

静的サイトジェネレータで 楽しく創る ウェブサイト

アトリエ未来 [著]



第四版

Middlemanは
静的サイトジェネレータです
部品を共用してサクサク制作できます
高級言語 SlimとSassも学べます

静的サイトジェネレータで 楽しく創るウェブサイト

[著] アトリエ未来

令和五年三月三日

■免責

本書は情報の提供のみを目的としています。

本書の内容を実行・適用・運用したことで何が起きようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。

また本書では、™、®、©などのマークは省略しています。

始めに

楽しいプログラミングの世界へようこそ。

情報技術に囲まれた生活を送るわたくしたち。多くの先人が築いた歴史の上に今日があります。^{こんにち}

ウェブサイトには、ヘッダーやナビゲーションメニュー、フッターなど、共通する部品が多数あります。こういった部品を纏めて管理できると作成が楽になります。

本書では、Ruby 製の 静的サイトジェネレータ Middleman を使ったウェブサイト作成法をご紹介いたします。さらに HTML に代えて Slim を、CSS に代えて Sass を使うと制作が楽になりますので、記法等のご案内をしています。また簡易記法として知られる Markdown と、CSS 設計手法として Flou をご紹介しています。

さらに巻末には、習得したい基本的なショートカット&コマンドを付けました。 ^{*1}

現代の魔法、それがプログラミングです。自由自在にコンピュータを操って、幸せな未来へと大きく羽ばたいていってください。

♣ 対象読者

簡単なウェブサイトの作成ができる、初心者の方を想定しています。楽しく快適にサクサク創ることのお役に立てば幸いです。

♣ プログラムのダウンロード

この本で紹介した Middleman の為の テンプレート 「水芭蕉 (mizubasho) ^{*2}」 を、GitHub 上 ^{*3}で公開していますので、ご利用下さい。

♣ 謝辞

Re:VIEW Starter^aを用いて、快適に執筆することができました。作者の kauplan さんに厚く御礼申し上げます。

^a <https://kauplan.org/reviewstarter/>



^{*1} コンピュータとして Mac をお使いの方を対象に執筆しておりますので、Linux や Windows をお使いの方は、キーボード操作等、一部異なる箇所がございます。適宜読み替えていただければ幸いです。

^{*2} 水芭蕉の花言葉 (<https://hananokotoba.com/mizubasho/>)

^{*3} <https://github.com/Atelier-Mirai/mizubasho>

表紙絵の女の子は、千葉県松戸市在住のフリーランス SD イラストレーター 早瀬ひろむ^aさんの作品です。素敵なイラストを描いてくださいり、ありがとうございます。

また、背景の砂浜と海の絵は、T.H^bさんの作品です。砂浜で遊ぶ童心に還り愉しくなります。



^a <https://hiromu-hayase.tumblr.com>

^b <https://www.ac-illust.com/main/profile.php?id=lHPQ0AoI>

早瀬ひろむ さん

♣ 著者紹介



卓越した技能を有する者として認められる国家資格「応用情報技術者」を保持。平成 30 年より「アトリエ未来^a」を創業。HTML 講座や Ruby 講座などプログラミングの個人指導や、IT パスポート講座等の資格講座の開催、ウェブサイト作成等を受注している。

趣味の将棋は、日本将棋連盟より三段の免状を允許。日本の美しい自然や豊かな精神性を宿す熊野古道を歩くことや、花に囲まれた日々を愛している。

^a <https://atelier-mirai.net/>

【コラム】金の延棒クイズ

二進数の不思議を感じる、ちょっと豊かな気分に成れるクイズです。

(正解はこの本のどこかにあるよ。最後まで読んでね。)



七日の給料の支払いとして金の延べ棒が一本あります。これを使って仕事をしてくれる方への日当を支払いたいと思います。

六回鋏を入れ七等分すれば日払いできますが、金の延べ棒を六回も切り取るのは大変です。

必要最小限の二回切り取ることをしたいですが、どことどこを切れば良いでしょうか？

目次

始めに

i

第 1 章 導入	1
1.1 作成例のご紹介	2
1.2 静的サイトジェネレータの利便性	3
1.3 Middleman ^{ミドルマン} のご紹介	3
1.4 コード例の紹介	4
第 2 章 静的サイトジェネレータ Middleman	7
2.1 インストール	8
2.2 新しいサイトの作成	12
2.3 離型	13
2.4 ディレクトリ構造	14
2.5 開発サイクル	17
2.6 ビルド & デプロイ	18
2.7 Frontmatter (前付け)	20
2.8 テンプレート言語	22
2.9 ヘルパーメソッド	23
2.10 レイアウト	32
2.11 パーシャル (部分・断片ファイル)	34
2.12 パーシャルに変数を渡す	36
2.13 データファイル	37
2.14 パーシャルやヘルパなど 様々な手法でのタグ生成	40
2.15 動的ページ	44
2.16 ファイルサイズ最適化	46
2.17 画像形式の変換	50
2.18 画像の遅延読み込み	53
第 3 章 進化した HTML 記述言語 Slim	57
3.1 Slim の 特徴	58
3.2 イントロダクション	59
3.3 ラインインジケータ	60
3.4 HTML タグ	62
3.5 テキストの展開	66

3.6	埋め込みエンジン	66
3.7	Slim の設定	67
第 4 章	構文的に優れたスタイルシート Sass	69
4.1	Sass の特徴	70
4.2	Sass の形式	71
4.3	一行コメントを使う	71
4.4	変数	72
4.5	入れ子 その 1	72
4.6	入れ子 その 2	74
4.7	スタイルファイルの分割管理	75
4.8	ミックスイン	75
4.9	数値の操作	80
4.10	オリジナル関数の定義	81
第 5 章	軽量マークアップ言語 Markdown	85
5.1	Markdown の特徴	86
5.2	段落	86
5.3	改行	86
5.4	テキストの装飾	86
5.5	見出し	87
5.6	箇条書き	87
5.7	引用	88
5.8	水平線	88
5.9	リンク	89
5.10	画像	90
5.11	表	90
5.12	コード	91
5.13	チェックボックス	91
5.14	注釈	91
5.15	HTML	92
第 6 章	簡潔な CSS 設計 FLOU	93
6.1	どんな設計があるの？	94
6.2	F、L、O、U の 4 つに分けよう	95
6.3	FLOU 設計のメリットは？	99
6.4	FLOU で書くときのポイントは？	102
6.5	まとめ	103

付録 A 基本的な ショートカット & コマンド	107
A.1 Mac の為の ショートカット	107
A.2 Atom の為の ショートカット	108
A.3 基本的なコマンド一覧	108
終わりに	113

第 1 章

導入

ウェブサイトを作成する際、少し大きなサイトになると、ヘッダーやフッターなど、共通する部品を纏めて楽に管理したくなります。また、素の HTML や CSS で、構築するのは辛くなりますので、 Slim や Sass など、より効率的な言語を使いたくなるものです。こういったときに活躍するのが、「静的サイトジェネレータ(生成機)」です。

静的サイトジェネレータとして知られる多くのシステムがありますが、ここでは、Ruby^{*1}ベースでとても使いやすい、Middleman をご紹介いたします。

【この章の内容】

1.1	作成例のご紹介	2
1.2	静的サイトジェネレータの利便性	3
1.3	Middleman ^{ミドルマン} のご紹介	3
1.4	コード例の紹介	4

^{*1} Ruby とは、まつもとゆきひろ さんが考案した、柔軟性に富む優れた書き味で人気のプログラミング言語です。オブジェクト指向スクリプト言語 Ruby(<https://www.ruby-lang.org/ja/>)

1.1 作成例のご紹介

水芭蕉

夏の思い出 ギャラリー キッチンシング



春過ぎて
夏來にけらし
衣干すてふ
持統天皇
白妙の
天の香見山

夏の思い出

夏が来れば 思い出す
はるかな尾瀬 とおい空
霧のなかに うかびくる
やさしい影 野の小路
水芭蕉の花が 咲いている
夢見て咲いている 水のほどり
石楠花色に たそがれる
はるかな尾瀬 遠い空

夏が来れば 思い出す
はるかな尾瀬 野の旅よ
花のなかに そよそよ
ゆれゆれる 浮き島よ
水芭蕉の花が 句っている
夢見て句っている 水のほどり
まなこつぶれば なつかしい
はるかな尾瀬 遠い空



梅雨前に訪れた尾瀬の風景
(クリックすると大きくなります)

© 令和四年 水芭蕉

持続天皇の御製「春過ぎて夏來にけらし 白妙の衣干すてふ 天の香具山」。百人一首でもよく知られています。当時の奈良を偲ぶ蓮の花咲くヒーローイメージが目を引きます。

尾瀬の「水芭蕉」を謳った唱歌「夏の思い出」を紹介しています。^{*2}

1.2 静的サイトジェネレータの利便性

作成例に相応しく簡潔に、「ヘッダー、ナビゲーションメニュー、コンテンツ、フッター」から成り立っています。

慣れている方であれば、HTML で文書構造を、CSS で配置と装飾を、JavaScript で動的表現をと、書き下してすぐに完成させられることと思います。

そしてこのような簡潔なウェブサイトであっても、各ページの HTML や CSS は多数の行数となり、管理が大変になります。「ヘッダー」一つの修正であっても、全ページの更新が必要ですし、長大な HTML / CSS から目的箇所を見いだすのも至難の業となります。

「ヘッダー」「ナビゲーションメニュー」など、個々の部品ごとに、HTML や CSS を管理できたらどうでしょうか。とっても楽に管理できるのではないかでしょうか。

また、素の HTML は `<h1></h1>` などと開始タグや終了タグが煩わしく、CSS も `h1 { font-size: 48px; } h1 a { text-decoration: none; }` と個々の要素ごとに記述するのも大変です。

このような状況を解決するのが「静的サイトジェネレータ」です。

1.3 ミドルマン Middlemanのご紹介

静的サイトジェネレータとして知られる多くのシステムがありますが、ここでは、Ruby ^{*3} ベースでとても使いやすい、Middleman をご紹介いたします。

middleman とは、「周旋人」「仲人」「媒介者」という意味です。「ヘッダー」「ナビゲーションメニュー」など部品ごとに、素の HTML や CSS に代えて、Slim や Sass で書いたソースコードを、完成形の HTML ファイルに変換、取り持ってくれます。

^{*2} 水芭蕉は里芋科の多年草で、尾瀬を始め日本各地で群生しています。静的サイトジェネレータとして紹介する Middleman に因み、「み」から始まる花として選びました。

^{*3} Ruby とは、まつもとゆきひろさんが考案した、柔軟性に富む優れた書き味で人気のプログラミング言語です。オブジェクト指向スクリプト言語 Ruby (<https://www.ruby-lang.org/ja/>)

1.4 コード例の紹介

♣ ヘッダー

ヘッダー部分を HTML に代えて Slim で記述しています。開始タグや終了タグが取れ、CSS ライクにクラス名の記述もでき、すっきりしています。

▼ _header.slim

```
header
  a.rainbow.gradation.text href="index.html" 水芭蕉
nav
  ul
    li
      a href="#omoide"
        i.fa-brands.fa-envira.fa-lg.fa-fw
        | 夏の思い出
```

CSS に代えて SCSS で記述しています。「入れ子」にしたり変数名も使用できます。

▼ _header.scss

```
header {
  font-size: clamp(48px, 12vw, 54px);
  font-weight: bold;

  a {
    color: inherit;
    text-decoration: none;
  }

  @media (min-width: 768px) {
    justify-self: start;
  }
}
```

♣ レイアウトファイル

ウェブサイトの各ページの構造は、ほぼ共通しています。そのため、枠組みとなるレイアウトファイルを用意し、コンテンツ部分のみを流し込めるようにすると、開発効率が上がります。

以下の例をご覧ください。

```
doctype html
html
  head
    meta charset="utf-8"
    title = current_page.data.title || "水芭蕉"
    meta content="width=device-width" name="viewport"
```

```

/ ファビコン & アップルタッチアイコン
= favicon_tag 'favicon.ico'
= favicon_tag 'apple-touch-icon-180x180.png',
  rel: 'apple-touch-icon', sizes: '180x180', type: 'image/png'

/ 検索エンジン用の紹介文
meta content="夏の思い出 水芭蕉を謳います" name="description"

/ スタイルシートやスクリプトファイルを読み込む
= stylesheet_link_tag "style"
= javascript_include_tag "site"

body
/ 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'
```

<head>タグには、ウェブページの設定事項いろいろを記述していくますが、文字コードやビューポートなど概ね全てのページに共通しています。

また、<body>タグに書かれる事柄も、「ヘッダー」や「フッター」など各ページで共通するコードに関しては = partial 'header' と、先に紹介した 部分ファイル _header.slim を呼び出し、描画するようにすると、変更があった際の修正も容易で、また他のサイトを作成する際にも使い回しが利くなどの利点が得られます。

= yield は、各ページの「コンテンツ」に置換されます。トップページであれば、index.html.slim を、お問い合わせページであれば、contact.html.slim を書くことのみに集中できます。

♣ トップページ

作成例のトップページは「ヘッダー、ナビゲーションメニュー、コンテンツ、フッター」から成り立っています。

この中の「ヘッダー、ナビゲーションメニュー、フッター」に関しては、全てのページに共通する部分として、レイアウトファイルで既に描画していますので、トップページでは、「コンテンツ」部分のみを集中して書くことができます。

コンテンツは、ヒーローイメージと夏の思い出という記事部分に別れています。纏めて書くこともできますが、部品ごとに分けると管理もしやすく見通しも良くなりますので、詳細は三つの部分ファイル _hero.slim と _omoide.slim に書き、それぞれを呼び出し、描画するようにします。

▼ index.html.slim

```
/ ヒーローイメージ  
= partial 'hero'  
  
/ 夏の思い出  
= partial 'omoide'
```

▼ _hero.slim

```
.hero  
  figure  
    = image_tag "kaguyama.webp", alt: "藤原宮と蓮、天香久山"  
  h1  
    span.lefted.text.ml-3  
      = link_to "持統天皇", "https://ja.wikipedia.org/wiki/持統天皇",  
                class: "gold gradation text"  
  
    span.lefted.silver.gradation.glow.text 春過ぎて  
    span.lefted.silver.gradation.glow.text 夏來にけらし 白妙の  
    span.centered.silver.gradation.glow.text 衣干すてふ 天の香具山  
  
  = javascript_include_tag "glow-text.js"
```

▼ _omoide.slim

```
section.omoide#omoide  
  h2.gold.title 夏の思い出  
  article  
    p  
      | 夏が来れば 思い出す  
    br  
      | はるかな尾瀬 とおい空  
  
  figure.rotate-3.frame.gallery  
    = link_to "images/oze_mizubasho.webp", title: "梅雨前に訪れた尾瀬の風景" do  
      = image_tag "oze_mizubasho.webp", alt: "尾瀬と水芭蕉"  
    figcaption  
      | 梅雨前に訪れた尾瀬の風景
```

以上、簡単ではありましたが、静的サイトジェネレータ「Middleman」を使ってのウェブサイト作成方法がイメージできたのではないでしょうか。

大まかな全体像をご紹介したところで、次章以降でより詳細をご案内して参ります。

第 2 章

静的サイトジェネレータ ミドルマン Middleman

静的サイトを構築する使いやすいフレームワーク それが「Middleman」です。
インストールから始まり、様々な便利な機能の紹介を、サンプルコードとともにに行っていきます。

【この章の内容】

2.1	インストール	8
2.2	新しいサイトの作成	12
2.3	雛型	13
2.4	ディレクトリ構造	14
2.5	開発サイクル	17
2.6	ビルド & デプロイ	18
2.7	Frontmatter (前付け)	20
2.8	テンプレート言語	22
2.9	ヘルパーメソッド	23
2.10	レイアウト	32
2.11	パーシャル (部分・断片ファイル)	34
2.12	パーシャルに変数を渡す	36
2.13	データファイル	37
2.14	パーシャルやヘルパなど 様々な手法でのタグ生成	40
2.15	動的ページ	44
2.16	ファイルサイズ最適化	46
2.17	画像形式の変換	50
2.18	画像の遅延読み込み	53

2.1 インストール

公式ガイドは、Middleman: 作業を効率化するフロントエンド開発ツール^{*1} です。ここから抜粋、説明を加えながら、ご案内していきます。

Middleman は RubyGems のパッケージマネージャを使って配布されます。つまり Middleman を使うには Ruby のランタイムと RubyGems の両方が必要だということです。

Mac 利用者の方は、以下に従い、インストールしてください。^{*2}

♣ Command Line Tools のインストール

macOS では ソースコードのコンパイルに Xcode の Command Line Tools が必要です。ターミナルアプリ^{*3} より以下のコマンドを実行します。

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

♣ HomeBrew のインストール

HomeBrew^{*4} は、macOS（または Linux）用パッケージマネージャーです。

HomeBrew を使うと、開発に必要な様々なプログラムを簡単にインストールすることが出来ます。公式サイトを始め、使い方について書かれた様々なサイトがございますので、詳細はそちらをご覧ください。

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

♣ rbenv のインストール

rbenv は、Ruby のバージョン管理を行うためのツールです。Ruby は毎年クリスマスにメジャー バージョンが提供されます。最新版を用いることが好ましいですが、様々な事情により、以前のバージョンを使いたい場合もあります。そこで、それぞれのプロジェクトに応じた Ruby のバージョンを仕様できるよう、rbenv をインストールします。

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

rbenv がインストールできたら、パスを通します。「パスとは、小道、道筋、進路、通り道などの意味を持つ英単語で、IT の分野では、コンピュータ内で特定の資源の所在を表す文字列のことをパ

^{*1} <https://middlemanapp.com/jp/>

^{*2} Windows 利用者の方は、公式インストールビデオ (<https://youtu.be/nNc5Pm4IYeE>) が公開されていますので、そちらをご覧ください。

