React.js 2016 最佳实践

原文：https://blog.risingstack.com/react-js-best-practices-for-2016/

感谢作者：Péter Márton

React中文社区-大量资源群 245192933

近几个月React相关话题依旧火热，相信越来越多的开发者在尝试这样一项技术，我们团队也在PC和移动端不断总结经验。2016来了，这应该是React走向成熟的一年，不管你是新手，还是已经对React有所了解，是时候总结一下最佳实践了，让我们看看国外的开发者总结了哪些好的实践吧~

==========译文分割线==========

2015可以算是React之年了，关于其版本发布和开发者大会的话题遍布全球。关于去年React的发展里程碑详情，可以查看我们整理的React 2015这一年。

2016年最有趣的问题可能是，我们该如何编写一个应用呢，有什么推荐的库或框架？

> 作为一个长时间使用React.js的开发者，我已经有自己的答案和最佳实践了，但你可能不会同意我说的所有点。我对你的想法和意见很感兴趣，请留言进行讨论。

如果你只是刚开始接触React.js，请阅读React.js教程，或Pete Hunt的React howto。

数据处理

在React.js应用中处理数据超级简单的，但同时还是有些挑战。

这是因为你可以使用多种方式，来给一个React组件传递属性数据，从而构建出渲染树。但这种方式并不总是能明显地看出，你是否应该更新某些视图。

2015开始涌现出一批具有更强功能和响应式解决方案的Flux库，让我们一起看看：

Flux

根据我们的经验，Flux通常被过度使用了（就是大家在不需使用的场景下，还是使用了）。

Flux提供了一种清爽的方式存储和管理应用的状态，并在需要的时候触发渲染。

Flux对于那些应用的全局state（译者注：为了对应React中的state概念，本文将不对state进行翻译）特别有用，比如：管理登录用户的状态、路由状态，或是活跃账号状态。如果使用临时变量或者本地数据来处理这些状态，会非常让人头疼。

我们不建议使用Flux来管理路由相关的数据，比如/items/:itemId。应该只是获取它并存在组件的state中，这种情况下，它会在组件销毁时一起被销毁。

如果需要Flux的更多信息，建议阅读The Evolution of Flux Frameworks。

使用Redux

Redux是一个JavaScript app的可预测state容器。

如果你觉得需要Flux或者相似的解决方案，你应该了解一下redux，并学习Dan Abramov的redux入门指南，来强化你的开发技能。

Rudux发展了Flux的思想，同时降低了其复杂度。

扁平化state

API通常会返回嵌套的资源，这让Flux或Redux架构很难处理。我们推荐使用normalizr这类库来尽可能地扁平化state。

像这样：

```
const data = normalize(response, arrayOf(schema.user))
```

```
state = _.merge(state, data.entities)
```

（我们使用isomorphic-fetch与API进行通信）

使用immutable state

共享的可变数据是罪恶的根源——Pete Hunt, React.js Conf 2015

不可变对象是指在创建后不可再被修改的对象。

不可变对象可以减少那些让我们头痛的工作，并且通过引用级的比对检查来提升渲染性能。比如在shouldComponentUpdate中：

```
shouldComponentUpdate(nexProps) {
// 不进行对象的深度对比
return this.props.immutableFoo !== nexProps.immutableFoo
}
```

如何在JavaScript中实现不可变

比较麻烦的方式是，小心地编写下面的例子，总是需要使用deep-freeze-node（在变动前进行冻结，结束后验证结果）进行单元测试。

```
return {
    ...state,
    foo
}
```

```
return arr1.concat(arr2)
```

相信我，这是最明显的例子了。

更简单自然的方式，就是使用Immutable.js。

```
import { fromJS } from 'immutable'

const state = fromJS({ bar: 'biz' })
const newState = foo.set('bar', 'baz')
```

Immutable.js非常快，其背后的思想也非常美妙。就算没准备使用它，还是推荐你去看看Lee Byron的视频Immutable Data and React，可以了解到它内部的实现原理。

Observables and reactive解决方案

如果你不喜欢Flux/Redux，或者想要更加reactive，不用失望！还有很多方案供你选择，这里是你可能需要的：

- cycle.js（“一个更清爽的reactive框架”）
- rx-flux（“Flux与Rxjs结合的产物”）
- redux-rx（“Redux的Rxjs工具库”）
- mobservable（“可观测的数据，reactive的功能，简洁的代码”）

