Vue项目最佳实践

资源

- Vue-CLI 3.0
- vue-element-admin

目标

- 代码规范
- 项目配置
- 权限
- 导航
- 数据mock
- 环境变量
- 测试
- 优化、告警、发布和部署

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

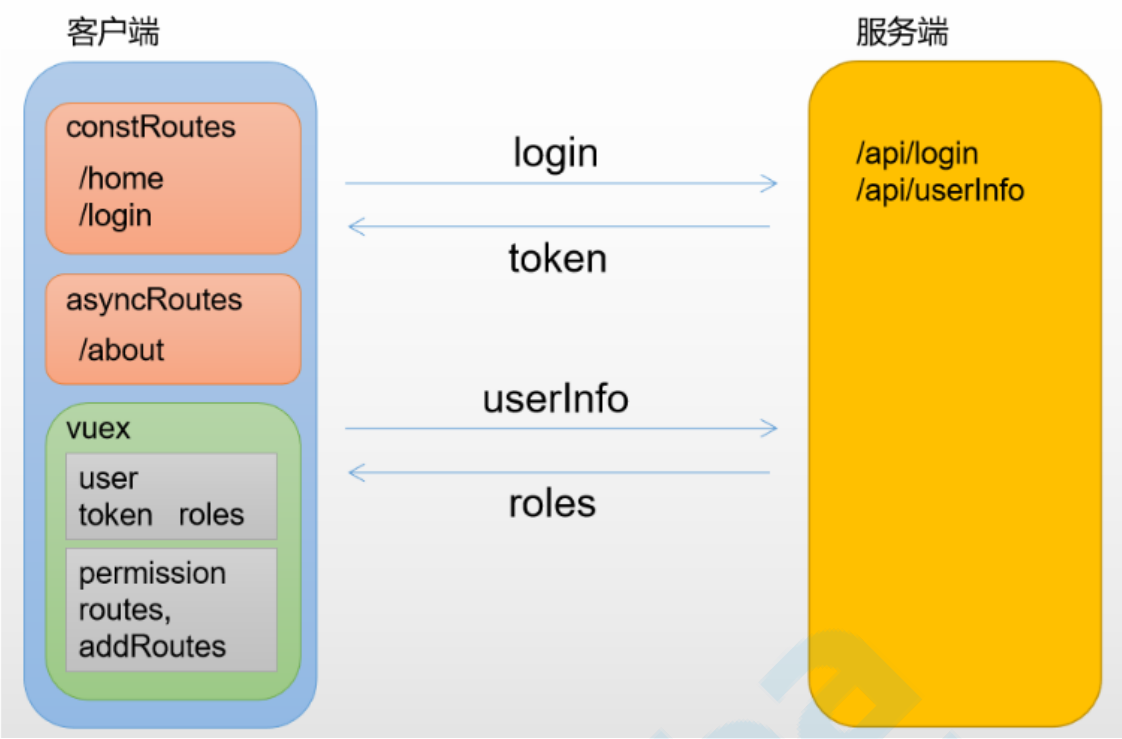
```
# 只能用于服务端
foo=bar
# 可用于客户端
VUE_APP_DONG=dong
```

