by robin wieruch

the Road to learn 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 and JimmyLv

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

此版本發布於 2018-02-27

在 Tweet 上分享此書！

請在 Twitter 上面替作者 Robin Wieruch and JimmyLv 宣傳!

對此書所建議的 tweet 是:

I just bought The Road to learn React by @rwieruch #ReactJs
https://roadtoreact.com/course-details?courseId=THE_ROAD_TO_LEARN_REACT

Contents

前言

《React 学习之道》会教您一些 React 的基础知识。通过这套教程，您可以用纯 React 构建一个真正可用的应用程序，而不需要去理会其他复杂的工具。我将为您逐一介绍从开发环境的准备到部署上线的全部过程。本书每一章都包含一些额外的索引资料以及课后练习。在读完本书之后，您将会有能力依靠自己构建一个 React 应用。我，Robin Wieruch，以及整个社区会持续维护和更新这些资料。

通过《React 学习之道》，在开始陷入到更庞大的 React 生态圈之前，我想为您奠定一个良好的基础。它会通过一个真实可用的 React 应用来解释基本概念、设计模式以及最佳实践。

您将会学习构建您自己的 React 应用。这个应用会涉及一些真正可用的功能，比如分页，客户端缓存，以及像搜索和排序这样的交互功能。另外在这个过程中，您会慢慢从 JavaScript ES5 过渡到 JavaScript ES6。我希望这本书能充分体现我对 React 和 JavaScript 的热忱所在，并帮助您能够开始您的开发旅程。

读者赠言

Muhammad Kashif[1]：“《React 学习之道》是一本独一无二的书，我推荐给任何想要学习 React 基础和进阶技巧的学生或者专业人士。她包含了诸多启发性的小提示和绝无仅有的技术点。书中虽然引用了大量例子和参考资料，但最后都被用到我们要解决的问题上，这体现了编写本书令人惊叹地缜密。我有17年的互联网和桌面开发经验，阅读本书之前，我在学习 React 的过程中缺并不顺利。而这本书就像魔术一样有用。”

Andre Vargas[2]：“Robin Wieruch 的《React 学习之道》是一本非常牛的书！我所学到的绝大部分有关 React 甚至是 ES6的知识都是通过她得来的！”

Nicholas Hunt-Walker, Instructor of Python at a Seattle Coding School[3]：“这是一本我读过的最严谨和最实用的编程书籍之一。一本完整的 React 和 ES6使用说明。”

Austin Green[4]：“非常感谢，真的很喜欢这本书。完美的学习曲线，不管是 React，ES6，还是抽象编程概念。”

Nicole Ferguson[5]：“这个周末跟着 Robin 的课程学习 React，我发现这一切太有意思了。这几乎让我感到羞愧。”

Karan[6]：“刚刚完成这个课程。这是全世界最好的学习 React 和 JS 的一本书。完美展现了 ES 的优雅。膜拜！:)”

Eric Priou[7]：“Robin 的《React 学习之道》是必读的一本书。简明扼要地介绍了 React 和 Javascript。”

一个新手开发：“作为一个开发新手，我刚刚完成了这本书的学习，非常感谢写了这本书。她非常容易上手，我相信自己在接下来的几天可以开始从头开发一个新应用。这本书比我之前试过的官方 React 入门文档好很多（由于缺乏细节，我并未能够完成）。每个章节后面的练习题对我有很好的激励效果。”

一个学生：“这是最好的学习 React 的一本书。我们可以一边做练习项目，一边学习知识点，然后还能紧扣我们的学习主题。我发现「边码边学」是最好的掌握编程的方法，而这本书完完全全是这样教我的。”

Thomas Lockney[8]：“这是一本非常扎实的介绍 React 的书，而不是试着把事情搞复杂。我本来只想尝试理解看看这本书到底讲了什么，然后我得出了上面的结论。我并没有跟着所有

[1] https://twitter.com/appsdevpk/status/848625244956901376
[2] https://twitter.com/andrevar66/status/853789166987038720
[3] https://twitter.com/nhuntwalker/status/845730837823840256
[4] https://twitter.com/AustinGreen/status/845321540627521536
[5] https://twitter.com/nicoleffe/status/833488391148822528
[6] https://twitter.com/kvssl992/status/889197346344493056
[7] https://twitter.com/erixtekila/status/840875459730657283
[8] https://www.goodreads.com/review/show/1880673388

的脚注来学习我还没有注意到的新的ES6语法（我当然不会说我一直没有注意到，Bob）（译者注：这个是在博客中与另外一个朋友互动的话）。对于那些没有及时了解到这些新功能，并且很勤奋的跟着练习的朋友们，我想很肯定地对你们说，你们能学到的会不仅仅是这本书所教的东西。”

儿童教育

这本书应该能让任何人都可以学习 React。但是，由于这本书一开始是用英文写的，并不是所有人都有能力使用这些资源。所以我想用这个项目来支持那些发展中国家教孩子们英语的项目。

- 2017/4/11 - 2017/04/18，学习 React 的回馈[9]

[9] https://www.robinwieruch.de/giving-back-by-learning-react/

问题解答

我怎么获取最新的版本？ 你可以订阅[10]邮件通知或者在 Twitter[11] 上关注我来获取更新。一旦你拿到这本书的副本, 她会自己保持更新。只是如果有更新，必须下载新的副本。

这本书用的是最新版本的 **React** 吗？ 这本书总是会随着 React 版本更新而更新。通常情况下书籍在发行后很快就过时了。因为这本书是自出版的，所以我可以随时更新它。

这本书包括 **Redux** 的内容吗？ 不包括。不过我写了第二本书。《React 学习之道》会给你奠定一个坚实的基础，这样你可以继续之后的高阶内容。同样一个应用，在本书中，示例应用程序的实现将会向你证明，不需要 Redux 也可以搭建一个完整的 React 应用程序。在读完这本书之后，你应该有能力自己搭建一个不用 Redux 的应用。然后你可以读我的第二本书来学习Redux[12]。

这本书会用到 **JavaScript ES6**吗？ 是的。但是别担心。如果你熟悉 ES5就可以了。所有的 JavaScript ES6语法，在我们学习 React 的过程中，都逐步会从 ES5转换成 ES6。每个语法都会有详细的解释。这本书不只是用来学习 React，同时也包含所有相关的 ES6语法。

以后你会添加新的章节吗？ 你可以从更新日志章节找到已经发生的重大更新。当然也会有一些较小的改动不会被公示出来。总体来说，这取决于社区是否让我继续写这本书。如果大家很喜欢这本书，我会继续写更多的章节以及改进已有的内容。我会保持书中的内容包含最新的最佳实践，概念和设计模式。

我在读这本书的时候遇到困难，怎么获取帮助？ 这本书有一个 Slack 聊天组[13]给读者。你可以加入这个频道获取帮助，或者帮助其他人。不管怎样，帮助其他人也有助于你自己的学习。

如果我在书中发现一些问题，有地方可以帮助解决吗？ 如果你发现问题，请加入 Slack 聊天组。另外，你可以看看本书Github 上的问题[14]。有可能你的问题已经有人遇到了，并且解决方案也在上面。如果找不到同样的问题，请新建一个新的问题来解释你遇到的麻烦，比如提供一个截图，和一些细节信息（比如页数、Nodejs 的版本等）。之后，我会修复所有问题，然后发布一个新的版本。

我能提供帮助来改进这本书吗？ 是的。你可以在在 GitHub 上贡献[15]直接提出你的想法和代码。我并不宣称自己是一个专家，也不是用英语母语进行写作。我会非常感谢您的帮助。

我可以为这个项目做出其他支持吗？ 是的。欢迎联系我。我贡献了非常多的个人时间到开源教程和学习资源的制作。你可以看看我的介绍[16]。我非常高兴能得到您在Patreon[17]的赞

[10] https://www.getrevue.co/profile/rwieruch

[11] https://twitter.com/rwieruch

[12] https://roadtoreact.com/course-details?courseId=TAMING_THE_STATE

[13] https://slack-the-road-to-learn-react.wieruch.com/

[14] https://github.com/rwieruch/the-road-to-learn-react/issues

[15] https://github.com/rwieruch/the-road-to-learn-react

[16] https://www.robinwieruch.de/about/

[17] https://www.patreon.com/rwieruch

助。

写这本书的有什么特别的出发点吗？ 是的。我想让你花点时间想一想谁比较适合学习 React。这个人可能已经表现出一些兴趣，可能已经学习过一段时间，或者有可能并没有发现自己其实可以学习 React。你可以找到他们然后把这本书分享给他。这对我真的非常重要，这本书就旨与他人分享。

更新日志

10. January 2017:

- v2 Pull Request[18]
- even more beginner friendly
- 37% more content
- 30% improved content
- 13 improved and new chapters
- 140 pages of learning material
- + interactive course of the book on educative.io[19]

08. March 2017:

- v3 Pull Request[20]
- 20% more content
- 25% improved content
- 9 new chapters
- 170 pages of learning material

15. April 2017:

- upgrade to React 15.5

5. July 2017:

- upgrade to node 8.1.3
- upgrade to npm 5.0.4
- upgrade to create-react-app 1.3.3

17. October 2017:

- upgrade to node 8.3.0

[18] https://github.com/rwieruch/the-road-to-learn-react/pull/18
[19] https://www.educative.io/collection/5740745361195008/5676830073815040
[20] https://github.com/rwieruch/the-road-to-learn-react/pull/34

- upgrade to npm 5.5.1
- upgrade to create-react-app 1.4.1
- upgrade to React 16
- v4 Pull Request[21]
- 15% more content
- 15% improved content
- 3 new chapters (Bindings, Event Handlers, Error Handling)
- 190+ pages of learning material
- +9 Source Code Projects[22]

[21] https://github.com/rwieruch/the-road-to-learn-react/pull/72
[22] https://roadtoreact.com/course-details?courseId=THE_ROAD_TO_LEARN_REACT

怎么读这本书？

我希望这本书在你打算写一个应用的时候，能够教会你 React。这是一个实践性很强的 React 学习指南，而不是一个 React 资料索引。你会在这本书的指引下写一个 Hacker News（译者注：一个著名黑客论坛）应用，会需要和真实 API 交互。她包括了很多有趣的话题，包括 React 状态管理，缓存和交互效果（排序和搜索）。在这个过程中你可以学到 React 的最佳实践和设计模式。

另外，本书可以让你平滑地从 Javascript ES5过渡到 JavaScript ES6。React 采用了非常多的 JavaScript ES6语法，我希望告诉你怎么使用它们。

一般来说，本书每一章都会在前一章基础上继续搭建。每一章都会教你一些新的知识点。务必不要跳过任何内容。你应该实践和领会每一个步骤。你可以自己写出实现，然后阅读书中的内容。每一章之后，我提供了一些阅读资料和练习题。如果你真的想学好 React，我非常推荐读完这些阅读资料并上手完成练习题。完成一章的学习之后，请保证自己对所学内容熟练掌握之后，再继续后面的章节。

最终，你会拥有一个完整的可以部署上线的应用。我非常渴望能看到你的杰作，如果你完成了这本书，请给我发消息。最后一章会给你很多选择来继续学习和应用 React。总之你会发现在我的个人网站[23]有非常多的 React 的相关话题。

既然你在看这本书，我猜你还是个 React 新手。那真是太好了。最后我希望你能给我你的反馈，来帮助我完善学习资料，以帮助人人都可以掌握 React。你可以直接提交代码到GitHub[24]，或者在Twitter[25]给我发消息。

[23] https://www.robinwieruch.de/

[24] https://github.com/rwieruch/the-road-to-learn-react

[25] https://twitter.com/rwieruch

你可以期望学到什么（目前为止...）

- Hacker News 的 React 版本[26]
- 没有复杂的配置
- 用 create-react-app 来初始化你的应用
- 高效而轻量级的代码
- 只用 React setState 来做状态管理（目前为止...）
- 从 JavaScript ES5一路平滑过渡到 ES6
- React setState 和生命周期函数的用法
- 和真实 API 的交互（Hacker News）
- 高级用户交互
  - 客户端排序
  - 客户端过滤
  - 服务器端搜索
- 客户端缓存的实现
- 高阶函数和高阶组件
- 用 Jest 进行组件的切片 (snapshot) 测试
- 用 Enzyme 进行组件的单元测试
- 过程中学到一些有用的工具库
- 过程中的练习题和扩展阅读
- 认同和巩固你的所学
- 将你的应用部署到产品环境

[26] https://intense-refuge-78753.herokuapp.com/

React 简介

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

你好，我叫 React。

为什么你应该学习 **React** 近年来，单页面应用[27]（SPA，single page application）变得越来越流行。像 Angular、Ember 以及 Backbone 这些框架，帮助 JavaScript 开发者构建了超越纯 JavaScript（vanilla JavaScript）和 jQuery 的现代 Web 应用。这个流行的解决方案清单并不够详尽，现在仍然有大量的 SPA 框架。如果你去关注他们的发布日期的话，大部分都属于第一代 SPA：Angular 发布于2010年，Backbone 发布于2010年，以及 Ember，发布于2011年。

Facebook 在2013年首次发布了 React。React 并不是一个 SPA 框架，而是一个视图库。也就是 MVC[28]（Model View Controller，模型-视图-控制器）里的 V。它的功能仅仅是把组件渲染成浏览器中的可见元素。但是，围绕 React 周边的整个生态系统让构建单页面应用成为可能。

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

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

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

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

第一代 SPA 框架更贴近企业级。它们缺乏足够的灵活性。React 时刻保持创新并且被 Airbnb、Netflix 和 Facebook[29] 这些技术界的领导企业所采用。这些企业都对 React 的未来，以及 React 生态圈给予了充分的资助。

React 可能是构建现代 Web 应用最佳选择之一。虽然它仅仅提供了视图层抽象，但是 React 生态圈组成了一个整体的灵活且可替换的框架[30]。React 拥有简单整洁的 API、神奇的生态圈以及很棒的社区。你可以看一下我使用 React 的经验：我为什么从 Angular 转移到了 React[31]。我强烈推荐你仔细考虑一下，选择 React 而不是其他的框架或库。毕竟每个人都迫切地想要知道 React 接下来会引领我们走向何方。

练习

- 阅读《我为什么从 Angular 转移到了 React》[32]

[27] https://en.wikipedia.org/wiki/Single-page_application
[28] https://www.wikiwand.com/zh/MVC
[29] https://github.com/facebook/react/wiki/Sites-Using-React
[30] https://www.robinwieruch.de/essential-react-libraries-framework/
[31] https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/
[32] https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/

- 阅读《React 灵活的生态圈》[33]

[33] https://www.robinwieruch.de/essential-react-libraries-framework/

基本要求

如果你之前就有使用其他 SPA 框架或者库的经验，你应该已经非常熟悉 Web 开发的基本内容了。如果你刚刚开始学习 Web 开发，在学习 React 之前，你应该要了解一下 HTML、CSS 和 JavaScript ES5。本书会可以平滑地过度到 JavaScript ES6 及其后的版本。我建议你加入本书官方的 Slack 群组[34]，在这里你可以找到找到或者提供给他人必要的帮助。

编辑器和终端

那么开发环境呢？你需要一个可用的编辑器或者 IDE，以及一个终端（命令行工具）。你可以参考我编写的环境搭建指南[35]。这篇指南主要针对的是 macOS 用户，但是你可以在其他操作系统上找到类似的替代方案。同样你也可以在网上找到大量关于如何设置 Web 开发环境的更详细的文章。

你也可以选择使用 git 和 GitHub 把来保存与本书相关的个人项目以及学习进度。这篇简短的指南[36]讲述了它们的使用方法。但是再次提醒一下，你并不一定非要这样做，因为如果所有东西都从头学起可能会给你带来更多压力。所以如果你只是一个新手，并且希望专注于本书的基础内容的话，你可以跳过这一步。

Node 和 NPM

环境设置的最后一步是安装 node 和 npm[37]。这两个工具都是用来管理你在本书中所用到的各种库的。本书中提及的 node 包需要通过 npm （node package manager，Node 包管理器）来安装，这些包可能是一些库，或者集成在一起的框架。

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

Command Line

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

[34] https://slack-the-road-to-learn-react.wieruch.com/
[35] https://www.robinwieruch.de/developer-setup/
[36] https://www.robinwieruch.de/git-essential-commands/
[37] https://nodejs.org/en/

node 和 npm

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

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

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

Command Line

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

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

Command Line

```
npm install <package>
```

以 React 为例，应该写成：

Command Line

```
npm install react
```

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

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

Command Line

```
npm init -y
```

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

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

另外还有一个需要提及的 npm 命令：

Command Line

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

`--save-dev` 标记表示该 node 包只是用作开发环境的一部分，并不会被作为你产品代码的一部分发布。哪种 node 包适用这个场景呢？设想你需要一些 node 包辅助测试你的应用，然后需要通过 npm 来安装这些包，但是不希望他们混入产品代码里面。测试过程应该只会发生在开发阶段，而不是在线上部署运行的时候。因为那个时候已经用不到测试代码了，你的应用应该已经被测试完而且可以被你的用户使用了。这可能是你唯一的使用 `--save-dev` 的场景。

你可能会遇到更多的 npm 命令，但目前来说这些已经足够了。

练习

- 搭建一个简易的 npm 项目
  - 使用 `mkdir <文件夹名>` 创建一个新的文件夹
  - 通过 `cd <文件夹名>` 进入该文件夹
  - 运行 `npm init -y` 或者 `npm init` 来初始化一个 npm 项目
  - 安装一个本地包，比如 React：`npm install react`
  - 仔细看一下 *package.json* 文件和 *node_modules/* 文件夹里面的内容
  - 找找看有没有什么办法把 *react* 从项目中卸载掉
- 阅读更多关于 npm[38] 的内容。

[38] https://docs.npmjs.com/

安装 React

有很多种方式可以让你创建一个 React 应用。

第一种方式是通过 CDN。听起来可能会有点复杂。CDN 指的是内容分发网络[39]（content delivery network）。一些公司会把他们公开的文件放置在 CDN 上供人们访问。其中可以是像 React 这样的库，毕竟整个打包的 React 库就只有一个 *react.js* 文件，可以直接托管在任何地方然后引入到你的应用中。

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

Code Playground

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></\
script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.developmen\
t.js"></script>
```

但是如果你有 npm 来安装 React 的话为什么还需要 CDN 呢？

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

Command Line

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

这种方式通常适用于那些使用 npm 做包管理的项目。

不过这还不够。你还可能需要设置 Babel[40] 来让你的项目支持 JSX（React 的专用语法）和 JavaScript ES6。Babel 把你的 JavaScript ES6 和 JSX 代码转译（transpile）成浏览器可以支持的版本，毕竟并不是所有的浏览器都支持这些高级语法。这一步设置包含一堆的配置和工具，对于一个新手来说可能会感觉到不小的压力。

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

练习：

- 阅读更多安装 React[41] 的内容。

[39] https://en.wikipedia.org/wiki/Content_delivery_network
[40] http://babeljs.io/
[41] https://facebook.github.io/react/docs/installation.html

零配置搭建 React 应用

在本书中，你将使用 create-react-app[42] 来创建应用。在得到广泛支持的情况下，Facebook 在2016年创建了这样一个零配置的 React 初始化套件。96% 的人向初学者推荐了它[43]。使用 *create-react-app*，各种工具和配置都会在后台集成，而开发人员只需要专注于实现就好。

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

Command Line

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

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

Command Line

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

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

Command Line

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

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

Folder Structure

