Vue项目最佳实践

资源

Vue-CLI 3.0

目标

- 项目配置
- 权限管理
- 导航菜单
- 数据mock
- 测试

知识点

项目配置策略

基础配置：指定应用上下文、端口号，vue.config.js

```
const port = 7070;

module.exports = {
  publicPath: '/best-practice', // 部署应用包时的基本 URL
  devServer: {
    port,
  }
};
```

配置webpack：`configureWebpack`

范例：设置一个组件存放路径的别名，vue.config.js

```
const path=require('path')

module.exports = {
  configureWebpack: {
    resolve: {
      alias: {
        comps: path.join(__dirname, 'src/components'),
      }
    }
  }
}
```

范例：设置一个webpack配置项用于页面title，vue.config.js

```
module.exports = {
  configureWebpack: {
    name: "vue项目最佳实践"
  }
};
```

在宿主页面使用lodash插值语法使用它，./public/index.html

```
<title><%= webpackConfig.name %></title>
```

> webpack-merge合并出最终选项

范例：基于环境有条件地配置，vue.config.js

```
// 传递一个函数给configureWebpack
// 可以直接修改，或返回一个用于合并的配置对象
configureWebpack: config => {
  config.resolve.alias.comps = path.join(__dirname, 'src/components')
  if (process.env.NODE_ENV === 'development') {
    config.name = 'vue项目最佳实践'
  } else {
    config.name = 'Vue Best Practice'
  }
}
```

配置webpack：`chainWebpack`

webpack-chain称为链式操作，可以更细粒度控制webpack内部配置。

范例：svg icon引入

- 下载图标，存入src/icons/svg中
- 安装依赖：svg-sprite-loader

```
npm i svg-sprite-loader -D
```

- 修改规则和新增规则，vue.config.js

```
// resolve定义一个绝对路径获取函数
const path = require('path')

function resolve(dir) {
  return path.join(__dirname, dir)
}
//...
chainWebpack(config) {
    // 配置svg规则排除icons目录中svg文件处理
    // 目标给svg规则增加一个排除选项exclude:['path/to/icon']
    config.module.rule("svg")
      .exclude.add(resolve("src/icons"))

    // 新增icons规则，设置svg-sprite-loader处理icons目录中的svg
    config.module.rule('icons')
      .test(/\.svg$/)
      .include.add(resolve('./src/icons')).end()
      .use('svg-sprite-loader')
      .loader('svg-sprite-loader')
        .options({symbolId: 'icon-[name]'})
}
```

- 使用图标，App.vue

```
<template>
  <svg>
    <use xlink:href="#icon-wx" />
  </svg>
</template>
<script>
  import '@/icons/svg/wx.svg'
</script>
```

- 自动导入

创建icons/index.js

```
const req = require.context('./svg', false, /\.svg$/)
req.keys().map(req);
```

创建SvgIcon组件，components/SvgIcon.vue

```
<template>
  <svg :class="svgClass" v-on="$listeners">
    <use :xlink:href="iconName" />
  </svg>
</template>

<script>
export default {
  name: 'SvgIcon',
  props: {
    iconClass: {
      type: String,
      required: true
    },
    className: {
      type: String,
      default: ''
    }
  },
  computed: {
    iconName() {
      return `#icon-${this.iconClass}`
    },
    svgClass() {
      if (this.className) {
        return 'svg-icon ' + this.className
      } else {
        return 'svg-icon'
      }
    }
  }
}
</script>

<style scoped>
.svg-icon {
  width: 1em;
  height: 1em;
  vertical-align: -0.15em;
  fill: currentColor;
  overflow: hidden;
```

```
}
</style>
```

环境变量和模式

如果想给多种环境做不同配置，可以利用vue-cli提供的**模式**。默认有`development`、`production`、`test`三种模式，对应的，它们的配置文件形式是`.env.development`。

