by robin wieruch

the Road to React

《React 学习之道》The Road to learn React (简体中文版)

通往 React 实战大师之旅：掌握 React 最简单，且最实用的教程

Robin Wieruch, JimmyLv 和 Jiahao Li

這本書的網址是 http://leanpub.com/the-road-to-learn-react-chinese

此版本發布於 2020-12-13

在 Tweet 上分享此書！

請在 Twitter 上面替作者 Robin Wieruch, JimmyLv 和 Jiahao Li 宣傳!

對此書所建議的 tweet 是:

I am going to learn #ReactJs with The Road to learn React by @rwieruch Join me on my journey https://roadtoreact.com

針對此書的建議 hashtag 是 #ReactJs.

若想知道其他人對這本書的看法，可點此連結來搜尋 Twitter 上的 hashtag:

#ReactJs

Contents

前言

《通向 React 之路》旨在讲授 React 基础知识。你将会用 React 搭建一个真实的应用程序，无需其他复杂的工具。从项目初始化到服务器部署，我会为你详细讲解其中的每一步，并在每一章都附有参考资料和练习。读完此书后你将能够搭建自己的 React 应用。我和社区会持续维护和更新这些资料。

在深入到广阔的 React 生态系统之前，应该先打稳基础。我会用更关注 React 自身的方式来讲解它的概念，尽量减少工具和外部状态管理的使用，从而提供更多关于 React 自身的知识。我会通过一个真实的 React 应用来讲解这些通用概念、模式和最佳实践。

大体上你将会学到如何从头开始写一个 React 应用，包含分页、客户端和服务端搜索，以及一些高级的交互方式，例如排序等等。希望在阅读本书的过程中能让你感受到我对 React 和 JavaScript 的热爱，让它们伴你一同走进 React 的世界。

关于作者

我是一名来自德国的软件和互联网工程师，致力于学习和教授 JavaScript 编程。在获得计算机科学硕士学位后，我没有停止过自我学习。在初创公司的那段日子里，无论是工作还是业余时间，我整天都在和 JavaScript 打交道。这段经历使我积攒了大量经验，也最终引领我走向了 JavaScript 传道者之路。

有几年，我和一群出类拔萃的工程师一起在一家叫做 Small Improvements 的公司工作，开发大型应用程序。公司经营的是一款 SaaS 产品，能够让客户提供反馈给商家。产品的前端是用的 JavaScript，后端是 Java。产品最初版本的前端使用 Java 的 Wicket 框架和 jQuery。没过多久，第一代 SPA（单页面应用）开始流行起来，公司决定把前端迁移到 Angular 1.x。然而在和 Angular 纠缠了两年多以后，我们渐渐发现它并不是管理复杂多变的应用状态的最佳方案，转而投入了 React 和 Redux 的怀抱，这使得产品大获成功。

我在公司时，会定期地在我的网站上写一些关于 Web 开发的文章。感谢那时的读者给了我很多很棒的反馈，让我的写作和教学水平得到了提升。就这样日积月累，我渐渐习得了教练技巧。我发现我早期的文章倾向于包含过量的信息，让学生不堪重负，我慢慢地学会了每次只关注一个主题。

如今，我作为自由职业者在继续软件开发工程师和教育者的工作。通过设定明确的目标和短期反馈环，我能看到学生们可以得到快速成长，我的成就感也来源于此。我的网站[1]上有更多关于我个人的信息，以及如何支持或和我一起工作的方式。

[1] (https://www.robinwieruch.de/about)

FAQ

如何获取更新?

我通过两种方式发布内容更新：通过邮件订阅更新[2]或者在 Twitter 上关注我[3]。无论是哪种方式，我都竭力保证内容的质量。当收到本书内容更新的通知时，你就可以从我的网站上下载新的版本。

学习资料是最新的吗?

一般编程相关的书籍在它们发布之后很容易过时。但是本书是我自己出版的，所以当书中相关的内容有新发布的时候，我可以及时地更新版本。

如果我在 **Amazon** 上已经购买了此书，可以索取电子版吗?

如果你已经在 Amazon 上购买此书，你可能已经看到我的网站上也上架了。因为我会用 Amazon 变现一部分通常来说免费的内容，在此我衷心感谢你的支持，并邀请你注册我的网站[4]。当创建好账号之后，请把你的邮箱和 Amazon 的购买收据通过邮件发给我，我会为你在网站上解锁相应内容。当有了账号之后，你就可以一直获得最新的内容。

如果你购买了该书的纸质版，请尽可能地物尽其用。我特意把书的尺寸做得大一些，以便于显示更多代码，也方便你在阅读的同时做笔记。

如果阅读本书时我需要帮助怎么办?

我们有一个 Slack 聊天组[5]，专为自学本书的人而创建。你可以加入这个群组来获取帮助，或者帮助他人，因为在帮助他人的同时你也可以加深自己的理解。

我可以为内容改进做贡献吗?

如果你有任何反馈，请不要犹豫发送邮件给我，我会认真考虑你的建议。但如果是 bug 追踪或者疑难解答类的邮件可能就不要期望有回复了，Slack 群组才是用来做这个的。

我可以怎样支持该项目?

如果你发现我的课程很有帮助并愿意贡献内容，请在我网站的关于页面[6]上找到更多关于如何提供支持的信息。当然如果读者们能帮我多多宣传我的书如何帮助你们成长，也是非常有帮助的，这可以让更多想提升 web 开发技能的人接触到我。用以上任何方法提供支持都可以帮助我更好的打造有深度的课程和提供免费的资料。

是什么驱使你写这本书?

我一直都想讲授这个话题。我经常发现网上的一些资料没有更新，又或只更新其中的一小部分。有时人们难以找到持续更新的自学材料。我希望能提供连贯的、持续更新的学习体验。并且我也希望能够通过这个项目给予那些不幸的人以帮助，为他们提供免费内容或是其他帮助[7]。

[2] https://www.getrevue.co/profile/rwieruch
[3] https://twitter.com/rwieruch
[4] https://www.robinwieruch.de/
[5] https://slack-the-road-to-learn-react.wieruch.com/
[6] https://www.robinwieruch.de/about/
[7] https://www.robinwieruch.de/giving-back-by-learning-react/

本书的面向人群

- JavaScript 初学者，了解基本的 JS、CSS 和 HTML 知识。如果你刚开始接触 Web 开发，对 JS、CSS 和 HTML 有基本概念，那么本书可以提供所有学习 React 需要的内容。如果你觉得自己的 JavaScript 水平还不够，那可以在继续本书之前了解更多关于 JavaScript 的知识。当然，你也会在本书的参考中获得更多基础知识的相关资料。
- 从 jQuery 时代来的 JavaScript 老手：如果你在 jQuery、MooTools 和 Dojo 的年代大量使用过 JavaScript，可能会和当下的 JavaScript 趋势有点难以接轨。然而它们底层其实从来没有变过，依然是 JavaScript 和 HTML，所以本书应该可以帮你用正确的姿势开始学习 React。
- JavaScript 爱好者，已有其他现代单页面应用框架的经验：如果你用过 Angular 或者 Vue，虽然 React 应用写起来会很不一样，但这些框架的基础构件都是 JavaScript 和 HTML。一旦适应了用 React 的方式去思考问题，这对你来说就不是什么问题了。
- 掌握其他编程语言的人：你可能在编程原理方面比其他人更加熟练。当习惯了 JavaScript 和 HTML 之后，你就可以愉快地和我一起学习 React 了。
- 如果你的专业是在设计方向，用户交互设计是体验设计，也无需担心。你可能已经对 HTML 和 CSS 很熟悉了，只需要再学习一些 JavaScript 的基础知识，就能够阅读本书了。现如今的 UI/UX 和实现方式的关系越来越密切，了解 React 相关的知识会让你的工作获益匪浅，本书将很好地带你了解它的工作原理。
- 如果你是研发团队的产品负责人或是产品经理，本书会为你详细剖析 React 应用中的所有不可或缺的部分。每一章节都讲解了一个 React 概念、模式、技术，并用其来增加新功能，或是提升整体架构。这是一个全面的 React 教程。

React 基础

在第一部分的学习中，我们会通过创建第一个 React 项目，介绍 React 的基础知识。然后，我们将通过实现真实的功能来探索 React 的新内容，就像开发真实的 Web 应用程序一样。最后，我们将拥有一个可正常使用的 React 应用程序，该应用程序具备客户端和服务器端搜索，获取远程数据以及高级状态管理等功能。

你好 React

单页面应用（SPA[8]）在第一代 SPA 框架如 Angular（由 Google 开发）、Ember、Knockout 和 Backbone 中变得越来越受欢迎。使用这些框架使构建 Web 应用更加简单，这些框架超越了普通的 JavaScript 和 jQuery。Facebook 于 2013 年发布了 React，这是另一种 SPA 的解决方案。这些框架全部使用 JavaScript 创建整个 Web 应用。

让我们暂时回到 SPA 出现之前：在过去，网站和 Web 应用是在服务器渲染的。用户在浏览器中访问 URL，并从 Web 服务器请求 HTML 文件及其所有相关的 HTML、CSS 和 JavaScript 文件。经过一段网络延迟后，用户会在浏览器（客户端）中看到渲染后的 HTML，随后开始与其进行交互。每进行一次页面跳转（也就是访问另一个 URL）就会再次启动这一系列的过程。在过去，本质上所有关键的步骤都由服务器完成，而客户端仅通过展示一个个渲染后的页面扮演着一个小角色。虽然使用标准的 HTML 和 CSS 构建应用程序，但是只使用了少量的 JavaScript 使实现交互（如下拉菜单）或高级样式（如定位提示工具）成为可能。jQuery 就是实现这类功能最受欢迎的 JavaScript 库。

相反，现代的 JavaScript 将焦点从服务器转移到了客户端。举个最极端的例子：用户访问 URL 并请求一个 最小的 *HTML* 文件和一个 相关的 *JavaScript* 文件。经过一段网络延迟后，用户会在浏览器中看到 由 *JavaScript* 渲染的 *HTML* 并开始与其进行交互。每一次额外的页面跳转都 不会从 Web 服务器请求更多的文件，而是会使用最初请求的 JavaScript 来渲染新的页面。同样，用户的其他所有交互也都在客户端上进行处理。在现代的版本中，服务器主要通过一个小体积的 HTML 文件跨线提供 JavaScript。然后，这个 HTML 文件在客户端执行所有相关的 JavaScript，以使用 HTML 和 JavaScript 渲染整个应用程序进行交互。

在其他 SPA 解决方案中，React 使这其成为可能。本质上来说，SPA 是创建整个应用程序的一个 JavaScript 主体，这些 JavaScript 整齐的组织在文件夹和文件中，而 SPA 框架提供了构建它的所有工具。当用户访问您的 Web 应用的 URL 时，JavaScript 应用程序会通过网络一次性发送至您的浏览器。从此，React 和其他 SPA 框架就接管了浏览器中渲染和处理用户交互的任务。

随着 React 的兴起，组件的概念开始流行起来。每个组件都使用 HTML、CSS 和 JavaScript 定义外观。一旦定义好一个组件后，它就可以用在组件结构中来创建整个应用。尽管 React 作为一个库非常重视组件的概念，但周围的生态系统却使其成为了一个非常灵活的框架。React 拥有便捷的 API，还拥有稳定而繁荣的生态系统以及热情活跃的社区。我们非常欢迎您的加入:-)

练习

- 阅读关于为什么我从 Angular 换成 React[9]
- 阅读关于如何学习一个框架[10]
- 阅读关于如何学习 React[11]

[8] https://en.wikipedia.org/wiki/Single-page_application
[9] https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/
[10] https://www.robinwieruch.de/how-to-learn-framework/
[11] https://www.robinwieruch.de/learn-react-js

- 阅读关于为什么框架很重要[12]
- 浏览React 所需的 JavaScript 基础[13] – 不用太关注 React，看一下自己对 React 中用到的 JavaScript 特性了解多少

[12] https://www.robinwieruch.de/why-frameworks-matter
[13] https://www.robinwieruch.de/javascript-fundamentals-react-requirements

基本要求

阅读本书之前，你需要熟悉 Web 开发的基础知识，例如怎样使用 HTML，CSS 和 JavaScript。这也有助于理解 APIs[14]，因为它们将会覆盖全文。

编辑器和终端

我提供了设置指南[15]来帮助进行一般的 Web 开发。在这次学习过程中，你需要一个文本编辑器（例如 Sublime Text）和命令行工具（例如 iTerm）或者 IDE（例如 Visual Studio Code）。对于初学者我推荐使用 Visual Studio Code （又名 VS Code），它为高级编辑器提供了集成的命令行工具，是一种多合一的解决方案，并且它是 Web 开发人员的热门选择。

在整个学习过程中，我将使用术语命令行，这跟术语命令行工具，终端和集成终端同义。术语编辑器，文本编辑器和 *IDE* 同样适用，这取决于你决定使用哪种设置。

可选地，我推荐使用 GitHub 来管理本书中的练习项目，并且我提供了有关如何使用这些工具的简短指南[16]。Github 极为出色的版本控制，让你在犯了错误或者只是想采用一种更直接的方式来执行操作时，可以看到你所做的更改。这也是以后与他人共享代码的好方法。

如果你不想在本地计算机上设置编辑器/终端组合或 IDE，那么在线编辑器 CodeSandbox[17] 也是可行的选择。虽然 CodeSandbox 是在线共享代码的出色工具，但本机设置是学习用不同方法创建 Web 应用的更好工具。另外，如果你想专业地开发应用程序，则本地设置是必需的。

Node 和 NPM

在开始之前，我们需要安装 node and npm[18]。这两个工具都是用来管理你将会用到的各种库（node 包）的。这些 node 包可以是库，也可以是整个框架。我们将通过 npm（node 包管理器）安装外部 node 包。

你可以使用 `node --version` 命令在命令行中验证 node 和 npm 版本。如果你在终端中没有收到表示安装了哪个版本的输出，则需要安装 node 和 npm:

```
node --version
*vXX.YY.ZZ
npm --version
*vXX.YY.ZZ
```

如果你已经安装了 Node 和 npm，请确保你安装的是最新版本。如果你不熟悉 npm 或需要复习，我创建的 npm 速成教程[19]将带你快速入门。

[14] https://www.robinwieruch.de/what-is-an-api-javascript/
[15] https://www.robinwieruch.de/developer-setup/
[16] https://www.robinwieruch.de/git-essential-commands/
[17] https://codesandbox.io/
[18] https://nodejs.org/en/
[19] https://www.robinwieruch.de/npm-crash-course

建立一个 React 项目

在本书中，我们将使用 create-react-app[20] 来引导创建你的应用程序。这是 Facebook 在 2016 年针对 React 推出的零配置入门工具包，该工具集成的内容是 Facebook 官方的主观想法，96% 的 React 用户都会将它推荐给初学者[21]。使用 *create-react-app* 时，各种工具和配置都发生在后台，而开发者只需将重点放在应用程序的实现上。

安装了 Node 和 npm 之后，使用命令行在项目的专用文件夹中键入以下命令。我们将使用 *hacker-stories* 作为项目名，但你可以选择任何你喜欢的名称：

```
npx create-react-app hacker-stories
```

初始化完成后，进入生成的项目文件夹：

```
cd hacker-stories
```

现在我们可以在编辑器或 IDE 中打开我们的应用程序。对于 Visual Studio Code，你只需要在命令行输入 `code .` 即可。最后会展示以下的目录结构，或者会有一些变化，取决于不同的 *create-react-app* 版本：

```
hacker-stories/
--node_modules/
--public/
--src/
----App.css
----App.js
----App.test.js
----index.css
----index.js
----logo.svg
--.gitignore
--package-lock.json
--package.json
--README.md
```

这是一些最重要的文件夹和文件的细分：

- **README.md**：*.md* 拓展名表示该文件是一个 Markdown 文件。Markdown 是一种轻量级的标记语言，具有纯文本格式语法。许多源代码项目都带有 *README*.md 文件，该文件提供了有关该项目的说明和有用信息。当我们将项目推送到 GitHub 之类的平台

[20] https://github.com/facebook/create-react-app
[21] https://twitter.com/dan_abramov/status/806985854099062785

时，该 *README.md* 文件通常显示有关其存储库中包含的内容的信息。因为您使用 create-react-app 来创建项目，所以 *README.md* 应该和官方的 create-react-app GitHub 代码库[22]相同。

- **node_modules/**：此文件夹包含通过 npm 安装的所有 node 包。因为我们使用了 create-react-app，因此已经有两个 node 模块被安装了。我们不会去触碰这个文件夹，因为通常会在命令行使用 npm 来安装或卸载 node 包。
- **package.json**：此文件向您展示了依赖的 node 包列表和项目的其他配置信息。
- **package-lock.json**：此文件描述了分解之后所有 node 包的版本。我们一般不会触碰这个文件。
- **.gitignore**：此文件描述了使用 Git 时不应该被添加到你的 Git 代码库的所有文件和文件夹，因为这些文件和文件夹仅位于本地的项目中。*node_modules/* 目录就是一个例子。与其他人共享 *package.json* 文件就已经足够了，其他人可以在没有依赖文件夹的时候通过 `npm install` 来重新安装这些依赖。
- **public/**: 这个文件夹包含了一些开发时的文件，比如 *public/index.html*。在开发时这个索引文件将展示在 *localhost:3000* 或者其他托管的域名下。默认的设置会自动处理 *index.html* 与 *src/* 中其他 JavaScript 文件的关联关系。

首先，你需要的所有文件都位于 *src/* 文件夹下。我们主要关注用于实现 React 组件的 *src/App.js* 文件。它将会用来实现您的应用，但稍后您可能需要将组件拆分到多个文件中，每个文件中会维护一个或多个组件。

另外，您将找到一个用于测试的 *src/App.test.js* 文件，以及一个 *src/index.js* 文件作为 React 世界的入口点。在后续的部分中，您将会深入了解这两个文件。还有一个 *src/index.css* 和一个 *src/App.css* 分别设置了应用的常规样式和组件的样式，这是打开它们时的默认样式。您还会在以后修改它们。

在了解了 React 项目的文件夹和文件结构之后，让我们来看一下一些可用的命令来帮助我们开始。项目的所有特定的命令都能在 *package.json* 中的 *scripts* 属性下找到。它们看起来大概是这样：

```
{
  ...
 },
 "scripts": {
   "start": "react-scripts start",
   "build": "react-scripts build",
   "test": "react-scripts test",
   "eject": "react-scripts eject"
 },
 ...
}
```

这些命令脚本可以在 IDE 集成的终端，或者命令行中通过 `npm run <script>` 命令来执行。其中 `run` 可以在执行 `start` 和 `test` 命令脚本时省略。如下：

[22] https://github.com/facebook/create-react-app

```
# 在 http://localhost:3000 运行应用
npm start

# 运行测试
npm test

# 构建用于生产的应用程序
npm run build
```

上面介绍的 npm scripts 中有个叫 `eject` 的命令，它不应该在练习过程中使用。因为这是一个单向操作。一旦执行了 eject 操作，您将无法回滚。本质上仅当你对当前选择的配置不满意或者想更改某些内容时，才使用此命令来访问 create-react-app 中的所有所有构建工具和配置项。在这里，我们将保持默认的配置。

练习题

- 通过 React 的 create-react-app 文档[23] 以及 入门指南[24] 了解更多内容。
- 了解更多有关 create-react-app 中支持的 JavaScript 特性[25]。
- 了解更多有关 create-react-app 的目录结构[26].
  - 逐个了解你的 React 项目中的文件夹和文件。
- 了解更多有关 create-react-app 中可用的脚本[27].
  - 在命令行中使用 `npm start` 启动你的 React 应用并在浏览器中检查它。
    * 在命令行中通过按 `Control + C` 退出。
  - 运行 `npm test` 脚本。
  - 运行 `npm run build` 脚本并验证你的项目中是否会添加一个 *build/* 文件夹（之后你可以将其删除）。注意，以后可以通过 build 文件夹来部署你的应用[28]。
- 每当我们在学习过程中更改代码时，请确保在浏览器中检查输出以获得视觉上的反馈。

[23] https://github.com/facebook/create-react-app
[24] https://create-react-app.dev/docs/getting-started
[25] https://create-react-app.dev/docs/supported-browsers-features
[26] https://create-react-app.dev/docs/folder-structure
[27] https://create-react-app.dev/docs/available-scripts
[28] https://www.robinwieruch.de/deploy-applications-digital-ocean/

认识 React 组件

我们的第一个 React 组件在 *src/App.js* 文件中，它和下面的这个例子很相似。内容可能稍有不同，因为 create-react-app 有时候会更新默认的组件结构。

src/App.js

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          Href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

除非有另外的说明，否则该文件将会是贯穿这个教程的核心。我们先将组件简化为更轻便的版本，这样就可以在入门时不用关注那么多 create-react-app 生成的模板代码了。

src/App.js

```
import React from 'react';

function App() {
  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
}

export default App;
```

首先，这个 App 组件是一个 JavaScript 函数。一般称它为 函数组件，因为还有其他形式的 React 组件（稍后介绍 组件类型）。其次，截止目前 App 函数组件不接收任何参数（稍后介绍 **props**）。最后，App 组件返回的像 HTML 的代码称之为 JSX（稍后介绍 **JSX**）。

这个函数组件的实现细节和其他任何 JavaScript 函数一样。在整个 React 旅程中，你会在实践中体会到这一点。

src/App.js

```
import React from 'react';

function App() {
  // 在这儿做点什么

  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
}

export default App;
```

跟所有的 JavaScript 函数一样，在函数体内定义的变量会在每次运行函数的时候被重新定义。

src/App.js

```
import React from 'react';

function App() {
  const title = 'React';

  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
}

export default App;
```

如果这个变量不需要使用 App 组件中的任何内容（比如：参数来源于函数签名），我们也可以在 App 组件外部定义这个变量：

src/App.js

```
import React from 'react';

const title = 'React';

function App() {
  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
}

export default App;
```

我们将在下一节中使用这个变量！

练习

- 检查上一节的源码[29]。
- 如果你不确定什么时候在 JavaScript（或 React）中使用 `const`，`let`，或者 `var` 来声明变量，可以阅读更多关于它们之间的区别[30]

[29] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Meet-the-React-Component
[30] https://www.robinwieruch.de/const-let-var

- 阅读更多关于 const[31]
- 阅读更多关于 let[32]

思考一下在 App 组件返回的 HTML 中展示 `title` 的方式有哪些。我们将在下一节中使用这个变量。

[31] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
[32] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

React JSX

回想一下，我提到 App 组件返回的类似 HTML 的输出。此输出称为 JSX，它将 HTML 和 JavaScript 混合在一起。让我们看看它如何展示变量：

src/App.js

```
import React from 'react';

const title = 'React';

function App() {
  return (
    <div>
      <h1>Hello {title}</h1>
    </div>
  );
}

export default App;
```

再次使用 `npm start` 启动你的应用程序，并在浏览器中查找渲染的变量，它应该显示为："Hello React"。

让我们聚焦到 HTML，它在 JSX 中的表达几乎相同。带有标签的输入字段可以定义如下：

src/App.js

```
import React from 'react';

const title = 'React';

function App() {
  return (
    <div>
      <h1>Hello {title}</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />
    </div>
  );
}

export default App;
```

我们在此处指定了三个 HTML 属性：`htmlFor`，`id` 和 `type`。这里的 `id` 和 `type` 在原生 HTML 中很熟悉，`htmlFor` 可能是新的属性。`htmlFor` 在 HTML 中反映了 `for` 属性。JSX 替换了一些 HTML 内部属性，但是你可以在 React 文档中找到所有支持的 HTML 属性[33]，属性遵循驼峰式命名约定。随着对 React 的更多了解，将会遇到更多的 JSX 特定属性，例如 `className` 和 `onClick`，而不是 `class` 和 `onclick`。

稍后，我们将重新探索 HTML 输入字段的实现细节；现在，让我们回到 JSX 中的 JavaScript。我们已经定义了要在 App 组件中显示的字符串基础类型，也可以使用 JavaScript 对象完成此操作：

src/App.js

```
import React from 'react';

const welcome = {
  greeting: 'Hey',
  title: 'React',
};

function App() {
  return (
    <div>
      <h1>
        {welcome.greeting} {welcome.title}
      </h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />
    </div>
  );
}

export default App;
```

请记住，JSX 花括号中的所有内容都可使用 JavaScript 表达式（例如函数执行）：

[33] https://reactjs.org/docs/dom-elements.html#all-supported-html-attributes

src/App.js

```
import React from 'react';

function getTitle(title) {
  return title;
}

function App() {
  return (
    <div>
      <h1>Hello {getTitle('React')}</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />
    </div>
  );
}

export default App;
```

JSX 最初是为 React 发明的，但在流行之后，它对其他现代库和框架也适用。这是我喜欢 React 的地方之一。现在不需要任何额外的模板语法（花括号除外），我们现在就可以在 HTML 中使用 JavaScript。

练习

- 检查上一节的源码[34]。
- 确认上一节之后的变更[35]。
- 阅读更多关于 React 的 JSX[36]。
- 定义更多基础和复杂的 JavaScript 数据类型，并在 JSX 中渲染它们。
- 尝试在 JSX 中渲染 JavaScript 数组。如果过于复杂，请不要担心，因为你将在下一节中详细了解。

[34] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-JSX

[35] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Meet-the-React-Component...hs/React-JSX?expand=1

[36] https://reactjs.org/docs/introducing-jsx.html

React 中的列表

到目前为止，我们已经在 JSX 中渲染了一些原始变量。接下来，我们将渲染项目列表。我们先用示例数据进行实验，然后将其应用于从远程 API 获取数据。首先，让我们将数组定义为变量。和以前一样，我们可以在组件外部或内部定义变量。以下是在外部定义变量：

src/App.js

```
import React from 'react';

const list = [
  {
    title: 'React',
    url: 'https://reactjs.org/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  },
  {
    title: 'Redux',
    url: 'https://redux.js.org/',
    author: 'Dan Abramov, Andrew Clark',
    num_comments: 2,
    points: 5,
    objectID: 1,
  },
];

function App() { ... }

export default App;
```

我在这里使用一个 . . . 作为占位符，以使代码段简洁（没有 App 组件实现细节），并专注于必要部分（App 组件外部的 `list` 变量）。在整个学习过程中，我将使用 . . . 作为我之前用于练习的代码块的占位符。如果你感到困惑，可以随时使用大多数章节结尾处提供的 CodeSandbox 链接来验证代码。

列表中的每个项目都有一个标题，一个 URL，一个作者，一个标识符（`objectID`），分数（表示该项目的受欢迎程度）以及评论数。接下来，我们将在 JSX 中动态渲染列表：

src/App.js

```
function App() {
  return (
    <div>
      <h1>My Hacker Stories</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />

      <hr />

      {/* render the list here */}
    </div>
  );
}
```

你可以使用数组内置的 JavaScript map 方法[37]遍历每个项目并返回他们的新版本：

Code Playground

```
const numbers = [1, 4, 9, 16];

const newNumbers = numbers.map(function(number) {
  return number * 2;
});

console.log(newNumbers);
// [2, 8, 18, 32]
```

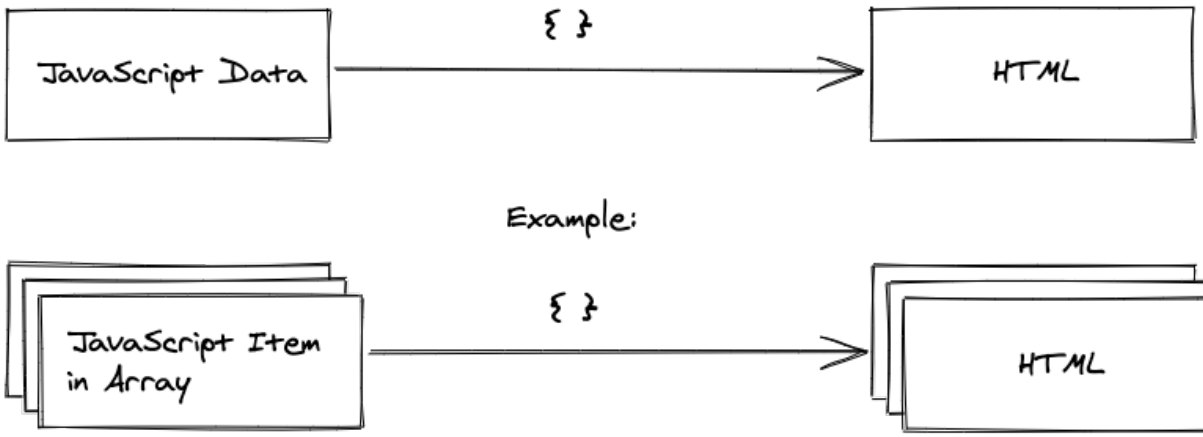
在这种情况下，我们不会从一种 JavaScript 数据类型映射到另一种。相反，我们返回渲染列表中每个项目的 JSX 片段：

[37] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

src/App.js

```
function App() {
  return (
    <div>
      ...

      <hr />

      {list.map(function(item) {
        return <div>{item.title}</div>;
      })}
    </div>
  );
}
```

实际上，我最初对 React 感到惊叹的瞬间之一是使用简洁的 JavaScript 来将 JavaScript 对象列表映射为 HTML 元素，而没有任何其他 HTML 模板语法。它只是 HTML 中的 JavaScript。



React 现在会显示每个项目，但是你仍然可以改进代码，以便 React 更优雅地处理高级动态列表。通过为每个列表项的元素分配 key 属性，React 可以在列表发生更改（例如重新排序）时识别已修改的项目。幸运的是，我们的列表项带有一个标识符：

src/App.js

```
function App() {
  return (
    <div>
      ...

      <hr />

      {list.map(function(item) {
        return (
          <div key={item.objectID}>
            {item.title}
          </div>
        );
      })}
    </div>
  );
}
```

我们要避免使用数组中项目的索引来保证 key 属性是稳定的标识符。例如，如果列表更改顺序，React 将无法正确识别别项目：

Code Playground

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

到目前为止，每个项目仅显示标题。让我们尝试显示更多项目属性：

src/App.js

```
function App() {
  return (
    <div>
      ...

      <hr />

      {list.map(function(item) {
        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>
        );
      })}
    </div>
  );
}
```

map 函数简洁地内联在你的 JSX 中。在 map 函数中，我们可以访问每个项目及其属性。每个项目的 `url` 属性都可用作锚标签 `<a>` 的动态 `href` 属性。JSX 中的 JavaScript 不仅可以用于显示项目，还可以动态分配 HTML 属性。

练习

- 检查上一节的源码[38]。
- 确认上一节之后的变更[39]。
- 阅读更多关于为何需要 React 的 key 属性的信息（0[40]，1[41]，2[42]）。如果你还不了解实现，请不要担心，只需关注它对动态列表造成的问题。
- 回顾标准内置数组方法[43] -* 尤其是原生 JavaScript 中可用的 *map*，*filter* 和 *reduce*。
- 如果你返回 `null` 而不是 JSX，会发生什么？

[38] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Lists-in-React
[39] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-JSX...hs/Lists-in-React?expand=1
[40] https://dev.to/jtonzing/the-significance-of-react-keys---a-visual-explanation--5617
[41] https://www.robinwieruch.de/react-list-key
[42] https://reactjs.org/docs/lists-and-keys.html
[43] https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/

- 用更多项目扩展列表，让示例更真实。
- 练习在 JSX 中使用不同的 JavaScript 表达式。

认识另一种形式的 React 组件

到目前为止，我们只用了 App 组件创建我们的应用程序。在上一节中，我们用 App 组件来表示在 JSX 中渲染列表的一切，它应该随着你的需求不断扩展，并且最终能够处理一些更复杂的任务。为了解决这个问题，我们将 App 组件的一些职责分解为一个独立的 List 组件：

src/App.js

```
const list = [ ... ];

function App() { ... }

function List() {
  return list.map(function(item) {
    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>
    );
  });
}
```

可选项：如果你觉得这个组件看起来有点怪，那是因为返回的 JSX 最外层是以 JavaScript 开始的。我们也可以在它外面包一层 HTML，但在这儿我们还是使用之前的版本。

src/App.js

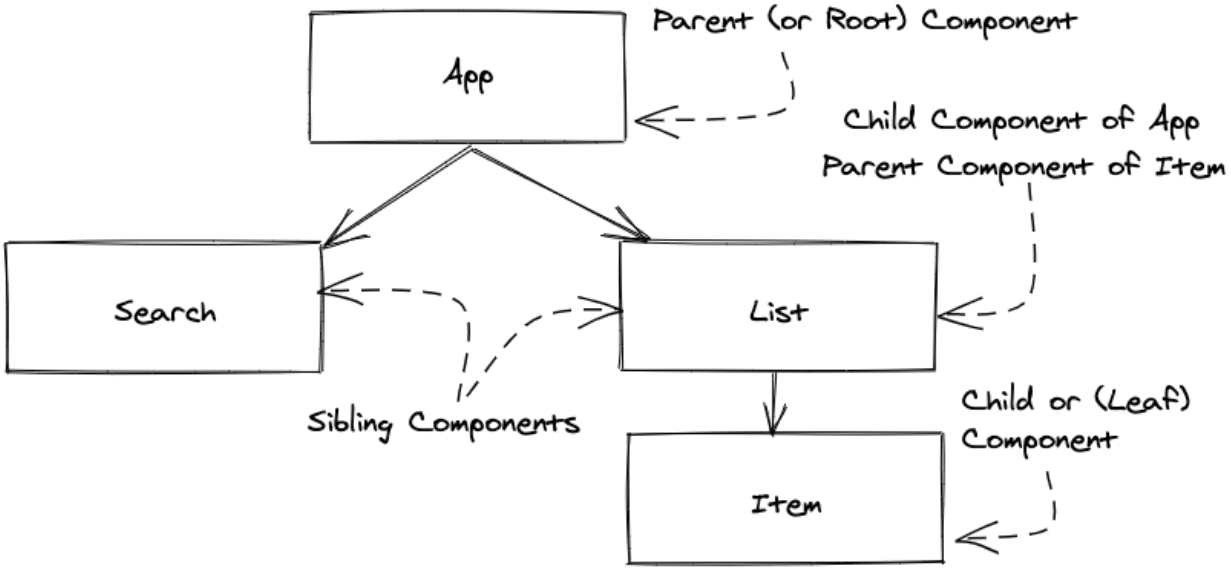
```
function List() {
  return (
    <div>
      {list.map(function(item) {
        return (... );
      })}
    </div>
  );
}
```

现在就可以在 App 组件中使用 List 组件了：

src/App.js

```
function App() {
  return (
    <div>
      <h1>My Hacker Stories</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />

      <hr />

      <List />
    </div>
  );
}
```

你刚刚创建了第一个 React 组件！通过这个例子，我们可以看到那些封装的有意义任务的组件是如何在更大的 React 应用中工作的。



大一点的 React 应用具备组件层次结构（也称为组件树）。一般来说最顶层的入口组件（比如：APP）会在它里面嵌套组件树。App 组件是 List 组件的父组件，相应的，List 组件也是 App 组件的子组件。在组件树中，App 是根组件，这个组件不会渲染其他叶子组件（比如：List）。App 可以有多个子组件，List 组件也是。如果 App 组件有其他子组件，那这个子组件称为 List 组件的兄弟组件。

练习

- 检查上一节的源码[44]。
- 确认上一节之后的变更[45]。
- 在纸上绘制一下 App 组件和 List 组件树。用其他可能的组件（比如：在 App 组件中提取 input 和 label 的 Search 组件）来扩展这个组件树。尝试找出其他组件哪些可以作为独立的组件被提取。
- 如果在 App 组件中使用 Search 组件，那对于 List 组件来说，Search 组件是它的兄弟组件还是父组件或子组件?
- 思考一下，如果我们使用全局变量来处理 `list` 会出现什么问题？我们即将在接下来的章节中介绍如何处理这个问题。

---

[44] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Meet-another-React-Component

[45] ttps://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Lists-in-React...hs/Meet-another-React-Component?expand=1

实例化 React 组件

接下来，我将简单介绍一下 JavaScript 类，以帮助阐明 React 组件。从技术上讲，它们没有什么关系，需要注意这一点，但这对于您理解组件概念是一个合适的类比。

类在面向对象编程语言中非常常用。JavaScript 的编程模式非常灵活，允许函数式编程和面向对象编程共存。要概括 JavaScript 类以进行面向对象的编程，思考以下的 *Developer* 类：

Code Playground

```
class Developer {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getName() {
    return this.firstName + ' ' + this.lastName;
  }
}
```

每个类都有一个构造函数 constructor，接收一些参数并将其分配给类的实例。类还可以定义与主体相关的函数（例如 `getName`），称为 方法 or 类方法 .

