**对 React 组件进行单元测试**

前端开发的一个特点是更多的会涉及用户界面，当开发规模达到一定程度时，几乎注定了其复杂度会成倍的增长。

无论是在代码的初始搭建过程中，还是之后难以避免的重构和修正bug过程中，常常会陷入逻辑难以梳理、无法掌握全局关联的境地。

而单元测试作为一种“提纲挈领、保驾护航”的基础手段，为开发提供了“围墙和脚手架”，可以有效的改善这些问题。

作为一种经典的开发和重构手段，单元测试在软件开发领域被广泛认可和采用；前端领域也逐渐积累起了丰富的测试框架和最佳实践。

*本文将按如下顺序进行说明:*

* I. 单元测试简介
* II. React 单元测试中用到的工具
* III. 用测试驱动 React 组件重构
* IV. React 单元测试常见案例

**I. 单元测试简介**

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。

简单来说，单元就是人为规定的最小的被测功能模块。单元测试是在软件开发过程中要进行的最低级别的测试活动，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。

**测试框架**

测试框架的作用是提供一些方便的语法来描述测试用例，以及对用例进行分组。

**断言(assertions)**

断言是单元测试框架中核心的部分，断言失败会导致测试不通过，或报告错误信息。

对于常见的断言，举一些例子如下：

* 同等性断言 Equality Asserts
  + expect(sth).toEqual(value)
  + expect(sth).not.toEqual(value)
* 比较性断言 Comparison Asserts
  + expect(sth).toBeGreaterThan(number)
  + expect(sth).toBeLessThanOrEqual(number)
* 类型性断言 Type Asserts
  + expect(sth).toBeInstanceOf(Class)
* 条件性测试 Condition Test
  + expect(sth).toBeTruthy()
  + expect(sth).toBeFalsy()
  + expect(sth).toBeDefined()

**断言库**

断言库主要提供上述断言的语义化方法，用于对参与测试的值做各种各样的判断。这些语义化方法会返回测试的结果，要么成功、要么失败。常见的断言库有 Should.js, Chai.js 等。

**测试用例 test case**

为某个特殊目标而编制的一组测试输入、执行条件以及预期结果，以便测试某个程序路径或核实是否满足某个特定需求。

一般的形式为：

it('should ...', **function**() {

...

expect(sth).toEqual(sth);

});

复制代码

**测试套件 test suite**

通常把一组相关的测试称为一个测试套件

一般的形式为：

describe('test ...', **function**() {

it('should ...', **function**() { ... });

it('should ...', **function**() { ... });

...

});

复制代码

**spy**

正如 spy 字面的意思一样，我们用这种“间谍”来“监视”函数的调用情况

通过对监视的函数进行包装，可以通过它清楚的知道该函数被调用过几次、传入什么参数、返回什么结果，甚至是抛出的异常情况。

var spy = sinon.spy(MyComp.prototype, 'componentDidMount');

...

expect(spy.callCount).toEqual(1);

复制代码

**stub**

有时候会使用stub来嵌入或者直接替换掉一些代码，来达到隔离的目的

一个stub可以使用最少的依赖方法来模拟该单元测试。比如一个方法可能依赖另一个方法的执行，而后者对我们来说是透明的。好的做法是使用stub 对它进行隔离替换。这样就实现了更准确的单元测试。

var myObj = {

prop: **function**() {

return 'foo';

}

};

sinon.stub(myObj, 'prop').callsFake(**function**() {

return 'bar';

});

myObj.prop(); // 'bar'

复制代码

**mock**

mock一般指在测试过程中，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

广义的讲，以上的 spy 和 stub 等，以及一些对模块的模拟，对 ajax 返回值的模拟、对 timer 的模拟，都叫做 mock 。

**测试覆盖率(code coverage)**

用于统计测试用例对代码的测试情况，生成相应的报表，比如 istanbul 是常见的测试覆盖率统计工具

**II. React 单元测试中用到的工具**

**Jest**

不同于"传统的"(其实也没出现几年)的 jasmine / Mocha / Chai 等前端测试框架 -- Jest的使用更简单，并且提供了更高的集成度、更丰富的功能。