```
hackernews/
  README.md
  node_modules/
  package.json
  .gitignore
  public/
    favicon.ico
    index.html
    manifest.json
```

[42] https://github.com/facebookincubator/create-react-app

[43] https://twitter.com/dan_abramov/status/806985854099062785

```
src/
  App.css
  App.js
  App.test.js
  index.css
  index.js
  logo.svg
  registerServiceWorker.js
```

简单划分一下这些文件和文件夹，目前你并不需要做一个很全面的了解：

- **README.md:** 后缀名为 .md 表示这是一个 markdown 文件。Markdown 是一个用纯文本创建格式化文档的标记语言。很多源代码项目包含一个 *README.md* 文件，其中包含了这个项目的一些基本的指令和介绍。当你把项目发布到一些平台后，比如 GitHub，当在这个平台访问该项目的时候就会直接看到 *README.md* 里的内容。因为你使用的是 *create-react-app*，所以你的 *README.md* 文件会跟 create-react-app 官方 GitHub [44]仓库的内容一样。
- **node_modules/:** 这个文件夹包含了所有通过 npm 安装的 node 包。在你使用了 *create-react-app* 之后，就有一堆 node 包已经被安装了。通常你不需要特别去关心这个文件夹里面的内容，只需要在命令行用 npm 安装或者卸载 node 包就可以。
- **package.json:** 这个文件包含了 node 包依赖列表和一些其他的项目配置。
- **.gitignore:** 这个文件包含了所有不应该添加到 git 仓库（repository）中的文件和文件夹。他们应该只能存活在你本地项目文件夹中。一个典型的例子是 *node_modules/*，把 *package.json* 共享给你的伙伴们就足够他们获取和安装所有的依赖了，没必要把整个依赖打包共享给他们。
- **public/:** 这个文件夹包含了所有你的项目构建出的产品文件。最终所有你写在 *src/* 文件夹里面的代码都会在项目构建的时候被打包放在 public 文件夹下。
- **manifest.json** 和 **registerServiceWorker.js:** 在这个阶段不用担心这些文件用来干什么，我们不会在这个项目中用到他们。

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

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

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

[44] https://github.com/facebookincubator/create-react-app

Command Line

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

// 运行所有测试
npm test

// 构建项目的产品文件
npm run build
```

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

练习：

- 通过 `npm start` 运行并在浏览器中访问你的应用（你可以通过 `Control + C` 来退出此命令）
- 运行交互式的 `npm test` 脚本
- 运行 `npm run build` 并确认项目中出现了 *build/* 文件夹（你可以在之后把它删除掉；注意 build 文件夹可以用来部署你的应用[45]）
- 熟悉一下项目的文件结构
- 熟悉一下不同文件的具体内容
- 阅读更多关于 npm 和 create-react-app[46] 的内容

[45] https://www.robinwieruch.de/deploy-applications-digital-ocean/

[46] https://github.com/facebookincubator/create-react-app

JSX 简介

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

src/App.js

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

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

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

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

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

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

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

src/App.js

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

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

export default App;
```

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

src/App.js

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

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

export default App;
```

现在再次运行 `npm start`，你就能看到效果了。

另外，你应该注意到了代码中的 `className` 属性。它其实就是标准 HTML 中的 `class` 属性。但是因为一些技术上的原因，JSX 把一些 HTML 的内部属性替换成了不同的。你可以在 React 文档支持的 HTML 属性[47]中找到相关的内容。他们都遵守驼峰命名法（camelCase convention）。在接下来的学习过程中，你将会遇到更多的 JSX 专有的属性。

练习：

- 定义更多的变量然后再 JSX 中引用并渲染（render）它们
  - 使用一个复杂对象来表示一个拥有姓氏（first name）和名字（last name）的用户
  - 把该用户的属性放到 JSX 中渲染
- 阅读更多关于 JSX[48] 的内容
- 阅读更多 React 组件、元素和实例[49]的内容

[47] https://facebook.github.io/react/docs/dom-elements.html

[48] https://facebook.github.io/react/docs/introducing-jsx.html

[49] https://facebook.github.io/react/blog/2015/12/18/react-components-elements-and-instances.html

ES6 const 和 let

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

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

Code Playground

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

被关键字 let 声明的变量可以被改变。

Code Playground

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

当一个变量需要被重新赋值的话，你应该使用 let 去声明它。

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

Code Playground

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

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

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

在你的应用中，你应该用 const 来代替 var。

src/App.js

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

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

export default App;
```

练习：

- 阅读更多关于 ES6 const[50] 的内容
- 阅读更多关于 ES6 let[51] 的内容
- 研究更多关于不可变数据结构的内容
  - 在通常情况下，为什么他们是有意义的
  - 为什么他们会被 React 和它的生态使用

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

ReactDOM

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

src/index.js

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

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

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

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

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

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

Code Playground

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

练习：

- 打开 *public/index.html* 文件，找到 React 应用并放置在你的 HTML 的位置
- 查看更多关于 元素渲染[52] 的内容

[52] https://facebook.github.io/react/docs/rendering-elements.html

模块热替换

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

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

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

src/index.js

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

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

if (module.hot) {
  module.hot.accept();
}
```

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

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

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

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

练习：

- 改变几次你的 *src/App.js* 中的源代码，来观察 HMR 的工作方式
- 观看 Dan Abramov 的视频 Live React: Hot Reloading with Time Travel[53] 的前十分钟

[53] https://www.youtube.com/watch?v=xsSn0QynTHs

JSX 中的复杂 Javascript

让我们回到你的 App 组件中。到目前为止你在你的 JSX 中渲染了一些简单的变量。现在你可以开始渲染一个列表了。这个列表一开始可以是一些示例数据，但是以后你可以从一个外部 API[54] 中获取数据。这会让人更加兴奋。

首先你需要定义一个列表。

src/App.js

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

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

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

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

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

[54] https://www.robinwieruch.de/what-is-an-api-javascript/

src/App.js

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

export default App;
```

在 JSX 中使用 HTML 中的 JavaScript 是很强大的。通常情况下你可以用 `map` 来将一个列表转换成另一个列表。在这个例子中，你使用 `map` 函数将一个列表转换成一组 HTML 元素。到目前为止，每个成员只有 `title` 会被显示。让我们显示一些它们的其他属性。

src/App.js

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

```
export default App;
```

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

React 会帮你完成所有的工作然后逐一显示每个成员。但你应该在 React 中添加一个辅助属性，借此发挥出它的潜能以提高性能。你需要给列表的每一个成员加上一个关键字（key）属性。这样的话 React 就可以在列表发生变化的时候识别其中成员的添加、更改和删除的状态。这个示例数据中已经有一个标识符了。

src/App.js

```
{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>
  );
})}
```

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

src/App.js

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

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

练习：

- 查看更过关于 React 列表和关键字[55] 的内容
- 简要重述 JavaScript 中标准内建数组函数[56]
- 在 JSX 中使用更多的 JavaScript 表达式

[55] https://facebook.github.io/react/docs/lists-and-keys.html

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

ES6 箭头函数

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

Code Playground

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

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

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

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

Code Playground

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

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

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

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

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

src/App.js

```
{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>
  );
})}
```

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

src/App.js

```
{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>
)}
```

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

练习：

- 阅读更多关于 ES6 箭头函数[57] 的内容

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

ES6 类

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

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

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

Code Playground

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

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

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

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

让我们看看如何实例化这个类，以及如何使用它的方法。

Code Playground

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

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

src/App.js

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

...

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

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

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

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

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

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

练习：

- 阅读更多关于 ES6 类[58] 的内容

[58] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes

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

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

ES6

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

现在我们值得休息一下。巩固下这章的内容，你可以试验一下目前为止所编写的代码。你可以在官方代码库[59]中找到源代码。

[59] https://github.com/rwieruch/hackernews-client/tree/4.1

React 基础

本章将指导你了解 React 的基础知识。由于静态组件会有些枯燥，所以这章的内容会包含组件的状态与交互。此外，你将学习使用不同方式声明组件以及如何保持组件的可组合性和可复用性。准备好创造你自己的组件。

组件内部状态

组件内部状态也被称为局部状态，允许你保存、修改和删除存储在组件内部的属性。使用 ES6 类组件可以在构造函数中初始化组件的状态。构造函数只会在组件初始化时调用一次。

让我们引入类构造函数。

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);
  }

  ...

}
```

当你使用 ES6 编写的组件有一个构造函数时，它需要强制地调用 `super();` 方法，因为这个 App 组件是 `Component` 的子类。因此在你的 APP 组件要声明 `extends Component` 。你会在后续内容中更详细地了解使用 ES6 编写的组件。

你也可以调用 `super(props);`，它会在你的构造函数中设置 `this.props` 以供在构造函数中访问它们。否则当在构造函数中访问 `this.props` ， 会得到 `undefined`。稍后你将了解更多关于 React 组件的 props。

现在，在你的示例中，组件中的初始状态应该是一个列表。

src/App.js

```
const list = [
  {
    title: 'React',
    url: 'https://facebook.github.io/react/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  },
  ...
];

class App extends Component {
```

```
  constructor(props) {
    super(props);

    this.state = {
      list: list,
    };
  }

  ...

}
```

state 通过使用 `this` 绑定在类上。因此，你可以在整个组件中访问到 state。例如它可以用在 `render()` 方法中。此前你已经在 `render()` 方法中映射一个在组件外定义静态列表。现在你可以在组件中使用 state 里的 list 了。

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.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>
        )}
      </div>
    );
  }
}
```

现在 list 是组件的一部分。它驻留在组件的 state 中。你可以从 list 中添加、修改或者删除

列表项。每次你修改组件的内部状态，组件的 `render` 方法会再次运行。这样你可以简单地修改组件内部状态，确保组件重新渲染并且展示从内部状态获取到的正确数据。

但是需要注意，不要直接修改 state。你必须使用 `setState()` 方法来修改它。你将在接下来的章节了解到它。

练习：

- 练习使用 state
  - 在构造函数中定义更多的初始化 state
  - 在 `render()` 函数中访问使用 state
- 阅读更多关于 ES6类构造函数[60]

[60] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes#Constructor

ES6 对象初始化

在 ES6 中，你可以通过简写属性更加简洁地初始化对象。想象下面的对象初始化：

Code Playground

```
const name = 'Robin';

const user = {
  name: name,
};
```

当你的对象中的属性名与变量名相同时，您可以执行以下的操作：

Code Playground

```
const name = 'Robin';

const user = {
  name,
};
```

在应用程序中，你也可以这样做。列表变量名和状态属性名称共享同一名称。

Code Playground

```
// ES5
this.state = {
  list: list,
};

// ES6
this.state = {
  list,
};
```

另一个简洁的辅助办法是简写方法名。在 ES6 中，你能更简洁地初始化一个对象的方法。

Code Playground

```
// ES5
var userService = {
  getUserName: function (user) {
    return user.firstname + ' ' + user.lastname;
  },
};

// ES6
const userService = {
  getUserName(user) {
    return user.firstname + ' ' + user.lastname;
  },
};
```

最后值得一提的是你可以在 ES6 中使用计算属性名。

Code Playground

```
// ES5
var user = {
  name: 'Robin',
};

// ES6
const key = 'name';
const user = {
  [key]: 'Robin',
};
```

或许你目前还觉得计算属性名没有意义。为什么需要他们呢? 在后续的章节中，当你为一个对象动态地根据 key 分配值时便会涉及到。在 JavaScript 中生成查找表是很简单的。

练习：

- ES6 对象初始化练习
- 阅读更多关于 ES6 对象初始化[61]

[61] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Object_initializer

单向数据流

现在你的组件中有一些内部的 state。但是你还没有操纵它们，因此 state 是静态的。一个练习 state 操作好方法是增加一些组件的交互。

让我们为列表中的每一项增加一个按钮。按钮的文案为“Dismiss”，意味着将从列表中删除该项。这个按钮在你希望保留未读列表和删除不感兴趣的项时会非常有用。

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.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>
            <span>
              <button
                onClick={() => this.onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

这个类方法 `onDismiss()` 还没有被定义，我们稍后再来做这件事。目前先把重点放在按钮元素的 ‘ onClick ‘ 事件处理器上。正如你看见的，`onDismiss()` 方法被另外一个函数包裹在 ‘ onClick ‘ 事件处理器中，它是一个箭头函数。这样你可以拿到 `item` 对象中的 `objectID` 属

性来确定那一项会被删除掉。另外一种方法是在 ‘ onClick ‘ 处理器之外定义函数，并只将已定义的函数传到处理器。在后续的章节中会解释更多关于元素处理器的细节。

你有没有注意到按钮元素是多行代码的？元素中一行有多个属性会看起来比较混乱。所以这个按钮使用多行格式来书写以保持它的可读性。这虽然不是强制的，但这是我的极力推荐的代码风格。

现在你需要来完成 `onDismiss()` 的功能，它通过 id 来标示那一项需要被删除。此函数绑定到类，因此成为类方法。这就是为什么你访问它使用 `this.onDismiss()` 而不是 `onDismiss()`。`this` 对象是类的实例，为了将 `onDismiss()` 定义为类方法，你需要在构造函数中绑定它。绑定稍后将在另一章中详细解释。

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

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

  render() {
    ...
  }
}
```

下一步，你需要在类中定义它的功能和业务逻辑。类方法可以用以下方式定义。

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onDismiss = this.onDismiss.bind(this);
```

```
  }

  onDismiss(id) {
    ...
  }

  render() {
    ...
  }
}
```

现在你可以定义方法内部的功能。总的来说你希望从列表中删除由 id 标识的项，并且保存更新后的列表到 state 中。随后这个更新后列表被使用到再次运行的 `render()` 方法中并渲染，最后这个被删除项就不再显示了。

你可以通过 JavaScript 内置的 filter 方法来删除列表中的一项。fitler 方法以一个函数作为输入。这个函数可以访问列表中的每一项，因为它会遍历整个列表。通过这种方式，你可以基于过滤条件来判断列表的每一项。如果该项判断结果为 true，则该项保留在列表中。否则将从列表中过滤掉。另外，好的一点是这个方法会返回一个新的列表而不是改变旧列表。它遵循了 React 中不可变数据的约定。

src/App.js

```
onDismiss(id) {
  const updatedList = this.state.list.filter(function isNotId(item) {
    return item.objectID !== id;
  });
}
```

在下一步中，你可以抽取函数并将其传递给 filter 函数。

src/App.js

```
onDismiss(id) {
  function isNotId(item) {
    return item.objectID !== id;
  }

  const updatedList = this.state.list.filter(isNotId);
}
```

另外，可以通过使用 ES6 的箭头函数让代码更简洁。

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
}
```

你甚至可以内联到一行内完成，就像在按钮的 `onClick` 事件处理器做的一样，但如此会损失一些可读性。

src/App.js

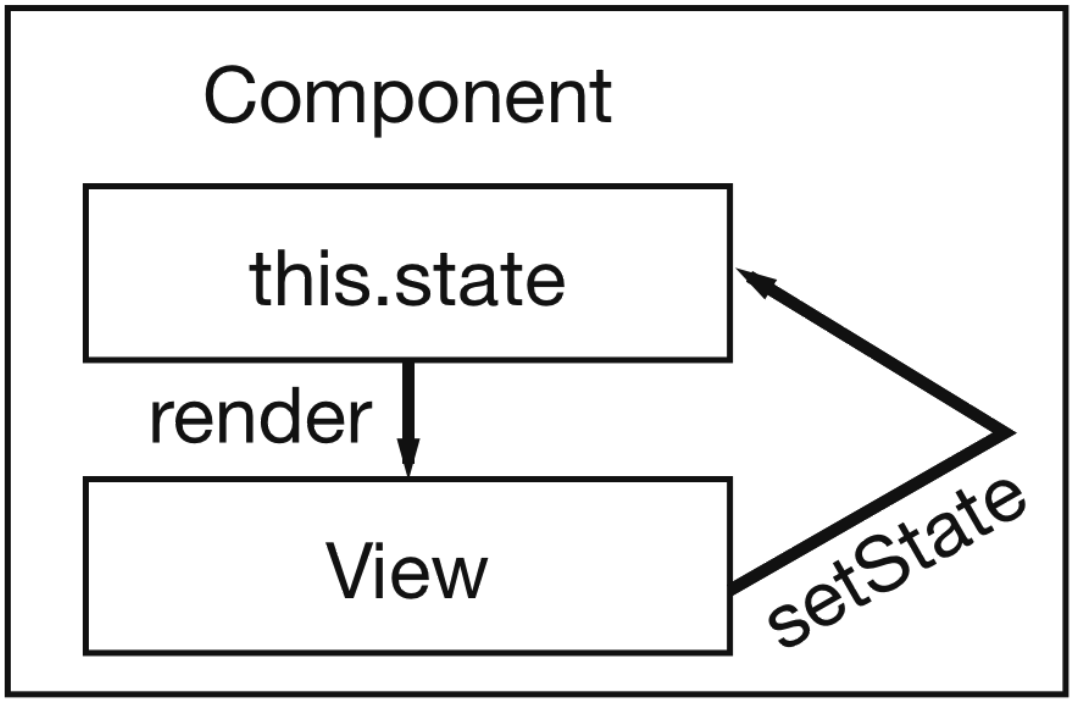
```
onDismiss(id) {
  const updatedList = this.state.list.filter(item => item.objectID !== id);
}
```

现在已经从列表中删除了点击项，但是 state 还并没有更新。因此你需要最后使用类方法 `setState()` 来更新组件 satate 中的列表了。

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
  this.setState({ list: updatedList });
}
```

现在重新运行你的程序并尝试点击“Dismiss”按钮，它应该是工作的。你现在所练习的就是 React 中的单向数据流。你在界面通过 `onClick` 触发一个动作，再通过函数或类方法修改组件的 state，最后组件的 `render()` 方法再次运行并更新界面。



Internal state update with unidirectional data flow

练习：

- 阅读更多关于 React 的状态与生命周期[62]

[62] https://facebook.github.io/react/docs/state-and-lifecycle.html

绑定

当使用 ES6 编写的 React 组件时，了解在 JavaScript 类的绑定会非常重要。在前面章节，你已经在构造函数中绑定了 `onDismiss()` 方法

src/App.js

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      list,
    };

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

  ...
}
```

为什么一开始就需要这么做呢？绑定的步骤是非常重要的，因为类方法不会自动绑定 `this` 到实例上。让我们通过下面的代码来做验证。

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

组件正确的渲染，但是当你点击按钮时候，你会在开发调试控制台中得到 `undefined` 。这是使用 React 主要的 bug 来源，因为当你想在类方法中访问 `this.state` 时，由于 `this` 是 `undefined` 所以并不能被检索到。所以为了确保 `this` 在类方法中是可访问的，你需要将 `this` 绑定到类方法上。

在下面的组件中，类方法在构造函数中正确绑定。

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

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

  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

再次尝试点击按钮，这个 `this` 对象就指向了类的实例。你现在就可以访问到 `this.state` 或者是后面会学习到的 `this.props` 。

类方法的绑定也可以写起其他地方，比如写在 `render()` 函数中。

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe.bind(this)}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

但是你应该避免这样做，因为它会在每次 `render()` 方法执行时绑定类方法。总结来说组件每次运行更新时都会导致性能消耗。当在构造函数中绑定时，绑定只会在组件实例化时运行一次，这样做是一个更好的方式。

另外有一些人们提出在构造函数中定义业务逻辑类方法。

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.onClickMe = () => {
      console.log(this);
    }
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
```

```
      </button>
    );
  }
}
```

你同样也应该避免这样，因为随着时间的推移它会让你的构造函数变得混乱。构造函数目的只是实例化你的类以及所有的属性。这就是为什么我们应该把业务逻辑应该定义在构造函数之外。

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.doSomething = this.doSomething.bind(this);
    this.doSomethingElse = this.doSomethingElse.bind(this);
  }

  doSomething() {
    // do something
  }

  doSomethingElse() {
    // do something else
  }

  ...
}
```

最后值得一提的是类方法可以通过 ES6 的箭头函数做到自动地绑定。

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe = () => {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
```

```
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

如果在构造函数中的重复绑定对你有所困扰，你可以使用这种方式代替。React 的官方文档中坚持在构造函数中绑定类方法，所以本书也会采用同样的方式。

练习：

- 尝试绑定不同的方法并在控制台中打印 `this` 对象

事件处理

本章节会让你对元素的事件处理有更深入的了解，在你的应用程序中，你将使用下面的按钮来从列表中忽略一项内容。

src/App.js

```
...

<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

这已经是一个复杂的例子了，因为你必须传递一个参数到类的方法，因此你需要将它封装到另一个（箭头）函数中，基本上，由于要传递给事件处理器使用，因此它必须是一个函数。下面的代码不会工作，因为类方法会在浏览器中打开程序时立即执行。

src/App.js

```
...

