**奶茶点餐项目技术规范文档**

**技术规范文档**

**1. ES6（ECMAScript 6）**

参考资料：

<https://es6.ruanyifeng.com/#docs/reference>    （主要全面讲解）

<https://www.cnblogs.com/burningmyself/p/7451393.html>（主要运用简介）

**(1) 简介**

ECMAScript 6 简称 ES6，是 JavaScript 语言的下一代标准，已经在2015年6月正式发布了。它的目标是使得 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

ECMAScript 和 JavaScript 的关系：前者是后者的语法规格，后者是前者的一种实现

[Babel](http://babeljs.io/)：将ES6代码转为ES5代码

**(2) 新特性**

新特性

let、const

let 定义的变量不会被变量提升，const 定义的常量不能被修改，let 和 const 都是块级作用域

ES6前，js 是没有块级作用域 {} 的概念的。（有函数作用域、全局作用域、eval作用域）

ES6后，let 和 const 的出现，js 也有了块级作用域的概念，前端的知识是日新月异的~

变量提升：在ES6以前，var关键字声明变量。无论声明在何处，都会被视为声明在函数的最顶部；不在函数内即在全局作用域的最顶部。这样就会引起一些误解。例如：

console.log(a); // undefined

var a = 'hello';

# 上面的代码相当于

var a;

console.log(a);

a = 'hello';

# 而 let 就不会被变量提升

console.log(a); // a is not defined

let a = 'hello';

const 定义的常量不能被修改

var name = "bai";

name = "ming";console.log(name); // mingconst name = "bai";

name = "ming"; // Assignment to constant variable.console.log(name);

**(3) import、export**

// 全部导入

import people from './example'

// 将整个模块当作单一对象进行导入，该模块的所有导出都会作为对象的属性存在

import \* as example from "./example.js"

console.log(example.name)

console.log(example.getName())

// 导入部分，引入非 default 时，使用花括号

import {name, age} from './example'

// 导出默认, 有且只有一个默认

export default App

// 部分导出

export class App extend Component {};

**(4) class、extends、super**

ES5中最令人头疼的的几个部分：原型、构造函数，继承，有了ES6我们不再烦恼！

ES6引入了Class（类）这个概念。

class Animal {

constructor() {

this.type = 'animal';

}

says(say) {

console.log(this.type + ' says ' + say);

}

}

let animal = new Animal();

animal.says('hello'); //animal says hello

class Cat extends Animal {

constructor() {

super();

this.type = 'cat';

}

}

let cat = new Cat();

cat.says('hello'); //cat says hello

上面代码首先用class定义了一个“类”，可以看到里面有一个constructor方法，这就是构造方法，而this关键字则代表实例对象。简单地说，constructor内定义的方法和属性是实例对象自己的，而constructor外定义的方法和属性则是所有实力对象可以共享的。

Class之间可以通过extends关键字实现继承，这比ES5的通过修改原型链实现继承，要清晰和方便很多。上面定义了一个Cat类，该类通过extends关键字，继承了Animal类的所有属性和方法。

super关键字，它指代父类的实例（即父类的this对象）。子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法，子类就得不到this对象。

ES6的继承机制，实质是先创造父类的实例对象this（所以必须先调用super方法），然后再用子类的构造函数修改this。

// ES5

var Shape = function(id, x, y) {

this.id = id,

this.move(x, y);

};

Shape.prototype.move = function(x, y) {

this.x = x;

this.y = y;

};

var Rectangle = function id(ix, x, y, width, height) {

Shape.call(this, id, x, y);

this.width = width;

this.height = height;

};

Rectangle.prototype = Object.create(Shape.prototype);

Rectangle.prototype.constructor = Rectangle;

var Circle = function(id, x, y, radius) {

Shape.call(this, id, x, y);

this.radius = radius;

};

Circle.prototype = Object.create(Shape.prototype);

Circle.prototype.constructor = Circle;

// ES6

class Shape {

constructor(id, x, y) {

this.id = id this.move(x, y);

}

move(x, y) {

this.x = x this.y = y;

}

}

class Rectangle extends Shape {

constructor(id, x, y, width, height) {

super(id, x, y) this.width = width this.height = height;

}

}

class Circle extends Shape {

constructor(id, x, y, radius) {

super(id, x, y) this.radius = radius;

}

}

**(5) arrow functions （箭头函数）**

// ES5

var arr1 = [1, 2, 3];

var newArr1 = arr1.map(function(x) {

return x + 1;

});

// ES6

let arr2 = [1, 2, 3];

let newArr2 = arr2.map((x) => {

x + 1

});