^{*3} より高性能な、iTerm2 (<https://www.iterm2.com>) がお薦めです。

^{*4} homebrew (https://brew.sh/index_ja)

ス」^{*5}と言います。パスを通すことにより、インストールしたプログラムをコマンドラインから実行できるようになります。

```
# シェルに bash を使っている場合
$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bash_profile

# シェルに zsh を使っている場合
% echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.zshrc
```

rbenv のインストールが終わったら、初期設定を行います。

```
$ rbenv init
```

`eval "$(rbenv init -)"`を、`~/.bash_profile` または `~/.zshrc` に追記するよう、メッセージが表示されますので、お使いのエディタで、`~/.bash_profile` または `~/.zshrc` を開き、末尾に追記します。

ここまでで、rbenv のインストールは完了です。

一旦ターミナルを閉じて、再度開きます。

♣ Ruby のインストール

```
$ rbenv install -l
```

と、コマンド入力すると、インストール可能な Ruby のバージョンが次のように表示されます。

```
rbenv install -l
2.6.10
2.7.6
3.0.4
3.1.2
jruby-9.3.6.0
mruby-3.1.0
picoruby-3.0.0
rbx-5.0
truffleruby-22.2.0
truffleruby+graalvm-22.2.0
```

```
Only latest stable releases for each Ruby implementation are shown.
Use 'rbenv install --list-all / -L' to show all local versions.
```

様々な Ruby のバージョンがあることが分かります。

ここでは一番新しい 3.1.2 をインストールすることにしましょう。

```
$ rbenv install 3.1.2
```

^{*5} 出典: IT 用語辞典 (<https://e-words.jp/w/%E6%A0%A1.html>)

しばらくすると、インストールが完了します。

♣ RubyGems とは

RubyGems は、Ruby 言語用のパッケージ管理システムであり、Ruby のプログラムと ("gem" と呼ばれる) ライブラリの配布用標準フォーマットを提供している。gem を容易に管理でき、gem を配布するサーバの機能も持つ。^{*6}

有志の方々により Ruby でつくられたパッケージ (=便利なプログラム) は、RubyGems 公式サイト^{*7} で公開されており、様々なパッケージが提供されています。

ここでは、複数の gem を用いる際に定番となっている、Bundler と、人気の フレームワーク Ruby on Rails のインストール方法をご紹介いたします。

♣ Bundler のインストール

RubyGems に登録されているところによると、

Bundler は、必要な gems とバージョンを正確に追跡してインストールすることで、Ruby プロジェクトに一貫した環境を提供します。

とのことです。

あるプロジェクトでは、X, Y という二つの gem を使います。そして、X という gem が機能するためには、A という別の gem の特定のバージョンを必要としており、Y という gem を機能させるためには、B という別の gem の特定のバージョンを必要とするような状況を考えてみましょう。

それぞれの gem が要求するバージョンを適切に管理していくのは、プロジェクトで用いる gem が増えるに従い大変となるのが容易に想像できます。

Bundler を導入すると、適切な gem が使われるよう、バージョン管理を自動で行うことが出来ます。

Bundler 自身も gem として提供されていますので、次のコマンドでインストール出来ます。

```
$ gem install bundler
```

.....

【おまけ】Ruby on Rails のインストール

本書では、登場しませんが、人気のウェブフレームワーク Ruby on Rails を使いたい方もいらっしゃることと思います。

```
$ gem install rails
```

でインストールできます。

Middleman と Ruby on Rails は、良く似ています。

^{*7} <https://rubygems.org>

- 静的サイトなら Middleman
 - 動的サイトなら Ruby on Rails
- と、容易に使い分けられることでしょう。
-

♣ Middleman のインストール

準備が整いましたので、次のコマンドを実行してください。

```
$ gem install middleman
```

これで、Middleman のインストールは完了です。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ます。

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

【コラム】Bundler を使って gem のバージョン管理を行っている場合

gem のバージョン依存関係を管理するために、Bundler を使う旨、お話をいたしました。Bundler を使う場合、Gemfile というファイルの中に、使いたい gem の名前と、(必要があれば) その gem のバージョン指定を書きます。そして次のコマンドにより、Gemfile に記述された gem インストールします。

```
# Bundler を使って Gemfile に記述された gem インストールする。  
$ bundle install
```

```
% bundle install
Fetching gem metadata from https://rubygems.org/.....
Resolving dependencies...
Using concurrent-ruby 1.1.9
Using bundler 2.3.5
(略)
Using middleman-core 4.4.2
Using middleman 4.4.2
Using middleman-autoprefixer 3.0.0
Using middleman-livereload 3.4.7
Using middleman-minify-html 3.4.1
Installing in_threads 1.6.0
Using image_optim 0.25.0
Using image_optim_pack 0.2.3
Using middleman-imageoptim 0.3.0
Bundle complete! 8 Gemfile dependencies, 64 gems now installed.
Use `bundle info [gemname]` to see where a bundled gem is installed.
```

とメッセージが表示されます。 `middleman 4.4.2` と表示されていますので、Middleman のバージョン 4.4.2 が使われることが確認できます。

このコマンドにより、各 gem の依存関係を解決するよう、適切なバージョンの gem が Bundler により、インストールされ、使用する様々な gem のバージョンを記した `Gemfile.lock` というファイルが作られます。

Bundler 管理下で、Middleman を使う場合には、以下のコマンドを用います。

```
# 新たに Middleman を使ったサイトを作成する。
$ bundle exec middleman init

# コーディング中に、ブラウザで結果を確認する。
$ bundle exec middleman server

# 公開用のサイトデータを出力する。
$ bundle exec middleman build
```

(先頭に `bundle exec` が付いていることに着目です。)

2.2 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ディレクトリを作る必要があります。

- 既存ディレクトリを、Middleman で開発できるようにする場合には、ターミナルから以下のコマンドを実行してください。

```
$ middleman init
```

2. 新規に、Middleman で開発するディレクトリを作成する場合には、ターミナルから以下のコマンドを実行してください。

```
$ 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ます。

`source` ディレクトリは、ウェブサイト構築用に、slim や sass 等のソースファイルを置くディレクトリです。

`config.rb` は、Middleman の設定を行うためのファイルです。`Gemfile` は、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です。

2.3 雛型

そのままでも使えますが、より便利に使えるよう、調整した雛型を作成しましたので、ご活用下さい。

`config.rb` や `Gemfile` の設定が済んでおり、作りやすいよう `index.html` の雛形などが同梱されています。

```
$ bundle exec middleman init my_new_site -T Atelier-Mirai/mizubasho
```

`my_new_site` というディレクトリが作られています。以下のコマンドで、`my_new_site` ディレクトリに移動しましょう。

```
$ cd my_new_site
```

今後の作業は、このディレクトリ内で行っていきます。

2.4 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/                # ウェブサイト公開用のHTMLなどが
|                           出力されているディレクトリ
+-- 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/               # 定型的なHTMLを簡単に記述するために
|                           自作したプログラムを置くディレクトリ
+-- source/               # ウェブサイト用のソースファイルを置く
|   +-- images/            # 画像ファイル用のディレクトリ
|   +-- index.html.slim    # slimで記述したトップページのファイル。
|                           コンパイルすると、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
|       +-- style.css.scss
```

♣ 主要なディレクトリ

Middleman は特定の目的のために source, build, data と lib ディレクトリを利用します。各ディレクトリは Middleman のルートディレクトリに存在します。

source ディレクトリ

source ディレクトリには利用するテンプレートの JavaScript、CSS や画像を含む、ビルドされる ウェブサイトのソースファイルが置かれます。

build ディレクトリ

build ディレクトリは静的サイトのファイルがコンパイルされ出力される ディレクトリです。

data ディレクトリ

ローカルデータ機能によって data ディレクトリの中に YAML や JSON ファイルを作成し、これらのファイルの情報をテンプレートの中で利用することができます。data フォルダはプロジェクトの 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。詳細については ローカルデータ を確認してください。

helper ディレクトリ

helper ディレクトリには、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 独自のヘルパやクラスを追加することができます。

それぞれのファイルは、次のように記述します。(雛形を導入すると、既に記述されています。)

▼ Gemfile

```

source "https://rubygems.org"

# 静的サイトジェネレータ Middleman
gem "middleman"
gem "webrick"

# ベンダープリフィックスを自動付与する
gem "middleman-autoprefixer"

# ファイル更新の際に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としてSlimを使用する
# (Slim v4.1.0を用いることを指定しています。)
# (Middlemanの不具合により、より新しいSlimは使用不可です。)
gem "slim", "4.1.0"

# HTML圧縮を行う
gem "middleman-minify-html"

# html_builder(DSL)用にnokogiriも入れる
gem "nokogiri"

# img タグに width属性, height属性を付与する為に
gem "fastimage"

```

▼ config.rb

```

# JavaScript圧縮用ライブラリを読み込む
require "uglifier"

# コード変更すると、自動再読み込みされる
activate :livereload

# 相対URLを使う
activate :relative_assets
set :relative_links, true

# ベンダープリフィックスを自動的に付与する
activate :autoprefixer do |prefix|
  prefix.browsers = "last 2 versions"
end

# レイアウトファイルの指定
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,
compressor: proc {
  ::Uglifier.new(
    mangle: { toplevel: true },
    compress: { unsafe: true },
    harmony: true
  )
}
# アセットファイルのURLにハッシュを追加
activate :asset_hash
# テキストファイルのgzip圧縮
activate :gzip
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
#   # (html, css, javascript の圧縮も無効化すると、
#   # 学習用に読みやすいHTMLが出力される)
#   pretty: true, sort attrs: false
# }

# 動的サイトの設定例
# data.cats.each do |neko|
#   proxy "/#{neko.name}.html", "/neko_template.html",
#         locals: { neko_data: neko }, ignore: true
# end

```

▼ source/layouts/site.slim

```

doctype html
html
  head
    meta charset="utf-8"
    meta name="viewport" content="width=device-width"
    title
      = current_page.data.title || "さくら商会"
      = stylesheet_link_tag "style"

  body
    = yield

```

2.5 開発サイクル

♣ Middleman server

Middleman は開発のスタート時点から開発コード^{*8}とプロダクションコード^{*9}を分離します。これにより開発中に Slim, Sass など開発に有益なツールを利用することができます。

Middleman を使う時間の大半は開発サイクルになります。コマンドラインから、プロジェクトディレクトリの中でプレビューウェブサーバを起動してください。

```
$ bundle exec middleman server
```

このコマンドはローカルの ウェブサーバを起動します。

```
$ bundle exec middleman
== The Middleman is loading
== LiveReload accepting connections from ws://192.168.0.4:35729
== View your site at "http://Mac-mini.local:4567", "http://192.168.0.4:4567"
== Inspect your site configuration at "http://Mac-mini.local:4567/_middleman"
, "http://192.168.0.4:4567/_middleman"
```

と、表示されていれば、無事にサーバーが起動しています。

.....

```
(略)
== Port "4567" is in use. This should not have happened. Please start "middleman server" again.
bundler: failed to load command: middleman (/Users/mirai/.rbenv/versions/3.1.0
/bin/middleman)
(略)
```

とエラーメッセージが表示されるときには、他のターミナルで既に Middleman が起動しています。他のターミナルで起動している Middleman を終了してから、再度挑戦してください。

.....

ブラウザを開き、<http://localhost:4567/> にアクセスして下さい。それから、source ディレクトリ内で、ファイルを作成や編集して下さい。プレビューウェブサーバが起動しているおかげで、ブラウザ上で編集内容を確認できます。

プレビューウェブサーバを停止したい場合には、コマンドラインから Ctrl + C (Ctrl キーと C キーを同時に押す) で、停止できます。

.....

^{*8} 開発コードとは、作成中に用いるコードのことです。

^{*9} プロダクションコードとは、最終的に出来上がってウェブサイトの公開するためのコードのことです。

サーバー起動中は、このターミナルウィンドウからはコマンド入力が行えません。Git 操作等のコマンド入力は、もう一つ別のターミナルウィンドウ（タブ）を開き、そちらのウィンドウ（タブ）から行います。

.....

♣ LiveReload (自動再読み込み)

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機能が実装されています。まず Gemfile に middleman-livereload を追記してください。（雛形の Gemfile には既に記述済です）

▼ 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。（雛形の config.rb には既に記述済です）

▼ config.rb

```
activate :livereload
```

これでページ内容に変更があると、自動的にブラウザが再読み込み（リロード）されます。

2.6 ビルド & デプロイ

♣ 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、middleman build を実行してください。

```
$ bundle exec middleman build
```

.....

```
% bundle exec middleman build
bundler: failed to load command: middleman (/Users/mirai/.rbenv/versions/3.1.2
/bin/middleman)
```

とエラーメッセージが表示される場合には、Bundler による gem ファイルのバージョン管理が行われていない状態です。

```
% bundle install
```

として、Gemfile に記述した gem をインストールして下さい。

.....

このコマンドは `source` フォルダにあるファイル毎に静的ファイルを作ります。Slim ファイルや Sass ファイルがコンパイルされ、ファイル圧縮などの機能が実行されます。そして、ビルドが完了すると、必要なものはすべて `build` ディレクトリに用意されます。

```
% bundle exec middleman build
:image_optim is deprecated. Please use `:imageoptim` instead.
YAML Exception parsing /Users/mirai/rails_projects/my_new_site/source/fruits.h
tml.slim: (<unknown>): did not find expected key while parsing a block mapping
at line 1 column 1
    create  build/stylesheets/foundation/modern-css-reset-a537b384.css
    create  build/stylesheets/style-85d5d9f4.css
    create  build/images/tora_s-6b7d4713.jpg
    create  build/images/logo-cd6002e6.jpg
    create  build/images/sea-2adf6e5e.jpg

    create  build/images/tama-ca888398.jpg
    create  build/images/zou-3128b283.jpg
    (略)
imageoptim  build/images/tama-ca888398.jpg (18.62% / 33KiB smaller)
imageoptim  fixed file mode on build/images/tama-ca888398.jpg file to match
source
imageoptim  build/images/apple-touch-icon-180x180-7af643fc.png (12.62% / 7Ki
B smaller)
imageoptim  build/imageoptim.manifest.yml updated
imageoptim  Total savings: 120KiB
    create  build/javascripts/magnific-popup-custom-205d8eda.js.gz (272 Byte
s smaller)
    create  build/javascripts/site-17e02c7a.js.gz (76 Bytes smaller)
    create  build/index.html.gz (5.6 KB smaller)
    create  build/kitchen_sink.html.gz (8.4 KB smaller)
    create  build/fruits.html.gz (49 Bytes smaller)
    create  build/stylesheets/foundation/modern-css-reset-a537b384.css.gz (4
15 Bytes smaller)
    create  build/stylesheets/style-85d5d9f4.css.gz (10.1 KB smaller)
gzip  Total gzip savings: 24.9 KB
Project built successfully.
```

とメッセージが表示されました。

`build` ディレクトリを覗いてみると、`index.html` など、お馴染のファイル達が作られていることが確認できます。

適宜、公開（デプロイ）^{*10}してください。

♣ 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。ハッシュ文字列とは何でしょうか？

ハッシュ文字列を付与しないときには、次のように出力されます。

^{*10} ソースコード管理に Git を、ウェブサイトの公開に Netlify を使用すると、より快適に公開できます。

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

ハッシュ文字列を付与すると、次のようにになります。

```
<link href="/stylesheets/style-85d5d9f4.css" rel="stylesheet">
```

`style.css` が、`style-85d5d9f4.css` となりました。この `style` のあとに付与される文字列のことを、ハッシュ文字列といいます。このハッシュ文字列は、スタイルシートが更新される都度、異なる文字列となります。そのため、(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ウェブ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。

ハッシュ文字列を付与するには、`config.rb` に次のように記述します。(雛形の `config.rb` には既に記述済です)

▼ config.rb

```
activate :asset_hash
```

2.7 Frontmatter (前付け)

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。

Frontmatter はテンプレートの最上部に記述します。そして、行頭から行末まで 3 つのハイフン --- によって、その他の部分から分離して書きます。このブロックの中では、テンプレート内で `current_page.data` ハッシュとして使えるデータを作ることができます。つまり、Frontmatter(前付け)の中で、`title: "大好きな果物の紹介"` と書いたハッシュデータは、テンプレートの中で、`current_page.data.title` として取得できます。

それでは、簡単な使い方の例を紹介します。`fruits.html.slim` を次のように書いてみます。

▼ source/fruits.html.slim

```
---
title: "大好きな果物の紹介"

my_favorite_fruits_list:
- あけび
- かき
- さくらんぼ
---



# 大好きな果物




- current_page.data.my_favorite_fruits_list.each do |fruit|
  li = fruit
```

レイアウトファイル `site.slim` は、次のようになっていますとします。

▼ source/layouts/site.slim

```
doctype html
html
  head
    meta charset="utf-8"
    meta name="viewport" content="width=device-width"
    title
      = current_page.data.title || "さくら商会"
      = stylesheet_link_tag "style"

  body
    = yield
```

そうすると、ビルドして出来上がる html ファイル `fruits.html` は次のようにになります。

▼ build/fruits.html

```
<!DOCTYPE html><html><head><meta charset=utf-8 /><meta content="width=device-width" />
<title>大好きな果物の紹介</title><link href="stylesheets/style-85d5d9f4.css" rel="stylesheet" /></head><body><h1>大好きな果物</h1><ol><li>あけび</li><li>かき</li><li>さくらんぼ</li></ol></body></html>
```

少しでも配信時間を速くできるように、空白等を削除して、生成されています。出来上がったウェブサイトを公開する場合には、これは都合が良いのですが、出力された内容を確認する為に、綺麗に整形しましょう。

