

目 次

まえがき	3
Remark で広げる Markdown の世界 緑豆はるさめ	4
Vivliostyle で縦組シナリオを組版する 小形 克宏	13
EPUB ファイルから Vivliostyle で PDF を作りたい！ 田嶋 淳	31
CSS で View を定義する意義 アカベコ	44
Vivliostyle のこれから開発課題 村上 真雄	50
あとがき	60

まえがき

「Vivliostyle で本を作ろう Vol. 2」をお手にとっていただきありがとうございます。Vivliostyle は、Web ページのレイアウトに用いられる CSS の技術を使い、紙のページをレイアウトする CSS 組版を実現するプロジェクトで、AGPL ライセンスのフリーソフトウェアとして公開されています。私のような Web エンジニアにとって CSS 組版が馴染み深いのは当然のことながら、最近ではそれ以外の業界から CSS 組版が選ばれるようになっています。なぜ Word や LaTeX、InDesign などの盤石な組版ソフトウェアがある中、わざわざ CSS 組版という手段をとるのか？本書で紹介する様々な事例を見れば、その理由がわかることでしょう。

そこのあなた！次の本は Vivliostyle で作ってみませんか？

免責

本書の内容に関して、正確性は保証できません。本書に書かれた情報を利用することで生じた結果に対して、著者はいかなる責任も負いません。それはそうとして、もしこの本を読んで気づいたことや感想などがありましたら、編集者 Twitter アカウント [@spring_raining](https://twitter.com/spring_raining) や harusamex.com@gmail.com、各寄稿者へぜひご連絡下さい。きっと喜んで答えてくれることと思います。

無料公開について

本書は電子版が Vivliostyle サンプルページ [1] に全篇が公開されています！実際に Vivliostyle viewer を介してブラウザの印刷機能を使うことで、今ご覧になっている紙面をそのまま PDF として保存することもできます。また、本書の CSS やビルドスクリプトも GitHub [2] しているので、ぜひ参考にしてください。

[1] <https://vivliostyle.org/ja/samples/>

[2] <https://github.com/spring-raining/tbf07-draft>

Remark で広げる Markdown の世界

緑豆はるさめ (@spring_raining)

はじめましてこんにちは、はるさめです。本誌は名目上 Vivliostyle について紹介する同人誌なのですが、またしても空気を読まず Vivliostyle ではない OSS プロジェクト「Remark」について紹介したいと思います。

Remark

Remark とは「Markdown processor」という紹介文の通り、Remark で書かれたテキストを読み込み様々な変換を施すことができる JavaScript 製のライブラリです。Remark は様々なライブラリと組み合わせて目的の形式のテキストに変換でき、**Rehype** と一緒に使うことで Markdown を HTML に変換できます。同様の処理をしてくれるライブラリとしては Marked.js が有名ですが、Remark の強力な機能は、Markdown を **抽象構文木 (AST)** に変換することで、より柔軟に構文を改造できる点です。^[1]

なお、非常に紛らわしいのですが、GitHub 上で検索すると「Remark」という名前のプロジェクトが 2 つ見つかります。今回紹介するプロジェクトは `gnab/remark` ではなく、`remarkjs/remark` のほうです。公式サイトも <https://remarkjs.com> ではなく <https://remarkjs.org> なので気をつけてください。

Unified エコシステム

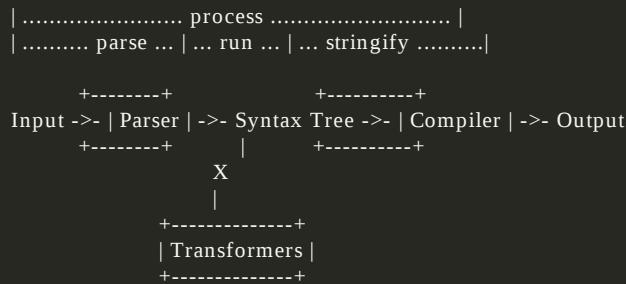
Remark をはじめとしたライブラリ群は役割ごとに細かくパッケージ化されており、それぞれの目的に応じて複数を組み合わせて用います。それぞれのパッケージは総称して **Unified** というプロジェクトに属しているのですが、各パッケージの役割が少しづつ分かれています。

- **remark/rehype** : Markdown/HTML の構文を解析・構築する処理系。それそれに **parse** と **stringify** の 2 つのパッケージがあり、**remark-parse** は Markdown から mdast をへ変換でき、**remark-stringify** は mdast からテキストの Markdown を構築できる

[1] Haskell 製のライブラリ **Pandoc** も同様の方針で実装されたテキスト変換ツールで、様々な形式のテキストを入力・出力できます

- **mdast/hast** : Markdown/HTML の構文を解析して得られる AST の仕様。それぞれの仕様は **unist** という仕様を拡張して定義されており、GitHub 上では **syntax-tree** という Organization で管理されている
- **remark-rehype/rehype-remark** : mdast (hast) から hast (mdast) に変換するパッケージ。実際の処理は **mdast-util-to-hast/hast-util-to-mdast** が実施している。
- **vfile** : ファイルのパス情報を抽象化して管理するパッケージ
- **unified** : Unified ファミリーの複数のライブラリを合成し、処理を実行する関数を得るためのパッケージ

これらの処理をまとめた、unified の README にある便利な図を引用します。



Remark を使って文章を変換しようとする際は、`parse`、`run`、`stringify` の 3 つの処理を経ることになります。例えば Markdown から HTML へ変換する際は、**remark-parse** を使って mdast 形式に解析し、**remark-rehype** を使って hast 形式に変換した後 **rehype-stringify** で HTML 形式のテキストを出力します。

Markdown をペースしてみる

まずは Remark を使って Markdown を HTML に変換するところから始めてみます。ひとまず以下のパッケージをインストールします。 [2]

```
npm i -s unified remark-parse remark-rehype rehype-stringify
```

[2] 以降の例では Node.js 上での実行を前提としますが、Remark はブラウザ上でも問題無く動作します

インストールした Remark を使って簡単な Markdown をパースしてみます。以下のコードは、`input` に Markdown で記述した文字列を用意しており、`processor.process()` で HTML に変換しています。

```
const unified = require('unified');
const markdown = require('remark-parse');
const remark2rehype = require('remark-rehype');
const html = require('rehype-stringify');

const processor = unified()
  .use(markdown)
  .use(remark2rehype)
  .use(html);
const input = `
# はじめてのRemark
**太字**_斜体_テキスト
`;
processor.process(input).then(({ contents }) => {
  console.log(contents);
});
```

このコードを実行すると、以下の出力が得られます。まさに期待した通りの HTML です！

```
<h1>はじめてのRemark</h1>
<p><strong>太字</strong><em>斜体</em>テキスト </p>
```

「Unified エコシステム」で紹介した通り、`processor` は `parse`、`run`、`stringify` を連続して実行したものです。以下のコードで、`parse` 終了時と `run` 終了時の内容を見てみましょう。

```
const inspect = require('unist-util-inspect');

const parsed = processor.parse(input);
console.log(inspect(parsed));
const transformed = processor.runSync(parsed);
console.log(inspect(transformed));
```

2回のコンソール出力では、それぞれ以下のように Markdown と HTML の AST が確認できます [3]。それぞれの AST について、少し踏み込んで見ます。

Markdown の AST (mdast)

```

root[2] (1:1-4:1, 0-30)
  |-- heading[1] (2:1-2:14, 1-14) [depth=1]
    └-- text: "はじめてのRemark" (2:3-2:14, 3-14)
  |-- paragraph[3] (3:1-3:15, 15-29)
    |-- strong[1] (3:1-3:7, 15-21)
      └-- text: "太字" (3:3-3:5, 17-19)
    |-- emphasis[1] (3:7-3:11, 21-25)
      └-- text: "斜体" (3:8-3:10, 22-24)
    └-- text: "テキスト" (3:11-3:15, 25-29)

```

ASTはその名の通り、木構造で構成されています。木構造とは、ノードと呼ばれる項目同士が紐付いて1つになった構造のことで、1つの根 (root) ノードが1つ以上の子ノードを持ち、そのノードがまた別の子ノードを持つ...というつながりにより木のように広がる構造を持ちます。

mdastでは、葉（子ノードを持たないノード）は基本的に `text` というノードになり、`heading`（見出し）、`paragraph`（段落）のようにそれぞれの属性を表すノードが親となります。また、`heading` の `depth`（見出しの大きさ）のようにそのノード自体にも任意の情報を持たせることができます。また、各ノードの (1:1-4:1, 0-37) のような数字は、そのノードが元の文章の何行目・何文字目にあたるかを表しており、この情報を使って元文章に注釈をつけるといった活用ができるようになります。

HTML の AST (hast)

```

root[3] (1:1-4:1, 0-30)
  |-- element[1] (2:1-2:14, 1-14) [tagName="h1"]
    └-- text: "はじめてのRemark" (2:3-2:14, 3-14)
  |-- text: "\n"
  |-- element[3] (3:1-3:15, 15-29) [tagName="p"]
    |-- element[1] (3:1-3:7, 15-21) [tagName="strong"]
      └-- text: "太字" (3:3-3:5, 17-19)
    |-- element[1] (3:7-3:11, 21-25) [tagName="em"]
      └-- text: "斜体" (3:8-3:10, 22-24)
    └-- text: "テキスト" (3:11-3:15, 25-29)

```

[3] `unist-util-inspect` は AST を分かりやすく表示させるユーティリティーです。`inspect` せずに表示させると、同じ構造を持つ Object が得られます。

`hast` も基本的な構造は同じですが、`paragraph` などの代わりに `element` というノードが用いられています。この構造を見てピンと来たかと思いますが、`hast` は実のところ HTML のタグの構造と全く同じです。`tagName` はそのノードが何の HTML タグに置き換わるかを示しています。

Parser を拡張してみる

AST による文章の構造化により、Remark は要求に応じて構文を定義することが簡単にできることが特徴です。それでは、本章で実際に独自の Markdown を作ってみます。今回はふりがなをふることができる「ルビ」を独自の構文で定義してみましょう。ルビを表現するための構文は色々と考えられますが、今回は某小説投稿サイトにしたがって以下のようないくつかのルールの構文を作ります。

- 縦棒 `|` でルビの開始地点を表す
- 二重山括弧 `《》` の中にルビ本体を記述する

すなわち、禁書目録を表すためには `| 禁書目録《インデックス》` のように書く、ということになります。

Parser を拡張するためにはいくつか方法がありますが、今回は `remark-parse` の動作に手を加えることで実現しようと思います。まず、Markdown を mdast 形式に変換する際にこの構文を正しく解析できるよう、プラグインとなる関数を用意するところから始めます。

```
function rubyAttacher() {
  const { Parser } = this;
  if (!Parser) {
    return;
  }
  const {inlineTokenizers, inlineMethods} = Parser.prototype;
}
```

プラグイン関数中の `remark-parse` 実装は `Parser` として定義されており、その中でも `Tokenizer` (字句解析; 文字列を適切な箇所で区切る処理) は大別して `blockTokenizers` と `inlineTokenizers` という名前で用意されます。今回のルビを字句解析する処理は、`inlineTokenizers` に追加します。

```

function rubyLocator(value, fromIndex) {
  return value.indexOf(' | ', fromIndex);
}
function rubyTokenizer(eat, value, silent) {
  if (value.charAt(0) !== ' | ') {
    return;
  }
  const rtStartIndex = value.indexOf(' <');
  const rtEndIndex = value.indexOf('> ', rtStartIndex);
  if (rtStartIndex <= 0 || rtEndIndex <= 0) {
    return;
  }
  const rubyRef = value.slice(1, rtStartIndex);
  const rubyText = value.slice(rtStartIndex + 1, rtEndIndex);
  if (silent) {
    return true; // Silentモードはconsumeせずtrueを返す
  }
  const now = eat.now(); // テキスト中の現在の位置を取得
  now.column += 1;
  now.offset += 1;
  return eat(value.slice(0, rtEndIndex + 1))({
    type: 'ruby',
    rubyText,
    children: this.tokenizeInline(rubyRef, now),
    data: { hName: 'ruby' },
  });
}

```

remark-parse でオリジナルの字句解析を実装するためには、**locator** と **tokenizer** の 2 つの関数が必要です。 locator はその文法が何文字目から始まるかを指示する関数で、 tokenizer で実際に文字列を区切る処理を実装します。ここでの locator は、単にルビの開始地点（ | の位置）を返すだけの関数です。

