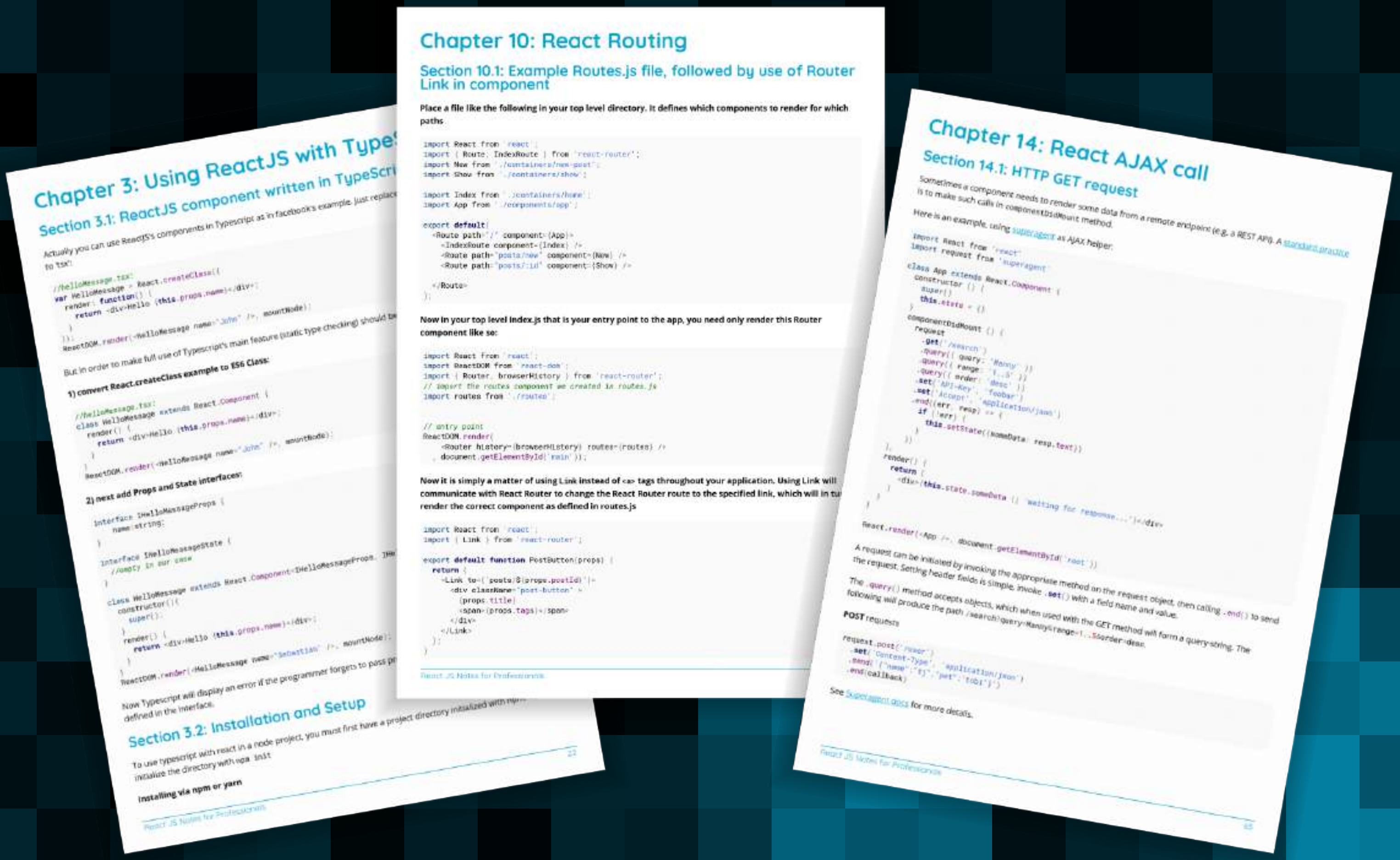
专业人员笔记

React JS

专业人员笔记



Chapter 3: Using ReactJS with TypeScript

Section 3.1: ReactJS component written in TypeScript

Section 3.2: Installation and Setup

Chapter 10: React Routing

Section 10.1: Example Routes.js file, followed by use of Router Link in component

Chapter 14: React AJAX call

Section 14.1: HTTP GET request

100多页

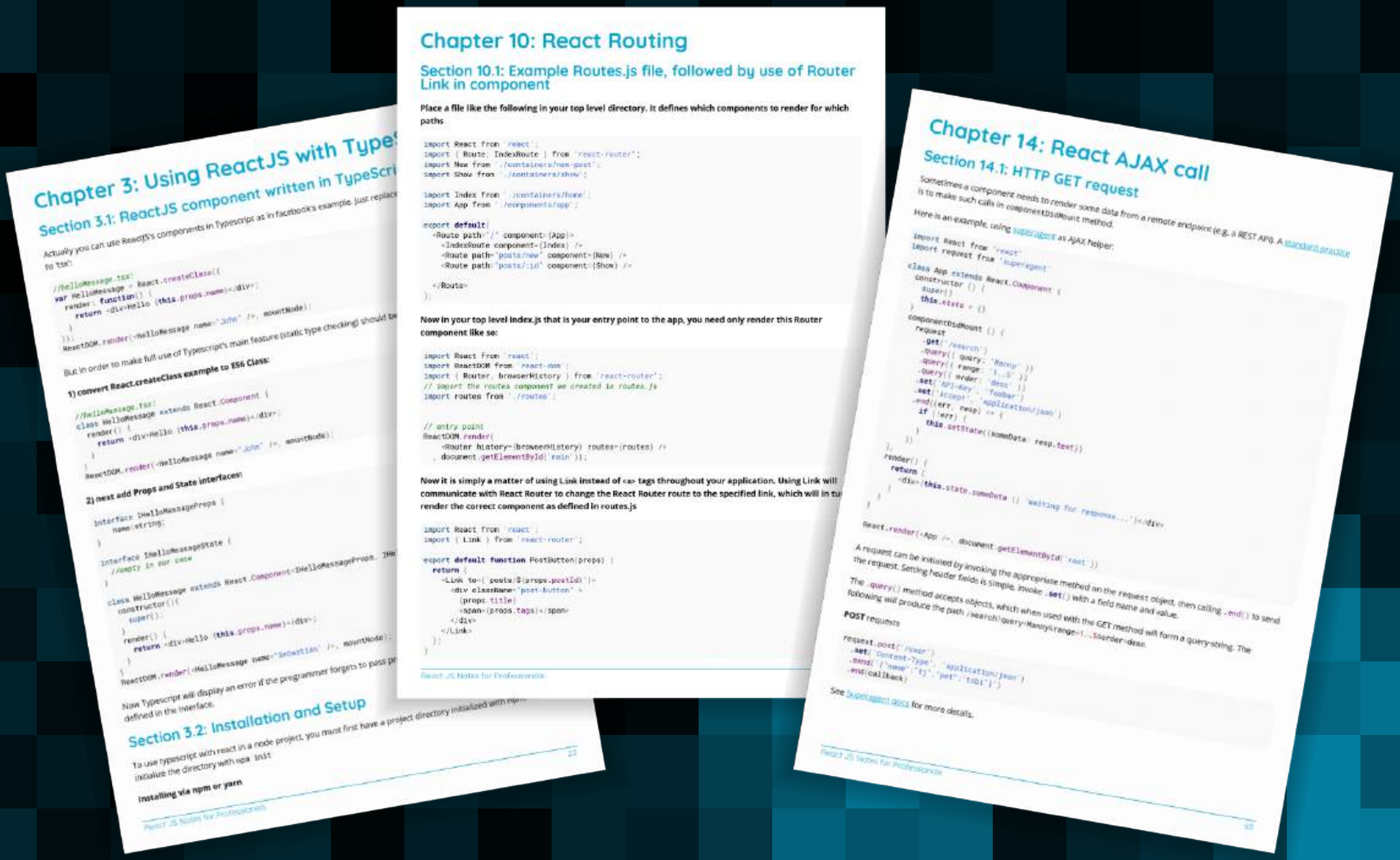
专业提示和技巧

GoalKicker.com
免费编程书籍

免责声明
这是一本非官方的免费书籍，旨在教育用途，与官方React JS组织或公司无关。
所有商标和注册商标均为其各自所有者的财产

React JS

Notes for Professionals



Chapter 3: Using ReactJS with TypeScript

Section 3.1: ReactJS component written in TypeScript

Section 3.2: Installation and Setup

Chapter 10: React Routing

Section 10.1: Example Routes.js file, followed by use of Router Link in component

Chapter 14: React AJAX call

Section 14.1: HTTP GET request

100+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is
not affiliated with official React JS group(s) or company(s).
All trademarks and registered trademarks are
the property of their respective owners

目录

Contents

关于

欢迎随意免费分享此 PDF，
本书的最新版本可从以下网址下载：
https://goalkicker.com/ReactJSBook

这本*React JS 专业笔记*是从Stack Overflow Documentation汇编而成，内容由 Stack Overflow 的优秀人士撰写。文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

这是一本非官方的免费书籍，旨在教育用途，与官方 React JS 组织或公司及 Stack Overflow 无关。所有商标和注册商标均为其各自公司所有者所有。

本书中提供的信息不保证正确或准确，使用风险自负

请将反馈和更正发送至web@petercv.com

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/ReactJSBook

This *React JS Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official React JS group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：React入门

版本	发布日期
0.3.0	2013-05-29
0.4.0	2013-07-17
0.5.0	2013-10-16
0.8.0	2013-12-19
0.9.0	2014-02-20
0.10.0	2014-03-21
0.11.0	2014-07-17
0.12.0	2014-10-28
0.13.0	2015-03-10
0.14.0	2015-10-07
15.0.0	2016-04-07
15.1.0	2016-05-20
15.2.0	2016-07-01
15.2.1	2016-07-08
15.3.0	2016-07-29
15.3.1	2016-08-19
15.3.2	2016-09-19
15.4.0	2016-11-16
15.4.1	2016-11-23
15.4.2	2017-01-06
15.5.0	2017-04-07
15.6.0	2017-06-13
15.6.1	2017-06-14
15.6.2	2017-09-25
16.0.0	2017-09-26
16.1.0	2017-11-09
16.1.1	2017-11-13
16.3.0	2018-03-29
16.3.1	2018-04-03
16.3.2	2018-04-16

第1.1节：什么是ReactJS？

ReactJS是一个开源的、基于组件的前端库，仅负责应用程序的视图层。它由Facebook维护。

ReactJS使用基于虚拟DOM的机制将数据（视图）填充到HTML DOM中。虚拟DOM运行速度快，原因在于它只更改单个DOM元素，而不是每次都重新加载整个DOM。

> 一个React应用由多个组件组成，每个组件负责输出一小块可重用的HTML。组件可以嵌套在其他组件中，从而允许通过简单的构建块构建复杂的应用程序。组件还可以维护内部状态——例如，TabList组件可能存储一个对应当前打开标签的变量。

Chapter 1: Getting started with React

Version	Release Date
0.3.0	2013-05-29
0.4.0	2013-07-17
0.5.0	2013-10-16
0.8.0	2013-12-19
0.9.0	2014-02-20
0.10.0	2014-03-21
0.11.0	2014-07-17
0.12.0	2014-10-28
0.13.0	2015-03-10
0.14.0	2015-10-07
15.0.0	2016-04-07
15.1.0	2016-05-20
15.2.0	2016-07-01
15.2.1	2016-07-08
15.3.0	2016-07-29
15.3.1	2016-08-19
15.3.2	2016-09-19
15.4.0	2016-11-16
15.4.1	2016-11-23
15.4.2	2017-01-06
15.5.0	2017-04-07
15.6.0	2017-06-13
15.6.1	2017-06-14
15.6.2	2017-09-25
16.0.0	2017-09-26
16.1.0	2017-11-09
16.1.1	2017-11-13
16.3.0	2018-03-29
16.3.1	2018-04-03
16.3.2	2018-04-16

Section 1.1: What is ReactJS?

ReactJS is an open-source, component based front end library responsible only for the **view layer** of the application. It is maintained by Facebook.

ReactJS uses virtual DOM based mechanism to fill in data (views) in HTML DOM. The virtual DOM works fast owning to the fact that it only changes individual DOM elements instead of reloading complete DOM every time

> A React application is made up of multiple **components**, each responsible for outputting a small, reusable piece of HTML. Components can be nested within other components to allow complex applications to be built out of simple building blocks. A component may also maintain internal state - for example, a TabList component may store a variable corresponding to the currently open tab.

React允许我们使用一种称为JSX的领域特定语言来编写组件。JSX允许我们使用HTML编写组件，同时混合JavaScript事件。React会在内部将其转换为虚拟DOM，最终输出我们的HTML。

React会快速且自动地对组件中的状态变化进行“反应”，通过利用虚拟DOM重新渲染HTML DOM中的组件。虚拟DOM是实际DOM的内存中表示。通过在虚拟DOM中完成大部分处理，而不是直接在浏览器的DOM中操作，React能够快速响应，只添加、更新和移除自上次渲染周期以来发生变化的组件。

第1.2节：安装或设置

ReactJS是一个包含在单个文件中的JavaScript库，文件名为react-<version>.js，可以包含在任何HTML页面中。人们通常还会安装React DOM库react-dom-<version>.js，与主React文件一起使用：

基本包含

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script type="text/javascript">
// 在这里或单独的文件中使用 React JavaScript 代码
    </script>
    </body>
</html>
```

要获取 JavaScript 文件，请访问官方 React 文档的安装页面。

React 还支持JSX 语法。JSX 是 Facebook 创建的一个扩展，为 JavaScript 添加了 XML 语法。为了使用 JSX，您需要包含 Babel 库，并将<script type="text/javascript">更改为<script type="text/babel">以便将 JSX 转换为 JavaScript 代码。

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
    <script type="text/babel">
// 在此处或单独文件中使用 React JSX 代码
    </script>
    </body>
</html>
```

通过 npm 安装

你也可以通过 npm 安装 React，方法如下：

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

要在你的 JavaScript 项目中使用 React，可以这样做：

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

React allows us to write components using a domain-specific language called JSX. JSX allows us to write our components using HTML, whilst mixing in JavaScript events. React will internally convert this into a virtual DOM, and will ultimately output our HTML for us.

React "*reacts*" to state changes in your components quickly and automatically to rerender the components in the HTML DOM by utilizing the virtual DOM. The virtual DOM is an in-memory representation of an actual DOM. By doing most of the processing inside the virtual DOM rather than directly in the browser's DOM, React can act quickly and only add, update, and remove components which have changed since the last render cycle occurred.

Section 1.2: Installation or Setup

ReactJS is a JavaScript library contained in a single file `react-<version>.js` that can be included in any HTML page. People also commonly install the React DOM library `react-dom-<version>.js` along with the main React file:

Basic Inclusion

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script type="text/javascript">
        // Use react JavaScript code here or in a separate file
    </script>
    </body>
</html>
```

To get the JavaScript files, go to the installation page of the official React documentation.

React also supports JSX syntax. JSX is an extension created by Facebook that adds XML syntax to JavaScript. In order to use JSX you need to include the Babel library and change `<script type="text/javascript">` to `<script type="text/babel">` in order to translate JSX to Javascript code.

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
    <script type="text/babel">
        // Use react JSX code here or in a separate file
    </script>
    </body>
</html>
```

Installing via npm

You can also install React using npm by doing the following:

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

To use React in your JavaScript project, you can do the following:

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

```
ReactDOM.render(<App />, ...);
```

通过 Yarn 安装

Facebook 发布了自己的包管理器，名为 Yarn，也可以用来安装 React。安装
Yarn 后，只需运行以下命令：

yarn add react react-dom

然后你可以在项目中以完全相同的方式使用 React，就像通过 npm 安装 React 一样。

第1.3节：使用无状态函数的Hello World

无状态组件的理念来源于函数式编程。这意味着：一个函数
总是返回与其输入完全相同的结果。

例如：

```
const statelessSum = (a, b) => a + b;

let a = 0;
const statefulSum = () => a++;
```

从上面的例子可以看出，statelessSum 总是会返回给定 a 和 b 的相同值。
然而，statefulSum 函数即使没有参数，也不会返回相同的值。这种类型的函数行为也称为副作用。因为该组件会影响外部的某些东西。

因此，建议更多地使用无状态组件，因为它们是无副作用的，并且总是会产生相同的行为。这正是你希望在应用中实现的，因为状态波动是可维护程序的最糟糕情况。

React 组件中最基本的类型是无状态组件。那些纯粹依赖其属性（props）且不需要任何内部状态管理的 React 组件，可以写成简单的 JavaScript 函数。这些被称为无状态函数组件，因为它们仅仅是props的函数，没有任何需要跟踪的state。

这里有一个简单的例子来说明无状态函数组件的概念：

```
// 在 HTML 中
<div id="element"></div>

// 在 React 中
const MyComponent = props => {
    return <h1>Hello, {props.name}!</h1>;
};

ReactDOM.render(<MyComponent name="Arun" />, element);
// 将渲染 <h1>Hello, Arun!</h1>
```

注意，这个组件所做的只是渲染一个包含name属性的h1元素。该组件不
维护任何状态。这里还有一个 ES6 的例子：

```
import React from 'react'

const HelloWorld = props => (
    <h1>Hello, {props.name}!</h1>
)
```

```
ReactDOM.render(<App />, ...);
```

Installing via Yarn

Facebook released its own package manager named Yarn, which can also be used to install React. After installing Yarn you just need to run this command:

yarn add react react-dom

You can then use React in your project in exactly the same way as if you had installed React via npm.

Section 1.3: Hello World with Stateless Functions

Stateless components are getting their philosophy from functional programming. Which implies that: A function returns all time the same thing exactly on what is given to it.

For example:

```
const statelessSum = (a, b) => a + b;

let a = 0;
const statefulSum = () => a++;
```

As you can see from the above example that, statelessSum is always will return the same values given a and b. However, statefulSum function will not return the same values given even no parameters. This type of function's behaviour is also called as a *side-effect*. Since, the component affects somethings beyond.

So, it is advised to use stateless components more often, since they are *side-effect free* and will create the same behaviour always. That is what you want to be after in your apps because fluctuating state is the worst case scenario for a maintainable program.

The most basic type of react component is one without state. React components that are pure functions of their props and do not require any internal state management can be written as simple JavaScript functions. These are said to be `Stateless Functional Components` because they are a function only of `props`, without having any `state` to keep track of.

Here is a simple example to illustrate the concept of a `Stateless Functional Component`:

```
// In HTML
<div id="element"></div>

// In React
const MyComponent = props => {
    return <h1>Hello, {props.name}!</h1>;
};

ReactDOM.render(<MyComponent name="Arun" />, element);
// Will render <h1>Hello, Arun!</h1>
```

Note that all that this component does is render an `h1` element containing the `name` prop. This component doesn't keep track of any state. Here's an ES6 example as well:

```
import React from 'react'

const HelloWorld = props => (
    <h1>Hello, {props.name}!</h1>
)
```

```
HelloWorld。propTypes = {
    name: React.PropTypes.string.isRequired
}

export default HelloWorld
```

由于这些组件不需要一个后台实例来管理状态，React有更多的优化空间。实现方式很简洁，但截至目前尚未对无状态组件实现此类优化。

第1.4节：创建可复用组件的绝对基础

组件和属性（Props）

由于React只关注应用程序的视图，React开发的大部分工作将是创建组件。组件代表应用程序视图的一部分。“Props”只是用于JSX节点的属性（例如<SomeComponent someProp="some prop's value" />），是应用程序与组件交互的主要方式。在上面的代码片段中，在SomeComponent内部，我们可以访问this.props，其值为对象{someProp: "some prop's value"}。

将React组件视为简单函数是很有用的——它们以“props”作为输入，输出为标记。许多简单组件更进一步，使自己成为“纯函数”，意味着它们不产生副作用，并且是幂等的（给定一组输入，组件总是产生相同的输出）。通过实际将组件创建为函数而非“类”，可以正式实现这一目标。创建React组件有三种方式：

- **函数式（“无状态”）组件**

```
const FirstComponent = props => (
    <div>{props.content}</div>
);
```

- **React.createClass()**

```
const SecondComponent = React.createClass({
    render: function () {
        return (
            <div>{this.props.content}</div>
        );
    }
});
```

- **ES2015 类**

```
class ThirdComponent extends React.Component {
    render() {
        return (
            <div>{this.props.content}</div>
        );
    }
}
```

这些组件的使用方式完全相同：

```
const ParentComponent = function (props) {
    const someText = "FooBar";
```

```
HelloWorld.propTypes = {
    name: React.PropTypes.string.isRequired
}

export default HelloWorld
```

Since these components do not require a backing instance to manage the state, React has more room for optimizations. The implementation is clean, but as of yet no such optimizations for stateless components have been implemented.

Section 1.4: Absolute Basics of Creating Reusable Components

Components and Props

As React concerns itself only with an application's view, the bulk of development in React will be the creation of components. A component represents a portion of the view of your application. "Props" are simply the attributes used on a JSX node (e.g. `<SomeComponent someProp="some prop's value" />`), and are the primary way our application interacts with our components. In the snippet above, inside of SomeComponent, we would have access to `this.props`, whose value would be the object `{someProp: "some prop's value"}`.

It can be useful to think of React components as simple functions - they take input in the form of "props", and produce output as markup. Many simple components take this a step further, making themselves "Pure Functions", meaning they do not issue side effects, and are idempotent (given a set of inputs, the component will always produce the same output). This goal can be formally enforced by actually creating components as functions, rather than "classes". There are three ways of creating a React component:

- **Functional ("Stateless") Components**

```
const FirstComponent = props => (
    <div>{props.content}</div>
);
```

- **React.createClass()**

```
const SecondComponent = React.createClass({
    render: function () {
        return (
            <div>{this.props.content}</div>
        );
    }
});
```

- **ES2015 Classes**

```
class ThirdComponent extends React.Component {
    render() {
        return (
            <div>{this.props.content}</div>
        );
    }
}
```

These components are used in exactly the same way:

```
const ParentComponent = function (props) {
    const someText = "FooBar";
    return (
        <FirstComponent content={someText} />
        <SecondComponent content={someText} />
        <ThirdComponent content={someText} />
    );
}
```

以上示例都会生成相同的标记。

函数组件内部不能有“状态”。所以如果你的组件需要有状态，那么请选择基于类的组件。更多信息请参阅创建组件。

最后需要注意的是，React 的 props 一旦传入就是不可变的，意味着它们不能在组件内部被修改。如果组件的父组件更改了 prop 的值，React 会处理用新的 props 替换旧的，组件将使用新值重新渲染。

请参阅《React 思维方式》和《可复用组件》，深入了解 props 与组件的关系。

第1.5节：创建 React 应用

create-react-app 是由 Facebook 创建的 React 应用脚手架生成器。它提供了一个配置良好的开发环境，易于使用且设置最小化，包括：

- ES6 和 JSX 转译
- 开发服务器支持热模块重载
- 代码规范检查
- CSS 自动添加前缀
- 构建脚本支持 JS、CSS 和图片打包，以及源码映射
- Jest 测试框架

安装

首先，使用 Node 包管理器（npm）全局安装 create-react-app。

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

然后在你选择的目录中运行生成器。

```
create-react-app my-app
```

进入新创建的目录并运行启动脚本。

```
cd my-app/
npm start
```

配置

create-react-app 默认情况下故意不支持配置。如果需要非默认用法，例如使用编译型 CSS 语言如 Sass，则可以使用 eject 命令。

```
npm run eject
```

这允许编辑所有配置文件。注意：此过程不可逆。

替代方案

```
    return (
        <FirstComponent content={someText} />
        <SecondComponent content={someText} />
        <ThirdComponent content={someText} />
    );
}
```

The above examples will all produce identical markup.

Functional components cannot have "state" within them. So if your component needs to have a state, then go for class based components. Refer Creating Components for more information.

As a final note, react props are immutable once they have been passed in, meaning they cannot be modified from within a component. If the parent of a component changes the value of a prop, React handles replacing the old props with the new, the component will rerender itself using the new values.

See Thinking In React and Reusable Components for deeper dives into the relationship of props to components.

Section 1.5: Create React App

create-react-app is a React app boilerplate generator created by Facebook. It provides a development environment configured for ease-of-use with minimal setup, including:

- ES6 and JSX transpilation
- Dev server with hot module reloading
- Code linting
- CSS auto-prefixing
- Build script with JS, CSS and image bundling, and sourcemaps
- Jest testing framework

Installation

First, install create-react-app globally with node package manager (npm).

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

Then run the generator in your chosen directory.

```
create-react-app my-app
```

Navigate to the newly created directory and run the start script.

```
cd my-app/
npm start
```

Configuration

create-react-app is intentionally non-configurable by default. If non-default usage is required, for example, to use a compiled CSS language such as Sass, then the eject command can be used.

```
npm run eject
```

This allows editing of all configuration files. N.B. this is an irreversible process.

Alternatives

替代的 React 模板包括：

- 飞地
- nwb
- motion
- rackt-cli
- budō
- rwb
- quik
- sagui
- roc

构建 React 应用

要构建生产就绪的应用，请运行以下命令

```
npm run build
```

第1.6节：你好，世界

无 JSX

这是一个基本示例，使用 React 的主 API 创建一个 React 元素，并使用 React DOM API 在浏览器中渲染该 React 元素。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
<title>你好，React！</title>

   <!-- 引入 React 和 ReactDOM 库 -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>

  </head>
  <body>
    <div id="example"></div>

    <script type="text/javascript">

      // 创建一个 React 元素 rElement
      var rElement = React.createElement('h1', null, '你好，世界！');

      // dElement 是一个 DOM 容器
      var dElement = document.getElementById('example');

      // 在 DOM 容器中渲染 React 元素
ReactDOM.render(rElement, dElement);

    </script>

</body>
</html>
```

使用 JSX

Alternative React boilerplates include:

- enclave
- nwb
- motion
- rackt-cli
- budō
- rwb
- quik
- sagui
- roc

Build React App

To build your app for production ready, run following command

```
npm run build
```

Section 1.6: Hello World

Without JSX

Here's a basic example that uses React's main API to create a React element and the React DOM API to render the React element in the browser.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>

  </head>
  <body>
    <div id="example"></div>

    <script type="text/javascript">

      // create a React element rElement
      var rElement = React.createElement('h1', null, 'Hello, world!');

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

With JSX

与其从字符串创建 React 元素，不如使用 JSX（由 Facebook 创建的 JavaScript 扩展，用于向 JavaScript 添加 XML 语法），它允许编写

```
var rElement = React.createElement('h1', null, 'Hello, world!');
```

作为等价的（对于熟悉 HTML 的人来说更易读）

```
var rElement = <h1>Hello, world!</h1>;
```

包含 JSX 的代码需要用 <script type="text/babel"> 标签包裹。该标签内的所有内容将使用 Babel 库（需要额外引入，除了 React 库之外）转换为纯 JavaScript。

所以最终上述示例变为：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- 引入 React 和 ReactDOM 库 -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
    <!-- 引入 Babel 库 -->
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>


  <body>
   <div id="example"></div>

    <script type="text/babel">

      // 使用 JSX 创建一个 React 元素 rElement
      var rElement = <h1>Hello, world!</h1>;

      // dElement 是一个 DOM 容器
      var dElement = document.getElementById('example');

      // 在 DOM 容器中渲染 React 元素
ReactDOM.render(rElement, dElement);

    </script>


  <original></body>
</html></original>
```

第1.7节：Hello World组件

React 组件可以定义为继承基础React.Component类的 ES6 类。在其最简形式中，组件必须定义一个render方法，该方法指定组件如何渲染到 DOM。render方法返回 React 节点，这些节点可以使用类似 HTML 标签的 JSX 语法来定义。以下示例展示了如何定义一个最简组件：

```
import React from 'react'

class HelloWorld extends React.Component {
    render() {
```

Instead of creating a React element from strings one can use JSX (a Javascript extension created by Facebook for adding XML syntax to JavaScript), which allows to write

```
var rElement = React.createElement('h1', null, 'Hello, world!');
```

as the equivalent (and easier to read for someone familiar with HTML)

```
var rElement = <h1>Hello, world!</h1>;
```

The code containing JSX needs to be enclosed in a **<script** type="text/babel"**>** tag. Everything within this tag will be transformed to plain Javascript using the Babel library (that needs to be included in addition to the React libraries).

So finally the above example becomes:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
    <!-- Include the Babel library -->
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>

  </head>
  <body>
   <div id="example"></div>

    <script type="text/babel">

      // create a React element rElement using JSX
      var rElement = <h1>Hello, world!</h1>;

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

Section 1.7: Hello World Component

A React component can be defined as an ES6 class that extends the base React.Component class. In its minimal form, a component *must* define a render method that specifies how the component renders to the DOM. The render method returns React nodes, which can be defined using JSX syntax as HTML-like tags. The following example shows how to define a minimal Component:

```
import React from 'react'

class HelloWorld extends React.Component {
    render() {
        return <h1>你好, 世界!</h1>
    }
}

export default HelloWorld
```

组件也可以接收props。这些是由其父组件传递的属性，用于指定组件自身无法知道的一些值；属性还可以包含一个函数，组件在某些事件发生后可以调用该函数——例如，一个按钮可以接收一个用于其onClick属性的函数，并在每次点击时调用它。编写组件时，可以通过组件本身的props对象访问其props：

```
import React from 'react'

class Hello extends React.Component {
    render() {
        return <h1>你好, {this.props.name}!</h1>
    }
}

导出默认 Hello
```

上面的示例展示了组件如何渲染其父组件通过name属性传入的任意字符串。注意，组件不能修改它接收到的props。

组件可以被渲染在任何其他组件内部，或者如果它是最顶层组件，则可以直接渲染到DOM中，方法是使用ReactDOM.render，并提供组件和你希望React树渲染到的DOM节点：

```
import React from 'react'
import ReactDOM from 'react-dom'
import Hello from './Hello'

ReactDOM.render(<Hello name="Billy James" />, document.getElementById('main'))
```

到现在为止，你已经知道如何创建一个基本组件并接受props。让我们更进一步，介绍一下state。

为了演示，我们让Hello World应用在给出全名时，只显示名字的第一部分。

```
import React from 'react'

class Hello extends React.Component {

    constructor(props){

        //由于我们继承了默认构造函数,
        //先处理默认操作。
        super(props);

        //从prop中提取名字的第一部分
        let firstName = this.props.name.split(" ")[0];

        //在构造函数中，可以随意修改当前上下文的
        //state属性。
        this.state = {
            name: firstName
        }
```

```
        return <h1>Hello, World!</h1>
    }
}

export default HelloWorld
```

A Component can also receive `props`. These are properties passed by its parent in order to specify some values the component cannot know by itself; a property can also contain a function that can be called by the component after certain events occur - for example, a button could receive a function for its `onClick` property and call it whenever it is clicked. When writing a component, its `props` can be accessed through the `props` object on the Component itself:

```
import React from 'react'

class Hello extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}!</h1>
    }
}

export default Hello
```

The example above shows how the component can render an arbitrary string passed into the `name` prop by its parent. Note that a component cannot modify the props it receives.

