目录

**[补充常用语法](#_Toc13673_WPSOffice_Level1)** **[1](#_Toc13673_WPSOffice_Level1)**

[声明和解构](#_Toc21773_WPSOffice_Level2) [1](#_Toc21773_WPSOffice_Level2)

[类型](#_Toc10107_WPSOffice_Level2) [4](#_Toc10107_WPSOffice_Level2)

[字符串](#_Toc20128_WPSOffice_Level2) [5](#_Toc20128_WPSOffice_Level2)

[Iterator遍历器](#_Toc16472_WPSOffice_Level2) [5](#_Toc16472_WPSOffice_Level2)

[数组](#_Toc11885_WPSOffice_Level2) [6](#_Toc11885_WPSOffice_Level2)

[对象](#_Toc28775_WPSOffice_Level2) [7](#_Toc28775_WPSOffice_Level2)

[函数](#_Toc17404_WPSOffice_Level2) [8](#_Toc17404_WPSOffice_Level2)

[Class](#_Toc549_WPSOffice_Level2) [9](#_Toc549_WPSOffice_Level2)

[ES6模块](#_Toc19616_WPSOffice_Level2) [10](#_Toc19616_WPSOffice_Level2)

[异步处理](#_Toc3897_WPSOffice_Level2) [11](#_Toc3897_WPSOffice_Level2)

[Rxjs](#_Toc3714_WPSOffice_Level2) [13](#_Toc3714_WPSOffice_Level2)

**[Vue相关说明](#_Toc21773_WPSOffice_Level1)** **[14](#_Toc21773_WPSOffice_Level1)**

[基于模块开发](#_Toc29659_WPSOffice_Level2) [14](#_Toc29659_WPSOffice_Level2)

[Vue项目构建](#_Toc12510_WPSOffice_Level2) [14](#_Toc12510_WPSOffice_Level2)

[Vue 组件命名](#_Toc11931_WPSOffice_Level2) [15](#_Toc11931_WPSOffice_Level2)

[组件表达式简单化](#_Toc25414_WPSOffice_Level2) [16](#_Toc25414_WPSOffice_Level2)

[验证组件的 props](#_Toc5908_WPSOffice_Level2) [16](#_Toc5908_WPSOffice_Level2)

[组件结构化](#_Toc13289_WPSOffice_Level2) [17](#_Toc13289_WPSOffice_Level2)

[Vue组件模版](#_Toc27544_WPSOffice_Level2) [18](#_Toc27544_WPSOffice_Level2)

[对组件文件进行代码校验](#_Toc13620_WPSOffice_Level2) [19](#_Toc13620_WPSOffice_Level2)

[组件属性、方法注释描述](#_Toc29910_WPSOffice_Level2) [19](#_Toc29910_WPSOffice_Level2)

[使用好的IDE辅助开发](#_Toc21824_WPSOffice_Level2) [19](#_Toc21824_WPSOffice_Level2)

[提供组件 API 文档](#_Toc14337_WPSOffice_Level2) [20](#_Toc14337_WPSOffice_Level2)

**[业务代码要求](#_Toc10107_WPSOffice_Level1)** **[21](#_Toc10107_WPSOffice_Level1)**

[封装桶](#_Toc7660_WPSOffice_Level2) [21](#_Toc7660_WPSOffice_Level2)

[页面跳转](#_Toc13076_WPSOffice_Level2) [21](#_Toc13076_WPSOffice_Level2)

[请求接口](#_Toc2710_WPSOffice_Level2) [21](#_Toc2710_WPSOffice_Level2)

[状态管理](#_Toc15376_WPSOffice_Level2) [22](#_Toc15376_WPSOffice_Level2)

[页面组件](#_Toc18640_WPSOffice_Level2) [22](#_Toc18640_WPSOffice_Level2)

[SCSS样式](#_Toc9079_WPSOffice_Level2) [23](#_Toc9079_WPSOffice_Level2)

# 补充常用语法

## **声明和解构**

ES6中 新增了let、const 声明方式

* let声明

let 与 var 声明变量的写法类似，不同于 var，let声明的变量只在块级作用域内有效。

<!-- 示例 -->

{

let a = 10;

var b = 1;

}

console.log(a) // 输出：ReferenceError: a is not defined.

console.log(b) // 输出：1

var命令会发生“变量提升”现象，即变量可以在声明之前使用，值为undefined。let命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。