手作業で行っても良いのですが、[Syncer^{*11}](#) というサイトがありますので、これを使ってみます。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset=utf-8 />
    <meta content="width=device-width" name="viewport" />
    <title>大好きな果物の紹介</title>
    <link href="stylesheets/style-85d5d9f4.css" rel="stylesheet" />
  </head>
  <body>
    <h1>大好きな果物</h1>
    <ol>
      <li>あけび</li>
      <li>かき</li>
      <li>さくらんぼ</li>
    </ol>
  </body>
</html>
```

Frontmatter(前付け) と、出来上がった `fruits.html` との対応関係を見てください。個々のページによって、タイトルは異なりますが、同じレイアウトファイル(雛形)に、ページのタイトルを渡

^{*11} <https://lab.syncer.jp/Tool/HTML-PrettyPrint/>

すことができています。

また、`my_favorite_fruits_list` として、好きな果物を列挙しましたが、それぞれの果物が Ruby 持つ強力な `each` メソッドにより取り出され、``タグに展開される点にも着目してください。もしも希望ならば、「たんかん」や「なし」など、他の果物も簡単に追加できます。

.....
先ほど紹介したアセットハッシュが付与され `<link href="stylesheets/style-85d5d9f4.css" rel="stylesheet" />` となっています。
.....

2.8 テンプレート言語

テンプレート言語とは、ウェブサイトの作成を簡単にするために用いられる言語のことです。`Slim` や `Sass` が有名です。`Slim` 記法で記述すると `HTML` が、`Sass` 記法では `CSS` が容易に書けるようになるとともに、ページ内で変数やループを使えるようになります。

`Middleman` 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。具体的には、`slim` で書かれた `index` ページは、ファイル名の `index.html` に、さらに `slim` と拡張子を付けた `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。

以下に例を挙げます。

▼ index.html.slim

```
h1 ようこそ
ul
  - 3.times do |num|
    li = "#{num}回目"
```

これをビルドすると、次のようになります。

▼ index.html

```
<h1>ようこそ</h1>
<ul>
  <li>1回目</li>
  <li>2回目</li>
  <li>3回目</li>
</ul>
```

- `3.times do |num| li = "#{num}回目"` は、Ruby での繰り返しの書き方です。

2.9 ヘルパメソッド

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に、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。

♣ リンクヘルパ

リンクを記述する際に、リンクヘルパを使うと簡単に書くことができます。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。

▼ index.html.slim

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

これをビルドすると、次のようにになります。

▼ index.html

```
<a href="https://mysite.com">わたくしのサイト</a>
```

`link_to` メソッド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。

▼ index.html.slim

```
= 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
```

ビルドすると、次のようにになります。

▼ index.html

```
<a href="http://mysite.com">
  
  <p>
    わたくしの会社へようこそ
  </p>
</a>
```

♣ メールヘルパ

リンクが簡単に作成できたように、メールを容易に送信できるようタグを作ってくれる、メールヘルパもあります。

▼ index.html.slim

```
= mail_to 'taro@example.com', "メール"
```

ビルドすると、次のようにになります。

▼index.html

```
<a href="mailto:taro@example.com">メール</a>
```

♣ 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。

▼index.html.slim

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

▼index.html

```

```

♣ アセットヘルパ

<head>タグ内で良く用いられるアセットヘルパとして、次の三種類があります。

favicon_tag

ファビコンを作成する際に、使うと便利なタグです。

▼index.html.slim

```
= favicon_tag "favicon.ico"
= favicon_tag "apple-touch-icon-180x180.png",
  rel: "apple-touch-icon",
  type: "image/png",
  sizes: "180x180"
```

▼index.html

```
<link href="images/favicon.ico" rel="icon" type="image/ico">
<link href="images/apple-touch-icon-180x180.png" rel="apple-touch-icon" type="image/png" sizes="180x180">
```

画像は images ディレクトリに置く約束ですので、`href="images/favicon.ico"` と展開されます。画像ディレクトリのパスが変更された際にも、 config.rb で設定するのみで良いので、保守効率も上がります。

stylesheet_link_tag

スタイルシートを読み込む際に、使うと便利なタグです。

▼ index.html.slim

```
= stylesheet_link_tag "style"
```

▼ index.html

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

スタイルシートは `stylesheet` ディレクトリに置く約束ですので、`href="stylesheets/style.css"` と展開されます。短く簡潔に書くことが出来ます。

javascript_include_tag

スクリプトを読み込む際に、使うと便利なタグです。

▼ index.html.slim

```
= javascript_include_tag "site"
```

▼ index.html

```
<script src="javascripts/site.js"></script>
```

スクリプトは `javascripts` ディレクトリに置く約束ですので、`href="javascripts/site.js"` と展開されます。

stylesheet_link_tag, javascript_include_tag の使い分け

CDN *¹² で配信されるコードには、`link` タグや `script` タグで書き、自分で書いたものにはヘルパタグの利用が便利です。

▼ Magnific Popup ライブラリを使用する際の例

```
/ CDN 配信
link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/magnific-popup.js/1.1.0/magnific-popup.min.css"
script src="https://cdnjs.cloudflare.com/ajax/libs/magnific-popup.js/1.1.0/jquery.magnific-popup.min.js"

/ 自作
= stylesheet_link_tag    "magnific-popup-custom"
= javascript_include_tag "magnific-popup-custom"
```

♣ ページクラス

サイト階層に対応する `body` タグの `class` 属性が生成されると便利です。`page_classes` はこれらの属性名を生成します。`projects/rockets/saturn-v.html.slim` にページがあるとすると、レイアウトには次のように表示されます。

*¹² CDN(コンテンツデリバリネットワーク)とは、Web 上で送受信されるコンテンツをインターネット上で効率的に配信するために構築されたネットワーク。(https://e-words.jp/w/CDN.html)

▼ source/projects/rockets/saturn-v.html.slim

```
body class=page_classes
  h1 サターンV型ロケット
  p 月面着陸を成し遂げたアポロ計画で使われたロケットです。
```

▼ build/projects/rockets/saturn-v.html

```
<body class="projects rockets saturn-v">
  <h1>サターンV型ロケット</h1>
  <p>月面着陸を成し遂げたアポロ計画で使われたロケットです。</p>
</body>
```

これにより簡単にページに projects や rockets、saturn-v のスタイルを適用できます。

♣ 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できます。良く使う html 部品があれば、生成用のヘルパを創ると便利です。

ここではカスタム定義ヘルパの例として、ISBN 番号から Amazon へのリンクを生成するヘルパを作ってみます。

カスタム定義ヘルパは、 helpers ディレクトリに配置すると分かりやすいです。ファイル名は任意ですが、ここでは custom_helper.rb として、作成します。

▼ helpers/custom_helper.rb

```
# Stringクラスの拡張
class String
  def to_amazon_url
    "https://www.amazon.co.jp/dp/#{$self}"
  end

  def to_image
    "#{$self}.jpg"
  end
end

# 本を紹介するためのカスタムヘルパの定義
def introduce(book)
  title = book.title
  desc = book.desc
  isbn = book.isbn
  url = isbn.to_s.to_amazon_url
  image = isbn.to_s.to_amazon_book_image

  t = content_tag :h2, class: 'title' do title end
  i = image_tag(image)
  d = content_tag :p, class: 'desc' do desc end

  content_tag :div, class: 'book' do
    link_to url, target: '_blank' do
      (t + i + d)
    end
  end
end
```

```
    end
  end
end
```

カスタムヘルパが定義できましたので、実際に使ってみます。

▼ source/books.html.slim

```
---
```

title: "大好きな本の紹介"

my_favorite_books_list:

- title: CSSグリッドで作る HTML5&CSS3 レッスンブック
 desc: |-
 CSSグリッドをベースになると、Webページ制作がシンプルになります。
 本書では、サンプルを作りながらステップ・バイ・ステップで学習することにより、
 モバイルファーストで本格的なレスポンシブに対応した実践的なWeb制作に関する知識をひきこみ得ることができます。
 これからHTML5 & CSS3を使ったWebサイトの構築を学びたい人に最適の一冊です。
 isbn: 4802611897
- title: Ruby on Rails 6 実践ガイド
 desc: |-
 本書では、1つの企業向け顧客管理システムを作る過程で、RailsによるWebアプリケーション開発の基礎知識とさまざまなノウハウを習得していきます。各章末には演習問題が設けられているので、理解度を確かめながら確実に読み進められます。
 著者が試行錯誤を繰り返した上でのベストプラクティスを提供し、読者は、実際に業務システムを構築しながらRailsのさまざまな機能、方法、作法、メソッド、テクニックを学ぶことができます。
 isbn: 4295008052

```
---
```

h1 大好きな本

- current_page.data.my_favorite_books_list.each do |book|
 = introduce(book)

これをビルドすると、次のような HTML が出力されます。

▼ books.html

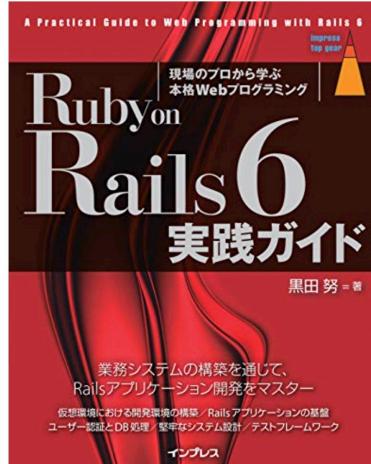
```
<div class="book">
  <a href="https://www.amazon.co.jp/dp/4295008052" target="_blank">
    <h2 class="title">
      Ruby on Rails 6 実践ガイド
    </h2>
    
    <p class="desc">
      本書では、1つの企業向け顧客管理システムを作る過程で、RailsによるWebアプリケーション開発の基礎知識とさまざまなノウハウを習得していきます。各章末には演習問題が設けられているので、理解度を確かめながら確実に読み進められます。著者が試行錯誤を繰り返した上でのベストプラクティスを提供し、読者は、実際に業務システムを構築しながらRailsのさまざまな機能、方法、作法、メソッド、テクニックを学ぶことができます。
    </p>
  </a>
</div>
```

```
</p>
</a>
</div>
```

本書の書名と、ISBN番号を元に取得した本の画像が表示されています。そして画像をクリックするとAmazonへ跳ぶようリンクが設定されています。

良く使う定型的な部品を表示させるとときには、カスタムヘルパを作成することも作業効率化のために役立ちますので、参考になさってください。

Ruby on Rails 6 実践ガイド



本書では、1つの企業向け顧客管理システムを作る過程で、RailsによるWebアプリケーション開発の基礎知識とさまざまなノウハウを習得していきます。各章末には演習問題が設けられているので、理解度を確かめながら確実に読み進められます。著者が試行錯誤を繰り返した上でのベストプラクティスを提供し、読者は、実際に業務システムを構築しながらRailsのさまざまな機能、方法、作法、メソッド、テクニックを学ぶことができます。

【コラム】 YAML 解説

先ほど、Frontmatter(前付け)の紹介の際に使った果物の例を思い出してみましょう。果物の紹介の場合には、次のようにFrontmatter(前付け)を書いていました。

```
---
my_favorite_fruits_list:
- あけび
- かき
- さくらんぼ
---
```

本を紹介する場合には、たくさんの行が現れてきました。もし、本書の書名だけで良ければ、次のように書くことも出来ます。

```
---
my_favorite_books_list:
- CSSグリッドで作る HTML5&CSS3 レッスンブック
- Ruby on Rails 6 実践ガイド
---
```

ここではより詳しい本の属性として「書名」「簡単な内容紹介」「ISBN 番号」を扱いますので、

```
---  
my_favorite_books_list:  
- title: Ruby on Rails 6 実践ガイド  
  desc: |-  
    本書では、1つの企業向け顧客管理システムを作る過程で、RailsによるWebアプリケーション開発の基礎知識とさまざまなノウハウを習得していきます。各章末には演習問題が設けられているので、理解度を確かめながら確実に読み進められます。  
    著者が試行錯誤を繰り返した上でのベストプラクティスを提供し、読者は、実際に業務システムを構築しながらRailsのさまざまな機能、方法、作法、メソッド、テクニックを学ぶことができます。  
  isbn: 4295008052  
---
```

と書いています。すると、

```
- current_page.data.my_favorite_books_list.each do |book|  
  li = book.title  
  li = book.desc  
  li = book.isbn
```

として、それぞれの属性の値を取得、表示させることができます。

`introduce` ヘルパに `book` オブジェクトを渡しているので、`introduce` ヘルパ内で、それぞれの本を紹介するための HTML が作られることとなります。

```
- current_page.data.my_favorite_books_list.each do |book|  
  = introduce(book)
```

♣ html builder

前回は、`content_tag` を使って、カスタム定義ヘルパを作成しました。`content_tag` は、タグの文字列を生成して返してくれるメソッドですが、入れ子になっている場合など、扱いにくいことがあります。Nokogiri という、HTML の解析や生成を行うための gem を使うと、複雑な HTML を作成する場合にも便利ですので、使ってみます。

`Nokogiri::HTML::Builder`^{*13} や、 `Ruby on Rails 6 実践ガイド` 黒田 努 著 p328^{*14} に、より詳しく説明がありますので、お読みください。ここでは適宜引用してご紹介いたします。

まず HTML のマークアップを簡単に行うメソッド `markup` を以下のように定義します。

*13 https://www.oiax.jp/rails/tips/nokogiri_html_builder.html

*14 <https://www.amazon.co.jp/dp/4295008052>

▼ helpers/html_builder.rb

```
# Ruby on Rails 6 実践ガイド 黒田 努 著 p328より引用
module HtmlBuilder
  def markup(tag_name = nil, options = {})
    root = Nokogiri::HTML::DocumentFragment.parse("")
    Nokogiri::HTML::Builder.with(root) do |doc|
      if tag_name
        doc.method_missing(tag_name, options) do
          yield(doc)
        end
      else
        yield(doc)
      end
    end
    root.to_html.html_safe
  end
end
```

これで、`markup` メソッドが使えるようになりました。それでは利用者一覧を表示するための「カスタム定義ヘルパー」を作成します。

▼ helpers/custom_helper.rb

```
def table_of_users(users)
  markup do |m|
    m.table(id: 'users', class: 'ui table') do
      m.tr do
        m.th '姓'
        m.th '名'
        m.th '性別'
      end
      users.each do |u|
        attrs = {}
        attrs[:class] = 'admin' if u[:admin]
        m.tr(attrs) do
          m.td(:class => 'family_name bold') do
            m.text u[:family_name]
          end
          m.td u[:given_name]
          m.td u[:gender] == 'male' ? '男' : '女'
        end
      end
    end
  end
end
```

カスタム定義ヘルパーができましたので、それでは、使ってみましょう。

▼ source/users.html.slim

```
---
title: "利用者一覧"

my_favorite_users_list:
```

```

- family_name: 浅倉
  given_name: 南
  gender: female
  admin: true

- family_name: 上杉
  given_name: 達也
  gender: male
  admin: false

- family_name: 上杉
  given_name: 和也
  gender: male
  admin: false

---


# 利用者一覧


= table_of_users(current_page.data.my_favorite_users_list)

```

これをビルドすると、次のような HTML が生成されます。

▼ index.html

```


| 姓  | 名  | 性別 |
|----|----|----|
| 浅倉 | 南  | 女  |
| 上杉 | 達也 | 女  |
| 上杉 | 和也 | 女  |


```

利用者が三人ですので、直接 slim や html を書いても手間ではありませんが、多くの利用者を表示する際にはとても役立ちます。

管理者属性 (admin) を持つ浅倉南さんの行に `class="admin"` とクラス属性が付与されている点にも注目です。手作業で行うとミスが起きやすいですが、カスタムヘルパ内に書かれているので確実です。

また `<table>` タグを使って書きましたが `` & `` タグで書き直すこととなった際にも容易に修正できます。

さらに、データファイル (後述) として作成した `users.yml` から、各利用者のデータを取得、纏めて表示する際などにも威力を発揮します。

利用者一覧

姓 名 性別
浅倉 南 女
上杉 達也 男
上杉 和也 男

2.10 レイアウト

レイアウト機能を使うと、ウェブサイトのそれぞれのページを囲むための共通 HTML が使えるようになります。つまり、ウェブサイトの各ページには、`<head>(略)</head>`や、`<header>(略)</header>`、`<footer>(略)</footer>`など、共通する「外枠」がありますが、この「外枠」を共用する機能がレイアウト機能です。

それでは、早速使ってみましょう。まずは次のようにレイアウトファイルを用意します。

▼ source/layouts/site.slim

```
doctype html
html
  head
    meta charset="utf-8"
    meta name="viewport" content="width=device-width"
    title
      = current_page.data.title || "さくら商会"
      = stylesheet_link_tag "style"

  body
    = yield
```

そして、各ページに固有の内容を、それぞれのファイルに記述します。

▼ source/about.html.slim

```
h1 会社紹介
```

▼ source/contact.html.slim

```
h1 お問い合わせ
```

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

▼ build/about.html

```
<html>
<head>
  <meta charset="utf-8">
  <meta content="width=device-width" name="viewport">
  <title>さくら商会</title>
  <link href="stylesheets/style.css" rel="stylesheet">
</head>

<body>
  <h1>会社紹介</h1>
</body>
</html>
```

▼ build/contact.html

```
<html>
<head>
  <meta charset="utf-8">
  <meta content="width=device-width" name="viewport">
  <title>さくら商会</title>
  <link href="stylesheets/style.css" rel="stylesheet">
</head>

