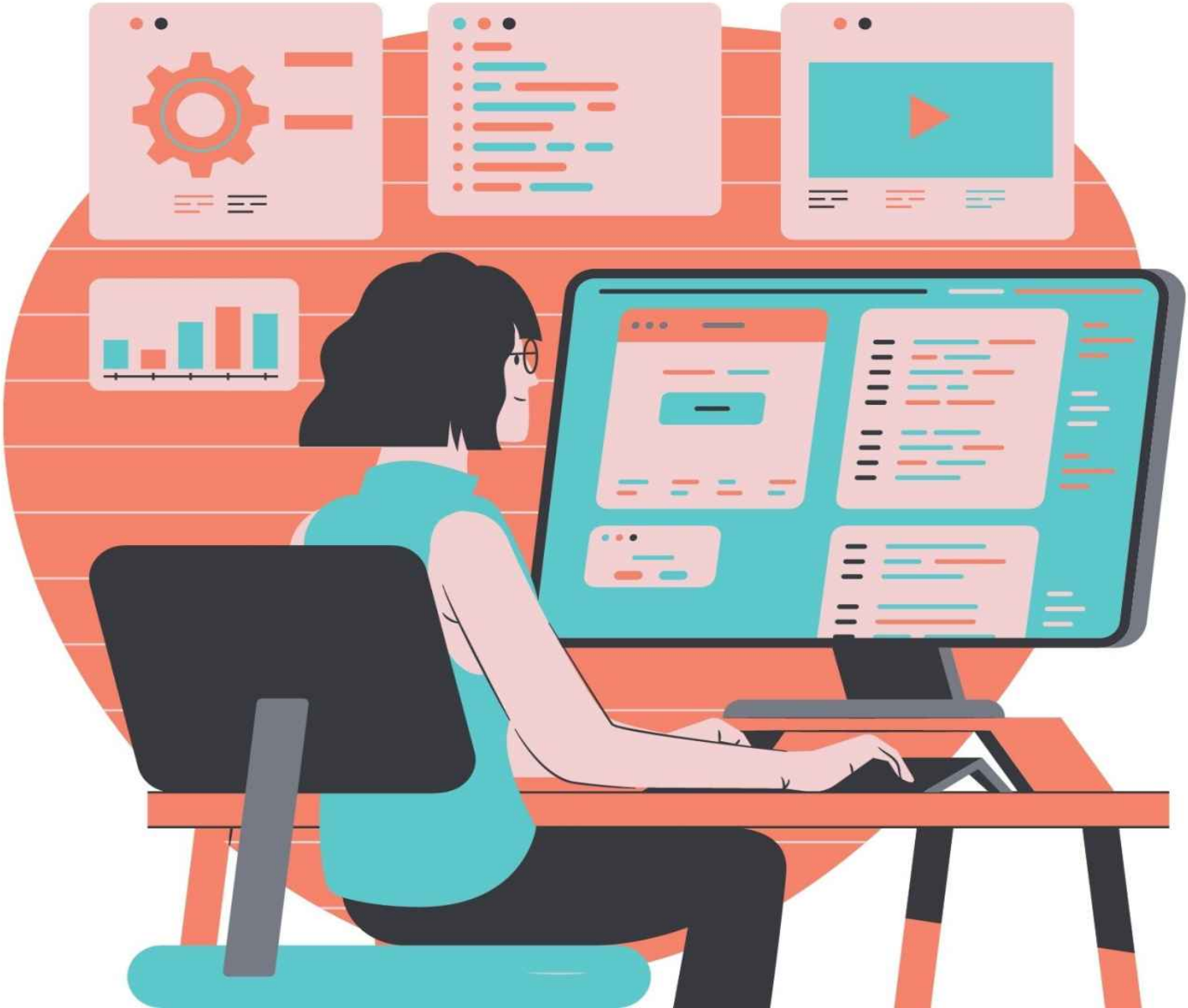
１週間で！
基礎から学ぶ
Ruby on Rails

Minatomi著



目次

- はじめに
  - 対象読者
  - 本書の最終目標
  - 本書に書かれていないこと
  - 題材
  - 本書の進め方
- 環境を整えよう
  - cloud9
  - cloud9の画面説明
- Rubyの復習
  - オブジェクト指向
  - 継承
- MVCモデル
- 【基礎編】新しくアプリのテンプレートを作成しよう
  - フォルダ構成の説明
- モデル作成
  - rails g modelコマンドでモデルを作ってみよう
  - 初期データを入れよう
  - rails console
  - まとめ
  - データベースについて
  - マイグレーションファイル

index画面の開発
- ルーティング
- コントローラ
- ビュー

bootstrapの導入

show画面の開発
- ルーティング
- コントローラ
- ビュー
- リンク

new画面の開発
- ルーティング
- コントローラ
- ビュー

createの開発
- ルーティング
- コントローラ
- バリデーション

edit画面の開発
- ルーティング
- コントローラ
- ビュー

updateの開発
- ルーティング

- コントローラ
- リンク

destroyの開発

- ルーティング
- コントローラ
- リンク

【実践編】アプリをより良くしていこう

- フラッシュメッセージ
- レイアウトを少し整える
- before_action
- resourcesメソッド
- パーシャル

【応用編】検索と集計を実装しよう

- 検索
- トータルを計算する

最終課題

RESTful

おわりに

完走者特典

はじめに

本書はRubyベースの高性能ウェブフレームワークであるRuby on Rails(以下Rails)の入門書です。
Railsはプログラミング言語Rubyをベースに作られたWebサービスを作るためのフレームワークです。フレームワークを直訳すると枠組みという意味になるのですが、いまいちピンとこないですよね。
直感的なイメージでいうと、フレームワークとは「更地の街」です。そこにはまだ建物はありません。ですが、町としての最低限の水道や電気、道路などはあります。私たちはこの更地の街にビルや家、公園などを作って街を育てていきます。町を作る人やチームによって全く雰囲気の異なる街が出来上がるように、作る人やチームが異なると、同じフレームワークを使っても、全く異なるウェブサービスが出来上がります。Railsを使っているウェブサービスの例としては、CookpadやCrowdWorks、食べログ、Twitter, huluなどがあります。
街の例えで言うならば、本書はまず一軒の家を建築することを目指します。この一軒の家の建築がRailsでは最も基本的です。
家一つを作るにも、私たちは新しく木材を切ったり、鉄を溶かして鉄筋を作る必要はありません。Railsがほとんどの資材を用意してくれているので、私たちは資材を組み合わせて家を建てていくだけです。
しかし、いくら資材を用意されているからといっても、家を一軒建築するのは簡単なことではありません。HTML/CSSやRubyの入門書を難なくこなせた方でも、Railsは難しく感じるでしょう。
一般的なRailsの教科書は、街の作り方を網羅的に解説しています。だからこそ、家を一軒建てるには何が必要か、どのような順序で進めていけば良いのか、

などの初心者が本当に知るべきことにページを割けず、逆に情報不足となってしまっています。
そこで本書では、Railsが提供する機能の大部分を削ぎ落とし、Railsでウェブサービスを作る上で必ず必要となる「CRUD」という考え方にフォーカスして解説しています。
CRUDとは、Create, Read, Update, Deleteの頭文字を取ったもので、データに対する一連の処理を表します。例えば、メモサービスを考えてみると、メモの新規登録、メモの参照、メモの更新、メモの削除のことです。
たった 4 つしかありませんが、Railsでウェブサービスを作る上では、ほとんどの機能はCRUDのいずれかに分類されます。
本書では、単一のモデル(データ)に対するCRUDの作成を 1 から順番に丁寧に解説していきます。先程の例でいうと、単一モデルのCRUDが、一軒の家の建築に相当します。

対象読者

- HTML/CSS, Rubyの基礎は学習し終えた方
- Railsをこれから学ぼうと思っている方
- Railsの教材を読んでみたけど、良く分からなかった方
- 他の言語でウェブ開発を学習された方

本書の最終目標

本書を読み終えた方が到達していただきたい最終目標は、１から単一モデルのCRUDを作成できる様になることです。
CRUDはRailsの中でも基本中の基本ですし、根幹です。本当に大事なところなのですが、CRUD作成がRails入門者には難しく、なかなかこの壁を突破することができません。
本書では、小さなアプリを少しずつ開発していくことで、CRUDやMVCなどの諸概念に自然に触れ、少しずつ慣れていくことを目指しています。

本書に書かれていないこと

- gitの使い方やgitを使ったチーム開発
- 開発手法
- ２つ以上のモデルの関連
- デプロイ方法
- ログイン機能 ... etc

題材

本書では、簡単な家計簿アプリを作りながらRailsのCRUDを学んでいきます。
家計簿アプリの機能は非常に単純です。

- (C: create)家計簿データの登録
- (R: read)参照、一覧表示
- (U: update)更新
- (D: destroy)削除

良くあるブログアプリの作成よりも、もっと実用的な物の方が面白いと思い、家計簿を題材としました。
本書で完成するアプリをベースに、機能を追加したり、デザインを工夫することもできます。

本書の進め方

あくまでも１つの例ですが、本書の進め方をご紹介します。

- まずはコードを書かずに本書をざっと一通り読んでいただいて、全体の流れや用語を拾っていただければと思います。本書では専門的な用語には全て説明をつけているつもりですが、説明が難解だったり不足している部分もあります。１度目の読書で分からない単語をリストアップし、google検索などを使って調べてみてください。
- ２週目から実際にコードを書いてアプリを作成してください。ただし、本書の最後は少し応用的な内容になっていますので。２週目は「destroyの実装」まで進んでいただいたら、一度終了してください。
- ２週目が終わると、本書の最初から３週目をスタートさせます。３週目では、本書を最後までやり切ってください。３週本書を読んでいただくと、大まかなRailsでの開発が理解できるようになってきます。かなり実務で書くコードに近づきます。
- ４週目は、本書を参考にしながら、オリジナルのアプリを読者自身で考えて、作成してください。例えば、映画や本の評論、体重・体脂肪率管理アプリなどは、本書で学習した範囲で作成することができます。

１週目で全く理解できなくても、全然問題ではありません。
脳は新しい概念や用語を理解するのに、6回の見聞きが必要だと言われています。6回見聞きする事で、脳が慣れ、理解できるようになります。理解できないのではなく、脳が馴染んでいないだけです。

４週目をクリアすると、めでたく免許皆伝です！４週目以降は、より専門的な書籍で学習を進めてください。

環境を整えよう

まずはRailsで開発をしていく準備をしていきましょう。開発するための準備を「開発環境構築」と言います。

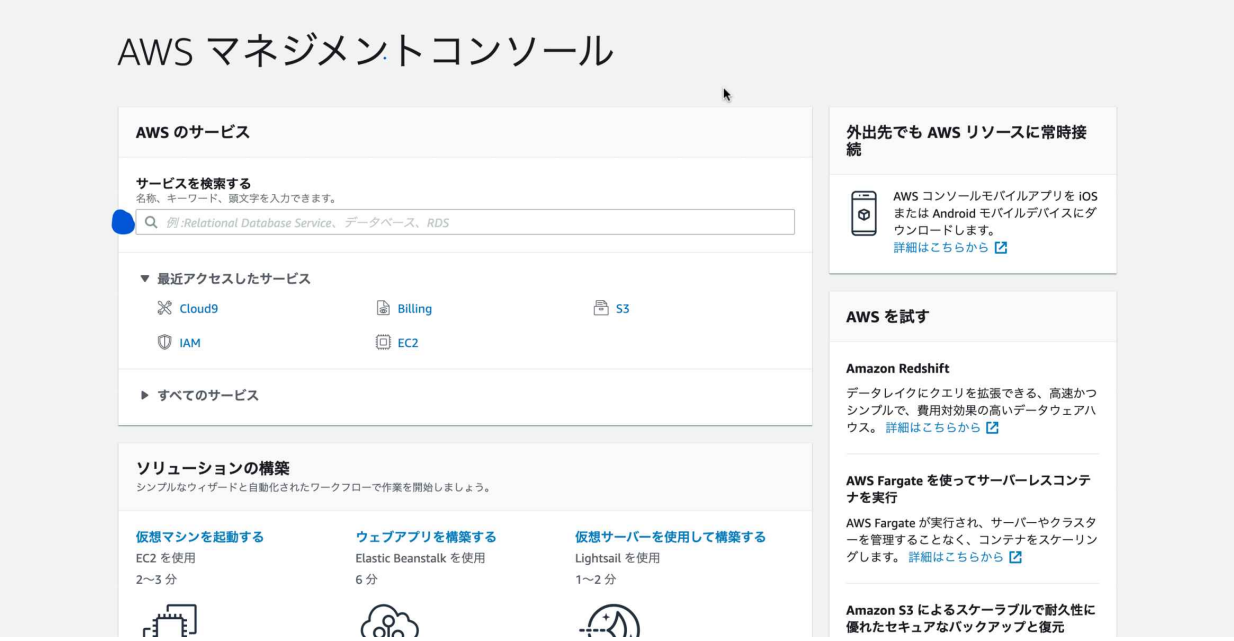
環境というと、自然とかエコみたいな言葉を連想してしまいそうですが、プログラミングの世界では、いろいろなところに環境という言葉が登場します。

環境の「環」の字には輪っか、めぐる、囲むの意味がある様に、環境は「プログラミングを取り巻く周辺」のことを意味します。プログラミングを行うパソコンやサーバーは、プログラミングのすぐ近く周辺に存在するので環境の一部です。

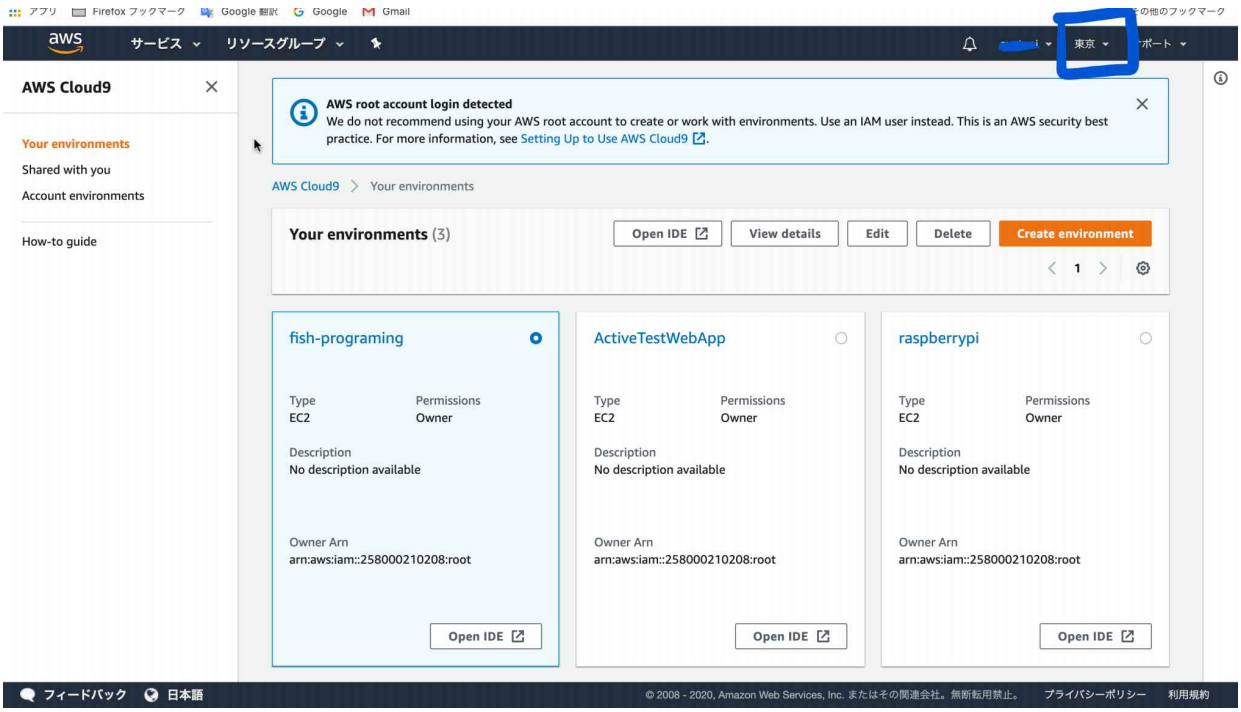
今回は、環境構築が比較的簡単なcloud9を使っていきます。

cloud9

cloud9はもともと独立したサービスでしたが、Amazonに買収され、Amazonが展開するクラウド事業AWS(Amazon Web Services)に取り込まれました。
cloud9はブラウザさえあれば、どこからでも接続することができる、リモート(クラウド)開発環境です。
ここからは、AWSのアカウントを持っているところから始めるので、AWSのアカウント作成がまだの方は、先にアカウントの作成をお願いします。
さて、AWSのコンソールへログインしたら、以下のような画面になっていると思います。



青い丸印のところで、検索ができるので「cloud9」と入力してください。出てきた候補をクリックすると、cloud9のページへ移動します。
cloud9の画面に移動したら、まずは右上の地域が書かれているところをクリック(下記画像を参照)して「東京」を選んでください。



選択できる地域はたくさんありますが、これら地域はリージョンと呼ばれています。

リージョンとは、AWSのサーバが置かれている場所のことです。具体的な住所までは明かされていません。

リージョンによって使えるサービスが異なってくるのですが、cloud9は日本にある東京リージョンでも使うことができます。東京でなくても良いのですが、地理的に近い方が通信が早いので、基本的には一番近いところを選びます。

次に、上の画像に見えている、オレンジ色のボタン「Create environment」を押して自分専用のサーバを作成していきます。

サーバの名前を登録

「Create environment」を押すと、サーバの作成画面へ移ります。

この画面で最初に入力する「name」は、自分が識別できれば良いので、「Rails-CRUD」とでもしておきましょう。「description」は説明欄になります。必須ではないので、空欄で問題ありません。

nameを入力したら、またオレンジ色の「next step」を押して進みます。

各種設定

次の画面では、サーバーのサイズなどを設定します。

特に何も変更しなくて良いので、そのまま「next step」を押して進みます

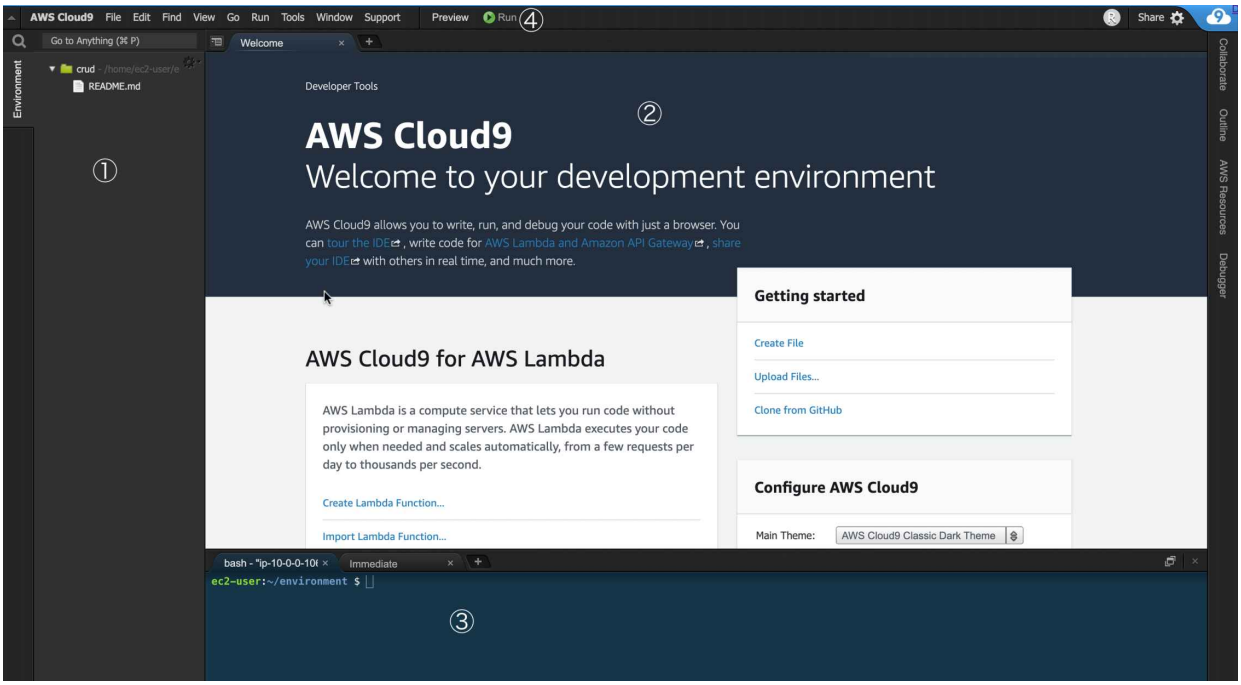
確認画面

最後に確認画面があります。

おそらく問題はないので、最後に「Create environment」をクリックして完了です。

しばらくするとcloud9が立ち上がります。

cloud9の画面説明



cloud9の画面は基本的に、３つのパートに分かれています。

①がファイルやフォルダの一覧を見ることができる場所です。フォルダを開いて、その中にあるファイルを確認したり、フォルダを新規で作成できたりします。

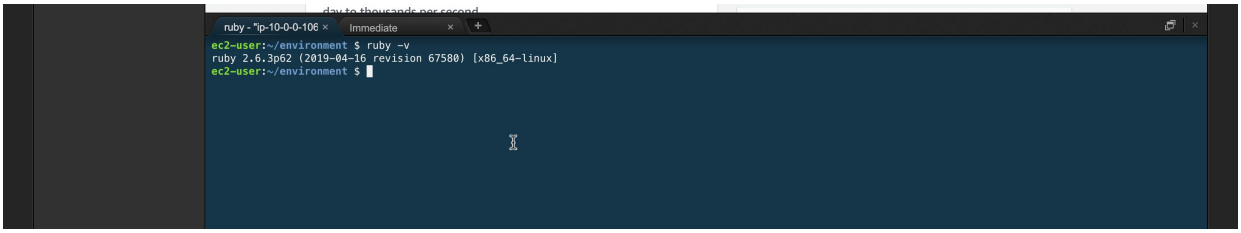
②はエディタと呼ばれる部分で、コードを書いていく場所です。

③はターミナルと呼ばれていて、コマンドを実行するところです。

上記３つのパートに加えて、

④はヘッダメニューです。本書の中で「cloud9のメニュー」と呼べば、ここを指します。

まずは、ターミナルでRubyのバージョンだけ調べておきましょう。以下の画像の様に、ターミナルに「ruby -v」と入力し、エンターを押して、コマンドを実行します。



