Sass 入门篇

第 1 章 Sass 简介

===概念===

Sass 是一门高于 CSS 的元语言，它能用来清晰地、结构化地描述文件样式，有着比普通 CSS 更加强大的功能。

Sass 能够提供更简洁、更优雅的语法，同时提供多种功能来创建可维护和管理的样式表。

Sass 是采用 Ruby 语言编写的一款 CSS 预处理语言，它诞生于 2007 年，是最大的成熟的 CSS 预处理语言。最初它是为了配合 HAML（一种缩进式 HTML 预编译器）而设计的，因此有着和 HTML 一样的缩进式风格。

===Sass 和 SCSS 有什么区别？===

Sass 和 SCSS 其实是同一种东西，我们平时都称之为 Sass，两者之间不同之处有以下两点：

1、文件扩展名不同，Sass 是以“.sass”后缀为扩展名，而 SCSS 是以“.scss”后缀为扩展名

2、语法书写方式不同，Sass 是以严格的缩进式语法规则来书写，不带大括号({})和分号(;)，而 SCSS 的语法书写和我们的 CSS 语法书写方式非常类似。

示例：

---Sass 语法

```
$font-stack: Helvetica, sans-serif    //定义变量
$primary-color: #333 //定义变量

body
  font: 100% $font-stack
  color: $primary-
```

---SCSS 语法

```
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

===Sass 和 CSS 写法有差别：===

Sass 和 CSS 写法的确存在一定的差异，由于 Sass 是基于 Ruby 写出来，所以其延续了 Ruby 的书写规范。在书写 Sass 时不带有大括号和分号，其主要是依靠严格的缩进方式来控制的。如：

---Sass 写法：

```
body
  color: #fff
  background: #f36
```

--- CSS 写法：

```
body{
  color:#fff;
  background:#f36;
}
```

SCSS 和 CSS 写法无差别：

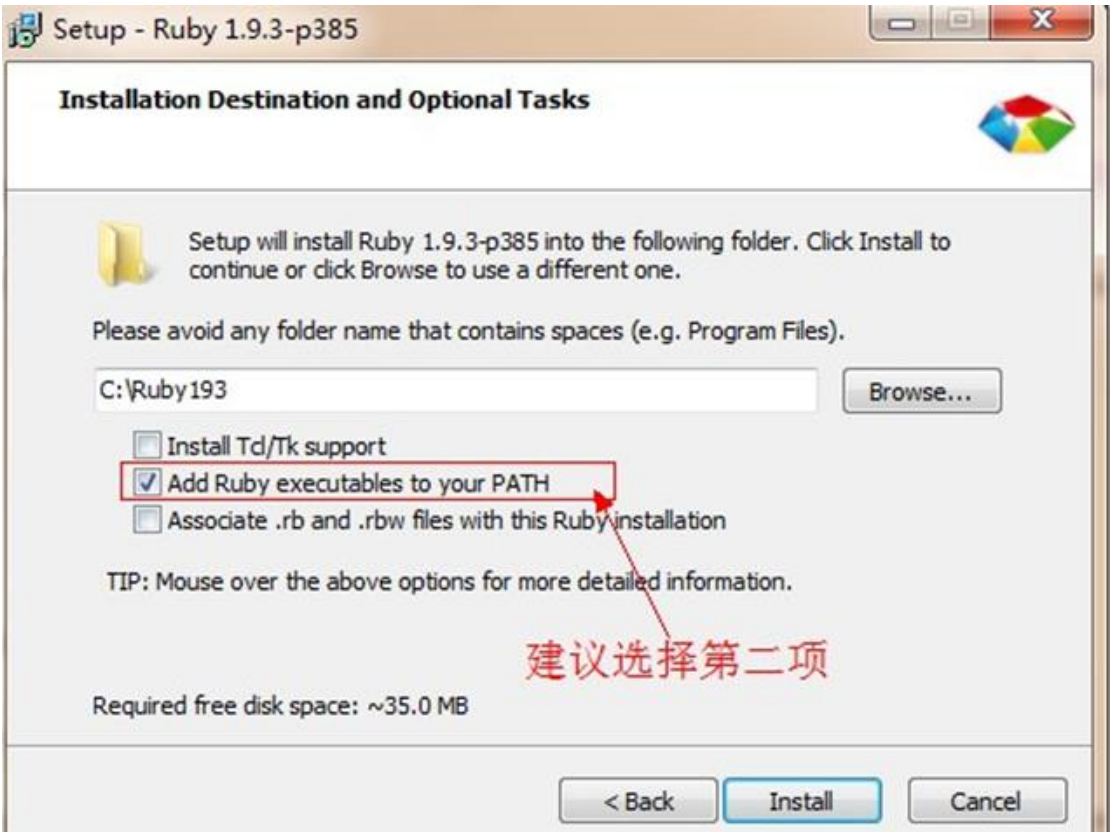
SCSS 和 CSS 写法无差别，这也是 Sass 后来越来越受大众喜欢原因之一。简单点说，把你现有的“.css”文件直接修改成“.scss”即可使用。

第 2 章 Sass 安装环境

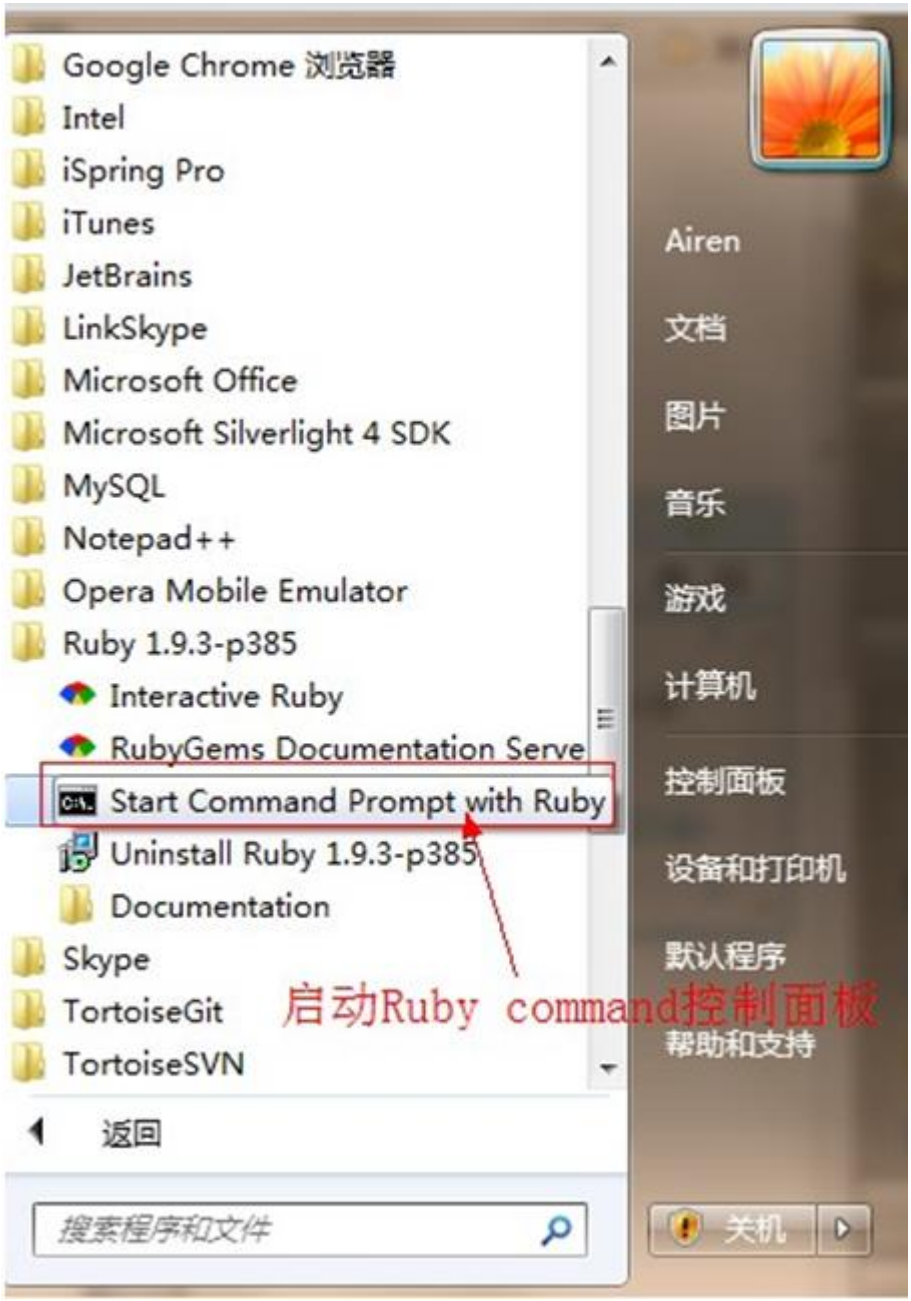
===安装指南（windows 版）===

---1、在 Windows 平台下安装 Ruby 需要先有 Ruby 安装包，到 Ruby 的官网（http://rubyinstaller.org/downloads）下载对应需要的 Ruby 版本。

---2、Ruby 安装文件下载好后，可以按应用软件安装步骤进行安装 Ruby。在安装过程中，建议将其安装在 C 盘下，在安装过程中选择第二个选项（不选中，就会出现编译时找不到 Ruby 环境的情况）。如下图所示：



---3、Ruby 安装完成后，在开始菜单中找到新安装的 Ruby，并启动 Ruby 的 Command 控制面板。



== 1、通过命令安装 Sass==

打开电脑的命令终端，输入下面的命令：

