前端规范

M站前端开发手册 v1.0.0

目录

**[前端开发手册](#_Toc12786_WPSOffice_Level1)** **[4](#_Toc12786_WPSOffice_Level1)**

**[一、 项目规范](#_Toc14537_WPSOffice_Level1)** **[4](#_Toc14537_WPSOffice_Level1)**

[（一） 开始](#_Toc14537_WPSOffice_Level2) [4](#_Toc14537_WPSOffice_Level2)

[（二） 目录结构](#_Toc25558_WPSOffice_Level2) [4](#_Toc25558_WPSOffice_Level2)

[（三） 别名](#_Toc30305_WPSOffice_Level2) [10](#_Toc30305_WPSOffice_Level2)

[（四） 路由](#_Toc30746_WPSOffice_Level2) [10](#_Toc30746_WPSOffice_Level2)

[（五） Context](#_Toc10645_WPSOffice_Level2) [15](#_Toc10645_WPSOffice_Level2)

[（六） 页面](#_Toc2339_WPSOffice_Level2) [17](#_Toc2339_WPSOffice_Level2)

[（七） 组件](#_Toc1705_WPSOffice_Level2) [24](#_Toc1705_WPSOffice_Level2)

[（八） 插件](#_Toc4345_WPSOffice_Level2) [26](#_Toc4345_WPSOffice_Level2)

[（九） Vuex状态树](#_Toc8976_WPSOffice_Level2) [33](#_Toc8976_WPSOffice_Level2)

[（十） 开发与部署](#_Toc27811_WPSOffice_Level2) [34](#_Toc27811_WPSOffice_Level2)

**[二、 JavaScript](#_Toc25558_WPSOffice_Level1)** **[37](#_Toc25558_WPSOffice_Level1)**

[（一） 命名](#_Toc6035_WPSOffice_Level2) [37](#_Toc6035_WPSOffice_Level2)

[（二） 变量属性](#_Toc10812_WPSOffice_Level2) [38](#_Toc10812_WPSOffice_Level2)

[（三） 字符串](#_Toc30855_WPSOffice_Level2) [40](#_Toc30855_WPSOffice_Level2)

[（四） 运算符](#_Toc30912_WPSOffice_Level2) [40](#_Toc30912_WPSOffice_Level2)

[（五） 对象](#_Toc31986_WPSOffice_Level2) [41](#_Toc31986_WPSOffice_Level2)

[（六） 数组](#_Toc20279_WPSOffice_Level2) [42](#_Toc20279_WPSOffice_Level2)

[（七） 解构](#_Toc13194_WPSOffice_Level2) [43](#_Toc13194_WPSOffice_Level2)

[（八） 函数](#_Toc19998_WPSOffice_Level2) [44](#_Toc19998_WPSOffice_Level2)

[（九） 类](#_Toc23791_WPSOffice_Level2) [47](#_Toc23791_WPSOffice_Level2)

[（十） 模块](#_Toc13636_WPSOffice_Level2) [48](#_Toc13636_WPSOffice_Level2)

[（十一） 迭代器](#_Toc31900_WPSOffice_Level2) [49](#_Toc31900_WPSOffice_Level2)

[（十二） 异步处理](#_Toc22907_WPSOffice_Level2) [49](#_Toc22907_WPSOffice_Level2)

**[三、 Html](#_Toc30305_WPSOffice_Level1)** **[51](#_Toc30305_WPSOffice_Level1)**

[（一） 语法](#_Toc32344_WPSOffice_Level2) [51](#_Toc32344_WPSOffice_Level2)

[（二） Meta](#_Toc8387_WPSOffice_Level2) [53](#_Toc8387_WPSOffice_Level2)

[（三） H5语义](#_Toc31790_WPSOffice_Level2) [54](#_Toc31790_WPSOffice_Level2)

**[四、 Css](#_Toc30746_WPSOffice_Level1)** **[55](#_Toc30746_WPSOffice_Level1)**

[1. 代码](#_Toc218_WPSOffice_Level2) [55](#_Toc218_WPSOffice_Level2)

[2. 顺序](#_Toc20795_WPSOffice_Level2) [55](#_Toc20795_WPSOffice_Level2)

[3. 属性](#_Toc7006_WPSOffice_Level2) [55](#_Toc7006_WPSOffice_Level2)

[4. 声明](#_Toc18383_WPSOffice_Level2) [56](#_Toc18383_WPSOffice_Level2)

[5. 命名](#_Toc11232_WPSOffice_Level2) [56](#_Toc11232_WPSOffice_Level2)

[6. 注释](#_Toc26036_WPSOffice_Level2) [57](#_Toc26036_WPSOffice_Level2)

**[五、 异常日志](#_Toc10645_WPSOffice_Level1)** **[58](#_Toc10645_WPSOffice_Level1)**

[（一） 目的](#_Toc24466_WPSOffice_Level2) [58](#_Toc24466_WPSOffice_Level2)

[（二） 资源](#_Toc12189_WPSOffice_Level2) [58](#_Toc12189_WPSOffice_Level2)

[（三） 代码](#_Toc11134_WPSOffice_Level2) [58](#_Toc11134_WPSOffice_Level2)

**[六、 单元测试](#_Toc2339_WPSOffice_Level1)** **[58](#_Toc2339_WPSOffice_Level1)**

[（一） 目的](#_Toc17667_WPSOffice_Level2) [58](#_Toc17667_WPSOffice_Level2)

[（二） 覆盖率](#_Toc7530_WPSOffice_Level2) [58](#_Toc7530_WPSOffice_Level2)

[（三） 匹配器](#_Toc32767_WPSOffice_Level2) [59](#_Toc32767_WPSOffice_Level2)

[（四） 异步代码](#_Toc17616_WPSOffice_Level2) [63](#_Toc17616_WPSOffice_Level2)

[（五） 模拟器](#_Toc24801_WPSOffice_Level2) [67](#_Toc24801_WPSOffice_Level2)

[（六） 全局函数](#_Toc31886_WPSOffice_Level2) [72](#_Toc31886_WPSOffice_Level2)

[（七） Expect](#_Toc2966_WPSOffice_Level2) [74](#_Toc2966_WPSOffice_Level2)

[（八） Vue DOM和事件模拟](#_Toc25413_WPSOffice_Level2) [79](#_Toc25413_WPSOffice_Level2)

**[七、 Eslint 规范](#_Toc1705_WPSOffice_Level1)** **[80](#_Toc1705_WPSOffice_Level1)**

**[八、 网络安全](#_Toc4345_WPSOffice_Level1)** **[81](#_Toc4345_WPSOffice_Level1)**

前端开发手册

|  |  |  |  |
| --- | --- | --- | --- |
| **版本号** | **制定团队** | **更新日期** | **备注** |
| 1.0.0 | 铁甲二手机前端 | 2019.6.26 | 增加设计规范 |

# 项目规范

## 开始

1. 从svn://检出代码
2. 进入webApp项目目录。  
   安装依赖包：  
   $ npm install
3. 运行开发环境：  
   $ npm run dev   
   默认端口号是3000，可以在util/serverConfig.js文件中修改port端口。尽量不要修改，如果修改了，不要提交该文件。