范例：定义一个开发时可用的配置项，创建.env.dev

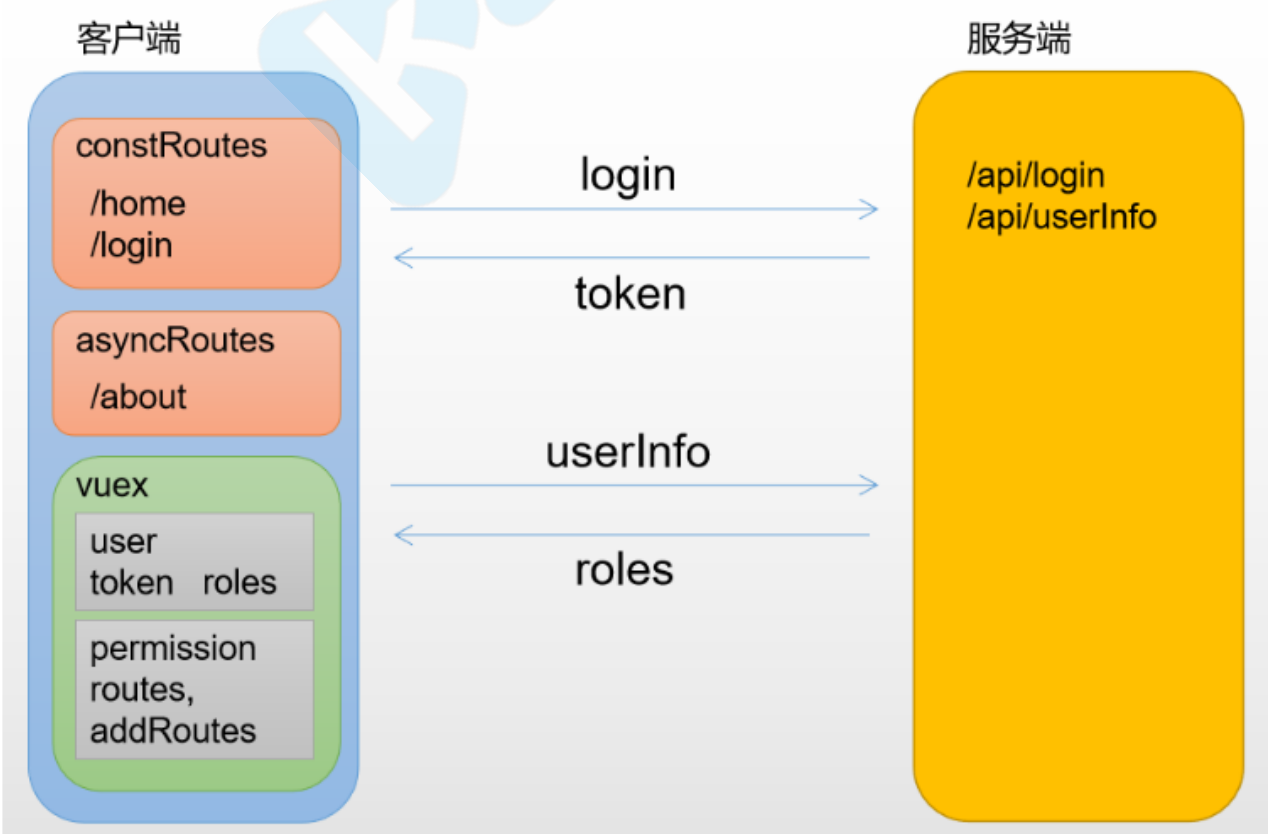
```
# 只能用于服务端
foo=bar
# 可用于客户端
VUE_APP_DONG=dong
```

修改mode选项覆盖模式名称，package.json

```
"serve": "vue-cli-service serve --mode dev"
```

权限控制

> 参考代码：git reset --hard step-7



路由定义

路由分为两种：

- `constantRoutes`：通用路由可直接访问
- `asyncRoutes`：权限路由，需要先登录，获取角色后才能判断是否可以访问

router/index.js

```
// 通用页面：不需要守卫，可直接访问
export const constRoutes = [...
];

// 权限页面：受保护页面，要求用户登录并拥有访问权限的角色才能访问
export const asyncRoutes = [
  {
    path: "/about",
    component: () =>
      import(/* webpackChunkName: "home" */ "@/views/About.vue"),
    name: "about",
    meta: {
      title: "About",
      icon: "denglong",
      roles: ['admin','editor']
    },
  }
];
```

路由守卫

默认路由守卫规则：

- 已登录访问登录页：跳转首页
- 已登录访问其他页：
  - 已获取角色：放行
  - 为获取角色：请求角色 =》过滤可访问路由 =》动态增加到router
- 未登录访问白名单页面：放行
- 未登录访问其他页：跳转至登录页

