SassGuidelines 中文

极客学院出版

前言

关于作者

Hugo Giraudel，24 岁，来自法国的前端工程师，现居柏林，目前在 Edenspiekermann 工作。

拥有多年的 Sass 开发经验，开发的项目包括：SassDoc, SitePoint Sass Reference 和 Sass-Compatibility。更多信息，请查阅这里的项目列表。

已出版图书 《CSS3 Pratique du Design Web》（法语版）和《Jump Start Sass》（英语版）。

贡献

Sass指南是我利用空暇时间维持的一个免费项目。不用多说,相信你也能够理解，持续更新、编写文档和其他相关事宜是一件工作量不小的事情。幸运的是，我从许多非常棒的贡献者那里得到了帮助，从而让该规范翻译成了多种语言。请向他们致敬！

现在，如果你想要为它做些贡献，一个很好的建议是：使用社交网络分享它，也可以在 GitHub repository 上提交修正错误的 issue 或者 pull-request。

在开始之前最后但不是最终的一句话：如果你喜欢这份样式指南，如果它对你或你的团队大有裨益，请考虑支持它，从而鼓励我持续迭代不断改进本文！

关于Sass

Sass 的开发文档中如此描述自己：

> Sass 是 CSS 的一个扩展,它使 CSS 的使用起来更加优雅和强大。

Sass 的终极目标是解决 CSS 的缺陷。如我们所知，CSS 并不是一个完美的语言。CSS 虽然简单易学，却也能迅速制造严重的混淆，尤其是在工程浩大的项目中。

这就是 Sass 出现的契机，作为一种元语言，通过提供额外的功能和工具可以改善 CSS 的语法。同时，保留了 CSS 的原有特性。

Sass 存在的关键不是将 CSS 变成一种全功能编程语言，它只是想修复缺陷。正因如此，学习 Sass 如同学习 CSS 一样简单：它只在 CSS 的基础上添加了几个额外功能。

话虽如此，使用这些功能的方式却是多种多样的。有一些是好的，有一些是坏的，还有一些令人费解。这份样式指南就是为了给你一个统一的和历经实践的方式来编写 Sass 代码。

Ruby Sass or LibSass

Sass 的第一次提交还要追溯到距今八年之久的 2006 年底——可见它已经走过了一段漫长的道路。最开始是基于 Ruby，随后便各种版本滋生。其中最成功的要属LibSass（使用 C/C++ 语言编写），它与 Ruby 原生版本具有最佳兼容性。

在 2014 年， Ruby Sass 和 LibSass 团队决定同步推出下一个版本。从那时起，LibSass 开始积极释放版本以校验与 Ruby Sass 的不同，最后剩下的不一致之处被汇总在Sass-Compatibility 项目中。如果你知道两个版本中尚未被发现的不一致之处，请提交一个 issue 使更多开发者了解。

回到选择编译器的问题上来。实际上，这只取决于你。如果是在一个 Ruby on Rails 的项目中，最好使用 Ruby Sass，它在这种情况下是最合适的。当然你也要知道，在未来 Ruby Sass 会一直引领 LibSass 的开发并作为其开发参考。如果你正在查找从 Ruby Sass 切换到 LibSass 的方法，请参考这篇文章。

另一方面，LibSass 更关注于自身与项目之间的整合。如果你想在非 Ruby 项目中使用，比如 NodeJS，node-sass 会是个不错的选择。使用 LibSass 最主要的优势还是因为它的速度，而且比 Ruby Sass 更快。

Sass Or SCSS

有一个剧烈的争论关于 Sass 名字中的含义，并对此有充足的理由：Sass 意味着一个预处理器和它独有的语法。这样很不方便，不是吗？

如你所知，Sass 最初定义的语法，其中决定性的特征是缩进敏感。很快，Sass 的维护者决定提供一个被称为 SCSS（Sassy CSS）的语法以弱化 Sass 和 CSS 之间的差异。

从那时起，Sass（预处理器）开始提供两种不同的语法：Sass（非全大写，please），也被称为**缩进语法**，和 SCSS。使用哪一种语法完全取决于你，两者在功能上是完全等同的，只是在审美上有所偏颇。

Sass 的空白敏感语法通过缩进以摆脱大括号、分号和其他符号，从而实现了简洁凝练的语法格式。与之相比，SCSS 则更容易学习，因为它只是在 CSS 上添加了一点点额外的功能。

我自己更喜欢 SCSS，因为它更接近 CSS 的原生面貌，对开发者来说具有友好性。因此，样式指南全文将使用 SCSS 而不是 Sass 语法格式来演示。你可以通过左侧的可选面板切换到 Sass 的缩进语法。

其他预编译器

忽略其他特性，Sass 对自己的定位首先是一个预处理器。其最主要的竞争对手包括 Less，一个基于 NodeJS 的预处理器，因著名 CSS 框架 Bootstrap（从 v4 版本开始）的使用而声名鹊起。此外还有 Stylus，该预处理器对编写形式无所限制，学习难度稍高。

为什么选择 Sass 胜过其他预处理器？，这始终是一个待解决的问题。就在刚刚，我们还建议在 Ruby 项目中使用 Sass，因为 Sass 初创于 Ruby 并且在 Ruby on Rails 中运行良好。现在随着 LibSass 与 Sass 的逐步接近，上述建议显然已经不再绝对和必须。

我之所以喜欢 Sass 源于它最大程度保留了 CSS 的原生特性。Sass 的设计基于非常坚实的设计原则：大部分的设计顺其自然的来源于核心设计师们的信条，比如添加额外的功能会增加语言的复杂度，但以实用性衡量极具价值的话便得以保留；同时，使用 Sass 来美化一个块级元素，那么美化前后的效果应该是可预见和可推理的。同时，Sass 比其他预处理器更加关注细节。据我所知，核心设计者们非常关心 Sass 与 CSS 在细节上的一致性，并确保所有的常用方式具有高度一致的表现。换言之，Sass 的目标是解决开发者遇到的切实问题，提供高效的函数化解 CSS 的短板。

预处理器之外，我们还需要提及一下后处理器工具。得益于 postCSS 和 CSSNext 项目，它们最近比较受业界瞩目。

PostCSS 常被视为（并不正确）一种“后处理器”工具。从这个名字可以看出，大家认为使用预处理器解析生成 CSS 后，才会让 PostCSS 上场处理 CSS，说实话，做这种工具需要的不是 PostCSS，而只是“处理器”。

PostCSS 可以让开发者访问样式的基本单位，比如选择器、样式和属性值，并使用 JavaScript 处理各种操作，最终生成 CSS。一个典型事例就是基于 PostCSS 创建的 Autoprefixer，该工具根据 CanIUse 提供的浏览器支持度信息给属性添加合理的浏览器前缀。

PostCSS 是一款强大的构建库，可以和任何预处理器（也可以是原生 CSS）共同工作，但是目前 PostCSS 还不易于使用。如果开发者想使用 PostCSS，那么他需要具备一些 JavaScript 的能力来构建项目，此外，PostCSC 的 API 也还处在变化之中。相对来说，Sass 只支持一些用于编写 CSS 的特性，而 PostCSS 则提供了使用 JavaScript 直接访问 CSS 抽象语法树的能力。

简而言之，Sass 简单易学，可以解决大部分的问题，而 PostCSS 上手略难，但威力巨大，所以你应该都去尝试一些这两种工具。实际上，PostCSS 提供了一个官方的 SCSS 解析器来处理 Sass 相关的任务。

感谢 Cory Simmons 对本节所作出的技术支持。

目录

1

简介

为什么需要一个样式指南

一个样式指南并不是一份便于阅读并使代码处于理想状态的文档。它在一个项目的生命周期中是一份关键文档，描述了编写代码的方式和采用这样方式的原因。它可能在小项目中显得有些矫枉过正，但却能在保持代码库整洁，提高可扩展性和可维护性上提供诸多便利。

不用多说相信你也可以理解，参与项目的开发者越多，代码样式指南就越显的必要。与之相同，项目的规模越大，代码样式指南也会越加重要。

Harry Roberts 的 CSS Guidelines 就非常好:

> 样式指南（注意不是视觉风格指南）用于团队是一个很有价值工具：
>
> - 长时间内便于创建和维护项目
> - 便于不同能力的和专业的开发使用
> - 便于任何时间加入团队的不同开发人员
> - 便于新员工培训
> - 便于开发人员创建代码库

免责声明

首先第一件事是：**这不是一份 CSS 样式指南**。本文档不会讨论诸如约定 CSS 类名、模块化开发模式和有关 ID 的疑惑等 CSS 范畴内的问题。本文档中的准则只着眼于处理 Sass 的专有内容。

此外，这份样式指南是我独创的，所以会显得有些**个人主观倾向**。你可以将它看成是我通过多年实践研究出的方法和建议的集合。这也让我有机会接触到少数极具见地的资源，所以一定要浏览一下*扩展阅读*。

显然，这里讲的肯定不是进行 Sass 编程的唯一方式，而且它是否符合你的项目要求还有待检验。

核心原则

最后，如果有一件事是我想从整个样式指南中传授的，那就是：Sass 以简为美，简约至上。

感谢我过去使用 Sass 时傻傻的尝试，比如 bitwise operators、iterators and generators 和 a JSON parser，从而认识到了可以用预处理器来做什么。

同时，CSS 是一门简单的语言，那么 Sass 在书写常规 CSS 的时候就不应该更复杂。KISS principle (Keep It Simple Stupid) 在这里是一个核心原则，甚至在有些情况下要优先于DRY principle (Don’t Repeat Yourself)。

有时候，一点点重复可以更好的保持代码的可维护性，而不是建立一个头重脚轻、臃肿复杂、不可维护的系统。

此外，请允许我再一次引用 Harry Roberts 的观点，**实用胜过完美**。有些时候，你可能会发现自己违背了这里所描述的规则。如果感觉自己的方式有道理，感觉很正确，那就继续做吧。编写代码从来都不是一家之言。

扩展本文

本文中的大部分内容都极具实际参考意义。我学习和使用 Sass 已经有好几年了，其中积累了大量的开发经验，所以对于其他人来说某些观点可能会有一些不适应。

尽管如此，我认为有必要做一些事方便大家自由扩展本文。扩展本文非常简单，有专门的文档来制定代码的编写方式，对于其中的特殊规则，会在下面做出解释。

点击这里可以查看位于 [SassDoc repositoryhref="https://github.com/SassDoc/sassdoc/blob/master/GUIDELINES.md") 上的一个 Styleguide 扩展：

> 这是一个由 Felix Geisendörfer 开发的 Node Styleguide 扩展。这份文档全面覆盖了 Node Styleguide 的内容。

2

语法格式

如果你问我一个样式指南首先要描述什么，我会告诉你：编写代码的通用准则。

当几个开发者在同一项目中编写 CSS 时，迟早会陷入各自为政的境地。编码样式指南通过规范统一的样式，不仅防止了这种混乱现象，也减轻了阅读和维护代码的难度。

概括地说，我们希望做到（受 CSS Guidelines 所启发）：

- 使用两个空格代表缩进，而不是使用tab键；
- 理想上，每行保持为80个字符宽度；
- 正确书写多行CSS规则；
- 有意义的使用空格。

```
// Yep
.foo {
  display: block;
  overflow: hidden;
  padding: 0 1em;
}

// Nope
.foo {
    display: block; overflow: hidden;

    padding: 0 1em;
}
```

字符串

无论你是否相信，字符串在 CSS 和 SCSS 中都占有重要地位。大多数的 CSS 值不是长度就是标识符，所以遵循固定的编程规范处理 Sass 中的字符串是非常重要的一项工作。

编码

为了避免潜在的字符编码问题，强力建议在入口文件 (页 0)中通过 `@charset` 指令使用 UTF-8 编码格式。请确保该指令是文件的第一条语句，并排除其他字符编码声明。

```
@charset 'utf-8';
```