定义 Developer 类只是其中一部分；另外还需要进行实例化。类定义是对功能蓝图的描绘，当使用 `new` 语句创建实例时会被用到。

Code Playground

```
// 类定义
class Developer { ... }

// 类实例化
const robin = new Developer('Robin', 'Wieruch');

console.log(robin.getName());
// "Robin Wieruch"

// 实例化另一个
const dennis = new Developer('Dennis', 'Wieruch');

console.log(dennis.getName());
// "Dennis Wieruch"
```

如果有一个 JavaScript 类的定义存在，则可以创建它的 多个实例。它类似 React component，它只有 一个组件定义，但可以有 多个组件实例：

src/App.js

```
// 定义 App 组件
function App() {
  return (
    <div>
      <h1>My Hacker Stories</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />

      <hr />

      {/* 创建一个 List 组件实例 */}
      <List />
      {/* 创建另一个 List 组件实例 */}
      <List />
    </div>
  );
}

// 定义 List 组件
function List() { ... }
```

一旦我们定义了 组件，我们就可以在 JSX 中任何地方像 HTML 元素一样使用它。元素会产生一个 组件实例，换句话说，组件被实例化了。您可以根据需要创建任意数量的组件实例。它与 JavaScript 的类定义和用法没有太大区别。

练习

- 熟悉术语 组件声明, 实例, 和 元素。
- 通过创建一个 List 组件的多个组件实例来进行实验。
- 想想看，如何给每个 List 组件赋予自己的 `list` 属性。

React DOM

现在我们已经了解了组件定义及其实例化，接下来我们来看看 App 组件的实例化。它从一开始就在我们的应用程序的 *src/index.js* 文件中：

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

import App from './App';

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

在导入 React 库的代码下面，有另一个被导入的库，称为 `react-dom`，其中的 `ReactDOM.render()` 方法将使用 JSX 替换 HTML 中的一个 DOM 节点，该过程是将 React 集成到了 HTML 中。`ReactDOM.render()` 这个方法需要传入两个参数；首先是要渲染的 JSX，你可以直接传入一个简单的 JSX，而无需任何组件的实例化，也可以直接传入一个组件的实例。

Code Playground

```
ReactDOM.render(
  <h1>Hello React World</h1>,
  document.getElementById('root')
);
```

第二个参数指定了 React 应用在何处输入你的 HTML。它需要在 *public/index.html* 文件中找到一个 `id='root'` 的元素。这是一个基本的 HTML 文件。

练习

- 打开 *public/index.html*，查看 React 应用程序在何处放置 HTML。
- 设想一下我们如何在使用 HTML 的外部 Web 应用程序中包含 React 应用程序。
- 阅读更多关于 React 中元素渲染[46]的信息。

[46] https://reactjs.org/docs/rendering-elements.html

React 函数定义（高阶）

以下的重构建议是可选的，用来辅助说明 JavaScript/React 不同的模式。就算不用这些模式，你一样可以做出完整的 React 应用，所以如果下面这些内容看起来太复杂了，也不需要灰心。

src/App.js 里所有的组件都是函数组件。JavaScript 有很多种声明函数的方式。到目前为止，我们都是用 function 来声明，但箭头函数看起来会更简洁一些：

Code Playground

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

// arrow function declaration
const () => { ... }
```

在声明箭头函数时，如果只有一个参数，可以把括号去掉。但有多个参数的时候，括号是必须的：

Code Playground

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

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

// not allowed
const item, index => { ... }

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

用箭头函数声明 React 函数组件会让它们看起来更加简洁：

src/App.js

```
const App = () => {
  return (
    <div>
      ...
    </div>
  );
};

const List = () => {
  return list.map(function(item) {
    return (
      <div key={item.objectID}>
        ...
      </div>
    );
  });
};
```

对于其他的函数也一样，比如我们用到的 JavaScript 数组的 map 函数：

src/App.js

```
const List = () => {
  return list.map(item => {
    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>
    );
  });
};
```

如果一个箭头函数什么也没做，只是返回了某些结果，——换句话说，如果一个箭头函数没有执行任何任务，只返回信息——，你还可以去掉块体（花括号）。在简写体中，会附一个隐式的返回声明，所以 return 也是可以去掉的：

Code Playground

```
// with block body
count => {
  // perform any task in between

  return count + 1;
}

// with concise body
count =>
  count + 1;
```

这也可以应用在 App 和 List 组件上，因为它们除了返回 JSX 之外什么都没做。同样也可以用在 map 里的箭头函数上：

src/App.js

```
const App = () => (
  <div>
    ...
  </div>
);

const List = () =>
  list.map(item => (
    <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>
  ));
```

去掉了 function 声明、花括号和 return 语句后，我们的 JSX 现在看起来更简洁了。但始终要记得，这只是可选的步骤，使用一般的函数声明、而不是箭头函数，带花括号的箭头函数体、而非省略它，这些都是可接受的。有时函数体是必要的，当我们需要在函数签名和返回语句之间引入业务逻辑时：

Code Playground

```
const App = () => {
  // perform any task in between

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

确保你理解了这个重构的概念，因为后面我们将在有和没有函数体的箭头函数组件之间灵活切换，取决于组件的需求来决定使用哪种形式。

练习

- 检查上一节的源码[47]。
- 确认上一节之后的变更[48]。
- 阅读更多关于 JavaScript 箭头函数[49]的文章。
- 熟练掌握有块体和返回的箭头函数、以及没有返回语句的简写体。

[47] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Component-Definition

[48] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Meet-another-React-Component...hs/React-Component-Definition?expand=1

[49] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

JSX 中的处理函数

我们还没使用到 App 组件里的输入框及其标签。在 JSX 之外的 HTML 中，input 有一个 onchange 处理函数[50]。我们将要学习如何在 React 组件中的 JSX 上使用 onchange 处理函数。让我们先把 App 组件从简洁体重构回块体，以便添加更多实现细节。

src/App.js

```
const App = () => {
  // do something in between

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />

      <hr />

      <List />
    </div>
  );
};
```

随后定义一个函数，普通函数或者箭头函数都可以，用来处理 input 的 change 事件。在 React 中，这种函数叫做（事件）处理函数。现在可以把这个函数传给 input 的 onChange 属性了（JSX 命名的属性）。

src/App.js

```
const App = () => {
  const handleChange = event => {
    console.log(event);
  };

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" onChange={handleChange} />
```

[50] https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onchange

```
    <hr />

    <List />
   </div>
  );
};
```

在浏览器中打开应用，再打开浏览器的开发者工具，就可以看到在输入框打字之后的日志了。这被称为合成事件，由一个 JavaScript 对象构成。通过这个对象，我们可以拿到输入框发出的值：

src/App.js

```
const App = () => {
  const handleChange = event => {
    console.log(event.target.value);
  };

  return ( ... );
};
```

合成事件本质上就是把浏览器的原生事件[51]包了一层，附加了一些有用的函数可以阻止浏览器默认行为（比如，用户点击表单提交按钮之后浏览器会刷新页面）。有时你会用到这个事件，有时不需要。

这就是如何通过给 HTML 元素附加 JSX 处理函数来响应用户交互的方式。只把函数传给这些处理函数，而不是函数的返回值，除非返回值也是一个函数：

Code Playground

```
// don't do this
<input
  id="search"
  type="text"
  onChange={handleChange()}
/>

// do this instead
<input
  id="search"
  type="text"
  onChange={handleChange}
/>
```

[51] https://developer.mozilla.org/en-US/docs/Web/Events

HTML 和 JavaScript 在 JSX 里合作地很愉快。HTML 中的 JavaScript 可以展示对象，可以传递 JavaScript 的基本类型给 HTML 属性（例如把 `href` 传给 `<a>`），也可以通过给元素属性传递函数来处理事件。

我个人更倾向于用箭头函数来定义事件处理函数，这样更简洁。但是在稍大的 React 组件中，我发现自己也会使用函数声明，因为这样能更好地和组件中其他的变量声明区分开来。

练习

- 检查上一节的源码[52]。
- 确认上一节之后的变更[53]。
- 阅读更多关于 React 的事件[54]的文章。

[52] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Handler-Function-in-JSX
[53] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-Component-Definition...hs/Handler-Function-in-JSX?expand=1
[54] https://reactjs.org/docs/events.html

React Props

我们现在把 `list` 变量作为当前应用的一个全局变量。我们在 App 组件内直接从全局作用域使用它，在 List 组件内同样这样使用。如果你只有一个变量的话这种方式是可行的，但这无法扩展为在不同文件的多个组件内使用多个变量。

通过使用所谓的 props ，我们可以把变量作为信息从一个组件传给另一个组件。在使用 props 之前，我们要先把 list 从全局作用域移至 App 组件内部，并按它的实际领域重命名。

src/App.js

```
const App = () => {
  const stories = [
    {
      title: 'React',
      url: 'https://reactjs.org/',
      author: 'Jordan Walke',
      num_comments: 3,
      points: 4,
      objectID: 0,
    },
    {
      title: 'Redux',
      url: 'https://redux.js.org/',
      author: 'Dan Abramov, Andrew Clark',
      num_comments: 2,
      points: 5,
      objectID: 1,
    },
  ];

  const handleChange = event => { ... };

  return ( ... );
};
```

接下来我们将使用 **React props** 来把这个数组传递给 List 组件：

src/App.js

```
const App = () => {
  const stories = [ ... ];

  const handleChange = event => { ... };

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" onChange={handleChange} />

      <hr />

      <List list={stories} />
    </div>
  );
};
```

这个变量在 App 组件中被称为 `stories`，并且我们把它以这个名字传给了 List 组件。然而在 List 组件实例化时，它被赋值给了 `list` 属性。我们在 List 组件的函数签名中通过 `props` 对象中的 `list` 访问它。

src/App.js

```
const List = props =>
  props.list.map(item => (
    <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>
  ));
```

通过这样操作，我们已经避免了 list/stories 变量在 App 组件内污染全局作用域。因为 `stories` 不是在 App 组件被直接使用的，而是在它的其中一个子组件内，我们把它作为 props 传给了 List 组件。然后我们可以通过函数签名的第一个参数 `props` 来访问它。

练习

- 检查上一节的源码[55]。
- 确认上一节之后的变更[56]。
- 阅读更多关于如何向 React 组件传递 props[57]。

[55] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Props
[56] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Handler-Function-in-JSX...hs/React-Props?expand=1
[57] https://www.robinwieruch.de/react-pass-props-to-component

React State

React Props 用于沿着组件树向下传递信息；**React State** 用于实现应用交互。我们可以通过与应用的交互来改变它的表现。

首先，我们可以从 React 中获得一个叫作 useState 的工具函数去管理 state，useState 函数被称作 hook。React 中有不止一个 hook，它们与 React 中的 state 管理和其它事务有关，你可以通过接下来的章节学习它们。现在，我们只关注 React 中的 useState hook。

src/App.js

```
const App = () => {
  const stories = [ ... ];

  const [searchTerm, setSearchTerm] = React.useState('');

  ...
};
```

React 的 `useState` hook 以一个初始 *state* 作为参数，我们可以使用空字符串。同时它会返回包含两个值的数组，第一个值（`searchTerm`）表示当前 *state*；第二个值（`setSearchTerm`）是一个更新这个 *state* 的函数。我有时会把这个函数称作 *state* 更新函数。

如果你不熟悉这种一个数组返回两个值的语法，可以参考阅读 JavaScript 数组解构[58]的内容。它可以用于更简洁地读取数组。简而言之，这是数组的解构，并且其好处可以直观看到：

Code Playground

```
// basic array definition
const list = ['a', 'b'];

// no array destructuring
const itemOne = list[0];
const itemTwo = list[1];

// array destructuring
const [firstItem, secondItem] = list;
```

在 React 中，`useState` hook 是返回一个数组的函数。再以下面的 JavaScript 示例作比较：

[58] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Array_destructuring

Code Playground

```
function getAlphabet() {
  return ['a', 'b'];
}

// no array destructuring
const itemOne = getAlphabet()[0];
const itemTwo = getAlphabet()[1];

// array destructuring
const [firstItem, secondItem] = getAlphabet();
```

数组解构只是逐个获取数组值的一种简写方式，如果你在 React 中不以解构的方式表示数组，那么它的可读性就会降低：

src/App.js

```
const App = () => {
  const stories = [ ... ];

  // less readable version without array destructuring
  const searchTermState = React.useState('');
  const searchTerm = searchTermState[0];
  const setSearchTerm = searchTermState[1];

  ...
};
```

React 团队之所以选择数组解构，是因为它语法简洁并且有命名解构后的变量的能力。下面的代码片段是一个销毁数组的例子：

src/App.js

```
const App = () => {
  const stories = [ ... ];

  const [searchTerm, setSearchTerm] = React.useState('');

  ...
};
```

在我们初始化 state，且可以获取当前 state 和 state 更新函数之后，我们就可以使用它们显示当前的 state 并通过 APP 组件内部的事件处理函数对其进行更新了。

src/App.js

```
const App = () => {
  const stories = [ ... ];

  const [searchTerm, setSearchTerm] = React.useState('');

  const handleChange = event => {
    setSearchTerm(event.target.value);
  };

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" onChange={handleChange} />

      <p>
        Searching for <strong>{searchTerm}</strong>.
      </p>

      <hr />

      <List list={stories} />
    </div>
  );
};
```

当用户在输入框中输入内容时，输入框内容的改变会被处理函数以当前内部值捕获。事件处理函数的逻辑使用 state 更新函数去设置新的 state。在给组件设置了新的 state 之后，组件会被重新渲染，这意味着该组件函数会被再次执行。新的 state 变为了当前的 state，并可以在组件的 JSX 中显示。

练习

- 检查 上一节的源码[59]。
- 确认 上一节之后的变更[60]。
- 阅读更多关于 JavaScript 数组解构[61]。

[59] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-State
[60] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-Props...hs/React-State?expand=1
[61] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Array_destructuring

- 阅读更多关于 React 的 useState hook (0[62], 1[63]), 使用它让你的 React 组件具有交互性。

[62] https://www.robinwieruch.de/react-usestate-hook
[63] https://reactjs.org/docs/hooks-state.html

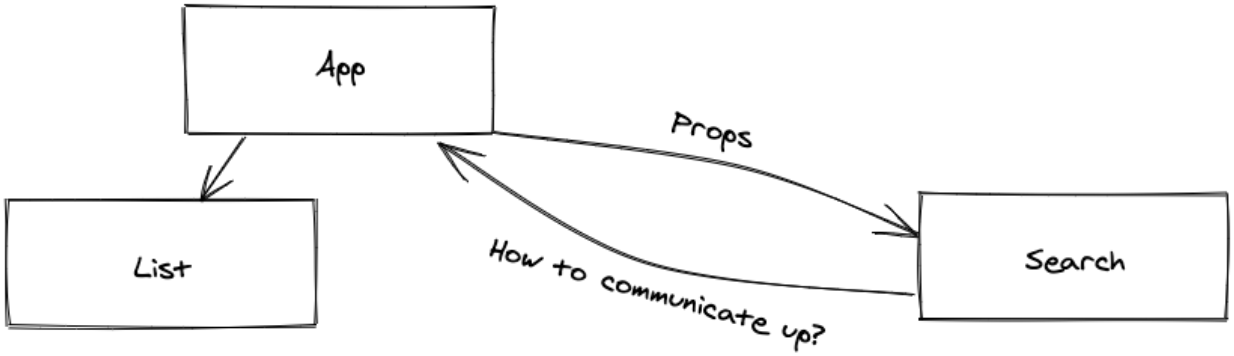
JSX 回调处理函数

接下来，我们将专注在输入框和标签上，通过分离一个独立的 Search 组件，并在 App 组件中创建一个实例。通过这个过程，Search 组件将成为 List 组件的兄弟组件，反之亦然。我们还将把处理函数和 state 移动到 Search 组件内来保持我们功能的完整性。

src/App.js

```
const App = () => {
  const stories = [ ... ];

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <Search />

      <hr />

      <List list={stories} />
    </div>
  );
};

const Search = () => {
  const [searchTerm, setSearchTerm] = React.useState('');

  const handleChange = event => {
    setSearchTerm(event.target.value);
  };

  return (
    <div>
      <label htmlFor="search">Search: </label>
      <input id="search" type="text" onChange={handleChange} />

      <p>
        Searching for <strong>{searchTerm}</strong>.
      </p>
    </div>
  );
};
```

我们有一个抽取出的 Search 组件，它可以处理 state 并且显示 state 但没有暴露出它的内容。该组件将 `searchTerm` 显示为文本，但并未与父组件或兄弟组件共享此信息。由于 Search 组件除了显示搜索项外什么也不做，因此它对其他组件毫无用处。



由于 props 只能往下传递，因此无法将 JavaScript 数据信息沿组件树往上传递。但是，我们可以引入回调函数：回调函数被引入（A），在其他地方使用了回调函数（B），但调用在回调函数的位置（C）。

src/App.js

```
const App = () => {
  const stories = [ ... ];

  // A
  const handleSearch = event => {
    // C
    console.log(event.target.value);
  };

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <Search onSearch={handleSearch} />

      <hr />

      <List list={stories} />
    </div>
  );
};

const Search = props => {
  const [searchTerm, setSearchTerm] = React.useState('');
```

```
  const handleChange = event => {
    setSearchTerm(event.target.value);

    // B
    props.onSearch(event);
  };

  return ( ... );
};
```

在您的代码中使用可以省略掉注释 A，B 和 C，因为它们只是在提醒每个代码块要执行的任务。考虑一下回调函数的概念：我们将一个函数从一个组件（App）传递到另一个组件（Search）；我们在第二个组件（Search）中调用它；但实际上它在第一个组件（App）中被执行。这样，我们就能在组件树中往上传递信息。一个组件中使用的处理函数成为回调处理函数，并将该函数通过 React props 往下传递给其他组件。React props 在组件树中始终将信息往下传递，而回调函数作为 props 传递时可以用来和上层组件通信。

练习

- 检查上一节的源码[64]。
- 确认上一节之后的变更[65]。
- 根据需要多次访问处理函数和回调处理函数的概念。

[64] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Callback-Handler-in-JSX

[65] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-State...hs/Callback-Handler-in-JSX?expand=1

React 状态提升

目前，Search 组件仍然持有其内部 state。虽然我们创建了一个回调处理函数来将信息向上传递给 App 组件，但我们还没有使用它。我们需要弄清楚如何在多个组件之间共享 Search 组件的 state。

在将 list 作为 props 传递给 List 组件之前，还需要在 App 中使用搜索词来过滤此列表。我们需要将 state 从 Search 提升到 App 组件，以便与更多组件共享 state。

src/App.js

```
const App = () => {
 const stories = [ ... ];

 const [searchTerm, setSearchTerm] = React.useState('');

 const handleSearch = event => {
  setSearchTerm(event.target.value);
 };

 return (
  <div>
   <h1>My Hacker Stories</h1>

   <Search onSearch={handleSearch} />

   <hr />

   <List list={stories} />
  </div>
 );
};

const Search = props => (
 <div>
  <label htmlFor="search">Search: </label>
  <input id="search" type="text" onChange={props.onSearch} />
 </div>
);
```

前面我们已经了解了回调处理函数，因为它帮助我们保持了一个从 Search 组件到 App 组件的开放通信通道。Search 组件不再管理 state，只是在文本输入到输入框之后，将事件向上传递给 App 组件。你也可以将 `searchTerm` 作为 prop 向下传递，并在 App 或 Search 组件中再次展示它。

总是在组件中管理 state，其中每个感兴趣的组件，要么是管理 state 的组件（直接使用来自 state 的信息），要么是管理组件下游的组件（使用 prop 的信息）。如果下游的组件需要更新 state，请向下传递一个回调处理函数给它（参考 Search 组件）。如果组件需要使用 state（例如展示它），则将其作为 props 向下传递。

通过管理 App 组件中的搜索功能 state，在将 `list` 传递给 List 组件之前，我们最终可以用有状态的 `searchTerm` 来过滤列表：

src/App.js

```
const App = () => {
 const stories = [ ... ];

 const [searchTerm, setSearchTerm] = React.useState('');

 const handleSearch = event => {
  setSearchTerm(event.target.value);
 };

 const searchedStories = stories.filter(function(story) {
  return story.title.includes(searchTerm);
 });

 return (
  <div>
   <h1>My Hacker Stories</h1>

   <Search onSearch={handleSearch} />

   <hr />

   <List list={searchedStories} />
  </div>
 );
};
```

在这里，JavaScript 数组的内置过滤函数[66] 可以用来创建一个新的过滤数组。Filter 函数接受一个函数作为参数，该函数访问数组中的每一项并返回 true 或 false。如果函数返回 true，则表示满足条件，该项将被保留在新创建的数组中；如果函数返回 false，则将该项删除。

[66] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

Code Playground

```
const words = [
 'spray',
 'limit',
 'elite',
 'exuberant',
 'destruction',
 'present'
];

const filteredWords = words.filter(function(word) {
 return word.length > 6;
});

console.log(filteredWords);
// ["exuberant", "destruction", "present"]
```

Filter 函数会检查 `searchTerm` 是否出现在我们 story 的标题中，但是它对字母的大小写仍然过于敏感。如果我们搜索 “react”，你的渲染列表中不会有过滤出来的 “React” story。为了解决这个问题，我们必须将 story 的标题和 `searchTerm` 改为小写。

src/App.js

```
const App = () => {
 ...

 const searchedStories = stories.filter(function(story) {
   return story.title
     .toLowerCase()
     .includes(searchTerm.toLowerCase());
 });

 ...
};
```

现在你应该可以搜索 “eact”，”React” 或者 “react”，并看到两个展示 stories 中的其中一个。你刚刚向应用程序添加了一个交互功能。

剩下的部分展示了几个重构步骤。最后，我们将使用最终的重构版本，因此理解并保留这些步骤是有意义的。正如前面学到的那样，我们可以使用 JavaScript 箭头函数让该函数更加整洁：

src/App.js

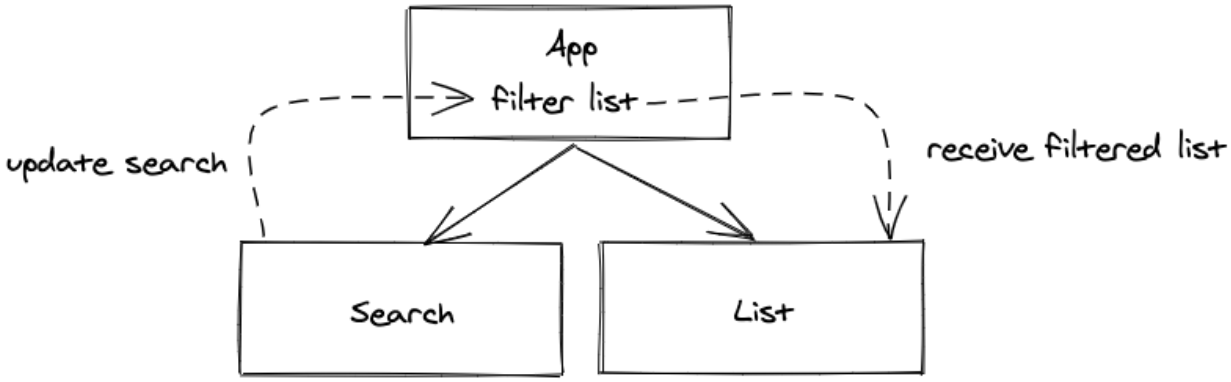
```
const App = () => {
 ...

 const searchedStories = stories.filter(story => {
   return story.title
     .toLowerCase()
     .includes(searchTerm.toLowerCase());
 });

 ...
};
```

此外，我们可以将 return 语句转换为立即返回，因为在返回之前没有其他任务（业务逻辑）发生。

src/App.js

```
const App = () => {
 ...

 const searchedStories = stories.filter(story =>
   story.title.toLowerCase().includes(searchTerm.toLowerCase())
 );

 ...
};
```

这就是 filter 函数的内联函数的重构步骤。它有许多变体，在可读性和简洁性之间保持平衡并不总是那么简单。然而，我觉得只要尽可能保持简洁就能够在大部分时间保持可读性。



现在，我们可以在 App 组件中使用 Search 组件的回调函数来更新 state，从而在 React 中操作 state。当前 state 用作列表的过滤器。通过回调处理函数，我们使用了来自 App 组件中的 Search 组件的信息来更新共享 state，并间接地在 List 组件中获取了过滤后的列表。

练习

- 检查上一节的源码[67]。
- 确认上一节之后的变更[68]。

[67] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Lifting-State-in-React

[68] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Callback-Handler-in-JSX...hs/Lifting-State-in-React?expand=1

受控组件

受控组件并不一定是 React 组件，也可以是 HTML 元素。这一节我们将学习如何把 Search 组件以及内部的输入框转化为一个受控组件。

让我们看一个展示为什么我们需要在 React 应用中遵循受控组件概念的情景。在我们应用了以下改变之后：赋予 `searchTerm` 一个初始 state，你能发现在你浏览器上出现的错误吗?

src/App.js

```
const App = () => {
 const stories = [ ... ];

 const [searchTerm, setSearchTerm] = React.useState('React');

 ...
};
```

当这个列表基于初始搜索被过滤的时候，输入框不会展示初始的 `searchTerm` 。我们期望输入框反应 `searchTerm` 的实际初始 state，但它只反映了被过滤的列表。

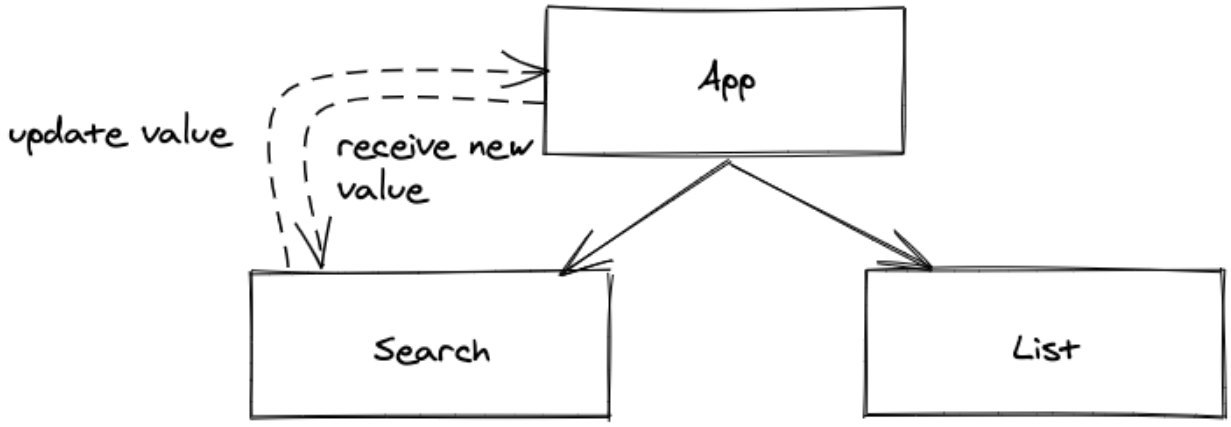
因此我们需要把 Search 组件以及内部的输入框转化成一个受控组件。到目前为止，这个输入框还没有获知任何关于 `searchTerm` 的信息。它只使用了 change 事件来通知我们发生的改变。而实际上它还拥有一个 `value` 属性。

src/App.js

```
const App = () => {
 const stories = [ ... ];

 const [searchTerm, setSearchTerm] = React.useState('React');

 ...

 return (
   <div>
     <h1>My Hacker Stories</h1>

     <Search search={searchTerm} onSearch={handleSearch} />

     ...
   </div>
 );
};
```

```
const Search = props => (
 <div>
  <label htmlFor="search">Search: </label>
  <input
   id="search"
   type="text"
   value={props.search}
   onChange={props.onSearch}
  />
 </div>
);
```

现在输入框开始用 React state 中的 `searchTerm` 作为正确的初始值。并且当我们改变 `searchTerm` 的值时，我们强制输入框使用了 React state 中的值（通过 props）。而之前输入框仅仅是以原生 HTML 的方式管理着自己的内部状态。



我们在这一节学习了受控组件，再结合之前学习过程中的所有章节，可以发现另一个被称为单向数据流的概念。

Visualization

```
UI -> Side-Effect -> State -> UI -> ...
```

一个 React 应用和其中的组件一开始都有一个初始 state, 可能通过 props 的方式向下传给其他组件。它会在第一次时被渲染成 UI 。一旦副作用发生，比如用户输入或者从远端 API 加载数据，这个改变会在 React state 中被捕获。而一旦 state 被改变，所有受到改变的 state 或者被暗中改变的 props 影响的组件将重新渲染（组件的函数会被重新调用）。

在之前的章节中，我们也学习了 React 的 组件生命周期。首先所有组件将会在组件层级中从上往下地被实例化。这包含了所有被基于初始值（比如初始 state ）实例化的 hooks （比如 `useState`）。那时 UI 等待着诸如用户交互之类的副作用。一旦 state 发生了改变（比如当

前 state 被 useState 中的更新 state 的函数改变），所有受到改变的 state 或者 props 影响的组件将会重新渲染。

每一次运行一个组件的函数会从 hooks 中使用最新值（比如当前 state ）并且不会重新执行初始化（比如初始 state）。这可能会显得有些奇怪，因为有人可能会假定 useState hooks 函数会用初始值重新初始化，但其实并没有。Hooks 只会在组件第一次渲染时初始化一次，而之后 React 会在内部追踪其最新值。

练习

- 检查上一节的源码[69]。
- 确认上一节之后的变更[70]。
- 阅读更多关于 React 受控组件[71]的文章。
- 尝试在你的 React 组件中通过 console.log() 观察你的改变是怎么渲染的，包括初始时和输入栏改变之后。

[69] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Controlled-Components

[70] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Lifting-State-in-React...hs/React-Controlled-Components?expand=1

[71] https://www.robinwieruch.de/react-controlled-components/

Props 处理（高级）

Props 经组件树由父级向下传递给子级。由于我们经常使用 props 在组件间传递信息，有时也会通过其他中间组件来进行传递，所以知道一些小技巧可以让传递 props 更方便。

注意：下面推荐的重构是为了让你学习不同的 *JavaScript/React* 模式，尽管在没有这些模式的情况下你仍然可以构建完整的 *React* 应用程序。考虑一下这些先进的 *React* 技术，会让你的源代码更加简洁。

React props 是一个 JavaScript 对象，否则我们就无法访问 React 组件中的 `props.list` 或 `props.onSearch` 。因为 `props` 是一个对象，它只是将信息从一个组件传递给另一个组件，所以我们可以对它应用一些 JavaScript 技巧。例如，用现代的 JavaScript 对象解构[72]来访问对象的属性：

Code Playground

```
const user = {
 firstName: 'Robin',
 lastName: 'Wieruch',
};

// without object destructuring
const firstName = user.firstName;
const lastName = user.lastName;

console.log(firstName + ' ' + lastName);
// "Robin Wieruch"

// with object destructuring
const { firstName, lastName } = user;

console.log(firstName + ' ' + lastName);
// "Robin Wieruch"
```

如果我们需要访问一个对象的多个属性，用一行代码来代替多行代码往往更简单、更优雅。这就是为什么在 JavaScript 中已经广泛使用对象解构的原因。让我们把这些知识运用到我们的 Search 组件中的 React props 上。首先，我们要把 Search 组件的箭头函数从简写体重构成块体：

[72] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

src/App.js

```
const Search = props => {
 return (
   <div>
     <label htmlFor="search">Search: </label>
     <input
       id="search"
       type="text"
       value={props.search}
       onChange={props.onSearch}
     />
   </div>
 );
};
```

接着，我们可以在组件的函数体中对 `props` 对象进行解构：

src/App.js

```
const Search = props => {
 const { search, onSearch } = props;

 return (
   <div>
     <label htmlFor="search">Search: </label>
     <input
       id="search"
       type="text"
       value={search}
       onChange={onSearch}
     />
   </div>
 );
};
```

这就是对 React 组件中的 `props` 对象的基本解构，这样就可以在组件中方便的使用 `props` 的属性。但是，我们也不得不将 Search 组件的箭头函数从简写体重构为块体，在函数体中用对象解构来访问 `props` 的属性。如果我们按照这样的模式来做的话，这样的情况会经常发生，而且对我们来说并不容易，因为我们必须不断地重构我们的组件。我们可以将这一切更进一步，直接在组件的函数签名中对 `props` 对象进行解构，再次省略组件的块体：

src/App.js

```
const Search = ({ search, onSearch }) => (
 <div>
  <label htmlFor="search">Search: </label>
  <input
   id="search"
   type="text"
   value={search}
   onChange={onSearch}
  />
 </div>
);
```

React `props` 本身很少在组件中使用，相反，所有包含在 `props` 对象中的信息都会被使用。通过在函数签名中马上解构 `props` 对象，我们可以方便地访问所有信息，而不需要处理 `props` 容器。这是本节的基本知识，不过，我们可以通过下面的高级课程更进一步。