<button
  onClick={this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

当使用 `onClick={doSomething()}` 时，`doSomething()` 函数会在浏览器打开程序时立即执行，由于监听表达式是函数执行的返回值而不再是函数，所以点击按钮时不会有任何事发生。但当使用 `onClick={doSomething}` 时，因为 `doSomething` 是一个函数，所以它会在点击按钮时执行。同样的规则也适用于在程序中使用的 `onDismiss()` 类方法。

然而，使用 `onClick={this.onDismiss}` 并不够，因为这个类方法需要接收 item.objectID 属性来识别那个将要被忽略的项，这就是为什么它需要被封装到另一个函数中来传递这个属性。这个概念在 JavaScript 中被称为高阶函数，稍后会做简要解释。

src/App.js

```
...

<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

其中一个解决方案是在外部定义一个包装函数，并且只将定义的函数传递给处理程序。因为需要访问特定的列表项，所以它必须位于 map 函数块的内部。

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item => {
          const onHandleDismiss = () =>
            this.onDismiss(item.objectID);

          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>
              <span>
                <button
                  onClick={onHandleDismiss}
                  type="button"
                >
                  Dismiss
```

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

毕竟，传给元素事件处理器的内容必须是函数。作为一个示例，请尝试以下代码：

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
            ...
            <span>
              <button
                onClick={console.log(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

它会在浏览器加载该程序时执行，但点击按钮时并不会。而下面的代码只会在点击按钮时执行。它是一个在触发事件时才会执行的函数。

src/App.js

```
...

<button
  onClick={function () {
    console.log(item.objectID)
  }}
  type="button"
>
  Dismiss
</button>

...
```

为了保持简洁，你可以把它转成一个 JavaScript ES6 的箭头函数，和我们在 `onDismiss()` 类方法时做的一样。

src/App.js

```
...

<button
  onClick={() => console.log(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

经常会有 React 初学者在事件处理中使用函数遇到困难，这就是为什么我要在这里更详细的解释它。最后，你应该使用下面的代码来拥有一个可以访问 `item` 对象的 `objectID` 属性简洁的内联 JavaScript ES6 箭头函数。

src/App.js

```
class App extends Component {
  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
          <div key={item.objectID}>
            ...
            <span>
              <button
                onClick={() => this.onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

另一个经常会被提到的性能相关话题是在事件处理程序中使用箭头函数的影响。例如，onClick 事件处理中的 onDismiss() 方法被封装在另一个箭头函数中以便能传递项标识。因此每次 render() 执行时，事件处理程序就会实例化一个高阶箭头函数，它可能会对你的程序性能产生影响，但在大多数情况下你都不会注意到这个问题。假设你有一个包含1000个项目的巨大数据表，每一行或者列在事件处理程序中都有这样一个箭头函数，这个时候就需要考虑性能影响，因此你可以实现一个专用的按钮组件来在构造函数中绑定方法，但这是一个不成熟的优化。所以现在，专注到学习 React 会更有价值。

练习：

- 尝试在按钮的 onClick 处理程序中使用函数的不同方法。

和表单交互

让我们在程序中加入表单来体验 React 和表单事件的交互，我们将在程序中加入搜索功能，列表会根据输入框的内容对标题进行过滤。

第一步，你需要在 JSX 中定义一个带有输入框的表单。

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input type="text" />
        </form>
        {this.state.list.map(item =>
          ...
        )}
      </div>
    );
  }
}
```

在下面的场景中，将会使用在输入框中的内容作为搜索字段来临时过滤列表。为了能根据输入框的值过滤列表，你需要将输入框的值储存在你的本地状态中，但是如何访问这个值呢？你可以使用 React 的合成事件来访问事件返回值。

让我们为输入框定义一个 `onChange` 处理程序。

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
```

```
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        ...
      </div>
    );
  }
}
```

这个函数被绑定到组件上，因此再次成为一个类方法，你必定义方法并 bind 它。

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  onSearchChange() {
    ...
  }

  ...
}
```

在元素中使用监听时，你可以在回调函数的签名中访问到 React 的合成事件。

src/App.js

```
class App extends Component {

  ...

  onSearchChange(event) {
    ...
  }

  ...
}
```

event 对象的 target 属性中带有输入框的值，因此你可以使用 `this.setState()` 来更新本地的搜索词的状态了。

src/App.js

```
class App extends Component {

  ...

  onSearchChange(event) {
    this.setState({ searchTerm: event.target.value });
  }

  ...
}
```

此外，你应该记住在构造函数中为 `searchTerm` 定义初始状态，输入框在开始时应该是空的，因此初始值应该是空字符串。

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
      searchTerm: '',
    };
```

```
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...
}
```

现在你将会把输入框每次变化的输入值储存到组件的内部状态中。

关于一个在 React 组件中更新状态的简要说明，在使用 `this.setState()` 更新 `searchTerm` 时应该把这个列表也传递进去来保留它才是公平的，但事实并非如此，React 的 `this.setState()` 是一个浅合并，在更新一个唯一的属性时，他会保留状态对象中的其他属性，因此即使你已经在列表状态中排除了一个项，在更新 `searchTerm` 属性时也会保持不变。

让我们回到你的程序中，现在列表还没有根据储存在本地状态中的输入字段进行过滤。基本上，你已经具有了根据 `searchTerm` 临时过滤列表的所有东西。那么怎么暂时的过滤它呢？你可以在 `render()` 方法中，在 map 映射列表之前，插入一个过滤的方法。这个过滤方法将只会匹配标题属性中有 `searchTerm` 内容的列表项。你已经使用过了 JavaScript 内置的 filter 功能，让我们再用一次，你可以在 map 函数之前加入 filter 函数，因为 filter 函数返回一个新的数组，所以 map 函数可以这样方便的使用。

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        {this.state.list.filter(...).map(item =>
          ...
        )}
      </div>
    );
```

```
  }
}
```

让我们用一种不同的方式来处理过滤函数，我们想在 ES6 组件类之外定义一个传递给过滤函数的函数参数，在这里我们不能访问到组件内的状态，所以无法访问 `searchTerm` 属性来作为筛选条件求值，我们需要传递 `searchTerm` 到过滤函数并返回一个新函数来根据条件求值，这叫做高阶函数。

一般来说，我不会提到高阶函数，但在 React 中了解高阶函数是有意义的，因为在 React 中有一个高阶组件的概念，你将在这本书的后面了解到这个概念。但现在，让我们关注到过滤器的功能。

首先，你需要在 App 组件外定义一个高阶函数。

src/App.js

```
function isSearched(searchTerm) {
  return function(item) {
    // some condition which returns true or false
  }
}

class App extends Component {

  ...

}
```

该函数接受 `searchTerm` 并返回另一个函数，因为所有的 filter 函数都接受一个函数作为它的输入，返回的函数可以访问列表项目对象，因为它是传给 filter 函数的函数。此外，返回的函数将会根据函数中定义的条件对列表进行过滤。让我们来定义条件。

src/App.js

```
function isSearched(searchTerm) {
  return function(item) {
    return item.title.toLowerCase().includes(searchTerm.toLowerCase());
  }
}

class App extends Component {

  ...

}
```

条件是列表中项目的标题属性和输入的 searchTerm 参数相匹配，你可以使用 JavaScript 内置的 includes 功能来实现这一点。只有满足匹配时才会返回 true 并将项目保留在列表中。当不匹配时，项目会从列表中移除。但需要注意的是，你需要把输入内容和待匹配的内容都转换成小写，否则，搜索词 'redux' 和列表中标题叫 'Redux' 的项目无法匹配。由于我们使用的是一个不可变的列表，并使用 filter 函数返回一个新列表，所以本地状态中的原始列表根本就没有被修改过。

还有一点需要注意，我们使用了 Javascript 内置的 includes 功能，它已经是一个 ES6 的特性了。这在 ES5 中该如何实现呢？你将使用 indexOf() 函数来获取列表中项的索引，当项目在列表中时，indexOf() 将会返回它的索引。

Code Playground

```
// ES5
string.indexOf(pattern) !== -1

// ES6
string.includes(pattern)
```

另一个优雅的重构可以用 ES6 箭头函数完成，它可以让函数更加整洁：

Code Playground

```
// ES5
function isSearched(searchTerm) {
  return function(item) {
    return item.title.toLowerCase().includes(searchTerm.toLowerCase());
  }
}

// ES6
const isSearched = searchTerm => item =>
  item.title.toLowerCase().includes(searchTerm.toLowerCase());
```

人们会争论哪个函数更易读，就我个人而言，我更习惯第二个。React 的生态使用了大量的函数式编程概念。通常情况下，你会使用一个函数返回另一个函数（高阶函数）。在 JavaScript ES6 中，可以使用箭头函数更简洁的表达这些。

最后，你需要使用定义的 isSearched() 函数来过滤你的列表，你从本地状态中传递 searchTerm 属性返回一个根据条件过滤列表的输入过滤函数。之后它会映射过滤后的列表用于显示每个列表项的元素。

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        {this.state.list.filter(isSearched(this.state.searchTerm)).map(item =>
          ...
        )}
      </div>
    );
  }
}
```

搜索功能现在应该起作用了，在浏览器中自己尝试一下。

练习：

- 阅读更多 React 事件[63] 相关内容
- 阅读更多 高阶函数[64] 相关内容

[63] https://facebook.github.io/react/docs/handling-events.html
[64] https://en.wikipedia.org/wiki/Higher-order_function

ES6 解构

在 JavaScript ES6 中有一种更方便的方法来访问对象和数组的属性，叫做解构。比较下面 JavaScript ES5 和 ES6 的代码片段。

Code Playground

```
const user = {
  firstname: 'Robin',
  lastname: 'Wieruch',
};

// ES5
var firstname = user.firstname;
var lastname = user.lastname;

console.log(firstname + ' ' + lastname);
// output: Robin Wieruch

// ES6
const { firstname, lastname } = user;

console.log(firstname + ' ' + lastname);
// output: Robin Wieruch
```

在 JavaScript ES5 中每次访问对象的属性都需要额外添加一行代码，但在 JavaScript ES6 中可以在一行中进行。可读性最好的方法是在将对象解构成多个属性时使用多行。

Code Playground

```
const {
  firstname,
  lastname
} = user;
```

对于数组一样可以使用解构，同样，多行代码会使你的代码保持可读性。

Code Playground

```
const users = ['Robin', 'Andrew', 'Dan'];
const [
  userOne,
  userTwo,
  userThree
] = users;

console.log(userOne, userTwo, userThree);
// output: Robin Andrew Dan
```

也许你已经注意到，程序组件内的状态对象也可以使用同样的方式解构，你可以让 map 和 filter 部分的代码更简短。

src/App.js

```
  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        ...
        {list.filter(isSearched(searchTerm)).map(item =>
          ...
        )}
      </div>
    );
```

你也可以使用 ES5 或者 ES6 的方式来做：

Code Playground

```
// ES5
var searchTerm = this.state.searchTerm;
var list = this.state.list;

// ES6
const { searchTerm, list } = this.state;
```

但由于这本书大部分时候都使用了 JavaScript ES6，所以你也可以坚持使用它。

练习：

- 阅读更多ES6 解构[65]的相关内容

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

受控组件

你已经了解了 React 中的单向数据流，同样的规则适用于更新本地状态 `searchTerm` 来过滤列表的输入框。当状态变化时，`render()` 方法将再次运行，并使用最新状态中的 `searchTerm` 值来作为过滤条件。

但是我们是否忘记了输入元素的一些东西？一个 HTML 输入标签带有一个 `value` 属性，这个属性通常有一个值作为输入框的显示，在本例中，它是 `searchTerm` 属性。然而，看起来我们在 React 好像并不需要它。

这是错误的，表单元素比如 `<input>`，`<textarea>` 和 `<select>` 会以原生 HTML 的形式保存他们自己的状态。一旦有人从外部做了一些修改，它们就会修改内部的值，在 React 中这被称为不受控组件，因为它们自己处理状态。在 React 中，你应该确保这些元素变为受控组件。

你应该怎么做呢？你只需要设置输入框的值属性，这个值已经在 `searchTerm` 状态属性中保存了，那么为什么不从这里访问呢？

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <form>
          <input
            type="text"
            value={searchTerm}
            onChange={this.onSearchChange}
          />
        </form>
        ...
      </div>
    );
  }
}
```

就是这样。现在输入框的单项数据流循环是自包含的，组件内部状态是输入框的唯一数据来源。

整个内部状态管理和单向数据流可能对你来说比较新，但你一旦习惯了它，你就会自然而然的在 React 中实现它。一般来说，React 带来一种新的模式，将单向数据流引入到单页面应用的生态中，到目前为止，它已经被几个框架和库所采用。

练习

- 阅读更多React 表单[66]相关内容

[66] https://facebook.github.io/react/docs/forms.html

拆分组件

现在，你有一个大型的 App 组件。它在不停地扩展，最终可能会变得混乱。你可以开始将它拆分成若干个更小的组件。

让我们开始使用一个用于搜索的输入组件和一个用于展示的列表组件。

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search />
        <Table />
      </div>
    );
  }
}
```

你可以给组件传递属性并在组件中使用它们。至于 App 组件，它需要传递由本地状态 (state) 托管的属性和它自己的类方法。

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search
          value={searchTerm}
          onChange={this.onSearchChange}
        />
        <Table
          list={list}
```

```
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

现在你可以接着 App 组件定义这些组件。这些组件仍然是 ES6 类组件，它们会渲染和之前相同的元素。

第一个是 Search 组件。

src/App.js

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

class Search extends Component {
  render() {
    const { value, onChange } = this.props;
    return (
      <form>
        <input
          type="text"
          value={value}
          onChange={onChange}
        />
      </form>
    );
  }
}
```

第二个是 Table 组件。

src/App.js

```
...

class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        {list.filter(isSearched(pattern)).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>
            <span>
              <button
                onClick={() => onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

现在你有了三个 ES6 类组件。你可能已经注意到，`props` 对象可以通过这个类实例的 `this` 来访问。props 是 properties 的简写，当你在 App 组件里面使用它时，它有你传递给这些组件的所有值。这样，组件可以沿着组件树向下传递属性。

从 App 组件中提取这些组件之后，你就可以在别的地方去重用它们了。因为组件是通过 props 对象来获取它们的值，所以当你在别的地方重用它时，你可以每一次都传递不同的 props，这些组件就变得可复用了。

练习：

- 从已经完成的 Search 和 Table 组件中找出可以进一步提取的组件。

- 但是不要现在就去做，否则在接下来的几个章节你会遇到冲突。

可组合组件

在 props 对象中还有一个小小的属性可供使用：`children 属性`。通过它你可以将元素从上层传递到你的组件中，这些元素对你的组件来说是未知的，但是却为组件相互组合提供了可能性。让我们来看一看，当你只将一个文本（字符串）作为子元素传递到 Search 组件中会怎样。

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search
          value={searchTerm}
          onChange={this.onSearchChange}
        >
          Search
        </Search>
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

现在 Search 组件可以从 props 对象中解构出 children 属性。然后它就可以指定这个 children 应该显示在哪里。

src/App.js

```
class Search extends Component {
  render() {
    const { value, onChange, children } = this.props;
    return (
      <form>
        {children} <input
          type="text"
          value={value}
          onChange={onChange}
        />
      </form>
    );
  }
}
```

现在，你应该可以在输入框旁边看到这个 “Search” 文本了。当你在别的地方使用 Search 组件时，如果你喜欢，你可以选择一个不同的文本。总之，它不仅可以把文本作为子元素传递，还可以将一个元素或者元素树（它还可以再次封装成组件）作为子元素传递。children 属性让组件相互组合到一起成为可能。

练习：

- 阅读更多关于 React 组件模型[67] 的内容

[67] https://facebook.github.io/react/docs/composition-vs-inheritance.html

可复用组件

可复用和可组合组件让你能够思考合理的组件分层，它们是 React 视图层的基础。前面几章提到了可重用性的术语。现在你可以复用 Search 和 Table 组件了。甚至 App 组件都是可复用的了，因为你可以在别的地方重新实例化它。

让我们再来定义一个可复用组件 Button，最终会被更频繁地复用。

src/App.js

```
class Button extends Component {
  render() {
    const {
      onClick,
      className,
      children,
    } = this.props;

    return (
      <button
        onClick={onClick}
        className={className}
        type="button"
      >
        {children}
      </button>
    );
  }
}
```

声明这样一个组件可能看起来有点多余。你将会用 `Button` 组件来替代 `button` 元素。它只省去了 `type="button"`。当你想使用 Button 组件的时候，你还得去定义除了 type 之外的所有属性。但这里你必须要考虑到长期投资。想象在你的应用中有若干个 button，但是你想改变它们的一个属性、样式或者行为。如果没有这个组件的话，你就必须重构每个 button。相反，Button 组件拥有单一可信数据源。一个 Button 组件可以立即重构所有 button。一个 Button 组件统治所有的 button。

既然你已经有了 button 元素，你可以用 Button 组件代替。它省略了 type 属性，因为 Button 组件已经指定了。

src/App.js

```
class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        {list.filter(isSearched(pattern)).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>
            <span>
              <Button onClick={() => onDismiss(item.objectID)}>
                Dismiss
              </Button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Button 组件期望在 props 里面有一个 `className` 属性. `className` 属性是 React 基于 HTML 属性 `class` 的另一个衍生物。但是当使用 Button 组件时，我们并没有传递任何 `className` 属性，所以在 Button 组件的代码中，我们更应该明确地标明 `className` 是可选的。

因此，你可以使用默认参数，它是一个 JavaScript ES6 的特性。

src/App.js

```
class Button extends Component {
  render() {
    const {
      onClick,
      className = '',
      children,
    } = this.props;
```

