* [首页](https://km.woa.com/?kmref=km_header)
* [发现](https://km.woa.com/discovery?kmref=km_header)
* [悦读](https://km.woa.com/read?kmref=km_header)
* [乐问](https://km.woa.com/q?kmref=km_header)
* [直播](https://tencent.lexiangla.com/lives?company_from=tencent&kmref=km_header)
* [应用](javascript:void(0))
* [我的K吧](javascript:void(0))
* [[](https://km.woa.com/user/rickyrqzhao)](https://km.woa.com/user/rickyrqzhao)

窗体顶端



窗体底端

[](https://km.woa.com/group/wfe)[腾讯视频VIP前端团队](https://km.woa.com/group/29573)

* [首页](https://km.woa.com/group/29573)
* [团队微博](https://km.woa.com/group/29573/twitters)
* [团队讨论](https://km.woa.com/group/29573/topics)
* [团队文章](https://km.woa.com/group/29573/articles)
* [团队文档](https://km.woa.com/group/29573/docs)
* [团队活动](https://km.woa.com/group/29573/events)
* [团队投票](https://km.woa.com/group/29573/surveys)
* [团队日历](https://km.woa.com/group/29573/calendars)
* [团队会议](https://km.woa.com/group/29573/meetings)

目录

基本概念

为什么要进行代码测试？

测试类型分类

单元测试 (Unit Test)

集成测试 (Integration Test)

端对端测试 (End to End Test)

单元测试特点

单元测试实战

技术选型

为什么选择 Jest ？

测试环境搭建

1. 安装依赖

2. 测试环境配置

3. 执行测试

常用断言

单元测试常用技巧

1. 使用 beforeEach 或 afterEach 处理多次测试重复的依赖环境初始化工作或设置全局变量

2. 使用 beforeAll 或 afterAll 处理测试一次性的环境初始化工作

3. 使用 describe.each 或 test.each 进行批量测试

4. 使用 [describe|test].only 或 [describe|test].skip 调试测试用例或测试套件

5. 使用 Object.defineProperty 模拟全局对象或方法

6. 模拟 iframe 场景

7. 使用 jest.fn mock 函数

8. 使用 jest.spyOn mock 对象的方法

9. 使用 jest.requireActual mock 模块的部分函数

10. 测试异步函数

11. 使用模拟计时器

12. 使用 snapshot 快照测试UI组件

持续集成

踩坑记录

附录

单元测试开发模式

Jest 常用配置项说明

覆盖率指标说明

覆盖率信息文件报告说明

前端单元测试总结

* [elvenzhang](https://km.woa.com/user/elvenzhang)

* 2021年09月02日 17:02

* 浏览(435)

* [收藏(40)](javascript:void(0);)

* [评论(3)](javascript:void(0);)

* [分享](javascript:void(0);)

持续更新 单测技巧 和 踩坑记录 ~

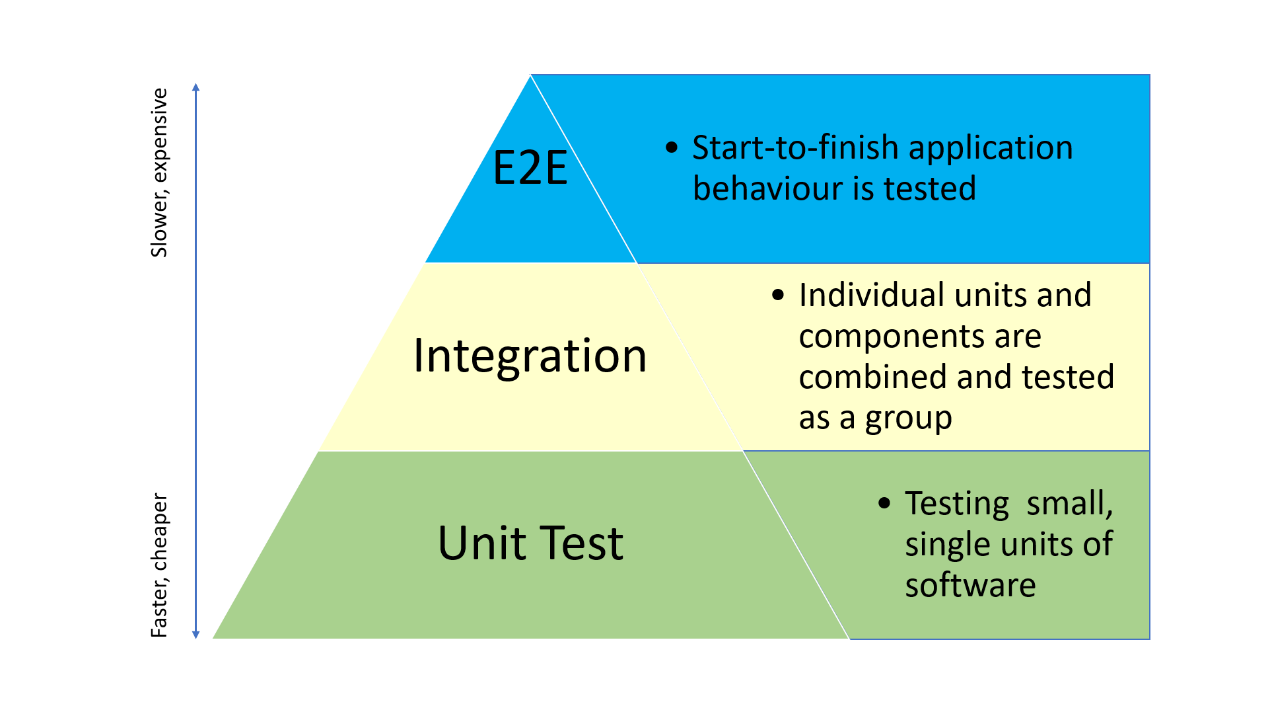
**基本概念**

**为什么要进行代码测试？**

1. 测试可以确保得到预期的结果，验证功能完整性
2. 测试作为现有代码行为的描述
3. 促使开发者写可测试的代码，一般可测试的代码可读性也会高一点，保证代码的质量
4. 大规模代码重构时，如果依赖的组件有修改，受影响的组件能在测试中发现错误，能保证重构的正确性

**测试类型分类**

对于前端接入自动化测试来说，最先接入单元测试更可靠。  
因为单元测试是持续集成的重要的一环，是整个测试组合的基石，且单元测试成本更低、效率更高



（从下往上，集成度更高，成本更高，粒度更粗，效率更低）

**单元测试 (Unit Test)**

* 单元测试就是对程序模块（软件设计的最小单元）来进行正确性检验的测试工作  
    
  在过程化编程中，最小单元就是单个程序、函数、过程等  
    
  对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法

**集成测试 (Integration Test)**

* 集成测试在单元测试的基础上，对经过单元测试后的各个模块组合在一起进行测试，查看组合后的代码工作是否符合预期

**端对端测试 (End to End Test)**

* 端到端的测试，就是以用户的角度、模拟用户对产品进行操作，并判断每次操作的结果是否符合预期。对于 web 前端来说，就是聚焦于用户与界面之间的交互

**单元测试特点**

* 单元测试是相互独立的
  + 任何给定的行为都应该在一个且只有一个测试中指定
  + 一个测试的执行/执行顺序不会影响其他
* 单元测试是轻量级测试
  + 可重复的
  + 快速的
  + 一致的
  + 容易读写的
* 单元测试也是代码
  + 它们应该达到与正在测试的代码相同的质量级别。还可以对它们进行重构，使它们更易于维护和可读
* 单元测试方向
  + UI 性单元测试：对 UI 节点和交互验证
  + 功能性单元测试：对函数单元、接口单元、整合性单元测试