<body>
  <h1>お問い合わせ</h1>
</body>
</html>
```

レイアウトファイル `site.slim` の `= yield` に、`about.html.slim` や、`contact.html.slim` に書いた内容が差し込まれています。

素の `html` で書く際には、`<head>(略)</head>`など、各ページで共通する部分も全て、それぞれのページに記述しなければなりませんでした。共通する部分をレイアウトファイルに纏めることで、各ページには固有の内容のみを記述すれば良くなりますので、見通しも良くなり、開発効率も上がります。

レイアウトファイルは `source/layouts/site.slim` ですが、個別ページは `source/about.html.slim` や `source/contact.html.slim` です。**拡張子が異なる** ことに注意してください。

♣ カスタムレイアウト

既定では `Middleman` は作成するウェブサイトのすべてのページに同じレイアウト (`site.slim`) を適用しますが、例えばトップページとその傘下にある下層ページなど、それぞれのページが用いるレイアウトを指定したい場合があります。

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指定するには、`config.rb` に次のように記述します。

▼ config.rb

```
# レイアウトファイルの指定
set :layout, "site"
page "index.html", layout: "top"
```

♣ 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。

全てのページでレイアウトを適用したくない場合**▼ config.rb**

```
set :layout, false
```

特定のページ (no_layout.html) にレイアウトを適用したくない場合**▼ config.rb**

```
page 'no_layout.html', layout: false
```

2.11 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。

ウェブサイトには、ヘッダー・ナビゲーション・フッターなど複数のページで共通して使われる部分があります。また一つのページでしか用いられなくても、その記述が長くなる場合など、別ファイルに分離すると見通しがよくなります。こうした際に用いるのが「パーシャル」です。

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 ディレクトリに置かれる _footer.slim と名付けられた footer パーシャルを示します。

▼ source/_footer.slim

```
footer
  p
    | Copyright (C) 令和四年 さくら商会 All rights reserved.
```

こうして準備したパーシャルファイルを利用するには、partial メソッドを使います。partial メソッドを使って既定のレイアウトにパーシャルを配置した例を次に示します。

▼ source/layouts/site.slim

```
doctype html
html
```

```

head
  meta charset="utf-8"
  meta name="viewport" content="width=device-width"
  title
    = current_page.data.title || "さくら商会"
    = stylesheet_link_tag "style"

body
  = yield
  = partial "footer"

```

パーシャルファイル名は _footer.slim と _(アンダースコア) が付いていますが、partial × ソッドに使いたいパーシャルファイル名を渡す際には = partial "footer" と _(アンダースコア) は不要です。

それでは次のコードを用意し、ビルドしましょう。

▼ source/about.html.slim

```
h1 会社紹介
```

build ディレクトリ以下に、会社紹介ページ about.html が出力されています。

▼ build/about.html

```

<!DOCTYPE html>
<html>
<head>
  <meta charset=utf-8 />
  <meta content="width=device-width" name=viewport />
  <title>さくら商会</title>
  <link href="stylesheets/style-85d5d9f4.css" rel=stylesheet />
</head>
<body>
  <h1>会社紹介</h1>
  <footer>
    <p>Copyright (C) 令和四年 さくら商会 All rights reserved.</p>
  </footer>
</body>
</html>

```

= partial "footer" の部分が _footer.slim で置き換えられていることが確認できます。

フッターは全てのページで使われますが、内容を更新したくなった場合は、 _footer.slim を編集するだけで全てのページが更新されるので、とても便利です。

2.12 パーシャルに変数を渡す

そして、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。

飼っているペットの猫たちの紹介をするウェブサイトの例を挙げてみます。

▼ source/_cat.slim

```
h2 = name
= image_tag image_filename
p = introduction
```

▼ source/cat.html.slim

```
h1 猫たちの紹介

= partial "cat",
  locals: { name: "タマ",
            image_filename: "tama.jpg",
            introduction: "タマです。白黒の猫です。"}

= partial "cat",
  locals: { name: "ミケ",
            image_filename: "mike.jpg",
            introduction: "ミケです。茶色の猫です。"}
```

ビルドすると、 build ディレクトリ以下に 次のように出力されます。

▼ build/cats.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset=utf-8 />
  <meta content="width=device-width" name=viewport />
  <title>さくら商会</title>
  <link href="stylesheets/style.css" rel=stylesheet />
</head>

<body>
  <h1>猫たちの紹介</h1>

  <h2>タマ</h2>
  
  <p>タマです。白黒の猫です。</p>

  <h2>ミケ</h2>
  
  <p>ミケです。茶色の猫です。</p>

</body>
</html>
```

_cat.slim には `h2 = name` と書かれていましたが、出力された cats.html では `<h2>タマ</h2>` に置き換えられていることが確認できます。

どのタグを使って猫の紹介をするのか、その形式は共通していますので、こういった場合、パーシャルファイル _cat.slim を使うと、同じタグの繰り返しを何度も書かずに済みます。

そして、置き換えたい猫のデータを、`locals: { name: "タマ"}` とハッシュ形式で渡すことで、個別に異なる猫のデータだけを必要な箇所に指し込むことが出来ます。

使用するタグを変更したい場合にも、パーシャルファイルを編集するだけで、ページが更新されますようになります、とても便利ですので使いこなしていきましょう。

猫たちの紹介 タマ



タマです。白黒の猫です。

2.13 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^{*15} から抜き出すと便利な場合があります。

先ほど登場した「猫を紹介するページ」の例では、cats.html.slim 内に、個々の猫に関するデータを直接記述し、`locals: {}` として、パーシャルファイルに渡していました。

猫が十匹二十匹と増えてくると、逐一ハッシュを記述するのも大変です。また、index.html の他に、もしかすると tama.html や mike.html、あるいは タマ.html やミケ.html というページでも、猫のデータを使いたいかもしれません。^{*16}

このような場合、猫のデータのみを別ファイルにし、必要に応じて読み出せると便利です。

Middleman はこういった用途のために「データファイル」機能を備えています。

データファイルは `data` ディレクトリの中に YAML(ヤムル) 形式のデータとして作ることができます。拡張子は `yml` です。作成したならば、それぞれの `slim` ファイル内で、この情報を使うことができます。`data` ディレクトリは、`source` ディレクトリと同じように、プロジェクトのルートに置きます。

それでは、ページのコンテンツデータ（ここでは、猫たちの紹介データ）を、YAML(ヤムル) 形式

^{*15} レンダリングとは、何らかの抽象的なデータ集合を元に、一定の処理や演算を行って文字や画像や映像、音声などを生成すること。例えば、一つのウェブページに含まれる、HTML や CSS などによる描画内容の記述、スクリプトによる動作の記述、画像ファイルやフォントファイルなど外部のデータなどを組み合わせ、ブラウザのウィンドウ内にページ内容の描画を行うことを指す。（<https://e-words.jp/w/レンダリング.html>）

^{*16} 以前は、半角英数に限られていた URL ですが、日本語も使えるようになっています。

のデータファイルに抜き出してみましょう。`data` ディレクトリ内に `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。白黒の猫です。
- name: ミケ
  image_filename: mike.jpg
  introduction: ミケです。茶色の猫です。
```

`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 YAML(ヤムル) 形式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 (JSON(ジェイソン) 形式に良く似ているので、馴染み易いかもしれません。)

▼ `data/cats.yml`

```
[  
  { name: タマ,  
    image_filename: tama.jpg,  
    introduction: タマです。白黒の猫です。},  
  { name: ミケ,  
    image_filename: mike.jpg,  
    introduction: ミケです。茶色の猫です。}  
]
```

次の形式になっていることが読み取れるでしょうか。

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

`[]` 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)」と言います。`name: タマ` のように、鍵(key): 値(value)の形になっています。(項目名: データと読むことができます。)

YAML(ヤムル) 形式には二種類の書き方があります。お好みでお使いください。

用意したデータファイルは、次のように使います。

▼ `source/cats2.html.slim`

`h1 猫たちの紹介`

```
- data.cats.each do |cat|
  h2 = cat.name
  = image_tag cat.image_filename
  p = cat.introduction
```

これを、ビルドすると、 build ディレクトリ以下に 次のように出力されます。

▼ build/cats2.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset=utf-8 />
  <meta content="width=device-width" name=viewport />
  <title>さくら商会</title>
  <link href="stylesheets/style.css" rel=stylesheet />
</head>

<body>
  <h1>猫たちの紹介</h1>

  <h2>タマ</h2>
  
  <p>タマです。白黒の猫です。</p>

  <h2>ミケ</h2>
  
  <p>ミケです。茶色の猫です。</p>

</body>
</html>
```

パーシャルファイルを使った場合と全く同じになります。cats.html.slim では、別に用意したパーシャルファイル _cat.slim へ、`locals: {}` としてハッシュ形式でデータを渡していましたが、cats2.html.slim では、_cat.slim 相当のコードが直接 cats2.html.slim に書かれており、cats.yml ファイルから猫データを読み込みレンダリングしています。

パーシャルファイルに引数として猫データを渡してレンダリングするのか、直接データファイルから猫データを読み取りレンダリングするのか、どちらも同じ結果が得られます。状況に応じて適宜活用してください。

【コラム】Ruby 文法解説

source/cats2.html.slim を解説します。

`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。^{*17}

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。`data.cats` には、「配列」と呼ばれるデータ形式で、全ての猫たちのデータが納められています。

`.each` は、Ruby で、配列の各々の要素（つまり一匹一匹の猫）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、次に続く Ruby コードを実行するよう指示する構文です。

`|cat|` には、猫一匹分の全てのデータ (`name`(名前), `image_filename`(画像ファイル名), `introduction`(紹介文)) が入っていますので、`cat.name` は、取り出した猫一匹の `name`(名前) です。(タマやミケなど、個々の猫の名前になります)

`h2 =` ですが、`h2` は、`<h2>`タグになります。`=` を書くことで、Ruby でいろいろ処理を行った結果を、`h2` タグの内容として出力できるので、`<h2>タマ</h2>` のようになります。

`= image_tag cat.image_filename` は、Middleman で用意されている `image_tag` ヘルパを使った書き方です。`image_tag` ヘルパに引数として `cat.image_filename` を渡すと、``タグを生成してくれます。

以上で、`cats.yml` という、猫たちの紹介データを用いて、`html` を作成できました。

データと枠との分離

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は済むので、とっても楽です。データとそれを納める枠とを分離することで、管理が容易になります。活用していきましょう。

2.14 パーシャルやヘルパなど 様々な手法でのタグ生成

パーシャルやヘルパ、データファイルなど様々なことを扱ってきました。今までのまとめとして、九種類の写真ギャラリーを作成してみます。最終的に出来上がる HTML タグは同じものとなります。作りやすさや分かりやすさなど、ご参考になれば幸いです。

写真展 その1



写真展 その2



写真展 その3



*12 Ruby とは、まつもとゆきひろ さんが考案した、柔軟性に富む優れた書き味で人気のプログラミング言語です。オブジェクト指向スクリプト言語 Ruby(<https://www.ruby-lang.org/ja/>)

▼ HTML で書いた例

```
// html で書いた例
<h3>写真展 その1</h3>
<div class="gallery">
  <a href="images/tora.jpg" title="The Cleaner">
    
  </a>
  <a href="images/tsuru.jpg" title="Winter Dance">
    
  </a>
  <a href="images/zou.jpg" title="The Uninvited Guest">
    
  </a>
</div>
```

最も基本となる形です。

▼ Slim で書いた例

```
// slim で書いた例
h3 写真展 その2
.gallery
  a href="images/tora.jpg" title="The Cleaner"
    img src="images/tora_s.jpg"
  a href="images/tsuru.jpg" title="Winter Dance"
    img src="images/tsuru_s.jpg"
  a href="images/zou.jpg" title="The Uninvited Guest"
    img src="images/zou_s.jpg"
```

Slim 記法で簡潔に記すことが出来ました。HTML と比較するととてもすっきりしています。

▼ helper メソッドを使って書いた例

```
h3 写真展 その3
.gallery
  = link_to "images/tora.jpg", title: "The Cleaner"
  = image_tag "tora_s.jpg"
  = link_to "images/tsuru.jpg", title: "Winter Dance"
  = image_tag "tsuru_s.jpg"
  = link_to "images/zou.jpg", title: "The Uninvited Guest"
  = image_tag "zou_s.jpg"
```

`link_to` や `image_tag` といったヘルパが標準で備わっています。Ruby on Rails でも良く用いる書き方でお薦めです。

▼ データを格納した配列を使って書いた例

```
h3 写真展 その4
.gallery
  - doubutsu = [ { picture: "tora",      title: "The Cleaner" },
                  { picture: "tsuru",     title: "Winter Dance" },
                  { picture: "zou",       title: "The Uninvited Guest" }]
  - doubutsu.each do |kemono|
```

```

- picture = kemono[:picture]
- title   = kemono[:title]
= link_to "images/#{picture}.jpg", title: title
  = image_tag "#{picture}_s.jpg"

```

Ruby は、簡潔で親しみやすく理解しやすいプログラミング言語です。 `doubutsu` 配列にそれぞれの 獣 の写真と文字を格納します。そして配列から一つ一つの獣を取り出し、リンクと画像を生成する基本的な Ruby のコードです。

▼ パーシャルファイルに、データを渡して書いた例

```

h3 写真展 その5
.gallery
  = partial "gallery", locals: { picture: "tora",      title: "The Cleaner" }
  = partial "gallery", locals: { picture: "tsuru",     title: "Winter Dance" }
  = partial "gallery", locals: { picture: "zou",       title: "The Uninvited Guest" }
>}

```

▼ _gallery.slim

```

= link_to "images/#{picture}.jpg", title: title
  = image_tag "#{picture}_s.jpg"

```

呼び出し元では、`= partial "gallery"` と、パーシャルファイルを表示するよう記述しています。そして、パーシャルファイルに渡す個々の獣情報を `locals: { picture: "tora", title: "The Cleaner" }` と、ハッシュ形式で表現しています。

▼ 4, 5 の併せ技

```

h3 写真展 その6
.gallery
  - doubutsu = [ { picture: "tora",      title: "The Cleaner" },
                 { picture: "tsuru",     title: "Winter Dance" },
                 { picture: "zou",       title: "The Uninvited Guest" }]

  - doubutsu.each do |kemono|
    = partial "gallery", locals: kemono

```

データ配列から `each` メソッドにより個々の獣データを取得、パーシャルファイルに渡す例です。

▼ データファイルを使った例

```

h3 写真展 その7
.gallery
  - data.doubutsu.each do |kemono|
    = partial "gallery", locals: kemono

```

▼ data/doubutsu.yml

```

- picture: tora
  title: The Cleaner

```

```
- picture: tsuru
  title: Winter Dance
- picture: zou
  title: The Uninvited Guest
```

データファイルを使った例です。コードとデータが分離され、すっきりしています。データファイルから取得したデータを、パーシャルに渡し、HTML を生成しています。

▼ データファイルとカスタムヘルパを使った例

```
h3 写真展 その8
.gallery
  - data.doubutsu.each do |kemono|
    = animal_photo kemono
```

▼ helpers/doubutsu.rb

```
def animal_photo(data)
  link_to "images/#{data.picture}.jpg", title: data.title do
    image_tag "#{data.picture}_s.jpg"
  end
end
```

データファイルから読み取ったデータを、パーシャルではなくカスタムヘルパに渡し HTML を生成します。カスタムヘルパ内では Middleman 標準の `link_to`, `image_tag` を使ってています。

▼ 全てをカスタムヘルパで生成する例

```
h3 写真展 その9
= animals_photo
```

▼ helpers/doubutsu.rb

```
def animals_photo
  markup do |m|
    m.div(class: 'gallery') do
      data.doubutsu.each do |kemono|
        m.a(href: "images/#{kemono[:picture]}.jpg", title: kemono[:title]) do
          m << image_tag("#{kemono[:picture]}_s.jpg")
        end
      end
    end
  end
end
```

データファイルからの読み込みも含めて、全てをカスタムヘルパに 委ねた例です。 Slim 自体は僅か二行ととてもすっきりしました。カスタムヘルパ内では、`HtmlBuilder` を使って書いています。少し記法が独特ですが `content_tag` では書きにくい際に活躍します。

以上九種類をご紹介いたしました。適宜、ご活用下さい。

2.15 動的ページ

Middleman にはテンプレートファイルと一対一の対応関係を持たないページを生成する機能があります。この機能により、変数に応じて複数のファイルを作り出すテンプレートを使えるようになります。該当するページはないにもかかわらず、代わって応答してくれるところから Proxy(プロキシ・代理人) と呼ばれます。

具体例を挙げます。先ほどは、猫たちの紹介を、`cats.html` の中で行いましたが、それぞれの猫の専用ページ、`タマ.html` や `ミケ.html` というページを作ることができます。Proxy(プロキシ・代理人) となってくれる、猫専用ページの雛形ページを作成し、猫の名前を与えると、代理人がそれぞれのページの代わりに応答してくれる感覚になります。

この Proxy(プロキシ・代理人) を作るには、`config.rb` で `proxy` メソッドを使って、「作りたいページのパス」「使いたいテンプレートのパス」を与えます。

それでは、猫たち専用ページを作成するために、`config.rb` を設定してみましょう。

▼ config.rb

```
data.cats.each do |neko|
  proxy "/#{neko.name}.html", "/neko_template.html", locals: { neko_data: neko },
> ignore: true
end
```

一行目の `data.cats.each do |neko|` は、`cats.yml` からデータを取得します。そして取得した一匹一匹の猫データは `neko` という変数に格納されます。

二行目ですが `neko` には一匹一匹の猫データが格納されていますので、`neko.name` で猫の名前を取り出することができます。ですので `"/#{neko.name}.html"` は `タマ.html` や `ミケ.html` となります。つまり `proxy "タマ.html", "/neko_template.html"` `proxy "ミケ.html", "/neko_template.html"` と書いたのと同じことになります。

`proxy` は「代理人」ですから、ブラウザの URL 欄に `https://example.com/タマ.html` と入力された際に `タマ.html` の代理人として、`neko_template.html` が代わりに応答するよう設定しています。

