# SASS

> 在以前的时候，当我们去书写CSS样式的时候，存在一些问题
>
> 1. 对于有规律的样式，我们无法去通过编程去实殃
> 2. 对于CSS当中，某些共有的值，但是这些值后期又会发生变化的时候，我们无法通过变量去完成（`var`虽然说可以使用变量，它是它只是支持chrome浏览器，IE不支持）
> 3. 当我们在进行CSS的选择器书写的时候，如果要写后代选择器，子代选择器则会很麻烦

上面的这些问题在后面都已经实现掉了，后面出现了一个概念叫**可编程的CSS**

在可编程的CSS里面，也经历了几代的发展

1. 第一代叫`less`，它是一个非常简便的语法，前期有很多CSS的框架都是通过它来实现的，并且在浏览器里面可以直接书写`less`，只需要导入一个js插件就可以

   ```
   <style type="text/less"></style>
   ```

   这么写前期很好，但是后期就觉得有问题了，因为这个less是实时转换的，它需要经过JS去在线转换成CSS，这个时候如果网页比较大，就会感觉到页面很卡

   后来又提供了一种新的方法，就是提前就把`less`编译成`css`，这样也解决这个问题了，目前`bootstrap3`采用的就是这种方法

2. 第二代可编程CSS在第一代的上面又扩展了很多功能，主要的就是`sass、stylus`

---

# 什么是SASS

sass是一种可编程的CSS语言，其基础的核心有 4 个

1. 变量
2. 嵌套
3. 混合
4. 继承

围绕上面的4个点可以开展SASS的基础语法学习

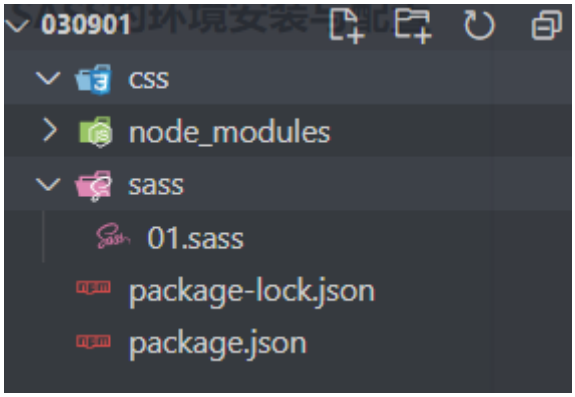
首先在学习之前，我们要安装sass的编译环境

# SASS的环境安装与配置

首先我们先通过npm去安装编程sass的包，这个包叫`node-sass`

```
$ npm install node-sass --save-dev
```

当安装完成了这个包以后，我们就要先配置一下这个包然后再使用




```
.login
    width: 100px
    height: 200px
```

将上面的`01.sass`编译成CSS文件，然后在`package.json`的`script`里面配置命令

```
"build":"node-sass ./sass/01.sass -o ./css"
```

在我们初次试验的过程当中，体验并不是很好，因为`SASS`是一个严格的语法标准，空格不能多也不能少，分号不能多也不能省，所以写起来的时候约束很大，最初并不是很多程序员喜欢用这个东西

正是因为存在这样的问题，所以后期的SASS开发人员就在这个标准下面出了一个新的语法，这个语法是完全兼容CSS的写法叫`SCSS`

**SASS是标准，SCSS是语法**

```
.register{
    width:100px;
    height:200px;
    border:1px solid black;
}
```

当我们了解这个语法以后，我们还要需要配置启动命令，但是我们不能像上面那样，每次都手动去执行一遍编译命令，需要让它自动编译

```
"dev":"node-sass --watch ./scss --output ./css"
```

上面的命令中`--watch`代表监控某个文件夹，简写为`-w`，`--output`代表输出到某一个文件夹，可以简写为`-o`，这个命令还可以简化如下面的方式

```
"dev":"node-sass -w ./scss -o ./css"
```

## 变量

变量是可编程CSS里面最重要的一个点，在SASS当中也是支持的，它使用`$变量名`来定义变量

```
$w:100px;
$primaryColor:#f8f8f8;
.div1{
   width: $w;
   background: $primaryColor;
}
```

生成好的CSS如下

```
.div1 {
  width: 100px;
  background: #f8f8f8;
}
```

sass的变量是可以发生覆盖的行为的，其实也相当于重新赋值

```
// 变量
$a:100px;
.div1{
    height: $a;
}
$a:200px;
.div2{
    width: $a;
}
```