修改mode选项覆盖模式名称，package.json

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

权限控制

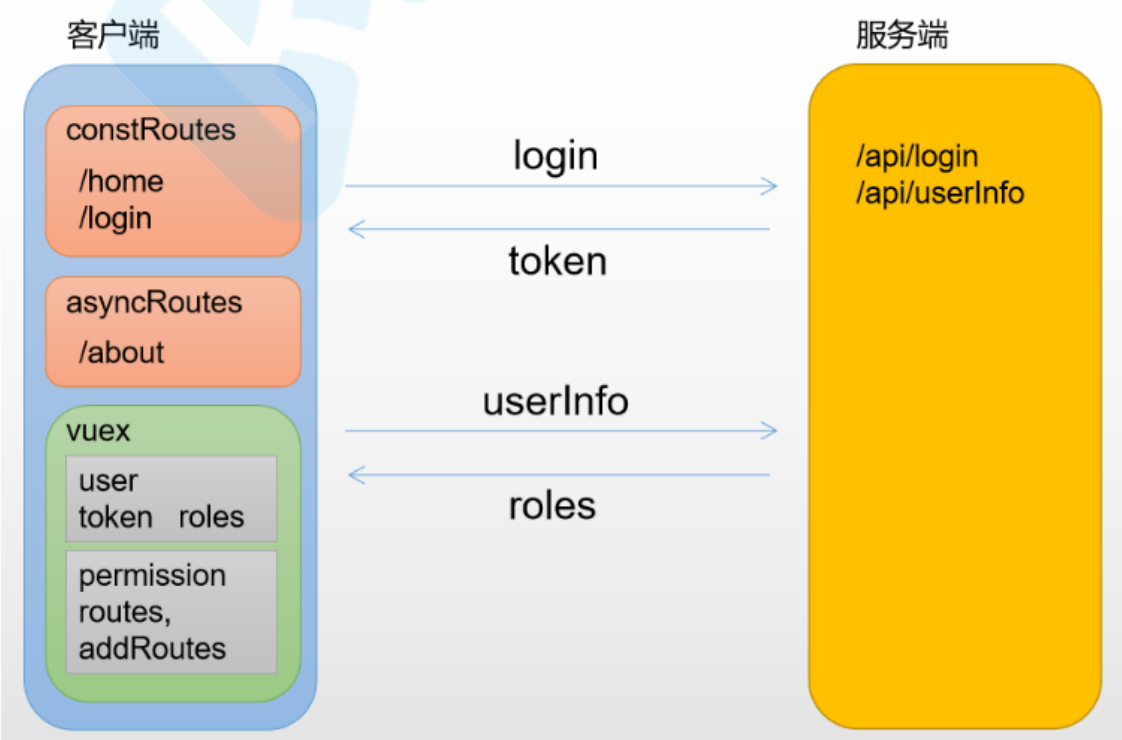
路由分为两种：`constantRoutes`和`asyncRoutes`，前者是默认路由可直接访问，后者中定义的路由需要先登录，获取角色并过滤后动态加入到Router中。



- 路由定义，router/index.js
- 创建用户登录页面，views/Login.vue
- 路由守卫：创建./src/permission.js，并在main.js中引入