```
gem install sass
```

提醒一下，在使用 Mac 的同学，可能需要在上面的命令前加上"sudo"，才能正常安装：

```
sudo gem install sass
```

== 2、通过 Compass 来安装 Sass==

通过安装 compass 来安装 Sass，因为 Compass 是基于 Sass 开发的一个框架。也就是说，你安装了 Compass，也就同时安装好了 Sass。

同样的在你的命令终端输入下面的命令：

```
sudo gem install sass
```

执行完上面的命令之后，就开始安装 Compass 和 Sass。

注：Compass 是一个成熟的、基于 Sass 开发的一个框架，这里面集成了很多写好的 mixins 和 Sass 函数。不过在此暂不做过多阐述。

==3、本地安装 Sass==

由于有时候直接使用上面的命令安装会让你无法正常实现安装（网络受限原因），当碰到这种情况之时，那么安装需要特殊去处理，可以通过下面的方法来实现 Sass 的正常安装：

可以到 Rubygems(http://rubygems.org/) 网站上将 Sass 的安装包（http://rubygems.org/gems/sass）下载下来，然后在命令终端输入：

```
gem install <把下载的安装包拖到这里>
```

直接回车即可安装成功。

注：在 iOSX 系统平台，可以直接将下载的安装包拖到 "gem install" 后面，如果在是 Windows 系统，需要手功输入安装的文件路径。

==4、淘宝 RubyGems 镜像安装 Sass==

除了下载 Sass 安装包到本地安装之外，碰到网络原因无法安装时还可以使用淘宝 RubyGems 镜像安装 Sass。只是我们需要通过 gem sources 命令来配置源，先移除默认的 https://rubygems.org 源，然后添加淘宝的源 https://ruby.taobao.org：

第一步：移动默认的源

```
gem sources --remove https://rubygems.org/
```

第二步：指定淘宝的源

```
gem sources -a https://ruby.taobao.org/
```

第三步：查看指定的源是不是淘宝源

```
gem sources -l
```

返回结果如下：

```
*** CURRENT SOURCES ***
https://ruby.taobao.org
```

请确保只有 ruby.taobao.org。如果无误之后，执行下面的命令：

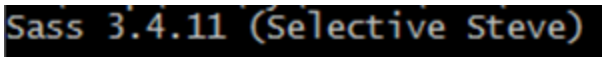
```
gem install sass
```

===查测 Sass 及更新===

通过上面的几种方法都可以安装 Sass，但是，我们要如何确认自己是否安装 Sass 成功了呢？其实很简单，只需要通过下面的命令即可：

```
sass -v
```

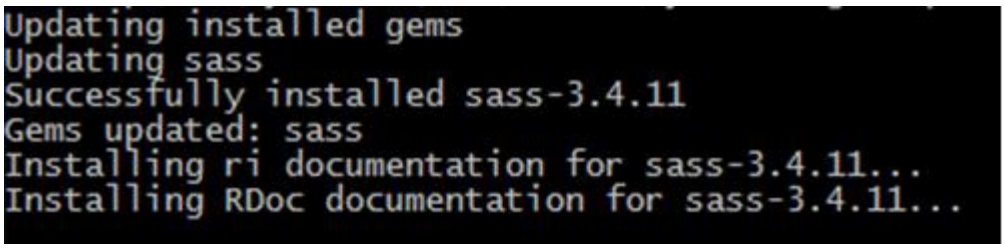
如果在你的命令终端能看到类似这样的信息就表示你的电脑安装 Sass 已成功。也就是说可以正常的使用 Sass 了。



--更新 Sass

维护 Sass 的团队会不断的为 Sass 添加新的功能，那么如何确保自己已安装的 Sass 也具有这些新的功能特性呢？不会是卸载了重新安装吧（虽然安装也就是一个命令的事情）？ 其实不需要这么麻烦，只需要在命令终端执行：

```
gem update sass
```



===卸载（删除）Sass===

在常期使用的时候难免会碰到无法解决的问题，有时候可能需要卸载 Sass，然后再重新安装 Sass。那么怎么卸载 Sass 呢？

```
gem uninstall sass
```

第 3 章 Sass 的语法格式及编译调试

===Sass 语法格式===

这里说的 Sass 语法是 Sass 的最初语法格式，他是通过 tab 键控制缩进的一种语法规则，而且这种缩进要求非常严格。另外其不带有任何的分号和大括号。常常把这种格式称为 Sass 老版本，其文件名以“.sass”为扩展名。

来看一个 Sass 语法格式的简单示例。假设我们有一段这样的 CSS 代码：

现在用 Sass 的语法格式来编写：

```
body {
        font: 100% Helvetica, sans-serif;
        color: #333;
    }
```

现在用 Sass 的语法格式来编写：