生成的CSS

```
.div1 {
  height: 100px;
}
.div2 {
  width: 200px;
}
```

变量在赋值的时候还可以使用**默认值**，如下所示

```
$a:100px;
$a:200px !default;      //这里相当于定义变量$a，如果a之前有值就使用之前的值，如果没有值则使用默认值
.div1{
    width: $a;
}
```

最终生成的css如下

```
.div1 {
  width: 100px;
}
```

## 嵌套

在我们书写CSS语法选择器的时候，经常会有子级选择器，后代选择器，这个时候在SASS里面非常简单

```
// 嵌套
.div1{
    width: 200px;
    .left{
        background-color: red;
    }
    >.right{
```

```
        height: 50px;
        .right-text{
            font-weight: bold;
            h2{
                text-align: center;
            }
        }
    }
}
```

生成的CSS如下

```
.div1 {
    width: 200px;
}
.div1 .left {
    background-color: red;
}
.div1>.right {
    height: 50px;
}
.div1>.right .right-text {
    font-weight: bold;
}
.div1>.right .right-text h2 {
    text-align: center;
}
```

在进行SCSS嵌套的时候，如果我要指向父级选择器，可以使用`&`来进行

```
// 嵌套
.div1 {
    >.right {
        &+li{
            border-top-color: red;
        }

    }
}
```

生成的CSS如下

```
.div1>.right+li {
  border-top-color: red;
}
```

**注意：变量如果与嵌套结合的时候，变量有区域性了**

```
// 变量的区域性
$primaryColor:tomato;
.div1{
    background-color: $primaryColor;
    $w:100px;           //这里定义了一个局部变量
    .a{
        border-top: $primaryColor;
```

```
        width: $w;
    }
}
.div2{
    color: $primaryColor;
    height: $w;              //这里使用局部变量就会报错
}
```

### `@at-root`的使用

我们的选择器是可以嵌套的，同时我们的变量又是有区域性的，如果发生了一种情况，我想使用内部的变量，但是选择器是要生成全局的，怎么办呢?

```
//嵌套
.a {
    line-height: 50px;
    .b {
        color: red;
        .c {
            width: 100px;
            $w: 250px;
            //跳出所有的嵌套，到最外边
            @at-root {
                .d {
                    width: $w;
                }
            }
        }
    }
}
```

最终生成的css如下

```
.a {
  line-height: 50px;
}
.a .b {
  color: red;
}
.a .b .c {
  width: 100px;
}
/*这里的d生成的时候就是全局的*/
.d {
  width: 250px;
}
```

当我们使用了`@at-root`的时候，这个时候生成的选择器会到全局，但是又出了另一个问题，我们如果想使用`@at-root`跳出`@media`的媒体查询的时候，这个时候怎么办呢

### @at-root跳出媒体查询

```
@media only screen and (max-width:768px) {
    .a {
        line-height: 40px;
        .b {
```

```
            color: red;
            .c {
                width: 100px;
                $a: 250px;
                @at-root(without: media) {
                    .d{
                        width: $a;
                    }
                }
            }
        }
    }
}
```

生成的CSS

```
@media only screen and (max-width: 768px) {
  .a {
    line-height: 40px;
  }

  .a .b {
    color: red;
  }

  .a .b .c {
    width: 100px;
  }
}
/*这个时候生成的css就跳出了媒体查询*/
.d {
  width: 250px;
}
```

## 混合

混合也叫`mixin`，它最主要的作用就是用于将一些代码混合到其它的一些代码当中，后期这么做可以将公共的代码提取出来以后，再进行混合，这样操作起来非常方便

```
// 混合
// 定义一个混合器，把公共的代码提取出来
@mixin flexbox {
    display: flex;
    flex-direction: row;
    justify-content: center;
    align-items: center;
}

.login {
    width: 300px;
    height: 300px;
    border: 1px solid red;
    @include flexbox();
}

.reg {
```

```
    width: 400px;
    height: 200px;
    border: 2px dashed blue;
    @include flexbox();
}
```

生成的css

```
.login {
  width: 300px;
  height: 300px;
  border: 1px solid red;
  display: flex;
  flex-direction: row;
  justify-content: center;
  align-items: center;
}

.reg {
  width: 400px;
  height: 200px;
  border: 2px dashed blue;
  display: flex;
  flex-direction: row;
  justify-content: center;
  align-items: center;
}
```

定义一个混合器我们使用`@mixin 混合器名称`，在调用混合器的时候，我们使用`@include 混合器名称`，如上所示，我们定义了一个`flexbox`的混合器，同时在后面调用了它

我们又发现一个问题，在刚刚的弹性盒子的混合器里面，有些值是会发生变化的，如`flex-direction/justify-content/align-itmes`，所以能否对在混合器里面设置变量呢

