# 代码组织和测试

本章将专注在几个重要话题来保证在一个规模增长的应用中代码的可维护性。你将了解如何去组织代码，以便在构建你的工程目录和文件时时遵循最佳实践。本章你将学会的另外一个话题是测试，这对你的代码健壮性非常重要。本章也会结合之前的练习项目来为你介绍这几个话题。

# ES6模块：Import 和 Export

在 JavaScript ES6 中你可以从模块中导入和导出某些功能。这些功能可以是函数、类、组件、常量等等。基本上你可以将所有东西都赋值到一个变量上。模块可以是单个文件，或者一个带有入口文件的文件夹。

本书的开头，在你使用*create-react-app*初始化你的应用后，应该有几条`import`和`export`语句已经在应用初始化的文件中。现在正合适解释这些。

`import`和`export`语句可以帮助你在多个不同的文件间共享代码。在此之前，JavaScript 生态中已经有好几种方案了。曾经一度很糟，你需要的是遵循一套标准范式，而不是为了同一件事而采取多种方法。从 JavaScript ES6 后，现在是一种原生的方式了。

此外这些语言还有利于代码分割。代码风格就是将代码分配到多个文件中去，以保持代码的重用性和可维护性。前者得以成立是因为你可以在不同的文件中导入相同的代码片段。而后者得以成立是因为你维护的代码是唯一的代码源。

最后但也很重要的是，它能帮助你思考代码封装。不是所有的功能都需要从一个文件导出。其中一些功能应该只在定义它的文件中使用。一个文件导出的功能是这个文件公共 API。只有导出的功能才能被其他地方重用。这遵循了封装的最佳实践。

那么，我们回到实践中。如何让这些`import`和`export`语句工作起来？下面的几个例子展示使用这些语句在两个文件中共享一个或多个变量。最后，这个方式可以扩展到多个文件中，而不单单只共享变量。

你可以导出一个或者多个变量。这称为一个命名的导出。

{title="Code Playground: file1.js",lang="javascript"}

```
const firstname = 'robin';
const lastname = 'wieruch';

export { firstname, lastname };
```

并在另外一个文件用相对第一文件的相对路径导入。

{title="Code Playground: file2.js",lang="javascript"}

```
import { firstname, lastname } from './file1.js';

console.log(firstname);
// output: robin
```

你也可以用对象的方式导入另外文件的全部变量。

{title="Code Playground: file2.js",lang="javascript"}

```
import * as person from './file1.js';

console.log(person.firstname);
// output: robin
```

导入可以有一个别名。可能发生在在输入多个文件中有相同命名的导出的时候。这就是为什么你可以使用别名。

{title="Code Playground: file2.js",lang="javascript"}

```
import { firstname as foo } from './file1.js';

console.log(foo);
// output: robin
```

最后但也很重要的是，还存在一种 `default` 语句。可以被用在一些使用情况下：

- 为了导出和导入单一功能
- 为了强调一个模块输出 API 中的主要功能
- 这样可以向后兼容ES5只有一个导出物的功能

{title="Code Playground: file1.js",lang="javascript"}

```
const robin = {
  firstname: 'robin',
  lastname: 'wieruch',
};

export default robin;
```

你可以在导入 default 输出时省略花括号。

{title="Code Playground: file2.js",lang="javascript"}

```
import developer from './file1.js';

console.log(developer);
// output: { firstname: 'robin', lastname: 'wieruch' }
```

此外，输入的名称可以与导入的 default 名称不一样，你也可以将其与命名的导出和导入语句使用同一个名称。

{title="Code Playground: file1.js",lang="javascript"}

```
const firstname = 'robin';
const lastname = 'wieruch';

const person = {
  firstname,
  lastname,
};

export {
  firstname,
  lastname,
};

export default person;
```

{title="Code Playground: file2.js",lang="javascript"}

```
import developer, { firstname, lastname } from './file1.js';

console.log(developer);
// output: { firstname: 'robin', lastname: 'wieruch' }
```

```
console.log(firstname, lastname);
// output: robin wieruch
```

在命名的导出中，你可以省略多余行直接导出变量。

{title="Code Playground: file1.js",lang="javascript"}

```
export const firstname = 'robin';
export const lastname = 'wieruch';
```

这些是 ES6 模块的主要功能。它们能帮助你组织你的代码，维护你的代码，设计可重用的模块 API。你也可以为了测试导入和导出功能。你将会在接下来的章节中做到这一点。

