

"SMACSS is becoming one of the most useful contributions to front-end discussion in years"



Scalable and Modular Architecture for CSS

日本語

A flexible guide to developing sites small and large.

by Jonathan Snook

Copyright 2012 Jonathan Snook
All Rights Reserved

SMACSS: Scalable and Modular Architecture for CSS 日本語
<http://smacss.com>

ISBN 978-0-9856321-2-0

Snook.ca Web Development, Inc.
Ottawa, Ontario, Canada
<http://snook.ca>

Translation by Yuya Saito

著者について

こんにちは、Jonathan Snookです。私はウェブデベロッパ/デザイナーでウェブサイトを1994年から趣味で作りはじめ、1999年からは仕事として作っています。

また、Snook.caというブログを運営していて、そこで私はウェブ開発に関するティップスやブックマークを紹介しています。またカンファレンスやワークショップに登壇したりもし、世界を旅して私が知っていることを共有できて非常にありがたいと思っています。

これまでに共著が2冊、1つはThe Art and Science of CSS(Sitepoint刊)、もう1つがAccelerated DOM Scripting(Apress刊)です。また.net magazineやA List Apart、Sitepoint.comなど多くのオンライン、オフライン媒体に寄稿しています。

本書は最も最近のYahoo! Mailのリデザインを含め、サイトの規模の大小を問わず数百ものウェブプロジェクトに関わってきた経験を、共有するために執筆しました。

この場を借りてコミュニティの皆さんに心からの感謝をします。皆さん全員が今の私のキャリアを作ってくれました。特にKitt Hodsdonは私にこの本を書くように促し、全員に共有してくれた。最後に私がよりよい人間であることを手助けしてくれている2人の子どもたち、HaydenとLucasにも感謝したい。

翻訳者について

本書は私、齊藤祐也が翻訳をしました。Jonathan Snook氏と同じくウェブデザイナー/フロントエンドデベロッパーをしています。翻訳は本業ではありませんが、これまでにいくつか海外のブログ記事やオープンソースソフトウェアのドキュメントの翻訳を行っています。

またen.ja ossというオープンソースソフトウェアのドキュメント翻訳を行うコミュニティの主宰をしています。

Twitterでは@cssradarというアカウントで海外のフロントエンド情報を毎日紹介しています。

本書の翻訳にあたり、谷拓樹、石本光司、佐藤歩の3名に多くの助言をいただきました。この場を借りて皆様に感謝いたします。

イントロダクション

私はこれまでに数えきれないほどのウェブサイトを作成してきた。数百ものウェブサイトを作ってきたからには私が『ある1つの手法』を発見したと思っているかもしれないが、そんな手法は存在しない。私が見つけたのはCSSをより体系立て、より構造化させることで制作とメンテナンスをより容易に行うテクニックだ。

大規模なプロジェクトにおいて最適な構造を見つけ出すために自分自身(そして自分の周りの人たち)のプロセスを分析した。コンセプトは小規模なサイトでの開発では漠然としたものだったが、より複雑なプロジェクトで開発することによって明確になってきた。大規模なサイト、または大規模なチームでの開発と小規模なサイト、複雑ではなく、変更も少ない、での開発では苦勞するポイントが異なるが、以下で紹介していく手法は大規模なサイトであっても小規模なサイトであっても同じように有効なアプローチである。

SMACSS (スマックスと発音する) は厳格なフレームワークというよりはスタイルガイドである。インストール、またはダウンロードするライブラリは本書にはない。SMACSSはデザインプロセスを分析するための手法であり、厳格なフレームワークを柔軟な思考過程とする手法だ。そしてCSSを使ったウェブサイトの開発に対する一貫したアプローチをドキュメント化する試みでもある。いったい誰が現在においてCSSを使わずにサイトを制作するだろうか!? このアプローチのすべてを活用するのも、部分的に活用するのも、どちらでも最適な方を選んでほしい。もちろん使わなくてもかまわない。このアプローチがすべての人の好みに合わないのは理解できる。ウェブ開発においてほぼすべての質問に対する回答は『時と場合による』だからだ。

何が書かれているの？

私の考えはCSSの基礎設計概念に関連したいくつかのトピックに区分されていて、それぞれの考えはセクションごとに詳細に紹介されている。セクションを順番通りに読んでもいいし、無視してもいい、または関連の強い部分を選んで読んでもかまわない。1,000ページに渡るような本ではなく、セクションは比較的短く、簡単に会得することができるだろう。

では早速はじめて行こう！

CSSルールのカテゴリライズ

どんなプロジェクトでもなんらかの構成は必要だ。すべての新しいスタイルをファイルの最後に追加していくことはスタイルを発見するのも難しくするし、ほかのプロジェクトメンバーを非常に混乱させることになる。もちろんすでになんらかの構成をもっていることだろう。本書を読み進めていくうちに現在のプロセスをよりよくする方法を見つけてほしいし、もし私がラッキーであれば、そのプロセスを向上させる新たな手法が見つかることだろう。

自分の手元で自由にできるIDセクタやクラスセクタ、どんなセクタからどれを使うかどのように決めるだろうか？どうやってどの要素にスタイルの魔法をかけるのか決めていっているだろうか？そしてどのようにサイトやスタイルの構成を理解しやすくしているだろうか？

SMACSSの根底にあるものがカテゴリライズだ。CSSのルールをカテゴリライズすることでパターンが見えはじめ、そうすることでよりよいプラクティスをそのパターンに対して実行できるようになる。

カテゴリには以下の5つがある：

1. ベース
2. レイアウト
3. モジュール
4. 状態(ステート)
5. テーマ

このカテゴリを混ぜ合わせたスタイルをよく見かけるが、何に対してスタイルを追加しているのかに対してより敏感であれば、これらのルールが絡み合うことから生まれる複雑性を回避することができる。

それぞれのカテゴリはいくつかのガイドラインを当てはめることができる。これらのカテゴリによる緩やかな分離こそ開発プロセス中にどのようにコーディングしていくか、そしてなぜその方法でコーディングするのか、という疑問を投げかけるきっかけとなる。

何かに対してカテゴリ化をすることのほとんどの目的はデザインの中で繰り返されるパターンを体系立てるためだ。繰り返しは結果として少ないコードとなり、メンテナンス性をより高め、ユーザ体験における一貫性を向上させる。これらはどれをとってもいいことばかりだ。ルールに対する例外はたとえ有益であっても理にかなったものであるべきだ。

ベースルールはデフォルトだ。このルールはほとんど単独で1つの要素セクタとなるが、属性セクタや疑似クラスセクタ、子セクタ、兄弟セクタを利用することもできる。原則的にベーススタイルはあるページにおいての要素に対して、その要素の見え目がこうであると定義するものになる。

ベーススタイルの例:

```
html, body, form { margin: 0; padding: 0; }
input[type=text] { border: 1px solid #999; }
a { color: #039; }
a:hover { color: #03C; }
```

レイアウトルールはページをセクション毎に分割する。レイアウトは1つ以上のモジュールを保持する。

モジュールはデザインにおいて再利用可能なモジュラーパーツだ。吹き出しであり、サイドバーセクションであり、商品リストなどはその例になる。

状態(ステート)ルールはある特定の状態においてモジュールやレイアウトがどうスタイルされるかを説明する方法だ。隠れているのか、展開されているのか？利用可能なのか、非活性なのか？要素の大きさを問わずモジュールやレイアウトがスクリーン上でどういう見た目になるのかを説明するために存在し、またホームページや内部

ページのような異なるビューでモジュールはどう見えるのかを説明するために存在する。

最後にテーマルールは状態(ステート)ルールとモジュールやレイアウトがどう見えるかという点では同様のルールとなる。ほとんどのサイトではテーマレイヤーが必要にはならないが、存在を知っておくことはよいことだ。

命名規則

5つのカテゴリにルールを分離した上で命名規則を持つことは特定のスタイルがどのカテゴリに属するのか、ページ全体のスコープにおいてスタイルの役割を持つのかを瞬時に理解できるという利点になる。

私にとってはレイアウト、状態(ステート)、モジュールのルールをプリフィックスを使って区別する方法がわかりやすい。レイアウトには`l-`を使っているが、`layout-`としてもいいだろう。`grid-`のようなプリフィックスはより明確にレイアウトとそうでないスタイルを区別するのに役立つ。状態(ステート)ルールでは、`is-hidden`や`is-collapsed`のような形で`is-`を利用している。こうすることで非常に読みやすい方法で要素を説明することができる。

モジュールはどのプロジェクトでも大部分を占めることになる。結果として`.module-`のようなプリフィックスをすべてのモジュールにつけてしまうと不必要に冗長になってしまう。モジュールルールではモジュールそのものの名称を付けるにとどめるのがいいだろう。

クラスの例

```
/* モジュールの例 */
.example { }

/* 吹き出しモジュール */
.callout { }

/* 吹き出しモジュールと状態(ステート)ルール */
.callout.is-collapsed { }

/* フォームフィールドモジュール */
.field { }

/* インラインレイアウト */
.l-inline { }
```

モジュール内で関連し合う要素にはベースとなるモジュール名をプリフィックスとして利用する。例えばこのサイトではコードの例は `.exm` で囲まれていて、キャプションには `.exm-caption` というクラスが付与されている。こうすることでキャプションクラスを見るだけでコード例に関連していることを一目で知ることができ、それに対するクラスを見つけやすくなる。

ほかのモジュールのバリエーションとなるモジュールに対しても同じようにベースとなるモジュール名をプリフィックスとして利用する。このようなサブクラスについてはモジュールの章で詳しく解説する。

これらの命名規則は本書内で繰り返し利用される。ここで述べたほかの事柄と同じように、これらのガイドラインを厳格に守らなければならない、というように考えないで欲しい。規則を作り、ドキュメントし、継続利用していくことこそが大事なことだ。

ベースルール

ベースルールは1つの要素に対して要素セレクタ、子孫セレクタ、または子セレクタ、疑似クラスを使って適応される。クラスやIDセレクタは使用しない。ページ全体で要素がどう見えるのかを定義するデフォルトスタイルとなる。

ベーススタイルの例

```
body, form {
    margin: 0;
    padding: 0;
}

a {
    color: #039;
}

a:hover {
    color: #03F;
}
```

ベーススタイルには見出しのサイズ、デフォルトリンクスタイル、デフォルトフォントスタイルそしてbodyの背景が含まれる。ベーススタイルにおいては!importantが必要になるケースはない。

白以外を背景色として設定しているユーザも少なからずいるので、bodyの背景を指定することを強くおすすめしておきたい。背景が白であるという見込みがあるだけでは、場合によってはデザインが変わってしまうし、選択したフォントカラーによっては最悪の場合、サイトが利用できない状態に陥ってしまうことも考えられる。

CSSリセット

CSSリセットとはマージン、パディングを含むほかのプロパティのデフォルト値を除去、あるいはリセットするためのベーススタイルセットで、クロスブラウザ環境においてサイト制作のための一貫性のある基礎を作ることを目的にしている。

多くのリセットフレームワークは必要以上に強引で場合によっては問題を解決するよりも問題を生み出すことにもなりかねない。マージンやパディングを要素から削除することは、もう1度設定することを意味し、結果的にブラウザが読み込むコードが増えることになることもある。

多くの方はスタイルのリセットはサイト開発を行うのに便利なツールだと考えているが、リセットフレームワークがどのような欠点を抱えているのかきちんと理解し、それに基づいて利用してほしい。

プロジェクトに関わらず、自分自身がよく使うデフォルトスタイルを開発することももちろんよいことだと言える。

レイアウトルール

CSSはそもそもページ上に要素をレイアウトするために利用されてきた。しかしページにおける主要なコンポーネントとそうでないものとはその指示が異なる。小さなコンポーネント、例えば吹き出しやログインフォーム、あるいはナビゲーションアイテムなどはヘッダーやフッターなどの主要なコンポーネントの中に配置される。本書では小さなコンポーネントをモジュールと定義し、次の章で詳しく紹介する。ここでいうレイアウトとは主要なコンポーネントを指す。

レイアウトスタイルそのものにも再利用性を元に主要なスタイルとそうではないスタイルに分けて考えることができる。ヘッダーやフッターなどのような主要なレイアウトスタイルは伝統的にIDセレクトを使ってスタイルされるがページ内の全てのコンポーネントで利用される要素がないかについて少し時間をかけて考えてほしい。それらの共通要素についてはクラスセレクトを利用しよう。

レイアウトの宣言

```
#header, #article, #footer {
  width: 960px;
  margin: auto;
}

#article {
  border: solid #CCC;
  border-width: 1px 0 0;
}
```

サイトによってはより汎用化された960.gs¹のようなレイアウトフレームワークが必要になるかもしれない。そのようなマイナーなレイ

1.<http://960.gs/>

アウトスタイルはIDの代わりにクラス名を使ってページ内で何度も利用できるようにスタイルする。

一般的にはレイアウトスタイルは1つのIDあるいはクラスセレクタのみの利用で十分だ。しかし例えばユーザ設定などによって異なるレイアウトを持つようなケースに対応する場合は、レイアウトスタイルとして宣言はされるが、また別のレイアウトスタイルとのコンビネーションになることも考えられる。

高次レベルのレイアウトスタイルがほかのレイアウトスタイルに影響を与える場合の例

```
#article {
  float: left;
}

#sidebar {
  float: right;
}

.l-flipped #article {
  float: right;
}

.l-flipped #sidebar {
  float: left;
}
```

このレイアウト例では、`.l-flipped`クラスがbodyなどの高次レベルの要素に適応された場合に#articleと#sidebarのコンテンツが入れ替わる。サイドバーは右から左に移動し、記事コンテンツはその反対となる。

2つのレイアウトスタイルを利用して流動的なレイアウトから固定レイアウトに切り替える。

```
#article {
  width: 80%;
  float: left;
}

#sidebar {
  width: 20%;
  float: right;
}

.l-fixed #article {
  width: 600px;
}

.l-fixed #sidebar {
  width: 200px;
}
```

この例では、`.l-fixed`クラスはパーセンテージを使った流動的なレイアウトからピクセルを使った固定のレイアウトにデザインを変更する。

もう1点このレイアウト例で注目してほしいのが命名規則だ。IDセクタを使った宣言では命名は的確で命名規則を利用していない。一方クラスベースのセクタでは`l-`プリフィックスを利用している。こうすることでスタイルの目的を明確にしモジュールや状態(ステート)との区別もしやすくなる。レイアウトスタイルではIDセクタを利用するとしたら、主要な要素にのみ利用する。IDセクタにも同じような命名規則をするのはもちろん問題ないが、クラスベースのセクタに比べるとその必要性が薄い。

IDセクタの利用

補足するとID属性をHTMLで利用すること自体はよいことでもあるし、場合によっては絶対に必要にこともある。例えばJavaScriptから利用するための効率的なフックとしてなどがそうだ。CSSという点で

