**React单元测试方案**

**前置知识**

**为什么要进行测试**

1. 测试可以确保得到预期的结果
2. 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
4. 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

**测试类型**

* 单元测试：指的是以原件的单元为单位，对软件进行测试。单元可以是一个函数，也可以是一个模块或一个组件，基本特征就是只要输入不变，必定返回同样的输出。一个软件越容易些单元测试，就表明它的模块化结构越好，给模块之间的耦合越弱。React的组件化和函数式编程，天生适合进行单元测试
* 功能测试：相当于是黑盒测试，测试者不了解程序的内部情况，不需要具备编程语言的专门知识，只知道程序的输入、输出和功能，从用户的角度针对软件界面、功能和外部结构进行测试，不考虑内部的逻辑
* 集成测试：在单元测试的基础上，将所有模块按照设计要求组装成子系统或者系统，进行测试
* 冒烟测试：在正式全面的测试之前，对主要功能进行的与测试，确认主要功能是否满足需要，软件是否能正常运行

**开发模式**

* [TDD](https://link.juejin.im?target=https%3A%2F%2Fbaike.baidu.com%2Fitem%2FTDD%2F9064369): 测试驱动开发，英文为Testing Driven Development，强调的是一种开发方式，以测试来驱动整个项目，即先根据接口完成测试编写，然后在完成功能是要不断通过测试，最终目的是通过所有测试
* [BDD](https://link.juejin.im?target=https%3A%2F%2Fbaike.baidu.com%2Fitem%2FBDD%2F10735732%3Ffr%3Daladdin): 行为驱动测试，英文为Behavior Driven Development，强调的是写测试的风格，即测试要写的像自然语言，让项目的各个成员甚至产品都能看懂测试，甚至编写测试

TDD和BDD有各自的使用场景，BDD一般偏向于系统功能和业务逻辑的自动化测试设计；而TDD在快速开发并测试功能模块的过程中则更加高效，以快速完成开发为目的。

**技术选型：Jest + Enzyme**

**Jest**

Jest是Facebook开源的一个前端测试框架，主要用于React和React Native的单元测试，已被集成在create-react-app中。Jest特点：

1. 易用性：基于Jasmine，提供断言库，支持多种测试风格
2. 适应性：Jest是模块化、可扩展和可配置的
3. 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
4. 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
5. Mock系统：Jest实现了一个强大的Mock系统，支持自动和手动mock
6. 支持异步代码测试：支持Promise和async/await
7. 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

**Enzyme**

Enzyme是Airbnb开源的React测试工具库库，它功能过对官方的测试工具库ReactTestUtils的二次封装，提供了一套简洁强大的 API，并内置Cheerio，

实现了jQuery风格的方式进行DOM 处理，开发体验十分友好。在开源社区有超高人气，同时也获得了React 官方的推荐。

**测试环境搭建**

安装Jest、Enzyme，以及babel-jest。如果React的版本是15或者16，需要安装对应的enzyme-adapter-react-15和enzyme-adapter-react-16并配置。

import Enzyme from 'enzyme';

import Adapter from 'enzyme-adapter-react-16';

Enzyme.configure({ adapter: new Adapter() });

复制代码

在package.json中的script中增加"test: jest --config .jest.js"

.jest.js文件

module.exports = {

setupFiles: [

'./test/setup.js',

],

moduleFileExtensions: [

'js',

'jsx',

],

testPathIgnorePatterns: [

'/node\_modules/',

],

testRegex: '.\*\\.test\\.js$',

collectCoverage: false,

collectCoverageFrom: [

'src/components/\*\*/\*.{js}',

],

moduleNameMapper: {

"\\.(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|m4a|aac|oga)$": "<rootDir>/\_\_mocks\_\_/fileMock.js",

"\\.(css|less|scss)$": "<rootDir>/\_\_mocks\_\_/styleMock.js"

},

transform: {

"^.+\\.js$": "babel-jest"

},

};

