React 单元测试

这一小节来讲 React 中的测试，虽然不涉及 React 的使用，但是却关系到我们开发的 React 的质量。

毫无疑问，对代码进行测试是最佳实践，可以保证代码质量，不过，对于软件代码质量毫不关心的读者可以略过这一节。

测试的目的

测试对于软件开发非常重要，简单来说，测试就是尽力发现软件中的缺陷（俗称 bug），当我们发现不了更多的 bug 时，说明这个软件质量可以接受了。

然而，没有 bug 的软件我还没见过呢。

在互联网时代，我们更是不可能等到所有 bug 都修复了才上线，那样黄花菜都凉了，稍微有一些工作经验的人都会有这样的体会。

所以，事实上，测试是**尽力**发现软件中的 bug。当我们发现 bug 数量和严重程度呈稳定的下降趋势，直到低于一个门槛（无须降低为 0，只需要降低到可接受的程度），没有更多更严重的 bug 出现，就说明这个软件的质量可以接受，可以上线了。

这样当然要比达到“零 bug 软件”要容易多了，但是，不要因此以为这就是一件没有困难的任务。为了让 bug 的数量和严重程度足够低，我们开发者必须严格要求自己，只有保证我们写的每一小块代码都经受住测试的考验，这些小块代码集合在一起的时候才可能（只是有可能）不会出很多 bug，如果我们写的小块代码质量都无法保证，那大项目的代码根本无法保证。

这一小节重点讲对”小块代码“的测试，也就是单元测试。

单元测试

单元测试的内容足够讲一本书了，所以，我只针对性地讲一讲 React 组件的单元测试，并且只有三个要点：

1. 用 Jest；
2. 用 Enzyme；
3. 保持 100% 的代码覆盖率。

Jest

在 JavaScript 的世界里，单元测试的框架很多，品牌最老名气最响的是 Mocha，不过，不要纠结于名气，请使用 Jest。你不会后悔的，接下来我告诉你为什么。

我们先假设，作为开发者，你是在团队中工作。所谓团队，就是有很多人一起工作，而且随着业务和团队的发展，人会越来越多，潜台词就是——不确定因素越来越多。

人和人之间交流会出现偏差，人的水平有高低之分，人也会犯错，总之，你不能指望所有人都把事情做得尽善尽美。

具体到单元测试这件事上来，“测试驱动”是开发喊了这么多年，为什么真正做到这一点的团队依然不多呢？因为，当团队变大之后，很多问题也就出现了。

1. **单元测试用例庞大，执行时间过长。**

想象一下，一个代码库里假设有一千个单元测试用例，即使每个单元测试用例平均只需要 10 毫秒，那总时间也就需要 10 秒钟。好，假设代码库进一步扩大，有了一万个单元测试用例，那就跑一遍就需要 100 秒，已经超过了一分钟。这还只是保守估计，实际上单元测试用例的运行时间只会比这长。开发者如果每次修改都需要等待这么漫长的单元测试运行时间，肯定会三心二意上网去看其他东西。

2. **单元测试用例之间相互影响。**

你可能也有这样的体验，代码库中的单元测试突然失败了，但是你修改的代码根本不会取影响失败的那个单元测试用例，怎么回事？这往往是因为某个成员以前的代码写得不好，影响了一个全局变量。当然，谁都知道单元测试应该在 setup 时创建环境，在 teardown 时恢复环境，可是，总会有人有马虎大意的情况，这时候你怎么办？要么你只好去修复一个本不是你改坏的代码，要么你干脆删掉那段不可靠的单元测试代码，不管怎样，这都会打击你支持“测试驱动开发”的决心。

Jest 较好地解决了上面说的问题，因为 Jest 最重要的一个特性，就是支持并行执行。

Mocha 之类老牌单元测试框架，把所有的单元测试都放在一个环境中执行，这就使所有单元测试访问的是同样一个全局变量空间，所以只要测试代码没写好，就会互相影响。而且，为了保证执行正常，所有的单元测试必须一个接一个地执行，这是体系架构决定的，没有办法。