### 练习

- 阅读 ES6 import
- 阅读 ES6 export

## 代码组织与 ES6 模块

你可能想知道：为什么不按照 *src/App.js* 文件中的代码分割方式呢？这个文件中，我们已经有了多个文件/文件夹（模块）了。为了学习 React 的缘故，将这些模块放到一个地方是合理的。但是一旦你的应用增长，你应该考虑将这些组件放到多个模块中去，只有这种方式你的应用才能扩展。

接下来，我会提供几种你可能会用到的模块结构。我会推荐在读完本书后作为一个练习应用。为了保持本书的简洁性，我不会展示代码分割，并且会在接下来的章节继续使用 *src/App.js* 文件。

一种可能的模块结构类似：

{title="Folder Structure",lang="text"}

```
src/
  index.js
  index.css
  App.js
  App.test.js
  App.css
  Button.js
```

```
  Button.test.js
  Button.css
  Table.js
  Table.test.js
  Table.css
  Search.js
  Search.test.js
  Search.css
```

这里将组件封装到各自文件中，但是这看起来不是很好。你可以看到非常多的命名冗余，并且只有文件的扩展文字不同。另外一种模块的结构大概类似：

{title="Folder Structure",lang="text"}

```
src/
  index.js
  index.css
  App/
    index.js
    test.js
    index.css
  Button/
    index.js
    test.js
    index.css
  Table/
    index.js
    test.js
    index.css
  Search/
    index.js
    test.js
    index.css
```

这看起来比之前清晰多了。文件名中的 index 名称表示他是这个文件夹的入口文件。这仅仅是一个命名共识，你也可以使用你习惯的命名。在这个模块结构中，一个组件被 JavaScript 文件中组件声明，样式文件，测试共同定义。

另外一个步骤可能要将 App 组件中的变量抽出。这些变量用来组合出 Hacker News 的 API URL。

{title="Folder Structure",lang="text"}

```
src/
  index.js
  index.css
# leanpub-start-insert
  constants/
    index.js
  components/
# leanpub-end-insert
    App/
      index.js
      test.js
      index..css
    Button/
      index.js
      test.js
      index..css
    ...
```

自然这些模块会分割到 *src/constants/* 和 *src/components/* 中去。现在 *src/constants/index.js* 文件可能看起来类似下面这样：

{title="Code Playground: src/constants/index.js",lang="javascript"}

```
export const DEFAULT_QUERY = 'redux';
export const DEFAULT_HPP = '100';
export const PATH_BASE = 'https://hn.algolia.com/api/v1';
export const PATH_SEARCH = '/search';
export const PARAM_SEARCH = 'query=';
export const PARAM_PAGE = 'page=';
export const PARAM_HPP = 'hitsPerPage=';
```

*App/index.js* 文件可以导入这些变量，以便使用。

{title="Code Playground: src/components/App/index.js",lang=javascript}

```
import {
  DEFAULT_QUERY,
  DEFAULT_HPP,
  PATH_BASE,
  PATH_SEARCH,
  PARAM_SEARCH,
  PARAM_PAGE,
  PARAM_HPP,
} from '../constants/index.js';
```

```
...
```

当你使用 *index.js* 这个命名共识的时候，你可以在相对路径中省略文件名。

{title="Code Playground: src/components/App/index.js",lang=javascript}

```
import {
  DEFAULT_QUERY,
  DEFAULT_HPP,
  PATH_BASE,
  PATH_SEARCH,
  PARAM_SEARCH,
  PARAM_PAGE,
  PARAM_HPP,
# leanpub-start-insert
} from '../constants';
# leanpub-end-insert

...
```

但是 *index.js* 文件名称后面发生了什么？这个约定是在 node.js 世界里面被引入的。index 文件是一个模块的入口。它描述了一个模块的公共 API。外部模块只允许通过 *index.js* 文件导入模块中的共享代码。考虑用下面虚构的模块结构进行演示：

{title="Folder Structure",lang="text"}

```
src/
  index.js
  App/
    index.js
  Buttons/
    index.js
    SubmitButton.js
    SaveButton.js
    CancelButton.js
```

这个 *Buttons/* 文件夹有多个按钮组件定义在了不同的文件中。每个文件都 `export default` 特定的组件，使组件能够被 *Buttons/index.js* 导入。*Buttons/index.js* 文件导入所有不同的表现的按钮，并将他们导出作为模块的公共 API。

