Sass 快速入门

书栈(BookStack.CN)

目　录

致谢

当前文档 《Sass 快速入门》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-02-27。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/sass-quickstart

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

> 来源(书栈小编注)：https://www.sass.hk/guide/

1. 使用变量;

`sass` 让人们受益的一个重要特性就是它为 `css` 引入了变量。你可以把反复使用的 `css` 属性值定义成变量，然后通过变量名来引用它们，而无需重复书写这一属性值。或者，对于仅使用过一次的属性值，你可以赋予其一个易懂的变量名，让人一眼就知道这个属性值的用途。

`sass` 使用 `$` 符号来标识变量(老版本的 `sass` 使用 `!` 来标识变量。改成$是多半因为 `!highlight-color` 看起来太丑了。)，比如 `$highlight-color` 和 `$sidebar-width` 。为什么选择 `$` 符号呢？因为它好认、更具美感，且在CSS中并无他用，不会导致与现存或未来的 `css` 语法冲突。

1-1. 变量声明;

`sass` 变量的声明和 `css` 属性的声明很像：

```
1. $highlight-color: #F90;
```

这意味着变量 `$highlight-color` 现在的值是 `#F90` 。任何可以用作 `css` 属性值的赋值都可以用作 `sass` 的变量值，甚至是以空格分割的多个属性值，如 `$basic-border: 1px solid black;` ，或以逗号分割的多个属性值，如 `$plain-font: "Myriad Pro"、Myriad、"Helvetica Neue"、Helvetica、"Liberation Sans"、Arial和sans-serif; sans-serif;` 。这时变量还没有生效，除非你引用这个变量—我们很快就会了解如何引用。

与 `CSS` 属性不同，变量可以在 `css` 规则块定义之外存在。当变量定义在 `css` 规则块内，那么该变量只能在此规则块内使用。如果它们出现在任何形式的 `{…}` 块中（如 `@media` 或者 `@font-face` 块），情况也是如此：

```
1. $nav-color: #F90;
2. nav {
3.   $width: 100px;
4.   width: $width;
5.   color: $nav-color;
6. }
7.
8. //编译后
9.
10. nav {
11.   width: 100px;
12.   color: #F90;
13. }
```

在这段代码中， `$nav-color` 这个变量定义在了规则块外边，所以在这个样式表中都可以像 `nav` 规则块那样引用它。 `$width` 这个变量定义在了 `nav` 的 `{ }` 规则块内，所以它只能在 `nav` 规则块内使用。这意味着是你可以在样式表的其他地方定义和使用 `$width` 变量，不会对这里造成影响。

只声明变量其实没啥用处，我们最终的目的还是使用它们。上例已介绍了如何使用 `$nav-color` 和 `$width` 这两个变量，接下来我们将进一步探讨变量的使用方法。

1-2. 变量引用;

凡是 `css` 属性的标准值（比如说1px或者bold）可存在的地方，变量就可以使用。 `css` 生成时，变量会被它们的值所替代。之后，如果你需要一个不同的值，只需要改变这个变量的值，则所有引用此变量的地方生成的值都会随之改变。

```
$highlight-color: #F90;
.selected {
  border: 1px solid $highlight-color;
}

//编译后

.selected {
  border: 1px solid #F90;
}
```

看上边示例中的 `$highlight-color` 变量，它被直接赋值给 `border` 属性，当这段代码被编译输出 `css` 时， `$highlight-color` 会被 `#F90` 这一颜色值所替代。产生的效果就是给 `selected` 这个类一条1像素宽、实心且颜色值为 `#F90` 的边框。

在声明变量时，变量值也可以引用其他变量。当你通过粒度区分，为不同的值取不同名字时，这相当有用。下例在独立的颜色值粒度上定义了一个变量，且在另一个更复杂的边框值粒度上也定义了一个变量：

```
$highlight-color: #F90;
$highlight-border: 1px solid $highlight-color;
.selected {
  border: $highlight-border;
}

//编译后

.selected {
  border: 1px solid #F90;
}
```

这里， `$highlight-border` 变量的声明中使用了 `$highlight-color` 这个变量。产生的效果就跟你直接为 `border` 属性设置了一个 `1px` `$highlight-color solid` 的值是一样的。最后，我们来了解一下变量命名的实用技巧，以结束关于变量的介绍。

1-3. 变量名用中划线还是下划线分隔;

`sass` 的变量名可以与 `css` 中的属性名和选择器名称相同，包括中划线和下划线。这完全取决于个人的喜好，有些人喜欢使用中划线来分隔变量中的多个词（如 `$highlight-color` ），而有些人喜欢使用下划线（如 `$highlight_color` ）。使用中划线的方式更为普遍，这也是 `compass` 和本文都用的方式。

不过， `sass` 并不想强迫任何人一定使用中划线或下划线，所以这两种用法相互兼容。用中划线声明的变量可以使用下划线的方式引用，反之亦然。这意味着即使 `compass` 选择用中划线的命名方式，这并不影响你在使用 `compass` 的

样式中用下划线的命名方式进行引用：

```
$link-color: blue;
a {
  color: $link_color;
}

//编译后

a {
  color: blue;
}
```

在上例中， `$link-color` 和 `$link_color` 其实指向的是同一个变量。实际上，在 `sass` 的大多数地方，中划线命名的内容和下划线命名的内容是互通的，除了变量，也包括对混合器和Sass函数的命名。但是在 `sass` 中纯 `css` 部分不互通，比如类名、ID或属性名。