复制代码

* setupFiles：配置文件，在运行测试案例代码之前，Jest会先运行这里的配置文件来初始化指定的测试环境
* moduleFileExtensions：代表支持加载的文件名
* testPathIgnorePatterns：用正则来匹配不用测试的文件
* testRegex：正则表示的测试文件，测试文件的格式为xxx.test.js
* collectCoverage：是否生成测试覆盖报告，如果开启，会增加测试的时间
* collectCoverageFrom：生成测试覆盖报告是检测的覆盖文件
* moduleNameMapper：代表需要被Mock的资源名称
* transform：用babel-jest来编译文件，生成ES6/7的语法

**Jest**

[**globals**](https://link.juejin.im?target=https%3A%2F%2Ffacebook.github.io%2Fjest%2Fdocs%2Fen%2Fapi.html) **API**

* describe(name, fn)：描述块，讲一组功能相关的测试用例组合在一起
* it(name, fn, timeout)：别名test，用来放测试用例
* afterAll(fn, timeout)：所有测试用例跑完以后执行的方法
* beforeAll(fn, timeout)：所有测试用例执行之前执行的方法
* afterEach(fn)：在每个测试用例执行完后执行的方法
* beforeEach(fn)：在每个测试用例执行之前需要执行的方法

全局和describe都可以有上面四个周期函数，describe的after函数优先级要高于全局的after函数，describe的before函数优先级要低于全局的before函数

beforeAll(() => {

console.log('global before all');

});

afterAll(() => {

console.log('global after all');

});

beforeEach(() =>{

console.log('global before each');

});

afterEach(() => {

console.log('global after each');

});

describe('test1', () => {

beforeAll(() => {

console.log('test1 before all');

});

afterAll(() => {

console.log('test1 after all');

});

beforeEach(() => {

console.log('test1 before each');

});

afterEach(() => {

console.log('test1 after each');

});

it('test sum', () => {

expect(sum(2, 3)).toEqual(5);

});

it('test mutil', () => {

expect(sum(2, 3)).toEqual(7);

});

});

复制代码

[**config**](https://link.juejin.im?target=https%3A%2F%2Ffacebook.github.io%2Fjest%2Fdocs%2Fen%2Fconfiguration.html)

Jest拥有丰富的配置项，可以写在package.json里增加增加jest字段来进行配置，或者通过命令行--config来指定配置文件。

[**jest对象**](https://link.juejin.im?target=https%3A%2F%2Fjestjs.io%2Fdocs%2Fen%2Fjest-object.html)

* jest.fn(implementation)：返回一个全新没有使用过的mock function，这个function在被调用的时候会记录很多和函数调用有关的信息
* jest.mock(moduleName, factory, options)：用来mock一些模块或者文件
* jest.spyOn(object, methodName)：返回一个mock function，和jest.fn相似，但是能够追踪object[methodName]的调用信息，类似Sinon

[**Mock Functions**](https://link.juejin.im?target=https%3A%2F%2Ffacebook.github.io%2Fjest%2Fdocs%2Fen%2Fmock-functions.html)

使用mock函数可以轻松的模拟代码之间的依赖，可以通过fn或spyOn来mock某个具体的函数；通过mock来模拟某个模块。具体的API可以看[mock-function-api](https://link.juejin.im?target=https%3A%2F%2Ffacebook.github.io%2Fjest%2Fdocs%2Fen%2Fmock-function-api.html" \t "_blank)。

[**快照**](https://link.juejin.im?target=https%3A%2F%2Ffacebook.github.io%2Fjest%2Fdocs%2Fen%2Fsnapshot-testing.html)

快照会生成一个组件的UI结构，并用字符串的形式存放在\_\_snapshots\_\_文件里，通过比较两个字符串来判断UI是否改变，因为是字符串比较，所以性能很高。

要使用快照功能，需要引入react-test-renderer库，使用其中的renderer方法，jest在执行的时候如果发现toMatchSnapshot方法，会在同级目录下生成一个\_\_snapshots文件夹用来存放快照文件，以后每次测试的时候都会和第一次生成的快照进行比较。可以使用jest --updateSnapshot来更新快照文件。

[**异步测试**](https://link.juejin.im?target=https%3A%2F%2Ffacebook.github.io%2Fjest%2Fdocs%2Fen%2Fasynchronous.html)