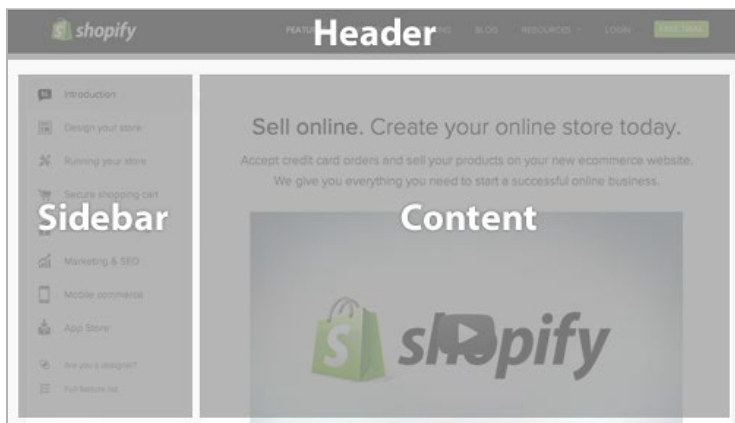
はパフォーマンスはIDセレクトとクラスセレクトとに大きな違いはないし、詳細度を高めてしまうことによりスタイルをより複雑にしていってしまうことがある。

レイアウト例

理論と実装は往々にして異なるものなので、実際のウェブサイトを見ながらどの部分がレイアウトで、どの部分がモジュールとなるのかについて見ていこう。



Shopifyのウェブサイトを見てみると、ほとんどのウェブサイトで見られるパターンを見つけることができる。例えば、ヘッダー、メインコンテンツエリア、サイドバーそしてフッターなどがそうだ。



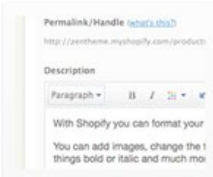



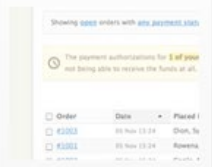

執筆時点ではこのウェブサイトでは図に沿った形で分離でき、主要なセクションそれぞれに対してID属性が割り当てられている。

CSSの構造はおそらくこのようになるだろう:

```
#header { ... }
#primarynav { ... }
#maincontent { ... }

<div id="header"></div>
<div id="primarynav"></div>
<div id="maincontent"></div>
```

非常に分かりやすい例でおそらく『これをやる方法を説明するの?』と思っていることだろう。それではページの別の部分を見ていこう。

| | | |
|--|---|--|
|  |  |  |
| <p>Content Management</p> <p>The built-in CMS feature allows you to create webpages, blog about products and more.</p> | <p>Ecommerce Analytics</p> <p>Learn where your businesses' customers come from. We also integrate with Google Analytics.</p> | <p>Mobile Commerce</p> <p>Built-in mobile commerce features include an iPhone app and a mobile storefront.</p> |
|  |  |  |
| <p>On-Board Coaching</p> <p>We advise you on how to sell online. You can also contact our support team by phone or email.</p> | <p>Manage & Accept Orders</p> <p>Keep track of all your orders and securely charge your customer's credit cards.</p> | <p>Marketing Features</p> <p>Built-in SEO, coupon codes, A/B testing and other features help you sell your items.</p> |

フィーチャーセクションを見てみるとニュースアイテムのグリッドある。CNNの現在のマークアップdivコンテナにさらに子要素としてdivが利用されている。私の場合はこのようなケースでは順序のないリストを使うだろう。では早速変更してみよう。

フィーチャーセクションのHTML例

```
<div>
<h2>Featured</h2>
<ul>
  <li><a href="#">...</a></li>
  <li><a href="#">...</a></li>
  ...
</ul>
</div>
```

SMACSSアプローチを考えない場合、おそらく `featured` というIDをDIVに付与しようとし、コンテンツのスタイルをそこから始めるだろう。

フィーチャーアイテムのリストに対する1つのスタイルアプローチ

```
div#featured ul {
  margin: 0;
  padding: 0;
  list-style-type: none;
}

div#featured li {
  float: left;
  height: 100px;
  margin-left: 10px;
}
```

このアプローチにはいくつかの前提条件が存在する:

1. ページ上に確実に1つしかフィーチャーセクションは存在しない
2. リストアイテムは左にフロートする
3. リストアイテムは100ピクセルの高さになる

これらは妥当な前提条件ではあるが、これこそが小規模サイトがこの構造を採用しても問題ない最もたる例と言える。おそらく変更はないし、おそらく今以上は複雑になるということは考えにくい。しかしそれはおそらくだ。大規模サイトではより高い確率で変更があり、そのためページ内のコンポーネントに対しての変更の可能性も高くなり、それらに対するスタイルを再変更することになるだろう。

もう一度コード例を見返してほしい。間違いなくいくつかの最適化を行うことができるはずだ。IDセレクトはタグセレクトと一緒に使う必要性はないし、リストは `div` に対して直接の子孫になるので、子セレクト (`>`) を利用することができたはずだ。

レイアウト観点で考えると、注意すべきなのはどうそれぞれのアイテムが関連しているかだけであり、モジュールのデザインそのものも、このレイアウトがどのコンテキストで利用されるのかについてもあまり考える必要はない。

OLまたはULにグリッドモジュールを割り当てる

```
.l-grid {
  margin: 0;
  padding: 0;
  list-style-type: none;
}

.l-grid > li {
  display: inline-block;
  margin: 0 0 10px 10px;

  /* IE7でinline-blockをブロック要素で実現するハック */
  *display: inline;
  *zoom: 1;
}
```

このアプローチでどんな問題を解決し、そしてどんな問題が発生したのか？(どんな解決も100%すべての問題を解決できることはほとんどない)

1. グリッドレイアウトはどんなコンテナであってもフロートスタイルのレイアウトを適応できるようになった
2. 適応性の深度を1つ減らすことができた (詳しくは適応性の深度についての章を参照)
3. セレクタの詳細度を減らすことができた
4. 必要条件から高さを排除することができた。その列内でもっとも高さのあるアイテムの高さにあわせる形になった

反対にどんな問題点があるか？

1. 子セレクタを利用することでIE6で正しく動作しなくなった。(子セレクタを利用しなければこの問題は解決できる。)
2. CSSはサイズが大きくなり、複雑性も高まった。

サイズが大きくなることについては争点にはなるが、非常にわずかだ。再利用可能なモジュールとなったので、サイト全体でコードを重複させることなく繰り返し利用することができる。複雑性の高まりもわずかだろう。古いブラウザのためのハックは好き嫌いはあるかもしれないが、必須になる。しかしセレクタが複雑ではないため詳細度への影響を最小限に止めながらこのレイアウトを拡張することもできる。

モジュールルール

前の章で少しだけ触れたが、モジュールとはページ内の個別のコンポーネントである。モジュールはナビゲーションであり、カルーセルであり、ダイアログであり、ウィジェットなどであり、ページの本質となる。またモジュールはレイアウトコンポーネント内に配置される。そしてモジュールはほかのモジュール内に配置されることもある。それぞれのモジュールはスタンドアロンのコンポーネントとして存在できるようにデザインされるべきである。そうすることでページはより柔軟になる。正しく実装されれば、モジュールは別のレイアウト内に簡単に移動できるし、移動してもくずれることはない。

モジュールに対するルールセットを定義する場合はIDや要素セレクタを避け、クラスセレクタのみを利用する。モジュールはいくつかの要素を含むことになるため子孫セレクタや子セレクタを使ってそれらの要素を指定することになる。

モジュール例

```
.module > h2 {
  padding: 5px;
}

.module span {
  padding: 5px;
}
```

要素セレクタを避ける

要素セレクタが予見しやすいものであれば子セレクタや子孫セレクタを要素セレクタに対して適応することは問題ない。もしspanがモ

ジュール内で毎回同じようにスタイルされるのであれば、`.module span`のように記述してもいい。

一般的な要素をスタイルする

```
<div class="fld">
  <span>Folder Name</span>
</div>

/* フォルダーモジュール */
.fld > span {
  padding-left: 20px;
  background: url(icon.png);
}
```

問題になるのはプロジェクトがより複雑になるにつれ、コンポーネントの機能を拡張する必要があり、このような一般的な要素をルール内に持つことはより制限となってしまう点だ。

一般的な要素をスタイルする

```
<div class="fld">
  <span>フォルダ名</span>
  <span>(32個)</span>
</div>
```

こうなると困ったことになる。フォルダーモジュール内の両方の要素にアイコンを表示はしたくない。このことが次のポイントに続く:

セマンティックを含む場合にのみ要素セレクタを使うこと。spanやdivにはセマンティック性はない。見出しには若干ある。要素に対してクラスを設定した場合は多分にセマンティック性を含む。

一般的な要素をスタイルする

```
<div class="fld">
  <span class="fld-name">Folder Name</span>
  <span class="fld-items">(32 items)</span>
</div>
```

要素に対してクラスを付与することで要素がどんな意味を持つのかというセマンティック性を向上させ、そうすることでそれらをスタイルする際の曖昧さを排除できる。

もしも要素セレクタを使う場合にはクラスセレクタ内の1つ下の階層にとどめるべきだ。言い換えると子セレクタを利用する状況であるべきである。または要素セレクタを割り当てるべきかどうか考えている要素が確実にほかの要素と勘違いされないという自信がなければならぬ。セマンティック性がspanやdivのように一般的であればあるほど、衝突を生み出すことになる。見出しなどのような、より強いセマンティック性を持つ場合はコンテナ内で素のまま利用されることが多く、要素セレクタをうまく使えることになる。

新たなコンテキスト

モジュールアプローチを行うことでどのタイミングでコンテキストの変更が発生しやすいかを理解しやすくなる。例えば新たなポジションのコンテキストの必要性はレイアウトレベルかモジュールのルートで発生する。

モジュールのサブクラス

同じモジュールを異なるセクションで利用する場合、まずは親要素を使ってそのモジュールのスタイルを変更しようとするだろう。

サブクラス

```
.pod {
  width: 100%;
}
.pod input[type=text] {
  width: 50%;
}
#sidebar .pod input[type=text] {
  width: 100%;
}
```


このアプローチの問題点は詳細度の問題を解決するためにセレクトアを追加するか、あるいは!importantを使い始めてしまうことだ。

podの例を詳しく説明するとinputには2つの異なる幅がある。サイト全体を通じてinputはlabelと並列に表示するため、inputは領域内で50%の幅を持つ必要がある。サイドバーではその領域が狭すぎるためinputの幅を100%にしlabelはその上に表示する。一見これで問題ないように思えるが新たにコンポーネントを追加する必要があったとし、それが.podのスタイルとほとんど同じスタイルだとするとクラスを再利用したくなる。しかしこのpodは少し特別でページ内のどこでも固定の幅を持つとする。若干の違いではあるが幅が180pxだとする。

詳細度の問題解決

```
.pod {
  width: 100%;
}
.pod input[type=text] {
  width: 50%;
}
#sidebar .pod input[type=text] {
  width: 100%;
}

.pod-callout {
  width: 200px;
}
#sidebar .pod-callout input[type=text],
.pod-callout input[type=text] {
  width: 180px;
}
```

#sidebarの詳細度を上書きするためセレクトアを繰り返すことになる。

ここで気がつかなければならないのは限定条件となるサイドバーレイアウト内はpodのサブクラスが必要となりスタイルもそれに合わせる必要があることだ。

詳細度の問題解決

```
.pod {
    width: 100%;
}
.pod input[type=text] {
    width: 50%;
}
.pod-constrained input[type=text] {
    width: 100%;
}

.pod-callout {
    width: 200px;
}
.pod-callout input[type=text] {
    width: 180px;
}
```

モジュールのサブクラスを作成するにはベースモジュールとサブモジュールのクラス名をHTML要素に適応する必要がある。

HTML内でのサブクラス

```
<div class="pod pod-constrained">...</div>
<div class="pod pod-callout">...</div>
```

場所を条件としたスタイルは可能な限り避けてほしい。ページ内でもサイト内でもモジュールの使用方法に合わせてモジュールの見た目を変更する場合はモジュールのサブクラスを利用するほうがいい。

詳細度の問題を解決するために(そしてもしIE6を無視できるのであれば)、以下の例のように複数クラスを使うこともできる。

サブクラス

```
.pod.pod-callout { }  
  
<!-- HTML内 -->  
<div class="pod pod-callout"> ... </div>
```

呼び出しの順番に依存する状態は問題になるかもしれない。例えば Yahoo! Mailでは別の場所からコードを呼び出すこともある。ベースボタンのスタイルがあり、そしてメール作成画面では特別な種類のボタンがある。しかしアドレス帳に連絡先を追加しようとすると別のプロダクトであるAddress Bookからコンポーネントを呼び出す (Yahoo!内ではアドレス帳は別のプロダクトである)。アドレス帳にはアドレス帳のベースボタンスタイルがあり、メールにあったサブクラスのボタンを上書きしてしまう。

もしプロジェクト内で呼び出し順が懸念される場合はとくに詳細度の問題について気をつけておくべきだ。

IDが適応されている特定のレイアウトコンポーネントはモジュールに対してより詳細なスタイルを供給することができるが、モジュールからサブクラスを作成することでモジュールは不必要に詳細度の高めずにすみ、サイトの内の別のセクションに移動することも容易に可能となる。

状態(ステート)ルール

状態とはほかのすべてのスタイルを拡張し上書きするものである。例えばアコーディオンセクションは畳まれているか広がっている状態にある。メッセージは成功かエラーの状態にある。

状態は一般的にレイアウトルールと同じ要素に適応されるか、モジュールクラスのベースとなる要素に適応される。

要素に状態を適応する

```
<div id="header" class="is-collapsed">
  <form>
    <div class="msg is-error">
      There is an error!
    </div>
    <label for="searchbox"
class="is-hidden">Search</label>
    <input type="search" id="searchbox">
  </form>
</div>
```

ヘッダ要素にはIDのみがあるため、もしあるとしたら、どんなスタイルであってもレイアウト目的のスタイルであると予測できる、そしてis-collapsedは畳まれた状態を意味する。この状態のルールがなければ、デフォルトは広がった状態であると推測することもできるだろう。

msgは単純でエラー状態が適応されている。代わりに成功状態が同じように適応されるのを想像するのは難しいことではない。

最後にフィールドラベルにはhiddenが付与されていて見えないようにしてあるが、スクリーンリーダーは読み出すことができる。この

場合は状態をベース要素に付与しモジュールもレイアウトも上書きしていない。

モジュールとは何が違うのか

サブモジュールスタイルと状態スタイルには非常に多くの類似点が見られる。双方とも現存する要素の見た目を変更するが、両者は大きな2点が異なっている:

1. 状態スタイルはレイアウトやモジュールに割り当てることができ
2. 状態スタイルはJavaScriptに依存するという意味を持つ

2つ目のポイントが最も大切な区別のポイントとなる。サブモジュールスタイルは要素に対してレンダリングのタイミングでスタイルを付与して以降変更されることはない。一方状態スタイルはクライアントサイドでページが表示されている間でも要素の状態が変更することを意味する。

例えばタブをクリックするとタブが活性化し、結果として `is-active` や `is-tab-active` クラスを割り当てるのが順当と言える。ダイアログの閉じるボタンをクリックするとダイアログは非表示になる、先ほどと同じ様に `is-hidden` クラスを付与するのが適切となる。

!importantの利用

状態はスタンドアロンとして作られるべきで1つのクラスセレクタのみを利用するべきだ。

状態スタイルは非常に複雑なルールセットを上書きするため、`!important` を利用することは問題にはならない。勇気を持って言うが推奨してもいい。(過去に私は `!important` は絶対に必要にならないと言っていたが複雑なシステム内では必要になるケースがある。) 通常は2つの異なる状態スタイルを同じモジュールに割り当てることはないし、2つの状態スタイルが同じスタイルセットにたい

して影響を与えることはない。そのため!importantを使った場合の詳細度の衝突はごく少ないはずだ。

上記を踏まえて、改めて気をつけてほしい。!importantの利用は本当に絶対に必要な場合のみとすること(次の例で理由その理由を説明する)。そして!importantの利用はほかの全てのルールタイプでは避け、状態スタイルのみで利用することを徹底してほしい。

モジュール内で状態スタイルを結合する

状態スタイルは必然的に継承に頼ってスタイルを適応することができない。状態は特定のモジュールに対して非常に独自性の高いスタイルを行う必要がある。

状態スタイルがある特定のモジュールのために利用される場合には状態スタイルのクラス名はモジュール名を含んでいるべきだろう。それらの状態スタイルはほかのサイト全体のステートルールの側ではなく、モジュールスタイルの側に記述すべきだ。

モジュール用の状態スタイル

```
.tab {
  background-color: purple;
  color: white;
}

.is-tab-active {
  background-color: white;
  color: black;
}
```

もし、CSSをその場で呼び出す場合には一般的な状態スタイルはベースと全体に属していると考えるべきで、最初のページ読み込みの際に呼び出しておくべきだ。特定モジュール用の状態スタイルであればそのモジュールが呼び出されるまでは呼び出しは必要ではない。

テーマルール

テーマルールが利用されるプロジェクトは決して多くはない、そのため個別のカテゴリとして分けるべきかどうか悩んだが、Yahoo! Mailでも利用したようにいくつかのプロジェクトでは必要になる。

言わずもがなだとは思いますが、テーマでは色やアプリケーションやサイトが持つルック&フィールを定義する。テーマをそれ自体のスタイルとして定義しておくことで別のテーマの再定義を容易にできる。プロジェクト内でテーマを持つ必要性はユーザに対して表面的なデザイン変更を提供したい場合に考えられるだろう。

例えばサイト内で別セクションでは別の色を使いたい場合やユーザ自身が色を変更できるような場合、または言語や国によって異なるテーマを設定したい場合などがある。

テーマ

テーマはすべての主要なルールに対して影響を及ぼす。デフォルトのリンク色を上書きしたり、モジュールの色やボタンなどを変更することもできる。またレイアウトを異なる配置にすることもできるし、状態がどう見えるのかを変更することもできる。

例としてダイアログモジュールに青いボーダー色をつけるとして、ボーダーそのものはモジュール内で初期設定を行いテーマで色を設定する。

モジュール内テーマ

```
/* module-name.css 内で定義 */  
.mod {  
    border: 1px solid;  
}  
  
/* theme.css 内で定義 */  
.mod {  
    border-color: blue;  
}
```

テーマの規模によってはテーマ特有のクラスを設定するほうが簡単になるかも知れない。Yahoo! Mailの場合ではページ内の特定の領域のみでテーマを設定した。こうすることでユーザに対してカスタマイズの自由を与えつつも全体のデザインの調和を崩すことなく新しいテーマを作成することができた。

より規模の大きなテーマを作成する場合にはtheme-プリフィックスを特定のテーマコンポーネントに対して利用することで、より多くの要素に対して簡単にテーマを定義することができる。

テーマクラス

```
/* theme.css 内で定義 */  
.theme-border {  
    border-color: purple;  
}  
  
.theme-background {  
    background: linear-gradient( ... );  
}
```

Yahoo! Mailの場合、50を超える全てのテーマで一貫性を保つためMustacheテンプレートを利用し色の値、背景画像を設定し、プロダクション環境用の最終的なCSSを生成している。

タイポグラフィ

テーマのもう1つの1面として、他言語対応などの大規模なケースではフォントそのものを再定義することもある。中国語や韓国語などの言語ではより複雑な文字を利用するためフォントサイズが小さいと読みづらくなることもある。その結果としてフォントスタイルに対して個別のルールを持つことで複数のコンポーネント内でフォントサイズの変更を容易に行うことができる。

フォントに関わる設定は通常はベース、モジュール、状態スタイルに影響を与える。フォントのスタイルは一般的にレイアウトレベルでは定義しない。それはやはりレイアウトが位置や配置のためにあるものであってフォントや色などのスタイル変更のためではないからだ。

テーマファイルと同じ様にフォントクラス(`font-large`のような)を定義する必要はないだろう。もし定義するのであれば、サイト内で3から6のフォントサイズに止めておくべきだ。もし6個以上のフォントサイズがプロジェクト内で宣言されている場合、ユーザはその違いに気がつくことはないのに、メンテナンス性を低下させることになる。

テーマの命名

テーマやタイポグラフィ用のクラスに命名するのは往々にして気持ちが良いものではない。それは我々がそういった命名をセマンティックではないと考える業界に身をおいているからだ。テーマコンポーネントの場合はそのそも本質的に見た目についてのルールになりセマンティックではない。タイポグラフィの場合は、それほど問題にはならない。デザインとは結局ビジュアルの階層構造を意味し、タイポグラフィはそれらを反映するべきだからだ。結果として使うことになる命名規則はHTML内で見出しを定義するのと同じく、重要性に応じるような規則であるべきだ。

状態の変更について

Photoshopドキュメントが目の前にあり、それをHTMLとCSS(もしくはJavaScriptも使って)でコーディングして欲しいと言われている。

ぱっと見た印象では構成から直接コードに落とし込みそうにも思えるが、ページ上にあるいくつかのコンポーネントは様々な状態を表現する必要がある。デフォルト状態がどのような見た目になっているのか、そして状態が変わった場合にそれをどう見せるのか。

状態が変わるとは？

状態の変更は以下の3つで表現できる:

1. クラス名
2. 疑似クラス
3. メディアクエリ

クラス名の変更はJavaScriptによって実行される。何らかのインタラクション、例えばマウスの動きやキーボードの入力のようなイベントが発生することによって新しいクラスが追加され、見た目の表現も変わる。

疑似クラスの変更は、当然のことながら疑似クラスを使って実行される。非常に数多くの疑似クラスがあり、状態の変更を表現するのにJavaScriptに頼らなくてもすむようになった。疑似クラスは疑似クラスが適応できる子要素、子孫要素にのみに対してのスタイルの変更を行うことができる。そうでなければ、やはりJavaScriptが必要になる。

最後にメディアクエリはビューポートのサイズの違いのような評価ポイントでどうスタイルされるべきかを表現する。

モジュールベースのシステムではそれぞれのモジュールに対して状態を元にしたデザインをどう当てはめるかについて考慮しておく必要がある。『何をデフォルト状態とするべきか』という疑問を常に考えておくことでプログレッシブエンハンスメントについて積極的に考えることにもつながる。プログレッシブエンハンスメントもまた問題に対するまた少し違うアプローチだ。

クラス名による変化

クラス名の変更はほとんどの場合は理解しやすい。クラス名は要素に対して付与され要素を別の状態に変化させる。例えば開くアイコンをクリックして要素を表示したり、反対に隠したりする。

JavaScriptでクラス名を追加し状態を変更する

```
// jQueryを使った場合
$('.btn-close').click(function(){
    $(this).parents('.dialog').addClass('is-hidden');
});
```

このjQueryを使った例では`btn-close`を付与されたすべての要素をクリックイベントハンドラを付与する。ユーザがボタンをクリックすると、イベントの発生源からDOMツリーを辿って`dialog`クラスが付与された親要素を見つかるまで探索する。そしてその要素に対して`is-hidden`という状態クラスを付与する。

通常は状態の変更を行うことは大きな影響を伴う。

よくあるインターフェイスのデザインとしてボタンをクリックされるとメニューを表示するなどだが、こういった場合、メニューは押された状態となり、さらに表示状態になる。このような変更に対応するにはどのような手段があるだろうか？その手段はHTMLの構造に大きく影響される。例えばYahoo!ではメニューはメニューはリンクエントリがあってから呼び出され、DOMの一番上に挿入される。この

ためボタンとメニューをフックするために命名規則を使わざるを得ない。

ドキュメント内でボタンとメニューが異なるパーツである場合

```
<div id="content">
  <div class="toolbar">
    <button id="btn-new" class="btn"
data-action="menu">New</button>
  </div>
</div>
<div id="menu-new" class="menu">
  <ul> ... </ul>
</div>
```

`data-action`はJavaScriptのクリックコールに反応し、「メニューを呼び出して。」という命令を発する。ボタンのIDを取得し、それにあうメニューを取得する。以下はjQueryを使った例:

jQueryを使ってメニューを呼び出す

```
// ボタンにクリックハンドラを付与する
$('#btn-new').click(function(){
  // クリックされたボタンをjQueryを使って取得
  var el = $(this);

  // ボタンの状態を変更
  el.addClass('is-pressed');

  // メニューをbtn-以降の文字列から探して、
  // メニューのセレクタとして追加
  $('#menu-' +
el.id.substr(4)).removeClass('is-hidden');
});
```

この例からわかるとおり、1つのアイテムが2つの異なるパーツのアイテムの状態をJavaScriptを使って変更している。

しかし、もしメニューがボタンのすぐ隣に配置してあったらどうだろうか？

ボタンとメニューがドキュメント内の同じパートにある場合

```
<div id="content">
  <div class="toolbar">
    <button id="btn-new" class="btn"
      data-action="menu">New</button>
    <div id="menu-new" class="menu">
      <ul> ... </ul>
    </div>
  </div>
</div>
```

前の例のコードと同じでも動作するため変更する必要はない。しかしここでは別のコードを使う。はじめは親要素にクラスを付与し、ボタンとメニューのスタイルはそのクラスを使って行うと考えるだろう。

親要素にクラスを付与し子要素をスタイルする

```
<div id="content">
  <div class="toolbar is-active">
    <button id="btn-new" class="btn"
      data-action="menu">New</button>
    <div id="menu-new" class="menu">
      <ul> ... </ul>
    </div>
  </div>
</div>

/* CSS */

.is-active .btn { color: #000; }
.is-active .menu { display: block; }
```

このアプローチの問題点はHTMLの構造が同じである必要があることだ。まずコンテナとなる要素が必須となり、メニューもボタンもそのコンテナの中になければならない。ツールバーに対して追加のボタンが必要ないことが前提となってしまう。

もう1つのアプローチとしては前の例と同じく活性化用のクラスをボタンに追加し、兄弟セレクトを使ってメニューを選択された状態にする方法がある。

兄弟セレクトを使ってメニューを選択状態にする

```
<div id="content">
  <div class="toolbar">
    <button id="btn-new" class="btn is-active"
      data-action="menu">New</button>
    <div id="menu-new" class="menu">
      <ul> ... </ul>
    </div>
  </div>
</div>

/* CSS */

.btn.is-active { color: #000; }
.btn.is-active + .menu { display: block; }
```

私は親要素に対して状態クラスを付与するアプローチよりこの方法のほうが、状態をモジュールに対してより正確に適応にできるので好ましいと考えている。しかしそれでもボタンのHTMLとメニューのHTMLとは依存関係がある。それぞれは常に並んで配置されている必要がある。もしプロジェクト内でこの一貫性を保てるのであれば、このアプローチは有効だろう。

なぜ親要素と兄弟要素の状態変更のアプローチは問題となり得るのか

それぞれのモジュールに対して状態を付与するのに比べてこのアプローチがなぜ問題となるかという、このルールセットをどこに記述するべきかが曖昧になるからだ。メニューは単なるメニューではなくボタンメニューとなってしまう。もしこのモジュールの選択された状態を変更したいと考えたとき、ボタンCSSのそばに変更すべきCSSがあるのか、メニューCSSのそばにあるのか明確ではない。

このことからそれぞれのボタンに対して状態を付与する方が好ましいアプローチといえる。モジュール単位をよりわかりやすく分離するほうが、サイトを制作し、テストし、拡張しやすくなる。

属性セレクトアを使った状態の変更

どのブラウザをサポートするかによっては、状態の変更を属性セレクトアを使って扱うこともできる。こうすることで以下のような利点が考えられる:

- ・ レイアウト、モジュールのクラスから状態を切り離すことができる
- ・ 複数の状態の変化を扱うのが楽になる

デフォルト、クリックされた状態、非活性の状態という複数の状態になり得るボタンを例に見てみよう。

以下はサブモジュールの命名規則を使った例となる。

サブモジュールの命名規則

```
.btn { color: #333; }  
.btn-pressed { color: #000; }  
.btn-disabled { opacity: .5; pointer-events: none; }
```

ボタンに切り替えの状態がある場合は状態の命名規則を使う方がわかりやすい。

状態の命名規則

```
.btn { color: #333; }  
.is-pressed { color: #000; }  
.is-disabled { opacity: .5; pointer-events: none; }
```

この2つの例はSMACSSがほとんどの場合に命名規則と明晰さについて書かれているものであることを明示するよい例だ。どちらの例を使っても全く問題はない。それでは属性セレクトアを使ったアプローチについて見てみよう。

属性セレクタの例

```
.btn[data-state=default] { color: #333; }
.btn[data-state=pressed] { color: #000; }
.btn[data-state=disabled] { opacity: .5;
pointer-events: none; }

<!-- HTML -->
<button class="btn"
data-state="disabled">Disabled</button>
```

属性のdata-プリフィックスはHTML5の仕様の一部で属性を自由に追加することができ、さらに将来的なHTMLの属性に関わる仕様と衝突しない。ボタンの状態を変更するのにクラスの削除や追加は必要なく、1つの属性の値を変更すれば事足りる。

jQueryを使って状態を変更する

```
// それぞれのボタンに対しクリックハンドラを関連づける
$(".btn").bind("click", function(){
    // クリックした状態に変更
    $(this).attr('data-state', 'pressed');
});
```

jQueryのようなJavaScriptライブラリを使えば状態の管理のためにクラスを変更することは決して難しくはない。jQueryにはhasClass、addClass、そしてtoggleClassというメソッドがありクラス名を編集することは、非常に簡単にできる。