用户登录状态维护

维护用户登录状态：路由守卫 => 用户登录 => 获取token并缓存

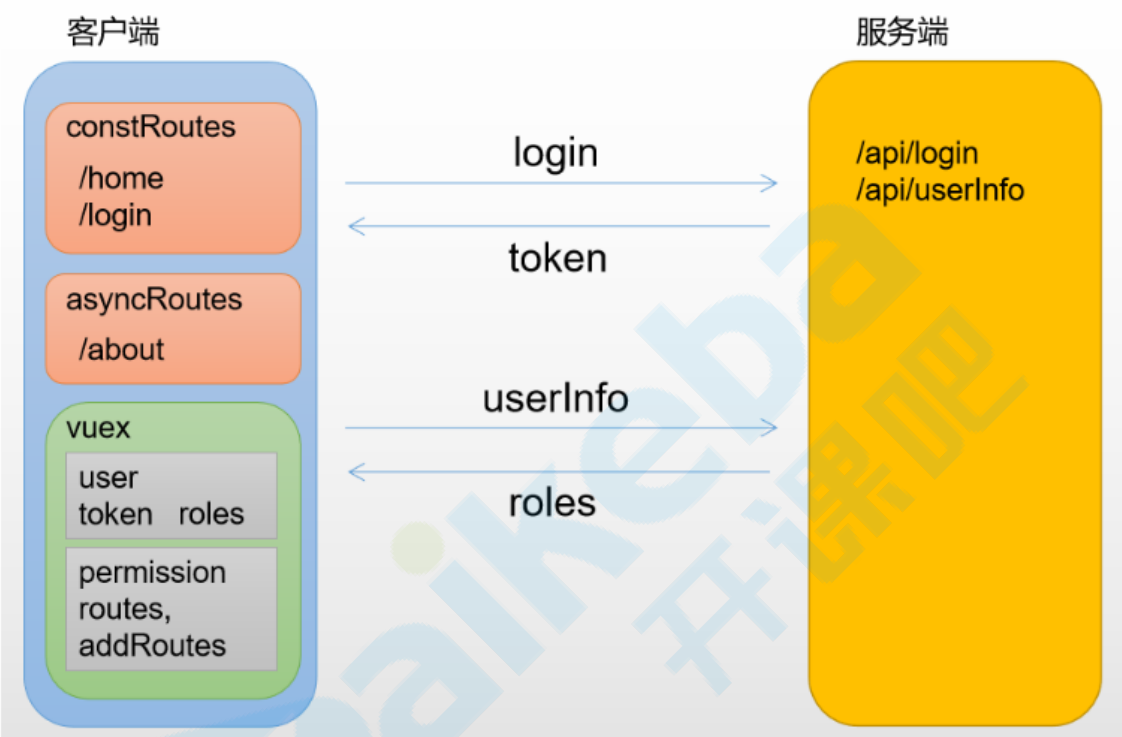


- 路由守卫：src/permission.js

- 请求登录：components/Login.vue
- user模块：维护用户数据、处理用户登录等，store/modules/user.js
- 测试~

用户角色获取和权限路由过滤

登录成功后，请求用户信息获取用户角色信息，然后根据角色过滤asyncRoutes，并将结果动态添加至router



- 维护路由信息，实现动态路由生成逻辑，store/modules/permission.js
- 获取用户角色，判断用户是否拥有访问权限，permission.js

```
// 引入store
import store from './store'

router.beforeEach(async (to, from, next) => {
  // ...
  if (hasToken) {
    if (to.path === '/login') {}
    else {
      // 若用户角色已附加则说明权限以判定，动态路由已添加
      const hasRoles = store.getters.roles && store.getters.roles.length > 0;

      if (hasRoles) {
        // 说明用户已获取过角色信息，放行
        next()
      } else {
        try {
          // 先请求获取用户信息
```

```
          const { roles } = await store.dispatch('user/getInfo')

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

          // 添加至路由器
          router.addRoutes(accessRoutes)

          // 继续路由切换，确保addRoutes完成
          next({ ...to, replace: true })
        } catch (error) {
          // 出错需重置令牌并重新登录（令牌过期、网络错误等原因）
          await store.dispatch('user/resetToken')
          next(`/login?redirect=${to.path}`)
          alert(error || '未知错误')
        }
      }
    }
  } else {
      // 未登录...
  }
})
```

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：

页面中某些按钮、链接有时候需要更细粒度权限控制，这时候可以封装一个指令v-permission，放在需要控制的按钮上，从而实现按钮级别权限控制

- 创建指令，src/directives/permission.js
- 测试，About.vue

> 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" v-permission="['admin',
> 'editor']">
>      用户管理</el-tab-pane>
>  <el-tab-pane label="配置管理" name="second" v-permission="['admin',
> 'editor']">
>      配置管理</el-tab-pane>
>  <el-tab-pane label="角色管理" name="third" v-permission="['admin']">
>      角色管理</el-tab-pane>
>  <el-tab-pane label="定时任务补偿" name="fourth" v-permission="['admin',
> 'editor']">
>      定时任务补偿</el-tab-pane>
> </el-tabs>
> ```
>
> 此时只能使用v-if来实现
>
> ```
> <template>
>   <el-tab-pane v-if="checkPermission(['admin'])">
> </template>
>
> <script>
> export default {
>    methods: {
>      checkPermission(permissionRoles) {
>          return roles.some(role => {
>              return permissionRoles.includes(role);
>          });
>      }
>    }
> }
> </script>
> ```
>
> 自定义指令参考

自动生成导航菜单

导航菜单是根据路由信息并结合权限判断而动态生成的。它需要对应路由的多级嵌套，所以要用到递归组件。

- 创建侧边栏组件，components/Sidebar/index.vue
- 创建侧边栏菜项目组件，layout/components/Sidebar/SidebarItem.vue
- 创建侧边栏菜单项组件，layout/components/Sidebar/Item.vue