Jest支持对异步的测试，支持Promise和Async/Await两种方式的异步测试。

[**常见断言**](https://link.juejin.im?target=https%3A%2F%2Ffacebook.github.io%2Fjest%2Fdocs%2Fzh-Hans%2Fexpect.html)

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. 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
3. not：用来取反
4. toEqual(value)：用于对象的深比较
5. 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
6. 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
7. toBeNull(value)：只匹配null
8. toBeUndefined(value)：只匹配undefined
9. toBeDefined(value)：与toBeUndefined相反
10. toBeTruthy(value)：匹配任何使if语句为真的值
11. toBeFalsy(value)：匹配任何使if语句为假的值
12. toBeGreaterThan(number)： 大于
13. toBeGreaterThanOrEqual(number)：大于等于
14. toBeLessThan(number)：小于
15. toBeLessThanOrEqual(number)：小于等于
16. toBeInstanceOf(class)：判断是不是class的实例
17. anything(value)：匹配除了null和undefined以外的所有值
18. resolves：用来取出promise为fulfilled时包裹的值，支持链式调用
19. rejects：用来取出promise为rejected时包裹的值，支持链式调用
20. toHaveBeenCalled()：用来判断mock function是否被调用过
21. toHaveBeenCalledTimes(number)：用来判断mock function被调用的次数
22. assertions(number)：验证在一个测试用例中有number个断言被调用
23. extend(matchers)：自定义一些断言

**Enzyme**

**三种渲染方法**

1. shallow：浅渲染，是对官方的Shallow Renderer的封装。将组件渲染成虚拟DOM对象，只会渲染第一层，子组件将不会被渲染出来，使得效率非常高。不需要DOM环境， 并可以使用jQuery的方式访问组件的信息
2. render：静态渲染，它将React组件渲染成静态的HTML字符串，然后使用Cheerio这个库解析这段字符串，并返回一个Cheerio的实例对象，可以用来分析组件的html结构
3. mount：完全渲染，它将组件渲染加载成一个真实的DOM节点，用来测试DOM API的交互和组件的生命周期。用到了jsdom来模拟浏览器环境

三种方法中，shallow和mount因为返回的是DOM对象，可以用simulate进行交互模拟，而render方法不可以。一般shallow方法就可以满足需求，如果需要对子组件进行判断，需要使用render，如果需要测试组件的生命周期，需要使用mount方法。

**常用方法**

1. simulate(event, mock)：模拟事件，用来触发事件，event为事件名称，mock为一个event object
2. instance()：返回组件的实例
3. find(selector)：根据选择器查找节点，selector可以是CSS中的选择器，或者是组件的构造函数，组件的display name等
4. at(index)：返回一个渲染过的对象
5. get(index)：返回一个react node，要测试它，需要重新渲染
6. contains(nodeOrNodes)：当前对象是否包含参数重点 node，参数类型为react对象或对象数组
7. text()：返回当前组件的文本内容
8. html()： 返回当前组件的HTML代码形式
9. props()：返回根组件的所有属性
10. prop(key)：返回根组件的指定属性
11. state()：返回根组件的状态
12. setState(nextState)：设置根组件的状态
13. setProps(nextProps)：设置根组件的属性

**编写测试用例**

**组件代码**

todo-list/index.js

import React, { Component } from 'react';

import { Button } from 'antd';

export default class TodoList extends Component {

constructor(props) {

super(props);

this.handleTest2 = this.handleTest2.bind(this);

}

handleTest = () => {

console.log('test');

}

handleTest2() {

console.log('test2');

}

componentDidMount() {}

render() {

return (

<div className="todo-list">

{this.props.list.map((todo, index) => (<div key={index}>

<span className="item-text ">{todo}</span>

<Button onClick={() => this.props.deleteTodo(index)} >done</Button>

</div>))}

</div>

);

}

}

复制代码

**测试文件setup设置**

const props = {

list: ['first', 'second'],

deleteTodo: jest.fn(),

};

const setup = () => {

const wrapper = shallow(<TodoList {...props} />);

return {

props,

wrapper,

};