同时在相同的作用域中，let不能被重复声明。

<!-- 示例 -->

console.log(foo); // 输出undefined

var foo = 2;

console.log(bar); // 报错ReferenceError

let bar = 2;

let bar = 3; // 报错，let声明不能在一个作用域中多次声明

* const声明

const声明与let声明相似，但const声明的是常量，常量**不能被重新赋值**，否则将编译错误。但如果定义的常量是对象，对象的属性值是可以被重新赋值的。

<!-- 示例 -->

const CAT\_LIVES\_NUM = 9;

const kitty = {

name: 'Aurora',

numLives: CAT\_LIVES\_NUM

};

// 错误

kitty = {

name: 'Danielle',

numLives: CAT\_LIVES\_NUM

};

kitty.name = 'Kitty'; // 正确

kitty.numLives--; // 正确

* 解构

解构是ES6的一个重要特性，所谓解构，就是将**声明的一组变量与相同结构的数组或者对象的元素数值一一对应，并将变量对应的元素进行赋值**。解构可以帮助开发者轻松地实现多返回值的场景，这样不仅写法简洁，也会增强代码的可读性。

ES6中常用的有数组解构和对象解构。

**数组解构**

数组解构是最简单的解构类型

<!-- 示例 -->

let input = [1, 2];

let [first, second] = input;

console.log(first); // 输出：1

console.log(second); // 输出：2

<!-- 也可作用于已声明的变量 -->

[first, second] = [second, first]; // 变量交换

<!-- 或作用于函数参数 -->

function f([first, second]) {

console.log(first + second);

}

f([1, 2]); // 输出：3

**对象解构**

对象解构有趣的地方是一些原本需要多行编写的代码，用对象解构的方式编写一行就能完成，代码很简洁，可读性强。

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，**变量必须与属性同名，才能取到正确的值**。

<!-- 示例 -->

let info = {x: 0, y: 10, width: 15, height: 20};

let {x, width, offset} = info;

console.log(x, width); // 输出：0 15

console.log(offset); // 输出：undefined

let {x, width, offset} = info;

<!-- 其实是下面形式的缩写 -->

let {x: x, width: width, offset: offset} = info;

也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。

对象解构也可以指定默认值。

let {x, y = 5} = {x: 1};

console.log(x); // 输出：1

console.log(y); // 输出：5

对象解构常作用于函数参数。

<!-- 函数对象作为入参，同时为变量设置缺省值 -->

initOptions({

intervalAngel = 0,

levelSpacing = 2,

cornerRadius = 3,

iconSize = 35,

innerRadius = 60,

radiusStep = 70,

arcLength = 90,

secondLength = null,

fullCircle = false

} = {}) {

// 执行代码

}

解构虽然方便，但使用时还得多注意，特别时深层嵌套的场景，是比较容易出错的。

## **类型**

Javascript中数据类型分为基本类型和引用类型。

基本类型：Undefined、Null、Boolean、Number和String。

引用类型：Array、Object等除了基本数据类型外的类型。

基本类型和引用类型区别主要表现在复制变量值和传递参数时：

**复制变量值**

如果从一个变量向另一个变量**复制基本类型的值**，会在变量对象上创建一个新值，然后把该值复制到为新变量分配的位置上。

let a = 5;

let b = a;

<!-- a、b是还是相互独立的 -->

如果从一个变量向另一个变量**复制引用类型的值**，同样也会将存储在变量对象中的值复制一份放到为新变量分配的空间中。不同的是，这个值的副本实际上是一个指针，而这个指针指向存储在堆中的一个对象。**复制操作结束后，两个变量实际上将引用一个对象。因此改变其中的一个变量，就会影响另一个变量。**

let a = ｛num: 5｝;

let b = a; b.num = 10;

console.log(a.num); // 输出： 10

**传递参数**

javascript中所有函数的参数都是按值传递的。把函数外部的值复制给函数内部参数，就和把值从一个变量复制到另一个变量一样。

也就是说参数中传入对象，就是传入对象的引用，在函数内部对参数对象进行设置会同时影响对象内容。

function f(data) {

data.num += 10;

}

let a = {num: 5};

f(a);

console.log(a.num); // 输出：15

总而言之，除了基本类型，所有的变量在传递过程中，传递的都是引用。如果不想改变传入对象的值，必须对对象进行深拷贝赋值。请务必注意关注数据处理过程中数据对象的流向及变化。

## **字符串**

* 模板字符串

模板字符串（template string）是增强版的字符串，用反引号（`）标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

<!-- 示例 -->

function authorize(user, action) {

if (!user.hasPrivilege(action)) {

throw new Error(

// 传统写法为

// 'User '

// + user.name

// + ' is not authorized to do '

// + action

// + '.'