路由

现在几乎所有app都有路由功能。如果你在浏览器中使用React.js，你将会接触到这个点，并为其选择一个库。

我们选择的是出自优秀rackt社区的react-router，这个社区总是能为React.js爱好者们带来高质量的资源。

要使用react-router需要查看它的文档，但更重要的是：如果你使用Flux/Redux，我们推荐你将路由state与store或全局state保持同步。

同步路由state可以让Flux/Redux来控制路由行为，并让组件读取到路由信息。

Redux的用户可以使用redux-simple-router来省点事儿。

代码分割，懒加载

只有一小部分webpack的用户知道，应用代码是可以分割成多个js包的。

```
require.ensure([], () => {
    const Profile = require('./Profile.js')
    this.setState({
        currentComponent: Profile
    })
})
```

这对于大型应用十分有用，因为用户浏览器不用下载那些很少会使用到的代码，比如Profile页。

多js包会导致额外的HTTP请求数，但对于HTTP/2的多路复用，完全不是问题。

与chunk hashing 结合可以优化缓存命中率。

下个版本的react-router将会对代码分隔做更多支持。

对于react-router的未来规划，可以去看博文Ryan Florence: Welcome to Future of Web Application Delivery。

组件

很多人都在抱怨JSX，但首先要知道，它只是React中可选的一项能力。

最后，它们都会被Bable编译成JavaScript。你可以继续使用JavaScript编写代码，但是在处理HTML时使用JSX会感觉更自然。特别是对于那些不懂js的人，他们可以只修改HTML相关的部分。

> JSX是一个类似于XML的JavaScript扩展，可以配合一个简单的语法编译工具来使用它。——深入浅出JSX

如果你想了解更多JSX的内容，查看文章JSX Looks Like An Abomination - But it’s Good for You。

使用类

React中可以顺畅地使用ES2015的Class语法。

```
class HelloMessage extends React.Component {
    render() {
        return <div>Hello {this.props.name}</div>
    }
}
```

我们在高阶组件和mixins之间更看重前者，所以抛弃createClass更像是一个语法问题，而不是技术问题。（译者注：在Class语法中，React组件的mixins方法将无法使用。）我们认为使用createClass和React.Component没有对错之分。

属性类型（PropType）

如果你以前不检查props的类型，那么2016你应该开始改正了。它会帮你节省未来很多时间，相信我。

```
MyComponent.propTypes = {
    isLoading: PropTypes.bool.isRequired,
    items: ImmutablePropTypes.listOf(
        ImmutablePropTypes.contains({
            name: PropTypes.string.isRequired,
        })
    ).isRequired
}
```

是的，同时也尽可能使用react-immutable-proptypes检查Immutable.js的props。

高阶组件（Higher order components）

minins将死，ES6的Class将不对其进行支持，我们需要寻找新的方法。

什么是高阶组件？

```
PassData({ foo: 'bar' })(MyComponent)
```

简单地，你创建一个从原生组件继承下来的组件，并且扩展了原始组件的行为。你可以在多种场景来使用它，比如鉴权：requireAuth({ role: 'admin' })(MyComponent)（检查用户是否在高级组件中，如果还没有登录就进行跳转），或者将组件与Flux/Redux的store相连通。

在RisingStack，我们也喜欢分离数据拉取和controller类的逻辑到高阶组件中，这样可以尽可能地保持view层的简单。

测试

好的代码覆盖测试是开发周期中的重要一环。幸运的是，React.js社区有很多这样的库来帮助我们。

组件测试

我们最喜爱的组件测试库是AirBnb的enzyme。有了它的浅渲染特性，可以对组件的逻辑和渲染结果进行测试，非常棒对不对？它现在还不能替代selenium测试，但是将前端测试提升到了一个新高度。

```
it('simulates click events', () => {
    const onButtonClick = sinon.spy()
    const wrapper = shallow(
        <Foo onButtonClick={onButtonClick} />
    )
    wrapper.find('button').simulate('click')
    expect(onButtonClick.calledOnce).to.be.true
})
```

看起来很清爽，不是吗？

你使用chai来作为断言库吗？你会喜欢chai-envzime的。

Redux测试

测试一个reducer非常简单，它响应actions然后将原来的state转为新的state：

```
it('should set token', () => {
    const nextState = reducer(undefined, {
        type: USER_SET_TOKEN,
        token: 'my-token'
    })

    // immutable.js state output
    expect(nextState.toJS()).to.be.eql({
        token: 'my-token'
    })
})
```

