# 目錄



# Preparation



# React



# Redux



# react-router

# 《从零学习 React 技术栈·理论篇》教程简介

> Hello大家好，我是余博伦，在接下来的一段时间里，由我和大家从零开始共同学习React技术栈的相关知识。教程将会以连载的形式发布在我的个人博客和微信公众号上，以文字教程为主，辅以一些代码示例供同学们参考，在连载结束之后，我会将所有内容整理为电子书提供下载。连载时教程会在每日早晨由公众号推送，同时为了方便一些外链和代码示例，可以在本博客查看教程。
>
> 作者：余博伦
>
> 教程谢绝任何形式转载。

React 是由 Facebook 主导开发的一个 JavaScript 框架。其实它和之前一些流行过的 MVVM 框架，例如 Angular 不同，React 主要只专注于 MVC 中的 V，也就是视图层。React 是当前最流行的，专门用来构建前端UI界面的框架。

React 的优点是它很快、很轻。并且它组件化的思想在开发构建界面时也对我们有很大的帮助，React 风格的代码，在我们学过之后就会了解到，它这种很规范化的写法，在一个需要共同开发协作的项目组或团队中，也是非常有益处的。

就好像100个人写 jQuery 就可能会有100种写法，但是不管让谁来写 React 组件，我们都能保证他写出来的代码和标准是差不多的。

所以假如你对 JavaScript 已经有了相当的掌握，想要学习框架来开发一些Web应用的话，选择 React 准是没错的。

现在的 React 已经不仅仅是一个框架，它逐步发展成了一个非常庞大的生态体系。

本教程会介绍React全家桶当中最为通用流行的一些框架和库，内容主要涉及到 React/React-router/Redux 这几个库，通过学习呢，在教程的结尾我们将能够开发完成一个利用上述三个工具库实现的 TodoList 应用。

现在你去看React官网文档，或者一些比较新的 React 教程，我们在书写 JavaScript 的时候呢，都已经开始采用 ES6 的语法，这些语法乍看起来可能会比较陌生。不过实际上，ES6实现的语法糖和一些新的方法 是能够极大地方便我们的编码的。

教程使用的是目前React发行的最新版本(v15.6.1，同时 v15.5.* 之后的版本的代码也是适用于React@16的)新版本当中引入了非常多的 ES6 的新特性，所以代码看起来和我们以前习惯的 jQuery 风格代码会有很多的不同。

在正式学习 React 技术栈之前，推荐你最好掌握一些 ES6 的基本语法，例如箭头函数、Class类、let、const等一些新的声明变量的方法等等，当然如果你不了解也别担心，在接下来的教程当中，每当涉及到这些新的语法时，我也会稍作简介。

**JavaScript** 参考文档

# 教程主要内容介绍

教程主要会分为六个部分进行讲解，在简要的基础知识准备和开发环境搭建之后，我们会分别对React/Redux/react-router的关键知识点进行学习，之后我们还会介绍到如何在React应用中编写样式，在最后一个部分，我们将一同实现一个运用上述React技术栈实现的Todolist应用。

在第一部分我们将会一同学习本教程的主角，也就是如何使用React。在这一部分当中，我们不光会介绍如何编码，也会讲解React当中一些关键的原理，以及在开发过程中运用到的 的一些最佳实践等内容。

接下来我们会一同学习Redux的使用，首先我会帮助大家理解状态管理到底是一个什么样的概念，之后会依次介绍redux当中几个关键的概念如何实现，如何编写以及如何运用。

再然后呢，我们会介绍到前端路由的概念。react技术栈当中，有一个非常棒的库，react-router来实现前端路由的功能，比较特殊的是，这个库目前以及发行到了4.0版本，而之前的版本也都在维护，各个大版本之间有比较多的变化，一些比较细节的内容，即使是它的官方文档也没有写清楚。我在这里呢，会着重教给大家最新版的使用方法，以及一些官方文档都没有提到的配置方法和小技巧。

我们还会学习到如何在 React 应用中开发编写样式，在这里部分，我会介绍几种比较主流流行的解决方案供大家参考，你可以在日后正式的工作当中，自行选取你觉得最为合适的开发样式的方法。

在教程的最后一个部分，我们将会学习到开发Rect应用的一般思路，以及利用之前学过的知识一同实现一个TodoList应用。

有的同学呢可能还会有一些疑问，比方说，现在还有一些别的很流行的框架，为什么我非得选 React 不可呢？

首先，React 相较于其他框架，其生态圈发展最为完整成熟，有非常多现成的完整的解决方案。并且它适用于大中型应用的开发，便于团队中多人之间协作，很多大厂就在正式的项目中使用了 React，并且学会 React 之后，你的能力将不止局限于浏览器之中，React还可以拓宽到开发原生应用，以及刚刚发布的ReactVR虚拟现实，甚至是物联网等各个领域。

# 为什么会有这个系列教程？

互联网上什么都有，杂乱无章。信息太多，相当于没有信息，选择太多，相当于没有选择。React 的中文资源比较少，大多数都已经过时，使用的是一两年前的版本，跟不上官方的版本更迭，且有一些中文资料由于翻译的不准确存在一些知识性的错误，很有可能会误导初学

者。

中文的学习资源还是太少，而且良莠不齐。国内前端学习者普遍英文水平还不够，况且现在前端发展这么快，等学好英语考过四六级，说不定 React 已经过气了。

另外，网上还没有综合 React 技术栈所有库的最新版本的教程和代码示例。一些教程虽然非常优秀，但随着 React 及相关库的新版本发布，这些教程已经过期，甚至提供的示例代码已经不能正常运行了。

# 本教程相较其他**React**学习资源的优点在哪里？

我在准备教程的过程中查阅大量资料，综合了国内外所有优秀的 React 学习资源，萃取其中最精华的知识点，选择最为流行的 React 技术栈，立足最新版本的官网文档，在帮助新手入门上手的同时，也会对一些重要的知识概念进行讲解，满足初级、中级各个学习阶段和水平的同学。

全部采用当前发布的正式版本库进行教学，确保我用起来是这个样子，你学完之后用起来也是这个样子。

# 本教程的前置知识

想要学习本教程的同学最好对 JavaScript 基础知识、ES6 新特性等相关内容掌握了解。熟悉 JavaScript 中变量、对象、函数等基本概念，ES6 中const/class/箭头函数/解构赋值等新语法的基本使用。本教程 90% 的内容为 JavaScript 相关知识，学习者仅需熟悉基础的 HTML/CSS 即可。

# 本教程使用的框架版本及软件依赖

框架及库

- react@15.6.1
- redux@3.7.2
- react-router@4.2.0

软件及工具

- npm/cnpm
- webpack
- create-react-app
- codepen/codepan
- vs code

- chrome
- VS Code React技术栈代码补全插件

代码示例

- React on Codepen
- react-router on Codepen
- Redux on Codepen
- React Demo Project on Github

学习资源

- React官方中文文档
- 余博伦个人博客
- 知乎专栏·从零学习React技术栈
- 知乎专栏·LeanReact
- 知乎专栏·pure render

# 1-1-React 新版本及 ES6 简介

1. 课程简介
2. Getting Started变化 JSXTransformer Create-react-app
3. React语法变化 Component
4. ES6基本语法 import const Class Arrow Function

第一节课呢，我们就来简单介绍一下这些应用在React开发当中，属于 ES6 的新的关键字和语法糖，也好为我们之后的学习打下一个基础，当然我更推荐同学们对 ES6 有一个比较全面的了解之后再开始学习 React，不过你也不需要担心，以后在课程中每当遇到涉及 ES6 新语法的问题时，我都会稍作讲解。

**ECMAScript 6 入门**

# import

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

我们就从头开始看。首先是在JS当中引用其他库文件的语法变化。之前我们一般都是通过 `require` 方法把库文件导出的方法保存在一个变量中。在ES6当中引入了一组个新的关键字 `import/export`，如果有同学对 Java 或 Python 有所了解的话，对 `import` 语句应该不会感到陌生，一般我们都会在文件的开头引入我们需要使用的模块或方法。

我们在一个文件中导入的模块或方法是从另一个文件中导出的。如果是使用 `export default` 语句导出的方法，我们直接定义其变量名称，这样的方法每个文件只能导出一个：

```
// myFunction.js
export default function() {
    console.log('This is default function!');
}
// index.js
import myFunction from 'myFunction';
```

仅用 `export` 导出的方法，在使用时，则需要把它们包含在大括号里：

```
// myAnotherFunction.js
export const foo = 'bar';
export function bar() {
    console.log('foo');
}
// index.js
import { foo, bar } from 'myAnotherFunction';
```

# const

```
const title = <h1>React Learning</h1>
```

`const` 关键字在ES6语法中，被用来声明常量。不过这并不表示声明的常量中数据不可变。在es6中，`const` 声明的其实是一个只读的指针，也即是指针的位置不能改变，但其指向的值事实上是可以操作的，我们来看一下下面这个例子：

```
// 我们先来定义一个常量空数组 a
const a = [];
// 输出空数组的内容
console.log(a);
// 返回结果 []
// 通过下面的操作可以向数组添加内容
a.push(2);
// 输出添加了元素之后的内容
console.log(a);
// 返回结果 [2]
// 但假如我想要将 a 指向一个新的数组则会报错
a = [];
// Uncaught TypeError: Assignment to constant variable.
```

上述代码你可以打开 Chrome 控制台自己输入测试体验一下。

我们用JSX创建的元素一般是不变的，所以通过 `const` 关键字来声明一个 React 的元素，而不是我们以往经常使用的 `var`

# class

在 React 之前的版本当中，我们一般使用一个名为 `createClass` 的方法来声明一个组件（现在这一API已经移除，如果非要使用的话需要单独引入名为 create-react-class 的包）。而在 ES6 当中，我们可以通过 class 关键字继承 React组件 的原型来创建组件。

在使用 props 或 state 的组件中，我们都会看到一个名为 constructor 的方法，这个方法中一般都会调用一个叫做 super 的关键字。这里的 super 其实是为了调用其继承父类的构造函数，由此将子类的 this 初始化，这样我们才能够在后面的代码中调用 this.props 或者 this.state

这么讲很多同学可能还是不太明白，不如我们直接来做一下实验：

```
class Animal {
  constructor(name) {
    this.nickName = name
  }
}

class Cat extends Animal {
  constructor(name) {
    super(name)
    document.write(this.nickName)
  }
}

const myCat = new Cat('Tom')
// 这时我们就可以在子类 Cat 中访问父类 Animal 的 nickName 属性。你可以用在线编辑器注释掉 super 方法来测试它带来的影响。
```