## 目录结构

webApp

|----.nuxt

|----api

|----assets

|----components

|----layouts

|----middleware

|----node\_modules

|----pages

|----plugins

|----server

|----static

|----store

|----test

|----util

|----nuxt.config.js

|----package.json

1. 【server】文件夹下的index.js是项目的运行的起始文件。服务器端使用的是基于Node平台的express Web开发框架。Nuxt.js 作为express的插件使用。Nuxt.js分为开发模式和生产模式，代码会自行判断不需要做修改。该文件中不能使用es6的import引用，需使用require引用。
2. 【layouts】文件夹是创建自定义的布局。默认有两个文件default.vue和error.vue。  
   default.vue是默认布局文件，可以通过修改该页面来扩展默认布局。  
   error.vue是错误页面文件,这个布局文件不需要包含 <nuxt/> 标签。你可以把这个布局文件当成是显示应用错误（404，500等）的组件。  
   layouts下也可以创建自定义布局，比如：新建tmp.vue 来实现自定义布局，只需要在pages的页面中的posts.vue的layout属性中指定’tmp’.  
   假设我们要创建一个 模板布局 并将其保存到layouts/tmp.vue:  
   <template>   
    <div>  
    <div>我的模板导航栏在这里</div>   
    <nuxt/>  
    </div>   
   </template>  
   然后我们必须告诉页面 (即pages/posts.vue) 使用您的自定义布局：  
   <template> <!-- Your template --> </template>   
   <script>   
   export default {   
    layout: 'blog' // page component definitions   
   }   
   </script>  
   注：<nuxt/> 组件用于显示页面的主体内容。
3. 【pages】用于组织应用的路由及视图，Nuxt.js 依据 pages 目录结构自动生成 [vue-router](https://github.com/vuejs/vue-router) 模块的路由配置。这一点和vue不太一样。Vue中只有组件的概念，没有页面的概念，Nuxt.js是服务器端渲染页面的概念，需要在页面中配置head标签，并且有异步数据。Vue的路由指向的是组件，Nuxt.js的路由是pages文件夹下的文件，可以有层级多文件夹的模式。
4. 【components】是页面中的组件，是可复用的 Vue 实例。通过 Prop 向子组件传递数据，是单向数据流模式。有自己的生命周期，和vue单页开发方式一样。Nuxt.js 不会扩展增强该目录下 Vue.js 组件，即这些组件不会像页面组件那样有 asyncData 方法的特性。
5. 【store】文件夹中是vue中的状态管理模块。Nuxt.js 框架集成了 [Vuex 状态树](http://vuex.vuejs.org/) 的相关功能配置，在 store 目录下创建一个 index.js 文件可激活这些配置。每个文件相当于一个模块，Nuxt.js自动加载。这个和vue中store不太一样，vue是需要向根组件传递store对象的。
6. 【middleware】是中间件文件夹，在应用中的特定页面设置中间件。中间件允许您定义一个自定义函数运行在一个页面或一组页面渲染之前。  
   比如判断user页面是否已经登录，如果未登录则重定向到login页面,例子代码如下：  
   pages/user.vue  
   <template> <h1>Secret page</h1> </template>   
   <script>   
   export default {   
    middleware: 'authenticated'   
   }   
   </script>  
   middleware/authenticated.js  
   export default function ({ store, redirect }) { // If the user is not authenticated  
    if (!store.state.authenticated) {  
    return redirect('/login')  
    }   
   }
7. 【plugins】文件夹中是插件js，Nuxt.js允许您在运行Vue.js应用程序之前执行js插件。这在您需要使用自己的库或第三方模块时特别有用。  
   需要注意的是，在任何 Vue 组件的生命周期内，只有 beforeCreate 和 created 这两个方法会在 客户端和服务端被调用。其他生命周期函数仅在客户端被调用。
8. 【api】是页面调用接口服务器的http请求方法集合，按照业务模块来创建js文件。  
   比如用户模块：  
   user.js  
    import request from '../util/request';  
    //用户名密码登录  
    export function login(params){}  
    //手机号校验码登录  
    export function loginByMobile(params){}
9. 【assets】文件夹中包含的是用于组织未编译的静态资源如 LESS、SASS 或 JavaScript、images等资源文件。该文件夹下的文件会被webpack按照定义的规则处理。比如css/js压缩,小图片会被转化为base64格式等。
10. 【static】文件夹下是静态资源，此类文件不会被 Nuxt.js 调用 Webpack 进行构建编译处理。 服务器启动的时候，该目录下的文件会映射至应用的根路径 / 下。
11. 【.nuxt】是代码运行和存放编译后的代码文件夹。
12. 【test】是所有页面的单元测试和ui测试的js文件。文件名格式：index.spec.js，使用的是jest测试框架
13. 【util】是公共处理文件夹，所有的公共操作都可以在该文件夹下创建，js的公共处理模块合集。
14. 【nuxt.config.js】文件用于组织Nuxt.js 应用的个性化配置，以便覆盖默认配置。该文件中可以配置服务port、host；公共的页面head头；loading加载条；全局css、plugins、models、webpack等配置。
15. 【package.json】文件用于描述应用的依赖关系和对外暴露的脚本接口。Npm运行命令配置等。

## 别名

别名符号包括两种~和@，使用别名就省掉了../../这种多级的引用文件的方式。推荐大家使用。

1. 符号~ 或 @表示的是[src目录](https://zh.nuxtjs.org/api/configuration-srcdir)
2. 符号~~ 或 @@表示的是[根目录](https://zh.nuxtjs.org/api/configuration-rootdir)  
   默认情况下，src目录和根目录相同。提示: 在您的 vue 模板中, 如果你需要引入 assets 或者 static 目录, 使用 ~/assets/your\_image.png 和 ~/static/your\_image.png方式。

## 路由

1. 基础路由  
   假设 pages 的目录结构如下：  
   pages/   
   --| user/   
   -----| index.vue   
   -----| one.vue   
   --| index.vue  
   那么，Nuxt.js 自动生成的路由配置如下  
   router: {   
    routes: [   
    { name: 'index', path: '/', component: 'pages/index.vue' },  
    { name: 'user', path: '/user', component: 'pages/user/index.vue' },   
    { name: 'user-one', path: '/user/one', component: 'pages/user/one.vue' }   
    ]   
   }
2. 动态路由  
   在 Nuxt.js 里面定义带参数的动态路由，需要创建对应的以下划线作为前缀的 Vue 文件 或 目录  
   pages/

--| \_slug/

-----| comments.vue

-----| index.vue

--| users/

-----| \_id.vue

--| index.vue  
 Nuxt.js 生成对应的路由配置表为：  
 router: {

routes: [

{

name: 'index',

path: '/',

component: 'pages/index.vue'

},

{

name: 'users-id',

path: '/users/:id?',

component: 'pages/users/\_id.vue'

},

{

name: 'slug',

path: '/:slug',

component: 'pages/\_slug/index.vue'

},

{

name: 'slug-comments',

path: '/:slug/comments',

component: 'pages/\_slug/comments.vue'

}

]

}  
 你会发现名称为 users-id 的路由路径带有 :id? 参数，表示该 路由是可选的。如果你想将它设置为必选的路由，需要 在 users/\_id 目录内创建一个 index.vue 文件。

1. 路由参数校验  
   Nuxt.js 可以让你在动态路由组件中定义参数校验方法。  
   举个例子： pages/users/\_id.vue  
   export default {  
    validate ({ params }) { // 必须是number类型 return /^\d+$/.test(params.id)   
    }  
    }  
   如果校验方法返回的值不为 true或Promise中resolve 解析为false或抛出Error ， Nuxt.js 将自动加载显示 404 错误页面或 500 错误页面。
2. 嵌套路由  
   你可以通过 vue-router 的子路由创建 Nuxt.js 应用的嵌套路由。

创建内嵌子路由，你需要添加一个 Vue 文件，同时添加一个 与该文件同名的目录用来存放子视图组件。  
 注：别忘了在父组件(.vue文件) 内增加 <nuxt-child/> 用于 显示子视图内容。  
 假设文件结构如：  
 pages/

--| users/

-----| \_id.vue

-----| index.vue

--| users.vue

Nuxt.js 自动生成的路由配置如下：  
 router: {

routes: [

{

path: '/users',

component: 'pages/users.vue',

children: [

{

path: '',

component: 'pages/users/index.vue',

name: 'users'

},

{

path: ':id',

component: 'pages/users/\_id.vue',

name: 'users-id'

}

]

}

]

}

## Context

context 变量的可用属性一览:

| 属性字段 | 类型 | 可用 | 描述 |  |
| --- | --- | --- | --- | --- |
| app | Vue 根实例 | 客户端 & 服务端 | 包含所有插件的 Vue 根实例。例如：在使用 axios 的时候，你想获取 $axios 可以直接通过 context.app.$axios 来获取 |  |
| isClient | Boolean | 客户端 & 服务端 | 是否来自客户端渲染（废弃。请使用 process.client ） |  |
| isServer | Boolean | 客户端 & 服务端 | 是否来自服务端渲染（废弃。请使用 process.server ） |  |
| isStatic | Boolean | 客户端 & 服务端 | 是否来自 nuxt generate 静态化（预渲染）（废弃。请使用 process.static ） |  |
| isDev | Boolean | 客户端 & 服务端 | 是否是开发 dev 模式，在生产环境的数据缓存中用到 |  |
| isHMR | Boolean | 客户端 & 服务端 | 是否是通过模块热替换 webpack hot module replacement (仅在客户端以 dev 模式) |  |
| route | [Vue Router 路由](https://router.vuejs.org/zh/api/" \l "%E8%B7%AF%E7%94%B1%E5%AF%B9%E8%B1%A1%E5%B1%9E%E6%80%A7) | 客户端 & 服务端 | Vue Router 路由实例 |  |
| store | [Vuex 数据](https://vuex.vuejs.org/zh/api/) | 客户端 & 服务端 | Vuex.Store 实例。****只有**[vuex 数据流](https://zh.nuxtjs.org/guide/vuex-store)**存在相关配置时可用**** |  |
| env | Object | 客户端 & 服务端 | nuxt.config.js 中配置的环境变量，见 [环境变量 api](https://zh.nuxtjs.org/api/configuration-env) |  |
| params | Object | 客户端 & 服务端 | route.params 的别名 |  |
| query | Object | 客户端 & 服务端 | route.query 的别名 |  |
| req | [http.Request](https://nodejs.org/api/http.html" \l "http_class_http_incomingmessage) | 服务端 | Node.js API 的 Request 对象。如果 Nuxt 以中间件形式使用的话，这个对象就根据你所使用的框架而定。nuxt generate 不可用 |  |
| res | [http.Response](https://nodejs.org/api/http.html" \l "http_class_http_serverresponse) | 服务端 | Node.js API 的 Response 对象。如果 Nuxt 以中间件形式使用的话，这个对象就根据你所使用的框架而定。nuxt generate 不可用 |  |
| redirect | Function | 客户端 & 服务端 | 用这个方法重定向用户请求到另一个路由。状态码在服务端被使用，默认 302 redirect([status,] path [, query]) |  |
| error | Function | 客户端 & 服务端 | 用这个方法展示错误页：error(params) 。params 参数应该包含 statusCode 和 message 字段 |  |
| nuxtState | | Object | 客户端 | Nuxt 状态，在使用 beforeNuxtRender 之前，用于客户端获取 Nuxt 状态，仅在 universal 模式下可用 |  |
| beforeNuxtRender(fn) | | Function | 服务端 | 使用此方法更新 \_\_NUXT\_\_ 在客户端呈现的变量，fn 调用 (可以是异步) { Components, nuxtState } ，参考 [示例](https://github.com/nuxt/nuxt.js/blob/cf6b0df45f678c5ac35535d49710c606ab34787d/test/fixtures/basic/pages/special-state.vue) |  |

