



下载APP



## 21 | 单元测试：如何使用 TDD 开发一个组件？

2021-12-08 大圣

《玩转Vue 3全家桶》

课程介绍 >



讲述：大圣

时长 09:35 大小 8.78M



你好，我是大圣。

上一讲我们学习了不少组件库里的经典组件，用 TypeScript 搭建起了 TypeScript + Vite + Sass 的组件库开发基础环境，并且实现了 Container 布局组件。

今天我们来聊另外一个大幅提升组件库代码可维护性的手段：单元测试。在理解单元测试来龙去脉的基础上，我还会给你演示，如何使用测试驱动开发的方式实现一个组件，也就是社区里很火的 TDD 开发模式。



### 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，这是百度百科对单元测试的定义。而我的理解是，在我们日常代码开发中，会经常写 Console 来确认代码执行效果是否符合预期，这其实就算是测试的雏形了，我们把代码中的某个函数或者功能，传入参数后，校验输出是否符合预期。

下面的代码中我们实现了一个简单的 add 函数，并且使用打印 3 和 add(1,2) 的结果来判断函数输出。

add 函数虽然看起来很简单，但实际使用时可能会遇到很多情况。比如说 x 如果是字符串，或者对象等数据类型的时候，add 结果是否还可以符合预期？而且 add 函数还有可能被你的同事不小心加了其他逻辑，这都会干扰 add 函数的行为。

[复制代码](#)

```
1 function add(x,y){
2   return x+y
3 }
4
5 console.log(3 == add(1,2))
```

为了让 add 函数的行为符合预期，你希望能添加很多 Console 的判断逻辑，并且让这些代码自动化执行。

我们来到 src 目录下，新建一个 add.js。下面的代码中，我们定义了函数 test 执行测试函数，可以给每个测试起个名字，方便调试的时候查找，expect 可以判断传入的值和预期是否相符。

[复制代码](#)

```
1 function add(x,y){
2   return x+y
3 }
4
5 function expect(ret){
6   return {
7     toBe(arg){
8       if(ret!==arg){
9         throw Error(`预计和实际不符,预期是${arg},实际是${ret}`)
10      }
11    }
12  }
```

```
13 }
14 function test(title, fn){
15   try{
16     fn()
17     console.log(title, '测试通过')
18   }catch(e){
19     console.log(e)
20     console.error(title, '测试失败')
21   }
22 }
23 test('测试数字相加', ()=>{
24   expect(add(1,2)).toBe(3)
25 })
```

命令行执行 `node add.js` 以后，我们就可以看到下面的结果。如果每次 Git 提交代码之前，我们都能执行一遍 `add.js` 去检查 `add` 函数的逻辑，`add` 函数相当于有了个自动检查员，这样就可以很好地提高 `add` 函数的可维护性。

[复制代码](#)

```
1 → ailemente git:(main) X node add.js
2 测试数字相加 测试通过
```

下一步，我们如果想让 `add` 函数支持更多的数据类型，比如我们想支持数字字符串的相加，又要怎么处理呢？我们可以先写好测试代码，在下面的代码中，我们希望数字 1 和字符串 2 也能以数字的形式相加。

[复制代码](#)

```
1 test('测试数字和字符串数字相加', ()=>{
2   expect(add(1, '2')).toBe(3)
3 })
```

我们在命令行里执行 `node add.js` 之后，就会提示下面的报错信息，这说明现在代码还没有符合新的需求，我们需要进一步丰富 `add` 函数的逻辑。

```
→ ailemente git:(main) x node add.js
测试数字想加 测试通过
Error: 预计和实际不符,预期仕3, 实际是12
    at Object.toBe (/Users/woniuppp/vuejs/ailemente/add.js:30:15)
    at /Users/woniuppp/vuejs/ailemente/add.js:50:22
    at test (/Users/woniuppp/vuejs/ailemente/add.js:19:5)
    at Object.<anonymous> (/Users/woniuppp/vuejs/ailemente/add.js:49:1)
    at Module._compile (internal/modules/cjs/loader.js:1072:14)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1101:10)
    at Module.load (internal/modules/cjs/loader.js:937:32)
    at Function.Module._load (internal/modules/cjs/loader.js:778:12)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:76:12)
    at internal/main/run_main_module.js:17:47
测试数字和字符串数字想加 测试失败
```