参考代码：src/permission.js

```
router.beforeEach(async (to, from, next) => {
  // 获取令牌判断用户是否登录
  const hasToken = localStorage.getItem('token')

  if (hasToken) { // 已登录
    if (to.path === '/login') { // 访问登录页
      // 若已登录没有必要显示登录页，重定向至首页
      next('/')
    } else { // 访问其他页
      const hasRoles = store.getters.roles && store.getters.roles.length

      if (hasRoles) {…
      } else {…
      }
    }
  } else {// 未登录
    if (whiteList.indexOf(to.path) !== -1) { // 访问白名单中页面…
    } else { // 访问其他页       You, a few seconds ago • Uncommitted chang
    }
  }
})
```

用户登录

请求登录`dispatch('user/login')`，Login.vue

```
login() {
  this.$store
    .dispatch("user/login", { username: this.username })
    .then(() => {
      this.$router.push({
        path: this.$route.query.redirect || "/"
      });
    })
    .catch(error => {
      alert(error);
    });
}
```

```
login({ commit }, userInfo) {
  return login(userInfo).then((res) => {
    commit("setToken", res.data);
    localStorage.setItem("token", res.data);
  });
```

用户角色获取

登录成功后，请求用户角色信息

```
// 先请求获取用户信息
const { roles } = await store.dispatch('user/getInfo')
```

```
getInfo({ commit, state }) {
  return getInfo(state.token).then(({data: roles}) => {
    commit("setRoles", roles);
    return {roles}
  })
```

权限路由过滤

根据角色过滤asyncRoutes，并动态添加至router。

请求生成路由，src/permission.js

```
// 根据当前用户角色过滤出可访问路由
const accessRoutes = await store.dispatch(
  'permission/generateRoutes', roles)
```

生成路由，store/modules/permission.js

```
generateRoutes({ commit }, roles) {
  return new Promise(resolve => {
    // 根据角色做过滤处理
    const accessedRoutes = filterAsyncRoutes(asyncRoutes, roles);
    commit("setRoutes", accessedRoutes);
    resolve(accessedRoutes);
  });
}
```

```
export function filterAsyncRoutes(routes, roles) {
  const res = [];

  routes.forEach(route => {
    // 复制一份
    const tmp = { ...route };
    // 如果用户有访问权则加入结果路由表
    if (hasPermission(roles, tmp)) {
      // 如果存在子路由则递归过滤之
      if (tmp.children) {
        tmp.children = filterAsyncRoutes(tmp.children, roles);
      }
      res.push(tmp);
    }
  });

  return res;
}
```

异步获取路由表

可以当用户登录后**向后端请求可访问的路由表**，从而动态生成可访问页面，操作和原来是相同的，这里多了一步将后端返回路由表中**组件名称和本地的组件映射**步骤：

```
// 前端组件名和组件映射表
const map = {
  //xx: require('@/views/xx.vue').default  // 同步的方式
  xx: () => import('@/views/xx.vue')     // 异步的方式
}
// 服务端返回的asyncRoutes
const asyncRoutes = [
  { path: '/xx', component: 'xx',... }
]
// 遍历asyncRoutes，将component替换为map[component]
function mapComponent(asyncRoutes) {
```

```
    asyncRoutes.forEach(route => {
        route.component = map[route.component];
        if(route.children) {
            route.children.map(child => mapComponent(child))
        }
    })
}
mapComponent(asyncRoutes)
```

按钮权限

页面中按钮、链接需要更细粒度权限控制时，可封装一个指令`v-permission`，实现按钮级别的权限控制。

创建指令，src/directives/permission.js

```
const permission = {
  inserted(el, binding) {
    // 获取指令的值：按钮要求的角色数组
    const { value:pRoles } = binding;
    // 获取用户角色
    const roles = store.getters && store.getters.roles;

    if (pRoles && pRoles instanceof Array && pRoles.length > 0) {
      // 判断用户角色中是否有按钮要求的角色
      const hasPermission = roles.some(role => {      yt0379, 9 mo

      // 如果没有权限则删除当前dom
      if (!hasPermission) {
        el.parentNode && el.parentNode.removeChild(el);
      }
    } else {
      throw new Error(`需要指定按钮要求角色数组，如v-permission="['admi
    }
  }
};
```

