Contents

[npm 4](#_Toc32786762)

[npm淘宝镜像 4](#_Toc32786763)

[git 4](#_Toc32786764)

[git bash 4](#_Toc32786765)

[从github获取 4](#_Toc32786766)

[更新至github 4](#_Toc32786767)

[1 Vue.js 4](#_Toc32786768)

[1.1 Install 4](#_Toc32786769)

[v-bind 5](#_Toc32786770)

[动态绑定 5](#_Toc32786771)

[Style 5](#_Toc32786772)

[Class 5](#_Toc32786773)

[Vuex 5](#_Toc32786774)

[State 5](#_Toc32786775)

[Mutation 6](#_Toc32786776)

[computed 8](#_Toc32786777)

[Sass 8](#_Toc32786778)

[Sass 变量 8](#_Toc32786779)

[Sass 作用域 9](#_Toc32786780)

[!global 10](#_Toc32786781)

[Sass 嵌套规则与属性 10](#_Toc32786782)

[Sass 嵌套属性 12](#_Toc32786783)

[Sass @import 12](#_Toc32786784)

[Sass Partials 15](#_Toc32786785)

[Sass @mixin 与 @include 15](#_Toc32786786)

[定义一个混入 16](#_Toc32786787)

[使用混入 16](#_Toc32786788)

[Sass @include 混入语法： 16](#_Toc32786789)

[向混入传递变量 17](#_Toc32786790)

[可变参数 20](#_Toc32786791)

[浏览器前缀使用混入 21](#_Toc32786792)

[Sass @extend 与 继承 21](#_Toc32786793)

[Sass String(字符串) 函数 23](#_Toc32786794)

[Sass 数字函数 25](#_Toc32786795)

[Sass 列表(List)函数 28](#_Toc32786796)

[Sass Map(映射)函数 30](#_Toc32786797)

[Sass 选择器函数 32](#_Toc32786798)

[Sass Introspection 函数 34](#_Toc32786799)

[Sass 颜色函数 36](#_Toc32786800)

[Sass 颜色设置 36](#_Toc32786801)

[Sass 颜色获取 38](#_Toc32786802)

[Sass 颜色操作 40](#_Toc32786803)

[CSS 43](#_Toc32786804)

[选择器 43](#_Toc32786805)

[Id选择器 43](#_Toc32786806)

[派生选择器 43](#_Toc32786807)

[类选择器 43](#_Toc32786808)

[属性选择器 44](#_Toc32786809)

[属性 44](#_Toc32786810)

[Padding 44](#_Toc32786811)

[Margin 44](#_Toc32786812)

[Box-shadow 45](#_Toc32786813)

[background 45](#_Toc32786814)

[border 45](#_Toc32786815)

[text-align 45](#_Toc32786816)

[Flex 45](#_Toc32786817)

[主轴对齐justify-content 45](#_Toc32786818)

[交叉轴对齐 align-items 45](#_Toc32786819)

[多轴对齐 align-content 45](#_Toc32786820)

[flex-direction: row | row-reverse | column | column-reverse; 46](#_Toc32786821)

[flex-flow: <flex-direction> || <flex-wrap>; 47](#_Toc32786822)

[传参 47](#_Toc32786823)

[IconFont 47](#_Toc32786824)

[pixel 47](#_Toc32786825)

[JS 47](#_Toc32786826)

[Math 47](#_Toc32786827)

[Erros 47](#_Toc32786828)

[Vue不是一个内部命令也不是一个外部 47](#_Toc32786829)

[skill 48](#_Toc32786830)

[为数组中的不同的项分配class 48](#_Toc32786831)

[Display：flex与v-show不兼容 48](#_Toc32786832)

[如何设置微信开发者工具里的手机模拟器 48](#_Toc32786833)

[flex中独占一行/列 48](#_Toc32786834)

[子元素相对父元素定位 48](#_Toc32786835)

[v-if和v-show的区别 48](#_Toc32786836)

[1.共同点 49](#_Toc32786837)

[2.区别 49](#_Toc32786838)

[uni-segmented-control 51](#_Toc32786839)

<https://developer.android.com/guide>

https://baijiahao.baidu.com/s?id=1581222739674858335&wfr=spider&for=pc

# npm

## npm淘宝镜像

npm install -g cnpm --registry=https://registry.npm.taobao.org

npm init

npm i vant-weapp -S --production

# git

## git bash

cd c:\android\fittingChart

git init

git remote add origin https://github.com/zxswzj/jellyClassTable

# 从github获取

git stash

git reset --hard FETCH\_HEAD

git pull origin master -f

## 更新至github

Git add .