};

const setupByRender = () => {

const wrapper = render(<TodoList {...props} />);

return {

props,

wrapper,

};

};

const setupByMount = () => {

const wrapper = mount(<TodoList {...props} />);

return {

props,

wrapper,

};

};

复制代码

**使用 snapshot 进行 UI 测试**

it('renders correctly', () => {

const tree = renderer

.create(<TodoList {...props} />)

.toJSON();

expect(tree).toMatchSnapshot();

});

复制代码

当使用toMatchSnapshot的时候，会生成一份组件DOM的快照，以后每次运行测试用例的时候，都会生成一份组件快照和第一次生成的快照进行对比，如果对组件的结构进行修改，那么生成的快照就会对比失败。可以通过更新快照重新进行UI测试。

**对组件节点进行测试**

it('should has Button', () => {

const { wrapper } = setup();

expect(wrapper.find('Button').length).toBe(2);

});

it('should render 2 item', () => {

const { wrapper } = setupByRender();

expect(wrapper.find('button').length).toBe(2);

});

it('should render item equal', () => {

const { wrapper } = setupByMount();

wrapper.find('.item-text').forEach((node, index) => {

expect(node.text()).toBe(wrapper.props().list[index])

});

});

it('click item to be done', () => {

const { wrapper } = setupByMount();

wrapper.find('Button').at(0).simulate('click');

expect(props.deleteTodo).toBeCalled();

});

复制代码

判断组件是否有Button这个组件，因为不需要渲染子节点，所以使用shallow方法进行组件的渲染，因为props的list有两项，所以预期应该有两个Button组件。

判断组件是否有button这个元素，因为button是Button组件里的元素，所有使用render方法进行渲染，预期也会找到连个button元素。

判断组件的内容，使用mount方法进行渲染，然后使用forEach判断.item-text的内容是否和传入的值相等使用simulate来触发click事件，因为deleteTodo被mock了，所以可以用deleteTodo方法时候被调用来判断click事件是否被触发。

**测试组件生命周期**

//使用spy替身的时候，在测试用例结束后，要对spy进行restore，不然这个spy会一直存在，并且无法对相同的方法再次进行spy。

it('calls componentDidMount', () => {

const componentDidMountSpy = jest.spyOn(TodoList.prototype, 'componentDidMount');

const { wrapper } = setup();

expect(componentDidMountSpy).toHaveBeenCalled();

componentDidMountSpy.mockRestore();

});

复制代码

使用spyOn来mock 组件的componentDidMount，替身函数要在组件渲染之前，所有替身函数要定义在setup执行之前，并且在判断以后要对替身函数restore，不然这个替身函数会一直存在，且被mock的那个函数无法被再次mock。

**测试组件的内部函数**

it('calls component handleTest', () => { // class中使用箭头函数来定义方法

const { wrapper } = setup();

const spyFunction = jest.spyOn(wrapper.instance(), 'handleTest');

wrapper.instance().handleTest();

expect(spyFunction).toHaveBeenCalled();

spyFunction.mockRestore();

});

it('calls component handleTest2', () => { //在constructor使用bind来定义方法

const spyFunction = jest.spyOn(TodoList.prototype, 'handleTest2');

const { wrapper } = setup();

wrapper.instance().handleTest2();

expect(spyFunction).toHaveBeenCalled();

spyFunction.mockRestore();

});

复制代码

使用instance函数来取得组件的实例，并用spyOn方法来mock实例上的内部方法，然后用这个实例去调用那个内部方法，就可以用替身来判断这个内部函数是否被调用。如果内部方法是用箭头函数来定义的时候，需要对实例进行mock；如果内部方法是通过正常的方式或者bind的方式定义的，那么需要对组件的prototype进行mock。其实对生命周期或者内部函数的测试，可以通过一些state的改变进行判断，因为这些函数的调用一般都会对组件的state进行一些操作。