A component can be rendered within any other component, or directly into the DOM if it's the topmost component, using `ReactDOM.render` and providing it with both the component and the DOM Node where you want the React tree to be rendered:

```
import React from 'react'
import ReactDOM from 'react-dom'
import Hello from './Hello'

ReactDOM.render(<Hello name="Billy James" />, document.getElementById('main'))
```

By now you know how to make a basic component and accept `props`. Lets take this a step further and introduce `state`.

For demo sake, let's make our Hello World app, display only the first name if a full name is given.

```
import React from 'react'

class Hello extends React.Component {

    constructor(props){

        //Since we are extending the default constructor,
        //handle default activities first.
        super(props);

        //Extract the first-name from the prop
        let firstName = this.props.name.split(" ")[0];

        //In the constructor, feel free to modify the
        //state property on the current context.
        this.state = {
            name: firstName
        }
    } //看，JSX 基于类的定义中不需要逗号！

render() {
        return <h1>你好, {this.state.name}!</h1>
    }
}

导出默认 Hello
```

注意： 每个组件可以有自己的状态，或者接受父组件的状态作为属性（prop）。

Codepen 示例链接。

```
    } //Look maa, no comma required in JSX based class defs!

    render() {
        return <h1>Hello, {this.state.name}!</h1>
    }
}

export default Hello
```

Note: Each component can have it's own state or accept it's parent's state as a prop.

Codepen Link to Example.

第2章：组件

第2.1节：创建组件

这是基础示例的扩展：

基本结构

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class FirstComponent extends Component {
    render() {
        return (
            <div>
你好, {this.props.name}! 我是 FirstComponent。
            </div>
        );
    }
}

render(
    <FirstComponent name={ 'User' } />,
    document.getElementById('content')
);
```

上述示例被称为无状态组件，因为它不包含状态（React 语境中的状态）。

在这种情况下，有些人更倾向于使用基于 ES6

箭头函数的无状态函数组件。

无状态函数组件

在许多应用中，有智能组件持有状态，但渲染的是仅接收属性并返回HTML作为JSX的哑组件。无状态函数组件更具可重用性，并且对应用性能有积极影响。

它们有两个主要特征：

1. 渲染时接收一个包含所有传递下来的属性的对象
2. 必须返回要渲染的JSX

```
// 在模块中使用JSX时必须导入React
import React from 'react';
import PropTypes from 'prop-types';

const FirstComponent = props => (
    <div>
你好, {props.name}! 我是一个FirstComponent。
    </div>
);

// 箭头组件也可以有属性验证
FirstComponent.propTypes = {
    name: PropTypes.string.isRequired,
}

// 要在另一个文件中使用FirstComponent，必须通过export导出：
```

Chapter 2: Components

Section 2.1: Creating Components

This is an extension of Basic Example:

Basic Structure

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class FirstComponent extends Component {
    render() {
        return (
            <div>
                Hello, {this.props.name}! I am a FirstComponent.
            </div>
        );
    }
}

render(
    <FirstComponent name={ 'User' } />,
    document.getElementById('content')
);
```

The above example is called a **stateless** component as it does not contain state (in the React sense of the word).

In such a case, some people find it preferable to use Stateless Functional Components, which are based on ES6 arrow functions.

Stateless Functional Components

In many applications there are smart components that hold state but render dumb components that simply receive props and return HTML as JSX. Stateless functional components are much more reusable and have a positive performance impact on your application.

They have 2 main characteristics:

1. When rendered they receive an object with all the props that were passed down
2. They must return the JSX to be rendered

```
// When using JSX inside a module you must import React
import React from 'react';
import PropTypes from 'prop-types';

const FirstComponent = props => (
    <div>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

//arrow components also may have props validation
FirstComponent.propTypes = {
    name: PropTypes.string.isRequired,
}

// To use FirstComponent in another file it must be exposed through an export call:
```

```
export default FirstComponent;
```

有状态组件

与上面展示的“无状态”组件相反，“有状态”组件拥有一个状态对象，可以通过setState方法进行更新。状态必须在constructor中初始化，才能被设置：

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

class SecondComponent extends Component {
    constructor(props) {
        super(props);

        this.state = {
            toggle: true
        };

        // 这是为了在将 onClick 作为回调传递时绑定上下文
        this.onClick = this.onClick.bind(this);
    }

onClick() {
        this.setState((prevState, props) => ({
            toggle: !prevState.toggle
        }));
    }

render() {
        return (
            <div onClick={this.onClick}>
你好, {this.props.name}! 我是 SecondComponent。
                <br />
切换状态是: {this.state.toggle}
            </div>
        );
    }
}
```

使用PureComponent替代Component扩展组件时，会自动实现shouldComponentUpdate()生命周期方法，进行浅层的属性和状态比较。这样可以通过减少不必要的渲染次数，提升应用性能。这假设你的组件是“纯”的，并且在相同的状态和属性输入下总是渲染相同的输出。

高阶组件

高阶组件（HOC）允许共享组件功能。

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

const PrintHello = ComposedComponent => class extends Component {
    onClick() {
console.log('hello');
    }

    /* 高阶组件接收另一个组件作为参数
   然后用额外的属性渲染它 */
render() {
        return <ComposedComponent {...this.props } onClick={this.onClick} />
    }
}
```

```
export default FirstComponent;
```

Stateful Components

In contrast to the 'stateless' components shown above, 'stateful' components have a state object that can be updated with the `setState` method. The state must be initialized in the `constructor` before it can be set:

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

class SecondComponent extends Component {
    constructor(props) {
        super(props);

        this.state = {
            toggle: true
        };

        // This is to bind context when passing onClick as a callback
        this.onClick = this.onClick.bind(this);
    }

    onClick() {
        this.setState((prevState, props) => ({
            toggle: !prevState.toggle
        }));
    }

    render() {
        return (
            <div onClick={this.onClick}>
                Hello, {this.props.name}! I am a SecondComponent.
                <br />
                Toggle is: {this.state.toggle}
            </div>
        );
    }
}
```

Extending a component with `PureComponent` instead of `Component` will automatically implement the `shouldComponentUpdate()` lifecycle method with shallow prop and state comparison. This keeps your application more performant by reducing the amount of un-necessary renders that occur. This assumes your components are 'Pure' and always render the same output with the same state and props input.

Higher Order Components

Higher order components (HOC) allow to share component functionality.

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

const PrintHello = ComposedComponent => class extends Component {
    onClick() {
        console.log('hello');
    }

    /* The higher order component takes another component as a parameter
    and then renders it with additional props */
    render() {
        return <ComposedComponent {...this.props } onClick={this.onClick} />
    }
}
```

```
const FirstComponent = props => (
    <div onClick={ props.onClick }>
       你好，{props.name}! 我是 FirstComponent。
    </div>
);

const ExtendedComponent = PrintHello(FirstComponent);
```

当你想在多个组件之间共享逻辑，而不管它们如何渲染时，会使用高阶组件。

第2.2节：基础组件

给定以下HTML文件：

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React 教程</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

你可以使用以下代码在单独的文件中创建一个基础组件：

scripts/example.js

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class FirstComponent extends Component {
  render() {
    return (
      <div className="firstComponent">
       你好，世界！我是一个FirstComponent组件。
      </div>
    );
  }
}
ReactDOM.render(
  <FirstComponent />, // 注意这与上面存储的变量相同
  document.getElementById('content')
);
```

你将得到以下结果（注意div#content内部的内容）：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
```

```
const FirstComponent = props => (
    <div onClick={ props.onClick }>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

const ExtendedComponent = PrintHello(FirstComponent);
```

Higher order components are used when you want to share logic across several components regardless of how different they render.

Section 2.2: Basic Component

Given the following HTML file:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

You can create a basic component using the following code in a separate file:

scripts/example.js

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class FirstComponent extends Component {
  render() {
    return (
      <div className="firstComponent">
        Hello, world! I am a FirstComponent.
      </div>
    );
  }
}
ReactDOM.render(
  <FirstComponent />, // Note that this is the same as the variable you stored above
  document.getElementById('content')
);
```

You will get the following result (note what is inside of `div#content`):

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React 教程</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content">
        <div className="firstComponent">
          你好，世界！我是 FirstComponent。
        </div>
    </div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

第2.3节：组件嵌套

ReactJS 的强大之处在于它允许组件嵌套。来看下面的两个组件：

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        你好, 世界! 我是一个 CommentList。
      </div>
    );
  }
});

var CommentForm = reactCreateClass({
  render: function() {
    return (
      <div className="commentForm">
        你好, 世界! 我是一个 CommentForm。
      </div>
    );
  }
});
```

你可以在不同组件的定义中嵌套并引用这些组件：

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>评论</h1>
        <CommentList /> // 上面定义的，可以重复使用
        <CommentForm /> // 这里也是
      </div>
    );
  }
});
```

进一步的嵌套可以通过三种方式完成，每种方式都有其适用场景。

```
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content">
        <div className="firstComponent">
            Hello, world! I am a FirstComponent.
        </div>
    </div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Section 2.3: Nesting Components

A lot of the power of ReactJS is its ability to allow nesting of components. Take the following two components:

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

var CommentForm = reactCreateClass({
  render: function() {
    return (
      <div className="commentForm">
        Hello, world! I am a CommentForm.
      </div>
    );
  }
});
```

You can nest and refer to those components in the definition of a different component:

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList /> // Which was defined above and can be reused
        <CommentForm /> // Same here
      </div>
    );
  }
});
```

Further nesting can be done in three ways, which all have their own places to be used.

1. 不使用 children 的嵌套

（续上文）

```
var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        <ListTitle/>
你好， 世界！ 我是一个 CommentList。
      </div>
    );
  }
});
```

这是 A 组合 B，B 组合 C 的样式。

优点

- 分离 UI 元素简单且快速
- 根据父组件的状态，轻松向子组件传递属性

缺点

- 对组合架构的可见性较低
- 可重用性较差

适用情况

- B 和 C 只是展示组件
- B 应该负责 C 的生命周期

2. 使用子组件进行嵌套

（续上文）

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>评论</h1>
        <CommentList>
            <ListTitle/> // 子组件
        </CommentList>
        <CommentForm />
      </div>
    );
  }
});
```

这是 A 组合 B，且 A 告诉 B 组合 C 的方式。赋予父组件更多的控制权。

优点

- 更好的组件生命周期管理
- 更清晰的组合架构可见性
- 更好的可复用性

缺点

1. Nesting without using children

(continued from above)

```
var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        <ListTitle/>
        Hello, world! I am a CommentList.
      </div>
    );
  }
});
```

This is the style where A composes B and B composes C.

Pros

- Easy and fast to separate UI elements
- Easy to inject props down to children based on the parent component's state

Cons

- Less visibility into the composition architecture
- Less reusability

Good if

- B and C are just presentational components
- B should be responsible for C's lifecycle

2. Nesting using children

(continued from above)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList>
            <ListTitle/> // child
        </CommentList>
        <CommentForm />
      </div>
    );
  }
});
```

This is the style where A composes B and A tells B to compose C. More power to parent components.

Pros

- Better components lifecycle management
- Better visibility into the composition architecture
- Better reusuability

Cons

- 注入 props 可能会稍显昂贵
- 子组件的灵活性和能力较低

适用情况

- B 应该接受未来或其他地方组合不同于 C 的内容
- A 应该控制 C 的生命周期

B 会使用 this.props.children 来渲染 C，但 B 没有结构化的方式来知道这些子组件的用途。因此，B 可能通过向下传递额外的 props 来丰富子组件，但如果 B 需要确切知道它们是什么，#3 可能是更好的选择。

3. 使用 props 进行嵌套

（续上文）

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>评论</h1>
        <CommentList title={ListTitle}/> // prop
        <CommentForm />
      </div>
    );
  }
});
```

这是 A 组合 B，且 B 为 A 提供了一个选项，让 A 传入某些内容以实现特定目的的组合方式。更结构化的组合。

优点

- 作为特性组合
- 易于验证
- 更好的可组合性

缺点

- 注入 props 可能会稍显昂贵
- 子组件的灵活性和能力较低

适用情况

- B 有定义用于组合某物的特定特性
- B 只应知道如何渲染，而不是渲染什么

#3 通常是制作公共组件库的必备条件，但在一般情况下也是制作可组合组件并明确定义组合特性的良好实践。#1 是制作可用功能最简单快捷的方法，但 #2 和 #3 应该在各种用例中提供一定的优势。

第2.4节：Props（属性）

Props 是传递信息给 React 组件的一种方式，它们可以是任何类型，包括函数——有时称为回调函数。

在 JSX 中，props 通过属性语法传递

- Injecting props can become a little expensive
- Less flexibility and power in child components

Good if

- B should accept to compose something different than C in the future or somewhere else
- A should control the lifecycle of C

B would render C using `this.props.children`, and there isn't a structured way for B to know what those children are for. So, B may enrich the child components by giving additional props down, but if B needs to know exactly what they are, #3 might be a better option.

3. Nesting using props

(continued from above)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList title={ListTitle}/> //prop
        <CommentForm />
      </div>
    );
  }
});
```

This is the style where A composes B and B provides an option for A to pass something to compose for a specific purpose. More structured composition.

Pros

- Composition as a feature
- Easy validation
- Better composaiblility

Cons

- Injecting props can become a little expensive
- Less flexibility and power in child components

Good if

- B has specific features defined to compose something
- B should only know how to render not what to render

#3 is usually a must for making a public library of components but also a good practice in general to make composable components and clearly define the composition features. #1 is the easiest and fastest to make something that works, but #2 and #3 should provide certain benefits in various use cases.

Section 2.4: Props

Props are a way to pass information into a React component, they can have any type including functions - sometimes referred to as callbacks.

In JSX props are passed with the attribute syntax

```
<MyComponent userID={123} />
```

在 MyComponent 的定义内部，userID 现在可以通过 props 对象访问

```
// MyComponent 内的渲染函数
render() {
    return (
        <span>用户的ID是{this.props.userID}</span>
    )
}
```

定义所有props及其类型，并在适用时定义默认值非常重要：

```
// 在MyComponent底部定义
MyComponent.propTypes = {
    someObject: React.PropTypes.object,
    userID: React.PropTypes.number.isRequired,
    title: React.PropTypes.string
};

MyComponent.defaultProps = {
    someObject: {},
title: '我的默认标题'
}
```

在此示例中，prop someObject 是可选的，但prop userID 是必需的。如果你未向MyComponent提供 userID，React引擎将在运行时控制台警告你缺少必需的prop。请注意，这个警告仅在React库的开发版本中显示，生产版本不会记录任何警告。

使用 defaultProps 可以简化

```
const { title = '我的默认标题' } = this.props;
console.log(title);
```

到

```
console.log(this.props.title);
```

这也是使用对象数组和函数的一个保障。如果你没有为对象提供默认属性，如果没有传递该属性，下面的代码将会抛出错误：

```
if (this.props.someObject.someKey)
```

在上面的例子中，this.props.someObject是undefined，因此对someKey的检查会抛出错误，代码会中断。通过使用defaultProps，你可以安全地使用上述检查。

第2.5节：组件状态 - 动态用户界面

假设我们想要以下行为——我们有一个标题（比如h3元素），点击它时，我们希望它变成一个输入框，以便修改标题名称。React通过组件状态和if else语句使这变得非常简单和直观。（代码说明如下）

```
// 我使用了ReactBootstrap元素。但代码同样适用于普通的html元素
var Button = ReactBootstrap.Button;
var Form = ReactBootstrap.Form;
```

```
<MyComponent userID={123} />
```

Inside the definition for MyComponent userID will now be accessible from the props object

```
// The render function inside MyComponent
render() {
    return (
        <span>The user's ID is {this.props.userID}</span>
    )
}
```

It's important to define all props, their types, and where applicable, their default value:

```
// defined at the bottom of MyComponent
MyComponent.propTypes = {
    someObject: React.PropTypes.object,
    userID: React.PropTypes.number.isRequired,
    title: React.PropTypes.string
};

MyComponent.defaultProps = {
    someObject: {},
    title: 'My Default Title'
}
```

In this example the prop someObject is optional, but the prop userID is required. If you fail to provide userID to MyComponent, at runtime the React engine will show a console warning you that the required prop was not provided. Beware though, this warning is only shown in the development version of the React library, the production version will not log any warnings.

Using defaultProps allows you to simplify

```
const { title = 'My Default Title' } = this.props;
console.log(title);
```

to

```
console.log(this.props.title);
```

It's also a safeguard for use of object array and functions. If you do not provide a default prop for an object, the following will throw an error if the prop is not passed:

```
if (this.props.someObject.someKey)
```

In example above, this.props.someObject is undefined and therefore the check of someKey will throw an error and the code will break. With the use of defaultProps you can safely use the above check.

Section 2.5: Component states - Dynamic user-interface

Suppose we want to have the following behaviour - We have a heading (say h3 element) and on clicking it, we want it to become an input box so that we can modify heading name. React makes this highly simple and intuitive using component states and if else statements. (Code explanation below)

```
// I have used ReactBootstrap elements. But the code works with regular html elements also
var Button = ReactBootstrap.Button;
var Form = ReactBootstrap.Form;
var FormGroup = ReactBootstrap.FormGroup;
var FormControl = ReactBootstrap.FormControl;

var Comment = reactCreateClass({
  getInitialState: function() {
    return {show: false, newTitle: ''};
  },

handleTitleSubmit: function() {
     //处理输入框提交的代码 - 例如，发起一个ajax请求以更改数据库中的名称

  },

handleTitleChange: function(e) {
    //更改表单输入框中的名称的代码。newTitle初始化为空字符串。我们需要
用用户当前在表单中输入的字符串更新它
    this.setState({newTitle: e.target.value});
  },

changeComponent: function() {
    // 这会切换show变量，该变量用于动态UI
    this.setState({show: !this.state.show)};
  },

render: function() {

    var clickableTitle;

    if(this.state.show) {
clickableTitle = <Form inline onSubmit={this.handleTitleSubmit}>
                             <FormGroup controlId="formInlineTitle">
                                 <FormControl type="text" onChange={this.handleTitleChange}>
                             </FormGroup>
                         </Form>;
    } else {
clickabletitle = <div>
                            <Button bsStyle="link" onClick={this.changeComponent}>
                                <h3> Default Text </h3>
                            </Button>
                         </div>;
    }

    return (
        <div className="comment">
            {clickableTitle}
        </div>
    );
  }
});

ReactDOM.render(
    <Comment />, document.getElementById('content')
);
```

代码的主要部分是**clickableTitle**变量。根据状态变量**show**，它可以是一个表单元素或按钮元素。React允许组件嵌套。

因此我们可以在渲染函数中添加一个{clickableTitle}元素。它查找clickableTitle变量。根据值'this.state.show'，它显示相应的元素。

```
var FormGroup = ReactBootstrap.FormGroup;
var FormControl = ReactBootstrap.FormControl;

var Comment = reactCreateClass({
  getInitialState: function() {
    return {show: false, newTitle: ''};
  },

  handleTitleSubmit: function() {
     //code to handle input box submit - for example, issue an ajax request to change name in
database
  },

  handleTitleChange: function(e) {
    //code to change the name in form input box. newTitle is initialized as empty string. We need to
update it with the string currently entered by user in the form
    this.setState({newTitle: e.target.value});
  },

  changeComponent: function() {
    // this toggles the show variable which is used  for dynamic UI
    this.setState({show: !this.state.show)};
  },

  render: function() {

    var clickableTitle;

    if(this.state.show) {
        clickableTitle = <Form inline onSubmit={this.handleTitleSubmit}>
                             <FormGroup controlId="formInlineTitle">
                                 <FormControl type="text" onChange={this.handleTitleChange}>
                             </FormGroup>
                         </Form>;
    } else {
        clickabletitle = <div>
                            <Button bsStyle="link" onClick={this.changeComponent}>
                                <h3> Default Text </h3>
                            </Button>
                         </div>;
    }

    return (
        <div className="comment">
            {clickableTitle}
        </div>
    );
  }
});

ReactDOM.render(
    <Comment />, document.getElementById('content')
);
```

The main part of the code is the **clickableTitle** variable. Based on the state variable **show**, it can be either be a Form element or a Button element. React allows nesting of components.

So we can add a {clickableTitle} element in the render function. It looks for the clickableTitle variable. Based on the value 'this.state.show', it displays the corresponding element.

第2.6节：无状态函数组件的变体

```
const languages = [
  'JavaScript',
  'Python',
  'Java',
  'Elm',
  'TypeScript',
  'C#',
  'F#'
]
```

```
// 一行代码
const Language = ({language}) => <li>{language}</li>

Language.propTypes = {
message: React.PropTypes.string.isRequired
}
```

```
/**
* 如果有多行代码。
* 请注意圆括号在这里是可选的，
* 但为了可读性最好使用它们
*/
const LanguagesList = ({languages}) => {
  <ul>
    {languages.map(language => <Language language={language} />)}
  </ul>
}

LanguagesList.PropTypes = {
  languages: React.PropTypes.array.isRequired
}
```

```
/**
* 如果除了 JSX 展示之外还有其他工作需要做，使用这种语法
 * 例如需要进行一些数据处理。
* 请注意，return 后必须有圆括号，
 * 否则 return 将不会返回任何内容（未定义）
 */
const LanguageSection = ({header, languages}) => {
  // 做一些工作
  const formattedLanguages = languages.map(language => language.toUpperCase())
  return (
    <fieldset>
      <legend>{header}</legend>
      <LanguagesList languages={formattedLanguages} />
    </fieldset>
  )
}

LanguageSection.PropTypes = {
  header: React.PropTypes.string.isRequired,
  languages: React.PropTypes.array.isRequired
}
```

```
ReactDOM.render(
  <LanguageSection
```

Section 2.6: Variations of Stateless Functional Components

```
const languages = [
  'JavaScript',
  'Python',
  'Java',
  'Elm',
  'TypeScript',
  'C#',
  'F#'
]
```

```
// one liner
const Language = ({language}) => <li>{language}</li>

Language.propTypes = {
  message: React.PropTypes.string.isRequired
}
```

```
/**
* If there are more than one line.
* Please notice that round brackets are optional here,
* However it's better to use them for readability
*/
const LanguagesList = ({languages}) => {
  <ul>
    {languages.map(language => <Language language={language} />)}
  </ul>
}

LanguagesList.PropTypes = {
  languages: React.PropTypes.array.isRequired
}
```

```
/**
 * This syntax is used if there are more work beside just JSX presentation
 * For instance some data manipulations needs to be done.
 * Please notice that round brackets after return are required,
 * Otherwise return will return nothing (undefined)
 */
const LanguageSection = ({header, languages}) => {
  // do some work
  const formattedLanguages = languages.map(language => language.toUpperCase())
  return (
    <fieldset>
      <legend>{header}</legend>
      <LanguagesList languages={formattedLanguages} />
    </fieldset>
  )
}

LanguageSection.PropTypes = {
  header: React.PropTypes.string.isRequired,
  languages: React.PropTypes.array.isRequired
}
```

```
ReactDOM.render(
  <LanguageSection
header="Languages"
    languages={languages} />,
  document.getElementById('app')
)
```

这里你可以找到它的工作示例。

第2.7节：setState的陷阱

在异步环境中使用setState时应谨慎。例如，你可能会尝试在get请求的回调中调用setState：

```
class 我的类 extends React.Component {
    constructor() {
          super();

          this.state = {
             用户: {}
          };
     }

componentDidMount() {
          this.fetchUser();
     }

fetchUser() {
         $.get('/api/users/self')
              .then((用户) => {
                      this.setState({用户: 用户});
                 });
     }

render() {
          return <h1>{this.state.user}</h1>;
     }
}
```

这可能会引发问题——如果回调在组件卸载后被调用，那么this.setState将不再是一个函数。每当出现这种情况时，你应该小心确保对setState的使用是可取消的。

在这个例子中，你可能希望在组件卸载时取消XHR请求：

```
class 我的类 extends React.Component {
    constructor() {
          super();

          this.state = {
               user: {},
               xhr: null
          };
     }

componentWillUnmount() {
          let xhr = this.state.xhr;

          // 取消xhr请求，这样回调就不会被调用
          if (xhr && xhr.readyState != 4) {
xhr.abort();
          }
     }
```

```
    header="Languages"
    languages={languages} />,
  document.getElementById('app')
)
```

Here you can find working example of it.

Section 2.7: setState pitfalls

You should use caution when using `setState` in an asynchronous context. For example, you might try to call `setState` in the callback of a get request:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {}
        };
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });
    }

    render() {
        return <h1>{this.state.user}</h1>;
    }
}
```

This could call problems - if the callback is called after the `Component` is dismounted, then `this.setState` won't be a function. Whenever this is the case, you should be careful to ensure your usage of `setState` is cancellable.

In this example, you might wish to cancel the XHR request when the component dismounts:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {},
            xhr: null
        };
    }

    componentWillUnmount() {
        let xhr = this.state.xhr;

        // Cancel the xhr request, so the callback is never called
        if (xhr && xhr.readyState != 4) {
            xhr.abort();
        }
    }
componentDidMount() {
        this.fetchUser();
    }

fetchUser() {
        let xhr = $.get('/api/users/self')
            .then((user) => {
                this.setState({用户: 用户});
            });

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

异步方法被保存为状态。在componentWillUnmount中，你执行所有清理工作——包括取消XHR请求。

你也可以做更复杂的事情。在这个例子中，我创建了一个接受this对象作为参数的'stateSetter'函数，并在调用cancel函数后阻止this.setState：

```
function stateSetter(context) {
    var cancelled = false;
    return {
cancel: function () {
            cancelled = true;
        },
setState(newState) {
            if (!cancelled) {
context.setState(newState);
            }
        }
    }
}

class Component extends React.Component {
    constructor(props) {
        super(props);
        this.setter = stateSetter(this);
        this.state = {
user: 'loading'
        };
    }
componentWillUnmount() {
        this.setter.cancel();
    }
componentDidMount() {
        this.fetchUser();
    }
fetchUser() {
       $.get('/api/users/self')
          .then((用户) => {
                this.setter.setState({user: user});
            });
    }
render() {
        return <h1>{this.state.user}</h1>
    }
}
```

这是可行的，因为cancelled变量在我们创建的setState闭包中是可见的。

```
    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        let xhr = $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });

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

The async method is saved as a state. In the `componentWillUnmount` you perform all your cleanup - including canceling the XHR request.

You could also do something more complex. In this example, I'm creating a 'stateSetter' function that accepts the this object as an argument and prevents `this.setState` when the function `cancel` has been called:

```
function stateSetter(context) {
    var cancelled = false;
    return {
        cancel: function () {
            cancelled = true;
        },
        setState(newState) {
            if (!cancelled) {
                context.setState(newState);
            }
        }
    }
}

class Component extends React.Component {
    constructor(props) {
        super(props);
        this.setter = stateSetter(this);
        this.state = {
            user: 'loading'
        };
    }
    componentWillUnmount() {
        this.setter.cancel();
    }
    componentDidMount() {
        this.fetchUser();
    }
    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setter.setState({user: user});
            });
    }
    render() {
        return <h1>{this.state.user}</h1>
    }
}
```

This works because the `cancelled` variable is visible in the `setState` closure we created.

第3章：使用ReactJS和TypeScript

第3.1节：用TypeScript编写的ReactJS组件

实际上，你可以像Facebook的示例那样在TypeScript中使用ReactJS的组件。只需将'jsx'文件的扩展名替换为'tsx'：

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

但是为了充分利用 Typescript 的主要特性（静态类型检查），需要做几件事：

1) 将 React.createClass 示例转换为 ES6 类：

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

2) 接下来添加 Props 和 State 接口：

```
interface IHelloMessageProps {
    name:string;
}

interface IHelloMessageState {
  //在我们的例子中为空
}

class HelloMessage extends React.Component<IHelloMessageProps, IHelloMessageState> {
  constructor(){
    super();
  }
render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

如果程序员忘记传递props，或者添加了接口中未定义的props，Typescript现在会显示错误。

第3.2节：安装和设置

要在Node项目中使用Typescript和React，首先必须用npm初始化项目目录。初始化目录的命令是 npm init

通过npm或yarn安装

Chapter 3: Using ReactJS with TypeScript

Section 3.1: ReactJS component written in TypeScript

Actually you can use ReactJS's components in Typescript as in facebook's example. Just replace 'jsx' file's extension to 'tsx':

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

But in order to make full use of Typescript's main feature (static type checking) should be done couple things:

1) convert React.createClass example to ES6 Class:

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

2) next add Props and State interfaces:

```
interface IHelloMessageProps {
    name:string;
}

interface IHelloMessageState {
  //empty in our case
}

class HelloMessage extends React.Component<IHelloMessageProps, IHelloMessageState> {
  constructor(){
    super();
  }
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

Now Typescript will display an error if the programmer forgets to pass props. Or if they added props that are not defined in the interface.