上記の tokenizer は、《 と 》の位置を元にルビの対象となるテキスト `rubyRef` とルビの内容 `rubyText` を取り出す処理を書いたものです。`eat` という関数は tokenizer を読みすすめるための関数で、引数にルビとして consume (消費) する文字列を与えることでその分だけ字句解析を進めます。`eat` の返値は関数になっており、消費した文字列に対応する mdast のノードを与えることで AST の構文木に追加できます。また、`now` は tokenizer の開始地点が元の文章のどの位置に対応するかを表します。参考コードのように、読み進める文字数だけ `column` と `offset` の値を更新した上で `tokenizeInline` に与えることで、再帰的に実行される tokenizer の位置情報を更新しています。

```
function rubyAttacher() {
  ...
  rubyTokenizer.locator = rubyLocator;
  inlineTokenizers.ruby = rubyTokenizer;
  inlineMethods.splice(inlineMethods.indexOf('text'), 0, 'ruby');
}
```

定義した locator と tokenizer を利用するよう rubyAttacher を修正します。 inlineMethod には適用する tokenizer の名前が配列で示されており、配列内の順番がそのまま tokenizer を実行する順番 (=優先順位) になります。これでプラグイン関数は完成です。 [4]

```
const processor = unified()
  .use(markdown)
  .use(rubyAttacher)
  .use(remark2rehype)
  .use(html);
```

プラグインは unified のメソッドチェーンに付け加えるだけで利用できます（順番に気をつけてください）。早速 mdast へのパース結果を見てみましょう。

```
root[1]
└─ paragraph[2]
    └─ text: "とある魔術の"
        └─ ruby[1] [rubyText="インデックス"][data={"hName":"ruby"}]
            └─ text: "禁書目録"
```

正しくパースされているようです！ ルビの内容を rubyText に入れることで、このあと HTML への変換時に用いることができます。また、 hName という名前のプロパティは remark-rehype が読み取りに対応しており、HTML 変換時のタグ名を指定できます。

Transformer を拡張してみる

次に、解析された構文木を正しく HTML に変換するため Transformer を改造します。 remark-rehype にはオプションとして handler を追加できるため、ルビ用の handler を用意する形で実装します。

[4] 参考コードの rubyTokenizer はとても簡単なケースにしか対応しておらず、まだまだ改善すべき点があります。例えばルビの入れ子が含まれた場合どうなるでしょうか？

```

const all = require('mdast-util-to-hast/lib/all');
const u = require('unist-builder');

function rubyHandler(h, node) {
  const rtStart = node.children.length > 0
    ? node.children[node.children.length - 1].position.end
    : node.position.start;
  const rtNode = h(
    {
      start: rtStart,
      end: node.position.end,
    },
    'rt',
    [u('text', node.rubyText)]
  );
  return h(node, 'ruby', [...all(h, node), rtNode]);
}

```

`h` は mdast ノードから hast ノードへ変換する関数となっており、この関数の引数として実際の HTML タグ名などを指定します。`all` はすべての子ノードに `h` を適用するヘルパー関数で、`u` もまた unist ノードを作成するヘルパー関数です。この例では、`rtNode` という新しいノードを作成し、それを `<ruby>` タグのノードの子として挿入していることがわかります。

作成した `handler` は、以下の形式で利用します。オプションとして `handlers` にオブジェクト形式で与えることで、名前の一致する mdast ノードはこの関数を通して hast ノードが生成されるようになります。

```

const processor = unified()
  .use(markdown)
  .use(rubyAttacher)
  .use(remark2rehype, {
    handlers: { ruby: rubyHandler },
  })
  .use(html);

```

すると、出力される hast は以下のようになります。

```

root[1]
└─ element[2] [tagName="p"]
  ├─ text: "とある魔術の"
  └─ element[2] [tagName="ruby"]

```

```
  |- text: "禁書目録"
    |- element[1] [tagName="rt"]
      |- text: "インデックス"
```

handler 無しでは存在しなかった `rt` タグが追加されました！これにより、最終的に以下の HTML が output されます。

```
<p>とある魔術の<ruby>禁書目録<rt>インデックス</rt></ruby></p>
```

これで Markdown で自由に HTML のルビを挿入できるようになりました！

まとめ

Vivliostyle と Remark は実際には無関係なライブラリです。しかし、Markdown の変換環境の構築によって Vivliostyle が読み込む HTML を素早く出力できるようになれば、そのまま文書完成までの時間が短縮できる大きなメリットがあります。Markdown は書きやすいたくだけなくルールが少ない点も特徴であり、自分で文法を追加する余地が多くあります。ぜひ目的に適した拡張を追加して、快適な執筆環境を作ってみてください。

Vivliostyle で縦組シナリオを組版する

小形克宏（電腦マヴォ合同会社） [1]

はじめに

私は小さなエージェント会社に勤務している。仕事はマンガの編集・制作だ。ちょっと前に、あるオリジナル・シナリオをマンガ化するクラウドファンディングをおこなった [2]。

幸い目標額を達成できたので、今度はそのシナリオを読みやすく組み、リターン品として支援者にお届けすることになった。そこで、この機会に以前から興味のあった Vivliostyle に挑戦することを思い立った。まず、出発点である Word のオリジナル原稿を示しておこう（図1）。

廃止A 「メディアが取り上げる事件の多くは、不審者による殺人事件などです。だから遺族の恨みや報復感情を世に訴えやすい」
 存置B「それはメディアの問題です」
 存置A「争点をずらしているよ」
 片桐「遺族の気持ちを死刑の理由にするならば、もしも天涯孤獨の人が殺されたとき、遺族はいいからその罪は軽くなります。それでいいのですか」
 また黙り込む存置派。ああなるほど、というようにうなずく何人かの見学者。
 ○サブ。
 プロデューサー「片桐弁護士の話は説得力があるな」
 ディレクター「現場の人ですから」

図1: オリジナルのWordによる原稿

それでも、仕事で InDesign は使ってきただものの、HTML やら CSS は耳学問のみ。具体的に言うと「class 定義、@page ルールってなんでしたっけ」というレベル。実際、私がまず最初にしたことは、図書館に行って、HTML や CSS の入門書を読むことだった。まあ、そこから始まる過程全てを再現しても、読者には退屈なだけだろう。そこで、本稿では組版的に興味深いトピックに絞って追うことにする。

[1] <https://www.mavo.co.jp/>

[2] 「森達也原作マンガ『死刑（仮）』第1話45ページを完成させたい」 <https://camp-fire.jp/projects/view/153218>

シナリオ組版ってなんだ!?

私自身、映画や演劇を見るのは好きだが、作る側での関わりはない。だからシナリオについてもほとんど知識がない。まずは組む対象を知らなければ。そこでネット上をあれこれ画像検索したところ、以下のような組み方の特徴があることがわかった。

1. 役名から一定の間隔を空け、同じ位置でセリフが始まる → セリフ部分の行頭が揃う
2. 役名の文字数は作品により幅がある → あらかじめ文字数を勘案してセリフの開始位置を決めるうまく揃わない
3. ト書きはフォント、インデントや囲みなどにより、役名 + セリフから強く区別される

衆目の一致するところ、組版的に最も面倒臭いのは 1 と 2 だろう。図にすると以下のようになる（図2）。

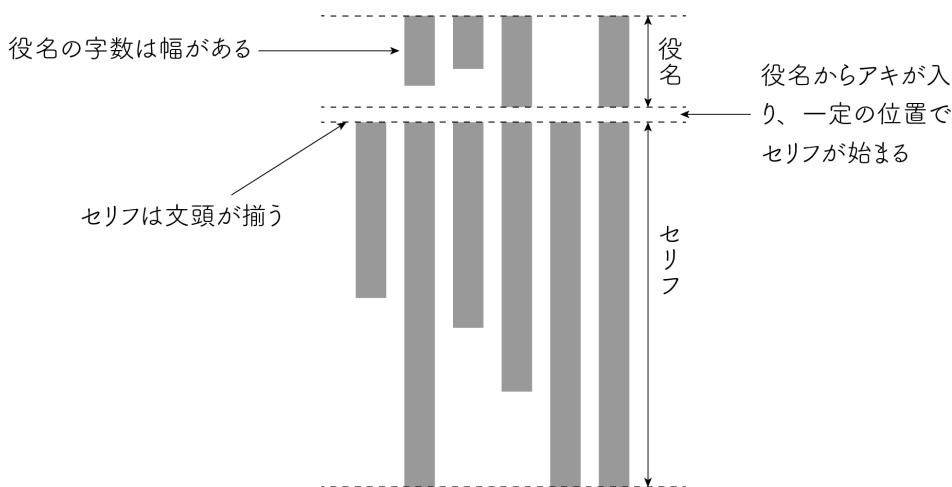


図2: シナリオ組版の特徴

じっと見るうちにハタと気づいた。ア、ナルホドネ、要するにこれは「突き出しインデント + タブ区切り」なのだ。

突き出しインデントとはなにか。たぶん最も身近で見かける例は、この原稿でも数行前で使っている箇条書きだ。ご覧いただくと分かるとおり、折り返した2行目の行頭がインデントされ、1行目の文章部分の最初と揃っている。結果として文頭の数字部分が文章部分から突き出ることになる。シナリオの場合には、この突き出た数字部分が何文字かに増え、役名になるわけである。

しかしこれだけでは実現できない。「一定の位置に揃える」ために役名とセリフをタブで区切ることが重要だ。タブとは任意の位置に文字を揃える機能のこと。文字数が0の場合において、タブの入力位置（つまり行頭）からその次の文字までの距離を「タブ幅」という。たとえばタブ幅が6emとすれば、タブより前に6em未満=1~5emの文字があった場合でも、タブの次の文字は同じ位置に揃ってくれる（図3）。シナリオの場合、役名が1~5文字に収まれば、セリフ部分の開始位置がきれいに揃うわけだ。

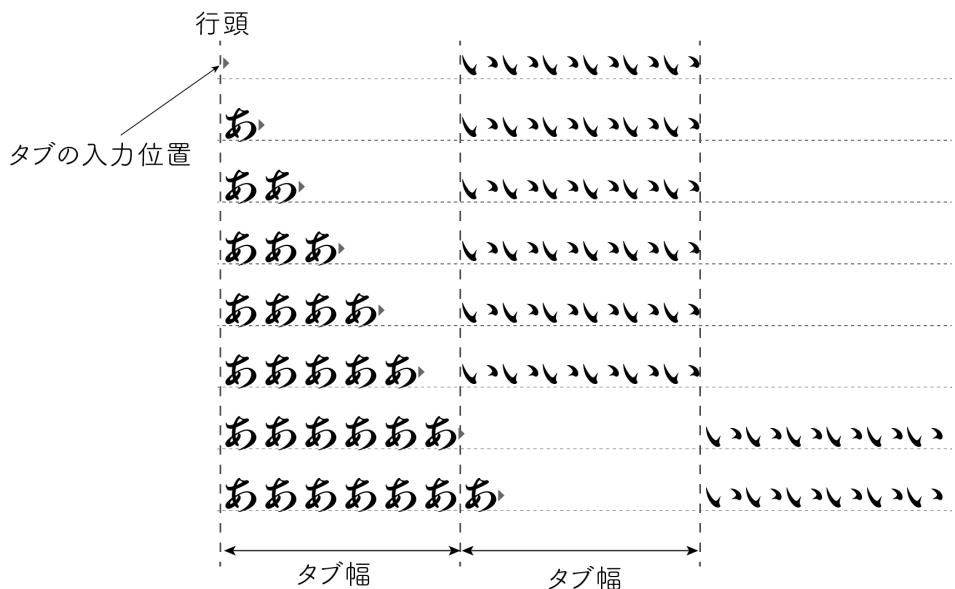


図3: タブ幅6emの場合のふるまい。タブの前の文字数が6em未満の場合は、タブの次の文字の開始位置がきれいに揃う

InDesignで仮組みしてみる

こうしてシナリオ組版のアルゴリズムは分かった。しかしこでいきなり不馴れた HTML + CSS に挑戦するのは、私には敷居が高すぎる。そこで、まず手慣れた InDesign でテスト組版をすることにした。いろいろ試した結果、あらかじめ役名とセリフをタブで区切っておいたテキストに対し、次のような段落スタイルを適用することで実現できることが分かった（図4）。

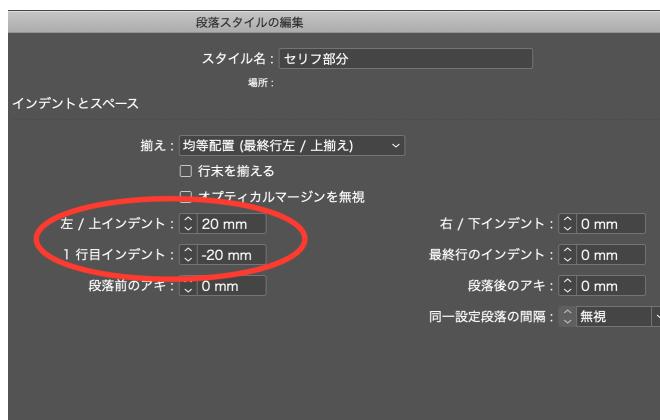


図4: シナリオ組版を実現するInDesignの段落スタイル設定

ここでポイントとなるのは、左／上インデントと1行目インデントの数値だ。設定パネルを見ると、同じ数の正の値が前者に、負の値が後者に与えられていることが分かるだろう。これにより、①段落全体を特定の数値でインデント（左/上インデント）させた上で、②1行目だけが段落全体のインデントと同じ量だけ突き出すことになる。以下はこの設定の適用結果だ（図5）。

