HTML講座　発展篇

Middleman, Slim, Sass, Git

目次

HTML講座 発展篇

Middleman, Slim, Sass, Git

1. Middleman(ミドルマン)

静的サイトジェネレータ。 Webサイト構築の際、ヘッダー、フッターなどを部品化。 Slim -> html, Sass -> css への変換も担う。

2. Slim(スリム)

htmlを効果的に記述する。 `<>` や</>を省略でき、簡潔に書ける。

3. Sass(サス)

cssを効果的に記述できる。 入れ子ができる。

4. Git(ギット)

ソースコードの管理を行う。 変更を元に戻したり、バックアップやソースコードの共用が可能。

環境構築

Mac 開発環境整備 備忘録を参考に、環境整備する。

それでは初めていきましょう!

0.1. 用意するもの

- Mac (または PC)
  プログラミングを *Mac* で行うと、いろいろな開発ツールが豊富に提供されており、快適にコーディングを行うことができます。特段の理由がない限り、Mac がお薦めです。 また、広いディスプレイは、エディタやターミナルを開きつつ、調べ物のためにブラウザを開くなど、とても効率が上がります。

- キーボード
  アメリカ西部のカウボーイたちは、馬が死ぬと馬はそこに残していくが、どんなに砂漠を歩こうとも、鞍は自分で担いで往く。 馬は消耗品であり、鞍は自分の体に馴染んだインタフェースだからだ。 いまやパソコンは消耗品であり、キーボードは大切な、生涯使えるインタフェースであることを忘れてはいけない。 【東京大学 名誉教授 和田英一】

  いっさいの妥協を許さず、上質のこだわりを形にした最上級クラスの小型キーボード Happy Hacking Keyboard

- トラックパッド
  GUI操作の為のマウスも良いですが、トラックパッドはもっと快適です。 あなたのお気に入りのコンテンツをこれまで以上に快適にスクロールしたりスワイプできるようになります。 感圧タッチテクノロジーも加わったので、強めに押すと様々な操作ができ、一つのクリックからより多くのことを引き出せます。

  Magic Trackpad 2

0.2. 環境構築

- 多言語対応テキストエディタ：ATOMを入手しよう。 ATOMとは、GitHub社により提供されているオープンソースソフトウェア(OSS)です。 無料でありながら、とっても高機能。プログラマ御用達のテキストエディタです。 プログラムを作る際には、テキストエディタ（エディタ）と呼ばれる、プログラム作成用のソフトウェアを用います。 ダウンロードサイト : https://atom.io/

- パッケージとは、ATOMの拡張機能のことです。ATOMは標準でも十分に快適に使えるように作られていますが、たくさんの有志の方が、様々な便利な拡張機能を提供して下さっています。 RailsでのWebアプリ開発に愛用している Atom プラグイン: https://qiita.com/Atelier-Mirai/items/7ba376d7eda116a663b5

- ターミナルの設定 Macには、標準で ターミナルと呼ばれるアプリケーションが付属しています。これでも十分ですが、iTerm2 (https://www.iterm2.com) が非常に優れていますので、こちらの利用をお薦めいたします。

- macOS用パッケージマネージャ HomeBrewを導入する。

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
  Homebrew/install/master/install.sh)"
```

- Middleman をインストールする。

```
$ gem install middleman
```

- Slim をインストールする。

```
Middleman の中で、設定を行う。
```

- Sass をインストールする。

```
Middleman の中で、設定を行う。
```

- Git をインストールする

```
$ brew install git
```

環境構築

- none
  - 0.1. 用意するもの
  - 0.2. 環境構築

Middleman

静的サイトを構築する使いやすいフレームワーク Middleman はモダンな web 開発のショートカットやツールを 採用した静的サイトジェネレータです。

https://middlemanapp.com/jp/

1. インストール

```
$ gem install middleman
```

インストール後、新しいコマンドと、3つの便利な機能が追加される。

```
$ middleman init     # 新たに Middleman を使ったサイトを作成する。
$ middleman server   # コーディング中に、ブラウザで結果を確認する。
$ middleman build    # 公開用のサイトデータを出力する。
```

2. 新しいサイトの作成

開発を始めるに Middleman が動作するプロジェクトフォルダを作る必要があります。

- 既存フォルダを、Middlemanで開発できるようにする場合

  ```
  $ middleman init
  ```

- 新規に、Middlemanで開発するフォルダを作成する場合

  ```
  $ middleman init my_new_site
  ```

これにより、いくつかのディレクトリとファイルが自動生成される。

source ディレクトリ: webサイト構築用に、slim, sass 等のソースファイルを置くディレクトリ config.rb: middleman の設定を行うためのファイル Gemfile: Middlemanが必要とする、gem(=便利なプログラム)を記述したファイル

3. ディレクトリ構造

```
my_middleman_site/
  +-- .gitignore               # git の対象にしたくないファイルを記述する
  +-- Gemfile                  # Middlemanが必要とするgemを記述する
  +-- Gemfile.lock
  +-- config.rb                # Middleman の各種設定用ファイル
  +-- build/                   # 静的サイトのファイルがコンパイルされ出力されるディレクトリ
  +-- data/                    # データファイルを置くと、テンプレート内(=slim)で利用できる
  +-- helper/                  # よくあるHTMLの作業を簡単にためのプログラムを自作できる
  +-- source/                  # web サイトのソースファイルを置くディレクトリ
      +-- images/              # 画像ファイル
      +-- index.html.slim      # slimで記述。middlemanがコンパイルし、index.htmlになる
      +-- javascripts/         # サイトで必要なjavascript用のディレクトリ
      ¦   +-- site.js
      +-- layouts/             # レイアウトファイル(後述)を置くディレクトリ
      ¦   +-- layout.slim
      +-- stylesheets          # スタイルシートを置くディレクトリ
          +-- site.css.scss
```

それぞれのファイルは、次のように記述します。

Gemfile

```
source 'https://rubygems.org'

# 静的サイトジェネレータ Middleman
gem 'middleman'
# ベンダープリフィックス 自動付与する
gem 'middleman-autoprefixer'
# ファイル更新の際、ブラウザを再読み込みする
gem 'middleman-livereload'
# テンプレートエンジンはSlimを使用する
gem 'slim'
# イメージ圧縮を行う
gem "middleman-imageoptim"
# HTML圧縮を行う
gem "middleman-minify-html"
```

config.rb

```
# 自動再読み込み
activate :livereload

# ベンダープリフィックス付与
activate :autoprefixer do |prefix|
  prefix.browsers = "last 2 versions"
end

# レイアウト
set :layout, 'site'
page 'index.html', layout: 'top'
page 'no_layout.html', layout: false

# ビルド時の設定
configure :build do
  # HTML 圧縮
  activate :minify_html
  # CSS 圧縮
  activate :minify_css
  # # JavaScript 圧縮
  activate :minify_javascript
  # # イメージ 圧縮
  activate :imageoptim
  # アセットファイルの URL にハッシュを追加
  # activate :asset_hash
end

# Slim の設定
set :slim, {
  # デバック用に html をきれいにインデントし属性をソートしない
  # pretty: true, sort_attrs: false,

  # 属性のショートカット
  # Slim コード中、「&text name="user"」と書くと、
  # <input type="text" name="user"> とレンダリングされる。
  shortcut: {'&' => {tag: 'input', attr: 'type'}, '#' => {attr: 'id'}, '.' => {attr: 'class'}}
}
```

source/layouts/layout_sample.slim

```
doctype html
html
  head
    meta charset="utf-8"
    meta name="viewport" content="width=device-width, initial-scale=1"
    title
      = current_page.data.title || "株式会社さくら商会"

    / ファビコンの指定
    = favicon_tag 'favicon.ico'
    link rel="apple-touch-icon" sizes="180x180" href="/images/apple-touch-icon-180x180.png"

    / 検索エンジン用にサイトの紹介文章
    meta name="description" content="お花見ならさくら商会。屋台の出店からライトアップ、場所とりまでお任せくださ

    / fontawesome
    link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.13.0/css/all.css"

    / jQuery
    script src="https://code.jquery.com/jquery-3.5.0.min.js"

    / IE Buster
    script src="https://cdn.jsdelivr.net/npm/ie-buster@1.1.0/dist/ie-buster.min.js"

    / stylesheet の読み込み (style.cssを読み込む)
    = stylesheet_link_tag "style"

    / JavaScript (site.jsを読み込む)
    = javascript_include_tag "site"

  body.site
    / 部分ファイル _header.slim を読み込む
    = partial 'header'
    / この = yield が、index.html.slim等、各々の内容に置き換えられる
    = yield
    / 部分ファイル _footer.slim を読み込む
    = partial 'footer'
```

4. 開発サイクル

4.1. middleman server

コマンドラインから, プロジェクトフォルダの中でプレビューサーバを 起動してください。