**单元测试实战**

**技术选型**

* [Jest](https://jestjs.io/) 是一款开箱即用的测试框架，其中包含了 [Expect](https://jestjs.io/docs/en/expect) 断言接口、[Mock](https://jestjs.io/docs/en/mock-function-api) 接口、Snapshot 快照、测试覆盖率统计等等全套测试功能
* [Vue Test Utils](https://vue-test-utils.vuejs.org/) 是 Vue.js 官方的单元测试实用工具库

**为什么选择 Jest ？**

* **易用性**：基于 Jasmine，提供断言库，支持多种测试风格
* **适应性**：Jest 是模块化、可扩展和可配置的
* **沙箱和快照**：Jest 内置了 JSDOM，能够模拟浏览器环境，并且并行执行
* **快照测试**：Jest 能够对 React 组件树进行序列化，生成对应的字符串快照，通过比较字符串提供高性能的 UI 检测 (Vue 可通过 jest-serializer-vue 序列化快照)
* **Mock 系统**：Jest 实现了一个强大的 Mock 系统，支持自动和手动 mock
* **支持异步代码测试**：支持 Promise 和 async/await
* **自动生成静态分析结果**：内置 Istanbul，测试代码覆盖率，并生成对应的报告

**测试环境搭建**

**1. 安装依赖**

* Jest + Typescript
* # 使用 Typescript 编写测试代码
* npm i -D jest ts-jest typescript @types/jest
* Vue2.x
* # 详细文档：https://vue-test-utils.vuejs.org/
* npm i -D @vue/test-utils @vue/cli-plugin-unit-jest
* Vue3.x
* # 详细文档：https://next.vue-test-utils.vuejs.org/
* npm i -D @vue/test-utils@next vue-jest@next

**2. 测试环境配置**

参考项目实例 → <https://git.woa.com/vft/micro-channel>

* tsconfig.json
* {
* "compilerOptions": {
* "rootDir": ".",
* "baseUrl": ".",
* "target": "ES5",
* "module": "ESNext",
* "strict": true,
* "noEmit": false,
* "moduleResolution": "Node",
* "esModuleInterop": true,
* "experimentalDecorators": true,
* "noUnusedLocals": true,
* "emitDecoratorMetadata": true,
* "noImplicitThis": false,
* "allowSyntheticDefaultImports": true,
* "allowJs": true,
* "types": ["jest", "node"],
* "lib": ["ESNext", "DOM", "DOM.Iterable", "ScriptHost"],
* "paths": {
* "@/\*": ["src/\*"]
* }
* },
* "include": ["src/\*\*/\*.ts", "types/\*\*/\*.d.ts", "\_\_tests\_\_/\*\*/\*.ts"],
* "exclude": ["node\_modules", "dist"]
* }
* jest.config.js
* # 自动生成 jest.config.js
* npx jest --init
* # Jest + Typescript
* npx ts-jest config:init

/\*\*

\* @type {import('ts-jest/dist/types').InitialOptionsTsJest}

\*/

module.exports = {

displayName: 'ChannelService',

rootDir: \_\_dirname,

verbose: true,

preset: 'ts-jest',

testEnvironment: 'jsdom',

collectCoverage: true,

coverageDirectory: '<rootDir>/coverage',

collectCoverageFrom: ['src/\*\*/\*.ts'],

testMatch: ['\*\*/\_\_tests\_\_/\*.spec.[jt]s?(x)'],

moduleNameMapper: {

'^@/(.\*)$': '<rootDir>/src/$1',

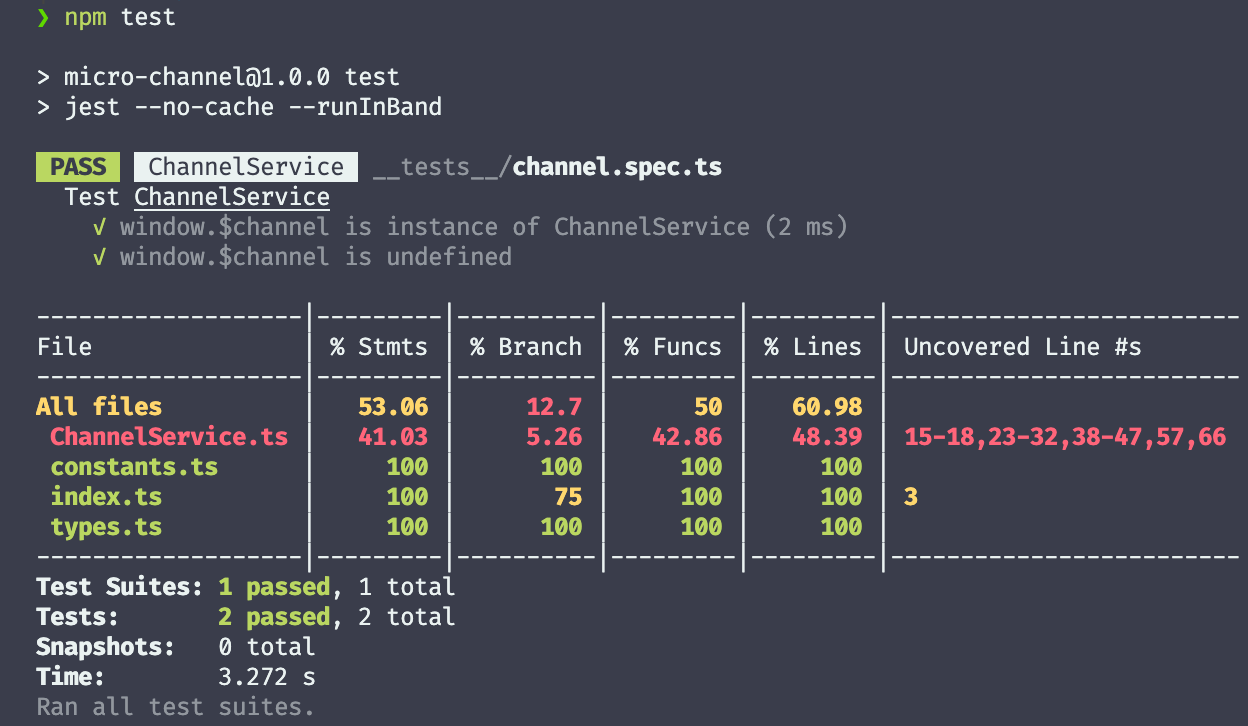
},

};

* package.json
* {
* ...
* "scripts": {
* ...
* "test": "jest --no-cache --runInBand"
* },
* ...
* }