引用

CSS 中不要求字符串必须用引号包裹，甚至是字符串中包含空格的。就拿 `font-family` 属性来说：无论你是否使用引号包裹，CSS 解析器都可以正确解析。

因此，Sass **也不**强制要求字符串必须被引号包裹。更棒的是（你也会如此认为），被引号包裹和没被包裹的一对字符串完全等同（例如，`'abc'` 严格等同于 `abc`）。

话虽如此，不强制要求字符串被引号包裹的毕竟是少数，所以，在 Sass 中**字符串应该始终被单引号(`'`)所包裹**（在 querty 键盘中单引号比双引号更容易输入）。即使不考虑与其他语言的一致性，单是考虑 CSS 的近亲 JavaScript，也有数条理由这么做：

- 如果颜色名不被引号包裹，将会被解析为颜色值，显然这会导致严重问题；
- 大多数的语法高亮机制将会因未被引号包裹的字符串而罢工；
- 提高可读性；
- 没有正当理由不去用引号包裹字符串。

```
// Yep
$direction: 'left';

// Nope
$direction: left;
```

CSS 规范建议，将 `@charset` 指令用双引号包裹起来 才是有效的. 不过，Sass 在编译的时候已经自动修正了相关信息，所以无论何种方式都可以生成正确的代码，即使是只有 `@charset` 。

作为 CSS 的值

CSS 中类似 `initial` 或 `sans-serif` 的标识符无须引用起来。事实上，`font-family: 'sans-serif'` 该声明是错误的，因为 CSS 希望获得的是一个标识符，而不是一个字符串。因此，我们无须引用这些值。

```
// Yep
$font-type: sans-serif;

// Nope
$font-type: 'sans-serif';

// Okay I guess
$font-type: unquote('sans-serif');
```

就像上例这样，我们就可以区别用于 CSS 值的字符串（CSS 标识符）和 Sass 的字符串类型了（比如 map 的值）。

我们没有引用前者，但却使用单引号包裹了它。

包含引号的字符串

如果字符串内包含了一个或多个单引号，一种解决方案就是使用双引号包裹整个字符串，从而避免使用转义字符。

```
// Okay
@warn 'You can\'t do that.';

// Okay
@warn "You can't do that.";
```

URLs

URL 最好也用引号包裹起来，原因和上面所描述一样：

```
// Yep
.foo {
  background-image: url('/images/kittens.jpg');
```

```
}

// Nope
.foo {
  background-image: url(/images/kittens.jpg);
}
```

数字

在 Sass 中，数字类型包括了长度、持续时间、频率、角度等等无单位数字类型。Sass 允许在运行中计算这些度量值。

零值

当数字小于 `1` 时，应该在小数点前写出 `0.` 永远不要显示小数尾部的 `0`。

```
// Yep
.foo {
  padding: 2em;
  opacity: 0.5;
}

// Nope
.foo {
  padding: 2.0em;
  opacity: .5;
}
```

单位

当定义长度时，`0` 后面不需要加单位。

```
// Yep
$length: 0;

// Nope
$length: 0em;
```

注意，该建议只是针对于长度而言，对于类似 `transition-delay` 的时间属性就是不适合的。理论上，如果持续时间的属性值为无单位的 0，那么该属性值就会被认为是无效的。虽然并不是所有的浏览器都这么严格检查属性值，但确实有一些浏览器会这么做。简而言之：只有长度可以使用无单位的 0 作为属性值。

在 Sass 中最常见的错误，是简单地认为单位只是字符串，认为它会被安全的添加到数字后面。这虽然听起来不错，但却不是单位正确的解析方式。可以把单位认为是代数符号，例如，在现实世界中，`5` 英寸乘以 `5` 英寸得到 `25` 英寸。Sass 也适用这样的逻辑。

将一个单位添加给数字的时候，实际上是让该数值乘以 *1* 个*单位*。

```
$value: 42;

// Yep
$length: $value * 1px;

// Nope
$length: $value + px;
```

需要注意的是加上一个 *0unit* 也可以正确解析，但是这种方式在某种程度上会造成一些混乱，所以我更愿意推荐上面的方式。事实上，将一个数字转换为其他兼容单位时，加 0 操作并不会造成错误。更多信息请参考这篇文章.

```
$value: 42 + 0px;
// -> 42px

$value: 1in + 0px;
// -> 1in

$value: 0px + 1in;
// -> 96px
```

这一切最终取决于你想要达到怎样的效果。只要记住，像添加一个字符串一样添加一个单位并不是一种好的处理方式。

要删除一个值的单位，你需要除以*相同类型的* *1* *单位*。

```
$length: 42px;

// Yep
$value: $length / 1px;

// Nope
$value: str-slice($length + unquote(''), 1, 2);
```

给一个数值以字符串形式添加单位的结果是产生一个字符串，同时要防止对数据的额外操作。从一个带有单位的数值中分离数字部分也会产生字符串，但这些都不是你想要的。更多信息请参考这篇文章：Use lengths, not strings。

计算

最高级运算应该始终被包裹在括号中。这么做不仅是为了提高可读性，也是为了防止一些 Sass 强制要求对括号内内容计算的极端情况。

```
// Yep
.foo {
  width: (100% / 3);
}

// Nope
.foo {
  width: 100% / 3;
}
```

Magic numbers

"幻数"，是古老的学校编程给*未命名数值常数*的命名。基本上，它们只是*能工作*™但没有任何逻辑思维的随机数。

相信不用多说你也会理解，**幻数是一场瘟疫，应不惜一切代价以避免**。当你对数值的解析方式无法找到一个合理解释时，你可以对此提交一个内容宽泛的评论，包括你是怎样遇见这个问题以及你认为它应该怎样工作。承认自己不清楚一些机制的解析方式，远比让以后的开发者从零开始弄清它们更有帮助。

```
/**
 * 1. Magic number. This value is the lowest I could find to align the top of
 * `.foo` with its parent. Ideally, we should fix it properly.
 */
.foo {
  top: 0.327em; /* 1 */
}
```

CSS-Tricks 上有一篇文章 讨论了 CSS 中的 magic numbers，建议你阅读一下。

颜色

颜色在 CSS 中占有重要地位。当涉及到操纵色彩时，Sass 通过提供少数的函数功能，最终成为了极具价值的助手。

Sass 非常善于操纵颜色，以下文章都讨论了在 Sass 中对颜色的操作，建议阅读：

- How to Programmatically Go From One Color to Another
- Using Sass to Build Color Palettes
- Dealing with Color Schemes in Sass

颜色格式

为了尽可能简单地使用颜色，我建议颜色格式要按照以下顺序排列：

1. HSL 值；
2. RGB 值；
3. 十六进制（使用小写并尽可能简写）

除非是为了快速开发出原型，否则不建议使用 CSS 颜色关键字。这是因为颜色关键字都是英文单词，对于非英语母语者会造成理解困难。此外，颜色关键字的语义化并不准确，比如 `grey` 比 `darkgrey` 的颜色更深一些；`grey` 和 `gray` 之间的差别也会造成一致性的问题。

HSL 表示法不仅仅是最易于理解的颜色表示方法，而且也便于开发者通过调整色调、饱和度和亮度来惊喜地调整颜色。

相比于 HSL 表示法，RGB 表示法的优势在于表示近似红绿蓝的颜色时更加简洁明了，但是表示红绿蓝的混合色时就不如 HSL 表示法更易于理解了。

最后，十六进制对于人类的思维来说是比较难以理解的，除非必要，否则请优先考虑前几种方式。

```
// Yep
.foo {
  color: hsl(0, 100%, 50%);
}

// Also yep
.foo {
```

```
  color: rgb(255, 0, 0);
}

// Meh
.foo {
  color: #f00;
}

// Nope
.foo {
  color: #FF0000;
}

// Nope
.foo {
  color: red;
}
```

使用 HSL 值或者 RGB 值，通常在逗号（`,`）后面追加一个空格，而不在前后括号（`(`, `)`）和值之间添加空格。

```
// Yep
.foo {
  color: rgba(0, 0, 0, 0.1);
  background: hsl(300, 100%, 100%);
}

// Nope
.foo {
  color: rgba(0,0,0,0.1);
  background: hsl( 300, 100%, 100% );
}
```

颜色和变量

当一个颜色被多次调用时，最好用一个有意义的变量名来保存它。

```
$sass-pink: hsl(330, 50%, 60%);
```

现在，你就可以在任何需要的地方随意使用这个变量了。不过，如果你是在一个主题中使用，我不建议固定的使用这个变量。相反，可以使用另一个标识方式的变量来保存它。

```
$main-theme-color: $sass-pink;
```

这样做可以防止一个主题变化而出现此类结果 `$sass-pink: blue` 。这篇文章 介绍了为什么妥善处理颜色问题如此重要。

变亮和变暗颜色

`lighten` 和 `darken` 函数都是通过增加或者减小HSL中颜色的亮度来实现调节的。基本上，它们就是 `adjust-color` 函数添加了 `$lightness` 参数的别名。

问题是，这些函数经常并不能实现预期的结果。另一方面，通过混合 `白色` 或 `黑色` 实现变量或变暗的 `mix` 函数，是一个不错的方法。

和上述两个函数相比，使用 `mix` 的好处是，当你降低颜色的比例时，它会渐进的接近黑色（或者白色），而 `darken` 和 `lighten` 立即变换颜色到黑色或白色。

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
lighten()	#d379a6	#dfa0bf	#ecc6d9	#f9ecf3						
mix() w/ white	#cb6497	#d175a3	#d786ae	#dc97ba	#e2a9c5	#e8bad1	#edcbdc	#f3dce8	#f9edf4	
darken()	#ad3972	#862d59	#602040	#3a1326	#140606					
mix() w/ dark	#b24a7e	#9e4270	#8a3a62	#763154	#632946	#4f2138	#3b182a	#27101b	#130309	

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
lighten()	#ff7f33	#ff9966	#ffbf99	#ffdfcc						
mix() w/ white	#ff6f19	#ff7f33	#ff8f4c	#ff9f66	#ffaf7f	#ffbf99	#ffcfb2	#ffdfcc	#ffefe5	
darken()	#cc4c00	#993900	#662600	#331300						
mix() w/ dark	#e55500	#cc4c00	#b24200	#993900	#7f2f00	#662600	#4c1c00	#321200	#190900	

> 有关 lighten/darken 和 mix 之间差异的示例来源于KatieK

如果你不想每次都写 `mix` 函数，你可以创建两个易用的 `tint` 和 `shade` (Compass 的一部分)来处理相同的事：

```
/// Slightly lighten a color
/// @access public
/// @param {Color} $color - color to tint
/// @param {Number} $percentage - percentage of `$color` in returned color
/// @return {Color}
@function tint($color, $percentage) {
  @return mix(white, $color, $percentage);
}

/// Slightly darken a color
/// @access public
/// @param {Color} $color - color to shade
/// @param {Number} $percentage - percentage of `$color` in returned color
/// @return {Color}
@function shade($color, $percentage) {
```

```
  @return mix(black, $color, $percentage);
}
```

`scale-color` 函数的设计初衷是为了更流畅地调试属性——以实际的高低为调试基础。它如同 `mix` 一样好用，并且提供了更清晰地调用约定。比例因子并不完全相同。

列表

列表就是 Sass 的数组。列表是一个一维的数据结构（不同于 maps (页 0)），用于保存任意类型的数值（包括列表，从而产生嵌套列表）。

列表需要遵守以下规范：

- 除非列表太长不能写在 80 字符宽度的单行中，否则应该始终单行显示；
- 除非适用于 CSS，否则应该始终使用逗号作为分隔符；
- 要么使用内联形式，要么使用多行形式；
- 始终使用括号包裹；
- 始终不要添加尾部的逗号。

```
// Yep
$font-stack: ('Helvetica', 'Arial', sans-serif);

// Yep
$font-stack: (
  'Helvetica',
  'Arial',
  sans-serif,
);

// Nope
$font-stack: 'Helvetica' 'Arial' sans-serif;

// Nope
$font-stack: 'Helvetica', 'Arial', sans-serif;

// Nope
$font-stack: ('Helvetica', 'Arial', sans-serif,);
```