```
$ cd my_project
$ middleman server
```

このコマンドはローカルの Web サーバを起動します: http://localhost:4567/ source フォルダでファイルを作成編集し, プレビュー Web サーバ上で 反映された変更を確認することができます。 コマンドラインから Ctrl + C を使って プレビューサーバを停止できます。

4.2. LiveReload

Middleman にはサイト内のファイルを編集するたびにブラウザを自動的にリロードする 拡張がついています。 まず Gemfile に middleman-livereload を追記してください。

Gemfile

```
gem 'middleman-livereload'
```

続いて config.rb を開いて次の行を 追加してください。

config.rb

```
activate :livereload
```

これであなたのブラウザはページ内容に変更があると自動的にリロードされます。

5. ビルド & デプロイ

5.1. middleman build でサイトをビルド

静的サイトのコードを出力する準備ができたら、サイトをビルドする必要があります。 コマンドラインを使い, プロジェクトフォルダの中から middleman build を実行してください。

```
$ cd my_project
$ middleman build
```

サイトをビルドすることで, 必要なものはすべて build ディレクトリに 用意されます。 適宜、公開してください。 (ソースコード管理にGitを、Webサイトの公開にNetlifyの使用がおすすめです。)

5.2. アセットハッシュの付与

プロダクション環境では一般的にアセットファイル名にハッシュ文字列を付与します。

config.rb

```
configure :build do
  # アセットファイルの URL にハッシュを追加
  activate :asset_hash
```

ハッシュ文字列を付与しないときには、

```
<link href="/stylesheets/style.css" rel="stylesheet" />
```

と出力されます。

ハッシュ文字列を付与すると、

```
<link href="/stylesheets/style-795ee195.css" rel="stylesheet" />
```

と出力されます。 styleのあとに付与される文字列は、スタイルシートが更新される都度、異なる文字列となるため、(ブラウザのキャッシュ機能による)「新しいスタイルシートをWebサイトにアップしても、見た目が変わらない」という課題を解消することができます。

6. Frontmatter

Frontmatterは、本の前付けの意味です。 Frontmatter は YAMLフォーマット(形式)でテンプレート上部に記述することができる、ページ固有の変数です。

source/contact.html.slim

```
---
title: "お問い合わせ"
my_list:
  - one
  - two
  - three
---

h1 リスト
ol
  - current_page.data.my_list.each do |f|
    li = f
```

source/layouts/layout.slim

```
doctype html
html
  head
    meta charset="utf-8"
    title
      = current_page.data.title || "会社名"
    = stylesheet_link_tag "site"
    = javascript_include_tag "site"
  body
    = yield
```

でき上がったcontact.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>お問い合わせ</title>
  </head>
  <body>
    <h1>リスト</h1>
    <ol>
      <li>one</li>
      <li>two</li>
      <li>three</li>
    </ol>
  </body>
</html>
```

7. テンプレート言語

Middleman は HTML の開発を簡単にするためにたくさんのテンプレート言語へのアクセスを提供します。テンプレート言語はページ内で変数やループを使えるようにするシンプルなものから、ページをHTMLに変換するまったく異なったフォーマットを提供するものにまで及びます。 Middleman は Slim, Sass のサポートを搭載しています。

Middleman で使うテンプレートはそのファイル名にテンプレート言語の拡張子を 含みます。slim で書かれたシンプルな index ページはファイル名の index.html と、slim の拡張子を含む index.html.slim という名前になります。 sourceディレクトリの直下に、index.html.slimを置き、ビルドすると、 buildディレクトリに、index.htmlが出力されます。

index.html.slim の例

```
h1 ようこそ
ul
  - 5.times do |number|
    li
      | カウント:
      = number
```

ビルド後のindex.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>お問い合わせ</title>
  </head>
  <body>
    <h1>ようこそ</h1>
    <ul>
      <li> カウント: 0 </li>
      <li> カウント: 1 </li>
      <li> カウント: 2 </li>
      <li> カウント: 3 </li>
      <li> カウント: 4 </li>
    </ul>
  </body>
</html>
```

8. ヘルパーメソッド

テンプレートヘルパはよくある HTML の作業を簡単にするため、テンプレートの中で使用できるメソッドです。 基本的なメソッドのほとんどは Ruby on Rails のビューヘルパを利用したことのある人にはお馴染みのものです。

8.1. リンクヘルパ

基本的な使い方は、link_toメソッドに引数として、「リンク名」と「リンク先URL」を与えます。

```
= link_to 'わたくしのサイト', 'http://mysite.com'
```

link_to はより複雑な内容のリンクを生成できるように、ブロックをとることもできます。

```
= link_to 'http://mysite.com' do
  = image_tag 'mylogo.png', alt: 'ロゴマーク'
  p
    | わたくしの会社へようこそ
```

```
<a href="http://mysite.com">
  <img src="/images/mylogo.png" alt="ロゴマーク">
  <p>
    わたくしの会社へようこそ
  </p>
</a>
```

8.2. イメージヘルパ

source/images/ディレクトリ内の画像を表示させたい場合には、 イメージヘルパを使って、次のように書くこともできます。

```
= image_tag 'mylogo.png', alt: 'ロゴマーク'
```

```
<img src="/images/mylogo.png" alt="ロゴマーク">
```

8.3. カスタム定義ヘルパ

Middleman によって提供されるヘルパに加え、コントローラやビューの中からアクセスできる独自のヘルパを追加することができます。 良く使うhtml部品生成用のヘルパを創っておくと便利です。

- カスタム定義ヘルパの例：ファイル名を与えるとチラシ用のhtmlを出力してくれる

helpers/custom_helper.rb

```
def chirashi_tag(title: , filename: )
  "<div class='col-xs-12 col-md-4'>" +
    "<div class='card leaflet'>" +
      "<a href=\"/pdf/#{filename}.pdf\">" +
        "<img src=\"/pdf/thumbnail/#{filename}.jpg\" class='card-img-top w-100' alt=\"\">" +
      "</a>" +
      "<div class='card-footer'>" +
          "#{title}" +
      "</div>" +
    "</div>" +
  "</div>"
end
```

- 実際に、index.html.slimの中で使ってみる

source/index.html.slim

```
= chirashi_tag title: "春の大売り出し", filename: "chirashi"
```

- ビルドして出力された html

build/index.html

```
<div class="col-xs-12 col-md-4">
  <div class="card leaflet">
    <a href="/pdf/chirashi.pdf">
      <img src="/pdf/thumbnail/chirashi.jpg" class="card-img-top w-100" alt="">
    </a>
    <div class="card-footer">春の大売り出し</div>
  </div>
</div>
```

9. レイアウト

レイアウト機能はテンプレート間で共有する, 個別ページを囲むための共通 HTML の 使用を可能にします。 "layout" は "header" や "footer" 両方を含むことで個別ページのコンテンツを 囲みます。

レイアウトの例

source/layouts/layout.slim

```
doctype html
html
  head
    title "株式会社さくら商会"
  body
    / この = yield の部分が、各slimで置き換えられる。
    = yield
```

slim で、各ページに固有の内容を記述します。 **source/index.html.slim**

```
h1 ようこそ、さくら商会へ
```

source/layouts/about.html.slim

```
h1 わたくしたちの会社について
```

ビルドすると、buildディレクトリ以下に、index.htmlやabout.htmlが出力されます。

build/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>株式会社さくら商会</title>
  </head>
  <body>
    <h1>ようこそ、さくら商会へ</h1>
  </body>
</html>
```

build/about.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>株式会社さくら商会</title>
  </head>
  <body>
    <h1>わたくしたちの会社について</h1>
  </body>
</html>
```

全てのページに共通する部分を、レイアウトファイルに纏め、 各ページに固有の内容を、個別ページに記述することで、開発効率が上がります。 レイアウトファイルは、ファイル名が、source/layouts/layout.slim であるのに対し、 個別ページは、ファイル名が、source/index.html.slim や、source/about.html.slimと、拡張子が異なることに注意してください。

9.1. カスタムレイアウト

デフォルトでは, Middleman はそのサイトのすべてのページに同じレイアウト(layout.slim)を適用します。

複数のレイアウトを使い, どのページがどのレイアウトを使うのか指定したい場合があります。例えば, トップページと、その傘下にある下層ページのような場合です。

source/layouts/top.slim

```
doctype html
html
  head
    title "株式会社さくら商会"
  body
    h1 トップページです。
    p 我が社のヒーローイメージです。
    = image_tag 'hero.jpg', alt: 'ヒーローイメージ'
    = yield
```

source/layouts/site.slim

