

React.js Essential

React 精髓

使用 React.js 构建可扩展和可维护的 Web 应用

[英] Artemij Fedosejev 著
奇舞团 译

本书是一本使用React构建可扩展Web应用的简明教程。作者首先介绍了如何创建简单和复杂的React元素，在此基础上继续讲解了如何创建无状态和有状态的React组件。与此同时，读者会学习到通过React解决问题的流程，理解React组件强大的生命周期方法，掌握如何在React组件中集成第三方JavaScript库。接着，本书还探讨了如何基于Jest框架编写和运行单元测试，基于测试来确保React代码的可靠与稳定。

最后，作者在React应用中引入Flux架构，进一步提升了React应用的灵活性。

本书读者对象

如果你对原生JavaScript或jQuery、Angular.js、Backbone.js等前端框架有一定经验，并且希望构建可维护、可扩展的Web用户界面，那么本书正适合你阅读。



本书翻译工作由月影领衔的奇舞团翻译小组承担，由赵岩、柏盼、孟之杰负责翻译，由李松峰、梁超负责审校。

[PACKT] open source*
PUBLISHING community experience distilled



博文视点Broadview



@博文视点Broadview



策划编辑：张春雨
责任编辑：徐津平
封面设计：吴海燕



本书主要内容

- 安装React工具使开发更高效
- 创建带属性和子元素的React元素
- 无状态和有状态的React组件
- 使用JSX加速React开发
- 利用生命周期方法为React组件增加响应能力
- 在React组件中集成第三方JavaScript库
- 对React组件应用Flux架构
- 使用Jest测试React组件

上架建议：Web开发/JavaScript

ISBN 978-7-121-28646-9



9 787121 286469 >

定价：65.00元

React.js Essentials

React精髓

[英] Artemij Fedosejev 著
奇舞团 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书面向初中级前端开发者，从头到尾、由浅入深地介绍了使用 React 实现组件化 Web 应用的完整流程。作者从 React 元素、React 组件等基本的概念讲起，循序渐进地讨论了组件状态和生命周期，为开发完整的 React 应用打下了基础。与第三方 JavaScript 框架集成，以及对 React 组件进行单元测试，都是开发 React 应用的重要内容，本书也有详细讲解。最后，为进一步提升 React 应用的灵活性，作者还以实例展示了如何引入 Flux 架构，让读者的开发技能更上一层楼。

Copyright © 2015 Packt Publishing. First published in the English language under the title 'React.js Essentials'.

本书简体中文版专有出版权由 Packt Publishing 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2015-8414

图书在版编目（CIP）数据

React 精髓 / (英) 费多耶夫 (Fedosejev, A.) 著；奇舞团译. —北京：电子工业出版社，2016.5

书名原文：React.js Essentials

ISBN 978-7-121-28646-9

I. ①R… II. ①费… ②奇… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字(2016)第 086472 号

策划编辑：张春雨

责任编辑：徐津平

印 刷：北京天宇星印刷厂

装 订：北京天宇星印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：12.25 字数：244 千字

版 次：2016 年 5 月第 1 版

印 次：2016 年 5 月第 1 次印刷

定 价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

目录

1	给项目预先安装一些有用的工具	1
	了解我们的项目	2
	安装 Node.js 和 npm	3
	安装 Git	4
	从 Twitter Streaming API 中获取数据	5
	使用 Snapkite 引擎来过滤数据	6
	创建项目结构	9
	创建 package.json	10
	复用 Node.js 模块	11
	使用 Gulp.js 来构建应用	12
	创建一个网页	15
	小结	16
2	创建你的第一个 React 元素	17
	理解虚拟 DOM	18
	安装 React	19
	使用 JavaScript 创建 React 元素	20
	type 参数	22
	props 参数	22
	children 参数	23
	渲染 React 元素	27
	使用 JSX 来创建 React 元素	28
	小结	30

3 创建你的第一个 React 组件	31
无状态与有状态.....	31
创建第一个无状态 React 组件.....	32
创建第一个有状态 React 组件.....	37
小结.....	44
4 让 React 组件变得可响应	45
使用 React 解决问题.....	45
规划 React 应用程序.....	47
创建一个 React 组件容器.....	49
小结.....	57
5 结合其他库来使用 React 组件.....	59
在 React 组件中使用其他库.....	59
理解 React 组件的生命周期方法.....	64
挂载方法	66
卸载方法	71
小结.....	72
6 更新 React 组件	73
理解组件生命周期的更新方法.....	73
componentWillReceiveProps()方法.....	74
shouldComponentUpdate()方法	77
componentWillUpdate()方法	77
componentDidUpdate()方法	78
设置 React 组件的默认属性.....	79
验证 React 组件的属性.....	83
创建 Collection 组件	85
小结.....	91

7	构建复杂的 React 组件	93
	创建 TweetList 组件.....	93
	创建 CollectionControls 组件	98
	创建 CollectionRenameForm 组件	105
	创建 Button 组件.....	111
	创建 CollectionExportForm 组件.....	112
	小结.....	114
8	使用 Jest 来测试 React 应用程序.....	115
	为什么要写单元测试.....	115
	创建测试套件、规范和期望.....	115
	安装并运行 Jest.....	121
	创建更多的测试规范和期望.....	122
	测试 React 组件.....	130
	小结.....	137
9	使用 Flux 完善 React 架构	139
	分析当前应用的架构.....	139
	理解 Flux	142
	创建分发器.....	143
	创建动作生成器.....	144
	创建存储.....	145
	小结.....	150
10	使用 Flux 提升应用的可维护性.....	151
	借助 Flux 实现解耦	152
	重构 Stream 组件	155
	创建 CollectionStore.....	161
	创建 CollectionActionCreators.....	167
	重构 Application 组件	169

重构 Collection 组件171

重构 CollectionControls 组件175

重构 CollectionRenameForm 组件178

重构 TweetList 组件180

重构 StreamTweet 组件181

编译.....181

1

给项目预先安装一些有用的工具

Charles F. Kettering 曾经说过：

我只关心未来，因为我的余生都会在那里度过。

这位美国杰出的发明家在软件工程师这个职业诞生之前，就给我们留下了这条最重要的建议。然而，半个世纪后，我们仍在思考是什么导致了混乱的代码，或者是说混乱的思维模式。

你一定有过这种经历：你接手了之前开发者的一份代码，然而这份代码没有任何的文档，你不得不花费数周的时间来理解这份难以调试的“伪自解释代码”。更可怕的是，项目还在持续增长，代码也越来越复杂难懂。大幅修改这份代码是很危险的，并且没有人想碰这份“丑陋”的遗留代码。而重写整个项目需要很大的成本，所以目前的做法就是不停地打补丁来修复遇到的 Bug。众所周知，维护软件的成本往往比原来开发软件的成本还高。

那么今天，应该如何针对未来编写软件呢？我认为是发明一个不变的、简单的工作模式。不管你的项目规模增长到多么大，这一工作模式的复杂性总是保持不变。这个工作模式就是项目的架构，理解了这个架构，也就明白了整个软件是如何运行的。

如果你了解现代的 Web 开发，尤其是前端开发，你会发现我们正生活在一个激动人心的时代。互联网公司和开发者都在解决开发速度与成本，以及代码质量与用户体验之间的矛盾。

2013 年，Facebook 发布了 React，这是一个用于构建 UI 的开源的 JavaScript 库。可以在 <http://facebook.github.io/react/> 上深入了解 React。2015 年年初，Facebook

的 Tom Occhino 总结了 React 为什么这么“牛”：

React 将命令式的 API 包装成为声明式的 API，React 牛在改变了我们写代码的方式。

我们知道，声明式编程代码比命令式代码要少。声明式是告诉计算机去做什么而不用管如何去做，命令式则要描述如何去做。JavaScript DOM API 的调用就是一个命令式编程的例子，jQuery 也是。

Facebook 在生产环境中使用 React 已有多年，例如在 Instagram 和其他产品线中。其实 React 同样适用于比较小的项目，比如下面这个购物清单就是用 React 编写的：<http://fedosejev.github.io/shopping-list-react>。我认为 React 是现如今对开发人员来说最好的构建 UI 的 JavaScript 库之一。

本书的目标是让大家理解 React 的基本原理。为此，我们会逐个地讲解概念，同时告诉大家怎么把这些概念落实到应用中。这样一步一步过来，最终将做出一个实时的 Web 应用，在此期间，我们会随时解决新出现的问题，并讨论 React 对这些问题的解决方案。

我们还将了解 Flux，它实现了一个单向的数据流。通过 Flux 和 React，我们能够写出可预料和可管理的应用代码，将来添加代码就可以增加它的功能，而不会增加项目的复杂度。不管你添加多少新功能，这个 Web 应用的工作模式会一直保持不变。

新技术与已往的技术总会有所不同，React 也不例外。实际上，React 的一些核心概念可能与我们的直觉相反，它们会激发我们深度思考，甚至会让我们感觉是在开历史的倒车，但不要急于下结论。正如你所想的，React 这么设计是有足够理由的，毕竟经验丰富的 Facebook 工程师在生产环境下的核心业务开发中构建和使用了 React。我建议大家保持开放的思维来学习 React，我相信在阅读完这本书后，大家会理解并认同这些新概念。

请跟随本书一起开启 React 的学习之旅，听从 Charles F. Kettering 的建议，为我们的未来好好打算打算！

了解我们的项目

我坚信学习新技术的最佳动力是一个让人兴奋的迫不及待想去构建的项目。作为一位经验丰富的开发者，你可能开发过许多成功的商业项目，这些项目共享了一些类似的功能

和设计模式，甚至还包括目标用户。本书将带大家开发一个令人耳目一新的项目，一个你在日常工作中不太可能会做的项目。这个项目不仅是一个有趣的尝试，让你能通过它学到新知识，而且也会满足你的好奇心，并拓展你的想象空间。当然，考虑到你也很忙，所以这个项目不会耗费太长的时间。

这个项目的名字是 **Snapterest**，是一个发现并收集发布在 Twitter 上的公开照片的 Web 应用。可以认为它是一个照片仅来源于 Twitter 的 Pinterest (www.pinterest.com)。换句话说，我们将实现一个有以下核心功能的网站：

- 实时收集和展示推文。
- 在集合中添加和删除推文。
- 查看收集的推文。
- 将推文信息导出为一段 HTML 代码以供分享。

当开始开发一个新项目时，要做的第一件事就是准备好所使用的工具。对于这个项目来说，你可能不熟悉我们将要使用的工具，所以让我们先来介绍一下它们，以及如何对其进行安装和配置。

如果在安装和配置这些工具的过程中有任何的问题，可以上 <https://github.com/fedosejev/react-essentials> 提交一个 issue，描述清楚你是怎么做的，得到的错误信息是什么，我会尽力帮助你解决这些问题。

在本书中，我将假定你使用 Mac 或者 Windows 来做开发，如果你是一个 Unix 的开发者，那么你应该非常了解自己的包管理工具，使用它来安装本章中需要的工具应该没什么问题。

让我们从安装 Node.js 开始。

安装 Node.js 和 npm

Node.js (<https://nodejs.org>) 是一个允许我们使用熟悉的客户端语言——JavaScript 来编写服务端代码的运行环境。它提供了事件驱动和非阻塞的 I/O 模型，这使得 Node.js 适用于数据密集型 and 实时的应用程序。这意味着，当在我们的项目中使用 Node.js，一旦在推文消息流到达服务器，我们就可以对消息流进行处理——而这真是我们想要的。

开始安装 Node.js，我们安装的版本是 v0.10.40，因为在写本书的时候 Jest 支持的最高版本的 Node.js 就是 v0.10.40（Jest 是一个 Facebook 出品的测试框架，第 8 章会介绍它）。

访问 <http://nodejs.org/dist/v0.10.40/> 并下载适合你的开发系统的安装包。

- OS X: <http://nodejs.org/dist/v0.10.40/node-v0.10.40.pkg>。
- Windows 64-bit: <http://nodejs.org/dist/v0.10.40/x64/node-v0.10.40-x64.msi>。
- Windows 32-bit: <http://nodejs.org/dist/v0.10.40/x64/node-v0.10.40-x86.msi>。

运行并按步骤安装 Node.js，安装完成后，打开终端/命令提示符，输入下面的命令来查看是否安装成功了：

```
node -v
```

如果输出以下结果就表示安装成功：

```
v0.10.40
```

Node.js 有着丰富的模块生态系统来供我们使用，一个模块就是一个可以在应用中反复使用的 Node.js 应用。在本书编写的时候 Node.js 已经有 120 000 多个模块了。Node.js 怎么管理这么多的模块呢？npm 可以满足这个要求，npm 是一个管理 Node.js 模块的包管理器，实际上，npm 和 Node.js 会一起被安装进来，所以现在你的环境已经安装了 npm，在终端/命令提示符输入下面的命令来检测：

```
npm -v
```

你应该会看到这样的输出：

```
1.4.28
```

可以在 www.npmjs.com 上了解更多 npm 的知识。现在我们已经为 Node.js 的应用安装好了环境。

安装 Git

在本书中，我们通过 Git 来安装 Node.js 的模块，如果你没有安装 Git，请参考 <https://>

`git-scm.com/book/en/v2/Getting-Started-Installing-Git` 来将 Git 安装到你的系统中。

从 Twitter Streaming API 中获取数据

我们的 React 应用将从 Twitter 中获取数据，Twitter 提供了 **Streaming API**，使得任何人都可以从公开的推文信息流中获取 JSON 格式的数据。

要开始使用 Twitter 的 Streaming API 前，你需要先完成以下两步。

1. 创建 Twitter 账号。访问 `https://twitter.com` 并且注册账号，如果你之前注册过，直接登录就可以。
2. 创建 Twitter 应用。访问 `https://apps.twitter.com` 并点击 **Create New App**（创建新应用程序）。你需要填写 **Application Details**（应用详情）这个表单，点击同意 **Developer Agreement**（开发者协议），然后点击 **Create your Twitter application**（创建 Twitter 应用），你将看到自己的应用页面。最后切换到 **Keys and Access Tokens**（密钥和访问令牌）标签页。

在这个页面的 **Application Settings**（应用设置）区域，找到下面两个必要的数据。

- **Consumer Key**（API Key），例如：
`jqRDrAlKQCbcBu2o4iclpnvem`
- **Consumer Secret**（API Secret），例如：
`wJcdogJih7uLpjzcs2JtAvdSyCVlqHIRUWI70aHOAf7E3wWIgD`

记录下来这两个数据，本章后面需要它们。

现在我们希望生成 access token。在同样的页面上可以看到 **Your Access Token** 区域是空的，点击 **Create my access token** 会创建下面两部分信息。

- **Access Token**，例如：
`12736172-R017ah2pE2OOctmi46IAE2n0z3u2DV6IqsEcPa0THR`
- **Access Token Secret**，例如：
`4RTJJWIEzIDcs5VX1PMVZolXGZG7L3Ez7Iz1gMdZucDaM`

同样记录下来这两个值。access token 对于你来说是唯一的，你需要保证它的私密性，不能和任何人分享。

现在我们做好使用 Twitter Streaming API 的准备了。

使用 Snapkite 引擎来过滤数据

从 Twitter Streaming API 获取到的推文数据量会比你需要的数据量大很多，所以我们需要找到一种方法将其中有意义的推文数据过滤并显示出来。我建议你查阅一下 Twitter Streaming API 的文档：<https://dev.twitter.com/streaming>，尤其是这个页面中介绍的方法：<https://dev.twitter.com/streaming/reference/post/statuses/filter>。你会看到 Twitter 只提供了少量的过滤器供我们使用，所以我们需要找到另一种方法来进一步过滤数据。

幸运的是，有一个叫作 **Snapkite 引擎** 的 Node.js 应用提供了这个功能。它连接着 Twitter Streaming API，使用内置的或自定义规则的过滤器就可以将过滤后的推文数据输出到一个 web socket 连接上。我们即将开发的 React 应用会直接监听这个 socket 连接上的事件，并在推文到达时直接处理。

我们先安装 Snapkite 引擎。

首先需要克隆 Snapkite 引擎的仓库。克隆就是将程序源代码从 GitHub 上复制到本地目录。在这本书中，我们假设本地目录是 Home 主目录。打开终端/命令提示符，输入下面命令：

```
cd ~  
git clone https://github.com/snapkite/snapkite-engine.git
```

这个命令将创建一个 ~/snapkite-engine/ 目录。接下来我们安装 snapkite-engine 依赖的其他 Node.js 模块。其中一个是 node-gyp 模块。所使用的系统环境（Windows 或 UNIX）不同，需要安装的工具也会不同，工具列表在这个 Web 页面上：<https://github.com/TooTallNate/node-gyp#installation>。

安装完这些工具后，你就可以安装 node-gyp 模块了：

```
npm install --global node-gyp
```

现在进入到~/snapkite-engine 目录:

```
cd snapkite-engine/
```

然后执行下面的命令:

```
npm install
```

使用这个命令可以安装 Snapkite 引擎依赖的 Node.js 模块。现在让我们来配置 Snapkite 引擎, 假设你在~/snapkite-engine/ 目录, 通过下面的命令将 ./example.config.json 文件复制到 ./config.json:

```
cp example.config.json config.json
```

如果你使用的是 Windows, 则执行下面的命令:

```
copy example.config.json config.json
```

打开 config.json, 编辑配置项, 从 trackKeywords 开始。这个配置项设置了需要抓取的关键字, 例如, 如果需要抓取关键字 "my", 我们可以这么更改:

```
"trackKeywords": "my"
```

下一步, 我们需要设置 Twitter Streaming API 相关的 key。把 consumerKey、consumerSecret、accessTokenKey 和 accessTokenSecret 4 个值设置为之前创建 Twitter 应用时保存的值, 其他属性保持默认值就可以了, 如果你想了解这些值分别代表什么含义, 请查看与 Snapkite 引擎相关的文档: <https://github.com/snapkite/snapkite-engine>。

接下来安装 **Snapkite Filters**。Snapkite Filters 是一个根据定义的规则来过滤推文的 Node.js 模块。它有许多种不同的过滤器, 我们可以任意组合它们来过滤推文, 过滤器列表详见 <https://github.com/snapkite/snapkite-filters>。

在我们的应用中, 需要用到以下 4 个过滤器。

- **Is Possibly Sensitive:** <https://github.com/snapkite/snapkite-filter-is-possibly-sensitive>
- **Has Mobile Photo:** <https://github.com/snapkite/snapkite-filter-has-mobile-photo>

- **Is Retweet:** <https://github.com/snapkite/snapkite-filter-is-retweet>
- **Has Text:** <https://github.com/snapkite/snapkite-filter-has-text>

我们来安装这些过滤器。首先跳转到~/snapkite-engine/filters/目录:

```
cd ~/snapkite-engine/filters/
```

接着使用下面的命令来克隆这些过滤器:

```
git clone https://github.com/snapkite/snapkite-filter-is-possibly-sensitive.git
git clone https://github.com/snapkite/snapkite-filter-has-mobile-photo.git
git clone https://github.com/snapkite/snapkite-filter-is-retweet.git
git clone https://github.com/snapkite/snapkite-filter-has-text.git
```

然后设置这些过滤器。为此,我们需要给每个过滤器创建一个 **JSON** 格式的配置文件,并在文件里面定义一些属性。幸运的是,每个过滤器都提供了示例的配置文件,我们可以复制出来,然后根据需要做出相应的更改就可以了。确认你在~/snapkite-engine/filters/目录下,然后执行下面的命令(在 Windows 环境下使用 copy 命令来复制,并且将路径中的正斜线替换成反斜线):

```
cp snapkite-filter-is-possibly-sensitive/example.config.json snapkite-filter-is-possibly-sensitive/config.json
cp snapkite-filter-has-mobile-photo/example.config.json snapkite-filter-has-mobile-photo/config.json
cp snapkite-filter-is-retweet/example.config.json snapkite-filter-is-retweet/config.json
cp snapkite-filter-has-text/example.config.json snapkite-filter-has-text/config.json
```

以上这些配置文件不需要我们进行更改,因为这些配置已经符合我们的要求了。

最后,我们需要告诉 Snapkite 引擎哪些过滤器是我们需要用到的,用编辑器打开~/snapkite-engine/config.json 文件,找到下面的内容:

```
"filters": []
```

将这部分修改为：

```
"filters": [  
  "snapkite-filter-is-possibly-sensitive",  
  "snapkite-filter-has-mobile-photo",  
  "snapkite-filter-is-retweet",  
  "snapkite-filter-has-text"  
]  
]
```

现在你已经成功安装了 Snapkite 引擎和一些 Snapkite 过滤器。现在让我们检查一下是否可以成功运行。跳转到~/snapkite-engine/然后执行下面的命令：

npm start

应该不会出现任何错误信息。如果你收到一些错误信息，并且不知道如何修复，请访问<https://github.com/fedosejev/react-essentials/issues>，创建一个 issue，将收到的错误信息复制粘贴进去。

接下来，我们来创建项目结构。

创建项目结构

现在开始创建我们的项目结构。组织项目结构听起来像是一个简单的任务，其实不然，一个深思熟虑的项目结构会有助于我们理解应用程序的底层架构。稍后在本书介绍 Flux 应用程序架构时，会有相关示例。首先，在用户根目录下创建项目目录：~/snapterest/。

紧接着，我们在这个目录下创建下面两个子目录。

- ~/snapterest/source/：我们将把 JavaScript 的源文件放在这个目录。
- ~/snapterest/build/：我们将把编译后的 JavaScript 文件和 HTML 文件放在这个目录。

现在，在~/snapterest/source/目录下创建 components/子目录，此时你的项目目录如下：

- ~/snapterest/source/components/
- ~/snapterest/build/

项目的基本结构准备好之后，开始创建应用程序文件。首先，我们在 `~/snapterest/source/` 目录下创建最重要的文件 `app.js`。`~/snapterest/source/app.js` 文件将是我们的应用程序的入口点。

现在 `~/snapterest/source/app.js` 文件还是空的，我们把它放一边，先来讨论另一个更重要的问题。

创建 package.json

你以前听说过 **DRY** 法则吗？它代表 **Don't Repeat Yourself**，这个法则促进了软件开发中的一个核心原则——代码复用。最好的代码是不需要你亲自去实现的。事实上，我们项目的目标之一就是写尽可能少的代码。你可能没有意识到这一点，但 **React** 可以帮助我们实现这一目标。它不仅可以节省现在编写代码的时间，而且在今后维护和改进代码时，它将为节省更多的时间。

我们可以运用以下方式减少代码的书写量：

- 通过声明式的编程风格来书写我们的代码。
- 重用其他人编写的代码。

在这个项目中，我们将使用这两种方式来减少代码的书写量。第一种方式是由 **React** 本身提供的。**React** 让我们别无他选，只能使用声明式风格来编写 **JavaScript** 代码。这意味着 **React** 会替我们告诉浏览器怎么去做（如同我们使用 **jQuery** 时的情况），我们只需告诉 **React** 我们想要什么，如何做是 **React** 的事情。这对我们是有利的。

Node.js 和 **npm** 包含了第二种方式。我在这一章前面提到过，有数十万不同的 **Node.js** 应用程序可供我们使用。这意味着我们的应用程序所依赖的功能或许已经有人实现过了。

问题是我们怎么知道从哪里可以获得想要复用的 **Node.js** 应用程序。我们可以通过 `npm install <package-name>` 命令来对其进行安装。在 **npm** 环境下，一个 **Node.js** 应用程序被称为一个包，每个 **npm** 包都有一个 `package.json` 文件，这个文件描述了与这个包相关联的元数据。可以在 <https://docs.npmjs.com/files/package.json> 上了解更多关于 `package.json` 文件中的字段含义。

安装依赖包之前，我们首先将自己的项目初始化为一个包。如果你需要将你的包提交到 `npm` 仓库中以便其他人来使用，那么 `package.json` 是唯一必须包含进去的文件。我们打算建立一个 `Node.js` 应用程序，也不会向 `npm` 提交我们的项目。但是 `package.json` 只是 `npm` 命令能理解的元数据文件，因此可以使用它来存储我们的应用所需要的包。只要将依赖列表存储在 `package.json` 中，就随时可以使用 `npm install` 命令来自动安装它们了。

怎样在应用程序中创建 `package.json` 文件呢？幸运的是，`npm` 附带了一个交互式工具，这个工具会向我们提出几个问题，基于我们的回答，它会为项目创建一个 `package.json` 文件。

确认你处于 `~/snapterest/` 目录下，然后在终端/命令提示符中执行以下命令：

```
npm init
```

这个工具将问你的第一个问题是你的包名。它会为目录名称建议一个默认的名称。在我们的环境中将显示 `name: (snapterest)`。按下回车键便可以将包名设置为默认名称（`snapterest`）。下一个问题是你的软件包的版本，在这里是 `version: (1.0.0)`，按回车键确认。如果我们计划将包提交到 `npm` 以供其他人使用，那么这两个字段将是最重要的。不过由于我们现在不会提交 `npm`，所以就可以将所有问题的答案设置为默认值，继续按回车键直到 `npm init` 完成执行并退出。这时，如果你进入到 `~/snapterest/` 目录，会发现一个新文件：`package.json`。

现在我们来安装其他将要用到的 `Node.js` 应用程序。一个应用建立在多个单独的应用之上，这种方式叫作模块化，而这些单独的应用则被称为模块。所以，现在就可以把我们依赖的 `Node.js` 应用称为 `Node.js` 模块了。

复用 `Node.js` 模块

正如前面所提到的，我们的开发过程中有一步叫作构建。在这一步中，我们的构建脚本会将源文件和所依赖的包转换为单个文件，并交给浏览器执行。构建中最重要的一部分被称为打包。但是我们需要打包哪些文件？原因又是什么？让我们来思考一下。前面简单提到过，我们不是要创建一个 `Node.js` 应用程序，而是要尽可能复用已有的 `Node.js` 模块。这是否意味着我们将要在非 `Node.js` 应用中复用 `Node.js` 模块？这有可能吗？事实上，有一

种方式可以做到。

Browserify 这个工具可以打包所有依赖文件,让打包后的文件在客户端 JavaScript 应用中使用,以达到复用 Node.js 模块的目的。可以在 <http://browserify.org> 了解更多关于 Browserify 的信息,在~/snapterest/目录中执行下面的命令来安装 Browserify:

```
npm install --save-dev browserify
```

注意--save-dev 标志。它会告诉 npm 将 Browserify 添加到 package.json 中作为开发的依赖。接下来,将模块名称作为一个依赖添加到 package.json 文件中以记录我们使用的依赖关系。这样,如果我们以后需要的话,就可以很容易地使用 npm install 命令来安装依赖。另外,运行应用程序所需的依赖和开发应用程序所需要的依赖是有区别的。Browserify 是在构建时(而不是在运行时)使用的,所以是一个开发依赖,因此使用 --save-dev 的标志。如果现在查看 package.json 文件,你会看到:

```
"devDependencies": {  
  "browserify": "^11.0.1"  
}
```

注意,现在 npm 在项目目录~/snapterest/下面创建了一个名为 node_modules 的子目录,这是依赖模块在本地保存的目录。

恭喜你安装了第一个 Node.js 模块! Browserify 将会允许我们在客户端 JavaScript 应用中使用 Node.js 模块。这将是我们的构建过程的一部分。到这里,我认为可以介绍一下我们的构建工具了。

使用 Gulp.js 来构建应用

现如今,任何现代化的客户端应用程序都牵涉了许多问题,这些问题都是由不同的技术来单独处理的。单独处理每个问题简化了管理整个项目过程的复杂性,而缺点是,需要将项目中的各个部分连接成一个连贯的应用程序。就像汽车工厂有将零件组装成汽车的机器人一样,开发者也有把单个模块组装成项目的工具,叫作构建工具。组装的过程称为构建过程。根据项目的规模和复杂性,构建过程花费的时间可能是几毫秒,也可能是几小时。

Node.js 生态环境中有一个很棒的自动化构建工具：**Gulp.js**。可以在 <http://gulpjs.com> 上了解更多有关 Gulp.js 的知识。让我们先来安装 Gulp.js：

```
npm install --save-dev gulp
```

还是这个参数，我们需要将这个模块当作开发的依赖，而不是运行的依赖。只是这一次我们还需要将这个模块安装到全局：

```
npm install --global gulp
```

这样我们就可以在终端/命令提示符中直接执行这个命令，通过下面的命令来检查安装是否成功：

```
gulp
```

如果看到这样的输出：

```
No gulpfile found
```

就意味着你成功安装了 Gulp.js。

gulpfile 是干什么用的呢？我们通过这个文件来描述我们的构建过程。在项目目录 `~/snapterest/` 下创建 `gulpfile.js` 文件，并且将以下内容输入到文件中：

```
var gulp = require('gulp');

gulp.task('default', function() {
  console.log('I am about to learn the essentials of React.js');
});
```

好，现在执行 `gulp` 命令，然后你可能会看到像下面这样的输出：

```
Using gulpfile ~/snapterest/gulpfile.js
Starting 'default'...
I am about to learn the essentials of React.js
Finished 'default' after 62 μs
```

默认情况下，当执行 `gulp` 时，它会执行名为 `default` 的任务（这在我们的意料之中）。现在我们有了一个使用 Gulp.js 的构建系统。接下来让我们创建一个任务：使用 Browserify 打包资源和依赖。

使用下面的内容替换 `gulpfile.js` 文件的内容:

```
var gulp = require('gulp');
var browserify = require('browserify');
var babelify = require('babelify');
var source = require('vinyl-source-stream');

gulp.task('default', function () {
  return browserify('./source/app.js')
    .transform(babelify)
    .bundle()
    .pipe(source('snapterest.js'))
    .pipe(gulp.dest('./build/'));
});
```

正如你所看到的,我们通过 `require()` 函数导入了三个新的依赖模块:`browserify`、`babelify` 和 `vinyl-source-stream`, 我们已经安装了 `browserify` 模块, 现在来安装 `babelify` 模块:

```
npm install --save-dev babelify
```

`babelify` 模块能够解析我们编写的 JSX 语法, 这部分将在下一章中详细介绍。

为什么需要安装 `vinyl-source-stream` 模块呢? 简单来说, 是因为它让我们可以同时使用 `Browserify` 和 `Gulp`。如果你对它感兴趣, 可以访问 <https://www.npmjs.com/package/vinyl-source-stream> 来进一步了解。我们来安装 `vinyl-source-stream` 模块:

```
npm install --save-dev vinyl-source-stream
```

现在我们可以测试 `default` 任务, 执行下面的命令:

```
gulp
```

你会看到这样的输出:

```
Using gulpfile ~/snapterest/gulpfile.js
Starting 'default'...
Finished 'default' after 48 ms
```

更重要的是，如果现在检查你的~/snapterest/build/目录，可以看到生成了一个 snapterest.js 文件，并且文件中已经包含了一些代码。这就是依赖于 Node.js 模块并且可以在浏览器上运行的 JavaScript 应用程序（现在还是空的）。

创建一个网页

如果你急切希望看到一些 React 的效果，那么有好消息要告诉你：马上就能看到了！还需要做的就是创建 index.html 文件，同时引入 snapterest.js 脚本。