**3. 执行测试**

npm test



**常用断言**

文档：<https://jestjs.io/docs/expect#reference>

1. expect(value)：要测试一个值进行断言的时候，要使用 expect 对值进行包裹
2. test('the number is >= 90', () => {
3. const number = 100;
4. expect(number).toBeGreaterThanOrEqual(90);
5. });
6. toBe(value)：使用 Object.is 来进行比较，如果进行浮点数的比较，要使用toBeCloseTo
7. const can = {
8. name: 'pamplemousse',
9. ounces: 12,
10. };
11. describe('the can', () => {
12. test('has 12 ounces', () => {
13. expect(can.ounces).toBe(12);
14. });
15. test('has a sophisticated name', () => {
16. expect(can.name).toBe('pamplemousse');
17. });
18. });
19. toBeCloseTo(number, numDigits): 比较浮点数
20. expect(0.2 + 0.1).toBeCloseTo(0.3, 5);
21. not：用来取反
22. test('the name is not elven', () => {
23. const name = 'eleven';
24. expect(name).not.toBe('elven');
25. });
26. toEqual(value)：用于对象的深比较
27. const can1 = {
28. flavor: 'grapefruit',
29. ounces: 12,
30. };
31. const can2 = {
32. flavor: 'grapefruit',
33. ounces: 12,
34. };
35. describe('the La Croix cans on my desk', () => {
36. test('have all the same properties', () => {
37. expect(can1).toEqual(can2);
38. });
39. test('are not the exact same can', () => {
40. expect(can1).not.toBe(can2);
41. });
42. });
43. toMatch(regexpOrString)：用来检查字符串是否匹配，可以是正则表达式或者字符串
44. test('the name is eleven', () => {
45. const name = 'eleven';
46. expect(name).toMatch(/eleven/);
47. expect(name).toMatch('eleven');
48. });
49. toContain(item)：用来判断 item 是否在一个数组中，也可以用于字符串的判断
50. test('fruits contain apple', () => {
51. const fruits = ['apple', 'banana', 'orange'];
52. expect(fruits).toContain('apple');
53. });
54. toBeNull()：只匹配 null
55. test('the value is null', () => {
56. const value = null;
57. expect(name).toBeNull();
58. });
59. toBeUndefined()：只匹配 undefined
60. test('window.$channel is undefined', () => {
61. window.$channel = undefined;
62. expect(window.$channel).toBeUndefined();
63. });
64. toBeDefined()：与 toBeUndefined 相反
65. test('window.$channel is undefined', () => {
66. window.$channel = {};
67. expect(window.$channel).toBeDefined();
68. });
69. toBeTruthy()：匹配任何使 if 语句为真的值
70. // demo
71. let value = false;
72. const getValue = () => (value = true);
73. const isTrue = () => value === true;
74. if (isTrue()) {
75. console.log(value);
76. }
77. // test case
78. test('isTrue return true', () => {
79. getValue();
80. expect(isTrue()).toBeTruthy();
81. });
82. toBeFalsy()：匹配任何使 if 语句为假的值
83. // test case
84. test('isTrue return false', () => {
85. expect(isTrue()).toBeFalsy();
86. });
87. toBeGreaterThan(number)： 大于
88. test('the number is > 90', () => {
89. const number = 100;
90. expect(number).toBeGreaterThan(90);
91. });
92. toBeGreaterThanOrEqual(number)：大于等于
93. test('the number is >= 90', () => {
94. const number = 100;
95. expect(number).toBeGreaterThanOrEqual(90);
96. });
97. toBeLessThan(number)：小于
98. test('the number is < 90', () => {
99. const number = 80;
100. expect(number).toBeLessThan(90);
101. });
102. toBeLessThanOrEqual(number)：小于等于
103. test('the number is <= 90', () => {
104. const number = 90;
105. expect(number).toBeLessThanOrEqual(90);
106. });
107. toBeInstanceOf(class)：判断是不是 class 的实例
108. test('window.$channel is instance of ChannelService', () => {
109. expect(window.$channel).toBeInstanceOf(ChannelService);
110. });
111. anything()：匹配除了 null 和 undefined 以外的所有值
112. test('map calls its argument with a non-null argument', () => {
113. const mock = jest.fn();
114. [1].map((x) => mock(x));
115. expect(mock).toBeCalledWith(expect.anything());
116. });
117. resolves：用来取出 promise 为 fulfilled 时包裹的值，支持链式调用
118. test('resolves to orange', () => {
119. return expect(Promise.resolve('orange')).resolves.toBe('orange');
120. });
121. test('resolves to orange', async () => {
122. await expect(Promise.resolve('orange')).resolves.toBe('orange');
123. await expect(Promise.resolve('apple')).resolves.not.toBe('orange');
124. });
125. rejects：用来取出 promise 为 rejected 时包裹的值，支持链式调用
126. test('rejects to network error', () => {
127. return expect(Promise.reject(new Error('network error'))).rejects.toThrow('network error');
128. });
129. test('rejects to network error', async () => {
130. await expect(Promise.reject(new Error('network error'))).rejects.toThrow('network error');
131. });
132. toHaveBeenCalled()：用来判断 mock function 是否被调用过
133. test('mock function have been called', () => {
134. const mockFn = jest.fn();
135. mockFn();
136. expect(mockFn).toHaveBeenCalled();
137. });
138. toHaveBeenCalledTimes(number)：用来判断 mock function 被调用的次数
139. test('mock function have been called', () => {
140. const mockFn = jest.fn();
141. [1, 2].map((num) => mockFn(num));
142. expect(mockFn).toBeCalledWith(expect.anything());
143. expect(mockFn).toHaveBeenCalledTimes(2);
144. });
145. assertions(number)：验证在一个测试用例中有 number 个断言被调用
146. test('doAsync calls both callbacks', async () => {
147. expect.assertions(2);
148. function callback1(data) {
149. expect(data).toBeTruthy();
150. }
151. function callback2(data) {
152. expect(data).toBeTruthy();
153. }
154. await doAsync(callback1, callback2);
155. });
156. extend(matchers)：自定义断言
157. expect.extend({
158. toBeWithinRange(received, floor, ceiling) {
159. const pass = received >= floor &amp;amp;&amp;amp; received <= ceiling;
160. if (pass) {
161. return {
162. message: () =>
163. `expected ${received} not to be within range ${floor} - ${ceiling}`,
164. pass: true,
165. };
166. }
167. return {
168. message: () =>
169. `expected ${received} to be within range ${floor} - ${ceiling}`,
170. pass: false,
171. };
172. },
173. });
174. test('numeric ranges', () => {
175. expect(100).toBeWithinRange(90, 110);
176. expect(101).not.toBeWithinRange(0, 100);
177. expect({apples: 6, bananas: 3}).toEqual({
178. apples: expect.toBeWithinRange(1, 10),
179. bananas: expect.not.toBeWithinRange(11, 20),
180. });
181. });

自定义断言的类型定义

declare global {

namespace jest {

interface Matchers<R> {

toBeWithinRange(a: number, b: number): R;

}

}

}

**单元测试常用技巧**

**1. 使用 beforeEach 或 afterEach 处理多次测试重复的依赖环境初始化工作或设置全局变量**

* [beforeEach(fn, timeout)](https://jestjs.io/docs/api#beforeeachfn-timeout) 在每个测试用例执行前运行
* [afterEach(fn, timeout)](https://jestjs.io/docs/api#aftereachfn-timeout) 在每个测试用例执行结束后运行
* beforeEach(() => {
* Object.defineProperty(window, 'TenvideoJSBridge', {
* writable: true,
* value: undefined,
* });
* });
* afterEach(() => {
* // 还原所有被 mock 的 timer
* jest.useRealTimers();
* });
* test('window.TenvideoJSBridge is undefined', () => {
* expect(window.TenvideoJSBridge).toBeUndefined();
* });
* test('window.TenvideoJSBridge is defined', () => {
* // 使用 Jest mock 的 timer
* jest.useFakeTimers();
* expect(window.TenvideoJSBridge).toBeUndefined();
* // 模拟 App 1000ms 后注入 TenvideoJSBridge 对象
* const inject = jest.fn(() => {
* window.TenvideoJSBrdige = {};
* });
* setTimeout(inject, 1000);
* expect(inject).not.toBeCalled();
* // 快速执行当前 pending timer，跳过等待时间
* jest.runOnlyPendingTimers();
* expect(inject).toBeCalled();
* expect(window.TenvideoJSBridge).toBeDefined();
* });