## 页面

页面相当于vue中的一个组件，只不过在该组件中扩展了一些方法，来完成服务器端渲染的目的。

1. asyncData  
    你可能想要在服务器端获取并渲染数据。Nuxt.js添加了asyncData方法使得你能够在渲染组件之前异步获取数据。  
    类型： Function  
    asyncData方法会在组件（限于页面组件）每次加载之前被调用。它可以在服务端或路由更新之前被调用。在这个方法被调用的时候，第一个参数被设定为当前页面的上下文对象，你可以利用 asyncData方法来获取数据并返回给当前组件。  
   export default {   
    data () { return { project: 'default' } },   
    asyncData (context) { return { project: 'nuxt' } }  
    }  
    注意：由于asyncData方法是在组件 初始化 前被调用的，所以在方法内是没有办法通过 this 来引用组件的实例对象。
2. Fetch  
   fetch 方法用于在渲染页面前填充应用的状态树（store）数据， 与 asyncData 方法类似，不同的是它不会设置组件的数据。  
   类型： Function  
    如果页面组件设置了 fetch 方法，它会在组件每次加载前被调用（在服务端或切换至目标路由之前）。  
    fetch 方法的第一个参数是页面组件的[上下文对象](https://zh.nuxtjs.org/api/" \l "%E4%B8%8A%E4%B8%8B%E6%96%87%E5%AF%B9%E8%B1%A1) context，我们可以用 fetch 方法来获取数据填充应用的状态树。为了让获取过程可以异步，你需要返回一个 Promise，Nuxt.js 会等这个 promise 完成后再渲染组件。  
   注意：由于fetch方法是在组件 初始化 前被调用的，所以在方法内是没有办法通过 this 来引用组件的实例对象。  
   <template>   
    <h1>Stars: {{ $store.state.stars }}</h1> </template>   
   <script>   
    export default {   
    async fetch ({ store, params }) {  
    let{data}=await axios.get('http://my-api/stars')   
    store.commit('setStars', data) }  
    }   
   </script>  
   如果要在fetch中调用并操作store，请使用store.dispatch，但是要确保在内部使用async / await等待操作结束：  
   <script>   
    export default {   
    async fetch ({ store, params }) {   
    await store.dispatch('GET\_STARS'); }   
    }   
   </script>  
   store/index.js  
   export const actions = {   
    async GET\_STARS ({ commit }) {   
    const{data}=await axios.get('http://my-api/stars')   
    commit('SET\_STARS', data) }  
    }
3. Head  
   Nuxt.js 使用了 [vue-meta](https://github.com/nuxt/vue-meta) 更新应用的 头部标签(Head) 和 html 属性  
   类型： Object 或 Function  
   使用 head 方法设置当前页面的头部标签。  
   在 head 方法里可通过 this 关键字来获取组件的数据，你可以利用页面组件的数据来设置个性化的 meta 标签。  
   <template> <h1>{{ title }}</h1> </template>  
    <script>   
   export default {   
    data () { return { title: 'Hello World!' } },  
    head () { return {   
    title: this.title,   
    meta: [   
    { hid: 'description', name: 'description', content: 'My custom description' }  
    ]   
    }   
   } }  
    </script>  
   注意：为了避免子组件中的meta标签不能正确覆盖父组件中相同的标签而产生重复的现象，建议利用 hid 键为meta标签配一个唯一的标识编号。
4. Layout  
   layouts 根目录下的所有文件都属于个性化布局文件，可以在页面组件中利用 layout 属性来引用  
   类型： String 或 Function (默认值： 'default')  
   使用 layout 属性来为页面指定使用哪一个布局文件：  
   export default {   
    layout: 'blog', // 或 layout (context) { return 'blog' }   
   }  
   在上面的例子中， Nuxt.js 会使用 layouts/blog.vue 作为当前页面组件的布局文件。
5. Middleware  
   在应用中的特定页面设置中间件  
   类型： String 或 Array(数组元素类型： String)  
   例子：  
   pages/secret.vue  
   <template>   
    <h1>Secret page</h1>   
   </template>   
   <script>   
    export default {   
    middleware: 'authenticated'   
    }   
   </script>  
   middleware/authenticated.js  
   export default function ({ store, redirect }) { (!store.state.authenticated) {  
    return redirect('/login')  
    }  
    }
6. scrollToTop  
   用于控制页面渲染前是否滚动至页面顶部  
   类型： Boolean (默认值： false)  
   默认情况下，从当前页面切换至目标页面时，Nuxt.js 会让目标页面滚动至顶部。但是在嵌套子路由的场景下，Nuxt.js 会保持当前页面的滚动位置，除非在子路由的页面组件中将 scrollToTop 设置为 true。  
   <script>   
    export default {   
    scrollToTop: true  
    }   
   </script>
7. Transition  
   Nuxt.js 使用 Vue.js 的[<transition>](http://vuejs.org/v2/guide/transitions.html" \l "Transitioning-Single-Elements-Components)组件来实现路由切换时的过渡动效  
   类型： String 或 Object 或 Function  
   如果想给某个页面自定义过渡特效的话，只要在该页面组件中配置 transition 字段即可：  
   export default {   
   // 可以是字符   
   transition: ''   
   // 或对象   
   transition: {}   
   // 或函数 transition (to, from) {}   
   }
8. Validate  
   Nuxt.js 可以让你在动态路由对应的页面组件中配置一个校验方法用于校验动态路由参数的有效性。  
   类型： Function  
   async validate({ params, query, store }) {   
    // await operations   
    return true // 如果参数有效  
   }  
   Nuxt.js 可以让你在动态路由对应的页面组件（本例为： pages/users/\_id.vue）中配置一个校验方法。  
   如果校验方法返回的值不为 true， Nuxt.js 将自动加载显示 404 错误页面。  
   export default {   
    validate ({ params }) {   
    // Must be a number   
    return /^\d+$/.test(params.id)   
    }  
    }  
   您还可以在验证函数执行期间抛出预期或意外错误：  
   export default {   
    async validate ({ params, store }) {   
    // 使用自定义消息触发内部服务器500错误 throw new Error('Under Construction!')  
    }  
   }
9. watchQuery  
   监听参数字符串更改并在更改时执行组件方法 (asyncData, fetch, validate, layout, ...)  
   类型: Boolean or Array (默认: [])  
   使用watchQuery属性可以监听参数字符串的更改。 如果定义的字符串发生变化，将调用所有组件方法(asyncData, fetch, validate, layout, ...)。 为了提高性能，默认情况下禁用。  
   如果您要为所有参数字符串设置监听， 请设置： watchQuery: true.  
   export default { watchQuery: ['page'] }

## 组件

1. Nuxt  
   该组件只适用于在[布局](https://zh.nuxtjs.org/guide/views" \l "%E5%B8%83%E5%B1%80)中显示页面组件（即非布局内容）。  
   例如 (layouts/default.vue)：  
   <template>   
    <div>   
    <div>页头</div>  
    <nuxt/>  
    <div>页脚</div>   
    </div>   
   </template>
2. Nuxt-child  
   该组件用于显示[嵌套路由](https://zh.nuxtjs.org/guide/routing" \l "%E5%B5%8C%E5%A5%97%E8%B7%AF%E7%94%B1)场景下的页面内容。
3. Nuxt-link  
   用于在页面中添加链接至别的页面。  
   目前 <nuxt-link> 的作用和 [<router-link>](https://router.vuejs.org/zh-cn/api/router-link.html) 一致，推荐阅读 [Vue路由文档](https://router.vuejs.org/zh-cn/api/router-link.html) 来了解它的使用方法。  
   例如 (pages/index.vue)：  
   <template>   
    <div>   
    <h1>Home page</h1>   
    <nuxt-link to="/about">关于</nuxt-link>   
    </div>   
   </template>  
   别名: <n-link>, <NuxtLink>, 和 <NLink>  
   为了提高Nuxt.js应用程序的响应能力，当链接将显示在视口中时，Nuxt.js将自动预获取代码分割页面。  
   要禁用链接页面的预获取，可以使用no-prefetch：  
   <n-link to="/about" no-prefetch>About page not pre-fetched</n-link>
4. No-ssr  
   该组件用于设置组件不在服务器渲染中呈现。  
   <template>   
    <div>   
    <sidebar />   
    <no-ssr placeholder="Loading...">   
    <!-- 此组件仅在客户端呈现 -->   
    <comments />   
    </no-ssr>   
    </div>   
   </template>