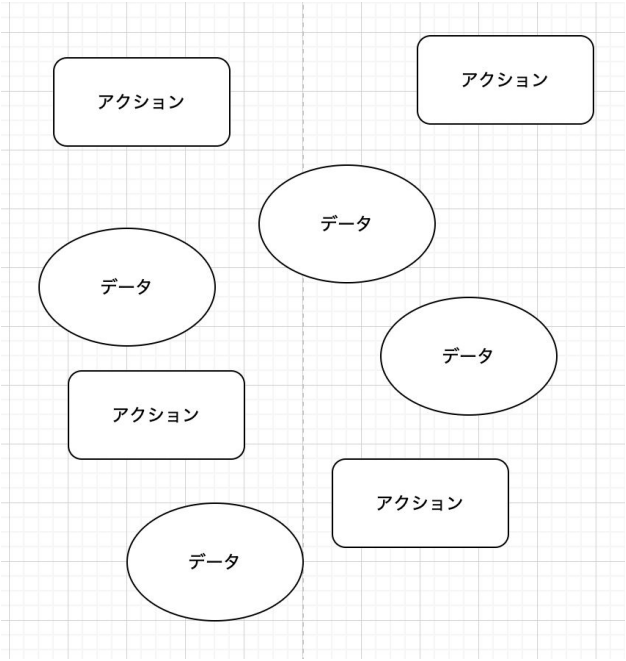
本書執筆時点のRubyのバージョンは2.6.3でした。ここが多少違っていても、特に問題はありません。とにかく、ruby -vが動けば、Rubyは使えるということです。

Rubyの復習

実際にアプリを作る前に、さらっとRubyの復習をしておきましょう。
Rubyは日本人の方が作った、オブジェクト指向のプログラミング言語です。非常に純粋なオブジェクト指向言語であり、プログラムを綺麗に書けることが特徴です。

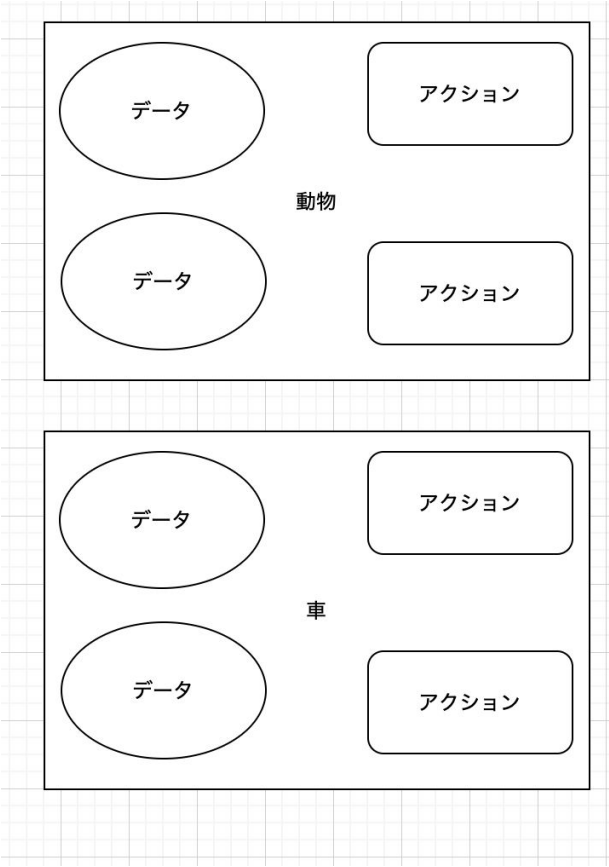
オブジェクト指向

オブジェクト指向というのは、プログラミングスタイルの一つです。
RPGゲームでは、魔法メインで戦うのか、物理攻撃メインで戦うのか、守りを優先するのかなどのスタイルがありますが、プログラミングにも、どの様な方針でプログラムを組み立てていくのかというスタイルがあります。
オブジェクト指向の他には、手続き型や関数型があります。
オブジェクト指向を理解するために、オブジェクト指向ではない手続き型との比較をみてみましょう。
手続き型は、とにかく単純で、順番にプログラムを書いていくだけです。そのため、データや処理(アクション)があちらこちらに分散し、プログラミングコードが複雑になってしまいます。



そこで、関連するデータや処理をまとめようというのがオブジェクト指向です。
例えば、動物と車が登場するプログラムを見てみましょう(実際のプログラムにはもっとたくさんのデータが存在します)。車が動くのと、動物が動くのとでは、処理が

異なりますが、処理の名前でしか分けられていないため、非常にプログラムが見辛いです。
それを、次のように、動物は動物のデータとアクション、車は車のデータとアクションというように、はっきりと分けてしまいます。



データと処理を囲んでしまうことにより、プログラムがすっきりします。この囲んでいるものをオブジェクト指向ではクラスと呼んでいます。
データと処理のセットに名前を付けたものがクラスです。
ただし、クラスは実際にデータを持っていません。具体的なデータを持ち、実際にアクションを引き起こすものを、クラスのインスタンスと言います。
インスタンスは、newメソッドでクラスから作るのでしたね。

```
dog = Dog.new
car = Car.new
```

継承

継承はRubyの中でも、Railsの中でも特に重要な概念です。
継承というと、弟子と師匠のような関係をイメージされるかもしれません。

```
class 弟子 < 師匠
end
```

このようなコードを書くと、師匠が持つデータやメソッドを全て弟子に引き継ぐことができます。
ですが、継承という言葉が持つイメージと、実際の継承の使われ方には、若干のギャップがある場合もあります。
継承の使い方は、「受け継ぐ」ものもあれば、「まとめる」という使い方もできます。例えば、犬と猫のクラスがあったとします。

```
class Dog
  def hello
    puts "hello"
  end
end

class Cat
```

```
    def hello
      puts "hello"
    end
  end
```

このとき、全く同じhelloメソッドを2回も書いているのは、コーデイングするときにも手間ですし、変更があったときには2回も編集しなくてはいけません。そこで、次のように、動物クラスを作って、共通するものをまとめます

```
class Animal
  def hello
    puts "hello"
  end
end

class Dog < Animal
end

class Cat < Animal
end
```

共通のものを抜き出し、まとめることを「抽象化」と言います。
ゴボウとニンジンに共通している部分だけを取り出すと、「根菜」になりますね。いやいや、両方ただの「野菜」だいうこともできます。このように、抽象化には、どのような共通点を抜き出すかによって、抽象化したものが異なります。
抜き出す情報量の違いを「抽象度」と言いいます。根菜と野菜では、野菜の方が抽象度が高いです。よりざっくりとした説明の方が抽象度は高いということです。
継承は抽象度が高いものから低いものへ行います。

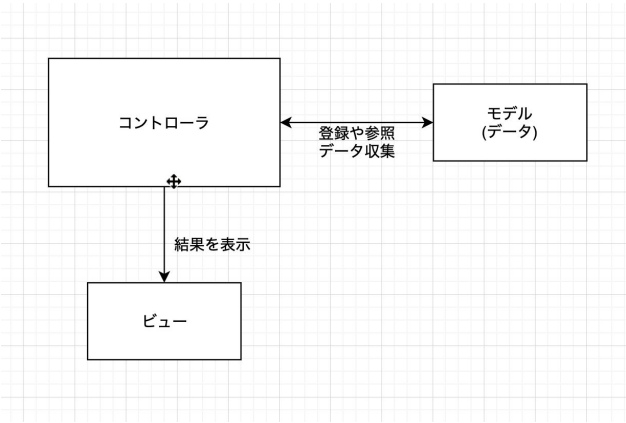
```
class 根菜 < 野菜
end

class Dog < Animal
end
```

ちょうど継承の記号が、抽象度の大小記号のようにもなっていますね。根菜(抽象度低い) < 野菜(抽象度高い)
犬と動物を比べても、動物の方が抽象的です。

MVCモデル

Rubyのクラスや継承といった概念は理解していただけたでしょうか？
Railsへ入っていく前の準備体操として、RailsのベースとなっているMVCモデルについて簡単に解説します。
MVCモデルは、あるプログラムを書く際に、プログラムの役割をはっきりと分けようという考え方の１つです。MVCの他に、MVVMなどもあります。MVVMは主にフロントエンド(JavaScript)で使われています。
MVCはウェブサーバを作るときに良く使われています。Mはデータそのものを表すモデル、Vは画面上に表示されるものを表すビュー、Cは、MやVの使い分けをするコントローラをそれぞれ表しています。



また、Railsでは、役割を分けるだけでなく、モデルはここ、コントローラはここに書くというように、M,V,C毎にフォルダで分けることで、開発もスムーズに行えるような工夫がしてあります。
Rails入門時には、これらの違いはいまいち良くわからないかもしれません。
私も、あるコードをコントローラに記述していたのですが、先輩にコードをみてもらったとき、「このコードはモデルに書いて」と言われて「どうやってどこに書くのか判断すれば良いんだ」と悩みました。

なので、最初は気にせずに、とにかく動くものを作ることを目指しましょう。やっているうちに、自然とわかる様になってきます。慣れるかどうかの問題です。

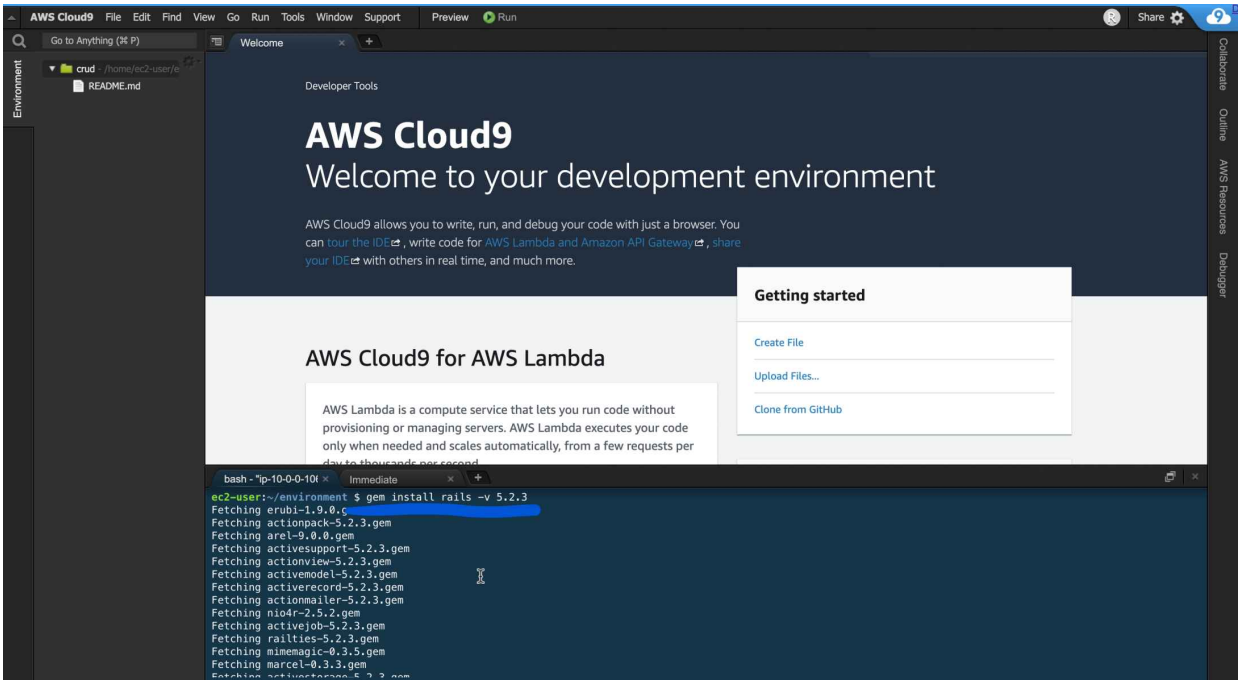
【基礎編】新しくアプリのテンプレートを作成しよう

さて、前置きが長くなりましたが、これから本当にRailsのコーディングに入っていきます。
これからは、コマンドによる操作が多くなってきます。ターミナルで実行するコマンドには、以下のように「$」マークを書いておきます。例えば、以下のような感じです。

```
$ rails c
```

コマンドを実際に動かすときには、「$」マークは入力しないで、後に続く「rails c」だけ書いてください。
また、コマンド実行時には、特にスペースに気をつけてください。「railsc」だとエラーになってしまいます。
rails開発最初の一歩目は、cloud9のターミナルで、railsをインストールしましょう。
railsをインストールするときは、バージョンに注意です。railsの最新バージョンは、執筆時には6.0.0ですが、バージョン6以降webpack(高度なjavascriptを使うことを前提としている)の使用が必要になり、多くの初心者を苦しめています。私のお勧めは、rails5.2系です。5 系の中でも、5.2からはいろいろと便利な機能が追加されています。
それでは、インストールしましょう。下の画像のように、以下のコマンドを実行(入力してエンターキーを押す)してください。

```
$ gem install rails -v 5.2.3
```



少し時間がかかりますが、完了したらまた「ec2-user:~/environment $」に戻ります。

無事にインストールできているか確認しましょう

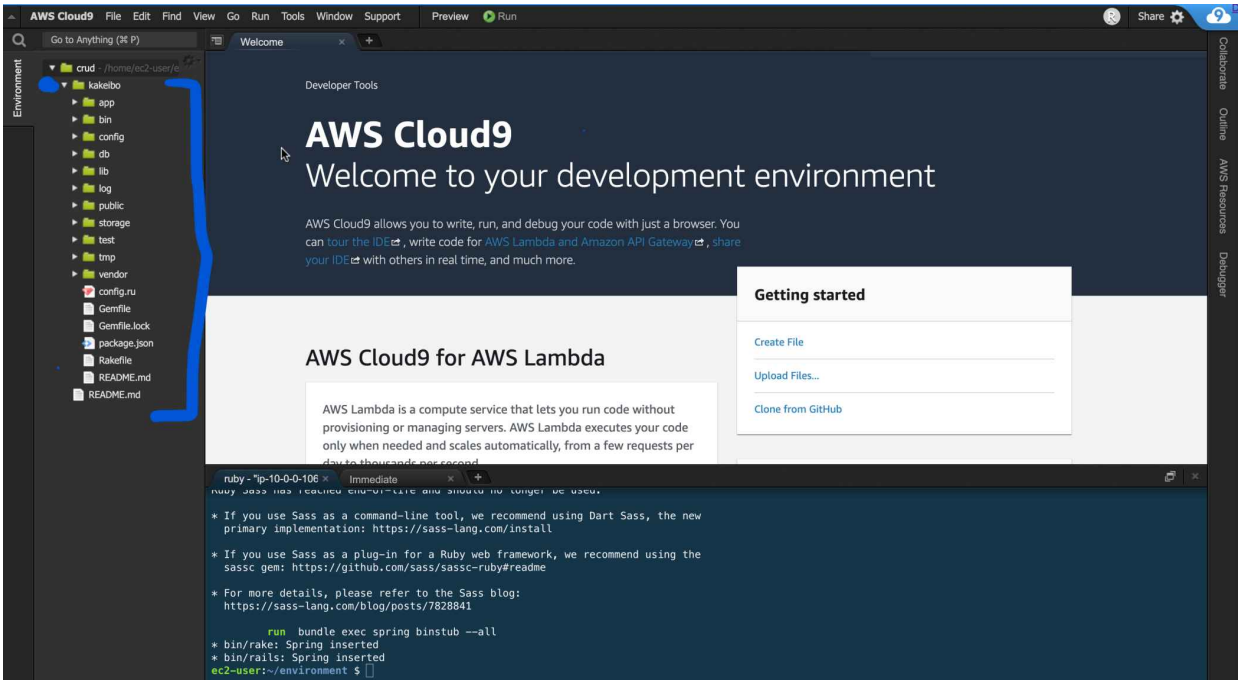
```
$ rails -v
```

このコマンドで「5.2.3」と表示されればOKです。

続いて、以下のコマンドで新しくrailsプロジェクトを作成しましょう。プロジェクトとは、簡単に言えば、アプリに関する全てのファイルが入ったフォルダのことです。

```
$ rails new kakeibo
```

kakeiboというのはプロジェクトの名前です。
時間がかかりますが、成功すると、kakeiboというフォルダが作られます。
kakeiboフォルダの中には、大量にファイルやフォルダが作られます。kakeiboフォルダの左にある三角形をクリックすると、図のようにフォルダが開かれます。



ここまでできたら、ターミナルで以下のコマンドを実行し、kakeiboアプリの中へ入っておきましょう。kakeiboフォルダの外からアプリを操作することはできないためです。cdはchange directoryの略で、directory(ディレクトリ)とはフォルダの別名です。

```
$ cd kakeibo
```

そうすると、ec2-user:~/environment の部分が、ec2-user:~/environment/kakeibo (master)のように変わります。
railsでアプリを作っているときは、特別な状態を除いて、常にこの状態になっていることに注意してください。よく、railsが動かないという状態になっている方の話を聞くのですが、ほとんどの場合が、ec2-user:~/environmentのままになっています。

フォルダ構成の説明

コードを触っていく前に、フォルダの構成について、最初に説明させていただきます。
プロジェクトフォルダには、たくさんファイルやフォルダがありますが、開発で主に触れるのは、appフォルダ、dbフォルダ、configフォルダの 3 つです。
appフォルダは最もよく触るフォルダで、MVCの全てのファイルが入っています。
dbフォルダはデータベースに関連するフォルダです。後で説明するマイグレーションファイルなどが入ります。
configフォルダにはアプリ全般のコンフィグ(設定)ファイルを入れておくフォルダです。configフォルダの中にもたくさんファイルがあるのですが、本書で触れるのはroutes.rbのみです。
最も多く触れる、appフォルダを見ていきましょう。

