58同城

对于 Vue 应用的测试调研分析

贾磊

目录

DEEP
THINKING
深入思考

一、为什么需要测试

自动化测试能够预防无意引入的 bug，并鼓励开发者将应用分解为**可测试、可维护的函数、模块、类和组件**。

这能够帮助你和你的团队更快速、自信地构建复杂的应用。

二、何时测试

越早越好!

尽快开始编写测试。拖得越久，应用就会有越多的依赖和复杂性，想要开始添加测试也就越困难。

与开发前的技术实施方案有异曲同工之妙，前期技术方案拆解的越细致，开发过程中产生的问题相对越少，效率也相对较高；

测试也是如此，前期测试用例罗列的越全面，开发过程中就会思考的更多，代码就会更健壮，（显而易见的）bug 数量就会相对减少，从而提高交付质量。

这也是我们自测验证的一种有效手段，更是一种潜移默化中提高自身能力的思想。

三、测试的类型

- **单元测试**：检查给定函数、类或组合式函数的输入是否产生预期的输出或副作用。
- **组件测试**：检查你的组件是否正常挂载和渲染、是否可以与之互动，以及表现是否符合预期。这些测试比单元测试导入了更多的代码，更复杂，需要更多时间来执行。
- **端到端测试**：检查跨越多个页面的功能，并对生产构建的应用进行实际的网络请求。这些测试通常涉及到建立一个数据库或其他后端。

每种测试类型在你的应用的测试策略中都发挥着作用，保护你免受不同类型的问题的影响。

四、单元测试——定义

编写单元测试是为了验证小的、独立的代码单元是否按预期工作。一个单元测试通常覆盖一个单个函数、类、组合式函数或模块。单元测试侧重于逻辑上的正确性，只关注应用整体功能的一小部分。他们可能会模拟你的应用环境的很大一部分（如初始状态、复杂的类、第三方模块和网络请求）。

一般来说，单元测试将捕获函数的业务逻辑和逻辑正确性的问题。

以这个 increment 函数为例：

因为它很独立，可以很容易地调用 increment 函数并断言它是否返回了所期望的内容，所以我们将编写一个单元测试。

如果任何一条断言失败了，那么问题一定是出在 increment 函数上。

```
// helpers.js
export function increment(current, max = 10) {
  if (current < max) {
    return current + 1
  }
  return current
}
```

四、单元测试——单个函数


```
// helpers.test.js
// ---------------------------------------- vitest ------------
// import { increment } from './helpers';
// import { describe, test, expect } from 'vitest';
// describe('increment', () => {
//   test('increments the current number by 1', () => {
//     expect(increment(0, 10)).toBe(1)
//   })
//   test('does not increment the current number over the max', () => {
//     expect(increment(10, 10)).toBe(10)
//   })
//   test('has a default max of 10', () => {
//     expect(increment(10)).toBe(10)
//   })
// })
// ---------------------------------------- jest --------------
const increment = require('./helpers');
test('执行函数 increment(0, 10)， 期待得到 1', () => {
  expect(increment(0, 10)).toBe(1);
})
test('执行函数 increment(10, 10)，期待得到最大值 10', () => {
  expect(increment(10, 10)).toBe(10);
})
test('执行函数 increment(10)，期待得到默认值的最大值 10', () => {
  expect(increment(10)).toBe(10);
})
```

四、单元测试——单个函数

DEEP THINKING 深入思考

```
# 执行 npm run test
# ---------------------------------------- vitest -----------------------------
# > vite-demo@0.0.0 test /Users/a58/Test/前端文章/测试/vite-demo
# > vitest
# DEV v0.29.2 /Users/a58/Test/前端文章/测试/vite-demo
# ✓ src/helpers.spec.js (3)
# Test Files 1 passed (1)
# Tests 3 passed (3)
# Start at 09:45:43
# Duration 304ms (transform 48ms, setup 0ms, collect 35ms, tests 3ms)
# PASS Waiting for file changes...
# press h to show help, press q to quit
#---------------------------------------- jest -----------------------------
# jest
> jest-demo@1.0.0 test /Users/a58/Test/前端文章/测试
> jest
PASS jest-demo/helpers.test.js
✓ 执行函数 increment(0, 10)， 期待得到 1 (1 ms)
✓ 执行函数 increment(10, 10)，期待得到最大值 10
✓ 执行函数 increment(10)，期待得到默认值的最大值 10
Test Suites: 1 passed, 1 total
Tests: 3 passed, 3 total
Snapshots: 0 total
Time: 0.224 s, estimated 1 s
Ran all test suites.
```