`User ${user.name} is not authorized to do ${action}.`);

}

}

## **Iterator遍历器**

它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

Iterator 接口主要供for...of消费。

原生具备 Iterator 接口的数据结构如下：

* Array
* Map
* Set
* String
* TypedArray
* 函数的 arguments 对象
* NodeList 对象

## **数组**

* 数组扩展运算符

扩展运算符（spread）是三个点（...）。它好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列。

<!-- 示例 -->

console.log(1, ...[2, 3, 4], 5)

// 1 2 3 4 5

<!-- 与解构赋值结合 -->

const [first, ...rest] = [1, 2, 3, 4, 5];

first // 1

rest // [2, 3, 4, 5]

<!-- 正确识别四个字节的 Unicode 字符 -->

'x\uD83D\uDE80y'.length // 4

[...'x\uD83D\uDE80y'].length // 3

* 常用新增方法

**Array.prototype.find()**

数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined。

<!-- 示例 -->

let data = [{name: 'Jone'}, {name: 'Smith'}, {name: 'Luise'}];

console.log(data.find((item) => item.name === 'Jone')); // 输出：{name: 'Jone'}

**Array.prototype.findIndex()**

数组实例的findIndex方法的用法与find方法非常类似，返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回-1。

<!-- 示例 -->

let data = [{name: 'Jone'}, {name: 'Smith'}, {name: 'Luise'}];

console.log(data.findIndex((item) => item.name === 'Jone')); // 输出：0

**Array.prototype.includes()**

方法返回一个布尔值，表示某个数组是否包含给定的值，与字符串的includes方法类似。

<!-- 示例 -->

[1, 2, 3].includes(2) // true

[1, 2, 3].includes(4) // false

[1, 2, NaN].includes(NaN) // true

## **对象**

* 对象扩展运算符与解构赋值

对象的扩展运算符（...）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中。

<!-- 示例 -->

let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };

console.log(x) // 1

console.log(y) // 2

console.log(z) // { a: 3, b: 4 }

<!-- 解构赋值要求等号右边是一个对象 -->

let { x, y, ...z } = null; // 运行时错误

let { x, y, ...z } = undefined; // 运行时错误

<!-- 解构赋值必须是最后一个参数 -->

let { ...x, y, z } = someObject; // 句法错误

let { x, ...y, ...z } = someObject; // 句法错误

<!-- 解构赋值必须是最后一个参数扩展运算符可以用于合并两个对象-->

let ab = { ...a, ...b };

// 等同于

let ab = Object.assign({}, a, b);

注意，解构赋值的拷贝是浅拷贝，即如果一个键的值是复合类型的值（数组、对象、函数）、那么解构赋值拷贝的是这个值的引用，而不是这个值的副本。

* 常用新增方法

**Object.assign()**

Object.assign方法用于对象的合并，将源对象（source）的所有**可枚举属性**，复制到目标对象（target）。

<!-- 示例 -->

const target = { a: 1 };

const source1 = { b: 2 };

const source2 = { c: 3 };

Object.assign(target, source1, source2);

target // {a:1, b:2, c:3}

注意，如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性

Object.assign拷贝的属性是有限制的，只拷贝源对象的自身属性（不拷贝继承属性），也不拷贝不可枚举的属性（enumerable: false）。

Object.assign方法实行的是浅拷贝，而不是深拷贝。

**Object.getOwnPropertyDescriptors()**

返回指定对象所有自身属性（非继承属性）的描述对象。

该方法的引入目的，主要是为了解决Object.assign()无法正确拷贝get属性和set属性的问题。

<!-- 示例 -->

const shallowMerge = (target, source) => Object.defineProperties(

target,

Object.getOwnPropertyDescriptors(source)

);

**Object.keys()，Object.values()，Object.entries()**

它们都返回一个遍历器对象，可以用for...of循环进行遍历，唯一的区别是keys()是对键名的遍历、values()是对键值的遍历，entries()是对键值对的遍历。成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性。

<!-- 示例 -->

let obj = {name: 'Jone', age: 16};

console.log(Object.keys(obj)); // 输出：['name', 'age']

console.log(Object.values(obj)); // 输出：['Jone', 16]

console.log(Object.entries(obj));// 输出：[['name', 'Jone'], ['age', 16]]

