Ubuntu）

macOSまたはWSL/Ubuntuを利用している方はターミナルで次のコマンドを実行します。MSYS2/MinGWを利用している方は次の項「Gemfileの編集（MSYS2/MinGWのみ）」に進んでください。

```
$ bundle lock --add-platform x64-mingw32 x86-mingw32
Fetching gem metadata from https://rubygems.org/.....
Fetching gem metadata from https://rubygems.org/.
Resolving dependencies.....
Writing lockfile to /Users/taro/rails/asagao/Gemfile.lock
```

bundle lockコマンドは、後述するGemfile.lockファイルを作成または更新するコマンドです。オプション--add-platformを指定することにより、MSYS2/MinGW用のGemパッケージがGemfile.lockファイルに含まれるようになります。

bundle lockコマンドの実行を省略すると、bundle installコマンドの実行時に次のような警告メッセージが出力されます。

```
The dependency tzinfo-data (>= 0) will be unused by any of the platforms Bundler is installing for. Bundler is installing for ruby but the dependency is only for x86-mingw32, x86-mswin32, x64-mingw32, java. To add those platforms to the bundle, run `bundle lock --add-platform x86-mingw32 x86-mswin32 x64-mingw32 java`.
```

この警告メッセージが出ないようにする方法はもうひとつあります。テキストエディタでGemfileの末尾にある次のような記述を削除することです。

```
# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

RailsアプリケーションをMSYS2/MinGWの環境で動かす計画がないのであれば、この部分は不要です。

Gemfileの編集（MSYS2/MinGWのみ）

MSYS2/MinGWを利用している場合は、テキストエディタでGemfileの末尾に次の行を追加してください。

```
gem 'wdm', platforms: [:mingw, :x64_mingw]
```

macOSやWSL/Ubuntuを利用しているのならこの記述は不要ですが、存在してもかまいません。本書のサポートサイトで配布されているサンプルコードでは上記の変更が施されています。

bundle installコマンドの実行（全プラットフォーム共通）

最後に、ターミナルで次のコマンドを実行し、Railsアプリケーションの開発に必要なGemパッケージ群をインストールします。

```
$ bundle install
Fetching gem metadata from https://rubygems.org/.....
Fetching gem metadata from https://rubygems.org/.
Resolving dependencies...
Fetching rake 12.3.1
Installing rake 12.3.1
```

```
Fetching concurrent-ruby 1.0.5
Installing concurrent-ruby 1.0.5
Fetching i18n 1.0.1
Installing i18n 1.0.1
（中略）
Bundle complete! 15 Gemfile dependencies, 65 gems now installed.
Use `bundle info [gemname]` to see where a bundled gem is installed.
```

下から2行目の「Bundle complete!」というメッセージが、インストールの成功を示しています。なお、bundle installの代わりにbundleとだけ入力しても同じ結果が得られます。

以上で、Railsアプリケーションを動かす準備が整いました。

rails newコマンドのオプション

先ほどrails newコマンドでアプリケーションの骨格を作成したとき、本書では末尾に-BCMT --skip-coffee -d sqlite3というオプションを追加しました。各オプションの意味は次のとおりです。

rails newコマンドのオプション

オプション	意味
-B, --skip-bundle	bundleコマンドを実行しない
-C, --skip-action-cable	Action Cableのファイル群を生成しない
-M, --skip-action-mailer	Action Mailerのファイル群を生成しない
-T, --skip-test	テスト関連のファイル群を生成しない
--skip-coffee	CoffeeScriptを使用しない
-d, --database	データベース管理システムの種類を指定する

bundleコマンドについてはあとで説明しますが、通常はアプリケーションの骨格が作られたあとで自動的にこのコマンドが実行されます。しかし、本書ではbundleコマンドを実行する前にやっておきたいことがあるので、オプション-Bを付けました。

それから、本書では扱わない機能（Action Cable、Action Mailer、テスト）に関連するファイル群を生成しないようにするオプション-CMTを付けました。これらの機能を使いたい場合は、適宜オプションを減らしてください。また、本書ではCoffeeScript（コーヒースクリプト）を使わないのでオプション--skip-coffeeを加えました。

オプション-dはデータベース管理システムの種類を指定するためのものです。本書ではSQLite3を用いるのでsqlite3を指定しています。ただし、オプション-dのデフォルト値はsqlite3ですので、SQLite3を利用するのならこのオプションは省略できます。

rails newコマンドに指定できるすべてのオプションを知りたければ、オプション-hを付けてrails newコマンドを実行してください。

rails newでのバージョン指定

Railsの複数のバージョンをインストールしているときは、「rails new」コマンドは一番新しいRailsを使ってアプリケーションを作成します。特定のバージョンのRailsでアプリケーションを作成したいときは、railsのすぐ次に「_バージョン番号_」を加えます。

```
$ rails _5.1.6_ new asagao -BCMT --skip-coffee -d sqlite3
```

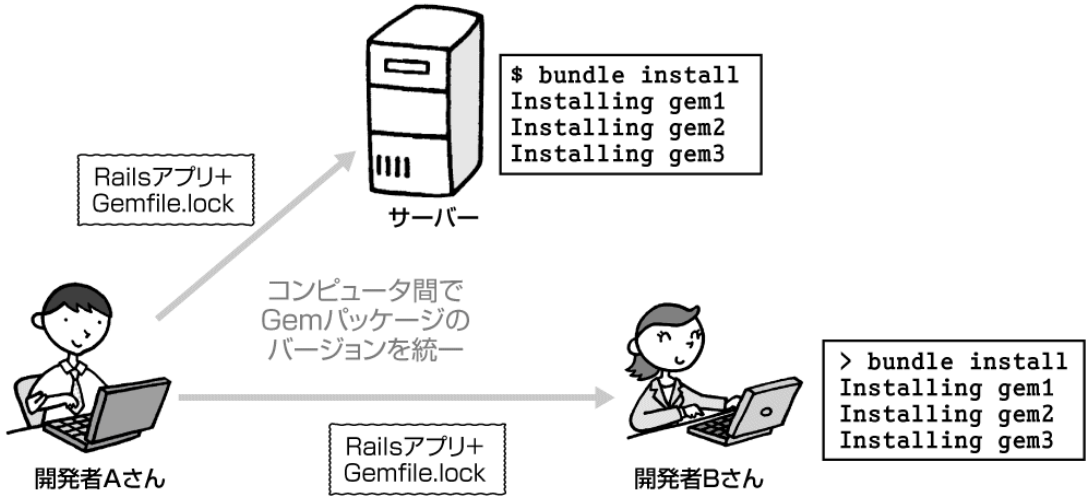
Bundler

ここで、Bundlerについて簡単に解説しておきましょう。**Bundler**は、Gemパッケージの管理とインストールのためのツールです。Railsアプリケーションなど Rubyで書いたアプリケーションのために使われます。

Railsアプリケーションはたくさんの Gemパッケージを必要としますが、複数の開発者が共同で開発を進めるときには、開発者の間で同じバージョンの Gemパッケージを用意しなければなりません。開発用のパソコンと本番サーバーの間でも同様です。

Bundlerは、特定のRailsアプリケーションに必要なGemパッケージをリストアップし、複数のコンピュータの間で簡単に同期が取れるようにするものです。図のように開発者Bさんが開

発者AさんからRailsアプリケーションのソースコードを受け取ったとします。Bさんは「bundle install」を実行するだけで、必要なGemパッケージをインストールして、環境を揃えることができます。



Bundlerの働き

「rails new」でアプリケーションを作成すると、アプリケーションのルートディレクトリに自動的にGemfileができます。このファイルには、Bundlerを使ってインストールするGemパッケージの一覧が記述されています。先ほど作った自分のasagaoアプリケーションのGemfileを開いてみましょう。この中の「gem 'パッケージ名'」が必要なGemパッケージの指定です。

LIST chapter01/Gemfile

```
1 source 'https://rubygems.org'
2 git_source(:github) { |repo| "https://github.com/#{repo}.git" }
3
4 ruby '2.5.1'
5
6 # Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
7 gem 'rails', '~> 5.2.0'
8 # Use sqlite3 as the database for Active Record
9 gem 'sqlite3'
```

```
10 # Use Puma as the app server
11 gem 'puma', '~> 3.11'
12 # Use SCSS for stylesheets
13 gem 'sass-rails', '~> 5.0'
   （以下省略）
```

4行目にruby '2.5.1'という記述があります。これは、ここに指定されたバージョンのRubyでしかこのRailsアプリケーションを起動できないことを意味します。もし別のバージョンのRubyでもこのRailsアプリケーションを動かしたいのであれば、制限を緩める必要があります。Gemfileの4行目を次のように変更すれば、バージョン2.4.4以上のRubyでの実行が許されるようになります。

```
ruby '>= 2.4.4'
```

Gemfileにおけるバージョン指定

Gemfileを編集するときは、次のようにGemパッケージのバージョンを指定できます。

```
gem 'rails', '5.2.0'              # バージョン5.2.0が必要
gem 'sqlite3'                     # どのバージョンでもOK
gem 'sass-rails', '~> 5.0'        # バージョン5.0以上、6未満
gem 'uglifier', '>= 1.3.0'        # バージョン1.3.0以上
gem 'listen', '>= 3.0.5', '< 3.2' # バージョン3.0.5以上、3.2未満
```

「bundle install」を実行すると、Bundlerは各パッケージが依存しているほかのパッケージをすべて調べてインストールします。同時に、Gemfileと同じディレクトリにGemfile.lockというファイルを作成し、そこに必要なパッケージのバージョン番号を記録します。Gemfile.lockは自動的に生成されるものなので、内容を編集する必要はありません。

asagaoアプリケーションのGemfile.lockを見てみましょう。

LIST chapter01/Gemfile.lock

```
GEM
  remote: https://rubygems.org/
  specs:
    actioncable (5.2.0)
      actionpack (= 5.2.0)
      nio4r (~> 2.0)
      websocket-driver (>= 0.6.1)
    actionmailer (5.2.0)
      actionpack (= 5.2.0)
      actionview (= 5.2.0)
      activejob (= 5.2.0)
      mail (~> 2.5, >= 2.5.4)
      rails-dom-testing (~> 2.0)
  （以下省略）
```

Gemfile.lockファイルと一緒にRailsアプリケーションを受け取った別の開発者が「bundle install」を実行すると、BundlerはGemfile.lockのほうを読み込んで、そこに記述されているバージョンのGemパッケージをインストールします。

bundleコマンドのサブコマンド

bundleコマンドでは、lock、installのほかに次のサブコマンドが使えます。

bundle list
必要なGemパッケージをすべて一覧表示します。

bundle check
必要なGemパッケージがインストールされているか調べます。インストールされていれば、「The Gemfile's dependencies are satisfied」と表示されます。

bundle update

Gemfile.lockを無視して、GemfileをもとにGemパッケージをインストールし直し、Gemfile.lockを作り直します。Gemパッケージを最新版にしたいときに使います。

1.5 Railsを動かしてみよう

前節で作成したRailsアプリケーションを起動してみましょう。また、ちょっとしたプログラミングをしてアプリケーションを改造してみましょう。インストールがうまくいったかどうかを確認するためと、作業をしながらRailsのしくみを何となく実感してみるためです。

アプリケーションの起動

1.3節で作ったasagaoディレクトリに移動してから「bin/rails s」コマンドを実行すると、Pumaというウェブサーバーがパソコン内で起動します。「bin/rails server」と入力しても同じです。

```
$ bin/rails s
=> Booting Puma
=> Rails 5.2.0 application starting in development
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.12.0 (ruby 2.5.1-p57), codename: Llamas in Pajamas
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://0.0.0.0:3000
Use Ctrl-C to stop
```

なお、MSYS2/MinGWでは、「Puma starting in single mode...」のメッセージの前に次のような警告が出力されますが、支障はありませんので無視してください。

```
*** SIGUSR2 not implemented, signal based restart unavailable!
*** SIGUSR1 not implemented, signal based restart unavailable!
*** SIGHUP not implemented, signal based logs reopening unavailable!
```

MSYS2/MinGW環境でLoadErrorが出た場合

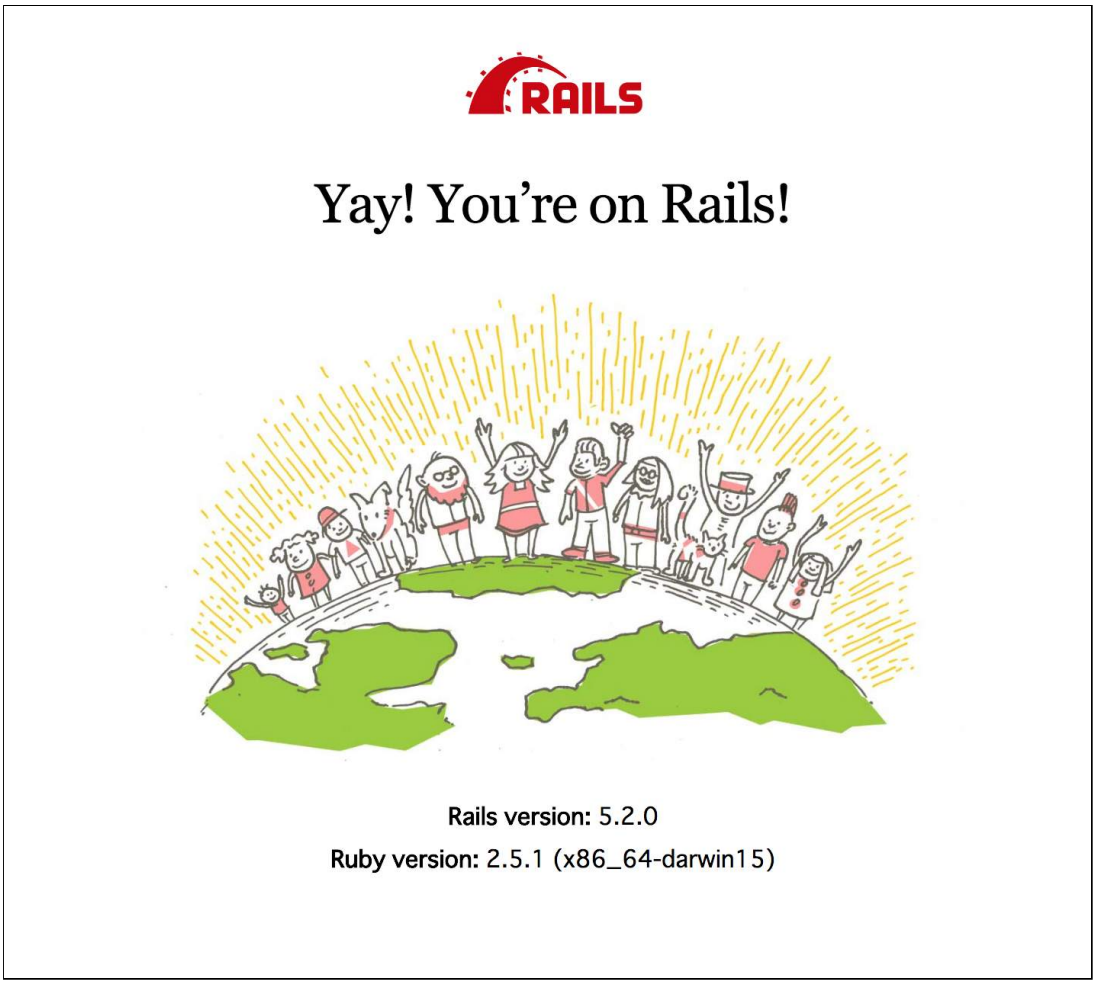
本書の執筆時点では、Gemパッケージsqlite3がWindows版のRuby 2.5に対応していないため、次の節でRailsサーバーを起動すると次のようなエラーが出てしまいます。

```
cannot load such file -- sqlite3/sqlite3_native (LoadError)
```

近い将来（おそらくは本書が刊行されるまでに）この問題は解消されるはずですが、もし上記のエラーに遭遇したら次の2つのコマンドを順に実行してください。

```
$ gem uninstall -a sqlite3
$ gem install sqlite3 --platform ruby -N
```

ウェブブラウザを開いて、URL欄に「http://localhost:3000/」と入力し、このサーバーにアクセスしてください。Railsアプリケーションの初期画面が表示されます。表示されないときは、URLを「http://127.0.0.1:3000/」に変えてみてください。



RESULT Railsアプリケーションの初期画面

この画面が表示されれば、Railsのインストールは成功です。ターミナルでCtrl-Cキーを押して、サーバーを終了してください。

PostgreSQLまたはMySQLを利用する場合

データベースにSQLite3ではなくPostgreSQLまたはMySQLを利用している方は、「bin/rails s」コマンドを実行する前に、データベースを設定してターミナルで「bin/rails db:create」を実行してください（Chapter 4を参照）。Railsアプリケーションがデータベースに接続しようとするからです。SQLite3では自動的にデータベースが作られます。

Railsアプリケーションのディレクトリ構造

いったんサーバーを終了したら、作成したディレクトリの中身を見てみましょう。macOSユーザーの方はFinderで、Windowsユーザーの方はエクスプローラーでデスクトップのrailsフォルダを開いてください。そこにasagaoフォルダができているはずです。このフォルダの中のファイルを修正したり追加したりするのが、Railsアプリケーションを開発する方法です。

asagaoフォルダの下には、app、binなどのフォルダ（ディレクトリ）があります。それぞれのディレクトリの役割は次のとおりです。頻繁に使うことになるのは、app、config、dbの各ディレクトリです。次の項ではappディレクトリとconfigディレクトリの中のファイルを修正して簡単なテストページを作ります。

ディレクトリの役割

ディレクトリ	役割
app	モデル、ビュー、コントローラのコード。各Chapterで利用。
bin	各種スクリプトファイル。HINTを参照。
config	ルーティングやデータベースなどの設定ファイル。各Chapterで利用。
db	マイグレーションスクリプトやシードデータ。Chapter 4で解説。
doc	開発者向けのドキュメントを置く。本書では利用しない。
lib	自作のライブラリやrakeファイル。
log	ログが出力される。
public	アプリケーションを介さずに送信する静的なファイルを配置。
storage	Active Storageが利用する。Chapter 13で解説。
test	テストスクリプト。本書では利用しない。
tmp	キャッシュなどのテンポラリディレクトリ。
vendor	プラグインを配置。本書では利用しない。

binディレクトリのスクリプト

bin/を付けずにrailsコマンドを使うと、RubyGemsでインストールしたrailsコマンドが動きます。bin/railsとすると、binディレクトリの下のスクリプトが使われます。Railsアプリケーション（本書ではasagao）は特定のバージョンのRails（本書では5.2.0）で動かす必要があります。パソコンにいくつ

ものバージョンのRailsをインストールしている場合は、bin/railsによって確実に特定のバージョンのRailsを動かすことができます。

コントローラとアクションの作成

Railsの初期画面ではなく、自分が作ったページをサイトのトップページにしてみましょう。そのためにはコントローラとビューを作成する必要があります。

■コントローラを作成する前に

コントローラを作成する前に、次の内容のファイルをconfig/initializersディレクトリの下に作成してください。ファイル名はgenerators.rbとします。

LIST chapter01/config/initializers/generators.rb

```
1 Rails.application.config.generators do |g|
2   g.helper false        # ヘルパーを生成しない
3   g.assets false        # CSS, JavaScript ファイルを生成しない
4   g.skip_routes true    # config/routes.rb を変更しない
5   g.test_framework false # テストスクリプトを生成しない
6 end
```

これは、後述の「bin/rails g」コマンドが生成するファイルを減らして、Railsの学習を進めやすくするためのものです。

■コントローラの作成

それではコントローラを作成しましょう。ターミナルに戻り、「bin/rails g」コマンドを実行します。「rails g controller コントローラ名 アクション名」でコントローラとアクションを作成できま

す。ここではTopControllerとindexアクションを作成することにします。「bin/rails g」は「bin/rails generate」としても同じです。

```
$ bin/rails g controller top index
      create  app/controllers/top_controller.rb
      invoke  erb
      create    app/views/top
      create    app/views/top/index.html.erb
```

ディレクトリapp/controllersを開いてみましょう。中にtop_controller.rbというRubyファイルができています。このファイルをテキストエディタで開くと、次のように書かれています。

```
class TopController < ApplicationController
  def index
  end
end
```

これは、TopControllerというクラスを記述したものです。TopControllerクラスはApplicationControllerクラスを継承しています。このように、Railsではコントローラを1つのRubyのクラスで表します。

TopControllerクラスの中には、indexメソッドができています。このindexメソッドがトップページを表示するときに呼ばれる「アクション」になります。

RubyのクラスについてはChapter 2を、コントローラとアクションの詳細についてはChapter 3を参照してください。

■ルーティングの設定

コントローラとアクションを作っただけでは、Railsアプリケーションのページにはなりません。URLのパスとコントローラを結び付けるルーティングを設定する必要があります。configディレク

トリの下のroutes.rbを開き、「Rails.application.routes.draw do」と「end」の間の記述をすべて削除して、次のように書き直してください。

LIST chapter01/config/routes.rb

```
1 Rails.application.routes.draw do
2   root "top#index"
3 end
```

これにより、トップページ（「https://www.oiax.jp/」のように/で表されるページ）に対応するコントローラとアクションがTopControllerのindexアクションになります。ルーティングに関しては、Chapter 3とChapter 5を参照してください。

「bin/rails s」コマンドでもう一度サーバーを起動し、ブラウザで「http://localhost:3000/」を開いてください。TopControllerのindexアクションのページが表示されます。

RESULT TopControllerのindexアクション

ビューの作成

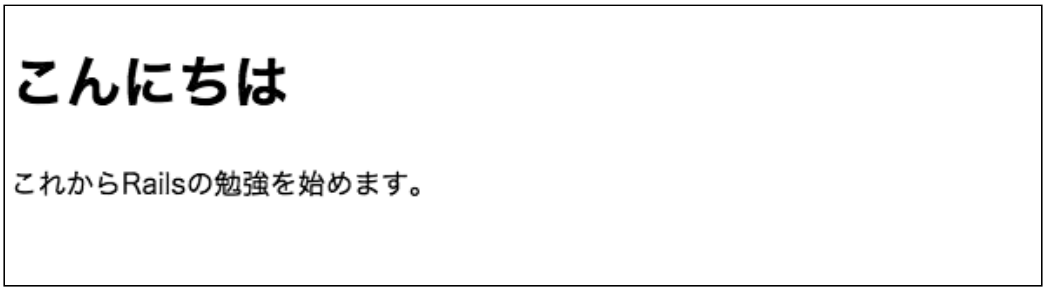
■ビューの作成

indexアクションに対応するビューのテンプレートを編集してみましょう。ディレクトリapp/views/topを開き、ファイルindex.html.erbを開いてください。拡張子が.erbのファイルは、Railsのビューのためのテンプレートファイルです。ファイル名は、「アクション名（ここではindex）.html.erb」となります。

このファイルを次のように編集して保存してみましょう。

```
<h1>こんにちは</h1>
<p>これからRailsの勉強を始めます。</p>
```

ブラウザで「http://localhost:3000/」を再読み込みすると、テンプレートファイルに記述した内容が表示されます。



RESULT ビューのテンプレートの内容が表示される

文字コードはUTF-8で保存する

Railsでは、文字コードはUTF-8を使います。Railsで使用するRubyのソースファイル（.rb）やテンプレートファイル（.erb）をテキストエディタで保存するときは、必ず文字コードをUTF-8にしてください。

WindowsでTeraPadを使っている方は、「UTF-8」ではなく「UTF-8N」を選択ください。秀丸エディタでは、［エンコードの種類］から「Unicode（UTF-8）」を選択し、［BOMを付ける］オプションのチェックを外してください。

アクションはあとからでも作れる

「bin/rails g」でコントローラを作成するときは、「bin/rails g controller top」のようにアクション名を指定せずにコントローラ名だけを指定できます。TopControllerクラスにindexメソッドを自分で記述すれば、それがアクションになり、ビューのディレクトリにindex.html.erbファイルを作成すれば、それがテンプレートとして使われます。

■変数の表示

これだけではつまらないので、もう少しRailsの機能を使ってみましょう。app/controllersの下のtop_controller.rbを開き、indexアクションに@messageという変数を記述します。

@messageのあとの"と"の間には、好きな言葉を入れてください。文字コードはUTF-8にして保存します。

LIST chapter01/app/controllers/top_controller.rb

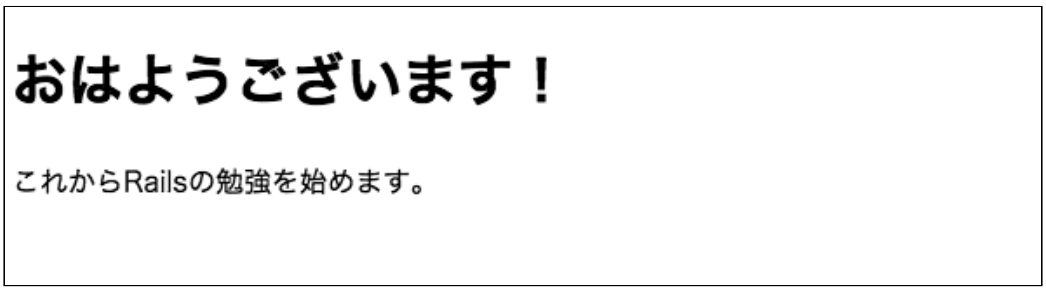
```
1 class TopController < ApplicationController
2   def index
3     @message = "おはようございます！"
4   end
5 end
```

続いてapp/views/topの下のテンプレートindex.html.erbに<%= %>という部分を作り、間に変数@messageを置きます。

LIST chapter01/app/views/top/index.html.erb

```
1 <h1><%= @message %></h1>
2 <p>これからRailsの勉強を始めます。</p>
```

ブラウザで「http://localhost:3000/」を再読み込みすると、コントローラのアクションに記述した変数がビューに反映され、ブラウザ上に変数@messageの値が表示されます。



RESULT コントローラの変数がビューに反映される

再読み込みだけで開発が進む

Railsアプリケーションの開発中は、サーバーを起動したままにしておきます。ソースコードを修正したら、ブラウザで再読み込みするだけで修正点が確認できます。サーバーを再起動する必要はありません。「bin/rails g」などのコマンドを実行したいときは、ターミナルをもう1つ開いて作業をするとよいでしょう。

Railsは頻繁にバージョンアップされる

Ruby on Railsは数週間から数か月おきに細かいバージョンアップが行われ、数年おきに大きなバージョンアップが行われます。もし本書の刊行後にRails 6.0がリリースされたとしても、そのバージョンでは本書のサンプルプログラムがそのまま動くとは限りません。本書を読み進める際には、Railsバージョン5.2を使用してください。また、オイアクス社のサポートサイトを必ず参照してください。

Chapter 1のまとめ

- Ruby on Railsは、プログラミング言語Rubyによるウェブアプリケーション開発のためのフレームワークです。
- Ruby on Railsをインストールするには、Rubyをインストールしてから、RubyGemsでRails関連のGemパッケージをインストールします。
- ターミナルでrails newコマンドを実行すると、アプリケーションの骨格を作成できます。
- アプリケーションに必要なGemパッケージを管理・インストールするにはBundlerを使います。
- Railsアプリケーションは、ウェブサーバーPumaを動かしながら開発します。

- **コントローラにアクションを追加し、アクションに対応するテンプレートを記述すると、ウェブページができます。**

練習問題

［A］　次の文の内容が正しい場合は○、間違っている場合は×を記入してください。

- （　）Rubyは21世紀の初めに登場した比較的新しいプログラミング言語です。
- （　）Ruby on Railsは、オープンソース方式で開発されているフレームワークです。
- （　）Ruby on Railsは、macOS、Windows、Linuxなどさまざまなで動作します。
- （　）Railsをインストールするには、パッケージ・マネージャのRubyGemsを使います。
- （　）新しいGemパッケージを導入するときは、テキストエディタでGemfile.lockを編集します。

［B］　次の空欄を埋めてください。

- Railsの原則DRYは、「Don't Repeat Yourself」の略で［　　］という意味です。
- Railsは［　　］という設計哲学で作られており、規約に従ってアプリケーションを開発することで、記述量を大幅に減らすことができます。

［C］　次の空欄を埋めて、変数@descriptionの値を表示させてください。

コントローラ

```
def index
  @message = "こんにちは"
```

```
  @description = "これからRailsアプリケーションを作ります。"
end
```

テンプレート

```
<h1><%= @message %></h1>
<p>                                    </p>
```

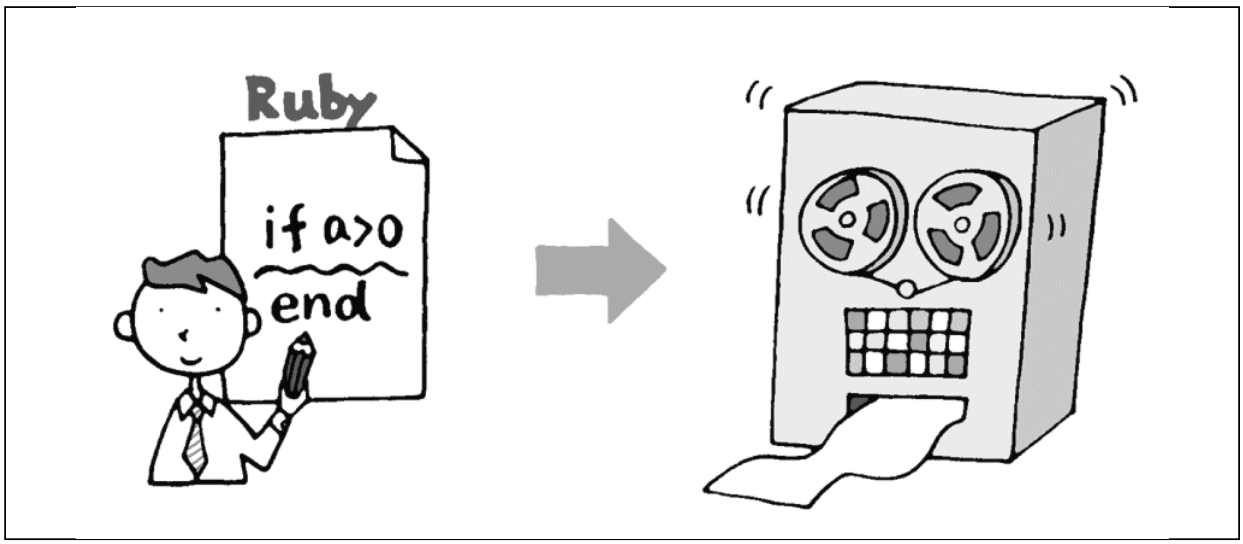
Chapter

2　Ruby言語の基礎を学ぼう

Rubyを使ったことがない人は、Ruby on Railsに取りかかる前に、言語の学習をしておきましょう。Rubyは学びやすく書きやすいオブジェクト指向言語です。本書では、Ruby 2.5を使って文法や特徴を説明します。

これから学ぶこと

- 変数や演算子、条件分岐といったRuby言語でのプログラミングの基本を学びます。
- メソッドの書き方とブロックを扱うメソッドの使い方を学びます。
- 数値や文字列、配列、ハッシュ、日時、日付といったよく使うオブジェクトについて学びます。
- Rubyでのクラスの書き方を学びます。



Rubyはプログラミング言語の一種です。Rubyにはプログラミングを楽しくするためのおもしろい特徴がたくさんあります。Rubyを使ってどんなプログラムが書けるでしょうか？

2.1 変数と式

まずは、変数や式、演算子といったプログラムの基本的な書き方から見てみましょう。Rubyの文法は簡単に覚えられます。数値や文字列の使い方を覚えながら文法の基礎を学び、Rubyのおもしろさを体感してみましょう。

■準備作業（MSYS2/MinGW環境のみ）

MSYS2/MinGW環境で学習している方は、ターミナルで次のコマンドを実行してから本文を読み進めてください。

```
$ alias ruby='winpty ruby'
```

このコマンドを実行しないと、対話式の（途中でキーボードからの入力が必要となる）Rubyプログラムが正常に動きません。ターミナルを開くたびにこのコマンドを実行する必要があります。

Rubyの基本的な使い方

簡単なRubyプログラムを書いてみましょう。テキストエディタで次のソースコードを記述して、2-1-1.rbというファイル名を付けて保存してください。ファイルを保存するときは、必ず文字コードをUTF-8にしてください。

LIST chapter02/2-1-1.rb

```
1 puts "こんにちは"
```

Rubyプログラムを実行するには、ターミナルを使います。ターミナルを開いて、ファイルを保存したディレクトリ（ここでは自分のホームディレクトリの下のrails/chapter2とします）に移動してください。「ruby ファイル名」を入力すると、Rubyが起動してプログラムを実行します。

```
$ cd rails/chapter2
$ ruby 2-1-1.rb
```

RESULT

```
こんにちは
```

「puts "こんにちは"」は文字列を表示するコードです。putsの部分を**メソッド**と言い、"こんにちは"の部分を**引数**（ひきすう）と言います。

メソッドは「機能の呼び出し」、引数は「メソッドに指定する数値や文字列」と覚えてください。この例では省略していますが、「puts("こんにちは")」のように引数を( )で囲むこともできます。

putsメソッドは、引数をターミナルに改行を付けて表示するメソッドです。改行なしで表示したいときは、putsの代わりにprintメソッドを使います。

コンパイラとインタプリタ

Rubyは、インタプリタ型のプログラミング言語です。つまり、コンパイルして実行ファイルを作成するのではなく、Rubyプログラムがテキストファイルに記述されたソースコードを読み込みながら、コードを解釈して実行します。

ただし、Ruby 1.9以降の言語実装（YARV）では、処理を高速化するために、テキストのコードをいったん独自のバイナリーコードに変換してから実行します。現在のRubyはコンパイラとインタプリタの中間型とも言えます。

■変数

変数は、数値や文字列などさまざまな**オブジェクト**を指し示すのに使います。オブジェクトとは、とりあえずは「操作の対象になるもので、数値や文字列のようないろんな種類があるもの」と考えてください。

等号（=）で変数とオブジェクトを結べば、その場で変数ができます。次の例では、「こんにちは」という文字列を示す変数messageと、123という数値を示す変数numを作っています。

LIST chapter02/2-1-2a.rb

```
1 message = "こんにちは"
2 puts message
3 num = 123
4 puts num
```

RESULT

```
こんにちは
123
```

代入演算子

=を代入演算子と言います。また、上の例の「message = "こんにちは"」のことを「変数messageに"こんにちは"を代入する」と言います。Chapter 2では、「代入する」「入れる」「指し示す」は同じことだと考えてください。代入の意味について詳しくは、「2.4　クラス」の「参照とコピー」を参照してください。

変数名には、半角のアルファベットと数字、アンダースコア（_）を使い、変数名の1文字目は半角アルファベットまたはアンダースコアにしてください。文法上は漢字やひらがなも使えますが、普通は使いません。

Rubyの変数には種類があります。messageのような変数を**ローカル変数**と言います。変数名をアルファベットの大文字で始めると**定数**になります。また、変数名を$や@などの記

号で始めると、変数の種類が変わります。本書では、ローカル変数のほかにインスタンス変数と定数も使います（グローバル変数とクラス変数は扱いません）。

Rubyの変数の種類

変数名	種類
message	ローカル変数
$message	グローバル変数
@message	インスタンス変数（「2.4　クラス」を参照）
@@message	クラス変数
MESSAGE	定数（「2.4　クラス」を参照）

数値や文字列が持っている機能を使うには、「変数.メソッド」という形でメソッドを呼び出します。次の例は、文字列の文字数を調べるものです。

LIST chapter02/2-1-2b.rb

```
1 message = "こんにちは"
2 puts message.length
```

RESULT

```
5
```

改行が文の終わりになる

Rubyでは、文の終わりを示すためのセミコロン（;）は必要ありません。1行に複数の文を書きたいときには、次のようにセミコロンを使います。

```
a = 2; b = 3
```

Chapter 2では、紙面の都合上セミコロンを使っている例がありますが、自分で入力するときは、セミコロンの代わりに改行を入れるほうがRubyらしいプログラムになります。

Rubyのコメント

Rubyのソースコードでは、#から改行までがコメントと解釈されます。コメントはプログラムの動作に影響を与えません。

```
# ごあいさつ
puts "こんにちは"
puts "さようなら" # ここにもコメントを書けます
```

数値と文字列

プログラミングの基本となる数値と文字列の扱い方を学びましょう。

■数値

足し算や掛け算のような計算をするには**数値**を使います。Rubyには、オブジェクトの種類を表す**クラス**があります。数値を表すクラスには、IntegerとFloatの2つがあります。

FixnumクラスとBignumクラス

かつてRubyには整数を表すFixnumクラスと非常に大きな整数を表すBignumクラスが存在しましたが、Ruby 2.4で両者はIntegerクラスに統合されました。もしこれらのクラス名をプログラムの中で使用すると、「constant ::Fixnum is deprecated」のような警告メッセージが出力されます。

変数に整数を代入するとIntegerクラス、小数点付きの数（浮動小数点数）を代入するとFloatクラスのオブジェクトができます。クラスについて詳しくは「2.4　クラス」を参照してください。

```
num = 1234        # Integer
pi = 3.14159      # Float
```

整数値と小数点付きの値を計算したときは、自動的にFloatに変換されます。このため、数値の種類はあまり気にする必要はありません。

ただし、整数同士の割り算の結果は常に整数になります。次の例では、5を2で割った結果は2に、5を2.0で割った結果は2.5となります。

LIST chapter02/2-1-3a.rb

```
1 a = 5
2 b = 2
3 c = 2.0
4 puts a / b # 整数同士
5 puts a / c # 整数と浮動小数点数
```

RESULT

```
2
2.5
```

1234や"こんにちは"のようにソースコードに直接書かれた数値や文字列のことを**リテラル**と呼びます。リテラルもオブジェクトなので、123.to_sや"こんにちは".lengthのようにメソッドを呼び出せます。

変数に型はない

Rubyには「変数の型」はありません。ある変数に数値を入れたあとで、同じ変数に文字列を入れ直すこともできます。変数に型はありませんが、変数が指し示すオブジェクトには種類があります。

```
a = 123        # aはIntegerオブジェクトを指す
a = "hello"    # aはStringオブジェクトを指す
```

■文字列

文字の並びを一重引用符または二重引用符で囲むと、**文字列**ができます。Rubyの文字列は、Stringクラスのオブジェクトです。一重引用符と二重引用符はどちらを使ってもかまいませんが、本書ではおもに二重引用符を使います。

```
hello = 'こんにちは'
goodbye = "さようなら"
```

文字列は演算子+で連結できます。次の例は、2つの変数xとyにセットされた文字列同士を連結して表示します。

LIST chapter02/2-1-3b.rb

```
1 x = "ABC"
2 y = "DEF"
3 puts x + y
```

RESULT

```
ABCDEF
```

二重引用符で囲まれた文字列の中に#{○○}という形の部分があると、「○○」を文字列に変換した結果が埋め込まれます。これを**式展開**と呼びます。一重引用符内では式展開は行われません。

LIST chapter02/2-1-3c.rb

```
1 puts "2かける3は#{2 * 3}です。"
2 name = "佐藤"
3 puts "#{name}さん、こんにちは。"
```

RESULT

```
2かける3は6です。
佐藤さん、こんにちは。
```

バックスラッシュ記法

二重引用符の中に二重引用符を書きたい場合など、文字列に特別な文字を埋め込むときは、"¥""のように「バックスラッシュ＋文字」の形で記述します。Macの日本語キーボードで¥を入力するには、option キーを押しながら ¥ キーを押します。Windowsの日本語キーボードでは円記号（¥）のキーを押してください。なお、一重引用符の中で使えるバックスラッシュ記法は¥'と¥¥だけです。

よく使われるバックスラッシュ記法

記法	特別な文字
¥n	改行
¥r	復帰
¥t	タブ
¥"	二重引用符
¥'	一重引用符
¥¥	バックスラッシュ
¥xNN	番号で文字を表す。NNは16進数
¥uNNNN	ユニコード番号で文字を表す。NNNNは16進数

文字列を囲む記号

%q、%Q、%を使うと、一重引用符と二重引用符以外の好きな記号で文字列を囲むことができます。HTMLのタグなど、文字列に引用符が含まれるときに使うと便利です。%Qと%は二重引用符と同じく式やバックスラッシュ記法を埋め込むことができ、%qは一重引用符と同じく埋め込めません。

```
msg = %q/こんにちは/                    # / /で囲う
msg = %Q(こんにちは)                    # ( )で囲う
url = %!<a href="http://www.google.com">Google</a>!  # ! !で囲う
```

■数値と文字列の変換

Rubyは、数値と文字列を自動的に相互変換しないことに注意してください。数値と文字列を+で連結したり、>で比較したりするとエラーになります。

```
a = "4"; b = 9
c = a + b # エラー
```

数値と文字列を互いに変換するには、メソッドを使う必要があります。次の例は、数値と文字列を連結、足し算、掛け算する例です。数値を文字列に変換するにはto_sメソッドを使います。文字列を整数に変換するにはto_iメソッドを、浮動小数点数に変換するにはto_fメソッドを使います。

LIST chapter02/2-1-3d.rb

```
1 a = 4; b = "9"
2 puts a.to_s + b
3 puts a + b.to_i
4 puts a * b.to_i
```

RESULT

```
49
13
36
```

ある変数が文字列以外のオブジェクトであるときや、何のオブジェクトを指しているかわからないときは、文字列の連結の代わりに式展開を使えばエラーになりません。

```
x = 123
s = "xの値は#{x}です"
```

■配列

配列とは、いくつものオブジェクトを並べてまとめるためのオブジェクトです。Rubyの配列はArrayクラスのオブジェクトです。配列中の個々のオブジェクトを配列の**要素**と呼びます。

配列を作るには、[と]の間にカンマで要素を区切って並べます。配列の要素を取り出したり、要素の値を変えたりするには、「配列名[番号]」と記述します。要素の番号は0から始まります。0が1番目、1が2番目です。

LIST chapter02/2-1-4a.rb

```
1 animals = ["dog", "cat", "elephant"]
2 puts animals[0]        # 1番目を表示
3 animals[1] = "bat"     # 2番目を変更
4 puts animals[1]        # 2番目を表示
```

RESULT

```
dog
bat
```

現在の要素数より大きな番号を指定して要素を取り出そうとすると、何もないことを表すnilが返されます。また、大きな番号を指定して要素を加えると、配列は自動的に大きくなります。

次の例の1つ目の「puts animals[5]」は、nilなので何も表示されません。

LIST chapter02/2-1-4b.rb

```
1 animals = ["dog", "cat", "elephant"]
2 puts animals[5]        # 6番目はnil
```

```
3 animals[5] = "whale"    # 6番目を作成
4 puts animals[5]         # 6番目を表示
```

RESULT

```
（ここにnilが出力されている）
whale
```

配列には、要素として雑多なオブジェクトを入れることができます。たとえば、文字列、数値、その他のオブジェクトをごちゃ混ぜにして加えてもかまいません。配列自体もオブジェクトですので、ある配列を別の配列の要素にすることもできます。

```
cabinet = [123, "hello", false, ["apple", "orange"]]
```

nil、true、false

Rubyには特殊なオブジェクトとして、nil、true、falseが用意されています。nilは「何もない」ことを表すオブジェクトです。trueは「真」、falseは「偽」を表すオブジェクトです。次の例に出てくるempty?メソッドの戻り値は、trueかfalseになります。

配列を操作するには、Arrayクラスのメソッドを使います。たとえば、lengthメソッドは配列の要素数を返します。配列が空かどうかを調べるには、empty?メソッドを使います。また、配列に要素を追加するには、<<演算子（メソッド）を使います。

LIST chapter02/2-1-4c.rb

```
1 colors = ["red", "blue", "yellow", "pink"]
2 puts colors.empty?
3 colors << "green"
4 puts colors.length
```

RESULT

```
false
5
```

配列の要素を順番に取り出すループの書き方は「2.2　条件分岐、メソッド、ブロック」の「繰り返しとブロック」、配列で使える便利なメソッドは「2.3　いろいろなオブジェクト」の「配列、ハッシュ、範囲」で紹介します。

%w(〜)という記法

%w( )の中にスペースで区切って文字列を並べると、配列を簡潔に記述できます。次の2行は同じ結果になります。文字列の中で式展開やバックスラッシュ記法を使いたいときは、大文字の%W( )を使います。( )は、< >のように好きな記号を使えます。

```
animals = ["dog", "cat", "elephant"]
animals = %w(dog cat elephant)
```

inspectとp

Rubyのすべてのオブジェクトでは、inspectメソッドが使えます。inspectメソッドは、オブジェクトの内容を読みやすい形にした文字列を返します。また、pメソッドはinspectメソッドの結果を表示するメソッドで、「p 変数」は「puts 変数.inspect」と同じ結果になります。この2つのメソッドは、デバッグやRubyの学習のために使えます。

```
arr = [1, 2, 3]
puts arr.inspect       # [1, 2, 3] と表示
obj = Object.new
p obj                  # #<Object:0x007fcd81842820> と表示
```

式と演算子

プログラムを組み立てるのに必要となる、式と演算子について見てみましょう。

■式

式とは、リテラルや変数、演算子、メソッドなどを組み合わせたものです。式を組み合わせたり並べたりすれば、Rubyのプログラムができます。次のものは式です。

```
"hello"              # リテラル（"hello"を返す）
name                 # 変数（nameの値を返す）
2 * 3                # 演算子式（掛け算の結果6を返す）
a > 0                # 条件式（trueかfalseを返す）
str.length           # メソッド呼び出し（メソッドの戻り値を返す）
if a > 0 then "OK" end  # if式（最後の式の値を返す）
```

式は何らかの値を返します。「式が値を返す」ことは、後述のHINT「irbを使おう」を実行してみれば実感できるでしょう。式は代入演算子=の右辺にできるほか、if式で条件式に使ったり、メソッドの引数に指定したりできます。

Rubyでは、式と文の区別はあまりありません。たいていの構文は何らかの値を返します。次のようにif式の結果を変数に入れることもできます。変数resultには、aが0以上なら"OK"、そうでなければnilがセットされます（if式については「2.2　条件分岐、メソッド、ブロック」を参照）。

```
result = (if a > 0 then "OK" end)
```

irbを使おう

Rubyファイルを書かずにRubyの機能を試すには、irbを使います。irbはRubyと一緒にインストールされるプログラムです。ターミナルからirbと入力してください。

```
$ irb
irb(main):001:0>
```

ただし、MSYS2/MinGW環境ではirbが正常に動作しませんので、次のコマンドを実行してからirbと入力してください。

```
$ alias irb='winpty ruby -e "require %{irb}; IRB.start(__FILE__)"'
```

ターミナルを開くたびに上記を実行する必要がありますので、エディタで~/.bashrcの末尾にこのコマンドを追加しておくとよいでしょう。なお、MSYS2/MinGW環境でirbを使う場合、日本語を入力できないという制限があります。

irbの上でRubyの式を入力すると、=>の次に式の値が表示されます。

```
irb(main):001:0> s = "hello"
=> "hello"
irb(main):002:0> s.length
=> 5
```

irbを終了するにはexitと入力します。

■演算子

式同士で計算や比較を行うには、**演算子**を使います。四則演算と余りの計算のための演算子には、+（足し算）、-（引き算）、*（掛け算）、/（割り算）、%（余り）があります。

文字列の連結には足し算と同じ+を使うことと、整数同士の割り算の結果は整数になることについては、すでに紹介しました。次は、四則演算と余りの計算の例です。

LIST chapter02/2-1-5a.rb

```
1 a = 7; b = 3
2 puts a + b      # 足し算
```

```
3 puts a - b     # 引き算
4 puts a * b     # 掛け算
5 puts a / b     # 割り算（整数同士の結果は整数）
6 puts a % b     # 余り
```

RESULT

```
10
4
21
2
1
```

+=（足して代入）のように「計算して代入する」演算子もあります。「n += 2」は「nの値に2を足したものを代入」ですので、nの値を2増やします。「n -= 1」は「nの値から1を引いたものを代入」ですので、nの値を1つ減らします。

LIST chapter02/2-1-5b.rb

```
1 n = 2
2 n += 2  # 2を足して代入
3 puts n
4 n -= 1  # 1を引いて代入
5 puts n
```

RESULT

```
4
3
```

演算子には、**優先順位**があります。たとえば「2 + 3 * 4」では*は+よりも優先されるため、「3 * 4」が先に計算されて結果は14になります。優先順位の低い演算子を先に優先さ

せるには、「(2 + 3) * 4」のようにかっこでくくります。

LIST chapter02/2-1-5c.rb

```
1 a = 2; b = 3; c = 4
2 puts a + b * c        # 掛け算が優先される
3 puts (a + b) * c      # 足し算が優先される
```

RESULT

```
14
20
```

Rubyでの演算子の種類と優先順位は、次のようになります。

演算子の優先順位

優先順位	演算子	機能
高い	+ ! ~	単項のプラス、否定、補数
	**	べき乗
	-	単項のマイナス
	* / %	乗算、除算、剰余
	+ -	加算、減算
	<< >>	ビットシフト
	&	論理積（ビット演算）
	| ^	論理和、排他的論理和（ビット演算）
	> >= < <=	比較演算子（大きい、小さい）
	<=> == === != =~ !~	比較演算子（等号）、パターンマッチ
	&&	論理積（かつ）
	||	論理和（または）
	.. ...	範囲
	?:	条件演算子
	= += -= *= /= %=	代入、計算して代入
	not	否定

低い	and or	論理積（かつ）、論理和（または）

演算子の直後で改行できる

Rubyでは、カンマや演算子の直後で改行すれば、文が続いていると見なされます（次の例の変数c）。変数dのように演算子の前で改行すると、「d = a」で文が終わっていると見なされ、dの値は2になります。変数eのように行の終わりに¥（Windowsでは¥）を置けば、文が続いていることを指定できます。

```
a = 2; b = 3
c = a +
b       # cは5
d = a
+ b     # dは2
e = a ¥
+ b     # eは5
```

ただし、メソッド呼び出しのピリオドの前では改行できます。次の例では、変数s2には"HELLO"が入ります。

```
s = "hello"
s2 = s
  .upcase
```

2.2 条件分岐、メソッド、ブロック

ここでは、Rubyでの条件分岐、ループ、メソッドの書き方を紹介します。if式を使って条件分岐のコードを書いたり、自分で作ったメソッドを呼び出したりしてみましょう。また、eachメソッドとブロックによる繰り返しについても紹介します。

条件分岐

「もし○○ならば□□をする」のように、条件によって処理を変えるには、**条件分岐**の構文を使います。Rubyでは、条件分岐のためにif式とunless式が用意されています。そのほかにcase式もありますが、本書では扱いません。

■条件式

「a == b」（aとbが等しいかどうか）のように、ある条件が成立しているかどうかを調べる式を**条件式**と言います。if式やunless式で条件分岐を行うときに使います。

```
if a == b
  puts "aとbは同じ！"
end
```

次にRubyのおもな比較演算子をまとめます。

比較演算子

演算子	意味	例
==	等しい	a ==

		b
===	等しい（case式で内部的に使われる）	a === b
!=	等しくない	a != b
>	より大きい	a > b
>=	等しいか、より大きい	a >= b
<	より小さい	a < b
<=	等しいか、より小さい	a <= b
<=>	右辺が左辺より小さいとき－1、等しいとき0、大きいとき1を返す（Arrayオブジェクトのsortメソッドなどで使われる）	a <=> b

「または」「かつ」「ではない」のように、比較演算子と組み合わせて使う論理演算子には、次のものがあります。&&のような記号の演算子と、andのような英単語の演算子の2種類が用意されています。本書のサンプルでは、andとorではなく、&&と||を使うことにします。

論理演算子

演算子	意味	例
&&	かつ	a > 1 && b > 2
and	かつ	a > 1 and b > 2
||	または	a > 1 || b > 2
or	または	a > 1 or b > 2
!	ではない	!(a == b)
not	ではない	not a == b

条件式の結果は、「c = (a == b)」のように取り出すことができます。正しい場合の結果はtrue（真）、正しくない場合はfalse（偽）になります。

次の例では、条件式の結果を変数bに入れています。入力された数値が0以上10未満だったらtrue、それ以外の場合はfalseが表示されます。

LIST chapter02/2-2-1a.rb

```
1 print "数を入力してください："
2 num = gets.to_i
3 b = (0 <= num && num < 10)
4 puts b
```

RESULT 5と入力した場合

```
数を入力してください：5
true
```

0と空文字列は真になる

「if num」のように、変数を条件式にした書き方を考えてみましょう。Rubyではnumが0のときは真になります。また、空文字列""も真になります。条件式が偽となるのは、falseかnilのときだけです。変数が0かどうか、空文字列かどうかで条件分岐を作るときは注意してください。

&&、||、and、orの優先順位

&&と||では、優先順位は&&のほうが高いのですが、andとorの優先順位は同じ、という点に注意してください。次の例の「a || b && c」では、「b && c」が優先されるので結果はtrueになります。「a or b and c」では、andとorの優先順位が同じなので左側の「a or b」が先に評価されて、結果はfalseになります。

```
a = true; b = true; c = false
a || b && c     # trueになる
a or b and c    # falseになる
```

■if式

「もし○○ならば□□をする」といった条件分岐を作るときは、if式を使います。if式では、条件式が正しい場合に実行されるプログラムを「if 条件式」と「end」で囲みます。if式だけでなく、unless式やメソッド定義、クラス定義、ブロックなど、Rubyの制御構造はすべてendで閉じます。

```
if 条件式
  条件式が正しい場合に実行するプログラム
end
```

次の例は、入力した数字が偶数なら「偶数です。」と表示するものです。

LIST chapter02/2-2-1b.rb

```
1 print "整数を入力してください："
2 num = gets.to_i
3 if num % 2 == 0
4   puts "偶数です。"
5 end
```

RESULT 100と入力した場合

```
整数を入力してください：100
偶数です。
```

このコードでputsの前に空白があることに注意してください。条件分岐やループ、メソッド、クラスの中では、プログラムを読みやすくするためにインデント（字下げ）を行います。Rubyプログラマは、タブ文字を使わずにスペース2個でインデントするのが一般的です。

if式を1行で書くときは、条件式の後ろにthenを書く必要があります。

```
if num % 2 == 0 then puts "偶数です。" end
```

次の例のように「if 条件式」を後ろに置くこともできます。Rubyプログラマは、この「後置のif」を頻繁に使います。

```
puts "偶数です。" if num % 2 == 0
```

「もし○○ならば□□を、もし◎◎ならば△△を、そうでなければ◇◇する」のように、条件を組み合わせるときは、elsifやelseを記述します。elsifのつづりに注意してください。「else if」や「elseif」は間違いです。

```
if 条件式1
  条件式1が正しい場合に実行するプログラム
elsif 条件式2
  条件式2が正しい場合に実行するプログラム
else
  条件式1も条件式2も正しくないときに実行するプログラム
end
```

次の例は、入力した数字が1500以上なら「送料無料です。」、0より大きく1500未満なら「送料300円です。」、それ以外なら「入力が間違っています。」と表示するものです。

LIST chapter02/2-2-1c.rb

```
1 print "価格を入力してください："
2 num = gets.to_i
3 if num >= 1500
4   puts "送料無料です。"
```

```
5 elsif 0 < num && num < 1500
6   puts "送料300円です。"
7 else
8   puts "入力が間違っています。"
9 end
```

RESULT 1000と入力した場合

```
整数を入力してください：1000
送料300円です。
```

　if式のほかに、条件式が正しくないときにプログラムを実行するunless式もあります。unless式ではif式と同様にelseを記述できますが、elsifにあたるものは記述できません。

```
unless num % 2 == 0
  puts "偶数ではありません。"
end
```

　後置のifと同様に後置のunlessも使えます。

```
puts "偶数ではありません。" unless num % 2 == 0
```

ifの中のローカル変数

　Rubyはif式やunless式のコードを読み込むと、そのコードを実行しない場合でも、その中にローカル変数があれば、ローカル変数を作成します。変数の値はnilになります。

　次の例は、変数numが奇数であっても、変数messageが作成されます。messageの値はnilなので何も表示されませんが、エラーにはなりません。

```
if num % 2 == 0
  message = "偶数です。"
```

```
end
puts message
```

メソッド

メソッドとは、コードのかたまりに名前を付けたものです。Rubyが用意しているメソッドだけでなく、自分で作ったメソッドを利用することもできます。

■メソッドの定義と呼び出し

Rubyでメソッドを作るには、次のように「def メソッド名」で始めて「end」で閉じます。メソッドに引数があるときは、メソッド名の後ろにかっこで囲んで並べます。

```
def メソッド名(引数, 引数, ……)
  メソッドの内容となるプログラム
end
```

次は、引数のないメソッドhelloの例です。メソッドの内容は、putsメソッドで2回文字列を出力するだけのものです。引数のないメソッドを呼び出すときは、helloのようにメソッド名を記述するだけで済ませることができます。

LIST chapter02/2-2-2a.rb

```
1 def hello
2   puts "こんにちは。"
3   puts "それではまた。"
4 end
5
6 hello
```

RESULT

```
こんにちは。
それではまた。
```

メソッド名には半角のアルファベット、数字、アンダースコア（_）を使い、1文字目は半角アルファベットまたはアンダースコアにしてください。メソッド名の最後には?か!を付けることもできます。

ローカル変数のスコープ

「スコープ」とは、変数が通用する範囲のことです。ローカル変数をメソッド内で作成すると、そのメソッド内でしか通用しません。つまり、メソッド外に同名の変数があると、別々の変数と見なされます。また、ブロックの中でもローカル変数はスコープを持ちます。なお、if式やunless式の中のローカル変数はスコープを持ちません。

```
num = 123
def method
  num = 456
  puts num
end

method          # 456と表示（メソッド内のnum）
puts num        # 123と表示（メソッド外のnum）
```

■引数と戻り値

メソッドに**引数**を持たせるには、メソッド定義で(と)の間に引数名をカンマで区切って並べます。**戻り値**を返したいときは、最後に変数名を1行書けば済みます。Rubyでは、メソッド内で最後に実行された式の値が戻り値になります。

次に示すのは、引数を2つ持ち、戻り値を返すメソッドの例です。引数baseとheightを元に、三角形の面積を計算します。resultの1行で変数resultの値が戻り値になります。

引数を持つメソッドを呼び出すときは、triangle(11, 9)のようにメソッド名の後ろに引数を並べます。

LIST chapter02/2-2-2b.rb

```
1 def triangle(base, height)
2   result = base * height / 2.0
3   result
4 end
5
6 area = triangle(11, 9)
7 puts "面積は、#{area}です。"
```

RESULT

```
面積は、49.5です。
```

最後の式の値が戻り値になるわけですから、次のように簡単に書いても同じです。

```
def triangle(base, height)
  base * height / 2.0
end
```

メソッドを呼び出すときは、「triangle 11, 9」のようにかっこを省略することもできます。これまでに使ってきたputsメソッドでは、かっこを省略していますが、puts("こんにちは")のようにかっこを付けてもかまいません。

return

メソッドの途中で戻り値を指定してメソッドから抜ける場合は、returnを使用します。次の例では、引数のどちらかが負の値であったら戻り値がnilになります。

```
def triangle(base, height)
  return nil if base < 0 || height < 0
  base * height / 2.0
end
```

?と!が付くメソッド

Rubyのメソッドの中には、メソッド名に?や!が付いているものがあります。?が付いたメソッドは、trueまたはfalseを返すことを表します。たとえば、StringクラスやArrayクラスのempty?メソッドは、文字列や配列が空かどうかを調べます。

```
if message.empty? then puts "空です。" end
```

!が付いたメソッドは、そのオブジェクトの中身を変更することを表します。たとえば、Stringクラスのchomp!メソッド（末尾の改行を取り除く）などです。!が付かない同名のメソッドは、オブジェクトを変更せずに新しいオブジェクトを返します。

```
line.chomp!          # lineの末尾の改行を取り除く
line2 = line.chomp   # 末尾の改行を取り除いた新しい文字列を返す
```

メソッド名に?や!を付けるのは、Rubyの文法というより慣習です。自分でメソッドを作るときも、メソッドの機能に合わせて?や!を付けるとよいでしょう。

繰り返しとブロック

ブロックは、Rubyのおもしろい特徴の1つです。ブロックを使いこなせば、繰り返しの処理をとても簡潔に記述できるようになります。

■ブロックの利用

Rubyでは、メソッドを呼び出すときに、**ブロック**と呼ばれるコードのかたまりを渡すことができます。ブロックは繰り返しの処理によく使われます。次のソースを書いて実行してみましょう。0から9までの数字が表示されます。

LIST chapter02/2-2-3a.rb

```
1 10.times do |i|
2   print i, ", "
3 end
```

RESULT

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

10.timesの10は、10という値を持つ数値オブジェクトで、timesは数値オブジェクトのtimesメソッドの呼び出しです。timesメソッドには、ブロックを渡すことができます。

ブロックとは、doからendまでのプログラムのひとかたまりの部分です。ブロックを渡されたtimesメソッドは、数値の回数（10）だけブロックを繰り返し実行します。

また、ブロックには|と|で囲んでブロックパラメータを指定できます。上記の例ではiがブロックパラメータです。timesメソッドは、iの値を0から9まで1つずつ増やしながら、繰り返しブロックを実行します。その結果「print i, ", "」でiの値が10回表示されます。

ブロックの呼び出し

ブロックは、doとendの代わりに{と}で囲むこともできます。上記の例は、次のように記述できます。

```
10.times { |i| print i, ", " }
```

数値オブジェクトには、指定された数まで値を増やしながら繰り返すuptoメソッドもあります。次の例は2から6までの数値を表示するものです。メソッドに引数とブロックを両方渡すときは、「メソッド名(引数) do ～ end」と記述します。

LIST chapter02/2-2-3b.rb

```
1 2.upto(6) do |i|
2   print i, ", "
3 end
```

RESULT

```
2, 3, 4, 5, 6,
```

数値オブジェクトの繰り返しメソッド

uptoメソッドの代わりにdowntoメソッドで6.downto(2)とすると、6から2まで数を減らしながら表示できます。stepメソッドで0.step(10, 2)とすると、0から10まで数を2つずつ増やしながら繰り返します。

■eachメソッド

配列（Array）には、ブロックを使った便利なメソッドがあります。代表的なものがeachメソッドです。eachメソッドは、ブロックパラメータ（次の例ではitem）に配列の要素を入れながら、配列の要素数だけブロックを繰り返し実行します。配列の要素を順番に処理するには、eachメソッドを使うのが普通です。

LIST chapter02/2-2-3c.rb

```
1 arr = ["apple", "orange", "grape"]
2 arr.each do |item|
3   print item + ", "
4 end
```

RESULT

```
apple, orange, grape,
```

配列の要素とともに要素の番号も扱いたいときは、each_with_indexメソッドを使います。ブロックパラメータには、「|item, i|」のように要素とその番号が入ります。

LIST chapter02/2-2-3d.rb

```
1 arr = ["apple", "orange", "grape"]
2 arr.each_with_index do |item, i|
3   print "#{i}.#{item}"
4   print ", " if i < arr.length - 1
5 end
```

RESULT

```
0.apple, 1.orange, 2.grape
```

■繰り返し以外のブロック

ブロックを受け取るメソッドは、繰り返しのために使われるとは限りません。たとえば、Fileクラスのクラスメソッドopenは、ブロックを使って呼び出せます。openメソッドは、「ファイルを開く→ブロックパラメータにFileオブジェクトを渡してブロックのコードを実行→ファイルを閉じる」という処理をします。ファイルを閉じるコードを書かなくてもよいので便利です。

次の例は、output.txtというファイルを作ってその中に「こんにちは」という文字列を書き込みます。

LIST chapter02/2-2-3e.rb

```
1 File.open("output.txt", "w", encoding: "utf-8") do |file|
2   file.puts "こんにちは"
3 end
```

RESULT output.txtの内容

```
こんにちは
```

openメソッドの第2引数"w"は書き込み用にファイルを開くことを示します。読み込み用なら"r"とするか、第2引数を省略します。第3引数「encoding: "utf-8"」はハッシュでファイルの文字コードを指定しています（ハッシュについては2.3節を参照）。

Railsにも、繰り返し以外の目的でブロックを受け取るメソッドがあります。代表的なものは、フォームを作成するform_forメソッド（Chapter 6）です。

例外処理

例外処理とは、プログラムの実行中に何らかのエラー（例外）が発生したときに、そのエラーを捕まえて後始末を行うしくみです。Rubyでは例外処理を「begin 〜 end」で記述します。この中で例外が発生すると、プログラムの実行が中断されて、すぐにrescue以下の節が実行されます。rescue節では、エラーメッセージの表示などを記述します。

rescue節の下にはelse節（省略可能）を置いて、例外が発生しなかった場合の処理を記述することもできます。

```
begin
  例外が発生する可能性のあるプログラム
```

```
rescue
  例外の発生後に処理するプログラム
else
  例外が発生しなかった場合のプログラム
end
```

メソッドの中では、beginなしでrescue節やelse節を置いて例外処理を行うこともできます。

```
def メソッド名(引数)
  例外が発生する可能性のあるプログラム
rescue
  例外の発生後に実行するプログラム
else
  例外が発生しなかった場合のプログラム
end
```

次の例は、ファイルからデータを1行読み込んで表示するものですが、ファイル操作の部分をbeginブロックで囲んでいます。存在しないファイルを開いたときは、例外が発生してrescue節に処理が移ります。「print f.gets」は実行されません。この例のrescue節では、warnメソッドで「エラー発生！」という文字列を表示しています。warnメソッドはターミナルに警告メッセージを出力しますが、プログラムの実行を止めることはありません。

LIST chapter02/2-2-4a.rb

```
1 begin
2   File.open("some.txt", encoding: "utf-8") do |f|
3     print f.gets
4   end
5 rescue
```

```
6   warn "エラー発生！"
7 end
```

RESULT

```
エラー発生！
```

例外が発生したときは、例外を表すオブジェクトを調べ、例外の種類やメッセージを取り出すことができます。例外オブジェクトを調べるには、rescue節を「rescue => 変数」のように記述します。

次の例では、変数eに例外オブジェクトを取り出しています。例外オブジェクトのクラス名はErrno::ENOENT、例外のメッセージは「No such file or directory……」であることがわかります。

LIST chapter02/2-2-4b.rb

```
1 begin
2   File.open("some.txt", encoding: "utf-8") do |f|
3     print f.gets
4   end
5 rescue => e
6   warn "#{e.class} / #{e.message}"
7 end
```

RESULT

```
Errno::ENOENT / No such file or directory @ rb_sysopen - some.txt
```

例外を発生させるには

プログラムの中でエラー処理のために自分で例外を発生させるには、raiseメソッドを使います。raiseメソッドに文字列を指定すると、RuntimeErrorクラスによる例外が発生します。

```
begin
  raise "ファイルがない！" unless File.exist?("some.txt")
rescue => e
  warn e.message
end
```

2.3 いろいろなオブジェクト

Rubyには、数値や文字列のほかに便利なオブジェクトがいろいろ用意されています。そうしたオブジェクトのうち、Railsの開発でよく使うことになるシンボル、配列とハッシュ、日時と日付の扱い方について紹介します。

シンボル

Rubyの**シンボル**は、「名前」を表すオブジェクトです。シンボルを作るには:catのように名前の前にコロンを付けます。「シンボルとは何か」を説明をする前に、シンボルのおもな使い方を紹介しましょう。

シンボルは、メソッドや変数の名前を示すのに使われます。これはシンボルのもともとの使い方です。たとえば、respond_to?メソッドにシンボルを渡すと、そのオブジェクトがメソッドを持っているかどうかを調べられます。

```
obj.find(1) if obj.respond_to?(:find)
```

メソッドや変数の名前だけでなく、アプリケーションで特別な意味を持つ名前を表すのにもシンボルがよく使われます。次の例は、Chapter 5で紹介するリンクの作り方で、シンボルでURLのパスを表しています。

```
link_to "会員一覧", :members
```

Railsの開発でよく使うことになるのは、ハッシュのキーをシンボルにすることです。次の例はChapter 3のもので、link_toメソッドの第3引数にハッシュでオプションを渡しています。「class: "menu"」は、「:class => "menu"」と書いても同じです。

```
link_to "Home", root_path, class: "menu"
```

シンボルとは何か、簡単に説明しておきましょう。シンボルは文字列とよく似ていますが、その正体は「文字列を整数で表したもの」です。

- シンボルを作ると、Rubyの内部では「文字列→一意の整数」の変換が行われ、整数値として管理される。
- シンボルは、1つの文字列が1つのオブジェクトに1対1で対応する。
- シンボルの内容は変更できない。upcase!のような!付きのメソッドを持たない。

文字列の代わりにシンボルを使うと、プログラムの実行を少々効率化できます。しかし、多くのRubyプログラマは、むしろソースコードを読みやすくするためにシンボルを使います。シンボルを使えば、ある文字列が特別な意味を持つ「名前」を表していることを明示できます。また、"name"よりも:nameのほうが書きやすい、という理由もあります。

シンボルのリテラル

シンボルの名前にはどんな文字でも使えますが、「:名前」の名前の部分は、変数名やメソッド名に使えるものしか使えません。:1234のように数字で始めることはできません。:1234のような名前のシンボルを作りたいときは、:"1234"と記述します。

変数名やメソッド名で使う@や?は、:@memberや:empty?のようにそのまま記述できます。

配列、ハッシュ、範囲

ここでは、配列、ハッシュ、範囲の使い方を紹介します。この3つは、「オブジェクトの集まりを扱うオブジェクト」です。

■配列の便利なメソッド

これまで何度か紹介してきた配列（Arrayオブジェクト）には、検索用のメソッドがいくつも用意されています。include?メソッドは、引数が配列に含まれているときにtrueを返します。

```
fruits = ["apple", "orange", "banana"]
puts "OK" if fruits.include?("banana")
```

if式で「XがA、B、C、……のどれかの場合」という条件を記述するときは、配列とinclude?を使うと簡単に書けます。

```
print "合い言葉："
word = gets.chomp
if ["apple", "orange", "banana"].include?(word)
  puts "OK"
end
```

in?メソッド

include?とは逆に、あるオブジェクトが配列に含まれているかどうかを調べるin?メソッドもあります。fruits.include?("banana")は"banana".in?(fruits)とも書けます。ただし、in?メソッドはRubyのものではなく、RailsのActive Supportが用意しているメソッドです。

配列には、ブロックを使って条件を調べるためのメソッドがたくさんあります。all?メソッドで配列の「すべての要素が条件を満たす」かどうかを、any?メソッドで「どれか1つが条件を満たす」かどうかを調べられます。

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
puts "OK" if numbers.all? { |item| item > 0 }       # すべて0より大きいか
puts "OK" if numbers.any? { |item| item % 2 == 0 }  # 偶数が含まれるか
```

ある条件を満たす要素を1つ取り出すには、detectメソッド（別名findメソッド）を使います。条件を満たす要素をすべて取り出して配列にするには、find_all（別名selectメソッド）を使います。

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
number = numbers.detect { |item| item > 5 }    # 5より大きい数1つ
numbers2 = numbers.find_all { |item| item > 5 } # 5より大きい数すべて
```

mapメソッド（別名collectメソッド）は、ブロックの戻り値を集めた新しい配列を作ります。頻繁に使われるメソッドですので、覚えておきましょう。次の例では、文字列を大文字に変えた配列を作ります。

LIST chapter02/2-3-2a.rb

```
1 fruits = ["apple", "orange", "banana"]
2 big_fruits = fruits.map { |item| item.upcase }
3 p big_fruits
```

RESULT

```
["APPLE", "ORANGE", "BANANA"]
```

「&:メソッド名」という書き方

上記の例は、次のように書き換えられます。メソッドの引数に&:methodを渡すと、「{ |item| item.method }」というブロックを渡すのと同じ結果になります。

```
big_fruits = fruits.map(&:upcase)
```

■ハッシュ

ハッシュ（連想配列）も「オブジェクトの集まりを扱うオブジェクト」で、Hashクラスのオブジェクトです。ハッシュと配列の違いは、値を入れたり取り出したりするときに、番号の代わりにキー（名前）を使うことです。

ハッシュの要素はキーと値のペアからなります。ハッシュを作成するには、{ }の間に「キー => 値」をカンマで区切って並べます。

次の例では、キーを国名の文字列、値を人口としたハッシュを作っています。「キー => 値」の間には改行を入れていますが、改行しなくてもかまいません。

LIST chapter02/2-3-2b.rb

```
1 population = {
2   "France" => 65027000,
3   "Germany" => 81768000,
4   "Italy" => 60705991
5 }
6 puts population["Italy"]        # キーが"Italy"の値
7 population["Japan"] = 127760000 # 新しい要素を追加
8 puts population["Japan"]        # キーが"Japan"の値
```

RESULT

```
60705991
127760000
```

ハッシュのキーがシンボルのときには、「:キー => 値」の代わりに「キー: 値」という形で並べられます。本書ではこの書き方を使います。キーとコロンの間にはスペースを入れないようにご注意ください。

```
population = { fr: 65027000, de: 81768000, it: 60705991 }
puts population[:it]
```

ハッシュの要素（キーと値）を順に取り出すには、eachメソッドを使います。キーと値の2つがeachメソッドのブロックパラメータに入ります。

次の例は、カンマ区切りのファイルを読み込んでデータをハッシュに取り込むものです。

LIST chapter02/2-3-2c.rb

```
books = {}
File.open("books.txt", encoding: "utf-8") do |f|
  f.each_line do |line|
    cols = line.chomp.split(",")
    books[cols[0]] = cols[1]
  end
end

books.each do |key, val|
  puts "#{key}、#{val}円"
end
```

LIST chapter02/books.txt

```
坊ちゃん,300
こころ,380
明暗,700
```

RESULT

```
坊ちゃん、300円
こころ、380円
明暗、700円
```

ここで使っているeach_lineはFileクラスのメソッドで、ファイルの内容を1行ずつ読み込んで、文字列を返します。splitメソッドはStringクラスのメソッドで、文字列を区切り文字で分

割して配列を返します。

ハッシュの要素の順番

Ruby 1.8までのハッシュには要素の順番がありませんでした。eachメソッドでキーと値を取り出すときに、どのような順番になるかは不定でした。Ruby 1.9以降のハッシュには順番があります。要素は{ }の間に指定した順で並べられ、あとから追加した要素は最後に加えられます。

■ハッシュとメソッドの引数

ここで、ハッシュがメソッドの引数になるときの注意点について述べます。次の例で、triangleメソッドはハッシュを引数にしています。こうしたメソッドでは、「base: 2.3, height: 3.4」のように順番を逆にしても、同じ結果になるので便利です。

LIST chapter02/2-3-2d.rb

```
1 def triangle(params)
2   params[:base] * params[:height] / 2.0
3 end
4 area = triangle(height: 3.4, base: 2.3)
5 puts "三角形の面積：#{area}"
```

RESULT

```
三角形の面積：3.9099999999999997
```

ハッシュを引数にしたメソッドの呼び出しを見てみましょう。

```
triangle({ height: 3.4, base: 2.3 })
```

ハッシュを引数の最後に指定するときは、{と}を省略できます。

```
triangle(height: 3.4, base: 2.3)
```

さらに、Rubyではメソッド呼び出しの( )を省略できるので、次のようにも書けます。

```
triangle height: 3.4, base: 2.3
```

Railsでは、ハッシュの引数を持つメソッドがたくさん用意されています。Chapter 3のlink_toメソッドやChapter 4のwhereメソッドなどです。そうしたメソッドを呼び出すときは、かっこを省略することがよくあります。省略によって直感的でシンプルなコードが書けますが、わかりにくいときはかっこを記述してください。

■範囲

範囲は、配列やハッシュと同じく集まりを表すオブジェクト（Rangeクラスのインスタンス）です。範囲を作るには、最初の要素と最後の要素をピリオド2つでつなげます。次の範囲は、1から10までの整数を表します。

```
range = 1..10
```

最後の要素を含めないときは、ピリオド3つを使います。次の範囲は、1から9までの整数を表します。

```
range = 1...10
```

文字列を範囲に使うこともできます。次の例は、aからhまでのアルファベット1文字を表します。

```
range = "a".."h"
```

eachメソッドを使えば、範囲の中の要素を順に取り出せます。次の例は、7の段の掛け算を4つだけ表示します。

LIST chapter02/2-3-2e.rb

```
1 (1..4).each { |num| puts "7 x #{num} = #{7 * num}" }
```

RESULT

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
```

■日時と日付

Railsアプリケーションにおいて、**日時**と**日付**はやや異なった扱いになります。日時とはRailsを構成するパッケージのひとつActive Supportが提供するActiveSupport::TimeWithZoneクラスのインスタンスです。本書ではこれを「日時オブジェクト」と呼びます。

このオブジェクトはクラス名が示すとおりタイムゾーン（時間帯）の情報を持ちます。日時オブジェクトは、過去から未来に向かって流れる時間軸におけるある一点を指します。つまり、1日のうちの特定の時刻（たとえば、正午）ではなく、あるタイムゾーンにおけるある日付と時刻の組み合わせを指します。たとえば「日本時間における2018年1月1日午前9時30分15秒」のような時点を表します。デフォルトのタイムゾーンはシステム（Railsアプリケーションが動作しているコンピュータ）で設定されているものになります。タイムゾーンを切り替える方法については、本書では扱いません。

Ruby本体にも日時を表すTimeクラスが存在しますが、通常Railsアプリケーションの開発でTimeクラスのインスタンスを直接扱うことはありません。ただし、Active Supportによって付け加えられたTimeクラスのメソッドを使うことはあります。たとえば、クラスメソッドTime.currentは「今」を表す日時オブジェクトを返します。

他方、日付を表すオブジェクトは、Rubyの標準ライブラリに含まれているDateクラスのインスタンスです。これはタイムゾーンとは無関係に「2018年1月1日」のような日付を表します。日時オブジェクトを日付オブジェクトに変換するにはto_dateメソッドを用います。次の例は、「今日」を表す日付オブジェクトを返します。

```
Time.current.to_date
```

Rails開発ではDate.todayメソッドを使わない

Dateクラスにも「今日」を返すtodayメソッドがありますが、Railsアプリケーションでは使わないほうがよいでしょう。このメソッドはRailsアプリケーションで設定されているタイムゾーンを考慮しないので、アプリケーションを国際化する際の妨げとなります。

日付オブジェクトや日時オブジェクトから年月日単位の値を取り出すには、それぞれyear、month、dayメソッドを用います。曜日はwdayメソッドで取り出せます（日曜が0、月曜が1、……、土曜が6）。

```
t = Time.current
puts t.month # 月を表示
puts t.wday  # 曜日を表示
```

また日時オブジェクトには時分秒単位の値を取り出すためのメソッドhour、min、secが用意されています。

```
t = Time.current
puts t.hour # 時を表示
puts t.min  # 分を表示
```

次の例は、読者のパソコンで設定されているタイムゾーンにおける「2018年4月21日20時12分25秒」を表す日時オブジェクトを作る式です。

```
Time.zone.local(2018, 4, 21, 20, 12, 25)
```

日時オブジェクトには、いくつかの便利なメソッドが備わっています。たとえば、yesterdayメソッドとtomorrowメソッドは、日時オブジェクトの指す時点を基準にして「24時間前」と「24時間後」を返します。また、last_weekメソッドとnext_weekメソッドは、「1週間前の同時刻」と「1週間後の同時刻」を返します。weekの部分をweek、month、yearで置き換えれば、それぞれ「前（次）の週の同時刻」、「前（次）の月の同時刻」、「前（次）の年の同時刻」を返します。次の例は「今から1か月後の時点の24時間後」を示す日時オブジェクトを返します。

```
Time.current.next_month.tomorrow
```

月の大小、うるう年

日時オブジェクトに対して月が関係する計算を行うときには、月の大小やうるう年のことを考慮に入れる必要があります。今日の日付が「2018年5月31日」あるいは「2018年5月31日」であれば、式Time.current.next_monthは「2018年6月30日の同時刻」を返します。また、今日の日付が「2020年1月29日」から「2020年1月31日」の間であれば、この式は「2020年2月29日の同時刻」を返します。

beginning_of_dayメソッドは、「その日の初め」を返します。weekの部分をweek、month、yearで置き換えれば、それぞれ「その週の初め」、「その月の初め」、「その年の初め」を返します。次の例は「明日の午前0時」を示す日時オブジェクトを返します。

```
Time.current.tomorrow.beginning_of_day
```

advanceメソッドに「days: 日数」を指定すると、その日数だけ日時を進めます。次の例は3日後の午前0時です。「hours: 3」（3時間後）、「days: -2」（2日前）、「month: 6」（6か月後）のような指定もできます。

```
Time.current.advance(days: 3).beginning_of_day
```

日付および日時を好きな形式の文字列に変換するには、strftimeメソッドを使います。引数の文字列に「%文字」を埋め込むと、そこが日付と時刻のパーツに置き換わります。次の例では、「年/月/日 時:分」という形式で現在の日時を表示します。

LIST chapter02/2-3-4.rb

```
1 require "active_support/all"
2
3 time = Time.current
4 puts time.strftime("%Y/%m/%d %H:%M")
```

RESULT

```
2018/04/23 11:58
```

上記のコードの1行目ではrequireメソッドによりActive Supportの機能をすべて取り込んでいます。単なるRubyスクリプトの中でTime.currentメソッドを利用するには、この準備作業が必要です。

strftimeメソッドで使えるおもな書式は次のとおりです。

strftimeメソッドのおもな書式

書式	機能
%a	英語の曜日の略名（Sun、Mon、……）
%A	英語の曜日名（Sunday、Monday、……）
%b	英語の月の略名（Jan、Feb、……）
%B	英語の月名（January、February、……）
%d	日
%H	時（24時間制）
%I	時（12時間制）
%m	月（1月は01、2月は02、……）

%M	分
%p	午前、午後（AMまたはPM）
%S	秒
%y	2桁の年
%Y	4桁の年
%Z	タイムゾーン名
%%	%の文字

2.4 クラス

Rubyは、あらゆるものをオブジェクトとして扱います。文字列や配列のようにあらかじめ組み込まれているクラスのオブジェクトだけでなく、自分でクラスを作ることもできます。自分用のクラスを書いたり使ったりする方法を見てみましょう。

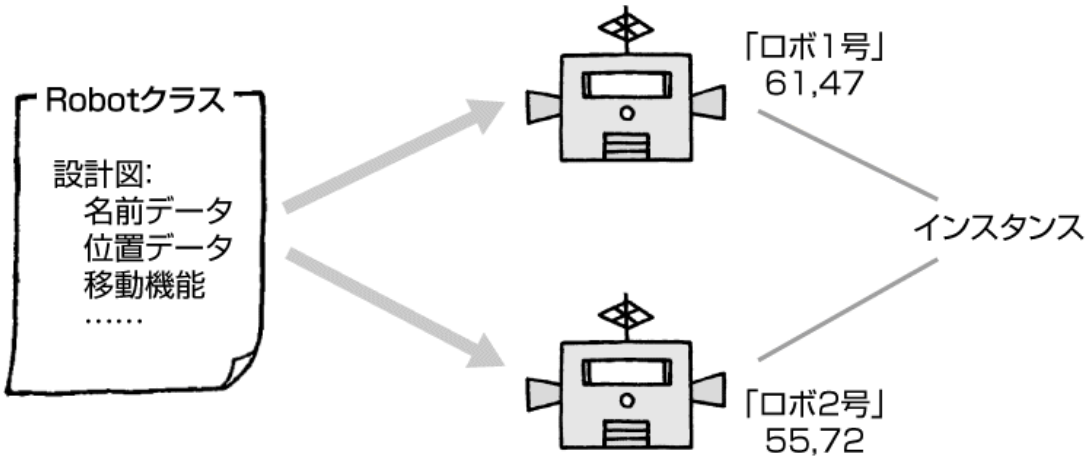
Rubyのオブジェクト

Rubyのオブジェクトはクラス（ロボットの設計図）とインスタンス（ロボット）からできています。自分でロボットクラスを定義したら、それをもとにロボットを作成して使うことができます。

■クラスとインスタンス

Rubyのようなオブジェクト指向言語での**オブジェクト**とは、「データと手続きをまとめたモノ」です。たとえば、ゲームソフトの中で「ロボット」というものをオブジェクトとして扱えば、名前やヒットポイントなどのデータや、移動や攻撃といった手続きを1つの単位にまとめることができ、効率よくプログラムが書けます。

オブジェクトについて考えるときは、クラスとインスタンスの2つに分けます。**クラス**は、オブジェクトの設計図です。設計図に基づいて実際にコンピュータのメモリ上に作られたオブジェクトを、**インスタンス**と呼びます。



クラスとインスタンス

クラスを自分で記述するには、「class クラス名 ～ end」という構文を使います。たとえば、ロボットを表すRobotというクラスは、次のように記述します（中身はまだ空です）。Rubyでは、クラス名は必ず大文字で始めることに注意してください。

クラスを記述すると、newメソッドでインスタンスを作成できます。

LIST chapter02/2-4-1a.rb

```
1 class Robot
2 end
3
4 robo1 = Robot.new
5 p robo1
```

RESULT

```
#<Robot:0x007fe1bd845b38>
```

newメソッドによって、Robotクラスの設計図に基づいたインスタンスが1つできて、変数robo1はそのインスタンスを指し示します。

本書では、「○○オブジェクト」という言葉を「○○クラスのインスタンス」の意味で使います。「robo1はRobotオブジェクト」と言ったら、「変数robo1はRobotクラスのインスタンスを

指す」ということです。

■参照とコピー

　前節までは=を何気なく使っていましたが、厳密に言うと代入演算子=は左辺の変数に「右辺の値への参照（オブジェクトリファレンス）」を代入します。つまり、左辺に右辺の値を参照させるのです。

　次の例を試してみましょう。「s2 = s1」とすると、s2がs1と同じインスタンスを参照することになります。object_idメソッドでオブジェクト番号を表示させると、同じインスタンスを指していることがわかります。

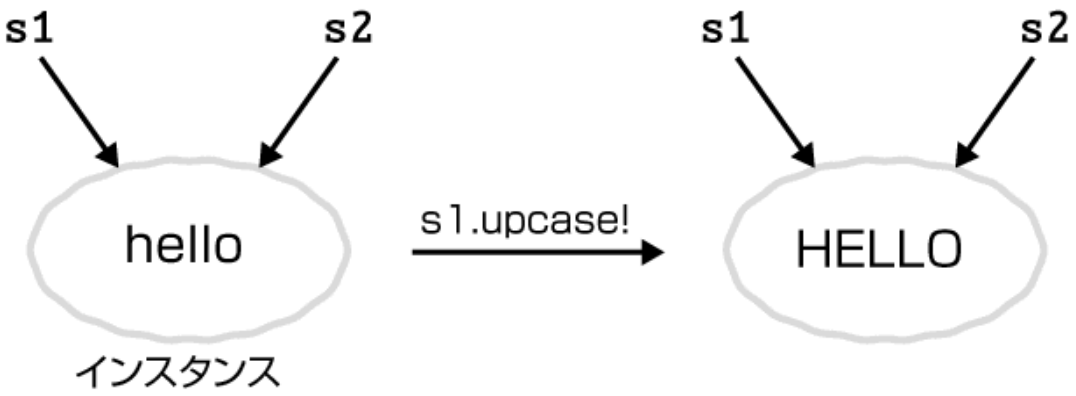
　s1.upcase!でs1の文字列を大文字に変更したあと、s2を表示すると、大文字になります。s1とs2が同じものを指しているためです。

LIST chapter02/2-4-1b.rb

```
1 s1 = "hello"
2 s2 = s1
3 puts s1.object_id
4 puts s2.object_id
5
6 s1.upcase!
7 puts s2
```

RESULT

```
70296032054580
70296032054580
HELLO
```



インスタンスの参照

複数の変数で別のオブジェクトを扱っているつもりなのに、実は同じものを指していたため、片方を変更するともう一方も変更されてしまった、ということがあります。そうした間違いを避けるには、dupメソッドでオブジェクトの複製を作ります。次の例では、s1とs2は別のインスタンスを参照し、持っている文字列はどちらも"hello"になります。dupメソッドは、すべてのオブジェクトに用意されています。

```
s1 = "hello"
s2 = s1.dup
```