当需要给列表添加一个新列表项时，请遵守其提供的 API，不要试图手动给列表添加列表项。

```
$shadows: (0 42px 13.37px hotpink);

// Yep
$shadows: append($shadows, $shadow, comma);

// Nope
$shadows: $shadows, $shadow;
```

在这篇文章中介绍了许多合理使用列表的技巧和注意事项。

Maps

在 Sass 中，样式开发者可以使用 map 这种数据结构 —— Sass 团队使 map 可以映射关联数组、哈希表甚至是 Javascript 对象。map 是一种映射任何类型的键值对，包括内嵌类型的 map，但是我不建议使用 map 存储复杂数据类型。

map 的使用应该遵循下述规范：

- 冒号(`:`)之后添加空格；
- 左开括号(`(`)要和冒号 (`:`)写在同一行；
- 如果键是字符串（99% 都是字符串），则**使用括号包裹起来**。
- 每一个键值对单独一行；
- 每一个键值对以逗号(`,`)结尾；
- 为最后一个键值对添加**尾部逗号** (`,`)，方便添加新键值对、删除和重排已有键值对；
- 单独一行书写右闭括号 (`)`)；
- 右闭括号 (`)`)和分号(`;`)之间不使用空格和换行。

示例：

```
// Yep
$breakpoints: (
  'small': 767px,
  'medium': 992px,
  'large': 1200px,
);

// Nope
$breakpoints: ( small: 767px, medium: 992px, large: 12
```

自从 Sass 支持 map 依赖具有很多关于它的文章，我建议你阅读以下三篇：Using Sass Maps, Extra Map functions in Sass, Real Sass, Real Maps.

CSS规则集

在这里，极有可能颠覆每个人对书写 CSS 规则集的认知（根据众多规范整理而成，包括CSS Guidelines）：

- 相关联的选择器写在同一行；不相关联选择器分行书写；
- 最后一个选择器和左开大括号（`{`）中间书写一个空格；
- 每个声明单独一行；
- 冒号（`:`）后添加空格；
- 所有声明的尾部都添加一个分号（`;`）；
- 右闭大括号（`}`）单独一行；
- 右闭大括号（`}`）添加空行。

示例：

```
// Yep
.foo, .foo-bar,
.baz {
  display: block;
  overflow: hidden;
  margin: 0 auto;
}

// Nope
.foo,
.foo-bar, .baz {
    display: block;
    overflow: hidden;
    margin: 0 auto }
```

添加与 CSS 相关的规范时，我们需要注意：

- 本地变量在其他任何变量之前声明，并和其他声明用空行分隔开；
- 不需 `@content` 的混合宏在放在其他声明之前；
- 嵌套选择器在新行声明；
- 需要 `@content` 的混合宏在嵌套选择器后声明；
- 右闭大括号（`}`）上一行无需空行；

示例:

```
.foo, .foo-bar,
.baz {
  $length: 42em;

  @include ellipsis;
  @include size($length);
  display: block;
  overflow: hidden;
  margin: 0 auto;

  &:hover {
    color: red;
  }

  @include respond-to('small') {
    overflow: visible;
  }
}
```

声明顺序

难以想象竟有这么多关于划分 CSS 声明顺序的讨论。具体而言，有如下两派：

- 坚持以字母顺序排列；
- 以类型（`position`，`display`，`colors`，`font`，miscellaneous...）顺序排列；

这两种方式各有利弊。一方面，字母排序方式通俗易懂（至少对于语言中使用拉丁字母的人来说），所以排序的过程完全没有争议。但是，这种排序的结果却十分奇怪，如 `bottom` 和 `top` 竟然彼此不相邻。为什么 `animations` 属性出现在 `display` 属性之前？字母排序方式有太多诸如此类的怪相了。

```
.foo {
  background: black;
  bottom: 0;
  color: white;
  font-weight: bold;
  font-size: 1.5em;
  height: 100px;
  overflow: hidden;
  position: absolute;
  right: 0;
  width: 100px;
}
```

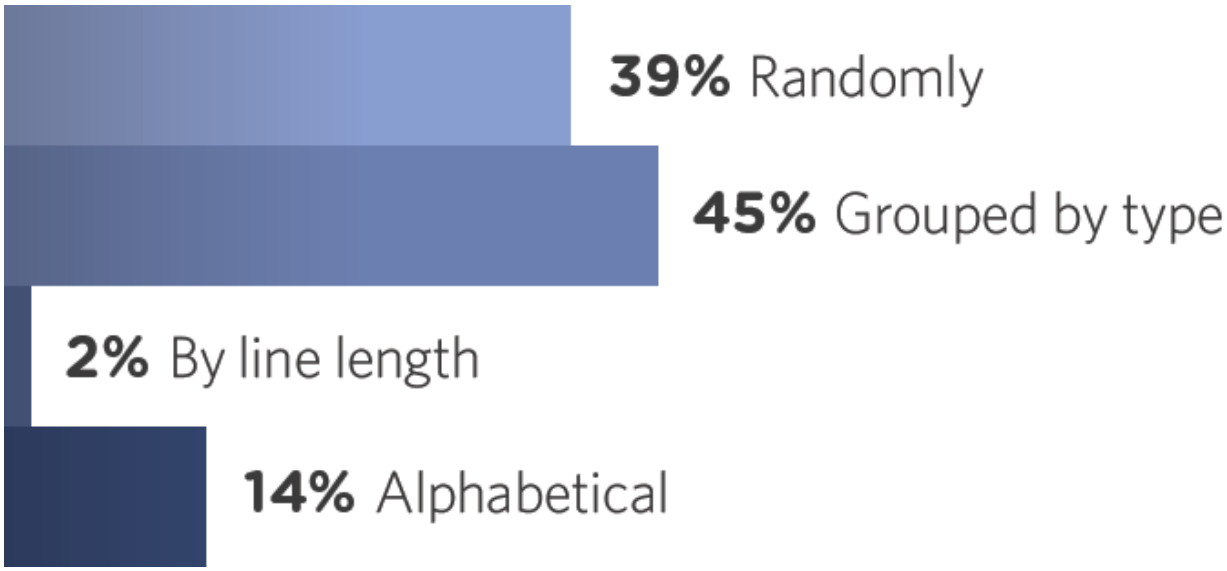
另一方面，按照类型排序则让属性显得更具有意义。每个和字体相关的属性被声明在一起，`top` 和 `bottom` 也结合在一起，最终审阅CSS规则集感觉就像是在读故事。除非你坚持诸如 Idiomatic CSS的规定，不然类型声明顺序可以有更丰富充实的表现。`white-space` 应该放在哪里：font还是dispaly? `overflow` 应该归属何处？如何进行组内排序（如果是字母排序，这岂不成了个笑话）？

```
.foo {
  height: 100px;
  width: 100px;
  overflow: hidden;
  position: absolute;
  bottom: 0;
  right: 0;
  background: black;
  color: white;
  font-weight: bold;
  font-size: 1.5em;
}
```

此外也有其他类型排序的分支，比如Concentric CSS，他看起来相当流行。Concentric CSS 的基础是依赖盒模型定义顺序：由外而内。

```
.foo {
  width: 100px;
  height: 100px;
  position: absolute;
  right: 0;
  bottom: 0;
  background: black;
  overflow: hidden;
  color: white;
  font-weight: bold;
  font-size: 1.5em;
}
```

我必须说我不能对此下任何判定。一份 CSS-Tricks 做的统计报告确认，超过 45% 的开发者使用类型顺序声明，而只有 14% 使用字母顺序。此外还有 39% 的开发者随意而为，这其中就包括我。



> 图表展示了开发者排列 CSS 声明顺序的方式

因此，我不会在此强加规范选择怎样的声明顺序。只要你长久的在自己的样式表中保持一致的风格，那么选择喜欢的声明顺序就可以了（也就说不要太随便）。

最新研究 表明，使用CSS Comb（按照类型排序）对 CSS 进行排序，按类型顺序声明，Gzip 压缩文件大小平均达到 2.7%,而按字母顺序排序压缩的文件大小平均达到 1.3%。

选择器嵌套

Sass 中一个正在被众多开发者滥用的功能，就是**选择器嵌套**。选择器嵌套为样式表作者提供了一个通过局部选择器的相互嵌套实现全局选择的方法。

一般规则

比如下述Sass选择器的嵌套：

```
.foo {
  .bar {
    &:hover {
      color: red;
    }
  }
}
```

生成的 CSS:

```
.foo .bar:hover {
  color: red;
}
```

从 Sass3.3 开始，可以在同一行中使用最近选择器引用(`&`)来实现高级选择器，比如：

```
.foo {
  &-bar {
    color: red;
  }
}
```

生成的 CSS:

```
.foo-bar {
  color: red;
}
```

这种方式通常被用来配合 BEM 全名方式使用，基于原选择器（比如 `.block` ）生成 `.block__element` and `.block--modifier` 选择器。

传说中，使用 `&` 能在当前选择器下产生新的选择器，这样代码库中选择器无法控制，因为他们本身不存在

选择器嵌套最大的问题是将使最终的代码难以阅读。开发者需要花费巨大精力计算不同缩进级别下选择器具体的表现效果。CSS 最终的表现效果往往不是浅显易懂的。

选择器越具体则声明语句越冗长，而且对最近选择器的引用（`&`）也越频繁。在某些时候，出现混淆选择器路径和探索下一级选择器的错误率很高，这非常不值得。

为了防止此类情况，我们今年就 the Inception rule 讨论了很多很多。它建议嵌套不要超过三层，我的一件比较激进，**建议尽量避免使用嵌套**。

虽然我们在下一节看到这条规则有一些例外，但这一观点还是很受欢迎的。更多信息请阅读：《小心嵌套陷阱》和 《避免选择器的过渡嵌套》.

例外

首先，在最外层选择器中嵌套伪类和伪元素是被允许，也是受推荐的。

```
.foo {
  color: red;

  &:hover {
    color: green;
  }

  &::before {
    content: 'pseudo-element';
  }
}
```

使用选择器嵌套选择伪类和伪元素不仅仅有道理的（因为它的处理功能与选择器紧密相关），而且有助于保持总体的一致性。

当然，如果使用类似 `.is-active` 类名来控制当前选择器状态，也可以这样使用选择器嵌套。

```
.foo {
  // …

  &.is-active {
    font-weight: bold;
  }
}
```

这并不是最重要的，当一个元素的样式在另一个容器中有其他指定的样式时，可以使用嵌套选择器让他们保持在同一个地方。

```
.foo {
  // …

  .no-opacity & {
    display: none;
  }
}
```

这所有的一切，有些是无关紧要的细节，关键是要保持一致性。如果你觉得完全有信心搞定选择器嵌套，然后你就使用了选择器嵌套。可你还要确保你的整个团队也能搞定选择器的嵌套。

3

命名约定

在本节，我们不会讨论适用于大规模和可维护的最佳 CSS 命名方案，因为这不仅仅超过了个人的能力范围，也不是一个Sass样式指南可以解决的问题。我个人推荐遵从 CSS Guidelines 的建议。

良好的命名对保持整体代码的一致性和可读性非常重要，在 Sass 中可以命名的地方如下：

- 变量；
- 函数；
- 混合宏。

由于 Sass 占位符遵循和类名相同的命名模式，因此被视为常规的 CSS 选择器，也就在这个列表中故意忽略掉了。

就变量、函数和混合宏的命名而言，我们坚持一些很 *CSS-y* 的风格：**小写连字符分隔**，有意义的命名。

```
$vertical-rhythm-baseline: 1.5rem;

@mixin size($width, $height: $width) {
  // …
}

@function opposite-direction($direction) {
  // …
}
```

常量

如果你恰巧是一个框架开发者或某个库的维护者，你会发现自己正在使用的变量并不需要在所有情况下都进行更新：此时是多么类似一个常量。不幸的是（或者幸运的是？），Sass 不提供任何方式定义这样的实体，所以我们要坚持严格的命名约定来阐述我们的想法。

对于众多编程语言，我建议使用全大写方式书写常量。这不仅是一个由来已久的编程习惯，而且可以很好的与小写连字符变量加以区别。

```
// Yep
$CSS_POSITIONS: (top, right, bottom, left, center);

// Nope
$css-positions: (top, right, bottom, left, center);
```

如果你在 Sass 中使用常量，请参考这篇文章：如何在 Sass 中使用常量.