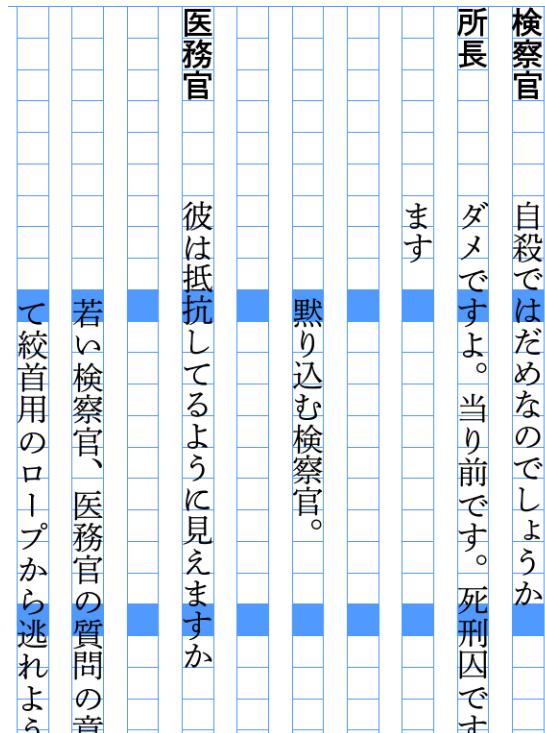


図5: 前図の設定を適用した結果

CSSで実現するための試行錯誤

ここまでをまとめると、前項で解説した「役名とセリフをタブで区切る」「段落全体をインデントさせ、1行目のみインデントと同じ量だけ突き出す」の2つを、HTML + CSS で実現すればよいということだ。やったね、これで楽勝じゃん。

もちろん、世の中それほど甘くない。とはいっても突き出しインデントに限れば、ありふれた表現だけに CSS でも簡単に実現できる。

```
p {  
    margin-left: 6em; /* ブロック全体を 6em インデント */  
    text-indent: -6em; /* 先頭行のみ 6em 突き出し */  
}
```

つまりブロック全体の左／上の余白サイズを `margin` プロパティで、ブロック 1 行目のインデントを `text-indent` プロパティで指定する。

問題はタブ区切りだ。役名とセリフをタブ区切りしたテキストを HTML にしてブラウザで表示させると以下のようになってしまう（図6）。なお、この段階では縦組は後回しにしている。

存置B それはメディアの問題です。

存置A 争点をずらしているよ。

片桐 遺族の気持ちを死刑の理由にするならば、もしも天涯孤独の人が殺されたとき、遺族はいないからその罪は軽くなります。それでいいのですか。

図6: タブを使用したHTMLの表示結果

役名とセリフの区切りが、スペース (U+0020) 1 つに置き換わってしまった。これでは狭苦しくていけないし、そもそもセリフの開始位置が揃っておらず、セリフ 2 行目もインデントされていない。では、HTML でタブを表現するにはどうすればよいのか。

調べると `white-space` というプロパティがあることが分かった。これを使えばソース中のタブ等をそのまま表示してくれるらしい。そこで HTML の `p` 要素に対し、以下のように指定した。

```
<p style="white-space: pre;">
<strong>存置B</strong>それはメディアの問題です。
</p>
<p style="white-space: pre;">
<strong>存置A</strong>争点をずらしているよ。
</p>
<p style="white-space: pre;">
<strong>片桐</strong>遺族の気持ちを死刑の理由にするならば、もしも天涯孤独の人...
</p>
```

これをブラウザで表示させたのが以下のものだ（図7）。

存置B それはメディアの問題です。

存置A 爭点をずらしているよ。

片桐 遺族の気持ちを死刑の理由にするならば、もしも天涯孤独の人が殺されたとき、遺

図7: white-spaceプロパティを使ったHTMLの表示結果。行が折り返されなくなってしまった

3行目を見ていたらと分かるとおり、行が折り返されていない。white-space プロパティはコードを表示するためのものなので、これは当然の振る舞いと言える。また、タブの方も 1行目と 2行目は役名の文字の長さがタブ幅を超てしまったらしく、セリフの開始位置が3行目と揃っていない。

その後、white-space: pre のかわりに white-space: pre-wrap の指定をすると、タブ等の空白文字をそのまま表示し、行の折り返しもしてくれることを知った。とはいえ、タブ幅を指定できるわけではないので、図7と同様の問題は残る。やっぱりダメだこりや。

ここでハタと気づいた。そもそもなぜHTMLでは強制的にタブがスペースに置き換えられたのか。考えてみれば、同じようにタブを入力した場合でも、OS、アプリケーションなど環境によってタブ幅は変わる（設定により変えられる）。HTML + CSS では多様なデバイスで同じ表示結果が求められることを考えれば、むしろタブを機能させないのは合理的な判断なのだ。つまり、私のアプローチがあさっての方向に暴走していたわけだ。となると新たにタブを使わないCSSでの表現方法を見つけないといけない……。

村上さんのアドバイス降臨！

困ったな、なんとかならないっすかね、とslackで泣きついたら、Vivliostyleの開発者・村上真雄さんから、速攻で以下のようなアドバイスをいただいた（以下、村上CSSと略）。

- style=“white-space: pre;”は削除
- 役名（ タグ）ではじまるセリフの段落を他の段落と区別できるように class をつける
- CSSでセリフの段落に対し、以下のように指定

```

p.line {
    margin-inline-start: 6em;
    margin-block-start: 0;
    margin-block-end: 0;
}
p.line > strong:first-child {
    margin-inline-start: -6em;
    min-inline-size: 6em;
    display: inline-block;
}

```

見ると分かるとおり、上記では突き出しインデント + タブ区切りを実現するだけでなく、最新のプロパティを使うことで、同一の指定で縦組にも横組にも使えるよう工夫されている。すばらしい！いやあ、コミュニティって神。みんなも Vivliostyle の slack に参加するといいよ！ [3]

以下、この村上 CSS の詳細について、その後調べたことを元に解説していこう。

論理プロパティについて

まず、村上 CSS で使われている以下のプロパティは、新しい概念にもとづく「論理プロパティ」と呼ばれるものだ。

- margin-block-start
- margin-block-end
- margin-inline-start
- min-inline-size

ここでいう「論理」（logical）とは、従来のような物理的（physical）ではなく論理的な方向・サイズを扱うことに由来する。

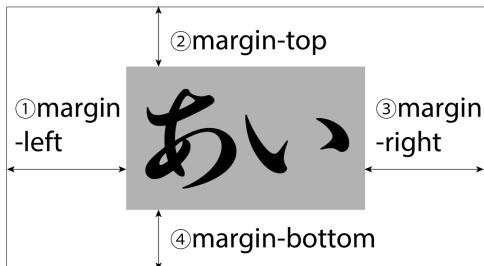
たとえば、これまで使われてきた物理プロパティ `margin-top`、`margin-right`、`margin-bottom`、`margin-left` に対応する論理プロパティは `margin-block-start`、`margin-inline-end`、`margin-block-end`、`margin-inline-start` だ。この例から分かるように、今まででは `top`、`right`、`bottom`、`left` という物理的（絶対的）な方向を使って指定してきたが、論理プロパティでは `start` や `end` という論理的（相対的）な方向を使っている。

[3] Vivliostyle 公式サイトの「コミュニティ」ページ <https://vivliostyle.org/ja/community/> から参加可能。

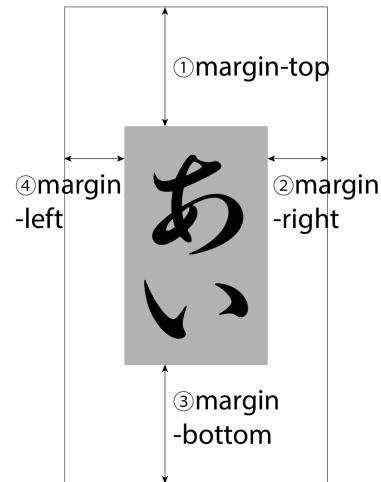
これを使う最大のメリットは特定の組み方向に依存しないで記述できることだ。今まで横組を前提に書かれた CSS を縦組に切り換えるとすれば、方向やサイズに関連するプロパティの値を一つ一つ書き換えなければならなかつた。しかし論理プロパティを使えば、writing-mode プロパティで組み方向の指定を変更するだけですむ（図8）
[4]。

[4] より詳細は以下の記事を参照。「[CSS]知っておくと便利な論理プロパティ、ボックスモデルにおける古い方法とこれからの方針」<https://coliss.com/articles/build-websites/operation/css/new-css-logical-properties.html>

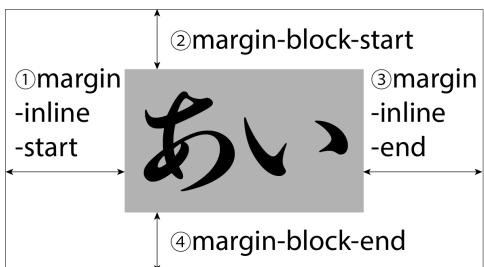
■物理プロパティ／横組の場合



■物理プロパティ／縦組の場合



■論理プロパティ／横組の場合



■論理プロパティ／縦組の場合

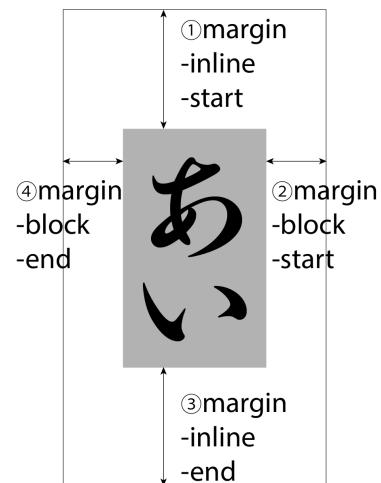


図8: 物理プロパティで横組から縦組に切り換えようとする場合、横組のプロパティの値を、図中の同じ丸付き数字の縦組プロパティに書き写さなければならない。対して論理プロパティでは丸付き数字とプロパティが縦組／横組で変わらないことから分か るように、こうした複雑な修整は不要だ

論理プロパティを規定する仕様が“CSS Logical Properties and Values Level 1”^[5] だ。
writing-mode プロパティを規定する“CSS Writing Modes Level 3”^[6] を実装するには論理
プロパティが必要になることもあり、まだワーキングドラフトの段階であるにも関わらず、
プラウザへの実装は順調に進んでいる^[7]。

突き出しインデントについて

その論理プロパティを使って、どうやって突き出しインデントを実現するのか。あらためて InDesign における突き出しインデントの実現方法を振り返ると、以下の通りだった。

1. 段落全体を特定の数値でインデント（左／上インデント）させる
2. 1行目のみ段落全体のインデントと同じ量だけ突き出す

そして物理プロパティでは、margin でブロック全体を正の値で、text-indent で 1 行目だけ同じ数の負の値を指定するのだった。

```
p {
    margin-left: 6em; /* ブロック全体を 6em インデント */
    text-indent: -6em; /* 先頭行のみ 6em 突き出し */
}
```

では、論理プロパティではどう実現するか。まずト書き等と区別するため、あらかじめ HTML で指定したい箇所を class で定義しておく（ここでは class 名を “line” とした。もちろんこれは物理プロパティでも必要）。

```
<p class="line">
<strong>存置B</strong>それはメディアの問題です。
</p>
<p class="line">
<strong>存置A</strong>争点をすらしているよ。
</p>
<p class="line">
<strong>片桐</strong>遺族の気持ちを死刑の理由にするならば、もしも天涯孤独の人...
</p>
```

[5] <https://drafts.csswg.org/css-logical/>

[6] <https://www.w3.org/TR/css-writing-modes-3/>

[7] <https://wpt.fyi/results/css/css-logical?label=experimental&label=master&aligned>

そのうえで、CSS で “line” ブロック全体の左／上余白を 6em に指定する。

```
p.line {  
    margin-inline-start: 6em;  
    margin-block-start: 0;  
    margin-block-end: 0;  
}
```