```
doctype html
html
  head
    title "株式会社さくら商会"
  body
    = yield
```

トップページである、`index.html` には、`top.slim` というレイアウトファイルを適用し、下層ページである、それ以外のページには、`site.slim` という下層ページ用のレイアウトファイルを使うようにするためには、`config.rb` に次のように記述します。

config.rb

```
# レイアウトの指定
page 'index.html', layout: 'top'
set :layout, 'site'
```

9.2. 完全なレイアウト無効化

いくつかの場合では, まったくレイアウトを使いたくない場合があります。 `config.rb` で次のように書くと、レイアウトを無効化できます。

config.rb

```
# 全てのページで、レイアウトを適用したくない場合
set :layout, false
```

config.rb

```
# 特定のページ(no_layout.html)にはレイアウトを適用したくない場合
page 'no_layout.html', layout: false
```

10. パーシャル（部分・断片ファイル）

パーシャルはコンテンツの重複を避けるためにページ全体にわたってそのコンテンツを共有する方法です。 パーシャルはページテンプレートとレイアウトで使うことができます。

パーシャルのファイル名は `_（アンダースコア）` から始まります。例として source フォルダに置かれる _footer.slim と 名付けられた footer パーシャルを示します。

source/_footer.slim

```
footer
  | Copyright 株式会社さくら商会
```

次に, "partial" メソッドを使ってデフォルトのレイアウトにパーシャルを配置します。

source/layouts/layout.slim

```
doctype html
html
  head
    title "株式会社さくら商会"
  body
    = yield
    / この = partial "footer" の部分が、``` _footer.slim ``` で置き換えられる。
    = partial "footer"
```

ビルドすると、buildディレクトリ以下に、各ページ が出力されます。

build/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>株式会社さくら商会</title>
  </head>
  <body>
    <footer>Copyright 株式会社さくら商会</footer>
  </body>
</html>
```

パーシャルを使い始めると, 変数を渡すことで異なった呼び出しを行いたくなるかもしれません。次の方法で対応出来ます。

source/_cat_introduction.slim 猫の自己紹介用のパーシャル

```
h2 = "#{name}"
= image_tag "#{image_filename}"
p = "#{introduction}"
```

source/index.html.slim 呼び出し元の猫の紹介ページ

```
h1 猫の紹介
= partial 'cat_introduction',
           locals: { name: 'タマ',
                     image_filename: 'tama.jpg',
                     introduction: 'タマです。白黒の可愛い毛並みの猫です。'}
= partial 'cat_introduction',
           locals: { name: 'ミケ',
                     image_filename: 'mike.jpg',
                     introduction: 'ミケです。茶色の可愛い毛並みの猫です。'}
```

build/index.html 出力されたhtmlファイル

```
<h1>猫の紹介</h1>

<h2>
  タマ
</h2>
<img src="/images/tama.jpg" alt="Tama">
<p>
  タマです。白黒の可愛い毛並みの猫です。
</p>
<h2>
  ミケ
</h2>
<img src="/images/mike.jpg" alt="Mike">
<p>
  ミケです。茶色の可愛い毛並みの猫です。
</p>
```

11. データファイル

ページのコンテンツデータをレンダリングから抜き出すと便利な場合があります。(例：猫の紹介データ) データファイルは data フォルダの中に YAML(ヤムル)形式のデータとして作ることができ, テンプレートの中でこの情報を使うことができます。 data フォルダは, source フォルダと同じように, プロジェクトのルートに置かれます。

ディレクトリ構造(再掲)

```
my_middleman_site/
  +-- .gitignore              # git の対象にしたくないファイルを記述する
  +-- Gemfile                 # Middlemanが必要とするgemを記述する
  +-- Gemfile.lock
  +-- config.rb               # Middleman の各種設定用ファイル
  +-- build/                  # 静的サイトのファイルがコンパイルされ出力されるディレクトリ
  +-- data/                   # データファイルを置くと、テンプレート内(=slim)で利用できる
  +-- helper/                 # よくあるHTMLの作業を簡単にためのプログラムを自作できる
  +-- source/                 # web サイトのソースファイルを置くディレクトリ
      +-- images/             # 画像ファイル
      +-- index.html.slim     # slimで記述。middlemanがコンパイルし、index.htmlになる
      +-- javascripts/        # サイトで必要なjavascript用のディレクトリ
      ¦   +-- site.js
      +-- layouts/            # レイアウトファイル(後述)を置くディレクトリ
      ¦   +-- layout.slim
      +-- stylesheets         # スタイルシートを置くディレクトリ
          +-- site.css.scss
```

それでは、ページのコンテンツデータ（ここでは、猫の紹介データ）を、YAML形式のデータファイルに抜き出してみましょう。 data ディレクトリの中に、cats.ymlという名前で作成します。

data/cats.yml

```
- name: タマ
  image_filename: tama.jpg
  introduction: タマです。白黒の可愛い毛並みの猫です。
- name: ミケ
  image_filename: mike.jpg
  introduction: ミケです。茶色の可愛い毛並みの猫です。
```

`name:` の後には、「半角スペース」を入れて、「タマ」と書きます。 `image_filename:` の前には、「半角スペース２つ」が必要です。

一般的に、ymlファイルは上のように書きますが、下のように書くこともできます。

data/cats.yml

```
[ { name: タマ,
    image_filename: tama.jpg,
    introduction: タマです。白黒の可愛い毛並みの猫です。},
  { name: ミケ,
    image_filename: mike.jpg,
    introduction: ミケです。茶色の可愛い毛並みの猫です。}]
```

```
[
  {一匹目の猫のデータ},
  {二匹目の猫のデータ}
]
```

の形式になっていることが読み取れるでしょうか。([] は、配列といい、個々のデータ(要素)を纏めたものです。 {} は、「ハッシュ(連想配列、辞書)」といい、「name: タマ」のように、「鍵: 値」の形になっていることが特徴です。)

用意したデータファイルは、次のように使います。

source/index.html.slim

```
- data.cats.each do |cat|
  li = cat.name
  li = cat.image_filename
  li = cat.introduction
```

解説です。 slim では、テンプレート中に、Rubyコードを書くことができます。 Ruby コードを書くときには、 `-`（ハイフン）で始めます。 `data.cats` は、Middleman で、データファイルを読み込むための書き方です。このように書くことで、Middlemanは、dataディレクトリにあるcats.ymlから情報を読み込み、Ruby コードで扱えるようにします。 `.each` は、Ruby で、配列の各々の要素について処理するよう、指示するコードです。 `do` は、コードブロックと呼ばれ、以下に続くRubyコードを実行するよう指示する構文です。 `|cat|` には、猫一匹分のデータが入っています。 `cat.name` は、取り出した猫一匹のnameです。(タマやミケなど、個々の猫の名前になります) `li =` ですが、liは、 `<li>タグ` です。 `=` を書くことで、Ruby でいろいろ処理を行った結果を、liタグの内容として出力できるので、 `<li>タマ</li>` のようになります。

纏めると、ビルドして、buildディレクトリに出力されるhtmlは次のようになります。

build/index.html

```
<li>ミケです。茶色の可愛い毛並みの猫です。</li>
<li>tama.jpg</li>
<li>タマです。白黒の可愛い毛並みの猫です。</li>
<li>ミケ</li>
<li>mike.jpg</li>
<li>ミケです。茶色の可愛い毛並みの猫です。</li>
```

cats.yml という、猫の紹介データを用いて、htmlを作成できました。 たくさんのデータがあるときに、同じようなhtmlを繰り返し書かずに済むので、とっても楽です。

12. ファイルサイズ最適化

12.1. CSS と JavaScript の圧縮

Middleman は CSS のミニファイや JavaScript の圧縮処理を行うので, ファイル最適化 について心配することはありません。ほとんどのライブラリはデプロイを行うユーザの ためにミニファイや圧縮されたバージョンを用意していますが, そのファイルは 読めず編集できなかったりします。Middleman はプロジェクトの中にコメント付きの オリジナルファイルを取っておくので, 必要に応じて読んだり編集することができます。 そして, プロジェクトのビルド時には Middleman は最適化処理を行います。

config.rb で, サイトのビルド時に minify_css 機能と minify_javascript 機能を 有効化します。

config.rb

```
configure :build do
  activate :minify_css
  activate :minify_javascript
end
```

12.2. 画像圧縮

ビルド時に画像も圧縮したい場合, `middleman-imageoptim` を試してみましょう。

config.rb

```
activate :imageoptim
```

12.3. HTML 圧縮

Middleman は HTML 出力を圧縮する公式の拡張機能を提供しています。 Gemfile に middleman-minify-html を追加します:

Gemfile

```
gem "middleman-minify-html"
```

`bundle install` を実行し，`config.rb` を開いて次を追加します。

config.rb

```
activate :minify_html
```

ソースを確認すると HTML が圧縮されていることがわかります。

Slim

Slim は 不可解にならない程度に view の構文を本質的な部品まで減らすことを目指したテンプレート言語です。標準的な HTML テンプレートからどれだけのものを減らせるか、検証するところから始まりました。(<,>, 閉じタグなど) 多くの人が Slim に興味を持ったことで, 機能的で柔軟な構文に成長しました。

公式サイトをもとに簡略化。

簡単な特徴

- すっきりした構文
  - 閉じタグの無い短い構文 (代わりにインデントを用いる)
  - 閉じタグを用いた HTML 形式の構文
  - 設定可能なショートカットタグ (デフォルトでは `#` は `<div id="...">` に, `.` は `<div class="...">` に)
- 安全性
  - デフォルトで自動 HTML エスケープ
- 柔軟な設定
- プラグインを用いた拡張性:
  - インクルード
- 高性能
  - ERB に匹敵するスピード

1. リンク

- ホームページ: http://slim-lang.com

2. イントロダクション

2.1. Slim とは?

Slim は 高速, 軽量なテンプレートエンジンです。 Slim の核となる構文は1つの考えによって導かれています: "この動作を行うために最低限必要なものは何か。"

2.2. なぜ Slim を使うのか?