尽管变量自身提供了很多有用的地方，但是 `sass` 基于变量提供的更为强大的工具才是我们关注的焦点。只有当变量与 `sass` 的其他特性一起使用时，才能发挥其全部的潜能。接下来，我们将探讨其中一个非常重要的特性，即规则嵌套。

2. 嵌套CSS 规则;

`css` 中重复写选择器是非常恼人的。如果要写一大串指向页面中同一块的样式时，往往需要一遍又一遍地写同一个 `ID` ：

```
#content article h1 { color: #333 }
#content article p { margin-bottom: 1.4em }
#content aside { background-color: #EEE }
```

像这种情况， `sass` 可以让你只写一遍，且使样式可读性更高。在Sass中，你可以像俄罗斯套娃那样在规则块中嵌套规则块。 `sass` 在输出 `css` 时会帮你把这些嵌套规则处理好，避免你的重复书写。

```
#content {
  article {
    h1 { color: #333 }
    p { margin-bottom: 1.4em }
  }
  aside { background-color: #EEE }
}
```

```
 /* 编译后 */
#content article h1 { color: #333 }
#content article p { margin-bottom: 1.4em }
#content aside { background-color: #EEE }
```

上边的例子，会在输出 `css` 时把它转换成跟你之前看到的一样的效果。这个过程中， `sass` 用了两步，每一步都是像打开俄罗斯套娃那样把里边的嵌套规则块一个个打开。首先，把 `#content` （父级）这个 `id` 放到 `article` 选择器（子级）和 `aside` 选择器（子级）的前边：

```
#content {
  article {
    h1 { color: #333 }
    p { margin-bottom: 1.4em }
  }
  #content aside { background-color: #EEE }
}
```

```
 /* 编译后 */
#content article h1 { color: #333 }
#content article p { margin-bottom: 1.4em }
#content aside { background-color: #EEE }
```

然后， `#content article` 里边还有嵌套的规则， `sass` 重复一遍上边的步骤，把新的选择器添加到内嵌的选择器前边。

一个给定的规则块，既可以像普通的CSS那样包含属性，又可以嵌套其他规则块。当你同时要为一个容器元素及其子元

素编写特定样式时，这种能力就非常有用了。

```
#content {
  background-color: #f5f5f5;
  aside { background-color: #eee }
}
```

容器元素的样式规则会被单独抽离出来，而嵌套元素的样式规则会像容器元素没有包含任何属性时那样被抽离出来。

```

#content { background-color: #f5f5f5 }
#content aside { background-color: #eee }
```

大多数情况下这种简单的嵌套都没问题，但是有些场景下不行，比如你想要在嵌套的选择器里边立刻应用一个类似于 `:hover` 的伪类。为了解决这种以及其他情况， `sass` 提供了一个特殊结构 `&` 。

2-1. 父选择器的标识符&;

一般情况下， `sass` 在解开一个嵌套规则时就会把父选择器（ `#content` ）通过一个空格连接到子选择器的前边（ `article` 和 `aside` ）形成（ `#content article` 和 `#content aside` ）。这种在CSS里边被称为后代选择器，因为它选择ID为 `content` 的元素内所有命中选择器 `article` 和 `aside` 的元素。但在有些情况下你却不会希望 `sass` 使用这种后代选择器的方式生成这种连接。

最常见的一种情况是当你为链接之类的元素写 `:hover` 这种伪类时，你并不希望以后代选择器的方式连接。比如说，下面这种情况 `sass` 就无法正常工作：

```
article a {
  color: blue;
  :hover { color: red }
}
```

这意味着 `color: red` 这条规则将会被应用到选择器 `article a :hover` ， `article` 元素内链接的所有子元素在被 `hover` 时都会变成红色。这是不正确的！你想把这条规则应用到超链接自身，而后代选择器的方式无法帮你实现。

解决之道为使用一个特殊的 `sass` 选择器，即父选择器。在使用嵌套规则时，父选择器能对于嵌套规则如何解开提供更好的控制。它就是一个简单的 `&` 符号，且可以放在任何一个选择器可出现的地方，比如 `h1` 放在哪，它就可以放在哪。

```
article a {
  color: blue;
  &:hover { color: red }
}
```

当包含父选择器标识符的嵌套规则被打开时，它不会像后代选择器那样进行拼接，而是 `&` 被父选择器直接替换：

```
article a { color: blue }
article a:hover { color: red }
```

在为父级选择器添加 `:hover` 等伪类时，这种方式非常有用。同时父选择器标识符还有另外一种用法，你可以在父选择器之前添加选择器。举例来说，当用户在使用IE浏览器时，你会通过 `JavaScript` 在 `<body>` 标签上添加一个ie的类名，为这种情况编写特殊的样式如下：

```
#content aside {
  color: red;
  body.ie & { color: green }
}
```

```

/*编译后*/
#content aside {color: red};
body.ie #content aside { color: green }
```

`sass` 在选择器嵌套上是非常智能的，即使是带有父选择器的情况。当 `sass` 遇到群组选择器（由多个逗号分隔开的选择器形成）也能完美地处理这种嵌套。

2-2. 群组选择器的嵌套;

在 `CSS` 里边，选择器 `h1` `h2` 和 `h3` 会同时命中h1元素、h2元素和h3元素。与此类似， `.button` `button` 会命中button元素和类名为.button的元素。这种选择器称为群组选择器。群组选择器的规则会对命中群组中任何一个选择器的元素生效。

```
.button, button {
  margin: 0;
}
```

当看到上边这段代码时，你可能还没意识到会有重复性的工作。但会很快发现：如果你需要在一个特定的容器元素内对这样一个群组选择器进行修饰，情况就不同了。 `css` 的写法会让你在群组选择器中的每一个选择器前都重复一遍容器元素的选择器。

```
.container h1, .container h2, .container h3 { margin-bottom: .8em }
```