Git commit

Git push origin master

# Vue.js

## Install

$ cnpm install vue

$ cnpm install --global vue-cli

$ vue init webpack my-project

$ cd my-project

$ cnpm install

$ cnpm run dev

## v-bind

v-bind:value="checkedGoods"

data() {

return {

checkedGoods: ['1', '2', '3']

}

}

## 动态绑定

### Style

:style="[{ 'margin-top': item2.margintop + 'px' }, { height: item2.height + 'px' }]"

### Class

:class="[{ popColorBlockSelected: i == blockToken }]"

## Vuex

### State

当一个组件需要获取多个状态时候，将这些状态都声明为计算属性会有些重复和冗余。为了解决这个问题，我们可以使用 mapState 辅助函数帮助我们生成计算属性，让你少按几次键：

// 在单独构建的版本中辅助函数为 Vuex.mapState

import { mapState } from 'vuex'

export default {

// ...

computed: mapState({

// 箭头函数可使代码更简练

count: state => state.count,

// 传字符串参数 'count' 等同于 `state => state.count`

countAlias: 'count',

// 为了能够使用 `this` 获取局部状态，必须使用常规函数

countPlusLocalState (state) {

return state.count + this.localCount

}

})

}

当映射的计算属性的名称与 state 的子节点名称相同时，我们也可以给 mapState 传一个字符串数组。

computed: mapState([

// 映射 this.count 为 store.state.count

'count'

])

### Mutation

更改 Vuex 的 store 中的状态的唯一方法是提交 mutation。Vuex 中的 mutation 非常类似于事件：每个 mutation 都有一个字符串的 事件类型 **(type)** 和 一个 **回调函数 (handler)**。这个回调函数就是我们实际进行状态更改的地方，并且它会接受 state 作为第一个参数：

const store = new Vuex.Store({

state: {

count: 1

},

mutations: {

increment (state) {

// 变更状态

state.count++

}

}

})

你不能直接调用一个 mutation handler。这个选项更像是事件注册：“当触发一个类型为 increment 的 mutation 时，调用此函数。”要唤醒一个 mutation handler，你需要以相应的 type 调用 **store.commit** 方法：

store.commit('increment')

#### 提交载荷（Payload）

store.commit('increment', 10)

store.commit('increment', {

amount: 10

})

store.commit({

type: 'increment',

amount: 10

})

#### Mutation 需遵守 Vue 的响应规则

既然 Vuex 的 store 中的状态是响应式的，那么当我们变更状态时，监视状态的 Vue 组件也会自动更新。这也意味着 Vuex 中的 mutation 也需要与使用 Vue 一样遵守一些注意事项：

1. 最好提前在你的 store 中初始化好所有所需属性。
2. 当需要在对象上添加新属性时，你应该

* 使用 Vue.set(obj, 'newProp', 123), 或者
* 以新对象替换老对象。例如，利用[对象展开运算符](https://github.com/tc39/proposal-object-rest-spread)我们可以这样写：

state.obj = { ...state.obj, newProp: 123 }

#### 使用常量替代 Mutation 事件类型

#### Mutation 必须是同步函数

现在想象，我们正在 debug 一个 app 并且观察 devtool 中的 mutation 日志。每一条 mutation 被记录，devtools 都需要捕捉到前一状态和后一状态的快照。然而，在上面的例子中 mutation 中的异步函数中的回调让这不可能完成：因为当 mutation 触发的时候，回调函数还没有被调用，devtools 不知道什么时候回调函数实际上被调用——实质上任何在回调函数中进行的状态的改变都是不可追踪的。

#### 在组件中提交 Mutation

你可以在组件中使用 this.$store.commit('xxx') 提交 mutation，或者使用 mapMutations 辅助函数将组件中的 methods 映射为 store.commit 调用（需要在根节点注入 store）。

import { mapMutations } from 'vuex'

export default {

// ...

methods: {

...mapMutations([

'increment', // 将 `this.increment()` 映射为 `this.$store.commit('increment')`

// `mapMutations` 也支持载荷：

'incrementBy' // 将 `this.incrementBy(amount)` 映射为 `this.$store.commit('incrementBy', amount)`

]),

...mapMutations({

add: 'increment' // 将 `this.add()` 映射为 `this.$store.commit('increment')`

})

}

}

## computed

# Sass

## Sass 变量

Sass 变量可以存储以下信息：

* 字符串
* 数字
* 颜色值
* 布尔值
* 列表
* null 值

Sass 变量使用 **$** 符号：

$variablename: value;



## Sass 作用域

Sass 变量的作用域只能在当前的层级上有效果，如下所示 h1 的样式为它内部定义的 green，p 标签则是为 red。



## !global



**注意：**所有的全局变量我们一般定义在同一个文件，如：**\_globals.scss**，然后我们使用 [**@include**](https://www.runoob.com/sass/sass-mixin-include.html) 来包含该文件。