箭头函数小细节：当你的函数有且仅有一个参数的时候，是可以省略掉括号的；当你函数中有且仅有一个表达式的时候可以省略{}

**(6) This**

JavaScript语言的this对象一直是一个令人头痛的问题，运行上面的代码会报错，这是因为setTimeout中的this指向的是全局对象。

class Animal {

constructor() {

this.type = 'animal';

}

says(say) {

setTimeout(function() {

console.log(this.type + ' says ' + say);

}, 1000);

}

}

var animal = new Animal();

animal.says('hi'); //undefined says hi

解决办法：

// 传统方法1: 将this传给self,再用self来指代this

says(say) {

var self = this;

setTimeout(function() {

console.log(self.type + ' says ' + say);

}, 1000);

}

// 传统方法2: 用bind(this),即

says(say) {

setTimeout(function() {

console.log(this.type + ' says ' + say);

}.bind(this), 1000);

}

// ES6: 箭头函数

// 当我们使用箭头函数时，函数体内的this对象，就是定义时所在的对象says(say) {

setTimeout(() => {

console.log(this.type + ' says ' + say);

}, 1000);

}

**(7) template string （模板字符串）**

解决了 ES5 在字符串功能上的痛点。

第一个用途：字符串拼接。将表达式嵌入字符串中进行拼接，用 和${}来界定。

// es5

var name1 = "bai";

console.log('hello' + name1);

// es6

const name2 = "ming";

console.log(`hello${name2}`);

第二个用途：在ES5时我们通过反斜杠来做多行字符串拼接。ES6反引号 `` 直接搞定。

// es5

var msg = "Hi \

man!";

// es6

const template = `<div>

<span>hello world</span>

</div>`;

另外：includes repeat

// includes：判断是否包含然后直接返回布尔值

let str = 'hahah';

console.log(str.includes('y')); // false

// repeat: 获取字符串重复n次

let s = 'he';

console.log(s.repeat(3)); // 'hehehe'

**(8) destructuring （解构）**

简化数组和对象中信息的提取。

ES6前，我们一个一个获取对象信息；

ES6后，解构能让我们从对象或者数组里取出数据存为变量

// ES5

var people1 = {

name: 'bai',

age: 20,

color: ['red', 'blue']

};

var myName = people1.name;

var myAge = people1.age;

var myColor = people1.color[0];

console.log(myName + '----' + myAge + '----' + myColor);

// ES6

let people2 = {

name: 'ming',

age: 20,

color: ['red', 'blue']

}

let { name, age } = people2;

let [first, second] = people2.color;

console.log(`${name}----${age}----${first}`);

**(9) default 函数默认参数**

// ES5 给函数定义参数默认值

function foo(num) {

num = num || 200;

return num;

}

// ES6

function foo(num = 200) {

return num;

}

**(10)   rest arguments （rest参数）**

解决了 es5 复杂的 arguments 问题

function foo(x, y, ...rest) {

return ((x + y) \* rest.length);

}

foo(1, 2, 'hello', true, 7); // 9

**(11)   Spread Operator （展开运算符）**

第一个用途：组装数组

let color = ['red', 'yellow'];

let colorful = [...color, 'green', 'blue'];

console.log(colorful); // ["red", "yellow", "green", "blue"]

第二个用途：获取数组除了某几项的其他项

let num = [1, 3, 5, 7, 9];

let [first, second, ...rest] = num;

console.log(rest); // [5, 7, 9]

**(12)   对象**

对象初始化简写

// ES5

function people(name, age) {

return {

name: name,

age: age

};

}

// ES6

function people(name, age) {

return {

name,

age

};

}

对象字面量简写（省略冒号与 function 关键字）

// ES5

var people1 = {

name: 'bai',

getName: function () {

console.log(this.name);

}

};

// ES6

let people2 = {

name: 'bai',

getName () {

console.log(this.name);

}

};

另外：Object.assign()

ES6 对象提供了Object.assign()这个方法来实现浅复制。Object.assign()可以把任意多个源对象自身可枚举的属性拷贝给目标对象，然后返回目标对象。第一参数即为目标对象。在实际项目中，我们为了不改变源对象。一般会把目标对象传为{}

const obj = Object.assign({}, objA, objB)

// 给对象添加属性

this.seller = Object.assign({}, this.seller, response.data)

**(13)   Promise**

用同步的方式去写异步代码

// 发起异步请求

fetch('/api/todos')

.then(res => res.json())

.then(data => ({

data

}))

.catch(err => ({

err

}));

**(14)   Generators**

生成器（ generator）是能返回一个迭代器的函数。

生成器函数也是一种函数，最直观的表现就是比普通的function多了个星号\*，在其函数体内可以使用yield关键字,有意思的是函数会在每个yield后暂停。

这里生活中有一个比较形象的例子。咱们到银行办理业务时候都得向大厅的机器取一张排队号。你拿到你的排队号，机器并不会自动为你再出下一张票。也就是说取票机“暂停”住了，直到下一个人再次唤起才会继续吐票。