Jest 是 Facebook 出品的一个测试框架，相对其他测试框架，其一大特点就是就是内置了常用的测试工具，比如自带断言、测试覆盖率工具，实现了开箱即用。

此外， Jest 的测试用例是并行执行的，而且只执行发生改变的文件所对应的测试，提升了测试速度。

**四个基础单词**

编写单元测试的语法通常非常简单；对于jest来说，由于其内部使用了 Jasmine 2 来进行测试，故其用例语法与 Jasmine 相同。

实际上，只要先记这住四个单词，就足以应付大多数测试情况了：

* describe： 定义一个测试套件
* it：定义一个测试用例
* expect：断言的判断条件
* toEqual：断言的比较结果

describe('test ...', **function**() {

it('should ...', **function**() {

expect(sth).toEqual(sth);

expect(sth.length).toEqual(1);

expect(sth > oth).toEqual(true);

});

});

复制代码

**配置**

Jest 号称自己是一个 “Zero configuration testing platform”，只需在 npm scripts里面配置了test: jest，即可运行npm test，自动识别并测试符合其规则的（一般是 \_\_test\_\_ 目录下的）用例文件。

实际使用中，适当的自定义配置一下，会得到更适合我们的测试场景：

//jest.config.js

module.exports = {

modulePaths: [

"<rootDir>/src/"

],

moduleNameMapper: {

"\.(css|less)$": '<rootDir>/\_\_test\_\_/NullModule.js'

},

collectCoverage: true,

coverageDirectory: "<rootDir>/src/",

coveragePathIgnorePatterns: [

"<rootDir>/\_\_test\_\_/"

],

coverageReporters: ["text"],

};

复制代码

在这个简单的配置文件中，我们指定了测试的“根目录”，配置了覆盖率（内置的istanbul）的一些格式，并将原本在webpack中对样式文件的引用指向了一个空模块，从而跳过了这一对测试无伤大雅的环节

//NullModule.js

module.exports = {};

复制代码

另外值得一提的是，由于jest.config.js是一个会在npm脚本中被调用的普通 JS 文件，而非XXX.json或.XXXrc的形式，所以 nodejs 的各自操作都可以进行，比如引入 fs 进行预处理读写等，灵活性非常高，可以很好的兼容各种项目

**babel-jest**

由于是面向src目录下测试其React代码，并且还使用了ES6语法，所以项目下需要存在一个.babelrc文件：

{

"presets": ["env", "react"]

}

复制代码

以上是基本的配置，而实际由于webpack可以编译es6的模块，一般将babel中设为{ "modules": false }，此时的配置为：

//package.json

"scripts": {

"test": "cross-env NODE\_ENV=test jest",

},

复制代码

//.babelrc

{

"presets": [

["es2015", {"modules": false}],

"stage-1",

"react"

],

"plugins": [

"transform-decorators-legacy",

"react-hot-loader/babel"

],

"env": {

"test": {

"presets": [

"es2015", "stage-1", "react"

],

"plugins": [

"transform-decorators-legacy",

"react-hot-loader/babel"

]

}

}

}

复制代码

**Enzyme**

Enzyme 来自于活跃在 JavaScript 开源社区的 Airbnb 公司，是对官方测试工具库（react-addons-test-utils）的封装。

这个单词的伦敦读音为 ['enzaɪm]，酵素或酶的意思，Airbnb 并没有给它设计一个图标，估计就是想取用它来分解 React 组件的意思吧。

它模拟了 jQuery 的 API，非常直观并且易于使用和学习，提供了一些与众不同的接口和几个方法来减少测试的样板代码，方便判断、操纵和遍历 React Components 的输出，并且减少了测试代码和实现代码之间的耦合。

一般使用 Enzyme 中的 mount 或 shallow 方法，将目标组件转化为一个 ReactWrapper对象，并在测试中调用其各种方法：

import Enzyme,{ mount } from 'enzyme';

...

describe('test ...', **function**() {

it('should ...', **function**() {

wrapper = mount(

<MyComp isDisabled={true} />

);

expect( wrapper.find('input').exists() ).toBeTruthy();