## 插件

1. 第三方插件  
   我们可以在应用中使用第三方模块，一个典型的例子是在客户端和服务端使用 axios 做 HTTP 请求。  
   首先我们需要安装 npm 包：  
   npm install --save axios  
   然后，在页面内这样使用：  
   <template> <h1>{{ title }}</h1> </template>  
    <script>   
    import axios from 'axios'   
    export default {   
    async asyncData ({ params }) {  
    let{data}=await axios.get(`https://posts/${params.id}`)   
    return { title: data.title } }   
    }   
   </script>
2. Vue插件  
   假如我们想使用 [vue-notifications](https://github.com/se-panfilov/vue-notifications) 显示应用的通知信息，我们需要在程序运行前配置好这个插件。  
   首先增加文件 plugins/vue-notifications.js：  
   import Vue from 'vue'   
   import VueNotifications from 'vue-notifications'   
   Vue.use(VueNotifications)  
   然后, 在 nuxt.config.js 内配置 plugins 如下：  
   module.exports={   
    plugins: ['~/plugins/vue-notifications']   
   }
3. 注入 $root 和 context  
   有时您希望在整个应用程序中使用某个函数或属性值，此时，你需要将它们注入到Vue实例（客户端），context（服务器端）甚至 store(Vuex)。按照惯例，新增的属性或方法名使用$作为前缀。  
   （1）.注入 Vue 实例  
   将内容注入Vue实例，避免重复引入，在Vue原型上挂载注入一个函数，所有组件内都可以访问(不包含服务器端)。  
   plugins/vue-inject.js:  
   import Vue from 'vue'   
   Vue.prototype.$myInjectedFunction = (string) =>   
    console.log("This is an example", string)  
   nuxt.config.js:  
   export default {   
    plugins: ['~/plugins/vue-inject.js']  
    }  
   这样，您就可以在所有Vue组件中使用该函数。  
   example-component.vue:  
   export default {   
    mounted(){ this.$myInjectedFunction('test') }   
   }  
   （2）.注入Context  
   context注入方式和在其它vue应用程序中注入类似。  
   plugins/ctx-inject.js:  
   export default (  
   { app }, inject) => {   
    app.myInjectedFunction = (string) =>   
    console.log('Okay, another function', string)   
   }  
   nuxt.config.js:  
   export default {   
    plugins: ['~/plugins/ctx-inject.js']   
   }  
   现在，只要您获得context，你就可以使用该函数（例如在asyncData和fetch中）。   
   ctx-example-component.vue:  
   export default {   
    asyncData(context){   
    context.app.myInjectedFunction('ctx!')  
    }  
    }  
   （3）.同时注入  
   如果您需要同时在context，Vue实例，甚至Vuex中同时注入，您可以使用inject方法,它是plugin导出函数的第二个参数。 将内容注入Vue实例的方式与在Vue应用程序中进行注入类似。系统会自动将$添加到方法名的前面。  
   plugins/combined-inject.js:  
   export default ({ app }, inject) => {   
    inject('myInjectedFunction', (string) =>   
    console.log('That was easy!', string))  
    }  
   nuxt.config.js:  
   export default {  
    plugins: ['~/plugins/combined-inject.js']   
   }  
   现在您就可以在context，或者Vue实例中的this，或者Vuex的actions/mutations方法中的this来调用myInjectedFunction方法。   
   ctx-example-component.vue:  
   export default {   
    mounted(){   
    this.$myInjectedFunction('works in mounted') },   
    asyncData(context){   
    context.app.$myInjectedFunction('works with context') }   
   }  
   store/index.js:  
   export const state = () => ({ someValue: '' })   
   export const mutations = {   
    changeSomeValue(state, newValue) {   
    this.$myInjectedFunction('accessible in mutations')   
    state.someValue = newValue }   
   }   
   export const actions = {   
    setSomeValueToWhatever ({ commit }) {   
    this.$myInjectedFunction('accessible in actions')  
    const newValue = "whatever"   
    commit('changeSomeValue', newValue) }  
    }
4. 只在浏览器中使用的插件  
   不支持ssr的系统，插件只在浏览器里使用，这种情况下下，你可以用 ssr: false ，使得插件只会在客户端运行。  
   nuxt.config.js:  
   module.exports = {  
    plugins: [   
    { src: '~/plugins/vue-notifications', ssr: false }  
    ]  
    }  
   您可以通过检测process.server这个变量来控制插件中的某些脚本库只在服务端使用。当值为 true 表示是当前执行环境为服务器中。 此外，可以通过检查process.static是否为true来判断应用是否通过nuxt generator生成。您也可以组合process.server和process.static这两个选项，确定当前状态为服务器端渲染且使用nuxt generate命令运行。  
   注意：由于Nuxt.js 2.4，模式已被引入作为插件的选项来指定插件类型，可能的值是：client 或 server, ssr:false 在下一个主要版本中弃用,将过渡为 mode: 'client'。  
   nuxt.config.js:  
   export default {   
    plugins: [   
    { src: '~/plugins/both-sides.js' },   
    { src: '~/plugins/client-only.js', mode: 'client' }, { src: '~/plugins/server-only.js', mode: 'server' }   
    ]   
   }
5. 传统命名插件  
   如果假设插件仅在 客户端 或 服务器端 运行，则 .client.js 或 .server.js可以作为插件文件的扩展名应用，该文件将自动包含在相应客户端或者服务端上。  
   nuxt.config.js:  
   export default {   
    plugins: [   
    '~/plugins/foo.client.js', // only in client side '~/plugins/bar.server.js', // only in server side '~/plugins/baz.js' // both client & server  
    ]  
    }

## Vuex状态树

1. 状态树  
   Nuxt.js 会尝试找到应用根目录下的 store 目录，如果该目录存在，它将做以下的事情：  
   （1）引用 vuex 模块

（2）将 vuex 模块 加到 vendors 构建配置中去

（3）设置 Vue 根实例的 store 配置项  
 store 目录下的每个 .js 文件会被转换成为状态树[指定命名的子 模块](http://vuex.vuejs.org/en/modules.html) （当然，index 是根模块）

1. 普通方式  
   Nuxt.js允许您拥有一个 store 目录，其中包含与模块对应的每个文件。
2. nuxtServerInit 方法  
   如果在状态树中指定了 nuxtServerInit 方法，Nuxt.js 调用它的时候会将页面的上下文对象作为第2个参数传给它（服务端调用时才会酱紫哟）。当我们想将服务端的一些数据传到客户端时，这个方法是灰常好用的。  
   举个例子，假设我们服务端的会话状态树里可以通过 req.session.user 来访问当前登录的用户。将该登录用户信息传给客户端的状态树，我们只需更新 store/index.js 如下  
   actions: {   
    nuxtServerInit ({ commit }, { req }) {  
    if (req.session.user) {   
    commit('user', req.session.user) } }  
    }  
   注意：异步nuxtServerInit操作必须返回Promise来通知nuxt服务器等待它们。  
   actions: {   
    async nuxtServerInit({ dispatch }) {  
    await dispatch('core/load') }   
   }

## **开发与部署**

1. 命令列表  
   "scripts": {  
    "test": "jest",  
    "dev": "cross-env NODE\_ENV=development nodemon server/index.js --watch server",  
    "build": "nuxt build",  
    "start": "cross-env NODE\_ENV=production node server/index.js",  
    "generate": "nuxt generate",  
    "eslint": "eslint"  
   }

| 命令 | 描述 |
| --- | --- |
| nuxt | 启动一个热加载的Web服务器（开发模式） [localhost:3000](http://localhost:3000/)。 |
| nuxt build | 利用webpack编译应用，压缩JS和CSS资源（发布用）。 |
| nuxt start | 以生产模式启动一个Web服务器 (nuxt build 会先被执行)。 |
| nuxt generate | 编译应用，并依据路由配置生成对应的HTML文件 (用于静态站点的部署)。 |

如果使用了 Koa/Express 等 Node.js Web 开发框架，并使用了 Nuxt 作为中间件，可以自定义 Web 服务器的启动入口：

| 命令 | 描述 |
| --- | --- |
| NODE\_ENV=development nodemon server/index.js | 启动一个热加载的自定义 Web 服务器（开发模式）。 |
| NODE\_ENV=production node server/index.js | 以生产模式启动一个自定义 Web 服务器 (需要先执行 nuxt build)。 |

1. 开发模式  
   可通过以下命令以开发模式启动带热加载特性的 Nuxt 服务：  
   nuxt   
   // 或   
   npm run dev

3.发布部署  
Nuxt.js 提供了两种发布部署应用的方式：服务端渲染应用部署 和 静态应用部署。  
（1）服务端渲染应用部署  
 部署 Nuxt.js 服务端渲染的应用不能直接使用 nuxt 命令，而应该先进行编译构建，然后再启动 Nuxt 服务，可通过以下两个命令来完成：  
 nuxt build   
 nuxt start  
（2）静态应用部署  
Nuxt.js 可依据路由配置将应用静态化，使得我们可以将应用部署至任何一个静态站点主机服务商。  
可利用下面的命令生成应用的静态目录和文件：  
npm run generate  
这个命令会创建一个 dist 文件夹，所有静态化后的资源文件均在其中。  
（3）单页面应用程序部署 (SPA)  
nuxt generate 在 build/generate 时间内仍然需要SSR引擎，同时具有预渲染所有页面的优势，并具有较高的SEO优化和页面加载能力。 内容在构建时生成。例如，我们不能将它用于内容依赖于用户身份验证或实时API的应用程序（至少对于第一次加载）。  
SPA应用的想法很简单！ 使用时启用SPA模式 mode: 'spa' 或 --spa，并且我们运行打包，生成在导报后自动启动，生成包含常见的meta和资源链接，但不包含页面内容。

因此，对于SPA部署，您必须执行以下操作：

◆ 将nuxt.config.js中的mode更改为spa。  
 ◆ 运行 npm run build.  
 ◆ 自动生成dist/文件夹，部署到您的服务器，如Surge，GitHub Pages或nginx。  
  
另一种可能的部署方法是在spa模式下将Nuxt用作框架中的中间件。这有助于减少服务器负载，并在无法进行SSR的项目中使用Nuxt。

# JavaScript

## 命名

1. 命名的单词应具有描述性，变量，函数使用驼峰式命名  
    正例：book，bookQuery，bookList  
    反例：a, abc, abc123, a\_
2. 类名使用帕斯卡时命名  
    正例：class BookModel{}  
    反例：class bookmodel{}
3. 如果文件只有一个输出类,就把文件名和类名的定义保持一致  
    export default class BookModel{}  
    正例：import BookModel from ‘./BookModel’  
    反例：import BookModel from ‘./book\_model’
4. 导出默认的函数时，文件名和函数名保持一致  
    export default function parseHtml(){}
5. 导出单例、函数库、空对象时使用帕斯卡式命名  
    export default const HtmlUtil = {  
    html:{}  
    }