## Sass 嵌套规则与属性

Sass 嵌套 CSS 选择器类似于 HTML 的嵌套规则。



## Sass 嵌套属性

很多 CSS 属性都有同样的前缀，例如：font-family, font-size 和 font-weight ， text-align, text-transform 和 text-overflow。

在 Sass 中，我们可以使用嵌套属性来编写它们：



## Sass @import

Sass 可以让我们减少可以帮助我们减少 CSS 重复的代码，节省开发时间。

我们可以安装不同的属性来创建一些代码文件，如：变量定义的文件、颜色相关的文件、字体相关的文件等。

类似 CSS，Sass 支持 **@import** 指令。

@import 指令可以让我们导入其他文件等内容。

CSS @import 指令在每次调用时，都会创建一个额外的 HTTP 请求。但，Sass @import 指令将文件包含在 CSS 中，不需要额外的 HTTP 请求。

Sass @import 指令语法如下：

@import filename;

**注意：**包含文件时不需要指定文件后缀，Sass 会自动添加后缀 .scss。

此外，你也可以导入 CSS 文件。



## Sass Partials

如果你不希望将一个 Sass 的代码文件编译到一个 CSS 文件，你可以在文件名的开头添加一个下划线。这将告诉 Sass 不要将其编译到 CSS 文件。

但是，在导入语句中我们不需要添加下划线。

Sass Partials 语法格式：

\_filename;



**注意：**请不要将带下划线与不带下划线的同名文件放置在同一个目录下，比如，\_colors.scss 和 colors.scss 不能同时存在于同一个目录下，否则带下划线的文件将会被忽略。

## Sass @mixin 与 @include

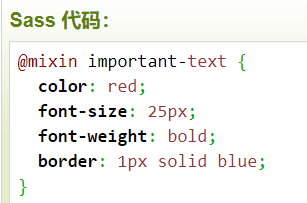
@mixin 指令允许我们定义一个可以在整个样式表中重复使用的样式。

@include 指令可以将混入（mixin）引入到文档中。

### 定义一个混入

混入(mixin)通过 @mixin 指令来定义。 @mixin name { property: value; property: value; ... }

以下实例创建一个名为 "important-text" 的混入：



**注意：**Sass 的连接符号 - 与下划线符号 \_ 是相同的，也就是 @mixin important-text { } 与 @mixin important\_text { } 是一样的混入。

### 使用混入

@include 指令可用于包含一混入：

## Sass @include 混入语法：

selector {  
  @include mixin-name;  
}



## 向混入传递变量

混入可以接收参数。

我们可以向混入传递变量。

定义可以接收参数的混入：





### 可变参数

有时，不能确定一个混入（mixin）或者一个函数（function）使用多少个参数，这时我们就可以使用 **...** 来设置可变参数。

例如，用于创建盒子阴影（box-shadow）的一个混入（mixin）可以采取任何数量的 box-shadow 作为参数。



## 浏览器前缀使用混入



## Sass @extend 与 继承

@extend 指令告诉 Sass 一个选择器的样式从另一选择器继承。

如果一个样式与另外一个样式几乎相同，只有少量的区别，则使用 @extend 就显得很有用。

以下 Sass 实例中，我们创建了一个基本的按钮样式 .button-basic，接着我们定义了两个按钮样式 .button-report 与 .button-submit，它们都继承了 .button-basic ，它们主要区别在于背景颜色与字体颜色，其他的样式都是一样的。



将以上代码转换为 CSS 代码，如下所示：



使用 @extend 后，我们在 HTML 按钮标签中就不需要指定多个类 class="button-basic button-report" ，只需要设置 class="button-report" 类就好了。

@extend 很好的体现了代码的复用。

## Sass String(字符串) 函数

|  |  |
| --- | --- |
| **函数** | **描述 & 实例** |
| quote(*string*) | 给字符串添加引号。  **实例:** quote(runoob) 结果: "runoob" |
| str-index(*string*,*substring*) | 返回 substring 子字符串第一次在 string 中出现的位置。如果没有匹配到子字符串，则返回 null。  str-index(abcd, a) => 1  str-index(abcd, ab) => 1  str-index(abcd, X) => null  str-index(abcd, c) => 3 |
| str-insert(*string*, *insert*, *index*) | 在字符串 string 中 index 位置插入 insert。  **实例:** str-insert("Hello world!", " runoob", 6) 结果: "Hello runoob world!" |
| str-length(*string*) | 返回字符串的长度。  **实例:** str-length("runoob") 结果: 6 |
| str-slice(*string*, *start*, *end*) | 从 string 中截取子字符串，通过 start-at 和 end-at 设置始末位置，未指定结束索引值则默认截取到字符串末尾。  str-slice("abcd", 2, 3) => "bc"  str-slice("abcd", 2) => "bcd"  str-slice("abcd", -3, -2) => "bc"  str-slice("abcd", 2, -2) => "bc" |
| to-lower-case(*string*) | 将字符串转成小写  **实例:** to-lower-case("RUNOOB") 结果: "runoob" |
| to-upper-case(*string*) | 将字符串转成大写  **实例:** to-upper-case("runoob") 结果: "RUNOOB" |
| unique-id() | 返回一个无引号的随机字符串作为 id。不过也只能保证在单次的 Sass 编译中确保这个 id 的唯一性。  **实例:** unique-id() Result: uad053b1c |
| unquote(*string*) | 移除字符串的引号  **实例:** unquote("runoob") 结果: runoob |

