Airbnb React/JSX 编码规范

算是最合理的React/JSX编码规范之一了

此编码规范主要基于目前流行的JavaScript标准，尽管某些其他约定(如async/await，静态class属性)可能在不同的项目中被引入或者被禁用。目前的状态是任何stage-3之前的规范都不包括也不推荐使用。

内容目录

Basic Rules 基本规范

- 每个文件只写一个模块.
    - 但是多个无状态模块可以放在单个文件中. eslint: `react/no-multi-comp`.
- 推荐使用JSX语法.
- 不要使用 `React.createElement`，除非从一个非JSX的文件中初始化你的app.

创建模块

Class vs React.createClass vs stateless

- 如果你的模块有内部状态或者是`refs`, 推荐使用 `class extends React.Component` 而不是 `React.createClass`. eslint: `react/prefer-es6-class` `react/prefer-stateless-function`

React JSX

```
// bad
const Listing = React.createClass({
  // ...
  render() {
    return <div>{this.state.hello}</div>;
  }
});

// good
class Listing extends React.Component {
  // ...
  render() {
    return <div>{this.state.hello}</div>;
  }
}
```

如果你的模块没有状态或是没有引用`refs`， 推荐使用普通函数（非箭头函数）而不是类:

React JSX

```
// bad
class Listing extends React.Component {
  render() {
    return <div>{this.props.hello}</div>;
  }
}

// bad (relying on function name inference is discouraged)
const Listing = ({ hello }) => (
  <div>{hello}</div>
);

// good
function Listing({ hello }) {
  return <div>{hello}</div>;
}
```

Mixins

- 不要使用 mixins.

> 为什么? Mixins 会增加隐式的依赖，导致命名冲突，并且会以雪球式增加复杂度。在大多数情况下Mixins可以被更好的方法替代，如：组件化，高阶组件，工具模块等。

Naming 命名

- **扩展名**: React模块使用 `.jsx` 扩展名.
- **文件名**: 文件名使用帕斯卡命名. 如, `ReservationCard.jsx`.
- **引用命名**: React模块名使用帕斯卡命名，实例使用骆驼式命名. eslint: `react/jsx-pascal-case`

React JSX

```jsx
// bad
import reservationCard from './ReservationCard';

// good
import ReservationCard from './ReservationCard';

// bad
const ReservationItem = <ReservationCard />;

// good
const reservationItem = <ReservationCard />;
```

- **模块命名**: 模块使用当前文件名一样的名称. 比如 `ReservationCard.jsx` 应该包含名为 `ReservationCard`的模块. 但是，如果整个文件夹是一个模块，使用 `index.js`作为入口文件，然后直接使用 `index.js` 或者文件夹名作为模块的名称:

React JSX

```jsx
// bad
import Footer from './Footer/Footer';

// bad
import Footer from './Footer/index';

// good
import Footer from './Footer';
```

- **高阶模块命名**: 对于生成一个新的模块，其中的模块名 `displayName` 应该为高阶模块名和传入模块名的组合. 例如, 高阶模块 `withFoo()`, 当传入一个 `Bar` 模块的时候， 生成的模块名 `displayName` 应该为 `withFoo(Bar)`.

> 为什么? 一个模块的 `displayName` 可能会在开发者工具或者错误信息中使用到，因此有一个能清楚的表达这层关系的值能帮助我们更好的理解模块发生了什么，更好的Debug.

React JSX

```jsx
// bad
export default function withFoo(WrappedComponent) {
  return function WithFoo(props) {
    return <WrappedComponent {...props} foo />;
  }
}

// good
export default function withFoo(WrappedComponent) {
  function WithFoo(props) {
    return <WrappedComponent {...props} foo />;
  }

  const wrappedComponentName = WrappedComponent.displayName
    || WrappedComponent.name
    || 'Component';

  WithFoo.displayName = `withFoo(${wrappedComponentName})`;
  return WithFoo;
}
```

- **属性命名**: 避免使用DOM相关的属性来用作其他的用途。

> 为什么？对于`style`和`className`这样的属性名，我们都会默认它们代表一些特殊的含义，如元素的样式，CSS class的名称。在你的应用中使用这些属性来表示其他的含义会使你的代码更难阅读，更难维护，并且可能会引起bug。

React JSX

```jsx
// bad
<MyComponent style="fancy" />

// good
<MyComponent variant="fancy" />
```

Declaration 声明模块

- 不要使用 `displayName` 来命名React模块，而是使用引用来命名模块， 如 class 名称.