```
$font-stack: Helvetica, sans-serif
    $primary-color: #333
body
        font: 100% $font-stack
                color: $primary-color
```

在整个 Sass 代码中，我们没看到类似 CSS 中的大括号和分号。

注：这种语法格式对于前端人员都不太容易接受，而且容易出错。

==二、SCSS 语法格式==

SCSS 是 Sass 的新语法格式，从外形上来判断他和 CSS 长得几乎是一模一样，代码都包裹在一对大括号里，并且末尾结束处都有一个分号。其文件名格式常常以“.scss”为扩展名。

同样的这段 CSS 代码：

```
body {
        font: 100% Helvetica, sans-serif;
        color: #333;
}
```

我们使用 SCSS 语法格式将按下面这样来书写：

```
    $font-stack: Helvetica, sans-serif;
    $primary-color: #333;
    body {
        font: 100% $font-stack;
        color: $primary-color;
    }
```

在此特别提醒：“.sass”只能使用 Sass 老语法规则（缩进规则），“.scss”使用的是 Sass 的新语法规则，也就是 SCSS 语法规则（类似 CSS 语法格式）。

===Sass 编译===

在项目中还是引用“.css”文件，Sass 只不过是做为一个预处理工具，提前帮你做事情，只有你需要时候，他才有攻效。

Sass 的编译。因为 Sass 开发之后，要让 Web 页面能调用 Sass 写好的东西，就得有这么一个过程，这个过程就称之为 Sass 编译过程。
Sass 的编译有多种方法：

-命令编译
-GUI 工具编译
-自动化编译

****命令编译****

命令编译是指使用你电脑中的命令终端，通过输入 Sass 指令来编译 Sass。这种编译方式是最直接也是最简单的一种方式。因为只需要在你的命令终端输入：

单文件编译：

sass <要编译的 Sass 文件路径>/style.scss:<要输出 CSS 文件路

径>/style.css

这是对一个单文件进行编译，如果想对整个项目所有 Sass 文件编译成 CSS 文件，可以这样操作：

多文件编译：

```
sass sass/:css/
```

上面的命令表示将项目中“sass”文件夹中所有“.scss”(“.sass”)文件编译成“.css”文件，并且将这些 CSS 文件都放在项目中“css”文件夹中。

--缺点及解决方法：

在实际编译过程中，你会发现上面的命令，只能一次性编译。每次个性保存“.scss”文件之后，都得重新执行一次这样的命令。如此操作太麻烦，其实还有一种方法，就是在编译 Sass 时，开启“watch”功能，这样只要你的代码进行任保修改，都能自动监测到代码的变化，并且给你直接编译出来：

```
sass --watch <要编译的 Sass 文件路径>/style.scss:<要输出 CSS 文件路径>/style.css
```

当然，使用 sass 命令编译时，可以带很多的参数：

watch 举例：

来看一个简单的示例，假设我本地有一个项目，我要把项目中“bootstrap.scss”编译出“bootstrap.css”文件，并且将编译出来的文件放在“css”文件夹中，我就可以在我的命令终端中执行：

```
sass --watch
sass/bootstrap.scss:css/bootstrap.css
```

一旦我的 bootstrap.scss 文件有任何修改，只要我重新保存了修改的文件，命令终端就能监测，并重新编译出文件

*****GUI 界面工具编译*****

或许你会说，我一直讨厌使用命令来做事情，我喜欢那种能看得到的界面操作。那么你可以考虑使用 GUI 界面工具来对 Sass 进行编译。当然不同的 GUI 工具操作方法略有不同。如果在此也一一对编译的界面工具做详细的介绍。我们可能需要写一本书来介绍这些编译工具的操作了。所以我们这里做一下简单介绍，对于 GUI 界面编译工具，目前较为流行的主要有：

-1、Koala (http://koala-app.com/)
-2、Compass.app（http://compass.kkbox.com/）
-3、Scout（http://mhs.github.io/scout-app/）
-4、CodeKit（https://incident57.com/codekit/index.html）
-5、Prepros（https://prepros.io/）

相比之下，我比较推荐使用以下两个：

- Koala (http://www.w3cplus.com/preprocessor/sass-gui-tool-koala.html)
- CodeKit (http://www.w3cplus.com/preprocessor/sass-gui-tool-codekit.html)

自动化编译

如果喜欢自动化的研究，应该都知道 Grunt 和 Gulp 这两个东东。如果您正在使用其中的任何一种，那么你也可以通过他们来配置 Sass 的编译。这里仅列出两个示例代码（具体情况要根据您的项目环境来做一定的修改，不建议生搬硬套，容易发生命案，呵呵。

--1、Grunt 配置 Sass 编译的示例代码

```
module.exports = function(grunt) {
                    grunt.initConfig({
                        pkg:
                grunt.file.readJSON('package.json'),
                        sass: {
                            dist: {
                                files: {
                                    'style/style.css' :
                                    'sass/style.scss'
                                        }
                                    }
                                },
                                watch: {
                                    css: {
                                        files: '**/*.scss',
                                        tasks: ['sass']
                                    }
                                }
                            });
                        grunt.loadNpmTasks('grunt-contrib-sass');
                        grunt.loadNpmTasks('grunt-contrib-watch');
                            grunt.registerTask('default',['watch']);
                        }
```

--2、Gulp 配置 Sass 编译的示例代码

```
var gulp = require('gulp');
var sass = require('gulp-sass');

gulp.task('sass', function () {
    gulp.src('./scss/*.scss')
        .pipe(sass())
        .pipe(gulp.dest('./css'));
});

gulp.task('watch', function() {
    gulp.watch('scss/*.scss', ['sass']);
});

gulp.task('default', ['sass','watch']);
```

=== 常见的编译错误 ===

在编译 Sass 代码时常常会碰到一些错误，让编译失败。这样的错误有系统造成的也有人为造成的，但大部分都是人为过失引起编译失败。

而最为常见的一个错误就是字符编译引起的。在 Sass 的编译的过程中，是不是支持“GBK”编码的。所以在创建 Sass 文件时，就需要将文件编码设置为“utf-8”。

另外一个错误就是路径中的中文字符引起的。建议在项目中文件命名或者文件目录命名不要使用中文字符。而至于人为失误造成的编译失败，在编译过程中都会有具体的说明，大家可以根据编译器提供的错误信息进行对应的修改。

=== 不同样式风格的输出方法 ===

众所周知，每个人编写的 CSS 样式风格都不一样，有的喜欢将所有样式代码都写在同一行，而有的喜欢将样式分行书写。在 Sass 中编译出来的样式风格也可以按不同的样式风格显示。其主要包括以下几种样式风格：

--1、嵌套输出方式 nested
--2、展开输出方式 expanded
--3、紧凑输出方式 compact
--4、压缩输出方式 compressed

*** 嵌套输出方式 nested ***

1、嵌套输出方式 nested

Sass 提供了一种嵌套显示 CSS 文件的方式。例如

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

在编译的时候带上参数“ --style nested”：

```
sass --watch test.scss:test.css --style nested
```

编译出来的 CSS 样式风格：

```
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
  }

nav li {
  display: inline-block;
  }

nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
  }
```

*** 展开输出方式 expanded ***

2、嵌套输出方式 expanded

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

在编译的时候带上参数“ --style expanded”：