在 Codepen 上试试

# 箭头函数

箭头函数arrow function 是 ES6 中很棒的一个新特性，可以让你少写很多代码。从此在代码中几乎不用再拼写 function 了（当然并不是说 arrow function 适用于所有场景）。在 React 当中使用箭头函数还有别的功效，我们来看一个简单的计数器的例子：

```
class IncrementWidget extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      num: 0
    }
    // 我们可以通过 bind 方法手动绑定 this
    // this.handleClick = this.handleClick.bind(this)
  }
  handleClick() {
    this.setState({
      num: this.state.num + 1
    })
  }
  // 我们也可以直接在声明事件处理函数的时候使用箭头函数语法绑定 this
  /*
  handleClick = () => {
    this.setState({
      num: this.state.num + 1
    })
  }
  */
  render() {
    return (
      <div>
        <p>{this.state.num}</p>
        {/* 因为是匿名函数，所以会导致每次更新时组件都会重新渲染，这样写代码最简洁但是对性能有影响。
*/}
        <button onClick={() => this.handleClick()}>Add</button>
      </div>
    )
  }
}

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

### 在 Codepen 上试试

假如不使用 bind 我们会发现这一段代码是无法正常工作的，但是采用之前的 createClass 方法则不会，这是因为在 ES5 语法中，React 已经默认为所有方法绑定了 this，而在用 class 声明的组件中则不会有这种默认绑定。

给所有的事件处理函数加上 `bind(this)` 确实很不爽。

这时候箭头函数就派上了用场，只需要按照上述示例当中的语法书写，this 也就会被自动绑定到函数中去啦。方法任选其一，至于你在实际的项目当中想要怎么写还是看你的个人习惯或者是项目需求了。

# 总结

最后让我们来复习一下本节课所学的内容。

首先，我们介绍了 import 和 export 语句，在实际的开发过程中，我们肯定不能把所有的代码都写在一个文件当中，在 ES6 中新加入的 import 和 export 可以方便我们很好地组织引用代码的各个模块。

然后我们介绍了 const 关键字，一种新的定义常量的方法，常量事实上定义的是一个只读的指针，假如我们指向的是一个数组或者对象，还可以通过一些方法例如 array.push 操作其中的数据。

接下来我们介绍了 JS 当中的 class 类的实现，通过 class 来声明一个 React组件 和之前通过 createClass 方法创建组件有很多区别，我们又对 constructor 方法和 super 关键字稍做了解释，super 主要是用来调用父类的构造函数，以使得我们在子类中能够正确地获取 this

最后我们介绍了 arrow function 箭头函数，这其实相当于是一种实现函数缩写的语法糖，可以很方便地让我们少写很多代码。另外在 react 组件中使用箭头函数来声明事件处理方法也有实现自动绑定 `this` 的功能。

# 2-npm 的安装配置说明

# npm 的安装配置

工欲善其事，必先利其器。在正式学习React技术栈之前，我们先来介绍一下之后经常要使用到的npm.

npm是一个基于nodejs的JavaScript包管理工具，全称叫做node package manager，所谓的包呢，其实就是可复用的代码，每个人都可以把自己编写的代码库发布到npm的源，英文叫做registry，上面呢进行管理，你也可以下载别人开发好的包，在你自己的应用当中使用。

我们所熟知的，jQuery/Bootstrap/React等框架或库都被托管在npm上。通过使用npm作为项目的包管理工具，我们可以很方便地在我们的开发项目中引入以及管理第三方的框架或者库，而不需要像以前前端开发的原始时期一样，手动复制粘贴代码文件。

其实npm的安装以及基本的使用方法都非常的简单，只是由于国内糟糕的网络原因，在安装完成之后，我们还需要手动进行一些调整和配置，另外一方面也是照顾到初学者，所以在课程正式开始之前，我们先专门介绍一下npm的安装以及如何通过npm来管理我们的React开发项目。

npm的安装非常简单，不管你是用的是什么操作系统，我们只需要打开nodejs官方，网站会自动匹配你的系统显示相应的安装包，点击最新版本的下载按钮，等待安装包下载完成。

之后只需要双击打开安装包，稍等待一段时间，具体等待时间的长短和你的网速及硬件配置有关，等待安装程序预配置完成之后，根据提示，点击下一步，同意用户协议，再不停地点击下一步，在安装内容当中，确保配置环境变量的选项是被勾选中的，之后再确定进入下一步，等待安装完成即可。

之后我们可以打开控制台，输入npm或node检验是否安装成功。附加version参数可以查看我们安装的npm及node的版本。

由于npm官方的服务器在国外，在国内使用可能会遇到很多网络问题，而且速度也非常慢，为了方便我们的开发，我们需要手动切换npm到国内的镜像源。国内最稳定的镜像源是淘宝提供的。

打开淘宝npm镜像的官网，我们可以看到简要的使用说明。首先，在这里，我们可以获取到镜像的地址，先复制备用。之后打开命令行，注意到这里，因为我们的npm默认是安装在系统文件夹下的，所以需要以管理员模式打开命令行，否则在安装包的过程中可能会出现一些权限问题。

在命令行中，我们输入

```
npm config set registry https://registry.npm.taobao.org
```

来修改npm默认的安装源，通过

```
npm config get registry
```

来检验一下刚才的配置是否成功。

除此之外，我们还可以使用淘宝镜像提供的 `cnpm` 工具，通过 `cnpm` 来安装包一般速度会更快一些，我们可以直接复制文档中的命令：

```
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

粘贴在命令行中，回车进行安装，初次安装需要等待的时间可能会比较久，一定要有耐心。

等待安装完成之后，我们可以在命令行输入 `cnpm -v` 来测试是否安装成功。

# 使用npm安装react

接下来，我们可以尝试一下，使用npm安装React到我们的项目中。在我们的工作目录，按住shift点击鼠标右键，在弹出的菜单中选取“在此处打开命令行”，创建一个项目的文件夹，例如learn-react，使用cd命令切换至文件夹中，输入：

```
npm init -y
```

使用默认设置初始化我们的项目的npm配置，在项目目录中会生成一个package.json文件，里面会保存我们项目的基本信息、命令脚本以及依赖的库等信息。再然后，我们可以通过命令：

```
cnpm install react react-dom --save
```

来安装react，并将其保存到我们的项目依赖当中。现在再来看一下package.json文件，我们可以看到其中的dependencies已经保存了react的信息了。

# yarn

其实还有另外一个包管理工具yarn，yarn是由facebook推出的包管理客户端，优于npm客户端的是它会缓存已经下载过的包并做了一些其他方面的优化，速度要比npm快，还添加了一些别的npm不具备的特性。yarn的官网及文档全部都有中文，对用户十分友好，有兴趣的同学可

以去尝试，不过在本教程当中，我们仍然会继续使用npm作为我们的包管理工具。

# 1-3-React 开发环境配置

如果我们使用 JSX 语法或 ES6 的新特性，我们编写的 React 代码是无法直接在浏览器中运行的，所以我们需要为 React 专门配置开发环境，用来将 React 代码编译为可以直接在浏览器中运行的代码，或者通过一些工具和库在浏览器中处理我们编写的 React 代码，以使其正确地运行。

## 体验 React

假如你只是想体验一下 React 的话，最快且最简单的方式是使用 React 官方提供的 Codepen 模板，只需要打开一个网址即可在浏览器中体验编写 React 代码。同样是由于国内的网络原因，部分同学使用Codepen的体验可能不会特别好，没有关系，再介绍给大家一个替代品，codepan，也只需要打开一个网址：

https://codepan.net/boilerplate/react

就可以开始愉快地编码啦，并且这个网站还是可以离线运行的单页应用。

## 使用 React CDN

我们也可以不使用包管理工具，直接在网页中引入 React ，官方同样为我们提供了一个体验 React 的网页模板。

## 使用 create-react-app 命令行工具

通过npm，我们可以安装许多命令行工具。 React 官方专门为我们准备了专用的 React 项目生成工具create- react -app，只需要简单几行代码即可生成 React 项目，并且在开发时还支持实时更新，自动重载等功能。

如果是我们完全地手工配置，则需要配置安装webpack/babel等工具库。所以对于初学者或想要快速开发应用的同学，create- react -app就是你最好的选择。

并且在之后的课程当中，我们同样会使用create- react -app来创建我们的教学示例应用。

# 2-1-JSX 简介

JSX 其是一个语法扩展，它既不是单纯的字符串，也不是 HTML，虽然长得和 HTML 很像甚至基本上看起来一样。但事实上它是 React 内部实现的一种，允许我们直接在 JS 里书写 UI 的方式。

有些同学来看 JSX 可能也会觉得它像一种模板语言之类的。事实上也不是，它就是基于 JavaScript，在 React 当中的一种语法扩展的实现。

JSX 被用来创建 React 当中的 Elements，React 当中的元素。然后 React 再通过一些方法，把 JSX 创建的元素，渲染成我们在浏览器当中看到的 DOM元素。

在正式介绍 JSX 的相关知识前，我再强调几遍：

- JSX 不是 HTML
- JSX 不是 HTML
- JSX 不是 HTML

JSX 尽管使用了非常多的类似 HTML 的标签语法，但 JSX 只是一种在 JavaScript 当中书写 UI 的实现，所以你在理解的时候不要把 HTML 的相关知识先入为主地代入。不要问为什么可以这样写，怎么能这么写，怎么和 HTML 不一样？一类的问题，因为本来就不是同一种东西，另外也不要觉得 JSX 奇怪，不要觉得不适应，你写写就适应了。而且相信我，JSX 已经是书写 UI 最高效还保持了良好可读性的一种实现了。

另外 JSX 也有一些和 HTML 语法的差异，本篇教程的内容不可能涵盖全部，但以后出现在教程代码中的特例我都会强调说明。

# JSX实现原理

我们想要在浏览器里直接运行采用 JSX 语法的 JavaScript 显然暂时是不可能实现的，在实际的生产过程中，我们需要利用 Babel 一类的转译器来将我们的 JSX 语法或者 ES6 语法转译成浏览器可以直接运行的 JavaScript，事实上 JSX 在经过转译之后，会变成 React 创建 Element 的一个方法：

```
ReactDOM.render(
    <p>Hello world!</p>,
    document.getElementById('container')
)
```

转译之后就会变成下面这样：

```
// 事实上你书写的所有标签语法最后都会被转换成创建元素的 JS 方法。
ReactDOM.render(
    React.createElement('p',null,`Hello world!`),
    document.getElementById('container')
)
```

# JSX 基本语法

在本地配置 React 的开发环境是相对来说比较复杂的操作。为了让同学们尽快上手，我们一开始还是在在线代码编辑器里编写我们的代码。

比如我们可以使用 React 官方推荐的 Codepen

打开 Codepen 网站，当然为了方便你日后的学习，保存你的代码，最好注册一个账号。（注册理所当然需要 Google 人机验证等步骤，所以请自备梯子，学习编程这方面的技能早晚要掌握，如果你怎么都打不开这个网站或者怎么都注册不好，可以先放过，老老实实使用大清局域网）。

之后我们新建一个 pen，然后在 JS 编辑窗口的设置面板里添加 react 和 react-dom 两个库，再将 preprocessor 预处理器设置成 Babel，确定保存，就可以开始愉快地编码啦！

- Codepen 在线代码示例

当然，如果你的网速较慢或者无法顺利访问到国际网络，还可以使用离线Web应用 Codepan

- 使用 Codepan

## JSX 元素

```
const title = <h1>React Learning</h1>
```

我们用JSX创建的元素对象一般来说是不变的，所以通过 `const` 关键字来声明一个 React元素，而不是我们以往经常使用的 `var`

为了能让我们的 JSX 元素在页面上渲染出来预览查看，我们还需要添加两段代码：

在 HTML 窗格里添加：

```
<div id="root"></div>
```

在JS窗格的最底部添加：

```
// 这便是将 JSX元素渲染成 DOM 的方法
ReactDOM.render(title,document.getElementById('root'))
```

我们通过 ReactDOM 的 render 方法，将 title 元素渲染至 id 为 root 的页面容器当中。

## JSX 属性

JSX 的标签同样可以拥有自己的属性：

```
const title = <h1 id="main">React Learning</h1>
```

但它和 HTML 又不是完全相同的，例如我们想要为 JSX标签添加 class 的时候需要：

```
// 注意是 className 而不是 class
const title = <h1 className="main">React Learning</h1>
```

所有支持的 HTML属性在这里可以查阅。

## JSX 嵌套

JSX 的标签也可以像 HTML 一样相互嵌套，一般有嵌套解构的 JSX 元素外面，我们习惯于为它加上一个小括号：

```
const title = (
    <div>
        <h1 className="main">React Learning</h1>
        <p>Let's learn JSX</p>
    </div>
)
```

需要注意的是，JSX 在嵌套时，最外层有且只能有一个标签，否则就会出错获：

```
// 这是一个错误示例
const title = (
    <h1 className="main">React Learning</h1>
    <p>Let's learn JSX</p>
)
```

## JSX表达式

在 JSX 元素中，我们同样可以使用 JavaScript 表达式，在 JSX 当中的表达式需要用一个大括号括起来：

```
function sayhi(name) {
  return 'Hi,' + name
}

const title = (
    <div>
        <h1 className="main">React Learning</h1>
        <p>Let's learn JSX. <span>{sayhi('you')}</span></p>
    </div>
)
```

好了，有关 JSX 的基础知识，我们掌握到这里就差不多了。在之后的课程中，随着我们学习的进一步深入，也会在需要的时候介绍更深入的有关 JSX 的知识。

# 2-2-组件类型

这一节的内容会比较多，如果是刚刚入门的新同学一时半会儿可能会接受不了，而且基本都属于理论知识。如果你在阅读时发现理解有困难也不需要灰心，可以把本篇教程当作随时可供查阅的文档，等到你在实践中积累了一定的代码量之后再回过头来阅读文章就会感觉非常轻松啦。

- 元素与组件Element & Component
- 函数定义与类定义组件Functional & Class
- 展示与容器组件Presentational & Container
- 有状态与无状态组件Stateful & Stateless
- 受控与非受控组件Controlled & Uncontrolled
- 组合与继承Composition & Inheritance

## 元素与组件 Element & Component

元素

元素是构建React应用的最小单位。元素所描述的也就是你在浏览器中能够看到的东西。根据我们在上节课中讲到的内容，我们在编写React代码时一般用JSX来描述React元素。

在作用上，我们可以把React元素理解为DOM元素；但实际上，React元素只是JS当中普通的对象。React内部实现了一套叫做React DOM的东西，或者我们称之为Virtual DOM也就是虚拟DOM.通过一个树状结构的JS对象来模拟DOM树。