> 该指令只能删除挂载指令的元素，对于那些额外生成的和指令无关的元素无能为力，比如：

```
<el-tabs>
 <el-tab-pane label="用户管理" name="first">
     用户管理</el-tab-pane>
 <el-tab-pane label="角色管理" name="third">
     角色管理</el-tab-pane>
</el-tabs>
```

尝试添加`v-permission="..."`并不能删除动态生成的内容部分

```
<el-tab-pane label="用户管理" name="first"
             v-permission="['admin', 'editor']">用户管理</el-tab-pane>
```

此时可使用v-if来实现

```
<template>
  <el-tab-pane v-if="checkPermission(['admin'])">
</template>
```

导航菜单

导航菜单可根据前面生成的最终路由信息动态生成。

侧边栏组件，components/Sidebar/index.vue

```
<el-scrollbar wrap-class="scrollbar-wrapper">
  <el-menu
    :default-active="activeMenu"
    :background-color="variables.menuBg"
    :text-color="variables.menuText"
    :unique-opened="false"
    :active-text-color="variables.menuActiveText"
    :collapse-transition="false"
    mode="vertical"
  >
    <sidebar-item
      v-for="route in permission_routes"
      :key="route.path"
      :item="route"
      :base-path="route.path"
    />
  </el-menu>
</el-scrollbar>
```

侧边栏菜项目组件，components/Sidebar/SidebarItem.vue

默认菜单项规则：

- 跳转链接：没有子路由或只有一个需要展示子路由

```
<template v-if="hasOneShowingChild(item.children,item) && (!onlyOneChild.chil
  <router-link v-if="onlyOneChild.meta" :to="resolvePath(onlyOneChild.path)">
    <el-menu-item :index="resolvePath(onlyOneChild.path)" :class="{'submenu-t
      <!-- 抽象出一个item组件处理icon和title -->
      <item :icon="onlyOneChild.meta.icon||(item.meta&&item.meta.icon)" :titl
    </el-menu-item>
  </router-link>
</template>
```

- 其他情况：嵌套子菜单

```vue
<el-submenu v-else ref="subMenu" :index="resolvePath(item.path)" popper-append-to-b
  <!-- 标题 -->
  <template v-slot:title>
    <item v-if="item.meta" :icon="item.meta && item.meta.icon" :title="item.meta.ti
  </template>
  <!-- 子菜单 -->
  <sidebar-item
    v-for="child in item.children"
    :key="child.path"
    :is-nest="true"
    :item="child"
    :base-path="resolvePath(child.path)"
    class="nest-menu"
  />
</el-submenu>
```

菜单项组件，components/Sidebar/Item.vue

```vue
render(h, context) {
  const { icon, title } = context.props;
  const vnodes = [];

  if (icon) {
    vnodes.push(<svg-icon icon-class={icon} />);
  }

  if (title) {
    vnodes.push(<span slot="title">{title}</span>);
  }
  return vnodes;
}
```

数据交互

常见需求：

- 统一配置请求库
- 请求拦截和响应拦截
- 数据mock
- 请求代理

请求封装

对axios做一次封装，统一处理配置、请求和响应拦截，utils/request.js

```
// 创建axios实例
const axios = Axios.create({
  baseURL: process.env.VUE_APP_BASE_API, // url基础地址,
  // withCredentials: true, // 跨域时若要发送cookies需设置
  timeout: 5000 // 超时
});

// 请求拦截
axios.interceptors.request.use(…

// 响应拦截
axios.interceptors.response.use(…
```

数据mock

数据模拟两种常见方式，**本地mock**和**线上mock**

本地mock

在vue.config.js中定义模拟接口

```
module.exports = {
  devServer: {
    before(app) {
      // 定义接口
    }
  }
}
```