我们把 add 函数改成下面的代码，再执行 add.js 后，就会提示你两个测试都通过了，这样我们就确保新增逻辑的时候，也没有影响到之前的代码逻辑。

[复制代码](#)

```
1 function add(x,y){
2   if(Number(x)==x && Number(y)==y){
3     return Number(x) + Number(y)
4   }
5   return x+y
6 }
7
```

这是一个非常简单的场景演示，但这个例子能够帮助你快速了解什么是单元测试。下一步，我们要在 Vue 中给我们的组件加上测试。

## 组件库引入 Jest

我们选择 Facebook 出品的 Jest 作为我们组件库的测试代码，Jest 是现在做测试的最佳选择了，因为它内置了断言、测试覆盖率等功能。

不过，因为我们组件库使用 TypeScript 开发，所以需要安装一些插件，通过命令行执行下面的命令，vue-jest 和 @vue/test-utils 是测试 Vue 组件必备的库，然后安装 babel 相关的库，最后安装 Jest 适配 TypeScript 的库。代码如下：

[复制代码](#)

```
1 npm install -D jest@26 vue-jest@next @vue/test-utils@next
2 npm install -D babel-jest@26 @babel/core @babel/preset-env
3 npm install -D ts-jest@26 @babel/preset-typescript @types/jest
```

安装完毕后，我们要在根目录下新建.babel.config.js。下面的配置目的是让 babel 解析到 Node 和 TypeScript 环境下。

[复制代码](#)

```
1 module.exports = {
2   presets: [
3     ['@babel/preset-env', { targets: { node: 'current' } }],
4     '@babel/preset-typescript',
5   ],
6 }
7
```

然后，我们还需要新建 jest.config.js，用来配置 jest 的测试行为。不同格式的文件需要使用不同命令来配置，对于.vue 文件我们使用 vue-jest，对于.js 或者.jsx 结果的文件，我们就要使用 babel-jest，而对于.ts 结尾的文件我们使用 ts-jest，然后匹配文件名是 xx.spec.js。这里请注意，**Jest 只会执行.spec.js 结尾的文件。**

[复制代码](#)

```
1 module.exports = {
2   transform: {
3     // .vue文件用 vue-jest 处理
4     '^.+\\.vue$': 'vue-jest',
5     // .js或者.jsx用 babel-jest处理
6     '^.+\\.jsx?$': 'babel-jest',
7     // .ts文件用ts-jest处理
8     '^.+\\.ts$': 'ts-jest'
9   },
10  testMatch: ['**/?(*.)+(spec).[jt]s?(x)']
11 }
12
```

然后配置 package.json，在 scripts 配置下面新增 test 命令，即可启动 Jest。

[复制代码](#)

```
1 "scripts": {
2   "dev": "vite",
3   "build": "vue-tsc --noEmit && vite build",
4 }
```

```
4     "serve": "vite preview",
5     "lint": "eslint --fix --ext .js,vue src/",
6     "test": "jest",
7 }
```

完成上面的操作之后，配置工作就告一段落了，可以开始输入代码做测试了。

我们可以在 src 目录下新增 test.spec.js，再输入下面代码来进行测试。在这段代码中，我们使用 expect().toBe() 来判断值是否相等，使用 toHaveBeenCalled 来判断函数是否执行。更多的断言函数你可以去 [官网](#) 查看，这些函数可以覆盖我们测试场景的方方面面。

[复制代码](#)

```
1
2
3
4
5 function sayHello(name, fn) {
6   if (name === '大圣') {
7     fn()
8   }
9 }
10 test('测试加法', () => {
11   expect(1+2).toBe(3)
12 })
13 test('测试函数', () => {
14   const fn = jest.fn()
15   sayHello('大圣', fn)
16   expect(fn).toHaveBeenCalled()
17 })
18
```

## TDD 开发组件

好，通过之前的讲解，我们已经学会如何使用 Jest 去测试函数。下一步我们来测试 Vue3 的组件，其实，Vue 的组件核心逻辑也是函数。

这里我们借助 Vue 官方推荐的 [@vue/test-utils](#) 库来测试组件的渲染，我们新建 src/components/button 文件夹，新建 Button.spec.ts。

参考 [Element3 的 button 组件](#)，el-button 组件可以通过传递 size 来配置按钮的大小。现在我们先根据需求去写测试代码，因为现在 Button.vue 还不存在，所以我们可以先根据 Button 的行为去书写测试案例。

