学习 单元测试

Enzyme官方文档翻译

Enzyme(酶)是一个Javascript测试库，提供了类似Jquery的简洁灵活的API来操作和遍历DOM。Enzyme本身不提供测试框架或者断言库功能，只提供了渲染、操作、遍历的功能，可以简单的接入到任意的测试框架以及使用任意的断言库。

Enzyme 2.x和3.x的重要变动

Adapter

原本使用2.X不需要安装额外的库，enzyme升级到3.x后多了一个“Adapter”系统，因此需要额外安装“Adapter”来告诉enzyme当前运行的React版本，官方提供了React 0.13.x, 0.14.x, 15.x, and 16.x的“Adapter”。
目前在项目中使用的是React 0.14版本，因此在enzyme 3.x下使用enzyme需要

1. 命令行中安装enzyme和react 0.14的adapter

```
1. yarn add -D enzyme enzyme-adapter-react-14
```

2. setup
   在使用enzyme之前使用顶层API `configure`配置适配器，这里也可以设置其他特性，例如禁用lifecycle

不再保留元素引用

官方demo：

```
import React from 'react';
import Icon from './path/to/Icon';

const ICONS = {
  success: <Icon name="check-mark" />,
  failure: <Icon name="exclamation-mark" />,
};

const StatusLabel = ({ id, label }) => <div>{ICONS[id]}
{label}{ICONS[id]}</div>;
```

```
import { shallow } from 'enzyme';
import StatusLabel from './path/to/StatusLabel';
import Icon from './path/to/Icon';

const wrapper = shallow(<StatusLabel id="success"
label="Success" />);

const iconCount = wrapper.find(Icon).length;
```

在v2.x中iconCount为1，在v3.x中为2。原因是v2.x中找到所有匹配selector的元素后将重复的移除了，因此在v2.x中返回了一个元素。在v3.x中元素会被转换为底层的react elements，因此具有不同的引用，所以就有2个元素返回。

`children()`方法的变动

官方demo：

```
class Box extends React.Component {
  render() {
    return <div className="box">{this.props.children}</div>;
  }
}
class Foo extends React.Component {
  render() {
    return (
      <Box bam>
        <div className="div" />
      </Box>
    );
  }
}
```

在使用`mount()`的时候

```
const wrapper = mount(<Foo />);
```

`wrapper.find(Box).children()`在v2.x和v3.x下的区别：

- v2.x下

```
wrapper.find(Box).children().debug();
// => <div className="div" />
```

- v3.x下：

```
wrapper.find(Box).children().debug();
// =>
// <div className="box">
//   <div className="div" />
// </div>
```

两者的主要区别是：v2.x只会返回Props.children的结果，类似于浅渲染，v3.x会如期望那样返回真实渲染的结果。

通过组件实例直接调用组件内部方法改变State需要`update()`

官方demo：

```
1. class Counter extends React.Component {
2.   constructor(props) {
3.     super(props);
4.     this.state = { count: 0 };
5.     this.increment = this.increment.bind(this);
6.     this.decrement = this.decrement.bind(this);
7.   }
8.   increment() {
9.     this.setState({ count: this.state.count + 1 });
10.  }
11.  decrement() {
12.    this.setState({ count: this.state.count - 1 });
13.  }
14.  render() {
15.    return (
16.      <div>
17.        <div className="count">Count: {this.state.count}
    </div>
18.        <button className="inc" onClick={this.increment}
    >Increment</button>
19.        <button className="dec" onClick={this.decrement}
    >Decrement</button>
20.      </div>
21.    );
22.  }
23. }
```

```
const wrapper = shallow(<Counter />);
// 通过事件模拟下v2.x和v3.x表现一致都会自动更新状态
wrapper.find('.count').text(); // => "Count: 0"
wrapper.find('.inc').simulate('click');
wrapper.find('.count').text(); // => "Count: 1"
wrapper.find('.inc').simulate('click');
wrapper.find('.count').text(); // => "Count: 2"
wrapper.find('.dec').simulate('click');
wrapper.find('.count').text(); // => "Count: 1"

// 调用实例上的方法 v3.x不会自动更新
wrapper.find('.count').text(); // => "Count: 0"
wrapper.instance().increment();
wrapper.find('.count').text(); // => "Count: 0" (would have been "Count: 1" in v2)
wrapper.instance().increment();
wrapper.find('.count').text(); // => "Count: 0" (would have been "Count: 2" in v2)
wrapper.instance().decrement();
wrapper.find('.count').text(); // => "Count: 0" (would have been "Count: 1" in v2)


// 在v3.x下需要手动更新
wrapper.find('.count').text(); // => "Count: 0"
wrapper.instance().increment();
wrapper.update();
wrapper.find('.count').text(); // => "Count: 1"
wrapper.instance().increment();
wrapper.update();
wrapper.find('.count').text(); // => "Count: 2"
wrapper.instance().decrement();
wrapper.update();
wrapper.find('.count').text(); // => "Count: 1"
```