なお、整数（Integer）、シンボル、true、false、nilは例外で、同じ値なら同じオブジェクトになります。

オブジェクトの種類を調べるには

ある変数が指しているオブジェクトがどのクラスのインスタンスであるかを調べるには、次のメソッドが使えます。

- classメソッド：クラスを返します。
- kind_of?メソッドまたはis_a?メソッド：オブジェクトが引数に指定したクラスのインスタンスなら、trueを返します。親クラスを指定してもtrueになります。
- instance_of?メソッド：引数に指定したクラスのインスタンスなら、trueを返します。親クラスを指定するとfalseになります。

```
s1 = "hello"
puts s1.class                  # String
puts s1.kind_of?(String)       # true
puts s1.kind_of?(Object)       # true
puts s1.instance_of?(String)   # true
puts s1.instance_of?(Object)   # false
```

インスタンスメソッド

インスタンスごとにデータを持たせるには、インスタンス変数を使います。同じクラスのインスタンス間で共通の手続きは、インスタンスメソッドで実装します。

■初期化とメソッド

クラスの中に「def メソッド名 ～ end」を記述すると、**インスタンスメソッド**ができます。インスタンスメソッドは、「変数.メソッド名」のように呼び出して、オブジェクトから情報を得たり、オブジェクトを操作したりできます。

次の例は、現在位置を移動するmoveメソッドと、ロボットの情報を文字列で返すto_sメソッドをRobotクラスに加えるものです。ロボット名は変数@nameに、現在位置の座標は@xと@yに入っていることにします。

```
class Robot
  def move(x, y)
    @x += x; @y += y
  end

  def to_s
    "#{@name}: #{@x},#{@y}"
```

```
  end
end
```

@を付けた変数は、クラスのインスタンスごとに作られる**インスタンス変数**です。インスタンス変数@nameと@x、@yにあらかじめ名前と位置を入れておくために、初期化用のメソッドを作りましょう。

クラスの中には、initializeという初期化用のメソッドを置くことができます。initializeは、newでインスタンスを作るときに自動的に実行されます。

initializeメソッドの引数の数はいくつでもよいですし、なくてもかまいません。次の例では、引数nameの値をインスタンス変数@nameに保存します。また、@xと@yの値を0にしています。

```
class Robot
  def initialize(name)
    @name = name
    @x = @y = 0
  end
end
```

Robot.new("ロボ1号")のように引数付きでnewメソッドを呼び出すと、initializeの引数nameが「ロボ1号」になります。

initialize、move、to_sの各メソッドをまとめると、次のプログラムができます。2つのRobotオブジェクトを作成し、それぞれの情報を表示するものです。

LIST chapter02/2-4-2a.rb

```
1 class Robot
2   def initialize(name)
3     @name = name
4     @x = @y = 0
```

```
  end

  def move(x, y)
    @x += x; @y += y
  end

  def to_s
    "#{@name}: #{@x},#{@y}"
   end
end

robo1 = Robot.new("ロボ1号") # ロボットのインスタンス1
robo2 = Robot.new("ロボ2号") # ロボットのインスタンス2
puts robo1
robo2.move(10, 20)
puts robo2
```

RESULT

```
ロボ1号: 0,0
ロボ2号: 10,20
```

ロボ1号の現在位置は「0,0」のままですが、2号ではmoveメソッドで位置を変えているので「10,20」となります。このように、インスタンス変数を使えば同じクラスのオブジェクトに別々のデータを持たせることができます。

インスタンス変数の初期値はnil

ローカル変数aをまだ作成していないときは、「b = a」というコードを実行するとエラーになります。インスタンス変数@aでは、いきなり「b = @a」としてもエラーにはならず、bにはnilが入ります。作成していないインスタンス変数はnilになります。

■メソッドの呼び出し制限

Rubyでは、メソッド定義の前にpublic、protected、privateを付けることで、メソッドの呼び出しを制限できます。本書では、public付きのメソッドを**パブリックメソッド**、private付きのメソッドを**プライベートメソッド**と呼びます。

呼び出し制限

レベル	機能
public	メソッドはどこからでも呼び出せる。
protected	同じクラスやサブクラス内のメソッドの中だけで呼び出せる。レシーバ（後述）を付けても呼び出せる（本書では使わない）。
private	同じクラスやサブクラス内のメソッドの中だけで呼び出せる。レシーバを付けると呼び出せない。

次の例では、ロボットの現在位置が負の数になると、crashメソッドを呼び出します。crashメソッドはプライベートメソッドなので、Robotクラスおよびサブクラスの中でしか呼び出せません。インスタンスメソッドmoveの中では呼べますが、robo1.crashのようにクラスの外では呼べません。

LIST chapter02/2-4-2b.rb

```
1 class Robot
2   def initialize(name)
3     @name = name
4     @x = @y = 0
5   end
6
7   def move(x, y)        # パブリックメソッド
8     @x += x; @y += y
9     crash if @x < 0 || @y < 0
```

```
10   end
11
12   private def crash     # プライベートメソッド
13     puts "ドカン！"
14   end
15 end
16
17 robo1 = Robot.new("ロボ1号")
18 robo1.move(200, -100)   # エラーは発生しない
19 robo1.crash             # エラーが発生する
```

RESULT

```
ドカン！
2-4-2b.rb:20:in `<main>': private method `crash' called for
#<Robot:0x007f8d299a8a90 @name="ロボ1号", @y=-100, @x=200>
(NoMethodError)
```

ところで、プライベートメソッドの定義では次のようにprivateを前の行に置く書き方もあります。

```
private
def crash
  puts "ドカン！"
end
```

実は、この書き方のほうが一般的です。defの左にprivateを置く書き方はRuby 2.1（2014年末リリース）で導入された比較的新しい言語仕様で、いまだにあまり普及していないのです。しかし、独立した行にprivateとだけ書くと、次に独立したpublicまたはprotectedが出現するまでの間に定義されたメソッドはすべてプライベートメソッドになります。

defの左にprivateを置くようにすると、複数個のプライベートメソッドを連続して定義する場合にソースコードの記述量が少し多くなりますが、あるメソッドがプライベートなのかどうかを見分けやすくなりますし、プライベートメソッドの定義をそのままパブリックメソッドの前に移動できるというメリットも生まれます。本書では新しい書き方を採用します。

レシーバとself

robo1.moveのrobo1のように、メソッドを呼び出す対象をレシーバ（受け取るもの）と呼びます。Rubyでは、メソッドの呼び出しを「オブジェクトに対してメッセージを送信する」と考えるからです。レシーバを省略すると、現在のオブジェクトを表すselfがレシーバと見なされます。

属性の書き方

インスタンス変数@nameを作っても、オブジェクトの外からはrobo1.nameのようにインスタンス変数にアクセスできません。robo1.nameのような書き方をしたいときは、変数と同名のメソッドを用意する必要があります。

次のソースのnameメソッド（読み出しメソッド）は、戻り値として変数@nameを返すので、「name = robo1.name」で@nameを取り出せます。name=のようにメソッド名に=を付けると、代入演算子の代わりとなるメソッド（書き込みメソッド）になります。「robo1.name = "ロボ1号"」とするとname=メソッドが呼ばれ、代入演算子の右辺が引数になり、@nameが"ロボ1号"を指すようになります。

LIST chapter02/2-4-3a.rb

```
1 class Robot
2   def name            # 名前の読み出し
3     @name
4   end
5
6   def name=(name)     # 名前の書き込み
```

```
  @name = name
 end
end

robo1 = Robot.new
robo1.name = "ロボ1号"
puts robo1.name
```

RESULT

```
ロボ1号
```

オブジェクト内のデータにアクセスするには、メソッドを書かなければなりません。このような読み出しや書き込み用のメソッドを**アクセサメソッド**と呼び、アクセサメソッドでやり取りできるデータを**属性**と呼びます。属性の実体は変数ではなく、インスタンス変数とメソッドの組み合わせであることに注意しましょう。

アクセサメソッドをいちいち書くのは面倒なので、もっと簡単な書き方が用意されています。「attr_reader :name」と記述すると読み出し用メソッドが、「attr_writer :name」と記述すると書き込み用メソッドが、自動的に追加されます。

```
class Robot
  attr_reader :name
  attr_writer :name
end
```

読み書き両方のメソッドを作りたいときは、attr_accessorを使います。メソッドをいくつも作りたいときは、「attr_accessor :x, :y」のように複数並べます。

次の例は、Robotクラスに読み出し専用の属性nameと読み書きできる属性scoreを設定したものです。

LIST chapter02/2-4-3b.rb

```
class Robot
  attr_reader :name
  attr_accessor :score

  def initialize(name)
    @name = name
    @x = @y = 0
    @score = 10
  end
end

robo1 = Robot.new("ロボ1号")
robo2 = Robot.new("ロボ2号")
robo2.score = 90 # スコアを変更
puts robo1.name, robo1.score
puts robo2.name, robo2.score
```

RESULT

```
ロボ1号
10
ロボ2号
90
```

クラス内での=付きメソッドに注意

アクセサメソッドはrobo.nameのようにクラスの外から呼べますが、クラス内のほかのメソッドの中で呼ぶこともできます。クラス内ではレシーバを省略してnameだけで呼べます。

ただし、=付きのメソッドでレシーバを省略すると、ローカル変数と見なされてしまいます。=付きのメソッドを呼ぶときは、「self.name =」のようにレシーバselfを必ず付けてください。

```
def change_name(new_name)
  old_name = name       # nameメソッドの呼び出し
  name = new_name       # 注意！ nameはローカル変数になる
  self.name = new_name  # 正しいname=メソッドの呼び出し
end
```

クラスメソッドと定数

Rubyのクラスでは、インスタンス変数やインスタンスメソッドだけでなく、クラス自体に機能を持たせることができます。

■クラスメソッド

クラスのインスタンスではなく、クラス自体に特定の機能を持たせるには、**クラスメソッド**を作ります。クラスメソッドは「クラス名.メソッド名」のように呼び出せます。クラスメソッドを定義するには、クラスの中で「def self.メソッド名 ～ end」と記述します。

Railsのモデルでは、レコードの取り出し（Chapter 4）のように、データベースのテーブル全体を対象にするメソッドをクラスメソッドとして用意しています。次の例は、そうしたRailsのクラスメソッドをまねて、Robotクラスにクラスメソッドloadを持たせたものです。

loadメソッドは、カンマ区切りテキストを読み込んで、Robotオブジェクトの配列を返します。クラスメソッドの中では、selfはRobotクラスを指すので、16行目のnewはRobot.newと同じことになります。

LIST chapter02/2-4-4a.rb

```
1 class Robot
2   def initialize(name, x, y)
```

```
  @name = name
  @x = x; @y = y
end

def to_s
  "#{@name}: (#{@x}, #{@y})"
end

def self.load(fname)
  robots = []
    File.open(fname, encoding: "utf-8") do |f|
      f.each_line do |line|
        cols = line.chomp.split(",")
        robots << new(cols[0], cols[1].to_i, cols[2].to_i)
      end
    end
  robots
 end
end

robots = Robot.load("robots.txt")
robots.each { |r| puts r }
```

LIST chapter02/robots.txt

```
ロボ1号,83,14
ロボ2号,5,51
ロボ3号,78,66
```

RESULT

```
ロボ1号: (83, 14)
ロボ2号: (5, 51)
ロボ3号: (78, 66)
```

クラスメソッドの書き方

クラスメソッドは次のようにも記述できます。こちらの書き方が好きなプログラマも多いです。「class << self 〜 end」の間には、クラスメソッドを複数置くことができます。メソッド名にself.を付ける必要はありません。

```
class Robot
  class << self
    def load(fname)
      クラスメソッドの内容……
    end
  end
end
```

■クラス定数

定数は、参照先を変更できない変数です。Rubyでは、変数名をアルファベットの大文字で始め、=でオブジェクトを指せば定数になります。クラス定義の外側で定数を作ると、その定数はプログラムのどこからでも使えます。次のPIは、円周率を表す定数です。

```
PI = 3.14159
```

クラス定義の中で定数を作ると、そのクラス専用のデータである**クラス定数**になります。次の例では、Shapeクラスの中で定数PIを作っています。

次の例のクラスメソッドcircleは、引数rを半径として円の面積を返すものです。

LIST chapter02/2-4-4b.rb

```
1 class Shape
2   PI = 3.14159
3
4   def self.circle(r)
5     r * r * PI
6   end
7 end
8
9 puts Shape::PI
10 puts Shape.circle(5)
```

RESULT

```
3.14159
78.53975
```

クラス定数は、クラスの中ではインスタンスメソッドの中からも、クラスメソッドの中からもPIで参照できます。クラスの外部からはShape::PIのようにクラス名に::を付けて参照します。

継承とミックスイン

Rubyには、既存のクラスを拡張する方法がいろいろあります。オブジェクト指向プログラミングでは標準的なクラスの継承と、Rubyの特徴であるミックスインについて見てみましょう。

■継承

継承とは、既存のクラス（親クラス）を元に新しいクラス（サブクラス）を作成し、親クラスの機能をそっくり取り込む方法です。サブクラスに機能を追加すれば、親クラスとサブクラスの機能の両方が使えるようになります。

サブクラスを作成するには、クラスを記述するときに「class サブクラス名 < 親クラス名」とします。たとえば、Robot（通常のロボット）のサブクラスとしてFlyingRobot（飛行するロボット）を作るには、次のようにします。

```
class FlyingRobot < Robot
end
```

FlyingRobotクラスにzという属性を新しく加えてみましょう。moveメソッドでは、親クラスの属性x、yを使えるようにしつつ、z属性で高さを表せるようにします。

ここで使っているsuperは、親クラスの同名のメソッドを呼び出すものです。super(x, y)でRobotクラスのmoveメソッドを呼び出せば、「@x += x; @y += y」を繰り返し書かなくても、xとyの位置を変更できます。さらに「@z += z」を加えれば、moveメソッドの機能を拡張できます。

```
class FlyingRobot < Robot
  def move(x, y, z)
    super(x, y)
    @z += z
  end
end
```

実際のサンプルを見てみましょう。FlyingRobotクラスでは、moveのほかにinitializeとto_sの各メソッドも上書きして、親クラスの機能に新しい機能を追加しています。

サブクラスのメソッドの引数が親クラスと同じときは、superの引数を省略できます。FlyingRobotクラスのinitializeでsuperとしていますが、これはsuper(name)と同じことになります。

LIST chapter02/2-4-5a.rb

```
class Robot
  def initialize(name)
    @name = name
    @x = @y = 0
  end

  def move(x, y)
    @x += x; @y += y
  end

  def to_s
    "#{@name}: #{@x},#{@y}"
  end
end

class FlyingRobot < Robot
  def initialize(name)
    super
    @z = 0
  end

  def move(x, y, z)
    super(x, y)
    @z += z
  end

  def to_s
    super + ",#{@z}"
```

```
29   end
30 end
31
32 robo1 = FlyingRobot.new("飛行ロボ1号")
33 robo1.move(20, 10, 30)
34 puts robo1
```

RESULT

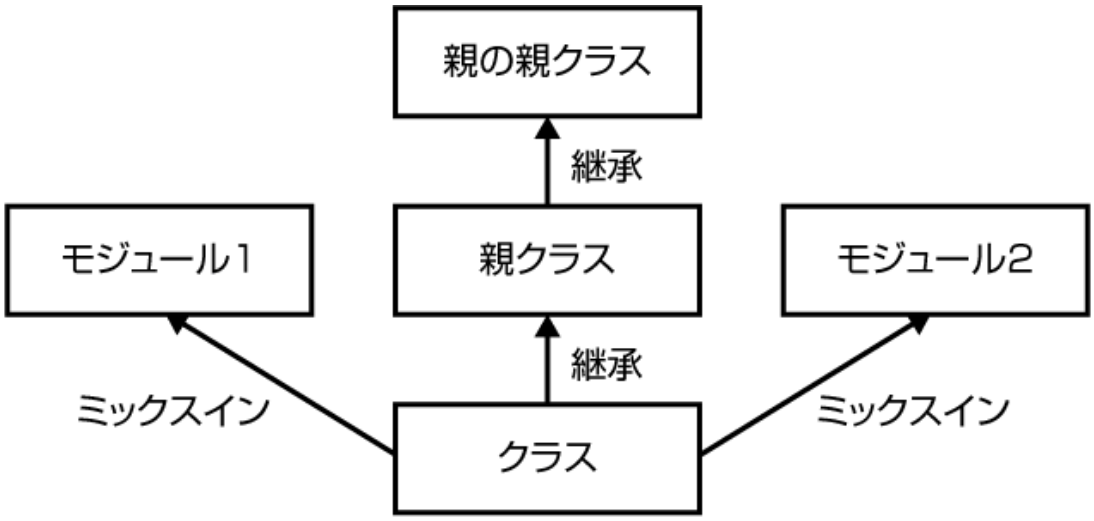
```
飛行ロボ1号: 20,10,30
```

■モジュールとミックスイン

Rubyには、クラスに似たものとして**モジュール**があります。「module モジュール名 ～ end」でモジュールを定義し、その中にメソッドを記述できます。モジュールにメソッドをまとめておけば、その機能をクラスに取り入れてクラスの機能を拡張できます。クラスにモジュールを取り込むことを**ミックスイン**と言います。

モジュールは、クラスとほとんど同じものです。モジュールがクラスと違うのは、継承ができないことと、newでインスタンスを作れないことです。モジュールは親クラスにはなれず、親クラスを持つこともできませんが、クラスにミックスインできます。

Rubyではクラスの継承は単一継承です。つまり、複数の親クラスを持つサブクラスを作れません。しかし、複数のモジュールをクラスに取り込むことができます。



継承とミックスイン

次のRadarモジュールは、ロボット間の距離を計算して返すメソッドdistance_toを備えています。この機能をRobotクラスに取り込む（ミックスインする）には、クラスの中で「include Radar」のようにincludeにモジュールを指定します。するとrobo1.distance_to(robo2)のように、Radarのインスタンスメソッドを利用できます。

distance_toメソッドを利用するには、Radarモジュールを取り込むクラスのオブジェクトが属性xとyを持っている必要があります。

LIST chapter02/2-4-5b.rb

```
1 module Radar
2   def distance_to(other)
3     Math.sqrt((self.x - other.x) ** 2 + (self.y - other.y) ** 2)
4   end
5 end
6
7 class Robot
8   include Radar
9   attr_accessor :name, :x, :y
10
```

```
11   def initialize(name)
12     @name = name
13     @x = @y = 0
14   end
15
16   def move(x, y)
17     @x += x; @y += y
18   end
19 end
20
21 robo1 = Robot.new("ロボ1号")
22 robo2 = Robot.new("ロボ2号")
23 robo2.move(12, 35)
24 puts "距離は #{robo1.distance_to(robo2)} です。"
```

RESULT

```
距離は 37.0 です。
```

Kernelモジュール

Kernelモジュールは、putsのようによく使われるメソッドを集めたモジュールです。ObjectクラスはKernelモジュールをミックスインしており、すべてのクラスはObjectクラスのサブクラスです。これにより、Rubyプログラムのどこからでもputsメソッドを呼び出せます。

ここまでの例で登場したKernelモジュールのメソッドには、gets、p、print、puts、raise、rand、require、warnがあります。

名前空間としてのモジュールとクラス

ミックスインのほかに、モジュールは名前空間として使うこともできます。モジュールの中でクラスを定義すれば、「モジュール名::クラス名」でクラスを参照できます。たとえば、RailsではActiveRecordモ

ジュールの中でBaseクラスを定義しています。このBaseクラスはActiveRecord::Baseで参照できるので、ActionController::Baseのようなほかの Baseクラスと区別できます。

```
module ActiveRecord
  class Base
  end
end
```

モジュールだけでなく、クラスの中でクラスを定義することもできます。「11.2　エラーページのカスタマイズ」では、ApplicationControllerクラスの中で独自の例外クラスを作成しています。

```
class ApplicationController < ActionController::Base
  class Forbidden < StandardError; end
end
```

Rubyのクラスの特徴

最後に、Rubyのクラスの実体について、種明かしをしておきましょう。Rubyでは、「class クラス名 ～ end」で記述したクラス自体もオブジェクトです。Robotはクラスの名前であると同時に、Robotクラスを表すオブジェクトを指す定数にもなります。

次のように、object_idメソッドでオブジェクト番号を表示してみると実感できるでしょう。Rubyでは、すべてのオブジェクトに1つずつ固有の番号が振られます。

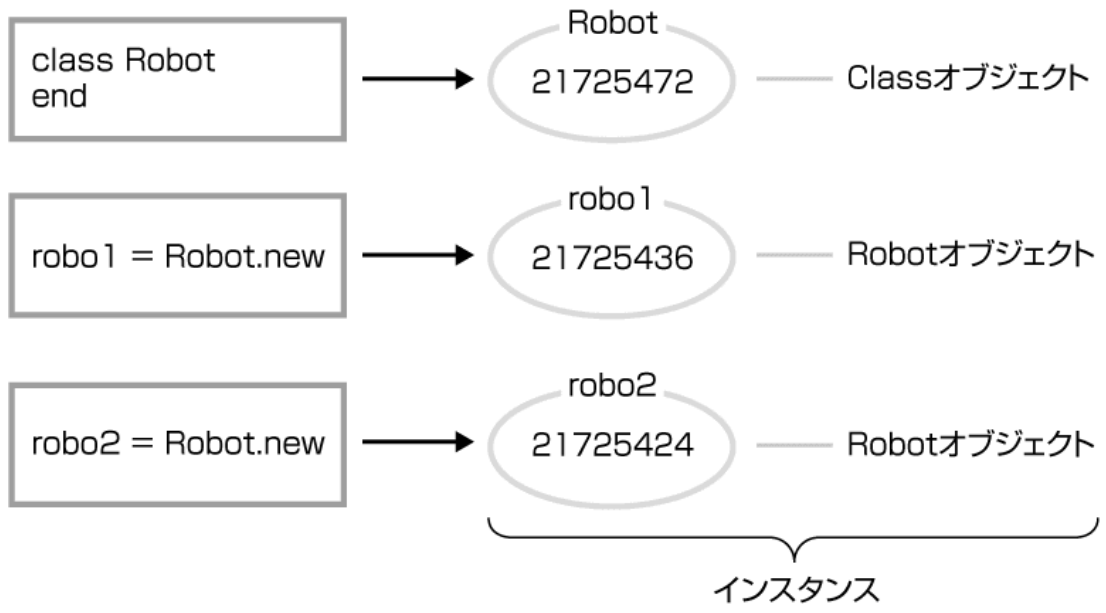
LIST chapter02/2-4-6.rb

```
1 class Robot
2 end
3
4 robo1 = Robot.new
5 robo2 = Robot.new
```

```
6 puts "Robot: #{Robot.object_id}"
7 puts "robo1: #{robo1.object_id}"
8 puts "robo2: #{robo2.object_id}"
```

RESULT

```
Robot: 70317616245600
robo1: 70317616245540
robo2: 70317616245520
```



3つのオブジェクトができる

定数Robotは、Robotクラスを表すオブジェクトを指しています。Robotクラスを表すオブジェクトもやはり、何かのクラスのインスタンスです。そのクラスとは、Classクラスです。Classクラスは「クラスに関する機能を備えたクラス」です。

RobotはClassクラスのインスタンスを参照する定数なので、普通の変数のように別の変数で参照したり、メソッドの引数にしたりできます。

```
klass = Robot       # 変数klassはClassオブジェクトを指す
r = klass.new       # Robot.newと同じ
```

```
r.kind_of?(Robot)    # クラスはメソッド引数にもなれる
```

次のようにクラス定義の直下にコードを書くと、クラスが読み込まれるときに「ここも実行されます」が表示されます。Rubyでは、「class ～ end」はクラスの定義というだけでなく、実行されるコードでもあります。

```
class Robot
  puts "ここも実行されます"
end
```

Chapter 2のまとめ

- Rubyでは、数値や文字列も含めてすべてをオブジェクトとして扱います。
- 条件分岐には、if式やunless式を使います。
- メソッドを作ると、コードをひとかたまりの部分に分けて呼び出せるようになります。メソッドには引数や戻り値を指定できます。
- 繰り返しの処理には、eachのようなブロックを受け取るメソッドを使います。
- オブジェクトの集まりを扱うために、配列やハッシュを利用できます。
- シンボルを使うとハッシュやメソッドの引数を読みやすくできます。
- クラスは、「class クラス名 ～ end」の間に記述します。
- クラスの属性を読み書きするにはアクセサメソッドを使います。
- クラスを拡張するには、継承やミックスインを使います。

練習問題

［A］　次の空欄を埋め、入力した数値に消費税を加えた値を表示してください。小数点以下を切り落とすには、数値オブジェクトのto_iメソッドを使います。なお、消費税率は8％とします。

```
print "価格を入力してください："
price = gets.chomp
price =
puts "税込み#{price}円です。"
```

［B］　配列flowersの要素を1つずつ表示するプログラムを書いてください。

```
flowers = ["carnation", "tulip", "cosmos"]

```

［C］　空欄を埋めて、属性を取り出すアクセサメソッドをBookクラスに加えてください。

```
class Book

  def initialize(title, author, price)
    @title = title
    @author = author
    @price = price
  end
end
```

```
book1 = Book.new("彼岸過迄", "夏目漱石", 540)
puts "#{book1.title}、#{book1.author}著、#{book1.price}円"
```

Part 2

Ruby on Railsの基本

このPartでは、Ruby on Railsでウェブアプリケーションを作るために知っておかなければならないことを解説します。モデル、ビュー、コントローラの使い方をしっかり学べば、誰でも自分のウェブアプリケーションをすばやく構築できるようになります。

Chapter

3　コントローラとビュー

　このChapterからは、いよいよRailsでウェブアプリケーションを作り始めます。RailsのMVCアーキテクチャのうち、まずはコントローラとビューの使い方を学びましょう。理解の鍵になるのは、「アクション」という概念です。この言葉に注目して読み進めてください。

これから学ぶこと

- HTTPプロトコルとRailsの動作のしくみについて、基本的な点を押さえておきます。
- URLとアクションを結び付けるルーティングについて学びます。
- コントローラが複数のアクションで構成されていること、アクションはそれぞれテンプレートと結び付いていることを学びます。
- コントローラとアクションの細かい機能をいくつか紹介します。
- テンプレートファイルの書き方を学びます。
- レイアウトテンプレートと部分テンプレートの使い方を学びます。
- Morning Gloryサイトの最初のバージョンを作成します。

コントローラとビューを使えば、Railsで簡単なウェブサイトを作成できます。この2つとウェブサイトにはどんな関係があるでしょうか？ この2つを使うには、まず何をすればよいでしょうか？

3.1 RailsとHTTPの基本

ここでは、Ruby on Railsの基本的な処理の流れについて解説します。Railsでは、URLの形式（パターン）とコントローラ、アクション、ビューが密接に関係していることを知っておいてください。

HTTPの基礎知識

HTTP（HyperText Transfer Protocol）は、ブラウザとサーバーがHTMLや画像などの情報をやり取りするときに使われるプロトコル（通信手段）です。Railsなどでウェブアプリケーションを構築する際には、HTTPのしくみをある程度押さえておく必要があります。

■リクエストとレスポンス

HTTPはとても単純なプロトコルです。ブラウザがサーバーにリクエスト（要求）を送り、サーバーはそれに応えてレスポンス（応答）を返します。次の例は、ブラウザが「http://localhost:3000/members/123」というURLをリクエストするときに送信する内容です。1行目は、「GET リソース HTTPのバージョン」です。ここで言う「リソース」とは、URLのパスとクエリーの部分です（次の「URL」の項を参照）。2行目以降はHTTPヘッダーと呼ばれ、ブラウザとサーバーに関する情報が書かれています。

実際のHTTPヘッダーの内容はもっと複雑ですが、ここではわかりやすいように単純化しています。

```
GET /members/123 HTTP/1.1
Host: localhost:3000
UserAgent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/53
7.36 (KHTML, like Gecko) Chrome/64.0.3282.167 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
image/apng,*/*;q=0.8
Accept-Language: ja,en-US;q=0.9,en;q=0.8
```

するとサーバーはそれに応えて次のような内容のレスポンスを送り返します。1行目は「HTTPのバージョン ステータスコード メッセージ」です。2行目以降はHTTPヘッダーです。ヘッダーの直後に1行空けて、実際のHTMLや画像の内容が送信されます。ブラウザはここからHTML文書や画像を読み取って表示します。

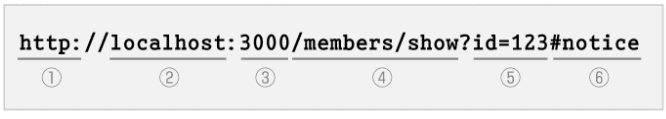
```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Cache-Control: max-age=0, private, must-revalidate

<!DOCTYPE html>
<html>
<head>
  <title>Asagao</title>
</head>
<body>
.....
</body>
</html>
```

HTTPとセッション

HTTPは基本的に1つのリクエストと1つのレスポンスだけで完結します。何度リクエストを送っても、「前回と同じ人のリクエストだな」とは認識されません。「同じユーザーに対して、リクエストのたびにそのユーザー用のレスポンスを返す」といったしくみ（セッション）を作るには、サーバー側で工夫が必要になります。Railsは洗練されたセッション管理機能を備えています（Chapter 8を参照）。

URL

ブラウザは、アドレス欄のURLを解釈してサーバーにリクエストを送ります。URLはいくつかのパーツに分かれています。ブラウザはサーバー名とポート番号を元にサーバーに接続し、パスとクエリーの部分を「GET /members/show?id=123」のようにリクエストします。



①スキーム（HTTPやFTPなどのプロトコルの種類）
②サーバー名（例:localhost、www.impress.co.jp）
③ポート番号（省略すると80）
④パス
⑤クエリー（省略可）
⑥フラグメント（ページ内のターゲット、省略可）

URLの各パーツの意味

普通のウェブサーバーは/index.htmlが要求されると、そのままサーバーのディスクにあるファイルindex.htmlを返します。Railsでは、パスとファイルが対応しているわけではありません。RailsはURLのパスとクエリーを解

釈してコントローラのアクションを呼び出し、レンダリングの結果をブラウザに返します。

パラメータをパスに含める

サーバー上のプログラムにパラメータを渡すには、「?名前=値」のようにURLのクエリー部分に情報を入れるだけでなく、次のようにパラメータをパス部分に入れてしまうこともできます。

```
http://localhost:3000/members/123
```

簡潔でわかりやすいURLは、Railsアプリケーションの特徴のひとつです。Railsでは、こうした形式のパラメータも簡単に取り出せるようになっています。

■GETとPOST

ブラウザからのリクエストの1行目「GET /members/123 HTTP/1.1」の「GET」のことを、HTTPでは「メソッド」と呼びます。一番よく使われるメソッドは**GET**ですが、ウェブアプリケーションでは**POST**もよく使われます。ブラウザのアドレス欄にURLを入力したり、リンクをクリックしたりすると、GETメソッドでリクエストが送られます。HTMLのフォームで送信ボタンを押したときは、GETまたはPOSTメソッドでリクエストが送られます。

POSTメソッドでは、次のようにリクエストを送ります。HTTPヘッダーのあとに1行空けて、「名前=値&名前=値」の形でデータを加えます。

```
POST /bbs HTTP/1.1
Host: localhost:3000
Content-Type: application/x-www-form-urlencoded
UserAgent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/53
7.36 (KHTML, like Gecko) Chrome/64.0.3282.167 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
image/apng,*/*;q=0.8
Accept-Language: ja,en-US;q=0.9,en;q=0.8

name=%E5%A4%AA%E9%83%8E&mail=taro%40example.com&comment=%E3%81%93
%E3%82%9
3%E3%81%AB%E3%81%A1%E3%81%AF
```

GETメソッドとPOSTメソッドの間には、形式上の違いだけでなく、役割上の違いもあります。一般に、GETメソッドはウェブサーバーからデータを取得するときに用い、POSTメソッドはウェブサーバーの状態を変更するときに用います。「掲示板にコメントを投稿する」、「商品をカートに入れる」などのリクエストは、POSTメソッドで行うのが標準的です。

その他のメソッド

HTTPには、GETとPOSTのほかに次のメソッドもあります。ただし、HTMLのフォームが対応しているのはGETとPOSTだけです。Railsでは、PATCHとDELETEにも役割を与えています（Chapter 5を参照）。

HTTPのメソッド

メソッド	機能
GET	リソースの要求
POST	リソースの送信
HEAD	ヘッダー部分だけを要求
PUT	リソースの置換
PATCH	リソースの一部更新
DELETE	リソースの削除

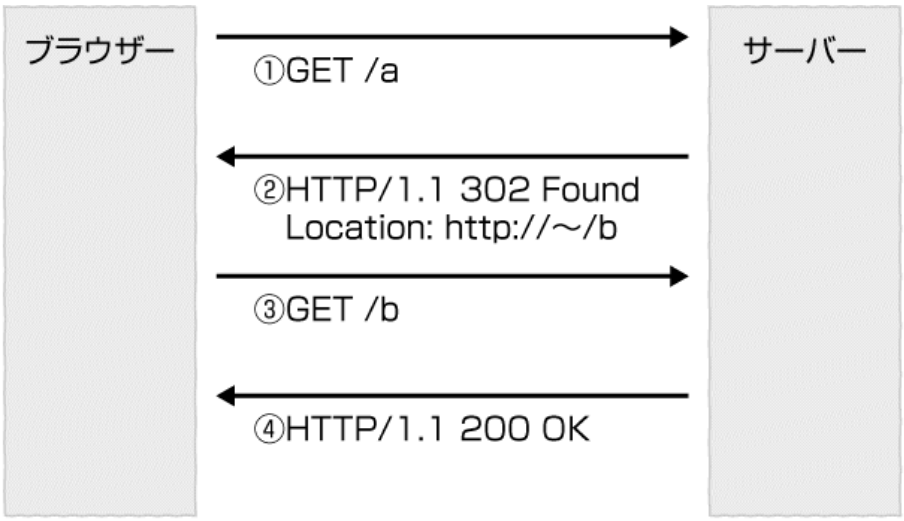
■リダイレクション

HTTPの機能のうち、ウェブアプリケーションでよく使われる**リダイレクション**について紹介しておきましょう。リダイレクションとは、ブラウザが要求したURLとは別のURLを示して、ブラウザに再要求させるしくみです。これを利用すると、ブラウザに表示されるページを強制的に移動することができます。

リダイレクションを行うには、サーバーのレスポンスのステータスコードを「200」ではなく、「301」、「302」、「303」または「307」とします。新しいURLはヘッダーの「Location」に指定します。

```
HTTP/1.1 302 Found
Content-Type: text/html; charset=utf-8
Location: http://localhost:3000/members/123
```

このレスポンスを受け取ったブラウザは、新しいURLでリクエストし直します。



リダイレクション

Railsでは、アクションの中でredirect_toメソッドを呼び出すことで、自動的にリダイレクションの処理ができます。「3.2　コントローラとアクション」の「リダイレクション」を参照してください。

ステータスコード

HTTPでは、成功したときも、リダイレクションを行うときも、何かエラーが発生したときも、まったく同じ形式でレスポンスを返します。1行目のステータスコードが変わるだけです。たとえば、要求されたリソースがサーバーになければ、おなじみの「404」を返します。

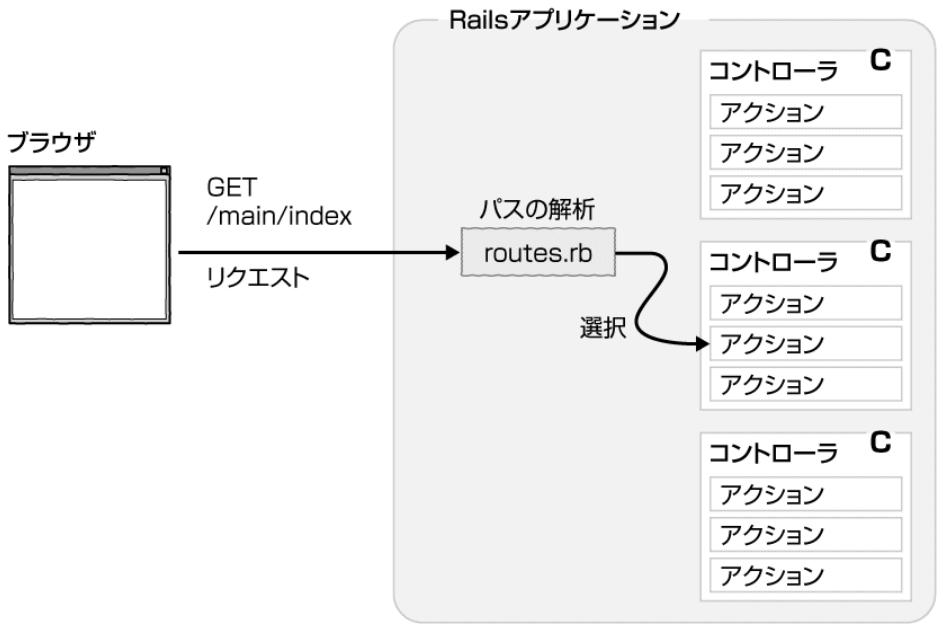
```
HTTP/1.1 404 Not Found
Content-Type: text/html; charset=utf-8

<html><head><title>見つかりません</title><head>……
```

このほかに、アクセス禁止の場合は「403 Forbidden」、サーバーエラーの場合は「500 Internal Server Error」を返します。

Railsのリクエスト処理の流れ

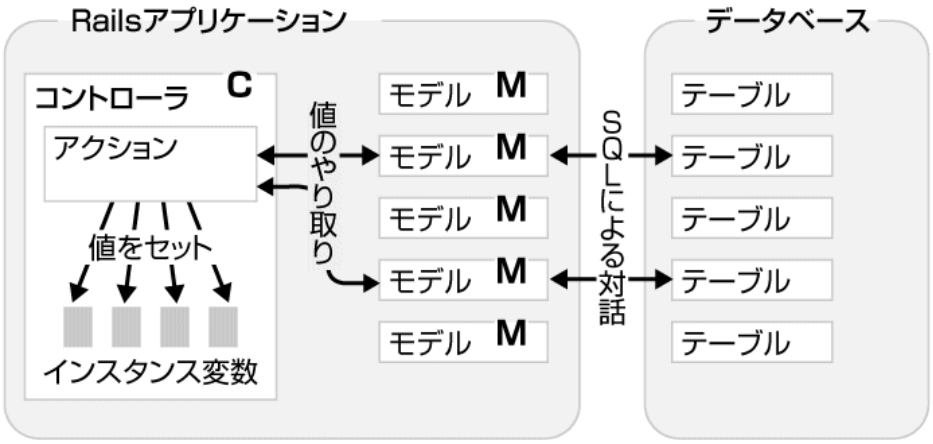
Railsで作成したウェブアプリケーションに対して、ブラウザがHTTPでリクエストを送ったときに、どのような動作が行われるのかをざっと見てみましょう。ブラウザからのリクエストを受けると、Railsはパスを調べ、routes.rbに従ってどのコントローラのどのアクションを選べばよいかを決めます。コントローラは複数あり、さらにコントローラ

の中にアクションが複数あります。routes.rbはconfigディレクトリの下に置かれているファイルで、編集するとパスとアクションの対応関係を変更できます。



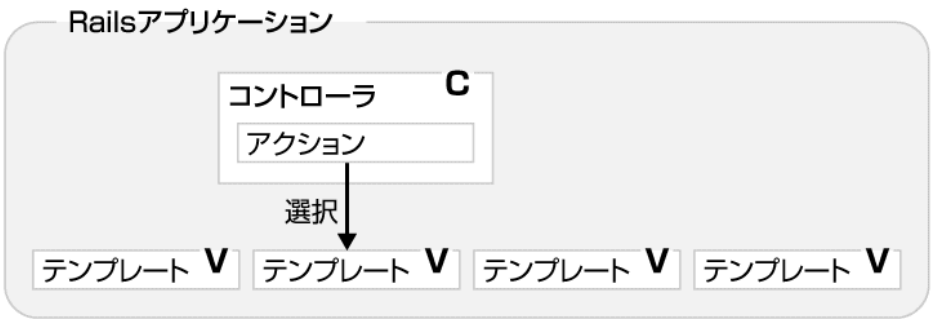
パスからアクションが選ばれる（ルーティング）

Railsは選ばれたアクション（メソッド）を実行します。アクションには、モデルとの間で情報のやり取りをするプログラムを書きます。モデルはデータベースのテーブルと対応しています。アクションはモデルから取得した情報のうち、表示に必要なものをインスタンス変数に保存します。



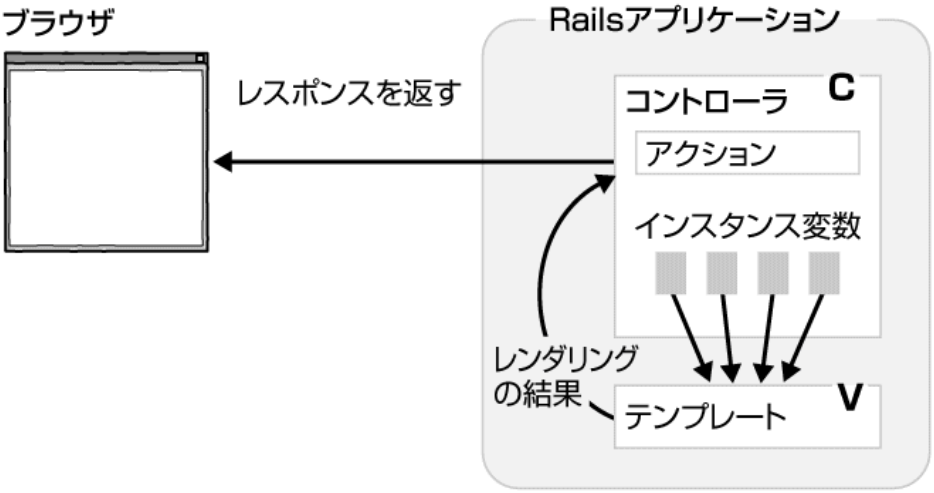
モデルを通じてデータベースにアクセスし、インスタンス変数に情報を保存

アクションは、ブラウザへのレスポンスを作成するためにビュー（テンプレート）を選びます。特に指定しない限り、アクションと同名のテンプレートが選ばれます。



アクションがテンプレートを選択する

選ばれたテンプレートは、コントローラのインスタンス変数を参照して、その値をHTMLソースの中に埋め込みます。この処理を**レンダリング**と呼びます。できあがったHTML文書は、コントローラによってブラウザに送り返され、画面に表示されます。



ビューはインスタンス変数を取り出してHTML文書を生成する

ルーティング

Railsでは、リクエストされたURLのパスから特定のコントローラとアクションを選ぶことを**ルーティング**と呼びます。ルート（route）は「経路」という意味です（「根」の意味のrootではありません）。

■アクションの追加

ルーティングの設定は、configディレクトリの下のファイルroutes.rbで行います。routes.rb自体もRubyプログラムです。「Rails.application.routes.draw do 〜 end」の間に加えたコードがルーティングの設定になります。

一例として、asagaoアプリケーションに新しいアクションを追加してみましょう。routes.rbに次の設定を追加します。これにより、「/about」というパスがTopControllerのaboutアクションと結び付きます。また、about_pathメソッドが「/about」を返すようになります。

LIST chapter03/config/routes.rb

```
1 Rails.application.routes.draw do
2   root "top#index"
3   get "about" => "top#about", as: "about"
    （以下省略）
```

TopControllerにaboutアクションを追加します。これは何もしない空のメソッドです。

LIST chapter03/app/controllers/top_controller.rb

```
1 class TopController < ApplicationController
2   def index
3     @message = "おはようございます！"
4   end
5
6   def about
7   end
8 end
```

app/views/topディレクトリの下でabout.html.erbを作成し、内容を次のようにします。インスタンス変数@page_titleは、あとでtitleタグを記述するためにも利用します（「3.4　モックアップの作成」の「モックアップのレイアウトテンプレート」を参照）。

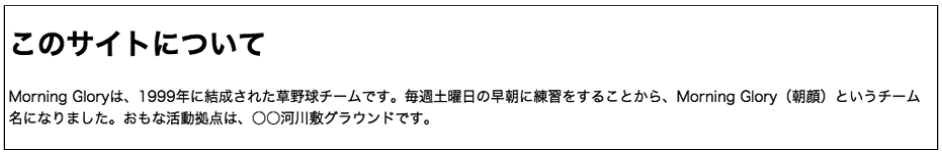
LIST chapter03/app/views/top/about.html.erb

```
1 <% @page_title = "このサイトについて" %>
2
3 <h1><%= @page_title %></h1>
4
5 <p>Morning Gloryは、1999年に結成された草野球チームです。毎週土曜日の早朝に練習をすることから、Morning Glory（朝顔）というチーム名になりました。おもな活動拠点は、○○河川敷グラウンドです。</p>
```

ターミナルで次の2つのコマンドを順に実行し、Railsサーバーを起動してください。

```
$ cd ~/rails/asagao
$ bin/rails s
```

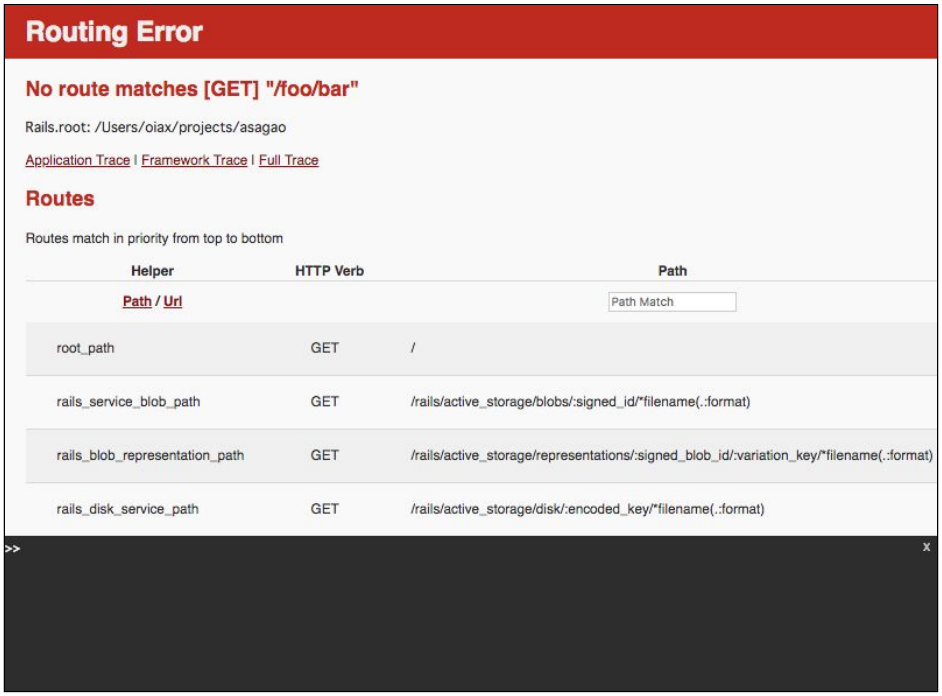
そして、ブラウザで「http://localhost:3000/about」というURLを開くと、自己紹介ページが表示されます。



RESULT /aboutで表示されるページ

■ルーティングエラー

ルーティングの設定にはないパスをブラウザで開こうとすると、ルーティングエラーが発生します。ブラウザに「http://localhost:3000/foo/bar」のように存在しないページを指定してみましょう。



ルーティングエラー

このエラーは、次の場合にも発生します。

- ルーティングの設定を忘れたり、書き間違えたりした。
- HTTPメソッドが合わない。POSTしか受け付けないアクションにGETでアクセスしようとしたときなど。
- app/assetsやpublicの下に存在しないファイルにアクセスしようとした。

このエラー画面を「Not Found」のような自分のエラーページに変える方法は、「11.2　エラーページのカスタマイズ」を参照してください。

■ルーティングの詳細

ここで、ルーティングの書き方を細かく紹介しましょう。まず、Chapter 1で記述したように、トップページを指定するための特殊な書き方があります。rootメソッドに「"コントローラ名#アクション名"」を指定すると、パスが「/」の場合のルーティングを指定できます。TopControllerではなくHomeControllerなら次のように指定します。

```
root "home#index"
```

特定のパスに対応するアクションをGETメソッドで呼び出すには、「get "パス" => "コントローラ名#アクション名"」と記述します。たとえば、「/about」というパスでTopControllerのaboutアクションをGETメソッドで呼び出したいときは、次のようにします。

```
get "about" => "top#about"
```

同様に、「/login」というパスでSessionsControllerのloginアクションをPOSTメソッドで呼び出すなら、次のように記述します。

```
post "login" => "sessions#login"
```

「"info/company" => "info#company"」のように「"コントローラ名/アクション名" => "コントローラ名#アクション名"」というパターンの場合は、次のように簡略化して書けます。

```
get "info/company"
```

ルーティングにはasオプションで名前を付けることができます。次の例では、helpという名前のルーティングを作っています。コントローラやビューでhelp_pathというメソッドを呼び出すと、"/help"という文字列が返ります。

```
get "help" => "documents#help", as: "help"
```

パラメータをパスの中に埋め込みたいときは、:yearのように「コロン＋パラメータ名」を使います。次の例では、「/articles/2018/04」というパスによってArticlesControllerのshowアクションが呼び出され、params[:year]で「2018」、params[:month]で「04」が取り出せるようになります。

```
get "articles/:year/:month" => "articles#show"
```

実際のRailsのアプリケーション開発では、Chapter 5で紹介するリソースベースのルーティングを使うことがほとんどで、ここで紹介したリソースベースではないルーティングは補助的に使うのが普通です。

パスを返すメソッドとシンボル

ルーティングのasオプションを指定すると、about_pathのようなパスを返すメソッドが利用できるようになります。また、「root "top#index"」の指定によって、root_pathメソッドは"/"を返すようになります。このとき、link_toメソッドやredirect_toメソッドには、:aboutや:rootのように、シンボルでパスを指定できます。シンボルは、:aboutのように、パスを返すメソッド名から_pathを除いたものにします。

```
link_to "Home", :root
link_to "このサイトについて", :about
```

link_toメソッドについては、「3.3　テンプレート」の「リンク」を参照してください。

3.2 コントローラとアクション

RailsアプリケーションはMVCアーキテクチャに基づいて3つの部分で構成されていますが、本書では「C→V→M」の順に学習します。まずはC、つまりコントローラの基本的なしくみを見てみましょう。

コントローラの基本

コントローラについてまず知っておくことは、「コントローラの中には複数のアクションが含まれる」ことと、「コントローラの名前の付け方には決まりがある」ことです。

■コントローラクラスの書き方

コントローラは、モデルからデータを受け取り、ビューにレンダリングを行わせます。ブラウザからのリクエストを受け取り、レスポンスを返すのもコントローラです。コントローラは、ApplicationControllerクラスのサブクラスとして実装します。

コントローラクラス

```
class TopController < ApplicationController
  before_action :prepare

  def index                                  アクション
                                  ➡ビュー：index.html.erb
  end

  def about                                  アクション
                                  ➡ビュー：about.html.erb
  end

  private
  def prepare                          プライベートメソッド

  end
end
```

コントローラクラスの例

コントローラクラスのパブリックメソッドを**アクション**と呼びます。アクションは、ウェブサイトの1ページまたは1つの機能に相当します。

アクションで行うことは、テンプレート（ビュー）で表示するデータを用意して、インスタンス変数に値をセットすることです。アクションは、原則として同名のテンプレートを使います。indexアクションを呼び出すと、index.html.erbでレンダリングされます。

また、アクションの目的によっては、HTMLページを表示する代わりに「別のページに移動せよ」という指示（リダイレクション）をブラウザに返すこともできます。

アクションではないメソッド

コントローラクラスにアクションではないメソッドを記述する場合は、プライベートメソッドにする必要があります。そうしないと、意図しないコードが直接実行される可能性が生まれてしまいます。Chapter 8で学習するアクション・コールバックに指定するメソッドはプライベートにしてください。

■命名規約

Chapter 1で紹介したとおり、Railsの原則の1つは「設定より規約」です。モデル、コントローラ、ビューに関連するクラス名やファイル名には、**命名規約**、つまり名前の付け方に決まりごとがあります。コントローラ名を「members」とした場合のコントローラやビューに関する名前は次のようになります。

コントローラの命名規約

名前	例	ルール
コントローラクラス名	MembersController	○○Controller、先頭は大文字
コントローラファイル名	members_controler.rb	○○_controller.rb
テンプレートのディレクトリ名	app/views/members	app/views/○○

Chapter 4から解説するモデルにも命名規約があります。モデル名を「member」とした場合の名前は、次のようになります。

モデルの命名規約

名前	例	ルール
データベーステーブル名	members	先頭は小文字、複数形にする
モデルクラス名	Member	先頭は大文字
モデルクラスのファイル名	member.rb	○○.rb

データベーステーブル名に2つの単語からなる名前を付けたいときは、shopping_cartsのようにアンダースコアで単語と単語を結びます。すると、コントローラクラス名はShoppingCartsController、モデルクラス名はShoppingCartとなります。各単語の頭文字を大文字にする表記法は「キャメルケース」と呼ばれます。

application_controller.rbの使い方

コントローラクラスの親クラスであるApplicationControllerは、app/controllersディレクトリの下のapplication_controller.rbに記述されています。ApplicationControllerクラスにメソッドを加えれば、すべてのコントローラで共通して使える機能を作れます。Chapter 11では、ユーザーの認証や例外処理のためにapplication_controller.rbに共通のメソッドを記述しています。

ApplicationControllerの親クラスは、Railsが用意しているActionController::Baseクラスです。

コントローラクラスの階層

アクションで使える機能

ここではアクションの中で使える基本的な機能を練習していきましょう。

■練習の準備

ここから練習用のコントローラを使ってアクションとテンプレートについて説明します。Chapter 1で作成したasagaoアプリケーションにLessonControllerを追加しましょう。

```
$ bin/rails g controller lesson
```

次に、config/routes.rbを開いて、最後のendの前に次の行を追加してください。「/lesson/step番号」というパスでLessonControllerのstep1からstep18までのアクションを呼び出せるようにしています。また、:nameという名前のパラメータが使えます。uptoメソッドについては、「ブロックの利用」を参照してください。

LIST chapter03/config/routes.rb

```
1 Rails.application.routes.draw do
2   root "top#index"
3   get "about" => "top#about", as: "about"
4
5   1.upto(18) do |n|
```

```
  get "lesson/step#{n}(/:name)" => "lesson#step#{n}"
 end
end
```

app/controllersの下のlesson_controller.rbに「step○○」というアクションを記述して、練習用のコードを書いていきます。

■パラメータの取得

アクションに渡されるパラメータには、2種類あります。1つは、ルーティングの設定によって「http://localhost:3000/lesson/step1/Sato」の「Sato」のようにパスの中に埋め込まれたパラメータです。もう1つは、URLのクエリー部分に「?名前=値」の形を使って「http://localhost:3000/lesson/step1?name=Sato」のように渡されるパラメータです。どちらのパラメータも、「params[:パラメータ名]」で取り出せます。

LessonControllerにアクションstep1を追加して、実験してみましょう。ここで使っている「render plain: "文字列"」は、テンプレートファイルを使わずに、直接文字列を送信する方法です。

ファイルを保存するときは、文字コードをUTF-8にするのを忘れないようにしてください。

LIST chapter03/app/controllers/lesson_controller.rb

```
class LessonController < ApplicationController
 def step1
  render plain: "こんにちは、#{params[:name]}さん"
 end
  （以下省略）
```

「bin/rails s」でサーバーを起動し、ブラウザに「http://localhost:3000/lesson/step1/Sato」と入力すると、次のように表示されます。「http://localhost:3000/lesson/step1?name=Sato」でも同じです。

RESULT

```
こんにちは、Satoさん
```

「?name=%E4%BD%90%E8%97%A4」（?name=佐藤）のようにURLに使えない文字がエンコードされているときでも、Railsが自動的にデコードしてくれるので、漢字やひらがなも取得できます。ただし、文字コードはUTF-8に統一する必要があります。

paramsの機能

params[:パラメータ名]のparamsは、パラメータを含んだハッシュを返すメソッドです。単なるハッシュではなく、Hashを継承したActionController::Parametersというクラスのオブジェクトです。このクラスのハッシュは、文字列でもシンボルでも値を取り出せます。つまり、params[:name]とparam["name"]は同じ値を返します。

paramsが返すパラメータには、コントローラ名とアクション名も含まれています。params[:controller]でコントローラ名、params[:action]でアクション名を取り出せます。step2アクションで試してみましょう。

LIST chapter03/app/controllers/lesson_controller.rb

```
（省略）
 8  def step2
 9    render plain: params[:controller] + "#" + params[:action]
10  end
（以下省略）
```

ブラウザで「http://localhost:3000/lesson/step2」を開くと、次のように表示されます。

RESULT

```
lesson#step2
```

requestオブジェクト

リクエストを送ってきたユーザーの情報を取得するには、requestメソッドが返すオブジェクトを使います。たとえば、ユーザーのIPアドレスはrequest.remote_ipで取り出せます。環境変数を得るにはrequest.envを、リクエストヘッダーを得るにはrequest.headersを使います。たとえば、ブラウザの種類を得るにはrequest.env["HTTP_USER_AGENT"]またはrequest.headers["User-Agent"]とします。

リダイレクション

「3.1　RailsとHTTPの基本」で紹介したHTTPのリダイレクションは、Railsではredirect_toメソッドで簡単に行えます。redirect_toメソッドを呼び出すと、レンダリングは行われずにブラウザにステータスコード302と新しいURLが送られます。

アクションstep3とstep4を作って試してみましょう。redirect_toメソッドの引数は「action: リダイレクト先のアクション名」とします。ブラウザで「http://localhost:3000/lesson/step3」を開くと、新しいURL「http://localhost:3000/lesson/step4」が送られ、ブラウザはstep4に移動します。

LIST chapter03/app/controllers/lesson_controller.rb

```
（省略）
12  def step3
13    redirect_to action: "step4"
14  end
15
16  def step4
17    render plain: "step4に移動しました。"
18  end
（以下省略）
```

RESULT

```
step4に移動しました。
```

redirect_toメソッドの引数には、「redirect_to "/lesson/step4"」のようにパスやURLを文字列で渡すこともできます。

Chapter 6で作成するように、リソースを扱うコントローラでは、create、update、destroyの各アクションでリダイレクションを行うのが定番の方法です。

リダイレクションのステータスコード

HTTP のリダイレクション用のステータスコードには、301（恒久的な移動）、302（発見）、303（他を参照）、307（一時的な移動）があります。HTTP 1.1の仕様では、フォームの送信後のリダイレクションには303を使うことになっていますが、世の中のウェブアプリケーションの多くは伝統的に302を使い続けています。Railsも302を採用しています。

リダイレクションのステータスコードを使い分けたいときは、redirect_toメソッドにstatusオプションを付けます。

```
redirect_to action: "step4", status: 301
```

■フラッシュ

Railsでリダイレクションを行うときは、同時に**フラッシュ**という機能がよく使われます。フラッシュは、アクションとアクションの間で情報を受け渡す機能です。リダイレクションの前でflashオブジェクトに情報を入れておくと、リダイレクション後のアクションでその文字列を取り出すことができます。

アクションstep5とstep6を作って試してみましょう。「http://localhost:3000/lesson/step5」を開くと、flash[:notice]に「step6に移動します。」という文字列が記録されてから、「http://localhost:3000/lesson/step6」に移動し、記録した文字列が表示されます。

LIST chapter03/app/controllers/lesson_controller.rb

```
（省略）
20  def step5
21    flash[:notice] = "step6に移動します。"
22    redirect_to action: "step6"
23  end
24
25  def step6
26    render plain: flash[:notice]
27  end
（以下省略）
```

RESULT

```
step6に移動します。
```

フラッシュのデータは、「アクション→リダイレクション→アクション」の処理が済むと消去されます。上記のstep6アクションのページでリダイレクション後にブラウザをリロードすると、flash[:notice]はnilになります。

フラッシュの名前が:noticeまたは:alertの場合は、redirect_toメソッドの第2引数にハッシュを加えて、リダイレクションとフラッシュの設定を同時に行うこともできます（実例は「6.2　レコードの作成、更新、削除」の「createアクション」を参照）。

```
redirect_to @member, notice: "会員を登録しました。"
```

また、:noticeと:alertの場合は、flash[:notice]やflash[:alert]をflash.notice、flash.alertと書くこともできます。

フラッシュのしくみは、セッション機能を利用して実現されています。そのため、ブラウザのクッキーを無効にしていると利用できません。

flash.now

リダイレクションではなく、1つのアクション内でフラッシュを使うことができます。flash.now[:名前]に文字列を入れると、テンプレートではflash[:名前]でメッセージを取り出せます。

```
def create
  unless @member.save
    flash.now[:error] = "保存に失敗しました。"
    render "new"
  end
end
```

3.3 テンプレート

ここでは、RailsのMVCのうちで「V」（ビュー）にあたるもの、つまりテンプレート（erbファイル）の機能を紹介します。テンプレートの使い方を覚えれば、効率よく短時間でページ作成ができるようになります。

テンプレートの基本

Railsのページデザインでは、コントローラのアクションでインスタンス変数を用意し、テンプレートにその変数を埋め込むのが基本的な流れとなります。

■テンプレートの書式

Railsの**テンプレート**（erbファイル）は、HTML文書の中にRubyコードを埋め込んだものです。<% %>または<%= %>で囲んだ部分は、Rubyのコードとして解釈されます。このRubyコードによってHTML文書が動的に書き換えられてブラウザに送られます。

<%= %>の間にRubyの式（リテラル、変数、メソッド呼び出しなど）を記述すると、式の結果が文字列に変換されてその場所に挿入されます。一方、<% %>の間に記述されたRubyのコードは評価されますが、文字列の挿入は行われません。

次の例では、1行目の<% ～ %>は計算を行って変数priceに値を代入するだけです。2行目の<%= ～ %>では、変数priceの値がHTML文書の中に埋め込まれます。

テンプレート

```
<p><% price = (2000 * 1.08).floor %></p>
<p><%= price %>円</p>
```

ブラウザへのレスポンス

```
<p></p>
<p>2160円</p>
```

テンプレートの中にはどのようなRubyコードを書いてもかまわないのですが、あまり複雑な処理を記述するのは避けるべきです。「コントローラのアクションで用意したデータをインスタンス変数に記録し、そのインスタンス変数をテンプレートに埋め込む」という基本線から大きく外れないように心がけてください。

to_sメソッドで出力される

<%= 式 %>が出力されるときは、その中の式に対してto_sメソッドが呼ばれ、その結果の文字列が出力されます。「<%= 式.to_s %>が表示される」と考えてください。Rubyのオブジェクトはすべてto_sメソッドを備えています。

下記のstep8の例で@priceが存在しないときは、 円 という結果になります。存在しないインスタンス変数はnilを返し、nil.to_sは空文字列""を返すからです。

前節3.2で作ったLessonControllerにstep7アクションを作成し、作成した変数をテンプレートに表示させてみましょう。step6までの練習と違い、普通にテンプレートを使うときはrenderメソッドは不要です。

LIST chapter03/app/controllers/lesson_controller.rb

```
（省略）
27  def step7
28    @price = (2000 * 1.08).floor
29  end
（以下省略）
```

テンプレートのファイル名は、「アクション名.html.erb」です。ディレクトリapp/views/lessonの下にstep7.html.erbという名前のファイルを作り、次のように記述してください。

LIST chapter03/app/views/lesson/step7.html.erb

```
1 <p><%= @price %>円</p>
```

ブラウザで「http://localhost:3000/lesson/step7」を開くと、結果が表示されます。

RESULT

```
2160円
```

テンプレートファイルがなかったり、ファイル名を間違えたりしたときは、「Template is missing」というエラーが表示されます。ただし、アクションの中でrenderメソッドを使用せず、デフォルトのテンプレートファイルが存在しない場合は、例外UnknownFormatが発生します。

コントローラとテンプレートのオブジェクトは別

テンプレートをレンダリングする際には、「ActionView::Baseクラスを継承し、ヘルパーのモジュールをミックスインしたクラス」が自動的に作成されて、そのインスタンスのもとでコードが実行されます。そのため、テンプレート内ではAction Viewとヘルパーのメソッドが使えます。後述のnumber_with_delimiterやlink_toといったメソッドは、Action Viewのメソッドです。

paramsやflashのようにどちらでも使えるメソッドがあるのでややこしいのですが、コントローラとテンプレートは、別のオブジェクトであることに注意してください。Rubyにはオブジェクト間でインスタンス変数を共有する機能があり、Railsはこの機能を利用してアクションのインスタンス変数をテンプレートに渡しています。

メソッドがなくてテンプレートがある場合

コントローラクラスの中にアクションのメソッドがなくても、app/viewsディレクトリに「アクション名.html.erb」というファイルがあれば、コントローラはそのファイルをレンダリングします。アクションの中ですることが何もなければ、そのメソッドは省略できます。ただし、筆者は空のメソッドでもコントローラの中に書いておくほうがわかりやすいと思います。

アクションもテンプレートファイルもない場合は、「Unknown action」というエラーが表示されます。

■renderメソッド

アクションの実行が終わると、Railsは自動的にアクションと同名のテンプレートを使ってHTMLを生成（レンダリング）します。アクションの中でrenderメソッドを呼び出すと、別のテンプレートを使ってレンダリングを行うこともできます。

別のアクション用のテンプレートを共有するには、引数にアクション名を指定します。次の例では、アクションstep8はテンプレートstep7.html.erbでレンダリングします。

LIST chapter03/app/controllers/lesson_controller.rb

```
（省略）
31  def step8
32    @price = 1000
33    render "step7"
34  end
（以下省略）
```

RESULT

```
1000円
```

なお、コントローラにstep7というアクションがなくても、app/views/lessonディレクトリにstep7.html.erbがあればレンダリングできます。

app/views/lessonディレクトリ以外にあるテンプレートを使いたいときは、「another/show」のようにapp/viewsディレクトリを基点としたパス名を指定します。.html.erbは省略できます。

```
def show
  render "another/show"
end
```

二度レンダリング・リダイレクトはできない

レンダリングは1つのアクションにつき一度だけと決められています。また、1つのアクションの中では、renderメソッドとredirect_toメソッドはどちらかを一度だけしか使えません。renderメソッドを二度呼び出したり、renderメソッドとredirect_toメソッドを両方呼び出したりすると、例外DoubleRenderErrorが発生します。

■HTML特殊文字の変換

HTMLでは、<、>、&はタグなどを表す特別な記号です。Railsのテンプレートでこれらの文字をそのまま埋め込むと、HTML文書の構造が変わってしまうかもしれません。悪意のあるユーザーは、ウェブアプリケーションを攻撃するために、わざとこうした文字を送信してページの表示を作り変えることがあります。

これを防ぐために、HTMLの特殊文字を表示するときは<、>、&に変換する必要があります。<%= %>は自動的にこの変換を行います。

次のインスタンス変数@commentは、危険なJavaScriptのコードを含んでいるとします。

LIST chapter03/app/controllers/lesson_controller.rb

```
（省略）
36  def step9
37    @comment = "<script>alert('危険！')</script>こんにちは。"
38  end
（以下省略）
```

LIST chapter03/app/views/lesson/step9.html.erb

```
1 <p><%= @comment %></p>
```

「http://localhost:3000/lesson/step9」を開いてHTMLのソースを表示すると、<と>は、<と>に変換されています。

RESULT （HTMLのソース）

```
<p>&lt;script&gt;alert(&#39;危険！&#39;)&lt;/script&gt;こんにちは。</p>
```

逆に、HTMLのタグをそのまま出力したい場合でも、特殊文字がいちいち変換されてしまいます。タグをタグとして出力するには、<%= %>の代わりに<%== %>を使います。

LIST chapter03/app/controllers/lesson_controller.rb

```
（省略）
40  def step10
41    @comment = "<strong>安全なHTML</strong>"
42  end
（以下省略）
```

LIST chapter03/app/views/lesson/step10.html.erb

```
1 <p><%== @comment %></p>
```

RESULT （HTMLのソース）

```
<p><strong>安全なHTML</strong></p>
```

HTML特例文字の変換を抑えるには次のようにhtml_safeメソッドを用いることもできます。

```
<p><%= @comment.html_safe %>
```

このメソッドは文字列に「HTML文書に埋め込んでも安全である」という印を付けます。Action Viewが用意しているlink_toメソッドやimage_tagメソッドのようにタグを生成するメソッドは、「.html_safe付きの文字列を返す」と考えてください。

link_toの引数での特殊文字

link_toメソッドのようにタグを生成するメソッドの引数でも、特殊文字の変換が行われます。Railsでは、細かいことを考えずに<%= %>を書き、Action Viewのメソッドを使っていれば、安全なHTMLを生成できるようになっています。

引数にタグをわざと埋め込みたいときは、html_safeメソッドを使います。

```
link_to "<b>Top</b>", "/"
結果：<a href="/">&lt;b&gt;Top&lt;/b&gt;</a>
link_to "<b>Top</b>".html_safe, "/"
結果：<a href="/"><b>Top</b></a>
```

書式の指定とヘルパーメソッド

Railsのビューの機能をいろいろと試してみましょう。

■数値、日付、文字列

小数点以下の桁数など文字列や数字の書式を揃えたいときは、Rubyのsprintfメソッドが使えます。sprintfは、第1引数に書式を指定し、第2引数以降に書式に埋め込む変数を並べます。次の例は、人口、面積（小数点以下切り捨て）、人口密度（小数点以下2桁まで）を表示するものです。

LIST chapter03/app/controllers/lesson_controller.rb

```
（省略）
44  def step11
45    @population = 704414
46    @surface = 141.31
47  end
（以下省略）
```

LIST chapter03/app/views/lesson/step11.html.erb

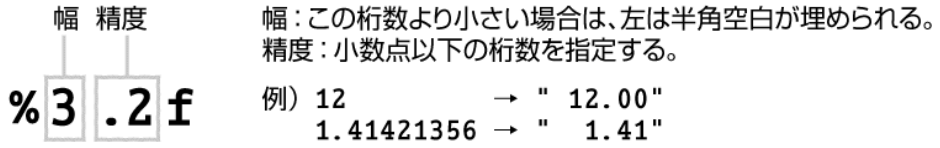
```
1 <p>
2 <%= sprintf("人口%d人、面積%.0f平方キロ、人口密度%.2f人/平方キロ",
3           @population, @surface, @population/@surface) %></p>
```

RESULT

```
人口704414人、面積141平方キロ、人口密度4984.88人/平方キロ
```

sprintfの書式の中では、「%文字」の部分に第2引数以降に指定した引数が埋め込まれます。よく使われるのは、%d（整数）、%f（浮動小数点数）、%s（文字列）です。%と文字の間には、幅と精度

（どちらも省略可）を指定できます。



幅と精度

日時や日付の書式は、日時オブジェクトや日付オブジェクトのメソッドstrftimeで揃えられます。次の例は、strftimeで年月日時分秒を表示するものです。Timeクラスについては、「2.3　いろいろなオブジェクト」の「日時と日付」を参照してください。

LIST chapter03/app/controllers/lesson_controller.rb

```
（省略）
49  def step12
50    @time = Time.current
51  end
（以下省略）
```

LIST chapter03/app/views/lesson/step12.html.erb

```
1 <p><%= @time.strftime("%Y/%m/%d(%a) %H:%M:%S") %></p>
```

RESULT

```
2018/01/01(Mon) 17:37:31
```

Rubyのメソッドだけでなく、RailsのAction Viewが備えているメソッドも利用できます。たとえば、3桁ごとにカンマを入れて数値を表示させたいときは、number_with_delimiterメソッドを使います。

LIST chapter03/app/controllers/lesson_controller.rb

```
（省略）
53  def step13
54    @population = 127767944
55  end
（以下省略）
```

LIST chapter03/app/views/lesson/step13.html.erb

```
1 <p>人口<%= number_with_delimiter(@population) %>人</p>
```

RESULT

```
人口127,767,944人
```

また、Action Viewにはsimple_formatやtruncateなどの文字列を整えるヘルパーメソッドもあります。これらについては、Morning Gloryサイトの開発で実際に使う際に説明します。

■ヘルパーメソッドの作成

テンプレート内で使われるヘルパーメソッドを自分で書くこともできます。例として、改行をHTMLのbrタグに変換するtiny_formatメソッドを定義してみましょう。

app/helpersディレクトリの下にlesson_helper.rbというファイルを新規作成して、次のような内容を書き入れてください。

LIST chapter03/app/helpers/lesson_helper.rb

```
1 module LessonHelper
2   def tiny_format(text)
3     h(text).gsub("¥n", "<br />").html_safe
4   end
5 end
```

hメソッドは「<」→「<」のようにHTML特殊文字を変換します。そして、gsubメソッドで改行文字（"¥n"）を
に一括置換します。最後にhtml_safeメソッドで
をそのまま出力します。

app/helpersディレクトリの下で定義されたモジュールを**ヘルパーモジュール**と言います。ヘルパーモジュールはすべてのテンプレートにミックスインされます。

tiny_formatメソッドを使ってみましょう。

LIST chapter03/app/controllers/lesson_controller.rb

```
（省略）
57   def step14
58     @message = "ごきげんいかが？¥nRailsの勉強をがんばりましょう。"
59   end
（以下省略）
```

テンプレート内では、Action Viewのメソッドと同じようにヘルパーメソッドを呼び出せます。

LIST chapter03/app/views/lesson/step14.html.erb

```
1 <p><%= tiny_format(@message) %></p>
```

RESULT

ごきげんいかが？
Railsの勉強をがんばりましょう。

ヘルパーモジュールとコントローラの関係

ヘルパーメソッドの数が増えてきたら、ヘルパーモジュールをいくつも作ってメソッドを分けるとよいでしょう。ヘルパーモジュールには「○○Helper」という形式の名前を付け、ファイルには「○○_helper.rb」のような対応する名前を付けてください。

たとえば、LessonController用のテンプレートだけで使用するヘルパーメソッドをヘルパーモジュールLessonHelperで定義するのは、よい考えです。ただし、ヘルパーモジュールはすべて、どのテンプレートにもミックスインされます。つまり、tiny_formatメソッドはテンプレートapp/views/top/index.html.erbの中でも使用できます。ヘルパーモジュールとコントローラの間には特別な関係はありません。このため、ヘルパーモジュールの間でメソッド名が重複しないようにしてください。

リンクと画像

RailsのAction Viewには、リンクや画像のためのタグを簡単に生成するヘルパーメソッドも用意されています。

■リンク

テンプレート内でリンク用のタグ（HTMLのaタグ）を作るには、link_toメソッドを使います。link_toメソッドの第1引数にはリンクのテキスト、第2引数にはパスを指定します。

次の例は、トップページへのリンクを作成するものです。root_pathは「/」を返すメソッドで、:rootと指定することもできます（「3.1　RailsとHTTPの基本」のHINT「パスを返すメソッドとシンボル」を参照）。

LIST chapter03/app/views/lesson/step15.html.erb

```
1 <p><%= link_to "Home", root_path %></p>
```

Home

RESULT

link_toメソッドの第2引数には、次のようにパスを指定できます。こうしたパスの指定方法は、コントローラのredirect_toメソッドの引数にも使えます。「5.1　RESTとルーティング」の「オブジェクトでパスを表す」も参照してください。

- "http://www.oiax.jp/"や"/help"のようなURLやパスを表す文字列。
- root_pathのようにルーティングの設定で使えるようになるパス。および、members_pathのようにリソースを表すパス。
- @memberのようにリソースを表すモデルオブジェクトや配列。
- :rootや:membersのようにパスを表すシンボル。
- コントローラ、アクション、パラメータを表すハッシュ。たとえば、LessonControllerのstep1アクションのパスを作るには、「link_to "Step1", controller: "lesson", action: "step1", name: "Sato"」のようにします。Railsに昔からある書き方ですが、本書では使いません。

link_toメソッドの第3引数には、ハッシュでmethodオプションやdataオプションを追加できます。methodオプションはHTTPメソッドの種類を指定します。dataオプションはリンク先に進むかどうかを示す確認メッセージを表示する際に利用します。「6.2　レコードの作成、更新、削除」の「会員の削除」を参照してください。

```
<%= link_to "削除", member, method: :delete,
    data: { confirm: "本当に削除しますか?" } %>
```

また、「属性名: 値」を追加すれば、aタグの属性になります。次の例は、<a href="/" class="menu">のようなタグになります。

```
<%= link_to "Home", root_path, class: "menu" %>
```

現在のページだったらリンクにしない

link_toメソッドの代わりにlink_to_unless_currentメソッドも使えます。このメソッドを使うと、「指定のパスが現在のページのものだったらリンクの代わりにテキストだけ表示する」ということができます。メニュー用のリンクを作るときに使うと便利です。

```
<p><%= link_to_unless_current "Home", root_path %></p>
```

表示するテキストにタグを加えたいときは、link_to_unless_currentメソッドにブロックを渡します。

```
<%= link_to_unless_current("Home", root_path) do %>
  <span class="current">Home</span>
<% end %>
```

■画像

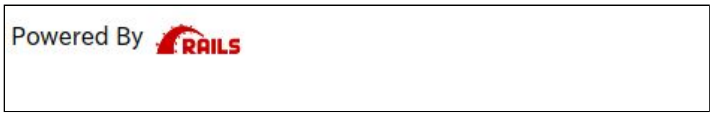
リンクと同様に、画像用のタグ（HTMLのimgタグ）を作成するメソッドも用意されています。image_tagメソッドに画像のファイル名とオプションを指定すると、自動的にタグができます。

Ruby on Railsのロゴ画像を表示してみましょう。まず、ブラウザで「http://rubyonrails.org/images/rails-logo.svg」を開いて、rails-logo.svgをダウンロードし、app/assets/imagesディレクトリに保存します。そして、次のような HTMLテンプレートを作成してください。

LIST chapter03/app/views/lesson/step16.html.erb

```
1 <p>Powered By
2 <%= image_tag("rails-logo.svg", size: "64x20",
3       alt: "Ruby on Rails", align: "top") %></p>
```

image_tagメソッドの第1引数には画像ファイル名、第2引数にはハッシュでオプションを指定します。オプションはHTMLのimgタグに指定する属性と同じものが指定できます。縦と横の幅は「size: "64x20"」のように指定できます。



RESULT

画像ファイルの標準的な置き場所は、app/assets/imagesディレクトリです。ただし、image_tag("/images/rails.png")のようにパスを/で始めれば、publicディレクトリにある画像が使われます。詳しくは「12.2　アセット・パイプライン」のHINT「publicディレクトリの下に置く場合」を参照してください。

画像にリンクを張りたいときは、link_toメソッドとimage_tagメソッドを組み合わせます。

```
<p>Powered By
<%= link_to(image_tag("rails.png", size: "64x20",
                alt: "Ruby on Rails", align: "top" ),
    "http://rubyonrails.org/") %></p>
```

■一般的なタグの出力

HTMLのタグは、通常はRubyのコードとは別にして<%= %>の外に置きます。Rubyのコードを使ってタグを記述する必要があるときは、tagメソッドやcontent_tagメソッドを使います。次の例は、
と こんにちは に変換されます。content_tagメソッドは、ブロックの中に内容を記述できます。

```
<%= tag(:br) %>
<%= content_tag(:p, class: "p1") do %>
こんにちは
<% end %>
```

条件分岐と繰り返し

条件分岐や繰り返しの構文を利用すると、テンプレートを効率よく簡潔に記述できます。

■条件分岐

テンプレートでは、「<% if 条件式 %> 〜 <% else %> 〜 <% end %>」でテキストやHTMLのタグを囲むと、条件式の結果によって表示を切り替えることができます。間には「<% elsif 条件式 %>」をはさむこともできます。

次の例では、変数@zaikoが0以上なら在庫数を表示し、そうでなければ「品切れです。」と表示します。

LIST chapter03/app/controllers/lesson_controller.rb

```
    （省略）
61  def step17
62    @zaiko = 10
63  end
    （以下省略）
```

LIST chapter03/app/views/lesson/step17.html.erb

```
1 <% if @zaiko > 0 %>
2 残り<%= @zaiko %>個です。
3 <% else %>
4 品切れです。
5 <% end %>
```

RESULT

```
残り10個です。
```

if式だけでなく、unless式も使えます。

```
<% unless @zaiko == 0 %>
残り<%= @zaiko %>個です。
<% else %>
品切れです。
<% end %>
```

余分な改行やスペースの除去

<% -%>や<%= -%>のように-付きで閉じると、-%>の後ろの改行は取り除かれます。また、<%- %>のように-を付けて始めると、行頭から<%-までの空白が取り除かれます。

テンプレート

```
<p>価格は、<%= @price -%>
円です。</p>
```

ブラウザへのレスポンス

```
<p>価格は、2100円です。</p>
```

■繰り返し

整数のtimesメソッドや、配列やハッシュのeachメソッドなどのブロックを<% %>で記述すると、表示を繰り返すことができます。繰り返し表示したいテキストやタグは、ブロック内に記述します。たとえば次のようにすると「ランランラン」が表示されます。

```
<% 3.times do %>
ラン
<% end %>
```

ブロックを使うと、リスト（ulタグ）やテーブル（tableタグ）に配列やハッシュのデータを表示するときに効率のよい記述ができます。次の例は、ハッシュのeachメソッドを使ってテーブルの行を繰り返し表示するものです。

LIST chapter03/app/controllers/lesson_controller.rb

```
  （省略）
65   def step18
66     @items = { "フライパン" => 2680, "ワイングラス" => 2550,
67                "ペッパーミル" => 4515, "ピーラー" => 945 }
68   end
69 end
```

LIST chapter03/app/views/lesson/step18.html.erb

```
1 <table border="1" cellpadding="4">
2 <% @items.each do |key, val| %>
3 <tr>
4   <th><%= key %></th>
5   <td style="text-align: right"><%= number_with_delimiter(val) %>円</td>
6 </tr>
7 <% end %>
8 </table>
```

フライパン	2,680円
ワイングラス	2,550円
ペッパーミル	4,515円
ピーラー	945円

RESULT

3.4 モックアップの作成

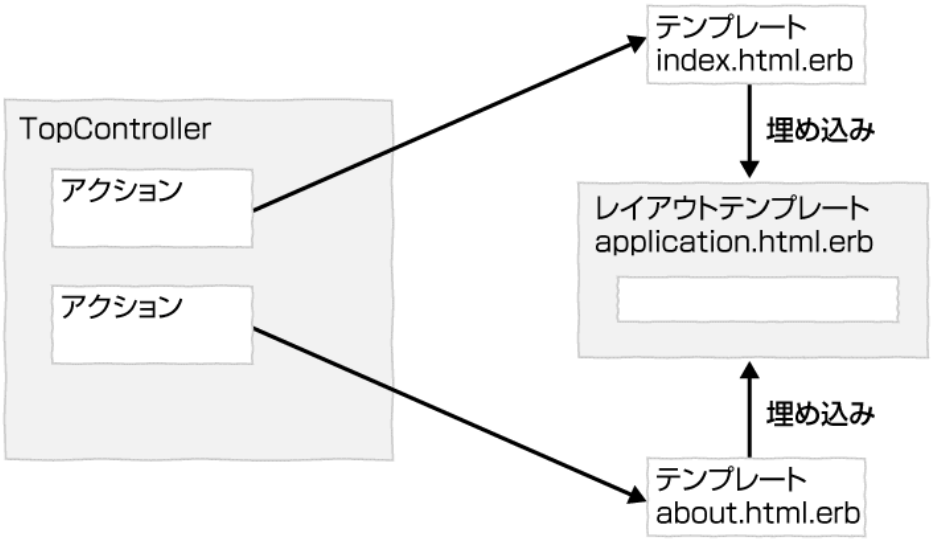
ここからは、今まで学んだことを応用して、本書のサンプルサイトMorning Glory（asagaoアプリケーション）のモックアップ（実装前の見本サイト）を作成します。Chapter 4からは、このモックアップにモデル（データベース）の機能を加えていきます。

レイアウトテンプレート

ウェブサイトでは、ページごとのデザインを統一する必要があります。一般的には、ページ全体を囲む「枠」の中に、ページごとのコンテンツを入れるという形を取ります。Railsではこの「枠」をレイアウトテンプレートで簡単に作成できます。

■レイアウトテンプレートとは

全体の枠となる**レイアウトテンプレート**は、app/views/layoutsディレクトリの下に置いたテンプレートファイルに記述します。Railsはレンダリングを行う際に、各アクション用のテンプレートをレンダリングし、それをレイアウトテンプレートの中に埋め込んで、HTML全体を出力します。



レイアウトテンプレート

「bin/rails new」コマンドでアプリケーションを作成すると、app/views/layoutsディレクトリの下にapplication.html.erbというデフォルトのレイアウトテンプレートができます。<%= yield %>の部分がアクション用のテンプレート（つまりページごとのコンテンツ）が埋め込まれる場所です。

デフォルトのテンプレート

```
<!DOCTYPE html>
<html>
  <head>
    <title>Asagao</title>
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-track': 'reload'
%>
    <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>

  <body>
    <%= yield %>
  </body>
</html>
```

Chapter 1で作ったトップページ（TopControllerのindexアクション）をブラウザで開き、HTMLソースを確認してみましょう。

トップページのHTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Asagao</title>
    <meta name="csrf-param" content="authenticity_token" />
<meta name="csrf-token"
content="dN4ovClBx9BiTgsqVQkLLont1tf4VYv4F0IboO3JwbP8XR38t7QDDWsE+Btg9x7
W4cpmaSa8Ec83ON3y//1H7g==" />
```

```
    <link rel="stylesheet" media="all" href="/assets/application.self-
f0d704deea029cf000697e2c0181ec173a1b474645466ed843eb5ee7bb215794.css?
body=1" data-turbolinks-track="reload" />
    <script src="/assets/rails-ujs.self-
551fbd47b981dacbb84a270f9123074caf39eb72aaf6f478ab597c6f81435e4b.js?body=1"
data-turbolinks-track="reload"></script>
<script src="/assets/activestorage.self-
6f0d773d8a366fac20308619a437ca72decef5467e2d9f7a3019afd7bb2ee72e.js?body=1"
data-turbolinks-track="reload"></script>
<script src="/assets/turbolinks.self-
2db6ec539b9190f75e1d477b305df53d12904d5cafdd47c7ffd91ba25cbec128.js?body=1"
data-turbolinks-track="reload"></script>
<script src="/assets/application.self-
66347cf0a4cb1f26f76868b4697a9eee457c8c3a6da80c6fdd76ff77e911715e.js?body=1"
data-turbolinks-track="reload"></script>
  </head>

  <body>
    <h1>おはようございます！</h1>
<p>これからRailsの勉強を始めます。</p>

  </body>
</html>
```

<%= yield %>の部分にindex.html.erbの内容が埋め込まれるほかに、<head> 〜 </head>の間にcsrf_meta_tags、csp_meta_tag、stylesheet_link_tagとjavascript_include_tagの各メソッドが作ったタグが埋め込まれます。

こうしたメソッドとタグについては、「6.2　レコードの作成、更新、削除」のHINT「csrf_meta_tagsメソッド」、および「12.2　アセット・パイプライン」を参照してください。

csp_meta_tagメソッド

csp_meta_tagメソッドは、Rails 5.2で導入された新しいメソッドで、Content Security Policy（CSP）に関するタグを埋め込むために使用します。初期状態のRailsアプリケーションはCSPを利用しないため、このメソッドは何の効果も持ちません。本書ではCSPに関する説明を省略します。

モックアップのレイアウトテンプレート

始めにapplication.html.erbの頭の部分を次のように書き換えましょう。

LIST chapter03/app/views/layouts/application.html.erb

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title><%= page_title %></title>
5     <%= csrf_meta_tags %>
6     <%= csp_meta_tag %>
7
8     <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-track': 'reload'
%>
9     <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reload' %>
10   </head>
    （以下省略）
```