线上mock

诸如`easy-mock`这类线上mock工具优点是使用简单，mock工具强大，还能整合swagger。

环境搭建

- 线上使用：登录easy-mock
- 搭建本地服务（基于docker）
  - 安装docker desktop
  - 创建docker-compose.yml

```
version: '3'

services:
  mongodb:
    image: mongo:3.4.1
    # volumes:
    # /apps/easy-mock/data/db是数据库文件存放地址，根据需要修改为本地地址
    # - '/apps/easy-mock/data/db:/data/db'
    networks:
      - easy-mock
    restart: always

  redis:
    image: redis:4.0.6
    command: redis-server --appendonly yes
    # volumes:
    # /apps/easy-mock/data/redis 是 redis 数据文件存放地址，根据需要修改为本地地址
    # - '/apps/easy-mock/data/redis:/data'
    networks:
      - easy-mock
    restart: always

  web:
    image: easymock/easymock:1.6.0
    # easy-mock 官方给出的文件，这里是 npm start，这里修改为 npm run dev
    command: /bin/bash -c "npm run dev"
    ports:
      - 7300:7300
    # volumes:
      # 日志地址，根据需要修改为本地地址
      # - '/apps/easy-mock/logs:/home/easy-mock/easy-mock/logs'
    networks:
      - easy-mock
    restart: always

networks:
  easy-mock:
```

  - 启动：`docker-compose up`

使用介绍

- 创建一个项目
- 创建需要的接口：

  登录接口`user/login`

```
{
  "code": function({_req}) {
    const {username} = _req.body;
    if (username === "admin" || username === "jerry") {
      return 1
    } else {
      return 10008
    }
  },
  "data": function({_req}) {
    const {username} = _req.body;
    if (username === "admin" || username === "jerry") {
      return username
    } else {
      return ''
    }
  }
}
```

  用户角色接口：user/info

```
{
  code: 1,
  "data": function({_req}) {
    return _req.headers['authorization'].split(' ')[1] === 'admin' ?
['admin'] : ['editor']
  }
}
```

- 调用：修改base_url，.env.development

```
VUE_APP_BASE_API = 'http://localhost:7300/mock/5f6301c446875b001d8a2961'
```

解决跨域

如果请求的接口在另一台服务器上，开发时则需要设置代理避免跨域问题

代理配置，vue.config.js

```
devServer: {
  port: port,
  proxy: {
    // 代理 /dev-api/user/login 到 http://127.0.0.1
    [process.env.VUE_APP_BASE_API]: {
      target: `http://127.0.0.1:3000/`,
      changeOrigin: true,
      pathRewrite: {
        ["^" + process.env.VUE_APP_BASE_API]: ""
      }
    }
  },
```

测试

测试分类

常见的开发流程里，都有测试人员，他们不管内部实现机制，只看最外层的输入输出，这种我们称为**黑盒测试**。比如你写一个加法的页面，会设计N个用例，测试加法的正确性，这种测试我们称之为**E2E测试**。

还有一种测试叫做**白盒测试**，我们针对一些内部核心实现逻辑编写测试代码，称之为**单元测试**。

更负责一些的我们称之为**集成测试**，就是集合多个测试过的单元一起测试。

测试的好处

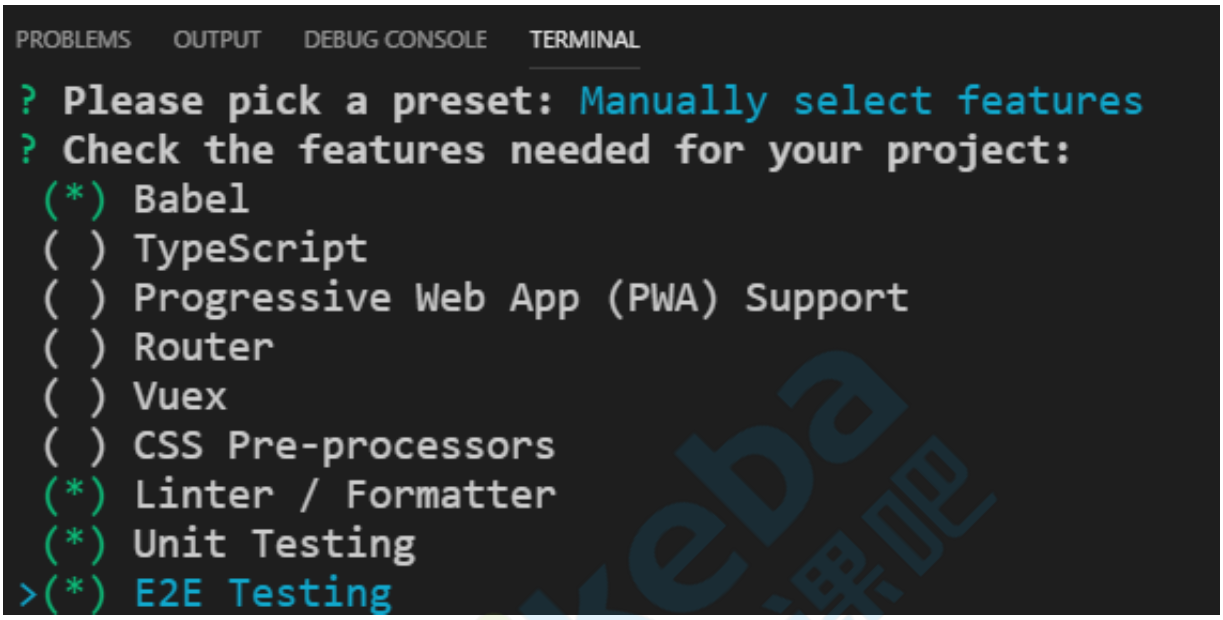
- 提供描述组件行为的文档
- 节省手动测试的时间
- 减少研发新特性时产生的 bug
- 改进设计
- 促进重构