ここまでをブラウザで表示させたのが以下のもの（図9）。しかし、これではまだ半分、突き出しになっていない。

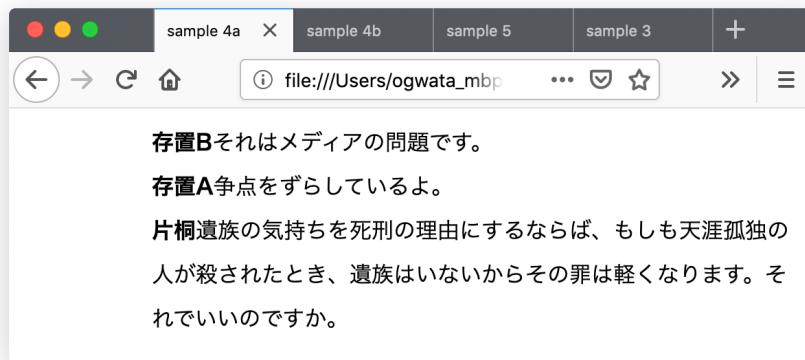


図9: ブロック全体を、左／上方向の余白として6emを指定する

そこで上記に加えて、“line” ブロックの 1 行目だけに負の値、-6em を指定したいわけだが、この特定のブロックの 1 行目「だけ」を特定するのに技が必要だ。

あらためて HTML をじっと見ると、役名が必ず子要素 `strong` で強調されている。これをキーにすれば特定できそうだ。子要素だけを指定するには親要素と子要素を `>` でつなげばよい。

いや待ってほしい、`strong` は強調としてもよく使われるから、役名だけに使っていると決め打ちすると痛い目に遭う。もう一段、正確性を増すための絞り込みが必要だ。そこで疑似要素 `first-child` を使うことで、「`strong` 要素が最初の要素である場合」と限定する。そのうえで、そこに負の値、-6em を指定する。以下のように。

```
p.line > strong:first-child {
    margin-inline-start: -6em;
}
```

これをブラウザで表示させたのが以下のもの（図10）。ぶじに突き出しインデントになつた。

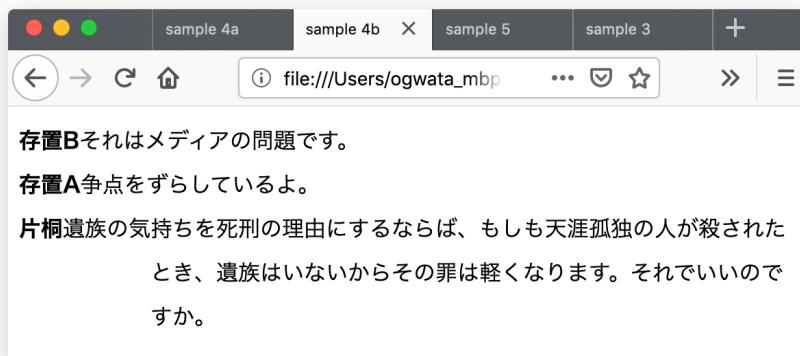


図10: 1行目だけ6em分突き出すように指定した

セリフ部分の開始位置を揃える

あとはセリフ部分の開始位置を揃えればよい。再び HTML を観察すると、指定しようとする “line” は例外なく 2 つの要素で成り立っていることが分かる。つまり、行頭は `strong` 要素で指定された役名で始まり、それにセリフが続いている。ということは、行頭の要素（役名部分）のために「一定の領域」を確保し、それにセリフの領域が続くと考えればよい。上記 CSS の `margin-inline-start: -6em;` に続けて、以下のように指定する。

```
min-inline-size: 6em;
display: inline-block;
```

この `min-inline-size` プロパティは任意の要素における最小サイズ（横組の場合は水平方向、縦組の場合は垂直方向）を指定するものだ。この場合 CSS の前行まで「“line”において最初の要素として `strong` が使われているもの」を指定しているので、ここでもそれが継承される。言い換えれば、“line”に役名が存在する場合、役名の領域の領域の長さとして 6em 確保される。そして、それに隣接してセリフの領域が始まるようになるわけだ。

別の言い方をすると、役名が 5em 以内であれば、セリフの領域は必ず行頭から 6em の位置で始まる。これは前の方で述べた「タブ幅」の振る舞いそのものであることに注意してほしい。

加えて `display` プロパティを使って、縦組でも横組でも 2 つの要素が必ず直列に配置されるよう指定する。この場合の値は、組み方向に関わらず `margin` と `padding` のサイズが有効になる `inline-block` が適当だろう。ここまで村上 CSS を、ブラウザで表示させたのが以下のものだ（図11）。

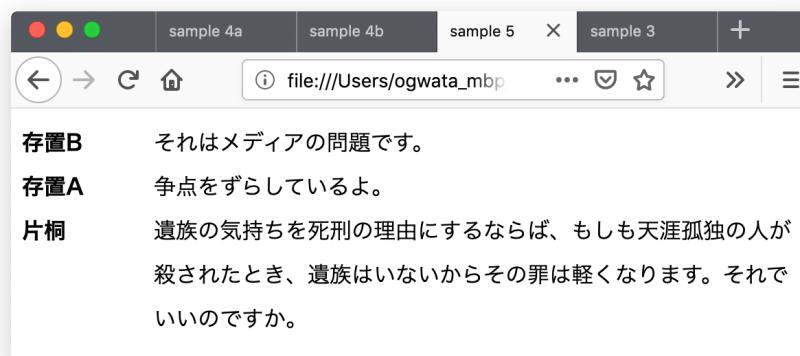


図11: 役名から一定の間隔をとって、セリフ部分の開始位置がきれいに揃った

仕上げ：縦組、2段組み、縦中横の指定

さて、これで冒頭「シナリオ組版ってなんだ!？」の項で示した 3 大特徴のうち、2 つまで達成できることになる。残りはト書き部分の区別化だが、これはト書き用の `class` を定義したうえで、論理プロパティ `margin-block-start`、`margin-inline-end`、`margin-block-end`、`margin-inline-start` をあてればよい。

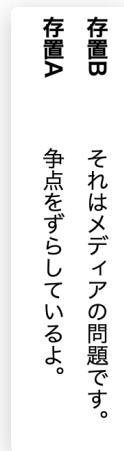


図12: 縦組なのにアルファベットが
横転している

あとは縦組と2段組み、版面の指定だ。これもゼロからやるとなれば大変だが、Vivliostyle のサンプルページを使えば、ぐっと楽ができる。公開されているうち、同じ縦組／2段組みの例である「雑誌レイアウト『白馬岳』」の“style.css”^[8] を雛型として、合わない部分だけ修整していくことにする。となると、HTML も調整が必要だろう。

大筋としては `class` 名を HTML と CSS とで合わせ、不要な記述があればばっさり削除していく。前項まで検討してきた役名 + セリフ部分のプロパティも忘れずに書き加えておこう。

他に縦組特有の作業として、「縦中横」の指定が必要だ。これを指定しないと以下のようにアルファベットや数字だけが横転してしまう（図12）。

[8] https://github.com/vivliostyle/vivliostyle_doc/blob/gh-pages/samples/shirouma/style.css

まず、HTML で縦中横にする文字を `span` 要素で囲み、`class` 指定をしておく。『白馬岳』の“style.css”では `class` 名を `text-combine` としているので、これを生かして HTML で以下のように指定する。

```
<p class="line">
  <strong>存置<span class="text-combine">B</span></strong>
  それはメディアの問題です。
</p>
<p class="line">
  <strong>存置<span class="text-combine">A</span></strong>
  争点をずらしているよ。
</p>
```

『白馬岳』の“style.css”では、以下のように縦中横を実現する `text-combine-upright` プロパティが指定されている。

```
.text-combine {
  text-combine-upright: all;
}
```

ちなみにこの“style.css”は書かれたのが少し古いうで、ベンダープレフィックスがついた複数の縦中横用プロパティも記述されている。しかし、現在の Vivliostyle はブラウザによって必要なプレフィックスを自動補完してくれるの で、`text-combine-upright` プロパティ以外は削除した。さあ、ここまでをブラウザで表示させてみよう（図13）。

うまくいったようだ。あとは本文や見出しのフォントを調整して出来あがりだ（図14）。

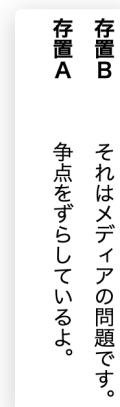


図13: アルファベットが正立した。
これが「縦中横」だ

片桐
顔を見合わせる存置派三人。

あなたたちは遺族の心情を死刑存置の理由にしながら、死刑によって新たな遺族が生まれることをどう考えているのですか。

えーとね、片桐さん、あなたね、罪人と一般人の家族を一緒ににするのですか。

(毅然と) 当事者ではないという意味では同じです。

(勢いづいて) 罪人であつてもその家族に罪はない。

そもそも実際に起きる殺人事件の8割は身内の犯行です。つまり加害者家族と被害者遺族は簡単には分けられない。

(困惑したように) …だから?

メディアが取り上げる事件の多くは、不審者による殺人事件などです。だから遺族の恨みや報復感情を世に訴えやすい。

それはメディアの問題です。

争点をすらしているよ。

遺族の気持ちを死刑の理由にするならば、もしも天涯孤独の人が殺されたとき、遺族はいないからその罪は軽くなります。それでいいのですか。

また黙り込む存置派。ああなるほど、というようになんずく何人かの見学者。

○サブ

プロデューサー 片桐弁護士の話は説得力があるな。

ディレクター 現場の人ですから。

プロデューサー だからこそ、彼に対する反発も強い。

A D そうなんですか。

ディレクター ネットで検索してみな。鬼畜弁護士とかこいつを死刑にしたいとか、そんな書き込みばかりだ。

A D 言われてスマホで片桐の名前で検索するA D。うわあこりやひでえや、とつぶやく。

誰かこいつの家族を殺してくれ、みたいな書き込みもありますよ。

ディレクター それは、…通報ぎりぎりだな。

プロデューサー おい。このまま存置が劣勢だと、番組としてまずいぞ。

ディレクター、あわててフロアディレクターに、「存置派、誰か発言しないのか」と檄を飛ばす。フロアディレクターからの指示のカンベを見たMCが、「さて、存置派のお三方、反論はないですか」と声をあげる。

図14: 段組みを指定。上の段と下の段とできれいに行が揃っている。これで完成だ

おわりに

以上、Vivliostyle における縦組シナリオ組版を報告した。数年前から、出版業界では InDesign の独占がつづいている。この無競争状態が健全とは言えないことは誰の目にも明らかだろう。日本語組版が未来へ進歩をつづけるためにも、強力なオルタナティブの登場が待望されている。Vivliostyle がその答えになるかどうか、それは私達ユーザのサポートにかかっていると言えるだろう。

EPUB ファイルから Vivliostyle で PDF を作りたい！

田嶋 淳 (@JunTajima) [1]

印刷 DTP データからの EPUB 作成の仕事を長年やっていて、ずっと解決できないでいた悩みがあった。それは、「EPUB を PDF に出力すること」である。Web 方面の人はなぜそんなことが必要なのか不思議に思うかもしれない。それは「出版社に内容を確認してもらうため」に必要になるのだ（「検収」と呼ぶ）。もちろんわれわれ制作者は、できる限り元データの情報を失わない形で電子化をするのだけど、だからといってチェックもせずにそのまま販売はできない。そしてチェックのためには、できるだけ元の本の版面に近い形で EPUB を表示させ、それを元の本と見比べる必要が出てくる。出版社によっては外部の校正者に依頼するケースもあるから、可能ならプリントできる PDF 形式の変換データも渡すことが必要になってくるのだ。ところが長いことこれが随分難しい話だった。商業用 EPUB ビューアには例外なく「印刷の機能がない」からだ。理由はまあ明確で、もし購入した電子本を印刷の機能を使って PDF にできてしまえば、それはすなわち海賊版の作成補助になってしまうからだ。これはどこの出版社も納得しないだろうからまあ仕方ない。ただし、前述の理由で製作段階では紙に出力できる PDF を作る必要はあるから、現場ではスクリプト処理で連続スクリーンショットを取り、それを PDF 化するというような力技が必要になっていた。画像の塊だからこの PDF はファイルサイズが恐ろしくデカい。減色処理をしても平気で 400～500MB にもなる。これだと作成そのものにも時間がかかるし、データを送付するのも大変だ。どうにかできないか、というのが長年の悩みだった。そこで Vivliostyle である。EPUB の表示に対応済みと村上さんからお聞きしたとき、これならもしかして行けるのではないかとすぐに思った。なにしろ本来の素性が CSS 組版エンジンなので、組版表示の能力はかなり期待はできる。そこが貧弱だと検収用には使えないのだ。ということで村上さんにご指導をいただき、早速試してみた。

やりたいこと

最初に、やりたいことを箇条書きでまとめておく。

[1] <http://densyodamasii.com/>

1. EPUB を CSS の標準的な表現に沿った形で PDF 化したい
2. 元の本の版面にできるだけ合わせたい
3. Web にアップロードせずローカルで処理したい
4. 校正指示のためにページ番号は入れたい
5. 元 EPUB の CSS はできれば触りたくない
6. Mac 標準のツール群でどうにかしたい