- Slim によって メンテナンスが容易な限りなく最小限のテンプレートを作成でき, 正しい文法の HTML が書けることを保証します。
- Slim の構文は美しく, テンプレートを書くのがより楽しくなります。Slim は主要なフレームワークで互換性があるので, 簡単に始めることができます。
- Slim のアーキテクチャは非常に柔軟なので, 構文の拡張やプラグインを書くことができます。

そう, Slim は速い! Slim は開発当初からパフォーマンスに注意して開発されてきました。 私たちの考えでは, あなたは Slim の機能と構文を使うべきです。Slim はあなたのアプリケーションのパフォーマンスに悪影響を与えないことを保証します。

2.3. どうやって使い始めるの?

Slim を gem としてインストール:

```
gem install slim
```

あなたの Gemfile に `gem 'slim'` と書いてインクルードするか, ファイルに `require 'slim'` と書く必要があります。これだけです! 後は拡張子に .slim を使うだけで準備完了です。

2.4. 構文例

Slim テンプレートがどのようなものか簡単な例を示します:

```
doctype html
html
  head
    title Slim のファイル例
    meta name="keywords" content="template language"
    meta name="author" content=author
    link rel="icon" type="image/png" href=file_path("favicon.png")
    javascript:
      alert('Slim は javascript の埋め込みに対応しています!')

  body
    h1 マークアップ例

    #content
      p このマークアップ例は Slim の典型的なファイルがどのようなものか示します。

    == yield

    - if items.any?
      table#items
        - for item in items
          tr
            td.name = item.name
            td.price = item.price
    - else
      p アイテムが見つかりませんでした。いくつか目録を追加してください。
        ありがとう!

    div id="footer"
      == render 'footer'
      | Copyright &copy; #{@year} #{@author}
```

インデントについて, インデントの深さはあなたの好みで選択できます。マークアップを入れ子にするには最低1つのスペースによるインデントが必要なだけです。

3. ラインインジケータ

3.1. テキスト |

パイプを使うと, Slim はパイプよりも深くインデントされた全ての行をコピーします。

```
body
  p
    |
      一行目
      二行目
      三行目
```

```
<body><p>一行目 二行目 三行目</p></body>
```

改行を入れたい場合、次のように書けます。

```
body
  p
    | 一行目
    br
    | 二行目
    br
    | 三行目
```

```
<body><p>一行目<br>二行目<br>三行目</p></body>
```

3.2. インライン html < (HTML 形式)

HTML タグを直接 Slim の中に書くことができます。Slim では, 閉じタグを使った HTML タグ形式や HTML と Slim を混ぜてテンプレートの中に書くことができます。

```
<html>
  head
    title Example
  <body>
    - if articles.empty?
    - else
      table
        - articles.each do |a|
          <tr><td>#{a.name}</td><td>#{a.description}</td></tr>
  </body>
</html>
```

3.3. 制御コード -

ダッシュは制御コードを意味します。制御コードの例としてループと条件文があります。`end` は `-` の後ろに置くことができません。ブロックはインデントによってのみ定義されます。 複数行にわたる Ruby のコードが必要な場合, 行末にバックスラッシュ `\` を追加します。

```
body
  - if articles.empty?
    | 在庫なし
```

3.4. 出力 =

イコールはバッファに追加する出力を生成する Ruby コードの呼び出しを Slim に命令します。Ruby のコードが複数行にわたる場合, 例のように行末にバックスラッシュを追加します。

```
= javascript_include_tag \
   "jquery",
   "application"
```

行末・行頭にスペースを追加するために修飾子の `>` や `<` がサポートされています。

- `=>` は末尾のスペースを伴った出力をします。 末尾のスペースが追加されることを除いて, 単一の等合 (`=`) と同じです。
- `=<` は先頭のスペースを伴った出力をします。先頭のスペースが追加されることを除いて, 単一の等号 (`=`) と同じです。

3.5. HTML エスケープを伴わない出力 ==

単一のイコール (`=`) と同じですが, `escape_html` メソッドを経由しません。 末尾や先頭のスペースを追加するための修飾子 `>` と `<` はサポートされています。

- `==>` は HTML エスケープを行わずに, 末尾のスペースを伴った出力をします。末尾のスペースが追加されることを除いて, 二重等号 (`==`) と同じです。
- `==<` は HTML エスケープを行わずに, 先頭のスペースを伴った出力をします。先頭のスペースが追加されることを除いて, 二重等号 (`==`) と同じです。

3.6. コードコメント /

コードコメントにはスラッシュを使います。スラッシュ以降は最終的なレンダリング結果に表示されません。コードコメントには `/` を, html コメントには `/!` を使います。

```
body
  p
    / この行は表示されません。
      この行も表示されません。
    /! html コメントとして表示されます。
```

構文解析結果は以下:

```
<body><p><!--html コメントとして表示されます。--></p></body>
```

3.7. HTML コメント /!

html コメントにはスラッシュの直後にエクスクラメーションマークを使います (`<!-- ... -->`)。

4. HTML タグ

4.1. <!DOCTYPE> 宣言

doctype キーワードでは, とても簡単な方法で複雑な DOCTYPE を生成できます。

```
doctype html
```

```
<!DOCTYPE html>
```

4.2. 行頭・行末にスペースを追加する (<, >)

a タグの後に > を追加することで末尾にスペースを追加するよう Slim に強制することができます。

```
a> href='url1' リンク1
a> href='url2' リンク2
```

< を追加することで先頭にスペースを追加できます。

```
a< href='url1' リンク1
a< href='url2' リンク2
```

これらを組み合わせて使うこともできます。

```
a<> href='url1' リンク1
```

4.3. インラインタグ

タグをよりコンパクトにインラインにしたくなることがあるかもしれません。

```
ul
  li.first: a href="/a" A リンク
  li: a href="/b" B リンク
```

可読性のために, 属性を囲むことができるのを忘れないでください。

```
ul
  li.first: a[href="/a"] A リンク
  li: a[href="/b"] B リンク
```

4.4. テキストコンテンツ

タグと同じ行で開始するか、入れ子にするか、どちらかを選択できます。

```
body
  h1 id="headline" 私のサイトへようこそ。
```

```
body
  h1 id="headline"
    | 私のサイトへようこそ。
```

4.5. 動的コンテンツ (= と ==)

同じ行で呼び出すか、入れ子にすることができます。 Rubyコードにより、page_headline を定義している場合に使います。

```
body
  h1 id="headline" = page_headline
```

```
body
  h1 id="headline"
    = page_headline
```

4.6. 属性

タグの後に直接属性を書きます。通常の属性記述にはダブルクォート `"` か シングルクォート `'` を使わなければなりません (引用符で囲まれた属性)。

```
a href="http://slim-lang.com" title='Slim のホームページ' Slim のホームページへ
```

引用符で囲まれたテキストを属性として使えます。

4.6.1. 属性の囲み

区切り文字が構文を読みやすくするのであれば, `{...}` , `(...)` , `[...]` で属性を囲むことができます。

```
body
  h1(id="logo") = page_logo
  h2[id="tagline" class="small tagline"] = page_tagline
```

属性を囲んだ場合, 属性を複数行にわたって書くことができます:

```
h2[id="tagline"
   class="small tagline"] = page_tagline
```