说到这里我们可以稍微讲一下，React为什么会有这一层虚拟DOM呢？在课程介绍中我们曾经提到过，React很快、很轻。它之所以快就是因为这一套虚拟DOM的存在，React内部还实现了一个低复杂度高效率的Diff算法，不同于以往框架，例如angular使用的脏检查。在应用的数据改变之后，React会尽力少地比较，然后根据虚拟DOM只改变真实DOM中需要被改变的部分。React也藉此实现了它的高效率，高性能。

当然这不是虚拟DOM唯一的意义，通过这一层单独抽象的逻辑让React有了无限的可能，就比如react native的实现，可以让你只掌握JS的知识也能在其他平台系统上开发应用，而不只是写网页，甚至是之后会出现的React VR或者React物联网等等别的实现。

话说回来，元素也就是React DOM之中描述UI界面的最小单位。刚才我们说到了，元素其实就是普通的JS对象。不过我们用JSX来描述React元素在理解上可能有些困难，事实上，我们也可以不使用JSX来描述：

```
const element = <h1>Hello, world</h1>
// 用JSX描述就相当于是调用React的方法创建了一个对象
const element = React.createElement('h1', null, 'Hello, world')
```

组件

要注意到，在React当中元素和组件是两个不同的概念，之所以在前面讲了这么多，就是担心大家不小心会混淆这两个概念。首先我们需要明确的是，组件是构建在元素的基础之上的。

React官方对组件的定义呢，是指在UI界面中，可以被独立划分的、可复用的、独立的模块。其实就类似于JS当中对function函数的定义，它一般会接收一个名为props的输入，然后返回相应的React元素，再交给ReactDOM，最后渲染到屏幕上。

# 函数定义与类定义组件 Functional & Class

新版本的React里提供了两种定义组件的方法。当然之前的React.createClass也可以继续用，不过我们在这里先不纳入我们讨论的范围。

第一种函数定义组件，非常简单啦，我们只需要定义一个接收props传值，返回React元素的方法即可：

```
function Title(props) {
  return <h1>Hello, {props.name}</h1>
}
```

甚至使用ES6的箭头函数简写之后可以变成这样：

```
const Title = props => <h1>Hello, {props.name}</h1>
```

第二种是类定义组件，也就是使用ES6中新引入的类的概念来定义React组件：

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

之后呢，根据我们在上一节课中了解到的，组件在定义好之后，可以通过JSX描述的方式被引用，组件之间也可以相互嵌套和组合。

# 展示与容器组件 Presentational & Container

接下来我们还会介绍一些更深入的关于组件概念，现在听起来可能会比较抽象枯燥，不过接下来要介绍的这几个概念在之后的课程中都是会被应用到的，同学们也可以根据自己的实际情况，在学习完后续的课程之后，再返回来听听看，相信一定会对你理解React有所帮助。

首先是最重要的一组概念：展示组件与容器组件。同样，在课程介绍中我们提到的，React并不是传统的MVVM框架，它只是在V层，视图层上下功夫。同学们应该对MVVM或MVC都有所了解，那么既然我们的框架现在只有V层的话，在实际开发中应该如何处理数据与视图的关系呢？

为了解决React只有V层的这个问题，更好地区分我们的代码逻辑，展示组件与容器组件这一对概念就被引入了。这同样也是我们在开发React应用时的最佳实践。

我们还是先看一个具体的例子来解释这两个概念：

```
class CommentList extends React.Component {
  constructor(props) {
    super(props)
    this.state = { comments: [] }
  }

  componentDidMount() {
    $.ajax({
      url: "/my-comments.json",
      dataType: 'json',
      success: function(comments) {
        this.setState({comments: comments})
      }.bind(this)
    })
  }

  renderComment({body, author}) {
    return <li>{body}—{author}</li>
  }

  render() {
    return <ul> {this.state.comments.map(this.renderComment)} </ul>
  }

}
```

这是一个回复列表组件，乍看上去很正常也很合理。但实际上在开发React应用时，我们应该避免写出这样的组件，因为这类组件担负的功能太多了。它只是一个单一的组件，但需要同时负责初始化state，通过ajax获取服务器数据，渲染列表内容，在实际应用中，可能还会有更多的功能依赖。这样，在后续维护的时候，不管是我们要修改服务器数据交互还是列表样式内容，都需要去修改同一个组件，逻辑严重耦合，多个功能在同一个组件中维护也不利于团队协作。

通过应用展示组件与容器组件的概念，我们可以把上述的单一组件重构为一个展示回复列表组件和回复列表容器：

```
// 展示组件

class CommentList extends React.Component {
  constructor(props) {
    super(props)
  }

  renderComment({body, author}) {
    return <li>{body}—{author}</li>
  }

  render() {
    return <ul> {this.props.comments.map(this.renderComment)} </ul>
  }

}

// 容器组件

class CommentListContainer extends React.Component {
  constructor() {
    super()
    this.state = { comments: [] }
  }

  componentDidMount() {
    $.ajax({
      url: "/my-comments.json",
      dataType: 'json',
      success: function(comments) {
        this.setState({comments: comments})
      }.bind(this)
    })
  }

  render() {
    return <CommentList comments={this.state.comments} />
  }
}
```

像这样回复列表如何展示与如何获取回复数据的逻辑就被分离到两个组件当中了。我们再来明确一下展示组件和容器组件的概念：

展示组件

- 主要负责组件内容如何展示
- 从props接收父组件传递来的数据

- 大多数情况可以通过函数定义组件声明

容器组件

- 主要关注组件数据如何交互
- 拥有自身的state，从服务器获取数据，或与redux等其他数据处理模块协作
- 需要通过类定义组件声明，并包含生命周期函数和其他附加方法

那么这样写具体有什么好处呢？

- 解耦了界面和数据的逻辑
- 更好的可复用性，比如同一个回复列表展示组件可以套用不同数据源的容器组件
- 利于团队协作，一个人负责界面结构，一个人负责数据交互

# 有状态与无状态组件 Stateful & Stateless

有状态组件

意思是这个组件能够获取储存改变应用或组件本身的状态数据，在React当中也就是state，一些比较明显的特征是我们可以在这样的组件当中看到对this.state的初始化，或this.setState方法的调用等等。

无状态组件

这样的组件一般只接收来自其他组件的数据。一般这样的组件中只能看到对this.props的调用，通常可以用函数定义组件的方式声明。它本身不会掌握应用的状态数据，即使触发事件，也是通过事件处理函数传递到其他有状态组件当中再对state进行操作。

我们还是来看具体的例子比较能清楚地说明问题，与此同时，我们已经介绍了三组概念，为了防止混淆，我这里特意使用了两个展示组件来做示例，其中一个是有状态组件，另一个是无状态组件，也是为了证明，并不是所有的展示组件都是无状态组件，所有的容器组件都是有状态组件。再次强调一下，这是两组不同的概念，以及对组件不同角度的划分方式。

```
//Stateful Component
class StatefulLink extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      active: false
    }
  }
  handleClick() {
    this.setState({
      active: !this.state.active
    })
  }
  render() {
```

```
    return <a
          style={{ color: this.state.active ? 'red' : 'black' }}
          onClick={this.handleClick.bind(this)}
         >
           Stateful Link
         </a>
  }
}

// Stateless Component
class StatelessLink extends React.Component {
  constructor(props) {
    super(props)
  }
  handleClick() {
    this.props.handleClick(this.props.router)
  }
  render() {
    const active = this.props.activeRouter === this.props.router
    return (
        <li>
            <a
              style={{ color: active ? 'red' : 'black' }}
              onClick={this.handleClick.bind(this)}
             >
                Stateless Link
            </a>
    </li>
    )
  }
}

class Nav extends React.Component {
  constructor() {
    super()
    this.state={activeRouter: 'home'}
  }
  handleSwitch(router) {
    this.setState({activeRouter: router})
  }
  render() {
    return (
    <ul>
        <StatelessLink activeRouter={this.state.activeRouter} router='home' handleClic
k={this.handleSwitch.bind(this)} />
        <StatelessLink activeRouter={this.state.activeRouter} router='blog' handleClic
k={this.handleSwitch.bind(this)} />
        <StatelessLink activeRouter={this.state.activeRouter} router='about' handleCli
ck={this.handleSwitch.bind(this)} />
    </ul>
    )
  }
}
```

上述的例子可能稍有些复杂，事实上，在React的实际开发当中，我们编写的组件大部分都是无状态组件。毕竟React的主要作用是编写用户界面。再加上ES6的新特性，绝大多数的无状态组件都可以通过箭头函数简写成类似下面这样：

```
/* function SimpleButton(props) {
  return <button>{props.text}</button>
} */

const SimpleButton = props => <button>{props.text}</button>
```

# 受控与非受控组件 Controlled & Uncontrolled

受控组件

一般涉及到表单元素时我们才会使用这种分类方法，在后面一节课程表单及事件处理中我们还会再次谈论到这个话题。受控组件的值由props或state传入，用户在元素上交互或输入内容会引起应用state的改变。在state改变之后重新渲染组件，我们才能在页面中看到元素中值的变化，假如组件没有绑定事件处理函数改变state，用户的输入是不会起到任何效果的，这也就是“受控”的含义所在。

非受控组件

类似于传统的DOM表单控件，用户输入不会直接引起应用state的变化，我们也不会直接为非受控组件传入值。想要获取非受控组件，我们需要使用一个特殊的ref属性，同样也可以使用defaultValue属性来为其指定一次性的默认值。

我们还是来看具体的例子：

```
class ControlledInput extends React.Component {
  constructor() {
    super()
    this.state = {value: 'Please type here...'}
  }

  handleChange(event) {
    console.log('Controlled change:',event.target.value)
    this.setState({value: event.target.value})
  }

  render() {
    return (
      <label>
        Controlled Component:
        <input type="text"
               value={this.state.value}
               onChange={(e) => this.handleChange(e)}
        />
      </label>
    )
  }
}

class UncontrolledInput extends React.Component {
  constructor() {
    super()
  }

  handleChange() {
    console.log('Uncontrolled change:',this.input.value)
  }

  render() {
    return (
        <label>
          Uncontrolled Component:
          <input type="text"
                 defaultValue='Please type here...'
                 ref={(input) => this.input = input}
                 onChange={() =>this.handleChange()}
          />
        </label>
    )
  }
}
```

通常情况下，React当中所有的表单控件都需要是受控组件。但正如我们对受控组件的定义，想让受控组件正常工作，每一个受控组件我们都需要为其编写事件处理函数，有的时候确实会很烦人，比方说一个注册表单你需要写出所有验证姓名电话邮箱验证码的逻辑，当然也有

一些小技巧可以让同一个事件处理函数应用在多个表单组件上，但生产开发中并没有多大实际意义。更有可能我们是在对已有的项目进行重构，除了React之外还有一些别的库需要和表单交互，这时候使用非受控组件可能会更方便一些。

# 组合与继承 Composition & Inheritance

前面我们已经提到了，React当中的组件是通过嵌套或组合的方式实现组件代码复用的。通过props传值和组合使用组件几乎可以满足所有场景下的需求。这样也更符合组件化的理念，就好像使用互相嵌套的DOM元素一样使用React的组件，并不需要引入继承的概念。

当然也不是说我们的代码不能这么写，来看下面这个例子：

```
// Inheritance
class InheritedButton extends React.Component {
  constructor() {
    super()
    this.state = {
      color: 'red'
    }
  }
  render() {
    return (
    <button style={{backgroundColor: this.state.color}} class='react-button'>Inherited
 Button</button>
    )
  }
}

class BlueButton extends InheritedButton {
  constructor() {
    super()
    this.state = {
      color: '#0078e7'
    }
  }
}

// Composition
const CompositedButton = props => <button style={{backgroundColor:props.color}}>Compos
ited Button</button>

const YellowButton = () => <CompositedButton color='#ffeb3b' />
```

但继承的写法并不符合React的理念。在React当中props其实是非常强大的，props几乎可以传入任何东西，变量、函数、甚至是组件本身：

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  )
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  )
}
```

React官方也希望我们通过组合的方式来使用组件，如果你想实现一些非界面类型函数的复用，可以单独写在其他的模块当中在引入组件进行使用。

# 2-3-组件数据

今天我们要一起学习一下 React 组件当中对于数据的处理。

- props
- state
- context

# props

- 传入变量
- 传入函数
- 传入组件
- props.children

props其实就是属性Properties的缩写。

在形式上，props之于JSX就相当于attributes之于HTML。从写法上来看呢，我们为组件传入props就可以像为HTML标签添加属性一样：

**JSX**