```
sass --watch test.scss:test.css --style expanded
```

这个输出的 CSS 样式风格和 nested 类似，只是大括号在另起一行，同样上面的代码，编译出来：

```
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}
nav li {
  display: inline-block;
}
nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}
```

*** 紧凑输出方式 compact ***

2、嵌套输出方式 compact

```
nav {
        ul {
           margin: 0;
           padding: 0;
           list-style: none;
        }

        li { display: inline-block; }

        a {
           display: block;
           padding: 6px 12px;
           text-decoration: none;
        }
      }
```

在编译的时候带上参数“ --style compact”：

```
sass --watch test.scss:test.css --style compact
```

该方式适合那些喜欢单行 CSS 样式格式的朋友，编译后的代码如下：

```
nav ul {
      margin: 0;
      padding: 0;
      list-style: none;
}

nav li {
      display: inline-block;
}
nav a {
      display: block;
      padding: 6px 12px;
      text-decoration: none;
}
```

***** 压缩输出方式 compressed *****

2、压缩输出方式 compressed

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

在编译的时候带上参数“ --style compressed”：

```
sass --watch test.scss:test.css --style compressed
```

压缩输出方式会去掉标准的 Sass 和 CSS 注释及空格。也就是压缩好的 CSS 代码样式风格：

```
nav ul{
    margin:0;
    padding:0;
    list-style:none
}
nav li{
    display:inline-block
}
nav a{
    display:block;
    padding:6px 12px;
    text-decoration:none
}
```

编译出来的 CSS 样式风格的选择完全是个人喜好问题，可以根据自己喜欢的风格选择参数。

一段时间之后，你实际上就不再需要写 CSS 代码了，只用写 Sass 代码。在这种情况下，你只需要设定输出格式为压缩格式，知道输出的 CSS 代码可以直接使用即可。

===Sass 的调试===

Sass 调试一直以来都是一件头痛的事情，使用 Sass 的同学都希望能在浏览器中直接调试 Sass 文件，能找到对应的行数。值得庆幸的是，现在实现并不是一件难事，只要你的浏览器支持“sourcemap”功能即可。早一点的版本，需要在编译的时候添加“--sourcemap” 参数：

```
sass --watch --scss --sourcemap
style.scss:style.css
```

在 Sass3.3 版本之上（我测试使用的版本是 3.4.7），不需要添加这个参数也可以：

```
sass --watch style.scss:style.css
```

在命令终端，你将看到一个信息：

```
>>> Change detected to: style.scss
   write style.css
   write style.css.map
```

这时你就可以调试你的 Sass 代码。

第 4 章 Sass 的基本特性-基础

=== 声明变量 ===

定义变量的语法：

在有些编程语言中（如，JavaScript）声明变量都是使用关键词“var”开头，但是在 Sass 不使用这个关键词，而是使用大家都喜欢的美元符号“$”开头。我想用一张图来解释，我一直坚信，一图胜千言万语：

上图非常清楚告诉了大家，Sass 的变量包括三个部分：

1、声明变量的符号“$”
2、变量名称
3、赋予变量的值

来看一个简单的示例，假设你的按钮颜色可以给其声明几个变量：

```
$brand-primary : darken(#428bca, 6.5%) !default; // #337ab7
$btn-primary-color : #fff !default;
$btn-primary-bg : $brand-primary !default;
$btn-primary-border : darken($btn-primary-bg, 5%) !default;
```

如果值后面加上!default 则表示默认值。

注：了解 Bootstrap 的 Sass 版本的同学，就一眼能看出，上面的示例代码是 Bootstrap 定义 primarybutton 的颜色。

=== 普通变量与默认变量 ===

--普通变量

定义之后可以在全局范围内使用。

```
$fontSize: 12px;
body{
    font-size:$fontSize;
}
```

编译后的 css 代码：

```
body{
```

```
    font-size:12px;
}
```

--默认变量

sass 的默认变量仅需要在值后面加上 !default 即可。

```
$baseLineHeight:1.5 !default;
body{
    line-height: $baseLineHeight;
}
```

编译后的 css 代码：

```
body{
    line-height:1.5;
}
```

sass 的默认变量一般是用来设置默认值，然后根据需求来覆盖的，覆盖的方式也很简单，只需要在默认变量之前重新声明下变量即可。

```
$baseLineHeight: 2;
$baseLineHeight: 1.5 !default;
body{
    line-height: $baseLineHeight;
}
```

编译后的 css 代码：

```
body{
    line-height:2;
}
```

可以看出现在编译后的 line-height 为 2，而不是我们默认的 1.5。默认变量的价值在进行组件化开发的时候会非常有用。

=== 变量的调用 ===

在 Sass 中声明了变量之后，就可以在需要的地方调用变量。调用变量的方法也非常的简单。

比如在定义了变量

```
$brand-primary : darken(#428bca, 6.5%) !default; // #337ab7
$btn-primary-color: #fff !default;
$btn-primary-bg : $brand-primary !default;
$btn-primary-border : darken($btn-primary-bg, 5%) !default;
```

在按钮 button 中调用，可以按下面的方式调用

```
.btn-primary {
    background-color: $btn-primary-bg;
    color: $btn-primary-color;
    border: 1px solid $btn-primary-border;
}
```

编译出来的 CSS:

```
.btn-primary {
     background-color: #337ab7;
     color: #fff;
     border: 1px solid #2e6da4;
}
```

=== 局部变量和全局变量 ===

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，从 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。

--全局变量与局部变量

先来看一下代码例子：

```
//SCSS
$color: orange !default;//定义全局变量(在选择器、函数、混合宏...的外面定义的变量为全局变量)
.block {
   color: $color;//调用全局变量
}
em {
   $color: red;//定义局部变量
   a {
      color: $color;//调用局部变量
   }
}
```

```
    span {
      color: $color;//调用全局变量
    }
```

css 的结果：

```
    //CSS
    .block {
      color: orange;
    }
    em a {
      color: red;
    }
    span {
      color: orange;
    }
```

上面的示例演示可以得知，在元素内部定义的变量不会影响其他元素。如此可以简单的理解成，全局变量就是定义在元素外面的变量，如下代码：

```
    $color:orange !default;
```

$color 就是一个全局变量，而定义在元素内部的变量，比如 $color:red; 是一个局部变量。

除此之外，Sass 现在还提供一个 !global 参数。!global 和 !default 对于定义变量都是很有帮助的。我们之后将会详细介绍这两个参数的使用以及其功能。

--全局变量的影子

当在局部范围（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。

上面例子中的 em 选择器内的变量 $color 就是一个全局变量的影子。

```
    //SCSS
$color: orange !default;//定义全局变量
.block {
  color: $color;//调用全局变量
}
em {
  $color: red;//定义局部变量（全局变量 $color 的影子）
  a {
```