数据交互

数据交互流程：

api服务 => axios请求 => 本地mock/线上mock/服务器api

封装request

对axios做一次封装，统一处理配置、请求和响应拦截。

> 安装axios: `npm i axios -S`

- 创建@/utils/request.js
- 设置VUE_APP_BASE_API环境变量，创建.env.development文件
- 编写服务接口，创建@/api/user.js

数据mock

数据模拟两种常见方式，**本地mock**和**线上esay-mock**

本地mock：利用webpack-dev-server提供的before钩子可以访问express实例，从而定义接口

- 修改vue.config.js，给devServer添加相关代码
- 调用接口，@/store/modules/user.js

线上esay-mock

诸如easy-mock这类线上mock工具优点是使用简单，mock工具库也比较强大，还能根据swagger规范生成接口。

使用步骤：

1. 登录easy-mock

> 若远程不可用，可以搭建本地easy-mock服务（nvm + node + redis + mongodb）
>
> 先安装node 8.x、redis和mongodb
>
> 启动命令：

    - 切node v8：`nvm list`，`nvm use 8.16.0`
    - 起redis：`redis-server`
    - 起mongodb：`mongod`
    - 起easy-mock项目：`npm run dev`

2. 创建一个项目
3. 创建需要的接口

```
// 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 ''
    }
  }
}

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

4. 调用：修改base_url，.env.development