测试actions也很简单，但是异步actions就不一样了。测试异步的redux actions我们推荐redux-mock-store，它能帮不少忙。

```
it('should dispatch action', (done) => {
    const getState = {}
    const action = { type: 'ADD_TODO' }
    const expectedActions = [action]

    const store = mockStore(getState, expectedActions, done)
    store.dispatch(action)
})
```

关于更深入的redux测试，请参考官方文档。

使用npm

虽然React.js并不依赖代码构建工具，我们推荐Webpack和Browserify，它们都具有npm出色的能力。Npm有很多React.js的package，还可以帮助你优雅地管理依赖。

（请不要忘记复用你自己的组件，这是优化代码的绝佳方式。）

包大小（Bundle size）

这本身不是一个React相关的问题，但多数人都会对其React进行打包，所以我在这里提一下。

当你对源代码进行构建时，要保持对包大小的关注。要将其控制在最小体积，你需要思考如何require/import依赖。

查看下面的代码片段，有两种方式可以对输出产生重大影响：

```
import { concat, sortBy, map, sample } from 'lodash'

// vs.
import concat from 'lodash/concat';
import sortBy from 'lodash/sortBy';
import map from 'lodash/map';
import sample from 'lodash/sample';
```

查看Reduce Your bundle.js File Size By Doing This One Thing，获取更多详情。

我们喜欢将代码分隔到vendors.js和app.js，因为第三方代码的更新频率比我们自己带吗低很多。

对输出文件进行hash命名（WebPack中的chunk hash），并使用长缓存，我们可以显著地减少访问用户需要下载的代码。结合代码懒加载，优化效果可想而知。

如果你对WebPack还很陌生，可以去看超赞的React webpack指南。

组件级的hot reload

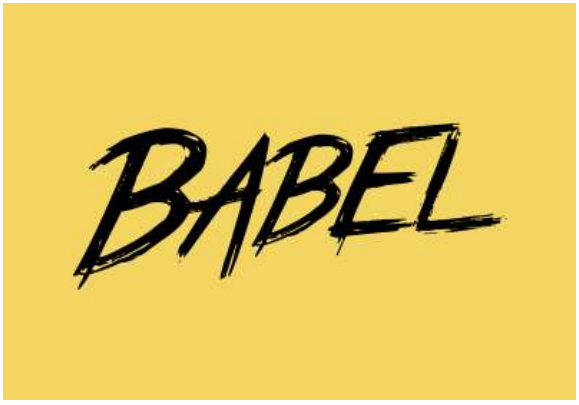
如果你曾使用livereload写过单页面应用，你可能知道当在处理一些与状态相关的事情，一点代码保存整个页面就刷新了，这种体验有多烦人。你需要逐步点击操作到刚才的环节，然后在这样的重复中奔溃。

在React开发中，是可以reload一个组件，同时保持它的state不变——耶，从此无需苦恼！

搭建hot reload，可参考react-transform-boilerplate。

使用ES2015

前面提到过，在React.js中使用的JSX，最终会被Babel.js进行编译。



Bable的能力还不止这些，它可以让我们在浏览器中放心地使用ES6/ES2015。在RisingStack，我们在服务器端和客户端都使用了ES2015的特性，ES2015已经可以在最新的LTS Node.js版本中使用了。

代码检查（Linters）

也许你已经对你的代码制定了代码规范，但是你知道React的各种代码规范吗？我们建议你选择一个代码规范，然后照着下面说的来做。

在RisingStack，我们强制将linters运行在持续集成（CI）系统，已经git push功能上。查看pre-push和pre-commit。

我们使用标准的JavaScript代码风格，并使用eslint-plugin-react来检查React.js代码。

（是的，我们已经不再使用分号了）

GraphQL和Relay

GraphQL和Relay是相关的新技术。在RisingStack，我们不在生产环境使用它们，暂时保持关注。

我们写了一个Relay的MongoDB ORM，叫做graffiti，可以使用你已有的mongoose models来创建GraphQL server。

如果你想学习这些新技术，我们建议你去看看这个库，然后写几个demo玩玩。

这些React.js最佳实践的核心点

有些优秀的技术和库其实跟React都没什么关系，关键在于要关注社区都在做些什么。2015这一年，React社区被Elm架构启发了很多。

如果你知道其他2016年大家应该使用的React.js工具，请留言告诉我们。