言うまでもなく状態を表現するには様々な選択肢がある。

CSSアニメーションとクラスベースの状態変更

アニメーションは非常に興味深い存在で、振る舞いを定義するべきではないレイヤーで定義してしまっているという議論もある。CSSはスタイルを行うためにあり、JavaScriptこそ振る舞いを定義するべきだということだ。

理解しておくべき違いはCSSが見た目上の状態を定義するのに使われているということだ。JavaScriptを使って要素の状態を切り替えることはできるが、JavaScriptは状態を説明するのに使うべきではない。つまりJavaScriptを使ってインラインのスタイルを追加するべきではないともいえる。

これまでJavaScriptを使ってアニメーションを行っていたのは、その方法しかなかったからだ。(HTML+TIME²にあるにも関わらず。)

このようにして考えてみると様々なシチュエーションに対するアプローチが見えてくる。例えばページ上でメッセージが短い間のみ表示され、フェードアウトするということが不自然ではなくなるわけだ。

状態の変化をJavaScriptで扱う

```
function showMessage (s) {
  var el = document.getElementById('message');
  el.innerHTML = s;

  /* set state */
  el.className = 'is-visible';
  setTimeout(function(){
    /* set state back */
    el.className = 'is-hidden';
  }, 3000);
}
```

メッセージの状態は隠れた状態から表示の状態に変更し、また隠れた状態に戻る。JavaScriptをこれらの状態を変更するのに使い、CSSを使ってこれらの間にアニメーション(CSSトランジションかアニメーションを使って)を追加する。

2.<http://www.w3.org/TR/NOTE-HTMLplusTIME>

トランジションをCSSで扱う

```
@keyframes fade {
  0% { opacity:0; display:block; }
  100% { opacity:1; display:block; }
}

.is-visible {
  display: block;
  animation: fade 2s;
}

.is-hidden {
  display: none;
  animation: fade 2s reverse;
}
```

この例は残念ながら実際には動作しない。現在のブラウザの実装ではアニメーションしないプロパティがあるからだ。しかしブラウザ実装はCSS3アニメーションのレコメンデーションに対してアップデートしている最中だ。そのアップデートを待つ間、以下のように対応する必要がある。

現状のブラウザでアニメーションを行う

```
@-webkit-keyframes fade {
  0% { opacity:0; }
  100% { opacity:1; display:block; }
}

.is-visible {
  opacity: 1;
  animation: fade 2s;
}

.is-hidden {
  opacity: 0;
  animation: fade 2s reverse;
}

.is-removed {
  display: none;
}

/* JavaScriptを追加 */
function showMessage (s) {
  var el = document.getElementById('message');
  el.innerHTML = s;

  /* set state */
  el.className = 'is-visible';
  setTimeout(function(){
    /* set state back */
    el.className = 'is-hidden';
    setTimeout(function(){
      el.className = 'is-removed';
    }, 2000);
  }, 3000);
}
```

この例ではアニメーションは行うがJavaScriptを使ってアニメーションが終わるタイミングで要素を削除している。

この方法だとスタイル(状態)と振る舞いの分離を保つことができる。

疑似クラスを使っての変更

これまで見てきたように、クラスと属性を使ってモジュールの状態の変更を扱うことができる。しかしCSSでは様々な疑似クラスを使って状態とその変化を管理することもできる。

CSS2.1の仕様の中で最も便利な疑似クラスはユーザのインタラクションに反応する`:hover`、`:focus`、そして`:active`のような"動的な"疑似クラスだ。CSS3にはこれに加えて新たな疑似クラスがある。多くはHTMLの構造を元にスタイルできるものだ(例えば`:nth-child`や`:last-child`)。またフォームのインタラクションに反応するようなUIに関する疑似クラスもCSS3で追加されていて、使い方によっては非常に便利だろう。

モジュールのデフォルトの状態は疑似要素なしで定義し、モジュールの2次的な状態を定義するのに疑似クラスを用いる。

疑似クラスを使って状態を定義する

```
.btn {
  background-color: #333; /* グレー */
}

.btn:hover {
  background-color: #336; /* 青っぽい色 */
}

.btn:focus {
  /* 青っぽいフォーカスリング */
  box-shadow: 0 0 3px rgba(48,48,96,.3);
}
```

モジュールがサブクラス化されるとサブモジュール用の疑似クラスの状態もデザインする必要があるため複雑になる可能性がある。

疑似クラスを使ってサブモジュールの状態を定義する

```
.btn {
  background-color: #333; /* グレー */
}

.btn:hover {
  background-color: #336; /* 青っぽい色 */
}

.btn:focus {
  /* 青っぽいフォーカスリング */
  box-shadow: 0 0 3px rgba(48,48,96,.3);
}

/* デフォルトのボタンの状態を
 * ボタンの中から選ぶ
 */
.btn-default {
  background-color: #DEDB12; /* 黄色っぽい色 */
}

.btn-default:hover {
  background-color: #B8B50B; /* 暗い黄色 */
}

/* フォーカスの状態は変更なし */
```

この例では1つのモジュールから5つのバリエーションを作ったことになる: メインモジュール、サブモジュール、そしてそれぞれのモジュールで利用する疑似クラスの状態。これに対してクラスベースの状態を追加する場合はさらに複雑になる。

モジュール、サブモジュール、クラスの状態、疑似クラスの状態。

```
.btn { ... }  
.btn:hover { ... }  
.btn:focus { ... }  
  
.btn-default { ... }  
.btn-default:hover { ... }  
  
.btn.is-pressed { ... }  
.btn.is-pressed:hover { ... }  
  
.btn-default.is-pressed { ... }  
.btn-default.is-pressed:hover { ... }
```

大抵の場合はこれほど状態管理が必要となるインターフェイスは必要とされないだろう。しかしスタイルを適切な構成にしておくことがプロジェクトをより簡単にメンテナンスするということが明確になっただろう。

メディアクエリを使った変更

クラスや疑似クラスを使った状態の変更は一般的と言えるだろう。一方でメディアクエリもこれまでJavaScriptを使ってでしか出来なかった状態の管理に別のアプローチとしてよく使われ始めている。アダプティブデザインやレスポンシブウェブデザイン³はメディアクエリを使って様々な基準に反応する。プリントスタイルシートはメディアクエリの初期の例であり、ページをプリントアウトする際にどう要素が見えるべきかを定義するのに利用できる。

メディアクエリはlink要素に対してメディア属性を使うことで別のスタイルシートとして定義することもできるし、@mediaのブロック内で特定のスタイルシート内で定義することもできる。

3.<http://www.alistapart.com/articles/responsive-web-design/>

スタイルシートのリンク

```
<link href="main.css" rel="stylesheet">
<link href="print.css" rel="stylesheet"
media="print">

/* main.css 内*/
@media screen and (max-width: 400px) {
    #content { float: none; }
}
```

ほとんどの場合、ブレイクポイントを定義し、それに付随するスタイルを特定のブレイクポイント内でのみ適応するというのがメディアクエリの例としてあげられている。

SMACSSでは特定のモジュールに付随するスタイルをほかのモジュールのそれと分離するのを目的としている。どういうことかというと、メインのCSSファイル内であろうと、別のメディアクエリスタイルシート内であろうと、1つのブレイクポイントを持つのではなく、モジュールステートの近くにメディアクエリを利用するということになる。

モジュラーメディアクエリ

```
/* navアイテムのデフォルト状態 */
.nav > li {
    float: left;
}

/* 小さなスクリーンでのnavアイテムの状態 */
@media screen and (max-width: 400px) {
    .nav > li { float: none; }
}

... レイアウト内のどこか ...

/* デフォルトレイアウト */
.content {
    float: left;
    width: 75%;
}

.sidebar {
    float: right;
    width: 25%;
}

/* 小さなスクリーンでのレイアウトの状態 */
@media screen and (max-width: 400px) {
    .content, .sidebar {
        float: none;
        width: auto;
    }
}
```

メディアクエリの宣言はおそらく(ほとんどの場合)何度も記述することになる。しかしこうすることでモジュールについてのすべての情報を1カ所で管理できる。モジュールについての情報を集中管理できる(特に1つのCSSファイル内で)ことはモジュールのテストを分離することを可能にし、(アプリケーションの制作の方法によっては)モジュラー化されたテンプレートとCSSを初期のページ読み込みの後でも呼び出すことができるようになる。

すべては状態だ

この章では3つの種類の状態の変更について紹介した: クラス、疑似クラス、そしてメディアクエリだ。インターフェイスのモジュール性について考慮するだけではなく、様々な状態でそれらのモジュールがどのような表現になるのかも考慮することでスタイルを適正に分割でき、そうすることでメンテナンス性も向上するのだ。

適応性の深度

CSSが実際にどのように動作するのかを学ぶと、CSSにはセレクタがあり、そのセレクタを使ってページ上のスタイルを適応させたいHTML要素を選択することを学ぶ。CSSは時を経て成長し、より多くのセレクタを使ってより強力なことができるようになってきた。しかしスタイルシートにルールセットを追加するたびに、HTMLとCSSとの関連性を深めてしまうことになる。

ウェブサイトによくあるCSSのブロックを見ていこう。

HTMLとCSSが密接に関連している例

```
#sidebar div {
  border: 1px solid #333;
}

#sidebar div h3 {
  margin-top: 5px;
}

#sidebar div ul {
  margin-bottom: 5px;
}
```

この例を見てわかるように、HTMLがどのような構成になるべきかがある程度確定してしまっている。ここではサイドバーには最低1つの見出しとリストを持ったセクションがあると想定している。このサイトがあまり変更されない場合には、このCSSでも何ら問題がない。私の場合だとこの2年ブログのデザインを変更していないので、私自身の規模拡張性についてはニーズがそれほどはない。もしこのアプローチを変更頻度が高く、より複雑な要求を満たさなければならないような大規模サイトで適応したら、確実に問題となるだろう。より多くのルールをより複雑なセレクタを使って追加すること

になり、メンテナンスの悪夢にうなされてしまうのが目に見えている。

いったいどこに問題があるのだろうか？この例のCSSには2つの懸念がある：

1. 定義されたHTML構造への依存性が高い
2. セレクタが適応されるHTMLの深度が深すぎる

深度を最小限にとどめる

深度が深くなるということは、特定のHTML構造への依存を強制するという問題点がある。そうなるとページ上のコンポーネントを自由に動かすことができなくなってしまう。先ほどのサイドバーの例に戻ってみて欲しい。このモジュールをどうやったらフッターなどのページの別の場所で再現できるだろうか？このままでは確実にルールを重複させるしかなくなってしまう。

ルールの重複

```
#sidebar div, #footer div {
  border: 1px solid #333;
}

#sidebar div h3, #footer div h3 {
  margin-top: 5px;
}

#sidebar div ul, #footer div ul {
  margin-bottom: 5px;
}
```

ノードのルートは

にあるのでここからスタイルを作成し始めるべきだろう。

ルールの単純化

```
.pod {
  border: 1px solid #333;
}

.pod > h3 {
  margin-top: 5px;
}

.pod > ul {
  margin-bottom: 5px;
}
```

このpodはコンテナであり、いまだに特定のHTMLの構造に依存してはいるが、以前に比べると深度は圧倒的に浅くなっている。その代わりに先ほどの2つのIDが付与された要素のみだった例と比べてこの場合はpodクラスをいくつもの要素に対して割り当てる必要がある。当然のことながらすべてのパラグラフに対してクラス名を付与するような少し前の時代に戻ることは避けたい。

浅い適応深度のアプローチをすることの利点はこれらのモジュールを容易に動的なコンテンツ用のテンプレートに変更することができることだ。Yahoo!の例ではテンプレートのほとんどをMustacheに依存している。以下はこの例のpodをテンプレートに変換した場合の例だ:

Mustacheテンプレートの例

```
<div class="pod">
  <h3>{{heading}}</h3>
  <ul>
    {{#items}}
    <li>{{item}}</li>
    {{/items}}
  </ul>
</div>
```

私たちはメンテナンス性、パフォーマンス、そしてソースの読みやすさでうまくバランスをとるようにしている。深度を深くするとい

うことはHTMLが『クラスだらけ』の状態にはならないということになるが、同時にメンテナンス性を低くし、読みづらさも増す。もちろん、すべてに対してクラスを付けたくもない(付ける必要もない)。例のようにh3やulにクラスを付与するのはより柔軟なシステムが必要でない場合を除いてあまり必要ではない。

最後の例をより詳しく見てみると、このデザインパターンが非常によくあるものであることに気がつくだろう。このパターンは見出しとボディがあるコンテナである(これにフッターが追加されることもある)。現時点ではulが使われているが、可能性としてはolであることも、divであることも考えられる。

重ねて、それぞれのバリエーションに対してルールを重複することもできる。

ルールの重複

```
.pod > ul, .pod > ol, .pod > div {
  margin-bottom: 5px;
}
```

代替手段としてpodのボディにクラスを付けることもできる。

クラスを使って単純化する

```
.pod-body {
  margin-bottom: 5px;
}
```

モジュールルールのアプローチに従えば、.podクラスを指定する必要はない。.pod-bodyのクラス名だけで十分にpodモジュールとの関連を可視化できるし、コードの観点だけで言えば問題なく動作する。

Mustacheテンプレートの例

```
<div class="pod">
  <h3>{{heading}}</h3>
  <ul class="pod-body">
    {{#items}}
    <li>{{item}}</li>
    {{/items}}
  </ul>
</div>
```

この小さな変更の結果、適応性の深度をもっとも浅い状態に減らすことができた。1つのセレクタを使っているということは、起こりえる詳細度の問題も回避している。あらゆる点でwin-winだ。

セレクトパフォーマンス

仕事としてこれまでパフォーマンスに対して数多くのテストを行ってきた。アプリケーションに対していくつものツールを使いどこがボトルネックになるのかを明確にしてきた。そのツールのうちの1つがGoogle Page Speed⁴で、このツールではJavaScriptやレンダリングのパフォーマンスを向上するための推奨設定を知らせてくれる。これらの推奨を知る前に、ブラウザがどのようにCSSを評価するのかについて知る必要がある。

CSSはどのように評価されるのか

要素に対するスタイルは要素が生成されるタイミングで評価される

往々にしてページについて完全にコンテンツや要素が入ったドキュメントとして認識しているが、ブラウザはドキュメントを一連の流れのように捕らえるように作られている。ドキュメントをサーバーから取得するところから始まって、ドキュメントが完全にダウンロードし終える前でもレンダリングは開始する。それぞれのノードはサーバーから取得しながらビューポート内で評価され、レンダリングされる。