```
VUE_APP_BASE_API = 'http://localhost:7300/mock/5e9032aab92b8c71eb235ad5'
```

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

- 添加代理配置，vue.config.js
- 创建一个独立接口服务器，~/server/index.js

> 代码下载：
> https://github.com/57code/vue-study/tree/step-7
> 命令用下面这个：
>
> ```
> git fetch origin step-7
> git reset --hard step-7
> ```
>
> 注意切换一个新分支

项目测试

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。

组件的单元测试有很多好处：

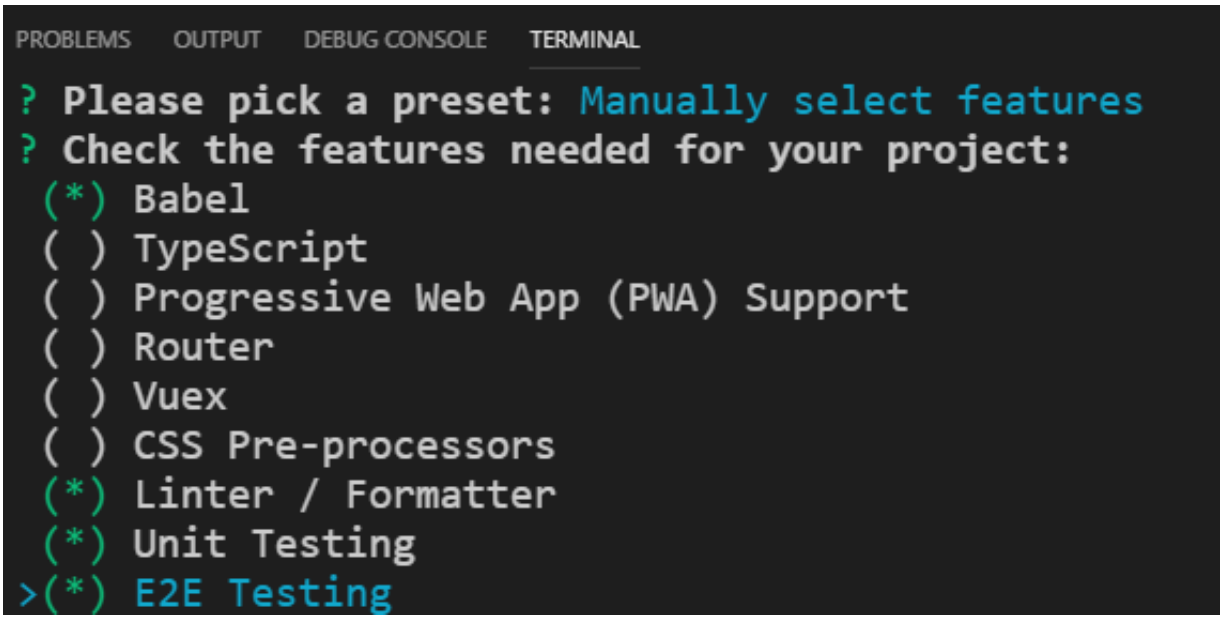
- 提供描述组件行为的文档
- 节省手动测试的时间
- 减少研发新特性时产生的 bug
- 改进设计
- 促进重构

准备工作

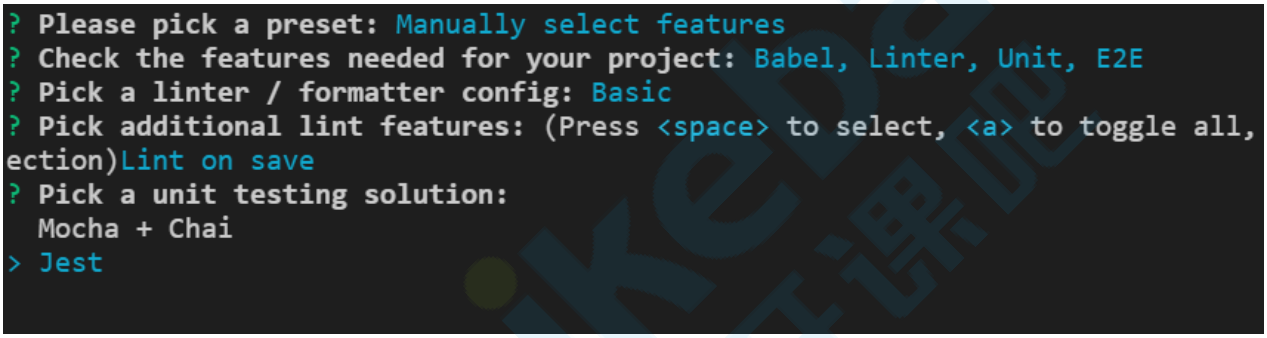
在vue-cli中，预置了Mocha+Chai和Jest两套单测方案，我们的演示代码使用Jest，它们语法基本一致

新建vue项目时

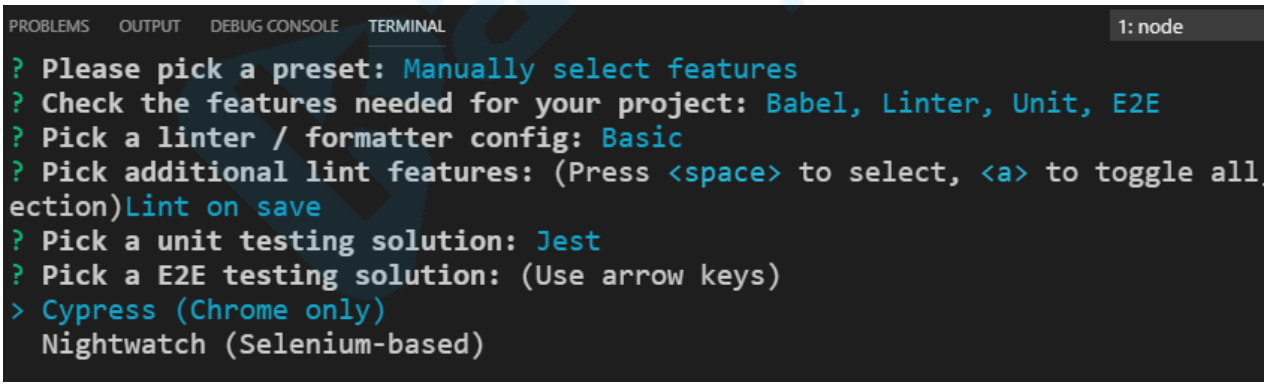
- 选择特性`Unit Testing`和`E2E Testing`



- 单元测试解决方案选择：`Jest`



- 端到端测试解决方案选择：Cypress



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('Kaikeba', () => {
    // 测试用例 test case
    it('测试add函数', () => {
        // 断言 assert
        expect(add(1, 3)).toBe(3)
        expect(add(1, 3)).toBe(4)
        expect(add(-2, 3)).toBe(1)
    })
})
```

