React 简介

本章会对 React 做一个介绍，你也许会这样问自己：为什么我要首先学习 React 呢？本章可以回答你这个问题。然后，你会学习零配置构建第一个 React 应用，进而深入整个生态圈。进一步地你会了解关于 JSX 和 ReactDOM 的相关知识。所以准备好开始构建你的第一个 React 组件吧。

你好，我叫 React。

为什么你应该学习 React 近年来，单页面应用（SPA，single page application）变得越来越流行。像 Angular、Ember 以及 Backbone 这些框架，帮助 JavaScript 开发者构建了超越纯 JavaScript（vanilla JavaScript）和 jQuery 的现代 Web 应用。这个流行的解决方案清单并不够详尽，现在仍然有大量的 SPA 框架。如果你去关注他们的发布日期的话，大部分都属于第一代 SPA：Angular 发布于2010年，Backbone 发布于2010年，以及 Ember，发布于2011年。

Facebook 在2013年首次发布了 React。React 并不是一个 SPA 框架，而是一个视图库。也就是 MVC（Model View Controller，模型-视图-控制器）里的 V。它的功能仅仅是把组件渲染成浏览器中的可见元素。但是，围绕 React 周边的整个生态系统让构建单页面应用成为可能。

那么为什么你应该选 React 而不是其他第一代 SPA 框架呢？其他的第一代框架尝试一次性解决很多问题，而 React 仅仅帮助你构建视图层。它更多的是一个库而非框架。其背后的思路是：应用的视图应该是一系列层次分明的可组合的组件。

通过使用 React，你可以在引入更多应用部件之前重点关注视图层。其他的每一个部件都是 SPA 的一部分。这所有的部分是构成一个成熟应用的基础。这样做有两个优点。

首先，你可以按部就班地学习 SPA 的每一部分。你不用担心要一次性理解全部。这与其他的框架不同，其他的框架会在开始就需要你了解所有的内容。本书的重点把 React 作为首要的目标。其他的部分则会在后面一一讲解。

其次，SPA 的各部分都是可替换的。这样就使得 React 的周边生态圈充满新的创意。各种各样的解决方案相互之间竞争，你可以挑选最吸引你或者最适合你的使用场景的那一个。

第一代 SPA 框架更贴近企业级。它们缺乏足够的灵活性。React 时刻保持创新并且被 Airbnb、Netflix 和 Facebook 这些技术界的领导企业所采用。这些企业都对 React 的未来，以及 React 生态圈给予了充分的资助。

React 可能是构建现代 Web 应用最佳选择之一。虽然它仅仅提供了视图层抽象，但是 React 生态圈组成了一个整体的灵活且可替换的框架。React 拥有简单整洁的 API、神奇的生态圈以及很棒的社区。你可以看一下我使用 React 的经验：我为什么从 Angular 转移到了 React。我强烈推荐你仔细考虑一下，选择 React 而不是其他的框架或库。毕竟每个人都迫切地想要知道 React 接下来会引领我们走向何方。

练习

- 阅读《我为什么从 Angular 转移到了 React》
- 阅读《React 灵活的生态圈》

基本要求

如果你之前就有使用其他 SPA 框架或者库的经验，你应该已经非常熟悉 Web 开发的基本内容了。如果你刚刚开始学习 Web 开发，在学习 React 之前，你应该要了解一下 HTML、CSS 和 JavaScript ES5。本书会可以平滑地过度到 JavaScript ES6 及其后的版本。我建议你加入本书官方的 Slack 群组，在这里你可以找到找到或者提供给他人必要的帮助。

编辑器和终端

那么开发环境呢？你需要一个可用的编辑器或者IDE，以及一个终端（命令行工具）。你可以参考我编写的环境搭建指南。这篇指南主要针对的是macOS用户，但是你可以在其他操作系统上找到类似的替代方案。同样你也可以在网上找到大量关于如何设置 Web 开发环境的更详细的文章。

你也可以选择使用 git 和 GitHub 把来保存与本书相关的个人项目以及学习进度。这篇简短的指南讲述了它们的使用方法。但是再次提醒一下，你并不一定非要这样做，因为如果所有东西都从头学起可能会给你带来更多压力。所以如果你只是一个新手，并且希望专注于本书的基础内容的话，你可以跳过这一步。