`ref()`返回真实的ref而不是wrapper

在v2.x下ref返回的是wrapper实例，在v3.x下和真实的react一致，当ref定义在dom元素上时返回的是Dom Element，在React组件上时返回的是组件的实例。
在v2.x下：

```
const wrapper = mount(<Box />);
// this is what would happen with enzyme v2
expect(wrapper.ref('abc')).toBeInstanceOf(wrapper.constructor);
```

在v3.x下:

- 直接定义在dom元素上的ref

```
const wrapper = mount(<Box />);
// this is what happens with enzyme v3
expect(wrapper.ref('abc')).toBeInstanceOf(Element);
```

- 定义在React元素上的ref

```
class Bar extends React.Component {
  render() {
    return <Box ref="abc" />;
  }
}
```

```
const wrapper = mount(<Bar />);
expect(wrapper.ref('abc')).toBeInstanceOf(Box);
```

在`mount`下，`.instance()`可以级联在任何返回的wrapper对象上调用

在v3.x下可以在任何返回wrapper对象的方法后调用`instance()`，将会返回组件的实例，因此可以在实例后再调用`.setState()`，更加灵活。

在`mount`下，`.getNode()`被废弃，使用`.instance()`代替

在`shallow`下，`.getNode()`被废弃，使用`getElement()`代替

移除了私有属性和方法

- `.node`

- `.nodes`
- `.renderer`
- `.unrendered`
- `.root`
- `.options`

支持真实的CSS选择器

v2.x中的CSS选择器使用的是enzyme自己的不完整的CSS解析，在v3.x中支持了真实的CSS选择器。

节点Equality时忽略undefined值

即在v3.x中下面两个节点被认为是equal，在v2.x中被认为是不等的两个节点

```
1. <div />
2. <div className={undefined} id={undefined} />
```

默认开启生命周期方法

在v2.x中生命周期方法默认是禁用的，需要手动开启。在v3.x中生命周期方法默认开启。
在v3.x中可以通过一下两种方式关闭：

1. 通过顶层API `configure()` 配置默认关闭：

```
1. import Enzyme from 'enzyme';
2. Enzyme.configure({ disableLifecycleMethods: true });
```

2. 在渲染的时候通过flag关闭

```
1. import { shallow } from 'enzyme';
2. // ...
3. const wrapper = shallow(<Component />, { disableLifecycl
   eMethods: true });
```

安装

对应版本具体安装参考
官方指南：http://airbnb.io/enzyme/docs/installation/
目前我们的项目使用react@0.14.8 react-dom@0.14.8
首先安装test utilities addon

```
1. yarn add -D react-addons-test-utils@0.14
```