## 变量属性

1. 定义常量使用const，不要使用var，let  
   正例：const first = 1; const two = 2;  
   反例：var first = 1; var two = 2;
2. 定义变量使用let替换var  
   正例：let first = 1; let two = 2;  
   反例：var first = 1; var two = 2;
3. 使用 . 来访问对象的属性  
   const obj = {name:‘张三’}  
   正例：obj .name;  
   反例：obj [‘name’];
4. 使用变量访问对象的属性时使用[]  
   const obj = {name:‘张三’}  
   getProp(prop){  
    return obj[prop];  
   }
5. var 声明会被提升至该作用域的顶部，但它们赋值不会提升
6. Let和const不会变量提升，被赋予了一种称为「[暂时性死区](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let" \l "Temporal_dead_zone_and_errors_with_let)]
7. 匿名函数表达式的变量名会被提升，但函数内容并不会。

|  |
| --- |
| function example() {  console.log(anonymous); // => undefined  anonymous(); // => TypeError anonymous is not a function  var anonymous = function() {  console.log('anonymous function expression');  };  } |

1. 命名的函数表达式的变量名会被提升，但函数名和函数函数内容并不会。

|  |
| --- |
| function example() {  console.log(named); // => undefined  named(); // => TypeError named is not a function  superPower(); // => ReferenceError superPower is not defined  var named = function superPower() {  console.log('Flying');  };  }  // the same is true when the function name// is the same as the variable name.function example() {  console.log(named); // => undefined  named(); // => TypeError named is not a function  var named = function named() {  console.log('named');  }  } |

1. 函数声明的名称和函数体都会被提升。

|  |
| --- |
| function example() {  superPower(); // => Flying  function superPower() {  console.log('Flying');  }  } |

## 字符串

1. 字符串使用单引号’’  
   正例：const name = ‘张三’;  
   反例：const name = “张三”;
2. 字符串需要多个连接符时，使用模板字符串代替  
   正例：let position = `张三是${leader}，年纪${age}）`;  
   反例：let position= ‘张三是’+ leader+’,年纪’+age;

## 运算符

1. 优先使用 === 和 !== 而不是 == 和 !=
2. 条件表达式例如 if 语句通过抽象方法 ToBoolean 强制计算它们的表达式并且总是遵守下面的规则：

* ****对象**** 被计算为 ****true****
* ****Undefined**** 被计算为 ****false****
* ****Null**** 被计算为 ****false****
* ****布尔值**** 被计算为 ****布尔的值****
* ****数字**** 如果是 ****+0、-0、或 NaN**** 被计算为 ****false****, 否则为 ****true****
* ****字符串**** 如果是空字符串 '' 被计算为 ****false****，否则为 ****true  
  if([0])//true****

1. 使用简单写法  
   正例：  
   if(name){  
   };  
   if(collec.length){  
   };  
   反例：  
   if(name !== ‘’){  
   };  
   if(collec.length>0){  
   };

## 对象

1. 使用字面值创建对象  
   正例：const obj = {};  
   反例：const obj= new Object();
2. 对象的属性和方法，采集简洁方式描述和属性  
   let name = ‘张三’  
   正例：const obj = { name, out(){return name;}};  
   反例：const obj = { name:name, out:function(){return name;}}
3. 不要在对象中的属性和方法使用保留字，有些浏览器会出现兼容性问题,使用同义词替换  
   正例：const obj = { defaults：{key:‘k1}, hidden:true}  
   反例：const obj = { default：{key:‘k1}, private:true}

## 数组

1. 使用字面值创建数组  
   正例：const items = [];  
   反例：const items= new Array();
2. 向数组添加元素时，使用方法push代替直接赋值  
   let items = [];  
   正例：items.push(1);  
   反例：items[item.length] = 1;
3. 使用扩展运算符...复制数组  
   let other = [1,2,3,4,5]  
   正例：const items = [...other];  
   反例：for(){items[i] = other[i]};
4. 使用Array#from把一个类数组对象转换为数组  
   let items = Array.form(obj);

## 解构

1. 使用对象解构赋值  
   正例：  
    function getFullName(obj){  
    const {first, last} = obj  
    return `${first}${last}`;  
    }  
   反例：  
    function getFullName(obj){  
    const first = obj.first;  
    const last = obj.last;  
    return `${first}${last}`;  
    }
2. 使用数组解构赋值  
   const items = [1,2,3];  
   正例：const[first,second] = items;  
   反例：const first = items[0]; const second = item[1];
3. 需要回传多个值时，使用对象解构，而不是数组解构  
   正例：  
    function getFullName(obj){  
    return {first, last};  
    }  
   反例：  
    function getFullName(obj){  
    return [first, last];  
    }

## 函数

1. 使用函数声明代替函数表达式(因为函数声明是可命名的，所以他们在调用栈中更容易被识别。此外，函数声明会把整个函数提升（hoisted），而函数表达式只会把函数的引用变量名提升。这条规则使得[箭头函数](https://github.com/yuche/javascript" \l "arrow-functions)可以取代函数表达式。)  
   正例：function name(){};  
   反例：let name = function(){};
2. 如果需要js块立即执行可以使用立即调用的函数表达式 (IIFE)  
   (()=>{  
    console.log(‘立即执行’);  
   })();
3. 不要在一个非函数代码块（if、while 等）中声明一个函数，把那个函数赋给一个变量。浏览器允许你这么做，但它们的解析表现不一致。
4. 注意: ECMA-262 把 block 定义为一组语句。函数声明不是语句。  
   正例：  
    function getFullName(){  
    let good  
    if(true){  
    good = () =>{};  
    }  
    }  
   反例：  
    function getFullName(obj){  
    if(true){  
    function good(){}  
    }  
    }
5. 不要把参数命名为 arguments。这将取代原来函数作用域内的 arguments 对象  
   正例：function name(key, value, arguments){};  
   反例：function name(key, value, args){};
6. 不要使用 arguments。可以选择 rest 语法 ... 替代(使用 ... 能明确你要传入的参数。另外 rest 参数是一个真正的数组，而 arguments 是一个类数组)  
   正例：

function getFullName(){

const args = Array.prototype.slice.call(arguments);

return args.join('');

}

反例：

function getFullName(...args){

return args.join('');

}

1. 直接给函数的参数指定默认值，不要使用一个变化的函数参数  
   正例：

function fullName(option={}){

//;

}

反例：

function fullName(option){

option = option || {};

}

1. 当你必须使用函数表达式（或传递一个匿名函数）时，使用箭头函数符号  
   正例：

[1,2,3].map((val)=>{

let y = val + 1;

return val+y;

})

反例：

[1,2,3].map(function(val){

let y = val + 1;

return val+y;

})

1. 如果一个函数适合用一行写出并且只有一个参数，那就把花括号、圆括号和 return 都省略掉。如果不是，那就不要省略。  
   正例：

[1,2,3].map(val=>val+1)

## 类

1. 使用 es6 定义 class，避免直接操作 prototype   
   正例：

Class Book{

custructor(){};

getName(){

return ‘’;

}

})

反例：

Function Book(){}

Book.prototype.getName(){

return ‘’;

})

1. 类继承使用extends（是一个内建的原型继承方法并且不会破坏 instanceof）
2. 如果类方法需要链式调用，方法返回this

## 模块

1. 使用es6的标准export、import  
   正例：  
   import { es6 } from './LibEs6;

export default es6;

1. Import 命令 不要使用通配符\*  
   正例：  
   import { es6 } from './LibEs6;  
   反例：  
   import \* as LibEs6 from './LibEs6;
2. 不要从 import 中直接 export  
   正例：  
   import { es6 } from './LibEs6;  
   export default es6;  
   反例：  
   export { es6 as default } from './LibEs6;

## 迭代器

1. 使用高阶函数例如 foreach()、map() 和 reduce() 替代 for-of  
   const items = [1,2,3,4];  
   正例：  
   items.foreach((num)=>{});  
   反例：  
   for(let i=0; i<items.length; i++){}
2. 现在还不要使用 generators,因为它们现在还没法很好地编译到 ES5

## 异步处理

1. 使用promise代替函数回调  
   正例：

|  |
| --- |
| function timeout(ms) {  return new Promise((resolve, reject) => {  setTimeout(resolve, ms, 'done');  });}  timeout(100).then((value) => {  console.log(value);}); |

反例：

|  |
| --- |
| setTimeout(function(){ console.log();  }, ms, 'done'); |

1. 使用async和await处理多个异步操作等待操作

|  |
| --- |
| async function timeout(ms) {  await new Promise((resolve) => {  setTimeout(resolve, ms);  });  }  async function asyncPrint(value, ms) {  await timeout(ms);  console.log(value);  }  asyncPrint('hello world', 50); |

# Html

## 语法

1. 在属性上，使用双引号，不要使用单引号
2. 属性名全小写，用中划线做分隔符
3. 不要在自动闭合标签结尾处使用斜线  
   正例：

|  |
| --- |
| <img src=”...”> |

反例：

|  |
| --- |
| <img src=”...”/> |

1. 不要忽略可选的关闭标签，例：</li> 和 </body>  
   正例：

|  |
| --- |
| <section>   <p>这是一个段落。</p>   <p>这是一个段落。</p> </section> |

反例：

|  |
| --- |
| <section>   <p>这是一个段落。   <p>这是一个段落。 </section> |

1. 尽可能减少div嵌套
2. 在页面开头使用这个简单地doctype来启用标准模式，使其在每个浏览器中尽可能一致的展现  
   正例：

|  |
| --- |
| <!DOCTYPE html>  <html>  ...  </html> |

1. 应在html标签上加上lang属性。这会给语音工具和翻译工具帮助，告诉它们应当怎么去发音和翻译。  
   正例：

|  |
| --- |
| <!DOCTYPE html>  <html lang="en-us">  ...  </html> |

1. 使用小写的元素名称  
   正例：

|  |
| --- |
| <section>    <p>这是一个段落。</p> </section> |

反例：

|  |
| --- |
| <SECTION>    <p>这是一个段落。</p> </SECTION> |

1. 图片通常使用 alt 属性。 在图片不能显示时，它能替代图片显示  
   正例：

|  |
| --- |
| <img src="html5.gif" **alt=**"HTML5" style="width:128px;height:128px"> |