在~/snapterest/build/目录下创建 index.html 文件，将下面的 HTML 内容添加到文件中：

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="x-ua-compatible" content="ie=edge, chrome=1" />
    <title>Snapterest</title>
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css">
  </head>
  <body>
    <div id="react-application">
      I am about to learn the essentials of React.js.
    </div>
    <script src="./snapterest.js"></script>
  </body>
</html>
```

在浏览器中打开~/snapterest/build/index.html 文件，你会看到这样的文字：**I am about to learn the essentials of React.js**。是的，我们已经完成了项目的配置，现在该去了解一下 React 了！

小结

在这一章，我们首先学习了使用 React 来构建现代化 Web 应用程序界面的好处，之后讨论了本书中将要构建的项目，最后我们安装了需要的所有工具，并创建了项目结构。在下一章，我们将安装 React，认真研究 React 是如何工作的，并创建我们的第一个 React 元素。

2

创建你的第一个 React 元素

我们知道,要创建一个简单的 Web 应用程序需要编写 HTML、CSS 和 JavaScript 代码。我们之所以使用三种不同的技术,是因为想分离出三个不同的关注点:

- 内容 (HTML)
- 样式 (CSS)
- 逻辑 (JavaScript)

这样的分离适合于创建 Web 页面,因为传说的 Web 页面都是由不同的人来开发的:有人负责使用 HTML 来编写结构,并且使用 CSS 来应用样式,还有人负责使用 JavaScript 来为不同的元素添加各种不同的动态行为。这是一种以内容为中心的开发方式。

现如今,我们通常不再认为网站就是网页的集合。相反,我们构建的 Web 应用可能只有一个页面,这个页面不再是简单地把内容进行布局排列,它已经成为了 Web 应用的容器。这样只有一个页面的 Web 应用程序被称为单页面应用程序 (**Single Page Application**),简称单页应用。你可能会问,如何在单页应用中显示其余的内容呢?我们需要使用 HTML 创建一个额外的布局吗?另外,Web 浏览器怎么知道如何去渲染?

这些都是很好的问题。让我们看看单页应用是如何工作的。当 Web 浏览器加载一个 Web 页面的时候,就会创建这个 Web 页面的文档对象模型 (**DOM**)。DOM 将页面描述为一个树的结构,实际上,只需要通过 HTML 标签就可以描述页面的结构布局——不管你创建的是传统的 Web 页面还是单页应用,两者的区别在于接下来所需要的工作。如果你创建的是传统的 Web 页面,那么现在你已经完成了 Web 页面的布局。反之,如果你创建的是一个单页应用,那么你还需要使用 Web 浏览器提供的 **JavaScript DOM API** 来操纵 DOM 以

创建额外的元素。可以访问 https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model 了解更多。

然而，通过 JavaScript 操作或更改 DOM 有两个问题：

- 如果你决定直接使用原生的 JavaScript DOM API，那么编程风格就很重要，正如上一章所讨论的，这样的编程风格会导致代码库变得难以维护。
- DOM 的更改是很慢的，不像其他 JavaScript 代码可以很快优化。

幸运的是，React 为我们解决了这两个问题。

理解虚拟 DOM

为什么我们需要操作 DOM 呢？因为通常我们的 Web 应用程序都不是静态的。Web 应用程序的状态反映在浏览器绘制的用户界面（UI）上，事件发生时触发状态的改变。那么到底什么样的事件需要我们关注呢？主要有以下两种。

- 用户事件：用户按下键盘、点击鼠标、滚动屏幕、缩放页面尺寸等。
- 服务端事件：应用程序从服务器接收到数据或错误等。

处理这些事件时会发生什么呢？通常，我们会更新应用程序所依赖的数据，而这些数据代表数据模型的状态。相应地，当数据模型的状态发生变化时，我们可能想要通过更新 UI 来反映这种变化。简单说就是需要同步 UI 和数据模型的状态。我们想要其中一个变化后另一个也做出反应，反之亦然。那么，怎样才能实现呢？

有一种方式可以将应用程序的 UI 状态与底层数据模型的状态进行同步，叫作双向数据绑定。双向数据绑定有几种不同的方式。其中一种是键值观察（Key-Value Observing, KVO），Ember.js、Knockout、Backbone 和 iOS 等前端框架都使用这种方式。还有一种是脏检测，Angular 在用。

但 React 没有采用双向数据绑定的方式，而是提出了一个不同的解决方案，称为**虚拟 DOM**。虚拟 DOM 是真实 DOM 的一个快速的、仅存在于内存中的映射，它是一个抽象，允许我们把 JavaScript 和 DOM 当作是反应式的。让我们看看它是如何工作的。

1. 只要数据模型的状态发生变化，虚拟 DOM 和 React 就会将 UI 重绘为虚拟 DOM 的形式。
2. React 会接着计算出这两个虚拟 DOM 之间的区别：数据改变之前的虚拟 DOM 和数据改变后的（即当前的）虚拟 DOM。这两个虚拟 DOM 的区别就是在真实 DOM 中需要实际去更改的部分。
3. 最后，React 在真实 DOM 上更新需要更新的部分。

对比两种不同的虚拟 DOM 然后在真实 DOM 上更新需要更新的部分，这个过程是很迅速的，更好的一点是，React 开发者完全不需要关心哪些部分需要重绘。使用 React 会给人一种感觉：好像每次状态变化时都会重新渲染整个 DOM 一样。

如果你想了解更多关于虚拟 DOM 的知识，包括其背后的逻辑，以及它与数据绑定有何区别，我强烈建议你来看一下 PeteHunt 在 Facebook 上发表的这篇内容丰富的讨论：
<https://www.youtube.com/watch?v=-DX3vJiqxm4>。

现在我们已经了解了虚拟 DOM，让我们安装 React，并创建第一个 React 元素来改变真实的 DOM。

安装 React

在使用 React 之前，我们首先需要安装它。这里介绍两种安装方式：最简方式和使用 `npm install` 命令的安装方式。

最简方式就是直接将下面的 `<script>` 标签添加到 `~/snapterest/build/index.html` 文件中。

- 在开发环境中使用 React，添加下面的这段代码：

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0-beta3/react.js"></script>
```
- 在线上环境中使用 React，添加下面的这段代码：

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0-beta3/react.min.js"></script>
```

这两份代码的区别会在后面几章再做介绍。对于这个项目来说，我们使用开发环境中

的 React。

写这本书的时候, React 的最新版本是 0.14.0-beta3, 随着时间的推移, React 会不断更新, 要确保你使用的是最新版本的 React。如果 React 有了重大变化, 导致不兼容本书中的代码示例, 可以访问 <https://github.com/fedosejev/react-essentials> 来看代码示例和最新版 React 之间的兼容性问题。

第 1 章介绍了 **Browserify**, Browserify 可以让我们在项目中使用 `require()` 函数来导入项目依赖的模块。同样地, 我们也可以使用 `require()` 把 React 导入到我们的项目中。也就是说, 使用 `npm install` 安装 React 可以替代上面介绍的在 `index.html` 文件中添加 `<script>` 标签的方式。

1. 切换到 `~/snapterest/` 目录, 执行以下命令:

```
npm install --save react@0.14.0-beta3 react-dom@0.14.0-beta3
```

2. 使用文本编辑器打开 `~/snapterest/source/app.js` 文件, 分别将 React 和 ReactDOM 库导入到 React 和 ReactDOM 变量上:

```
var React = require('react');  
var ReactDOM = require('react-dom');
```

`react` 包中包含 React 的核心方法, 让我们可以用声明的方式来描述需要渲染的内容, `react-dom` 包提供了负责渲染 DOM 的方法。你可以上 <https://facebook.github.io/react/blog/2015/07/03/react-v0.14-beta-1.html#two-packages> 查看为什么 Facebook 的开发者认为将 React 库分为两个独立的包是一种好的方式。

现在我们已经完成了在项目中使用 React 的准备工作, 接下来开始创建我们的第一个 React 元素。

使用 JavaScript 创建 React 元素

首先来了解一些基本的 React 术语。这些术语能够帮助我们清楚地了解 React 是由什么组成的。随着时间的推移这些术语可能会更新, 可以经常关注官方文档: <http://facebook.github.io/react/docs/glossary.html>。

就像 DOM 是一个节点树一样，React 的虚拟 DOM 也是一个 React 节点树，React 中的一个重要概念是 `ReactDOM`，它是虚拟 DOM 的基本构件，可以是以下任意一种核心类型。

- `ReactDOMElement`：它是 React 中的基础类型，是 DOM 中 `Element` 的一种轻量、无状态、不可变的虚拟表示形式。
- `ReactDOMText`：它是一个数字或者字符串，它代表了文本内容，是 DOM 中的文本节点的虚拟表示形式。

`ReactDOMElement` 和 `ReactDOMText` 都是 `ReactDOMNode`。一个 `ReactDOMNode` 的数组称为 `ReactDOMFragment`。你可以在本章中看到这两种类型的例子。

我们先来实践一个 `ReactDOMElement` 的例子。

1. 将以下代码添加到 `~/snapterest/source/app.js` 文件中：

```
var reactElement = React.createElement('h1');
ReactDOM.render(reactElement, document.getElementById(
  'react-application'));
```

2. 现在你的 `app.js` 文件应该是这样：

```
var React = require('react');
var ReactDOM = require('react-dom');
var reactElement = React.createElement('h1');
ReactDOM.render(reactElement, document.getElementById('react-
application'));
```

3. 切换到 `~/snapterest/` 目录并执行 Gulp 的 default 任务：

gulp

你可能会看到这样的输出：

```
Starting 'default'...
Finished 'default' after 1.73 s
```

4. 切换到 `~/snapterest/build/` 目录，并且在浏览器中打开 `index.html` 文件，你将看到一个空白的 Web 页面。打开浏览器的开发者工具审查这个空白页面的 HTML 元素，可以看到其中一行的内容是：

```
<h1 data-reactid=".0"></h1>
```

至此我们已经创建了第一个 React 元素，看一下我们是如何做到的。

React 库的入口点是 React 对象，这个对象有一个叫作 `createElement()` 的方法，这个方法有三个参数，`type`、`props` 和 `children`：

```
React.createElement(type, props, children);
```

让我们来详细看看每一个参数。

type 参数

`type` 参数必须是一个字符串或者是一个 `ReactClass`：

- 如果是字符串，则必须是一个 HTML 标签名，例如 `'div'`、`'p'`、`'h1'` 等。React 支持所有通用的 HTML 标签和属性，如果想要了解 React 支持的所有 HTML 标签和属性，你可以访问 <http://facebook.github.io/react/docs/tags-and-attributes.html>。
- `ReactClass` 是通过 `React.createClass()` 方法创建的，将会在第 3 章详细介绍。

`type` 参数描述了 HTML 标签或者 `ReactClass` 将会怎样被渲染，在我们上面的例子中，我们会渲染 `h1` HTML 标签。

props 参数

`props` 参数是一个 JavaScript 对象，它会被从父元素传递到子元素（而不是相反），它具有一些不可变的属性。

当我们通过 React 来创建 DOM 元素时，可以通过 `props` 对象的属性来传递表示 HTML 的属性，如 `class`、`style` 等。例如下面这个例子：

```
var React = require('react');
var ReactDOM = require('react-dom');
var reactElement = React.createElement('h1', { className: 'header' });
ReactDOM.render(reactElement,
  document.getElementById('react-application'));
```

上面这段代码会创建一个 class 属性值为 header 的 h1 HTML 元素：

```
<h1 class="header" data-reactid=".0"></h1>
```

请注意，我们这里的属性名称是 className，而不是 class。这是因为 class 是 JavaScript 的关键字。如果使用 class 作为属性名，它将会被 React 忽略，并且会在浏览器的控制台输出以下警告：

Warning: Unknown DOM property class. Did you mean className?

Use className instead.

你可能想知道为什么在 h1 标签上会有 data-reactid=".0" 属性？我们并没有向 props 对象传递它，那么它是从哪里来的呢？它被 React 添加并用来跟踪 DOM 节点的，它可能会在 React 未来的版本中被移除。

children 参数

children 参数描述了这个元素应当含有的子元素，如果子元素存在的话。子元素可以是任何类型的 ReactNode，例如由 ReactElement 表示的虚拟 DOM 元素、由 ReactText 表示的字符串或数字，或者 ReactFragment，即多个 ReactNode 的数组。

让我们看看下面这个例子：

```
var React = require('react');
var ReactDOM = require('react-dom');
var reactElement = React.createElement('h1', { className: 'header' }, 'This is React');
ReactDOM.render(reactElement, document.getElementById('react-application'));
```

上面这段代码会创建一个 h1 HTML 标签，它有一个 class 的属性和一个内容为 This is React 的文本节点：

```
<h1 class="header" data-reactid=".0">This is React</h1>
```

在这里，h1 标签就相当于是一个 ReactElement，而字符串 This is React 相当于一个 ReactText。

接下来，我们再来创建一个 React 元素，并且创建一些 React 元素作为它的子元素：


```
var React = require('react');
var ReactDOM = require('react-dom');

var h1 = React.createElement('h1', { className: 'header', key: 'header' },
  'This is React');
var p = React.createElement('p', { className: 'content', key: 'content' },
  "And that's how it works.");
var reactFragment = [ h1, p ];
var section = React.createElement('section', { className: 'container'},
  reactFragment);

ReactDOM.render(section, document.getElementById('react-application'));
```

我们创建了 `h1`、`p` 和 `section` 这三个 **React** 元素，`h1` 和 `p` 都有它们各自的子节点文本 “This is React” 和 “And that's how it works.”，`section` 有一个名为 `reactFragment` 的子节点，它是一个拥有 `h1` 和 `p` 两个 `ReactElement` 的数组，也是一个 `ReactNode` 的数组。`reactFragment` 数组里面的每个 `ReactElement` 必须有一个 `key` 的属性，这个属性用来帮助 **React** 识别不同的 `ReactElement`。因此，我们可以得到下面的 **HTML** 标记：

```
<section class="container" data-reactid=".0">
  <h1 class="header" data-reactid=".0.$header">This is React</h1>
  <p class="content" data-reactid=".0.$content">And that's how it
works.</p>
</section>
```

现在我们知道了如何创建 **React** 元素，如果我们想要创建多个同一类型的 **React** 元素呢？是不是需要一遍又一遍地重复调用 `React.createElement('type')` 方法呢？这样做是可行的，但我们并不需要这样做，因为 **React** 提供了一个名为 `React.createFactory()` 的工厂函数，工厂函数就是用来创建其他函数的函数。确切地说，`React.createFactory(type)` 这个工厂函数会创建一个函数，该函数创建一个确定类型的 `ReactElement`。

思考下面的例子：

```
var React = require('react');
var ReactDOM = require('react-dom');
```

```
var listItemElement1 = React.createElement('li', { className: 'item-1', key:
'item-1' }, 'Item 1');
var listItemElement2 = React.createElement('li', { className: 'item-2', key:
'item-2' }, 'Item 2');
var listItemElement3 = React.createElement('li', { className: 'item-3', key:
'item-3' }, 'Item 3');

var reactFragment = [ listItemElement1, listItemElement2,
listItemElement3 ];
var listOfItems = React.createElement('ul', { className: 'list-of-items' },
reactFragment);
```

ReactDOM.render(listOfItems, document.getElementById('react-application'));

它会产出这样的 HTML:

```
<ul class="list-of-items" data-reactid=".0">
  <li class="item-1" data-reactid=".0.$item-1">Item 1</li>
  <li class="item-2" data-reactid=".0.$item-2">Item 2</li>
  <li class="item-3" data-reactid=".0.$item-3">Item 3</li>
</ul>
```

我们可以通过创建工厂函数来简化这个例子:

```
var React = require('react');
var ReactDOM = require('react-dom');

var createListItemElement = React.createFactory('li');
var listItemElement1 = createListItemElement({ className: 'item-1', key:
'item-1' }, 'Item 1');
var listItemElement2 = createListItemElement({ className: 'item-2', key:
'item-2' }, 'Item 2');
var listItemElement3 = createListItemElement({ className: 'item-3', key:
'item-3' }, 'Item 3');
```

```
var reactFragment = [ listItemElement1, listItemElement2,
listItemElement3 ];
var listOfItems = React.createElement('ul', { className: 'list-of-items' },
reactFragment);

ReactDOM.render(listOfItems, document.getElementById('react-application'));
```

在上面的例子中，我们先调用了 `React.createFactory()` 函数，并给它传入了 `li` 这个 HTML 标签名以作为类型参数。接着 `React.createFactory()` 函数返回一个新的函数，我们可以使用这个新的函数方便快捷地创建 `li` 类型的元素。我们将这个函数命名为 `createListItemElement`，然后调用这个函数三次，每次只需给它传入 `props` 和 `children` 这两个参数。注意 `React.createFactory()` 和 `React.createElement()` 是一样的，都可以传入 HTML 标签字符串（例如 `li`）或者 `ReactClass` 对象类型的参数。

React 提供了一些内置的工厂函数来创建通用的 HTML 标签，可以在 `React.DOM` 对象下调用它们，例如 `React.DOM.ul()`、`React.DOM.li()`、`React.DOM.div()` 等。使用这些函数可以使上面的例子更加简洁：

```
var React = require('react');
var ReactDOM = require('react-dom');

var listItemElement1 = React.DOM.li({ className: 'item-1', key: 'item-1' },
'Item 1');
var listItemElement2 = React.DOM.li({ className: 'item-2', key: 'item-2' },
'Item 2');
var listItemElement3 = React.DOM.li({ className: 'item-3', key: 'item-3' },
'Item 3');

var reactFragment = [ listItemElement1, listItemElement2, listItemElement3 ];
var listOfItems = React.DOM.ul({ className: 'list-of-items' }, reactFragment);

ReactDOM.render(listOfItems, document.getElementById('react-application'));
```

好，现在我们已经知道怎样创建一个 `ReactNode` 树了，但是，在开始后面的内容之前必须先讨论一下下面这行重要的代码：

```
ReactDOM.render(listOfItems, document.getElementById('react-application'));
```

可能你已经猜到了，它将我们的 `ReactNode` 树渲染成了 `DOM`，让我们仔细看看它是怎么运作的。

渲染 React 元素

`ReactDOM.render()` 方法接收三个参数：`ReactElement`、一个常规的 `DOMELEMENT` 和一个 `callback` 函数：

```
ReactDOM.render(ReactElement, DOMELEMENT, callback);
```

`ReactElement` 是你已经创建的 `ReactNode` 树中的根节点。常规的 `DOMELEMENT` 是这个树的容器 `DOM` 节点。`callback` 函数将会在这个树渲染或者更新之后执行。注意，如果 `ReactElement` 之前已经渲染到父 `DOM Element`，那么 `ReactDOM.render()` 将只更新已经渲染的 `DOM` 树，并且只改变那些相较于最新版 `ReactElement` 确实需要改变的 `DOM`。这就是虚拟 `DOM` 只需要更改少量 `DOM` 的原因。

到现在为止，我们一直假设我们是在 `Web` 浏览器中创建虚拟 `DOM` 的。这是可以理解的，因为毕竟 `React` 是一个 `UI` 库，而所有 `UI` 都在 `Web` 浏览器中渲染。那么你知道什么情况下在客户端渲染 `UI` 会比较慢吗？可能有读者已经猜到了，我要说的是初始页面加载。在这一章开始时就提到了初始页面加载的问题——使用 `React` 后，我们不再创建静态的 `Web` 页面。相反，当浏览器加载我们的 `Web` 应用时，它只接收到最低限度的 `HTML` 标签，通常是 `Web` 应用程序的容器或父元素。然后，利用 `JavaScript` 代码创建 `DOM` 的其余部分，但为此往往需要从服务器请求额外的数据。只有接收到请求的数据，`JavaScript` 代码才会开始更新 `DOM`。然而，请求这些数据是需要时间的，这就是 `DOM` 更新比较慢的原因。那么我们该如何解决这个问题呢？

解决方案可能有点出乎意料：在服务器端而非浏览器中做 `DOM` 更新，就像我们处理静态的 `Web` 页面一样，浏览器会接收到完整的 `HTML`，这部分 `HTML` 可以完整地渲染出应用程序的初始 `UI` 界面。这听起来可能很简单，但是我们不能在服务器上更新 `DOM`，因为 `DOM` 不能离开浏览器单独存在。等等，我们可以做到吗？

我们的虚拟 `DOM` 只依赖 `JavaScript`，而且我们知道，通过 `Node.js` 可以在服务器上运行 `JavaScript`。所以从技术上讲，我们可以在服务器端使用 `React` 库，并且创建 `ReactNode`

树。问题是我们怎样才能将这个树渲染成一个字符串，然后把它发送到客户端呢？

React 有一个名为 `ReactDOMServer.renderToString()` 的方法可以实现：

```
var ReactDOMServer = require('react-dom/server');
ReactDOMServer.renderToString(ReactElement);
```

它会接收一个 `ReactElement` 作为参数，并渲染出初始的 HTML，这不仅比在客户端渲染 DOM 快，而且能改善 Web 应用程序的搜索引擎优化（SEO）效果。

说到生成静态页面，用 React 也可以做到：

```
var ReactDOMServer = require('react-dom/server');
ReactDOMServer.renderToStaticMarkup(ReactElement);
```

类似于 `ReactDOMServer.renderToString()`，这个方法也需要一个 `ReactElement` 作为参数并且输出一个 HTML 字符串。但是，它不会创建 React 内部使用的额外的 DOM 属性，它只会生成更短的 HTML 字符串，方便我们迅速传输。

现在你不仅知道了如何使用 React 的元素创建一个虚拟的 DOM 树，而且也知道了如何在客户端和服务端上对它进行渲染。下一个问题就是我们能否更迅速直观地来做这件事。

使用 JSX 来创建 React 元素

当我们通过反复调用 `React.createElement()` 方法来创建虚拟 DOM 时，将这种反复调用直观转化为具有层次结构的 HTML 标签会变得很难。不要忘记，虽然我们使用的是虚拟 DOM，但仍然要为内容和 UI 创建一个结构化的布局。如果看一眼 React 代码就能知道页面布局的话岂不是很爽？

JSX 是一个可选的类似 HTML 的语法，通过它，我们不使用 `React.createElement()` 就可以创建虚拟 DOM 树。

先看看前面的这个没有使用 JSX 语法的例子：

```
var React = require('react');
var ReactDOM = require('react-dom');
```

```
var listItemElement1 = React.DOM.li({ className: 'item-1', key: 'item-1' },
  'Item 1');
var listItemElement2 = React.DOM.li({ className: 'item-2', key: 'item-2' },
  'Item 2');
var listItemElement3 = React.DOM.li({ className: 'item-3', key: 'item-3' },
  'Item 3');

var reactFragment = [ listItemElement1, listItemElement2,
  listItemElement3 ];
var listOfItems = React.DOM.ul({ className: 'list-of-items' },
  reactFragment);

ReactDOM.render(listOfItems, document.getElementById('react-application'));
```

我们在这个例子中使用 JSX 语法，如下：

```
var React = require('react');
var ReactDOM = require('react-dom');

var listOfItems = <ul className="list-of-items">
    <li className="item-1">Item 1</li>
    <li className="item-2">Item 2</li>
    <li className="item-3">Item 3</li>
</ul>;

ReactDOM.render(listOfItems, document.getElementById('react-application'));
```

正如你所看到的，JSX 允许我们在 JavaScript 代码中使用类似 HTML 的语法。更重要的是，我们现在可以清晰地看到渲染后的 HTML 布局结构。而且只有在“无效的”JavaScript 代码被解释之前，JSX 语法才能转换成有效的 JavaScript 语法。

在上一章，我们安装了可以将 JSX 代码转换为 JavaScript 的 babelify 模块，这个转换在每次执行 gulpfile.js 文件的 default 任务时都会发生¹：

¹ Babel 6.0 之后，如果要转换 JSX 语法，需要首先运行 **npm install babel-preset-react** 安装 react 相关的插件，然后代码也要由 `.transform (babelify)` 改成 `.transform(babelify, {presets : ["react"]})`。——译者注

```
gulp.task('default', function () {  
  return browserify('./source/app.js')  
    .transform(babelify, {presets : ["react"]})  
    .bundle()  
    .pipe(source('snapterest.js'))  
    .pipe(gulp.dest('./build/'));  
});
```

正如你所看到的, `.transform(babelify)` 函数会在 JSX 与其他 JavaScript 代码合并之前将 JSX 代码转换为 JavaScript 代码。

执行下面的命令来测试转换效果:

gulp

然后, 切换到 `~/snapterest/build/` 目录, 在浏览器中打开 `index.html` 文件, 可以看到一个有三个元素的列表。

React 团队提供了一个在线的 JSX 转换工具, 可以使用它来了解 JSX 是怎么工作的, 访问地址: <http://facebook.github.io/react/jsx-compiler.html>。

刚开始使用 JSX, 你可能会觉得很习惯, 但它是一个非常直观和方便的工具。而且你可以选择是否使用它。我认为使用 JSX 会节省开发时间, 所以我在这本书的代码示例中使用了 JSX。如果你选择不使用它, 那么我相信你在本章学到的知识已足够将 JSX 语法转换成 `React.createElement()` 函数调用形式的 JavaScript 代码。

如果你对本章的知识有任何疑问, 可以访问 <https://github.com/fedosejev/react-essentials> 并创建一个 issue 来反馈。

小结

在这一章开始时, 我们讨论了如何解决单页应用的问题。然后学习了什么是虚拟 DOM 和怎么使用 React 来构建它。我们还安装了 React, 并且只使用 JavaScript 创建了第一个 React 元素。接下来学习了如何在浏览器和服务器上渲染 React 元素。最后, 我们通过使用 JSX 以一种更简单的方法创建了 React 元素。

在下一章, 我们将深入研究 React 组件。

3

创建你的第一个 React 组件

在上一章中，我们学习了怎样创建 React 元素和怎样使用它们来渲染 HTML 标记。我们也已经知道使用 JSX 生成 React 元素多么容易实现。至此，我们已经足够了解使用 React 来创建静态 Web 页面的原理了。然而这并不是我们决定学习 React 的原因。我们不想仅仅使用静态 HTML 元素来创建网站，而是希望建立交互式用户界面，让它可以响应用户事件和服务器事件。响应事件是什么意思？怎样能让一个静态 HTML 元素做出响应？怎么能让一个 React 元素做出响应？这一章将通过介绍 React 组件来回答上述及其他相关问题。

无状态与有状态

做出响应的意思是从一个状态转换到另一个状态。这意味着首先你需要有一个状态，并且有改变这种状态的能力。我们在 React 元素中提到过状态或者改变这种状态的能力了吗？没有，它们是无状态的。其唯一目的是构建和渲染虚拟 DOM 元素。事实上，只要传入的参数相同，我们就希望它们以完全相同的方式渲染，这样才好预测它们。这也是使用 React 库的关键好处之一——方便理解 Web 应用的工作方式。

怎么给一个无状态的 React 元素添加状态呢？如果我们不能把状态封装在 React 元素中，那么就需要把 React 元素封装到一个已经有状态的东西里。假设我们使用一个简单的状态机来代表用户界面，那么每个用户行为都会触发状态机状态的改变。每个状态由不同的 React 元素表示。在 React 中，这个状态机就是 React 组件。

创建第一个无状态 React 组件

我们通过下面这个例子看看怎么样创建 React 组件：

```
var React = require('react');
var ReactDOM = require('react-dom');

var ReactClass = React.createClass({
  render: function () {
    return React.createElement('h1', { className: 'header' }, 'React
Component');
  }
});

var reactComponentElement = React.createElement(ReactClass);
var reactComponent = ReactDOM.render(reactComponentElement, document.
getElementById('react-application'));
```

你应该对这个例子中的一些代码已经很熟悉了，其余的部分可以分为简单的三步：

1. 创建 React 类。
2. 创建 React 组件元素。
3. 创建 React 组件。

让我们来仔细看一下怎么创建 React 组件：

1. 调用 `React.createClass()` 函数并定义一个规范对象作为其参数，以创建 `ReactClass`。在这一章中，我们的重点是详细学习这个对象。
2. 调用 `React.createElement()` 函数并将 `ReactClass` 作为其 `type` 参数，以创建 `ReactComponentElement`。第 2 章介绍过 `type` 参数的类型既可以是字符串也可以是 `ReactClass`。这一章将讲述关于 `ReactClass` 的更多内容。
3. 调用 `ReactDOM.render()` 函数并将 `ReactComponentElement` 作为其参数，以创建 `ReactComponent`。

作为参数传给 `React.createClass()` 的规范对象是用来定义组件外观的地方，是定义 React 组件的规范。本章后面会把规范对象称为 React 组件，我们将继续学习这个非常重

要的概念。

规范对象封装组件的状态并描述怎样渲染组件。最基本的，React 组件需要有一个 `render()` 方法，因此它至少会返回 `null` 或者 `false`。下面是最简单的规范对象：

```
{
  render: function () {
    return null;
  }
}
```

如你所想，`render()` 函数负责告诉 React 怎样渲染这个 React 组件。它可以返回 `null`，就像上面的例子，这时不会渲染任何内容。它也可以返回 `ReactDOM`，第 2 章已经介绍了怎样创建 `ReactDOM` 了：

```
{
  render: function () {
    return React.createElement('h1', { className: 'header' }, 'React
Component');
  }
}
```

这个例子展示了如何将 React 元素封装到 React 组件中。这里的 `ReactDOM` 的 `type` 参数是 `h1`，有一个属性对象，以及唯一的子元素 `ReactText`。然后，调用 React 组件的 `render()` 方法时返回它。事实上，将 React 元素封装到一个 React 组件中并不影响它将如何被渲染：

```
<h1 class="header" data-reactid=".0">React Component</h1>
```

正如你看到的，这里生成的 HTML 标记与第 2 章没有使用 React 组件时是一样的。在这种情况下，你可能想知道，既然没有 `render()` 函数也可以来渲染出同样的标记，那么多一个它又有什么好处呢？

好处是：与任何其他函数一样，有了这个 `render()` 函数就能在返回值之前选择返回什么值。到目前为止，你已经看到 `render()` 函数的两个例子：一个返回 `null`，另一个返回 React 元素。我们可以合并两者，并添加一个条件判断来决定渲染什么：

```
{
  render: function () {
    var elementState = {
      isHidden: true
    };

    if (elementState.isHidden) {
      return null;
    }

    return React.createElement('h1', { className: 'header' }, 'React
Component');
  }
}
```

在这个例子中，我们定义了 `elementState` 变量，它引用了一个具有 `isHidden` 属性的对象。这个对象担当 **React** 组件的状态。如果我们想隐藏 **React** 元素，需要设置 `elementState` 的值。当 `isHidden` 为 `true` 时，`render()` 函数将返回 `null`。在这种情况下，**React** 将什么都不渲染。逻辑上，设置 `elementState.isHidden` 为 `false`，将返回我们的 **React** 元素，并渲染出预期的 **HTML** 标记。你可能会问一个问题：怎么设置 `elementState.isHidden` 的值为 `false` 或 `true` 呢？或者通常怎么改变它的值？

让我们想一想在什么情景下我们可能想要改变这个状态。一种是当用户与用户界面有交互行为时，另一种是当服务端发送数据时，或者当过了一段时间后我们想渲染一些其他内容时。`render()` 函数不知道这些事件，而且也不应该知道，因为它唯一的作用是根据传递的数据返回一个 **React** 元素。我们怎么把数据传递给它？

使用 **React** API 有两种方法可以将数据传递给 `render()` 函数：

- `this.props`
- `this.state`

`this.props` 应该看起来很熟悉，我们在第 2 章曾学习过 `React.createElement()` 函数接受 `props` 参数。我们使用它将属性传递给 **HTML** 元素，但并没有讨论传递属性之后会发生什么，以及为什么属性传递给 `props` 对象后会被渲染。

放在 `props` 对象中的任何数据传递给 `React.createElement()` 函数后，可以通过

`this.props` 在 `render()` 函数中都访问到。只要可以从 `this.props` 访问到数据，就可以渲染它：

```
{
  render: function () {
    var elementState = {
      isHidden: true
    };

    if (elementState.isHidden) {
      return null;
    }

    return React.createElement('h1', { className: 'header' }, this.
props.header);
  }
}
```

在这个例子中，我们在 `render()` 函数中使用 `this.props` 访问 `header` 属性。然后将 `this.props.header` 直接作为一个字符串子元素传递给 `React.createElement()` 函数。

在上面的例子中，我们可以将 `isHidden` 的值作为 `this.props` 对象的另一个属性来传递：

```
{
  render: function () {
    if (this.props.isHidden) {
      return null;
    }

    return React.createElement('h1', { className: 'header' }, this.
props.header);
  }
}
```

也可以使用 `this.props` 来处理需要渲染的数据：