让我们以另一个场景来深入学习 React 中 props 处理的高级方式：为了准备这个场景，我们将用之前学过的关于 React 的 `props` 对象解构的知识，从 List 组件中提取一个新的 Item 组件：

src/App.js

```
const List = ({ list }) =>
 list.map(item => <Item key={item.objectID} item={item} />);

const Item = ({ item }) => (
 <div>
  <span>
   <a href={item.url}>{item.title}</a>
  </span>
  <span>{item.author}</span>
  <span>{item.num_comments}</span>
  <span>{item.points}</span>
 </div>
);
```

现在，Item 组件中的入参 `item` 与前面讨论的 `props` 有一个共同点：它们都是 JavaScript 对象。另外，即使 `item` 对象已经从 Item 组件的函数签名中的 `props` 中解构了，但它并没有直接在 Item 组件中使用。`item` 对象只是将其信息（对象属性）传递给了元素。

你会在接下来持续的讨论中看到，上面代码中所示的解决方案是没有问题的。不过，我想再给大家展示另外两种变体，因为这里有很多关于 JavaScript 对象的知识需要学习。让我们从嵌套解构开始，看看它是如何工作的：

Code Playground

```
const user = {
  firstName: 'Robin',
  pet: {
    name: 'Trixi',
  },
};

// without object destructuring
const firstName = user.firstName;
const name = user.pet.name;

console.log(firstName + ' has a pet called ' + name);
// "Robin has a pet called Trixi"

// with nested object destructuring
const {
  firstName,
  pet: {
    name,
  },
} = user;

console.log(firstName + ' has a pet called ' + name);
// "Robin has a pet called Trixi"
```

嵌套解构可以帮助我们从深度嵌套的对象中访问其属性（比如说 user 中 pet 的 name 属性）。现在，在我们的 Item 组件中，由于 item 对象从来没有在 Item 组件的 JSX 元素中被直接使用过，所以我们也可以在组件的函数签名中进行嵌套解构：

src/App.js

```
// Variation 1: Nested Destructuring

const Item = ({
 item: {
   title,
   url,
   author,
   num_comments,
   points,
 },
}) => (
```

```
 <div>
   <span>
     <a href={url}>{title}</a>
   </span>
   <span>{author}</span>
   <span>{num_comments}</span>
   <span>{points}</span>
 </div>
);
```

嵌套解构可以帮助我们在函数签名中收集所有需要的 `item` 对象的信息，以便在组件元素中直接使用。然而，嵌套解构在函数签名中因为缩进而引入了很多杂乱的信息。虽然它在这里不是最可读的方式，但在其他场景中还是很有用的。

让我们利用 JavaScript 的展开运算符和剩余运算符进行另一种实现。为了做好准备，我们将把 List 和 Item 组件重构为以下的实现。我们不再将 List 组件中的 `item` 作为对象传递给 Item 组件，而是将 `item` 对象的每一个属性都传递给它：

src/App.js

```
// Variation 2: Spread and Rest Operators
// 1. Iteration

const List = ({ list }) =>
 list.map(item => (
   <Item
     key={item.objectID}
     title={item.title}
     url={item.url}
     author={item.author}
     num_comments={item.num_comments}
     points={item.points}
   />
 ));

const Item = ({ title, url, author, num_comments, points }) => (
 <div>
   <span>
     <a href={url}>{title}</a>
   </span>
   <span>{author}</span>
   <span>{num_comments}</span>
   <span>{points}</span>
 </div>
);
```

现在，尽管 Item 组件的函数签名更简洁了，但这些杂乱无章的东西最后反而落在了 List 组件中，因为每个属性都是单独传递给 Item 组件的。我们可以使用 JavaScript 的展开运算符[73]来改进这种方法：

Code Playground

```
const profile = {
 firstName: 'Robin',
 lastName: 'Wieruch',
};

const address = {
 country: 'Germany',
 city: 'Berlin',
 code: '10439',
};

const user = {
  ...profile,
  gender: 'male',
  ...address,
};

console.log(user);
// {
//   firstName: "Robin",
//   lastName: "Wieruch",
//   gender: "male"
//   country: "Germany,
//   city: "Berlin",
//   code: "10439"
// }
```

JavaScript 的展开运算符允许我们将一个对象的所有键/值对分散到另一个对象上。这也可以在 React 的 JSX 中实现。我们可以使用 JavaScript 的展开运算符来将对象的所有键/值对作为属性/值对传递给 JSX 元素，而不是像以前那样通过 props 从 List 组件到 Item 组件一次一个地传递每个属性：

[73] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

src/App.js

```
// Variation 2: Spread and Rest Operators
// 2. Iteration

const List = ({ list }) =>
 list.map(item => <Item key={item.objectID} {...item} />);

const Item = ({ title, url, author, num_comments, points }) => (
 <div>
  <span>
   <a href={url}>{title}</a>
  </span>
  <span>{author}</span>
  <span>{num_comments}</span>
  <span>{points}</span>
 </div>
);
```

这个重构使得从 List 组件到 Item 组件传递信息的过程更加简洁。最后，我们用 JavaScript 的剩余参数[74]锦上添花。JavaScript 的剩余操作符总是出现在一个对象解构的最后：

Code Playground

```
const user = {
 id: '1',
 firstName: 'Robin',
 lastName: 'Wieruch',
 country: 'Germany',
 city: 'Berlin',
};

const { id, country, city, ...userWithoutAddress } = user;

console.log(userWithoutAddress);
// {
//   firstName: "Robin",
//   lastName: "Wieruch"
// }

console.log(id);
// "1"
```

[74] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters

```
console.log(city);
// "Berlin"
```

尽管两者的语法相同 (都是三个点)，但剩余运算符不应该与展开运算符相混淆。剩余运算符发生在解构的右侧，而展开运算符发生在左侧。剩余运算符总是用来将一个对象和它的某些属性分开。

现在可以在 List 组件中使用它来将 `objectID` 从 item 中分离出来，因为 `objectID` 只用作 `key`，不用于 Item 组件。只有剩下的项作为属性/值对分散到 Item 组件中（如前所述）：

src/App.js

```
// Variation 2: Spread and Rest Operators (final)

const List = ({ list }) =>
  list.map(({ objectID, ...item }) => <Item key={objectID} {...item} />);

const Item = ({ title, url, author, num_comments, points }) => (
 <div>
   <span>
     <a href={url}>{title}</a>
   </span>
   <span>{author}</span>
   <span>{num_comments}</span>
   <span>{points}</span>
 </div>
);
```

在这个最后的变体中，剩余运算符被用来从 `item` 对象的其余部分中解构 `objectID`。之后，`item` 与它的键/值对被分散到 Item 组件中。虽然这个最后的变体非常简洁，但它包含了一些可能有些人不知道的高级 JavaScript 功能。

在这一节中，我们学习了 JavaScript 的对象解构，这种方法通常用于 `props` 对象，也可以用于其他对象，比如 `item` 对象。我们也学习了如何使用嵌套解构（变体1），但在我们的例子中，它并没有增加任何好处，因为它只是让组件变大了。在未来，你会发现更多类似的嵌套解构的实用用例。最后但并非不重要的一点，你已经了解了 JavaScript 的展开运算符和剩余运算符，这两个运算符不应该相互混淆，它们可以在 JavaScript 对象上执行，并以最简洁的方式将 `props` 对象从一个组件传递给另一个组件。最后，我想再次指出，在接下来的章节中我们将保留这个最初的版本：

src/App.js

```
const List = ({ list }) =>
 list.map(item => <Item key={item.objectID} item={item} />);

const Item = ({ item }) => (
 <div>
  <span>
   <a href={item.url}>{item.title}</a>
  </span>
  <span>{item.author}</span>
  <span>{item.num_comments}</span>
  <span>{item.points}</span>
 </div>
);
```

它可能不是最简洁的，但却是最容易理解的。变体1的嵌套解构并没有增加太多好处，而变体2可能会增加太多高级的 JavaScript 功能（展开运算符、剩余运算符），而这些功能并不是每个人都熟悉。毕竟，所有这些变体都有利有弊。在重构一个组件时，一定要以可读性为目标，尤其是在一个团队中工作时，要确保他们使用的是通用的 React 代码风格。

练习

- 检查上一节的源码[75]。
  - 确认上一节之后的变更[76]。
- 阅读更多关于 JavaScript 的解构赋值[77]的信息。
- 思考 JavaScript 的数组解构 – 被用在 React 的 `useState` hook 中 – 和对象解构之间的区别。
- 阅读更多关于 JavaScript 的展开运算符[78]的信息。
- 阅读更多关于 JavaScript 的剩余参数[79]的信息。
- 从上一节课中了解一下 JavaScript （展开运算符，剩余参数，解构）以及与 React 相关的知识（例如 props）。
- 继续用你喜欢的方式去处理 React 的 props。如果你依然不确定使用哪种方式，考虑一下在上一节中的版本。

[75] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Props-Handling
[76] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-Controlled-Components...hs/Props-Handling?expand=1
[77] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
[78] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax
[79] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters

React 副作用

接下来我们要给 Search 组件增加一个新的功能，让 Search 组件记住最近一次搜索操作，这样每次重启应用之后，应用就会在浏览器里把它打开。

首先，用浏览器的本地存储来存储 `searchTerm` 和它的标识符。然后，用存储的值，如果存在的话，作为 `searchTerm` 的初始 state。否则就和之前一样，使用我们的初始 state（这里指“React”）作为默认的初始值：

src/App.js

```
const App = () => {
 ...

 const [searchTerm, setSearchTerm] = React.useState(
   localStorage.getItem('search') || 'React'
 );

 const handleSearch = event => {
   setSearchTerm(event.target.value);

   localStorage.setItem('search', event.target.value);
 };

 ...
);
```

当用户使用输入框然后刷新浏览器标签页的时候，浏览器应该可以记住最后一个搜索项。在 React 里使用本地存储可以被看作是一种副作用，因为我们跨出了 React 的领域去和浏览器的 API 产生了互动。

不过这里还有个问题。处理函数应该只关心如何更新 state，但它现在还有一个副作用。如果我们在应用的其他地方调用 `setSearchTerm` 就可能会破坏已有功能，因为无法确定本地缓存是否也会被更新。我们可以通过把对副作用的处理固定在某个地方来解决这个问题。这里将在每次 `searchTerm` 发生变化的时候，用到 **React** 的 **useEffect Hook** 来触发副作用：

src/App.js

```
const App = () => {
 ...

 const [searchTerm, setSearchTerm] = React.useState(
   localStorage.getItem('search') || 'React'
 );

 React.useEffect(() => {
   localStorage.setItem('search', searchTerm);
 }, [searchTerm]);

 const handleSearch = event => {
   setSearchTerm(event.target.value);
 };

 ...
);
```

React 的 useEffect Hook 需要两个参数：第一个参数是一个会产生副作用的函数。在我们这个例子里，副作用是指当用户输入搜索词 `searchTerm` 并存进浏览器的本地存储。第二个参数是所依赖的变量数组。如果任何一个依赖的变量发生变化，包含副作用的这个函数就会被调用。对于我们来说，每次 `searchTerm` 改变都会调用这个函数；当组件第一次渲染的时候，它也会被初始化调用。

如果 useEffect 的依赖数组是个空数组，那么这个管理副作用的函数只会被执行一次，也就是在组件第一次渲染之后。这个 hook 使我们可以有选择性地使用 React 的组件生命周期。它会在组件第一次挂载的时候被触发，也可以在它的依赖更新时被触发。

使用 React 的 `useEffect` 而不是处理函数来管理副作用，会让我们的应用更加健壮。无论何时何地使用 `setSearchTerm` 更新 `searchTerm`，本地存储都会同步更新。

练习

- 检查上一节的源码[80]。
- 确认上一节之后的变更[81]。
- 阅读更多关于 React 的 useEffect Hook 的文章：(0[82], 1[83])。
- 在第一个函数参数里使用 `console.log()` 来测试 React useEffect Hook 的依赖数组。也看一下依赖数组为空的情况下，日志是什么样的。

[80] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Side-Effects

[81] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Props-Handling...hs/React-Side-Effects?expand=1

[82] https://reactjs.org/docs/hooks-effect.html

[83] https://reactjs.org/docs/hooks-reference.html#useeffect

自定义 React Hook（高级）

到这里我们已经讲到了 React 里两个最常用的 hook：useState 和 useEffect。useState 用来使你的应用可交互；useEffect 可以让组件有选择地使用生命周期。

最终我们会讲到更多 React 自带的 hook （在这里或是其他材料里），但肯定不会涵盖所有的。接下来我们要来对付 自定义的 **React Hook**；也就是自己来打造一个 hook。

我们将用到两个已有的 hook 来创建一个新的自定义 hook 叫做 `useSemiPersistentState`，这样命名的原因是它不但管理 state 还可以和本地存储同步。它不是一个完全的持久化，因为清除浏览器的本地存储会将应用相关的数据都删掉。我们可以从把所有相关的实现细节从 App 组件里抽出来，放到这个新的自定义 hook 里开始：

```
const useSemiPersistentState = () => {
 const [searchTerm, setSearchTerm] = React.useState(
   localStorage.getItem('search') || ''
 );

 React.useEffect(() => {
   localStorage.setItem('search', searchTerm);
 }, [searchTerm]);
};

const App = () => {
...
};
```

到这里，我们只是把之前在 App 组件里用到 `useState` 和 `useEffect` 的部分用函数把包装起来了而已。在被调用之前，还需要让它把 App 组件需要用到的值返回出来：

src/App.js

```
const useSemiPersistentState = () => {
 const [searchTerm, setSearchTerm] = React.useState(
   localStorage.getItem('search') || ''
 );

 React.useEffect(() => {
   localStorage.setItem('search', searchTerm);
 }, [searchTerm]);

 return [searchTerm, setSearchTerm];
};
```

这里我们遵循了 React 原生 hook 的两个惯例。首先是命名规则，每个 hook 都使用“use”作为前缀。其次是把返回值放在一个数组里。现在我们可以在 App 组件里使用自定义 hook 及其返回值，和往常一样用数组解构的方式：

src/App.js

```
const App = () => {
 const stories = [ ... ];

 const [searchTerm, setSearchTerm] = useSemiPersistentState();

 const handleSearch = event => {
   setSearchTerm(event.target.value);
 };

 const searchedStories = stories.filter(story =>
   story.title.toLowerCase().includes(searchTerm.toLowerCase())
 );

 return (
   ...
 );
};
```

自定义 hook 的另外一个目标应该是可复用性。目前这个自定义 hook 的内部看起来还是搜索相关的，但它应该是用来管理 state 里的值并将它同步到本地存储的。那么让我们来重新命一下名：

```
const useSemiPersistentState = () => {
 const [value, setValue] = React.useState(
   localStorage.getItem('value') || ''
 );

 React.useEffect(() => {
   localStorage.setItem('value', value);
 }, [value]);

 return [value, setValue];
};
```

我们用自定义 hook 处理的是一个抽象的值“value”。在 App 组件里，我们可以在数组解构的过程中，使用有业务含义的词来命名它所返回的当前 state 和 state 更新函数，例如 `searchTerm` 和 `setSearchTerm`。

这个自定义 hook 还有一个问题。如果在同一个 React 应用中多次使用到它的时候，会造成本地存储里的值被覆盖。我们可以通过传一个键值 `key`，来解决这个问题：

src/App.js

```
const useSemiPersistentState = key => {
 const [value, setValue] = React.useState(
   localStorage.getItem(key) || ''
 );

 React.useEffect(() => {
   localStorage.setItem(key, value);
 }, [value, key]);

 return [value, setValue];
};

const App = () => {
 ...

 const [searchTerm, setSearchTerm] = useSemiPersistentState(
   'search'
 );

 ...
};
```

因为键值来自外部，自定义 hook 假设它可能会发生变化，所以要把它放到 `useEffect` 的依赖数组里。如果不做这一步，当键值在不同的渲染之间产生了变化的时候，副作用就可能会使用过期的键值。

我们还可以进一步改进它，让自定义 hook 从外部接受一个初始 state：

src/App.js

```
const useSemiPersistentState = (key, initialState) => {
 const [value, setValue] = React.useState(
   localStorage.getItem(key) || initialState
 );

 ...
};

const App = () => {
 ...
```

```
 const [searchTerm, setSearchTerm] = useSemiPersistentState(
   'search',
   'React'
 );

 ...
};
```

这样你就创建了你的第一个自定义 hook。如果你还没准备好开始使用自定义 hook，那么你可以还原代码，然后像以前一样在 App 组件里使用 `useState` 和 `useEffect`。

然而，了解更多关于自定义 hook 的知识会给你更多的选择。自定义 hook 可以把复杂的实现细节包装起来，从而使组件的逻辑看起来更加清晰；还可以被复用到更多的 React 组件中；甚至可以作为一个三方库开源出去。用喜欢的搜索引擎搜索一下你就会发现，已经有成百上千个 React hook 供你使用，而不需要关心它们是如何实现的。

练习

- 检查上一节的源码[84]。
- 确认上一节之后的变更[85]。
- 阅读更多关于 React Hooks[86] 的文章来深入理解这个概念。它们和 React 函数组件的关系，就如同豆浆和油条，理解它们对你来说是至关重要的。(0[87], 1[88])

[84] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Custom-Hooks
[85] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-Side-Effects...hs/React-Custom-Hooks?expand=1
[86] https://www.robinwieruch.de/react-hooks
[87] https://reactjs.org/docs/hooks-overview.html
[88] https://reactjs.org/docs/hooks-custom.html

React Fragments

JSX 里有个需要注意的地方，尤其当我们创建专门的 Search 组件时，必须引入一个 HTML 元素把它包起来才能渲染：

```
const Search = ({ search, onSearch }) => (
 <div>
  <label htmlFor="search">Search: </label>
  <input
   id="search"
   type="text"
   value={search}
   onChange={onSearch}
  />
 </div>
);
```

通常一个 React 组件返回的 JSX 只能有一个顶层元素。如果需要渲染多个同级的顶层元素，就必须把它们放在数组里。因为是一组元素，我们又必须给每个兄弟元素加上 key 属性：

src/App.js

```
const Search = ({ search, onSearch }) => [
 <label key="1" htmlFor="search">
  Search:{' '}
 </label>,
 <input
  key="2"
  id="search"
  type="text"
  value={search}
  onChange={onSearch}
 />,
];
```

这是一种解决 JSX 里同时需要多个顶层元素的方式。但看起来可读性并不好，而且需要一个额外的 key 属性，显得很冗余。另外一种解决方式就是使用 **React fragment**：

src/App.js

```
const Search = ({ search, onSearch }) => (
 <>
  <label htmlFor="search">Search: </label>
  <input
   id="search"
   type="text"
   value={search}
   onChange={onSearch}
  />
 </>
);
```

Fragment 可以用来把多个元素用一个顶层元素包起来，又不会产生额外的渲染结果。Search 里的元素：输入框和标签，现在应该都可以显示在浏览器里了。所以如果你想要省略外面包的这层 `<div>` 或 `<span>` 元素，JSX 允许你用一个空的 tag 替换掉它们，这样就不会在渲染出来的 HTML 里引入中间元素了。

练习

- 检查上一节的源码[89]。
- 确认上一节之后的变更[90]。
- 阅读更多关于 React fragments[91] 的内容。

[89] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Fragments
[90] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-Custom-Hooks...hs/React-Fragments?expand=1
[91] https://reactjs.org/docs/fragments.html

可复用组件

仔细看一下搜索组件。label 元素有文本值 “Search: “，它的 id 和 htmlFor 属性值均是 `search` 特有的，同时有 `onSearch` 回调处理函数。该组件通常与搜索功能相关联，使得该组件在应用程序的非搜索功能中的可复用性较差。如果将两个搜索组件同时渲染，由于2个组件的 htmlFor 与 id 值是相同的，当用户点击其中一个标签时会破坏焦点，从而引入 bug。

该 Search 组件实际没有任何的” 搜索” 能力，只需花费很少的精力即可覆盖其他通用的搜索场景，以使该组件可在应用程序的其余部分中复用。让我们向 Search 组件传递一个 `id` 和 `label` 参数，将值和回调函数重命名为更抽象的名称，并相应地重命名该组件：

src/App.js

```
const App = () => {
 ...

 return (
   <div>
     <h1>My Hacker Stories</h1>

     <InputWithLabel
       id="search"
       label="Search"
       value={searchTerm}
       onInputChange={handleSearch}
     />

     ...
   </div>
 );
};

const InputWithLabel = ({ id, label, value, onInputChange }) => (
 <>
   <label htmlFor={id}>{label}</label>
    
   <input
     id={id}
     type="text"
     value={value}
     onChange={onInputChange}
   />
 </>
);
```

目前它尚未完全可复用。如果我们希望在 input 中输入数字（number）或电话号码（tel）之类的数据，则需要将 input 的 type 属性从外部传入：

src/App.js

```
const InputWithLabel = ({
 id,
 label,
 value,
 type = 'text',
 onInputChange,
}) => (
 <>
  <label htmlFor={id}>{label}</label>
   
  <input
   id={id}
   type={type}
   value={value}
   onChange={onInputChange}
  />
 </>
);
```

在 App 组件中，没有向 InputWithLabel 组件传递 type 参数，所以 type 没有从外部指定，而是从函数的 默认参数[92]中得到。

只需一些更改，我们就将专用的 Search 组件变成了更可复用的组件。我们对内部实现细节的命名进行了抽象，并为新组件提供了更多的接口，以便可以从外部传递必要的参数。我们还没有在其他地方使用该组件，但是我们增强了它的能力。

练习

- 检查上一节的源码[93]
- 确认上一节之后的变更[94]
- 阅读更多关于复用 React 组件[95]

[92] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters

[93] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Reusable-React-Component

[94] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-Fragments...hs/Reusable-React-Component?expand=1

[95] https://www.robinwieruch.de/react-reusable-components

- 在之前我们使用纯文本 “Search:” 中带着 “:”。你现在将如何处理呢？你会使用 `label="Search:"` 作为参数将其传递给 InputWithLabel 组件呢，还是将其硬编码在 InputWithLabel 组件 `<label htmlFor={id}>{label}:</label>` 中使用？稍后章节我们将学习如何抉择。

React 组件组合

现在我们将要学习如何通过一个开合标签，像 HTML 元素一样使用 React 元素：

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <InputWithLabel
        id="search"
        value={searchTerm}
        onInputChange={handleSearch}
      >
        Search
      </InputWithLabel>

      ...
    </div>
  );
};
```

不像之前使用 `label` prop 那样，而是把“Search”字样放在了组件元素标签之间。在组件 InputWithLabel 里，可以通过 **React** 的 **children** prop 来获取这个信息。不使用 `label` prop，而是用 `children` prop 在需要的地方渲染所有从上层传下来的东西：

src/App.js

```
const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onInputChange,
  children,
}) => (
  <>
    <label htmlFor={id}>{children}</label>
     
    <input
      id={id}
```

```
      type={type}
      value={value}
      onChange={onInputChange}
    />
  </>
);
```

现在 React 组件元素表现的和原生 HTML 很类似了。所有在组件元素之间传递的部分，在组件中都可以通过 `children` 获取到，然后渲染出来。有时当使用一个 React 组件时，你希望在外部有更多自由去决定组件内部应该渲染什么：

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <InputWithLabel
        id="search"
        value={searchTerm}
        onInputChange={handleSearch}
      >
        <strong>Search:</strong>
      </InputWithLabel>

      ...
    </div>
  );
};
```

有了 React 这个特性，我们就可以交织使用 React 组件。我们可以用它来传递一个 JavaScript 字符串，也可以把字符串用 HTML 元素 `<strong>` 包起来，还不止如此，组件也可以被当作 React children 传递。

练习

- 检查上一节的源码[96]。

[96] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Component-Composition

- 确认上一节之后的变更[97]。
- 阅读更多关于 React 组件组合的文章（0[98]，1[99]）。
- 创建一个简单的渲染字符串的文字组件，然后将它作为 `children` 传给 InputWithLabel 组件。

[97] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Reusable-React-Component...hs/React-Component-Composition?expand=1

[98] https://www.robinwieruch.de/react-component-composition

[99] https://reactjs.org/docs/composition-vs-inheritance.html

指令式 React

React 天生是声明式的，从 JSX 开始一直到 hook。我们告诉 React 去渲染什么，而不是如何去渲染。在使用 React 管理副作用的 hook 时（useEffect），我们表述的是什么时候去实现什么事情，而不是如何去实现。然而，有些时候我们希望通过指令式的方式去访问渲染好的 JSX 元素，比如下列这些场景：

- 通过 DOM API 对元素做读写操作：
  - 测量（读）一个元素的宽或高
  - 设置（写）一个输入框的聚焦状态
- 实现更加复杂的动画效果：
  - 设置过场动画
  - 串联多个动画
- 集成三方库：
  - D3[100]：一个常见的指令式图表库

因为 React 中的指令式编程大多数时候是啰嗦且反直觉的，我们这里就只举一个小例子，如何用指令式的方式把焦点设在输入框上。如果用声明式，只需要简单地设置输入框的自动聚焦（autofocus）属性：

```
const InputWithLabel = ({ ... }) => (
 <>
  <label htmlFor={id}>{children}</label>
   
  <input
   id={id}
   type={type}
   value={value}
   autoFocus
   onChange={onInputChange}
  />
 </>
);
```

这样就可以工作了，不过仅限于这些可复用的组件中的一个，且渲染一次的时候。举例来说，如果 App 组件渲染了两个 InputWithLabel 组件，只有最后一个被渲染的组件在它渲染的时候会得到自动聚焦。不过既然这是一个可复用的 React 组件，我们可以传一个特定的属性，让开发者决定他的输入框是否应该被自动聚焦：

[100] https://d3js.org/

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <InputWithLabel
        id="search"
        value={searchTerm}
        isFocused
        onInputChange={handleSearch}
      >
        <strong>Search:</strong>
      </InputWithLabel>

      ...
    </div>
  );
};
```

`isFocused` 和 `isFocused={true}` 是等价的。在组件内部，使用这个新的 prop 作为 `autoFocus` 属性的值：

src/App.js

```
const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onInputChange,
  isFocused,
  children,
}) => (
  <>
    <label htmlFor={id}>{children}</label>
     
    <input
      id={id}
      type={type}
      value={value}
      autoFocus={isFocused}
```

```
    onChange={onInputChange}
   />
  </>
);
```

功能已经有了，但这仍然是声明式的实现方式。我们在告诉 React 去做什么，而不是如何去做。尽管可以用声明式实现，我们还是尝试把这里重构成指令式的。我们希望通过程序调用输入框 DOM API 的 `focus()` 方法，在它被渲染之后：

src/App.js

```
const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onInputChange,
  isFocused,
  children,
}) => {
  // A
  const inputRef = React.useRef();

  // C
  React.useEffect(() => {
    if (isFocused && inputRef.current) {
      // D
      inputRef.current.focus();
    }
  }, [isFocused]);

  return (
    <>
      <label htmlFor={id}>{children}</label>
       
       {/* B */}
      <input
        ref={inputRef}
        id={id}
        type={type}
        value={value}
        onChange={onInputChange}
      />
    </>
```

```
  );
};
```

所有必要的操作都用注释标记出来了，这里会一步一步讲解：

- (A) 首先，使用 **React useRef hook** 创建一个 `ref` （引用）。这个 `ref` 对象是一个持久的值，在整个 React 组件生命周期里不会受到影响。它有一个 `current` （当前）属性，和 `ref` 对象本身正好相反，它是可以被更改的。
- (B) 其次，`ref` 通过 JSX 保留的 `ref` 标签传给输入框，这个元素实例就会被赋给可变的 `current` 属性。
- (C) 再次，使用 React useEffect Hook 进入 React 的生命周期，当组件渲染（或是它的依赖发生改变）时，使焦点落在输入框上。
- (D) 最后，因为 `ref` 被传给了输入框的 `ref` 属性，它的 `current` 属性让我们可以访问到当前元素。通过程序使其聚焦作为一种副作用，但仅当设置了 `isFocused` 并且 `current` 属性存在时。

这只是个如何在 React 中从声明式转换为指令式编程的例子。因为并不是所有时候都能使用声明的方式，所以指令式还是有必要的。本课是为了学习在 React 中如何使用 DOM API。

练习

- 检查上一节的源码[101]。
- 确认上一节之后的变更[102]。
- 阅读更多关于 React useRef Hook[103] 的文章。

[101] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Imperative-React

[102] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-Component-Composition...hs/Imperative-React?expand=1

[103] https://reactjs.org/docs/hooks-reference.html#useref

JSX 中的内联处理函数

到目前为止，我们拥有的 stories 列表只是一个无状态的变量。我们可以使用搜索的功能来筛选渲染的列表，但是如果我们删除过滤器，列表本身将保持不变。过滤器只是通过第三方临时更改，但我们不能操作真实的列表。

为了获得对列表的控制权，把它作为 React 的 useState Hook 中的初始 state，可以让它具有 state。返回值是当前 state（`stories`）和 state 更新函数（`setStories`）。我们没有使用自定义 `useSemiPersistentState` hook，因为我们不想每次都用缓存列表打开浏览器。相反，我们总是想要用初始列表开始。

src/App.js

```
const initialStories = [
  {
    title: 'React',
    ...
  },
  {
    title: 'Redux',
    ...
  },
];

const useSemiPersistentState = (key, initialState) => { ... };

const App = () => {
  const [searchTerm, setSearchTerm] = ...

  const [stories, setStories] = React.useState(initialStories);

  ...
};
```

这应用程序的行为是一样的，因为现在从 `useState` 返回 `stories`，仍然会被过滤到 `searchedStories`，并展示在列表中。接下来，我们将要通过移除列表中的一个 item 来操作列表：

src/App.js

```
const App = () => {
  ...

  const [stories, setStories] = React.useState(initialStories);

  const handleRemoveStory = item => {
    const newStories = stories.filter(
      story => item.objectID !== story.objectID
    );

    setStories(newStories);
  };

  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      ...

      <hr />

      <List list={searchedStories} onRemoveItem={handleRemoveStory} />
    </div>
  );
};
```

App 组件中的回调函数将要删除的 item 作为参数接收，并且通过删除所有不符合条件的 items 来筛选当前 stories。然后将返回的 stories 设置为新的 state，List 组件会传递这个函数给它的子组件。它没有使用这个新的信息，它仅是传递下去：

src/App.js

```
const List = ({ list, onRemoveItem }) =>
 list.map(item => (
  <Item
   key={item.objectID}
   item={item}
   onRemoveItem={onRemoveItem}
  />
 ));
```

最后，我们可以在 Item 组件的另一个处理函数中使用传入函数，将 `item` 传递给它。一个按钮元素可以被用来触发实际的事件。

src/App.js

```
const Item = ({ item, onRemoveItem }) => {
 const handleRemoveItem = () => {
  onRemoveItem(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>
   <span>
    <button type="button" onClick={handleRemoveItem}>
     Dismiss
    </button>
   </span>
  </div>
 );
};
```

我们可能只能传递 item 的 `objectID`，因为这是我们在 App 组件的回调函数中需要的全部，但是我们不确定这个处理函数之后可能需要什么信息。删除一个 item 可能需要的不仅是一个标识符。如果我们调用 `onRemoveItem` 处理函数，这个 item 应该被完整传递，不仅仅是它的标识符。

我们已经使用 React 的 useState Hook，让 stories 列表具有状态。并将搜索到的 stories 作为 List 组件的 props 进行传递。为了能够在各自的组件被使用，实现了回调函数

（handleRemoveStory）和处理函数（handleRemoveItem）。由于处理函数仅是一个函数，在这种情况下它不会返回任何内容，为了完整性，我们可以移除这个块体。

src/App.js

```
const Item = ({ item, onRemoveItem }) => {
  const handleRemoveItem = () =>
    onRemoveItem(item);

  ...
};
```

由于我们在函数组件中积累了处理函数，这会造成我们源代码可读性降低。有时我会在函数组件中，重构处理函数，从箭头函数返回普通函数语句，仅是为了使组件更加容易探索。

src/App.js

```
const Item = ({ item, onRemoveItem }) => {
  function handleRemoveItem() {
    onRemoveItem(item);
  }

  ...
};
```

在本节中，我们使用了 props，处理函数，回调函数和 state。这些都是之前课程学到的。现在我们将追踪内联处理函数，它允许我们在 JSX 中正确执行函数。在 Item 组件中传入函数作为内联处理函数有两种解决方案。首先，使用 JavaScript 的 bind 方法：

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <div>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
    <span>
      <button type="button" onClick={onRemoveItem.bind(null, item)}>
        Dismiss
      </button>
    </span>
  </div>
);
```

在函数上使用 JavaScript 的 bind 方法[104] 允许我们直接绑定参数，在函数执行的时候使用它。bind 的方法返回一个附加了绑定参数的新函数。

第二个更为流行的解决方案是使用一个包装箭头函数，它允许我们潜入像 `item` 类的参数：

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <div>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
    <span>
      <button type="button" onClick={() => onRemoveItem(item)}>
        Dismiss
      </button>
    </span>
  </div>
);
```

这是一个快速的解决方案，因为有时我们不想把一个函数组件的简短函数体重构回块体，为了定义一个适当的函数处理在函数签名和返回语句之间。尽管这个方式相比其他更加简洁，但是由于 JavaScript 逻辑可能会被隐藏在 JSX 中，会导致代码难以调试。如果包装箭头函数使用块体而不是简写体来封装超过一行的实现逻辑，它会变得更加冗长。应该避免这种情况：

Code Playground

```
const Item = ({ item, onRemoveItem }) => (
  <div>
    ...
    <span>
      <button
        type="button"
        onClick={() => {
          // do something else

          // note: avoid using complex logic in JSX

          onRemoveItem(item);
        }}
```

[104] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind

```
    >
      Dismiss
    </button>
  </span>
 </div>
);
```

可以接受三个处理函数版本中，其中两个是内联和一个常规处理函数。非内联的处理函数会移动详细的实现进入函数组件块体，内联函数会移动详细的实现到 JSX 中。

练习

- 检查 上一节的源代码[105]。
- 确认 上一节之后的变更[106]。
- 复习处理函数，回调函数和内联处理函数。

[105] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Inline-Handler-in-JSX
[106] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Imperative-React...hs/Inline-Handler-in-JSX?expand=1

React 异步数据

在这个应用中我们有两种交互方式：搜索列表以及从列表中删除项目。第一个交互是通过应用在列表中的第三方 state（`searchTerm`）产生的影响；第二种交互是不可撤销的从列表中删除项目。

有的时候我们必须先渲染出一个组件，然后再请求第三方 API 得到数据并将其显示在组件上。接下来我们将在应用中模拟这种异步数据过程。在真实应用里，异步数据是从真正的远端 API 中获取的。我们从一个简单返回 promise 的函数出发，一直到数据被解析完成结束。被解析的对象数据包括了之前的 stories 列表数据：

src/App.js

```
const initialStories = [ ... ];

const getAsyncStories = () =>
  Promise.resolve({ data: { stories: initialStories } });
```

在 APP 组件中，使用一个空数组作为初始的 state，而不是使用 `initialStories`。我们希望从一个空的 stories 列表开始，并模拟异步获取这些 stories 的数据的过程。在新的 `useEffect` hook 中，调用方法并返回被解析的 promise。由于 useEffect 的依赖数组是个空数组，因此管理副作用的函数仅在组件首次渲染后运行：

src/App.js

```
const App = () => {
  ...

  const [stories, setStories] = React.useState([]);

  React.useEffect(() => {
    getAsyncStories().then(result => {
      setStories(result.data.stories);
    });
  }, []);

  ...
};
```

虽然我们启动应用的时候数据并非同时到达，但由于它是立刻渲染的，使数据看起来似乎是同步到达的。因为每个对 API 的网络请求都会带来延迟，所以现在让我们给它加上一个真实的延迟。首先，删除之前的简写版本：

src/App.js

```
const getAsyncStories = () =>
  new Promise(resolve =>
    resolve({ data: { stories: initialStories } })
  );
```

其次在解析 promise 时，将其延迟几秒钟：

src/App.js

```
const getAsyncStories = () =>
  new Promise(resolve =>
    setTimeout(
      () => resolve({ data: { stories: initialStories } }),
      2000
    )
  );
```

再次启动应用之后，你应该能看到列表数据会延迟几秒后再渲染出来。而由于初始的 stories 是个空数组，在 APP 组件渲染之后副作用 hook 会运行一次以获取异步数据。在解析 promise 以及将数据设置为组件的 state 之后，组件会再次渲染并显示异步加载的 stories 列表。

练习

- 检查上一节的源码[107]。
- 检查上一节之后的变更[108]。
- 阅读更多关于JavaScript Promises[109]。

[107] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Asynchronous-Data

[108] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Inline-Handler-in-JSX...hs/React-Asynchronous-Data?expand=1

[109] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

React 条件渲染

在 React 中处理异步数据使我们处于条件状态：有数据，无数据。这种情况已经覆盖，由于我们的初始 state 是一个空列表而不是 `null`，如果是 `null`，我们则必须在 JSX 中处理这个问题。但是，由于它是 `[]`，我们对用于搜索的空数组进行了 `filter()` 处理，依然返回一个空数组。这导致在 List 组件的 `map()` 函数中不渲染任何内容。

但在实际的应用程序中，异步数据通常有两个以上的条件状态。考虑延迟数据加载时向用户显示加载指示器：

src/App.js

```
const App = () => {
  ...

  const [stories, setStories] = React.useState([]);
  const [isLoading, setIsLoading] = React.useState(false);

  React.useEffect(() => {
    setIsLoading(true);

    getAsyncStories().then(result => {
      setStories(result.data.stories);
      setIsLoading(false);
    });
  }, []);

  ...
};
```

使用JavaScript 三元运算符[110], 我们可以将该条件状态在 JSX 中内联为条件渲染

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      ...

      <hr />
```

[110] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

```
      {isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List
          list={searchedStories}
          onRemoveItem={handleRemoveStory}
        />
      )}
    </div>
  );
};
```

异步数据有错误处理，在我们的模拟环境中不会发生这种情况，但是如果我们开始从另一个第三方 API 获取数据，则可能会出错。引入另一个 state 来处理错误状态，并在 Promise 完成时在 Promise 的 `catch()` 中处理它。

src/App.js

```
const App = () => {
  ...

  const [stories, setStories] = React.useState([]);
  const [isLoading, setIsLoading] = React.useState(false);
  const [isError, setIsError] = React.useState(false);

  React.useEffect(() => {
    setIsLoading(true);

    getAsyncStories()
      .then(result => {
        setStories(result.data.stories);
        setIsLoading(false);
      })
      .catch(() => setIsError(true));
  }, []);

  ...
};
```

接下来，为用户提供反馈，以防其他条件渲染出现问题。这次，它要么渲染一些内容，要么什么都不渲染。用 `&&` 简写的形式来避免在三元运算符的另一测返回 `null`。

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      ...