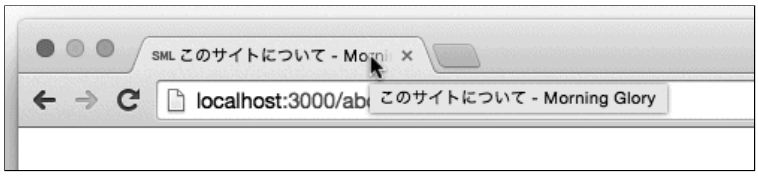
HTMLのtitleタグの中身は、ヘルパーメソッドのpage_titleで作ることにします。app/helpers/application_helper.rbのApplicationHelperモジュールに次のようにpage_titleメソッドを追加します。インスタンス変数@page_titleがあればページのタイトルは「○○ - Morning Glory」に、なければ「Morning Glory」だけとなります。

LIST chapter03/app/helpers/application_helper.rb

```
1 module ApplicationHelper
2   def page_title
3     title = "Morning Glory"
4     title = @page_title + " - " + title if @page_title
5     title
```

```
  end
end
```

TopControllerのaboutアクションのテンプレートを作ったときに、変数@page_titleに文字列「このサイトについて」を入れました（「3.1　RailsとHTTPの基本」の「ルーティング」を参照）。「http://localhost:3000/about」をブラウザで開くとタイトルが次のように変わります。



RESULT

Railsはアクション用のテンプレートや部分テンプレートをレンダリングしたあとで、レイアウトテンプレートをレンダリングします。about.html.erb内で設定した変数@page_titleが、application.html.erb内のpage_titleメソッドで使えるのはそのためです。

■レイアウトテンプレートの切り替え

コントローラやアクションごとにレイアウトテンプレートを切り替えて、1つのサイトで複数のデザインを使い分けることもできます。レイアウトテンプレートを指定する方法は、3つあります。1つ目は、app/views/layoutsディレクトリにtop.html.erbのようにファイル名が「コントローラ名.html.erb」のテンプレートを置くことです。それがTopControllerのレイアウトテンプレートになります。

2つ目は、コントローラでlayoutメソッドを使うことです。引数は、layoutsディレクトリに置いたテンプレートファイル名から拡張子を除いたものにします（引数はシンボルではなく文字列にしてください）。

```
class TopController < ApplicationController
  layout "simple"
```

3つ目は、アクション内でrenderメソッドにlayoutオプションを付けてテンプレートファイル名を指定することです。layoutオプションを使えば、ある条件の場合にだけレイアウトを切り替えることができます。

```
  def show
    if 何かエラーが発生...
      render layout: "error"
```

```
  end
end
```

なお、レイアウトテンプレートを使わずに、アクション用のテンプレートの中身をそのままHTML全体とするには、「render layout: false」とします。

レイアウトの継承

ApplicationControllerにlayoutメソッドを記述すると、すべてのコントローラにレイアウトが継承されます。たとえば、次のようにApplicationControllerでレイアウトを指定したとします。

```
class ApplicationController < ActionController::Base
  layout "application"
end
```

そして、次のようにTopControllerとHelpControllerを定義したとします。

```
class TopController < ApplicationController

end

class HelpController < ApplicationController
  layout "simple"
end
```

この場合、TopControllerではapp/views/layouts/application.html.erbが、HelpControllerではapp/views/layouts/simple.html.erbがレイアウトとして使われます。

部分テンプレート

サイトの中ではページ全体の枠だけでなく、各ページに入れる「パーツ」も共通にする必要が出てきます。たとえば、ページの上部に置くメニューバーや、ページの左右に置くサイドバーなどです。こうした共通パーツは**部分テンプレート**で作ります。

■部分テンプレートの使い方

部分テンプレート用のファイルは_menu_bar.html.erbのようにファイル名の前に_（アンダースコア）を付けます。ファイルを置くディレクトリはアクション用テンプレートと同じです。たとえば次のような部分テンプレートを書いたとします。

部分テンプレートの例

```
<nav>
  <ul>
    <li><%= link_to 'TOP', :root %></li>
    <li><%= link_to 'ニュース', :articles %></li>
  </ul>
</nav>
```

この部分テンプレートを別のテンプレートに埋め込むには、renderメソッドを使います。引数には、ファイル名から先頭の_と拡張子を除いたものになります。

部分テンプレートの埋め込み

```
<%= render "menu_bar" %>
```

部分テンプレートは、レイアウトテンプレートに埋め込んだり、複数のコントローラのテンプレートで共通に使ったりすることもあります。本書では、共通の部分テンプレートは、app/viewsディレクトリの下にsharedというディレクトリを作成して、そこに置くことにします。別のディレクトリにある部分テンプレートを埋め込むには、renderメソッドの引数を「ディレクトリ名/テンプレート名」とします。

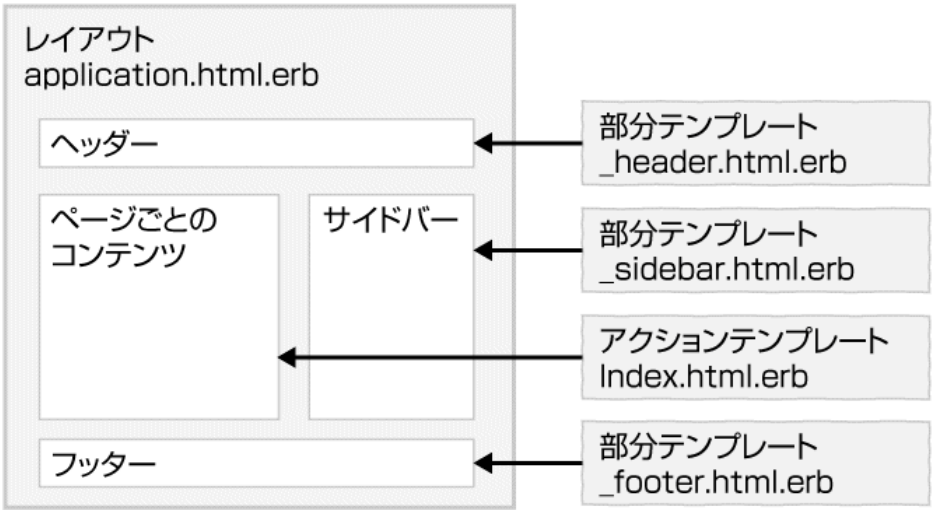
別のディレクトリにある部分テンプレートの埋め込み

```
<%= render "shared/menu_bar" %>
```

引数の中では_は付けませんが、ファイル名は「_menu_bar.html.erb」のように_で始めている点に注意してください。

■モックアップの部分テンプレート

サイトMorning Gloryでは、次のようなデザインでページを構成することにします。部分テンプレートを使ってヘッダー、サイドバー、フッターを作ります。



RESULT

まず、レイアウトテンプレートの<body> 〜 </body>の部分を次のように変更します。

LIST chapter03/app/views/layouts/application.html.erb

```
（省略）
13  <body>
14    <div id="container">
15      <header>
16        <%= render "shared/header" %>
17      </header>
18      <main>
19        <%= yield %>
20      </main>
21      <aside id="sidebar">
22        <%= render "shared/sidebar" %>
23      </aside>
24      <footer>
25        <%= render "shared/footer" %>
26      </footer>
27    </div>
28  </body>
29 </html>
```

ヘッダー（header要素）、コンテンツ部分（main要素）、サイドバー（aside要素）、フッター（footer要素）の4つの要素で構成し、その内容を部分テンプレートにします。さらに、全体をid="container"のdivタグで囲みます。

次に、app/viewsディレクトリの下にsharedディレクトリを作成し、その中に部分テンプレート_header.html.erbを作成します。

LIST chapter03/app/views/shared/_header.html.erb

```
<%= image_tag "logo.gif", size: "272x48", alt: "Morning Glory" %>

<nav class="menubar">
  <ul>
    <%= menu_link_to "TOP", :root %>
    <%= menu_link_to "ニュース", "#" %>
    <%= menu_link_to "ブログ", "#" %>
    <%= menu_link_to "会員名簿", "#" %>
    <%= menu_link_to "管理ページ", "#" %>
  </ul>
</nav>
```

テンプレートの内容は、ロゴ画像とメニューとなるリンクです。まだ作っていないページは「#」をリンク先にします。

ロゴ画像logo.gifは、ダウンロードしたサンプルソースのapp/assets/imagesからコピーして、自分のapp/assets/imagesの下に置いてください。

続いて、メニューの中で使うヘルパーメソッドmenu_link_toをapp/helpersの下のapplication_helper.rbに記述します。

LIST chapter03/app/helpers/application_helper.rb

```
module ApplicationHelper
  （省略）
  def menu_link_to(text, path, options = {})
    content_tag :li do
      link_to_unless_current(text, path, options) do
        content_tag(:span, text)
      end
```

```
    end
  end
end
```

メソッドmenu_link_toの第3引数でoptions = {}という書き方をしている点に注意してください。これは、第3引数が省略可能であり、もし省略された場合は等号（=）の右側の値がデフォルト値として使われることを意味しています。つまり、menu_link_to("HOME", :root)のように呼び出した場合、仮引数optionsには空のハッシュがセットされます。

link_to_unless_currentメソッドを使って、現在のページの場合はリンクにせずに、spanタグでテキストを囲むようにします。liタグやspanタグを生成するのに、content_tagメソッドを利用しています。

さらに、サイドバーの部分テンプレート_sidebar.html.erbを次のように記述します。サイドバーには、ニュースと会員のブログ記事へのリンクを並べることにします。

LIST chapter03/app/views/shared/_sidebar.html.erb

```
<%= render "shared/login_form" %>

<h2>最新ニュース</h2>
<ul>
  <% 5.times do |i| %>
    <li><%= link_to "ニュースの見出し", "#" %></li>
  <% end %>
</ul>

<h2>会員のブログ</h2>
<ul>
  <% 5.times do |i| %>
    <li><%= link_to "ブログの見出し", "#" %></li>
  <% end %>
</ul>
```

また、サイドバーの上部にはログインフォームを置きます。ログインフォームは部分テンプレート_login_form.html.erbに分けて、部分テンプレートの中に部分テンプレートを埋め込みます。実際に会員がログインする機能はChapter 8で作成します。

LIST chapter03/app/views/shared/_login_form.html.erb

```
<h2>ログイン</h2>
<form id="login_form">
  <div>
    <label>ユーザー名：</label>
    <input type="text">
  </div>
  <div>
    <label>パスワード：</label>
    <input type="password">
  </div>
  <div>
    <input type="submit" value="ログイン">
  </div>
</form>
```

フッターとなる部分テンプレート_footer.html.erbを以下のように記述し、「このサイトについて」のページへのリンクを張ります。:aboutの代わりにabout_pathと書いてもかまいません。

LIST chapter03/app/views/shared/_footer.html.erb

```
<%= link_to "このサイトについて", :about %> |
Copyright (C) <%= link_to "Oiax Inc.", "http://www.oiax.co.jp/" %>
2007-2018
```

■トップページの修正

トップページ（TopControllerのindexアクション）の内容もモックアップにふさわしい内容にしておきましょう。ニュースの内容に変わるダミーのテキストを入れておきます。Chapter 1で作ったテンプレート全体を次の内容で置き換えます。

LIST chapter03/app/views/top/index.html.erb

```
<% 5.times do |x| %>
  <h2>見出し</h2>
  <p>
    ここに本文が入ります。ここに本文が入ります。ここに本文が入ります。ここに本文が入ります。
```

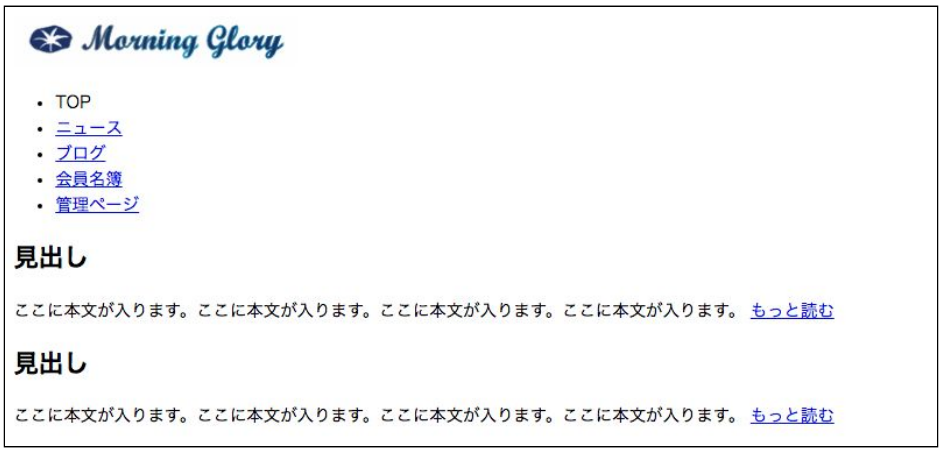
```
5    <%= link_to "もっと読む", "#" %>
6  </p>
7 <% end %>
```

TopControllerのindexアクションに書いた内容は消しておきます。

LIST chapter03/app/controllers/top_controller.rb

```
1 class TopController < ApplicationController
2   def index
3   end
  （以下省略）
```

修正後のトップページをブラウザで表示すると、次のようになります。まだスタイルシートを作っていないので、地味な表示です。



RESULT

スタイルシート

各ページで共通して使う「枠」や「パーツ」などのHTMLは、レイアウトテンプレートや部分テンプレートに記述します。一方、背景色や文字色、フォントサイズ、余白、枠線などのデザインは、**スタイルシート（CSS）**に記述します。

CSSファイルは、app/assets/stylesheetsの下に置きます。このディレクトリには初期状態でapplication.cssというファイルがあり、次のような内容です。

LIST chapter03/app/assets/stylesheets/application.css

```
 1 /*
 2  * This is a manifest file that'll be compiled into application.css, ...
   （省略）
12  *
13  *= require_self
14  *= require_tree .
15  */
```

コメントの中の*=で始まる行は必要ですので、消したり書き換えたりしないでください。この行の役割については、「12.2　アセット・パイプライン」を参照してください。

このファイルの16行目以降にスタイルを記述すれば、ページに反映されます。以下に示すCSSファイルの内容は一部省略しています。サンプルソースのapp/assets/stylesheetsからapplication.cssをコピーして、自分のapp/assets/stylesheetsの下に置いてください。

まず、全体に共通する要素のデザインを記述します。

LIST chapter03/app/assets/stylesheets/application.css

```
   （省略）
17 /* ページ全体 */
18 body {
19   background-color: white;
20   color: black;
21   margin: 0; padding: 0;
22   font-family: Meiryo, sans-serif;
23 }
24
25 /* リンク */
26 a:link { color: #00c; }
27 a:visited { color: #00c; }
28 a:hover { color: #f00; }
29 a img { border: none; }
   （以下省略）
```

続いて、コンテンツ部分とサイドバーの大きさを指定します。id属性が「content」と「sidebar」のdivタグにfloatプロパティを指定して、段組みにしています。

LIST chapter03/app/assets/stylesheets/application.css

```
（省略）
/* 全体の枠 */
div#container {
  margin: 0 auto;
  padding-top: 5px;
  width: 780px;
}

/* 左の枠（コンテンツを入れる）*/
main {
  float: left;
  width: 530px;
  padding: 10px 10px 10px 0;
}

/* 右の枠（サイドバーを入れる）*/
aside#sidebar {
  float: left;
  width: 230px;
  background-color: #e8ffff;
  padding: 5px;
  font-size: 86%;
}
（以下省略）
```

特定の部分の中でリンクの色を変えたいときは、「nav.menubar a」のように子孫セレクタを活用します。

LIST chapter03/app/assets/stylesheets/application.css

```
（省略）
/* メニューバーのリンク */
nav.menubar a { text-decoration: none; }

/* メニューバーのリンク（未訪問）*/
```

```
nav.menubar a:link { color: #ccc; }

/* メニューバーのリンク（訪問済）*/
nav.menubar a:visited { color: #ccc; }
  （以下省略）
```

以上でモックアップの完成です。ブラウザで「http://localhost:3000/」を表示すると、スタイルシートで色が付いたページが表示されます。

RESULT

実際には、テンプレートとスタイルシートを交互に修正しながら作成することになるでしょうが、大まかな手順は以上のとおりです。

Chapter 3のまとめ

- **Railsは、HTTPプロトコルでリクエストを受けると、ルーティングの設定に従ってコントローラのアクションを呼び出します。**

- コントローラの中には、複数のアクションが含まれます。コントローラはRubyのクラス、アクションはメソッドとして記述します。
- アクションが実行されると、そのアクションに対応したテンプレート（erbファイル）でレンダリングが行われます。
- アクションの中では、renderメソッドでテンプレートを選んだり、redirect _toメソッドで別のページに移動したりできます。
- テンプレートの中では、<% %>や<%= %>の中にRubyのコードを記述します。
- サイト中のページで共通して使う全体の枠を作るには、レイアウトテンプレートを利用します。ページ中のパーツは部分テンプレートで作成できます。
- サイトの色やフォントはスタイルシートでデザインします。

練習問題

［A］　空欄の中に適切な語句を記入してください。

- 記事の投稿など、サーバーの状態を変更するときには、HTTPの　　メソッドで送信します。
- redirect_toメソッドを使うと、ブラウザに新しいURLを示して別のページへの　　を行うことができます。
- routes.rbを編集すると、URLのパスから特定のアクションを選ぶ　　の設定を変更できます。

［B］　routes.rbに「get "about" => "top#about", as: "about"」という設定をしたとします。テンプレートの中で、TopControllerのaboutアクションへのリンクを作成してください。リンクのテキストは「このサイトについて」としてください。

```
<p>　　</p>
```

［C］　テンプレートの中で、配列@countriesの中身をHTML のリストに並べて表示させてください。

アクション

```
@countries = ["イタリア", "フランス", "ドイツ"]
```

テンプレート

```
<ul>

</ul>
```

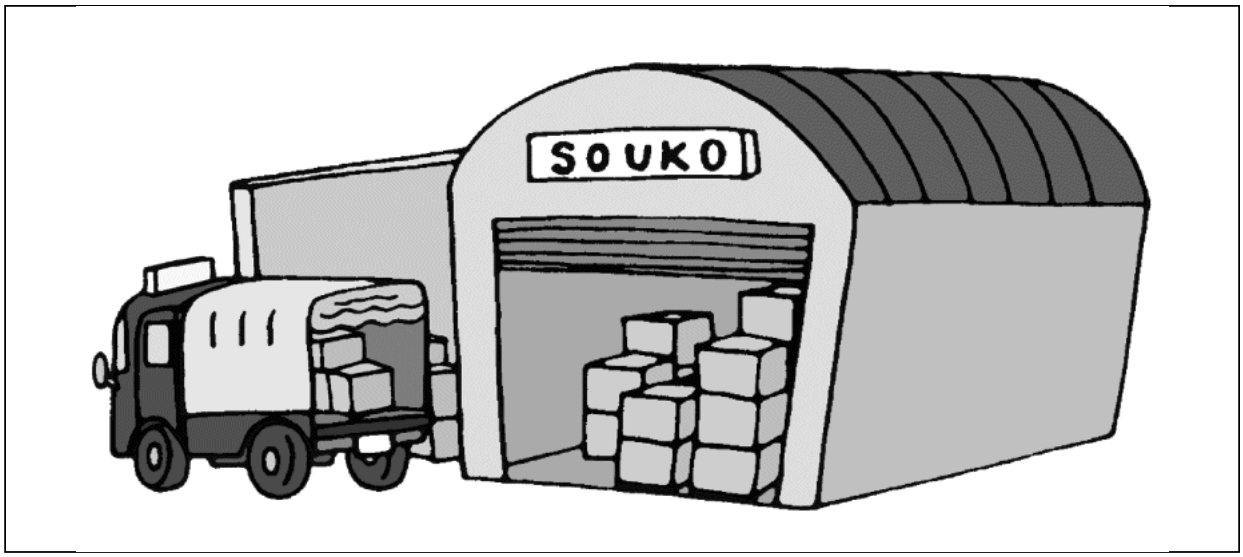
Chapter

4　データベースとモデル

　このChapterでは、MVCアーキテクチャのうちで、データベースとのやり取りを行うコンポーネント、つまりモデルを扱います。モデルについて学習する前に、データベースの作成やテーブルの定義、データの用意が必要なので、そうした準備についても詳しく解説します。

これから学ぶこと

- データベースのしくみとモデルの関係について学びます。
- アプリケーション用のデータベースを作成します。
- モデルを作成する方法を学びます。
- マイグレーションスクリプトを記述してテーブルを定義します。
- シードデータを使って開発用のデータベースに初期データを入れます。
- さまざまなメソッドを使ってテーブルからレコードを取り出したり、検索したりする方法を学びます。



Railsのモデルは、データベースのテーブルに対応したオブジェクトです。モデルの学習のためにデータベースを準備するには何が必要でしょうか？ モデルを使ってデータベースから情報を取り出すには、どうすればよいでしょうか？

4.1 データベースとモデルの基本

このChapterからは、いよいよデータベースを使い始めます。まずは、データベースとRailsのモデルとの関係を紹介します。この節の最後では、実際にデータベースを作成してみます。

データベースとは

何らかのデータ（人物の情報や商品の情報など）を集め、データの操作や検索を行えるようにしたものを**データベース**と呼びます。データベースにはいろいろな種類がありますが、現在よく使われているのはリレーショナルデータベース（関係データベース）です。本書で「データベース」と言えばリレーショナルデータベースのことです。

データベースは、**テーブル**の集合でできています。テーブルとは、Excelの表のように情報を縦横に並べたものです。ただし、Excelとは違って並べ方に決まりがあります。各行（**レコード**）が1つのデータを表し、列（**カラム**またはフィールド）がそのデータの内容を表します。

各列ではデータの型が決まっています。たとえば、価格の列の型を整数とすると、その列には「1890」のような数値を入れます。商品名の列の型は文字列にして「掛け時計」のような文字列を入れます。

列（カラム）

商品番号（整数）	商品名（文字列）	価格（整数）	発売日（日付）
1	掛け時計	1890	2018/03/09
2	ティッシュケース	1050	2017/12/15
3	マグカップ	609	2018/02/28
4	パンダ箸置き	525	2018/04/02

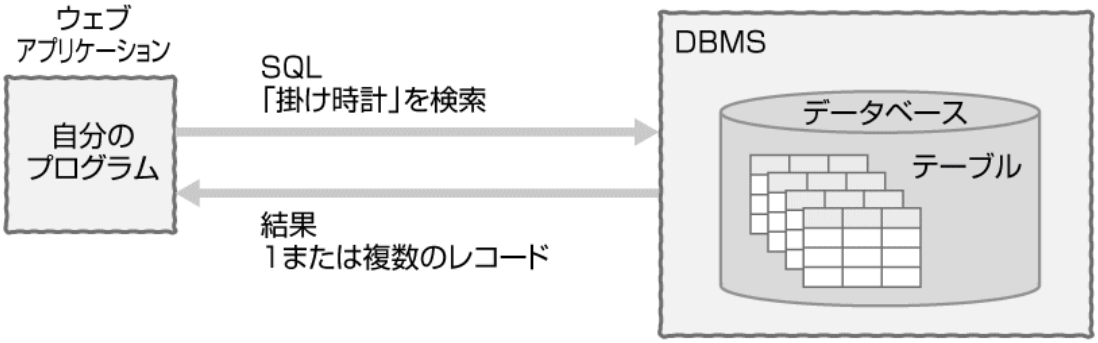
行（レコード、1つのデータ）

テーブル

こうしたいくつものテーブルからなるデータベースは、**データベース管理システム**（**DBMS**）を通して利用します。ウェブアプリケーションでよく使われるDBMSには、オープンソースのMySQLやPostgreSQLなどがあります。本書ではSQLite3を使います。SQLite3は、デスクトップアプリケーション向けのDBMSで、サーバー向けではありませんが、Railsとデータベースの学習には十分な機能を備えています。

MySQLやPostgreSQLを利用したい場合

MySQLやPostgreSQLを利用して本書の学習を進めたい方は、ブラウザでサポートサイト（https://www.oiax.jp/rails5book）を開き、「MySQLを利用したい場合」もしくは「PostgreSQLを利用したい場合」という節を探し、その解説を参照してください。



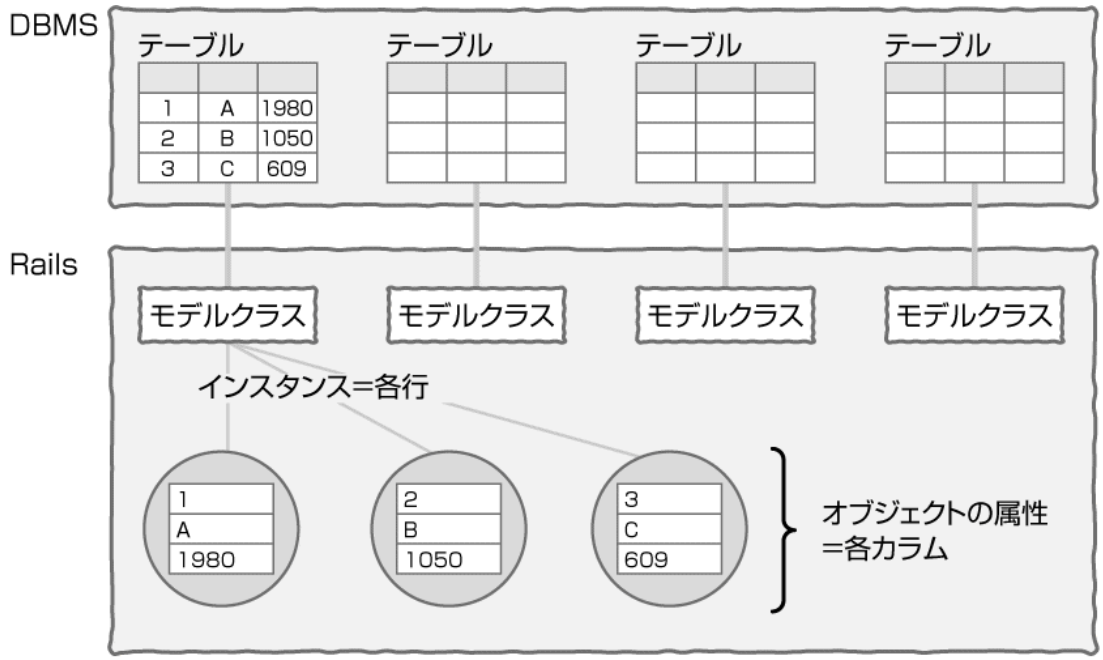
DBMSの利用

Railsのモデル

Railsでは、データベースとのやり取りを行うクラスを**モデル**と呼びます。

■データベースとモデル

モデルは、データベースのテーブルに対応するRubyのクラスです。モデルクラスのインスタンスは、1つの行（レコード）を表すオブジェクトになり、テーブルの列（カラム）に相当する属性を持ちます。たとえば、商品情報のモデルクラスがあるとすると、そのインスタンスは「商品名」、「価格」などの属性を持ちます。



データベースとRailsのモデル

Railsのモデルを使うと、直感的で記述しやすいコードでデータベースを扱うことができます。たとえば123番の商品を表すレコードを取り出すには、モデルクラスのメソッドに番号を渡します。レコードから値を取り出したり値を入れたりするときには、「変数名.カラム名」のように記述します（findメソッドについては「4.4　レコードの取り出しと検索」の「findメソッド」を参照）。

```
product = Product.find(123)  # 123番の商品
name = product.name          # nameカラムから値を取り出す
product.price = 1980         # priceカラムに値を設定
```

Railsのモデルは、メソッド呼び出しを自動的にSQL文に変換してDBMSに送信します。これによって、RailsではSQLの文法を知らなくてもデータベースに対する基本的な操作ができます。本書ではSQLについて詳しく解説しませんが、クエリーメソッドを細かく使いこなすときにはSQLの知識が必要になります。本書とは別にSQLの入門書を読んでおくとよいでしょう（クエリーメソッドについては「4.4　レコードの取り出しと検索」の「クエリーメソッドとリレーションオブジェクト」を参照）。

SQLとは

SQLとは、リレーショナルデータベースで使われる問い合わせ用の言語です。たとえばテーブルproductsからすべてのレコードを取り出すには、次のように記述してDBMSに送ります。

```
SELECT * FROM products
```

SELECT文ではWHERE句で検索条件を指定したり、ORDER句でソートの順番を指定したりできます。次の例は、priceカラムが1000未満のレコードを取り出し、priceカラムの値順（降順）に並べるものです。

```
SELECT * FROM products WHERE price < 1000 ORDER BY price DESC
```

■主キー

先のfindメソッドに渡している番号は、**主キー**の値です。主キーとは、レコードを識別するためのカラムです。1つのテーブルでは、複数のレコード間で主キーの値は重複できません。

Railsの規約では、テーブルに決まった形式の主キーを1個だけ設定することになっています。主キーとなるカラムの名前はidです。値は整数の連番になります。「impress2018-123」のような会社のルールに従った番号でレコードを識別したいときは、主キーのidとは別にproduct_numberのようなカラムを作成するとよいでしょう。

Railsの規約に合わない形式の主キーが設定されているテーブルを取り扱う方法については、「4.2　テーブルの作成」のHINT「主キーとして使用するカラム名の指定」を参照してください。

テーブルから特定のレコードを取り出すときは、モデルクラスのfindメソッドにidの値、つまり主キーの番号を渡します。また、Chapter 10で紹介するように、テーブルを関連付けるときは、主キーを使って1つのテーブルのレコードから別のテーブルのレコードを参照します。

主キー（idカラム）

id	name	product_number
1	掛け時計	impress2018-004
2	ティッシュケース	impress2017-030
3	マグカップ	impress2018-099
4	パンダ箸置き	impress2018-123

主キー

データベースの設定

一からデータベースを作成するときは、次の順で作業をします。この節では、1.と2.の作業を行い、「4.2　テーブルの作成」で3.を、「4.3　データの保存」で4.を行います。

1. DBMSへの接続の設定
2. データベースの作成
3. テーブルの定義と作成
4. テーブルに入れるデータの作成

■接続の設定

DBMSへの接続の設定は、configディレクトリにあるdatabase.ymlで行います。このファイルは、Railsアプリケーションを作成したときに自動的に作られます。内容はYAML形式で書かれています（YAMLについては、「7.2　メッセージの日本語化」の「YAML」を参照）。

database.ymlをテキストエディタで開いて、内容を確認しましょう。「rails new」コマンドは、DBMSを指定しないときはデフォルトでSQLite3用の設定を作ります。

LIST chapter04/config/database.yml

```
  （省略）
default: &default
␣␣adapter: sqlite3
␣␣pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
␣␣timeout: 5000

development:
␣␣<<: *default
␣␣database: db/development.sqlite3

  （省略）
test:
␣␣<<: *default
␣␣database: db/test.sqlite3

production:
␣␣<<: *default
␣␣database: db/production.sqlite3
```

YAML形式では、行頭の字下げ（インデント）幅が重要な意味を持ちます。そこで本書ではYAML形式のソースコードに含まれる行頭のスペースを␣という記号で表現します。

本書に沿ってRailsの学習を進めるうえでは、database.ymlを修正する必要はありません。MySQLやPostgreSQLを利用したい方は、サポートサイトを参照してください。

■3つのモード

Railsには、アプリケーションの**モード**（**環境**とも呼ばれます）が3種類あり、それぞれ別のデータベースを使います。database.ymlでは、development:、test:、production:の下にそれぞれ開発用、テスト用、本番用のデータベースの設定を記述します。

3つのモード

モード名		説明
開発	development	コードを書きながらブラウザで確認するための環境。
テスト	test	自動テストのための環境。
本番	production	ウェブサイトを一般に公開するときの環境。

config/environmentsディレクトリの下には、3つのモードごとの設定ファイルdevelopment.rb、test.rb、production.rbがあり、「本番ではキャッシュを使い、開発では使わない」というような設定が書かれています。

3つのモードは環境変数で区別されます。コマンド「export RAILS_ENV=production」を実行してからrailsコマンドを実行すると、アプリケーションは本番モードのもとで動きます。

モードを調べるには

Railsアプリケーションの中では、「Rails.env == "production"」のようにしてモードを表す文字列を調べられます。また、Rails.envは「モード名?」メソッドを持っていて、Rails.env.development?は開発モードならtrue、そうでなければfalseを返します。

```
if Rails.env.production?
  # 本番だけ実行されるコード...
end
```

データベースの作成

database.ymlの設定に従って、開発用にデータベースを作成しましょう。

■データベースの作成

ターミナルで次のコマンドを実行してください。SQLite3ではこのコマンドを実行しなくても自動的にデータベースができますが、MySQLやPostgreSQLを使っているときは必須です。

```
$ bin/rails db:create
```

SQLite3は、1つのデータベースを1つのファイルとして扱います。db:createによってdbディレクトリの下には2つのファイルdevelopment.sqlite3（開発用のデータベース）とtest.sqlite3（テスト用のデータベース）ができます。

本番用のデータベースを作成するときは、RAILS_ENV=productionを付けます。すると、dbディレクトリの下にproduction.sqlite3ができます。RAILS_ENVはRailsの実行モードを表す環境変数です。

```
$ bin/rails db:create RAILS_ENV=production
```

また、データベースを削除したいときは、次のコマンドを使います。本番用のデータベースを削除したいときは、やはりRAILS_ENV=productionを付けます。

```
$ bin/rails db:drop
```

データベースの中にテーブルを作成するには、次の4.2節で紹介するマイグレーションの機能を使います。また、データベースがちゃんとできたかどうかを確認する方法は、「4.3　データの保存」の「データベースの確認」を参照してください。

■タイムゾーンの設定

モデルを作成したり、マイグレーションを行ったりする前に、Railsアプリケーションのタイムゾーン（標準時の地域）を設定しておきましょう。configディレクトリのapplication.rbを開き、次のように書き換えます。

LIST chapter04/config/application.rb

```
（省略）
20 module Asagao
```

```
21  class Application < Rails::Application
    （省略）
30    # Don't generate system test files.
31    config.generators.system_tests = nil
32
33    config.time_zone = "Tokyo"
34  end
35 end
```

Railsは、データベースのテーブルに時刻情報を保存する際に、時刻をUTC（協定世界時）に変えて保存します。日本標準時は協定世界時から9時間の時差があります。データベースから時刻を取り出して表示するときは、上記の設定によってUTCに9時間が足されたものになります。

タイムゾーンの切り替え

Railsのタイムゾーン機能を使えば、日本にいるユーザーには＋9時間の時刻を見せ、ニューヨークにいるユーザーには－4時間の時刻を見せる、といったことができます。本書では使いませんが、コントローラの中で次のように記述すれば、ユーザーに見せる地方標準時を切り替えられます。

```
Time.zone = "Eastern Time (US & Canada)"
```

4.2 テーブルの作成

ここでは、データベースの中にテーブルを作成し、テーブルにカラム（＝モデルの属性）を加えます。テーブルとカラムの作成は、Railsのマイグレーション機能を使うと簡単に行えます。

モデルの作成

データベースのテーブルに対応するモデルを作成しましょう。本書では、草野球チームMorning Gloryの会員情報をmembersテーブルで管理することにし、そのテーブルに対応するMemberモデルを作ります。

モデルを作成するには、コントローラを作成したときのように、ターミナルから「bin/rails g」コマンドを実行します。「bin/rails g model モデル名」でモデルを作成できます。

Chapter 1で作成したasagaoディレクトリの下で次のように実行してください。

```
$ bin/rails g model member
```

Chapter 3で紹介したとおり、Railsの命名規約ではモデルに対応するデータベースのテーブル名はmembersのように複数形になります。モデルのクラス名は、Memberのように頭が大文字の単数形になります（「3.2　コントローラとアクション」の「命名規約」を参照）。

app/modelsディレクトリの下には、「モデル名.rb」のファイルができます。Memberモデルではmember.rbです。モデルに関するコードはこのファイルに記述します。作成したばかりのモデルクラスは中身が空になっていますが、ApplicationRecordクラスを継承しているのでこれだけでも機能します。

```
class Member < ApplicationRecord
end
```

application_record.rbの使い方

モデルクラスの親クラスであるApplicationRecordは、app/modelsディレクトリの下のapplication_record.rbに記述されています。このクラスにメソッドや定数を加えれば、すべてのモデルで共通して使えます。Chapter 13では、複数のモデルで共用する定数をApplicationRecordで定義しています。

ApplicationRecordの親クラスは、Railsが用意しているActiveRecord::Baseクラスです。

モデルクラスの階層

モデル名の指定

モデル名を作成するときは、「bin/rails g model Member」のようにMemberを大文字で始めてもかまいません。また、entry_imageと指定してもEntryImageとしても、EntryImageモデル（テーブル名はentry_images）ができます。

ただし、membersのように複数形にしてはいけません。Membersモデルができてしまいます。モデル名は単数形にするのが決まりです。

マイグレーション

データベースの中にMemberモデルに対応するmembersテーブルを作りましょう。

■マイグレーションスクリプト

Railsでは、データベースのテーブルの作成や変更に**マイグレーション**という機能を使います。マイグレーションを使うと、自動的にテーブルのカラムを定義できるだけでなく、開発の途中でカラムを追加したり変更したりする作業も楽に行えます。

「bin/rails g model member」でモデルを作成すると、db/migrateディレクトリの下に20180401034353_create_members.rbのような「年月日時分秒_create_テーブル名.rb」というファイルができます。このファイルを**マイグレーションスクリプト**と呼びます。

マイグレーションスクリプトを開いてみましょう。CreateMembersクラス（マイグレーションクラス）にchangeメソッドがあります。changeメソッドの中のcreate_tableメソッドはテーブルの作成を行うもので、このメソッドに渡すブロックの中にカラムを記述します。

```
class CreateMembers < ActiveRecord::Migration[5.2]
  def change
    create_table :members do |t|

      t.timestamps
    end
  end
end
```

CreateMembersクラスの親クラスにActiveRecord::Migration[5.2]が指定されている点に注目してください。角かっこの中の「5.2」は、このマイグレーションスクリプトがRailsバージョン5.2のbin/rails g modelコマンドによって生成されたことを示しています。

カラムは「t.カラムの型:カラム名」という形で記述します。たとえば、nameという文字列型のカラムを追加するには、次のようにします。

```
  create_table :members do |t|
    t.string :name
```

```
  t.timestamps
end
```

主キーとして使用するカラム名の指定

create_tableメソッドでテーブルを作るとき、主キーであるidカラムは自動的に追加されるので、ブロックの中で記述する必要はありません。もし主キーのカラム名をid以外のもの（たとえばmember_id）にしたい場合は、次のようにprimary_keyオプションで指定します。

```
create_table :members, primary_key: "member_id" do |t|
```

そして、モデルクラスの定義の中で主キーのカラム名を指定します。

```
class Member < ApplicationRecord
  self.primary_key = "member_id"
end
```

カラムの型とRubyのクラス

create_tableメソッドの「t.カラムの型」や後述するadd_columnメソッドには、次の表の1列目のように型を指定できます。カラムはモデルクラスの属性になり、カラムの型は属性のクラスの種類になります。カラムの型とRubyのクラスの関係は、完全に1対1にはなっていません。

また、DBMSの種類によっても違いがあります。たとえば、SQLite3には厳密なカラム型はありませんが、MySQLやPostgreSQLはいろいろな種類の型を用意しています。次の表には参考としてMySQLおよびPostgreSQLでのカラム型も掲載しています。

カラムの型とRubyのクラス

マイグレーションの型	Rubyのクラス	MySQL	PostgreSQL
integer	FixedNum	int(11)	integer
decimal	BigDecimal	decimal(10,0)	decimal
float	Float	float	float
boolean	TrueClass/FalseClass	tinyint(1)	boolean
string	String	varchar(255)	character varying

text	String	text	text
date	Date	date	date
datetime	ActiveSupport::TimeWithZone	datetime	timestamp
time	Time	time	time
timestamp	ActiveSupport::TimeWithZone	timestamp	timestamp
binary	String	blob	bytea

membersテーブルの作成

では、asagaoで実際に使用するデータベースの構造を作っていきましょう。membersテーブルのためのマイグレーションスクリプトを次のように書き換えてください。

LIST chapter04/db/migrate/20180523132805_create_members.rb

```
1 class CreateMembers < ActiveRecord::Migration[5.2]
2   def change
3     create_table :members do |t|
4       t.integer :number, null: false                    # 背番号
5       t.string :name, null: false                       # ユーザー名
6       t.string :full_name                               # 本名
7       t.string :email                                   # メールアドレス
8       t.date :birthday                                  # 生年月日
9       t.integer :sex, null: false, default: 1           # 性別 (1:男, 2:女)
10      t.boolean :administrator, null: false, default: false # 管理者フラグ
11
12      t.timestamps
13    end
14  end
15 end
```

ユーザー名（ログイン名）、本名、メールアドレスを文字列型、背番号と性別を整数型のカラムとして作ります。生年月日は日付型（date型＝Dateクラス）にします。論理値型の管理者フラグは、trueの場合は管理者、falseでは一般ユーザーとします。

主キーのカラムは、指定しなくてもidという名前で自動的に作られます（主キーについては「4.1　データベースとモデルの基本」の「主キー」を参照）。

「t.integer :number」や「t.string :name」に付いている「null: false」というオプションに注目してください。これは、空の値（Rubyではnil、SQLではNULL）が保存されないように、カラムにNOT NULL制約を付ける指定です。これにより、空の値を保存しようとすると、DBMSがエラーを出します。

また、「t.integer :sex」に付いている「null: false, default: 1」のオプションは、「カラムにNOT NULL制約を付け、デフォルト値を1とする」という指定です。これにより、sex属性に値を入れないで保存すると自動的に値が1になります。同様に、administrator属性のデフォルト値をfalseにしています。

created_atカラムとupdated_atカラム

上記のマイグレーションスクリプトでは、t.timestampsによってcreated_atとupdated_atという時刻型のカラムが2つできます。Railsは、レコードを作成したときにcreated_atカラムに自動的にその時刻を入れます。また、レコードを更新したときにはupdated_atカラムに時刻を入れます。この2つのカラムを作っておけば、レコードの作成と更新の時刻を調べられるようになります。

続いて、マイグレーションを実行しましょう。ターミナルで次のコマンドを実行します。開発用のデータベースにテーブルmembersが作成され、カラムが加えられます。

```
$ bin/rails db:migrate
```

本番用のデータベースでマイグレーションを行いたいときは、次のようにRAILS_ENV=productionを付けてマイグレーションを実行します。

```
$ bin/rails db:migrate RAILS_ENV=production
```

マイグレーションを行うとdbディレクトリの下にschema.rbというファイルができます。Railsはマイグレーション完了時にその時点におけるデータベースの構造を再現するスクリプトをこのファイルに書き込みます。プログラマが直接このファイルを編集してはなりません。

「type」というカラム名に注意

原則として、カラム名にはtypeという名前は付けられません。本書では解説しませんが、typeカラムはRailsの「単一テーブル継承（single table inheritance; STI）」という機能のために使われます。「種類」を表すカラムを作りたいときは、kindのように別の名前にするか、member_typeのような名前を使ってください。

どうしてもtypeというカラム名を使いたい場合は、モデルクラス定義の中で次のように記述してください。

```
class Member < ApplicationRecord
  self.inheritance_column = nil
```

こうすれば、typeカラムから特別な意味が取り除かれ、普通のカラムとして使えるようになります。

マイグレーションの詳細

ここでは、マイグレーションの細かい機能と使い方を紹介します（あくまで「例」ですので、自分のasagaoアプリケーションでは実行しないでください）。

■カラムの追加

開発中には、新しいテーブルを作るだけでなく、既存のテーブルにカラムを追加したくなることがあります。この作業もマイグレーションで簡単に行えます。

「bin/rails g migration クラス名」を実行すれば、db/migrateディレクトリの下に20180401040230_alter_members.rbのようなファイル名のマイグレーションスクリプトが作成されます。

```
$ bin/rails g migration AlterMembers
```

新しいマイグレーションスクリプトのAlterMembersクラスには、changeメソッドができます。たとえば、membersテーブルに電話番号を表すphoneカラムを追加したいときは、changeメソッドの中にadd_columnメソッドを記述します。

```
class AlterMembers < ActiveRecord::Migration[5.2]
  def change
    add_column :members, :phone, :string
  end
end
```

add_columnメソッドの引数には、テーブル名、追加するカラム名、カラムの型を順に指定します。

「bin/rails db:migrate」を実行すると、新しいバージョンである20180401040230_alter_members.rbが読み込まれ、changeメソッドによってphoneカラムが追加されます。

■マイグレーションのバージョン

マイグレーションスクリプトのファイル名の「年月日時分秒」は、マイグレーションのバージョンを表しています。上記の20180523140500_alter_members.rbのマイグレーションを行ったあとで、もう一度「bin/rails db:migrate」を実行しても何も起こりません。すでにバージョンが20180523140500になっているためです。

古いバージョンに戻したいときは、次のようにバージョンを指定します（ここでは20180523132805_create_members.rbのバージョン）。すると、AlterMembersのchangeメソッドを打ち消す操作が行われ、phoneカラムが削除されます。

```
$ bin/rails db:migrate VERSION=20180523132805
```

マイグレーションのバージョンは、データベース内のschema_migrationsテーブルで管理されています。schema_migrationsテーブルは、bin/rails db:migrateコマンドを実行すると自動的に作成されます。

bin/rails db:migrate:statusコマンドを実行すると、現在のマイグレーションのバージョンを確認できます。up印が付いているのが実行済みのマイグレーションです。

```
$ bin/rails db:migrate:status

database: /Users/taro/rails/asagao/db/development.sqlite3

 Status   Migration ID    Migration Name
--------------------------------------------------
   up     20180523132805  Create members
  down    20180523140500  Modify members
```

データベース定義を古いバージョンに戻すには、bin/rails db:rollbackコマンドも使えます。次のコマンドは、直前に行われたマイグレーションを1つだけ取り消します。

```
$ bin/rails db:rollback
```

これをマイグレーションの「ロールバック」と呼びます。一度に複数個のマイグレーションをロールバックしたい場合は、次のようにSTEPオプションを付加します。

```
$ bin/rails db:rollback STEP=3
```

なお、changeメソッド内にcreate_tableメソッドがあれば、データベース定義のバージョンを下げるときにテーブルが削除されます。

■カラムの変更と削除

マイグレーションでカラムを変更するメソッドには、次のものがあります。

マイグレーション用メソッド

メソッド	機能
add_column(テーブル名, カラム名, 型, オプション)	カラムの追加
rename_column(テーブル名, カラム名, 新しい名前)	カラム名の変更
change_column(テーブル名, カラム名, 型, オプション)	カラムの型の変更
remove_column(テーブル名, カラム名)	カラムの削除

次のマイグレーションスクリプトは、rename_columnメソッドを使ってnameカラムの名前をnicknameに変更します。

```
class AlterMembers < ActiveRecord::Migration
  def change
    rename_column :members, :name, :nickname
  end
end
```

change_columnメソッドとremove_columnメソッドには、changeメソッドの中で使用するとマイグレーションのロールバックができないという制限があります。この2つのメソッドをchangeメソッドに入れてロールバックするとエラーになります。

changeメソッドではできないロールバックを行いたいときは、代わりにupメソッドとdownメソッドの2つを記述してください。upメソッドにマイグレーションを進める処理を書き、downメソッドに取り消す処理を書けば、ロールバックができます。

```
class AlterMembers < ActiveRecord::Migration
  def up
    rename_column :members, :name, :nickname
    change_column :members, :sex, :integer, null: false, default: 2
  end
```

```
  def down
    change_column :members, :sex, :integer, null: false, default: 1
    rename_column :members, :nickname, :name
  end
end
```

■インデックス

カラムには**インデックス**を加えることができます。インデックスとは索引のための情報です。カラムにインデックスを加えると、特定のカラムを使った検索を高速化できます。ただし、テーブルと別に索引情報を持つことになるので、メモリがその分消費されます。

Railsのマイグレーションスクリプトでインデックスを設定するには、add_indexメソッドを使います。省略可能なオプションとしてunique（重複禁止）とname（インデックス名）を指定できます。インデックス名を省略すると、「テーブル名_カラム名_index」がインデックス名になります。

```
add_index :members, :name, unique: true, name: 'name_index'
```

インデックスを削除するには、remove_indexメソッドを使います。columnオプションでカラム名を指定するか、nameオプションでインデックス名を指定します。

```
remove_index :members, column: 'name'
remove_index :members, name: 'name_index'
```

add_indexやremove_indexは、change、up、downの各メソッド内に記述できます。

インデックスは必須

Railsの基本を学習したり、自分用のちょっとしたアプリケーションを書いたりするときにはインデックスは必要ありませんが、実用に耐えるアプリケーションを開発するにはインデックスは必須です。インデックスなしでレコードの数が数百万件になると、反応に何十秒もかかるウェブサイトになることがあります。

たとえば、membersテーブルのnameカラムが頻繁に検索対象になるなら、インデックスを加えます。また、Chapter 10で紹介する外部キーには必ずインデックスを加えましょう。

なお、主キーのidには自動的にインデックスができるので、add_indexで指定する必要はありません。

■開発中と本番でのマイグレーション

開発中にテーブルのカラム定義を変更する方法としては、次の2通りがあります。

1. 新しいマイグレーションスクリプトを追加し、マイグレーションを行う。
2. 既存のマイグレーションスクリプトを書き換え、マイグレーションを最初からやり直す。

アプリケーションのリリース前なら、1.と2.どちらの方法を採ってもかまいません。2.のようにマイグレーションを最初からやり直すには、bin/rails db:migrate:resetコマンドを使います。

```
$ bin/rails db:migrate:reset
```

リリース後の本番サーバーでカラムを変更する場合は、データベースの破棄はできないので、1.の方法を採ることになります。ただし、前回のリリースから次のリリースまでの開発中には、前回のリリース後に追加したマイグレーションスクリプトを書き換えてもかまいません。

なお、MSYS2/MinGW環境でSQLite3を利用している場合、（少なくともRails 5.2.0では）このコマンドは正常に動きません。代わりに次の2つのコマンドを使用してください。

```
$ rm db/development.sqlite3
$ bin/rails db:migrate
```

リリース後のマイグレーションは慎重に

アプリケーションをリリースしたあとのマイグレーションは簡単ではありません。カラムの型を変えるなどテーブル定義に重要な変更を行ったときは、新しい定義に合わせてデータを変換するプログラムを別に書く必要があります。リリース後のマイグレーションは失敗する可能性も少なくありません。本番用のデータベースを試験環境にコピーして、必ず予行練習をするようにしてください。また、データベースのバックアップを取るようにしてください。失敗した場合の対処の手順（バックアップからデータベースを復旧し、アプリケーションのバージョンを戻す）をマニュアル化して、失敗の予行練習もするとよいでしょう。

4.3 データの保存

データベースができ、データベースにテーブルが作成できました。しかし、テーブルの中に実際にデータ（レコード）がないと、モデルの学習もアプリケーションの開発もできません。テーブルに開発用のデータを入れてみましょう。

レコードの作成と更新

モデルを使ってテーブルにレコードを保存する方法を見てみましょう。

Railsコンソール

モデルの機能を簡単にチェックするには、サーバーを起動せずに**Railsコンソール**を使うのが便利です。Chapter 2で紹介したirbと同様にRailsが提供するさまざまなクラスを使ってみることができます。

ターミナルでbin/rails consoleコマンドを実行するとRailsコンソールが起動します。省略形のbin/rails cコマンドも使えます。

```
$ bin/rails c
```

MSYS2/MinGW環境でRailsコンソールを使う

MSYS2/MinGW環境でRailsコンソールを起動するには、次のように通常のコマンドの前にwinpty rubyを加える必要があります。

```
$ winpty ruby bin/rails c
```

なお、MSYS2/MinGW環境におけるRailsコンソールには日本語を入力できないという制限があります。日本語の表示は可能です。

Railsコンソールが起動すると次のような表示になります。

```
Loading development environment (Rails 5.2.0)
irb(main):001:0>
```

irb(main):001:0>の部分はプロンプト（コマンド入力待ちであることを示す記号）です。Rubyのコードを入力すると、その結果（「式.inspect」の戻り値）が=>の右に表示されます。また、SQLが実行されたときはそのSQL文が表示されます。

```
irb(main):001:0> Member.count
   (0.2ms)  SELECT COUNT(*) FROM "members"
=> 0
```

■レコードの作成

テーブルに新しいレコードを追加する手順は、「モデルクラスのインスタンスの作成→saveメソッドの呼び出し」となります。membersテーブルに新しい会員を追加するには、newでインスタンスを作り、属性を設定して、saveメソッドでレコードを保存します。saveメソッドの呼び出しを忘れると、データベースには何も反映されません。

Railsコンソールで試してみましょう。

```
irb(main):002:0> member = Member.new
=> #<Member id: nil, number: nil, name: nil, full_name: nil, （略） >
irb(main):003:0> member.number = 1
=> 1
irb(main):004:0> member.name = "Taro"
```

```
=> "Taro"
irb(main):005:0> member.save
   (0.2ms)  begin transaction
  SQL (0.9ms)  INSERT INTO "members" ("number", "name", （略）
   (0.8ms)  commit transaction
=> true
```

モデルクラスのnewには、「カラム名: 値」を並べたハッシュを渡すこともできます。

```
irb(main):006:0> member = Member.new(number: 1, name: "Taro")
=> #<Member id: nil, number: 1, name: "Taro", full_name: nil, （略） >
irb(main):007:0> member.save
   (0.2ms)  begin transaction
  SQL (1.3ms)  INSERT INTO "members" ("number", "name", （略）
   (1.1ms)  commit transaction
=> true
```

モデルオブジェクトを作成したあとで、「member.assign_attributes(ハッシュ)」で値を入れることもできます。

```
irb(main):008:0> member = Member.new
=> #<Member id: nil, number: nil, name: nil, full_name: nil, （略） >
irb(main):009:0> member.assign_attributes(number: 1, name: "Taro")
=> nil
irb(main):010:0> member.save
   (0.5ms)  begin transaction
  SQL (2.0ms)  INSERT INTO "members" ("number", "name", （略）
   (4.5ms)  commit transaction
=> true
```

クラスメソッドcreateを使うと、モデルオブジェクトの作成と保存が同時に行われます。

```
irb(main):011:0> member = Member.create(number: 1, name: "Taro")
   (0.1ms)  begin transaction
  SQL (0.6ms)  INSERT INTO "members" ("number", "name",  （略）
   (1.0ms)  commit transaction
=> #<Member id: 4, number: 1, name: "Taro", full_name: nil,  （略）  >
```

saveとsave!の違い

saveメソッドはレコードの保存に成功すると、trueを返します。Chapter 7で紹介するバリデーションをモデルに加えると、保存の前に値の検証が行われます。検証に成功してレコードを保存すればtrueを返し、検証が失敗すれば保存を行わずにfalseを返します。

レコードを保存するメソッドには、saveメソッドのほかに!付きのsave!メソッドもあります。save!では、検証が失敗したときにfalseが返るのではなく、例外が発生します。

保存するときにデータベースにエラーが発生したときは、saveでもsave!でも例外が発生します。

メソッド	検証に成功	検証に失敗	データベースにエラー
save	true	false	例外発生
save!	true	例外発生	例外発生

■レコードの更新

すでにテーブルに入っているレコードの情報を変更するときも、saveメソッドを使います。次の例では、モデルクラスのfirstメソッドで最初のレコードを取り出してから、numberカラムを変更して保存しています。

```
irb(main):012:0> member = Member.first
  Member Load (0.3ms)  SELECT  "members".* FROM "members"  （略）
=> #<Member id: 1, number: 1, name: "Taro", full_name: nil,  （略）  >
```

```
irb(main):013:0> member.number = 41
=> 2
irb(main):014:0> member.save
   (1.3ms)  begin transaction
  SQL (1.9ms)  UPDATE "members" SET "number" = ?, "updated_at" = ?
  （略）
   (0.6ms)  commit transaction
=> true
```

assign_attributesメソッドでも属性を変更できます。

```
irb(main):015:0> member.assign_attributes(number: 51, name: "Ichiro")
=> nil
irb(main):016:0> member.save
   (0.2ms)  begin transaction
  SQL (0.7ms)  UPDATE "members" SET "number" = ?, "name" = ?,  （略）
   (0.8ms)  commit transaction
=> true
```

update_attributesメソッドは、assign_attributesメソッドとsaveメソッドを合わせたものです。オブジェクトの属性を変更し、直ちにデータベースに保存します。

```
irb(main):017:0> member.update_attributes(number: 55, name: "Hideki")
   (0.2ms)  begin transaction
  SQL (0.4ms)  UPDATE "members" SET "number" = ?, "name" = ?,  （略）
   (0.6ms)  commit transaction
=> true
```

Railsコンソールはexitで終了できます。

```
irb(main):018:0> exit
```

シードデータの投入

レコードを保存する方法を覚えたら、次にシードデータを使って開発用のデータベースにデータを入れましょう。

■シードデータの使い方

アプリケーションを本番用のサイトでリリースする際には、サイトを公開する前にあらかじめデータベースに初期化用のデータを入れておく必要が出てきます。たとえば、管理者のアカウント情報や都道府県の名前などです。そうしたデータを**シードデータ**と呼びます。

「rails new」コマンドでアプリケーションを作成すると、dbディレクトリの下にseeds.rbというファイルができます。このファイルの中にシードデータを保存するスクリプトを書きます。たとえば、seeds.rbの中でMemberモデルを使って会員情報を保存するスクリプトを書くとします。

```
Member.create(number: 1, name: "Taro", administrator: true)
```

bin/rails db:seedコマンドを実行すれば、このスクリプトが実行されて、開発用データベースに会員「Taro」のレコードが保存されます。bin/rails db:createコマンドやbin/rails db:migrateコマンドと同様にRAILS_ENV=productionを加えれば、本番用のデータベースに保存されます。

```
$ bin/rails db:seed
```

■モード別のシードデータ

ウェブアプリケーションの開発中には、ブラウザ上に実際にデータを表示してみる必要があります。ある程度「もっともらしい」データを入れておいたほうが、楽しく作業ができるでしょう。

本書では、本番用のデータではなく、開発用のデータとしてシードデータを用いることにします。これは、Railsの標準的な方法とは言えませんが、筆者たちが実際のアプリケーション開発で行っている方法です。便利ですので読者の方も使ってみてください。

まず、db/seeds.rbの中身をすべて削除してから、次のコードを書き入れます。

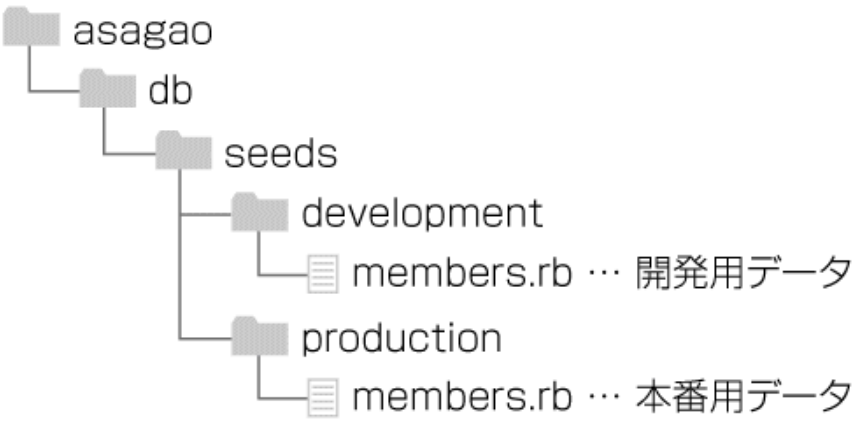
LIST chaper4/db/seeds.rb

```
1 table_names = %w(members)
2 table_names.each do |table_name|
3   path = Rails.root.join("db/seeds", Rails.env, table_name + ".rb")
4   if File.exist?(path)
5     puts "Creating #{table_name}..."
6     require path
7   end
8 end
```

このコードは、db/seeds/developmentディレクトリの下に「テーブル名.rb」があれば、それをrequireメソッドで実行するものです。本番モードではdb/seeds/productionディレクトリの下のファイルを実行します。

Railsアプリケーションのパスの取得

上記のseeds.rbで使っているRails.rootは、アプリケーションのルートパス（たとえば/Users/taro/rails/asagao）を表すオブジェクトを返します。これはPathnameクラスのオブジェクトです。このオブジェクトのjoinメソッドにディレクトリ名をいくつも渡せば、「/Users/taro/rails/asagao/db/seeds/development/members.rb」のようにパスを組み立てられます。



データ投入のためのディレクトリ構成

dbディレクトリの下にseedsディレクトリを作成し、さらにその中にdevelopmentディレクトリを作成してください。developmentディレクトリの中にファイルmembers.rbを作成して、10人の会員を作成するコードを記述します。

LIST chaper4/db/seeds/development/members.rb

```
1 names = %w(Taro Jiro Hana John Mike Sophy Bill Alex Mary Tom)
2 fnames = ["佐藤", "鈴木", "高橋", "田中"]
3 gnames = ["太郎", "次郎", "花子"]
4 0.upto(9) do |idx|
5   Member.create(
6     number: idx + 10,
7     name: names[idx],
8     full_name: "#{fnames[idx % 4]} #{gnames[idx % 3]}",
9     email: "#{names[idx]}@example.com",
10    birthday: "1981-12-01",
11    sex: [1, 1, 2][idx % 3],
12    administrator: (idx == 0)
13  )
14 end
```

■シードデータの再投入

bin/rails db:seedコマンドは単にdb/seeds.rbに書かれたスクリプトを実行するだけですので、すでにシードデータが投入されている状態では使えません。データベースをクリアしてシードデータを投入するコマンドは2つあります。

- bin/rails db:reset
- bin/rails db:migrate:reset db:seed

最初にデータベースを破棄する点は両者共通です。前者では、db/schema.rbを実行してデータベース構造を復元してから、シードデータを投入します。後者では、最初からマイグレーションをやり直してから、シードデータを投入します。前者のほうが実行にかかる時間が短縮できますが、マイグレーションスクリプトを書き換えた場合は後者を使用する必要があります。

さて、Railsの学習中はマイグレーションスクリプトとシードデータを頻繁に書き換えることになりますので、2番目のコマンドのほうが使用頻度が高くなります。そこで、もっと簡単に呼び出せるコマンドを用意しましょう。lib/tasksディレクトリの下に、新規ファイルdatabase.rakeを次のような内容で作成してください（コードの解説は省略）。

LIST chaper4/lib/tasks/database.rake

```
1 namespace :db do
2   desc "Rebuild the development database from scratch"
3   task :rebuild => :environment do
4     sh "rm -f db/development.sqlite3"
5     Rake::Task["db:migrate"].invoke
6     Rake::Task["db:seed"].invoke
7   end
8 end
```

このファイルを設置した結果、次のコマンドが使えるようになります。

```
bin/rails db:rebuild
```

本書では、シードデータを再投入する際にこのコマンドを使用します。なお、このコマンドはデータベース管理にSQLite3を使っていることが前提となっています。PostgreSQLやMySQLを利用している場合は使えません。

db:rebuildタスク

ファイルdatabase.rakeでは、タスク実行ツールRakeで使用するカスタムタスクdb:rebuildが定義されています。この中では、開発モード用のSQLite3データベースを記録しているファイルdb/development.sqlite3を削除し、マイグレーションを実行し、シードデータを投入するまでの一連の流れが記述されています。

本文では簡単にシードデータの再投入を行う目的でタスクdb:rebuildを定義したように書きましたが、実を言えば、別の問題を回避しようとしたことが発端でした。

本書の執筆中に筆者たちはMSYS2/MinGWでbin/rails db:resetコマンドやbin/rails db:migrate:resetコマンドがうまく動かないという問題に遭遇しました。次のようなエラーメッセージが出て止まってしまうのです。

```
Permission denied @ unlink_internal - C:/Ruby25-
x64/msys64/home/oiax/rails/asagao/db/development.sqlite3
Couldn't drop database 'db/development.sqlite3'
rails aborted!
(以下省略)
```

これはMSYS2/MinGWでSQLite3を利用する場合にだけ発生します。本書執筆時点では直接的な解決法が見つからなかったので、代替策としてタスクdb:rebuildを用意することにしました。Railsの将来のバージョンではこの問題が解消されるかもしれません。

データベースの確認

データベースの作成、テーブルの作成、データの投入がうまくいっているかどうかを確認するために、SQLite3のコマンドを入力してみるのもよいでしょう。SQLite3のコンソールを開くには、ターミナルで次のコマンドを実行します。ただし、MSYS2/MinGW環境では先頭にwinptyを付けてください。

```
$ sqlite3 db/development.sqlite3
```

「.tables」と入力すると、データベース内のテーブル一覧を表示します。

```
sqlite> .tables
```

「.schema テーブル名」でテーブル定義を表示できます。

```
sqlite> .schema members
```

SQLのSELECT文を入力すれば、membersテーブルのレコードの一覧を表示できます。

```
sqlite> SELECT * from members;
```

「.quit」とするとSQLite3からログアウトします。

```
sqlite> .quit
```

4.4 レコードの取り出しと検索

Railsのモデル（Active Record）は、データベースからデータを取り出したり、検索したりするための強力な機能を備えています。4.3節で投入したデータを使ってモデルの検索機能を確かめましょう。

findとfind_by

テーブルのレコード1つをモデルオブジェクトとして取り出すには、findメソッドかfind_byメソッドを使います。

■findメソッド

モデルのクラスメソッドfindメソッドにidカラム（主キー）の値を指定すると、その値を持つレコード（モデルオブジェクト）を取り出せます。

このメソッドを試す前に、実際のレコードのidカラムを調べておきましょう。クラスメソッドidsメソッドを使えば、テーブルに存在するすべてのレコードの主キーを配列として取得できます。

```
irb(main):001:0> Member.ids
   (0.2ms)  SELECT "members"."id" FROM "members"
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

この中から好きなidを選んで、findメソッドでレコードを取り出し、変数memberに入れてみましょう。

```
irb(main):002:0> member = Member.find(3)
  Member Load (2.9ms)  SELECT  "members".* FROM "members"  （略）
=> #<Member id: 3, number: 12, name: "Hana", full_name: "高橋 花子"
（略） >
```

変数memberに対して、カラムと同名のメソッド（属性）を呼ぶと、そのカラムの値が返ります。

```
irb(main):003:0> member.email
=> "Hana@example.com"
```

レコードに存在しないidをfindメソッドに指定すると、例外ActiveRecord::RecordNotFoundが発生します。Member.find(123)のように適当な数字を入れて試してください。

```
irb(main):004:0> Member.find(123)
  Member Load (0.3ms)  SELECT  "members".* FROM "members"  （略）
ActiveRecord::RecordNotFound: Couldn't find Member with 'id'=123
（省略）
```

■find_byメソッド

モデルクラスのクラスメソッドfind_byは、あるカラムを使ってレコードを検索し、最初に一致したものを返します。次の例は、nameカラムが「Taro」であるレコードを1つ取り出します。引数には「name: "Taro"」のようにハッシュで「カラム名: 値」を指定します。

```
irb(main):005:0> member = Member.find_by(name: "Taro")
  Member Load (0.6ms)  SELECT  "members".* FROM "members"  （略）
```

```
=> #<Member id: 1, number: 10, name: "Taro", full_name: "佐藤 太郎",
（略） >
```

検索対象のカラムはいくつも指定できます。次の例は、性別（sex）が1（男性）で、管理者（administrator）ではないレコードを取り出します。

```
irb(main):006:0> member = Member.find_by(sex: 1, administrator: false)
  Member Load (8.8ms)  SELECT  "members".* FROM "members"  （略）
=> #<Member id: 2, number: 11, name: "Jiro", full_name: "鈴木 次郎",
（略） >
```

指定された条件に一致するレコードがない場合は、nilが返ります。

```
irb(main):007:0> member = Member.find_by(sex: 2, administrator: true)
  Member Load (0.7ms)  SELECT  "members".* FROM "members"  （略）
=> nil
```

findメソッドとfind_byメソッドの使い分け

findメソッドを使うときは、例外が発生することを前提にしたプログラムを作ります。「11.2　エラーページのカスタマイズ」では例外ActiveRecord::RecordNotFoundを捕捉して「見つかりません」ページを表示する方法を紹介しています。

場合によっては例外を発生させたくないこともあります。そうしたときはfind_byメソッドにidを指定してください。レコードが見つからない場合は、例外は発生せずにnilが返ります。

```
member = Member.find_by(id: 123)
```

クエリーメソッドとリレーションオブジェクト

クエリーメソッドを使うと、検索条件を読みやすいコードで記述できます。クエリーメソッドで検索条件を細かく指定する方法を見てみましょう。

■クエリーメソッドとリレーションオブジェクト

クエリーメソッドの中で一番よく使われるのは、SQL文のWHERE句にあたるwhereメソッドです。whereメソッドを使うと、検索条件に一致する複数のレコードを取り出せます。

次の例では、引数にハッシュで「nameカラムの値がTaro」という検索条件を指定しています。Railsコンソールの結果には、一見モデルオブジェクトの配列のようなものが表示されます。

```
irb(main):008:0> members = Member.where(name: "Taro")
  Member Load (0.4ms)  SELECT "members".* FROM "members" WHERE
（略）
=> #<ActiveRecord::Relation [#<Member id: 1, number: 10, name: "Taro",
（略）
```

クエリーメソッドが実際に返すのは、配列ではなくActiveRecord::Relationクラスのオブジェクトです。本書では、これを**リレーションオブジェクト**と呼びます。このオブジェクトの役割は、データベースからデータを取り出すための検索条件を保持することと、検索を実行してその結果をモデルの配列として使えるようにすることです。

リレーションオブジェクトが保持している検索条件から作られるSQL文を調べるには、to_sqlメソッドを使います。ここでは見やすいように、whereメソッドの呼び出しのあとに「; nil」を付けて結果の表示を消しています。

```
irb(main):009:0> members = Member.where(name: "Taro") ; nil
=> nil
irb(main):010:0> puts members.to_sql
SELECT "members".* FROM "members" WHERE "members"."name" =
```

```
'Taro'
=> nil
```

引数には、WHERE句に指定する検索条件を文字列で指定することもできます。

```
irb(main):011:0> members = Member.where("number < 20")
  Member Load (0.5ms)  SELECT "members".* FROM "members" WHERE
(number < 20)
=> #<ActiveRecord::Relation [#<Member id: 1, number: 10, name: "Taro",
（略）
```

■検索が実行されるタイミング

whereメソッドなどでリレーションオブジェクトを作成すると、そのままでは検索は実行されません。検索条件を保持しているだけです。

リレーションオブジェクトはRubyの配列と同じ名前のメソッドを持っています。each、map、lengthなどです。そうしたメソッドを呼び出すと、そのときに初めてSQLによる検索を実行し、モデルオブジェクトの配列を利用できるようになります。

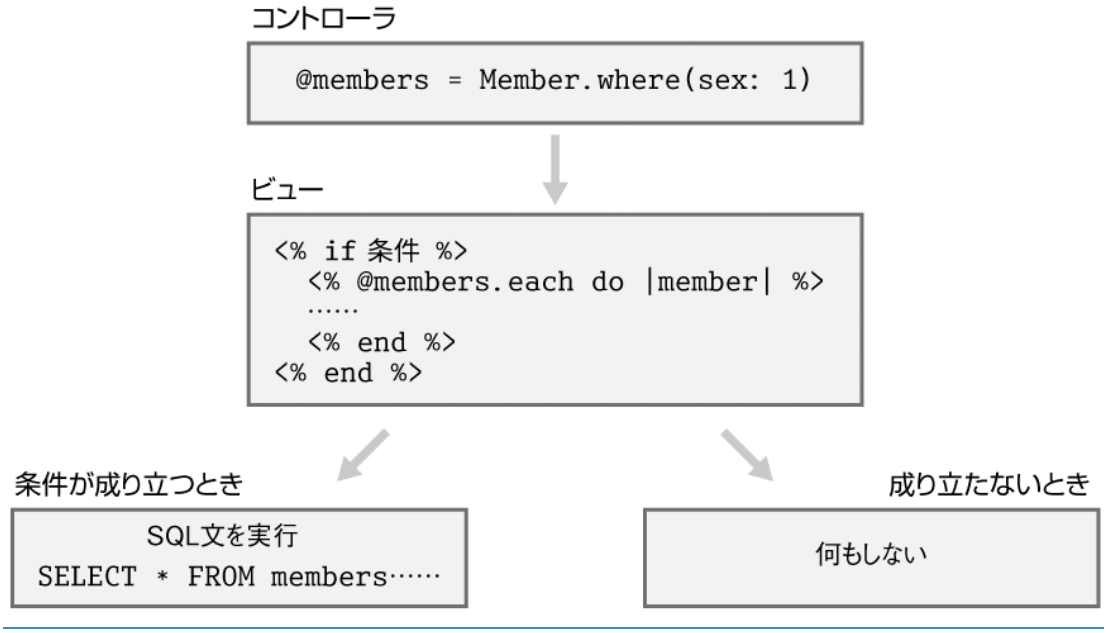
これまでの例では「#<Member id: 1, number: 10,……」のように検索結果が表示されていますが、これはRailsコンソールがinspectメソッドを呼び出したために、検索が実行されてしまったからです。

たとえば、コントローラの中でインスタンス変数@membersにリレーションオブジェクトをセットしたとします。この段階では検索は実行されません。

```
@members = Member.where(sex: 1)
```

次のようにビューの中でeachメソッドを呼び出すと、そのときに検索が実行されます。もし「if 条件」が成立しないときは、検索は実行されません。

```
<% if 条件 %>
  <% @members.each do |member| %>
    <%= member.name %>
  <% end %>
<% end %>
```



Lazy Loading

このように、リレーションオブジェクトは見かけ上は配列と同じように振る舞いますが、「実際に検索を実行してデータを取り出すのは、データが必要になったとき」という特徴を持っています（Lazy Loadingと呼ばれます）。これにより、すっきりしたコードで余計な検索の実行を省くことができます。

なお、即座に検索を実行してモデルオブジェクトの配列を作りたい場合は、loadメソッドを使います。

```
@members = Member.where(sex: 2).load
```

インスタンス変数@membersはリレーションオブジェクトですが、内部にモデルオブジェクトの配列を持つ状態になります。

lengthメソッドとsizeメソッド

lengthメソッドやsizeメソッドを使うと、リレーションオブジェクトが何件の検索結果を返すのかを調べられます。この2つのメソッドは動作が違います。lengthメソッドは、検索を実行してモデルオブジェクトの配列を作り、その配列の長さを返します。sizeメソッドは、後述するcountメソッドを呼び出して、SQLの機能で件数を直接取得します。普通はsizeメソッドを使いますが、検索条件が確定していて、あとで必ずモデルオブジェクトの配列を作ることになるなら、lengthメソッドのほうがデータベースへの問い合わせが減ります。

■クエリーメソッドの重ね合わせ

リレーションオブジェクトに対してさらにクエリーメソッドを呼び出すと、検索条件を追加できます。次の例では、「nameカラムの値がTaro」かつ「numberカラムの値が20未満」という検索条件ができます。

```
members = Member.where(name: "Taro")
members = members.where("number < 20")
```

この例は、次のように1行にまとめられます。

```
members = Member.where(name: "Taro").where("number < 20")
```

検索結果のソート順を指定するにはクエリーメソッドorderを用います。次の例は、性別（sexカラム）の値が2（女性）の会員を背番号（numberカラム）の昇順で取り出します。

```
irb(main):012:0> members = Member.where(sex: 2).order("number")
  Member Load (0.9ms)  SELECT "members".* FROM "members" WHERE
（略）
```

```
=> #<ActiveRecord::Relation [#<Member id: 3, number: 12, name: "Hana"
（略）
```

orderメソッドを使って降順で並べたいときは、引数をハッシュにして「カラム名: :desc」を指定します。

```
members = Member.where(sex: 2).order(number: :desc)
```

このほかのおもなクエリーメソッドには次のものがあります。selectメソッドやgroupメソッドなどでSQLのSELECT文を構成できます。ただし、SQLとは違って、クエリーメソッドをつなげる順番は自由です。

クエリーメソッド一覧

メソッド	SELECT文の構成部分	補足
select(文字列)	SELECT	
from(文字列)	FROM	
joins(シンボル／文字列)	JOIN	テーブル結合。
left_outer_joins(シンボル／文字列)	LEFT OUTER JOIN	テーブル結合。
includes(シンボル)	DBMSによる	関連付けの名前を指定すると、関連するモデルオブジェクトを同時にロードする。
where(ハッシュ／文字列／配列)	WHERE	検索条件の指定。
group(文字列)	GROUP BY	
having(文字列)	HAVING	
limit(整数)	LIMIT	取得するレコード数の上限。
offset(整数)	DBMSによる	何行目からレコードを取得するか。
order(文字列)	ORDER BY	ソート対象のカラム名。
reorder(文字列)	ORDER BY	ORDER BY句の上書き。

reverse_order	ORDER BY	ORDER BY句の昇順・降順を逆にする。
lock(文字列)	DBMSによる	MySQLのロック機能を使う。
readonly	—	レコードを読み込み専用にする。
distinct	—	検索結果の重複をなくす。
none	—	何も検索しないリレーションオブジェクトを返す。

■ファインダーメソッドとの組み合わせ

リレーションオブジェクトには**ファインダーメソッド**と呼ばれるメソッドが備わっています。たとえば、firstメソッドは、検索条件に一致するレコードを先頭から1個だけ取り出し、モデルオブジェクトを1個（存在しなければnilを）返します。

```
irb(main):013:0> member = Member.where(sex: 2).order(:number).first
  Member Load (0.4ms)  SELECT  "members".* FROM "members" WHERE
（略）
=> #<Member id: 3, number: 12, name: "Hana", full_name: "高橋 花子"
（略）
```

末尾からレコードを1個だけ取り出すlastメソッドもあります。なお、firstメソッドやlastメソッドを使うときは、orderメソッドでソート順を指定してください。ソート順がないと結果が不定になり、見つけにくいバグを埋め込むことになりかねません。

ファインダーメソッドにはほかに、findとfind_byがあります。この2つのメソッドの機能は、紹介済みのモデルのクラスメソッドfind、find_byと同じです。次の例は、女性会員の中から主キーが1のレコードを取り出します。

```
Member.where(sex: 2).find(1)
```

ただし、ファインダーメソッドはクエリーメソッドの検索条件に縛られる点に注意してください。主キーが1である会員が存在したとしても、その会員の性別が1でなければ例外

ActiveRecord::RecordNotFoundが発生します。find_by(id: 1)とすれば例外が発生せずにnilが返ります。

■whereメソッドの便利な使い方

whereメソッドに「カラム名: 配列」というハッシュを渡すと、カラムの値が「複数の候補のどれかと同じ」という検索条件を指定できます。次の例では、背番号が15か17か19の会員を取り出します。

```
irb(main):014:0> members = Member.where(number: [15,17,19])
  Member Load (0.6ms)  SELECT "members".* FROM "members" WHERE
"members"."nu
mber" IN (15, 17, 19)
=> #<ActiveRecord::Relation [#<Member id: 6, number: 15, name:
"Sophy" （略）
irb(main):015:0> members.map(&:name)
=> ["Sophy", "Alex", "Tom"]
```

ハッシュの値を範囲オブジェクトにすると、あるカラムの値がその範囲にあるという検索条件を指定できます。次の例では、背番号が12以上14以下の会員を取り出します。

```
irb(main):016:0> members = Member.where(number: 12..14)
  Member Load (0.4ms)  SELECT "members".* FROM "members" WHERE
("members"."nu
mber" BETWEEN 12 AND 14)
irb(main):017:0> members.map(&:name)
=> ["Hana", "John", "Mike"]
```

whereメソッドの直後にnotメソッドを指定すると、「○○でない」という検索条件が使えます。whereメソッドは引数なしにして、notメソッドの引数に条件を指定してください。次の例は、「ユーザー名がTaroでない」会員を検索します。

```
Member.where.not(name: "Taro")
```

■プレースホルダー

whereメソッドには文字列を指定することができます。この文字列に疑問符（?）を含めると、**プレースホルダー**になります。プレースホルダーとは、指定した値をSQL文の中に埋め込むための印です。

次の例をご覧ください。見やすいように、whereメソッドの呼び出しのあとに「; nil」を付けて結果の表示を消しています。

```
irb(main):018:0> name = "Taro"
=> "Taro"
irb(main):019:0> members = Member.where("name = ?", name) ; nil
=> nil
irb(main):020:0> puts members.to_sql
SELECT "members".* FROM "members" WHERE (name = 'Taro')
=> nil
```

whereメソッドには引数を2つ指定しています。第1引数はプレースホルダーを入れたSQL文のWHERE句で、第2引数は文字列です。リレーションオブジェクトは、プレースホルダーの位置に'Taro'のように一重引用符付きで第2引数の文字列を埋め込みます。

第1引数にプレースホルダーがいくつもある場合には、その個数に応じて引数の数を増やしてください。

プレースホルダーを使うと、単純なwhereメソッドでは作れない検索条件を指定できます（ただしSQLの知識が必要です）。たとえば、「または」という条件はこれまで紹介した機能

では作れません。次のようにSQLのORとプレースホルダーを使えば、「ユーザー名がJiroであるか、または管理者である」会員を検索できます。

```
members = Member.where("name = ? OR administrator = ?", "Jiro", true)
```

SQLインジェクションを防ぐ

whereメソッドで検索を行うときは、次のように変数を直接埋め込んではなりません。

```
@members = Member.where("name = '#{name}'")
```

変数の中にSQL文にとって意味のある文字（一重引用符など）が含まれていると、データベースが不正に利用されることがあるからです。このようなSQLを悪用したサーバーへの攻撃は「SQLインジェクション」と呼ばれます。whereメソッドで変数を使うときは、ハッシュで指定するか、プレースホルダーで変数を展開すれば、SQL文にとって意味のある文字は適切に処理されます。

たとえば、変数nameが"Ta'ro"という値なら、Member.where("name = ?", name)が返すリレーションオブジェクトは次のSQL文を作ります。

```
SELECT "members".* FROM "members" WHERE (name = 'Ta''ro')
```

■集計用のメソッド

モデルクラスでは、次のような便利な集計用のメソッドが用意されています。

集計用のメソッド

メソッド	機能
average(カラム名)	平均
count	レコードの数
maximum(カラム名)	最大値
minimum(カラム名)	最小値
sum(カラム名)	合計

たとえば、numberカラムのうち数字が一番大きいものを取り出すには、次のようにします。

```
irb(main):021:0> Member.maximum("number")
  (0.7ms)  SELECT MAX("members"."number") FROM "members"
=> 19
```

集計用のメソッドは、リレーションオブジェクトでも使えます。次の例は、男性会員の数を数えます。

```
irb(main):022:0> Member.where(sex: 1).count
  (0.2ms)  SELECT COUNT(*) FROM "members" WHERE "members"."sex" = ? （略）
=> 7
```

SQL文を直接指定するには

モデルクラスでは、findメソッドだけでなくfind_by_sqlメソッドを使ってSQL文を直接指定することもできます。

```
members = Member.find_by_sql(
  "SELECT * FROM members WHERE number = 11")
```

whereメソッドと同じくプレースホルダー（?）も使えます。ただし、whereとは異なり、引数全体を配列で指定する必要があります。

```
members = Member.find_by_sql(
  ["SELECT * FROM members WHERE name = ?", "Taro"])
```

Railsコンソールの出力を変えるには

Railsコンソール上で、式の値が長々と表示されてわずらわしいときは、次の指定で値の出力を止められます。

```
irb(main):001:0> conf.echo = false
irb(main):002:0> member = Member.first
  Member Load (0.4ms)  SELECT  `members`.* FROM `members`  （略）
irb(main):003:0>
```

値の表示を元に戻すには、「conf.echo = true」と入力します。

さらに、SQLの出力も止めたい場合は次のように入力します。SQLの出力を再開したいときは、コンソールを開き直してください。

```
irb(main):003:0> ApplicationRecord.logger = nil
irb(main):004:0> member = Member.first
irb(main):005:0>
```

Chapter 4のまとめ

- データベースはテーブルの集合でできています。モデルは、データベースのテーブルに対応するオブジェクトです。モデルクラスのインスタンスは、テーブルの1つのレコードにあたります。
- データベースの設定は、database.ymlファイルで行います。3つの環境に合わせて、3つのデータベースを用意します。
- データベースの中にテーブルを作成するには、マイグレーションスクリプトを記述します。

- **データベースの作成やマイグレーションには、bin/railsコマンドを使います。**
- **本書では、開発用のデータベースに初期データを投入するために、シードデータを使います。**
- **モデルを使ってレコードを保存するには、saveメソッドを使います。**
- **モデルを使ってレコードを取り出すには、findメソッドやfind_byメソッドを使います。**
- **モデルクラスに対してwhereやorderなどのクエリーメソッドを呼び出すと、検索条件群を保持するリレーションオブジェクトを返します。**

練習問題

［A］　データベースに書籍を管理するbooksテーブルを作成することにします。マイグレーションスクリプトにカラムの定義を記述してください。title（書籍名）、author（著者名）、price（価格）の3つのカラムを用意し、カラムの型はそれぞれ文字列、文字列、整数とします。また、すべてのカラムにNOT NULL制約を付けてください。

```
class CreateBooks < ActiveRecord::Migration[5.2]
  def change
    create_table :books do |t|
      
      
      t.timestamps
    end
```

```
  end
end
```

［B］　Bookモデルを使って、問題［A］のbooksテーブルに書籍のデータを保存するためのコードを書きます。2つの空欄に書籍名を設定するコードと、データを保存するコードを書いてください。書籍名は好きなものでかまいません。

```
book = Book.new
[                                        ]
book.author = "夏目漱石"
book.price = 1200
[                                        ]
```

［C］　Bookモデルを使って、booksテーブルからidが123の書籍をモデルオブジェクトに取り出すコードを書いてください。

```
[                                        ]
```

［D］　booksテーブルから著者が「夏目漱石」の書籍を取り出します。Bookモデルとクエリーメソッドを使ってリレーションオブジェクトを作ってください。

```
[                                        ]
```

［E］　booksテーブルから価格が3000円未満の書籍を取り出します。Bookモデルとクエリーメソッドを使ってリレーションオブジェクトを作ってください。

