结构化CSS设计思维

LESS、SASS等预处理器给CSS开发带来了语法的灵活和便利，其本身却没有给我们带来结构化设计思维。很少有人讨论CSS的架构设计，而很多框架本身，如Bootstrap确实有架构设计思维作为根基。 要理解这些框架，高效使用这些框架，甚至最后实现自己的框架，必须要了解结构化CSS设计思想。

我不是前端专家，但是我想，是否一定要等成为了专家才能布道？那是不是太晚了。 所以我是作为一个CSS的学习者，给其他CSS学习者分享一下结构化CSS设计的学习心得。我更多的是一个后端开发者，后端开发的成熟思想必定能给前端带来新鲜血液。

前言

CSS从根本来讲就是一系列的规则，对Html（作为内容标识和呈现）的风格化得一系列规则，因此，语法级别也就是两个重要因素：选择器和应用的风格。 简单而超能。

一开始，CSS很容易学习，容易上手。当我们的网站成长为企业级网站时，我们的CSS开始成为臃肿、重复和难以为维护的一个混沌 。这种状态在软件开发中被称为Big ball of mud ，更为常见的讲法是面条式代码，是一种缺乏良好架构和设计的表象。 解决这个问题的方法和思维在后台软件开发中已经是比较多的讨论也相对成熟。同为软件开发的CSS开发，毋庸置疑，却可以这些借鉴思维。

面向方面

如前所述，CSS只是一系列规则，这是对CSS的初步认识，但我们不能停留在这个认识。如同，我们认识到所有物体都由原子组成，但是这个认识并不完全能够替代化学在分子级别的研究。

这些规则必须要进一步分化，各自有不同作用和角色，不能平面化。表面上同是选择器+风格化的CSS规则，根据它们的业务意义，应该划分为不同的类别或者方面 (Aspect 参考用词Aspect-oriented programming)。

这些方面分别是：

- 基本（元素）
- 布局
- 模块

- 状态
- 主题

明确的分析你要写的CSS规则属于哪个方面，在实现上区分这些方面并保持这样的分离，
是往结构化走的重要一步。

基本（元素）规则和布局规则

之所以把这两类单独提出来，是因为这是我们平时理解的css似乎就完全只有这两类。 我们对那些划归这两类的风格没有异议，而对那些属于这两类的反而不太理解。 还是先简单了解一下这两类本身的范围。

基本规则

是应用于基本元素级别的规则，作用于全局统一（默认）的风格。 最好的一个案例：CSS的重置框架reset.css以及bootstrap的改进方案normalize.css, 就完全是基本风格规则，不包含任何其他类型的规则。 虽然其作用和我们平时项目中的基本风格不相同，用来理解基本风格的范畴是及其恰当的。

布局规则

在我们一开始谈前端的结构化时，脑海中第一浮现的设计就是这个分层结构树(或则分形的思维)：

页面 Page => 布局 Layout => 模块 Module => 元素 Element 。

一个页面由布局组成，每个布局局部由一个或多个模块组成，一个模块有n个元素组成，看上去简单而完美，真正的结构化、模块化。 然而，现实世界总是非线性的。在实际的项目中，严格的层次关系设计，遇到了各类“特例”需要打破这个结构。

比如，AngularJS是MVC架构，更准确一些，它是层次结构化MVC, 一个大的MVC又由其它几个粒度更小的MVC组成， 特别是ui-router的嵌套状态和视图把这个结构表达的更清楚。 从设计的思维上，称之为分形更恰当。

当需要模块与模块之间的通信和信息交流时，这种结构却不能自然的支持，于是，有一个事件系统创造出来弥补这个缺陷。

之所以有这些“特例”，根本原因就是分形思维只适合在模块这一级别，而不能往上扩展到布局和页面界别，也不能往下扩展到元素级别。

布局就是布局，应该作为一个独立的方面存在。
布局规则中，我们之关注组件之间的相互关系，不关心组件自身的设计，也不关心布局

所在的位置。

比如，用list（ol或者ul)做布局用时：

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

.layout-grid > li {
    display: inline-block;
    margin: 0 0 10px 10px;
}
```

‘list-style-type’和’display’的设置，我们可以明显看出是布局，’margin’和’padding’似乎更像基本风格规则。 然而，从使用的目的来看，它们都是用于布局的方面。

这个例子，我们可以看出对规则的划分不是按CSS的技术特性，而是按**业务**特性：它们的作用，它们的“含义”。

模块规则

这个类别含义是很明确清楚，只是强调一下，模块可以放在布局的组件中，也可以放在另外一个模块内部，是嵌套的，就是前面说的分形。

用class和语义标签

我们一般都用class来定义模块，如果需要用到标签则只能是有语义的标签。如heading系列：

```
.module > h2{
    padding: ......
}
```

用subclass定义嵌套元素风格

如bootstrap的 `listgroup`：

```
<ul class="list-group">
  <li class="list-group-item">First item</li>
  <li class="list-group-item">Second item</li>
  <li class="list-group-item">Third item</li>
</ul>
```

可以看到，在 `list-group` 之外，它又另外定义了 `list-group-item` 来修饰 `li` ，而不是用以下方式省略掉子类的声明和使用：