命名空间

如果你打算分发你的 Sass 代码，比如一个库、框架、栅格系统或者其他的什么，为了防止与其他人的代码发生冲突，你就可能会考虑使用命名空间包裹你所有的变量、函数、混合宏和占位符。

举例来说，如果你参加了一个名为 *Sassy Unicorn* 的项目 —— 这意味着你可以向其贡献代码，你可能会考虑使用 `su-` 作为一个命名空间。这确实非常独特，既不会引发命名冲突，又足够短小而没有书写困难。

```
$su-configuration: ( … );

@function su-rainbow($unicorn) {
  // …
}
```

关于 CSS 的全局命名，Kaelig 写过 一篇非常具有思考价值的文章。

需要注意的是，自动命名空间功能绝对是即将到来的Sass4.0中重构的 `@import` 的一个设计目标。随着即将取得结果，将会越来越少的需要手动命名，最终，手动命名库名实际上会越来越难用。

4

注释

CSS 是一个棘手的语言，充满了骇客行为和古怪的事情。因此，应该大量注释，特别是如果有人打算六个月或一年后要继续阅读和更新这些代码。不要让任何人处于如此境地：**这不是我写的，上帝，为什么会这样**。

CSS 的实现很简单，但我们需要为此付出巨大的注释量。解释如下：

- 一个文件的结构或者作用；
- 规则集的目标；
- 使用幻数背后的目的；
- CSS 声明的原因；
- CSS 声明的顺序；
- 方法执行背后的逻辑思维。

在这里，我可能还遗漏了其他各种各样的缘由。在代码完成之时立即注释，往往只需花费一点时间；而过一阵时间再来为一小段代码注释，则是完全不现实和令人恼怒的。

标示注释

理想上，**任何** CSS 规则集之前都应该使用 C 风格注释来解释 CSS 块的核心。这个注释也要记录对规则集特定部分编号的解释。比如：

```
/**
 * Helper class to truncate and add ellipsis to a string too long for it to fit
 * on a single line.
 * 1. Prevent content from wrapping, forcing it on a single line.
 * 2. Add ellipsis at the end of the line.
 */
.ellipsis {
  white-space: nowrap; /* 1 */
  text-overflow: ellipsis; /* 2 */
  overflow: hidden;
}
```

基本上，任何不能明显看出意义的地方都应该注释，但不要随处注释。记住不要**注释太多**，掌握尺度让每一处注释都有意义。

当注释 Sass 的一个特定部分时，应该使用 Sass 单行注释而不是 C 风格的注释块。这么做将不会输出注释，即使是在开发时的扩展模式。

```
// Add current module to the list of imported modules.
// `!global` flag is required so it actually updates the global variable.
$imported-modules: append($imported-modules, $module)
```

在 CSS 编程指南中的注释一节中也提到，支持这种方式的注释。

文档

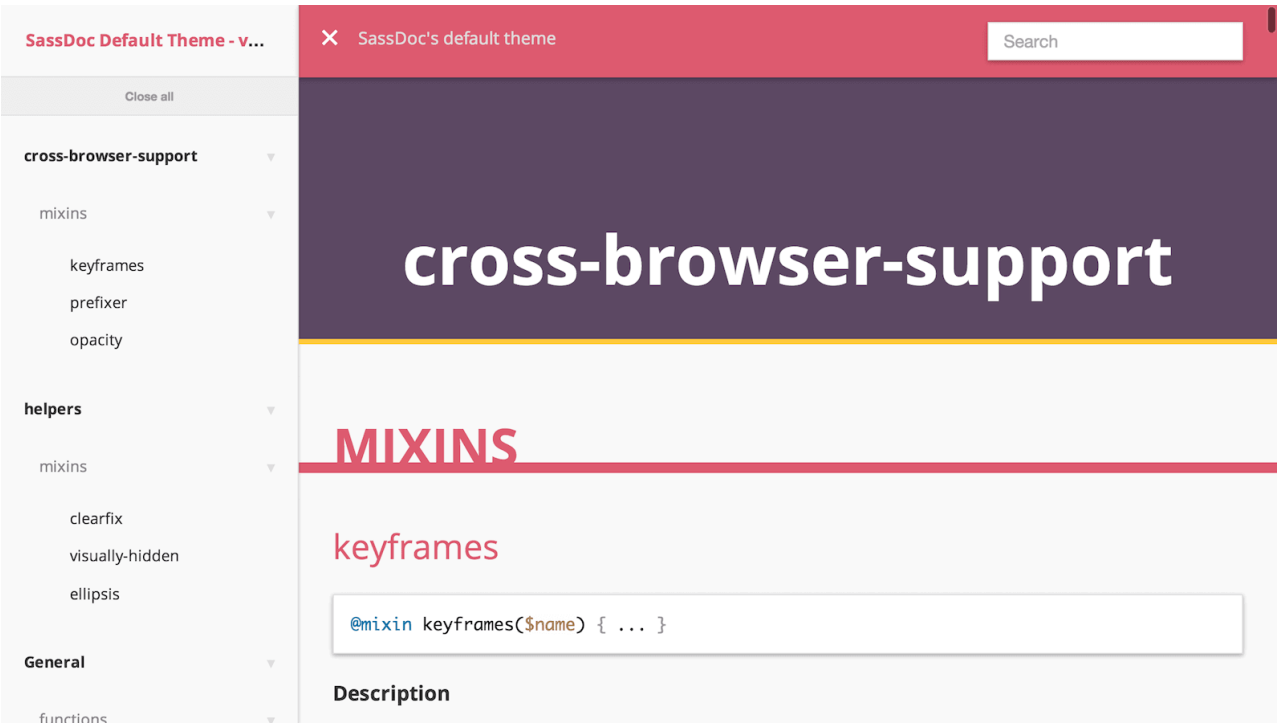
每一个旨在代码库中复用的变量、函数、混合宏和占位符，都应该使用 SassDoc 记录下来作为全部 API 的一部分。

```
/// Vertical rhythm baseline used all over the code base.
/// @type Length
$vertical-rhythm-baseline: 1.5rem;
```

需要三个反斜杠(/)

SassDoc 主要有两个作用：

- 作为公有或私有 API 的一部分，在所有的地方使用一个注释系统强制标准化注释。
- 通过使用任意的 SassDoc 终端(CLI tool, Grunt, Gulp, Broccoli, Node...)，能够生成 API 文档的 HTML 版本。



> 本文档由 SassDoc 生成

这里有一个深入整合 SassDoc 生成文档的例子：

```
/// Mixin helping defining both `width` and `height` simultaneously.
///
/// @author Hugo Giraudel
```

```
///
/// @access public
///
/// @param {Length} $width - Element's `width`
/// @param {Length} $height [$width] - Element's `height`
///
/// @example scss - Usage
///   .foo {
///     @include size(10em);
///   }
///
///   .bar {
///     @include size(100%, 10em);
///   }
///
/// @example css - CSS output
///   .foo {
///     width: 10em;
///     height: 10em;
///   }
///
///   .bar {
///     width: 100%;
///     height: 10em;
///   }
@mixin size($width, $height: $width) {
  width: $width;
  height: $height;
}
```

5

结构

在项目开发周期中，构建一个 CSS 项目可能会是你遇见的最困难的事情之一。构建完成后，保持整体结构的一致性并使所有设置有意义，则更加困难。

幸运的是，使用 CSS 预处理器的一个最主要好处就是可以拆分代码库到几个文件中，而又不会影响性能（正像 CSS 指令 `@import` 的功能）。感谢 Sass 重载了 `@import` 指令，在开发中即使使用大量文件都是安全的（实际上非常推荐），部署时所有文件都会被编译进一个单一样式表。

最重要的是，我无法形容我是多么需要设置大量的文件夹——即使是小项目中。这就像是在家里，你不会将所有的纸片放在同一个盒子中。你可以使用文件夹：一个为房产，一个为银行，一个为账单，等等。没有理由在构架 CSS 项目时不这么做。拆分代码库到多个有意义的文件夹，当你回头来找东西的时候就会发现是那么容易。

有很多受欢迎的构建 CSS 项目的体系结构：OOCSS, Atomic Design, Bootstrap 式, Foundation 式...它们各有优劣，难分伯仲。

我自己使用的方式，与 Jonathan Snook 的 SMACSS 非常相似，其致力于保持代码简洁易见。

我认为，项目之间的结构是极其具体的。你完全可以随意摒弃或调整建议方案，拥有最适合自己需求的体系系统。

组件

首先，在**可以运行**和**运行良好**两种状态之间存在着巨大的差别。其次，CSS 是一个相当容易被混淆的语言。使用的 CSS 越少，工作会越愉快。没人想处理兆字节量的 CSS 代码。保持样式表简短而高效，就不会有诸多诡异。将接口视为组件的集合来使用往往是非常棒的思维。

组件可以是任意的，前提是遵循以下规范：

- 可以做一件事，只做一件；
- 在整个项目中可以重用，具有可复用性；
- 各自独立。

例如，搜索框就应该被视为一个组件，可以在不同位置、不同页面、多种环境下重复使用。它不应该受限于 DOM 中的位置（页脚、侧边栏、主内容区...）。

几乎所有的接口都可以被视为小组件，而且强烈建议坚持这种模式。这不仅仅会精简整个项目中 CSS 的代码量，而且也会比维护一个到处无逻辑的烂摊子容易得多。

组件结构

理想情况下，每个组件都应该拥有自己的文件夹（存在于 `components` 文件之下，详见7-1 模式 (页 0)），比如 `components/_button.scss`。每个组件的样式应该包含以下内容：

- 组件本身的样式
- 和组件样式有关的变量、修饰器以及状态
- 如有需要，设置组件的子级样式

如果你希望可以定制组件的主题（主题文件置于 `themes/` 文件夹之内），可以限制样式中可以被修改的种类，比如尺寸、内间距、外间距以及对齐方式等等，可以开放颜色、阴影、字体、背景等方面的样式。

一个组件文件内可以存在与该组件密切相关的变量、占位符、混合宏甚至是函数，但是要牢记，应该避免对其他组件样式的引用，否则将会让项目整体的依赖关系变得难以维护。

下面是一个 Button 组件的示例：

```
// Button-specific variables
$button-color: $secondary-color;

// … include any button-specific:
// - mixins
// - placeholders
// - functions

/**
 * Buttons
 */
.button {
  @include vertical-rhythm;
  display: block;
  padding: 1rem;
  color: $button-color;
  // … etc.

  /**
   * Inlined buttons on large screens
   */
  @include respond-to('medium') {
    display: inline-block;
  }
```

```
}

/**
 * Icons within buttons
 */
.button > svg {
  fill: currentcolor;
  // … etc.
}

/**
 * Inline button
 */
.button--inline {
  display: inline-block;
}
```