## Sass 数字函数

|  |  |
| --- | --- |
| **函数** | **描述 & 实例** |
| abs(*number*) | 返回一个数值的绝对值。  **实例:** abs(15) 结果: 15 abs(-15) 结果: 15 |
| ceil(*number*) | 向上取整  **实例:** ceil(15.20) 结果: 16 |
| comparable(*num1*, *num2*) | 返回一个布尔值，判断 *num1* 与 *num2* 是否可以进行比较  **实例:** comparable(15px, 10px) 结果: true comparable(20mm, 1cm) 结果: true comparable(35px, 2em) 结果: false |
| floor(*number*) | 向下取整  **实例:** floor(15.80) 结果: 15 |
| max(*number...*) | 返回最大值  **实例:** max(5, 7, 9, 0, -3, -7) 结果: 9 |
| min(*number...*) | 返回最小值  **实例:** min(5, 7, 9, 0, -3, -7) 结果: -3 |
| percentage(*number*) | 将数字转化为百分比的表达形式。  **实例:** percentage(1.2) 结果: 120 |
| random() | 返回 0-1 区间内的小数，  **实例:** random() 结果: 0.45673 |
| random(*number*) | 返回 1 至 number 之间的整数，包括 1 和 limit。  **实例:** random(6) 结果: 4 |
| round(*number*) | 返回最接近该数的一个整数，四舍五入。  **实例:** round(15.20) 结果: 15 round(15.80) 结果: 16 |

## Sass 列表(List)函数

|  |  |
| --- | --- |
| append(*list*, *value*, [*separator*]) | 将单个值 *value* 添加到列表尾部。*separator* 是分隔符，默认会自动侦测，或者指定为逗号或空格。  **实例:** append((a b c), d) 结果: a b c d append((a b c), (d), comma) 结果: a, b, c, d |
| index(*list*, *value*) | 返回元素 *value* 在列表中的索引位置。  **实例:** index(a b c, b) 结果: 2 index(a b c, f) 结果: null |
| is-bracketed(*list*) | 判断列表中是否有中括号  **实例:** is-bracketed([a b c]) 结果: true is-bracketed(a b c) 结果: false |
| join(*list1*, *list2*, [*separator, bracketed*]) | 合并两列表，将列表 *list2* 添加到列表 *list1* 的末尾。*separator* 是分隔符，默认会自动侦测，或者指定为逗号或空格。 *bracketed* 默认会自动侦测是否有中括号，可以设置为 true 或 false。  **实例:** join(a b c, d e f) 结果: a b c d e f join((a b c), (d e f), comma) 结果: a, b, c, d, e, f join(a b c, d e f, $bracketed: true) 结果: [a b c d e f] |
| length(*list*) | 返回列表的长度  **实例:** length(a b c) 结果: 3 |
| list-separator(*list*) | 返回一列表的分隔符类型。可以是空格或逗号。  **实例:** list-separator(a b c) 结果: "space" list-separator(a, b, c) 结果: "comma" |
| nth(*list*, *n*) | 获取第 *n* 项的值。  **实例:** nth(a b c, 3) 结果: c |
| set-nth(*list*, *n*, *value*) | 设置列表第 *n* 项的值为 *value*。  **实例:** set-nth(a b c, 2, x) 结果: a x c |
| zip(*lists*) | 将多个列表按照以相同索引值为一组，重新组成一个新的多维度列表。  **实例:** zip(1px 2px 3px, solid dashed dotted, red green blue) 结果: 1px solid red, 2px dashed green, 3px dotted blue |