{title="Code Playground: src/Buttons/index.js",lang="javascript"}

```
import SubmitButton from './SubmitButton';
import SaveButton from './SaveButton';
import CancelButton from './CancelButton';

export {
  SubmitButton,
  SaveButton,
  CancelButton,
};
```

现在 *src/App/index.js* 可以通过定位在 *index.js* 文件模块的公共 API 导入这些按钮。

{title="Code Playground: src/App/index.js",lang=javascript}

```
import {
  SubmitButton,
  SaveButton,
  CancelButton
} from '../Buttons';
```

在这些约束下，通过其他文件导入而不是通过 *index.js* 模块的话会是糟糕的实践。这会破坏封装的原则。

{title="Code Playground: src/App/index.js",lang=javascript}

```
// 糟糕的实践，不要这样做

import SubmitButton from '../Buttons/SubmitButton';
```

现在你知道如何在模块与封装约束下重构你的代码。如我所说，为了保持本书的简洁，我不会这么做。但是你应该在读完这本书后做些重构。

## 练习

在你读完本书后，重构你的 *src/App.js* 文件到多个组件模块中去。

# 快照测试和 Jest

本书不会深入测试这个话题，但是不得不提一下。在编程中测试代码是基本，并应该被视为必不可少的。你应该想去保持高质量的代码并确保一切如预期般工作。

也许你听过测试金字塔。其中有端到端测试，集成测试和单元测试。如果你对这些不熟悉，本书会简单描述下。单元测试用来测试一块独立的小块代码。它可以是被一个单元测试覆盖的一个函数。然而有时候这个测试单元单独运行得很好，但是结合其他单元会就不能正常工作了。这些单元需要被视为一个组单元。集成测试可以覆盖验证是否这些单元组如预期般工作。最后但不意味最不重要，端到端测试是一个真实用户场景的模拟。可能是自动地启动一个浏览器，模拟一个用户在 Web 应用中的登录流程。单元测试相对来说快速而且易于书写和维护，端到端测试反之。

每种测试我们需要多少呢？你需要很多的单元测试去覆盖代码中不同的函数。然后，你需要一些基础测试，去覆盖最重要的函数功能的联动，是否如预期一样工作。最后但也很重要的是，你可能需要一点点端到端测试去模拟你 Web 应用程序中的关键情境。这就把测试简单说了一遍。

你应该怎样将这些知识点拿去测试你的 React 测试呢？React 中测试的基础是组件测试，基本可以视作单元测试，还有部分的快照测试。在后面的章节中管理组件相关的测试需要用到一个叫 Enzyme 的库。本章中，你会主要关注另外一种测试：快照测试。这里正好引入 Jest。

Jest 是一个在 Facebook 使用的测试框架。在 React 社区，它被用来做 React 的组件测试。幸好 *create-react-app* 已经包含了 Jest，所以你不需要担心启动配置的问题。

我们开始测试第一个组件吧。在此之前，你必须先将需要测试的组件从 *src/App.js* 导出。之后，你可以在不同的单个文件里去测试，相信你已经在代码组织章节学会了怎么去做。

{title="src/App.js",lang=javascript}

```
...

class App extends Component {
  ...
}
```

```
...

export default App;

# leanpub-start-insert
export {
  Button,
  Search,
  Table,
};
# leanpub-end-insert
```

在 *App.test.js* 文件中，你可以看到 *create-react-app* 创建的第一个测试。它验证了 App 组件在渲染的时候没有任何错误发生。

{title="src/App.test.js",lang=javascript}

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
});
```

“it” 块描述了一个测试用例。它需要带有一段测试的描述，这个测试块可以成功或者失败。或者你可以将它包裹在一个 “describe” 块中来定义一个测试套件。一个测试套件可能包含一系列关于特定组件的 “it” 块。在后面你会看到 “describe” 块的。这两种块都是用来区分和组织你的测试用例的。

你可以使用 *create-react-app* 提供的命令行测试脚本来运行测试用例。

{title="Command Line",lang="text"}