Section 3.2: Installation and Setup

To use typescript with react in a node project, you must first have a project directory initialized with npm. To initialize the directory with `npm init`

Installing via npm or yarn

你可以通过 npm 执行以下操作来安装React：

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

Facebook 发布了自己的包管理器，名为 Yarn，也可以用来安装 React。安装 Yarn 后，只需运行以下命令：

```
yarn add react react-dom
```

然后你可以在项目中以完全相同的方式使用 React，就像通过 npm 安装 React 一样。

在Typescript 2.0+中安装react类型定义

要使用typescript编译代码，请使用npm或yarn添加/安装类型定义文件。

```
npm install --save-dev @types/react @types/react-dom
```

或者，使用 yarn

```
yarn add --dev @types/react @types/react-dom
```

在旧版本 Typescript 中安装 react 类型定义

你必须使用一个名为 tsd 的独立包

```
tsd install react react-dom --save
```

添加或更改 Typescript 配置

要使用 JSX（一种将 javascript 与 html/xml 混合的语言），你需要更改 typescript 编译器配置。在项目的 typescript 配置文件（通常名为 tsconfig.json）中，你需要添加 JSX 选项，如下所示：

"compilerOptions": { "jsx": "react" },

该编译器选项基本上告诉 typescript 编译器将代码中的 JSX 标签转换为 javascript 函数调用。

为了避免 typescript 编译器将 JSX 转换为普通的 javascript 函数调用，使用

```
"compilerOptions": {
    "jsx": "preserve"
},
```

第3.3节：TypeScript中的无状态React组件

React组件如果是其props的纯函数且不需要任何内部状态，可以写成JavaScript函数，而不是使用标准的类语法，如：

```
import React from 'react'

const HelloWorld = (props) => (
    <h1>Hello, {props.name}!</h1>
);
```

在TypeScript中也可以使用React.SFC类实现相同功能：

You can install React using npm by doing the following:

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

Facebook released its own package manager named Yarn, which can also be used to install React. After installing Yarn you just need to run this command:

```
yarn add react react-dom
```

You can then use React in your project in exactly the same way as if you had installed React via npm.

Installing react type definitions in Typescript 2.0+

To compile your code using typescript, add/install type definition files using npm or yarn.

```
npm install --save-dev @types/react @types/react-dom
```

or, using yarn

```
yarn add --dev @types/react @types/react-dom
```

Installing react type definitions in older versions of Typescript

You have to use a separate package called tsd

```
tsd install react react-dom --save
```

Adding or Changing the Typescript configuration

To use JSX, a language mixing javascript with html/xml, you have to change the typescript compiler configuration. In the project's typescript configuration file (usually named `tsconfig.json`), you will need to add the JSX option as:

"compilerOptions": { "jsx": "react" },

That compiler option basically tells the typescript compiler to translate the JSX tags in code to javascript function calls.

To avoid typescript compiler converting JSX to plain javascript function calls, use

```
"compilerOptions": {
    "jsx": "preserve"
},
```

Section 3.3: Stateless React Components in TypeScript

React components that are pure functions of their props and do not require any internal state can be written as JavaScript functions instead of using the standard class syntax, as:

```
import React from 'react'

const HelloWorld = (props) => (
    <h1>Hello, {props.name}!</h1>
);
```

The same can be achieved in Typescript using the `React.SFC` class:

```
import * as React from 'react';

class GreeterProps {
name: string
}

const Greeter : React.SFC<GreeterProps> = props =>
    <h1>Hello, {props.name}!</h1>;
```

注意，名称React.SFC是React.StatelessComponent的别名，因此两者都可以使用。

第3.4节：无状态且无属性的组件

最简单的无状态且无属性的React组件可以写成：

```
import * as React from 'react';

const Greeter = () => <span>Hello, World!</span>
```

但是，该组件无法访问 **this**.props，因为 typescript 无法判断它是否是一个 react 组件。要访问它的props，请使用：

```
import * as React from 'react';

const Greeter: React.SFC<{}> = props => () => <span>Hello, World!</span>
```

即使组件没有明确定义属性，它现在也可以访问 **props.children**，因为所有组件本质上都有 children。

另一个类似的无状态且无属性组件的良好用例是在简单的页面模板中。以下是一个示例简单的 Page 组件，假设项目中已有假设的 Container、NavTop 和 NavBottom组件：

```
import * as React from 'react';

const Page: React.SFC<{}> = props => () =>
    <Container>
        <NavTop />
        {props.children}
        <NavBottom />
    </Container>

const LoginPage: React.SFC<{}> = props => () =>
    <Page>
登录密码: <input type="password" />
    </页面>
```

在这个例子中，Page 组件以后可以被任何其他实际页面用作基础模板。

```
import * as React from 'react';

class GreeterProps {
   name: string
}

const Greeter : React.SFC<GreeterProps> = props =>
    <h1>Hello, {props.name}!</h1>;
```

Note that, the name React.SFC is an alias for React.StatelessComponent So, either can be used.

Section 3.4: Stateless and property-less Components

The simplest react component without a state and no properties can be written as:

```
import * as React from 'react';

const Greeter = () => <span>Hello, World!</span>
```

That component, however, can't access **this**.props since typescript can't tell if it is a react component. To access its props, use:

```
import * as React from 'react';

const Greeter: React.SFC<{}> = props => () => <span>Hello, World!</span>
```

Even if the component doesn't have explicitly defined properties, it can now access **props.children** since all components inherently have children.

Another similar good use of stateless and property-less components is in simple page templating. The following is an examplinary simple Page component, assuming there are hypothetical Container, NavTop and NavBottom components already in the project:

```
import * as React from 'react';

const Page: React.SFC<{}> = props => () =>
    <Container>
        <NavTop />
        {props.children}
        <NavBottom />
    </Container>

const LoginPage: React.SFC<{}> = props => () =>
    <Page>
        Login Pass: <input type="password" />
    </Page>
```

In this example, the Page component can later be used by any other actual page as a base template.

第4章：React中的状态

第4.1节：基本状态

React组件中的状态对于管理和传递应用中的数据至关重要。它表示为一个JavaScript对象，具有*组件级别*的作用域，可以被视为组件的私有数据。

在下面的示例中，我们在组件的constructor函数中定义了一些初始状态，并在render函数中使用它。

```
class ExampleComponent extends React.Component {
  constructor(props){
    super(props);

    // 设置我们的初始状态
    this.state = {
greeting: 'Hiya Buddy!'
    };
  }

render() {
    // 我们可以通过 this.state 访问 greeting 属性
    return(
      <div>{this.state.greeting}</div>
    );
  }
}
```

第4.2节：常见反模式

你不应该将props保存到state中。这被认为是一种反模式。例如：

```
export default class 我的组件 extends React.Component {
    constructor() {
        super();

        this.state = {
            url: ''
        }

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

onChange(e) {
        this.setState({
url: this.props.url + '/days=?' + e.target.value
        });
    }

componentWillMount() {
        this.setState({url: this.props.url});
    }

render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />
```

Chapter 4: State in React

Section 4.1: Basic State

State in React components is essential to manage and communicate data in your application. It is represented as a JavaScript object and has *component level* scope, it can be thought of as the private data of your component.

In the example below we are defining some initial state in the `constructor` function of our component and make use of it in the `render` function.

```
class ExampleComponent extends React.Component {
  constructor(props){
    super(props);

    // Set-up our initial state
    this.state = {
      greeting: 'Hiya Buddy!'
    };
  }

  render() {
    // We can access the greeting property through this.state
    return(
      <div>{this.state.greeting}</div>
    );
  }
}
```

Section 4.2: Common Antipattern

You should not save `props` into `state`. It is considered an anti-pattern. For example:

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            url: ''
        }

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

    onChange(e) {
        this.setState({
            url: this.props.url + '/days=?' + e.target.value
        });
    }

    componentWillMount() {
        this.setState({url: this.props.url});
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />
URL: {this.state.url}
            </div>
        )
    }
}
```

属性url被保存在state中然后被修改。相反，选择将更改保存到状态中，然后使用state和props一起构建完整路径：

```
export default class 我的组件 extends React.Component {
    constructor() {
        super();

        this.state = {
            days: ''
        }

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

onChange(e) {
        this.setState({
days: e.target.value
        });
    }

render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

URL: {this.props.url + '/days?=' + this.state.days}
            </div>
        )
    }
}
```

这是因为在React应用中，我们希望有一个单一的真实数据源——即所有数据由一个组件负责，且仅由一个组件负责。该组件负责将数据存储在其状态中，并通过props将数据分发给其他组件。

在第一个例子中，MyComponent类及其父组件都在各自的状态中维护着'url'。如果我们在MyComponent中更新s tate.url，这些更改不会反映到父组件中。我们失去了单一真实数据源，且跟踪数据在应用中的流动变得越来越困难。与此形成对比的是第二个例子——url仅维护在父组件的状态中，并作为prop使用于

MyComponent - 因此我们保持单一的真实数据源。

第4.3节：setState()

更新React应用程序UI的主要方式是调用setState()函数。该函数会对你提供的新状态和之前的状态执行*浅合并*，并触发组件及其所有子孙组件的重新渲染。

参数

1. updater：它可以是一个包含多个键值对的对象，这些键值对将合并到状态中，或者是一个返回此类对象的函数。

```
                URL: {this.state.url}
            </div>
        )
    }
}
```

The prop `url` is saved on `state` and then modified. Instead, choose to save the changes to a state, and then build the full path using both `state` and `props`:

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            days: ''
        }

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

    onChange(e) {
        this.setState({
            days: e.target.value
        });
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.props.url + '/days?=' + this.state.days}
            </div>
        )
    }
}
```

This is because in a React application we want to have a single source of truth - i.e. all data is the responsibility of one single component, and only one component. It is the responsibility of this component to store the data within its state, and distribute the data to other components via props.

In the first example, both the MyComponent class and its parent are maintaining 'url' within their state. If we update state.url in MyComponent, these changes are not reflected in the parent. We have lost our single source of truth, and it becomes increasingly difficult to track the flow of data through our application. Contrast this with the second example - url is only maintained in the state of the parent component, and utilised as a prop in MyComponent - we therefore maintain a single source of truth.

Section 4.3: setState()

The primary way that you make UI updates to your React applications is through a call to the `setState()` function. This function will perform a *shallow merge* between the new state that you provide and the previous state, and will trigger a re-render of your component and all decedents.

Parameters

1. `updater`: It can be an object with a number of key-value pairs that should be merged into the state or a function that returns such an object.

2. callback（可选）：一个函数，在setState()成功执行后调用。
   由于 React 并不保证对 setState() 的调用是原子的，因此如果你想在确认 setState() 已成功执行后执行某些操作，这有时会很有用。

用法：

setState 方法接受一个 updater 参数，该参数可以是一个包含多个键值对的对象，这些键值对将合并到状态中，或者是一个函数，该函数根据 prevState 和 props 计算并返回这样的对象。

使用 setState() 并传入一个对象作为 updater

```
//
// 一个 ES6 风格的组件示例，在简单的按钮点击时更新状态。
// 还演示了何时可以直接设置状态，何时应使用 setState。
//
class Greeting extends React.Component {
    constructor(props) {
        super(props);
        this.click = this.click.bind(this);
        // 设置初始状态（仅允许在构造函数中）
        this.state = {
greeting: 'Hello!'
        };
    }
click(e) {
        this.setState({
greeting: 'Hello World!'
        });
    }
render() {
        return(
            <div>
                <p>{this.state.greeting}</p>
                <button onClick={this.click}>点击我</button>
            </div>
        );
    }

}
```

使用setState()并传入函数作为updater

```
//
// 这通常用于在更新任何值之前检查或利用之前的状态。

//

this.setState(function(previousState, currentProps) {
  return {
counter: previousState.counter + 1
  };
});
```

这比使用对象参数更安全，因为多次调用setState()可能会被React合并并一次性执行，当使用当前属性设置状态时，这是推荐的方法。

```
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
```

2. `callback` (`optional`): a function which will be executed after `setState()` has been executed successfully. Due to the fact that calls to `setState()` are not guaranteed by React to be atomic, this can sometimes be useful if you want to perform some action after you are positive that `setState()` has been executed successfully.

Usage:

The `setState` method accepts an `updater` argument that can either be an object with a number of key-value-pairs that should be merged into the state, or a function that returns such an object computed from `prevState` and `props`.

Using `setState()` with an Object as `updater`

```
//
// An example ES6 style component, updating the state on a simple button click.
// Also demonstrates where the state can be set directly and where setState should be used.
//
class Greeting extends React.Component {
    constructor(props) {
        super(props);
        this.click = this.click.bind(this);
        // Set initial state (ONLY ALLOWED IN CONSTRUCTOR)
        this.state = {
            greeting: 'Hello!'
        };
    }
    click(e) {
        this.setState({
              greeting: 'Hello World!'
        });
    }
    render() {
        return(
            <div>
                <p>{this.state.greeting}</p>
                <button onClick={this.click}>Click me</button>
            </div>
        );
    }

}
```

Using `setState()` with a Function as `updater`

```
//
// This is most often used when you want to check or make use
// of previous state before updating any values.
//

this.setState(function(previousState, currentProps) {
  return {
    counter: previousState.counter + 1
  };
});
```

This can be safer than using an object argument where multiple calls to `setState()` are used, as multiple calls may be batched together by React and executed at once, and is the preferred approach when using current props to set state.

```
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
```

这些调用可能会被 React 使用Object.assign()批量处理，导致计数器增加了1而不是3。

函数式方法也可以用来将状态设置逻辑移出组件。这允许状态逻辑的隔离和重用。

```
// 组件类外部，可能在另一个文件/模块中

function incrementCounter(previousState, currentProps) {
    return {
counter: previousState.counter + 1
    };
}

// 组件内部

this.setState(incrementCounter);
```

调用setState()，传入一个对象和一个回调函数

```
//
// setState 完成后，控制台将输出 'Hi There'
//

this.setState({ name: 'John Doe' }, console.log('Hi there'));
```

第4.4节：状态、事件和受控组件

这是一个带有“受控”输入字段的 React 组件示例。每当输入字段的值发生变化时，都会调用一个事件处理程序，该处理程序使用输入字段的新值更新组件的状态。
事件处理程序中对setState的调用将触发render的调用，从而更新 DOM 中的组件。

```
import React from 'react';
import {render} from 'react-dom';


class ManagedControlDemo extends React.Component {

  constructor(props){
    super(props);
    this.state = {message: ""};
  }

handleChange(e){
    this.setState({message: e.target.value});
  }

render() {
    return (
      <div>
        <legend>在此输入内容</legend>
          <input
onChange={this.handleChange.bind(this)}
            value={this.state.message}
autoFocus />
        <h1>{this.state.message}</h1>
      </div>
   );
```

```
this.setState({ counter: this.state.counter + 1 });
```

These calls may be batched together by React using `Object.assign()`, resulting in the counter being incremented by 1 rather than 3.

The functional approach can also be used to move state setting logic outside of components. This allows for isolation and re-use of state logic.

```
// Outside of component class, potentially in another file/module

function incrementCounter(previousState, currentProps) {
    return {
        counter: previousState.counter + 1
    };
}

// Within component

this.setState(incrementCounter);
```

Calling `setState()` with an Object and a callback function

```
//
// 'Hi There' will be logged to the console after setState completes
//

this.setState({ name: 'John Doe' }, console.log('Hi there'));
```

Section 4.4: State, Events And Managed Controls

Here's an example of a React component with a "managed" input field. Whenever the value of the input field changes, an event handler is called which updates the state of the component with the new value of the input field. The call to `setState` in the event handler will trigger a call to `render` updating the component in the dom.

```
import React from 'react';
import {render} from 'react-dom';


class ManagedControlDemo extends React.Component {

  constructor(props){
    super(props);
    this.state = {message: ""};
  }

  handleChange(e){
    this.setState({message: e.target.value});
  }

  render() {
    return (
      <div>
        <legend>Type something here</legend>
          <input
            onChange={this.handleChange.bind(this)}
            value={this.state.message}
            autoFocus />
        <h1>{this.state.message}</h1>
      </div>
   );
  }
}


render(<ManagedControlDemo/>, document.querySelector('#app'));
```

非常重要的是要注意运行时的行为。每当用户更改输入字段中的值时

- `handleChange` 会被调用，因此
- `setState` 会被调用，因此
- `render` 会被调用

小测验，输入字段中输入一个字符后，哪些DOM元素会发生变化

1.所有这些——顶层div、legend、input、h1
2.只有input和h1
3.没有变化
4.DOM是什么？

你可以在这里多做实验以找到答案

```
  }
}


render(<ManagedControlDemo/>, document.querySelector('#app'));
```

Its very important to note the runtime behavior. Every time a user changes the value in the input field

- `handleChange` will be called and so
- `setState` will be called and so
- `render` will be called

Pop quiz, after you type a character in the input field, which DOM elements change

1. all of these - the top level div, legend, input, h1
2. only the input and h1
3. nothing
4. whats a DOM?

You can experiment with this more here to find the answer

第5章：React中的Props

第5.1节：介绍

props 用于从父组件向子组件传递数据和方法。

关于props的有趣事实

1. 它们是不可变的。
2. 它们允许我们创建可重用的组件。

基本示例

```
class Parent extends React.Component{
  doSomething(){
console.log("父组件");
  }
render() {
    return <div>
         <Child
text="这是子组件1号"
           title="标题1"
onClick={this.doSomething} />
         <Child
text="这是子项编号2"
           title="标题2"
onClick={this.doSomething} />
      </div>
  }
}

class Child extends React.Component{
  render() {
    return <div>
       <h1>{this.props.title}</h1>
       <h2>{this.props.text}</h2>
      </div>
  }
}
```

正如示例中所示，得益于props，我们可以创建可复用的组件。

第5.2节：默认props

defaultProps允许你为组件的props设置默认值或备用值。 defaultProps在你从不同视图调用组件时非常有用，某些视图使用固定的props，但在其他视图中你需要传递不同的值。

语法

ES5

```
var MyClass = React.createClass({
  getDefaultProps: function() {
    return {
randomObject: {},
```

Chapter 5: Props in React

Section 5.1: Introduction

props are used to pass data and methods from a parent component to a child component.

Interesting things about props

1. They are immutable.
2. They allow us to create reusable components.

Basic example

```
class Parent extends React.Component{
  doSomething(){
     console.log("Parent component");
  }
  render() {
    return <div>
         <Child
           text="This is the child number 1"
           title="Title 1"
           onClick={this.doSomething} />
         <Child
           text="This is the child number 2"
           title="Title 2"
           onClick={this.doSomething} />
      </div>
  }
}

class Child extends React.Component{
  render() {
    return <div>
       <h1>{this.props.title}</h1>
       <h2>{this.props.text}</h2>
      </div>
  }
}
```

As you can see in the example, thanks to props we can create reusable components.

Section 5.2: Default props

defaultProps allows you to set default, or fallback, values for your component props. defaultProps are useful when you call components from different views with fixed props, but in some views you need to pass different value.

Syntax

ES5

```
var MyClass = React.createClass({
  getDefaultProps: function() {
    return {
      randomObject: {},
      ...
    };
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.defaultProps = {
randomObject: {},
    ...
}
```

ES7

```
class MyClass extends React.Component {
    static defaultProps = {
        randomObject: {},
      ...
    };
}
```

getDefaultProps() 或 defaultProps 的结果将被缓存并用于确保 `this.props.randomObject` 在父组件未指定时会有一个默认值。

第5.3节：PropTypes

`propTypes` 允许你指定组件所需的 `props` 及其类型。你的组件即使不设置 `propTypes` 也能工作，但定义它们是良好实践，因为这会使你的组件更易读，作为给其他开发者的文档，并且在开发过程中，如果你尝试设置与定义类型不同的prop，React会发出警告。

一些原始的propTypes和常用的propTypes有 -

```
optionalArray: React.PropTypes.array,
  optionalBool: React.PropTypes.bool,
  optionalFunc: React.PropTypes.func,
  optionalNumber: React.PropTypes.number,
  optionalObject: React.PropTypes.object,
  optionalString: React.PropTypes.string,
  optionalSymbol: React.PropTypes.symbol
```

如果你给任何propType附加了isRequired，那么在创建该组件实例时必须提供该属性。如果你不提供required的propTypes，则无法创建组件实例。

语法

ES5

```
var MyClass = React.createClass({
  propTypes: {
randomObject: React.PropTypes.object,
```

```
      ...
    };
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.defaultProps = {
    randomObject: {},
    ...
}
```

ES7

```
class MyClass extends React.Component {
    static defaultProps = {
        randomObject: {},
        ...
    };
}
```

The result of `getDefaultProps()` or `defaultProps` will be cached and used to ensure that `this.props.randomObject` will have a value if it was not specified by the parent component.

Section 5.3: PropTypes

`propTypes` allows you to specify what `props` your component needs and the type they should be. Your component will work without setting `propTypes`, but it is good practice to define these as it will make your component more readable, act as documentation to other developers who are reading your component, and during development, React will warn you if you you try to set a prop which is a different type to the definition you have set for it.

Some primitive `propTypes` and commonly useable `propTypes` are -

```
  optionalArray: React.PropTypes.array,
  optionalBool: React.PropTypes.bool,
  optionalFunc: React.PropTypes.func,
  optionalNumber: React.PropTypes.number,
  optionalObject: React.PropTypes.object,
  optionalString: React.PropTypes.string,
  optionalSymbol: React.PropTypes.symbol
```

If you attach `isRequired` to any `propType` then that prop must be supplied while creating the instance of that component. If you don't provide the **required** `propTypes` then component instance can not be created.

Syntax

ES5

```
var MyClass = React.createClass({
  propTypes: {
    randomObject: React.PropTypes.object,
callback: React.PropTypes.func.isRequired,
    …
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.propTypes = {
randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    …
};
```

ES7

```
class MyClass extends React.Component {
      static propTypes = {
randomObject: React.PropTypes.object,
          callback: React.PropTypes.func.isRequired,
        …
    };
}
```

更复杂的属性验证

同样，PropTypes 允许你指定更复杂的验证

验证一个对象

```
...
    randomObject: React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    }).isRequired,
…
```

验证对象数组

```
...
    arrayOfObjects: React.PropTypes.arrayOf(React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    })).isRequired,
…
```

第5.4节：使用展开运算符传递props

替代写法

```
var component = <Component foo={this.props.x} bar={this.props.y} />;
```

当每个属性需要作为单个属性值传递时，您可以使用 ES6 中支持的数组展开运算符 … 来传递所有值。组件现在将如下所示。

```
    callback: React.PropTypes.func.isRequired,
    ...
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.propTypes = {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
};
```

ES7

```
class MyClass extends React.Component {
     static propTypes = {
        randomObject: React.PropTypes.object,
        callback: React.PropTypes.func.isRequired,
        ...
    };
}
```

More complex props validation

In the same way, PropTypes allows you to specify more complex validation

Validating an object

```
...
    randomObject: React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    }).isRequired,
...
```

Validating on array of objects

```
...
    arrayOfObjects: React.PropTypes.arrayOf(React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    })).isRequired,
...
```

Section 5.4: Passing down props using spread operator

Instead of

```
var component = <Component foo={this.props.x} bar={this.props.y} />;
```

Where each property needs to be passed as a single prop value you could use the spread operator ... supported for arrays in ES6 to pass down all your values. The component will now look like this.

```
var component = <Component {...props} />;
```

请记住，传入对象的属性会被复制到组件的 props 上。

顺序很重要。后面的属性会覆盖前面的属性。

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

另一种情况是，你也可以使用展开运算符只传递部分 props 给子组件，然后你
可以再次使用 props 的解构语法。

当子组件需要很多 props 但不想一个一个传递时，这非常有用。

```
const { foo, bar, other } = this.props // { foo: 'foo', bar: 'bar', other: 'other' };
var component = <Component {...{foo, bar}} />;
const { foo, bar } = component.props
console.log(foo, bar); // 'foo bar'
```

第5.5节：Props.children 和组件组合

组件的“子”组件可通过一个特殊的属性props.children访问。这个属性对于“组合”组件非常有用，且可以使JSX标记更直观，或更能反映最终DOM的预期结构：

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {this.props.children}
            </div>
        </article>
    );
}
```

这使我们在后续使用该组件时，可以包含任意数量的子元素：

```
var ParentComponent = function () {
    return (
        <SomeComponent heading="惊人的文章框" >
            <p className="first"> 大量内容 </p>
            <p> 或者没有 </p>
        </SomeComponent>
    );
}
```

组件也可以操作props.children。由于props.children可能是数组，也可能不是数组，React为其提供了React.Children 工具函数。假设在前面的例子中，我们想要将每个段落包裹在自己的<section>元素中：

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
```

```
var component = <Component {...props} />;
```

Remember that the properties of the object that you pass in are copied onto the component's props.

The order is important. Later attributes override previous ones.

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

Another case is that you also can use spread operator to pass only parts of props to children components, then you can use destructuring syntax from props again.

It's very useful when children components need lots of props but not want pass them one by one.

```
const { foo, bar, other } = this.props // { foo: 'foo', bar: 'bar', other: 'other' };
var component = <Component {...{foo, bar}} />;
const { foo, bar } = component.props
console.log(foo, bar); // 'foo bar'
```

Section 5.5: Props.children and component composition

The "child" components of a component are available on a special prop, `props.children`. This prop is very useful for "Compositing" components together, and can make JSX markup more intuitive or reflective of the intended final structure of the DOM:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {this.props.children}
            </div>
        </article>
    );
}
```

Which allows us to include an arbitrary number of sub-elements when using the component later:

```
var ParentComponent = function () {
    return (
        <SomeComponent heading="Amazing Article Box" >
            <p className="first"> Lots of content </p>
            <p> Or not </p>
        </SomeComponent>
    );
}
```

Props.children can also be manipulated by the component. Because props.children may or may not be an array, React provides utility functions for them as React.Children. Consider in the previous example if we had wanted to wrap each paragraph in its own **`<section>`** element:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
```

```
                {React.Children.map(this.props.children, function (child) {
                    return (
                        <section className={child.props.className}>
                       React.cloneElement(child)
                        </section>
                    );
                })}
            </div>
        </article>
    );
}
```

注意使用React.cloneElement来移除子元素<p>标签的props——因为props是不可变的，这些值不能被直接更改。相反，必须使用一个没有这些props的克隆元素。

此外，在循环中添加元素时，要注意React在重新渲染时如何协调子元素，并强烈建议在循环中添加的子元素上包含一个全局唯一的key属性。

第5.6节：检测子组件的类型

有时在遍历子组件时，了解组件的类型非常有用。为了遍历子组件，你可以使用React.Children.map工具函数：

```
React.Children.map(this.props.children, (child) => {
  if (child.type === MyComponentType) {
    ...
  }
});
```

子对象暴露了type属性，你可以将其与特定组件进行比较。

```
                {React.Children.map(this.props.children, function (child) {
                    return (
                        <section className={child.props.className}>
                            React.cloneElement(child)
                        </section>
                    );
                })}
            </div>
        </article>
    );
}
```

Note the use of React.cloneElement to remove the props from the child **<p>** tag - because props are immutable, these values cannot be changed directly. Instead, a clone without these props must be used.

Additionally, when adding elements in loops, be aware of how React reconciles children during a rerender, and strongly consider including a globally unique key prop on child elements added in a loop.

Section 5.6: Detecting the type of Children components

Sometimes it's really useful to know the type of child component when iterating through them. In order to iterate through the children components you can use React Children.map util function:

```
React.Children.map(this.props.children, (child) => {
  if (child.type === MyComponentType) {
    ...
  }
});
```

The child object exposes the type property which you can compare to a specific component.

第6章：React组件生命周期

生命周期方法用于在组件生命周期的不同阶段运行代码并与组件交互。这些方法围绕组件的挂载、更新和卸载进行。

第6.1节：组件创建

当创建一个React组件时，会调用多个函数：

- 如果你使用React.createClass（ES5），会调用5个用户定义的函数
- 如果你使用class Component extends React.Component（ES6），会调用3个用户定义的函数

getDefaultProps()（仅ES5）

这是调用的第一个方法。

如果组件实例化时未定义属性值，则该函数返回的属性值将作为默认值使用。

在以下示例中，如果未另行指定，this.props.name将默认为Bob：

```
getDefaultProps() {
  return {
initialCount: 0,
    name: 'Bob'
  };
}
```

getInitialState()（仅限ES5）

这是**第二个**被调用的方法。

getInitialState()的返回值定义了React组件的初始状态。React框架将调用此函数并将返回值赋给this.state。

在下面的示例中，this.state.count将被初始化为this.props.initialCount的值：

```
getInitialState() {
  return {
count : this.props.initialCount
  };
}
```

componentWillMount()（ES5和ES6）

这是**第三个**被调用的方法。

此函数可用于在组件被添加到DOM之前进行最终修改。

```
componentWillMount() {
  ...
}
```

render()（ES5和ES6）

这是**第四个**被调用的方法。

render()函数应该是组件状态和属性的纯函数。它返回一个单一元素

Chapter 6: React Component Lifecycle

Lifecycle methods are to be used to run code and interact with your component at different points in the components life. These methods are based around a component Mounting, Updating, and Unmounting.

Section 6.1: Component Creation

When a React component is created, a number of functions are called:

- If you are using React.createClass (ES5), 5 user defined functions are called
- If you are using class Component extends React.Component (ES6), 3 user defined functions are called

getDefaultProps() (ES5 only)

This is the **first** method called.

Prop values returned by this function will be used as defaults if they are not defined when the component is instantiated.