```
{
  render: function () {
    if (this.props.isHidden) {
      return null;
    }

    var header = this.props.tweets.length + ' Latest Tweets';
    return React.createElement('h1', { className: 'header' }, header);
  }
}
```

如你所见，我们通过 `this.props.tweets` 访问推文数组并获得了它的长度。然后，我们在长度后面再连接 'Latest Tweets' 字符串。返回的字符串存储在变量 `header` 中，这是处理后的子字符串元素，我们将它传给 `React.createElement()` 函数。

注意在上一个例子中，我们通过 `this.props` 传递 `isHidden`，而不是将它存储在 `render()` 函数中。我们移除了 `elementState` 对象，因为在 `render()` 函数中不需要关心状态。它是一个纯函数，即它不应该改变状态或者访问真实 DOM，或以其他方式与浏览器进行交互。记住，我们也许会在服务器上使用 React（服务器没有浏览器），而无论什么环境我们都希望 `render()` 函数得到相同的结果。

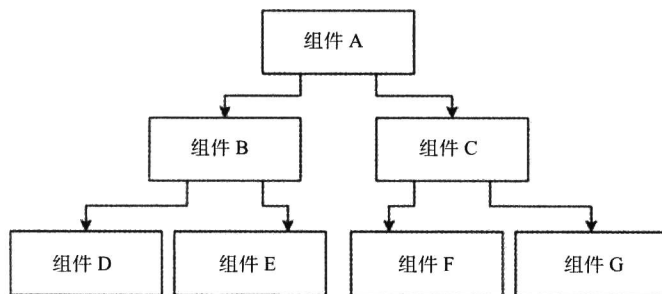
如果 `render()` 函数不管理状态，那该怎么管理呢？我们怎么设置状态？在 React 中处理用户或者浏览器事件时又怎么更新状态呢？

在本章前面，我们学习了在 React 中可以通过 React 组件来表示 UI。React 组件分两种：

- 有状态
- 无状态

等等，不是说 React 组件是状态机吗？没错，每个状态机确实需要一个状态。然而尽可能地保持一些 React 组件无状态是更好的做法。

React 组件是可组合的。换句话说，React 组件可以分层。比如，我们有一个父组件，它有两个子组件，而每个子组件又分别有另外两个子组件。所有组件都是有状态的，它们可以管理自己的状态：



如果顶层组件更新了状态，想要在层级中找出最终将要渲染的子组件容易吗？不容易。有一种设计模式可以消除这种不必要的复杂度。思想是把组件分为两个关注点：如何处理用户界面的交互逻辑和如何呈现数据。

- 在 React 组件中，只有少数是有状态的。它们应该位于组件层级中的顶部。它们封装了所有的交互逻辑，管理用户界面状态，然后使用 props 将状态向下传递到无状态组件。
- 大多数的 React 组件都是无状态的。它们通过 `this.props` 从它们的父组件接收状态数据并渲染数据。

在前面的例子中，我们通过 `this.props` 接收到 `isHidden` 状态数据，然后渲染数据，这个组件是无状态的。

接下来，让我们创建一个有状态的组件。

创建第一个有状态 React 组件

有状态组件最合适处理交互逻辑和管理状态。它们让你更容易推理出应用程序是怎样工作的。这种推理对于创建可维护的应用程序有关键的作用。

React 把组件的状态保存在 `this.state` 中，初始值是 `getInitialState()` 函数的返回值。然而，这取决于我们告诉 React `getInitialState()` 函数返回什么。我们将这个函数添加到组件中：

```
{
  getInitialState: function () {
    return {
```

```
        isHidden: false
    };
},

    render: function () {
        if (this.state.isHidden) {
            return null;
        }

        return React.createElement('h1', { className: 'header' }, 'React
Component');
    }
}
```

在这个例子中，`getInitialState()` 函数返回一个对象，该对象仅有一个值为 `false` 的 `isHidden` 属性。这是我们的 **React** 组件和用户界面的初始状态。注意在 `render()` 函数中，我们正在使用 `this.state.isHidden` 来代替 `this.props.isHidden`。

在本章的前面，我们学习了可以通过 `this.props` 或者 `this.state` 将数据传递到组件的 `render()` 函数中。那么这两者有什么不同呢？

- `this.props` 存储的是从父级传递过来的只读数据。它属于父级，并且不能被它的子元素改变。这个数据应该被认为是不可改变的。
- `this.state` 存储的数据是组件私有的。它能被组件修改。当 `state` 更新后组件会自动重新渲染。

怎样更新组件的 `state`？**React** 提供了一个通用的方式：使用 `setState(data, callback)` 更新 `state`。这个函数有两个参数：

- `data` 函数，表示下一个状态
- `callback` 函数，你将很少用到它，因为 **React** 已经保持了用户界面是最新的。

React 怎么保持用户界面是新的呢？每当你更新组件的状态时，它都会调用组件的 `render()` 函数，包括所有子组件在内都会被重新渲染。事实上，每次调用 `render()` 函数时它都重新渲染全部虚拟 **DOM**。

当调用 `this.setState()` 函数，并向它传递一个表示下一状态的数据对象时，**React**

会将合并下一个状态和当前状态。在合并过程中，React 会使用下一个状态覆盖当前状态。没有被覆盖的当前状态将成为下一状态的一部分。

想象一下这是我们的当前状态：

```
{
  isHidden: true,
  title: 'Stateful React Component'
}
```

我们调用 `this.setState(nextState)`，`nextState` 如下：

```
{
  isHidden: false
}
```

React 会将两个状态合并成一个新的：

```
{
  isHidden: false,
  title: 'Stateful React Component'
}
```

`isHidden` 属性被更新了，`title` 属性没有被删除和更新。

最终，现在我们知道了怎样更新组件的状态，让我们创建一个有状态的组件并对用户事件做出响应：

```
{
  getInitialState: function() {
    return {
      isHeaderHidden: false,
      title: 'Stateful React Component'
    };
  },

  handleClick: function() {
    this.setState({
      isHeaderHidden: !this.state.isHeaderHidden
    });
  }
}
```



```
    },  
  
    render: function() {  
      var headerElement = React.createElement('h1', { className: 'header',  
key: 'header' }, this.state.title);  
      var buttonElement = React.createElement('button', { className: 'btn  
btn-default', onClick: this.handleClick, key: 'button' }, 'Toggleheader');  
      if (this.state.isHeaderHidden) {  
        return React.createElement('div', null, [buttonElement]);  
      }  
  
      return React.createElement('div', null, [buttonElement,  
headerElement ]);  
    }  
  }  
}
```

在这个例子中，我们创建了一个切换按钮用来显示和隐藏标题。首先我们通过 `getInitialState()` 函数的设置了解初始状态对象，它有两个属性：被设置 `false` 的 `isHeaderHidden`，被设置 'Stateful React Component' 为 `title`。

我们可以通过 `this.state` 访问 `render()` 函数中的 `state` 对象。在 `render()` 函数内部，我们创建三个 React 元素：`h1`、`button` 和 `div`。`div` 元素将作为 `h1` 和 `button` 的父元素。不过，`div` 元素可能包含两个子元素，`headerElement` 和 `buttonElement`，也可能仅有一个子元素 `buttonElement`。到底返回哪个 `div`，取决于 `this.state.isHeaderHidden` 的值。组件的当前状态直接影响 `render()` 函数渲染。虽然你已经很熟悉 `render()` 函数，但在这个例子中还是有一些我们之前没有见过的新内容。

我们给 `ReactComponent` 对象添加了一个新属性，叫作 `handleClick()`，它跟 `React` 没有什么特别的关系。它是业务逻辑的一部分，我们使用它来处理 `onClick` 事件。你可以将自己的属性添加到 `ReactComponent` 对象上。我们可以从这个组件上的其他方法访问组件对象上的属性，通过 `this` 引用即可。例如，在 `render()` 和 `handleClick()` 中都可以通过 `this.state` 访问 `state` 对象。

`handleClick()` 函数可以做什么呢？它通过给 `isHeaderHidden` 属性设置一个新值来更新组件的状态，该新值为现有的 `this.state.isHeaderHidden` 取反。

```
this.setState({
  isHeaderHidden: !this.state.isHeaderHidden
});
```

`handleClick()` 函数会响应用户界面做出的动作。用户界面是一个用户可以点击的 `button` 元素，我们可以绑定一个事件处理函数。在 `React` 中，你可以通过 `createElement()` 函数的 `props` 参数来为一个 `React` 元素绑定一些事件处理函数。

```
React.createElement('button', { className: 'btn btn-default', onClick:
  this.handleClick }, 'Toggle header');
```

`React` 使用驼峰命名规范来定义事件处理函数，例如 `onClick`。可以在 <http://facebook.github.io/react/docs/events.html#supported-events> 上查看所有支持的事件。

默认情况下，`React` 会在冒泡阶段触发事件处理函数，但是也可以在事件处理函数名后添加 `Capture` 来告诉 `React` 在捕获阶段触发它们，比如 `onClickCapture`。

`React` 在 `SyntheticEvent` 对象中包装了一个浏览器原生的事件，来保证所有被支持的事件在 `Internet Explorer 8` 及以上浏览器中表现相同。

`SyntheticEvent` 对象提供了与浏览器原生事件相同的 API，这意味着你可以照常使用 `stopPropagation()` 和 `preventDefault()` 方法。如果因为某些原因而需要访问浏览器原生事件，可以通用 `nativeEvent` 属性来做。如果需要处理 `touch` 事件，可以简单地调用 `React.initializeTouchEvents(true)`。

注意，在上一个例子中，将 `onClick` 属性传递到 `createElement()` 函数中并不会在渲染后的 `HTML` 标记中创建内联的事件处理函数：

```
<button class="btn btn-default" data-reactid=".0.$button">Toggle
header</button>
```

这是因 `React` 其实不会将事件处理函数绑定到 `DOM` 节点本身。相反，`React` 仅在最顶层使用一个事件监听器来监听所有事件，并把它们代理到相对应的事件处理函数。

在前面的例子中，我们学习了怎么创建一个有状态的 `React` 组件，用户可以通过交互来改变其状态。我们创建并绑定了一个事件处理函数处理 `click` 事件来更新 `isHeaderHidden` 属性的值。但是你有没有注意到，当用户交互时并没有更新存储在 `state`

中的另一属性 `title` 的值？这看上去是不是有些奇怪？`state` 中的有些数据从没有改变过。这引出了一个重要的问题：我们不应该在 `state` 中放什么？

问自己一个问题：我们可以从组件的 `state` 中移除并且不影响用户页面更新的数据有哪些？边问自己边删除数据，直到你完全确定在不影响用户界面正常更新的前提下没有任何数据可删了。

在我们的例子中，`state` 对象上有一个 `title` 属性，我们可以把它移动到 `render()` 函数中，并且不会破坏切换按钮的交互性。这个组件依然将按着预期运行：

```
{
  getInitialState: function () {
    return {
      isHeaderHidden: false
    };
  },

  handleClick: function () {
    this.setState({
      isHeaderHidden: !this.state.isHeaderHidden
    });
  },

  render: function () {
    var title = 'Stateful React Component';

    var headerElement = React.createElement('h1', { className: 'header',
key: 'header' }, title);
    var buttonElement = React.createElement('button', { className: 'btn
btn-default', onClick: this.handleClick, key: 'button' }, 'Toggle header');

    if (this.state.isHeaderHidden) {
      return React.createElement('div', null, [ buttonElement ]);
    }

    return React.createElement('div', null, [ buttonElement, headerElement ]);
  }
}
```

```
}
```

相反，如果我们把 `isHeaderHidden` 属性移到 `state` 对象外面，就会破坏组件的交互性，因为用户每次点击按钮后，`render()` 函数不再会被自动触发。这就是一个破坏交互的例子：

```
{
  getInitialState: function () {
    return {};
  },
  isHeaderHidden: false,

  handleClick: function () {
    this.isHeaderHidden = !this.isHeaderHidden;
  },

  render: function () {
    var title = 'Stateful React Component';
    var headerElement = React.createElement('h1', { className: 'header', key:
'header' }, title);
    var buttonElement = React.createElement('button', { className:
'btn btn-default', onClick: this.handleClick, key: 'button' }, 'Toggle
header');
    if (this.isHeaderHidden) {
      return React.createElement('div', null, [ buttonElement ]);
    }

    return React.createElement('div', null, [ buttonElement,
headerElement ]);
  }
}
```

这是一个反模式。

请记住这个经验法则：组件的 `state` 应该用来存储组件的事件处理函数随时可能会改变的数据，以达到重新渲染并保持组件的用户界面最新的目的。将 `state` 对象中组件状态保持为最小可能表示形式，并在 `render()` 函数中使用 `state` 和 `props` 来计算数据的其余部

分。无论在 `state` 中放了什么，都需要自己更新。而在 `render()` 函数中放入的内容都会通过 `React` 自动更新。好好利用 `React` 的优势吧。

小结

通过本章的学习，我们到达了一个重要的里程碑：我们学会了怎样通过创建 `React` 组件来封装状态并创建交互式用户页面。我们讨论了无状态和有状态的 `React` 组件，以及两者之间的不同之处。还讨论了浏览器事件，以及怎么在 `React` 中处理它们。

在下一章中，我们将规划 `Snapterest` 应用程序。你将学习怎么使用 `React` 解决问题，以及如何创建复合的 `React` 组件。

4

让 React 组件变得可响应

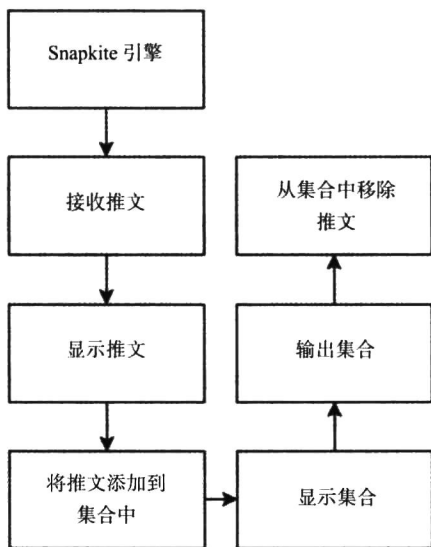
现在你已经知道如何创建有状态和无状态的 React 组件，我们可以尝试将 React 组件组合在一起构建更复杂的用户界面。事实上，我们现在可以开始创建第 1 章讨论过的 Web 应用程序 **Snapterest** 了。在这个过程中，我们将学习如何规划 React 应用程序并创建可组合的 React 组件。让我们开始吧。

使用 React 解决问题

编写 Web 应用程序之前，要考虑 Web 应用要解决什么问题。尽早且尽可能清晰地定义问题是走向成功解决方案（即有用的 Web 应用）的最重要一步。如果在开发过程之初没有定义清楚问题，或者定义得不准确，那么此后你将不得不停下来，并重新思考该怎么做，甚至要扔掉已经完成的一些代码并重新写。这是非常低效的，作为一个专业的软件开发者，你和你的团队的时间都是非常宝贵的，因此，花点时间提前想清楚问题是非常有益处的。在本书开始的时候，我强调过使用 React 的好处之一是代码重用，这意味着你将在更短的时间内做更多的事。因此，在看 React 代码之前，让我们先讨论要解决的问题是什么，而且要记得用 React 来思考。

我们将创建的 **Snapterest** 是一个 Web 应用程序，它会实时接收来自 **Snapkite** 引擎服务器的推文，并将推文逐条显示给用户。虽然我们不知道 **Snapterest** 会在什么时候收到一条新推文，但是当新推文到达时，它至少应该显示 1.5 秒钟以便用户有足够的时间看到并点击它。点击推文会将它添加到一个现有的推文集合或者创建一个新的集合。最终用户能够将集合导出为一段 HTML 代码。

对于我们正在构建的应用，上述描述非常笼统。让我们把它分解成更小的任务，如下图所示：



1. 实时地从 Snapkite 引擎服务器接收推文。
2. 每条推文至少显示 1.5 秒。
3. 通过用户的点击事件将推文添加到一个集合中。
4. 显示集合中的推文。
5. 为集合创建 HTML 代码并导出。
6. 通过用户点击事件从集合中移除推文。

你可以确定哪些任务能使用 React 解决吗？记住 React 是一个用户界面库，因此，任何与用户界面或用户界面交互相关的任务都可以使用 React 解决。在前面的列表中，除了第一个任务外，其他的任务 React 都能胜任，因为第一个任务描述的是数据获取，与用户界面毫无关系。任务 1 将会使用其他库来解决，我们将在下一章讨论。任务 2 和任务 4 描述的内容是需要被显示的，React 组件是最合适的选择。任务 3 和任务 6 描述的是用户事件，如我们在第 3 章所介绍的，用户事件处理可以被很好地封装在 React 组件中。任务 5 怎样使用 React 来解决呢？第 2 章我们讨论过 `ReactDOMServer.renderToStaticMarkup()`

方法能将 React 元素渲染成静态的 HTML 标记字符串,这正是我们解决任务 5 所需的方案。

现在我们已经为每一个任务确定了潜在的解决方案,让我们思考一下我们将要怎么把它们结合在一起创建一个功能全面的 Web 应用程序。

有两种方法来创建可组合的 React 应用:

- 先构建单个的 React 组件,然后将它们组合起来形成更高层级的 React 组件,自底向上来构建层级。
- 从最顶级的 React 元素开始,然后实现它的子组件,自顶向下来构建层级。

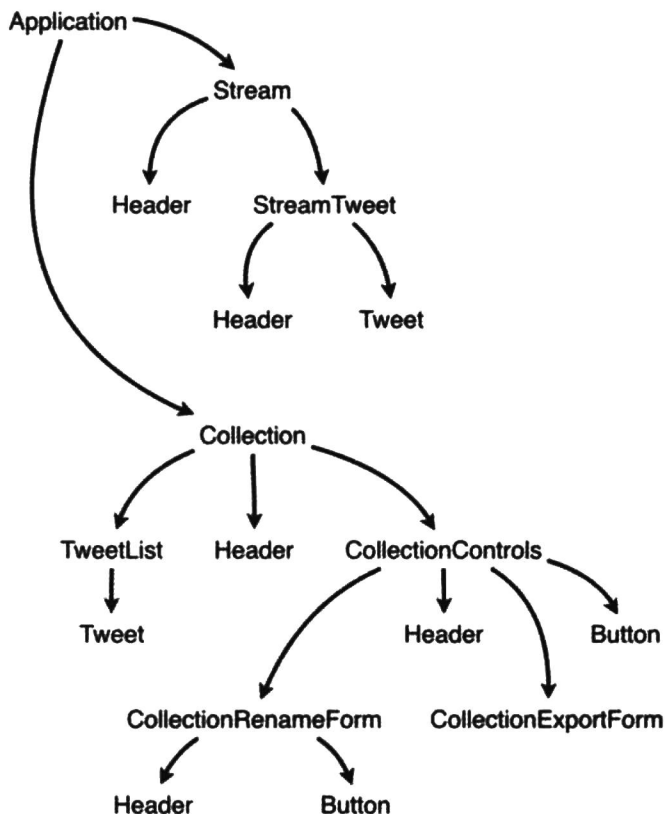
从观察和理解应用架构的角度来看,第二种策略更有优势。我认为在考虑各个部分的功能如何实现之前,先了解所有组件如何组合在一起更重要。

规划 React 应用程序

规划 React 应用时应遵循下面两条简单的原则:

- 每个 React 组件应该代表一个用户界面元素。它应该封装最小的可复用元素。
- 多个 React 组件应该组成一个独立的 React 组件。最终,整个用户页面应该封装成一个 React 组件。

参见下页图,先从最上层的 React 组件 Application 开始。它将封装我们的整个 React 应用程序,它有两个子组件: Stream 和 Collection。Stream 组件将负责连接到一个消息流,接收和显示最新的消息。Stream 组件有两个子组件: StreamTweet 和 Header。StreamTweet 组件将负责显示最新的消息,它由 Header 和 Tweet 组合而成。Header 组件将会渲染头部,它没有子组件。Tweet 组件会渲染来自推文的一张图片。注意我们已经可以复用 Header 组件两次了。



我们的 React 组件的层次图

Collection 组件负责显示收集控件和推文列表。它有两个子组件：CollectionControls 和 tweetlist。前者又有两个子组件：CollectionRenameForm 组件将渲染一个表单，用来重命名集合；CollectionExportForm 组件将渲染一个表单，用来将集合导出到一个叫作 **CodePen** 的服务，这是一个 HTML、CSS 和 JavaScript 的演示网站，可以在 <http://codepen.io> 上了解更多关于 Codepen 的信息。你可能已经注意到，我们将在 CollectionFenameForm 和 CollectionControls 组件中复用 Header 和 Button 组件。TweetList 组件将渲染一个推文列表。每一条推文将被渲染成一个 Tweet 组件。在 Collection 组件中，我们将再次复用 Header 组件。事实上，我们总共要复用 5 次 Header 组件。这对我们来说能省很大事。正如我们在前一章中讨论的，我们应该尽可能保持更多组件是无状态的。因此，总共 11 个组件中只有下面 5 个组件存储状态：

- Application
- CollectionControls
- CollectionRenameForm
- Stream
- StreamTweet

有了规划之后，我们开始实现吧。

创建一个 React 组件容器

让我们首先编辑应用程序的 JavaScript 主文件，使用下面的代码片段替换 `~/snapterest/source/app.js` 文件内容：

```
var React = require('react');
var ReactDOM = require('react-dom');
var Application = require('./components/Application.react');

ReactDOM.render(<Application />, document.getElementById('react-
application'));
```

这个文件仅有四行代码，实现的是：将 `document.getElementById('react-application')` 作为 `<Application />` 组件的部署目标，并将 `Application` 组件渲染到 DOM 中。Web 应用程序的整个用户界面都将被封装在这个 `<Application />` 组件中。

接下来，切换到 `~/snapterest/source/components/` 目录并创建 `Application.react.js` 文件。我们约定所有 React 组件的文件名都以 `react.js` 结尾，这样我们就可以很轻易地分辨 React 与非 React 文件。

让我们看一下 `Application.react.js` 文件的内容：

```
var React = require('react');
var Stream = require('./Stream.react');
var Collection = require('./Collection.react');

var Application = React.createClass({
```

```
getInitialState: function() {
  return {
    collectionTweets: {}
  };
},
addTweetToCollection: function(tweet) {
  var collectionTweets = this.state.collectionTweets;

  collectionTweets[tweet.id] = tweet;

  this.setState({
    collectionTweets: collectionTweets
  });
},
removeTweetFromCollection: function(tweet) {
  var collectionTweets = this.state.collectionTweets;

  delete collectionTweets[tweet.id];

  this.setState({
    collectionTweets: collectionTweets
  });
},
removeAllTweetsFromCollection: function() {
  var collectionTweets = this.state.collectionTweets;

  delete collectionTweets[tweet.id];

  this.setState({
    collectionTweets: {}
  });
},
removeAllTweetsFromCollection: function () {
```

```

    this.setState({
      collectionTweets: {}
    });
  },

  render: function() {
    return (
      <div className="container-fluid">
        <div className="row">
          <div className="col-md-4 text-center">

            <Stream onAddTweetToCollection={this.addTweetToCollection} />
          </div>
          <div className="col-md-8">

            <Collection
              tweets={this.state.collectionTweets}
              onRemoveTweetFromCollection={this.
removeTweetFromCollection}
              onRemoveAllTweetsFromCollection={this.
removeAllTweetsFromCollection}/>

          </div>
        </div>

      </div>
    );
  }
});

module.exports = Application;

```

这个组件的代码比 `app.js` 多了很多,但是这些代码可以很容易地分为三个逻辑部分:

- 引入依赖模块
- 定义 React 组件
- 作为模块导出这个 React 组件

大多数 React 组件中都可以看到这样的逻辑分割，因为包装成 **CommonJS** 模块才能使用 Browserify 引入它们。事实上，这个源文件的第一部分和第三部分的写法都是 CommonJS 规定的，与 React 无关。使用这种模块规范的目的是将应用程序分解成模块以便复用。因为 React 组件和 CommonJS 模块都可以封装代码并使代码更灵活，所以它们在一起自然可以很好地工作。将最终的用户界面逻辑封装在一个 CommonJS 模块形式的 React 组件中，其他模块就可以复用这个被封装好的 React 组件了。

Application.react.js 文件的引入逻辑使用 require() 函数引入了依赖模块：

```
var React = require('react');
var Stream = require('./Stream.react');
var Collection = require('./Collection.react');
```

这里 Application 组件引入了下面两个子组件：

- Stream 组件将在用户界面中渲染信息流部分。
- Collection 组件将在用户页面中渲染集合部分。

我们也需要引入 React 库，但这部分代码都是按照 CommonJS 模块规范编写的，与 React 本身无关。

Application.react.js 文件的第二部分逻辑创建带有以下方法的 ReactApplication 组件：

- getInitialState()
- addTweetToCollection()
- removeTweetFromCollection()
- removeAllTweetsFromCollection()
- render()

只有 getInitialState() 和 render() 方法是 React API，其他方法都是这个组件封装的应用程序逻辑的一部分。讨论完这个组件的 render() 方法会渲染什么内容之后，我们再仔细分析每个逻辑方法：

```
render: function () {
  return (
    <div className="container-fluid">
```

```

<div className="row">
  <div className="col-md-4 text-center">

    <Stream onAddTweetToCollection={this.addTweetToCollection} />
  </div>
  <div className="col-md-8">
    <Collection
      tweets={this.state.collectionTweets}
      onRemoveTweetFromCollection={this.
removeTweetFromCollection}
      onRemoveAllTweetsFromCollection={this.
removeAllTweetsFromCollection} />

  </div>
</div>

</div>
);
}

```

这段代码使用 **Bootstrap** 框架定义了网页布局。如果你不熟悉 **Bootstrap**，我强烈推荐你访问 <http://getbootstrap.com> 上的文档。掌握了这个框架你就能用最快的速度最简单的方法搭建用户界面原型。不过即使你不知道 **Bootstrap**，也不影响理解后面的内容。我们将网页划分为两列：一个小的和一个大的。小的包含 **Stream** 组件，大的包含 **Collection** 组件。可以想象我们的网页被划分成两个不等的部分，它们都包含 **React** 组件。

我们这样使用 **Stream** 组件：

```
<Stream onAddTweetToCollection={this.addTweetToCollection} />
```

Stream 组件有一个 `onAddTweetToCollection` 属性，**Application** 组件将自己的 `addTweetToCollection()` 函数作为这个属性的值。`addTweetToCollection()` 函数会添加一条推文到集合中。这是 **Applicaton** 组件中的一个自定义方法，我们可以用 `this` 关键字来引用它。

让我们看一下 `addTweetToCollection()` 做了什么：

```
addTweetToCollection: function (tweet) {  
  var collectionTweets = this.state.collectionTweets;  
  
  collectionTweets[tweet.id] = tweet;  
  
  this.setState({  
    collectionTweets: collectionTweets  
  });  
},
```

这个函数引用存储在当前 `state` 中的 `CollectionTweets`，添加一条新推文到 `CollectionTweets` 对象，并通过调用 `setState()` 函数来更新 `state`。在 `Stream` 组件中，当 `addTweetToCollection()` 函数被调用时，一条新推文会作为参数被传入。这是一个子组件更新其父组件 `state` 的例子。

这是 `React` 的一个重要机制，它的工作过程如下。

1. 父组件传递一个回调函数作为子组件的属性。子组件可以通过 `this.props` 变量访问这个回调函数。
2. 每当子组件想要更新父组件的 `state` 时，它就会调用这个回调函数并传递所有必要的数据到父组件的新状态中。
3. 父组件更新它的 `state`，而且 `state` 更新会触发 `render()` 函数重新渲染所有必要的子组件。

这就是 `React` 中父组件与子组件的交互机制。这个机制允许子组件将应用程序状态管理委托到它的父组件，子组件只需要关心如何渲染自己就行了。了解了这个机制之后，我们还将多次使用它，因为大部分 `React` 组件要保持无状态。应该只有少量的父组件负责存储和管理应用程序的 `state`。这个最佳实践允许我们按照以下两个不同的关注点来有序地组织 `React` 组件：

- 管理应用程序的 `state` 和渲染。
- 只关注渲染并且将应用程序的 `state` 管理委托到父组件上。

Application 组件的第二个子组件 Collection 如下：

```
<Collection
  tweets={this.state.collectionTweets}
  onRemoveTweetFromCollection={this.removeTweetFromCollection}
  onRemoveAllTweetsFromCollection={this.removeAllTweetsFromCollection}
/>
```

这个组件有如下一些属性。

- tweets：引用当前推文集合。
- onRemoveTweetFromCollection：这个函数从集合中删除特定的推文
- onRemoveAllTweetFromCollection：这个函数从集合中删除所有的推文。

Collection 组件的属性仅仅关注下面两点：

- 如何访问应用程序的 state。
- 如何改变应用程序的 state。

显然，onRemoveTweetFromCollection 和 onRemoveAllTweetsFromCollection 函数允许 Collection 组件改变 Application 组件的 state。另一方面，tweets 属性把 Application 组件的 state 传递给 Collection 组件，使 Collection 组件获得访问 state 的只读权限。

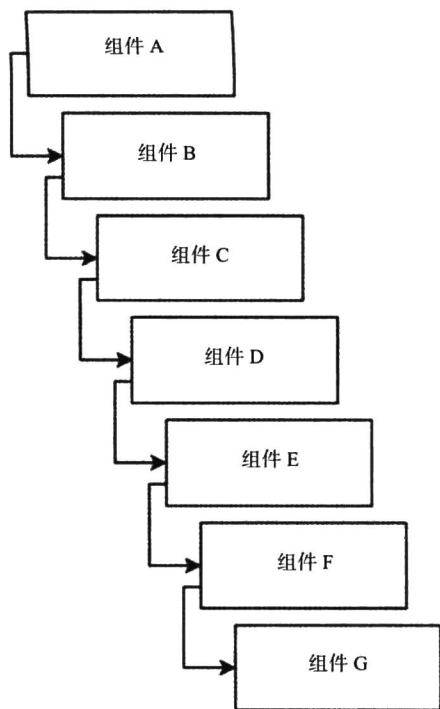
你能觉察到在 Application 和 Collection 组件之间的数据的单向流动吗？以下是它的工作过程：

1. 使用 Application 组件的 getInitialState() 方法初始化 collection Tweets 数据。
2. collectionTweets 数据作为 tweets 属性传递给 Collection 组件。
3. Collection 组件调用 removeTweetFromCollection 和 removeAllTweetFromCollection，更新 Application 中的 collectionTweets 数据，然后再次开始循环。

注意，Collection 组件不能直接改变 Application 组件的 state。Collection 组件有通过 this.props 对象访问 state 的只读权限，并仅可以通过调用父组件传递的回

调函数来更新父组件的 `state`。在 `Collection` 组件中，这些回调函数是 `this.props.onRemoveTweetFromCollection` 和 `this.props.onRemoveAllTweetFromCollection`。

在 `React` 组件层次中，这种数据流动的简单思维模型有助于增加组件的数量，而不增加用户页面的复杂性。比如，它可以有 10 个层级的 `React` 组件嵌套，如下图所示。



在这个层次结构中，如果组件 `G` 要改变根组件 `A` 的 `state`，其所用的方法与组件 `B`、组件 `F` 或者其他任何组件使用的方法完全相同。然而，在 `React` 中，你不应该将数据直接从组件 `A` 传递到组件 `G`。相反，你首先可以把它传递给组件 `B`，然后给组件 `C`，然后给组件 `D`，依此类推，直至组件 `G`。组件 `B` 到组件 `F` 必须携带一些 `transit` 属性，这些属性实际上只对组件 `G` 有用。这看起来可能是在浪费时间，但是这个设计使我们更容易调试应用程序，并可以推理出它是如何工作的。想优化应用程序的架构总是有办法的。`Flux` 就是一种优化方案，本书后面会讨论它。

最后，再看一下改变 `Application` 的 `state` 的两个方法：