```
        color: $color;//调用局部变量
    }
}
```

--什么时候声明变量？

我的建议，创建变量只适用于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：

1、该值至少重复出现了两次；
2、该值至少可能会被更新一次；
3、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。

温馨小提示：您在学习 sass 时，除了在我们网页上可以做练习，还有一个便利在线编辑器网址如下：

http://sassmeister.com/

=== 嵌套-选择器嵌套 ===

Sass 中还提供了选择器嵌套功能，但这也并不意味着你在 Sass 中的嵌套是无节制的，因为你嵌套的层级越深，编译出来的 CSS 代码的选择器层级将越深，这往往是大家不愿意看到的一点。这个特性现在正被众多开发者滥用。

选择器嵌套为样式表的作者提供了一个通过局部选择器相互嵌套实现全局选择的方法，Sass 的嵌套分为三种：

1、选择器嵌套
2、属性嵌套
3、伪类嵌套

1、选择器嵌套

假设我们有一段这样的结构：

```
<header>
<nav>
    <a href="##">Home</a>
    <a href="##">About</a>
    <a href="##">Blog</a>
```

```
</nav>
<header>
```

想选中 header 中的 a 标签，在写 CSS 会这样写：

```
nav a {
  color:red;
}

header nav a {
  color:green;
}
```

那么在 Sass 中，就可以使用选择器的嵌套来实现：

```
nav {
  a {
    color: red;

    header & {
      color:green;
    }
  }
}
```

=== 嵌套-属性嵌套 ===

Sass 中还提供属性嵌套，CSS 有一些属性前缀相同，只是后缀不一样，比如：border-top/border-right，与这个类似的还有 margin、padding、font 等属性。假设你的样式中用到了：

```
.box {
    border-top: 1px solid red;
    border-bottom: 1px solid green;
}
```

在 Sass 中我们可以这样写：

```
.box {
  border: {
    top: 1px solid red;
    bottom: 1px solid green;
  }
```

```
}
```

=== 嵌套-伪类嵌套 ===

其实伪类嵌套和属性嵌套非常类似，只不过他需要借助`&`符号一起配合使用。我们就拿经典的“clearfix”为例吧：

```
.clearfix{
&:before,
&:after {
    content:"";
    display: table;
  }
&:after {
    clear:both;
    overflow: hidden;
  }
}
```

编译出来的 CSS：

```
clearfix:before, .clearfix:after {
  content: "";
  display: table;
}
.clearfix:after {
  clear: both;
  overflow: hidden;
}
```

避免选择器嵌套：

1、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的选择器具体的表现效果。

2、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，我们应该尽可能避免选择器嵌套。然而，显然只有少数情况适应这一措施。

=== 混合宏-声明混合宏 ===

如果你的整个网站中有几处小样式类似，比如颜色，字体等，在 Sass 可以使用变量来统一处理，那么这种选择还是不错的。但当你的样式变得越来越复杂，需要重复使用大段的样式时，使用变量就无法达到我们目了。这个时候 Sass 中的混合宏就会变得非常有意义。在这一节中，主要向大家介绍 Sass 的混合宏。

1、声明混合宏

--不带参数混合宏：

在 Sass 中，使用“@mixin”来声明一个混合宏。如：

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

其中 @mixin 是用来声明混合宏的关键词，有点类似 CSS 中的 @media、@font-face 一样。border-radius 是混合宏的名称。大括号里面是复用的样式代码。

--带参数混合宏：

除了声明一个不带参数的混合宏之外，还可以在定义混合宏时带有参数，如：

```
@mixin border-radius($radius:5px){
    -webkit-border-radius: $radius;
    border-radius: $radius;
}
```

--复杂的混合宏：

上面是一个简单的定义混合宏的方法，当然， Sass 中的混合宏还提供更为复杂的，你可以在大括号里面写上带有逻辑关系，帮助更好的做你想做的事情,如：

```
@mixin box-shadow($shadow...) {
  @if length($shadow) >= 1 {
    @include prefixer(box-shadow, $shadow);
  } @else{
    $shadow:0 0 4px rgba(0,0,0,.3);
    @include prefixer(box-shadow, $shadow);
  }
}
```

这个 box-shadow 的混合宏，带有多个参数，这个时候可以使用“ … ”来替代。简单的解释一下，当 $shadow 的参数数量值大于或等于“ 1 ”时，表示有多个阴

影值，反之调用默认的参数值“ 0 0 4px rgba(0,0,0,.3) ”。

注：复杂的混合宏中的逻辑关系（@if...@else）后面小节会有讲解。

=== 混合宏-调用混合宏 ===

在 Sass 中通过 @mixin 关键词声明了一个混合宏，那么在实际调用中，其匹配了一个关键词“@include”来调用声明好的混合宏。例如在你的样式中定义了一个圆角的混合宏“border-radius”：

```
@mixin border-radius{
    -webkit-border-radius: 3px;
    border-radius: 3px;
}
```

在一个按钮中要调用定义好的混合宏“border-radius”，可以这样使用：

```
button {
    @include border-radius;
}
```

这个时候编译出来的 CSS:

```
button {
  -webkit-border-radius: 3px;
  border-radius: 3px;
}
```

=== 混合宏的参数--传一个不带值的参数 ===

Sass 的混合宏有一个强大的功能，可以传参，那么在 Sass 中传参主要有以下几种情形：

A) 传一个不带值的参数

在混合宏中，可以传一个不带任何值的参数，比如：

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

在混合宏“border-radius”中定义了一个不带任何值的参数“$radius”。

在调用的时候可以给这个混合宏传一个参数值：

```
.box {
  @include border-radius(3px);
}
```

这里表示给混合宏传递了一个“border-radius”的值为“3px”。

编译出来的 CSS:

```
.box {
  -webkit-border-radius: 3px;
  border-radius: 3px;
}
```

=== 混合宏的参数--传一个带值的参数 ===

在 Sass 的混合宏中，还可以给混合宏的参数传一个默认值，例如：

```
@mixin border-radius($radius:3px){
  -webkit-border-radius: $radius;
  border-radius: $radius;
}
```

在混合宏“border-radius”传了一个参数“$radius”，而且给这个参数赋予了一个默认值“3px”。

在调用类似这样的混合宏时，会多有一个机会，假设你的页面中的圆角很多地方都是“3px”的圆角，那么这个时候只需要调用默认的混合宏“border-radius”：

```
.btn {
  @include border-radius;
}
```

编译出来的 CSS:

```
.btn {
  -webkit-border-radius: 3px;
  border-radius: 3px;
}
```

但有的时候，页面中有些元素的圆角值不一样，那么可以随机给混合宏传值，如：

```
.box {
  @include border-radius(50%);
}
```

编译出来的 CSS:

```
.box {
  -webkit-border-radius: 50%;
  border-radius: 50%;
}
```

=== 混合宏的参数--传多个参数 ===

Sass 混合宏除了能传一个参数之外，还可以传多个参数，如：

```
@mixin center($width,$height){
  width: $width;
  height: $height;
  position: absolute;
  top: 50%;
  left: 50%;
  margin-top: -($height) / 2;
  margin-left: -($width) / 2;
}
```

在混合宏“center”就传了多个参数。在实际调用和其调用其他混合宏是一样的：

```
.box-center {
  @include center(500px,300px);
}
```

编译出来 CSS:

```
.box-center {
  width: 500px;
  height: 300px;
  position: absolute;
  top: 50%;
  left: 50%;
  margin-top: -150px;
  margin-left: -250px;
}
```

有一个特别的参数“…”。当混合宏传的参数过多之时，可以使用参数来替代，如：