迭代器：当你调用一个generator时，它将返回一个迭代器对象。这个迭代器对象拥有一个叫做next的方法来帮助你重启generator函数并得到下一个值。next方法不仅返回值，它返回的对象具有两个属性：done和value。value是你获得的值，done用来表明你的generator是否已经停止提供值。继续用刚刚取票的例子，每张排队号就是这里的value，打印票的纸是否用完就这是这里的done。

// 生成器

function \*createIterator() {

yield 1;

yield 2;

yield 3;

}

// 生成器能像正规函数那样被调用，但会返回一个迭代器

let iterator = createIterator();

console.log(iterator.next().value); // 1

console.log(iterator.next().value); // 2

console.log(iterator.next().value); // 3

迭代器对异步编程作用很大，异步调用对于我们来说是很困难的事，我们的函数并不会等待异步调用完再执行，你可能会想到用回调函数，（当然还有其他方案比如Promise比如Async/await）。

生成器可以让我们的代码进行等待。就不用嵌套的回调函数。使用generator可以确保当异步调用在我们的generator函数运行一下行代码之前完成时暂停函数的执行。

那么问题来了，咱们也不能手动一直调用next()方法，你需要一个能够调用生成器并启动迭代器的方法。就像这样子的：

function run(taskDef) {

// taskDef 即一个生成器函数

// 创建迭代器，让它在别处可用

let task = taskDef();

// 启动任务

let result = task.next();

// 递归使用函数来保持对 next() 的调用

function step() {

// 如果还有更多要做的

if (!result.done) {

result = task.next();

step();

}

}

// 开始处理过程

step();

}

**2. Vue.js**

参考资料：<https://cn.vuejs.org/v2/style-guide/#%E8%A7%84%E5%88%99%E5%BD%92%E7%B1%BB>

<https://www.runoob.com/vue2/vue-install.html>

<https://blog.csdn.net/caijunfen/article/details/86548895>

**(1) 规范目的**

为提高团队协作效率，便于后台人员添加功能及前端后期优化维护，输出高质量的文档

**(2) 命名规范**

为了让大家书写可维护的代码，而不是一次性的代码

让团队当中其他人看你的代码能一目了然

甚至一段时间时候后你再看你某个时候写的代码也能看

**(3) 普通变量命名规范**

命名方法 ：驼峰命名法

命名规范 ：

命名必须是跟需求的内容相关的词，比如说我想申明一个变量，用来表示我的学校，那么我们可以这样定义const mySchool = "我的学校";

命名是复数的时候需要加s,比如说我想申明一个数组，表示很多人的名字，那么我们可以这样定义const names = new Array();

**(4) 常量**

命名方法 : 全部大写

命名规范 : 使用大写字母和下划线来组合命名，下划线用以分割单词。

const MAX\_COUNT = 10

const URL = 'https://www.baidu.com/'

**(5) 组件命名规范**

官方文档推荐及使用遵循规则：

PascalCase (单词首字母大写命名)是最通用的声明约定

kebab-case (短横线分隔命名) 是最通用的使用约定

组件名应该始终是多个单词的，根组件 App 除外

有意义的名词、简短、具有可读性

命名遵循 PascalCase 约定

公用组件以 Abcd (公司名缩写简称) 开头，如（AbcdDatePicker,AbcdTable）

页面内部组件以组件模块名简写为开头，Item 为结尾，如（StaffBenchToChargeItem，StaffBenchAppNotArrItem）

使用遵循 kebab-case 约定

在页面中使用组件需要前后闭合，并以短线分隔，如（<abcd-date-picker></abcd-date-picker>，<abcd-table></abcd-table>）

导入及注册组件时，遵循 PascalCase 约定

同时还需要注意：必须符合自定义元素规范: 切勿使用保留字。

**(6) method 方法命名命名规范**

驼峰式命名，统一使用动词或者动词+名词形式

 //bad

 go、nextPage、show、open、login

   // good

 jumpPage、openCarInfoDialog

请求数据方法，以 data 结尾

 //bad

 takeData、confirmData、getList、postForm

 // good

 getListData、postFormData

复制代码

init、refresh 单词除外

尽量使用常用单词开头（set、get、go、can、has、is）

附： 函数方法常用的动词:

get 获取/set 设置,

add 增加/remove 删除

create 创建/destory 移除

start 启动/stop 停止

open 打开/close 关闭,

read 读取/write 写入

load 载入/save 保存,

create 创建/destroy 销毁

begin 开始/end 结束,

backup 备份/restore 恢复

import 导入/export 导出,

split 分割/merge 合并

inject 注入/extract 提取,

attach 附着/detach 脱离

bind 绑定/separate 分离,

view 查看/browse 浏览

edit 编辑/modify 修改,

select 选取/mark 标记

copy 复制/paste 粘贴,

undo 撤销/redo 重做

insert 插入/delete 移除,

add 加入/append 添加

clean 清理/clear 清除,

index 索引/sort 排序

find 查找/search 搜索,

increase 增加/decrease 减少

play 播放/pause 暂停,

launch 启动/run 运行

compile 编译/execute 执行,

debug 调试/trace 跟踪

observe 观察/listen 监听,

build 构建/publish 发布

input 输入/output 输出,

encode 编码/decode 解码

encrypt 加密/decrypt 解密,

compress 压缩/decompress 解压缩

pack 打包/unpack 解包,

parse 解析/emit 生成

connect 连接/disconnect 断开,

send 发送/receive 接收

download 下载/upload 上传,

refresh 刷新/synchronize 同步

update 更新/revert 复原,

lock 锁定/unlock 解锁

check out 签出/check in 签入,

submit 提交/commit 交付

push 推/pull 拉,

expand 展开/collapse 折叠

begin 起始/end 结束,

start 开始/finish 完成

enter 进入/exit 退出,

abort 放弃/quit 离开

obsolete 废弃/depreciate 废旧,

collect 收集/aggregate 聚集

**(7) views 下的文件命名**

只有一个文件的情况下不会出现文件夹，而是直接放在 views 目录下面，如 index.vue

尽量是名词,且使用驼峰命名法

开头的单词就是所属模块名字（workbenchIndex、workbenchList、workbenchEdit）

名字至少两个单词（good: workbenchIndex）（bad:workbench）

**(8) props 命名**

在声明 prop 的时候，其命名应该始终使用 camelCase，而在模板中应该始终使用 kebab-case

<!-- bad -->

<script>

props: {

 'greeting-text': String

}

</script>

<welcome-message greetingText="hi"></welcome-message>

<!-- good -->

<script>

props: {

 greetingText: String

}

</script>

<welcome-message greeting-text="hi"></welcome-message>

**(9) 例外情况**

作用域不大临时变量可以简写，比如：str，num，bol，obj，fun，arr。

循环变量可以简写，比如：i，j，k 等。

**(10)   结构化规范**

目录文件夹及子文件规范

以下统一管理处均对应相应模块

以下全局文件文件均以 index.js 导出，并在 main.js 中导入

以下临时文件，在使用后，接口已经有了，发版后清除

src                               源码目录

|-- api                              接口，统一管理

|-- assets                           静态资源，统一管理

|-- components                       公用组件，全局文件

|-- filters                          过滤器，全局工具

|-- icons                            图标，全局资源

|-- datas                            模拟数据，临时存放

|-- lib                              外部引用的插件存放及修改文件

|-- mock                             模拟接口，临时存放

|-- router                           路由，统一管理

|-- store                            vuex, 统一管理

|-- views                         视图目录

|   |-- staffWorkbench               视图模块名

|   |-- |-- staffWorkbench.vue       模块入口页面

|   |-- |-- indexComponents          模块页面级组件文件夹

|   |-- |-- components               模块通用组件文件夹

**(11)   vue 文件基本结构**

 <template>

   <div>

     <!--必须在div中编写页面-->

   </div>

 </template>

 <script>

   export default {

     components : {

     },

     data () {

       return {

       }

     },

     mounted() {

     }，

     methods: {

     }

  }

 </script>

 <!--声明语言，并且添加scoped-->

 <style lang="scss" scoped>

 </style>

**(12)   多个特性的元素规范**

多个特性的元素应该分多行撰写，每个特性一行。(增强更易读)

<!-- bad -->

<img src="https://vuejs.org/images/logo.png" alt="Vue Logo">

<my-component foo="a" bar="b" baz="c"></my-component>

<!-- good -->

<img

 src="https://vuejs.org/images/logo.png"

 alt="Vue Logo"

>

<my-component

 foo="a"

 bar="b"

 baz="c"

>

</my-component>

**(13)   元素特性的顺序**

原生属性放前面，指令放后面

如下所示：

 - class

 - id,ref

 - name

 - data-\*

 - src, for, type, href,value,max-length,max,min,pattern

 - title, alt，placeholder

 - aria-\*, role

 - required,readonly,disabled

 - is

 - v-for

 - key

 - v-if

 - v-else-if

 - v-else

 - v-show

 - v-cloak

 - v-pre

 - v-once

 - v-model

 - v-bind,:

 - v-on,@

 - v-html

 - v-text

**(14)   组件选项顺序**

如下所示：

 - components

 - props

 - data

 - computed

 - created

 - mounted

 - metods

 - filter

 - watch

**(15)   注释规范**

代码注释在一个项目的后期维护中显的尤为重要，所以我们要为每一个被复用的组件编写组件使用说明，为组件中每一个方法编写方法说明

务必添加注释列表

公共组件使用说明

各组件中重要函数或者类说明

复杂的业务逻辑处理说明

特殊情况的代码处理说明,对于代码中特殊用途的变量、存在临界值、函数中使用的 hack、使用了某种算法或思路等需要进行注释描述

多重 if 判断语句

注释块必须以/\*\*（至少两个星号）开头\*\*/

单行注释使用//

**(16)   单行注释**

注释单独一行，不要在代码后的同一行内加注释。例如：

 bad

 var name =”abc”; // 姓名

 good

 // 姓名

 var name = “abc”;

**(17)   多行注释**

组件使用说明，和调用说明

     /\*\*

     \* 组件名称

     \* @module 组件存放位置

     \* @desc 组件描述

     \* @author 组件作者

     \* @date 2017年12月05日17:22:43

     \* @param {Object} [title]    - 参数说明

     \* @param {String} [columns] - 参数说明

     \* @example 调用示例

     \*  <hbTable :title="title" :columns="columns" :tableData="tableData"></hbTable>

     \*\*/

**(18)   编码规范**

优秀的项目源码，即使是多人开发，看代码也如出一人之手。统一的编码规范，可使代码更易于阅读，易于理解，易于维护。尽量按照 ESLint 格式要求编写代码

**(19)   源码风格**

使用 ES6 风格编码

定义变量使用 let ,定义常量使用 const

静态字符串一律使用单引号或反引号，动态字符串使用反引号

 // bad

 const a = 'foobar'

 const b = 'foo' + a + 'bar'

 // acceptable

 const c = `foobar`

 // good

 const a = 'foobar'

 const b = `foo${a}bar`

 const c = 'foobar'

复制代码

解构赋值

数组成员对变量赋值时，优先使用解构赋值

 // 数组解构赋值

 const arr = [1, 2, 3, 4]

 // bad

 const first = arr[0]

 const second = arr[1]

 // good

 const [first, second] = arr

复制代码

函数的参数如果是对象的成员，优先使用解构赋值

 // 对象解构赋值

 // bad

 function getFullName(user) {

   const firstName = user.firstName

   const lastName = user.lastName

 }

 // good

 function getFullName(obj) {

   const { firstName, lastName } = obj

 }

 // best

 function getFullName({ firstName, lastName }) {}

复制代码

拷贝数组

使用扩展运算符（...）拷贝数组。

 const items = [1, 2, 3, 4, 5]

 // bad

 const itemsCopy = items

 // good

 const itemsCopy = [...items]

复制代码

箭头函数

需要使用函数表达式的场合，尽量用箭头函数代替。因为这样更简洁，而且绑定了 this

 // bad

 const self = this;

 const boundMethod = function(...params) {

   return method.apply(self, params);

 }

 // acceptable

 const boundMethod = method.bind(this);

 // best

 const boundMethod = (...params) => method.apply(this, params);

复制代码

模块

如果模块只有一个输出值，就使用 export default，如果模块有多个输出值，就不使用 export default，export default 与普通的 export 不要同时使用

 // bad

 import \* as myObject from './importModule'

 // good

 import myObject from './importModule'

复制代码

如果模块默认输出一个函数，函数名的首字母应该小写。

 function makeStyleGuide() {

 }

 export default makeStyleGuide;

复制代码

如果模块默认输出一个对象，对象名的首字母应该大写。

 const StyleGuide = {

   es6: {

   }

 };

 export default StyleGuide;

复制代码

指令规范

指令有缩写一律采用缩写形式

 // bad

 v-bind:class="{'show-left'：true}"

 v-on:click="getListData"

 // good

 :class="{'show-left'：true}"

 @click="getListData"

复制代码

v-for 循环必须加上 key 属性，在整个 for 循环中 key 需要唯一

 <!-- good -->

 <ul>

   <li v-for="todo in todos" :key="todo.id">

     {{ todo.text }}

   </li>

 </ul>

 <!-- bad -->

 <ul>

   <li v-for="todo in todos">

     {{ todo.text }}

   </li>

 </ul>

复制代码

避免 v-if 和 v-for 同时用在一个元素上（性能问题）

以下为两种解决方案：

将数据替换为一个计算属性，让其返回过滤后的列表

 <!-- bad -->

 <ul>

   <li v-for="user in users" v-if="user.isActive" :key="user.id">

     {{ user.name }}

   </li>

 </ul>

 <!-- good -->

 <ul>

   <li v-for="user in activeUsers" :key="user.id">

     {{ user.name }}

   </li>

 </ul>

 <script>

 computed: {

   activeUsers: function () {

     return this.users.filter(function (user) {

       return user.isActive

     })