Node 和 NPM

环境设置的最后一步是安装 node 和 npm。这两个工具都是用来管理你在本书中所用到的各种库的。本书中提及的 node 包需要通过 npm （ node package manager，Node包管理器）来安装，这些包可能是一些库，或者集成在一起的框架。

你可以在命令行验证 node 和 npm 的安装版本。如果没有看到任何输出结果，那就说明你需要安装他们，下面是我在写这本书的时候使用的版本：

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

```
node --version
*v8.3.0
npm --version
*v5.5.1
```

node 和 npm

本章是关于 node 和 npm 的速成教程。这个教程可能并不足够详尽，但你仍能学习到所有必要的内容。如果你已经非常熟悉他们的话，可以跳过本章。

Node 包管理器（ npm，node package manager ）帮助你通过命令行安装第三方 node 包（ package ）。这些包可能是一系列的工具函数、库或者是集成的框架。他们都是构建你应用的依赖。你可以选择把这些包安装到 node 的全局（ global ）文件夹中，或者是放到你项目本地（ project local ）文件夹中。

全局 node 包只需要一次性地安装在全局目录，可以在终端的任何地方使用。你可以通过以下命令来安装一个全局 node 包：

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

```
npm install -g <package>
```

通过 `-g` 标记指定 npm 安装一个全局的包。项目的本地包则只能在你应用里面使用。例如，React 作为一个库，将会以本地包的形式导入到你的应用中使用。你可以通过下面的命令来安装一个本地包：

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

```
npm install <package>
```

以 React 为例，应该写成：

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

```
npm install react
```

Node 包安装完成后将会保存在 *node_modules/* 文件夹里面，并且附加在会在 *package.json* 的依赖列表之后。

那么如何创建你项目专属的 *node_modules/* 文件夹和 *package.json* 文件呢？npm 可以通过一条命令来创建 npm 项目和 *package.json* 文件。只有该文件存在，你才能通过 npm 安装新的本地包。

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

```
npm init -y
```

`-y` 标记将把你的 `package.json` 内容初始化成默认值。如果你不加这个标记，就需要特别设置该文件的内容。完成 npm 项目初始化之后，你就可以通过 `npm install <包名>` 来安装新的 node 包了。

关于 *package.json* 额外多说一句。通过这个文件你可以在不共享本地包的情况下把项目共享给其他的开发人员。因为这个文件中已经有了所有 node 包的引用，这些包又被叫做依赖（dependency），每个人都可以在不包含所有依赖的情况下拷贝你的项目，因为 *package.json* 中列出了所有的依赖。只需要通过一个简单的 `npm install` 命令就可以获取所有依赖然后安装到 *node_modules/* 文件夹下面。

另外还有一个需要提及的 npm 命令：

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

```
npm install --save-dev <package>
```

`--save-dev` 标记表示该 node 包只是用作开发环境的一部分，并不会被作为你产品代码的一部分发布。哪种 node 包适用这个场景呢？设想你需要一些 node 包辅助测试你的应用，然后需要通过 npm 来安装这些包，但是不希

望他们混入产品代码里面。测试过程应该只会发生在开发阶段，而不是在线上部署运行的时候。因为那个时候已经用不到测试代码了，你的应用应该已经被测试完而且可以被你的用户使用了。这可能是你唯一的使用 `--save-dev` 的场景。

你可能会遇到更多的 npm 命令，但目前来说这些已经足够了。

练习

- 搭建一个简易的 npm 项目
- 使用 `mkdir <文件夹名>` 创建一个新的文件夹
- 通过 `cd <文件夹名>` 进入该文件夹
- 运行 `npm init -y` 或者 `npm init` 来初始化一个 npm 项目
- 安装一个本地包，比如 React：`npm install react`
- 仔细看一下 *package.json* 文件和 *node_modules/* 文件夹里面的内容
- 找找看有没有什么办法把 *react* 从项目中卸载掉
- 阅读更多关于 npm 的内容。