```
removeTweetFromCollection: function (tweet) {  
  var collectionTweets = this.state.collectionTweets;  
  
  delete collectionTweets[tweet.id];  
  
  this.setState({  
    collectionTweets: collectionTweets  
  });  
},
```

`removeTweetFromCollection()` 方法从存储在 `Application` 组件的推文集合中移除一条推文，它需要从组件的 `state` 中得到当前的 `collectionTweet` 对象，然后根据给定的 ID 从 `state` 对象中删除一条推文，并使用一个新的 `collectionTweets` 对象来更新组件的 `state`。

此外 `removeAllTweetsFromCollection()` 方法会从组件 `state` 中移除所有推文：

```
removeAllTweetsFromCollection: function () {  
  this.setState({  
    collectionTweets: {}  
  });  
},
```

这些方法都是在子组件 `Collection` 中调用的，因为该组件没有其他方法可以改变 `Application` 组件的 `state`。

小结

在这一章中，我们学习了如何使用 `React` 解决问题。我们首先把问题分解成一些小问题，并讨论了如何使用 `React` 解决它们。然后，创建一些需要实现的 `React` 组件。最后，我们创建了第一个可组合的 `React` 组件，并学习了如何让父组件与它的子组件交互。在下一章，我们将实现我们的子组件并学习 `React` 生命周期相关的方法。

5

结合其他库来使用 React 组件

React 是一个用来构建用户界面的库。如果我们想要整合其他库来负责接收数据该怎么做呢？在上一章，我们列出了 **Snapterest** 应用程序应该完成的 5 个任务，并确定其中 4 个跟用户界面相关，而另一个是完全关于数据接收的：实时从 **Snapkite** 服务器接收推文。

在这一章，我们将学习如何整合 React 与外部 JavaScript 库，并学习 React 组件的生命周期方法，与此同时解决接收数据的重要任务。

在 React 组件中使用其他库

本书前面讨论过，**Snapterest** 应用程序将接收实时的推文流。第 1 章安装的 **Snapkite Engine** 库就是用来连接 Twitter Streaming API, 过滤传入的推文, 并将推文发到应用程序的。反过来，应用程序需要一种连接实时数据流的方法，并监听新推文。

幸运的是，我们不需要自己实现这个功能，而是可以复用另一个 **Snapkite** 模块：`snapkite-stream-client`。让我们安装这个新模块。

切换到 `~/snapterest` 目录并执行下面的命令：

```
npm install --save snapkite-stream-client
```

这个命令将安装 `snapkite-stream-client` 模块，并作为一个依赖模块将它添加到 `package.json` 文件。

现在我们准备在一个 React 组件中重用这个 `snapkite-stream-client` 模块。

上一章我们创建的 Application 组件有两个子组件：Stream 和 Collection。在这一章我们将创建 Stream 组件。

让我们开始创建~/snapterest/source/components/Stream.react.js 文件：

```
var React = require('react');
var SnapkiteStreamClient = require('snapkite-stream-client');
var StreamTweet = require('./StreamTweet.react');
var Header = require('./Header.react');

var Stream = React.createClass({

  getInitialState: function () {
    return {
      tweet: null
    }
  },

  componentDidMount: function () {
    SnapkiteStreamClient.initializeStream(this.handleNewTweet);
  },

  componentWillUnmount: function () {
    SnapkiteStreamClient.destroyStream();
  },

  handleNewTweet: function (tweet) {
    this.setState({
      tweet: tweet
    });
  },

  render: function () {
    var tweet = this.state.tweet;

    if (tweet) {
      return (
```

```

    <StreamTweet
      tweet={tweet}
      onAddTweetToCollection={this.props.onAddTweetToCollection} />
  );
}

return (
  <Header text="Waiting for public photos from Twitter..." />
);
}
});

module.exports = Stream;

```

首先引入 Stream 组件所依赖的下列模块。

- React: React 库。
- StreamTweet 和 Header: React 组件。
- snapkite-stream-client: 工具库。

然后定义 React 组件。以下是 Stream 组件要实现的方法：

- `getInitialState()`
- `componentDidMount()`
- `componentWillUnmount()`
- `handleNewTweet()`
- `render()`

我们已经很熟悉 `getInitialState()` 和 `render()` 方法了，它们是 React API 的一部分。我们也已经知道任何 React 组件都必须至少实现 `render()` 方法。让我们看一下 Stream 组件的 `render()` 方法：

```

render: function () {
  var tweet = this.state.tweet;

  if (tweet) {
    return (

```

```
    <StreamTweet
      tweet={tweet}
      onAddTweetToCollection={this.props.onAddTweetToCollection} />
  );
}

return (
  <Header text="Waiting for public photos from Twitter..." />
);
}
```

正如你所看到的，我们创建了一个新变量 `tweet` 引用组件 `state` 对象的 `tweet` 属性。然后，我们检查这个变量是否引用了一个真实的 `tweet` 对象。如果是，`render()` 方法返回 `StreamTweet` 组件，否则返回 `Header` 组件。

`StreamTweet` 组件渲染一个头部和一条来自流的新推文，而 `Header` 组件仅渲染一个头部。

你有没有注意到 `Stream` 组件自己不渲染任何内容，而是返回另外两个组件中的一个来做实际的渲染？`Stream` 组件的目的是封装应用程序的逻辑并将渲染委托到其他 `React` 组件中。在 `React` 中，至少应该有一个组件来封装应用程序的逻辑、存储，并管理应用程序的 `state`。这个组件通常是根组件或者组件层级中较高级别的组件。如果可能，其他 `React` 子组件都不需要有 `state`。如果把所有 `React` 组件都看成视图（`View`），那么 `Stream` 组件就是一个控制器视图（`ControllerView`）。

现在我们知道 `Stream` 组件渲染什么了，让我们讨论一下其他方法：

```
getInitialState: function () {
  return {
    tweet: null
  }
}
```

`getInitialState()` 方法返回包含一个 `tweet` 属性的初始 `state` 对象，值为 `null`。`Stream` 组件将不断接收新推文，每次接到新推文后需要重新渲染它的子组件。为了实现这一点，我们需要在组件的 `state` 中存储当前的推文。而更新 `state` 之后，`React` 将调用 `render()` 方法重新渲染所有子组件。为了这个目的，我们来实现 `handleNewTweet` 方法：

```
handleNewTweet: function (tweet) {  
  this.setState({  
    tweet: tweet  
  });  
},
```

`handleNewTweet()` 方法接收一个 `tweet` 对象，并将该对象赋值给组件 `state` 的 `tweet` 属性。

新推文来自哪里？什么时候来？让我们看一下 `componentDidMount()` 方法：

```
componentDidMount: function () {  
  SnapkiteStreamClient.initializeStream(this.handleNewTweet);  
},
```

这个函数调用 `SnapkiteStreamClient` 对象的 `initializeStream()` 方法，并传入 `this.handleNewTweet` 回调函数作为参数。`SnapkiteStreamClient` 是一个外部库，我们可以使用它的 API 来初始化推文流。每当 `SnapkiteStreamClient` 接收到推文后，都会调用 `this.handleNewTweet` 函数。

为什么我们将这个方法命名为 `componentDidMount()`？其实这不是我们命名的，是 `React` 命名的。事实上，`componentDidMount()` 方法是一个 `React` API，一个 `React` 组件生命周期方法。它仅仅被调用一次，在组件完成初始化渲染之后执行。在这个时候，`React` 已经创建了 `DOM` 树，由我们的组件表示，此时可以使用其他 `JavaScript` 库来访问这个 `DOM` 树。

`componentDidMount()` 最适合用来整合 `React` 和其他 `JavaScript` 库。我们在这里使用外部的 `SnapkiteStreamClient` 库来连接推文流。

现在我们知道了在 `React` 组件中何时初始化外部 `JavaScript` 库。但是反过来说，我们在 `componentDidMount()` 方法中做的事情应该在什么时候销毁和清理干净？一个好办法是在卸载组件之前清理所有内容。为此，`React` API 提供了另一个组件生命周期方法：`componentWillUnmount()`。

```
componentWillUnmount: function () {  
  SnapkiteStreamClient.destroyStream();  
},
```


`componentWillUnmount()` 方法在组件即将卸载之前被调用。在 `componentWillUnmount()` 方法中,我们调用 `SnapkiteStreamClient` 对象的 `destroyStream()` 方法。`destroyStream()` 会清理与 `SnapkiteStreamClient` 的连接,然后就能安全卸载 `Stream` 组件了。

你可能会问,什么是组件的生命周期方法?为什么会需要它呢?

理解 React 组件的生命周期方法

思考一下 `React` 组件可以做什么?它描述了要渲染什么内容。我们知道它使用 `render()` 方法做这些。然而,有时候仅仅使用 `render()` 方法是不够的,因为我们可能想在组件渲染之前或之后做些事情。另外,我们可能还需要决定是否调用 `render()` 方法。

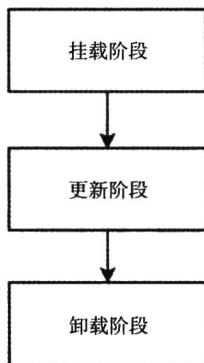
我们所描述的是 `React` 组件被渲染的过程。这个过程有不同的阶段,例如渲染前、渲染中、渲染后等。在 `React` 中,这个过程就叫作组件的生命周期。每个 `React` 组件都会经历这个过程。我们所需要的是一种连接到这个过程的方法,并且在这个过程的不同阶段调用我们自己的函数,以便有更大的控制权。为了这个目的,`React` 提供一些方法,在组件生命周期过程中,我们可以使用这些方法以便在某个阶段发生时得到通知。这些方法就叫作组件的生命周期方法,它们按特定顺序被调用。

所有 `React` 组件生命周期方法可以分为下面三个阶段,参见下页图。

- **挂载 (Mounting)**: 这个阶段发生在组件被插入 `DOM` 时。
- **更新 (Updating)**: 这个阶段发生在组件被重新渲染成虚拟 `DOM` 并决定实际 `DOM` 是否需要更新时。
- **卸载 (Unmounting)**: 这个阶段发生在组件从 `DOM` 中被删除时。

用 `React` 的术语来说,把组件插入 `DOM` 叫挂载,相反在 `DOM` 中删除组件叫卸载。

学习 `React` 组件生命周期方法最好的方式是实际使用它们。让我们创建在这一章的前面已经讨论过的 `StreamTweet` 组件。这个组件将实现大部分 `React` 生命周期方法。



进入 `~/snapterest/source/components/` 目录，然后创建 `StreamTweet.react.js` 文件：

```
var React = require('react');
var ReactDOM = require('react-dom');
var Header = require('./Header.react');
var Tweet = require('./Tweet.react');

var StreamTweet = React.createClass({
  // 在这里定义其他组件生命周期方法

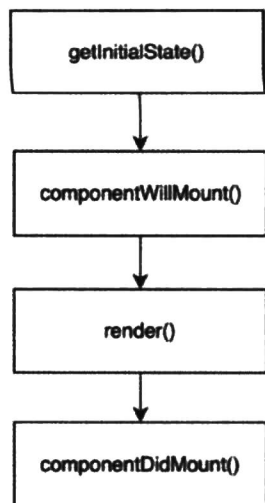
  render: function () {
    console.log('[Snapterest] StreamTweet: Running render()');

    return (
      <section>
        <Header text={this.state.headerText} />
        <Tweet
          tweet={this.props.tweet}
          onClick={this.props.onAddTweetToCollection} />
      </section>
    );
  }
});

module.exports = StreamTweet;
```

StreamTweet 组件现在除了 `render()` 之外还没有其他生命周期方法，我们下面就逐个创建并讨论它们。

如下图所示，其中的四个方法是在组件挂载阶段被调用的：



如上图所示，被调用的方法如下：

- `getInitialState()`
- `componentWillMount()`
- `render()`
- `componentDidMount()`

在这一章中，我们将会讨论除了 `render()` 之外的其他三个方法。当组件被插入 DOM 的时候它们仅被调用一次。让我们仔细看看它们中的每一个。

挂载方法

现在让我们来看看一些有用的挂载方法。

`getInitialState()` 方法

`getInitialState()` 方法是第一个被调用的，它会在 React 将组件插入 DOM 之前被调用。如果你想让组件有一个 `state`，那么可以使用这个方法返回初始的组件 `state`。在

StreamTweet 组件中, 将下面这行注释:

```
// 在这里定义其他组件生命周期方法
```

替换为:

```
getInitialState: function () {  
  console.log('[Snapterest] StreamTweet: 1. Running getInitialState()');  
  
  return {  
    numberOfCharactersIsIncreasing: null,  
    headerText: null  
  };  
},
```

在 StreamTweet 组件的 `getInitialState()` 方法中, 我们将执行以下步骤。

1. 在浏览器的控制台中输出以下消息:

```
[Snapterest] StreamTweet: 1. Running getInitialState().
```

2. 返回一个对象, 将它的属性 `numberOfCharactersIsIncreasing` 和 `HeaderText` 都设置为 `null`。

`numberOfCharactersIsIncreasing` 将监测下一条将要被显示的推文是否比当前显示的推文有更多的字符。在组件的下一个生命周期方法中, 我们会将它设为布尔值。

`headerText` 将存储 StreamTweet 渲染 Header 组件时所需要的文本。

和所有其他挂载方法一样, `getInitialState()` 仅仅会被调用一次。

componentWillMount()方法

`componentWillMount()` 方法是第二个被调用的, 它会在 React 将组件即将插入 DOM 时被调用。在 StreamTweet 组件中, 将下面这段代码添加到 `getInitialState()` 方法后面:

```
componentWillMount: function () {  
  console.log('[Snapterest] StreamTweet: 2. Running componentWillMount()');
```

```
this.setState({
  numberOfCharactersIsIncreasing: true,
  headerText: 'Latest public photo from Twitter'
});

window.snapterest = {
  numberOfReceivedTweets: 1,
  numberOfDisplayedTweets: 1
};
},
```

在这个方法中我们做了一系列的事情。首先，我们记录了这个方法被调用的事实。事实上，出于演示的目的，我们将记录每个组件的生命周期方法。在浏览器中运行这段代码时，你应该可以在 JavaScript 控制台中看到这些日志信息按着预期的顺序被打印出来了。

接下来，我们使用 `this.setState()` 方法来更新组件的 `state`：

- 设置 `numberOfCharactersIsIncreasing` 属性值为 `true`。
- 设置 `headerText` 属性值为字符串 `'Latest public photo from Twitter'`。

因为这是组件将要渲染的第一条推文，我们知道在第一条推文中，字符数当然是从无增加到有的。所以，我们将它设置为 `true`，同时分配默认的文本 `'Latest public photo from Twitter'` 给头部，

如你所知，调用 `this.setState()` 方法会触发组件的 `render()` 方法，所以看起来像是在组件挂载阶段 `render()` 方法被调用了两次。然而在这个例子中，**React** 知道还没有开始渲染，所以它将仅仅调用 `render()` 方法一次。

最后，我们在这个方法中定义一个 `snapterest` 全局对象，它有以下两个属性。

- `numberOfReceivedTweets`：这个属性用来记录所有接收的推文数量。
- `numberOfDisplayedTweets`：这个属性用来记录所有显示的推文数量。

我们将 `numberOfReceivedTweets` 的值设置为 1，因为我们知道当接收到第一条推文时，`componentWillMount()` 方法仅调用一次。我们也知道当接收到第一条推文时，`render()` 方法会被调用，所以将 `numberOfDisplayedTweets` 的值也设置为 1：

```
window.snapterest = {
```

```
    numberOfReceivedTweets: 1,  
    numberOfDisplayedTweets: 1  
  };
```

这个全局对象不是 React 的一部分，也不是应用程序逻辑的一部分，即使我们删除它，程序也将会按照预期工作。window.snapterest 是一个方便的工具，用来随时记录我们处理了多少条推文。我们使用 window.snapterest 仅仅是为了示范。我强烈建议你不要在真实的项目中将属性添加到一个全局对象中，因为你可能会重写现有的属性，或者属性也可能被后面其他不属于你的 JavaScript 代码重写。如果日后你决定在产品中部署 Snapterest，那么请确保从 StreamTweet 组件中删除了这个全局变量 window.snapterest 及相关的代码。

Snapterest 在浏览器中运行了几分钟之后，你可以打开 JavaScript 控制台并键入 snapterest.numberOfReceivedTweets 和 snapterest.numberOfDisplayedTweets 命令。这些命令将会输出一些数字，帮助你更好地了解新推文来的速度有多快，以及它们中有多少没有被显示。在下一个组件生命周期方法中，我们将在 window.snapterest 对象中添加更多的属性。

componentDidMount()方法

componentDidMount() 方法是第三个被调用的。它在 React 将组件插入到 DOM 之后立即被调用。更新后的 DOM 现在可以被访问，这意味着这个方法是初始化其他需要访问这些 DOM 的 JavaScript 库的最佳地方。

在这一章的前面，我们创建了 Stream 组件，使用 componentDidMount() 方法初始化了外部的 snapkite-stream-client JavaScript 库。

让我们看一下这个组件的 componentDidMount() 方法。在 StreamTweet 组件的 componentWillMount() 方法后面添加如下代码：

```
componentDidMount: function () {  
  console.log('[Snapterest] StreamTweet: 3. Running componentDidMount()');  
  
  var componentDOMRepresentation = ReactDOM.findDOMNode(this);
```

```
    window.snapterest.headerHtml = componentDOMRepresentation.  
children[0].outerHTML;  
    window.snapterest.tweetHtml = componentDOMRepresentation.children[1].  
outerHTML;  
  },
```

这里，我们使用 `ReactDOM.findDOMNode()` 方法引用代表 `StreamTweet` 组件的 DOM。我们传递 `this` 作为参数来引用当前的组件（在这个例中是 `StreamTweet`）。`componentDOMRepresentation` 变量指向 DOM 树，我们可以遍历并访问它的各种属性。为了更好地了解这个 DOM 树的样子，让我们仔细看看 `StreamTweet` 组件的 `render()` 方法：

```
render: function () {  
  console.log('[Snapterest] StreamTweet: Running render()');  
  
  return (  
    <section>  
      <Header text={this.state.headerText} />  
      <Tweet  
        tweet={this.props.tweet}  
        onClick={this.props.onAddTweetToCollection} />  
    </section>  
  );  
}
```

使用 JSX 的最大好处之一就是只需通过观察组件的 `render()` 方法，可以很容易地确定组件中有多少子元素。这里，我们可以看到一个父元素 `<section />` 有两个子元素 `<Header />` 和 `<Tweet />`。

所以当我们使用 DOM API 的 `children` 属性来遍历 DOM 树时，我们可以确定它有如下两个子元素。

- `componentDOMRepresentation.children[0]`：表示 `<Header />` 组件的 DOM。
- `componentDOMRepresentation.children[1]`：表示 `<Tweet />` 组件的 DOM。

每个元素的 `outerHTML` 属性获取 HTML 字符串,这个字符串表示每个元素的 DOM 树。为了方便,我们在全局 `window.snapterest` 对象中引用这个 HTML 字符串,正如本章前面讨论的。

如果你使用其他 JavaScript 库,例如将 **jQuery** 和 **React** 一起使用,那么使用 `componentDidMount()` 方法是整合两者的一个机会。如果你想发送一个 AJAX 请求,或者使用 `setTimeout()`、`setInterval()` 设置定时器,那么也可以在这个方法中操做。一般来说,`componentDidMount()` 应该是整合 **React** 库和非 **React** 库时应该优先选择的组件生命周期方法。

到目前为止,我们已经学习了 **React** 组件提供的基本挂载方法。在 `StreamTweet` 组件中使用了提到的所有三个方法。我们也讨论了 `StreamTweet` 的 `render()` 方法。这些知识是我们理解 **React** 将怎么样初始渲染 `StreamTweet` 组件所需要知道的。在它第一次渲染时,**React** 将按顺序执行以下方法:

1. `getInitialState()`
2. `componentWillMount()`
3. `render()`
4. `componentDidMount()`

这些方法在 **React** 组件的挂载阶段被调用。它们仅执行一次,除非我们卸载组件并再次挂载它。

接下来,让我们来讨论一下 **React** 组件的卸载阶段。

卸载方法

现在让我们看一个常用的卸载方法

`componentWillUnmount()`方法

在这个阶段 **React** 仅提供一个方法,那就是 `componentWillUnmount()`。它在 **React** 即将从 DOM 中删除并销毁组件之前被调用。对于清理组件在安装或更新阶段创建的所有数据,这个方法是很有用的。这正是我们在 `StreamTweet` 组件中要做的。在 `StreamTweet` 组件的 `componentDidMount()` 方法后面添加这段代码:


```
componentWillUnmount: function () {  
  console.log('[Snapterest] StreamTweet: 8. RunningcomponentWillUnmount()');  
  
  delete window.snapterest;  
},
```

在 `componentWillUnmount()` 方法中，我们使用 `delete` 操作符删除全局 `window.snapterest` 对象：

```
delete window.snapterest;
```

删除 `window.snapterest` 将会保证你的全局对象不被污染。如果你已经在 `componentDidMount()` 方法中创建了任何额外的 DOM 元素，那么 `componentWillUnmount()` 是删除它们的最好地方。在整合 React 组件和其他 JavaScript API 时，你可以将 `componentDidMount()` 和 `componentWillUnmount()` 方法想象成一个两步骤机制：

1. 在 `componentDidMount()` 方法中初始化它们。
2. 在 `componentWillUnmount()` 方法中结束它们。

这种方式可以使需要操作 DOM 的第三方 JavaScript 库与 React 渲染的 DOM 保持同步。

这些就是我们所需要知道的关于有效地卸载 React 组件的内容。

小结

在这一章，我们创建了 Stream 组件并学习了怎么整合 React 组件和外部 JavaScript 库，还学习了关于 React 组件生命周期的方法。我们也关注并详细讨论了挂载与卸载方法，并开始实现 StreamTweet 组件。

下一章将讲述组件生命周期的更新方法。我们将同时实现 Header 和 Tweet 组件，并学习怎样设置组件的默认属性。

6

更新 React 组件

在上一章中，我们学习了 React 组件可以经历三个阶段：

- 挂载阶段
- 更新阶段
- 卸载阶段

我们已经讨论了挂载和卸载阶段。在这一章中，我们会讨论更新阶段。在此阶段，React 组件已经被插入到 DOM 中。这个 DOM 代表组件的当前状态，并且当状态改变，React 需要评估一个新的状态将如何改变先前渲染的 DOM。

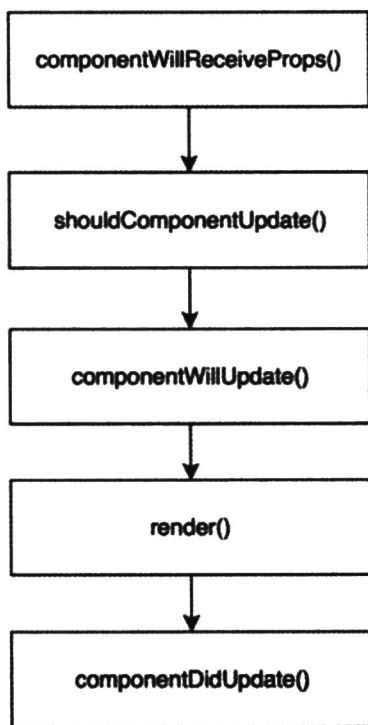
React 提供给我们一些方法，用来影响在更新过程中会渲染什么，并让我们知道更新何时发生。这些方法允许我们控制从当前组件状态到下一个组件状态的过渡过程。下面我们就来学习关于 React 组件更新方法的更多有用特性。

理解组件生命周期的更新方法

React 组件有 5 个属于组件更新阶段的生命周期方法：

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

下图可以帮你更好地总览这 5 个方法：



我们已经很熟悉 `render()` 方法了，下面让我们讨论一下其他 4 个方法。

componentWillReceiveProps()方法

首先我们在 `StreamTweet` 组件中使用 `componentWillReceiveProps()` 方法。在 `StreamTweet.react.js` 文件的 `componentDidMount()` 方法后面添加如下代码：

```
componentWillReceiveProps: function (nextProps) {  
  console.log('[Snapterest] StreamTweet: 4. Running  
componentWillReceiveProps()');  
  
  var currentTweetLength = this.props.tweet.text.length;  
  var nextTweetLength = nextProps.tweet.text.length;  
  var isNumberOfCharactersIncreasing = (nextTweetLength >  
currentTweetLength);
```

```
var headerText;

this.setState({
  numberOfCharactersIsIncreasing: isNumberOfCharactersIncreasing
});

if (isNumberOfCharactersIncreasing) {
  headerText = 'Number of characters is increasing';
} else {
  headerText = 'Latest public photo from Twitter';
}

this.setState({
  headerText: headerText
});

window.snapterest.numberOfReceivedTweets++;
},
```

这个方法在组件生命周期的更新阶段被第一个调用，具体来说，就是当组件从它的父组件接收到新属性时被调用。

此方法为我们提供了一个机会，让我们可以使用 `this.props` 对象和 `nextProps` 对象来比较当前组件和下一个组件的属性。根据比较结果，可以选择使用 `this.setState()` 函数来更新组件的 `state`，在这种场景下将不会触发额外的渲染。

让我们看一下以下代码：

```
var currentTweetLength = this.props.tweet.text.length;
var nextTweetLength = nextProps.tweet.text.length;
var isNumberOfCharactersIncreasing = (nextTweetLength >
currentTweetLength);
var headerText;

this.setState({
  numberOfCharactersIsIncreasing: isNumberOfCharactersIncreasing
});
```

我们首先会获得当前推文和下一条推文的长度。当前推文的长度可以通过 `this.props.tweet` 获得，下一条推文的长度可以通过 `nextProps` 获得。然后比较它们的长度，检查下一条推文是否比当前推文长。将比较结果保存在 `isNumberOfCharactersIncreasing` 变量中。最终设置 `numberOfCharactersIsIncreasing` 属性的值为变量 `isNumberOfCharactersIncreasing` 来更新组件的 `state`。

然后设置头部文本如下：

```
if (isNumberOfCharactersIncreasing) {
  headerText = 'Number of characters is increasing';
} else {
  headerText = 'Latest public photo from Twitter';
}

this.setState({
  headerText: headerText
});
```

如果下一条推文更长，就将头部文本设置为 `'Number of characters is increasing'`；反之，就将头部设置为 `'Latest public photo from Twitter'`。然后再一次通过设置 `headerText` 属性的值为变量 `headerText` 来更新组件的 `state`。

注意，我们在 `componentWillReceiveProps()` 方法中调用了 `this.setState()` 函数两次。这是为了说明不管在 `componentWillReceiveProps()` 方法中调用 `this.setState()` 多少次，它都不会触发组件额外的渲染。React 做了内部优化，会把状态更新操作放在一起批量执行。

`StreamTweet` 组件每次接收到新推文时，`componentWillReceiveProps()` 方法都会被调用一次，因此可以用这个方法来计算接收到的推文总数：

```
window.snapterest.numberOfReceivedTweets++;
```

现在，我们知道怎样检测下一条推文是否比当前显示的推文长了，但是我们如何才能选择不渲染下一条推文呢？

shouldComponentUpdate()方法

通过 `shouldComponentUpdate()` 方法中，我们可以决定组件的下一个状态是否触发组件的重新渲染。这个方法返回一个布尔值，默认为 `true`。但是也可以使它返回 `false`，此时下面的组件方法都不会被调用：

- `componentWillUpdate()`
- `render()`
- `componentDidUpdate()`

如果跳过对组件的 `render()` 方法的调用，就会阻止对该组件的重新渲染，这将提高应用程序的性能，因为没有额外的 DOM 改变。

`shouldComponentUpdate()` 方法在组件生命周期的更新阶段被第二个调用。

这个方法可以很好地阻止显示下一条内容少于一个字符的推文。在 `StreamTweet` 组件的 `componentWillReceiveProps()` 方法后面加入如下代码：

```
shouldComponentUpdate: function (nextProps, nextState) {  
  console.log('[Snapterest] StreamTweet: 5. Running  
  shouldComponentUpdate()');  
  
  return (nextProps.tweet.text.length > 1);  
},
```

如果下一条推文的长度大于 1，`shouldComponentUpdate()` 会返回 `true`，`StreamTweet` 组件会渲染下一条推文。否则，`shouldComponentUpdate()` 返回 `false`，`StreamTweet` 组件不会渲染下一条推文。

componentWillUpdate()方法

`componentWillUpdate()` 方法在 `React` 即将更新 DOM 之前被调用。它得到以下两个参数。

- `nextProps`：下一个属性对象。
- `nextState`：下一个状态对象。

你可以使用这些参数来准备 DOM 更新。不过，你不能在 `componentWillUpdate()` 方法中使用 `this.setState()`。如果你想更新组件的状态以响应其属性更改，那么请在 `componentWillReceiveProps()` 方法中做，React 会在属性变化时调用它。

为了演示 `componentWillUpdate()` 方法什么时候被调用，我们需要在 `StreamTweet` 组件中记录它。在 `shouldComponentUpdate()` 方法后面添加如下代码：

```
componentWillUpdate: function (nextProps, nextState) {
  console.log('[Snapterest] StreamTweet: 6. Running
  componentWillUpdate()');
},
```

在调用 `componentWillUpdate()` 方法之后，React 调用 `render()` 方法来执行 DOM 更新。然后 `componentDidUpdate()` 方法被调用。

componentDidUpdate()方法

`componentDidUpdate()` 方法在 React 更新 DOM 之后会被立即调用。它得到如下两个参数。

- `prevProps`: 上一个属性对象。
- `prevState`: 上一个状态对象。

我们将使用这个方法的操作更新后的 DOM 或者执行渲染后的操作。在 `StreamTweet` 组件中，我们将使用 `componentDidUpdate()` 在全局对象中增加已显示推文的数量。在 `componentWillUpdate()` 方法中添加如下代码：

```
componentDidUpdate: function (prevProps, prevState) {
  console.log('[Snapterest] StreamTweet: 7. Running
  componentDidUpdate()');

  window.snapterest.numberOfDisplayedTweets++;
},
```

在 `componentDidUpdate()` 被调用之后，更新阶段结束。当组件的状态更新或者父组件传来一个新属性时，一个新阶段又开始了。或者，调用 `forceUpdate()` 方法也会触发一个新的更新阶段，只是在触发更新的组件上会跳过 `shouldComponentUpdate()` 方

法。不过，在所有子组件的更新阶段，`shouldComponentUpdate()` 仍然会像往常一样被调用。实践中尽量不要用 `forceUpdate()` 方法，这样能提高应用程序的可维护性。

React 组件的生命周期方法就讨论到这里了。

设置 React 组件的默认属性

上一章讲过，`StreamTweet` 组件会渲染两个子组件：`Header` 和 `Tweet`。

现在让我们创建这两个组件。切换到 `~/snapterest/source/components/` 目录，并创建 `Header.react.js` 文件：