[复制代码](#)

```
1 import Button from './Button.vue'
2 import { mount } from '@vue/test-utils'
3 describe('按钮测试', () => {
4   it('按钮能够显示文本', () => {
5     const content = '大圣小老弟'
6     const wrapper = mount(Button, {
7       slots: {
8         default: content
9       }
10    })
11    expect(wrapper.text()).toBe(content)
12  })
13   it('通过size属性控制大小', () => {
14     const size = 'small'
15     const wrapper = mount(Button, {
16       props: {
17         size
18       }
19     })
20     // size内部通过class控制
21     expect(wrapper.classes()).toContain('el-button--small')
22   })
23
24 })
```

我们首先要从 @vue/test-utils 库中导入 mount 函数，这个函数可以在命令行里模拟 Vue 的组件渲染。在 Button 的 slot 传递了文本之后，wrapper.text() 就能获取到文本内容，然后对 Button 渲染结果进行判断。之后，我们利用 size 参数，即可通过渲染不同的 class 来实现按钮的大小，这部分内容我们很熟悉了，在 [第 20 讲](#)里的 Container 组件中就已经实现过了。

然后我们在命令行执行 npm run test 来执行所有的测试代码。命令行终端上提示，我们期望 button 上含有 el-button-small class，但是实际上 button 上并没有这个 class，所以就会报错。具体报错信息你可以参考下图。



**FAIL** src/components/button/Button.spec.ts

● 按钮测试 > 通过 size 属性控制大小

expect(received).toContain(expected) // indexOf

Expected value: "el-button--small"

Received array: []

```

20 |     })
21 |     // size内部通过class控制
> 22 |     expect(wrapper.classes()).toContain(`el-button--${size}`)
    |                                ^
23 |   })
24 |   // it('全局设置按钮默认大小', () => {
25 |     //   const size = 'mini'


```

at Object.<anonymous> (src/components/button/Button.spec.ts:22:31)

之后，我们再通过实现 Button 组件的逻辑，去处理这个错误信息，这就是 TDD 测试驱动开发的方法。我们实现功能的过程就像小时候写作业，而测试代码就像批改作业的老师。

TDD 的优势就相当于有一位老师，在我们旁边不停做批改，哪怕一开始所有题都做错了，只要我们不断写代码，把所有题都回答正确，也能最后确保全部功能的正确。

我们通过接收 size 去渲染 button 的 class，我们来到 button.vue 中，通过下面的代码可以接收 size 参数，并且成功渲染出对应的 class。

 复制代码

```

1 <template>
2   <button
3     class="el-button"
4     :class="[size ? `el-button--${size}` : '']"
5   >
6     <slot />
7   </button>
8 </template>
9 <script setup lang="ts">
10
11 import {computed, withDefaults} from 'vue'
12 interface Props {
13   size?: "" | 'small' | 'medium' | 'large'
14 }
15 const props = withDefaults(defineProps<Props>(), {
16   size: ""
17 })
18 </script>
19

```



进行到这里还没有结束，class 还要通过 Sass 去修改浏览器页面内的大小。

为了让你抓住重点，这里的 Sass 代码我放几个核心逻辑，完整代码你可以在项目的 [GitHub](#) 里看到。

[复制代码](#)

```
1 @include b(button){
2   display: inline-block;
3   cursor: pointer;
4   background: $--button-default-background-color;
5   color: $--button-default-font-color;
6   @include button-size(
7     $--button-padding-vertical,
8     $--button-padding-horizontal,
9     $--button-font-size,
10    $--button-border-radius
11  );
12  @include m(small) {
13    @include button-size(
14      $--button-medium-padding-vertical,
15      $--button-medium-padding-horizontal,
16      $--button-medium-font-size,
17      $--button-medium-border-radius
18    );
19  }
20  @include m(large) {
21    @include button-size(
22      $--button-large-padding-vertical,
23      $--button-large-padding-horizontal,
24      $--button-large-font-size,
25      $--button-large-border-radius
26    );
27  }
28 }
```

前面的代码中通过 b(button) 渲染 el-button 的样式，内部使用变量都可以在 mixin 中找到。通过 b 和 button-size 的嵌套，就能实现按钮大小的控制。button 渲染的结果，你可以参考下方的截图。

```

<button class="el-button el-button--large">按钮</button> == $0
<button class="el-button el-button--small">...</button>
<hr>
<section class="el-container is-vertical">...</section> flex
<hr>
<section class="el-container is-vertical">...</section> flex
<hr>
<section class="el-container">...</section> flex
</div>
<script type="module" src="/src/main.ts"></script>
/body>
html>