非常幸运， `sass` 的嵌套特性在这种场景下也非常有用。当 `sass` 解开一个群组选择器规则内嵌的规则时，它会把每一个内嵌选择器的规则都正确地解出来：

```
.container {
  h1, h2, h3 {margin-bottom: .8em}
}
```

首先 `sass` 将 `.container` 和 `h1` `.container` 和 `h2` `.container` 和 `h3` 分别组合，然后将三者重新组合成一个群组选择器，生成你前边看到的普通 `css` 样式。对于内嵌在群组选择器内的嵌套规则，处理方式也一样：

```
nav, aside {
  a {color: blue}
```

```
3. }
```

首先 `sass` 将 `nav` 和 `a` `aside` 和 `a` 分别组合，然后将二者重新组合成一个群组选择器：

```
1. nav a, aside a {color: blue}
```

处理这种群组选择器规则嵌套上的强大能力，正是 `sass` 在减少重复敲写方面的贡献之一。尤其在当嵌套级别达到两层甚至三层以上时，与普通的 `css` 编写方式相比，只写一遍群组选择器大大减少了工作量。

有利必有弊，你需要特别注意群组选择器的规则嵌套生成的 `css` 。虽然 `sass` 让你的样式表看上去很小，但实际生成的 `css` 却可能非常大，这会降低网站的速度。

关于选择器嵌套的最后一个方面，我们看看 `sass` 如何处理组合选择器，比如>、+和~的使用。你将看到，这种场景下你甚至无需使用父选择器标识符。

2-3. 子组合选择器和同层组合选择器：>、+和~；

上边这三个组合选择器必须和其他选择器配合使用，以指定浏览器仅选择某种特定上下文中的元素。

```
1. article section { margin: 5px }
2. article > section { border: 1px solid #ccc }
```

你可以用子组合选择器>选择一个元素的直接子元素。上例中，第一个选择器会选择article下的所有命中section选择器的元素。第二个选择器只会选择article下紧跟着的子元素中命中section选择器的元素。

在下例中，你可以用同层相邻组合选择器 `+` 选择 `header` 元素后紧跟的 `p` 元素：

```
1. header + p { font-size: 1.1em }
```

你也可以用同层全体组合选择器 `~` ，选择所有跟在 `article` 后的同层 `article` 元素，不管它们之间隔了多少其他元素：

```
1. article ~ article { border-top: 1px dashed #ccc }
```

这些组合选择器可以毫不费力地应用到 `sass` 的规则嵌套中。可以把它们放在外层选择器后边，或里层选择器前边：

```
1. article {
2.   ~ article { border-top: 1px dashed #ccc }
3.   > section { background: #eee }
4.   dl > {
5.     dt { color: #333 }
6.     dd { color: #555 }
7.   }
8.   nav + & { margin-top: 0 }
9. }
```

`sass` 会如你所愿地将这些嵌套规则一一解开组合在一起：

```
1. article ~ article { border-top: 1px dashed #ccc }
2. article > footer { background: #eee }
3. article dl > dt { color: #333 }
4. article dl > dd { color: #555 }
5. nav + article { margin-top: 0 }
```

在 `sass` 中，不仅仅 `css` 规则可以嵌套，对属性进行嵌套也可以减少很多重复性的工作。

2-4. 嵌套属性;

在 `sass` 中，除了CSS选择器，属性也可以进行嵌套。尽管编写属性涉及的重复不像编写选择器那么糟糕，但是要反复写 `border-style` `border-width` `border-color` 以及 `border-*` 等也是非常烦人的。在 `sass` 中，你只需敲写一遍 `border` ：

```
1. nav {
2.   border: {
3.   style: solid;
4.   width: 1px;
5.   color: #ccc;
6.   }
7. }
```

嵌套属性的规则是这样的：把属性名从中划线-的地方断开，在根属性后边添加一个冒号:，紧跟一个 `{ }` 块，把子属性部分写在这个 `{ }` 块中。就像 `css` 选择器嵌套一样， `sass` 会把你的子属性一一解开，把根属性和子属性部分通过中划线-连接起来，最后生成的效果与你手动一遍遍写的 `css` 样式一样：

```
1. nav {
2.   border-style: solid;
3.   border-width: 1px;
4.   border-color: #ccc;
5. }
```

对于属性的缩写形式，你甚至可以像下边这样来嵌套，指明例外规则：

```
1. nav {
2.   border: 1px solid #ccc {
3.   left: 0px;
4.   right: 0px;
5.   }
6. }
```

这比下边这种同等样式的写法要好：

```
1. nav {
2.   border: 1px solid #ccc;
```

```
  border-left: 0px;
  border-right: 0px;
}
```