4.6.2. 引用符で囲まれた属性

例:

```
a href="http://slim-lang.com" title='Slim のホームページ' Slim のホームページへ
```

引用符で囲まれたテキストを属性として使えます:

```
a href="http://#{url}" #{url} へ
```

4.6.3. Ruby コードを用いた属性

`=` の後に直接 Ruby コードを書きます。コードにスペースが含まれる場合, `(...)` の括弧でコードを囲まなければなりません。ハッシュを `{...}` に, 配列を `[...]` に書くこともできます。

```
body
  table
    - for user in users
      td id="user_#{user.id}" class=user.role
        a href=user_action(user, :edit) Edit #{user.name}
        a href=(path_to_user user) = user.name
```

4.6.4. 真偽値属性

属性値の `true` , `false` や `nil` は真偽値として 評価されます。属性を括弧で囲む場合, 属性値の指定を省略することができます。

```
input type="text" disabled="disabled"
input type="text" disabled=true
input(type="text" disabled)

input type="text"
input type="text" disabled=false
input type="text" disabled=nil
```

4.7. ショートカット

4.7.1. ID ショートカット # と class ショートカット .

`id` と `class` の属性を次のショートカットで指定できます。

```
body
  h1#headline
    = page_headline
  h2#tagline.small.tagline
    = page_tagline
  .content
    = show_content
```

これは次に同じです

```
body
  h1 id="headline"
    = page_headline
  h2 id="tagline" class="small tagline"
    = page_tagline
  div class="content"
    = show_content
```

4.7.2. タグショートカット

`:shortcut` オプションを設定することで独自のタグショートカットを定義できます。Rails アプリケーションでは, `config/initializers/slim.rb` のようなイニシャライザに定義します。Middleman アプリでは, `config.rb` の中に以下を記述します。

```
Slim::Engine.set_options shortcut: {'c' => {tag: 'container'}, '#' => {attr: 'id'}, '.' => {attr:
```

Slim コードの中でこの様に使用できます。

```
c.content テキスト
```

レンダリング結果

```
<container class="content">テキスト</container>
```

4.7.3. 属性のショートカット

カスタムショートカットを定義することができます (id の `#` , class の `.` のように)。

例として, type 属性付きの input 要素のショートカット `&` を追加します。

```
Slim::Engine.set_options shortcut: {'&' => {tag: 'input', attr: 'type'}, '#' => {attr: 'id'}, '.'
```

Slim コードの中でこの様に使用できます。

```
&text name="user"
&password name="pw"
&submit
```

レンダリング結果

```
<input type="text" name="user" />
<input type="password" name="pw" />
<input type="submit" />
```

別の例として, role 属性のショートカット `@` を追加します。

```
Slim::Engine.set_options shortcut: {'@' => 'role', '#' => 'id', '.' => 'class'}
```

Slim コードの中でこの様に使用できます。

```
.person@admin = person.name
```

レンダリング結果

```
<div class="person" role="admin">Daniel</div>
```

5. テキストの展開

Ruby の標準的な展開方法を使用します。テキストはデフォルトで html エスケープされます。

```
body
  h1 ようこそ #{current_user.name} ショーへ。
```

6. 埋め込みエンジン

slim中に、cssなども記述できます。

例:

```
    css:
      body: {
        background: yellow;
      }

    scss class="myClass":
      $color: #f00;
      body { color: $color; }

    markdown:
      #Header
        #{"Markdown"} からこんにちは!
        2行目!
```

レンダリング結果：

```
<style type="text/css">body{background: yellow}</style>
<style class="myClass" type="text/css">body{color:red}</style>
<h1 id="header">Header</h1>
<p>Markdown からこんにちは! 2行目!</p>
```

対応エンジン:

フィルタ	必要な gems	種類	説明
ruby:	なし	ショートカット	Ruby コードを埋め込むショートカット
javascript:	なし	ショートカット	javascript コードを埋め込み、script タグで囲む
css:	なし	ショートカット	css コードを埋め込み、style タグで囲む
scss:	sass	コンパイル時	scss コードを埋め込み、style タグで囲む
markdown:	redcarpet/rdiscount/kramdown	コンパイル時 + 展開	Markdown をコンパイルし、テキスト中の #{variables} を展開

7. Slim の設定

7.1. デフォルトオプション

通常、slim で生成される html ファイルは、空白等を取り除き、コンパクトに圧縮されています。 これにより、Webサイトに公開した際の速度改善等を望むことができます。 そして、デバッグ時には、これを無効化することができます。 `Middleman` アプリケーションでは、 `config.rb` 中に、以下のように記述します。

```
# デバック用に html をきれいにインデントし属性をソートしない
Slim::Engine.set_options pretty: true, sort_attrs: false
```

7.2. 構文ハイライト

様々なテキストエディタのためのプラグインがあります。:

- Atom用language-slimパッケージ

7.3. テンプレート変換

7.3.1. htmlファイルからslimファイルへ変換

```
$ html2slim index.html index.html.slim
```

7.3.2. slimファイルからhtmlファイルへ変換

```
$ slimrb -p index.html.slim > index.html
```

Slim

css と sass の比較

Sass: Syntactically Awesome Style Sheets の略。 公式ガイド： https://sass-lang.com/guide

Sassには、sass形式とscss形式がある。 従来のcssと記法が同じであるため、scss形式が広く使われている。 sass記法は、`{}`や、`;`を取り除き、よりすっきりとさせている。

1. Sassの特徴

- 一行コメント `//` が使える。
- 変数が使える。
- 入れ子(ネスト)ができる。
- スタイルファイルを分けて管理できる。
- ミックスイン（部品の再利用）ができる。
    - メディアクエリの記述が楽
- 演算ができる。
    - 幅や高さなどの四則演算。
    - 色の演算。
- オリジナル関数の定義

2. 一行コメントを使う

`sass` では、一行コメント `//` が使える。 複数行選択し、 `Command + /` で、全てコメントにできる。 css に コンパイルされると、コメントは残らない。 (コンパイルされたcssにコメントを残したいのであれば、 `/* */` を使う。)

3. 変数を使う

SCSS

```
$font-stack:    Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

CSS

```
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}
```

`scss` では、「変数」を用いることができる。

`$font-stack` や、 `$primary-color` が、変数である。スタイルを変更したくなった際には、変数の値を変更するだけでよいため、管理が楽である。

4. 入れ子(ネスト)にできる。

SCSS

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

CSS

```
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}
nav li {
  display: inline-block;
}
nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}
```

htmlの構造そのままに記述していけるので、楽である。

4.1. 入れ子(ネスト) その2

SCSS

```
a {
  color: pink;
  &:hover {
    color: red;
  }
  &.active {
    color: blue;
  }
}
```

CSS

```
a {
  color: pink;
}
a:hover {
  color: red;
}
a.active {
  color: blue;
}
```

`&` で繋いで、纏めて書ける。

5. スタイルファイルを分けて管理できる。

一つの大きなスタイルシートを管理するのは大変である。 このため、機能ごとに個々のスタイルシートに分け、管理する。

CSS の 設計思想は各種あるが、FLOCSS(フロックス)が良いと思う。 https://github.com/hiloki/flocss

5.1. 基本原則

FLOCSSは次の3つのレイヤーと、**Object**レイヤーの子レイヤーで構成されます。

1. Foundation - reset/normalize/base...
2. Layout - header/main/sidebar/footer...
3. Object
    i. Component - grid/button/form/media...
    ii. Project - articles/ranking/promo...
    iii. Utility - clearfix/display/margin...

次の例は、Sassを採用した場合の例です。

```
├── foundation
│   ├── _base.scss
│   └── _reset.scss
├── layout
│   ├── _footer.scss
│   ├── _header.scss
│   ├── _main.scss
│   └── _sidebar.scss
└── object
    ├── component
    │   ├── _button.scss
    │   ├── _dialog.scss
    │   ├── _grid.scss
    │   └── _media.scss
    ├── project
    │   ├── _articles.scss
    │   ├── _comments.scss
    │   ├── _gallery.scss
    │   └── _profile.scss
    └── utility
        ├── _align.scss
        ├── _clearfix.scss
        ├── _margin.scss
        ├── _position.scss
        ├── _size.scss
        └── _text.scss
```

モジュール単位でファイルを分割することによって、ページ単位またはプロジェクト単位でのモジュールの追加・削除の管理が容易になります。

これらを統括するための `app.scss` のようなファイルからは次のように参照します。

```
// ==============================================================================
// Foundation
// ==============================================================================

@import "foundation/_reset";
@import "foundation/_base";

// ==============================================================================
// Layout
// ==============================================================================

@import "layout/_footer";
@import "layout/_header";
@import "layout/_main";
@import "layout/_sidebar";

// ==============================================================================
// Object
// ==============================================================================

// ------------------------------------------------------------------------------
// Component
// ------------------------------------------------------------------------------

@import "object/component/_button";
@import "object/component/_dialog";
@import "object/component/_grid";
@import "object/component/_media";