准备工作

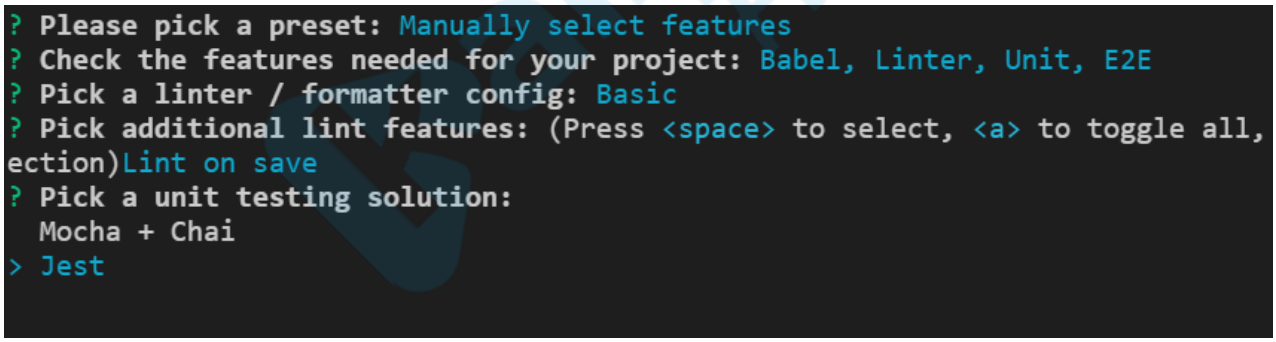
在vue-cli中，预置了Mocha+Chai和Jest两套单测方案，我们的演示代码使用Jest，它们语法基本一致

新建vue项目时

选择特性`Unit Testing`和`E2E Testing`



单元测试解决方案选择：`Jest`



在已存在项目中集成

集成Jest：`vue add @vue/unit-jest`

集成cypress：`vue add @vue/e2e-cypress`

编写单元测试

单元测试（unit testing），是指对软件中的最小可测试单元进行检查和验证。

新建test/unit/kaikeba.spec.js，`*.spec.js`是命名规范

```
function add(num1, num2) {
  return num1 + num2
}

// 测试套件 test suite
describe('add方法', () => {
  // 测试用例 test case
  it('应该能正确计算加法', () => {
    // 断言 assert
    expect(add(1, 3)).toBe(4)
  })
})
```

> 更多断言API

执行单元测试

执行：`npm run test:unit`

```
FAIL tests/unit/kaikeba.spec.js
  ● Kaikeba › 测试加法

    expect(received).toBe(expected) // Object.is equality

    Expected: 3
    Received: 4

       6 | describe('Kaikeba', () => {
       7 |     it('测试加法', () => {
    >  8 |         expect(add(1, 3)).toBe(3)
         |                           ^
       9 |         expect(add(1, 3)).toBe(4)
      10 |         expect(add(-2, 3)).toBe(1)
      11 |     })

      at Object.toBe (tests/unit/kaikeba.spec.js:8:27)

PASS tests/unit/example.spec.js

Test Suites: 1 failed, 1 passed, 2 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        1.703s
```