In the following example, this.props.name will be defaulted to Bob if not specified otherwise:

```
getDefaultProps() {
  return {
    initialCount: 0,
    name: 'Bob'
  };
}
```

getInitialState() (ES5 only)

This is the **second** method called.

The return value of getInitialState() defines the initial state of the React component. The React framework will call this function and assign the return value to this.state.

In the following example, this.state.count will be intialized with the value of this.props.initialCount:

```
getInitialState() {
  return {
    count : this.props.initialCount
  };
}
```

componentWillMount() (ES5 and ES6)

This is the **third** method called.

This function can be used to make final changes to the component before it will be added to the DOM.

```
componentWillMount() {
  ...
}
```

render() (ES5 and ES6)

This is the **fourth** method called.

The render() function should be a pure function of the component's state and props. It returns a single element

该元素表示渲染过程中的组件，应当是原生
DOM组件的表示（例如`<p />`）或复合组件。如果不需要渲染任何内容，可以返回`null`或
`undefined`。

该函数会在组件的属性或状态发生任何变化后被重新调用。

```
render() {
   return (
      <div>
Hello, {this.props.name}!
      </div>
   );
}
```

`componentDidMount()`（ES5和ES6）

这是调用的第五个方法。

组件已挂载，现在可以访问组件的DOM节点，例如通过refs。

该方法应用于：

- 准备定时器
- 获取数据
- 添加事件监听器
- 操作DOM元素

```
componentDidMount() {
  ...
}
```

ES6 语法

如果组件是使用 ES6 类语法定义的，则不能使用函数getDefaultProps()和getInitialState()。

相反，我们将defaultProps声明为类的静态属性，并在类的构造函数中声明状态的结构和初始状态。这两者都在类实例构造时设置，在调用任何其他 React 生命周期函数之前完成。

下面的示例演示了这种替代方法：

```
class MyReactClass extends React.Component {
  constructor(props){
    super(props);

    this.state = {
count: this.props.initialCount
    };
  }

upCount() {
    this.setState((prevState) => ({
      count: prevState.count + 1
    }));
  }

render() {
    return (
      <div>
```

which represents the component during the rendering process and should either be a representation of a native DOM component (e.g. `<p />`) or a composite component. If nothing should be rendered, it can return `null` or `undefined`.

This function will be recalled after any change to the component's props or state.

```
render() {
  return (
    <div>
      Hello, {this.props.name}!
    </div>
  );
}
```

`componentDidMount()` (ES5 and ES6)

This is the **fifth** method called.

The component has been mounted and you are now able to access the component's DOM nodes, e.g. via `refs`.

This method should be used for:

- Preparing timers
- Fetching data
- Adding event listeners
- Manipulating DOM elements

```
componentDidMount() {
  ...
}
```

ES6 Syntax

If the component is defined using ES6 class syntax, the functions `getDefaultProps()` and `getInitialState()` cannot be used.

Instead, we declare our `defaultProps` as a static property on the class, and declare the state shape and initial state in the constructor of our class. These are both set on the instance of the class at construction time, before any other React lifecycle function is called.

The following example demonstrates this alternative approach:

```
class MyReactClass extends React.Component {
  constructor(props){
    super(props);

    this.state = {
      count: this.props.initialCount
    };
  }

  upCount() {
    this.setState((prevState) => ({
      count: prevState.count + 1
    }));
  }

  render() {
    return (
      <div>
你好，{this.props.name}!<br />
       你点击了按钮 {this.state.count} 次。<br />
        <button onClick={this.upCount}>点击这里!</button>
      </div>
    );
  }
}

MyReactClass.defaultProps = {
  name: 'Bob',
initialCount: 0
};
```

替换 getDefaultProps()

组件属性的默认值通过设置类的 defaultProps 属性来指定：

```
MyReactClass.defaultProps = {
  name: 'Bob',
initialCount: 0
};
```

替换 getInitialState()

设置组件初始状态的惯用方法是在构造函数中设置 this.state：

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

  this.state = {
count: this.props.initialCount
  };
}
```

第6.2节：组件移除

componentWillUnmount()

该方法称为before组件从DOM中卸载。

这是执行清理操作的好地方，例如：

- 移除事件监听器。
- 清除定时器。
- 停止套接字。
- 清理redux状态。

```
componentWillUnmount(){
  ...
}
```

在componentWillUnMount中移除附加事件监听器的示例

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

export default class SideMenu extends Component {
```

```
        Hello, {this.props.name}!<br />
        You clicked the button {this.state.count} times.<br />
        <button onClick={this.upCount}>Click here!</button>
      </div>
    );
  }
}

MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Replacing getDefaultProps()

Default values for the component props are specified by setting the defaultProps property of the class:

```
MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Replacing getInitialState()

The idiomatic way to set up the initial state of the component is to set this.state in the constructor:

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

  this.state = {
    count: this.props.initialCount
  };
}
```

Section 6.2: Component Removal

componentWillUnmount()

This method is called **before** a component is unmounted from the DOM.

It is a good place to perform cleaning operations like:

- Removing event listeners.
- Clearing timers.
- Stopping sockets.
- Cleaning up redux states.

```
componentWillUnmount(){
  ...
}
```

An example of removing attached event listener in componentWillUnMount

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

export default class SideMenu extends Component {
```

```
constructor(props) {
    super(props);
    this.state = {
      ...
    };
    this.openMenu = this.openMenu.bind(this);
    this.closeMenu = this.closeMenu.bind(this);
  }

componentDidMount() {
document.addEventListener("click", this.closeMenu);
  }

componentWillUnmount() {
document.removeEventListener("click", this.closeMenu);
  }

openMenu() {
   ...
  }

closeMenu() {
   ...
  }

render() {
    return (
      <div>
        <a
href      = "javascript:void(0)"
          className = "closebtn"
onClick   = {this.closeMenu}
        >
          ×
        <a>
        <div>
         其他结构
        </div>
      </div>
    );
  }
}
```

第6.3节：组件更新

componentWillReceiveProps(nextProps)

这是属性更改时调用的第一个函数。

当组件的属性发生变化时，React 会使用新的属性调用此函数。你可以通过this.props访问旧的属性，通过nextProps访问新的属性。

使用这些变量，您可以在旧属性和新属性之间进行一些比较操作，或者因为属性变化而调用函数等。

```
componentWillReceiveProps(nextProps){
  if (nextProps.initialCount && nextProps.initialCount > this.state.count){
    this.setState({
count : nextProps.initialCount
    });
```

```
constructor(props) {
  super(props);
  this.state = {
    ...
  };
  this.openMenu = this.openMenu.bind(this);
  this.closeMenu = this.closeMenu.bind(this);
}

componentDidMount() {
  document.addEventListener("click", this.closeMenu);
}

componentWillUnmount() {
  document.removeEventListener("click", this.closeMenu);
}

openMenu() {
  ...
}

closeMenu() {
  ...
}

render() {
  return (
    <div>
      <a
        href      = "javascript:void(0)"
        className = "closebtn"
        onClick   = {this.closeMenu}
      >
        ×
      </a>
      <div>
        Some other structure
      </div>
    </div>
  );
}
}
```

Section 6.3: Component Update

componentWillReceiveProps(nextProps)

This is the **first function called on properties changes**.

When **component's properties change**, React will call this function with the **new properties**. You can access to the old props with *this.props* and to the new props with *nextProps*.

With these variables, you can do some comparison operations between old and new props, or call function because a property change, etc.

```
componentWillReceiveProps(nextProps){
  if (nextProps.initialCount && nextProps.initialCount > this.state.count){
    this.setState({
      count : nextProps.initialCount
    });
  }
}
```

shouldComponentUpdate(nextProps, nextState)

这是在属性变化时调用的第二个函数，也是状态变化时调用的第一个函数。

默认情况下，如果另一个组件/你的组件更改了你的组件的属性/状态，**React** 将渲染你的组件的新版本。在这种情况下，该函数总是返回 true。

你可以重写此函数，更精确地选择你的组件是否必须更新。

此函数主要用于优化。

如果函数返回false，更新流程将立即停止。

```
componentShouldUpdate(nextProps, nextState){
  return this.props.name !== nextProps.name ||
    this.state.count !== nextState.count;
}
```

componentWillUpdate(nextProps, nextState)

此函数的作用类似于componentWillMount()。更改不会反映在DOM中，因此您可以在更新执行之前进行一些更改。

/!\ ： 你不能使用 this.setState()。

```
componentWillUpdate(nextProps, nextState) {}
```

render()

有一些变化，所以重新渲染组件。

componentDidUpdate(prevProps, prevState)

和 componentDidMount() 一样：DOM 已刷新，所以你可以在这里对 DOM 进行一些操作。

```
componentDidUpdate(prevProps, prevState) {}
```

第6.4节：不同状态下生命周期方法的调用

此示例作为其他示例的补充，讲述如何使用生命周期方法以及方法何时被调用。

此示例总结了哪些方法（componentWillMount、componentWillReceiveProps 等）会被调用以及在不同状态下组件调用顺序的不同：

当组件初始化时：

1.getDefaultProps
2.getInitialState
3.componentWillMount
4.render
5. componentDidMount

当组件状态发生变化时：

1. shouldComponentUpdate

```
  }
}
```

shouldComponentUpdate(nextProps, nextState)

This is the **second function called on properties changes and the first on state changes**.

By default, if another component / your component change a property / a state of your component, **React** will render a new version of your component. In this case, this function always return true.

You can override this function and **choose more precisely if your component must update or not**.

This function is mostly used for **optimization**.

In case of the function returns **false**, the **update pipeline stops immediately**.

```
componentShouldUpdate(nextProps, nextState){
  return this.props.name !== nextProps.name ||
    this.state.count !== nextState.count;
}
```

componentWillUpdate(nextProps, nextState)

This function works like `componentWillMount()`. **Changes aren't in DOM**, so you can do some changes just before the update will perform.

/! : you cannot use **this.setState()**.

```
componentWillUpdate(nextProps, nextState){}
```

render()

There's some changes, so re-render the component.

componentDidUpdate(prevProps, prevState)

Same stuff as `componentDidMount()` : **DOM is refreshed**, so you can do some work on the DOM here.

```
componentDidUpdate(prevProps, prevState){}
```

Section 6.4: Lifecycle method call in different states

This example serves as a complement to other examples which talk about how to use the lifecycle methods and when the method will be called.

This example summarize Which methods (componentWillMount, componentWillReceiveProps, etc) will be called and in which sequence will be different for a component **in different states**:

When a component is initialized:

1. getDefaultProps
2. getInitialState
3. componentWillMount
4. render
5. componentDidMount

When a component has state changed:

1. shouldComponentUpdate

2.componentWillUpdate
3.render
4. componentDidUpdate

当组件属性发生变化时：

1.componentWillReceiveProps
2.shouldComponentUpdate
3.componentWillUpdate
4.render
5. componentDidUpdate

当组件卸载时：

1. componentWillUnmount

第6.5节：React组件容器

在构建React应用时，通常希望根据组件的主要职责将其划分为展示组件和容器组件。

展示组件只关注数据的展示——它们可以被视为，且通常被实现为，将模型转换为视图的函数。通常它们不维护任何内部状态。
容器组件负责管理数据。这可以通过它们自己的状态内部完成，或者通过作为状态管理库（如Redux）的中介来实现。容器组件不会直接显示数据，而是将数据传递给展示组件。

```
// 容器组件
import React, { Component } from 'react';
import Api from 'path/to/api';

class CommentsListContainer extends Component {
    constructor() {
        super();
        // 设置初始状态
        this.state = { comments: [] }
    }

componentDidMount() {
        // 发起API调用并用返回的评论更新状态
Api.getComments().then(comments => this.setState({ comments }));
    }

render() {
        // 将我们的状态评论传递给展示组件
        return (
            <CommentsList comments={this.state.comments} />;
        );
    }
}

// 展示组件
const CommentsList = ({ comments }) => (
    <div>
        {comments.map(comment => (
            <div>{comment}</div>
        )}
    </div>
);
```

2. componentWillUpdate
3. render
4. componentDidUpdate

When a component has props changed:

1. componentWillReceiveProps
2. shouldComponentUpdate
3. componentWillUpdate
4. render
5. componentDidUpdate

When a component is unmounting:

1. componentWillUnmount

Section 6.5: React Component Container

When building a React application, it is often desirable to divide components based on their primary responsibility, into Presentational and Container components.
Presentational components are concerned only with displaying data - they can be regarded as, and are often implemented as, functions that convert a model to a view. Typically they do not maintain any internal state. Container components are concerned with managing data. This may be done internally through their own state, or by acting as intermediaries with a state-management library such as Redux. The container component will not directly display data, rather it will pass the data to a presentational component.

```
// Container component
import React, { Component } from 'react';
import Api from 'path/to/api';

class CommentsListContainer extends Component {
    constructor() {
        super();
        // Set initial state
        this.state = { comments: [] }
    }

    componentDidMount() {
        // Make API call and update state with returned comments
        Api.getComments().then(comments => this.setState({ comments }));
    }

    render() {
        // Pass our state comments to the presentational component
        return (
            <CommentsList comments={this.state.comments} />;
        );
    }
}

// Presentational Component
const CommentsList = ({ comments }) => (
    <div>
        {comments.map(comment => (
            <div>{comment}</div>
        )}
    </div>
);
```

```
CommentsList.propTypes = {
comments: React.PropTypes.arrayOf(React.PropTypes.string)
}
```

```
CommentsList.propTypes = {
    comments: React.PropTypes.arrayOf(React.PropTypes.string)
}
```

第7章：表单与用户输入

第7.1节：受控组件

受控表单组件通过value属性定义。受控输入的值由
React管理，用户输入不会直接影响渲染的输入内容。相反，value属性的更改
需要反映这一变化。

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
name: ''
    };
  }

onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

render() {
    return (
      <div>
        <label for='name-input'>姓名: </label>
        <input
id='name-input'
          onChange={this.onChange}
          value={this.state.name} />
      </div>
    )
  }
}
```

上述示例演示了value属性如何定义输入的当前值，以及onChange事件处理程序如何使用用户输入更新组件的状态。

表单输入应尽可能定义为受控组件。这确保组件状态与输入值始终同步，即使值被用户输入以外的触发器更改。

第7.2节：非受控组件

非受控组件是没有value属性的输入。与受控组件相反，保持组件状态与输入值同步是应用程序的责任。

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
name: 'John'
```

Chapter 7: Forms and User Input

Section 7.1: Controlled Components

Controlled form components are defined with a `value` property. The value of controlled inputs is managed by React, user inputs will not have any direct influence on the rendered input. Instead, a change to the `value` property needs to reflect this change.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: ''
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          value={this.state.name} />
      </div>
    )
  }
}
```

The above example demonstrates how the `value` property defines the current value of the input and the `onChange` event handler updates the component's state with the user's input.

Form inputs should be defined as controlled components where possible. This ensures that the component state and the input value is in sync at all times, even if the value is changed by a trigger other than a user input.

Section 7.2: Uncontrolled Components

Uncontrolled components are inputs that do not have a `value` property. In opposite to controlled components, it is the application's responsibility to keep the component state and the input value in sync.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: 'John'
    };
  }

onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

render() {
    return (
      <div>
        <label for='name-input'>姓名: </label>
        <input
id='name-input'
          onChange={this.onChange}
          defaultValue={this.state.name} />
      </div>
    )
  }
}
```

这里，组件的状态通过onChange事件处理器更新，就像受控组件一样。
但是，提供的是defaultValue属性，而不是value属性。它决定了输入在首次渲染时的初始值。组件状态的任何后续更改不会自动反映到输入值；如果需要此功能，应使用受控组件。

```
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          defaultValue={this.state.name} />
      </div>
    )
  }
}
```

Here, the component's state is updated via the `onChange` event handler, just as for controlled components. However, instead of a `value` property, a `defaultValue` property is supplied. This determines the initial value of the input during the first render. Any subsequent changes to the component's state are not automatically reflected by the input value; If this is required, a controlled component should be used instead.

第8章：React模板 [React + Babel + Webpack]

第8.1节：react-starter项目

关于本项目

这是一个简单的样板项目。本文将指导你搭建 ReactJs + Webpack + Babel 的环境。

让我们开始吧

我们需要使用 Node 包管理器来启动 Express 服务器并管理整个项目的依赖。如果你是 Node 包管理器的新手，可以在这里查看。注意：这里需要安装 Node 包管理器。

创建一个合适名称的文件夹，并通过终端或图形界面进入该文件夹。然后在终端输入 npm init，这将创建一个 package.json 文件，不用担心，它会问你一些问题，比如项目名称、版本、描述、入口点、Git 仓库、作者、许可证等。这里入口点很重要，因为当你运行项目时，Node 会首先查找它。最后它会让你确认所提供的信息，你可以输入 yes或进行修改。好了，package.json 文件就准备好了。

Express 服务器设置运行 npm install express@4 --save。这是本项目所需的所有依赖。这里的 save 标志很重要，没有它，package.json 文件不会被更新。package.json 的主要任务是存储依赖列表。它会添加 express 版本 4。你的 package.json 看起来会像这样"dependencies": { "express": "^4.13.4", ... },

下载完成后，你会看到有一个 node_modules 文件夹以及我们依赖的子文件夹。现在在项目根目录下创建一个新的文件server.js。现在我们开始设置 Express 服务器。我会先粘贴所有代码，稍后再解释。

```
var express = require('express');
// 创建我们的应用
var app = express();

app.use(express.static('public'));

app.listen(3000, function () {
  console.log('Express 服务器正在使用端口：3000');
});
```

var express = require('express'); 这将使你能够访问整个 express API。

var app = express(); 会将 express 库作为函数调用。*app.use();* 让你为 express 应用添加功能。*app.use(express.static('public'));* 将指定在我们的 Web 服务器中公开的文件夹名称。*app.listen(port, function(){})* 这里我们的端口将是*3000*，调用的函数将验证我们的 Web 服务器是否正常运行。就是这样，express 服务器已设置完成。

现在进入我们的项目，创建一个新文件夹 public 并创建index.html文件。index.html是你应用的默认文件，Express 服务器会查找此文件。index.html是一个简单的 HTML 文件，内容如下

```
<!DOCTYPE html>
<html>

<head>
```

Chapter 8: React Boilerplate [React + Babel + Webpack]

Section 8.1: react-starter project

About this Project

This is simple boilerplate project. This post will guide you to set up the environment for ReactJs + Webpack + Bable.

Lets get Started

we will need node package manager for fire up express server and manage dependencies throughout the project. if you are new to node package manager, you can check here. Note : Installing node package manager is require here.

Create a folder with suitable name and navigate into it from terminal or by GUI.Then go to terminal and type `npm init` this will create a package.json file, Nothing scary , it will ask you few question like name of your project ,version, description, entry point, git repository, author, license etc. Here entry point is important because node will initially look for it when you run the project. At the end it will ask you to verify the information you provide. You can type *yes* or modify it. Well that's it , our *package.json* file is ready.

Express server setup run *npm install express@4 --save*. This is all the dependencies we needed for this project.Here save flag is important, without it *package.js* file will not be updated. Main task of *package.json* is to store list of dependencies. It will add express version 4. Your *package.json* will look like `"dependencies": { "express": "^4.13.4", ............. },`

After complete download you can see there is *node_modules* folder and sub folder of our dependencies. Now on the root of project create new file *server.js* file. Now we are setting express server. I am going to past all the code and explain it later.

```
var express = require('express');
// Create our app
var app = express();

app.use(express.static('public'));

app.listen(3000, function () {
  console.log('Express server is using port:3000');
});
```

var express = require('express'); this will gave you the access of entire express api.

var app = express(); will call express library as function. *app.use();* let the add the functionality to your express application. *app.use(express.static('public'));* will specify the folder name that will be expose in our web server. *app.listen(port, function(){})* will here our port will be *3000* and function we are calling will verify that out web server is running properly. That's it express server is set up.

Now go to our project and create a new folder public and create *index.html* file. *index.html* is the default file for you application and Express server will look for this file. The *index.html* is simple html file which looks like

```
<!DOCTYPE html>
<html>

<head>
```

```
  <meta charset="UTF-8"/>
</head>

<body>
  <h1>你好，世界</h1>
</body>

</html>
```

然后通过终端进入项目路径，输入*node server.js*。然后你会看到 * console.log('Express 服务器正在使用端口：3000');*。

打开浏览器，在地址栏输入http://localhost:3000，你将看到你好，世界。

现在进入 public 文件夹并创建一个新文件app.jsx。JSX 是一个预处理步骤，它为你的 JavaScript 添加了 XML 语法。你完全可以在没有 JSX 的情况下使用 React，但 JSX 使 React 更加优雅。以下是app.jsx的示例代码

```
ReactDOM.render(
  <h1>你好，世界!!!</h1>,
  document.getElementById('app')
);
```

现在去index.html并修改代码，应该看起来像这样

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23
               /browser.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react.js">
    </script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react-dom.js">    </script>
</head>

<body>
  <div id="app"></div>

  <script type="text/babel" src="app.jsx"></script>
</body>

</html>
```

完成以上步骤后，你就全部完成了，希望你觉得简单。

第8.2节：设置项目

您需要使用Node包管理器来安装项目依赖。请从Nodejs.org下载适合您操作系统的node。Node包管理器随node一起提供。

您也可以使用Node版本管理器来更好地管理您的node和npm版本。它非常适合在不同node版本上测试您的项目。但不建议在生产环境中使用。

在系统上安装node后，继续安装一些必要的包，以使用Babel和Webpack启动您的第一个React项目。

```
  <meta charset="UTF-8"/>
</head>

<body>
  <h1>hello World</h1>
</body>

</html>
```

And go to the project path through the terminal and type *node server.js*. Then you will see * console.log('Express server is using port:3000');*.

Go to the browser and type *http://localhost:3000* in nav bar you will see *hello World*.

Now go inside the public folder and create a new file *app.jsx*. JSX is a preprocessor step that adds XML syntax to your JavaScript.You can definitely use React without JSX but JSX makes React a lot more elegant. Here is the sample code for *app.jsx*

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

Now go to *index.html* and modify the code , it should looks like this

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23
               /browser.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react.js">
    </script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react-dom.js">    </script>
</head>

<body>
  <div id="app"></div>

  <script type="text/babel" src="app.jsx"></script>
</body>

</html>
```

With this in place you are all done, I hope you find it simple.

Section 8.2: Setting up the project

You need Node Package Manager to install the project dependencies. Download node for your operating system from Nodejs.org. Node Package Manager comes with node.

You can also use Node Version Manager to better manage your node and npm versions. It is great for testing your project on different node versions. However, it is not recommended for production environment.

Once you have installed node on your system, go ahead and install some essential packages to blast off your first React project using Babel and Webpack.

在我们真正开始在终端输入命令之前，先了解一下Babel和Webpack的用途。

您可以通过在终端运行npm init来启动项目。按照初始设置进行操作。之后，在终端运行以下命令——

依赖项：

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

开发依赖：

```
npm install babel-core babel-loader babel-preset-es2015 babel-preset-react babel-preset-stage-0
webpack webpack-dev-server react-hot-loader --save-dev
```

可选开发依赖：

```
npm install eslint eslint-plugin-react babel-eslint --save-dev
```

您可以参考此示例 package.json

在项目根目录创建.babelrc，内容如下：

```
{
  "presets": ["es2015", "stage-0", "react"]
}
```

可选地在项目根目录创建.eslintrc，内容如下：

```
{
  "ecmaFeatures": {
    "jsx": true,
    "modules": true
  },
  "env": {
    "browser": true,
    "node": true
  },
  "parser": "babel-eslint",
  "rules": {
    "quotes": [2, "single"],
    "strict": [2, "never"],
  },
  "plugins": [
    "react"
  ]
}
```

创建一个.gitignore文件以防止将生成的文件上传到你的git仓库。

```
node_modules
npm-debug.log
.DS_Store
dist
```

创建一个webpack.config.js文件，内容至少包含以下内容。

```
var path = require('path');
var webpack = require('webpack');
```

Before we actually start hitting commands in the terminal. Take a look at what Babel and Webpack are used for.

You can start your project by running `npm init` in your terminal. Follow the initial setup. After that, run following commands in your terminal-

Dependencies:

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

Dev Dependecies:

```
npm install babel-core babel-loader babel-preset-es2015 babel-preset-react babel-preset-stage-0
webpack webpack-dev-server react-hot-loader --save-dev
```

Optional Dev Dependencies:

```
npm install eslint eslint-plugin-react babel-eslint --save-dev
```

You may refer to this sample package.json

Create `.babelrc` in your project root with following contents:

```
{
  "presets": ["es2015", "stage-0", "react"]
}
```

Optionally create `.eslintrc` in your project root with following contents:

```
{
  "ecmaFeatures": {
    "jsx": true,
    "modules": true
  },
  "env": {
    "browser": true,
    "node": true
  },
  "parser": "babel-eslint",
  "rules": {
    "quotes": [2, "single"],
    "strict": [2, "never"],
  },
  "plugins": [
    "react"
  ]
}
```

Create a `.gitignore` file to prevent uploading generated files to your git repo.

```
node_modules
npm-debug.log
.DS_Store
dist
```

Create `webpack.config.js` file with following minimum contents.

```
var path = require('path');
var webpack = require('webpack');
```

```
module.exports = {
  devtool: 'eval',
  entry: [
    'webpack-dev-server/client?http://localhost:3000',
    'webpack/hot/only-dev-server',
    './src/index'
  ],
output: {
path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
publicPath: '/static/'
  },
plugins: [
    new webpack.HotModuleReplacementPlugin()
  ],
module: {
loaders: [{
test: /\.js$/,
loaders: ['react-hot', 'babel'],
      include: path.join(__dirname, 'src')
    }]
  }
};
```

最后，创建一个sever.js文件，以便能够运行npm start，内容如下：

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('./webpack.config');

new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
historyApiFallback: true
}).listen(3000, 'localhost', function (err, result) {
  if (err) {
    return console.log(err);
  }

console.log('Serving your awesome project at http://localhost:3000/');
});
```

Create src/app.js file to see your React project do something.

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

export default class App extends Component {
  render() {
    return (
      <h1>你好，世界。</h1>
    );
  }
}
```

在终端运行 node server.js 或 npm start ，前提是你已经在 package.json 中定义了 start 的含义

```
module.exports = {
  devtool: 'eval',
  entry: [
    'webpack-dev-server/client?http://localhost:3000',
    'webpack/hot/only-dev-server',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ],
  module: {
    loaders: [{
      test: /\.js$/,
      loaders: ['react-hot', 'babel'],
      include: path.join(__dirname, 'src')
    }]
  }
};
```

And finally, create a `sever.js` file to be able to run `npm start`, with following contents:

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('./webpack.config');

new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
  historyApiFallback: true
}).listen(3000, 'localhost', function (err, result) {
  if (err) {
    return console.log(err);
  }

  console.log('Serving your awesome project at http://localhost:3000/');
});
```

Create `src/app.js` file to see your React project do something.

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

export default class App extends Component {
  render() {
    return (
      <h1>Hello, world.</h1>
    );
  }
}
```

Run `node server.js` or `npm start` in the terminal, if you have defined what `start` stands for in your `package.json`

第9章：ReactJS与jQuery的结合使用

第9.1节：ReactJS与jQuery

首先，你需要导入jquery库。我们还需要导入findDOMNode，因为我们将操作DOM。显然，我们也在导入React。

```
import React from 'react';
import { findDOMNode } from ‘react-dom’;
import $ from ‘jquery’;
```

我们定义了一个箭头函数 ‘handleToggle’ ，当图标被点击时触发。我们只是通过点击图标来显示和隐藏一个引用名为 ‘toggle’ 的div。

```
handleToggle = () => {
    const el = findDOMNode(this.refs.toggle);
    $(el).slideToggle();
};
```

现在让我们设置引用名为 ‘toggle’

```
<ul className="profile-info additional-profile-info-list" ref="toggle">
  <li>
    <span className="info-email">办公邮箱</span>    me@shuvohabib.com
```

我们将在其中触发‘handleToggle’ 的点击事件的 div 元素。

```
 <div className="ellipsis-click" onClick={this.handleToggle}>
   <i className="fa-ellipsis-h"/>
 </div>
```

让我们回顾下面的完整代码，它的样子如何。

```
import React from ‘react’;
import { findDOMNode } from ‘react-dom’;
import $ from ‘jquery’;

export default class FullDesc extends React.Component {
    constructor() {
        super();
    }

   handleToggle = () => {
        const el = findDOMNode(this.refs.toggle);
        $(el).slideToggle();
    };

render() {
        return (
            <div className="long-desc">
                <ul className="profile-info">
                    <li>
                        <span className="info-title">用户名: </span> Shuvo Habib
```

Chapter 9: Using ReactJS with jQuery

Section 9.1: ReactJS with jQuery

Firstly, you have to import jquery library . We also need to import findDOmNode as we're going to manipulate the dom. And obviously we are importing React as well.

```
import React from 'react';
import { findDOMNode } from ‘react-dom’;
import $ from ‘jquery’;
```

We are setting an arrow function ‘handleToggle’ that will fire when an icon will be clicked. We're just showing and hiding a div with a reference naming ‘toggle’ onClick over an icon.

```
handleToggle = () => {
    const el = findDOMNode(this.refs.toggle);
    $(el).slideToggle();
};
```

Let's now set the reference naming ‘toggle’

```
<ul className="profile-info additional-profile-info-list" ref="toggle">
  <li>
    <span className="info-email">Office Email</span>   me@shuvohabib.com
  </li>
</ul>
```

The div element where we will fire the ‘handleToggle’ on onClick.

```
 <div className="ellipsis-click" onClick={this.handleToggle}>
   <i className="fa-ellipsis-h"/>
 </div>
```

Let review the full code below, how it looks like .

```
import React from ‘react’;
import { findDOMNode } from ‘react-dom’;
import $ from ‘jquery’;

export default class FullDesc extends React.Component {
    constructor() {
        super();
    }

    handleToggle = () => {
        const el = findDOMNode(this.refs.toggle);
        $(el).slideToggle();
    };

    render() {
        return (
            <div className="long-desc">
                <ul className="profile-info">
                    <li>
                        <span className="info-title">User Name : </span> Shuvo Habib
                    </li>
                </ul>

                <ul className="profile-info additional-profile-info-list" ref="toggle">
                    <li>
                        <span className="info-email">办公邮箱</span> me@shuvohabib.com
                    </li>
                </ul>

                <div className="ellipsis-click" onClick={this.handleToggle}>
                    <i className="fa-ellipsis-h"/>
                </div>
            </div>
        );
    }
}
```

我们完成了！这就是我们如何在 React 组件中使用jQuery的方法。

```
                    </li>
                </ul>

                <ul className="profile-info additional-profile-info-list" ref="toggle">
                    <li>
                        <span className="info-email">Office Email</span> me@shuvohabib.com
                    </li>
                </ul>

                <div className="ellipsis-click" onClick={this.handleToggle}>
                    <i className="fa-ellipsis-h"/>
                </div>
            </div>
        );
    }
}
```

We are done! This is the way, how we can use **jQuery in React** component.