**Manual Mocks**

1. 对全局的模块(moduleName)进行手动模拟，需要在node\_modules平级的位置新建一个\_\_mocks\_\_文件夹，并在文件夹中新建一个moduleName的文件
2. 对某个文件(fileName)进行手动模拟，需要在被模拟的文件平级的位置新建一个\_\_mocks\_\_文件夹，然后在文件夹中新建一个fileName的文件

add/index.js

import { add } from 'lodash';

import { multip } from '../../utils/index';

export default function sum(a, b) {

return add(a, b);

}

export function m(a, b) {

return multip(a, b);

}

复制代码

add/\_\_test\_\_/index.test.js

import sum, { m } from '../index';

jest.mock('lodash');

jest.mock('../../../utils/index');

describe('test mocks', () => {

it('test sum', () => {

expect(sum(2, 3)).toEqual(5);

});

it('test mutilp', () => {

expect(m(2, 3)).toEqual(7);

});

});

复制代码

\_*mocks*\_:

在测试文件中使用mock()方法对要进行mock的文件进行引用，Jest就会自动去寻找对应的\_\_mocks\_\_中的文件并进行替换，lodash中的add和utils中的multip方法就会被mock成对应的方法。可以使用自动代理的方式对项目的异步组件库(fetch、axios)进行mock，或者使用fetch-mock、jest-fetch-mock来模拟异步请求。

**对异步方法进行测试**

async/index.js

import request from './request';

export function getUserName(userID) {

return request(`/users/${userID}`).then(user => user.name);

}

async/request.js

const http = require('http');

export default function request(url) {

return new Promise((resolve) => {

// This is an example of an http request, for example to fetch

// user data from an API.

// This module is being mocked in \_\_mocks\_\_/request.js

http.get({ path: url }, (response) => {

let data = '';

response.on('data', \_data => (data += \_data));

response.on('end', () => resolve(data));

});

});

}

复制代码

mock request:

const users = {

4: {

name: 'hehe',

},

5: {

name: 'haha',

},

};

export default function request(url) {

return new Promise((resolve, reject) => {

const userID = parseInt(url.substr('/users/'.length), 10);

process.nextTick(() => {

users[userID] ?

resolve(users[userID]) :

reject({

error: `User with ${userID} not found.`,

});

});

});

}

复制代码

request.js可以看成是一个用于请求数据的模块，手动mock这个模块，使它返回一个Promise对象，用于对异步的处理。

**测试Promise**

// 使用'.resolves'来测试promise成功时返回的值

it('works with resolves', () => {

// expect.assertions(1);

expect(user.getUserName(5)).resolves.toEqual('haha')

});

// 使用'.rejects'来测试promise失败时返回的值

it('works with rejects', () => {

expect.assertions(1);

return expect(user.getUserName(3)).rejects.toEqual({

error: 'User with 3 not found.',

});

});

// 使用promise的返回值来进行测试

it('test resolve with promise', () => {

expect.assertions(1);

return user.getUserName(4).then((data) => {

expect(data).toEqual('hehe');

});

});

it('test error with promise', () => {

expect.assertions(1);

return user.getUserName(2).catch((e) => {

expect(e).toEqual({

error: 'User with 2 not found.',

});

});

});

复制代码

当对Promise进行测试时，一定要在断言之前加一个return，不然没有等到Promise的返回，测试函数就会结束。可以使用.promises/.rejects对返回的值进行获取，或者使用then/catch方法进行判断。

**测试Async/Await**

// 使用async/await来测试resolve

it('works resolve with async/await', async () => {

expect.assertions(1);

const data = await user.getUserName(4);

expect(data).toEqual('hehe');

});

// 使用async/await来测试reject

it('works reject with async/await', async () => {

expect.assertions(1);

try {

await user.getUserName(1);

} catch (e) {

expect(e).toEqual({

error: 'User with 1 not found.',

});

}

});

复制代码

使用async不用进行return返回，并且要使用try/catch来对异常进行捕获。

**代码覆盖率**

代码覆盖率是一个测试指标，用来描述测试用例的代码是否都被执行。统计代码覆盖率一般要借助代码覆盖工具，Jest集成了Istanbul这个代码覆盖工具。

**四个测量维度**

1. 行覆盖率(line coverage)：是否测试用例的每一行都执行了
2. 函数覆盖率(function coverage)：师傅测试用例的每一个函数都调用了
3. 分支覆盖率(branch coverage)：是否测试用例的每个if代码块都执行了
4. 语句覆盖率(statement coverage)：是否测试用例的每个语句都执行了