```
    ...
  }
}
```

这样当使用 Button 组件时，若没有指定 `className` 属性，它的值就是一个空字符串，而非 `undefined`。

练习：

- 阅读更多关于 ES6 默认参数[68] 的内容 ## Component Declarations 组件声明

现在你已经有四个 ES6 类组件了，但是你可以做得更好。让我来介绍一下函数式无状态组件 (functional stateless components)，作为除了 ES6 类组件的另一个选择。在重构你的组件之前，让我来介绍一下 React 不同的组件类型。

- 函数式无状态组件: 这类组件就是函数，它们接收一个输入并返回一个输出。输入是 props，输出就是一个普通的 JSX 组件实例。到这里，它和 ES6 类组件非常的相似。然而，函数式无状态组件是函数（函数式的），并且它们没有本地状态（无状态的）。你不能通过 `this.state` 或者 `this.setState()` 来访问或者更新状态，因为这里没有 `this` 对象。此外，它也没有生命周期方法。虽然你还没有学过生命周期方法，但是你已经用到了其中两个：`constructor()` and `render()`。constructor 在一个组件的生命周期中只执行一次，而 `render()` 方法会在最开始执行一次，并且每次组件更新时都会执行。当你阅读到后面关于生命周期方法的章节时，要记得函数式无状态组件是没有生命周期方法的。
- **ES6** 类组件: 在你的四个组件中，你已经使用过这类组件了。在类的定义中，它们继承自 React 组件。`extend` 会注册所有的生命周期方法，只要在 React component API 中，都可以在你的组件中使用。通过这种方式你可以使用 `render()` 类方法。此外，通过使用 `this.state` 和 `this.setState()`，你可以在 ES6 类组件中储存和操控 state。
- **React.createClass**: 这类组件声明曾经在老版本的 React 中使用，仍然存在于很多 ES5 React 应用中。但是为了支持 JavaScript ES6，Facebook 声明它已经不推荐使用了[69]。他们还在 React 15.5 中加入了不推荐使用的警告[70]。你不会在本书使用它。

因此这里基本只剩下两种组件声明了。但是什么时候更适合使用函数式无状态组件而非 ES6 类组件？一个经验法则就是当你不需要本地状态或者组件生命周期方法时，你就应该使用函数式无状态组件。最开始一般使用函数式无状态组件来实现你的组件，一旦你需要访问 state 或者生命周期方法时，你就必须要将它重构成一个 ES6 类组件。在我们的应用中，为了学习 React，我们采取了相反的方式。

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

[69] https://facebook.github.io/react/blog/2015/03/10/react-v0.13.html

[70] https://facebook.github.io/react/blog/2017/04/07/react-v15.5.0.html

让我们回到你的应用中。App 组件使用内部状态，这就是为什么它必须作为 ES6 类组件存在的原因。但是你的其他三个 ES6 类组件都是无状态的，它们不需要使用 `this.state` 或者 `this.setState()`，甚至都不需要使用生命周期函数。让我们一起把 Search 组件重构成一个函数式无状态组件。Table 和 Button 组件的重构会留做你的练习。

src/App.js

```
function Search(props) {
  const { value, onChange, children } = props;
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

基本上就是这样了。props 可以在函数签名[71]（译者注：这里应指函数入参）中访问，返回值是 JSX。你已经知道 ES6 解构了，所以在函数式无状态组件中，你可以优化之前的写法。最佳实践就是在函数签名中通过解构 props 来使用它。

src/App.js

```
function Search({ value, onChange, children }) {
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

但是它还可以变得更好。你已经知道，ES6 箭头函数允许让你保持你的函数简洁。你可以移除函数的块声明（译者注：即花括号 {}）。在简化的函数体中，表达式会自动作为返回值，因此你可以将 return 语句移除。因为你的函数式无状态组件是一个函数，你同样可以用这种方式来简化它。

[71] https://developer.mozilla.org/zh-CN/docs/Glossary/Signature/Function

src/App.js

```
const Search = ({ value, onChange, children }) =>
  <form>
    {children} <input
      type="text"
      value={value}
      onChange={onChange}
    />
  </form>
```

最后一步对于强制只用 props 作为输入和 JSX 作为输出非常有用。这之间没有任何别的东西。但是你仍然可以在 ES6 箭头函数块声明中做一些事情。

Code Playground

```
const Search = ({ value, onChange, children }) => {

  // do something

  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

但是你现在并不需要这样做，这也是为什么你可以让之前的版本没有块声明。当使用块声明时，人们往往容易在这个函数里面做过多的事情。通过移除块声明，你可以专注在函数的输入和输出上。

现在你已经有一个轻量的函数式无状态组件了。一旦你需要访问它的内部组件状态或者生命周期方法，你最好将它重构成一个 ES6 类组件。另外，你也已经看到，JavaScript ES6 是如何被用到 React 组件中并让它们变得更加的简洁和优雅。

练习：

- 将 Table 和 Button 组件重构成函数式无状态组件
- 阅读更多关于 ES6 类组件和函数式无状态组件[72] 的内容

[72] https://facebook.github.io/react/docs/components-and-props.html

给组件声明样式

让我们给你的应用和组件添加一些基本的样式。你可以复用 *src/App.css* 和 *src/index.css* 文件。因为你是用 *create-react-app* 来创建的，所以这些文件应该已经在你的项目中了。它们应该也被引入到你的 *src/App.js* 和 *src/index.js* 文件中了。我准备了一些 CSS，你可以直接复制粘贴到这些文件中，你也可以随意使用你自己的样式。

首先，给你的整个应用声明样式。

src/index.css

```
body {
  color: #222;
  background: #f4f4f4;
  font: 400 14px CoreSans, Arial,sans-serif;
}

a {
  color: #222;
}

a:hover {
  text-decoration: underline;
}

ul, li {
  list-style: none;
  padding: 0;
  margin: 0;
}

input {
  padding: 10px;
  border-radius: 5px;
  outline: none;
  margin-right: 10px;
  border: 1px solid #dddddd;
}

button {
  padding: 10px;
  border-radius: 5px;
  border: 1px solid #dddddd;
```

```
  background: transparent;
  color: #808080;
  cursor: pointer;
}

button:hover {
  color: #222;
}

*:focus {
  outline: none;
}
```

其次，在 App 文件中给你的组件声明样式。

src/App.css

```
.page {
  margin: 20px;
}

.interactions {
  text-align: center;
}

.table {
  margin: 20px 0;
}

.table-header {
  display: flex;
  line-height: 24px;
  font-size: 16px;
  padding: 0 10px;
  justify-content: space-between;
}

.table-empty {
  margin: 200px;
  text-align: center;
  font-size: 16px;
}
```

```
.table-row {
  display: flex;
  line-height: 24px;
  white-space: nowrap;
  margin: 10px 0;
  padding: 10px;
  background: #ffffff;
  border: 1px solid #e3e3e3;
}

.table-header > span {
  overflow: hidden;
  text-overflow: ellipsis;
  padding: 0 5px;
}

.table-row > span {
  overflow: hidden;
  text-overflow: ellipsis;
  padding: 0 5px;
}

.button-inline {
  border-width: 0;
  background: transparent;
  color: inherit;
  text-align: inherit;
  -webkit-font-smoothing: inherit;
  padding: 0;
  font-size: inherit;
  cursor: pointer;
}

.button-active {
  border-radius: 0;
  border-bottom: 1px solid #38BB6C;
}
```

现在你可以在一些组件中使用这些样式。但是别忘了使用 React 的 `className`，而不是 HTML 的 `class` 属性。

首先，将它应用到你的 App ES6 类组件中。

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
          >
            Search
          </Search>
        </div>
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

其次，将它应用到你的 Table 函数式无状态组件中。

src/App.js

```
const Table = ({ list, pattern, onDismiss }) =>
  <div className="table">
    {list.filter(isSearched(pattern)).map(item =>
      <div key={item.objectID} className="table-row">
        <span>
          <a href={item.url}>{item.title}</a>
        </span>
        <span>{item.author}</span>
        <span>{item.num_comments}</span>
        <span>{item.points}</span>
        <span>
```

```
          <Button
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
            Dismiss
          </Button>
        </span>
      </div>
    )}
  </div>
```

现在你已经给你的应用和组件添加了基本的 CSS 样式，看起来应该非常不错。如你所知，JSX 混合了 HTML 和 JavaScript。现在有人呼吁将 CSS 也加入进去，这就叫作内联样式 (inline style)。你可以定义 JavaScript 对象，并传给一个元素的 style 属性。

让我们通过使用内联样式来使 Table 的列宽自适应。

src/App.js

```
const Table = ({ list, pattern, onDismiss }) =>
  <div className="table">
    {list.filter(isSearched(pattern)).map(item =>
      <div key={item.objectID} className="table-row">
        <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
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
            Dismiss
          </Button>
        </span>
```

```
      </div>
    )}
  </div>
```

现在样式已经内联了。你可以在你的元素之外定义一个 style 对象，这样可以让它变得更整洁。

Code Playground

```
const largeColumn = {
  width: '40%',
};

const midColumn = {
  width: '30%',
};

const smallColumn = {
  width: '10%',
};
```

随后你可以将它们用于你的 columns：`<span style={smallColumn}>`。

总而言之，关于 React 中的样式，你会找到不同的意见和解决方案。现在你已经用过纯 CSS 和内联样式了。这足以开始。

在这里我不想下定论，但是想给你一些更多的选择。你可以自行阅读并应用它们。但是如果你刚开始使用 React，目前我会推荐你坚持纯 CSS 和内联样式。

- styled-components[73]
- CSS Modules[74]

[73] https://github.com/styled-components/styled-components
[74] https://github.com/css-modules/css-modules

你已经学习了编写一个 React 应用所需要的基础知识了！ 让我们来回顾一下前面几个章节：

- React
  - 使用 `this.state` 和 `setState()` 来管理你的内部组件状态
  - 将函数或者类方法传递到你的元素处理器
  - 在 React 中使用表单或者事件来添加交互
  - 在 React 中单向数据流是一个非常重要的概念
  - 拥抱 controlled components
  - 通过 children 和可复用组件来组合组件
  - ES6 类组件和函数式无状态组件的使用方法和实现
  - 给你的组件声明样式的方法
- ES6
  - 绑定到一个类的函数叫作类方法
  - 解构对象和数组
  - 默认参数
- General
  - 高阶函数

该休息一下了，吸收这些知识然后转化成你自己的东西。你可以用你已有的代码来做个实验。另外，你可以进一步阅读官方文档[75]

你可以在官方代码仓库[76]找到源码。

[75] https://facebook.github.io/react/docs/installation.html

[76] https://github.com/rwieruch/hackernews-client/tree/4.2

使用真实的 API

现在是时候使用真实的 API 了，老是处理样本数据会变得很无聊。

如果你对 API 不熟悉，我建议你去读读我的博客，里面有关于我是怎样了解 API 的[77]。

你知道 Hacker News[78] 这个平台吗？它是一个很棒的技术新闻整合平台。在本书中，你将使用它的 API 来获取热门资讯。它有一个基础 API [79] 和一个搜索 API [80]来获取数据。后者使我们可以去搜索 Hacker News 上的资讯。你也可以通过 API 规范来了解它的数据结构。

[77] https://www.robinwieruch.de/what-is-an-api-javascript/

[78] https://news.ycombinator.com/

[79] https://github.com/HackerNews/API

[80] https://hn.algolia.com/api

生命周期方法

在你开始在组件中通过 API 来获取数据之前，你需要知道 React 的生命周期方法。这些方法是嵌入 React 组件生命周期中的一组挂钩。它们可以在 ES6 类组件中使用，但是不能在无状态组件中使用。

你还记得前章中讲过的 JavaScript ES6 类以及如何在 React 中使用它们吗？除了 `render()` 方法外，还有几个方法可以在 React ES6 类组件中被覆写。所有的这些都是生命周期方法。现在让我们来深入了解他们：

通过之前的学习，你已经知道两种能够用在 ES6 类组件中的生命周期方法：`constructor()` 和 `render()`。

constructor（构造函数）只有在组件实例化并插入到 DOM 中的时候才会被调用。组件实例化的过程称作组件的挂载（mount）。

`render()` 方法也会在组件挂载的过程中被调用，同时当组件更新的时候也会被调用。每当组件的状态（state）或者属性（props）改变时，组件的 `render()` 方法都会被调用。

现在你了解了更多关于这两个生命周期方法的知识，也知道它们什么时候会被调用了。你也已经在前面的学习中使用过它们了。但是 React 里还有更多的生命周期方法。

在组件挂载的过程中还有另外两个生命周期方法：`componentWillMount()` 和 `componentDidMount()`。构造函数（constructor）最先执行，`componentWillMount()` 会在 `render()` 方法之前执行，而 `componentDidMount()` 在 `render()` 方法之后执行。

总而言之，在挂载过程中有四个生命周期方法，它们的调用顺序是这样的：

- constructor()
- componentWillMount()
- render()
- componentDidMount()

但是当组件的状态或者属性改变的时候用来更新组件的生命周期是什么样的呢？总的来说，它一共有5个生命周期方法用于组件更新，调用顺序如下：

- componentWillReceiveProps()
- shouldComponentUpdate()
- componentWillUpdate()
- render()
- componentDidUpdate()

最后但同样重要的，组件卸载也有生命周期。它只有一个生命周期方法：`componentWillUnmount()`。

但是毕竟你不用一开始就了解所有生命周期方法。这样可能吓到你，而你也并不会用到所有的方法。即使在一个很大的 React 应用当中，除了 `constructor()` 和 `render()` 比较常用外，你只会用到一小部分生命周期函数。即使这样，了解每个生命周期方法的适用场景还是对你有帮助的：

- **constructor(props)** - 它在组件初始化时被调用。在这个方法中，你可以设置初始化状态以及绑定类方法。
- **componentWillMount()** - 它在 `render()` 方法之前被调用。这就是为什么它可以用作去设置组件内部的状态，因为它不会触发组件的再次渲染。但一般来说，还是推荐在 `constructor()` 中去初始化状态。
- **render()** - 这个生命周期方法是必须有的，它返回作为组件输出的元素。这个方法应该是一个纯函数，因此不应该在这个方法中修改组件的状态。它把属性和状态作为输入并且返回（需要渲染的）元素
- **componentDidMount()** - 它仅在组件挂载后执行一次。这是发起异步请求去 API 获取数据的绝佳时期。获取到的数据将被保存在内部组件的状态中然后在 `render()` 生命周期方法中展示出来。
- **componentWillReceiveProps(nextProps)** - 这个方法在一个更新生命周(update lifecycle)中被调用。新的属性会作为它的输入。因此你可以利用 `this.props` 来对比之后的属性和之前的属性，基于对比的结果去实现不同的行为。此外，你可以基于新的属性来设置组件的状态。
- **shouldComponentUpdate(nextProps, nextState)** - 每次组件因为状态或者属性更改而更新时，它都会被调用。你将在成熟的 React 应用中使用它来进行性能优化。在一个更新生命周期中，组件及其子组件将根据该方法返回的布尔值来决定是否重新渲染。这样你可以阻止组件的渲染生命周期（render lifecycle）方法，避免不必要的渲染。
- **componentWillUpdate(nextProps, nextState)** - 这个方法是 render() 执行之前的最后一个方法。你已经拥有下一个属性和状态，它们可以在这个方法中任由你处置。你可以利用这个方法在渲染之前进行最后的准备。注意在这个生命周期方法中你不能再触发 `setState()`。如果你想基于新的属性计算状态，你必须利用 `componentWillReceiveProps()`。
- **componentDidUpdate(prevProps, prevState)** - 这个方法在 render() 之后立即调用。你可以用它当成操作 DOM 或者执行更多异步请求的机会。
- **componentWillUnmount()** - 它会在组件销毁之前被调用。你可以利用这个生命周期方法去执行任何清理任务。

之前你已经用过了 `constructor()` 和 `render()` 生命周期方法。对于 ES6 类组件来说他们是常用的生命周期方法。实际上 `render()` 是必须有的，否则它将不会返回一个组件实例。

还有另一个生命周期方法：`componentDidCatch(error, info)`。它在 React 16[81] 中引入，用来捕获组件的错误。举例来说，在你的应用中展示样本数据本来是没问题的。但是可能会有列表的本地状态被意外设置成 `null` 的情况发生（例如从外部 API 获取列表失败时，你把本地状态设置为空了）。然后它就不能像之前一样去过滤（filter）和映射（map）这个列表，

[81] https://www.robinwieruch.de/what-is-new-in-react-16/

因为它不是一个空列表（`[]`）而是 `null`。这时组件就会崩溃，然后整个应用就会挂掉。现在你可以用 `componentDidCatch()` 来捕获错误，将它存在本地的状态中，然后像用户展示一条信息，说明应用发生了错误。

练习：

- 阅读更多关于 React 生命周期函数[82]的内容。
- 阅读更多关于 React 中状态与生命周期函数的关系[83]的内容。
- 阅读更多关于组件错误处理[84]的内容。

[82] https://facebook.github.io/react/docs/react-component.html
[83] https://facebook.github.io/react/docs/state-and-lifecycle.html
[84] https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html

获取数据

现在你已经做好了从 Hacker News API 获取数据的准备。我们可以用上文所提到过的 `componentDidMount()` 生命周期方法来获取数据。你将使用 JavaScript 原生的 fetch API 来发起请求。

在开始之前，让我们设置好 URL 常量和默认参数，来将 API 请求分解成几步。

src/App.js

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

const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';

...
```

在 JavaScript ES6 中，你可以用模板字符串（template strings）[85]去连接字符串。你将用它来拼接最终的 API 访问地址。

Code Playground

```
// ES6
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${DEFAULT_QUERY}`;

// ES5
var url = PATH_BASE + PATH_SEARCH + '?' + PARAM_SEARCH + DEFAULT_QUERY;

console.log(url);
// output: https://hn.algolia.com/api/v1/search?query=redux
```

这样就可以保证以后你 URL 组合的灵活性。

让我们开始使用 API 请求，在这个请求中将用到上述的网址。整个数据获取的过程在下面代码中一次给出，但后面会对每一步做详细解释。

[85] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals

src/App.js

```
...

class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result: null,
      searchTerm: DEFAULT_QUERY,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  setSearchTopStories(result) {
    this.setState({ result });
  }

  fetchSearchTopStories(searchTerm) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}`)
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(e => e);
  }

  componentDidMount() {
    const { searchTerm } = this.state;
    this.fetchSearchTopStories(searchTerm);
  }