```
npm test
```

**注意**：如果当你第一次运行 App 组件测试的时候碰到了错误，可能是因为是在组件的 `componentDitMount()` 方法中触发的 `fetchSearchTopStories()` 中使用的 fetch 不被支持的原因。你可以通过下面两部解决：

- 在命令行安装：`npm install isomorphic-fetch`

- 在你的 *App.js* 引入： `import fetch from 'isomorphic-fetch'`

Jest 赋予你写快照测试的能力。这些测试会生成一份渲染好的组件的快照，并在作和未来的快照的比对。当一个未来的测试改变了，测试会给出提示。你可以接受这个快照改变，因为你有意改变了组件实现，或者拒绝这个改变并要去调查错误的原因。快照测试可以非常好地和单元测试互补，因为这仅会比对渲染输出的差异。这并不会增加巨额的维护成本，因为只有在你有意改变组件中渲染输出的时候，才需要接受快照改变。

Jest 将快照保存在一个文件夹中。只有这样它才可以和未来的快照比对。此外这些快照也可以通过一个文件夹共享。

当你写快照之前，可能需要安装一个工具库。

{title="Command Line",lang="text"}

```
npm install --save-dev react-test-renderer
```

现在你可以用第一份快照测试来扩展 App 组件测试了。第一步，从 node 包中引入新功能，并将之前测试 App 组件的 “it” 块包裹在一个描述性的 “describe” 块中。这个测试套件仅用来测试 App 组件。

{title="src/App.test.js",lang=javascript}

```
import React from 'react';
import ReactDOM from 'react-dom';
# leanpub-start-insert
import renderer from 'react-test-renderer';
# leanpub-end-insert
import App from './App';

# leanpub-start-insert
describe('App', () => {
# leanpub-end-insert

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
  });

# leanpub-start-insert
});
# leanpub-end-insert
```

现在你可以使用 “test” 块来实现第一个快照测试了。

{title="src/App.test.js",lang=javascript}

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App from './App';

describe('App', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
  });

# leanpub-start-insert
  test('has a valid snapshot', () => {
    const component = renderer.create(
      <App />
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
# leanpub-end-insert

});
```

重新运行你的测试，看测试是成功还是失败。按理它们应该能成功。一旦你改变了 App 组件中的 render 块的输出，这个测试应该会失败。然后你可以决定是否需要更新快照，或去调查 App 组件。

基本上 `renderer.create()` 函数会创建一份你的 App 组件的快照。它会模拟渲染，并将 DOM 存储在快照中。之后，会期望这个快照和上传测试运行的快照匹配。使用这种方式，可以确保你的 DOM 保持稳定而不会意外被改变。

我们来给我们的组件添加更多的测试，第一步，Search 组件：

{title="src/App.test.js",lang=javascript}

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
# leanpub-start-insert
```

```
import App, { Search } from './App';
# leanpub-end-insert

...

# leanpub-start-insert
describe('Search', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Search>Search</Search>, div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Search>Search</Search>
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
# leanpub-end-insert
```

Search 组件中有两个和 App 组件测试中类似的测试。第一个测试简单地渲染 Search 组件成 DOM，并验证这个渲染过程没有错误。如果这里有错误的话，即使测试块中没有任何断言（比如 expect， match， equal ），测试也会中断。第二个快照测试用来渲染组件的存储快照并且和之前的快照做比对。当快照改变了，测试会失败。

接下来，你可以使用在 Search 组件中相同的测试方式，去测试 Button 组件。

{title="src/App.test.js",lang=javascript}

```
...
# leanpub-start-insert
import App, { Search, Button } from './App';
# leanpub-end-insert

...

# leanpub-start-insert
describe('Button', () => {

  it('renders without crashing', () => {
```

```
    const div = document.createElement('div');
    ReactDOM.render(<Button>Give Me More</Button>, div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Button>Give Me More</Button>
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
# leanpub-end-insert
```

最后但也很重要的是，你可以给表格组件一些初始化的 props 来做渲染一个简单的列表。

{title="src/App.test.js",lang=javascript}

```
...
# leanpub-start-insert
import App, { Search, Button, Table } from './App';
# leanpub-end-insert

...