安装 React

有很多种方式可以让你创建一个 React 应用。

第一种方式是通过 CDN。听起来可能会有点复杂。CDN 指的是内容分发网络（content delivery network）。一些公司会把他们公开的文件放置在 CDN 上供人们访问。其中可以是像 React 这样的库，毕竟整个打包的 React 库就只有一个 *react.js* 文件，可以直接托管在任何地方然后引入到你的应用中。

怎样使用 CDN 引入 React 呢？你可以通过在 HTML 中内嵌一个指向该 CDN 的 url 的 `<script>` 标签。比如对于 React，你需要这两个文件（库）：*react* 和 *react-dom*。

{title="Code Playground",lang="javascript"}

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.de
<script crossorigin src="https://unpkg.com/react-dom@16/umd/reac
```

但是如果你有 npm 来安装 React 的话为什么还需要 CDN 呢？

如果你的项目有一个 *package.json* 文件，你可以通过命令行来安装 *react* 和 *react-dom*。不过这样做的条件是你已经通过 `npm init -y` 命令初始化了一个 npm 项目和 `package.json` 文件。你可以通过一条命令安装多个 node 包：

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

```
npm install react react-dom
```

这种方式通常适用于那些使用 npm 做包管理的项目。

不过这还不够。你还可能需要设置 Babel 来让你的项目支持 JSX（React 的专用语法）和 JavaScript ES6。Babel 把你的 JavaScript ES6 和 JSX 代码转译（transpile）成浏览器可以支持的版本，毕竟并不是所有的浏览器都支持这些高级语法。这一步设置包含一堆的配置和工具，对于一个新手来说可能会感觉到不小的压力。

由于这个原因，Facebook 引入了 *create-react-app* 作为零配置的 React 解决方案。下一章会讲解如何使用这个工具来搭建一个 React 应用。

练习:

- 阅读更多安装 React 的内容。

零配置搭建 React 应用

在本书中，你将使用 create-react-app 来创建应用。在得到广泛支持的情况下，Facebook在2016年创建了这样一个零配置的 React 初始化套件。96% 的人向初学者推荐了它。使用 *create-react-app*，各种工具和配置都会在后台集成，而开发人员只需要专注于实现就好。

首先你需要把它安装成 node 的全局包。你就可以在命令行创建和初始化 React 应用了。

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

```
npm install -g create-react-app
```

你可以通过检查 *create-react-app* 的版本来验证是否安装成功。

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

```
create-react-app --version
*v1.4.1
```

现在你就可以创建你的第一个 React 应用了。我把它命名为 *hackernews*，你也可以选择另一个名字。创建过程需要花费一段时间。创建成功后，直接切换到该文件夹。

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

```
create-react-app hackernews
cd hackernews
```

现在你可以在你的编辑器里面打开这个项目。类似下面的文件结构（或者是略有不同，根据你的 *create-react-app* 的版本的不同）会呈现在你面前：

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

```
hackernews/
  README.md
  node_modules/
  package.json
  .gitignore
  public/
    favicon.ico
    index.html
    manifest.json
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    registerServiceWorker.js
```

简单划分一下这些文件和文件夹，目前你并不需要做一个很全面的了解：

- **README.md:** 后缀名为 .md 表示这是一个 markdown 文件。Markdown 是一个用纯文本创建格式化文档的标记语言。很多源代码项目包含一个 *README.md* 文件，其中包含了这个项目的一些基本的指令

和介绍。当你把项目发布到一些平台后，比如 GitHub，当在这个平台访问该项目的时候就会直接看到 *README.md* 里的内容。因为你使用的是 *create-react-app*，所以你的 *README.md* 文件会跟 create-react-app 官方 GitHub 仓库的内容一样。

- **node_modules/:** 这个文件夹包含了所有通过 npm 安装的 node 包。在你使用了 *create-react-app* 之后，就有一堆 node 包已经被安装了。通常你不需要特别去关心这个文件夹里面的内容，只需要在命令行用 npm 安装或者卸载 node 包就可以。
- **package.json:** 这个文件包含了 node 包依赖列表和一些其他的项目配置。
- **.gitignore:** 这个文件包含了所有不应该添加到 git 仓库（repository）中的文件和文件夹。他们应该只能存活在你本地项目文件夹中。一个典型的例子是 *node_modules/*，把 *package.json* 共享给你的伙伴们就足够他们获取和安装所有的依赖了，没必要把整个依赖打包共享给他们。
- **public/:** 这个文件夹包含了所有你的项目构建出的产品文件。最终所有你写在 *src/* 文件夹里面的代码都会在项目构建的时候被打包放在 public 文件夹下。
- **manifest.json** 和 **registerServiceWorker.js:** 在这个阶段不用担心这些文件用来干什么，我们不会在这个项目中用到他们。

总之，你不需要去修改提到的这些文件和文件夹。所有你需要的文件都在 *src/* 文件夹中。首要关注的是实现 React 组件的 *src/App.js* 文件。它主要用于实现你的应用，不过之后你可能会把你的组件分离到多个文件中，其中每个文件来维护一个或者多个特定的组件。

除此之外，你会发现还有一个用于测试的 *src/App.test.js* 和作为 React 世界的入口的 *src/index.js*。在后面的章节中你会逐渐熟悉这两个文件。另外，还有控制你项目整体样式和组件样式的 *src/index.css* 文件和 *src/App.css* 文件，他们都被设置成了默认的样式。

create-react-app 创建的是一个 npm 项目。你可以通过 npm 来给你的项目安装和卸载 node 包。另外它还附带了下面几个 npm 脚本：

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