然后安装相应react版本的adapter

```
1. yarn add -D enzyme enzyme-adapter-react-14
```

API参考：

> 具体参数和Example参考官方文档：http://airbnb.io/enzyme/docs/api/

浅渲染Shallow Rendering

shallow 方法只会渲染出组件的第一层 DOM 结构，其嵌套的子组件不会被渲染出来，因此渲染效率高，适合用来做单元测试。

`.at(index) => ShallowWrapper`

返回当前wrapper中索引为index(以0开始)的warpper

`.childAt(index) => ShallowWrapper`

返回当前wrapper下索引为index的子节点wrapper

`.children([selector]) => ShallowWrapper`

返回当前wrapper下所有的子节点的wrapper

`.closest(selector) => ShallowWrapper`

通过遍历当前节点的祖先元素获取第一个匹配selector的节点wrapper

`.contains(nodeOrNodes) => Boolean`

如果当前渲染中包含所给节点或节点数组（判断节点时会判断props），返回true，否则返回false。

> 传入的必须是ReactElement或者JSX表达式

`.containsAllMatchingElements(nodes) => Boolean`

如果当前节点下**相似性**包含全部提供的`nodes`的子节点（如果给了props则判断，没有则忽略），则返回true，否则返回false。

`.containsAnyMatchingElements(nodes) => Boolean`

如果当前节点下**相似性**包含任意一个提供的`nodes`的子节点（如果给了props则判断，没有则忽略），则返回true，否则返回false。

`.containsMatchingElement(node) => Boolean`

表现和上面两种一致，区别是只能传单个节点

`.context([key]) => Any`

返回当前wrapper的节点的context hash，如果提供了key，则只返回值。

`.debug() => String`

为了调试的目的，返回当前wrapper的 HTML-like的字符串表示。

`.dive([options]) => ShallowWrapper`

浅渲染当前wrapper下的一个非DOM子元素，并且返回被wrapper包裹的结果。

> 只能被单个non-DOM（非DOM）子元素调用

`.equals(node) => Boolean`

判断当前wrapper的根节点渲染树是否和传入的节点一致。

> props为undefined会被忽略

`.every(selector) => Boolean`

判断wrapper中的所有节点是否都匹配`selector`。

`.everyWhere(fn) => Boolean`

判断wrapper中的所有节点是否都满足传入的断言函数`fn`。

`.exists() => Boolean`

判断当前节点是否存在

`.filter(selector) => ShallowWrapper`

返回匹配满足`selector`的节点的新的wrapper

`.filterWhere(fn) => ShallowWrapper`

返回满足断言函数`fn`的节点的新的wrapper

`.find(selector) => ShallowWrapper`

返回当前节点中匹配`selector`的所有节点的wrapper

`.findWhere(fn) => ShallowWrapper`

返回当前节点中满足断言函数`fn`的所有节点的wrapper

`.first() => ShallowWrapper`

返回一组节点集合中的第一个节点的wrapper

`.forEach(fn) => Self`

迭代当前wrapper的每一个节点调用提供的`fn`，fn的第一个参数是wrapper包裹的相应节点，第二个参数是索引index。和数组的forEach方法类似，但是返回值为当前wrapepr自身。

`.get(index) => ReactElement`

返回当前wrapper的指定索引为index的ReactElement节点，而不是wrapper。

`.hasClass(className) => Boolean`

判断当前节点是否有指定的`className`。

`.html() => String`

返回当前渲染树的HTML标记字符串。

> 只能被单节点调用

`.instance() => ReactComponent`

返回传入`shallow()`方法作为根节点渲染的组件实例。

> 只能被根节点的wrapper实例调用

`.is(selector) => Boolean`

判断当前节点是否匹配`selector`

`.isEmpty() => Boolean`

判断当前节点是否为空。

> 已弃用，使用`.exists()`代替

`.key() => String`

返回当前wrapper节点的key值。

`.last() => ShallowWrapper`

与`first()`方法对应，返回当前wrapper节点中的最后一个。

`.map(fn) => Array<Any>`

与`forEach()`类似，迭代每一个wrapper，但是返回值是每一个`fn`的返回值。

`.matchesElement(node) => Boolean`

判断当前wrapper节点是否与所给节点**相似**（wrapper节点上包含传入节点的所有props并且值相等，即使元素类型一样也会返回true）

`.name() => String|null`

返回渲染节点的name

- 如果是组件：设置了displayName则返回displayName，否则返回组件name。
- 如果是DOM节点：返回tag name
- null：返回null

`.not(selector) => ShallowWrapper`

筛选出不匹配`selector`的节点的wrapper

`.parent() => ShallowWrapper`

返回当前wrapper节点的直接父元素的wrapper

`.parents([selector]) => ShallowWrapper`

返回当前wrapper节点的所有父元素/祖先元素的wrapper，可以提供`selector`作为筛选。

`.prop(key) => Any`

返回wrapper的根节点的`prop`上属性为`key`的值。在`shallow wrapper`上调用时返回的是渲染的组件的根节点的props，而不是组件自身的props，如果要返回组件自身的props需要调用`wrapper.instance().props()`