第10章：React 路由

第10.1节：示例 Routes.js 文件，随后是在组件中使用 Router Link

在你的顶层目录中放置如下文件。它定义了针对不同路径渲染哪些组件

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import New from './containers/new-post';
import Show from './containers/show';

import Index from './containers/home';
import App from './components/app';

export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="posts/new" component={New} />
    <Route path="posts/:id" component={Show} />

  </Route>
);
```

现在在你的顶层 index.js 文件中，也就是应用的入口点，你只需像下面这样渲染这个 Router 组件：

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';
// 导入我们在 routes.js 中创建的路由组件
import routes from './routes';


// 入口点
ReactDOM.render(
    <Router history={browserHistory} routes={routes} />, docume
  nt.getElementById('main'));
```

现在只需在整个应用中使用 Link 替代 &lt;a&gt; 标签。使用 Link 会与 React Router 通信，将 React Router 路由更改为指定的链接，进而渲染 routes.js 中定义的正确组件。

```
import React from 'react';
import { Link } from 'react-router';

export default function PostButton(props) {
  return (
    <Link to={`posts/${props.postId}`}>
      <div className="post-button" >
        {props.title}
        <span>{props.tags}</span>
      </div>
    </Link>
  );
}
```

Chapter 10: React Routing

Section 10.1: Example Routes.js file, followed by use of Router Link in component

Place a file like the following in your top level directory. It defines which components to render for which paths

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import New from './containers/new-post';
import Show from './containers/show';

import Index from './containers/home';
import App from './components/app';

export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="posts/new" component={New} />
    <Route path="posts/:id" component={Show} />

  </Route>
);
```

Now in your top level index.js that is your entry point to the app, you need only render this Router component like so:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';
// import the routes component we created in routes.js
import routes from './routes';


// entry point
ReactDOM.render(
    <Router history={browserHistory} routes={routes} />
  , document.getElementById('main'));
```

Now it is simply a matter of using `Link` instead of `<a>` tags throughout your application. Using Link will communicate with React Router to change the React Router route to the specified link, which will in turn render the correct component as defined in routes.js

```
import React from 'react';
import { Link } from 'react-router';

export default function PostButton(props) {
  return (
    <Link to={`posts/${props.postId}`}>
      <div className="post-button" >
        {props.title}
        <span>{props.tags}</span>
      </div>
    </Link>
  );
}
```

第10.2节：React 路由异步加载

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';

import Index from './containers/home';
import App from './components/app';

// 单组件懒加载使用此方法
const ContactComponent = () => {
  return {
getComponent: (location, callback)=> {
        require.ensure([], require => {
          callback(null, require('./components/Contact')["default"]);
        }, 'Contact');
      }
    }
};

//用于多个组件
 const groupedComponents = (pageName) => {
  return {
getComponent: (location, callback)=> {
        require.ensure([], require => {
          switch(pageName){
            case 'about' :
callback(null, require( "./components/about" )["default"]);
                    break ;
            case 'tos' :
callback(null, require( "./components/tos" )["default"]);
                    break ;
          }
        }, "groupedComponents");
      }
    }
};
export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="/contact" {...ContactComponent()} />
    <Route path="/about" {...groupedComponents('about')} />
    <Route path="/tos" {...groupedComponents('tos')} />
  </Route>
);
```

Section 10.2: React Routing Async

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';

import Index from './containers/home';
import App from './components/app';

//for single Component lazy load use this
const ContactComponent = () => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          callback(null, require('./components/Contact')["default"]);
        }, 'Contact');
      }
    }
};

//for multiple componnets
 const groupedComponents = (pageName) => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          switch(pageName){
            case 'about' :
                callback(null, require( "./components/about" )["default"]);
                    break ;
            case 'tos' :
                callback(null, require( "./components/tos" )["default"]);
                    break ;
          }
        }, "groupedComponents");
      }
    }
};
export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="/contact" {...ContactComponent()} />
    <Route path="/about" {...groupedComponents('about')} />
    <Route path="/tos" {...groupedComponents('tos')} />
  </Route>
);
```

第11章：组件之间的通信

第11.1节：无状态函数组件之间的通信

在本例中，我们将使用Redux和React Redux模块来处理应用状态，并自动重新渲染我们的函数组件，当然还包括React和React Dom

你可以在这里查看完整演示

下面的示例中，我们有三个不同的组件和一个连接组件

- UserInputForm：该组件显示一个输入框，当输入框的值发生变化时，它会调用inputChange方法，该方法在props中（由父组件提供），如果也提供了数据，则会在输入框中显示该数据。
- UserDashboard：该组件显示一条简单消息，并且嵌套了UserInputForm组件，它还将inputChange方法传递给UserInputForm组件，UserInputForm组件反过来使用该方法与父组件通信。
  - UserDashboardConnected：该组件仅使用ReactRedux connect方法包装了UserDashboard组件。这使我们更容易管理组件状态并在状态变化时更新组件。
- App：该组件仅渲染UserDashboardConnected组件。

```
const UserInputForm = (props) => {

  let handleSubmit = (e) => {
e.preventDefault();
  }

  return(
    <form action="" onSubmit={handleSubmit}>
      <label htmlFor="name">请输入您的姓名</label>
      <br />
      <input type="text" id="name" defaultValue={props.data.name || ''} onChange={
props.inputChange } />
    </form>
  )

}


const UserDashboard = (props) => {

  let inputChangeHandler = (event) => {
    props.updateName(event.target.value);
  }

  return(
    <div>
      <h1>嗨 { props.user.name || '用户' }</h1>
      <UserInputForm data={props.user} inputChange={inputChangeHandler} />
    </div>
```

Chapter 11: Communicate Between Components

Section 11.1: Communication between Stateless Functional Components

In this example we will make use of `Redux` and `React Redux` modules to handle our application state and for auto re-render of our functional components., And ofcourse `React` and `React Dom`

You can checkout the completed demo here

In the example below we have three different components and one connected component

- **UserInputForm**: This component display an input field And when the field value changes, it calls `inputChange` method on `props` (which is provided by the parent component) and if the data is provided as well, it displays that in the input field.
- **UserDashboard**: This component displays a simple message and also nests `UserInputForm` component, It also passes `inputChange` method to `UserInputForm` component, `UserInputForm` component inturn makes use of this method to communicate with the parent component.
  - **UserDashboardConnected**: This component just wraps the `UserDashboard` component using `ReactRedux connect` method., This makes it easier for us to manage the component state and update the component when the state changes.
- **App**: This component just renders the `UserDashboardConnected` component.

```
const UserInputForm = (props) => {

  let handleSubmit = (e) => {
    e.preventDefault();
  }

  return(
    <form action="" onSubmit={handleSubmit}>
      <label htmlFor="name">Please enter your name</label>
      <br />
      <input type="text" id="name" defaultValue={props.data.name || ''} onChange={
props.inputChange } />
    </form>
  )

}


const UserDashboard = (props) => {

  let inputChangeHandler = (event) => {
    props.updateName(event.target.value);
  }

  return(
    <div>
      <h1>Hi { props.user.name || 'User' }</h1>
      <UserInputForm data={props.user} inputChange={inputChangeHandler} />
    </div>
  )

}

const mapStateToProps = (state) => {
  return {
user: state
  };
}
const mapDispatchToProps = (dispatch) => {
  return {
updateName: (data) => dispatch( Action.updateName(data) ),
  };
};

const { connect, Provider } = ReactRedux;
const UserDashboardConnected = connect(
  mapStateToProps,
mapDispatchToProps
)(UserDashboard);



const App = (props) => {
  return(
    <div>
      <h1>无状态函数组件之间的通信</h1>
      <UserDashboardConnected />
    </div>
  )
}



const user = (state={name: '约翰'}, action) => {
  switch (action.type) {
    case 'UPDATE_NAME':
      return Object.assign( {}, state, {name: action.payload}  );

    default:
      return state;
  }
};

const { createStore } = Redux;
const store = createStore(user);
const Action = {
updateName: (data) => {
    return { type : 'UPDATE_NAME', payload: data }
  },
}


ReactDOM.render(
  <Provider store={ store }>
    <App />
  </Provider>,
document.getElementById('application')
);
```

[JS Bin URL](#)

```
  )

}

const mapStateToProps = (state) => {
  return {
    user: state
  };
}
const mapDispatchToProps = (dispatch) => {
  return {
    updateName: (data) => dispatch( Action.updateName(data) ),
  };
};

const { connect, Provider } = ReactRedux;
const UserDashboardConnected = connect(
  mapStateToProps,
  mapDispatchToProps
)(UserDashboard);



const App = (props) => {
  return(
    <div>
      <h1>Communication between Stateless Functional Components</h1>
      <UserDashboardConnected />
    </div>
  )
}



const user = (state={name: 'John'}, action) => {
  switch (action.type) {
    case 'UPDATE_NAME':
      return Object.assign( {}, state, {name: action.payload}  );

    default:
      return state;
  }
};

const { createStore } = Redux;
const store = createStore(user);
const Action = {
  updateName: (data) => {
    return { type : 'UPDATE_NAME', payload: data }
  },
}


ReactDOM.render(
  <Provider store={ store }>
    <App />
  </Provider>,
  document.getElementById('application')
);
```

[JS Bin URL](#)

第12章：如何搭建基础的webpack、react和babel环境

第12.1节：如何构建带有图片的定制“Hello world”流水线

步骤1：安装Node.js

你将要构建的构建流水线基于Node.js，因此首先必须确保你已经安装了Node.js。关于如何安装Node.js的说明，你可以查看这里的SO文档

步骤2：将你的项目初始化为Node模块

在命令行中打开你的项目文件夹，使用以下命令：

```
npm init
```

在本示例中，你可以选择默认设置，或者如果你想了解这些设置的含义，可以查看这篇关于设置包配置的SO文档。

步骤3：安装必要的npm包

在命令行运行以下命令以安装此示例所需的软件包：

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

然后对于开发依赖运行此命令：

```
npm install --save-dev babel-core babel-preset-react babel-preset-es2015 webpack babel-loader css-
loader style-loader file-loader image-webpack-loader
```

最后，webpack 和 webpack-dev-server 更适合全局安装，而不是作为项目的依赖项，如果你更愿意将其作为依赖项添加，那也可以，我个人不这样做。以下是运行的命令：

```
npm install --global webpack webpack-dev-server
```

步骤3：在项目根目录添加 .babelrc 文件

这将设置 babel 使用你刚安装的预设。你的 .babelrc 文件应如下所示：

```
{
  "presets": ["react", "es2015"]
}
```

步骤4：设置项目目录结构

在你的根目录下建立如下的目录结构：

```
|- node_modules
|- src/
  |- 组件/
  |- 图片/
  |- 样式/
  |- index.html
```

Chapter 12: How to setup a basic webpack, react and babel environment

Section 12.1: How to build a pipeline for a customized "Hello world" with images

Step 1: Install Node.js

The build pipeline you will be building is based in Node.js so you must ensure in the first instance that you have this installed. For instructions on how to install Node.js you can checkout the SO docs for that here

Step 2: Initialise your project as an node module

Open your project folder on the command line and use the following command:

```
npm init
```

For the purposes of this example you can feel free to take the defaults or if you'd like more info on what all this means you can check out this SO doc on setting up package configuration.

Step 3: Install necessary npm packages

Run the following command on the command line to install the packages necessary for this example:

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

Then for the dev dependencies run this command:

```
npm install --save-dev babel-core babel-preset-react babel-preset-es2015 webpack babel-loader css-
loader style-loader file-loader image-webpack-loader
```

Finally webpack and webpack-dev-server are things that are worth installing globally rather than as a dependency of your project, if you'd prefer to add it as a dependency then that will work to, I don't. Here is the command to run:

```
npm install --global webpack webpack-dev-server
```

Step 3: Add a .babelrc file to the root of your project

This will setup babel to use the presets you've just installed. Your .babelrc file should look like this:

```
{
  "presets": ["react", "es2015"]
}
```

Step 4: Setup project directory structure

Set yourself up a directory stucture that looks like the below in the root of your directory:

```
|- node_modules
|- src/
  |- components/
  |- images/
  |- styles/
  |- index.html
  |- index.jsx
|- .babelrc
|- package.json
```

注意：node_modules、.babelrc 和 package.json 应该都已经存在于之前的步骤中，我只是包含它们以便你能看到它们的位置。

步骤5：用Hello World项目文件填充项目

这对构建流水线的过程其实不太重要，所以我直接给你这些代码，你可以复制粘贴：

src/components/HelloWorldComponent.jsx

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

class HelloWorldComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {name: '学生'};
    this.handleChange = this.handleChange.bind(this);
  }

handleChange(e) {
    this.setState({name: e.target.value});
  }

render() {
    return (
      <div>
        <div className="image-container">
          <img src="./images/myImage.gif" />
        </div>
        <div className="form">
          <input type="text" onChange={this.handleChange} />
          <div>
我的名字是 {this.state.name} ，我是个聪明人，因为我搭建了一个 React 构建
管道
</div>
       </div>
     </div>
    );
  }
}

export default HelloWorldComponent;
```

src/images/myImage.gif

随意替换成你喜欢的任何图片，这只是为了证明我们也可以打包图片。如果你提供自己的图片并且命名不同，那么你需要更新HelloWorldComponent。jsx 以反映您的更改。同样，如果您选择了具有不同文件扩展名的图像，则需要修改 webpack.config.js 中图像加载器的 test 属性，使用适当的正则表达式以匹配您的新文件扩展名。

src/styles/styles.css

```
.form {
margin: 25px;
```

```
  |- index.jsx
|- .babelrc
|- package.json
```

NOTE: The `node_modules`, `.babelrc` and `package.json` should all have already been there from previous steps I just included them so you can see where they fit.

Step 5: Populate the project with the Hello World project files

This isn't really important to the process of building a pipeline so I'll just give you the code for these and you can copy paste them in:

src/components/HelloWorldComponent.jsx

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

class HelloWorldComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {name: 'Student'};
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.setState({name: e.target.value});
  }

  render() {
    return (
      <div>
        <div className="image-container">
          <img src="./images/myImage.gif" />
        </div>
        <div className="form">
          <input type="text" onChange={this.handleChange} />
          <div>
            My name is {this.state.name} and I'm a clever cloggs because I built a React build
pipeline
          </div>
        </div>
      </div>
    );
  }
}

export default HelloWorldComponent;
```

src/images/myImage.gif

Feel free to substitute this with any image you'd like it's simply there to prove the point that we can bundle up images as well. If you provide your own image and you name it something different then you'll have to update the `HelloWorldComponent.jsx` to reflect your changes. Equally if you choose an image with a different file extension then you need to modify the `test` property of the image loader in the webpack.config.js with appropriate regex to match your new file extension..

src/styles/styles.css

```
.form {
  margin: 25px;
padding: 25px;
  border: 1px solid #ddd;
  background-color: #eaeaea;
  border-radius: 10px;
}

.form div {
  padding-top: 25px;
}

.image-container {
  display: flex;
  justify-content: center;
}
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>学习构建 React 流水线</title>
</head>
<body>
  <div id="content"></div>
  <script src="app.js"></script>
</body>
</html>
```

index.jsx

```
import React from 'react';
import { render } from 'react-dom';
import HelloWorldComponent from './components/HelloWorldComponent.jsx';

require('./images/myImage.gif');
require('./styles/styles.css');
require('./index.html');

render(<HelloWorldComponent />, document.getElementById('content'));
```

第6步：创建webpack配置

在项目根目录下创建一个名为webpack.config.js的文件，并将以下代码复制到其中：

webpack.config.js

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

var config = {
context: path.resolve(__dirname + '/src'),
  entry: './index.jsx',
output: {
filename: 'app.js',
    path: path.resolve(__dirname + '/dist'),
  },
devServer: {
contentBase: path.join(__dirname + '/dist'),
    port: 3000,
open: true,
```

```
  padding: 25px;
  border: 1px solid #ddd;
  background-color: #eaeaea;
  border-radius: 10px;
}

.form div {
  padding-top: 25px;
}

.image-container {
  display: flex;
  justify-content: center;
}
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Learning to build a react pipeline</title>
</head>
<body>
  <div id="content"></div>
  <script src="app.js"></script>
</body>
</html>
```

index.jsx

```
import React from 'react';
import { render } from 'react-dom';
import HelloWorldComponent from './components/HelloWorldComponent.jsx';

require('./images/myImage.gif');
require('./styles/styles.css');
require('./index.html');

render(<HelloWorldComponent />, document.getElementById('content'));
```

Step 6: Create webpack configuration

Create a file called webpack.config.js in the root of your project and copy this code into it:

webpack.config.js

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

var config = {
  context: path.resolve(__dirname + '/src'),
  entry: './index.jsx',
  output: {
    filename: 'app.js',
    path: path.resolve(__dirname + '/dist'),
  },
  devServer: {
    contentBase: path.join(__dirname + '/dist'),
    port: 3000,
    open: true,
  },
module: {
    loaders: [
      {
test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      },
      {
test: /\.css$/,
        loader: "style!css"
      },
      {
test: /\.gif$/,
        loaders: [
          'file?name=[path][name].[ext]',
          'image-webpack',
        ]
      },
      { test: /|.(html)$/,
loader: "file?name=[path][name].[ext]"
      }
    ],
  },
};

module.exports = config;
```

步骤7：为您的流水线创建npm任务

为此，您需要在项目根目录下package.json文件中定义的JSON的scripts键中添加两个属性。使您的scripts键如下所示：

```
  "scripts": {
    "start": "webpack-dev-server",
    "build": "webpack",
    "test": "echo \"错误：未指定测试\" && exit 1"
  },
```

测试脚本已经存在，您可以选择是否保留它，这对本例来说并不重要。

步骤8：使用流水线

从命令行，如果你在项目根目录，现在应该能够运行以下命令：

```
npm run build
```

这将打包你构建的小应用程序，并将其放置在项目文件夹根目录下创建的 dist/目录中。

如果你运行命令：

```
npm start
```

那么你构建的应用程序将通过webpack开发服务器实例在默认的网页浏览器中提供服务。

```
  },
  module: {
    loaders: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      },
      {
        test: /\.css$/,
        loader: "style!css"
      },
      {
        test: /\.gif$/,
        loaders: [
          'file?name=[path][name].[ext]',
          'image-webpack',
        ]
      },
      { test: /\.(html)$/,
        loader: "file?name=[path][name].[ext]"
      }
    ],
  },
};

module.exports = config;
```

Step 7: Create npm tasks for your pipeline

To do this you will need to add two properties to the scripts key of the JSON defined in the package.json file in the root of your project. Make your scripts key look like this:

```
  "scripts": {
    "start": "webpack-dev-server",
    "build": "webpack",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

The `test` script will have already been there and you can choose whether to keep it or not, it's not important to this example.

Step 8: Use the pipeline

From the command line, if you are in the project root directory you should now be able to run the command:

```
npm run build
```

This will bundle up the little application you've built and place it in the `dist/` directory that it will create in the root of your project folder.

If you run the command:

```
npm start
```

Then the application you've built will be served up in your default web browser inside of a webpack dev server instance.

第13章：React.createClass 与 extends React.Component

第13.1节：创建React组件

让我们通过比较两个代码示例来探讨语法差异。

React.createClass（已弃用）

这里我们有一个`const`，赋值为一个React类，后面跟着`render`函数，完成一个典型的基础组件定义。

```
import React from 'react';

const MyComponent = React.createClass({
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

让我们将上面的React.createClass定义转换为使用ES6类。

```
import React from 'react';

class MyComponent extends React.Component {
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

在这个例子中，我们现在使用ES6类。对于React的变化，我们现在创建一个名为MyComponent的类，并继承自React.Component，而不是直接访问React.createClass。这样，我们使用更少的React模板代码，更多的JavaScript。

附注：通常这会与Babel等工具一起使用，将ES6编译为ES5，以便在其他浏览器中运行。

第13.2节：“this”上下文

使用 React.createClass 会自动正确绑定**this**上下文（值），但使用

ES6 类时情况并非如此。

React.createClass

注意 onClick 声明中绑定了 **this**.`handleClick` 方法。当该方法被调用时，React 会为 `handleClick` 应用正确的执行上下文。

Chapter 13: React.createClass vs extends React.Component

Section 13.1: Create React Component

Let's explore the syntax differences by comparing two code examples.

React.createClass (deprecated)

Here we have a `const` with a React class assigned, with the `render` function following on to complete a typical base component definition.

```
import React from 'react';

const MyComponent = React.createClass({
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Let's take the above React.createClass definition and convert it to use an ES6 class.

```
import React from 'react';

class MyComponent extends React.Component {
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

In this example we're now using ES6 classes. For the React changes, we now create a class called **MyComponent** and extend from React.Component instead of accessing React.createClass directly. This way, we use less React boilerplate and more JavaScript.

PS: Typically this would be used with something like Babel to compile the ES6 to ES5 to work in other browsers.

Section 13.2: "this" Context

Using React.createClass will automatically bind **this** context (values) correctly, but that is not the case when using ES6 classes.

React.createClass

Note the `onClick` declaration with the **this**.`handleClick` method bound. When this method gets called React will apply the right execution context to the `handleClick`.

```
import React from 'react';

const MyComponent = React.createClass({
  handleClick() {
console.log(this); // React 组件实例
  },
render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
});

export default MyComponent;
```

React.Component

使用 ES6 类时，**this** 默认是 **null**，类的属性不会自动绑定到 React 类（组件）实例。

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
handleClick() {
    console.log(this); // null
  }
render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

我们有几种方法可以绑定正确的this上下文。

案例1：内联绑定：

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
handleClick() {
console.log(this); // React组件实例
  }
render() {
    return (
      <div onClick={this.handleClick.bind(this)}></div>
    );
  }
}

export default MyComponent;
```

```
import React from 'react';

const MyComponent = React.createClass({
  handleClick() {
    console.log(this); // the React Component instance
  },
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
});

export default MyComponent;
```

React.Component

With ES6 classes **this** is **null** by default, properties of the class do not automatically bind to the React class (component) instance.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // null
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

There are a few ways we could bind the right **this** context.

Case 1: Bind inline:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick.bind(this)}></div>
    );
  }
}

export default MyComponent;
```

案例2：在类构造函数中绑定

另一种方法是在constructor内部更改this.handleClick的上下文。这样可以避免内联重复。许多人认为这是更好的方法，完全避免修改JSX：

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
handleClick() {
console.log(this); // React组件实例
  }
render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

案例3：使用ES6匿名函数

你也可以使用ES6匿名函数，而无需显式绑定：

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
handleClick = () => {
console.log(this); // React组件实例
  }
render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

第13.3节：声明默认属性和属性类型

我们使用和声明默认属性及其类型的方式发生了重要变化。

React.createClass

在此版本中，`propTypes` 属性是一个对象，我们可以在其中声明每个属性的类型。
`getDefaultProps` 属性是一个返回对象的函数，用于创建初始属性。

```
import React from 'react';

const MyComponent = React.createClass({
```

Case 2: Bind in the class constructor

Another approach is changing the context of `this.handleClick` inside the `constructor`. This way we avoid inline repetition. Considered by many as a better approach that avoids touching JSX at all:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Case 3: Use ES6 anonymous function

You can also use ES6 anonymous function without having to bind explicitly:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick = () => {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Section 13.3: Declare Default Props and PropTypes

There are important changes in how we use and declare default props and their types.

React.createClass

In this version, the `propTypes` property is an Object in which we can declare the type for each prop. The `getDefaultProps` property is a function that returns an Object to create the initial props.

```
import React from 'react';

const MyComponent = React.createClass({
```

```
propTypes: {
name: React.PropTypes.string,
    position: React.PropTypes.number
  },
getDefaultProps() {
    return {
name: 'Home',
      position: 1
    };
  },
render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

此版本将`propTypes`用作实际**MyComponent**类的一个属性，而不是作为`createClass`定义对象的一部分属性。

现在，getDefaultProps 已经变成了类上的一个名为 defaultProps 的普通对象属性，因为它不再是一个“get”函数，而只是一个对象。这样可以避免更多的 React 样板代码，这只是普通的 JavaScript。

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
render() {
    return (
      <div></div>
    );
  }
}
MyComponent.propTypes = {
  name: React.PropTypes.string,
  position: React.PropTypes.number
};
MyComponent.defaultProps = {
  name: 'Home',
position: 1
};

export default MyComponent;
```

此外，propTypes 和 defaultProps 还有另一种语法。如果你的构建环境开启了 ES7 属性初始化器，这是一种快捷方式：

```
import React from 'react';

class MyComponent extends React.Component {
  static propTypes = {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  };
```

```
  propTypes: {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  },
  getDefaultProps() {
    return {
      name: 'Home',
      position: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

This version uses `propTypes` as a property on the actual **MyComponent** class instead of a property as part of the `createClass` definition Object.

The `getDefaultProps` has now changed to just an Object property on the class called defaultProps, as it's no longer a "get" function, it's just an Object. It avoids more React boilerplate, this is just plain JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
MyComponent.propTypes = {
  name: React.PropTypes.string,
  position: React.PropTypes.number
};
MyComponent.defaultProps = {
  name: 'Home',
  position: 1
};

export default MyComponent;
```

Additionally, there is another syntax for `propTypes` and `defaultProps`. This is a shortcut if your build has ES7 property initializers turned on:

```
import React from 'react';

class MyComponent extends React.Component {
  static propTypes = {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  };
  static defaultProps = {
    name: 'Home',
position: 1
  };
constructor(props) {
    super(props);
  }
render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

第13.4节：混入（Mixins）

我们只能在 React.createClass 方式中使用mixins。

React.createClass

在此版本中，我们可以通过 mixins 属性向组件添加mixins，该属性接受一个可用 mixins 的数组。这些 mixins 会扩展组件类。

```
import React from 'react';

var MyMixin = {
  doSomething() {

  }
};
const MyComponent = React.createClass({
  mixins: [MyMixin],
handleClick() {
    this.doSomething(); // 调用 mixin 的方法
  },
render() {
    return (
      <button onClick={this.handleClick}>执行操作</button>
    );
  }
});

export default MyComponent;
```

React.Component

使用 ES6 编写的 React 组件时不支持 React mixins。此外，React 中的 ES6 类也不支持它们。原因是它们被认为是有害的。

第13.5节：设置初始状态

我们设置初始状态的方式有所变化。

React.createClass

我们有一个getInitialState函数，它简单地返回一个初始状态的对象。

```
  static defaultProps = {
    name: 'Home',
    position: 1
  };
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

Section 13.4: Mixins

We can use `mixins` only with the React.createClass way.

React.createClass

In this version we can add `mixins` to components using the mixins property which takes an Array of available mixins. These then extend the component class.

```
import React from 'react';

var MyMixin = {
  doSomething() {

  }
};
const MyComponent = React.createClass({
  mixins: [MyMixin],
  handleClick() {
    this.doSomething(); // invoke mixin's method
  },
  render() {
    return (
      <button onClick={this.handleClick}>Do Something</button>
    );
  }
});

export default MyComponent;
```

React.Component

React mixins are not supported when using React components written in ES6. Moreover, they will not have support for ES6 classes in React. The reason is that they are considered harmful.

Section 13.5: Set Initial State

There are changes in how we are setting the initial states.

React.createClass

We have a `getInitialState` function, which simply returns an Object of initial states.