```
[                                        ]
```

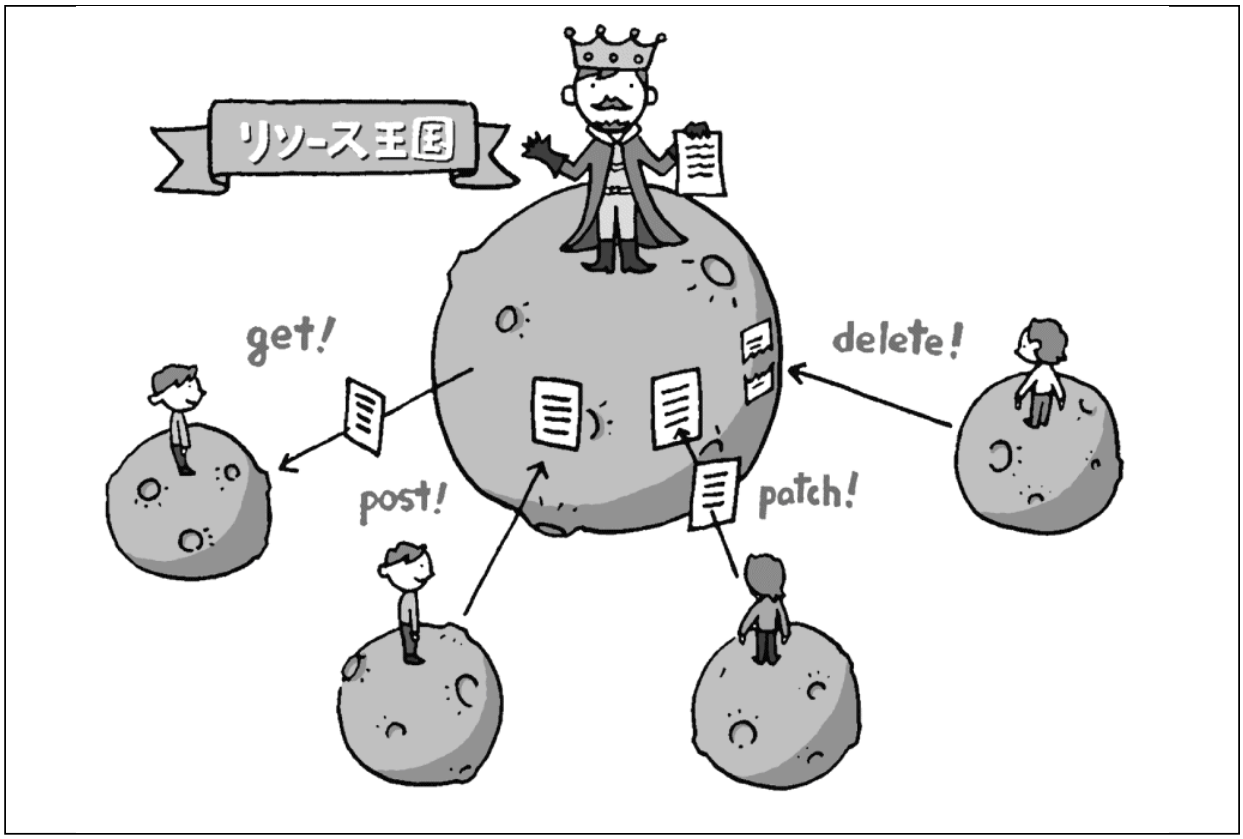
Chapter

5　リソースを扱うコントローラ

　コントローラとモデルの基本について学習が終われば、その2つを組み合わせて本格的なアプリケーションの開発を始められます。このChapterでは、RailsのRESTの原則に従って、サイトの会員情報をリソースとして扱うコントローラの作成を始めます。

これから学ぶこと

- リソースベースのルーティングの設定について学びます。
- リソースを表すURLのパスの書き方について学びます。
- 会員情報を扱うMembersControllerを作成し、会員の一覧ページと詳細情報のページを実装します。
- 7つのアクションの名前と機能を覚えます。



Railsアプリケーションでは、RESTの原則に従ってデータを「リソース」として扱います。リソースの表示、作成、更新、削除の機能を用意すれば、アプリケーションの組み立てが簡単になります。RESTフルなアプリケーションを作るには、何をすればよいでしょうか？

5.1 RESTとルーティング

RESTは、Railsアプリケーションを開発するうえで重要な概念です。RESTに基づくことによって、モデルを扱うコントローラを効率のよいパターンで記述しながら、セキュリティに配慮したアプリケーションを開発できます。

リソースベースのルーティング

REST（REpresentational State Transfer）とは、ネットワーク上に置かれたリソースを操作するアプリケーション作成のスタイルです。RailsのREST機能の中心は、リソースベースのルーティングです。

■RESTとリソース

Railsには、RESTに基づいた（「RESTフル」と呼ばれます）作法でウェブアプリケーションを作成する機能があります。RESTの機能を利用すれば、Railsの原則「Don't Repeat Yourself」や「設定より規約」を推し進めることができ、アプリケーションの開発や保守がより簡単になります。

Railsにおける**リソース**とは、コントローラが扱う対象に名前を付けたものです。リソース名を設定するには、config/routes.rbにresourcesメソッドを1行記述するだけです。引数にはリソース名の複数形を指定します。たとえば、会員情報をリソースとして扱うには、次の行を記述します。

```
resources :members
```

これだけで、MembersControllerに対して後述の7つのアクションのルーティングが設定できます。これをRESTフルなルーティング、またはリソースベースのルーティングと呼びます。リソースを扱うコントローラは、MembersControllerのように「リソース名の複数形＋Controller」という名前にするのが基本です。

Railsだけでなく、ウェブアプリケーション一般におけるRESTの意味については、コラム「RESTの原則」を参照してください。

モデルと対応しないリソース

resourcesメソッドに指定するリソース名はたいていはモデル名と同じですが、リソースとモデルは1対1に対応していなくてもかまいません。resourcesメソッドは、リソース名に対応したコントローラに対して、7つのアクションのルーティングを自動的に設定するだけです。Chapter 8では、ログイン機能のためにSessionsControllerを作りますが、Sessionモデルは必要ありません。

■リソースを扱うコントローラ

リソースを扱うコントローラでは、リソースベースのルーティングのパターンに従い、決まった名前のアクションを7つ用意します。これによって、モデルを操作するコントローラは一貫したスタイルで作ればよいことになり、開発効率を向上できます。

7つのアクションには、原則として次の機能を持たせます。この7つは、データベースの基本操作であるCRUD（Create、Read、Update、Delete）を実装したものでもあります。

index
リソースの一覧を表示する（テーブルのレコード一覧を表示する）。

new
リソースを追加する（テーブルに新しいレコードを作成する）ためのフォームを表示する。

create
リソースを作成する（テーブルに新しいレコードを作成する）。

show
リソースの属性を表示する（レコードの内容を表示する）。

edit
リソースを更新する（既存のレコードのカラムを更新する）ためのフォームを表示する。

update
リソースを更新する（既存のレコードのカラムを更新する）。

destroy
リソースを削除する（テーブルからレコードを削除する）。

■パスとHTTPメソッド

7つのアクションを呼び出すには、次ページの表のパスとHTTPメソッドの組み合わせを使います。リソースの集合を扱うindexアクションとcreateアクションは同じURLになり、HTTPメソッドで区別されます。個別のリソースを扱うshow、update、destroyも同じURLになり、HTTPメソッドで区別されます。

show、edit、update、destroyの各アクションを呼ぶときは、/members/123の「123」のように、モデルの主キーを示すidパラメータが必須です。このidパラメータは、コントローラの中でparams[:id]で取り出せます。

リソースベースのルーティングでのパスとHTTPメソッド

アクション	パス	HTTPメソッド
index	/members	GET
show	/members/123	GET
new	/members/new	GET
edit	/members/123/edit	GET
create	/members	POST
update	/members/123	PATCH
destroy	/members/123	DELETE

PATCHメソッドとDELETEメソッド

HTMLの仕様では、フォームのmethod属性にはGETとPOSTしか指定できません。このため、リソースベースのルーティングを使ったときでも、PATCHメソッドとDELETEメソッドは実際のリクエストではPOSTメソッドで送信されます。

Railsは隠しパラメータ_methodを使い、PATCHメソッドでは_method=patch、DELETEメソッドでは_method=deleteをリクエストに加えることでメソッドの種類を擬似的に表します。

PUTメソッドとPATCHメソッド

Rails 3まではリソースの更新（updateアクション）にはPUTメソッドが使われていましたが、Rails 4からはPATCHメソッドに変わりました。「リソースの置き換え」を意味するPUTメソッドよりも、「リソースの一部変更」という意味のPATCHメソッドのほうがふさわしい、という考えからです。

Rails 4以降でもPUTメソッドを使えますので、Rails 3で作ったアプリケーションをRails 4やRails 5にアップグレードするときは、PUTをPATCHに修正しなくても動きます。

■アクションの追加

7つのアクション以外にも、リソースベースのルーティングでは任意のアクションを追加できます。アクションを追加するには、resourcesメソッドにブロックを渡し、ブロックの中で「HTTPメソッドを表すメソッド アクション名」を記述します。

その際、会員一覧のようにリソースの集合を表すアクションは、onオプションに:collectionを指定します。会員の状態変更のように個別のリソースを扱うアクションは、onオプションに:memberを指定します。

次の例は、MembersControllerにsearch、suspend、restoreの3つのアクションを加える設定です。getやpatchのようにHTTPメソッドを表すメソッドには、複数のアクションを指定できます。

```
resources :members do
  get "search", on: :collection          # メンバーの検索
```

```
  patch "suspend", "restore", on: :member  # メンバーの停止・再開
end
```

追加されたアクションを呼び出すパスとHTTPメソッドは次のとおりです。searchアクションはリソースの集合を扱うので、idパラメータはありません。suspendアクションとrestoreアクションは個別のリソースを扱うので、idパラメータが必須になります。

追加のアクション

アクション	パス	HTTPメソッド
search	/members/search	GET
suspend	/members/123/suspend	PATCH
restore	/members/123/restore	PATCH

アクションを追加するには、もう1つ書き方があります。onオプションで指定する代わりにcollectionブロックまたはmemberブロックで囲む方法です。上記の例と同じ結果になります。

```
resources :members do
  collection { get "search" }          # メンバーの検索
  member { patch "suspend", "restore" } # メンバーの停止・再開
end
```

逆に、7つのアクションのうち、特定のアクションを使わないときは、resourcesメソッドにonlyオプションやexceptオプションを渡します。次の例では、MembersControllerのindexアクションとshowアクションのルーティングだけを設定します。

```
resources :members, only: [:index, :show]
```

次の例では、destroyを除く6つのアクションのルーティングを設定します。

```
resources :members, except: [:destroy]
```

リソースとパスの指定

リソースベースのルーティングを設定すると、link_toやredirect_toなどのメソッドでシンプルなパスの指定ができます。

■パスを返すメソッド

resourcesメソッドでリソースを指定すると、コントローラのアクションを表すパスを「リソース名_path」の形のメソッドで取得できるようになります。「resources :members」では、次のメソッドが使えるようになります。

アクション	パスを返すメソッド	戻り値の例
index	members_path	/members
show	member_path(member)	/members/123
new	new_member_path	/members/new
edit	edit_member_path(member)	/members/123/edit
create	members_path	/members
update	member_path(member)	/members/123
destroy	member_path(member)	/members/123

リソースの集合を扱うindexとcreateでは、members_pathのようにmemberが複数形になることに注意してください。個別のリソースを扱うshowなどでは単数形です。newは集合でも個別でもありませんが、単数形です。

パスを返すメソッドは、コントローラでもビューでも使えます。ビューの中で、会員一覧のページ（indexアクション）へのリンクは次のように作れます。

```
link_to "会員一覧", members_path
```

member_pathとedit_member_pathのように、個別のリソースを扱うアクションへのパスを得るには、引数にモデルオブジェクトを渡します。すると、モデルのidがidパラメータになります。たとえば、会員の詳細情報ページ（showアクション）へのリンクは次のようになります。

```
link_to @member.name, member_path(@member)
```

members_pathとmember_pathが返すパスは複数のアクションを表しますが、アクションの区別はHTTPメソッドで行います。たとえば、削除のためのリンクでmethodオプションにDELETEメソッドを指定すれば、destroyアクションへのリンクになります。

```
link_to "削除", member_path(@member), method: :delete
```

前述のようにsearch、suspend、restoreの各アクションを追加したときは、次のメソッドがパスを返すようになります。リソースの集合を扱うsearchアクションでは、search_members_pathのようにmemberが複数形になります。

アクション	パスを返すメソッド	戻り値の例
search	search_members_path	/members/search
suspend	suspend_member_path(member)	/members/123/suspend
restore	restore_member_path(member)	/members/123/restore

「リソース名_url」メソッド

上記の「リソース名_path」メソッドの_pathの部分を_urlに変えると、http://で始まるURLを返すメソッドになります。次のリンクは、aタグのhref属性が「href="http://localhost:3000/members"」になります（PC上で動かしている場合）。

```
link_to "会員一覧", members_url
```

■オブジェクトでパスを表す

さらに簡略化したパスの表現を見てみましょう。link_toメソッドの第2引数にモデルオブジェクトを渡すと、「/members/123」のようにmember_pathメソッドと同じパスに変換されます。

```
link_to  member.name, @member
link_to "削除", @member, method: :delete
```

editアクションやsuspendアクションのように、個別のリソースを扱うアクションは、配列を使って[:アクション名, オブジェクト]で表せます。次の例は「/members/123/edit」や「/members/123/suspend」のようなパスになります。

```
link_to "編集", [:edit, @member]
link_to "停止", [:suspend, @member], method: :patch
```

indexアクションやnewアクションのようにidパラメータを取らないアクションでは、前述の「○○_path」の「_path」を取った文字列をシンボルにしたものが使えます。

```
link_to "会員一覧", :members
link_to "新規追加", :new_member
```

以上のモデルオブジェクト、配列、シンボルで表したパスは、コントローラのredirect_toメソッドの引数としても使えます。

RESTの原則

REST（REpresentational State Transfer）を提唱したのは、HTTPの規格の執筆者の1人であり、Apacheプロジェクトの創設者であるロイ・フィールディングです。RESTとは、大まかに言えば、

機能ではなくリソースを中心にして物事を考えることです。RESTに従ったウェブアプリケーションは、次のような原則を持つとされています。

1. すべてのリソースはURLで表される一意なアドレスを持つ。
2. リソースに対する基本操作は、取得（表示）、追加（新規登録）、更新、削除の4つである。対応するHTTPのメソッドGET、POST、PATCH（Rails 3以前ではPUT）、DELETEでリクエストする。
3. クライアントもサーバーもセッションの状態を記憶する必要がない。

1.はウェブの原則とも言えるものですが、RESTではさらにこの原則を進めています。たとえば、1人の会員情報は1つのリソースなので、情報の表示、更新、削除は同じURLで表します。

2.は、現在のHTTPのメソッドの使われ方があいまいなので、メソッドの役割を明確化しようというものです。たとえば、リソースの状態を変更するのにGETを使うことは禁じられます。

3.は、リソースの表示や操作のための情報は、すべてHTTPのリクエストとレスポンスに含ませるということです。リクエストとレスポンスは毎回独立しており、前回のリクエストとレスポンスから影響を受けるべきではありません。したがって、複数のページにわたってフォームに情報を入力させるときは、前のページの入力項目をセッションに保存するのは原則に反します（フォームの隠し入力欄に入れておくのが正しいやり方です）。

5.2 7つのアクション

RailsのRESTの原則について学んだら、リソースを扱うコントローラを実際に作成してみましょう。会員のデータをリソースとして扱うコントローラを作成し、7つのアクションを書いていきます。

MembersControllerの作成

Morning Gloryのサイトであるasagaoアプリケーションに、「bin/rails g」コマンドでMembersControllerを追加します。コントローラ名は「members」と必ず複数形にしてください。

```
$ bin/rails g controller members
```

config/routes.rbにresourcesメソッドでリソースベースのルーティングを設定します。

LIST chapter05/config/routes.rb

```
1 Rails.application.routes.draw do
2   root "top#index"
3   get "about" => "top#about", as: "about"
4
5   1.upto(18) do |n|
6     get "lesson/step#{n}(/:name)" => "lesson#step#{n}"
7   end
8
```

```
  resources :members
end
```

app/controllers/members_controller.rbを開き、MembersControllerに7つのアクションを記述しましょう。筆者は、実際にアプリケーションを開発するときは、たいていアクションをこの順番で並べます。

この7つのアクションの名前は丸暗記してください。何も見ないでコントローラに7つのアクションを書けるのがRailsプログラマです。

LIST chapter05/app/controllers/members_controller.rb

```
class MembersController < ApplicationController
  def index
  end

  def show
  end

  def new
  end

  def edit
  end

  def create
  end

  def update
  end

```

```
20   def destroy
21   end
22 end
```

各アクションの中身はあとで実装することにして、とりあえず空にしておきます。

このChapterでは、indexアクションとshowアクションを実装し、それ以外の5つのアクションは、Chapter 6で実装します。ただし、indexアクションを作る際に、少し寄り道をして会員を検索するsearchアクションも作ります。

会員の一覧ページ

7つのアクションのうち、会員の一覧を表示するindexアクションとそのテンプレートを実装しましょう。同時に、会員を検索するsearchアクションも作ります。

■indexアクション

まず、ヘッダー用の部分テンプレート_header.html.erbを開き、「会員名簿」の行の"#"を:membersに変えてメニューのリンク先をindexアクションにします。

LIST chapter05/app/views/shared/_header.html.erb

```
（省略）
8     <%= menu_link_to "会員名簿", :members %>
（以下省略）
```

続いて、MembersControllerのindexアクションを実装します。2行目の記号#で始まる行にはコメントを加えています。メソッドの内部ではクエリーメソッドのorderを使ってリレーションオブジェクトをインスタンス変数@membersに取り出しています。orderメソッドによって並びは背番号（numberカラムの値）順になります。

LIST chapter05/app/controllers/members_controller.rb

```
class MembersController < ApplicationController
  # 会員一覧
  def index
    @members = Member.order("number")
  end
  （以下省略）
```

app/viewsディレクトリの下にmembersディレクトリを作成し、その中にindexアクション用のテンプレートindex.html.erbを次のように作成します。

LIST chapter05/app/views/members/index.html.erb

```
<% @page_title = "会員名簿" %>
<h1><%= @page_title %></h1>

<% if @members.present? %>
 <table class="list">
  <thead>
   <tr>
    <th>背番号</th>
    <th>ユーザー名</th>
    <th>氏名</th>
   </tr>
  </thead>
  <tbody>
   <% @members.each do |member| %>
    <tr>
     <td style="text-align: right"><%= member.number %></td>
     <td><%= member.name %></td>
     <td><%= member.full_name %></td>
```

```
19         </tr>
20       <% end %>
21     </tbody>
22   </table>
23 <% else %>
24   <p>会員情報がありません。</p>
25 <% end %>
```

1行目では、HTMLタイトルの生成に使われるインスタンス変数@page_titleをセットしています（「3.4　モックアップの作成」の「モックアップのレイアウトテンプレート」を参照）。

@members.eachのブロック（14～20行）でMemberモデルのオブジェクトmemberを取り出し、背番号、ユーザー名、氏名を表示します。

4行目の「if @members.present?」は会員一覧が空かどうか調べるもので、空の場合は「会員情報がありません。」と表示します。

まだindexアクションは完成していませんが、この段階で表示を確認してみましょう。「bin/rails s」コマンドでサーバーを起動し、ブラウザでトップページを開き、メニューの「会員名簿」をクリックすれば、会員の一覧が表示されます。

RESULT 作成途中の会員一覧ページ（indexアクション）

ブラウザのURL入力欄が「http://localhost:3000/members」になっていることも確認してください。

■スタイルの適用

前項で表示した会員一覧表にスタイルを適用するため、app/assets/stylesheetsディレクトリに新規ファイルtable.cssを追加します。

LIST chapter05/app/assets/stylesheets/table.css

```
1 /* 表：一覧表示、詳細表示 */
2 table.list, table.attr {
3   font-size: 90%;
4   width: 100%;
5 }
6
7 table.list th, table.attr th {
```

```
  background-color: #499;
  color: white;
  font-weight: normal;
}

table.list td, table.list th,
table.attr td, table.attr th {
  padding: 4px;
}

table.list th {
  text-align: left;
}

table.attr th {
  text-align: right;
}

table.list td, table.attr td {
  background-color: #cee;
}
```

ブラウザをリロードすると、次のような画面表示となります。

RESULT 作成途中の会員一覧ページ（indexアクション）

■各種リンクの設置

次に、indexアクション用のテンプレートに他のアクションへのリンクを設置していきます。まず、表の上に新規追加（newアクション）へのリンクを加えます。

LIST chapter05/app/views/members/index.html.erb

```
1 <% @page_title = "会員名簿" %>
2 <h1><%= @page_title %></h1>
3
4 <div class="toolbar"><%= link_to "会員の新規登録", :new_member %></div>
5
6 <% if @members.present? %>
```

（以下省略）

divタグで囲んでclass="toolbar"属性を加えているのは、スタイルシートapplication.cssで次のように書かれている効果をこの部分に適用するためです。すなわち、リンクの上下に15pxの隙間が追加され、フォントサイズが90%に縮小され、行全体が右寄せで表示されます。

```
ul.toolbar,
div.toolbar {
  padding: 15px 0;
  font-size: 90%;
  text-align: right;
}
```

編集（editアクション）と削除（destroyアクション）へのリンクを設置するため、表に「操作」列を追加します。

LIST chapter05/app/views/members/index.html.erb

```
（省略）
 8    <thead>
 9      <tr>
10        <th>背番号</th>
11        <th>ユーザー名</th>
12        <th>氏名</th>
13        <th>操作</th>
14      </tr>
15    </thead>
（省略）
22          <td>
23            <%= link_to "編集", [:edit, member] %> |
24            <%= link_to "削除", member, method: :delete,
```

```
25                 data: { confirm: "本当に削除しますか?" } %>
26           </td>
   （以下省略）
```

ユーザー名を表示している部分を修正して、詳細ページ（showアクション）へのリンクを設定します。

LIST chapter05/app/views/members/index.html.erb

```
   （省略）
20     <td><%= link_to member.name, member %></td>
   （以下省略）
```

ブラウザをリロードすると、次のような画面表示となります。

RESULT 完成した会員一覧ページ（indexアクション）

会員検索機能

会員の検索用にsearchアクションを追加します。まず、先ほどconfig/routes.rbに加えたresources :membersという行を次のように書き換えてください。

LIST chapter05/config/routes.rb

```
（省略）
6  resources :members do
7    get "search", on: :collection
8  end
9 end
```

コントローラの中に会員の検索を行うsearchアクションを加えます。

LIST chapter05/app/controllers/members_controller.rb

```
（省略）
7  # 検索
8  def search
9    @members = Member.search(params[:q])
10   render "index"
11 end
（以下省略）
```

ここでは、検索機能の実体をMemberクラスのsearchメソッドで実装することにしました。後述のように、パラメータの「q」には検索ワードが入ってきます。

「render "index"」によって、テンプレートはindexアクションと同じindex.html.erbでレンダリングします。

アクションはできるかぎり簡潔に

アクションの記述で肝心なのは、「長々とコードを書かない」ことです。ちょっとでもややこしくなりそうなら、モデルクラスに機能を移しましょう。

Memberクラスにクラスメソッドsearchを作成します。引数の検索ワードが空でなければ、SQLのLIKEを使ってユーザー名または氏名から検索するリレーションオブジェクトを作成します。

「class << self」という書き方については、「2.4　クラス」の「クラスメソッドの書き方」を参照してください。

LIST chapter05/app/models/member.rb

```
1 class Member < ApplicationRecord
2   class << self
3     def search(query)
4       rel = order("number")
5       if query.present?
6         rel = rel.where("name LIKE ? OR full_name LIKE ?",
7           "%#{query}%", "%#{query%")}
8       end
9       rel
10    end
11  end
12 end
```

クラスメソッドsearchの中で、ローカル変数relがおもしろい働きをしています。まず、4行目で「numberカラムでソートする」という設定を持つリレーションオブジェクトが変数relにセットされます。次に、6行目でそのリレーションオブジェクトに「nameカラムまたはfull_nameカラムを対象にレコードを絞り込む」という設定が追加されます。そして、9行目でそのリレーションオブジェクトをメソッドの戻り値として返しています。

present?メソッドとblank?メソッド

Railsアプリケーションの中では、すべてのオブジェクトに対してpresent?メソッドとblank?メソッドを呼ぶことができます。この2つのメソッドは、ユーザーが入力したデータやレコードのカラムが空かどうかを調べるのに使います。

オブジェクトがnil、false、空文字列、空白文字（改行とタブを含む）だけを含む文字列、空の配列、空のハッシュの場合は、blank?メソッドはtrueを返し、そうでなければfalseを返します。present?メソッドはその逆です。

なお、日本語で使われる全角空白も空白文字として扱われます。

テンプレートindex.html.erbのh1タグの下に検索用のフォームを設置します。

LIST chapter05/app/views/members/index.html.erb

```
1 <% @page_title = "会員名簿" %>
2 <h1><%= @page_title %></h1>
3
4 <%= form_tag :search_members, method: :get, class: "search" do %>
5   <%= text_field_tag "q", params[:q] %>
6   <%= submit_tag "検索" %>
7 <% end %>
8
9 <div class="toolbar"><%= link_to "会員の新規登録", :new_member %>
</div>
    （以下省略）
```

form_tagは一般的なフォームを生成するメソッドで、引数には送信先のパス（action属性）を指定します。form_tagはデフォルトでHTTPメソッドをPOSTにするので、methodオプションで:getを指定してGETメソッドに変えます。

form_tagメソッドのブロックにはフォームの内容を記述します。text_field_tagはテキスト入力欄（type属性がtextのinputタグ）を作るメソッドで、第1引数にname属性の値、

第2引数にvalue属性の値を指定します。submit_tagメソッドは、送信ボタン（type属性がsubmitのinputタグ）を作ります。

ブラウザをリロードすると、会員一覧表の上に検索フォームが現れます。検索ボックスに「佐藤」と入力して「検索」ボタンを押してみましょう。氏名に「佐藤」のある会員だけが表示されます。ブラウザのURL入力欄は「http://localhost:3000/members/search?...」のようになります。

RESULT 会員の検索（searchアクション）

会員の詳細ページ

会員情報の詳細、つまり1つのレコードの各カラムの値を表示するページを作りましょう。showアクションでは、findメソッドにidパラメータを渡します。findメソッドが返すモデルオブジェクトを変数@memberに入れます。

LIST chapter05/app/controllers/members_controller.rb

```
（省略）
# 会員情報の詳細
def show
  @member = Member.find(params[:id])
end
（以下省略）
```

インスタンス変数とNoMethodError

コントローラやテンプレートの中で、インスタンス変数@memberのつづりを間違えると、NoMethodError（メソッドがない）というエラーが示されます。初期化されていないインスタンス変数の値はnilになり、nilにはnumberなどのメソッドがないためです。

エラーの表示が「メソッドがない」でも、その原因はインスタンス変数のスペルミス、ということがよくあるので注意してください。

app/views/membersディレクトリの下にテンプレートshow.html.erbを作成します。HTMLのテーブル内にレコードのカラム、つまりモデルオブジェクト@memberの属性を並べます。

LIST chapter05/app/views/members/show.html.erb

```
<% @page_title = "会員の詳細" %>

<h1><%= @page_title %></h1>

<div class="toolbar"><%= link_to "編集", [:edit, @member] %></div>

<table class="attr">
  <tr>
    <th width="150">背番号</th>
```

```
    <td><%= @member.number %></td>
  </tr>
  <tr>
    <th>ユーザー名</th>
    <td><%= @member.name %></td>
  </tr>
  <tr>
    <th>氏名</th>
    <td><%= @member.full_name %></td>
  </tr>
  <tr>
    <th>性別</th>
    <td><%= @member.sex == 1 ? "男" : "女" %></td>
  </tr>
  <tr>
    <th>誕生日</th>
    <td><%= @member.birthday&.strftime("%Y年%m月%d日") %>
</td>
  </tr>
  <tr>
    <th>メールアドレス</th>
    <td><%= @member.email %></td>
  </tr>
  <tr>
    <th>管理者</th>
    <td><%= @member.administrator? ? "○" : "－" %></td>
  </tr>
</table>
```

先ほど作った会員一覧のページで、ユーザー名の「Taro」や「Hana」をクリックすると、その会員の情報が表示されます。ブラウザのURL欄が「http://localhost:3000/members/1」や「http://localhost:3000/members/3」のようになっていることも確認してください。

RESULT 会員の詳細ページ（showアクション）

&.演算子

show.html.erbで誕生日を表示するところでは、birthday.strftime("%Y年%m月%d日")とせずに、birthday&.strftime("%Y年%m月%d日")のように&.演算子（通称「ぼっち演算子」）を使っています。これは、birthdayがnilの場合に備えた書き方です。そのままstrftimeメソッドを呼び出すと、誕生日がない場合には例外NoMethodErrorが発生します。&.演算子を使えば、簡潔な書き方で例外の発生を防げます。

&.演算子は左辺のオブジェクトがnilのときnilを返し、nilでないときは右辺に書かれたメソッドを呼び出してその戻り値を返します。

属性?メソッド

show.html.erbの一番下で使っている@member.administrator?という?付きのメソッドは、administrator属性がtrueかfalseかを調べるものです。@member.administratorとしても同じですが、?を付けることでtrueかfalseかを調べていることを明示できます。

カラム名に?を加えた名前を持つメソッドは、テーブルにそのカラムがあれば使えるようになるメソッドです。Memberモデルの場合は、number?メソッドやname?メソッドも使えます。値がnil、false、空文字列、空白文字だけからなる文字列の場合はfalseを返し、そうでなければtrueを返します。present?メソッドとは違い、値が数値0の場合はfalseを返します。

Chapter 5のまとめ

- RESTとは、リソースという概念を中心にしてウェブアプリケーションを組み立てる考え方です。Railsのリソースは、コントローラが扱う対象に名前を付けたものです。
- リソース名は、routes.rbでresourcesメソッドを使って設定します。リソース名を設定すると、リソースを扱うコントローラに対して、リソースベースのルーティングが作られます。
- リソースベースのルーティングでは、index、show、new、edit、create、update、destroyの7つのアクションが基本となります。
- リソース名を設定すると、「○○_path」という形のメソッドでURLのパスを生成できるようになります。
- リソースの一覧ページはindexアクション、詳細情報のページはshowアクションで実装します。

練習問題

［A］　Chapter 4の練習問題で作ったBookモデルを扱うBooksControllerを作ることにします。routes.rbで「resources :books」と指定すると、BooksControllerの各アクションとURLのパス、HTTPメソッドはどのような関係になるでしょうか。空欄を埋めてください。idパラメータには1が入ることにします。

アクション、URLのパスとHTTPメソッドの対応

index	/books	GET
show		GET
new	/books/new	GET
edit	/books/1/edit	GET
	/books	POST
update	/books/1	
destroy	/books/1	DELETE

［B］　問題［A］のBooksControllerを記述します。indexアクションにはレコードの一覧を書籍名（title）順に取り出すコードを記述してください。showアクションにはidパラメータを元にレコードを1つ取り出すコードを記述してください。

```
class BooksController < ApplicationController
  def index
    @books =
  end

  def show
    @book =
```

```
  end
end
```

［C］　問題［B］のindexアクションとshowアクション用のテンプレートを記述します。空欄を埋めて、index.html.erbではループの中にshowアクションへのリンクを作成してください。show.html.erbでは変数@bookが持っている書籍名（title）、著者名（author）、価格（price）を表示してください。

index.html.erb

```
<ul>
  <% books.each do |book| %>
    <li><%=                    %></li>
  <% end %>
</ul>
```

show.html.erb

```
<p>書籍名：<%=                    %>、
著者名：<%=                    %>、
価格：<%=                    %>円</p>
```

Chapter

6　リソースの作成と更新

Chapter 5に引き続き、リソースを扱うコントローラを実装していきましょう。会員情報を表示するだけでなく、新規作成、更新、削除の機能をアプリケーションに追加します。

これから学ぶこと

- レコードの新規作成や更新を行うフォームをテンプレートに記述する方法を学びます。
- フォームから送信されたデータを元にして、レコードの新規作成や更新を行う方法を学びます。また、レコードを削除する方法も学びます。

ブラウザからデータベースを操作するには、項目を入力できるフォームを用意する必要があります。モデルと結び付いたフォームはどのように書けばよいでしょうか？ フォームから送信された値を保存するには何をすればよいでしょうか？

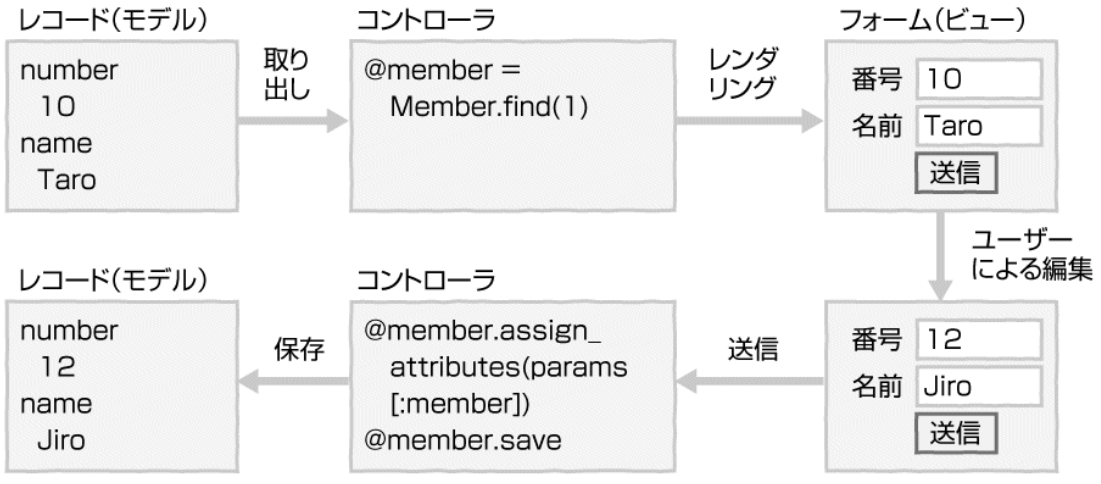
6.1 フォームとモデル

ウェブアプリケーションでは、HTMLのフォームを通じてデータベースのデータを作成したり変更したりする処理がよく行われます。Railsには、そうした処理のためにモデルとHTMLのフォームを簡単に連携させる機能が備わっています。

モデルとフォームの連携

データベースからデータを取り出し、フォームに表示して、ユーザーの操作によって情報を更新する、といった一連の流れは、Railsでは次の図のようになります。コントローラではモデルのオブジェクトをテンプレートに渡し、テンプレートではそのオブジェクトを使ってHTMLのフォームを作成します。フォームからデータが送信されたときは、送信されたデータをモデルに渡せばデータベースに保存できます。

新しいレコードを作成するときも、レコードを取り出す手順がないだけで、基本的な手順は変わりません。作成の場合は、コントローラで新しいオブジェクトを作ってテンプレートに渡すことになります。



モデルとフォームの連携

フォームから送信されたデータは、コントローラでparamsメソッドを使って取り出します。Memberモデルのフォームからデータを送信すると、paramsが返すハッシュの中身は次のようになります（updateアクションの場合）。

```
{ "utf8" => "✓", "_method" => "patch",
  "authenticity_token" =>

"WZA1imwTexQ4hH24JhU06m9qsrBhQq7pkE5PZjReUo2h/SeHGGR3lv4m...
VfbJMwFP4tqNwg==",
  "member" => { "number" => "10", "name" => "Taro",
    "full_name" => "佐藤 太郎", "sex" => "1",
    "birthday(1i)" => "1981", "birthday(2i)" => "12",
    "birthday(3i)" => "1",
    "email" => "Taro@example.com", "administrator" => "1" },
  "commit" => "更新", "action" => "update", "controller" => "members",
"id" => "1" }
```

ハッシュが入れ子になっている点に注目してください。:memberをキーにしてparams[:member]を取り出すと、それもハッシュになっています。このparams[:member]をモデルオブジェクトのassign_attributesメソッドに渡せば、ハッシュの内容がモデルの属性にセットされます。

```
@member.assign_attributes(params[:member]) # 属性をセット
@member.save                               # レコードを保存
```

フォームの記述

Chapter 5のようにリソースベースのルーティングを設定したときは、テンプレートでフォームを記述するのにform_forメソッドを使います。

■newアクションとeditアクション

まず、MembersControllerのnewアクションとeditアクションを記述しましょう。newアクションでは、Member.newで新しいモデルオブジェクトを作成します。newメソッドの引数には、誕生日の初期値を指定して1980年1月1日としています（これは好きな日付でかまいません）。

editアクションでは、findメソッドにidパラメータを渡してモデルオブジェクトを取り出します。

LIST chapter06/app/controllers/members_controller.rb

```
（省略）
18  # 新規作成フォーム
19  def new
20    @member = Member.new(birthday: Date.new(1980, 1, 1))
21  end
22
23  # 更新フォーム
24  def edit
25    @member = Member.find(params[:id])
26  end
（以下省略）
```

■新規作成フォーム

app/views/membersディレクトリの下に、テンプレートnew.html.erbを作成しましょう。最初は次のように記述してください。

LIST chapter06/app/views/members/new.html.erb

```
1 <%= form_for @member do |form| %>
2 <% end %>
```

form_forメソッドにはnewアクションで作ったモデルオブジェクトを渡します。form_forのブロックの中にフォームの内容を記述しますが、とりあえず空にします。ブロックパラメータのformには、ActionView::Helpers::FormBuilderクラスのオブジェクト（以下、**フォームビルダー**と呼びます）が渡されます。

ターミナルで「bin/rails s」コマンドを実行してサーバーを起動し、トップページから［会員名簿］→［会員の新規登録］をクリックしてください。ブラウザでHTMLのソースを表示すると、次のフォームができています。action属性は"/members"、method属性は"post"なので、createアクションを呼び出すフォームになります。

```
<form class="new_member" id="new_member" action="/members"
accept-charset
="UTF-8" method="post">
<input name="utf8" type="hidden" value="&#x2713;" />
<input name="authenticity_token" type="hidden"
value="Ec0h6jYlRunWTmI6/YY
Ou+Kc1dInbP1GFIXDy0d+2svpoDPnQlJKaxDsCAF0+whsuyHBloxQBllN+I3i
kfoFhA==" />
</form>
```

type属性がhiddenのinputタグ（表示されない入力欄）が2つ作られます。name="utf8"は古いInternet Explorerにパラメータを UTF-8で送信させるためのダミーパラメータです。name="authenticity_token"については、HINT「RailsのCSRF対策」を参照してください。

RailsのCSRF対策

CSRFは、クロス・サイト・リクエスト・フォージェリ（Cross-site request forgery）の略で、ウェブサイトへの攻撃手法の1つです。攻撃対象のウェブサイトに送信を行うフォームやリンクなどを別のサイトに用意し、ログイン中のユーザーにうっかりクリックさせる仕掛けです。勝手にブログへの投稿を行わせたり、ユーザーを退会させたりすることができてしまいます。

RailsはCSRF対策として、上記のform_forの例のようにHTTPメソッドがGET以外のフォームやリンクでは、authenticity_tokenの文字列（value="Ec0h6jYl..."）を埋め込みます。この文字列はRailsがユーザーのセッションごとにユーザー別に用意するもので、アクションの実行の前にチェックされ、文字列が不正の場合は例外が発生します。

■更新フォーム

app/views/membersディレクトリの下に、テンプレートedit.html.erbを作成し、new.html.erbと同じコードを記述してみましょう。

LIST chapter06/app/views/members/edit.html.erb

```
1 <%= form_for @member do |form| %>
2 <% end %>
```

会員の一覧ページで［編集］をクリックし、ブラウザでHTMLのソースを表示すると、次のフォームができています。

```
<form class="edit_member" id="edit_member_1" action="/members/1"
accept-c
harset="UTF-8" method="post">
<input name="utf8" type="hidden" value="&#x2713;" />
<input name="_method" type="hidden" value="patch" />
<input name="authenticity_token" type="hidden"
value="h/+4XGGC1iycQNs09Ib
Jh9pn+ZDtckDtpgcwCsmyFvR/kqpRFfXarlrisQ99+89Qg9rt1EZOu/L/en4jHz
```

```
bJuw==" />
</form>
```

action属性は"/members/1"、method属性は"post"です。表示されない入力欄にname属性が"_method"、value属性が"patch"のものがあります。フォームが送信されると、「"_method" => "patch"」というパラメータが送られるので、RailsはHTTPメソッドがPATCHであると判断し、updateアクションを呼び出します。

form_forメソッドは、引数のモデルオブジェクトを調べ、保存されていなければcreate用のフォームを作成し、保存済みならupdate用のフォームを作成します。

form_forのオプション

フォームの送信先のパスとHTTPメソッドを自分で指定したいときは、urlオプションとmethodオプションを指定します。

```
form_for @member, url: member_path(@member), method: :patch
```

パラメータ名を変えたいときは、asオプションを使います。次の例ではフォームの入力欄のname属性はname="user[number]"のようになり、コントローラではparams[:user]でデータが取り出せるようになります。

```
form_for @member, as: "user"
```

formタグにclass属性やid属性を指定したいときは、htmlオプションにハッシュを指定します。

```
form_for @member, html: { class: "member", id: "main_form" }
```

フォームの部品の記述

form_forメソッドのブロックパラメータ（|form|）に渡されるフォームビルダーは、さまざまなフォームの部品を作るメソッドを備えています。たとえば、1行テキスト入力欄（<input type="text"/>）を作成するには、text_fieldメソッドを使います。引数にモデルの属性名（テーブルのカラム名）を指定すれば、その属性を編集する入力欄になります。

```
<%= form_for @member do |form| %>
背番号：<%= form.text_field :number %>
<% end %>
```

すると、次のようなinputタグができます。

```
背番号：<input id="member_name" name="member[number]" size="30"
 type="text" value="10" />
```

@memberにデータが入っているときは、value="10"のように属性の値がセットされます。name属性はname="member[number]"のように「モデル名[属性名]」の形になります。

フォームが送信されると、この入力欄のデータは「"member" => { "number" => "10" }」のような入れ子のハッシュになります。

フォームビルダーのメソッドの詳細を見る前に、newアクションとeditアクションのテンプレートを作成して感じをつかんでみましょう。

たいていの場合、newアクションとeditアクションは同じ入力欄を使いますので、部分テンプレートで共有します。app/views/membersの下に_form.html.erbを作成し、次のように記述してください。HTMLのテーブルを使って属性名とフォームの部品を並べます。

LIST chapter06/app/views/members/_form.html.erb

```
1 <table class="attr">
2   <tr>
3     <th><%= form.label :number, "背番号" %></th>
4     <td><%= form.text_field :number, size: 8 %></td>
```

```
</tr>
<tr>
  <th><%= form.label :name, "ユーザー名" %></th>
  <td><%= form.text_field :name %></td>
</tr>
<tr>
  <th><%= form.label :full_name, "氏名" %></th>
  <td><%= form.text_field :full_name %></td>
</tr>
<tr>
  <th>性別</th>
  <td>
    <%= form.radio_button :sex, 1 %>
    <%= form.label :sex_1, "男" %>
    <%= form.radio_button :sex, 2 %>
    <%= form.label :sex_2, "女" %>
  </td>
</tr>
<tr>
  <th><%= form.label :birthday, "誕生日",
          for: "member_birthday_1i" %></th>
  <td><%= form.date_select :birthday,
          start_year: 1940, end_year: Time.current.year,
          use_month_numbers: true %></td>
</tr>
<tr>
  <th><%= form.label :email, "メールアドレス" %></th>
  <td><%= form.text_field :email %></td>
</tr>
```

```
34   <tr>
35     <th>管理者</th>
36     <td>
37       <%= form.check_box :administrator %>
38       <%= form.label :administrator, "管理者" %>
39     </td>
40   </tr>
41 </table>
```

一般メンバーがメンバー全員の管理者フラグを書き換えられるという不自然な仕様については、とりあえず気にしないでください。Chapter 15で本格的な管理ページを作る際に仕様を考え直しましょう。

続いて、new.html.erbを次のように書き換えてください。

LIST chapter06/app/views/members/new.html.erb

```
1 <% @page_title = "会員の新規登録" %>
2
3 <h1><%= @page_title %></h1>
4
5 <%= form_for @member do |form| %>
6   <%= render "form", form: form %>
7   <div><%= form.submit %></div>
8 <% end %>
```

form_forメソッドのブロック内にrenderメソッドで部分テンプレート_form.html.erbを埋め込みます。また、フォームビルダーのsubmitメソッドで送信ボタンを作成します。

部分テンプレートにローカル変数を渡す

上記のrenderメソッドには「form: form」というオプションを渡しています。これは、部分テンプレートに変数formを渡す指定です。テンプレートの間では、インスタンス変数を共有できますが、ローカル変数は共有できません。部分テンプレートの中で親テンプレートのローカル変数を使いたいときは、「X: Y」という形のオプションをrenderメソッドに加えます。親テンプレートのローカル変数をYに指定すると、部分テンプレートの中でXという名前のローカル変数が使えるようになります。

同様にedit.html.erbにも_form.html.erbを埋め込みます。

LIST chapter06/app/views/members/edit.html.erb

```
1 <% @page_title = "会員情報の編集" %>
2
3 <h1><%= @page_title %></h1>
4
5 <div class="toolbar"><%= link_to "会員の詳細に戻る", @member %>
</div>
6
7 <%= form_for @member do |form| %>
8   <%= render "form", form: form %>
9   <div><%= form.submit %></div>
10 <% end %>
```

会員一覧ページから「会員の新規登録」をクリックして、newアクション用のフォームを表示してみましょう。入力欄は空の状態です。

RESULT newアクションのページ

会員一覧ページから会員の名前の右にある［編集］をクリックして、editアクション用のフォームを表示してみましょう。editアクションではモデルオブジェクトにMemberモデルの属性が入っているので、入力欄は値が最初から入っている状態になります。

RESULT editアクションのページ

フォームビルダーのメソッド

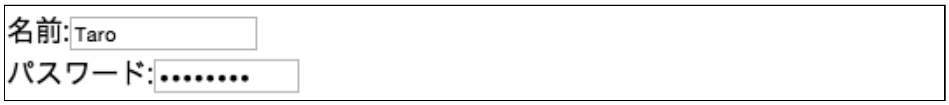
フォームビルダーが用意しているメソッドを紹介しましょう。さまざまなフォームの部品を作ることができます。どのメソッドでも、第1引数にはモデルの属性名を指定します。また、たいていのメソッドでは引数の最後にハッシュのオプションでタグの属性を並べることができます。

■1行テキスト、パスワード、隠し項目

1行のテキスト入力欄（<input type="text" />）は、text_fieldメソッドで記述します。オプションにはsizeなどの属性を指定できます。password_fieldメソッドを使うとパスワード入力欄（<input type="password" />）になり、hidden_fieldメソッドを使うと表示されない欄（<input type="hidden" />）になります。

```
名前：<%= form.text_field :name, size: 16 %><br />
パスワード：<%= form.password_field :password, size: 12 %>
```

```
<%= form.hidden_field :some_value %>
```

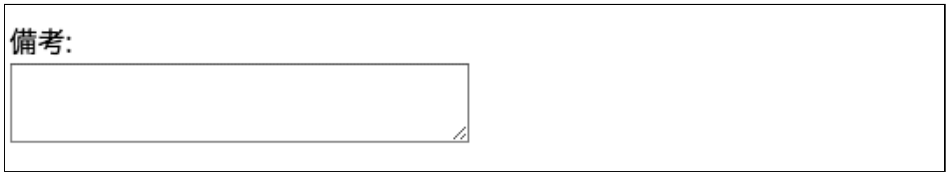


1行テキスト入力欄

■複数行テキスト

複数行のテキスト入力欄（textareaタグ）は、text_areaメソッドで記述します。幅（cols属性）と高さ（rows属性）は、オプションで指定します。

```
備考：<br />
<%= form.text_area :remarks, cols: 40, rows: 3 %>
```



複数行テキスト入力欄

■チェックボックス

チェックボックス（<input type="checkbox" />）は、check_boxメソッドで記述します。第2引数にはHTMLの属性をハッシュで指定できます。チェックボックスの値（value属性）を設定したいときは、第3引数と第4引数にオンの場合とオフの場合の値を並べます。

```
<%= form.check_box :administrator, {}, "on", "off" %> 管理者
```

チェックボックス

value属性の値を設定しないと、自動的にオンの値は「1」、オフの値は「0」となります。フォームデータをモデルオブジェクトに入れると、「1」がtrue、「0」がfalseになります。

```
<%= form.check_box :administrator %> 管理者
```

最初からチェックが付いた状態にしたいときは、「@member.administrator = true」のようにコントローラでモデルオブジェクトに値を入れておきます。

■ラジオボタン

ラジオボタン（<input type="radio" />）は、radio_buttonメソッドで記述します。第2引数には、ラジオボタンの値（value属性）を指定します。最初からチェックが付いた状態にしたいときは、「@member.sex = 2」のようにコントローラで値を入れておきます。

```
<%= form.radio_button :sex, 1 %> 男
<%= form.radio_button :sex, 2 %> 女
```

ラジオボタン

■選択リスト

プルダウンメニューによる選択リスト（selectタグとoptionタグの組み合わせ）は、selectメソッドで記述します。第2引数には、選択肢を配列で指定します。

```
<% options = ["本町", "東町", "南町", "北町", "その他"] %>
地域：<%= form.select :area, options %>
```

選択リスト

上記の例では、<option value="本町">本町</option>のようなタグを生成します。<option value="1">本町</option>のようにvalue属性の値を指定したいときは、次のように配列の配列で文字列と値を並べます。

```
<%
options = [
  ["本町", 1],
  ["東町", 2],
  ["南町", 3],
  ["北町", 4],
  ["その他", 5]
]
%>
地域：<%= form.select :area, options %>
```

最初から選択肢のどれかが選択された状態にしたいときは、「@member.area = "東町"」や「@member.area = 3」のようにコントローラで値を入れておきます。

■日付と時刻の選択

日付や時刻を選択できるように、自動的に複数のリストを並べる機能もあります。日付の選択にはdate_selectメソッド、日付と時刻の選択にはdatetime_selectメソッドを使います。この2つのメソッドを用いると、送信されたデータをモデルオブジェクトにそのまま格納するだけで、自動的に日付型や日時型の値として保存できます。

オプションとして最初の年（start_year）、最後の年（end_year）、月の表示を数字にするかどうか（use_month_numbers）を指定できます。

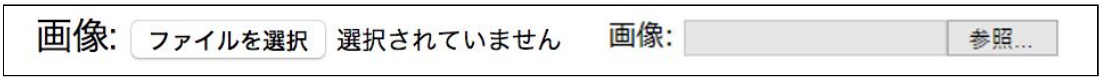
```
誕生日：<%= form.date_select :birthday,
            start_year: 1940, end_year: Time.current.year,
            use_month_numbers: true %>
```

日付の選択

■ファイルのアップロード

ファイルのアップロード用の部品（<input type="file" />）は、file_fieldメソッドで記述します。ファイルのアップロードの実際の例は、Chapter 13を参照してください。

```
画像：<%= form.file_field :uploaded_file %>
```



ファイルのアップロード

この部品はブラウザによって表示が違います。左はMacのChromeのもの、右はWindowsのMicrosoft Edgeのものです。

■送信ボタン

フォームの送信ボタン（<input type="submit" />）は、submitメソッドで作ります。ボタンの上の文字を変えたいときは、引数に指定します。

```
<%= form.submit "更新" %>
```

送信ボタン

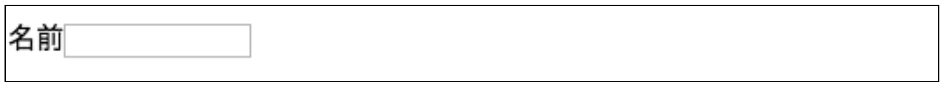
一般的なボタン（buttonタグ）を作るにはbuttonメソッドを利用します。引数でボタンの上の文字を指定し、typeオプションでtype属性の値を指定します。type属性の値は"submit"、"reset"、"button"のいずれかで、デフォルトの値は"submit"です。

```
<%= form.button "Click Me!", type: "button" %>
```

■ラベル

入力欄に対応するラベル（labelタグ）を作成するには、labelメソッドを使います。引数には、対応する入力欄の属性名を指定します。第2引数にはラベルのテキストを指定できます。

```
<%= form.label :name, "名前" %> <%= form.text_field :name, size: 16 %>
```



ラベル

第2引数を省略するとロケールテキストに記述した属性名が使われます（「7.2　メッセージの日本語化」を参照）。7.2節では、モデルの属性名に日本語の名前を付けて、会員情報フォームのlabelメソッドを修正します。

日付とラジオボタンのラベル

text_fieldメソッドの引数を:nameとすると、inputタグのid属性がid="member_name"となります。labelメソッドの引数を:nameとすると、labelタグのfor属性がfor="member_name"となる

ので、ラベルと入力欄を関連付けられます。先に記述したテンプレート_form.html.erbでは、こうしたケースから外れる入力欄があるので、工夫を加えています。

誕生日用のdate_selectメソッドは、3つのselectタグになります。labelメソッドにはオプション「for: "member_birthday_1i"」を付けてfor属性の値を変え、1番目のselectタグを指すようにしています。

性別用のラジオボタンは、id属性がid="member_sex_1"とid="member_sex_2"の2つのinputタグになります。labelメソッドの引数を:sex_1、:sex_2とすることで、for属性がfor="member_sex_1"、for="member_sex_2"となるようにしています。

scaffoldの利用

Railsには、scaffold（「足場」の意）というモデルとコントローラを簡単に作成するためのしくみが用意されています。

次のコマンドを実行すると、Memberモデルとマイグレーションスクリプト、MembersControllerとビューが一気にできあがります。「number:integer name:string」によって、マイグレーションスクリプトには整数型のnumberと文字列型のnameカラムが自動的に追加されます。

```
$ bin/rails g scaffold Member number:integer name:string
```

MembersControllerには7つのアクションが自動的に記述され、アクション用のテンプレートも用意されます。あとは、「bin/rails db:migrate」と「bin/rails s」で実際にアプリケーションが動きます。

scaffoldが作成したコードを編集しながら開発を行うこともできます。ただし、結局は自分の手でモデルやコントローラ、ビューを大幅に書き直すことになるので、実用的なサイトを作成するにはそれなりの手間がかかります。本書ではscaffoldを利用しません。

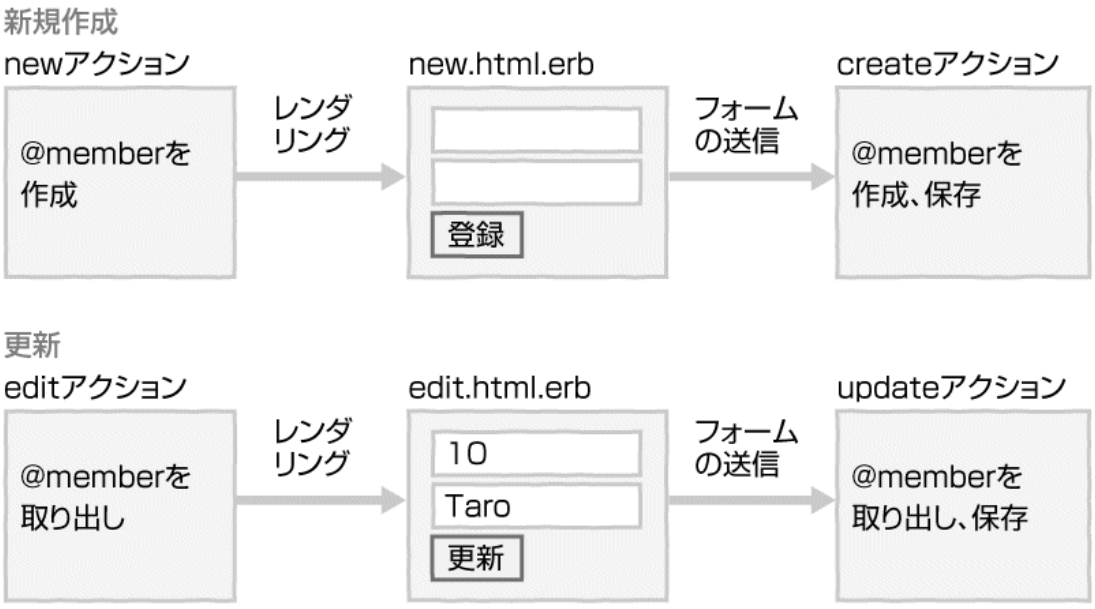
6.2 レコードの作成、更新、削除

ここでは、create、update、destroyの各アクションを実装し、レコードの作成、更新、削除を行います。「4.3　データの保存」で紹介したモデルのメソッドを見返しながら機能を確認してください。

作成と更新の流れ

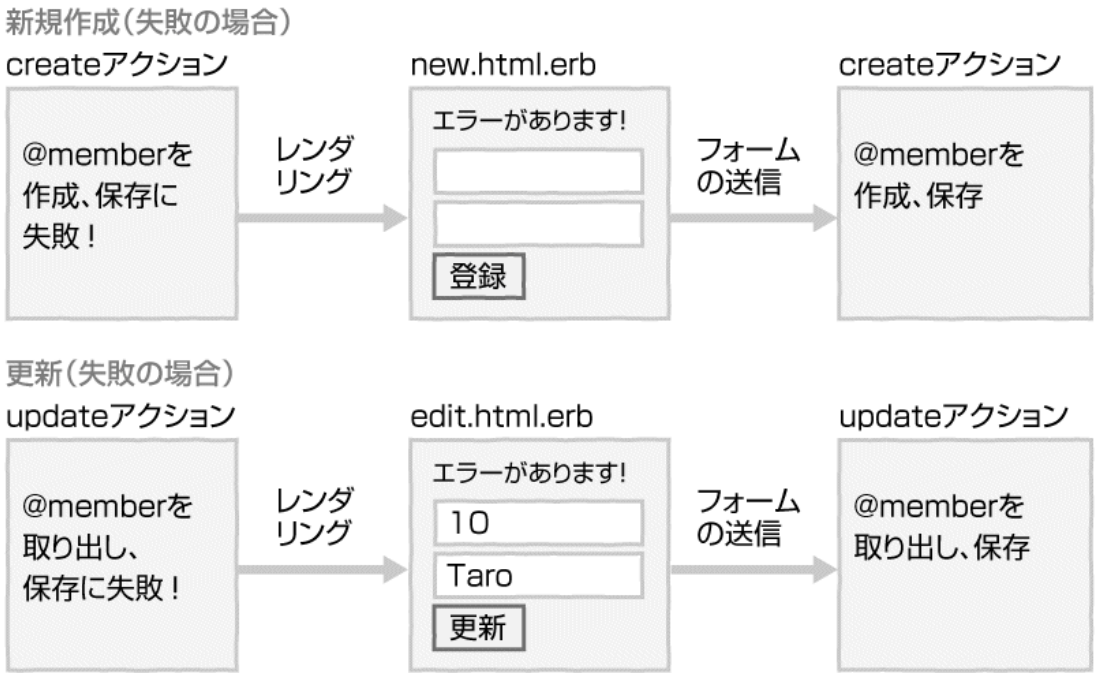
データベースの基本操作を行うには、Chapter 5で紹介した7つのアクションを使います。このうち、indexアクションとshowアクションはすでに実装しました。

データベースのレコードの作成と更新には、フォームを表示するアクションとレコードの保存を行うアクションを組み合わせます。作成では、newアクションのフォームからcreateアクションを呼び出します。更新では、editアクションのフォームからupdateアクションを呼び出します。



作成と更新のためのアクション

Chapter 7で解説するバリデーション（値の検証）によってエラーが発生すると、レコードの保存が失敗します。失敗したときは、createアクションはnewアクションのテンプレート、updateアクションはeditアクションのテンプレートを使ってエラーメッセージを表示します。エラーメッセージを表示したフォームからは、再びcreateアクションやupdateアクションを呼び出せます。



エラーが発生した場合

会員の新規登録と更新

createアクションとupdateアクションを実装して、会員の新規登録と更新の機能を追加しましょう。

■ストロング・パラメータの無効化

アクションの実装を始める前に、config/application.rbを次のように変更します（35行目を挿入）。

LIST chapter06/config/application.rb

```
（省略）
    config.time_zone = "Tokyo"

    config.action_controller.permit_all_parameters = true
  end
end
```

この設定により、「ストロング・パラメータ」と呼ばれるセキュリティ機能が無効になります。この設定をしないと、本章のサンプルコードは正常に動作しません。config/application.rbの設定変更後は、アプリケーションの再起動が必要です。Ctrl-Cでasagaoを停止し、「bin/rails s」でサーバーを起動してください。

ストロング・パラメータはあとで必ず有効化する

ストロング・パラメータは、ブラウザから送信されるパラメータに制限をかけて、セキュリティを向上させる機能です。ウェブサイトを一般に公開するときには必ず有効化してください。しかし、Rails初学者の多くはストロング・パラメータの機能を少し難しいと感じるようです。そこで、本書ではPart 4に入るまで無効化した状態で開発を進めることにしました。

■createアクション

MembersControllerのcreateアクションの内容を次のように記述します。

LIST chapter06/app/controllers/members_controller.rb

```
（省略）
  # 会員の新規登録
  def create
    @member = Member.new(params[:member])
    if @member.save
```

```
32      redirect_to @member, notice: "会員を登録しました。"
33    else
34      render "new"
35    end
36  end
    （以下省略）
```

30行目のnewメソッドで、フォームから送られたパラメータを使ってモデルオブジェクト@memberを作成します。31行目のsaveメソッドでデータベースにレコードを保存します。

保存に成功したときは、saveメソッドがtrueを返します。成功したときは、redirect_toメソッドでオブジェクト@memberが表すパスにリダイレクトします。@memberのidが123であれば、「/members/123」にリダイレクトします。

「notice: "会員を登録しました。"」は、フラッシュに値を設定するオプションです。「3.2　コントローラとアクション」の「フラッシュ」を参照してください。

エラーによってsaveメソッドがfalseを返したときは、renderメソッドでnewアクションのテンプレートを表示します。

フラッシュの文字列を表示する部分をまだ作っていませんでした。レイアウトテンプレートapplication.html.erbを次のように修正します。

LIST chapter06/app/views/layouts/application.html.erb

```
    （省略）
17    <main>
18      <% if flash.notice %>
19        <p class="notice"><%= flash.notice %></p>
20      <% end %>
21
22      <%= yield %>
23    </main>
    （以下省略）
```

さらに、フラッシュ表示部分に枠線と背景色を付けるため、app/assets/stylesheetsディレクトリにCSSファイルflash.cssを次の内容で作成します。

LIST chapter06/app/assets/stylesheets/flash.css

```
/* フラッシュ */
p.notice {
  border: 1px solid blue;
  padding: 3px;
  background-color: #ccf;
}
```

ブラウザで会員一覧ページを開き、［会員の新規登録］をクリックしてください。フォームの項目をすべて埋めて［Create Member］ボタンを押すと、createアクションが呼び出され、レコードが保存されて、showアクションにリダイレクトします。

RESULT createアクションの結果

■updateアクション

MembersControllerのupdateアクションの内容は次のように記述します。

LIST chapter06/app/controllers/members_controller.rb

```
（省略）
38  # 会員情報の更新
39  def update
40    @member = Member.find(params[:id])
41    @member.assign_attributes(params[:member])
42    if @member.save
43      redirect_to @member, notice: "会員情報を更新しました。"
44    else
45      render "edit"
46    end
47  end
（以下省略）
```

findメソッドでレコードを取り出し、assign_attributesメソッドでフォームからのデータをセットします。

createアクションと同様に、saveメソッドでレコードを保存します。保存に成功したらshowアクションにリダイレクトし、失敗したらeditアクションのテンプレートを表示します。

会員一覧ページで、名前の横の［編集］をクリックしてください。たとえば背番号を「30」に変えて［Update Member］ボタンを押してみましょう。updateアクションが呼び出され、レコードが変更されて、showアクションにリダイレクトします。

RESULT updateアクションの結果

createアクションとupdateアクションの内容は、どんなモデルでも上記の書き方が基本です。これを「型」として暗記して、すらすら書けるようになってください。

会員の削除

会員を削除するdestroyアクションを実装しましょう。findメソッドでレコードを取り出し、destroyメソッドで削除します。フラッシュにメッセージを入れて、indexアクションにリダイレクトします。

LIST chapter06/app/controllers/members_controller.rb

```
（省略）
49  # 会員の削除
50  def destroy
51    @member = Member.find(params[:id])
52    @member.destroy
```

```
53    redirect_to :members, notice: "会員を削除しました。"
54  end
    （以下省略）
```

Chapter 5で作ったindexアクションのテンプレートの中で、destroyアクションへのリンクを作っている部分を確認しましょう。

LIST chapter06/app/views/members/index.html.erb

```
    （省略）
26    <%= link_to "削除", member, method: :delete,
27          data: { confirm: "本当に削除しますか?" } %>
    （以下省略）
```

これにより、次のようなHTMLが生成されます。

```
<a href="/members/10" data-confirm="本当に削除しますか?"
      data-method="delete" rel="nofollow">削除</a>
```

ここでは、rails-ujsと呼ばれるJavaScriptライブラリが重要な働きをします。ユーザーがdata-method属性が付いているリンクをクリックすると、rails-ujsは見えないフォームをページの中に埋め込み、そのデータをDELETEメソッドで送信します（実際はPOSTメソッドですが、_methodパラメータによる擬似的なHTTPメソッドです）。また、data-confirm属性の付いているリンクは、JavaScriptによってメッセージが表示されます。

csrf_meta_tagsメソッド

app/views/layoutsディレクトリにあるレイアウトテンプレートapplication.html.erbには、headタグ内に次の記述があります。link_toメソッドのmethodオプションが正しく機能するためには、この2行が不可欠です。

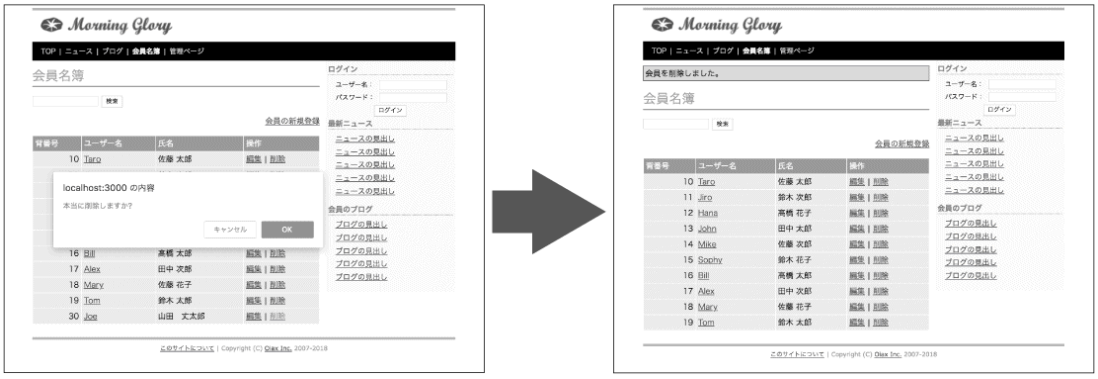
```
    <%= csrf_meta_tags %>
(省略)
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
```

　csrf_meta_tagsメソッドは、CSRF対策のauthenticity_tokenをHTMLに埋め込むものです。methodオプション付きでlink_toメソッドが呼び出されると、rails-ujsはこのauthenticity_tokenを使って見えないフォームを作成します。

　javascript_include_tagメソッドは、app/assets/javascriptsディレクトリにあるJavaScriptプログラムを読み込むタグを生成します。このディレクトリにあるapplication.jsの13行目に次のような記述があり、この結果JavaScriptライブラリrails-ujsが取り込まれます。行頭の記号//=についてはChapter 12で解説します。

```
//= require rails-ujs
```

　会員一覧ページで、名前の横の［削除］をクリックしてみましょう。レコードが削除されて、indexアクションにリダイレクトします。



RESULT destroyアクションの結果

　ブラウザ上で開発用のデータベースを変更したときは、bin/rails db:rebuildコマンドを実行すれば、いつでもデータを元に戻せます（「4.3　データの保存」の「シードデータの投入」を参照）。

Chapter 6のまとめ

- レコードの新規作成のためには、newアクションでフォームを表示し、createアクションにフォームの値を送信します。更新のためにはeditアクションでフォームを表示し、updateアクションにフォームの値を送信します。
- テーブルのデータと結び付いたフォームを作るには、テンプレートにform_forメソッドを記述します。
- form_forメソッドのブロックには、text_fieldやcheck_boxといったフォームビルダーオブジェクトのメソッドを使って入力欄を記述します。
- レコードを削除するときは、destroyアクションでモデルオブジェクトのdestroyメソッドを呼び出します。

練習問題

［A］　Chapter 4の練習問題で作成したbooksテーブルを更新するためのフォームを作成します。入力欄を表示させるメソッドの呼び出しを空欄に記入してください。

```
<%= form_for @book do |form| %>
  <div>書名：<%=                    %></div>
  <div>著者：<%=                    %></div>
  <div>価格：<%=                    %></div>
  <div><%= form.submit %></div>
<% end %>
```

［B］　フォームから送られた値を元にbooksテーブルのレコードを新規作成するcreateメソッドと、レコードを更新するupdateメソッドを作成します。空欄を埋めてメソッドを完成させてください。

```
class BooksController < ApplicationController
  def create
    @book =
    @book.save
    redirect_to @book, notice: "作成しました。"
  end

  def update
    @book = Book.find(params[:id])

    @book.save
    redirect_to @book, notice: "更新しました。"
  end
end
```

Part 3

Ruby on Railsの応用

このPartでは、実用的なウェブアプリケーションに不可欠なさまざまな機能をRuby on Railsで実装する方法について学んでいきます。例えば、バリデーション、ログイン・ログアウト、モデルクラスのコールバック、モデル間の1対多の関連付けなどです。

Chapter

7　バリデーションと国際化

Rails用語のバリデーションとは、モデルオブジェクトの各属性の値が妥当・有効（valid）であることを確かめることです。データベースに不正な形式の文字列や範囲外の数値が保存されないようにするための、非常に重要なプロセスです。

これから学ぶこと

- フォームから送信されたデータをチェックするために、バリデーション（値の検証）のしくみを学びます。
- バリデーションによって発生したエラーを表示する方法を学びます。
- Railsの国際化機能について学び、エラーメッセージを日本語化します。

Railsではどのようにしてバリデーション（値の検証）を行うのでしょうか？ どのようなタイプのバリデーションがあるのでしょうか？ どのように日本語でエラーメッセージをウェブページ上に出力するのでしょうか？

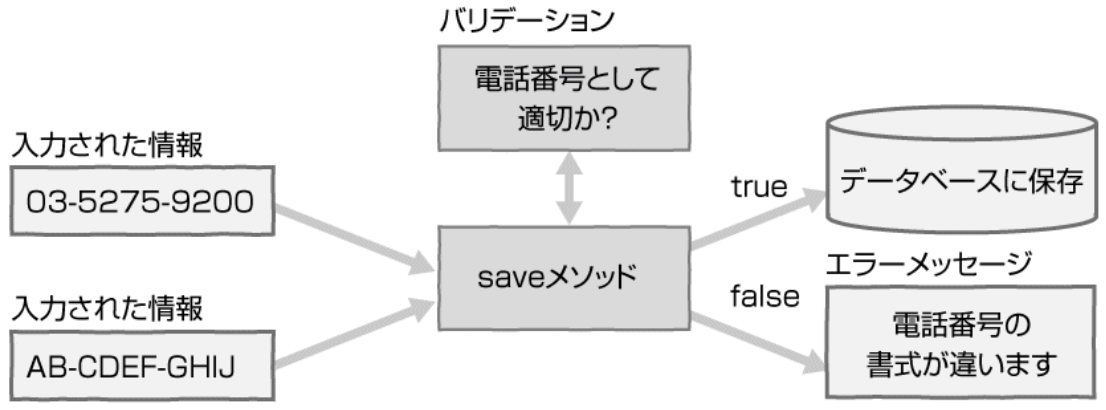
7.1 バリデーション

ここではモデルの機能のうち、バリデーションを取り上げます。ユーザーから渡されたデータをデータベースに保存するときに必ず行う、重要なステップです。また、バリデーションによって発生したエラーをユーザーに見せる方法を紹介します。

バリデーション

■バリデーションとエラー情報

テーブルのレコードを作成・更新するときには、「フォームから送られた値を調べ、値の形式や範囲が間違っていたら適切なエラーメッセージを出す」という処理を必ず行います。この定型的な処理を簡単にこなすのが、Railsのモデルに用意された**バリデーション**（値の検証）です。



バリデーション

モデルオブジェクトのsaveメソッドを呼び出すと、saveメソッドはバリデーションを実行します。バリデーションに引っかかると、エラーオブジェクトにエラー情報が格納され、saveメソッドはfalseを返します。

バリデーションを設定するには、モデルクラス内でvalidatesメソッドを使います。次のようにvalidatesメソッドをMemberモデルに加えて試してみましょう。これは、背番号（number）が空かどうかをチェックします。

```
class Member < ApplicationRecord
  validates :number, presence: true
```

Railsコンソールで動作確認をしましょう。ターミナルでbin/rails cコマンドを実行してください。MSYS2/MinGW環境の方はwinpty ruby bin/rails cコマンドを使います。

```
irb(main):001:0> member = Member.first
  Member Load (0.6ms)  SELECT  "members".* FROM "members"  （略）
=> #<Member id: 1, number: 10, name: "Taro", full_name: "佐藤 太郎",
（略） >
irb(main):002:0> member.number = nil
=> nil
irb(main):003:0> member.save
   (0.2ms)  begin transaction
   (0.1ms)  rollback transaction
=> false
```

ここでは背番号をnilにしてデータの保存を試みています。saveメソッドがfalseを返していることが、バリデーション失敗を示しています。

モデルのエラーオブジェクト（ActiveModel::Errorsクラスのオブジェクト）は、errorsメソッドで取り出せます。エラー情報の内容を見るには、エラーオブジェクトのmessagesメソッドを使います。エラー情報は、「{ :属性名 => ["メッセージ1", "メッセージ2"] }」という形式のハッシュで格納されています。

```
irb(main):004:0> member.errors.messages
=> {:number=>["can't be blank"]}
```

エラーの有無は、エラーオブジェクトのempty?で確認できます。エラーオブジェクトをハッシュのように扱って、属性ごとのエラー情報を取り出すこともできます。

```
irb(main):005:0> member.errors.empty?
=> false
irb(main):006:0> member.errors[:number]
=> ["can't be blank"]
```

valid? メソッドとinvalid? メソッド

saveメソッドを使わずにバリデーションを行うには、valid?メソッドかinvalid?メソッドを使います。チェックに引っかかればvalid?メソッドはfalseを返し、invalid?メソッドはtrueを返します。

```
member.number = nil
member.valid?   # false
member.invalid?  # true
```

■validatesメソッドの書き方

validatesメソッドの引数には、シンボルでモデルの属性名を指定し、そのあとにハッシュで「バリデーションの種類: true」を並べれば、その種類のバリデーションが行われます。属性名もバリデーションの種類も複数並べられます。

```
validates :number, :name, presence: true
```

「バリデーションの種類: { オプション: オプションの値 }」とすると、バリデーションごとにオプションを指定できます。

```
validates :name, length: { maximum: 20 }
```

■会員情報の検証

Memberモデルにバリデーションの機能を加えていきましょう。会員の背番号、ユーザー名、氏名でバリデーションを行います。

まず、背番号には「空を禁止、1以上100未満の整数、会員の間で重複を禁止」という制限を付けることにして、次のようにvalidatesメソッドを記述します。

LIST chapter07/app/models/member.rb

```
1 class Member < ApplicationRecord
2   validates :number, presence: true,
3     numericality: {
4       only_integer: true,
5       greater_than: 0,
6       less_than: 100,
7       allow_blank: true
8     },
9     uniqueness: true
    （以下省略）
```

numericalityオプションにはハッシュでサブオプションを付けて、細かい調整をしています。only_integer: trueで「整数のみ」、greater_than: 0とless_than: 0で「1以上100未満」という意味になります。

ところで、空の背番号が禁止されているのに4行目でallow_blank: trueが指定されているのは奇妙な感じがするかもしれません。これは、2行目で設定しているバリデーション

presence: trueとの重複を避けるためです。4行目を削除すると、空の会員番号を入力したときに「背番号が空」というエラーのほかに、「背番号が数値ではない」というエラーが発生します。これは、ユーザーにとって少々わずらわしい状況です。

ユーザー名は、「空を禁止、半角英数字のみ、文字列の先頭はアルファベット、2文字以上20文字以下、会員の間で重複を禁止（大文字小文字を区別しない）」とします。

LIST chapter07/app/models/member.rb

```
（省略）
10  validates :name, presence: true,
11    format: { with: /¥A[A-Za-z][A-Za-z0-9]*¥z/, allow_blank: true },
12    length: { minimum: 2, maximum: 20, allow_blank: true },
13    uniqueness: { case_sensitive: false }
（以下省略）
```

11行目にある/¥A[A-Za-z][A-Za-z0-9]*¥z/は正規表現と呼ばれるオブジェクトです。

正規表現

正規表現とはある文字列のパターンを表します。11行目で使われている正規表現は4つの部分に分かれます。最初の¥Aは文字列の先頭、次の[A-Za-z]はアルファベット1文字、その次の[A-Za-z0-9]*は任意の長さの（0個以上の）半角英数字、最後の¥zは文字列の末尾を示します。本書では正規表現の詳しい解説はしません。興味のある方は、次のページを参照してください。

Rubyリファレンスマニュアル（正規表現）
https://docs.ruby-lang.org/ja/latest/doc/spec=2fregexp.html

氏名は、「空を禁止、20文字以下」とします。

LIST chapter07/app/models/member.rb

```
（省略）
14  validates :full_name, presence: true, length: { maximum: 20 }
```

```
15
16  class << self
17    def search(query)
    （以下省略）
```

presenceバリデーションの働き

presenceで「値が空」と見なされるのは、nil、false、空文字列です。空白文字（半角空白、改行、タブ）だけを並べた値もエラーにします。これは、blank? メソッドがtrueを返す場合と同じです。なお、全角空白も空白文字として扱われます。

■メールアドレスのチェック

続いて、メールアドレスのバリデーションをMemberモデルに追加します。会員のメールアドレスは空であってもかまいませんが、「Taro@example@com」のような不正な形式の文字列はエラーとします。

メールアドレスの形式に関する規則は非常に複雑なので、それをチェックするしくみを自分で作るのは困難です。このような場合には、既存のGemパッケージを利用することを考えましょう。

ブラウザでhttps://rubygems.orgを開き、キーワード「email validate」で検索すると多くのパッケージがヒットします。この中からダウンロード数や最終リリース日などを参考にしてよさそうなものを選択します。本書では、email_validatorを採用します。

Gemfileを次のように書き換えてください。

```
(1-32行省略)
gem 'bootsnap', '>= 1.1.0', require: false

gem 'email_validator', '~> 1.6'
```

```
group :development, :test do
(以下省略)
```

ここで、もしRailsサーバーが起動していたらCtrl-Cで止めてください。そして、ターミナルでbundle installコマンドを実行してから、Railsサーバーを起動し直します。

```
$ bundle install
$ bin/rails s
```

MSYS2/MinGW環境でLoadErrorが出た場合

Chapter 1でも書きましたが、MSYS2/MinGW環境では次のようなエラーが出るかもしれません。

```
cannot load such file -- sqlite3/sqlite3_native (LoadError)
```

次の2つのコマンドを順に実行してからRailsサーバーを起動してください。

```
$ gem uninstall -a sqlite3
$ gem install sqlite3 --platform ruby -N
```

なお、このエラーはbundle installコマンドを実行するたびに発生する可能性があります。以降のChapterでは説明を繰り返しませんが、そのようなときには上記の2つのコマンドを実行してください。

Memberモデルのソースコードを次のように書き換えます。

LIST chapter07/app/models/member.rb

```
（省略）
14  validates :full_name, length: { maximum: 20 }
15  validates :email, email: { allow_blank: true }
16
```

```
17  class << self
18    def search(query)
    （以下省略）
```

Rails標準のvalidatesメソッドにはemailというオプションはありませんが、Gemパッケージemail_validatorにより拡張されています。単にemail: trueとだけ指定すると、空文字が不正なメールアドレスと判定されてしまうので、サブオプションallow_blankにtrueを指定しています。

エラーメッセージの表示

バリデーションを設定しただけでは、保存に失敗したときのメッセージは表示されません。テンプレートにエラーメッセージの表示を加えましょう。

エラーメッセージ専用のテンプレートを用意することにします。app/views/sharedディレクトリの下に部分テンプレート_errors.html.erbを作成し、次のように記述してください。エラーオブジェクトのfull_messagesメソッドは、「エラーを出した属性名＋メッセージ」の配列を返しますので、それをHTMLのリストにします。

LIST chapter07/app/views/shared/_errors.html.erb

```
1 <% if obj.errors.present? %>
2  <div id="errors">
3    <h3>エラーがあります。</h3>
4    <ul>
5      <% obj.errors.full_messages.each do |msg| %>
6        <li><%= msg %></li>
7      <% end %>
8    </ul>
```

```
 9   </div>
10 <% end %>
```

会員情報のフォームを表示するテンプレート_form.html.erbの先頭に、この_errors.html.erbを埋め込みます。「obj: @member」で_errors.html.erbのローカル変数objがモデルオブジェクトを参照するようにします。

LIST chapter07/app/views/members/_form.html.erb

```
1 <%= render "shared/errors", obj: @member %>
  （以下省略）
```

さらに、app/assets/stylesheetsディレクトリにエラー表示用のスタイルを記述したerrors.cssを作成します。ソースコードの掲載は省略しますので、サンプルソースのchapter07/app/assets/stylesheetsディレクトリからコピーしてください。

会員一覧のページを開き、適当な会員の［編集］リンクをクリックして動作確認をしましょう。背番号の欄に「aaa」と入力し、ユーザー名の欄を「aaa!」のような不正なものにします。また、氏名欄を空にし、メールアドレスとして「Taro@example@com」のような不正な値を入力します。そして、［Update Member］ボタンを押して保存しようとすると、次の画面になります。

RESULT エラーメッセージの表示

フォームの上部にエラーメッセージが表示され、エラーを起こした入力欄と対応するラベルは赤い四角で囲まれます。エラーメッセージが英語になっていますが、これは次節の7.2節で日本語にします。

入力欄を囲むタグ

モデルオブジェクトにエラーがあるときは、エラーを起こした入力欄は、class属性が"field_with_errors"のdivタグで囲まれます。

```
<div class="field_with_errors"><input id="member_name"
 name="member[name]" size="30" type="text" value="" /></div>
```

CSSでdiv.field_with_errorsにスタイルを記述すれば、エラーを起こした入力欄に色を付けられます。

```
div.field_with_errors {
  background-color: #fcc;
  padding: 2px;
}
```

エラーメッセージの順番

「Number is not a number」や「Name can't be blank」などのエラーメッセージの順番は、バリデーションを実行した順番になります。これは、モデルクラスの中でvalidatesメソッドを並べた順番です。フォームの入力欄の並びと同じ順でエラーを表示したいときは、validatesメソッドの並べ方を調整してください。

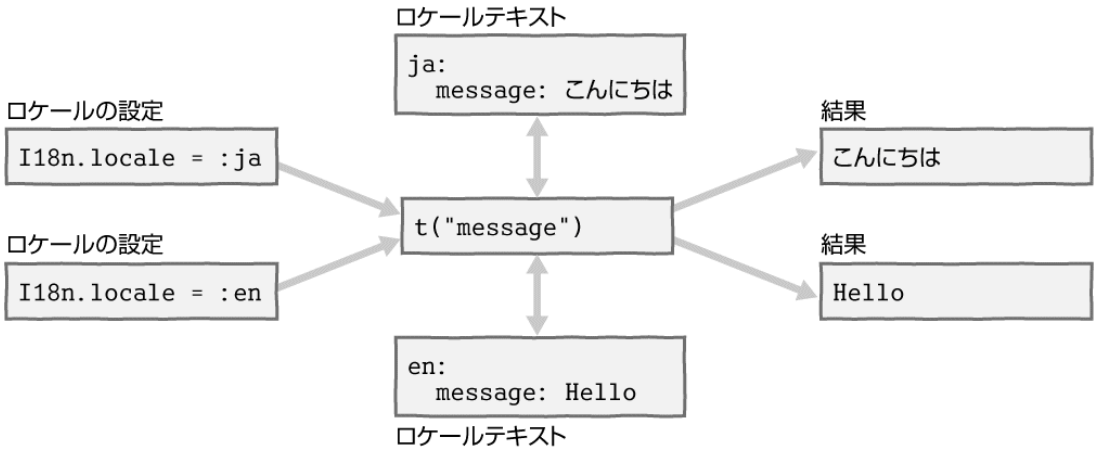
7.2 メッセージの日本語化

Railsの国際化機能を使うと、日本語や英語などさまざまな言語のテキストをブラウザ上に表示できます。この機能を使って、前節で作成したエラーメッセージを日本語に変えてみましょう。

Railsの国際化機能

国際化とは英語でinternationalization、略してi18nと呼ばれるものです。複数の言語に合わせてテキストを切り替えたり、地域ごとに日付や数値の書式を切り替えたりする機能のことです。

Railsの国際化機能は、単純なものです。I18nというクラスにロケール（言語＋地域の種類）を設定し、ロケールに合わせてYAML形式のロケールテキストから文字列を読み込んで表示します。YAMLの書式については後述します。



Railsの国際化機能

ロケールを設定するには、コントローラでI18n.localeにロケールを表す文字列またはシンボルを指定します。日本語は「ja」、英語は「en」です。「en-US」（アメリカ英語）や「zh-

TW」（台湾の中国語）のように「言語-地域」の形でも指定できます。

```
I18n.locale = "ja"
```

この指定がなければ、デフォルトのロケール「en」（英語）が使われます。asagaoアプリケーションのデフォルトのロケールを日本語に設定しましょう。configディレクトリのapplication.rbを次のように書き換えてください。

LIST chapter07/config/application.rb

```
（省略）
33    config.time_zone = "Tokyo"
34    config.i18n.default_locale = :ja
（以下省略）
```

application.rbは自動的に再読み込みされないので、サーバーを起動中のときは、終了して再起動してください。

i18nとL10n

国際化と似た概念に地域化があります。英語でLocalization、略してL10nです。大ざっぱに言えば、i18nは多言語対応のことで、L10nはアプリケーションを特定の言語に対応させる（たとえば、英語のソフトを日本語化する）ことです。

この7.2節で行うことは、Railsの国際化機能を使ってアプリケーションを日本語化すること、つまりi18nを使ってL10nを行うことと言えます。

エラーメッセージの日本語化

Railsの国際化機能を使って、会員情報のエラーメッセージとモデルの属性名を日本語化しましょう。

■Gemパッケージrails-i18n

レコードの保存に失敗したときに表示される「can't be blank」の部分を日本語化しましょう。これは、Gemパッケージrails-i18nを使うと楽にできます。rails-i18nは、さまざまな言語のためのエラーメッセージや日付の書式などを集めたものです。

Railsアプリケーションのディレクトリの直下にあるGemfileを開いて、bootsnapの次の行にrails-i18nの設定を加えてください。GemfileとBundlerの説明は、「1.4　アプリケーションの新規作成」の「Bundler」にあります。

LIST chapter07/Gemfile

```
（省略）
33 gem 'bootsnap', '>= 1.1.0', require: false
34
35 gem 'email_validator', '~> 1.6'
36 gem 'rails-i18n', '~> 5.1'
（以下省略）
```

ターミナルでrails-i18nをインストールします。

```
$ bundle install
```

これにより、モデルは日本語用のロケールテキストからエラーメッセージを読み込むようになります。ロケールテキストの内容を確認してみましょう。次のサイトでrails-i18nのソースをのぞいてみましょう。

https://github.com/svenfuchs/rails-i18n

このページから「rails」→「locale」→「ja.yml」をクリックします。ページの途中に次のような記述があります。

```
errors:
  format: "%{attribute}%{message}"
  messages:
    accepted: を受諾してください
    blank: を入力してください
    present: は入力しないでください
    confirmation: と%{attribute}の入力が一致しません
    empty: を入力してください
    equal_to: は%{count}にしてください
    even: は偶数にしてください
  （後略）
```

これが日本語用のエラーメッセージになります。エラーオブジェクトのaddメソッドの第2引数には、:acceptedや:blankなどのエラーを表すシンボルを指定できます。

GitHub

GitHubは、オープンソースのソフトウェアをインターネットを通じて共同開発するためのウェブサイトです。GitHubのrails-i18nは、Rails向けのさまざまな言語のロケールテキストを集めるためのプロジェクトです。ちなみに、rails-i18nの管理者グループには本書の著者たちが参加しています。

Railsの開発チームもGitHubを使っており、次のページで開発中のRailsのソースコードを閲覧できます。

https://github.com/rails/rails

■モデルの属性名の日本語化

rails-i18nによって、エラーメッセージ「Name can't be blank」の「can't be blank」の部分は「を入力してください。」のように日本語化できましたが、「Name」の部分はまだです。自分でロケールテキストを記述して、日本語のテキストを用意する必要があります。

Railsアプリケーションのconfigディレクトリの下にlocalesディレクトリがあります。ここがロケールテキストを置く場所です。ここにja.ymlという名前のファイルを新しく作成してください。次のように記述して、Memberモデルの属性に日本語の名前を付けましょう。

LIST chapter07/config/locales/ja.yml