在四个维度中，如果代码书写的很规范，行覆盖率和语句覆盖率应该是一样的。会触发分支覆盖率的情况有很多种，主要有以下几种：

* ||，&&，？，！
* if语句
* switch语句

**例子**

function test(a, b) {

a = a || 0;

b = b || 0;

if (a && b) {

return a + b;

} else {

return 0;

}

}

test(1, 2);

// test();

复制代码

当执行test(1,2)的时候，代码覆盖率为

当执行test()的时候，代码覆盖率为

**设置阈值**

stanbul可以在命令行中设置各个覆盖率的门槛，然后再检查测试用例是否达标，各个维度是与的关系，只要有一个不达标，就会报错。

当statement和branch设置为90的时候，覆盖率检测会报

当statemen设置为80t、branch设置为50的时候，覆盖率检测会通过

在Jest中，可以通过coverageThreshold这个配置项来设置不同测试维度的覆盖率阈值。global是全局配置，默认所有的测试用例都要满足这个配置才能通过测试。还支持通配符模式或者路径配置，如果存在这些配置，那么匹配到的文件的覆盖率将从全局覆盖率的计算中去除，独立使用各自设置的阈值。

{

...

"jest": {

"coverageThreshold": {

"global": {

"branches": 50,

"functions": 50,

"lines": 50,

"statements": 50

},

"./src/components/": {

"branches": 40,

"statements": 40

},

"./src/reducers/\*\*/\*.js": {

"statements": 90,

},

"./src/api/very-important-module.js": {

"branches": 100,

"functions": 100,

"lines": 100,

"statements": 100

}

}

}

}

复制代码

**集成到脚手架**

在项目中引用单元测试后，希望每次修改需要测试的文件时，能在提交代码前自动跑一边测试用例，保证代码的正确性和健壮性。

在项目中可以使用husky和lint-staged，用来触发git的[hooks](https://link.juejin.im?target=https%3A%2F%2Fgit-scm.com%2Fbook%2Fzh%2Fv2%2F%25E8%2587%25AA%25E5%25AE%259A%25E4%25B9%2589-Git-Git-%25E9%2592%25A9%25E5%25AD%2590" \t "_blank)，做一些代码提交前的校验。

* [husky](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Ftypicode%2Fhusky)：在项目中安装husky以后，会在 .git/hooks 中写入 pre-commit 等脚本激活[钩子](https://link.juejin.im?target=https%3A%2F%2Fgit-scm.com%2Fbook%2Fzh%2Fv2%2F%25E8%2587%25AA%25E5%25AE%259A%25E4%25B9%2589-Git-Git-%25E9%2592%25A9%25E5%25AD%2590" \t "_blank)，在 Git 进行相关操作时触发
* [lint-staged](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Fokonet%2Flint-staged)：名字中的staged表示的就是Git中的暂存区，它只会对将要加入暂存区中的内容进行lint

在package.json中，precommit执行lint-staged，对lint-staged进行配置，对所有的js文件进行eslint检查，对src/components中的js文件进行测试。

{

"scripts": {

"precommit": "lint-staged",

},

"lint-staged": {

"ignore": [

"build/\*",

"node\_modules"

],

"linters": {

"src/\*.js": [

"eslint --fix",

"git add"

],

"src/components/\*\*/\*.js": [

"jest --findRelatedTests --config .jest.js",

"git add"

]

}

},

}

复制代码

对containers中的文件进行修改，然后推进暂存区的时候，会进行eslint的检查，但是不会进行测试

对components中的todo-list进行修改，eslint会进行检查，并且会执行todo-list这个组件的测试用例，因为改变了组件的结构，所以快照进行UI对比就会失败

作者：supot  
链接：https://juejin.im/post/5b6c39bde51d45195c079d62  
来源：掘金  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。