そこで、代わりに応答する `neko_template.html` に一匹一匹の猫のデータを渡してあげる必要があります。データファイル `cats.yml` から取り出した一匹一匹の猫のデータは `neko` に格納されていますので、`locals: { neko_data: neko }` と書くことで、この `neko` を `neko_template.html` に渡せます。すると `neko_template.html` 内で一匹一匹の猫のデータを `neko_data` という変数名で参照できるようになります。

それでは「代理人」となる `neko_template.html.slim` の中身を見てみましょう。

▼ source/neko_template.html.slim

```
h2 = neko_data.name
= image_tag neko_data.image_filename
p = neko_data.introduction
```

以前に登場してきたものと同じ内容です。 neko_data には、一匹一匹の猫のデータが入っていますので neko_data.name と記述し、猫の名前など、必要な情報を取り出しています。

レイアウトファイルを次のように準備して、ビルドしてみましょう。

▼ source/layouts/site.slim

```
doctype html
html
  head
    meta charset="utf-8"
    meta name="viewport" content="width=device-width"
    title
      = current_page.data.title || "さくら商会"
      = stylesheet_link_tag "style"

  body
    = yield
```

出力された二つのファイル タマ.html や ミケ.html は、次のようになります。

▼ タマ.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset=utf-8 />
  <meta content="width=device-width" name=viewport />
  <title>さくら商会</title>
  <link href="stylesheets/style.css" rel=stylesheet />
</head>
<body>
  <h2>タマ</h2>
  <p>タマです。白黒の猫です。</p>
</body>
</html>
```

▼ ミケ.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset=utf-8 />
  <meta content="width=device-width" name=viewport />
  <title>さくら商会</title>
  <link href="stylesheets/style.css" rel=stylesheet />
</head>
<body>
  <h2>ミケ</h2>
```

```
<p>ミケです。茶色の猫です。</p>
</body>
</html>
```

猫の雛形ファイル `neko_template.html.slim` と、サイトのレイアウトファイル `site.slim` を元に、(猫の情報のみが異なるのみで) 二つのほとんど同じ HTML ファイルが生成されました。

人の手で同じ作業をしようとすると、労力もかかり、また間違いも起きがちです。何よりもおもしろいのが、より創造的なことに生命力を使えるよう Middleman より機能提供されていますので、活用していきましょう。

ミケ



ミケです。茶色の猫です。

変数名について

`neko` という変数名やファイル名に違和感を感じた方もいらっしゃるかもしれません。一般的には `neko` に代えて `cat` が良く用いられます。しかし、`cat` という同じ語が大量に現れることとなりますので、ここでは混乱を避け、区別する意図から `neko` としました。

エディタ上でそれぞれの語の意味に応じて色分けされるように、ときにはローマ字の変数名を使ってみるのも区別しやすくなり良いかもしれません。

2.16 ファイルサイズ最適化

ウェブサイトを公開するときには、応答速度の改善のために、HTML、CSS、JavaScript や、画像ファイルの圧縮を行うことが推奨されています。

Middleman にも、もちろんファイル最適化の為の機能が用意されており、`Gemfile` や `config.rb` に簡単な設定を書くことで使えるようになります。(雛形を利用した場合には、既に記述済みです。)

♣ HTML の圧縮

Middleman で、HTML 出力を圧縮するためには、公式に用意されている拡張機能 `middleman-minify-html` を用います。`Gemfile` に次のように記述します。

▼ Gemfile

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

そして、ターミナルから、`bundle install` を実行します。

```
% bundle install
```

これで、HTML 出力圧縮のための `gem` ファイル、`middleman-minify-html` のインストールが完了です。

そして、`config.rb` を開いて、次のように追加します。

▼ config.rb

```
configure :build do
  activate :minify_html
end
```

これで、設定は完了です。ビルドして出力された HTML ファイルを確認してみてください。スペースやタブや改行などが削除され、圧縮されていることが分かります。

♣ CSS の 圧縮

CSS の圧縮はさらに簡単です。次のように `config.rb` に記述するだけで完了します。

▼ config.rb

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

♣ JavaScript の 圧縮

JavaScript の圧縮は少し複雑ですが、次のように `config.rb` に記述すると完了します。

▼ config.rb

```
require "uglifier"

configure :build do
  activate :minify_javascript,
  compressor: proc {
    ::Uglifier.new(
      mangle: { toplevel: true },
      compress: { unsafe: true },
      harmony: true
    )
  }
end
```

♣ 画像ファイルの圧縮

次世代画像形式の WebP、そして AVIF へ^{*18} という記事より、引用してご紹介いたします。

長い間、Web の静止画に関しては「写真の JPEG、ロゴやイラストの GIF、透過画像の PNG」という明確な使い分けが確立されてきました。WebP はこのすべてを置き換えることができる次世代のフォーマットです。

ウェッピー 次世代画像形式のWebP

長い間、Web の静止画に関しては「写真の JPEG、ロゴやイラストの GIF、透過画像の PNG」という明確な使い分けが確立されてきました。WebP はこのすべてを置き換えることができる次世代のフォーマットです。

♣ WebP は JPEG/GIF/PNG(APNG) をカバーする魅力的なフォーマット

WebP を使うことで、これまで用途や画像の特徴ごとに使い分けが必要だったフォーマットの一本化が可能になります。主な特徴を簡単に紹介しましょう。

- 高い圧縮率：同等画質の JPEG と比較して 20~30% のサイズ削減（JPEG の置き換え）
- 不可逆圧縮と透過アニメーションの併用（透過アニメーションでも画質を犠牲にしてサイズを削減できる）（GIF/PNG の置き換え）
- 画質劣化のない可逆圧縮もサポート（GIF/PNG の置き換え）



♣ さらに次世代のフォーマット、AVIF も

- 多様な色空間やサンプリング方式をサポート
- WebP よりもさらに高画質でコンパクト（同じサイズでも画質が高く、JPEG に特有のブロック

*18 <https://ics.media/entry/201001/>

クノイズも発生しない)

- Amazon・Netflix・Google・Microsoft・Mozilla 等の幅広い企業によるコンソーシアムが共同で開発 (Facebook や Apple も後から参画)



今まで登場してきた猫たちの画像ファイルでも実験してみました。猫のイラストは [イラスト AC](#)*19 より取得、横 900px, 縦 1200px に切り抜いたものを使いました。

jpg と webp のサイズ比較

jpg	サイズ	webp	サイズ	減少サイズ	減少率
tama.jpg	79KB	tama.webp	36KB	43KB	54%
mike.jpg	88KB	mike.webp	41KB	47KB	53%

jpg 形式から webp 形式に変換することで、半分以下にまで小さくなりました。

今日では 全てのブラウザが webp 形式に対応しています。ImageMagick *20 等のツールを導入することで、簡単に画像形式を変換することができます。ネット上でオンラインで変換してくれるサイトもございます。

画像の表示も速くなり、利用者も快適に閲覧できますので、使っていきましょう。

♣ デバッグ時のオプション

通常 Slim で生成される html ファイルは、ウェブサイトに公開した際の速度改善が望めるよう、空白等を取り除きコンパクトに圧縮されています。

*19 <https://www.ac-illust.com>

*20 ImageMagick の使いかた 日本語マニュアル (<https://imagemagick.biz/>)

デバック時には、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。

▼ config.rb

```
set :slim, {
  pretty: true, sort_attrs: false
}
```

併せて、圧縮にかかる設定も、コメントアウトして無効化しておきましょう。

▼ config.rb

```
configure :build do
  # HTML圧縮
  # activate :minify_html
  # CSS圧縮
  # activate :minify_css
  # JavaScript圧縮
  # activate :minify_javascript,
  #   compressor: proc {
  #     ::Uglifier.new(
  #       mangle: { toplevel: true },
  #       compress: { unsafe: true },
  #       harmony: true
  #     )
  #   }
  # 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
  # テキストファイルのgzip圧縮
  # activate :gzip
end
```

2.17 画像形式の変換

画像形式を変換するにはどうしたら良いのでしょうか？ 様々な手法がありますが、ここでは、簡単なコマンド操作で処理できる [ImageMagick *21](#) をご紹介いたします。

[Imagemagick の使い方日本語マニュアル*22](#) というサイトがあります。「日本語のマニュアルが少ない画像加工ツール ImageMagick の使い方を、初心者の方にも解りやすいよう詳しく丁寧に解説している」サイトです。

インストールから、画像の拡大縮小、形式変換、色の加工など、様々な使い方が紹介されています。適宜引用抜粋しつつ、その使い方をご紹介いたします。

*21 `magic` ではなく `magick` と綴られています。一般名詞を組み合わせて、固有名詞化する際に、このような事例が時折見られます。

*22 <https://imagemagick.biz>

♣ インストール

既に Mac 用パッケージマネージャ HomeBrew の導入が完了しておりますので、ターミナルから以下のコマンドを入力することで、ImageMagick のインストールは完了します。

```
$ brew install imagemagick
```

♣ 画像形式の変換

フォーマット変換とも呼ばれます。 `tama.jpg` という jpg 形式の画像を、 `tama.webp` という webp 形式の画像に変換するには、次のようにします。

まず、画像があるディレクトリに移動します。そのためには、現在、どのディレクトリにいるのか、知る必要があります。そのためのコマンドが `pwd` です。

```
% pwd
/Users/haruka/my_new_site
```

現在、 `/Users/haruka/my_new_site` というディレクトリにいることが分かりました。

`haruka` は、このコンピュータを利用している、ユーザ名です。 Finder アプリでは、サイドバーに、家のアイコンと共に表示されています。

`my_new_site` は、Middleman で新規作成したディレクトリ（ウェブサイト）の名前です。画像ファイルは、`source/images/` ディレクトリにありますから、そこまで移動します。

移動するためのコマンドは、 `cd` です。

```
% cd source
% cd images
```

と入力して、ディレクトリを移動しましょう。

`pwd` コマンドで、現在いるディレクトリを確認してみます。

```
% pwd
/Users/haruka/my_new_site/source/images
```

となり、無事に移動することが出来ました。

ちなみに上のディレクトリに戻るには

```
% cd ..
```

と入力することで、一つ上のディレクトリに戻れます。

そのディレクトリにどのようなファイルがあるか、表示するには、 `ls` コマンドを用います。

```
% ls  
apple-touch-icon-180x180.png      tora.jpg  
favicon.ico                      (略)
```

とたくさん表示されました。

jpg ファイルのみに絞って表示させるには、以下のコマンドを用います。

```
% ls *.jpg  
logo.jpg                  tama.jpg          tsuru_s.jpg  
mike.jpg                 tora.jpg          (略)
```

tama.jpg があることが確認できました。

それでは、見つかった tama.jpg を webp 形式に変換しましょう。

一つのファイルの画像形式を変換する

```
$ convert tama.jpg tama.webp
```

これで、変換完了です。

複数のファイルの画像形式を変換する

一つ一つファイル名を指定してコマンド入力して変換するのは、大変です。纏めて変換することも出来ます。そのためには以下のコマンドを入力します。

```
$ for i in *.jpg; do convert $i ${i%jpg}webp; done
```

これで、全ての jpg ファイルを webp 形式に変換できました。

簡単なシェルコマンドについてご紹介いたしました。より詳しく学びたい方には、巻末の参考文献「1日1問、半年以内に習得 シェル・ワンライナー 160本ノック」がお薦めです。シェルコマンドを極めるための様々な技法が紹介されていますので、是非ご一読下さい。

♣ 画像の拡大縮小

4k など 巨大な画像を縮小するには、以下のコマンドで行えます。

アスペクト比を維持

```
$ convert tama_large.jpg -resize 640x480 tama_small.jpg
```

アスペクト比が維持されるよう(画像の横幅が 640px になるか、あるいは画像の縦が 480px になるまで)、画像が縮小されます。

アスペクト比を無視

```
$ convert tama_large.jpg -resize 640x480! tama_small.jpg
```

サイズ指定の後に !(感嘆符) を付けると、アスペクト比を無視し、強制的に画像サイズを、横 640px、縦 480px に縮小します。

アスペクト比を維持（幅のみを指定）

```
$ convert tama_large.jpg -resize 640x tama_small.jpg
```

画像の横幅のみを指定すると、画像の縦はアスペクト比が維持されるように適宜調整して縮小されます。

アスペクト比を維持（縦のみを指定）

```
$ convert tama_large.jpg -resize x480 tama_small.jpg
```

画像の縦のみを指定すると、画像の横はアスペクト比が維持されるように適宜調整して縮小されます。

パーセンテージで縮小

```
$ convert tama_large.jpg -resize 25% tama_small.jpg
```

パーセンテージを指定すると、アスペクト比が維持して縮小されます。

2.18 画像の遅延読み込み

一般に画像の読み込みは時間がかかるものです。写真を中心のサイトを作っている際などはなおさらです。

このために様々な仕様が規定されました。`<img srcset="...*23` や、`<picture><source srcset="...*24` などです。

ここでは、簡単に行える方法として、`` をご紹介いたします。

`loading` ブラウザーがどのように画像を読み込むかを示します。

`eager` 画像が現在可視ビューポートに入っているかどうかにかかわらず、直ちに画像を読み込

^{*23} ``: 画像埋め込み要素 (<https://developer.mozilla.org/ja/docs/Web/HTML/Element/img>)

^{*24} `<picture>`: 画像要素 (<https://developer.mozilla.org/ja/docs/Web/HTML/Element/picture>)

みます（これが既定値です）。

`lazy` 画像がブラウザーで定義されたビューポートからの距離に達するまで、画像の読み込みを遅延させます。これは、画像が必要とされるのが合理的に確実になるまで、処理に必要なネットワークやストレージの帯域幅を使用しないようにするためです。これは一般的に、ほとんどの典型的な使用法において、コンテンツの性能を向上させることができます。

全ての `` タグに `loading="lazy"` が付くよう、ヘルパメソッドを用意しました。

▼ helpers/lazy.rb

```
module Padrino
  module Helpers
    module AssetTagHelpers
      def image_tag(url, options={})
        options = { :src => image_path(url) }.update(options)
        options[:alt] ||= image_alt(url) unless url.to_s =~ /\A(?:cid|data):|\A\Z/
        # img タグに loading="lazy" を追加する
        options[:loading] ||= :lazy

        # 画像名から自動で img タグの alt 属性, width 属性, height 属性を付与する
        # (Middleman::Extensions::AutomaticAltTags より)
        unless url.include?('://')
          options[:alt]    ||= ''
          options[:width]  ||= ''
          options[:height] ||| ''
        end

        real_path = url.dup
        unless real_path.start_with?('/')
          real_path = File.join(config[:images_dir], real_path)
        end

        file = app.files.find(:source, real_path)

        if file && file[:full_path].exist?
          # alt 属性
          if options[:alt].empty?
            options[:alt] = File.basename(file[:full_path].to_s, '.*').capitalize!
          end

          # width 属性, height 属性
          if options[:width].empty? || options[:height].empty?
            w, h = FastImage.size(file[:full_path])
            options[:width]  = w
            options[:height] = h
          end
        end
      end

      tag(:img, options)
    end
  end
end
```

```
    end  
  end  
end
```

▼ ソースコード slim

```
= image_tag "logo.webp"
```

▼ ビルド結果 html

```

```

ブラウザでのレイアウトシフトを回避するために、`` タグは `` と、`width` 属性、`height` 属性 の記述が求められるようになりましたので、こちらも自動で付与するようにしています。

以上、静的サイトジェネレータ `Middleman` の使い方を紹介してきました。

少し規模が大きくなると、素の HTML や CSS を使ってウェブサイトを構築していくのは大変になります。先人の方々が効果的に開発できるよう、素晴らしい資源を提供してくださっていますので、活用ていきましょう。

そしてその恵みに感謝しつつ、後世に生きる方々のために貢献できるよう、何かを残せたら仕合せです。

第 3 章

進化した HTML 記述言語 Slim

Slim は、HTML を簡単に書けるように進化した言語です。公式サイト^{*1} をもとに簡単にご紹介いたします。

【この章の内容】

3.1	Slim の 特徴	58
3.2	イントロダクション	59
3.3	ラインインジケータ	60
3.4	HTML タグ	62
3.5	テキストの展開	66
3.6	埋め込みエンジン	66
3.7	Slim の設定	67

^{*1} <https://github.com/slim-template/slim/blob/master/README.jp.md>

3.1 Slim の特徴

Slim^{*2} 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。

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。



- 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="..."> に展開されます。
- 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
- 高性能
 - ERB に匹敵するスピード

^{*2} <https://github.com/slim-template/slim>

3.2 イントロダクション

♣ 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、「この動作を行うために最低限必要なものは何か。」との考えによって導かれています。

♣ 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、簡単に始めるすることができます。
- 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。

♣ どうやって使い始めるの？

Slim を gem としてインストールすることによって、使い始めることができます。(Middleman を使う場合は、Middleman 内で、Gemfile に書いていますので、以下の操作は不要です。)

```
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。

♣ 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:

▼ Slim の例

```
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}

```

♣ 字下げ(インデント)について

字下げ(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字下げ(インデント)が必要なだけです。

3.3 ラインインジケータ

♣ テキスト

|(パイプ)を使うと、Slimはパイプよりも深く字下げされた全ての行をコピーします。

▼ slim

```

body
  p
    |
      一行目
      二行目
      三行目

```

▼ html

```
<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。

▼ slim

```

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

```

▼ html

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

♣ インライン HTML

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。

▼ slim

```
<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>
```

♣ 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。ブロックは字下げ (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。

▼ slim

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

♣ 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。

▼ slim

```
= 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、单一の等号 (=) 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、單一の等号 (=) と同じです。

♣ 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。

♣ コードコメント / と HTML 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 /! を使用します。

▼ slim

```
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。
```

▼ html

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

3.4 HTML タグ

♣ <!DOCTYPE> 宣言

`doctype` キーワードを使うことで、DOCTYPE を生成できます。

▼ slim