// ------------------------------------------------------------------------------
// Project
// ------------------------------------------------------------------------------

@import "object/project/_articles";
@import "object/project/_comments";
@import "object/project/_gallery";
@import "object/project/_profile";

// ------------------------------------------------------------------------------
// Utility
// ------------------------------------------------------------------------------

@import "object/utility/_align";
@import "object/utility/_clearfix";
@import "object/utility/_margin";
@import "object/utility/_position";
@import "object/utility/_size";
@import "object/utility/_text";
```

5.2. ミックスイン @mixin

CSS では、一度定義した CSS の再利用は難しいですが、Sass ではミックスインを使うことで CSS の再利用が可能になります。

- Web Design Leaves より引用。

ミックスインを使うとプロパティやセレクタをまとめてワンセットにしておいて、それらを読み込むことができます。ミックスインは @mixin ディレクティブ(指示命令)を用いて定義し、@include ディレクティブで定義したミックスインを呼び出します。 ミックスインでは引数(ひきすう)を取ることができるので、より使い回しが柔軟にできます。 ミックスインは、@mixin の後に半角スペースを置き、任意の名前で定義します。

```
@mixin grayBox {  // ミックスインの定義
  margin: 20px 0;
  padding: 10px;
  border: 1px solid #999;
  background-color: #eee;
  color: #333;
}

.foo {
  @include grayBox;  // 定義したミックスインの呼び出し
}
```

```
.foo {
  margin: 20px 0;
  padding: 10px;
  border: 1px solid #999;
  background-color: #EEE;
  color: #333;
}
```

5.2.1. 引数を使ったミックスイン

ミックスインは引数を取ることができます。引数はミックスイン名の後に括弧を書き、その括弧の中に記述します。引数が複数ある場合は、カンマで区切って記述します。

```
@mixin my-border($color, $width, $radius) {
  border: {
    color: $color;
    width: $width;
    radius: $radius;
  }
}

p {
  @include my-border(blue, 1px, 3px);
}
```

```
p {
  border-color: blue;
  border-width: 1px;
  border-radius: 3px;
}
```

5.2.2. 引数に初期値を設定

引数の初期値を設定しておくと、初期値と同じ値を使用する場合は、引数を省略することができます。よく使う値があれば、初期値を設定しておくと便利です。

引数の初期値を設定するには、変数 と同じ書式（名前は「$」から始め、「:」で区切って値を指定）で記述します。

呼び出す際は、引数が初期値と同じ場合は、（）や値は省略することができます。初期値と値が異なる場合だけ、引数の値を指定します。

```
@mixin kadomaru($radius: 5px) {
  border-radius: $radius;
}

.foo {
  @include kadomaru;
}

.bar {
  @include kadomaru();
}

.baz {
  @include kadomaru(10px);
}
```

```
.foo {
  border-radius: 5px;
}
.bar {
  border-radius: 5px;
}
.baz {
  border-radius: 10px;
}
```

5.3. メディアクエリへの活用

https://www.tam-tam.co.jp/tipsnote/html_css/post10708.html より引用

レスポンシブWebデザインではメディアクエリ（media queries）を書くことが多くなります。通常のCSSではブレイクポイントを変更したくなったときに、すでに書いてしまった箇所を直していくのはとても大変です。 Sass（scss記法）を使えば、変数や@mixinを使うことで1箇所で管理することが容易になります。

```
$breakpoints: (
  'tablet':  'screen and (min-width: 481px)',
  'desktop': 'screen and (min-width: 960px)',
) !default;

@mixin mq($breakpoint: tablet) {
  @media #{map-get($breakpoints, $breakpoint)} {
    @content;
  }
}
```

@mixinを呼び出すときは以下のようになります。

```
.foo {
  color: blue;
  @include mq() { // 引数を省略（初期値はtabletの481px）
    color: yellow;
  }
  @include mq(desktop) { // 引数を個別に指定
    color: red;
  }
}
```

このように出力（コンパイル）されます。

```
.foo {
  color: blue;
}

@media screen and (min-width: 481px) {
  .foo {
    color: yellow;
  }
}

@media screen and (min-width: 960px) {
  .foo {
    color: red;
  }
}
```

- Web Design Leaves より引用。

5.4. 数値の操作（四則演算）

Sass では数値型の値に計算の記号を使うことで計算をすることができます。単位を省略すると元の単位にあわせて計算されます。

```
.thumb {
  width: 200px - (5 * 2) - 2;
  padding: 5px + 3px;
  border: 1px * 2 solid #ccc;
}
```

```
.thumb {
  width: 188px;
  padding: 8px;
  border: 2px solid #ccc;
}
```

計算で使える記号には、以下のようなものがあります。

- 加算： `+（プラス）`
- 減算： `-（ハイフン）`
- 乗算： `*（アスタリスク）`
- 除算： `/（スラッシュ）`

- 剰余： `%`（パーセント）

実際には、以下のように変数を使って計算することが多いと思います。

```
$main_width: 600px;
$border_width: 2px;

.foo {
  $padding: 5px;
  width: $main_width - $padding * 2 - $border_width *2;
}
```

```
.foo {
  width: 586px;
}
```

5.5. 色関連の関数

関数名	機能	実行例
rgba($color, $alpha)	RGB値に透明度を指定	rgba(blue, 0.2) => rgba(0, 0, 255, 0.2)
lighten($color, $amount)	明るい色を作成	lighten(#800, 20%) => #e00
darken($color, $amount)	暗い色を作成	darken(hsl(25, 100%, 80%), 30%) => hsl(25, 100%, 50%)
mix($color1, $color2, [$weight])	中間色を作成	mix(#f00, #00f, 25%) => #3f00bf
saturate($color, $amount)	彩度の高い色を作成	saturate(#855, 20%) => #9e3f3f
desaturate($color, $amount)	彩度の低い色を作成	desaturate(#855, 20%) => #726b6b

5.6. オリジナル関数の定義

@function を使って、自分で独自の function（関数）を定義することができます。以下が構文です。

```
@function 関数名($引数) {
  // 処理の記述（必要であれば）
  @return 戻り値;
}
```

@function で関数の宣言をします。独自の関数名を指定して、引数を設定し、@return を使って戻り値を返します。引数は必須ではありません。

以下はサイズを変更する独自関数の例です。

```
@function _changeSize($value, $size) {
  @return $value * $size;
}

.foo {
  width: _changeSize(200px, 1/3) ;
}

.bar {
  height: _changeSize(50px, 1.6);
}
```

```
.foo {
  width: 66.66667px;
}

.bar {
  height: 80px;
}
```

5.7. コメント

独自に定義した関数の場合、その関数がどのようなものなのかをコメントとして残しておくと便利です。引数の型や単位が必要か等を記述するようにすると良いと思います。

```
// @function _linearGradient  グラデーションの値を返す
// @param    {Color} $baseColor
// @param    {Number} percentage 0-100% default: 20%
// @return   {String} linear-gradient value
@function _linearGradient($baseColor, $amount: 20%) {
  @return "linear-gradient(" + $baseColor + "," + darken($baseColor, $amount) + ")";
}

.foo {
  background-image :_linearGradient(#cccccc);
}
```

上記の例では、第2引数に初期値を設定しています。

```
.foo {
  background-image: "linear-gradient(#cccccc, #999999)";
}
```

Git(ギット)

分散型バージョン管理システム。 ソースコードの管理を行う。

３つの場所がある

- ローカル
- ステージングエリア(どのファイルをリモートへプッシュするのか管理する)
- リモート

ローカル環境で、いろいろ開発。 変更をコミット（保存） ブランチを切る（ブランチは現在のコミットへのポインタ） リモート環境（共有用） 変更を元に戻したり、バックアップやソースコードの共用が可能。

GUIなら、SourceTree 書籍

- わかばちゃんと学ぶGit
  - 漫画や可愛いイラストが豊富で、分かりやすい。

導入スライド

参考 git switchとrestoreの役割と機能について

イントールから良く使うコマンドまで

Homebrew (Macパッケージマネージャ)を導入 ターミナルより、

```
$ brew install git
```

で、gitをインストールする。

1. gitの初期設定

1.0.1. Gitのバージョン確認

```
$ git version
```

Git を使うためには、利用者名とメールアドレスが必要

1.0.2. 利用者情報の登録

```
git config --global user.name  "Yamada Taro"
git config --global user.email "taro@example.com"
```

1.1. git commit で Atom を使う

```
$ git config --global core.editor "atom --wait"
```

使い方

適宜 コードを書く 書き終わったらgit commit Atomが起動するので、コミットメッセージを記す ⌘-S で保存 ⌘-W でタブを閉じる git push

1.1.1. 利用者情報確認

```
$ git config user.name
$ git config user.email
```

1.1.2. 利用者情報確認(一覧)