  ...
}
```

这段代码做了很多事。我想把它分成更小的代码段，但是那样又会让人很难去理解每段代码之间的关系。接下来我就来详细解释代码中的每一步。

首先，你可以移除样本列表了，因为你将从 Hacker News API 得到一个真实的列表。这些样本数据已经没用了。现在组件将一个空的列表结果以及一个默认的搜索词作为初始状态。这个默认搜索词也同样用在 Search 组件的输入字段和第一个 API 请求中。

其次，在组件挂载之后，它用了 `componentDidMount()` 生命周期方法去获取数据。在第一次获取数据时，使用的是本地状态中的默认搜索词。它将获取与“redux”相关的资讯，因为它是默认的参数。

再次，这里使用的是原生的 fetch API。JavaScript ES6 模板字符串允许组件利用 `searchTerm` 来组成 URL。该 URL 是原生 fetch API 函数的参数。返回的响应需要被转化成 JSON 格式的数据结构。这是在处理 JSON 数据结构时，原生的 fetch API 中的强制步骤。最后将处理后的响应赋值给组件内部状态中的结果。此外，我们用一段 catch 代码来处理出错的情况。如果在发起请求时出现错误，这个函数会进入到 catch 中而不是 then 中。在本书之后的章节中，将涵盖错误处理的内容。

最后但同样重要的是，不要忘记在构造函数中绑定你的组件方法。

现在你可以用获取的数据去代替样本数据了。然而，你必须注意一点，这个结果不仅仅是一个数据的列表。它也是一个复杂的对象，它包含了元数据信息以及一系列的 hits，在我们的应用里就是这些资讯[86]。你可以在 `render()` 方法中用 `console.log(this.state);` 将这些信息打印出来，以便有一个直观的认识。

在接下来的步骤中，你将把之前的得到的结果渲染出来。但我们不会什么都渲染，在刚开始没有拿到结果时，我们会返回空。一旦 API 请求成功，我们会将结果保存在状态里，然后 App 组件将用更新后的状态重新渲染。

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;

    if (!result) { return null; }

    return (
      <div className="page">
        ...
        <Table
          list={result.hits}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
```

[86] https://hn.algolia.com/api

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

让我们回顾一下在组将的整个生命周期中发生了什么。首先组件通过构造函数得到初始化，之后它将初始化的状态渲染出来。但是你阻止了组件的显示，因为此时本地状态中的结果为空。React 允许组件通过返回 null 来不渲染任何东西。接着 `componentDidMount()` 生命周期函数执行。在这个方法中你从 Hacker News API 中异步地拿到了数据。一旦数据到达，组件就通过 `setSearchTopStories()` 函数改变组件内部的状态。之后，由于状态的更新，`update` 的生命周期开始运行。组件再次执行 `render()` 方法，但这次组件的内部状态中的结果已经填充，不再是空了。因此组件将重新渲染 Table 组件的内容。

你使用了大多数浏览器支持的原生 fetch API 来执行对 API 的异步请求。*create-react-app* 中的配置保证了它被所有浏览器支持。你也可以使用第三方库来代替原生 fetch API，例如：superagent[87] 和 axios[88]。

让我们重回到你的应用，现在你应该可以看到资讯列表了。然而，现在应用中仍然存在两个 bug。第一，“Dismiss”按钮不工作。因为它还不能处理这个复杂的 result 对象。当我们点击“Dismiss”按钮时，它仍然在操作之前那个简单的 result 对象。第二，当这个列表显示出来之后，你再尝试搜索其他的东西时，它只会在客户端过滤已有的列表，即使初始化的资讯搜索是在服务器端进行的。我们期待的行为是：当我们使用 Search 组件时，从 API 拿到新的结果，而不是去过滤样本数据。不用担心，两个 bug 都将在之后的章节中得到修复。

练习

- 阅读更多关于 ES6 模板字符串[89]的内容。
- 阅读更多关于原生 fetch API[90]的内容。
- 阅读更多关于在 React 中获取数据[91]的内容。

[87] https://github.com/visionmedia/superagent
[88] https://github.com/mzabriskie/axios
[89] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals
[90] https://developer.mozilla.org/en/docs/Web/API/Fetch_API
[91] https://www.robinwieruch.de/react-fetching-data/

扩展操作符

“Dismiss”按钮之所以不工作，是因为 `onDismiss()` 方法不能处理复杂的 result 对象。它现在还只能处理一个本地状态中的简单列表。但是现在这个列表已经不再是简单的平铺列表了。现在，让我们去操作这个 result 对象而不是去操作列表。

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    ...
  });
}
```

那现在 `setState()` 中发生了什么呢？很遗憾，这个 result 是一个复杂的对象。资讯（hits）列表只是这个对象的众多属性之一。所以，当某一项资讯从 result 对象中移除时，只能更新资讯列表，其他的属性还是得保持原样。

解决方法之一是直接改变 result 对象中的 hits 字段。我将演示这个方法，但实际操作中我们一般不这样做。

Code Playground

```
// don`t do this
this.state.result.hits = updatedHits;
```

React 拥护不可变的数据结构。因此你不应该改变一个对象（或者直接改变状态）。更好的做法是基于现在拥有的资源来创建一个新的对象。这样就没有任何对象被改变了。这样做的好处是数据结构将保持不变，因为你总是返回一个新对象，而之前的对象保持不变。

因此你可以用 JavaScript ES6 中的 `Object.assign()` 函数来到达这样的目的。它把接收的第一个参数作为目标对象，后面的所有参数作为源对象。然后把所有的源对象合并到目标对象中。只要把目标对象设置成一个空对象，我们就得到了一个新的对象。这种做法是拥抱不变性的，因为没有任何源对象被改变。以下是代码实现：

Code Playground

```
const updatedHits = { hits: updatedHits };
const updatedResult = Object.assign({}, this.state.result, updatedHits);
```

当遇到相同的属性时，排在后面的对象会覆写先前对象的该属性。现在让我们用它来改写 `onDismiss()` 方法：

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    result: Object.assign({}, this.state.result, { hits: updatedHits })
  });
}
```

这已经是一个解决方案了。但是在 JavaScript ES6 以及之后的 JavaScript 版本中还有一个更简单的方法。现在我将向你介绍扩展操作符。它只由三个点组成：...。当使用它时，数组或对象中的每一个值都会被拷贝到一个新的数组或对象。

让我们先来看一下 ES6 中数组的扩展运算符，虽然你现在还用不到它。

Code Playground

```
const userList = ['Robin', 'Andrew', 'Dan'];
const additionalUser = 'Jordan';
const allUsers = [ ...userList, additionalUser ];

console.log(allUsers);
// output: ['Robin', 'Andrew', 'Dan', 'Jordan']
```

这里 `allUsers` 是一个全新的数组变量，而变量 `userList` 和 `additionalUser` 还是和原来一样。用这个运算符，你甚至可以合并两个数组到一个新的数组中。

Code Playground

```
const oldUsers = ['Robin', 'Andrew'];
const newUsers = ['Dan', 'Jordan'];
const allUsers = [ ...oldUsers, ...newUsers ];

console.log(allUsers);
// output: ['Robin', 'Andrew', 'Dan', 'Jordan']
```

现在让我们来看看对象的扩展运算符。它并不是 JavaScript ES6 中的用法。它是针对下一个 JavaScript 版本的提出的[92]，然而它已经在 React 社区开始使用了。这就是为什么需要在 *create-react-app* 配置中加入了这个功能。

本质上来说，对象的扩展运算符和数组的扩展运算符是一样的，只是用在了对象上。

[92] https://github.com/sebmarkbage/ecmascript-rest-spread

Code Playground

```
const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const age = 28;
const user = { ...userNames, age };

console.log(user);
// output: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }
```

类似于之前数组的例子，以下是扩展多个对象的例子。

Code Playground

```
const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const userAge = { age: 28 };
const user = { ...userNames, ...userAge };

console.log(user);
// output: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }
```

最终，它可以用来代替 `Object.assign()`。

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    result: { ...this.state.result, hits: updatedHits }
  });
}
```

现在 “Dismiss” 按钮可以再次工作了，因为 `onDismiss()` 方法已经能够处理这个复杂的 result 对象了，并且知道当要忽略掉列表中的某一项时怎么去更新列表了。

练习

- 阅读更多 ES6 Object.assign()[93]的内容。
- 阅读更多 ES6 数组的扩展操作符[94]的内容。
  - 对象的扩展操作符在其中也有简单提到

[93] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign
[94] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator

条件渲染

React 应用很早就引入了条件渲染。但本书还没有提到过，因为目前为止还没有合适的用例。条件渲染用于你需要决定渲染哪个元素时。有些时候也可以是渲染一个元素或者什么都不渲染。其实最简单的条件渲染，只需要用 JSX 中的 if-else 就可以实现。

组件内部状态中的 `result` 对象的初始值为空。当 API 的结果还没返回时，此时的 App 组件没有返回任何元素。这已经是一个条件渲染了，因为在某个特定条件下，render() 方法提前返回了。根据条件，App 组件渲染它的元素或者什么都不渲染。

现在，让我们更进一步。因为只有 Table 组件的渲染依赖于 `result`，所以将它包在一个独立的条件渲染中才比较合理。即使 `result` 为空，其它的所有组件还是应该被渲染。你只需要在 JSX 中加上一个三元运算符就可以达到这样的目的。

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
          >
            Search
          </Search>
        </div>
        { result
          ? <Table
            list={result.hits}
            pattern={searchTerm}
            onDismiss={this.onDismiss}
          />
          : null
        }
      </div>
    );
  }
}
```

这是实现条件渲染的第二种方式。第三种则是运用 && 逻辑运算符。在 JavaScript 中，true && 'Hello World' 的值永远是“Hello World”。而 false && 'Hello World' 的值则永远是 false。

Code Playground

```
const result = true && 'Hello World';
console.log(result);
// output: Hello World

const result = false && 'Hello World';
console.log(result);
// output: false
```

在 React 中你也可以利用这个运算符。如果条件判断为 true，&& 操作符后面的表达式的值将会被输出。如果条件判断为 false，React 将会忽略并跳过后面的表达式。这个操作符可以用来实现 Table 组件的条件渲染，因为它返回一个 Table 组件或者什么都不返回。

src/App.js

```
{ result &&
  <Table
    list={result.hits}
    pattern={searchTerm}
    onDismiss={this.onDismiss}
  />
}
```

这是 React 中使用条件渲染的一些方式。你可以在条件渲染代码大全[95]中找到更多的选择，了解不同的条件渲染方式和它们的适用场景。

现在，你应该能够在你的应用中看到获取的数据。并且当数据正在获取时，你也可以看到除了 Table 组件以外的所有东西。一旦请求完成并且数据存入本地状态之后，Table 组件也将被渲染出来。因为 render() 方法再次执行，而且这时条件渲染判定为展示 Table 组件。

练习

- 阅读更多关于 React 条件渲染[96]的内容。
- 阅读更多关于实现条件渲染的不同方法[97]的内容。

[95] https://www.robinwieruch.de/conditional-rendering-react/
[96] https://facebook.github.io/react/docs/conditional-rendering.html
[97] https://www.robinwieruch.de/conditional-rendering-react/

客户端或服务端搜索

目前当你使用 Search 组件的输入栏时，你会在客户端过滤这个列表。所以你现在要做的是使用 Hacker News API 在服务器端来进行搜索。否则，你将只能处理第一次从 `componentDidMount()` 拿到的默认搜索词的 API 响应。

你可以在 App 组件中定义一个 `onSearchSubmit()` 方法。当 Search 组件执行搜索时，可以用这个方法来从 Hacker News API 获取结果。这与 `componentDidMount()` 生命周期方法中的获取数据的方式相同，但是这次搜索的内容变了，不用初始设定里的默认搜索词了。

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result: null,
      searchTerm: DEFAULT_QUERY,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...

  onSearchSubmit() {
    const { searchTerm } = this.state;
    this.fetchSearchTopStories(searchTerm);
  }

  ...
}
```

现在 Search 组件需要增加一个新的按钮了。这个按钮需要显示地触发搜索请求。否则每次当你改变输入框中的值时，你就会向 Hacker News API 发起请求。但你想要的是用一个明确的 `onClick()` 处理器来帮你控制它。

你确实可以通过某种方式（延迟）来去除 onChange() 的抖动，从而省去这个按钮。但是这样也会增加复杂度而且可能得不偿失。我们现在就用简单的方式来做就好了。

首先，把 onSearchSubmit() 方法传给 Search 组件。

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
            onSubmit={this.onSearchSubmit}
          >
            Search
          </Search>
        </div>
        { result &&
          <Table
            list={result.hits}
            pattern={searchTerm}
            onDismiss={this.onDismiss}
          />
        }
      </div>
    );
  }
}
```

随后，在你的 Search 组件中加一个按钮。将这个按钮设置为 type="submit"，并通过表单（form）的 onSubmit 属性去传递 onSubmit() 方法。你可以复用 children 属性，但这里它会被用作按钮的显示内容。

src/App.js

```
const Search = ({
  value,
  onChange,
  onSubmit,
  children
}) =>
  <form onSubmit={onSubmit}>
    <input
      type="text"
      value={value}
      onChange={onChange}
    />
    <button type="submit">
      {children}
    </button>
  </form>
```

在 Table 组件中，你可以移除过滤功能了，因为已经不会在客户端进行过滤（搜索）了。同时别忘记移除 `isSearched()` 函数。它也不会使用了。现在，当你点击“Search”按钮时，搜索结果将直接从 Hacker News API 中得到。

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        ...
        { result &&
          <Table
            list={result.hits}
            onDismiss={this.onDismiss}
          />
        }
      </div>
    );
  }
```

```
}

...

const Table = ({ list, onDismiss }) =>
  <div className="table">
    {list.map(item =>
      ...
    )}
  </div>
```

现在当你尝试去搜索时，你会注意到浏览器重新加载了。这是提交 HTML 表单后的浏览器原生行为。在 React 中，你会经常遇到用 `preventDefault()` 事件方法来阻止类似于这样的浏览器原生行为。

src/App.js

```
onSearchSubmit(event) {
  const { searchTerm } = this.state;
  this.fetchSearchTopStories(searchTerm);
  event.preventDefault();
}
```

现在你已经能搜索不同的资讯了。非常棒，这说明你在和一个真正 API 打交道，这样也就不再需要在客户端进行搜索了。

练习

- 阅读更多关于 React 中的合成事件[98]的内容。
- 试一试 Hacker News API[99]

[98] https://facebook.github.io/react/docs/events.html
[99] https://hn.algolia.com/api

分页抓取

你仔细看过返回回来的数据结构吗？Hacker News API[100] 返回的不仅仅只有资讯（hits）列表。确切地说它返回的是一个分页列表。利用分页属性（在第一个响应中为 `0`），将具有相同搜索词的下一页传给 API，你就可以获取更多分页的子列表了。

首先，让我们改造一下可组合的 API 常量，以便处理分页数据。

src/App.js

```
const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';
```

现在你可以使用新常量将分页参数添加到你的 API 请求中。

Code Playground

```
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAG\
E}`;

console.log(url);
// output: https://hn.algolia.com/api/v1/search?query=redux&page=
```

`fetchSearchTopStories()` 函数接收分页作为第二个参数。如果你不提供第二个参数，它将使用 `0` 作为初始参数并发起请求。因此 `componentDidMount()` 和 `onSearchSubmit()` 方法在第一个请求中默认获取第一页。之后的请求将根据提供的第二个参数抓取下一个页面的数据。

src/App.js

```
class App extends Component {

  ...

  fetchSearchTopStories(searchTerm, page = 0) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}\
${page}`)
      .then(response => response.json())
```

[100] https://hn.algolia.com/api

```
      .then(result => this.setSearchTopStories(result))
      .catch(e => e);
  }

  ...

}
```

现在你可以使用在 fetchSearchTopStories() 中 API 返回中的当前页。你也可以通过 onClick 点击事件来使用这个方法，以便抓取更多的资讯。现在让我们来实现通过按钮从 Hacker News API 中获取更多的分页数据的功能。你只需要定义 onClick() 事件处理器，这个处理器以当前的搜索词和下一页的页码作为参数（当前页码 + 1）。

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    const page = (result && result.page) || 0;
    return (
      <div className="page">
        <div className="interactions">
        ...
        { result &&
          <Table
            list={result.hits}
            onDismiss={this.onDismiss}
          />
        }
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopStories(searchTerm, page + 1\
)}>
            More
          </Button>
        </div>
      </div>
    );
  }
}
```

此外，当结果还没有返回时，你应该保证 render() 方法中的默认分页为 0。记住 render() 方法是在 componentDidMount() 生命周期方法去异步获取数据之前调用的。

这里还遗漏了一步。你抓取了下一个分页的数据，但新数据会覆盖你之前的分页数据。理想的情况下，result 对象中新的列表和本地状态中老的列表应该合并起来才对。现在让我们来实现将新的数据添加到老的数据上而不是去覆盖它。

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  const oldHits = page !== 0
    ? this.state.result.hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    result: { hits: updatedHits, page }
  });
}
```

现在在 setSearchTopStories() 方法中做了以下一些操作。首先，你从 result 对象中拿到 hits 字段和 page 字段。

第二，你必须检查老的 hits 字段是否存在。当页码为0时，这应该是一个来自 componentDidMount() 或者 onSearchSubmit() 方法的新的搜索请求；所以 hits 是空的。但是当你通过点击“More”按钮去抓取更多的分页数据时，页码不为0；此时它是下一页。老的 hits 已经储存在状态中等待着与新的分页合并。

第三，你不想覆盖老的 hits。你可以合并老的 hits 及 API 返回的新的 hits。这两个列表的合并可以通过 JavaScript ES6 数据扩展操作符完成。

第四，将合并后的 hits 和页码设置到本地组件的状态中。

你还可以做一个最后的调整。当你尝试点击“More”按钮时，它只抓取一定数量的资讯。但在每个请求的中，你可以通过设置 API URL 来获取更多的资讯。同样地，你还可以添加更多的可组合路径常量。

src/App.js

```
const DEFAULT_QUERY = 'redux';
const DEFAULT_HPP = '100';

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

现在你可以使用这些常量来扩展 API URL 了。

src/App.js

```
fetchSearchTopStories(searchTerm, page = 0) {
  fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${\
page}&${PARAM_HPP}${DEFAULT_HPP}`)
    .then(response => response.json())
    .then(result => this.setSearchTopStories(result))
    .catch(e => e);
}
```

现在，我们能够在一个请求中从 Hacker News API 获取更多的数据了。如你所见，功能强大的 Hacker News API 为你提供了大量的方法，以便让你用真实数据做练习。在学习新的东西时，你应该利用真实的 API 来让整个过程更加有趣。这就是我如何利用 API 提供的便利[101]来学习新的编程语言或库的。

练习

- 体验 Hacker News API 变量[102]

[101] https://www.robinwieruch.de/what-is-an-api-javascript/
[102] https://hn.algolia.com/api

客户端缓存

每次提交表单都会发起一个对 Hacker News API 的请求。你可能先搜索了“redux”，然后搜索了“react”，最后再次搜索了“redux”。这样它总共发起了3次请求。但是你搜索了“redux”两次并且每次都会执行一次异步操作去获取数据。如果有客户端的缓存，它将保存每次搜索的结果。当需要请求 API 的时候，它首先检查这个请求的结果是否已经在缓存中。如果在，那就使用缓存数据。否则再发 API 请求去获取数据。

为了实现在客户端对搜索结果的缓存，你必须在你的内部组件的状态中存储多个结果（`results`）而不是一个结果（`result`）。这些结果对象将会与搜索词映射成一个键值对。而每一个从 API 得到的结果会以搜索词为键（key）保存下来。

此时，在本地状态中，你的 result 看起应该是这样：

Code Playground

```
result: {
  hits: [ ... ],
  page: 2,
}
```

假设你已经发起了两次 API 请求。一次搜索“redux”，另一次搜索“react”。那你的 results 对象看起来应该是这样：

Code Playground

```
results: {
  redux: {
    hits: [ ... ],
    page: 2,
  },
  react: {
    hits: [ ... ],
    page: 1,
  },
  ...
}
```

让我们用 React 的 `setState()` 方法来实现客户端缓存。首先，在初始化组件状态中重命名 `result` 对象为 `results`。其次，定义一个临时的 `searchKey` 用来储存单个 `result`。

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
    };

    ...