**2. 使用 beforeAll 或 afterAll 处理测试一次性的环境初始化工作**

* [beforeAll(fn, timeout)](https://jestjs.io/docs/api#beforeallfn-timeout) 在所有测试用例执行前运行一次
* [afterAll(fn, timeout)](https://jestjs.io/docs/api#afterallfn-timeout) 在所有测试用例执行后运行一次
* const setEnvPromise = async () => { ... };
* const clearEnvPromise = async () => { ... };
* describe('Test AppJSBridge in App environment', () => {
* beforeAll(() => {
* // 如果初始化工作是异步，可直接返回 Promise，测试用例会在初始化完成后执行
* return setEnvPromise();
* });
* afterAll(() => {
* return clearEnvPromise();
* });
* test('Env.isApp should be true', () => {
* expect(Env.isApp).toBeTruthy();
* });
* });
* [作用域(scoping)](https://jestjs.io/docs/setup-teardown#scoping)

默认情况下，before 和 after 的块可以应用到文件中的每一个测试用例。可以通过 describe 来将测试用例分组，当 before 和 after 的块放在 describe 内时，则只适用于 describe 块内的测试用例

beforeAll(() => console.log('global: beforeAll'));

afterAll(() => console.log('global: afterAll'));

beforeEach(() => console.log('global: beforeEach'));

afterEach(() => console.log('global: afterEach'));

test('global test', () => console.log('global: test'));

describe('Scoped / Nested block', () => {

beforeAll(() => console.log('scope: beforeAll'));

afterAll(() => console.log('scope: afterAll'));

beforeEach(() => console.log('scope: beforeEach'));

afterEach(() => console.log('scope: afterEach'));

test('', () => console.log('scope: test'));

});

// global: beforeAll

// global: beforeEach

// global: test

// global: afterEach

// scope: beforeAll

// global: beforeEach

// scope: beforeEach

// scope: test

// scope: afterEach

// global: afterEach

// scope: afterAll

// global: afterAll

**3. 使用 describe.each 或 test.each 进行批量测试**

如果使用不同的数据重复相同的测试套件或测试用例，可使用 describe.each 或 test.each 进行批量测试

* [describe.each(table)(name, fn, timeout)](https://jestjs.io/docs/api#describeeachtablename-fn-timeout)
* const add = (a: number, b: number) => a + b;
* describe.each([
* [1, 1, 2],
* [1, 2, 3],
* [2, 1, 3],
* ])('.add(%i, %i)', (a, b, expected) => {
* test(`returns ${expected}`, () => {
* expect(add(a, b)).toBe(expected);
* });
* test(`returned value not be greater than ${expected}`, () => {
* expect(add(a, b)).not.toBeGreaterThan(expected);
* });
* test(`returned value not be less than ${expected}`, () => {
* expect(add(a, b)).not.toBeLessThan(expected);
* });
* });
* [test.each(table)(name, fn, timeout)](https://jestjs.io/docs/api#testeachtablename-fn-timeout)
* test.each([
* [1, 1, 2],
* [1, 2, 3],
* [2, 1, 3],
* ])('.add(%i, %i)', (a, b, expected) => {
* expect(a + b).toBe(expected);
* });

**4. 使用 [describe|test].only 或 [describe|test].skip 调试测试用例或测试套件**

在单个测试文档如果含有非常多的测试用例或套件，测试用例可能会出现全局属性相互影响的问题，可通过以下方法调试单个测试套件或测试用例

// 跳过单个测试套件

describe.skip('', () => {});

// 只运行单个测试套件

describe.only('', () => {});

// 跳过单个测试用例

test.skip('', () => {});

// 只运行单个测试用例

test.only('', () => {});

* 建议

如果有一个测试用例(套件)在大型的测试案例下执行失败，但单独执行不會失败，那可能是其他测试用例干扰了这个测试用例。通常可用 beforeEach 來清除一些共享状态来解决问题。若不确定是否正在修改哪些共享状态，也可尝试用 beforeEach 來输出调试日志

**5. 使用 Object.defineProperty 模拟全局对象或方法**

通过 Object.defineProperty 修改的对象无法还原，需要自己保存原始对象，在测试执行后还原

注意 mock 的对象或方法是否会影响到其他测试用例，需要在测试用例执行后还原初始状态

如果测试用例使用了 JSDOM 下未定义的方法或对象，执行会报错，可使用 Object.defineProperty 解决问题

test('TenvideoJSBridge.invoke throw error', () => {

Object.defineProperty(window, 'TenvideoJSBridge', {

writable: true,

value: {

invoke: jest.fn().mockImplementation(() => {

throw new Error('inovke error');

}),

},

});

const bridge = new AppJSBridge();

const apiName = 'reloadWKCookie';

return expect(bridge.invoke('reloadWKCookie', null, { timeout: 200 })).rejects.toEqual(

createInvokeTimeoutError(apiName)

);

});

**6. 模拟 iframe 场景**

/\*\*

\* 模拟iframe下的对象

\* @param object

\* @returns

\*/

function mockObjectInIframe(value: AnyObject) {

const originalSelf = { ...window.self };

const originalTop = { ...window.top };

Object.defineProperty(window, 'self', {

writable: true,

value: {},

});

Object.defineProperty(window, 'top', {

writable: true,

value,

});

return function resetMocks() {

Object.defineProperty(window, 'self', {

writable: true,

value: originalSelf,

});

Object.defineProperty(window, 'top', {

writable: true,

value: originalTop,

});

};

}

test('window.$channel.context is window.parent.$channel.context', () => {

const resetMocks = mockObjectInIframe({ $channel: new ChannelService() });

const isEmbedded = window.self !== window.top;

window.$channel = new ChannelService();

expect(isEmbedded).toBeTruthy();

expect(window.parent?.$channel).toBeInstanceOf(ChannelService);

expect(window.$channel?.context).toEqual(window.parent.$channel?.context);

// 需要重置mock，避免影响其他测试用例

resetMocks();

});

**7. 使用 jest.fn mock 函数**

* [jest.fn(implementation)](https://jestjs.io/docs/jest-object#jestfnimplementation)

使用 jest.fn 对函数 mock 之后，mockFn 会有一个 mock 对象属性，可以获取 mock 函数的调用次数，调用参数，返回值

* mockFn.mock.calls 返回一个二维数组
  + mock.calls[0] 表示第一次调用，返回是一个 args 数组。
  + mock.calls[1][0] 表示第二次调用的第一个参数值
* mockFn.mock.results 返回一个一维数组
  + 每个数组项返回一个包含 type 和 value 字段的对象
  + {
  + /\*\* return: 正常返回, throw: 抛出异常 \*/
  + type: 'return' | 'throw',
  + /\*\* 对应的正常返回的值或异常信息 \*/
  + value: unknown
  + }
  + 数组索引表示 mock 函数的调用次数
* 示例
* function forEach(items: number[], callback: (x: number) => number) {
* for (const item of items) {
* callback(item);
* }
* }
* test('forEach', () => {
* const mockCallback = jest.fn((x: number) => 42 + x);
* forEach([1, 2], mockCallback);
* console.log(mockCallback.mock.calls);
* // [
* // [1],
* // [2]
* // ]
* // 调用次数
* expect(mockCallback.mock.calls.length).toBe(2);
* // 第一次调用的参数
* expect(mockCallback.mock.calls[0][0]).toBe(1);
* // 第二次调用的参数
* expect(mockCallback.mock.calls[1][0]).toBe(2);
* console.log(mockCallback.mock.results);
* // [
* // {
* // type: 'return',
* // value: 43,
* // },
* // {
* // type: 'return',
* // value: 44
* // }
* // ]
* // 第一次调用返回的结果
* expect(mockCallback.mock.results[0].value).toBe(43);
* // 第二次调用返回的结果
* expect(mockCallback.mock.results[1].value).toBe(44);
* });
* mock 函数返回值
* test('jest.fn() 返回值', () => {
* const mockFn = jest.fn();
* console.log(mockFn()); // undefined
* // 仅会返回一次
* mockFn.mockReturnValueOnce(10);
* expect(mockFn()).toBe(10);
* console.log(mockFn()); // undefined;
* mockFn.mockReturnValueOnce('x').mockReturnValue(true);
* expect(mockFn()).toBe('x');
* expect(mockFn()).toBe(true);
* expect(mockFn()).toBe(true);
* });
* test('jest.fn() 返回 promsie', async () => {
* const mockFn = jest.fn().mockResolvedValue('name');
* const result = await mockFn();
* // 断言 mockFn 通过 await 关键字执行后返回值为 name
* expect(result).toBe('name');
* });