      <hr />

      {isError && <p>Something went wrong ...</p>}

      {isLoading ? (
        <p>Loading ...</p>
      ) : (
        ...
      )}
    </div>
  );
};
```

在 JavaScript 中，`true && 'Hello World'` 总是返回 'Hello world'。`false && 'Hello World'` 总是返回 false。在 React 中，我们可以利用这种行为来获得优势。如果条件为 true，则逻辑 `&&` 运算符之后的表达式将作为输出。如果条件是 false，React 将忽略它并跳过表达式。

条件渲染不仅适用于异步数据，最简单的例子是使用按钮切换一个布尔标志的 state，如果是 true，则渲染某些内容，如果是 false，则不渲染任何内容。

这个功能非常强大，因为它使您能够有条件地呈现 JSX。它是 React 中另一个使您的 UI 更加动态的工具。正如我们所发现的，对于异步数据等更复杂的控制流而言，通常是必需的。

练习

- 检查上一节的源码[111]。
- 确认上一节之后的变更[112]。
- 阅读更多React 条件渲染[113]。

[111] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Conditional-Rendering
[112] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-Asynchronous-Data...hs/React-Conditional-Rendering?expand=1
[113] https://www.robinwieruch.de/conditional-rendering-react/

React 状态进阶

虽然在这个应用中的所有状态管理都大量使用了 React 中的 useState hook。但是更复杂的 state 管理可以用 React 中的 useReducer hook。由于 JavaScript 中的 reducers 概念在社区中已有半壁江山，因此我们不会在这赘述，但在这一节的末尾会给你提供大量的练习。

我们会用新的 `useReducer` hook 替换 `useState` hook 来管理 `stories` state。首先，在你的组件外部引入一个 reducer 函数。一个 reducer 函数总是接收 `state` 和 `action`。基于这两个参数，reducer 再返回一个新的 state。

src/App.js

```
const storiesReducer = (state, action) => {
  if (action.type === 'SET_STORIES') {
    return action.payload;
  } else {
    throw new Error();
  }
};
```

通常来说，一个 reducer 的 `action` 会关联一个 `type`。如果这个 type 与 reducer 中的条件吻合，就可以继续我们的操作。如果 reducer 中并不包含这个条件，那就抛出一个错误提醒自己这个实现没有被覆盖。`storiesReducer` 函数包含一个 type，并且没有使用当前的状态来计算出一个新的 state，而是直接返回入参 action 的 `payload`。那新的 state 就自然是 `payload` 了。

在 App 组件中，用 `useReducer` 替换 `useState` 来进行 `stories` 的状态管理。新的 hook 接收一个 reducer 函数和一个初始 state 作为参数，并且返回一个包含两项内容的数组。数组的第一项是当前的 state，第二项是 state 的更新函数 (也称 dispatch 函数)。

src/App.js

```
const App = () => {
  ...

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    []
  );

  ...
};
```

新的 dispatch 函数可以用来代替 `useState` 中返回的 `setStories` 函数。与 `useState` 中直接设置 state 的更新函数不同，`useReducer` state 更新函数会为 reducer 匹配一个 action。action 会包含一个 `type` 和一个可选的 payload。

src/App.js

```
const App = () => {
  ...

  React.useEffect(() => {
    setIsLoading(true);

    getAsyncStories()
      .then(result => {
        dispatchStories({
          type: 'SET_STORIES',
          payload: result.data.stories,
        });
        setIsLoading(false);
      })
      .catch(() => setIsError(true));
  }, []);

  ...

  const handleRemoveStory = item => {
    const newStories = stories.filter(
      story => item.objectID !== story.objectID
    );

    dispatchStories({
      type: 'SET_STORIES',
      payload: newStories,
    });
  };

  ...
};
```

尽管我们现在是用 reducer 和 React 中的 `useReducer` hook 来管理 stories 的 state，但是在浏览器中的表现形式还是和之前一样的。现在我们就带入 reducer 的理念来处理多个 state 转换以实现一个轻量级的版本。

到目前为止，都是 `handleRemoveStory` 处理函数计算出新的 stories。把这段逻辑移进 reducer 函数，并且以 action 来管理 reducer 是可行的，这是从命令式编程到声明式编程的另一种情况。我们只需告诉 reducer 要做什么功能而不是自己去实现。除此之外的其他内容都封装在 reducer 中。

src/App.js

```
const App = () => {
  ...

  const handleRemoveStory = item => {
    dispatchStories({
      type: 'REMOVE_STORY',
      payload: item,
    });
  };

  ...
};
```

现在，reducer 函数必须覆盖在新的条件下满足 state 转换这种场景。如果移除 story 的条件符合，reducer 也包含所有移除 story 的实现细节，action 给出所有必要的信息：每一个 story 的标识符，那就可以在当前的 state 中移除对应的 story 并且返回一个新的 stories 筛选列表作为 state。

src/App.js

```
const storiesReducer = (state, action) => {
  if (action.type === 'SET_STORIES') {
    return action.payload;
  } else if (action.type === 'REMOVE_STORY') {
    return state.filter(
      story => action.payload.objectID !== story.objectID
    );
  } else {
    throw new Error();
  }
};
```

所有的 if else 语句最终会随着在一个 reducer 函数中添加更多 state 变化而变得杂乱。用 switch 来重构一下 state 转化语句，可读性会比较高：

src/App.js

```
const storiesReducer = (state, action) => {
  switch (action.type) {
    case 'SET_STORIES':
      return action.payload;
    case 'REMOVE_STORY':
      return state.filter(
        story => action.payload.objectID !== story.objectID
      );
    default:
      throw new Error();
  }
};
```

我们已经介绍了 JavaScript 中最简版本的 reducer。它涉及了两个 state 的改变，并展示了如何用当前 state 和 action 来计算生成新的 state，并且使用了一些业务逻辑 (删除 story)。现在我们可以将异步获取的 story 列表数据设置为 state，并且只用一个 state 管理的 reducer 结合 `useReducer` hook 就可以删除列表中的 story。

为了全面掌握 JavaScript 中 reducers 的概念以及 React 中 useReducer hook 的用法，可以访问练习中给出的链接。

练习

- 检查 上一节的源代码[114].
- 确认 上一节之后的变更[115].
- 了解更多JavaScript 中的 reducers[116].
- 了解更多 React 中的 reducers 和 useReducer(0[117], 1[118]).

[114] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Advanced-State

[115] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-Conditional-Rendering...hs/React-Advanced-State?expand=1

[116] https://www.robinwieruch.de/javascript-reducer

[117] https://www.robinwieruch.de/react-usereducer-hook

[118] https://reactjs.org/docs/hooks-reference.html#usereducer

不合理的状态

可能你已经注意到了在 App 组件里每个 state 之间是没有关联的，虽然看起来好像应该是属于一起的，因为都使用了 `useState` hook 的缘故。技术上来讲，所有和异步数据相关的 state 应该是统一的，不仅仅是 stories 作为实际的数据，还应包括它的加载和错误状态。

在一个 React 组件中出现多个 `useState` 本身没什么问题，但要警惕连续使用 state 更新函数。这些条件状态可能会导致不合理的状态，以及在 UI 上意想不到的行为。我们试着把伪数据获取函数改成下面这样，来模拟错误处理：

src/App.js

```
const getAsyncStories = () =>
  new Promise((resolve, reject) => setTimeout(reject, 2000));
```

不合理的状态会出现在获取异步数据出错的时候。保存 error 的 state 更新了，但标识 loading 的 state 没被取消。在 UI 上，这会让用户同时看到无休止的 loading 标识和一条错误信息；比较好的处理方式应该只给用户展示错误信息，隐藏 loading 标识。不合理的状态很难定位到，却经常引发 UI 上的问题，这也是它臭名远扬的原因。

所幸的是，我们可以通过把分散在多个 `useState` 和 `useReducer` 里但是同属一体的状态，挪到一个 `useReducer` hook 里，来减少出错的机率。找到下面这些 `useState` hooks：

src/App.js

```
const App = () => {
  ...

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    []
  );
  const [isLoading, setIsLoading] = React.useState(false);
  const [isError, setIsError] = React.useState(false);

  ...
};
```

把它们合并到一个 `useReducer` hook 里，用一个更复杂的 state 对象，作为统一管理 state 的方式：

src/App.js

```
const App = () => {
  ...

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    { data: [], isLoading: false, isError: false }
  );

  ...
};
```

所有和异步数据读取相关的操作必须通过这个新的 dispatch 函数来更新 state：

src/App.js

```
const App = () => {
  ...

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    { data: [], isLoading: false, isError: false }
  );

  React.useEffect(() => {
    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    getAsyncStories()
      .then(result => {
        dispatchStories({
          type: 'STORIES_FETCH_SUCCESS',
          payload: result.data.stories,
        });
      })
      .catch(() =>
        dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
      );
  }, []);

  ...
};
```

因为这里引入了新的类型来转换 state，所以在 `storiesReducer` 里需要处理它们：

src/App.js

```
const storiesReducer = (state, action) => {
  switch (action.type) {
    case 'STORIES_FETCH_INIT':
      return {
        ...state,
        isLoading: true,
        isError: false,
      };
    case 'STORIES_FETCH_SUCCESS':
      return {
        ...state,
        isLoading: false,
        isError: false,
        data: action.payload,
      };
    case 'STORIES_FETCH_FAILURE':
      return {
        ...state,
        isLoading: false,
        isError: true,
      };
    case 'REMOVE_STORY':
      return {
        ...state,
        data: state.data.filter(
          story => action.payload.objectID !== story.objectID
        ),
      };
    default:
      throw new Error();
  }
};
```

每次 state 转换，都返回一个新的 state 对象，包含所有当前 state 对象里的键值对（使用 JavaScript 的展开运算符），再用新的属性覆盖。举例来说，`STORIES_FETCH_FAILURE` 重置了 `isLoading`、并设置了 `isError` 布尔标识，但保留了其他的 state，例如 `stories`。这就是如何绕过之前可能会出现的问题：如果发生了错误，应该取消加载状态。

也请仔细观察 `REMOVE_STORY` action 的变化。它需要操作的是 `state.data`，而不再是一个简单的 `state`。现在的 state 是一个复杂的对象，里面有 data、loading 和 error 状态，不仅仅是一个单纯的 stories 列表。在其余的代码里也需要解决这个变化产生的影响：

src/App.js

```
const App = () => {
  ...

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    { data: [], isLoading: false, isError: false }
  );

  ...

  const searchedStories = stories.data.filter(story =>
    story.title.toLowerCase().includes(searchTerm.toLowerCase())
  );

  return (
    <div>
      ...

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List
          list={searchedStories}
          onRemoveItem={handleRemoveStory}
        />
      )}
    </div>
  );
};
```

再试着用抛错的数据获取函数，检查是否所有功能都能正常工作：

src/App.js

```
const getAsyncStories = () =>
  new Promise((resolve, reject) => setTimeout(reject, 2000));
```

我们从多个 `useState` 带来的不可靠的 state 转换脱离出来，转而使用 React 的 `useReducer` hook 来实现可预见的 state 转换。这个用 reducer 管理的 state 对象囊括了所有关于 stories

的数据，包括加载中和错误状态，也实现了如何从 stories 列表中删除特定 story 这样的功能。我们没有完全摆脱掉不合理的状态，因为始终都有可能遗漏某个关键的布尔值，但我们向更可预见的状态管理迈进了一步。

练习

- 检查上一节的源码[119]。
- 确认上一节之后的变更[120]。
- 阅读教程里之前提到的关于 JavaScript 和 React 中的 reducer 的章节。
- 阅读更多关于该如何选用 useState 和 useReducer[121] 的文章。

[119] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Impossible-States

[120] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-Advanced-State...hs/React-Impossible-States?expand=1

[121] https://www.robinwieruch.de/react-usereducer-vs-usestate

React 获取数据

我们目前正在获取数据，但它仍然是来自我们自己用 Promise 设置的假数据。到现在为止关于异步 React 和高级 State 管理的课程都在为我们从真正的第三方 API 中获取数据做准备。我们将使用可靠且丰富的 Hacker News API[122] 来请求流行的科技新闻。

我们将直接从 API 中获取数据，而不是使用 `initialStories` 数组和 `getAsyncStories` 函数(你可以删除这些)。

src/App.js

```
// A
const API_ENDPOINT = 'https://hn.algolia.com/api/v1/search?query=';

const App = () => {
  ...

  React.useEffect(() => {
    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    fetch(`${API_ENDPOINT}react`) // B
      .then(response => response.json()) // C
      .then(result => {
        dispatchStories({
          type: 'STORIES_FETCH_SUCCESS',
          payload: result.hits, // D
        });
      })
      .catch(() =>
        dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
      );
  }, []);

  ...
};
```

首先，`API_ENDPOINT` (A) 用于获取热门科技新闻的某个查询 (搜索主题)。在本示例中，我们获取 React 相关的新闻。其次，用浏览器原生的 fetch API[123] 来执行这个请求 (B), 对于 fetch API，需要将返回数据翻译成 JSON (C)。最后，返回的结果遵循不同的数据结构 (D)，我们将其作为有效数据发送给我们的组件 state。

[122] https://hn.algolia.com/api
[123] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

在上面的代码示例中，我们使用了 JavaScript 的模版字符串[124] 进行字符串插值。在 JavaScript 中没有这个功能的时候，我们会在字符串上使用 + 运算符来代替。

Code Playground

```
const greeting = 'Hello';

// + operator
const welcome = greeting + ' React';
console.log(welcome);
// Hello React

// template literals
const anotherWelcome = `${greeting} React`;
console.log(anotherWelcome);
// Hello React
```

检查你的浏览器，查看从 Hacker News API 中获取的初始查询的相关内容。由于我们对样本内容使用了相同的数据结构，所以我们不需要改变什么，用搜索功能获取数据后仍然可以对它们进行过滤。我们将在下一节中改变这个行为。对于 App 组件来说，这里没有太多的数据获取需要实现，虽然这都是学习如何在 React 中把异步数据作为 state 管理的一部分。

练习

- 检查上一节的源码[125]
- 确认上一节之后的变更[126]
- 通读 Hacker News[127] 和它的 API[128]
- 阅读更多关于连接到远程 API 的 浏览器原生 fetch API[129]
- 阅读更多关于 JavaScript 模板字符串[130]。

[124] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals
[125] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Data-Fetching-with-React
[126] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/React-Impossible-States...hs/Data-Fetching-with-React?expand=1
[127] https://news.ycombinator.com/
[128] https://hn.algolia.com/api
[129] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
[130] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

React 重新获取数据

至此，App 组件使用预定义的查询条件获取了一次 stories 列表。这之后，用户可以在客户端搜索 stories。现在我们将把这个功能从客户端移到服务端，用实际的 `searchTerm` 作为动态查询条件来请求 API。

首先，移除 `searchedStories`，因为我们将通过 API 搜索获取 stories 。只传递常规的 stories 给 List 组件：

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      ...

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </div>
  );
};
```

第二步，使用组件 state 中实际的 `searchTerm`，而不是像之前那样使用硬编码作为搜索关键词。如果 `searchTerm` 是空字符串，则什么都不做。

src/App.js

```
const App = () => {
  ...

  React.useEffect(() => {
    if (searchTerm === '') return;

    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    fetch(`${API_ENDPOINT}${searchTerm}`)
      .then(response => response.json())
      .then(result => {
        dispatchStories({
```

```
        type: 'STORIES_FETCH_SUCCESS',
        payload: result.hits,
      });
    })
    .catch(() =>
      dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
    );
  }, []);

  ...
};
```

现在按照搜索关键词初始化搜索，因此我们需要实现数据的重新获取。如果 `searchTerm` 改变，再次执行获取数据的副作用函数。如果 `searchTerm` 没有展示（如 null，空字符串，undefined），则什么都不做（作为一种更普遍的状况）：

src/App.js

```
const App = () => {
  ...

  React.useEffect(() => {
    if (!searchTerm) return;

    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    fetch(`${API_ENDPOINT}${searchTerm}`)
      .then(response => response.json())
      .then(result => {
        dispatchStories({
          type: 'STORIES_FETCH_SUCCESS',
          payload: result.hits,
        });
      })
      .catch(() =>
        dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
      );
  }, [searchTerm]);

  ...
};
```

我们将该功能从客户端搜索改为了服务端搜索。使用 `searchTerm` 获取一组被服务端筛选的

列表，而不是在客户端筛选预先定义的列表。服务端的搜索不仅发生在初始数据获取时，同时也在 `searchTerm` 改变时获取数据。该功能现在完全在服务端完成。

每次在输入框中输入内容就重新获取一次数据并不是最佳方案，这一点我们将在稍后修复。因为这种实现会给 API 造成压力，如果你发起太多次的请求，则可能会发生一些错误。

练习

- 检查 上一节的源码[131].
- 确认 上一节之后的变更[132]。

[131] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Data-Re-Fetching-in-React
[132] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Data-Fetching-with-React...hs/Data-Re-Fetching-in-React?expand=1

React 中 Memoized 函数（高级）

前一节我们讲了处理函数、回调函数以及内联函数。现在我们将介绍 **memoized** 处理函数，它可以被用于处理函数和回调函数的最顶层，接下来我们会把所有的数据请求的逻辑移动至副作用外部的一个单独的函数中（A）；然后将其用 `useCallback` Hook 包裹（B）；并且在 `useEffect` Hook 中调用它。

src/App.js

```
const App = () => {
  ...

  // A
  const handleFetchStories = React.useCallback(() => {
    if (!searchTerm) return;

    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    fetch(`${API_ENDPOINT}${searchTerm}`)
      .then(response => response.json())
      .then(result => {
        dispatchStories({
          type: 'STORIES_FETCH_SUCCESS',
          payload: result.hits,
        });
      })
      .catch(() =>
        dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
      );
  }, [searchTerm]); // E

  React.useEffect(() => {
    handleFetchStories(); // C
  }, [handleFetchStories]); // D

  ...
};
```

应用的行为与之前一致，只是重构了一下实现逻辑。与在副作用中使用匿名函数进行请求数据相比，我们将其重构成了一个对应用来说更可用的函数。

让我们一起来探讨一下为什么在这儿需要 React 的 `useCallback` Hook。每当其依赖数组（E）改变时这个 Hook 就会创建一个 memoized 函数。结果，因为 `useEffect` Hook 依赖于新的函数（D）所以它会再次运行（C）：

Visualization

1. change: searchTerm
2. implicit change: handleFetchStories
3. run: side-effect

如果我们没有使用 `useCallback` Hook 创建 memoized 函数的话，每当有一个 App 组件渲染时就会创建一个新的 `handleFetchStories` 方法。`handleFetchStories` 方法每次都会被创建，并且会在 `useEffect` Hook 中执行获取数据，随后获取到的数据会被存在组件的 state 中。因为组件的 state 改变了，所以组件就会重新渲染并创建一个新的 `handleFetchStories` 方法。副作用函数会被触发再次获取数据，这样我们就会陷入一个无限循环中：

Visualization

1. define: handleFetchStories
2. run: side-effect
3. update: state
4. re-render: component
5. re-define: handleFetchStories
6. run: side-effect

...

只有当搜索内容改变时 `useCallback` Hook 才会更改函数。那是当我们想触发重新获取数据时，因为输入字段中有新的输入，并且我们希望看到列表中展示新的数据。

通过将获取数据的函数移动到 `useEffect` Hook 外，让它可以在应用的其他部分复用。我们暂时不会使用它，但这是理解 `useCallback` Hook 的一种方式。现在当 `searchTerm` 改变时，`useEffect` Hook 将会运行，因为每次 `searchTerm` 更改时都会重新定义 `handleFetchStories`。因为 `useEffect` Hook 依赖于 `handleFetchStories`，所以获取数据的副作用会再次运行。

练习

- 检查 上一节源码[133]。
- 确认 上一节之后的变更[134]。
- 阅读更多关于 React 的 useCallback Hook[135]。

[133] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Memoized-Handler-in-React
[134] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Data-Re-Fetching-in-React...hs/Memoized-Handler-in-React?expand=1
[135] https://reactjs.org/docs/hooks-reference.html#usecallback

使用 React 进行显式数据获取

每次在输入框中输入内容就重新获取数据不是最佳的选择。因为我们使用的是第三方 API 去获取数据，频繁的调用可能引发 限速[136]，接口将返回错误。

要解决这个问题，我们把获取数据的实现方式从隐式改为显式。换句话说，仅在点击确认按钮时才重新获取数据。首先，我们添加一个确认按钮：

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <InputWithLabel
        id="search"
        value={searchTerm}
        isFocused
        onInputChange={handleSearchInput}
      >
        <strong>Search:</strong>
      </InputWithLabel>

      <button
        type="button"
        disabled={!searchTerm}
        onClick={handleSearchSubmit}
      >
        Submit
      </button>

      ...
    </div>
  );
};
```

然后，监听输入框、按钮的函数会处理相应逻辑并更新组件状态。输入框的处理函数仍然会用来更新 `searchTerm`，按钮的处理函数会组装 `API_ENDPOINT` 和 最新的 `searchTerm` 并更新到 `url` 状态中：

[136] https://en.wikipedia.org/wiki/Rate_limiting

src/App.js

```
const App = () => {
  const [searchTerm, setSearchTerm] = useSemiPersistentState(
    'search',
    'React'
  );

  const [url, setUrl] = React.useState(
    `${API_ENDPOINT}${searchTerm}`
  );

  ...

  const handleSearchInput = event => {
    setSearchTerm(event.target.value);
  };

  const handleSearchSubmit = () => {
    setUrl(`${API_ENDPOINT}${searchTerm}`);
  };

  ...
};
```

之后，将每次因输入框值改变 `searchTerm(url)` 而导致的数据请求，改为每次用户点击确认按钮来触发：

src/App.js

```
const App = () => {
  ...

  const handleFetchStories = React.useCallback(() => {
    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    fetch(url)
      .then(response => response.json())
      .then(result => {
        dispatchStories({
          type: 'STORIES_FETCH_SUCCESS',
          payload: result.hits,
        });
      })
```

```
      .catch(() =>
        dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
      );
  }, [url]);

  React.useEffect(() => {
    handleFetchStories();
  }, [handleFetchStories]);

  ...
};
```

之前的 searchTerm 有两个用途：更新（存储）输入框的值和触发获取数据，现在它仅用于更新（存储）输入框的值。现在当用户点击确认按钮时，url 状态会被更新并引起相应的副作用以获取新的数据。

练习

- 检查上一节的源码[137].
- 确认上一节之后的变更[138].
- 为什么 url 的状态用 useState 而不是 useSemiPersistentState 来管理?
- 为什么在 handleFetchStories 函数中不再检查空的 searchTerm?

[137] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Explicit-Data-Fetching-with-React

[138] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Memoized-Handler-in-React...hs/Explicit-Data-Fetching-with-React?expand=1

React 中的第三方库

此前，我们引入了原生浏览器提供的 fetch API，完成对 Hacker News API 的请求。然而不是所有浏览器都支持这个 API，特别是一些老版本的浏览器。另外，一旦你开始在无头浏览器环境[139] 中测试你的应用，fetch API 就会出现问题。这里有几种方式可以在旧版本浏览器上使用 fetch polyfills[140]) 和测试（同构式 fetch），但是这些概念对现在来说有点偏离主题。

一个替代方案是用一个稳定的库如 axios[141] 来替代原生 fetch API，它执行对远程 API 的异步请求。在本节中，我们将学习怎样使用 npm 注册表中的一个库来替换浏览器原生的 API。首先，在命令行安装 axios：

Command Line

```
npm install axios
```

然后，将 axios 引入到 App 组件文件里：

src/App.js

```
import React from 'react';
import axios from 'axios';

...
```

使用 `axios` 代替 `fetch`，它的使用看起来与原生 fetch API 几乎相同。它将 URL 作为一个参数，并返回一个 promise。你不需要将返回的结果转换成 JSON，因为 axios 把结果包装成了 JavaScript 的数据对象。你只是需要确保你的代码适配返回的数据结构: {title=”src/App.js”,lang=”javascript”} ~~~ const App = () ⇒ { ...

const handleFetchStories = React.useCallback(() ⇒ { dispatchStories({ type: ‘STORIES_FETCH_-INIT’ });

[139] https://en.wikipedia.org/wiki/Headless_browser
[140] https://en.wikipedia.org/wiki/Polyfill_(programming)
[141] https://github.com/axios/axios

leanpub-start-insert

```
axios
  .get(url)
```

leanpub-end-insert .then(result ⇒ { dispatchStories({ type: 'STORIES_FETCH_SUCCESS', # leanpub-start-insert payload: result.data.hits, # leanpub-end-insert }); }) .catch(() ⇒ dispatchStories({ type: 'STORIES_FETCH_FAILURE' })); }, [url]);

... }; ~~~

在代码中，我们调用 axios `axios.get()` 用于一个显式 HTTP GET 请求[142]，这也是我们在浏览器的原生 fetch API 中默认使用的 HTTP 方法。你可以使用其他 HTTP 方法，比如使用 `axios.post()` 发起 HTTP POST 请求。

通过这些例子，我们可以看到 axios 是一个执行远程 API 请求的强大的库。当请求变得复杂的时候或与老版本浏览器一起工作，或者用于测试时，相比原生 fetch API，我更推荐使用 axios。

练习

- 检查上一节的源码[143]
- 确认上一节之后的变更[144]
- 阅读更多关于 React 中流行的库[145]
- 阅读更多关于 为什么框架重要[146]
- 阅读更多关于 axios[147]

---

[142] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET

[143] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Third-Party-Libraries-in-React

[144] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Explicit-Data-Fetching-with-React...hs/Third-Party-Libraries-in-React?expand=1

[145] https://www.robinwieruch.de/react-libraries

[146] https://www.robinwieruch.de/why-frameworks-matter

[147] https://github.com/axios/axios

React 中的 Async / Await（高阶）

在 React 里经常需要处理异步数据，所以了解 promise 的另外一种语法形式是大有裨益的，也就是：async / await。下面这段对 `handleFetchStories` 的重构展示了如何使用它们，不包含错误处理：

src/App.js

```
const App = () => {
  ...

  const handleFetchStories = React.useCallback(async () => {
    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    const result = await axios.get(url);

    dispatchStories({
      type: 'STORIES_FETCH_SUCCESS',
      payload: result.data.hits,
    });
  }, [url]);

  ...
};
```

想使用 async / await，需要给函数加 `async` 关键字。一旦开始使用 `await` 关键字，代码读起来就像同步的了。`await` 关键字后面的动作会等到 promise 正常返回后才会执行，也就是说代码会等待：

src/App.js

```
const App = () => {
  ...

  const handleFetchStories = React.useCallback(async () => {
    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    try {
      const result = await axios.get(url);

      dispatchStories({
        type: 'STORIES_FETCH_SUCCESS',
        payload: result.data.hits,
      });
```

```
    } catch {
      dispatchStories({ type: 'STORIES_FETCH_FAILURE' });
    }
  }, [url]);

  ...
};
```

如果需要和以前一样加入错误处理，则要用到 `try` / `catch` 语句。如果 `try` 里面出错了的话，代码会跳到 `catch` 里做错误处理。`then` / `catch` 语法和 async / await 里的 `try` / `catch` 语法在 JavaScript 和 React 里处理异步数据是同样有效的。

练习

- 检查上一节的源码[148]。
  - 确认上一节之后的变更[149]。
- 阅读更多关于 React 中的数据获取[150]的文章。
- 阅读更多 JavaScript 中的 async / await[151] 的文章。

[148] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Async-Await-in-React
[149] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Third-Party-Libraries-in-React...hs/Async-Await-in-React?expand=1
[150] https://www.robinwieruch.de/react-hooks-fetch-data
[151] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

React 表单

前一节中我们加入了一个显式获取数据的按钮。接下来，我们将使用 HTML 表单来完善它，将搜索输入框、按钮都封装起来。

表单在 JSX（React）和 HTML 中没有太大差别。我们将分几步进行重构，首先把输入框和按钮都用 `form` 元素包裹起来：

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <form onSubmit={handleSearchSubmit}>
        <InputWithLabel
          id="search"
          value={searchTerm}
          isFocused
          onInputChange={handleSearchInput}
        >
          <strong>Search:</strong>
        </InputWithLabel>

        <button type="submit" disabled={!searchTerm}>
          Submit
        </button>
      </form>

      <hr />

      ...
    </div>
  );
};
```

我们不再把 `handleSearchSubmit` 监听函数绑定在提交按钮上，而是绑在新的 `form` 元素上。另外把提交按钮的 `type` 属性设置为 `submit`，这样 `form` 元素就能处理提交事件。

监听函数用来处理表单事件，我们需要在其中执行 `event.preventDefault()`，来避免浏览器默认行为（这会导致页面重新加载）。

src/App.js

```
const App = () => {
  ...

  const handleSearchSubmit = event => {
    setUrl(`${API_ENDPOINT}${searchTerm}`);

    event.preventDefault();
  };

  ...
};
```

现在，我们可以使用键盘的 `Enter` 键来执行搜索功能。接下来，我们将该组件抽取为独立的 SearchForm 组件：

src/App.js

```
const SearchForm = ({
  searchTerm,
  onSearchInput,
  onSearchSubmit,
}) => (
  <form onSubmit={onSearchSubmit}>
    <InputWithLabel
      id="search"
      value={searchTerm}
      isFocused
      onInputChange={onSearchInput}
    >
      <strong>Search:</strong>
    </InputWithLabel>

    <button type="submit" disabled={!searchTerm}>
      Submit
    </button>
  </form>
);
```

把这个新组件引入到 App 组件中。App 组件仍然替表单组件管理状态，因为 App 组件的状态会通过 props(`stories.data`) 传递给 List 组件使用：

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <SearchForm
        searchTerm={searchTerm}
        onSearchInput={handleSearchInput}
        onSearchSubmit={handleSearchSubmit}
      />

      <hr />

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </div>
  );
};
```

表单在 React 和 HTML 中没有太大差别。当我们通过按钮去提交输入框的数据时，我们可以把这些元素包裹进含有 `onSubmit` 监听器的 `form` 元素中，而提交按钮仅需要设置 `type="submit"`。

练习

- 检查上一节的源码[152]。
  - 确认上一节之后的变更[153]。
- 试试不使用 `preventDefault` 的效果。
- 阅读更多关于 preventDefault for Events in React[154]。
- 阅读更多关于 React Component Composition[155]。

[152] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Forms-in-React

[153] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Async-Await-in-React...hs/Forms-in-React?expand=1

[154] https://www.robinwieruch.de/react-preventdefault

[155] https://www.robinwieruch.de/react-component-composition

React 的遗留问题

自2013年以来，React 发生了很多变化。库的迭代，React 应用的编写方式，尤其是它的组件都发生了翻天覆地的变化。然而，许多 React 应用都是在过去几年中构建的，所以并不是所有的东西都是按照目前的情况来创建的。本书的这一节主要介绍了 React 的遗留问题。

我不会涵盖所有被认为是 React 中的遗留功能，因为有些功能已经不止一次被改造过。你可能会在老的 React 应用中看到这个功能的上一个迭代，但可能会和现在的不同。

在本节中，我们将对一个现代 React 应用[156]和它的遗留 d 版本[157]进行比较。我们会发现，现代 React 和传统 React 之间的大部分差异都是由于类组件和函数组件造成的。

[156] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/react-modern-final
[157] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/react-legacy

React 类组件

React 组件经历了很多变化，从 **createClass components** 到 **class components**，再到 **function components**。今天再翻开 React 应用看看，我们很可能会在现代到函数组件看到类组件。

Code Playground