});

});

复制代码

**sinon**

图中这位“我牵着马”的并不是卷帘大将沙悟净...其实图中的故事正是人所皆知的“特洛伊木马”；大概意思就是希腊人围困了特洛伊人十多年，久攻不下，心生一计，把营盘都撤了，只留下一个巨大的木马（里面装着士兵），以及这位被扒光还被打得够呛的人，也就是此处要谈的主角sinon，由他欺骗特洛伊人 --- 后面的剧情大家就都熟悉了。

所以这个命名的测试工具呢，也正是各种伪装渗透方法的合集，为单元测试提供了独立而丰富的 spy, stub 和 mock 方法，兼容各种测试框架。

虽然 Jest 本身也有一些实现 spy 等的手段，但 sinon 使用起来更加方便。

**III. 用测试驱动 React 组件重构**

这里不展开讨论经典的 “测试驱动开发”(TDD - test driven development) 理论 -- 简单的说，把测试正向加诸开发，先写用例再逐步实现，就是TDD，这是很好理解的。

而当我们反过头来，对既有代码补充测试用例，使其测试覆盖率不断提高，并在此过程中改善原有设计，修复潜在问题，同时又保证原有接口不收影响，这种 TDD 行为虽然没人称之为“测试驱动重构”(test driven refactoring)，但“重构”这个概念本身就包含了用测试保驾护航的意思，是必不可少的题中之意。

对于一些组件和共有函数等，完善的测试也是一种最好的使用说明书。

**失败-编码-通过 三部曲**

由于测试结果中，成功的用例会用绿色表示，而失败的部分会显示为红色，所以单元测试也常常被称为 “Red/Green Testing” 或 “Red/Green Refactoring” , 这也是 TDD 中的一般性步骤：

1. 添加一个测试
2. 运行所有测试，看看新加的这个是不是失败了；如果能成功则重复步骤1
3. 根据失败报错，有针对性的编写或改写代码；这一步的唯一目的就是通过测试，先不必纠结细节
4. 再次运行测试；如果能成功则跳到步骤5，否则重复步骤3
5. 重构已经通过测试的代码，使其更可读、更易维护，且不影响通过测试
6. 重复步骤1

**解读测试覆盖率**

这就是 jest 内置的 istanbul 输出的覆盖率结果。

之所以叫做“伊斯坦布尔”，是因为土耳其地毯世界闻名，而地毯是用来"覆盖"的🤦‍♀️。

表格中的第2列至第5列，分别对应四个衡量维度：

* 语句覆盖率（statement coverage）：是否每个语句都执行了
* 分支覆盖率（branch coverage）：是否每个if代码块都执行了
* 函数覆盖率（function coverage）：是否每个函数都调用了
* 行覆盖率（line coverage）：是否每一行都执行了

测试结果根据覆盖率被分为“绿色、黄色、红色”三种，应该视具体情况尽量提高相应模块的测试覆盖率。

**优化依赖 让 React 组件变得 testable**

合理编写组件化的 React，并将足够独立、功能专一的组件作为测试的单元，将使得单元测试变得容易；

反之，测试的过程让我们更易厘清关系，将原本的组件重构或分解成更合理的结构。分离出的子组件往往也更容易写成stateless的无状态组件，使得性能和关注点更加优化。

**明确指定 PropTypes**

对于一些之前定义并不清晰的组件，可以统一引入 prop-types，明确组件可接收的props；一方面可以在开发/编译过程中随时发现错误，另外也可以在团队中其他成员引用组件时形成一个明晰的列表。

**IV. React 单元测试常见案例**

**用例的预处理或后处理**

可以用beforeEach和afterEach做一些统一的预置和善后工作，在每个用例的之前和之后都会自动调用：

describe('test components/Comp', **function**() {

**let** wrapper;

**let** spy;

beforeEach(**function**() {

jest.useFakeTimers();

spy = sinon.spy(Comp.prototype, 'componentDidMount');

});

afterEach(**function**() {

jest.useRealTimers();

wrapper && wrapper.unmount();

didMountSpy.restore();

didMountSpy = null;

});

it('应该正确显示基本结构', **function**() {

wrapper = mount(

<Comp ... />

);

expect(wrapper.find('a').text()).toEqual('HELLO!');

});

...