```
1 ja:
2 ␣␣activerecord:
3 ␣␣␣␣models:
4 ␣␣␣␣␣␣member: 会員情報
5 ␣␣␣␣attributes:
6 ␣␣␣␣␣␣member:
7 ␣␣␣␣␣␣␣␣number: 背番号
8 ␣␣␣␣␣␣␣␣name: ユーザー名
9 ␣␣␣␣␣␣␣␣full_name: 氏名
10 ␣␣␣␣␣␣␣␣sex: 性別
11 ␣␣␣␣␣␣␣␣sex_1: 男
12 ␣␣␣␣␣␣␣␣sex_2: 女
13 ␣␣␣␣␣␣␣␣birthday: 誕生日
14 ␣␣␣␣␣␣␣␣email: メールアドレス
15 ␣␣␣␣␣␣␣␣administrator: 管理者
```

YAML形式で入力するときの注意

インデントはタブではなく半角空白で入力し、「number: 背番号」のような行はコロンのあとに必ず半角空白を入れてください。ファイルを保存するときは、文字コードはUTF-8にしてください。

「number: 背番号」のように文字列を指定するときは、「number: "背番号"」と二重引用符で囲むこともできます。文字列中に空白が含まれるときは、二重引用符が必須です。

前節までに作った会員情報のフォームを次のように変更しましょう。

LIST chapter07/app/views/members/_form.html.erb

```
1 <%= render "shared/errors", obj: @member %>
2
3 <table class="attr">
4   <tr>
5     <th><%= form.label :number %></th>
6     <td><%= form.text_field :number, size: 8 %></td>
7   </tr>
8   <tr>
9     <th><%= form.label :name %></th>
10    <td><%= form.text_field :name %></td>
11  </tr>
12  <tr>
13    <th><%= form.label :full_name %></th>
14    <td><%= form.text_field :full_name %></td>
15  </tr>
16  <tr>
17    <th><%= Member.human_attribute_name(:sex) %></th>
18    <td>
19      <%= form.radio_button :sex, 1 %>
20      <%= form.label :sex_1 %>
21      <%= form.radio_button :sex, 2 %>
22      <%= form.label :sex_2 %>
23    </td>
24  </tr>
25  <tr>
26    <th><%= form.label :birthday, for: "member_birthday_1i" %>
</th>
```

```
27    <td><%= form.date_select :birthday,
28            start_year: 1940, end_year: Time.current.year,
29            use_month_numbers: true %></td>
30  </tr>
31  <tr>
32    <th><%= form.label :email %></th>
33    <td><%= form.text_field :email %></td>
34  </tr>
35  <tr>
36    <th><%= Member.human_attribute_name(:administrator) %>
</th>
37    <td>
38      <%= form.check_box :administrator %>
39      <%= form.label :administrator %>
40    </td>
41  </tr>
42 </table>
```

labelメソッドの第2引数を削除して、ja.ymlのテキストを使うようにしています。17行目と36行目で使っているクラスメソッドhuman_attribute_nameは、モデルの属性名をロケールテキストから取り出すものです。

サーバーを起動して、会員名簿のページから適当な会員の［編集］リンクをクリックし、フォームの表示を確認してください。各フィールドのラベルが以前と同様に表示されていて、フォームの送信ボタンのラベルテキストが「Update Member」から「更新する」に変化していればOKです。送信ボタンのラベルテキストはGemパッケージrails-i18nに含まれています。

さらに、入力欄に不正なデータを入力してフォームを送信し、エラーメッセージが日本語で表示されることも確認してください。

RESULT エラーメッセージの日本語化

ロケールテキストの再読み込み

開発モードでは、config/localesディレクトリの下のロケールテキストをサーバーの起動中に変更してもかまいません。自動的に再読み込みが行われます。ただし、新しいファイルを追加した場合は読み込まれないので、サーバーを再起動する必要があります。

■エラーメッセージのカスタマイズ

モデルのエラーメッセージはrails-i18nのものを使うだけでなく、カスタマイズできます。自分のロケールテキストでja:→activerecord:→errors:→messages:の下に「エラー名: 文字列」を記述すれば、それが優先されます。

たとえば、「は不正な値です。」のテキストを変えたければ、次のように記述します。

```
ja:
  activerecord:
    （中略）
  errors:
    messages:
      invalid: "の書式が正しくありません。"
```

新しいエラーメッセージを追加することもできます。会員のユーザー名が不正なときは、「は半角英数字で入力してください。」と表示したいときは、次のようにロケールテキストにエラーメッセージを追加します。

LIST chapter07/config/locales/ja.yml

```
 1 ja:
 2 ␣␣activerecord:
   （省略）
14 ␣␣␣␣␣␣␣␣␣email: メールアドレス
15 ␣␣␣␣␣␣␣␣␣administrator: 管理者
16 ␣␣␣␣errors:
17 ␣␣␣␣␣␣messages:
18 ␣␣␣␣␣␣␣␣␣invalid_member_name: は半角英数字で入力してください。
```

Memberモデルで、name属性のバリデーションを変更します。formatにmessageオプションを追加し、ロケールテキストで設定したエラー名を指定します。

LIST chapter07/app/models/member.rb

```
   （省略）
10   validates :name, presence: true,
11     format: {
```

```
12      with: /¥A[A-Za-z][A-Za-z0-9]*¥z/,
13      allow_blank: true,
14      message: :invalid_member_name
15    },
16    length: { minimum: 2, maximum: 20, allow_blank: true },
17    uniqueness: { case_sensitive: false }
    （以下省略）
```

会員情報入力フォームで、ユーザー名の欄に「Taro!」と入力して更新ボタンを押すと、次のようなエラーメッセージが示されます。

RESULT カスタマイズしたエラーメッセージ

■YAML

Railsでは、ロケールテキストの記述に**YAML**を使っています。YAMLとは、データ形式またはファイル形式の1つで、配列やハッシュのようなデータを人間に読みやすい形で記述できるようにしたものです。YAMLは「YAML Ain't Markup Language」（YAMLはマークアップ言語ではない）の略とされています。冗談のような名前が付いているのは、複雑になりすぎたXMLに対する批判の気持ちが込められているのかもしれません。

ここでは、YAMLの書き方とRubyでの簡単な利用法を紹介します。次は、YAMLで配列を表したものです。各行の先頭に-を付ければ、配列になります。

```
- cat
- tiger
- lion
```

このYAMLをanimals.ymlというファイル名で保存すると、Rubyプログラムでは次のように配列を取り出せます。

```
require "yaml"

animals = YAML.load(File.new("animals.yml"))
animals.each { |animal| puts animal }
```

YAMLでは、ハッシュを表すこともできます。「キー: 値」を各行に並べれば、それがハッシュのキーと値になります。

次ページのYAMLは、2つの要素からなるハッシュを表し、ハッシュの中にハッシュを入れて会員のデータを表しています。配列やハッシュの中にさらに配列やハッシュを入れるときは、半角空白でインデントします（タブ文字は使えません）。

```
member1:
  number: 11
  name: Taro
```

```
member2:
  number: 12
  name: Hanako
```

members.ymlというファイル名で保存すると、次のようにハッシュを取り出せます。

```
require "yaml"

members = YAML.load(File.new("members.yml"))
members.each do |key, val|
  puts "#{val['number']}¥t#{val['name']}"
end
```

Railsでは、ロケールテキストのほかに、データベースの設定（database.yml）にYAMLを使っています。database.ymlについては「4.1　データベースとモデルの基本」で見ました。

YAMLについてさらに知りたい方は、次のサイトを参照してください（2018年4月現在の情報です）。

プログラマーのためのYAML入門
https://magazine.rubyist.net/articles/0009/0009-YAML.html
YAML仕様書（英語）
http://yaml.org/spec/1.2/spec.html

国際化機能の使い方

ここでは、Railsの国際化機能の一般的な使い方を紹介します。

テキスト

日本語用のテキストを用意するには、ロケールテキストの中でja:の下にネストしたハッシュを記述します。

```
ja:
  messages:
    hello: "こんにちは"
```

Railsアプリケーションの中でロケールテキストを読み出すには、I18nクラスのtranslateメソッド（別名t）を使います。tメソッドには、ハッシュのキーをピリオドでつないだ文字列を指定します。ロケールの設定が「ja」の場合はja:の下のテキストが使われ、「en」の場合はen:の下のテキストが使われます。

```
s = I18n.t("messages.hello")
```

コントローラやテンプレートの中では、I18nを付けずにtメソッドが利用できます。

```
<%= t("messages.hello") %>
```

ロケールテキストで%{ 〜 }を使うと、テキストの中に値を埋め込めるようになります。次は、あいさつの中に名前を埋め込む例です。

```
ja:
  messages:
    hello: "%{name}さん、こんにちは"
```

tメソッドの引数に次のようにハッシュを追加すれば、「Taroさん、こんにちは」というテキストになります。

```
<%= t("messages.hello", name: "Taro") %>
```

ハッシュの上書きに注意

1つのロケールテキストには、ja:は1つしか書けません。次のように2つ書くと、下のハッシュ"ja"が上のハッシュ"ja"を消して上書きしてしまいます。

```
ja:
  activerecord:
   （中略）
ja:
  messages:
```

ロケールテキストを記述するときは、ハッシュが上書きされないようにしてください。ただし、複数のロケールテキストを使うときは、ファイルごとにja:が必要です。

■日付と時刻

日付や時刻の表記は、言語や地域ごとに違いがあります。日本語では年月日を「2018/05/05」のように並べますが、アメリカ英語では「May 5, 2018」と月日年の順になります。こうした違いも、Railsの国際化機能で扱えます。

日付や時刻を現在のロケールに従った文字列にするには、I18nクラスのlocalizeメソッド（別名l）を使います。lメソッドの引数には、DateクラスやTimeクラス、またはActiveSupport::TimeWithZoneクラスのオブジェクトを渡します。

次の例は、「2018/05/05 14:25:06」のような文字列を返します。

```
s = I18n.l(Time.current)
```

tメソッドと同様に、コントローラやテンプレートの中では、I18nを付けずにlメソッドが利用できます。

```
<%= l(Time.current) %>
```

lメソッドの第2引数には、ハッシュでformatオプションを指定できます。短めの書式を使うときは:shortを、長めなら:longを指定します。次の例は、「2018年05月05日(土) 17時06分56秒 +0900」のような文字列になります。

```
<%= l(Time.current, format: :long) %>
```

lメソッドで使われる日付と時刻の書式は、rails-i18nのja.ymlの中で設定されています。date:の下が日付の書式、time:の下が日付と時刻の書式です。「%○○」の書式は、strftimeメソッドと同じです（「2.3　いろいろなオブジェクト」の「日時と日付」を参照）。formatオプションを省略したときは、defaultの書式が使われます。

```
ja:
  date:
    （中略）
    formats:
      default: "%Y/%m/%d"
      long: "%Y年%m月%d日(%a)"
      short: "%m/%d"
    （中略）
  time:
    am: 午前
    formats:
      default: "%Y/%m/%d %H:%M:%S"
      long: "%Y年%m月%d日(%a) %H時%M分%S秒 %z"
      short: "%y/%m/%d %H:%M"
    pm: 午後
```

自分で新しい書式を追加したり、rails-i18nの書式を上書きしたりしたいときは、アプリケーションのロケールテキストに書式を記述してください。次の例は、mediumという名前の書

式を追加します。

```
ja:
  （中略）
  time:
    formats:
      medium: "%Y年%m月%d日(%a) %H:%M"
```

formatオプションで:mediumを指定すると、「2018年05月05日 (土) 17:15」のような文字列になります。

```
<%= l(Time.current, format: :medium) %>
```

Chapter 7のまとめ

- **フォームの値が正しいかどうかをチェックするには、バリデーションを使います。バリデーションは、モデルクラスにvalidatesメソッドを記述して設定します。**
- **エラーメッセージを日本語化するには、Railsの国際化機能を使います。Gemパッケージrails-i18nをインストールすると簡単です。**
- **モデルの属性に日本語名を付けるには、YAML形式でロケールテキストを記述します。**

練習問題

［A］　Chapter 4の練習問題で作成したBookモデルに属性title、author、priceのためにバリデーションを記述してください。どの属性も値を空にしてはいけないものとします。priceは、0以上の整数とします。

```
class Book < ApplicationRecord
  
end
```

［B］　Bookモデルのためのロケールテキストを作成します。属性title、author、priceの日本語ラベルをYAML形式で記述してください。

```
ja:
  activerecord:
    models:
      book: 書籍
    attributes:
      book:
```

Chapter

8　単数リソース

　このChapterでは、Morning Gloryのサイトに①ログイン・ログアウト機能、②メンバーが自分自身の情報を閲覧・変更する機能（マイアカウント機能）、③メンバーが自分自身のパスワードを変更する機能を作りながら、単数リソース、セッション、アクション・コールバックなどの重要な概念について学んでいきます。

これから学ぶこと

- 「単数リソース」という概念とそれを設定する方法について学びます。
- セッションを利用してログイン・ログアウト機能をMorning Gloryのサイトに追加します。
- アクション・コールバックという概念とそれを設定する方法について学びます。
- ログイン中のメンバーだけに特定のページへのアクセスを許可する方法を学習します。
- メンバーが自分の情報を閲覧・編集できるマイアカウントページを作成します。
- メンバーが自分のパスワードを変更する機能を実装します。

Ruby on Railsの学習では「リソース」という概念の理解がとても重要です。Chapter 5とChapter 6で学んだ普通のリソースと本章で登場する「単数リソース」はどう違うのでしょうか？

8.1 単数リソース

この節では「単数リソース」という重要な概念について説明します。Chapter 5とChapter 6で学んだ「普通のリソース」との違いに注意しながら読み進めてください。

単数リソースとは

Chapter 5とChapter 6でリソースというRails用語とその具体的な実装方法を学びました。草野球チームMorning Gloryのメンバー全体をmembersという名前のリソースとして取り扱いました。このmembersリソースの重要な特徴は、それが集合的な概念であるということです。0人以上のメンバーからなる「集まり」を表現しています。

しかし、ウェブアプリケーションを構成する要素の中には、多くとも1個しか存在しないものがあります。この種のリソースを**単数リソース**（singular resource）と呼びます。この章では、単数リソースの例として「セッション」、「自分のアカウント情報」、「自分のパスワード」を取り上げます。

Rails用語の「セッション」については次節で詳しく説明しますが、一言で表現すれば「メンバーのログイン状態を管理するための情報」を指します。ウェブアプリケーションMorning Gloryには同時に複数のメンバーがログインできるので、セッションを「単数リソース」と呼ぶのは不自然に思えるかもしれません。しかし、視点を変えれば非常に自然な呼び方であることがわかります。

2人のメンバーAとBがそれぞれ別のパソコンからMorning Gloryにログインしているとします。AさんはBさんのセッションの中身を見ることができません。ということは、Aさんにとってセッションは実質的に1個しか存在しないことになります。そして、AさんがMorning Gloryからログアウトすれば、セッションの数は0になります。

同様に「自分のアカウント情報」も単数リソースと見なせます。アカウント情報とはmembersテーブルに格納されている自分自身のレコードのことです。それは0個または1個し

か存在しません。「自分のアカウント情報」が0個であるとは、その人がMorning Gloryのメンバーではない（未登録もしくは退会済み）ということを示します。

単数リソースのルーティング

単数リソースのルーティングを設定するには、config/routes.rbの中でresourcesメソッドではなく、単数形のresourceメソッドを使います。「自分のアカウント情報」を扱うコントローラがAccountsControllerなら、次のように記述します。

```
resource :account
```

メソッドの引数は単数の:accountとする点に注意してください。このように書いてもこのリソースを扱うコントローラの名前はAccountsController（複数形のsに注意）となります。

この設定により、次のパスとHTTPメソッドでアクションを呼び出せるようになります。

単数リソースのルーティング

アクション	パス	HTTP メソッド	パスを示すシンボル	パスを返すメソッド
show	/account	GET	:account	account_path
new	/account/new	GET	:new_account	new_account_path
edit	/account/edit	GET	:edit_account	edit_account_path
create	/account	POST	:account	account_path
update	/account	PATCH	:account	account_path
destroy	/account	DELETE	:account	account_path

単数リソースなので、集合を扱うindexアクションはありません。また、show、edit、update、destroyの各アクションには、idパラメータは必要ありません。なぜなら、ログインしていれば「自分のid」はセッションに格納されているからです。

ここでひとつ注意していただきたい点があります。Rails初心者の方は、あるリソースを定義したらそれと同じ名前のデータベーステーブルも作らなければならないと考えがちですが、それは誤りです。単数リソースaccountが扱うのはmembersテーブルです。ただし、membersリ

ソースがmembersテーブルに格納されたレコードの集合を操作するのに対し、accountリソースはmembersテーブルに格納された特定のレコードだけを操作します。

　また、次の節で見るように、リソースとデータベーステーブルが結び付かない場合もあります。Railsではセッションのデータをブラウザのクッキー（詳しくは後述）に格納します。

8.2 セッションを使ったログイン機能

会員制サイトのようなウェブアプリケーションでは、ユーザーを識別し、有効な会員であるかどうかを判定するしくみが必要です。Railsのセッション機能を利用して、ユーザーが会員としてログインする機能を実装してみましょう。

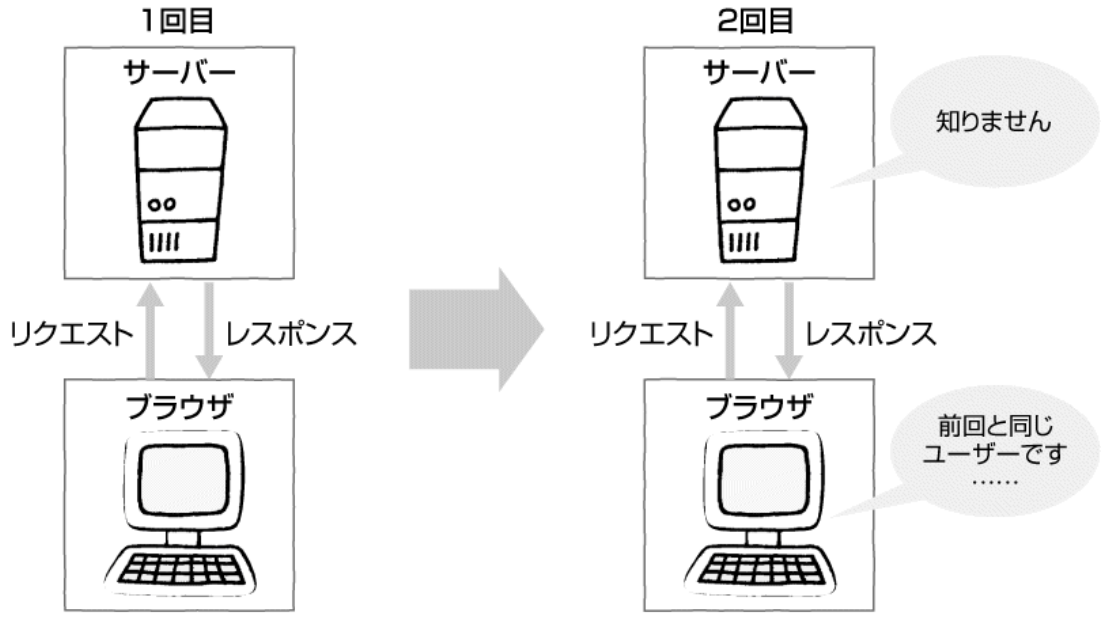
セッションとは

ログイン機能を実装する前に、Railsのセッション機能について紹介しましょう。

Railsのセッション

ウェブアプリケーションでは、複数のページにわたってブラウザとサーバーの接続を保つ仕掛けを作ります。そうした接続状態を**セッション**と呼びます。あるユーザーがログインしてからログアウトするまで、そのユーザーをほかのユーザーと区別するには、セッションが必要になります。

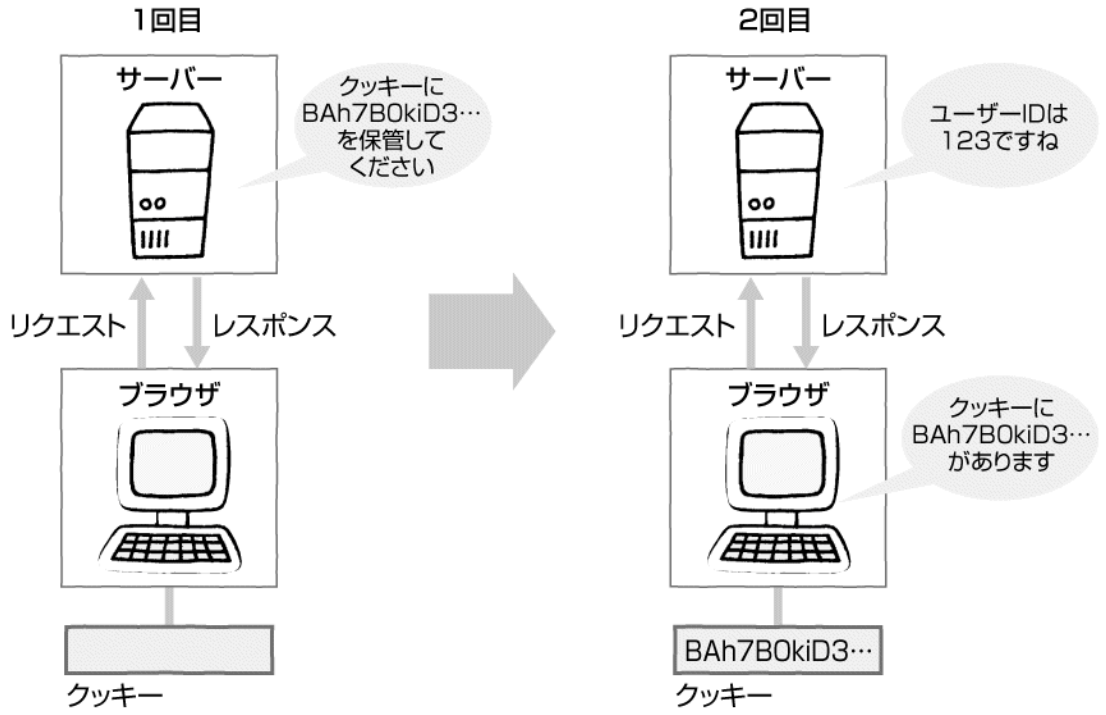
HTTPでは、ブラウザがページを1つリクエストしてサーバーがレスポンスを1つ返すと、ブラウザとサーバーの関係はそこで終わりになります。セッションを使わないと、同じユーザーが何度リクエストを送っても、サーバーは同じユーザーとは見なしません。



通常のHTTPプロトコル

複数のリクエストにわたって何らかの状態を保っておきたいときは、クッキーを使います。**クッキー**は、ブラウザがパソコンのハードディスクに保存する小さなデータです。ウェブアプリケーションでセッションを実現するには、クッキーを使うのが一般的です。

Railsは、セッション用のデータをブラウザのクッキーに保存して、ユーザーを識別します。このセッションデータにユーザーごとの情報を出し入れできます。

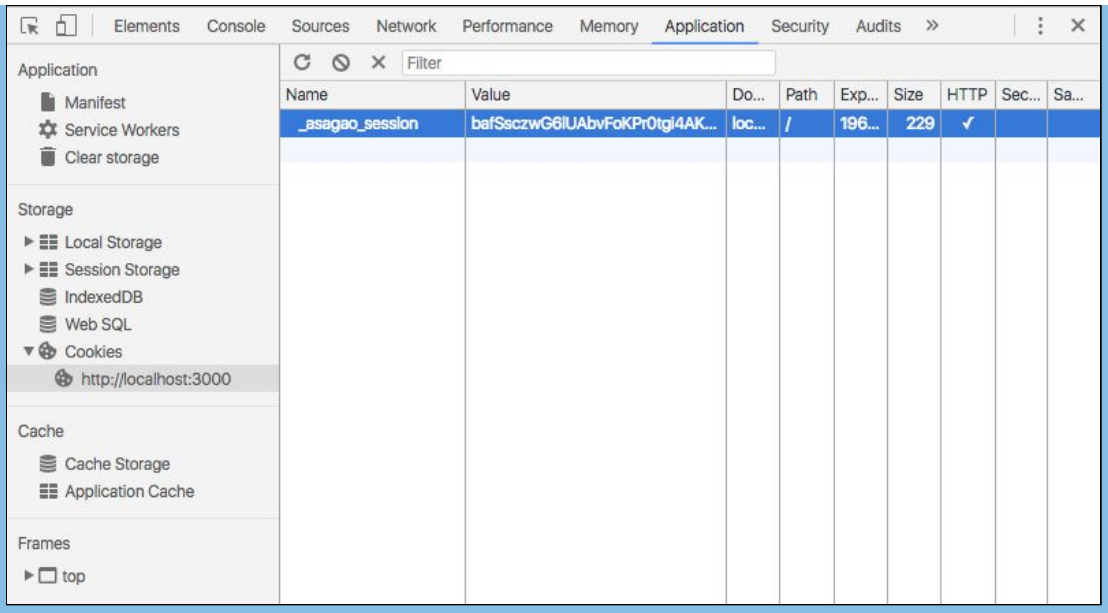


Railsのセッション

Railsはセッションデータを符号化してクッキーに保存しますが、暗号化はしませんので、ユーザーがデータを解読できることに注意してください。ただし、セッションデータが改ざんされた場合はエラーになるしくみになっています。

クッキーの中身を見る

次の手順でブラウザに保存されたクッキーの中身を見ることができます。Google Chromeの場合は、ページ上を右クリックして［検証］を選んでDevToolsを開き、［Application］タブの［Storage］→［Cookies］を開いてください。Firefoxの場合は、ページ上を右クリックして［ページの情報を表示］を選び、［セキュリティ］タブを開いて［Cookieを表示］ボタンをクリックします。次の図ではGoogle Chromeでasagaoの_asagao_sessionという名前のクッキーデータを表示しています。



Google Chromeでクッキーを表示

■セッションデータへのアクセス

Railsでセッションを利用するのはとても簡単です。セッションデータに何か値を入れるには、コントローラで「session[:データ名] = 値」とするだけです。データ名を変えれば、複数のデータをセッションデータに入れることもできます。

たとえば、ログインの処理を行うアクションの中で、セッションデータに会員のidを保存するには、次のようにします。

```
session[:member_id] = member.id
```

別のアクションで「session[:データ名]」から値を取り出せば、ログインしている会員のidがわかります。

```
id = session[:member_id]
member = Member.find(id)
```

ログアウト処理のためにセッションデータを消すには、deleteメソッドを使います。

```
session.delete(:member_id)
```

この例では会員のidカラムの値、つまり数値を保存しています。セッションデータにはたいていのオブジェクトを保存できますが、セッションデータはクッキーに保存されるので、サイズに上限があります。セッションデータには会員のidのような小さな値だけを保存するようにしてください。なお、フラッシュの文字列もセッションデータに保存されるしくみになっています。

普通のクッキーを使うには

セッションデータではなく、一般的なクッキーを使うには、コントローラで「cookies[:クッキー名]」に値を指定します。

```
cookies[:name] = "sato"
name = cookies[:name]
```

クッキーにオプションを指定したいときは、ハッシュを使います。次の例でvalueはクッキーの値、expiresは有効期限となる日時です。このほかのオプションとして、path（有効なパス）、domain（有効なドメイン）、secure（HTTPSのみ）、httponly（JavaScriptに使わせない）が指定可能です。

```
cookies[:name] = { value: "sato", expires: 30.days.from_now }
```

セッションデータの保存期間

セッションデータを保存したクッキーは、ブラウザを閉じると消去されます。ログイン状態のユーザーがブラウザを閉じると、ログアウト状態になります。ブラウザを閉じたり開いたりしてもログイン状態を保つ（いわゆる「自動ログイン」）には、セッションデータではなく通常のクッキーに情報を保存する必要があります。会員のidをクッキーに保存したいときは、cookiesの代わりにcookies.signedを使って、符号化されたデータを保存するとよいでしょう。

```
cookies.signed[:member_id] = member.id
```

パスワードの保存

Ruby on Railsにはパスワードを安全に取り扱うためのしくみが標準で用意されています。

■ハッシュ値とbcrypt

ウェブアプリケーションの開発において、データベースに生の（平文の）パスワードを保存するのは情報セキュリティの観点から望ましくありません。その代わりに、パスワードのハッシュ値（あるいはダイジェスト）と呼ばれる値をデータベースに保存するのが定石です。

ハッシュ値とは、あるデータを暗号学的ハッシュ関数と呼ばれる方式で変換したものです。ハッシュ値から元のデータを復元することはできません。また、異なる2つのデータから作ったハッシュ値同士が同じ値になる（衝突する）可能性は極めて低いです。

暗号学的ハッシュ関数にはmd5、sha、bcryptなどさまざまな種類がありますが、通常Ruby on Railsではbcryptを利用します。ただし、そのためにはGemパッケージbcrypt-rubyを組み込む必要があります。初期状態のGemfileにおいてコメントアウトされているbcrypt-ruby設定の行から、行頭のコメント記号（#）を削除してください。

LIST chapter08/Gemfile

```
（省略）
23 # Use ActiveModel has_secure_password
24 gem 'bcrypt', '~> 3.1.7'
（以下省略）
```

Bundlerでbcryptをインストールしましょう。

```
$ bundle install
```

■ハッシュ値を保存するカラムの追加

次に、パスワードのハッシュ値（ダイジェスト）を保存するためのカラムpassword_digestをmembersテーブルに加えましょう。Rails標準の方法でパスワードを取り扱う場合、このカラム名を使用する必要があります。

「bin/rails g」コマンドでマイグレーションスクリプトを作成しましょう。マイグレーションのクラス名はAlterMembers1とします。

```
$ bin/rails g migration AlterMembers1
```

作成されたマイグレーションスクリプトを次のように書き直します。

LIST chapter08/db/migrate/20180526135718_alter_members1.rb

```
1 class AlterMembers1 < ActiveRecord::Migration[5.2]
2   def change
3     add_column :members, :password_digest, :string
4   end
5 end
```

マイグレーションを実行します。

```
$ bin/rails db:migrate
```

■クラスメソッドhas_secure_password

クラスメソッドhas_secure_passwordを用いると、パスワードの保存と認証のためのしくみをモデルクラスに追加することができます。

LIST chapter08/app/models/member.rb

```
1 class Member < ApplicationRecord
2   has_secure_password
3
4   validates :number, presence: true,
    （以下省略）
```

この変更の結果として、Memberクラスにpasswordおよびpassword_confirmationという名前の2つの属性が定義されます。前者はパスワードそのもの、後者は確認用のパスワードです。また、password属性に対する次のようなバリデーションが設定されます。

- レコードの挿入時には、パスワードは空であってはならない。
- パスワードの長さは72文字以下である。
- パスワードと確認用のパスワードは一致しなければならない。

これらのバリデーションを無効にしたい場合は、次のようにvalidationオプションにfalseを指定してください。

```
has_secure_password validation: false
```

さて、レコード挿入時にパスワードが必須となりましたので、開発用のシードデータを修正しなければなりません。メンバー全員に仮のパスワードとして「asagao!」を設定します。

LIST chapter08/db/seeds/development/members.rb

```
    （省略）
12    administrator: (idx == 0),
13    password: "asagao!",
14    password_confirmation: "asagao!"
15  )
16 end
```

データベースをリセットしてエラーが出なければOKです。

```
$ bin/rails db:rebuild
```

ユーザーの認証

ユーザーが送信した名前とパスワードを調べ、ログイン状態に変える機能を作成しましょう。

authenticateメソッド

モデルクラス定義の中でクラスメソッドhas_secure_passwordを呼び出すと、password属性とpassword_confirmation属性が定義されるほかに、authenticateメソッドも追加されます。Railsコンソールを起動して、このメソッドを使ってみましょう。MSYS2/MinGW環境の方はコマンドの前にwinpty rubyを付けてください。

```
$ bin/rails c
irb(main):001:0> member = Member.first
  Member Load (0.6ms)  SELECT  "members".* FROM "members"  （略）
=> #<Member id: 1, number: 10, name: "Taro", full_name: "佐藤 太郎",
（略） >
irb(main):002:0> member.authenticate("detarame")
=> false
irb(main):003:0> member.authenticate("asagao!")
=> #<Member id: 1, number: 10, name: "Taro", full_name: "佐藤 太郎",
（略） >
```

authenticateメソッドは引数として生の（平文の）パスワードを取り、それが正しいパスワードであるかどうかを調べ、正しければモデルオブジェクト自体を返し、誤っていればfalseを返します。

■SessionsControllerの作成

では、authenticateメソッドを使って、ユーザーをログイン状態にするコントローラSessionsControllerを作成しましょう。config/routes.rbにリソースの設定を記述します。

LIST chapter08/config/routes.rb

```
（省略）
 9  resources :members do
10    get "search", on: :collection
11  end
12
13  resource :session, only: [:create, :destroy]
14 end
```

ここではresourcesメソッドではなく、単数形のresourceメソッドを使い、「単数リソース」を設定しています。引数の:sessionも単数形にする点に注意してください。

SessionsControllerのアクションはcreateとdestroyの2つしかないので、onlyオプションでアクションの種類を限定します。

「bin/rails g」コマンドでコントローラを作成します。単数リソースを扱うコントローラでもコントローラ名は複数形にします。名前はsessionsとします。

```
$ bin/rails g controller sessions
```

SessionsControllerは次のように実装します。

LIST chapter08/app/controllers/sessions_controller.rb

```
1 class SessionsController < ApplicationController
2   def create
3     member = Member.find_by(name: params[:name])
4     if member&.authenticate(params[:password])
```

```
    session[:member_id] = member.id
   else
    flash.alert = "名前とパスワードが一致しません"
   end
   redirect_to :root
  end

  def destroy
   session.delete(:member_id)
   redirect_to :root
  end
end
```

createアクションはログインフォームの送信先となるものです。まず、モデルのクラスメソッドfind_byを使って送信された名前に合致するメンバーを取り出し変数memberにセットします。そして、authenticateメソッドで送信されたパスワードが正しいかどうかをチェックします。ログインフォームから存在しない名前が送信されてきた場合は、変数memberにnilがセットされるので、ぼっち演算子（&.）を用いてエラーの発生を防いでいます。

パスワードが正しい場合は、Memberオブジェクトのidをセッションデータmember_idに保存します。正しくない場合は、フラッシュ（「3.2　コントローラとアクション」の「フラッシュ」を参照）に警告メッセージをセットします。いずれの場合も、トップページにリダイレクトします。

destroyアクションはログアウトのリンク先となるもので、セッションデータmember_idをクリアして、トップページにリダイレクトします。

■current_memberメソッドの定義

続いて、ApplicationControllerにcurrent_memberメソッドを追加します。このメソッドは、セッションデータ:member_idに値がセットされていれば該当するMemberオブジェクトを返し、そうでなければnilを返す、というものです。

LIST chapter08/app/controllers/application_controller.rb

```
1 class ApplicationController < ActionController::Base
2   private def current_member
3     Member.find_by(id: session[:member_id]) if session[:member_id]
4   end
5   helper_method :current_member
6 end
```

5行目のhelper_methodは、引数に指定された名前のメソッドをテンプレートの中でも使えるメソッド（ヘルパーメソッド）として登録します。

■ログインフォームの実装

Chapter 3で作ったモックアップのログインフォームを実際に使えるものに変えましょう。

LIST chapter08/app/views/shared/_login_form.html.erb

```
1 <h2>ログイン</h2>
2 <% if flash.alert %><p class="alert"><%= flash.alert %></p><% end %>
3 <%= form_tag :session, id: "login_form" do %>
4   <div>
5     <label>ユーザー名：</label>
6     <input type="text" name="name">
7   </div>
8   <div>
9     <label>パスワード：</label>
10    <input type="password" name="password">
11  </div>
12  <div>
```

```
13     <input type="submit" value="ログイン">
14   </div>
15 <% end %>
```

2行目では、ログインに失敗したときのフラッシュのメッセージを表示しています。

3行目のform_tagメソッドでフォームを作ります。フォームデータの送信先は:sessionというシンボルが示しています。SessionsControllerのcreateアクションです。また、formタグのid属性を保持するため、form_tagメソッドの第2引数にオプションを加えています。さらに、6行目と10行目ではユーザー名とパスワードを入力するためのinput要素にname属性を加えました。

サイドバーの部分テンプレートでは、current_memberメソッドを使ってログイン中にはフォームが表示されないようにします。

LIST chapter08/app/views/shared/_sidebar.html.erb

```
1 <%= render "shared/login_form" unless current_member %>
2
3 <h2>最新ニュース</h2>
  （以下省略）
```

ページ上部のヘッダー内には、ログイン中の場合にメンバーの氏名とログアウトのリンクを表示します。

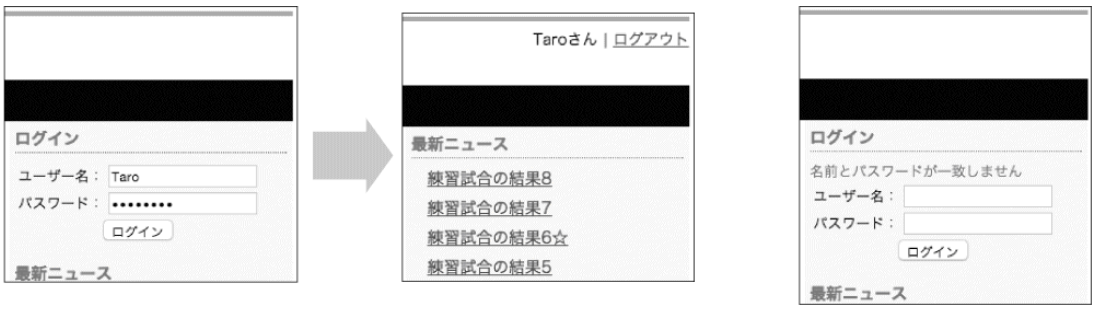
LIST chapter08/app/views/shared/_header.html.erb

```
1 <%= image_tag "logo.gif", size: "272x48", alt: "Morning Glory" %>
2
3 <% if current_member %>
4   <ul class="account-menu">
5     <%= current_member.name + "さん" %>
6     <%= menu_link_to "ログアウト", :session,
```

```
7         method: :delete, data: { confirm: "ログアウトしますか？" } %>
8   </ul>
9 <% end %>
10
11 <nav class="menubar">
   （以下省略）
```

ユーザー名とログアウトリンクの表示位置とスタイルを整えるため、app/assets/stylesheetsディレクトリに新たなスタイルシートheader.cssを追加します。ソースコードの掲載は省略しますので、サンプルソースのchapter08/app/assets/stylesheetsディレクトリからコピーしてください。

サーバーを起動して、ブラウザでログインフォームにユーザー名「Taro」、パスワード「asagao!」を入力してログインしてみましょう。ログイン後には、ログアウトのリンクを試してみましょう。また、間違ったユーザー名とパスワードを入力して、エラーメッセージの確認もしてください。



ログイン前　ログイン後　失敗した場合

ログイン・ログアウト機能

8.3 アクション・コールバックを利用したアクセス制御

ログイン機能ができれば、会員には会員限定のコンテンツを見せ、一般ユーザーにはそれ以外のコンテンツを見せる、といったアクセス制御を行えるようになります。

アクション・コールバックとは

あるコントローラではどのアクションが呼ばれたとしても常に一定の前処理をしておきたいという状況があります。そのようなときには、Railsの**アクション・コールバック**機能を使うと便利です。

アクション・コールバックを設定するための書き方は2つあります。第1の方法では、次のようにブロックを使用します。

```
class ExampleController < ApplicationController
  before_action do
    アクション実行前に行う処理
  end
end
```

このように書くとExampleControllerのすべてのアクションの実行前にブロックの内側に書かれたコードが実行されます。before_actionは、アクション実行前に行う処理を登録するためのクラスメソッドです。ほかにも、after_actionおよびaround_actionという2つのクラスメソッドが使えますが、本書では扱いません。

特定のアクションだけをコールバックの対象としたい場合は、次のようにonlyオプションにアクション名を表すシンボルの配列を指定します。

```
class ExampleController < ApplicationController
  before_action only: [:index, :show] do
    アクション実行前に行う処理
  end
end
```

逆に、指定されたアクションをコールバックの対象から除外したい場合は、onlyオプションの代わりにexceptオプションを指定してください。

第2の方法では、専用のプライベートメソッドを定義して、そのシンボルをコールバック設定メソッドの引数として指定します。

```
class ExampleController < ApplicationController
  before_action :do_something

  private def :do_something
    アクション実行前に行う処理
  end
end
```

この書き方でも第1の方法と同様にonlyオプションとexceptオプションが使えます。

```
before_action :do_something, only: [:index, :show]
before_action :do_another, except: [:index, :search]
```

 会員限定のコンテンツ

では、アクション・コールバックを利用してasagaoにアクセス制限機能を加えます。まず、ApplicationControllerを次のように書き換えてください。

LIST chapter08/app/controllers/application_controller.rb

```
  （省略）
5  helper_method :current_member
6
7  class LoginRequired < StandardError; end
8  class Forbidden < StandardError; end
9
10  private def login_required
11    raise LoginRequired unless current_member
12  end
13 end
```

7行目と8行目で一般的な例外を表すStandardErrorクラスを継承してLoginRequiredクラスとForbiddenクラスを定義し、11行目で例外LoginRequiredを発生させています。プライベートメソッドlogin_requiredはbefore_actionコールバックとして使うために作りました。この例外を捕捉してエラーページを表示する方法についてはChapter 11で解説します。

MembersControllerでは、このlogin_requiredをアクション・コールバックに指定します。こうすれば、ログイン前のユーザーが会員名簿を開こうとしてもエラーになります。

LIST chapter08/app/controllers/members_controller.rb

```
1 class MembersController < ApplicationController
2  before_action :login_required
3
4  # 会員一覧
5  def index
  （以下省略）
```

最後に、ヘッダーのメニューを修正し、「会員名簿」と「管理ページ」のリンクはログインしていないと見えないようにしましょう。

LIST chapter08/app/views/shared/_header.html.erb

```
（省略）
15     <%= menu_link_to "ブログ", "#" %>
16     <% if current_member %>
17       <%= menu_link_to "会員名簿", :members %>
18       <%= menu_link_to "管理ページ", "#" %>
19     <% end %>
20   </ul>
21 </nav>
```

8.4 マイアカウントページの作成

この節では、メンバーが自分自身の情報を編集できる「マイアカウント」ページを作成します。

ルーティングの設定

セッションと同様に「自分のアカウント情報」も単数リソースとして扱うことができます。アプリケーションの中で自分のアカウント情報は多くても1つしかありません。routes.rbを次のように書き換えてください。

LIST chapter08/config/routes.rb

```
（省略）
13  resource :session, only: [:create, :destroy]
14  resource :account, only: [:show, :edit, :update]
15 end
```

これでAccountsControllerへのルーティングが設定されます。show、edit、udpateの3つのアクションしか使わないのでonlyオプションで限定します。

この設定により次のパスとHTTPメソッドでアクションを呼び出せるようになります。

accountリソースのルーティング

アクション	パス	HTTP メソッド	パスを示すシンボル	パスを返すメソッド
show	/account	GET	:account	account_path
edit	/account/edit	GET	:edit_account	edit_account_path
update	/account	PATCH	:account	account_path

ルーティングの一覧を見るには

ターミナルで「bin/rails routes」を実行すると、routes.rbで設定されたパスとHTTPメソッドの一覧を表示できます。ただし、そのまま実行すると出力内容が多すぎて把握しにくいかもしれません。オプション-cにコントローラ名を付けると、そのコントローラへのルーティングのみが表示されます。

```
$ bin/rails routes -c accounts
      Prefix Verb  URI Pattern             Controller#Action
edit_account GET   /account/edit(.:format) accounts#edit
     account GET   /account(.:format)      accounts#show
             PATCH /account(.:format)      accounts#update
             PUT   /account(.:format)      accounts#update
```

AccountsController の作成

単数リソースを扱うAccountsControllerを作成しましょう。

■アカウント情報の表示

「bin/rails g accounts」コマンドでAccountsControllerを作成しましょう。コマンドには「show edit」を加えてテンプレートファイルも同時に作成します。

```
$ bin/rails g controller accounts show edit
```

生成されたaccounts_controller.rbを次のように書き換えます。

LIST chapter08/app/controllers/accounts_controller.rb

```
1 class AccountsController < ApplicationController
2   before_action :login_required
```

```
3
4  def show
5    @member = current_member
6  end
   （以下省略）
```

login_requiredでログインメンバーだけがアクセスできるようにしたうえで、showアクションでインスタンス変数@memberに自分のMemberオブジェクトをセットしています。

showアクションのテンプレートは、MembersControllerのshowアクションのものと内容がだいたい同じなので、部分テンプレートを共用することにします。

まず、app/views/membersディレクトリに新たな部分テンプレート_body.html.erbを作成してください。そして、app/views/members/show.html.erbのtable要素全体を切り取って、_body.html.erbに貼り付けます。

LIST chapter08/app/views/members/_body.html.erb

```
1 <table class="attr">
2   <tr>
3     <th width="150">背番号</th>
4     <td><%= @member.number %></td>
5   </tr>
   （省略）
26   <tr>
27     <th>管理者</th>
28     <td><%= @member.administrator? ? "○" : "－" %></td>
29   </tr>
30 </table>
```

show.html.erbの切り取ったところに、部分テンプレートを埋め込みます。

LIST chapter08/app/views/members/show.html.erb

```
（省略）
5 <div class="toolbar"><%= link_to "編集", [:edit, @member] %></div>
6
7 <%= render "body" %>
```

bin/rails gコマンドにより生成されたshowアクションのテンプレート全体を、次の内容で置き換えます。

LIST chapter08/app/views/accounts/show.html.erb

```
1 <% @page_title = "マイアカウント" %>
2
3 <h1><%= @page_title %></h1>
4
5 <ul class="toolbar">
6   <%= menu_link_to "アカウント情報の編集", :edit_account %>
7 </ul>
8
9 <%= render "members/body" %>
```

6行目でmenu_link_toメソッドの第2引数に指定されているシンボル:edit_accountは、AccountsControllerのeditアクションのパスを示します。9行目では、部分テンプレートを埋め込んでいます。

さらに、サイトのヘッダーを修正して「○○さん」の部分をshowアクションへのリンクにします。

LIST chapter08/app/views/shared/_header.html.erb

```
1 <%= image_tag "logo.gif", size: "272x48", alt: "Morning Glory" %>
2
3 <% if current_member %>
```

```
4   <ul class="account-menu">
5     <%= menu_link_to current_member.name + "さん", :account %>
6     <%= menu_link_to "ログアウト", :session,
7           method: :delete, data: { confirm: "ログアウトしますか？" } %>
8   </ul>
9 <% end %>
    （以下省略）
```

■アカウント情報の編集

準備作業として、app/views/membersディレクトリにある部分テンプレート_form.html.erbを_member_form.html.erbという名前に変えてapp/views/sharedディレクトリに移動します。

```
$ mv app/views/members/_form.html.erb
app/views/shared/_member_form.html.erb
```

そして、app/views/membersディレクトリにあるnew.html.erbを次のように書き換えます。

LIST chapter08/app/views/members/new.html.erb

```
    （省略）
5 <%= form_for @member do |form| %>
6   <%= render "shared/member_form", form: form %>
7   <div><%= form.submit %></div>
8 <% end %>
```

同様に、同じディレクトリにあるedit.html.erbを次のように書き換えます。

LIST chapter08/app/views/members/edit.html.erb

```
（省略）
<%= form_for @member do |form| %>
  <%= render "shared/member_form", form: form %>
  <div><%= form.submit %></div>
<% end %>
```

AccountsControllerのeditアクションのテンプレート全体を、次の内容で置き換えます。

LIST chapter08/app/views/accounts/edit.html.erb

```
<% @page_title = "アカウント情報の編集" %>

<h1><%= @page_title %></h1>

<div class="toolbar"><%= link_to "マイアカウントに戻る", :account %></div>

<%= form_for @member, as: "account", url: :account do |form| %>
  <%= render "shared/member_form", form: form %>
  <div><%= form.submit %></div>
<% end %>
```

form_forメソッドで使われているasオプションとurlオプションについてはChapter 6のコラム「form_forのオプション」で説明しました。パラメータ名をaccountに、送信先のURLを/accountに変更しています。これらのオプションを付けないと、パラメータ名はmemberとなり、送信先のURLは/members/123のようになります。

フォームの中では「render "shared/member_form"」として、MembersController用に作った部分テンプレートを使い回しています。ただし、次の修正を加える必要があります。

LIST chapter08/app/views/shared/_member_form.html.erb

```
（省略）
34   </tr>
35   <% if controller.kind_of?(MembersController) %>
36     <tr>
37       <th><%= Member.human_attribute_name(:administrator) %>
</th>
38       <td>
39         <%= form.check_box :administrator,
40               disabled: !current_member.administrator? %>
41         <%= form.label :administrator %>
42       </td>
43     </tr>
44   <% end %>
45 </table>
```

35行目のcontrollerはコントローラオブジェクトを返すメソッドです。kind_of?でクラスの種類を調べ、MembersControllerの場合だけ管理者フラグのチェックボックスが使えるようにしています。AccountsControllerでは表示されません。

AccountsControllerにeditアクションとupdateアクションを追加します。

LIST chapter08/app/controllers/accounts_controller.rb

```
（省略）
 8  def edit
 9    @member = current_member
10  end
11
12  def update
13    @member = current_member
14    @member.assign_attributes(params[:account])
```

```
15    if @member.save
16      redirect_to :account, notice: "アカウント情報を更新しました。"
17    else
18      render "edit"
19    end
20  end
21 end
```

処理の流れはMembersControllerのupdateアクションとほぼ同じです。

サーバーを起動して、ユーザー名「Jiro」、パスワード「asagao!」でログインしてみましょう。ヘッダーの［Jiroさん］をクリックし、［アカウント情報の編集］をクリックすると、自分自身の情報を編集するフォームが表示されます。

RESULT アカウント情報の編集

8.5 パスワード変更機能

この節では、メンバーが自分自身のパスワードを変更する機能を作ります。また、メンバー追加フォームに初期パスワードを設定するフィールドを追加します。

独立したパスワード変更フォームを作る理由

「自分のパスワード」は「自分のアカウント情報」の一部と言えます。しかし、前節で作成したAccountsControllerの一部としては作りにくいところがあります。アカウント情報の変更フォームの中に単にパスワードの入力欄を置くだけでは、ユーザーを戸惑わせる可能性があります。パスワード入力欄を空にしたまま「更新」ボタンをクリックするとどういう結果を招くのか、必ずしもはっきりしないからです。パスワードが変更されないのか、パスワードが消えるのか、あるいはエラーになるのか、よくわかりません。

説明文をパスワード入力欄の横に加える手もありますが、それよりもパスワード変更フォームを独立させたほうが好ましいです。ユーザーは自分のパスワードを変えたいと思っているのですから、パスワード入力欄を空にしたままではダメであることは明らかです。

また、パスワードを変更する際には、現在の（変更前の）パスワードを入力させたり、新しいパスワードを2回入力させたりしたいものです。このちょっと複雑なユーザーインターフェースをわかりやすい形で提供するためにも、独立したパスワード変更フォームを用意すべきです。

ルーティングの設定

「自分のパスワード」は「自分のアカウント情報」と同様に単数リソースとして扱えます。routes.rbを次のように書き換えてください。

LIST chapter08/config/routes.rb

```
（省略）
13   resource :session, only: [:create, :destroy]
14   resource :account, only: [:show, :edit, :update]
15   resource :password, only: [:show, :edit, :update]
16 end
```

この変更の結果、次のパスとHTTPメソッドでアクションを呼び出せるようになります。

passwordリソースのルーティング

アクション	パス	HTTP メソッド	パスを示すシンボル	パスを返すメソッド
show	/password	GET	:password	password_path
edit	/password/edit	GET	:edit_password	edit_password_path
update	/password	PATCH	:password	password_path

定義された3つのアクションのうち、実質的な意味を持つのはeditとupdateです。前者がパスワードの変更フォームを表示し、後者でパスワードのバリデーションと保存を行います。showアクションでは単にマイアカウントページへのリダイレクションを行います。なぜこのアクションが必要であるかについては、後述します。

Memberモデルの変更

次に、Memberモデルのソースコードを次のように書き換えます。

LIST chapter08/app/models/member.rb

```
（省略）
21   validates :email, email: { allow_blank: true }
22
23   attr_accessor :current_password
24   validates :password, presence: { if: :current_password }
```

```
25
26  class << self
    （以下省略）
```

Chapter 2で学んだクラスメソッドattr_accessorを用いて読み書き可能な属性current_passwordをMemberクラスに追加しています。パスワード変更フォームに「現在のパスワード」を入力する欄を設け、現在のパスワードに関するバリデーションを行うためにこの属性を利用します。

24行目では、password属性に対するバリデーションを設定しています。もともとpassword属性には空文字を禁止するバリデーションが設定されていますが、このバリデーションはオブジェクトが新規作成されるときにしか働きません。ここでは、属性current_passwordに値がセットされている場合には常に「空文字でもnilでもない」ことを確認するようにしています。

パスワード変更フォームの作成

単数リソースを扱うPasswordsControllerを作成しましょう。テンプレートが必要となるのはeditアクションだけです。

```
$ bin/rails g controller passwords edit
```

生成されたpasswords_controller.rbを次のように書き換えます。

LIST chapter08/app/controllers/passwords_controller.rb

```
1 class PasswordsController < ApplicationController
2   before_action :login_required
3
4   def show
5     redirect_to :account
```

```
  end

  def edit
    @member = current_member
  end
  （以下省略）
```

パスワード変更フォームのテンプレートは次のようになります。

LIST chapter08/app/views/passwords/edit.html.erb

```
<% @page_title = "パスワードの変更" %>
<h1><%= @page_title %></h1>

<div class="toolbar"><%= link_to "マイアカウントに戻る", :account %></div>

<%= form_for @member, as: "account", url: :password do |form| %>
  <%= render "shared/errors", obj: @member %>

  <table class="attr">
    <tr>
      <th><%= form.label :current_password %></th>
      <td><%= form.password_field :current_password %></td>
    </tr>
    <tr>
      <th><%= form.label :password %></th>
      <td><%= form.password_field :password %></td>
    </tr>
    <tr>
```

```
19       <th><%= form.label :password_confirmation %></th>
20       <td><%= form.password_field :password_confirmation %></td>
21     </tr>
22   </table>
23
24   <div><%= form.submit "変更" %></div>
25 <% end %>
```

form_forメソッドに付けているasオプションとurlオプションの説明は前節で行ったので、ここでは繰り返しません。

パスワード変更フォームで使用するラベルテキストをロケールファイルに加えます。

LIST chapter08/config/locales/ja.yml

```
1 ja:
2 ␣␣activerecord:
  （省略）
14 ␣␣␣␣␣␣␣␣email: メールアドレス
15 ␣␣␣␣␣␣␣␣administrator: 管理者
16 ␣␣␣␣␣␣␣␣current_password: 現在のパスワード
17 ␣␣␣␣␣␣␣␣password: 新しいパスワード
18 ␣␣␣␣␣␣␣␣password_confirmation: 新しいパスワードの確認
19 ␣␣␣␣errors:
20 ␣␣␣␣␣␣messages:
21 ␣␣␣␣␣␣␣␣invalid_member_name: は半角英数字で入力してください。
```

そして、マイアカウントページにパスワード変更フォームへのリンクを設置します。

LIST chapter08/app/views/accounts/show.html.erb

```
（省略）
<ul class="toolbar">
  <%= menu_link_to "アカウント情報の編集", :edit_account %>
  <%= menu_link_to "パスワードの変更", :edit_password %>
</ul>

<%= render "members/body" %>
```

ブラウザでパスワード変更フォームの表示を確認してください。

RESULT 自分のパスワードの編集

新しいパスワードの保存

続いて、PasswordsControllerのupdateアクションを次のように実装します。

LIST chapter08/app/controllers/passwords_controller.rb

```
（省略）
12  def update
13    @member = current_member
14    current_password = params[:account][:current_password]
15
16    if current_password.present?
17      if @member.authenticate(current_password)
18        @member.assign_attributes(params[:account])
19        if @member.save
20          redirect_to :account, notice: "パスワードを変更しました。"
21        else
22          render "edit"
23        end
24      else
25        @member.errors.add(:current_password, :wrong)
26        render "edit"
27      end
28    else
29      @member.errors.add(:current_password, :empty)
30      render "edit"
31    end
32  end
33 end
```

17行目でブラウザから送られてきた「現在のパスワード」が正しいかどうかを確認しています。正しければパスワードの変更を試み、誤っていればエラーメッセージを出してパスワード入力フォームを再表示します。

パスワードの変更を試みる場合、新しいパスワードが空であったり、新しいパスワードとそのパスワードを確認するための文字列が異なっていたりしたときはバリデーションエラーとなります。新しいパスワードの保存に成功したら、マイアカウントページにリダイレクションします。

エラーメッセージ用のテキストをロケールファイルに加えます。

LIST chapter08/config/locales/ja.yml

```
ja:
  activerecord:
    （省略）
    errors:
      messages:
        invalid_member_name: は半角英数字で入力してください。
        wrong: が正しくありません。
```

ブラウザでパスワード変更フォームを開き、3つのパスワード入力欄にさまざまな組み合わせの文字列を入力して、正しく変更されるか、正しくバリデーションが行われるか、フラッシュメッセージやエラーメッセージが正しく表示されるか、確認してください。

showアクションが必要な理由

updateアクションでバリデーションエラーが発生したとき、ユーザーのブラウザにはパスワード変更フォームが再表示されます。このときブラウザのアドレスバーにはhttp://localhost:3000/passwordというURLが表示されています。ユーザーがこのURLをコピーして別のブラウザのアドレスバーに貼り付けるとどうなるでしょうか。あるいは、このURLをお気に入り登録して、別の機会に訪問したらどうなるでしょうか。

これはGETメソッドでURLパス/passwordにアクセスするということなので、PasswordsControllerのshowアクションが呼び出されます。そして、ユーザーはマイアカウントページへと誘導されます。

このような使い方は稀かもしれませんが、起こりうることです。このように考えると、なぜPasswordsControllerにshowアクションが必要であるかがわかります。もしそれがなかったら、ユーザーがURLをコピペしただけでルーティングエラーが発生し、ユーザーを戸惑わせることになります。

なお、ログインしていない状態でURLパス/passwordにアクセスした場合、例外ApplicationController::LoginRequiredが発生します。これは想定どおりの動きです。

メンバー追加フォームの修正

では、メンバー追加フォームに初期パスワードを入力するフィールドを追加して本章を締めくくることにしましょう。app/views/sharedディレクトリの部分テンプレート_member_form.html.erbを次のように書き換えてください。

LIST chapter08/app/views/shared/_member_form.html.erb

```
（省略）
31  <tr>
32    <th><%= form.label :email, "メールアドレス" %></th>
33    <td><%= form.text_field :email %></td>
34  </tr>
35  <% if @member.new_record? %>
36    <tr>
37      <th><%= form.label :password, "パスワード" %></th>
38      <td><%= form.text_field :password %></td>
39    </tr>
40  <% end %>
41  <% if controller.kind_of?(MembersController) %>
（以下省略）
```

35行目で使われているnew_record?メソッドはモデルオブジェクトがデータベースに保存されていないときにtrueを返します。メンバーを新規登録するときだけパスワードの入力欄が表示されるようにしています。使い勝手を考慮して、ここでは入力内容が隠されない普通のテキスト入力欄としています。

ブラウザでメンバー追加フォームの表示を確認してください。また、実際に新規メンバーを登録し、そのメンバーとしてログインできることを確認してください。

RESULT メンバー追加フォームにパスワード入力フィールドを追加

Chapter 8のまとめ

- ウェブサイトへのログイン・ログアウト機能、マイアカウントページ、パスワード変更機能などを作るには、単数リソースのルーティングを設定します。
- 単数リソースを設定するにはconfig/routes.rbでresourceメソッドを呼び出します。
- 複数のページにわたってブラウザとサーバーの間で維持される接続状態をセッションと呼びます。
- ユーザーをログイン状態にするには、セッションデータにユーザーのidを保存します。保存したセッションデータからユーザーのidを取り出せばユーザーを識別できます。

- **クラスメソッドhas_secure_passwordを用いると、パスワードの保存と認証のためのしくみをモデルクラスに追加できます。**
- **現在ログインしているメンバーを取得するcurrent_memberのようなメソッドをApplicationControllerで定義しておくと会員制サイトを開発しやすくなります。**
- **ログイン中のメンバーだけに特定のページへのアクセスを許可するには、before_actionコールバックでセッションデータを調べます。**

練習問題

［A］　あるRailsアプリケーションで会員のためのモデルクラスUserが次のように定義されているとします。

```
class User < ApplicationRecord
  has_secure_password
end
```

会員がログインする機能をSessionsControllerのcreateアクションで実装するとして、次の空欄を埋めてください。ただし、このアクションには2つのパラメータemailとpasswordが送られてきて、前者が会員のメールアドレス、後者が会員のパスワードを表します。

```
class SessionsController < ApplicationController
  def create
    user = User.find_by(email: params[:email])
    if user&.                    (params[:password])
```

```
      session[:user_id] =
    else
      flash.alert = "メールアドレスとパスワードが一致しません"
    end
    redirect_to :root
  end
end
```

［B］　ApplicationControllerにcurrent_userメソッドを記述して、セッションデータからユーザーのidを取り出し、モデルオブジェクトを取得することにします。current_userメソッドの空欄を埋めてください。

```
class ApplicationController < ActionController::Base
  private def current_user
    User.                    if session[:user_id]
  end
  helper_method :current_user
end
```

［C］　次のように単数リソースを設定したとします。AccountsControllerのshowアクションにリンクするように、link_toメソッドの第2引数を埋めてください。

config/routes.rb

```
resource :account
```

テンプレート

```
<%= link_to "マイアカウント",                    %>
```

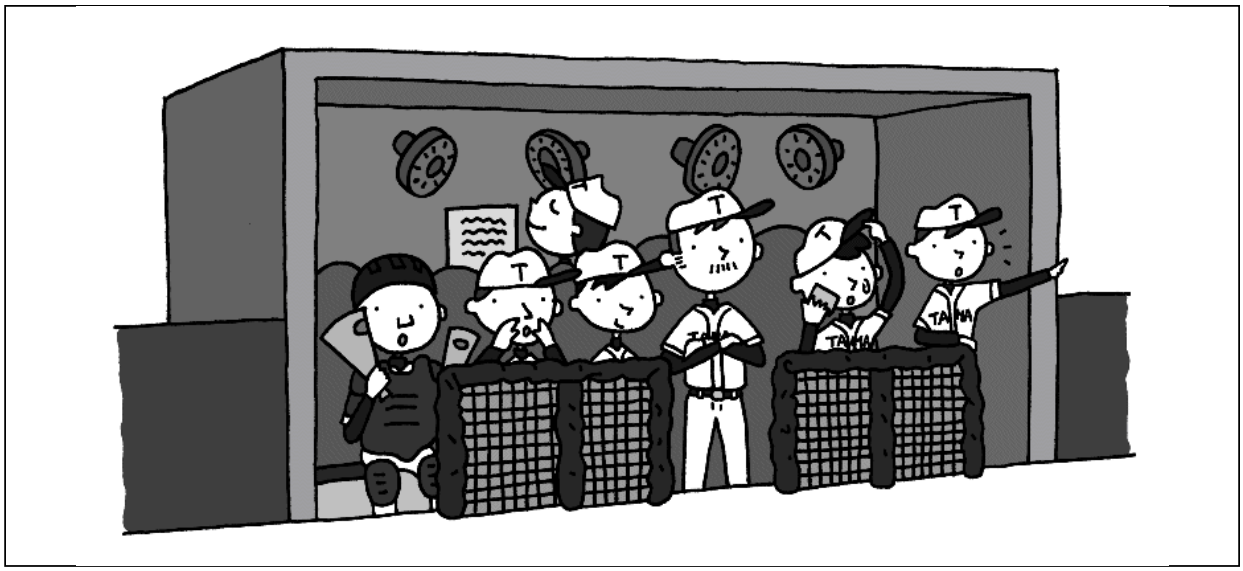
Chapter

9 Active Recordの活用

このChapterではMorning Gloryのサイトに野球チームの活動情報を掲載するために、記事情報を扱う機能をアプリケーションに加えます。この作業を通じて、コールバック、スコープ、ページネーションなどの重要なActive Recordの側面について学習します。

これから学ぶこと

- ニュース記事の表示と編集機能をMorning Gloryサイトに追加します。
- モデルクラスにおいてバリデーションやレコード保存の前後に特定の処理を実行するコールバック機能について学びます。
- モデルにスコープ（レコードの検索の仕方に名前を付けたもの）を定義する方法を学習します。
- Rubyコードによる自由な形式でカスタムバリデーションを設定する方法を学習します。
- ページネーションを実現するGemパッケージkaminariを紹介します。



Railsのモデルにはコールバックとスコープという機能があります。これらを利用すると何が便利になるでしょうか？ また、ページネーションとは何でしょうか？

9.1 ニュース記事の表示と編集

Chapter 8までに学んだ知識を活かしてニュース記事の表示と編集機能の基本的な部分を作りましょう。

Articleモデルの作成

このChapterではMorning Gloryのサイトにニュース記事を配信する機能を加えます。基本的な考え方はChapter 5および6で作った会員情報の管理機能と同じです。記事を収めるテーブル名はarticles、対応するモデル名はArticleとします。そしてリソースarticlesを設定し、ArticlesControllerの7つのアクションで記事の一覧、個別表示、登録、編集、削除を行います。

まずは、「bin/rails g」コマンドでArticleモデルを作りましょう。

```
$ bin/rails g model article
```

db/migrateディレクトリにマイグレーションスクリプトができるので、テーブルのカラムを定義します。

LIST chapter09/db/migrate/20180527021452_create_articles.rb

```
1 class CreateArticles < ActiveRecord::Migration[5.2]
2   def change
3     create_table :articles do |t|
4       t.string :title, null: false          # タイトル
5       t.text :body, null: false             # 本文
6       t.datetime :released_at, null: false  # 掲載開始日時
```

```
    t.datetime :expired_at              # 掲載終了日時
    t.boolean :member_only, null: false, default: false
                                        # 会員のみフラグ
    t.timestamps null: false
  end
 end
end
```

掲載開始日時（released_at）は、その日時になるまで記事をサイトに表示しない、というものです。掲載終了日時（expired_at）は、その日時が過ぎたら記事をサイトに表示しない、というものです。のちほどArticleモデルのクラスには、日付を調べて必要な記事を取り出す機能を加えます。

また、掲載開始日時は必須ですが、掲載終了日時は空（NULL）でも許容しています。member_onlyカラムがtrueである記事は、ログイン中のメンバーだけが閲覧できます。

マイグレーションを実行して、データベースにarticlesテーブルを追加しましょう。

```
$ bin/rails db:migrate
```

ブラウザで確認する開発モードのシードデータを用意しておきましょう。dbディレクトリのseeds.rbで1行目にarticlesを追加します（「4.3　データの保存」の「シードデータの投入」を参照）。

LIST chapter09/db/seeds.rb

```
table_names = %w(members articles)
  （以下省略）
```

db/seeds/developmentディレクトリの下にarticles.rbを作成し、次のように記述します。

LIST chapter09/db/seeds/development/articles.rb

```
body =
  "Morning Gloryが4対2でSunflowerに勝利。¥n¥n" +
  "2回表、6番渡辺の二塁打から7番山田、8番高橋の連続タイムリーで2点先制。" +
  "9回表、ランナー一二塁で2番田中の二塁打で2点を挙げ、ダメを押しました。¥n¥n" +
  "投げては初先発の山本が7回を2失点に抑え、伊藤、中村とつないで逃げ切りました。"

0.upto(9) do |idx|
  Article.create(
    title: "練習試合の結果#{idx}",
    body: body,
    released_at: 8.days.ago.advance(days: idx),
    expired_at: 2.days.ago.advance(days: idx),
    member_only: (idx % 3 == 0)
  )
end
```

2行目と4行目で文字列の末尾に改行文字（¥n）を2つ重ねているのは、そこで段落の境界とするためです。

11行目と12行目では日時を進めるためにadvanceメソッドを使用しています。たとえば8.days.ago.advance(days: 3)は現在日時の8日前の3日後、すなわち5日前の日時を返します。また、2.days.ago.advance(days: 5)は現在日時から3日後の日時を返します。

開発用のデータを投入するため、ターミナルで次のコマンドを実行してください。

```
$ bin/rails db:rebuild
```

バリデーションの追加

Articleモデルに次のようなバリデーションを加えることにしましょう。

- タイトル、本文、掲載開始日時は空ではならない。
- タイトルは80文字を超えてはならない。
- 本文は2000文字を超えてはならない。

ほかにも、掲載開始日時が掲載終了日時よりも前でなくてはならないというバリデーションを加えるべきですが、これについては次節で扱います。

Articleモデルのソースコードを次のように書き換えてください。

LIST chapter09/app/models/article.rb

```
1 class Article < ApplicationRecord
2   validates :title, :body, :released_at, presence: true
3   validates :title, length: { maximum: 80 }
4   validates :body, length: { maximum: 2000 }
5 end
```

また、バリデーションのエラーメッセージが日本語で表示されるように、config/localesディレクトリの下のja.ymlにArticleモデルの属性名を追加しましょう。

LIST chapter09/config/locales/ja.yml

```
1 ja:
2 ␣␣activerecord:
3 ␣␣␣␣models:
4 ␣␣␣␣␣␣member: 会員情報
5 ␣␣␣␣␣␣article: ニュース記事
6 ␣␣␣␣attributes:
7 ␣␣␣␣␣␣member:
```

```
 8 ␣␣␣␣␣␣␣␣␣number: 背番号
   （省略）
19 ␣␣␣␣␣␣␣␣␣password_confirmation: 新しいパスワードの確認
20 ␣␣␣␣␣␣␣article:
21 ␣␣␣␣␣␣␣␣␣title: タイトル
22 ␣␣␣␣␣␣␣␣␣body: 本文
23 ␣␣␣␣␣␣␣␣␣released_at: 掲載開始日時
24 ␣␣␣␣␣␣␣␣␣expired_at: 掲載終了日時
25 ␣␣␣␣␣␣␣␣␣member_only: 会員限定
26 ␣␣␣␣errors:
   （以下省略）
```

では、Railsコンソールを起動し、バリデーションが正しく機能することを確かめましょう。MSYS2/MinGW環境の方は最初のコマンドの前にwinpty rubyを付けてください。

```
$ bin/rails c
irb(main):001:0> article = Article.first
  Member Load (0.6ms)  SELECT  "members".* FROM "members"  （略）
=> #<Article id: 1, title: "練習試合の結果0", body: （略） >
irb(main):002:0> article.title = "A" * 80
=> "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
（略）"
irb(main):003:0> article.valid?
=> true
irb(main):004:0> article.title = "A" * 81
=> "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
（略）"
irb(main):005:0> article.valid?
=> false
```

```
irb(main):006:0> article.errors.full_messages_for(:title)
=> ["タイトルは80文字以内で入力してください"]
```

式"A" * 80は、「A」を80回繰り返す文字列を返します。制限いっぱいの80文字のタイトルの場合はバリデーションが通り、81文字に変えるとバリデーションが失敗しています。

full_messages_forは指定された属性で発生したバリデーションエラーのエラーメッセージを返すメソッドです。バリデーションエラーは複数個発生する可能性があるので、戻り値は配列となります。

ルーティングの設定

config/routes.rbで、記事をリソースとして扱うルーティングを追加しましょう。複数形のsを忘れないように注意してください。

LIST chapter09/config/routes.rb

```
  （省略）
13   resource :session, only: [:create, :destroy]
14   resource :account, only: [:show, :edit, :update]
15   resource :password, only: [:show, :edit, :update]
16
17   resources :articles
18 end
```

念のため、ターミナルでArticleControllerへのルーティングを確認しましょう。

```
$ bin/rails routes -c articles
      Prefix Verb   URI Pattern                Controller#Action
    articles GET    /articles(.:format)        articles#index
             POST   /articles(.:format)        articles#create
```

```
 new_article GET    /articles/new(.:format)      articles#new
edit_article GET    /articles/:id/edit(.:format) articles#edit
     article GET    /articles/:id(.:format)      articles#show
             PATCH  /articles/:id(.:format)      articles#update
             PUT    /articles/:id(.:format)      articles#update
             DELETE /articles/:id(.:format)      articles#destroy
```

ArticlesControllerの作成

では、具体的な開発作業に移ります。まず、ターミナルでArticlesControllerのソースコードを生成します。

```
$ bin/rails g controller articles
```

app/controllersディレクトリにarticles_controller.rbというファイルが作られていますので、それを次のように書き換えてください。

LIST chapter09/app/controllers/articles_controller.rb

```
1 class ArticlesController < ApplicationController
2   before_action :login_required, except: [:index, :show]
  （以下省略）
```

ArtclesControllerには、MembersControllerと同じく7つの基本的なアクションを実装します。そのうち、indexアクションとshowアクション以外は、メンバーとしてログインしていることがアクセスの条件となります。

indexアクション

では、ニュース一覧を表示するindexアクションから作り始めましょう。いったん記事の掲載期間のことは考慮せずに作ります。単に掲載開始日時の新しいものから古いものへ並べ替えるだけです。

LIST chapter09/app/controllers/articles_controller.rb

```
1 class ArticlesController < ApplicationController
2   before_action :login_required, except: [:index, :show]
3
4   # 記事一覧
5   def index
6     @articles = Article.order(released_at: :desc)
7   end
    （以下省略）
```

app/views/articlesディレクトリに新規ファイルindex.html.erbを作成し、次の内容を書き込みます。

LIST chapter09/app/views/articles/index.html.erb

```
1 <% @page_title = "ニュース記事一覧" %>
2 <h1><%= @page_title %></h1>
3
4 <div class="toolbar"><%= link_to "新規作成", :new_article %></div>
5
6 <% if @articles.present? %>
7   <table class="list">
8     <thead>
9       <tr>
10        <th>タイトル</th>
11        <th>日時</th>
```

```
12        <th>操作</th>
13      </tr>
14    </thead>
15    <tbody>
16      <% @articles.each do |article| %>
17        <tr>
18          <td><%= link_to article.title, article %></td>
19          <td><%= article.released_at.strftime("%Y/%m/%d %H:%M") %>
</td>
20          <td>
21            <%= link_to "編集", [:edit, article] %> |
22            <%= link_to "削除", article, method: :delete,
23              data: { confirm: "本当に削除しますか?" } %>
24          </td>
25        </tr>
26      <% end %>
27    </tbody>
28  </table>
29 <% else %>
30   <p>ニュースがありません。</p>
31 <% end %>
```

また、ヘッダーのメニューにある［ニュース］のリンクを修正します。

LIST chapter09/app/views/shared/_header.html.erb

```
    （省略）
13    <%= menu_link_to "TOP", :root %>
14    <%= menu_link_to "ニュース", :articles %>
```

```
15    <%= menu_link_to "ブログ", "#" %>
```

（以下省略）

サーバーを起動し、ブラウザで「http://localhost:3000/」を開いてください。［ニュース］をクリックすると記事一覧が表示されます。

RESULT ニュース記事の一覧

■showアクション

次に、記事の詳細ページを表示するshowアクションをArticlesControllerに追加します。

LIST chapter09/app/controllers/articles_controller.rb

```
（省略）
9  # 記事詳細
10 def show
11   @article = Article.find(params[:id])
```

```
12  end
```

（以下省略）

app/views/articlesディレクトリに新規ファイルshow.html.erbを作成し、次の内容を書き込みます。

LIST chapter09/app/views/articles/show.html.erb

```
1 <% @page_title = @article.title %>
2 <h1><%= @article.title %></h1>
3
4 <% if current_member %>
5   <div class="toolbar"><%= link_to "編集", [:edit, @article] %></div>
6 <% end %>
7
8 <table class="attr">
9   <tr>
10    <th width="100">タイトル</th>
11    <td><%= @article.title %></td>
12  </tr>
13  <tr>
14    <th>本文</th>
15    <td><%= simple_format(@article.body) %></td>
16  </tr>
17  <tr>
18    <th>掲載開始日時</th>
19    <td><%= @article.released_at.strftime("%Y/%m/%d %H:%M") %>
</td>
20  </tr>
21  <tr>
```

```
22     <th>掲載終了日時</th>
23     <td><%= @article.expired_at.try(:strftime, "%Y/%m/%d %H:%M")
%></td>
24   </tr>
25   <tr>
26     <th>会員限定</th>
27     <td><%= @article.member_only? ? "○" : "－" %></td>
28   </tr>
29 </table>
```

15行目で使われているヘルパーメソッドsimple_formatは、引数に与えられた文字列を次のルールに沿って変換します。

- 2個以上の連続する改行を段落の区切りと見なし、各段落を<p>タグで囲む。
- 単独の改行に
タグを追加する。
- 許されていないHTMLタグ（<script>タグや<blink>タグなど）を取り除く。
- HTMLで特殊な意味を持つ記号（&、<、>）をエスケープする。

ブラウザに戻りメンバーとしてログインした状態で、記事一覧のページから記事のタイトルをクリックすると次のように表示されます。

RESULT ニュース記事の詳細ページ

■newアクションとeditアクション

続いて、newアクション（新規作成ページ）とeditアクション（編集ページ）を作成します。

LIST chapter09/app/controllers/articles_controller.rb

```
（省略）
14  # 新規登録フォーム
15  def new
16    @article = Article.new
17  end
18
19  # 編集フォーム
20  def edit
21    @article = Article.find(params[:id])
```

```
22  end
```

（以下省略）

app/views/articlesディレクトリに新規ファイルnew.html.erbを作成し、次の内容を書き込みます。

LIST chapter09/app/views/articles/new.html.erb

```
1 <% @page_title = "ニュース記事の新規登録" %>
2 <h1><%= @page_title %></h1>
3
4 <%= form_for @article do |form| %>
5   <%= render "form", form: form %>
6   <div><%= form.submit %></div>
7 <% end %>
```

app/views/articlesディレクトリに新規ファイルedit.html.erbを作成し、次の内容を書き込みます。

LIST chapter09/app/views/articles/edit.html.erb

```
1 <% @page_title = "ニュース記事の編集" %>
2 <h1><%= @page_title %></h1>
3
4 <p><%= link_to "記事の詳細に戻る", @article %></p>
5
6 <%= form_for @article do |form| %>
7   <%= render "form", form: form %>
8   <div><%= form.submit %></div>
9 <% end %>
```

app/views/articlesディレクトリに新規ファイル_form.html.erbを作成し、次の内容を書き込みます。

LIST chapter09/app/views/articles/_form.html.erb

```
1 <%= render "shared/errors", obj: @article %>
2
3 <table class="attr">
4   <tr>
5     <th><%= form.label :title %></th>
6     <td><%= form.text_field :title, size: 20 %></td>
7   </tr>
8   <tr>
9     <th><%= form.label :body %></th>
10    <td><%= form.text_area :body, rows: 10, cols: 45 %></td>
11  </tr>
12  <tr>
13    <th><%= form.label :released_at, for: "article_released_at_1i" %>
</th>
14    <td><%= form.datetime_select :released_at,
15            start_year: 2000, end_year: Time.current.year + 1,
16            use_month_numbers: true %></td>
17  </tr>
18  <tr>
19    <th><%= form.label :expired_at, for: "article_expired_at_1i" %>
</th>
20    <td>
21      <%= form.datetime_select :expired_at,
22            start_year: 2000, end_year: Time.current.year + 1,
23            use_month_numbers: true %>
```

```
24     </td>
25   </tr>
26   <tr>
27     <th><%= Article.human_attribute_name(:member_only) %></th>
28     <td>
29       <%= form.check_box :member_only %>
30       <%= form.label :member_only %>
31     </td>
32   </tr>
33 </table>
```

ブラウザで記事一覧のページから［新規作成］や［編集］をクリックして、表示を確認しましょう。

RESULT ニュース記事の編集フォーム

■createアクションとupdateアクション

フォームの送信先となるcreateアクションとupdateアクションを実装します。MembersControllerとほぼ同じパターンで作れます。

LIST chapter09/app/controllers/articles_controller.rb

```
（省略）
24  # 新規作成
25  def create
26    @article = Article.new(params[:article])
27    if @article.save
28      redirect_to @article, notice: "ニュース記事を登録しました。"
29    else
30      render "new"
31    end
32  end
33
34  # 更新
35  def update
36    @article = Article.find(params[:id])
37    @article.assign_attributes(params[:article])
38    if @article.save
39      redirect_to @article, notice: "ニュース記事を更新しました。"
40    else
41      render "edit"
42    end
43  end
44 end
```

ブラウザで実際にフォームを送信して、正しく記事を保存できることを確認してください。また、記事のタイトルや本文を空にしたまま送信して、バリデーションエラーが発生することも確

かめてください。

■destroyアクション

最後に、レコードを削除するdestroyアクションを実装します。

LIST chapter09/app/controllers/articles_controller.rb

```
（省略）
45  # 削除
46  def destroy
47    @article = Article.find(params[:id])
48    @article.destroy
49    redirect_to :articles
50  end
51 end
```

ブラウザで記事を削除できることを確認してください。以上で、ArticlesControllerの仮実装が完了しました。

9.2 Active Recordコールバック

Active Recordのモデルにはコールバックという便利な機能が備わっています。

 Active Recordコールバックとは

私たちはChapter 8でアクション・コールバックという概念について学びました。アクション実行の前やあとに決まりきった処理を行いたい場合に利用します。具体的には、ログインしていないユーザーによるアクション実行を禁止するためにbefore_actionコールバックを設定しました。

Railsのモデルにも、バリデーションやレコード保存の前後に指定の処理を実行するコールバック機能があります。コントローラで使用するコールバックと区別したいときは、**Active Recordコールバック**と呼びます。

モデルにコールバックを設定するための書き方は2つあります。第1の方法では、次のようにブロックを使用します。before_saveはオブジェクトをデータベースに保存する前に実行すべき処理を登録するためのクラスメソッドです。

```
class Article < ApplicationRecord
  before_save do
    記事を保存する前に行う処理
  end
end
```

第2の方法では、専用のプライベートメソッドを定義して、そのシンボルをコールバック設定メソッドの引数として指定します。

```
class Article < ApplicationRecord
  before_save :do_something

  private def do_something
    記事を保存する前に行う処理
  end
end
```

Active Recordコールバックを設定するためのおもなクラスメソッドは次のとおりです。

メソッド	実行されるタイミング
before_validation	バリデーション前
after_validation	バリデーション後
before_save	保存前（新規作成、更新の両方）
before_create	新規作成前
before_update	更新前
after_update	更新後
after_create	新規作成後
after_save	保存後（新規作成、更新の両方）

before_validationとafter_validationについては、次のようにonオプションに:createまたは:updateを指定すると、新規作成または更新の場合だけコールバックが呼ばれます。

```
before_validation on: :create do
  バリデーションの前に行う処理
end
```

saveメソッドを実行すると、新規作成の場合はbefore_save→before_create→after_create→after_saveのメソッドが順に呼ばれます。

更新の場合には、before_save→before_update→after_update→after_saveの順になります。

また、レコードの削除（destroyメソッド）の前後に実行するメソッドを指定する、before_destroyとafter_destroyもあります（本書では解説しません）。

no_expiration属性

前節で作成した記事の投稿・編集フォームでは、掲載終了日時を空にすることができず不便です。そこで、記事を投稿するフォームにチェックボックス［掲載終了日時を設定しない］を設置し、ユーザーがこれにチェックを付けたら掲載終了日時を保存しないことにします。

このチェックボックスに対応する属性no_expirationは、articlesテーブルのカラムではなく、クラスの普通の属性として実装します。Articleクラスにno_expirationのアクセサメソッドを追加します。

LIST chapter09/app/models/article.rb

```
（省略）
def no_expiration
  expired_at.nil?
end

def no_expiration=(val)
  @no_expiration = val.in?([true, "1"])
end
（以下省略）
```

no_expirationメソッドは、expired_atの有無によってfalseかtrueを返します。等号付きのno_expiration=メソッドは、引数がtrueか文字列の「1」であれば変数@no_expiration

をtrueにし、そうでなければfalseにします。

ブラウザはHTMLフォームのチェックボックスにチェックが付いていなければ文字列の「0」を、チェックが付いていれば文字列の「1」を送ってきます。したがって、11行目では引数が文字列の「1」であるかどうかを調べれば十分です。しかし、Railsコンソールで article.no_expiration = trueと入力するような用途を考慮し、val.in?([true, "1"])と記述しています。

テーブルのカラムにないフィールドを作る

上記のno_expirationのように、テーブルのカラムにないフィールドをフォームに加える必要が出てくることがあります。そうしたときは、フィールド名と同名のアクセサメソッドをモデルクラスに加えて属性を作ります。

Chapter 13では、クラスメソッドattributeを用いて属性を作る別の方法を紹介します。

コールバックを使って掲載終了日時を消す

チェックボックス［掲載終了日時を設定しない］がオンの場合（no_expiration属性がtrueの場合）は、expired_at属性をnilにするコードを加えましょう。このとき、モデルのコールバックを用いると実装しやすいです。

LIST chapter09-2/app/models/article.rb

```
（省略）
14  before_validation do
15    self.expired_at = nil if @no_expiration
16  end
（以下省略）
```

15行目のコードはバリデーションが行われる前に実行されます。変数@no_expirationが真であれば（nilでもfalseでもなければ）属性expired_atの値をnilにしています。

フォームの書き換え

記事の投稿・編集フォームに［掲載終了日時を設定しない］というチェックボックスを加えましょう。

LIST chapter09/app/views/articles/_form.html.erb

```
（省略）
20    <td>
21      <div>
22        <%= form.check_box :no_expiration %>
23        <%= form.label :no_expiration %>
24      </div>
25      <div>
26        <%= form.datetime_select :expired_at,
27              start_year: 2000, end_year: Time.current.year + 1,
28              use_month_numbers: true %>
29      </div>
30    </td>
（以下省略）
```

また、ラベルを日本語化するためにロケールデータに項目を加えます。

LIST chapter09/config/locales/ja.yml

```
（省略）
20 ␣␣␣␣␣␣article:
21 ␣␣␣␣␣␣␣␣title: タイトル
22 ␣␣␣␣␣␣␣␣body: 本文
23 ␣␣␣␣␣␣␣␣released_at: 掲載開始日時
24 ␣␣␣␣␣␣␣␣expired_at: 掲載終了日時
```

```
25 ␣␣␣␣␣␣␣␣␣no_expiration: 掲載終了日時を設定しない
26 ␣␣␣␣␣␣␣␣␣member_only: 会員限定
27 ␣␣␣␣errors:
   （以下省略）
```

ブラウザで記事一覧のページから［新規作成］や［編集］をクリックして、チェックボックスが表示されることを確認してください。また、チェックボックスをオンにしたときに、掲載終了日時が設定されないことを確かめてください。

RESULT チェックボックスが追加された

「12.4　JavaScript」では、チェックボックスのオン・オフ切り替えに応じて掲載終了日時の入力欄の表示・非表示を切り替える機能を作成します。

9.3 スコープ

Railsのモデルには、レコードの検索をシンプルに書き表すためにスコープの機能が用意されています。この節ではArticleモデルにスコープを追加して、ある条件に合致する記事を取り出します。

スコープの記述

スコープは、レコードの検索の仕方に名前を付けたものです。モデルクラスの中で「scope :スコープ名, -> { クエリーメソッド }」のように記述します。たとえば、open_to_the_publicというスコープを作って、一般公開の（会員限定ではない）記事を取り出す機能を加えたければ、次のように記述します。

```
class Article
  scope :open_to_the_public, -> { where(member_only: false }
```

-> { }という記法については次のHINT「Procオブジェクト」で説明しますが、スコープのこの記述法は「一種の公式」として丸暗記してもよいでしょう。

なお、記号->の右側に付く中かっこのペアはdoとendで置き換えることができます。つまり、上記のコードは次のようにも書けます。