```
git config --list
```

1.1.3. 利用者情報の保存先と表示

```
cat ~/.gitconfig
```

ターミナルコマンド | コマンド名 | 説明 | |:----------|:--------------------------------| | cd | ディレクトリの移動 | | ls | ディレクトリの内容を表示 | | mkdir | ディレクトリを新規に作成 | | rm | ファイルを削除 | | cp | ファイルをコピー | | mv | ファイルの移動とファイル名の変更 | | cat | ファイルの中身を表示 |

2. Giの基本操作

2.0.1. git init

ローカルリポジトリの新規作成

```
git init
```

2.0.2. git add

ステージへの追加

```
$ git add <ファイル名>
$ git add <ディレクトリ名>
$ git add .
```

2.0.3. git commit

変更をコミットする

```
git commit
git commit -m "メッセージ"
```

- git commit を実行すると atomが立ち上がるので、 コメントを入力しファイルを保存し閉じるとコミットが完了する。
- -m でgitエディタを立ち上げる事なくコミットできる。

3. 状態の確認方法

3.0.1. git status

現在の変更状況をファイル単位で確認する

```
git status
```

・ワークツリーとステージ ・ステージとリポジトリ 上記のそれぞれで変更されたファイルが表示される。

3.0.2. git diff

現在の変更差分を確認する

```
# git add する前の変更分
git diff
git diff <ファイル名>

# git add した後の変更分
git diff --staged
```

3.0.3. git log

変更履歴を確認する

```
# 1行で表示する
git log --oneline

# ファイルの変更差分を表示する
git log -p index.html

# 表示するコミット数を制限する
git log -n <コミット数>
```

4. ファイルの移動・削除の記録方法

4.0.1. git rm

ファイルの削除を記録する

```
# ファイルごと削除
git rm <ファイル名>
git rm -r <ディレクトリ名>

# ファイルを残したい時
git rm --cached <ファイル名>
```

パスワードファイルなど間違えてコミットしてしまった場合には `--cached` オプションを使ってgitの追跡から外す

4.0.2. git mv

ファイルの移動を記録する(ファイル名変更)

```
$ git mv <旧ファイル> <新ファイル>
```

以下のコマンドと同じ

```
mv <旧ファイル>  <新ファイル>
git rm <旧ファイル>
git add <新ファイル>
```

5. 変更を取り消す処理

5.0.1. git checkout

ファイルの変更を取り消す

```
git checkout -- <ファイル名>
git checkout -- <ディレクトリ名>

# 前変更を取り消す
git checkout -- .
```

ブランチ名とファイル名が同じ場合、区別をつける為に `--` をつける。

5.0.2. git reset HEAD

ステージングを取り消す

```
git reset HEAD <ファイル名>
git reset HEAD <ディレクトリ名>

# 全変更を取り消す
git reset HEAD .
```

ステージングは取り消されるが、ファイルの変更は取り消されないので、ファイルの変更まで取り消したい場合は「git checkout」コマンドを使う。

6. リモート

6.0.1. git add origin

リモートリポジトリを新規登録する

```
git add origin https:github.com/user/repo.git
```

- originというショートカットでURLのリモートリポジトリを登録する
- 今後は `origin` という名前でgithubリポジトリにアップしたり取得したりできる
- gitではメインのリモートリポジトリの事を `origin` と呼んでいる

6.0.2. git clone

既存のリモートリポジトリをローカルに複製する

```
git clone リモートリポジトリ名
```

6.0.3. git push

リモートリポジトリ（Github）へ送信する

```
git push <リモート名><ブランチ名>
git push origin master

# 次回以降 git push だけでよくなるコマンド
git push -u origin master
```

6.0.4. git remote

リモートを表示する

```
git remote

# 対応するURLを表示
git remote -v
```

6.0.5. git remote show

リモートの詳細情報を表示する

```
git remote show <リモート名>
git remote show origin
```

- FetchとPushのURL
- リモートブランチ
- git pullの挙動
- git pushの挙動

6.0.6. git remote rename

リモートを変更・削除する

```
# リモートを変更する場合
git remote rename <旧リモート名> <新リモート名>
git remote rename tutorial new_tutorial

# リモートを削除する場合
git remote rm <リモート名>
git remote rm new_tutorial
```

6.0.7. git remote add

リモートリポジトリを新規追加する

```
git remote add <リモート名> <リモートURL>
git remote add tutorial https://github.com/user/repo.git

# 「origin」で登録した時と同じように次回からは「tutorial」というショートカットでプッシュやプルをする事ができる。
```

リモートリポジトリを複数登録するのはどんな時か？

・チーム開発とは別に自分でもリモートリポジトリを持っておきたい場合。 ・複数のチームで開発する場合。

7. リモートから情報を取得する

1.フェッチ 2.プル

7.0.1. git fetch

```
git fetch <リモート名>
git fetch origin

# フェッチした情報は　remotes/リモート/ブランチ　に保存される。

# フェッチした情報をマージコマンドでワークツリーに取り込む
git merge origin/master
```

フェッチされた情報はローカルリポジトリの **remotes/リモート/ブランチ** に保存される。

この情報をワークツリーに反映させるには **git mergeコマンド** を使う必要がある。

```
**補足**

# ブランチの中身を全て確認する（-aはallの略）
# 現在いるブランチに「*」がつく
git branch -a

# ブランチを切り替えてフェッチした内容を確認する
git checkout remotes/origin/master

# 元のマスターブランチに切り替える
git checkout master
```

7.0.2. git pull

```
git pull <リモート名> <ブランチ名>
git pull origin master

# 上記コマンドは省略可能
git pull

# git pullは下記コマンドと同じ事をしている
git fetch origin master
git merge origin/master
```

git pullはリモートリポジトリからローカルリポジトリのワークツリーに反映させるのを一度にやってしまう。便利な反面注意点がある。

プルを実行する際の注意点

プルしてきたブランチは現在いるブランチに統合される為、思わぬブランチに統合されファイルがぐちゃぐちゃになってしまう危険性がある。 そのため慣れないうちはフェッチを使った方が安全。

8. ブランチ

並行して複数の機能を開発するためにあるのがブランチ

8.0.1. ブランチを新規追加する

```
git branch <ブランチ名>
git branch feature

# ブランチを作成するだけでブランチの切り替えまでは行わないので注意
```

8.0.2. ブランチの一覧を表示する

```
git branch

# 全てのブランチを表示する
# -aはallの略でリモート追跡ブランチも表示される
git branch -a
```

8.0.3. ブランチを切り替える

```
git checkout <既存ブランチ名>
git checkout feature

# ブランチを新規作成して切り替える
git checkout -b <新ブランチ名>
```

8.0.4. ブランチを変更・削除する

```
# 自分が作業しているブランチの名前を変更する
git branch -m <ブランチ名>
git branch -m new_branch

# ブランチを削除する
# masterにマージされていない変更が残っている場合はエラーが出る
git branch -d <ブランチ名>
git branch -d feature

# ブランチを強制削除する
# masterにマージされていない変更が残っていても強制削除される
git branch -D <branch名>
```

8.0.5. リモート追跡ブランチとは

```
# リモートリポジトリからfetchした情報が保存されるブランチ
git branch -a
# を実行すると

remotes/origin/master
remotes/origin/feature

# などと表示される。

# このブランチの内容を参照したりマージしたりする場合は
# 頭の remotes/ はつけなくて良い。

git merge origin/master
git status origin/feature
```

9. マージ

マージとは他の人の変更内容を取り込む作業のこと マージには以下の２種類がある。

・`Auto Merge` ・`Fast Forward`

9.0.1. Auto Merge:基本的なマージ

`masterブランチ` の内容が `developブランチ` を分岐した時より進んでしまっている場合、両方の変更を取り込んだマージコミットが作成される。 マージコミットは２つのperentをもつ。

```
# 指定したブランチが作業中のブランチにマージされる
git merge <ブランチ名>
git merge <リモート名/ブランチ名>
git merge origin/master
```

9.0.2. Fast Forward

`hotfixブランチ` の変更をmasterにマージする際に、 `masterブランチ` に変更がなければ、 `masterブランチ` が `hotfixブランチ` に移動するだけで `hotfixブランチ` の内容を取り込む事ができる。

このようなマージを `Fast Foward` という。

```
# Fast Fowardでコミットメッセージを残すには
# 設定でFast Fowardを無効にする

git config --global merge.ff false
```

9.0.3. コンフリクト

`masterブランチ` と `developブランチ` で 同じファイルの同じ行が編集されていた場合、 マージした際に `コンフリクト` が起きる。

コンフリクトが起きないようにする為には

・複数人で同じファイルを変更しない ・pullやmergeをする際に変更中の状態をなくしておく（commitやstashをしておく）

10. プルリクエスト

自分の変更したコードをリポジトリに取り込んでもらえるよう依頼する機能