四、单元测试

如前所述，单元测试通常适用于独立的业务逻辑、组件、类、模块或函数，不涉及 UI 渲染、网络请求或其他环境问题。

这些通常是纯 JavaScript/TypeScript 模块。一般来说，在 Vue 应用中为业务逻辑编写单元测试与使用其他框架的应用没有明显区别。

但有两种情况，你必须对 Vue 的特定功能进行单元测试:

1. **组合式函数**
2. **组件**

四、单元测试——组件的单元测试

DEEP THINKING 深入思考

组件的单元测试

一个组件可以通过两种方式测试:

1. 白盒：单元测试

白盒测试知晓一个组件的实现细节和依赖关系。它们更专注于将组件进行更 **独立** 的测试。这些测试通常会涉及到模拟一些组件的部分子组件，以及设置插件的状态和依赖性（例如 Vuex）。

2. 黑盒：组件测试

黑盒测试不知晓一个组件的实现细节。这些测试尽可能少地模拟，以测试组件在整个系统中的集成情况。它们通常会渲染所有子组件，因而会被认为更像一种“集成测试”。

四、单元测试——推荐方案

推荐方案

- Vitest

因为由 create-vue 创建的官方项目配置是基于 Vite 的，所以推荐使用一个可以利用同一套 Vite 配置和转换管道的单元测试框架。Vitest 正是一个针对此目标设计的单元测试框架，它由 Vue / Vite 团队成员开发和维护。在 Vite 的项目集成它会非常简单，而且速度非常快。

其他选择

- Peeky 是另一速度极快的单元测试运行器，对 Vite 集成提供第一优先级支持。它也是由 Vue 核心团队成员创建的，并提供了一个基于图形用户界面（GUI）的测试界面。
- Jest 是一个广受欢迎的单元测试框架，并可通过 vite-jest 这个包在 Vite 中使用。不过，只推荐你在已有一套 Jest 测试配置、且需要迁移到基于 Vite 的项目时使用它，因为 Vitest 提供了更无缝的集成和更好的性能。

五、组件测试——建议测试方向

在 Vue 应用中，主要用组件来构建用户界面。因此，当验证应用的行为时，组件是一个很自然的独立单元。从粒度的角度来看，组件测试位于单元测试之上，可以被认为是集成测试的一种形式。你的 Vue 应用中大部分内容都应该由组件测试来覆盖，**建议每个 Vue 组件都应有自己的组件测试文件**。

组件测试应该捕捉组件中的 prop、事件、提供的插槽、样式、CSS class 名、生命周期钩子，和其他相关的问题。

组件测试不应该模拟子组件，而应该像用户一样，**通过与组件互动来测试组件和其子组件之间的交互**。例如，组件测试应该像用户那样点击一个元素，而不是编程式地与组件进行交互。

组件测试主要需要关心组件的公开接口而不是内部实现细节。对于大部分的组件来说，公开接口包括触发的事件、prop 和插槽。当进行测试时，请记住，**测试这个组件做了什么，而不是测试它是怎么做到的。**

五、组件测试——推荐的做法

- **推荐的做法**
  - 对于 视图 的测试：根据输入 prop 和插槽断言渲染输出是否正确。
  - 对于 交互 的测试：断言渲染的更新是否正确或触发的事件是否正确地响应了用户输入事件。

- **应避免的做法**

不要去断言一个组件实例的私有状态或测试一个组件的私有方法。测试实现细节会使测试代码太脆弱，因为当实现发生变化时，它们更有可能失败并需要更新重写。

组件的最终工作是渲染正确的 DOM 输出，所以专注于 DOM 输出的测试提供了足够的正确性保证（如果你不需要更多其他方面测试的话），同时更加健壮、需要的改动更少。

不要完全依赖快照测试。断言 HTML 字符串并不能完全说明正确性。应当编写有意图的测试。

如果一个方法需要测试，把它提取到一个独立的实用函数中，并为它写一个专门的单元测试。如果它不能被直截了当地抽离出来，那么对它的调用应该作为交互测试的一部分。

五、组件测试——推荐方案

DEEP THINKING 深入思考

推荐方案

- Vitest 对于组件和组合式函数都采用**无头渲染**的方式 (例如 VueUse 中的 useFavicon 函数)。组件和 DOM 都可以通过 @testing-library/vue 来测试。
- Cypress 组件测试 会预期其准确地渲染样式或者触发原生 DOM 事件。可以搭配 @testing-library/cypress 这个库一同进行测试。