**8. 使用 jest.spyOn mock 对象的方法**

* [jest.spyOn(object, methodName, accessType?)](https://jestjs.io/docs/jest-object#jestspyonobject-methodname)

1. mock 模块函数
2. import utils from '@/utils';
3. test('returns winner', () => {
4. // utils.getWinner 会被取代成一个空的 mock function
5. const spy = jest.spyOn(utils, 'getWinner');
6. spy.mockImplementation((p1, p2) => p2);
7. ...
8. spy.mockRestore() // 还原初始 function
9. });
10. mock window.localStorage.setItem
11. setItemFn = jest.spyOn(window.localStorage.\_\_proto\_\_, 'setItem');
12. 模拟 iframe 跨域报错的场景
13. test('getJSBridge throw error', async () => {
14. const spy = jest.spyOn(window, 'top', 'get');
15. spy.mockImplementation(() => {
16. throw new Error('inaccessible');
17. });
18. const { default: bridge } = await import('../../../libs/jsbridge/JSBridge');
19. expect(bridge.constructor.name).toBe('AppJSBridge');
20. expect(window.JSBridge).toEqual(bridge);
21. spy.mockRestore();
22. });

**9. 使用 jest.requireActual mock 模块的部分函数**

直接使用 jest.mock(@module/api) 会 mock 整个模块

* [jest.mock(moduleName, factory, options)](https://jestjs.io/docs/jest-object#jestmockmodulename-factory-options)
* [jest.requireActual(moduleName)](https://jestjs.io/docs/jest-object#jestrequireactualmodulename)
* import { functionToMock } from '@module/api';
* jest.mock('@module/api', () => {
* const original = jest.requireActual('@module/api');
* return {
* ...original,
* functionToMock: jest.fn(),
* };
* });
* test('mock api function', () => {
* functionToMock.mockImplementation(() => ({ mockedValue: 2 }));
* });

**10. 测试异步函数**

/\*\*

\* 模拟 TenvideoJSBridge.invoke 通过回调返回结果

\* @param response 返回值

\* @param options 选项

\*/

export function mockInvokeResponse(response: AnyObject, options: { delay?: number } = {}) {

const { delay = 0 } = options;

const invoke: TenvideoJSBridge['invoke'] = jest.fn((\_apiName, \_params, callback) => {

if (delay > 0) {

setTimeout(() => callback?.(response), delay);

return;

}

callback?.(response);

});

Object.defineProperty(window, 'TenvideoJSBridge', {

writable: true,

value: {

invoke,

},

});

}

1. 使用 done
2. test('The onTenvideoJSBridgeReady event is triggered', (done) => {
3. const bridge = new AppJSBridge();
4. expect.assertions(1);
5. bridge.invoke('getAppInfo').then((res) => {
6. expect(res).toEqual({
7. errCode: -403,
8. errMsg: 'TenvideoJSBridge 在非 APP 环境下，不支持调用 JSAPI',
9. result: null,
10. });
11. done();
12. });
13. document.dispatchEvent(new Event('onTenvideoJSBridgeReady'));
14. });
15. 在测试用例直接返回 promise
16. test('TenvideoJSBridge.invoke return a plain object without "result" field', () => {
17. const response = {
18. element: 'Video',
19. layout: {
20. posX: 0,
21. posY: 0,
22. width: 750,
23. height: 375,
24. },
25. };
26. mockInvokeResponse(response);
27. const bridge = new AppJSBridge();
28. return bridge
29. .invoke('getNativeElementLayout', { element: 'Video' }, { timeout: 1500 })
30. .then((res) => {
31. expect(window.TenvideoJSBridge?.invoke).toHaveBeenCalled();
32. expect(res).toEqual(response);
33. })
34. .catch((err) => {
35. expect(err).toEqual(new Error('timeout'));
36. });
37. });
38. 使用 async 和 await
39. test('TenvideoJSBridge.invoke timeout', async () => {
40. mockInvokeResponse('{}', { delay: 2000 });
41. const bridge = new AppJSBridge();
42. const apiName = 'getMainUserInfo';
43. try {
44. await bridge.invoke(apiName, null, { timeout: 1500 });
45. } catch (err) {
46. expect(err).toEqual(new Error(`TenvideoJSBridge.invoke('${apiName}') 调用超时无响应`));
47. }
48. });
49. 使用 [.resolves](https://jestjs.io/docs/asynchronous#resolves--rejects) 和 [.rejects](https://jestjs.io/docs/asynchronous#resolves--rejects)
50. test('the data is peanut butter', () => {
51. return expect(fetchData()).resolves.toBe('peanut butter');
52. });
53. test('the fetch fails with an error', () => {
54. return expect(fetchData()).rejects.toMatch('error');
55. });
56. // 结合 async 和 await 使用
57. test('the data is peanut butter', async () => {
58. await expect(fetchData()).resolves.toBe('peanut butter');
59. });
60. test('the fetch fails with an error', async () => {
61. await expect(fetchData()).rejects.toMatch('error');
62. });

**11. 使用模拟计时器**

常用的计时器函数包括 setTimeout、setInterval、clearTimeout、clearInterval 等，用到這些计时器的函数要等到倒计时结束后触发。如果计时器要跑几秒后才出发，或是要确认某函数是否在固定周期内被调用了几次，不可能真的去等待倒计时完后才能验证结果，那会很浪费时间。所以应该要使用 mock 函数來 mock 掉那些计时器，通过 Jest 提供的功能來控制时间，其中就有时间快进的功能，减少测试要等待的时间

* [jest.useFakeTimers(implementation?: 'modern' | 'legacy')](https://jestjs.io/docs/jest-object#jestusefaketimersimplementation-modern--legacy) 使用模拟的计时器函数
* [jest.useRealTimers()](https://jestjs.io/docs/jest-object#jestuserealtimers) 使用真实的计时器函数
* [jest.clearAllTimers()](https://jestjs.io/docs/jest-object#jestclearalltimers) 清除当前所有 pending 的计时器
* [jest.advanceTimersByTime(msToRun)](https://jestjs.io/docs/jest-object#jestadvancetimersbytimemstorun) 调用会将所有计时器提前 msToRun 毫秒
* [jest.runOnlyPendingTimers()](https://jestjs.io/docs/jest-object#jestrunonlypendingtimers) 马上执行当前 pending macro-tasks 触发回调 (即只执行 setTimeout() 或 setInterval() 至目前为止已 queued (排队) 的 task), 如果在当前的 setTimeout 回调内又使用了 setTimeout，则新的计时器就不能 jest.runOnlyPendingTimers() 触发回调
* afterEach(() => {
* jest.clearAllTimers();
* jest.useRealTimers();
* });
* test('TenvideoJSBridge is not injected after timeout 1500ms', (done) => {
* jest.useFakeTimers();
* const bridge = new AppJSBridge();
* const apiName = 'getAppInfo';
* bridge.invoke(apiName, null, { timeout: 1500 }).catch((err) => {
* expect(err).toEqual(new Error(`TenvideoJSBridge.invoke('${apiName}') 调用超时无响应`)); // ✅
* done();
* });
* window.dispatchEvent(new Event('load'));
* jest.advanceTimersByTime(1800);
* expect(window.TenvideoJSBridge).toBe(undefined);
* });
* test('TenvideoJSBridge.invoke timeout', (done) => {
* jest.useFakeTimers();
* mockInvokeResponse('{}', { delay: 2000 });
* const bridge = new AppJSBridge();
* const apiName = 'getMainUserInfo';
* bridge.invoke(apiName, null, { timeout: 1500 }).catch((err) => {
* expect(err).toEqual(new Error(`TenvideoJSBridge.invoke('${apiName}') 调用超时无响应`)); // ✅
* done();
* });
* jest.runOnlyPendingTimers();
* });