});

复制代码

**调用组件的“私有”方法**

对于一些组件中，如果希望在测试阶段调用到其一些内部方法，又不想对原组件改动过大的，可以用instance()取得组件类实例：

it('应该正确获取组件类实例', **function**() {

var wrapper = mount(

<MultiSelect

name="HELLOKITTY"

placeholder="select sth..." />

);

var wi = wrapper.instance();

expect( wi.props.name ).toEqual( "HELLOKITTY" );

expect( wi.state.open ).toEqual( false );

});

复制代码

**异步操作的测试**

作为UI组件，React组件中一些操作需要延时进行，诸如onscroll或oninput这类高频触发动作，需要做函数防抖或节流，比如常用的 lodash 的 debounce 等。

所谓的异步操作，在不考虑和 ajax 整合的集成测试的情况下，一般都是指此类操作，只用 setTimeout 是不行的，需要搭配 done 函数使用：

//组件中

**const** Comp = (props)=>(

<input type="text" id="searchIpt" onChange={ debounce(props.onSearch, 500) } />

);

复制代码

//单元测试中

it('应该在输入时触发回调', **function**(done) {

**var** spy = jest.fn();

**var** wrapper = mount(

<Comp onChange={ spy } />

);

wrapper.find('#searchIpt').simulate('change');

setTimeout(()=>{

expect( spy ).toHaveBeenCalledTimes( 1 );

done();

}, 550);

});

复制代码

**一些全局和单例的模拟**

一些模块中可能耦合了对 window.xxx 这类全局对象的引用，而完全去实例化这个对象可能又牵扯出很多其他的问题，难以进行；此时可以见招拆招，只模拟一个最小化的全局对象，保证测试的进行：

//fakeAppFacade.js

**var** facade = {

router: {

current: **function**() {

**return** {name:null, params:null};

}

}, appData: {

symbol: "&yen;"

}

};

window.\_appFacade = facade;

module.exports = facade;

复制代码

//测试套件中

**import** fakeFak **from** '../fakeAppFacade';

复制代码

另外比如 LocalStroage 这类对象，测试端环境中没有原生支持，也可以简单模拟一下：

//fakeStorage.js

var \_util = {};

var fakeStorage = {

"set": **function**(k, v) {

\_util['\_fakeSave\_'+k] = v;

},

"get": **function**(k) {

return \_util['\_fakeSave\_'+k] || null;

},

"remove": **function**(k) {

delete \_util['\_fakeSave\_'+k];

},

"has": **function**(k) {

return \_util.hasOwnProperty('\_fakeSave\_'+k);

}

};

module.exports = fakeStorage;

复制代码

**棘手的 react-bootstrap/modal**

在一个项目中用到了 react-bootstrap 界面库，测试一个组件时，由于包含了其 Modal 模态弹窗，而弹窗组件是默认渲染到 document 中的，导致难以用普通的 find 方法等获取

解决的办法是模拟一个渲染到容器组件原处的普通组件：

//FakeReactBootstrapModal.js

import React, {Component} from 'react';

class FakeReactBootstrapModal extends Component {

constructor(props) {

super(props);

}

**render**() { //原生的 react-bootstrap/Modal 无法被 enzyme 测试

const {

show,

bgSize,

dialogClassName,

children

} = this.props;

return show

? <div className={

`fakeModal ${bgSize} ${dialogClassName}`

}>{children}</div>

: null;

}

}

export default FakeReactBootstrapModal;

复制代码

同时在组件渲染时，加入判断逻辑，使之可以支持自定义的类代替 Modal 类：

//ModalComp.js

**import** { Modal } **from** 'react-bootstrap';

...

render() {

**const** MyModal = **this**.\_modalClass || Modal;

**return** (<MyModal

bsSize={props.mode>1 ? "large" : "middle"} dialogClassName="custom-modal">

...