- assets　cssやjavascript,画像を入れておく場所
- channels リアルタイムチャットを使うときに使うフォルダ
- controllers MVCのCに相当するファイルを入れる
- helpers 補助的なメソッドを記述する
- jobs railsの裏方で動いてくれる機能を使うときに使う
- mailers メール機能を使うときに使う
- models MVCのM
- views MVCのV

この中でも、本書で触れるのはcontrollers, models, viewsの 3 つのみです。

モデル作成

ここから開発を始めていきます。

まずは、データそのものを表すモデルを作成しましょう。開発現場でもモデルが一番最初に作られます。

モデルとは実はただのクラスなのですが、データベースと連携するクラスをRailsではモデルと呼んでいます。

rails g modelコマンドでモデルを作ってみよう

Railsでクラスを作るには、直接ファイルを作成するのではなく、専用のコマンドを使うことが多いです。専用のコマンドを使うと、関連するファイルやファイルの中身もある程度自動的に記述してくれるため、便利です。
モデルを作成する前に、モデルの名前を決めておきましょう。家計簿は英語で「Household account book」と訳されるので、ここでは短くbookという名前のモデルを作ります。また、bookには帳簿という意味もあります。
モデルを作成するには、以下のコマンドを使います。表示の関係上、以下のコードは 2 行に表示されることもありますが、人つながりのコマンドですので、「amount:integer」まで記入してからエンターキーを押して実行してください。

```
$ rails g model Book inout:integer category:string year:integer month:integer amount:integer
```

このコマンドを実行すると、色々とメッセージが出てきます。特に、緑色でcreateと書かれたものは、新しくファイルが作られたことを表しています。
このコマンドで

- db/migrate/20200804015035_create_books.rb
- app/models/book.rb
- test/models/book_test.rb
- test/fixtures/books.yml

というファイルが作られました。
下の２つは今回使わないので、無視します。
一番上のファイルは、マイグレーションファイルと呼ばれるものです。
マイグレーションファイルは、データベースへの変更を記述しておくファイルです。例えば、テーブルを作成したい時や、カラムを追加したい時はマイグレーションファイルを作成します。
ただし、マイグレーションファイルは変更を記述するだけなので、実行しないと意味がありません。以下のコマンドでまだ実行されていないマイグレーションファイルを実行してくれます。

```
$ rails db:migrate
```

このコマンドで、以下のように出力されればOKです

```
==  20200804015035  CreateBooks: migrating
============================================
-- create_table(:books)
   -> 0.0016s
==  20200804015035  CreateBooks: migrated (0.0021s)
===============================
```

細かい数字は違っていても問題ではありません。
上から２つ目はモデルのファイルです。cloud9左のファイル群から、appフォルダの中にあるmodelsフォルダのbook.rbというファイルを見つけて、ダブルクリックすると、ファイルを開くことができます。

```
class Book < ApplicationRecord
end
```

ここでは、ApplicationRecordというクラスを継承した、Bookというクラスを定義しています。Bookの中にはメソッドなど、何も定義されていないですよね。これはRubyの復習の章で見た、Animalクラスを継承したDogクラスと似ています。

```
class Dog < Animal
end
```

ApplicationRecordを継承することにより、Bookクラスに何もメソッドを定義しなくても、いろいろなメソッドが既に使える状態になっています。

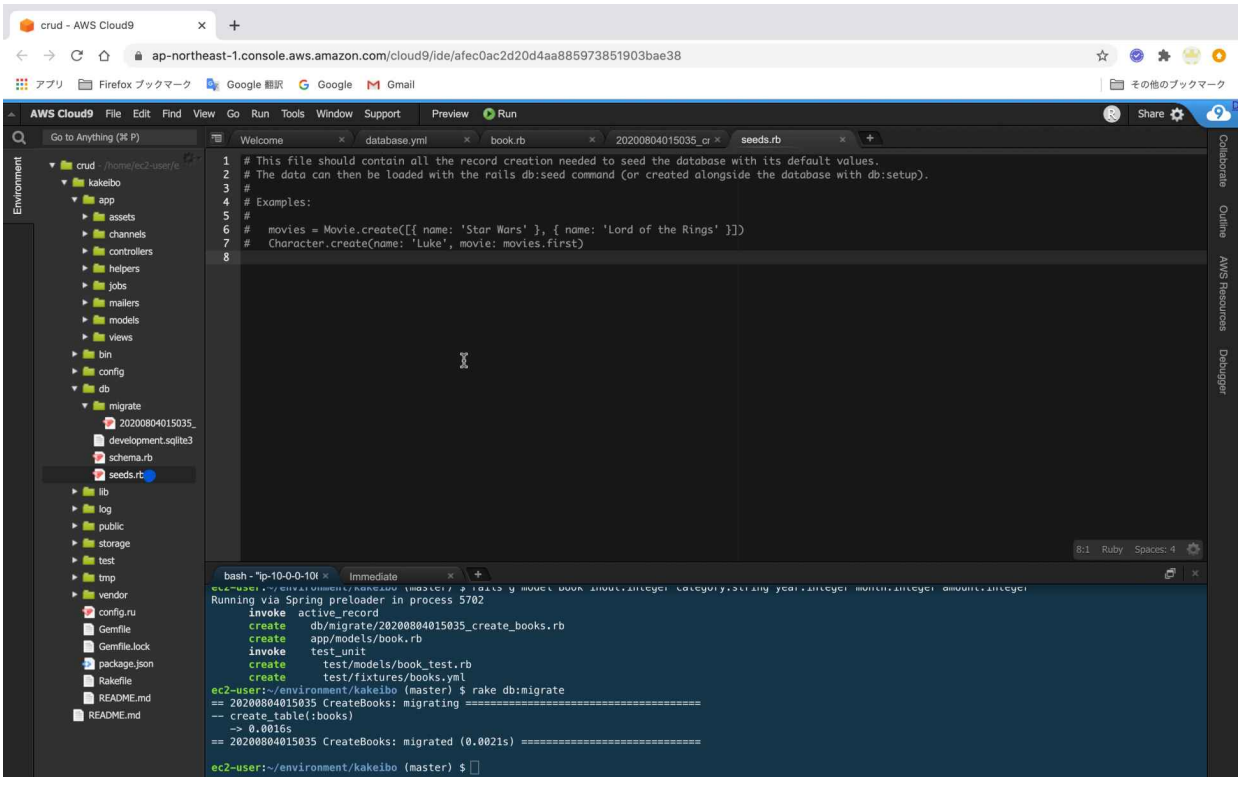
初期データを入れよう

データベースは作成しましたが、肝心のデータが 1 つもない状態です。このまま開発しても、データがないとつまらないので、適当にデータを先に登録しておきましょう。

ここでは、seedを使った方法と、rails consoleを使った方法の 2 つを紹介します。開発ではどちらもよく使います。

まずはRailsのseedという機能を使って、データを登録しましょう。seedは「種」という意味です。seedでは、seeds.rbというファイルに、データを登録するコードを書いて、実行することにより、データベースへデータを登録することができます。

ここからコードを書いていくわけですが、編集するファイルはdbフォルダのseeds.rbというファイルです。



今後は短く、編集するファイルを「db/seeds.rb」のように書きます。dbフォルダにあるseeds.rbファイルという意味です。
また、コードと編集するファイル名を合わせて以下のようにも書きます。

- db/seeds.rb

```
編集するコード
```

では、実際にseeds.rbへプログラムを書いていきましょう

- db/seeds.rb

```
Book.create(inout: 1, category: "給料", year: 2020, month: 7, amount: 30)
Book.create(inout: 2, category: "家賃", year: 2020, month: 7, amount: 8)
Book.create(inout: 2, category: "食費", year: 2020, month: 7, amount: 6)
Book.create(inout: 2, category: "光熱費・水道", year: 2020, month: 7, amount: 3)
Book.create(inout: 2, category: "保険", year: 2020, month: 7, amount: 2)
```

なかなかリアなデータですね。もちろんこれは架空のデータです。著者の実際の家計簿ではないですよ。
ここでデータの種類について説明しておきます。

- inout： 1なら収入、2なら支出を表しています
- category： 収支の種類
- year： 対象年度
- month： 対象月
- amount： 金額(万円)

これらのデータをそれぞれカラムとも呼びます。「Bookモデルのmonthカラム」の様な使い方です。
createは、新しくデータを作るためのメソッド(クラスメソッド)です。引数として、データを渡せば、データベースへ登録してくれます。idカラムやcreated_at, updated_atはcreateを実行したタイミングで自動的に付与されます。
seeds.rbファイルは、コードを書くだけではダメで、実行しなくてはいけません。以下のコマンドで実行することができます。

```
$ rails db:seed
```

特にエラーが出なければOKです。
rails db:seedコマンドが行なっていることは、非常に単純で、seeds.rbファイルを実行しているだけです。

rails console

データがちゃんと入ったか、確認しましょう。
ここで使うのは、rails consoleという機能です。railsアプリ内のモデルを自由に操作することができる機能です。irbのrails版ですね。
irbと同じく、rails consoleではただRubyのプログラムを実行するだけです。rails consoleに書いたコードが、どこかのファイルへ保存されるわけではありません。rails consoleへ書いたコードはrails consoleを終了させると消えてしまいます。
ターミナルへrails cと入力して、実行しましょう

```
$ rails c
```

以下のような状態になれば大丈夫です。

```
Running via Spring preloader in process 6673
Loading development environment (Rails 5.2.4.3)
2.6.3 :001 >
```

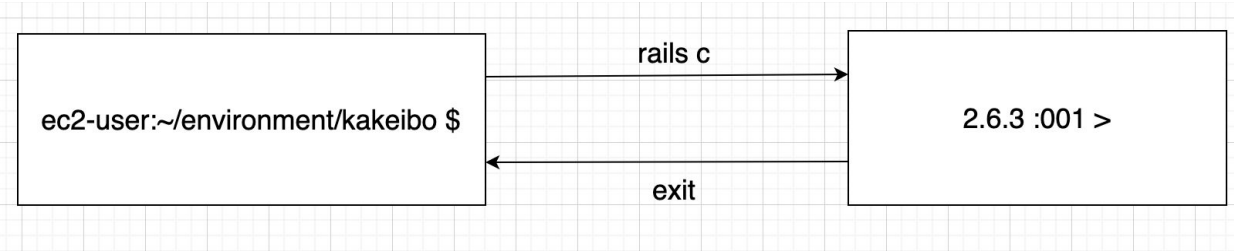
この「>」に続けて、rubyのコードを入力することができます。
通常のターミナルコマンドとの区別がややこしいので、rails consoleで入力するコードは、以下のように先頭に「>」を付けます。

```
> コード
```

通常のターミナルで実行するコマンドは、引き続き先頭に「$」を付けます。
rails consoleを終了するには、

```
> exit
```

とします。



exitで抜けた方は、もう一度rails consoleに戻ってきてください。
それでは、先ほどseedで登録したデータを全て取得してみましょう

```
> Book.all
```

allというのは、データベースに保存されているbooksテーブルのレコードを全て取得するメソッドです。メソッドなので当然返り値があるのですが、rails consoleではメ

ソッドの返り値は次の様に出力されます。

```
2.6.3 :001 > Book.all
  Book Load (0.4ms) SELECT "books".* FROM "books" LIMIT ?  [["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<Book id: 1, inout: 1, category: "給料", year: 2020, month: 7, amount: 30, created_at: "2020-08-04 02:16:40", updated_at: "2020-08-04 02:16:40">, #<Book id: 2, inout: 2, category: "家賃", year: 2020, month: 7, amount: 8, created_at: "2020-08-04 02:16:40", updated_at: "2020-08-04 02:16:40">, #<Book id: 3, inout: 2, category: "食費", year: 2020, month: 7, amount: 6, created_at: "2020-08-04 02:16:40", updated_at: "2020-08-04 02:16:40">, #<Book id: 4, inout: 2, category: "光熱費・水道", year: 2020, month: 7, amount: 3, created_at: "2020-08-04 02:16:40", updated_at: "2020-08-04 02:16:40">, #<Book id: 5, inout: 2, category: "保険", year: 2020, month: 7, amount: 2, created_at: "2020-08-04 02:16:40", updated_at: "2020-08-04 02:16:40">]>
```

うわっ！見にくいですね。
見やすくする方法もあるのですが、ここではちょっと我慢してこのまま続けましょう。
よくみると、先ほどseedで登録したデータがあるのが分かります。1 つ抜粋してみます。

```
#<Book id: 1, inout: 1, category: "給料", year: 2020, month: 7, amount: 30, created_at: "2020-08-04 02:16:40", updated_at: "2020-08-04 02:16:40">
```

idとcreated_at, updated_atという見慣れない 3 つのカラムにも値が入っています。
idはデータを識別するための固有の番号です。idは、データを新規登録するたびに計算されて、自動的に全てのデータに付与されるようになっています。
created_at, updated_atはそれぞれデータの作成日時と更新日時です。こちらもidと同じく、Railsが自動的に値を入れてくれます。
idカラムの値は絶対に被らないので、特定のデータだけデータが欲しいとき(例えば、詳細画面や更新画面)に使います。
rails consoleで次のコードを実行してください。2 行書かれていますが、1 行書いたら、エンターキーで実行してください。

```
> id = 2
> Book.find(id)
```

このコードでデータベースに登録されているデータを 1 件だけ取得できます。2というのが、データ固有の番号(id)です。つまり、idが2である家計簿データをデータベースから取り出しています。

rails consoleでは、データの取得だけでなく、データの登録をすることもできます。

```
> Book.create(inout: 1, category: "副業", year: 2020, month: 7, amount: 2)
```

これがデータを登録するもう一つの方法です。seeds.rbに書いたコードと同じコード(値は異なる)を書いて実行すればいいだけです。rails consoleで書いたコードは消えますが、実行した結果は消えないので、このデータはrails consoleを終了しても、ちゃんとデータベースへ登録されたままになります。
rails consoleの中でデータを更新することもできます。
データの更新(update)は、特定のデータに対して行います。そのため、まず 1 行目ではfindメソッドによってデータベースからデータを取り出しています。
２行目で、updateメソッドを使い、データを更新しています。この更新はもちろんデータベースへも反映されています。

```
> book = Book.find(1)
> book.update(amount: 31)
```

findメソッドで取り出したインスタンスからは個別のカラムの値を取り出すこともできます。

```
> book.year
```

逆に、値を入れることもできます。

```
> book.amount = 32
```

ただし、これでは値を入れただけなので、データベースには反映されません。最後にsaveメソッドで保存してはじめてデータベースへ反映されます。

```
> book.save
```

データの取得や新規登録(create)はBookに対して行い、更新(update)や保存(save)はインスタンスに対して行います。ここは少し注意が必要です。
これらのメソッドは使っているうちに自然と覚えてくるので、今ここで暗記しようとする必要はありません。

まとめ

- seedを使ってデータ登録

db/seeds.rbへデータ登録のためのコードを書く
例

```
Book.create(inout: 1, category: "副業", year: 2020, month: 7, amount: 2)
```

次のコマンドでseeds.rbに記述したコードを実行する

```
$ rails db:seed
```

- rails consoleを使ってデータ登録

以下のコマンドでrails consoleを起動

```
$ rails c
```

Railsのコードを実行してデータを登録する
例

```
> Book.create(inout: 1, category: "副業", year: 2020, month: 7, amount: 2)
```

- その他のデータ操作

```
# idを使ったデータの取得
> book = Book.find(2)

# カラムの値の取り出し
> book.year

# データの更新
> book.year = 2019
> book.save

もしくは

> book.update(year: 2019)
```

データベースについて

データベースについて、もう少し詳しく知っておきましょう。

データベースは名前の通り、データを溜めておくための箱です。パソコン上の場所やスペースの方がイメージしやすいかもしれません。

データベースはプロジェクトごとに存在します。例えば、現在作成している家計簿アプリにも１つのデータベースが存在し、また別のアプリを作成するには、別のデータベースが必要です。

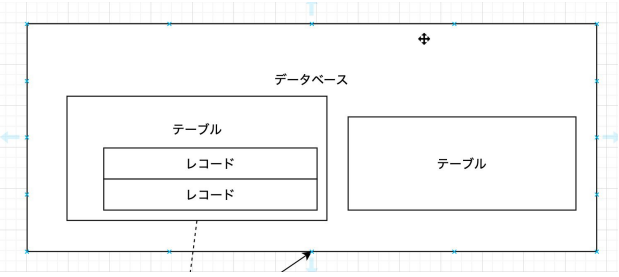
データベースには、どのようにデータが保存されているのでしょうか？

まず、データベースの中で、関連のあるデータごとに、テーブルといわれる単位で分割されています。顧客データや売り上げデータがごちゃ混ぜだと使いにくいですね。これらを顧客テーブル、売り上げテーブルというように分けます。

顧客テーブルの中で、一人分のデータはレコードという単位で保存されています。

１つの商品の売り上げデータも１レコードとして保存されています。

ちょうど、エクセルファイルがデータベス、エクセルのシートがテーブルで、シートの中の１行が一人の顧客を表すイメージです。

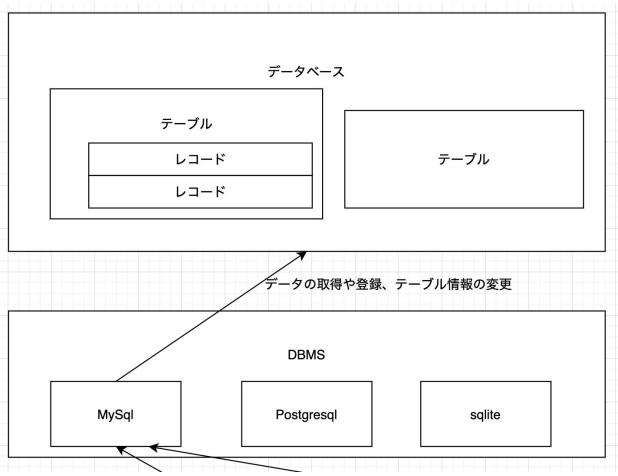


ただ、データベースは場所であるため、それ自体は何の動作もしません。

データベースに対して実際にデータを登録したり、データを取得するのは、DBMS(データベースマネージメントシステム)と呼ばれるソフトウェアが担当しています。DBMSにも様々な種類がありますが、無料で使えるものには、MySQLや

PostgreSQL,SQLiteなどがあります。SQLiteは開発段階でよく使われ、MySQLやPostgreSQLは実運用でよく使われます。
DBMSでは、SQLというRubyとは全く別の特別な言語を用いて、データベースを作ったり、テーブルを追加したり、レコードを追加や更新、削除などを行います。
次の例では、家計簿テーブルに保存されている全てのデータを取得するSQLの例です。このコードは特別な場所で動かす必要があるので、実行しなくても構いません。

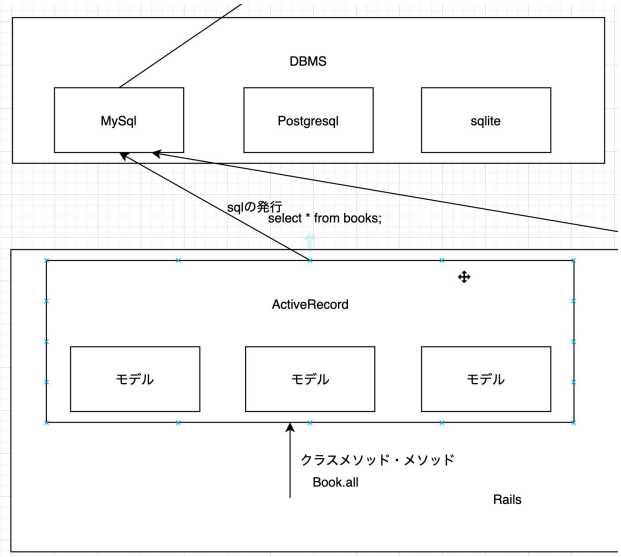
```
select * from books;
```



Rubyの他に、また新しくSQLも覚えるなんて大変ですよね。。
ご安心ください。そこはRailsがちゃんとうまいことやってくれます。RailsにはActiveRecordというライブラリ(Rubyの機能を拡張するもの)が入っており、ActiveRecordを経由することで、SQLを触ることなく、データベースを操作できるようになっています。