```
import React from 'react';

const MyComponent = React.createClass({
  getInitialState () {
    return {
activePage: 1
    };
  },
render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

在这个版本中，我们将所有状态声明为构造函数中的简单初始化属性，而不是使用getInitialState函数。感觉不那么“React API”驱动，因为这只是普通的JavaScript。

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activePage: 1
    };
  }
render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

第13.6节：使用ES6/React的“this”关键字通过ajax从服务器获取数据

```
import React from 'react';

class SearchEs6 extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
searchResults: []
        };
    }

showResults(response){
        this.setState({
searchResults: response.results
        })
    }
```

```
import React from 'react';

const MyComponent = React.createClass({
  getInitialState () {
    return {
      activePage: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

In this version we declare all state as a simple **initialisation property in the constructor**, instead of using the `getInitialState` function. It feels less "React API" driven since this is just plain JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activePage: 1
    };
  }
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

Section 13.6: ES6/React “this” keyword with ajax to get data from server

```
import React from 'react';

class SearchEs6 extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
            searchResults: []
        };
    }

    showResults(response){
        this.setState({
            searchResults: response.results
        })
    }
search(url){
        $.ajax({
            type: "GET",
            dataType: 'jsonp',
            url: url,
success: (data) => {
                this.showResults(data);
            },
error: (xhr, status, err) => {
                console.error(url, status, err.toString());
            }
        });
    }

render() {
        return (
            <div>
                <SearchBox search={this.search.bind(this)} />
                <Results searchResults={this.state.searchResults} />
            </div>
        );
    }
}
```

```
    search(url){
        $.ajax({
            type: "GET",
            dataType: 'jsonp',
            url: url,
            success: (data) => {
                this.showResults(data);
            },
            error: (xhr, status, err) => {
                console.error(url, status, err.toString());
            }
        });
    }

    render() {
        return (
            <div>
                <SearchBox search={this.search.bind(this)} />
                <Results searchResults={this.state.searchResults} />
            </div>
        );
    }
}
```

第14章：React AJAX调用

第14.1节：HTTP GET请求

有时组件需要渲染来自远程端点（例如REST API）的一些数据。一个标准做法是在componentDidMount方法中进行此类调用。

下面是一个示例，使用 superagent作为AJAX辅助工具：

```
import React from 'react'
import request from 'superagent'

class App extends React.Component {
  constructor () {
    super()
    this.state = {}
  }
componentDidMount () {
   request
.get('/search')
      .query({ query: 'Manny' })
      .query({ range: '1..5' })
      .query({ order: 'desc' })
      .set('API-Key', 'foobar')
      .set('Accept', 'application/json')
      .end((err, resp) => {
        if (!err) {
          this.setState({someData: resp.text})
        }
      })
  },
render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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

可以通过调用request对象上的相应方法来发起请求，然后调用.end()来发送请求。设置请求头字段很简单，只需调用.set()并传入字段名和值。

query() 方法接受对象，当与 GET 方法一起使用时，将形成查询字符串。以下内容将生成路径 /search?query=Manny&range=1..5&order=desc。

POST请求

```
request.post('/user')
.set('Content-Type', 'application/json')
  .send('{"name":"tj","pet":"tobi"}')
.end(callback)
```

详情请参见Superagent 文档。

Chapter 14: React AJAX call

Section 14.1: HTTP GET request

Sometimes a component needs to render some data from a remote endpoint (e.g. a REST API). A standard practice is to make such calls in componentDidMount method.

Here is an example, using superagent as AJAX helper:

```
import React from 'react'
import request from 'superagent'

class App extends React.Component {
  constructor () {
    super()
    this.state = {}
  }
  componentDidMount () {
    request
      .get('/search')
      .query({ query: 'Manny' })
      .query({ range: '1..5' })
      .query({ order: 'desc' })
      .set('API-Key', 'foobar')
      .set('Accept', 'application/json')
      .end((err, resp) => {
        if (!err) {
          this.setState({someData: resp.text})
        }
      })
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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

A request can be initiated by invoking the appropriate method on the request object, then calling .end() to send the request. Setting header fields is simple, invoke .set() with a field name and value.

The .query() method accepts objects, which when used with the GET method will form a query-string. The following will produce the path /search?query=Manny&range=1..5&order=desc.

POST requests

```
request.post('/user')
  .set('Content-Type', 'application/json')
  .send('{"name":"tj","pet":"tobi"}')
  .end(callback)
```

See Superagent docs for more details.

第14.2节：HTTP GET请求及数据循环

下面的示例展示了如何将从远程来源获取的一组数据渲染到组件中。

我们使用内置于大多数浏览器中的fetch发起AJAX请求。生产环境中请使用fetch的polyfill以支持旧版浏览器。你也可以使用任何其他请求库（例如axios、SuperAgent，甚至纯JavaScript）。

我们将接收到的数据设置为组件状态，以便在render方法中访问。在那里，我们使用map循环遍历数据。别忘了始终为循环元素添加唯一的key属性（或prop），这对React的渲染性能非常重要。

```
import React from 'react';

class Users extends React.Component {
  constructor() {
    super();
    this.state = { users: [] };
  }

componentDidMount() {
fetch('/api/users')
.then(response => response.json())
      .then(json => this.setState({ users: json.data }));
  }

render() {
    return (
      <div>
        <h1>用户</h1>
        {
          this.state.users.length == 0
            ? '正在加载用户...'
            : this.state.users.map(user => (
              <figure key={user.id}>
                <img src={user.avatar} />
                <figcaption>
                  {user.name}
                </figcaption>
              </figure>
            ))
        }
      </div>
    );
  }
}

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

JSBin上的工作示例。

第14.3节：在React中使用Ajax而不依赖第三方库——即使用原生JavaScript

以下代码适用于IE9及以上版本

```
import React from 'react'
```

Section 14.2: HTTP GET request and looping through data

The following example shows how a set of data obtained from a remote source can be rendered into a component.

We make an AJAX request using fetch, which is build into most browsers. Use a fetch polyfill in production to support older browsers. You can also use any other library for making requests (e.g. axios, SuperAgent, or even plain Javascript).

We set the data we receive as component state, so we can access it inside the render method. There, we loop through the data using map. Don't forget to always add a unique key attribute (or prop) to the looped element, which is important for React's rendering performance.

```
import React from 'react';

class Users extends React.Component {
  constructor() {
    super();
    this.state = { users: [] };
  }

  componentDidMount() {
    fetch('/api/users')
      .then(response => response.json())
      .then(json => this.setState({ users: json.data }));
  }

  render() {
    return (
      <div>
        <h1>Users</h1>
        {
          this.state.users.length == 0
            ? 'Loading users...'
            : this.state.users.map(user => (
              <figure key={user.id}>
                <img src={user.avatar} />
                <figcaption>
                  {user.name}
                </figcaption>
              </figure>
            ))
        }
      </div>
    );
  }
}

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

Working example on JSBin.

Section 14.3: Ajax in React without a third party library - a.k.a with VanillaJS

The following would work in IE9+

```
import React from 'react'
```

```
class App extends React.Component {
  constructor () {
    super()
    this.state = {someData: null}
  }
componentDidMount () {
    var request = new XMLHttpRequest();
    request.open('GET', '/my/url', true);

    request.onload = () => {
      if (request.status >= 200 && request.status < 400) {
        // 成功！
        this.setState({someData: request.responseText})
      } else {
        // 我们已连接到目标服务器，但服务器返回了错误
        // 可能通过更改状态来处理该错误。
      }
    };

request.onerror = () => {
      // 发生了某种连接错误。
      // 可能通过更改状态来处理该错误。
    };

request.send();
  },
render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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

```
class App extends React.Component {
  constructor () {
    super()
    this.state = {someData: null}
  }
  componentDidMount () {
    var request = new XMLHttpRequest();
    request.open('GET', '/my/url', true);

    request.onload = () => {
      if (request.status >= 200 && request.status < 400) {
        // Success!
        this.setState({someData: request.responseText})
      } else {
        // We reached our target server, but it returned an error
        // Possibly handle the error by changing your state.
      }
    };

    request.onerror = () => {
      // There was a connection error of some sort.
      // Possibly handle the error by changing your state.
    };

    request.send();
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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

第15章：组件之间的通信

第15.1节：子组件到父组件

将数据发送回父组件，为此我们只需**将一个函数作为属性从父组件传递给子组件**，且**子组件调用该函数**。

在此示例中，我们将通过将一个函数传递给子组件并在子组件内部调用该函数来更改父组件的状态。

```
import React from 'react';

class Parent extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count: 0 };

        this.outputEvent = this.outputEvent.bind(this);
    }
outputEvent(event) {
        // 事件上下文来自子组件
        this.setState({ count: this.state.count++ });
    }

render() {
        const variable = 5;
        return (
            <div>
Count: { this.state.count }
                <Child clickHandler={this.outputEvent} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return (
            <button onClick={this.props.clickHandler}>
             Add One More
            </button>
        );
    }
}

export default Parent;
```

注意，父组件的outputEvent方法（用于更改父组件状态）是由子组件按钮的onClick事件调用的。

第15.2节：无关组件

如果你的组件之间没有父子关系（或者虽然相关但关系过于远，比如曾孙辈），唯一的办法是有某种信号，一个组件订阅该信号，另一个组件写入该信号。

Chapter 15: Communication Between Components

Section 15.1: Child to Parent Components

Sending data back to the parent, to do this we simply **pass a function as a prop from the parent component to the child component**, and **the child component calls that function**.

In this example, we will change the Parent state by passing a function to the Child component and invoking that function inside the Child component.

```
import React from 'react';

class Parent extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count: 0 };

        this.outputEvent = this.outputEvent.bind(this);
    }
    outputEvent(event) {
        // the event context comes from the Child
        this.setState({ count: this.state.count++ });
    }

    render() {
        const variable = 5;
        return (
            <div>
                Count: { this.state.count }
                <Child clickHandler={this.outputEvent} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return (
            <button onClick={this.props.clickHandler}>
                Add One More
            </button>
        );
    }
}

export default Parent;
```

Note that the Parent's `outputEvent` method (that changes the Parent state) is invoked by the Child's button `onClick` event.

Section 15.2: Not-related Components

The only way if your components does not have a parent-child relationship (or are related but too further such as a grand grand grand son) is to have some kind of a signal that one component subscribes to, and the other writes into.

任何事件系统的两个基本操作是：**订阅/监听**一个事件以接收通知，以及**发送/触发/发布/分发**一个事件以通知想要接收的对象。

至少有三种模式可以实现这一点。你可以在这里找到一个比较。

以下是简要总结：

- 模式1：事件发射器/目标/分发器：监听者需要引用源对象来订阅。
  - 订阅：`otherObject.addEventListener('click', () => { alert('click!'); });`
  - 分发：`this.dispatchEvent('click');`
- 模式2：发布/订阅：你不需要特定引用触发事件的源，有一个全局对象在任何地方都可访问，负责处理所有事件。
  - 订阅：`globalBroadcaster.subscribe('click', () => { alert('click!'); });`
  - 分发：`globalBroadcaster.publish('click');`
- 模式3：信号：类似于事件发射器/目标/分发器，但这里不使用任意字符串。每个可能发出事件的对象都需要有一个特定名称的属性。这样，你就能准确知道对象可以发出哪些事件。
  - 订阅：`otherObject.clicked.add( () => { alert('click'); });`
  - 分发：`this.clicked.dispatch();`

第15.3节：父组件到子组件

这实际上是最简单的情况，在React世界中非常自然，而且你很可能已经在使用它了。

你可以**将props传递给子组件**。在这个例子中，`message`是我们传递给子组件的prop，message这个名字是任意选择的，你可以随意命名。

```
import React from 'react';

class Parent extends React.Component {
    render() {
        const 变量 = 5;
        return (
            <div>
                <Child message="message for child" />
                <Child message={变量} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return <h1>{this.props.message}</h1>
    }
}

export default Parent;
```

这里，<Parent />组件渲染了两个<Child />组件，分别在第一个组件中传递了message 给child，第二个组件中传递了5。

Those are the 2 basic operations of any event system: **subscribe/listen** to an event to be notify, and **send/trigger/publish/dispatch** a event to notify the ones who wants.

There are at least 3 patterns to do that. You can find a comparison here.

Here is a brief summary:

- Pattern 1: **Event Emitter/Target/Dispatcher**: the listeners need to reference the source to subscribe.
  - to subscribe: `otherObject.addEventListener('click', () => { alert('click!'); });`
  - to dispatch: `this.dispatchEvent('click');`
- Pattern 2: **Publish/Subscribe**: you don't need a specific reference to the source that triggers the event, there is a global object accessible everywhere that handles all the events.
  - to subscribe: `globalBroadcaster.subscribe('click', () => { alert('click!'); });`
  - to dispatch: `globalBroadcaster.publish('click');`
- Pattern 3: **Signals**: similar to Event Emitter/Target/Dispatcher but you don't use any random strings here. Each object that could emit events needs to have a specific property with that name. This way, you know exactly what events can an object emit.
  - to subscribe: `otherObject.clicked.add( () => { alert('click'); });`
  - to dispatch: `this.clicked.dispatch();`

Section 15.3: Parent to Child Components

That the easiest case actually, very natural in the React world and the chances are - you are already using it.

You can **pass props down to child components**. In this example `message` is the prop that we pass down to the child component, the name message is chosen arbitrarily, you can name it anything you want.

```
import React from 'react';

class Parent extends React.Component {
    render() {
        const variable = 5;
        return (
            <div>
                <Child message="message for child" />
                <Child message={variable} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return <h1>{this.props.message}</h1>
    }
}

export default Parent;
```

Here, the `<Parent />` component renders two `<Child />` components, passing `message for child` inside the first component and 5 inside the second one.

总之，你有一个组件（父组件）渲染另一个组件（子组件），并向它传递一些props。

In summary, you have a component (parent) that renders another one (child) and passes to it some props.

第16章：无状态函数式组件

第16.1节：无状态函数式组件

组件让你将用户界面拆分成独立的、可复用的部分。这就是React的魅力；我们可以将一个页面拆分成许多小的可复用组件。

在React v14之前，我们可以使用React.Component（ES6中）或React.createClass（ES5中）创建有状态的React组件，无论是否需要管理任何状态数据。

React v14引入了一种更简单的定义组件的方式，通常称为**无状态函数式组件**。这些组件使用普通的JavaScript函数。

例如：

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

这个函数是一个有效的React组件，因为它接受一个包含数据的单一props对象参数，并返回一个React元素。我们称这类组件为函数式组件，因为它们本质上是JavaScript函数。

无状态函数式组件通常专注于UI；状态应由更高级别的“容器”组件管理，或通过Flux/Redux等管理。无状态函数式组件不支持状态或生命周期方法。

优点：

1. 无类的开销
2. 不必担心这个关键词
3. 易于编写且易于理解
4. 不必担心管理状态值
5. 性能提升

总结：如果你正在编写一个不需要状态且希望创建可复用UI的React组件，你可以不创建标准的React组件，而是将其写成一个**无状态函数组件**。

让我们来看一个简单的例子：

假设我们有一个页面，可以注册用户、搜索已注册用户，或显示所有已注册用户的列表。

这是应用程序的入口点，index.js：

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

import HomePage from './homepage'

ReactDOM.render(
    <HomePage/>,
document.getElementById('app')
);
```

Chapter 16: Stateless Functional Components

Section 16.1: Stateless Functional Component

Components let you split the UI into *independent*, *reusable* pieces. This is the beauty of React; we can separate a page into many small reusable **components**.

Prior to React v14 we could create a stateful React component using React.Component (in ES6), or React.createClass (in ES5), irrespective of whether it requires any state to manage data or not.

React v14 introduced a simpler way to define components, usually referred to as **stateless functional components**. These components use plain JavaScript functions.

For example:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

This function is a valid React component because it accepts a single props object argument with data and returns a React element. We call such components **functional** because they are literally JavaScript *functions*.

Stateless functional components typically focus on UI; state should be managed by higher-level “container” components, or via Flux/Redux etc. Stateless functional components don't support state or lifecycle methods.

Benefits:

1. No class overhead
2. Don't have to worry about **this** keyword
3. Easy to write and easy to understand
4. Don't have to worry about managing state values
5. Performance improvement

Summary: If you are writing a React component that doesn't require state and would like to create a reusable UI, instead of creating a standard React Component you can write it as a **stateless functional component**.

Let's take a simple example :

Let's say we have a page that can register a user, search for registered users, or display a list of all the registered users.

This is entry point of the application, index.js:

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

import HomePage from './homepage'

ReactDOM.render(
    <HomePage/>,
    document.getElementById('app')
);
```

HomePage组件提供了注册和搜索用户的UI。注意它是一个典型的React组件，包含状态、UI和行为代码。已注册用户列表的数据存储在state变量中，但我们可复用的List（如下所示）封装了列表的UI代码。

homepage.js:

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

import List from './list';

export default class Temp extends Component{

    constructor(props) {
        super();
        this.state={users:[], showSearchResult: false, searchResult: []};
    }

registerClick(){
        let users = this.state.users.slice();
        if(users.indexOf(this.refs.mail_id.value) == -1){
            users.push(this.refs.mail_id.value);
            this.refs.mail_id.value = '';
            this.setState({users});
        }else{
alert('user already registered');
        }
    }

searchClick(){
        let users = this.state.users;
        let index = users.indexOf(this.refs.search.value);
        if(index >= 0){
            this.setState({searchResult: users[index], showSearchResult: true});
        }else{
alert('未找到该邮箱ID对应的用户');
        }
    }

hideSearchResult(){
        this.setState({showSearchResult: false});
    }

render() {
        return (
            <div>
                <input placeholder='email-id' ref='mail_id'/>
                <input type='submit' value='点击这里注册'
onClick={this.registerClick.bind(this)}/>
                <input style={{marginLeft: '100px'}} placeholder='搜索' ref='search'/>
                <input type='submit' value='点击这里注册'
onClick={this.searchClick.bind(this)}/>
                {this.state.showSearchResult ?
                    <div>
搜索结果:
                        <List users={[this.state.searchResult]}/>
                        <p onClick={this.hideSearchResult.bind(this)}>关闭</\p>
                    </div>
                    :
                    <div>
注册用户:
```

The HomePage component provides the UI to register and search for users. Note that it is a typical React component including state, UI, and behavioral code. The data for the list of registered users is stored in the state variable, but our reusable List (shown below) encapsulates the UI code for the list.

homepage.js:

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

import List from './list';

export default class Temp extends Component{

    constructor(props) {
        super();
        this.state={users:[], showSearchResult: false, searchResult: []};
    }

    registerClick(){
        let users = this.state.users.slice();
        if(users.indexOf(this.refs.mail_id.value) == -1){
            users.push(this.refs.mail_id.value);
            this.refs.mail_id.value = '';
            this.setState({users});
        }else{
            alert('user already registered');
        }
    }

    searchClick(){
        let users = this.state.users;
        let index = users.indexOf(this.refs.search.value);
        if(index >= 0){
            this.setState({searchResult: users[index], showSearchResult: true});
        }else{
            alert('no user found with this mail id');
        }
    }

    hideSearchResult(){
        this.setState({showSearchResult: false});
    }

    render() {
        return (
            <div>
                <input placeholder='email-id' ref='mail_id'/>
                <input type='submit' value='Click here to register'
onClick={this.registerClick.bind(this)}/>
                <input style={{marginLeft: '100px'}} placeholder='search' ref='search'/>
                <input type='submit' value='Click here to register'
onClick={this.searchClick.bind(this)}/>
                {this.state.showSearchResult ?
                    <div>
                        Search Result:
                        <List users={[this.state.searchResult]}/>
                        <p onClick={this.hideSearchResult.bind(this)}>Close this</p>
                    </div>
                    :
                    <div>
                        Registered users:
```

```
                        <br/>
                        {this.state.users.length ?
                            <List users={this.state.users}/>
                            :
                            "没有注册用户"
                        }
                    </div>
                }
            </div>
        );
    }
}
```

最后，我们的无状态函数组件List，用于显示注册用户列表和搜索结果，但自身不维护任何状态。

list.js:

```
import React from 'react';
var colors = ['#6A1B9A', '#76FF03', '#4527A0'];

var List = (props) => {
    return(
        <div>
            {
props.users.map((user, i)=>{
                    return(
                        <div key={i} style={{color: colors[i%3]}}>
                            {user}
                        </div>
                    );
                })
            }
        </div>
    );
}

export default List;
```

参考资料: https://facebook.github.io/react/docs/components-and-props.html

```
                        <br/>
                        {this.state.users.length ?
                            <List users={this.state.users}/>
                            :
                            "no user is registered"
                        }
                    </div>
                }
            </div>
        );
    }
}
```

Finally, our **stateless functional component** `List`, which is used display both the list of registered users *and* the search results, but without maintaining any state itself.

`list.js`:

```
import React from 'react';
var colors = ['#6A1B9A', '#76FF03', '#4527A0'];

var List = (props) => {
    return(
        <div>
            {
                props.users.map((user, i)=>{
                    return(
                        <div key={i} style={{color: colors[i%3]}}>
                            {user}
                        </div>
                    );
                })
            }
        </div>
    );
}

export default List;
```

Reference: https://facebook.github.io/react/docs/components-and-props.html

第17章：性能

第17.1节：使用ReactJS进行性能测量

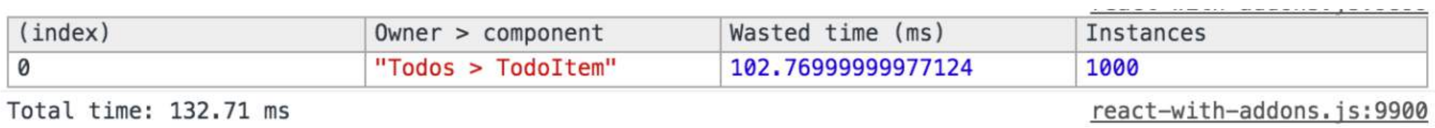
你无法改进你无法测量的东西。要提升React组件的性能，你应该能够对其进行测量。ReactJS提供了*addon*工具来测量性能。导入`react-addons-perf`模块以测量性能

```
import Perf from 'react-addons-perf' // ES6
var Perf = require('react-addons-perf') // 使用npm的ES5
var Perf = React.addons.Perf; // 使用react-with-addons.js的ES5
```

你可以使用导入的Perf模块中的以下方法：

- Perf.printInclusive()
- Perf.printExclusive()
- Perf.printWasted()
- Perf.printOperations()
- Perf.printDOM()

最重要且你大多数时间都会用到的是Perf.printWasted()，它会以表格形式显示你各个组件的浪费时间



```
(index)   Owner > component    Wasted time (ms)      Instances
0         "Todos > TodoItem"   102.76999999977124    1000
Total time: 132.71 ms                                react-with-addons.js:9900
```

你可以注意表格中的**浪费时间**列，并利用上面的**技巧与窍门**部分来提升组件的性能

参考React官方指南以及Benchling工程团队关于React性能的优秀文章

第17.2节：React的diff算法

生成将一棵树转换为另一棵树的最少操作数的复杂度是O(n^3)，其中n是树中节点的数量。React依赖两个假设来在线性时间——O(n)内解决这个问题

> 1. 同一类的两个组件会生成相似的树，不同类的两个组件会生成不同的树。

> 2. 可以为元素提供一个在不同渲染中保持稳定的唯一键。

为了判断两个节点是否不同，React区分了三种情况

1. 如果两个节点类型不同，则它们不同。

- 例如，<div>...</div> 与 <span>...</span> 是不同的

Chapter 17: Performance

Section 17.1: Performance measurement with ReactJS

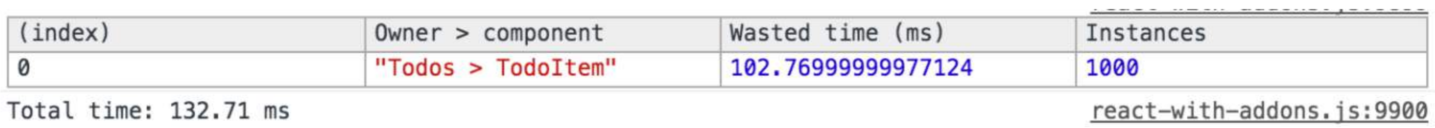
You can't improve something you can't measure. To improve the performance of React components, you should be able to measure it. ReactJS provides with *addon* tools to measure performance. Import the `react-addons-perf` module to measure the performance

```
import Perf from 'react-addons-perf' // ES6
var Perf = require('react-addons-perf') // ES5 with npm
var Perf = React.addons.Perf; // ES5 with react-with-addons.js
```

You can use below methods from the imported `Perf` module:

- Perf.printInclusive()
- Perf.printExclusive()
- Perf.printWasted()
- Perf.printOperations()
- Perf.printDOM()

The most important one which you will need most of the time is `Perf.printWasted()` which gives you the tabular representation of your individual component's wasted time



```
(index)   Owner > component    Wasted time (ms)      Instances
0         "Todos > TodoItem"   102.76999999977124    1000
Total time: 132.71 ms                                react-with-addons.js:9900
```

You can note the **Wasted time** column in the table and improve Component's performance using **Tips & Tricks** section above

Refer the React Official Guide and excellent article by Benchling Engg. on React Performance

Section 17.2: React's diff algorithm

Generating the minimum number of operations to transform one tree into another have a complexity in the order of O(n^3) where n is the number of nodes in the tree. React relies on two assumptions to solve this problem in a linear time - O(n)

> 1. Two components of the same class will generate similar trees and tw components of different classes will generate different trees.

> 2. It is possible to provide a unique key for elements that is stable across different renders.

In order to decide if two nodes are different, React differentiates 3 cases

1. Two nodes are different, if they have different types.

- for example, `<div>...</div>` is different from `<span>...</span>`

2.每当两个节点的键不同时

- 例如，`<div key="1">...</div>` 与 `<div key="2">...</div>`不同

此外，如果你想优化性能，接下来的内容至关重要且非常重要

> 如果它们[两个节点]类型不同，React甚至不会尝试匹配它们渲染的内容。
> 它只会从DOM中移除第一个节点，然后插入第二个节点。

原因如下

> 一个元素生成的DOM不太可能看起来像另一个元素生成的DOM。React不会花时间去匹配这两种结构，
> 而是直接从头重新构建树结构。

第17.3节：基础知识 - HTML DOM与虚拟DOM

HTML DOM代价高昂

每个网页在内部都表示为一个对象树。这种表示称为文档对象模型（Document Object Model）。
此外，它是一个语言无关的接口，允许编程语言（如JavaScript）访问HTML元素。

换句话说

> HTML DOM 是一种获取、更改、添加或删除 HTML 元素的标准。

然而，这些DOM 操作非常昂贵。

虚拟 DOM 是一种解决方案

因此，React 团队提出了抽象HTML DOM并创建自己的虚拟 DOM的想法，以便计算我们需要对HTML DOM执行的最少操作次数，从而复制应用程序的当前状态。

虚拟 DOM 节省了不必要的 DOM 修改时间。

具体如何实现？

在任何时刻，React 都将应用程序状态表示为一个虚拟 DOM。每当应用程序状态
发生变化时，React 会执行以下步骤以优化性能

1. 生成一个表示应用程序新状态的新虚拟 DOM
2. 比较旧的虚拟 DOM（表示当前的 HTML DOM）与新的虚拟 DOM3. 基于第 2 步，找到将旧的虚拟

DOM（表示当前的 HTML DOM）转换为新的虚拟 DOM 的最少操作数

- 要了解更多信息，请阅读 React 的 Diff 算法

4. 在找到这些操作后，它们会被映射为相应的*HTML DOM*操作

2. Whenever two nodes have different keys

- for example, `<div key="1">...</div>` is different from `<div key="2">...</div>`

Moreover, **what follows is crucial and extremely important to understand** if you want to optimise performance

> If they [two nodes] are not of the same type, React is not going to even try at matching what they render.
> It is just going to remove the first one from the DOM and insert the second one.

Here's why

> It is very unlikely that a element is going to generate a DOM that is going to look like what a would generate. Instead of spending time trying to match those two structures, React just re-builds the tree from scratch.

Section 17.3: The Basics - HTML DOM vs Virtual DOM

HTML DOM is Expensive

Each web page is represented internally as a tree of objects. This representation is called *Document Object Model*. Moreover, it is a language-neutral interface that allows programming languages (such as JavaScript) to access the HTML elements.

In other words

> The HTML DOM is a standard for how to get, change, add, or delete HTML elements.

However, those **DOM operations** are extremely **expensive**.

Virtual DOM is a Solution

So React's team came up with the idea to abstract the *HTML DOM* and create its own *Virtual DOM* in order to compute the minimum number of operations we need to apply on the *HTML DOM* to replicate current state of our application.

The Virtual DOM saves time from unnecessary DOM modifications.

How Exactly?

At each point of time, React has the application state represented as a `Virtual DOM`. Whenever application state changes, these are the steps that React performs in order to optimise performance

1. Generate a new *Virtual DOM* that represents the new state of our application
2. Compare the old Virtual DOM (which represents the current HTML DOM) vs the new Virtual DOM
3. Based on 2. find the minimum number of operations to transform the old Virtual DOM (which represents the current HTML DOM) into the new Virtual DOM

- to learn more about that - read React's Diff Algorithm

4. After those operations are found, they are mapped into their equivalent *HTML DOM* operations

- 请记住，虚拟DOM只是HTML DOM的一个抽象，它们之间存在同构关系

5. 现在已经找到并转换为等效的HTML DOM操作的最小操作数，这些操作现在直接应用于应用程序的HTML DOM，这节省了不必要修改HTML DOM的时间。

注意：应用于虚拟DOM的操作开销很小，因为虚拟DOM是一个JavaScript对象。

第17.4节：技巧与窍门

当两个节点类型不同时，React不会尝试匹配它们——它只是从DOM中移除第一个节点并插入第二个节点。这就是第一个提示的原因

1.如果你发现自己在两个输出非常相似的组件类之间切换，可能需要将它们合并为同一个类。

2.如果你知道组件不会改变，可以使用shouldComponentUpdate来防止组件重新渲染，例如

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

- remember, the *Virtual DOM* is only an abstraction of the *HTML DOM* and there is a isomorphic relation between them

5. Now the minimum number of operations that have been found and transferred to their equivalent *HTML DOM* operations are now applied directly onto the application's *HTML DOM*, which saves time from modifying the *HTML DOM* unnecessarily.

Note: Operations applied on the Virtual DOM are cheap, because the Virtual DOM is a JavaScript Object.

Section 17.4: Tips & Tricks

When two nodes are not of the same type, React doesn't try to match them - it just removes the first node from the DOM and inserts the second one. This is why the first tip says

1. If you see yourself alternating between two components classes with very similar output, you may want to make it the same class.

2. Use shouldComponentUpdate to prevent component from rerender, if you know it is not going to change, for example

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

第18章：服务器端渲染简介

第18.1节：渲染组件

在服务器上渲染组件有两种选择：renderToString和 renderToStaticMarkup。

renderToString

这将在服务器上将React组件渲染为HTML。该函数还会向HTML元素添加data-react-属性，这样客户端的React就不必再次渲染元素。

```
import { renderToString } from "react-dom/server";
renderToString(<App />);
```

renderToStaticMarkup

这将把React组件渲染为HTML，但不带data-react-属性，不建议用于将在客户端渲染的组件，因为组件会重新渲染。

```
import { renderToStaticMarkup } from "react-dom/server";
renderToStaticMarkup(<App />);
```

Chapter 18: Introduction to Server-Side Rendering

Section 18.1: Rendering components

There are two options to render components on server: `renderToString` and `renderToStaticMarkup`.

renderToString

This will render React components to HTML on server. This function will also add `data-react-` properties to HTML elements so React on client won't have to render elements again.