```
const SimpleButton = props =>
    {/*<button style={{ color: props.color }}>Submit</button>*/}
    <button className={props.color}>Submit</button>

ReactDOM.render(<SimpleButton color="blue" />, document.getElementById('root'))
```

**HTML**

```
<button class="blue">Submit</button>
```

在概念上，props对于组件就相当于JS中参数之于函数。我们可以抽象出这样一个函数来解释：

```
Component(props) = View
```

组件函数接受props作为参数最后返回视图内容。props比起原生HTML的属性要强大很多。HTML的属性只能传递某些固定的字符，props几乎可以传递所有的内容，包括变量、函数、甚至是组件本身。

**props**是只读的

在React中，props都是自上向下传递，从父组件传入子组件。并且props是只读的，我们不能在组件中直接修改props的内容。也即是说组件只能根据传入的props渲染界面，而不能在其内部对props进行修改。

**props**类型检查

正是因为props的强大，什么类型的内容都可以传递，所以在开发过程中，为了避免错误类型的内容传入，我们可以为props添加类型检查。

**props**默认值

由于props是只读的，我们不能直接为props赋值。React专门准备了一个方法定义props的默认值。

```
import React from 'react'
import PropTypes from 'prop-types'

const Title = props => <h1>{props.title}</h1>

Title.defaultProps = {
  title: 'Wait for parent to pass props.'
}

Title.propTypes = {
  title: PropTypes.string.isRequired
}
```

# state

- 初始化
- setState方法
- 向下传递数据

state如果要翻译的话可以翻译为状态。一个组件可以有状态，就像我们之前介绍过的有状态组件，也可以是整个React应用的状态。

在React中state也是我们进行数据交互的地方，又或者叫做state management状态管理。一个应用需要进行数据交互，比如同服务器之间的交互，同用户输入进行交互。话反过来，从API获取数据，处理用户输入也就是我们需要用到state的时候。

下面我们通过一些例子来说明state的用法以及一些需要特别注意的地方。

先看一个最简单的计数器例子，在新版本的React当中，我们通过类定义组件来声明一个有状态组件，之后在它的构造方法中初始化组件的state，我们可以先赋予它默认值。

之后就可以在组件中通过this.state来访问它，既然是state那么肯定涉及到数据的改变，因此我们还需额外定义一个负责处理state变化的函数，这样的函数中一般都会包含this.setState这个方法。

要注意到，和之前的props一样，初始化state之后，如果我们想改变它，是不可以直接对其赋值的，直接修改state的值没有任何意义，因为这样的操作脱离了React运行的逻辑，不会触发组件的重新渲染。所以需要this.setState这个方法，在改变state的同时，触发React内部的一系列函数，最后在页面上重新渲染出组件。

```
class Counter extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      counter: 0
    }
  }

  addOne() {
    this.setState((prevState) =>({
      counter: prevState.counter + 1
    }))
  }

  render() {
    return (
      <div>
        <p>{ this.state.counter }</p>
        <button
          onClick={() => this.addOne()}>
          Increment
        </button>
      </div>
    )
  }
}
```

根据我们上面学习到的有关props和state的知识，我们再来一起看一个稍微复杂一些的示例：

```
const Title = props => <h1>{props.title}</h1>

const UserInfo = props => <p>{props.firstName + ' ' + props.lastName}</p>

class LikeButton extends React.Component {
  constructor(props) {
    super(props)
  }
  handleClick() {
    this.props.handleClick()
  }
  render() {
    return <button onClick={() => this.handleClick()}>{this.props.mark}</button>
  }
}

class App extends React.Component {
  constructor(props) {
    super(props)
    this.state = { mark: '☆' }
  }
  handleButtonClick() {
    const mark = (this.state.mark === '★') ? '☆' : '★'
    this.setState({ mark: mark })
  }
  render() {
    return (
    <div>
        <Title title='Do you like this guy?' />
        <UserInfo firstName='Justin' lastName='Bieber' />
        <LikeButton mark={this.state.mark} handleClick={() => this.handleButtonClick()
} />
    </div>
    )
  }
}

const Root = props => <div className='container'>{props.children}</div>

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

同学们要注意到，在这个例子中，我依旧沿袭了上节课中展示组件和容器组件的概念。只要我们乐意，所有的React组件都可以拥有自己的state，但在实际的生产中，我们写出的90%的都是无状态组件。假如state分散在应用的各个组件当中，有的负责后端数据交互，有的负责

处理用户输入，有的负责界面变换逻辑等等，后期维护起来是相当困难的。因此在开发React应用的过程中，我们应该尽量把state集中统一管理，通过props把state的数据传递到需要的组件当中。

比方说下面这个例子，在这个简易的货币汇率转换器中，3个组件，一个输入，两个输出都需要使用到同一个名为money的state，那么我们就需要为3个组件建立一个统一的父组件来管理他们公用的state，通过props将state中的值传递到子组件中，同样也可以将处理state的函数以props的方式传递下去，在子组件中触发父组件修改state的函数，以此来达到更改state重新渲染组件的目的。

```
class CurrencyConvertor extends React.Component {
  constructor() {
    super()
    this.state = {
      money : 100
    }
  }
  handleInputChange(event) {
    this.setState({
      money: event.target.value
    })
  }
  render() {
    return (
    <div>
        <h2>汇率转换</h2>
        <YuanInput money={this.state.money} handleChange={(e) => this.handleInputChang
e(e)} />
        <MoneyConvertor type='美元' unit='dollar' money={this.state.money} rate={0.1453
} />
        <MoneyConvertor type='日元' unit='yen' money={this.state.money} rate={16.1814}
/>
    </div>
    )
  }
}

class YuanInput extends React.Component {
  constructor(props) {
    super(props)
  }
  handleChange(event) {
    this.props.handleChange(event)
  }
  render() {
    return (
      <p>
        <label>
          人民币
          <input name='yuan' onChange={(e)=>this.handleChange(e)} value={this.props.mo
```

```
ney} />
        </label>
      </p>
    )
  }
}

const MoneyConvertor = props => (
  <p><label>
    {props.type}
    <input name={props.unit} value={(props.money*props.rate).toFixed(2)} disbled />
  </label></p>
)
```

随着我们开发应用的逐步扩展，它的state会变得越来越庞大复杂，假如分散到各个组件当中，对于日后应用的维护者来说将是一个噩梦。怎么处理怎么储存应用的state非常值得我们深入去思考，由此也就引发了一个问题——状态管理。这也正是我们学习过React之后，在下一个部分的教程中将要介绍的Redux专注解决的问题。

# context

集中统一管理state也会引发另外一个问题，React当中的数据都是自上向下传递的，加入我们的state在最外层的一个组件中管理，但真正使用state中数据的组件可能在N层子组件的嵌套中，那么我们就需要在每层的子组件中都传递props下去，直至使用这个数据的子组件获取到为止，虽然可以让我们更清楚地看到数据流的传递和变化，但写起来真的非常痛苦。

React提供了一个名为context的实验性质的API来解决这一问题。我们直接来看这个对比props和context的例子：

```
const UserInfo = (props, context) => (
  <div className="UserInfo-name">
    {context.lastName + ', ' + props.firstName}
  </div>
)
UserInfo.contextTypes = {
  lastName: React.PropTypes.string
}

const Column = props => <div className='column'><UserInfo firstName={props.firstName}
/></div>

const Row = props => <div className='row'><Column firstName={props.firstName} /></div>

const Container = props => <div className='container'><Row firstName={props.firstName}
 /></div>

class App extends React.Component {
  constructor() {
    super()
    this.state = { firstName: 'Bolun'}
  }
  getChildContext() {
    return {lastName: "Yu"}
  }
  render() {
    return (
      <div className='App'>
        <Container firstName={this.state.firstName} />
      </div>
    )
  }
}
App.childContextTypes = {
  lastName: React.PropTypes.string
}
```

在 Codepen 上试试。

通过props我们需要传递4层才能将数据传递到我们的目标组件中，而利用context的特性，只需要在最外层的应用组件中定义，直接在目标组件中使用即可。我们通过组件中的特殊方法getChildContext来定义context，另外要注意的是，必须用childContextTypes和contextTypes各自声明context的类型，否则我们无法获取context中的数据，只会返回一个空对象。

另外在小项目（比方说 Todo/货币汇率转换器/RSS阅读器 这类数据源单一好理解好控制的这类应用我们就叫它小项目）中，你可以试着把 state 玩溜，在初具规模的项目中我们还有一些更完整的解决方案，在以后的教程当中会介绍给同学们。

# 2-4-组件生命周期

- React是如何渲染组件的
- 渲染组件时会触发的生命周期方法
  - 挂载流程
  - 更新流程
  - 卸载流程

# React是如何渲染组件的

我们就按照平时书写React代码的顺序来理清React把组件代码渲染到最终的真实DOM中的流程。

一般来讲，我们都会先定义一个组件。我们想要在页面中看到这个组件的渲染结果的化，就需要以JSX的形式将组件传入ReactDOM.render方法的第一个参数，我们要理解，这里的JSX经过React内部的转译，其实是一个React.createElement创建React元素的方法。render方法获取到React元素之后会将它实例化，之后它会根据实例化的React元素创建出真实的DOM元素，再根据第二个参数的指向，将创建好的元素插入到目标DOM容器当中。

为了加深理解，还是稍微来了解一下React本身运行流程。

在新版本的React当中，React的底层被重写了。React换上了一个新的引擎，这个引擎叫做React Fiber.React Fiber 作用的也即是React最核心的功能，它将React应用界面更新的过程分为了两个主要的部分：

- 调度过程
- 执行过程

前面还是一样，React对虚拟DOM进行Diff操作，对比应用前后改变的部分，之后进入调度阶段，React Fiber会计算出最优化的调度方法，以防止我们更新DOM是发生卡顿阻塞等状况，然后进入执行过程提交调度计算的结果，最终由ReactDOM负责将页面中的内容渲染到页面当中。

在调度过程中，有4个生命周期函数会被触发：

- componentWillMount
- componentWillReceiveProps
- shouldComponentUpdate
- componentWillUpdate

在执行过程中，有3个生命周期函数会被触发：

- componentDidMount
- componentDidUpdate
- componentWillUnmount

# React组件生命周期方法

React为了方便我们更好地控制自己的应用，提供了许多预置的生命周期方法。就好像我们上面介绍的挂载流程，这些固定的生命周期方法分别会在组件的挂载流程、更新流程、卸载流程中触发。

假如有同学实在是不理解上一个部分介绍的渲染原理也没有关系，在学习下面的内容之前，我们只需要明确，React的初次render会进行挂载流程，挂载之后的后续渲染进行的都是更新流程，最后，我们也可以通过ReactDOM.unmountComponentAtNode方法将组件卸载，这其中进行的就是卸载流程。

