Jest 单元测试

> https://www.jestjs.cn/docs/getting-started

学习目标

- 了解单元测试与自动化测试概念
- 掌握 Jest 的基本用法 、 断言、mock
- 掌握 Vitest 完成组件的单元测试

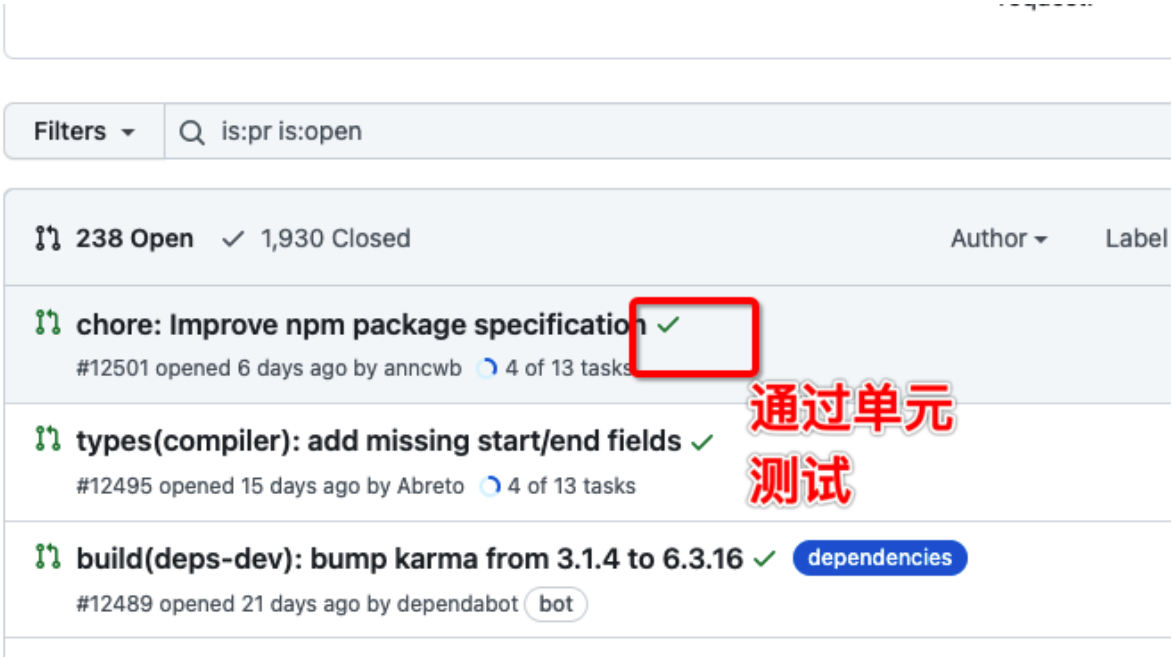
参考代码

my-jest.zip 2.64 MB

单元测试的必要性

假设有一个陌生人给 Vue 提交了一份代码，比如说你通过某种算法的优化提高的 vdom diff 的性能。尤大审核的时候通过静态比对觉得这个提交确实很有意义。但是并不能确定他对其他部分的代码会造成什么影响。这个时候会怎么办呢。是不是就陷入两难了。又想增加新的功能又不希望对原有功能造成影响。

其实在真实的环境中不会发生这样的问题，原因是在一个有良好工程化规范的项目中都会有单元测试的保驾护航。也就是说 Vue 的所有功能都会有对应的验证程序自动化进行确认。尤大神在看到你的代码提交的时候就已经能够看到这个代码更新后对以前的代码是否造成功能的影响了。



这个就是单元测试的必要性。

这个章节主要先学习单元测试是如何编写的。后面我们再学习如何自动运行单元测试。

单元测试的特点

1. 必要性：JavaScript 缺少类型检查，编译期间无法定位到错误，单元测试可以帮助你测试多种异常情况。
2. 正确性：测试可以验证代码的正确性，在上线前做到心里有底。
3. 自动化：通过 console 虽然可以打印出内部信息，但是这是一次性的事情，下次测试还需要从头来过，效率不能得到保证。通过编写测试用例，可以做到一次编写，多次运行。
4. 保证重构：互联网行业产品迭代速度很快，迭代后必然存在代码重构的过程，那怎么才能保证重构后代码的质量呢？有测试用例做后盾，就可以大胆的进行重构