4. <http://code.google.com/speed/page-speed/>

HTMLドキュメントの例

```
<body>
  <div id="content">
    <div class="module intro">
      <p>Lorem Ipsum</p>
    </div>
    <div class="module">
      <p>Lorem Ipsum</p>
      <p>Lorem Ipsum</p>
      <p>Lorem Ipsum <span>Test</span></p>
    </div>
  </div>
</body>
```

ブラウザは1番上からドキュメントを評価し、まず`body`を探し出す。この時、ブラウザは`body`が空であると認識する。それ以外は何も評価されていない。そしてブラウザは何が算出済みのスタイルなのかを確定させ、要素に対して適応する。フォントは何か、色は何か、または行間は? これらを算出し終わると、スクリーンに描画を行う。

次に`contents`というIDが付与された`div`を探し出す。ここでも`div`は空だと認識し、ほかに何も評価されていない。ブラウザはスタイルを算出し`div`が描画される。ブラウザは`body`に再描画が必要かをどうかを確定する。要素の幅は変更されたか、高さは変更されたかなどを見る。(ほかにも評価のポイントはあるが幅と高さは子要素が親要素に対して影響を及ぼす最も一般的な例だろう)

このプロセスがドキュメントの最後にたどり着くまで続く。

Firefoxでの再フロー/再描画のプロセスを視覚化した動画を <http://youtu.be/ZTnIxIA5KGw>で見ることができる。

CSSは右から左に評価される

特定の要素に対してCSSのルールが適応されるかを確定するために、一番右のルールから評価を始め、左端まで行う。

例えば `body div#content p { color: #003366; }` のようなルールの場合、ページ内でレンダリングされながらすべての要素に対して、その要素が段落要素であるかを確認する。仮に段落要素だとしたら、DOMの最上部に向かってコンテンツというIDが付与された `div` を探す。その要素が見つかったら、またDOMの最上部に向かって探索を始め、`body` が見つかるまで続く。

このように右から左に向かって評価を行うことでブラウザはそのルールが特定の要素に対して適応するかを確定し、ビューポートへのレンダリングを早くしている。どのルールなのかを確定するのは問題ではなく、いくつくらい `node` に対してスタイルが適応できるかを確定する必要があるのかが問題になる。

どのルールが優先されるのか？

ページ上にそれぞれの要素がレンダリングされつつ、どのスタイルを適応するかを確定する必要がある。少しだけ Google Page Speed の推奨設定⁵を見てほしい。そこにはパフォーマンスがよくないとされる4つの大きなルールがある。

- ・ 子孫セクタのルール。例: `#content h3`
- ・ 子セクタまたは隣接セクタのルール。例: `#content > h3`
- ・ 必要以上のセクターがあるルール。例: `div#content > h3`
- ・ リンクではない要素に対する `:hover` を利用したルール。例: `div#content:hover`

これらの推奨の中で最も大事なポイントは、スタイルを確定するのに1つの要素以上を評価することは非効率であるということだ。端的に言ってしまうと、ルールには1つのセクタしか使うことが出来ないとも言える。クラスセクタ、IDセクタ、要素セクタ、あるいは属性セクタがそうだ。この推奨を額面通りに受け止めると、`<p class="bodytext">` というようなコーディングをしていた時

5. <http://code.google.com/speed/page-speed/docs/rendering.html#UseEfficientCSSSelectors>

代に戻れと提案しているのと同じになる。(Google検索やGmailのCSSを実際に見てみるとこの推奨通りにコーディングしてある。)

制約を設けるのはいいが、自分を苦しめるのはよくない

実際にはもう少し実践的にすべてに対してクラスを付けることと、HTMLとCSSの間を強い関係性を作ってしまう深いセレクトタの利用との間のバランスを見つけることが大切だ。

私自身は以下の3つのシンプルなガイドラインに沿って評価されるべき要素を限定している:

1. 子セレクトタを使う
2. 一般的な要素に対するタグセレクトタを使わない
3. クラス名を右端のセレクトタとして使う

例を挙げると `.module h3` はページ内に十数個ほどしかのH3が存在しないのであればおそらく問題にはならない。H3はDOMの中でどれくらい深い場所にあるのか? 4段階下なのか?(例: `html > body > #content > h3`)、それとも10段階(例: `html > body > #content > div > div > ... > h3`)? DOMの探索を子セレクトタを使って限定できるか? もし `.module > h3` (IE6、ごめんなさい)とできるのであれば、もし12個のH3がページ内にあるとしたら、24の要素だけ評価されればよいことになる。もし `.module div` とした場合、ページには900のdivがある(Yahoo! MailのInboxページを開いたら903あった)のもっとたくさんのDOM探索が必要になる。3段階、`<div><div><div></div></div></div>` のような単純な場合でも6回の評価が必要となる。階乗で計算できるので、4段階であれば24回、5段階であれば120回の評価を行う必要がある。

以上のことを踏まえた上で、この単純な最適化はやり過ぎとも言える。パフォーマンスについてのテストをずっと行っているSteve Sounders氏の実験(2009年に行った)によれば、ベストケースとワーストケースの差分は50msだった。言い換えると、セレクトタのパフォーマンスについては考慮するべきではあるが、多くの時間を費やすべきではない。

HTML5とSMACSS

SMACSSはHTML5であろうと、HTML4であろうと、それ以外のHTMLであろうと同じように相性がいい。これはSMACSSアプローチには2つのコアとなるゴールがあるからだ:

1. HTMLとコンテンツのセマンティックな価値を向上すること
2. 特定のHTML構造への依存を低減すること

HTML5はいくつものHTMLとコンテンツのセマンティックな価値を高める要素を追加している。`section`、`header`、そして`aside`のようなタグは`div`よりも限定用法となっている。さらに新たな`input`の値は数値、日付とテキストフィールドとの区別を行うことを可能にしている。追加となったタグも属性もより用法が限定されていて、それはよいことだと言える。

しかし新しいタグが追加されたとしても、タグそのものだけではページ上の特定のモジュールを説明するのには不十分だ。`nav`要素は常に全く同じ種類で同じスタイルのナビゲーションであり続けるだろうか？

<nav> の実装

```
<nav class="nav-primary">
  <h1>メインナビゲーション</h1>
  <ul>...</ul>
</nav>

<nav class="nav-secondary">
  <h1>外部リンク</h1>
  <ul>...</ul>
</nav>
```

メインナビゲーションはページの横並びに配置されるが、2次的なナビゲーション(例: サイドバーなど用)はアイテムを縦に並べる。ここではクラス名がそれぞれの種類を区別している。

クラス名はコンテンツをより詳細に説明する手助けをする。それはHTML5で供給されるものよりもより詳細だ。これこそが、1つめのゴールであるHTMLのセクションのセマンティック性を向上させる。

直感的に以下のようなコードを記述するだろう:

```
<nav> のCSS

nav.nav-primary li {
    display: inline-block;
}

nav.nav-secondary li {
    display: block;
}
```

こう記述することでこれらのクラスはnav要素でしか使われないことを意味することになる。もしコードが絶対に変わらないとしたら問題にはならないだろう。しかし、本書の目的が拡張性のあるプロジェクトについてなので、プロジェクト内でどう物事が変化していくかの例を見ていこう。

メインナビゲーションは1段階しかない。しかしクライアントがやってきて、すべての要素にドロップダウンを追加する必要があると追加で希望を出してくる。HTMLの構造は変更しなければならない。

<nav>の実装

```
<nav class="nav-primary">
  <h1>メインナビゲーション</h1>
  <ul>
    <li>アバウト
      <ul>
        <li>チーム</li>
        <li>所在地</li>
      </ul>
    </li>
  </ul>
</nav>
```

このサブとなるナビゲーションでどのようにアイテムを横ではなく縦に並ぶようにスタイルするべきだろうか？

元々あったCSSを考えると、`<nav class="nav-secondary">`をすべてのリスト内に記述しなければならなくなる。

CSSを拡張し内部のリストアイテムをターゲットにすることも可能だ。

<nav>CSSの拡張

```
nav.nav-primary li {
  display: inline-block;
}

nav.nav-secondary li,
nav.nav-primary li li {
  display: block;
}
```

ほかの方法としては`nav`要素のみで利用できるクラスを利用するのをやめることだ。これが2つ目のゴールである、特定のHTML構造への依存性を減らすことにつながる。

SMACSSスタイルの<nav>CSS

```
.l-inline li {
    display: inline-block;
}

.l-stacked li {
    display: block;
}
```

このケースではレイアウトルールを利用することを意味するように変更した。なぜなら個別のモジュール(リストアイテム)に対してどのように内包されるべきかについて影響を与えていたためだ。`.l-stacked`クラスをサブナビゲーションである

に付与することができ、こうすることで求めている結果を得ることができる。

リストアイテムを子要素として必須条件とするとレイアウトルールはまだ特定のHTML要素に依存していることにはなる。ことわざにもあるように、何かを成し遂げる方法は様々だ。例えば以下のようにすべての子要素がそのスタイルを引き受けるようにしたいかもしれない。

SMACSSスタイルの<nav>CSS

```
.l-inline > * {
    display: inline-block;
}

.l-stacked > * {
    display: block;
}
```

このアプローチの弱点はこのルールでは、このリストアイテムだけではなくすべての要素が評価されなければならない点にあるが、直接の子孫のみをターゲットにすることで探索を最小限には止めている。こうすることでこのスタイルを子要素に利用したい場合に、`inline`と`stacked`クラスを付与するだけでよくなる。

<nav>の実装

```
<nav class="l-inline">
  <h1>メインナビゲーション</h1>
  <ul>
    <li>アバウト
      <ul class="l-stacked">
        <li>チーム</li>
        <li>所在地</li>
      </ul>
    </li>
  </ul>
</nav>
```

この単純な例でもCSSを単純にしセクタを複雑にしなくてすむようにできた。HTMLも理解しやすいままだ。

2つのゴールを覚えておいて欲しい: セマンティック性を向上し、特定のHTML構造への依存性を減らすこと。

プロトタイプ

よいプログラマはパターンを好み、よいデザイナーもまたパターンを好む。パターンは親しみやすさを作りだし、再利用を促す。SMACSSはデザインからパターンを見だし、コード化することについて紹介している本だ。

プロトタイプはコンポーネント全体、あるいは一部を可視化し、デザイン言語をコード化し、ブロックを構築することを補助するものだ。ウェブデザインの世界は再利用できるコンポーネントを好み、いくつものBootstrap⁶(様々なサイトのコンポーネント)、または960.gs⁷(レイアウトグリッド)のようなフレームワークにも見て取ることができる。

Yahoo!ではプロトタイプチームがブロックを構築し、プロダクションで利用している。全く同じ基礎からブロックを構築するため、複数のプロダクトでも一貫性は保たれることになる。

プロトタイプのゴール

プロトタイプにはいくつかのゴールがある:

- ・ 状態を表示する
- ・ ローカライズのレビュー
- ・ 依存関係の分離

状態

デフォルト状態、折り畳みの状態、そしてエラーの状態などの定義したどんな状態も、それぞれの状態を可視化することが大切なこと

6.<http://twitter.github.com/bootstrap/>

7.<http://960.gs/>

であり、モジュールがそれに従って正確に作られていることを確認する必要がある。

もしモジュールがデータを元にしたものであれば、実際のデータか、偽のデータをプロトタイプで利用し、正しくレンダリングされるかをプロトタイプで確認する必要がある。

ローカリゼーション

プロジェクトが複数の言語をサポートする必要がある場合、モジュールを実際に異なる言語の文字列を使ってテストしレイアウトが崩れないことを確認できることは非常に有意義だろう。

依存関係

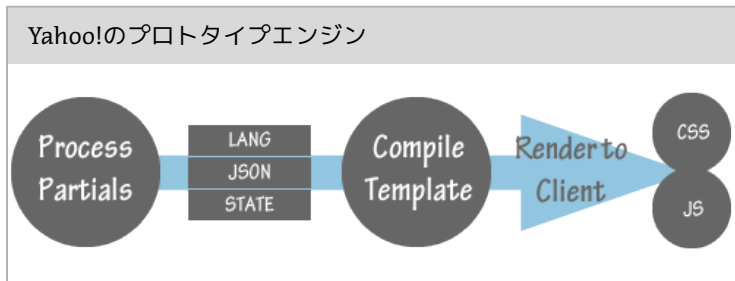
最後に依存関係を分離することも非常に大事なことだ。どのCSSとJavaScriptの依存関係がモジュールを正しく表示するのに必要なのか。大規模なプロジェクトでコンポーネントの後読みがされるような場合、依存関係を分離し、必要最小限に抑えるということは、モジュールを効果的に作成したことと同じ意味となり、そのモジュール群はサイトのどこに配置しようとほかのモジュールに対してマイナスになる影響を与えないという意味にもなる。

Yahoo!ではモジュールは個別のCSSファイルとして分離され、結合ハンドラを使って必要な際に同梱される。例えば受信箱が読み込まれると、ボタン、メッセージリスト、サイドバー、タブ、ヘッダーのCSSファイルが結合される。ユーザが検索ページに行くと、結合ハンドラは検索特有のスタイルを結合する。検索ではデフォルトメッセージリストとサイドバーのバリエーションを利用しているので、サブクラスモジュールのみを呼び出せばいい。

パズルのピース

Yahoo!ではプロトタイプエンジンを作成しこのプロセスを容易にしている。同様の機能が必要かどうかはプロジェクトの規模によるだろう。

プロトタイプエンジンはmustache template⁸を使っている。偽のデータはJSONファイルとして保存しており、ローカリゼーション用の文字列はキー/値のペアをテキストファイルに保存してある。そして、CSSとJavaScriptの依存関係は必要な場合に呼び出される。こうすることでチームでメニューやダイアログ、フォームを個別で見ることできるし、サイト全体の一部として見ることもできる。全員が機能やデザインを事前に確認することができるわけだ。これらのアセットをエンジニア側に転換し、連携をシームレスにすることもできる。



我々のプロトタイプエンジンのケースだと、いくつかの状態の管理についてはモジュールがレンダリングされる前に行われる。通常であればサーバサイドでのプロセスが必要になる条件が必要になる要素やデータのフィルタリングなどを管理している。状態の管理はHTML要素に対してクラス名を追加するというだけではないのだ。

自分のプロトタイプ

特に小さなサイトではモジュールを生成するのに完成されたエンジンは必要ないだろう。しかしコンポーネントを分離して簡単にレビューできるフォーマットを持つておくことはやはり便利だ。例えばMailChimpではサイトを作るのに利用しているデザインパターンのチートシートを内部で持っている⁹。このドキュメントではチームでサイト内で利用される様々なモジュールを表示し、それぞれのコードも見ることができる。

8. <http://mustache.github.com/>

9. <http://www.flickr.com/photos/aaronwalter/5579386649/>

覚えていて欲しいことはパターンはすばらしいということだ。それらのパターンをコード化することはもまたすばらしいことで、それらのパターンをレビューしテストできるプロセスがあることはさらにすばらしいことだ。

プリプロセッサ

CSSは確かに素晴らしいが、まだ多くのデザイナー、デベロッパが欲しいと思っている機能が欠けている部分がある。この隙間を埋めるために、そして開発効率を上げるために、様々なツールが公開されている。

そのうちの1つがCSSプリプロセッサだ。ここではプリプロセッサとは何か、何ができるのか、そして拡張性の高いモジュラーCSSを書くためにどう役に立つのかについて紹介していく。

プリプロセッサとは?