```
// 在 http://localhost:3000 启动应用
npm start
```

```
// 运行所有测试
npm test

// 构建项目的产品文件
npm run build
```

这些脚本存在 *package.json* 中，现在这样一个 React 样板项目就创建完成了。接下来可以通过练习来在浏览器中运行刚刚创建的应用。

练习：

- 通过 `npm start` 运行并在浏览器中访问你的应用（你可以通过 `Control + C` 来退出此命令）
- 运行交互式的 `npm test` 脚本
- 运行 `npm run build` 并确认项目中出现了 *build/* 文件夹（你可以在之后把它删除掉；注意 build 文件夹可以用来部署你的应用）
- 熟悉一下项目的文件结构
- 熟悉一下不同文件的具体内容
- 阅读更多关于 npm 和 create-react-app 的内容

JSX 简介

现在你可以开始了解 JSX 了。这是 React 特有的语法。前面提到过，*create-react-app* 已经创建了一个样板项目给你。所有的文件都会按照默认实现提供。我们现在来深入探索一下项目的源代码。首先你需要接触的是 *src/App.js* 文件。

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

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
```

```
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save
        </p>
      </div>
    );
  }
}

export default App;
```

不要被 import/export 语句和类（class）声明给绕晕了，这些都是 JavaScript ES6 的特性，我们将在后面的章节讲解。

在这个文件中有一个叫做 App 的 React ES6 类组件（class component）。这是一个组件声明。基本上，在你声明了一个组件以后，你可以在你项目的任何地方使用它。它可以创建一个**组件**的**实例**（instance），或者说，可以实例化这个组件。

`render()` 方法包含了它所返回的元素（element）。元素是组件的构成部分。理解清楚组件、实例和元素之间的区别是很有帮助的。

不久之后你将看到 App 组件会在什么地方实例化。否则你不会在浏览器中看到它渲染的结果。现在你看到的 App 组件只是它的声明，而不是在使用它。你可以通过在 JSX 代码的某些地方通过 `<App />` 来实例化它。

render 方法中的代码看起来和 HTML 非常像，这就是 JSX。JSX 允许你在 JavaScript 中混入 HTML 结构。如果你习惯于 HTML 和 JavaScript 分离的话，可能会对这种做法感到费解，但其实 JSX 非常强大。所以最好从基本的 HTML 来写你的 JSX 代码。那么我们先删除掉文件中所有的不必要的内容。

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

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <h2>Welcome to the Road to learn React</h2>
      </div>
    );
  }
```

```
}

export default App;
```

现在在你的 `render()` 方法里面仅仅返回了 HTML，不再有 JavaScript。我们来把 "Welcome to the Road to learn React" 定义成一个变量。你可以使用花括号（curly braces）把变量引入到 JSX 中。

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

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
# leanpub-start-insert
    var helloWorld = 'Welcome to the Road to learn React';
# leanpub-end-insert
    return (
      <div className="App">
# leanpub-start-insert
        <h2>{helloWorld}</h2>
# leanpub-end-insert
      </div>
    );
  }
}

export default App;
```