**12. 使用 snapshot 快照测试UI组件**

import { shallowMount } from "@vue/test-utils";

import LightCountDown from "../LightCountDown.vue";

describe('LightCountDown.vue', () => {

const data = {

teamType: 'count',

landscape: true,

src: 'http://placehold.it/100x100',

title: '恭喜全场电力达到188万',

desc: '<span class="count-num">3</span>s后即将解锁全员特效',

color: '',

};

test('snapshot', () => {

const wrapper = shallowMount(LightCountDown, {

propsData: { ...data },

});

expect(wrapper.props('src')).toBe(data.src);

expect(wrapper).toMatchSnapshot();

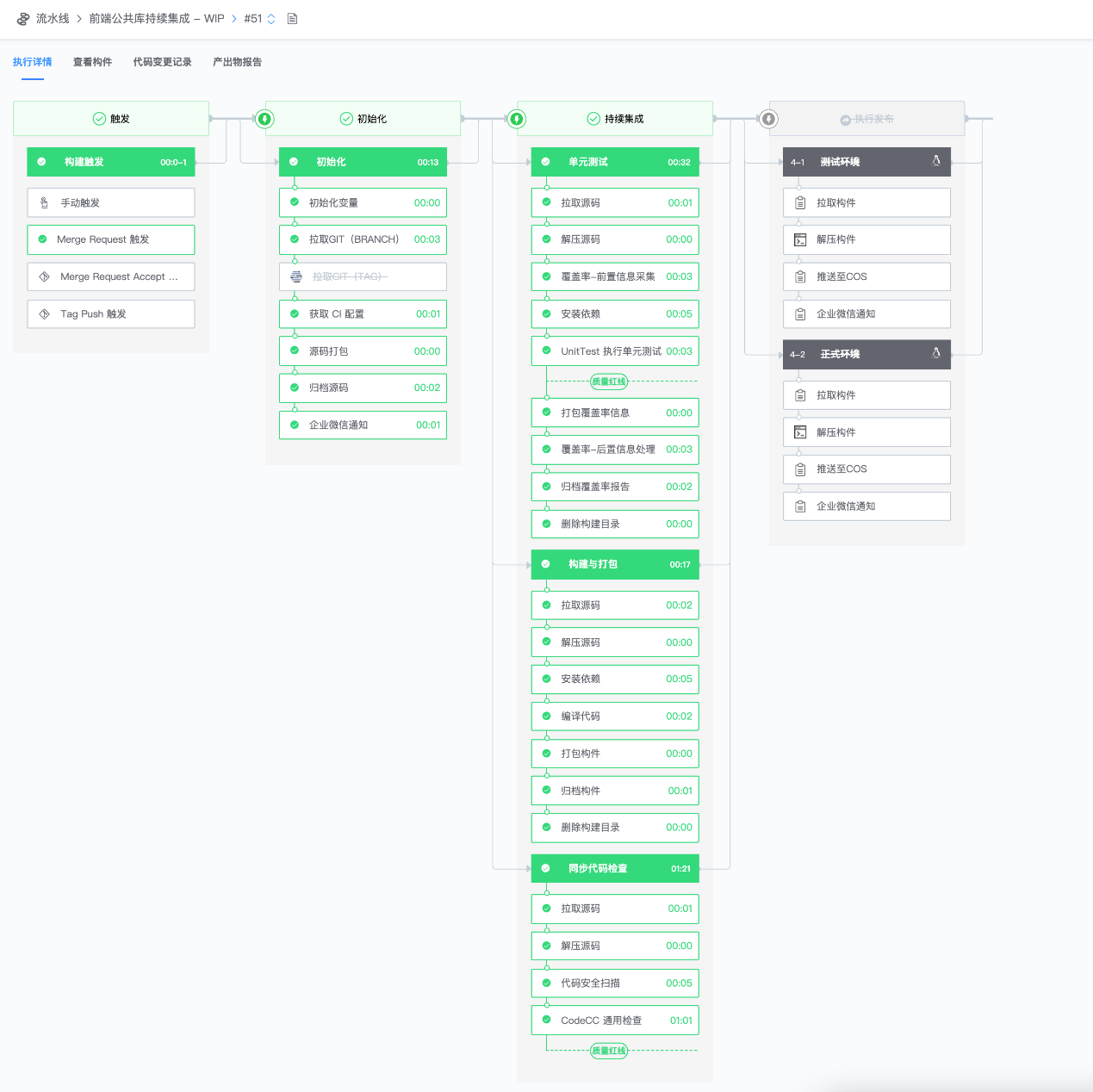
});

})

**持续集成**

通过 Merge Request 触发蓝盾流水线执行单元测试和上报覆盖率

* [JS使用流水线做单元测试&覆盖率搜集方案](https://iwiki.woa.com/pages/viewpage.action?pageId=727468844)
* [前端公共库持续集成流水线](http://devops.oa.com/console/pipeline/paycomponent/p-a6fd5ccb488c4405a6a8faf3e2d904d9/history)



**踩坑记录**

1. 在 monorepo (yarn workspace, lerna) 和 typescript 的项目下， 运行测试时，使用 —-no-cache，遇到了 ts-jest 编译失败的问题
2. FAIL @rich-gift/core \_\_tests\_\_/libs/jsbridge/AppQQVideoJSBridge.spec.ts
3. ● Test suite failed to run
4. TypeError: Unable to require `.d.ts` file.
5. This is usually the result of a faulty configuration or import. Make sure there is a `.js`, `.json` or another executable extension available alongside `index.ts`.
6. at getOutput (../../node\_modules/ts-jest/dist/compiler.js:165:23)
7. at Object.compile (../../node\_modules/ts-jest/dist/compiler.js:208:25)
8. at TsJestTransformer.process (../../node\_modules/ts-jest/dist/ts-jest-transformer.js:101:41)
9. at ScriptTransformer.transformSource (../../node\_modules/@jest/transform/build/ScriptTransformer.js:453:35)
10. at ScriptTransformer.\_transformAndBuildScript (../../node\_modules/@jest/transform/build/ScriptTransformer.js:523:40)
11. at ScriptTransformer.transform (../../node\_modules/@jest/transform/build/ScriptTransformer.js:579:25)

有两种解决方案：

1. 设置 tsconfig.json/compilerOptions.preserveSymlinks 为 true
2. 设置 jest.config.js/globals.ts-jest.isolatedModules 为 true

**附录**

**单元测试开发模式**

* TDD - (测试驱动开发）侧重点偏向开发，通过测试用例来规范约束开发者编写出质量更高、bug 更少的代码
* BDD - (行为驱动开发) 由外到内的开发方式，从外部定义业务成果，再深入到能实现这些成果，每个成果会转化成为相应的包含验收标准