```
doctype html
```

▼ html

```
<!DOCTYPE html>
```

♣ 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。

▼ slim

```
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。

▼ slim

```
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。

▼ slim

```
a<> href="url" リンク
```

♣ 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。

▼ slim

```
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、属性を囲むこともできます。

▼ slim

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

♣ 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。

▼ slim

```
body
  h1 id="headline" ようこそ
```

▼ slim

```
body
  h1 id="headline"
    | ようこそ
```

♣ 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。

▼ slim

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

▼ slim

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

♣ 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オート " か シングルクオート ' を使わなければなりません (引用符で囲まれた属性)。

▼ slim

```
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。

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、三種類の括弧 ({...}、(...))、[...]) で属性を囲むことができます。

▼ slim

```
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。

▼ slim

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

引用符で囲まれた属性

例は、次のようになります。

▼ slim

```
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。

▼ slim

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

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。

▼ slim

```
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
```

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。

▼ slim

```
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
```

♣ ショートカット**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。

▼ slim

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

これは次に同じです

▼ slim

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

3.5 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。

▼ slim

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

3.6 埋め込みエンジン

slim 中に、css なども記述できます。

例を以下に挙げます。

▼ slim

```
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!
```

レンダリング結果は、次のようにになります。

▼ html

```
<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>
```

Ruby, JavaScript, CSS, SCSS, Markdown 形式を使用することができます。

▼表 3.1

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

3.7 Slim の設定

♣ 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^{*3}

♣ テンプレート変換

有志の方により、gem ファイルが公開されていますので、それを用いると、html と slim を相互変換できます。

html ファイルから slim ファイルへ変換

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

ブラウザから変換できるサイト <https://html2slim.herokuapp.com/> もございます。

slim ファイルから html ファイルへ変換

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

^{*3} <https://github.com/slim-template/language-slim>

第 4 章

構文的に優れたスタイルシート Sass

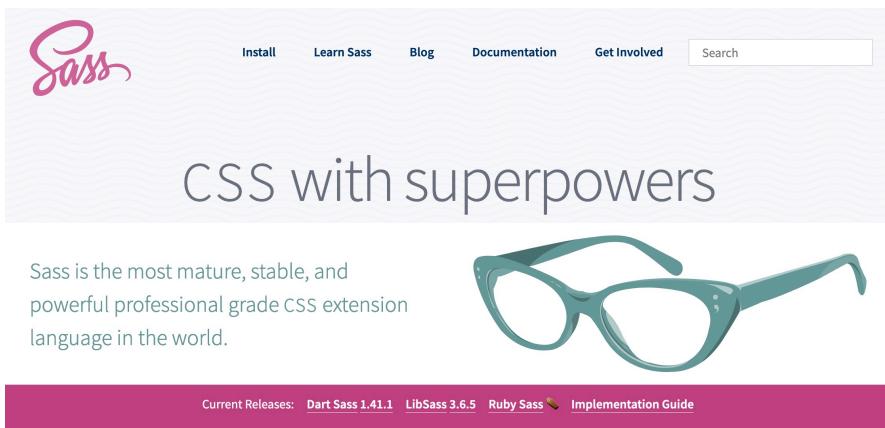
ウェブサイトを作成する際、少し規模が大きくなると、通常の HTML での記述が困難となるのと同様、CSS もより書きやすくなるよう、その能力の向上が求められます。こういった用途に応えて開発されたのが、**Sass** です。**Sass** は、Syntactically Awesome Style Sheets の略で、「構文的に優れたスタイルシート」の意味です。

【この章の内容】

4.1	Sass の特徴	70
4.2	Sass の形式	71
4.3	一行コメントを使う	71
4.4	変数	72
4.5	入れ子 その1	72
4.6	入れ子 その2	74
4.7	スタイルファイルの分割管理	75
4.8	ミックスイン	75
4.9	数値の操作	80
4.10	オリジナル関数の定義	81

4.1 Sass の特徴

Sass の [公式サイト^{*1}](#) と、その使い方を詳細に解説されているサイト [Web Design Leaves^{*2}](#) より適宜引用しつつ使用法などをご紹介します。



まず公式サイトには、次のように案内されています。

スーパーパワーを持った CSS。Sass は、世界で最も成熟した、安定した、強力なプロフェッショナルグレードの CSS 拡張言語です。

Sass は、すべてのバージョンの CSS と完全に互換性があります。私たちはこの互換性を重視しており、利用可能であらゆる CSS ライブラリをシームレスに使用することができます。

Sass は、他のどの CSS 拡張言語よりも多くの機能と能力を誇っています。Sass のコアチームは、他の言語に追いつくだけでなく、先を行くために絶え間ない努力を続けています。Sass は約 15 年間、コアチームによって積極的にサポートされてきました。業界では、Sass が最高の CSS 拡張言語として何度も選ばれています。

Sass は、いくつかのハイテク企業と何百人もの開発者からなるコンソーシアムによって積極的にサポートされ、開発されています。

それでは、特徴を見ていきましょう。

- 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である。

^{*1} <https://sass-lang.com/>

^{*2} https://www.webdesignleaves.com/pr/css/css_basic_08.html

- 演算ができる。
 - 幅や高さなどの四則演算ができる。
 - 色の演算もできる。
- オリジナル関数の定義もできる。

4.2 Sass の形式

Sass には、Sass 形式と SCSS 形式があります。

▼ Sass 形式

```
body
  font-family: Helvetica, sans-serif
  color:      #333
```

▼ SCSS 形式

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

どちらも同じルールセットの宣言になります。どちらの記法を採用するかは慣れや好みです。

Sass 形式では、{}や、;を取り除き、よりすっきりとさせており、簡潔さを追求した記法となっています。

SCSS 形式は従来の CSS と同じ記法です。CSS 形式で書かれたスタイルシートは、そのまま SCSS 形式としても解釈することができ、また学習コストも低いため、SCSS 形式が多く使われています。

4.3 一行コメントを使う

CSS では、コメントとして、範囲コメント /* */ のみが、使用可能でした。

Sass では、一行コメント // が使えます。Atom など、主なエディタでは、複数行選択し、Command + / で、全てコメントにすることができます。この一行コメントは、Sass から CSS にコンパイルされると、コメントは残りませんので、開発中の注釈・覚書などとして、活用することができます。

コンパイルされた CSS にコメントを残したいのであれば、従来通りの範囲コメント /* */ を用いることができます。

4.4 変数

Sass では、「変数」を用いることができます。^{*3}

`$font-stack` や、`$primary-color` が、変数です。早速、使ってみましょう。

▼ SCSS

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

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

これをコンパイルすると、以下の CSS になります。

▼ CSS

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

`$font-stack` が、`Helvetica, sans-serif` に、`$primary-color` が、`#333` に、置き換わっています。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良く、またその意図するところも明確にすることができるため、開発が容易となります。

4.5 入れ子 その1

HTML を記述していくと、タグは抱合関係になっていきます。例えば次のようにです。

▼ HTML

```
<nav>
  <ul>
    <li>
      <a href="#">index.html</a>
    </li>
    <li>
      <a href="#">about.html</a>
    </li>
    <li>
      <a href="#">contact.html</a>
    </li>
  </ul>
</nav>
```

^{*3} CSS でもカスタムプロパティを使って、実現できます。

このような HTML に対して、スタイルを適用するとき、Sass を使うと、とても簡単に記述できます。

▼ SCSS

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

`nav` の中に `ul` があり、`ul` の中に `li` がありと、それぞれの抱合関係が明確であり、HTML の構造そのままに記述していくので、とても楽にスタイルシートを書くことができます。

これをコンパイルすると、次の CSS になります。

▼ CSS

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

nav ul li {
  display: inline-block;
}

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

このように入れ子（ネスト）になっている場合、CSS では、スタイルの適用先を限定するために、`nav ul li a` と書かなくてはなりませんでしたが、不要になり、とても円滑に記述できます。早速、活用していきましょう。

4.6 入れ子 その2

CSS の「擬似クラス」も楽に記述できます。MDN Web Docs^{*4} には、次のように書かれています。

CSS の擬似クラスは、セレクタに付加するキーワードであり、選択された要素に対して特定の状態を指定します。例えば :hover 擬似クラスで、利用者のポインタが当たったときにボタンの色を変更することができます。:active 疑似クラスは、ユーザーによってアクティブ化されている要素を表します。マウスを使用する場合は、「アクティブ化」とは普通、左ボタンを押し下げたときに始まります。

▼ CSS

```
/* 通常のリンクの色 */
a {
  color: pink;
}

/* 利用者のポインタが当たっているすべてのリンク */
a:hover {
  color: blue;
}

/* アクティブ化されている <a> をすべて選択します */
a:active {
  color: red;
}
```

このような場合も Sass では簡潔に記述できます。

▼ SCSS

```
a {
  color: pink;

  &:hover {
    color: red;
  }

  &.active {
    color: blue;
  }
}
```

& で繋ぐことにより、纏めて書けることに着目してください。

^{*4} <https://developer.mozilla.org/ja/docs/Web/CSS/Pseudo-classes>

4.7 スタイルファイルの分割管理

少し大きなウェブサイトを作成していくと、スタイルシートは 1000 行、2000 行と、すぐには理解できなくなるくらいの行数へと増えています。

一つの大きなスタイルシートを管理するのはとても大変です。このため、機能ごとに個々のスタイルシートに分け、管理することで、全体の見通しも良くなります。^{*5}

スタイルシートの分割管理には、いくつかの指針がありますが、ここでは分かりやすく簡潔な FLOW(フロウ) に従って分割した例を示します。^{*6}

▼ style.css.scss

```
// リセットCSS など基盤となる部分のスタイルシート
@import "foundation";

// 要素の配置に関するスタイルシート
@import "layout";

// それぞれの部品に関するスタイルシート
@import "object";

// 色や間隔の設定などあると便利なスタイルシート
@import "utility";
```

foundation.scss, layout.scss, object.scss, utility.scss の四つのファイルを取り込んで、一つの style.css.scss が出来上がります。

取り込まれる部品であることを分かりやすくするために、ファイル名の先頭に _(アンダーバー) を付けて、_foundation.scss, _layout.scss, _object.scss, _utility.scss とすることもあります。

一つの巨大なスタイルシートを管理するのは、とても大変ですので、機能ごと、部品ごとにファイルの分割し、管理するのがお薦めです。

4.8 ミックスイン

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。

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

^{*5} CSS では @import や @layer を使えます。

^{*6} 従来の@import に代えて@use や forward を用いるよう推奨されています。Middleman 非対応のため @import を使ってています。

を置き、任意の名前で定義します。

具体的に、使い方を見ていきましょう。

▼ SCSS

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

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

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

これをコンパイルすると、次のようにになります。

▼ CSS

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

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

card クラスや、photo クラスといった複数のクラスに、同じスタイルを適用したい場合に、同じ CSS を繰り返し書かずに済みます。もし、一つのクラスにしか適用しないスタイルであっても、ミックスインを使うことで、意味や修正が容易となりますので、活用していきましょう。

♣ 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。

それでは、具体例を見ていきましょう。

▼ SCSS

```
// 枠線の色、幅、丸め具合を指定するミックスイン
@mixin my-border($color, $width, $radius) {
  border: {
    color: $color;
    width: $width;
    radius: $radius;
  }
}

// 先に定義した 枠線ミックスインを取り込む
p {
  @include my-border(blue, 1px, 3px);
}
```

▼ CSS

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

枠線の指定など、良く使いますので、このようなミックスインを定義しておき、適宜取り込むようにすると、開発効率がとても上がりますので、お薦めです。

♣ 引数に初期値を設定する

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、変数と同じ書式で記述します。つまり、名前は \$ から始め、 : で区切って値を指定します。

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。

具体例を見ていきましょう。

▼ SCSS

```
// 角丸用のミックスイン
// 初期値は 5px
@mixin kadomaru($radius: 5px) {
  border-radius: $radius;
}

.foo {
  // 角丸ミックスインを取り込む
  // 引数を与えていないので 初期値の 5px が使われる
  @include kadomaru;
```

```

}

.bar {
    // 角丸ミックスインを取り込む
    // 引数を与えていないので 初期値の 5px が使われる
    @include kadomaru();
}

.baz {
    // 角丸ミックスインを取り込む
    // 引数を与えたので 10px が使われる
    @include kadomaru(10px);
}

```

これをコンパイルすると、次の CSS が得られます。

▼ CSS

```

.foo {
    border-radius: 5px;
}

.bar {
    border-radius: 5px;
}

.baz {
    border-radius: 10px;
}

```

♣ メディアクエリへの活用

Sass の変数と mixin で変更に強いメディアクエリをつくる^{*7} という記事があります。

Sass を使うとメディアクエリが書きやすくなりますので、引用してご紹介いたします。

レスポンシブウェブデザインではメディアクエリを書くことが多くなります。通常の CSS ではブレー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 を使えば、変数やミックスインを使うことで一ヶ所で管理することが容易になります。

それでは、利用例を挙げて行きましょう。

▼ SCSS

```

// ブレークポイントの定義
$breakpoints: (
    "tablet": "(min-width: 481px)",
    "desktop": "(min-width: 961px)",
) !default;

```

^{*7} https://www.tam-tam.co.jp/tipsnote/html_css/post10708.html

```
// メディアクエリ用のミックスイン mq の定義
@mixin mq($bp: tablet) {
  @media #{map-get($breakpoints, $bp)} {
    @content;
  }
}
```

@mixin を呼び出すときは次のように使います。

▼ SCSS

```
.card {
  color: blue;

  // 引数を省略
  // 初期値の タブレット用のメディアクエリ
  // (min-width: 481px) が展開される
  @include mq() {
    color: yellow;
  }

  // 引数を指定
  // 指定した デスクトップ用のメディアクエリ
  // (min-width: 961px) が展開される
  @include mq(desktop) {
    color: red;
  }
}
```

これをコンパイルすると、次の CSS が得られます。

▼ CSS

```
.card {
  color: blue;

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

  @media (min-width: 961px) {
    .card {
      color: red;
    }
  }
}
```

\$breakpoints は、Sass の **マップ型変数** で書かれていますので、少し見慣れない感じもしますが、ブレークポイントの定義をしている変数です。ブレークポイントの数値を変更したい場合や、大画面用のブレークポイントを追加したい場合など、\$breakpoints を書き換えるのみで、対応できます。

ウェブサイトのレスポンシブ対応をするに当たってはメディアクエリは不可欠です。Sass を使うと、簡単にメディアクエリを書くことができますので、活用していきましょう。 *8

4.9 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。

早速、使用例を見ていきましょう。

▼ SCSS

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

▼ CSS

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

幅など、それぞれのプロパティが、計算されているのが分かることと思います。手計算して、`width: 188px;` と算出しても良いですが、計算間違いも起りますし、`width: 200px - (5 * 2) - 2;` と、その導出過程を明示することで、意図も明らかになります。 *9

計算で使える記号には、以下のようなものがあります。

演算子の種類

演算	記号
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)

*8 Media Queries Level 4 にて、範囲指定構文が提唱されています。`@media (width <= 480px){}` と分かりやすく書くことができます。Safari での対応が待たれるところです。

*9 CSS にも プロパティ値を計算するための `calc()` 関数がありますので、これを使って計算することができます。Sass ではより強力かつ柔軟に演算が行えるようになっています。

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。

▼ SCSS

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

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

▼ CSS

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

♣ 色を操作する関数

Sass で計算できるのは、数値だけではありません。スタイルシートでよく用いる「色」も操作することができます。より明るい色にしたい、暗い色にしたいなど、そのための専用の関数が用意されています。

色を操作する関数

関数名	機能	実行例
lighten(\$color, \$amount)	明るい色を作成	lighten(#800, 20%) => #e00
darken(\$color, \$amount)	暗い色を作成	darken(#800, 10%) => #500
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
rgba(\$color, \$alpha)	RGB 値に透明度を指定	rgba(blue, 0.2) => rgba(0, 0, 255, 0.2)

例えば、`color: #800;` (赤茶色) を、もう少し明るい色にしたい場合には、`color: lighten(#800, 20%);` と書くことで、`color: #e000` (緋色) となり、明るくなります。

様々な関数が用意されていますので、使って行きましょう。

4.10 オリジナル関数の定義

色の操作をする関数はとても便利でした。そして、Sass では、自分で独自の関数を定義することもできます。

構文は、次の通りです。

▼ SCSS

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

`@function` で関数の宣言をします。独自の関数名を命名し、(必要であれば) 引数を設定します。関数の戻り値(計算結果)は、`@return` を使って返します。

例として、大きさを変更する独自関数を作ってみましょう。

▼ SCSS

```
// 引数で与えられた $value の $scale 倍を返す関数  
// ($size の 初期値は 1 にしている)  
@function change-size($value, $scale: 1) {  
  @return $value * $scale;  
}  
  
.small.card {  
  // 幅を 200px の 2/3 倍にする  
  width: change-size(200px, 2/3);  
}  
  
.card {  
  // 倍率の指定をしないときには 初期値の 1 が使われる  
  width: change-size(200px);  
}  
  
.large.card {  
  // 幅を 200px の 1.5 倍にする  
  height: change-size(200px, 1.5);  
}
```

▼ CSS

```
.small.card {  
  width: 133.3333333333px;  
}  
  
.card {  
  width: 200px;  
}  
  
.large.card {  
  width: 300px;  
}
```

カードの大小によって、幅を変えることができました。

♣ コメント

独自に定義した関数の場合、後から使う人のために、その関数がどのようなものなのか、働きや引数の型や単位の有無など使用方法が分かるようにコメントを残すようにしましょう。

ここでは、グラデーションを作成する関数を例に挙げます。 *10

▼ SCSS

```
// @function gradation グラデーションの値を返す
// @param {Color} $base-color 元の色
// @param {Number} $amount 0-100%の数値 既定値は20%
// @return {String} グラデーションの値を返す
@function gradation($base-color, $amount: 20%) {
  @return linear-gradient($base-color, darken($base-color, $amount));
}