感谢 David Khourshid 对本节做出的技术支持。

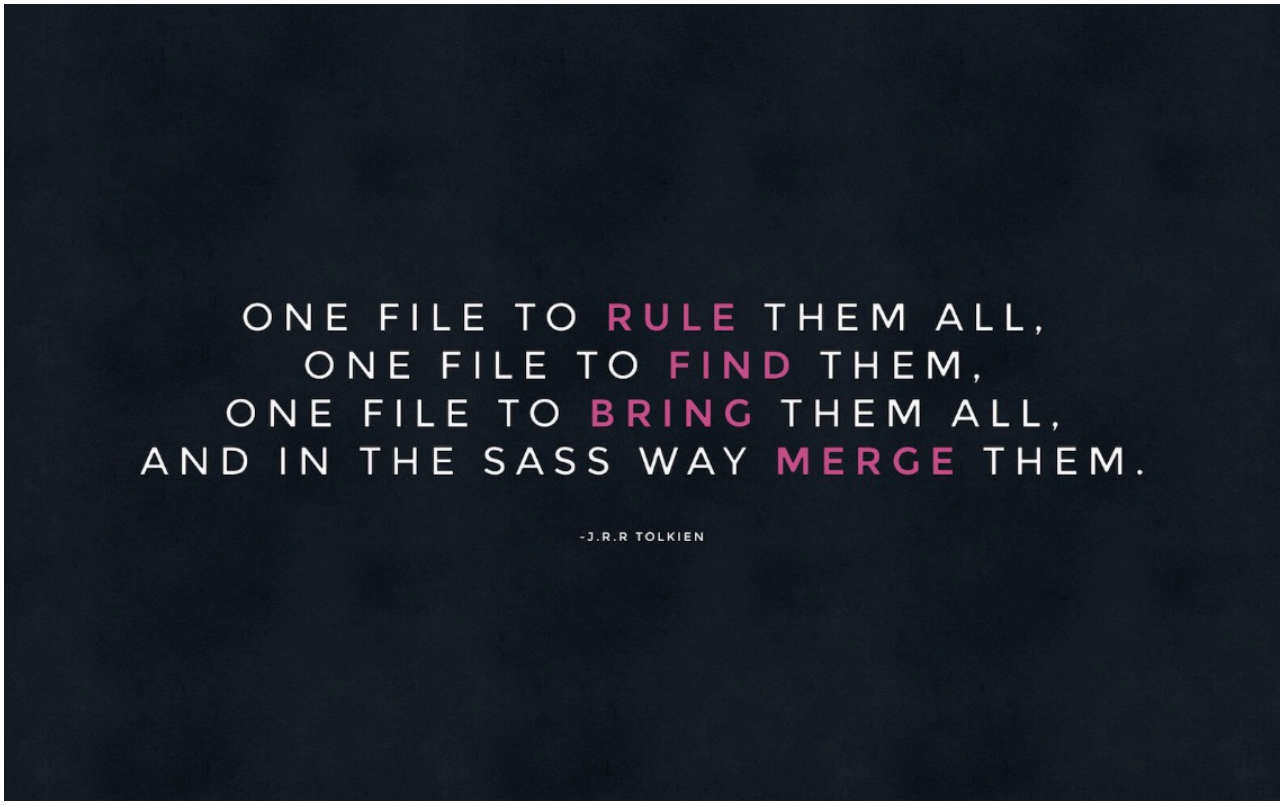
7-1模式

回到结构这个话题上来，好吗？通常我使用自称为 **7-1 模式**的结构：7 个文件夹，1 个文件。基本上，你需要将所有的部件放进 7 个不同的文件夹和一个位于根目录的文件（通常命名为 `main.scss` ）中——这个文件编译时会引用所有文件夹而形成一个 CSS 样式表。

- `abstracts/`
- `base/`
- `components/`
- `layout/`
- `pages/`
- `themes/`
- `vendors/`

当然还有它：

- `main.scss`



> 壁纸来源自 Julien He

理想情况下，目录层次如下所示：

```
sass/
|
|- abstracts/
|   |- _variables.scss    # Sass Variables
|   |- _functions.scss    # Sass Functions
|   |- _mixins.scss       # Sass Mixins
|   |- _placeholders.scss # Sass Placeholders
|
|- base/
|   |- _reset.scss        # Reset/normalize
|   |- _typography.scss   # Typography rules
|   …                     # Etc.
|
|- components/
|   |- _buttons.scss      # Buttons
|   |- _carousel.scss     # Carousel
|   |- _cover.scss        # Cover
|   |- _dropdown.scss     # Dropdown
|   …                     # Etc.
|
|- layout/
|   |- _navigation.scss   # Navigation
|   |- _grid.scss         # Grid system
|   |- _header.scss       # Header
|   |- _footer.scss       # Footer
|   |- _sidebar.scss      # Sidebar
|   |- _forms.scss        # Forms
|   …                     # Etc.
|
|- pages/
|   |- _home.scss         # Home specific styles
|   |- _contact.scss      # Contact specific styles
|   …                     # Etc.
|
|- themes/
|   |- _theme.scss        # Default theme
|   |- _admin.scss        # Admin theme
|   …                     # Etc.
|
|- vendors/
|   |- _bootstrap.scss    # Bootstrap
```

```
|   |- _jquery-ui.scss    # jQuery UI
|   …                     # Etc.
|
`- main.scss              # Main Sass file
```

文件命名要遵循如上统一的命名规则：使用连字符界定。

Base文件夹

`base/` 文件夹存放项目中的模板文件。在这里，可以找到重置文件、排版规范文件或者一个样式表——定义一些 HTML 元素公认的标准样式（我喜欢命名为 `_base.scss` ）。

- `_base.scss`
- `_reset.scss`
- `_typography.scss`

如果你的项目中使用了*大量的* CSS 动画，那么你有必要考虑添加一个 `\_animations.scss` 文件来统一管理这些动画。如果只是偶尔使用一些动画，也可以将这些动画融入到调用它们的文件中。

Layout文件夹

`layout/` 文件夹存放构建网站或者应用程序使用到的布局部分。该文件夹存放网站主体（头部、尾部、导航栏、侧边栏...）的样式表、栅格系统甚至是所有表单的 CSS 样式。

- `_grid.scss`
- `_header.scss`
- `_footer.scss`
- `_sidebar.scss`
- `_forms.scss`
- `_navigation.scss`

`layout/` 文件夹也会被称为 `partials/`，具体使用情况取决于个人喜好。

Components文件夹

对于小型组件来说，有一个 `components/` 文件夹来存放。相对于 `layout/` 的*宏观*（定义全局线框结构），`components/` 更专注于局部组件。该文件夹包含各类具体模块，基本上是所有的独立模块，比如一个滑块、一个加载块、一个部件……由于整个网站或应用程序主要由微型模块构成，`components/` 中往往有大量文件。

- `_media.scss`
- `_carousel.scss`
- `_thumbnails.scss`

`components/` 文件夹也会被称为 `modules/`，具体使用情况取决于个人喜好。

Pages文件夹

如果页面有特定的样式，最好将该样式文件放进 `pages/` 文件夹并用页面名字。例如，主页通常具有独特的样式，因此可以在 `pages/` 下包含一个 `_home.scss` 以实现需求。

- `_home.scss`
- `_contact.scss`

取决于各自的开发流程，这些文件可以使用你自己的前缀命名，避免在最终样式表中与他人的样式表发生合并。一切完全取决于你。

Themes文件夹

在大型网站和应用程序中，往往有多种主题。虽有多种方式管理这些主题，但是我个人更喜欢把它们存放在 `themes/` 文件夹中。

- `_theme.scss`
- `_admin.scss`

这个文件夹与项目的具体实现有密切关系，并且在许多项目中是并不存在的。

Abstracts 文件夹

`abstracts/` 文件夹包含了整个项目中使用到的 Sass 辅助工具，这里存放着每一个全局变量、函数、混合宏和占位符。

该文件夹的经验法则是，编译后这里不应该输出任何 CSS，单纯的只是一些 Sass 辅助工具。

- `_variables.scss`
- `_mixins.scss`
- `_functions.scss`
- `_placeholders.scss`

当项目体量庞大工具复杂时，通过主题而不是类型分类整理可能更有帮助，比如排版（`_typography.scss`）、主题（`_theming.scss`）等。每一个文件都包含所有的相关信息：变量、函数、混合宏和占位符。这样做可以让维护更加单，特别针对于文件较长的情况。

`abstracts/` 文件夹也会被称为 `helpers/` 或 `utilities`，具体使用情况取决于个人喜好。

Vendors文件夹

最后但并非最终的是，大多数的项目都有一个 `vendors/` 文件夹，用来存放所有外部库和框架（Normalize, Bootstrap, jQueryUI, FancyCarouselSliderjQueryPowered……）的 CSS 文件。将这些文件放在同一个文件中是一个很好的说明方式:"嘿，这些不是我的代码，无关我的责任。"

- `_normalize.scss`
- `_bootstrap.scss`
- `_jquery-ui.scss`
- `_select2.scss`

如果你重写了任何库或框架的部分，建议设置第 8 个文件夹 `vendors-extensions/` 来存放，并使用相同的名字命名。

例如，`vendors-extensions/_boostrap.scss` 文件存放所有重写 Bootstrap 默认 CSS 之后的 CSS 规则。这是为了避免在原库或者框架文件中进行二次编辑——显然不是好方法。

入口文件

主文件（通常写作 `main.scss`）应该是整个代码库中唯一开头不用下划线命名的 Sass 文件。除 `@import` 和注释外，该文件不应该包含任何其他代码。

文件应该按照存在的位置顺序依次被引用进来：

1. `abstracts/`
2. `vendors/`
3. `base/`
4. `layout/`
5. `components/`
6. `pages/`
7. `themes/`

为了保持可读性，主文件应遵守如下准则：

- 每个 `@import` 引用一个文件；
- 每个 `@import` 单独一行；
- 从相同文件夹中引入的文件之间不用空行；
- 从不同文件夹中引入的文件之间用空行分隔；
- 忽略文件扩展名和下划线前缀。

```
@import 'abstracts/variables';
@import 'abstracts/functions';
@import 'abstracts/mixins';
@import 'abstracts/placeholders';

@import 'vendors/bootstrap';
@import 'vendors/jquery-ui';

@import 'base/reset';
@import 'base/typography';

@import 'layout/navigation';
@import 'layout/grid';
@import 'layout/header';
```

```
@import 'layout/footer';
@import 'layout/sidebar';
@import 'layout/forms';

@import 'components/buttons';
@import 'components/carousel';
@import 'components/cover';
@import 'components/dropdown';

@import 'pages/home';
@import 'pages/contact';

@import 'themes/theme';
@import 'themes/admin';
```

这里还有另一种引入的有效方式。令人高兴的是，它使文件更具有可读性；令人沮丧的是，更新时会有些麻烦。不管怎么说，由你决定哪一个最好，这没有任何问题。 对于这种方式，主要文件应遵守如下准则：

- 每个文件夹只使用一个 `@import`
- 每个 `@import` 之后都断行
- 每个文件占一行
- 新的文件跟在最后的文件夹后面
- 文件扩展名都可以省略

```
@import
  'abstracts/variables',
  'abstracts/functions',
  'abstracts/mixins',
  'abstracts/placeholders';

@import
  'vendors/bootstrap',
  'vendors/jquery-ui';

@import
  'base/reset',
  'base/typography';

@import
  'layout/navigation',
  'layout/grid',
  'layout/header',
  'layout/footer',
```

```
  'layout/sidebar',
  'layout/forms';

@import
  'components/buttons',
  'components/carousel',
  'components/cover',
  'components/dropdown';

@import
  'pages/home',
  'pages/contact';

@import
  'themes/theme',
  'themes/admin';
```

关于 Globbing

在计算机编程中，通配符扩展模式通常使用通配符来匹配多个文件名，比如 `*.scss`，其工作机制是通过表达式而不是文件名列表来匹配文件组。在 Sass 中，可以在入口文件中通过通配符扩展的形式导入其他文件，导入后的入口文件类似如下所示：

```
@import 'abstracts/*';
@import 'vendors/*';
@import 'base/*';
@import 'layout/*';
@import 'components/*';
@import 'pages/*';
@import 'themes/*';
```