```
import { renderToString } from "react-dom/server";
renderToString(<App />);
```

renderToStaticMarkup

This will render React components to HTML, but without `data-react-` properties, it is not recommended to use components that will be rendered on client, because components will rerender.

```
import { renderToStaticMarkup } from "react-dom/server";
renderToStaticMarkup(<App />);
```

第19章：搭建React环境

第19.1节：简单的React组件

我们希望能够编译以下组件并将其渲染到网页中

文件名: src/index.jsx

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

class ToDo extends React.Component {
    render() {
        return (<div>我正在工作</div>);
    }
}

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

第19.2节：安装所有依赖

```
# 安装 react 和 react-dom
$ npm i react react-dom --save

# 安装用于打包的 webpack
$ npm i webpack -g

# 安装用于模块加载、打包和转译的 babel
$ npm i babel-core babel-loader --save

# 安装用于 react 和 es6 的 babel 预设
$ npm i babel-preset-react babel-preset-es2015 --save
```

第19.3节：配置 webpack

在工作目录根目录下创建文件 `webpack.config.js`

文件名: webpack.config.js

```
module.exports = {
entry: __dirname + "/src/index.jsx",
    devtool: "source-map",
output: {
path: __dirname + "/build",
        filename: "bundle.js"
    },
module: {
loaders: [
            {test: /\.jsx?$/, exclude: /node_modules/, loader: "babel-loader"}
        ]
    }
}
```

Chapter 19: Setting Up React Environment

Section 19.1: Simple React Component

We want to be able to compile below component and render it in our webpage

Filename: src/index.jsx

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

class ToDo extends React.Component {
    render() {
        return (<div>I am working</div>);
    }
}

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

Section 19.2: Install all dependencies

```
# install react and react-dom
$ npm i react react-dom --save

# install webpack for bundling
$ npm i webpack -g

# install babel for module loading, bundling and transpiling
$ npm i babel-core babel-loader --save

# install babel presets for react and es6
$ npm i babel-preset-react babel-preset-es2015 --save
```

Section 19.3: Configure webpack

Create a file `webpack.config.js` in the root of your working directory

Filename: webpack.config.js

```
module.exports = {
    entry: __dirname + "/src/index.jsx",
    devtool: "source-map",
    output: {
        path: __dirname + "/build",
        filename: "bundle.js"
    },
    module: {
        loaders: [
            {test: /\.jsx?$/, exclude: /node_modules/, loader: "babel-loader"}
        ]
    }
}
```

第19.4节：配置babel

在工作目录根目录下创建一个文件`.babelrc`

文件名: .babelrc

```
{
    "presets": ["es2015","react"]
}
```

第19.5节：用于react组件的HTML文件

在项目目录根目录下设置一个简单的html文件

文件名: index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="App"></div>
    <script src="build/bundle.js" charset="utf-8"></script>
  </body>
</html>
```

第19.6节：转译并打包你的组件

使用webpack，你可以打包你的组件：

$ webpack

这将在build目录中创建我们的输出文件。

在浏览器中打开HTML页面以查看组件的运行效果

Section 19.4: Configure babel

Create a file `.babelrc` in the root of our working directory

Filename: .babelrc

```
{
    "presets": ["es2015","react"]
}
```

Section 19.5: HTML file to use react component

Setup a simple html file in the root of the project directory

Filename: index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="App"></div>
    <script src="build/bundle.js" charset="utf-8"></script>
  </body>
</html>
```

Section 19.6: Transpile and bundle your component

Using webpack, you can bundle your component:

$ webpack

This will create our output file in `build` directory.

Open the HTML page in a browser to see component in action

第20章：在React中使用Flow

如何使用Flow类型检查器来检查React组件中的类型。

第20.1节：使用Flow检查无状态函数组件的属性类型

```
类型 Props = {
  posts: 数组<文章>,
  dispatch: 函数,
  children: React元素
}

常量 App容器 =
  ({ posts, dispatch, children }: Props) => (
    <div className="main-app">
      <头部 {...{ posts, dispatch }} />
      {children}
    </div>
  )
```

第20.2节：使用Flow检查属性类型

```
导入 React, { 组件 } 从 'react';

类型 Props = {
posts: 数组<文章>,
  dispatch: 函数,
  children: React元素
}

类 帖子 extends 组件 {
  属性: Props;

渲染 () {
    // 其余代码在此处
  }
}
```

Chapter 20: Using React with Flow

How to use the Flow type checker to check types in React components.

Section 20.1: Using Flow to check prop types of stateless functional components

```
type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

const AppContainer =
  ({ posts, dispatch, children }: Props) => (
    <div className="main-app">
      <Header {...{ posts, dispatch }} />
      {children}
    </div>
  )
```

Section 20.2: Using Flow to check prop types

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

type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

class Posts extends Component {
  props: Props;

  render () {
    // rest of the code goes here
  }
}
```

第21章：JSX

第21.1节：JSX中的Props

在JSX中，有几种不同的方式来指定props。

JavaScript表达式

你可以通过用{}包裹，传递任何JavaScript表达式作为prop。例如，在以下JSX中：

```
<MyComponent count={1 + 2 + 3 + 4} />
```

在MyComponent内部，props.count的值将是10，因为表达式1 + 2 + 3 + 4被计算了。

if语句和for循环在JavaScript中不是表达式，因此不能直接在JSX中使用。

字符串字面量

当然，你也可以直接传递任何字符串字面量作为prop。这两个JSX表达式是等价的：

```
<MyComponent message="hello world" />

<MyComponent message={'hello world'} />
```

当你传递一个字符串字面量时，其值会进行HTML反转义。因此这两个JSX表达式是等价的：

```
<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />
```

这种行为通常不相关。这里只是为了完整性而提及。

属性默认值

如果你没有为属性传递值，它默认是true。这两个JSX表达式是等价的：

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

然而，React团队在他们的文档中表示不推荐使用这种方式，因为它可能会与ES6对象简写{foo}混淆，后者是{foo: foo}的简写，而不是{foo: true}。他们说这种行为只是为了匹配HTML的行为。

展开属性

如果你已经有一个props对象，并且想在JSX中传递它，你可以使用...作为展开运算符来传递整个props对象。这两个组件是等价的：

```
function Case1() {
  return <Greeting firstName="Kaloyab" lastName="Kosev" />;
}
```

Chapter 21: JSX

Section 21.1: Props in JSX

There are several different ways to specify props in JSX.

JavaScript Expressions

You can pass **any JavaScript expression** as a prop, by surrounding it with `{}`. For example, in this JSX:

```
<MyComponent count={1 + 2 + 3 + 4} />
```

Inside the `MyComponent`, the value of `props.count` will be `10`, because the expression `1 + 2 + 3 + 4` gets evaluated.

If statements and for loops are not expressions in JavaScript, so they can't be used in JSX directly.

String Literals

Of course, you can just pass any `string literal` as a `prop` too. These two JSX expressions are equivalent:

```
<MyComponent message="hello world" />

<MyComponent message={'hello world'} />
```

When you pass a string literal, its value is HTML-unescaped. So these two JSX expressions are equivalent:

```
<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />
```

This behavior is usually not relevant. It's only mentioned here for completeness.

Props Default Value

If you pass no value for a prop, **it defaults to `true`**. These two JSX expressions are equivalent:

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

However, the React team says in their docs **using this approach is not recommended**, because it can be confused with the ES6 object shorthand `{foo}` which is short for `{foo: foo}` rather than `{foo: true}`. They say this behavior is just there so that it matches the behavior of HTML.

Spread Attributes

If you already have props as an object, and you want to pass it in JSX, you can use `...` as a spread operator to pass the whole props object. These two components are equivalent:

```
function Case1() {
  return <Greeting firstName="Kaloyab" lastName="Kosev" />;
}
```

```
function Case2() {
  const person = {firstName: 'Kaloyan', lastName: 'Kosev'};
  return <Greeting {...person} />;
}
```

第21.2节：JSX中的子元素

在同时包含开始标签和结束标签的JSX表达式中，这些标签之间的内容会作为一个特殊的属性传递：props.children。有几种不同的方式来传递子元素：

字符串字面量

你可以在开始标签和结束标签之间放置一个字符串，props.children 就是该字符串。这对于许多内置的HTML元素非常有用。例如：

```
<MyComponent>
    <h1>Hello world!</h1>
</MyComponent>
```

这是有效的JSX，MyComponent中的props.children将简单地是<h1>Hello world!</h1>。

注意，HTML是未转义的，因此你通常可以像写HTML一样编写JSX。

请记住，在这种情况下JSX：

- 删除行首和行尾的空白；
- 删除空白行；
- 标签相邻的新行将被移除；
- 字符串字面量中间出现的新行将被合并为一个空格。

JSX 子元素

您可以提供更多的 JSX 元素作为子元素。这对于显示嵌套组件非常有用：

```
<MyContainer>
  <MyFirstComponent /><
  MySecondComponent /><
/MyContainer>
```

您可以混合使用不同类型的子元素，因此可以将字符串字面量与 JSX 子元素一起使用。
这是 JSX 类似于 HTML 的另一种方式，因此这既是有效的 JSX，也是有效的 HTML：

```
<div>
  <h2>这里是一个列表</h2>
  <ul>
    <li>项目 1</li>
    <li>项目 2</li>
  </ul>
</div>
```

请注意，React 组件不能返回多个 React 元素，但单个 JSX 表达式可以有多个子元素。因此，如果你想让组件渲染多个内容，可以像上面的示例那样将它们包裹在一个div中。

```
function Case2() {
  const person = {firstName: 'Kaloyan', lastName: 'Kosev'};
  return <Greeting {...person} />;
}
```

Section 21.2: Children in JSX

In JSX expressions that contain both an opening tag and a closing tag, the content between those tags is passed as a special prop: props.children. There are several different ways to pass children:

String Literals

You can put a string between the opening and closing tags and props.children will just be that string. This is useful for many of the built-in HTML elements. For example:

```
<MyComponent>
    <h1>Hello world!</h1>
</MyComponent>
```

This is valid JSX, and props.children in MyComponent will simply be <h1>Hello world!</h1>.

Note that **the HTML is unescaped**, so you can generally write JSX just like you would write HTML.

Bare in mind, that in this case JSX:

- removes whitespace at the beginning and ending of a line;
- removes blank lines;
- new lines adjacent to tags are removed;
- new lines that occur in the middle of string literals are condensed into a single space.

JSX Children

You can provide more JSX elements as the children. This is useful for displaying nested components:

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

You can **mix together different types of children, so you can use string literals together with JSX children**. This is another way in which JSX is like HTML, so that this is both valid JSX and valid HTML:

```
<div>
  <h2>Here is a list</h2>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

Note that a React component **can't return multiple React elements, but a single JSX expression can have multiple children**. So if you want a component to render multiple things you can wrap them in a div like the example above.

JavaScript表达式

你可以通过将任意 JavaScript 表达式用{}包裹起来作为子元素传递。例如，这些表达式是等价的：

```
<MyComponent >foo</MyComponent >

<MyComponent >{'foo'}</MyComponent >
```

这通常用于渲染任意长度的 JSX 表达式列表。例如，下面代码渲染了一个 HTML 列表：

```
const Item = ({ message }) => (
  <li>{ message }</li>
);

const TodoList = () => {
  const todos = ['finish doc', 'submit review', 'wait stackoverflow review'];
  return (
    <ul>
      { todos.map(message => (<Item key={message} message={message} />)) }
    </ul>
  );
};
```

请注意，JavaScript 表达式可以与其他类型的子元素混合使用。

作为子元素的函数

通常，插入到 JSX 中的 JavaScript 表达式会被求值为字符串、React 元素或这些内容的列表。
然而，props.children 的工作方式就像其他属性一样，它可以传递任何类型的数据，而不仅仅是 React 知道如何渲染的类型。例如，如果你有一个自定义组件，你可以让它将回调作为 props.children 传入：

```
const ListOfTenThings = () => (
  <Repeat numTimes={10}>
    {(index) => <div key={index}>这是列表中的第 {index} 项</div>}
  </Repeat>
);

// 调用 children 回调 numTimes 次以生成重复的组件
const Repeat = ({ numTimes, children }) => {
  let items = [];
  for (let i = 0; i < numTimes; i++) {
    items.push(children(i));
  }
  return <div>{items}</div>;
};
```

传递给自定义组件的子元素可以是任何内容，只要该组件在渲染之前将它们转换成 React 能理解的内容即可。这种用法不常见，但如果你想扩展 JSX 的能力，它是可行的。

被忽略的值

请注意，false、null、undefined 和 true 都是有效的子元素。但它们不会被渲染。这些 JSX 表达式

JavaScript Expressions

You can pass any JavaScript expression as children, by enclosing it within {}. For example, these expressions are equivalent:

```
<MyComponent>foo</MyComponent>

<MyComponent>{'foo'}</MyComponent>
```

This is often useful for rendering a list of JSX expressions of arbitrary length. For example, this renders an HTML list:

```
const Item = ({ message }) => (
  <li>{ message }</li>
);

const TodoList = () => {
  const todos = ['finish doc', 'submit review', 'wait stackoverflow review'];
  return (
    <ul>
      { todos.map(message => (<Item key={message} message={message} />)) }
    </ul>
  );
};
```

Note that JavaScript expressions can be mixed with other types of children.

Functions as Children

Normally, JavaScript expressions inserted in JSX will evaluate to a string, a React element, or a list of those things. However, `props.children` works just like any other prop in that it can pass any sort of data, not just the sorts that React knows how to render. For example, if you have a custom component, you could have it take a callback as `props.children`:

```
const ListOfTenThings = () => (
  <Repeat numTimes={10}>
    {(index) => <div key={index}>This is item {index} in the list</div>}
  </Repeat>
);

// Calls the children callback numTimes to produce a repeated component
const Repeat = ({ numTimes, children }) => {
  let items = [];
  for (let i = 0; i < numTimes; i++) {
    items.push(children(i));
  }
  return <div>{items}</div>;
};
```

Children passed to a custom component can be anything, as long as that component transforms them into something React can understand before rendering. This usage is not common, but it works if you want to stretch what JSX is capable of.

Ignored Values

Note that **`false`**, **`null`**, **`undefined`**, and **`true`** are valid children. But they simply don't render. These JSX expressions

都会渲染成相同的结果：

```
<MyComponent  />

<MyComponent></MyComponent>

<MyComponent >{false}</MyComponent >

<MyComponent >{null}</MyComponent >

<MyComponent >{true}</MyComponent >
```

这对于有条件地渲染 React 元素非常有用。以下 JSX 只有在 showHeader 为 true 时才会渲染 a：

```
<div>
{showHeader && <Header />}
 <Content />
</div>
```

一个重要的注意点是，一些“假值”，例如数字 0，仍然会被 React 渲染。例如，
这段代码不会按你预期的那样工作，因为当 props.messages 是空数组时，0 会被打印出来：

```
<div>
{props.messages.length &&
   <MessageList messages={props.messages} />
 }
</div>
```

解决此问题的一种方法是确保&&前的表达式始终为布尔值：

```
<div>
{props.messages.length > 0 &&
   <MessageList messages={props.messages} />
 }
</div>
```

最后，请记住，如果你想让false、true、null或undefined这样的值出现在输出中，必须先将其转换为字符串：

```
<div>
我的JavaScript变量是 {String(myVariable)}。
</div>
```

will all render to the same thing:

```
<MyComponent />

<MyComponent></MyComponent>

<MyComponent>{false}</MyComponent>

<MyComponent>{null}</MyComponent>

<MyComponent>{true}</MyComponent>
```

This is extremely useful to conditionally render React elements. This JSX only renders a if `showHeader` is true:

```
<div>
  {showHeader && <Header />}
 <Content />
</div>
```

One important caveat is that some "falsy" values, such as the `0` number, are still rendered by React. For example, this code will not behave as you might expect because `0` will be printed when `props.messages` is an empty array:

```
<div>
  {props.messages.length &&
   <MessageList messages={props.messages} />
 }
</div>
```

One approach to fix this is to make sure that the expression before the `&&` is always boolean:

```
<div>
  {props.messages.length > 0 &&
   <MessageList messages={props.messages} />
 }
</div>
```

Lastly, bare in mind that if you want a value like **`false`**, **`true`**, **`null`**, or **`undefined`** to appear in the output, you have to convert it to a string first:

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

第22章：React表单

第22.1节：受控组件

受控组件绑定到一个值，其变化通过基于事件的回调在代码中处理。

```
class CustomForm extends React.Component {
constructor() {
    super();
    this.state = {
        person: {
            firstName: '',
            lastName: ''
        }
    }
}

handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}

render() {
    return (
        <form>
            <input
type="text"
name="firstName"
            value={this.state.firstName}
            onChange={this.handleChange.bind(this)} />

            <input
type="text"
name="lastName"
value={this.state.lastName}
            onChange={this.handleChange.bind(this)} />
        </form>
    )
}
```

}

在此示例中，我们用一个空的 person 对象初始化状态。然后将两个输入框的值绑定到 person 对象的各个键上。
接着，当用户输入时，我们在 handleChange 函数中捕获每个值。
由于组件的值绑定到了状态，我们可以通过调用 setState() 来在用户输入时重新渲染。

注意： 在处理受控组件时，如果不调用 setState()，用户虽然可以输入，但不会看到输入内容，因为 React 只有在被告知时才会渲染更改。

同样重要的是要注意，输入的名称与 person 对象中键的名称相同。
这使我们能够以字典形式捕获值，如下所示。

```
handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
```

Chapter 22: React Forms

Section 22.1: Controlled Components

A controlled component is bound to a value and its changes get handled in code using event based callbacks.

```
class CustomForm extends React.Component {
constructor() {
    super();
    this.state = {
        person: {
            firstName: '',
            lastName: ''
        }
    }
}

handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}

render() {
    return (
        <form>
            <input
            type="text"
            name="firstName"
            value={this.state.firstName}
            onChange={this.handleChange.bind(this)} />

            <input
            type="text"
            name="lastName"
            value={this.state.lastName}
            onChange={this.handleChange.bind(this)} />
        </form>
    )
}
```

}

In this example we initialize state with an empty person object. We then bind the values of the 2 inputs to the individual keys of the person object. Then as the user types, we capture each value in the `handleChange` function. Since the values of the components are bound to state we can rerender as the user types by calling `setState()`.

NOTE: Not calling `setState()` when dealing with controlled components, will cause the user to type, but not see the input because React only renders changes when it is told to do so.

It's also important to note that the names of the inputs are same as the names of the keys in the person object. This allows us to capture the value in dictionary form as seen here.

```
handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}
```

person[event.target.name] 与 person.firstName || person.lastName 相同。当然，这取决于当前正在输入的是哪个输入框。由于我们不知道用户会在哪里输入，使用字典并将输入名称与键的名称匹配，可以让我们无论 onChange 从哪里调用，都能捕获用户输入。

```
}
```

`person[event.target.name]` is the same is a `person.firstName || person.lastName`. Of course this would depend on which input is currently being typed in. Since we don't know where the user will be typing, using a dictionary and matching the input names to the names of the keys, allows us to capture the user input no matter where the onChange is being called from.

第23章：用户界面解决方案

假设我们从程序中使用的现代用户界面获得一些灵感，并将其转换为 React 组件。这就是“用户界面解决方案”主题的内容。感谢署名。

第23.1节：基本面板

```
import React from 'react';

class Pane extends React.Component {
    constructor(props) {
        super(props);
    }

render() {
        return React.createElement(
            'section', this.props
        );
    }
}
```

第23.2节：面板

```
import React from 'react';

class Panel extends React.Component {
    constructor(props) {
        super(props);
    }

render(...elements) {
        var props = Object.assign({
className: this.props.active ? 'active' : '',
            tabIndex: -1
        }, this.props);

        var css = this.css();
        if (css != '') {
elements.unshift(React.createElement(
                'style', null,
css
            ));
        }

        return React.createElement(
            'div', props,
...元素
        );
    }

    static title() {
        return '';
    }
    static css() {
        return '';
    }
}
```

Chapter 23: User interface solutions

Let's say we get inspired of some ideas from modern user interfaces used in programs and convert them to React components. That's what "**User interface solutions**" topic consists of. Attribution is appretiated.

Section 23.1: Basic Pane

```
import React from 'react';

class Pane extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'section', this.props
        );
    }
}
```

Section 23.2: Panel

```
import React from 'react';

class Panel extends React.Component {
    constructor(props) {
        super(props);
    }

    render(...elements) {
        var props = Object.assign({
            className: this.props.active ? 'active' : '',
            tabIndex: -1
        }, this.props);

        var css = this.css();
        if (css != '') {
            elements.unshift(React.createElement(
                'style', null,
                css
            ));
        }

        return React.createElement(
            'div', props,
            ...elements
        );
    }

    static title() {
        return '';
    }
    static css() {
        return '';
    }
}
```

与简单面板的主要区别是：

- 当面板被脚本调用或鼠标点击时，面板具有焦点；面板为每个组件拥有 title
- 静态方法，因此可以被其他面板组件通过重写 title 来扩展（这里的原因是该函数可以在渲染时再次调用以实现本地化，但在本示例范围内 title 没有实际意义）；
- 它可以包含在 css 静态方法中声明的单独样式表（你可以预加载文件内容来自 PANEL.css）。

第23.3节：标签页

```
import React from 'react';

class Tab extends React.Component {
    constructor(props) {
        super(props);
    }

render() {
        var props = Object.assign({
className: this.props.active ? 'active' : ''
        }, this.props);
        return React.createElement(
            'li', props,
React.createElement(
                'span', props,
props.panelClass.title()
            )
        );
    }
}
```

Tab实例的panelClass属性必须包含用于描述的panel类。

第23.4节：PanelGroup

```
import React from 'react';
import Tab from './Tab.js';

class PanelGroup extends React.Component {
    constructor(props) {
        super(props);
        this.setState({
panels: props.panels
        });
    }

render() {
        this.tabSet = [];
        this.panelSet = [];
        for (let panelData of this.state.panels) {
            var tabIsActive = this.state.activeTab == panelData.name;
            this.tabSet.push(React.createElement(
Tab, {
名称: panelData.name,
                    激活: tabIsActive,
                    面板类: panelData.class,
                    鼠标按下: () => this.openTab(panelData.name)
                    }
```

Major differences from simple pane are:

- panel has focus in instance when it is called by script or clicked by mouse;
- panel has `title` static method per component, so it may be extended by other panel component with overridden `title` (reason here is that function can be then called again on rendering for localization purposes, but in bounds of this example `title` doesn't make sense);
- it can contain individual stylesheet declared in `css` static method (you can pre-load file contents from `PANEL.css`).

Section 23.3: Tab

```
import React from 'react';

class Tab extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        var props = Object.assign({
            className: this.props.active ? 'active' : ''
        }, this.props);
        return React.createElement(
            'li', props,
            React.createElement(
                'span', props,
                props.panelClass.title()
            )
        );
    }
}
```

`panelClass` property of `Tab` instance must contain class of *panel* used for description.

Section 23.4: PanelGroup

```
import React from 'react';
import Tab from './Tab.js';

class PanelGroup extends React.Component {
    constructor(props) {
        super(props);
        this.setState({
            panels: props.panels
        });
    }

    render() {
        this.tabSet = [];
        this.panelSet = [];
        for (let panelData of this.state.panels) {
            var tabIsActive = this.state.activeTab == panelData.name;
            this.tabSet.push(React.createElement(
                Tab, {
                    name: panelData.name,
                    active: tabIsActive,
                    panelClass: panelData.class,
                    onMouseDown: () => this.openTab(panelData.name)
                }
            ));
            this.panelSet.push(React.createElement(
                panelData.class, {
                    id: panelData.name,
                    active: tabIsActive,
                    ref: tabIsActive ? 'activePanel' : null
                }
            ));
        }
        return React.createElement(
            'div', { className: 'PanelGroup' },
            React.createElement(
                'nav', null,
React.createElement(
                    'ul', null,
...this.tabSet
                )
            ),
...this.panelSet
        );
    }

openTab(name) {
        this.setState({ activeTab: name });
        this.findDOMNode(this.refs.activePanel).focus();
    }
}
```

panels PanelGroup 实例的属性必须包含对象数组。每个对象声明了关于面板的重要数据：

- name - 控制器脚本使用的面板标识符；
- class - 面板的类。

别忘了将属性activeTab设置为所需标签的名称。

说明

当标签处于激活状态时，所需面板的DOM元素会获得类名active（表示该面板将可见），并且当前获得焦点。

第23.5节：带有`PanelGroup`的示例视图

```
import React from 'react';
import Pane from './components/Pane.js';
import Panel from './components/Panel.js';
import PanelGroup from './components/PanelGroup.js';

class MainView extends React.Component {
constructor(props) {
        super(props);
    }

render() {
        return React.createElement(
            'main', null,
React.createElement(
                Pane, { id: 'common' },
                React.createElement(
                    PanelGroup, {
```

```
            ));
            this.panelSet.push(React.createElement(
                panelData.class, {
                    id: panelData.name,
                    active: tabIsActive,
                    ref: tabIsActive ? 'activePanel' : null
                }
            ));
        }
        return React.createElement(
            'div', { className: 'PanelGroup' },
            React.createElement(
                'nav', null,
                React.createElement(
                    'ul', null,
                    ...this.tabSet
                )
            ),
            ...this.panelSet
        );
    }

    openTab(name) {
        this.setState({ activeTab: name });
        this.findDOMNode(this.refs.activePanel).focus();
    }
}
```

`panels` property of `PanelGroup` instance must contain array with objects. Every object there declares important data about panels:

- **`name`** - identifier of panel used by controller script;
- **`class`** - panel's class.

Don't forget to set property `activeTab` to name of needed tab.

Clarification

When tab is down, needed panel is getting class name `active` on DOM element (means that it gonna be visible) and it's focused now.

Section 23.5: Example view with `PanelGroup`s

```
import React from 'react';
import Pane from './components/Pane.js';
import Panel from './components/Panel.js';
import PanelGroup from './components/PanelGroup.js';

class MainView extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'main', null,
            React.createElement(
                Pane, { id: 'common' },
                React.createElement(
                    PanelGroup, {
```

```
面板: [
                            {
名称: 'console',
                           面板类: ConsolePanel
                            },
                            {
名称: 'figures',
                           面板类: FiguresPanel
                            }
                        ],
活动标签: 'console'
                    }
                )
            ),
React.createElement(
             Pane, { id: 'side' },
             React.createElement(
                 PanelGroup, {
                    面板: [
                            {
名称: 'properties',
                           面板类: PropertiesPanel
                            }
                        ],
活动标签: 'properties'
                    }
                )
            )
        );
    }
}

class ConsolePanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return '控制台';
    }
}

class FiguresPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return '图形';
    }
}

class PropertiesPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return '属性';
    }
}
```

```
                        panels: [
                            {
                                name: 'console',
                                panelClass: ConsolePanel
                            },
                            {
                                name: 'figures',
                                panelClass: FiguresPanel
                            }
                        ],
                        activeTab: 'console'
                    }
                )
            ),
            React.createElement(
                Pane, { id: 'side' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'properties',
                                panelClass: PropertiesPanel
                            }
                        ],
                        activeTab: 'properties'
                    }
                )
            )
        );
    }
}

class ConsolePanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Console';
    }
}

class FiguresPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Figures';
    }
}

class PropertiesPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Properties';
    }
}
```

第24章：以Flux方式使用ReactJS

当你的ReactJS前端应用计划扩展时，使用Flux方法非常方便，
因为它结构有限，并且只需少量新代码即可使运行时的状态更易于更改。

第24.1节：数据流

> 这是全面概述的大纲。

Flux模式假设使用单向数据流。

1. 动作—描述动作类型(type)及其他输入数据的简单对象。
2. 调度器—单一动作接收器和回调控制器。可以把它想象成应用程序的中央枢纽。
3. 存储—包含应用程序状态和逻辑。它在调度器中注册回调，并在数据层发生变化时向视图发出事件。当数据层发生变化时。
4. 视图—接收来自存储的变化事件和数据的React组件。当有变化时，它会触发重新渲染。当某些内容发生变化时。

> 根据Flux数据流，视图也可以创建动作并将其传递给调度器以响应用户交互。

已还原

为了更清楚起见，我们可以从结尾开始。

- 不同的React组件（视图）从不同的存储中获取有关已做更改的数据。

> 少数组件可以称为controller-views，因为它们提供了将数据从存储中获取并传递给其后代链的粘合代码。Controller-views代表页面中的任何重要部分。

- Stores可以被看作是回调函数，用于比较动作类型和其他输入数据，以实现应用程序的业务逻辑。
- Dispatcher是通用的动作接收器和回调容器。
- Actions不过是带有必需的 type 属性的简单对象。

> 以前，你会想使用常量来表示动作类型和辅助方法（称为**action creators**）。

Chapter 24: Using ReactJS in Flux way

It comes very handy to use Flux approach, when your application with ReactJS on frontend is planned to grow, because of limited structures and a little bit of new code to make state changes in runtime more easing.

Section 24.1: Data Flow

> This is outline of comprehensive Overview.

Flux pattern assumes the use of unidirectional data flow.

1. **Action** — simple object describing action `type` and other input data.
2. **Dispatcher** — single action receiver and callbacks controller. Imagine it is central hub of your application.
3. **Store** — contains the application state and logic. It registers callback in dispatcher and emits event to view when change to the data layer has occurred.
4. **View** — React component that receives change event and data from store. It causes re-rendering when something is changed.

> As of Flux data flow, views may also **create actions** and pass them to dispatcher for user interactions.

Reverted

To make it more clearer, we can start from the end.

- Different React components (*views*) get data from different stores about made changes.

> Few components may be called **controller-views**, cause they provide the glue code to get the data from the stores and to pass data down the chain of their descendants. Controller-views represent any significant section of the page.

- *Stores* can be remarked as callbacks that compare action type and other input data for business logic of your application.
- *Dispatcher* is common actions receiver and callbacks container.
- *Actions* are nothing than simple objects with required `type` property.

> Formerly, you'll want to use constants for action types and helper methods (called **action creators**).

第25章：React、Webpack与TypeScript安装

第25.1节：webpack.config.js

```
module.exports = {
    entry: './src/index',
    output: {
path: __dirname + '/build',
        filename: 'bundle.js'
    },
module: {
rules: [{
test: /\.tsx?$/,
            loader: 'ts-loader',
            exclude: /node_modules/
        }]
    },
resolve: {
extensions: ['.ts', '.tsx']
    }
};
```

主要组件包括（除了标准的entry、output和其他webpack属性）：

加载器

为此，您需要创建一个规则，用于测试.ts和.tsx文件扩展名，指定 ts-loader作为加载器。

解析TS扩展名

您还需要在resolve数组中添加.ts和.tsx扩展名，否则webpack无法识别它们。

第25.2节：tsconfig.json

这是一个最小的tsconfig，帮助您快速启动。

```
{
    "include": [
        "src/*"
    ],
    "compilerOptions": {
        "target": "es5",
        "jsx": "react",
        "allowSyntheticDefaultImports": true
    }
}
```

让我们逐一了解这些属性：

include

这是一个源代码数组。这里我们只有一个条目，src/*，指定了将包含 src 目录中的所有内容进行编译。

compilerOptions.target

Chapter 25: React, Webpack & TypeScript installation

Section 25.1: webpack.config.js

```
module.exports = {
    entry: './src/index',
    output: {
        path: __dirname + '/build',
        filename: 'bundle.js'
    },
    module: {
        rules: [{
            test: /\.tsx?$/,
            loader: 'ts-loader',
            exclude: /node_modules/
        }]
    },
    resolve: {
        extensions: ['.ts', '.tsx']
    }
};
```

The main components are (in addition to the standard entry, output and other webpack properties):

The loader

For this you need to create a rule that tests for the .ts and .tsx file extensions, specify ts-loader as the loader.

Resolve TS extensions

You also need to add the .ts and .tsx extensions in the resolve array, or webpack won't see them.

Section 25.2: tsconfig.json

This is a minimal tsconfig to get you up and running.