   }

 }

 </script>

复制代码

将 v-if 移动至容器元素上 (比如 ul, ol)

 <!-- bad -->

 <ul>

   <li v-for="user in users" v-if="shouldShowUsers" :key="user.id">

     {{ user.name }}

   </li>

 </ul>

 <!-- good -->

 <ul v-if="shouldShowUsers">

   <li v-for="user in users" :key="user.id">

     {{ user.name }}

   </li>

 </ul>

复制代码

Props 规范

Props 定义应该尽量详细

// bad 这样做只有开发原型系统时可以接受

props: ['status']

// good

props: {

 status: {

   type: String,

   required: true,

   validator: function (value) {

     return [

       'syncing',

       'synced',

       'version-conflict',

       'error'

     ].indexOf(value) !== -1

   }

 }

}

复制代码

其他

避免 this.$parent

调试信息 console.log() debugger 使用完及时删除

除了三目运算，if,else 等禁止简写

 // bad

 if (true)

     alert(name);

 console.log(name);

 // bad

 if (true)

 alert(name);

 console.log(name)

 // good

 if (true) {

     alert(name);

 }

 console.log(name);

复制代码

CSS 规范

通用规范

统一使用"-"连字符

省略值为 0 时的单位

// bad

 padding-bottom: 0px;

 margin: 0em;

// good

 padding-bottom: 0;

 margin: 0;

复制代码

如果 CSS 可以做到，就不要使用 JS

建议并适当缩写值，提高可读性，特殊情况除外

“建议并适当”是因为缩写总是会包含一系列的值，而有时候我们并不希望设置某一值，反而造成了麻烦，那么这时候你可以不缩写，而是分开写。

当然，在一切可以缩写的情况下，请务必缩写，它最大的好处就是节省了字节，便于维护，并使阅读更加一目了然。

 // bad

 .box{

   border-top-style: none;

   font-family: palatino, georgia, serif;

   font-size: 100%;

   line-height: 1.6;

   padding-bottom: 2em;

   padding-left: 1em;

   padding-right: 1em;

   padding-top: 0;

 }

 // good

 .box{

   border-top: 0;

   font: 100%/1.6 palatino, georgia, serif;

   padding: 0 1em 2em;

 }

复制代码

声明应该按照下表的顺序

左到右，从上到下

显示属性       自身属性       文本属性和其他修饰

display    width      font

visibility  height     text-align

position  margin    text-decoration

float padding   vertical-align

clear border    white-space

list-style  overflow color

top  min-width      background

 // bad

 .box {

   font-family: 'Arial', sans-serif;

   border: 3px solid #ddd;

   left: 30%;

   position: absolute;

   text-transform: uppercase;

   background-color: #eee;

   right: 30%;

   isplay: block;

   font-size: 1.5rem;

   overflow: hidden;

   padding: 1em;

   margin: 1em;

 }

 // good

 .box {

   display: block;

   position: absolute;

   left: 30%;

   right: 30%;

   overflow: hidden;

   margin: 1em;

   padding: 1em;

   background-color: #eee;

   border: 3px solid #ddd;

   font-family: 'Arial', sans-serif;

   font-size: 1.5rem;

   text-transform: uppercase;