现在再次运行 `npm start`，你就能看到效果了。

另外，你应该注意到了代码中的 `className` 属性。它其实就是标准 HTML 中的 `class` 属性。但是因为一些技术上的原因，JSX 把一些 HTML 的内部属性替换成了不同的。你可以在 React 文档支持的 HTML 属性中找到相关的内容。他们都遵守驼峰命名法（camelCase convention）。在接下来的学习过程中，你将会遇到更多的 JSX 专有的属性。

练习：

- 定义更多的变量然后再 JSX 中引用并渲染（render）它们
- 使用一个复杂对象来表示一个拥有姓氏（first name）和名字（last name）的用户

- 把该用户的属性放到 JSX 中渲染
- 阅读更多关于 JSX 的内容
- 阅读更多 React 组件、元素和实例的内容

ES6 const 和 let

我猜你已经注意到了，我们在前面的例子中使用的是关键字 `var` 来声明变量 `helloWorld` 的。JavaScript ES6中引入了另外两个声明变量的关键字： `const` 和 `let`。在JavaScript ES6中，你将会很少能看到 `var` 了。

被 `const` 声明的变量不能被重新赋值或重新声明。换句话说，它将不能再被改变。你可以使用它创建不可变数据结构，一旦数据结构被定义好，你就不能再改变它了。

{title="Code Playground",lang="javascript"}

```
// 这种写法是不可行的
const helloWorld = 'Welcome to the Road to learn React';
helloWorld = 'Bye Bye React';
```

被关键字 `let` 声明的变量可以被改变。

{title="Code Playground",lang="javascript"}

```
// 这种写法是可行的
let helloWorld = 'Welcome to the Road to learn React';
helloWorld = 'Bye Bye React';
```

当一个变量需要被重新赋值的话，你应该使用 `let` 去声明它。

然而，你必须小心地使用 `const` 。使用 `const` 声明的变量不能被改变，但是如果这个变量是数组或者对象的话，它里面持有的内容可以被更新。它里面持有的内容不是不可改变的。

{title="Code Playground",lang="javascript"}

```
// allowed
const helloWorld = {
  text: 'Welcome to the Road to learn React'
};
helloWorld.text = 'Bye Bye React';
```

但是，□□不同的□声明方式应该在什么时候使用呢？有很多的选择。我的建议是在任何你可以使用 `const` 的时候使用它。这表示尽管对象和数组的内容是可以被修改的，你仍希望保持该数据结构不可变。而如果你想要改变你的变量，就使用 `let` 去□声明它。

□React和它的生态是拥抱不可变的。这就是为什么 `const` □应该是你定义一个变量时的默认选择。当然，一个复杂的□对象中的内容还是可能会被改变，□请当心这种改变。

在你的应用中，你应该用 `const` 来代替 `var`。

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

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
# leanpub-start-insert
    const helloWorld = 'Welcome to the Road to learn React';
# leanpub-end-insert
    return (
      <div className="App">
        <h2>{helloWorld}</h2>
      </div>
    );
  }
}

export default App;
```

练习:

- 阅读更多关于 ES6 const 的内容
- 阅读更多关于 ES6 let 的内容
- □研究更多关于不可变数据结构的内容
- 在通常情况下，为什么他们是有意义的

- 为什么他们会被React和它的生态使用

ReactDOM

在你学习这个 App 组件之前，你可能想知道它被用在了什么地方。它在你的React世界的入口文件 *src/index.js* 中

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

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

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

简单来说，`ReactDOM.render()` 会使用你的 JSX 来替换你的HTML中的一个 DOM 节点。这样你就可以很容易地把 React 集成到每一个其他的应用中。`ReactDOM.render()` 可以在你的应用中被多次使用。你可以在多个地方使用它来加载简单的 JSX 语法、单个 React 组件、多个 React 组件或者整个应用。但是在一个纯 React 的应用中，你只会使用一次用来加载你的整个组件树。

`ReactDOM.render()` 有两个传入参数。第一个是准备渲染的 JSX。第二个参数指定了React应用在你的HTML中的放置的位置。这个位置是一个 `id='root'` 的元素。你可以在文件 *public/index.html* 中找到这个id属性。

In the implementation `ReactDOM.render()` already takes your App component. However, it would be fine to pass simpler JSX as long as it is JSX. It doesn't have to be an instantiation of a component.

在实现中，`ReactDOM.render()` 总会很好地渲染你的 App 组件。你可以将一个简单的 JSX 直接用 JSX 的方式传入，而不用必须传入一个组件的实例。

{title="Code Playground",lang=javascript}