Vitest 和基于浏览器的运行器之间的主要区别是速度和执行上下文。简而言之，基于浏览器的运行器，如 Cypress，可以捕捉到基于 Node 的运行器（如 Vitest）所不能捕捉的问题（比如样式问题、原生 DOM 事件、Cookies、本地存储和网络故障），但基于浏览器的运行器比 Vitest 慢几个数量级，因为它们要执行打开浏览器，编译样式表以及其他步骤。Cypress 是一个基于浏览器的运行器，支持组件测试。请阅读 Vitest 文档的“比较”这一章 了解 Vitest 和 Cypress 最新的比较信息。

*注：无头渲染，组件库的一种设计理念—— 比如 Headless UI，强调组件的状态及交互逻辑，不提供标签和样式（即不提供渲染的界面样式），交予使用者更灵活地定制化适合自己的界面样式。

五、组件测试——组件挂载库

DEEP THINKING 深入思考

组件挂载库

组件测试通常涉及到单独挂载被测试的组件，触发模拟的用户输入事件，并对渲染的 DOM 输出进行断言。有一些专门的工具库可以使这些任务变得更简单。

@testing-library/vue 是一个 Vue 的测试库，专注于测试组件而不依赖其他实现细节。因其良好的设计使得代码重构也变得非常容易。它的指导原则是，测试代码越接近软件的使用方式，它们就越值得信赖。

@vue/test-utils 是官方的底层组件测试库，用来提供给用户访问 Vue 特有的 API。@testing-library/vue 也是基于此库构建的。

我们推荐使用 @testing-library/vue 测试应用中的组件，因为它更匹配整个应用的测试优先级。只有在你构建高级组件、并需要测试内部的 Vue 特有 API 时再使用 @vue/test-utils。

其他选择

Nightwatch 是一个端到端测试运行器，支持 Vue 的组件测试。 (Nightwatch v2 版本的 示例项目)

六、端到端（E2E）测试——定义

虽然单元测试为所写的代码提供了一定程度的验证，但单元测试和组件测试在部署到生产时，对应用整体覆盖的能力有限。因此，端到端测试针对的可以说是应用最重要的方面：当用户实际使用你的应用时发生了什么。

端到端测试的重点是多页面的应用表现，针对你的应用在生产环境下进行网络请求。他们通常需要建立一个数据库或其他形式的后端，甚至可能针对一个预备上线的环境运行。

端到端测试通常会捕捉到路由、状态管理库、顶级组件（常见为 App 或 Layout）、公共资源或任何请求处理方面的问题。如上所述，它们可以捕捉到单元测试或组件测试无法捕捉的关键问题。

端到端测试不导入任何 Vue 应用的代码，而是**完全依靠在真实浏览器中**浏览整个页面来测试你的应用。

端到端测试验证了你的应用中的许多层。可以在你的本地构建的应用中，甚至是一个预上线的环境中运行。针对预上线环境的测试不仅包括你的前端代码和静态服务器，还包括所有相关的后端服务和基础设施。

六、端到端（E2E）测试——选择测试框架时的注意事项

选择一个端到端测试解决方案

因为不可靠且拖慢了开发过程，市面上对 Web 上的端到端测试的评价并不好，但现代端到端工具已经在创建更可靠、更有用和交互性更好的测试方面取得了很大进步。在选择端到端测试框架时，以下小节会为你给应用选择测试框架时需要注意的事项提供一些指导。

- 跨浏览器测试
- 更快的反馈
- 第一优先级的调试体验
- 无头模式下的可见性

六、端到端（E2E）测试——推荐方案

DEEP THINKING 深入思考

推荐方案

- Cypress

总的来说， Cypress 提供了最完整的端到端解决方案，其具有信息丰富的图形界面、出色的调试性、内置断言和存根、抗剥落性、并行化和快照等诸多特性。而且如上所述，它还提供对 组件测试 的支持。不过，它只支持测试基于 Chromium 的浏览器和 Firefox。

其他选项

Playwright 也是一个非常好的端到端测试解决方案，支持测试范围更广的浏览器品类（主要是 WebKit 型的）。查看这篇文章 《为什么选择 Playwright》 了解更多细节。