CSSプリプロセッサは特別な文法を追加し、それらの文法をCSSに変換できるツールだ。CSSの文法に限りなく近い形で文法が追加しているプリプロセッサもあれば、単純化に比重を置いているものもある。

Stylus¹⁰プリプロセッサの例を見て欲しい。

Stylusを使った例

```
@import 'vendor'

body
  font 12px Helvetica, Arial, sans-serif

a.button
  border-radius 5px
```

10.<http://learnboost.github.com/stylus/>

Ruby¹¹やCoffeeScript¹²を知っていれば、見たことがあるような文法だろう。中括弧とセミコロンを省略する代わりに、ホワイトスペースに依存している。インデントによってどのプロパティがどのルールに付与されるのかを定義している。CSSではプロパティ名にスペースが入ることがないため、プロパティ名の後に続くスペースがプロパティと値の区切りになっている。

Stylusに加えて、Sass¹³(Compass¹⁴)、Less¹⁵の2つがほかによく使われているプリプロセッサだ。

プリプロセッサのインストール

特にコマンドラインにそれほど詳しくないデザイナーから不満があがるのが、これらのプリプロセッサをインストールするのがややこしくなり得ることだ。環境によってはそうかもしれないし、場合によってはアプリケーションをアプリケーションフォルダにドラッグするだけのものもある。

例えば、MacでSassをインストールするにはコマンドラインツールを開いて以下のコマンドを実行するだけで済む:

Sassのインストール

```
sudo gem install sass
```

またはCompassをインストール

```
sudo gem install compass
```

Gem¹⁶はRubyのパッケージマネージャコマンドラインツールで、アプリケーションをインストールするのに使う。そしてMac OS Xの新しいバージョンであればインストールされている。

11.<http://www.ruby-lang.org/en/>

12.<http://coffeescript.org/>

13.<http://sass-lang.com/>

14.<http://compass-style.org/>

15.<http://lesscss.org/>

16.<http://rubygems.org/>

Sassのインストールが終わったら、作業フォルダをSassで監視するように設定する。

Sassの実行

```
sass --watch before:after
```

beforeには.scss(CSSに変換する前のファイル)を格納したフォルダを指定し、afterには変換後のCSSを格納するフォルダを指定する。このコマンドでbeforeフォルダの.scssファイルに対する変更は自動で、.cssに変換され、afterフォルダに格納されていく。開発、そしてテストを行うのに便利なコマンドだ。(.scssは普通の.cssと同じようなものだがSassの文法を使っている。これについては後述する)

Compass Mac app¹⁷というツールもあり、コマンドラインを使わなくてもSassを利用することが可能だ。

Lessはnpm、node package manager¹⁸を利用してインストールできる。npmはデフォルトでインストールされていないので、npmをインストールし終わらないとLessをインストールできない。Lessにはクライアントサイドで使えるJavaScriptバージョンもあるので、開発に利用できる。

Lessのクライアントサイド実装

```
<link rel="stylesheet/less" type="text/css"
href="styles.less">
<script src="less.js"></script>
```

注意して欲しいのはこの状態で公開せずCSSに変換してから公開すること。

17.<http://compass.handlino.com/>

18.<https://github.com/isaacs/npm>

LessをコマンドラインでCSSに変換する

```
lessc styles.less
```

サイトを開発するのにコマンドラインを利用することは増えてきている。GUIでは解決できない問題を排除してくれる。

プリプロセッサの便利な機能

プリプロセッサにはCSSの開発を楽にする様々な興味深い機能がある。例をいくつか挙げると：

- ・ 変数
- ・ 演算
- ・ ミックスイン
- ・ 入れ子ルール
- ・ 関数
- ・ 補間
- ・ ファイルインポート
- ・ 拡張

これらはいったいどういう意味なのか、いくつかを詳しく見ていこう。(以下の例ではSassを利用するが、LessやStylusにも同じコンセプトに対して似たアプローチがある)

変数

1時間以上CSSを編集したことがある人なら、誰でもCSSファイルに色の値を設定し、その値を使って色の指定をどこでも利用できたらと思うことだろう。Sassでは\$というプリフィックスを使って変数を定義することができる。

変数の利用

```
$color: #369;

body {
  color: $color;
}

.callout {
  border-color: $color;
}
```

コンパイラはこの指定を以下のように書き換える。

変数の変換

```
body {
  color: #369;
}

.callout {
  border-color: #369;
}
```

サイト全体への変換を1箇所で行える非常に便利な機能だ。(ちなみにW3CもCSS Variables¹⁹の仕様ドラフトを進めている。)

入れ子ルール

CSSをコーディングしていると、セレクタが連鎖状態になることはよくあることだ。

19. <http://dev.w3.org/csswg/css-variables/>

セレクタの連鎖

```
.nav > li {
  float: left;
}

.nav > li > a {
  display: block;
}
```

入れ子ルールを使えばこれらのスタイルをグループ化してわかりやすく記述できるようになる。

Sassの入れ子ルール

```
.nav {
  > li {
    float: left;
    > a {
      display: block;
    }
  }
}
```

それぞれのスタイルは自分自身の親となるセレクタの中に記述される。上記の例は下記のようにCSSに変換される。

SassからCSSに変換

```
.nav > li {
  float: left; }
.nav > li > a {
  display: block; }
```

入れ子ルールを使うことでどのスタイルがどの要素と関連があるのかが明確になる。しかし、自分でインデントを付けるのとあまり変わらない。`.nav`を毎回タイプしなくてもよくなる。

ミックスイン

ミックスインは非常に強力な機能だ。ミックスインとはグループ化したスタイルを再利用するための機能で、引数を使ってミックスインの出力結果を変更することもできる。よく使われるミックスインの例がベンダープリフィックスを省略する場合だ。(もちろん繰り返しになるCSSであればどこでも使える。)

border-radiusのミックスイン例

```
@mixin border-radius($size) {
  -webkit-border-radius: $size;
  -moz-border-radius: $size;
  border-radius: $size;
}
```

ミックスインを定義したら、CSS内のどこでも**include**文を使って呼び出すことができる。

border-radiusミックスインの利用例:

```
.callout {
  @include border-radius(5px);
}
```

The preprocessor will then compile that into this:

プリプロセッサは上記を以下のように変換する:

border-radiusミックスインをCSSに変換

```
.callout {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
  border-radius: 5px;
}
```

関数

ミックスインの例はすでに関数のように見えるが、関数には値の計算に関する強力な機能がある。例えば、`lighten`関数は色の値と割合で値の明度を調整することができる。

lighten関数を使って色の値を調整する

```
$btnColor: #036;
.btn {
  background-color: $btnColor;
}
.btn:hover {
  background-color: lighten($btnColor, 20%);
}
```

プリプロセッサは上記を以下のように変換する:

色関数をCSSに変換する

```
.btn {
  background-color: #003366;
}
.btn:hover {
  background-color: #0066cc;
}
```

Sassにはこの例に似た便利な関数がいくつかあり、Compassではそれにもっと関数を追加できる。(もしSassを利用しているのであればCompassも併せて利用することを推奨する。)

拡張

拡張(`extend`)とは、あるモジュールのプロパティをほかのモジュールのプロパティで拡張できる。Sassでは`extend`文を使う。

Sassの拡張(extend)

```
.btn {
  display: block;
  padding: 5px 10px;
  background-color: #003366;
}
.btn-default {
  @extend .btn;
  background-color: #0066cc;
}
```

btnのスタイルをbtn-defaultにも追加する。Sassは2つ目の宣言内で重複するルールを繰り返さず、1つ目のルールセットにセレクタを追加する。

SassのextendをCSSに変換

```
.btn, .btn-default {
  display: block;
  padding: 5px 10px;
  background-color: #003366; }

.btn-default {
  background-color: #0066cc; }
```

拡張はシンプルなセレクタのみ可能で、例えば#main .btnで拡張することはできない。拡張(extend)機能についてはSMACSSアプローチにどう影響を与えるのかについてをこの章の後半で紹介していく。

そのほかにもたくさん

プリプロセッサの機能としては、ここで紹介したのはほんの1例だ。ほかにも様々な機能や例がそれぞれのプリプロセッサの言語にある。はじめは圧倒されるかもしれないが、すべての機能を全部使い切る必要はまったくない。

考え得る問題点とその回避方法

こんな言葉を聞いたことがあるだろう。『強靱な力には大きな責任も伴う』。プリプロセッサは多くの機能を提供し、変換前のCSSをきれいに無駄のない状態に保ってくれる。しかし、変換されれば、その魔法は肥大化したCSSになり、デバッグしづらくなる。言い換えれば、はじめの状態に戻っただけだとも言える。肥大化したコードは、手で書こうと、DreamweaverのようなWYSIWYGツールで書こうと、プリプロセッサを使おうと生み出してしまうことは可能だ。もちろん、その反対にどの手段を使おうと素晴らしいコードを書くこともまた可能だ。

それではどこが問題となりやすいのかを見ていこう。

深い入れ子ルール

入れ子ルールを使い始めると、やりすぎの状態に陥るのは簡単だ。以下のような例を考えて欲しい:

Sassの深い入れ子ルール

```
#sidebar {
  width: 300px;
  .box {
    border-radius: 10px;
    background-color: #EEE;
    h2 {
      font-size: 14px;
      font-weight: bold;
    }
    ul {
      margin: 0;
      padding: 0;
      a {
        display: block;
      }
    }
  }
}
```

この例は決して珍しい例ではない。実際に私もこのような例をたくさん見てきた。以下はCSSに変換した例だ:

Sassの深い入れ子ルールをCSSに変換

```
#sidebar {
  width: 300px; }
#sidebar .box {
  border-radius: 10px;
  background-color: #EEE; }
#sidebar .box h2 {
  font-size: 14px;
  font-weight: bold; }
#sidebar .box ul {
  margin: 0;
  padding: 0; }
#sidebar .box ul a {
  display: block; }
```

SMACSSにおける入れ子ルール

SMACSSアプローチでは適応性の深度を最低限に抑えるため、もともと深い入れ子を使わない。レイアウトとモジュールの分割によってこれらの問題を回避できる。SMACSSでは先ほどの例は以下のように書く:

SMACSSによる深い入れ子ルール

```
#sidebar {
  width: 300px;
}

.box {
  border-radius: 10px;
  background-color: #EEE;
}

.box-header {
  font-size: 14px;
  font-weight: bold;
}

.box-body {
  margin: 0;
  padding: 0;
  a {
    display: block;
  }
}
```

そもそもほとんど入れ子になる状態がない! それは必要なスタイルを適応するのに長いセレクトタが必要ないからだ。モジュール内の要素をターゲットにする場合にのみ入れ子を追加する。

長いセレクトタの連鎖はブラウザが要素に対してスタイルが適応されるか否かを確定するのに余計な負荷をかけるだけだ。

不必要な拡張

先ほど使ったデフォルトボタンスタイル拡張(extend)の例をもう1度見てみよう:

Sassを使ってボタンの拡張

```
.btn {
  display: block;
  padding: 5px 10px;
  background-color: #003366;
}
.btn-default {
  @extend .btn;
  background-color: #0066cc;
}
```

こうすることで以下のHTML、`btn`と`btn-default`の両方を要素に追加する必要がなくなる。HTML上で複数宣言していたのをCSSでの複数宣言に変更したということだ。

リンクにクラスを付与する

```
<a class="btn-default">My button</a>
```

モジュールを拡張して、サブモジュールを作成すると、HTMLで複数のクラスを付与する必要がなくなる。そのため、命名規則についてはより重要になってくると言える。`btn`というモジュール名とそれを拡張している`small`サブモジュールがあったとすると、`small`クラスをHTMLに付与することになるので、役割が明確ではなくなってしまう。`btn-small`とすることで、サブモジュールとしても機能し、目的も明確にできる。

Sassを使ったソースを見れば、`btn-default`が`@extend`を使っているので、サブモジュールであることはわかる。変換されたCSSをみても、`btn-default`は`btn`クラスと連携しているのでサブモジュールだとわかる。

モジュールの拡張で発生するミスは全く関係のないモジュールを拡張しはじめた場合だ。

Sassのextendを様々なモジュールで利用する

```
.box {
  border-radius: 5px;
  box-shadow: 0 0 3px 0 #000000;
  background-color: #003366;
}
.btn {
  @extend .box;
  background-color: #0066cc;
}
```

複数のモジュール間でextendを利用することで、モジュールのスタイルも複数の箇所に散在することになる。

そして、CSS内ですべての密接に関連づけてしまうと、Ajaxなどを使った読み込みができなくなるか、アプリケーションに対して読み込むスタイルに併せてCSSへの変換を行う必要が出てくる。この例の場合では、ボタンとボックススタイルは同じタイミングでロードされている必要がある。

同じモジュール間のextendでもAjaxを使ったスタイルの読み込みを行うプロジェクトを複雑にする。例えば、Yahoo!の場合、デフォルトボタン用のスタイルはページのロード時に読み込むが、二次的なスタイル、例えばメール作成画面用などは、画面が切り替わった時のみ読み込みを行う。こうすることで、初期ページ読み込みを早くすることが可能だからだ。

SMACSSの拡張方法

SMACSSアプローチでは拡張はCSSレベルではなく、HTML上で複数クラスを使うことでHTMLレベルのみで解決できる。

SMACSSモジュールクラスをボタンに割り当てる

```
<a class="btn btn-default">ボタン</a>
```

こうすることで、モジュールのルート要素がどこであるかが明確になる。ブラウザ上でHTMLを見る場合、モジュールがどこで始まり、どこで終わるかを区別をつけるのは難しい。ルートモジュール名にはハイフンがついていないため、モジュールを区別しやすくなる。

HTMLに対して(あまり必要とは思えないかもしれない)複数のクラスを追加することは、クラスが**ありすぎる**状態のように見えるかもしれない。しかし、これらのクラスは**不必要**ではない。要素の目的を明確にし、ひいてはセマンティック性を向上することになる。

Mixinsの使いすぎ

ミックスインは繰り返しをさけるために便利な方法だ。しかしCSSのクラスもまた繰り返しを避けるのに便利な方法でもある。もし同じCSSルールを様々な場所で使い始めたら、別のクラスにまとめるほうがいい。