Jest 完成一个单元测试

Jest 是 Facebook 开源的一套 JavaScript 测试框架， 它集成了断言、JSDom、覆盖率报告等开发者所需要的所有测试工具。

掌握内容

- jest-cli 的使用
- 目录规范
- 测试单元与用例 Case
- 断言

```bash
npm i jest -g
```

测试加法程序

add.js

```javascript
const add = (a, b) => a + b;
module.exports = add;
```

测试程序

./__tests__/add.spec.js

```javascript
const add = require("../add");

describe("测试Add函数", () => {
  test("add(1,2) === 3", () => {
    expect(add(1, 2)).toBe(3);
```

```
  });
  test("add(1,1) === 2", () => {
    expect(add(1, 1)).toBe(2);
  });
});
```

测试

```JavaScript
(base) ➜  my-npm jest
 PASS  src/__tests__/add.spec.js
  测试Add函数
    ✓ add(1,2) === 3 (1 ms)
    ✓ add(1,1) === 2

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.325 s
Ran all test suites.
```

概念总结

jest 文件和目录命名规范

待测试文件: `hello.js` 测试脚本文件取名: `hello.test.js` or `hello.spec.js`
测试目录: `tests` or `__tests__`

测试函数

```JavaScript
test("测试用列描述信息",()=>{

})

// or

it("测试用例描述信息",()=>{

})
```

断言函数

> 测试即运行结果是否与我们预期结果一致 断言函数用来验证结果是否正确

```JavaScript
expect(运行结果).toBe(期望的结果);
//常见断言方法
expect({a:1}).toBe({a:1})//判断两个对象是否相等
expect(1).not.toBe(2)//判断不等
expect({ a: 1, foo: { b: 2 } }).toEqual({ a: 1, foo: { b: 2 } })
expect(n).toBeNull(); //判断是否为null
expect(n).toBeUndefined(); //判断是否为undefined
expect(n).toBeDefined(); //判断结果与toBeUndefined相反
expect(n).toBeTruthy(); //判断结果为true
expect(n).toBeFalsy(); //判断结果为false
expect(value).toBeGreaterThan(3); //大于3
expect(value).toBeGreaterThanOrEqual(3.5); //大于等于3.5
expect(value).toBeLessThan(5); //小于5
expect(value).toBeLessThanOrEqual(4.5); //小于等于4.5
expect(value).toBeCloseTo(0.3); // 浮点数判断相等
expect('Christoph').toMatch(/stop/); //正则表达式判断
expect(['one','two']).toContain('one'); //不解释
```

分组函数

```JavaScript
describe("关于每个功能或某个组件的单元测试",()=>{

  // 不同用例的单元测试

})
```

常见命令

```JavaScript
{

  "nocache": "jest --no-cache", //清除缓存

  "watch": "jest --watchAll", //实时监听

  "coverage": "jest --coverage",  //生成覆盖测试文档

  "verbose": "npx jest --verbose" //显示测试描述

}
```

异步测试

异步测试脚本执行完，单元测试就结束了，如果需要延时才能断言的结果，单元测试函数需要设置 `done` 形参，在定时回调函数中调用，显示的通过单元测试已完成。

```JavaScript
module.exports = fn => {
    setTimeout(() => fn(), 1000)
}
```

```Bash
const delay = require('../delay')
it('异步测试', (done) => {
  delay(() => {
      done()
```

```
  })
})
```

快进功能

基于 jest 提供的两个方法 `jest.useFakeTimers` 和 `jest.runAllTimers` 可以更优雅的对延时功能的测试。

```JavaScript
const delay = require("../delay");
it("异步测试", (done) => {
  // 开启定时函数模拟
  jest.useFakeTimers();

  delay(() => {
    done();
  });
  //快进，使所有定时器回调
  jest.runAllTimers();
});
```

什么是 Mock 测试

如何孤立一个函数进行测试。

mock 测试就是在测试过程中，对于某些不容易构造或者不容易获取的对象，用一个虚拟的对象来创建以便测试的测试方法。

fetch.js

```JavaScript
const axios = require('axios')
exports.getData = () => axios.get('/abc/bcd')
```

- 需要调用 axios 库 → 就不是单元测试了
- axios 调用的 http 服务必须存在， 符合幂等性

```JavaScript
// __tests__/fetch.spec.js
const { getData } = require("../fetch");
const axios = require("axios");
jest.mock("axios");
it("fetch", async () => {
  // 模拟第一次接收到的数据
  axios.get.mockResolvedValueOnce("123");
  // 模拟每一次接收到的数据
  axios.get.mockResolvedValue("456");

  const data1 = await getData();
  const data2 = await getData();
  expect(data1).toBe("123");
  expect(data2).toBe("456");
});
```