简单来说就是 TDD 先写测试模块，再写主功能代码，然后能让测试模块通过测试，而 BDD 是先写主功能模块，再写测试模块

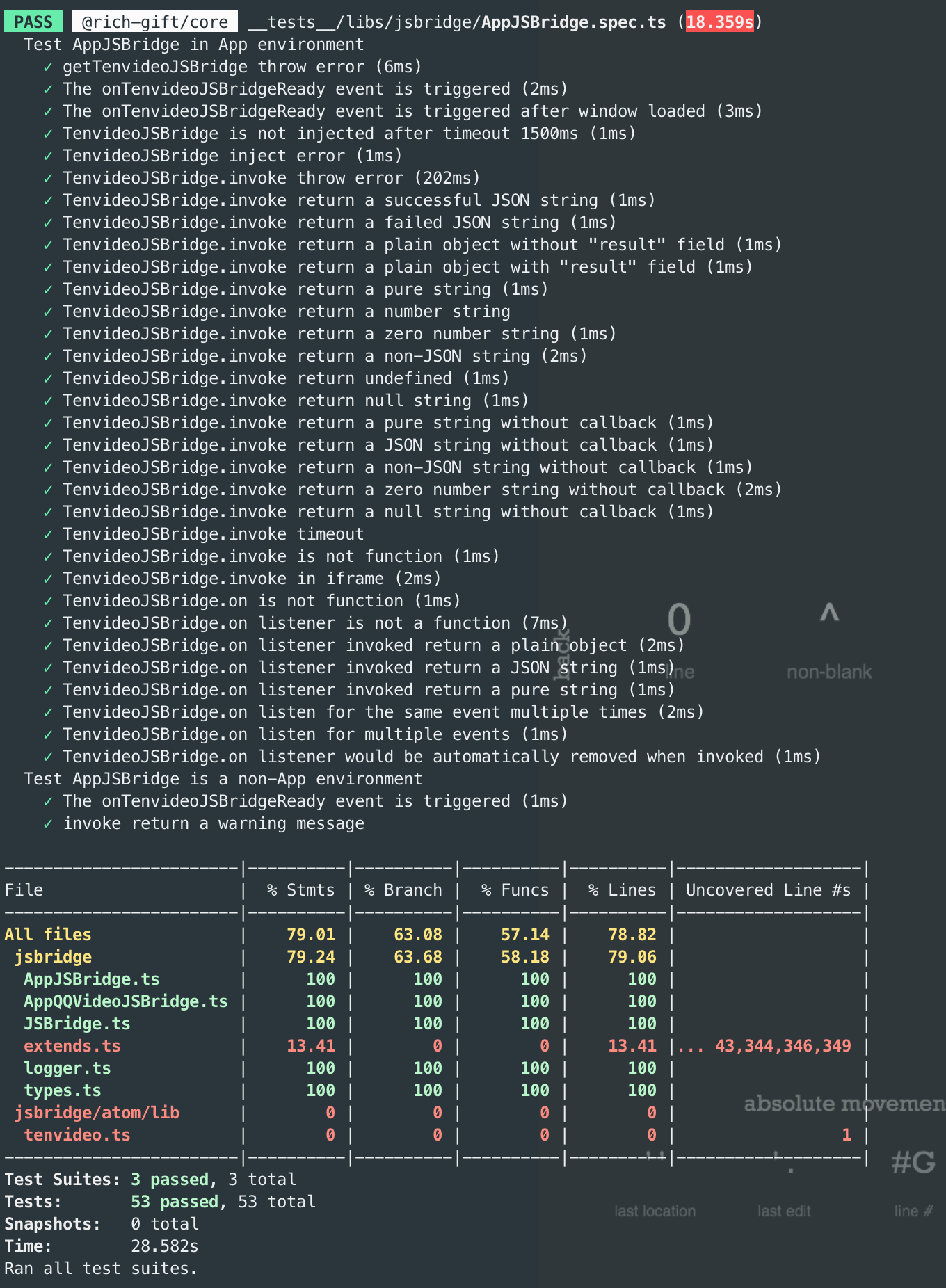
**Jest 常用配置项说明**

Jest 配置说明文档：<https://jestjs.io/docs/configuration>

* jest.config.js
* {
* // 设置 Jest 配置中 <rootDir> 模版字符串的值。默认是 Jest 配置所在的目录,如果你的配置写在 package.json 文件中，那就是 package.json 文件所在的目录，如果都没有，则是你运行 jest 命令所在的目录
* rootDir: './',
* // 定义 Jest 从哪些目录里面去搜索测试文件。默认值是 <rootDir>
* roots: ['<rootDir>'],
* // 定义在每个测试文件名旁边高亮显示的名字，方便区分不同包的测试
* displayName: '@rich-gift/core',
* // 预设配置
* preset: '@vue/cli-plugin-unit-jest/presets/typescript-and-babel',
* // 模块使用的文件扩展名
* moduleFileExtensions: [
* 'js',
* 'jsx',
* 'json',
* 'vue',
* 'ts',
* 'tsx'
* ],
* // 设置哪些文件中的代码是需要被相应的编译器转换成 Jest 能识别的代码
* transform: {
* // process \*.vue files with vue-jest
* '^.+\\.vue$': require.resolve('vue-jest'),
* '.+\\.(css|styl|less|sass|scss|svg|png|jpg|ttf|woff|woff2)$':
* require.resolve('jest-transform-stub'),
* '^.+\\.jsx?$': require.resolve('babel-jest'),
* '^.+\\.tsx?$': require.resolve('ts-jest')
* },
* // 设置哪些文件不需要编译的
* // 排除 node\_modules/lodash-es 包以外的都被忽略
* transformIgnorePatterns: ['/node\_modules/(?!lodash-es)'],
* // support the same @ -> src alias mapping in source code
* // 处理目录或模块别名的路径映射，比如：将 @ 表示 /src 目录
* moduleNameMapper: {
* '^@/(.\*)$': '<rootDir>/src/$1'
* },
* // 表示测试用例运行的环境
* testEnvironment: 'jest-environment-jsdom-fifteen',
* // serializer for snapshots
* // 将保存的快照测试结果进行序列化，使得其更美观
* snapshotSerializers: [
* 'jest-serializer-vue'
* ],
* // Jest使用 glob 模式来检测需要运行的测试文件
* testMatch: [
* '\*\*/tests/unit/\*\*/\*.spec.[jt]s?(x)',
* '\*\*/\_\_tests\_\_/\*.spec.[jt]s?(x)'
* ],
* // https://github.com/facebook/jest/issues/6766
* // 设置 jsdom 环境的 URL
* testURL: 'http://localhost/',
* // 设置文件监视插件
* watchPlugins: [
* require.resolve('jest-watch-typeahead/filename'),
* require.resolve('jest-watch-typeahead/testname')
* ],
* // 设置测试环境中的全局变量
* globals: {
* // ts-jest 编译配置 https://kulshekhar.github.io/ts-jest/docs/getting-started/options
* 'ts-jest': {
* // 禁用类型检查并将每个文件作为一个独立的模块进行编译
* isolatedModules: true,
* // 开启 Babel 编译
* // ts-jest 会尝试在你项目的 package.json文件中找到一个.babelrc、.babelrc.js、babel.config.js 文件或 babel 部分，并将其作为配置传递给 babel-jest 处理器
* babelConfig: true
* }
* },
* // 配置或设置测试环境，每个文件将在每个测试文件中运行一次
* setupFiles: ['./jest.setup.js'],
* // 初始化测试用例前要执行的js，一般我们会在这里提供一些全局的变量
* setupFilesAfterEnv: ['./tests/fixtures/inversify.test.config.ts'],
* // 是否在运行期间展示每个单独测试用例的测试情况（Pass or Fail），测试错误信息以及总体通过情况仍然会显示在底部
* // https://jestjs.io/zh-Hans/docs/configuration#verbose-boolean
* verbose: true,
* // 表示是否应该在执行测试的同时收集覆盖率信息
* collectCoverage: true,
* // 输出覆盖率文件的目录
* coverageDirectory: '<rootDir>/coverage',
* // 表示需要收集覆盖率报告的范围
* collectCoverageFrom: ['./libs/jsbridge/\*\*/!(index).{ts,tsx}', '!\*\*/node\_modules/\*\*'],
* // 配置自定义报告程序
* reporters: [
* 'default',
* '@tencent/dwt-reporter'
* ],
* }
* jest.steup.js
* import Vue from 'vue';
* import CompositionApi from '@vue/composition-api';
* Vue.use(CompositionApi);