Sass 并不直接支持通配符扩展的机制，这是因为 CSS 样式是对声明顺序非常敏感的，当我们使用通配符扩展的形式导入文件时，文件通常按照字典序导入，这种方式无法控制文件的导入顺序，继而会引起样式的错乱。

也就是说，在一个严格基于组件构成的架构中，必须十分注意组件之间的样式顺序，避免遗漏和错误覆盖任何样式。所以必须保证文件顺序对样式没有影响，方能使用通配符扩展模式。使用通配符扩展模式最大的好处就是无需再花费时间处理入口文件中文件的增加和删除。

在 Ruby Sass 中有一个 sass-globbing 包可以用于解析通配符扩展机制。如果使用的是 node-sass，可以使用 Node.js 或者构建工具（Gulp，Grunt等 等）来解析通配符扩展机制。

Shame 文件

另一个有意思的方面，由业内已流行的 Harry Roberts, Dave Rupert 和 Chris Coyier 引起的，那就是将所有的CSS声明、Hack行为和我们不支持的行为放入一个 shame file。该文件命名为 `_shame.scss`，在所有文件之后被引用，放在所有样式表的最后。

```
/**
 * Nav specificity fix.
 *
 * Someone used an ID in the header code (`#header a {}`) which trumps the
 * nav selectors (`.site-nav a {}`). Use !important to override it until I
 * have time to refactor the header stuff.
 */
.site-nav a {
    color: #BADA55 !important;
}
```

6

响应式设计和断点

无需在此对响应式网页设计多做介绍，它已经无所不在了。但是你可能会疑惑：**为什么在 Sass 样式指南中会有关于响应式网页设计的章节？**正因为有诸多降低断点使用难度的方式，所以我认为在这里列出来会是个不错的主意。

命名断点

我认为使用媒体查询而不针对特定设备是安全可靠的做法。特别要指出的是，不应该赞成专门针对 iPad 或黑莓手机设计媒体查询。媒体查询应该关注屏幕尺寸，直到当前设计遇到阻断而将所有工作过继给下一个媒体查询。

与之相同的观点是，断点不应该用设备来命名，而应使用更通用的方式。特别是，现在有一些手机比平板更大，而有一些平板比电脑更大……

```
// Yep
$breakpoints: (
  'medium': (min-width: 800px),
  'large': (min-width: 1000px),
  'huge': (min-width: 1200px),
);

// Nope
$breakpoints: (
  'tablet': (min-width: 800px),
  'computer': (min-width: 1000px),
  'tv': (min-width: 1200px),
);
```

就此来说，任何不与特定设备关联而表达清晰的命名约定，都会因其广泛的通用性获得认可。

```
$breakpoints: (
  'seed': (min-width: 800px),
  'sprout': (min-width: 1000px),
  'plant': (min-width: 1200px),
);
```

上面的示例使用了嵌套的 map，但这并不是强制或绝对的，你完全可以使用字符串来代替（比如 `'(min-width: 800px)'`）。

断点管理器

一旦用自己钟意的方式命名完断点，就需要有一种方式在实际的媒体查询中使用它。有太多方法可以做这件事，我自己非常乐意使用 `map-get()` 函数读取断点地图的方法。这套系统简洁而高效。

```
/// Responsive breakpoint manager
/// @access public
/// @param {String} $breakpoint - Breakpoint
/// @requires $breakpoints
@mixin respond-to($breakpoint) {
  $raw-query: map-get($breakpoints, $breakpoint);

  @if $raw-query {
    $query: if(
      type-of($raw-query) == 'string',
      unquote($raw-query),
      inspect($raw-query)
    );

    @media #{$query} {
      @content;
    }
  } @else {
    @error 'No value found for `#{$breakpoint}`. '
         + 'Please make sure it is defined in `$breakpoints` map.';
  }
}
```

更多有关 Sass 中媒体查询的信息，请参考 SitePoint 和 CSS-Tricks 中优秀的实践文章。

媒体查询用法

就在不久之前，有一个关于应该在哪里书写媒体查询的热门讨论：媒体查询是应该与选择器写在一起（Sass 允许这种方式），还是要彻底地分离开？我想说我是**媒体查询紧贴选择器**方式的狂热捍卫者，并且认为这会和**组件**一样表现得很棒。

```
.foo {
  color: red;

  @include respond-to('medium') {
    color: blue;
  }
}
```

生成结果：

```
.foo {
  color: red;
}

@media (min-width: 800px) {
  .foo {
    color: blue;
  }
}
```

可能你已经了解到，这种习惯会导致 CSS 输出文件中出现重复的媒体查询语句。不过测试了和下面的话认为一旦 Gzip（或者其他相同软件）完成压缩就不会有什么问题：

> ……我们反复测试了贴合与分离两种媒体查询方式对性能的影响，结论是即使在最差情况下也没有明显差异，而在最好情况下差异更是少之又少。
>
> — Sam Richards 关于Breakpoint 的看法

如果现在你仍担心媒体查询的副本问题，你可以使用工具来合并它们，比如这个 gem，但是我有必要警告你移动相关 CSS 代码可能会有副作用。 是否了解资源顺序是非常重要的。

7

变量

变量是任何编程语言的精髓。变量让值得以重用，避免了一遍遍地复制副本。最重要的是，使用变量让更新一个值变得很方便。不用查找、替换，更不用手动检索。

然而CSS是一个将所有鸡蛋装在一个大篮子中的语言，不同于其他语言，这里没有真正的作用域。因此，我们需要十分重视由于添加变量而引起的冲突。

我的建议只适用于创建变量并感觉确有必要的情况下。不要为了某些骇客行为而声明新变量，这丝毫没有作用。只有满足所有下述标准时方可创建新变量：

- 该值至少重复出现了两次；
- 该值至少可能会被更新一次；
- 该值所有的表现都与变量有关（非巧合）。

基本上，没有理由声明一个永远不需要更新或者只在单一地方使用变量。

作用域

Sass 中变量的作用域在过去几年已经发生了一些改变。直到最近，规则集和其他范围内声明变量的作用域才默认为本地。如果已经存在同名的全局变量，则局部变量覆盖全局变量。从 Sass 3.4 版本开始，Sass 已经可以正确处理作用域的概念，并通过创建一个新的局部变量来代替。

本部分讨论下**全局变量的影子**。当在局部范围内（选择器内、函数内、混合宏内……）声明一个已经存在于全局范围内的变量时，局部变量就成为了全局变量的影子。基本上，局部变量只会在局部范围内覆盖全局变量。

以下代码片可以解析**变量影子**的概念。

```
// Initialize a global variable at root level.
$variable: 'initial value';

// Create a mixin that overrides that global variable.
@mixin global-variable-overriding {
  $variable: 'mixin value' !global;
}

.local-scope::before {
  // Create a local variable that shadows the global one.
  $variable: 'local value';

  // Include the mixin: it overrides the global variable.
  @include global-variable-overriding;

  // Print the variable's value.
  // It is the **local** one, since it shadows the global one.
  content: $variable;
}

// Print the variable in another selector that does no shadowing.
// It is the **global** one, as expected.
.other-local-scope::before {
  content: $variable;
}
```

!default 标识符

如果创建一个库、框架、栅格系统甚至任何的 Sass 片段，是为了分发经验或者被其他开发者使用，那么与之配置的所有变量都应该使用 `!default` 标志来定义，方便其他开发者重写变量。

```
$baseline: 1em !default;
```

多亏如此，开发者才能在引入你的库之前定义自用的 `$baseline`，引入后又不必担心自己的值被重定义了。

```
// Developer’ s own variable
$baseline: 2em;

// Your library declaring `$baseline`
@import ’your-library’;

// $baseline == 2em;
```

!global 标识符

!global 标志应该只在局部范围的全局变量被覆盖时使用。定义根级别的变量时，!global 标志应该省略。

```
// Yep
$baseline: 2em;

// Nope
$baseline: 2em !global;
```

多变量或maps

使用 maps 比使用多个不同的变量有明显优势。最重要的优势就是 map 的遍历功能，这在多个不同变量中是不可能实现的。

另一个支持使用 map 的原因，是它可以创建 `map-get()` 函数以提供友好 API 的功能。比如，思考一下下述 Sass 代码：

```
/// Z-indexes map, gathering all Z layers of the application
/// @access private
/// @type Map
/// @prop {String} key - Layer's name
/// @prop {Number} value - Z value mapped to the key
$z-indexes: (
  'modal': 5000,
  'dropdown': 4000,
  'default': 1,
  'below': -1,
);

/// Get a z-index value from a layer name
/// @access public
/// @param {String} $layer - Layer's name
/// @return {Number}
/// @require $z-indexes
@function z($layer) {
  @return map-get($z-indexes, $layer);
}
```

8

扩展

`@extend` 指令是 Sass 中既强大易于误解的指令。该指令作为一个警示，告知 Sass 对选择器 A 的样式化正好存在与选择器B共通的地方。不用多说，这是书写模块化 CSS 的好助手。

实际上，`@extend` 的实际作用是维护继承前后选择器之间的。这也就是说：

- 选择器是受限的（比如：在 `.foo > .bar` 中 `.bar` 必须有一个父级 `.foo`）；
- 选择器所收到的限制会传递给后续继承的选择器上（比如 `.baz { @extend .bar; }` 会生成 `.foo > .bar, .foo > .baz`）；
- 被继承的选择器会被要继承的选择器匹配。

要理解这些现象，最直接的方式就是看看当没有限制时继承样式所产生的选择器数量剧增的结果。如果 `.baz .qux` 继承了 `.foo .bar`，那么就会生成 `.foo .baz .qux` 或 `.baz .foo .qux`，这是因为 `.foo` 和 `.baz` 是常见的选择器，它们可以成为父级、祖父级等等。

始终使用选择器占位符定义选择器之间的关系，而不是类名，这能让你更加轻松地更换命名约定。此外，因为选择器之间的关系只被定义在了占位符中，所以很少会产生意料之外的选择器。

当继承 `.class` 或 `%placeholder` 时，如果父类和子类是同一类型，那么建议只使用 `@extend` 来实现，比如 `.error` 是 `.warning` 的一种，那么 `.error` 就可以通过 `@extend .warning` 来实现。

```
%button {
  display: inline-block;
  // … button styles

  // Relationship: a %button that is a child of a %modal
  %modal > & {
    display: block;
  }
}

.button {
  @extend %button;
}

// Yep
.modal {
  @extend %modal;
}

// Nope
.modal {
  @extend %modal;
```

```
  > .button {
    @extend %button;
  }
}
```

扩展选择器在许多情境下是有用和值得的。始终牢记下面这些规则，谨慎使用 `@extend` 指令：

- 优先继承 `%placeholders`，而不是具体的选择器；
- 当继承 `.class` 时，只继承单一的 `.class`，不要使用[复杂选择器]complex selector；
- 尽可能少的继承自 `%placeholders`；
- 避免继承常见的父类选择器（比如： `.foo .bar` ）或者是常见的相邻选择器（比如： `.foo ~ .bar` ），否则会让选择器的数量急速增加。

通常来说， `@extend` 有助于减少文件体积大小，因为它的操作本质上是合并选择器而不是赋值样式。话虽如此，当你使用 Gzip 压缩文件时， `@extend` 对于文件压缩的好处几乎是可以忽略的。

这也就是说，如果你不能使用类似 Gzip 的工具，那么就可以考虑使用 `@extend` 来减少不必要的重复，特别是当样式文件的大小成为性能瓶颈的时候，这种方式尤为有效。

继承和媒体查询

应该只在同一个媒体查询作用域下继承选择器，将媒体查询视为一种对作用域的限制。

```
%foo {
  content: 'foo';
}

// Nope
@media print {
  .bar {
    // This doesn't work. Worse: it crashes.
    @extend %foo;
  }
}

// Yep
@media print {
  .bar {
    @at-root (without: media) {
      @extend %foo;
    }
  }
}

// Yep
%foo {
  content: 'foo';

  &-print {
    @media print {
      content: 'foo print';
    }
  }
}

@media print {
  .bar {
    @extend %foo-print;
  }
}
```

有关 `@extend` 的好与坏，开发者们之间的观点大有不同，你可以阅读以下文章了解其中的利弊：

- What Nobody Told you About Sass Extend
- Why You Should Avoid Extend
- Don’t Over Extend Yourself

总而言之，我建议只将 `@extend` 用于维护选择器之间的关系。如果两个选择器是类似的，那么最好使用 `@extend`；如果它们之间没有关系，只是具有相同的样式，那么使用 `@mixin` 会更好。更多有关两者的用法，请看这篇文章：When to use extend and when to use a mixin.