Jest 不同，Jest 为每一个单元测试文件创造一个独立的运行环境，换句话说，Jest 会启动一个进程执行一个单元测试文件，运行结束之后，就把这个执行进程废弃了，这个单元测试文件即使写得比较差，把全局变量污染得一团糟，也不会影响其他单元测试文件，因为其他单元测试文件是用另一个进程来执行。

更妙的是，因为每个单元测试文件之间再无纠葛，Jest 可以启动多个进程同时运行不同的文件，这样就充分利用了电脑的多 CPU 多核，单进程 100 秒才完成的测试执行过程，8 核只需要 12.5 秒，速度快了很多。

Jest 还有很多其他友好的特性，大家可以自己去发掘，这里废话不多说，只想安利各位，**测试 React 或者 JavaScript 代码，用 Jest!**

使用 create-react-app 产生的项目自带 Jest 作为测试框架，不奇怪，因为 Jest 和 React 一样都是出自 Facebook。

运行下面的命令，就可以进入交互式的”测试驱动开发“模式：

```
npm test
```

Enzyme

虽然最好的 React 测试框架出自 Facebook 家，最受欢迎的 React 测试工具库却出自 Airbnb，这个工具库叫做 Enzyme。Enzyme 这个单词的含义是“酶”，至于命名原因已经无法考据，可能寓意着快速分解。

不过因为 Enzyme 不是 Facebook 家出品，所以使用 Enzyme 还真稍微有些麻烦——在 create-react-app 产生的应用中并不包含 Enzyme，需要我们自己来添加。

在项目目录下，通过下面的命令来安装 `enzyme`：

```
npm i --save-dev enzyme enzyme-adapter-react-16
```

可以注意到，我们不光要安装 enzyme，还要安装 `enzyme-adapter-react-16`，这个库是用来作为适配器的。因为不同 React 版本有各自特点，所用的适配器也会不同，我们的项目中使用的是 16.4 之后的版本，所以用 `enzyme-adapter-react-16`；如果用 16.3 版本，需要用 `enzyme-adapter-react-16.3`；如果用 16.2 版本，需要用 `enzyme-adapter-react-16.2`；如果用更老的版本 15.5，需要用 `enzyme-adapter-react-15`。具体各个 React 版本对应什么样的 Adapter，请参考 enzyme官方文档。

现在，可以在测试代码中使用 enzyme 了。我们以之前秒表应用中的 ControlButtons 组件为例，来说明如何做单元测试。

我们创造一个 `ControlButtons.test.js`，来容纳对应的测试用例，因为所有后缀为 `.test.js` 的文件都会被 Jest 认作是测试用例文件。

在代码中，需要使用 Adapter，代码如下：

```
import {configure} from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
configure({adapter: new Adapter()});
```

我们对 ControlButtons 组件的测试，就是要渲染它一次，看看渲染结果如何，enzyme 就能帮助我们做这件事。

比如，我们想要保证渲染出来的内容必须包含两个按钮，其中一个按钮的 class 名是 `left-btn`，另一个是 `right-btn`，那么我们就需要下面的单元测试用例：

```
import {shallow} from 'enzyme';

it('renders without crashing', () => {
  const wrapper = shallow(<ControlButtons />);
  expect(wrapper.find('.left-btn')).toHaveLength(1);
  expect(wrapper.find('.right-btn')).toHaveLength(1);
});
```

在这里我们使用了 `shallow`，其实也可以使用 `mount`。

shallow 和 mount 的区别，就是 shallow 只会渲染被测试的 React 组件这一层，不会渲染子组件；而 mount 则是完整地渲染 React 组件包括其所有子组件，包括触发 `componentDidMount` 生命周期函数。

原则上，能用 shallow 就尽量用 shallow，首先是为了测试性能考虑，其次是可以减少组件之间的影响，比如，一个组件 `Foo` 有子组件 `Bar`，如下：

```
const Foo = () => ()
    <div>
       {/* other logic */}
       <Bar />
    </div>
)
```

如果用 mount 去渲染 Foo，会连带 Bar 一起完全渲染，如果 Bar 出了什么毛病，那 Foo 的单元测试也过不了；如果用 shallow，只知道 Bar 曾经被用，即使 Bar 哪里出了问题，也不影响 Foo 的单元测试。