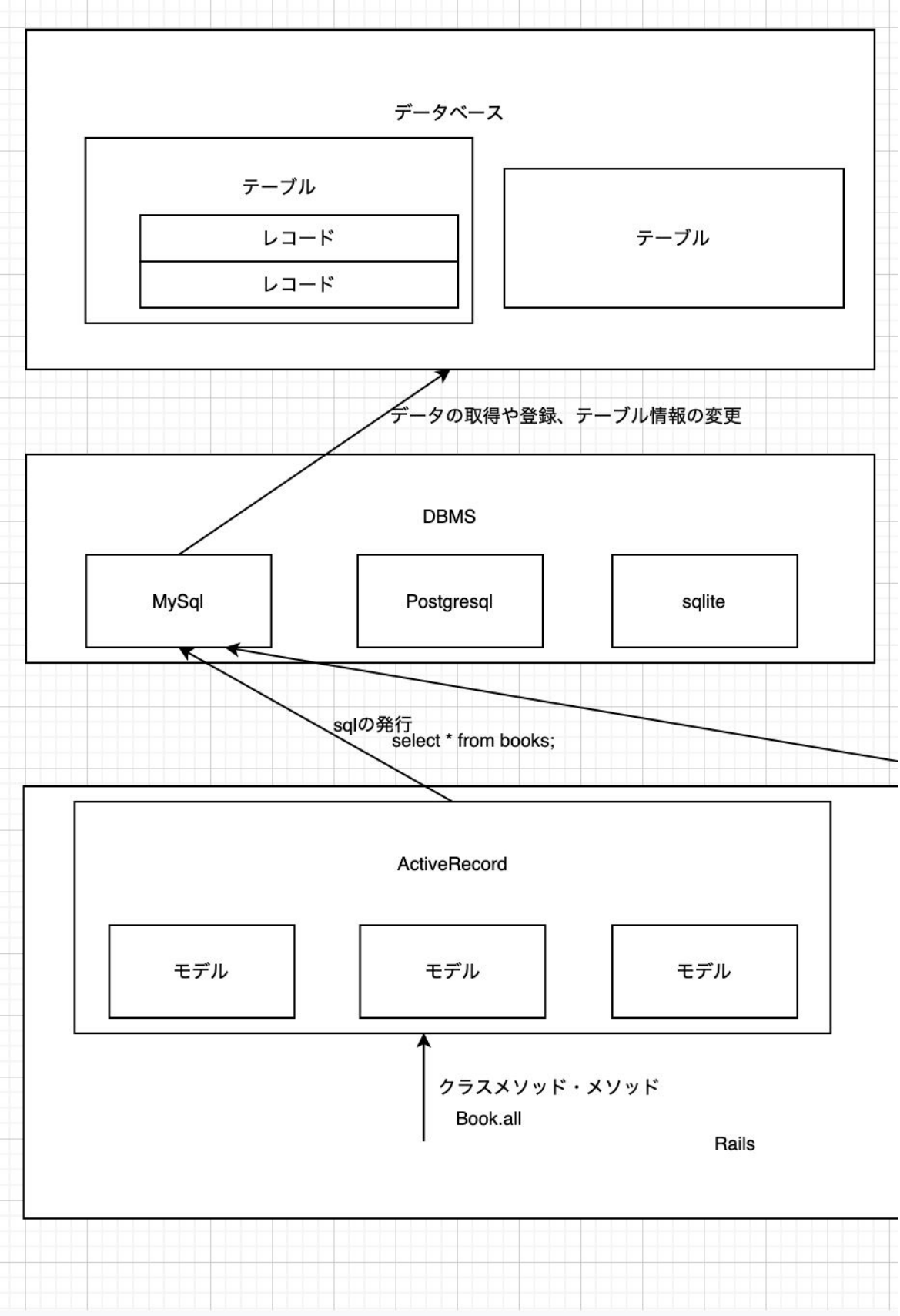
これまでにも、データベースへデータを登録したり、データを取得したりしましたが、SQLを書いていないですよね。
ActiveRecordがやっていることは、RubyのコードをSQLに変換することです。

```
Book.all
 ↓ ActiveRecord
select * from books;
```

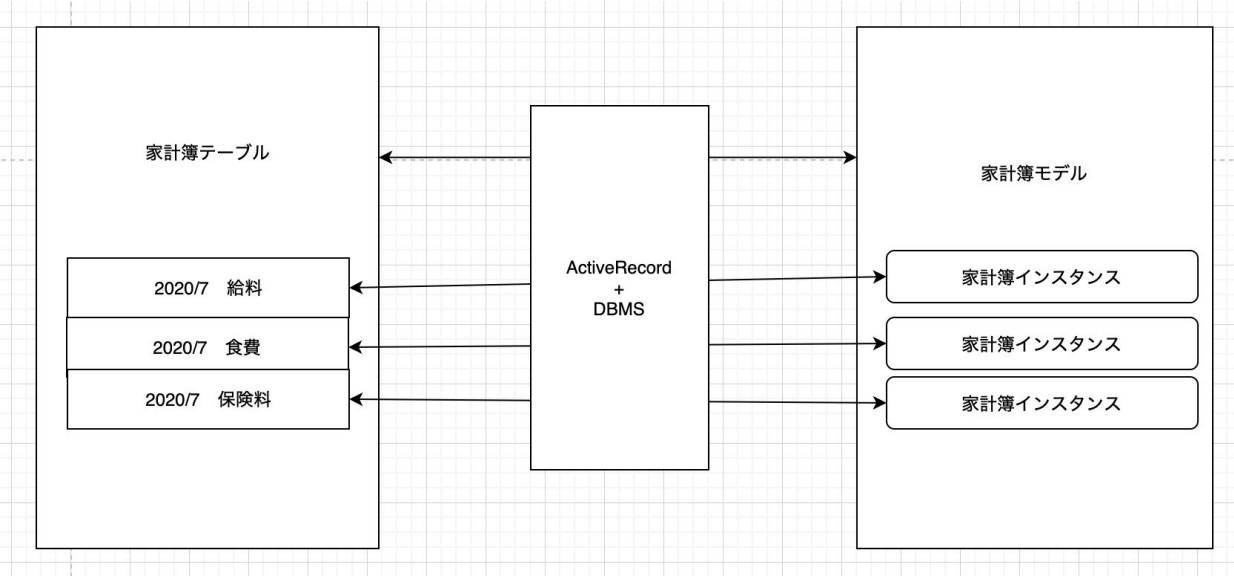


Railsの中のモデル(Bookクラス)にはすでにActiveRecordの機能が組み込まれており、Bookクラスを通してデータベース上のbooksテーブルを操作できるようになっています。
先ほど、seeds.rbを作ってデータを登録したり、rails cからデータを更新したりしましたが、この時すでにActiveRecordの機能を使っています。例えば、allだったり、create, update, saveなどのメソッドです。

私たちは、これらのメソッドを使うだけで、あとはActiveRecordが勝手にSQLに変換し、DBMSへ命令を送ります。そして、DBMSがSQL通りに動き、データベースを操作します。


データベース
テーブル
レコード
レコード
テーブル
データの取得や登録、テーブル情報の変更
DBMS
MySql
Postgresql
sqlite
sqlの発行
select * from books;
ActiveRecord
モデル
モデル
モデル
クラスメソッド・メソッド
Book.all
Rails

RailsとデータベースはActiveRecordとDBMSによって結び付けられていますが、さらに、モデルとテーブル、モデルのインスタンスとレコードがそれぞれ 1 対 1 に対応しています。



ActiveRecordはRubyのコードをSQLへ変換する機能の他に、データベースから取得したデータをインスタンスへと変換するという機能があります。
つまり、データベースの家計簿テーブルに「idが1である2020年7月の給料」データがあり、それをActiveRecord経由で取得すると、自動的に、家計簿モデルのインスタンスが返されます。

```
> id = 1
> book = Book.find(id)
```

このコードが同じことを意味しています。findメソッドでデータベースからデータを取得しています。

そして、返ってきた結果をbookという変数へ代入しています。このbookに入っているものが家計簿モデル(Book)のインスタンスとなります。
ちなみに、ActiveRecordを使わないで、単純にデータベースから取得したデータは次のようなものです。

```
1|1|　給　料　|2020|7|31|2020-08-04　02:44:04.847295|2020-08-07 00:35:39.78224
```

このデータを自動的にBookクラスのインスタンスへ変換してくれるんですから、すごくありがたいですね。

マイグレーションファイル

データベースの話の最後に、マイグレーションについてもう少し解説を付け加えます。
ActiveRecordの機能により、データの登録や更新、検索などはモデルを介して操作できることを見てきました。
では、データベース側の家計簿テーブルは最初から存在していたのでしょうか？
まさか、そんなことはありえないですよね。
データベースのテーブル自体を操作するには、Railsの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、テーブルを新しく作ったり、変更したりすることができます。
マイグレーションは、「マイグレーションファイルの作成」と、「マイグレーションファイルの実行」という 2 段階に分けて行います。
少し前で、

```
rails g model ~
```

というコマンドでモデルを作成しました。
この時、モデルと一緒にマイグレーションファイルも作られています。マイグレーションファイルはdb/migrateというフォルダの中にあります。一つだけファイルがあるの思うので、そちらを見てみましょう。ファイル名の数値はファイルが作られた時間です

- db/migrate/20200804015035_create_books.rb

```
class CreateBooks < ActiveRecord::Migration[5.2]
  def change
    create_table :books do |t|
      t.integer :inout
      t.string :category
      t.integer :year
      t.integer :month
      t.integer :amount

      t.timestamps
    end
  end
end
```

このファイルの内容については本書では詳しくは触れません。inoutがintegerで、categoryがstringとかは見ていただけると、直感的に分かるかと思います。
マイグレーションファイルが作成するところまで行ったら、次は以下のコマンドで実行します。

```
$ rails db:migrate
```

見覚えありますよね？ね？こちらも少し前で実行しました。
実は、このコマンドでマイグレーションファイルを元にテーブルが新しく作られたのでした。
マイグレーションファイルは、実行したら、そのあとは基本的に編集しません。実行したらそれで終わりです。
開発を進めていく中で、新しくモデルを作ることになったら、モデルとマイグレーションファイルを作り、マイグレーションファイルを実行するという流れで進んでいきます。
実際の開発現場だと、モデルは数十個存在します。開発途中でテーブルに変更を加える事は多々あるので、マイグレーションファイルも実際の現場では、モデルの数以上に存在します。

index画面の開発

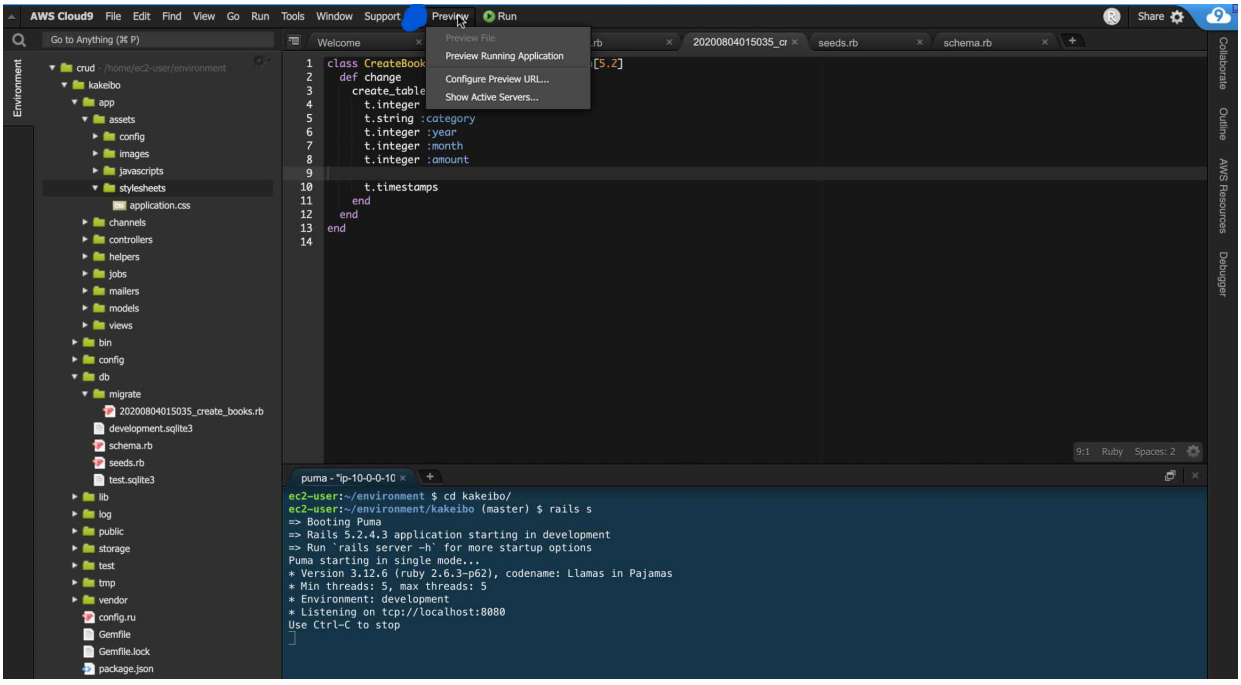
モデルについてはこれくらいにしておき、ここからはMVCのCとVについて見ていきます。CはモデルやビューやViewを使い分ける処理をする部分です。

開発を進めていく前に、ちょっと試しにサーバというものを動かしてみましょう。

```
$ rails s
```

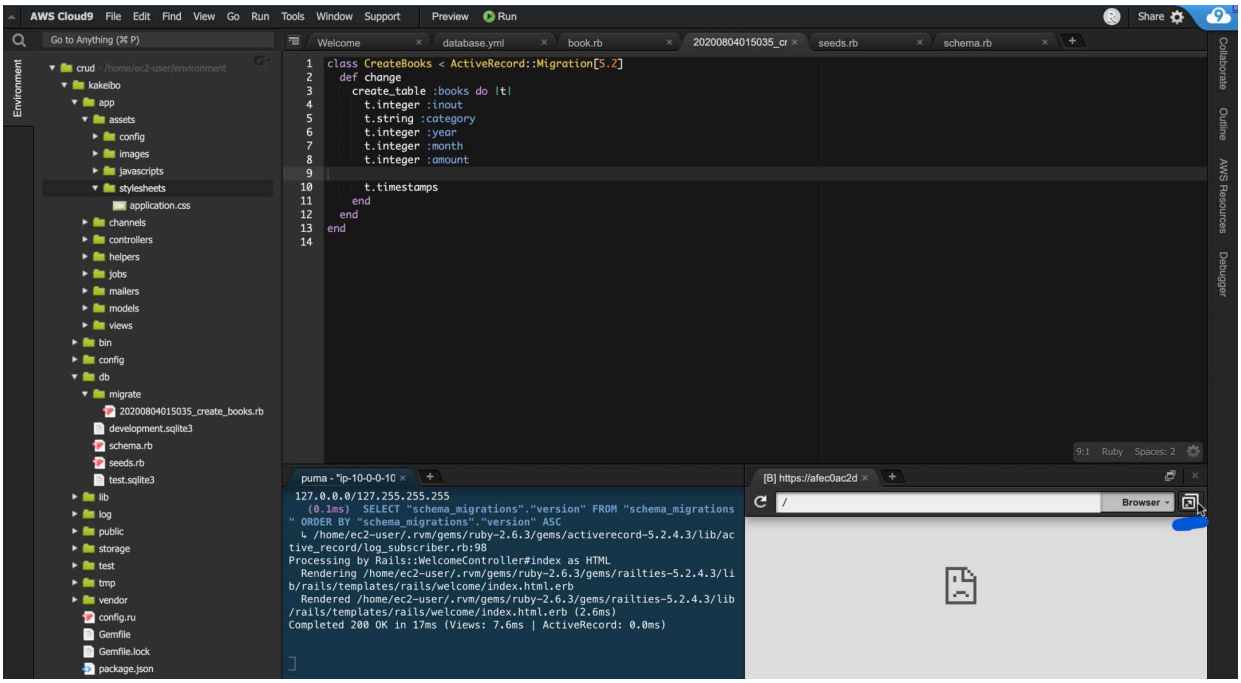
このコマンドでサーバが動き出します。sはserverの頭文字です。

cloud9では、動き出したアプリを見るために、プレビュー機能がついています。

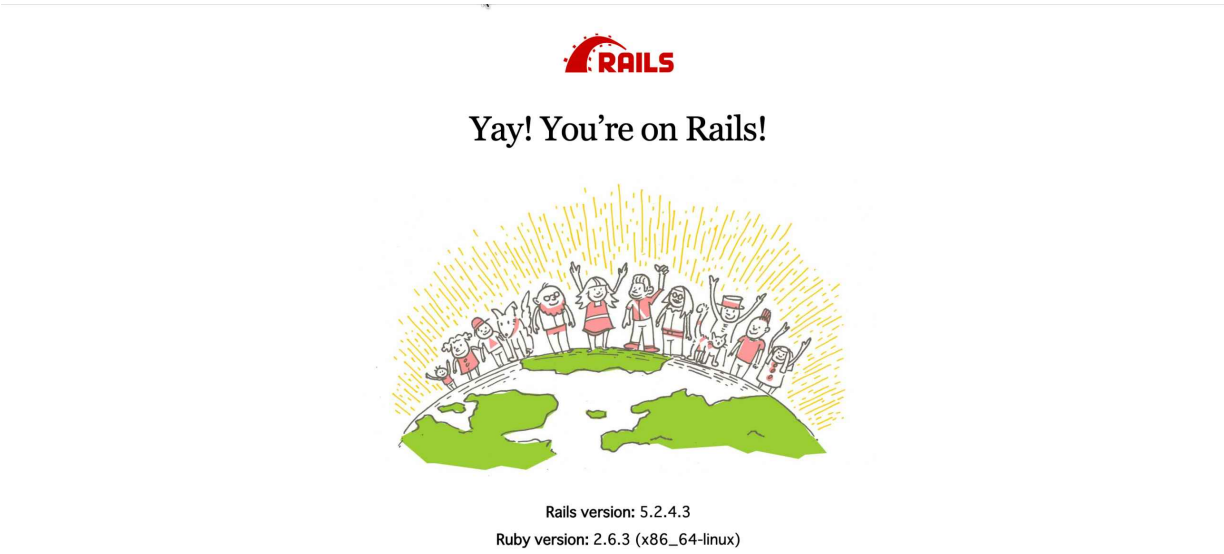


cloud9 のヘッダメニューから Preview を押して、「Preview Running Application」をクリックしてください。

そうすると、プレビュー画面が画面右下に開きますが、このままだと見れないので、
下記画像の青印のボタンを押して、
別タブで開いてください。



下の画像のようなウェルカムページが表示されると、サーバの起動に成功したということです。



ここで私たちが起動したのは、ウェブサーバというものです。ウェブサーバはブラウザからのリクエスト(urlを入力したり、リンクで飛んだり)すると、リクエストに応じたレスポンス(html)を返すプログラムのことです。

> [ブラウザ]リクエスト -> ウェブサーバ -> [ブラウザ]レスポンス(HTML)

ブラウザのアドレスバーにurlが記載されていますが、これがリクエストです。ウェルカムページが表示されたということは、Railsがリクエストを解釈し、適切なウェブページを返すことに成功したことになります。
動いているサーバを止めるには、ターミナルで、「Ctrl+c」を押してください。Ctrlキーを押しながらcキーを押します。
開発を進めている間は、サーバーは動かしっぱなしでも問題ありません。特にappフォルダの中のファイルに変更があっても、サーバを再起動(Ctrl+cで止めて、rails

sで起動)しなくても編集が反映されるようになっています。
サーバーの起動確認ができたので、開発に移っていきましょう。
これから私たちが作っていくのは、家計簿アプリでしたね。
この章ではまずは、全ての登録データを表示する画面、通称一覧画面を作っていきましょう。一覧画面は目次(index)ページという意味もあります。一覧画面は最初に表示されるページで、ここから他のページへリンクで移動します。