属性和选择器嵌套是非常伟大的特性，因为它们不仅大大减少了你的编写量，而且通过视觉上的缩进使你编写的样式结构更加清晰，更易于阅读和开发。

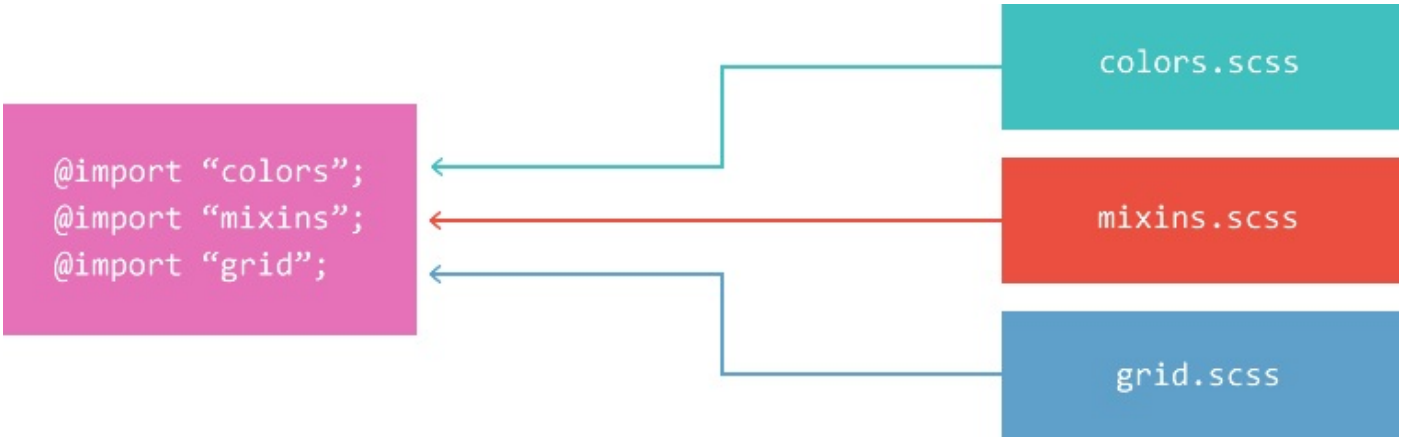
即便如此，随着你的样式表变得越来越大，这种写法也很难保持结构清晰。有时，处理这种大量样式的唯一方法就是把它们分拆到多个文件中。 `sass` 通过对 `css` 原有 `@import` 规则的改进直接支持了这一特性。

3. 导入SASS文件;

`css` 有一个特别不常用的特性，即 `@import` 规则，它允许在一个 `css` 文件中导入其他 `css` 文件。然而，后果是只有执行到 `@import` 时，浏览器才会去下载其他 `css` 文件，这导致页面加载起来特别慢。

`sass` 也有一个 `@import` 规则，但不同的是， `sass` 的 `@import` 规则在生成 `css` 文件时就把相关文件导入进来。这意味着所有相关的样式被归纳到了同一个 `css` 文件中，而无需发起额外的下载请求。另外，所有在被导入文件中定义的变量和混合器（参见2.5节）均可在导入文件中使用。

使用 `sass` 的 `@import` 规则并不需要指明被导入文件的全名。你可以省略 `.sass` 或 `.scss` 文件后缀（见下图）。这样，在不修改样式表的前提下，你完全可以随意修改你或别人写的被导入的 `sass` 样式文件语法，在 `sass` 和 `scss` 语法之间随意切换。举例来说， `@import` `"sidebar";`这条命令将把 `sidebar.scss` 文件中所有样式添加到当前样式表中。



本节将介绍如何使用 `sass` 的 `@import` 来处理多个 `sass` 文件。首先，我们将学习编写那些被导入的 `sass` 文件，因为在一个大型 `sass` 项目中，这样的文件是你最常编写的那一类。接着，了解集中导入 `sass` 文件的方法，使你的样式可重用性更高，包括声明可自定义的变量值，以及在某一个选择器范围内导入 `sass` 文件。最后，介绍如何在 `sass` 中使用 `css` 原生的 `@import` 命令。

通常，有些 `sass` 文件用于导入，你并不希望为每个这样的文件单独地生成一个 `css` 文件。对此， `sass` 用一个特殊的约定来解决。

3-1. 使用SASS部分文件;

当通过 `@import` 把 `sass` 样式分散到多个文件时，你通常只想生成少数几个 `css` 文件。那些专门为 `@import` 命令而编写的 `sass` 文件，并不需要生成对应的独立 `css` 文件，这样的 `sass` 文件称为局部文件。对此， `sass` 有一个特殊的约定来命名这些文件。

此约定即， `sass` 局部文件的文件名以下划线开头。这样， `sass` 就不会在编译时单独编译这个文件输出 `css` ，而只把这个文件用作导入。当你 `@import` 一个局部文件时，还可以不写文件的全名，即省略文件名开头的下划线。举例来说，你想导入 `themes/_night-sky.scss` 这个局部文件里的变量，你只需在样式表中写 `@import` `"themes/night-sky";` 。

局部文件可以被多个不同的文件引用。当一些样式需要在多个页面甚至多个项目中使用时，这非常有用。在这种情况

下，有时需要在你的样式表中对导入的样式稍作修改， `sass` 有一个功能刚好可以解决这个问题，即默认变量值。

3-2. 默认变量值;

一般情况下，你反复声明一个变量，只有最后一处声明有效且它会覆盖前边的值。举例说明：

```
$link-color: blue;
$link-color: red;
a {
color: $link-color;
}
```

在上边的例子中，超链接的 `color` 会被设置为 `red` 。这可能并不是你想要的结果，假如你写了一个可被他人通过 `@import` 导入的 `sass` 库文件，你可能希望导入者可以定制修改 `sass` 库文件中的某些值。使用 `sass` 的 `!default` 标签可以实现这个目的。它很像 `css` 属性中 `!important` 标签的对立面，不同的是 `!default` 用于变量，含义是：如果这个变量被声明赋值了，那就用它声明的值，否则就用这个默认值。

```
$fancybox-width: 400px !default;
.fancybox {
width: $fancybox-width;
}
```

在上例中，如果用户在导入你的 `sass` 局部文件之前声明了一个 `$fancybox-width` 变量，那么你的局部文件中对 `$fancybox-width` 赋值 `400px` 的操作就无效。如果用户没有做这样的声明，则 `$fancybox-width` 将默认为 `400px` 。