<!-- 遍历对象 -->

for (let [key, value] of Object.entries(obj)) {

console.log(key, value);

}

// name Jone

// age 16

## **函数**

* 默认参数

ES6语法中支持初始化默认参数。如果函数的某个参数设置了默认值，当该函数被调用时，如果没有给这个参数传值或者传的值为空时，这个参数就是设置的默认值。

function f(x = 4, y) {

Return x > y ? x : y;

}

let result = f(null, 1); // 运行正常，返回4;

* 剩余参数

当需要同时操作多个参数，或者并不知道会有多少参数传递进来时，可以在参数中使用扩展运算符，将参数放到一个变量中。

function f(x, ...args) {

console.log(args);

}

f(1, 2, 4, 5, 4,); // 输出：[2, 4, 5, 4]

* 箭头函数

JavaScript里，this的值在函数被调用的时候才会指定。而使用箭头函数能保存函数创建时的 this值，而不是调用时的值。

箭头函数有几个使用注意点：

（1）函数体内的this对象是固定的，就是定义时所在的对象，而不是使用时所在的对象。

（2）不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。

（3）不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。

（4）不可以使用yield命令，因此箭头函数不能用作 Generator 函数。

this指向的固定化，并不是因为箭头函数内部有绑定this的机制，实际原因是箭头函数根本没有自己的this，导致内部的this就是外层代码块的this。正是因为它没有this，所以也就不能用作构造函数。

箭头函数可以让this指向固定化，这种特性很有利于封装回调函数。

## **Class**

ES6中引入class概念，通过class关键字可以定义类。ES6 的class可以看作是一个语法糖，新的class写法让对象原型的写法更加清晰、更像面向对象编程的语法。

<!-- 通常写法 -->

function Point(x, y) {

this.x = x;

this.y = y;

}

Point.prototype.toString = function () {

return '(' + this.x + ', ' + this.y + ')';

};

<!-- class写法 -->

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

toString() {

return `(${this.x}, ${this.y})`;

}

}

* 修饰符

在当前class语法中没有public、private、protected。只有static用于声明静态方法、属性。

静态属性指的是 Class 本身的属性，即Class.propName，而不是定义在实例对象（this）上的属性。

因为没有提供声明私有方法属性的操作，所有想要定义的私有方法属性加上下划线\_前缀来标明这是私有的。

* Class继承

Class 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。

<!-- 示例 -->

class Point { /\* ... \*/ }

class ColorPoint extends Point {

constructor(...arg) {

super(...arg);

}

}

子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类自己的this对象，必须先通过父类的构造函数完成塑造，得到与父类同样的实例属性和方法，然后再对其进行加工，加上子类自己的实例属性和方法。如果不调用super方法，子类就得不到this对象。

## **ES6模块**

模块在其自身的作用域里执行，而不是在全局作用域里。这意味着定义在一个模块里的变量，函数，类等等在模块外部是不可见的，除非你明确地使用export形式之一导出它们。 相反，如果想使用其它模块导出的变量、函数、类、接口等的时候，你必须要导入它们，可以使用 import形式之一。

模块是自声明的；两个模块之间的关系是通过在文件级别上使用import和export建立的。

* 模块导入导出

**导出**

任何声明（比如变量，函数，类，类型别名或接口）都能够通过添加export关键字来导出。

class ZipCodeValidator {

isAcceptable(s) {

return s.length === 5 && numberRegexp.test(s);

}

}

export { ZipCodeValidator };

export { ZipCodeValidator as mainValidator };

或者一个模块可以包裹多个模块，并把他们导出的内容联合在一起通过语法：export \* from "module"。

export \* from "./StringValidator";

export \* from "./LettersOnlyValidator";

export \* from "./ZipCodeValidator";

**导入**

模块的导入操作与导出一样简单。 可以使用以下 import形式之一来导入其它模块中的导出内容。

<!-- 导入一个模块中的某个导出内容 -->

import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();

<!-- 对导入内容重命名 -->

import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";

let myValidator = new ZCV();

<!-- 将整个模块导入到一个变量，并通过它来访问模块的导出部分 -->

import \* as validator from "./ZipCodeValidator";

let myValidator = new validator.ZipCodeValidator();