```
ReactDOM.render(
  <h1>Hello React World</h1>,
```

```
  document.getElementById('root')
);
```

练习:

- 打开 *public/index.html* 文件，找到 React 应用并放置在你的HTML的位置
- 查看更多关于 元素渲染 的内容

模块热替换

作为一个开发者，你可以在 *src/index.js* 中做一件事情来提高你的开发体验。但是这件事情是可选的，不要让它在你刚开始学习 React 的事情占用你过多的时间。

用 *create-react-app* 创建的项目有一个优点，那就是可以让你在更改源代码的时候浏览器自动刷新页面。试试改变 *src/App.js* 文件中的变量 `helloWorld` ，修改过后浏览器应该就会刷新页面。但有一个更好的方式来实现这个功能。

模块热替换（HMR）是一个帮助你在浏览器中重新加载应用的工具，并且无需再让浏览器刷新页面。你可以在 *create-react-app* 中很容易地开启这个工具：在你 React 的入口文件 *src/index.js* 中，添加一些配置代码。

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

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

ReactDOM.render(
  <App />,
  document.getElementById('root')
);

# leanpub-start-insert
if (module.hot) {
  module.hot.accept();
}
# leanpub-end-insert
```

配置完成。接下来再尝试在你的 src/App.js 文件中更改一下变量 `helloWorld`，浏览器应该不会刷新页面，但是应用还是会重新加载并且显示正确的输出。HMR 带来很多的优点：

设想你正在使用 `console.log()` 调试你的代码。由于浏览器不再会刷新页面，所以即使你更改了你的代码，这些调试信息也会完整地保持在你的开发控制台中。这让调试变得很方便。

在一个正在开发的应用中，刷新页面将会降低你的生产效率：你必须得等待页面加载完毕。一个大的应用可能会花很多秒钟才能刷新完页面。使用HMR可以避免这个缺点。

使用 HMR 最大的好处是你可以保持应用的状态。设想你的应用中有一个对话框，其中包含很多步骤，而现在你正在第三步当中，基本上这就特别奇怪。如果没有 HMR 的话，当你更改源代码的时候你的浏览器将会刷新整个页面，你就不得不再次打开这个对话框，并且从步骤一开始导航到步骤三。而如果你使用 HMR 的话，你的对话框将会始终保持打开在步骤三的状态。尽管你的源代码改变了，但是应用的状态也会被保持。应用本身会被重新加载，而不是页面被重新加载。

练习：

- 改变几次你的 *src/App.js* 中的源代码，来观察 HMR 的工作方式
- 观看 Dan Abramov 的视频 Live React: Hot Reloading with Time Travel 的前十分钟

JSX 中的复杂 Javascript

让我们回到你的 App 组件中。到目前为止你在你的 JSX 中渲染了一些简单的变量。现在你可以开始渲染一个列表了。这个列表一开始可以是一些示例数据，但是以后你可以从一个外部 API 中获取数据。这会让人更加兴奋。

首先你需要定义一个列表。

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

```
import React, { Component } from 'react';
import './App.css';

# leanpub-start-insert
const list = [
```

```
  {
    title: 'React',
    url: 'https://facebook.github.io/react/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  },
  {
    title: 'Redux',
    url: 'https://github.com/reactjs/redux',
    author: 'Dan Abramov, Andrew Clark',
    num_comments: 2,
    points: 5,
    objectID: 1,
  },
];
# leanpub-end-insert

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

这个示例数据反应的是我们准备用 API 获取的数据。列表中的每一个成员都有标题、链接和作者信息。另外它还包含有标识符、分数（表示这个文章的流行程度）和评论的数量。

现在你可以在你的 JSX 中使用 JavaScript 内置的 `map` 函数。这个函数可以让你遍历你的列表来显示其中的成员。同样的，你需要用花括号把 JavaScript 包含在你的 JSX 中。

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