  }

  ...

}
```

searchKey 的值必须在发起请求之前设置。它的值来自 searchTerm。你可能会想：为什么我们不直接使用 searchTerm 呢？这是在我们继续之前需要理解的重点。searchTerm 是一个动态的变量，因此它随输入的关键字变化而变化。然而，这里你需要的是一个稳定的变量。它保存最近一次提交给 API 的搜索词，也可以用它来检索结果集中的某个结果。由于它指向缓存中的当前返回结果，因此还可以在 render() 方法中用来显示当前结果。

src/App.js

```
componentDidMount() {
  const { searchTerm } = this.state;
  this.setState({ searchKey: searchTerm });
  this.fetchSearchTopStories(searchTerm);
}

onSearchSubmit(event) {
  const { searchTerm } = this.state;
  this.setState({ searchKey: searchTerm });
  this.fetchSearchTopStories(searchTerm);
  event.preventDefault();
}
```

现在，你必须去调整一下内部组件状态中储存结果的位置。它应该通过 searchKey 来存储每个结果。

src/App.js

```
class App extends Component {

  ...

  setSearchTopStories(result) {
    const { hits, page } = result;
    const { searchKey, results } = this.state;

    const oldHits = results && results[searchKey]
      ? results[searchKey].hits
      : [];

    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      }
    });
  }

  ...

}
```

在 `results` 集中，`searchKey` 用作键名（key），其值用来保存更新后的 hits 和 page。

首先，你必须从组件状态中检索 `searchKey`。记住 `searchKey` 是在 `componentDidMount()` 和 `onSearchSubmit()` 中设置的。

第二，和之前一样，老的 hits 需要合并到新的 hits 中。但是这次老的 hits 是以 `searchKey` 为键名从 `results` 集中找到的。

第三，在状态里，一个新的 `result` 可以设置在 `results` 集中。让我们来看看 `setState()` 中的 `results` 对象是什么样子。

src/App.js

```
results: {
  ...results,
  [searchKey]: { hits: updatedHits, page }
}
```

下半部分的代码是为了保证，通过 searchKey 将更新后的 result 对象保存在 results 集中。它包含 hits 和 page 属性的对象。而 searchKey 的值就是搜索词。现在你学会了 [searchKey]: ... 这样的语法。这个语法中，ES6 是通过计算得到属性名的。它可以帮助你实现动态分配对象的值。

上半部分的代码则是用对象扩展运算符将所有其它包含在 results 集中的 searchKey 展开。否则，你将会失去之前所有储存过的 results。

现在你以搜索词为键名，将所有结果储存了起来。这是实现缓存的第一步。接下来，你可以根据稳定的 searchKey 从 results 集中检索 result。这就是为什么一开始你必须引进 searchKey 作为一个稳定的变量。不然用动态的 searchTerm 去检索当前的 result 时，这个检索过程会崩溃，因为它的值可能在你使用 Search 组件时改变过了。

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
      results[searchKey].hits
    ) || [];
```

```
    return (
      <div className="page">
        <div className="interactions">
          ...
        </div>
        <Table
          list={list}
          onDismiss={this.onDismiss}
        />
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopStories(searchKey, page + 1)\
}>
            More
          </Button>
        </div>
      </div>
    );
  }
}
```

当 searchKey 没有对应的结果时，你默认得到一个空列表，所以现在可以节省 Table 组件的条件渲染了。此外，你需要传 searchKey 给“More”按钮来代替 searchTerm。否则，你抓取分页的搜索词将是 searchTerm 这个可能变化的值。另外，确保“Search”组件的输入字段用的是动态的 searchTerm。

这样，搜索功能就可以再次工作了。它会保存所有从 Hacker News API 返回的结果。

此外，onDismiss() 方法也需要优化。它仍还在处理 result 对象。现在它必须处理 results 了。

src/App.js

```
  onDismiss(id) {
    const { searchKey, results } = this.state;
    const { hits, page } = results[searchKey];

    const isNotId = item => item.objectID !== id;
    const updatedHits = hits.filter(isNotId);

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
```

```
      }
    });
  }
```

这样“Dismiss”按钮就可以再次工作了。

然而，现在我们还不能阻止应用对每一次搜索都发起一个 API 请求。即使我们保存了某一个结果，但也还没有任何阻止重复请求的检查。也就是说缓存功能仍不完整。虽然应用缓存了所有结果，但它还没有将这些结果利用起来。所有我们的最后一步就是：如果搜索的结果已经存在于缓存中，就阻止 API 请求。

src/App.js

```
class App extends Component {

  constructor(props) {

    ...

    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  needsToSearchTopStories(searchTerm) {
    return !this.state.results[searchTerm];
  }

  ...

  onSearchSubmit(event) {
    const { searchTerm } = this.state;
    this.setState({ searchKey: searchTerm });

    if (this.needsToSearchTopStories(searchTerm)) {
      this.fetchSearchTopStories(searchTerm);
    }

    event.preventDefault();
  }
```

```
    ...
}
```

现在就算你重复搜索一个词，客户端也只会发起一次请求。以这种方式进行缓存的话分页的数据也会保存下来，因为结果的每一页都将保存在 `results` 集中。这是不是一个很强大的方法来引入缓存呢？而且 Hacker News API 提供了你所需的一切，甚至可以高效地缓存分页数据。

错误处理

你与 Hacker News API 交互的所有功能都已就位。你甚至引入了一种优雅的方式来缓存 API 结果，而且通过分页列表功能还可以从 API 中获取无尽的资讯子列表。但是我们还忽略了一项。但我们忘了一件事情，不幸的是，它在日常开发中经常被忽略：错误处理。人们常常容易沉浸于主逻辑的开发中，却忘记错误也会随之而来。

在本章中，引入了一个高效的方法来为你的应用添加一个当发生错误的 API 请求时的错误处理。其实你已经掌握了在 React 中处理错误的基础知识，也就是本地状态和条件渲染。本质上来讲，错误只是 React 的另一种状态。当一个错误发生时，你先将它存在本地状态中，而后利用条件渲染在组件中显示错误信息。听起来是不是很简单。现在让我们在 App 组件中实现它，因为它是向 Hacker News API 发起请求的组件。首先，你要在本地状态中引入 error 这个状态。它初始化为 null，但当错误发生时它会被置成一个 error 对象。

src/App.js

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
    };

    ...
  }

...

}
```

第二，通过结合使用 catch 和 `setState()`，你可以捕获错误对象并将它存在本地状态中。如果 API 请求失败，catch 就会执行。

src/App.js

```
class App extends Component {

  ...

  fetchSearchTopStories(searchTerm, page = 0) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}\
${page}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(e => this.setState({ error: e }));
  }

  ...

}
```

第三，如果错误发生了，你可以在 `render()` 方法中在本地状态里获取到 error 对象，然后利用条件渲染来显示一个错误信息。

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error
    } = this.state;

    ...

    if (error) {
      return <p>Something went wrong.</p>;
    }

    return (
      <div className="page">
```

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

就这么简单。如果你想测试错误处理是否工作，你可以把 API URL 换成一个不存在的 URL。

src/App.js

```
const PATH_BASE = 'https://hn.foo.bar.com/api/v1';
```

之后，你应该得到一个错误信息而不是应用界面。你可以自己决定错误信息渲染的位置。在这种情况下，整个 app 不再显示。显然这不是最佳用户体验。我们将 Table 组件和错误信息择一渲染如何？这样的话当错误发生时，除了 Table 组件，应用其余的部分仍然可见。

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
      results[searchKey].hits
    ) || [];
```

```
    return (
      <div className="page">
        <div className="interactions">
          ...
        </div>
        { error
          ? <div className="interactions">
            <p>Something went wrong.</p>
          </div>
          : <Table
            list={list}
            onDismiss={this.onDismiss}
          />
        }
        ...
      </div>
    );
  }
}
```

最后，别忘了把 URL 还原成一个真实的 URL。

src/App.js

```
const PATH_BASE = 'https://hn.algolia.com/api/v1';
```

你的应用应该仍然可以工作，不过此时如果 API 请求失败应用就有错误处理了。

练习

- 阅读更多关于 React 组件的错误处理[103]的内容。

[103] https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html

你已经学会使用 React 去与 API 交互了！现在让我们回顾一下本章内容：

- React
  - 针对不同用例的 ES6 类组件生命周期方法
  - componentDidMount() 如何用于 API 交互
  - 条件渲染
  - 表单上的合成事件
  - 错误处理
- ES6
  - 用模板字符串去组合字符串
  - 扩展运算符用于不可变数据结构
  - 可计算的属性名称
- General
  - Hacker News API 交互
  - 浏览器原生 fetch API
  - 客户端和服务器端搜索
  - 数据分页
  - 客户端缓存

同样地，此时小憩一下，将学过的知识点消化理解并应用是有必要的。你还可以在之前写过的代码上做做实验，小试牛刀。

你可以在官方代码库[104]中找到源码。

[104] https://github.com/rwieruch/hackernews-client/tree/4.3

代码组织和测试

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

ES6模块：Import 和 Export

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

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

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

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

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

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

你可以导出一个或者多个变量。这称为一个命名的导出。

Code Playground: file1.js

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

export { firstname, lastname };
```

并在另外一个文件用相对第一文件的相对路径导入。

Code Playground: file2.js

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

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

你也可以用对象的方式导入另外文件的全部变量。

Code Playground: file2.js

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

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

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

Code Playground: file2.js

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

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

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

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

Code Playground: file1.js

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

export default robin;
```

你可以在导入 default 输出时省略花括号。

Code Playground: file2.js

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

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

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

Code Playground: file1.js

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

const person = {
  firstname,
  lastname,
};

export {
  firstname,
  lastname,
};

export default person;
```

Code Playground: file2.js

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

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

在命名的导出中，你可以省略多余行直接导出变量。

Code Playground: file1.js

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

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

练习

- 阅读 ES6 import[105]
- 阅读 ES6 export[106]

[105] https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/import
[106] https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/export

代码组织与 ES6 模块

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

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

一种可能的模块结构类似：

Folder Structure

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

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

Folder Structure

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

```
    index.js
    test.js
    index.css
  Table/
    index.js
    test.js
    index.css
  Search/
    index.js
    test.js
    index.css
```

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

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

Folder Structure

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

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

Code Playground: src/constants/index.js

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

App/index.js 文件可以导入这些变量，以便使用。

Code Playground: src/components/App/index.js

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

...
```

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

Code Playground: src/components/App/index.js

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

...
```

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

Folder Structure

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

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

Code Playground: src/Buttons/index.js

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

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

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

Code Playground: src/App/index.js

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

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

Code Playground: src/App/index.js

```
// 糟糕的实践，不要这样做

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

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

练习

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

快照测试和 Jest

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

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

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

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

Jest[107] 是一个在 Facebook 使用的测试框架。在 React 社区，它被用来做 React 的组件测试。幸好 *create-react-app* 已经包含了 Jest，所以你不需要担心启动配置的问题。

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

src/App.js

```
...

class App extends Component {
  ...
}

...

export default App;

export {
  Button,
  Search,
```

[107] https://facebook.github.io/jest/

```
  Table,
};
```

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

src/App.test.js

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

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

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

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

Command Line

```
npm test
```

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

- 在命令行安装：`npm install isomorphic-fetch`
- 在你的 *App.js* 引入：`import fetch from 'isomorphic-fetch'`

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

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

当你写快照之前，可能需要安装一个工具库。

Command Line

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

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

src/App.test.js

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

describe('App', () => {

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

});
```

现在你可以使用“test”块来实现第一个快照测试了。

src/App.test.js

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

describe('App', () => {

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

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <App />
```

```
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

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

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

我们来给我们的组件添加更多的测试，第一步，Search 组件：

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App, { Search } from './App';

...

describe('Search', () => {

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

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

});
```

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

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

src/App.test.js

```
...
import App, { Search, Button } from './App';

...

describe('Button', () => {

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

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

});
```

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

src/App.test.js

```
...
import App, { Search, Button, Table } from './App';

...

describe('Table', () => {

  const props = {
    list: [
```

```
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
  };

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

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

});
```

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

练习

- 当组件 render() 方法的返回值有改变时，请留意测试会如何失败的?
  - 接受或者拒绝一个快照变更。
- 在后面章节中，当组件实现有改变时，保持你的快照最新。
- 读一下官方文档 React 中的 Jest[108]。

单元测试和 Enzyme

Enzyme[109] 是一个由 Airbnb 维护的测试工具，可以用来断言、操作、遍历 React 组件。你可以用它来管理单元测试，在 React 测试中与快照测试互补。

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

[108] https://facebook.github.io/jest/docs/tutorial-react.html
[109] https://github.com/airbnb/enzyme

```
npm install --save-dev enzyme react-addons-test-utils enzyme-adapter-react-16
```

第二步，你需要在测试启动配置中引入，并为 React 初始化这个适配器。

src/App.test.js

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

Enzyme.configure({ adapter: new Adapter() });
```

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

src/App.test.js

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

...

describe('Table', () => {

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

  ...

  it('shows two items in list', () => {
    const element = shallow(
```

```
      <Table { ...props } />
    );

    expect(element.find('.table-row').length).toBe(2);
  });

});
```

浅渲染组件不会渲染它的子组件。这样的话，你可以让测试只对一个组件负责。

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

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

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

练习：

- 使用 Enzyme 对你的 Button 组件写一个单元测试
- 在接下来的章节中，保持你的单元测试的更新
- 了解更多 Enzyme 和它的渲染 API[110]

[110] https://github.com/airbnb/enzyme

组件接口和 PropTypes

你可能知道 TypeScript[111] 或者 Flow[112] 在 JavaScript 中引入了类型接口。一个类型语言更不容易出错，因为代码会根据它的程序文本进行验证。编辑器或者其他工具可以在程序运行之前就捕获这些错误，可以让你的应用更健壮。

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

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

第一步，你需要为 React 额外安装一个库。

Command Line

```
npm install prop-types
```

现在你可以导入 PropTypes。

src/App.js

```
import PropTypes from 'prop-types';
```

我们开始为组件添加一个 props 接口：

src/App.js

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

Button.propTypes = {
  onClick: PropTypes.func,
```

[111] https://www.typescriptlang.org/
[112] https://flowtype.org/

```
  className: PropTypes.string,
  children: PropTypes.node,
};
```

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

- PropTypes.array
- PropTypes.bool
- PropTypes.func
- PropTypes.number
- PropTypes.object
- PropTypes.string

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

- PropTypes.node
- PropTypes.element

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

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

src/App.js

```
Button.propTypes = {
  onClick: PropTypes.func.isRequired,
  className: PropTypes.string,
  children: PropTypes.node.isRequired,
};
```

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

```
Table.propTypes = {
  list: PropTypes.array.isRequired,
  onDismiss: PropTypes.func.isRequired,
};
```

你可以将数组 PropTypes 的元素定义的更加明确：

src/App.js

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

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

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

src/App.js

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

你可以将它替换为 React 默认的 prop：

src/App.js

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

Button.defaultProps = {
  className: '',
};
```

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

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

练习：

- 为 Search 组件定义 PropTypes 接口
- 在接下来的章节中，确保 PropTypes 接口一直被更新
- 了解更多 React PropTypes[113]

[113] https://facebook.github.io/react/docs/typechecking-with-proptypes.html

你已经学习到了如何组织和测试你的代码。让我们来回顾一下这最后几章吧：

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

你可以在官方代码库[114]中找到源码。

[114] https://github.com/rwieruch/hackernews-client/tree/4.4

高级 React 组件

本章将重点介绍高级 React 组件的实现。我们将了解什么是高阶组件以及如何实现它们。此外，我们还将深入探讨 React 中更高级的主题，并用它实现复杂的交互功能。

引用 DOM 元素

有时我们需要在 React 中与 DOM 节点进行交互。`ref` 属性可以让我们访问元素中的一个节点。通常，访问 DOM 节点是 React 中的一种反模式，因为我们应该遵循它的声明式编程和单向数据流。当我们引入第一个搜索输入组件时，就已经了解这些了。但是在某些情况下，我们仍然需要访问 DOM 节点。官方文档提到了三种情况：

- 使用 DOM API（focus 事件，媒体播放等）
- 调用命令式 DOM 节点动画
- 与需要 DOM 节点的第三方库集成（例如 D3.JavaScript[115]）

让我们通过 Search 组件这个例子看一下。当应用程序第一次渲染时，input 字段应该被聚焦。这是需要访问 DOM API 的一种用例。本章将展示渲染时聚焦 input 字段是如何工作的，但由于这个功能对于应用程序并不是很有用，所以我们将在本章之后省略这些更改。尽管如此，你仍然可以为自己的应用程序保留它。

通常，无状态组件和 ES6 类组件中都可以使用 `ref` 属性。在聚焦 input 字段的用例中，我们就需要一个生命周期方法。这就是为什么接下来会先在 ES6 类组件中展示如何使用 `ref` 属性。

第一步是将无状态组件重构为 ES6 类组件。

src/App.js

```
class Search extends Component {
  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
        />
        <button type="submit">
          {children}
```

[115] https://d3js.org/

```
        </button>
      </form>
    );
  }
}
```

ES6 类组件的 `this` 对象可以帮助我们通过 `ref` 属性引用 DOM 节点。

src/App.js

```
class Search extends Component {
  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
          ref={(node) => { this.input = node; }}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

现在，你可以通过使用 this 对象、适当的生命周期方法和 DOM API 在组件挂载的时候来聚焦 input 字段。

src/App.js

```
class Search extends Component {
  componentDidMount() {
    if(this.input) {
      this.input.focus();
    }
  }

  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
          ref={(node) => { this.input = node; }}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

当应用程序渲染时，input 字段应该被聚焦。这就是 `ref` 属性的基本用法。

但是我们怎样在没有 `this` 对象的无状态组件中访问 `ref` 属性呢？接下来我们在无状态组件中演示。

src/App.js

```
const Search = ({
  value,
  onChange,
  onSubmit,
  children
}) => {
  let input;
  return (
    <form onSubmit={onSubmit}>
      <input
        type="text"
        value={value}
        onChange={onChange}
        ref={(node) => input = node}
      />
      <button type="submit">
        {children}
      </button>
    </form>
  );
}
```

现在我们能够访问 input DOM 元素。由于在无状态组件中，没有生命周期方法去触发聚焦事件，这个功能对于聚焦 input 字段这个用例而言没什么用。但是在将来，你可能会遇到其他一些合适的需要在无状态组件中使用 `ref` 属性的情况。

练习

- 阅读 React 中的 ref 属性概述[116]
- 阅读 在 React 中使用 ref 属性[117]

[116] https://facebook.github.io/react/docs/refs-and-the-dom.html
[117] https://www.robinwieruch.de/react-ref-attribute-dom-node/

加载……

现在让我们回到应用程序。当向 Hacker News API 发起搜索请求时，我们想要显示一个加载指示符。

请求是异步的，此时应该向用户展示某些事情即将发生的某种反馈。让我们在 `src / App.js` 中定义一个可重用的 Loading 组件。

src/App.js

```
const Loading = () =>
  <div>Loading ...</div>
```

现在我们需要存储加载状态 (isLoading)。根据加载状态 (isLoading)，决定是否显示 Loading 组件。

src/App.js