**默认导出**

每个模块都可以有一个default导出。 默认导出使用 default关键字标记；并且一个模块只能够有一个default导出。 需要使用一种特殊的导入形式来导入 default导出。

<!-- ZipCodeValidator.js -->

export default class ZipCodeValidator {

static numberRegexp = /^[0-9]+$/;

isAcceptable(s: string) {

return s.length === 5 && ZipCodeValidator.numberRegexp.test(s);

}

}

<!-- Test.ts -->

import validator from "./ZipCodeValidator";

let myValidator = new validator();

## **异步处理**

* Promise

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。

Promise创建便开始执行。

Promise对象代表一个异步操作，有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。

Promise对象的状态改变，只有两种可能：从pending变为fulfilled和从pending变为rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 resolved（已定型）。**如果改变已经发生了，再对Promise对象添加回调函数，也会立即得到这个结果。**

<!-- 示例 -->

let a = 5;

let promise = new Promise((resolve, reject) => {

resolve(++a);

});

promise.then(data => console.log(data)); // 输出：6

setTimeout(() => {

promise.then(data => console.log(data)); // 输出：6

}, 100);

**Promise.prototype.then()**

它的作用是为 Promise 实例添加状态改变时的回调函数。前面说过，then方法的第一个参数是resolved状态的回调函数，第二个参数（可选）是rejected状态的回调函数。

**Promise.prototype.catch()**

用于指定发生错误时的回调函数。

**Promise.prototype.finally()**

finally方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。

* async/await

ES2017 标准引入了 async 函数，使得异步操作变得更加方便，同时async函数的返回值是 Promise 对象。async 函数常用于一组异步操作。

async表示函数里有异步操作，await表示紧跟在后面的表达式需要等待结果。

<!-- 示例 -->

const asyncReadFile = async function () {

const f1 = await readFile('/etc/fstab');

const f2 = await readFile('/etc/shells');

console.log(f1.toString());

console.log(f2.toString());

};

如果await后面的异步操作出错，那么等同于async函数返回的 Promise 对象被reject。防止出错的方法，最好把await命令放在try...catch代码块之中。

<!-- 示例 -->

async function myFunction() {

try {

await somethingThatReturnsAPromise();

} catch (err) {

console.log(err);

}

}

如果有多个await命令，可以统一放在try...catch结构中。

## **Rxjs**

ReactiveX（Reactive Extensions），它是微软开发并维护的的基于Reactive Programming范式实现的一套工具库集合。Rx系列结合了观察者模式、迭代器模式和函数式编程，经过多年的验证，目前已成为业界流行的响应式编程优秀实践了。Rxjs就是其在JavaScript层面上的实现。

在业务操作中，能用Promise的场景Rxjs都适用。Promise只能针对单一的异步事件进行resolve()操作，而在Observable中，不仅能处理一个单一的异步事件，而且能以流的形式响应多个异步事件。

Rxjs用的比较多的典型场景有通过发布订阅实现组件间通讯、通过过滤操作符进行函数节流、嵌套请求的扁平处理等等...

[https://cn.rx.js.org/manual/overview.html#h11](https://cn.rx.js.org/manual/overview.html" \l "h11)