## Sass Map(映射)函数

|  |  |
| --- | --- |
| **函数** | **描述 & 实例** |
| map-get(*map*, *key*) | 返回 Map 中 *key* 所对应的 value(值)。如没有对应的 key，则返回 null 值。  **实例:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) map-get($font-sizes, "small") 结果: 12px |
| map-has-key(*map*, *key*) | 判断 *map* 是否有对应的 *key*，存在返回 true，否则返回 false。  **实例:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) map-has-key($font-sizes, "big") 结果: false |
| map-keys(*map*) | 返回 *map* 中所有的 key 组成的队列。  **实例:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) map-keys($font-sizes) 结果: "small", "normal, "large" |
| map-merge(*map1*, *map2*) | 合并两个 map 形成一个新的 map 类型，即将 *map2* 添加到 *map1*的尾部  **实例:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) $font-sizes2: ("x-large": 30px, "xx-large": 36px) map-merge($font-sizes, $font-sizes2) 结果: "small": 12px, "normal": 18px, "large": 24px, "x-large": 30px, "xx-large": 36px |
| map-remove(*map*, *keys...*) | 移除 *map* 中的 keys，多个 key 使用逗号隔开。  **实例:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) map-remove($font-sizes, "small") 结果: ("normal": 18px, "large": 24px) map-remove($font-sizes, "small", "large") 结果: ("normal": 18px) |
| map-values(*map*) | 返回 *map* 中所有的 value 并生成一个队列。  **实例:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) map-values($font-sizes) 结果: 12px, 18px, 24px |

## Sass 选择器函数

|  |  |
| --- | --- |
| **函数** | **描述 & 实例** |
| is-superselector(*super*, *sub*) | 比较两个选择器匹配的范围，即判断 *super* 选择器是否包含了 *sub* 选择器所匹配的范围，是的话返回 true，否则返回 false。  **实例:** is-superselector("div", "div.myInput") 结果: true is-superselector("div.myInput", "div") 结果: false is-superselector("div", "div") 结果: true |
| selector-append(*selectors*) | 将第二个 (也可以有多个) 添加到第一个选择器的后面。 selector.  **实例:** selector-append("div", ".myInput") 结果: div.myInput selector-append(".warning", "\_\_a") 结果: .warning\_\_a |
| selector-extend(*selector*, *extendee*, *extender*) |  |
| selector-nest(*selectors*) | 返回一个新的选择器，该选择器通过提供的列表选择器生成一个嵌套的列表。  **实例:** selector-nest("ul", "li") 结果: ul li selector-nest(".warning", "alert", "div") 结果: .warning div, alert div |
| selector-parse(*selector*) | 将字符串的选择符 *selector* 转换成选择器队列。  **实例:** selector-parse("h1 .myInput .warning") 结果: ('h1' '.myInput' '.warning') |
| selector-replace(*selector*, *original*, *replacement*) | 给定一个选择器，用replacement 替换 original 后返回一个新的选择器队列。  **实例:** selector-replace("p.warning", "p", "div") 结果: div.warning |
| selector-unify(*selector1*, *selector2*) | 将两组选择器合成一个复合选择器。如两个选择器无法合成，则返回 null 值。  **实例:** selector-unify("myInput", ".disabled") 结果: myInput.disabled selector-unify("p", "h1") 结果: null |
| simple-selectors(*selectors*) | 将合成选择器拆为单个选择器。  **实例:** simple-selectors("div.myInput") 结果: div, .myInput simple-selectors("div.myInput:before") 结果: div, .myInput, :before |

## Sass Introspection 函数

|  |  |
| --- | --- |
| **函数** | **描述 & 实例** |
| call(*function*, *arguments*...) | 函数的动态调用，即调用函数 function 参数为 arguments，并返回结果。 |
| content-exists() | 查看当前的混入是否传递 @content 块。 |
| feature-exists(*feature*) | 检查当前的 Sass 实现是否支持该特性。  **实例:** feature-exists("at-error"); 结果: true |
| function-exists(*functionname*) | 检测指定的函数是否存在  **实例:** function-exists("nonsense") 结果: false |
| get-function(*functionname*, css: false) | 返回指定函数。如果 css 为 true，则返回纯 CSS 函数。 |
| global-variable-exists(*variablename*) | 检测某个全局变量是否定义。  **实例:** variable-exists(a) 结果: true |
| inspect(*value*) | 返回一个字符串的表示形式，value 是一个 sass 表达式。 |
| mixin-exists(*mixinname*) | 检测指定混入 (mixinname) 是否存在。  **实例:** mixin-exists("important-text") 结果: true |
| type-of(*value*) | 返回值类型。返回值可以是 number, string, color, list, map, bool, null, function, arglist。  **实例:** type-of(15px) 结果: number type-of(#ff0000) 结果: color |
| unit(*number*) | 返回传入数字的单位（或复合单位）。  **实例:** unit(15px) 结果: px |
| unitless(*number*) | 返回一个布尔值，判断传入的数字是否带有单位。  **实例:** unitless(15px) 结果: false unitless(15) 结果: true |
| variable-exists(*variablename*) | 判断变量是否在当前的作用域下。  **实例:** variable-exists(b) 结果: true |