```
const Loading = () =>
  <div>Loading ...</div>
```

现在你将需要一个属性来存储加载状态。根据加载状态可以决定稍后是否显示所加载的组件。

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
      isLoading: false,
    };

    ...
  }

  ...

}
```

isLoading 的初始值是 false。在 App 组件挂载完成之前，无需加载任何东西。

当发起请求时，将加载状态 (isLoading) 设置为 true。最终，请求会成功，那时可以将加载状态 (isLoading) 设置为 false。

src/App.js

```
class App extends Component {

  ...

  setSearchTopStories(result) {
    ...

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      },
      isLoading: false
    });
  }

  fetchSearchTopStories(searchTerm, page = 0) {
    this.setState({ isLoading: true });

    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}\
${page}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(e => this.setState({ error: e }));
  }

  ...

}
```

最后一步，我们将在应用程序中使用 Loading 组件。基于加载状态 (isLoading) 的条件来决定渲染 Loading 组件或 Button 组件。后者为一个用于获取更多数据的按钮。

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <div className="interactions">
          { isLoading
            ? <Loading />
            : <Button
              onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}>
              More
            </Button>
          }
        </div>
      </div>
    );
  }
}
```

由于我们在 `componentDidMount()` 中发起请求，Loading 组件会在应用程序启动的时候显示。此时，因为列表是空的，所以不显示 Table 组件。当响应数据从 Hacker News API 返回时，返回的数据会通过 Table 组件显示出来，加载状态 (isLoading) 设置为 false，然后 Loading 组件消失。同时，出现了可以获取更多的数据的“More”按钮。一旦点击按钮，获取更多的数据，该按钮将消失，加载组件会重新出现。

练习：

- 使用第三方库，比如Font Awesome[118]，来显示加载图标，而不是“Loading ...”文本

[118] http://fontawesome.io/

高阶组件

高阶组件（HOC）是 React 中的一个高级概念。HOC 与高阶函数是等价的。它接受任何输入 - 多数时候是一个组件，也可以是可选参数 - 并返回一个组件作为输出。返回的组件是输入组件的增强版本，并且可以在 JSX 中使用。

HOC 可用于不同的情况，比如：准备属性，管理状态或更改组件的表示形式。其中一种情况是将 HOC 用于帮助实现条件渲染。想象一下现在有一个 List 组件，由于列表可以为空或无，那么它可以渲染一个列表或者什么也不渲染。当没有列表的时候，HOC 可以屏蔽掉这个不显示任何内容的列表。另一方面，这个简单的 List 组件不再需要关心列表存不存在，它只关心渲染列表。

我们接下来创建一个简单的 HOC，它将一个组件作为输入并返回一个组件。我们可以把它放在 `src / App.js` 文件中。

src/App.js

```
function withFoo(Component) {
  return function(props) {
    return <Component { ...props } />;
  }
}
```

有一个惯例是用“with”前缀来命名 HOC。由于我们现在使用的是 ES6，因此可以使用 ES6 箭头函数更简洁地表达 HOC。

src/App.js

```
const withFoo = (Component) => (props) =>
  <Component { ...props } />
```

在这个例子中，没有做任何改变，输入组件将和输出组件一样。它渲染与输入组件相同的实例，并将所有的属性 (props) 传递给输出组件，但是这个 HOC 没意义。我们来增强输出组件功能：当加载状态 (isLoading) 为 true 时，组件显示 Loading 组件，否则显示输入的组件。条件渲染是 HOC 的一种绝佳用例。

src/App.js

```
const withLoading = (Component) => (props) =>
  props.isLoading
    ? <Loading />
    : <Component { ...props } />
```

基于加载属性 (isLoading)，我们可以实现条件渲染。该函数将返回 Loading 组件或输入的组件。

一般来说，将对象展开然后作为一个组件的输入是非常高效的（比如说前面那个例子中的 props 对象）。请参阅下面的代码片段中的区别。

Code Playground

```
// before you would have to destructure the props before passing them
const { foo, bar } = props;
<SomeComponent foo={foo} bar={bar} />

// but you can use the object spread operator to pass all object properties
<SomeComponent { ...props } />
```

有一点应该避免。我们把包括 `isLoading` 属性在内的所有 props 通过展开对象传递给输入的组件。

然而，输入的组件可能不关心 `isLoading` 属性。我们可以使用 ES6 中的 rest 解构来避免它。

src/App.js

```
const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />
```

这段代码从 `props` 对象中取出一个属性，并保留剩下的属性。这也适用于多个属性。你可能已经在 解构赋值[119]中了解过它。

现在，我们以在 JSX 中使用 HOC。应用程序中的用例可能是显示“More”按钮或 Loading 组件。

Loading 组件已经封装在 HOC 中，缺失了输入组件。在显示 Button 组件或 Loading 组件的用例中，Button 是 HOC 的输入组件。增强的输出组件是一个 ButtonWithLoading 的组件。

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

src/App.js

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

const Loading = () =>
  <div>Loading ...</div>

const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />

const ButtonWithLoading = withLoading(Button);
```

现在所有的东西已经被定义好了。最后一步，就是使用 ButtonWithLoading 组件，它接收加载状态 (isLoading) 作为附加属性。当 HOC 消费加载属性 (isLoading) 时，再将所有其他 props 传递给 Button 组件。

src/App.js

```
class App extends Component {

  ...

  render() {
    ...
    return (
      <div className="page">
        ...
        <div className="interactions">
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}>
            More
          </ButtonWithLoading>
        </div>
```

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

当再次运行测试时，App 组件的快照测试会失败。执行 diff 在命令行可能显示如下：

Command Line

```
-     <button
-       className=""
-       onClick={[Function]}
-       type="button"
-     >
-       More
-     </button>
+     <div>
+       Loading ...
+     </div>
```

如果你认为是 App 组件有问题，现在可以选择修复该组件，或者选择接受 App 组件的新快照。因为本章介绍了 Loading 组件，我们可以在交互测试的命令行中接受已经更改的快照测试。

高阶组件是 React 中的高级技术。它可以使组件具有更高的重用性，更好的抽象性，更强的组合性，以及提升对 props，state 和视图的可操作性。如果不能马上理解，别担心。我们需要时间去熟悉它。

我们推荐阅读高阶组件的简单介绍[120]。这篇文章介绍了另一种学习高阶组件的方法，展示了如何用函数式的方式定义高阶组件并优雅的使用它，以及使用高阶组件解决条件渲染的问题。

练习：

- 阅读 高阶组件的简单介绍[121]
- 使用创建的高阶组件
- 思考一个适合使用高阶组件的场景
  - 如果想到了使用场景，请实现这个高阶组件

[120] https://www.robinwieruch.de/gentle-introduction-higher-order-components/
[121] https://www.robinwieruch.de/gentle-introduction-higher-order-components/

高级排序

我们已经实现了客户端和服务器端搜索交互。因为我们已经拥了 Table 组件，所以增强 Table 组件的交互性是有意义的。那接下来，我们为 Table 组件加入根据列标题进行排序的功能如何?

你自己写一个排序函数，但是一般这种情况，我个人更喜欢使用第三方工具库。lodash[122]就是这些工具库之一，当然你也可以选择适用于你的任何第三方库。让我们安装 lodash 并使用。

Command Line

```
npm install lodash
```

现在我们可以在 `src/App` 文件中导入 lodash 的 sort 方法。

src/App.js

```
import React, { Component } from 'react';
import fetch from 'isomorphic-fetch';
import { sortBy } from 'lodash';
import './App.css';
```

Table 组件中有好几列，分别是标题，作者，评论和评分。你可以定义排序函数，而每个函数接受一个列表并返回按照指定属性排序过的列表。此外，我们还需要一个默认的排序函数，该函数不做排序而只是用于返回未排序的列表。这将作为组件的初始状态。

src/App.js

```
...

const SORTS = {
  NONE: list => list,
  TITLE: list => sortBy(list, 'title'),
  AUTHOR: list => sortBy(list, 'author'),
  COMMENTS: list => sortBy(list, 'num_comments').reverse(),
  POINTS: list => sortBy(list, 'points').reverse(),
};

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

[122] https://lodash.com/

可以看到有两个排序函数返回一个反向列表。这是因为当用户首次点击排序的时候，希望查看评论和评分最高的项目，而不是最低的。

现在，SORTS 对象允许你引用任何排序函数。

我们的 App 组件负责存储排序函数的状态。组件的初始状态存储的是默认排序函数，它不对列表排序而只是将输入的 list 作为输出。

src/App.js

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  error: null,
  isLoading: false,
  sortKey: 'NONE',
};
```

一旦用户选择了一个不同的 `sortKey`，比如说 `AUTHOR`，App 组件将从 `SORTS` 对象中选取合适的排序函数对列表进行排序。

现在，我们要在 App 组件中定义一个新的类方法，用来将 `sortKey` 设置为 App 组件的状态。然后，`sortKey` 可以被用来选取对应的排序函数并对其列表进行排序。

src/App.js

```
class App extends Component {

  constructor(props) {

    ...

    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
    this.onSort = this.onSort.bind(this);
  }

  onSort(sortKey) {
    this.setState({ sortKey });
  }
```

```
  ...

}
```

下一步是将类方法和 `sortKey` 传递给 Table 组件。

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading,
      sortKey
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <Table
          list={list}
          sortKey={sortKey}
          onSort={this.onSort}
          onDismiss={this.onDismiss}
        />
        ...
      </div>
    );
  }
}
```

Table 组件负责对列表排序。它通过 `sortKey` 选取 `SORT` 对象中对应的排序函数，并列表作为该函数的输入。之后，Table 组件将在已排序的列表上继续 mapping。

src/App.js

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
  <div className="table">
    {SORTS[sortKey](list).map(item =>
      <div key={item.objectID} className="table-row">
        ...
      </div>
    )}
  </div>
```

理论上，列表可以按照其中的任意排序函数进行排序，但是默认的排序 (sortKey) 是 `NONE`，所以列表不进行排序。至此，还没有人执行 `onSort()` 方法来改变 `sortKey`。让我们接下来用一行列标题来扩展表格，每个列标题会使用列中的 Sort 组件对每列进行排序。

src/App.js

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
  <div className="table">
    <div className="table-header">
      <span style={{ width: '40%' }}>
        <Sort
          sortKey={'TITLE'}
          onSort={onSort}
        >
          Title
        </Sort>
      </span>
      <span style={{ width: '30%' }}>
        <Sort
          sortKey={'AUTHOR'}
          onSort={onSort}
        >
```

```
          Author
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        <Sort
          sortKey={'COMMENTS'}
          onSort={onSort}
        >
          Comments
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        <Sort
          sortKey={'POINTS'}
          onSort={onSort}
        >
          Points
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        Archive
      </span>
    </div>
    {SORTS[sortKey](list).map(item =>
      ...
    )}
  </div>
```

每个 Sort 组件都有一个指定的 `sortKey` 和通用的 `onSort()` 函数。Sort 组件调用 `onSort()` 方法去设置指定的 `sortKey`。

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button onClick={() => onSort(sortKey)}>
    {children}
  </Button>
```

如你所见，Sort 组件重用了我们的 Button 组件，当点击按钮时，每个传入的 `sortKey` 都会被 `onSort()` 方法设置。现在，我们应该能够通过点击列标题来对列表进行排序了。

这里有个改善外观的小建议。到目前为止，列标题中的按钮看起来有点傻。我们给 Sort 组件中的按钮添加一个合适的 `className`。

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button
    onClick={() => onSort(sortKey)}
    className="button-inline"
  >
    {children}
  </Button>
```

现在应该看起来不错。接下来的目标是实现反向排序。如果点击 Sort 组件两次，该列表应该被反向排序。首先，我们需要用一个布尔值来定义反向状态 (isSortReverse)。排序可以反向或不反向。

src/App.js

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  error: null,
  isLoading: false,
  sortKey: 'NONE',
  isSortReverse: false,
};
```

现在在排序方法中，可以评判列表是否被反向排序。如果状态中的 sortKey 与传入的 sortKey 相同，并且反向状态 (isSortReverse) 尚未设置为 true，则相反——反向状态 (isSortReverse) 设置为 true。

src/App.js

```
onSort(sortKey) {
  const isSortReverse = this.state.sortKey === sortKey && !this.state.isSortReve\
rse;
  this.setState({ sortKey, isSortReverse });
}
```

同样，将反向属性 (isSortReverse) 传递给 Table 组件。

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading,
      sortKey,
      isSortReverse
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <Table
          list={list}
          sortKey={sortKey}
          isSortReverse={isSortReverse}
          onSort={this.onSort}
          onDismiss={this.onDismiss}
        />
        ...
      </div>
    );
  }
}
```

现在 Table 组件有一个块体箭头函数用于计算数据。

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return(
    <div className="table">
      <div className="table-header">
        ...
      </div>
      {reverseSortedList.map(item =>
        ...
      )}
    </div>
  );
}
```

反向排序现在应该可以工作了。

最后值得一提，为了改善用户体验，我们可以思考一个开放性的问题：用户可以区分当前是根据哪一列进行排序的吗？目前为止，用户是区别不出来的。我们可以给用户一个视觉反馈。

每个 Sort 组件都已经有了其的特定 `sortKey`。它可以用来识别被激活的排序。我们可以将内部组件状态 `sortKey` 作为激活排序标识 (activeSortKey) 传递给 Sort 组件。

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return(
    <div className="table">
      <div className="table-header">
        <span style={{ width: '40%' }}>
          <Sort
            sortKey={'TITLE'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Title
          </Sort>
        </span>
        <span style={{ width: '30%' }}>
          <Sort
            sortKey={'AUTHOR'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Author
          </Sort>
        </span>
        <span style={{ width: '10%' }}>
          <Sort
            sortKey={'COMMENTS'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Comments
          </Sort>
```

```
        </span>
        <span style={{ width: '10%' }}>
          <Sort
            sortKey={'POINTS'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Points
          </Sort>
        </span>
        <span style={{ width: '10%' }}>
          Archive
        </span>
      </div>
      {reverseSortedList.map(item =>
          ...
      )}
    </div>
  );
}
```

现在在 Sort 组件中，我们可以基于 `sortKey` 和 `activeSortKey` 得知排序是否被激活。给 Sort 组件增加一个 `className` 属性，用于在排序被激活的时候给用户一个视觉反馈。

src/App.js

```
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = ['button-inline'];

  if (sortKey === activeSortKey) {
    sortClass.push('button-active');
  }

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass.join(' ')}
    >
```

```
      {children}
    </Button>
  );
}
```

这样定义 sortClass 的方法有点蠢，不是吗？有一个库可以让它看起来更优雅。首先，我们需要安装它。

Command Line

```
npm install classnames
```

其次，需要将其导入 src/App.js 文件。

src/App.js

```
import React, { Component } from 'react';
import fetch from 'isomorphic-fetch';
import { sortBy } from 'lodash';
import classNames from 'classnames';
import './App.css';
```

现在，我们可以通过条件式语句来定义组件的 className。

src/App.js

```
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = classNames(
    'button-inline',
    { 'button-active': sortKey === activeSortKey }
  );

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass}
    >
      {children}
```

```
    </Button>
  );
}
```

同样在运行测试时，我们会看到 Table 组件失败的快照测试，及一些失败的单元测试。由于我们再次更改了组件显示，因此可以选择接受快照测试。但是必须修复单元测试。在我们的 `src/App.test.js` 文件中，需要为 Table 组件提供 `sortKey` 和 `isSortReverse`。

src/App.test.js

```
...

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
    sortKey: 'TITLE',
    isSortReverse: false,
  };

  ...

});
```

可能需要再一次接受 Table 组件的失败的快照测试，因为我们给 Table 组件提供更多的 props。

现在，我们的高级排序交互完成了。

练习：

- 使用像Font Awesome[123]这样的库来指示（反向）排序
  - 就是在每个排序标题旁边显示向上箭头或向下箭头图标
- 阅读了解classnames[124]

[123] http://fontawesome.io/

[124] https://github.com/JedWatson/classnames

我们已经学会了 React 中的高级组件技术！现在来回顾一下本章：

- React
  - 通过 ref 属性引用 DOM 节点
  - 高阶组件是构建高级组件的常用方法
  - 高级交互在 React 中的实现
  - 帮助实现条件 classNames 的一个优雅库
- ES6
  - rest 解构拆分对象和数组

你可以在官方代码库[125]找到源代码。

[125] https://github.com/rwieruch/hackernews-client/tree/4.5

React 状态管理与进阶

在前面的章节中，你已经学习了 React 中状态管理的基础知识，本章将会深入这个话题。你将学习到状态管理的最佳实践，如何去应用它们以及为什么可以考虑使用第三方的状态管理库。

状态提取

在你的应用中，只有 App 是具有状态的 ES6 组件。在该组件的方法中，包含了许多应用的状态和业务的处理逻辑。可能你已经注意到了，我们给 Table 组件传入了大量属性。而这些参数中的绝大部分只有在 Table 组件中才被用到。所以有人可能会得出“App 组件不需要知道这些参数”的结论。

整个排序功能只有在 Table 组件中用到了，你可以将其移动到 Table 组件中，因为 App 组件根本不需要了解这些信息。将子状态（substate）从一个组件移动到其他组件中的重构过程被称为状态提取。在这里，你想要将 App 组件中用不到的状态移动到 Table 组件中。这里的状态是从父组件到子组件向下移动。

为了能够在 Table 组件中管理状态和添加方法，需要将其改写成 ES6 类的形式。从函数式无状态组件（functional stateless component）到 ES6 类形式组件的重构非常简单明了。

函数式无状态组件形式的 Table 组件：

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return(
    ...
  );
}
```

ES6 类形式的 Table 组件：

src/App.js

```
class Table extends Component {
  render() {
    const {
      list,
      sortKey,
      isSortReverse,
      onSort,
      onDismiss
    } = this.props;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

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

由于想要在 Table 组件中管理状态，你需要添加构造函数和初始状态。

src/App.js

```
class Table extends Component {
  constructor(props) {
    super(props);

    this.state = {};
  }

  render() {
    ...
  }
}
```

现在你可以将状态和有关排序的方法从 App 组件向下移动到 Table 组件中。

src/App.js

```
class Table extends Component {
  constructor(props) {
    super(props);

    this.state = {
      sortKey: 'NONE',
      isSortReverse: false,
    };

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

  onSort(sortKey) {
    const isSortReverse = this.state.sortKey === sortKey && !this.state.isSortRe\
verse;
    this.setState({ sortKey, isSortReverse });
  }

  render() {
    ...
  }
}
```

别忘了将挪走的状态和 onSort() 方法从 App 组件中移除。

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
      isLoading: false,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
```

```
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
  }

  ...

}
```

除此之外，你还可以让 Table 组件更加轻量。你还可以去掉从 App 组件传入的属性，因为现在这些属性可以由 Table 组件的内部状态控制。

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <Table
          list={list}
          onDismiss={this.onDismiss}
        />
        ...
      </div>
    );
  }
}
```

现在你就可以使用 Table 组件内的 onSort() 方法和状态了。

src/App.js

```
class Table extends Component {

  ...