React JSX

```
// bad
export default React.createClass({
  displayName: 'ReservationCard',
  // stuff goes here
});

// good
export default class ReservationCard extends React.Component {
}
```

Alignment 代码对齐

- 遵循以下的JSX语法缩进/格式. eslint: react/jsx-closing-bracket-location react/jsx-closing-tag-location

React JSX

```
// bad
<Foo superLongParam="bar"
     anotherSuperLongParam="baz" />

// good, 有多行属性的话, 新建一行关闭标签
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
/>

// 若能在一行中显示, 直接写成一行
<Foo bar="bar" />

// 子元素按照常规方式缩进
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
>
  <Quux />
</Foo>
```

Quotes 单引号还是双引号

- 对于JSX属性值总是使用双引号("), 其他均使用单引号('). eslint: jsx-quotes

> 为什么? HTML属性也是用双引号, 因此JSX的属性也遵循此约定.

React JSX

```
// bad
<Foo bar='bar' />

// good
<Foo bar="bar" />

// bad
<Foo style={{ left: "20px" }} />

// good
<Foo style={{ left: '20px' }} />
```

Spacing 空格

- 总是在自动关闭的标签前加一个空格，正常情况下也不需要换行. eslint: `no-multi-spaces`, `react/jsx-tag-spacing`

React JSX

```
// bad
<Foo/>

// very bad
<Foo                 />

// bad
<Foo
 />

// good
<Foo />
```

- 不要在JSX {} 引用括号里两边加空格. eslint: `react/jsx-curly-spacing`

React JSX

```
// bad
<Foo bar={ baz } />

// good
<Foo bar={baz} />
```

Props 属性

- JSX属性名使用骆驼式风格`camelCase`.

React JSX

```jsx
// bad
<Foo
  UserName="hello"
  phone_number={12345678}
/>

// good
<Foo
  userName="hello"
  phoneNumber={12345678}
/>
```

- 如果属性值为 `true`, 可以直接省略. eslint: `react/jsx-boolean-value`

React JSX

```jsx
// bad
<Foo
  hidden={true}
/>

// good
<Foo
  hidden
/>

// good
<Foo hidden />
```

- `<img>` 标签总是添加 `alt` 属性. 如果图片以presentation(感觉是以类似PPT方式显示?)方式显示， `alt` 可为空, 或者`<img>` 要包含`role="presentation"`. eslint: `jsx-a11y/alt-text`

React JSX

```jsx
// bad
<img src="hello.jpg" />

// good
<img src="hello.jpg" alt="Me waving hello" />

// good
<img src="hello.jpg" alt="" />

// good
<img src="hello.jpg" role="presentation" />
```

- 不要在 `alt` 值里使用如 “image”, “photo”, or “picture”包括图片含义这样的词， 中文也一样. eslint: `jsx-a11y/img-redundant-alt`

为什么? 屏幕助读器已经把 `img` 标签标注为图片了, 所以没有必要再在 `alt` 里说明了.

React JSX

```jsx
// bad
<img src="hello.jpg" alt="Picture of me waving hello" />

// good
<img src="hello.jpg" alt="Me waving hello" />
```

- 使用有效正确的 aria `role`属性值 ARIA roles. eslint: `jsx-a11y/aria-role`

React JSX

```jsx
// bad - not an ARIA role
<div role="datepicker" />

// bad - abstract ARIA role
<div role="range" />

// good
<div role="button" />
```

- 不要在标签上使用 `accessKey` 属性. eslint: `jsx-a11y/no-access-key`

为什么? 屏幕助读器在键盘快捷键与键盘命令时造成的不统一性会导致阅读性更加复杂.