```
class Article
  scope :open_to_the_public, -> do
    where(member_only: false)
  end
```

Procオブジェクト

Rubyの-> { }は、Procオブジェクトを作成する記法です。Procは「コードのかたまり」を表すオブジェクトです。プログラミング用語では「無名関数」と呼ばれます。次の例をご覧ください。

```
p = -> { Math.sqrt rand(100) }
puts p.call
```

Math.sqrtは引数の平方根を返すメソッドで、rand(100)は0から99までの整数をランダムに返します。{ }で囲まれたコード全体がProcオブジェクトになり、変数pにセットされます。Procオブジェクトのコードを呼び出すには、callメソッドを用います。結果としてp.callという式は0以上10未満の浮動小数点数を返します。

引数を取るProcオブジェクトを作ることもできます。次の例では、Procオブジェクトは引数nを取ります。

```
p = -> (n) { Math.sqrt rand(n) }
puts p.call(100)
```

-> { }の代わりにlambdaメソッドにブロックを渡すこともできます。次の例は、上の例と同じ結果になります。

```
p = lambda { |n| Math.sqrt rand(n) }
puts p.call(100)
```

open_to_the_publicスコープを使うには、次のように記述します。open_to_the_publicはArticleモデルのクラスメソッドになり、リレーションオブジェクトを返すようになります。Article.where(member_only: false).order(released_at: :desc)と同じ結果になります。

```
articles = Article.open_to_the_public.order(released_at: :desc)
```

次のように、リレーションオブジェクトからスコープのメソッドを呼び出しても同じ結果になります。

```
articles = Article.order(released_at: :desc).open_to_the_public
```

スコープの定義

では、実際にスコープを定義してみましょう。Articleモデルにopen_to_the_publicおよびvisibleという2つのスコープを加えます。前者についてはすでに説明しました。後者は、現在日時が掲載開始日時と掲載終了日時の間にある記事だけを取り出すためのスコープです。Articleモデルのソースコードを次のように変更してください。

LIST chapter09/app/models/article.rb

```
  （省略）
20      errors.add(:expired_at, :expired_at_too_old)
21    end
22  end
23
24  scope :open_to_the_public, -> { where(member_only: false) }
25
26  scope :visible, -> do
27    now = Time.current
28
29    where("released_at <= ?", now)
30      .where("expired_at > ? OR expired_at IS NULL", now)
31  end
32 end
```

visibleスコープの中身は複数行にわたるので、読みやすさを考慮してdoとendで囲みました。まず、変数nowに現在日時をセットしてから、whereメソッドを重ね合わせて検索条件を作っています。2番目のwhereメソッドでは、掲載終了日時が現在日時よりもあとであるか掲載終了日時がセットされていないという条件を表現しています。

サイドバーでの記事表示

定義された2つのスコープをサイトの中で使ってみましょう。サイドバーに最新のニュース記事のタイトルを表示することにします。サイドバー用の部分テンプレート_sidebar.html.erbを次のように書き換えてください。

LIST chapter09/app/views/shared/_sidebar.html.erb

```
1 <%= render "shared/login_form" %>
2
3 <h2>最新ニュース</h2>
4 <%
5   articles = Article.visible.order(released_at: :desc).limit(5)
6   articles = articles.open_to_the_public unless current_member
7 %>
8 <ul>
9   <% articles.each do |article| %>
10    <li><%= link_to article.title, article %></li>
11  <% end %>
12 </ul>
   （以下省略）
```

5行目と6行目でローカル変数articlesにサイドバーで表示すべき記事のリストをセットしています。Article.visibleで記事の範囲を狭めたあとで、orderメソッドでソートし、limitメソッド

で取得するレコード数の上限を定めています。そして、ユーザーがログインしていない状態では、スコープopen_to_the_publicでさらに限定しています。

この変更により、サイドバーは次のような表示になります。

RESULT サイドバーの記事リスト

ニュース記事一覧ページの変更

次に、ニュース一覧ページに表示する記事についても、ログインしていないユーザーにはmember_onlyフラグの立っている記事を見せないことにします。また、管理者以外のメンバーや訪問者には、現在日時が掲載開始日時と掲載終了日時の間にある記事だけを見せることにしましょう。ArticlesControllerのindexアクションを次のように書き換えてください。

LIST chapter09/app/controllers/articles_controller.rb

```
（省略）
4  # 記事一覧
5  def index
6    @articles = Article.order(released_at: :desc)
7
8    @articles = @articles.open_to_the_public unless current_member
9
10   unless current_member&.administrator?
11     @articles = @articles.visible
12   end
```

```
13  end
    （以下省略）
```

条件式current_member&.administrator?は、ユーザーがログインしていてかつ管理者であるときにtrueを返します。ぼっち演算子（&.）については、「5.1　RESTとルーティング」のHINT「&.演算子」を参照してください。

ログインしない状態、一般メンバーとしてログインした状態、および管理者としてログインした状態でニュース一覧がどのように変化するか観察してください。

ニュース記事詳細ページの変更

続いて、ニュース記事詳細ページにアクセス制限を加えます。考え方は、ニュース一覧ページと同じです。ArticlesControllerのshowアクションを次のように書き換えてください。

LIST chapter09/app/controllers/articles_controller.rb

```
    （省略）
15  # 記事詳細
16  def show
17    articles = Article.all
18
19    articles = articles.open_to_the_public unless current_member
20
21    unless current_member&.administrator?
22      articles = articles.visible
23    end
24
25    @article = articles.find(params[:id])
```

```
26  end
```

（以下省略）

訪問者や一般メンバーが閲覧権限のないページにアクセスすると、例外ActiveRecord::RecordNotFoundが発生します。

TopControllerの修正

ニュース記事は、Morning Gloryのトップページにも掲載します。TopControllerを修正して、記事を表示する機能を加えましょう。トップページでもopen_to_the_publiicスコープとvisibleスコープを使って、記事を限定します。

LIST chapter09/app/controllers/top_controller.rb

```
1 class TopController < ApplicationController
2   def index
3     @articles = Article.visible.order(released_at: :desc).limit(5)
4     @articles = @articles.open_to_the_public unless current_member
5   end
```

（以下省略）

トップページ用のテンプレートも書き換えます。

LIST chapter09/app/views/top/index.html.erb

```
1 <% @articles.each do |article| %>
2   <h2><%= article.title %></h2>
3   <p>
4     <%= truncate article.body, length: 80 %>
5     <%= link_to "もっと読む", article %>
```

```
  </p>
<% end %>
```

4行目で使われているヘルパーメソッドtruncateは引数に指定された文字列がある長さを超えていれば切って、省略記号（...）を加えて返します。デフォルトの長さは30文字です。lengthオプションで長さを指定できます。

validate do ... end

最後に掲載終了日時（expired_at属性）に関するバリデーションを設定します。

LIST chapter09/app/models/article.rb

```
（省略）
  before_validation do
    self.expired_at = nil if @no_expiration
  end

  validate do
    if expired_at && expired_at < released_at
      errors.add(:expired_at, :expired_at_too_old)
    end
  end

  scope :open_to_the_public, -> { where(member_only: false) }
（以下省略）
```

これまでバリデーションの設定に利用してきたのはクラスメソッドvalidatesです。このクラスメソッドは、属性名をシンボルで指定してオプションでバリデーションの種類や方式を指定します。

ここでは名前の末尾に「s」のないクラスメソッドvalidateを使用しています。このメソッドはブロックを取り、ブロックの中で自由なRubyコードによりバリデーションのやり方を記述することができます。ブロックの中身をご覧ください。

```
if expired_at && expired_at < released_at
  errors.add(:expired_at, :expired_at_too_old)
end
```

掲載終了日時が設定されていて、それが掲載開始日時よりも前の時点であればエラーとしています。メソッドerrorsはモデルのエラーオブジェクトを返します（Chapter 7を参照）。エラーオブジェクトのaddメソッドを呼び出すとモデルオブジェクトにバリデーションエラーが登録されます。

addメソッドの第1引数には属性名のシンボル、第2引数にはエラーの種類を示すシンボルを指定します。第2引数には、:emptyや:invalidなどの既定のものだけでなく、開発者自身が決めた任意のシンボルを指定できます。その場合は自分でエラーメッセージをロケールテキストに加える必要があります。

LIST chapter09/config/locales/ja.yml

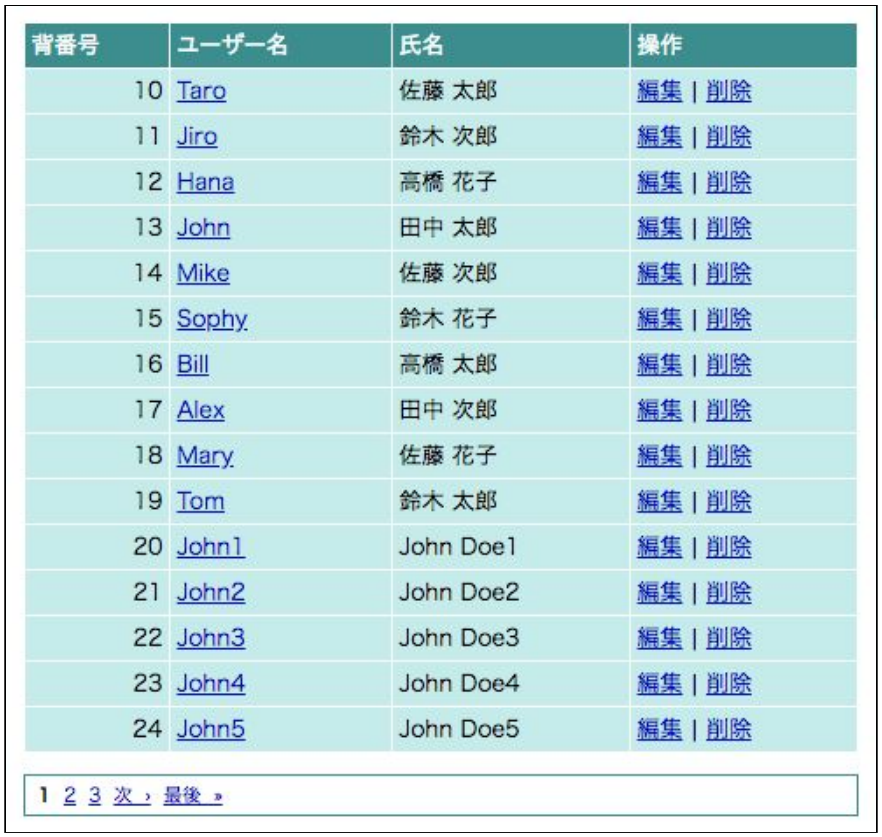
```
    （省略）
 27 ␣␣␣␣errors:
 28 ␣␣␣␣␣␣messages:
 29 ␣␣␣␣␣␣␣␣invalid_member_name: "は半角英数字で入力してください。"
 30 ␣␣␣␣␣␣␣␣wrong: が正しくありません。
 31 ␣␣␣␣␣␣␣␣expired_at_too_old: は掲載開始日より新しい日時にしてください。
```

9.4 ページネーション

データベースに収めたデータの数が多くなってくると、その一覧を表示するときに工夫が必要になってきます。ここでは、ページネーションの機能を導入して、一覧表示を複数のページに分ける方法を紹介します。

 Gemパッケージkaminari

ウェブページで大量の項目を一覧表示するときは、項目を複数のページに分けて見せるのが一般的です。Googleのような検索サイトの検索結果がよい例です。1ページに決まった数の検索結果が並び、ページ下部の「1 2 3 4 5……」のようなリンクをクリックすると、20番目以降、30番目以降、……の結果を見ることができます。こうしたしくみを**ページネーション**と呼びます。



背番号	ユーザー名	氏名	操作
10	Taro	佐藤 太郎	編集 | 削除
11	Jiro	鈴木 次郎	編集 | 削除
12	Hana	高橋 花子	編集 | 削除
13	John	田中 太郎	編集 | 削除
14	Mike	佐藤 次郎	編集 | 削除
15	Sophy	鈴木 花子	編集 | 削除
16	Bill	高橋 太郎	編集 | 削除
17	Alex	田中 次郎	編集 | 削除
18	Mary	佐藤 花子	編集 | 削除
19	Tom	鈴木 太郎	編集 | 削除
20	John1	John Doe1	編集 | 削除
21	John2	John Doe2	編集 | 削除
22	John3	John Doe3	編集 | 削除
23	John4	John Doe4	編集 | 削除
24	John5	John Doe5	編集 | 削除

1 2 3 次 › 最後 »

ページネーション

Rails自体にはページネーションの機能が含まれていないので、Gemパッケージを使う必要があります。ページネーション機能のGemパッケージはいくつかありますが、本書ではkaminariを使います。GemfileにkaminariとKaminari-i18nの指定を加えてください。

LIST chapter09/Gemfile

```
（省略）
35 gem 'email_validator', '~> 1.6'
36 gem 'rails-i18n'
37 gem 'kaminari'
38 gem 'kaminari-i18n'
39
40 group :development, :test do
（以下省略）
```

Bundlerを使ってkaminariとkaminari-i18nをインストールします。

```
$ bundle install
```

kaminariを導入すると、モデルのクエリーメソッドにpageメソッドが追加されます。pageメソッドには、現在のページ数を指定します。

```
@members = Member.page(2)
```

デフォルトで1ページあたり25件のレコードが取り出されます。この値を変更したければ、perメソッドで指定します。

```
@members = Member.page(2).per(10)
```

テンプレートでpaginateメソッドを使えば、ページネーションのリンクを埋め込めます。paginateメソッドには、pageメソッドで取り出したリレーションオブジェクトを渡します。

```
<%= paginate @members %>
```

ヘルパーメソッドpaginateは、次のようなHTMLを生成します。

```
<nav class="pagination" role="navigation" aria-label="pager">
  <span class="page current">1</span>
  <span class="page">
    <a rel="next" href="/members?page=2">2</a>
  </span>
  <span class="page">
    <a href="/members?page=3">3</a>
  </span>
```

```
  <span class="next">
    <a rel="next" href="/members?page=2">次 &rsaquo;</a>
  </span>
  <span class="last">
    <a href="/members?page=3">最後 &raquo;</a>
  </span>
</nav>
```

リンクにはpageパラメータが付くので、コントローラの中でpaginateメソッドに「page: params[:page]」とページ数を指定すれば、ユーザーがクリックしたページ番号に合わせてページネーションが動作するようになります。

ページネーション機能の実装

kaminariを使って会員情報とニュース記事の一覧にページネーションの機能を加えましょう。

■会員一覧

MembersControllerのindexアクションとsearchアクションを修正して、paginateメソッドを加えましょう。1ページあたりの件数は15とします。

LIST chapter09/app/controllers/members_controller.rb

```
（省略）
4  # 会員一覧
5  def index
6    @members = Member.order("number")
7      .page(params[:page]).per(15)
8  end
```

```
 9
10  # 検索
11  def search
12    @members = Member.search(params[:q])
13      .page(params[:page]).per(15)
14
15    render "index"
16  end
    （以下省略）
```

indexアクションのテンプレートでは、テーブルの下にpaginateメソッドを挿入して、ページネーションのリンクを作ります。

LIST chapter09/app/views/members/index.html.erb

```
    （省略）
35    </table>
36    <%= paginate @members %>
37  <% else %>
38    <p>会員情報がありません。</p>
39  <% end %>
```

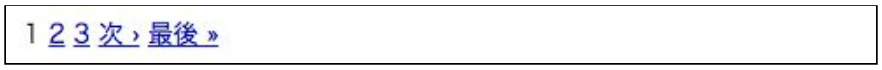
現在のシードデータでは、会員が10人しかいないので、ページネーションを確認できるように30人分のデータを追加します。ページネーションの確認に使うだけのものなので、ユーザー名や氏名は適当なものにします。

LIST chapter09/db/seeds/development/members.rb

```
    （省略）
14    password_confirmation: "password"
15  )
```

```
16 end
17
18 0.upto(29) do |idx|
19   Member.create(
20     number: idx + 20,
21     name: "John#{idx + 1}",
22     full_name: "John Doe#{idx + 1}",
23     email: "John#{idx+1}@example.com",
24     birthday: "1981-12-01",
25     sex: 1,
26     administrator: false,
27     password: "password",
28     password_confirmation: "password"
29   )
30 end
```

bin/rails db:rebuildコマンドでデータベースを作り直したら、会員の一覧を確認しましょう。ページネーションのリンクは次のような表示になります。



RESULT CSS指定前のページネーション

CSSでページネーションのリンクをデザインしましょう。app/assets/stylesheetsディレクトリにファイルpagination.cssを次の内容で作成してください。

LIST chapter09/app/assets/stylesheets/pagination.css

```
1 nav.pagination {
2   font-size: 75%;
3   padding: 4px 8px;
4   border: 1px solid #499;
```

```
  word-spacing: 4px;
}

nav.pagination span.current {
  font-weight: bold;
}
```

class属性がpaginationのnavタグがリンクを囲むタグです。class属性がcurrentのspanタグは、現在のページ番号を囲むものです。このスタイル追加によりページネーションのリンクは次の表示に変わります。

RESULT CSS指定後のページネーション

■ニュース記事一覧

ArticlesControllerのindexアクションにもページネーションを適用しましょう。1ページあたりの件数は5とします。

LIST chapter09/app/controllers/articles_controller.rb

```
（省略）
  def index
    @articles = Article.order(released_at: :desc)

    @articles = @articles.open_to_the_public unless current_member

    unless current_member&.administrator?
      @articles = @articles.visible
    end

```

```
14     @articles = @articles.page(params[:page]).per(5)
15   end
     （以下省略）
```

indexアクションのテンプレートでは、table要素の下にpaginateメソッドを加えます。

LIST chapter09/app/views/articles/index.html.erb

```
     （省略）
27     <% end %>
28   </table>
29   <%= paginate @articles %>
30 <% else %>
31   <p>ニュースがありません。</p>
32 <% end %>
```

会員情報と同様に、シードデータに記事を30件追加します。

LIST chapter09/db/seeds/development/articles.rb

```
     （省略）
13     member_only: (idx % 3 == 0)
14   )
15 end
16
17 0.upto(29) do |idx|
18   Article.create(
19     title: "Article#{idx+10}",
20     body: "blah, blah, blah...",
21     released_at: 100.days.ago.advance(days: idx),
22     expired_at: nil,
```

```
23     member_only: false
24   )
25 end
```

bin/rails db:rebuildコマンドでデータベースを作り直すと、ニュース記事の一覧では次のようにページネーションのリンクが表示されます。

RESULT ニュース記事のページネーション

Chapter 9のまとめ

- モデルでバリデーションやレコード保存の前後に実施したい処理のことを**ActiveRecordコールバック**と呼びます。
- バリデーションの前に特定の処理を実行したい場合は、クラスメソッドbefore_validationに続くブロックの中にその処理を記述します。
- モデルの**スコープ**とは、レコードの検索の仕方に名前を付けたものです。クラスメソッドscopeを用いて定義します。

- **クラスメソッドvalidateに続くブロックの中でエラーオブジェクトを操作することにより、自由な書き方でカスタムバリデーションを設定できます。**
- **一覧ページにページネーション機能を導入するには、Gemパッケージkaminariを使うと便利です。**

練習問題

［A］　次に示すのはlogディレクトリのログファイルbooks.logに「Hello.」というメッセージを書き込むためのコードです。

```
Logger.new(Rails.root.join("log/books.log")).info("Hello.")
```

　Bookモデルがレコード保存後に書名を同じログファイルに書き込むようにするにはどのように書けばよいですか。空欄を埋めてください。

```
class Book < ApplicationRecord
  [          ] do
    Logger.new(Rails.root.join("log/books.log")).info([          ])
  end
end
```

［B］　Bookモデルに「価格が0円である」という検索条件を示すスコープfreeを設定します。空欄を埋めてください。

```
class Book < ApplicationRecord
  [          ] :free, -> { [                    ] }
end
```

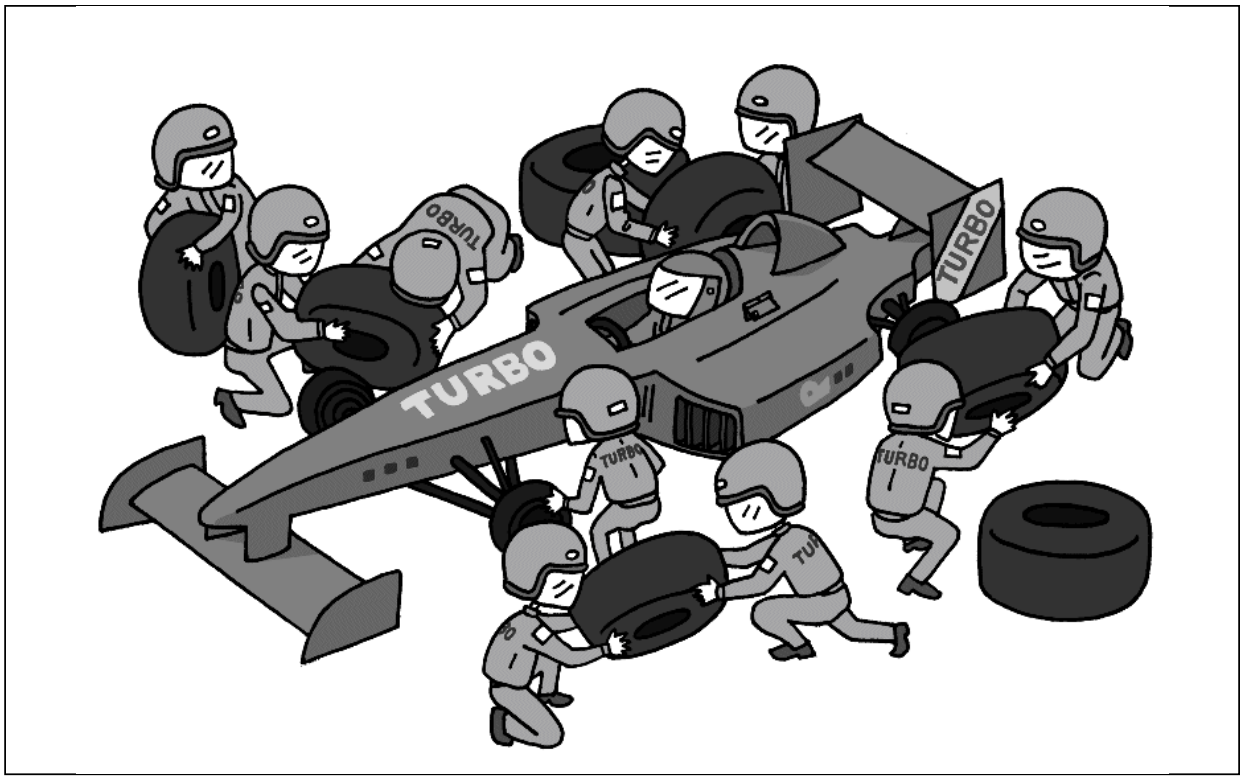
Chapter

10　モデル間の関連付け

実用的なウェブアプリケーションでは、データベースに複数のテーブルを用意して、テーブル間でデータを結び付けます。Railsはモデル間の関連付けを自然な形で表現できるため、複数のテーブルを同時に扱うプログラムを効率よく作成できます。

これから学ぶこと

- モデル間の関連付けと外部キーの役割について学びます。
- クラスメソッドhas_many、belongs_toを使って1対多の関連付けを作る方法を学びます。
- 1対多の関連付けを使って会員のブログ機能を作ります。



モデルとモデルが1対多で関連付けられたとき、それらの関係は自動車と車輪の関係にたとえることができます。関連付けはどのように設定すればよいでしょうか？ 関連付けによってどんなことができるようになるでしょうか？

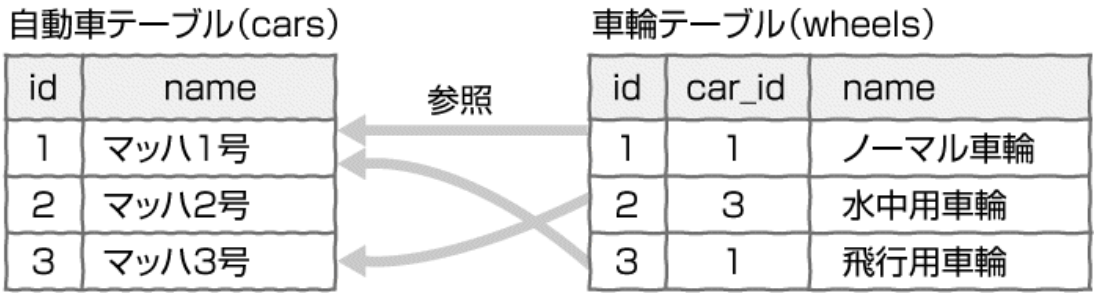
10.1 関連付けの概要

実用的なウェブアプリケーションであれば、モデル間の関連付けを扱うことになります。ここでは、関連付けに対するRailsの基本的アプローチを紹介します。

モデル間の関連付けと外部キー

自動車を例に、モデル間の**関連付け**（association）について考えてみましょう。データベースの中で、複数の自動車を1つのテーブルに収め、複数の車輪を別のテーブルに収めるとします。ある車輪がどの自動車に対するものなのかを表すには、自動車テーブルと車輪テーブルを外部キーを使って結び付けます。

自動車テーブルには、個々のレコードを識別するidカラム（主キー）があります。車輪テーブル内のレコードのほうにはcar_idカラムを持たせ、自動車テーブルのidカラムと同じ数値を格納すれば、自動車と車輪の結び付きを表すことができます。このように別のテーブルの主キーを参照するカラムのことを**外部キー**（foreign key）と呼びます。



外部キーによる参照

変数@carにモデルクラスCarのインスタンスがセットされているとき、次の式はその車輪の集合を表すリレーションオブジェクトを返します。

```
Wheel.where(car_id: @car.id)
```

しかし、Railsにはもっと直感的な書き方が用意されています。

```
@car.wheels
```

wheelsはCarクラスのインスタンスメソッドで、リレーションオブジェクトを返します。where、order、firstなどのメソッドを連結して呼び出すことができます。

```
@car.wheels.where(color: "red").order(:created_at).first
```

ただし、単にCarとWheelという2つのモデルがあって、wheelsテーブルにcar_idカラムがあるだけでは、このwheelsメソッドは使えません。Carクラスの中で次のように書く必要があります。

```
has_many :wheels
```

has_manyはモデル間の関連付けを指定するメソッドです。これによりCarモデルとWheelモデルの間に1対多の関連付けが設定され、Carモデルにインスタンスメソッドwheelsが追加されます。

relation、association、relationship

データベース用語の「リレーション」（relation）は、日常語の「リレーション」とは違う意味です。データベース用語では行と列という構造を持つデータを「リレーション」と言います。Excelシートの一部分とだいたい同じようなものだと思ってください。ただし、同じ列の値はすべて同じ型を持つという制約があります。Railsのリレーションオブジェクトは、この「リレーション」を保持するオブジェクトです。

一方で「関連付け」（association）は、ソフトウェア設計の分野で使われる用語です。2つのクラス（モデル）がお互いにどのように結び付いているのかを示します。ポイントになるのは、相手が単数か複数かということです。本文で例として挙げたCarモデルとWheelモデルは、「1対多」の関連付けを持ちます。「1個の自動車に複数の車輪が付く」ということを「1対多」と表現するわけです。

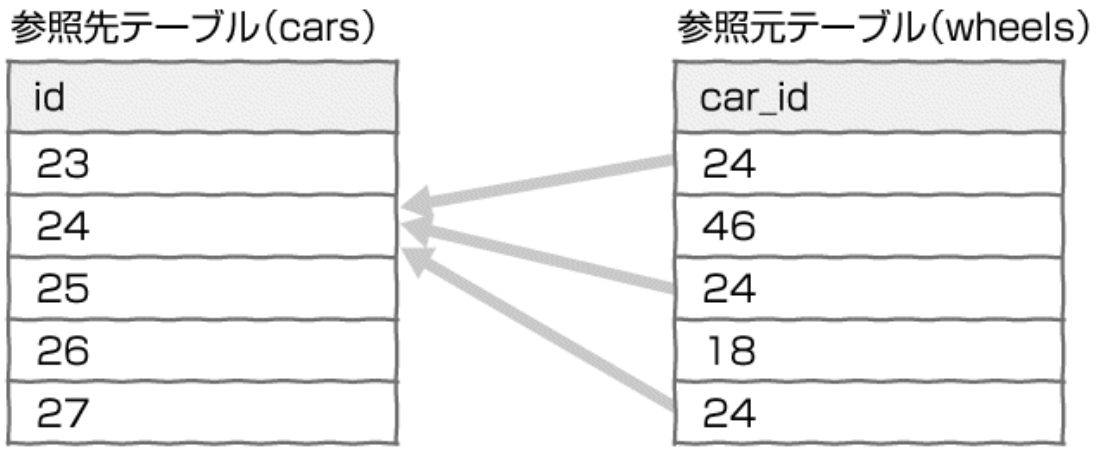
しかし、ややこしいことに「関連付け」とほぼ同義語として「リレーションシップ」（relationship）という用語が使われることもあり、普通は「関連」と訳しますが、「関係」と訳す場合もあります。本書では「行と列という構造を持つデータ」を「リレーション」と呼び、モデル同士の結び付きは「関連付け」で統一することにします。

関連付けを作るメソッド

Railsでは、モデル間の関連付けをモデルクラスのメソッドhas_manyおよびbelongs_toで作ります。自動車を表すモデルを例にして、この2つのメソッドの使い方を紹介しましょう。

■1対多の関連付け

has_manyは、**1対多**の関連付け、つまりテーブルの複数のレコードが別のテーブルのレコード1つを参照する結び付きを作ります。自動車と車輪を例にすると、自動車には複数の車輪があるので、図のようにwheelsテーブルの複数のレコードがcarsテーブルを参照しています。



has_many

参照先のモデルクラスでhas_many（～をたくさん持つ）を使えば、1対多の関連付けを指定できます。has_manyに渡す名前は複数形にします。

```
class Car < ApplicationRecord
  has_many :wheels
end
```

参照元では、belongs_to（～に属する）を使います。

```
class Wheel < ApplicationRecord
  belongs_to :car
end
```

これにより、@car.wheelsで参照元のモデルオブジェクトの集合を取り出したり、@wheel.carで参照先のモデルオブジェクトを取り出したりできます。

車輪を作成し、自動車に関連付けて保存するには、次のように記述します。

```
@wheel = Wheel.new
@wheel.car = @car
@wheel.save
```

逆に、自動車のほうから車輪を関連付けるには、<<で追加します。<<を使うと、関連付けと車輪のレコードの保存が同時に行われます。

```
@car.wheels << @wheel
```

車輪を自動車に結び付け、保存は行わないようにするには、wheels.buildのようにbuildメソッドを使います。引数にはハッシュでモデルの属性を指定できます。

```
@car.wheels.build(name: "車輪1")
```

ハッシュを複数指定することもできます。

```
@car.wheels.build({ name: "車輪1" }, { name: "車輪2" })
```

has_manyによって使えるようになったwheelsメソッドが返すのはリレーションオブジェクトです。したがって、集計用のメソッドやクエリーメソッドを呼び出せます。

```
@car.wheels.count # 車輪の数
@car.wheels.order("created_at DESC") # クエリーメソッド
```

このChapterのサンプルでは、会員情報（Memberモデル）とブログ記事（Entryモデル）の間で1対多の関連付けを作ります。

■命名規約とオプション

has_manyとbelongs_toでモデル間の関連付けを表すときには、名前について次のルールがあります。

- 外部キーのカラム名は、car_idのように「参照先のテーブル名（モデル名）を単数形にしたもの」＋「_id」とする。
- belongs_toに指定する名前は、テーブル名（モデル名）の単数形を使う。
- has_manyに指定する名前は、テーブル名（モデル名）の複数形を使う。

外部キーのカラム名がルールと異なるときは、foreign_keyオプションでカラム名を指定できます。次の例では外部キーにcar_idではなくvehicle_idを使うようにしています。

```
class Car < ApplicationRecord
  has_many :engines, foreign_key: "vehicle_id"
end
```

```
class Engine < ApplicationRecord
  belongs_to :car, foreign_key: "vehicle_id"
```

```
end
```

関連付けで使われるメソッド名を変えたい場合は、class_nameオプションを使います。次の例は、MotorモデルがCarモデルを参照しています。メソッド名をmotorsではなくenginesにしたい場合は、class_nameオプションに本当のモデル名を指定します。

```
class Car < ApplicationRecord
  has_many :engines, class_name: "Motor"
end
```

has_manyでよく使われるオプションにdependentがあります。dependentオプションを:destroyとすると、参照先のレコードを削除したときに参照元のレコードも自動的に削除されます。dependentオプションを:nullifyとすると、参照先のレコードを削除したときに参照元の外部キーがNULLになります。

次の例では、自動車レコードを削除すると、結び付いているエンジンレコードをすべて自動的に削除します。

```
class Car < ApplicationRecord
  has_many :engines, dependent: :destroy
end
```

10.2 会員ブログ関連モデルの準備

モデル間の関連付けで一番よく使われるのが1対多です。ここでは、Morning Gloryのサイトに会員のブログ機能を加え、ネストされたリソースを使って会員ごとのブログを表示できるようにします。

ブログ記事の関連付け

ここではブログ記事用のEntryモデルを作成し、Memberモデルとの間に1対多の関連付けを作ります。

■Entryモデルの作成

MemberモデルとEntryとの間の1対多の関連付けは、「会員はブログ記事をたくさん持つ」（会員はたくさんの記事の筆者になれる）、「ブログ記事は会員に属する」（記事は特定の会員を筆者とする）という結び付きです。

まず、「bin/rails g」コマンドでEntryモデル（entriesテーブル）を作成します。

```
$ bin/rails g model entry
```

次のようにマイグレーションスクリプトを記述します。

LIST chapter10/db/migrate/20180528133133_create_entries.rb

```
1 class CreateEntries < ActiveRecord::Migration[5.2]
2   def change
3     create_table :entries do |t|
4       t.references :member, null: false          # 外部キー
```

```
5     t.string :title, null: false                  # タイトル
6     t.text :body                                  # 本文
7     t.datetime :posted_at, null: false            # 投稿日
8     t.string :status, null: false, default: "draft" # 状態
9
10    t.timestamps null: false
11   end
12  end
13 end
```

4行目の「t.references :member」によってentriesテーブルに整数型のmember_idカラムが追加されます。referencesメソッドは指定されたシンボルに「_id」を加えた名前で整数型のカラムを追加します。「t.integer :member_id」と書いても同じですが、referencesメソッドを使えばこのカラムが外部キーであることを明示できます。

また、referencesメソッドを使った場合、暗黙のうちに外部キーにインデックスが設定されます。インデックスについては、「4.2　テーブルの作成」の「インデックス」を参照してください。もしインデックスを設定したくない場合は、次のようにindex: falseオプションを付ける必要があります。

```
t.references :member, null: false, index: false
```

8行目ではstatusカラムにデフォルト値として"draft"を設定しています。このカラムには、"draft"（下書き）、"member_only"（会員限定）、"public"（公開）のどれかの文字列を入れることにします。

マイグレーションを実行します。

```
$ bin/rails db:migrate
```

マイグレーションスクリプトに誤りがあって途中で止まった場合は、間違いを修正してからデータベースを再構築してください。

```
$ bin/rails db:rebuild
```

外部キー制約

データベースでテーブル同士を関連付けるときには、外部キーを作成すると同時に**外部キー制約**を付けることがあります。外部キー制約を付けると、テーブル間の関連付けに整合性が取れないときに、データベースが自動的にエラーを出すようになります。たとえば、ある外部キーに参照先のない値がセットされた場合などです。

MySQLやPostgreSQLで外部キー制約を設定するには、次のようにマイグレーションスクリプトでforeign_key: trueオプションを使います。これで、entriesテーブルのmember_idカラムにmembersテーブルのidカラムへの外部キー制約が設定されます。

```
class CreateEntries < ActiveRecord::Migration
  def change
    create_table :entries do |t|
      t.references :member, null: false, foreign_key: true
      （省略）
```

あるいは、次のようにadd_foreign_keyメソッドで外部キーを設定することもできます。

```
class CreateEntries < ActiveRecord::Migration
  def change
    create_table :entries do |t|
      t.references :member, null: false
      （省略）
    end

    add_foreign_key :entries, :members
```

```
  end
end
```

このメソッドの引数には、参照元のテーブル名と参照先のテーブル名をシンボルで指定します。参照元のカラム名は、参照先のテーブル名を単数形にして_idを付けたものになります（この例ではmember_id）。このルールに合わない場合は、columnオプションでカラム名を指定します。

```
add_foreign_key :entries, :members, column: "author_id"
```

■モデル間の関連付け

Memberモデルにhas_manyメソッドを記述して、「会員はブログ記事をたくさん持つ」という結び付きを作ります。

LIST chapter10/app/models/member.rb

```
1 class Member < ApplicationRecord
2   has_secure_password
3
4   has_many :entries, dependent: :destroy
5
6   validates :number, presence: true,
  （以下省略）
```

ある会員が削除されると、その会員が書いたブログ記事はすべて削除されるようにするため、dependentオプションにシンボル:destroyを指定しています。

次に、Entryモデルにbelongs_toメソッドを記述して、「ブログ記事は会員に属する」という結び付きを作ります。ただし、記事を書いた会員を参照するメソッド名はauthorにします。

LIST chapter10/app/models/entry.rb

```
1 class Entry < ApplicationRecord
2   belongs_to :author, class_name: "Member", foreign_key:
"member_id"
3 end
```

class_nameオプションとforeign_keyオプションの指定は必須です。この2つがないと、Railsはentriesテーブルにauthor_idというカラムがあり、Authorという名前のモデルを参照しているのだと誤解してしまいます。

以上の変更により、MemberモデルとEntryモデルが1対多で関連付けられました。変数@memberにMemberオブジェクトがセットされているとすると、テンプレートでその会員の書いたブログ記事のタイトルを並べるには次のようにします。並べ方は投稿日で昇順となります。

```
<ul>
  <% @member.entries.order(:posted_at).each do |e| %>
    <li><%= e.title %></li>
  <% end %>
</ul>
```

また、変数@entriesにEntryオブジェクトの配列がセットされているとすれば、テンプレートでブログ記事の著者名を並べるには次のようにします。

```
<ul>
  <% @entries.each do |e| %>
    <li><%= e.author.name %></li>
  <% end %>
</ul>
```

Entryモデルでの準備

次節で使うコントローラやビューで使う機能を先回りしてEntryモデルに用意しておきましょう。

■モデルの属性名の日本語化

まず、モデルの属性名をロケールテキストに追加しておきます。

LIST chapter10/config/locales/ja.yml

```
1 ja:
2 ␣␣activerecord:
3 ␣␣␣␣models:
4 ␣␣␣␣␣␣member: 会員情報
5 ␣␣␣␣␣␣article: ニュース記事
6 ␣␣␣␣␣␣entry: ブログ記事
7 ␣␣␣␣attributes:
  （省略）
27 ␣␣␣␣␣␣␣␣member_only: 会員限定
28 ␣␣␣␣␣␣entry:
29 ␣␣␣␣␣␣␣␣title: タイトル
30 ␣␣␣␣␣␣␣␣body: 本文
31 ␣␣␣␣␣␣␣␣posted_at: 日時
32 ␣␣␣␣␣␣␣␣status: 状態
33 ␣␣␣␣␣␣␣␣status_draft: 下書き
34 ␣␣␣␣␣␣␣␣status_member_only: 会員限定
35 ␣␣␣␣␣␣␣␣status_public: 公開
36 ␣␣␣␣errors:
  （以下省略）
```

■バリデーション

Entryモデルにバリデーションを設定します。「記事タイトルは空を禁止、200文字以内」、「本文と投稿日は空を禁止」、「状態の文字列は"draft"、"member_only"、"public"のいずれか」という設定です。

LIST chapter10/app/models/entry.rb

```
1 class Entry < ApplicationRecord
2   belongs_to :author, class_name: "Member", foreign_key: "member_id"
3
4   STATUS_VALUES = %w(draft member_only public)
5
6   validates :title, presence: true, length: { maximum: 200 }
7   validates :body, :posted_at, presence: true
8   validates :status, inclusion: { in: STATUS_VALUES }
    （以下省略）
```

4行目ではstatusカラムにセットできる値の配列を定数STATUS_VALUESとして定義し、8行目でバリデーションの設定に利用しています。

■ブログ記事を絞り込むスコープ

Entryモデルにスコープを設定し、見る人に応じて閲覧できるブログ記事を絞り込めるようにします。

LIST chapter10/app/models/entry.rb

```
1 class Entry < ApplicationRecord
    （省略）
8   validates :status, inclusion: { in: STATUS_VALUES }
9
10  scope :common, -> { where(status: "public") }
```

```
11   scope :published, -> { where("status <> ?", "draft") }
12   scope :full, ->(member) {
13     where("status <> ? OR member_id = ?", "draft", member.id) }
14   scope :readable_for, ->(member) { member ? full(member) :
common }
15 end
```

commonスコープは公開記事を選び出します。publishedスコープは下書き状態ではない記事、つまり公開記事と会員限定記事を選び出します。fullスコープでは引数にMemberオブジェクトを渡し、その会員が書いたか下書き状態ではない記事を選び出します。

readable_forスコープでは、commonスコープとfullスコープを組み合わせます。これにより、ログイン前のユーザーには公開記事だけを見せ、ログイン後には公開記事と会員限定記事、および自分の下書き記事を一覧表示できるようになります。

■ビュー用のメソッド

ビューで使う2つのクラスメソッドstatus_text、status_optionsを作っておきます。

LIST chapter10/app/models/entry.rb

```
1 class Entry < ApplicationRecord
2   belongs_to :author, class_name: "Member", foreign_key: "member_id"
3
4   STATUS_VALUES = %w(draft member_only public)
  （省略）
16   class << self
17     def status_text(status)
18       I18n.t("activerecord.attributes.entry.status_#{status}")
19     end
```

```
20
21   def status_options
22     STATUS_VALUES.map { |status| [status_text(status), status] }
23   end
24  end
25 end
```

status_textメソッドは、tメソッドを使ってstatusカラムの値を日本語にするものです（「7.2 メッセージの日本語化」の「国際化機能の使い方」を参照）。status_optionsメソッドは、定数STATUS_VALUESを使って、[["下書き", "draft"], ["会員限定", "member_only"], ["公開", "public"]]のような配列を作ります。このメソッドは、編集フォームで記事の状態を選択するリストを作るのに使います。

■シードデータ

開発用のシードデータを用意しましょう。db/seeds.rbの配列table_namesにentriesを追加します。

LIST chapter10/db/seeds.rb

```
1 table_names = %w(members articles entries)
2 table_names.each do |table_name|
  （以下省略）
```

次のようにEntryモデル用のシードデータを記述します。

LIST chapter10/db/seeds/development/entries.rb

```
1 body =
2   "今晩は久しぶりに神宮で野球観戦。内野B席の上段に着席。¥n¥n" +
3   "先発はヤクルトがブキャナン、広島はジョンソン。" +
4   "2回裏に中村選手のセーフティスクイズなどでヤクルトが3点を先取。" +
```

```
  "そして、8回裏には代打・荒木選手がレフトスタンドへ2号満塁ホームラン。¥n¥n" +
  "ブキャナン投手の今季初完封を見届けて、気分良く家路に着きました。"

%w(Taro Jiro Hana).each do |name|
  member = Member.find_by(name: name)
  0.upto(9) do |idx|
    Entry.create(
      author: member,
      title: "野球観戦#{idx}",
      body: body,
      posted_at: 10.days.ago.advance(days: idx),
      status: %w(draft member_only public)[idx % 3]
    )
  end
end
```

「Taro」、「Jiro」、「Hana」の3人に対してそれぞれ記事を10件作ります。記事の状態は3件ごとに、下書き、会員限定、公開の繰り返しにします。

データベースの再構築を行ってください。

```
$ bin/rails db:rebuild
```

10.3 会員ブログ機能の実装

ネストされたリソース

Morning Gloryのブログでは、全会員の記事一覧ページのほかに会員ごとの記事一覧ページも表示できるようにします。そのために、ネストされたリソースというテクニックを使います。

■ネストされたリソースのルーティング

「Taroさんのブログ記事」や「Hanakoさんのブログ記事」のようなページを作成するには、**ネストされたリソース**を設定します。MemberとEntryのように1対多で関連付けられているモデルをコントローラで扱うときは、特に便利です。

ネストされたリソースは、routes.rbで次のように設定します。リソースentriesは、リソースmembersの下に入れ子（ネスト）になっています。

```
resources :members do
  resources :entries
end
```

このルーティングによって、EntriesControllerのアクションには次のパスでアクセスできるようになります。

ネストされたリソースのルーティング

アクション	パス	HTTPメソッド
index	/members/123/entries	GET
show	/members/123/entries/456	GET

new	/members/123/entries/new	GET
edit	/members/123/entries/456/edit	GET
create	/members/123/entries	POST
update	/members/123/entries/456	PATCH
destroy	/members/123/entries/456	DELETE

idが123の会員の記事一覧ページなら、「/members/123/entries」のパスでアクセスできます。会員のidが123で、その会員の記事のidが456なら「/members/123/entries/456」で記事詳細ページにアクセスできます。

このとき、123はparams[:member_id]のように「リソース名の単数形＋_id」という名前を持つパラメータで取得できます。456はparams[:id]で取得できます。EntriesControllerのindexアクションとshowアクションは、たとえば次のようになるでしょう。

```
def index
  @member = Member.find(params[:member_id])
  @entries = @member.entries.order(posted_at: :desc)
end

def show
  @member = Member.find(params[:member_id])
  @entry = @member.entries.find(params[:id])
end
```

ネストされたリソースへのリンクを作成するときには、次のようにパスを返すメソッドを使えるようになります。Memberオブジェクトが@memberに、Entryオブジェクトが@entryに入っているものとします。

ネストされたリソースのパスを返すメソッド

アクション	パスを返すメソッド
index	member_entries_path(@member)

show	member_entry_path(@member, @entry)
new	new_member_entry_path(@member)
edit	edit_member_entry_path(@member, @entry)
create	member_entries_path(@member)
update	member_entry_path(@member, @entry)
destroy	member_entry_path(@member, @entry)

また、link_to、redirect_to、form_forの各メソッドには、パスを返すメソッドの代わりにパスを表す配列を指定することもできます。

ネストされたリソースのパスを表す配列

アクション	パスを表す配列
index	[@member, :entries]
show	[@member, @entry]
new	[:new, @member, :entry]
edit	[:edit, @member, @entry]
create	[@member, :entries]
update	[@member, @entry]
destroy	[@member, @entry]

たとえば、会員@memberの記事一覧へのリンクは次のように作れます。

```
<%= link_to "記事一覧", [@member, :entries] %>
```

本書のサンプルでは使いませんが、会員@memberの記事@entryを編集するリンクは次のように作れます。

```
<%= link_to "記事の編集", [:edit, @member, @entry] %>
```

■ルーティングの設定

実際にルーティングを設定しましょう。asagaoアプリケーションでは、会員ごとの記事一覧ページだけネストされたリソースを使うことにします。routes.rbを次のように書き換えます。

LIST chapter10/config/routes.rb

```
（省略）
 9  resources :members do
10    get "search", on: :collection
11    resources :entries, only: [:index]
12  end
13
14  resource :session, only: [:create, :destroy]
15  resource :account, only: [:show, :edit, :update]
16  resource :password, only: [:show, :edit, :update]
17
18  resources :articles
19  resources :entries
20 end
```

11行目でネストされたリソースを設定しています。indexアクションだけ使うので、onlyオプションで限定します。19行目では、「resources :entries」でブログ記事に対して通常のルーティングを設定します。

これにより、EntriesControllerのindexアクションには「/members/123/entries」と「/entries」という2つのパスにGETメソッドでアクセスできることになります。この2つのパスは、「会員ごとの記事一覧ページ」と「全会員の記事一覧ページ」のパスです。

ブログ記事の一覧と表示

ブログ記事の表示のための機能を作りましょう。

■EntriesControllerの作成

「bin/rails g」コマンドでEntriesControllerを作成します。4つのアクションindex、show、new、editのためのテンプレートも自動生成します。

```
$ bin/rails g controller entries index show new edit
```

EntriesControllerのindexアクションとshowアクションを次のように書き換えます。indexアクションではネストされたリソース（会員ごとの記事一覧）と通常のリソース（全会員の記事一覧）を扱います。

LIST chapter10/app/controllers/entries_controller.rb

```
1 class EntriesController < ApplicationController
2   before_action :login_required, except: [:index, :show]
3
4   # 記事一覧
5   def index
6     if params[:member_id]
7       @member = Member.find(params[:member_id])
8       @entries = @member.entries
9     else
10      @entries = Entry.all
11    end
12
13    @entries = @entries.readable_for(current_member)
14      .order(posted_at: :desc).page(params[:page]).per(3)
15  end
16
17  # 記事詳細
18  def show
```

```
19     @entry = Entry.readable_for(current_member).find(params[:id])
20   end
     （以下省略）
```

indexアクションでは、まずネストされたリソースを扱うかどうかをparams[:member_id]の有無で調べます。もしあれば、会員を@memberに取り出して、その会員の記事（1対多の結び付きにある記事）を@entriesに取り出します。params[:member_id]がなければ、すべての記事を取り出します。

さらに、@entriesに対してreadable_forスコープを呼び出して、閲覧できるブログ記事を絞り込んでいます。また、「9.4　ページネーション」で紹介したkaminariのページネーション機能を使っています。

showアクションでは、readable_forスコープとfindメソッドを組み合わせます。params[:id]に閲覧できない記事が指定されたときは、例外ActiveRecord::RecordNotFoundが発生します。

■記事の一覧表示

indexアクションのテンプレートの内容をすべて削除して、次の内容を書き込みます。

LIST chapter10/app/views/entries/index.html.erb

```
1 <% @page_title = @member ? @member.name + "さんのブログ" : "会員のブログ" %>
2 <h1><%= @page_title %></h1>
3
4 <% if current_member %>
5   <div class="toolbar"><%= link_to "ブログ記事の作成", :new_entry %></div>
6 <% end %>
7
```

```
 8 <% if @entries.present? %>
 9   <% @entries.each do |entry| %>
10     <h2><%= entry.title %></h2>
11     <p><%= truncate(entry.body, length: 80) %>
12       <%= link_to "もっと読む", entry %></p>
13     <%= render "footer", entry: entry %>
14   <% end %>
15   <%= paginate @entries %>
16 <% else %>
17   <p>記事がありません。</p>
18 <% end %>
```

1行目では、変数@memberの有無で特定の会員のブログか全会員のブログかを判別し、見出しを切り替えます。

4～6行目では、ユーザーがログイン中であれば、ブログ記事を作成するnewアクションへのリンクを表示します。

9～14行目のループで記事の一部を表示し、「もっと読む」にshowアクションへのリンクを張ります。また、部分テンプレート_footer.html.erbを埋め込んで、記事のフッターとします。

app/views/entriesディレクトリの下に、フッター用の部分テンプレート_footer.html.erbを次の内容で作成します。

LIST chapter10/app/views/entries/_footer.html.erb

```
 1 <ul class="entry-footer">
 2   <% if current_member %>
 3     <li><%= Entry.status_text(entry.status) %></li>
 4     <% if current_member == entry.author %>
 5       <%= menu_link_to "編集", [:edit, entry] %>
 6       <%= menu_link_to "削除", entry, method: :delete,
 7             data: { confirm: "本当に削除しますか？" } %>
```

```
 8     <% end %>
 9   <% end %>
10   <li>
11     by <%= link_to entry.author.name, [entry.author, :entries] %>
12   </li>
13   <li>
14     <%= entry.posted_at.strftime("%Y/%m/%d %H:%M") %>
15   </li>
16 </ul>
```

3行目では、Entryモデルのクラスメソッドstatus_textで記事の状態を表示します。4〜8行目では、ログインしている会員がその記事の著者である場合に「編集」と「削除」のリンクを表示します。11行目では、著者のユーザー名を会員ごとの記事一覧ページへのリンクにしています。

ここで、ヘルパーメソッドmenu_link_toのコードを次のように修正します。

LIST chapter10/app/helpers/application_helper.rb

```
（省略）
 8  def menu_link_to(text, path, options = {})
 9    content_tag :li do
10      condition = options[:method] || !current_page?(path)
11
12      link_to_if(condition, text, path, options) do
13        content_tag(:span, text)
14      end
15    end
16  end
17 end
```

「削除」リンクのリンク先URLが現在のページのURLと同じであるため、元の実装ではリンクになりません。methodオプションが設定されている場合は、常にリンク化するようにコードを変更しました。10行目のヘルパーメソッドcurrent_page?は引数に指定されたURLパスと現在のページのURLパスが一致するかどうかを調べてtrueまたはfalseを返します。

続いて、ブログ記事一覧にアクセスできるように、サイトのヘッダー用の部分テンプレートを修正して、メニュー項目「ブログ」をEntriesControllerのindexアクションへのリンクに変えましょう。

LIST chapter10/app/views/shared/_header.html.erb

```
（省略）
15   <%= menu_link_to "ブログ", :entries %>
（以下省略）
```

さらに、app/assets/stylesheetsディレクトリにブログ記事用のスタイルを記述したentries.cssを追加します。ソースコードの掲載は省略しますので、サンプルソースのchapter10/app/assets/stylesheetsディレクトリからコピーしてください。

ブラウザを開き、ログインしない状態でメニューの［ブログ］をクリックしてブログ記事一覧を表示してみましょう。

RESULT 全会員の記事一覧

記事のフッターの「Hana」をクリックすれば、Hanaさんの記事一覧が表示できます。

RESULT Hanaさんの記事一覧

ユーザー名「Hana」、パスワード「asagao!」でログインしてから記事一覧を表示すると、下書き記事も表示できます。

RESULT Hanaさんの記事一覧（ログイン後）

■記事の詳細表示

showアクションのテンプレートを作成します。ログインしている会員が記事の筆者と同じなら、編集リンクを表示します。記事のフッターには、一覧表示で作った部分テンプレート_footer.html.erbを使います。

LIST chapter10/app/views/entries/show.html.erb

```
1 <% @page_title = @entry.title + " - " + @entry.author.name + "さんのブログ" %>
2 <h1><%= @entry.author.name %>さんのブログ</h1>
3 <h2><%= @entry.title %></h2>
```

```
4
5 <%= simple_format(@entry.body) %>
6
7 <%= render "footer", entry: @entry %>
```

記事一覧のページから［もっと読む］をクリックすると、個別の記事が表示されます。

RESULT ブログ記事の表示

■サイドバーの修正

サイドバーにも最新のブログ記事の一覧を表示することにしましょう。サイドバーの部分テンプレートを次のように書き換えてください。

LIST chapter10/app/views/shared/_sidebar.html.erb

```
（省略）
14 <h2>会員のブログ</h2>
15 <%
16   entries = Entry.readable_for(current_member)
```

```
17     .order(posted_at: :desc).limit(5)
18 %>
19 <ul>
20   <% entries.each do |entry| %>
21     <li>
22       <%= link_to entry.title, entry %>
23       by <%= link_to entry.author.name, [entry.author, :entries] %>
24     </li>
25   <% end %>
26 </ul>
```

サイドバーの表示が次のように変わります。

RESULT サイドバーのブログ記事一覧

記事の作成、更新、削除

記事を編集したり削除したりする機能を加えてブログ記事の機能を完成させましょう。

newアクションとeditアクション

EntriesControllerで編集フォームを表示させるためのアクション、newとeditを次のように修正します。

LIST chapter10/app/controllers/entries_controller.rb

```
1 class EntriesController < ApplicationController
  （省略）
22   # 新規登録フォーム
23   def new
24     @entry = Entry.new(posted_at: Time.current)
25   end
26
27   # 編集フォーム
28   def edit
29     @entry = current_member.entries.find(params[:id])
30   end
  （以下省略）
```

newアクションでは、「posted_at: Time.current」として、デフォルトの投稿日時を現在の日時にしています。

editアクションでは、1対多の関連付けの機能を使ってcurrent_member.entriesから記事を取り出しています。ほかの会員のブログ記事を取り出そうとするとエラーになります。

■記事の編集フォーム

newアクションとeditアクションのテンプレートは型どおりに作ります。

LIST chapter10/app/views/entries/new.html.erb

```
1 <% @page_title = "ブログ記事の新規作成" %>
2 <h1><%= @page_title %></h1>
3
4 <%= form_for @entry do |form| %>
5   <%= render "form", form: form %>
```

```
  <div><%= form.submit %></div>
<% end %>
```

LIST chapter10/app/views/entries/edit.html.erb

```
<% @page_title = "ブログ記事の編集" %>
<h1><%= @page_title %></h1>

<div class="toolbar"><%= link_to "記事の表示に戻る", @entry %>
</div>

<%= form_for @entry do |form| %>
  <%= render "form", form: form %>
  <div><%= form.submit %></div>
<% end %>
```

部分テンプレート_form.html.erbを次のように作成します。

LIST chapter10/app/views/entries/_form.html.erb

```
<%= render "shared/errors", obj: @entry %>

<table class="attr">
  <tr>
    <th width="80"><%= form.label :title %></th>
    <td><%= form.text_field :title, size: 50 %></td>
  </tr>
  <tr>
    <th><%= form.label :body %></th>
    <td><%= form.text_area :body, rows: 10, cols: 45 %></td>
  </tr>
```

```
12  <tr>
13    <th><%= form.label :posted_at, for: "entry_posted_at_1i" %></th>
14    <td><%= form.datetime_select :posted_at,
15            start_year: 2000, end_year: Time.current.year + 1,
16            use_month_numbers: true %></td>
17  </tr>
18  <tr>
19    <th><%= form.label :status %></th>
20    <td><%= form.select :status, Entry.status_options %></td>
21  </tr>
22 </table>
```

記事の状態（statusカラム）を選択するために、selectメソッドで選択リスト（selectタグとoptionタグの組み合わせ）を作成します。selectメソッドの第2引数には、Entryモデルのクラスメソッドstatus_optionsが返す配列を渡します。

■記事の追加と更新

EntriesControllerにcreateアクションとupdateアクションを追加します。

LIST chapter10/app/controllers/entries_controller.rb

```
（省略）
32  # 新規作成
33  def create
34    @entry = Entry.new(params[:entry])
35    @entry.author = current_member
36    if @entry.save
37      redirect_to @entry, notice: "記事を作成しました。"
38    else
```

```
39      render "new"
40    end
41  end
42
43  # 更新
44  def update
45    @entry = current_member.entries.find(params[:id])
46    @entry.assign_attributes(params[:entry])
47    if @entry.save
48      redirect_to @entry, notice: "記事を更新しました。"
49    else
50      render "edit"
51    end
52  end
    （以下省略）
```

createアクションでは、記事を保存する前に「@entry.author = current_member」で現在ログインしている会員を記事の筆者にします。

ログインして記事の追加機能と編集機能を試してみましょう。

RESULT ブログ記事の編集フォーム

■記事の削除

最後に、EntriesControllerにdestroyアクションを追加して、ブログ管理機能の完成です。

LIST chapter10/app/controllers/entries_controller.rb

```
（省略）
54  # 削除
55  def destroy
56    @entry = current_member.entries.find(params[:id])
57    @entry.destroy
58    redirect_to :entries, notice: "記事を削除しました。"
59  end
60 end
```

自分の書いた記事を削除できるかどうか試してください。

1対1の関連付け

モデル間の関連付けには、このChapterで学んだ「1対多の関連付け」のほかに、「1対1の関連付け」と「多対多の関連付け」があります。後者はChapter 14で学ぶことになりますが、前者については本書では説明を省略します。

2つのモデルXとYが1対1で関連付けられているのであれば、原理的にはモデルは1つで十分です。Yのすべての属性をXに移動させてしまっても、すべての情報を取り扱えます。属性名が衝突したら適宜変更するだけです。

しかし、現実のRails開発ではしばしば「1対1の関連付け」が使われます。たとえば、顧客を表すCustomerモデルと顧客の住所を表すCustomerAddressモデルを考えてください。後者には郵便番号、都道府県、市区町村などが別々の属性として記録されます。この2つのモデルは1対1で関連付けることができます。

CustomerAddressモデルの各属性をCustomerモデルに移動させることも可能ですが、2つのモデルに分離しておいたほうが扱いやすくなります。たとえば、住所のわからない顧客を記録したいとき、データベーステーブルに多くのNULL値を記録しなくて済みます。バリデーションやコールバックもそれぞれのモデルで行ったほうがすっきり書けます。

読者の皆さんが本格的なアプリケーション開発を行う際には、ぜひインターネット上の情報などで「1対1の関連付け」について調べて活用してください。

Chapter 10のまとめ

- **複数のモデルを関連付けるには、外部キーを利用してテーブルのレコードが別のテーブルのレコードを参照するようにします。**
- **1対多の関連付けを作るには、参照元のモデルクラスにbelongs_to、参照先のモデルクラスにhas_manyを記述して、モデル名を指定します。**
- **ネストしたフォームを使うと、1対多で関連付けられている複数のモデルのデータを一度に扱えます。**

練習問題

［A］　書籍をまとめて管理できるように、データベースにshelves（単数形はshelf）テーブルを追加したとします。booksテーブルがshelvesテーブルのレコードを参照できるように、外部キーを加えることとします。マイグレーションスクリプトの空欄を埋めて、外部キーとなるカラム名を指定してください。

```
class CreateBooks < ActiveRecord::Migration[5.2]
  def change
    create_table :books do |t|
      
      t.string :title, null: false
      t.string :author, null: false
      t.integer :price, null: false

      t.timestamps
    end
  end
end
```

［B］　Shelfモデル（shelvesテーブル）とBookモデル（booksテーブル）を1対多で関連付けます。1つの本棚が複数の書籍を持っているという結び付きです。モデルクラスにそれぞれ必要な記述を加えてください。

　なお、本棚が削除されても書籍は削除されないこととします。

app/models/shelf.rb

```
class Shelf < ApplicationRecord
  
```

```
end
```

app/models/book.rb

```
class Book < ApplicationRecord
  
end
```

［C］　問題［A］、［B］のようにモデルを関連付けたあと、次のようにルーティングを設定します。

config/routes.rb

```
Rails.application.routes.draw do
  resources :shelves
    resources :books
  end
  resources :books
end
```

BooksControllerのindexアクションを作ります。パラメータにShelfモデルのidがある場合は、その本棚に属する書籍だけを一覧表示するようにしてください。書名（title）順に取り出すこととし、ページネーションの機能は使わないこととします。

app/controllers/books_controller.rb

```
class BooksController < ApplicationController
  def index
    if params[:shelf_id]
      @shelf = 
      @books = @shelf.
```

```
    else
      @books = Book.order("title")
    end
  end
end
```

Part 4

発展的な内容

このPartでは、ストロング・パラメータ、例外処理、アセット・パイプライン、Active Storage、多対多の関連付け、名前空間などを学んでいきます。初心者には少し難しい内容も含まれていますが、繰り返し読んで乗り越えましょう。

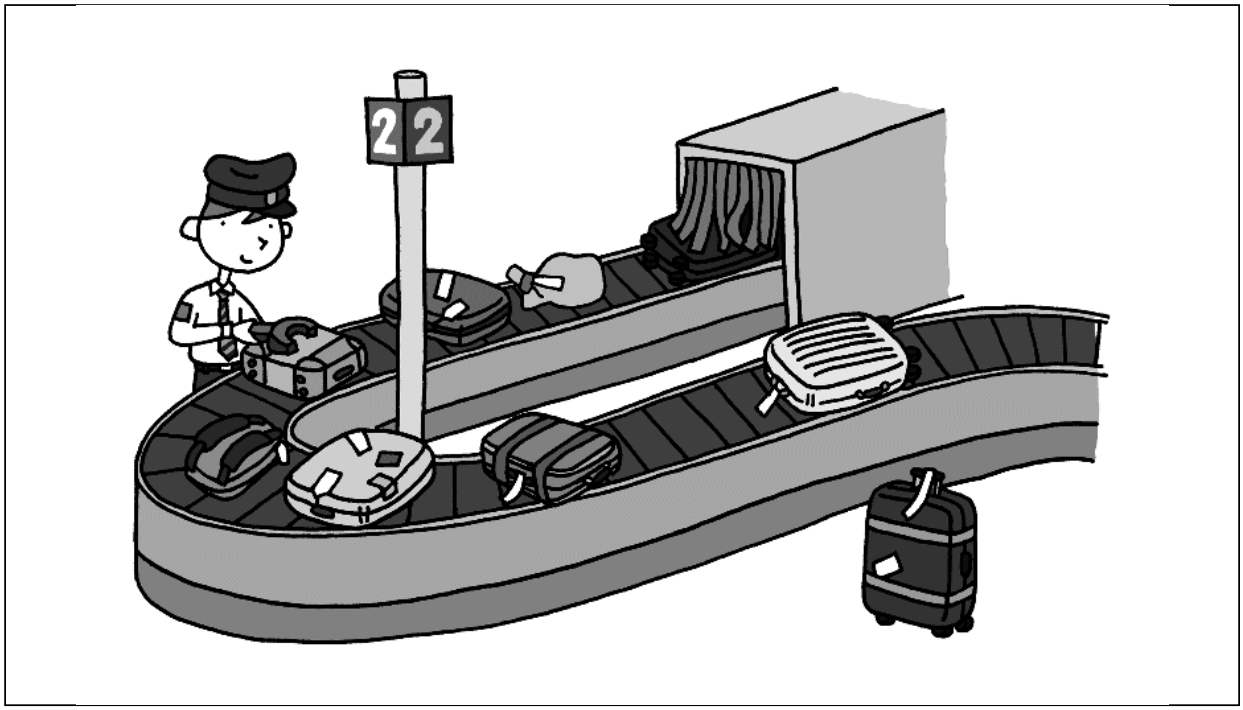
Chapter

11　セキュリティと例外処理

　ウェブアプリケーションを本番環境（実運用環境）で公開するときには、悪意のあるユーザーによる攻撃が行われる可能性を常に考慮する必要があります。また、ソフトウェアの不具合（バグ）やユーザーによる想定外の操作で例外（エラー）が発生する可能性にも備えておくべきです。

これから学ぶこと

- フォームから送信された情報を保存する際に、モデルの属性を保護する方法を学びます。
- 例外が発生したときのエラーページを自作する方法を解説します。



Railsアプリケーションの開発ではさまざまな手法による外部からの攻撃に備える必要があります。どのように防御すればよいのでしょうか？ また、例外（エラー）が発生したときにはブラウザにどのように伝えればよいでしょうか？

11.1 ストロング・パラメータ

ストロング・パラメータは、モデルの属性を攻撃から保護するしくみです。この節では、Chapter 5からChapter 10で作成した会員管理、マイアカウントページ、パスワード変更の機能、ニュース記事管理機能をストロング・パラメータに準拠した形に書き換えます。

ストロング・パラメータとは

■マスアサインメント脆弱性

「8.4　マイアカウントページの作成」では、自分のアカウント情報を更新するAccountsControllerのupdateアクションを次のように実装しました。

LIST chapter11/app/controllers/accounts_controller.rb

```
（省略）
12  def update
13    @member = current_member
14    @member.assign_attributes(params[:account])
15    if @member.save
16      redirect_to @member, notice: "アカウント情報を更新しました。"
17    else
18      render "edit"
19    end
20  end
（以下省略）
```

14行目のparams[:account]には、たとえば次のような内容のハッシュがセットされます。

```
{
  number: 11, name: "TK", full_name: "黒田 努",
  "birthday(1i)" => "1968",  "birthday(2i)" => "6", "birthday(3i)" => "12",
  email: "tsutomu@example.com", sex: "1"
}
```

自分自身で管理者フラグ（administrator属性）を変更できないように、マイアカウントの編集フォームには「管理者」の項目を省略してあります。しかし、ユーザーは次のような方法で私たち開発者の裏をかくことができます。

最近のブラウザには表示しているページのHTMLソースを書き換える機能が備わっています。そこで、「マイアカウント情報の編集」ページを開いて、HTMLソースの中から「背番号」フィールドのinputタグを探します。そして、そのname属性の値をaccount[number]からaccount[administrator]に書き換えます。最後に、「背番号」フィールドに「1」と書き込んで「更新する」ボタンを押せば、管理者でないメンバーが管理者になることができてしまいます。この問題は**マスアサインメント脆弱性**と呼ばれ、重大なセキュリティホールになります。

■ストロング・パラメータの有効化

ストロング・パラメータはこのマスアサインメント脆弱性への対策としてRails 4.0で導入されたしくみです。「6.2　レコードの作成、更新、削除」でストロング・パラメータを無効化しているので、ここで再び有効化しましょう。config/application.rbの36行目のtrueをfalseに書き換えます。あるいは、36行目の記述を削除してもかまいません。

LIST chapter11/config/application.rb

```
（省略）
33    config.time_zone = "Tokyo"
34    config.i18n.default_locale = :ja
35
```

```
36     config.action_controller.permit_all_parameters = false
37   end
38 end
```

Railsサーバーが起動中なら、いったん止めて起動し直します。ブラウザで「マイアカウント情報の編集」ページを表示して、「更新する」ボタンをクリックしてください。ActiveModel::ForbiddenAttributesErrorというエラーが表示されれば、ストロング・パラメータが有効化されていることがわかります。

■ストロング・パラメータの使い方

ストロング・パラメータの使い方について解説しましょう。AccountsControllerのupdateアクションを次のように書き換えたとします。params[:member]の後ろに.permit(:number, :name)を加えています。

```
def update
  @member = current_member
  @member.assign_attributes(params[:account].permit(:number, :name))
  if @member.save
    redirect_to :account, notice: "アカウント情報を更新しました。"
  else
    render "edit"
  end
end
```

permitメソッドがない場合は、params[:member]は次のようなハッシュを返します。実際には、これはHashクラスのオブジェクトではなく、Hashの機能を拡張したActionController::Parametersクラスのオブジェクトです。

```
{ "number" => "10", "name" => "Taro",
  "full_name" => "佐藤 太郎", "sex" => "1",
  "birthday(1i)" => "1981", "birthday(2i)" => "12",
  "birthday(3i)" => "1",
  "email" => "Taro@example.com" }
```

permitメソッドにモデルの属性名を配列で指定すると、params[:member]が返すハッシュは次のように変わります。

```
{ "number" => "10", "name" => "Taro" }
```

このように、permitメソッドを使うと、モデルオブジェクトに渡す属性に制限を加えて、余計な属性を保存させないようにできます。

例外ActiveModel::ForbiddenAttributesError

permitメソッドが返すActionController::Parametersオブジェクトには「permitメソッドを通った」という印が付いています。ストロング・パラメータが有効であるとき、この印が付いていないActionController::Parametersオブジェクトをモデルクラスのassign_attributesメソッドなどに渡すと、例外ActiveModel::ForbiddenAttributesErrorが発生します。

permitメソッドを使ってストロング・パラメータに対応したコントローラでは、この例外は発生しません。許可されていない属性がパラメータに含まれていたら、その属性は単純に無視されます。

■プライベートメソッドを活用する

「マイアカウント情報の編集」ページのフォームには、全部で6個の入力欄があります。これらに対応する属性の名前をpermitメソッドの引数に加えると、updateアクションの中身は次のようになります。誕生日入力欄は3個のセレクトボックスから構成されていますが、permitメソッドには:birthdayだけを指定すれば十分です。

```
    @member = current_member
    @member.assign_attributes(
      params[:account].permit(
        :number,
        :name,
        :full_name,
        :sex,
        :birthday,
        :email
      )
    )
  (以下省略)
```

さて、機能的にはこれで問題はありませんが、入力欄の数が多いのでアクションの中身がごちゃごちゃしています。そこで、assign_attributesメソッドの引数の中身をプライベートメソッドとして抜き出します。具体的には、次のようにaccount_paramsメソッドを定義します。

```
  private def account_params
    params[:account].permit(
      :number,
      :name,
      :full_name,
      :sex,
      :birthday,
      :email
    )
  end
```

すると、updateアクションを次のように簡潔に記述できるようになります。

```
    @member = current_member
    @member.assign_attributes(account_params)
(以下省略)
```

■requireメソッド

実は、先ほど作成したプライベートメソッドaccount_paramsにひとつ欠陥があります。フォームから送信されてきたパラメータに:accountキーが含まれていなかった場合に例外NoMethodErrorが発生してしまうという点です。:memberがないとparams[:account]がnilを返し、nilにはpermitメソッドがないからです。

もちろん通常の利用でこんなことは起きませんが、ユーザーが「マイアカウント情報の編集」ページのHTMLソースからすべての入力欄を削除してフォームを送信すれば発生します。

このような懸念に対応するにはrequireメソッドを利用します。

```
  private def account_params
    params.require(:account).permit(
      :number,
      :name,
      :full_name,
      :sex,
      :birthday,
      :email
    )
  end
```

params.require(:account)はparams[:account]と同じ働きをしますが、:accountがないときは例外ActionController::ParameterMissingを発生させます。

requireメソッドを利用する意味

account_paramsメソッドでrequireメソッドを使用することで、発生する例外の種類が変わりました。そのことに何か意味があるのでしょうか。

通常、例外NoMethodErrorはプログラムの中に不具合（バグ）が存在することを意味します。アプリケーションログの中にこの例外が発生した記録があれば、調査して修正する必要があります。

しかし、例外ActionController::ParameterMissingの発生は、ストロング・パラメータのしくみが働いて、いたずらや攻撃をはねのけたことを意味します。requireメソッドの利用によって何が起きているのかが明確になるのです。

次節では、本番環境で例外が発生したときにユーザーに見せるエラーページを作りますが、その際、発生した例外の種類によってエラーメッセージを切り替えます。

コントローラの修正

■マイアカウントページのストロング・パラメータ対応

それでは、マイアカウントページをストロング・パラメータに対応させましょう。まず、プライベートメソッドaccount_paramsをAccountsControllerに加えます。

LIST chapter11/app/controllers/accounts_controller.rb

```
（省略）
22  # ストロング・パラメータ
23  private def account_params
24    params.require(:account).permit(
25      :number,
26      :name,
27      :full_name,
28      :sex,
```

```
29       :birthday,
30       :email
31     )
32   end
33 end
```

udpateアクションを変更して、このaccount_paramsメソッドをモデルに渡すようにしましょう。

LIST chapter11/app/controllers/accounts_controller.rb

```
    （省略）
12   def update
13     @member = current_member
14     @member.assign_attributes(account_params)
    （以下省略）
```

実際にブラウザで自分のアカウント情報の編集を行って、正常に動作することを確認してください。

■会員情報のストロング・パラメータ対応

次に、会員管理機能をストロング・パラメータに対応させます。まず、プライベートメソッドmember_paramsをMembersControllerに加えます。会員を新規登録するときだけパスワードの入力が許される点に留意すると、次のような実装になります。

LIST chapter11/app/controllers/members_controller.rb

```
    （省略）
61   # ストロング・パラメータ
62   private def member_params
63     attrs = [
```

```
64      :number,
65      :name,
66      :full_name,
67      :sex,
68      :birthday,
69      :email,
70      :administrator
71    ]
72
73    attrs << :password if params[:action] == "create"
74
75    params.require(:member).permit(attrs)
76  end
77 end
```

createアクションとupdateアクション共通の属性のリストを配列attrsにセットし、createアクションが実行されているときだけ:passwordをこの配列に加え、permitメソッドに渡しています。params[:action]で現在実行中のアクション名が取得できます。

そして、createアクションとudpateアクションを変更します。

LIST chapter11/app/controllers/members_controller.rb

```
（省略）
33  # 会員の新規登録
34  def create
35    @member = Member.new(member_params)
36    if @member.save
37      redirect_to @member, notice: "会員を登録しました。"
38    else
39      render "new"
```

```
40     end
41   end
42
43   # 会員情報の更新
44   def update
45     @member = Member.find(params[:id])
46     @member.assign_attributes(member_params)
47     if @member.save
48       redirect_to @member, notice: "会員情報を更新しました。"
49     else
50       render "edit"
51     end
52   end
     （以下省略）
```

ブラウザで新規会員の追加と既存会員の編集を行って、正常に動作することを確認してください。

■パスワード変更機能のストロング・パラメータ対応

マイアカウントページとほぼ同様の手順で、パスワード変更機能をストロング・パラメータに対応させることができます。プライベートメソッドaccount_paramsをPasswordsControllerに加えます。

LIST chapter11/app/controllers/passwords_controller.rb

```
     （省略）
34   # ストロング・パラメータ
35   private def account_params
36     params.require(:account).permit(
```

```
37      :current_password,
38      :password,
39      :password_confirmation
40    )
41  end
42 end
```

そして、udpateアクションを変更します。

LIST chapter11/app/controllers/passwords_controller.rb

```
    （省略）
12  def update
13    @member = current_member
14    current_password = account_params[:current_password]
15
16    if current_password.present?
17      if @member.authenticate(current_password)
18        @member.assign_attributes(account_params)
    （以下省略）
```

ブラウザで自分のパスワードを変更して、正常に動作することを確認してください。

■ニュース記事管理機能のストロング・パラメータ対応

ニュース記事管理機能をストロング・パラメータに対応させます。プライベートメソッドarticle_paramsをEntriesControllerに加えます。articlesテーブルのカラムとして存在しないno_expiration属性を加えるのを忘れないようにしてください。

LIST chapter11/app/controllers/articles_controller.rb

```
  （省略）
68  # ストロング・パラメータ
69  private def article_params
70    params.require(:article).permit(
71      :title,
72      :body,
73      :released_at,
74      :no_expiration,
75      :expired_at,
76      :member_only
77    )
78  end
79 end
```

そして、createアクションとudpateアクションを変更します。

LIST chapter11/app/controllers/articles_controller.rb

```
  （省略）
40  # 新規作成
41  def create
42    @article = Article.new(article_params)
43    if @article.save
44      redirect_to @article, notice: "ニュース記事を登録しました。"
45    else
46      render "new"
47    end
48  end
49
50  # 更新
```

```
51  def update
52    @article = Article.find(params[:id])
53    @article.assign_attributes(article_params)
54    if @article.save
    （以下省略）
```

ブラウザでニュース記事の登録と編集を行って、正常に動作することを確認してください。

■ブログ記事管理機能のストロング・パラメータ対応

ブログ記事管理機能をストロング・パラメータに対応させます。プライベートメソッドentry_paramsをEntriesControllerに加えます。

LIST chapter11/app/controllers/entries_controller.rb

```
    （省略）
61  # ストロング・パラメータ
62  private def entry_params
63    params.require(:entry).permit(
64      :member_id,
65      :title,
66      :body,
67      :posted_at,
68      :status
69    )
70  end
71 end
```

そして、createアクションとudpateアクションを変更します。

LIST chapter11/app/controllers/entries_controller.rb

```
（省略）
32  # 新規作成
33  def create
34    @entry = Entry.new(entry_params)
35    @entry.author = current_member
36    if @entry.save
37      redirect_to @entry, notice: "記事を作成しました。"
38    else
39      render "new"
40    end
41  end
42
43  # 更新
44  def update
45    @entry = current_member.entries.find(params[:id])
46    @entry.assign_attributes(entry_params)
47    if @entry.save
（以下省略）
```

ブラウザでブログ記事の登録と編集を行って、正常に動作することを確認してください。

11.2 エラーページのカスタマイズ

この節では、例外の発生を捕まえて「ファイルが見つかりません」のようなエラーページをカスタマイズする方法を紹介します。

Railsの例外を処理する

Railsアプリケーションの起動中に例外が発生したときは、Railsが自動的に例外を捕捉してエラー表示を行います。エラー表示をカスタマイズしたいときは、コントローラの中でクラスメソッドrescue_fromメソッドを使用して、自分で例外を処理します。

ApplicationControllerにrescue_fromメソッドを並べていきます。コントローラはすべてApplicationControllerクラスを継承しているので、コントローラで発生した例外はここに記述したrescue_fromメソッドで処理できます。

LIST chapter11/app/controllers/application_controller.rb

```
（省略）
 7  class LoginRequired < StandardError; end
 8  class Forbidden < StandardError; end
 9
10  if Rails.env.production? || ENV["RESCUE_EXCEPTIONS"]
11    rescue_from StandardError, with: :rescue_internal_server_error
12    rescue_from ActiveRecord::RecordNotFound, with:
:rescue_not_found
13    rescue_from ActionController::ParameterMissing, with:
:rescue_bad_request
14  end
```

```
15
16  rescue_from LoginRequired, with: :rescue_login_required
17  rescue_from Forbidden, with: :rescue_forbidden
18
19  private def login_required
    （以下省略）
```

rescue_fromで5つの例外を処理しています。rescue_fromの第1引数には例外のクラスを指定します。withオプションには例外が発生したときに実行するメソッドの名前を指定します。たとえば、11行目ではfindメソッドが例外ActiveRecord::RecordNotFoundを発生させたときに、rescue_404メソッドを実行することを指定しています。

開発中はブラウザにエラーに関する情報が表示されたほうが便利なので、3つの例外は本番モードのときだけ処理しています。例外LoginRequiredとForbiddenは開発モードでも捕捉してカスタムエラーメッセージを表示します。ただし、エラー処理自体の開発をするため環境変数RESCUE_EXCEPTIONSがセットされているときには、すべての例外を捕捉します。環境変数をセットする方法については後述します。

rescue_fromの順番

rescue_fromメソッドを記述する順番に注意してください。親子関係にある例外クラスをrescue_fromに指定する場合は、親のほうを先に指定します。StandardErrorは、Rubyプログラムの実行中に発生するエラーを表すクラス（ArgumentError、NameError、RuntimeErrorなど）の親あるいは親の親です。10行目の記述を11～13行目の記述と入れ替えるとうまく動きません。

ApplicationControllerに例外発生に対応する4つのプライベートメソッドを加えましょう。

LIST chapter11/app/controllers/application_controller.rb

```
    （省略）
17  private def login_required
18    raise LoginRequired unless current_member
```

```
19  end
20
21  private def rescue_bad_request(exception)
22    render "errors/bad_request", status: 400, layout: "error",
23      formats: [:html]
24  end
25
26  private def rescue_login_required(exception)
27    render "errors/login_required", status: 403, layout: "error",
28      formats: [:html]
29  end
30
31  private def rescue_forbidden(exception)
32    render "errors/forbidden", status: 403, layout: "error",
33      formats: [:html]
34  end
35
36  private def rescue_not_found(exception)
37    render "errors/not_found", status: 404, layout: "error",
38      formats: [:html]
39  end
40
41  private def rescue_internal_server_error(exception)
42    render "errors/internal_server_error", status: 500, layout: "error",
43      formats: [:html]
44  end
45 end
```

これらのプライベートメソッドでは、renderメソッドでエラー用のテンプレートをレンダリングしています。statusオプションには、サーバーがブラウザに返すHTTPステータスコードを指定します。ここで使っているステータスコードは次のとおりです（ステータスコードについては「3.1　RailsとHTTPの基本」を参照）。

ステータスコード

ステータスコード	ステータス名	意味
400	Bad Request	リクエストの構文、形式、内容が正しくない。
403	Forbidden	リソースへのアクセス権限がない。
404	Not Found	リソースが存在しない。
500	Internal Server Error	アプリケーションでエラーが発生した。

renderメソッドにはlayoutオプションとformatsオプションも指定しています。エラー画面では、後述のerror.html.erbをレイアウトテンプレートにします。formatsオプションは、テンプレートの拡張子がhtmlであることを指定します。この指定がないと、URLのパスの拡張子がjpgやjsonのような場合にうまく処理できません。

エラー用テンプレート

■テンプレートの作成

エラー表示のためには、専用のレイアウトテンプレートを使うことにします。app/views/layoutsの下に新規ファイルerror.html.erbを作成し、次のように記述してください。

LIST chapter11/app/views/layouts/error.html.erb

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
```

```
    <title><%= page_title %></title>
    <%= stylesheet_link_tag "application", media: "all" %>
  </head>
  <body>
    <div id="container">
      <header>
        <%= image_tag "logo.gif", size: "272x48", alt: "Morning Glory" %>
        <nav class="menubar">
          <a href="/">TOP</a>
        </nav>
      </header>
      <main>
        <%= yield %>
      </main>
      <footer>
        <%= render "shared/footer" %>
      </footer>
    </div>
  </body>
</html>
```

app/viewsディレクトリの下にerrorsディレクトリを作成し、5つのテンプレートファイルを作成します。400エラー用のbad_request.html.erb、403エラー用のlogin_required.html.erbとforbidden.html.erb、404エラー用のnot_found.html.erb、500 エラー用のinternal_server_error.html.erbです。内容はそれぞれ次のようにします。

LIST chapter11/app/views/errors/bad_request.html.erb

```
<% @page_title = "Bad Request" %>
<h1>400 Bad Request</h1>
```

```
3 <p>リクエストの形式が正しくありません。</p>
```

LIST chapter11/app/views/errors/login_required.html.erb

```
1 <% @page_title = "ログインが必要です" %>
2 <h1><%= @page_title %></h1>
3
4 <%= render "shared/login_form" %>
```

LIST chapter11/app/views/errors/forbidden.html.erb

```
1 <% @page_title = "Forbidden" %>
2 <h1>403 Forbidden</h1>
3 <p>このページにはアクセスできません。</p>
```

LIST chapter11/app/views/errors/not_found.html.erb

```
1 <% @page_title = "Not Found" %>
2 <h1>404 Not Found</h1>
3 <p>ページが見つかりません。</p>
```

LIST chapter11/app/views/errors/internal_server_error.html.erb

```
1 <% @page_title = "Internal Server Error" %>
2 <h1>500 Internal Server Error</h1>
3 <p>システムエラーのためページが表示できません。</p>
```

■エラーページの確認

　では、エラーページが正しく表示されることを確認しましょう。まず、テキストエディタでconfig/environmentsディレクトリのdevelopment.rbを開き、config.consider_all_requests_localと書かれた行を探して、次のように変更します。

```
config.consider_all_requests_local = false
```

そして、ターミナルで次のコマンドを実行してサーバーを起動します。

```
$ RESCUE_EXCEPTIONS=1 bin/rails s
```

このようにコマンド本体の前にRESCUE_EXCEPTIONS=1を付けると、環境変数RESCUE_EXCEPTIONSに"1"という文字列がセットされます。この結果、本番モードと同様にすべての例外が捕捉されるようになります。

ログアウト状態で「http://localhost:3000/members」を開くと、次のようにログインフォームが表示されます。

RESULT ログインフォームの表示

404エラーを確認するには、「http://localhost:3000/articles/100」のように存在しないページを開きます。

RESULT 404エラーの表示

Chapter 15で初めて例外Forbiddenが実際に使われるので、まだ403エラーの確認はできません。また、400エラーや500エラーは普通の使い方では発生しないものです。そこで、これらのページを確認するには少し工夫が必要です。まず、config/routes.rbに次のルーティングを加えます。

LIST chapter11/config/routes.rb

```
1 Rails.application.routes.draw do
2   root to: "top#index"
3   get "about" => "top#about", as: "about"
4   get "bad_request" => "top#bad_request"
5   get "forbidden" => "top#forbidden"
6   get "internal_server_error" => "top#internal_server_error"
7
8   1.upto(18) do |n|
  （以下省略）
```

TopControllerに2つのアクションを追加します。

LIST chapter11/app/controllers/top_controller.rb

```
  （省略）
10   def bad_request
```

```
11    raise ActionController::ParameterMissing, ""
12  end
13
14  def forbidden
15    raise Forbidden, ""
16  end
17
18  def internal_server_error
19    raise
20  end
21 end
```

bad_requestアクションでは、raiseメソッドで400エラーを引き起こす例外ActionController::ParameterMissingを発生させています。第2引数には例外のメッセージとなる文字列を指定します（ここでは空文字を指定）。internal_server_errorアクションでは、引数なしでraiseメソッドを呼んで例外RuntimeErrorを発生させます。この例外はStandardErrorの子クラスです。

