Vue 单元测试初探

周炼赤

2019-12-17

目录

一、缘由、概念和规范

二、Vue Test Utils

目录

一、缘由、概念和规范

二、Vue Test Utils

一、缘由、概念和规范

➢ Why（为什么要单元测试）

业务很重要，除了业务，还有效率。

没有效率，就没有生产力。

不管在什么领域，我们在提效道路上一定会经历以下几个阶段：

- 规范标准化
- 机器自动化
- 系统平台化
- 人工智能化

一、缘由、概念和规范

➢ What（什么是单元测试）

单元测试，是指对软件中的最小可测试单元进行检查和验证，同时，单元测试的编写者必须是作者本人。

拥有单元测试的程序有以下几个好处：验证、设计、文档、回归。

一、缘由、概念和规范

1. 它是一种验证行为

程序中的每一项功能都是测试来验证它的正确性。

它为开发提供支援。就算是项目后期，我们也可以轻松的增加功能或更改程序结构，而不用担心这个过程中会破坏重要的东西。而且它为代码的重构提供了保障。

2. 它是一种设计行为

编写单元测试将使我们以调用者的角度来观察和思考。

特别是测试驱动（Test Driven），迫使我们把程序设计成易于调用和可测试的，有利于提高代码耦合性。

一、缘由、概念和规范

3. 它是一种编写文档的行为

单元测试是一种展示函数或类如何使用的最佳文档。

这份文档是可编译、可运行的，并且它保持最新，永远与代码同步。

4. 它具有回归性

自动化的单元测试避免了代码出现回归，编写完成之后，可以随时随地快速运行测试。

一、缘由、概念和规范

➢ How（单元测试用例设计）

首先，任何一个单元测试都应该包含：

- 正常输入
  - 离散覆盖参数值域
- 边界输入
  - 空值验证
  - 零值验证
  - 最大/小值验证
- 非法输入
  - 入参数据类型非法
  - 内存溢出验证

一、缘由、概念和规范

➢ How（单元测试用例设计）

其次，还有一些讲究：

- 幂等
- Mock——构造数据就是构造输入端的过程
  - 环境（JSDOM 等，保证离线场景下可以验证依赖浏览器 API 的程序逻辑）
  - 人机交互事件（比如 React 中的 enzyme）
  - 模块/第三方包（比如 axios 的请求拦截）
  - 类/函数（跟踪一个类/函数的使用/调用过程）
- 黑盒 → 白盒（覆盖率是衡量单元测试质量的核心指标）

一、缘由、概念和规范

➢ Auto（自动化单元测试）

Jest，一款自动化单元测试运行器，它基本上满足了前端单元测试的所有测试需求，而且零配置。有了 Jest，我们在写单元测试用例的配置成本已经很低了，所以，单元测试的成本，更多的是编写测试用例上。

要提高测试用例编写效率，我们主要从几个方向来提高：

- 模板：定制标准用例模板，让开发者做填空题，而非选择题。
- 规范：制定单元测试开发规范，帮助开发者写出统一的单元测试用例，方便维护。
- 渐进式：借助 bugfix/feature addition 过程逐步完善测试用例，最大化减轻前期压力。