React JSX

```jsx
// bad
<div accessKey="h" />

// good
<div />
```

- 避免使用数组的index来作为属性key的值，推荐使用唯一ID. (为什么?)

```
// bad
{todos.map((todo, index) =>
<Todo
  {...todo}
  key={index}
/>
)}

// good
{todos.map(todo => (
<Todo
  {...todo}
  key={todo.id}
/>
))}
```

- 对于所有非必须的属性，总是手动去定义`defaultProps`属性.

> 为什么? propTypes 可以作为模块的文档说明, 并且声明 defaultProps 的话意味着阅读代码的人不需要去假设一些默认值。更重要的是, 显示的声明默认属性可以让你的模块跳过属性类型的检查.

React JSX

```jsx
// bad
function SFC({ foo, bar, children }) {
return <div>{foo}{bar}{children}</div>;
}
SFC.propTypes = {
foo: PropTypes.number.isRequired,
bar: PropTypes.string,
children: PropTypes.node,
};

// good
function SFC({ foo, bar, children }) {
return <div>{foo}{bar}{children}</div>;
}
SFC.propTypes = {
foo: PropTypes.number.isRequired,
bar: PropTypes.string,
children: PropTypes.node,
};
SFC.defaultProps = {
bar: '',
children: null,
};
```

- 尽可能少地使用扩展运算符

> 为什么? 除非你很想传递一些不必要的属性。对于React v15.6.1和更早的版本，你可以给DOM传递一些无效的HTML属性

例外情况:

- 使用了变量提升的高阶组件

React JSX

```jsx
function HOC(WrappedComponent) {
return class Proxy extends React.Component {
  Proxy.propTypes = {
    text: PropTypes.string,
    isLoading: PropTypes.bool
  };

  render() {
    return <WrappedComponent {...this.props} />
  }
}
}
```

- 只有在清楚明白扩展对象时才使用扩展运算符。这非常有用尤其是在使用Mocha测试组件的时候。

React JSX

```
export default function Foo {
const props = {
  text: '',
  isPublished: false
}

return (<div {...props} />);
}
```

特别提醒：尽可能地筛选出不必要的属性。同时，使用prop-types-exact来预防问题出现。

React JSX

```
// good
render() {
const { irrelevantProp, ...relevantProps  } = this.props;
return <WrappedComponent {...relevantProps} />
}

// bad
render() {
const { irrelevantProp, ...relevantProps  } = this.props;
return <WrappedComponent {...this.props} />
}
```

Refs

- 总是在Refs里使用回调函数. eslint: `react/no-string-refs`

React JSX

```
// bad
<Foo
  ref="myRef"
/>

// good
<Foo
  ref={(ref) => { this.myRef = ref; }}
/>
```

Parentheses 括号

- 将多行的JSX标签写在 ( )里. eslint: `react/jsx-wrap-multilines`

React JSX

```
// bad
render() {
  return <MyComponent className="long body" foo="bar">
           <MyChild />
         </MyComponent>;
}

// good
render() {
  return (
    <MyComponent className="long body" foo="bar">
      <MyChild />
    </MyComponent>
  );
}

// good, 单行可以不需要
render() {
  const body = <div>hello</div>;
  return <MyComponent>{body}</MyComponent>;
}
```

Tags 标签

- 对于没有子元素的标签来说总是自己关闭标签. eslint: `react/self-closing-comp`

React JSX

```
// bad
<Foo className="stuff"></Foo>

// good
<Foo className="stuff" />
```

- 如果模块有多行的属性， 关闭标签时新建一行. eslint: `react/jsx-closing-bracket-location`

React JSX

```
// bad
<Foo
  bar="bar"
  baz="baz" />

// good
<Foo
  bar="bar"
  baz="baz"
/>
```

Methods 函数

- 使用箭头函数来获取本地变量.

React JSX

```
function ItemList(props) {
  return (
    <ul>
      {props.items.map((item, index) => (
        <Item
          key={item.key}
          onClick={() => doSomethingWith(item.name, index)}
        />
      ))}
    </ul>
  );
}
```

- 当在 `render()` 里使用事件处理方法时，提前在构造函数里把 `this` 绑定上去. eslint: `react/jsx-no-bind`

为什么? 在每次 `render` 过程中， 再调用 `bind` 都会新建一个新的函数，浪费资源.

```
// bad
class extends React.Component {
  onClickDiv() {
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv.bind(this)} />;
  }
}

// good
class extends React.Component {
  constructor(props) {
    super(props);

    this.onClickDiv = this.onClickDiv.bind(this);
  }

  onClickDiv() {
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv} />;
  }
}
```

- 在React模块中，不要给所谓的私有函数添加 _ 前缀，本质上它并不是私有的.