测试Vue组件

官方提供了用于单元测试的实用工具库`@vue/test-utils`

检查mounted之后预期结果

使用`mount`或`shallowMount`挂载组件，example.spec.js

```
import { mount } from '@vue/test-utils'

it('renders props.msg when passed', () => {
  const msg = 'new message'
  // 给组件传递属性
  const wrapper = shallowMount(HelloWorld, {
    propsData: { msg }
  })
  // expect(wrapper.text()).toMatch(msg)
  // 查找元素
  const h1 = wrapper.find('h1')
  expect(h1.text()).toBe('new message')
})
```

更新操作通常是异步的，dom更新结果放在`await`语句后面测试

```
<p class="p1" @click="foo = 'baz'">{{foo}}</p>
```

```
test('点击p之后验证更新结果 ', async () => {
  const wrapper = shallowMount(HelloWorld)
  // 模拟点击行为
  const p1 = wrapper.find('.p1')
  // 把变更状态操作放在await后面
  await p1.trigger('click')
  expect(p1.text()).toBe('baz')
})
```

获取自定义组件

```
<comp v-if="foo === 'baz'"></comp>
```

```
components: {
  comp: {
    name: 'comp',
    render(h) {
      return h('div', 'comp')
    }
  },
}
```

```
const comp = wrapper.findComponent({name: 'comp'})
expect(comp.exists()).toBe(true)
```

覆盖率

Jest自带覆盖率，很容易统计我们测试代码是否全面。

package.json里修改jest配置

```
"jest": {
    "collectCoverage": true,
    "collectCoverageFrom": ["src/**/*.{js,vue}"],
}
```

> 若采用独立配置，则修改jest.config.js:
>
> ```
> module.exports = {
>   "collectCoverage": true,
>   "collectCoverageFrom": ["src/**/*.{js,vue}"]
> }
> ```

再次执行npm run test:unit

```
-----------------------------|----------|----------|----------|----------|-------------------|
File                         |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
-----------------------------|----------|----------|----------|----------|-------------------|
All files                    |     0.37 |        0 |     0.95 |     0.37 |                   |
 src                         |        0 |        0 |        0 |        0 |                   |
  App.vue                    |        0 |      100 |      100 |        0 |          17,18,19 |
  main.js                    |        0 |      100 |        0 |        0 | 16,17,19,21,24,28 |
  permission.js              |        0 |        0 |        0 |        0 |... 37,38,43,44,47 |
 src/api                     |        0 |      100 |        0 |        0 |                   |
  user.js                    |        0 |      100 |        0 |        0 |               4,8 |
 src/components              |     9.09 |        0 |    14.29 |     9.09 |                   |
  HelloWorld.vue             |       50 |      100 |       50 |       50 |                24 |
  Notice.vue                 |        0 |      100 |        0 |        0 |    25,31,32,35,37 |
  SvgIcon.vue                |        0 |        0 |        0 |        0 |       22,25,26,28 |
 src/components/Sidebar      |        0 |        0 |        0 |        0 |                   |
  Item.vue                   |        0 |        0 |        0 |        0 |... 19,20,23,24,26 |
```

可以看到HelloWorld.vue的覆盖率是50%

> stmts是语句覆盖率（statement coverage）：是不是每个语句都执行了?
>
> Branch分支覆盖率（branch coverage）：是不是每个if代码块都执行了?
>
> Funcs函数覆盖率（function coverage）：是不是每个函数都调用了?
>
> Lines行覆盖率（line coverage）：是不是每一行都执行了?

通过分析报告可以找到没有覆盖的地方，coverage/lcov-report/index.html

```
14      },
15      data() {
16  2x    return {
17          foo: 'bar'
18        }
19      },
20      components: {
21        comp: {
22          name: 'comp',
23          render(h) {
24            return h('h', 'comp')
25          }
26        },
27      },
28    }
29    </script>
30
```

我们添加一些测试代码

```
test('验证comp组件render结果 ', () => {
  const h = (type, children) => ({type, children})
  const vnode = HelloWorld.components.comp.render(h)
  console.log(vnode);
  expect(vnode).toEqual({type: 'div', children: 'comp'})
})
```