ルーティング

最初にプログラミングするのは、ルーティングです。ルーティングは、リクエストを解析して、どのように処理をすれば良いのかを決めるプログラムです。
ブラウザからやってくるリクエストは、URLとHTTPメソッドと呼ばれる 2 つの値のペアで構成されます。HTTPメソッドにはGET,POST,PATCH,DELETEの 4 つがあります。
アドレスバーにURLを入力した時や、google検索で出てきたサイトをクリックした時は、通常GETになります。会員登録やブログの投稿などはPOSTというHTTPメソッドを使っています。サイトの表示がGET(得る)で、データの送信がPOST(送信)というイメージです。
ではコードを書いていきます。

- config/routes.rb

```
Rails.application.routes.draw do
  get "/books", to: "books#index"
end
```

ルーティングの記述はこれだけです。実質 1 行追加するだけですね。一覧ページを得るのでGETを使っています。
このコードは、
`get "/books"` :「https://xx.com/books」というURLと、GETというHTTPメソッドがペアでリクエストされたら、

to: "books#index" : books_controllerのindexアクションを実行する
という意味になります。

- 書式

```
HTTPメソッド URLパターン, to: "コントローラクラス名#メソッド名"
```

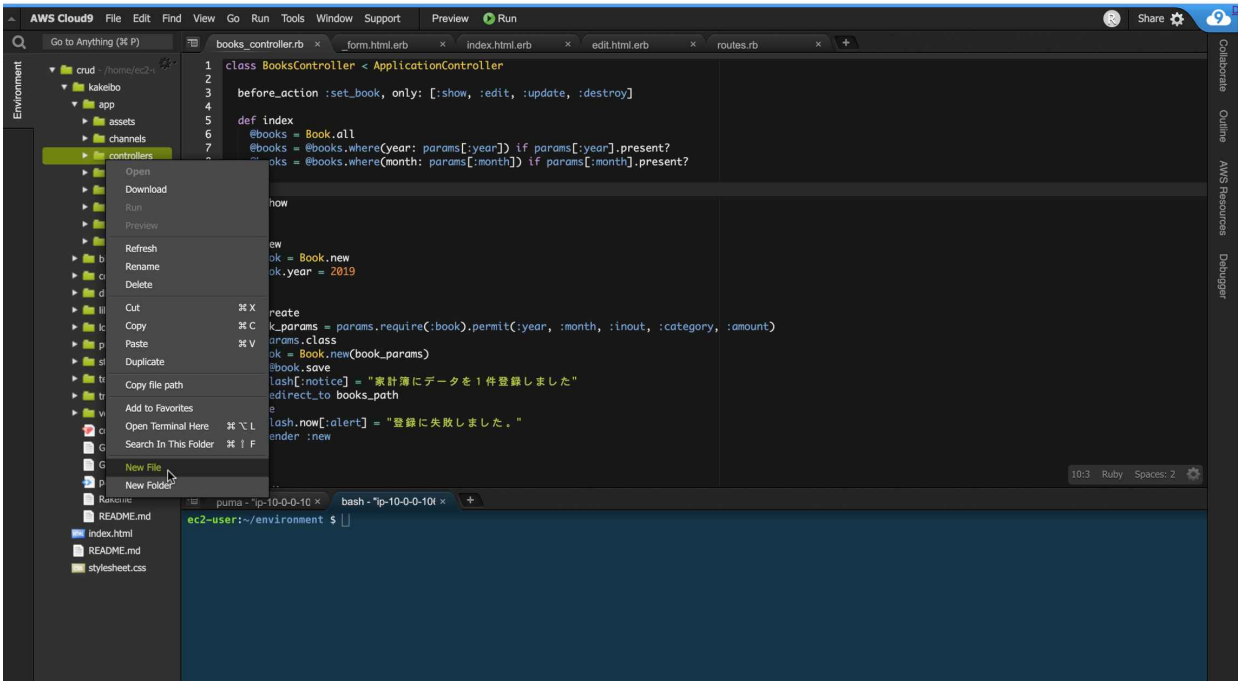
ルーティングはURL・HTTPメソッドのペアを読み取り、実行するコントローラ・メソッドを決めるだけです。
次に、コントローラを開発していきましょう。

コントローラ

コントローラは、処理の中枢です。主な機能はデータの準備と表示する画面(HTML)の指示ですが、表示する画面はrailsによってビューファイルから選択されるため、実質データの準備のみです。

コントローラを作成するには、モデルと同じように専用コマンドがあるのですが、本書では自分でファイルを作成します。app/controllers フォルダに books_controller.rbというファイルを作成してください。ファイル名のbookが複数形であることに注意です。

cloud9でファイルを作成するには、作成したい場所(フォルダ)の上で右クリック(または 2 本指でクリック)して、「New File」を選択します。空のファイルが作られるので、ファイル名を入力してください。



- app/controllers/books_controller.rb

```
class BooksController < ApplicationController

  def index
    @books = Book.all
  end

end
```

indexメソッドの中でBook.all```で家計簿データを全て取得(allメソッド)しています。Rubyのコードからデータベースへアクセスできるのは、ActiveRecordのおかげでしたね。
取得したデータは@booksという変数に入れています。@booksの中にはたくさんの家計簿データが配列のような形式で入っています。先頭に「@」をつけるのを忘れないでください。
コントローラの記述としては以上です。

ビュー

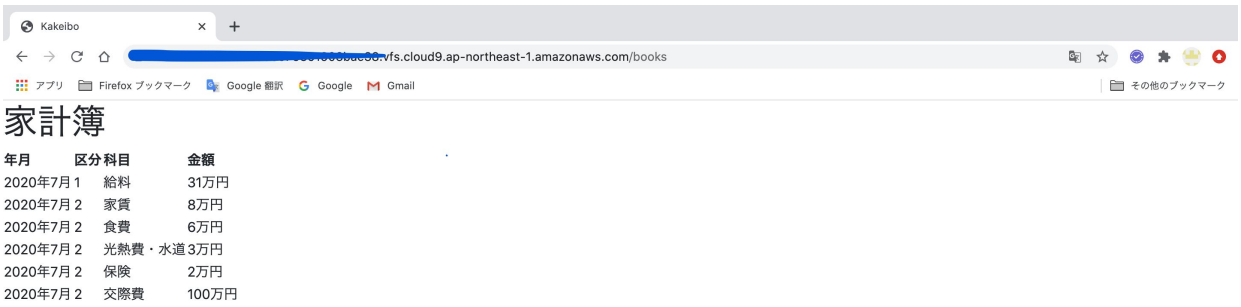
次に、コントローラで準備したデータを使って、HTMLファイルを組み立てていきます。データベースに保存されているデータをHTML形式として見やすく表現します。
Railsではerbというテンプレートエンジンを使います。テンプレートエンジンとは、簡単に言うと、RubyのコードをHTMLファイルの中で実行する仕組みです。
ビューはまずフォルダから作らなくてはいけません。app/viewsフォルダにbooksという名前のフォルダを作成してください。先ほどは「New File」からファイルを作成しましたが、ここでは「New Folder」でフォルダを作成します。
booksフォルダを作ったら、app/views/booksフォルダの中に、index.html.erbというファイルを作成します。

- app/views/books/index.html.erb

```
<h1>家計簿</h1>
<table>
  <tr>
    <th>年月</th>
    <th>区分</th>
    <th>科目</th>
    <th>金額</th>
  </tr>
  <% @books.each do |book| %>
  <tr>
```

```
    <td><%= book.year %>年<%= book.month %>月</td>
    <td><%= book.inout %></td>
    <td><%= book.category %></td>
    <td><%= book.amount %>万円</td>
  </tr>
  <% end %>
</table>
```

<% %>がRubyのコードを埋め込める部分です。この中でRubyのコードを埋め込むと、実行してくれます。
イコール付きの<%= %>は、ただ実行するだけでなく、実行した結果をHTMLファイルに表示することができます。
繰り返し処理である@books.each 自体は表示したくないので<% %>で囲んでいますが、book.yearは画面に表示したいので<%= %>で囲んでいます。
ここまでできたら、一通り動くので、試しに動かしてみましょう。
サーバを動かしていない場合は、「rails s」コマンドでサーバを起動しておきましょう。
先ほどの、ウェルカムページを「Preview」を使って表示してから、URLの末尾を「https://~.amazonaws.com/books」のように変えて、エンターキーでアクセスしてみてください。以下のように表示されたら成功です。



index.html.erbのコードを詳しく見ていきましょう。

まずは「each」から。eachは配列を繰り返し処理するメソッドでした。do~endの中に記載したコードが繰り返し処理する内容です。

eachでは、配列から一つずつ要素を取り出し、bookという変数へ代入します。そしてdo~endの中を処理すると、また次の要素をbookへ代入し、またdo~endの中身を処理します。配列の最後の要素にたどり着くまで繰り返します。

では、do~endの中身をみていきましょう。

```
<tr>
  <td><%= book.year %>年<%= book.month %>月</td>
  <td><%= book.inout %></td>
  <td><%= book.category %></td>
  <td><%= book.amount %>万円</td>
</tr>
```

trで囲まれていることから分かるように、これはテーブルの中の 1 行です。例えば「2020年7月 1 給料 32万円」のところを見てみると、HTMLに「2020年」を表示するためのコードが

```
<%= book.year %>年
```

となります。
bookには家計簿データ一つが入っています。そのデータに対して、.yearで年データを取り出しています。<%= %>で囲まれているので、取り出した値(2020)をHTMLファイルへ表示させています。
その結果、「2020年」と表示される、という流れです。
eachの中で繰り返しtrタグが作られるので、結果的に表示された画面には複数の家計簿データが縦に並びます。
たったこれだけのことですが、リクエストの解析、データベースからデータの取得、HTMLの動的生成など、割と凄いことをやっています。十分難しい内容ですので、あまり理解できていないと感じる場合は、もう一度、この章の先頭から読み直してみてください。

bootstrapの導入

さて、一覧画面は完成しましたが、ちょっと見栄えがよろしくないですね。
そこで、Bootstrapというものを導入して、簡単にカッコよくしてみましょう。
Bootstrapは、cssで装飾されたパーツが用意されたもので、cssを書かなくてもある程度見栄えを良くすることがきます。ちなみに、本書では一切cssを記述していません。bootstrapに頼れるだけ頼っています。
Bootstrapを使うためにまず変更するのは、app/views/layouts/application.html.erbというファイルです。このファイルはレイアウトファイルと呼ばれていて、画面共通のヘッダメニューなどを作る時に編集します。
先ほどのHTMLファイルはHTMLタグやBODYタグを書いていなかったですよね。それらがこのapplication.html.erbファイルに書かれています。画面を表示するときは、このファイルを土台として使っています。
Bootstrapを使うためには、このappplication.html.erbファイルへcssやjavascriptファイルの読み込みを記述します。元のファイルへ追加するコードはBootstrapのサイトに書かれているので、そこからコピーするのがいいかと思います。
https://getbootstrap.com/docs/4.5/getting-started/introduction/
このサイト内のCSSと書かれた場所に書いてあるコードをheadタグの中に、
JSに書かれているコードはbodyタグの`<%= yield %>`の次の行に追記します。

- app/views/layouts/application.html.erb

```
<!DOCTYPE html>
```

```
<html>
  <head>
    <title>Kakeibo</title>
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
                                        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.1/css/boo
tstrap.min.css" integrity="sha384-
VCmXjywReHh4PwowAiWNagnWcLhlEJLA5buUprzK8rxFgeH0kww/
aWY76TfkUoSX" crossorigin="anonymous">

      <%= stylesheet_link_tag 'application', media: 'all', 'data-
turbolinks-track': 'reload' %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track':
'reload' %>
  </head>

  <body>
    <%= yield %>
       <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"
integrity="sha384-
DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUe
w+OrCXaRkfj" crossorigin="anonymous"></script>
                                                    <script
```

```
src="https://cdn.jsdelivr.net/npm/popper.js@1.16.1/dist/umd/popper.min.js" integrity="sha384-9/reFTGAW83EW2RDu2S0VKaIzap3H66lZH81PoYlFhbGU+6BZp6G7niu735Sk7lN" crossorigin="anonymous"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.1/js/bootstrap.min.js" integrity="sha384-XEerZL0cuoUbHE4nZReLT7nx9gQrQreJekYhJD9WNWhH8nEW+0c5qq7aIo2Wl30J" crossorigin="anonymous"></script>
  </body>
</html>
```

この状態ではまだ読み込んだだけなので、bootstrapを使ってカッコ良くするための記述も書かなくてはいけません

- app/views/books/index.html.erb

```
<table class="table table-striped">
```

テーブルタグにクラスを追加してください。

この状態で、もう一度、アプリの画面を開き、更新してください。そうすると、テーブルが少しカッコよくなっていると思います。

家計簿

年月	区分	科目	金額
2020年7月	1	給料	32万円
2020年7月	2	家賃	8万円
2020年7月	2	食費	6万円
2020年7月	2	光熱費・水道	3万円
2020年7月	2	保険	2万円
2020年7月	1	副業	2万円

show画面の開発

一覧画面が完成したので、１つ分のデータを詳しく見る参照画面を作成しましょう。

全ての情報が一覧画面で見えているので、あまり意味がありませんが、業務アプリでは、何十という項目があり、一覧画面に表示する項目は限定的なのが普通です。そのため、全てのデータを見るために、参照画面というものが存在します。

ルーティング

まずはルーティングです。これを書かないと始まりません。

- config/routes.rb

```
Rails.application.routes.draw do
  get "/books", to: "books#index"
  get "/books/:id", to: "books#show", as: "book"
end
```

下から 2 行目が追加したコードです。showもページの参照なのでGETを使っています。
URLパターンが少し特殊ですね。:idというのは、具体的には決まっていないけど、何か値が入ることを表しています。
例えば、/books/1や/books/kakeiboのようなものが相当します。
そして、このURL(/books/:id)とHTTPメソッド(GET)をリクエストとして受け取ったら、books_controllerのshowメソッドを実行します。
後ろについているasはニックネームみたいなもので、後でリンクをつける時に役立ちます。

コントローラ

次にコントローラを開発しましょう。

- app/constrollers/books_controller.rb

```
class BooksController < ApplicationController

  def index
    @books = Book.all
  end

  def show
    @book = Book.find(params[:id])
  end

end
```

showメソッドを追加しました。URLパターンは「/books/:id」だったのですが、この「:id」に入った値を、コントローラの中ではparams[:id]というコードで受け取ることができます。「/books/6」というURLだと、params[:id]は6を返します。
もし/books/6というURLでアクセスされたら、以下の 2 つのコードは同じ意味になります

```
@book = Book.find(params[:id])
@book = Book.find(6)
```

このコードでは、findメソッドで、params[:id]と等しいidを持つデータをデータベースから検索しています。findメソッドは、見つかれば必ず 1 件だけ取ってくるので、単数系の変数@bookへ検索結果を代入しています。

ビュー

データの準備ができたので、そのデータを見ることができる画面を作っていきましょう。

app/views/booksフォルダへshow.html.erbというファイルを作成してください

- app/views/books/show.html.erb

```
<h1>家計簿詳細</h1>
<table class="table table-striped">
  <tr>
    <th>年月</th>
    <td><%= @book.year %>年<%= @book.month %>月</td>
  </tr>
  <tr>
    <th>区分</th>
    <td><%= @book.inout %></td>
  </tr>
  <tr>
    <th>科目</th>
    <td><%= @book.category %></td>
  </tr>
  <tr>
    <th>金額</th>
```

```
    <td><%= @book.amount %>万円</td>
  </tr>
</table>
```

showメソッドでは@bookという変数を用意しているので、ビューでも@bookという変数を使うことができます。＠マークはコントローラからビューへデータを受け渡しする時につける記号と覚えておいてください。

では、画面へアクセスしましょう。詳細画面はhttps://~.amazonaws.com/books/1というURLになります。indexの開発から進めてこられた方は、アドレスバーの末尾に/1を追記するだけです。

このように表示されればOKです。

コントローラでデータを準備して、ビューでそれを使うという流れはなんとなく理解できたでしょうか？

ビューファイルの中で、ただ単にRubyのコードを実行したいときは<% %>を使い、Rubyコードを実行し、さらにその結果をHTMLファイルへ表示したいときは<%= %>を使うのでした。

この画面で言うと、2020年7月や、1,給料, 32などがHTMLファイルへ表示されたRubyコードの実行結果です。

このようにHTMLの中でRubyのコードを使ったり、値を埋め込んだりできるのがテンプレートエンジンです。テンプレートエンジニにはerb以外もあるのですが(slimやhamlなど)、また機会があればご紹介します。本書中では登場しません。

リンク

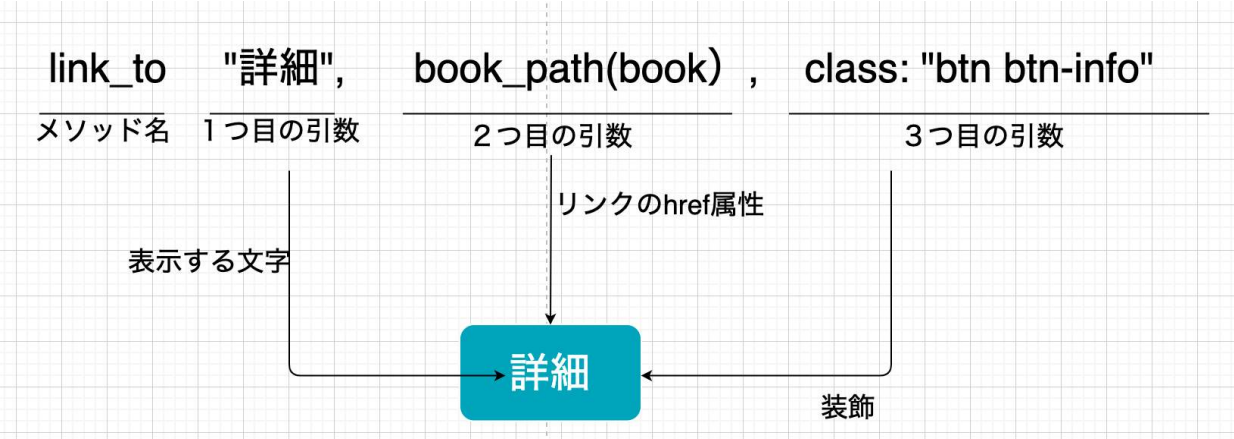
ここまでで、2 つの画面が完成しました。毎回ブラウザのアドレスバーを変えてアクセスするのはめんどくさいですし、現実的なアプリではないですよね。
そこで、2 つの画面へリンクを貼って、行き来できるようにしましょう。
一覧画面には参照画面へのリンク、参照画面には一覧画面へ戻るボタンをつけましょう。

- app/views/books/index.html.erb

```
<h1>家計簿</h1>
<table class="table table-striped">
  <tr>
    <th>年月</th>
    <th>区分</th>
    <th>科目</th>
    <th>金額</th>
    <th>リンク</th>
  </tr>
  <% @books.each do |book| %>
  <tr>
    <td><%= book.year %>年<%= book.month %>月</td>
    <td><%= book.inout %></td>
    <td><%= book.category %></td>
```

```
    <td><%= book.amount %>万円</td>
    <td>
      <%= link_to "詳細", book_path(book), class: "btn btn-info" %>
    </td>
  </tr>
  <% end %>
</table>
```

thタグとtdタグをそれぞれ１つずつ追加しています。
link_toメソッドはaタグによるリンクを作ってくれるメソッドです。
link_toメソッドに渡している２つ目の引数のbook_path(book)というのもメソッドで、/books/1のようなURLを表す文字列を返してくれます。
book_pathというメソッド名はどこからやってくるかというと、routes.rbで登録したルーティングのニックネーム(asで書いたやつ)+_pathという構造になっています。メソッドをどこかで定義するわけではなく、Railsがroutes.rbを読み、自動的に作ってくれるのです。
book_pathメソッドへインスタンスbookを渡すことで、自動的にidを取り出し、/books/2のようなURLを作ってくれます。
とりあえず、link_toでaタグを作るのだな～とぼんやり理解で問題ありません。