```
@mixin box-shadow($shadows...){
  @if length($shadows) >= 1 {
    -webkit-box-shadow: $shadows;
    box-shadow: $shadows;
  } @else {
    $shadows: 0 0 2px rgba(#000,.25);
    -webkit-box-shadow: $shadow;
    box-shadow: $shadow;
  }
}
```

在实际调用中：

```
.box {
  @include box-shadow(0 0 1px rgba(#000,.5),0 0 2px rgba(#000,.2));
}
```

编译出来的 CSS:

```
.box {
  -webkit-box-shadow: 0 0 1px rgba(0, 0, 0, 0.5), 0 0 2px rgba(0, 0, 0, 0.2);
  box-shadow: 0 0 1px rgba(0, 0, 0, 0.5), 0 0 2px rgba(0, 0, 0, 0.2);
}
```

=== 混合宏的参数--混合宏的不足 ===

混合宏在实际编码中给我们带来很多方便之处，特别是对于复用重复代码块。但其最大的不足之处是会生成冗余的代码块。比如在不同的地方调用一个相同的混合宏时。如：

```
@mixin border-radius{
  -webkit-border-radius: 3px;
  border-radius: 3px;
}

.box {
  @include border-radius;
  margin-bottom: 5px;
}
```

```
.btn {
  @include border-radius;
}
```

示例在“.box”和“.btn”中都调用了定义好的“border-radius”混合宏。先来看编译出来的 CSS：

```
.box {
  -webkit-border-radius: 3px;
  border-radius: 3px;
  margin-bottom: 5px;
}

.btn {
  -webkit-border-radius: 3px;
  border-radius: 3px;
}
```

上例明显可以看出，Sass 在调用相同的混合宏时，并不能智能的将相同的样式代码块合并在一起。这也是 Sass 的混合宏最不足之处。

=== 扩展/继承 ===

继承对于了解 CSS 的同学来说一点都不陌生，先来看一张图：

图中代码显示“.col-sub .block li,.col-extra .block li” 继承了 “.item-list ul li”选择器的 “padding : 0;” 和 “ul li” 选择器中的 “list-style : none outside none;”以及 * 选择器中的 “box-sizing:inherit;”。

在 Sass 中也具有继承一说，也是继承类中的样式代码块。在 Sass 中是通过关键词 “@extend”来继承已存在的类样式块，从而实现代码的继承。如下所示：

```
//SCSS
.btn {
  border: 1px solid #ccc;
  padding: 6px 10px;
  font-size: 14px;
}

.btn-primary {
  background-color: #f36;
  color: #fff;
  @extend .btn;
```

```
}

.btn-second {
    background-color: orange;
    color: #fff;
    @extend .btn;
}
```

编译出来之后：

```
//CSS
.btn, .btn-primary, .btn-second {
    border: 1px solid #ccc;
    padding: 6px 10px;
    font-size: 14px;
}

.btn-primary {
    background-color: #f36;
    color: #fff;
}

.btn-second {
    background-clor: orange;
    color: #fff;
}
```

从示例代码可以看出，在 Sass 中的继承，可以继承类样式块中所有样式代码，而且编译出来的 CSS 会将选择器合并在一起，形成组合选择器：

```
.btn, .btn-primary, .btn-second {
    border: 1px solid #ccc;
    padding: 6px 10px;
    font-size: 14px;
}
```

=== 占位符 %placeholder ===

Sass 中的占位符 %placeholder 功能是一个很强大，很实用的一个功能，这也是我非常喜欢的功能。他可以取代以前 CSS 中的基类造成的代码冗余的情形。因为 %placeholder 声明的代码，如果不被 @extend 调用的话，不会产生任何代码。来看一个演示：

```
%mt5 {
  margin-top: 5px;
}
%pt5{
  padding-top: 5px;
}
```

这段代码没有被 @extend 调用，他并没有产生任何代码块，只是静静的躺在你的某个 SCSS 文件中。只有通过 @extend 调用才会产生代码：

```
    //SCSS
%mt5 {
  margin-top: 5px;
}
%pt5{
  padding-top: 5px;
}

.btn {
  @extend %mt5;
  @extend %pt5;
}

.block {
  @extend %mt5;

  span {
    @extend %pt5;
  }
}
```

编译出来的 CSS

```
//CSS
.btn, .block {
  margin-top: 5px;
}

.btn, .block span {
  padding-top: 5px;
}
```

从编译出来的 CSS 代码可以看出，通过 @extend 调用的占位符，编译出来的代码会将相同的代码合并在一起。这也是我们希望看到的效果，也让你的代码变得更为干净。

=== 混合宏 VS 继承 VS 占位符 ===

初学者都常常纠结于这个问题“什么时候用混合宏，什么时候用继承，什么时候使用占位符？”其实他们各有各的优点与缺点，先来看看他们使用效果：

a) Sass 中的混合宏使用

举例代码见右侧 2-24 行

编译出来的 CSS 见右侧结果窗口。

总结：编译出来的 CSS 清晰告诉了大家，他不会自动合并相同的样式代码，如果在样式文件中调用同一个混合宏，会产生多个对应的样式代码，造成代码的冗余，这也是 CSSer 无法忍受的一件事情。不过他并不是一无事处，他可以传参数。

个人建议：如果你的代码块中涉及到变量，建议使用混合宏来创建相同的代码块。

b) Sass 中继承

同样的，将上面代码中的混合宏，使用类名来表示，然后通过继承来调用：

代码见右侧 26-48 行

总结：使用继承后，编译出来的 CSS 会将使用继承的代码块合并到一起，通过组合选择器的方式向大家展现，比如 .mt, .block, .block span, .header, .header span。这样编译出来的代码相对于混合宏来说要干净的多，也是 CSSer 期望看到。但是他不能传变量参数。

个人建议：如果你的代码块不需要专任何变量参数，而且有一个基类已在文件中存在，那么建议使用 Sass 的继承。

c) 占位符

最后来看占位符，将上面代码中的基类 .mt 换成 Sass 的占位符格式：

代码见右侧 50-72 行

总结：编译出来的 CSS 代码和使用继承基本上是相同，只是不会在代码中生成占位符 mt 的选择器。那么占位符和继承的主要区别的，“占位符是独立定义，不调用的时候是不会在 CSS 中产生任何代码；继承是首先有一个基类存在，不管调用与不调用，基类的样式都将会出现在编译出来的 CSS 代码中。”

来看一个表格：

=== 插值#{} ===

使用 CSS 预处理器语言的一个主要原因是想使用 Sass 获得一个更好的结构体系。比如说你想写更干净的、高效的和面向对象的 CSS。Sass 中的插值(Interpolation)就是重要的一部分。让我们看一下下面的例子：

```
$properties: (margin, padding);
@mixin set-value($side, $value) {
    @each $prop in $properties {
        #{$prop}-#{$side}: $value;
    }
}
.login-box {
    @include set-value(top, 14px);
}
```

@each...in...语句会在《Sass 进级篇》中 1-6 @each 循环 中讲解。

它可以让变量和属性工作的很完美，上面的代码编译成 CSS：

```
.login-box {
    margin-top: 14px;
    padding-top: 14px;
}
```

这是 Sass 插值中一个简单的实例。当你想设置属性值的时候你可以使用字符串插入进来。另一个有用的用法是构建一个选择器。可以这样使用：