 }

元素选择器应该避免在 scoped 中出现

官方文档说明：在 scoped 样式中，类选择器比元素选择器更好，因为大量使用元素选择器是很慢的。

分类的命名方法

使用单个字母加上"-"为前缀

布局（grid）（.g-）；

模块（module）（.m-）；

元件（unit）（.u-）；

功能（function）（.f-）；

皮肤（skin）（.s-）；

状态（.z-）。

统一语义理解和命名

布局（.g-）

语义       命名       简写

文档       doc  doc

头部       head       hd

主体       body       bd

尾部       foot ft

主栏       main       mn

主栏子容器   mainc     mnc

侧栏       side sd

侧栏子容器   sidec       sdc

盒容器   wrap/box wrap/box

模块（.m-）、元件（.u-）

语义       命名       简写

导航       nav  nav

子导航   subnav    snav

面包屑   crumb     crm

菜单       menu      menu

选项卡   tab  tab

标题区   head/title       hd/tt

内容区   body/content  bd/ct

列表       list   lst

表格       table       tb

表单       form fm

热点       hot  hot

排行       top  top

登录       login log

标志       logo logo

广告       advertise ad

搜索       search     sch

幻灯       slide sld

提示       tips  tips

帮助       help help

新闻       news       news

下载       download       dld

注册       regist      reg

投票       vote vote

版权       copyright cprt

结果       result      rst

标题       title tt

按钮       button    btn

输入       input       ipt

功能（.f-）

语义       命名       简写

浮动清除       clearboth cb

向左浮动       floatleft   fl

向右浮动       floatright fr

内联块级       inlineblock      ib

文本居中       textaligncenter       tac

文本居右       textalignright  tar

文本居左       textalignleft    tal

垂直居中       verticalalignmiddle vam

溢出隐藏       overflowhidden      oh

完全消失       displaynone    dn

字体大小       fontsize   fs

字体粗细       fontweight      fw

皮肤（.s-）

语义       命名       简写

字体颜色       fontcolor fc

背景       background    bg

背景颜色       backgroundcolor    bgc

背景图片       backgroundimage   bgi

背景定位       backgroundposition       bgp

边框颜色       bordercolor    bdc

状态（.z-）

语义       命名       简写

选中       selected  sel

当前       current    crt

显示       show      show

隐藏       hide hide

打开       open       open

关闭       close       close

出错       error       err

不可用   disabled  dis

**3. axios 中请求格式规范**

参考文献：<http://www.axios-js.com/zh-cn/docs/>

<https://blog.csdn.net/zyf13671493506/article/details/104745790/>

**1、GET：查询数据**

      1.1、通过URL传递参数

   //前端

   axios.get('/path?id=123').then(function(ret){

           //ret是对象

           console.log(ret.data)

       })

   //服务器

   app.get('/path',(req,res)=>{

           res.send('axios get 传递参数'+req.query.id)

       })

   //前端

   axios.get('/path/123').then(function(ret){

           //ret是对象

           console.log(ret.data)

       })

   //服务器

   app.get('/path/:id',(req,res)=>{

           res.send('axios get (Restful) 传递参数'+req.params.id)

       })

     1.2、通过params选项传递参数

   //前端

   axios.get('/path',{

       params:{

          id:123

       }

   }).then(function(ret){

       console.log(ret.data)

   })

    //服务器

   app.get('/path',(req,res)=>{

           res.send('axios get 传递参数'+req.query.id)

   })

**2、POST：添加数据**

      2.1、通过选项传递参数（默认传递的是json格式的数据）

   //前端

   axios.post('/path',{

       name:'ming',

       pwd:123

   }).then(function(ret){

       console.log(ret.data)

   })

   //服务器

   app.post('/path',(req,res)=>{

       res.send('axios post 传递参数' + req.body.name + '----' + req.body.pwd);

   })

      2.1、通过URLSearchParams传递参数（application/x-www-form-urlencoded）

   //客户端

   var params = new URLSearchParams();

   params.append('name','xiang');

   params.append('pwd','123');

   axios.post('/path',params),then(function(ret){

       console.log(ret.data)

   })

   //服务器

   app.post('/path',(req,res)=>{

       res.send('axios post 传递参数' + req.body.name + '----' + req.body.pwd);

   })

**3、PUT：修改数据**

   //前端

   axios.put('/path/123',{

       name:'ming',

       pwd:123

   }).then(function(ret){

       console.log(ret.data)

   })

   //服务器

   app.put('/path/:id',(req,res)=>{

       res.send('axios post 传递参数' + req.params.id + '----' + req.body.name + '----' +     req.body.pwd);

   })

**4、DELETE：删除数据**

4.1、通过URL传递参数

   //前端

   axios.delete('/path?id=123').then(function(ret){

           //ret是对象

           console.log(ret.data)

       })

   //服务器

   app.delete('/path',(req,res)=>{

           res.send('axios get 传递参数'+req.query.id)

       })

   //前端

   axios.delete('/path/123').then(function(ret){

           //ret是对象

           console.log(ret.data)

       })

   //服务器

   app.delete('/path/:id',(req,res)=>{

           res.send('axios get (Restful) 传递参数'+req.params.id)

       })

4.2、通过params选项传递参数

   //前端

   axios.delete('/path',{

       params:{

          id:123

       }

   }).then(function(ret){

       console.log(ret.data)

   })

    //服务器

   app.delete('/path',(req,res)=>{

           res.send('axios get 传递参数'+req.query.id)

})

**4. Mock.js**

参考文献：<https://www.jianshu.com/p/4579f40e6108>

<http://mockjs.com/>

**语法规范**

生成随机数据，拦截 Ajax 请求(前端测试专用)