`.reduce(fn[, initialValue]) => Any`

类似于数组的reduce

`.reduceRight(fn[, initialValue]) => Any`

reduce顺序变为从右到左

`.render() => CheerioWrapper`

返回当前节点子树的HTML渲染字符串的Cheerio对象

`.setContext(context) => Self`

设置根节点的`context`，并且重新渲染

> 只能被根节点的wrapper实例调用

`.setProps(nextProps) => Self`

设置根节点的`props`，并且调用`componentWillReceiveProps`生命周期方法

> 只能被根节点的wrapper实例调用

`.setState(nextState[, callback]) => Self`

在根节点实例上调用`setState`,并且触发重新渲染。

`.shallow([options]) => ShallowWrapper`

返回浅渲染后的当前节点wrapper

`.simulate(event[, ...args]) => Self`

模拟事件

> 在shalow renderer上不会事件冒泡，应该在真实的节点上模拟事件

`.slice([begin[, end]]) => ShallowWrapper`

类似数组的分片，返回分片后的wrapper

`.some(selector) => Boolean`

判断wrappers中是否有至少一个匹配`selector`

`.someWhere(fn) => Boolean`

判断wrappers中是否有至少一个满足断言函数

`.state([key]) => Any`

返回wrapper的根节点的state的hash，如果传入key则返回对应的值

`.tap(intercepter) => Self`

调用一个拦截器并返回自身

> 在调试链式调用的时候很有用

```
const result = shallow((
  <ul>
    <li>xxx</li>
    <li>yyy</li>
    <li>zzz</li>
  </ul>
)).find('li')
  .tap(n => console.log(n.debug()))
  .map(n => n.text());
```

`.text() => String`

返回当前渲染树的渲染文本

> 表现很诡异

```
const wrapper = shallow(<div><b>important</b></div>);
expect(wrapper.text()).to.equal('important');
```

```
const wrapper = shallow(<div><Foo /><b>important</b></div>);
expect(wrapper.text()).to.equal('<Foo />important');
```

`.type() => String|Function|null`

返回wrapper包裹的当前节点的类型

- 如果是组件：返回组件的constructor
- 如果是原生的DOM节点，返回标签名
- 如果是null：返回null

`.unmount() => Self`

卸载组件，用来模拟组件生命周期

`.update() => Self`

强制重新渲染，当外部调用改变组件的State时，使用`update()`进行re-render

```
class ImpureRender extends React.Component {
  constructor(props) {
    super(props);
    this.count = 0;
  }
  render() {
    this.count += 1;
    return <div>{this.count}</div>;
  }
}
```

```
const wrapper = shallow(<ImpureRender />);
expect(wrapper.text()).to.equal('0');
wrapper.update();
expect(wrapper.text()).to.equal('1');
```

全渲染Full Rendering

将React组件渲染为真实的DOM节点，因此可以使用DOM事件。`mount()`需要环境提供完整的DOM API,因此需要使用一个“Browser Like”的环境，例如JSDOM或者Phantoms，或者直接在真实浏览器下运行。

```
import { mount } from 'enzyme';
const wrapper = mount(<MyComponent />);
```

> 提供的API与Shallow Rendering几乎一致

静态渲染Static Rendering

将React组件渲染为静态的HTML字符串，然后返回一个Cheerio实例对象，使用Cheerio来分析HTML的结构。（Cheerio常在爬虫的时候用来分析爬到的HTML结构）

> 返回Cherrio对象，因此可以使用Cherrio的API

```
import { render } from 'enzyme';
const wrapper = render(<MyComponent />);
```

Selectors

1.任何合法的CSS选择器

2.Prop选择器

```
const wrapper = mount((
  <div>
    <span foo={3} bar={false} title="baz" />
  </div>
));

wrapper.find('[foo=3]');
wrapper.find('[bar=false]');
wrapper.find('[title="baz"]');
```

> 不能使用key 和 ref作为选择器使用

3.React Component Constructor

```
function MyComponent() {
  return <div />;
}

// find instances of MyComponent
const myComponents = wrapper.find(MyComponent);
```

4.React Component's displayName

5.对象属性选择器

```
const wrapper = mount((
  <div>
    <span foo={3} bar={false} title="baz" />
  </div>
));

wrapper.find({ foo: 3 });
wrapper.find({ bar: false });
wrapper.find({ title: 'baz' });
```

> 对象属性选择器中不能使用属性值为undefined的属性，会报错，可以使用`findWhere()`方法代替