```
class InputWithLabel extends React.Component {
  render() {
    const {
      id,
      value,
      type = 'text',
      onInputChange,
      children,
    } = this.props;

    return (
      <>
        <label htmlFor={id}>{children}</label>
         
        <input
          id={id}
          type={type}
          value={value}
          onChange={onInputChange}
        />
      </>
    );
  }
}
```

一个典型的类组件是一个带有强制 **render** 方法的 JavaScript 类，该类返回 JSX。该类从 `React.Component` 中扩展出来的，继承（[类的继承](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)）所有的 React 的组件特性（状态的管理，副作用的生命周期方法）。React props 通过类实例（`this`）来访问。

有一段时间，类组件是编写 React 应用的首选。最终，当函数组件被加入后，二者因其不同的目的共存着。

Code Playground

```
const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onInputChange,
  children,
}) => (
  <>
    <label htmlFor={id}>{children}</label>
     
    <input
      id={id}
      type={type}
      value={value}
      onChange={onInputChange}
    />
  </>
);
```

如果在传统的 App 中不使用副作用和 state，我们会使用函数组件而不是类组件。在 2018 年 React Hooks 引入之前，React 的函数组件无法处理副作用（`useEffect` hook）或状态（`useState`/`useReducer` hook）。因此，这些组件被称为无状态函数组件，只能在组件中输入 props 和输出 JSX。为了使用 state 或副作用，必须从函数组件重构到类组件。当状态和副作用都没有使用时，我们就使用类组件或更轻量级都函数组件。

加入 React Hooks 后，函数组件的工作原理和类组件一样，有 state 和副作用。而且由于它们之间已经没有实际的区别，并且它们更轻量级，所以社区选择来函数组件。

练习

- 了解更多 JavaScript 中的类[158]。
- 了解更多 如何将类组件重构成函数组件[159]。
- 了解更多关于 类组件语法[160] 这种语法并不流行，但更有效。
- 了解更多 深入类组件[161]。
- 了解更多 React 中的其它遗留问题和现代组件类型[162]。

[158] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes
[159] https://www.robinwieruch.de/react-hooks-migration
[160] https://github.com/the-road-to-learn-react/react-alternative-class-component-syntax
[161] https://reactjs.org/docs/react-component.html
[162] https://www.robinwieruch.de/react-component-types

命令式的 React

在 React 函数组件中，React 的 useRef Hook 主要用于命令式编程。在 React 的历史上，*ref* 及其用法有几个版本：

- String Refs (已弃用)
- Callback Refs
- createRef Refs (类组件专属)
- useRef Hook Refs (函数式组件专属)

最近，React 团队引入了 **React** 的 **createRef**，作为最新的相当于函数组件的 useRef 钩子：

Code Playground

```
class InputWithLabel extends React.Component {
 constructor(props) {
  super(props);

  this.inputRef = React.createRef();
 }

 componentDidMount() {
  if (this.props.isFocused) {
   this.inputRef.current.focus();
  }
 }

 render() {
  ...

  return (
   <>
    ...
    <input
     ref={this.inputRef}
     id={id}
     type={type}
     value={value}
     onChange={onInputChange}
    />
   </>
  );
```

```
 }
}
```

通过工具函数，在类的构造函数中创建 `ref`，在 JSX 中应用 `ref`，这里用在生命周期方法中使用。`ref` 也可以用在其他地方，比如将输入字段聚焦在按钮点击上。

在 React 的类组件中使用 createRef，而在 React 的函数组件中则使用 React 的 useRef Hook。随着 React 向函数组件的转变，如今其通常的做法是专门使用 useRef 钩子来管理 refs，并应用命令式编程原则。

练习

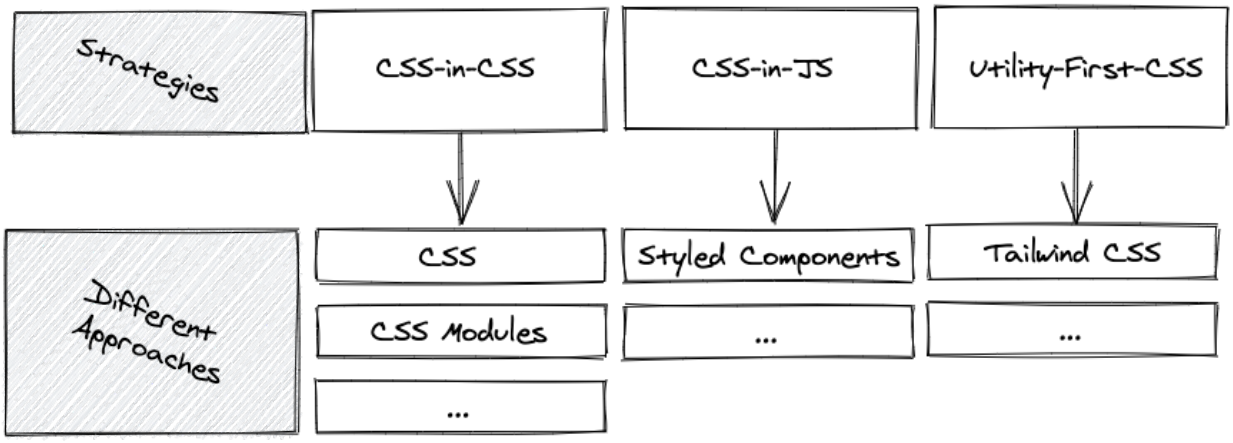
- 了解更多 React 中不同的 Ref 技术[163]。

[163] https://reactjs.org/docs/refs-and-the-dom.html

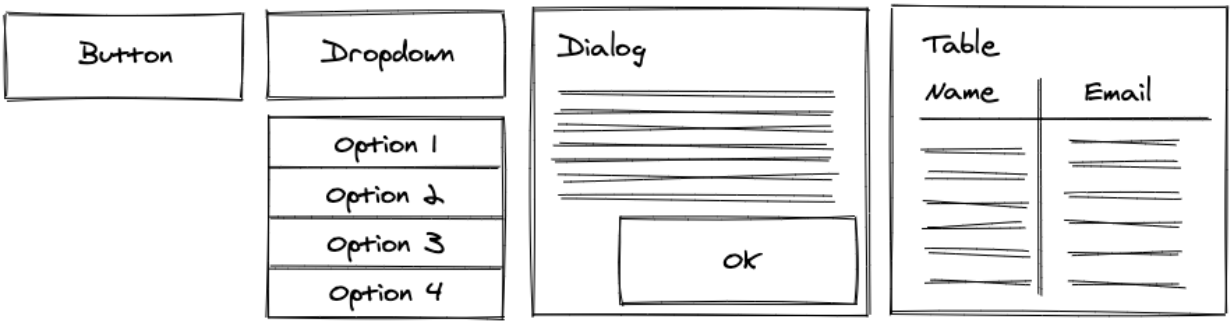
React 中的样式

React 应用中写样式的方法有很多，关于最好的样式策略和样式方法也争论了很久。我们将介绍几种方法，但不会给个权重来说明这些方法哪个更好。目前有正反两方面的论点，但主要是看什么最适合开发者和团队。

我们将从 React 中常见的 CSS 样式开始，但随后我们将探讨两种更高级的 **CSS-in-CSS**（CSS 模块）和 **CSS-in-JS**（样式组件）策略。CSS 模块和样式组件只是这两组策略中的两种方法。我们也将介绍如何在 React 应用程序中包含可扩展的矢量图形（SVG），如 Logo 或图标等。



如果你不想从头开始构建常见的 UI 组件（如按钮、对话框、下拉菜单），你可以选择一个适合 React 流行的 UI 库[164]，它默认提供了这些组件。不过，如果你在使用预构建的解决方案之前，先尝试着构建这些组件，对学习 React 更好。因此，我们将不会使用任何一个 UI 组件库。



以下是在 `create-react-app` 中预先配置的样式化方法和 SVG。如果你使用了构建工具（例如 Webpack），可能需要配置为导入 CSS 或 SVG 文件。因为我们使用的是 create-react-app，

[164] https://www.robinwieruch.de/react-libraries

所以我们可以立即使用这些文件作为资源。

CSS in React

React 中的 CSS

React 中的通用 CSS 与您可能已经学习的标准 CSS 相似。每个 Web 应用程序都为 HTML 元素提供了一个 class 属性（在 React 中为 className），该属性稍后将在 CSS 文件中设置样式。

src/App.js

```
const App = () => {
  ...

  return (
    <div className="container">
      <h1 className="headline-primary">My Hacker Stories</h1>

      <SearchForm
        searchTerm={searchTerm}
        onSearchInput={handleSearchInput}
        onSearchSubmit={handleSearchSubmit}
      />

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </div>
  );
};
```

因为 CSS 在下一步处理了边框，所以删除了 `<hr />`。我们将导入 CSS 文件，这是在 create-react-app 配置的帮助下完成的：

src/App.js

```
import React from 'react';
import axios from 'axios';

import './App.css';
```

此 CSS 文件将定义我们在 App 组件中使用的两个（以及更多）CSS 类。在 * src / App.css * 文件中，像下面这样定义它们：

src/App.css

```
.container {
  height: 100vw;
  padding: 20px;

  background: #83a4d4; /* fallback for old browsers */
  background: linear-gradient(to left, #b6fbff, #83a4d4);

  color: #171212;
}

.headline-primary {
  font-size: 48px;
  font-weight: 300;
  letter-spacing: 2px;
}
```

当再次启动应用程序时，应该会看到第一个样式已经在应用程序中生效。接下来，我们看一下 Item 组件。它的一些元素也会接收 “className” 属性，但是，我们在这里也使用了一种新的样式化技术。

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <div className="item">
    <span style={{ width: '40%' }}>
      <a href={item.url}>{item.title}</a>
    </span>
    <span style={{ width: '30%' }}>{item.author}</span>
    <span style={{ width: '10%' }}>{item.num_comments}</span>
    <span style={{ width: '10%' }}>{item.points}</span>
    <span style={{ width: '10%' }}>
      <button
```

```
      type="button"
      onClick={() => onRemoveItem(item)}
      className="button button_small"
     >
      Dismiss
     </button>
    </span>
  </div>
);
```

如你所见，我们也可以使用 HTML 元素的原生 `style` 属性。在 JSX 中，style 可以作为一个内联的 JavaScript 对象传递这些属性。这样我们就可以在 JavaScript 文件中定义动态样式属性，而不是大部分的静态 CSS 文件。这种方法被称为内联样式，它对于快速原型设计和动态样式定义非常有用。然而，内联样式应该慎用，因为单独的样式定义可以让 JSX 更简洁。

在你的 *src/App.css* 文件中，定义新的 CSS 类。使用了基本的 CSS 特性。此示例中不包括来自 CSS 扩展（如 Sass）的高级 CSS 功能（如嵌套），因为它们是可选配置[165]。

src/App.css

```
.item {
  display: flex;
  align-items: center;
  padding-bottom: 5px;
}

.item > span {
  padding: 0 5px;
  white-space: nowrap;
  overflow: hidden;
  white-space: nowrap;
  text-overflow: ellipsis;
}

.item > span > a {
  color: inherit;
}
```

前一个组件中的按钮样式仍然缺失，所以我们将定义一个基本的按钮样式和两个更特定的按钮样式（小和大）。其中一个按钮样式已经被使用了，另一个将在下一步中使用。

[165] https://create-react-app.dev/docs/adding-a-sass-stylesheet/

src/App.css

```
.button {
  background: transparent;
  border: 1px solid #171212;
  padding: 5px;
  cursor: pointer;

  transition: all 0.1s ease-in;
}

.button:hover {
  background: #171212;
  color: #ffffff;
}

.button_small {
  padding: 5px;
}

.button_large {
  padding: 10px;
}
```

除了 React 中的样式方法外，命名约定 ([CSS 指南[166]) 是另一个主题。最后的 CSS 片段遵循了 BEM 规则，用下划线 (_) 定义了一个类针对修改。选择任何适合你和你的团队的命名惯例。事不宜迟，我们将对下一个 React 组件设置样式。

src/App.js

```
const SearchForm = ({ ... }) => (
  <form onSubmit={onSearchSubmit} className="search-form">
    <InputWithLabel ... >
      <strong>Search:</strong>
    </InputWithLabel>

    <button
      type="submit"
      disabled={!searchTerm}
      className="button button_large"
    >
      Submit
    </button>
```

[166] https://developer.mozilla.org/en-US/docs/MDN/Contribute/Guidelines/Code_guidelines/CSS

```
  </form>
);
```

我们还可以将 `className` 属性作为 prop 传递给 React 组件。例如，我们可以使用此选项为 SearchForm 组件传递灵活的样式, 该样式来自 CSS 文件的一系列预定义类的 `className` 属性。最后，设置 InputWithLabel 组件的样式:

src/App.js

```
const InputWithLabel = ({ ... }) => {
  ...

  return (
    <>
      <label htmlFor={id} className="label">
        {children}
      </label>
       
      <input
        ref={inputRef}
        id={id}
        type={type}
        value={value}
        onChange={onInputChange}
        className="input"
      />
    </>
  );
};
```

在 *src/App.css* 文件中，添加其余的类:

src/App.css

```
.search-form {
  padding: 10px 0 20px 0;
  display: flex;
  align-items: baseline;
}

.label {
  border-top: 1px solid #171212;
  border-left: 1px solid #171212;
  padding-left: 5px;
```

```
  font-size: 24px;
}

.input {
  border: none;
  border-bottom: 1px solid #171212;
  background-color: transparent;

  font-size: 24px;
}
```

为了简单起见，我们在 *src/App.css* 文件中分别定义了标签和输入等元素的样式。然而，在实际应用中，最好在 *src/index.css* 文件中全局定义一次这些元素。由于 React 组件被拆分成多个文件，共享样式就成了一种必然。

这是我们大多数人已经学习的基本 CSS，以更加动态的内联样式编写。但是，如果没有像 Sass 这样的 CSS 扩展（语法上很棒的样式表），则内联样式会变得很麻烦，因为 CSS 嵌套之类的功能在原生 CSS 中不可用。

练习

- 检查上一节的源码[167]
- 确认上一节之后的变更[168]
- 阅读更多关于creative-react-app 中的 CSS 样式表[169]
- 阅读更多关于creative-react-app 中的 Sass[170]，以利用更高级的 CSS 功能，如嵌套。
- 尝试将“className”属性从 App 传递到 SearchForm 组件，其值为“button_small”或“button_large”，并将其用作按钮元素的“className”。

[167] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/CSS-in-React
[168] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/react-modern-final...hs/CSS-in-React?expand=1
[169] https://create-react-app.dev/docs/adding-a-stylesheet
[170] https://create-react-app.dev/docs/adding-a-sass-stylesheet

React 中的 CSS 模块化

CSS Modules 是一种高级的 **CSS-in-CSS** 方法。CSS 文件保持不变，你可以应用 Sass 等 CSS 扩展，但它在 React 组件中的使用会发生变化。要在 `create-react-app` 中启用 CSS 模块，请将 *src/App.css* 文件重命名为 *src/App.module.css*。这个操作是在你的项目目录的命令行中进行的。

Command Line

```
mv src/App.css src/App.module.css
```

在重命名后的 *src/App.module.css* 中，像之前一样从第一个 CSS 类开始定义：

src/App.module.css

```
.container {
  height: 100vw;
  padding: 20px;

  background: #83a4d4; /* fallback for old browsers */
  background: linear-gradient(to left, #b6fbff, #83a4d4);

  color: #171212;
}

.headlinePrimary {
  font-size: 48px;
  font-weight: 300;
  letter-spacing: 2px;
}
```

再用相对路径导入 *src/App.module.css* 文件。这一次，把它作为一个 JavaScript 对象导入，对象的名称（这里是 `styles`）由你决定：

src/App.js

```
import React from 'react';
import axios from 'axios';

import styles from './App.module.css';
```

不需要将 `className` 定义为一个映射到 CSS 文件的字符串，而是直接从 `styles` 对象中访问 CSS 类，并通过 `JavaScript in JSX` 表达式将其分配给你的元素：

src/App.js

```
const App = () => {
  ...

  return (
    <div className={styles.container}>
      <h1 className={styles.headlinePrimary}>My Hacker Stories</h1>

      <SearchForm
        searchTerm={searchTerm}
        onSearchInput={handleSearchInput}
        onSearchSubmit={handleSearchSubmit}
      />

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </div>
  );
};
```

有各种方法可以通过 `styles` 对象向元素的单个 `className` 属性添加多个 CSS 类。在这里，我们使用的是 `JavaScript` 模板字符串：

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <div className={styles.item}>
    <span style={{ width: '40%' }}>
      <a href={item.url}>{item.title}</a>
    </span>
    <span style={{ width: '30%' }}>{item.author}</span>
    <span style={{ width: '10%' }}>{item.num_comments}</span>
    <span style={{ width: '10%' }}>{item.points}</span>
    <span style={{ width: '10%' }}>
      <button
        type="button"
        onClick={() => onRemoveItem(item)}
        className={`${styles.button} ${styles.buttonSmall}`}
```

```
    >
      Dismiss
    </button>
  </span>
 </div>
);
```

我们还可以在 JSX 中再次添加内联样式作为更多的动态样式。也可以添加像 Sass 这样的 CSS 扩展来实现 CSS 嵌套等高级功能。不过我们会坚持使用原生 CSS 的功能：

src/App.module.css

```
.item {
  display: flex;
  align-items: center;
  padding-bottom: 5px;
}

.item > span {
  padding: 0 5px;
  white-space: nowrap;
  overflow: hidden;
  white-space: nowrap;
  text-overflow: ellipsis;
}

.item > span > a {
  color: inherit;
}
```

下面是 *src/App.module.css* 文件中按钮 CSS 类：

src/App.module.css

```
.button {
  background: transparent;
  border: 1px solid #171212;
  padding: 5px;
  cursor: pointer;

  transition: all 0.1s ease-in;
}

.button:hover {
```

```
  background: #171212;
  color: #ffffff;
}

.buttonSmall {
  padding: 5px;
}

.buttonLarge {
  padding: 10px;
}
```

这里有一个向伪 BEM 命名惯例的转变，与上一节的 `button_small` 和 `button_large` 形成对比。如果前面的命名约定成立，我们只能用 `styles['button_small']` 来访问样式，由于 JavaScript 对对象下划线的限制，这使得它更加啰嗦。同样的缺点也适用于用破折号（-）定义的类。相比之下，现在我们可以使用 `styles.buttonSmall` 来代替（参见：Item 组件）：

src/App.js

```
const SearchForm = ({ ... }) => (
  <form onSubmit={onSearchSubmit} className={styles.searchForm}>
    <InputWithLabel ... >
      <strong>Search:</strong>
    </InputWithLabel>

    <button
      type="submit"
      disabled={!searchTerm}
      className={`${styles.button} ${styles.buttonLarge}`}
    >
      Submit
    </button>
  </form>
);
```

SearchForm 组件也会接收这些样式。它必须使用字符串插值，通过 JavaScript 的模板字符串在一个元素中使用两个样式。另一种方法是 classnames[171] 库，它作为项目依赖通过命令行安装：

[171] https://github.com/JedWatson/classnames

src/App.js

```
import cs from 'classnames';

...

// somewhere in a className attribute
className={cs(styles.button, styles.buttonLarge)}
```

该库也提供了条件样式。最后，继续介绍 InputWithLabel 组件：

src/App.js

```
const InputWithLabel = ({ ... }) => {
  ...

  return (
    <>
      <label htmlFor={id} className={styles.label}>
        {children}
      </label>
       
      <input
        ref={inputRef}
        id={id}
        type={type}
        value={value}
        onChange={onInputChange}
        className={styles.input}
      />
    </>
  );
};
```

并在 *src/App.module.css* 文件中完成剩下的样式：

src/App.module.css

```
.searchForm {
  padding: 10px 0 20px 0;
  display: flex;
  align-items: baseline;
}

.label {
  border-top: 1px solid #171212;
  border-left: 1px solid #171212;
  padding-left: 5px;
  font-size: 24px;
}

.input {
  border: none;
  border-bottom: 1px solid #171212;
  background-color: transparent;

  font-size: 24px;
}
```

与上一节同样的注意事项：其中一些样式，如 `input` 和 `label`，在没有 CSS 模块的全局 *src/index.css* 文件中可能更有效。

同样，CSS Modules 就像任何其他 CSS-in-CSS 方法一样，可以使用 Sass 来实现更高级的 CSS 功能，比如嵌套。CSS 模块的优势在于，每次当一个样式没有被定义时，我们都会在 JavaScript 中收到一个错误。在标准的 CSS 方法中，JavaScript 和 CSS 文件中未匹配的样式可能会被忽略。

练习

- 检查 上一节的源码[172]
- 确认 上一节之后的变更[173]
- 了解更多 create-react-app 中的 CSS Modules[174]。

[172] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/CSS-Modules-in-React
[173] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/react-modern-final...hs/CSS-Modules-in-React?expand=1
[174] https://create-react-app.dev/docs/adding-a-css-modules-stylesheet

React 中的样式组件

有了前面 CSS-in-CSS 的方法，**Styled Components** 是 **CSS-in-JS** 的几种方法之一。我选择 **Styled Components** 是因为它是最流行的。它是作为 JavaScript 的依赖，所以我们必须在命令行上安装它：

Command Line

```
npm install styled-components
```

然后在你的 *src/App.js* 文件中引入它：

src/App.js

```
import React from 'react';
import axios from 'axios';
import styled from 'styled-components';
```

顾名思义，CSS-in-JS 发生在你的 JavaScript 文件中。在你的 *src/App.js* 文件中，定义你的第一个样式组件：

src/App.js

```
const StyledContainer = styled.div`
  height: 100vw;
  padding: 20px;

  background: #83a4d4;
  background: linear-gradient(to left, #b6fbff, #83a4d4);

  color: #171212;
`;

const StyledHeadlinePrimary = styled.h1`
  font-size: 48px;
  font-weight: 300;
  letter-spacing: 2px;
`;
```

当使用样式化组件时，你使用 JavaScript 模板字符串的方式与 JavaScript 函数的使用方式相同。模板字符串之间的所有内容都可以看作是一个参数，而 `styled` 对象则让你可以作为函数访问所有必要的 HTML 元素（如：div、h1）。一旦函数被调用了样式，它就会返回一个 React 组件，可以在你的 App 组件中使用：

src/App.js

```
const App = () => {
  ...

  return (
    <StyledContainer>
      <StyledHeadlinePrimary>My Hacker Stories</StyledHeadlinePrimary>

      <SearchForm
        searchTerm={searchTerm}
        onSearchInput={handleSearchInput}
        onSearchSubmit={handleSearchSubmit}
      />

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </StyledContainer>
  );
};
```

这种 React 组件与普通的 React 组件遵循相同的规则。它的元素标签之间传递的所有内容都会自动作为 React `children` prop 传递下去。对于 Item 组件，我们这次没有使用内联样式，而是为它定义了一个专门的样式组件。`StyledColumn` 使用 React prop 动态接收其样式：

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <StyledItem>
    <StyledColumn width="40%">
      <a href={item.url}>{item.title}</a>
    </StyledColumn>
    <StyledColumn width="30%">{item.author}</StyledColumn>
    <StyledColumn width="10%">{item.num_comments}</StyledColumn>
    <StyledColumn width="10%">{item.points}</StyledColumn>
    <StyledColumn width="10%">
      <StyledButtonSmall
        type="button"
        onClick={() => onRemoveItem(item)}
```

```
    >
      Dismiss
    </StyledButtonSmall>
   </StyledColumn>
  </StyledItem>
);
```

灵活的 `width` prop 可作为内联函数的参数，在样式组件的模板字符串中访问。函数的返回值作为字符串使用，并且由于我们可以省略箭头函数的主体并立即返回结果，所以它成为一个简洁的内联函数：

src/App.js

```
const StyledItem = styled.div`
  display: flex;
  align-items: center;
  padding-bottom: 5px;
`;

const StyledColumn = styled.span`
  padding: 0 5px;
  white-space: nowrap;
  overflow: hidden;
  white-space: nowrap;
  text-overflow: ellipsis;

  a {
    color: inherit;
  }

  width: ${props => props.width};
`;
```

样式组件中默认有 CSS 嵌套等高级功能，嵌套的元素是可以访问的，并且可以用 `&`CSS 操作符选择当前元素：

src/App.js

```
const StyledButton = styled.button`
  background: transparent;
  border: 1px solid #171212;
  padding: 5px;
  cursor: pointer;

  transition: all 0.1s ease-in;

  &:hover {
    background: #171212;
    color: #ffffff;
  }
`;
```

你也可以通过向库中的函数传递另一个组件来创建一个特殊的样式组件。定义的按钮组件将会从之前定义的 StyledButton 组件中接收所有的基本样式：

src/App.js

```
const StyledButtonSmall = styled(StyledButton)`
  padding: 5px;
`;

const StyledButtonLarge = styled(StyledButton)`
  padding: 10px;
`;

const StyledSearchForm = styled.form`
  padding: 10px 0 20px 0;
  display: flex;
  align-items: baseline;
`;
```

当我们使用像 StyledSearchForm 这样的样式组件时，它的底层元素 (`form`, `button`) 被用于真正的 HTML 输出。我们可以在那里继续使用原生的 HTML 属性（`onSubmit`、`type`、`disabled`）：

src/App.js

```
const SearchForm = ({ ... }) => (
  <StyledSearchForm onSubmit={onSearchSubmit}>
    <InputWithLabel
      id="search"
      value={searchTerm}
      isFocused
      onInputChange={onSearchInput}
    >
      <strong>Search:</strong>
    </InputWithLabel>

    <StyledButtonLarge type="submit" disabled={!searchTerm}>
      Submit
    </StyledButtonLarge>
  </StyledSearchForm>
);
```

最后，InputWithLabel 组件用其尚未定义的样式组件装饰一下：

src/App.js

```
const InputWithLabel = ({ ... }) => {
  ...

  return (
    <>
      <StyledLabel htmlFor={id}>{children}</StyledLabel>
       
      <StyledInput
        ref={inputRef}
        id={id}
        type={type}
        value={value}
        onChange={onInputChange}
      />
    </>
  );
};
```

而与之相匹配的样式化组件也被定义在同一个文件中：

src/App.js

```
const StyledLabel = styled.label`
  border-top: 1px solid #171212;
  border-left: 1px solid #171212;
  padding-left: 5px;
  font-size: 24px;
`;

const StyledInput = styled.input`
  border: none;
  border-bottom: 1px solid #171212;
  background-color: transparent;

  font-size: 24px;
`;
```

带有样式化组件的 CSS-in-JS 将定义样式的重点转移到实际的 React 组件上。Styled Components 是作为 React 组件定义的样式，没有中间的 CSS 文件。如果一个样式组件没有在 JavaScript 中使用，你的 IDE 或编辑器会告诉你。样式化组件被打包在 JavaScript 文件中的其他 JavaScript 资源中，用于准备部署在生产环境的应用程序中。在使用 CSS-in-JS 策略时，没有额外的 CSS 文件，只有 JavaScript。CSS-in-JS 和 CSS-in-CSS 这两种策略及其方法（如：Styled Components 和 CSS Modules）在 React 开发人员中很受欢迎。选择使用最适合你和你的团队的方法。

练习

- 检查 上一节的源码[175]
- 确认 上一节之后的变更[176]
- 了解更多 React 中的样式组件[177]：
  - 当使用 Styled Components 时，通常没有 *src/index.css* 文件用于全局样式。了解如何在使用 Styled Components 时使用全局样式。

[175] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Styled-Components-in-React
[176] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/react-modern-final...hs/Styled-Components-in-React?expand=1
[177] https://www.robinwieruch.de/react-styled-components

React 中的 SVG

要创建一个现代 React 应用程序，我们可能需要使用 SVG。例如我们可能不想给每个按钮元素都加上文字，而是希望用一个图标来显示，从而变得轻量级。在本节中，我们将在一个 React 组件中使用一个可扩展的矢量图形（SVG）作为图标。

这一部分是基于我们前面介绍的 “React 中的 CSS “的基础上，让 SVG 图标有一个好的外观和感觉。使用不同的样式方法或者完全不使用样式都是可以接受的，尽管没有加上样式的 SVG 可能会显得不伦不类。

这个 SVG 的图标来自于 Flaticon’s Freepick[178]。这个网站上的许多 SVG 都是免费的，但需要注明作者。你可以从这里[179]下载 SVG 图标，并将其作为 *src/check.svg* 放到你的项目中。下载 SVG 文件是推荐的方式，这里是 SVG 的定义：

Code Playground

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- Generator: Adobe Illustrator 18.0.0, SVG Export Plug-In . SVG Version: 6.00 Bui\
ld 0)  -->
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/D\
TD/svg11.dtd">
<svg version="1.1" id="Capa_1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http:\
//www.w3.org/1999/xlink" x="0px" y="0px"
  viewBox="0 0 297 297" style="enable-background:new 0 0 297 297;" xml:space="prese\
rve">
 <g>
  <path d="M113.636,272.638c-2.689,0-5.267-1.067-7.168-2.97L2.967,166.123c-3.956-3\
.957-3.956-10.371-0.001-14.329l54.673-54.703
    c1.9-1.9,4.479-2.97,7.167-2.97c2.689,0,5.268,1.068,7.169,2.969l41.661,41.676L2\
25.023,27.332c1.9-1.901,4.48-2.97,7.168-2.97l0,0
    c2.688,0,5.268,1.068,7.167,2.97l54.675,54.701c3.956,3.957,3.956,10.372,0,14.32\
8L120.803,269.668
    C118.903,271.57,116.325,272.638,113.636,272.638z M24.463,158.958l89.173,89.209\
l158.9-158.97l-40.346-40.364L120.803,160.264
    c-1.9,1.902-4.478,2.971-7.167,2.971c-2.688,0-5.267-1.068-7.168-2.97l-41.66-41.\
674L24.463,158.958z"/>
 </g>
</svg>
```

因为我们再次使用了 create-react-app，所以我们可以马上导入 SVG（类似于 CSS）作为 React 组件。在 *src/App.js* 中，使用下面的语法导入 SVG：

[178] https://www.flaticon.com/authors/freepik
[179] https://www.flaticon.com/free-icon/check_109748

src/App.js

```
import React from 'react';
import axios from 'axios';

import './App.css';
import { ReactComponent as Check } from './check.svg';
```

我们正在导入一个 SVG，SVG 可以有许多不同用途（如 logo、背景）。用一个 SVG 组件作为 `height` 和 `width` 属性来代替一个按钮文本。

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <div className="item">
    <span style={{ width: '40%' }}>
      <a href={item.url}>{item.title}</a>
    </span>
    <span style={{ width: '30%' }}>{item.author}</span>
    <span style={{ width: '10%' }}>{item.num_comments}</span>
    <span style={{ width: '10%' }}>{item.points}</span>
    <span style={{ width: '10%' }}>
      <button
        type="button"
        onClick={() => onRemoveItem(item)}
        className="button button_small"
      >
        <Check height="18px" width="18px" />
      </button>
    </span>
  </div>
);
```

无论你使用的是哪种样式的方法，你都可以给你的 SVG 按钮中的图标加上悬停效果。在基本的 CSS 方法中，*src/App.css* 文件中看起来像下面这样：

src/App.css

```
.button:hover > svg > g {
  fill: #ffffff;
  stroke: #ffffff;
}
```

creat-react-app 项目让使用 SVG 变得简单直接，不需要额外的配置。如果你用 Webpack 等构建工具从头开始创建一个 React 项目，那就不一样了，因为你必须自己去处理。总之，SVG 可以让你的应用程序更加平易近人，所以在你觉得适合的时候就可以使用它们。

练习

- 检查上一节的源码[180]
- 确认上一节之后的变更[181]
- 阅读更多关于create-react-app 中的 SVGs[182]。
- 阅读更多关于React 中的 SVG 背景图案[183]。
- 在你的应用程序中添加另一个 SVG 图标。

---

[180] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/CSS-in-React-SVG

[181] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/CSS-in-React...hs/CSS-in-React-SVG?expand=1

[182] https://create-react-app.dev/docs/adding-images-fonts-and-files

[183] https://www.robinwieruch.de/react-svg-patterns

React 维护

一旦 React 应用开始变得越来越庞大，维护将会成为重中之重。我们将介绍性能优化，类型安全，测试以及项目结构等方面来应对这种情况。每一个方面都能在不降低质量的情况下增强您的应用，以能承担更多的功能。

性能优化通过确保有效利用可用资源来防止应用程序变慢；诸如 TypeScript 之类的类型化编程语言会在反馈循环中更早的发现错误；相比于类型化编程，测试给了我们更多明确的反馈，并且提供了一种了解哪些操作可能对应用造成危害的方式；最后，项目结构支持将素材组织到文件和文件夹中，这在团队成员异地合作的情况下十分有用。

React 性能（高级）

本节只是为了学习 React 的性能改进而已。因为 React 开箱即用的特点，我们在大多数 React 应用中并不需要优化。尽管有很多检测 JavaScript 和 React 性能的更复杂的工具，但是我们会坚持使用简单的 `console.log()` 和我们的浏览器开发者工具来记录日志。

不要在第一次渲染时运行

之前，我们介绍了 React 中用于产生副作用的 useEffect Hook。它在组件第一次渲染或挂载时运行，然后每次重新渲染或更新也会运行。通过将空的依赖数组作为第二个参数传递个它，我们就能告诉 Hook 仅在第一次渲染时运行。比如，使用 React 的 useState Hook 检查自定义 Hook 的状态管理，并使用 React 的 useEffect Hook 将其半永久状态与 local storage 一起使用：

src/App.js

```
const useSemiPersistentState = (key, initialState) => {
  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );

  React.useEffect(() => {
    console.log('A');
    localStorage.setItem(key, value);
  }, [value, key]);

  return [value, setValue];
};
```

仔细研究开发者工具，我们使用此自定义 Hook 能发现第一次组件渲染时的日志。为组件的初始渲染运行副作用没有任何意义，因为除了初始值之外，没有其他东西可以存储在 local storage 中。所以这是一个多余的函数调用，应该仅在每次组件更新或重新渲染时运行。