```
@mixin generate-sizes($class, $small, $medium, $big) {
    .#{$class}-small { font-size: $small; }
    .#{$class}-medium { font-size: $medium; }
    .#{$class}-big { font-size: $big; }
}
@include generate-sizes("header-text", 12px, 20px, 40px);
```

编译出来的 CSS:

```
.header-text-small { font-size: 12px; }
.header-text-medium { font-size: 20px; }
.header-text-big { font-size: 40px; }
```

一旦你发现这一点，你就会想到超级酷的 mixins，用来生成代码或者生成另一个 mixins。然而，这并不完全是可能的。第一个限制，这可能会很删除用于 Sass 变量的插值。

```
$margin-big: 40px;
$margin-medium: 20px;
$margin-small: 12px;
@mixin set-value($size) {
    margin-top: $margin-#{$size};
}
.login-box {
    @include set-value(big);
}
```

上面的 Sass 代码编译出来，你会得到下面的信息：

```
error style.scss (Line 5: Undefined variable: “$margin-".)
```

所以，#{}语法并不是随处可用，你也不能在 mixin 中调用：

```
@mixin updated-status {
    margin-top: 20px;
    background: #F00;
}
$flag: "status";
.navigation {
    @include updated-#{$flag};
}
```

上面的代码在编译成 CSS 时同样会报错：

```
error style.scss (Line 7: Invalid CSS after "...nclude updated-": expected "}", was "#{$flag};")
```

幸运的是，可以使用 @extend 中使用插值。例如：

```
%updated-status {
    margin-top: 20px;
    background: #F00;
}
.selected-status {
    font-weight: bold;
}
$flag: "status";
.navigation {
    @extend %updated-#{$flag};
    @extend .selected-#{$flag};
```

```
}
```

上面的 Sass 代码是可以运行的，因为他给了我们力量，可以动态的插入 .class 和 %placeholder。当然他们不能接受像 mixin 这样的参数，上面的代码编译出来的 CSS:

```
.navigation {
    margin-top: 20px;
    background: #F00;
}
.selected-status, .navigation {
    font-weight: bold;
}
```

在 Sass 的社区正在积极讨论插值的局限性，谁又知道呢，也许我们很快将能够使用这些技术也说不定呢。

=== 注释 ===

注释对于一名程序员来说，是极其重要，良好的注释能帮助自己或者别人阅读源码。在 Sass 中注释有两种方式，我暂且将其命名为：

1、类似 CSS 的注释方式，使用 ”/* ”开头，结属使用 ”*/ ”
2、类似 JavaScript 的注释方式，使用“//”

两者区别，前者会在编译出来的 CSS 显示，后者在编译出来的 CSS 中不会显示，来看一个示例：

```
//定义一个占位符

%mt5 {
  margin-top: 5px;
}

/*调用一个占位符*/

.box {
  @extend %mt5;
}
```

编译出来的 CSS

```
.box {
  margin-top: 5px;
```

```
}

/*调用一个占位符*/
```

=== 数据类型 ===

Sass 和 JavaScript 语言类似，也具有自己的数据类型，在 Sass 中包含以下几种数据类型：

1、数字: 如，1、 2、 13、 10px；
2、字符串：有引号字符串或无引号字符串，如，"foo"、 'bar'、 baz；
3、颜色：如，blue、 #04a3f9、 rgba(255,0,0,0.5)；
4、布尔型：如，true、 false；
5、空值：如，null；
6、值列表：用空格或者逗号分开，如，1.5em 1em 0 2em 、Helvetica, Arial, sans-serif。

SassScript 也支持其他 CSS 属性值（property value），比如 Unicode 范围，或 !important 声明。然而，Sass 不会特殊对待这些属性值，一律视为无引号字符串(unquoted strings)。

后两个小节详细讲解字符串和值列表数据类型，其它类型与 JavaScript 中的用法一致。

=== 字符串 ===

SassScript 支持 CSS 的两种字符串类型：

1、有引号字符串 (quoted strings)，如 "Lucida Grande" 、'http://sass-lang.com'；
2、无引号字符串 (unquoted strings)，如 sans-serifbold。

在编译 CSS 文件时不会改变其类型。只有一种情况例外，使用 #{ }插值语句(interpolation) 时，有引号字符串将被编译为无引号字符串，这样方便了在混合指令(mixin) 中引用选择器名。

```
@mixin firefox-message($selector) {
  body.firefox #{$selector}:before {
    content: "Hi, Firefox users!";
  }
}
@include firefox-message(".header");
```

编译为：

```
body.firefox .header:before {
```

content: "Hi, Firefox users!"; }

需要注意的是：当 deprecated = property syntax 时 （暂时不理解是怎样的情况），所有的字符串都将被编译为无引号字符串，不论是否使用了引号。

=== 值列表 ===

所谓值列表 (lists) 是指 Sass 如何处理 CSS 中：

margin: 10px 15px 0 0

或者：

font-face: Helvetica, Arial, sans-serif

像上面这样通过空格或者逗号分隔的一系列的值。

事实上，独立的值也被视为值列表——只包含一个值的值列表。

Sass 列表函数（Sass list functions）赋予了值列表更多功能（Sass 进级会有讲解）：

1、nth 函数（nth function） 可以直接访问值列表中的某一项；
2、join 函数（join function） 可以将多个值列表连结在一起；
3、append 函数（append function） 可以在值列表中添加值；
4、@each 规则（@each rule）则能够给值列表中的每个项目添加样式。

值列表中可以再包含值列表，比如 1px 2px, 5px 6px 是包含 1px 2px 与 5px 6px 两个值列表的值列表。如果内外两层值列表使用相同的分隔方式，要用圆括号包裹内层，所以也可以写成 (1px 2px) (5px 6px)。当值列表被编译为 CSS 时，Sass 不会添加任何圆括号，因为 CSS 不允许这样做。(1px 2px) (5px 6px)与 1px 2px 5px 6px 在编译后的 CSS 文件中是一样的，但是它们在 Sass 文件中却有不同的意义，前者是包含两个值列表的值列表，而后者是包含四个值的值列表。

可以用 () 表示空的列表，这样不可以直接编译成 CSS，比如编译 font-family: ()时，Sass 将会报错。如果值列表中包含空的值列表或空值，编译时将清除空值，比如 1px 2px () 3px 或 1px 2px null 3px。

第 5 章 Sass 的基本特性-运算

=== 加法 ===

程序中的运算是常见的一件事情，但在 CSS 中能做运算的，到目前为止仅有 calc() 函数可行。但在 Sass 中，运算只是其基本特性之一。在 Sass 中可以做各种数学计算，在接下来的章节中，主要和大家一起探讨有关于 Sass 中的数学运算。

（一）、加法

加法运算是 Sass 中运算中的一种，在变量或属性中都可以做加法运算。如：

```
.box {
  width: 20px + 8in;
}
```

编译出来的 CSS:

```
.box {
  width: 788px;
}
```

但对于携带不同类型的单位时，在 Sass 中计算会报错，如下例所示：

```
.box {
  width: 20px + 1em;
}
```

编译的时候，编译器会报错：“Incompatible units: 'em' and ‘px'.”

=== 减法 ===

Sass 的减法运算和加法运算类似，我们通过一个简单的示例来做阐述：