```
.list-group > li {

    ...
}
```

为什么要这样？ 可以作为一个思考题放在这。

状态规则

状态和子模块有时候很相似，却有亮的明显区别：

- 状态改变布局风格或模块风格。
- 状态大部分时候和Javascript相联系。

这是什么意思呢，我们看看例子：

最经典的案例就是表单数据的有效性，一般都会引入class定义，类似 `is-valid` ；还有就是tab当前激活的状态 `is-tab-active` 等。 前者，会改变表单的布局：增加warning信息；后者，会改变tab模块的显示背景来表明当前tab是被选中的。而以上两个类也都会由javascript根据用户操作，动态的添加到相应的DOM元素中去。

从状态规则的两个关键词：改变和javascript，我们能很明显的看出它如其他规则的区别，仍然重点在它的用途和业务含义。 它最重要的一个业务逻辑就是：状态规则与时间相关，这也足以给它一个独立的地位，与模块规则的维度呈正交关系。

正交设计的延伸阅读：

> - Cohesion and Coupling: Principles of Orthogonal, Object-Oriented Programming
> - 变化驱动：正交设计

主题规则

主题是整个网站的风格全面的改变，可以跨项目的才改一次，因而可以在编译阶段进行如bootstrap的customize用于这种场景；还有就是在一个项目之内也容许用户动态改变。这些绝对是与其他规则不在一个方面，必定要独立出来。

这类规则会涉及到所有其他类型的规则，如：基本，模块甚至布局和状态，虽然代码量和工作量都较大，概念上却很清楚，这里就不再展开。

基本规则和模块规则的正交案例

在比较中，我们看看类别之间的区别。

场景

比如说我们网页中需要一个表格来显示一些信息，如 iPhone 7 的产品参数 https://www.apple.com/cn/iphone-7/specs/ 。

为它写一个简单的风格, 没有任何问题：

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

td{
  border: 1px solid #000;
  border-width: 1px 1px;
}
```


之后我们拿到一个新的需求，同样用表格但是用来比较不同型号的产品的参数，如 https://www.apple.com/cn/iphone/compare/。

为了给客户更好的体验，需要对表格风格做相应的调整，如相间隔的列用不同的背景色区分，表格的行之间需要实线间隔，而列之间则不要。 而且，这些修改不能影响之前信息表格的风格。

覆盖方式解决表格的变体

直观的解决方案,我们引用一个类 `comparison` 来覆盖之前的**基本规则**：

```
.comparision {
  border-width: 0px 0;
}
.comparison tr > td:nth-child(even){
  background-color: #AAA;
}

.comparison  tr > td {
   border-width: 1px 0;
}
```

完全依照新需求，做了三件事情： 1. 去标题的间隔线 2. 去掉了内容行之间的竖线间隔 3. 双列背景灰显。 点击参看在线Demo。

模块方式解决表格变体

然而，用模块的方式更为清楚，更容易扩展。

基本风格（全局）

首先，与OO中提出基类的思维类似，这里我们也提出公共的风格部分：

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

信息表风格类 info

然后，为原来的信息表格写出一个分支风格（OO中的子类）。

```
.info {
  border: 1px solid #000;
  border-width: 1px 1px;
}

.info  tr > td {
   border: 1px solid #000;
   border-width: 1px 1px;
}
```

比较表信息类 comparison

最后，为新的比较表格写出另外一个分支风格。

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

.comparison tr > td:nth-child(even){
```

```
    background-color: #AAA;
}
```

点击参看在线Demo。

比较分析

覆盖的方式中，尽管也用了类 comparison ,从设计的概念和使用的方式可以看到，和模块中的 comparison 还是不同的，其业务的语义性弱化很多，以至于作为开发者对其命名的准确程度都不太在意了。 这是一个很不好的倾向。

基本风格的 width 和 border-collapse 确确实实是全局的风格，不多一点也不少一点。

结构非常清晰，风格之间没有复杂的覆盖关系： 比如比较表格中的 border-width 不会像前面的实现那样，先 td{border-width: 1px 1px;} 然后又 .comparison tr > td {border-width: 1px 0;} 覆盖掉。 可以想象在实际项目中，更多层次的覆盖和更多规则的引入会带来多少的复杂度和差错率，你怎么能准确的判断到底是那个规则再起作用？

状态规则和模块规则的正交案例

因篇幅和时间问题，这个案例到下次再整理了。

设计思维回顾

分形

非常强大的思维，对它本身似乎有点陌生，当说起递归、全息理论是否更熟悉一些？ 这些都可以看作是分形思维的应用。

面向方面编程

有时候又称为面向切面编程，曾经是个炙手可热的名词，现在好像没怎么提起，不是不再适用而是思维已经进入常用编程思维，不再需要强调了。

正交设计

和面向方面有些雷同，在这重复也算一个强调吧。 另外，面向方面只能算正交设计的一种实现方式吧。

语义性

当你看到这“蓝色 ”两个字时， 你脑子里想到的是“蓝色”还是“红色” ？

语义设计就是要让命名和内容一致，不要扭曲人性。 提升一个层次：我们要让代码文档化。

覆盖和模块

覆盖是无结构，典型的“修补”编程法，甚至当不同需求被引入的前后顺序不同时，会导致不同的代码结构，随意性太强。 模块有设计，有业务含义，可维护性很强。