正如前面所提到的，没有在每次重新渲染时运行的 Hook，也没有办法以 React 管用的方式告诉 `useEffect` Hook 仅在每次重新渲染时调用其函数。然而，通过使用 React 的 useRef Hook 使其在重新渲染时保持其 `ref.current` 属性不变，我们可以保持组件生命周期的 组成状态（无需在状态更新时重新渲染组件）：

src/App.js

```
const useSemiPersistentState = (key, initialState) => {
  const isMounted = React.useRef(false);

  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );

  React.useEffect(() => {
    if (!isMounted.current) {
      isMounted.current = true;
    } else {
      console.log('A');
      localStorage.setItem(key, value);
    }
  }, [value, key]);

  return [value, setValue];
};
```

我们正在将 `ref` 及其可变的 `current` 属性用于状态管理，该操作并不会触发重新渲染。一旦 Hook 在组件第一次渲染时被调用，那么就会用一个名为 `isMounted` 的 `false` 布尔变量初始化 ref 的 `current` 值。结果是，并没有调用 `useEffet` 中的副作用函数，而副作用函数只有当布尔变量 `isMounted` 变为 `true` 后才会被调用。每当 Hook 再次运行（或重新渲染组件）时，都会在副作用中判断布尔值变量，只有当变量为 `true` 时，才会执行副作用函数。在组件的整个生命周期中，`isMounted` 布尔值将一直为 `true`，这样做是为了避免第一次使用我们的自定义 Hook 时调用副作用函数。

以上只是关于防止组件第一次渲染时调用一个简单的函数的问题，但是想象一下在副作用函数中有一个开销很大的计算，或者自定义 Hook 在应用中被频繁的使用，那么使用这种方法就可以更实用地避免不必要的函数调用。

提示：这种方式不仅仅适用于性能优化，还可以为了仅在组件重新渲染时产生副作用而使用。我使用过几次这种方式，并且我认为您也能发现更多这种方式适用的场景。

不要进行不必要重新渲染

在前面，我们探讨了 React 的重新渲染机制，接下来我们将为了应用和 List 组件重复此练习。为这两个组件添加一条日志记录。

src/App.js

```
const App = () => {
  ...

  console.log('B:App');

  return ( ... );
};

const List = ({ list, onRemoveItem }) =>
  console.log('B:List') ||
  list.map(item => (
    <Item
      key={item.objectID}
      item={item}
      onRemoveItem={onRemoveItem}
    />
  ));
```

因为 List 组件没有函数体，并且开发人员很懒，不想为简单的日志记录重构该组件，所以 List 组件使用了 `||` 运算符。这是一个很 tricky 的方式向没有函数体的功能组件添加日志记录。因为在操作符左侧的 `console.log()` 始终为 false，所以操作符右侧的总会运行。

Code Playground

```
function getTheTruth() {
  if (console.log('B:List')) {
    return true;
  } else {
    return false;
  }
}

console.log(getTheTruth());
// B:List
// false
```

让我们关注在浏览器开发者工具中真正的日志，你应该看到相似的输出，首先 App 组件渲染，然后就是它的子组件（比如：List 组件）。

Visualization

```
B:App
B:List
B:App
B:App
B:List
```

由于副作用在第一次渲染时触发了获取数据的操作，因此只有 App 组件会渲染，因为 List 组件被加载状态替换了。当数据获取完成之后，两个组件都会再次渲染。

Visualization

```
// initial render
B:App
B:List

// data fetching with loading
B:App

// re-rendering with data
B:App
B:List
```

到目前为止，这种行为是可以接收的，因为所有内容都按时渲染了。现在我们将通过在 SearchForm 组件中输入内容，使该实验更进一步。当您每在元素中输入一个字符后，您应该会看到相应的改变：

Visualization

```
B:App
B:List
```

但是 List 组件不应该重新渲染，搜索功能不是通过按钮执行的，因此传递给 List 组件的 `list` 应该保持不变。这就是让很多人感到惊讶的 React 的默认行为。

如果一个父组件重新渲染，他的子组件也会随之重新渲染。React 默认就会这么做，因为阻止子组件的重新渲染可能会导致很多问题，并且 React 的重新渲染机制执行的很快。

可是有时候我们想防止组件重新渲染。比如，如果表格中展示的巨大的数据集合并不受更新的影响，那么就不应该重新渲染。如果组件有所更改，则执行相等性的检查会更有效。因此我们可以使用 React 的 memo API 对 props 进行相等性检查：

src/App.js

```
const List = React.memo(
  ({ list, onRemoveItem }) =>
    console.log('B:List') ||
    list.map(item => (
      <Item
        key={item.objectID}
        item={item}
        onRemoveItem={onRemoveItem}
      />
    ))
);
```

然而，当向 SearchForm 中输入值的时候输出仍然没有改变：

Visualization

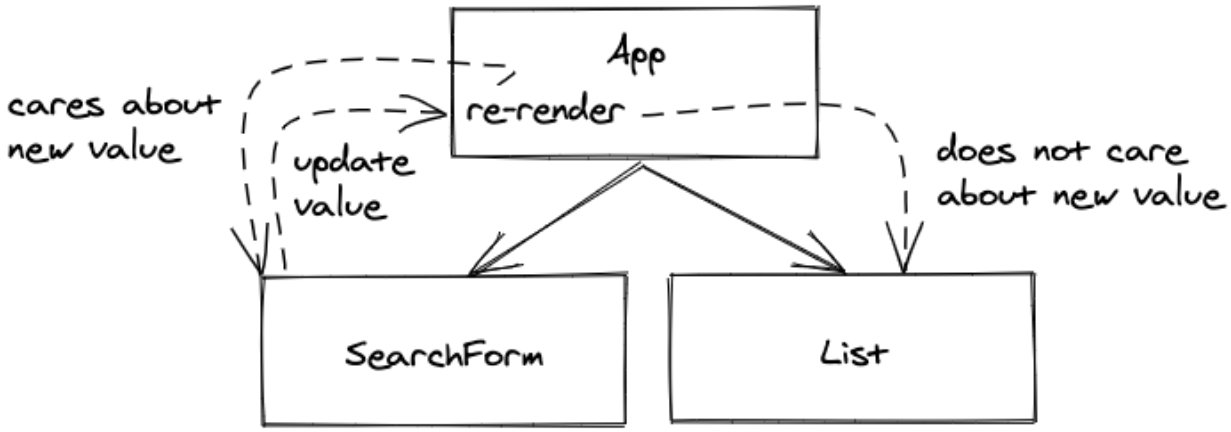
```
B:App
B:List
```

虽然传给 List 组件的 `list` 是相同的，但是 `onRemoveItem` 回调方法却不是，如果 App 组件重新渲染了，它依然会创建一个新的回调。此前，我们使用 React 的 useCallback Hook 阻止这一行为，仅在组件重新渲染并且依赖项中有改变时才创建一个函数。

src/App.js

```
const App = () => {
  ...

  const handleRemoveStory = React.useCallback(item => {
    dispatchStories({
      type: 'REMOVE_STORY',
      payload: item,
    });
  }, []);

  ...

  console.log('B:App');

  return (... );
};
```

因为回调函数在函数签名中获取了作为参数传递的 `item`，所以它没有任何依赖关系，并且只在 App 组件初始渲染的时候被声明了一次。传递给 List 组件的 props 都不应该变化，尝试着将 `memo` 和 `useCallback` 一起使用来搜索输入在 SearchForm 中的内容。



虽然传递给组件的所有 props 都保持不变，但是如果它们的父组件强制重新渲染，那么它们也会再次渲染。这是 React 的默认行为，因为重新渲染的机制足够快，所以它在大多数情况下都是这么运作。但是，如果重新渲染降低了 React 应用的性能，那么 `memo` 将有助于防止重新渲染。

有时单独使用 `memo` 并没有什么用，每次父组件中都会重新定义回调，并且将其作为变化了的 props 传给该组件，这会触发另一次重新渲染。在这种情况下，`useCallback` 可以使回调只有在依赖项发生变化时才进行改变。

不要反复运行大开销的计算

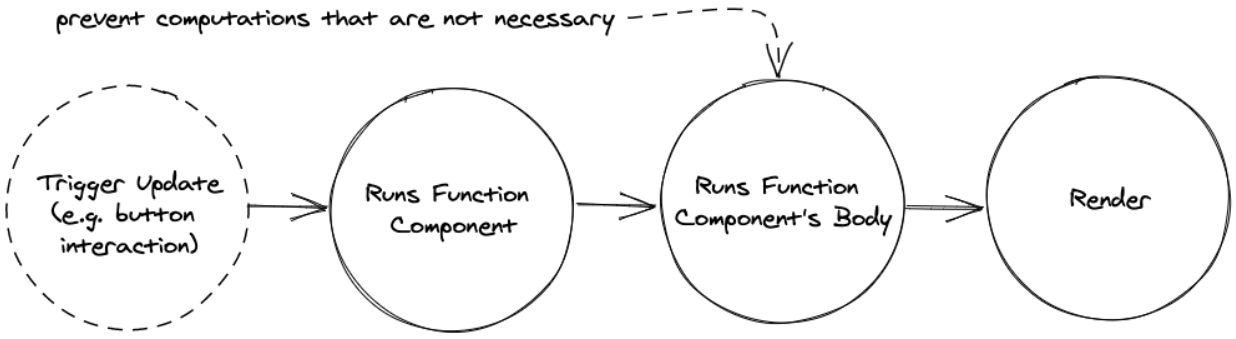
有时候，每次渲染都会在 React 组件的函数签名和 return 块之间进行频繁的计算。对于这种情况，我们必须首先在我们当前的应用程序中创建一个用例。

src/App.js

```
const getSumComments = stories => {
  console.log('C');

  return stories.data.reduce(
    (result, value) => result + value.num_comments,
    0
  );
};

const App = () => {
  ...
```

```
  const sumComments = getSumComments(stories);

  return (
    <div>
      <h1>My Hacker Stories with {sumComments} comments.</h1>

      ...
    </div>
  );
};
```

如果所有的参数都传递给函数，那么就可以允许在组件外部使用它。它防止了每次渲染都创建一个函数，因此 `useCallback` Hook 变得没那么必要了。但该函数仍然在每次渲染时计算 comments 的总和，这对于更大开销的计算来说就成为了一个问题。



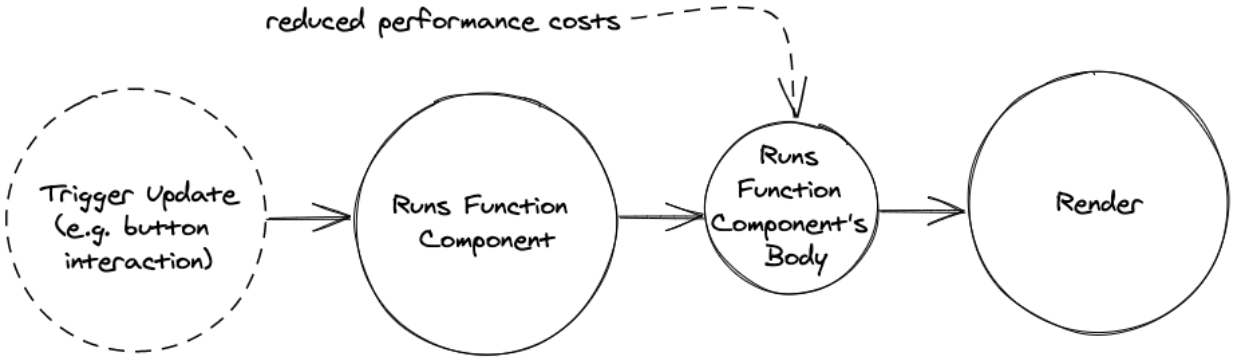
每一次在 SearchForm 组件输入内容时，此计算都会再次输出“C”。对于像这种没那么大的开销的计算来说还可以接受，但是可以想象一下如果计算时间要花费500毫秒以上，这会给重新渲染带来延迟，因此组件中所有的组件都必须等待此计算。我们可以告诉 React 仅在其依赖项之一改变的情况下运行函数，如果未更改任何依赖关系，那么函数结果将保持不变。React 的 useMemo Hook 在这里为我们提供了帮助：

src/App.js

```
const App = () => {
  ...

  const sumComments = React.useMemo(() => getSumComments(stories), [
    stories,
  ]);

  return ( ... );
};
```

每次在 SearchForm 中输入内容的时候，不会再次计算，只有当依赖项的数组（在这儿指

stories）改变时才会再次运行。毕竟，这只应用于那些因计算量开销很大而导致渲染（或重新渲染）时造成延迟的组件。



现在，我们已经了解了使用 useMemo、useCallback 和 memo 的场景，请记住一般情况下不一定必须使用这些工具。只有在遇到性能瓶颈时才应用这些优化方式，大多数情况下并不需要优化，因为 React 的渲染机制非常高效。有时，像 memo 之类优化方式的开销比组件重新渲染的开销还要大。

练习

- 检查 上一节的源码[184].
- 确认 上一节之后的变更[185].
- 阅读更多关于 React 的 memo API[186].
- 阅读更多关于 React 的 useCallback Hook[187].
- 下载 *React* 开发者工具作为浏览器拓展程序，并通过浏览器的开发者工具在浏览器的应用程序打开它，然后尝试其各种功能。比如，您可以用它可视化 React 组件树并且更新组件。
- 使用“删除”按钮从 List 中移除一项时 SearchForm 会重新渲染吗？如果会的话，请使用上述性能优化的方法防止其重新渲染。
- 当从 List 中移除一项时，List 中的其余项会重新渲染吗？如果会的话，请使用上述性能优化的方法防止其重新渲染。
- 移除所有的性能优化操作保持应用的简单性，我们现在的应用程序并没有遇到性能瓶颈。尽量避免过早优化[188]，如果遇到性能问题，请参考本节内容。

[184] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Performance-in-React
[185] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/react-modern-final...hs/Performance-in-React?expand=1
[186] https://reactjs.org/docs/react-api.html#reactmemo
[187] https://reactjs.org/docs/hooks-reference.html#usecallback
[188] https://en.wikipedia.org/wiki/Program_optimization

在 React 中使用 TypeScript

在 JavaScript 和 React 中使用 TypeScript，对于开发强大的应用程序有许多好处。TypeScript 集成后不会在运行时在命令行或浏览器中出现类型错误，而是会在 IDE 内编译时显示它们。它缩短了 JavaScript 开发的反馈循环的周期。TypeScript 不仅改善了开发人员的体验，也让代码变得更加自文档化，并且增加可读性，因为每个变量都使用了类型定义。同样的，移动代码块或执行更大的代码库的重构也变得更加高效。诸如 TypeScript 之类的静态类型语言是发展趋势，因为它们比诸如 JavaScript 之类的动态类型语言具有更多优势。所以，尽可能多地了解 Typescript[189] 是非常有用的。

要在 React 中使用 TypeScript，请使用命令行将 TypeScript 及其依赖项安装到你的应用程序中。如果遇到障碍，请遵循官方 TypeScript 安装说明中的 create-react-app[190]：

Command Line

```
npm install --save typescript @types/node @types/react
npm install --save typescript @types/react-dom @types/jest
```

接下来，将所有 JavaScript 文件 (*.js*) 重命名为 TypeScript 文件 (*.tsx*)。

Command Line

```
mv src/index.js src/index.tsx
mv src/App.js src/App.tsx
```

然后在命令行中重新启动开发服务器。你可能会在浏览器和 IDE 中遇到编译错误的问题。如果是 IDE 没有正确显示编译错误，请尝试为你的编辑器安装 TypeScript 插件，或者为 IDE 安装扩展程序。在 React 设置中初始化 TypeScript 之后，我们将为整个 *src/App.tsx* 文件添加安全类型，从编写自定义 hook 的参数开始：

src/App.tsx

```
const useSemiPersistentState = (
  key: string,
  initialState: string
) => {
  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );

  React.useEffect(() => {
    localStorage.setItem(key, value);
  }, [value, key]);
```

[189] https://www.typescriptlang.org/index.html
[190] https://create-react-app.dev/docs/adding-typescript/

```
  return [value, setValue];
};
```

为函数的参数添加类型定义的做法主要关注在 Javascript，而不是 React。我们定义该函数需要两个参数，并且两个参数都是 JavaScript 中的基本类型 string。同样的，我们可以定义该函数的返回值类型为字符串数组，并告诉一些类似 `state updater function` 的方法，该函数不返回任何值 (`void`)：

src/App.tsx

```
const useSemiPersistentState = (
  key: string,
  initialState: string
): [string, (newValue: string) => void] => {
  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );

  React.useEffect(() => {
    localStorage.setItem(key, value);
  }, [value, key]);

  return [value, setValue];
};
```

尽管与 React 有关，但考虑到以前对自定义 hook 的类型安全性的改进，我们没有在函数体内部的 React hooks 中添加类型。这是因为类型推断在大多数情况下对于 React hooks 都适用。如果一个 React useState Hook 的 初始化 *state* 是一个 JavaScript 的字符串类型，那么返回的 当前的 *state* 将被推断为一个字符串类型，而返回的 *state* 更新函数将仅以字符串作为参数，并且没有任何返回值：

Code Playground

```
const [value, setValue] = React.useState('React');
// value is inferred to be a string
// setValue only takes a string as argument
```

我们有多种方法来实现为 React 应用程序及其组件添加类型安全。我们可以从应用程序的一个较小的组件的 props 和 state 开始进行修改。例如，Item 组件接收一个 story（此处为 `item`）以及一个回调处理函数（此处为 `onRemoveItem`）。我们可以像之前一样，为两个函数的参数添加内联的类型：

src/App.tsx

```
const Item = ({
  item,
  onRemoveItem,
}: {
  item: {
    objectID: string;
    url: string;
    title: string;
    author: string;
    num_comments: number;
    points: number;
  };
  onRemoveItem: (item: {
    objectID: string;
    url: string;
    title: string;
    author: string;
    num_comments: number;
    points: number;
  }) => void;
}) => (
  <div>
    ...
  </div>
);
```

这里的代码会有两个问题：代码冗长，并且有重复项。我们可以通过在 *src/App.js* 顶部的组件外部添加自定义的 Story 类型来解决这两个问题：

src/App.tsx

```
type Story = {
  objectID: string;
  url: string;
  title: string;
  author: string;
  num_comments: number;
  points: number;
};

...
```

```
const Item = ({
  item,
  onRemoveItem,
}: {
  item: Story;
  onRemoveItem: (item: Story) => void;
}) => (
  <div>
    ...
  </div>
);
```

`item` 定义为 `Story` 类型；`onRemoveItem` 函数将接受一个 `Story` 类型的参数，并且没有任何返回值。接下来，可以通过在外部定义 Item 组件的 props 的类型的方式来整理一下代码：

src/App.tsx

```
type ItemProps = {
  item: Story;
  onRemoveItem: (item: Story) => void;
};

const Item = ({ item, onRemoveItem }: ItemProps) => (
  <div>
    ...
  </div>
);
```

这是用 TypeScript 定义 React 组件的 props 的最常用的方法。我们可以在组件树中向上寻找到 List 组件，并为 List 组件的 props 添加类型定义：

src/App.tsx

```
type Story = {
  ...
};

type Stories = Array<Story>;

...

type ListProps = {
  list: Stories;
  onRemoveItem: (item: Story) => void;
```

```
};

const List = ({ list, onRemoveItem }: ListProps) =>
  list.map(item => (
    <Item
      key={item.objectID}
      item={item}
      onRemoveItem={onRemoveItem}
    />
  ));
```

我们在 `ItemProps` 和 `ListProps` 中重复定义了 `onRemoveItem` 方法。为了更加简洁，你可以将其提取为一个独立定义的 `OnRemoveItem` 的类型，并且在需要使用 `onRemoveItem` 属性的地方复用它。需要注意的是，随着组件被拆分成不同的文件，开发会变得越来越困难。因此，我们将在此处保留重复项。

当我们我们已经定义了 `Story` 和 `Stories` 类型，便可以将它们重新用于其他组件。将 Story 类型添加到 App 组件中的回调处理器中：

src/App.tsx

```
const App = () => {
  ...

  const handleRemoveStory = (item: Story) => {
    dispatchStories({
      type: 'REMOVE_STORY',
      payload: item,
    });
  };

  ...
};
```

`reducer` 方法也可以管理 `Story` 类型，除非我们不关心 `state` 和 `action` 的类型。作为应用程序的开发人员，我们需要知道传递给该 reducer 函数的参数对象及其类型：

src/App.tsx

```
type StoriesState = {
  data: Stories;
  isLoading: boolean;
  isError: boolean;
};

type StoriesAction = {
  type: string;
  payload: any;
};

const storiesReducer = (
  state: StoriesState,
  action: StoriesAction
) => {
  ...
};
```

带有 `string` 和 `any`（TypeScript 通配符）类型定义的 `Action` 类型看起来不太明确；而且我们通过它无法获得任何类型的安全性，因为 actions 是不可区分的。我们可以通过将每个 action 的类型指定为接口来优化，并使用联合类型（此处为 `StoriesAction`）以确保最终类型安全：

src/App.tsx

```
interface StoriesFetchInitAction {
  type: 'STORIES_FETCH_INIT';
}

interface StoriesFetchSuccessAction {
  type: 'STORIES_FETCH_SUCCESS';
  payload: Stories;
}

interface StoriesFetchFailureAction {
  type: 'STORIES_FETCH_FAILURE';
}

interface StoriesRemoveAction {
  type: 'REMOVE_STORY';
  payload: Story;
}
```

```
type StoriesAction =
  | StoriesFetchInitAction
  | StoriesFetchSuccessAction
  | StoriesFetchFailureAction
  | StoriesRemoveAction;

const storiesReducer = (
  state: StoriesState,
  action: StoriesAction
) => {
  ...
};
```

现在的 stories state，current state 和 action 都定义了类型，并且返回新的 state（通过推断出来的）现在是安全类型。例如，如果你使用未定义的 `action` 类型将 action 分派给 reducer，则会出现类型错误。或者，如果你将 store 以外的其他属性传递给删除 story 的 action，则也会出现类型错误。

在 List 组件中，组件的 return 语句仍然存在类型安全问题。可以通过给 List 组件包装 HTML `div` 元素或 React 片段来解决此问题：

src/App.tsx

```
const List = ({ list, onRemoveItem }: ListProps) => (
  <>
    {list.map(item => (
      <Item
        key={item.objectID}
        item={item}
        onRemoveItem={onRemoveItem}
      />
    ))}
  </>
);
```

根据 TypeScript 和 React 相关的 github issue 的说法：“这是由于编译器的限制，功能组件无法返回除 *JSX* 表达式或 *null* 以外的任何内容，否则它会报告一个隐含的错误消息，内容是指其他类型不可分配给 *Element*。”

接下来关注 SearchForm 组件，该组件具有事件的回调处理器：

src/App.tsx

```
type SearchFormProps = {
  searchTerm: string;
  onSearchInput: (event: React.ChangeEvent<HTMLInputElement>) => void;
  onSearchSubmit: (event: React.FormEvent<HTMLFormElement>) => void;
};

const SearchForm = ({
  searchTerm,
  onSearchInput,
  onSearchSubmit,
}: SearchFormProps) => (
  ...
);
```

我们通常使用 `React.SyntheticEvent` 代替 `React.ChangeEvent` 或 `React.FormEvent` 。回到 App 组件，我们使用该类型来定义回调处理器：

src/App.tsx

```
const App = () => {
  ...

  const handleSearchInput = (
    event: React.ChangeEvent<HTMLInputElement>
  ) => {
    setSearchTerm(event.target.value);
  };

  const handleSearchSubmit = (
    event: React.FormEvent<HTMLFormElement>
  ) => {
    setUrl(`${API_ENDPOINT}${searchTerm}`);

    event.preventDefault();
  };

  ...
};
```

现在只剩下 InputWithLabel 组件了。在处理该组件的 props 之前，让我们看一下 React 的 useRef Hook 中的 `ref`。这里的问题在于还不能推断出返回值：

src/App.tsx

```
const InputWithLabel = ({ ... }) => {
  const inputRef = React.useRef<HTMLInputElement>(null!);

  React.useEffect(() => {
    if (isFocused && inputRef.current) {
      inputRef.current.focus();
    }
  }, [isFocused]);
```

我们将返回的 `ref` 设置为安全类型，并将其定义为只读属性，因为我们仅对它执行 `focus` 方法（只读）。React 将会帮助我们将 DOM 元素设置为 `current` 属性。

最后，我们将对 InputWithLabel 组件的 props 进行安全类型的检查。需要注意的是，`children` 属性具有其 React 的特定类型，并且还有带问号的可选类型：

src/App.tsx

```
type InputWithLabelProps = {
  id: string;
  value: string;
  type?: string;
  onInputChange: (event: React.ChangeEvent<HTMLInputElement>) => void;
  isFocused?: boolean;
  children: React.ReactNode;
};

const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onInputChange,
  isFocused,
  children,
}: InputWithLabelProps) => {
  ...
};
```

`type` 和 `isFocused` 属性都是可选的。使用 `TypeScript` 的时候，你可以告诉编译器不需要将这些作为 prop 传递给组件。`children` 属性有多个适用于此概念的 TypeScript 类型，其中最通用的是 `React` 库中的 `React.ReactNode`。

最后，我们的整个 React 应用程序都引入了 TypeScript 类型，从而很容易在编译时发现类型错误。将 TypeScript 添加到 React 应用程序时，首先需要在函数的参数中添加类型定义。

这些函数可以是 JavaScript 函数，自定义的 React hook 或 React 函数组件。当我们在使用 React 时，了解表单元素，事件和 JSX 的特定类型很重要。

练习

- 检查 上一节的源码[191]。
- 确认 上一节之后的变更[192]。
- 深入研究 React + TypeScript 备忘单[193]，因为这里也涵盖了我们在本节中面临的最常见用例。无需从头至尾了解所有内容。
- 在接下来进行各章节中的学习时，请使用 TypeScript 删除或保留你的类型。如果你选择后者，则每当遇到编译错误时都添加新的类型。

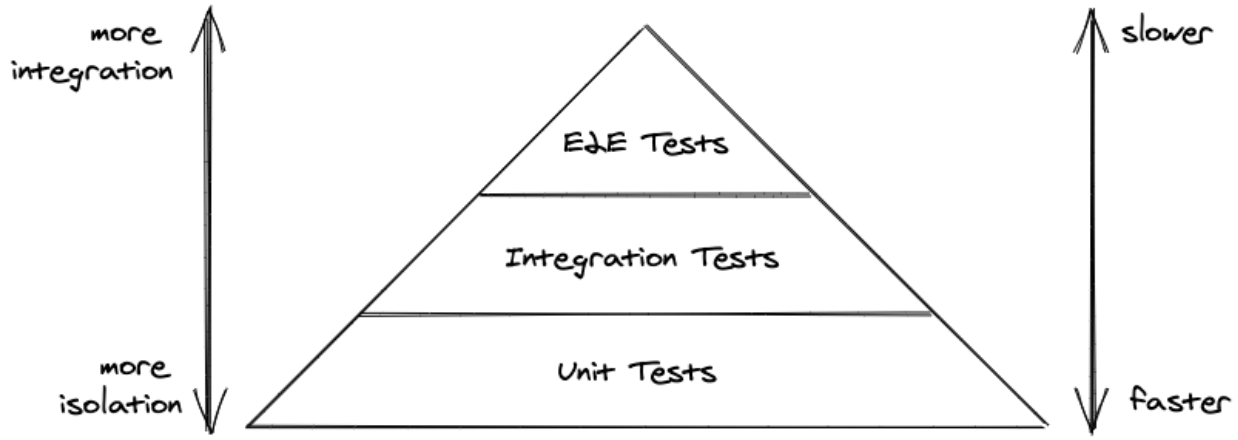
[191] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/TypeScript-in-React
[192] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/react-modern-final...hs/TypeScript-in-React?expand=1
[193] https://github.com/typescript-cheatsheets/react-typescript-cheatsheet#reacttypescript-cheatsheets

从单元测试到集成测试

测试代码对于编程来说是至关重要的，应该被视为认真的开发人员的一项必修课。在产品环境使用前，我们要先验证我们的源代码的质量和功能。测试金字塔[194]将作为我们的指南。

测试金字塔包含了端到端测试，基础测试和单元测试。单元测试用于小而独立的代码块，比如一个函数或组件。集成测试帮助我们弄清楚这些单元是否能够很好地协作。端到端测试模拟了一个真实的情景，比如一个网页应用的登录流程。单元测试的编写很快，维护也很简单；端到端测试则相反。

我们希望有许多单元测试来覆盖函数和组件。之后，我们可以使用一部分集成测试来确保最重要的函数和组件按照我们预期的方式协作。最后，我们或许需要一些端到端测试来模拟关键情景。在这次学习中，我们将覆盖单元测试和集成测试，以及一种被称为快照测试的特殊的组件测试技术。端到端测试将是本次练习的一部分。



由于有许多测试库[195]，对于初学者来说从中选择一个可能是一个挑战。我们将使用 FaceBook 的 Jest[196] 作为测试框架，以避免让本教程过于主观。其他大部分 React 的测试库以 Jest 作为基础，所以这也是其足够好的一个证明。

从单元测试到集成测试

通常来说单元测试和集成测试的边界是模糊的。测试包含 Item 组件的 List 组件可以被看做是一个集成测试，但是它也可以被看做是对于两个高耦合组件的单元测试。在这一小节中，我们从单元测试开始，逐渐向集成测试过渡。这两者之间的过渡如光谱一样模糊没有边界。

让我们在 *src/App.test.js* 目录下以一个伪测试作为开始吧：

[194] https://martinfowler.com/articles/practical-test-pyramid.html
[195] https://www.robinwieruch.de/react-testing-tutorial
[196] https://jestjs.io/

```
describe('something truthy', () => {
  it('true to be true', () => {
    expect(true).toBe(true);
  });
});
```

幸运的是，create-react-app 自带 Jest。你可以在命令行中使用交互式 create-react-app 测试脚本来运行测试。所有测试用例的输出都将显示在你的命令行界面中。

```
npm test
```

Jest 在命令运行的时候会匹配所有文件名后缀为 *test.js* 的文件。运行成功的测试将显示为绿色；失败的则显示为红色：

```
describe('something truthy', () => {
  it('true to be true', () => {
    expect(true).toBe(false);
  });
});
```

Jest 里的测试由测试套件 (`describe`) 组成，而测试套件又由测试用例 (`it`) 组成，测试用例中的断言 (`expect`) 来决定测试是否通过：

```
// test suite
describe('truthy and falsy', () => {
  // test case
  it('true to be true', () => {
    // test assertion
    expect(true).toBe(true);
  });

  // test case
  it('false to be false', () => {
    // test assertion
    expect(false).toBe(false);
  });
});
```

`it` 块描述了一个测试用例。它附带了一个在测试成功或失败时返回的测试描述。我们也可以将这个块写入 `describe` 块，该块用许多个 `it` 块定义了一个针对特殊组件的测试套件。这两个块都用来组织你的测试用例。需要注意的是 `it` 函数在 JavaScript 社区中被称为单测试用例函数；但是在 Jest 中，`it` 通常被用做 `test` 函数的别名。

```
describe('something truthy', () => {
  test('true to be true', () => {
    expect(true).toBe(false);
  });
});
```

为了在 Jest 中使用 React 组件，我们引入了一个在测试环境渲染组件的实用的库：

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

当然了，在你测试你的组件前，你必须从你的 *src/App.js* 文件中将其导出：

```
...

export default App;

export { SearchForm, InputWithLabel, List, Item };
```

在 *src/App.test.js* 文件中将组件连同之前安装好的工具库一起引入：

```
import React from 'react';
import renderer from 'react-test-renderer';

import App, { Item, List, SearchForm, InputWithLabel } from './App';
```

为 Item 组件编写第一个测试。该测试用例用工具库渲染出给定的 `item` 组件：

```
import React from 'react';
import renderer from 'react-test-renderer';

import App, { Item, List, SearchForm, InputWithLabel } from './App';

describe('Item', () => {
  const item = {
    title: 'React',
    url: 'https://reactjs.org/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  };
```

```
  it('renders all properties', () => {
    const component = renderer.create(<Item item={item} />);

    expect(component.root.findByType('a').props.href).toEqual(
      'https://reactjs.org/'
    );
  });
});
```

组件或元素的属性信息可以通过 porps 属性获取到。在测试的断言中，我们获取到锚点标签 (a) 和它的 href 属性，同时执行了一个可用性检查。如果测试变绿，我们就可以确定 Item 组件锚点标签的 href 属性被设置了正确的 url 属性。在同一个测试用例中，我们可以为其他 Item 组件的属性增加更多的测试断言：

```
describe('Item', () => {
  const item = {
    title: 'React',
    url: 'https://reactjs.org/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  };

  it('renders all properties', () => {
    const component = renderer.create(<Item item={item} />);

    expect(component.root.findByType('a').props.href).toEqual(
      'https://reactjs.org/'
    );

    expect(
      component.root.findAllByType('span')[1].props.children
    ).toEqual('Jordan Walke');
  });
});
```

因为有多个 span 元素，我们先获取到所有的 span 元素然后选择第二个（索引为1，因为从0开始计数）再将其 React 的 children 属性与 Item 组件的 author 作对比。不过，这个测试还不够彻底。一旦 Item 组件中的 span 元素的顺序发生变化，测试就会失败。为了避免这样的情况，可以将断言改成：

```
describe('Item', () => {
  const item = { ... };

  it('renders all properties', () => {
    ...

    expect(
      component.root.findAllByProps({ children: 'Jordan Walke' })
        .length
    ).toEqual(1);
  });
});
```

这下测试断言不再那么具体了。它只是测试了是否有一个元素具有 Item 的 `author` 属性。你可以用这种方式去测试 Item 其他的属性。或者你可以把它们留作后面的练习。

我们已经测试了 Item 组件是否如期渲染了字符或者 HTML 属性 (`href`)，但是我们还没有测试回调处理函数。下面的测试用例通过 `button` 元素的 `onClick` 属性模拟了一个点击事件，从而实现了这个断言。

```
describe('Item', () => {
  const item = { ... };

  it('renders all properties', () => {
    ...
  });

  it('calls onRemoveItem on button click', () => {
    const handleRemoveItem = jest.fn();

    const component = renderer.create(
      <Item item={item} onRemoveItem={handleRemoveItem} />
    );

    component.root.findByType('button').props.onClick();

    expect(handleRemoveItem).toHaveBeenCalledTimes(1);
    expect(handleRemoveItem).toHaveBeenCalledWith(item);

    expect(component.root.findAllByType(Item).length).toEqual(1);
  });
});
```

Jest 允许我们将一个特殊的测试函数当做 Item 组件的 prop 传入。这些函数被称为 **spy** 、

stub 或 **mock**；它们被用做不同的测试场景。`jest.fn()` 将返回一个真实函数的 mock，这可以让我们捕捉到函数什么时候被调用。因此，我们可以使用 `toHaveBeenCalledTimes` 断言来断言函数被调用的次数；也可以用 `toHaveBeenCalledWith` 来验证传入函数的参数。