如果只是一直讲解概念，感觉会很抽象不好理解，所以我们不如来亲眼看一看这三个流程是如何运行，如何触发生命周期函数的：

```
class Number extends React.Component {
  constructor(props) {
    super(props)
    console.log('%cconstructor'+'%c 子组件已构造', 'font-weight:bold', 'color: blue')
  }

   componentWillMount() {
      console.group("%c React 挂载", 'color: #00d8ff')
      console.log('%ccomponentWillMount'+'%c 组件即将挂载', 'font-weight:bold', '')
   }

   componentDidMount() {
      console.log('%ccomponentDidMount'+'%c 组件已挂载', 'font-weight:bold', '')
      console.groupEnd();
   }

   componentWillReceiveProps(newProps) {
      console.group("%c React Updating", 'color: green')
      console.log('%ccomponentWillReceiveProps'+'%c 组件即将接收props', 'font-weight:bol
d', '')
      console.log('newProps', newProps.counter)
      console.log('this.props', this.props.counter)
   }

   shouldComponentUpdate(newProps, newState) {
      const result = true
      console.info('%cshouldComponentUpdate'+'%c 返回判断是否要更新组件' +`%c ${result}`,
'font-weight:bold', 'color: #ff3c41', 'font-weight:bold;color: #236fd4')
      if (!result) console.groupEnd()
      return result;
```

```
    }

    componentWillUpdate(nextProps, nextState) {
       console.log('%ccomponentWillUpdate'+'%c 组件即将更新', 'font-weight:bold', '')
       console.log('nextProps', nextProps.counter)
       console.log('this.props', this.props.counter)
    }

    componentDidUpdate(prevProps, prevState) {
       console.log('%ccomponentDidUpdate' + '%c 组件已更新', 'font-weight:bold', '')
       console.log('prevProps', prevProps.counter)
       console.log('this.props', this.props.counter)
       console.groupEnd();
    }

    componentWillUnmount() {
       console.group("%c React Unmounting", 'color: brown')
       console.log('%ccomponentWillUnmount'+'%c 组件即将卸载','font-weight:bold' , 'color
: gray')
       console.groupEnd();
    }

    render() {
       console.log('%crender'+'%c 组件渲染中...', 'font-weight:bold', '')
       console.log('this.props', this.props.counter)
       return <p>{ this.props.counter }</p>
    }
}

class Counter extends React.Component {
  constructor(props) {
    super(props)
    console.log('%cconstructor'+'%c 父组件已构造', 'font-weight:bold', 'color: #ae63e4')
    this.state = {
      counter: 0
    }
    console.log('this.state', this.state.counter)
  }

  addOne() {
    console.log('%caddOne()'+'%c 事件处理函数触发', 'font-weight:bold', '')
    console.log('prevState', this.state.counter)
    this.setState((prevState) =>({
      counter: prevState.counter + 1
    }))
  }

  unMount() {
    ReactDOM.unmountComponentAtNode(document.getElementById('root'));
  }

  update() {
    this.forceUpdate()
```

```
  }

  render() {
    console.log('%crender'+'%c 父组件渲染中...', 'font-weight:bold', '')
    console.log('nextState', this.state.counter)
    return (
      <div>
        <Number counter={this.state.counter} />
        <button
          onClick={() => this.addOne()}>
          增加
        </button>
        <button
          onClick={() => this.update()}>
          强制更新
        </button>
        <button
          onClick={() => this.unMount()}>
          卸载
        </button>
      </div>
    )
  }
}

const render = () => ReactDOM.render(
  <Counter />,
  document.getElementById('root')
)

document.getElementById('render').addEventListener('click',render)
```

这里我们定义了两个组件，父组件会传递state到子组件当中，顺便state的变化和props的传递流程我们也可以在这个示例中看到。子组件的每个生命周期函数中我们都在控制台输出了相关的信息，这样在它被触发的时候，我们在控制台中就能够观察到。

挂载流程

首先是组件初次渲染的挂载流程，我们通过一个按钮绑定渲染函数来手动触发组件的渲染。

初次挂载组件时，我们定义的组件类首先会被构造声明。在渲染之前会触发componentWillMount，渲染完成会触发componentDidMount这两个生命周期函数。

componentWillMount方法会在render之前被触发，只会在组件被渲染出来之前触发一次，如果你是使用ES6的Class声明组件的话，时机和作用几乎与constructor相同，在componentWillMount方法里对state定义或改变不会触发重新渲染。

componentDidMount在初次渲染完成之后被触发，也只会触发一次，在这个方法里你已经可以访问渲染出的DOM元素了。官方推荐在这个函数中进行一些例如ajax请求的操作，所以它也是我们最经常使用的生命周期函数。

更新流程

我们是在子组件中观察生命周期函数运行的，之前已经提到过了，这个示例中父组件会通过props向子组件传递state中的计数值，所以我们首先观察到的被触发的生命周期函数叫做componentWillReceiveProps这是在我们调用this.setState()方法之后，子组件收到新的props之后会触发的函数。值得提醒的一点是，正如这个函数命名中的Will，更新流程运行到这一步，组件的props还没有改变，我们可以通过nextProps获取到新的props，但如果我们试着查看this.props会发现它还是之前的值。

接下来被触发的shouldComponentUpdate是一个很特殊的函数，它默认会返回ture，但假如这个函数返回false的话，更新流程就会被跳过，render也不会继续被触发。假如你想要提升你应用的性能，可以在这个函数中自定义一些判断来跳过不需要被更新的组件。

然后通过实验我们也可以看到，组件在初次渲染流程或者使用forceUpdate方法时是不会触发这个函数的。

接下来的两个生命周期函数componentWillUpdate和componentDidUpdate分别会在render的前后被触发。

需要注意的是在componentWillUpdate无法调用this.setState()你可以理解为更新流程到这一步想要再更改state已经晚了。如果有需要你可以在之前的componentWillReceiveProps中更新state，React会把改变合并到一个更新流程里进行。

而componentDidUpdate则是另一个比较适合我们发起ajax请求的地方，在这个方法里我们还可以比较前后的props变化，再决定是否发起网络请求。一个比较实际的使用场景是保存用户输入到服务器，用户可能会来来回回修改输入的内容，但假如我们判断在修改前后数据最终没有改变，就没有必要发起不必要的网络请求了。

卸载流程

组件既然可以被挂载，同样也可以被卸载，我们可以通过一个名为ReactDOM.unmountComponentAtNode的方法来卸载已经挂载的React组件。在卸载流程中，名为componentWillUnmount的函数会被触发。在组件挂载之后，我们可能定义了一些计时器、绑定了事件监听函数等等，在卸载流程的生命周期函数中，也是我们解绑这些函数的合适位置。

# 实例

接下来呢，我们一起来实现一个小的例子，掌握一下生命周期函数的具体使用方法。

我们要写的是一个从Reddit异步获取帖子内容的小应用。

我们还是沿用之前的展示组件与容器组件的划分方法，一共要写两个组件。

首先我们来写应用的展示组件。

我们通过 const 关键字声明，使用箭头函数定义一个简单的名为RedditList的组件。

一般我们的组件包含多层嵌套的JSX的时候，我们都会习惯在最外面套上一层小括号，因为浏览器在某些情况下可能会自动给换行的代码加上分号，在最外面套上小括号可以消除这一影响。

首先我们定义组件的标题，也就是帖子板块的名称，在这里我们使用字符串模板的方法，就可以省去一些运算符拼接字符串。

之后我们要使用JSX来渲染一个列表。在之前的课程中我们介绍过，JSX当中是可以使用JS的表达式的。一般在渲染列表的时候，我们会拿到一个数组数据，之后通过数组的map方法来渲染出列表的每一项。在这里我们就直接使用JS当中数组的map方法，在传入的函数当中，我们还是使用箭头函数的语法，数组中的每一项对应也返回列表中的一项。

需要特别强调的是，JSX当中渲染列表项的时候，每一项必须带有key属性作为识别列表中某一项的标识。React在渲染组件时需要通过key属性的值来判定列表中的每一项内容。

接下来我们来编写容器组件 RedditFetch . 我们通过类定义的方法来声明它。在它的构造函数里，我们先初始化这个应用的state，在刚才的展示组件当中，我们使用的是一个数组数据，因此在初始化的时候，我们也要把对应的状态数据初始化为数组。

接下来我们定义一个从服务器异步获取数据的方法fetchFromApi，在这里我们使用的是一个名为axios的专门用来发起异步请求的库，当然你也可以用jQuery或者原生的JS方法，你可以选择你自己熟悉的实现方式。在请求成功之后呢，重新设置我们应用的state，获取到posts帖子的数据。

之后就是最关键的部分啦。在这里我们要使用到两个生命周期函数，第一个是componentDidMount，我们刚才介绍过，这里是最适合发起异步服务器请求的地方。因此在其中我们调用刚才定义好的fetchFromApi方法。

再然后是componentWillUnMount，这个生命周期函数是在组件的卸载过程中被触发的，一般在这个生命周期函数中，我们都是做一些解除取消的方法，比方说在这里，我们可以取消掉向服务器发起的请求。

最后在组件的render方法中，调用刚才定义的展示组件RedditList，传入对应的props值，板块名称以及帖子数据，就都OK啦。

最后在ReactDOM的渲染方法中，我们为整个应用传入板块的名称，效果就是我们现在看到的。

# 总结

- React组件渲染包含三个流程：挂载流程、更新流程、卸载流程
- 各个生命周期函数会在特定的时刻触发并适用于不同的使用场景

- 通过使用生命周期函数我们可以对应用进行更精准的控制
- 如果你需要发起网络请求，将其安排在合适的生命周期函数中是值得推荐的做法
- 了解掌握React组件渲染的流程和原理对我们更深入掌握React非常有帮助

# 2-5-表单及事件处理

## 表单

记得在之前，组件类型一课中，我们介绍了一组受控组件与非受控组件的概念。受控与非受控组件就是专门适用于React当中的表单元素的。

在HTML中，表单元素与其他元素最大的不同是它自带值或数据，而且在我们的应用中，只要是有表单出现的地方，就会有用户输入，就会有表单事件触发，就会涉及的数据处理。

在我们用React开发应用时，为了更好地管理应用中的数据，响应用户的输入，编写组件的时候呢，我们就会运用到受控组件与非受控组件这两个概念。

React推荐我们在绝大多数情况下都使用受控组件。这样可以保证表单的数据在组件的state管理之下，而不是各自独立保有各自的数据。

### 表单元素

我们在组件中声明表单元素时，一般都要为表单元素传入应用状态中的值，可以通过state也可以通过props传递，之后需要为其绑定相关事件，例如表单提交，输入改变等。在相关事件触发的处理函数中，我们需要根据表单元素中用户的输入，对应用数据进行相应的操作和改变，来看下面这个例子：

```
class ControlledInput extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      value: ""
    }
  }

  handleChange(event) {
    this.setState({
      value: event.target.value
      })
  }

  render() {
    return <input
              type="text"
              value={this.state.value}
              onChange={() => this.handleChange()}
            />
  }
}
```

受控组件的输入数据是一直和我们的应用状态绑定的，在上面这个例子中，事件处理函数中一定要有关state的更新操作，这样表单组件才能及时正确响应用户的输入，我们可以把setState语句注释掉来试验一下。

### textarea

### HTML

```
<textarea>
  Hello there, this is some text in a text area
</textarea>
```

### JSX

```
<textarea value={this.state.value} onChange={this.handleChange} />
```

在这里我们还是要重申在第一节课中就强调过的，JSX中使用的和HTML标签同名的元素并不等同于原生的HTML标签，这只是React内部抽象出来的一种标签的写法，只是看起来一样而已，下面我们就需要介绍一下表单元素中，JSX和HTML不一样的，需要注意的地方。

在HTML中，textarea标签当中的内容都是在其开闭合标签之间的子节点当中的。而在JSX中，为了统一，textarea也可以定义一个名为value的熟悉，用来传入应用状态中的相关值。

### select

### HTML

```
<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option selected value="coconut">Coconut</option>
  <option value="mango">Mango</option>
</select>
```

### JSX

```
<select value={this.state.value} onChange={this.handleChange}>
    <option value="grapefruit">Grapefruit</option>
    <option value="lime">Lime</option>
    <option value="coconut">Coconut</option>
    <option value="mango">Mango</option>
</select>
```

select也是一样，注意这里的写法，同样我们可以为JSX当中的select标签定义value属性，与应用状态中相关数据值相同的option将会被默认选中。

使用受控组件和非受控组件都是有响应的适用场景的，就拿input来讲，比方说它是一个搜索框，我们需要在应用中实现根据搜索框内容输入异步返回相关搜索建议的功能，那么此处的input就应该是受控组件。而假如它是Todo应用中用来添加新事项的输入框，我们就没有特别的理由需要实时获取其中的数据，只需要在添加事项的事件触发时获取输入框中的值即可，这个地方就可以使用非受控组件。

## 事件

### HTML

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

### JSX

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

React元素的事件属性几乎与HTML中的事件相关属性相同，不过在React当中，事件相关的属性是一小驼峰的方式命名的。在这里还是要强调一下，React元素中的事件处理也是React内部的抽象封装，这里只是说，我们在JSX中写出来，看上去差不多，并不代表这是HTML原生的事件属性。

React元素相关的事件属性我们可以在官方文档的这个页面中找到https://facebook.github.io/react/docs/events.html

新版本的React中，我们可以通过类和函数声明React组件，在这两种形式的声明当中，我们都可以为其定义事件处理函数，函数定义的组件只需要在其方法内部再定义事件触发的函数即可，而如果是类声明组件，就像我们在之前的课程中已经强调过的，类定义组件中的自定义方法默认是没有绑定this的，因此加入我们需要在事件处理函数中调用this.setState一类的方法，就必须要手动将this绑定在相应的事件处理函数上。

# 动手试试

想要亲自实践的话，可以试着编写一个非常简单的 Markdown 编辑器：

```
class MarkdownEditor extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {value: MD_CONTENT};
  }

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

  getRawMarkup() {
    var md = new Remarkable();
    return { __html: md.render(this.state.value) };
  }

  render() {
    return (
      <div className="MarkdownEditor">
        <div className="left">
          <h3>Input</h3>
          <textarea
            onChange={this.handleChange}
            defaultValue={this.state.value} />
        </div>
        <div className="right">
        <h3>Output</h3>
        <div
          className="content"
          dangerouslySetInnerHTML={this.getRawMarkup()}
        />
        </div>
      </div>
    );
  }
}
```