# Vue相关说明

## **基于模块开发**

始终基于模块的方式来构建应用，每一个子模块只做一件事情，一个模块是应用程序中独立的一个部分。

**每一个 Vue 组件首先必须专注于解决一个单一的问题，独立的、可复用的、微小的和可测试的。**

如果我们的组件做了太多的事或是变得臃肿，请**将其拆分成更小的组件并保持单一的原则**，也请保证组件可独立的运行。

## **Vue项目构建**

良好的项目结构应该是易于扩展维护的，各个功能、模块负责的内容应该保持单一独立。

Vue提供了便利的脚手架工具（vue-cli），通过它可以快速完成项目生成配置。业务开发中，我们需要对生成的项目src目录结构做出部分调整：

project

│ package.json

│ vue.config.js

├─public

└─src

│ App.vue

│ main.js

│ plugins.js // 负责导入工具及类库

│ polyfills.js // 负责导入需要的浏览器兼容补丁

│ styles.scss // 全局的样式文件

├─app

├─assets // 静态资源管理

├─constants // 常量管理

├─common

└─theme

app文件夹负责具体业务内容，基本结构如下：

─app

├─api // 负责接口管理

│ index.js // 导出挂载至vue.prototype.$api以便通过this.$api调用

│ interceptors.js // 请求拦截管理（统一错误处理，请求头设置等等）

├─router // 负责页面路由跳转

│ index.js

│ interceptors.js

├─store // 负责应用状态管理

│ index.js // 导出文件

│ store.js // 全局应用状态管理，有需要再划分子模块状态管理

└─views // 视图组件模块管理

│ view.route.js

├─module-a

└─module-b

common文件夹负责应用公用的组件、类、方法等

─common

├─components // 公用组件

├─directives // 公用指令

├─filters // 公用过滤器

├─services // 公用服务

└─utils // 公用方法、类

theme文件夹负责应用公用样式、定义的样式变量、方法、等等

─theme

│ \_function.scss // 定义的样式方法

│ \_index.scss // 定义的样式变量、方法统一导出

│ \_mixin.scss // 应用样式混入

│ \_variables.scss // 应用样式变量

└─element-ui // 用的UI组件是element的

\_element-ui.scss // element组件全局样式变更

\_element-variables.scss // element样式变量调整

## **Vue 组件命名**

组件的命名需遵从以下原则：

* 有意义的: 不过于具体，也不过于抽象
* 简短: 2 到 3 个单词
* 具有可读性: 以便于沟通交流

同时还需要注意：

* 必须符合自定义元素规范: 使用连字符分隔单词，切勿使用保留字。
* 全局组件使用app- 前缀作为命名空间。

<!-- 推荐 -->

<app-header></app-header>

<app-card></app-card>

<range-slider></range-slider>

## **组件表达式简单化**

Vue.js 的表达式是 100% 的 Javascript 表达式。这使得其功能性很强大，但也带来潜在的复杂性。因此，我们应该尽量保持表达式的简单化。

如果我们发现写了太多复杂并难以阅读的行内表达式，那么可以使用 method 或是 computed 属性来替代其功能。

为什么？

* 复杂的行内表达式难以阅读。
* 行内表达式是不能够通用的，这可能会导致重复编码的问题。
* IDE 基本上不能识别行内表达式语法，所以使用行内表达式 IDE 不能提供自动补全和语法校验功能。

## **验证组件的 props**

在 Vue.js 中，组件的 props 即 API，一个稳定并可预测的 API 会使得你的组件更容易被其他开发者使用。

组件 props 通过自定义标签的属性来传递。属性的值可以是 Vue.js 字符串(:attr="value" 或 v-bind:attr="value")或是不传。你需要保证组件的 props 能应对不同的情况。

为什么？

验证组件 props 可以保证你的组件永远是可用的（防御性编程）。即使其他开发者并未按照我们预想的方法使用时也不会出错。

怎么做？

* 提供默认值。
* 使用 type 属性校验类型。
* 使用 props 之前先检查该 prop 是否存在。

<!-- 推荐 -->

props: {

max: {

type: Number, // 这里添加了数字类型的校验

default() { return 10; },

},

min: {

type: Number,

default() { return 0; },

}

}

## **组件结构化**

按照一定的结构组织，使得组件便于理解。

导出一个清晰、组织有序的组件，使得代码易于阅读和理解。同时也便于标准化。

为什么？

* 按功能或性质对 properties、data、computed、watches、methods 中的属性或方法进行分类，使得这些对象内的属性便于查找 。
* 合理组织，使得组件易于阅读。（name; extends; props, data 和 computed; components; watch 和 methods; lifecycle methods 等）。
* 使用 name 属性。借助于 vue devtools 可以让你更方便的测试。
* 使用组件名作为样式作用域空间。合理的 CSS 结构，如 BEM。

怎么做？

<template>

<div class="page"></div>

</template>

<script>

export default {

name: 'page',

// 资源选项

components: {},

directives: {},

filters: {},

// 其他选项

provide: function() {

return {}

},

inject: [],

// 数据、方法选项

props: {},

data: function() {

return {};

},

computed: {},

watch: {},

methods: {},

// 生命周期钩子函数，按周期位置顺序书写

created: function () {},

mounted: function () {},

destroyed: function () {}

};

</script>

<style scoped lang="scss">

/\*使用组件名作为样式作用域空间\*/

.page {

}

</style>

## **Vue组件模版**

统一的模板格式更有利于形成团队的书写阅读习惯。

<!-- 不推荐 -->

<template>

<div class="page">

<li

v-for="user in activeUsers"

:key="user.id"