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
```

断言API简介

- `describe`： 定义一个测试套件
- `it`：定义一个测试用例

- `expect`：断言的判断条件

  > 这里面仅演示了toBe，更多断言API

测试Vue组件

vue官方提供了用于单元测试的实用工具库`@vue/test-utils`

- 创建一个vue组件components/Kaikeba.vue
- 测试该组件，test/unit/kaikeba.spec.js

```
import Kaikeba from '@/components/Kaikeba.vue'

describe('Kaikeba.vue', () => {
  // 检查组件选项
  it('要求设置created生命周期', () => {
    expect(typeof Kaikeba.created).toBe('function')
  })
  it('message初始值是vue-test', () => {
    // 检查data函数存在性
    expect(typeof Kaikeba.data).toBe('function')
  // 检查data返回的默认值
    const defaultData = Kaikeba.data()
    expect(defaultData.message).toBe('vue-test')
  })
})
```

```
FAIL tests/unit/kaikeba.spec.js
 ● KaikebaComp › 初始data是vue-text

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

   Expected: "hello!"
   Received: "vue-text"

     33 |
     34 |     const defaultData = KaikebaComp.data()
   > 35 |     expect(defaultData.message).toBe('hello!')
        |                                 ^
     36 |   })
     37 |
     38 | //   // 检查 mount 中的组件实例

     at Object.toBe (tests/unit/kaikeba.spec.js:35:33)

PASS tests/unit/example.spec.js
```

检查mounted之后预期结果

使用@vue/test-utils挂载组件

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

it("mount之后测data是开课吧", () => {
    const wrapper = mount(Kaikeba);
    expect(wrapper.vm.message).toBe("开课吧");
});

it("按钮点击后", () => {
    const wrapper = mount(KaikebaComp);
    wrapper.find("button").trigger("click");
    // 测试数据变化
    expect(wrapper.vm.message).toBe("按钮点击");
    // 测试html渲染结果
    expect(wrapper.find("span").html()).toBe("<span>按钮点击</span>");
    // 等效的方式
    expect(wrapper.find("span").text()).toBe("按钮点击");
});
```

测试覆盖率

Jest自带覆盖率，很容易统计我们测试代码是否全面。如果用的mocha，需要使用istanbul来统计覆盖率。

- 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}"]
> }
> ```

- 在此执行npm run test:unit

```
> vue-cli-service test:unit

 PASS  tests/unit/kaikeba.spec.js
 PASS  tests/unit/example.spec.js
----------------|----------|----------|----------|----------|-------------------|
File            |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------------|----------|----------|----------|----------|-------------------|
All files       |    15.79 |      100 |        0 |    15.79 |                   |
 src            |        0 |      100 |        0 |        0 |                   |
  main.js       |        0 |      100 |        0 |        0 |    1,2,3,4,6,8,11 |
  router.js     |        0 |      100 |        0 |        0 |       1,2,3,5,22  |
  store.js      |        0 |      100 |      100 |        0 |             1,2,4 |
 src/components |      100 |      100 |      100 |      100 |                   |
  Kaikeba.vue   |      100 |      100 |      100 |      100 |                   |
 src/views      |        0 |      100 |      100 |        0 |                   |
  Home.vue      |        0 |      100 |      100 |        0 |                10 |
----------------|----------|----------|----------|----------|-------------------|

Test Suites: 2 passed, 2 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        1.653s
Ran all test suites.
➜  vue-test git:(master) x
```

> %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?