```

```

.el-button--large {
  padding: 15px 25px;
  font-size: 12px;
  border-radius: 3px;
}

.el-button {
  display: inline-block;
  line-height: 1;
  white-space: nowrap;
  cursor: pointer;
  background: > #ffffff;
  color: #606266;
  -webkit-appearance: none;
  text-align: center;
  border: 1px solid #dcdf66;
  border-color: > #dcdf66;
  box-sizing: border-box;
  outline: none;
  margin: 0;
}

```

然后我们接着往下进行，想要设置按钮的大小，除了通过 props 传递，还可以通过全局配置的方式设置默认大小。我们进入到代码文件 src/main.ts 中，设置全局变量 \$AILEMENTE 中的 size 为 large，并且还可以通过 type="primary" 或者 type="success" 的方式，设置按钮的主体颜色，代码如下：

[复制代码](#)

```

1  const app = createApp(App)
2  app.config.globalProperties.$AILEMENTE = {
3    size: 'large'
4  }
5  app.use(ElContainer)
6    .use(ElButton)
7    .mount('#app')
8
9
10

```

首先我们要支持全局的 size 配置，在 src 目录下新建 util.ts，写入下面的代码。我们通过 vue 提供的 getCurrentInstance 获取当前的实例，然后返回全局配置的 \$AILEMENTE。这里请注意，由于很多组件都需要读取全局配置，所以我们封装了 useGlobalConfig 函数。

[复制代码](#)


```

1  import { getCurrentInstance, ComponentInternalInstance } from 'vue'
2
3  export function useGlobalConfig(){
4    const instance: ComponentInternalInstance | null = getCurrentInstance()
5    if(!instance){
6      console.log('useGlobalConfig 必须得在setup里面整')
7      return
8    }
9    return instance.appContext.config.globalProperties.$AILEMENTE || {}
10

```

```
11 }
```

这时我们再回到 Button.vue 中，通过 computed 返回计算后的按钮的 size。如果 props.size 没传值，就使用全局的 globalConfig.size；如果全局设置中也没有 size 配置，按钮就使用 Sass 中的默认大小。

 复制代码

```
1 <template>
2   <button
3     class="el-button"
4     :class="[
5       buttonSize ? `el-button--${buttonSize}` : '',
6       type ? `el-button--${type}` : ''
7     ]"
8   >
9     <slot />
10  </button>
11 </template>
12
13 <script lang="ts">
14   export default{
15     name: 'ElButton'
16   }
17 </script>
18
19 <script setup lang="ts">
20
21   import {computed, withDefaults} from 'vue'
22   import { useGlobalConfig } from '../util';
23
24   interface Props {
25     size?: "" | 'small' | 'medium' | 'large',
26     type?: "" | 'primary' | 'success' | 'danger'
27   }
28   const props = withDefaults(defineProps<Props>(), {
29     size: "",
30     type: ""
31   })
32   const globalConfig = useGlobalConfig()
33   const buttonSize = computed(() => {
34     return props.size || globalConfig.size
35   })
36 </script>
37
```

我们来到 `src/App.vue` 中，就可以直接使用 `el-button` 来显示不同样式的按钮了。

[复制代码](#)

```
1 <el-button type="primary">
2   按钮
3 </el-button>
4 <el-button type="success">
5   按钮
6 </el-button>
7 <el-button>按钮</el-button>
8 <el-button size="small">
9   按钮
10 </el-button>
11
```

不同按钮的显示效果如下所示：

[极客时间](#)

然后我们进入 `jest.config.js` 中，新增下面的配置，`collectCoverage` 标记的意思是我们需要收集代码测试覆盖率。

[复制代码](#)

```
1 module.exports = {
2   transform: {
3     // 用 `vue-jest` 处理 `*.vue` 文件
4     '^.+\\.vue$': 'vue-jest', // vuejest 处理.vue
5     '^.+\\.jsx?$': 'babel-jest', // babel jest处理js or jsx
6     '^.+\\.tsx?$': 'ts-jest', // ts-jest 处理.ts .tsx
7   },
8   testMatch: ['**/?(*.)+(spec).[jt]s?(x)'],
9   collectCoverage: true,
10  coverageReporters: ['json', 'html'],
11 }
12
```

然后在执行 `npm run test` 后，项目的根目录下就会出现一个 `coverage` 目录。

我们打开下面的 `index.html` 后，就可以在浏览器中看到测试覆盖率的报告。对照下图我们可以看到，`button` 组件的测试覆盖率 100%，`util` 下面有两行代码飘红，也就是没有测试的逻辑。