在 Codepen 上试试。

# 3-1-Redux 简介

记得在课程开始的时候我们就讲到，我们使用React的主要情境不是内容网页，而是Web应用。Web应用与我们传统的新闻站、博客等类型的网站有很大的不同。Web应用会涉及到非常多的数据交互、异步传输、甚至我们需要让所有的用户交互和应用功能在同一个页面里完成，也就是开发单页面应用。

换句话讲，我们要开发的Web应用会涉及非常多的状态state改变。包括与服务器的数据交互，界面上导航或其他部件的切换、处理用户输入等等。可以想象，等到我们的应用复杂到一定程度时，如果应用的状态分散在各个组件当中，有的组件只控制自己样式改变的状态，有的组件之间需要相互通信，传递数据。各种各样的数据，状态改变错综复杂，后期维护起来简直会是一场噩梦。万一产品经理这时候再提出说要改几个需求，添加个功能，跳楼的心估计都有了。

之前我们介绍过了React当中有状态组件和无状态组件的概念，React本身也推荐我们尽量控制有状态组件的个数，开发过程中我们编写的组件90%都应该是无状态组件，然后通过props来接收数据。

React本身只留给了我们state和props来管理应用数据的状态，context只是实验性质的一种传递数据的方式。也就是说，React本身并没有很好地提供应用状态管理的解决方案，在一些小应用中我们可能感觉不到状态管理是一个问题，但是随着应用的复杂度增加，这个问题会暴露得越来越明显。这时候我们就需要使用到Redux

既然React推荐我们尽量少得编写有状态组件，那么我们何不干脆把一个应用的数据状态统一在一个地方管理？这便是Redux的理念。

Redux把应用的数据统一储存在一个全局对象中。把应用所有的数据交互和状态改变，统一用固定形式的action动作对象来描述。经由名为Reducer的方法来判断不同的动作如何改变应用状态。最后，我们通过Store对象来负责执行action动作，获取state应用状态、订阅状态改变时触发的事件。

```
// Reducer
const counter = (state = { value: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { value: state.value + 1 };
    case 'DECREMENT':
      return { value: state.value - 1 };
    default:
      return state;
  }
}

class Counter extends Component {
  // State
  constructor() {
    super();
    this.state = counter(undefined, {});
  }

  dispatch(action) {
    this.setState(prevState => counter(prevState, action));
  }
  // Actions
  increment = () => {
    this.dispatch({ type: 'INCREMENT' });
  };

  decrement = () => {
    this.dispatch({ type: 'DECREMENT' });
  };

  render() {
    return (
      <div>
        {this.state.value}
        <button onClick={this.increment}>+</button>
        <button onClick={this.decrement}>-</button>
      </div>
    )
  }
}
```

事实上，Redux提供的更多的是一种应用状态管理问题的解决思路。它的源码很少，只有几kb，我们甚至可以用一些非常基本的函数方法来模拟实现redux的全部功能。并且，即使我们在应用中引入Redux，编写的大部分也都是原生的JS函数和对象而已。也就是说，只要你能够掌握Redux的理念，你甚至无需使用Redux这个库本身，也能够通过它的方式来编写代码解决问题。换句话讲，你可以用自己的代码实现 Redux 的所有核心功能，所以 Redux 也是最适合拿来教学的一个状态管理库。

可能实践中 Redux 在处理数据的流程上有一些繁琐，在对一些比较复杂（例如带有异步）的操作方面比较吃力，但学习 Redux 仍然能让你对应用的状态管理这一方面的功能有一个全面的理解。

在正式开始学习Redux之前，还有一点需要提醒各位同学。任何一个新的框架和库的出现都是为了解决某些或某个特定的问题。我们使用一个框架或库也应该秉持“我是为了解决遇到的问题才使用这个框架或库”的原则。而不是因为某个框架比较流行比较火，在根本没有了解其出现的背景下就开始盲目使用。我们的教程只是为了帮助大家掌握React技术栈的使用方法和理念，而在实际的工作中是否要运用我们了解到的这些工具，还需要具体情况具体分析，等到我们真正遇到了实际的问题时，再拿出相应的库来解决。

要记得，永远不要为了用框架而用框架。本教程的目的也只是为了教给你React主要技术栈的使用方法，并不是告诉你，以后只要开发Web应用就必须使用我们所学的这些工具库，而是在你真正遇到需要这些工具库来解决的问题时再去实际应用。

# 3-2-Action

- action
  - action type
  - action payload
- action Creator
- action Creator Generator

我们还是来看上节课计数器的例子：

```
// Reducer
const counter = (state = { value: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { value: state.value + 1 };
    case 'DECREMENT':
      return { value: state.value - 1 };
    default:
      return state;
  }
}

class Counter extends React.Component {
  // State
  constructor() {
    super();
    this.state = counter(undefined, {});
  }

  dispatch(action) {
    this.setState(prevState => counter(prevState, action));
  }
  // Actions
  increment = () => {
    this.dispatch({ type: 'INCREMENT' });
  };

  decrement = () => {
    this.dispatch({ type: 'DECREMENT' });
  };

  render() {
    return (
      <div>
        <p>{this.state.value}</p>
        <button onClick={this.increment}>+</button>
        <button onClick={this.decrement}>-</button>
      </div>
    )
  }
}

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

我们现在只看有关dispatch的这一部分：

```
dispatch(action) {
    this.setState(prevState => counter(prevState, action));
  }
  // Actions
  increment = () => {
    this.dispatch({ type: 'INCREMENT' });
  };

  decrement = () => {
    this.dispatch({ type: 'DECREMENT' });
  };
```

拿其中的increment方法来说，我们相当于将一个对象传递到了counter函数当中：

```
this.setState(prevState => counter(prevState, { type: 'INCREMENT' }));
```

这里的带有type属性的js对象`{ type: 'INCREMENT' }`就是我们在Redux中定义的action动作。在形式上，action就是带有type属性的JS对象。在Redux的约定中，我们要将所有改变应用状态的操作规范为一个个action，在action中，我们也可以附上要用来修改应用状态state的数据：

```
{ type: 'ADD_TODO', text: 'Use Redux' }
{ type: 'REMOVE_TODO', id: 42 }
{ type: 'LOAD_ARTICLE', response: { ... } }
```

例如上述的这个例子也可以改写为下面这样，我们将自己操作数据的部分去掉，之后根据action传递的具体值来增减状态数据：

```
const counter = (state = { value: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT':
    case 'DECREMENT':
      return { value: state.value + action.num };
    default:
      return state;
  }
}

class Counter extends React.Component {
  ...

  dispatch(action) {
    this.setState(prevState => counter(prevState, action));
  }
  // Actions
  increment = () => {
    this.dispatch({ type: 'INCREMENT', num: 1 });
  };

  decrement = () => {
    this.dispatch({ type: 'DECREMENT', num: -1 });
  };

  ...
}
```

根据代码，我们了解到，目前为止，我们的action有两个作用，一个是定义我们的应用可以进行的动作或操作的类型，另一个是传递改变应用状态的数据。在Redux的约定中，action只有type属性是必须包含的，其他的数据如何定义全在于你想要如何使用，当然如果你希望你定义的action能够规范一些的话，也可以遵从Flux Standard Action的标准：

```
{
  // action 类型
  type: 'INCREMENT',
  // payload 中返回我们要传递的数据，用来修改应用 state
  payload: {
    num: 1
  },
  // payload 数据未获取成功时返回 true
  error: false,
  // 一些不必要在 payload 中传递的其他数据
  meta: {
    success: true
  }
}
```

根据Flux标准，action必须是JS对象，必须包含type属性，可以有payload/error/meta几个属性，除此之外不允许有任何其他属性。在开发应用的过程中，实施规范可以更好地方便团队之间的协同，而不至于不同的开发者写出各种奇奇怪怪的action来。

再进一步，为了防止我们的代码中出现很多写死的数值，减少hard code，可以使用在Redux中称之为action creator的方法来创建action

```
function addNumber(num) {
    return { type: 'INCREMENT', num }
}

function minusNumber(num) {
    return { type: 'DECREMENT', num }
}

...

increment = () => {
    this.dispatch(addNumber(1));
  };

  decrement = () => {
    this.dispatch(minusNumber(-1));
  };
```

继续将我们的代码抽象，我们可以提炼出生成action creator的generator

```
/*function counterActionGenerator(type, num) {
  return function(num) {
    let action = { type, num : num };
    return action;
  }
}*/

const counterActionGenerator = (type, num) => (num) => {
    let action = { type, num : num }
    return action
}

const addNumber = counterActionGenerator('INCREMENT', null)
const minusNumber = counterActionGenerator('DECREMENT', null)
```

另外，为了更清晰地表达我们应用可以进行的所有action动作，还可以在应用的开头定义所有的action type

```
// actionTypes.js
export const INCREMENT = 'INCREMENT'
export const DECREMENT = 'DECREMENT'

// action.js
import { INCREMENT, DECREMENT } from './actionTypes'

const counterActionGenerator = (type, num) => (num) => {
    let action = { type, num : num }
    return action
}

const addNumber = counterActionGenerator(INCREMENT, null)
const minusNumber = counterActionGenerator(DECREMENT, null)
```

普通人第一眼看到这样的定义时肯定会觉得写这样代码的人脑子有毛病，把字符串定义成同名的常量没有任何意义。在小型项目中，定义这些类型的常量可能确实有些多费手续，但是在一些需要团队合作的大中型项目中，预先定义好应用的action类型有许多好处：

- 一个是我们可以在同一个文件中规范应用所有action的命名（因为你的程序不是说写出来就完了，更多的时候还需要后期维护更新）
- 定义action类型的文件类似于一个说明文档，当你想要为应用添加新特性时，可以查阅已有的应用action类型，这样可以避免冲突（我们在正式的开发当中肯定需要相互协作，添加功能的时候也不可能随性添加，要遵从之前的标准和规范）
- 每次代码提交到版本库时，我们也可以在这个文件中很轻松地查阅那些action类型被增加修改或删除了
- （可以减少一些typo错误）假如你打错字的话，在import这一步你的代码就会报错，这样方便你更快地调试，而不是在程序运行后才发现一些字符串错误。

不过话说回来，你的代码具体怎么组织，还是看你开发项目的实际需求，没有必要强行应用某种约定，一切都以解决实际问题为出发点。

# 3-3-Reducer

这节课我们来介绍 Redux 当中的 Reducer.

在之前的课程中，我们定义了这样一个函数：

```
const counter = (state = { value: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { value: state.value + 1 };
    case 'DECREMENT':
      return { value: state.value - 1 };
    default:
      return state;
  }
}
```

counter函数内部呢是一个switch结构，它接受两个参数，state和action，并返回一个新的state，我们可以把这样的函数抽象为：

```
(previousState, action) => newState
```

在Redux中，我们把像这样的，根据应用现有状态和触发的action返回新的状态的函数称为reducer.

话又说回来，为什么这样的函数就被称之为reducer呢？

我们注意到redux的官方文档里专门有一句对reducer命名的解释：

> It's called a reducer because it's the type of function you would pass to Array.prototype.reduce(reducer, ?initialValue)

应该翻译为：

> 之所以将这样的函数称之为reducer，是因为这种函数与被传入 `Array.prototype.reduce(reducer, ?initialValue)` 的回调函数属于相同的类型。

为什么这么讲呢？我们来看一个array使用reduce方法的具体例子：

```
// 这里的callback是和reducer非常相似的函数
// arr.reduce(callback, [initialValue])

var sum = [0, 1, 2, 3].reduce(function(acc, val) {
  return acc + val;
}, 0);
// sum = 6

/* 注意这当中的回调函数 (prev, curr) => prev + curr
 * 与我们redux当中的reducer模型 (previousState, action) => newState 看起来是不是非常相似呢
 */
[0, 1, 2, 3, 4].reduce( (prev, curr) => prev + curr );
```

我们再来看一个简单的具体的reducer的例子：