いくつかのモジュールで灰色の背景と青い角丸ボーダーを共有しているとすると、ミックスインを作ろうと考えるかもしれない。

Sassのミックスインの一般的なパターン

```
@mixin theme-border {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE;
}

.callout {
  @include theme-border;
}

.summary {
  @include theme-border;
}
```

以下のようにCSSに変換される:

ミックスインをCSSに変換

```
.callout {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE; }

.summary {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE; }
```

SMACCの繰り返しのためのパターン

このケースでは見た目に関わるスタイルが様々なコンテナに付与されるため、個別のクラスとして独立させる方が良さそうだ。

一般的なパターン用のクラスを定義する

```
.theme-border {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE;
}

.callout {
}

.summary {
}
```

Then apply it to elements as required:

そして必要な要素に対して付与する:

クラスを付与する

```
<div class="callout theme-border"></div>
<div class="summary theme-border"></div>
```

引数を利用できるミックスイン

引数を利用するミックスインは、CSSでできること以上のことができる。同じことをCSSのみで実践しようとしたら、様々なバリエーション用のクラスを定義する必要がある。この章の始めの方で紹介した `border-radius` のミックスインは引数利用のミックスインの好例だ。

プリプロセッサをSMACSSで活かす

これまで、SMACSSでどう解決するかと比べながらプリプロセッサに対していくつかのよくある落とし穴を見てきた。これらの問題に対する答えは、いつだって節度を持って利用してほしい、ということだ。変換後のCSSを見直して、最終的な結果が自分が求めているものかを確認しよう。もし、繰り返しがたくさんあるようであれば、自分のアプローチに対してリファクタが必要であるというサインだ。

それではプリプロセッサがよりよいモジュールアプローチを行う手助けになる例をいくつか見ていこう。

入れ子ルールと状態を元にしたメディアクエリ

状態の変更の章で紹介したように、メディアクエリは状態の変更の管理に利用できる。多くのチュートリアルで別のスタイルシートを用意して、その状態にあったスタイルをそれぞれのファイルに記述する方法が紹介されているが、こうしてしまうとモジュールの定義が様々なファイルに散在し、管理が難しくなってしまう。

Sassはメディアクエリを入れ子にすることができるので、状態の変更に対するスタイルをモジュールの近くで定義することが可能となる。

入れ子ルールを使ったメディアクエリの例:

Sassを使ったメディアクエリの入れ子

```
.nav > li {
  width: 100%;

  @media screen and (min-width: 320px) {
    width: 100px;
    float: left;
  }

  @media screen and (min-width: 1200px) {
    width: 250px;
  }
}
```

デフォルトの状態とその変更後の状態と同じモジュール内で定義することができる。Sassではメディアクエリの中にさらにメディアクエリを入れ子にすることもでき、それらのメディアクエリの条件を結合することもできる。

以下が入れ子の例をCSSに変換した例となる:

Sassを使ったメディアクエリの入れ子をCSSに変換

```
.nav > li {
  width: 100%; }
@media screen and (min-width: 320px) {
  .nav > li {
    width: 100px;
    float: left; } }
@media screen and (min-width: 1200px) {
  .nav > li {
    width: 250px; } }
```

Sassは別のメディアクエリを作成し、それぞれにセレクタを内包してくれる。この例ではあえて小さな画面の状態をデフォルトとし、320px以内となる場合に適応するようにしている。そして特定の幅になったら、ナビゲーションをフロートするように変更している。最終的に1200pxの幅になった際にも、フロートを再定義することは

ない。私はデフォルト状態から様々なメディアクエリ内でこのように継承していくアプローチが気に入っている。

さらに、モジュールに対してほかのどんな状態も同じモジュール内で定義することができる。

ファイルの編成

プリプロセッサはSMACSSが推奨する懸念の分離を促進している。

以下はプロジェクト内でどのようにファイルを分離するかのガイドラインだ:

- ・ ベースルールはすべて1つのファイルに記述する。
- ・ レイアウトの種類に応じて、すべて1ファイルに記述するか、主なレイアウトごとにファイルを作成する。
- ・ モジュールはそれぞれのファイルに記述する。
- ・ プロジェクトの規模によってはサブモジュールも個別のファイルに記述する。
- ・ グローバルで利用する状態は専用のファイルに記述する。
- ・ レイアウトやモジュールに関連するメディアクエリを含む状態は関連するモジュールファイルに記述する。

このようにファイルを分割することで、プロトタイプを行うのが容易になる。個別のコンポーネント用のHTMLテンプレートを作ることでもできるし、テンプレートに利用する、CSSとコンポーネント(またはコンポーネントのサブセット)はそれぞれを個別に分離してテストすることも可能だ。

プリプロセッサ用のコンポーネント、ミックスインや変数はそれぞれのファイルに記述する。

ディレクトリ構成のサンプル

```
+--layout/  
| +-grid.scss  
| +-alternate.scss  
+-module/  
| +-callout.scss  
| +-bookmarks.scss  
| +-btn.scss  
| +-btn-compose.scss  
+-base.scss  
+-states.scss  
+-site-settings.scss  
+-mixins.scss
```

最後にほかのファイルを読み出すためのCSSファイルを作る。多くのサイトではすべてのファイルを1つのマスタースタイルシートで呼び出すことになる。アセットの呼び出しに条件が必要になるプロジェクトでは、特定の画面にのみ読み込むためのファイルを用意する必要がある。

マスタースタイルファイル

```
@import  
  "site-settings",  
  "mixins",  
  "base",  
  "states",  
  "layout/grid",  
  "module/btn",  
  "module/bookmarks",  
  "module/callout";
```

プリプロセッサはこれらを1つのファイルとして変換してくれる。

公開前にはCSSを圧縮しておくこと。(環境によってデプロイ用のビルドスクリプトがすでにあるかもしれない。その場合はプリプロセッサのコンパイルもビルドプロセスに含んでおくことを確認してほしい。)

Sassを使ったCSSの圧縮

```
sass -t compressed master.scss master.css
```

プリプロセッサについてのまとめ

プリプロセッサとは何なのか、そしてどうインストールするかを見てきた。よく使われる機能といくつかの落とし穴も見てきた。最後にプリプロセッサがプロジェクトの構成を行うのに便利であることも紹介した。プリプロセッサはプロセスとして便利なものであることは間違いない。

ベースルールをあきらめる時

決して多いとは言えないがあまり使うことがないいくつかの要素がある。結果として(私もそうだったように)それらの要素を単一で変更されないベースルールとしてスタイルしてしまうかもしれない。ここまで読んで来ておそらくわかるように、変更は必ず発生する。その変更に対して計画し、将来的な変更がすでに実装したスタイルを複雑なものにしてしまわないようにすることができる。

どんな要素がこの問題に直面することになるだろうか? `button`、`table`、そして `input` 要素がもっとも一般的といえるだろう。早速、プロジェクトで起こりえそうな例を掘り下げて見ていこう。

Table

ウェブスタンダードの実装者の戦略からレイアウトにテーブルを使うという項目は消えて無くなった。その結果、プロジェクト内でテーブルを利用する必要性はそれほど多くはない。

もちろん、使わなければいけなくなるその日までだ。

この最初で唯一のテーブルは特定の、例えば比較表のデータを表示するのに利用している。比較表には特定のパディングがあり、コラム揃えがあり、ボーダーがあり、すばらしいデザインだ。

テーブルスタイル

```
table {
  width: 100%;
  border: 1px solid #000;
  border-width: 1px 0;
  border-collapse: collapse;
}

td {
  border: 1px solid #666;
  border-width: 1px 0;
}

td:nth-child(2n) {
  background-color: #EEE;
}
```

何日か、何週間か、何ヶ月か後に別のテーブルを追加する必要に迫られた。今回は別の目的のために利用する。見出しは左側に、データを右側に表示する必要がある。ボーダーはなくなり、背景を変更しなければならない。通常であればデフォルトスタイルを上書きすることだろう。

以前のスタイルを上書きする

```
table.info {
  border-width: 0;
}

td.info {
  border-width: 0;
}

td.info:nth-child(2n) {
  background-color: transparent;
}

.info > tr > th {
  text-align: left;
}

.info > tr > td {
  text-align: right;
}
```

問題となるのはスタイルを上書きしている理由が、ベースルールが単一の目的のためだけに存在しているからだ。ベースルールはデフォルトスタイルを定義するために利用されるべきで、特定のモジュールで拡張するように使うべきだ。比較表は単一の目的を持っていたが、カスタムデザインがあったため、モジュールだったわけだ。モジュール内でたった1回しか使われなかったとしてもその事実が変わることはない。

解決策は明らかだ: モジュールを作ればいい。

代わりにモジュールを作る

```
table {
  width: 100%;
  border-collapse: collapse;
}

.comparison {
  border: 1px solid #000;
  border-width: 1px 0;
}

.comparison > tr > td {
  border: 1px solid #666;
  border-width: 1px 0;
}

.comparison > tr > td:nth-child(2n) {
  background-color: #EEE;
}

.info > tr > th {
  text-align: left;
}

.info > tr > td {
  text-align: right;
}
```

テーブル要素にはベースとなるスタイルが設定されたままだ。私はこれまでテーブルがコンテナと同じ幅にならない場面に出くわしたことがほとんどない。同じように**border-collapse: collapse**を使わなかったこともない。これらはブラウザのデフォルトであるべきだと思える。

比較表モジュールは元々そうするべきであるように、独立することができた。子セレクタを使って影響を最小限に止めるようにしてある。もしテーブルの中にテーブルがあったとしても(普通は避けるべき)、比較表モジュールは内部のテーブルに影響を与えることはない。infoモジュールはたった2つの簡単なルールで単純化された。

全体的に少ないCSSで元々必要だった結果を得ることができたと同時にコードもより明確になった。Win-winだ。

すでに言及したように、`button`と`input`要素もテーブルと同じ運命を辿ることが多い。もしスタイルが特定の目的を果たすためにあるのであれば、モジュールを作るべきだ。そうすることで古いコードを上書きしたり、リライトしたりする必要を避けることができる。

アイコンモジュール

CSSスプライトはモダンウェブ開発における大黒柱になった。複数のアセットを1つのリソースに変換し、HTTPリクエストを最小限にしつつ、ロールオーバーの状態用の画像も必要の時にはすでに読み込み済みになる。

CSSスプライトテクニックが広く知られる前までは、画像はほかの要素を上レイヤーに持つことができる背景画像か、表層の画像としての2種類のコンテキストで使われてきた。現在ではすべて背景画像として利用することができ、要素の中で自由に位置を変えることができる。

この章では背景画像として利用する場合について紹介していく。

例を見ていくのが最もわかりやすいので、早速メニューとともに使われるアイコンを例に見ていこう。



メニューHTML

```
<ul class="メニュー">
  <li class="menu-inbox">受信箱</li>
  <li class="menu-drafts">下書き</li>
</ul>
```

このHTMLはわかりやすいものだろう。メニューアイテムのリストがあり、それぞれのアイテムに対してクラスを付与し個別でスタイルできるようにしてある。

メニューCSS

```
.menu li {
    background: url(/img/sprite.png) no-repeat 0 0;
    padding-left: 20px;
}

.menu .menu-inbox {
    background-position: 0 -20px;
}

.menu .menu-drafts {
    background-position: 0 -40px;
}
```

すべてのリストアイテムに1つのスプライトをセットして個別のリストアイテムは背景の位置を調整し、正しいアイコンを表示する。

表面上は非常に良さそうに見えるし、きちんと動作する。しかしいつも通り、物事をややこしくするエッジケースも考えられる。

- ・ リストアイテムという特定のHTML構造に依存する。
- ・ ほかのモジュールで利用する際にスプライトを再度定義する必要がある。
- ・ 要素内の配置は非常にもろい。フォントサイズを大きくするだけでもスプライトのほかのパーツが見えてしまうかもしれない。
- ・ 横並びのスプライトしか利用できず、xのポジションは0で固定されているため、右から左に文字が配置されるインターフェイスを実装するのが難しくなる。

これらの問題を解決するためにアイコンそのものがモジュールになるように変更する必要がある。これがアイコンモジュールだ。

HTMLをアイコンモジュールを作成できるように再構築する

```
<li><i class="ico ico-16 ico-inbox"></i> 受信箱</li>
```

多くの人は*i*タグを利用するのをためらうことだろう。しかしセマンティック性にかけていても、空の要素だったとしても、小さなタグなので利用することにしている。それではなぜコンテンツがないのか？この例ではアイコンの隣には常にテキストあるからだ。もしアイコンのみだった場合、タイトル属性を追加して、スクリーンリーダから読み取れるように、またツールチップで表示されるようにする。もしこの考えに賛成できず、spanを使う方が適切だと考えていることは決して間違いではない。

1つのタグに様々なアイコンクラスを付与することで、ほかのHTML依存を無くすことができる。ではなぜ、別のクラスなのか？それぞれは若干異なる役割を持っていて、最終的にすべて合わさって要素の代替として機能する。

アイコンモジュールCSS

```
.ico {
  display: inline-block;
  background: url(/img/sprite.png) no-repeat;
  line-height: 0;
  vertical-align: bottom;
}

.ico-16 {
  height: 16px;
  width: 16px;
}

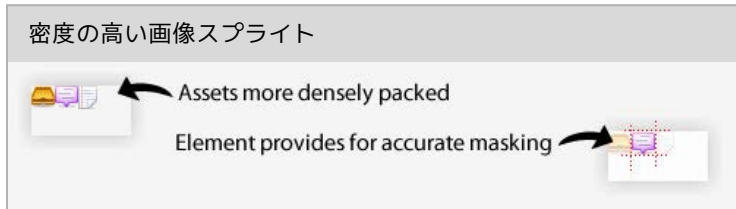
.ico-inbox {
  background-position: 20px 20px;
}

.ico-drafts {
  background-position: 20px 40px;
}
```


`ico`クラスは要素をインラインブロック要素に変換するための定義をしている。こうすることで要素の再現のベースとなる。`vertical-align`の値はテキストに対してどこにアイコンが配置されるかを決定しているので、調整が必要になるかもしれない。インターネットエクスプローラはブロック要素に対して`inline-block`を付与すると問題が発生する。しかしここではインライン要素に対して付与しているため、その問題は発生しない。代替りの手段として`{ zoom:1; display:inline; }`をブロック要素に付与することでIEでもブロック要素がインラインブロック要素のような振る舞いをさせることはできる。

`ico-16`クラスは高さと幅を指定するために利用している。もしプロジェクト内に1つのアイコンサイズしかない場合にはこれらのサイズ指定は`ico`で行う。アイコンサイズが異なる場合は、その特定のアイコンに対してクラスを付与して定義する。このサイトでは4つの異なるアイコンサイズを設定した。