## Sass 颜色函数

### Sass 颜色设置

|  |  |
| --- | --- |
| **函数** | **描述 & 实例** |
| rgb(red, green, blue) | 创建一个 Red-Green-Blue (RGB) 色。其中 R 是 "red" 表示红色，而 G 是 "green" 绿色，B 是 "blue" 蓝色。  **实例:** rgb(0, 0, 255); |
| rgba(red, green, blue, alpha) | 根据红、绿、蓝和透明度值创建一个颜色。  **实例:** rgba(0, 0, 255, 0.3); |
| hsl(hue, saturation, lightness) | 通过色相（hue）、饱和度(saturation)和亮度（lightness）的值创建一个颜色。  **实例:** hsl(120, 100%, 50%); // 绿色 hsl(120, 100%, 75%); // 浅绿色 hsl(120, 100%, 25%); // dark green hsl(120, 60%, 70%); // 柔和的绿色 |
| hsla(hue, saturation, lightness, alpha) | 通过色相（hue）、饱和度(saturation)、亮度（lightness）和透明（alpha）的值创建一个颜色。  **实例:** hsl(120, 100%, 50%, 0.3); // 绿色带有透明度 hsl(120, 100%, 75%, 0.3); // 浅绿色带有透明度 |
| grayscale(color) | 将一个颜色变成灰色，相当于 desaturate( color,100%)。  **实例:** grayscale(#7fffd4); 结果: #c6c6c6 |
| complement(color) | 返回一个补充色，相当于adjust-hue($color,180deg)。  **实例:** complement(#7fffd4); 结果: #ff7faa |
| invert(color, weight) | 返回一个反相色，红、绿、蓝色值倒过来，而透明度不变。  **实例:** invert(white); 结果: black |

### Sass 颜色获取

|  |  |
| --- | --- |
| **函数** | **描述 & 实例** |
| red(color) | 从一个颜色中获取其中红色值（0-255）。  **实例:** red(#7fffd4); 结果: 127 red(red); 结果: 255 |
| green(color) | 从一个颜色中获取其中绿色值（0-255）。  **实例:** green(#7fffd4); 结果: 255 green(blue); 结果: 0 |
| blue(color) | 从一个颜色中获取其中蓝色值（0-255）。  **实例:** blue(#7fffd4); 结果: 212 blue(blue); 结果: 255 |
| hue(color) | 返回颜色在 HSL 色值中的角度值 (0deg - 255deg)。  **实例:** hue(#7fffd4); 结果: 160deg |
| saturation(color) | 获取一个颜色的饱和度值(0% - 100%)。  **实例:** saturation(#7fffd4); 结果: 100% |
| lightness(color) | 获取一个颜色的亮度值(0% - 100%)。  **实例:** lightness(#7fffd4); 结果: 74.9% |
| alpha(color) | Returns the alpha channel of color as a number between 0 and 1.  **实例:** alpha(#7fffd4); 结果: 1 |
| opacity(color) | 获取颜色透明度值(0-1)。  **实例:** opacity(rgba(127, 255, 212, 0.5); 结果: 0.5 |