.card {
  background: gradation(#0f0);
}
```

これをコンパイルすると、次の CSS が得られます。

▼ CSS

```
.card {
  linear-gradient(rgb(0, 255, 0), rgb(0, 153, 0));
}
```

.card クラスの背景色が、緑のグラデーションとなりました。

ここでは、基本的な Sass の使い方の例を取り上げましたが、他にも [公式サイト^{*11}](#) や、[Web Design Leaves^{*12}](#) では、様々な使い方が解説されています。より深い知識、情報が必要となった際には、きっと役立つことと思いますので、ご参考になれば幸いです。

^{*10} ここでは、自作する例を挙げましたが、「簡単に美しいグラデーションを作れる便利サイト 4 選」(https://www.infoctl.co.jp/staff_blog/webmarketing/40006/) のようなサイトもございます。ご活用下さい。

^{*11} <https://sass-lang.com/>

^{*12} https://www.webdesignleaves.com/pr/css/css_basic_08.html

第 5 章

軽量マークアップ言語 Markdown

Markdown(マークダウン) は、文書を記述するための軽量マークアップ言語の一つです。「書きやすくて読みやすいプレーンテキストとして記述した文書を、妥当な HTML 文書へと変換できるフォーマット」として、ジョン・グルーバーやアーロン・スワーツにより作成されました。

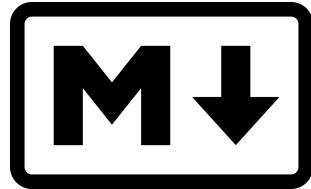
Markdown の記法の多くは、電子メールにおいてプレーンテキストを装飾する際の慣習から着想を得ています。

【この章の内容】

5.1	Markdown の特徴	86
5.2	段落	86
5.3	改行	86
5.4	テキストの装飾	86
5.5	見出し	87
5.6	箇条書き	87
5.7	引用	88
5.8	水平線	88
5.9	リンク	89
5.10	画像	90
5.11	表	90
5.12	コード	91
5.13	チェックボックス	91
5.14	注釈	91
5.15	HTML	92

5.1 Markdown の特徴

文書作成等に広く用いられている Markdown をご紹介いたします。
記法については様々なサイトで解説されていますが、ここでは比較的
良くまとまっているウィキペディア等を参考にご案内します。



- 簡潔な記法のため 容易に学習できる
- テキストベースで使い回しも容易
- 様々な活用が可能
 - メモ書きに
 - ブログ投稿に
 - スライド作成にも

5.2 段落

段落は1つ以上の連続したテキストで、空行によって分けられます。通常の段落をスペースやタブ
で字下げ(インデント)してはなりません。

▼ Markdown

これは段落です。2つの文があります。

これは別の段落です。ここにも2つの文があります。

5.3 改行

テキストに挿入された改行は、最終的な結果から取り除かれます。強制的に改行したい場合は、行
末に2つのスペースを挿入すればよいです。

5.4 テキストの装飾

以下のように記することで、テキストを装飾できます。* や _ などの記号の前後には、半角スペース
が必要です。

▼ Markdown

強調

斜体

強い強調

--太字--

~~打ち消し線~~

5.5 見出し

HTML の見出しあは、テキストの前にいくつかの # を置くことで作ることができます。

▼ Markdown

```
# レベル1の見出し (=h1)
## レベル2の見出し (=h2)
### レベル3の見出し (=h3)
#### レベル4の見出し (=h4)
##### レベル5の見出し (=h5)
##### レベル6の見出し (=h6)
```

h1 と h2 は、以下のようにも書けます。

▼ Markdown

レベル1の見出し (=h1)

レベル2の見出し (=h2)

5.6 箇条書き

♣ 順序付箇条書き

番号は自動で振られるので、全て 1. 項目 とすれば良いです。

▼ Markdown

1. 順序付きリストのアイテム
 1. 順序付きリストのサブアイテム その壱
 1. 順序付きリストのサブアイテム その弐
 1. 順序付きリストのサブアイテム その参

1. 順序付きリストの別のアイテム
 1. 順序付きリストのサブアイテム その壱
 1. 順序付きリストのサブアイテム その弐
 1. 順序付きリストの別のアイテム
 1. 順序付きリストの別のアイテム
 1. 順序付きリストの別のアイテム

♣ 順不同箇条書き

- (ハイフン) *(アスタリスク) どちらの記号も使えます。結果は同じになります。

▼ Markdown

- 順不同リストアイテム
 - サブアイテムはタブもしくは4つのスペースで字下げ(インデント)する
 - 順不同リストの別のアイテム
-
- * 順不同しリストアイテム
 - * サブアイテムはタブもしくは4つのスペースで字下げ(インデント)する
 - * 順不同リストの別のアイテム

5.7 引用

> を行頭に付けることで、引用文になります。

▼ Markdown

宮沢賢治 ポラーノの広場 より引用

> あのイーハトーヴォのすきとおった風、夏でも底に冷たさをもつ青いそら、
> うつくしい森で飾られたモリーオ市、郊外のぎらぎらひかる草の波
>
> またそのなかでいっしょになつたたくさんのひとたち、
> ファゼーロとロザーロ、羊飼のミーロや、顔の赤いこどもたち、
> 地主のテーモ、山猫博士のボーガント・デストゥバーゴなど、
> いまこの暗い巨きな石の建物のなかで考えていると、
> みんなむかし風のなつかしい青い幻燈のように思われます。
> では、わたくしはいつかの小さなみだしをつけながら、
> しづかにあの年のイーハトーヴォの五月から十月までを書きつけましょう。

5.8 水平線

一行の中に、3つ以上の -(ハイフン) や *(アスタリスク) _ (アンダースコア)だけを並べると水平線が作られます。

ハイフンやアスタリスクの間には空白を入れられます。次の例はすべて水平線になります。

▼ Markdown

```
* * *
***  
*****  
- - -  
-----
```

5.9 リンク

リンクは次のように記述できます。

▼ Markdown

```
[リンクのテキスト](リンクのURL)
```

これは、HTML に変換すると次のようになります。

▼ HTML

```
<a href="リンクのURL">リンクのテキスト</a>
```

URL が長かったり、良く使うリンクには名前を付けることもできます。

▼ Markdown

```
[リンクテキスト][名前]  
[名前]:URL
```

使うときには、次のように使います。

▼ Markdown

```
[名前][]
```

♣ ページ内リンク

次のように書くと、ページ内リンクを作成できます。

▼ Markdown

```
[ポラーノの広場](#ポラーノの広場)  
[坊ちゃん](#坊ちゃん)  
[高瀬舟](#高瀬舟)
```

```
# ポラーノの広場
```

あのイーハトーヴォのすきとおった風、夏でも底に冷たさをもつ青いそら、うつくしい
森で飾られたモリーオ市、郊外のぎらぎらひかる草の波

またそのなかでいっしょになつたたくさんのひとたち、ファゼ一口とロザ一口、羊飼の
ミー口や、顔の赤いこどもたち、地主のテーモ、山猫博士のボーガント・デストゥパーゴな
ど、いまこの暗い巨きな石の建物のなかで考えていると、みんなむかし風のなつかしい青い
幻燈のように思われます。では、わたくしはいつかの小さなみだしをつけながら、しづかに
あの年のイーハトーヴォの五月から十月までを書きつけましょう。

坊ちゃん

親譲りの無鉄砲で小供の時から損ばかりしている。小学校に居る時分学校の二階から飛び
降りて一週間ほど腰を抜かした事がある。なぜそんな無闇をしたと聞く人があるかも知れぬ
。別段深い理由でもない。新築の二階から首を出していたら、同級生の一人が冗談に、いく
ら威張っても、そこから飛び降りる事は出来まい。弱虫やーい。と囁したからである。小使
に負ぶさって帰って来た時、おやじが大きな眼をして二階ぐらいから飛び降りて腰を抜かす
奴があるかと云ったから、この次は抜かさずに飛んで見せますと答えた。

高瀬舟

高瀬舟は京都の高瀬川を上下する小舟である。徳川時代に京都の罪人が遠島を申し渡され
ると、本人の親類が牢屋敷へ呼び出されて、そこで暇乞をすることを許された。それから罪
人は高瀬舟に載せられて、大阪へ廻されることであつた。それを護送するのは、京都町奉行
の配下にある同心で、此同心は罪人の親類の中で、主立つた一人を大阪まで同船させること
を許す慣例であつた。これは上へ通つた事ではないが、所謂大目に見るのであつた、默許で
あつた。

5.10 画像

次のように記述することで、画像を掲載できます。

▼ Markdown

```
![代替テキスト](画像のURL)
```

これを、HTML に変換すると、次のようになります。

▼ HTML

```

```

5.11 表

表は次のように書きます。

▼ Markdown

Left align	Center align	Right align
Apple	Apple	Apple
Banana	Banana	Banana
Cherry	Cherry	Cherry

5.12 コード

プログラムコードを掲載したい場合もあります。コードを含める場合、3つずつの ``` (バッククオート) でコード全体をくくります。開始を表すバッククオートの3つ目に続けて、任意で言語名を明記することができます。

例えば、Ruby のプログラムコードの場合には、次のように書けます。

▼ Markdown

```
``` ruby
 puts "Hello world"
```
```

5.13 チェックボックス

一部の環境とはなりますが、チェックボックスを作成することもできます。

▼ Markdown

- [] タスク1
- [x] タスク2

5.14 注釈

注釈を書くこともできます。注釈は、^ (ハット) の後に数字を記します。インターネット・プロトコル・スイート という技術用語に注釈を付けてみます。

▼ Markdown

インターネットとは、「インターネット・プロトコル・スイート」[^1]を使用し、複数のコンピュータネットワークを相互接続した、地球規模の情報通信網のことである。

[^1]: TCP/IPプロトコル・スイート とも呼ばれる、標準化された通信規約のこと。

5.15 HTML

本文中に、一部の `html` タグを記述することもできます。

▼ Markdown

```
<figcaption>図1 りんご</figcaption>
```

Markdown は、そのまま文章として読んでも見やすく、また記法も簡単です。少ない学習量で習得することができ、ちょっとしたメモ書きから、ブログの投稿やスライドの作成など様々な分野で使われています。是非、ご利用下さい。

第 6 章

簡潔な CSS 設計 FLOU

少し大きなウェブサイトになると、数百行～数千行にも渡るスタイルシートを書くことになります。一つの巨大なスタイルシートではとても扱いにくいので、機能ごと、部品ごとに分割して管理することになりますが、どのようにスタイルシートを設計し、管理すれば良いのでしょうか。

BEM など様々な CSS 設計思想がありますが、簡素で分かり易く使える FLOU(フロウ)^{*1} をご紹介いたします。^{*2}

【この章の内容】

6.1	どんな設計があるの？	94
6.2	F、L、O、U の 4 つに分けよう	95
6.3	FLOU 設計のメリットは？	99
6.4	FLOU で書くときのポイントは？	102
6.5	まとめ	103

^{*1} FLOCSS を扱いきれないあなたに贈る、スリムな CSS 設計の話 (<https://webnaut.jp/technology/20170407-2421/>) より、抜粋引用

^{*2} CSS が本質的にグローバルスコープであるため、その対応として生まれた BEM 等ですが、いまいち美しくないようを感じます。全ブラウザでサポートされたカスケードレイヤーに期待しています。CSS の新機能カスケードレイヤー「@layer」CSS をレイヤー化して扱え、今までの実装方法が大きく変わる！ (<https://coliss.com/articles/build-websites/operation/css/css-cascade-layers.html>)

Webの開発をやったことのある方なら誰しも、「CSSって結局どう書くのがベストなの？」という悩みを感じたことがあるでしょう。一見簡単なCSSですが、一度書き始めるとそのあまりの自由さに、まるで大海原に放り出された赤子のような気分になってしまいます。今日はCSSの設計について考えてみましょう。



6.1 どんな設計があるの？

CSS設計について調べてみると、OOCSS、SMACSS、FLOCSSなど、いろいろな設計思想があることがわかります。

- OOCSS [Slide Share – Object Oriented CSS^{*3}](#)
- SMACSS [SMACSS^{*4}](#)
- FLOCSS [GitHub – FLOCSS^{*5}](#)

FLOCSSを例を見てみましょう。概観としては、

- Foundation
- Layout
- Object

の三つがあって、さらにObjectの中に、

- Component
- Project
- Utility

^{*3} <https://www.slideshare.net/stubbornella/object-oriented-css>

^{*4} <https://smacss.com/>

^{*5} <https://github.com/hiloki/flocss>

があり、これらにCSSを分類して書くという設計となっています。

FLOCSS の設計に従うことで、

- CSS がコンポーネント化されるので使いまわしや修正に強くなり、
 - MindBEMding のネーミングルールによってクラスの役割が明確化され、
 - 全体として管理のしやすいソースコードが出来上がる

というメリットを得られます。とはいって、「ハイハイ、これに従って書けばいいのね」と書き始めたのも束の間、ほとんどの人は思うことでしょう。

- 「このパーツって Component と Project のどっちだ…」
 - 「Utility 使いまくらないと全然レイアウト実現できない…」
 - 「Layout に書くこと少なすぎ…」

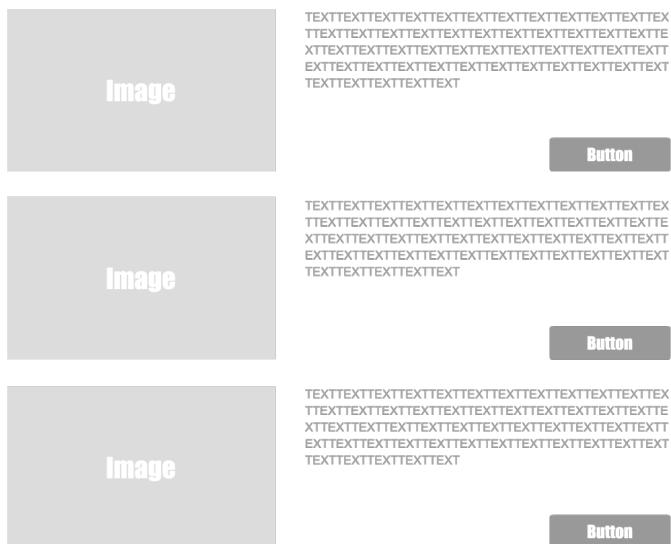
多くの Web サイト制作では、FLOCSS のポテンシャルを生かしきれずに、そのメリットがなんだかよくわからないまま CSS を書いていくことになってしまいがちです。

相当大規模なプロジェクトならまだしも、ページ数の知れている Web サイトなんかではここまで巨大な設計図を引かなくても、うまいこと組み立てられそうですよね。

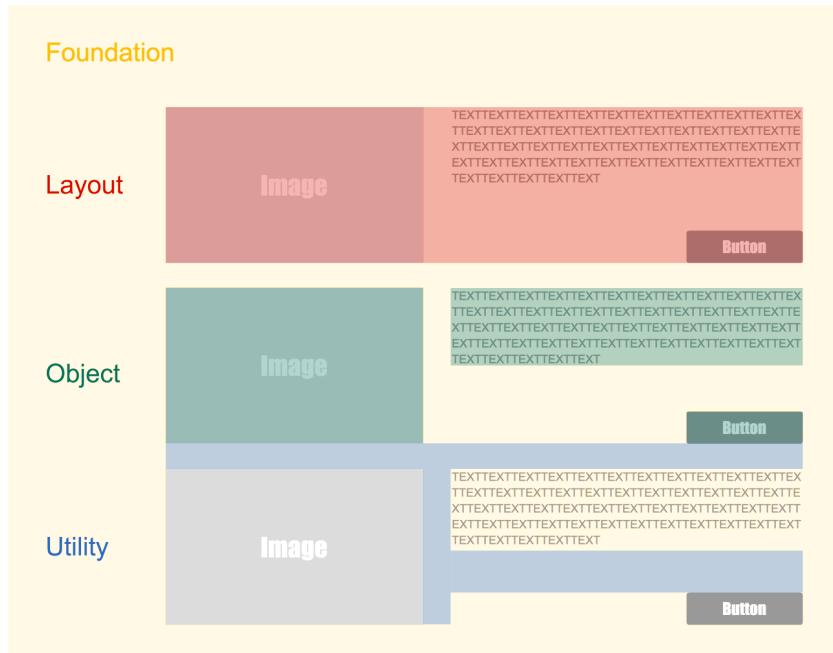
ということで、FLOCSS のいいところどりをした、よりスリムな設計「FLOU」を考えてみることにしましょう。

6.2 F、L、O、Uの4つに分けよう

基本のアーキテクチャとしては、Foundation、Layout、Object、Utility の 4 つに分けましょう。例えばこんな組み方にしたいとき、



それぞれの役割は、図にするとこのようになります。



F、L、O、U、の4つが、Webページに必要な要素を過不足なく担当できているのがわかると思います。具体的なコーディングルールは、下記のような感じです。

 Foundation

- CSS リセットなど、すべてのベースとなる CSS
 - 基本的にコードは追加しない
 - foundation.scss に記述

▼ リスト 6.1: modern-css-reset.css

```
/* Box sizing rules */
*,
*:before,
*:after {
  box-sizing: border-box;
}

/* Remove default margin */
body, h1, h2, h3, h4, p, figure, blockquote, dl, dd {
  margin: 0;
}

/* Remove list styles on ul, ol elements with a list role, which suggests default
styling will be removed */
ul[role="list"],
ol[role="list"] {
  list-style: none;
}
```

```

/* Set core root defaults */
html:focus-within {
  scroll-behavior: smooth;
}

/* Set core body defaults */
body {
  min-height: 100vh;
  text-rendering: optimizeSpeed;
  line-height: 1.5;
}