最後のクラス`ico-inbox`はスプライトの配置を正しい位置に設定している。固定のアイコンサイズを指定することで、親要素が大きくなっても、文字が右から左に流れるインターフェイスでも背景のポジションを変更する必要がなくなる。



密度の高い画像を使うことでよりよい圧縮率を達成できる。ファイルサイズが小さいと言うことはサイトのパフォーマンスをより最適化することができるということだ(もしまだ利用していないようであれば、Yahoo!のSmush.it²⁰サービスや、MacであればImageOptim²¹を使って画像の最適化を行って欲しい)。

20.<http://www.smushit.com/ysmush.it/>

21.<http://imageoptim.pornel.net/>

ここではプロジェクト内の特定のパーツをより柔軟にするためのリファクタ方法の例を紹介してきた。問題に対して様々な方法があり、表面的には問題ないように思えてもプロジェクトが進むうちに問題が表面化する場合もある。プロジェクトが進化することによって複雑さが表面化する。そしてそれらに対して最適な解を見つけ出すことこそが、ウェブ開発の醍醐味だ。

複雑な継承

この章では継承がどのようにベストなプランを駄目にするのかを見ていく。

カレンダーを例に一般的な状態ルールがテーブルセル内の継承による衝突とその回避方法を見ていこう。

カレンダーテーブル

```
<table class="cal">
  <tr>
    <td>1</td>
    <td>2</td>
    <td>3</td>
    <td>4</td>
    <td>5</td>
    <td>6</td>
    <td>7</td>
  </tr>
  <!-- 3-4回繰り返す -->
</table>
```

カレンダーには行列のあるテーブルがある。それぞれのセルが1日を示し、デフォルトスタイルは1日を示すセルが通常の日の場合を表示している。

| 日セル | | | | | | |
|--|----------|----------|----------|----------|----------|----------|
| <pre>.cal td { background-color: #EFEFEF; color: #333; }</pre> | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

このテーブルのそれぞれのセルには薄いグレーの背景と濃いグレーのテキストとなるようにスタイルしてある。それでは今日がいつなのかをハイライトしてみよう。

| 今日のスタイル | | | | | | |
|---|----------|----------|----------|----------|----------|----------|
| <pre>.cal td.cal-today { background-color: #F33; color: #000; }</pre> | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

`cal-today`クラスは今日を示すためのクラスが`cal`モジュールの1部であることを意味している。デフォルトスタイルを上書きできるようにこのスタイルの詳細度は上げている。セレクトタを単に`td.cal-today`とすることもできるが、このスタイルがデフォルト状態のルールの上に記述されている必要がある。もし`.cal-today`を使った場合、`!important`を使わなければこのスタイルは正しく動作しない。

プロジェクトの開発が進む度に、このような小さな決定を積み重ねていくということを認識しておくことは大事なことだ。今回の例では`.cal-today`クラスはテーブルセル(`<td>`)にしか割り当てることができないと宣言していて、それは`cal`クラスの中で利用することも宣言していることになる(この点についてはSMACSSアプローチを利用する限り、すでに明確ではある)。

例に戻って、すべて問題ないように見える。では、このカレンダーを週カレンダーの中に配置される詳細を見るためのカレンダーだとすると、このミニチュアカレンダーは選択された週がどの週なのかを表示する必要が出てくる。

選択された行

```
<tr class="is-selected">
  <td>1</td>
  <td class="cal-today">2</td>
  <td>3</td>
  ...
</tr>
```

選択された状態はアプリケーション全体で使われるため、ここで利用するのも問題はない。それでは選択された状態のスタイルはどのような見た目になるだろうか？

選択された行のルール

```
.is-selected {
  background-color: #FFD700; /* 黄色 */
  color: #000;
}
```

| | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----------|----------|----------|----------|----------|----------|----------|

Where's the background?

問題を見つけることができただろうか？問題はテーブル行に付与された背景色はより高い詳細度を持つベースの日を示すスタイルと今日を示すスタイルに上書きされてしまう点にある。

この状態に対して `!important` を追加することもできる。以前に言及したようにそうすることには問題はないが、同じ要素に対してスタイルを付与するための詳細度を高めることはできるが、セルに対

して継承されないためあまり役に立たない。`!important`は詳細度は上書きできても、継承を上書きすることはできないのだ。

やはり選択された状態を子要素に反映させるために新しいルールを定義する必要がある。

テーブルセル用の選択された行ルール

```
.is-selected td {
  background-color: #FFD700; /* 黄色 */
  color: #000;
}
```

| | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----------|----------|----------|----------|----------|----------|----------|

もしこのセレクトタがカレンダーの日セレクトタの後に定義されていれば、期待通りにすべて描画されることになる。

セルはどの色になるだろうか？答えは状況に依って違う、だ。このスタイルが`cal`クラスの後で宣言されたか、前で宣言されたか？もし後だとしたら、行内のすべてのセルは正しくスタイルされる。今日を示すセルは赤になり、このプロジェクトに関して言えば、それこそが求めるスタイルだ。

!importantが問題になる場合

議論のために、もし選択された状態のセルに`!important`を追加したらどうなるだろうか？今日を示すセルはほかの週と同じスタイルになってしまう。

!importantを追加するとどうなるのか？

```
.is-selected td {  
  background-color: #FFD700 !important; /* 黄色 */  
  color: #000 !important;  
}
```



今日を示すセルを正しくスタイルするためには、モジュールルールと状態ルールを合わせた新しいルールを作る以外に方法がない。

追加のルールを使って詳細度の問題を回避する

```
.is-selected td {  
  background-color: #FFD700 !important; /* 黄色 */  
  color: #000 !important;  
}  
  
.is-selected td.cal-today {  
  background-color: #F33 !important;  
  color: #000 !important;  
}
```

この最後の例から見て取れるように、正しく表示させるために新しいセレクタを追加しなければならなかったり、!importantを追加したりしなければならない。理想的とはとても言えない状況だ。

不完全な世界

この例の目的は継承が考え抜いた計画に大損害を与え、それに対する完璧な解決策がないことを実証することだ。SMACSSはこのような多くの問題を和らげようとするが最終的に時には理想的とは言えない解決をしなければならない場面もある。

しかし、このようなシチュエーションを最低限に止めることはプロジェクトのメンテナンス性を保つのに役立つだろう。

コードのフォーマット

個人個人にそれぞれ自分の方法がある。今使っているツールやテクニックは自らのトライアンドエラーから得たものか、誰かほかの人が使っているのを知っているかどちらかから得たものだ。私がウェブ開発を始めた頃は、Dreamweaverを使っていた。Dreamweaverには数多くの機能が搭載されていて、静的なHTMLサイトを素早く、効率的に構築することができた。その後、私の同僚がUltraeditを使って、彼がより早く仕事を終わらせているのを見て、私が元々使っていたツールセットの1部を保管するように使い方を覚えた。それと同じことがコードそのものにも言える。ほかの誰かが使っているテクニックやスタイルを見て、私の方法として取り入れてきたのだ。

このコードのフォーマットのセクションは私がどのようにコーディングをしているのかを簡単に紹介する。そうすることでほかの誰かが私のコーディングスタイルを取り込めることだろう。

単一行と複数行

私は長年、CSSを単一行アプローチ²²でコーディングしてきた。どういうことかと言うと、ルールセット内のすべてのプロパティは同じ行で宣言されるということだ。こうすることで左端にあるセレクトを素早く見通すことができる。セレクトの見通しやすさはプロパティがきれいに並んでいることよりも大事なことだったからだ。数年前まではルールセットで使われるプロパティのリストは非常に短かった。指折り数えられる以上のプロパティを利用することはあまり多くはなかった。このことが探しているセレクトとプロパティを同じ画面内で見ることが出来た理由だ。

無数のベンダープリフィックスとともにCSS3を利用し始め、すぐにこれまでの書き方では手に負えなくなってきた。それに加えて、よ

22. <http://orderedlist.com/resources/html-css/single-line-css/>

り大きなチームで働くことで、プロパティと値にそれぞれに行があった方が全員にとって容易になってきた。

バンダープリフィックスにあふれたCSS3プロパティはすべてを1行で納めると可読性が落ちる。

```
.module {
  display: block;
  height: 200px;
  width: 200px;
  float: left;
  position: relative;

  border: 1px solid #333;
  -moz-border-radius: 10px;
  -webkit-border-radius: 10px;
  border-radius: 10px;

  -moz-box-shadow: 10px 10px 5px #888;
  -webkit-box-shadow: 10px 10px 5px #888;
  box-shadow: 10px 10px 5px #888;

  font-size: 12px;
  text-transform: uppercase;
}
```

この例では11のプロパティが宣言されているが、モジュールに対してもう1つ、2つ別の機能を追加したらあと6つのプロパティの追加はあつという間だろう。これらを1行で納めてしまおうとすると、はじめのいくつかのプロパティはスクリーン内にあるが、残りはすべて右にスクロールしないと見ることができない。こうなってしまうと、ドキュメントを一瞥してどのプロパティが定義されているのかを確認しづらくなってしまふ。さらに単一行で書くとバージョンコントロールのdiff比較が難しくなってしまう。

ほかにも例で注目して欲しいところがいくつかある:

- ・ コロンの後のスペース
- ・ 各宣言には前に4スペース(タブではない)
- ・ プロパティを種類でグループ化
- ・ 開始の中括弧はルールセットと同じ行
- ・ 色の宣言は短縮形

すべては好みによるもので、全く異なるアプローチをしても問題ない。私自身はもっとも自然で理解しやすい方法だと思っている。

プロパティをグループ化

アルファベット順で並べ替える人もいれば、まったく気にしない人もいるし、それぞれの任意なシステムに従って並べ替えをする人もいる。私はその最後のカテゴリに属する人だ。だからといって完全に任意というわけでもない。最も大事なものからそうでないものの順に並べ替えていると説明できるが、それではいったい何を持って大事だと言えるのか？

私は以下の順にソートをしている:

1. ボックス
2. ボーダー
3. バックグラウンド
4. テキスト
5. その他

ボックスはボックスの表示とポジションに関わるプロパティで、`display`、`float`、`position`、`left`、`top`、`height`、`width`などが含まれる。これらはドキュメントにあるほかのフローに影響を与えるため私にとってはもっとも重要なプロパティだ。

ボーダーには`border`、あまり使われない`border-image`、そして`border-radius`が含まれる。

その次に来るのがバックグラウンド。CSS3のグラデーションの登場により、バックグラウンドの宣言は非常に冗長化しやすい。ここでもベンダープリフィックスの存在が問題をややこしくする。

CSS3の宣言の複数バックグラウンド例。例はLea VerouのCSS3 Pattern Gallery²³を参照した。

```
background-color: #6d695c;
background-image: url("/img/argyle.png");
background-image:
  repeating-linear-gradient(-30deg,
    rgba(255,255,255,.1), rgba(255,255,255,.1) 1px,
    transparent 1px, transparent 60px),
  repeating-linear-gradient(30deg,
    rgba(255,255,255,.1), rgba(255,255,255,.1) 1px,
    transparent 1px, transparent 60px),
  linear-gradient(30deg, rgba(0,0,0,.1) 25%,
    transparent 25%, transparent 75%, rgba(0,0,0,.1)
    75%, rgba(0,0,0,.1)),
  linear-gradient(-30deg, rgba(0,0,0,.1) 25%,
    transparent 25%, transparent 75%, rgba(0,0,0,.1)
    75%, rgba(0,0,0,.1));
background-size: 70px 120px;
```

CSS3のグラデーションを使うことで複雑なパターンを実現することができるようになったが、同時にバックグラウンドの宣言は非常に長くなってしまふ。この例ではベンダープリフィックスは記述していない。もし記述していたらどうなるか想像してみたい!

テキストはfont-family、font-size、text-transform、letter-spacingなどタイポグラフィーに関わるすべてのCSSプロパティを含む。

ここまで紹介してきたカテゴリに属さないプロパティは最後に追加される。

色の宣言

ここで触れるのもすこし馬鹿げているように思えるが、これまでいろいろな方法をいろいろなデベロッパが利用しているのを見てきた。blackやwhiteというキーワードを使う人もいるが、私は可能な限り3つか6つで宣言できるhexを使って宣言することにしてい

23.<http://leaverou.me/css3patterns/>

る。`#000`や`#FFF`はキーワードで宣言するのよりほんの少しではあるが短い。同じように`rgb`や`hsl`も`hex`の方が短いので使わない。`rgba`と`hsla`は`hex`で表現出来ないのももちろん利用することにはなる。

一貫性を保つ

最終的に最も大事なことは、SMACSSで説明されているように、スタンダードを作り、ドキュメントし、それに対して一貫性を保つことだ。そうすることでプロジェクトが大きくなっていくごとに、自分にとっても他者にとってもプラスに働くことが多い。

リソース

非常に多くのすばらしいツール、リソースがある。いくつかは本書で議論したコンセプトと直接関連するもので、いくつかはレポーターとして知っておきたい便利なツールたちだ。

CSS プリプロセッサ

- LESS²⁴
- Sass²⁵

コンポーネントベースのフレームワーク/方法論

- Object-Oriented CSS (OOCSS)²⁶
 - OOCSS for JavaScript Pirates Slides²⁷
 - MailChimp UI Library based on OOCSS²⁸
- BEM²⁹

その他のフレームワーク

- HTML5 Boilerplate³⁰
- normalize.css³¹
- Bootstrap³²
- 960.gs³³
- Eric Meyer CSS Reset³⁴

24.<http://lesscss.org/>

25.<http://sass-lang.com/>

26.<http://oocss.org/>

27.<http://speakerrate.com/talks/4642-oocss-for-javascript-pirates>

28.<http://www.flickr.com/photos/aaronwalter/5579386649/>

29.<http://bem.github.com/bem-method/html/all.en.html>

30.<http://html5boilerplate.com/>

ドキュメンテーション

- ・ [Front-end Style Guides](#)³⁵
- ・ [Knyle Style Sheets](#)³⁶

その他のリソース

[mustache](#)³⁷はロジックレスのテンプレート言語でYahoo!で利用されている。

[Pattern Primer](#)³⁸はPHPのスクリプトでHTMLのスニペットを1ページでプレビューすることができる。

[Terrifically](#)³⁹はOOCSSアプローチを使うためのJavaScript/jQueryフレームワーク。

31. <https://github.com/necolas/normalize.css/>

32. <http://twitter.github.com/bootstrap/>

33. <http://960.gs/>

34. <http://meyerweb.com/eric/tools/css/reset/>

35. <http://24ways.org/2011/front-end-style-guides>

36. <http://warpspire.com/posts/kss/>

37. <http://mustache.github.com/>

38. <https://github.com/adactio/Pattern-Primer>

39. <http://www.terrifically.org/>