個々の項目に関して説明すると、まず 1 については、表示確認目的である以上、標準的な表現である方が望ましい。現在いくつかの EPUB ビューアは日本語を読みやすくする目的で独自に表現の拡張を行なっているが、そういうものはリファレンス的な用途には不向きだ。2 については実際の表示確認のワークフローが底本との照らし合わせである以上必須になる。特に一行文字数は確実に合わせたい。これが合っていないと、校正におそろしく手間がかかる。これは実際にやってみればすぐわかる。可能ならば2つの版面を並べて「絵として比較できる」状態が望ましいのだ。括弧類や句読点のツメ処理などの関係でどのみち全く同じ組版にはならないけれど、見比べる以上共通点は少しでも多い方がよい。3 は販売前の商用 EPUB の制作に利用する以上は当然だ。データそのものを外部から見られる場所にアップロードするわけにはいかない。もし漏洩したら大問題になる。4 はまあできればよいけれど、修正箇所の指示で今出版社の担当者が大いに悩んでいそうなのは日々感じているので、入れられるなら入れてあげたい。5 は4を踏まえた上で、その CSS 指定のために元の EPUB のファイル群は可能なら触りたくない。触ればヒューマンエラーも起きるし、手間もかかる。で、その上で、できるなら追加でモジュール群などをインストールしてマシンを環境構築する手間は避けられるなら避けたい。これは会社の複数台のマシンをメンテナンスする労力からくる要望だ。

Vivliostyle で EPUB をローカル表示させ PDF 化する手順

ということを踏まえて、実際に Vivliostyle で EPUB をローカル表示させ、PDF にするまでをやってみた。手順は以下の通り。

1. Web サーバをローカルで立てる

Vivliostyle は本来サーバに置いて動かすようなソフトウェアなので、それをローカルで使うにはまず自分のマシン内にサーバを立てる必要がある。もちろんこれはローカルで完結していて外部からは見えない状態なのだが、いわば自分のマシン内に仮想的に立てたサーバ上のファイルをローカルのブラウザで見られる状態にする必要があるのだ。

Vivliostyle の公式説明では Node.js を使うことになっていたのだが、Node.js 自体のインストールを個々のマシンで行わなければならない上に、インストールで少々手こずってしまい、結果的に村上さんにアドバイスをいただいて Python のコマンドを使って解決した。Python ならば Mac に標準で入っているのでその方がむろん望ましい。

手順としては単純で、Mac に標準で入っている「ターミナル」上で「`python -m SimpleHTTPServer 8000`」のコマンドを打ち込むだけだ。これだけでポート 8000 を対象としてローカルサーバが起動する。次の画面のようになっていれば OK。

```
Last login: Mon Aug 26 17:05:51 on ttys000
[smdsmacmini:~ smds_macmini]$ python -m SimpleHTTPServer 8000
Serving HTTP on 0.0.0.0 port 8000 ...
```

図1: ローカルサーバが起動

2. Vivliostyle Viewerをローカルで起動

さて、無事ローカルサーバが立ち上がったら、その上でローカルのフォルダ内にある Vivliostyle Viewer を動かしてやる。例えばダウンロードしてきた `vivliostyle-js-2019.1.105` のフォルダが Mac のデスクトップ上の `vivliostyle_test` フォルダに置かれているなら、URL 指定は以下のようになる。

```
http://localhost:8000/Desktop/vivliostyle_test/vivliostyle-js-2019.1.105/viewer/vivliostyle-viewer.html
```

これをブラウザのアドレスバーに入力して無事に Vivliostyle Viewer の画面が表示されればひとまずは成功。

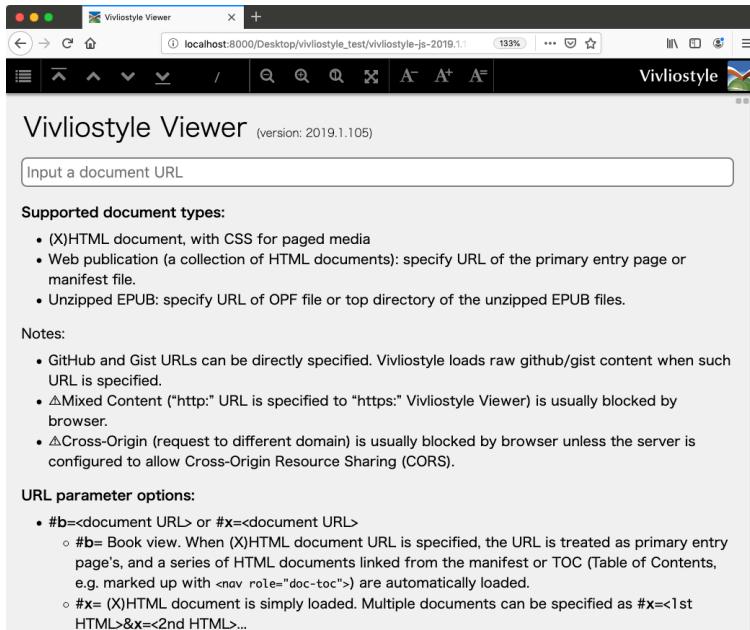


図2: Vivliostyle Viewerが表示された

デスクトップにフォルダを置きたくないというのであれば、「Documents」がMacのログインユーザの「書類」フォルダに当たるようなのでそちらでもいいだろう。今どんなフォルダがサーバ上から見えているのかを知りたければ、「localhost:8000」とだけアドレスバーに入れてやればリストが出てくるはず。

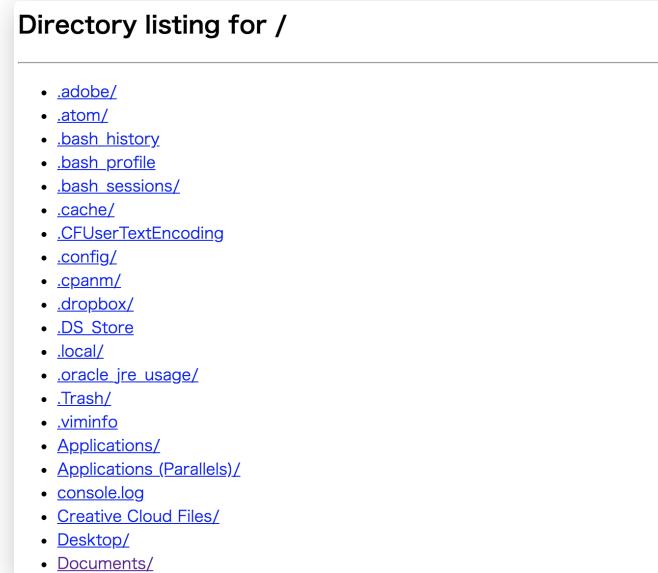


図3: フォルダ一覧

3. テストファイルをVivliostyleで表示

さてそれではいよいよ EPUB ファイルを Vivliostyle 上で表示してみる。手順 2 の Vivliostyle Viewer の URL の後に「**#b=**」を書き、「**http://localhost:8000**」の後に表示ファイルのパスを指定してやればよい。EPUB表示の場合は解凍したEPUBフォルダのパスを指定するか、あるいはEPUB内.opfファイルのパスを指定してやればOKだ。

例えばデスクトップの `vivliostyle_test` フォルダ内に置いた `testepubfolder` を表示させたいのなら

```
http://localhost:8000/Desktop/vivliostyle_test/vivliostyle-js-2019.1.105/viewer/vivliostyle-viewer.html#b=http://localhost:8000/Desktop/vivliostyle_test/testepubfolder/item/standard.opf
```

のような記述になる。これで無事に EPUB が表示されれば成功だ。

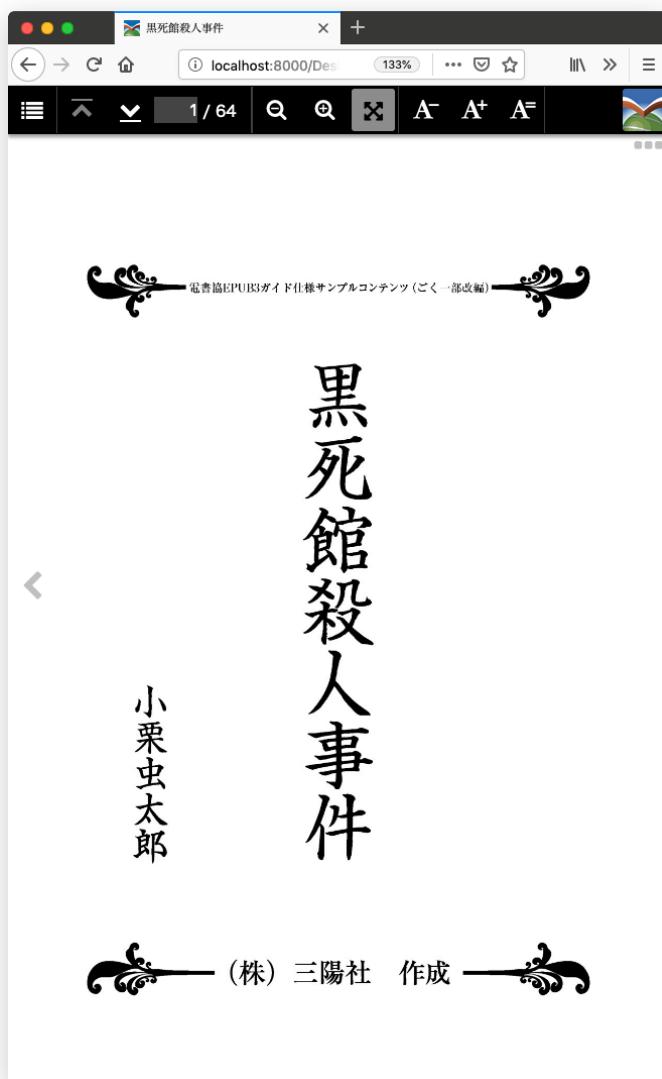


図4: EPUBの表示に成功

4. CSSを追記してページ番号を表示

では、これに加えてページ番号を表示させてみよう。Vivliostyle には表示の際に CSS を追加して表示させる機能があるのでそれを使う。ブラウザ上の Vivliostyle Viewer の環境設定で「Override Document Style Sheets」のチェックボックスをチェックし、「CSS Details」に CSS を書き込んでやればよい。

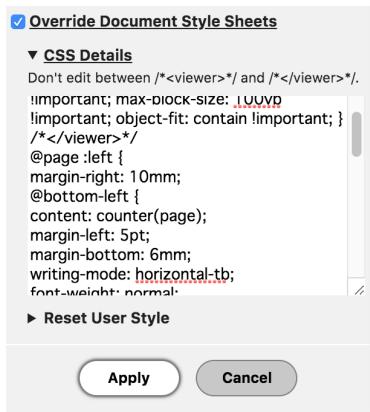


図5: CSSを追記

今回はページ番号の表示のために以下の内容を追記した。

```

@page :left {
margin-right: 10mm;
@bottom-left {
content: counter(page);
margin-left: 5pt;
margin-bottom: 6mm;
writing-mode: horizontal-tb;
font-weight: normal;
font-family: serif-ja, serif;
}
}
@page :right {
margin-left: 10mm;
@bottom-right {
content: counter(page);
margin-right: 5pt;
margin-bottom: 6mm;
writing-mode: horizontal-tb;
}
}

```

```

    font-weight: normal;
    font-family: serif-ja, serif;
}
}

@page :first {
  @bottom-left {
    content: "";
  }
  @top-left {
    content: "";
  }
}

```

最後の「`@page :first`」のブロックは1ページ目は書影なので番号を入れたくないから消しているだけなので、気にしないなら入れなくてもよいだろう。

同時に、環境設定メニューでチェックボックスをいくつかチェックしてやる。全ページレンダリング読み込み指定の「**Render All Pages**」と、画像表示最適化の「**Set image max-size to fit page**」および「**Keep aspect ratio**」だ。最終的にPDF化するのが目的なので全ページ出してくれないと困るし、画像表示最適化のチェックを入れておかないと画像がページ内にきちんと収まってくれなかったりする。