### Sass 颜色操作

|  |  |
| --- | --- |
| **函数** | **描述 & 实例** |
| mix(color1, color2, weight) | 把两种颜色混合起来。 weight 参数必须是 0% 到 100%。默认 weight 为 50%，表明新颜色各取 50% color1 和 color2 的色值相加。如果 weight 为 25%，那表明新颜色为 25% color1 和 75% color2 的色值相加。 |
| adjust-hue(color, degrees) | 通过改变一个颜色的色相值（-360deg - 360deg），创建一个新的颜色。  **实例:** adjust-hue(#7fffd4, 80deg); 结果: #8080ff |
| adjust-color(color, red, green, blue, hue, saturation, lightness, alpha) | 这个函数能够调整给定色彩的一个或多个属性值，包括 RGB 和 HSL 色彩的各项色值参数，另外还有 alpha 通道的取值。这些属性值的调整依赖传入的关键值参数，通过这些参数再与给定颜色相应的色彩值做加减运算。  **实例:** adjust-color(#7fffd4, blue: 25); 结果: |
| change-color(color, red, green, blue, hue, saturation, lightness, alpha) | 跟上面 adjust-color 类似，只是在该函数中传入的参数将直接替换原来的值，而不做任何的运算。  **实例:** change-color(#7fffd4, red: 255); 结果: #ffffd4 |
| scale-color(color, red, green, blue,  saturation, lightness, alpha) | 另一种实用的颜色调节函数。adjust-color 通过传入的参数简单的与本身的色值参数做加减，有时候可能会导致累加值溢出，当然，函数会把结果控制在有效的阈值内。而 scale-color 函数则避免了这种情况，可以不必担心溢出，让参数在阈值范围内进行有效的调节。  举个例子，一个颜色的亮度 lightness 取值在 0% ~ 100% 之间，假如执行 scale-color($color, $lightness: 40%)，表明该颜色的亮度将有 (100 - 原始值) × 40% 的增幅。  另一个例子，执行 scale-color($color, $lightness: -40%)，表明这个颜色的亮度将减少 (原始值 - 0) × 40% 这么多的值。  所有传参的取值范围都在 0% ~ 100% 之间，并且 RGB 同 HSL 的传参不能冲突。  scale-color(hsl(120, 70%, 80%), $lightness: 50%) => hsl(120, 70%, 90%)  scale-color(rgb(200, 150, 170), $green: -40%, $blue: 70%) => rgb(200, 90, 229)  scale-color(hsl(200, 70%, 80%), $saturation: -90%, $alpha: -30%) => hsla(200, 7%, 80%, 0.7) |
| rgba(color, alpha) | 根据红、绿、蓝和透明度值创建一个颜色。  **实例:** rgba(#7fffd4, 30%); 结果: rgba(127, 255, 212, 0.3) |
| lighten(color, amount) | 通过改变颜色的亮度值（0% - 100%），让颜色变亮，创建一个新的颜色。 |
| darken(color, amount) | 通过改变颜色的亮度值（0% - 100%），让颜色变暗，创建一个新的颜色。 |
| saturate(color, amount) | 提高传入颜色的色彩饱和度。等同于 adjust-color( color, saturation: amount) |
| desaturate(color, amount) | 调低一个颜色的饱和度后产生一个新的色值。同样，饱和度的取值区间在 0% ~ 100%。等同于 adjust-color(color, saturation: -amount) |
| opacify(color, amount) | 降低颜色的透明度，取值在 0-1 之。等价于 adjust-color(color, alpha: amount) |
| fade-in(color, amount) | 降低颜色的透明度，取值在 0-1 之。等价于 adjust-color(color, alpha: amount) |
| transparentize(color, amount) | 提升颜色的透明度，取值在 0-1 之间。等价于 adjust-color(color, alpha: -amount) |
| fade-out(color, amount) | 提升颜色的透明度，取值在 0-1 之间。等价于 adjust-color(color, alpha: -amount) |

# CSS

## 选择器

### Id选择器

### 派生选择器

### 类选择器

.important {color:red;}

#### 多类选择器

我们假设 class 为 important 的所有元素都是粗体，而 class 为 warning 的所有元素为斜体，class 中同时包含 important 和 warning 的所有元素还有一个银色的背景 。就可以写作：

.important {font-weight:bold;}

.warning {font-style:italic;}

.important.warning {background:silver;}

注意多类选择器中两个类之间没有空格。

两个类选择器之间有空格，是选择到.class1类下的.class2类子节点，即.class2类的节点要是.class1类子节点

.class1 { color: black; }

.class1 .class2 { color: red; }

<div class="class1">

I'm class1

<div class="class2">I'm class2,son of class1</div>

</div>

### 属性选择器

## 属性

### Padding

**padding:10px 5px 15px 20px;**

* 上填充是 10px
* 右填充是 5px
* 下填充是 15px
* 左填充是 20px

**padding:10px 5px 15px;**

* 上填充是 10px
* 右填充和左填充是 5px
* 下填充是 15px

**padding:10px 5px;**

* 上填充和下填充是 10px
* 右填充和左填充是 5px
* **padding:10px;**
  + 所有四个填充都是 10px

### Margin

格式与padding相同

### Box-shadow

box-shadow: 0 0 20rpx 0 rgba(0, 0, 0, 0.5);

### background

background: linear-gradient(#fff, #448de0);

### border

border: 2px solid #0ea391;

### text-align

文本的水平对齐方式

## 百分比

先说说百分比，百分比是相对父对象的

## Auto

块级元素水平方向的auto，块级元素的margin、border、padding以及content宽度之和等于父元素width。使用auto属性在父元素宽度变化的时候，该元素的宽度也会随之变化

****但是当该元素被设为浮动时，该元素的width就变成了内容的宽度了，由内容撑开，也就是所谓的有了包裹性。overflow | position:absolute | float:left/right都可以产生包裹性，替换元素也同样具有包裹性。****

在具有包裹性的元素上想利用width : auto；来让元素宽度自适应浏览器宽是不行的

text-align: center;

## 数组

Array.Spice（index，number，item）

# Flex

## 主轴对齐justify-content

justify-content: flex-start | flex-end | center | space-between | space-around;

space-between：两端对齐，项目之间间隔相等

space-around: 每个项目两侧的间隔相等，即项目之间的间隔比项目与边框的间隔大一倍

## 交叉轴对齐 align-items

align-items: flex-start | flex-end | center | baseline | stretch;