```
$full-width: 960px;
$sidebar-width: 200px;

.content {
  width: $full-width -  $sidebar-width;
}
```

编译出来的 CSS 如下：

```
.content {
  width: 760px;
```

```
}
```

同样的，运算时碰到不同类型的单位时，编译也会报错，如：

```
$full-width: 960px;

.content {
  width: $full-width -  1em;
}
```

编译的时候，编译器报“Incompatible units: 'em' and ‘px’.”错误。

=== 乘法 ===

Sass 中的乘法运算和前面介绍的加法与减法运算还略有不同。虽然他也能够支持多种单位（比如 em ,px , %），但当一个单位同时声明两个值时会有问题。比如下面的示例：

```
.box {
  width:10px * 2px;
}
```

编译的时候报“20px*px isn't a valid CSS value.”错误信息。

如果进行乘法运算时，两个值单位相同时，只需要为一个数值提供单位即可。上面的示例可以修改成：

```
.box {
  width: 10px * 2;
}
```

编译出来的 CSS:

```
.box {
  width: 20px;
}
```

Sass 的乘法运算和加法、减法运算一样，在运算中有不同类型的单位时，也将会报错。如下面的示例：

```
.box {
  width: 20px * 2em;
}
```

编译时报“40em*px isn't a valid CSS value.”错误信息。

=== 除法 ===

Sass 的乘法运算规则也适用于除法运算。不过除法运算还有一个特殊之处。众所周知“/”符号在 CSS 中已做为一种符号使用。因此在 Sass 中做除法运算时，直接使用“/”符号做为除号时，将不会生效，编译时既得不到我们需要的效果，也不会报错。一起先来看一个简单的示例：

```
.box {
  width: 100px / 2;
}
```

编译出来的 CSS 如下：

```
.box {
  width: 100px / 2;
}
```

这样的结果对于大家来说没有任何意义。要修正这个问题，只需要给运算的外面添加一个小括号( )即可：

```
.box {
  width: (100px / 2);
}
```

编译出来的 CSS 如下：

```
.box {
  width: 50px;
}
```

除了上面情况带有小括号，“/”符号会当作除法运算符之外，如果“/”符号在已有的数学表达式中时，也会被认作除法符号。如下面示例：

```
.box {
  width: 100px / 2 + 2in;
}
```

编译出来的 CSS：

```
.box {
  width: 242px;
```

```
}
```

另外，在 Sass 除法运算中，当用变量进行除法运算时，“/”符号也会自动被识别成除法，如下例所示：

```
$width: 1000px;
$nums: 10;

.item {
  width: $width / 10;
}

.list {
  width: $width / $nums;
}
```

编译出来的 CSS:

```
.item {
  width: 100px;
}

.list {
  width: 100px;
}
```

综合上述，”/ ”符号被当作除法运算符时有以下几种情况：

- 如果数值或它的任意部分是存储在一个变量中或是函数的返回值。
- 如果数值被圆括号包围。
- 如果数值是另一个数学表达式的一部分。

如下所示：

```
//SCSS
p {
  font: 10px/8px;                 // 纯 CSS，不是除法运算
  $width: 1000px;
  width: $width/2;                // 使用了变量，是除法运算
  width: round(1.5)/2;            // 使用了函数，是除法运算
  height: (500px/2);              // 使用了圆括号，是除法运算
  margin-left: 5px + 8px/2px; // 使用了加（+）号，是除法运算
}
```

编译出来的 CSS

```
p {
  font: 10px/8px;
  width: 500px;
  height: 250px;
  margin-left: 9px;
 }
```

Sass 的除法运算还有一个情况。我们先回忆一下，在乘法运算时，如果两个值带有相同单位时，做乘法运算时，出来的结果并不是我们需要的结果。但在除法运算时，如果两个值带有相同的单位值时，除法运算之后会得到一个不带单位的数值。如下所示：

```
.box {
  width: (1000px / 100px);
}
```

编译出来的 CSS 如下：

```
.box {
  width: 10;
}
```

=== 变量计算 ===

在 Sass 中除了可以使用数值进行运算之外，还可以使用变量进行计算，其实在前面章节的示例中也或多或少的向大家展示了。在 Sass 中使用变量进行计算，这使得 Sass 的数学运算功能变得更加实用。一起来看一个简单的示例：

```
$content-width: 720px;
$sidebar-width: 220px;
$gutter: 20px;

.container {
  width: $content-width + $sidebar-width + $gutter;
  margin: 0 auto;
}
```

编译出来的 CSS

```
.container {
  width: 960px;
  margin: 0 auto;
```

```
}
```

=== 数字运算 ===

在 Sass 运算中数字运算是较为常见的，数字运算包括前面介绍的：加法、减法、乘法和除法等运算。而且还可以通过括号来修改他们的运算先后顺序。和我们数学运算是一样的，一起来看个示例。

```
.box {
  width: ((220px + 720px) - 11 * 20 ) / 12 ;
}
```

编译出来的 CSS:

```
.box {
  width: 60px;
}
```

上面这个简单示例是一个典型的计算 Grid 单列列宽的运算。

=== 颜色运算 ===

所有算数运算都支持颜色值，并且是分段运算的。也就是说，红、绿和蓝各颜色分段单独进行运算。如：

```
p {
  color: #010203 + #040506;
}
```

计算公式为 01 + 04 = 05、02 + 05 = 07 和 03 + 06 = 09， 并且被合成为：

如此编译出来的 CSS 为：

```
p {
  color: #050709;
}
```

算数运算也能将数字和颜色值 一起运算，同样也是分段运算的。如：

```
p {
  color: #010203 * 2;
}
```

计算公式为 01 * 2 = 02、02 * 2 = 04 和 03 * 2 = 06， 并且被合成为：

```
p {
  color: #020406;
}
```

=== 字符运算 ===

在 Sass 中可以通过加法符号“+”来对字符串进行连接。例如：

```
$content: "Hello" + "" + "Sass!";
.box:before {
  content: " #{$content} ";
}
```

编译出来的 CSS：

```
.box:before {
  content: " Hello Sass! ";
}
```

除了在变量中做字符连接运算之外，还可以直接通过 +，把字符连接在一起：

```
div {
  cursor: e + -resize;
}
```

编译出来的 CSS:

```
div {
  cursor: e-resize;
}
```

注意，如果有引号的字符串被添加了一个没有引号的字符串 （也就是，带引号的字符串在 + 符号左侧)， 结果会是一个有引号的字符串。 同样的，如果一个没有引号的字符串被添加了一个有引号的字符串 （没有引号的字符串在 + 符号左侧)， 结果将是一个没有引号的字符串。 例如：

```
p:before {
  content: "Foo " + Bar;
  font-family: sans- + "serif";
}
```

编译出来的 CSS：

```
p:before {
  content: "Foo Bar";
  font-family: sans-serif;
  }
```

小伙伴们，到此为止《sass 基础入门（基础篇）》的课程已经全部讲解完了，后面还有《sass 基础入门（进阶篇）》即将上线，欢迎小伙伴们到时学习。

同时小伙伴们如果想看一下 sass 的案例课程可观看《Sass 和 Compass 必备技能之 Sass 篇》课程。