接下来我们将学习嵌套导入，它允许只在某一个选择器的范围内导入 `sass` 局部文件。

3-3. 嵌套导入;

跟原生的 `css` 不同， `sass` 允许 `@import` 命令写在 `css` 规则内。这种导入方式下，生成对应的 `css` 文件时，局部文件会被直接插入到 `css` 规则内导入它的地方。举例说明，有一个名为 `_blue-theme.scss` 的局部文件，内容如下：

```
aside {
  background: blue;
  color: white;
}
```

然后把它导入到一个CSS规则内，如下所示：

```
.blue-theme {@import "blue-theme"}

//生成的结果跟你直接在.blue-theme选择器内写_blue-theme.scss文件的内容完全一样。

.blue-theme {
  aside {
```

```
    background: blue;
    color: #fff;
  }
}
```

被导入的局部文件中定义的所有变量和混合器，也会在这个规则范围内生效。这些变量和混合器不会全局有效，这样我们就可以通过嵌套导入只对站点中某一特定区域运用某种颜色主题或其他通过变量配置的样式。

有时，可用 `css` 原生的 `@import` 机制，在浏览器中下载必需的 `css` 文件。 `sass` 也提供了几种方法来达成这种需求。

3-4. 原生的CSS导入;

由于 `sass` 兼容原生的 `css` ，所以它也支持原生的 `CSS@import` 。尽管通常在 `sass` 中使用 `@import` 时， `sass` 会尝试找到对应的 `sass` 文件并导入进来，但在下列三种情况下会生成原生的 `CSS@import` ，尽管这会造成浏览器解析 `css` 时的额外下载：

- 被导入文件的名字以 `.css` 结尾；
- 被导入文件的名字是一个URL地址（比如http://www.sass.hk/css/css.css），由此可用谷歌字体API提供的相应服务；
- 被导入文件的名字是 `CSS` 的url()值。 这就是说，你不能用 `sass` 的 `@import` 直接导入一个原始的 `css` 文件，因为 `sass` 会认为你想用 `css` 原生的 `@import` 。但是，因为 `sass` 的语法完全兼容 `css` ，所以你可以把原始的 `css` 文件改名为 `.scss` 后缀，即可直接导入了。

文件导入是保证 `sass` 的代码可维护性和可读性的重要一环。次之但亦非常重要的就是注释了。注释可以帮助样式作者记录写 `sass` 的过程中的想法。在原生的 `css` 中，注释对于其他人是直接可见的，但 `sass` 提供了一种方式可在生成的 `css` 文件中按需抹掉相应的注释。

4. 静默注释;

`css` 中注释的作用包括帮助你组织样式、以后你看自己的代码时明白为什么这样写，以及简单的样式说明。但是，你并不希望每个浏览网站源码的人都能看到所有注释。

`sass` 另外提供了一种不同于 `css` 标准注释格式 `/ … /` 的注释语法，即静默注释，其内容不会出现在生成的 `css` 文件中。静默注释的语法跟 `JavaScript` `Java` 等类 `C` 的语言中单行注释的语法相同，它们以 `//` 开头，注释内容直到行末。

```
1. body {
2.   color: #333; // 这种注释内容不会出现在生成的css文件中
3.   padding: 0; /* 这种注释内容会出现在生成的css文件中 */
4. }
```

实际上， `css` 的标准注释格式 `/ … /` 内的注释内容亦可在生成的 `css` 文件中抹去。当注释出现在原生 `css` 不允许的地方，如在 `css` 属性或选择器中， `sass` 将不知如何将其生成到对应 `css` 文件中的相应位置，于是这些注释被抹掉。

```
1. body {
2.   color /* 这块注释内容不会出现在生成的css中 */: #333;
3.   padding: 1; /* 这块注释内容也不会出现在生成的css中 */ 0;
4. }
```

你已经掌握了 `sass` 的静默注释，了解了保持 `sass` 条理性和可读性的最基本的三个方法：嵌套、导入和注释。现在，我们要进一步学习新特性，这样我们不但能保持条理性还能写出更好的样式。首先要介绍的内容是：使用混合器抽象你的相关样式。

5. 混合器;

如果你的整个网站中有几处小小的样式类似（例如一致的颜色和字体），那么使用变量来统一处理这种情况是非常不错的选择。但是当你的样式变得越来越复杂，你需要大段大段的重用样式的代码，独立的变量就没办法应付这种情况了。你可以通过 `sass` 的混合器实现大段样式的重用。

混合器使用 `@mixin` 标识符定义。看上去很像其他的 `CSS @` 标识符，比如说 `@media` 或者 `@font-face` 。这个标识符给一大段样式赋予一个名字，这样你就可以轻易地通过引用这个名字重用这段样式。下边的这段 `sass` 代码，定义了一个非常简单的混合器，目的是添加跨浏览器的圆角边框。

```
@mixin rounded-corners {
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border-radius: 5px;
}
```

然后就可以在你的样式表中通过 `@include` 来使用这个混合器，放在你希望的任何地方。 `@include` 调用会把混合器中的所有样式提取出来放在 `@include` 被调用的地方。如果像下边这样写：

```
notice {
  background-color: green;
  border: 2px solid #00aa00;
  @include rounded-corners;
}

//sass最终生成：

```

```
.notice {
  background-color: green;
  border: 2px solid #00aa00;
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border-radius: 5px;
}
```