## 多轴对齐 align-content

align-content: flex-start | flex-end | center | space-between | space-around | stretch;



## **flex-direction: row | row-reverse | column | column-reverse;**

**flex-wrap: nowrap | wrap | wrap-reverse ;**

## flex-flow: <flex-direction> || <flex-wrap>;

## 传参

<div class="round" style="--radius: var(--rad-50);"></div>

<style>

:root {

--rad-0: 0%;

--rad-50: 50%;

--rad-100: 100%;

}

.round {

display: block;

height: 40px;

width: 40px;

border: 1px solid #BADA55;

border-radius: var(--radius);

}

</style>

# IconFont

<https://www.iconfont.cn/>

# pixel

Huawei mate30 pro 2400\*1176

# JS

## Math

取整

Math.floor()

var num1 = 123.8976;

var result = Math.floor(num1);

console.log(result); //123

var result2 = Math.floor(num1\*100)/100;

console.log(result2) //向下取整保留两位小数 123.89

# Erros

## Vue不是一个内部命令也不是一个外部

cnpm i vue-cli -g

div没有内容的时候没有宽度，需要设置flex-basis和height

# skill

## 为数组中的不同的项分配class

:class="[{ light: i % 2 != 1 }, { dark: i % 2 != 0 }]"

:class="[{ 'nodisplay': false }]"

:style="[{ 'background-color': item }]"

:style="{background: 'linear-gradient(' + colorred + ', ' + colorgreen + ')'}"

:style="{color:(index==0?arr.conFontColor:'#000')}"

:style="{ border: cfg.classTextShowBorder ? '1px solid' + cfg.classTextBorderColor : transparent }"

//style中使用函数

:style="{ 'box-shadow': isEmphasis(i) ? '2px 2px 5px 4px ' + cfg.highlightBorderColor : 'transparent' }"

## Display：flex与v-show不兼容

v-show在如下情况下有效

<view @click="onExtendClassDurMax" v-show="cfgt.dur >= 240">></view>

<text @click="onExtendClassDurMax" v-show="cfgt.dur >= 240">></text>

加上display：flex后会导致v-show判断无效，始终处于显示状态

可以在需要隐藏的部分外加一个view即可。

<http://www.bootstrapmb.com/item/6901>

<http://www.bootstrapmb.com/item/6879>

## 如何设置微信开发者工具里的手机模拟器

Hbuildx连接手机，运行，调用uni.getSystemInfo读取sysInfo

screenWidth：模拟器设置【宽度】

screenHeight：模拟器设置【高度】

pixelRatio：模拟器设置【DPR】

## flex中独占一行/列

.container {

display: flex;

justify-content: space-between;

flex-wrap: wrap;

}

.item:nth-last-child(1){

flex-basis: 100%;

}

## 子元素相对父元素定位

**在父元素中加入position:relative;  
子元素中加入position:absolute;**

**然后在子元素中利用top/bottom/left/right进行定为**

**比如top：-20px；right：-20px**

## v-if和v-show的区别

简单来说，v-if 的初始化较快，但切换代价高；v-show 初始化慢，但切换成本低

### 1.共同点

都是动态显示DOM元素

### 2.区别

（1）手段：

v-if是动态的向DOM树内添加或者删除DOM元素；

v-show是通过设置DOM元素的display样式属性控制显隐；

（2）编译过程：

v-if切换有一个局部编译/卸载的过程，切换过程中合适地销毁和重建内部的事件监听和子组件；

v-show只是简单的基于css切换；

（3）编译条件：

v-if是惰性的，如果初始条件为假，则什么也不做；只有在条件第一次变为真时才开始局部编译（编译被缓存？编译被缓存后，然后再切换的时候进行局部卸载);

v-show是在任何条件下（首次条件是否为真）都被编译，然后被缓存，而且DOM元素保留；

（4）性能消耗：

v-if有更高的切换消耗；

v-show有更高的初始渲染消耗；

（5）使用场景：

v-if适合运营条件不大可能改变；

v-show适合频繁切换。

作者：一念天堂\_5e04  
链接：https://www.jianshu.com/p/934739ca1c0b  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## uni-segmented-control

这个空间外面不能用flex。Emmmmmm

## Vuex里面数组在push json类型的数据时，会改变之前push的数据，导致所有的数据都变成最后push的数据

复制一份数据，然后再push

let data = Object.assign({}, JSON.parse(JSON.stringify(index)));

array.push(data)

# materials

## [这可能是史上最全的CSS自适应布局总结教程](https://www.cnblogs.com/qieguo/p/5421252.html)

<https://www.cnblogs.com/qieguo/p/5421252.html>