# leanpub-start-insert
describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, obj
      { title: '2', author: '2', num_comments: 1, points: 2, obj
    ],
  };

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Table { ...props } />, div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Table { ...props } />
    );
    let tree = component.toJSON();
```

```
    expect(tree).toMatchSnapshot();
  });

});
# leanpub-end-insert
```

快照测试常常就保持这样。只需要确保组件输出不会改变。一旦输出改变了，你必须决定是否接受这个改变。否则当输出和期望输出不符合时，你需要去修复组件。

## 练习

- 当组件 render() 方法的返回值有改变时，请留意测试会如何失败的？
- 接受或者拒绝一个快照变更。
- 在后面章节中，当组件实现有改变时，保持你的快照最新。
- 读一下官方文档 React 中的 Jest。

## 单元测试和 Enzyme

Enzyme 是一个由 Airbnb 维护的测试工具，可以用来断言、操作、遍历 React 组件。你可以用它来管理单元测试，在 React 测试中与快照测试互补。

我们看看如何使用 Enzyme，第一步，因为 *create-react-app* 并不默认包含，你需要安装它。在 React 中还需要安装一个扩展库。

{title="Command Line",lang="text"}

```
npm install --save-dev enzyme react-addons-test-utils enzyme-ada
```

第二步，你需要在测试启动配置中引入，并为 React 初始化这个适配器。

{title="src/App.test.js",lang=javascript}

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
# leanpub-start-insert
import Enzyme from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
# leanpub-end-insert
import App, { Search, Button, Table } from './App';
```

```
# leanpub-start-insert
Enzyme.configure({ adapter: new Adapter() });
# leanpub-end-insert
```

现在你可以在 Table 的 “describe” 块中书写你第一个单元测试了。你会使用 `shallow()` 方法渲染你的组件，并且断言 Table 有两个子项，因为你传入了两个列表项。断言仅仅检查这个元素两个带有类名叫 `table-row` 的元素。

{title="src/App.test.js",lang=javascript}

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
# leanpub-start-insert
import Enzyme, { shallow } from 'enzyme';
# leanpub-end-insert
import Adapter from 'enzyme-adapter-react-16';
import App, { Search, Button, Table } from './App';

...

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, obj
      { title: '2', author: '2', num_comments: 1, points: 2, obj
    ],
  };

  ...

# leanpub-start-insert
  it('shows two items in list', () => {
    const element = shallow(
      <Table { ...props } />
    );

    expect(element.find('.table-row').length).toBe(2);
  });
# leanpub-end-insert

});
```

浅渲染组件不会渲染它的子组件。这样的话，你可以让测试只对一个组件负责。

Enzyme API 中总共有三种渲染机制。你已经知道了 `shallow()`，这里还有 `mount()` 和 `render()` 方法。这两种方式都会初始化父组件和所有的子组件。此外 `mount()` 还给予你调用组件生命周期的方法。但是什时候该使用哪种渲染机制呢？这里有一些建议：

- 不论怎样都优先尝试使用浅渲染（`shallow()`）
- 如果需要测试 `componentDidMount()` 或 `componentDidUpdate()`，使用 `mount()`
- 如果你想测试组件的生命周期和子组件的行为，使用 `mount()`
- 如果你想测试一个组件的子组件的渲染，并且不关心生命周期方法和减少些渲染的花销的话，使用 `render()`

你可以继续对你的组件单元测试。但要确保测试简单和可维护。否则你就需要在你的组件变更后，重构这些测试。这就是为什么 Facebook 在 Jest 中要首先引入快照测试的原因。

### 练习：

- 使用 Enzyme 对你的 Button 组件写一个单元测试
- 在接下来的章节中，保持你的单元测试的更新
- 了解更多 Enzyme 和 它的渲染 API

## 组件接口和 PropTypes

你可能知道 TypeScript 或者 Flow 在 JavaScript 中引入了类型接口。一个类型语言更不容易出错，因为代码会根据它的程序文本进行验证。编辑器或者其他工具可以在程序运行之前就捕获这些错误，可以让你的应用更健壮。

本书中不会为你介绍 Flow 或者 Typescript，但是有另外一种简洁的方式可以在组件中检查类型。React 有一种内建的类型检查器来防止出现 Bug。你可以使用 PropTypes 来描述你的组件接口。所有从父组件传递给子组件的 props 都会基于子组件的 PropTypes 接口得到验证。

本章会为你展示如何通过 PropTypes 使你的组件都类型安全。我会在接下来的章节里忽略接口的变化，因为这会加上些没必要的重构代码。但是你需要保证组件接口一直更新，确保组件类型安全。

第一步，你需要为 React 额外安装一个库。

{title="Command Line",lang="text"}

```
npm install prop-types
```

现在你可以导入 PropTypes。

{title="src/App.js",lang=javascript}