/* A elements that don't have a class get default styles */
a:not([class]) {
  text-decoration-skip-ink: auto;
}

/* Make images easier to work with */
img,
picture {
  max-width: 100%;
  display: block;
}

/* Inherit fonts for inputs and buttons */
input,
button,
textarea,
select {
  font: inherit;
}

/* Remove all animations and transitions for people that prefer not to see them */
>/
@media (prefers-reduced-motion: reduce) {
  html:focus-within {
    scroll-behavior: auto;
  }
  *,
  *::before,
  *::after {
    animation-duration: 0.01ms !important;
    animation-iteration-count: 1 !important;
    transition-duration: 0.01ms !important;
    scroll-behavior: auto !important;
  }
}

```

♣ Layout

- パーツの配置や、ラッパーとしての幅や高さなどを決定するクラス
- layout.scss に記述

```
.justify-left {  
    display: grid;  
    justify-content: flex-start;  
}  
  
.justify-right {  
    display: grid;  
    justify-content: flex-end;  
}  
  
.justify-center {  
    display: grid;  
    justify-content: center;  
}
```

♣ Object

- ページをまたいで使われる各種パーツを定義するクラス
- そのパーツ内で常に同様の振る舞いをするものに関してのみスタイルを定義
- object.scssに記述

```
.boxA {  
    width: 100px;  
    height: 100px;  
    color: red;  
}  
  
.boxB {  
    width: 200px;  
    height: 200px;  
    color: green;  
}
```

♣ Utility

- 調整用のクラス
- margin、padding、font-size、colorなどを付与するのに使用
- 他種類のパーツ間の空き調整や、パーツとして認められないような、自由な振る舞いをする要素に対してはこちらのクラスを使用
- utility.scssに記述

```
.mt10 { margin-top: 10px; }  
.mt20 { margin-top: 20px; }  
.mt30 { margin-top: 30px; }
```

この設計を、F、L、O、Uの頭文字をとって、ここでは便宜的にFLOUと呼ぶことにします。

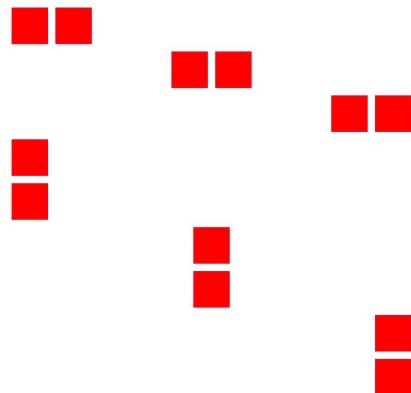
6.3

FLOU 設計のメリットは？

さて、この FLOU の設計を使うとどんなハッピーなことがあるのでしょうか。

♣ レイアウトが劇的に楽

まず言えるのが、「モジュールの配置についていちいち CSS を考える必要がなくなる」ということです。Layout に適切なクラスを用意しておくことによって、モジュール（Object）たちを柔軟に、かつスピーディーに配置していくことが可能になります。例えば、CSS Grid を使った Layout クラスを使えば、右の図のように柔軟にモジュールを配置できます。



▼ index.html

```
<div class="justify-left">
  <div class="box"></div>
  <div class="box"></div>
</div>

<div class="justify-center">
  <div class="box"></div>
  <div class="box"></div>
</div>

<div class="justify-right">
  <div class="box"></div>
  <div class="box"></div>
</div>

<div class="justify-left direction-row">
  <div class="box"></div>
  <div class="box"></div>
</div>

<div class="justify-center direction-row">
  <div class="box"></div>
  <div class="box"></div>
</div>

<div class="justify-right direction-row">
  <div class="box"></div>
  <div class="box"></div>
</div>
```

▼ layout.scss

```
.justify-left {
  display: grid;
  grid-auto-flow: column;
  justify-content: start;
}

.justify-center {
  display: grid;
  grid-auto-flow: column;
  justify-content: center;
}

.justify-right {
  display: grid;
  grid-auto-flow: column;
  justify-content: end;
}

.direction-row {
  display: grid;
  grid-auto-flow: row;
}

.box {
  display: inline;
  width: 50px;
  height: 50px;
  margin: 5px;
  background: red;
}
```

ここで使っている `.justify-left` などのクラスは、子要素に依存しないレイアウトのクラスとなるので、どんなモジュールに対しても使い回すことができます。^{*6}

また、配置のためのクラス以外にも、

▼ layout.scss

```
.w25 { width: 25%; }
.w50 { width: 50%; }
.w100 { width: 100%; }
.h25 { height: 25%; }
.h50 { height: 50%; }
.h100 { height: 100%; }
```

のように、ラッパーの幅や高さを指定したりする Layout クラスを用意しておくのも強力です。このように、レイアウトに関する記述を Layout クラスとして切り出し、使い回すことで、Object 内で定義するモジュールを柔軟に配置することができるようになります。

^{*6} 依存を完全に0にしているわけではありません。

♣ CSS の見通しが良くなる

FLOU を使う二つ目の利点として、「ファイル全体の見通しが良くなる」というものがあります。例えば、

▼ object.scss

```
.boxA {
  width:      100px;
  height:     100px;
  color:      #ff0000;
  margin-top: 20px;
}

.boxB {
  width:      200px;
  height:     200px;
  color:      #00ff00;
  margin-top: 20px;
}

.textA {
  color:      #ff0000;
  font-weight: bold;
  margin-top: 20px;
}
```

とするよりは、

▼ utility.scss

```
.mt20 {
  margin-top: 20px;
}
```

▼ object.scss

```
.boxA {
  width: 100px;
  height: 100px;
  color: #ff0000;
}

.boxB {
  width: 200px;
  height: 200px;
  color: #00ff00;
}

.textA {
  color: #ff0000;
  font-weight: bold;
}
```

として、各モジュールに html 上で

▼ index.html

```
<div class="boxA mt20">  
  boxA  
</div>
```

のように .mt20 を付与する方が、全体の見通しが良くなります。

ここでは、単に

- それぞれのモジュールに対する記述が減ったので見やすくなった。

ということに加えて、

- それぞれのモジュールにとって本質的でない記述が減った。

ということも重要です。

上の例で記述されていた margin-top: 20px; は他のモジュールとの関係を定義するのに必要なだけあって、それぞれのモジュールには直接的には関係がない記述です。

このように、モジュール自体が他のモジュールとの関係に関するプロパティを持っていると、想定してなかったモジュールの組み合わせによって、「上の空きが大きすぎる」や「横並びになってくれない」などの不具合が生じかねません。

一方 FLOU の設計であれば モジュール (Object) 配置 (Layout) 調整 (Utility) を分けたことで、

- モジュールのデザイン修正をしたい場合は Object クラスをいじって、
- 配置に関しては Layout クラスの変更で対応し、
- モジュール間の空きなどを調整したい場合は Utility クラスを変更すればよい。

ということになるので、仕様の変更に強くなります。

オブジェクトの修正でない、例えばレイアウトの変更や、数 px 単位の微妙な空き調整なんかは、結局 html 上でのクラスの付け替え作業になるので、他のページへの影響をケアする必要がなくなるというのも嬉しいところです。

6.4

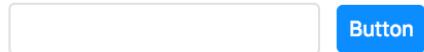
FLOU で書くときのポイントは？

FLOU では、どの粒度でモジュールを切り出すかがポイントです。理想的な切り出し方は、

- できるだけ小さい単位で切り出す
- しかし、常にセットで使うものに関しては一つのモジュールにまとめる

というところです。

例えば、こんな部品を作るとします。



この場合は、フォームとボタンとで一塊のモジュールとして切り出すのではなく、それぞれ別々の部品として切り出す方が良いでしょう。

△ セットで切り出す



◎ 細かく切り出す



なぜなら、もし「やっぱりこのページだけはフォームとボタンを縦並びに変更したい」となったときに、セットで切り出していた場合は、

- 従来の「フォームとボタンが横並びのモジュール」に加えて、
- 新規の「フォームとボタンを縦並びのモジュール」を CSS ファイルに追加する

ということになりますが、別々で切り出していくれば、

- html 上で、子要素を横並びにする Layout クラスから
子要素を縦並びにする Layout クラスに変更する

という対応だけで済み、モジュールのパターン（今回で言う「フォーム」と「ボタン」）を CSS 上で増やすことなくデザインが実現できるからです。

このように、モジュールはできるだけ小さい単位で管理し、配置に関しては Layout クラスと Utility クラスに任せることで、CSS の肥大化を抑えることができるのです。

とはいっても、「このパーツはどんな場面でも一緒に使う」と言い切れるものに関しては、一緒に管理するのが良いでしょう。

6.5 まとめ

CSS の設計に関しては、SMACSS や FLOCSS などのオブジェクト指向の設計に従うことで、メンテナンス性に優れたコードを書くことができます。

しかし、どの粒度でオブジェクトを切り出すのか、またオブジェクト以外のクラスはどう扱うかなどはサイトの特性によってまちまちなので、「どんなサイトにでも使える万能な設計手法」というものは存在しません。

そんなときは、今回ご紹介したミニマムな設計である「FLOU」をベースに最適解を模索し、自己流の設計としてアレンジしてみてはいかがでしょうか！

【コラム】Sassによる実装例

FLOUは次の4つの層で構成されます。

1. Foundation - reset / normalize / base...
2. Layout - justify-left / center / right...
3. Object - header/main/sidebar/footer...
4. Utility - margin / padding ...

次はSassを採用した場合の例です。

▼ SassによるFLOUの実装例

```
└── foundation
└── layout
└── object
    ├── _footer.scss
    ├── _header.scss
    ├── _main.scss
    └── _sidebar.scss
└── utility
    ├── _margin.scss
    ├── _padding.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っています。

`style.css.scss`など、個々のスタイルシートの例を挙げます。

▼ style.css.scss

```
// FLOUの設計思想に従い、サイト全体で使うスタイルシートを取り込む
@import "foundation/foundation";
@import "layouts/layout";
@import "objects/object";
@import "utilities/utility";
```

▼ foundation.scss

```
// リセットCSSなど、すべての基盤となるスタイルシート

// A modern CSS reset
@import "modern-css-reset";

// font-family の指定
$font-family: "Helvetica Neue",
    Arial,
    "Hiragino Kaku Gothic ProN",
    "Hiragino Sans", Meiryo,
    sans-serif;
html { font-family: $font-family !important; }
```

▼ layout.scss

```
// 部品の配置や、ラッパーとしての幅や高さなどを決定するクラス

// 左寄せ指定
.lefted.text {
    text-align: left !important;
}
```

▼ object.scss

```
// ページを跨いで使われる各種部品を定義するクラス
@import 'footer';
@import 'header';
```

▼ utility.scss

```
// 間隔や色など、各種調整用のクラス
@import "margin";
@import "padding";
```


付録 A

基本的な ショートカット & コマンド

GUI によるマウス操作は直感的ですぐに習得できます。そして良く使う操作は、特定のキー入力の組み合わせで実現できるように用意された「ショートカットキー」を使うと速くて楽にできるようになります。またコマンド操作が出来ることも生産性向上に繋がります。

習得したい基本的なショートカットとコマンドのご紹介です。是非、お役立てください。

A.1 Mac の為の ショートカット

♣ 入力ソース切替

キーバインド	動作
Ctrl + Space	日本語入力と英数入力を切替

♣ ファイル操作

キーバインド	動作	意味
Command + O	ファイルを開く	Open 開く
Command + S	ファイルを保存する	Save 保存する

♣ テキスト編集

キーバインド	動作	意味
Command + Z	元に戻す	
Command + X	切り取り	
Command + C	コピー	Copy
Command + V	貼付	
Command + A	全選択	All
Ctrl + H	カーソルの左の文字を削除	
Ctrl + D	カーソルの右の文字を削除	Delete
Ctrl + K	カーソル以降を切り取り	Kill line
Ctrl + T	カーソル前後の文字を入れ替	Transpose 転置

♣ カーソル移動

キーバインド	動作	意味
Ctrl + F	カーソルを右へ移動	Forward 先へ進んで
Ctrl + B	カーソルを左へ移動	Backward 後方へ
Ctrl + P	カーソルを上へ移動	Previous 以前の行へ
Ctrl + N	カーソルを下へ移動	Next 次の行へ
Ctrl + A	カーソルを行頭へ移動	Ahead 前方へ
Ctrl + E	カーソルを行末へ移動	End 行末へ

A.2 Atom の為の ショートカット

キーバインド	動作	意味
Shift + Command + D	行の複製	Duplicate
Shift + Ctrl + K	行の削除	Kill line
Command + /	コメントアウト	
Ctrl + F	検索 (ファイル内)	Find 見つけ出す
Shift + Ctrl + F	検索 (プロジェクト内)	Find 見つけ出す
Ctrl + G	任意行へカーソルを移動	Go
Command +]	インデントを追加	
Command + [インデントを削除	
Command + K →	画面分割	
Command + W	画面を閉じる	Window close
Shift + Ctrl + P	コマンドパレット	Pallet
Shift + Command + C	色を選択	pigments

A.3 基本的なコマンド一覧

♣ ls (LiSt)

カレントディレクトリにあるファイルやディレクトリを表示

コマンド	意味
ls	ファイルやディレクトリを表示
ls -l	ファイルやディレクトリを詳細表示
ls -a	ファイルやディレクトリを表示 (隠しファイル含む)
ls -la	ファイルやディレクトリを詳細表示 (隠しファイル含む)

♣ cd (Change Directory)

カレントディレクトリを移動

コマンド	意味
cd source	source ディレクトリに移動
cd ~ /	ホームディレクトリに移動
ls ..	親ディレクトリに移動

♣ pwd (Print Working Directory)

カレントディレクトリを表示

コマンド	意味
pwd	カレントディレクトリを表示

♣ touch

タイムスタンプ更新や空ファイル作成

コマンド	意味
touch index.html	index.html ファイルを作成

♣ mkdir (MaKe DIRectory)

新しいディレクトリを作成

コマンド	意味
mkdir stylesheets	stylesheets ディレクトリを作成

♣ mv (MoVe)

ファイル名変更や移動

コマンド	意味
mv master.css style.css	master.css から style.css にファイル名変更
mv style.css stylesheets	style.css を stylesheets ディレクトリに移動

♣ cp (CoPy)

ファイルをコピーする

コマンド	意味
cp master.css style.css	master.css を style.css にコピー
cp style.css stylesheets	style.css を stylesheets ディレクトリにコピー
cp -r stylesheets /tmp/	stylesheets ディレクトリを /tmp/ ディレクトリに再帰的 (Recursive) にコピー

♣ rm (ReMove)

ファイル削除

コマンド	意味
rm master.css	master.css を削除
rm -rf stylesheets	stylesheets ディレクトリを削除 (要注意)

♣ open

ターミナルから Finder でファイルを開く

コマンド	意味
open .	カレントディレクトリを Finder で開く
open ~/	ホームディレクトリ Finder で開く
atom ./	カレントディレクトリを Atom で開く (参考)

♣ source

ファイル内のコマンドを現在のシェルで実行

コマンド	意味
source ~/zshrc	~/zshrc を読み込み 設定を反映する

♣ history

コマンド履歴を表示

コマンド	意味
history	コマンド履歴を表示
!7	履歴 7 番のコマンドを実行
Command + P	直前のコマンドを表示 Enter で実行

♣ cat (conCATenate)

ファイル内容表示及び連結

コマンド	意味
cat a.txt	内容を表示
cat -n a.txt	行番号付きで内容を表示
cat a.txt b.txt > c.txt	結合して c.txt に書込

【コラム】金の延棒クイズ 【解答】

最後までお読みください、ありがとうございます。金の延棒クイズの解答です。

2回鉢を入れて、金の延棒を 1 と 2 と 4 の大きさに分割します。

一日目のお支払いには、1 の延棒を渡します。

二日目のお支払いには、2 の延棒を渡して、先に渡した1 の延棒は返してもらいます。

三日目のお支払いには、1 の延棒も渡します。

四日目のお支払いには、大きな4 の延棒を渡し、2 と 1 の延棒は返してもらいます。

五日目のお支払いには、1 の延棒も渡します。

六日目のお支払いには、2 の延棒を渡して、先に渡した1 の延棒は返してもらいます。

七日目のお支払いには、全ての延棒を渡します。

延棒の有無を 0 と 1 で表すと二進数と対応しています。

意外なところに潜む二進数。探してみてくださいね。

金の延棒	日当
421	
001	1
010	2
011	3
100	4
101	5
110	6
111	7

終わりに

本書では、静的サイトジェネレータ Middleman を中心にいろいろご紹介いたしましたが、いかがでしたでしょうか？ 皆様のウェブライフにお役立て頂ければとても仕合せです。

「福祉」。「福」「祉^{*1}」どちらも「めぐみ、さいわい」という意味を持ちます。

「熱き心、^{たくま}逞^{かしいな}しき腕、冷静な頭脳」

学生時代に言われた言葉ですが、福祉を生きる者は、人としての熱い思い、暖かい心を持ち、その上で、冷静な判断力を以て、力強く行動するのだと。

「工学」の「工」は、「天の^{ことわり}理^{かがい}」を、地に下ろす」意味です。

技術の産物としての社会ではなく、世界を^{かがいや}耀^{かがや}かせるために技術を用いてください。技術に使われるのではなく、技術を使いこなし、人の道に役立てる人となってください。

令和の御世を生きる皆さんが素晴らしい人生を生き、素晴らしい日本を創ることを願って筆を置きます。

いやさか
彌榮

*1 「祉い」と書いて、「さいわい」と読みます。天からの恵みがその身に止まる意味です。

静的サイトジェネレータで楽しく創るウェブサイト

令和五年三月三日

著 者 アトリエ未来

発行者 早乙女 遙香

連絡先 contact@atelier-mirai.net

<https://atelier-mirai.net>

© 令和五年 アトリエ未来