可以看到我们kaikeba.vue的覆盖率是100%，我们修改一下代码

```
<template>
  <div>
    <span>{{ message }}</span>
    <button @click="changeMsg">点击</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: "vue-text",
      count: 0
    };
  },
  created() {
    this.message = "开课吧";
  },
  methods: {
    changeMsg() {
      if (this.count > 1) {
```

```
        this.message = "count大于1";
      } else {
        this.message = "按钮点击";
      }
    },
    changeCount() {
      this.count += 1;
    }
  }
};
</script>
```

```
PASS tests/unit/kaikeba.spec.js
PASS tests/unit/example.spec.js
----------------|----------|----------|----------|----------|-------------------|
File            |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------------|----------|----------|----------|----------|-------------------|
All files       |    18.18 |       50 |        0 |    18.18 |                   |
 src            |        0 |      100 |        0 |        0 |                   |
  main.js       |        0 |      100 |        0 |        0 |   1,2,3,4,6,8,11  |
  router.js     |        0 |      100 |        0 |        0 |      1,2,3,5,22   |
  store.js      |        0 |      100 |      100 |        0 |          1,2,4    |
 src/components |    66.67 |       50 |      100 |    66.67 |                   |
  Kaikeba.vue   |    66.67 |       50 |      100 |    66.67 |           22,28   |
 src/views      |        0 |      100 |      100 |        0 |                   |
  Home.vue      |        0 |      100 |      100 |        0 |             10    |
----------------|----------|----------|----------|----------|-------------------|

Test Suites: 2 passed, 2 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        2.08s
Ran all test suites
```

现在的代码，依然是测试没有报错，但是覆盖率只有66%了，而且没有覆盖的代码行数，都标记了出来，继续努力加测试吧

> 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

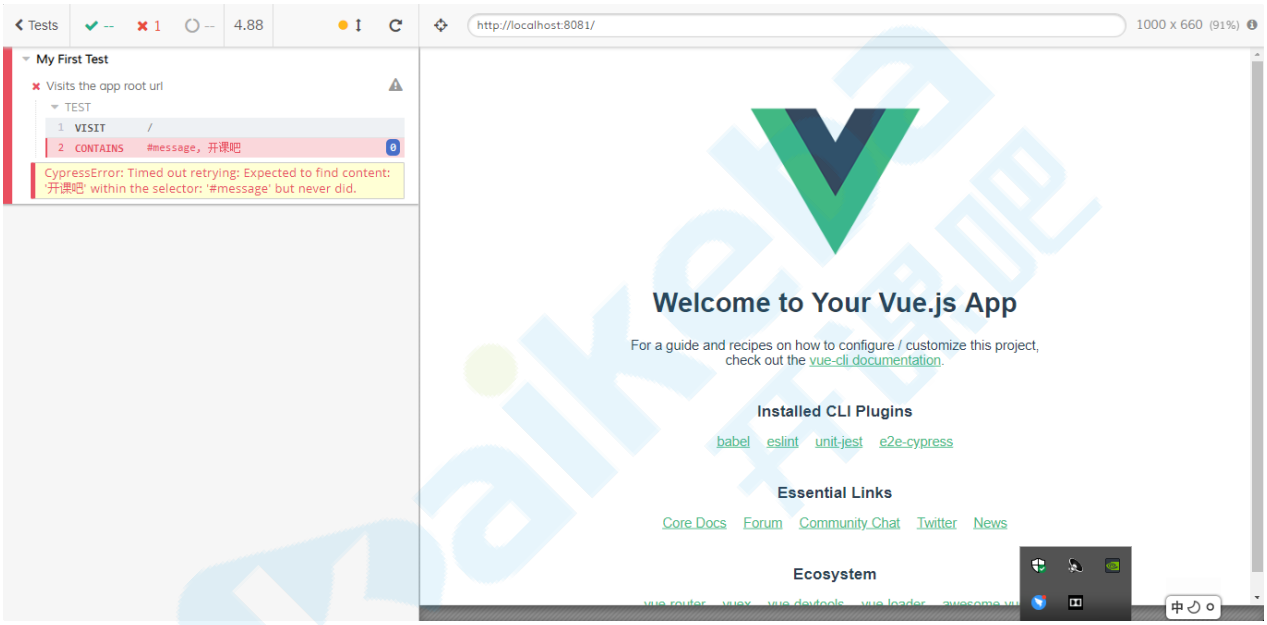
```
npm run test:e2e
```

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')

  })
})
```