其他用法

`jest.fn()`是创建 Mock 函数最简单的方式，如果没有定义函数内部的实现，`jest.fn()`会返回`undefined`作为返回值

```JavaScript
test('测试jest.fn()调用', () => {
  let mockFn = jest.fn();
  let result = mockFn(1, 2, 3);

  // 断言mockFn的执行后返回undefined
  expect(result).toBeUndefined();
  // 断言mockFn被调用
  expect(mockFn).toBeCalled();
  // 断言mockFn被调用了一次
  expect(mockFn).toBeCalledTimes(1);
  // 断言mockFn传入的参数为1, 2, 3
  expect(mockFn).toHaveBeenCalledWith(1, 2, 3);
})
```

jest.spyOn()

`jest.spyOn()` 方法同样创建一个 mock 函数，但是该 mock 函数不仅能够捕获函数的调用情况，还可以正常的执行被 spy 的函数。实际上，`jest.spyOn()` 是 `jest.fn()` 的语法糖，它创建了一个和被 spy 的函数具有相同内部代码的 mock 函数。

上图是之前 `jest.mock()` 的示例代码中的正确执行结果的截图，从 shell 脚本中可以看到 `console.log('fetchPostsList be called!');` 这行代码并没有在 shell 中被打印，这是因为通过 `jest.mock()` 后，模块内的方法是不会被 jest 所实际执行的。这时我们就需要使用 `jest.spyOn()`。

JavaScript

```
// functions.test.js

import events from '../src/events';

import fetch from '../src/fetch';

test('使用jest.spyOn()监控fetch.fetchPostsList被正常调用', async() => {

  expect.assertions(2);

  const spyFn = jest.spyOn(fetch, 'fetchPostsList');

  await events.getPostList();

  expect(spyFn).toHaveBeenCalled();

  expect(spyFn).toHaveBeenCalledTimes(1);

})
```

执行 `npm run test` 后，可以看到 shell 中的打印信息，说明通过 `jest.spyOn()`，`fetchPostsList` 被正常的执行了。

Dom 测试

所谓 Dom 测试是为了验证前端程序对 Dom 的操作是否正确。

为了测试方便，又不希望在浏览器环境中进行这时就可以在 Node 环境中进行，但是 Node 中并没有 Dom 模型。解决办法就是使用 jsdom 进行 Dom 的仿真。

JavaScript

```
// jsdom-config.js
const jsdom = require('jsdom') // eslint-disable-line
const { JSDOM } = jsdom

const dom = new JSDOM('<!DOCTYPE html><head/><body></body>', {
  url: 'http://localhost/',
```

```
  referrer: 'https://example.com/',
  contentType: 'text/html',
  userAgent: 'Mellblomenator/9000',
  includeNodeLocations: true,
  storageQuota: 10000000,
})
global.window = dom.window
global.document = window.document
global.navigator = window.navigator
```

- 生成 Dom 对象并绑定到全局

JavaScript

```
// dom.js
exports.generateDiv = () => {
  const div = document.createElement("div");
  div.className = "c1";
  document.body.appendChild(div);
};

// dom.test.js
const { generateDiv } = require('../dom')
require('../jsdom-config')
describe('Dom测试', () => {

    test('测试dom操作', () => {
        generateDiv()
        expect(document.getElementsByClassName('c1').length).toBe(1)
    })
})
```

快照测试

快照测试其实就是将对象实例序列化并进行持久化保存。等到然后进行代码比对。

就比如说在前端测试中，可以 Dom 对象做成快照。

每当你想要确保你的 UI 不会有意外的改变，快照测试是非常有用的工具。

典型的做法是在渲染了 UI 组件之后，保存一个快照文件， 检测他是否与保存在单元测试旁的快照文件相匹配。 若两个快照不匹配，测试将失败：有可能做了意外的更改，或者 UI 组件已经更新到了新版本。

JavaScript

```
// __tests__/snapshot.spec.js
const { generateDiv } = require('../dom')
require('../jsdom-config')
it("Dom的快照测试", () => {
    generateDiv()
    expect(document.getElementsByClassName('c1')).toMatchSnapshot()
})
```

Bash

```
jest    # 匹配快照
jest -u # 更新快照
```

Bash