画面を更新してみてみると、次のようになっているはずです。

家計簿

年月	区分	科目	金額	リンク
2020年7月	1	給料	32万円	詳細
2020年7月	2	家賃	8万円	詳細
2020年7月	2	食費	6万円	詳細
2020年7月	2	光熱費・水道	3万円	詳細
2020年7月	2	保険	2万円	詳細
2020年7月	1	副業	2万円	詳細

詳細ボタンを押して、詳細画面へ飛べば成功です！

リンクが文字ではなく、いい感じのボタンになっていますが、これはbootstrapの装飾によるものです。btnとbtn-infoという 2 つのcssクラスをつけると、リンクをボタンのようにしてくれます。

次に、詳細画面へ戻るボタンを追加しましょう。ファイルをあっちへこっちへ行き来して大変ですが、頭の中に、ファイル群の構造をイメージできるまで耐えてください。

- app/views/books/show.html.erb

```
    <td><%= @book.amount %>万円</td>
  </tr>
</table>
<%= link_to "戻る", books_path, class: "btn btn-secondary" %>
```

ファイルの末尾に 1 行追加しています。ここもlink_toメソッドですね。
books_pathは一覧画面のurlを表しています。
このボタンは灰色になるはずです。画面を更新して、ちゃんと動くか確認してください。

new画面の開発

これまでは登録されているデータを見るだけでしたが、次はデータを登録できるようにしましょう。

データの登録処理は 2 段階に分けて行います。1 つ目はデータを入力するnew画面の表示、2 つ目はサーバへ送信されたデータをデータベースへ登録するcreateです。

ここではまず第一段階であるnew画面を実装していきます。

ルーティング

恒例のルーティングの時間です。

- config/routes.rb

```
Rails.application.routes.draw do
  get "/books", to: "books#index"
  get "/books/new", to: "books#new", as: "new_book"
  get "/books/:id", to: "books#show", as: "book"
end
```

上から 3 行目に新しく追記しました。
あれ？一番下の行に追加じゃないの？って思われるかもしれません。
ルーティングは、サーバへ送られてきたURLを、routes.rbに書かれた順に上から同じかどうかチェックします。
「/books/new」というリクエストが送られてきたら、最初はルーティングに書かれてある「/books」と同じかどうかチェックします。これは同じでは無いですよね。
次に、上から 2 番目の「/books/new」と同じかどうか見ます。これは同じなので、toに記載されている「books_controllerのnewアクション」で処理を行います。
しかし、「/books/new」の前に「/books/:id」が書かれているとどうでしょう。

「/books/:id」の:idは何でもいいという意味なので、「/books/new」というリクエストと同じと判断されてしまいます。なので、「/books/new」というルーティングと同じかどうかチェックする前に、「books_controllerのshowアクション」で処理されてしまいます。
そのため、ルーティングでは「/books/:id」より前に「/books/new」を記述しています。

コントローラ

続いてコントローラです。
新規登録画面を表示するためのルーティングはnewというアクションで行います。

- app/controllers/books_controller.rb

```
class BooksController < ApplicationController

  def index
    @books = Book.all
  end

  def show
    @book = Book.find(params[:id])
  end

  def new
    @book = Book.new
  end

end
```

newメソッドを追加しました。
コントローラでは、データを準備するのですが、新規作成なのでデータベースから取ってくるデータはありません。ですが、@book変数に空っぽの家計簿モデルのインスタンスを入れています。
なぜこんなことをしているかというと、それはビューファイルを作成してから説明します。

ビュー

また新しくファイルを作成します。app/views/booksへnew.html.erbというファイルを作成してください。

- app/views/books/new.html.erb

```
<h1>家計簿新規登録</h1>

<%= form_with model: @book, local: true do |f| %>

  <div class="form-group">
    <label>年度</label>
    <%= f.number_field :year, class: "form-control" %>
  </div>

  <div class="form-group">
    <label>月</label>
    <%= f.number_field :month, class: "form-control" %>
  </div>

  <div class="form-group">
    <label>区分</label>
      <%= f.select :inout, [["収入",1],["支出", 2]] ,{}, { class: "form-
```

```
control" } %>
  </div>

  <div class="form-group">
    <label>科目</label>
    <%= f.text_field :category, class: "form-control" %>
  </div>

  <div class="form-group">
    <label>金額</label>
    <%= f.number_field :amount, class: "form-control" %>
  </div>

  <%= f.submit "登録", class: "btn btn-primary"%>
<% end %>
```

ファイルの作成を終えたら、https://~.amazonaws.com/books/newへアクセスして、画面を確認しましょう。

このコードは説明する部分が多そうですね。

まず、form_withは、htmlのformタグを生成するためのメソッドです。「生成する」というのは少し難しい表現ですが、「置き換わる」と考えても問題ありません。

つまり、form_withのところに、<form action=~というhtmlのタグが置き換わります。

```
<%= form_with model: @book, local: true do |f| %>
…
<% end %>

 ↓

<form action="~">
…
</form>
```

このようにhtmlを生成するメソッドをビューヘルパーと呼んでいます。これまでにも出てきた、link_toも`<a href=~`というaタグに置き換わるので、ビューヘルパーの一つです。
form_withの引数としては、まずBookモデルのインスタンス@bookがあります。@bookをmodelというキーでform_withへ渡しています。ここへモデルを渡すために、コントローラのnewメソッド内で空っぽのインスタンスを準備していました。
インスタンスをform_withに渡すことにより、インスタンスに入っている値を表示してくれたりします。試しに、次の様にコントローラを編集してみてください。

```
def new
  @book = Book.new
  @book.year = 2019
end
```

画面をリロードして新規登録画面を表示すると、年度のところに2019が入力されていると思います。
本来、入力フォームに値を初期表示しようとするとvalue属性に値を書かないといけないのですが、ここら辺をform_withは自動化してくれています。他にも、formのaction属性(データの送信先)も自動的に生成してくれています。
もう一つ渡しているlocalオプションですが、これをtrueに設定しないとうまく動作しないことがあります。
この設定で非同期通信という機能をオフにしているのですが、少し応用的な内容になってくるので、ここでは詳しくは解説しません。興味のある方は、非同期通

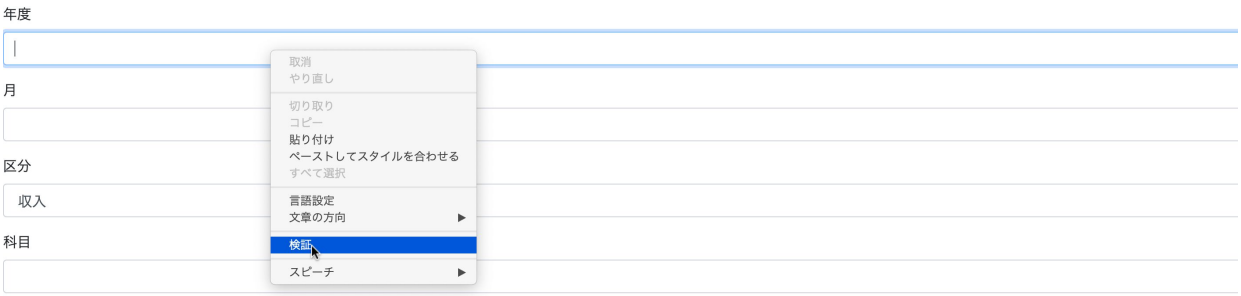
信が何か調べてみてください。

さて、次はform_withの中に入っているf.number_fieldですね。他にもf.がつくものはたくさんあります。これらはinputタグを生成するためのビューヘルパーです。fというのはform_withで作られる変数で、このfの中には@bookのデータも含まれています。fを通して@bookのデータがそれぞれのinputタグまで伝えられるため、先ほどの様に自動的に値が表示されたりします。

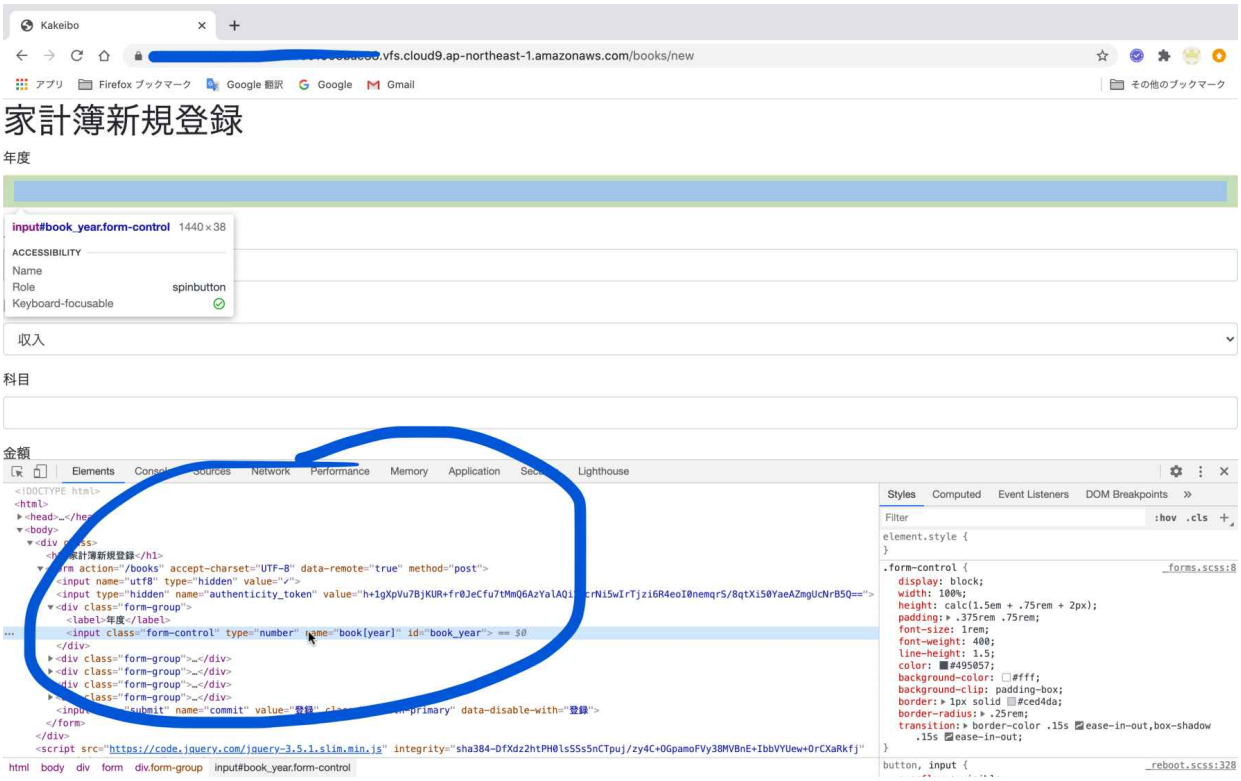
一つずつ細かく説明していくと大変なので、erbファイルのコードと、画面に表示されたウェブページのHTMLコードを照らし合わせて見て行く方法をご紹介します。

画面に表示されたウェブページのソースコードを見るには、ブラウザに搭載されている検証ツールというものを使います。

検証ツールは、ウェブページ上で、右クリック(Macなら 2 本指でクリック)し、出てきたメニューの「検証」または「開発者ツール」をクリックすると起動します。



画面上に現れたのが検証ツールです。



検証ツールでは、実際に今表示されている画面のHTMLコードをみることができます。このツールを使って、erbのコードと、対応する実際のHTMLタグを見比べて見ましょう。例えば、年度の入力部分だけ抜き出してみます。

- erb

```
<div class="form-group">
  <label>年度</label>
  <%= f.number_field :year, class: "form-control" %>
</div>
```

- html

```
<div class="form-group">
  <label>年度</label>
   <input class="form-control" type="number" name="book[year]"
id="book_year">
</div>
```

f.number_fieldは、typeがnumberのinputタグへ置き換わっているのが分かるかと思います。
nameやid属性も自動的につけてくれています。2019が初期表示されている場合は、value属性も含まれていると思います。
ビューの開発はとても複雑です。分からなくなったら、今書いているerbファイルが、画面上ではどのようなコードへ置き換わっているのか確認し、理解を深めていきましょう。
他のコードについても、どのように変化しているのか、一つ一つ見比べて見てください。
また、これらのコードは暗記する必要はありません。私も使うたびにgoogle検索しています。例えば「rails ビュー input number」などです。最初にrailsと入れておくのがコツです。
検索結果にはQiitaというエンジニア向けの記事投稿サイトがよく出てきますが、私もよく見ています。LGTM(いいね)が多くついているものは信用できますが、たまにコメントがたくさんついて炎上しているものもあります。炎上しているものやLGTMが少ないものは避けた方が無難です。

new画面開発の最後に、戻るボタンをつけておきましょう。登録しようとしたけど、やっぱり辞めたという場合に便利です。

- app/views/books/new.html.erb

```
<%= link_to "戻る", books_path, class: "btn btn-secondary" %>
<%= f.submit "登録", class: "btn btn-primary"%>
```

登録ボタンの直前に入れておきましょう。

createの開発

登録画面は完成しましたが、今のままでは登録ボタンを押しても動かないはずです。
この章では、入力されたデータをデータベースへ登録する処理を実装していきます。

ルーティング

まずはルーティングですね。ここで、初めてGET以外のHTTPメソッドが登場します。それはPOSTです。POSTとGETの違いには、機能的な面と意味的な面の２つ大きな違いがあります。
GETは情報の取得(get)であったのに対し、POSTは情報の送信を意味します。ちょうどGETと真逆ですね。これが意味的な違いです。通常、リンクをクリックしたり、アドレスバーへURLを入力し、エンターキーを押す操作は自動的にGETになります。
機能的な面では、GETはURLがリクエスト内容全体を表すのに対して、POSTでは、URLとは別に、私たちの見えないところでPOSTデータというものを一緒に送信しています。POSTデータの中身は、フォームなどで入力した値です。
それでは、ルーティングを書いていきましょう。

- config/routes.rb

```
Rails.application.routes.draw do
  get "/books", to: "books#index"
  post "/books", to: "books#create"
  get "/books/new", to: "books#new", as: "new_book"
  get "/books/:id", to: "books#show", as: "book"
end
```

3行目に追記しています。/booksというリクエストURLは、/books/:idというルーティングとは異なるので、最終行に追加してもいいのですが、同じリクエストURLを近くに並べた方が綺麗という理由で 3 行目に追加しています。

コントローラ

books_controller.rbへ次のcreateメソッドを追加します。追加する場所はnewメソッドのすぐ下です。

- app/controllers/books_controller.rb

```
  def create
        book_params = params.require(:book).permit(:year, :month, :inout, :category, :amount)
    @book = Book.new(book_params)
    @book.save
    redirect_to books_path
  end
```

コントローラもいよいよ複雑になってきましたね。
最初の行は、ストロングパラメータというRailsの機能です。
リクエストに含まれる全てのパラメータ(POSTデータや、urlに含まれるidなど)はparamsという変数に格納されます。showアクションでもparamsは登場しましたね。この時はidが格納されていました。
全てのデータがparamsにまとめられるので、paramsにはとにかくたくさんのデータが入っています。
そこで、ストロングパラメータを使って、登録に必要なデータだけを取り出します。

```
book_params = params.require(:book).permit(:year, :month, :inout, :category, :amount)
```

ストロングパラメータには、セキュリティ対策的な役割もあるのですが、ここでは「必要なデータを取り出す」と理解していただければ問題ありません。
最終的に、book_paramsには画面上で入力したデータだけが入っています。
次に、

```
@book = Book.new(user_params)
```

ですが、これはおそらく問題ないかと思います。book_paramsには、画面から入力したデータが入っており、このデータを元にBookモデルを新しくインスタンス化しています。こうすることで、データ入りのbookインスタンスが出来上がります。
この時点ではインスタンスがデータを持っているだけの状態です。これらのデータをデータベースへ保存するために、saveメソッドを使います。

```
@book.save
```

最後に、

```
redirect_to books_path
```

というコードで、books_path(一覧画面)へ移りなさいという「リダイレクト命令」をブラウザへ返却しています。
リダイレクト命令を受け取ったブラウザは、リダイレクト命令に記載してあるURL(ここでは一覧画面のURL)へ自動的にリクエストを送信しています。
少し面倒な手順をとりますが、リダイレクトを利用することで、登録完了したら自動的に一覧画面に戻るという動きを実現できます。
ここまでで登録はできるので、画面から何か入力して、データを登録してみてください。

バリデーション

登録処理は実装できましたが、現状だと、何も入力しなくても登録ができてしまいます。これだと間違って登録ボタンを押してしまうと、無駄なデータができてしまいますね。
そこで、Railsで使える検証機能(バリデーション)を使って、入力していない項目があった場合に、登録させないという処理を書きます。ウェブアプリを作る上では必須の機能です。
編集するファイルはモデルです。久しぶりの登場ですね。

- app/models/book.rb

```
class Book < ApplicationRecord

  validates :year, presence: true
  validates :month, presence: true
  validates :inout, presence: true
  validates :category, presence: true
  validates :amount, presence: true

end
```

バリデーションの書き方は

```
validates :カラム名, ルール, ルール,...
```

になっています。今回はpresence(入力必須にする)しか書いていませんが、複数のルールをつけることもできます。
バリデーションはカラム毎に記述します。
バリデーションは、saveやcreateなど、データベースへ保存しようとするタイミングで、ルールの検証を行います。検証に失敗する(ルール違反がある)と、@book.saveはfalseを返します。逆に登録成功すると、@book.saveはtrueを返します。
失敗した場合は、どこへも行かずに、新規登録画面のままであって欲しいですよね。現状では検証に失敗しても成功してもリダイレクトしてしまうので、成功と失敗で処理を分けたいです。
ちょうど、saveメソッドがtrueとfalseを返すので、ifを使って条件分岐できそうです。
コントローラを修正しましょう。

- app/controllers/books_controller.rb

```
  def create
        book_params = params.require(:book).permit(:year, :month, :inout, :category, :amount)
    @book = Book.new(book_params)
```

```
    if @book.save
      redirect_to books_path
    else
      render :new
    end
  end
```

ifで、@book.saveが成功(true)の時はリダイレクトし、そうでない時は、render?しています。
render :newは何をしているのでしょう？
renderは表示するビューファイルを指示するメソッドです。今まで、Railsが自動的にアクションと同じ名前のビューファイルを表示していたので、出てきませんでした。newメソッドならnew.html.erbという具合ですね。
登録画面をもう一度表示したいなら、登録画面を表示するという指示をrenderメソッドを使って出します。newメソッドへ移動するわけではありません。
ここまでできたら、登録画面をブラウザで表示して、何も入力しないパターン、全て入力したパターンの２つを実際に動かして挙動を確認してください。
何も入力しない方は、一見動かないように見えますが、実は新しくHTMLが送られてきているのです。ですが、ボタンを押す前の画面と全く同じなので、変わらないように見えます。
最後に、一覧画面に新規登録画面へのリンクをつけておきましょう。できる方は、何も見ずに、チャレンジしてください。

- app/views/books/index.html.erb

```
<h1>家計簿</h1>
<%= link_to "+新規", new_book_path, class: "btn btn-success" %>
```

new_bookというのがroutes.rbで定義した新規画面のURLのニックネームでした。

```
get "/books/new", to: "books#new", as: "new_book"
```