```
# leanpub-start-insert
import PropTypes from 'prop-types';
# leanpub-end-insert
```

我们开始为组件添加一个 props 接口：

{title="src/App.js",lang=javascript}

```
const Button = ({ onClick, className = '', children }) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

# leanpub-start-insert
Button.propTypes = {
  onClick: PropTypes.func,
  className: PropTypes.string,
  children: PropTypes.node,
};
# leanpub-end-insert
```

也就是说，你接受函数上所有的参数签名，并为之添加一个 PropTypes。基础的基本类型和复杂对象 PropTypes 有：

- PropTypes.array
- PropTypes.bool
- PropTypes.func
- PropTypes.number

- PropTypes.object
- PropTypes.string

此外，有另外两个 PropTypes 用来定义一个可渲染的片段（节点）。比如一段字符串，或者一个 React 元素。

- PropTypes.node
- PropTypes.element

你已经使用为 Button 组件 使用了 `node` PropTypes。关于全部的 PropTypes 定义，你可以在 React 官方文档中了解到。

现在为 Button 定义的所有 PropTypes 都是可选的。参数可以为 null 或者 undefined。但是对于那么几个需要强制定义的 props，你可以标记这些 props 是必须传递给组件的。

{title="src/App.js",lang=javascript}

```
Button.propTypes = {
# leanpub-start-insert
  onClick: PropTypes.func.isRequired,
# leanpub-end-insert
  className: PropTypes.string,
# leanpub-start-insert
  children: PropTypes.node.isRequired,
# leanpub-end-insert
};
```

`className` 不是必需的，因为它默认是空字符串。下一步你将为 Table 组件定义 PropTypes 接口。

{title="src/App.js",lang=javascript}

```
# leanpub-start-insert
Table.propTypes = {
  list: PropTypes.array.isRequired,
  onDismiss: PropTypes.func.isRequired,
};
# leanpub-end-insert
```

你可以将数组 PropTypes 的元素定义的更加明确：

{title="src/App.js",lang=javascript}

```
Table.propTypes = {
  list: PropTypes.arrayOf(
    PropTypes.shape({
      objectID: PropTypes.string.isRequired,
      author: PropTypes.string,
      url: PropTypes.string,
      num_comments: PropTypes.number,
      points: PropTypes.number,
    })
  ).isRequired,
  onDismiss: PropTypes.func.isRequired,
};
```

只有 `objectID` 是必须的，因为有部分代码依赖于它。其他的属性仅仅用来展示，就是说他们不是必须的。另外你也没办法保证 Hacker News API 总会给每一个对象都定义这些属性。

这就是 PropTypes的基本内容。但另一方面的是，你也可以在组件中定义默认 props。我们还拿 Button 组件说吧，`className` 属性在组件签名中可以有一个 ES6 的默认参数值。

{title="src/App.js",lang=javascript}

```
const Button = ({
  onClick,
  className = '',
  children
}) =>
  ...
```

你可以将它替换为 React 默认的 prop：

{title="src/App.js",lang=javascript}

```
# leanpub-start-insert
const Button = ({
  onClick,
  className,
  children
}) =>
# leanpub-end-insert
  <button
    onClick={onClick}
    className={className}
```

```
    type="button"
  >
    {children}
  </button>

# leanpub-start-insert
Button.defaultProps = {
  className: '',
};
# leanpub-end-insert
```

和 ES6 的默认参数一样，默认 prop 确保当父组件没有指定属性的时候，这个数据会被设置一个默认值。PropTypes 类型检查会在默认 props 生效后执行校验。

如果重新运行你的测试，你可能会在命令行看到一些组件的 PropTypes 错误。 可能是因为在测试中你没有传递在组件里定义为必需的 props。当你正确传递所需的值后，测试就会通过并避免这些错误。

## 练习：

- 为 Search 组件定义 PropTypes 接口
- 在接下来的章节中，确保 PropTypes 接口一直被更新
- 了解更多 React PropTypes

{pagebreak}

你已经学习到了如何组织和测试你的代码。让我们来回顾一下这最后几章吧：

- React
- PropTypes 允许你为组件定义测试检查
- Jest 允许你为组件属性快照测试
- Enzyme 允许你为组件书写单元测试
- ES6
- import 和 export 语句帮组你组织代码
- 概述
- 代码组织让你的代码符合最佳实践并具有可扩展性

你可以在官方代码库中找到源码。