1. 避免一行代码过长,每行代码尽量少于 80 个字符
2. HTML 注释可以写在 <!-- 和 --> 中  
   正例：

|  |
| --- |
| <!-- 这是注释 --> |

## Meta

1. 通过声明一个明确的字符编码，让浏览器快速的确定适合网页内容的渲染方式  
   正例：<meta charset="UTF-8">
2. 优先使用 IE 最新版本和 Chrome  
   正例：<meta http-equiv=”X-UA-Compatible” content=”IE=edge,chrome=1″/>
3. 页面描述不超过150个字符  
   正例：<meta name=”description” content=”不超过150个字符”/>
4. 页面关键词  
   正例：<meta name=”keywords” content=””/>
5. 网页作者  
   正例：<meta name=”author” content=”name, [email@gmail.com”/>](mailto:email@gmail.com\”/>)
6. 为移动设备添加 viewport  
   正例：<meta name=”viewport” content=”initial-scale=1, maximum-scale=1, minimum-scale=1, user-scalable=no”>
7. 设置页面不缓存

<meta http-equiv=”pragma” content=”no-cache”>

<meta http-equiv=”cache-control” content=”no-cache”>

<meta http-equiv=”expires” content=”0″>

## H5语义

1. <header> 元素为文档或节规定页眉。
2. <footer> 元素为文档或节规定页脚。
3. <nav> 元素定义导航链接集合。
4. <aside> 元素页面主内容之外的某些内容（比如侧栏）。
5. <section> 元素定义文档中的节
6. <article> 元素规定独立的自包含内容
7. <figure>元素规定自包含内容，比如图示、图表、照片、代码清单等。
8. <figcaption> 元素定义 <figure> 元素的标题。
9. <mark>元素定义重要的或强调的文本。
10. <main>元素规定文档的主内容。
11. <time>元素定义日期/时间。
12. <audio>元素定义声音或音乐内容
13. <video>元素 定义视频或影片内容

# Css

## 代码

1. 采用UTF-8编码，在CSS头部引用 @charset "utf-8";
2. 使用4个空格作为一个缩进层级，不允许使用2个空格或tab字符
3. 每行不得超过120个字符，除非单行不可分割
4. 选择器嵌套层级不大于3级，位置靠后的限定条件应尽可能精确。
5. 尽量不要使用@import，与<links>相比，@import速度较慢，增加了额外的页面请求，并可能导致其他无法预料的问题。
6. Media Query不得单独编排，必须与相关的规则一起定义。

## 顺序

1. 布局位置（position，top，right，z-index，display，float等）
2. 尺寸大小（宽度，高度，填充，边距）
3. 文字系列（字体，行高，字母间距，颜色文字对齐等）
4. 视觉效果（background，border等）
5. 其他（动画，转换等）

## 属性

1. CSS有些属性是可以缩写的，比如填充，边缘，字体，背景，边框等等。缩写代码可以提高用户的阅读体验
2. 当数值为小数时，小数点前面的“0”可以去除
3. “0像素”后面的单位可以去除
4. 16进制的颜色代码重叠的字符可以缩写的尽量缩写

## 声明

1. 当规则声明块中有多个样式声明时，每条样式独占一行。
2. 列表属性并排书写时，用逗号分隔，逗号后必须跟一个空格
3. 选择器与左大括号 {之间必须加一个空格
4. 属性名与冒号之间不允许包含空格，冒号与属性值之间必须包含空格
5. 在每条样式后面都以分号; 结尾。
6. 规则声明块的右大括号} 独占一行。
7. 所有最外层引号使用单引号 ' ' 。
8. 当一个属性有多个属性值时，以逗号, 分隔属性值，每个逗号后添加一个空格，当单个属性值过长时，每个属性值独占一行。
9. 每个规则声明间用空行分隔。

## 命名

1. 一律小写
2. 尽量用英文
3. 命名短且语义化要好
4. 名字长的单词可以选择使用烤串命名法，中间加横线来为选择器命名
   1. 布局：以g 为命名空间，例如：.g-wrap 、.g-header、.g-content。
   2. 状态：以s 为命名空间，表示动态的、具有交互性质的状态，例如：.s-current、s-selected。
   3. 工具：以u 为命名空间，表示不耦合业务逻辑的、可复用的的工具，例如：u-clearfix、u-ellipsis。
   4. 组件：以m 为命名空间，表示可复用、移植的组件模块，例如：m-slider、m-dropMenu。
   5. 钩子：以j 为命名空间，表示特定给JavaScript 调用的类名，例如：j-request、j-open。
5. 不建议使用下划线来命名，存在浏览器兼容性问题，其次是JavaScript的变量命名也是用下划线，容易造成混淆。
6. 不要滥用id标识，id在JS中是唯一的，不能多次使用，应该按需使用

## 注释

1. Css 只有一种注释方式 /\* Header \*/

# 异常日志

## 目的

待整理...

## 资源

待整理...

## 代码

待整理...

# 单元测试

## 目的

测试保证项目的正确性、完整性、安全性和可靠性。使用的测试框架是Jest框架和vue Test Utils测试。

## 覆盖率

1. 测试语句Statements 结果以百分比展示 100%
2. 测试分支 Branches 结果以百分比展示 100%
3. 测试函数 Functions 结果以百分比展示 100%
4. 测试代码函数 Lines 结果以百分比展示 100%

## 匹配器

1. 普通匹配器  
   expect 方法中的是期望的对象， toBe(4)是匹配的对象。运行测试如果发现和匹配对象不符合，则会报错

|  |
| --- |
| 精准匹配  test('two plus two is four', () => {  expect(2 + 2).toBe(4);  });  test('object assignment', () => {  **const** data = {one: 1};  data['two'] = 2;  expect(data).toEqual({one: 1, two: 2});  });  相反匹配  test('adding positive numbers is not zero', () => {  **for** (**let** a = 1; a < 10; a++) {  **for** (**let** b = 1; b < 10; b++) {  expect(a + b).not.toBe(0);  }  }  }); |
|  |

1. Truthiness  
   区分 undefined、null、false

|  |
| --- |
| 1. toBeNull 只匹配 null 2. toBeUndefined 只匹配 undefined 3. toBeDefined 与 toBeUndefined 相反 4. toBeTruthy 匹配任何 if 语句为真 5. toBeFalsy 匹配任何 if 语句为假   test('null', () => {  **const** n = null;  expect(n).toBeNull();  expect(n).toBeDefined();  expect(n).not.toBeUndefined();  expect(n).not.toBeTruthy();  expect(n).toBeFalsy();  });  test('zero', () => {  **const** z = 0;  expect(z).not.toBeNull();  expect(z).toBeDefined();  expect(z).not.toBeUndefined();  expect(z).not.toBeTruthy();  expect(z).toBeFalsy();  }); |

1. 数字  
   大多数的比较数字有等价的匹配器。

|  |
| --- |
| test('two plus two', () => {  const value = 2 + 2;  expect(value).toBeGreaterThan(3);  expect(value).toBeGreaterThanOrEqual(3.5);  expect(value).toBeLessThan(5);  expect(value).toBeLessThanOrEqual(4.5);  // toBe and toEqual are equivalent for numbers  expect(value).toBe(4);  expect(value).toEqual(4);  });  对于比较浮点数相等，使用 toBeCloseTo 而不是 toEqual，因为你不希望测试取决于一个小小的舍入误差。  test('两个浮点数字相加', () => {  const value = 0.1 + 0.2;  //expect(value).toBe(0.3); 这句会报错，因为浮点数有舍入误差  expect(value).toBeCloseTo(0.3); // 这句可以运行  }); |

1. 字符串

|  |
| --- |
| 您可以检查对具有 toMatch 正则表达式的字符串︰  test('there is no I in team', () => {  expect('team').not.toMatch(/I/);  });  test('but there is a "stop" in Christoph', () => {  expect('Christoph').toMatch(/stop/);  }); |

1. 数组

|  |
| --- |
| 你可以通过 toContain来检查一个数组或可迭代对象是否包含某个特定项：  const shoppingList = [  'diapers',  'kleenex',  'trash bags',  'paper towels',  'beer',  ];  test('the shopping list has beer on it', () => {  expect(shoppingList).toContain('beer');  expect(new Set(shoppingList)).toContain('beer');  }); |

1. 异常

|  |
| --- |
| 如果你想要测试的特定函数抛出一个错误，在它调用时，使用 toThrow。  function compileAndroidCode() {  throw new ConfigError('you are using the wrong JDK');  }  test('compiling android goes as expected', () => {  expect(compileAndroidCode).toThrow();  expect(compileAndroidCode).toThrow(ConfigError);  // You can also use the exact error message or a regexp  expect(compileAndroidCode).toThrow('you are using the wrong JDK');  expect(compileAndroidCode).toThrow(/JDK/);  }); |

## 异步代码

针对异步的代码测试，比如接口

1. 回调  
   最常见的异步模式就是回调模式

|  |
| --- |
| test('the data is peanut butter', done => {  function callback(data) {  expect(data).toBe('peanut butter');  done();  }  fetchData(callback);  });  如果 done()永远不会调用，这个测试将失败，这也是你所希望发生的。 |

1. Promise  
   如果您的代码使用 Promises，还有一个更简单的方法来处理异步测试

|  |
| --- |
| test('the data is peanut butter', () => {  return fetchData().then(data => {  expect(data).toBe('peanut butter');  });  });  test('the fetch fails with an error', () => {  expect.assertions(1);  return fetchData().catch(e => expect(e).toMatch('error'));  });  test('the data is peanut butter', () => {  return expect(fetchData()).resolves.toBe('peanut butter');  });  test('the fetch fails with an error', () => {  return expect(fetchData()).rejects.toMatch('error');  }); |