このニックネームに_pathをつけることでURLを返すメソッドへと生まれ変わります。
ビューファイルに表示してみるとわかりやすいです。

```
<h1>家計簿</h1>
<%= new_book_path %>
<%= link_to "+新規", new_book_path, class: "btn btn-success" %>
```

edit画面の開発

登録ができたので、次は登録したデータを更新する画面を作っていきましょう。
教科書はどんどん進んでいきますが、どんなに賢い人でも、すんなりと理解できるものではありません。よくわからないところは放置せずに何度も読み返してみてください。それでもダメなら、とりあえず進んでOKです！この加減はちょっと難しいですね。
この章では編集画面を作成します。新規登録と同じように、編集画面とデータベースへの反映の 2 段かいに別れます。更新処理は次のupdateの章で解説します。
ちなみに、このような入門書って実務のコードと全然違うんじゃないの？と思われるかもしれません。ですが、ここで書いたコードは実務とそこまで遠くないコードになっています。「アプリをより良くする」の章で、もっと実務に近いコードへ修正していきます。
あと、ルーティングを書いて、コントローラを書いて、ビューを書くという流れは実際の実務で著者が行っている作業の流れです。大袈裟に言えば、著者が実装するのを皆さんには追体験してもらっています。

ルーティング

ルーティングは以下のようになります。

- config/routes.rb

```
Rails.application.routes.draw do
  get "/books", to: "books#index"
  post "/books", to: "books#create"
  get "/books/new", to: "books#new", as: "new_book"
  get "/books/:id/edit", to: "books#edit", as: "edit_book"
  get "/books/:id", to: "books#show", as: "book"
end
```

上から 5 行目が追加したコードです。https://~.com/books/2/editのようなURLが相当します。このURLのニックネームはedit_bookとしています。/books/:id/editというURLだと一番最後に記述しても問題ないのですが、後々のことを考えてこの場所に記述しています。本書の最後の方でわかっていただけると思うので、今はモヤモヤするかもしれませんが、上記のように記述しておいてください。

コントローラ

次にコントローラですね。

- app/controllers/books_controller.rb

```
def edit
  @book = Book.find(params[:id])
end
```

createメソッドの下に追記しました。中身はshowと一緒ですね。ビューでは今現在登録されているデータを見せたいので、データベースからデータを取得して、@bookという変数へ入れています。

ビュー

次はビューですが、こちらはほとんど登録画面と同じです。新しくedit.html.erbというファイルを作成してください。

- edit.html.erb

```
<h1>家計簿更新</h1>

<%= form_with model: @book, local: true, method: "patch" do |f| %>

  <div class="form-group">
    <label>年度</label>
    <%= f.number_field :year, class: "form-control" %>
  </div>

  <div class="form-group">
    <label>月</label>
    <%= f.number_field :month, class: "form-control" %>
  </div>

  <div class="form-group">
    <label>区分</label>
```

```
      <%= f.select :inout, [["収入",1],["支出", 2]] ,{}, { class: "form-control" } %>
  </div>

  <div class="form-group">
    <label>科目</label>
    <%= f.text_field :category, class: "form-control" %>
  </div>

  <div class="form-group">
    <label>金額</label>
    <%= f.number_field :amount, class: "form-control" %>
  </div>

  <%= f.submit "更新", class: "btn btn-primary"%>
<% end %>
```

newと異なる部分が３箇所あります。
１つはh1タグの中の文字、もう１つはf.submitに表示している文字、最後の１つは、form_withにmethodという項目を追加しています。
データの送信は通常POSTで行うのですが、更新に関しては、更新専用のPATCHというHTTPメソッドを使用します。何も指定せずにフォームを作ると、通

信はPOSTになってしまうので、methodというキーワードでpatchを指定しています。
PATCHはデータの更新を行うためのHTTPメソッドになります。
まだ更新はできませんが、画面を見ることはできるので、試しに編集画面を確認してください。https://~.com/books/1/editというURLです。

updateの開発

開発も終盤に差し掛かってきました。残すところあと、updateとdestroyのみです。

updateも基本的にはcreateと同じです。

ルーティング

editでフォームを作るときにサラッと出てきましたが、データの更新はPATCHで行います。

- config/routes.rb

```
Rails.application.routes.draw do
  get "/books", to: "books#index"
  post "/books", to: "books#create"
  get "/books/new", to: "books#new", as: "new_book"
  get "/books/:id/edit", to: "books#edit", as: "edit_book"
  get "/books/:id", to: "books#show", as: "book"
  patch "/books/:id", to: "books#update"
end
```

こちらは最終行へ追加します。ニックネームを書いていないですが、書いていない場合は、すぐ上のニックネーム(book)を引き継ぎます。showと同じニックネーム・URLパターンになってしまいますが、HTTPメソッドが異なるのでちゃんと区別されます。

コントローラ

続いてコントローラ開発です。

- app/controllers/books_controller.rb

editメソッドの下にupdateメソッドを記述します。

```
  def update
    @book = Book.find(params[:id])
        book_params  =  params.require(:book).permit(:year,  :month,
:inout, :category, :amount)
    if @book.update(book_params)
      redirect_to books_path
    else
      render :edit
    end
  end
```

@book.update以外はすでに登場したものばかりですね。
２行目はfindメソッドにより、idで検索しています。
３行目はストロングパラメータです。ここで必要なカラムのみ取り出しています。
4行目で、updateメソッドを使い、データベースへ更新を行なっています。updateメソッドもデータベースへ保存するメソッドなので、バリデーションのルール検証を行

います。saveと同じく、検証に失敗すればfalse、成功すればtrueが返ってきます。
更新に成功すると、一覧画面へリダイレクトしています。一覧画面でなく、次のように書くと、更新した家計簿の詳細画面へリダイレクトすることもできます。

```
redirect_to book_path(@book)
```

登録に失敗(どのかの項目を空欄の状態で送信し、バリデーションルールに反した場合)した時は、renderメソッドにより、表示する画面をedit.html.erbに指定しています。
updateはcreateと同様に、新しく作るビューファイルは無いので、この時点で動かすことができます。
編集画面(https://~.amazonaws.com/books/2/edit)へアクセスして、どこかに未入力項目がある場合と、全ての項目に値を入力した場合で挙動を確認してください。

リンク

最後に、編集画面へのリンクもつけておきましょう。

- app/views/books/index.html.erb

```
  <td>
    <%= link_to "詳細", book_path(book), class: "btn btn-info" %>
      <%= link_to "編集", edit_book_path(book), class: "btn btn-warning" %>
  </td>
```

詳細ボタンのすぐ下の行にリンクを追加しています。btn-infoは青色のボタンでしたが、btn-warningは黄色いボタンになります。
編集画面には戻るボタンをつけておきましょう

- app/views/books/edit.html.erb

```
<%= link_to "戻る", books_path, class: "btn btn-secondary" %>
<%= f.submit "更新", class: "btn btn-primary"%>
```

更新ボタンの上に記述します。戻るボタンの色は灰色です。

destroyの開発

いよいよ最後の開発となりました。最後はこれまで登録してきたデータを削除する処理を実装します。間違えて登録したデータを削除できないと困りますよね。

ルーティング

削除はDELETEというHTTPメソッドを使います。

- config/routes.rb

```
Rails.application.routes.draw do
  get "/books", to: "books#index"
  post "/books", to: "books#create"
  get "/books/new", to: "books#new", as: "new_book"
  get "/books/:id/edit", to: "books#edit", as: "edit_book"
  get "/books/:id", to: "books#show", as: "book"
  patch "/books/:id", to: "books#update"
  delete "/books/:id", to: "books#destroy"
end
```

削除のルーティングも最終行へ追加します。

コントローラ

削除処理を実装します

- app/controllers/books_controller.rb

```
def destroy
  @book = Book.find(params[:id])
  @book.destroy
  redirect_to books_path
end
```

これまでのcreateやupdateに比べると、比較的直感的に理解できるのではないでしょうか。
２行目がfindでデータを取得するコードです。
３行目のdestroyメソッドによって、データベースからデータを削除します。
最後に、一覧画面へリダイレクトしています。
登録や更新ではバリデーションにより失敗する可能性があったのでifで分岐していましたが、destroyはバリデーションチェックが入らないので、失敗す要因がありません。なのですぐにリダイレクトしています。

リンク

一覧画面へ削除リンクを追加しておきましょう。

- app/views/books/index.html.erb

```
    <td>
      <%= link_to "詳細", book_path(book), class: "btn btn-info" %>
        <%= link_to "編集", edit_book_path(book), class: "btn btn-warning" %>
        <%= link_to "削除", book_path(book), method: "delete", class: "btn btn-danger" %>
    </td>
```

ボタンの並びの一番最後に追加しました。
注意するのは、link_toの中でmethodを指定している部分です。これを忘れてしまうと、ただの詳細画面へのリンクになってしまいます。
これで実際に動くか、確かめて見てください。
おそらく問題なく削除されると思うのですが、ボタンを押したらすぐに消えるのって少し怖いですよね。間違ってデータを消しかねないです。
そこで、削除ボタンを押したら、「本当に削除しますか？」という注意を出してみましょう。
やり方はすごく簡単です。リンクを以下のように修正してください

```
<%= link_to "削除", book_path(book), method: "delete", data: { confirm: '本当に削除しますか？' }, class: "btn btn-danger" %>
```

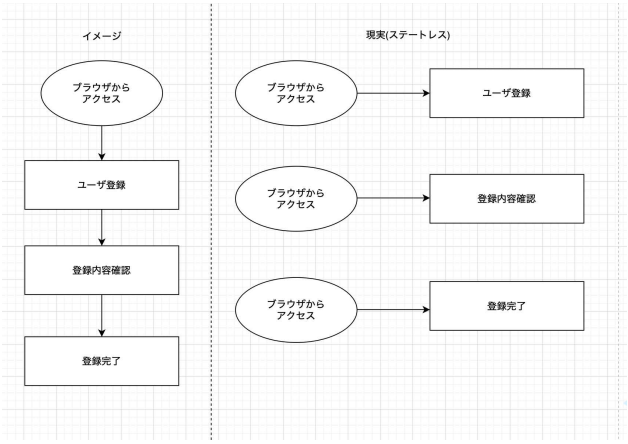
data: { confirm: '本当に削除しますか？' }という記述を追加しました。Railsでは、この１行を書くだけで、確認用の画面を表示してくれます。
こちらも実際に動かして確認してください。
お疲れ様でした！これで、家計簿モデルに対するCRUDを全て実装でき、簡単ですが家計簿アプリが出来上がりました。
本書の目標としては、ひとまずここまでの内容をぜひ押さえておいていただければ幸いです。
しかし、完成度としては70％といったところでしょうか。この後、もう少し開発は続きます。本書の最後の実装は少し難易度の高いものになっていますが、RPGでいうところの裏ボスに挑む感覚で、最後までチャレンジしていただければと思います。

【実践編】アプリをより良くしていこう

現時点で、ひとまず動く状態になっていると思います。
ただ、登録成功・失敗した後に何もメッセージが出ないので、どうなったか分かりづらかったり、実装面で言うと、同じコードが何箇所にも書かれている場所があります。
ここからはアプリをより良くするための実装となります。レベル感としては、業務で実装するレベルの内容になっています。
仕事で「CRUD作っておいて」と言われたら、実践編の内容を自力で実装する必要があります。
まだ基礎編の内容がイマイチという方は、もう一度基礎編の最初からやり直してみてください。

フラッシュメッセージ

改善の第一歩目は、登録成功・失敗したときに、画面にメッセージを表示するようにしましょう。現状だと、何もメッセージが表示されないので、登録に成功したのか、失敗したのかが分かりにくい状態になっています。
メッセージの表示には、flashという機能を使うのですが、flashを理解するためには、ステートレスという言葉を知っておかなければなりません。Railsはステートレスだからです。
具体例として、あるユーザー登録を考えてみましょう。
私たちは、ユーザー登録画面から必要な項目を入力した後、送信ボタンを押し、次の確認画面へ遷移します。確認画面で登録ボタンを押すと、登録処理が実行され、最後に完了画面が表示されます。
このように、私たちが登録した情報を覚えていてくれて、一連の流れに沿って画面が移り変わっていくように見えます。
しかし、実際は、サーバは私たちのことを覚えておらず、毎回新規のリクエストをサーバに向けて送っています。毎回新規のリクエストとして処理する方式をステートレスと言います。



アプリ作成者は、スムーズに登録ができるように、ボタンや隠し入力(typeがhiddenのinputタグ)、データベースによる一時的なデータ保存などを巧みに使って、私たちのことを覚えているように見せかけます。
図の右側を見ていただけると良く分かるのですが、ユーザ登録から登録内容確認画面へは直接つながっている訳ではなく、画面から画面へ何か情報を送るのは、実は意外と難しいのです。
そこで登場するのがflashという機能です。flashを使うと、次のリクエストまで一時的に情報を覚えていてくれます。映画「博士の愛した数式」の博士のように、体にメモを貼っているような感じです。すぐに剥がれるので、次の次のリクエストにはメモは無くなっています。
では実際にflashを使ってメッセージを表示して見ましょう。
主に 2 箇所修正します。1 つはflashにデータを登録するコントローラでの処理と、もう 1 つはflashに登録したデータを表示するビューでの処理です。
まずは登録部から実装しましょう。

- app/controllers/books_controller.rb

```
 def create
      book_params = params.require(:book).permit(:year, :month, :inout, :category, :amount)
   @book = Book.new(book_params)
   if @book.save
     flash[:notice] = "家計簿にデータを 1 件登録しました"
     redirect_to books_path
```

```
  else
    flash.now[:alert] = "登録に失敗しました。"
    render :new
  end
end
```

上から5行目と、下から4行目に変更を加えています。
登録に成功した場合は、noticeというキーワードで、メッセージを登録しています。
失敗した場合には、flashへalertというキーワードでメッセージを登録します。失敗した場合はリダイレクトしないので、わざわざflashを使う必要もないのですが、一時的なメッセージということでflashを利用しています。
noticeやalertというキーワードはある程度固定されていて、実務でもこの2つを使っています。
次に表示部を実装しましょう。
flashメッセージはcreateの他にupdateやdestroyメソッドからも発生します。どの画面で表示されるかをいちいち考えるのは面倒なので、全画面共通のレイアウトファイル(application.html.erb)にコードを書いてしまいましょう。

- app/views/layouts/application.html.erb

```
<body>
  <% if flash[:notice] %>
    <div class="alert alert-success" role="alert">
```

```
    <%= flash[:notice] %>
  </div>
<% end %>

<% if flash[:alert] %>
  <div class="alert alert-danger" role="alert">
    <%= flash[:alert] %>
  </div>
<% end %>
<%= yield %>
```

bodyの開始タグと、yieldの間にコードを追加します。
flashにnoticeというキーワードでメッセージが登録されていれば表示するものと、alertというキーワードでメッセージが登録されていたら表示するものの 2 つあります。
このように、ifを使ってタグ自体を表示するしないを決定することもできます。
もし、flashにnoticeというキーワードでメッセージが登録されていなければ、alert-successというクラスがついている側のdivタグは画面に表示されません。
試しに、新規登録画面で、登録失敗するパターンと成功するパターンで、どのような表示になるかを確認して見てください。
残り、updateとdestroyにもメッセージ登録用コードを追加しましょう。

- app/controllers/books_controller.rb

```
  def update
    @book = Book.find(params[:id])
        book_params  =  params.require(:book).permit(:year,  :month,
:inout, :category, :amount)
    if @book.update(book_params)
      flash[:notice] = "データを１件更新しました"
      redirect_to books_path
    else
      flash.now[:alert] = "更新に失敗しました。"
      render :edit
    end
  end

  def destroy
    @book = Book.find(params[:id])
    @book.destroy
    flash[:notice] = "削除しました"
    redirect_to books_path
  end
```

コードが多いですが、どこか分かるでしょうか？ 3 箇所コードを追加しています。flashという文字を手がかりに見つけてください。

編集ができたら、画面を実際に動かし、確認してください。
コントローラで登場したflashには２種類あったのが分かったでしょうか？
flashとflash.nowです。flashは次のリクエストまでメッセージを残す機能があるのですが、flash.nowは次のリクエストまでメッセージを残さず、「今」表示しないと消えてしまうものです。
flash.nowが使われているのは、renderメソッドのすぐ上なので、リダイレクトせずに画面に表示されます。
しかし、ビューファイルにflash[:alert]の表示コードがなかった時（今回はapplication.html.erbへ記述しているのでその心配はないですが)、flash.nowではなく、flashを使っていると、全く別の画面へ遷移したときに、flashメッセージが表示されてしまいます。flashは次のリクエストまでメッセージを残しているためです。
なので、リダイレクトせずに、現在のリクエストに対する画面にだけメッセージを表示したいときはflash.nowを使います。
つまり、redirect_toの時はflashを、renderの時はflash.nowを使うということです、

レイアウトを少し整える

メッセージが出るようになり、それらしくなってきました。
ここで、少しレイアウトを整えたいと思います。現状、画面の左右にぴったりとくっついていますが、本体を細身にして、最近のサイトの画面っぽくします。
この修正も全ての画面に適用したいので、application.html.erbへ記述します。

- app/views/layouts/application.html.erb

```
<body>
  <div class="container">
    <% if flash[:notice] %>
      <div class="alert alert-success" role="alert">
        <%= flash[:notice] %>
      </div>
    <% end %>

    <% if flash[:alert] %>
      <div class="alert alert-danger" role="alert">
        <%= flash[:alert] %>
      </div>
    <% end %>
    <%= yield %>
```

```
    </div>
```

フラッシュの表示部分からyieldまでをcontainerというクラスを持つdivで囲みます。これだけで画面がグッと見やすくなります。
containerはbootstrapが用意してくれているクラスで、幅を少し細くし、中央に配置してくれます。

before_action

ここからは、見た目や機能に変化がないコードの変更です。このように、見た目や機能に変化を起こさずに、コードをより良いものにすることを、「リファクタリング」と言います。
コントローラのshow, edit, update, destroyを見てください。全く同じコードがありますね。これを一つのメソッドとしてまとめて、尚且つ、自動的に実行するようにしたいと思います。

- app/controllers/books_controller.rb

```
def destroy
  @book.destroy
  flash[:notice] = "削除しました"
  redirect_to books_path
end

private

def set_book
  @book = Book.find(params[:id])
end
```

destroyメソッドの下にコードを追加しています。privateというのは、このコントローラの中でしか使いませんという目印のようなものです。
同時に、show,edit,update,destroyから以下のコードを削除してください

```
@book = Book.find(params[:id])
```

ここまでで、メソッドを新しく作り、複数の箇所で書かれていた同じコードを削除しました。
メソッドは作るだけでなく、使わないと意味がないのですが、コードを削除した代わりに、

```
def edit
  set_book
end
```

のようなコードを書いても、少し短くなっただけで、あまり意味はありませんね。
そこで、Railsに備わっているbefore_actionと呼ばれる機能を使います。
before_actionは、メソッドの実行前に予め決められたメソッドを実行しておいてくれる機能です。

- app/controllers/books_controller.rb

```
class BooksController < ApplicationController

  before_action :set_book, only: [:show, :edit, :update, :destroy]

  def index
```

ファイルの 2 行目にコードを追加します。before_actionの基本的な構文は次のようになっています

```
before_action :実行するメソッド名
```

before_actionは基本的に全てのコントローラ内のアクションの前に実行してしまうので、onlyというオプションで実行するアクションを制限しておきます。

resourcesメソッド

次にご紹介するのは、resourcesメソッドです。
routes.rbでは、１つのアクションに対して、１行追加してきました。
ですが、アプリがどんどん大きくなってくると、アプリの機能の数だけ行を追加しなくてはいけません。これは結構大変です。そこで、自動的にルーティングを生成してくれる、resourcesメソッドを使います。
その前に、現在のルーティングを確認して見ましょう。routes.rbファイルを見てみるのもいいのですが、ターミナルへ

```
$ rails routes
```

というコマンドを打つことで、Railsが現在認識しているルーティングを確認することができます。以下は今回作成したCRUDのルーティングのみを抜き出しています。また、スペースがかなり長いので削除しています。

```
    books GET    /books(.:format)          books#index
          POST   /books(.:format)          books#create
 new_book GET    /books/new(.:format)      books#new
edit_book GET    /books/:id/edit(.:format) books#edit
     book GET    /books/:id(.:format)      books#show
          PATCH  /books/:id(.:format)      books#update
```

```
        DELETE /books/:id(.:format)       books#destroy
```