ブラウザで「http://localhost:3000/bad_request」を開くと、次のように400エラーが表示されます。

RESULT 400エラーの表示

同様に、ブラウザで「http://localhost:3000/forbidden」を開くと、次のように403エラーが表示されます。

RESULT 403エラーの表示

さらに、ブラウザで「http://localhost:3000/internal_server_error」を開くと、次のように500エラーが表示されます。

RESULT 500エラーの表示

サンプルのエラーページは簡素なものですが、自分のサイトに合わせてデザインに凝ったエラーページやもっと親切なエラーページを作成してもよいでしょう。ただし、エラーページがさらにエラーを起こすことがないよう、十分注意してください。

ルーティングエラーの処理

実は、以上のようにApplicationControllerを書き換えただけでは、ルーティングエラーを処理できません。Railsのコンポーネントの中でルーティングを処理するのは、Action ControllerではなくAction Dispatchです。ルーティングエラーはAction Controllerに伝わりません。

まず、config/application.rbを次のように変更します。

LIST chapter11/config/application.rb

```
  （省略）
36    config.action_controller.permit_all_parameters = false
37
38    config.exceptions_app = ->(env) do
39      ErrorsController.action(:show).call(env)
40    end
41  end
42 end
```

ここでは、Action Controllerに伝わらない例外を処理する方法として、ErrorsControllerのshowアクションを呼ぶように指定しています。この書き方は、定石として覚えてください。

そして、app/controllersディレクトリに新規ファイルerrors_controller.rbを作成し、次のような内容を書き入れます。

LIST chapter11/app/controllers/errors_controller.rb

```
1 class ErrorsController < ActionController::Base
2   layout "error"
3
4   def show
5     ex = request.env["action_dispatch.exception"]
6
7     if ex.kind_of?(ActionController::RoutingError)
8       render "not_found", status: 404, formats: [:html]
9     else
10      render "internal_server_error", status: 500, formats: [:html]
```

```
  end
 end
end
```

式request.env["action_dispatch.exception"]で発生した例外を取得できます。それがActionController::RoutingErrorのインスタンスであれば、「ページが見つかりません。」というエラーページを表示します。

なお、ルーティングエラーを発生させたURLパスが.pngや.jsonで終わっていると、formatsオプションなしのrenderメソッドはそれらの形式でレスポンスを返そうとします。しかし、asagaoにはPNG形式やJSON形式のエラーページテンプレートが用意されていないので、「Missing template」エラーが発生してしまいます。この問題を回避するため、formats: [:html]というオプションを加え、常にHTML形式でエラーページを表示するようにしています。

さて、この章の学習を終えたら、テキストエディタでconfig/environmentsディレクトリのdevelopment.rbを開き、config.consider_all_requests_localと書かれた行を探して、値をtrueに戻してから次のChapterに進みましょう。

```
config.consider_all_requests_local = true
```

Chapter 11のまとめ

- **マスアサインメント脆弱性への対策を取るには、ストロング・パラメータを有効にする必要があります。**
- **独自のエラーページを作るには、rescue_fromに例外クラスとメソッド名を指定します。**
- **ルーティングエラーを独自の方法で処理するには、アプリケーション設定項目のconfig.exceptions_appに処理方法を記述します。**

練習問題

［A］　次に示すのは書籍を管理するBooksControllerのcreateアクションを抜き出したものです。ただし、ストロング・パラメータが無効であるという前提で実装されています。

```
def create
  @book = Book.new(params[:book])
  if @book.save
    redirect_to @book, notice: "書籍を登録しました。"
  else
    render "new"
  end
end
```

そして、ストロング・パラメータを有効にした場合のBooksControllerのコードを次に示します。空欄を埋めてください。

```
class BooksController < ApplicationController
（省略）
def create
  @book = Book.new(                    )
  if @book.save
    redirect_to @book, notice: "書籍を登録しました。"
  else
    render "new"
  end
end
（省略）
```

```
  private def book_params
    params.[ ](:book).[ ](
      :title,
      :author,
      :price
    )
  end
end
```

［B］　例外ActiveRecord::RecordNotFoundが発生したときに、ステータスコード404のカスタムエラーページを表示するようにしたApplicationControllerのコードを次に示します。空欄を埋めてください。

```
class ApplicationController < ActionController::Base
  [ ] ActiveRecord::RecordNotFound, [ ]:
:rescue_404

  private def rescue_404(exception)
    render "errors/not_found", status: [ ], layout: "error",
      formats: [:html]
  end
end
```

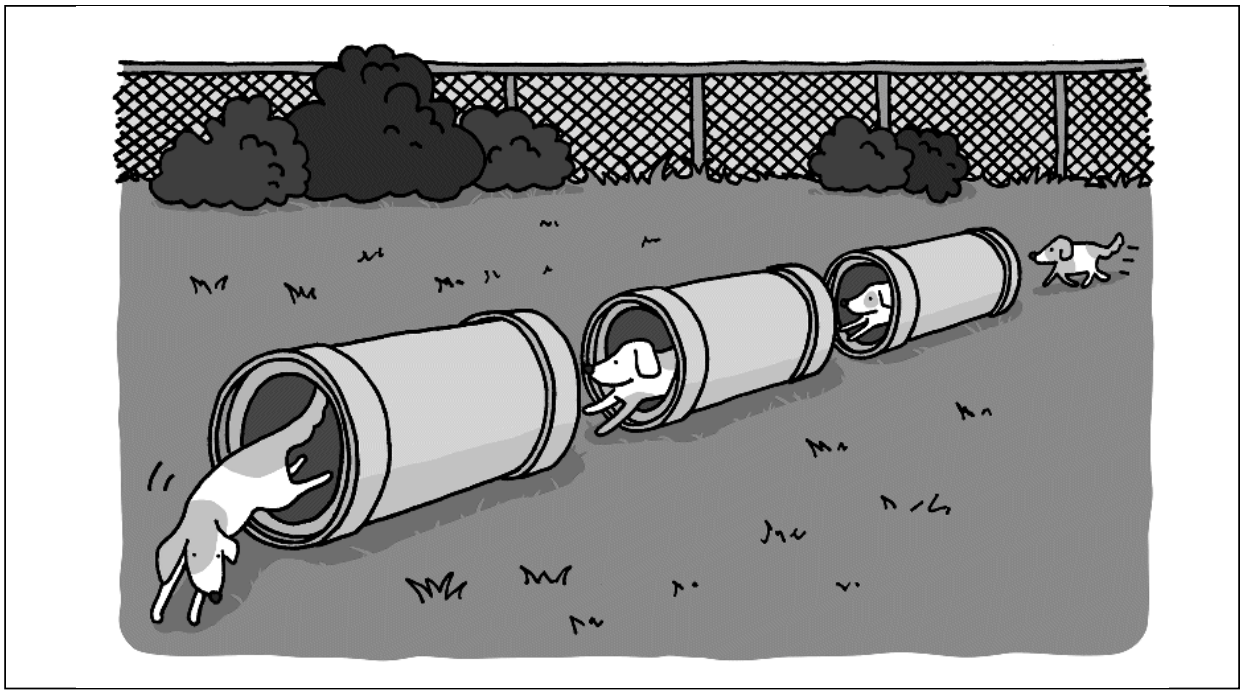
Chapter

12　アセット・パイプライン

　このChapterでは、外部サービスの秘密鍵などの資格情報を暗号化して保存する方法、本番モードでRailsサーバーを起動する方法、アセット・パイプラインを利用してスタイルシート、JavaScriptプログラム、画像ファイルなどを管理する方法などを紹介します。

これから学ぶこと

- 資格情報を暗号化して安全に管理する方法について学びます。
- 本番モードでRailsサーバーを起動する方法を習得します。
- アセット・パイプラインによるスタイルシート、JavaScriptプログラム、画像ファイルの処理について紹介します。
- JavaScriptでフォームの一部を動的に変化させる方法について学びます。



Railsのアセット・パイプラインにはどのような役割があるのでしょうか？ アセットとは何でしょうか？

12.1 暗号化された資格情報

Railsには外部サービスの秘密鍵など第三者の目から隠すべき情報を暗号化して管理する手段が標準で用意されています。

credentials.yml.encとmaster.key

本題に入る前に、Rails 5.2で導入された「暗号化された資格情報（Encrypted Credentials）」というしくみについて解説します。一般に「資格情報」とは、あるシステムやサービスを利用する権限を持つことを証明するための情報を指します。たとえば、ユーザー名、メールアドレス、パスワード、秘密鍵、公開鍵等の組み合わせが資格情報となります。当然ながら、資格情報は他人の目から隠しておく必要があります。

Railsの「暗号化された資格情報」では、configディレクトリの下に置かれる次の2つのファイルを用いて資格情報を保存します。

- credentials.yml.enc
- master.key

credentials.yml.encには資格情報が暗号化されて記録されています。master.keyは暗号を解くための「鍵」です。この2つのファイルを別々に保管すれば安全に資格情報を扱うことができます。

たとえば、私たちが作ったこのasagaoというRailsアプリケーションのソースコードを別の開発者に渡すことを考えましょう。その際、ソースコード全体をそのまま圧縮してメールに添付して送るのは危険です。第三者にメールの中身を盗み見られると、資格情報が漏れてしまいます。そこで、ソースコードを他人に渡すときはmaster.keyを除外しておき、別の安全な方法でmaster.keyを渡すようにするわけです。

さて、資格情報を編集するにはターミナルで次のコマンドを実行します。

```
$ bin/rails credentials:edit
```

するとテキストエディタnanoまたはVimが起動します。どちらになるかは、「1.3　テキストエディタの選択」で設定した環境変数EDITORの値によります。テキストエディタの編集画面には次のようなYAML形式のデータが現れます。

```
# aws:
#   access_key_id: 123
#   secret_access_key: 345
(以下省略)
```

これを編集してからテキストエディタを終了すると資格情報が暗号化されて、credentials.yml.encに書き込まれます。

secret_key_baseの生成

Railsアプリケーションを本番モードで起動するためには、credentials.yml.encにおいてsecret_key_baseというキーに値がセットされていなければなりません。この値は、セッション情報の署名やクッキーの暗号化に使われます。

rails newコマンドでRailsアプリケーションの骨格を生成した際にsecret_key_baseが設定されますが、master.keyを紛失してしまった場合には作り直す必要があります。また、本書のサポートサイトからダウンロードしたasagaoのソースコードにはcredentials.yml.encもmaster.keyも含まれません。

新たなmaster.keyとcredentials.yml.encのセットを生成するには、まずcredentials.yml.encを削除します。そして、ターミナルで前項で説明したコマンドを実行します。

```
$ bin/rails credentials:edit
```

テキストエディタが開いたら、そのまま閉じてください。configディレクトリに2つのファイルが作られているはずです。

資格情報の設定と参照

credentials.yml.encにはsecret_key_base以外の任意の資格情報を保存することができます。bin/rails credentials:editコマンドによりテキストエディタを開き、たとえば次のような設定を加えたとします。

```
foo:
  bar: example
```

すると、Railsアプリケーションのソースコードの中で次のように書けば、文字列exampleを取り出すことができます。

```
Rails.application.credentials.dig(:foo, :bar)
```

Chapter 13では各種ストレージサービスの資格情報を暗号化して保存するために、この方法を使います。

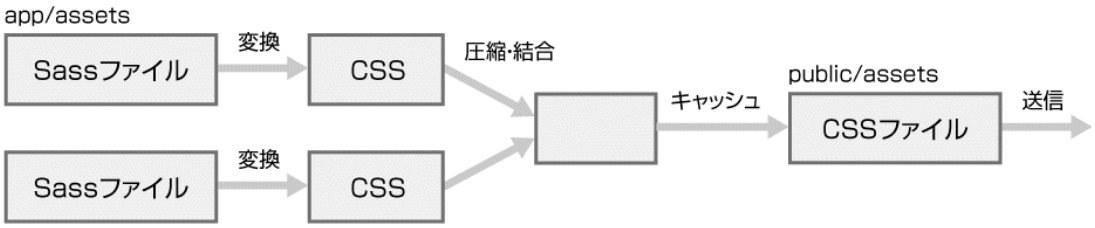
12.2 アセット・パイプライン

アセット・パイプラインは、CSSやJavaScriptの読み込みを高速化するための機能です。ここでは、アセット・パイプラインのしくみと役割について解説します。

アセット・パイプラインの働き

アセット・パイプラインの「アセット」は「資産」の意味で、画像ファイル、スタイルシート、JavaScriptプログラムなどを指します。アセット・パイプラインは、次の機能からなります。

- 変換：Sassで書かれたスタイルシートをCSS形式に変換したり、CoffeeScriptのスクリプトをJavaScriptに変換したりする。
- 圧縮と結合：CSSやJavaScriptの改行や空白を除去してサイズを縮小する。また、複数のCSSやJavaScriptを1つのファイルにまとめる。
- キャッシュ：public/assetsディレクトリの下にキャッシュファイルを生成する。ブラウザにはキャッシュファイルを送信する。



本番モード用のアセットファイル

アセット・パイプラインは、app/assetsディレクトリの下にあるCSS、JavaScript、画像に対して働きます。こうしたファイルをいわば「加工前の材料」と見なし、加工した結果をブラウザに送信することで高速化を図ります。また、変換機能によってブラウザが対応していないSassやCoffeeScriptのような新しい形式のファイルが使えるようになります。

一方、本番サーバー上でアプリケーションの起動中にCSSやJavaScriptを変更しても、変更結果は反映されません。後述のRakeタスクassets:precompileを実行してキャッシュファイルを生成する必要があります。

 本番モードの準備

アセット・パイプラインの動作の全体は、開発モードではなく本番モードで確認できます。パソコン上のアプリケーションを本番モードで動かしてみましょう。

Railsアプリケーションを本番モードで動かすためには、次の2つの環境変数をセットする必要があります。

- RAILS_ENV
- RAILS_SERVE_STATIC_FILES

まず、環境変数RAILS_ENVについては「4.1　データベースとモデルの基本」で説明しました。productionという値をセットすれば本番モードとなります。環境変数RAILS_SERVE_STATIC_FILESは、publicディレクトリの静的ファイル（CSS、JavaScript、画像などのファイル）をRailsアプリケーションがブラウザに返すかどうかを決めます。本番モードでは「返さない」という設定がデフォルトですが、この環境変数に何らかの値をセットすれば「返す」設定になります。

ターミナルで次のコマンドを実行してください。

```
$ export RAILS_ENV=production
$ export RAILS_SERVE_STATIC_FILES=1
```

公開サーバーでの静的ファイルの取り扱い

公開サーバーにおいては、ApacheやNginxなどのウェブサーバーが静的ファイルを直接ブラウザに返すように設定するのが一般的です。そのため、Railsの本番モードではpublicディレクトリのファイル

を返さない設定になっています。この節ではApacheやNginxを使わずに解説しますので、環境変数RAILS_SERVE_STATIC_FILESの設定を変更しています。

アセット・パイプラインの動作確認

本番モードでアプリケーションを起動するには、次のコマンドを実行して、キャッシュファイルを生成する必要があります。これを実行しないとエラーになります。

```
$ bin/rails assets:precompile
```

本番用のデータベースを作成します。環境変数DISABLE_DATABASE_ENVIRONMENT_CHECKに"1"をセットすることにより、本番用データベースに対する保護を解除しています。

```
$ DISABLE_DATABASE_ENVIRONMENT_CHECK=1 bin/rails db:setup
```

そして、Railsサーバーを起動します。

```
$ bin/rails s
```

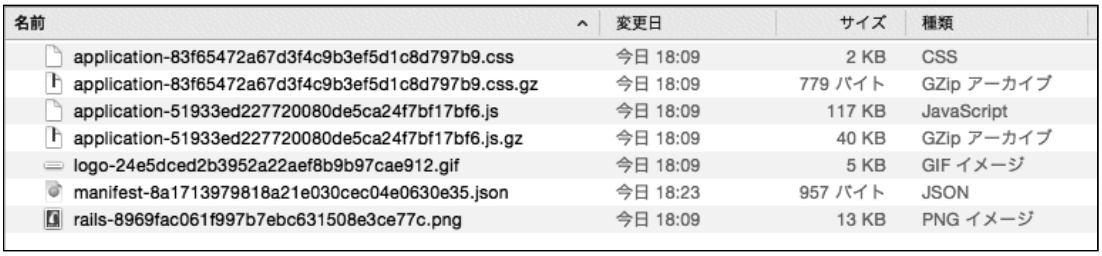
db:setupタスク

Railsのdb:setupタスクは、データベースの作成、スキーマ（テーブルとカラム定義）の読み込み、シードデータの投入という一連の作業を行います。db:resetタスクと似ていますが、最初にデータベースを削除しない点が異なります。

ブラウザで「http://localhost:3000/」を開いてトップページを表示し、HTMLのソースを見てみましょう。CSSファイルとJavaScriptファイルにリンクするタグがそれぞれ1つにまとめられ、「application-83f6547...css」のような長いファイル名になっています。

```
<link rel="stylesheet" media="all" href="/assets/application-
83f65472a67d
3f4c9b3ef5d1c8d797b9.css" data-turbolinks-track="true" />
<script src="/assets/application-51933ed227720080de5ca24f7bf17bf6.js"
dat
a-turbolinks-track="true"></script>
```

public/assetsディレクトリの下を見てみましょう。「rake assets:precompile」によって生成されたファイルが置かれています。「application-83f6547...css」や「application-51933ed...js」の中身をテキストエディタで見ると、ファイルを小さくするために空白や改行が取り除かれていることがわかります。



名前	変更日	サイズ	種類
application-83f65472a67d3f4c9b3ef5d1c8d797b9.css	今日 18:09	2 KB	CSS
application-83f65472a67d3f4c9b3ef5d1c8d797b9.css.gz	今日 18:09	779 バイト	GZip アーカイブ
application-51933ed227720080de5ca24f7bf17bf6.js	今日 18:09	117 KB	JavaScript
application-51933ed227720080de5ca24f7bf17bf6.js.gz	今日 18:09	40 KB	GZip アーカイブ
logo-24e5dced2b3952a22aef8b9b97cae912.gif	今日 18:09	5 KB	GIF イメージ
manifest-8a1713979818a21e030cec04e0630e35.json	今日 18:23	957 バイト	JSON
rails-8969fac061f997b7ebc631508e3ce77c.png	今日 18:09	13 KB	PNG イメージ

本番モード用のアセットファイル

なお、アセット・パイプラインがCSSファイルやJavaScriptファイルにリンクするタグをまとめるのは、本番モードだけです。「3.4　モックアップの作成」で見たとおり、開発モードでは、ファイルごとにタグが作成されます。

動作を確認したらサーバーを終了して、ターミナルを閉じ、新たにターミナルを開き直してください。これで環境変数の値がリセットされます。

publicディレクトリの下に置く場合

画像、CSSファイル、JavaScriptをpublicディレクトリの下に置くこともできます。その場合は、stylesheet_link_tag、javascript_include_tag、image_tagに指定するファイル名を「/」で始まるパスに変えます。こうして指定したファイルは、アセット・パイプラインで処理されません。

```
<%= stylesheet_link_tag "/stylesheets/layout" %>
```

キャッシュとフィンガープリント

Railsのassets:precompileタスクで生成したCSS、JavaScript、画像のファイル名には、「application-83f6547...css」の「83f6547...」のようなフィンガープリント（指紋）が付きます。フィンガープリントは、ファイルの内容から生成されたもので、ファイルの内容が更新されれば値が変化します。これによって、ブラウザや各種のサーバーがキャッシュに保存した古い内容を利用しないようにしています。つまり、CSS、JavaScript、画像ファイルを更新すれば、確実に新しいものがブラウザに反映されます。

12.3 Sass

アセット・パイプラインとともに使われるSassの機能を試してみましょう。

Sassの書式

Sass（Syntactically Awesome Stylesheets）は、CSSの文法を改良したスタイルシート言語です。ブラウザに読み込ませるときは、Sassのテキストを通常のCSSに変換して使います。

本書では、Sassの文法について細かい解説はしません。文法の詳細については、Sassのウェブサイトを参照してください。

Sass: Syntactically Awesome Stylesheets
http://sass-lang.com/

次のCSSをSassに変えてみましょう。id属性がerrorsのdivタグと、そのdivタグの中にあるh2タグのスタイルです。

```
div#errors { background-color: #fee; }
div#errors h2 { color: red; }
```

Sassでは、タグの入れ子関係を{ }の組み合わせでわかりやすく表すことができます。

```
div#errors {
  background-color: #fee;
  h2 {
    color: red;
```

```
  }
}
```

RailsでSassのこの書式を使うときは、スタイルシートファイルの拡張子を.scssにします。

SassとSCSS

Sassには2種類の書式があります。古い書式では、タグの入れ子関係をインデントで表します。現在では、上記の例のように{ }を組み合わせた新しい書式がもっぱら使われています。新しいほうの書式をSCSS（Sassy CSS）と呼びます。

本書を読むうえでは、「SassとSCSSは同じもの」と考えれば十分です。

Sassを使う

asagaoアプリケーションの一部のCSSをSass（SCSS形式）に変えてみましょう。app/assets/stylesheetsディレクトリの下にpagination.cssというファイルがあります。これのファイル名をpagination.scssに変更し、内容を次のように書き換えてください。

LIST chapter12/app/assets/stylesheets/pagination.scss

```
1 $border-color: #499;
2
3 nav.pagination {
4   font-size: 75%;
5   padding: 4px 8px;
6   border: 1px solid $border-color;
7   word-spacing: 4px;
8
9   span.current {
10    font-weight: bold;
```

```
11   }
12 }
```

SCSSの特徴のひとつは、変数が使えることです。1行目で$border-colorという変数に#499という値をセットし、6行目で利用しています。

また、SCSSではセレクタをネストすることができます。9〜11行目は、CSSでは次のように書かなければなりませんでした。

```
nav.pagination span.current {
  font-weight: bold;
}
```

変換後の結果を確認するには、開発モードでサーバーを起動し、ブラウザで「http://localhost:3000/assets/pagination.css」を開いてください。

CSSやJavaScriptでの画像指定

CSSやJavaScriptの中では、HTMLテンプレートと同様に<%=と%>で囲んでRubyのコードを埋め込めます。ただし、この記法を利用する場合は、header.css.erbのように、ファイル名の最後に拡張子.erbを付加します。次の例をご覧ください。background-imageプロパティに画像ファイルを指定しています。

```
div#header {
  background-image: url(<%= asset_path "rails.png" %>);
}
```

asset_pathメソッドは、フィンガープリントを含む/assets/rails-15ff13e9c3486ddd78bc91be8063523d.pngのようなファイル名を返します。

12.4 JavaScript

この節では、JavaScriptライブラリjQueryをasagaoに導入し、ユーザーがブラウザ上で行った操作に応じて入力フォームの一部を表示したり、隠したりする効果を実現します。

jQueryの導入

jQuery（ジェイクエリー）は、ブラウザ上でのDOM操作やAjax呼び出しを行うためのJavaScriptライブラリです。Rails 5.0まではrails newコマンドで生成されるGemfileにGemパッケージjquery-railsが記載されていたためはじめからjQueryが使えましたが、Rails 5.1以降は除外されることになりました。そこで、まずはGemfileを次のように書き換えてください。

LIST chapter12/Gemfile

```
（省略）
38 gem 'kaminari-i18n'
39 gem 'jquery-rails'
40
41 group :development, :test do
（以下省略）
```

そして、追加したGemパッケージをインストールします。

```
$ bundle install
```

さらに、app/assets/javascriptsディレクトリのapplication.jsを次のように書き換えてください。

LIST chapter12/app/assets/javascripts/application.js

```
（省略）
13 //= require rails-ujs
14 //= require activestorage
15 //= require turbolinks
16 //= require jquery
17 //= require_tree .
```

アセット・パイプラインで処理されるとき、行頭の記号//=が特別な意味を持ちます。この記号に続くrequireやrequire_treeはディレクティブと呼ばれ、アセット・パイプラインに対して特定の処理を指示します。ここでは「JavaScriptライブラリjQueryのソースコードをここに読み込め」という意味になります。

ニュース記事編集フォームの拡張

では、簡単なJavaScriptプログラミングに挑戦してみましょう。app/assets/javascriptsディレクトリに新しくファイルarticles.jsを作り、次のコードを入力してください。

LIST chapter12/app/assets/javascripts/articles.js

```
1 $(document).on("turbolinks:load", function() {
2   var cb = $("#article_no_expiration");
3   var field = $("#article_expired_at");
4
5   var changeExpiredAt = function() {
6     if (cb.prop("checked"))
7       field.hide()
```

```
 8     else
 9       field.show()
10   }
11
12   cb.bind("click", changeExpiredAt);
13
14   changeExpiredAt();
15 })
```

このJavaScriptプログラムはChapter 9で作ったニュース記事編集のフォームに機能を加えます。［掲載終了日時を設定しない］をチェックすると日時の入力欄を隠し、チェックを外すと入力欄を表示します。

1行目はjQueryを用いるときの決まり文句です。ブラウザで新しいページが表示されるとturbolinks:loadイベントが発生します。そのときに2行目以下のコードが実行され、日時入力欄の表示・非表示を切り替える機能がasagaoに組み込まれます。

6行目の条件式cb.prop("checked")は、チェックボックス［掲載終了日時を設定しない］にチェックが入っていればtrueを、チェックが外れていてればfalseを返します。

12行目ではjQueryのbindメソッドを用いて、チェックボックスがクリックされたときに関数changeExpiredAtが呼び出されるようにしています。14行目では、ページが読み込まれた直後に関数changeExpiredAtを呼び出しています。

JavaScriptに変換されたソースコードを確認するには、ブラウザで「http://localhost:3000/assets/articles.js」を開いてください。

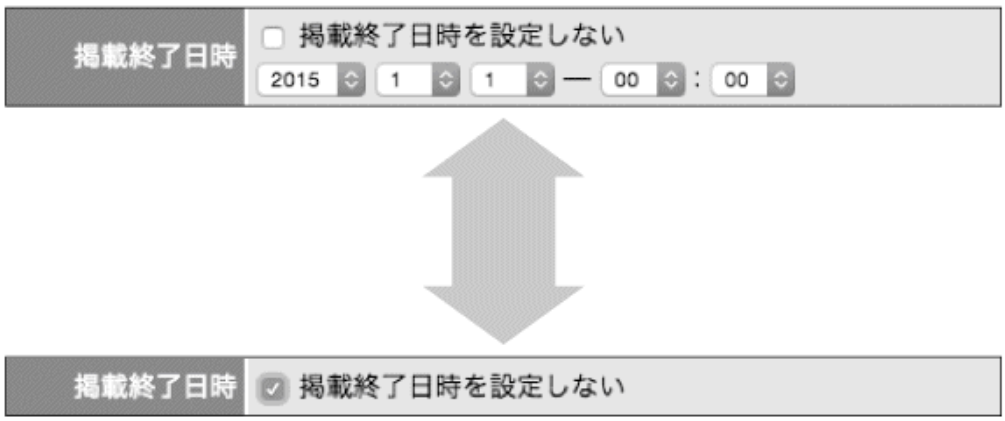
このJavaScriptプログラムを動かすため、フォームのテンプレートを次のように修正してください。

LIST chapter12/app/views/articles/_form.html.erb

```
   （省略）
20     <td>
21       <div>
```

```
22        <%= form.check_box :no_expiration %>
23        <%= form.label :no_expiration %>
24      </div>
25      <div id="article_expired_at">
26        <%= form.datetime_select :expired_at,
27            start_year: 2000, end_year: Time.current.year + 1,
28            use_month_numbers: true %>
29      </div>
30    </td>
    （以下省略）
```

掲載終了日時の入力欄を囲むdivタグにid属性を設定しています。その値article_expired_atがarticles.jsの3行目にある$("#article_expired_at")と対応します。この部分がhide()メソッドによって隠されたり、show()メソッドによって表示されたりします。



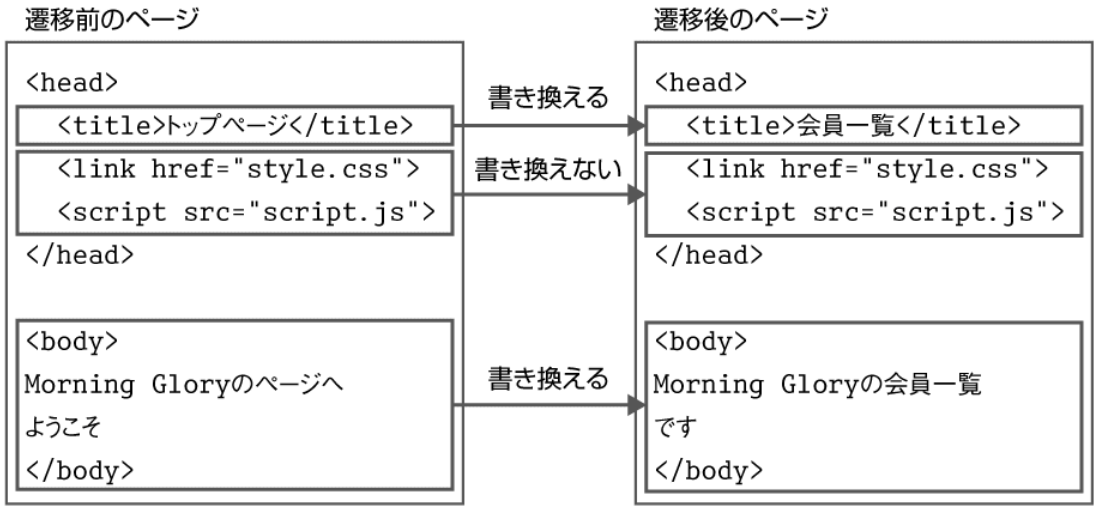
JavaScriptで作ったフォームの機能

Turbolinks

ここでは、JavaScriptに関連した機能として、Turbolinksについて簡単に説明しておきましょう。

Turbolinksは、ブラウザによるページの遷移を高速化する機能です。ブラウザ上で何かリンクをクリックして別のページに移ると、ブラウザはHTMLを読み込み、リンクされているCSSとJavaScriptをすべて解釈し直します。アセット・パイプラインを使うとCSSとJavaScriptの読み込みは高速になりますが、この「解釈」にかかる時間は減りません。

Turbolinksは、ページの遷移機能をブラウザから横取りし、JavaScriptを使ってサーバーからHTMLを取得してページを書き換えます。linkタグやscriptタグはページを遷移しても書き換えずにそのままにしておくので、解釈にかかる時間をなくすことができます。



Turbolinks

「3.4　モックアップの作成」で見たように、レイアウトテンプレートのheadタグには、次のようなstylesheet_link_tagとjavascript_include_tagがありました。

```
    <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-track': 'reload' %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reload' %>
```

この「'data-turbolinks-track': 'reload'」は、linkタグやscriptタグにdata-turbolinks-track="reload"という属性を加えます。Turbolinksはページ遷移のときに、この属性が付いているタグを書き換えないようにします。

なお、ブラウザでページを再読み込みしたときやURL欄に直接URLを入れたときは、Turbolinksは働かず、通常のページ読み込みとなります。

Turbolinksを使うときの注意

Turbolinksを使ったサイトでは、JavaScriptプログラムの記述に注意してください。jQueryを使ってページの読み込み時にある関数を実行する場合は、通常はreadyイベントに対して関数を紐付けますが、Turbolinksを利用する場合は対象がturbolinks:loadイベントとなります。

```
$(document).on("turbolinks:load", function () {
  // 読み込み時のスクリプト
});
```

また、ページを移動するときはlocation.hrefに値を設定しないでください。次のように、Turbolinks.visitにパスを指定するのが正しいやり方です。

```
Turbolinks.visit("/");
```

Webpacker

近年、JavaScriptの開発ではwebpackというビルドツールがよく使われています。特に、React、Angular、Vueなどのフレームワークを利用したい場合はほぼ必須となります。筆者自身も好んで使っています。

GemパッケージWebpacker（末尾の「er」に注意）を導入すると、Railsアプリケーションとwebpackを連携できます。実際、Rails 5.1では--webpackオプション付きでrails newコマンドを実行すると、Webpackerのセットアップまでを自動的に行ってくれます。

さて、本書ではWebpackerに関する解説を割愛することにしました。webpack、Node.js、Yarnなどの各種ツール類のセットアップから実際のプログラミング例までを説明しようとすれば、それだけで一冊の本になってしまいます。プログラミング言語JavaScriptそれ自体が腰を据えて学習すべき大きなテーマなのです。

今後、Rails開発者の間でWebpackerが普及するにつれ、アセット・パイプラインの重要性は下がっていくことになるでしょう。しかし、近い将来にアセット・パイプラインが廃れることもなさそうです。すでにアセット・パイプラインを利用したRailsアプリケーションは無数に存在し、これらをWebpackerのために書き換えるのは骨の折れる作業です。また、アセット・パイプラインのほうが導入のハードルがかなり低いので、JavaScriptプログラミングの比重が小さい開発プロジェクトでは使われ続けるはずです。そもそもアセット・パイプラインとWebpackerは共存できるので、webpackを使いたいという理由でアセット・パイプラインを捨てる必要はありません。

Chapter 12のまとめ

- 暗号化された資格情報を編集するには、ターミナルでbin/rails credentials:editコマンドを実行します。
- アセット・パイプラインの「資産（アセット）」とは、画像ファイル、スタイルシート、JavaScriptプログラムなどを指します。
- アセット・パイプラインの機能は、①変換、②圧縮と結合、③キャッシュからなります。
- キャッシュファイルを生成するには、ターミナルでbin/rails assets:precompileコマンドを実行します。
- スタイルシートの記述にはSass/SCSSの記法が使えます。
- JavaScriptライブラリjQueryはRails 5.1からデフォルトではインストールされません。Gemパッケージjquery-railsを導入する必要があります。

練習問題

［A］　次の文の内容が正しい場合は○、間違っている場合は×を記入してください。

- （　）アセット・パイプラインにより、configディレクトリの下にあるファイルmaster.keyの中身が暗号化されます。
- （　）アセット・パイプラインにより、app/assets/stylesheetsディレクトリの下にあるSass形式のスタイルシートはCSS形式に変換されます。
- （　）バージョン5.1以降のRuby on RailsにはJavaScriptライブラリjQueryが同梱されています。
- （　）Turbolinksは、JavaScriptのコンパイル時間を短縮するための機能です。

［B］　ターミナルでbin/rails credentials:editコマンドを実行し、「暗号化された資格情報」の内容を次のように編集しました。

```
remote:
  email: "taro@example.jp"
  password: "p@ssw0rd"
```

次に示すのは、Railsアプリケーションの中で "p@ssw0rd" という文字列を取り出すための式です。空欄を埋めてください。

```
Rails.application.
```

Chapter

13　ファイルのアップロード

　このChapterでは会員のプロフィール画像やブログ記事の添付画像をアップロードする機能をMorning Gloryのサイトに追加します。

これから学ぶこと

- Active Storageを使ってファイルをローカルディスクやクラウドストレージサービスにアップロードする方法を学びます。
- アップロードされたファイルとモデルオブジェクトの結び付け方を学習します。
- Morning Gloryの会員が自分のプロフィール写真やブログ記事の添付画像をアップロードする機能を作ります。
- モデルにカスタム属性を追加する方法を学びます。
- アップロードされた画像を指定されたサイズに拡大・縮小してページ上に表示します。
- モデルオブジェクトのリストの表示順を入れ替える方法を学習します。
- 主要なクラウドストレージサービスの資格情報を設定する手順を解説します。

モデルとモデルを多対多で関連付けるためには、データベース設計上でどんな工夫が必要となるでしょうか？

13.1 Active Storage

ウェブアプリケーションでは、HTMLフォームを通じて画像ファイルやPDFファイルなどをユーザーにアップロードさせることがあります。Rails 5.2で導入されたActive Storageを利用すると、そうした処理を簡単に実現できます。

Active Storageとは

■Active Storageのしくみ

Active StorageはRails 5.2で導入された新しい機能です。これを利用すれば、Amazon S3、Google Cloud Storage、Microsoft Azure Storage等のクラウドストレージサービスへ簡単にファイルをアップロードできるようになります。アップロードするファイルの中身は、画像、動画、音声、WordやPDF形式の文書等、形式は問いません。

Active Storageはまた、クラウドストレージサービスに保存されたファイルを配信する手段も提供します。それぞれのファイルに対して個別のURLが割り当てられます。ブラウザからこのURLに対して要求が届くと、Railsはクラウドサービス上のURLにリダイレクションします。

たとえば、あるRailsアプリケーションのトップページのURLがhttps://example.jpであるとします。このサイトからActive Storage経由でtest.pngというファイルがクラウドストレージサービスAmazon S3にアップロードされたとします。このとき、このファイルには、次のような形式のURLが割り当てられます。ただし、...の部分には、200を超える長さのランダムな文字列が入ります。

```
https://example.jp/rails/active_storage/representations/.../test.png
```

ブラウザがこのURLに対してアクセスすると、Railsは次のような形式のURLに対してリダイレクションします。ただし、...の部分は（さまざまなパラメータを含む）500を超える長さの文字列が入ります。

```
https://s3.ap-northeast-1.amazonaws.com/...
```

これは、Amazon S3の用語で事前署名付きURL（pre-signed URL）と呼ばれるものです。このURLには5分間の有効期限が設定されています。Google Cloud StorageやMicrosoft Azure Storageなどのほかのクラウドストレージサービスを利用する場合でも、だいたい同じようなしくみでファイルが配信されます。最終的なURLに有効期限が設定されているため、ファイルが一般に公開された状態にはなりません。

Active Storageのアクセス制御

Active Storageでは、ランダムな部分を含む複雑なURLから有効期限の設定された署名付きURLへのリダイレクションにより、第三者がアクセスできないようにしています。しかし、厳密に言えばファイルにアクセス制限がかかっていることにはなりません。

なぜなら、リダイレクション前のURLを知る人間がそれをコピーして広く配布すれば、誰でもファイルをダウンロードできる状態になるからです。5分間の有効期限にも縛られることもありません。ファイルに厳格なアクセス制限をかけるためには、署名付きURLへのリダイレクションを行うアクションを自分で実装する必要があります。

しかし、法律的な議論を無視すれば、リダイレクション前のURLを配布する行為とファイル自体を配布する行為の間に本質的な違いはありません。実用上はActive Storageが提供するレベルのアクセス制御で十分だと筆者は考えています。

■ディスクサービス

Active Storageは、アプリケーションサーバーのローカルディスクにファイルをアップロードすることもできます。この場合、アップロード先をディスクサービスと呼びます。ディスクサービスはおもに開発環境とテスト環境で利用されます。

ディスクサービスにアップロードされたファイルは、Railsルートディレクトリ直下のstorageディレクトリに置かれます。ディスクサービスを利用している場合でも、それぞれのファイルに対して個別のURLが割り当てられて、そのURLから別のURLにリダイレクションが行われる点は同じです。ただし、クラウドストレージサービスの場合と異なり、リダイレクション先のURLに有効期限は設定されません。

本番環境ではクラウドストレージサービスの利用がお勧め

本番環境でもディスクサービスを利用することは可能ですが、長期の安定的なシステム運用を目指すならクラウドストレージサービスを使ったほうがよいでしょう。

本番環境ではロードバランサの背後に複数台のアプリケーションサーバーを配置するシステム構成をしばしば採用します。ディスクサービスを利用した場合、Active Storageによってアップロードされたファイルは特定のアプリケーションサーバーのディスクにだけ書き込まれることになります。このファイルをすべてのアプリケーションサーバーから配信できるようにするには、それらの間でstorageディレクトリの中身を同期するしくみを用意しなければなりません。これはなかなか簡単なことではありません。アプリケーションサーバーの増設やアップロードされたファイル群のバックアップといった保守作業のことも考慮すると、さらにハードルが高くなります。

セットアップ手順

ImageMagickのインストール

Active Storageには画像のサイズを変換する機能がありますが、この機能を利用するにはImageMagickというソフトウェアが必要です。プラットフォーム別の手順に従ってインストールしてください。

macOSの場合

```
$ brew install imagemagick
```

WSL/Ubuntuの場合

```
$ sudo apt-get install -y imagemagick
```

MSYS2/MinGW（64ビット版）の場合

```
$ pacman -S mingw-w64-x86_64-imagemagick
```

MSYS2/MinGW（32ビット版）の場合

```
$ pacman -S mingw-w64-i686-imagemagick
```

■Gemパッケージmini_magickの導入

RailsでImageMagickを利用するためGemfileを次のように書き換えます（行頭のコメント記号を除去）。

LIST chapter13/Gemfile

```
（省略）
26 # Use ActiveStorage variant
27 gem 'mini_magick', '~> 4.8'
（以下省略）
```

そして、ターミナルで次のコマンドを実行します。

```
$ bundle install
```

■マイグレーションスクリプトの生成と実行

続いて、ターミナルで次のコマンドを実行します。

```
$ bin/rails active_storage:install
Copied migration
20180530032324_create_active_storage_tables.active_storage.rb from
active_storage
```

結果として、db/migrateディレクトリに2つのマイグレーションスクリプトが作られます。マイグレーションを実行してください。

```
$ bin/rails db:migrate
== 20180530032324 CreateActiveStorageTables: migrating
==========================
-- create_table(:active_storage_blobs)
   -> 0.0025s
-- create_table(:active_storage_attachments)
   -> 0.0027s
== 20180530032324 CreateActiveStorageTables: migrated (0.0054s)
===============
```

以上で、Active Storageを利用する準備が整いました。

13.2 プロフィール画像のアップロードと表示

この節では、Morning Gloryの会員一人ひとりに対してプロフィール画像を登録する機能を作ります。クラスメソッドhas_one_attachedを用いて会員と画像を結び付ける方法について学びましょう。

Memberモデルの拡張

■クラスメソッドhas_one_attached

Active Storageが提供するクラスメソッドhas_one_attachedを使うと、モデルオブジェクトに対して1個のファイルを添付できるようになります。Memberモデルのソースコードを次のように書き換えてください。

LIST chapter13/app/models/member.rb

```
1 class Member < ApplicationRecord
2   has_secure_password
3
4   has_many :entries, dependent: :destroy
5   has_one_attached :profile_picture
6
7   validates :number, presence: true,
    （以下省略）
```

クラスメソッドhas_one_attachedは、1個のファイルを添付するための属性をモデルクラスに追加します。上記の変更の結果、Memberモデルに属性profile_pictureが追加されま

す。

■クラスメソッドattribute

続いて、Memberモデルのソースコードを次のように書き換えてください。

LIST chapter13/app/models/member.rb

```
（省略）
5  has_one_attached :profile_picture
6  attribute :new_profile_picture
7
8  validates :number, presence: true,
（以下省略）
```

クラスメソッドattributeは、モデルに読み書き可能な属性を追加します。ここでは、Memberモデルに属性new_profile_pictureを加えています。データベーステーブルmembersにnew_profile_pictureというカラムはありませんが、あたかもそのようなカラムが存在するかのようにプログラミングできます。私たちは、この属性を新しいプロフィール画像を一時的に保存するために利用します。

Chapter 2で紹介したクラスメソッドattr_accessorも同様の働きをしますが、attributeで追加された属性には、その値が変化するとモデルオブジェクトが「データベースへの保存が必要である」状態に変わるという特徴があります。あとでこの特徴を利用します。

「プロフィール画像」フィールドの設置

では、会員情報の編集ページとマイアカウントページに「プロフィール画像」をアップロードするためのフィールドを追加しましょう。これらのページで共用している部分テンプレートを次のように書き換えてください。

LIST chapter13/app/views/shared/_member_form.html.erb

```
<%= render "shared/errors", obj: @member %>

<table class="attr">
  <tr>
    <th><%= form.label :new_profile_picture %></th>
    <td><%= form.file_field :new_profile_picture %></td>
  </tr>
  <tr>
    <th><%= form.label :number %></th>
    <td><%= form.text_field :number, size: 8 %></td>
  </tr>
  （以下省略）
```

フォームビルダーのfile_fieldメソッド（6.2節）でファイルアップロード用の部品をフォーム内に設置しています。フィールドの属性にprofile_pictureではなくnew_profile_pictureを指定する点に注意してください。

次に、new_profile_pictureフィールドのラベルが日本語で表示されるように、日本語用のロケールテキストを追加します。

LIST chapter13/config/locales/ja.yml

```
  （省略）
    attributes:
      member:
        new_profile_picture: プロフィール画像
        number: 背番号
        name: ユーザー名
  （以下省略）
```

続いて、ストロング・パラメータ（11.1節）の設定を変更します。まず、会員情報の変更を受け付けるコントローラのmember_paramsメソッドを次のように書き換えます。これによりprofile_pictureが「許可された属性」となります。

LIST chapter13/app/controllers/members_controller.rb

```
（省略）
61  # ストロング・パラメータ
62  private def member_params
63    attrs = [
64      :new_profile_picture,
65      :number,
66      :name,
（以下省略）
```

同様に、マイアカウントの変更を受け付けるコントローラのaccount_paramsメソッドを次のように書き換えます。

LIST chapter13/app/controllers/accounts_controller.rb

```
（省略）
22  # ストロング・パラメータ
23  private def account_params
24    params.require(:account).permit(
25      :new_profile_picture,
26      :number,
27      :name,
（以下省略）
```

まだ、画像アップロード機能は完成していませんが、この段階で動作確認をしましょう。ブラウザで会員TaroとしてMorning Gloryのサイトにログインし、マイアカウントページから［ア

カウント情報の編集］へ進むと次のような画面が表示されます。

RESULT フォームにプロフィール画像フィールドが追加されている

そして、適当な画像ファイルを「プロフィール画像」として選択し、［更新する］ボタンをクリックしてください。その結果、何もエラーが出なければOKです。

画像アップロード機能の実装

ここまでの変更により、Memberオブジェクトの属性new_profile_pictureに画像データがセットされるようになりました。しかし、この属性はクラスメソッドattributeで作られた仮想的な属性に過ぎません。この属性のデータを属性profile_pictureにもセットする必要があります。Memberモデルのソースコードを次のように書き換えてください。

LIST chapter13/app/models/member.rb

```
（省略）
28  validates :password, presence: { if: :current_password }
29
```

```
30  before_save do
31    if new_profile_picture
32      self.profile_picture = new_profile_picture
33    end
34  end
    （以下省略）
```

このコード追加により、画像アップロード機能が実際に動くようになります。ブラウザで動作確認をしてください。

なお、32行目は次のように書くこともできます。効果はまったく同じです。この書き方もよく使われるので、覚えておきましょう。

```
self.profile_picture.attach(new_profile_picture)
```

ところで、読者の皆さんは、なぜこんな面倒なことをするのだろうかと疑問に思われたのではないでしょうか。そもそも属性new_profile_pictureは何のために存在するのでしょうか。確かに、フォームから画像データをパラメータprofile_pictureとして送り、Memberオブジェクトの属性profile_pictureに直接セットしてもよさそうですね。

しかし、それではダメなのです。なぜなら、Memberオブジェクトの属性profile_pictureに画像データをセットすると、その瞬間に画像データが保存されてしまうからです。つまり、Memberオブジェクトでバリデーションエラーが発生しても会員のプロフィール画像が書き換わってしまうのです。これは、直感に反した動きです。また、次節で行う画像データ自体のバリデーションも不可能となります。

 アップロードされた画像の表示

では、会員の詳細ページやマイアカウントページにプロフィール画像が表示されるようにしましょう。これらのページで共用している部分テンプレート_body.html.erbを次のように書き換え

てください。

LIST chapter13/app/views/members/_body.html.erb

```
 1 <table class="attr">
 2   <tr>
 3     <th>プロフィール画像</th>
 4     <td>
 5       <% if @member.profile_picture.attached? %>
 6         <%= image_tag @member.profile_picture.variant(resize:
"128x128") %>
 7       <% end %>
 8     </td>
 9   </tr>
10   <tr>
11     <th width="150">背番号</th>
12     <td><%= @member.number %></td>
13   </tr>
     （以下省略）
```

変更点の解説はあと回しにして、まずは動作確認をしましょう。会員TaroとしてMorning Gloryのサイトにログインし、マイアカウントページを開き、適当な画像ファイルをプロフィール画像として登録してみましょう。

RESULT アップロードされたプロフィール画像

では、_body.html.erbの変更点を見ていきましょう。

5行目の@member.profile_picture.attached?は、プロフィール画像が添付されているかどうかをtrueまたはfalseで返します。この部分を単に@member.profile_pictureとだけ書くのは犯しがちな誤りです。クラスメソッドhas_one_attachedで追加された属性は常にActiveStorage::Attached::Oneクラスのインスタンスを返すので、ファイルが添付されていなくても5行目の条件が成立しています。

次に、6行目の<%= %>で囲まれたRubyコードをご覧ください。

```
image_tag @member.profile_picture.variant(resize: "128x128")
```

variantは画像データを変換するメソッドです。このメソッドは添付ファイルのデータが画像である場合にしか使えません。上の例ではオプションresizeにより画像の大きさを調整してい

ます。画像の縦横比を維持しつつ、幅128ピクセル、高さ128ピクセルの範囲で最も大きくなるように画像を拡大・縮小します。

オリジナルの画像サイズで表示したければ、次のようにvariantメソッドを外してください。

```
image_tag @member.profile_picture
```

画像をモノクロ（グレースケール）に変換したければ、typeオプションに"Grayscale"と指定します。

```
image_tag @member.profile_picture
  .variant(resize: "128x128", type: "Grayscale")
```

variantメソッドに指定できるオプションを調べるには

Active Storageはvariantメソッドに指定されたオプションをImageMagickに渡して、変換された結果を受け取っているだけなので、Active Storageのドキュメントを見てもvariantメソッドにどんなオプションを指定できるのかはわかりません。次のURLに指定可能なオプションのリストが掲載されています。

https://www.imagemagick.org/script/convert.php

ただし、上記のページは情報が多すぎるので、目的のオプションをすぐに見つけられないかもしれません。インターネット検索で調べるなら、キーワードとして「imagemagick」と「convert」の2つに「回転」や「切り抜き」などの単語を加えるとよいでしょう。

シードデータ

シードデータを投入するスクリプトを書き換えて、会員「Taro」にはじめからプロフィール写真が登録されるようにしましょう。

まず、PNG形式の適当な画像を用意して、db/seeds/developmentディレクトリにprofile.pngというファイル名で置いてください。サポートサイトで配布されているサンプルソースにも含まれています。

そして、同ディレクトリにあるmembers.rbを次のように書き換えてください。

LIST chapter13/db/seeds/development/members.rb

```
  （省略）
28   password_confirmation: "password"
29 )
30 end
31
32 filename = "profile.png"
33 path = Rails.root.join(__dir__, filename)
34 m = Member.find_by!(number: 10)
35
36 File.open(path) do |f|
37   m.profile_picture.attach(io: f, filename: filename)
38 end
```

そして、データベースを再構築し、ブラウザで会員「Taro」のプロフィール写真が表示されることを確認してください。

```
$ bin/rails db:rebuild
```

では、シードデータ投入スクリプトの中身を見ていきましょう。まず、32〜33行目をご覧ください。

```
filename = "profile.png"
path = Rails.root.join(__dir__, filename)
```

メソッド__dir__は、このスクリプトが置かれているディレクトリの絶対パスを返します。これに"profile.png"を加えて画像ファイルの絶対パスを作り、変数pathにセットしています。

34行目以降で、実際に会員とプロフィール画像とを結び付けています。

```
m = Member.find_by!(number: 10)

File.open(path) do |f|
  m.profile_picture.attach(io: f, filename: filename)
end
```

これは、外部のファイルからデータを読み込んでモデルオブジェクトに添付する際の定石です。フォームから送信されたデータを添付する場合と異なり、attachメソッドにioオプションとfilenameオプションを用いる必要があります。

ファイルのデータ形式に関するバリデーション

会員情報の編集ページやマイアカウントページから「プロフィール画像」として画像ファイルではないファイル、たとえばExcelファイルやPDFファイルをアップロードするとどのような結果になるでしょうか。実際に試してみると、ファイルのアップロードまでは成功しますが、プロフィール画像を表示するところで例外ActiveStorage::InvariableErrorが発生してしまいます。

そこで、Memberモデルのprofile_picture属性にデータ形式に関するバリデーションを加えましょう。まず、準備作業としてMemberクラスの親クラスであるApplicationRecordクラスでALLOWED_CONTENT_TYPESという定数を定義します。この定数の中身は配列で、その各要素はデータ形式を表す文字列（コンテントタイプ）です。この定数はMemberクラスの中で定義してもよいのですが、次節でブログ画像のバリデーションをする際にも使うため親クラスで定義しておきます。

LIST chapter13/app/models/application_record.rb

```
1 class ApplicationRecord < ActiveRecord::Base
2   self.abstract_class = true
3
4   ALLOWED_CONTENT_TYPES = %q{
5     image/jpeg
6     image/png
7     image/gif
8     image/bmp
9   }
10 end
```

次に、Memberモデルのソースコードを次のように書き換えます。

LIST chapter13/app/models/member.rb

```
（省略）
28   validates :password, presence: { if: :current_password }
29
30   validate if: :new_profile_picture do
31     if new_profile_picture.respond_to?(:content_type)
32       unless new_profile_picture.content_type.in?
(ALLOWED_CONTENT_TYPES)
33         errors.add(:new_profile_picture, :invalid_image_type)
34       end
35     else
36       errors.add(:new_profile_picture, :invalid)
37     end
38   end
39
```

```
40  before_save do
    （以下省略）
```

エラーメッセージを日本語で表示するためロケールテキストを追加します。

LIST chapter13/config/locales/ja.yml

```
    （省略）
37 ␣␣␣␣errors:
38 ␣␣␣␣␣␣messages:
39 ␣␣␣␣␣␣␣␣invalid_member_name: "は半角英数字で入力してください。"
40 ␣␣␣␣␣␣␣␣wrong: が正しくありません。
41 ␣␣␣␣␣␣␣␣expired_at_too_old: は掲載開始日より新しい日時にしてください。
42 ␣␣␣␣␣␣␣␣invalid_image_type: にはJPEG、PNG、GIF、PNG形式の画像を指定してください。
```

動作確認をしましょう。いったんサーバーを停止し、bin/rails db:rebuildコマンドでデータベースを作り直してください。改めてサーバーを起動し、適当な会員としてログインしてマイアカウントページから画像でないファイルをプロフィール画像としてアップロードしてみてください。エラーメッセージとして「プロフィール画像にはJPEG、PNG、GIF、PNG形式の画像を指定してください。」と表示されればバリデーションが正常に機能しています。

クラスメソッドvalidate（語末の「s」がないほう）については、Chapter 9で学習しました。このメソッドはブロックを取り、ブロックの中でバリデーションのやり方を定義します。ただし、今回はif: :new_profile_pictureというオプションが付けられています。

```
validate if: :new_profile_picture do
```

これは属性new_profile_pictureがnilでもfalseでもない場合だけ、ブロック内のコードを実行してバリデーションを行う、という意味になります。

次に、ブロックの中身をご覧ください。

```
  if new_profile_picture
    if new_profile_picture.respond_to?(:content_type)
      unless new_profile_picture.content_type.in?
(ALLOWED_CONTENT_TYPES)
        errors.add(:new_profile_picture, :invalid_image_type)
      end
    else
      errors.add(:new_profile_picture, :invalid)
    end
  end
```

respond_to?メソッドは、あるオブジェクトが特定のメソッドを持っているかどうかを調べてtrueまたはfalseを返します。ここでは、属性new_profile_pictureの値がcontent_typeメソッドを持っているかどうかで条件分岐をしています。

このメソッドを持っていない場合は、それがフォームからアップロードされたファイルデータではないことを示しているので、エラーシンボル:invalidをセットします。content_typeメソッドを持っている場合は、メソッドの戻り値が定数ALLOWED_CONTENT_TYPESに含まれていなければ、エラーシンボル:invalid_image_typeをセットします。このシンボルはRails標準のものではなく、開発者自身が決めたasagaoアプリケーション独自のものです。そのため、エラーメッセージをロケールテキストに加えました。

13.3 プロフィール画像の削除

この節では、会員のプロフィール画像を削除する機能を作ります。

添付ファイルの削除

現状の会員情報の編集ページやマイアカウントページには、アップロードされたプロフィール画像を削除する機能がありません。これを作るにはどうすればよいでしょうか。

モデルオブジェクトに添付されたファイルを削除するには、purgeメソッドを使います。実際に操作しながら、その利用法を把握することにしましょう。bin/rails cコマンドでRailsコンソールを起動してください。MSYS2/MinGW環境の方はコマンドの前にwinpty rubyを付けます。そして、次の3つの式を入力してみましょう（式を評価した結果の表示は省略しています）。

```
$ bin/rails c
irb(main):001:0> m = Member.find_by(name: "Taro")
irb(main):002:0> m.profile_picture.attached?
irb(main):003:0> m.profile_picture.purge
```

前節で会員Taroのプロフィール画像を登録したので2番目の式ではtrueという結果が返ります。ブラウザに戻って会員Taroのマイアカウントページを開き、プロフィール画像が削除されていることを確認してください。

モデルオブジェクトにファイルが添付されていない場合、purgeメソッドは何の効果もありません。つまり、エラーにはなりません。Railsコンソールに3番目の式を再度入力して確かめてみましょう。

以上の知識を踏まえ、モデルMemberのソースコードを次のように変更してください。

LIST chapter13/app/models/member.rb

```
（省略）
5  has_one_attached :profile_picture
6  attribute :new_profile_picture
7  attribute :remove_profile_picture, :boolean
8
9  validates :number, presence: true,
（省略）
41  before_save do
42    if new_profile_picture
43      self.profile_picture = new_profile_picture
44    elsif remove_profile_picture
45      self.profile_picture.purge
46    end
47  end
（以下省略）
```

前節で紹介したクラスメソッドattributeを用いて、「プロフィール画像を削除するかどうか」を表す属性remove_profile_pictureをMemberモデルに加えています。ただし、属性new_profile_pictureの場合とは異なり、第2引数にシンボル:booleanを指定しています。これにより、属性remove_profile_pictureの値は適切に型変換（casting）が行われるようになります。フォームから送られてくるパラメータの値はすべて文字列です。チェックボックスの場合、チェックされていれば"1"、チェックされていなければ"0"が送られてきます。これらをtrueとfalseに変換したいのです。

before_saveコールバックの中では、属性remove_profile_pictureが真であればプロフィール画像を削除しています。ただし、属性new_profile_pictureに画像データがセットされている場合は、プロフィール画像の削除は行いません。

チェックボックスの設置

データベーステーブルmembersにremove_profile_pictureというカラムはありませんが、フォームビルダーを用いて属性remove_profile_pictureのための部品を作ることができます。会員情報の編集ページとマイアカウントページで共用している部分テンプレート_member_form.html.erbを次のように書き換えてください。

LIST chapter13/app/views/shared/_member_form.html.erb

```
（省略）
    <th><%= form.label :new_profile_picture %></th>
    <td>
      <div><%= form.file_field :new_profile_picture %></div>
      <% if @member.profile_picture.attached? %>
        <div>
          <%= image_tag @member.profile_picture.variant(resize:
"128x128") %>
          <%= form.check_box :remove_profile_picture %>
          <%= form.label :remove_profile_picture %>
        </div>
      <% end %>
    </td>
  </tr>
（以下省略）
```

チェックボックスの右に表示するラベルをロケールテキストに追加します。

LIST chapter13/config/locales/ja.yml

```
（省略）
␣␣␣␣attributes:
```

```
 8 ␣␣␣␣␣␣member:
 9 ␣␣␣␣␣␣␣␣new_profile_picture: プロフィール画像
10 ␣␣␣␣␣␣␣␣remove_profile_picture: 画像を削除
11 ␣␣␣␣␣␣␣␣number: 背番号
   （以下省略）
```

ストロング・パラメータで属性remove_profile_pictureが許可されるようにAccountsControllerを変更します。

LIST chapter13/app/controllers/accounts_controller.rb

```
   （省略）
22  # ストロング・パラメータ
23  private def account_params
24    params.require(:account).permit(
25      :new_profile_picture,
26      :remove_profile_picture,
27      :number,
   （以下省略）
```

同様に、MembersControllerを変更します。

LIST chapter13/app/controllers/members_controller.rb

```
   （省略）
61 # ストロング・パラメータ
62 private def member_params
63   attrs = [
64     :new_profile_picture,
65     :remove_profile_picture,
```

```
66      :number,
    （以下省略）
```

では、動作確認をしましょう。会員Taroとしてログインし、マイアカウントページからプロフィール画像をアップロードし、再びマイアカウントページを開くと次のような画面になります。

RESULT プロフィール画像の削除

チェックボックス［画像を削除］をオンにして［更新する］ボタンをクリックすると、Taroのプロフィール画像が削除されます。同様に会員名簿のほうでも会員のプロフィール画像の登録・削除を試してください。

13.4 ブログ画像のアップロードと表示

この節では、Morning Gloryのブログ記事に対して複数の画像をアップロードする機能を作ります。

EntryImageクラスの作成

ここまでは会員に対して1個のプロフィール画像を添付する機能を作ってきました。この節では、ブログ記事（Entryオブジェクト）に対して複数の画像を結び付けたいと思います。

Active Storageにはモデルと複数個のファイルを結び付けるためのクラスメソッドhas_many_attachedが用意されていますが、このクラスメソッドで定義された属性には大きな制約があります。それは、ファイル群の一部を差し替えたり、順番を入れ替えたりできない、ということです。クラスメソッドhas_many_attachedの用途は非常に限られるので、本書では説明を省略します。

クラスメソッドhas_many_attachedが使えないとなるとどうすればよいでしょうか。単純な答えがあります。個々のブログ画像を添付するためのモデルクラスEntryImageを作ればよいのです。

ターミナルで次のコマンドを実行してください。

```
$ bin/rails g model entry_image
```

そして、生成されたマイグレーションスクリプトを次のように書き換えます。

LIST chapter13/db/migrate/20180601123044_create_entry_images.rb

```
1 class CreateEntryImages < ActiveRecord::Migration[5.2]
2   def change
```

```
  create_table :entry_images do |t|
    t.references :entry                          # 外部キー
    t.string :alt_text, null: false, default: "" # 代替テキスト
    t.integer :position                          # 表示位置

    t.timestamps
  end
end
```

文字列型のカラムalt_textには画像の代替テキストを格納します。整数型のカラムpositionはNOT NULL制約なしで定義します。このカラムの役割については、次節で説明します。

マイグレーションを実行します。

```
$ bin/rails db:migrate
```

続いて、EntryImageモデルのソースコードを次のように書き換えます。

LIST chapter13/app/models/entry_image.rb

```
class EntryImage < ApplicationRecord
  belongs_to :entry
  has_one_attached :data

  attribute :new_data

  validates :new_data, presence: { on: :create }

  validate if: :new_data do
```

```
10    if new_data.respond_to?(:content_type)
11     unless new_data.content_type.in?(ALLOWED_CONTENT_TYPES)
12       errors.add(:new_data, :invalid_image_type)
13     end
14    else
15     errors.add(:new_data, :invalid)
16    end
17  end
18
19  before_save do
20    self.data = new_data if new_data
21  end
22 end
```

基本的な考えはMemberモデルの場合と同じです。仮想的な属性new_dataに対してバリデーションを行い、before_saveコールバックで実際に画像データを保存しています。

しかし、違いがいくつかあります。7行目をご覧ください。

```
validates :new_data, presence: { on: :create }
```

新しい画像データが存在しない場合にはエラーとしています。EntryImageモデルはブログ画像を表すので、ファイルが添付されていなければ意味がありません。ただし、保存済みのモデルオブジェクトを更新する場合、新しい画像データはなくてもかまいません。代替テキストのみを変更する場合もあるからです。そのため、presenceオプションのonサブオプションに:createを付けています。これは、モデルオブジェクトを新しく作るときだけバリデーションを行うという意味です。

では、EntryモデルのとEntryImageモデルを1対多で関連付けて、モデル側の実装を終えましょう。

LIST chapter13/app/models/entry.rb

```
1 class Entry < ApplicationRecord
2   belongs_to :author, class_name: "Member", foreign_key: "member_id"
3   has_many :images, class_name: "EntryImage"
4
5   STATUS_VALUES = %w(draft member_only public)
  （以下省略）
```

画像のアップロードと削除

■ルーティングの設定

モデル側の準備が整ったので、コントローラ側の実装に移りましょう。まず、ルーティングの設定をします。

LIST chapter13/config/routes.rb

```
  （省略）
20   resources :articles
21   resources :entries do
22     resources :images, controller: "entry_images"
23   end
24 end
```

ブログ画像は必ず特定のブログ記事と結び付けられているので、このようにブロックを用いてentriesリソースの下の階層としてリソースを定義します。そして、EntryImagesControllerでブログ画像のCRUD操作を行うことにします。ただし、URLパスが冗長にならないようにリソース名はimagesとします。このようにcontrollerオプションを用いると、リソース名とコントローラ名の間を自由に関連付けることができます。

今回のルーティングの変更により、次のパスとHTTPメソッドでアクションを呼び出せるようになります。

ブログ画像操作のルーティング

アクション	パス	HTTPメソッド	パスを返すメソッド
index	/entries/123/images	GET	entry_images_path(entry)
show	/entries/123/images/99	GET	entry_image_path(entry, image)
new	/entries/123/images/99/new	GET	new_entry_image_path(entry)
edit	/entries/123/images/99/edit	GET	edit_entry_image_path(entry, image)
create	/entries/123/images	POST	entry_images_path(entry)
update	/entries/123/images/99	PATCH	entry_image_path(entry, image)
destroy	/entries/123/images/99	DELETE	entry_image_path(entry, image)

■コントローラの実装

続いて、コントローラとビュー（HTMLテンプレート）のソースコードを生成します。

```
$ bin/rails g controller entry_images index new edit
```

EntryImagesControllerのソースコードを次のように書き換えます。

LIST chapter13/app/controllers/entry_images_controller.rb

```
1 class EntryImagesController < ApplicationController
2   before_action :login_required
3
4   before_action do
```

```
5     @entry = current_member.entries.find(params[:entry_id])
6   end
7
8   # 画像一覧
9   def index
10    @images = @entry.images.order(:id)
11  end
12
13  # 編集フォームにリダイレクト
14  def show
15    redirect_to action: "edit"
16  end
17
18  # 新規登録フォーム
19  def new
20    @image = @entry.images.build
21  end
22
23  # 編集フォーム
24  def edit
25    @image = @entry.images.find(params[:id])
26  end
27
28  # 新規作成
29  def create
30    @image = @entry.images.build(image_params)
31    if @image.save
32      redirect_to [@entry, :images], notice: "画像を作成しました。"
33    else
```

```
34      render "new"
35    end
36  end
37
38  # 更新
39  def update
40    @image = @entry.images.find(params[:id])
41   @image.assign_attributes(image_params)
42    if @image.save
43      redirect_to [@entry, :images], notice: "画像を更新しました。"
44    else
45      render "edit"
46    end
47  end
48
49  # 削除
50  def destroy
51    @image = @entry.images.find(params[:id])
52    @image.destroy
53    redirect_to [@entry, :images], notice: "画像を削除しました。"
54  end
55
56  # ストロング・パラメータ
57  private def image_params
58    params.require(:image).permit(
59      :new_data,
60      :alt_text
61    )
```

```
  end
end
```

コントローラ全体の構造と各アクションの基本的な作り方は、MembersControllerやEntriesControllerと同じです。ただし、3つの重要な相違点があります。

1. ブロックを伴うbefore_actionの中でインスタンス変数@entryに値がセットされている。
2. 各アクションの中でインスタンス変数@entryを使って操作の対象となるブログ画像を取得している。
3. showアクションではeditアクションへのリダイレクションが行われている。

4〜6行目ではアクション・コールバックを設定しています。アクション・コールバックの設定方法には、ブロックを使う方法とメソッド名をシンボルで指定する方法があります。Chapter 8では後者の方法でアクセス制御を行いましたが、ここでは前者の方法を採用しています。特定のコントローラだけでbefore_actionコールバックを定義するときには、この書き方を使うと便利です。

EntryImagesControllerではすべてのアクションでインスタンス変数@entryが必要になるので、アクション実行前の準備作業としてインスタンス変数@entryに値をセットしています。

```
@entry = current_member.entries.find(params[:entry_id])
```

ブラウザから/entries/123/images/99のようなURLパスに対してアクセスがあると、Railsはentry_idパラメータに"123"という文字列を、idパラメータに"99"という文字列をセットします。before_actionブロックの内部では、entry_idパラメータの値を利用してブログ記事を取得し、インスタンス変数@entryにセットします。

showアクションでeditアクションへリダイレクションを行っているのは、やや瑣末とも言える話です。Morning GloryではActive Storageのしくみを通じてブログ画像を表示するので、ユーザーが普通に利用している限りshowアクションが呼び出されることはありません。しかし、

showアクションとupdateアクションのURLパスが共通しているため、ユーザーがupdateアクションのURLをコピーしたりお気に入り登録したりすると、showアクションにアクセスが来る可能性があります。その場合は「404 Not Found」のエラーページを表示するよりも、画像の編集ページに遷移させたほうが親切です。

■ブログ記事に添付された画像の一覧

ビュー（HTMLテンプレート）の実装に進みましょう。まずブログ記事のフッターに「画像」リンクを設置します。

LIST chapter13/app/views/entries/_footer.html.erb

```
（省略）
5       <%= menu_link_to "編集", [:edit, entry] %>
6       <%= menu_link_to "画像", [entry, :images] %>
7       <%= menu_link_to "削除", entry, method: :delete,
8           data: { confirm: "本当に削除しますか？" } %>
（以下省略）
```

そして、EntryImagesControllerのindexアクションのテンプレート全体を次の内容で置き換えてください。

LIST chapter13/app/views/entry_images/index.html.erb

```
1 <% @page_title = "ブログ記事の画像" %>
2 <h1><%= @page_title %></h1>
3 <h2><%= @entry.title %></h2>
4
5 <ul class="toolbar">
6   <%= menu_link_to "ブログ記事に戻る", @entry %>
7   <%= menu_link_to "画像の追加", [:new, @entry, :image] %>
8 </ul>
```

```
 9
10 <% if @images.present? %>
11   <table class="list">
12     <thead>
13       <tr>
14         <th>番号</th>
15         <th>画像</th>
16         <th>代替テキスト</th>
17         <th>操作</th>
18       </tr>
19     </thead>
20     <tbody>
21       <% @images.each_with_index do |image, index| %>
22         <tr>
23           <td><%= index + 1 %></td>
24           <td>
25             <%= image_tag image.data.variant(resize: "100x>"),
26               alt: image.alt_text %>
27           </td>
28           <td>
29             <%= image.alt_text %>
30           </td>
31           <td>
32             <div>
33               <%= link_to "編集", edit_entry_image_path(@entry, image)
%> |
34               <%= link_to "削除", entry_image_path(@entry, image),
35                 method: :delete, data: { confirm: "本当に削除しますか？" } %>
36             </div>
```

```
37           </td>
38         </tr>
39       <% end %>
40     </tbody>
41   </table>
42 <% else %>
43   <p>画像がありません。</p>
44 <% end %>
```

ソースコードの7行目をご覧ください。

```
<%= menu_link_to "画像の追加", [:new, @entry, :image] %>
```

ヘルパーメソッドmenu_link_toの第2引数には、シンボルとモデルオブジェクトを要素とする配列が指定されています。この行は次のようにも書くことができます。

```
<%= menu_link_to "画像の追加", new_entry_image_path(@entry) %>
```

この書き方は33行目で使われています。

```
<%= link_to "編集", edit_entry_image_path(@entry, image) %>
```

しかし、この行を次のように書き直すと例外NoMethodErrorが発生します。

```
<%= link_to "編集", [:edit, @entry, image] %>
```

変数imageがEntryImageクラスのインスタンスを参照しているため、Railsはこの配列を次のようなメソッド呼び出しに変換するのですが、そんなメソッドは定義されていません。

edit_entry_entry_image_path(@entry, image)

画像追加と画像編集

次に、画像追加フォームと画像編集フォームのためのテンプレート群を作成していきます。ここまで学習を進めてきた読者には、特に目新しい内容はありません。単に変更手順を説明するだけとします。

newアクションのテンプレート全体を次の内容で置き換えてください。

LIST chapter13/app/views/entry_images/new.html.erb

```
1 <% @page_title = "ブログ記事への画像追加" %>
2 <h1><%= @page_title %></h1>
3 <h2><%= @entry.title %></h2>
4
5 <% url = entry_images_path(@entry, @image) %>
6 <%= form_for @image, as: "image", url: url do |form| %>
7   <%= render "form", form: form %>
8   <div><%= form.submit %></div>
9 <% end %>
```

editアクションのテンプレート全体を次の内容で置き換えてください。

LIST chapter13/app/views/entry_images/edit.html.erb

```
1 <% @page_title = "ブログ記事の画像編集" %>
2 <h1><%= @page_title %></h1>
3 <h2><%= @entry.title %></h2>
4
5 <% url = entry_image_path(@entry, @image) %>
```

```
<%= form_for @image, as: "image", url: url do |form| %>
  <%= render "form", form: form %>
  <div><%= form.submit %></div>
<% end %>
```

フォームの中身を表示するための部分テンプレートを次の内容で作成します。

LIST chapter13/app/views/entry_images/_form.html.erb

```
<%= render "shared/errors", obj: @image %>

<table class="attr">
  <tr>
    <th><%= form.label :new_data %></th>
    <td><%= form.file_field :new_data %></td>
  </tr>
  <tr>
    <th><%= form.label :alt_text %></th>
    <td><%= form.text_field :alt_text, size: 40 %></td>
  </tr>
</table>
```

この部分テンプレートで使用する日本語のラベルをロケールテキストに追加します。

LIST chapter13/config/locales/ja.yml

```
（省略）
␣␣␣␣␣␣␣␣␣␣status_member_only: 会員限定
␣␣␣␣␣␣␣␣␣␣status_public: 公開
␣␣␣␣␣␣␣␣␣␣entry_image:
␣␣␣␣␣␣␣␣␣␣␣␣data: 画像ファイル
```

```
40 ＿＿＿＿＿＿＿＿＿alt_text: 代替テキスト
41 ＿＿＿＿errors:
42 ＿＿＿＿＿＿messages:
    （以下省略）
```

以上で、ブログ画像の追加、編集、削除を行えるようになりました。ブラウザで実際に動作確認をしてください。

RESULT ブログ記事の画像追加フォーム

RESULT ブログ記事に添付された画像一覧

画像の表示

では、一般の訪問者が見るブログ記事のページに画像を表示しましょう。記事の下にただ単に並べるだけではつまらないので、最初の画像だけ本文の上に、残りの画像は本文の下に載せるという仕様を採用します。

まず、ブログ記事を表示するためのHTMLテンプレートを次のように書き換えてください。

LIST chapter13/app/views/entries/show.html.erb

```
1 <% @page_title = @entry.title + " - " + @entry.author.name + "さんのブログ" %>
2 <h1><%= @entry.author.name %>さんのブログ</h1>
3 <h2><%= @entry.title %></h2>
4
```

```
5 <%= the_first_image(@entry) %>
6 <%= simple_format(@entry.body) %>
7 <%= other_images(@entry) %>
8 <%= render "footer", entry: @entry %>
```

そして、app/helpersディレクトリに新規ファイルentries_helper.rbを作成して、次のコードを書き入れてください。

LIST chapter13/app/helpers/entries_helper.rb

```
1 module EntriesHelper
2   def the_first_image(entry)
3     image = entry.images.order(:id)[0]
4
5     render_entry_image(image) if image
6   end
7
8   def other_images(entry)
9     buffer = "".html_safe
10
11    entry.images.order(:id)[1..-1]&.each do |image|
12      buffer << render_entry_image(image)
13    end
14
15    buffer
16  end
17
18  private def render_entry_image(image)
19    content_tag(:div) do
20      image_tag image.data.variant(resize: "530x>"),
```

```
21         alt: image.alt_text,
22         style: "display: block; margin: 0 auto 15px"
23       end
24     end
25 end
```

ブラウザで動作確認をします。

Morning Glory

Taroさん　ログアウト

TOP | ニュース | ブログ | 会員名簿 | 管理ページ

Taroさんのブログ

散歩

今日は土曜日で天気が良かったので、会社の近くにある芝公園を散歩してきました。

公開 | 編集 | 画像 | 削除 | by Taro | 2018/06/02 20:24

最新ニュース

- 練習試合の結果8
- 練習試合の結果7
- 練習試合の結果6
- 練習試合の結果5
- 練習試合の結果4

会員のブログ

- 散歩 by Taro
- 野球観戦9 by Taro
- 野球観戦8 by Hana
- 野球観戦8 by Jiro
- 野球観戦8 by Taro

RESULT 画像付きでブログ記事を表示

ヘルパーメソッドthe_first_imageはブログ記事に添付された最初の画像を表示するHTMLコードを生成します。もうひとつのヘルパーメソッドother_imagesは残りの画像を表示するHTMLコードを生成します。

3行目と11行目をご覧ください。

```
image = entry.images.order(:id)[0]
entry.images.order(:id)[1..-1]&.each do |image|
```

3行目で式entry.images.order(:id)はリレーションオブジェクトを返します。この時点ではデータベースへの問い合わせは行われていません。しかし、続く[0]によってデータベースへの問い合わせが発生し、リレーションオブジェクトの内部にEntryImageオブジェクトの配列が記録されます。そして、変数imageにその配列の1番目の要素がセットされます。

11行目でも同じ式entry.images.order(:id)が呼ばれてリレーションオブジェクトが返ってきます。このリレーションオブジェクトは3行目で返ってきたものを同じであるため、すでに画像オブジェクトの配列を持っています。そこで、データベースへの問い合わせを省略します。配列に対して[1..-1]を呼ぶと、2番目以降の配列の要素全部を返します。ちなみに..は範囲すなわちRangeオブジェクト（「2.3　いろいろなオブジェクト」）を作る記号です。なお、空の配列に対して[1..-1]を呼ぶとnilを返すため、eachメソッド呼び出しの前にぼっち演算子（&.）が必要です。

3行目はimage = entry.images.order(:id).firstと書いても結果は変わりません。しかし、この式はデータベースから1個だけレコードを取得します。そのため、11行目でもう一度データベースへの問い合わせが発生します。つまり、firstの代わりに[0]と書くことで問い合わせを1回減らせるのです。

次に、プライベートメソッドrender_entry_imageの中身をご覧ください。

```
content_tag(:div) do
  image_tag image.data.variant(resize: "530x>"),
    alt: image.alt_text,
```

```
      style: "display: block; margin: 0 auto 15px"
    end
```

content_tagメソッドとimage_tagメソッドを用いて、divタグで囲まれたimageタグを生成しています。variantメソッドのresizeオプションに指定された"530x>"という値は「幅が530ピクセルを超えていたら幅が530ピクセルになるように縮小せよ」という意味になります。つまり、幅の狭い画像は拡大されずにそのまま表示されます。

13.5 表示位置の入れ替え

この節では、ブログ記事に貼り付けられた画像の表示位置を入れ替える機能を作ります。

準備作業

■Gemパッケージacts_as_listの導入

モデルオブジェクトのリストの並び順を維持したり、順番を入れ替えたりするにはGemパッケージacts_as_listを利用すると便利です。Gemfileを次のように書き換えてください。

LIST chapter13/Gemfile

```
（省略）
gem 'jquery-rails'
gem 'acts_as_list'

group :development, :test do
（以下省略）
```

そして、ターミナルで次のコマンドを実行します。

```
$ bundle install
```

■モデルクラスにacts_as_listを導入

モデルクラスにacts_as_listを導入するには、クラス定義の中でクラスメソッドacts_as_listを呼び出します。EntryImageモデルのソースコードを次のように書き換えてください。

LIST chapter13/app/models/entry_image.rb

```
1 class EntryImage < ApplicationRecord
2   belongs_to :entry
3   has_one_attached :data
4   acts_as_list scope: :entry
5
6   attribute :new_data
    （以下省略）
```

このモデルが別のモデルの子にあたっていて、親オブジェクトに属する子オブジェクト集団の中で並び順を維持したい場合には、クラスメソッドbelongs_toに指定されている関連付けのシンボルを、クラスメソッドacts_as_listのscopeオプションに指定します。

ここで、ブログ記事の添付画像をすべて消去するためにデータベースの再構築を行います。

```
$ bin/rails db:rebuild
```

■ルーティングの設定

次に、画像の表示位置を入れ替えるアクションmove_higherとmove_lowerのためにルーティングを設定します。

LIST chapter13/config/routes.rb

```
    （省略）
21  resources :entries do
22    resources :images, controller: "entry_images" do
23      patch :move_higher, :move_lower, on: :member
```

```
  end
end
```

（以下省略）

この2つのアクションはいずれもPATCHメソッドで呼び出します。この設定変更によって、次の2つのURLパターンがRailsによって認識されるようになります。

- /entries/123/images/99/move_higher
- /entries/123/images/99/move_lower

そして、これらのパターンに沿ったURLパスを生成するメソッド呼び出しは、次のようなものになります。

- move_higher_entry_image(entry, image)
- move_lower_entry_image(entry, image)

機能の実装

■表示位置を入れ替えるアクションの実装

コントローラ側の実装に進みます。まず、EntryImagesControllerのindexアクションを次のように変更してください。Gemパッケージacts_as_listは、オブジェクトリストの並び順を制御するのにpositionカラムを使用します。

LIST chapter13/app/controllers/entry_images_controller.rb

```
（省略）
def index
  @images = @entry.images.order(:position)
end
（以下省略）
```

そして、アクションmove_higherとmove_lowerを追加します。

LIST chapter13/app/controllers/entry_images_controller.rb

```
（省略）
56  # 表示位置を上げる
57  def move_higher
58    @image = @entry.images.find(params[:id])
59    @image.move_higher
60    redirect_back fallback_location: [@entry, :images]
61  end
62
63  # 表示位置を下げる
64  def move_lower
65    @image = @entry.images.find(params[:id])
66    @image.move_lower
67    redirect_back fallback_location: [@entry, :images]
68  end
（以下省略）
```

Gemパッケージacts_as_listはモデルオブジェクトにmove_higherメソッドとmove_lowerメソッドを加えます。前者が上げるための、後者が下げるためのメソッドです。

60行目と67行目では初登場のredirect_backメソッドが使われています。このメソッドは、アクションの呼び出し元のURLパスにリダイレクションを行います。呼び出し元はHTTPヘッダーに書かれている「リファラ」によって判定します。fallback_locationオプションには、リファラが書かれていない場合のリダイレクション先を指定します。このオプションは省略できません。

■ヘルパーメソッドの変更

続いて、前節で作成した2つのヘルパーメソッドthe_first_imageおよびother_imageを変更します。

LIST chapter13/app/helpers/entries.rb

```
（省略）
 3    image = entry.images.order(:position)[0]
（省略）
11    entry.images.order(:position)[1..-1]&.each do |image|
（以下省略）
```

■HTMLテンプレートの変更

最後に、ブログ画像のリストページに画像の表示位置を変更するアクションへのリンクを設置します。

LIST chapter13/app/views/entry_images/index.html.erb

```
（省略）
31          <td>
32            <div>
33              <%= link_to "編集", edit_entry_image_path(@entry, image)
%> |
34              <%= link_to "削除", entry_image_path(@entry, image),
35                method: :delete, data: { confirm: "本当に削除しますか？" } %>
36            </div>
37            <div>
38              <%= link_to_unless image.first?, "上へ",
39                move_higher_entry_image_path(@entry, image),
40                method: :patch %> |
41              <%= link_to_unless image.last?,  "下へ",
```

```
42              move_lower_entry_image_path(@entry, image),
43              method: :patch %>
44            </div>
45          </td>
    （以下省略）
```

ヘルパーメソッドlink_to_unlessは、第1引数の値が真であれば残りの引数をlink_toメソッドに渡してaタグを生成します。もし第1引数の値が偽であれば第2引数の文字列をそのまま返します。

モデルオブジェクトのfirst?メソッドはそのオブジェクトがリストの先頭（最も上）にあればtrueを返し、そうでなければfalseを返します。逆に、last?メソッドはそのオブジェクトがリストの末尾（最も下）にあるかどうかを調べて返します。この2つのメソッドはGemパッケージacts_as_listによって追加されます。

以上で、ブログ記事に貼り付けられた画像の表示位置を入れ替える機能が完成しました。ブラウザで動作確認をしてください。

RESULT ブログ画像の表示順を変えるリンクを設置

13.6 クラウドストレージサービスの利用

この節では、インターネット上のクラウドストレージサービス（Amazon S3、Google Cloud Storage、Microsoft Azure Storage等）と連携してActive Storageを利用する方法について概要を説明します。これらのサービスを利用しない方は、読み飛ばしてもかまいません。

CA証明書の設置

全サービスに共通する準備作業として、SSL接続で使用するCA証明書を設置します。この手順を省略するとブラウザからファイルをアップロードしたときに、例外Faraday::SSLErrorが発生することがあります。ターミナルで次の2つのコマンドを実行してください。ただし、MSYS2/MinGWでは2番目のコマンドの先頭にsudoを付けないでください。

```
$ CERT_PATH=$(ruby -ropenssl -e "puts
OpenSSL::X509::DEFAULT_CERT_FILE")
$ sudo curl "https://curl.haxx.se/ca/cacert.pem" -o $CERT_PATH
```

Amazon S3

Amazon S3の概要

Amazon Simple Storage Service（Amazon S3）は、クラウドコンピューティングサービスAmazon Web Services（AWS）が提供するストレージサービスです。RailsのActive Storageは、Amazon S3に対してファイルをアップロードすることができます。

AWSやAmazon S3の詳しい利用法を解説することは本書の範囲を超えますが、以下に要点をまとめます。

■AWSアカウントの習得

まず（まだ取得していない方は）AWSのアカウントを取得してください。原則として、クレジットカードの登録が必要となります。アカウント作成から12か月間はAmazon S3の無料利用枠が使えます（最大5GB）。

■IAMユーザーの作成

アカウントを取得したら、AWSコンソールでIAMユーザーを作成してください。IAMとはIdentity and Access Managementの略語です。用途別に専用のIAMユーザーを作ります。たとえば、本書の学習のためにasagaoという名前のIAMユーザーを作成します。

このIAMユーザーに対して、アクセスキーを作成します。アクセスキーとはアクセスキーIDとシークレットアクセスキーを組み合わせたものです。この2つの情報を用いてAmazon S3にアクセスします。

また、このIAMユーザーに対してAmazonS3FullAccessという名前のポリシーを付与（アタッチ）します。AWS用語の「ポリシー」とはアクセス権限の組み合わせに名前を付けたものです。

■バケットの作成

続いて、Amazon S3のバケットを作成します。AWS用語の「バケット」とは、ファイルなどのオブジェクトを格納するための入れ物です。バケットには名前を付けます。ドメイン名と同様の命名規則に従った名前を付ける必要があります（例: asagao.foo.bar）。また、S3全体で一意でなければなりません。

また、バケットには「リージョン」という属性があります。S3のサービスが提供されている場所を意味します。AWSコンソールのバケットリストには「アジア・パシフィック（東京）」のようなリージョン名が表示されていますが、Active Storageの資格情報に使用するのは「ap-

northeast-1」のような文字列です。以下、この文字列を「リージョン」と呼びます。リージョン名からリージョンを調べるには、次のページを参照してください。

https://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region

なお、バケット自体のアクセス制限に関する設定はデフォルトのままで支障ありません。

■準備作業

Gemfileを次のように変更してから、bundleコマンドを実行してください。

LIST chapter13-aws/Gemfile

```
（省略）
40 gem 'acts_as_list'
41 gem 'aws-sdk-s3', require: false
42
43 group :development, :test do
（以下省略）
```

Chapter 12で説明したbin/rails credentials:editコマンドによりテキストエディタを開き、資格情報を次のように編集します。

```
aws:
  access_key_id: XXXX...
  secret_access_key: YYYY...
(以下省略)
```

ただし、XXXX...の部分にはアクセスキーIDを、YYYY...の部分にはシークレットアクセスキーを指定してください。テキストエディタを終了すると、暗号化された資格情報がcredentials.yml.encに書き込まれます。