1. Async/aweit  
   若要编写 async 测试，只要在函数前面使用 async 关键字传递到 test

|  |
| --- |
| test('the data is peanut butter', async () => {  const data = await fetchData();  expect(data).toBe('peanut butter');  });  test('the fetch fails with an error', async () => {  expect.assertions(1);  try { await fetchData(); }  catch (e) { expect(e).toMatch('error'); }  });  test('the data is peanut butter', async () => {  await expect(fetchData()).resolves.toBe('peanut butter');  });  test('the fetch fails with an error', async () => {  await expect(fetchData()).rejects.toThrow('error');  }); |

在这些情况下，async 和 await 仅仅只是语法糖，其本身的逻辑与上述使用 Promise 的示例等效。

## 模拟器

写测试的时候你经常需要在运行测试前做一些准备工作，和在运行测试后进行一些整理工作。 Jest 提供辅助函数来处理这个问题。

1. 重复设置  
   为多个test的前后做准备，可以使用beforeEach和afterEach，每个test执行前后，都会执行beforeEach和afterEach方法

|  |
| --- |
| beforeEach(() => {  initializeCityDatabase();  });  afterEach(() => {  clearCityDatabase();  });  test('city database has Vienna', () => {  expect(isCity('Vienna')).toBeTruthy();  });  test('city database has San Juan', () => {  expect(isCity('San Juan')).toBeTruthy();  }); |

1. 一次性设置  
   为多个test的前后做准备，可以使用beforeEach和afterEach，多个test执行前后，只会执行一次beforeAll和afterAll方法

|  |
| --- |
| beforeAll(() => {  initializeCityDatabase();  });  afterAll(() => {  clearCityDatabase();  });  test('city database has Vienna', () => {  expect(isCity('Vienna')).toBeTruthy();  });  test('city database has San Juan', () => {  expect(isCity('San Juan')).toBeTruthy();  }); |

1. 作用域  
   默认情况下，before 和 after 的块可以应用到文件中的每个测试。 此外可以通过 describe 块来将测试分组。 当 before 和 after的块在 describe 块内部时，则其只适用于该 describe 块内的测试。

|  |
| --- |
| beforeAll(() => console.log('1 - beforeAll'));  afterAll(() => console.log('1 - afterAll'));  beforeEach(() => console.log('1 - beforeEach'));  afterEach(() => console.log('1 - afterEach'));  test('', () => console.log('1 - test'));  describe('Scoped / Nested block', () => {  beforeAll(() => console.log('2 - beforeAll'));  afterAll(() => console.log('2 - afterAll'));  beforeEach(() => console.log('2 - beforeEach'));  afterEach(() => console.log('2 - afterEach'));  test('', () => console.log('2 - test'));});  // 1 - beforeAll  // 1 - beforeEach  // 1 - test  // 1 - afterEach  // 2 - beforeAll  // 1 - beforeEach  // 2 - beforeEach  // 2 - test  // 2 - afterEach  // 1 - afterEach  // 2 - afterAll  // 1 - afterAll |

1. [desribe和test块的执行顺序](https://jestjs.io/docs/zh-Hans/setup-teardown" \l "desribe%E5%92%8Ctest%E5%9D%97%E7%9A%84%E6%89%A7%E8%A1%8C%E9%A1%BA%E5%BA%8F)  
   Jest 会在所有真正的测试开始之前执行测试文件里所有的 describe 处理程序（handlers）。 这是在 before\* 和 after\* 处理程序里面 （而不是在 describe 块中）进行准备工作和整理工作的另一个原因。 当 describe 块运行完后,，默认情况下，Jest 会按照 test 出现的顺序（译者注：原文是in the order they were encountered in the collection phase）依次运行所有测试,，等待每一个测试完成并整理好，然后才继续往下走。

|  |
| --- |
| describe('outer', () => {  console.log('describe outer-a');  describe('describe inner 1', () => {  console.log('describe inner 1');  test('test 1', () => { console.log('test for describe inner 1');  expect(true).toEqual(true);  });  });  console.log('describe outer-b');  test('test 1', () => {  console.log('test for describe outer');  expect(true).toEqual(true);  });  describe('describe inner 2', () => {  console.log('describe inner 2');  test('test for describe inner 2', () => {  console.log('test for describe inner 2');  expect(false).toEqual(false);  });  });  console.log('describe outer-c');  });  // describe outer-a  // describe inner 1  // describe outer-b  // describe inner 2  // describe outer-c  // test for describe inner 1  // test for describe outer  // test for describe inner 2 |

1. 通用建议  
   如果测试失败，第一件要检查的事就是，当仅运行这条测试时，它是否仍然失败。 在 Jest 中很容易地只运行一个测试 — — 只需暂时将 test 命令更改为 test.only:

|  |
| --- |
| test.only('this will be the only test that runs', () => {  expect(true).toBe(false);  });  test('this test will not run', () => {  expect('A').toBe('A');  }); |

## 全局函数

1. afterAll(fn, timeout)  
   在所有的test之后执行，只执行一次
2. afterEach(fn, timeout)  
   在所有的test之后执行，每个test后面都执行一次
3. beforeAll(fn, timeout)  
   在所有的test之前执行，只执行一次
4. beforeEach(fn, timeout)  
   在所有的test之前执行，每个test前面都执行一次
5. describe(name, fn)  
   对test方法进行分组， desribe的执行顺序高于同级的test方法
6. describe.each(table)(name, fn, timeout)  
   对table数组执行对应的test
7. describe.only(name, fn)  
   只运行js文件中为only的描述块，其他的描述块skip跳过
8. describe.only.each(table)(name, fn)  
   只运行js文件中为only的each描述块，其他的each描述块skip跳过
9. describe.skip(name, fn)  
   如果不想运行特定的描述块，可以使用describe.skip
10. describe.skip.each(table)(name, fn)  
    如果不想运行特定的each描述块，可以使用describe.skip.each
11. test(name, fn, timeout)  
    测试函数，第一个参数是测试名称; 第二个参数是包含测试期望的函数
12. test.each(table)(name, fn, timeout)  
    对table数组进行测试，fn执行次数是数组的遍历次数
13. test.only(name, fn, timeout)  
    针对js文件中，只执行test.only方法，其他test方法忽略
14. test.only.each(table)(name, fn)  
    针对js文件中，只执行test.only.each方法，其他test.each方法忽略
15. test.skip(name, fn)  
    如果不想运行test，可以使用test.skip
16. test.skip.each(table)(name, fn)  
    如果不想运行test.each，可以使用test.skip.each
17. test.todo(name)  
    摘要输出中突出显示

## Expect

1. expect(value)  
   expect 为测试提供一个值，来匹配判断
2. expect.extend(matchers)  
   您可以使用expect.extend将自己的匹配器添加到Jest
3. expect.anything()  
   匹配任何东西，不包括null或undefined
4. expect.any(constructor)  
   匹配使用给定构造函数创建的任何内容
5. expect.arrayContaining(array)  
   匹配一个包含所需数组中所有元素的接收数组
6. expect.assertions(number)  
   验证在测试期间调用了一定数量的断言
7. expect.hasAssertions()  
   验证在测试期间至少调用一个断言
8. expect.not.arrayContaining(array)  
   匹配一个接收的数组，该数组不包含预期数组中的所有元素。
9. expect.not.objectContaining(object)  
   匹配任何不递归匹配预期属性的接收对象
10. expect.not.stringContaining(string)  
    如果它不是字符串，或者它是不包含确切预期字符串的字符串，则匹配接收到的值。
11. expect.not.stringMatching(string | regexp)  
    如果它不是字符串，或者它是与预期字符串或正则表达式不匹配的字符串，则匹配接收到的值。
12. expect.objectContaining(object)  
    匹配任何递归匹配预期属性的接收对象。也就是说，预期对象是接收对象的子集。因此，它匹配包含预期对象中存在的属性的接收对象。
13. expect.stringContaining(string)  
    匹配接收的值，如果它是包含确切的预期字符串的字符串。
14. expect.stringMatching(string | regexp)  
    匹配接收的值，如果它是与预期的字符串或正则表达式匹配的字符串。
15. expect.addSnapshotSerializer(serializer)  
    添加格式化特定于应用程序的数据结构的模块。
16. .not  
    测试值的反例
17. .resolves  
    测试promises的resolves
18. .rejects  
    测试promises的rejects
19. .toBe(value)  
    比较原始值或检查对象实例的引用一致性。它调用Object.is比较值，这比测试更===严格的相等运算符更好。
20. .toHaveBeenCalled()  
    以确保模拟功能得到调用
21. .toHaveBeenCalledTimes(number)  
    以确保模拟功能得到调用次数确切数字
22. .toHaveBeenCalledWith(arg1, arg2, ...)  
    以确保模拟函数被调用的具体参数
23. .toHaveBeenLastCalledWith(arg1, arg2, ...)  
    测试最后一次调用它的参数
24. .toHaveBeenNthCalledWith(nthCall, arg1, arg2, ....)  
    测试它被调用的参数
25. .toHaveReturned()  
    用来测试mock函数成功返回（即没有抛出错误）至少一次
26. .toHaveReturnedTimes(number)  
    以确保模拟函数返回成功（即未引发错误）的次一个确切的数字。对抛出错误的模拟函数的任何调用都不计入函数返回的次数。
27. .toHaveReturnedWith(value)  
    以确保模拟函数返回的特定值
28. .toHaveLastReturnedWith(value)  
    测试一个模拟函数的最后一个返回的特定值
29. .toHaveNthReturnedWith(nthCall, value)  
    测试一个模拟功能，第n个调用返回的特定值
30. .toHaveLength(number)  
    检查的对象有一个.length属性
31. .toHaveProperty(keyPath, value?)  
    检查对象中各种属性的存在和值
32. .toBeCloseTo(number, numDigits?)  
    检查浮点数完全相等
33. .toBeDefined()  
    检查一个变量是不是不确定的
34. .toBeFalsy()  
    检查布尔上下文中的值为false
35. .toBeGreaterThan(number)  
    检查是否大于number
36. .toBeGreaterThanOrEqual(number)  
    检查是否大于等于number
37. .toBeLessThan(number)  
    检查是否小于number
38. .toBeLessThanOrEqual(number)  
    检查是否小于等于number
39. .toBeInstanceOf(Class)  
    检查对象是一个类的实例。这个匹配器使用instanceof
40. .toBeNull()  
    检查某些内容是否为null
41. .toBeTruthy()  
    检查某些内容是否为false
42. .toBeUndefined()  
    检查某些内容是否为undefined
43. .toBeNaN()  
    检查某些内容是否为nan
44. .toContain(item)  
    检查的item是在数组中
45. .toContainEqual(item)  
    检查是否具有特定结构和值的item包含在一个阵列
46. .toEqual(value)  
    递归地比较对象实例的所有属性（也称为“深度”相等）。它调用Object.is比较原始值，这对于测试比===严格相等运算符更好
47. .toMatch(regexpOrString)  
    检查字符串中的正则表达式匹配
48. .toMatchObject(object)  
    检查JavaScript对象的属性的子集相匹配
49. .toMatchSnapshot(propertyMatchers?, hint?)  
    检查确保值与最新快照匹配
50. .toMatchInlineSnapshot(propertyMatchers?, inlineSnapshot)  
    检查确保值与最新快照匹配
51. .toStrictEqual(value)  
    检查对象具有相同类型和结构
52. .toThrow(error?)  
    检查异常抛出
53. .toThrowErrorMatchingSnapshot(hint?)  
    检查函数抛出匹配最近的快照错误
54. .toThrowErrorMatchingInlineSnapshot(inlineSnapshot)  
    检查函数抛出匹配与最新快照匹配