这并不是说我们就不管 Bar，Bar 的质量会由它自己的单元测试来检验，这就引出下一个话题——代码覆盖率。

代码覆盖率

你不能给自己的程序随便写几个单元测试，就说自己的代码已经测试好了，就像上面我只给 ControlButtons 组件写了一个测试用例，我并不能说整个秒表应用已经通过了测试。

你的代码测试覆盖率只有达到一定程度，才好说自己的代码已经被测试了。

剩下来就是一个纠结的问题：代码测试的覆盖率应该达到多少才算够？

以我个人的经验，代码覆盖率必须达到 100%，也就是说，一个应用不光所有的单元测试都要通过，而且所有单元测试都必须覆盖到代码 100% 的角落。

如果对覆盖率的要求低于 100%，时间一长，质量必定会越来越下滑。

遇到一个不好测试的代码，开发者倾向于不去考虑如何重构代码提高可测试性，而是直接忽略这部分代码不去测试，反正不要求 100% 嘛；遇到工期比较紧的时候，甚至会进一步降低代码覆盖率要求，用牺牲质量来加快开发速度，反正不要求 100% 嘛。

所以，如果你真的对代码质量认真负责的话，请坚守 100% 代码覆盖率的底线！

在 create-react-app 创造的应用中，已经自带了代码覆盖率的支持，运行下面的命令，不光会运行所有单元测试，也会得到覆盖率汇报。

```
npm test -- --coverage
```

代码覆盖率包含四个方面：

1. 语句覆盖率
2. 逻辑分支覆盖率
3. 函数覆盖率
4. 代码行覆盖率

只有四个方面都是 100%，才算真的 100%。

小结

这一小节相对较为简略，这是有原因的。虽然我可以讲解很详细的单元测试工具使用方法，但是这并不是最重要的，最重要也最困难的是具有质量意识。

如果你具备质量意识，只需要强调上面说的 3 点，你就知道该怎么做，不需要我多说你也会找到对应的使用方法；相反，如果你不具备质量意识，说遍所有测试技巧，你也不会去实践，多说无益。

就像我在本节开始所说的，如果你根本不在意软件质量，完全可以忽略掉这一小节。

留言

评论将在后台进行审核，审核通过后对所有人可见

Artyhacker 前端 @ 北京某小国企
很不好意思地说，写了一年多的react项目了，从来没有写过单元测试。。。今天开始研究！
▲ 0 评论 20天前

孤落无痕 前端开发
理想很美好，现实很骨感
▲ 0 评论 1月前

snowLu 前端小洋葱 @ lg
是呀，墨哥，写一本测试代码的小册吧
▲ 0 收起评论 1月前

程墨 Hulu
我只是很好奇，在我国企业中真的有很多人写单元测试代码吗？
1月前

请叫我王磊同学 前端工程师 @ 帆软软件
回复 程墨：我们写，主要还是正对数据流进行测试，作用一定程度还是有的，但是要是真的说作用有多明显也说不上，e2e能起到作用感觉更小。
1月前

zhangyanling77 前端开发 @ 成都
回复 程墨：很少很少会写单元测试
1月前

HaoliangWu 前端/全栈工程师 @ 云匠软件
回复 程墨：一般小公司的话，只能写写 utils 或者不写了，因为迭代速度太快，导致写了的单元测试还没怎么发挥作用就变为无效的测试用例了
1月前

评论审核通过后显示 评论

Farris 前端工程师
测试 React 或者 JavaScript 代码，用 Jest!

既然都这么说了，为何不详细地介绍下jest
▲ 3 收起评论 1月前

程墨 Hulu
详细介绍jest需要再写一本jest，说真的，我很怀疑读者中有多大比例对写测试代码感兴趣。
1月前

知集问答 全栈 @ 知集网络
测试代码呢
1月前

劳丁
有很大兴趣
1月前

Catherine酱
对Jest感兴趣就去看官方文档啊，本身jest的内容也不少的
1月前

ImCat
JEST 看官方文档，用过 JQuery 应该会上手很快吧，我没用过 React 刚学就用 Jest 开启测试，发现也很顺手
1月前

1 2 下一页

评论审核通过后显示 评论