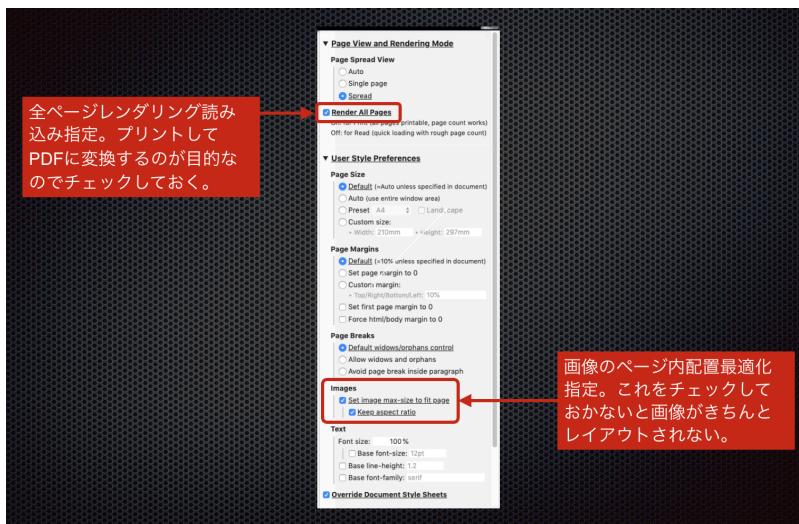


図6: 追加でチェックボックスをチェック

最後に「Apply」をクリックして設定を適用してやり、無事にページ番号が表示されていれば OK。

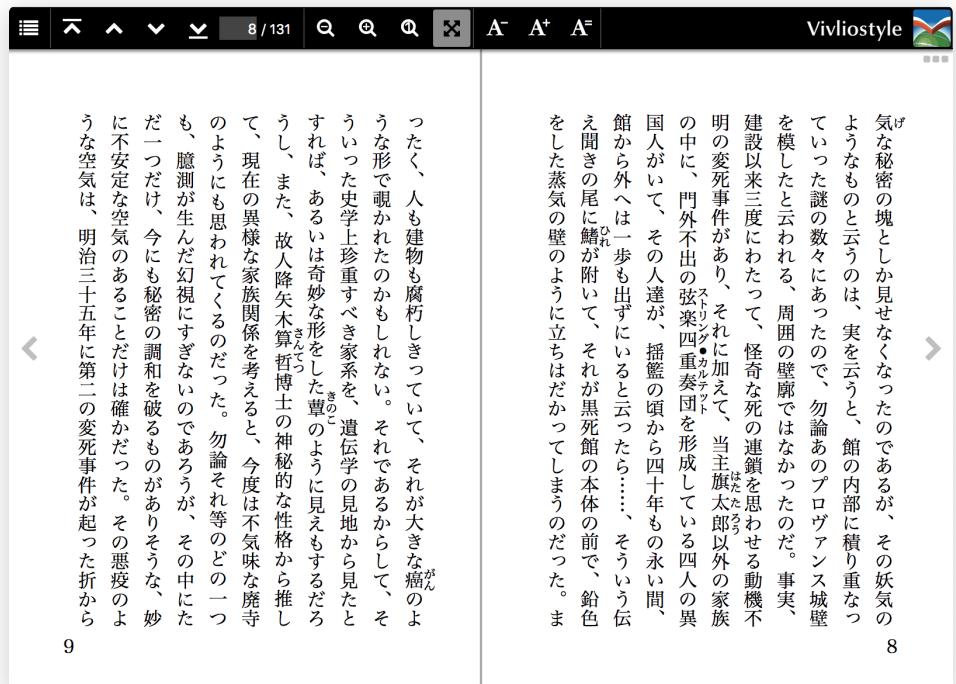


図7: ページ番号が表示された

5. プリントしてPDFを出力

ここまでできれば、あとはブラウザの画面を元の本と見比べて 1 行文字数などを調整してやり、プリントメニューから PDF としてプリントを実行するだけだ。サイズはブラウザのウインドウサイズで調整できるのでとてもラク。



図8: プリントメニューからPDFとしてプリント

おつかれさまでした！

ここまで手順をPerlで自動化してみた

さて、できるにはできたのだけれど、これを毎回仕事でやるのは相当ツライ。なのでここまで手順を Perl で自動化して楽をすることにした。

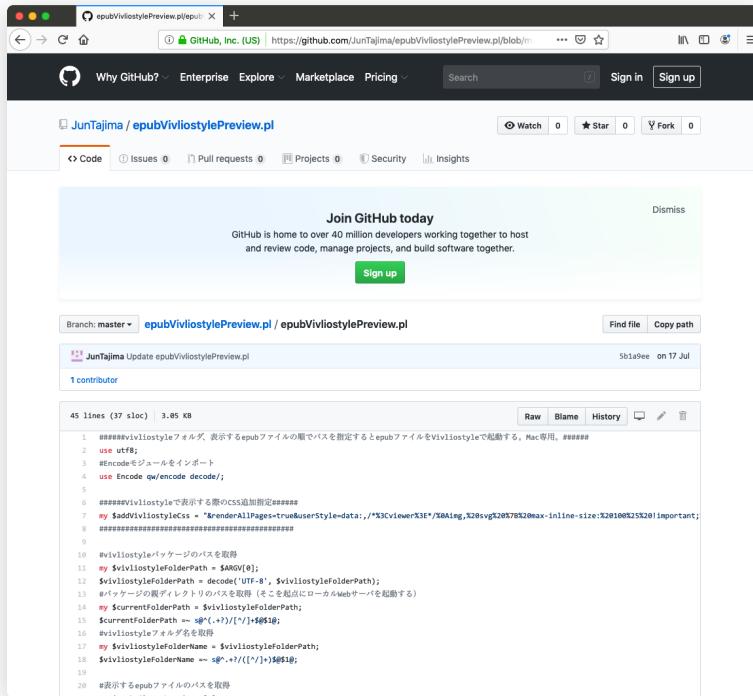


図9: Perlで自動化

まあ一応ソースコードを載せておくけれど、実際大したことはしていない。 [2]

[2] <https://github.com/JunTajima/epubVivliostylePreview.pl/blob/master/epubVivliostylePreview.pl>

```
#####vivliostyleフォルダ、表示するepubファイルの順でパスを指定すると epub ファ...
use utf8;
#Encodeモジュールをインポート
use Encode qw/encode decode/;

#####Vivliostyleで表示する際のCSS追加指定#####
my $addVivliostyleCss = "&renderAllPages=true&userStyle=data:/*%3Cviewer%3E*/%...
#####
#vivliostyleパッケージのパスを取得
my $vivliostyleFolderPath = $ARGV[0];
$vivliostyleFolderPath = decode('UTF-8', $vivliostyleFolderPath);
#パッケージの親ディレクトリのパスを取得（そこを起点にローカルWebサーバを起動...
my $currentFolderPath = $vivliostyleFolderPath;
$currentFolderPath =~ s@^(.+?)/[^/]+$@$1@;
#vivliostyle フォルダ名を取得
my $vivliostyleFolderName = $vivliostyleFolderPath;
$vivliostyleFolderName =~ s@^.+?/([^.]+)$@$1@;

#表示するepubファイルのパスを取得
my $epubFilePath = $ARGV[1];
$epubFilePath = decode('UTF-8', $epubFilePath);

#乱数代わりに日時の数字を取得して epub解凍フォルダ名決定
my $getDateTimeNowCommand = 'date "+%Y%m%d%H%M%S"';
my $datetimenow = `"$getDateTimeNowCommand`";
my $epubUnzipFoldername = "vivliostyleepubtmp_" . $datetimenow;
#テンポラリフォルダ無ければ作る
unless (-d $currentFolderPath . "/tmp") {mkdir $currentFolderPath . "/tmp"};
#テンポラリフォルダにEPUBファイルを解凍する
my $epubUnzipCommand = "unzip " . $epubFilePath . " -d " . $currentFolderPath . "/tmp";
system $epubUnzipCommand;

#ローカルWebサーバ起動処理
my $serverStartCommand = 'osascript -e \'tell application "Terminal" to do script "cd ' . $...
system $serverStartCommand;

#ディレイ処理
my $delayCommand = "osascript -e 'delay 2'";
system $delayCommand;

#Vivliostyle Viewer起動
my $openVivliostyleCmd = 'osascript -e \'tell application "Google Chrome" to open locat...
system $openVivliostyleCmd;
```

使い方はターミナルで「`$ perl スクリプトファイルのパス Vivliostyle のパッケージのパス EPUB ファイルのパス`」の順番で指定してやれば、自動でローカルサーバを起動し、Chrome の画面に Vivliostyle を使って EPUB を表示する。

注意点としては

- 展開後の EPUB フォルダではなく展開前の EPUB ファイルを指定
- Perl 内でシェルを呼んで osascript で Applescript でターミナルとか Chrome を立ち上げているので Mac 専用。EPUB の解凍とかにもシェルを使っている
- vivliostyle パッケージと同じフォルダに展開した EPUB ファイルはそのまま残るのであとで消す必要がある
- 追加モジュール等は使ってないので Mac なら割と環境を選ばず動くはず

ぐらいだろうか。

なお、職場用にはさらに Xojo で簡単な UI を付けてアプリ化した。そのあたりまでやらないと多分現場では使ってもらえない。



図10: Xojoでアプリ化

CSS で View を定義する意義

akabeko

アプリケーションや文書の View (外観) を定義する技術として見た、CSS の意義と期待について。

アプリケーション

Web を除くデスクトップやモバイル向けのアプリケーション開発では View を独自の技術と記法により定義することが多い。例として、これまで私が開発を経験したプラットフォームと View 技術をまとめる。

プラットフォーム	技術	定義・編集	保存形式
Windows	Win32/MFC	Visual Studio <u>Resource Editors</u>	独自 (プレーン テキスト)
"	Forms	Visual Studio <u>Form Designer</u>	C#
"	WPF	Visual Studio <u>XAML Designer</u>	XAML (XML)
macOS	AppKit	Xcode <u>Interface Builder</u>	Storyboard (XML)、XIB (XML)
iOS	UIKit	Xcode <u>Interface Builder</u>	Storyboard (XML)、XIB (XML)
Android	<u>View</u>	Android Studio <u>Layout Editor</u>	Layouts (XML)

こうして並べると定義・編集の方法こそ違えど、保存形式は XML 系が多い。これは 1990 ~ 2000 年代にかけて XML ブームがあったことが関係しているのかもしれない。データ構造を宣言的に定義するファイル形式として XML が好まれ、特に View は階層を持つことが多いため相性もよかった。

ならば XHTML のように装飾は CSS で定義するのかというと、そうなってはいない。いずれも独自の XML 属性により装飾している。以下は Android の例。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <android.support.design.widget.TabLayout
        android:id="@+id/tabs"
```

```

    style="@style/CustomTabLayout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/colorPrimary"
  />