Item 组件的单元测试到这里就完成了，因为我们已经测试了输入 (`item`) 和输出 (`onRemoveItem`)。不要将这两者与已经测试过的函数组件的输入（参数）和输出（JSX）混淆。最后的一个改进是增加一个公有的设置函数来让 Item 组件的测试更加简洁：

```
describe('Item', () => {
  const item = { ... };
  const handleRemoveItem = jest.fn();

  let component;

  beforeEach(() => {
    component = renderer.create(
      <Item item={item} onRemoveItem={handleRemoveItem} />
    );
  });

  it('renders all properties', () => {
    expect(component.root.findByType('a').props.href).toEqual(
      'https://reactjs.org/'
    );

    ...
  });

  it('calls onRemoveItem on button click', () => {
    component.root.findByType('button').props.onClick();

    ...
  });
});
```

在测试中，一个公有的设置（或拆卸）函数可以消除重复代码。因为组件必须在两个测试中渲染，而且两次渲染的 props 是相同的，所以我们可以将这部分代码放到一个公有的设置函数中。从这里开始，我们将进入 List 组件的测试：

```
...

describe('Item', () => {
  ...
});

describe('List', () => {
  const list = [
    {
      title: 'React',
      url: 'https://reactjs.org/',
      author: 'Jordan Walke',
      num_comments: 3,
      points: 4,
      objectID: 0,
    },
    {
      title: 'Redux',
      url: 'https://redux.js.org/',
      author: 'Dan Abramov, Andrew Clark',
      num_comments: 2,
      points: 5,
      objectID: 1,
    },
  ];

  it('renders two items', () => {
    const component = renderer.create(<List list={list} />);

    expect(component.root.findAllByType(Item).length).toEqual(2);
  });
});
```

该测试直接检查了在 List 组件中是否以两个 item 属性渲染了两个 Item 组件。你可以使用上一个 Item 组件测试中类似的方式，接着测试 List 组件的每一个回调处理函数 (`onRemoveItem`) 是否在每一个 Item 组件中被调用。那么这个这样的测试还是一个单元测试吗？或者说这已经属于集成测试了呢？

先把这个问题放一放，我们将继续测试带有 InputWithLabel 组件的 SearchForm 组件：

```
describe('SearchForm', () => {
  const searchFormProps = {
    searchTerm: 'React',
    onSearchInput: jest.fn(),
    onSearchSubmit: jest.fn(),
  };

  let component;

  beforeEach(() => {
    component = renderer.create(<SearchForm {...searchFormProps} />);
  });

  it('renders the input field with its value', () => {
    const value = component.root.findByType(InputWithLabel).props
      .value;

    expect(value).toEqual('React');
  });
});
```

在这个测试中，我们断言了 InputWithLabel 组件是否从 SearchForm 组件中接收到了正确的 prop。本质上来说这个测试在 InputWithLabel 组件之前就结束了，因为它只测试了组件的接口 (props)。可以说它仍然是一个单元测试，因为 InputWithLabel 组件的内部实现细节可以在不改变接口的情况下改变。你可以通过改变测试来获取到 InputWithLabel 组件内 input 组件的字段，因为所有的子组件和元素也会被渲染。

```
describe('SearchForm', () => {
  ...

  it('renders the input field with its value', () => {
    const value = component.root.findByType('input').props.value;

    expect(value).toEqual('React');
  });
});
```

这是我们第一次在 SearchForm 组件和 InputWithLabel 组件之间进行集成测试，这两个组件并不像 List 组件和 Item 组件那样高耦合。InputWithLabel 组件可以在其他组件中使用（高度可重用），而 Item 组件基本上是 List 组件中不可重用的部分。

```
describe('SearchForm', () => {
  const searchFormProps = {
    searchTerm: 'React',
    onSearchInput: jest.fn(),
    onSearchSubmit: jest.fn(),
  };

  ...

  it('changes the input field', () => {
    const pseudoEvent = { target: 'Redux' };

    component.root.findByType('input').props.onChange(pseudoEvent);

    expect(searchFormProps.onSearchInput).toHaveBeenCalledTimes(1);
    expect(searchFormProps.onSearchInput).toHaveBeenCalledWith(
      pseudoEvent
    );
  });

  it('submits the form', () => {
    const pseudoEvent = {};

    component.root.findByType('form').props.onSubmit(pseudoEvent);

    expect(searchFormProps.onSearchSubmit).toHaveBeenCalledTimes(1);
    expect(searchFormProps.onSearchSubmit).toHaveBeenCalledWith(
      pseudoEvent
    );
  });
});
```

和 Item 组件一样，最后两个测试也是对组件的回调处理函数进行了测试。对于 SearchForm 组件的接口和与 InputWithLabel 组件的集成来讲，所有的输入（非函数 props）和输出（回调处理函数）props 都被测试了。

你也可以测试像禁用按钮这样的边界情况。在已经渲染的测试组件中提供了一个 `update()` 方法，它可以帮助我们在特殊情况下为组件提供新的 props。

```
describe('SearchForm', () => {
  const searchFormProps = {
    searchTerm: 'React',
    onSearchInput: jest.fn(),
    onSearchSubmit: jest.fn(),
  };

  let component;

  beforeEach(() => {
    component = renderer.create(<SearchForm {...searchFormProps} />);
  });

  ...

  it('disables the button and prevents submit', () => {
    component.update(
      <SearchForm {...searchFormProps} searchTerm="" />
    );

    expect(
      component.root.findByType('button').props.disabled
    ).toBeTruthy();
  });
});
```

现在，我们将在应用程序的组件层次结构中再次深入。App 组件获取 list 数据后，会将其提供给 List 组件。在测试中引入 App 组件后，一个本地的测试会是这样：

```
describe('App', () => {
  it('succeeds fetching data with a list', () => {
    const list = [
      {
        title: 'React',
        url: 'https://reactjs.org/',
        author: 'Jordan Walke',
        num_comments: 3,
        points: 4,
        objectID: 0,
      },
      {
        title: 'Redux',
        url: 'https://redux.js.org/',
```

```
      author: 'Dan Abramov, Andrew Clark',
      num_comments: 2,
      points: 5,
      objectID: 1,
    },
  ];

  const component = renderer.create(<App />);

  expect(component.root.findByType(List).props.list).toEqual(list);
 });
});
```

在实际的 App 组件中，我们使用第三方库（axios）向一个远程 API 发出请求。这个 API 返回的数据是我们在测试中无法预知的，所以我们必须对其进行 mock。Jest 提供了 mock 整个库及其方法的机制。在本例中，我们要模拟 axios 的 `get()` 方法来返回我们想要的数据。

```
import React from 'react';
import renderer from 'react-test-renderer';
import axios from 'axios';

jest.mock('axios');

...

describe('App', () => {
 it('succeeds fetching data with a list', () => {
  const list = [ ... ];

  const promise = Promise.resolve({
    data: {
      hits: list,
    },
  });

  axios.get.mockImplementationOnce(() => promise);

  const component = renderer.create(<App />);

  expect(component.root.findByType(List).props.list).toEqual(list);
 });
});
```

测试是同步读取的，但我们还是要处理异步数据。当组件的状态更新时，组件应该重新渲染。我们可以用我们的工具库和 async/await 来做到：

```
describe('App', () => {
  it('succeeds fetching data with a list', async () => {
    const list = [ ... ];

    const promise = Promise.resolve({
      data: {
        hits: list,
      },
    });

    axios.get.mockImplementationOnce(() => promise);

    let component;

    await renderer.act(async () => {
      component = renderer.create(<App />);
    });

    expect(component.root.findByType(List).props.list).toEqual(list);
  });
});
```

我们并没有渲染 App 组件，而是通过 mock 获取数据的方法来模拟远程 API 的响应。为了专注于 happy path，我们告诉测试将组件视为异步更新组件。你可以将类似的策略应用到 unhappy path 上：

```
describe('App', () => {
  it('succeeds fetching data with a list', async () => {
    ...
  });

  it('fails fetching data with a list', async () => {
    const promise = Promise.reject();

    axios.get.mockImplementationOnce(() => promise);

    let component;

    await renderer.act(async () => {
      component = renderer.create(<App />);
```

```
    });

    expect(component.root.findByType('p').props.children).toEqual(
      'Something went wrong ...'
    );
  });
});
```

现在数据的获取和与远程 API 的集成已经测试完毕。我们从专注于单个组件的单元测试，转向了对多个组件的测试，以及它们与第三方如 axios 和远程 API 的集成。

快照测试

Jest 还可以让你对你的渲染组件进行快照，将其与未来抓取的快照结果进行对比，并在发现变动时通知你。然后根据所需的结果，可以接受或拒绝更改。这种机制是对单元和集成测试的很好补充，因为它只测试渲染输出的差异，而不需要大量的维护成本。要查看它的实际操作，请用你的第一个快照测试来扩展 Item 组件的测试套件：

```
describe('Item', () => {
  ...

  beforeEach(() => {
    component = renderer.create(
      <Item item={item} onRemoveItem={handleRemoveItem} />
    );
  });

  ...

  test('renders snapshot', () => {
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

再次运行你的测试，观察它们如何成功或失败的。一旦我们更改 *src/App.js* 文件中 Item 组件的渲染块的输出，由此改变了返回的 HTML 结构，快照测试就会失败。这时，我们就可以决定是更新快照还是对 Item 组件进行调查。

Jest 将快照存储在一个文件夹中，这样它就可以对未来的快照测试进行差别验证。用户可以跨团队共享这些快照以进行版本控制（例如：git）。第一次运行快照测试时，会在项目文件夹中创建快照文件。当再次运行测试时，快照按照预期将与最近一次测试运行的版本相匹配。这就是我们如何确保 DOM 保持不变的方法。

练习

- 检查上一节的源码[197]。
  - 确认上一节之后的变更[198]。
- 为所有其他的组件加上一个快照测试。
- 阅读更多关于测试 React 组件[199]的信息。
  - 阅读更多关于 Jest[200] 和 React 中的 Jest[201] 用于单元测试、集成测试和快照测试的信息。
- 阅读更多关于 React 中端到端测试[202]的信息。
- 在接下来的章节中继续学习课程的同时，保持你的测试结果为绿色，并且在你任何时候觉得应该加测试的时候加上新的测试。

---

[197] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/react-testing

[198] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/react-modern-final...hs/react-testing?expand=1

[199] https://www.robinwieruch.de/react-testing-tutorial

[200] https://jestjs.io/

[201] https://www.robinwieruch.de/react-testing-jest/

[202] https://www.robinwieruch.de/react-testing-cypress

React 项目结构

一个文件中有多个组件，你可能会好奇为什么一开始我们没有把在 *src/App.js* 文件里的组件放入不同文件里。我们已经有了多个组件在文件中，可以被定义在他们自己的文件/文件夹中（也称为模块）。为了方便学习，把这些组件放在一个地方更加实用。一旦我们应用需要扩展，我们将考虑拆分这些组件进入多个模块，以便它正确的扩展。

在我们重组 React 项目之前，请回顾 JavaScript 的导入和导出语句[203]。在 JavaScript 中，导入和导出文件是两个基本的概念，你必须在 React 之前学习。没有唯一正确的方法构建 React 项目，因为它们会随着项目结构自然发展。

为了学习这个过程，我们将完成一个简单的关于项目文件夹/文件结构的重构。之后，通常将会有其他的选项来重组这个项目或者 React 项目。你仍然可以继续用重组后项目，即使我们将继续开发 *src/App.js* 文件，为了保持简单。

在项目的文件夹的命令上，导航到 *src/* 文件夹并创建以下组件专用的文件：

Command Line

```
cd src
touch List.js InputWithLabel.js SearchForm.js
```

将 *src/App.js* 文件中的每个组件移动到它自己的文件，除了 List 组件必须与 *src/List.js* 中的 Item 组件共享它的位置。然后在每个文件中确保引入了 React 并从文件中导出需要被使用的组件。例如，*src/List.js* 文件：

src/List.js

```
import React from 'react';

const List = ({ list, onRemoveItem }) =>
  list.map(item => (
    <Item
      key={item.objectID}
      item={item}
      onRemoveItem={onRemoveItem}
    />
  ));

const Item = ({ item, onRemoveItem }) => (
  <div>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
```

[203] https://www.robinwieruch.de/javascript-import-export

```
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
    <span>
     <button type="button" onClick={() => onRemoveItem(item)}>
      Dismiss
     </button>
    </span>
   </div>
);

export default List;
```

由于仅有 List 组件使用了 Item 组件，我们可以保持它在相同的文件。如果 Item 组件在其他地方被使用，而导致情况变化，我们会给 Item 组件提供它自己的文件。*src/SearchForm.js* 文件中的 SearchForm 组件必须引入 InputWithLabel 组件。像 Item 组件，我们可以把 InputWithLabel 组件留在 SearchForm 旁，但是我们的目标是让 InputWithLabel 组件和其他组件可以重复使用，我们最后将引入它：

src/SearchForm.js

```
import React from 'react';

import InputWithLabel from './InputWithLabel';

const SearchForm = ({
  searchTerm,
  onSearchInput,
  onSearchSubmit,
}) => (
  <form onSubmit={onSearchSubmit}>
    <InputWithLabel
      id="search"
      value={searchTerm}
      isFocused
      onInputChange={onSearchInput}
    >
      <strong>Search:</strong>
    </InputWithLabel>

    <button type="submit" disabled={!searchTerm}>
      Submit
    </button>
```

```
  </form>
);

export default SearchForm;
```

App 组件必须引入全部它需要渲染的组件。它不需要引入 InputWithLabel，因为它仅被使用在 SearchForm 组件中。

src/App.js

```
import React from 'react';
import axios from 'axios';

import SearchForm from './SearchForm';
import List from './List';

...

const App = () => {
  ...
};

export default App;
```

在其他组件中被使用的组件现在有了自己的文件。仅当组件（比如：Item）被另一个组件（比如：List）专用时，我们将它保持在一个文件里。如果一个组件应作为可重用的组件（比如：InputWithLabel）被使用时，它也会有自己的文件。从这里开始，这里有几个关于构造你的文件夹/文件结构的策略。一个方案是为每一个组件创造一个文件夹：

Project Structure

```
- List/
-- index.js
- SearchForm/
-- index.js
- InputWithLabel/
-- index.js
```

在 *index.js* 文件里保留了关于组件详细的实现，而同一个文件夹中的其他组件有不同的职责，像样式、测试和类型：

Project Structure

```
- List/
-- index.js
-- style.css
-- test.js
-- types.js
```

如果使用 CSS-in-JS，那么就不需要使用 css 文件。关于所有有样式的组件，仍然可以有一个单独的 *style.js* 文件：

Project Structure

```
- List/
-- index.js
-- style.js
-- test.js
-- types.js
```

有时我们需要从面向技术的文件夹结构转向面向域文件夹结构，特别是随着项目的增长。通用的 *shared/* 文件夹被指定的组件跨域共享：

Project Structure

```
- Messages.js
- Users.js
- shared/
-- Button.js
-- Input.js
```

如果你扩展到更深的的文件夹结构中，在面向域的项目结构中每一个组件也将会有它自己的文件夹：

Project Structure

```
- Messages/
-- index.js
-- style.css
-- test.js
-- types.js
- Users/
-- index.js
-- style.css
-- test.js
-- types.js
```

```
- shared/
-- Button/
--- index.js
--- style.css
--- test.js
--- types.js
-- Input/
--- index.js
--- style.css
--- test.js
--- types.js
```

从小项目到大项目，这里有很多方式构建 React 项目。简单到复杂的文件夹结构，一级嵌套到二级文件嵌套，靠近实现逻辑，有专门的文件夹针对样式、类型和测试。关于文件夹/文件结构没有绝对正确的方法。

项目的需求会随着时间而变化，其结构也会随着变化。如果认为是正确的，保持所有的资源在一个文件里面，这里没有任何规则反对它。只需保持嵌套结构简单，否则你会迷失在文件夹中。

练习

- 检查上一节的源码[204]
- 确认上一节之后的变更[205]
- 阅读更多关于 JavaScript 的输入和输出语句[206]. * 阅读更多关于 React 文件夹结构[207]
- 如果你有信心，就保持当前的文件夹结构。正在进行的部分将被省略，仅使用 *src/App.js* 文件。

[204] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/React-Folder-Structure
[205] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/react-modern-final...hs/React-Folder-Structure?expand=1
[206] https://www.robinwieruch.de/javascript-import-export
[207] https://www.robinwieruch.de/react-folder-structure

真实 React 世界（高级）

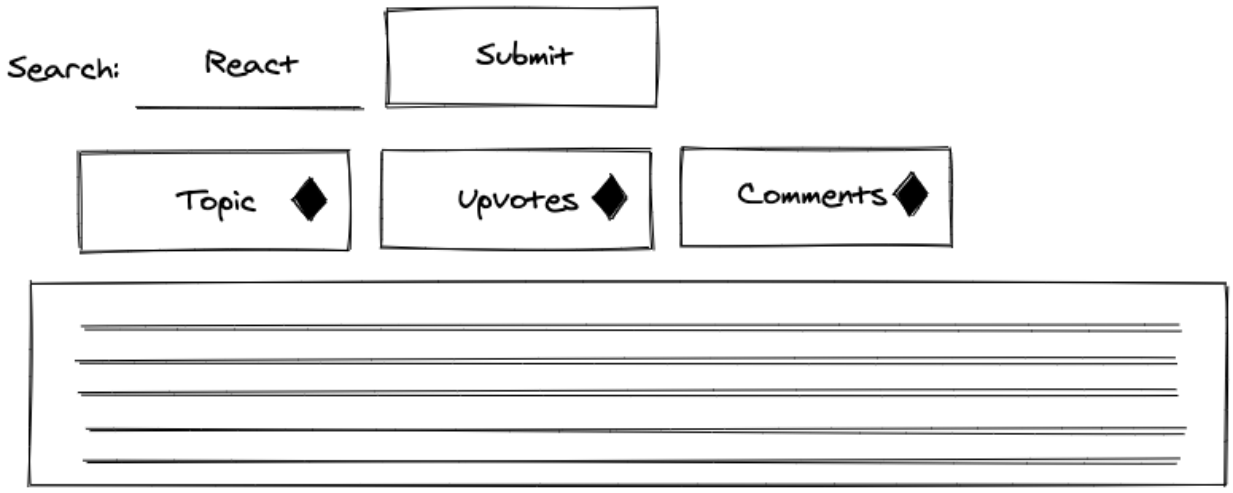
我们已经介绍了 React 大部分基础知识，它的遗留特性以及维护应用程序的技术。现在是时候开发真实世界的 React 应用程序。下面跟着的每一节将会有一个任务。尝试首先在没有选项提示的情况下去解决这些问题，但是要注意，这些任务在你第一次尝试时，是具有挑战性的。如果你需要帮助，使用选项提示或者按照本节的说明操作。

排序

任务: 在处理 items 列表时，经常包括互动，让数据更加能被用户接受。到目前为止，每一个 item 的属性都被列出来了。为了让它更加具有探索性，这个列表应该根据 title，author，comments，和 points 进行升序或者降序，对每个属性进行排序。按照一个方向排序就好，因为另一个方向的排序在下一章的内容。

可选择的提示:

- 在 App 组件或者 List 组件中，引入一种新的排序状态。
- 对于每个属性（例如：`title`, `author`, `points`, `num_comments`,）实现一个 HTML 按钮，用来设置这个属性的排序状态。
- 使用排序状态对 `list` 应用适当的排序函数
- 使用工具库，比如使用 Lodash[208] 的 `sortBy` 函数。



我们将把数据列表当作一个表来看。每一行代表列表里的一个 Item，每一个列代表 Item 的一个属性。针对每一列，标题提供了用户更多的指导：

[208] https://lodash.com/

src/App.js

```
const List = ({ list, onRemoveItem }) => (
  <div>
    <div style={{ display: 'flex' }}>
      <span style={{ width: '40%' }}>Title</span>
      <span style={{ width: '30%' }}>Author</span>
      <span style={{ width: '10%' }}>Comments</span>
      <span style={{ width: '10%' }}>Points</span>
      <span style={{ width: '10%' }}>Actions</span>
    </div>

    {list.map(item => (
      <Item
        key={item.objectID}
        item={item}
        onRemoveItem={onRemoveItem}
      />
    ))}
  </div>
);
```

我们使用的是最基本的内联样式布局。为了让标题布局匹配行，也在给 Item 组件中的每一行布局：

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <div style={{ display: 'flex' }}>
    <span style={{ width: '40%' }}>
      <a href={item.url}>{item.title}</a>
    </span>
    <span style={{ width: '30%' }}>{item.author}</span>
    <span style={{ width: '10%' }}>{item.num_comments}</span>
    <span style={{ width: '10%' }}>{item.points}</span>
    <span style={{ width: '10%' }}>
      <button type="button" onClick={() => onRemoveItem(item)}>
        Dismiss
      </button>
    </span>
  </div>
);
```

在当前的实现中，我们将移除样式属性，因为它占用了很多空间并且干扰了实际的实现逻辑（因此提取它到适当的 CSS）。但是我鼓励你留着它。

List 组件将处理这个新的排序状态。App 组件也可以这么做，但是仅在 List 里面，所以我们将状态管理提升到它的状态管理。排序状态初始值是 `'NONE'`，因此列表中的 items 会按照请求 API 获取的数据的顺序显示。此外，我们添加了一个新的处理程序来设置排序状态，根据一个特定的键。

src/App.js

```
const List = ({ list, onRemoveItem }) => {
  const [sort, setSort] = React.useState('NONE');

  const handleSort = sortKey => {
    setSort(sortKey);
  };

  return (
    ...
  );
};
```

在 List 组件的头部，按钮可以帮助我们设置每个列/属性的排序状态。一个内联处理程序被用来针对特定的键（`sortKey`）。当点击“Title”栏按钮时，`'TITLE'` 会变成新的排序状态。

src/App.js

```
const List = ({ list, onRemoveItem }) => {
  ...

  return (
    <div>
      <div>
        <span>
          <button type="button" onClick={() => handleSort('TITLE')}>
            Title
          </button>
        </span>
        <span>
          <button type="button" onClick={() => handleSort('AUTHOR')}>
            Author
          </button>
        </span>
        <span>
          <button type="button" onClick={() => handleSort('COMMENT')}>
            Comments
          </button>
        </span>
```

```
      <span>
        <button type="button" onClick={() => handleSort('POINT')}>
          Points
        </button>
      </span>
      <span>Actions</span>
    </div>

    {list.map(item => ... )}
  </div>
 );
};
```

新功能的状态管理已经被实现，但是当按钮被点击的时候我们不会看到任何东西。这是因为排序机制没有被应用到新的 `list`。

在 JavaScript 中，对数组进行排序是不是一个简单的事情，因为每一个 JavaScript 原始类型（比如：string，boolean，number）都有边缘问题，当数组按照它的属性进行排序的时候。我们将使用第三方库 Lodash[209] 来解决这个问题，它自带了很多 JavaScript 工具函数（比如：`sortBy`）。首先，在命令行安装它：

Command Line

```
npm install lodash
```

第二，在你文件的顶部，引入排序工具函数：

src/App.js

```
import React from 'react';
import axios from 'axios';
import { sortBy } from 'lodash';

...
```

第三，创造一个 JavaScript 对象（也称为字典），其中包含所有可能的 `sortKey` 和排序方法的映射。每一个被指定的排序键会被映射到一个函数，该函数对传入的**列表**进行排序。按 `'NONE'` 排序，返回未排序的列表，按 `'POINT'` 排序，返回一个列表，它的 items 是按照 `points` 属性排序的。

[209] https://lodash.com/

src/App.js

```
const SORTS = {
  NONE: list => list,
  TITLE: list => sortBy(list, 'title'),
  AUTHOR: list => sortBy(list, 'author'),
  COMMENT: list => sortBy(list, 'num_comments').reverse(),
  POINT: list => sortBy(list, 'points').reverse(),
};

const List = ({ list, onRemoveItem }) => {
  ...
};
```

有了排序 (sortKey) 状态和所有可能的排序变化，我们就可以对列表进行排序，然后在将其映射到每个 Item 组件上：

src/App.js

```
const List = ({ list, onRemoveItem }) => {
  const [sort, setSort] = React.useState('NONE');

  const handleSort = sortKey => {
    setSort(sortKey);
  };

  const sortFunction = SORTS[sort];
  const sortedList = sortFunction(list);

  return (
    <div>
      ...

      {sortedList.map(item => (
        <Item
          key={item.objectID}
          item={item}
          onRemoveItem={onRemoveItem}
        />
      ))}
    </div>
  );
};
```

就这样完成了。首先我们根据它的 `sortKey` 状态从字典中提取了排序函数。然后，我们在列表数据映射并渲染每个 Item 组件之前，将函数应用到了列表上。同样，排序状态的初始值是 `'NONE'` 时，意味着它不会进行排序。

其次为了给用户更多的互动，我们渲染了更多的 HTMl 按钮。然后，我们通过改变排序状态，添加了关于每个按钮的详细实现。最后，我们使用了排序状态来对实际的列表进行排序。

练习

- 检查上一节的源码[210]
- 确认上一节之后的变更[211]
- 阅读跟更多关于 Lodash[212]。
- 为什么我们使用比如 `points` 和 `num_comments` 数字的属性进行反转排序?
- 针对你当前排序，使用你的样式技巧给用户反馈。这个机制会像给当前选中的排序按钮换颜色一样直接。

[210] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Sort
[211] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/react-modern-final...hs/Sort?expand=1
[212] https://lodash.com/

逆序排序

任务：排序功能可以使用，但是只有一个方向的顺序。实现在点击两次按钮时逆序排序，所以它就变成了在正常（升序）和反向（降序）排序之间切换。

可选的提示：

- 考虑逆序或者升序排序可能只是 `sortKey` 的另一个 state （例如 `isReverse`）。
- 在 `handleSort` 处理函数中根据之前的排序设置新的 state。
- 使用新的 `isReverse` 状态，并且使用 JavaScript 数组的 `reverse()` 函数根据字典中的顺序对列表进行排序。

最初的排序方式对字符串和数组都有效，就像 JavaScript 数字的逆序排序一样，它可以将数字从高到低排列。现在我们需要另一种 state 来记录排序的方式，使其更加复杂：

src/App.js

```
const List = ({ list, onRemoveItem }) => {
  const [sort, setSort] = React.useState({
    sortKey: 'NONE',
    isReverse: false,
  });

  ...
};
```

接下来，给排序处理函数添加一些逻辑来判断传入的 `sortKey` 触发器是正常排序还是逆序排序。如果 `sortKey` 与 state 中的相同，则可能是逆序排序，但前提是排序的状态还没有反转：

src/App.js

```
const List = ({ list, onRemoveItem }) => {
  const [sort, setSort] = React.useState({
    sortKey: 'NONE',
    isReverse: false,
  });

  const handleSort = sortKey => {
    const isReverse = sort.sortKey === sortKey && !sort.isReverse;

    setSort({ sortKey: sortKey, isReverse: isReverse });
  };
```

```
  const sortFunction = SORTS[sort.sortKey];
  const sortedList = sortFunction(list);

  return (
    ...
  );
};
```

最后，根据新的 `isReverse` state，使用字典中的排序函数，如果没有内置的 JavaScript reverse 数组方法：

src/App.js

```
const List = ({ list, onRemoveItem }) => {
  const [sort, setSort] = React.useState({
    sortKey: 'NONE',
    isReverse: false,
  });

  const handleSort = sortKey => {
    const isReverse = sort.sortKey === sortKey && !sort.isReverse;

    setSort({ sortKey, isReverse });
  };

  const sortFunction = SORTS[sort.sortKey];

  const sortedList = sort.isReverse
    ? sortFunction(list).reverse()
    : sortFunction(list);

  return (
    ...
  );
};
```

现在逆序排序可以工作了，对于传给 state 更新函数的对象，我们使用了所谓的 **shorthand** 对象初始化符号：

src/App.js

```
const firstName = 'Robin';

const user = {
  firstName: firstName,
};

console.log(user);
// { firstName: "Robin" }
```

当对象中的属性名与变量名相同时，你可以省略键值对，只需要写上名字即可：

src/App.js

```
const firstName = 'Robin';

const user = {
  firstName,
};

console.log(user);
// { firstName: "Robin" }
```

如有必要，阅读更多关于 JavaScript 对象初始化[213]。

练习

- 检查上一节的源码[214].
- 确认上一节之后的变更[215].
- 考虑到这个缺点，将排序状态保留在 List 中，而不是 App 组件中。如果你不知道的话，那就根据 Title 进行排序之后，再搜索其他的内容。如果排序的状态在 App 组件中那会有什么不同呢?
- 使用你的风格设计技巧，给用户反映出当前活动排序及其逆序排序状态。它可以是每个活动排序按钮旁边的向上或向下的箭头[216])。

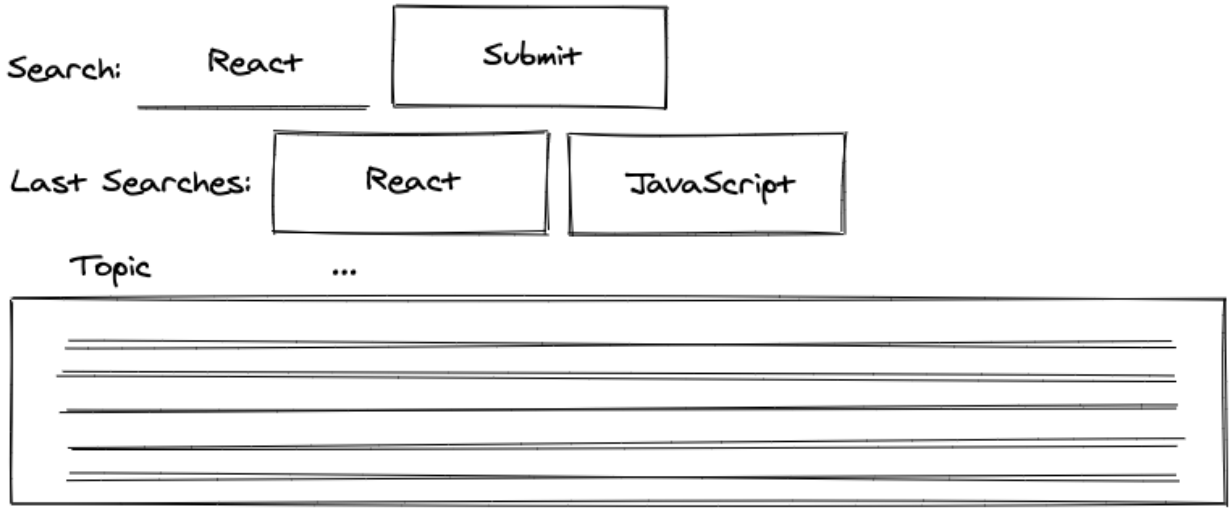
[213] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer
[214] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Reverse-Sort
[215] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Sort...hs/Reverse-Sort?expand=1
[216] https://www.flaticon.com/packs/arrow-set-2

记住上一次的搜索记录

任务：记住最近五次的搜索记录用来访问 API，并提供一个能在搜索词之间快速移动的按钮。当点击按钮时，将再次获取搜索词的内容结果。

提示：

- 不要为这个功能使用新的 state。相反，重用 `url` state 和 `setUrl` state 更新函数来重新从 API 获取内容。将它们适配为 `urls` 状态，并且使用 `setUrls` 设置 `urls`。`urls` 中的最后一个 URL 可以用来获取数据，并且 `urls` 中最后五个 URL 可以用来显示按钮。



首先，我们将所有的 `url` 重构到一个 `urls` state 中，同时将 `setUrl` 重构为 `setUrls` state 更新函数。与其使用字符串 `url` 初始化 state，倒不如把它变成一个数组，把初始的 `url` 作为其唯一的条目：

src/App.js

```
const App = () => {
  ...

  const [urls, setUrls] = React.useState([
    `${API_ENDPOINT}${searchTerm}`,
  ]);

  ...
};
```

第二，使用 `urls` 数组中最后一个 `url` 条目来获取数据，而不是使用当前 `url` state。如果 `urls` 中新添加了另一个 `url`，则就用它来获取数据：

src/App.js

```
const App = () => {

  ...

  const handleFetchStories = React.useCallback(async () => {
    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    try {
      const lastUrl = urls[urls.length - 1];
      const result = await axios.get(lastUrl);

      dispatchStories({
        type: 'STORIES_FETCH_SUCCESS',
        payload: result.data.hits,
      });
    } catch {
      dispatchStories({ type: 'STORIES_FETCH_FAILURE' });
    }
  }, [urls]);

  ...
};
```

第三，不要用 state 更新函数将 `url` 字符串储存为 state，而是将新的 `url` 与之前的 `urls` 数组合并成新的 state：

src/App.js

```
const App = () => {
  ...

  const handleSearchSubmit = event => {
    const url = `${API_ENDPOINT}${searchTerm}`;
    setUrls(urls.concat(url));

    event.preventDefault();
  };

  ...
};
```

每一次搜索时，新的 URL 都会储存在我们的 `urls` state 中。接下来，为最后五个 URL 中每个 URL 渲染一个按钮，我们将为这些按钮添加一个新的通用的处理函数，并且每次给一个特定的内联函数传递一个特定的 `url`：

src/App.js

```
const getLastSearches = urls => urls.slice(-5);

...

const App = () => {
  ...

  const handleLastSearch = url => {
    // do something
  };

  const lastSearches = getLastSearches(urls);

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <SearchForm ... />

      {lastSearches.map(url => (
        <button
          key={url}
          type="button"
          onClick={() => handleLastSearch(url)}
        >
          {url}
        </button>
      ))}

      ...
    </div>
  );
};
```

接下来，不要把按钮中的最后一个搜索的 URL 作为按钮的文本展示出来，而是用一个空字符串代替 API 的端点，只显示搜索词：

src/App.js

```
const extractSearchTerm = url => url.replace(API_ENDPOINT, '');

const getLastSearches = urls =>
  urls.slice(-5).map(url => extractSearchTerm(url));

...

const App = () => {
  ...

  const lastSearches = getLastSearches(urls);

  return (
    <div>
      ...

      {lastSearches.map(searchTerm => (
        <button
          key={searchTerm}
          type="button"
          onClick={() => handleLastSearch(searchTerm)}
        >
          {searchTerm}
        </button>
      ))}

      ...
    </div>
  );
};
```

`getLastSearches` 函数目前返回的是搜索词而不是 URL。实际上，传递给内联函数的是 `searchTerm`，而不是 `url`。通过映射到 `getLastSearches` 中的 `url` 列表，我们能在数组的 map 方法中提取每个 `url` 的搜索词。为了更加简洁，它也可以这样写：

src/App.js

```
const getLastSearches = urls =>
  urls.slice(-5).map(extractSearchTerm);
```

现在，因为点击其中一个按钮应该触发一次新的搜索，所以我们将为每个按钮使用的新处理函数提供功能。由于我们使用 `urls` state 来获取数据，并且我们知道最后一个 URL 总是用来获取数据的，所以在 `urls` 列表中添加一个新的 `url` 来触发另一次搜索请求：

src/App.js

```
const App = () => {
  ...

  const handleLastSearch = searchTerm => {
    const url = `${API_ENDPOINT}${searchTerm}`;
    setUrls(urls.concat(url));
  };

  ...
};
```

如果你将这个新的处理函数的实现逻辑和 `handleSearchSubmit` 对比一下，你可能会发现一些共同的功能。将这些共同的功能提取到一个新的处理函数和新提取的实用工具函数中：

src/App.js

```
const getUrl = searchTerm => `${API_ENDPOINT}${searchTerm}`;

...

const App = () => {
  ...

  const handleSearchSubmit = event => {
    handleSearch(searchTerm);

    event.preventDefault();
  };

  const handleLastSearch = searchTerm => {
    handleSearch(searchTerm);
  };

  const handleSearch = searchTerm => {
    const url = getUrl(searchTerm);
    setUrls(urls.concat(url));
  };

  ...
};
```

新的实用工具函数可以在 App 组件的其他任何地方使用。如果你提取的功能可以被两处使用，那么一定要检查一下它是否可以被第三方使用。

src/App.js

```
const App = () => {
  ...

  // important: still wraps the returned value in []
  const [urls, setUrls] = React.useState([getUrl(searchTerm)]);

  ...
};
```

该功能应该可以工作了，但如果同一个搜索词被多次使用，它就会出问题了，因为 `searchTerm` 被作为 `key` 属性用于每个按钮元素。我们可以通过将 `key` 与映射数组的 `index` 串联起来，使得 `key` 更加具体。

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      ...