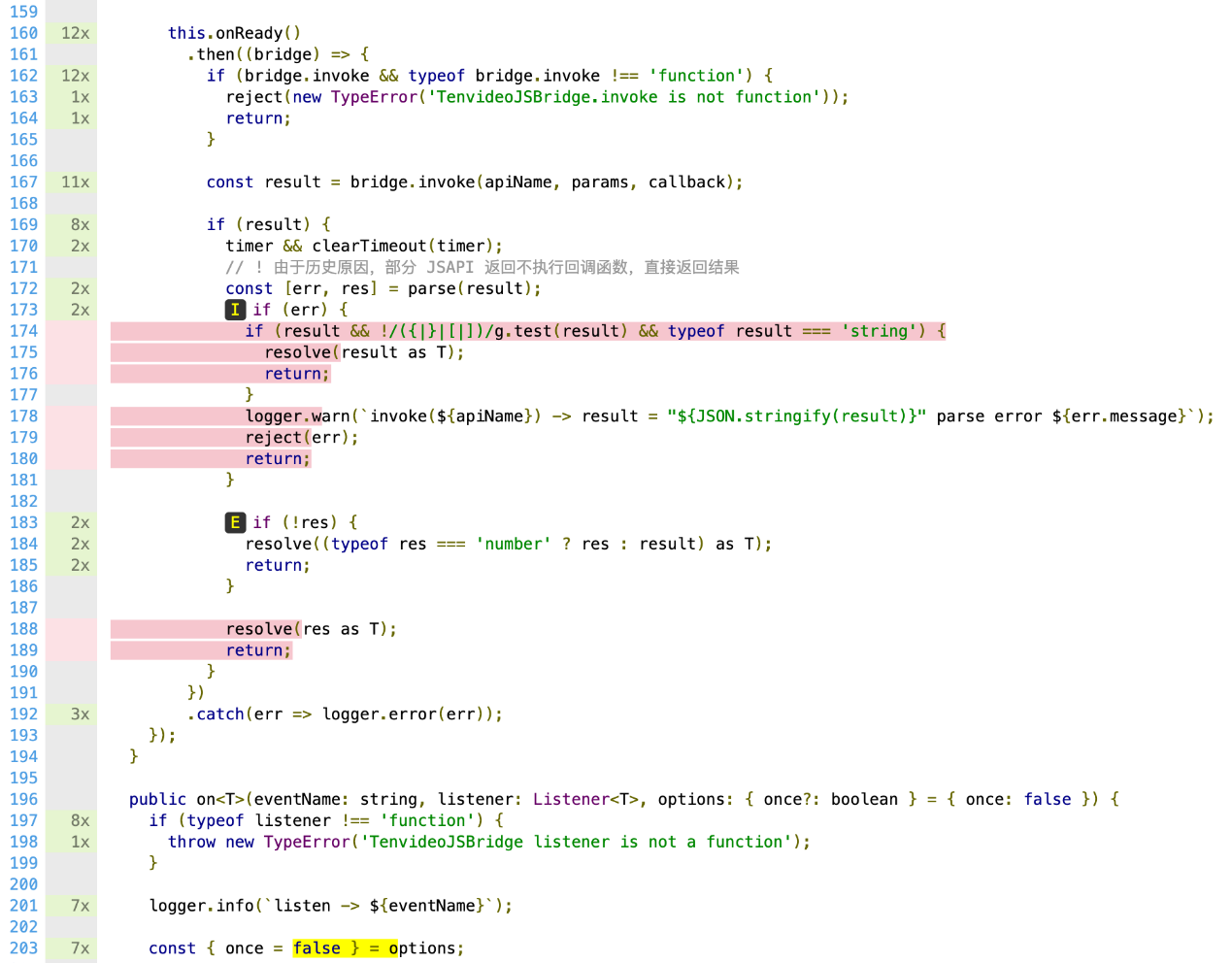
**覆盖率指标说明**

测试覆盖率能帮助我们了解我们的测试案例的有效性



* %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？

**覆盖率信息文件报告说明**

* E 表示 'else path not taken'，这意味着对于标记的 if / else 语句，if 语句已被测试到，但 else 语句没有被测试到
* I 表示 'if path not taken'，这是相反的情况，if 语句没有被测试到
* xN 表示该行被执行的次数
* 颜色说明：
  + 红色：未执行的行，或代码片断
  + 粉红色：未覆盖的语句
  + 橙色：未覆盖的函数
  + 黄色：未覆盖的分支

最后更新于 2021-09-07 11:23

本文申报

如果觉得我的文章对您有用，请随意赞赏

赏

仅供内部学习与交流，未经公司授权切勿外传

标签：[测试](https://km.woa.com/group/wfe/articles?tag_id=399)(1) [单元测试](https://km.woa.com/group/wfe/articles?tag_id=1395)(5) [Jest](https://km.woa.com/group/wfe/articles?tag_id=131727)(3) [前端](https://km.woa.com/group/wfe/articles?tag_id=960)(2) [UnitTest](https://km.woa.com/group/wfe/articles?tag_id=26082)(2)



本文专属二维码，扫一扫还能分享朋友圈

想要微信公众号推广本文章？[点击获取链接](javascript:void(0);)

我顶 (18)

收藏 (40)

* [转载 (1)](javascript:void(0);)

* [收录](javascript:void(0);)
* [评论](javascript:void(0);)(3)
* [反馈](javascript:void(0);)

分享到

大家评论

[](https://km.woa.com/user/tinniehe)

[tinniehe](https://km.woa.com/user/tinniehe)

2021-09-02 19:00:36

大佬就是牛 

 顶 [回复](javascript:void(0);)

[](https://km.woa.com/user/janusshi)

[janusshi](https://km.woa.com/user/janusshi)

2021-09-02 19:44:25

tql

 顶 [回复](javascript:void(0);)

[](https://km.woa.com/user/moonszhang)

[moonszhang](https://km.woa.com/user/moonszhang)

2021-09-03 10:19:22

tql

 顶 [回复](javascript:void(0);)

[](https://km.woa.com/user/rickyrqzhao)



[切换到更多功能](javascript:void(0);)

关于作者

[](https://km.woa.com/user/elvenzhang)

[elvenzhang(张奕恒)](https://km.woa.com/user/elvenzhang)

PCG\在线视频会员业务部\功能前端组员工

作者文章

* [前端单元测试总结](https://km.woa.com/posts/show/519991?kmref=author_post)
* [道具特效渲染优化](https://km.woa.com/posts/show/474496?kmref=author_post)
* [VIP前端阅读分享【第五周】](https://km.woa.com/posts/show/374840?kmref=author_post)

Copyright©1998-2021 Tencent Inc. All Rights Reserved

腾讯公司研发管理部 版权所有

[广告申请](https://km.woa.com/chartlets/km/) [反馈问题](javascript:void(0);)

[651/698/401 ms]