现在覆盖率是100%了，但是整个项目还很糟糕，我们还需要继续努力提高呀！

> Vue组件单元测试cookbook
>
> Vue Test Utils使用指南

E2E测试

借用浏览器的能力，站在用户测试人员的角度，输入框，点击按钮等，完全模拟用户，这个和具体的框架关系不大，完全模拟浏览器行为。

运行E2E测试

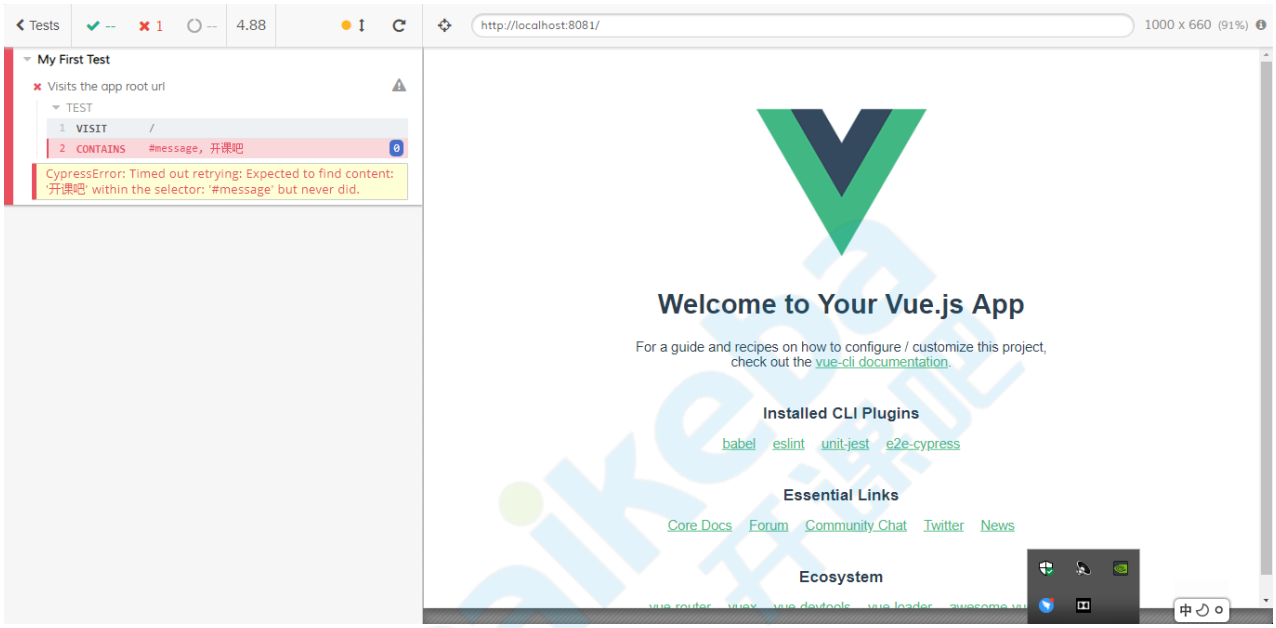
`npm run test:e2e`

修改e2e/spec/test.js

```
// https://docs.cypress.io/api/introduction/api.html

describe('端到端测试，抢测试人员的饭碗', () => {
  it('先访问一下', () => {
    cy.visit('/')
    // cy.contains('h1', 'Welcome to Your Vue.js App')
    cy.contains('span', '开课吧')

  })
})
```



测试未通过，因为没有使用Kaikeba.vue，修改App.vue

```
<div id="app">
    <img alt="Vue logo" src="./assets/logo.png">
    <!-- <HelloWorld msg="Welcome to Your Vue.js App"/> -->
    <Kaikeba></Kaikeba>
</div>

import Kaikeba from './components/Kaikeba.vue'
export default {
  name: 'app',
  components: {
    HelloWorld,Kaikeba
  }
}
```

测试通过~

测试用户点击

```
// https://docs.cypress.io/api/introduction/api.html

describe('端到端测试，抢测试人员的饭碗', () => {
  it('先访问一下', () => {
    cy.visit('/')
    // cy.contains('h1', 'Welcome to Your Vue.js App')
    cy.contains('#message', '开课吧')

    cy.get('button').click()
    cy.contains('span', '按钮点击')

  })
})
```