在 `.notice` 中的属性 `border-radius` `-moz-border-radius` 和 `-webkit-border-radius` 全部来自 `rounded-corners` 这个混合器。这一节将介绍使用混合器来避免重复。通过使用参数，你可以使用混合器把你样式中的通用样式抽离出来，然后轻松地在其他地方重用。实际上，混合器太好用了，一不小心你可能会过度使用。大量的重用可能会导致生成的样式表过大，导致加载缓慢。所以，首先我们将讨论混合器的使用场景，避免滥用。

5-1. 何时使用混合器;

利用混合器，可以很容易地在样式表的不同地方共享样式。如果你发现自己在不停地重复一段样式，那就应该把这段样式构造成优良的混合器，尤其是这段样式本身就是一个逻辑单元，比如说是一组放在一起有意义的属性。

判断一组属性是否应该组合成一个混合器，一条经验法则就是你能否为这个混合器想出一个好的名字。如果你能找到一个很好的短名字来描述这些属性修饰的样式，比如 `rounded-corners` `fancy-font` 或者 `no-bullets` ，那么往往能够构造一个合适的混合器。如果你找不到，这时候构造一个混合器可能并不合适。

混合器在某些方面跟 `css` 类很像。都是让你给一大段样式命名，所以在选择使用哪个的时候可能会产生疑惑。最重要的区别就是类名是在 `html` 文件中应用的，而混合器是在样式表中应用的。这就意味着类名具有语义化含义，而不仅仅是一种展示性的描述：用来描述 `html` 元素的含义而不是 `html` 元素的外观。而另一方面，混合器是展示性的描述，用来描述一条 `css` 规则应用之后会产生怎样的效果。

在之前的例子中， `.notice` 是一个有语义的类名。如果一个 `html` 元素有一个 `notice` 的类名，就表明了这个 `html` 元素的用途：向用户展示提醒信息。 `rounded-corners` 混合器是展示性的，它描述了包含它的 `css` 规则最终的视觉样式，尤其是边框角的视觉样式。混合器和类配合使用写出整洁的 `html` 和 `css` ，因为使用语义化的类名亦可以帮你避免重复使用混合器。为了保持你的 `html` 和 `css` 的易读性和可维护性，在写样式的过程中一定要铭记二者的区别。

有时候仅仅把属性放在混合器中还远远不够，可喜的是， `sass` 同样允许你把 `css` 规则放在混合器中。

5-2. 混合器中的CSS规则;

混合器中不仅可以包含属性，也可以包含 `css` 规则，包含选择器和选择器中的属性，如下代码：

```
@mixin no-bullets {
  list-style: none;
  li {
    list-style-image: none;
    list-style-type: none;
    margin-left: 0px;
  }
}
```

当一个包含 `css` 规则的混合器通过 `@include` 包含在一个父规则中时，在混合器中的规则最终会生成父规则中的嵌套规则。举个例子，看看下边的 `sass` 代码，这个例子中使用了 `no-bullets` 这个混合器：

```
ul.plain {
  color: #444;
  @include no-bullets;
}
```

`sass` 的 `@include` 指令会将引入混合器的那行代码替换成混合器里边的内容。最终，上边的例子如下代码：

```
ul.plain {
  color: #444;
  list-style: none;
}
ul.plain li {
  list-style-image: none;
  list-style-type: none;
  margin-left: 0px;
```

```
}
```

混合器中的规则甚至可以使用 `sass` 的父选择器标识符 `&` 。使用起来跟不用混合器时一样， `sass` 解开嵌套规则时，用父规则中的选择器替代 `&` 。

如果一个混合器只包含 `css` 规则，不包含属性，那么这个混合器就可以在文档的顶部调用，写在所有的 `css` 规则之外。如果你只是为自己写一些混合器，这并没有什么大的用途，但是当你使用一个类似于 `Compass` 的库时，你会发现，这是提供样式的好方法，原因在于你可以选择是否使用这些样式。

接下来你将学习如何通过给混合器传参数来让混合器变得更加灵活和可重用。

5-3． 给混合器传参；

混合器并不一定总得生成相同的样式。可以通过在 `@include` 混合器时给混合器传参，来定制混合器生成的精确样式。当 `@include` 混合器时，参数其实就是可以赋值给 `css` 属性值的变量。如果你写过 `JavaScript` ，这种方式跟 `JavaScript` 的 `function` 很像：

```
@mixin link-colors($normal, $hover, $visited) {
  color: $normal;
  &:hover { color: $hover; }
  &:visited { color: $visited; }
}
```

当混合器被 `@include` 时，你可以把它当作一个 `css` 函数来传参。如果你像下边这样写：

```
a {
  @include link-colors(blue, red, green);
}

//Sass最终生成的是：

a { color: blue; }
a:hover { color: red; }
a:visited { color: green; }
```

当你@include混合器时，有时候可能会很难区分每个参数是什么意思，参数之间是一个什么样的顺序。为了解决这个问题， `sass` 允许通过语法 `$name: value` 的形式指定每个参数的值。这种形式的传参，参数顺序就不必再在乎了，只需要保证没有漏掉参数即可：

```
a {
    @include link-colors(
      $normal: blue,
      $visited: green,
      $hover: red
  );
}
```

尽管给混合器加参数来实现定制很好，但是有时有些参数我们没有定制的需要，这时候也需要赋值一个变量就变成很

痛苦的事情了。所以 `sass` 允许混合器声明时给参数赋默认值。

5-4. 默认参数值;