```
class App extends Component {
  render() {
    return (
      <div className="App">
# leanpub-start-insert
        {list.map(function(item) {
          return <div>{item.title}</div>;
        })}
# leanpub-end-insert
      </div>
    );
  }
}
```

```
export default App;
```

在 JSX 中使用 HTML 中的 JavaScript 是很强大的。通常情况下你可以用 `map` 来将一个列表转换成另一个列表。在这个例子中，你使用 `map` 函数将一个列表转换成一组 HTML 元素。

到目前为止，每个成员只有 `title` 会被显示。让我们显示一些它们的其他属性。

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

```
class App extends Component {
  render() {
    return (
      <div className="App">
# leanpub-start-insert
        {list.map(function(item) {
          return (
            <div>
              <span>
                <a href={item.url}>{item.title}</a>
              </span>
              <span>{item.author}</span>
              <span>{item.num_comments}</span>
              <span>{item.points}</span>
            </div>
          );
        })}
# leanpub-end-insert
      </div>
    );
  }
}

export default App;
```

你可以看到 `map` 函数是如何简单地内联到你的 JSX 中的。每一个成员属性会被显示成一个 `<span>` 标签。此外，`url` 属性在另一个标签中被用作为 `href` 属性。

React 会帮你完成所有的工作然后逐一显示每个成员。但你应该在 React 中添加一个辅助属性，借此发挥出它的潜能以提高性能。你需要给列表的每一个成员加上一个关键字（key）属性。这样的话 React 就可以在列表发生变

化的时候识别其中成员的添加、更改和删除的状态。这个示例数据中已经有一个标识符了。

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

```
{list.map(function(item) {
  return (
# leanpub-start-insert
    <div key={item.objectID}>
# leanpub-end-insert
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
    </div>
  );
})}
```

你应该确保这个关键字属性是一个稳定的标识符。不要错误地使用列表成员在数组的索引作为关键字。列表成员的索引是完全不稳定的。在下面的这个例子中，当列表的排序改变了之后，React 将很难正确地识别这些成员。

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

```
// don't do this
{list.map(function(item, key) {
  return (
    <div key={key}>
      ...
    </div>
  );
})}
```

你现在可以显示列表的所有成员了。你可以开启你的应用，打开浏览器然后查看这些显示出的列表成员。

练习:

- 查看更过关于 React 列表和关键字 的内容
- 简要重述 JavaScript 中标准内建数组函数
- 在 JSX 中使用更多的 JavaScript 表达式

ES6 箭头函数

JavaScript ES6 引入了箭头函数。箭头函数表达式比普通的函数表达式更加简洁。

{title="Code Playground",lang="javascript"}

```
// function expression
function () { ... }

// arrow function expression
() => { ... }
```

但是你需要注意它的一些功能性。其中之一就是关于 `this` 对象的不同行为。一个普通的函数表达式总会定义它自己的 `this` 对象。但是箭头函数表达式仍然会使用包含它的语境下的 `this` 对象。不要被这种箭头函数的 `this` 对象困惑了。

关于箭头函数的括号还有一个值得关注的点。如果函数只有一个参数，你就可以移除掉参数的括号，但是如果有多个参数，你就必须保留这个括号。

{title="Code Playground",lang="javascript"}

```
// allowed
item => { ... }

// allowed
(item) => { ... }

// not allowed
item, key => { ... }

// allowed
(item, key) => { ... }
```

不管怎样，让我们再看一下 `map` 函数。你可以用 ES6 的箭头函数更加简洁地把它写出来。

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

```
# leanpub-start-insert
{list.map(item => {
# leanpub-end-insert
  return (
    <div key={item.objectID}>
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
    </div>
  );
})}
```

此外，在 ES6 的箭头函数中，你可以用简洁函数体来替换块状函数体（用花括号包含的内容），简洁函数体的返回不用显示声明。这样你就可以移除掉函数的 return 表达式。在这本书中这种表达式将会被更多地使用，所以你要确保能够在使用箭头函数的时候要明白块状函数体和简洁函数体的区别。

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