<android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
  />

</LinearLayout>

```

属性の値は直値や定数の他、`@+id/pager` のように外部リソース参照も指定可能。他の XAML や Storyboard なども設計は似通っている。そのため XML タグと属性の差を埋めれば共通化できそうに思えるが、参照しているリソースなどの仕組みは大きく異なるため難しい。結局は別物として学習・運用しなければならぬ、コストがかさむ。

Web ブラウザーがそうであったようにプラットフォームベンダー間で開発環境に対する協調がおこなわればよいのだが、それは難しいだろう。一方、このような状況を踏まえて Web 由来の技術をアプリケーション開発へ転用する動きもある。以下は代表的なもの。

- Apache Cordova [1]
- React Native [2]
- Electron [3]
- PWA (Progressive Web Apps) [4]

PWA を除き、他はネイティブ機能を呼び出す仕組みを提供している。そのためパフォーマンス問題に目をつぶれば、実質的に Web 技術でクロス プラットフォーム開発可能となる。もちろん View も CSS となるため運用と学習コストも低減されるだろう。

[1] <https://cordova.apache.org/>

[2] <https://facebook.github.io/react-native/>

[3] <https://electronjs.org/>

[4] <https://developer.mozilla.org/en-US/docs/Web/>

この中でも特に React Native の設計が面白い。他は View に既存 Web ブラウザーを利用するのだが React Native はレイアウトを [Yoga Layout](#) という独自エンジンにより処理する。

Yoga は CSS の Flexbox 的なレイアウトを実現するクロスプラットフォームの C++ ライブラリー。レイアウト機能限定だがネイティブなアプリケーション開発に CSS の知見を持ち込む事例と言ってよいだろう。これを利用するクロス プラットフォーム GUI ライブラリーに [Yue](#) があり、こちらも注目株だ。

徐々にではあるが、ネイティブなアプリケーション開発の世界に CSS 的な View 定義が浸透し始めている。こうした技術により、プラットフォーム間の View 定義が共通化されてゆくことに期待したい。

文書

私の専門はソフトウェア開発なので文書の View である組版について詳しくないのだが、いま在籍している会社が DTP 事業を手掛けていることもあり、Adobe InDesign で組版の基本を学ぶ研修を受けたことがある。そこでは段組み、文字詰め、禁則処理などの初步を教わった。

研修を通して読みやすさに対する技術の一端に触れたわけだが、同時に疑問を持った。

- InDesign で処理される組版技術は CSS のように規格化されてるのだろうか？
- 規格化されているとして、組版を処理するソフトウェアはそれに準拠しているのだろうか？
- 例えば InDesign と他のソフトウェア間で組版技術の互換はあるのだろうか？

規格については

- JIS X 4051:2004 日本語文書の組版方法 | 日本規格協会 JSA Group Webdesk [5]
- 日本語組版処理の要件 (JLREQ) [6]

などがある。では組版ソフトウェアの対応状況はいかほどか。Web でいう [Can I use](#) のようなサイトは存在しないようなので Google 検索や各種公式サイトから、ざっくりと調べてみた。

[5] https://webdesk.jsa.or.jp/books/W11M0090/index/?bunshyo_id=JIS%20X%204051:2004

[6] <https://www.w3.org/TR/jlreq/ja/>

製品	JIS X 4051/JLREQ 対応
<u>Adobe InDesign</u>	<u>InDesign</u> での CJK 文字の組版に JIS X 4051 へ言及あり。JLREQ は不明。
<u>QuarkXPress</u>	公式サイト、Google 検索ともに言及を見つけられず。
<u>LaTeX</u>	公式サイト検索では見つからないが Google 検索で <u>jreq -TeX Wiki</u> や <u>PXrubrica パッケージ</u> ~美しい日本のルビ組版～「電脳世界の奥底にて」など複数の言及あり。
<u>The Vivliostyle Project</u>	「日本語組版処理の要件(JLREQ)」とは何かや〈次世代 CSS 組版〉Vivliostyle プロジェクトなど複数の言及あり。
XML 組版	複数企業が独自に手掛けているようだが、見つけた言及は富士美術印刷株式会社 XML 対応のみ。

規格はあるものの、対応状況を明確に公開している製品はないようだ。そのため同じ文字詰めであっても、一方の操作と設定が他方で有効とは限らない。製品ごとの癖を学習して成果物を検証する必要がある。せっかく規格があるのに、現状はうまく活かせていないのではないか。

関連する動きとして JLREQ と CSS 関連に注目している。

- w3c/jreq [7]
- JLREQ と CSS (1) | 電書魂 [8]
- JLREQ と CSS (2) | 電書魂 [9]

これは Web ブラウザーやその技術を転用した EPUB 上で適切な組版を再現するための試みであると同時に、組版規格を共通のソフトウェア資産として定義・検証可能にするための絶好の機会ではなかろうか。

理想としては Web における w3c/csswg-drafts や tc39/ecma262 のように規格の議論と策定が公開され、採用するソフトウェア自身が対応状況を明示することが望ましい。Web もブラウザー間の互換性が問題になり、このような運用へ至った。文書の組版もそうなれば幸いだ。

[7] <https://github.com/w3c/jreq>

[8] <http://densyodamasii.com/?p=3222>

[9] <http://densyodamasii.com/?p=3258>

CSS で View を定義できたなら

CSS によりアプリケーションや文書の統一的な View 定義が可能となった世界を想像してみる。極めて楽観的な展望だが、中には実現するものもあるかもしれない。

アプリケーション

View や GUI において機能・構造と外観の分離が進み、HTML/CSS のような関係となるだろう。機能・構造はプラットフォーム固有だが、外観は CSS として共通化できる。例えばボタン。機能・構造にあたる API は異なれど、それらは共に CSS を参照する設計となるはず。

CSS の定義・参照まわりは Web の [CSS Modules](#) や [CSS in JS](#) のように設計されるだろう。最近は [SwiftUI](#) と [Jetpack Compose](#) などプログラミング言語中に宣言的な GUI 定義をする動きがあるため、Web でいう React + CSS in JS のようになる可能性も考えられる。

開発体験としては Web における CSS のエコシステムが再現されるだろう。View の共有が容易となるため、[Material Design](#) や諸々の CSS フレームワークがネイティブアプリケーションにやってくる。これらを利用せずに自作するとしても、Web フロントエンドの知見とツールを転用できるから開発コストは大きく低減されるはず。

Web 以外でも View のテストに [Storybook](#) を利用できるかもしれない。機能・構造と外観は分離され、View と密接な機能はクリックやホバーによる外観の変更だからテストは HTML/CSS で代替しやすくなる。例えばボタンの View をテストしたいとして、その構造表現はプラットフォーム固有 API でなく `<div>` や `` であっても問題ないはず。

サードパーティー製 View コンポーネントの普及も見込める。プラットフォーム共通化が進むことで開発者は大幅に増加するだろう。結果、View コンポーネントをライブラリーとして配信する動きが現在より活発となり、[GrapeCity](#) のようにコンポーネント販売ビジネスする企業も増えそうだ。

文書

CSS で View = 組版を定義できるなら、組版ソフトウェアと Web ブラウザー間で処理エンジンを共有する方向に進むんだろう。自作にこだわる強い理由がない限り、既存エンジンを利用するほうが開発コスト的に好ましい。

Web ブラウザーでは Opera と Microsoft が独自エンジンをやめて Chromium を採用した。これは多様性の観点では損失だが、仕様の再現性としてはメリットになる。エンジン開発の優劣を競うより、それを利用する部分に注力したほうがよいと判断したのだろう。組版ソフトウェアでもそうなるかもしれない。

例えば自動組版。これを Adobe InDesign で処理する場合は ExtendScript Toolkit や XMLなどを駆使するわけだが、CSS で組版可能なら Web ブラウザーも選択肢となり得る。本書の作成に用いられた Vivliostyle はその実例で、Headless Chrome により Chrome を呼び出して組版している。

いずれはアプリケーションと文書の垣根さえも取り去られ、グラフィック デザイナーがどちらも扱う時代がくるかもしれない。

Vivliostyle のこれから開発課題

村上 真雄 (@MurakamiShinyu)

Vivliostyle Foundation 代表

先日のイベント〈Vivliostyle 開発者とユーザーの集い 2019 夏〉^[1] の《第 1 部》開発者ミーティング「Vivliostyle 開発のこれまでと、これからへ」^[2] で、Vivliostyle.js ソースコードの TypeScript 化が完了したことと今後の開発が楽になったことと、開発課題(issue)を課題カテゴリーごとのプロジェクトに整理して GitHub Projects^[3] で管理するようにしたことを説明して、ミーティング参加者(30名くらい)にどのプロジェクトが重要か聞きました(複数回答)。結果は以下:

- Bugs (レイアウトの不具合など直す) : 7 人
- PDF printing (PDF 出力および印刷に関する問題を解決する) : 7 人
- Input formats (多様な入力文書フォーマットのサポート) : 7 人
- Paged media (CSS ページメディア関連仕様のサポート) : 15 人
- Typography (文字組版関連 CSS 仕様サポート) : 8 人
- Layout enhancement (高度なレイアウトの CSS 仕様サポート) : 10 人
- Web standards (Web 標準仕様のサポート) : 7 人
- Other improvements (その他の改良):
 - Improved error handling (エラー処理改良) : 5 人
 - UI multilingualization (UI の多言語化) : 3 人
 - Search function (Viewer に検索機能) : 3 人
 - Chrome extension (Chrome ブラウザ拡張) : 6 人
 - Documentation (ドキュメントの充実) : 12 人

この結果を見てわかるのは、どのプロジェクトもそれぞれ重要だけど、特に CSS ページメディア関連仕様や高度なレイアウトの CSS 仕様のサポートを拡充することへの期待が高いということです。それからドキュメントの充実ということも。

[1] Vivliostyle 開発者とユーザーの集い 2019 夏開催しました！ <https://vivliostyle.org/ja/blog/2019/09/04/vivliostyle-dev-user-meetup-tokyo20190831/>

[2] 「Vivliostyle開発のこれまでと、これからへ」 <http://bit.ly/vivdev20190831>

[3] <https://github.com/vivliostyle/vivliostyle.js/projects/>

これを踏まえて今後、とくに要望が多い項目を意識しながらも、どのプロジェクトも進めていきたいところです。とはいっても、どれもそんなに簡単なプロジェクトではありません。もっと皆様の協力が得られることが必要であるし、そのためには、その課題について理解を広めなければと思います。そこで、各プロジェクトの主な課題について、解説します。

Bugs (レイアウトの不具合など直す)

現在、約 30 数件のバグが残っています。プログラムの誤りや、実装が不十分であるために期待されるレイアウトにならないなどです。バグというより現在の実装での制限事項であって解決には本格的な処理の見直し・再設計が必要という問題は別のプロジェクトに割り振っているので、ここにあるのは比較的単純な問題です。

そのいくつか：

- [#544: column-fill: balance on vertical writing mode causes columns left-aligned](#)
 - 不具合：縦書きの段組の最後のページの段が左寄せになる。
 - 回避方法：縦書きの段組では `column-fill: auto` を指定する。（初期値 `column-fill: balance` が縦書きのとき問題あり）
- [#438: footnotes in table are ignored](#)
 - 不具合：テーブル内の脚注が無視される。
- [#534: Table break when there are footnotes](#)
 - 不具合：ページに脚注があるときテーブル内での改ページがされずテーブルの前で改ページが発生する。
- [#546: Inline-block or ruby at beginning of a block causes unexpected page/column break](#)
 - 不具合：ブロックの先頭にルビや inline-block があるときブロックの途中での改ページ／改段がされずにブロックの前で改ページ／改段が発生する。
- [#516: Absolute positioned elements causes unwanted page break when its height is more than page height](#)
 - 不具合：絶対位置指定で配置する要素の高さがページに収まらないとき同一ページに配置したい他の要素との間で改ページが発生する。

PDF printing (PDF出力および印刷に関する問題を解決する)

現状はブラウザの PDF 出力に依存するため、これらの問題があります。

- #437: please use CID instead of Type3
 - **問題** : Chrome ブラウザの「PDF に保存」で PDF を生成すると、OpenType/CFF フォントが Type3 フォントとして埋め込まれる。これでは印刷に向かない。
 - **代替手段その1** : TrueType フォントのみを使用する。
 - **代替手段その2** : Chrome 以外のブラウザ (Safari や Firefox) を使用する。
 - **代替手段その3 (PDF の後処理で解決)** : Vivliostyle + Chrome ブラウザで出力した PDF を Mac の「プレビュー」アプリで開いてファイルメニューの「書き出す」で「フォーマット: PDF」「Quartz フィルタ: Create Generic PDFX-3 Document」で別の PDF に書き出すと Type3 フォントの埋め込みの代わりに、テキストがグラフィック (アウトライン) に変換される。テキストデータを取り出せない PDF になるが、印刷には使える。
- #429: support PDF standard
 - **要望** : 印刷用の PDF の標準である PDF/X-1a や PDF/X-4 をサポートしてほしい。
 - **代替手段 (PDF の後処理で解決)** : 前項の「代替手段その3」でできた PDF を Acrobat Pro で開き、「プリフライト: PDF/X に変換」で「PDF/X-1a (Japan Color 2001 Coated) に変換」など選んで変換すると、PDF/X-1a 準拠の PDF になります。
- #419: Minimum width of borders is too thick
 - **問題** : Chrome ブラウザで PDF 出力すると border の最小幅は 1px = 0.75pt = 0.265mm で、border がそれより細くならない。
- #418: Add TrimBox and BleedBox to output PDF
 - **要望** : Vivliostyle でトンボ付きで PDF 出力したとき TrimBox と BleedBox の情報も出力されてほしい。
- #432: @page size value ignores decimal numbers
 - **問題** : Chrome ブラウザでは CSS の `@page { size: ... }` で指定のページサイズで PDF が生成されるが、そのとき整数ポイント単位に丸められてしまい正確なページサイズにならない。

Input formats (多様な入力文書フォーマットのサポート)

Vivliostyle は現在、入力文書フォーマットとして (X)HTML 文書、複数の HTML 文書をまとめた Web 出版物、および、ZIP 解凍済みの EPUB をサポートしています。また、文書内で使える数式のフォーマットとして、MathML、AsciiMath、TeX の数式をサポートしています（MathJax ライブラリを利用）。これをもっと充実させたいという要望があります。主なもの：

- [#541: Support non-unzipped EPUB loading](#)
 - **要望**：EPUB ファイルを直接（解凍していないくとも）ロードできるようにしてほしい。
- [#524: Vexflow and vextab](#)
 - **要望**：HTML 上で楽譜を表示するための API と簡易言語である VexFlow と VexTab をサポートしてほしい。
- [#168: Let authors use their own MathJax configuration](#)