これも見方が難しいのですが、慣れると分かりやすいツールです。
１行目を見てみましょう。

```
books GET    /books(.:format)          books#index
```

booksはURLのニックネームです。その後ろにHTTPメソッドが書かれています。
/booksというのがURLですね。
(.:format)というのは、/books.htmlとしてもいいよという意味です。htmlがフォーマットです。
最後に、books#indexというのは、books_controllerのindexアクション(メソッド)を意味しています。
つまり、「/booksというURLがHTTPメソッドGETでリクエストされたら、books_controllerのindexアクションを実行する」という意味になります。
もう一つ見てみましょう。

```
PATCH  /books/:id(.:format)       books#update
```

こちらはニックネームが書かれていないように見えますが、すぐ上にbookと書かれています。同じなので省略されているだけです。
リクエストする時、:idには具体的な数字が入るので、「/book/4というURLがHTTPメソッドPATCHでリクエストされたら、books_controllerのupdateアクションを実行する」という意味になります。
今、routesファイルには 7 行のルーティングが書かれていますが、この 7 つを 1 行で自動生成してくれるのが、resourcesメソッドです。今書かれている 7 行を削除するかコメントアウトして、

- config/routes.rb

```
Rails.application.routes.draw do
  resources :books
end
```

と変更してください。
変更できたら、ファイルを保存して、また$ rails routesコマンドでルーティングを確認してください。

```
    books GET    /books(.:format)          books#index
          POST   /books(.:format)          books#create
```

```
 new_book GET    /books/new(.:format)       books#new
edit_book GET    /books/:id/edit(.:format)  books#edit
     book GET    /books/:id(.:format)       books#show
          PATCH  /books/:id(.:format)       books#update
          PUT    /books/:id(.:format)       books#update
          DELETE /books/:id(.:format)       books#destroy
```

updateに対するルーティングが 2 つありますが、putの方は今は使われていない古い書き方になります。putを無視すれば、resourcesを使う前と全く同じになると思います。
/books/:id/editのルーティングを書いた時に、中途半端な位置に挿入しましたが、それはresourcesと同じ並びになるようにするためにあのような順番にしていました。

パーシャル

最後にご紹介するのはパーシャルです。
家計簿テーブルにカラムが追加されたり、新規登録画面の区分をプルダウンでの選択から、ラジオボタンに変えたり、フォームに変更を加えようと思うと、「new.html.erb」と「edit.html.erb」の２つのファイルを修正しなくてはいけません。これでは変更を加える際に、片方だけ変更して、もう片方は変更するのを忘れたなどのミスが起きてしまいます。
そこでRailsには、パーシャルという機能が備わっています。
２つのビューファイルの共通の部分を抜き出し、別ファイルとして保存しておきます。
この共通ファイルをビューからパーシャル機能を使って呼び出すことで、同じコードを何度も書く必要がなくなるのです。クラスにおける継承と少し似ていますね。
まずは共通部分を抜き出して、ファイルへ保存しましょう。ここで注意なのですが、パーシャルから呼び出すファイル名は必ず_(アンダーバー)から始まるようにします。
ただ単に抜き出しているだけでなく、少し変更してある部分もあるので、注意して記述してください。

- app/views/books/_form.html.erb

```
<h1>家計簿<%= type %></h1>

<%= form_with model: book, local: true, method: method do |f|
```

```
%>

  <div class="form-group">
    <label>年度</label>
    <%= f.number_field :year, class: "form-control" %>
  </div>

  <div class="form-group">
    <label>月</label>
    <%= f.number_field :month, class: "form-control" %>
  </div>

  <div class="form-group">
    <label>区分</label>
      <%= f.select :inout, [["収入",1],["支出", 2]] ,{}, { class: "form-
control" } %>
  </div>

  <div class="form-group">
    <label>科目</label>
    <%= f.text_field :category, class: "form-control" %>
  </div>

  <div class="form-group">
```

```
    <label>金額</label>
    <%= f.number_field :amount, class: "form-control" %>
  </div>

  <%= link_to "戻る", books_path, class: "btn btn-secondary" %>
  <%= f.submit type, class: "btn btn-primary"%>
<% end %>
```

edit.html.erbから変更箇所が４箇所あります。
まず、h1タグとf.submitで更新という文字をtypeという「変数」で置き換えています。変数は呼び出す側(new.html.erbやedit.html.erb)で準備します。ここで２箇所です。
３箇所目は、form_withのmodelへ渡している値が@bookからbookへ変更しています。
４箇所目が、同じくform_withのmethodへ渡している値です。"patch"からmethodへ変更しています。
ここで、３つの変数が新しく登場しました。type, book, methodです。この３つは、edit.html.erbとnew.html.erbで異なる部分でもあります。例えば、typeはeditの場合であれば「更新」ですし、newであれば「登録」となります。
次に、呼び出す側も修正しましょう

- app/views/books/new.html.erb

```
<%= render partial: "form", locals: { type: "登録", book: @book, method: nil } %>
```

ファイルの中身を全て消して、上の 1 行だけ記述してください。
renderはコントローラでも使われていた、表示する画面を指示するメソッドでしたね。引数として画面の名前ではなく、抜き出した共通部分のファイルを指定しています。
localsでは共通ファイルの中で使用する変数を定義しています。
methodにはHTTPメソッドを渡すのですが、newの場合は何も指定しなくてもPOSTになるので、何もないことを表すnilを渡しています。

- app/views/books/edit.html.erb

```
<%= render partial: "form", locals: { type: "更新", book: @book, method: "patch" } %>
```

editも同じように、全て削除して、1 行だけ記述します。こちらはHTTPメソッドを指定する必要があるのでmethod変数をpatchとしています。
パーシャルについて、どこを共通と見なすのか、意見が分かれるところでもあります。あまりにもlocalsで変数を渡していると、全然共通していないところを無理や

り共通化しようとしている可能性もあります。今回のように 3 つ程度なら許容範囲でしょう。
エンジニアや現場によって考え方が異なりますので、現場のコードをみて真似てみるのが一番安全です。

【応用編】検索と集計を実装しよう

ここまでで基本的なCRUDとしてはは完成しました。デザインも整えたし、リファクタリングまでしました。

家計簿アプリとしてはあともう少し、追加したい機能が２つあります。

１つはある年ある月の家計簿だけ見れるようにすること(検索機能)。

もう１つは、表示されている家計簿の収支を計算し、トータルとしてどれくらいプラスだったのか、あるいはマイナスだったのか(泣)がわかるような画面にすることです(サマリー機能)。

検索

まずは検索を実装してみしょう。
編集するファイルとしては、検索フォームを追加するapp/views/books/index.html.erbと、検索のためのコードを書くapp/controllers/books_controller.rbです。

- app/views/books/index.html.erb

```
<h1>家計簿</h1>

<div class="card">
  <div class="card-body">
    <%= form_with method: "get", local: true do |f| %>
      <div class="form-row">
        <div class="col">
          <%= f.number_field "year", placeholder: "年度を入力", class: "form-control" %>
        </div>
        <div class="col">
          <%= f.number_field "month", placeholder: "対象月を入力", class: "form-control" %>
        </div>
        <div class="col">
```

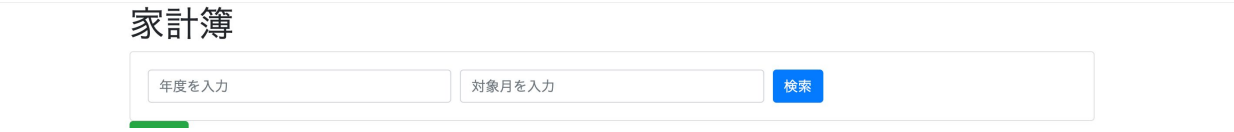
```
        <%= f.submit "検索", class: "btn btn-primary" %>
      </div>
    </div>
  <% end %>
 </div>
</div>
```

h1タグと新規ボタンの間に、コードを追加します。
bootstrapで装飾している部分が多く、少し見にくいので、主要な部分だけ抜き出してみると次のようになります

```
<%= form_with method: "get", local: true do |f| %>
  <%= f.number_field "year", placeholder: "年度を入力", class: "form-control" %>
  <%= f.number_field "month", placeholder: "対象月を入力", class: "form-control" %>
  <%= f.submit "検索", class: "btn btn-primary" %>
<% end %>
```

form_withはこれまでも登場しました。フォームを作るメソッドですね。
特に対象となるモデルはないので、モデルは指定していません。

また、本来であればモデルの設定されていないform_withfはurlオプションでフォームの送り先を記載するのですが、今開いているページと同じURLへフォームを送信する場合は省略することができます。そのため、form_withにも送り先を指定するコードを書いていません。
フォームは一般的にはPOSTでデータを送るのですが、データの登録ではなく、検索という「データ取得」を行うので、methodというキーワードでgetを指定しています。
ここまで書けたら、まだ検索はできないですが、一覧画面を見てみましょう。
上手くいくと、以下のような画面になります。



ビューファイルは、HTMLとRubyが入り交じるため、どのファイルと比べても複雑になりやすいです。コードを実行した結果、どのようになるのかイメージしにくいので、ブラウザの検証ルーツを使い、結果的に表示される画面のソースコードと見比べてみるのが一番わかりやすいです。
次に、コントローラを編集していきましょう。

- app/controllers/books_controller.rb

```
 def index
   @books = Book.all
            @books = @books.where(year: params[:year]) if params[:year].present?
            @books = @books.where(month: params[:month]) if params[:month].present?
```

```
end
```

2行目まではこれまでと同じです。
３行目は年度で検索を行うためのコードです。
whereというメソッドに検索するカラムと値を渡します

```
where(カラム名: 値)
```

３行目だと、カラム名がyearで、params[:year]が値になります。フォームの年度へ数字を入力して検索ボタンを押すことで、yearという名前で値が送られるのですが、送られた値は全てparamsが持っています。
このparamsからyearという名前のついた値を取り出すために、params[:year]としています。
３行目の最後にifがあります。このifは「後置のif」という書き方で、

```
条件が正しければ実行する処理 if 条件
```

という構文になっています。
今回の場合ですと、params[:year].present?が正しければ、検索を実施します。

present?というのは、「本当に中身入ってる？」という確認用のメソッドで、検索フォームに何も記入されずに検索ボタンを押された時は、present?がfalseを返すため、条件をみたさず、検索を実施しません。
年度や月が異なるデータを登録してみて、検索がうまく動くか動作確認してみましょう！

トータルを計算する

最後に、収支合計を計算して、表示してみましょう。
編集するファイルはapp/views/books/index.html.erbだけですが、2 つのパートに分かれています。1 つは計算パートで、もう 1 つは計算結果の表示パートです。
まずは計算パートを見てみましょう。

- app/views/books/index.html.erb

```
<% sum = 0 %>
<% @books.each do |book| %>
<tr>
  <td><%= book.year %>年<%= book.month %>月</td>
  <td><%= book.inout %></td>
  <td><%= book.category %></td>
  <td><%= book.amount %>万円</td>
  <td>
    <%= link_to "詳細", book_path(book), class: "btn btn-info" %>
      <%= link_to "編集", edit_book_path(book), class: "btn btn-warning" %>
     <%= link_to "削除", book_path(book), method: "delete", data: { confirm: '本当に削除しますか？' }, class: "btn btn-danger" %>
  </td>
```

```
</tr>
  <% if book.inout == 1 %>
    <% sum += book.amount %>
  <% else %>
    <% sum -= book.amount %>
  <% end %>
<% end %>
```

まず 1 行目で、変数を用意しています。値は最初 0 としています。sumは合計を表す変数で、家計簿の収入と支出を足したり引いたりしていきます。
次に、下から 6 行目～下から 2 行目までが新しく追加した部分です。eachの外側に書いてしまうとうまく動かないので、難しいですが追加する場所には注意が必要です。

```
  <% if book.inout == 1 %>
    <% sum += book.amount %>
  <% else %>
    <% sum -= book.amount %>
  <% end %>
```

inout項目が1の時は収入を表すので、合計値であるsumにその時の金額を足し算しています。

逆に、inoutが2の時は支出を表すので、合計値であるsumから金額を引いています。
家計簿データをeachを使って順番に表示していますが、せっかく繰り返し処理をしているので、ついでに計算もしてしまおうということです。
表示されている分の合計が計算され、sum変数に入っています。あとはこれを表示するだけです。これが表示部分のパートです。
繰り返し処理の`<% end %>`の次の行にコードを追加します。

```
  <% end %>
  <tr>
    <td></td>
    <td></td>
    <td>合計</td>
    <td><%= sum %>万円</td>
    <td></td>
  </tr>
</table>
```

index.html.erbのファイル全体を以下に載せておきます。

```
<h1>家計簿</h1>
```

```
<div class="card">
  <div class="card-body">
    <%= form_with method: "get", local: true do |f| %>
      <div class="form-row">
        <div class="col">
          <%= f.number_field "year", placeholder: "年度を入力", class: "form-control" %>
        </div>
        <div class="col">
          <%= f.number_field "month", placeholder: "対象月を入力", class: "form-control" %>
        </div>
        <div class="col">
          <%= f.submit "検索", class: "btn btn-primary" %>
        </div>
      </div>
    <% end %>
  </div>
</div>

<%= link_to "+新規", new_book_path, class: "btn btn-success" %>
<table class="table table-striped">
  <tr>
    <th>年月</th>
```

```
    <th>区分</th>
    <th>科目</th>
    <th>金額</th>
    <th>リンク</th>
  </tr>
  <% sum = 0 %>
  <% @books.each do |book| %>
  <tr>
    <td><%= book.year %>年<%= book.month %>月</td>
    <td><%= book.inout %></td>
    <td><%= book.category %></td>
    <td><%= book.amount %>万円</td>
    <td>
      <%= link_to "詳細", book_path(book), class: "btn btn-info" %>
        <%= link_to "編集", edit_book_path(book), class: "btn btn-warning" %>
       <%= link_to "削除", book_path(book), method: "delete", data: { confirm: '本当に削除しますか？' }, class: "btn btn-danger" %>
    </td>
  </tr>
    <% if book.inout == 1 %>
      <% sum += book.amount %>
    <% else %>
      <% sum -= book.amount %>
```

```
    <% end %>
  <% end %>
  <tr>
    <td></td>
    <td></td>
    <td>合計</td>
    <td><%= sum %>万円</td>
    <td>
    </td>
  </tr>
</table>
```

ここまで書けると、画面は以下のようになります。

おめでとうございます！完成しました！

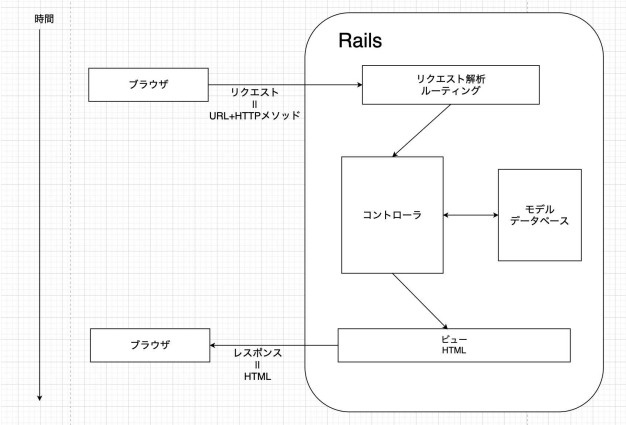
最終課題

お疲れ様でした。後もう少し残っていますが、これまでの復習を兼ねて、最終課題に挑戦してもらいたいと思います。

1. Bookモデルのinoutは、1なら収入、2なら支出を表していますが、/books画面を見てみると、1か2の数字だけが表示されていて、見にくいです。そこで、1なら収入という文字を、2なら支出という文字を画面に表示するようにしてください。
2. コントローラのupdateメソッドでは、更新に成功すると一覧画面へリダイレクトしていました。これを、詳細画面へリダイレクトするように変更してください。
3. コントローラのcreateでは、登録成功時に「"家計簿にデータを 1 件登録しました"」というメッセージをflashへ登録していました。このメッセージを、「"家計簿に2020年7月給料を登録しました"」のように、どのようなデータを登録したのかがわかるように変更してください。
4. 一覧画面では、合計だけ出しましたが、合計に加えて、収入合計、支出合計をそれぞれ計算してください。どのように表示しても構いません。
5. createとupdateアクションでストロングパラメータを使い、必要な項目を抜き出していました。この 2 つのコードは全く同じですよね。なので、set_bookを定義したように、新しくbook_paramsという名前のメソッドを作り重複しているコードを削除しましょう。ただし、before_actionではないです。Book.new(book_params)という感じでメソッドを呼び出します。

RESTful

随分と細かいところに注目してきたので、最後に、改めて、Railsの処理の流れをおさらいし、俯瞰的に眺めてみましょう。



まず、ブラウザからRailsアプリへURLとHTTPメソッドの組み合わせでリクエストを送信します。

ここから、Railsの処理が始まります。

最初にリクエスト内容を解析し、URLとHTTPメソッドを取り出します。routes.rbで、あるURLとHTTPメソッドの組み合わせがどのコントローラのどのメソッドへ処理が飛ぶのかを記載したので、そのルールにしたがって、コントローラへ処理は移動します。

コントローラでは、モデルを使ってデータベースへアクセスし、画面に必要なデータを準備します。

最後にビューファイルを、用意されたデータを使って描画し、erbファイルからhtmlファイルへ変換します。完成したhtmlファイルをブラウザへレスポンスとして返却して処理は完了します。

基本的に全てのこの流れで処理は完結します。

ここで、ルーティングにちょっと注目してみましょう。

ルーティングは、URLとHTTPメソッドの組み合わせが、どの処理に該当するのかを記述するのでした。
HTTPメソッドは、GET(データの取得)やPOST(データの登録または送信)、PATCH(データの更新)と意味が決まっていましたが、URLは何を意味するのでしょうか？
URLとは実は「リソース」を表現します。リソースとは平たく言えば「データ」のことを指します。
例えば、/books/2というURLは、「idが2である家計簿データ」のことを表し、/booksは「複数の家計簿データ」を表します。
まとめると、URLとHTTPメソッドの組み合わせというのは、「データと操作」を表しているのです。
「idが3である家計簿データの更新」なら、/books/3+PATCHになります。このリクエストをどこで処理するのが自然かというと、家計簿をコントロールしている、家計簿コントローラのupdate(更新)メソッドでしょう。この対応関係を記述したものが、ルーティング(routes.rb)ということになります。
そして、リソースと処理の組み合わせを、URLとHTTPメソッドで表現したルーティングをRESTfulと言います。RailsはRESTfulなルーティングを持つフレームワークです。

おわりに

本書を最後までお読みいただき、ありがとうございました！
そして、完走おめでとうございます！お疲れ様でした。
本書は単一モデルのCRUDに注目し、一つ一つの機能を丁寧に解説することを目指しました。決して簡単な内容ではなかったと思います。
ウェブサービス開発というのは、それだけ難しい技術なのです。
本書を一通り読んだだけでは、一から何も見ずにCRUDを作成するのは困難でしょう。しっくりくるまで繰り返し読み返していただければ、エンジニアになっても通用する力は身につくはずです。また、別の書籍やサイトで勉強するというのもアリです。異なる表現を見ることで、しっくりくる説明に出会えるかもしれません。
本書をお読みになった方が、スキルアップし、エンジニアとして社会に貢献していただければ、これ以上嬉しいことはありません。心より、応援しています！

完走者特典

本書出版後に、本書の続編となる、ログイン・アソシエーション編を執筆しました。
本書では家計簿アプリを作成していただいたのですが、残念ながら一人のユーザーしか使えない状態です。
続編では、作っていただいた家計簿アプリに、ユーザー登録やログイン・ログアウト処理、家計簿とユーザーの関連付を実装していただき、クラウド家計簿サービスへ仕上げていただきたいと思います。
続編もボリューミーな内容となっていますが、完走者限定で無料プレゼントしたいと思います。以下のリンクから、PDFで申し訳ないのですが、ダウンロードしていただき、引き続き学習にお役立てください。
完走者特典ダウンロードリンク
https://drive.google.com/file/d/1umR3Wg4xaPs_QS7BHdoO8n6TfTn4L6ut/view?usp=sharing

This book was made by the LeME