在一定程度上，测试覆盖率也能够体现出代码的可维护性，希望你可以用好这个指标。

File		Statements	Branches	Functions	Lines				
src	<div><div></div></div>	71.42%	5/7	66.66%	2/3	100%	1/1	71.42%	5/7
src/components/button	<div><div></div></div>	100%	8/8	100%	2/2	100%	1/1	100%	7/7

### All files / src util.ts

71.42% Statements 5/7 66.66% Branches 2/3 100% Functions 1/1 71.42% Lines 5/7

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1 1x import { getCurrentInstance } from 'vue'
2
3
4 1x export function useGlobalConfig(){
5 4x   const instance =getCurrentInstance()
6 4x   if(!instance){
7     console.log('useGlobalConfig 必须得在setup里面整')
8     return
9   }
10 4x   return instance.appContext.config.globalProperties.$AILEMENTE || {}
11 }
```

### All files src/components/button

100% Statements 8/8 100% Branches 2/2 100% Functions 1/1 100% Lines 7/7

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements		Branches		Functions		Lines	
Button.vue	<div><div></div></div>	100%	8/8	100%	2/2	100%	1/1	100%	7/7

最后，我们进入 `.husky/pre-commit` 文件，新增 `npm run test` 命令，这么做的目的是，确保测试通过的代码才能进入 `git` 管理代码，这会进一步提高代码的规范和可维护性。

```
1 #!/bin/sh
2
3 . "$(dirname "$0")/_/husky.sh"
4
5 npm run lint
6 npm run test
```

## 总结

今天的内容就到此结束了，我们来回顾一下今天学到的东西吧。

首先，我们学习了什么是自动化测试，我们实现了 test 和 expect 函数，通过它们来测试 add 函数。

然后，我们通过 jest 框架配置了 Vue 的自动化测试环境。通过安装 babel、@vue/test-utils、babel-vue、ts-babel 等插件，我们配置了 TypeScript 环境下的 Jest+Vue 3 的单元测试环境，并且匹配项目中.spect 结束的 js 和 vue 文件执行测试。

在 Jest 中，我们通过 describe 函数给测试分组，通过 it 执行测试，再利用 expect 语法去执行断言。我们还发现，借助 @vue/test-utils 库可以很方便地对 Vue 组件进行测试。

最后，我们一起体验了 TDD 测试驱动开发的开发模式。我们先根据功能需求，去写出测试案例，这个时候测试窗口就会报错，然后我们才开始实现功能，最终让测试代码全部通过，用这样的方式来检验开发的结果。**TDD 的优势就在于可以随时检验代码的逻辑，能极大提高代码的可维护性。**

现在我们有了 TypeScript，有了 Jest，下一讲我们将实现一个比较复杂的表单组件，它会包含组件的通信、方法传递等难点，敬请期待。

## 思考题

最后留个思考题，我们的 Button 组件怎么通过传递 circle 属性来显示圆角按钮呢？

欢迎你在评论区留下你的答案，也欢迎你把这一讲分享给你的同事和朋友们，我们下一讲再见！

 生成海报并分享 赞 6 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 20 | 组件库：如何设计你自己的通用组件库？

下一篇 22 | 表单：如何设计一个表单组件？

## 更多课程推荐

# 跟月影学可视化

系统掌握图形学与可视化核心原理

月影

奇虎 360 奇舞团团长

可视化 UI 框架 SpriteJS 核心开发者



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

## 精选留言 (5)

 写留言

小海

2021-12-08

赞, 发现两个小瑕疵

1. 在 Button.spec.ts 文件中引入 button.vue 组件时, 会提示找不到该模块, 后来看了 github 链接的源码才发现是需要 src 目录下增加 env.d.ts 文件, 才能使 TS 文件顺利引入 vue 文件的组件,



2. babel.config.js 在课程资料里是创建 .babel.config.js文件 但是源码里并没有"." 不晓得...

展开 ∨

共 1 条评论 >

👍 1



**海阔天空**

2021-12-08

感觉单元测试这块用得比较少，还是用console检查用得比较多，这可能和项目的迭代周期有关。单元测试确实比较更全面。

展开 ∨



👍 1



**小胖**

2021-12-09

接上一篇提问：上篇文章的几个布局组件，定义Props类型的时候。老师有时是使用type、有时用interface，有什么说法么？

展开 ∨

共 1 条评论 >

👍



**Geek\_a964f4**

2021-12-09

直到现在还没用过自动化测试呢~~

展开 ∨



👍



**南山**

2021-12-08

传入的circle属性，生成.btn--circle的classname，实现圆角样式



👍