```
// reducer接受state和action并返回新的state
const todos = (state = [], action) => {
  // 根据不同的action.type对state进行不同的操作，一般都是用switch语句来实现，当然你要用if...else
我也拦不住你
  switch (action.type) {
    case 'ADD_TODO':
      return [
        // 这里使用的是展开运算符语法
        ...state,
        {
          id: action.id,
          text: action.text,
          completed: false
        }
      ];
    // 不知道是什么action类型的话则返回默认state
    default:
      return state;
  }
};
```

如果非要翻译reducer的话，可以将其翻译为缩减器或者折叠器？

为了进一步加深理解，我们再了解一下reduce是什么东西，这个名词其实是函数式编程当中的一个术语，在更多的情况下，reduce操作被称为Fold折叠。

直观起见，我们还是拿JavaScript来理解。reduce属于一个高阶函数，它将其中的回调函数reducer递归应用到数组的所有元素上并返回一个独立的值。这也就是“缩减”或“折叠”的意义所在了。

总而言之一句话，**redux**当中的**reducer**之所以叫做**reducer**，是因为它和 **`Array.prototype.reduce`** 当中传入的回调函数非常相似。

当然，如果你认为这种命名不完美容易产生歧义，你完全可以去给redux提交一个PR，提供一种更加恰当的命名方式。

```
const count = function(state, action) {
    if(action.type == 'INCREMENT') {
        return state + 1;
    } else if(action.type == 'DECREMENT') {
        return state - 1;
    } else {
        return state;
    }
}
```

Reducer 的主体是一个switch结构的运算。要注意记得我们刚才提到的reducer的模型，它只是根据传入的状态数据state和action来判断返回一个新的state. reducer必须是一个纯函数，纯函数主要的含义就是它不可以修改影响输入值，并且没有副作用，副作用指的是例如函数中一些异步调用或者会影响函数作用域之外的变量一类的操作。

另外，注意我们返回新的state时使用的展开操作符的方法，在上面的示例中，这样返回的是一个全新的数组，而不是修改了传入的数组。这也就是我们使用 React 技术栈时尤其需要注意的一点：保证数据的immutability不可变性。

如果state的数据是一个普通的数字或者字符串我们直接也返回数字或字符串就可以了，但是为什么数组或者对象会需要这些特别的返回方式呢？我们可以来做一下实验：

```
var a = 1
var b = a
b = 2
console.log(a)
console.log(b)

var oldArray = [1, 2, 3]
var newArray = oldArray
newArray[1] = 233
console.log(oldArray)
console.log(newArray)

var prevArray = [4, 5, 6]
var nextArray = prevArray.slice()
// var nextArray = [...prevArray]
nextArray[1] = 666
console.log(prevArray)
console.log(nextArray)
```

我们先声明两个整数类型的变量a和b，先给a赋值等于1，之后赋值b等于a，然后我们将b赋值为2，之后查看两个变量的值，a还是1，而b变成了2.

那么如果我们的变量类型是数组的话，会发生什么情况呢？还是声明两个变量，这次是两个数组类型的变量，我们把oldArray赋值为包含元素1，2，3的数组，之后声明newArray，并赋值它等于oldArray，之后再将newArray的第二个元素修改为233，然后我们再来观察一下，我们会发现oldArray和newArray中第二个元素都被修改成了233，也就是说我们在修改newArray时oldArray也受到了影响。这是因为在复制时，我们直接写 newArray = oldArray只是对数组进行了浅拷贝，只是相当于给oldArray多起了一个别名，它俩指向的还是同一个数组。

这时我们就需要对我们的变量进行一些特别的操作。我们再来试一下，这次还是声明两个数组类型的变量，首先声明prevArray是以4，5，6为元素的数组，之后我们为nextArray赋值，不同的是，这里我们使用数组原生的slice方法不传入参数。这时我们再修改next Array中第二个元素的值，我们再来观察一下，这一次两个数组直接没有相互影响，我们通过slice方法实现了对数组变量的深拷贝，像之前的例子中，我们使用ES6的展开操作符也能取得相同的效果。

那么有同学看到这里肯定就想问了，为什么我们要保证reducer返回的是一个新的state数据，而不能直接修改传入的数值呢？首先一点呢，在我们的React应用当中，数据发生改变，界面也要跟着响应改变重新渲染，而这一步之间是需要有一个比较差异的diff操作的，因此改变之前和改变之后的数据都需要使用到。如果我们使用和修改的只是一个数据，那么比较差异就无从谈起了，另外通过使用redux可以实现的一些诸如时间旅行的功能也需要依赖这些不可变的数据来实现。

# 3-4-Store

这节课我们来介绍redux当中最核心的部分Store.

- Store
  - getState
  - dispatch
  - subscribe
  - unsubscribe
- createStore
  - reducer
  - defaultState

Store是我们储存状态数据state的地方。我们通过redux当中的createStore方法来创建一个store

```
import { createStore } from 'redux'
import counter from './reducers'
const store = createStore(counter)
```

我们先从redux库当中引入createStore方法，之后使用const关键字定义store，我们需要给createStore方法中传入我们之前定义好的reducer函数，这样就可以创建出来一个存储reducer方法处理的状态数据的store

那么createStore到底做了一些什么操作呢？我们可以先把生成的store对象输出来看一下，我们会发现它提供了3个主要的方法，另外一个replaceReducer现在还不需要看，我们基本上用不到它。

其中getState方法是用来获取当前存储中的状态数据。

dispatch则是具体将传入的action操作执行并触发listeners事件监听器的地方。

subscribe则是用来添加listener的方法，同时它会返回一个unsubscribe的方法用来取消listener。这里的listener说白了其实就是当状态数据改变时会自动触发的一些事件。

前面我们就介绍过了，redux是一个非常简单的库，在这里我们可以模拟一下createStore的源码，来加深同学们对store的理解：

```
const createStore = (reducer) => {
  let state
  let listeners = []
  // 用来返回当前的state
  const getState = () => state
  // 根据action调用reducer返回新的state并触发listener
  const dispatch = (action) => {
      state = reducer(state, action)
      listeners.forEach(listener => listener())
    }
  /* 这里的subscribe有两个功能
   * 调用 subscribe(listener) 会使用listeners.push(listener)注册一个listener
   * 而调用 subscribe 的返回函数则会注销掉listener
   */
  const subscribe = (listener) => {
      listeners.push(listener)
      return () => {
        listeners = listeners.filter(l => l !== listener)
      }
    }

  return { getState, dispatch, subscribe }
}
```

我们可以看到，createStore接受reducer作为参数，返回的是一个包含上述的三个方法的对象。其中getState就非常简单了，直接返回我们的状态数据；而dispatch则需要进行两个操作，一个是根据传入的action具体调用reducer来获取新的状态数据，之后还需要触发所有的listener，响应状态数据改变之后的动作；最后是subscribe，则是用来添加注册listener的，同时它还会返回一个去除注销listener的方法。虽然我们这里是在模拟实现redux，这也几乎就是我们要使用的redux当中createStore方法的全部功能了。

好，之前的课程中呢，我们都是在模拟redux，应用redux的理念写了一些原生的js方法，这节课呢，我们来看一下redux这个库本身的基本使用方法，下面我们来一起编写一个Redux使用的基本例子：

```
const counter = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
// 这里使用的是ES6的解构赋值的方法
const { createStore } = Redux
// import { createStore } from 'redux'

const store = createStore(counter)
console.log(store.getState())

/* document.addEventListener('click', () => {
  store.dispatch({ type: 'INCREMENT' })
}) */

const render = () => {
  document.body.innerText = store.getState()
}

store.subscribe(() => { console.log(store.getState()) })
let unsubscribe = store.subscribe(render)

render()

store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'DECREMENT' })

unsubscribe()
console.log('unsubscribed!')
```

我们还是实现一个最基本的计数器的功能，首先定义我们的reducer方法，我们使用const关键字声明一个名为counter的函数，传入的参数为state和action，注意到这里我们传入的state后面跟着一个赋值等于0，这里使用到的是ES6的新特性，默认参数，也就是说我们随后调用counter方法如果不传入state的话，它默认会传入0作为state的值。

之后我们写一个switch的逻辑结构，这里也相当于定了好了action的类型，如果是increment增加的话，我们就为state加上1，如果是decrement减少的话，就给state减去1，而如果传入的是我们应用没设计的其他动作，我们就不对state进行操作。

接下来我们从redux当中引入createStore方法，如果你和我一样是在浏览器中编写redux的代码的话，我们需要像之前引入react时一样，在js窗格的设置中引入redux库文件的链接，之后在我们的代码里用解构赋值的方法获取到redux库当中的createStore，这种赋值方法也是es6

当中的新特性，可以让我们直接获取到目标对象当中的同名方法。

之后呢，就是调用createStore方法来定义好我们的sotre对象。

然后，我们编写一个render方法，好让我们的状态数据能够在页面当中显示出来。

我们使用store的subscribe方法，将render绑定到listener里面，这样每当状态数据改变时，render方法就会被自动触发，页面当中的显示就能随之更新。

```
const counter = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
// 这里使用的是ES6的解构赋值的方法
const { createStore } = Redux
// import { createStore } from 'redux'

const store = createStore(counter)
console.log(store.getState())

/* document.addEventListener('click', () => {
  store.dispatch({ type: 'INCREMENT' })
}) */

const render = () => {
  document.getElementById('root').innerText = store.getState()
}

store.subscribe(() => { console.log(store.getState()) })
let unsubscribe = store.subscribe(render)

render()

document.getElementById('increment').addEventListener('click',function() {
  store.dispatch({ type: 'INCREMENT' })
})

document.getElementById('decrement').addEventListener('click',function() {
  store.dispatch({ type: 'DECREMENT' })
})

document.getElementById('unsubscribe').addEventListener('click',function() {
  unsubscribe()
  console.log('unsubscribed!')
})
```

接下来，再为我们的示例添加一些可以交互的功能来改变状态数据，我们在这里添加了三个按钮，分别绑定increment方法、decrement方法以及unsubscribe方法。接下来我们来编写这三个方法。increment直接调用store的dispatch方法，并传入action.type类型为increment作为参数，decrement方法也是类似，传入decrement类型的action作为参数。而unsubscribe方法呢，我们在本节课开始的时候介绍了，store的subscribe方法返回的就是一个现成的unsubscribe方法，因此我们之间赋值刚才的绑定render的方法给unsubscribe就能够获取到它返回的函数。

接下来我们再点击这几个按钮实际操作一下。当我们的事件绑定函数被触发时，页面当中显示的状态数据也就自动更新了。

我们使用redux编写应用时，首要的就是定义一个具体的store对象，因为不管是action还是reducer都是通过store当中的方法具体被调用了，而获取store需要使用到createstore方法，这个方法需要传入reducer作为参数，所以我们在编写代码时需要先写出reducer方法。话又绕了回来，reducer方法需要判断action的类型，所以事实上，最先要被定义好的应该是各种类型的action，action又是用来传入修改state状态数据的参数的，所以最早定义好的应该是应用的state状态数据。

我们在开发稍具规模的一些应用时，肯定是要先规划好应用要处理的数据结构。使用redux时最好先在一个单独的文件中定义好基本的action类型，其实这也就相当于在设计我们的应用都有哪些功能，用户可以触发哪些功能，对数据会造成什么影响。接下来就是获取到store，给我们的用户界面上面添加响应用户操作的事件，以及事件触发之后状态数据改变，我们的界面又要如何跟着响应改变。这其实也就是使用redux进行开发的一般思路啦。

# 3-5-Middleware

Middleware可以翻译为中间件，如果同学们对后端稍微有一些了解的话，对这个概念应该不会陌生，例如我们在查看特定的带有用户信息的网页之前需要登录验证，这个处理登陆验证的模块就可以被称为middleware中间件。

在我们开发Redux的过程中，也有类似的适用场景，比方说，每个action在触发之后，我们希望在控制台看到到底是哪个action被触发以及应用状态前后的变化：

```
console.log('prev state', store.getState())
store.dispatch(action)
console.log('next state', store.getState())
```

最直接的办法当然是在调用dispatch之前输出一次，调用完再输出一次。这样写当然看起来比较傻，而且有一些hard code的性质。那么我们如何做才能让我们的控制台记录功能更优雅一些呢？其实我们可以手动修改创建store的dispatch方法：

在这里呢，我们先用next变量来暂存原本的dispatch方法，之后将dispatch赋值到新的方法，在这个新的名为dispatchAndLog的方法里呢，我们调用了两次控制台记录，在这中间呢，我们来调用next也就是之前原本的dispatch方法来实现我们的需求。