**混合器的参数**

```
// 混合器的变量
@mixin flexbox($dir) {
    display: flex;
    flex-direction: $dir;
    justify-content: center;
    align-items: center;
}
.a{
    @include flexbox(row);
}
.b{
    @include flexbox(column);
}
```

上面的代码就是在定义混合器的时候定义了参数，然后在调用混合的时候传入了具体的值（这一个与方法的概念非常像）

**混合器参数的默认值**

混合器在定义参数的时候是可以设置为默认值的

```
// 混合器参数的默认值
@mixin flexbox($dir:row,$justify:flex-start,$align:stretch) {
    display: flex;
    flex-direction: $dir;
    justify-content: $justify;
    align-items: $align;
}
.a{
    @include flexbox();
}
.b{
    @include flexbox(column);
}
.c{
    @include flexbox(row,center,flex-end);
}
```

生成的css

```
.a {
  display: flex;
  flex-direction: row;
  justify-content: flex-start;
  align-items: stretch;
}
.b {
  display: flex;
  flex-direction: column;
  justify-content: flex-start;
  align-items: stretch;
}
.c {
  display: flex;
  flex-direction: row;
  justify-content: center;
  align-items: flex-end;
}
```

**混合器里面的占位符**

```
// 混合器当中的占位符
@mixin flexbox($dir:row, $justify:flex-start, $align:stretch) {
    display: flex;
    flex-direction: $dir;
    justify-content: $justify;
    align-items: $align;
    // 下面定义了一个占位符
    @content;
}
```

在调用混合器的时候，可以占位，也可以不占位

```
.a{
    @include flexbox();
}
.b{
    @include flexbox(){
        //向占位符里面添加内容
        text-align: center;
        line-height: 50px;
    };
    font-size: 36px;
}
```

最终生成的css代码

```
.a {
  display: flex;
  flex-direction: row;
  justify-content: flex-start;
  align-items: stretch;
}
.b {
  display: flex;
  flex-direction: row;
  justify-content: flex-start;
  align-items: stretch;
  text-align: center;
  line-height: 50px;
  font-size: 36px;
}
```

## 继承

在我们之前学习CSS的时候，CSS是有继承性，同时JS里面也有继承，继承指的是子级继承了父级的特点。同理在SASS里面，它也有继承的概念，并且可以通过具体的语法实现

```
// 继承
.page-header{
    font-size: 32px;
    text-decoration: underline black wavy;
    color: tomato;
    font-weight: bold;
}

.login-page{
    width: 100px;
    @extend .page-header;
}
```

最终生成的css如下

```
.page-header, .login-page {
  font-size: 32px;
  text-decoration: underline black wavy;
  color: tomato;
  font-weight: bold;
}

.login-page {
  width: 100px;
}
```

通过上面的代码，我们发现sass里面的继承其实就是帮我们生成了分组选择器，所以继承的时候是不需要限定选择器的类型（因为CSS里面的分组选择器左右是没有类型限定的，所以我们也不需要限定）

```
#a {
    width: 200px;
}
#b {
    @extend #a;
}
p{
    text-align: center;
}
h2{
    @extend p;
}
.c{
    @extend #b;
}
```

---

## 运算符

在scss里面它是支持最基本的四则运算的

```
$a:10;
$b:20;

.div1 {
    width: $a + $b + px;
    height: $b - $a + px;
    line-height: $b / $a;
    border-radius: $a * $b + px;
}
.div2{
    width: 100 + 2 * 20px;
    height: 20px + 20px;
}
```

在div1与div2进行对比的时候，我们发现px既可以使用`+`来连接，后面也可以省略`+`均可，这是因为`scss`与`css`当中`px`都是做为像素单位存在

同时要注意，在进行运算符操作的时候，我们的符号左右一定要保留一个空格位

## 函数

这里的函数与之前的函数的概念是一样的，都是为了实现某一个功能的封装

```
// 函数 主要就是用于实现某些功能的封装
$baseFontSize:16px !default;
@function pxToRem($currentSize){
    @return $currentSize /  $baseFontSize + rem;
}
.div1{
    width: pxToRem(32px);
}
```

上面其实就是一个最简单的函数，定义一个函数使用的是`@function`，可以将px转换成rem,最终生成的css如下

```
.div1 {
  width: 2rem;
}
```