      {lastSearches.map((searchTerm, index) => (
        <button
          key={searchTerm + index}
          type="button"
          onClick={() => handleLastSearch(searchTerm)}
        >
          {searchTerm}
        </button>
      ))}

      ...
    </div>
  );
};
```

这并不是完美的解决方案，因为 `index` 不是一个稳定的 key（尤其是当向列表中添加条目时）；然而，在这种场景下它并没有被破坏。功能现在可以运行了，但是你可以通过下面的任务对 UX 进行进一步的改进。

更多任务：

- （1）仅以按钮形式展示之前的五个搜索，而当前搜索不需要展示为按钮形式。提示：适配 `getLastSearches` 函数。
- （2）不要显示重复的搜索，搜索两次 “React” 不应该产生两个不同的按钮。提示：适配 `getLastSearches` 函数。
- （3）如果其中一个按钮被点击了，则将 `SearchForm` 组件的输入字段的值设置为最后一次搜索的内容。

5 个渲染的按钮来自于 `getLastSearches` 函数，在这个函数里面，我们取数组 `urls`，并返回其中的最后 5 项。现在我们将修改一下这个函数，让其返回最后 6 项而不是 5 项。之后，只有 5 个之前的搜索结果会以按钮的形式展示出来。

src/App.js

```
const getLastSearches = urls =>
  urls
    .slice(-6)
    .slice(0, -1)
    .map(extractSearchTerm);
```

如果相同的内容连续搜索了多次，就会出现重复的按钮，这很可能不是你想要的行为。如果将连续且相同的搜索合并到一个按钮中，那就是可以接受的了。我们也将在这个函数中解决这个问题。在将数组分割成前 5 次搜索内容之前，先将相同的搜索内容进行分组：

src/App.js

```
const getLastSearches = urls =>
  urls
    .reduce((result, url, index) => {
      const searchTerm = extractSearchTerm(url);

      if (index === 0) {
        return result.concat(searchTerm);
      }

      const previousSearchTerm = result[result.length - 1];

      if (searchTerm === previousSearchTerm) {
        return result;
      } else {
        return result.concat(searchTerm);
      }
    }, [])
    .slice(-6)
    .slice(0, -1);
```

reduce 函数以一个空数组作为其 result 开始，第一次循环我们将从第一个 url 中提取 searchTerm 并合并到 result 中。每一次提取的 searchTerm 都会与之前的进行比较，如果之前的搜索词与当前的不同，则将这个 searchTerm 添加到 result 中。如果搜索词相同，则返回 result 并且不向其添加任何搜索词。

最后，如果最后一个搜索按钮被点击了，SearchForm 的输入字段应该设置为新的 searchTerm。我们可以使用 SearchForm 组件中特定值的 state 更新函数来解决这个问题。

src/App.js

```
const App = () => {
  ...

  const handleLastSearch = searchTerm => {
    setSearchTerm(searchTerm);

    handleSearch(searchTerm);
  };

  ...
};
```

最后，将该功能的新渲染内容提取出来，作为一个独立的组件，以保持 App 组件的轻量简洁：

src/App.js

```
const App = () => {
  ...

  const lastSearches = getLastSearches(urls);

  return (
    <div>
      ...

      <LastSearches
        lastSearches={lastSearches}
        onLastSearch={handleLastSearch}
      />

      ...
    </div>
  );
};
```

```
const LastSearches = ({ lastSearches, onLastSearch }) => (
  <>
    {lastSearches.map((searchTerm, index) => (
      <button
        key={searchTerm + index}
        type="button"
        onClick={() => onLastSearch(searchTerm)}
      >
        {searchTerm}
      </button>
    ))}
  </>
);
```

这个功能并不容易。需要很多 React 的基础知识，同时也需要大量的 JavaScript 知识来完成它。如果你在自己实现它或者按照说明去做时没有遇到任何问题，那么你就已经做的很好了。如果你遇到了这样或那样的问题，也不用太过担心，也许你可以想出另一种解决方法，结果甚至可能比我展示的更加简单。

练习

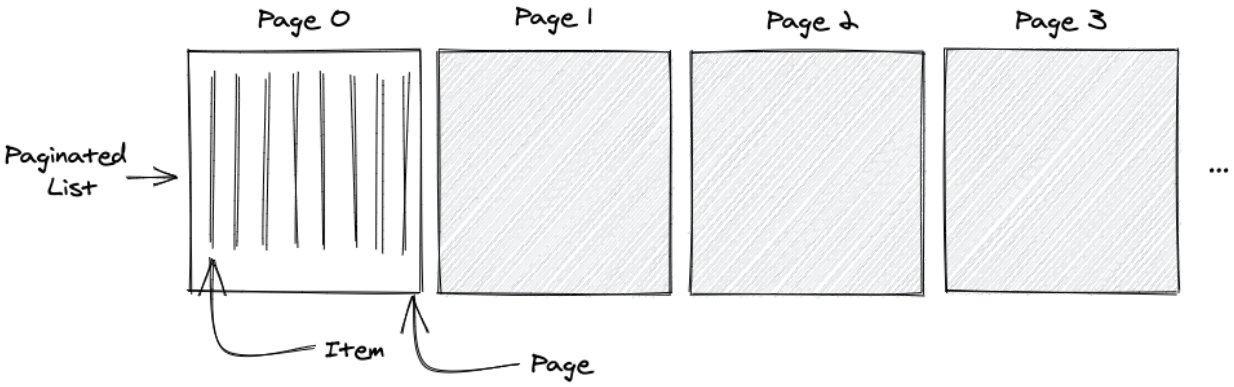
- 检查 上一节的源码[217]
- 确认 上一节之后的变更[218]

[217] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Remember-Last-Searches
[218] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Reverse-Sort...hs/Remember-Last-Searches?expand=1

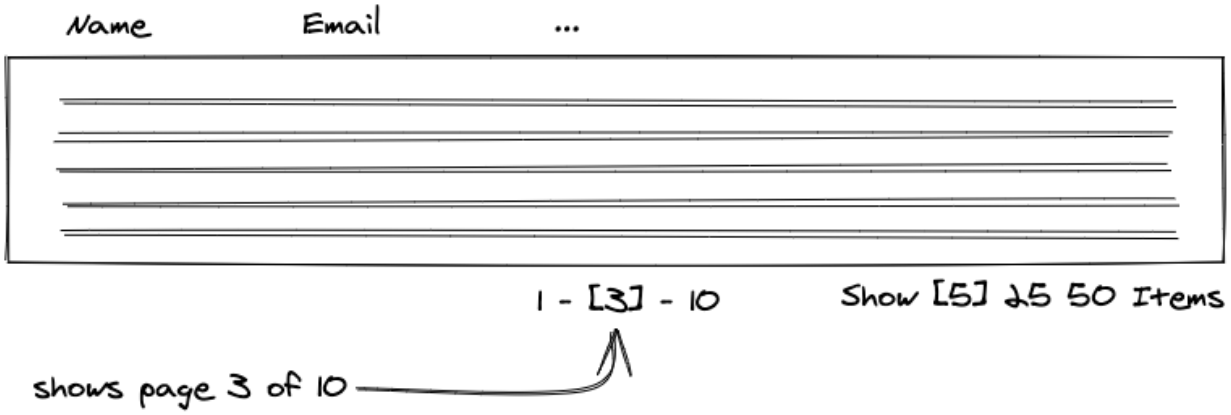
分页查询

通过 Hacker News API 搜索热门故事只是迈向全功能搜索引擎的一步，还有很多方法可以对搜索进行微调。仔细看看数据结构，观察 Hacker News API[219] 如何返回的不仅仅是一个 `hits` 列表。

具体来说，它返回的是一个分页列表。page 属性，在第一个响应中是 `0`，可以用来获取更多的分页列表作为结果。你只需要将下一个有相同搜索词的 page 传递给 API。

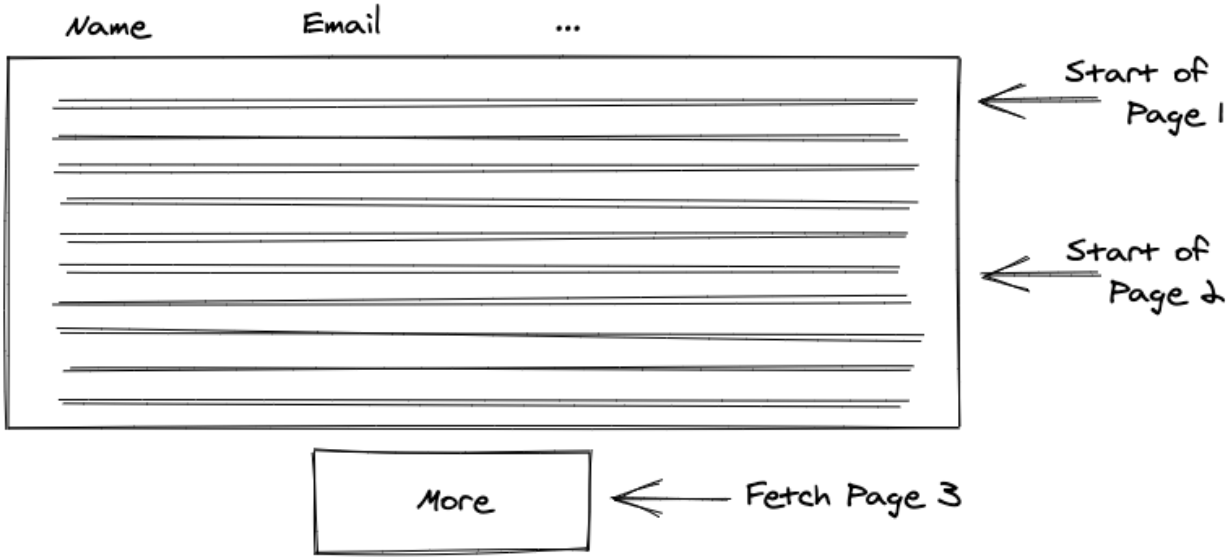


下面展示了如何利用 Hacker News 数据结构实现分页获取。如果你已经习惯了其他应用程序的分页，你的脑海中可能会有一排从 1-10 的按钮–当前选中的页面被高亮显示为 1-[3]-10，点击其中的一个按钮就会获取并显示这个数据子集。



相反，我们将把该功能实现为无限分页。我们不会在点击按钮时呈现一个单一的分页列表，而是将所有的分页列表作为一个列表，用一个按钮来获取下一页。每一个额外的分页列表都会在一个列表的末尾进行连接。

[219] https://hn.algolia.com/api



任务：不是只获取列表的第一页，而是扩展获取后续页面的功能，将其作为点击按钮时的无限分页来实现。

提示：

- 用分页获取所需的参数扩展 `API_ENDPOINT`。
- 从 `result` 中获取数据后，将 `page` 存储为 state。
- 每次搜索都会获取第一页 (`0`) 的数据。
- 每当用一个新的 HTML 按钮触发一个额外的请求时，就获取后续的页面 (`page + 1`)。

首先，扩展 API 常量，以便以后可以处理分页数据。我们将把这一个常量：

src/App.js

```
const API_ENDPOINT = 'https://hn.algolia.com/api/v1/search?query=';

const getUrl = searchTerm => `${API_ENDPOINT}${searchTerm}`;
```

改为一个可合成的 API 常量，其参数如下：

src/App.js

```
const API_BASE = 'https://hn.algolia.com/api/v1';
const API_SEARCH = '/search';
const PARAM_SEARCH = 'query=';

// careful: notice the ? in between
const getUrl = searchTerm =>
  `${API_BASE}${API_SEARCH}?${PARAM_SEARCH}${searchTerm}`;
```

幸运的是，我们不需要调整 API 端点，因为我们为它提取了一个通用的 `getUrl` 函数。但是，有一个地方我们必须在未来解决这个逻辑：

src/App.js

```
const extractSearchTerm = url => url.replace(API_ENDPOINT, '');
```

在接下来的步骤中，仅仅替换掉我们 API 端点的基础是不够的，我们的代码中已经没有了 API 端点。随着 API 端点的参数越来越多，URL 变得更加复杂。它将从 X 变为 Y：

src/App.js

```
// X
https://hn.algolia.com/api/v1/search?query=react

// Y
https://hn.algolia.com/api/v1/search?query=react&page=0
```

最好是提取搜索词，将 `?` 和 `&` 之间的所有内容提取出来。另外，考虑到 `query` 参数是直接放在 `?` 之后的，而所有其他参数如 `page` 都在它之后。

src/App.js

```
const extractSearchTerm = url =>
  url
    .substring(url.lastIndexOf('?') + 1, url.lastIndexOf('&'));
```

键 (`query=`) 也需要替换，只留下值 (`searchTerm`)：

src/App.js

```
const extractSearchTerm = url =>
  url
    .substring(url.lastIndexOf('?') + 1, url.lastIndexOf('&'));
    .replace(PARAM_SEARCH, '');
```

基本上，我们会修剪字符串，直到只留下搜索词：

src/App.js

```
// url
https://hn.algolia.com/api/v1/search?query=react&page=0

// url after  substring
query=react

// url after replace
react
```

从 Hacker News API 返回的结果为我们提供了 page 数据：

src/App.js

```
const App = () => {
  ...

  const handleFetchStories = React.useCallback(async () => {
    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    try {
      const lastUrl = urls[urls.length - 1];
      const result = await axios.get(lastUrl);

      dispatchStories({
        type: 'STORIES_FETCH_SUCCESS',
        payload: {
          list: result.data.hits,
          page: result.data.page,
        },
      });
    } catch {
      dispatchStories({ type: 'STORIES_FETCH_FAILURE' });
    }
```

```
  }, [urls]);

  ...
};
```

我们需要存储这些数据，以便以后进行分页获取：

src/App.js

```
const storiesReducer = (state, action) => {
  switch (action.type) {
    case 'STORIES_FETCH_INIT':
      ...
    case 'STORIES_FETCH_SUCCESS':
      return {
        ...state,
        isLoading: false,
        isError: false,
        data: action.payload.list,
        page: action.payload.page,
      };
    case 'STORIES_FETCH_FAILURE':
      ...
    case 'REMOVE_STORY':
      ...
    default:
      throw new Error();
  }
};

const App = () => {
  ...

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    { data: [], page: 0, isLoading: false, isError: false }
  );

  ...
};
```

用新的 page 参数扩展 API 端点。这个变化被我们之前过早的优化所覆盖，当时我们从 URL 中提取了搜索词。

src/App.js

```
const API_BASE = 'https://hn.algolia.com/api/v1';
const API_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';

// careful: notice the ? and & in between
const getUrl = (searchTerm, page) =>
  `${API_BASE}${API_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}`;
```

接下来，我们必须通过传递 `page` 参数来调整所有 `getUrl` 的调用。由于初始搜索和最后一次搜索总是获取第一页 (`0`)，我们将这一页作为参数传递给函数，以便检索适当的 URL：

src/App.js

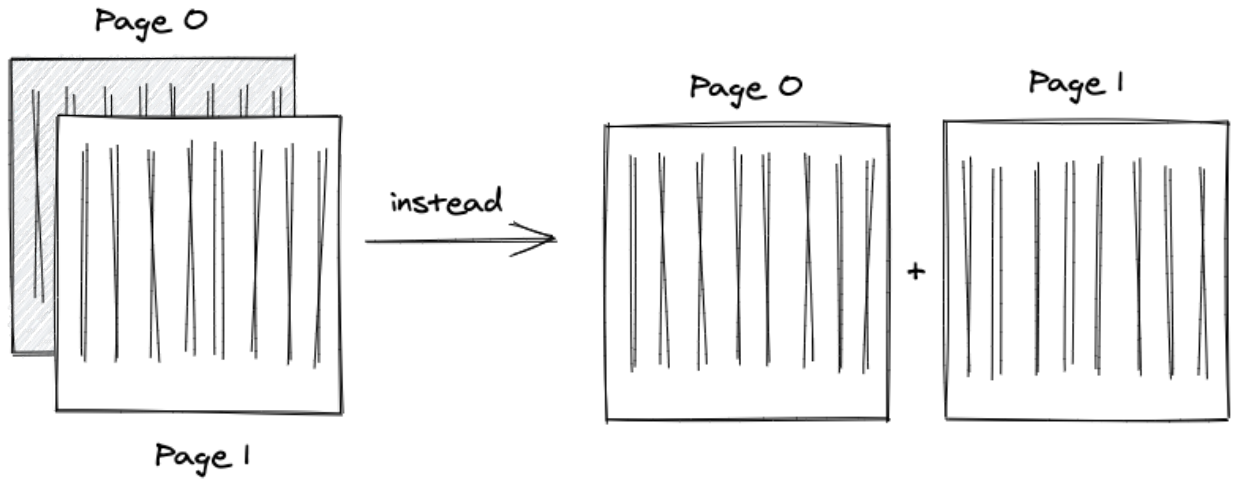
```
const App = () => {
  ...

  const [urls, setUrls] = React.useState([getUrl(searchTerm, 0)]);

  ...

  const handleSearchSubmit = event => {
    handleSearch(searchTerm, 0);

    event.preventDefault();
  };

  const handleLastSearch = searchTerm => {
    setSearchTerm(searchTerm);

    handleSearch(searchTerm, 0);
  };

  const handleSearch = (searchTerm, page) => {
    const url = getUrl(searchTerm, page);
    setUrls(urls.concat(url));
  };

  ...
};
```

为了在按钮被点击时获取下一页，我们需要在这个新的处理程序中增加 `page` 参数：

src/App.js

```
const App = () => {
  ...

  const handleMore = () => {
    const lastUrl = urls[urls.length - 1];
    const searchTerm = extractSearchTerm(lastUrl);
    handleSearch(searchTerm, stories.page + 1);
  };

  ...

  return (
    <div>
      ...

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}

      <button type="button" onClick={handleMore}>
        More
      </button>
    </div>
  );
};
```

我们已经用动态 page 参数实现了数据获取。初次和最后一次搜索总是使用第一页，而每次使用新的” 更多” 按钮获取数据都会使用递增的页面。不过在尝试这个功能的时候，有一个关键的 bug：新的取数并没有扩展之前的列表，而是完全取代了它。



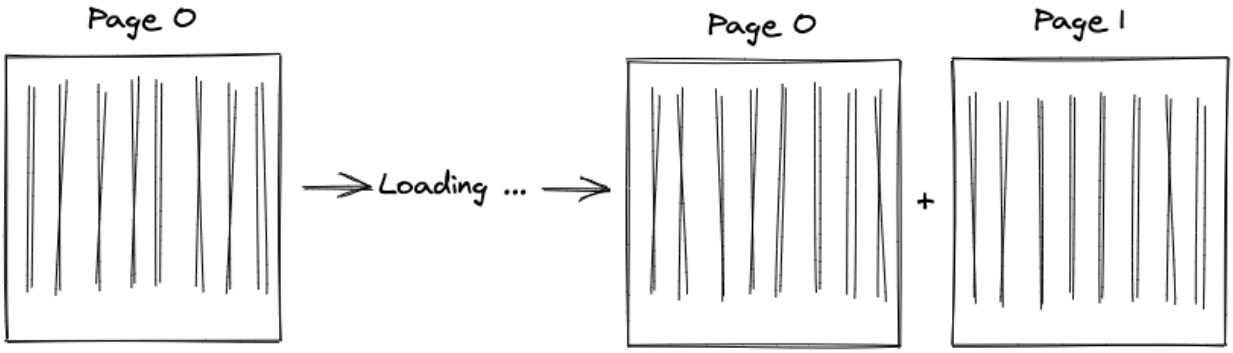
我们在 reducer 中解决这个问题，避免用新的 `data` 替换当前的 `data`，对分页列表进行连接：

src/App.js

```
const storiesReducer = (state, action) => {
  switch (action.type) {
    case 'STORIES_FETCH_INIT':
      ...
    case 'STORIES_FETCH_SUCCESS':
      return {
        ...state,
        isLoading: false,
        isError: false,
        data:
          action.payload.page === 0
            ? action.payload.list
            : state.data.concat(action.payload.list),
        page: action.payload.page,
      };
    case 'STORIES_FETCH_FAILURE':
      ...
    case 'REMOVE_STORY':
      ...
    default:
      throw new Error();
  }
};
```

用新按钮获取更多数据后，显示的列表会增长。但是仍有闪烁的现象，使用户体验没那么

好。在获取分页数据时，由于加载指示器出现，列表会消失片刻，并在请求解决后重新出现。



所需的行为是渲染列表。在开始时是一个空列表，并将" 更多" 按钮替换为仅针对待处理请求的加载指示器。当任务从单个列表发展到分页列表时，这是一个常见的 UI 重构，用于有条件的渲染。

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      ...

      <List list={stories.data} onRemoveItem={handleRemoveStory} />

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <button type="button" onClick={handleMore}>
          More
        </button>
      )}
    </div>
  );
};
```

现在可以持续获取热门故事的数据了。当使用第三方 API 时，探索它的边界总是一个好主意。每个远程 API 都会返回不同的数据结构，所以它的功能可能会有所不同，并且可以在消费 API 的应用程序中使用。

练习

- 检查 上一节的源码[220].
- 确认 上一节之后的变更[221].
- 重新访问 Hacker News API 文档[222]：是否有办法通过向 API 端添加更多参数来获取更多的页面列表项目？
- 重温本节开头讲到的分页和无限分页。你将如何实现一个普通的分页组件，按钮从 1-[3]-10，每个按钮只取并显示一页列表。
- 与其说有一个" 更多" 按钮，不如说用无限滚动技术实现无限分页？无限滚动不是明确地点击一个按钮来获取下一页，而是在浏览器的视口到达显示列表的底部后，就可以获取下一页。

---

[220] https://codesandbox.io/s/github/the-road-to-learn-react/hacker-stories/tree/hs/Paginated-Fetch

[221] https://github.com/the-road-to-learn-react/hacker-stories/compare/hs/Remember-Last-Searches...hs/Paginated-Fetch?expand=1

[222] https://hn.algolia.com/api

部署 React 应用

现在是时候让你的 React 应用走向世界了！有许多方法可以将 React 应用部署到生产环境中，也有许多提供这种服务的平台。在这里我们就简单的把它集中到一个平台上，之后你可以自己去看一看其他的托管平台。

构建过程

到目前为止，我们做的所有事情都是在应用的开发阶段，这时服务器会处理所有的事情：将所有文件打包成一个应用，并且运行在本地机器的 localhost 上。结果是，我们的代码并不能给别人使用。

下一步是让你的应用进入生产阶段，将其托管到远程服务器上，这个步骤称为部署，这样可以让其他用户访问到你的应用。在一个程序可以向用户开放之前，它需要被打包成一个基本的应用程序，冗余的代码、测试代码和重复的代码会被删掉。在工作中还有一个叫做 minification 的过程，这个过程会再次减小代码的体积。

幸运的是，在 create-react-app 的构建工具中自带了优化和打包的功能。首先在命令行中构建你的应用程序：

Command Line

```
npm run build
```

这一步将在你的项目中创建一个新的 *build/* 文件夹，其中包含了打包的应用程序。你现在可以把这个文件夹部署在一个托管平台上，但我们在进行实际操作之前将使用一个本地服务器来模拟这个过程。首先，在你的机器上安装一个 HTTP 服务器：

Command Line

```
npm install -g http-server
```

接下来，用这个本地 HTTP 服务器来管理你的应用程序：

Command Line

```
http-server build/
```

这个过程也可以按需进行，只需要一个命令就可以完成。

Command Line

```
npx http-server build/
```

输入其中一个命令后，会出现一个 URL，通过这个 URL 可以让您可以访问打包优化后的应用。它是通过一个本地 IP 地址发送的，可以通过您的本地网络提供，这意味着我们在本地机器上托管了应用程序。

部署到 Firebase

在我们用 React 构建了一个完整的应用之后，最后一步就是部署。这是一个让你的想法走向世界的转折点，从学习如何编码到制作应用。我们将使用 Firebase Hosting 进行部署。

Firebase 适用于 create-react-app，以及 Angular 和 Vue 等大多数库和框架。首先，将 Firebase CLI 安装到全局依赖中：

Command Line

```
npm install -g firebase-tools
```

使用全局安装的 Firebase CLI 可以让我们在部署应用程序时，不必担心项目依赖性。对于任何一个全局安装的节点包，记得尽可能频繁地使用相同的命令将其更新到更新的版本：

Command Line

```
npm install -g firebase-tools
```

如果你还没有 Firebase 项目，请注册一个 Firebase 账户[223]，并在那里创建一个新项目。然后你就可以将 Firebase CLI 与 Firebase 账户（Google 账户）关联起来：

Command Line

```
firebase login
```

命令行中会显示一个 URL，你可以在浏览器中打开，或者使用 Firebase CLI 打开它。选择一个 Google 账户来创建 Firebase 项目，并给予 Google 必要的权限。返回命令行，验证登录成功。

接下来，移动到项目的文件夹中，并执行下面的命令，为 Firebase 托管功能初始化一个 Firebase 项目：

Command Line

```
firebase init
```

接下来，选择主机选项。如果你对使用 Firebase Hosting 相关的其他工具感兴趣，可以添加其他选项：

[223] https://console.firebase.google.com/

Command Line

```
? Which Firebase CLI features do you want to set up for this folder? Press Space to \
select features, then Enter to confirm your choices.
 ? Database: Deploy Firebase Realtime Database Rules
 ? Firestore: Deploy rules and create indexes for Firestore
 ? Functions: Configure and deploy Cloud Functions
-> Hosting: Configure and deploy Firebase Hosting sites
 ? Storage: Deploy Cloud Storage security rules
```

Google 在登录后会意识到所有与账户相关联的 Firebase 项目，我们可以从 Firebase 平台上选择一个项目：

Command Line

```
First, let's associate this project directory with a Firebase project.
You can create multiple project aliases by running firebase use --add,
but for now we'll just set up a default project.

? Select a default Firebase project for this directory:
-> my-react-project-abc123 (my-react-project)
i  Using project my-react-project-abc123 (my-react-project)
```

还有一些其他的配置步骤需要定义。我们不使用默认的 *public/* 文件夹，而是要使用 create-react-app 的 *build/* 文件夹。如果你自己用 Webpack 等工具进行打包，你可以为 build 文件夹选择合适的名字：

Command Line

```
? What do you want to use as your public directory? build
? Configure as a single-page app (rewrite all urls to /index.html)? Yes
? File public/index.html already exists. Overwrite? No
```

在我们第一次执行 `npm run build` 后，create-react-app 应用程序会创建一个 *build/* 文件夹，这个文件夹包含了所有来自 *public/* 文件夹和 *src/* 文件夹的合并内容。由于这是一个单页应用，我们希望将用户重定向到 *index.html* 文件，这样 React 路由器就可以处理客户端的路由：

现在你的 Firebase 初始化已经完成了。这一步在你的项目文件夹中为 Firebase Hosting 创建了一些配置文件。你可以在 Firebase 的文档[224] 中阅读更多关于它们的内容，以配置重定向、404 页面或头文件。最后，在命令行中用 Firebase 部署你的 React 应用：

[224] https://firebase.google.com/docs/hosting/full-config

Command Line

```
firebase deploy
```

在成功部署后，你应该会看到一个类似的输出与你的项目标识符：

Command Line

```
Project Console: https://console.firebase.google.com/project/my-react-project-abc123\
/overview
Hosting URL: https://my-react-project-abc123.firebaseapp.com
```

访问这两个页面来观察结果，第一个链接导航到你的 Firebase 项目的仪表板，在那里你会看到一个新的 Firebase 托管的面板，第二个链接导航到你部署的 React 应用。

如果你看到你部署的 React 应用有一个空白页，请确保 *firebase.json* 中的 `public` 键/值对设置为 `build`，或者你为这个文件夹选择的任何名称。其次，确认你已经用 `npm run build` 为你的 React 应用运行了构建脚本。最后，查看官方将 create-react-app 应用部署到 Firebase 的故障排除区域[225]。用 `firebase deploy` 再尝试一次部署。

练习

- 了解更多 Firebase Hosting[226].
- 将您的域名关联到您的 Firebase 部署的应用程序[227].
- 可选：如果你想拥有一个托管的云服务器，请查看 DigitalOcean[228] 。这会消耗更多的精力，但它允许更多的控制，我把我所有的网站、网络应用和后端 API 都托管在那里[229] 。

[225] https://create-react-app.dev/docs/deployment
[226] https://firebase.google.com/docs/hosting/
[227] https://firebase.google.com/docs/hosting/custom-domain
[228] https://m.do.co/c/fb27c90322f3
[229] https://www.robinwieruch.de/deploy-applications-digital-ocean/

大纲

我们的 React 之路已经走到了尽头，希望你喜欢这本书，并希望你能从中获得一些帮助。如果你喜欢这本书，请与有兴趣了解更多 React 的朋友分享。此外，在Amazon[230] 或 Goodreads[231]上发表评论，将有助于我根据你的反馈提供创作的内容。

从这里开始，我建议你在参与另一本书、课程或教程之前，先扩展应用来创建自己的 React 项目。试用一周，通过部署将其带入生产，并联系我或其他人来展示它。我总是有兴趣看到我的读者创造的东西，并了解我如何帮助他们一起。

如果你正在为你的应用寻找扩展，我推荐你在掌握基础知识后的几种学习途径：

- 连接数据库或鉴权：不断壮大的 React 应用最终会需要持久性的数据。这些数据应该存储在数据库中，这样可以在浏览器会话后保持数据的完整性，以便与不同的用户共享。Firebase 是引入数据库的最简单的方法之一，而无需编写后台应用程序。在我那本名为“Firebase 学习之道”[232]的书中，你会发现一个关于如何在 React 中使用 Firebase 认证和数据库的分步指南。
- 连接后端： React 处理前端应用程序，到目前为止，我们只从第三方后端的 API 请求数据。你也可以用连接到数据库并管理认证/授权的后端应用程序引入 API。在“GraphQL 学习之道”[233]中，我教你如何使用 GraphQL 进行客户端-服务器通信。你将学习到如何将你的后端连接到数据库，如何管理用户会话，以及如何通过 GraphQL API 从前端到后端应用程序的对话。
- 状态管理：你在这次学习经历中专门使用 React 来管理本地组件状态。对于大多数应用来说，这是一个好的开始，但 React 也有外部状态管理解决方案。我在“Redux 学习之道”[234]一书中探讨了最流行的一种。
- 使用 **Webpack** 和 **Babel** 工具：我们在本书中使用 *create-react-app* 来设置应用程序。在某些时候，你可能会想学习它的工具，在没有 *create-react-app* 的情况下创建项目。我推荐使用 Webpack[235] 进行最低限度的设置，之后你可以应用更多的工具。
- 代码组织：回想一下关于代码组织的章节，并应用这些变化，如果你还没有。这将有助于将你的组件组织成结构化的文件和文件夹，它将帮助你理解代码拆分、可重用性、可维护性和模块 API 设计的原则。你的应用程序最终会成长并需要结构化的模块，所以最好现在就开始。
- 测试: 我们只对测试进行了简单的介绍。如果你对测试 Web 应用程序不熟悉，深入了解单元测试和集成测试[236]，特别是 React 应用程序。Cypress[237] 是 React 中端到端测试的一个有用的探索工具。

[230] https://amzn.to/2JH1P42
[231] https://www.goodreads.com/book/show/37503118-the-road-to-learn-react
[232] https://www.roadtofirebase.com/
[233] https://www.roadtographql.com/
[234] https://www.roadtoredux.com/
[235] https://www.robinwieruch.de/minimal-react-webpack-babel-setup/
[236] https://www.robinwieruch.de/react-testing-tutorial
[237] https://www.robinwieruch.de/react-testing-cypress

- 类型检查：前面我们在 React 中使用了 TypeScript，这是一个好的防止 bug 和提高开发者体验的实践方式。深入研究这个话题，让你的 JavaScript 应用更加强大。也许你最终会一直使用 TypeScript 而不是 JavaScript。
- **UI** 组件：很多初学者在项目中过早引入 Bootstrap 等 UI 组件库。在 React 中使用标准 HTML 元素的下拉、复选框或对话框是比较实用的。这些组件大多会管理自己的本地状态。复选框必须知道它是选中还是不选中，所以你应该把它们作为受控组件来实现。在你涵盖了这些关键的 UI 组件的基本实现之后，引入 UI 组件库应该会更容易。
- 路由：你可以用 react-router[238] 为你的应用程序添加路由。在我们创建的应用程序中，只有一个页面，但这个页面会增长。React Router 可以帮助管理跨多个 URL 的多个页面。当你将路由引入到你的应用程序中时，不会向 Web 服务器发出下一个页面的请求。路由器会在客户端处理这个问题。
- **React Native:** React Native[239] 将你的应用带到 iOS 和 Android 等移动设备上。一旦你掌握了 React，React Native 的学习曲线应该不会那么陡峭，因为它们有着相同的原理。与移动设备的唯一区别是布局组件、构建工具和你的移动设备的 API。

我邀请你访问我的网站[240]，寻找更多关于网络开发和软件工程的有趣话题。你也可以订阅我的电报[241]或推特[242]来获取最新的文章、书籍和课程。

感谢阅读 《*React* 学习之道》。

Regards,

Robin Wieruch

[238] https://github.com/ReactTraining/react-router
[239] https://facebook.github.io/react-native/
[240] https://www.robinwieruch.de
[241] https://www.getrevue.co/profile/rwieruch
[242] https://twitter.com/rwieruch