感谢 David Khourshid 对本节提供的技术支持。

9

混合宏

混合宏是整个 Sass 语言中最常用的功能之一。这是重用和减少重复组件的关键。这么做有很棒的原因：混合宏允许开发者在样式表中定义可复用样式，减少了对非语义类的需求，比如 `.float-left` 。

它们可以包含所有的 CSS 规则，并且在 Sass 文档允许的任何地方都表现良好。它们甚至可以像函数一样接受参数。不用多说，充满了无尽的可能。

不过我有必要提醒你滥用混合宏的破坏力量。再次重申一遍，使用混合宏的关键是**简洁**。建立混入大量逻辑而极具力量的混合宏看上去确实很有诱惑力。这就是所谓的过度开发，大多数开发者常常因此陷入困境。不要过度逻辑化你的代码，尽量保持一切简洁。如果一个混合宏最后超过了 20 行，那么它应该被分离成更小的块甚至是重建。

基础

话虽如此，混合宏确实非常有用，你应该学习使用它。经验告诉我们，如果你发现有一组 CSS 属性经常因同一个原因一起出现（非巧合），那么你就可以使用混合宏来代替。比如Nicolas Gallagher 的清除浮动应当放入一个混合宏的实例。

```
/// Helper to clear inner floats
/// @author Nicolas Gallagher
/// @link http://nicolasgallagher.com/micro-clearfix-hack/ Micro Clearfix
@mixin clearfix {
  &::after {
    content: '';
    display: table;
    clear: both;
  }
}
```

另一个有效的实例是通过在混合宏中绑定 `width` 和 `height` 属性，可以为元素设置宽高。这样不仅会淡化不同类型代码间的差异，也便于阅读。

```
/// Helper to size an element
/// @author Hugo Giraudel
/// @param {Length} $width
/// @param {Length} $height
@mixin size($width, $height: $width) {
  width: $width;
  height: $height;
}
```

更多复杂示例可以参考：《使用 Sass 混合宏创建三角形》，《使用混合宏创建长阴影》 以及 《使用混合宏为低版本浏览器创建线性渐变》。

无参混合宏

有时候我们使用混合宏只是为了避免重复声明相同的样式，这种情况下，往往不需要传递参数。所以，为了简洁起见，我们可以删除圆括号，使用 `@include` 关键字来表示当前行调用了混合宏。

```
// Yep
.foo {
  @include center;
}

// Nope
.foo {
  @include center();
}
```

参数列表

当混合宏需要处理数量不明的参数时，通常使用 `arglist` 而不是列表。可以认为 `arglist` 是 Sass 中隐藏而未被记录的第八个数据类型，通常当需要任意数量参数的时候，被隐式使用到参数中含有 `...` 标志的混合宏和函数中。

```
@mixin shadows($shadows...) {
  // type-of($shadows) == 'arglist'
  // …
}
```

现在，当要建立一个接收多个参数（默认为 3 或者更多）的混合宏时，在将它们合并为列表或者 map 之前，要反复考量这样做是否比一个个的单独存在更易于使用。

Sass 的混合宏和函数声明非常智能，你只需给函数/混合宏一个列表或 map，它会自动解析为一系列的参数。

```
@mixin dummy($a, $b, $c) {
  // …
}

// Yep
@include dummy(true, 42, 'kittens');

// Yep but nope
$params: (true, 42, 'kittens');
$value: dummy(nth($params, 1), nth($params, 2), nth($params, 3));

// Yep
$params: (true, 42, 'kittens');
@include dummy($params...);

// Yep
$params: (
  'c': 'kittens',
  'a': true,
  'b': 42,
);
@include dummy($params...);
```

更多有关多参数、列表参数的信息请参考这篇文章：Sass 中的不定参数和参数列表。

混合宏和浏览器前缀

通过使用自定义混合宏来处理 CSS 中未被支持或部分支持的浏览器前缀，是非常有吸引力的一种做法。但我们不希望这么做。首先，如果你可以使用 Autoprefixer，那就使用它。它会从你的项目中移除Sass代码，会一直更新并一定会进行比你手动添加前缀更棒的处理。

不幸的是，Autoprefixer 并不是总被支持的。如果你使用 Bourbon 或 Compass，你可能就已经知道它们都提供了一个混合宏的集合，用来为你处理浏览器前缀，那就用它们吧。

如果你不能使用 Autoprefixe，甚至也不能使用 Bourbon 和 Compass，那么接下来唯一的方式，就是使用自己的混合宏处理带有前缀的 CSS 属性。但是，请不要为每个属性建立混合宏，更不要无脑输出每个浏览器的前缀（有些根本就不存在）。

```
// Nope
@mixin transform($value) {
  -webkit-transform: $value;
  -moz-transform: $value;
  transform: $value;
}
```

比较好的做法是

```
/// Mixin helper to output vendor prefixes
/// @access public
/// @author HugoGiraudel
/// @param {String} $property - Unprefixed CSS property
/// @param {*} $value - Raw CSS value
/// @param {List} $prefixes - List of prefixes to output
@mixin prefix($property, $value, $prefixes: ()) {
  @each $prefix in $prefixes {
    -#{$prefix}-#{$property}: $value;
  }

  #{$property}: $value;
}
```

然后就可以非常简单地使用混合宏了：

```
.foo {
  @include prefix(transform, rotate(90deg), ('webkit', 'ms'));
}
```

请记住，这是一个糟糕的解决方案。例如，他不能处理那些需要复杂的前缀，比如 `flexbox`。在这个意义上说，使用 Autoprefixer 是一个更好地选择。

10

条件语句

你可能早已知晓，Sass 通过 `@if` 和 `@else` 指令提供了条件语句。除非你的代码中有偏复杂的逻辑，否则没必要在日常开发的样式表中使用条件语句。实际上，条件语句主要适用于库和框架。

无论何时，如果你感觉需要它们，请遵守下述准则：

- 除非必要，不然不需要括号；
- 务必在 `@if` 之前添加空行；
- 务必在左开大括号（`{`）后换行；
- `@else` 语句和它前面的右闭大括号（`}`）写在同一行；
- 务必在右闭大括号（`}`）后添加空行，除非下一行还是右闭大括号（`}`），那么就在最后一个右闭大括号（`}`）后添加空行。

```
// Yep
@if $support-legacy {
  // …
} @else {
  // …
}

// Nope
@if ($support-legacy == true) {
  // …
}
@else {
  // …
}
```

测试一个错误值时，通常使用 `not` 关键字而不是比较与 `false` 或 `null` 等值。

```
// Yep
@if not index($list, $item) {
  // …
}

// Nope
@if index($list, $item) == null {
  // …
}
```

通常将变量置于语句的左侧，而将结果置于右侧。如果使用相反的顺序，通常会增加阅读难度，特别是对于没有经验的开发者。

```
// Yep
@if $value == 42 {
  // …
}

// Nope
@if 42 == $value {
  // …
}
```

当使用条件语句并在一些条件下有内联函数返回不同结果时，始终要确保最外层函数有一个 `@return` 语句。

```
// Yep
@function dummy($condition) {
  @if $condition {
    @return true;
  }

  @return false;
}

// Nope
@function dummy($condition) {
  @if $condition {
    @return true;
  } @else {
    @return false;
  }
}
```

11

循环

因为Sass提供了如 lists (页 0) 和 maps (页 0) 这样的复杂数据结构，所以对于提供给开发者遍历这些数据结构的能力也无需惊讶。

然而，循环的出现意味着存在本不可能出现在Sass中的复杂逻辑。在使用循环之前，务必确定这么做是有道理的，并且确认这么做可以解决问题。

Each

`@each` 循环绝对是Sass提供的三个循环方式中最常用的。它提供了一个简洁的 API 来迭代列表或 map。

```
@each $theme in $themes {
  .section-#{$theme} {
    background-color: map-get($colors, $theme);
  }
}
```

当迭代一个 map 时，通常使用 `$key` 和 `$value` 作为变量名称来确保一致性。

```
@each $key, $value in $map {
  .section-#{$key} {
    background-color: $value;
  }
}
```

同时遵守下述规则，确保可读性：

- `each` 前添加空行；
- 除非下一行是右闭大括号（`}`），否则在所有右闭大括号（`}`）后面添加新行。

For

当需要聚合伪类 `:nth-*` 的时候，使用 `@for` 循环很有用。除了这些使用场景，如果*必须*迭代最好还是使用 `@each` 循环。

```
@for $i from 1 through 10 {
  .foo:nth-of-type(#{$i}) {
    border-color: hsl($i * 36, 50%, 50%);
  }
}
```

要坚持一贯的传统，始终使用 `$i` 作为变量名，除非有非常好的原因，否则永远不要使用 `to` 关键字：而是始终使用 `through`。许多开发者甚至不知道 Sass 有这个变化；使用它可能会造成混乱。

最后，确保遵循规范以保持可读性：

- `each` 前添加空行；
- 除非下一行是右闭大括号（`}`），否则在所有右闭大括号（`}`）后面添加新行。

While

绝对没有必要在真实的 Sass 项目中使用 `@while` 循环——**不要使用它**。

12

警告和错误

如果提到经常被开发者忽略的特性，那应该就是动态输出错误和提醒的功能了。事实上，Sass 自带三条自定义指令从标准输出系统（CLI，编译程序……）中打印内容：

- `@debug`；
- `@warn`；
- `@error`.