为了在 `@include` 混合器时不必传入所有的参数，我们可以给参数指定一个默认值。参数默认值使用 `$name: default-value` 的声明形式，默认值可以是任何有效的 `css` 属性值，甚至是其他参数的引用，如下代码：

```
@mixin link-colors(
    $normal,
    $hover: $normal,
    $visited: $normal
  )
{
  color: $normal;
  &:hover { color: $hover; }
  &:visited { color: $visited; }
}
```

如果像下边这样调用： `@include link-colors(red)` `$hover` 和 `$visited` 也会被自动赋值为 `red` 。

混合器只是 `sass` 样式重用特性中的一个。我们已经了解到混合器主要用于样式展示层的重用，如果你想重用语义化的类呢？这就涉及 `sass` 的另一个重要的重用特性：选择器继承。

6. 使用选择器继承来精简CSS;

使用 `sass` 的时候，最后一个减少重复的主要特性就是选择器继承。基于 `Nicole Sullivan` 面向对象的 `css` 的理念，选择器继承是说一个选择器可以继承为另一个选择器定义的所有样式。这个通过 `@extend` 语法实现，如下代码：

```
//通过选择器继承继承样式
.error {
  border: 1px solid red;
  background-color: #fdd;
}
.seriousError {
  @extend .error;
  border-width: 3px;
}
```

在上边的代码中， `.seriousError` 将会继承样式表中任何位置处为 `.error` 定义的所有样式。
以 `class="seriousError"` 修饰的 `html` 元素最终的展示效果就好像是 `class="seriousError error"` 。相关元素不仅会拥有一个 `3px` 宽的边框，而且这个边框将变成红色的，这个元素同时还会有一个浅红色的背景，因为这些都是在 `.error` 里边定义的样式。

`.seriousError` 不仅会继承 `.error` 自身的所有样式，任何跟 `.error` 有关的组合选择器样式也会被 `.seriousError` 以组合选择器的形式继承，如下代码：

```
//.seriousError从.error继承样式
.error a{  //应用到.seriousError a
  color: red;
  font-weight: 100;
}
h1.error { //应用到hl.seriousError
  font-size: 1.2rem;
}
```

如上所示，在 `class="seriousError"` 的 `html` 元素内的超链接也会变成红色和粗体。

本节将介绍与混合器相比，哪种情况下更适合用继承。接下来在探索继承的工作细节之前，我们先了解一下继承的高级用法。最后，我们将看看使用继承可能会有哪些坑，学习如何避免这些坑。

6-1. 何时使用继承;

5-1节介绍了混合器主要用于展示性样式的重用，而类名用于语义化样式的重用。因为继承是基于类的（有时是基于其他类型的选择器），所以继承应该是建立在语义化的关系上。当一个元素拥有的类（比如说 `.seriousError` ）表明它属于另一个类（比如说 `.error` ），这时使用继承再合适不过了。

这有点抽象，所以我们从几个方面来阐释一下。想象一下你正在编写一个页面，给 `html` 元素添加类名，你发现你的某个类（比如说 `.seriousError` ）另一个类（比如说 `.error` ）的细化。你会怎么做？

- 你可以为这两个类分别写相同的样式，但是如果有大量的重复怎么办？使用 `sass` 时，我们提倡的就是不要做重

复的工作。

- 你可以使用一个选择器组（比如说 `.error` `.seriousError` ）给这两个选择器写相同的样式。如果.error的所有样式都在同一个地方，这种做法很好，但是如果是分散在样式表的不同地方呢？再这样做就困难多了。
- 你可以使用一个混合器为这两个类提供相同的样式，但当 `.error` 的样式修饰遍布样式表中各处时，这种做法面临着跟使用选择器组一样的问题。这两个类也不是恰好有相同的样式。你应该更清晰地表达这种关系。
- 综上所述你应该使用 `@extend` 。让 `.seriousError` 从 `.error` 继承样式，使两者之间的关系非常清晰。更重要的是无论你在样式表的哪里使用 `.error` `.seriousError` 都会继承其中的样式。 现在你已经更好地掌握了何时使用继承，以及继承有哪些突出的优点，接下来我们看看一些高级用法。

6-2. 继承的高级用法;

任何 `css` 规则都可以继承其他规则，几乎任何 `css` 规则也都可以被继承。大多数情况你可能只想对类使用继承，但是有些场合你可能想做得更多。最常用的一种高级用法是继承一个 `html` 元素的样式。尽管默认的浏览器样式不会被继承，因为它们不属于样式表中的样式，但是你对 `html` 元素添加的所有样式都会被继承。

接下来的这段代码定义了一个名为 `disabled` 的类，样式修饰使它看上去像一个灰掉的超链接。通过继承a这一超链接元素来实现：

```
1. .disabled {
2.   color: gray;
3.   @extend a;
4. }
```

假如一条样式规则继承了一个复杂的选择器，那么它只会继承这个复杂选择器命中的元素所应用的样式。举例来说，如果 `.seriousError` `@extend` `.important.error` ， 那么 `.important.error` 和 `h1.important.error` 的样式都会被 `.seriousError` 继承， 但是 `.important` 或者 `.error下` 的样式则不会被继承。这种情况下你很可能希望 `.seriousError` 能够分别继承 `.important` 或者 `.error` 下的样式。

如果一个选择器序列（ `#main .seriousError` ） `@extend` 另一个选择器（ `.error` ），那么只有完全匹配 `#main .seriousError` 这个选择器的元素才会继承 `.error` 的样式，就像单个类名继承那样。拥有 `class="seriousError"` 的 `#main` 元素之外的元素不会受到影响。

像 `#main .error` 这种选择器序列是不能被继承的。这是因为从 `#main .error` 中继承的样式一般情况下会跟直接从 `.error` 中继承的样式基本一致，细微的区别往往使人迷惑。