## Vue DOM和事件模拟

Vue Test Utils 通过将它们隔离挂载，然后模拟必要的输入 (prop、注入和用户事件) 和对输出 (渲染结果、触发的自定义事件) 的断言来测试 Vue 组件

1. 挂载组件  
   被挂载的组件会返回到一个[包裹器](https://vue-test-utils.vuejs.org/zh/api/wrapper/)内，而包裹器会暴露很多封装、遍历和查询其内部的 Vue 组件实例的便捷的方法  
   通过 mount 方法来创建包裹器

|  |
| --- |
| // 从测试实用工具集中导入 `mount()` 方法  // 同时导入你要测试的组件import { mount } from '@vue/test-utils'import Counter from './counter'  // 现在挂载组件，你便得到了这个包裹器  const wrapper = mount(Counter)  //输出html  wrapper.html()  // 也便于检查已存在的元素  it('has a button', () => {  expect(wrapper.contains('button')).toBe(true)  })  // 你可以通过 `wrapper.vm` 访问实际的 Vue 实例  const vm = wrapper.vm  // 在控制台将其记录下来即可深度审阅包裹器  // 我们对 Vue Test Utils 的探索也由此开始  console.log(wrapper) |

1. 浅渲染  
    在测试用例中，我们通常希望专注在一个孤立的单元中测试组件，避免对其子组件的行为进行间接的断言。

额外的，对于包含许多子组件的组件来说，整个渲染树可能会非常大。重复渲染所有的子组件可能会让我们的测试变慢。

Vue Test Utils 允许你通过 shallowMount 方法只挂载一个组件而不渲染其子组件 (即保留它们的存根)：

|  |
| --- |
| import { shallowMount } from '@vue/test-utils'  const wrapper = shallowMount(Component)  wrapper.vm // 挂载的 Vue 实例 |

1. 断言触发事件  
   每个挂载的包裹器都会通过其背后的 Vue 实例自动记录所有被触发的事件。你可以用 wrapper.emitted() 方法取回这些事件记录。

|  |
| --- |
| wrapper.vm.$emit('foo')  wrapper.vm.$emit('foo', 123)  // 断言事件已经被触发  expect(wrapper.emitted().foo).toBeTruthy()  // 断言事件的数量  expect(wrapper.emitted().foo.length).toBe(2)  // 断言事件的有效数据expect(wrapper.emitted().foo[1]).toEqual([123]) |

1. 从子组件出发事件  
   你可以通过访问子组件实例来触发一个自定义事件。

|  |
| --- |
| <template>  <div>  <child-component @custom="onCustom" />  <p v-if="emitted">Emitted!</p>  </div></template>  <script>  import ChildComponent from './ChildComponent'  export default {  name: 'ParentComponent',  components: { ChildComponent },  data() {  return {  emitted: false  }  },  methods: {  onCustom() {  this.emitted = true  }  }  }</script>  import { shallowMount } from '@vue/test-utils'  import ParentComponent from '@/components/ParentComponent'  import ChildComponent from '@/components/ChildComponent'  describe('ParentComponent', () => {  it("displays 'Emitted!' when custom event is emitted", () => {  const wrapper = shallowMount(ParentComponent)  wrapper.find(ChildComponent).vm.$emit('custom')  expect(wrapper.html()).toContain('Emitted!')  })}) |

1. 操作组件状态  
   可以在包裹器上用 setData 或 setProps 方法直接操作组件状态。

|  |
| --- |
| wrapper.setData({ count: 10 })  wrapper.setProps({ foo: 'bar' }) |

1. 仿造Prop  
   你可以使用 Vue 在内置 propsData 选项向组件传入 prop。

|  |
| --- |
| import { mount } from '@vue/test-utils'  mount(Component, {  propsData: {  aProp: 'some value'  }}) |

你也可以用 wrapper.setProps({}) 方法更新这些已经挂载的组件的 prop

1. 应用全局的插件和混入  
   有些组件可能依赖一个全局插件或混入 (mixin) 的功能注入，比如 vuex 和 vue-router。

如果你在为一个特定的应用撰写组件，你可以在你的测试入口处一次性设置相同的全局插件和混入。但是有些情况下，比如测试一个可能会跨越不同应用共享的普通的组件套件的时候，最好还是在一个更加隔离的设置中测试你的组件，不对全局的 Vue 构造函数注入任何东西。我们可以使用 createLocalVue 方法来存档它们

|  |
| --- |
| import { createLocalVue } from '@vue/test-utils'  // 创建一个扩展的 `Vue` 构造函数const localVue = createLocalVue()  // 正常安装插件  localVue.use(MyPlugin)  // 在挂载选项中传入 `localVue`  mount(Component, {localVue}) |

1. 仿造注入  
   另一个注入 prop 的策略就是简单的仿造它们。你可以使用 mocks 选项：

|  |
| --- |
| import { mount } from '@vue/test-utils'  const $route = {  path: '/',  hash: '',  params: { id: '123' },  query: { q: 'hello' }}  mount(Component, {  mocks: {  // 在挂载组件之前  // 添加仿造的 `$route` 对象到 Vue 实例中  $route  }}) |

1. 存根组件  
   你可以使用 stubs 选项覆写全局或局部注册的组件。

|  |
| --- |
| import { mount } from '@vue/test-utils'  mount(Component, {  // 将会把 globally-registered-component 解析为  // 空的存根  stubs: ['globally-registered-component']}) |

1. 键盘、鼠标等其它事件  
   Wrapper 暴露了一个 trigger 方法。它可以用来触发 DOM 事件。

|  |
| --- |
| const wrapper = mount(MyButton)  wrapper.trigger('click') |

find 方法也会返回一个 Wrapper。假设 MyComponent 包含一个按钮，下面的代码会点击这个按钮

|  |
| --- |
| wrapper.find('button').trigger('click') |

其 trigger 方法接受一个可选的 options 对象。这个 options 对象里的属性会被添加到事件中。

注意其目标不能被添加到 options 对象中。

|  |
| --- |
| wrapper.find('button').trigger('click',{button:0}) |

1. API  
   (1)、mount() 创建一个包含被挂载和渲染的 Vue 组件的 [Wrapper](https://vue-test-utils.vuejs.org/zh/api/wrapper/)  
   (2)、shallowMount() 和 [mount](https://vue-test-utils.vuejs.org/zh/api/mount.html) 一样，创建一个包含被挂载和渲染的 Vue 组件的 [Wrapper](https://vue-test-utils.vuejs.org/zh/api/wrapper/)，不同的是被存根的子组件。  
   (3)、render() 将一个对象渲染成为一个字符串并返回一个 cheerio 包裹器。Cheerio 是一个类似 jQuery 的库，可以在 Node.js 中游览 DOM 对象。它的 API 和 Vue Test Utils 的 [Wrapper](https://vue-test-utils.vuejs.org/zh/api/wrapper/) 类似  
   (4)、renderToString() 将一个组件渲染为 HTML  
   (5)、选择器 很多方法的参数中都包含选择器。一个选择器可以是一个 CSS 选择器、一个 Vue 组件或是一个查找选项对象  
   (6)、createLocalVue() 返回一个 Vue 的类供你添加组件、混入和安装插件而不会污染全局的 Vue 类  
   (7)、createWrapper(node,[options]) 为一个被挂载的 Vue 实例或一个 HTML 元素创建一个 Wrapper  
   (8)、配置 存储在 config.stubs 中的存根会被默认使用，用到的组件存根。它们会被传入挂载选项的 stubs 覆写
2. Wrapper
3. WrapperArray
4. 挂载选项
5. 组件

# Eslint 规范

待整理...

# 网络安全

待整理...