目录

一、缘由、概念和规范

二、Vue Test Utils

二、 Vue Test Utils

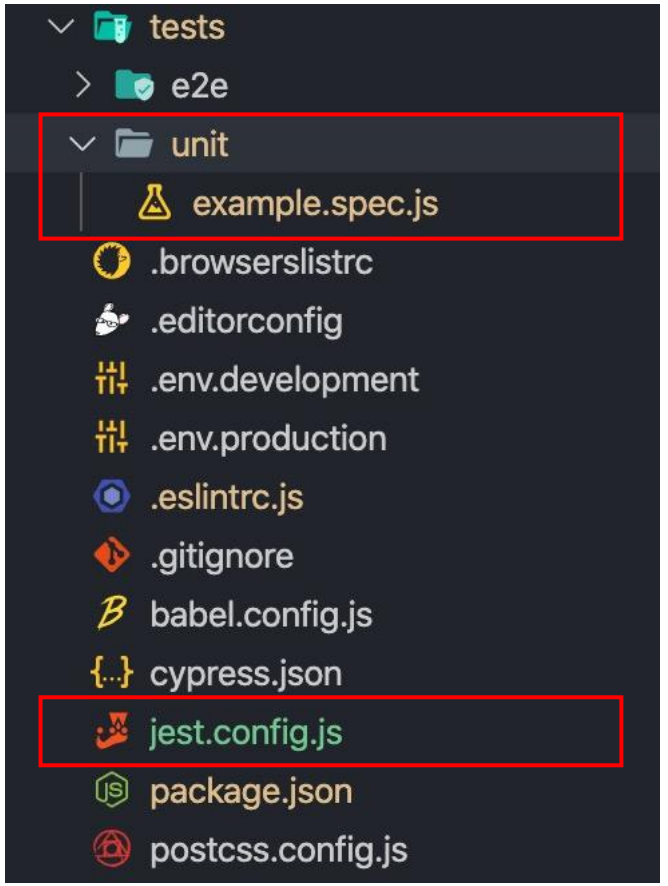
➢ 介绍

Vue Test Utils 是 Vue.js 官方的单元测试实用工具库。

➢ 安装

```
npm install @vue/test-utils

npm install @vue/cli-plugin-unit-jest
```




```
"scripts": {
  "serve": "vue-cli-service serve",
  "build": "vue-cli-service build",
  "lint": "eslint --ext .js,.vue src",
  "lint:fix": "eslint --fix --ext .js,.vue src",
  "test:e2e": "vue-cli-service test:e2e",
  "test:unit": "vue-cli-service test:unit"
```

二、 Vue Test Utils

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

import { shallowMount } from "@vue/test-utils"
import HelloWorld from "@/components/HelloWorld.vue"

describe("HelloWorld.vue", () => {
  it("renders props.msg when passed", () => {
    const msg = "new message"
    const wrapper = shallowMount(HelloWorld, {
      propsData: { msg }
    })
    expect(wrapper.text()).toMatch(msg)
  })
})
```

二、 Vue Test Utils

➢ 挂载组件

```
// counter.js

export default {
  template: `
    <div>
      <span class="count">{{ count }}</span>
      <button @click="increment">增加</button>
    </div>
  `,
  data() {
    return {
      count: 0
    }
  },
  methods: {
    increment() {
      this.count++
    }
  }
}
```

```
// test.js

// 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)

// 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)
```

二、 Vue Test Utils

➢ 测试组件渲染出来的 HTML

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

describe('Counter', () => {
  // 现在挂载组件，你便得到了这个包裹器
  const wrapper = mount(Counter)

  it('renders the correct markup', () => {
    expect(wrapper.html()).toContain('<span class="count">0</span>')
  })

  // 也便于检查已存在的元素
  it('has a button', () => {
    expect(wrapper.contains('button')).toBe(true)
  })
})
```

现在运行 npm test 进行测试。就看得到测试通过。

二、 Vue Test Utils

➢ 模拟用户交互

当用户点击按钮时，计数器应该递增。为了模拟这一行为，首先需要通过 wrapper.find() 定位该按钮，此方法返回一个该按钮元素的包裹器。
然后我们能够通过对该按钮包裹器调用 .trigger() 来模拟点击。

```
it('button click should increment the count', () => {
  expect(wrapper.vm.count).toBe(0)
  const button = wrapper.find('button')
  button.trigger('click')
  expect(wrapper.vm.count).toBe(1)
})
```