```
{
    "include": [
        "src/*"
    ],
    "compilerOptions": {
        "target": "es5",
        "jsx": "react",
        "allowSyntheticDefaultImports": true
    }
}
```

Let's go through the properties one by one:

include

This is an array of source code. Here we have only one entry, src/*, which specifies that everything in the src directory is to be included in compilation.

compilerOptions.target

指定我们要编译成 ES5 目标

compilerOptions.jsx

将此设置为`true`将使TypeScript自动将你的tsx语法从`<div />`编译为`React.createElement("div")`。

compilerOptions.allowSyntheticDefaultImports

一个方便的属性，允许你像导入ES6模块一样导入node模块，因此你不必这样写

```
import * as React from 'react'
const { Component } = React
```

你可以直接这样写

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

不会出现React没有默认导出的错误。

第25.3节：我的第一个组件

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';


interface AppProps {
    name: string;
}
interface AppState {
    words: string[];
}

class App extends Component<AppProps, AppState> {
    constructor() {
        super();
        this.state = {
words: ['foo', 'bar']
        };
    }

render() {
        const { name } = this.props;
        return (<h1>Hello {name}!</h1>);
    }
}

const root = document.getElementById('root');
ReactDOM.render(<App name="Foo Bar" />, root);
```

当在 React 中使用 TypeScript 时，一旦你下载了 React 的 DefinitelyTyped 类型定义（`npm install --save @types/react`），每个组件都需要你添加类型注解。

你可以这样做：

```
class App extends Component<AppProps, AppState> { }
```

其中 AppProps 和 AppState 分别是你组件的 props 和 state 的接口（或类型别名）。

Specifies that we want to compile to ES5 target

compilerOptions.jsx

Setting this to **true** will make TypeScript automatically compile your tsx syntax from `<div />` to `React.createElement("div")`.

compilerOptions.allowSyntheticDefaultImports

Handy property which will allow you to import node modules as if they are ES6 modules, so instead of doing

```
import * as React from 'react'
const { Component } = React
```

you can just do

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

without any errors telling you that React has no default export.

Section 25.3: My First Component

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';


interface AppProps {
    name: string;
}
interface AppState {
    words: string[];
}

class App extends Component<AppProps, AppState> {
    constructor() {
        super();
        this.state = {
            words: ['foo', 'bar']
        };
    }

    render() {
        const { name } = this.props;
        return (<h1>Hello {name}!</h1>);
    }
}

const root = document.getElementById('root');
ReactDOM.render(<App name="Foo Bar" />, root);
```

When using TypeScript with React, once you've downloaded the React DefinitelyTyped type definitions (`npm install --save @types/react`), every component will require you to add type annotations.

You do this like so:

```
class App extends Component<AppProps, AppState> { }
```

where `AppProps` and `AppState` are interfaces (or type aliases) for your components' props and state respectively.

第26章：如何以及为什么在React中使用key

每当你渲染一组 React 组件时，每个组件都需要有一个key属性。这个 key 可以是任何值，但必须在该列表中唯一。

当 React 需要对一组列表项进行渲染更改时，React 会同时遍历两个子元素列表，并在发现差异时生成变更。如果子元素没有设置 key，React 会扫描每个子元素。否则，React 会比较 key 以确定哪些被添加或移除。

第26.1节：基本示例

对于无类的 React 组件：

```
function SomeComponent(props){

    const ITEMS = ['cat', 'dog', 'rat']
    function getItemsList(){
        return ITEMS.map(item => <li key={item}>{item}</i>);
    }

    return (
        <ul>
            {getItemsList()}
        </ul>
    );
}
```

对于此示例，上述组件解析为：

```
<ul>
    <li key='cat'>cat</li>
    <li key='dog'>dog</li>
    <li key='rat'>rat</li>
<ul>
```

Chapter 26: How and why to use keys in React

Whenever you are rendering a list of React components, each component needs to have a key attribute. The key can be any value, but it does need to be unique to that list.

When React has to render changes on a list of items, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference. If there are no keys set for the children, React scans each child. Otherwise, React compares the keys to know which were added or removed from the list

Section 26.1: Basic Example

For a class-less React component:

```
function SomeComponent(props){

    const ITEMS = ['cat', 'dog', 'rat']
    function getItemsList(){
        return ITEMS.map(item => <li key={item}>{item}</i>);
    }

    return (
        <ul>
            {getItemsList()}
        </ul>
    );
}
```

For this example, the above component resolves to:

```
<ul>
    <li key='cat'>cat</li>
    <li key='dog'>dog</li>
    <li key='rat'>rat</li>
<ul>
```

第27章：React中的键

React中的键用于在内部识别同一层级的DOM元素列表。

因此，如果你正在遍历一个数组以显示一组li元素，每个li元素都需要一个由key属性指定的唯一标识符。这个标识符通常可以是数据库项的id或数组的索引。

第27.1节：使用元素的id

这里我们有一个待办事项列表，该列表作为props传递给我们的组件。

每个待办事项都有text和id属性。假设id属性来自后端数据存储，是一个唯一的数字值：

```
todos = [
  {
id: 1,
    text: 'value 1'
  },
  {
id: 2,
    text: 'value 2'
  },
  {
id: 3,
    text: 'value 3'
  },
  {
id: 4,
    text: '值 4'
  },
];
```

我们将每个迭代列表元素的 key 属性设置为 todo-${todo.id}，以便 React 能在内部识别它：

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo) =>
        <li key={ `todo-${todo.id}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

第27.2节：使用数组索引

如果手头没有唯一的数据库 ID，也可以像这样使用数组的数字索引：

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo, index) =>
```

Chapter 27: Keys in react

Keys in react are used to identify a list of DOM elements from the same hierarchy internally.

So if you are iterating over an array to show a list of li elements, each of the li elements needs a unique identifier specified by the key property. This usually can be the id of your database item or the index of the array.

Section 27.1: Using the id of an element

Here we are having a list of todo items that is passed to the props of our component.

Each todo item has a text and id property. Imagine that the id property comes from a backend datastore and is a unique numeric value:

```
todos = [
  {
    id: 1,
    text: 'value 1'
  },
  {
    id: 2,
    text: 'value 2'
  },
  {
    id: 3,
    text: 'value 3'
  },
  {
    id: 4,
    text: 'value 4'
  },
];
```

We set the key attribute of each iterated list element to `todo-${todo.id}` so that react can identify it internally:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo) =>
        <li key={ `todo-${todo.id}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Section 27.2: Using the array index

If you don't have unique database ids at hand, you could also use the numeric index of your array like this:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo, index) =>
        <li key={ `todo-${index}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

```
        <li key={ `todo-${index}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

第28章：高阶组件

高阶组件（简称“HOC”）是一种React应用设计模式，用于通过可复用代码增强组件。它们能够为现有组件类添加功能和行为。

高阶组件是一个纯JavaScript函数，接受一个组件作为参数，并返回一个具有扩展功能的新组件。

第28.1节：检查身份验证的高阶组件

假设我们有一个组件，只有在用户登录时才应显示。

因此，我们创建一个高阶组件，在每次render()时检查身份验证：

AuthenticatedComponent.js

```
import React from "react";

export function requireAuthentication(Component) {
    return class AuthenticatedComponent extends React.Component {

        /**
* 检查用户是否已认证，this.props.isAuthenticated        * 必须由您的应用程序逻辑设置
（或者使用 react-redux 从全局状态中获取）。

         */
isAuthenticated() {
            return this.props.isAuthenticated;
        }

        /**
* 渲染
         */
render() {
            const loginErrorMessage = (
                <div>
请<a href="/login">登录</a> 以查看此部分
应用程序。
                </div>
            );

            return (
                <div>
                    { this.isAuthenticated === true ? <Component {...this.props} /> :
loginErrorMessage }
                </div>
            );
        }
    };
}

export default requireAuthentication;
```

然后我们只需在应该对匿名用户隐藏的组件中使用这个高阶组件：

MyPrivateComponent.js

Chapter 28: Higher Order Components

Higher Order Components ("HOC" in short) is a react application design pattern that is used to enhance components with reusable code. They enable to add functionality and behaviors to existing component classes.

A HOC is a pure javascript function that accepts a component as it's argument and returns a new component with the extended functionality.

Section 28.1: Higher Order Component that checks for authentication

Let's say we have a component that should only be displayed if the user is logged in.

So we create a HOC that checks for the authentication on each render():

AuthenticatedComponent.js

```
import React from "react";

export function requireAuthentication(Component) {
    return class AuthenticatedComponent extends React.Component {

        /**
         * Check if the user is authenticated, this.props.isAuthenticated
         * has to be set from your application logic (or use react-redux to retrieve it from global
state).
         */
        isAuthenticated() {
            return this.props.isAuthenticated;
        }

        /**
         * Render
         */
        render() {
            const loginErrorMessage = (
                <div>
                    Please <a href="/login">login</a> in order to view this part of the
application.
                </div>
            );

            return (
                <div>
                    { this.isAuthenticated === true ? <Component {...this.props} /> :
loginErrorMessage }
                </div>
            );
        }
    };
}

export default requireAuthentication;
```

We then just use this Higher Order Component in our components that should be hidden from anonymous users:

MyPrivateComponent.js

```
import React from "react";
import {requireAuthentication} from "./AuthenticatedComponent";

export class MyPrivateComponent extends React.Component {
    /**
* 渲染
     */
render() {
        return (
            <div>
我的秘密搜索，只有经过身份验证的用户才能查看。
            </div>
        );
    }
}

// 现在用 requireAuthentication 函数包装 MyPrivateComponent
export default requireAuthentication(MyPrivateComponent);
```

这个示例在这里有更详细的描述。

第28.2节：简单的高阶组件

假设我们想在组件每次挂载时打印日志：

hocLogger.js

```
export default function hocLogger(组件) {
  return class extends React.Component {
componentDidMount() {
      console.log('嘿，我们已经挂载了！');
    }
render() {
      return <组件 {...this.props} />;
    }
  }
}
```

在你的代码中使用这个高阶组件（HOC）：

MyLoggedComponent.js

```
import React from "react";
import {hocLogger} from "./hocLogger";

export class MyLoggedComponent extends React.Component {
    render() {
        return (
            <div>
该组件在每次挂载时都会记录到控制台。
            </div>
        );
    }
}

// 现在用 hocLogger 函数包装 MyLoggedComponent
export default hocLogger(MyLoggedComponent);
```

```
import React from "react";
import {requireAuthentication} from "./AuthenticatedComponent";

export class MyPrivateComponent extends React.Component {
    /**
     * Render
     */
    render() {
        return (
            <div>
                My secret search, that is only viewable by authenticated users.
            </div>
        );
    }
}

// Now wrap MyPrivateComponent with the requireAuthentication function
export default requireAuthentication(MyPrivateComponent);
```

This example is described in more detail here.

Section 28.2: Simple Higher Order Component

Let's say we want to console.log each time the component mounts:

hocLogger.js

```
export default function hocLogger(Component) {
  return class extends React.Component {
    componentDidMount() {
      console.log('Hey, we are mounted!');
    }
    render() {
      return <Component {...this.props} />;
    }
  }
}
```

Use this HOC in your code:

MyLoggedComponent.js

```
import React from "react";
import {hocLogger} from "./hocLogger";

export class MyLoggedComponent extends React.Component {
    render() {
        return (
            <div>
                This component gets logged to console on each mount.
            </div>
        );
    }
}

// Now wrap MyLoggedComponent with the hocLogger function
export default hocLogger(MyLoggedComponent);
```

第29章：React 与 Redux

如今，Redux 已成为管理前端应用级状态的现状，那些从事“大型应用”开发的人常常对此深信不疑。本文将介绍为什么以及如何在 React 应用中使用状态管理库 Redux。

第29.1节：使用 Connect

使用createStore创建 Redux 存储。

```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp, { inistialStateVariable: "derp"})
```

使用connect将组件连接到 Redux 存储，并将存储中的属性传递给组件。

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

定义允许您的组件向 Redux 存储发送消息的操作。

```
/*
* 操作类型
 */

export const ADD_TODO = 'ADD_TODO'

export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

处理这些消息并在 reducer 函数中为存储创建一个新的状态。

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Chapter 29: React with Redux

Redux has come to be the status quo for managing application-level state on the front-end these days, and those who work on "large-scale applications" often swear by it. This topic covers why and how you should use the state management library, Redux, in your React applications.

Section 29.1: Using Connect

Create a Redux store with *createStore*.

```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp, { inistialStateVariable: "derp"})
```

Use *connect* to connect component to Redux store and pull props from store to component.

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Define actions that allow your components to send messages to the Redux store.

```
/*
 * action types
 */

export const ADD_TODO = 'ADD_TODO'

export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

Handle these messages and create a new state for the store in reducer functions.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

附录 A：安装

A.1 节：简单设置

设置文件夹

本示例假设代码位于 `src/`，输出放置在 `out/`。因此，文件夹结构应类似于

```
example/
|-- src/
|   |-- index.js
|   `-- ...
|-- out/
`-- package.json
```

设置包

假设已设置 npm 环境，我们首先需要设置 babel，以将 React 代码转译为符合 es5 标准的代码。

```
$npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
```

上述命令将指示 npm 安装核心 babel 库以及用于 webpack 的加载器模块。我们还安装了 babel 的 es6 和 react 预设，以便其理解 JSX 和 es6 模块代码。（关于预设的更多信息可见此处 Babel presets）

```
$npm i -D webpack
```

此命令将安装 webpack 作为开发依赖。（i 是 install 的简写，-D 是 --save-dev 的简写）

你可能还想安装任何额外的webpack包（例如额外的加载器或webpack-dev-server扩展）

最后我们需要实际的React代码

```
$npm i -D react react-dom
```

设置webpack

依赖项设置好后，我们需要一个webpack.config.js文件来告诉webpack该做什么

简单的webpack.config.js：

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

module.exports = {
entry: './src/index.js',
  output: {
path: path.resolve(__dirname, 'out'),
    filename: 'bundle.js'
  },
module: {
    loaders: [
      {
test: /\.js$/,
```

Appendix A: Installation

Section A.1: Simple setup

Setting up the folders

This example assumes code to be in `src/` and the output to be put into `out/`. As such the folder structure should look something like

```
example/
|-- src/
|   |-- index.js
|   `-- ...
|-- out/
`-- package.json
```

Setting up the packages

Assuming a setup npm environment, we first need to setup babel in order to transpile the React code into es5 compliant code.

```
$npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
```

The above command will instruct npm to install the core babel libraries as well as the loader module for use with webpack. We also install the es6 and react presets for babel to understand JSX and es6 module code. (More information about the presets can be found here Babel presets)

```
$npm i -D webpack
```

This command will install webpack as a development dependency. (**i** is the shorthand for install and **-D** the shorthand for --save-dev)

You might also want to install any additional webpack packages (such as additional loaders or the webpack-dev-server extension)

Lastly we will need the actual react code

```
$npm i -D react react-dom
```

Setting up webpack

With the dependencies setup we will need a webpack.config.js file to tell webpack what to do

simple webpack.config.js:

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
exclude: /(node_modules)/,
        loader: 'babel-loader',
        query: {
presets: ['es2015', 'react']
        }
      }
    ]
  }
};
```

该文件告诉 webpack 从 index.js 文件（假定位于 src/ 目录）开始，将其转换成一个单独的 bundle.js 文件，输出到 out 目录中。

module 块告诉 webpack 对遇到的所有文件进行正则表达式测试，如果匹配，则调用指定的加载器（此处为 babel-loader）。此外，exclude 正则表达式告诉 webpack 忽略 node_modules 文件夹中的所有模块使用该特殊加载器，这有助于加快转译过程。最后，query 选项告诉 webpack 传递给 babel 的参数，用于传递我们之前安装的 presets。

测试设置

现在剩下的就是创建 `src/index.js` 文件并尝试打包应用程序

src/index.js：

```
'use strict'

import React from 'react'
import { render } from 'react-dom'

const App = () => {
  return <h1>Hello world!</h1>
}

render(
  <App />,
document. getElementById('app')
)
```

该文件通常会将一个简单的 <h1>Hello world!</h1> 标题渲染到 id 为 'app' 的 html 标签中，但目前只需将代码转译一次即可。

$./node_modules/.bin/webpack . 将执行本地安装的 webpack 版本（如果你使用 -g 全局安装了 webpack，则使用 $webpack）

这将创建文件 out/bundle.js，里面包含转译后的代码，示例到此结束。

A.2 节：使用 webpack-dev-server

设置

在设置了一个简单的项目以使用 webpack、babel 和 react 后，执行 $npm i -g webpack-dev-server 将安装用于更快开发的开发 http 服务器。

修改 webpack.config.js

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

```
        exclude: /(node_modules)/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  }
};
```

This file tells webpack to start with the index.js file (assumed to be in src/) and convert it into a single bundle.js file in the `out` directory.

The `module` block tells webpack to test all files encountered against the regular expression and if they match, will invoke the specified loader. (`babel-loader` in this case) Furthermore, the `exclude` regex tells webpack to ignore this special loader for all modules in the `node_modules` folder, this helps speed up the transpilation process. Lastly, the `query` option tells webpack what parameters to pass to babel and is used to pass along the presets we installed earlier.

Testing the setup

All that is left now is to create the `src/index.js` file and try packing the application

src/index.js:

```
'use strict'

import React from 'react'
import { render } from 'react-dom'

const App = () => {
  return <h1>Hello world!</h1>
}

render(
  <App />,
  document. getElementById('app')
)
```

This file would normally render a simple `<h1>Hello world!</h1>` Header into the html tag with the id 'app', but for now it should be enough to transpile the code once.

`$./node_modules/.bin/webpack .` Will execute the locally installed version of webpack (use `$webpack` if you installed webpack globally with -g)

This should create the file `out/bundle.js` with the transpiled code inside and concludes the example.

Section A.2: Using webpack-dev-server

Setup

After setting up a simple project to use webpack, babel and react issuing `$npm i -g webpack-dev-server` will install the development http server for quicker development.

Modifying webpack.config.js

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

```
module.exports = {
  entry: './src/index.js',
  output: {
path: path.resolve(__dirname, 'out'),
    publicPath: '/public/',
filename: 'bundle.js'
  },
module: {
    loaders: [
      {
test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel',
query: {
presets: ['es2015', 'react']
        }
      }
    ]
  },
devServer: {
contentBase: path.resolve(__dirname, 'public'),
    hot: true
  }
};
```

修改内容在

- `output.publicPath` 设置了一个路径，用于从该路径提供我们的打包文件（详见Webpack配置文件获取更多信息）
- `devServer`
  - `contentBase` 用于提供静态文件的基础路径（例如`index.html`）
  - `hot` 设置webpack-dev-server在磁盘文件更改时进行热重载

最后我们只需要一个简单的index.html来测试我们的应用。

index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>React Sandbox</title>
  </head>
  <body>

    <div id="app" />

    <script src="public/bundle.js"></script>
  </body>
</html>
```

通过此设置运行 `$webpack-dev-server` 应该会在8080端口启动一个本地http服务器，连接后应该会渲染一个包含 `<h1>Hello world!</h1>` 的页面。

```
module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    publicPath: '/public/',
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  },
  devServer: {
    contentBase: path.resolve(__dirname, 'public'),
    hot: true
  }
};
```

The modifications are in

- `output.publicPath` which sets up a path to have our bundle be served from (see Webpack configuration files for more info)
- `devServer`
  - `contentBase` the base path to serve static files from (for example `index.html`)
  - `hot` sets the webpack-dev-server to hot reload when changes get made to files on disk

And finally we just need a simple index.html to test our app in.

index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>React Sandbox</title>
  </head>
  <body>

    <div id="app" />

    <script src="public/bundle.js"></script>
  </body>
</html>
```

With this setup running `$webpack-dev-server` should start a local http server on port 8080 and upon connecting should render a page containing a `<h1>Hello world!</h1>`.

附录B：React工具

B.1节：链接

查找React组件和库的地方；

- React组件目录
- JS.coach

Appendix B: React Tools

Section B.1: Links

Places to find React components and libraries;

- Catalog of React Components
- JS.coach

鸣谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供此内容，
更多更改可发送至 web@petercv.com 以发布或更新新内容

abhirathore2006	第10章
亚当	第16章
阿迪提亚·辛格	第17章
阿德里安·达拉š	第18章
艾哈迈德	第5章
akashrajkn	第二章
亚历克斯·杨	第一章和第四章
亚历山大	第四章
Alexg2195	第六章
阿努杰	第一章、第五章和第六章
阿隆	第25章
巴特·里奥丹	第1、2和12章
邦德	第二章
布拉德·科尔瑟斯特	第四章
布兰登·罗伯茨	第二章
brillout	第31章
查伊姆·弗里德曼	第22章
达克什·古普塔	第1章
丹尼洛·科瓦兰	第5章
大卫	第15章
丹尼斯·斯图肯	第27章和第28章
F.考德尔	第六章
法比安·舒尔茨	第14章
因素10	第5章
ghostffcode	第19章
詹卢卡·埃斯波西托	第1章
goldbullet	第二章
GordyD	第二章
hmnzr	第二章
伊南奇·古穆斯	第1章
ivarni	第二章
Jack7	第5章
贾加迪什·乌帕迪亚伊	第5章
杰森·伯恩	第14章
吉姆	第29章
吉米·吕	第5章和第20章
约翰·鲁德尔	第3章和第6章
jolyonruss	第1章和第2章
乔恩·陈	第1章和第2章
jonathangoodman	第二章
乔丹·亨德里克斯	第1章和第2章
胡安·德马尔科	第1章
贾斯塔巴兹	第二章
卡洛扬·科塞夫	第13章、第15章和第21章
库沙	第2章和第4章
莱昂纳多·博尔赫斯	第13章
莱昂	第三章

Credits

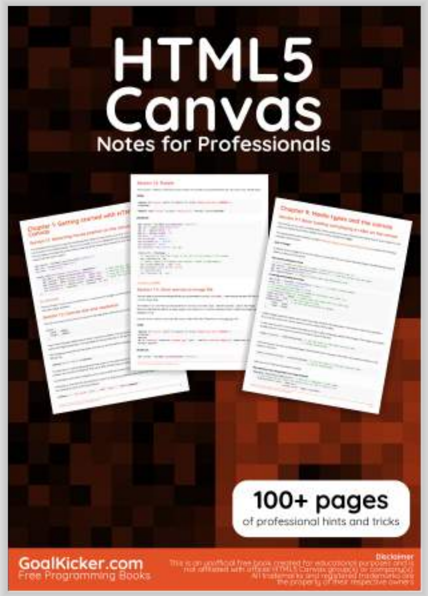
Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

abhirathore2006	Chapter 10
Adam	Chapter 16
Aditya Singh	Chapter 17
Adrián Daraš	Chapter 18
Ahmad	Chapter 5
akashrajkn	Chapter 2
Alex Young	Chapters 1 and 4
Alexander	Chapter 4
Alexg2195	Chapter 6
Anuj	Chapters 1, 5 and 6
Aron	Chapter 25
Bart Riordan	Chapters 1, 2 and 12
Bond	Chapter 2
Brad Colthurst	Chapter 4
Brandon Roberts	Chapter 2
brillout	Chapter 31
Chaim Friedman	Chapter 22
Daksh Gupta	Chapter 1
Danillo Corvalan	Chapter 5
David	Chapter 15
Dennis Stücken	Chapters 27 and 28
F. Kauder	Chapter 6
Fabian Schultz	Chapter 14
Faktor 10	Chapter 5
ghostffcode	Chapter 19
Gianluca Esposito	Chapter 1
goldbullet	Chapter 2
GordyD	Chapter 2
hmnzr	Chapter 2
Inanc Gumus	Chapter 1
ivarni	Chapter 2
Jack7	Chapter 5
Jagadish Upadhyay	Chapter 5
Jason Bourne	Chapter 14
Jim	Chapter 29
JimmyLv	Chapters 5 and 20
John Ruddell	Chapters 3 and 6
jolyonruss	Chapters 1 and 2
Jon Chan	Chapters 1 and 2
jonathangoodman	Chapter 2
JordanHendrix	Chapters 1 and 2
juandemarco	Chapter 1
justabuzz	Chapter 2
Kaloyan Kosev	Chapters 13, 15 and 21
Kousha	Chapters 2 and 4
leonardoborges	Chapter 13
Leone	Chapter 3

卢斯托伊科夫	第17章
马扬·格利克塞尔	第二章
马克·拉皮埃尔	第16章
马克斯·P·拉弗蒂	第1章和第5章
马扬克·舒克拉	第16章
麦格雷迪	第14章
Md Sifatul Islam	第1章
Md. Nahiduzzaman Rose	第1章
迈克尔·佩珀	第2章和第13章
米希尔	第8章
MMachinegun	第1章
m_callens	第二章
尼克·巴特利特	第1章
orvi	第1章
帕拉德·诺帕内	第8章
千月	第13章
QoP	第4、5和6章
拉贾布·沙基罗夫	第三章
随机用户	第11章
雷内·R	第30章
里法特	第20章
罗宾	第10章
rossipedia	第1章
萨尔曼·萨利姆	第六章
萨米·I.	第26章
谢尔吉·比希尔	第5章
沙宾·哈希姆	第1章
舒沃·哈比布	第9章
Simplans	第1章
sjmarshy	第二章
skav	第4章和第6章
sqzaman	第13章
桑尼·R·古普塔	第1章和第14章
thibmaek	第27章
蒂莫	第1、4、6和7章
user2314737	第1章
vintproykt	第23章和第24章
薇薇安	第六章
弗拉德·贝兹登	第二章
WitVault	第5章和第6章
扎克·布拉迪	第12章
扎卡里亚·里杜赫	第二章

lustoykov	Chapter 17
Maayan Glikser	Chapter 2
Mark Lapierre	Chapter 16
MaxPRafferty	Chapters 1 and 5
Mayank Shukla	Chapter 16
McGrady	Chapter 14
Md Sifatul Islam	Chapter 1
Md. Nahiduzzaman Rose	Chapter 1
Michael Peyper	Chapters 2 and 13
Mihir	Chapter 8
MMachinegun	Chapter 1
m_callens	Chapter 2
Nick Bartlett	Chapter 1
orvi	Chapter 1
parlad neupane	Chapter 8
Qianyue	Chapter 13
QoP	Chapters 4, 5 and 6
Rajab Shakirov	Chapter 3
Random User	Chapter 11
Rene R	Chapter 30
Rifat	Chapter 20
Robeen	Chapter 10
rossipedia	Chapter 1
Salman Saleem	Chapter 6
Sammy I.	Chapter 26
Sergii Bishyr	Chapter 5
Shabin Hashim	Chapter 1
Shuvo Habib	Chapter 9
Simplans	Chapter 1
sjmarshy	Chapter 2
skav	Chapters 4 and 6
sqzaman	Chapter 13
Sunny R Gupta	Chapters 1 and 14
thibmaek	Chapter 27
Timo	Chapters 1, 4, 6 and 7
user2314737	Chapter 1
vintproykt	Chapters 23 and 24
Vivian	Chapter 6
Vlad Bezden	Chapter 2
WitVault	Chapters 5 and 6
Zac Braddy	Chapter 12
Zakaria Ridouh	Chapter 2

你可能也喜欢



You may also like