```
var React = require('react');

var headerStyle = {
  fontSize: '16px',
  fontWeight: '300',
  display: 'inline-block',
  margin: '20px 10px'
};

var Header = React.createClass({

  getDefaultProps: function () {
    return {
      text: 'Default header'
    };
  },

  render: function () {
    return (
      <h2 style={headerStyle}>{this.props.text}</h2>
    );
  }
});

module.exports = Header;
```


Header 组件是一个无状态组件，它渲染了 h2 元素。头部文本是通过 `this.props.text` 属性从父组件传递而来，这使得组件更灵活，允许我们在任何需要头部组件的地方重复使用它。我们将在本书的后面再一次使用这个组件。

注意这个 h2 元素有一个 `style` 属性。

在 React 中,我们可以使用 JavaScript 对象来定义 CSS 规则,然后将这个对象作为 React 组件 `style` 属性的值。比如,在这个组件中,我们定义 `headerStyle` 变量来指向一个对象:

- 对象的每个 key 作为 CSS 属性。
- 对象的每个 value 作为 CSS 属性值。

如果 CSS 属性名中包含连字符,需要将其转换为驼峰大小写格式。比如,将 `font-size` 转换为 `fontSize`, 将 `font-weight` 转换为 `fontWeight`, 等等。

将 CSS 规则定义在 React 组件中有如下一些好处。

- 移植: 可以很容易地将组件与它的样式一起分享出去, 因为它们都在一个 JavaScript 文件中。
- 封装: 将样式内联可以限制它所影响的范围。
- 灵活: 可以使用 JavaScript 来计算 CSS 规则。

使用这项技术明显的缺点是 **CSP (Content Security Policies, 内容安全策略)** 会阻止内联样式生效。关于 CSP 的更多内容请访问 https://developer.mozilla.org/en-US/docs/Web/Security/CSP/Introducing_Content_Security_Policy。

Header 组件有一个方法, 我们还没有讨论过, 它就是 `getDefaultProps()`。如果你忘记传递 React 组件依赖的属性会怎么样? 组件可以使用 `getDefaultProps()` 方法设置一个默认值, 比如:

```
getDefaultProps: function () {  
  return {  
    text: 'Default header'  
  };  
},
```

在这个例子中，我们将 `text` 属性的默认值设置为 `'Default header'`。如果父组件传递了 `this.props.text` 属性，那么默认值将会被覆盖。

接下来，让我们创建 `Tweet` 组件，切换到 `~/snapterest/source/components/` 目录，并创建 `Tweet.react.js` 文件：

```
var React = require('react');

var tweetStyle = {
  position: 'relative',
  display: 'inline-block',
  width: '300px',
  height: '400px',
  margin: '10px'
};

var imageStyle = {
  maxHeight: '400px',
  boxShadow: '0px 1px 1px 0px #aaa',
  border: '1px solid #fff'
};

var Tweet = React.createClass({

  propTypes: {

    tweet: function(properties, propertyName, componentName) {

      var tweet = properties[propertyName];

      if (! tweet) {
        return new Error('Tweet must be set.');
```

```
    },

    onImageClick: React.PropTypes.func
  },

  handleClick: function () {
    var tweet = this.props.tweet;
    var onImageClick = this.props.onImageClick;
    if (onImageClick) {
      onImageClick(tweet);
    }
  },

  render: function () {
    var tweet = this.props.tweet;
    var tweetMediaUrl = tweet.media[0].url;

    return (
      <div style={tweetStyle}>
        <img src={tweetMediaUrl} onClick={this.handleClick}
style={imageStyle} />
      </div>
    );
  }
});

module.exports = Tweet;
```

这个组件渲染一个 `div` 元素和它的子元素 `img`。两个元素都有行内样式，并且 `img` 元素还有一个点击事件处理程序，就是 `this.handleClick`：

```
handleImageClick: function () {
  var tweet = this.props.tweet;
  var onImageClick = this.props.onImageClick;

  if (onImageClick) {
    onImageClick(tweet);
  }
}
```

```
}  
},
```

当用户点击推文的图片时，Tweet 组件检查父组件是否传递了 `this.props.onImageClick` 回调函数作为属性，并调用这个函数。`this.props.onImageClick` 是一个可选的 Tweet 组件属性，所以在使用它之前，需要检查它是否被传递过来了。另一方面，`tweet` 是必要属性。

我们如何才能确保组件接收所有必要属性？

验证 React 组件的属性

在 React 中，使用组件的 `propTypes` 对象来验证组件的属性：

```
propTypes: {  
  propertyName: validator  
}
```

在这个对象中，你需要指定一个属性名和一个用来确定属性是否有效的验证器函数。React 提供了一些预置的验证器供我们使用，它们都定义在 `React.PropTypes` 对象上，如下。

- `React.PropTypes.number`：验证属性是否为数字。
- `React.PropTypes.string`：验证属性是否为字符串。
- `React.PropTypes.bool`：验证属性是否为布尔值。
- `React.PropTypes.object`：验证属性是否为对象。
- `React.PropTypes.element`：验证属性是否为 React 元素。

想要了解所有 `React.PropTypes` 验证器，可以查看 <https://facebook.github.io/react/docs/reusable-components.html#prop-validation> 上的文档。

默认情况下，使用 `React.PropTypes` 验证器验证的所有属性都是可选的。可以把任何一个属性设置为 `isRequired`，来确保当没有传递该属性的时候会在 JavaScript 控制台显示一条警告信息。

```
propTypes: {
  propertyName: React.PropTypes.number.isRequired
}
```

也可以指定自定义验证器函数，该函数需要在验证失败的时候返回一个 `Error` 对象：

```
propTypes: {
  propertyName: function (properties, propertyName, componentName) {
    // ... 验证失败
    return new Error('A property is not valid.');
```

让我们看一下 `Tweet` 组件中的 `propTypes` 对象：

```
propTypes: {

  tweet: function(properties, propertyName, componentName) {

    var tweet = properties[propertyName];

    if (! tweet) {
      return new Error('Tweet must be set.');
```

如你所见，我们验证了 `Tweet` 组件的两个属性：`tweet` 和 `onImageClick`。

我们使用自定义的验证器函数来验证 `tweet` 属性。`React` 给这个函数传递了下面三个参数。

- `properties`: 组件的属性对象。

- `propertyName`: 我们将要验证的属性名。
- `componentName`: 组件的名字。

首先, 我们检查 `Tweet` 组件有没有收到 `tweet` 属性:

```
var tweet = properties[propertyName];

if (! tweet) {
  return new Error('Tweet must be set.');
```

然后, 我们假设 `tweet` 属性是一个对象, 并检查它有没有 `media` 属性:

```
if (! tweet.media) {
  return new Error('Tweet must have an image.');
```

这些检测如果失败会返回一个 `Error` 对象并将其打印在 JavaScript 控制台上。

我们将验证组件的另一个属性 `onImageClick`:

```
onImageClick: React.PropTypes.func
```

需要验证 `onImageClick` 属性的值是否是一个函数。在这里, 我们重用了 `React.PropTypes` 对象提供的验证器函数。如你所见, `onImageClick` 是一个可选属性, 因为我们没有添加 `isRequired`。

最终, 出于性能原因, `propTypes` 仅在 `React` 的开发版中被使用。

创建 Collection 组件

前面讲过最顶层的 `Application` 组件有两个子组件: `Stream` 和 `Collection`。

到目前为止, 我们已经讨论和实现了 `Stream` 组件及其子组件。接下来, 我们将焦点放在 `Collection` 组件上。

创建 `~/snapterest/source/components/Collection.react.js` 文件:

```
var React = require('react');
```

```
var ReactDOMServer = require('react-dom/server');
var CollectionControls = require('./CollectionControls.react');
var TweetList = require('./TweetList.react');
var Header = require('./Header.react');

var Collection = React.createClass({

  createHtmlMarkupStringOfTweetList: function () {
    var htmlString = ReactDOMServer.renderToStaticMarkup(
      <TweetList tweets={this.props.tweets} />
    );

    var htmlMarkup = {
      html: htmlString
    };

    return JSON.stringify(htmlMarkup);
  },
  getListOfTweetIds: function () {
    return Object.keys(this.props.tweets);
  },
  getNumberOfTweetsInCollection: function () {
    return this.getListOfTweetIds().length;
  },
  render: function () {
    var numberOfTweetsInCollection = this.getNumberOfTweetsInCollection();

    if (numberOfTweetsInCollection > 0) {

      var tweets = this.props.tweets;
      var htmlMarkup = this.createHtmlMarkupStringOfTweetList();
      var removeAllTweetsFromCollection =
this.props.onRemoveAllTweetsFromCollection;
      var handleRemoveTweetFromCollection =
this.props.onRemoveTweetFromCollection;
```

```
return (  
  <div>  
  
    <CollectionControls  
      numberOfTweetsInCollection={numberOfTweetsInCollection}  
      htmlMarkup={htmlMarkup}  
      onRemoveAllTweetsFromCollection={removeAllTweetsFromCollection} />  
  
    <TweetList  
      tweets={tweets}  
      onRemoveTweetFromCollection={handleRemoveTweetFromCollection} />  
  
  </div>  
  );  
}  
  
return <Header text="Your collection is empty" />;  
}  
});
```

module.exports = Collection;

Collection 组件负责渲染两类内容:

- 用户已经收集的推文。
- 用于操作该集合的用户界面控件。

让我们看一下这个组件的 render() 方法:

```
render: function () {  
  var numberOfTweetsInCollection = this.  
    getNumberOfTweetsInCollection();  
  
  if (numberOfTweetsInCollection > 0) {  
  
    var tweets = this.props.tweets;  
    var htmlMarkup = this.createHtmlMarkupStringOfTweetList();
```



```
    var removeAllTweetsFromCollection = this.props.
onRemoveAllTweetsFromCollection;

    var handleRemoveTweetFromCollection = this.props.
onRemoveTweetFromCollection;

    return (
      <div>

        <CollectionControls
          numberOfTweetsInCollection={numberOfTweetsInCollection}
          htmlMarkup={htmlMarkup}
          onRemoveAllTweetsFromCollection={removeAllTweetsFromCollection} />

        <TweetList
          tweets={tweets}
          onRemoveTweetFromCollection={handleRemoveTweetFromCollection} />

      </div>
    );
  }

  return <Header text="Your collection is empty" />;
}
```

首先，我们使用 `this.getNumberOfTweetsInCollection()` 方法在集合中获得推文的总数：

```
getNumberOfTweetsInCollection: function () {
  return this.getListOfTweetIds().length;
},
```

这个方法使用了另一个方法 `this.getListOfTweetIds()` 来获得推文的 ID 列表：

```
getListOfTweetIds: function () {
  return Object.keys(this.props.tweets);
},
```

调用 `this.getListOfTweetIds()` 函数会返回一个推文 ID 的数组，然后

`this.getNumberOfTweetsInCollection()` 返回这个数组的长度。

在 `render()` 方法中知道了收集到的推文数量后，需要做出一个选择：

- 如果集合不为空，那么渲染 `CollectionControls` 和 `TweetList` 组件。
- 否则，渲染 `Header` 组件。

这些组件渲染什么内容呢？

- `CollectionControls` 组件渲染集合名标题，以及一组允许用户改名字、清空和导出集合的按钮。
- `TweetList` 组件渲染一个推文列表。
- `Header` 组件渲染一个标题，包含一条集合为空的信息。

我们的想法是当集合不为空时仅显示集合。在这个例子中，我们创建了四个变量：

```
var tweets = this.props.tweets;
var htmlMarkup = this.createHtmlMarkupStringOfTweetList();
var removeAllTweetsFromCollection = this.props.
  onRemoveAllTweetsFromCollection;
var handleRemoveTweetFromCollection = this.props.
  onRemoveTweetFromCollection;
```

- `tweets` 变量引用从父组件传递过来的 `tweets` 对象。
- `htmlMarkup` 变量引用一个通过组件的 `this.createHtmlMarkupStringOfTweetList()` 方法返回的字符串。
- `removeAllTweetsFromCollection` 和 `handleRemoveTweetFromCollection` 变量引用从父组件传递过来的函数。

顾名思义，`this.createHtmlMarkupStringOfTweetList()` 方法通过 `TweetList` 组件生成的代表 `TweetList` 的 HTML 代码创建字符串：

```
createHtmlMarkupStringOfTweetList: function () {
  var htmlString = ReactDOMServer.renderToStaticMarkup(
    <TweetList tweets={this.props.tweets} />
  );
};
```

```
var htmlMarkup = {
  html: htmlString
};

return JSON.stringify(htmlMarkup);
},
```

`createHtmlMarkupStringOfTweetList()` 方法使用了第 2 章介绍过的 `ReactDOMServer.renderToStaticMarkup()` 方法, 我们传递 `TweetList` 组件作为它的参数:

```
var htmlString = ReactDOMServer.renderToStaticMarkup(
  <TweetList tweets={this.props.tweets} />
);
```

`TweetList` 组件有一个 `tweets` 属性, 它引用了通过父组件传递来的 `tweets` 属性。

由 `ReactDOMServer.renderToStaticMarkup()` 函数生成的 HTML 字符串保存在 `htmlString` 变量中。然后, 我们创建一个新 `htmlMarkup` 对象, 它的 `html` 属性引用 `htmlString` 变量。最后, 使用 `JSON.stringify()` 函数将 `htmlMarkup` 对象转换成 JSON 字符串。`createHtmlMarkupStringOfTweetList()` 函数的返回结果就是 `JSON.stringify(htmlMarkup)` 的结果。

这个方法展示了 React 组件的灵活性, 既可以使用相同的 React 组件来渲染 DOM 元素, 也可以生成 HTML 字符串传递给第三方 API。

另外一个有趣的发现是, 可以在 `render()` 方法之外使用 JSX 语法。事实上, 你可以在源码文件的任何地方使用 JSX 语法, 甚至在 `React.createClass` 函数外也可以。

让我们仔细看一下当集合不为空时 `Collection` 组件返回的是什么:

```
return (
  <div>

    <CollectionControls
      numberOfTweetsInCollection={numberOfTweetsInCollection}
      htmlMarkup={htmlMarkup}
      onRemoveAllTweetsFromCollection={removeAllTweetsFromCollection} />
```

```
<TweetList
  tweets={tweets}
  onRemoveTweetFromCollection={handleRemoveTweetFromCollection} />

</div>
);
```

因为 React 只允许有一个根元素，所以我们用 `div` 元素包裹 `CollectionControls` 和 `TweetList` 组件。让我们看一下每个组件并讨论一下它们的属性。

我们将以下三个属性传递给 `CollectionControls` 组件：

- `numberOfTweetsInCollection` 属性表示当前集合中推文的总数。
- `htmlMarkup` 属性表示一个 HTML 标记字符串，这个字符串是在组件中使用 `createHtmlMarkupStringOfTweetList` 方法生成的。
- `onRemoveAllTweetsFromCollection` 属性引用一个函数，用来删除集合中的所有推文。这个函数在第 4 章讨论应用程序组件的时候实现过了。

我们给 `TweetList` 组件传递两个属性：

- `tweets` 属性引用从父级应用程序组件传递来的 `tweets`。
- `onRemoveTweetFromCollection` 属性引用一个函数，用来删除存储在应用程序 `state` 中的一条推文。我们在第 4 章也已经讨论过了。

这就是 `Collection` 组件。

小结

在这一章，我们学习了组件生命期的更新方法，还讨论了怎么验证组件的属性和设置组件属性的默认值。我们的 `Snapterest` 应用也有了良好的进展，我们创建并讨论了 `Header`、`Tweet` 和 `Collection` 组件。

下一章会将焦点放在更复杂的 React 组件的构建上，并将完成我们的 `Snapterest` 应用。

7

构建复杂的 React 组件

在这一章中，我们将把之前学到的和 React 组件相关的所有东西用起来，以构建我们的应用程序中最复杂的组件，也就是作为 Collection 组件的子组件。这一章的目的是帮你形成坚实的 React 开发经验，并巩固 React 的基础。让我们开始吧！

创建 TweetList 组件

我们知道，Collection 组件拥有两个子组件：CollectionControls 和 TweetList。

我们首先来构建 TweetList 组件，创建~/snapterest/source/components/TweetList.react.js 文件，并输入下面的内容：

```
var React = require('react');
var Tweet = require('./Tweet.react.js');

var listStyle = {
  padding: '0'
};

var listItemStyle = {
  display: 'inline-block',
  listStyle: 'none'
};
```

```
var TweetList = React.createClass({
  getListOfTweetIds: function () {
    return Object.keys(this.props.tweets);
  },

  getTweetElement: function (tweetId) {
    var tweet = this.props.tweets[tweetId];
    var handleRemoveTweetFromCollection = this.props.
onRemoveTweetFromCollection;
    var tweetElement;

    if (handleRemoveTweetFromCollection) {
      tweetElement = (
        <Tweet
          tweet={tweet}
          onImageClick={handleRemoveTweetFromCollection} />
      );
    } else {
      tweetElement = <Tweet tweet={tweet} />;
    }

    return <li style={listItemStyle} key={tweet.id}>{tweetElement}</li>;
  },

  render: function () {
    var tweetElements = this.getListOfTweetIds().map(this.getTweetElement);

    return (
      <ul style={listStyle}>
        {tweetElements}
      </ul>
    );
  }
});

module.exports = TweetList;
```

TweetList 组件调用 render 方法来渲染一个推文的列表：

```
render: function () {
  var tweetElements = this.getListOfTweetIds().map(this.getTweetElement);

  return (
    <ul style={listStyle}>
      {tweetElements}
    </ul>
  );
}
```

首先，我们创建一个 Tweet 元素的列表：

```
var tweetElements = this.getListOfTweetIds().map(this.getTweetElement);

this.getListOfTweetIds() 方法返回一个推文 ID 列表的数组：

getListOfTweetIds: function () {
  return Object.keys(this.props.tweets);
},
```

然后，我们对这个数组中的每一个推文 ID 创建 Tweet 组件。在这里我们调用推文 ID 数组的 map() 方法，并将 this.getTweetElement 方法作为回调函数传入：

```
getTweetElement: function (tweetId) {
  var tweet = this.props.tweets[tweetId];
  var handleRemoveTweetFromCollection = this.props.
onRemoveTweetFromCollection;
  var tweetElement;

  if (handleRemoveTweetFromCollection) {
    tweetElement = (
      <Tweet
        tweet={tweet}
        onImageClick={handleRemoveTweetFromCollection} />
    );
  } else {
    tweetElement = <Tweet tweet={tweet} />;
  }
}
```



```
    }

    return <li style={listItemStyle} key={tweet.id}>{tweetElement}</li>;
  },
```

`getTweetElement()` 方法返回一个被 `` 元素包裹的 `Tweet` 元素, 我们知道, `Tweet` 组件有一个可选的 `onImageClick` 属性。什么时候需要调用这个可选的属性, 什么时候又不需要调用?

这里分为两种情形。第一种情形, 用户需要点击推文的图片以将这条推文从推文的集合中移除, 在这种情形下, `Tweet` 组件就需要对 `click` 事件做出响应, 所以我们就需要使用 `onImageClick` 这个属性。第二种情形, 用户将导出一个静态的推文集合, 没有任何用户交互。在这种情形下, 我们就不需处理 `onImageClick` 这个属性了。

下面就是我们在 `getTweetElement()` 方法中所做的事情:

```
var tweet = this.props.tweets[tweetId];
var handleRemoveTweetFromCollection = this.props.
  onRemoveTweetFromCollection;
var tweetElement;

if (handleRemoveTweetFromCollection) {
  tweetElement = (
    <Tweet
      tweet={tweet}
      onImageClick={handleRemoveTweetFromCollection} />
  );
} else {
  tweetElement = <Tweet tweet={tweet} />;
}
```

我们创建了一个 `tweet` 变量用以保存根据 `ID` 获取的推文信息, `ID` 是由参数 `tweetId` 提供的。然后, 我们创建了一个变量来保存通过父组件 `Collection` 传递下来的 `this.props.onRemoveTweetFromCollection` 属性。

接下来, 我们检查 `Collection` 组件的 `this.props.onRemoveTweetFromCollection` 属性是否存在。如果存在, 就创建一个带有 `onImageClick` 属性的 `Tweet`

组件：

```
tweetElement = (
  <Tweet
    tweet={tweet}
    onClick={handleRemoveTweetFromCollection} />
);
```

如果不存在，我们就创建一个不带 `handleImageClick` 属性的 `Tweet` 组件：

```
tweetElement = <Tweet tweet={tweet} />;
```

我们在下面这两种情况下来使用 `TweetList` 组件：

- 用 `TweetList` 组件渲染 `Collection` 组件中推文的集合，在这种情况下，就会有 `onRemoveTweetFromCollection`。
- 要用它将 `Collection` 组件中推文的集合渲染为 HTML 字符串，在这种情况下，就没有 `onRemoveTweetFromCollection`。

一旦生成了 `Tweet` 元素，并将它赋值给 `tweetElement` 变量后，我们就返回带有行内样式的 `` 元素：

```
return <li style={listItemStyle} key={tweet.id}>{tweetElement}</li>;
```

除了 `style` 属性外，`` 元素还有一个 `key` 的属性，`React` 通过它来确定每个动态创建的子元素。如果想了解更多关于动态子元素的信息，请访问 <https://facebook.github.io/react/docs/multiple-components.html#dynamic-children>。

这就是 `getTweetElement()` 方法的工作方式。在最后，`TweetList` 组件会返回一个无序的 `Tweet` 元素列表：

```
return (
  <ul style={listStyle}>
    {tweetElements}
  </ul>
);
```

创建 CollectionControls 组件

明白了 Collection 组件渲染什么以后，接下来我们就看看它的子组件。首先来看 CollectionControls，创建~/snapterest/source/components/CollectionControls.react.js 文件，内容如下：

```
var React = require('react');
var Header = require('./Header.react');
var Button = require('./Button.react');
var CollectionRenameForm = require('./CollectionRenameForm.react');
var CollectionExportForm = require('./CollectionExportForm.react');

var CollectionControls = React.createClass({
  getInitialState: function () {
    return {
      name: 'new',
      isEditingName: false
    };
  },

  getHeaderText: function () {
    var numberOfTweetsInCollection = this.props.
numberOfTweetsInCollection;
    var text = numberOfTweetsInCollection;

    if (numberOfTweetsInCollection === 1) {
      text = text + ' tweet in your';
    } else {
      text = text + ' tweets in your';
    }

    return (
      <span>
        {text} <strong>{this.state.name}</strong> collection
      </span>
    );
  },
},
```

```
toggleEditCollectionName: function () {
  this.setState({
    isEditingName: !this.state.isEditingName
  });
},

setCollectionName: function (name) {
  this.setState({
    name: name,
    isEditingName: false
  });
},

render: function () {

  if (this.state.isEditingName) {
    return (
      <CollectionRenameForm
        name={this.state.name}
        onChangeCollectionName={this.setCollectionName}
        onCancelCollectionNameChange={this.toggleEditCollectionName}
      />
    );
  }

  return (
    <div>
      <Header text={this.getHeaderText()} />

      <Button
        label="Rename collection"
        handleClick={this.toggleEditCollectionName} />

      <Button
        label="Empty collection"
        handleClick={this.props.onRemoveAllTweetsFromCollection} />
    </div>
  );
}
```

```
        <CollectionExportForm htmlMarkup={this.props.htmlMarkup} />
      </div>
    );
  }
});

module.exports = CollectionControls;
```

CollectionControls 组件渲染用户界面上的控件，这个组件允许用户：

- 重命名集合
- 清空集合
- 导出集合

每个集合都有一个名称，默认新建的集合名称为 `new`，用户可以随意更改。集合的名称是通过 CollectionControls 组件渲染的，显示在组件的标题上。使用这个组件来存储集合的名称是一个不错的选择。因为改变名称需要组件重新渲染，所以我们将集合的名称存储在组件的 `state` 对象上。

```
getInitialState: function () {
  return {
    name: 'new',
    isEditingName: false
  };
},
```

CollectionControls 组件可以渲染操作集合的控件，也可以渲染更改集合名称的表单，用户可以在这两者之间切换。我们需要一种方式来区分这两种状态，我们使用 `isEditingName` 这个属性来区分它们。默认情况下，`isEditingName` 的值是 `false`，因此，当 CollectionControls 组件被挂载时，用户不会看到更改集合名称的表单。我们来看看组件的 `render()` 方法：

```
render: function () {

  if (this.state.isEditingName) {
    return (
```

```

    <CollectionRenameForm
      name={this.state.name}
      onChangeCollectionName={this.setCollectionName}
      onCancelCollectionNameChange={this.toggleEdit
CollectionName}
    />
  );
}

return (
  <div>
    <Header text={this.getHeaderText()} />

    <Button
      label="Rename collection"
      handleClick={this.toggleEditCollectionName} />

    <Button
      label="Empty collection"
      handleClick={this.props.onRemoveAllTweetsFromCollection} />

    <CollectionExportForm htmlMarkup={this.props.htmlMarkup} />
  </div>
);
}

```

首先检查组件的状态属性 `this.state.isEditingName` 是否被设置为了 `true`。如果是，那么 `CollectionControls` 组件返回 `CollectionRenameForm` 组件，用来渲染更改集合名称的表单：

```

<CollectionRenameForm
  name={this.state.name}
  onChangeCollectionName={this.setCollectionName}
  onCancelCollectionNameChange={this.toggleEditCollectionName} />

```

`CollectionRenameForm` 用来渲染一个更改集合名称的表单，它接受下面三个属性：

- name 属性记录当前集合的名称。
- onChangeCollectionName 属性和 onCancelCollectionNameChange 属性会记录组件的事件方法。

本章后面将实现 CollectionRenameForm 组件。现在我们来仔细看看 setCollectionName 方法：

```
setCollectionName: function (name) {  
  this.setState({  
    name: name,  
    isEditingName: false  
  });  
},
```

这个方法通过设置 isEditingName 属性当前的值来显示或者隐藏更改集合名称的表单，同时修改集合的名称。在用户提交新集合名称时，需要调用这个方法。

下面再看看 toggleEditCollectionName 方法：

```
toggleEditCollectionName : function(){  
  this.setState({  
    isEditingName : !this.state.isEditingName  
  });  
},
```

这里通过!操作符对当前布尔值取反并设置 isEditingName 属性，从而显示或隐藏集合的名字。我们会在用户点击 **Rename collection**（重命名集合）或 **Cancel**（取消）按钮时调用这个方法。

如果 CollectionControls 组件将属性 this.state.isEditingName 的值设置为 false，就会返回集合控件：

```
return (  
  <div>  
    <Header text={this.getHeaderText()} />  
  
    <Button  
      label="Rename collection"
```

```

        handleClick={this.toggleEditCollectionName} />
      <Button
        label="Empty collection"
        handleClick={this.props.onRemoveAllTweetsFromCollection} />

      <CollectionExportForm htmlMarkup={this.props.htmlMarkup} />
    </div>
  );

```

这里将 Header 组件、两个 Button 组件、CollectionExportForm 组件包裹到一个 div 元素里面。上一章就用到 Header 组件了，它接收一个字符串类型的 text 属性。但是这里没有直接传入字符串，而是调用了 this.getHeaderText() 方法：

```
<Header text={this.getHeaderText()} />
```

接着，this.getHeaderText() 方法返回一个字符串，让我们仔细看看 this.getHeaderText() 这个方法：

```

getHeaderText: function () {
  var numberOfTweetsInCollection = this.props.numberOfTweetsInCollection;
  var text = numberOfTweetsInCollection;

  if (numberOfTweetsInCollection === 1) {
    text = text + ' tweet in your';
  } else {
    text = text + ' tweets in your';
  }

  return (
    <span>
      {text} <strong>{this.state.name}</strong> collection
    </span>
  );
},

```

这个方法基于集合里推文的数量而生成一个字符串。重要的是，这个方法不是返回一个字符串，而是返回 React 元素树，里面封装了字符串。首先，创建 numberOfTweetsInCollection 变量，存储集合中推文的数量。然后创建 text 变量，并将推文数量赋值给它。这时候，

text 变量存储了一个整数值。下一个任务是拼接正确的字符串，基于下面的规则：

- 如果 numberOfTweetsInCollection 变量的值为 1，我们需要给它拼接上 'tweet in your'。
- 否则，给它拼接上 'tweets in your'。

创建了标题的字符串后，返回下面的元素：

```
return (  
  <span>  
    {text} <strong>{this.state.name}</strong> collection  
  </span>  
);
```

最后将 text 变量的值、集合的名称及 collection 拼接起来，包裹进一个 元素中，运行后会返回类似下面的结果：

```
1 tweet in your new collection.
```

getHeaderText() 返回的字符串可以作为属性传递给 Header 组件。CollectionControls 组件 render() 方法中的下一个控件是 Button：

```
<Button  
  label="Rename collection"  
  handleClick={this.toggleEditCollectionName} />
```

我们将 Rename collection(重命名集合)作为组件的 label 属性，将 this.toggleEditCollectionName 方法作为组件的 handleClick 属性。因此，这个按钮将有 Rename collection(重命名集合)标签，并可以切换出更改集合名称的表单。

下一个控件仍然是 Button 组件：

```
<Button  
  label="Empty collection"  
  handleClick={this.props.onRemoveAllTweetsFromCollection} />
```

估计你已经猜到了，这个组件有一个 Empty collection(清空集合)的标签，可以从集合中移除所有的推文。

最后一个控件是 `CollectionExportForm` 组件：

```
<CollectionExportForm htmlMarkup={this.props.htmlMarkup} />
```

这个组件接收一个代表推文集合的 HTML 字符串，渲染一个按钮。我们将在本章后面创建这个组件。

现在，我们已经理解了 `CollectionControls` 组件渲染什么了。接下来介绍它的子组件，我们从 `CollectionRenameForm` 组件开始。

创建 `CollectionRenameForm` 组件

首先，我们创建 `~/snapterest/source/components/CollectionRenameForm.react.js` 文件，内容如下：

```
var React = require('react');
var ReactDOM = require('react-dom');
var Header = require('./Header.react');
var Button = require('./Button.react');

var inputStyle = {
  marginRight: '5px'
};

var CollectionRenameForm = React.createClass({

  getInitialState: function() {
    return {
      inputValue: this.props.name
    };
  },

  setInputValue: function (inputValue) {
    this.setState({
      inputValue: inputValue
    });
  },
```

```
handleInputChange: function (event) {
  var inputValue = event.target.value;

  this.setInputValue(inputValue);
},

handleFormSubmit: function (event) {
  event.preventDefault();
  var collectionName = this.state.inputValue;
  this.props.onChangeCollectionName(collectionName);
},

handleFormCancel: function (event) {
  event.preventDefault();
  var collectionName = this.props.name;
  this.setInputValue(collectionName);
  this.props.onCancelCollectionNameChange();
},

componentDidMount: function () {
  this.refs.collectionName.focus();
},

render: function () {
  return (
    <form className="form-inline" onSubmit={this.handleSubmit}>

      <Header text="Collection name:" />

      <div className="form-group">
        <input
          className="form-control"
          style={inputStyle}
          onChange={this.handleInputChange}
          value={this.state.inputValue}
          ref="collectionName" />
      </div>
    </form>
  );
}
```

```

    </div>

    <Button label="Change" handleClick={this.handleFormSubmit} />
    <Button label="Cancel" handleClick={this.handleFormCancel} />
  </form>
);
}
});

```

```
module.exports = CollectionRenameForm;
```

这个组件渲染一个表单来更改集合名称：

```

render: function () {
  return (
    <form className="form-inline" onSubmit={this.handleSubmit}>

      <Header text="Collection name:" />

      <div className="form-group">
        <input
          className="form-control"
          style={inputStyle}
          onChange={this.handleInputChange}
          value={this.state.inputValue}
          ref="collectionName" />
      </div>

      <Button label="Change" handleClick={this.handleFormSubmit} />
      <Button label="Cancel" handleClick={this.handleFormCancel} />
    </form>
  );
}

```

<form>元素包裹着 4 个子元素，它们是：

- 一个 Header 组件
- 一个 <input> 元素

- 两个 Button 组件

Header 组件渲染出 'Collection name' 这个字符串。<input>元素包裹在一个 className 属性名是 form-group 的<div>元素中。这个类名是第 4 章中讨论的 Bootstrap 框架的一部分，主要用于布局和样式。

<input>元素有很多个属性，我们来仔细看看：