Nightwatch v2 是一个基于 Selenium WebDriver 的端到端测试解决方案。它的浏览器品类支持范围是最广的。

七、用例指南——安装并引入 Vitest

添加 Vitest 到项目中

在一个基于 Vite 的 Vue 项目中，运行如下命令：

npm install -D vitest happy-dom @testing-library/vue

接着，更新你的 Vite 配置，添加上 test 选项：

```
import { defineConfig } from 'vitest/config'
import vue from '@vitejs/plugin-vue'

// https://vitejs.dev/config/
export default defineConfig({
  test: {
    // 启用类似 jest 的全局测试 API
    globals: true,
    // 使用 happy-dom 模拟 DOM
    // 这需要你安装 happy-dom 作为对等依赖（peer dependency）
    environment: 'happy-dom'
  },
  plugins: [vue()],
})
```

七、用例指南——配置 scripts 脚本： npm run test

接着在你的项目中创建名字以 *.test.js 结尾的文件。你可以把所有的测试文件放在项目根目录下的 test 目录中，或者放在源文件旁边的 test 目录中。Vitest 会使用命名规则自动搜索它们。

最后，在 package.json 之中添加测试命令，然后 npm run test 运行它：

```
"scripts": {
   "test": "vitest"
},
```

七、用例指南——测试组合函数

DEEP
THINKING
深入思考

测试组合函数

当涉及到测试组合式函数时，我们可以根据是否依赖宿主组件实例把它们分为两类。

当一个组合式函数使用以下 API 时，它依赖于一个宿主组件实例：

- 生命周期钩子
- 供给/注入

七、用例指南——只使用响应式 API 的组合式函数

如果一个组合式程序只使用响应式 API，那么它可以通过直接调用并断言其返回的状态或方法来进行测试。

```
// counter.js
import { ref } from 'vue'

export function useCounter() {
    const count = ref(0)
    const increment = () => count.value++
    return {
        count,
        increment
    }
}
```

```
// counter.test.js
import { test, expect } from 'vitest'
import { useCounter } from './counter.js'

test('useCounter', () => {
    const { count, increment } = useCounter()
    expect(count.value).toBe(0)
    increment()
    expect(count.value).toBe(1)
})
```

七、用例指南——依赖生命周期钩子或供给/注入的组合式函数

一个依赖生命周期钩子或供给/注入的组合式函数需要被包装在一个宿主组件中才可以测试。我们可以创建下面这样的帮手函数：

```
// foo.test-utils.js
import { createApp } from 'vue'

export function withSetup(composable) {
    let result
    const app = createApp({
        setup() {
            result = composable()
            // 忽略模板警告
            return () => { }
        }
    })
    app.mount(document.createElement('div'))
    // 返回结果与应用实例
    // 用来测试供给和组件卸载
    return [result, app]
}
```

七、用例指南——依赖生命周期钩子或供给/注入的组合式函数

DEEP
THINKING
深入思考

```
// foo.js
import { ref } from 'vue'

export function useFoo(val = 0) {
    const foo = ref(val)
    return {
        foo,
    }
}
```

```
// foo.test.js
import { withSetup } from './foo.test-utils'
import { useFoo } from './foo'

test('useFoo', () => {
    const [result, app] = withSetup(() => useFoo(123))
    // 为注入的测试模拟一方供给
    app.provide()
    // 执行断言
    expect(result.foo.value).toBe(1)
    // 如果需要的话可以这样触发
    app.unmount()
})
```

七、用例指南——一个组件测试 demo

对于更复杂的组合式函数，通过使用**组件测试**编写针对这个包装器组件的测试，这会容易很多。

一个组件测试的 demo

八、部分前端框架的测试框架选择

DEEP THINKING 深入思考

框架	测试框架	测试文件、用例条数、(快照)
vue3	vitest	Test Files 9 failed | 165 passed (174) Tests 12 failed | 2593 passed (2647)
react	jest	Test Suites: 2 skipped, 294 passed, 294 of 296 total Tests: 91 skipped, 7704 passed, 7795 total Snapshots: 176 passed, 176 total
nutui	jest	Test Suites: 7 failed, 77 passed, 84 total Tests: 6 failed, 442 passed, 448 total Snapshots: 6 updated, 66 passed, 72 total
element-plus	vitest	Test Files 138 passed (138) Tests 1504 passed (1504)
vant	jest	Test Suites: 169 passed, 169 total Tests: 964 passed, 964 total Snapshots: 392 passed, 392 total
ant-design	jest	Test Suites: 325 passed, 325 total Tests: 29 skipped, 4150 passed, 4179 total Snapshots: 2346 passed, 2346 total