> 为什么? _ 下划线前缀在某些语言中通常被用来表示私有变量或者函数。但是不像其他的一些语言，在JS中没有原生支持所谓的私有变量，所有的变量函数都是共有的。尽管你的意图是使它私有化，在之前加上下划线并不会使这些变量私有化，并且所有的属性（包括有下划线前缀及没有前缀的）都应该被视为是共有的。了解更多详情请查看Issue #1024, 和 #490 。

React JSX

```
// bad
React.createClass({
  _onClickSubmit() {
    // do stuff
  },

  // other stuff
});

// good
class extends React.Component {
  onClickSubmit() {
    // do stuff
  }

  // other stuff
}
```

- 在 `render` 方法中总是确保 `return` 返回值. eslint: `react/require-render-return`

React JSX

```
// bad
render() {
  (<div />);
}

// good
render() {
  return (<div />);
}
```

Ordering React 模块生命周期

- `class extends React.Component` 的生命周期函数:

1. 可选的 `static` 方法
2. `constructor` 构造函数
3. `getChildContext` 获取子元素内容
4. `componentWillMount` 模块渲染前
5. `componentDidMount` 模块渲染后
6. `componentWillReceiveProps` 模块将接受新的数据
7. `shouldComponentUpdate` 判断模块需不需要重新渲染
8. `componentWillUpdate` 上面的方法返回 `true`， 模块将重新渲染
9. `componentDidUpdate` 模块渲染结束
10. `componentWillUnmount` 模块将从DOM中清除, 做一些清理任务
11. *点击回调或者事件处理器* 如 `onClickSubmit()` 或 `onChangeDescription()`
12. *render 里的 getter 方法* 如 `getSelectReason()` 或 `getFooterContent()`

13. *可选的 render 方法* 如 `renderNavigation()` 或 `renderProfilePicture()`
14. `render` render() 方法

- 如何定义 `propTypes`, `defaultProps`, `contextTypes`, 等等其他属性...

React JSX

```jsx
import React from 'react';
import PropTypes from 'prop-types';

const propTypes = {
  id: PropTypes.number.isRequired,
  url: PropTypes.string.isRequired,
  text: PropTypes.string,
};

const defaultProps = {
  text: 'Hello World',
};

class Link extends React.Component {
  static methodsAreOk() {
    return true;
  }

  render() {
    return <a href={this.props.url} data-id={this.props.id}>{this.props.text}</a>;
  }
}

Link.propTypes = propTypes;
Link.defaultProps = defaultProps;

export default Link;
```

- `React.createClass` 的生命周期函数，与使用class稍有不同: eslint: `react/sort-comp`

1. `displayName` 设定模块名称
2. `propTypes` 设置属性的类型
3. `contextTypes` 设置上下文类型
4. `childContextTypes` 设置子元素上下文类型
5. `mixins` 添加一些mixins
6. `statics`
7. `defaultProps` 设置默认的属性值
8. `getDefaultProps` 获取默认属性值
9. `getInitialState` 或者初始状态
10. `getChildContext`
11. `componentWillMount`
12. `componentDidMount`

13. `componentWillReceiveProps`
14. `shouldComponentUpdate`
15. `componentWillUpdate`
16. `componentDidUpdate`
17. `componentWillUnmount`
18. *clickHandlers or eventHandlers* like `onClickSubmit()` or `onChangeDescription()`
19. *getter methods for `render`* like `getSelectReason()` or `getFooterContent()`
20. *Optional render methods* like `renderNavigation()` or `renderProfilePicture()`
21. `render`

isMounted

- 不要再使用 `isMounted`. eslint: `react/no-is-mounted`

> 为什么? `isMounted` 反人类设计模式:(), 在 ES6 classes 中无法使用， 官方将在未来的版本里删除此方法.

⬆ 回到顶部