```
<input
  className="form-control"
  style={inputStyle}
  onChange={this.handleInputChange}
  value={this.state.inputValue}
  ref="collectionName" />
```

- className 属性的值是 form-control, 这个 class 仍然是 Bootstrap 框架的一部分，主要用于样式。
- 此外, 我们使用 style 属性将 inputStyle 对象传入以自定义 input 的样式, inputStyle 对象只有 marginRight 这一条样式规则。
- value 属性将组件状态中的 this.state.inputValue 值设置为当前值。
- onChange 属性引入 this.handleInputChange 方法, 这个方法会对 onchange 做出处理。
- ref 是一个特殊的属性, 可以将它添加到 render() 返回的任意组件上。通过它可以在 render() 方法外边引用该组件。很快我们就会看到这样一个例子。

我们最需要关注的是最后三个属性: value、onChange 和 ref。value 是组件 state 的属性, 更新组件的状态是更改这个值的唯一方式。我们知道, 用户输入可以改变输入框的值, 这种行为适用于我们的组件吗? 不适用, 用户输入时, 输入框的值不会改变。这是因为<input>是通过组件而不是用户来控制的。在 CollectionRenameForm 组件中, <input>总是反映 this.state.inputValue 属性的值, 无论用户输入什么。总之, 值的变化由 CollectionRenameForm 组件控制, 而不是由用户控制。

那么, 怎样才能让输入框对用户的输入做出反应呢? 我们需要监听用户的输入, 然后更新 CollectionRenameForm 组件的状态, 接着用新的值重新渲染输入框。这样做可以让每一个输入框的 change 事件看起来像平常一样正常工作, 用户可以随意更改输入框

的值。

为此，我们需要`<input>`元素提供的 `onChange` 属性触发组件的 `this.handleChange` 方法：

```
handleInputChange: function (event) {  
  var inputValue = event.target.value;  
  this.setInputValue(inputValue);  
},
```

第 3 章讨论过，**React** 将 `SyntheticEvent` 的实例传递给事件处理程序。`handleInputChange()` 方法接收带有一个 `target` 属性的 `event` 对象，这个 `target` 属性有一个 `value` 属性。这个 `value` 属性是一个字符串，用来存储用户在输入框输入的值。我们通过 `this.setInputValue()` 方法来处理这个字符串：

```
setInputValue: function (inputValue) {  
  this.setState({  
    inputValue: inputValue  
  });  
},
```

`setInputValue()` 是一个辅助方法，它用新输入值来更新组件的状态。之后，组件会用新的值来渲染`<input>`元素。

`CollectionRenameForm` 组件刚开始挂载时，输入框的初始值是什么？让我们看看下面这段代码：

```
getInitialState: function() {  
  return {  
    inputValue: this.props.name  
  };  
},
```

正如你所看到的，我们将父组件集合的名称传入，并将这个名称设置为输入框的初始值。

组件挂载之后，我们想要将输入焦点设置为这个输入框，以使用户编辑集合名称。我们知道，一旦一个组件被插入到 **DOM** 中，**React** 就会调用 `componentDidMount()` 方法。

在这个方法中设置焦点是比较好的方式：

```
componentDidMount: function () {  
  this.refs.collectionName.focus();  
},
```

为此，我们就在这个输入框上调用了 `focus()` 方法。

在 `componentDidMount()` 方法里怎么引用元素呢？可以使用 `this.refs` 对象来引用 `input` 元素。因为前面我们为 `input` 元素设置了 `ref` 属性，属性值设置为了 `collectionName`，所以就可以通过 `this.refs.collectionName` 来引用这个元素了。

最后，我们来讨论两个表单按钮：

- `Change` 按钮提交表单并修改集合名称。
- `Cancel` 按钮提交表单但不会更改集合名称。

先看 `Change` 按钮：

```
<Button label="Change" handleClick={this.handleFormSubmit} />
```

当用户点击按钮的时候，`this.handleFormSubmit` 方法会被调用：

```
handleFormSubmit: function (event) {  
  event.preventDefault();  
  
  var collectionName = this.state.inputValue;  
  this.props.onChangeCollectionName(collectionName);  
},
```

这个方法阻止了默认的 `submit` 事件，然后从组件状态中取出输入的集合名称，将它传递给 `this.props.onChangeCollectionName()` 函数，`onChangeCollectionName()` 函数是由父级组件 `CollectionControls` 传递过来的，调用这个函数就会更改集合的名称。

再来看第二个表单按钮 `Cancel`：

```
<Button label="Cancel" handleClick={this.handleFormCancel} />
```

当用户点击这个按钮时，`this.handleFormCancel` 方法就会被调用：

```
handleFormCancel: function (event) {  
  event.preventDefault();  
  
  var collectionName = this.props.name;  
  this.setInputValue(collectionName);  
  this.props.onCancelCollectionNameChange();  
},
```

这里我们仍然阻止掉默认的 `submit` 事件，然后从父组件 `CollectionControls` 传递下来的属性中取得集合原来的名称，并将这个名称传递给 `this.setInputValue()` 函数调用，然后调用 `this.props.onCancelCollectionNameChange()` 方法将表单隐藏。

这就是我们的 `CollectionRenameForm` 组件。接下来创建 `Button` 组件，我们已经在 `CollectionRenameForm` 组件中使用它两次了。

创建 Button 组件

创建 `~/snapterest/source/components/Button.react.js` 文件，内容如下：

```
var React = require('react');  
  
var buttonStyle = {  
  margin: '10px 10px 10px 0'  
};  
  
var Button = React.createClass({  
  render: function () {  
    return (  
      <button  
        className="btn btn-default"  
        style={buttonStyle}  
        onClick={this.props.handleClick}>{this.props.label}</button>  
    );  
  }  
});
```



```
});
```

```
module.exports = Button;
```

Button 组件会渲染按钮。为什么按钮也要单独创建一个组件,而不是使用<button>元素?这么做有什么好处?可以把组件想象成一个包装了<button>元素和其他相关东西的一个包裹。在我们的例子中,大多数的<button>元素有相同的样式,因此将<button>元素和样式对象封装在一个组件中可以重用。所以这里使用 Button 组件。这个组件预计接收来自父组件的两个属性:

- label 属性是按钮的标签。
- handleClick 属性是一个回调函数,在用户点击按钮的时候会被调用。

现在,我们要创建 CollectionExportForm 组件。

创建 CollectionExportForm 组件

CollectionExportForm 组件用于将集合导出到一个第三方网站上(<http://CodePen.io>)。将推文集合导出到 CodePen 后,可以保存它或者与朋友分享它。让我们来看看这是如何做到的。

创建 ~/snapterest/source/components/CollectionExportForm.react.js 文件:

```
var React = require('react');

var formStyle = {
  display: 'inline-block'
};

var CollectionExportForm = React.createClass({
  render: function () {
    return (
      <form action="http://codepen.io/pen/define" method="POST"
        target="_blank" style={formStyle}>
```

```

        <input type="hidden" name="data" value={this.props.
htmlMarkup}/>
        <button type="submit" className="btn btn-default">Export as
HTML</button>
      </form>
    );
  }
});

module.exports = CollectionExportForm;

```

CollectionExportForm 组件渲染出一个包含<input>和<button>元素的表单。<input>元素是隐藏的，并且它的值被设置为了将要导出的 HTML 字符串，这个字符串是从父级组件传递下来的，<button>元素是用户在这个组件上唯一可以看到的，当用户点击 **Export as HTML** 按钮时，集合就会被提交到新窗口中打开的 CodePen.io 的页面，接着用户就可以修改和分享这个集合了。

祝贺你！现在你已经完成了一个功能齐全的 React Web 应用程序了，让我们看看怎么运行它。

首先，确认运行了第 1 章安装配置的 Snapkite 引擎，然后跳转到~/snapkite-engine/目录，并执行下面的命令：

```
npm start
```

接着，打开一个新的终端窗口，进入~/snapterest/目录，执行下面的命令：

```
gulp
```

现在，在浏览器中打开~/snapterest/build/index.html 文件，你可以看到出现了新的推文，点击将它们加到你的集合中，再次点击从集合中移除个别推文；点击 **Empty collection** 按钮将所有的推文从集合中移除；点击 **Rename collection** 按钮，输入一个新的名称，然后点击 **Change** 按钮；最后，点击 **Export as HTML** 按钮将推文集合导出到 CodePen.io。如果你对这一章或者前面某一章有疑问，访问 <https://github.com/fedosejev/react-essentials> 提交一个新的 issue 吧。

小结

在这一章，我们创建了 TweetList、CollectionControls、CollectionRenameForm、CollectionExportForm 和 Button 组件，并使用这些组件完成了一个功能齐全的 React 应用程序的构建。在下一章，我们将使用 Jest 来做测试，使用 Flux 来增强它。

8

使用 Jest 来测试 React 应用程序

现在我们已经创建了许多 React 组件，其中一些比较简单，另一些比较复杂。创建这些组件让我们获得了一定的自信，相信无论怎样自己都可以使用 React 构建复杂的用户界面，而且没有重大缺陷。有这样的自信很好，也是我们花时间来学习 React 的目的。但是，许多自信的 React 开发人员都会进入一个误区：不写单元测试。

什么是单元测试？顾名思义，单元测试就是用来测试应用程序的一个单元。在应用程序中，一个单元通常是一个函数，这意味着编写单元测试主要是为函数编写测试。

为什么要写单元测试

为什么应该编写单元测试呢？我想讲述一个自己亲身经历的故事。最近我创建的一个新网站上线了。几天过后，一位使用该网站的同事发给我一封电子邮件，邮件带有几个文件，都是会被网站拒绝的。我仔细检查了文件，要求两方 ID 匹配的条件也满足，然而文件还是被拒绝了，错误消息说 ID 不匹配。你能猜出是什么问题吗？

我写了一个函数，检查两个文件的 ID 是否匹配。该函数会检查 ID 值和 ID 类型，因此，如果 ID 值相同而 ID 类型不同，函数也会返回不匹配。事实证明，我同事的文件正好是这种情况。重要的是我怎么能阻止这种事发生呢？答案就是为函数写单元测试。

创建测试套件、规范和期望

怎么针对 JavaScript 函数编写一个测试呢？我们需要一个测试框架，幸运的是，

Facebook 拥有自己的针对 JavaScript 的单元测试框架 **Jest**。它是在另一个著名的 JavaScript 测试框架 **Jasmine** 上演变开发而来的。那些熟悉 Jasmine 的开发者会发现 Jest 的测试方法非常相似。不过在这里，我先假设你并没有测试框架的相关经验，所以先从讨论基础开始。

单元测试的基本思想是只测试应用程序中的某个功能，通常这个功能由一个函数实现。你需要单独测试这个函数，这意味着应用程序中这个函数依赖的其他部分在测试中是用不到的。相反，它们需要在测试中被模拟。模拟一个 JavaScript 对象就是创建一个假对象，来模拟实际对象的行为。在单元测试中，这个假对象叫作**模拟对象（mock）**，创建它的过程叫作**模拟（mocking）**。

当运行测试时，Jest 会自动模拟依赖关系。它会自动发现库中的测试并执行。让我们来看看下面的例子。

首先，创建~/snapterest/source/utils/目录，然后在这个目录下新建一个 TweetUtils.js 文件：

```
function getListOfTweetIds(tweets) {  
  return Object.keys(tweets);  
}  
  
module.exports.getListOfTweetIds = getListOfTweetIds;
```

TweetUtils.js 是我们应用中使用的一个模块，它有一个 getListOfTweetIds() 辅助函数。这个函数接收一个推文的对象，返回推文 ID 的列表。遵照 CommonJS 的模块规范，我们将这个函数导出。

```
module.exports.getListOfTweetIds = getListOfTweetIds;
```

现在我们来编写第一个 Jest 单元测试，这个单元测试用来测试 getListOfTweetIds() 函数。

创建一个新目录：~/snapterest/source/utils/__tests__/. Jest 会扫描目录结构，并执行 __tests__ 目录下的所有测试。因此，把目录命名为 __tests__ 很重要。

在 __tests__ 目录下创建 TweetUtils-test.js 文件：

```
jest.dontMock('../TweetUtils');
```

```
describe('Tweet utilities module', function () {

  it('returns an array of tweet ids', function () {

    var TweetUtils = require('../TweetUtils');
    var tweetsMock = {
      tweet1: {},
      tweet2: {},
      tweet3: {}
    };
    var expectedListOfTweetIds = [ 'tweet1', 'tweet2', 'tweet3' ];
    var actualListOfTweetIds = TweetUtils.
      getListOfTweetIds(tweetsMock);

    expect(actualListOfTweetIds).toEqual(expectedListOfTweetIds);
  });
});
```

首先，我们需要告诉 Jest 不去模拟 TweetUtils 模块：

```
jest.dontMock('../TweetUtils');
```

因为 Jest 会自动模拟 require() 函数引用的模块，所以这里需要我们手动执行。在测试中引入 TweetUtils 模块：

```
var TweetUtils = require('../TweetUtils');
```

如果没有 jest.dontMock('../TweetUtils') 调用，Jest 会返回 TweetUtils 模块的模拟数据，而不是真正的模块。而在我们的例子中，是需要真正的 TweetUtils 模块的，因为它正是我们需要测试的函数。

接下来，我们调用一个全局的 Jest 函数 describe()。理解它背后的概念是很重要的。在 TweetUtils-test.js 文件中，我们不仅是创建一个单独的测试，而且会创建一个测试套件。测试套件是一组测试的集合，用来测试一个更大的功能单元。例如，一个测试套件可以包含多个测试，每个测试都是这个大模块的一部分。在我们的例子中有一个包含了功能函数的 TweetUtils 模块。在这种情况下，我们将为 TweetUtils 模块创建测试套件，然后为每个函数创建单元测试，例如 getListOfTweetIds() 函数。

`describe()` 函数定义了一个测试套件，这个函数接收下面两个参数。

- 套件名称：该名称描述了即将被测试的功能，在这里就是 'Tweet utilities module' 测试集。
- 套件实现：实现该套件的函数。

在我们的例子中，测试套件是这样的：

```
describe('Tweet utilities module', function () {  
  // 这里是测试套件的实现……  
});
```

如何创建一个单独的测试？在 Jest 中，一个单独的测试称为**测试规范 (specs)**。测试规范需要调用另一个 Jest 的全局函数 `it()` 来定义，和 `describe()` 函数类似，`it()` 函数也接收两个参数，如下。

- 测试规范名称：这个名称描述了当前测试规范的功能，在这里类似于 'returns an array of tweet ids' 测试规范。
- 测试规范的实现：这里是一个实现测试规范的函数。

在这个例子中，测试规范是这样的：

```
it('returns an array of tweet ids', function () {  
  // 测试规范的实现……  
});
```

让我们仔细看看测试规范的实现：

```
var TweetUtils = require('../TweetUtils');  
var tweetsMock = {  
  tweet1: {},  
  tweet2: {},  
  tweet3: {}  
};  
var expectedListOfTweetIds = [ 'tweet1', 'tweet2', 'tweet3' ];  
var actualListOfTweetIds = TweetUtils.getListOfTweetIds(tweetsMock);  
  
expect(actualListOfTweetIds).toEqual(expectedListOfTweetIds);
```

这个测试规范测试了 TweetUtils 模块的 `getListOfTweetIds()` 方法在接收到推文对象后是否可以返回推文的 ID 列表。

首先，我们先导入 TweetUtils 模块：

```
var TweetUtils = require('../TweetUtils');
```

接着，创建一个和真实推文类似的模拟对象：

```
var tweetsMock = {  
  tweet1: {},  
  tweet2: {},  
  tweet3: {}  
};
```

这个模拟对象的唯一要求将是推文 ID 作为对象的键。值是什么并不重要，所以我们这里直接使用空对象。键的名称也不重要，所以我们选择的名称是 `tweet1`、`tweet2` 和 `tweet3`。这个模拟对象是不完全模拟实际推文对象的，它的唯一目的是用自己的键来模拟推文 ID。

下一步是创建推文 ID 的预期列表：

```
var expectedListofTweetIds = [ 'tweet1', 'tweet2', 'tweet3' ];
```

我们知道预期的推文 ID 列表是怎样的，因为我们用相同的 ID 模拟了 `tweets` 对象。

再下一步是从模拟的 `tweets` 对象中提取实际的推文 ID。为此，我们使用 `getListOfTweetIds()` 方法，它接收 `tweets` 对象，并返回推文 ID 的数组：

```
var actualListofTweetIds = TweetUtils.getListOfTweetIds(tweetsMock);
```

我们将 `tweetsMock` 对象传入这个方法，并将返回值赋给 `actualListofTweetIds` 变量。之所以将变量命名为 `actualListofTweetIds`，是因为这个推文的列表是由我们测试的实际的 `getListOfTweetIds()` 函数返回的。

最后一步涉及一个重要的新概念：

```
expect(actualListofTweetIds).toEqual(expectedListofTweetIds);
```

让我们思考一下测试的过程。我们需要一个从被测试方法 `getListOfTweetIds()`

返回的实际值，去匹配我们提前知道的期望值，匹配的结果将决定测试是通过还是失败。

为什么可以猜测到 `getListOfTweetIds()` 的返回值？因为我们已经提前准备好了输入值。这是我们的模拟对象：

```
var tweetsMock = {  
  tweet1: {},  
  tweet2: {},  
  tweet3: {}  
};
```

因此，它对于调用 `TweetUtils.getListOfTweetIds(tweetsMock)` 函数，我们可以预期的输出是：

```
[ 'tweet1', 'tweet2', 'tweet3' ]
```

因为 `getListOfTweetIds()` 函数里面的执行可能会出错，所以我们不能保证返回结果正确，我们只能期望它正确。

这就是为什么我们需要创建一个期望值。在 Jest 里，这个期望是使用一个带有实际值（例如 `actualListOfTweetIds` 对象）的 `expect()` 函数表示的：`expect(actualListOfTweetIds)`。

然后用链式调用一个匹配器函数比较实际值和期望值，告诉 Jest 是否满足期望：

```
expect(actualListOfTweetIds).toEqual(expectedListOfTweetIds);
```

在这个例子中，我们使用 `toEqual()` 匹配器函数去比较两个数组。要查询完整的内置匹配器列表，请访问 <https://facebook.github.io/jest/docs/api.html#expect-value>。

这就是创建测试规范的过程。测试规范包含一个或多个期望，每个期望都会测试代码的状态。测试规范可能是通过规范或失败规范。所有期望都满足是通过规范，否则就是失败规范。

好，现在我们已经完成了一个测试套件，包括一个测试规范和一个期望，那么怎么能运行它呢？

安装并运行 Jest

首先，先来安装 Jest 命令行工具（Jest CLI）：

```
npm install --save-dev jest-cli
```

这个命令会安装 Jest 命令行，并且将这个模块添加到~/snapterest/package.json 文件的开发依赖中。下一步，编辑 package.json 文件，将下面代码：

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1"  
},
```

替换为：

```
"scripts": {  
  "test": "jest"  
},
```

现在我们已经做好执行测试的准备了，切换到~/snapterest/目录，并且执行下面的命令：

```
npm test
```

你会看到终端输出下面这样的信息：

```
Using Jest CLI v0.4.18  
PASS source/utils/__tests__/TweetUtils-test.js (0.065s)  
1 test passed (1 total)  
Run time: 0.295s
```

这里使用的 Jest 命令行版本是 0.4.18，你使用的 Jest 的版本可能会比这个版本高。

信息中最关键的是下面这句话：

```
PASS source/utils/__tests__/TweetUtils-test.js (0.065s)
```

- PASS：这告诉我们测试通过了。
- source/utils/__tests__/TweetUtils-test.js：告诉我们具体运行的什么是测试。
- (0.065s)：告诉我们运行测试所花费的时间。

这些就是编写和测试一个小的单元测试的所有步骤。现在，让我们创建另一个测试。

创建更多的测试规范和期望

现在，我们要创建和测试集合工具模块。在~/snapterest/source/utils/目录下创建 CollectionUtils.js 文件，内容如下：

```
function getNumberOfTweetsInCollection(collection) {
  var TweetUtils = require('./TweetUtils');
  var listOfCollectionTweetIds =
    TweetUtils.getListOfTweetIds(collection);

  return listOfCollectionTweetIds.length;
}

function isEmptyCollection(collection) {
  return (getNumberOfTweetsInCollection(collection) === 0);
}

module.exports = {
  getNumberOfTweetsInCollection: getNumberOfTweetsInCollection,
  isEmptyCollection: isEmptyCollection
};
```

CollectionUtils 模块含有 getNumberOfTweetsInCollection() 和 isEmptyCollection() 两个方法。

我们先看看 getNumberOfTweetsInCollection() 方法：

```
function getNumberOfTweetsInCollection(collection) {
  var TweetUtils = require('./TweetUtils');
  var listOfCollectionTweetIds = TweetUtils.getListOfTweetIds
    (collection);

  return listOfCollectionTweetIds.length;
}
```

如你所见，这个函数先引入 `TweetUtils` 依赖，然后调用依赖的 `getListOfTweetIds()` 方法并传入 `collection` 对象作为参数，将 `getListOfTweetIds()` 函数返回的结果存储在数组变量 `listOfCollectionTweetIds` 中，`getNumberOfTweetsInCollection()` 方法最后返回数组的 `length`。

现在，我们再看看 `isEmptyCollection()` 方法：

```
function isEmptyCollection(collection) {  
  return (getNumberOfTweetsInCollection(collection) === 0);  
}
```

这里重用了 `getNumberOfTweetsInCollection()` 方法，检查 `getNumberOfTweetsInCollection()` 方法返回的结果是否等于零，然后返回检查结果：`true` 或 `false`。

最后，我们将这两个方法从模块中导出：

```
module.exports = {  
  getNumberOfTweetsInCollection: getNumberOfTweetsInCollection,  
  isEmptyCollection: isEmptyCollection  
};
```

`CollectionUtils` 模块创建完了，下一步来测试它。

在 `~/snapterest/source/utils/__tests__` 目录下创建 `CollectionUtils-test.js` 文件，内容如下：

```
jest.autoMockOff();  
  
describe('Collection utilities module', function () {  
  
  var CollectionUtils = require('../CollectionUtils');  
  
  var collectionTweetsMock = {  
    collectionTweet7: {},  
    collectionTweet8: {},  
    collectionTweet9: {}  
  };  
});
```

```
it('returns a number of tweets in collection', function
getNumberOfTweetsInCollection() {

    var actualNumberOfTweetsInCollection = CollectionUtils.
getNumberOfTweetsInCollection(collectionTweetsMock);
    var expectedNumberOfTweetsInCollection = 3;

    expect(actualNumberOfTweetsInCollection).toBe
(expectedNumberOfTweetsInCollection);

});

it('checks if collection is not empty', function isEmptyCollection() {

    var actualIsEmptyCollectionValue = CollectionUtils.
isEmptyCollection(collectionTweetsMock);

    expect(actualIsEmptyCollectionValue).toBeDefined();
    expect(actualIsEmptyCollectionValue).toBe(false);
    expect(actualIsEmptyCollectionValue).not.toBe(true);

});

});
```

CollectionUtils-test.js 文件是我们目前创建的最大的测试集了，其中有不少东西值得学习，让我们仔细看看。

第一件事情就是告诉 Jest 不要去做自动模拟数据：

```
jest.autoMockOff();
```

自动模拟是什么意思呢？在测试环境中，Jest 通常会用自己的 `require()` 方法，它的 `require()` 方法会加载真实的模块，然后将其替换为模拟的版本。结果，`require()` 方法在测试环境中的表现并不是预期那样，而且连测试模块内部依赖的模块也不会是预期那样的表现。鉴于此，我们对要测试的模块及该模块依赖的模式，都要调用 `jest.`

`dontMock()` 方法。有时候依赖可能很多。为了不用每次都调用 `jest.dontMock()`，可以调用 `jest.autoMockOff()` 方法来关闭这个特性。这样默认情况下 Jest 就不会模拟数据了，这意味着如果我们需要模拟数据，可以为每个模块显式地调用 `jest.mock()` 方法。

在 `CollectionUtils-test.js` 文件中，我们只导入了 `CollectionUtils` 模块和它的依赖模块 `TweetUtils`，所以可以用下面稍微麻烦点的代码代替 `jest.autoMockOff()`：

```
jest.dontMock('../CollectionUtils');
jest.dontMock('../TweetUtils');
```

通过调用 `jest.autoMockOff()` 把自动模拟数据关闭后，再在后面的代码中调用 `jest.autoMockOn()` 可以将它重新开启。

关闭了自动模拟数据后，我们来定义测试套件：

```
describe('Collection utilities module', function () {

  var CollectionUtils = require('../CollectionUtils');

  var collectionTweetsMock = {
    collectionTweet7: {},
    collectionTweet8: {},
    collectionTweet9: {}
  };

  // 测试规范的实现……
});
```

我们将这个测试套件命名为 `Collection utilities module`，因为这个名字恰当描述了这个模块。现在先来看看这个套件的实现。这里没有像前面那样立即定义规范，而是先将 `collectionutils` 模块导入，并创建 `collectiontweetsmock` 对象。可以这样做吗？当然可以，实现测试套件的函数也是 JavaScript 函数，我们可以在写测试规范之前先在这里做一些其他的事情。

这个测试集将实现多个测试规范，所有规范都将使用 `collectionTweetsMock` 对

象，因此将它定义在测试规范之外并在测试规范内部使用它是有意义的。你可能已经猜到了，`collectionTweetsMock` 对象会模拟推文的集合。

现在来实现具体的测试规范。

第一个测试规范将测试 `CollectionUtils` 模块是否返回推文集合中推文的数量：

```
it('returns a number of tweets in collection', function
getNumberOfTweetsInCollection() {

    var actualNumberOfTweetsInCollection = CollectionUtils.
getNumberOfTweetsInCollection(collectionTweetsMock);
    var expectedNumberOfTweetsInCollection = 3;

    expect(actualNumberOfTweetsInCollection).toBe
(expectedNumberOfTweetsInCollection);

});
```

首先从模拟的推文集合中获取真实的推文数量：

```
var actualNumberOfTweetsInCollection = CollectionUtils.
getNumberOfTweetsInCollection(collectionTweetsMock);
```

这里，我们调用 `getNumberOfTweetsInCollection()` 方法并将 `collectionTweetsMock` 对象传入，接着，我们定义期望返回的推文数量：

```
var expectedNumberOfTweetsInCollection = 3;
```

最后，调用全局函数 `expect()` 创建一个期望：

```
expect(actualNumberOfTweetsInCollection).toBe(expectedNumberOfTweetsInCo
llection);
```

我们使用 `toBe()` 匹配器来判断实际值和预期值是否一致。

如果现在执行 `npm test` 命令，你会看到这两个测试集都通过了测试：

Using Jest CLI v0.4.18

PASS source/utils/__tests__/TweetUtils-test.js (0.094s)

```
PASS source/utils/__tests__/CollectionUtils-test.js (0.104s)
2 tests passed (2 total)
Run time: 1.297s
```

测试套件要通过测试，那么它必须只包含通过规范。而规范要通过测试，则必须满足所有期望。到目前为止确实是这样。

运行一个小小的错误实验会怎么样？

打开 `~/snapterest/source/utils/CollectionUtils.js` 文件，找到 `getNumberOfTweetsInCollection()` 函数的下面这一行：

```
return listOfCollectionTweetIds.length;
```

修改为下面这样子：

```
return listOfCollectionTweetIds.length + 1;
```

这样一改会导致返回错误的推文数量。现在运行 `npm test` 一次。你应该看到所有在 `CollectionUtils-test.js` 里的规范都未通过。这就是我们感兴趣的结果：

```
FAIL source/utils/__tests__/CollectionUtils-test.js (0.12s)
Collection utilities module > it returns a number of tweets in collection
- Expected: 4 toBe: 3
    at Spec.getNumberOfTweetsInCollection (/Users/artemij/snapterest/
source/utils/__tests__/CollectionUtils-test.js:18:46)
    at Timer.listOnTimeout [as ontimeout] (timers.js:112:15)
...
```

之前我们没有遇到过未通过的测试，所以仔细看看这里都提供了哪些信息。

首先，它告诉我们 `CollectionUtils-test.js` 文件测试未通过：

```
FAIL source/utils/__tests__/CollectionUtils-test.js (0.12s)
```

接着，通过一个友好的方式告诉我们测试的是什么：

```
Collection utilities module > it returns a number of tweets in collection
```

再接着，告诉我们错在那里，预期之外的结果是什么：

- Expected: 4 toBe: 3

最后，Jest 输出了一个调用栈，提供了技术细节以方便我们快速定位具体是什么位置出现了这个问题：

```
at Spec.getNumberOfTweetsInCollection (/Users/artemij/snapterest/source/
utils/__tests__/CollectionUtils-test.js:18:46)
at Timer.listOnTimeout [as ontimeout] (timers.js:112:15)
```

注意，这里提示了具体哪一个测试规范没有达到期望的结果：

at Spec.getNumberOfTweetsInCollection

getNumberOfTweetsInCollection() 函数是实现该测试规范的函数：

```
it('returns a number of tweets in collection', function
getNumberOfTweetsInCollection() {
```

```
    // 测试规范的实现……
```

```
});
```

可见，给实现规范的函数起名字在测试未通过时是非常有用的。

好了，测试目的已经达到了。下面来恢复我们的~/snapterest/source/utils/CollectionUtils.js 文件：

```
return listOfCollectionTweetIds.length;
```

之前的测试套件和 CollectionUtils-test.js 文件的一个明显区别就是测试规范的数量。Jest 中一个测试集可以有多个测试规范，用来测试单个模块的不同方法。

CollectionUtils 模块有两个方法。针对这个模块，我们的测试套件有三个测试规范，让我们看看其他两个。

CollectionUtils-test.js 文件的下一个测试规范会检查推文集合是否非空：

```
it('checks if collection is not empty', function isEmptyCollection() {

    var actualIsEmptyCollectionValue = CollectionUtils.isEmptyCollection
(collectionTweetsMock);
```

```
expect(actualIsEmptyCollectionValue).toBeDefined();
expect(actualIsEmptyCollectionValue).toBe(false);
expect(actualIsEmptyCollectionValue).not.toBe(true);

});
```

首先,调用 `isEmptyCollection()` 方法,并将 `collectionTweetsMock` 对象参数传入,然后将返回的结果存储在 `actualIsEmptyCollectionValue` 变量上。需要注意的是,我们在这里重用了前面测试规范中使用的 `collectionTweetsMock` 对象。

接着,我们创建了三个期望:

```
expect(actualIsEmptyCollectionValue).toBeDefined();
expect(actualIsEmptyCollectionValue).toBe(false);
expect(actualIsEmptyCollectionValue).not.toBe(true);
```

你可能已经猜出来我们对 `actualIsEmptyCollectionValue` 变量的期望结果了,我们期望它是这样的:

```
expect(actualIsEmptyCollectionValue).toBeDefined();
```

这意味着 `isEmptyCollection()` 函数必须返回非 `undefined` 的值。

我们期望这个值是 `false`:

```
expect(actualIsEmptyCollectionValue).toBe(false);
```

在前面,我们使用了 `toEqual()` 匹配器来比较两个数组。`toEqual()` 是深度比较,适合比较数组。但这里是基本值的比较,比如 `false`,所以没必要大材小用。

最后,我们期望 `actualIsEmptyCollectionValue` 值不等于 `true`:

```
expect(actualIsEmptyCollectionValue).not.toBe(true);
```

`.not` 会将接下来的比较器取反,它会匹配预期 `toBe(true)` 的相反值 `false`。

注意,这里使用 `toBe(false)` 和 `not.toBe(true)` 都会得到同样的结果。

只有当这三个期望都得到满足时,这个测试规范才会通过。

到目前为止，我们已经测试了工具模块，但是如何用 Jest 测试 React 组件呢？这正是接下来的内容。

测试 React 组件

与测试工具模块类似，测试 React 组件同样要先创建 `__tests__` 目录。切换到 `~/snapterest/source/components/` 目录并创建 `__tests__` 目录。