二、 Vue Test Utils

➢ 常用技巧 —— 明白要测试的是什么

对于 UI 组件来说，我们不推荐一味追求行级覆盖率，因为它会导致我们过分关注组件的内部实现细节，从而导致琐碎的测试。

取而代之的是，我们推荐把测试撰写为**断言你的组件的公共接口，并在一个黑盒内部处理它。**

一个简单的测试用例将会断言一些输入（用户的交互或 prop 的改变）提供给某组件之后是否导致预期结果（渲染结果或触发自定义事件）。

比如，对于每次点击按钮都会将计数加一的 Counter 组件来说，其测试用例将会模拟点击并断言渲染结果会加 1。该测试并没有关注 Counter 如何递增数值，而只关注其输入和输出。

二、 Vue Test Utils

➢ 常用技巧 —— 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。

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：

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

const wrapper = shallowMount(Component)
wrapper.vm // 挂载的 Vue 实例
```

二、 Vue Test Utils

➢ 常用技巧 —— 断言触发的事件

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)

/*
`wrapper.emitted()` 返回以下对象:
{ foo: [[], [123]] }
*/
```

然后你可以基于这些数据来设置断言：

```
// 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])
```

二、 Vue Test Utils

➢ 常用技巧 —— 操作组件状态

你可以在包裹器上用 setData 或 setProps 方法直接操作组件状态：

```
wrapper.setData({ count: 10 })

wrapper.setProps({ foo: 'bar' })
```

二、 Vue Test Utils

➢ 常用技巧 —— 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'
  }
})
```

二、 Vue Test Utils

➢ 常用技巧 —— 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
  }
})
```

二、 Vue Test Utils

➢ 常用技巧 —— 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']
})
```

二、 Vue Test Utils

➢ 测试键盘、鼠标等其它 DOM 事件

Wrapper 暴露了一个 trigger 方法。它可以用来触发 DOM 事件。
以鼠标点击事件为例：

```
const wrapper = mount(MyButton)

wrapper.trigger('click')
```

另外，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。

```
const wrapper = mount(MyComponent)

wrapper.find('button').trigger('click')
```

二、 Vue Test Utils

➢ 选择一个测试运行器

测试运行器 (test runner) 就是运行测试的程序。
主流的 JavaScript 测试运行器有很多，Vue Test Utils 都能够支持，它是与测试运行器无关的。
选用测试运行器的时候也需要考虑一些事项：功能集合、性能和对单文件组件预编译的支持等。
Vue 官方推荐其中的两个测试运行器：

- Jest 功能最全，所需的配置最少。默认安装了 JSDOM，内置断言且命令行的用户体验非常好。不过对单文件组件的支持不够完备。Vue 官方已经创建了 vue-jest 预处理器来处理最常见的单文件组件特性，但仍不是 vue-loader 100% 的功能。
- mocha-webpack 是一个 webpack + Mocha 的包裹器，同时包含了更顺畅的接口和侦听模式。我们能够通过 webpack + vue-loader 得到完整的单文件组件支持，但这本身需要很多配置。

二、 Vue Test Utils

➢ 浏览器环境

Vue Test Utils 依赖浏览器环境。技术上讲你可以将其运行在一个真实的浏览器，但是我们并不推荐，因为在不同的平台上都启动真实的浏览器是很复杂的。我们推荐取而代之的是用 JSDOM 在 Node 虚拟浏览器环境运行测试。

Jest 测试运行器自动设置了 JSDOM。对于其它测试运行器来说，你可以在你的测试入口处使用 jsdom-global 手动设置 JSDOM。

好在，@vue/cli-plugin-unit-jest 已经帮我们做好了这一切。

二、 Vue Test Utils

➢ 更多

- API (mount()、shallowMount()、render() 等)
- Wrapper
  - 属性 (vm、element 等)
  - 方法 (find()、is()、setData()、text() 等)
- 挂载选项 (stubs、mocks、propsData 等)
- 其他

总结

- 单元测试的一些缘由、概念和规范。
- Vue Test Utils 与 Jest 的结合。

Thanks