>

{{ user.name }}

</li>

</div>

</template>

<!-- 推荐 -->

<template>

<div class="page">

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

{{ user.name }}

</li>

</div>

</template>

<template>

<div class="page">

<li v-for="user in activeUsers"

:key="user.id">

{{ user.name }}

</li>

</div>

</template>

## **对组件文件进行代码校验**

代码校验可以保持代码的统一性以及追踪语法错误。我们可以通过 vue-cli 来开始项目，vue-cli 默认会开启代码校验功能。

为什么？

* 保证所有的开发者使用同样的编码规范。
* 更早的感知到语法错误。

怎么做？

* 为了校验工具能够校验 \*.vue文件，你需要将代码编写在 <script>标签中，并使组件表达式简单化，因为校验工具无法理解行内表达式，配置校验工具可以访问全局变量 vue 和组件的 props。
* 在每次提交代码前运行npm run lint进行代码格式校验

## **组件属性、方法注释描述**

为什么？

* 没有为什么。

怎么做？

* 组件属性上加上注释进行描述。
* 组件方法上加上注释进行功能、参数描述。

## **使用好的IDE辅助开发**

推荐使用webstorm进行代码开发。

为什么？

* webstorm集成了海量插件功能，开箱即用。
* 自动加载项目中配置的Eslint、Jslint配置文件，对代码风格进行自动校验，高亮显示问题代码，统一团队成员编码风格。
* 代码联想、快速定位、支持代码检查和快速修复、支持插件扩展、支持本地代码回滚...

怎么做？

* 百度webstorm上官网进行下载安装，然后上淘宝花3元买一年份激活码。

## **提供组件 API 文档**

使用 Vue.js 组件的过程中会创建 Vue 组件实例，这个实例是通过自定义属性配置的。为了便于其他开发者使用该组件，对于这些自定义属性即组件API应该在 README.md 文件中进行说明。

// TODO:待明确具体格式要求，需先对文档生成工具进行技术选型

# 业务代码要求

## **封装桶**

简而言之就是通过该模块文件夹下的index.js文件，导出该模块需要暴露的内容，其它地方如果需要导入该模块所导出的内容，可以直接从index.js父目录路径获取。

<!-- 目录 -->

module

│ index.js

├─sample

│ sample.vue // 一个vue文件

└─sample2

main.js // main.js导出类Point

<!-- index.js -->

export \* from './sample2/main.js';

export {default as Sample} from './sample/sample.vue';

<!-- 其他文件.js -->

import {Sample, Point} from './module';

## **页面跳转**

在代码中控制页面跳转，统一通过在路由配置文件中设置的name进行导航。

<!-- 路由配置文件 -->

{

path: 'service-arrange-management',

name: 'ServiceArrangeManagement',

component: ServiceArrangeManagement

}

<!-- 业务代码 -->

router.push({name: ServiceArrangeManagement});

## **请求接口**

请求方法的编码请在接口管理文件夹对应模块的对象中通过Axios请求工具实现。所有业务性中间处理操作均在方法中进行，页面组件调用只应关注结果。

<!-- 示例方法 -->

// 服务编排数据列表查询接口

getServiceList: function (params) {

let url = '/sce-service/appService/list';

let headers = {

// 'Content-Type': 'application/json;charset=UTF-8'

};

return Axios.get(url, {

params: {

pageNum: params.pageNum || 1, // 可以设置默认值

pageSize: params.pageSize || 20,

catalog: params.catalog || undefined,

keyWords: params.keyWords || undefined

},

headers: headers

}).then(response => {

// 如果要使用的数据需要处理，在这里处理以后再返回。

return {data: response.data.rows, total: response.data.amount}

});

}

## **状态管理**

应用状态管理模块应根据需要进行创建，同时各模块的状态应该创建对应模块的状态管理文件来进行管理。

全局的状态管理中，只能有用于应用全局状态相关的内容。

## **页面组件**

每个页面组件使用与其同名的文件夹包裹，如果.vue文件中样式内容过长，应当移出单独建立.scss文件。

<!-- 路由配置文件 -->

service-arrange

│ service-arrange.vue

│ service-arrange.scss

├─arrange-content

├─arrange-menu

└─arrange-sidebar

对于同一模块的页面统一使用封装桶导出，路由配置文件中加载。

<!-- index.js -->

export const ServiceArrange = () => import('./service-arrange/service-arrange.vue');

export const ServiceArrangeManagement = () => import('./service-management/service-management.vue');

## **SCSS样式**

采用BEM风格命名？