我们将要测试的第一个 React 组件是 Header，在 `~/snapterest/source/components/` `__tests__` 目录下创建 `Header-test.js` 文件，内容如下。

```
jest.dontMock('../Header.react');

describe('Header component', function () {

  it('renders provided header text', function () {

    var React = require('react');
    var ReactDOM = require('react-dom');
    var TestUtils = require('react-addons-test-utils');
    var Header = require('../Header.react');

    var header = TestUtils.renderIntoDocument(
      <Header text="Testing..." />
    );

    var actualHeaderText = ReactDOM.findDOMNode(header).textContent;

    expect(actualHeaderText).toBe('Testing...');

    var defaultHeader = TestUtils.renderIntoDocument(
      <Header />
    );

    var actualDefaultHeaderText = ReactDOM.findDOMNode(
      defaultHeader).textContent;
```

```
    expect(actualDefaultHeaderText).toBe('Default header');  
  });  
});
```

现在,我们已经熟悉测试文件的结构了。首先,我们要告诉 Jest 不要去模拟 Header 组件。接着,定义测试套件并命名为 'Header component'。我们的测试套件有一个测试规范,名为 renders provided header text,顾名思义,就是要测试 Header 是否会渲染提供给它的标题文字。测试的实现涉及一些之前没讲过的新东西,让我们来仔细看看。

首先,导入 React 和 ReactDOM:

```
var React = require('react');  
var ReactDOM = require('react-dom');
```

接着,导入 React 相关的附加组件:

```
var TestUtils = require('react-addons-test-utils');
```

TestUtils 可以帮助用户使用所选的测试框架来测试 React 组件。当然,它可以很好地运行在 Jest 框架下。首先我们来安装它:

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

接着,由于要测试 Header 组件,所以这里需要引入 Header 组件:

```
var Header = require('../Header.react');
```

接下来的任务是将 Header 组件渲染到 DOM 中,TestUtils 插件中的帮助函数 renderIntoDocument() 正好可以做这件事情:

```
var header = TestUtils.renderIntoDocument(  
  <Header text="Testing..." />  
);
```

我们将 <Header text="Testing..." /> 这个 React 组件实例作为一个参数传入 renderIntoDocument() 函数。需要注意的是,这里的 Header 组件实例拥有一个 text 属性。renderIntoDocument() 函数返回这个组件的一个索引。

现在，我们有了一个已经渲染到 DOM 的 Header 组件的索引。下一步需要检查被渲染的是什么标题文本，因此需要：

1. 找到组件对应的 DOM 节点。
2. 获取 DOM 节点的文本内容。

React 为我们提供了一个获取组件 DOM 节点的方法，还记得是什么吗？对，是 `ReactDOM.findDOMNode()`。

```
ReactDOM.findDOMNode(header)
```

我们将 `header` 作为一个参数传入 `ReactDOM.findDOMNode()` 方法里面，因此 `ReactDOM.findDOMNode()` 最后会返回 DOM 节点，然后我们就可以访问它的 `textContent` 属性了：

```
ReactDOM.findDOMNode(header).textContent;
```

`textContent` 属性的值就是标题文本的实际值：

```
var actualHeaderText = ReactDOM.findDOMNode(header).textContent;
```

最后是创建一个期望去比较实际的值和预期的值：

```
expect(actualHeaderText).toBe('Testing...');
```

我们希望 `Testing...` 被当作 DOM 节点文本来渲染。

好，现在就可以测试给 Header 组件实例提供的标题文本是否渲染到了 DOM 中。

我们如果不为 Header 组件提供标题文本，会发生什么呢？

让我们渲染另一个 Header 组件的实例，然后再找找答案吧：

```
var defaultHeader = TestUtils.renderIntoDocument(  
  <Header />  
);
```

这一次，这个组件实例是没有 `text` 属性的，我们将这个组件的实例引用赋值给 `defaultHeader` 变量。让我们找到 `defaultHeader` 组件的 DOM 节点元素，并且获取这个节点的 `textContent` 属性：

```
var actualDefaultHeaderText = ReactDOM.findDOMNode(defaultHeader).  
textContent;
```

这会将 Header 组件默认渲染的标题文本返回。最后，我们创建一个期望去比较实际值和期望值：

```
expect(actualDefaultHeaderText).toBe('Default header');
```

在这个例子中，actualDefaultHeaderText 变量的值一定会是 Default header。

以上就是测试 React 组件渲染结果的过程，那怎么测试 React 组件的行为呢？

这正是下面将要讨论的内容。

在~/snapterest/source/components/__tests__/目录下创建 Button-test.js 文件，内容如下：

```
jest.dontMock('../Button.react');
```

```
describe('Button component', function () {  
  
  it('calls handler function on click', function () {  
  
    var React = require('react');  
    var TestUtils = require('react-addons-test-utils');  
    var Button = require('../Button.react');  
    var handleClick = jest.genMockFunction();  
  
    var button = TestUtils.renderIntoDocument(  
      <Button handleClick={handleClick} />  
    );  
  
    var buttonInstance = TestUtils.findRenderedDOMComponentWithTag(  
      button, 'button');  
  
    TestUtils.Simulate.click(buttonInstance);  
  
    expect(handleClick).toBeCalled();  
  });  
});
```

```
    var numberOfCallsMadeIntoMockFunction = handleClick.mock.calls.length;

    expect(numberOfCallsMadeIntoMockFunction).toBe(1);
  });
});
```

Button-test.js 文件要测试 Button 组件, 并且检查当用户点击了这个组件时能否触发事件处理函数。不多说, 让我们来看 'calls handler function on click' 规范的实现:

```
var React = require('react/addons');
var TestUtils = require('react-addons-test-utils');
var Button = require('../Button.react');
var handleClick = jest.genMockFunction();

var button = TestUtils.renderIntoDocument(
  <Button handleClick={handleClick} />
);

var buttonInstance = TestUtils.findRenderedDOMComponentWithTag(button,
  'button');

TestUtils.Simulate.click(buttonInstance);

expect(handleClick).toBeCalled();

var numberOfCallsMadeIntoMockFunction = handleClick.mock.calls.length;

expect(numberOfCallsMadeIntoMockFunction).toBe(1);
```

首先, 我们引入 React 测试附件模块和 Button 组件, 和平常做法相同, 我们告诉 Jest 不要模拟 Button 组件。

继续实现测试规范前, 先看看具体的实施方案:

1. 生成一个模拟函数。
2. 渲染 Button 组件的实例，将模拟函数作为点击事件的处理函数。
3. 找出渲染的 Button 组件的实例。
4. 在这个组件实例上模拟点击事件。
5. 检查是否触发了处理函数。
6. 检查模拟函数是否仅被调用了一次。

按照上述方案，首先让我们生成一个模拟函数：

```
var handleClick = jest.genMockFunction();
```

`jest.genMockFunction()` 函数返回一个新生成的 Jest 模拟函数，我们将它命名为 `handleClick`。

接着，将 Button 组件的实例渲染到 DOM 上：

```
var button = TestUtils.renderIntoDocument(  
  <Button handleClick={handleClick} />  
);
```

Button 组件实例把模拟的 `handleClick` 函数作为一个属性值来接收。

然后，找到渲染到 DOM 中的 Button 组件实例：

```
var buttonInstance = TestUtils.findRenderedDOMComponentWithTag(button,  
  'button');
```

`TestUtils.findRenderedDOMComponentWithTag()` 方法会从已经渲染到 DOM 的 Button 组件实例中找出 `button` 标签的元素，我们将它赋值到变量 `buttonInstance` 上。

再接下来，我们模拟一个组件实例上的点击事件：

```
TestUtils.Simulate.click(buttonInstance);
```

`TestUtils.Simulate` 模拟一个事件发送给 DOM 节点。在这个例子中，我们需要

模拟一个点击事件。所以这里调用了 `TestUtils.Simulate.click()` 方法，并将 `buttonInstance` 作为一个 DOM 节点元素传入。`TestUtils.Simulate` 能够模拟 React 支持的所有事件方法，比如 `click()`。

最后，我们创建一个期望：

```
expect(handleClick).toBeCalled();
```

你应该猜到了，我们期望 `handleClick` 模拟函数在测试中至少被调用一次。

我们还要检查它是否仅被调用了一次。怎么检测调用 `handleClick` 模拟函数的次数呢？所有的 Jest 模拟函数都有一个特殊的属性 `.mock`，这个属性保存了模拟函数被调用时的数据，因此可以通过检查 `handleClick` 模拟函数的 `.mock` 属性来看 `handleClick` 模拟函数被调用了多少次：

```
var numberOfCallsMadeIntoMockFunction = handleClick.mock.calls.length;
```

`.mock` 属性有一个自己的 `.calls` 属性，这是一个数组，代表了 `handleClick` 模拟函数所有的调用，这个数组的长度就是 `handleClick` 函数被调用的次数，我们将这个次数存储在 `numberOfCallsMadeIntoMockFunction` 变量上，然后创建另一个期望：

```
expect(numberOfCallsMadeIntoMockFunction).toBe(1);
```

我们期望这个模拟函数的调用次数恰好为 1。

现在测试都写完了，然而它们还不能执行。为什么？让我们来思考一下。我们使用 JSX 的语法在测试程序中调用 React 的组件，但是 Jest 并不支持 JSX 的语法，所以执行 `npm test` 时，所有的测试都会不通过并且会报告 `SyntaxError`。

我们需要做的就是配置 Jest，让它来支持 `babel-jest` 这个预处理模块。将下面的 `jest` 属性添加到 `~/snapterest/package.json` 文件中：

```
"jest": {
  "scriptPreprocessor": "<rootDir>/node_modules/babel-jest",
  "testFileExtensions": ["es6", "js"],
  "unmockedModulePathPatterns": [
    "<rootDir>/node_modules/react"
  ]
}
```

安装 babel-jest:

```
npm install --save-dev babel-jest
```

如果你想知道 unmockedModulePathPatterns 具体做了什么, 可以上 <https://facebook.github.io/jest/docs/api.html#config-unmockedmodulepathpatterns-array-string> 了解一下。

现在就可以执行我们的测试了。

切换到~/snapterest/目录并且执行下面的命令:

```
npm test
```

所有的测试应该都会 PASS:

```
PASS source/utils/__tests__/TweetUtils-test.js (0.134s)
PASS source/utils/__tests__/CollectionUtils-test.js (0.146s)
PASS source/components/__tests__/Button-test.js (2.802s)
PASS source/components/__tests__/Header-test.js (2.877s)
4 tests passed (4 total)
Run time: 4.728s
```

这样的日志信息会让你睡个好觉, 或安心地度过一个假期, 而不需要经常检查自己的电子邮箱。

小结

现在, 我们了解了怎样创建 React 组件, 以及怎样为它们写单元测试。

在这一章中, 我们了解了使用 Jest 的一些必要条件, Jest 是由 Facebook 推出的能够与 React 良好配合的单元测试框架。本章介绍了测试套件、规范、期望和匹配器, 并且创建了模拟数据和模拟点击事件。

在下一章中, 我们将了解 Flux 的基本架构, 以及如何利用它来提高 React 应用程序的可维护性。

9

使用 Flux 完善 React 架构

开发一款 Web App 的过程有别于建造一座桥的过程，后者会有一个明确的节点代表项目结束，但前者不是，它更像生物的进化历程，永不停息，应当由你或你的团队主观地决定什么时候应该停下来发布一个版本。

到目前为止，我们的应用 **Snapterest** 已经达到了可以发布的程度，这款基于 **React.js** 的 App 已经实现了基本功能而且可以正常运行了。

但是，这样就足够了吗？

并不是！我们曾在前面章节讨论过，在一款 App 的生命周期中，维护阶段的成本远比开发阶段的成本高得多，无论是时间成本还是精力成本。如果我们现在停止开发，也就意味着 **Snapterest** 进入了维护阶段。

然而自问一下，我们做好准备了吗？如果以后 **Snapterest** 需要增加新的功能，我们能在不改动核心代码的前提下完成吗？

分析当前应用的架构

为了回答上面的问题，我们先抛开代码细节来看一下应用的整体架构：

- `app.js` 负责渲染 `Application` 组件。
- `Application` 组件管理推文列表并渲染 `Stream` 组件和 `Collection` 组件。
- `Stream` 组件负责从 `SnapkiteStreamClient` 库中接收新推文并渲染 `StreamTweet` 组件和 `Header` 组件。

- Collection 组件负责渲染 CollectionControls 组件和 TweetList 组件。

现在你能整理出应用中完整的数据流吗？数据的入口在哪？推文的生命周期又是怎样的？我们来仔细梳理一下。

1. 首先，在 Stream 组件中，通过 SnapkiteStreamClient 库接收到一条新的推文。
2. 接着 Stream 组件将这条推文传递给 StreamTweet 组件。
3. StreamTweet 组件又将它传递给了 Tweet 组件，后者渲染出推文的图片。
4. 在用户点击这个图片时，就能把这条推文加入到已选择列表中。
5. Tweet 组件通过 handleImageClick(tweet) 回调函数把推文传给了 StreamTweet 组件。
6. StreamTweet 组件通过 onAddTweetToCollection(tweet) 回调函数把推文传给了 Stream 组件。
7. Stream 组件又通过 onAddTweetToCollection(tweet) 回调函数把推文传给了 Application 组件。
8. Application 组件把推文对象添加到 collectionTweets 中，并更新自己的 state 数据。
9. state 的改变触发 Application 组件进行重新渲染，其中 Collection 组件会渲染更新后的推文列表。
10. 最后 Collection 组件的子组件也能从 props 中获取到更新后的推文列表。

晕了吗？你觉得从长远看这种架构能支持今后业务的正常运行吗？它的可维护性高吗？答案当然是否定的！

既然不可行，那么我们来找一下目前架构的主要问题。按目前的架构，一条新的记录是从 Stream 组件开始进入应用，通过了层层组件向下传输到了 Tweet 组件，然后又层层向上传输到了 Application 组件，由 Application 组件负责保存和管理。

那么为什么要由 Application 组件保存和管理数据呢？因为 Application 组件

的两个子组件 `Stream` 和 `Collection` 都需要访问推文列表数据。为了传递数据，`Application` 组件还需要向这两个子组件提供一些回调函数。

- `Stream` 组件：

```
<Stream onAddTweetToCollection={this.addTweetToCollection} />
```

- `Collection` 组件：

```
<Collection
  tweets={this.state.collectionTweets}
  onRemoveTweetFromCollection={this.removeTweetFromCollection}
  onRemoveAllTweetsFromCollection={this.
    removeAllTweetsFromCollection} />
```

`Stream` 组件通过 `prop` 获取到 `onAddTweetToCollection()` 函数，用于把一条推文添加到列表中。`Collection` 组件则获取到 `onRemoveTweetFromCollection()` 和 `onRemoveAllTweetsFromCollection()` 两个函数，分别用于从列表中移除一条记录和移除所有记录。

这些回调函数会被向下层层传递，直到某个组件真正需要使用它们。在我们的应用中，`onAddTweetToCollection()` 函数只在 `Tweet` 组件中被使用，我们来数一下在真正被使用前它一共在组件间被传递了多少次：

```
Application > Stream > StreamTweet > Tweet
```

`Stream` 组件和 `StreamTweet` 组件中并没有使用 `onAddTweetToCollection()`，它只是被作为属性向下传递。

由于 `Snapterest` 只是一个非常小的应用，所以这种传递并没有显得特别烦琐，但是如果以后会不断给它加入一些新功能，其中的不方便简直就是一场噩梦：

```
Application > ComponentA > ComponentB > ComponentC > ComponentD > ComponentE >
ComponentF > ComponentG > Tweet
```

为了阻止这场噩梦的发生，我们需要解决两个问题：

- 改变数据进入应用的方式。
- 改变组件获取和修改数据的方式。

我们引入 Flux 架构来重新规划应用里数据的流动方式。

理解 Flux

Flux 是一种代码架构,它和 **React** 一样是由 Facebook 提出的。它不是一种框架或者库,确切地说它是一种解决问题的思路,一种用于构建具有良好扩展性的网页应用的思路。

引入 Flux 后,我们来重新规划应用中的数据流。在 Flux 的帮助下,应用中的数据只能单向流动,这个特性能够帮助我们理解应用的运行机制,而且不受应用规模的影响。所以我们可以很方便地添加新功能,而无须对旧代码做大量改动。

了解了 Flux 使用了和 **React** 相同的单向数据流的设计思想,也就理解了为什么它们能很好地协作。我们已经了解了数据单向流动在 **React** 组件中的实现细节,那么它在 Flux 架构中又是怎样实现的呢?

在 Flux 中,我们把应用的关注点分成 4 种逻辑模块:

- **Action** (动作)
- **Dispatcher** (分发器)
- **Store** (存储)
- **View** (视图)

每当应用状态改变时就会创建一个 **Action** 对象。例如,当应用接收到一条新的推文时就需要创建一个 **Action**。**Action** 对象具有一个 `type` 属性,标识了它的类型,**Action** 对象也具有改变为新 **state** 所需的其他所有属性。下面是一个动作对象 **action** 示例:

```
var action = {  
  type: 'receive_tweet',  
  tweet: tweet  
};
```

这是一个 `type` 为 `receive_tweet` 的动作,它还包含 `tweet` 属性。单从 `type` 属性你应该就能猜到这个动作是在应用接收到一条新的推文时创建的。

动作创建之后会怎样流动呢? 最终又会被哪个模块接收? 答案是存储。

存储负责管理应用中的数据。它们提供了读取数据的方法,但是没有提供修改数据的

方法。如果需要修改，则需要创建并分发一个动作来实现。

我们已经了解了怎样创建一个动作，但是它是怎样被分发的？

顾名思义，分发器就是用来做这个的。它负责把所有的动作分别分发到每个存储中：

- 它通过分发器管理注册过的回调函数。
- 所有的动作都会通过分发器被分发到每个存储中。

数据流如下：

```
Actions > Dispatcher > Stores
```

可以看到，分发器在数据流中占据核心地位：所有的动作都流经它，所有的存储都向它注册。在分发器中，动作会依次被分发，换句话说，只有当上一个动作处理完毕后，下一个动作才会被分发。在 Flux 架构中，动作不能脱离分发器独立使用。

创建分发器

现在让我们来实现数据流。首先创建一个分发器，Facebook 官方实现了一版分发器，我们拿来直接使用。

1. 打开~/snapterest 文件夹，然后执行这个命令：

```
npm install --save flux
```

flux 模块中包含了我们需要使用的 Dispatcher。

2. 接下来，在项目目录下创建一个 dispatcher 文件夹：~/snapterest/source/dispatcher，并在其中创建 AppDisaptcher.js 文件：

```
var Dispatcher = require('flux').Dispatcher;  
module.exports = new Dispatcher();
```

首先，我们引入 Facebook 实现的 Dispatcher 类，接着为它新建一个实例，然后导出这个实例。这样我们就可以在应用中使用这个实例了。

接下来，我们需要一个更优雅的方式来创建和分发动作。我们创建一些函数作为 action 的生成器（action creator），每个动作生成器都负责创建和分发一类动作。

创建动作生成器

让我们在项目目录中新建一个 actions 文件夹：~/snapterest/source/actions，然后在里面创建一个 TweetActionCreators.js 文件：

```
var AppDispatcher = require('../dispatcher/AppDispatcher');

function receiveTweet(tweet) {

  var action = {
    type: 'receive_tweet',
    tweet: tweet
  };

  AppDispatcher.dispatch(action);
}

module.exports = {
  receiveTweet: receiveTweet
};
```

动作生成器需要使用分发器来分发动作。我们引入之前创建的 AppDispatcher：

```
var AppDispatcher = require('../dispatcher/AppDispatcher');
```

接着，我们创建第一个动作生成器 receiveTweet()：

```
function receiveTweet(tweet) {

  var action = {
    type: 'receive_tweet',
    tweet: tweet
  };

  AppDispatcher.dispatch(action);
}
```

receiveTweet() 函数接收 tweet 对象作为参数，创建了一个 action 对象，其 type 属性值为 receive_tweet。作为属性，同时它也包含了 tweet 对象。现在每个存储都可

以接收到这个 tweet 对象了。

最后, `receiveTweet()` 方法调用了 `AppDispatcher` 对象中的 `dispatch()` 方法, 把动作对象分发出去。

```
AppDispatcher.dispatch(action);
```

所有在 `AppDispatcher` 中注册过的存储都能收到 `dispatch()` 方法分发的动作。

现在, 我们已经创建了 `AppDispatcher` 和 `TweetActionCreators`, 接下来创建我们的第一个存储。

创建存储

前面讲过, 在 Flux 架构中, 存储的作用是储存和管理数据, 同时对外向 React 组件提供数据。那么下一步, 我们打算创建一个存储, 用于储存从 Twitter 接收到的新推文。

在项目目录下创建一个 `stores` 文件夹: `~/snapterest/source/stores`, 在里面创建 `TweetStore.js` 文件, 如下。

```
var AppDispatcher = require('../dispatcher/AppDispatcher');
var EventEmitter = require('events').EventEmitter;
var assign = require('object-assign');

var tweet = null;

function setTweet(receivedTweet) {
  tweet = receivedTweet;
}

function emitChange() {
  TweetStore.emit('change');
}

var TweetStore = assign({}, EventEmitter.prototype, {

  addChangeListener: function (callback) {
```

```
    this.on('change', callback);
  },

  removeChangeListener: function (callback) {
    this.removeListener('change', callback);
  },

  getTweet: function () {
    return tweet;
  }

});

function handleAction(action) {
  if (action.type === 'receive_tweet') {
    setTweet(action.tweet);
    emitChange();
  }
}

TweetStore.dispatchToken = AppDispatcher.register(handleAction);

module.exports = TweetStore;
```

TweetStore.js 文件实现了一个简单的存储。我们可以把它分为 4 个部分来理解：

- 引入依赖模块，定义私有数据和方法。
- 创建包含公共方法的 TweetStore 对象。
- 创建 action 回调处理器并向 dispatcher 注册 store。
- 将 dispatchToken 附加给 TweetStore 对象，并导出。

在存储的第一部分，我们只是引入需要的依赖模块：

```
var AppDispatcher = require('../dispatcher/AppDispatcher');
var EventEmitter = require('events').EventEmitter;
var assign = require('object-assign');
```

我们引入了 AppDispatcher 模块，因为存储需要向它注册。接着引入了

EventEmitter 类，用来为存储提供处理事件的能力：

```
var EventEmitter = require('events').EventEmitter;
```

最后引入了 object-assign 模块，用于把多个对象的全部属性复制到一个目标对象中：

```
var assign = require('object-assign');
```

我们来安装这个模块：

```
npm install --save object-assign
```

现在已经引入了全部的依赖，接下来定义由存储管理的数据：

```
var tweet = null;
```

这个推文对象由 TweetStore 负责管理，它的初始值为 null，代表尚未收到新的推文。

接下来，定义两个私有方法：

```
function setTweet(receivedTweet) {  
  tweet = receivedTweet;  
}
```

```
function emitChange() {  
  TweetStore.emit('change');  
}
```

setTweet() 函数的功能是使用接收到的 receiveTweet 对象替换掉 tweet。emitChange() 函数的功能是在 TweetStore 对象上触发 change 事件。这些函数是 TweetStore 模块私有的，不能从外部访问。

TweetStore.js 文件的第二部分用于创建 TweetStore 对象：

```
var TweetStore = assign({}, EventEmitter.prototype, {  
  
  addChangeListener: function (callback) {  
    this.on('change', callback);  
  },  
},
```

```
removeChangeListener: function (callback) {
  this.removeListener('change', callback);
},

getTweet: function () {
  return tweet;
}

});
```

我们希望在发生数据变化时存储可以通知应用的其他部分, 所以这里引入了事件机制。在数据发生任何变化时都会触发 `change` 事件, 所有关注数据变化的对象都可以监听这个事件。为此, 我们需要创建 `addChangeListener()` 和 `removeChangeListener()` 两个方法, 分别用于添加和删除对 `change` 事件的监听程序。这两个方法都依赖于 `EventEmitter.prototype` 对象提供的方法, 所以需要将它们从 `EventEmitter.prototype` 上复制到 `TweetStore` 对象中, 通过 `assign()` 函数实现复制:

```
targetObject = assign(targetObject, sourceObject1, sourceObject2);
```

它会把 `sourceObject1` 和 `sourceObject2` 拥有的属性全部复制到 `targetObject` 中, 并返回 `targetObject` 对象。在我们的例子中, `sourceObject1` 对应 `EventEmitter.prototype`, `sourceObject2` 对应定义了一些存储方法的对象字面量:

```
{

  addChangeListener: function (callback) {
    this.on('change', callback);
  },

  removeChangeListener: function (callback) {
    this.removeListener('change', callback);
  },

  getTweet: function () {
    return tweet;
  }
}
```

```
}  
  
}
```

`assign()` 返回的 `targetObject` 包含了从源对象复制来的所有属性，也就是 `TweetStore` 对象。

不知你有没有注意到，`TweetStore` 中定义了 `getTweet()` 方法，但并没有将 `setTweet()` 定义为方法，这是为什么呢？

`TweetStore` 对象在最后被导出，也就意味着它的所有属性能被外部访问到。我们的目标是在外部通过 `TweetStore` 获取数据，但不能直接使用类似 `setTweet()` 的函数更新数据。更新存储中数据的唯一方法是创建 `action` 并通过分发器向注册过的存储分发出去。当存储接收到这个动作时，它可以决定怎么更新数据。

这是 Flux 架构中非常重要的一个特性。存储对自己管理的数据具有全部的控制权，只允许从外部读取数据，而不允许直接修改数据。只能通过动作在存储中改变数据。

`TweetStore.js` 文件的第三部分是创建一个 `action` 处理器（`action handler`），并向分发器注册这个动作处理程序。

首先，创建这个 `action handler`：

```
function handleAction(action) {  
  if (action.type === 'receive_tweet') {  
    setTweet(action.tweet);  
    emitChange();  
  }  
}
```

`handleAction()` 函数接收一个动作对象作为参数并检查它的 `type` 属性。在 Flux 中每一个存储都会接收所有的动作，但并不是每一个存储都需要全部的动作，所以它们有选择地处理动作。在 `TweetStore` 中，首先检查动作的 `type`，如果是 `receive_tweet`，则代表接收到一条新推文。这时 `TweetStore` 通过调用私有的 `setTweet()` 方法将 `tweet` 对象更新为动作中的 `action.tweet`。每当存储中数据变化时，它都需要向所有的订阅者发出通知。实现方法是通过调用私有的 `emitChange()` 函数，触发 `change` 事件，这样所有的外部订阅者就都能收到通知了。

下一项任务是向分发器注册 TweetStore。这时需要调用分发器的 `register()` 方法，并把动作处理程序作参数传入。每当分发器分发动作时，就会调用动作处理程序，动作则作为该回调函数的参数被传入。

看一下我们的例子：

```
TweetStore.dispatchToken = AppDispatcher.register(handleAction);
```

调用 `AppDispatcher` 的 `register()` 方法，把 `handleAction()` 函数作为参数传入。`register()` 方法会返回一个用于标记 `TweetStore` 的记号（`token`），这个记号在 `AppDispatcher` 的其他方法中还会用到，所以我们把它存下来，作为 `TweetStore` 的一个属性。

`TweetStore.js` 文件的第 4 部分是导出 `TweetStore` 对象：

```
module.exports = TweetStore;
```

以上就是创建存储的全过程。截至目前，我们已经实现了动作生成器、分发器及存储。让我们回过头梳理一下 Flux 架构：

1. 存储向分发器注册自己。
2. 动作生成器创建动作并通过分发器分发动作。
3. 存储处理和它相关的动作，并根据这些动作更新数据。
4. 存储向所有订阅者通报数据变化。

看完这些你可能会有疑惑，是谁触发的动作生成器？又是谁订阅了存储的变化呢？这都是一些非常好的问题，答案正在下一章等着你。

小结

在这一章中，我们分析了我们的 React 应用的架构，了解了 Flux 架构的核心思想，并动手实现了分发器、动作生成器及存储。下一章会把它们接入到我们的应用中，把应用的可维护性提升到一个新高度。

10

使用 Flux 提升应用的可维护性

我们之所以选用 Flux 架构，就是希望应用拥有容易维护的数据流。让我们先快速回顾一下前一章中实现的 `AppDispatcher`、`TweetActionCreators` 和 `TweetStore`：

- `TweetActionCreators` 用于创建和分发动作。
- `AppDispatcher` 用于把动作分发到存储中。
- `TweetStore` 用于管理和储存应用的数据。

但是目前的数据流仍不完整，还缺少了一些必要的部分，我们需要在应用中：

- 使用 `TweetActionCreators` 创建动作，开始数据的“旅程”。
- 通过 `TweetStore` 获取数据。

我们需要思考一些重要的问题：在应用中，数据流的起点在哪？这里的数据又是什么？如果能够想通这些问题，就会明白要从哪里下手使用 Flux 架构来重构我们的应用了。

`Snapterest` 的功能是接收和收集最新的推文，而这里和应用唯一有关的数据就是推文，所以在应用中，数据流的起点就是接收到一条新的推文。回想我们目前的架构，应用中的哪一部分是用来接收推文的呢？我们的 `Stream` 组件中有个 `componentDidMount()` 方法：

```
componentDidMount: function () {  
  SnapkiteStreamClient.initializeStream(this.handleNewTweet);  
}
```

是的，我们在 `Stream` 组件渲染之后初始化了推文列表。你也许会有疑惑：“我们之

前不是说过 React 组件只应该负责渲染用户界面吗？”确实是这样，但是按照之前的架构，Stream 组件确实负责了两件事：

- 渲染 StreamTweet 组件
- 初始化数据流

显然，这会为项目以后的维护工作埋下隐患。我们现在借助 Flux 来解开它们之间的耦合。

借助 Flux 实现解耦

首先，创建一个新的工具模块 WebAPIUtils，在~/snapterest/source/Utils/目录中创建 WebAPIUtils.js：

```
var SnapkiteStreamClient = require('snapkite-stream-client');
var TweetActionCreators = require('../actions/TweetActionCreators');

function initializeStreamOfTweets() {  SnapkiteStreamClient.initializeStream(TweetActionCreators.receiveTweet);
}

module.exports = {
  initializeStreamOfTweets: initializeStreamOfTweets
};
```

在这个工具模块中，我们首先引入了 SnapkiteStreamClient 和 TweetActionCreators 两个库。接下来，创建 initializeStreamOfTweets() 函数用于初始化推文列表，这和之前 Stream 组件中的 componentDidMount() 方法功能相同。但是有一点不同，当 SnapkiteStreamClient 接收到新的推文时，它会调用 TweetActionCreators.receiveTweet 方法，同时把收到的推文作为参数传进去：

```
SnapkiteStreamClient.initializeStream(TweetActionCreators.receiveTweet);
```

还记得吗，TweetActionCreators.receiveTweet() 函数接收一个 tweet 对象作为参数：

```
function receiveTweet(tweet) {  
  // ... 创建并分发'receive_tweet'动作  
}
```

`receiveTweet()` 会创建一个新的动作对象，这条推文将作为动作的一个属性。

接下来，从 `WebAPIUtils` 模块中导出 `initializeStreamOfTweets()` 方法：

```
module.exports = {  
  initializeStreamOfTweets: initializeStreamOfTweets  
};
```

现在，我们已经实现了数据初始化的方法及模块。但是我们要在哪里引入和执行它呢？由于它已经从 `Stream` 组件中分离出来，而且也不再依赖任何 `React` 组件，所以我们可以把它加在 `React` 组件渲染之前，加到 `app.js` 文件中：

```
var React = require('react');  
var ReactDOM = require('react-dom');  
var Application = require('./components/Application.react');  
var WebAPIUtils = require('./utils/WebAPIUtils');  
  
WebAPIUtils.initializeStreamOfTweets();  
  
ReactDOM.render(<Application />, document.getElementById  
( 'react-application' ));
```

我们需要做的是引入模块然后执行 `initializeStreamOfTweets()` 方法：

```
var WebAPIUtils = require('./utils/WebAPIUtils');  
  
WebAPIUtils.initializeStreamOfTweets();
```

执行之后，再调用 `React` 的 `render()` 方法：

```
ReactDOM.render(<Application />, document.getElementById  
( 'react-application' ));
```

作为试验，你可以试试在 `TweetActionCreators.receiveTweet` 中加入打印日志的代码，然后删除 `ReactDOM.render()` 相关的代码：

```
function receiveTweet(tweet) {  
  console.log("I've received a new tweet and now will dispatch it together  
with a new action.");  
  
  var action = {  
    type: 'receive_tweet',  
    tweet: tweet  
  };  
  
  AppDispatcher.dispatch(action);  
}
```

然后执行 gulp 进行编译。接着运行，你将在浏览器中看到以下内容：

I am about to learn the essentials of React.js.