  render() {
    const {
      list,
      onDismiss
    } = this.props;

    const {
      sortKey,
      isSortReverse,
    } = this.state;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return(
      <div className="table">
        <div className="table-header">
          <span style={{ width: '40%' }}>
            <Sort
              sortKey={'TITLE'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Title
            </Sort>
          </span>
          <span style={{ width: '30%' }}>
            <Sort
              sortKey={'AUTHOR'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Author
            </Sort>
```

```
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'COMMENTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Comments
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'POINTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Points
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            Archive
          </span>
        </div>
        { reverseSortedList.map((item) =>
          ...
        )}
      </div>
    );
  }
}
```

应用应该还是可以像之前一样正常运行，但是你已经做了非常重要的重构工作。相关的逻辑代码和状态信息从 App 组件移动到了 Table 组件中，这使得 App 组件更加轻量。此外因为 Table 的排序逻辑放在了组件内部，所以它的接口也更加轻量了。

状态提取的过程也可以反过来：从子组件到父组件，这种情形被称为状态提升。想象一下，你在子组件中处理了内部的状态信息。现在为了满足新的需求，在其父组件中也显示该组件的状态信息，你需要将状态提升到父组件中。但情况还不止这些，假如你需要在子组件的兄弟组件上显示该组件的状态，你还是需要将状态提升到父组件中。在父组件中处理内部状态，同时将状态信息暴露给相关的子组件。

练习：

- 了解更多关于 React 的状态提升[126] 的内容
- 在使用 Redux 之前学习 React[127]这篇文章中了解更多关于状态提升的信息

[126] https://facebook.github.io/react/docs/lifting-state-up.html
[127] https://www.robinwieruch.de/learn-react-before-using-redux/

再探：setState()

至此，你已经使用过 React 的 `setState()` 方法来管理组件的内部状态。你可以给该函数传入一个对象来改变部分的内部状态。

Code Playground

```
this.setState({ foo: bar });
```

但是 `setState()` 方法不仅可以接收对象。在它的第二种形式中，你还可以传入一个函数来更新状态信息。

Code Playground

```
this.setState((prevState, props) => {
  ...
});
```

为什么你会需要第二种形式呢？使用函数作为参数而不是对象，有一个非常重要的应用场景，就是当更新状态需要取决于之前的状态或者属性的时候。如果不使用函数参数的形式，组件的内部状态管理可能会引起 bug。

当更新状态需要取决于之前的状态或者属性时，为什么使用对象而不是函数会引起 bug 呢？这是因为 React 的 `setState()` 方法是异步的。React 依次执行 `setState()` 方法，最终会全部执行完毕。如果你的 `setState()` 方法依赖于之前的状态或者属性的话，有可能在按批次执行的期间，状态或者属性的值就已经被改变了。

Code Playground

```
const { fooCount } = this.state;
const { barCount } = this.props;
this.setState({ count: fooCount + barCount });
```

想象一下像 `fooCount` 和 `barCount` 这样的状态或属性，在你调用 `setState()` 方法的时候在其他地方被异步地改变了。在不断膨胀的应用中，你会有多个 `setState()` 调用。因为 `setState()` 是异步执行的，你可能像上面的例子一样，依赖了一个已经过期的值。

使用函数参数形式的话，传入 `setState()` 方法的参数是一个回调，该回调会在被执行时传入状态和属性。尽管 `setState()` 方法是异步的，但是通过回调函数，它使用的是执行那一刻的状态和属性。

Code Playground

```
this.setState((prevState, props) => {
  const { fooCount } = prevState;
  const { barCount } = props;
  return { count: fooCount + barCount };
});
```

现在让我们回到代码中来修复这个问题。我们会一起修复一个 `setState()` 依赖于状态和属性的地方，之后你就可以按照同样的方式修复代码中的其他地方。

`setSearchTopStories()` 方法依赖于之前的状态，因此它是个使用函数而不是对象作为 `setState()` 参数的绝佳例子。目前的代码片段如下。

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;
  const { searchKey, results } = this.state;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    results: {
      ...results,
      [searchKey]: { hits: updatedHits, page }
    },
    isLoading: false
  });
}
```

你从 state 变量中提取了一些值，但是更新状态时异步地依赖于之前的状态。现在你可以使用函数参数的形式来防止脏状态信息造成的 bug。

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
    ...
  });
}
```

你可以将已经实现的逻辑移动到函数内部，只需将在 `this.state` 上的操作改为 `prevState` 。

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
    const { searchKey, results } = prevState;

    const oldHits = results && results[searchKey]
      ? results[searchKey].hits
      : [];

    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    return {
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      },
      isLoading: false
    };
  });
}
```

如此可以修复脏状态所导致的问题。还有一个可以改进的地方，由于它是一个函数，你可以将该函数提取出来从而改善代码的可读性。这是使用函数参数形式相对于对象形式的另一个好处，该函数可以独立于组件。但是你需要使用一个高阶函数并将 `result` 传给它。毕竟，你是想根据 API 的获取结果来更新状态。

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;
  this.setState(updateSearchTopStoriesState(hits, page));
}
```

updateSearchTopStoriesState() 是一个高阶函数，因为它返回一个函数。你可以在 App 组件之外定义这个高阶函数。请注意现在函数的签名有了一些变化。

src/App.js

```
const updateSearchTopStoriesState = (hits, page) => (prevState) => {
  const { searchKey, results } = prevState;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  return {
    results: {
      ...results,
      [searchKey]: { hits: updatedHits, page }
    },
    isLoading: false
  };
};

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

搞定！setState() 中函数参数形式相比于对象参数来说，在预防潜在 bug 的同时，还可以提高代码的可读性和可维护性。此外，它可以在 App 组件之外进行测试。你可以将其导出并写个测试来当作练习。

练习：

- 了解更多关于在 React 中正确使用 state[128] 的内容
- 将所有使用 `setState()` 方法的地方重构为函数参数形式
  - 只重构那些需要的地方，即依赖于之前的属性或者状态
- 重新跑一遍测试，确保一切正常工作

[128] https://facebook.github.io/react/docs/state-and-lifecycle.html#using-state-correctly

驾驭 State

前面的章节已经说明，状态管理在大型的应用中是一个至关重要的话题。总体来说，不光是 React，很多单页面应用（SPA）框架都面临这个问题。近些年来应用变得越来越复杂。当今的 web 应用面临的一个重大挑战就是如何驾驭和控制状态。

与其他的解决方案相比，React 已经向前迈进了一大步。单向数据流和简单的组件状态管理 API 非常必要。这些概念使得推断状态和其改变更加容易，在组件级别以及一定程度上的应用级别的状态推断也更加容易。

在不断膨胀的应用中，推断状态的变化随之变得困难。`setState()` 方法使用对象形式而不是函数形式的话，如果在脏状态上进行操作，则可能会引入 bug。为了能够共享状态或者在兄弟组件之间隐藏不必要的状态，你需要将状态进行提升或者降低。有些状况下，组件需要将其状态提升，因为它的兄弟组件依赖于这些状态。也有可能你需要和相隔甚远的组件共享状态，所以你可能需要在其整个组件树中共享该状态。这样做的结果会使得在状态管理中涉及的组件范围很广。但是毕竟组件的主要职责只是描绘 UI，不是吗?

由于这些原因，存在一些独立的解决方案来解决状态管理问题。这些方案不仅仅可以在 React 中使用，但是却使得 React 的生态更加繁荣。你可以使用不同的解决方案来解决你的问题。为了解决规模化的状态管理问题，你可能已经听说过 Redux[129] 或者 MobX[130]。你可以在 React 应用中使用这两者其一。它们还有一些扩展，如 react-redux[131] 和 mobx-react[132] 来将其连接到 React 的视图层。

Redux 和 MobX 超出了本书的讨论范围。当读读完本书的时候，你将获得关于如何继续学习 React 及其生态系统的指导。其中一个学习路线是 Redux。在你深入外部状态管理主题之前，我推荐你阅读这篇文章[133]。它旨在给你一个关于如何学习外部状态管理的更好理解。

练习：

- 阅读更多关于 外部状态管理以及如何学习[134] 的内容
- 看看我的第二本电子书关于 React 中的状态管理[135]

[129] http://redux.js.org/docs/introduction/
[130] https://mobx.js.org/
[131] https://github.com/reactjs/react-redux
[132] https://github.com/mobxjs/mobx-react
[133] https://www.robinwieruch.de/redux-mobx-confusion/
[134] https://www.robinwieruch.de/redux-mobx-confusion/
[135] https://roadtoreact.com/

你已经学习了 React 的高级状态管理！让我们回顾一下前面几章的内容。

- React
  - 将状态提升或者下降到合适的组件中
  - setState 方法可以使用函数参数形式来阻止脏状态的 bug
  - 存在外部的解决方案帮助你掌握驾驭 state

你可以从 官方代码仓库[136] 中找到源代码。

[136] https://github.com/rwieruch/hackernews-client/tree/4.6

部署上线的最后步骤

在最后的章节中我们会向你展示如何部署你的应用到产品环境。你可以使用 Heroku 来免费托管和部署应用，在学习如何部署 React 应用的同时也可以了解更多 *create-react-app* 的相关特性。

弹出

接下来的步骤和知识对于部署产品环境来说并不是必须的，但我依然想要在这里讲解一下。*create-react-app* 提供了一个特性，既可以保持应用的可扩展性，又可以避免被第三方依赖绑架。被第三方依赖绑架通常意味着一旦我们采取了某项技术就没有退出机制的情况。幸运的是 *create-react-app* 为我们提供了一个逃生舱：”eject”。

在 *package.json* 中，你可以找到“start”、“test”和“build”这些命令，用来启动、测试和构建应用。最后的命令就是 *eject*。你可以试着去执行它，但是这个命令只能被执行一次并且不能撤回。这是一个破坏性的命令，一旦执行就不能反悔，如果你只是学习 React，那就没有理由离开 *create-react-app* 提供给你的便利环境。。

假设你运行了 `npm run eject`，它会复制所有的配置和依赖到 package.json 中，同时创建一个新的 *config/*文件夹。你会将整个项目完全转换成带有 Babel 和 Webpack 等工具的自定义配置。最终，你将可以完全控制所有这些工具。

官方文档也说了 *create-react-app* 更适合中小型项目，所以你不用感到愧疚运行“eject”命令来移除掉 *create-react-app* 并拿回控制权。

练习

- 阅读更多关于 eject[137] 的内容

[137] https://github.com/facebookincubator/create-react-app#converting-to-a-custom-setup

部署你的 App

说到底，没有哪个应用应该止步于本地环境（localhost），大家都想要有正式上线的一天。Heroku 是一个提供平台即服务（PaaS）的服务商，你可以在上面托管你的应用。他们提供了和 React 的无缝集成，确切地说，它可以让你在几分钟之内部署一个通过 *create-react-app* 创建的应用。和 *create-react-app* 的理念一样，它可以做到零配置部署。

在部署应用到 Heroku 之前，你需要先满足两个条件：

- 安装 Heroku 命令行工具[138]
- 创建一个 免费的 Heroku 账号[139]

如果你的 Mac 上已经安装了 Homebrew，你可以用终端安装 Heroku 命令行：

Command Line

```
brew update
brew install heroku-toolbelt
```

现在你可以使用 git 和 Heroku 命令行来部署你的应用了

Command Line

```
git init
heroku create -b https://github.com/mars/create-react-app-buildpack.git
git add .
git commit -m "react-create-app on Heroku"
git push heroku master
heroku open
```

搞定。我希望你的应用在线上运行一切正常。如果你遇到什么问题，这里有一些资源可以参考。

- Git 和 GitHub 入门[140]
- 零配置部署 React 应用[141]
- create-react-app 的 Heroku 部署套件[142]

[138] https://devcenter.heroku.com/articles/heroku-command-line
[139] https://www.heroku.com/
[140] https://www.robinwieruch.de/git-essential-commands/
[141] https://blog.heroku.com/deploying-react-with-zero-configuration
[142] https://github.com/mars/create-react-app-buildpack

概述

前面一章就是全书的最后一章节。我希望你享受阅读本书的过程，也希望这本书对你学习 React 有所帮助。如果你喜欢这本书，请将其作为学习 React 的一种方法推荐给你的朋友们。授人玫瑰，手有余香。此外，若不介意的话请花上五分钟在亚马逊[143]上写个短评。

但在阅读本书之后，你又将去向何处呢？你可以自行扩展这个应用，也可以尝试构建属于自己的 React 项目。在你深入另一本书、课程或教程之前，你应该动手创建一个属于自己的 React 项目。持续做上一个星期，把它上线部署到某个地方，然后可以通过 Twitter[144] 联系我把它展示出来。我很好奇你在看到本书之后会创造出点什么，我也很乐于跟我的粉丝们一起分享你的作品。你也可以在 GitHub[145] 找到我并分享你的代码库。

本书中我们只是操练了纯粹的 React，如果你需要进一步扩展你的应用程序，我可以再推荐几个学习途径：

- 状态管理：相信你已经使用过 React 中的 `this.setState()` 和 `this.state`，用于管理和存取组件内部状态。千里之行，始于足下。然而在更大型的项目中，你就会切身体会到 React 组件内部 state 的局限性[146]。因此你可以使用一个第三方的状态管理库，比如说 Redux 或 MobX[147]。你会发现在 Road to React[148] 授课平台上，《驾驭 React 中的状态》（"Taming the State in React"）这门课程将会传授 Redux、MobX 与 React 本地 state 的进阶内容。这门课程也会包含一本电子书，但我推荐大家可以深入到源代码和录屏教学中去。如果你喜欢这本书，那么毫无疑问你也应该看一看《驾驭 React 中的状态》。
- 项目实例：在学习了纯 React 内容过后，要将所学内容运用到自己的项目上，而不是急于学习其他新东西，这对你总是有好处的。你可以使用 React 编写自己的井字游戏（tic-tac-toe）或是一个简单的计算器。有很多不错的教程会教你，仅仅使用 React 就能打造出一些有意思的玩意儿。看看我所做的诸如 分页和无限滚动列表[149]，在 Twitter 墙上展示推文[150] 或是 为 React 应用集成 Stripe 支付功能[151]（译者注：Stripe 类似于支付宝）。
- 代码结构：在阅读本书的过程中，你应该对提到组织代码结构那一章有印象。如果还没来得及实践的话，你现在就可以实际运用起来，可以把你的组件们放到结构化的文件和目录（模块）当中。此外，这也能帮助你理解和学习关于代码拆分、可复用性、可维护性和模块 API 设计的原则。

[143] https://www.amazon.com/dp/B077HJFCQX?tag=21moves-20
[144] https://twitter.com/rwieruch
[145] https://github.com/rwieruch
[146] https://www.robinwieruch.de/learn-react-before-using-redux/
[147] https://www.robinwieruch.de/redux-mobx-confusion/
[148] https://roadtoreact.com/
[149] https://www.robinwieruch.de/react-paginated-list/
[150] https://www.robinwieruch.de/react-svg-patterns/
[151] https://www.robinwieruch.de/react-express-stripe-payment/

- 连接到数据库和/或认证：在一个不断增长的 React 应用程序中，你可能希望最终能够持久化存储数据。数据应该存储在数据库中，以便于浏览器会话结束后仍可继续使用，并能在应用程序的不同用户之间进行共享。引入数据库最简单的方法就是使用 Firebase。在这个全面的教程[152]当中，你会找到一份如何在 React 中使用 Firebase 进行身份验证（注册、登录、注销、…）且循序渐进的指南。除此之外，你还将使用 Firebase 的实时数据库来存储用户实体。在此之后，你可以为你的应用程序存储更多的数据。
- 测试：这本书对测试的涉及尚浅。如果你对测试这个大话题还不太熟悉的话，你应该尝试进一步了解单元测和集成测试的相关概念，特别是在 React 应用的上下文里面。从实现层面上来说，我会强烈推荐 Enzyme 和 Jest，通过单元测试和快照测试来改善你的 React 测试手法。
- 异步请求：你可以使用执行异步请求的第三方库来替代原生的 fetch API：superagent[153] 或 axios[154]。发送异步请求并没有完美的解决方案。但是通过更换 React 外围的组成部分，你可以切身体会到在 React 当中拥有这种灵活性是多么的强大[155]。在某些框架中通常你只能使用某一种方案，但是在诸如 React 这样的灵活的生态系统[156]当中，你可以任意更换解决方案。
- 路由：你可以使用 react-router[157] 为你的应用程序实现路由功能。到目前为止，你的应用程序还只有一个页面。React Router 则能够让你跨多个 URL 创建多个页面。在将路由引入你的应用之后，你不需要发送任何请求到服务端去获取下个页面。路由器（Router）将会帮你在客户端搞定一切。
- 类型检查：在某个章节，你已经使用过 React PropTypes 来定义组件接口。这是预防 bugs 的一种良好实践，但是 PropTypes 只能在运行时执行检查。你可以更进一步地，在编译时就引入静态类型检查。TypeScript[158] 就是备受欢迎的手段之一。但在 React 生态当中，通常情况下大家会使用 Flow[159]。如果你想要让你的应用程序更加健壮的话，我会推荐你去尝试一下 Flow。
- **Webpack** 和 **Babel** 相关工具：在本书中你已经使用过 *create-react-app* 来创建应用程序。到了某个节点，当你已经对 React 足够了解的时候，你可能就想要学习跟 React 相关的一些工具。这可以让你不用 *create-react-app* 也能初始化自己的项目。我会推荐你先了解如何使用 Webpack 和 Babel[160] 完成最少量的配置，然后你可以根据自己的情况去实践更多的工具。例如，你可以在应用程序中使用 ESLint[161] 来统一代码风格。
- **React Native**：React Native[162] 可以将你的应用程序带到移动设备上。React Native 使你能够把在 React 中所学到的知识应用到 iOS 和 Android 应用当中去。一旦你学会了

[152] https://www.robinwieruch.de/complete-firebase-authentication-react-tutorial/
[153] https://github.com/visionmedia/superagent
[154] https://github.com/mzabriskie/axios
[155] https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/
[156] https://www.robinwieruch.de/essential-react-libraries-framework/
[157] https://github.com/ReactTraining/react-router
[158] https://www.typescriptlang.org/
[159] https://flowtype.org/
[160] https://www.robinwieruch.de/minimal-react-webpack-babel-setup/
[161] https://www.robinwieruch.de/react-eslint-webpack-babel/
[162] https://facebook.github.io/react-native/

> React，React Native 的学习曲线就应该不会那么陡峭，两者都是相同的原则和理念。你只是会在移动设备上碰到一些跟 Web 应用有所不同的布局组件。

总之，我希望你能来我的网站[163]看一看，你会发现更多关于 Web 开发和软件工程的有趣内容。你也可以订阅我的更新[164]，大概每月一次就会送达你的收件箱，同时你可以通过 Patron 众筹[165]来支持我的这些内容。此外，在通向 React 之路（Road to React[166]）授课平台上会提供更多学习 React 生态圈的进阶课程。你不应该错过！

请允许我再啰嗦一次，如果你喜欢这本书，我希望你能花点时间想一想谁比较适合学习 React。找到他们然后把这本书分享给他。这对我真的非常重要，这本书就旨在与他人分享。随着时间的推移，越来越多阅读本书的人会与我分享他们的看法和意见，这也会帮助我改进这本书。

由衷地感谢你阅读这本书，《React 学习之道》。

Robin

[163] https://www.robinwieruch.de/
[164] https://www.getrevue.co/profile/rwieruch
[165] https://www.patreon.com/rwieruch
[166] https://roadtoreact.com/