在sass的函数里面，是可以通过命令`@return`来返回一个值的

## 条件判断

在SASS里面，条件判断使用的是`@if`这一个命令，语法格式如下

```
$type:2;
.login{
    // 如果type为1，则color为红色，如果type为2则color为蓝色，否则为黑色
    @if $type==1{
        color: red;
    }
    @else if $type==2{
        color: blue;
    }
    @else{
        color: black;
    }
}

$w:100;
.reg{
    @if $w > 80 {
        display: flex;
    }
    @else{
        display: inline-block;
    }
}

$h:90;
.div1{
    //如果判断不等于100
    @if $h != 100 {
        line-height: 50px;
    }
    @else{
        line-height: 60px;
```

```
    }
}
```

这个条件判断的语法与之前的JS里面的语法大同小异

## 条件表达式

当条件比较简单的时候我们的条件判断语句还可以写成条件表达式

```
// 条件表达式
$type:1;

.login{
    // $type为1则是红色，否则就是蓝色
    color: if($type==1,red,blue);
}
```

上面的条件表达式本质上面是一个函数，如下，我们可以自己写一个函数

```
$type:1;
@function bgg($condition, $trueValue, $falseValue) {
    @if $condition==true {
        @return $trueValue;
    }

    @else {
        @return $falseValue;
    }
}
.reg {
    color: bgg($type==1, red, blue);
}
```

## 循环

在SASS里面，如果对于一些有规律的样式，我们可以使用for循环去生成，SASS里面是有`@for`这个命令的，它的语法格式有两种情况

1. `@for $var from start to end`，这一种情况代表从start 到 end结束，但是**不包含**end
2. `@for $var from start through end`，这一种情也是从start到end结束，但是这个**包含**end

```
// 写在要写一个.item1到.item4,分别设置宽度为200px,400px,600px,800px
//for(var i=1;i<4;i++)
@for $i from 1 to 4 {
    .item#{$i}{
        width: $i * 200px;
    }
}

// for(var i=1;i<=4;i++)
@for $i from 1 through 4 {
    .a#{$i}{
        width: $i * 200px;
    }
}
```

最终生成的css如下

```
.item1 {
  width: 200px;
}
.item2 {
  width: 400px;
}
.item3 {
  width: 600px;
}
.a1 {
  width: 200px;
}
.a2 {
  width: 400px;
}
.a3 {
  width: 600px;
}
.a4 {
  width: 800px;
}
```

## 值列表

在sass里在，也是有数组的，只是换了个说法，叫值列表，定义值列表的方式如下

```
$colorList:red,blue,yellow,gray,deeppink;
```

在定义数组的时候，只要将值使用逗号隔开就可以了

1. 获取数组的长度，我们使用内置的函数`length()`去完成

   ```
   length($colorList);
   ```

2. sass里面的数组的索引是从1开始的，并且取数组的值也是通过内置函数`nth()`来完成

   ```
   background-color:nth($colorList,1);
   ```

3. sass可以结合之前的`@for`的指令来对数组进行遍历

   ```
   // 就想根据上面的颜色 生产出不同背景颜色的class样式，以bg-开头
   @for $i from 1 through length($colorList) {
       .bg-#{nth($colorList,$i)}{
           color: nth($colorList,$i);
       }
   }
   ```

4. 后期还可以进行`@each`的遍历

## 键值对列表

键值对可以理解为之前S里面的对象，也可以理解为ES6里面的`Map`集合，原理与操作都是相同的【键不同相同，通过值来取值】

**定义一个键值对**

```
$colorMap:(r:red, b:blue, y:yellow, g:gray, d:deeppink);
```

**键值对取值**

定义好键值对以后，如果要通过键去取值，则要通过`map-get`的方法去完成

```
.login{
    color: map-get($colorMap,r);
}
.reg{
    background-color: map-get($colorMap, d);
}
```

**键值对的遍历，可以参考@each章节**

## each遍历

each的遍历其实是为了简化我们之前的`@for`的遍历，它的过程与之前S里面的遍历过程很像，取到的是值

**下面的代码就是使用了each去遍历一个数组，可以与之前的`@for`指令去对比一下**

```
$colorList:red,blue,yellow,gray,deeppink;
//之前我们是使用了 @for来进行遍历，但是现在，我们还有一个新的指令@each也可以遍历
@each $item in $colorList {
    .bg-#{$item}{
        background-color: $item;
    }
}
```

**下面是键值对的遍历过程**

```
// 键值对的应用
//定义了一个颜色的键值对
$colorList:(default:#FFFFFF, primary:#418BCA, success:#5BB85D, info:#57BFDC,
warning:#EFAD4D, danger:#D9544F);

//$k代表物是键，$v代表的是值
@each $k,$v in $colorList {
    .btn-#{$k}{
        background-color: $v;
    }
    .bg-#{$k}{
        background-color: $v;
    }
    .text-#{$k}{
        color: $v;
    }
}
```