</MyModal>;

}

复制代码

而测试套件中，实现一个测试专用的子类：

//myModal.spec.js

**import** ModalComp **from** 'components/ModalComp';

**class** **TestModalComp** **extends** **ModalComp** {

**constructor**(props) {

**super**(props);

**this**.\_modalClass = FakeReactBootstrapModal;

}

}

复制代码

这样测试即可顺利进行，跳过了并不重要的 UI 效果，而各种逻辑都能被覆盖了

**模拟fetch请求**

在单元测试的过程中，难免碰到一些需要远程请求数据的情况，比如组件获取初始化数据、提交变化数据等。

要注意这种测试的目的还是考察组件本身的表现，而非重点关心实际远程数据的集成测试，所以我们无需真实的请求，可以简单的模拟一些请求的场景。

sinon 中有一些模拟 XMLHttpRequest 请求的方法, jest 也有一些第三方的库解决 fetch 的测试;

在我们的项目中，根据实际的用法，自己实现一个类来模拟请求的响应：

//FakeFetch.js

**import** { noop } **from** 'lodash';

**const** fakeFetch = (jsonResult, isSuccess=true, callback=noop)=>{

**const** blob = **new** Blob(

[JSON.stringify(jsonResult)],

{type : 'application/json'}

);

**return** (...args)=>{

console.log('FAKE FETCH', args);

callback.call(null, args);

**return** isSuccess

? Promise.resolve(

**new** Response(

blob,

{status:200, statusText:"OK"}

)

)

: Promise.reject(

**new** Response(

blob,

{status:400, statusText:"Bad Request"}

)

)

}

};

**export** **default** fakeFetch;

复制代码

//Comp.spec.js

import fakeFetch from '../FakeFetch';

const \_fc = window.fetch; //缓存“真实的”fetch

describe('test components/Comp', **function**() {

let wrapper;

afterEach(**function**() {

wrapper && wrapper.unmount();

window.fetch = \_fc; //恢复

});

it("应该在远程请求时响应onRemoteData", (**done**)=>{

window.fetch = fakeFetch({

brand: "GoBelieve",

tree: {

node: '总部',

children: null

}

});

let spy = jest.fn();

wrapper = mount(

<Comp onRemoteData={ spy } />

);

jest.useRealTimers();

\_clickTrigger(); //此时应该发起请求

setTimeout(()=>{

expect(wrapper.html()).toMatch(/总部/);

expect(spy).toHaveBeenCalledTimes(1);

**done**();

}, 500);

});

});

复制代码

**V. 总结**

单元测试作为一种经典的开发和重构手段，在软件开发领域被广泛认可和采用；前端领域也逐渐积累起了丰富的测试框架和方法。

单元测试可以为我们的开发和维护提供基础保障，使我们在思路清晰、心中有底的情况下完成对代码的搭建和重构；

需要注意的是，世上没有包治百病的良药，单元测试也绝不是万金油，秉持谨慎认真负责的态度才能从根本上保证我们工作的进行。

**扩展阅读：**

一个重构的实例 [mp.weixin.qq.com/s?\_\_biz=MzI…](https://link.juejin.im/?target=https%3A%2F%2Fmp.weixin.qq.com%2Fs%3F__biz%3DMzI0MDYzOTEyOA%3D%3D%26mid%3D2247483989%26idx%3D1%26sn%3De7f5187746c2f3b3700beb6ae98170a7%26chksm%3De9168fa2de6106b4308a2ebb613474cd8f0530e20532372c38c540864e6361bb825950cccb39%26scene%3D0%23rd)

**VI. 参考资料**

* https://juejin.im/post/59b5e79f6fb9a00a600f4216
* https://baike.baidu.com/item/单元测试
* http://www.cnblogs.com/jianglingli83/archive/2013/03/15/2961847.html
* http://blog.csdn.net/hustzw07/article/details/74178051
* https://zhuanlan.zhihu.com/p/28247899
* https://en.wikipedia.org/wiki/Test-driven\_development
* http://www.ruanyifeng.com/blog/2015/06/istanbul.html
* https://www.eurobricks.com/forum/index.php?/forums/topic/52658-lego-construction-site/