続いて、テキストエディタでconfig/storage.ymlを開き、次のように編集します。

LIST chapter13-aws/config/storage.yml

```
（省略）
10 amazon:
11 ␣␣service: S3
12 ␣␣access_key_id: <%= Rails.application.credentials.dig(:aws,
:access_key_id) %>
13 ␣␣secret_access_key: <%= Rails.application.credentials.dig(:aws,
:secret_access_key) %>
14 ␣␣region: XXXX...
15 ␣␣bucket: YYYY...
（以下省略）
```

ただし、XXXX...の部分にはAmazon S3のリージョンを、YYYY...の部分にはバケット名を指定してください。

さらに、テキストエディタでconfig/environments/development.rbを開き、次のように編集します。

LIST chapter13-aws/config/environments/development.rb

```
（省略）
31  config.active_storage.service = :amazon
（以下省略）
```

■動作確認

データベースをリセットしてから、サーバーを起動します。

```
$ bin/rails db:rebuild
$ bin/rails s
```

Taroユーザーでasagaoにログインし、適当な会員のプロフィール画像をアップロードします。正常にアップロードできたら、念のためAWSのコンソールでAmazon S3バケットの中身を開き、画像ファイルが登録されていることを確認してください。

Amazon S3へのアップロードに失敗した場合は、ブラウザに表示される例外とエラーメッセージを読んで原因を探ってください。次におもな例外と考えられる原因を列挙します。

- Aws::S3::Errors::InvalidAccessKeyId：アクセスキーIDが正しくない。
- Aws::S3::Errors::SignatureDoesNotMatch：シークレットアクセスキーが正しくない。
- Aws::S3::Errors::PermanentRedirect：リージョンが正しくない。
- Aws::S3::Errors::NoSuchBucket：バケット名が正しくない。
- Aws::S3::Errors::AccessDenied：IAMユーザーにAmazonS3FullAccessポリシーが付与されていない。

 Google Cloud Storage

■Google Cloud Storageの概要

Google Cloud Storage（GCS）とは、クラウドコンピューティングサービスGoogle Cloud Platform（GCP）が提供するストレージサービスです。RailsのActive Storageは、GCSに対してファイルをアップロードすることができます。

GCP/GCSの詳しい利用法を解説することは本書の範囲を超えますが、以下に要点をまとめます。

■GCPアカウントの作成

まず（まだ取得していない方は）GCPのアカウントを取得してください。GCSには期間無制限の無料枠（最大5GB）があり、本書の学習のために使用する程度であれば費用はかかりません。ただし、無料枠を利用するだけでもクレジットカードの登録が必要となります。

■プロジェクトの作成

GCPのアカウントができたら、まず専用のプロジェクトを作成します。プロジェクト名にはasagao-railsのような名前を指定してください。使用できる文字は、小文字の英字、数字、ハイフン、スペース、感嘆符です。

■IAMサービスアカウントの作成

次に「IAMと管理」からIAMサービスアカウントを作成します。IAMとはIdentity and Access Managementの略語です。サービスアカウント名は「基礎Ruby on Rails学習」のように自由に付けてかまいません。サービスアカウントの役割には「ストレージ管理者」を選択します。サービスアカウントIDはデフォルト値のままで問題ありません。

チェックボックス［新しい秘密鍵の提供］にチェックを入れてください。キーのタイプはデフォルトの「JSON」を選びます。

［作成］ボタンをクリックするとJSON形式のファイルのダウンロードが始まりますので、asagaoのconfigディレクトリの下にgcs.jsonという名前で保存してください。GCSではこのJSONファイルが資格情報となります。

■バケットの作成

続いて、バケットを作成します。GCP用語の「バケット」とは、ファイルなどのオブジェクトを格納するための入れ物です。バケットの名前にはasagao-foo-barのような文字列を指定します。使用できる文字は、小文字の英字、数字、ハイフン、アンダースコア（_）です。ただし、先頭と末尾には文字または数字を使用してください。ドメインの管理者であれば、storage.example.jpのようなドメイン名をバケット名として使用できます。

ストレージクラスと場所（リージョン）は目的に合わせて選択します。リージョンのリストは次のページで調べることができます。

https://cloud.google.com/compute/docs/regions-zones/regions-zones

ただし、GCSを無料枠内で利用したい場合は、ストレージクラスは「Regional」を選び、場所（リージョン）は「us-」で始まる選択肢から選んでください。その他の組み合わせでは費

用が発生する可能性があります。

■セットアップ

Gemfileを次のように変更してから、bundleコマンドを実行してください。

LIST chapter13-gcp/Gemfile

```
（省略）
40 gem 'acts_as_list'
41 gem 'google-cloud-storage', '~> 1.8', require: false
42
43 group :development, :test do
（以下省略）
```

続いて、テキストエディタでconfig/storage.ymlを開き、次のように編集します。

LIST chapter13-gcp/config/storage.yml

```
（省略）
17 # Remember not to checkin your GCS keyfile to a repository
18 google:
19 ␣␣service: GCS
20 ␣␣project: XXXX...
21 ␣␣credentials: <%= Rails.root.join("config/gcs.json") %>
22 ␣␣bucket: YYYY...
（以下省略）
```

ただし、XXXX...の部分にはプロジェクト名を、YYYY...の部分にはGCSのバケット名を指定してください。

さらに、テキストエディタでconfig/environments/development.rbを開き、次のように編集します。

LIST chapter13-gcp/config/environments/development.rb

```
（省略）
31  config.active_storage.service = :google
（以下省略）
```

資格情報を含むJSONファイルの扱いに注意

configディレクトリに保存したファイルgcs.jsonは暗号化されていない資格情報を含んでいます。そのため、第三者の手に渡らないように注意しなければなりません。

もしあなたがGitを用いてソースコードのバージョン管理をしているのなら、.gitignoreに/config/gcs.jsonという行を加え、このファイルをGitの管理から除外してください。

■動作確認

データベースをリセットしてから、サーバーを起動します。

```
$ bin/rails db:rebuild
$ bin/rails s
```

Taroユーザーでasagaoにログインし、適当な会員のプロフィール画像をアップロードします。正常にアップロードできたら、念のためGCPのコンソールでGCSバケットの中身を開き、画像ファイルが登録されていることを確認してください。

GCSへのアップロードに失敗した場合は、ブラウザに表示される例外とエラーメッセージを読んで原因を探ってください。次におもな例外と考えられる原因を列挙します。

- Google::Cloud::PermissionDeniedError：IAMサービスアカウントの役割に「ストレージ管理者」が選択されていない。
- OpenSSL::PKey::RSAError：gcs.jsonに含まれるプライベートキーが正しくない。

なお、configディレクトリにgcs.jsonが設置されていない場合は、例外RuntimeErrorが発生して、「The keyfile '/home/oiax/rails/asagao/config/gcs.json' is not a valid file.」のようなエラーメッセージが表示されます。また、config/storage.ymlに記載されたバケット名が正しくない場合には、例外NoMethodErrorが発生して「undefined method `create_file' for nil:NilClass」という（わかりにくい）エラーメッセージが表示されます。

 Microsoft Azure Storage

■Microsoft Azure Storageの概要

Microsoft Azure Storage（以下、Azure Storageと呼びます）とは、クラウドコンピューティングサービスMicrosoft Azureが提供するストレージサービスです。RailsのActive Storageは、Azure Storageに対してファイルをアップロードすることができます。

Microsoft AzureやAzure Storageの詳しい利用法を解説することは本書の範囲を超えますが、以下に要点をまとめます。

■Microsoft Azureアカウントの作成

まず（まだ取得していない方は）Microsoft Azureのアカウントを取得してください。Microsoft Azureには無料試用版があり、アカウント取得してから最初の30日間に最大22,500円分のサービスを無償で利用できます。ただし、無料試用版を利用する場合でもクレジットカードの登録が必要となります。

■リソースグループの作成

Microsoft Azureのアカウントができたら、Azureポータル（Azureの管理画面）にログインして専用のリソースグループを作成します。リソースグループ名にはasagao-railsのような名前を指定してください。使用できる文字は、小文字の英字、数字、ハイフンです。

■ストレージアカウントの作成

次に、ストレージアカウントを作成します。ストレージアカウント名は数字または英小文字からなる長さ3～24の文字列です。たとえば、asagao2018のような名前にします。ドット、ハイフン、アンダースコアなどの記号は使えないので注意してください。また、ストレージアカウント名はMicrosoft Azure全体で一意である必要があります。

以下、本書の学習用の推奨設定です。

- **デプロイメントモデル**：Resource Manager
- **アカウントの種類**：BLOBストレージ
- **場所**：「東日本」または「西日本」
- **レプリケーション**：ローカル冗長ストレージ（LRS）
- **パフォーマンス**：Standard
- **アクセス層**：ホット
- **安全な転送が必須**：無効
- **サブスクリプション**：無料試用版
- **リソースグループ**：既存のものを使用
- **ダッシュボードにピン留めする**：チェックを入れる

［作成］ボタンをクリックすると、ストレージアカウントが作成されます。ストレージアカウントの設定画面で「アクセスキー」という項目を選ぶと、キーが2つ表示されます。どちらか一方をコピーして資格情報として使用します。

■コンテナーの作成

続いて、コンテナーを作成します。Azure用語の「コンテナー」とは、ファイルなどのオブジェクトを格納するための入れ物です。コンテナーには名前を付けます。名前に使用できる文字は、英小文字、数字、ハイフンのみです。名前の長さは3文字以降63文字以下でなければなりません。

パブリック・アクセスレベルには「プライベート」を選択してください。［OK］ボタンをクリックするとコンテナーが作成されます。

■セットアップ

Gemfileを次のように変更してから、bundleコマンドを実行してください。

LIST chapter13-azure/Gemfile

```
（省略）
gem 'acts_as_list'
gem 'azure-storage', require: false

group :development, :test do
（以下省略）
```

Chapter 12で説明したbin/rails credentials:editコマンドによりテキストエディタを開き、次のように資格情報を追加します。

```
azure_storage:
  storage_access_key: XXXX...
(以下省略)
```

ただし、XXXX...の部分にはアクセスキーを指定してください。テキストエディタを終了すると、暗号化された資格情報がcredentials.yml.encに書き込まれます。

続いて、テキストエディタでconfig/storage.ymlを開き、次のように編集します。

LIST chapter13-azure/config/storage.yml

```
（省略）
microsoft:
  service: AzureStorage
  storage_account_name: XXXX...
  storage_access_key: <%=
Rails.application.credentials.dig(:azure_storage, :storage_access_key) %>
```

```
29 __container: YYYY...
   （以下省略）
```

ただし、XXXX...の部分にはストレージアカウント名を、YYYY...の部分にはコンテナー名を指定してください。

さらに、テキストエディタでconfig/environments/development.rbを開き、次のように編集します。

LIST chapter13-azure/config/environments/development.rb

```
   （省略）
31   config.active_storage.service = :microsoft
   （以下省略）
```

■動作確認

データベースをリセットしてから、サーバーを起動します。

```
$ bin/rails db:rebuild
$ bin/rails s
```

Taroユーザーでasagaoにログインし、適当な会員のプロフィール画像をアップロードします。正常にアップロードできたら、念のためAzureポータルでコンテナーの中身を開き、画像ファイルが登録されていることを確認してください。

Azure Storageへのアップロードに失敗した場合は、ブラウザに表示される例外とエラーメッセージを読んで原因を探ってください。次におもな例外と考えられる原因を列挙します。

- ActiveStorage::IntegrityError：config/credentials.yml.encに保存されているアクセスキーが正しくない。または、config/storage.ymlのコンテナー名が正しくない。
- Faraday::ConnectionFailed：config/storage.ymlのストレージアカウント名が正しくない。

Chapter 13のまとめ

- Active Storageを利用すれば、Amazon S3、Google Cloud Storage、Microsoft Azure Storageなどのクラウドストレージサービスへ簡単にファイルをアップロードして、配信できます。
- Active Storageはアプリケーションサーバーのローカルディスクにファイルをアップロードするためにも利用できますが、おもに開発・テスト用の機能です。
- 画像処理ソフトウェアImageMagickとGemパッケージmini_magickを導入すれば、Railsアプリケーション上で画像ファイルのリサイズ、回転、グレースケール化などの処理を行えます。
- モデルに添付ファイルの属性を追加するにはクラスメソッドhas_one_attachedを使用します。
- モデルに複数個のファイルを添付するためのクラスメソッドhas_many_attachedもありますが、ファイル群の一部を差し替えたり、順番を入れ替えたりできないという制約があります。
- Gemパッケージacts_as_listを利用すると、簡単にモデルオブジェクトのリストの並び順を維持したり、順番を入れ替えたりできます。

練習問題

［A］　BookモデルにActive Storageで表紙画像を添付するための属性cover_imageを追加します。空欄を埋めてください。

```
class Book < ApplicationRecord
  [          ] :cover_image
end
```

［B］　書籍の表紙画像を表示するHTMLテンプレートを記述します。インスタンス変数@bookに}Bookモデルのインスタンスがセットされているとして、空欄を埋めてください。ただし、画像を幅90ピクセル、高さ114ピクセルにリサイズして表示するものとします。

```
<%= image_tag @book.cover_image.[          ]
%>
```

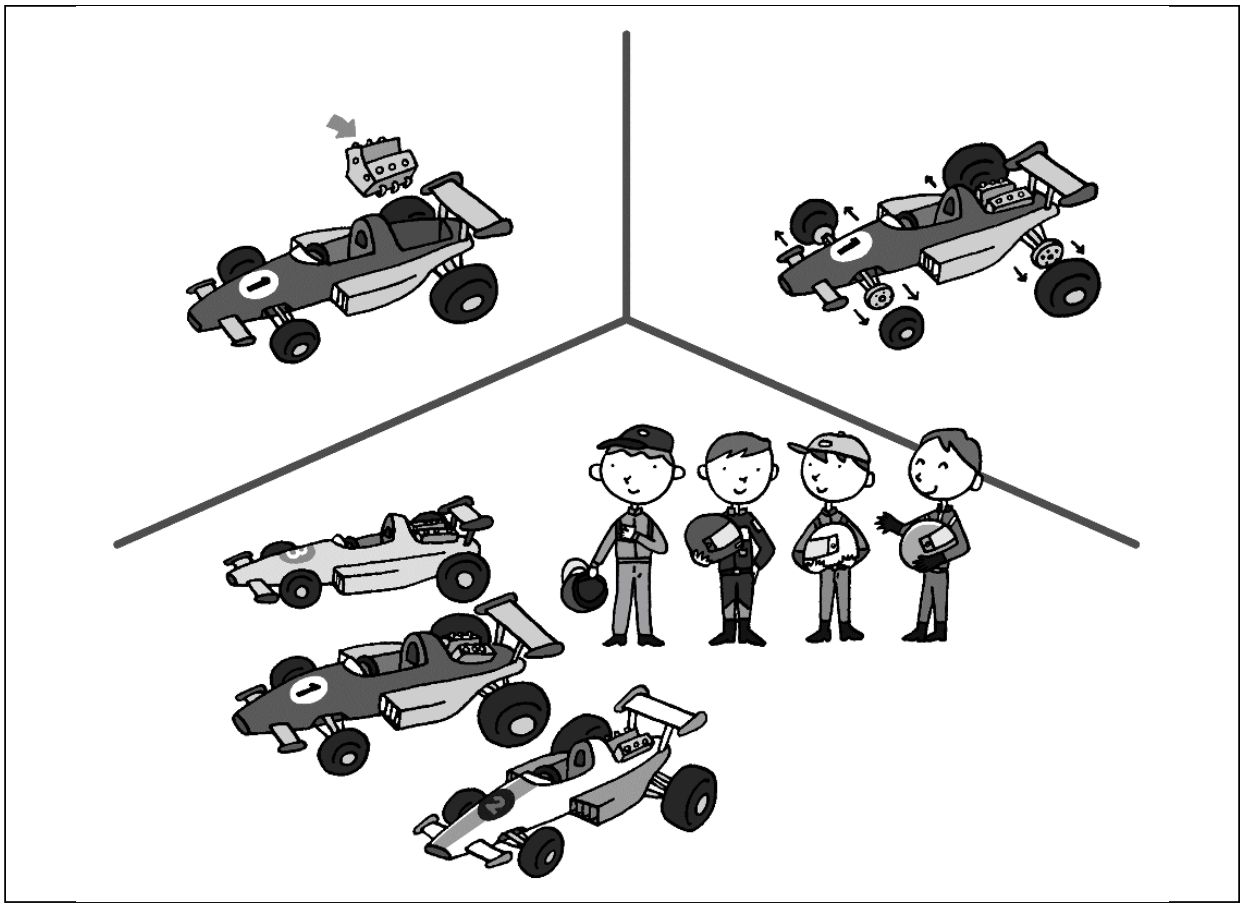
Chapter

14　多対多の関連付け

　Chaper 10では2つのモデル間を1対多で関連付ける方法を学びましたが、それだけでは十分ではありません。モデル同士がもっと自由に結び付くとき、多対多の関連付けを導入する必要があります。内容が少し難しくなってきますが、がんばって乗り切りましょう。

これから学ぶこと

- リレーショナルデータベースで多対多の関連付けを行うときの基本的な考え方を習得します。
- throughオプション付きでクラスメソッドhas_manyを呼び出して2つのモデル間に多対多の関連付けを設定する方法を学びます。
- 多対多の関連付けを使って会員がブログ記事に投票できる機能を作ります。



モデルとモデルを多対多で関連付けるためには、データベース設計上でどんな工夫が必要となるでしょうか？

14.1 多対多の関連付け

2つのモデル間を多対多で関連付けるとはどういう意味でしょうか。1対多の関連付けとどう違うのでしょうか。この節では基本的な考え方を整理しましょう。

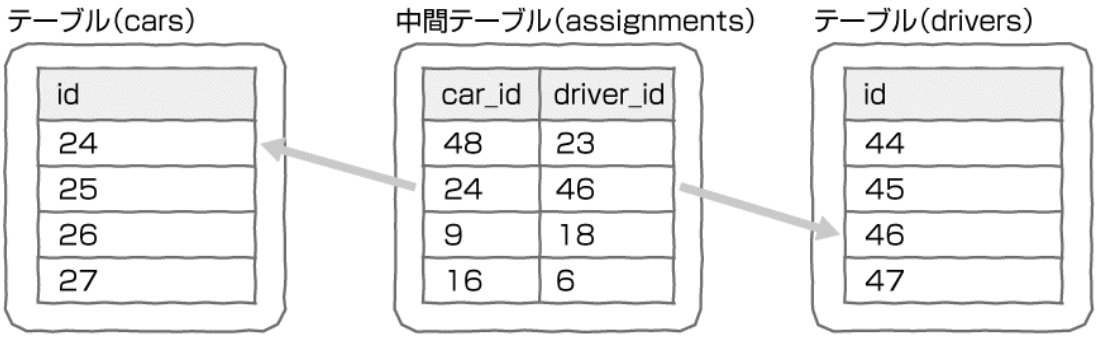
多対多の関連付けとは

Chapter 10で1対多の関連付けについて説明するときに、自動車を例に出しました。1つの自動車には複数個の車輪が付きます。どの車輪がどの自動車に装着されているのかを管理したいのであれば、車輪に自動車のIDを記録することになります。

では、自動車とドライバーについて考えてみましょう。あるレース場に複数の自動車があり、複数のドライバーがいるとします。個々のドライバーはすべての自動車のうちの一部だけを運転できます。誰がどの自動車を運転できるのかをデータベースで管理するには、どのように設計したらいいでしょうか。

素直に考えるとcarsテーブルとdriversテーブルを作り、driversテーブルに自動車のIDの配列を記録すればよさそうです。しかし、リレーショナル・データベースで配列を直接的に扱うのは不可能でないにせよ、あまり効率のよい方法ではありません。このようなケースでは、中間テーブルを別途作り、そのテーブルに自動車とドライバーの結び付きを記録するのが定石です。

中間テーブルの名前は何でもよいのですが、とりあえずassignmentsとしましょう。このテーブルに整数型のcar_idカラムとdriver_idカラムを定義します。前者はcarsテーブルのidカラムを参照し、後者はdriversテーブルのidカラムを参照します。このテーブルにレコードが1つ挿入されれば、それはある特定の自動車と特定のドライバーが結び付けられたことを意味します。



中間テーブルによる多対多の関連付け

多対多の関連付けを設定するメソッド

では、CarモデルとDriverモデルを多対多で関連付けるコードを書いてみましょう。まず、中間テーブルassignmentsに対応するモデルクラスAssignmentを次のように定義します。

```
class Assignment < ApplicationRecord
  belongs_to :car
  belongs_to :driver
end
```

Carモデルの定義は次のようになります。

```
class Car < ApplicationRecord
  has_many :assignments
  has_many :drivers, through: :assignments
end
```

2行目でCarモデルとAssignmentモデルを1対多で関連付けています。注目すべきは3行目のクラスメソッドhas_manyにthroughオプションが付いている点です。すでに設定されている関連付けの名前をこのオプションに指定すると、その関連付けを通じて（through）多対多の関連付けが設定されます。

反対側のDriverモデルの定義もCarモデルとそっくりになります。

```
class Driver < ApplicationRecord
  has_many :assignments
  has_many :cars, through: :assignments
end
```

このようにクラスメソッドhas_manyはthroughオプションの有無で意味が大きく変わります。このため、Railsプログラマはthroughオプション付きのhas_manyメソッドを1つの独立した機能と見なして「ハズメニースルー」と呼んだりします。

 多対多で関連付けられたオブジェクトの集合を操作するメソッド

中間モデルAssignmentを通じてCarモデルとDriverモデルが多対多で関連付けられたことにより、Carクラスのインスタンスメソッドdriversが定義されます。変数carにCarオブジェクトがセットされているとすれば、car.driversで「その自動車を運転できるドライバー」の集合を取り出せます。

ドライバーを作成し、自動車に関連付けるには、次のように記述します。ドライバーのレコードは<<によって自動的に保存されます。saveメソッドを呼ぶ必要はありません。

```
driver = Driver.new
car.drivers << driver
```

逆に、自動車を作成し、ドライバーに関連付けて保存することもできます。

```
car = Car.new
driver.cars << car
```

自動車とトライバーの間の関連付けを外すには、destroyメソッドを使用します。このメソッドはassignmentsテーブルから該当するレコードを削除します。carsテーブルとdriversテーブルには影響を与えません。

```
driver.cars.destroy(car)
```

このChapterでは、多対多の関連付けを設定する演習として会員がブログ記事に投票する機能を作ります。その際、ブログ記事（Entryモデル）と会員情報（Memberモデル）が投票テーブル（Voteモデル）を中間テーブルとして結び付けられることになります。

14.2 ［いいね］ボタンの作成（前編）

この節と次節では、Morning GloryのサイトにFacebookの［いいね］ボタンに似た投票機能を加えます。この節ではその準備作業として、throughオプション付きでhas_manyメソッドを使い会員と記事の間を投票モデルで結び付けます。

会員、記事、投票の関連付け

ここでは投票用のVoteモデルを作成し、MemberモデルとEntryモデルの間に多対多の関連付けを作ります。

■Voteモデルの作成

投票を記録するためにVoteモデル（votesテーブル）を作成し、EntryモデルとMemberモデルの両方に関連付けることにします。「bin/rails g」コマンドを実行しましょう。

```
$ bin/rails g model vote
```

マイグレーションスクリプトを編集し、EntryモデルとMemberモデルを参照する外部キーentry_idとmember_idを作成します。

LIST chapter14/db/migrate/20180602032827_create_votes.rb

```
1 class CreateVotes < ActiveRecord::Migration[5.2]
2   def change
3     create_table :votes do |t|
4       t.references :entry, null: false  # 外部キー
5       t.references :member, null: false # 外部キー
```

```
6
7       t.timestamps null: false
8     end
9   end
10 end
```

マイグレーションを実行してください。

```
$ bin/rails db:migrate
```

■モデル間の関連付け

では、投票機能を実現するためにモデルクラスのコードを変更していきましょう。まず、Voteモデルには、EntryモデルとMemberモデルに対してbelongs_toによる関連付けを設定します。

LIST chapter14/app/models/vote.rb

```
1 class Vote < ApplicationRecord
2   belongs_to :entry
3   belongs_to :member
    （以下省略）
```

次に、EntryモデルからMemberモデルの集合を参照できるような関連付けを作ります。

LIST chapter14/app/models/entry.rb

```
1 class Entry < ApplicationRecord
2   belongs_to :author, class_name: "Member", foreign_key: "member_id"
3   has_many :images, class_name: "EntryImage"
4   has_many :votes, dependent: :destroy
5   has_many :voters, through: :votes, source: :member
```

```
6
7   STATUS_VALUES = %w(draft member_only public)
    （以下省略）
```

5行目で初登場のオプションsourceが使われています。関連付けの名前に対象となるモデルの名前以外のものを使いたい場合にこのオプションを用います。単にhas_many :members, through: :votesとしてしまうと、EntryモデルとMemberモデルの関連性がわかりにくくなります。

続いて、逆方向の関連付けを設定します。

LIST chapter14/app/models/member.rb

```
1 class Member < ApplicationRecord
2   has_secure_password
3
4   has_many :entries, dependent: :destroy
5   has_many :votes, dependent: :destroy
6   has_many :voted_entries, through: :votes, source: :entry
7   has_one_attached :profile_picture
    （以下省略）
```

6行目でオプションsourceを使って関連付けの名前を指定しています。そのままhas_many :entries, through: :votesとしてしまうと、自分のブログ記事を表すentriesと重複してしまうからです。

■投票のルール

投票に関しては、「自分の記事には投票できない」「1つの記事に1回しか投票できない」というルールを作ります。会員が特定の記事に投票できるかどうかを調べられるように、Memberモデルにvotable_for?メソッドを用意しておきます。

ここで使っているexists?は、モデルクラスやリレーションオブジェクトで使えるメソッドで、引数の条件に合うレコードがあるかどうかを調べるものです。

LIST chapter14/app/models/member.rb

```
 1 class Member < ApplicationRecord
   （省略）
51   def votable_for?(entry)
52     entry && entry.author != self && !votes.exists?(entry_id: entry.id)
53   end
54
55   class << self
   （以下省略）
```

Voteモデルにvalidateメソッドを記述して、上記のルールに合わない投票は保存できないようにします。

LIST chapter14/app/models/vote.rb

```
 1 class Vote < ApplicationRecord
 2   belongs_to :entry
 3   belongs_to :member
 4
 5   validate do
 6     unless member && member.votable_for?(entry)
 7       errors.add(:base, :invalid)
 8     end
 9   end
10 end
```

errors.addの引数には、属性名の代わりに:baseを渡しています。特定の属性をエラーとするのではなく、モデルオブジェクト全体にエラーを加えたいときは、:baseを指定します。

■シードデータ

シードデータを修正し、開発用のデータには最初から投票が付いている状態にします。「0.upto(9) do |idx| 〜 end」のループの中で、それぞれ2件の記事に3人の会員が投票したことにします。

LIST chapter14/db/seeds/development/entries.rb

```
（省略）
%w(Taro Jiro Hana).each do |name|
 member = Member.find_by(name: name)
 0.upto(9) do |idx|
  entry = Entry.create(
   author: member,
   title: "野球観戦#{idx}",
   body: body,
   posted_at: 10.days.ago.advance(days: idx),
   status: %w(draft member_only public)[idx % 3]
  )

  if idx == 7 || idx == 8
   %w(John Mike Sophy).each do |name2|
    voter = Member.find_by(name: name2)
    voter.voted_entries << entry
   end
  end
```

```
  end
end
```

シードデータを再投入してください。

```
$ bin/rails db:rebuild
```

14.3 ［いいね］ボタンの作成（後編）

前節で準備したモデル間の関連付け、バリデーション、シードデータを利用して、Morning Glory会員ブログに記事に投票する機能と結果の表示を加えましょう。

ルーティングの設定

投票関連の機能は、EntriesControlerに実装することとし、次のアクションを追加します。

- like：［いいね］ボタンを押したときにvotesテーブルにレコードを作成する。
- unlike：自分の投票を削除する。
- voted：自分が投票した記事の一覧を表示する。

routes.rbを次のように修正します。

LIST chapter14/config/routes.rb

```
（省略）
21  resources :entries do
22    patch "like", "unlike", on: :member
23    get "voted", on: :collection
24    resources :images, controller: "entry_images" do
（以下省略）
```

likeアクションとunlikeアクションは「記事の状態を変更する」ものと見なし、HTTPメソッドをPATCHにします。votedアクションは集合を扱うものなので、onオプションに:collectionを指定します。

投票数とボタンの表示

ブログ記事のフッターには投票数を表示します。フッターの部分テンプレートを修正し、投稿日の右に星と投票数を加えます。

LIST chapter14/app/views/entries/_footer.html.erb

```
（省略）
14   <li>
15     <%= entry.posted_at.strftime("%Y/%m/%d %H:%M") %>
16   </li>
17   <% if (count = entry.votes.count) > 0 %>
18     <li><span class="vote">★<%= count %></span></li>
19   <% end %>
20 </ul>
```

ブラウザでブログ記事一覧のページを開くと、次のように表示されます。

RESULT 投票数の表示

記事の詳細ページの下には投票した名前のリストと［いいね］ボタンを設置することにします。ブログ記事のshowアクションのテンプレートを次のように修正して、部分テンプレート_votes.html.erbを指定します。

LIST chapter14/app/views/entries/show.html.erb

```
（省略）
8 <%= render "footer", entry: @entry %>
9 <%= render "votes" %>
```

app/views/entriesディレクトリの下に_votes.html.erbを作成し、次のように記述します。

LIST chapter14/app/views/entries/_votes.html.erb

```
1 <div class="vote">
2   <% @entry.voters.order("votes.created_at").each do |voter| %>
3     ★<%= voter.name %>
4   <% end %>
5
6   <% if current_member && current_member.votable_for?(@entry) %>
7     <%= link_to "★いいね!", [:like, @entry],
8       method: :patch, class: "button" %>
9   <% end %>
10 </div>
```

2〜4行目では@entry.votersにより多対多で関連付けられた会員（投票者）の配列を取得し、ループの中で会員の名前を表示しています。

6〜9行目では［いいね］ボタンをリンクで作成しています。リンク先はEntriesControllerのlikeアクションで、HTTPメソッドはPATCHとします。votable_for?メソッドでログイン会員が記事に投票できるかチェックし、投票できる場合だけボタンを表示するようにします。

ユーザー名「Taro」、パスワード「asagao!」でログインし、Hanaさんの記事「野球観戦8」を開いてみましょう。次のように表示されます。

RESULT 投票者の表示と［いいね］ボタン

なお、投票ボタン用のCSSは、entries.cssに書かれています。10.3節でコピーし忘れた方は、サンプルソースのsection14-2/app/assets/stylesheetsディレクトリから自分のapp/assets/stylesheetsディレクトリにコピーしてください。

投票した会員の並び順

上記の_votes.html.erbの中では、@entry.votesにクエリーメソッドorder("votes.created_at")を加えています。これは、「votesテーブルのcreated_atの順でソートする」という意味です。votesメソッドを呼ぶと、SQL文の中にentries、votes、membersの各テーブル名が同居することになります。単にorder("created_at")とすると、どのテーブルのcreated_atなのか不明になり、データベースがエラーを出します。そこで、"votes.created_at"としてvotesテーブルのcreated_atでソートすることを指定しています。

likeアクション

EntriesControllerにlikeアクションを加え、投票機能を作りましょう。

LIST chapter14/app/controllers/entries_controller.rb

```
1 class EntriesController < ApplicationController
  （省略）
61  # 投票
62  def like
63    @entry = Entry.published.find(params[:id])
64    current_member.voted_entries << @entry
65    redirect_to @entry, notice: "投票しました。"
66  end
  （以下省略）
```

publishedスコープで下書き以外の記事を取り出し、Memberモデルのvoted_entriesに対して<<で関連付けます。

Hanaさんの記事「野球観戦8」で［いいね］ボタンを押すと、Taroさんの名前が加わり、［いいね］ボタンが消えます。

RESULT 投票結果

14.4 自分が投票した記事一覧

自分の投票の記録の一覧と、投票をあとから削除できる機能も用意しておきましょう。

unlikeアクションとvotedアクション

EntriesControllerにunlikeアクションとvotedアクションを次のように実装します。

LIST chapter14/app/controllers/entries_controller.rb

```
 1 class EntriesController < ApplicationController
   （省略）
68   # 投票削除
69   def unlike
70     current_member.voted_entries.destroy(Entry.find(params[:id]))
71     redirect_to :voted_entries, notice: "削除しました。"
72   end
73
74   # 投票した記事
75   def voted
76     @entries = current_member.voted_entries.published
77       .order("votes.created_at DESC")
78       .page(params[:page]).per(15)
79   end
   （以下省略）
```

unlikeアクションでは、voted_entriesにdestroyメソッドを付け、Entryオブジェクトを渡しています。一見するとこのEntryオブジェクトが削除されるようですが、このオブジェクトとcurrent_memberを結び付けているvotesテーブルのレコードが削除されるだけです。

votedアクションではログイン会員が投票した記事の一覧を取り出します。throughオプション付きのhas_manyメソッドで作ったvoted_entriesに、下書き以外の記事を取り出すpublishedスコープ、クエリーメソッドのorder、それにページネーションのメソッドを加えます。

テンプレートの修正

votedアクションのテンプレートを作成します。

LIST chapter14/app/views/entries/voted.html.erb

```
1 <% @page_title = "投票した記事" %>
2 <h1><%= @page_title %></h1>
3
4 <% if @entries.present? %>
5   <ul>
6     <% @entries.each do |entry| %>
7     <li>
8       <%= link_to entry.title, entry %>
9       by <%= link_to entry.author.name, [entry.author, :entries] %>
10      - <%= link_to "削除", [:unlike, entry], method: :patch,
11            data: { confirm: "削除しますか?" } %>
12    </li>
13    <% end %>
14  </ul>
15  <%= paginate @entries %>
16 <% else %>
```

```
17   <p>記事がありません。</p>
18 <% end %>
```

HTMLのリストで記事の一覧を並べ、ページネーションリンクを加えた簡単なものです。記事と筆者へのリンクの横には、unlikeアクションを呼び出す削除リンクを並べます。

このvotedアクションのページにアクセスできるように、indexアクションのページにリンクを加えます。

LIST chapter14/app/views/entries/index.html.erb

```
   （省略）
 4 <% if current_member %>
 5   <ul class="toolbar">
 6     <%= menu_link_to "ブログ記事の作成", :new_entry %>
 7     <%= menu_link_to "投票した記事", :voted_entries %>
 8   </ul>
 9 <% end %>
   （以下省略）
```

ユーザー名「Sophy」、パスワード「asagao!」でログインし、ページ右上の［ブログ］→［投票した記事］をクリックすると、投票の記録が表示されます。

RESULT 投票した記事一覧

Chapter 14のまとめ

- リレーショナルデータベースで多対多の関連付けを行うときには、2つのテーブルを結び付きを記録するための**中間テーブル**を別途用意します。
- 中間テーブルのためのモデルでは、クラスメソッドbelongs_toを用いて2つのモデルそれぞれと関連付けを行います。
- モデルAとモデルBを多対多で関連付けるときには、まずモデルAと中間テーブルのモデルCを1対多で関連付け、has_manyクラスメソッドを**throughオプション付きで**呼び出します。

練習問題

［A］　書籍（Bookモデル）と利用者（Userモデル）の間を多対多で関連付けます。中間テーブルはlending_recordsとし、モデルクラスLendingRecordで扱います。このとき、これら3つのモデルクラスのソースコードをどのように記述すべきですか。空欄を埋めてください。

```
class Book < ApplicationRecord
  has_many :lending_records
  has_many :users, 
end

class User < ApplicationRecord
  has_many :lending_records
  has_many :books, 
end

class LendingRecord < ApplicationRecord
   :book
   :user
end
```

［B］　前問で行った関連付けを前提として、idカラムの値が5である利用者の持つ書籍のリストにidカラムの値が34である書籍を追加するにはどのように記述すべきですか。空欄を埋めてください。

```
user = User.find(5)
book = Book.find(34)
user.
```

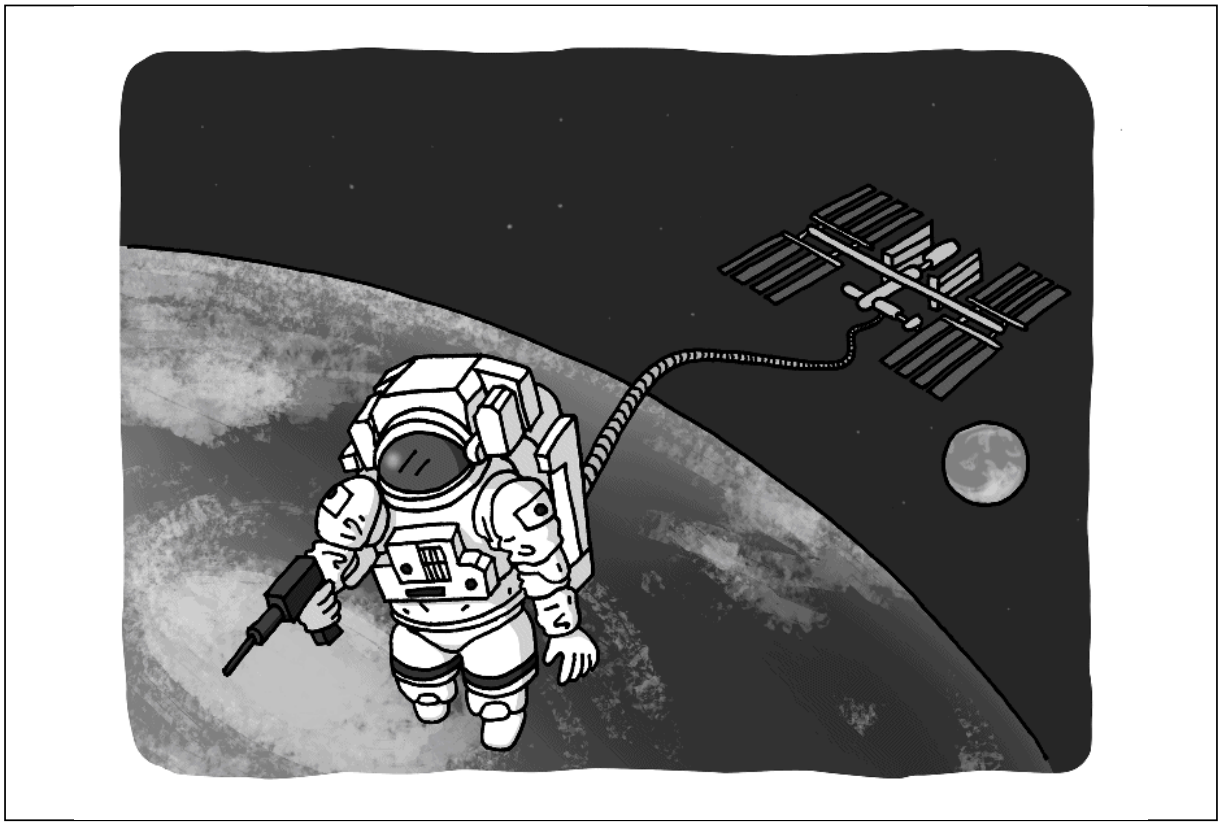
Chapter

15　名前空間

本書を締めくくるこのChapterでは、複雑なウェブアプリケーションを開発する際に避けて通れない「名前空間」という概念について解説します。そして、Morning Gloryサイトに管理ページの機能を作りながらこの概念の具体的な活用法を学んでいきます。

これから学ぶこと

- 名前空間付きのルーティングとコントローラについて学習します。
- 管理ページを作成し、asagaoアプリケーションを完成させます。



最終章ではMorning Gloryサイトに管理ページの機能を追加します。そこで登場する「名前空間」という概念の意味は何でしょうか？ 何のために「名前空間」を導入するのでしょうか？

15.1 名前空間付きのルーティングとコントローラ

ここまでで作ったアプリケーションは、会員情報やニュース記事を誰でも編集できるものでした。管理者用のコントローラを作成して、編集権限を管理者だけに限定する機能を加えましょう。

名前空間を導入する理由

Morning Gloryサイトの利用者は、操作権限の観点から次の3種類に分類できます。

- 訪問者（サイトにログインしていないユーザー）
- 一般会員（サイトにログインしているadministrator属性がfalseのユーザー）
- 管理者（サイトにログインしているadministrator属性がtrueのユーザー）

訪問者は公開されたニュース記事やブログ記事を閲覧できますが、それらの記事を投稿したり、会員名簿を閲覧したりすることはできません。現段階の仕様では、一般会員と管理者の間に大きな違いはありません。ニュース記事に掲載開始日時と掲載終了日時が設定されている場合、一般会員はその範囲の時点でしか記事を読めませんが、管理者はすべての記事を読めます。

なお、管理者には一般会員の持つ操作権限がすべて与えられています。そこで、以下の説明では「一般会員向け」という言葉を「一般会員と管理者向け」という意味で用います。

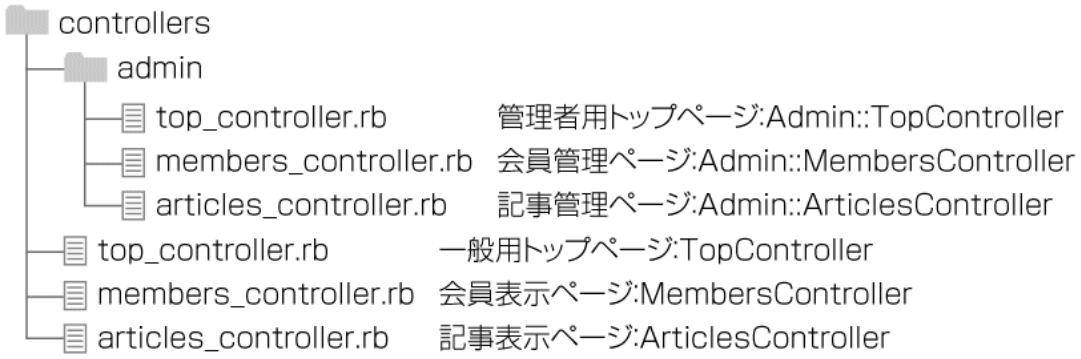
さて、このChapterでは操作権限に関してMorning Gloryサイトの仕様を大きく変えていきたいと思います。具体的には次の2つの仕様変更を行います。

- 管理者のみが会員を追加、編集、削除できる。
- 管理者のみがニュース記事を追加、編集、削除できる。

この種の仕様を導入する際の戦術は2通りです。ひとつは、個々の操作ごとにユーザーがその権限を持つかどうかを判定して、リンクの表示・非表示を切り替えたり、リレーションオブジェクトに検索条件を加えたり、例外Forbiddenを発生させたりするというものです。もうひとつの戦術では、ユーザーの種類ごとに別々のコントローラを用意します。後者の戦略を採用するときに登場するのが**名前空間**（namespace）という概念です。

現時点では、会員の表示、追加、編集、削除を行う機能はMembersControllerクラスで実装されています。私たちは次節で別のコントローラクラスAdmin::MembersControllerを作成します。このクラスの名前は記号::で2つの部分に分かれます。Adminの部分はモジュール名です。このモジュールが付いているため、MembersControllerクラスとAdmin::MembersControllerクラスは別物として区別されます。この状況を「両者は別の名前空間にある」と表現します。

コントローラに名前空間を導入する場合、app/controllersディレクトリの下に名前空間ごとのサブディレクトリを作るのが定石です。Morning Gloryのサイトでは、コントローラのソースファイルを次のように配置することにします。controllersディレクトリ直下のファイルが訪問者を含む全ユーザー向けで、controllers/adminディレクトリの下が管理者向けです。



コントローラの配置

管理TOPページへのルーティング設定

では、手始めに管理者向けのTOPページを作っていきましょう。まずは、ルーティングの設定をします。

LIST chapter15/config/routes.rb

```
  （省略）
30  namespace :admin do
31   root "top#index"
32  end
33 end
```

namespaceメソッドは文字どおりルーティングに名前空間（namespace）を導入します。引数に名前空間の名前をシンボルで指定し、ブロックの内部でその名前空間に属するルーティングを記述していきます。上記の変更の結果として、URLパス/adminからAdmin::TopControllerのindexアクションにルーティングが設定されます。URLパスを生成するメソッドはadmin_root_pathです。ヘルパーメソッドlink_toの第2引数に指定する場合は、次のようにシンボル:admin_rootも使えます。

```
<%= link_to "管理ページ", :admin_root %>
```

あるいは、次のようにシンボルの配列を指定しても同じ意味になります。

```
<%= link_to "管理ページ", [:admin, :root] %>
```

Admin::Baseクラス

すべてのコントローラの親クラスがApplicationControllerであるように、すべての管理者用コントローラの親クラスとなるクラスがあると便利です。app/controllersディレクトリの下にadminディレクトリを作成し、その中にbase.rbというファイルを次の内容で作ってください。

LIST chapter15/app/controllers/admin/base.rb

```
1 class Admin::Base < ApplicationController
2  before_action :admin_login_required
```

```
3
4   private def admin_login_required
5     raise Forbidden unless current_member&.administrator?
6   end
7 end
```

Adminモジュールの下にあるBaseクラスを定義しています。2行目でadmin_login_requiredメソッドをbefore_actionコールバックとして指定しています。この結果、Admin::Baseクラスを継承するすべてコントローラにおいて管理者以外のユーザーによるアクセスが禁止されます。

5行目では、ログインした会員が管理者でないときに例外Forbiddenを発生させています。ログイン前の会員がアクセスした場合はcurrent_memberメソッドがnilを返すので、ぼっち演算子&.を用いてエラーが発生しないようにしています（「5.1　RESTとルーティング」のHINT「&.演算子」を参照）。

Admin::TopController

Admin::Baseクラスを継承するコントローラを作成しましょう。まずは、管理ページのトップページのためのAdmin::TopControllerです。「bin/rails g」コマンドで名前空間付きのコントローラを作成するには、admin/topのように/区切りで名前空間とコントローラ名を指定します。

```
$ bin/rails g controller admin/top index
```

Admin::TopControllerのソースコードを次のように書き換えてください。

LIST chapter15/app/controllers/admin/top_controller.rb

```
1 class Admin::TopController < Admin::Base
2   def index
```

```
  end
end
```

名前空間がAdminであるコントローラ用のテンプレートは、app/views/adminディレクトリの下に配置します。Admin::TopControllerのindexアクションのテンプレートなら、app/views/admin/topディレクトリの下です。内容は、次のような簡単なものにします。

LIST chapter15/app/views/admin/top/index.html.erb

```
<% @page_title = "管理ページトップ" %>
<ul>
  <li><%= link_to "会員管理", "#" %></li>
  <li><%= link_to "ニュース記事管理", "#" %></li>
</ul>
```

現在のページが一般向けなのか管理用なのかが一目でわかるように、メニューバーを切り替えることにします。まず、app/views/sharedディレクトリに新規ファイル_menubar.html.erbを作成します。そして、同じディレクトリの_header.html.erbの11行目以降を切り取って、_menubar.html.erbに貼り付け、次のように書き換えます。

LIST chapter15/app/views/shared/_menubar.html.erb

```
<nav class="menubar">
  <ul>
    <%= menu_link_to "TOP", :root %>
    <%= menu_link_to "ニュース", :articles %>
    <%= menu_link_to "ブログ", :entries %>
    <% if current_member %>
      <%= menu_link_to "会員名簿", :members %>
      <% if current_member.administrator? %>
        <%= menu_link_to "管理ページ", :admin_root %>
      <% end %>
```

```
  <% end %>
 </ul>
</nav>
```

続いて、app/views/sharedディレクトリに次のような内容の新規ファイル_admin_menubar.html.erbを作成します。

LIST chapter15/app/views/shared/_admin_menubar.html.erb

```
<nav class="menubar" id="admin-menubar">
 <ul>
  <%= menu_link_to "管理TOP", :admin_root %>
  <%= menu_link_to "会員管理", "#" %>
  <%= menu_link_to "ニュース記事管理", "#" %>
  <%= menu_link_to "TOP", :root %>
 </ul>
</nav>
```

さらに、_header.html.erbを次のように書き換えてください。

LIST chapter15/app/views/shared/_header.html.erb

```
（省略）
  <%= menu_link_to "ログアウト", :session,
        method: :delete, data: { confirm: "ログアウトしますか？" } %>
 </ul>
<% end %>

<%=
 if controller.kind_of?(Admin::Base)
  render "shared/admin_menubar"
 else
```

```
15     render "shared/menubar"
16   end
17 %>
```

12行目で使っているcontrollerは、コントローラオブジェクトを返すメソッドです。controller.kind_of?(Admin::Base)で、現在のコントローラがAdmin::Baseのインスタンスであるかどうかを調べています。

最後に管理ページ用のスタイルシートを追加します。

LIST chapter15/app/assets/stylesheets/admin.css

```
1 nav#admin-menubar {
2   background-color: #800;
3 }
```

開発用のシードデータでは、ユーザー名が「Taro」の会員を管理者にしています。ユーザー名「Taro」、パスワード「asagao!」でログインして管理ページのトップを見てみましょう。

RESULT 管理ページトップ

また、ユーザー名「Jiro」とパスワード「asagao!」でログインして、ブラウザのアドレスバーに直接http://localhost:3000/adminと書き込んでみて、403エラーのページが表示される点も確認してください。

Adminモジュールは自動的にできる

Admin::Baseクラスの「Admin」は、Adminモジュールです。次のようにAdminモジュールの中にBaseクラスを定義しても同じです。

```
module Admin
  class Base < ApplicationController
  end
end
```

サンプルソースではAdminモジュールをどこにも定義していませんが、Railsがうまくやってくれます。Admin::Baseという名前空間付きのクラスが必要になると、Railsはadminディレクトリの下でbase.rbを探して読み込みます。その際にAdminモジュールがなければ、自動的に作成します。

15.2 会員管理ページの作成

会員情報を扱うMembersControllerの機能をAdmin::MembersControllerに移しましょう。

ルーティング設定

まずルーティングの設定を変更します。一般会員向けのMembersControllerではindexとshow以外のアクションを廃止します。

LIST chapter15/config/routes.rb

```
（省略）
12  resources :members, only: [:index, :show] do
13    get "search", on: :collection
14    resources :entries, only: [:index]
15  end
（以下省略）
```

そして、管理者向けのAdmin::MembersControllerへのルーティングを設定します。7つの基本アクションのほかにsearchアクションが必要となります。

LIST chapter15/config/routes.rb

```
（省略）
30  namespace :admin do
31    root to: "top#index"
32    resources :members do
33      get "search", on: :collection
```

```
34     end
35   end
36 end
```

以上の設定変更により新たなルーティングが次のように設定されます。

名前空間付きのルーティング

アクション	パス	HTTPメソッド	パスを返すメソッド
index	/admin/members	GET	admin_members_path
show	/admin/members/123	GET	admin_members_path(member)
new	/admin/members/new	GET	new_admin_member_path
edit	/admin/members/123/edit	GET	edit_admin_members_path(member)
create	/admin/members	POST	admin_members_path
update	/admin/members/123	PATCH	admin_members_path(member)
destroy	/admin/members/123	DELETE	admin_members_path(member)
search	/admin/members/search	GET	search_admin_members_path

新たなルーティングを利用して、管理者向けメニューバーのHTMLテンプレートを次のように変更し、会員管理ページへのリンクを設定してください。

LIST chapter15/app/views/shared/_admin_menubar.html.erb

```
1 <nav class="menubar" id="admin-menubar">
2   <ul>
3     <%= menu_link_to "管理TOP", :admin_root %>
4     <%= menu_link_to "会員管理", :admin_members %>
   （以下省略）
```

また、管理トップページでも会員管理ページへのリンクを設定してください。

LIST chapter15/app/views/admin/top/index.html.erb

```
1 <% @page_title = "管理ページトップ" %>
2 <ul>
3   <li><%= link_to "会員管理", :admin_members %></li>
4   <li><%= link_to "ニュース記事管理", "#" %></li>
5 </ul>
```

 Admin::MembersControllerの作成

Admin::MembersControllerは、すでに作ってあるMembersControllerをコピー＆ペーストして作るのが簡単です。次の作業を行ってください。

- app/controllersディレクトリの下のmembers_controller.rbをコピーして、app/controllers/adminディレクトリの下に貼り付ける。
- app/views/membersディレクトリをまるごとコピーして、app/views/adminディレクトリの下に貼り付ける。

これらの作業はGUIを持つツール（macOSのFinderやWindowsのエクスプローラー）で行ってもかまいませんが、ターミナルで操作することもできます。慣れれば、そのほうが早いかもしれません。実行するのコマンド群は次のとおりです。

```
$ cd app/controllers
$ cp members_controller.rb admin/
$ cd ../views
$ cp -r members admin/
$ cd ../..
```

adminディレクトリの下にコピーしたmembers_controller.rbを開いて、クラスの定義を次のように変えてください。2行目にあった「before_action :login_required」は必要ないので削除してください。

LIST chapter15/app/controllers/admin/members_controller.rb

```
1 class Admin::MembersController < Admin::Base
2   # 会員一覧
3   def index
    （以下省略）
```

createアクション、updateアクション、およびdestroyアクションの中にあるリダイレクト先を名前空間付きのものに変えます。

LIST chapter15/app/controllers/admin/members_controller.rb

```
    （省略）
35      redirect_to [:admin, @member], notice: "会員を登録しました。"
    （省略）
46      redirect_to [:admin, @member], notice: "会員情報を更新しました。"
    （省略）
56      redirect_to :admin_members, notice: "会員を削除しました。"
    （以下省略）
```

会員管理ページ用のテンプレート修正

app/views/adminディレクトリの下にコピーしたテンプレートを修正していきましょう。まずindexアクションです。

LIST chapter15/app/views/admin/members/index.html.erb

```
1 <% @page_title = "会員管理" %>
2 <h1><%= @page_title %></h1>
3
4 <%= form_tag :search_admin_members, method: :get, class: "search" do
%>
```

```
  5   <%= text_field_tag "q", params[:q] %>
  6   <%= submit_tag "検索" %>
  7 <% end %>
  8
  9 <div class="toolbar"><%= link_to "会員の新規登録",
:new_admin_member %></div>
    （省略）
 22       <% @members.each do |member| %>
 23         <tr>
 24           <td style="text-align: right"><%= member.number %></td>
 25           <td><%= link_to member.name, [:admin, member] %></td>
 26           <td><%= member.full_name %></td>
 27           <td>
 28             <%= link_to "編集", [:edit, :admin, member] %> |
 29             <%= link_to "削除", [:admin, member], method: :delete,
 30                 data: { confirm: "本当に削除しますか?" } %>
    （以下省略）
```

パスの指定を名前空間付きのものに変更しています。変更内容を箇条書きでまとめます。

- :search_members → :search_admin_members
- :new_member → :new_admin_member
- member → [:admin, member] （2箇所）
- [:edit, member] → [:edit, :admin, member]

アクションを表すシンボルが名前空間を表すシンボルよりも前に来る点に気をつけてください。初心者が間違えやすいところです。

showアクションのテンプレートではlink_toの引数を[:edit, @member]から[:edit, :admin, @member]に変えます。

LIST chapter15/app/views/admin/members/show.html.erb

```
（省略）
5 <div class="toolbar"><%= link_to "編集", [:edit, :admin, @member] %>
</div>
（以下省略）
```

一般会員向けMembersControllerのshowアクションのテンプレートでは「編集」リンク自体を削除してください。

LIST chapter15/app/views/members/show.html.erb

```
1 <%  page_title = "会員の詳細" %>
2
3 <h1><%= page_title %></h1>
4
5 <%= render "body" %>
```

newアクションとeditアクションのテンプレートでは、form_forの引数を@memberから[:admin, @member]に変えます。

LIST chapter15/app/views/admin/members/new.html.erb

```
（省略）
5 <%= form_for [:admin, @member] do |form| %>
（以下省略）
```

LIST chapter15/app/views/admin/members/edit.html.erb

```
（省略）
5 <div class="toolbar"><%= link_to "会員の詳細に戻る", [:admin,
@member] %></div>
6
```

```
7 <%= form_for [:admin, @member] do |form| %>
  （以下省略）
```

LIST chapter15/app/views/shared/_member_form.html.erb

```
  （省略）
54  <% if controller.kind_of?(Admin::MembersController) %>
  （以下省略）
```

これで、MembersControllerの機能をAdmin::MembersControllerに移せました。ブラウザで管理ページトップから［会員管理］をクリックして動作確認を行ってください。

RESULT 管理ページの会員一覧

MembersControllerの修正

新しい仕様では会員の編集は管理ページでしかできませんので、一般会員向けのMembersControllerにはindex、show、searchの3つのアクションだけを残します。

MembersControllerとそのテンプレートに対して、次の作業を行ってください。

- app/controllers/members_controller.rbを開き、newアクション以下のアクションをすべて削除する。プライベートメソッドmember_paramsも削除する。
- app/views/membersディレクトリの下から、new.html.erb、edit.html.erbの2つのファイルを削除する。

indexアクションのテンプレートapp/views/members/index.html.erbを開いて、次の行（9行目）を削除してください。

```
<div class="toolbar"><%= link_to "会員の新規登録", :new_member %>
</div>
```

また、会員の一覧表から「操作」の列を削除します。次の箇所を削除してください。修正前のテンプレートの17行目と24～28行目にあたります。

```
  <th>操作</th>
  （中略）
  <td>
    <%= link_to "編集", [:edit, member] %> |
    <%= link_to "削除", member, method: :delete,
          data: { confirm: "本当に削除しますか?" } %>
  </td>
```

ブラウザで一般会員向けの会員一覧を見て、会員管理関連のリンクが表示されなくなっていることを確認してください。

RESULT 一般会員向けの会員一覧

15.3 ニュース記事管理ページの作成

ニュース記事を扱うArticlesControllerの機能をAdmin::ArticlesControllerに移します。考え方は会員管理ページと同じですので、自信のある方は以下の説明を読む前に独力で移してみることをお勧めします。

ルーティングの設定

全ユーザー向けArticlesControllerへのルーティングでは、indexアクションとactionアクションだけを残します。

LIST chapter15/config/routes.rb

```
（省略）
21  resources :articles, only: [:index, :show]
（以下省略）
```

Admin::ArticlesControllerへのルーティングを追加します。

LIST chapter15/config/routes.rb

```
（省略）
30  namespace :admin do
31    root to: "top#index"
32    resources :members do
33      get "search", on: :collection
34    end
35    resources :articles
```

```
36   end
37 end
```

管理者向けメニューバーのHTMLテンプレートを次のように変更します。

LIST chapter15/app/views/shared/_admin_menubar.html.erb

```
1 <nav class="menubar" id="admin-menubar">
2   <ul>
3     <%= menu_link_to "管理TOP", :admin_root %>
4     <%= menu_link_to "会員管理", :admin_members %>
5     <%= menu_link_to "ニュース記事管理", :admin_articles %>
6     <%= menu_link_to "TOP", :root %>
7   </ul>
8 </nav>
```

また、管理トップページにも「ニュース記事管理」へのリンクを設置します。

LIST chapter15/app/views/admin/top/index.html.erb

```
1 <% @page_title = "管理ページトップ" %>
2 <ul>
3   <li><%= link_to "会員管理", :admin_members %></li>
4   <li><%= link_to "ニュース記事管理", :admin_articles %></li>
5 </ul>
```

Admin::ArticlesControllerの作成

ArticlesControllerをコピー＆ペーストしてAdmin::ArticlesControllerを作成しましょう。

- app/controllersディレクトリの下のarticles_controller.rbをコピーして、app/controllers/adminディレクトリの下に貼り付ける。
- app/views/articlesディレクトリをまるごとコピーして、app/views/adminディレクトリの下に貼り付ける。

ターミナルで操作する場合、以下のコマンド群を順に実行してください。

```
$ cd app/controllers
$ cp articles_controller.rb admin/
$ cd ../views
$ cp -r articles admin/
$ cd ../..
```

adminディレクトリの下のarticles_controller.rbを開いて、クラスの定義を変えてください。2行目にあるbefore_actionコールバックの設定も削除します。

LIST chapter15/app/controllers/admin/articles_controller.rb

```
1 class Admin::ArticlesController < Admin::Base
2   # 記事一覧
3   def index
    （以下省略）
```

同ファイルのindexアクションを次のように書き換えます。

LIST chapter15/app/controllers/admin/articles_controller.rb

```
1 class ArticlesController < Admin::Base
2   # 記事一覧
3   def index
4     @articles = Article.order(released_at: :desc)
5       .page(params[:page]).per(5)
```

```
 6  end
（以下省略）
```

同ファイルのshowアクションを次のように書き換えます。

LIST chapter15/app/controllers/admin/articles_controller.rb

```
（省略）
 8  # 記事詳細
 9  def show
10    @article = Article.find(params[:id])
11  end
（以下省略）
```

createアクション、updateアクション、およびdestroyアクションの中にあるリダイレクト先を変更します。

LIST chapter15/app/controllers/admin/articles_controller.rb

```
（省略）
27        redirect_to [:admin, @article], notice: "ニュース記事を登録しました。"
（省略）
38        redirect_to [:admin, @article], notice: "ニュース記事を更新しました。"
（省略）
48      redirect_to :admin_articles
（以下省略）
```

記事管理用のテンプレート修正

app/views/adminディレクトリの下にコピーしたテンプレートを修正します。まずindexアクションです。

LIST chapter15/app/views/admin/articles/index.html.erb

```
 1 <% @page_title = "ニュース一覧" %>
 2 <h1><%= @page_title %></h1>
 3
 4 <div class="toolbar"><%= link_to "新規作成", :new_admin_article %>
</div>
   （省略）
16     <% @articles.each do |article| %>
17       <tr>
18         <td><%= link_to article.title, [:admin, article] %></td>
19         <td><%= article.released_at.strftime("%Y/%m/%d %H:%M") %>
</td>
20         <td>
21           <%= link_to "編集", [:edit, :admin, article] %> }
22           <%= link_to "削除", [:admin, article], method: :delete,
23             data: { confirm: "本当に削除しますか?" } %>
24         </td>
25       </tr>
26     <% end %>
   （以下省略）
```

次にshowアクションのテンプレートを修正します。

LIST chapter15/app/views/admin/articles/show.html.erb

```
 1 <% @page_title = @article.title %>
 2 <h1><%= @article.title %></h1>
 3
 4 <div class="toolbar"><%= link_to "編集", [:edit, :admin, @article] %>
```

```
</div>
    （以下省略）
```

newアクションとeditアクションのテンプレートでは、form_forの引数を@articleから[:admin, @article]に変えます。

LIST chapter15/app/views/admin/articles/new.html.erb

```
    （省略）
 4 <%= form_for [:admin @article] do |form| %>
    （以下省略）
```

LIST chapter15/app/views/admin/articles/edit.html.erb

```
    （省略）
 4 <div class="toolbar"><%= link_to "記事の詳細に戻る", [:admin, @article] %></div>
 5
 6 <%= form_for [:admin, @article] do |form| %>
    （以下省略）
```

これで、ArticlesControllerの機能をAdmin::ArticlesControllerに移せました。ブラウザで確認してみましょう。

RESULT 管理ページの記事一覧

ArticlesControllerの修正

全ユーザー向けのArticlesControllerにはindexアクションとshowアクションだけを残します。次の作業を行ってください。

- app/controllers/articles_controller.rbを開き、newアクション以下のアクションをすべて削除する。プライベートメソッドarticle_paramsも削除する。
- app/views/articlesディレクトリの下から、new.html.erb、edit.html.erb、_form.html.erbの3つのファイルを削除する。

ArticlesControllerでは公開前の記事や期限切れの記事は表示しません。また、会員限定の記事は訪問者の目から隠します。indexアクションとshowアクションを修正して、visibleスコープとopen_to_the_publicスコープで制限します。また、before_actionコールバックの指定を削除します。

LIST chapter15/app/controllers/articles_controller.rb

```
class ArticlesController < ApplicationController
  # 記事一覧
  def index
    @articles = Article.visible.order(released_at: :desc)
    @articles = @articles.open_to_the_public unless current_member
    @articles = @articles.page(params[:page]).per(5)
  end

  # 記事詳細
  def show
    articles = Article.visible
    articles = articles.open_to_the_public unless current_member
    @article = articles.find(params[:id])
  end
end
```

また、全ユーザー向けの記事表示ページでは、ビジュアルデザインを変更して見やすくすることにします。indexアクションのテンプレートを次のものに差し替えます。

LIST chapter15/app/views/articles/index.html.erb

```
<% @page_title = "ニュース一覧" %>
<h1><%= @page_title %></h1>

<% if @articles.present? %>
  <% @articles.each do |article| %>
    <h2><%= article.title %></h2>
    <p>
      <%= truncate(article.body, length: 80) %>
      <%= link_to "もっと読む", article %>
```

```
10     </p>
11     <div class="article-footer">
12       <%= article.released_at.strftime("%Y/%m/%d %H:%M") %>
13     </div>
14   <% end %>
15 <% else %>
16   <p>ニュースがありません。</p>
17 <% end %>
```

showアクションのテンプレートは次のものに差し替えます。

LIST chapter15/app/views/articles/show.html.erb

```
1 <% @page_title = @article.title %>
2 <h1><%= @article.title %></h1>
3
4 <%= simple_format(@article.body) %>
5
6 <div class="article-footer">
7   <%= @article.released_at.strftime("%Y/%m/%d %H:%M") %>
8 </div>
```

app/assets/stylesheetsディレクトリに新規ファイルarticles.cssを作成します。

LIST chapter15/app/assets/stylesheets/articles.css

```
1 div.article-footer {
2   border-top: 1px #ccf dashed;
3   padding-top: 4px;
4   margin-bottom: 8px;
5   text-align: right;
```

```
6   font-size: 75%;
7 }
```

ブラウザで全ユーザー向けのニュース記事を見て、表示を確認してください。

RESULT 全ユーザー向けのニュース記事一覧

RESULT 全ユーザー向けのニュース記事詳細

これでMorning Gloryのサイト、asagaoアプリケーションは完成です。お疲れさまでした！オイアクス社のサポートサイトもご覧になってください。

https://www.oiax.jp/rails5book

Chapter 15のまとめ

- 管理ページを作るには、コントローラをディレクトリ分けして名前空間付きのコントローラを作成します。
- 名前空間付きのリソースを設定するにはnamespaceメソッドを使用します。

練習問題

［A］　次に示すのは、名前空間adminの下にリソースusersを設定するためのconfig/routes.rbです。空欄を埋めてください。

```
Rails.application.routes.draw do
  [          ] :admin do
    resources :users
  end
end
```

［B］　前問のように名前空間付きのリソースを設定したとします。Admin::UsersControllerのshowアクション、showアクション、およびdestroyアクションにリンクするように、link_toメソッドの第2引数を埋めてください。ただし、変数@userにUserクラスのインスタンスがセットされているものとします。

```
<%= link_to "会員の詳細", [          ] %>
<%= link_to "会員の編集", [          ] %>
<%= link_to "会員の削除", [          ],
  method: :delete, data: { confirm: "本当に削除しますか？" }%>
```

Appendix

付録A　参考文献と推薦図書

Ruby

- 『**たのしいRuby 第5版**』高橋征義、後藤裕蔵著、ソフトバンククリエイティブ刊、2016年
  Ruby初心者向けの入門書。本書のChapter 2ではもの足りなかった方にお勧め。
- 『**改訂2版 パーフェクトRuby**』Rubyサポーターズ著、技術評論社刊、2017年
  Rubyの入門書。Gemパッケージの作り方など応用的な内容を含む。
- 『**プロを目指す人のためのRuby入門 言語仕様からテスト駆動開発・デバッグ技法まで (Software Design plusシリーズ)**』伊藤淳一著、技術評論社刊、2017年
  Rubyの基礎知識よりもテスト駆動開発やデバッグのやり方など開発現場で必要になる知識に重点が置かれている。
- 『**プログラミング言語Ruby**』David Flanagan、まつもとゆきひろ著、オライリー・ジャパン刊、2009年
  Rubyの細かい機能まで網羅的に解説した本。Rubyについて深く知りたい方向け。
- 『**まつもとゆきひろ コードの世界**』まつもとゆきひろ著、日経BP刊、2009年

Rubyの開発者であるまつもと氏自身の考えが読める。Rubyに限らずプログラミング全般の話題を扱う。

- 『**JavaからRubyへ――マネージャのための実践移行ガイド**』Bruce A. Tate著、オライリー・ジャパン刊、2007年
  現代的なプログラミングにおけるRubyの利点を解説した本。企業でRubyやRailsを導入したい方向け。
- 『**Effective Ruby**』Peter J. Jones著、翔泳社刊、2015年
  効率的なRubyプログラミングを行うためのノウハウが集まっている。中級者から上級者向け。

Ruby on Rails

- 『**実践Ruby on Rails 4 現場のプロから学ぶ本格Webプログラミング**』黒田努著、インプレスジャパン刊、2014年
  本書の著者による中級者向けのRails学習書。ある企業向けに顧客管理システムを作るという設定で、一連の開発手順を解説したもの。
- 『**Ruby on Rails 5 アプリケーションプログラミング**』山田祥寛著、技術評論社刊、2017年
  Ruby on Railsの入門書。コンポーネント（ビュー、モデル、コントローラ）別の章立てになっており、リファレンスとして使いやすい。
- 『**パーフェクトRuby on Rails**』すがわらまさのり、前島真一、近藤宇智朗、橋立友宏著、技術評論社刊、2014年
  Ruby on Railsの入門書。OAuthによるユーザー認証やChefによるプロビジョニングなど中・上級者向けの内容に特徴がある。

- 『**Ruby on Rails環境構築ガイド**』黒田努著、インプレスジャパン刊、2013年
  Railsアプリケーションの開発環境構築から本番環境への配備（デプロイメント）までを解説した本。
- 『**Agile Web Development with Rails 5.1**』Sam Ruby、Dave Thomas、David Heinemeier Hansson、Pragmatic Bookshelf刊、2018年
  内容は英語で書かれているが、Railsプログラマなら手元に置いておきたい一冊。内容は中級者から上級者向け。

■SQL

- 『**初めてのSQL**』Alan Beaulieu著、オライリー・ジャパン刊、2006年
  SQLの入門書。初心者から中級者向け。言語仕様はMySQLに基づく。

■ウェブアプリケーション

- 『**Webを支える技術**』山本陽平著、技術評論社刊、2010年
  HTTPやURL、RESTについての解説書。ウェブアプリケーション設計の基本思想について学べる。

Appendix
付録B　練習問題の解答

Chapter 1

［A］の解答

- （×）――Rubyは21世紀の初めに登場した比較的新しいプログラミング言語です。
- （○）――Ruby on Railsは、オープンソース方式で開発されているフレームワークです。
- （○）――Ruby on Railsは、Windows、macOS、Linuxなどさまざまなで動作します。
- （○）――Railsをインストールするには、パッケージ・マネージャのRubyGemsを使います。
- （×）――新しいGemパッケージを導入するときは、テキストエディタでGemfile.lockを編集します。

- Rubyは1995年に初めて公開されました。
- Ruby on Railsは、David Heinemeier Hansson氏が中心となって開発しているオープンソース・ソフトウェアです。
- Ruby on Railsは、Rubyが動作する環境ならどこでも動作します。
- RailsはたくさんのGemパッケージで構成されています。
- Gemfile.lockではなくGemfileを編集します。

［B］の解答

- Railsの原則DRYは、「Don't Repeat Yourself」の略で**繰り返しを避けよ**という意味です。
- Railsは**設定より規約**という設計哲学で作られており、規約に従ってアプリケーションを開発することで、記述量を大幅に減らすことができます。

［C］の解答

```
<h1><%= @message %></h1>
<p><%= @description %></p>
```

- テンプレートに変数を埋め込むには、<%= %>の間に変数を入れます。

Chapter 2

［A］の解答

```
print "価格を入力してください："
price = gets.chomp
price = (price.to_i * 1.08).to_i
puts "税込み#{price}円です。"
```

- priceには文字列が入るので、to_iメソッドで数値に変換してから1.08をかけます。さらに、to_iメソッドを使って整数に変換します。

［B］の解答

```
flowers = ["carnation", "tulip", "cosmos"]
flowers.each do |flower|
  puts flower
end
```

- 配列の要素を列挙するにはeachメソッドにブロックを渡します。

［C］の解答

```
class Book
  attr_reader :title, :author, :price
  def initialize(title, author, price)
    @title = title
    @author = author
    @price = price
  end
end

book1 = Book.new("彼岸過迄", "夏目漱石", 540)
puts "#{book1.title}、#{book1.author}著、
#{book1.price}円"
```

- インスタンス変数を属性として取り出すには、attr_readerに変数名を指定します。「attr_accessor :title, :author, :price」としてもOKです。

Chapter 3

［A］の解答

- 記事の投稿など、サーバーの状態を変更するときには、HTTPの**POST**メソッドで送信します。
- redirect_toメソッドを使うと、ブラウザに新しいURLを示して別のページへの**リダイレクション**を行うことができます。
- routes.rbを編集すると、URLのパスから特定のアクションを選ぶ**ルーティング**の設定を変更できます。

［B］の解答

```
<p><%= link_to "このサイトについて", about_path %></p>
```

- リンクを作成するには、link_toメソッドにリンクのテキストとパスを指定します。ルーティングの「as: "about"」の設定で、about_pathが使えるようになります。about_pathの代わりに:aboutと指定してもOKです。

［C］の解答

```
<ul>
  <% @countries.each do |country| %>
    <li><%= country %></li>
  <% end %>
</ul>
```

- テンプレートにeachメソッドによるループを埋め込み、配列の要素を表示します。

Chapter 4

［A］の解答

```
class CreateBooks < ActiveRecord::Migration[5.2]
  def change
    create_table :books do |t|
      t.string :title, null: false  # 書名
      t.string :author, null: false # 著者名
      t.integer :price, null: false # 価格

      t.timestamps
    end
  end
end
```

- 文字列型のコラムを作成するにはt.stringを、整数型のコラムを作成するにはt.integerを使い、カラム名をシンボルで指定します。

［B］の解答

```
book = Book.new
book.title = "明暗"
book.author = "夏目漱石"
book.price = 1200
book.save
```

- レコードのカラム（モデルの属性）に値を入れるには、「オブジェクト.カラム名 = 値」とします。保存するにはsaveメソッドを使います。

［C］の解答

```
book = Book.find(123)
```

- idを指定してモデルオブジェクトを取り出すには、findメソッドを使います。

［D］の解答

```
books = Book.where(author: "夏目漱石")
```

- 条件を指定してモデルオブジェクトを取り出すには、クエリーメソッドのwhereを使います。

［E］の解答

```
books = Book.where("price < ?", 3000)
```

- 比較を使った条件を指定するには、whereメソッドにSQLのWHERE句を指定します。生の値は?を使って埋め込みます。

Chapter 5

［A］の解答

```
index   /books        GET
show    /books/1      GET
new     /books/new    GET
edit    /books/1/edit GET
create  /books        POST
update  /books/1      PATCH
destroy /books/1      DELETE
```

- showアクションのパスは「/リソース名の複数形/id」です。「/books」パスにPOSTメソッドで送信すると、createアクションの呼び出しになります。updateアクションで使われるHTTPメソッドはPATCHです。

［B］の解答

```
class BooksController < ApplicationController
  def index
    @books = Book.order("title")
  end

  def show
    @book = Book.find(params[:id])
  end
end
```

- レコードをソートして取り出すには、クエリーメソッドのorderを使います。idパラメータを元にレコードを1つ取り出すには、findメソッドを使います。

［C］の解答

index.html.erb

```
<ul>
  <% @books.each do |book| %>
    <li><%= link_to book.title, book %></li>
  <% end %>
</ul>
```

show.html.erb

```
<p>書籍名：<%= @book.title %>、
著者名：<%= @book.author %>、
```

```
価格：<%= @book.price %>円</p>
```

- indexアクションのテンプレートでは、link_toメソッドの第2引数にモデルオブジェクトを渡せば、showアクションへのリンクができます。showアクションのテンプレートでは、モデルの属性の値をそれぞれ表示します。

Chapter 6

［A］の解答

```
<%= form_for @book do |form| %>
  <div>書名：<%= form.text_field :title %></div>
  <div>著者：<%= form.text_field :author %></div>
  <div>価格：<%= form.text_field :price %></div>
  <div><%= form.submit %></div>
<% end %>
```

- テキスト入力欄を作るには、フォームビルダーオブジェクトのtext_fieldメソッドを使い、属性名を引数にします。

［B］の解答

```
class BooksController < ApplicationController
  def create
    @book = Book.new(params[:book])
```

```
    @book.save
    redirect_to @book, notice: "作成しました。"
  end

  def update
    @book = Member.find(params[:id])
    @book.assign_attributes(params[:book])
    @book.save
    redirect_to @book, notice: "更新しました。"
  end
end
```

- 送信されたフォームのデータはbookパラメータに入っています。createアクションではnewメソッドにbookパラメータを渡します。updateアクションではassign_attributesメソッドに渡します。

Chapter 7

［A］の解答

```
class Book < ApplicationRecord
  validates :title, :author
  validates :price, presence: true,
```

```
    numericality: { only_integer: true, greater_than: 0 }
end
```

- validatesメソッドに属性名とバリデーションの種類、オプションを指定します。「1以上の整数」は、numericalityで指定します。

［B］の解答

```
ja:
  activerecord:
    models:
      book: 書籍
    attributes:
      book:
        title: 書名
        author: 著者
        price: 価格
```

- book:の下でインデントして、「属性名: テキスト」を並べてください。

Chapter 8

［A］の解答

```
class SessionsController < ApplicationController
  def create
    user = User.find_by(email: params[:email])
    if user&.authenticate(params[:password])
      session[:user_id] = user.id
    else
      flash.alert = "メールアドレスとパスワードが一致しません"
    end
    redirect_to :root
  end
end
```

- セッションデータにユーザーのidカラムの値を保存します。

［B］の解答

```
class ApplicationController < ActionController::Base
  private def current_user
    User.find_by(id: session[:user_id] if session[:user_id]
  end
  helper_method :current_user
end
```

- セッションデータに保存したユーザーのidを元に、find_byメソッドでモデルオブジェクトを取り出します。

［C］の解答

テンプレート

```
<%= link_to "マイアカウント", :account %>
```

- 単数リソースでは、showアクションのパスは「単数形_path」メソッドまたは「単数形のシンボル」で指定します。account_pathとしてもOKです。

Chapter 9

［A］の解答

```
class Book < ApplicationRecord
  after_save do
    Logger.new(Rails.root.join("log/books.log")).info(title)
  end
end
```

- after_saveコールバックはモデルの保存後（新規作成、更新の両方）に呼ばれます。

［B］の解答

```
class Book < ApplicationRecord
  scope :free, -> { where(price: 0) }
```

```
end
```

- クラスメソッドscopeを使ってスコープを定義します。記号->に続くブロックの内側に検索条件を指定する式を記述します。

Chapter 10

［A］の解答

```
class CreateBooks < ActiveRecord::Migration[5.2]
  def change
    create_table :books do |t|
      t.references :shelf, null: false
      t.string :title, null: false
      t.string :author, null: false
      t.integer :price, null: false

      t.timestamps
    end
  end
end
```

- t.references :shelfによってbooksテーブルに整数型のshelf_idカラムが追加されます。

［B］の解答

app/models/shelf.rb

```
class Shelf < ApplicationRecord
  has_many :books
end
```

app/models/book.rb

```
class Book < ApplicationRecord
  belongs_to :shelf
end
```

- Shelfモデルにhas_manyメソッドを記述して、「本棚は書籍をたくさん持つ」という結び付きを作ります。
- Bookモデルにbelongs_toメソッドを記述して、「書籍はある本棚に属する」という結び付きを作ります。

［C］の解答

```
class BooksController < ApplicationController
  def index
    if params[:shelf_id]
      @shelf = Shelf.find(params[:shelf_id])
      @books = @shelf.books.order("title")
    else
```

```
    @books = Book.order("title")
  end
 end
end
```

- 4行目ではパラメータshelf_idの値（1以上の整数）をidカラムの値として持つレコードをshelvesテーブルから取得してオブジェクト化し、変数@shelfにセットしています。
- 5行目ではその@shelfに属する書籍のリストをorderメソッドにより「書籍名（title）」で並べ替えてから変数@booksにセットしています。

Chapter 11

［A］の解答

```
class BooksController < ApplicationController
 （省略）
 def create
  @book = Book.new(book_params)
  if @book.save
   redirect_to @book, notice: "書籍を登録しました。"
  else
   render "new"
  end
```

```
  end
  （省略）
  private def book_params
    params.require(:book).permit(
      :title,
      :author,
      :price
    )
  end
end
```

- フォームから送信されてきたパラメータをストロング・パラメータで処理するためプライベートメソッドbook_paramsからの戻り値をBook.newメソッドの引数に指定します。
- フォームから送信されてきたパラメータに:bookキーが含まれるかどうかをチェックするには、requireメソッドを呼び出します。

［B］の解答

```
class ApplicationController < ActionController::Base
  rescue_from ActiveRecord::RecordNotFound, with: :rescue_404

  private def rescue_404(exception)
    render "errors/not_found", status: 404, layout: "error",
      formats: [:html]
```

```
  end
end
```

- 例外処理の方法を設定するには、クラスメソッドrescue_fromをwithオプション付きで呼び出します。引数には例外クラスを指定します。
- サーバーがブラウザに返すHTTPステータスコードを指定するには、statusオプションにそのコードを指定してrenderメソッドを呼び出します。

Chapter 12

［A］の解答

- （×）——アセット・パイプラインにより、configディレクトリの下にある資格情報が暗号化されます。
- （○）——アセット・パイプラインにより、app/assets/stylesheetsディレクトリの下にあるSass形式のスタイルシートはCSS形式に変換されます。
- （×）——バージョン5.1以降のRuby on RailsにはJavaScriptライブラリjQueryが同梱されています。
- （×）——Turbolinksは、JavaScriptのコンパイル時間を短縮するための機能です。

- アセット・パイプラインは資格情報の暗号化とは無関係です。
- SassはCSSの文法を改良したスタイルシート言語で、アセット・パイプラインにより普通のCSSに変換されます。

- Railsアプリケーションでjqueryを使うにはGemパッケージjquery-railsを導入する必要があります。
- Turbolinksは、ブラウザによるページの遷移を高速化する機能です。

［B］の解答

```
Rails.application.credentials.dig(:remote, :password)
```

- config/credentials.yml.encに保存された資格情報を復号するための定石です。そのままの形で暗記しましょう。

Chapter 13

［A］の解答

```
class Book < ApplicationRecord
  has_one_attached :cover_image
end
```

- クラスメソッドhas_one_attachedを使うと、モデルオブジェクトに対して1個のファイルを添付できるようになります。

［B］の解答

```
<%= image_tag @book.cover_image.variant(resize: "90x114")
%>
```

- variantは画像データを変換するメソッドです。resizeオプションに文字列"90x114"を指定すると、画像の縦横比を維持したまま幅90ピクセル、高さ114ピクセルの範囲で最も大きくなるように画像を拡大・縮小します。

Chapter 14

［A］の解答

```
class Book < ApplicationRecord
  has_many :lending_records
  has_many :users, through: :lending_records
end

class User < ApplicationRecord
  has_many :lending_records
  has_many :books, through: :lending_records
end

class LendingRecord < ApplicationRecord
  belongs_to :book
```

```
  belongs_to :user
end
```

- モデルAとモデルBを多対多で関連付けるときには、モデルAと中間テーブルのモデルCを1対多で関連付け、クラスメソッドhas_manyをthroughオプション付きで呼び出します。同様にモデルAとモデルCも1対多で関連付け、クラスメソッドhas_manyを呼び出します。

［B］の解答

```
user = User.find(5)
book = Book.find(34)
user.books << book
```

- <<は演算子のように見えますが、「books <<」全体でひとつのメソッドです。

Chapter 15

［A］の解答

```
Rails.application.routes.draw do
  namespace :admin do
    resources :users
```

```
  end
end
```

- ルーティングに名前空間を導入するにはnamespaceメソッドを使用します。

［B］の解答

```
<%= link_to "会員の詳細", [:admin, @user] %>
<%= link_to "会員の編集", [:edit, :admin, @user] %>
<%= link_to "会員の削除", [:admin, @user],
  method: :delete, data: { confirm: "本当に削除しますか？" }%>
```

- 「会員の編集」へのリンクでは、[:admin, :edit, @user]ではなく[:edit, :admin, @user]と指定します。

■著者紹介

黒田 努（くろだ つとむ）

東京大学教養学部卒。同大学院総合文化研究科博士課程満期退学。ギリシャ近現代史専攻。専門調査員として、在ギリシャ日本国大使館に3年間勤務。中学生の頃に出会ったコンピュータの誘惑に負け、IT業界に転身。
株式会社ザッパラス技術部長、株式会社イオレ取締役を経て、技術コンサルティングとIT教育を事業の主軸とする株式会社オイアクスを設立。現在、同社代表取締役社長。また、2011年末にRuby on Railsによるウェブサービス開発専業の株式会社ルビキタスを知人と共同で設立し同社代表に就任（オイアクス社長と兼任）。
株式会社オイアクス：https://www.oiax.co.jp/
株式会社ルビキタス：https://rubyquitous.co.jp/
Twitter：tkrd_oiax

佐藤和人（さとう かずと）

東京大学文学部卒。「インターネットマガジン」でウェブ制作関連の記事を手がけ、現在フリーライター。プログラミングとウェブ関連技術がおもなテーマ。大学時代に黒田努にAWKを教わったのがプログラミングを始めたきっかけ。
『できるホームページHTML入門』『できる大事典HTML&CSS』『基礎Ajax+JavaScript』（いずれもインプレスジャパン）など著書多数。
2012年1月より株式会社ルビキタス勤務。2015年2月より株式会社ルビキタス取締役。

kazuto.book@gmail.com

■執筆協力

藤山啓子、町田耕

■監修者

株式会社オイアクス

Ruby on Rails専門のIT教育・コンサルティング会社。Railsを活用した生産性向上ノウハウの提供と人材育成をテーマに事業展開中。開発者やウェブデザイナーを対象とする各種のセミナー、ワークショップを実施。
会社ホームページ：https://www.oiax.co.jp/
ブログサイト：https://www.oiax.jp/

■STAFF

カバーデザイン	ハヤカワデザイン・早川いくを
本文デザイン（紙刊行版）	嶋健夫、轟木亜紀子（トップスタジオ）
DTP制作・EPUB制作	武藤 健志（株式会社トップスタジオ）
イラスト	チカツ タケオ
編集協力	TSUC

■商品に関する問い合わせ先

インプレスブックスのお問い合わせフォームより入力してください。
https://book.impress.co.jp/info/
上記フォームがご利用頂けない場合のメールでの問い合わせ先
info@impress.co.jp

- 本書の内容に関するご質問は、お問い合わせフォーム、メールまたは封書にて書名・ISBN・お名前・電話番号と該当するページや具体的な質問内容、お使いの動作環境などを明記のうえ、お問い合わせください。
- 電話やFAX等でのご質問には対応しておりません。なお、本書の範囲を超える質問に関しましてはお答えできませんのでご了承ください。
- インプレスブックス（https://book.impress.co.jp/）では、本書を含めインプレスの出版物に関するサポート情報などを提供しておりますのでそちらもご覧ください。
- 該当書籍の奥付に記載されている初版発行日から1年が経過した場合、もしくは該当書籍で紹介している製品やサービスについて提供会社によるサポートが終了した場合は、ご質問にお答えしかねる場合があります。

改訂4版基礎Ruby on Rails

2018年09月11日　初版第1刷発行

著者　　黒田 努・佐藤 和人
発行人　小川 亨
編集人　高橋隆志
発行所　株式会社インプレス

〒101-0051 東京都千代田区神田神保町一丁目 105番地
ホームページ　https://book.impress.co.jp/

ISBN978-4-2950-0460-8