```
# プルリクエストの順序

1. ブランチを切る
2. ファイルを編集する
3. ローカルにadd,commitする
4. githubにプッシュする
5. githubで「Pull request」タブの「New pull request」ボタンを選択。
6.「base」と「compare」を選択する。
7.「Create pull request」を選択。
8.タイトルとコメントを入力。
9.「Create pull request」を押す。
10.右側の「reviewers」からレビューしてもらいたい人を選択し通知を送る。


# レビュワーの作業順序

1. githubで「Pull request」タブのレビューするコードを選択。
2. 「File changed」からコードを確認する。
3. コードの修正依頼をする場合は修正するコードをホバーして「➕」を押す。
4. コメントを入力して「Add single comment」を押す
5. コードレビューがリクエストした人に送信される。


# プルリクエストした内容を承認する場合

1.githubで「Pull request」タブの「Review changes」を押す。
2.「Approve」を選択し「Submit review」を押す。


# 承認されたリクエストをマージする方法

1. githubで「Pull request」タブの中の「conversation」タブで「Merge pull request」を選択する。
2.マージメッセージを確認し「Comfirm merge」を押す。
3.「Delete branch」ボタンを押してプルリクエストブランチを削除する。


# マージした内容をローカルにも取り込みプリクエストブランチを削除する。

git checkout master
git pull origin master
git branch -d pull_request
```

11. リベース

変更を統合する際に、履歴をきれいに整えるために使うのがリベース 作業が完了したブランチを分岐元のブランチにくっつける時に使う機能の一つです。

```
git rebase <branch名>
```

11.0.1. リベースの注意点

githubにプッシュしたコミットをリベースするのはNG。 また `git push -f` で強制的にプッシュするのもNG。

11.0.2. マージとリベースのどちらが良いか？

マージ	リベース
コンフリクトの解消が比較的簡単	コンフリクトの解消が大変（コミット毎に解消する必要がある）
マージコミットがたくさんあると履歴が複雑化する	履歴をきれいに保つ事ができる

11.0.3. コンフリクトを事前に確認する方法

一度githubにプッシュしてプルリクエストを作成する。 コンフリクトしている場合はアラートが表示される。 アラートが表示された場合はリベースではなくマージを使うと楽。

11.0.4. プルにはマージ型とリベース型がある

```
# マージ型のプル（通常のプル fetch → mergeと同じ挙動）
git pull <リモート名> <ブランチ名>
git pull origin master

# リベース型のプル（fetch → rebaseと同じ挙動）
git pull --rebase <リモート名> <ブランチ名>
git pull--rebase origin master
```

11.0.5. リベース型のメリット

マージコミットが残らない githubの最新の情報を取得したいだけの時リベース型を使うのがおすすめ。

11.0.6. プルをリベース型に設定する

```
git config --global pull.rebase true

# masterブランチでpullする時のみ設定する場合
git config branch.master.rebase true
```

12. コミットの修正

12.0.1. 直前のコミットをやりなおす

```
git commit --amend
```

12.0.2. 複数のコミットをやりなおす

```
git rebase -i <コミットID>
git rebase -i HEAD~3

# 「i」は「interactive」の略
# 対話的リベースといって、やりとりしながら履歴を変更していく。
# 「HEAD~」とする事でHEADを基点に数値分の親コミットを指定する。
# 「HEAD^」とする事でHEADを基点にマージした場合の２つ目の親を指定できる。

① git rebase -i HEAD~3 を実行するとエディタが立ち上がって
直前のコミットが３つ表示される。
（順番が git log とは逆なので注意）

pick gh21f6d ヘッダー修正
pick 193954d ファイル追加
pick 8agha0d README修正

② やり直したいコミットの「pick」の部分を「edit」に変更する。
③ 保存してエディタを終了する。
④ editのコミットのところまでコミットの適用が止まる。
⑤ ファイルの内容を修正
⑥ ファイルをステージに追加
⑦ git commit --amend コマンドで修正
⑧ git rebase --continue コマンドで次のコミットへいく
⑨ 「pick」だとそのままコミット内容を適用して次へ進む。
```

12.0.3. 指定したコミットを削除する

```
git rebase -i HEAD~3

pick hncjk7d タイトルを修正
pick ksn6j4k 画像を追加
pick bgmk73h 背景画像を変更

# 消したいコミットを行毎削除すればOK！
```

12.0.4. 指定したコミットを並び替える

```
git rebase -i HEAD~3

pick hncjk7d タイトルを修正
pick ksn6j4k 画像を追加
pick bgmk73h 背景画像を変更

# 並び替えたい行を入れ替えればOK！！
```

12.0.5. 指定したコミットを一つにまとめる

```
git rebase -i HEAD~3

pick hncjk7d タイトルを修正
squash ksn6j4k 画像を追加
squash bgmk73h 背景画像を変更

#  pick を squash に変えれば直前のコミットと一つにまとまる。
#  squash の場合はコミットメッセージの入力が求められる。
```

12.0.6. コミットを分割する

```
git rebase -i HEAD~3

pick hncjk7d タイトルを修正
pick ksn6j4k 画像を追加
edit bgmk73h index.html と style.css を変更

①分割したい行の「pick」を「edit」に変更する

git reset HEAD^
git add index.html
git commit -m "index.htmlを修正"
git add style.css
git commit -m "style.cssを修正"
git rebase --continue

# git reset とはコミットを取り消してステージングしていない状態に戻すコマンド。
# HEAD^ は edit に変えたコミットを指す。
```

13. タグ

コミットを参照しやすくするために、分かりやすく名前を付けるのがタグ。 よくリリースポイントに使います。

13.0.1. タグの一覧を表示する

```
git tag

# パターンを指定してタグを表示
git tag -l "201705"
20170501_01
20170502_01
20170502_02
```

13.0.2. タグには２つの種類がある

・注釈付き版（annotated） ・軽量版（lightweight）

13.0.3. 注釈付き版のタグの作成

```
git tag -a <タグ名> -m "<メッセージ>"
git tag -a 20200203_01 -m "version 20200203_01"
```

13.0.4. 軽量版のタグの作成

```
git tag <タグ名>
git tag 20200203_01
```

13.0.5. 昔のコミットにタグを付ける

```
git tag -a <タグ名> <コミットID> -m "<メッセージ>"
git tag -a 20200203_01 hfk9dsh -m "version 20200203_01"

git tag <タグ名> <コミットID>
git tag 20200203_01 hfk9dsh
```

13.0.6. タグのデータを表示する

```
git show <タグ名>
git show 20200203_01
```

13.0.7. タグをリモートリポジトリに送信する方法

```
git push <リモート名> <タグ名>
git push origin 20200203_01

# タグを一斉に送信する
git push origin --tags
```

13.0.8. githubでタグを確認する

Codeタブ → Releaseタブ →　Tags

14. スタッシュ

作業を一時避難するコマンド 急に別の作業をすることになった場合に、現在のワーキングツリーとステージの内容を避難させる事ができる。

```
git stash

#メッセージを残したい場合
git stash save "メッセージ"
```

14.0.1. スタッシュした内容を確認する

```
git stash list

# 下記のようにスタッシュした数だけ表示される
stash@{0}
stash@{1}
stash@{2}
```

14.0.2. スタッシュした作業を復元する

```
# 最新の作業を復元する
git stash apply

# ステージの状況も復元する
git stash apply --index

# 特定の作業を復元する
git stash apply <スタッシュ名>
git stash apply stash@{1}
```

14.0.3. スタッシュした作業を削除する

```
# 最新の作業を削除する
git stash drop

# 特定の作業を削除する
git stash drop <スタッシュ名>
git stash drop stash@{1}

# 全作業を削除する
git stash clear
```

15. その他

15.0.1. コマンドにエイリアスをつける

```
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.br branch
git config --global alias.co
```

- エイリアスをつける事で設定が楽になる
- git config は設定を変更するコマンド
- `--global` をつけるとPC全体の設定を変更するコマンドになる
- `--global` を付けないと特定のプロジェクトにしか反映されない。
- 今回の省略コマンドは全てのプロジェクトで使用するので--grobalをつける

15.0.2. Gitで管理したくないファイルを外す

- パスワード情報などが記載されているファイル
- windwsやmacで自動生成されるファイル
- キャッシュ
- チーム開発に必要ないファイル

.gitignoreファイルに指定する

```
"4つの指定方法"

# 指定したファイルを除外
index.html

#ルートディレクトリを指定
/root.html

# ディレクトリ以下を除外
dir/

# /以外の文字列にマッチ「*」
/*/*.css
```

以下のコマンドと同じ