同时，在浏览器的控制台中会有如下输出：

```
[Snapkite Stream Client] Socket connected I've received a new tweet and now  
will dispatch it together with a new action.
```

每收到一条新的推文，这句话就会显示一次。尽管我们根本没有渲染任何 React 组件，但是 Flux 架构已经开始工作了：

1. 应用接收到一条新推文。
2. 创建一个动作并分发出去。
3. 因为没有任何存储向分发器注册过，所以没有存储会接收到这个新动作。换句话说，什么都不会发生。

现在，你应该清楚地知道了，React 和 Flux 是彻头彻尾的两个东西，它们之间并不相互依赖。

React 组件自然是需要渲染的，毕竟前面我们花了 9 章的篇幅才实现它们。那要怎么做呢？首先在应用中引入 TweetStore。思考一下要放在哪里呢？自然是一个需要使用推文数据的组件：Stream 组件。

重构 Stream 组件

让我们回顾一下 React 组件获取数据的两种常见方法：

- 调用 `jQuery.ajax()` 或 `napkiteStreamClient.initializeStream()` 等外部库。
- 通过 `props` 属性接收由父组件传来的数据。

但是从现在开始，我们希望 **React** 从组件中而不是储外部库来获取数据。请牢记这个思想，然后开始动手重构 Stream 组件。

修改之前的代码如下：

```
var React = require('react');
var SnapkiteStreamClient = require('snapkite-stream-client');
var StreamTweet = require('./StreamTweet.react');
var Header = require('./Header.react');

var Stream = React.createClass({

  getInitialState: function () {
    return {
      tweet: null
    }
  },

  componentDidMount: function () {
    SnapkiteStreamClient.initializeStream(this.handleNewTweet);
  },

  componentWillUnmount: function () {
    SnapkiteStreamClient.destroyStream();
  },

  handleNewTweet: function (tweet) {
    this.setState({
      tweet: tweet
    });
  }
});
```

```
    },

    render: function () {
      var tweet = this.state.tweet;

      if (tweet) {
        return (
          <StreamTweet
            tweet={tweet}
            onAddTweetToCollection={this.props.onAddTweetToCollection} />
        );
      }

      return (
        <Header text="Waiting for public photos from Twitter..." />
      );
    }
  });

  module.exports = Stream;
```

首先，我们引入 TweetStore，然后删掉 componentDidMount()、componentWillUnmount() 和 handleNewTweet() 这三个方法：

```
var React = require('react');
var StreamTweet = require('./StreamTweet.react');
var Header = require('./Header.react');
var TweetStore = require('../stores/TweetStore');

var Stream = React.createClass({

  getInitialState: function () {
    return {
      tweet: null
    }
  },
```

```

render: function () {
  var tweet = this.state.tweet;

  if (tweet) {
    return (
      <StreamTweet
        tweet={tweet}
        onAddTweetToCollection={this.props.onAddTweetToCollection} />
    );
  }

  return (
    <Header text="Waiting for public photos from Twitter..." />
  );
}
});

module.exports = Stream;

```

同时删除对 `snapkite-stream-client` 模块的引用，我们已经不再依赖它了。接下来，需要改变 `Stream` 组件中获取初始化推文数据的方式。

我们修改一下 `getInitialState()` 方法：

```

getInitialState: function () {
  return {
    tweet: TweetStore.getTweet()
  }
},

```

也许从代码上看，这个改动并不大，但对架构来说，却是一个巨大的改进。现在我们换用了 `getTweet()` 方法从 `TweetStore` 中获取数据。我们在上一章中介绍过，为了方便应用中的其他部分从存储中获取数据，存储会导出一些公共方法。`getTweet()` 就是其中之一，它通常被叫作 `getter` 方法。

你可以直接从存储中获取数据，但是不能用同样的方法更改数据，因为存储并没有相关的 `setter` 方法。这是有意而为的，目的是保持 `Flux` 架构的程序中数据流动的单向性。

这种结构能带来巨大好处，在以后应用的维护过程中你会逐渐体会到。

现在我们已经了解了怎么获取推文的初始化列表，但是我们要怎么处理后续接收到的新推文呢？一种方案是创建一个定时器周期性调用 `TweetStore.getTweet()` 来更新数据，但这并不是最优的方案，因为这种方案的前提是我们无法获知 `TweetStore` 更新推文的时机，但事实上我们有办法获取到。

具体怎么做呢？在上一章中，我们在 `TweetStore` 中实现了一些公共方法，例如 `addChangeListener()` 方法：

```
addChangeListener: function (callback) {  
  this.on('change', callback);  
}
```

以及 `removeChangeListener()` 方法：

```
removeChangeListener: function (callback) {  
  this.removeListener('change', callback);  
}
```

有了这些方法，我们可以准确获取到 `TweetStore` 更新数据的时机。具体来说就是调用 `addChangeListener()` 方法传入一个回调函数，这样每次有数据更新时回调函数就会被调用。那么问题来了，我们要在 `Stream` 组件的什么位置调用 `TweetStore.addChangeListener()` 方法呢？

由于在组件的整个生命周期中，我们只需要添加一次对 `TweetStore` 的 `change` 事件的监听，所以把它放在 `componentDidMount()` 方法中是个不错的选择：

```
componentDidMount: function () {  
  TweetStore.addChangeListener(this.onTweetChange);  
},
```

我们添加了对 `TweetStore` 中 `change` 事件的监听：`this.onTweetChange`，现在，每当 `TweetStore` 更新数据，它就会调用 `this.onTweetChange` 方法，我们一会儿就来实现这个方法。

别忘了在销毁组件之前，移除所有的监听。所以，我们需要在 `Stream` 组件中添加 `componentWillUnmount()` 方法：

```
componentWillUnmount: function () {
  TweetStore.removeChangeListener(this.onTweetChange);
},
```

移除监听和添加监听的方法几乎一样：调用 `TweetStore.removeChangeListener()` 方法，把 `this.onTweetChange` 同样作为参数传入。

现在来实现 `Stream` 组件的 `onTweetChange` 方法：

```
onTweetChange: function () {
  this.setState({
    tweet: TweetStore.getTweet()
  });
},
```

它通过调用 `TweetStore.getTweet()` 方法把组件 `state` 中的推文数据更新为从 `TweetStore` 中获取到的数据。

我们还需要对 `Stream` 组件做最后的改动。在后面的小节中，你会发现 `StreamTweet` 组件不再需要 `handleAddTweetToCollection()` 这个回调了，所以我们需要进行一些改动，把以下代码：

```
return (
  <StreamTweet
    tweet={tweet}
    onAddTweetToCollection={this.props.onAddTweetToCollection} />
);
```

替换成：

```
return (<StreamTweet tweet={tweet} />);
```

现在让我们完整地看一下重构后的 `Stream` 组件：

```
var React = require('react');
var StreamTweet = require('./StreamTweet.react');
var Header = require('./Header.react');
var TweetStore = require('../stores/TweetStore');

var Stream = React.createClass({
```



```
getInitialState: function () {
  return {
    tweet: TweetStore.getTweet()
  }
},

componentDidMount: function () {
  TweetStore.addChangeListener(this.onTweetChange);
},

componentWillUnmount: function () {
  TweetStore.removeChangeListener(this.onTweetChange);
},

onTweetChange: function () {
  this.setState({
    tweet: TweetStore.getTweet()
  });
},

render: function () {
  var tweet = this.state.tweet;

  if (tweet) {
    return (
      <StreamTweet tweet={tweet} />
    );
  }

  return (
    <Header text="Waiting for public photos from Twitter..." />
  );
}
});

module.exports = Stream;
```

我们来重新梳理一下，看看 Stream 组件是怎样保持推文列表实时更新的：

1. 通过调用 `getTweet()` 方法获取 TweetStore 中的最新推文列表作为初始数据。
2. 监听 TweetStore 中的数据变更事件。
3. 当 TweetStore 中推文数据改变时，通过调用 `getTweet()` 方法获取最新的推文数据，并更新到组件的 state 中。
4. 在组件销毁时，停止对 TweetStore 的监听。

这就是 React 组件和 Flux 存储交互的方式。

在继续把应用的剩余部分“Flux 化”之前，让我们总结一下目前的数据流。

- `app.js`：负责接收新的推文并对每条推文调用 `TweetActionCreators`。
- `TweetActionCreators`：负责创建并分发包含这条推文的动作。
- `AppDispatcher`：负责把所有的动作分发到所有的存储。
- `TweetStore`：向分发器注册，每当收到分发器的分发动作时，触发 `change` 事件。
- `Stream`：负责监听 `TweetStore` 的 `change` 事件，从 `TweetStore` 获取新的推文并更新 state，然后重新渲染页面。

发现了吗？现在我们在保持 Snapterest 整体可维护性的前提下，可以随意扩展 React 组件、动作生成器、存储等模块。因为在 Flux 架构中，应用数据保持单向流动，所以无论我们添加多少新的功能都相当于横向扩展，都是在复制这个模式。在以后维护应用的过程中，Flux 的这个特性会让我们受益良多。

让我们使用 Flux 来进一步改造我们的应用。

创建 CollectionStore

Snapterest 不仅可以储存最新一条推文，也可以储存用户创建的推文列表。现在我们用 Flux 来重构这组功能。

首先，创建一个列表存储。在 `~/snapterest/source/stores/` 目录中创建

CollectionStore.js 文件:

```
var AppDispatcher = require('../dispatcher/AppDispatcher');
var EventEmitter = require('events').EventEmitter;
var assign = require('object-assign');

var CHANGE_EVENT = 'change';

var collectionTweets = {};
var collectionName = 'new';

function addTweetToCollection(tweet) {
  collectionTweets[tweet.id] = tweet;
}

function removeTweetFromCollection(tweetId) {
  delete collectionTweets[tweetId];
}

function removeAllTweetsFromCollection() {
  collectionTweets = {};
}

function setCollectionName(name) {
  collectionName = name;
}

function emitChange() {
  CollectionStore.emit(CHANGE_EVENT);
}

var CollectionStore = assign({}, EventEmitter.prototype, {

  addChangeListener: function (callback) {
    this.on(CHANGE_EVENT, callback);
  },
```

```
removeChangeListener: function (callback) {
  this.removeListener(CHANGE_EVENT, callback);
},

getCollectionTweets: function () {
  return collectionTweets;
},

getCollectionName: function() {
  return collectionName;
}

});

function handleAction(action) {

  switch (action.type) {
    case 'add_tweet_to_collection':
      addTweetToCollection(action.tweet);
      emitChange();
      break;

    case 'remove_tweet_from_collection':
      removeTweetFromCollection(action.tweetId);
      emitChange();
      break;

    case 'remove_all_tweets_from_collection':
      removeAllTweetsFromCollection();
      emitChange();
      break;

    case 'set_collection_name':
      setCollectionName(action.collectionName);
      emitChange();
      break;
  }
}
```

```
    default: // ... 什么也不做

  }
}

CollectionStore.dispatchToken = AppDispatcher.register(handleAction);

module.exports = CollectionStore;
```

虽然 `CollectionStore` 包含了更多的功能,但结构上和 `TweetStore` 如出一辙。首先引入依赖,并用 `change` 事件的名称定义常量 `CHANGE_EVENT`:

```
var AppDispatcher = require('../dispatcher/AppDispatcher');
var EventEmitter = require('events').EventEmitter;
var assign = require('object-assign');

var CHANGE_EVENT = 'change';
```

接下来,定义数据及 4 个用于更新数据的私有方法:

```
var collectionTweets = {};
var collectionName = 'new';

function addTweetToCollection(tweet) {
  collectionTweets[tweet.id] = tweet;
}

function removeTweetFromCollection(tweetId) {
  delete collectionTweets[tweetId];
}

function removeAllTweetsFromCollection() {
  collectionTweets = {};
}

function setCollectionName(name) {
  collectionName = name;
}
```

这里两个变量分别用于储存推文列表和列表名称，储存推文列表的对象初始化为空，储存列表名称的变量初始值为 `new`。接下来，我们创建三个私有函数用于更新 `collectionTweets`：

- `addTweetToCollection()` 用于把 `tweets` 对象添加到 `collectionTweets` 中。
- `removeTweetFromCollection()` 用于从 `CollectionTweets` 中移除 `tweets` 对象。
- `removeAllTweetsFromCollection()` 会把 `CollectionTweets` 设置为空，用于移除全部 `tweets` 对象。

另外还有一个私有函数 `setCollectionName`，用于更新 `collectionName`。

这些函数方法之所以被叫作私有函数，是因为它们无法从 `CollectionStore` 外部访问，也就是说不能像这样调用：

```
CollectionStore.setCollectionName('impossible');
```

我们之前提到过，进行这种限制的目的是为了保证应用内数据流动的单向性。

此外还需要一个 `emitChange()` 方法，用于触发 `change` 事件。

接下来创建 `CollectionStore` 对象：

```
var CollectionStore = assign({}, EventEmitter.prototype, {

  addChangeListener: function (callback) {
    this.on(CHANGE_EVENT, callback);
  },

  removeChangeListener: function (callback) {
    this.removeListener(CHANGE_EVENT, callback);
  },

  getCollectionTweets: function () {
    return collectionTweets;
  },
```

```
    getCollectionName: function() {  
      return collectionName;  
    }  
  
  });
```

这段代码和 TweetStore 非常相似，但是多了两个方法：

- `getCollectionTweets()` 用于获取推文列表。
- `getCollectionName()` 用于获取列表的名字。

这些方法都能在 `CollectionStore.js` 外部访问，用于在 React 组件中获取 `CollectionStore` 中的数据。

接下来创建 `handleAction()` 方法：

```
function handleAction(action) {  
  
  switch (action.type) {  
  
    case 'add_tweet_to_collection':  
      addTweetToCollection(action.tweet);  
      emitChange();  
      break;  
  
    case 'remove_tweet_from_collection':  
      removeTweetFromCollection(action.tweetId);  
      emitChange();  
      break;  
  
    case 'remove_all_tweets_from_collection':  
      removeAllTweetsFromCollection();  
      emitChange();  
      break;  
  
    case 'set_collection_name':  
      setCollectionName(action.collectionName);  
      emitChange();  
  }  
}
```

```

        break;

        default: // ... 什么也不做

    }
}

```

这个函数用于接收 `AppDispatcher` 分发的动作,但是和在 `TweetStore` 中不同的是,在 `CollectionStore` 中我们需要对不止一个动作做出响应。事实上有 4 个和推文列表相关的动作需要关注:

- `add_tweet_to_collection` 用于将推文添加到列表中。
- `remove_tweet_from_collection` 用于从列表中移除推文。
- `remove_all_tweets_from_collection` 用于从列表中移除全部推文。
- `set_collection_name` 用于更新推文列表的名称。

还记得吗,每个存储都会接收所有动作,所以 `CollectionStore` 也会接收到 `receive_tweet` 这个动作,我们可直接忽略掉,就像在 `TweetStore` 中也会忽略 `add_tweet_to_collection`、`remove_tweet_from_collection`、`remove_all_tweets_from_collection`、`set_collection_name` 等动作。

接下来向 `AppDispatcher` 注册 `handleAction`,把函数返回的 `dispatchToken` 保存到 `CollectionStore` 对象中:

```
CollectionStore.dispatchToken = AppDispatcher.register(handleAction);
```

最后,把 `CollectionStore` 导出为一个模块:

```
module.exports = CollectionStore;
```

至此,列表存储已经完成了,接着我们来创建动作生成器。

创建 CollectionActionCreators

打开 `~/snapterest/source/actions/` 目录,创建 `CollectionActionCreators.js` 文件:


```
var AppDispatcher = require('../dispatcher/AppDispatcher');

module.exports = {

  addTweetToCollection: function (tweet) {

    var action = {
      type: 'add_tweet_to_collection',
      tweet: tweet
    };

    AppDispatcher.dispatch(action);
  },

  removeTweetFromCollection: function (tweetId) {

    var action = {
      type: 'remove_tweet_from_collection',
      tweetId: tweetId
    };

    AppDispatcher.dispatch(action);
  },

  removeAllTweetsFromCollection: function () {

    var action = {
      type: 'remove_all_tweets_from_collection'
    };

    AppDispatcher.dispatch(action);
  },

  setCollectionName: function (collectionName) {

    var action = {
      type: 'set_collection_name',
```

```

        collectionName: collectionName
    };

    AppDispatcher.dispatch(action);
}

};

```

CollectionStore 中接收的每一个动作，都需要一个对应的动作生成器函数：

- addTweetToCollection() 用于创建并分发类别为 add_tweet_to_collection 的动作，其中包含新的推文。
- removeTweetFromCollection() 用于创建并分发类别为 remove_tweet_from_collection 的动作，其中包含需要从列表中移除的推文的 ID。
- removeAllTweetsFromCollection() 用于创建并分发类别为 remove_all_tweets_from_collection 的动作。
- setCollectionName() 用于创建并分发类别为 set_collection_name 的动作，其中包含新的列表名称。

现在 CollectionStore 和 CollectionActionCreators 已经完成，可以开始重构 React 组件了。

重构 Application 组件

React 组件的重构从哪里入手呢？让我们从整个组件层级中最顶层的组件——Application 组件开始。

我们首先删掉 Application 组件中管理推文列表数据的逻辑，因为现在所有的数据已经由存储接管了。把 getInitialState()、addTweetToCollection()、removeTweetFromCollection() 及 removeAllTweetsFromCollection() 这 4 个方法删掉：

```

var React = require('react');
var Stream = require('./Stream.react');
var Collection = require('./Collection.react');

```

```
var Application = React.createClass({
  render: function () {
    return (
      <div className="container-fluid">

        <div className="row">
          <div className="col-md-4 text-center">

            <Stream onAddTweetToCollection={this.addTweetToCollection} />

          </div>
          <div className="col-md-8">

            <Collection
              tweets={this.state.collectionTweets}
              onRemoveTweetFromCollection={this.
removeTweetFromCollection}
              onRemoveAllTweetsFromCollection={this.
removeAllTweetsFromCollection} />

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

module.exports = Application;
```

现在, Application 组件只剩下一个 `render()` 方法, 用于渲染 Stream 和 Collection 组件。由于重构后的 Application 组件不再包含数据, 所以自然也不再需要给 Stream 和 Collection 组件传递数据属性了。

进一步把 Application 组件的 `render()` 方法修改为:

```
render: function () {
  return (
```

```

<div className="container-fluid">

  <div className="row">
    <div className="col-md-4 text-center">

      <Stream />

    </div>
    <div className="col-md-8">

      <Collection />

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

```

Flux 架构重构以后，Stream 组件包含了最新推文数据，Collection 组件包含了推文列表数据。与之相反，Application 不再包含数据，它变成了单纯包裹 Stream、Collection 两个组件的 HTML 容器。不管是代码结构还是模块功能，重构后的 Application 组件都比之前轻巧简单了不少，代码的可维护性也相应提升了很多。

重构 Collection 组件

接下来重构 Collection 组件，用下面的代码更新 Collection 组件：

```

var React = require('react');
var ReactDOMServer = require('react-dom/server');
var CollectionControls = require('../CollectionControls.react');
var TweetList = require('../TweetList.react');
var Header = require('../Header.react');
var CollectionUtils = require('../utils/CollectionUtils');
var CollectionStore = require('../stores/CollectionStore');

```

```
var Collection = React.createClass({

  getInitialState: function () {
    return {
      collectionTweets: CollectionStore.getCollectionTweets()
    }
  },

  componentDidMount: function () {
    CollectionStore.addChangeListener(this.onCollectionChange);
  },

  componentWillUnmount: function () {
    CollectionStore.removeChangeListener(this.onCollectionChange);
  },

  onCollectionChange: function () {
    this.setState({
      collectionTweets: CollectionStore.getCollectionTweets()
    });
  },

  createHtmlMarkupStringOfTweetList: function () {
    var htmlString = ReactDOMServer.renderToStaticMarkup(
      <TweetList tweets={this.state.collectionTweets} />
    );

    var htmlMarkup = {
      html: htmlString
    };

    return JSON.stringify(htmlMarkup);
  },

  render: function () {
    var collectionTweets = this.state.collectionTweets;
```

```

    var numberOfTweetsInCollection = CollectionUtils. getNumberOfTweets
    InCollection(collectionTweets);
    var htmlMarkup;

    if (numberOfTweetsInCollection > 0) {

        htmlMarkup = this.createHtmlMarkupStringOfTweetList();

        return (
            <div>

                <CollectionControls
                    numberOfTweetsInCollection={numberOfTweetsInCollection}
                    htmlMarkup={htmlMarkup} />

                <TweetList tweets={collectionTweets} />

            </div>
        );
    }

    return <Header text="Your collection is empty" />;
});

module.exports = Collection;

```

都有哪些改动呢？其实非常少。首先是引入了两个新的模块：

```

var CollectionUtils = require('../utils/CollectionUtils');
var CollectionStore = require('../stores/CollectionStore');

```

我们曾在第 8 章创建过一个 CollectionUtils 模块，现在用到的就是它。而引入 CollectionStore 则是为了获取数据。

接下来是 4 个似曾相识的方法：

- `getInitialState()` 方法负责从 `CollectionStore` 获取推文列表并赋给组件，我们之前讲过，`CollectionStore` 中提供了 `getCollectionTweets()` 方法用于获取数据。
- `componentDidMount()` 方法用于添加对 `CollectionStore` 中 `change` 事件的监听，以 `this.onCollectionChange()` 作为回调。每当 `CollectionStore` 中数据发生变化，`Collection` 组件的 `this.onCollectionChange()` 方法就会被调起。
- `componentWillUnmount()` 方法负责移除在 `componentDidMount()` 方法中添加的对 `Change` 事件的监听。
- `onCollectionChange()` 方法负责用从 `CollectionStore` 读取的数据来更新组件的 `state` 数据，继而触发组件重新渲染。

修改后，`Collection` 组件的 `render()` 方法也变得非常简单明了：

```
render: function () {
  var collectionTweets = this.state.collectionTweets;
  var numberOfTweetsInCollection = CollectionUtils. getNumberOfTweets
InCollection(collectionTweets);
  var htmlMarkup;

  if (numberOfTweetsInCollection > 0) {

    htmlMarkup = this.createHtmlMarkupStringOfTweetList();

    return (
      <div>

        <CollectionControls
          numberOfTweetsInCollection={numberOfTweetsInCollection}
          htmlMarkup={htmlMarkup} />

        <TweetList tweets={collectionTweets} />

      </div>
    );
  }
}
```

```
    return <Header text="Your collection is empty" />;  
  }
```

这里改用 `CollectionUtils` 模块的方法获取列表中推文的数量，同时现在需要向子组件 `CollectionControls` 和 `TweetList` 传递的属性也变少了。

重构 CollectionControls 组件

`CollectionControls` 组件也需要做一些改进。让我们先浏览一下重构后的代码，然后再依次讨论修改的内容及修改目的：

```
var React = require('react');  
var Header = require('./Header.react');  
var Button = require('./Button.react');  
var CollectionRenameForm = require('./CollectionRenameForm.react');  
var CollectionExportForm = require('./CollectionExportForm.react');  
var CollectionActionCreators = require('../actions/  
CollectionActionCreators');  
var CollectionStore = require('../stores/CollectionStore');  
  
var CollectionControls = React.createClass({  
  
  getInitialState: function () {  
    return {  
      isEditingName: false  
    };  
  },  
  
  getHeaderText: function () {  
    var numberOfTweetsInCollection = this.props.  
numberOfTweetsInCollection;  
    var text = numberOfTweetsInCollection;  
    var name = CollectionStore.getCollectionName();  
  
    if (numberOfTweetsInCollection === 1) {  
      text = text + ' tweet in your';  
    }  
  }  
});
```



```
    } else {
      text = text + ' tweets in your';
    }

    return (
      <span>
        {text} <strong>{name}</strong> collection
      </span>
    );
  },

  toggleEditCollectionName: function () {
    this.setState({
      isEditingName: !this.state.isEditingName
    });
  },

  removeAllTweetsFromCollection: function () {
    CollectionActionCreators.removeAllTweetsFromCollection();
  },

  render: function () {

    if (this.state.isEditingName) {
      return (
        <CollectionRenameForm
          onCancelCollectionNameChange={this.toggleEditCollectionName} />
      );
    }

    return (
      <div>
        <Header text={this.getHeaderText()} />

        <Button
          label="Rename collection"
          handleClick={this.toggleEditCollectionName} />
      </div>
    );
  }
};
```

```

    <Button
      label="Empty collection"
      handleClick={this.removeAllTweetsFromCollection} />

    <CollectionExportForm htmlMarkup={this.props.htmlMarkup} />
  </div>
);
}
});

module.exports = CollectionControls;

```

首先引入两个新的模块：

```

var CollectionActionCreators = require('../actions/
CollectionActionCreators');
var CollectionStore = require('../stores/CollectionStore');

```

列表名称不再由组件管理，改为从 CollectionStore 中获取：

```

var name = CollectionStore.getCollectionName();

```

接下来是一个比较关键的改动。修改删除按钮的点击回调，把 `handleChangeCollectionName()` 替换成 `removeAllTweetsFromCollection()`，然后我们来实现这个方法：

```

removeAllTweetsFromCollection: function () {
  CollectionActionCreators.removeAllTweetsFromCollection();
}

```

每当用户点击 Empty Collection 按钮时，就会调用 `removeAllTweetsFromCollection()` 方法，接着动作生成器创建并分发一个动作，当 CollectionStore 接收到这个动作后就会清空推文列表，并触发 change 事件。

下面，我们来重构 CollectionRenameForm 组件。

重构 CollectionRenameForm 组件

CollectionRenameForm 是一个受限 (controlled) 表单组件。这也就意味着所有表单的数据都储存在组件的 state 中, 改变表单数据值的唯一办法就是更新组件 state。它的初始数据应当从 CollectionStore 中获取, 现在让我们来实现这个逻辑。

首先需要引入 CollectionActionCreators 和 CollectionStore 两个模块:

```
var CollectionActionCreators = require('../actions/
CollectionActionCreators');
var CollectionStore = require('../stores/CollectionStore');
```

接着需要将 getInitialState() 方法:

```
getInitialState: function() {
  return {
    inputValue: this.props.name
  };
},
```

修改为:

```
getInitialState: function() {
  return {
    inputValue: CollectionStore.getCollectionName()
  };
},
```

后者与前者唯一的区别就是改从 CollectionStore 中获取 inputValue 的初始值。

接下来将 handleFormSubmit() 方法:

```
handleFormSubmit: function (event) {
  event.preventDefault();

  var collectionName = this.state.inputValue;
  this.props.onChangeCollectionName(collectionName);
},
```

修改为：

```
handleFormSubmit: function (event) {
  event.preventDefault();

  var collectionName = this.state.inputValue;
  CollectionActionCreators.setCollectionName(collectionName);
  this.props.onCancelCollectionNameChange();
},
```

这里主要的区别是，每当用户提交表单时，就会创建一个新的动作，然后集合存储就会更新其存储的列表名称：

```
CollectionActionCreators.setCollectionName(collectionName);
```

最后将 `handleFormCancel()` 方法：

```
handleFormCancel: function (event) {
  event.preventDefault();

  var collectionName = this.props.name;
  this.setInputValue(collectionName);
  this.props.onCancelCollectionNameChange();
}
```

修改为：

```
handleFormCancel: function (event) {
  event.preventDefault();

  var collectionName = CollectionStore.getCollectionName();
  this.setInputValue(collectionName);
  this.props.onCancelCollectionNameChange();
}
```

这里同样是从集合存储中获取集合名称：

```
var collectionName = CollectionStore.getCollectionName();
```

以上就是 `CollectionRenameForm` 组件中全部需要修改的内容。下一步我们来重

构 TweetList 组件。

重构 TweetList 组件

TweetList 组件的功能是渲染推文列表，其中每条推文都是一个 Tweet 组件，当推文被点击时，就把它从列表中移除。理解 TweetList 组件的功能后，不知你有没有想到使用 CollectionActionCreators 模块？

确实，我们这里需要用到 CollectionActionCreators 模块，首先添加对它的引用：

```
var CollectionActionCreators = require('../actions/
CollectionActionCreators');
```

然后创建 removeTweetFromCollection() 回调函数，它会在用户点击推文图片时被调用：

```
removeTweetFromCollection: function (tweet) {
  CollectionActionCreators.removeTweetFromCollection(tweet.id);
},
```

删除推文时调用 removeTweetFromCollection() 函数，传入推文 ID 作为参数，创建一个新动作。

最后我们需要确保 RemoveTweetFromCollection() 函数确实被调用了。在 getTweetElement() 方法中，将下面这行代码：

```
var handleRemoveTweetFromCollection = this.props.
onRemoveTweetFromCollection;
```

改为：

```
var handleRemoveTweetFromCollection = this.removeTweetFromCollection;
```

这个组件就已经重构完成了。重构旅程的下一站是 StreamTweet。

重构 StreamTweet 组件

StreamTweet 组件用于渲染推文图片，当图片被用户点击时把这条推文加入到推文列表中。你应该能猜到，在用户点击推文中的图片时，我们需要创建并分发一个新的动作。

首先在 StreamTweet 组件中添加对 CollectionActionCreators 模块的引用：

```
var CollectionActionCreators = require('../actions/
CollectionActionCreators');
```

接着添加一个 addTweetToCollection() 方法：

```
addTweetToCollection: function (tweet) {
  CollectionActionCreators.addTweetToCollection(tweet);
},
```

这是一个回调函数，每当用户点击推文的图片时就会被调用。让我们看一下 render() 方法中的这条语句：

```
<Tweet tweet={tweet} onClick={this.props.onAddTweetToCollection} />
```

把它改成：

```
<Tweet tweet={tweet} onClick={this.addTweetToCollection} />
```

到这里 StreamTweet 组件也重构完成了。

编译

以上就是把 Flux 架构引入到 React 应用中所需要做的全部工作了。现在回过头对比一下重构前后的代码，你会发现使用了 Flux 架构之后的代码非常容易理解。此外，我强烈推荐访问 Flux 的官网 (<https://facebook.github.io/flux/>)，更深入地学习它。

现在是时候检查一下我们的成果了。编译并运行我们的 Snapterest!

打开~/snapterest 目录，然后在终端中执行：

```
gulp
```

你会看到类似这样的输出：

```
Finished 'default' after 2.42 s
```

别忘了运行 Snapkite Engine 应用，我们曾在第 1 章介绍过它的安装和配置。现在，在浏览器中打开 `~/snapterest/build/index.html` 这个文件，你会在左侧看到最新的推文，点击它可以把它加到右边的列表中。

可以正常运行吗？看看控制台有没有报错。如果没有，那么恭喜你！

我们的 React.js 学习之旅已经接近尾声了。非常感谢你花费了大量金钱和时间来购买并学习这本书。我真心希望助你一臂之力，帮你成为一名优秀的 React 开发者，而且我确信你的付出很快就能得到回报。

等等，还不是说再见的时候，我希望和你继续保持交流。如果你有任何的疑惑、建议、想法、评论，或者你希望展示自己的 Snapterest 应用，都欢迎你访问 <https://github.com/fedosejev/react-essentials/issues>，创建一个 issue，我会尽快给你回复。

好好享受开发 React 应用的过程吧！