先让我们把 `@debug` 放一边，毕竟它主要是用作调试 SassScript，而这并不是我们的重点。然后我们就剩下了相互间没有明显差异的 `@warn` 和 `@error`，唯一的不同是其中一个可以中断编译器的工作，而另一个不能。我想让你猜猜具体每一个都是做什么的。

现在，在一个 Sass 项目中往往存在大量的错误和提醒。基本上任何混合宏和函数出错都会发生特定类型或参数的错误，或者显示假设的提醒。

警告

使用 Sass-MQ 中的这个函数可以转换 `px` 为 `em`，展示如下：

```
@function mq-px2em($px, $base-font-size: $mq-base-font-size) {
  @if unitless($px) {
    @warn 'Assuming #{$px} to be in pixels, attempting to convert it into pixels.';
    @return mq-px2em($px + 0px);
  } @else if unit($px) == em {
    @return $px;
  }

  @return ($px / $base-font-size) * 1em;
}
```

如果碰巧值是无单位的，这个函数就会默认单位是像素。就这一点而论，一个假设可能会带来风险，所以软件应该能够预测风险并提醒使用者。

错误

错误，与提示有所不同，将会中断编译器的下一步进程。基本上，它们中断编译并像堆栈跟踪一样在输出流中显示相关信息，这对调试很有帮助。因此，如果程序无法继续执行就应该抛出错误。如有可能，尝试解决这个问题并以显示提醒的方式代替。

举个例子，假设你创建了一个 getter 函数来从特定 map 中获取值。如果想要获取的值并不在 map 中，就可能会抛出错误。

```
/// Z-indexes map, gathering all Z layers of the application
/// @access private
/// @type Map
/// @prop {String} key - Layer's name
/// @prop {Number} value - Z value mapped to the key
$z-indexes: (
  'modal': 5000,
  'dropdown': 4000,
  'default': 1,
  'below': -1,
);

/// Get a z-index value from a layer name
/// @access public
/// @param {String} $layer - Layer's name
/// @return {Number}
/// @require $z-indexes
@function z($layer) {
  @if not map-has-key($z-indexes, $layer) {
    @error 'There is no layer named `#{$layer}` in $z-indexes. '
         + 'Layer should be one of #{map-keys($z-indexes)}.';
  }

  @return map-get($z-indexes, $layer);
}
```

更多有关如何高效使用 `@error` 的信息可以点击查看这篇文章：an introduction to error handling in-sass。

13

工具

和 Sass 一样受欢迎的 CSS 预处理器的优秀之处就在于，它提供了包括框架、插件、库和工具在内的整套开发环境。Sass 诞生 8 年以来，其本身的特性和 everything that can be written in Sass has been written in Sass 一文中的观点越来越相近。

不过，我的建议是最小程度的依赖于各种工具。管理依赖可能会是你特别不想面对的事情。此外，在 Sass 中很少需要外部依赖。

Compass

Compass 是 Sass 中主要的框架。其开发者 Chris Eppstein，是 Sass 的两位核心开发者之一。如果你想听一下我的看法，我想说这个框架一直很流行。

不过，我已经不再使用 Compass 了，主要原因是它很大程度上拖慢了 Sass。Ruby Sass 本身就比较慢，所以在此之上增加更多功能并无益处。

实际上，我们通常只使用了框架本身的一点点功能，而完整的 Compass 是庞大的。混合宏的跨浏览器兼容功能也只是冰山一角。数学函数、图像辅助、幽灵图……使用这个优秀的工具有太多的好处了。

不幸的是，这就是所有的语法糖而且没有一个是杀手级的特性。精灵图生成器虽然*非常优秀*，但也会报出一两个错误。不过 Grunticon 和 Grumpicon 就运行的很好，而且它们还有可以被插入到构建过程的优势。

虽然我不建议使用 Compass，但我也不会禁止使用它，特别是当它不兼容 LibSass 的时候（即使它正朝这个方向努力）。如果你感觉使用起来还不错，这当然可以，但是我认为最终你也不会从中收获更多。

Ruby Sass 目前正着手进行一些很棒的优化，目标是通过运用诸多函数和混合宏实现具有深度逻辑的样式。它们应该显著改善性能，而这往往是 Compass 和其他框架拖慢 Sass 的原因。

栅格系统

随着响应式网页设计地遍地开花，栅格系统布局已经成为了一种必然选择。为了固定大小并使设计风格统一，我们通常使用网格来给元素布局。为了避免反复地布局工作，一些非凡的想法认为应该使它们保持可复用性。

还是长话短说吧：我并非栅格系统的拥趸。当然我确实看到了它的潜力，但大多的是矫枉过正，而且主要是被设计师用来绘制红栏白底的原型。你还记得上一次有 *thank-God-I-have-this-tool-to-build-this-2-5-3.1-π-grid* 如此赞叹的时间吗？那就对了，从来没有过。因为在多数情况下，你只是想使用12列栅格布局，毫不奇特。

如果你正在项目中使用类似 Bootstrap 和 Foundation 的 CSS 框架，我建议善用它们来避免额外的依赖，因为此时它们很有可能就包含了一套栅格系统，。

如果你尚未依赖于特定的栅格系统，那么也许会乐意了解这里介绍的两个由 Sass 支持的栅格引擎：Susy 和 Singularity。它们都可以满足你的需求，所以只需从中挑选喜欢的一个来用即可并且可以放心即使是你的苛刻需求—哪怕是最多变的—也可以被实现。

或者你可以处理地更轻松些，就像使用 CSSWizardry Grids 的感觉。总而言之，任何选择都不会对你的代码风格有过多影响，所以这一点上一切取决于你。

SCSS-lint

审查代码是非常重要的事情。通常来说，遵从一份样式指南的规范可以帮助减少代码质量上的问题，但是没有人的工作是无懈可击的，何况总有些地方需要改善。所以，可以认为，审查代码和注释文档一样重要。

SCSS-lint 是一个帮你保持 SCSS 文件简洁而又具有可读性的工具。它是完全可定制化的，并且非常易于和其他工具集成。

幸运的是，本文档的描述非常类似于 SCSS-lint 的使用建议。为了根据 Sass 样式指南配置 SCSS-lint，建议如下配置：

```
linters:

  BangFormat:
    enabled: true
    space_before_bang: true
    space_after_bang: false

  BemDepth:
    enabled: true
    max_elements: 1

  BorderZero:
    enabled: true
    convention: zero

  ChainedClasses:
    enabled: false

  ColorKeyword:
    enabled: true

  ColorVariable:
    enabled: false

  Comment:
    enabled: false

  DebugStatement:
    enabled: true

  DeclarationOrder:
```

```
  enabled: true

DisableLinterReason:
  enabled: true

DuplicateProperty:
  enabled: false

ElsePlacement:
  enabled: true
  style: same_line

EmptyLineBetweenBlocks:
  enabled: true
  ignore_single_line_blocks: true

EmptyRule:
  enabled: true

ExtendDirective:
  enabled: false

FinalNewline:
  enabled: true
  present: true

HexLength:
  enabled: true
  style: short

HexNotation:
  enabled: true
  style: lowercase

HexValidation:
  enabled: true

IdSelector:
  enabled: true

ImportantRule:
  enabled: false

ImportPath:
  enabled: true
```

```
  leading_underscore: false
  filename_extension: false

Indentation:
  enabled: true
  allow_non_nested_indentation: true
  character: space
  width: 2

LeadingZero:
  enabled: true
  style: include_zero

MergeableSelector:
  enabled: false
  force_nesting: false

NameFormat:
  enabled: true
  convention: hyphenated_lowercase
  allow_leading_underscore: true

NestingDepth:
  enabled: true
  max_depth: 1

PlaceholderInExtend:
  enabled: true

PrivateNamingConvention:
  enabled: true
  prefix: _

PropertyCount:
  enabled: false

PropertySortOrder:
  enabled: false

PropertySpelling:
  enabled: true
  extra_properties: []

PropertyUnits:
  enabled: false
```

```
PseudoElement:
  enabled: true

QualifyingElement:
  enabled: true
  allow_element_with_attribute: false
  allow_element_with_class: false
  allow_element_with_id: false

SelectorDepth:
  enabled: true
  max_depth: 3

SelectorFormat:
  enabled: true
  convention: hyphenated_lowercase
  class_convention: '^(?:u|is|has)\-[a-z][a-zA-Z0-9]*$|^(?!u|is|has)[a-zA-Z][a-zA-Z0-9]*(?:\-[a-z][a-zA-Z0-9]*)?(?:\-

Shorthand:
  enabled: true

SingleLinePerProperty:
  enabled: true
  allow_single_line_rule_sets: false

SingleLinePerSelector:
  enabled: true

SpaceAfterComma:
  enabled: true

SpaceAfterPropertyColon:
  enabled: true
  style: one_space

SpaceAfterPropertyName:
  enabled: true

SpaceAfterVariableColon:
  enabled: true
  style: at_least_one_space

SpaceAfterVariableName:
  enabled: true
```

```
SpaceAroundOperator:
  enabled: true
  style: one_space

SpaceBeforeBrace:
  enabled: true
  style: space
  allow_single_line_padding: true

SpaceBetweenParens:
  enabled: true
  spaces: 0

StringQuotes:
  enabled: true
  style: single_quotes

TrailingSemicolon:
  enabled: true

TrailingZero:
  enabled: true

TransitionAll:
  enabled: false

UnnecessaryMantissa:
  enabled: true

UnnecessaryParentReference:
  enabled: true

UrlFormat:
  enabled: false

UrlQuotes:
  enabled: true

VariableForProperty:
  enabled: false

VendorPrefixes:
  enabled: true
  identifier_list: base
```

```
    include: []
    exclude: []

  ZeroUnit:
    enabled: true
```

如果你不确信 SCSS-lint 的必要性，建议阅读以下文章：《使用 SCSS-lint 审查你的 Sass 代码》，《通过 theguardian.com 改善你的 Sass 代码质量》 和 《一份强制实施的 SCSS 代码规范》.

如果你想将 SCSS-lint 插入到 Grunt 构建过程中，那么 Grunt 插件 grunt-scss-lint 一定会对你有所帮助。

此外，如果你在寻找一个运行 SCSS-lint 的简洁工具，那么 Thoughtbot (Bourbon, Neat...) 正在开发的 Hound 也会对你有所帮助。

14

总结概要

简而言之，本文主要包括以下几个方面的内容:

核心原则

- 创建编程规范的目的就是为了保证协作开发的一致性。即使你对本文有不同的意见，也要保证整体开发的一致性。?（页 0）
- 尽可能让 Sass 代码保持简洁。除非是绝对需要，否则绝没有必要构建复杂的系统。?（页 0）
- 请记住，有时候保持简洁（KISS，Keep It Simple, Stupid）比减少重复（Don't Repeat Yourself）更重要。?（页 0）

语法 & 格式

- 使用两个空格代表缩进，而不是使用tab键。? (页 0)
- 理想上，每行保持为 80 个字符宽度。? (页 0)
- 根据 CSS Guidelines 正确书写 CSS 属性。? (页 0)
- 有意义的使用空格。? (页 0)

字符串

- 强烈建议在样式表顶部使用 `@charset` 指令声明字符集。? (页 0)
- 除非用作 CSS 标识符，否则应该使用单引号包裹字符串和 URL。? (页 0)

数值

- 数字尾部不使用 0 ，并且强制在小于 1 的数字前使用 0。? (页 0)
- 长度样式属性值为 0 时不要添加单位。? (页 0)
- 使用括号包裹运算表达式。? (页 0)
- 不使用幻数。? (页 0)

颜色

- 颜色表示法的先后顺序：关键字 > HSL > RGB > 十六进制。? (页 0)
- 当减淡或加深颜色时，最好使用 `mix(..)` 替代 `darken(..)` 和 `lighten(..)` 。? (页 0)

列表

- 使用逗号分隔列表。? (页 0)
- 使用圆括号增加可读性。? (页 0)
- 列表尾部不要添加逗号（当它们是内联状态时）。? (页 0)

Maps

- 当 map 内部包含多个键值对时，将它们分成多行。？（页 0）
- 为了增加可维护性，map 内部的最后一个键值对应该添加一个逗号。？（页 0）
- 如果 map 的键是字符串，应该使用引号包裹起来。？（页 0）

声明顺序

- 只要保持开发的一致性，无论哪种声明顺序（根据字母表或者类型排序）都是可以接受的。？（页 0）

选择器嵌套

- 减少选择器嵌套。？（页 0）
- 对伪类和伪元素使用选择器嵌套。？（页 0）
- 媒体查询也可以嵌套到相关的选择器当中。？（页 0）

命名约定

- 坚持连字符分隔的命名方式。？（页 0）

注释

- CSS 中充满了机巧，当你有所疑问的时候，就应该写下相关的注释。？（页 0）
- 对于变量、函数、混合宏和占位符创建的公开 API，使用 SassDoc 来注释。？（页 0）

变量

- 在公开 API 中使用 `!default` 标志变量，让后期的修改更安全。？（页 0）
- 不要在顶层使用 `!global` 标志，应为这可能会在未来和 Sass 语法有冲突。？（页 0）

扩展

- 坚持扩展占位符，而不是扩展既有的 CSS 选择器。？（页 0）
- 尽可能少地扩展占位符，避免潜在的副作用。？（页 0）