```
# leanpub-start-insert
{list.map(item =>
# leanpub-end-insert
  <div key={item.objectID}>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
  </div>
# leanpub-start-insert
)}
# leanpub-end-insert
```

现在你的 JSX 变得更加简洁和可读了。函数声明表达式、花括号和返回声明都被省略了。开发者就可以更加专注在实现细节上。

练习:

- 阅读更多关于 ES6 箭头函数 的内容

ES6 类

JavaScript ES6 引入了类的概念。类通常在面向对象编程语言中被使用。JavaScript 的编程范式在过去和现在都是非常灵活的。你可以根据使用情况一边使用函数式编程一边使用面向对象编程。

尽管 React 为了例如不可变数据结构等的特性而拥抱函数式编程，但是它还是使用类来声明组件。这些组件被称为 ES6 类组件。React 混合使用了两种编程范式中的有益的部分。

作为 JavaScript ES6 类的例子，让我们先不管组件，思考以下这个 Developer 类。

{title="Code Playground",lang="javascript"}

```
class Developer {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  getName() {
    return this.firstname + ' ' + this.lastname;
  }
}
```

类都有一个用来实例化自己的构造函数。这个构造函数可以用来传入参数来赋给类的实例。此外，类可以定义函数。因为这个函数被关联给了类，所以它被称为方法。通常它被当称为类的方法。

这个 Developer 类只有类的声明。你可以使用它来创建多个类的示例。它和 ES6 类组件很类似，都有声明，但是你需要在别的地方实例化这个类。

让我们看看如何实例化这个类，以及如何使用它的方法。

{title="Code Playground",lang="javascript"}

```
const robin = new Developer('Robin', 'Wieruch');
console.log(robin.getName());
// output: Robin Wieruch
```

React 使用 JavaScript ES6 类来实现 ES6 类组件。你已经使用过一个 ES6 类组件了。

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

```
import React, { Component } from 'react';

...

class App extends Component {
  render() {
    ...
  }
}
```

这个 App 类继承自 `Component`。简单来说，你可以声明你的 App 组件，但是这个组件需要继承自另一个组件。继承是什么意思？在一个面向对象编程的语言中，你需要遵循继承原则。它可以把功能从一个类传递到另一个类。

这个 App 类就从 Component 类中继承了它的功能。这个 Component 类是从一个基本 ES6 类中继承来的 ES6 组件类。它有一个 React 组件所需要的所有功能。渲染（render）方法就是其中你可以使用的一个功能。之后你可以学到更多其他组件类的方法。

这个 `Component` 类封装了所有 React 类需要的实现细节。它使得开发者们可以在 React 中使用类来创建组件。

React `Component` 类暴露出来的方法都是公共的接口。这些方法中有一个方法必须被重写，其他的则不一定要被重写。你会在以后的讲述生命周期的章节中学到它们。这个 `render()` 方法是必须被重写的方法，因为它定义了一个 React 组件的输出。它必须被定义。

现在你已经知道了 JavaScript ES6 类的基本内容，以及它们是怎么在 React 中被继承为组件的。在本书描述 React 生命周期方法的地方，你将会学到关于更多 Component 的方法。

练习:

- 阅读更多关于 ES6 类 的内容

{pagebreak}

你已经学会如何开始一个你自己的 React 应用了！让我们回顾一下这一章的内容：

- React
- 使用 create-react-app 创建一个 React 应用
- JSX 混合使用了 HTML 和 JavaScript 在React 组件的方法中定义它的输出
- React 中，组件、示例和元素是不同的概念
- `ReactDOM.render()` 是 React 应用连接 DOM 的入口方法
- JavaScript 内建函数可以在 JSX 中使用
    - map 可以被用来把列表成员渲染成 HTML 的元素

ES6

- 根据不同的使用场景，选择用 `const` 和 `let` 来声明变量
- 在 React 应用中尽量使用 const 来声明变量
- 箭头函数可以用来是你的函数变得更简洁
- 在 React 中，通过继承类的方式来声明组件

现在我们值得休息一下。巩固下这章的内容，你可以试验一下目前为止所编写的代码。

你可以在官方代码库中找到源代码。