 - **要望**：数式を表示するのに使用される MathJax の設定を制作者側でカスタマイズできるようにしてほしい。

Paged media (CSSページメディア関連仕様のサポート)

[CSS Paged Media](#) および [CSS Generated Content for Paged Media](#) 仕様のサポートに関する課題です。主なもの：

- [#59: Root element styles are not inherited to page context](#)
 - **問題**：ルート要素に指定したスタイル（フォント指定など）が、ページコンテキストつまり `@page { ... }` のブロック内に継承されない。
- [#545: Support named strings for running headers and footers](#)
 - **要望**：本文中の見出しの内容のテキストを `string-set` プロパティで名前付き文字列としてセットして、それをページマージン領域に柱（欄外見出し）として表示できるようにする機能。
 - **代替手段**：現在の Vivliostyle では、複数の HTML 文書で構成される本の場合は、`env(pub-title)`、`env(doc-title)` で本のタイトル（メインの HTML のタイトル）と個別 HTML 文書のタイトルをそれぞれ出力できる。

- #424: Support for running elements?
 - **要望**：本文中の見出しの要素をページマージン領域に柱（欄外見出し）として表示できるようにする機能。string-set が単純なテキストだけ扱えるのに対して、こちらは HTML 要素を扱える。
 - **代替手段**：EPUB Adaptive Layout を使用する。`-epubx-flow-into: 名前` の指定がある要素を `-epubx-flow-from: 名前` の指定があるページ区画に配置できる。
- #425: Support for named pages?
 - **要望**：ページの種類（例：扉、序文、本文、奥付、など）ごとに名前付きのページのスタイルを `@page 名前 { ... }` で定義して、それを文書中の該当要素に `page: 名前` というプロパティ指定で割り当てる機能
 - **代替手段その1**：複数の HTML 文書で構成される本にして、ページの種類ごとに別のスタイルシートを HTML 文書に指定することで、ページのスタイルを切り替えることができる。
 - **代替手段その2**：EPUB Adaptive Layout を使用する。ページの種類ごとにページテンプレートを定義できる。
- #428: Support "@page :blank"
 - **要望**：改丁（章を奇数ページから開始するなど）で空白ページができるときに、その空白ページのスタイルを指定できるようにする `:blank` ページセレクターのサポート。
- #149: Add support for GCPM bookmarks
 - **要望**：本文中の HTML の見出しの要素に `bookmark-level` プロパティで階層レベルを設定して、それを Viewer の目次パネルや PDF の「しおり」に出力できるようにする。

Typography (文字組版関連CSS仕様サポート)

文字組版の問題：[CSS Text 3](#), [CSS Text 4](#) や [CSS Fonts](#) 仕様などのサポートに関する課題です。主なもの：

- #182: Support text-spacing
 - **要望**：行頭・行末・連続約物の詰め、和欧文間のアキなどを自動で処理する `text-spacing` プロパティのサポート。
 - **メモ**：日本語の文字組みを美しく読みやすくするためにほしい機能。いずれプラウザでこれが標準になるのが期待されるけれど、Vivliostyle 側で先行実装するのも可能だろう。

- [#204: Support unicode-range descriptor](#)
 - **要望** : Unicode コードポイントの範囲によってフォントを割り当てる `unicode-range` 記述子のサポート。
 - **メモ** : ブラウザでは標準的に使えるようになっているのに Vivliostyle で現在これが使えないのは残念なところ。
- [#154: Support the font-variant-* longhands](#)
 - **要望** : フォントの字形を切り替える `font-variant-*` (`font-variant-caps`, `font-variant-east-asian`, `font-variant-ligatures`, `font-variant-numeric`, `font-variant-position`) プロパティのサポート。

Layout enhancement (高度なレイアウトのCSS仕様サポート)

高度なレイアウトを実現する CSS 仕様のサポート : [CSS Page Floats](#)、[CSS Grid Layout](#)[CSS Multi-column Layout](#)などに関する課題です。

- [#539: Support CSS Grid Layout](#)
 - **要望** : CSS グリッドレイアウトのサポート。
 - **メモ** : すでにブラウザで実装されてるのでそれを利用可能にしたい。
- [#543: Update CSS Page Floats](#)
 - **要望** : CSS Page Float は現在の Vivliostyle に実装されています。これをもっと使いやすく便利なものにするために改良したい。
- [#107: Broken multi-column inside a non-root element](#)
 - **問題** : ルートあるいは `body` 以外の要素に指定した段組が正常に組版できない。
 - **メモ** : 現在の段組の実装は、EPUB Adaptive Layout でのページ区画で指定する段組の実装がベースで、ルートまたは `body` 要素に指定された段組をページエリア全体の段組の扱いに変えてサポートしている。それ以外の要素での段組のプロパティはそのままブラウザに渡されるので、ページの中で部分的に使われたときに機能するが、複数ページに渡るときの分割処理などはできていない。
- [#542: Support column-span](#)
 - **要望** : 段組の段抜きを指定する `column-span` プロパティのサポート。
 - **代替手段** : EPUB Adaptive Layout を使用する。ページテンプレートでページ区画を定義することで、段組で段抜きの区画があるようなレイアウトが実現できる。

Web standards (Web標準仕様のサポート)

Web 標準仕様のサポート：最新ブラウザでサポートされている HTML や CSS の機能など。他のプロジェクトに分類できるものを除きます。

- [#266: Support the fill, max-content, min-content, and fit-content values for width and height.](#)
 - **要望**：ボックスのサイズを指定する `width` や `height` プロパティの値 `min-content`, `max-content`, `fit-content` のサポート。
- [#265: Support 'base' element](#)
 - **要望**：HTML 文書のベース URL を指定する `<base>` 要素のサポート。
- [#540: Support CSS custom properties \(variables\)](#)
 - **要望**：CSS カスタムプロパティ (CSS 変数) のサポート。
 - **代替手段その1**: Sass など CSS のプリプロセッサの変数を使う。
 - **代替手段その2**: EPUB Adaptive Layout の名前付きの値の定義 `@-epubx-define` を使う。
- [#145: add support for user-select](#)
 - **要望**：ユーザーが文章を範囲選択できるかどうかを設定する `user-select` プロパティのサポート。

Other improvements (その他の改良)

他の課題として、エラー処理の改善、Viewer UI の課題、ブラウザ拡張機能の要望、ドキュメンテーションについて、などがあります。

- [#460: autokill option](#)
 - **要望**：組版処理の異常によって処理が止まらなくなったときに自動的に処理を止めるオプションがほしい。
- [Vivliostyle UI issues](#)
 - [Localization of Vivliostyle Viewer UI](#)
 - **要望**：Viewer UI 言語が現在は英語だけなのだけど、ほかの言語のリソースも追加して、ブラウザの言語設定によって切り替えられるようにしたい。
 - [Include a search function in the UI](#)
 - **要望**：Vivliostyle Viewer に、文書中のテキストの検索機能がほしい。現在は、ブラウザのテキスト検索で表示中のページ内しか検索されない。

- Browser extension of Vivliostyle viewer
 - 要望：ブラウザ拡張機能バージョンの Vivliostyle Viewer がほしい。（かつて存在した Vivliostyle Chrome Extension）
- #423: Documentation: supported-features
 - 要望：サポートする機能のドキュメンテーションを改善してほしい。

まとめ

以上、Vivliostyle の今後の開発課題の主なものでした。

開発体制の現状と、まずできること

さて、これらをどうしていくかです。現在このオープンソース・プロジェクトの主体である Vivliostyle Foundation は、昨年、それまで Vivliostyle を開発してきた会社（現在の社名は Trim-marks Inc.）がオープンソース製品の取り扱いを終了したことに伴い、会社から独立してオープンソース・プロジェクトを維持していくために発足したものです。代表の私は、元々の Vivliostyle 社の創業者ですが、自分が Vivliostyle.js のメインの開発者だったことはそれまでなくて、そのコードに少しずつ手を入れ始めたのは昨年からです。そんな私と数名のボランティアの協力で維持しているものなので、開発リソースはとても限られていて、資金があるわけでもありません。

この記事の最初のほうで「とくに要望が多い項目を意識しながらも、どのプロジェクトも進めていきたい」と書いたものの、実際は今すぐ取りかかれないことはバグを直していくことや、すでにブラウザで実装されている機能を Vivliostyle から利用可能にするなど、比較的簡単な作業に限られそうです。

要望が多い、Paged media (CSS ページメディア関連仕様のサポート) の柱（欄外見出し）の機能や、Layout enhancement (高度なレイアウトの CSS 仕様サポート) の段組レイアウトの制限をなくすことなどは、それぞれ本格的な開発作業を要するものであり、それらを進められるようにするためにには、資金的な支援が得られてじっくり開発に取り組めるようになることや、あるいはボランティアでの貢献をしたい開発者・協力者たちのコミュニティを育てられるかということに掛かっています。

Vivliostyle を活用してください！ それから、開発支援に感謝！

多くの人に使ってもらえることで Vivliostyle が世の中に認知されて、プロジェクトへの協力が得られやすくなることを期待しています。

Vivliostyle をどう活用するかは、そのオープンソースのライセンス (AGPLv3) に違反しないかぎり自由です。自費出版への利用などはもちろんですが、企業内での利用、商業出版への利用、出版コンテンツの閲覧サービスへの利用、Web コンテンツの印刷機能への利用など、いろいろと使い道があるでしょう。

そのような Vivliostyle 活用の相談については、コミュニティでの投稿歓迎ですが、オープンにできない案件については私に直接ご相談くだされば対応します（内容によって有料でのコンサルティングも）。また、現在の Vivliostyle に足りない機能や修正が必要な問題（すでに issue 登録されている課題でもそれ以外でも）がある場合、その開発についてご相談ください。オープンソースですので、特定ユーザー（企業など）の必要によりそのユーザーの負担（共同開発や開発費負担）で開発された機能もオープンソースで公開されて、あとから利用する別のユーザーにも恩恵があるということになります。現在 Vivliostyle で利用できる機能の一部はそのような形で支援を受けて開発されたものです。Vivliostyle のリリースノートの謝辞（Acknowledgements）に開発貢献・支援いただいた企業などのお名前を入れております。開発支援に感謝します。

開発に興味ある人は、一緒に Vivliostyle のソースコードを読もう

Vivliostyle.js ソースコードの TypeScript への移行は、ソースコードを読みやすくして、今後の開発がよりスムーズに行えるようにと行ったものですが、それでもこのプログラムを理解して開発できるまでになるのは容易ではないと思います。

そこで、みんなでソースコードを読もうという、Vivliostyle コードリーディング会をやっていこうという声が上がってます。そのスケジュールなど決まつたら案内しますのでご参加ください。

Vivliostyle Viewer (vivliostyle.js + vivliostyle-ui) をソースコードからビルドして、テスト、デバッグ実行する手順などは、[Vivliostyle.js Development](#) にあります。ぜひこの手順でビルドして試してみてください。

以下、Vivliostyle の開発に関係する URL のまとめです：

- Vivliostyle 公式サイト（日本語）：<https://vivliostyle.org/ja/>
 - コミュニティ：<https://vivliostyle.org/ja/community/>
 - ドキュメント：<https://vivliostyle.org/ja/docs/>
 - Vivliostyle.js Development:
<https://github.com/vivliostyle/vivliostyle.js/wiki/Development>

- vivliostyle.js: <https://github.com/vivliostyle/vivliostyle.js>
 - issues: <https://github.com/vivliostyle/vivliostyle.js/issues>
 - projects: <https://github.com/vivliostyle/vivliostyle.js/projects>
- vivliostyle-ui: <https://github.com/vivliostyle/vivliostyle-ui>
- vivliostyle-print: <https://github.com/vivliostyle/vivliostyle-print>
- vivliostyle-savepdf: <https://github.com/vivliostyle/vivliostyle-savepdf>

不具合や要望などフィードバックをお寄せください

Vivliostyle を使っていて気になったことや、必要な機能の要望などは、どんどんお寄せください。ユーザーからのフィードバックがあることは励みになります。

不具合や要望などのフィードバックは、GitHub の issue に登録してくれるとありがたいですが、Vivliostyle のコミュニティ: Slack、Twitter、Facebook グループでの質問やフィードバックも歓迎します。

Vivliostyle コミュニティをチェック！

- Vivliostyle コミュニティ <https://vivliostyle.org/ja/community/>

ここから参加できる Vivliostyle の Slack が、Vivliostyle 開発者とユーザーのコミュニティの中心になっています。現在の参加者数45名くらい。まだの方はぜひご参加ください。次のチャネルなどがあります：

- #general: いろいろお知らせなど
- #random: 自己紹介や雑多な話題をどうぞ
- #issues: Vivliostyle のバグや機能追加の要望をどうぞ
- #q-and-a: Vivliostyle についての質問と回答
- #techbookfest: 技術書典向けに Vivliostyle ユーザー会の合同誌を作る相談など

それから Vivliostyle の Twitter や Facebook グループもどうぞよろしく。Vivliostyle への皆様のご支援に感謝します！

著者プロフィール



緑豆はるさめ spring_raining

最近の悩みは業務プログラミングばかりで自発的な開発ができないことです。趣味開発してえよ……。



小形 克宏 ogwata

文字とコンピュータのフリーライター。電腦マヴォ合同会社 業務執行社員。



田嶋 淳 JunTajima

日々 DTP データから EPUB 作ってます。ビールの誘いは断らない。



アカベコ akabekobeko

楽しいことなら何でもやりたいプログラマー。Web フロントエンド、モバイル(iOS/Android)、デスクトップ(Native/Electron)開発します。



村上真雄 MurakamiShinyu

Vivliostyle Foundation 代表。CSS 組版ちょっとできます。

Vivliostyle で本を作ろう Vol. 2

2019年9月22日 初版発行

発行 Vivliostyle ユーザー会

編集 緑豆はるさめ

<https://hanusamex.com>

hanusamex.com@gmail.com

印刷 日光企画(表紙: NP ホワイト 200kg マット PP 加工 / 本文: 上質紙 90kg)

<https://www.nikko-pc.com>

© Vivliostyle User Group