**数据模板定义规范 DTD**

数据模板中的每个属性由 3 部分构成：属性名、生成规则、属性值：

// 属性名   name// 生成规则 rule// 属性值   value'name|rule': value

**注意：**

* 属性名 和 生成规则 之间用竖线 | 分隔。
* 生成规则 是可选的。
* 生成规则 有 7 种格式：

'name|min-max': value  
'name|count': value  
'name|min-max.dmin-dmax': value  
'name|min-max.dcount': value  
'name|count.dmin-dmax': value  
'name|count.dcount': value  
'name|+step': value

·

**生成规则 的 含义 需要依赖 属性值的类型 才能确定。**

·

* 属性值 中可以含有 @占位符。
* 属性值 还指定了最终值的初始值和类型。

**生成规则和示例：**

**1. 属性值是字符串 String**

//通过重复 string 生成一个字符串，重复次数大于等于 min，小于等于 max。'name|min-max': string

通过重复 string 生成一个字符串，重复次数等于 count。'name|count': string

**2. 属性值是数字 Number**

//属性值自动加 1，初始值为 number。'name|+1': number

//生成一个大于等于 min、小于等于 max 的整数，属性值 number 只是用来确定类型。'name|min-max': number

//生成一个浮点数，整数部分大于等于 min、小于等于 max，小数部分保留 dmin 到 dmax 位。'name|min-max.dmin-dmax': number

Mock.mock({

'number1|1-100.1-10': 1,

'number2|123.1-10': 1,

'number3|123.3': 1,

'number4|123.10': 1.123})// =>{

"number1": 12.92,

"number2": 123.51,

"number3": 123.777,

"number4": 123.1231091814}

**3. 属性值是布尔型 Boolean**

//随机生成一个布尔值，值为 true 的概率是 1/2，值为 false 的概率同样是 1/2。'name|1': boolean//随机生成一个布尔值，值为 value 的概率是 min / (min + max)，值为 !value 的概率是 max / (min + max)。'name|min-max': value

**4. 属性值是对象 Object**

//从属性值 object 中随机选取 count 个属性。'name|count': object//从属性值 object 中随机选取 min 到 max 个属性。'name|min-max': object

**5. 属性值是数组 Array**

//从属性值 array 中随机选取 1 个元素，作为最终值。'name|1': array//从属性值 array 中顺序选取 1 个元素，作为最终值。'name|+1': array//通过重复属性值 array 生成一个新数组，重复次数大于等于 min，小于等于 max。'name|min-max': array//通过重复属性值 array 生成一个新数组，重复次数为 count。'name|count': array

Mock.mock(/api\/test/, {

//通过重复属性值 array 生成一个新数组，重复次数为 1-3次。

"favorite\_games|1-3": [3,5,4,6,23,28,42,45],});

**6. 属性值是函数 Function**

执行函数 function，取其返回值作为最终的属性值，函数的上下文为属性 'name' 所在的对象。

'name': function

**7. 属性值是正则表达式 RegExp**

根据正则表达式 regexp 反向生成可以匹配它的字符串。用于生成自定义格式的字符串。

'name': regexp

Mock.mock({

'regexp1': /[a-z][A-Z][0-9]/,

'regexp2': /\w\W\s\S\d\D/,

'regexp3': /\d{5,10}/})// =>{

"regexp1": "pJ7",

"regexp2": "F)\fp1G",

"regexp3": "561659409"}

**数据占位符定义规范 DPD**

占位符 只是在属性值字符串中占个位置，并不出现在最终的属性值中。

占位符 的格式为：

@占位符@占位符(参数 [, 参数])注意：

用 @ 来标识其后的字符串是 占位符。  
占位符 引用的是 Mock.Random 中的方法。  
通过 Mock.Random.extend() 来扩展自定义占位符。  
占位符 也可以引用 数据模板 中的属性。  
占位符 会优先引用 数据模板 中的属性。  
占位符 支持 相对路径 和 绝对路径。

Mock.mock({

name: {

first: '@FIRST',

middle: '@FIRST',

last: '@LAST',

full: '@first @middle @last'

}})// =>{

"name": {

"first": "Charles",

"middle": "Brenda",

"last": "Lopez",

"full": "Charles Brenda Lopez"

}}

//引入mockjsimport Mock from 'mockjs'const Random = Mock.Random;//使用mockjs模拟数据

Mock.mock('/api/msdk/proxy/query\_common\_credit', {

"ret":0,

"data":

{

"mtime": "@datetime",//随机生成日期时间

"score": "@natural(1, 800)",//随机生成1-800的数字

"rank": "@natural(1, 100)",//随机生成1-100的数字

"stars": "@natural(0, 5)",//随机生成1-5的数字

"nickname": "@cname",//随机生成中文名字

}});

**注意**

get请求有带参数时，请把接口url写为正则匹配，否则匹配不到就报错（Mock.mock('此处为正则'）