现在你已经了解了通过继承能够做些什么事情，接下来我们将学习继承的工作细节，在生成对应 `css` 的时候， `sass` 具体干了些什么事情。

6-3. 继承的工作细节;

跟变量和混合器不同，继承不是仅仅用 `css` 样式替换@extend处的代码那么简单。为了不让你对生成的 `css` 感觉奇怪，对这背后的工作原理有一定了解是非常重要的。

`@extend` 背后最基本的想法是，如果 `.seriousError @extend .error` ， 那么样式表中的任何一处 `.error` 都用 `.error` `.seriousError` 这一选择器组进行替换。这就意味着相关样式会如预期那样应用到 `.error` 和 `.seriousError` 。当 `.error` 出现在复杂的选择器中，比如说 `h1.error` `.error a` 或者 `#main .sidebar input.error[type="text"]` ，那情况就变得复杂多了，但是不用担心， `sass` 已经为你考虑到了这些。

关于 `@extend` 有两个要点你应该知道。

- 跟混合器相比，继承生成的 `css` 代码相对更少。因为继承仅仅是重复选择器，而不会重复属性，所以使用继承往往比混合器生成的 `css` 体积更小。如果你非常关心你站点的速度，请牢记这一点。
- 继承遵从 `css` 层叠的规则。当两个不同的 `css` 规则应用到同一个 `html` 元素上时，并且这两个不同的 `css` 规则对同一属性的修饰存在不同的值，`css` 层叠规则会决定应用哪个样式。相当直观：通常权重更高的选择器胜出，如果权重相同，定义在后边的规则胜出。 混合器本身不会引起 `css` 层叠的问题，因为混合器把样式直接放到了 `css` 规则中，而继承存在样式层叠的问题。被继承的样式会保持原有定义位置和选择器权重不变。通常来说这并不会引起什么问题，但是知道这点总没有坏处。

6-4. 使用继承的最佳实践;

通常使用继承会让你的 `css` 美观、整洁。因为继承只会在生成 `css` 时复制选择器，而不会复制大段的 `css` 属性。但是如果你不小心，可能会让生成的 `css` 中包含大量的选择器复制。

避免这种情况出现的最好方法就是不要在 `css` 规则中使用后代选择器（比如 `.foo .bar` ）去继承 `css` 规则。如果你这么做，同时被继承的 `css` 规则有通过后代选择器修饰的样式，生成 `css` 中的选择器的数量很快就会失控：

```
1. .foo .bar { @extend .baz; }
2. .bip .baz { a: b; }
```

在上边的例子中， `sass` 必须保证应用到.baz的样式同时也要应用到 `.foo .bar` （位于class="foo"的元素内的class="bar"的元素）。例子中有一条应用到 `.bip .baz` （位于class="bip"的元素内的class="baz"的元素）的 `css` 规则。当这条规则应用到 `.foo .bar` 时，可能存在三种情况，如下代码：

```
1. <!-- 继承可能迅速变复杂 -->
2. <!-- Case 1 -->
3. <div class="foo">
4.   <div class="bip">
5.     <div class="bar">...</div>
6.   </div>
7. </div>
8. <!-- Case 2 -->
9. <div class="bip">
10.   <div class="foo">
11.     <div class="bar">...</div>
12.   </div>
13. </div>
14. <!-- Case 3 -->
15. <div class="foo bip">
16.   <div class="bar">...</div>
17. </div>
```

为了应付这些情况， `sass` 必须生成三种选择器组合（仅仅是.bip .foo .bar不能覆盖所有情况）。如果任何一条规则里边的后代选择器再长一点， `sass` 需要考虑的情况就会更多。实际上 `sass` 并不总是会生成所有可能的选择器组合，即使是这样，选择器的个数依然可能会变得相当大，所以如果允许，尽可能避免这种用法。

值得一提的是，只要你想，你完全可以放心地继承有后代选择器修饰规则的选择器，不管后代选择器多长，但有一个

前提就是，不要用后代选择器去继承。

7. 小结;

本文介绍了 `sass` 最基本部分,你可以轻松地使用 `sass` 编写清晰、无冗余、语义化的 `css` 。对于 `sass` 提供的工具你已经有了一个比较深入的了解，同时也掌握了何时使用这些工具的指导原则。

变量是 `sass` 提供的最基本的工具。通过变量可以让独立的 `css` 值变得可重用，无论是在一条单独的规则范围内还是在整个样式表中。变量、混合器的命名甚至 `sass` 的文件名，可以互换通用 `_` 和 `-` 。同样基础的是 `sass` 的嵌套机制。嵌套允许 `css` 规则内嵌套 `css` 规则，减少重复编写常用的选择器，同时让样式表的结构一眼望去更加清晰。 `sass` 同时提供了特殊的父选择器标识符 `&` ，通过它可以构造出更高效的嵌套。

你也已经学到了 `sass` 的另一个重要特性，样式导入。通过样式导入可以把分散在多个 `sass` 文件中的内容合并生成到一个 `css` 文件，避免了项目中有大量的 `css` 文件通过原生的 `css` `@import` 带来的性能问题。通过嵌套导入和默认变量值，导入可以构建更强有力的、可定制的样式。混合器允许用户编写语义化样式的同时避免视觉层面上样式的重复。你不仅学到了如何使用混合器减少重复，同时学习到了如何使用混合器让你的 `css` 变得更加可维护和语义化。最后，我们学习了与混合器相辅相成的选择器继承。继承允许你声明类之间语义化的关系，通过这些关系可以保持你的 `css` 的整洁和可维护性。