九、初次接入测试暴露的问题

DEEP THINKING 深入思考

比如在我们的组件库为 SearchInput 编写测试用例时抛出的两个问题：

问题一：首先是 cannot find module ‘xxx’，抛开引入的问题不说，在自己的组件库再去引入其他的库的组件，这种做法本身就不推荐。

```
FAIL src/packages/searchinput/__tests__/searchinput.spec.ts
  ● Test suite failed to run

    Cannot find module '@nutui/icons-vue' from 'src/packages/searchinput/index.taro.vue'

    Require stack:
      src/packages/searchinput/index.taro.vue
      src/packages/searchinput/__tests__/searchinput.spec.ts

      15 | </template>
      16 | <script lang="ts">
    > 17 | import { IconFont } from '@nutui/icons-vue';
         | ^
      18 | import { createComponent } from '@/packages/utils/create';
      19 | const { create } = createComponent('searchinput');
      20 | export default create({

      at Resolver.resolveModule (node_modules/.pnpm/jest-resolve@26.6.2/node_modules/jest-resolve/build/index.js
:306:11)
      at Object.<anonymous> (src/packages/searchinput/index.taro.vue:17:1)
```

九、初次接入测试暴露的问题

DEEP THINKING 深入思考

问题二：注释掉上面有关 IconFont 相关的代码，再次运行测试脚本，抛出 cannot read property ‘value’ of undefined，导致测试修改 value 用例不通过。

说明 change 这块的代码逻辑可能有问题，但其实 e.target.value 和 e.detail.value 都可以正确获取到。

e.detail.value 是微信小程序的语法；e.target.value 是 h5 的语法；而 TaroEvent 整合了这两种写法。

```
TypeError: Cannot read property 'value' of undefined

  44 |
  45 |         const change = (e) => {
> 46 |             _content.emit('update:value', e.detail.value);
     |                                                    ^
  47 |         };
  48 |
  49 |         const confirmInput = () => {
```

```
● value should be update

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

  Expected: "20230308"
  Received: "20230307"

    57 |
    58 |     await wrapper.find('input').setValue('20230308')
  > 59 |     expect(wrapper.props('value')).toBe('20230308')
       |                                    ^
    60 | })
    61 |
    62 | test('value should be update', async () => {

    at Object.<anonymous> (src/packages/searchinput/__tests__/searchinput.spec.ts:59:36)
```

九、初次接入测试暴露的问题

这其实跟测试框架的运行环境有关。像 jest 通过 testEnvironment 去设置测试的环境为 node 还是 jsdom 这种类似浏览器的环境。而 Vitest 通过 test.environment 配置 'node' | 'jsdom' | 'happy-dom' | 'edge-runtime' | string。

```
// jest.config.js
module.exports = {
  testEnvironment: 'jsdom',
};
```

```
// vite.config.ts
import { defineConfig } from 'vitest/config'
import vue from '@vitejs/plugin-vue'
export default defineConfig({
  test: {
    // 启用类似 jest 的全局测试 API
    globals: true,
    // 使用 happy-dom 模拟 DOM
    // 这需要你安装 happy-dom 作为对等依赖（peer dependency）
    environment: 'happy-dom'
  },
  plugins: [vue()],
})
```

十、总结

通过上面的了解，我们会发现其实编写测试用例没有那么复杂。

如果是对纯函数编写测试用例，我们只需要学习测试框架 vitest 或 jest，在每个 test 中通过 expect 去调用我们的待测试函数，然后通过测试框架提供的 API toBe、toEqual 等去做值的比较。当然，测试框架还提供了一些模拟 API 。

如果是对组件编写测试用例，除了要学习测试框架，我们还需要学习获取组件信息/DOM 节点的手段，比如 Vue 提供的 Vue Test Utils、React 提供的以 act() 封装的测试库等。

为什么要编写测试用例？因为像组件库、公共函数库等这类的工程，一般是没有 QA 介入帮忙测试的，需要开发者自行测试来保证交付质量。

所以，我倾向将**测试用例看作是开发者的自测**。当然，也不能为了编写测试用例而强行堆砌。主要还是为自己注入一种思维，在开发时要多想想，养成一个好的习惯，避免一些浅显的错误。