```
const store = createStore(counter)
// 将原本的dispatch方法保留并附加上控制台输出的语句
let next = store.dispatch
store.dispatch = function dispatchAndLog(action) {
  console.log('prev state', store.getState())
  let result = next(action)
  console.log('next state', store.getState())
  return result
}
```

当然这种直接修改dispatch的方式也是不够优雅的，而且直接修改掉原本redux的原生方法其实也算是一种比较脏的处理。另外在现实中，我们肯定会面临一些同时使用多个middleware的情况，所以我们需要把middleware和dispatch的逻辑拆分开来：

我们试着来定义两个middleware，注意到这里middleware函数的写法，我们定义的方法返回了一个返回函数的函数。像我们上面的例子当中，我们想要给一个store加上middleware，必须先获取这个store，之后保存这个store原本的dispatch，在进行一些别的操作之后调用action作为参数传入dispatch，也就是说我们对store、dispatch、action的调用是一种固定的模式，那么我们就可以把这三项提炼出来都当作函数的参数依次传入使用，这也叫作函数的柯里化。

之后我们还需要定义一个名为applyMiddleware的方法，用来给我们的目标store加上特定的middleware，applyMiddleware方法接受两个参数，第一个参数是store，第二个参数则是middleware方法组成的数组。注意到这里我们需要先使用reverse方法颠倒middlewares数组的次序，这样才能保证它与action传递的次序一致。之后呢就是依次为dispatch加上middleware并返回了。

```
let store = createStore(counter);

const confirmationMiddleware = store => next => action => {
    if (confirm('Are you sure?')) {
      next(action)
    }
    return false
}

const loggerMiddleware = store => next => action => {
    console.log('prev state', store.getState())
    let result = next(action)
    console.log('next state', store.getState())
    return result
}

function applyMiddleware(store, middlewares) {
  // 让middlewares的顺序调整成action传递的顺序
  middlewares = middlewares.slice()
  middlewares.reverse()

  let dispatch = store.dispatch
  middlewares.forEach(middleware =>
    dispatch = middleware(store)(dispatch)
  )

  return { ...store, dispatch }
}

store = applyMiddleware(store, [ loggerMiddleware, confirmationMiddleware ])
```

其实Redux本身已经为我们实现了非常完备的middleware支持，前面的这些代码演示只是为了让同学们更好地理解middleware的概念和其实现的原理，在实际的开发中，我们可以直接从redux当中引入applyMiddleware方法，使用的方式也是非常简单，在createStore方法中当作reducer之后的参数传入即可：

```
const { createStore, applyMiddleware } = Redux

const store = createStore(counter, applyMiddleware(loggerMiddleware, confirmationMiddl
eware))
```

# 3-6-react-redux

这节课中，我们将学习React与Redux的协同使用，在前面几节课的示例当中，我们要么是只使用了react或者单独使用了redux，接下来呢，让我们来继续重构之前的计数器的例子，不过这一回呢，我们要正式地协同使用react和redux了：

```
const counter = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}

const { createStore } = Redux;

const store = createStore(counter);
console.log(store.getState());

/* const render = () => {
  document.body.innerText = store.getState();
}; */

const Counter = ({
    value,
    onIncrement,
    onDecrement
    }) => (
    <div>
        <p>{value}</p>
        <button onClick={onIncrement}>+</button>
        <button onClick={onDecrement}>-</button>
    </div>
    )

const render = () => {
    ReactDOM.render(
        <Counter
      value={store.getState()}
      onIncrement={() =>
        store.dispatch({
          type: 'INCREMENT'
        })
      }
      onDecrement={() =>
```

```
        store.dispatch({
          type: 'DECREMENT'
        })
      }
    />,
        document.getElementById('root')
        )
}

store.subscribe(() => { console.log(store.getState()) })
let unsubscribe = store.subscribe(render);

render();
```

redux部分的代码呢，几乎不需要怎么改动，我们还是使用一样的reducer创建一样的store.react是专门用来构建用户界面的框架，那么自然而然，我们在这里需要改动的，是我们之前直接用原生js实现的render方法。另外，希望同学们还没有忘了，使用react进行开发时的最佳实践，用展示组件和容器组件将我们界面的展示结构和数据处理分开来进行。我们先来写比较简单的展示组件，在这里，我们只需要3个界面元素，一个用来显示数字，再加上两个按钮，一个用来增加一个用来减少。

展示组件当中的所有数据和事件处理函数都是通过props传递过来的。在前面几节课当中，我们一直都在使用ES6的新特性，解构赋值。在react组件中我们同样可以使用，在传入参数中直接使用解构赋值的方法获取props，比方说我们这里，获取到的直接就是对应的props.value/props.increment等。

好，写到这里呢。我们先来观察一下我们需要用props传入的3个参数，在之前我们单独使用react的时候，我们是通过react自身的state来控制状态数据的，而现在有的redux，我们可以发现，其实我们需要的value就是redux目前返回的state，另外两个事件函数increment和decrement其实也就对应着同种类型action的执行。也就是说，目前的这3个参数，我们都可以直接从redux当中获取到。

那么接下来我们就直接试着改写之前的render函数，在这里调用ReactDOM的render方法，给value传入store.getState()也就是状态数据，而onIncrement方法就相当于是dispatch了type为increment的action，与之相似，onDecrement方法也是类似。

好了，改写到这一步呢我们已经实现了用react对render方法的重构，我们来点击这两个按钮测试一下交互，我们的react和redux协同使用的应用以及可以正常运行了。

当然，如果这么写的话，每次state改变，我们都需要调用ReactDOM.render方法，这样的写法显然有些耗费效率，我们还可以继续优化我们的代码：

```
const counter = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
```

```
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}

const { createStore } = Redux;

const store = createStore(counter);
console.log(store.getState());

const Counter = ({
    value,
    onIncrement,
    onDecrement
    }) => (
    <div>
        <p>{value}</p>
        <button onClick={onIncrement}>+</button>
        <button onClick={onDecrement}>-</button>
    </div>
    )

class CounterContainer extends React.Component {
  componentDidMount() {
    this.unsubscribe = store.subscribe(() =>
      this.forceUpdate()
    );
  }

  componentWillUnmount() {
    this.unsubscribe();
  }

  render() {
    return (
      <Counter
        value={store.getState()}
        onIncrement={() =>
          store.dispatch({
            type: 'INCREMENT'
          })
        }
        onDecrement={() =>
          store.dispatch({
            type: 'DECREMENT'
          })
        }
      />
    )
  }
}
```

```
store.subscribe(() => { console.log(store.getState()) })

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

这一步呢，我们要完整写出counter的容器组件。在CounterContainer容器组件当中，我们先把之前的render方法直接复制过来，这一部分是不需要改变的。但是这一回呢，我们就不能频繁地调用ReactDOM的render方法了。

之前单独使用react时，我们通过setState方法可以触发React的更新渲染，而目前呢，状态数据全部交由redux管理了，redux当中的状态数据发生改变时，响应的是我们通过subscribe方法绑定的listener，所以在这里我们需要绑定上一个合适的监听函数。

React专门为我们提供了主动触发更新渲染的方法forceUpdate，在这里呢，我们也要配合react的生命周期函数，在componentDidMount方法当中，通过store.subscribe绑定this.forceUpdate()方法，作为状态数据改变时的响应。之后为了让我们的代码更加完善呢，也可以在componentWillUnmount生命周期函数当中进行解绑。

这样改写完之后，我们就只需要调用一次ReactDOM的render方法，之后都通过更新渲染的操作来响应状态数据的改变了。

在这里呢，我们要明确一下，react是用来构建用户界面的框架，而redux则是为了解决应用的状态管理问题而专门开发出来的库。这两者之间没有必然的联系，不是说我使用react的时候就必须使用redux，也不是说使用redux就只能用react构建界面。只是说这两个框架和库在一起搭配使用非常的合适，所以它们才算在了同一个技术栈或生态圈里，而且react官方也专门为我们提供了连接react和redux的库，名字就叫作react-redux.

事实上，在正式的开发当中，我们可以使用React官方为我们提供的库react-redux来绑定React和Redux，通过其提供的Provider和connect方法，我们可以很轻松地传递Redux管理的state到React组件中，像刚才我们手写的countercontainer这种容器组件，也可以直接通过connect这些方法自动生成：

```
const counter = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}

const { createStore } = Redux;
```

```
const store = createStore(counter);
console.log(store.getState());

const { Provider, connect } = ReactRedux;

const Counter = ({
    count,
    onIncrement,
    onDecrement
    }) => (
    <div>
        <p>{count}</p>
        <button onClick={onIncrement}>+</button>
        <button onClick={onDecrement}>-</button>
    </div>
    )
// 在开发模式下，如果不指定类型会报错
Counter.propTypes = {
  number: React.PropTypes.number.isRequired,
  onIncrement: React.PropTypes.func.isRequired,
  onDecrement: React.PropTypes.func.isRequired
}

const mapStateToProps = (state) => ({
    count: state
  })

const mapDispatchToProps = (dispatch) => ({
  onIncrement: () =>
          dispatch({
            type: 'INCREMENT'
          }),
  onDecrement: () =>
          dispatch({
            type: 'DECREMENT'
          })
})

const CounterContainer = connect(mapStateToProps, mapDispatchToProps)(Counter)

store.subscribe(() => { console.log(store.getState()) })

ReactDOM.render(
        <Provider store={store}>
          <CounterContainer />
        </Provider>,
        document.getElementById('root')
        )
```

### Provider

Provider的作用就是向我们的应用传递store的一个容器组件，一般我们都会把它套在应用组件的最外层：

```
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

### connect()

connect方法可以根据我们现有的展示组件自动生成一个容器组件。我们向其传递了两个方法作为参数：

- mapStateToProps(state)
  - 用来对应state和props，并且传入此参数时在state改变时会触发组件更新。
- mapDispatchToProps(dispatch)
  - 根据store的dispatch来传递相应的方法触发action事件。

其实就相当于connect在获取了Redux的store之后，再根据我们传入的方法，把我们需要的部分对应到props属性中，再传递到我们的组件当中。这样在组件里，我们就可以直接通过组件的props来访问Redux的store中的值和方法。

react-redux内部实现了很多优化，可以不用像我们之前手动调用render或forceUpdate方法频繁地重新渲染组件，而且这些优化是我们自己手写很难实现的，所以在正式的项目中，我们最好还是使用官方的这个库来进行react和redux的连接。

# 4-1-react-router-4.0 简介

在Web应用开发当中，前端扮演着越来越重要的角色，很多原本在后端负责的工作现在都可以放到前端来进行。例如对路由的控制。

在Web应用中，我们引入了一个新的概念，叫做前端路由。无需后端返回页面，我们就可以通过前端路由实现url的改变以及页面的切换。

较早前的实现前端路由的方法呢，比如说在angular1当中是通过hash的形式来处理的（也就是带#号的锚链接），前端路由实现的方法多种多样，比方说现在我们也可以通过操纵DOM中的history对象来实现。

为了更好地理解react-router，我们先来试着自己实现一下前端路由的功能。

其实我们简单拆解一下，前端路由主要需要实现两个基本的功能，一个是改变当前的url，另一个是根据改变了的url更改显示相应的页面内容。我们可以通过history.pushState方法来操作url地址。

至于根据改变后的url触发相关页面变化，浏览器本身会自带一个onpopstate事件，但是只有在我们点击返回或前进按钮时才会正常触发，所以我们需要自己动手实现一个绑定事件的功能。

在这里第一步呢，我们需要改写原本浏览器当中的history对象，在pushState方法被调用时，除了改变url呢，我们还需要让它触发onpushstate事件。我们这里呢，之间采用执行匿名函数的方法，完成对history对象的pushState方法的修改。

第二步呢，就需要来定义我们的window.onpopstate方法了，onpopstate这个方法呢，本来就是需要我们指向一个具体的函数的，在这里呢，我们只进行最简单的操作，在路由切换改变的时候呢，在页面当中显示出具体的路由。

接下来还有一步，就是为页面上的所有链接添加上事件绑定函数，这样在页面中的链接被点击之后才能够触发我们的pushState方法而不是直接从浏览器跳转了。

```
// patch history pushState
(function(history){
    var pushState = history.pushState;
    history.pushState = function(state) {
        if (typeof history.onpushstate == "function") {
            history.onpushstate({state: state});
        }
        return pushState.apply(history, arguments);
    }
})(window.history);
// Add trigger function
window.onpopstate = history.onpushstate = function(event) {
  document.getElementById('state').innerHTML = "location: " + document.location + ", s
tate: " + JSON.stringify(event.state);
}
// Bind events to all links
var elements = document.getElementsByTagName('a');
for(var i = 0, len = elements.length; i < len; i++) {
    elements[i].onclick = function (event) {
        event.preventDefault();
        var route = event.target.getAttribute('href');
        history.pushState({page: route}, route, route);
        console.log('current state', history.state)
    }
}
```

这样，每当history.pushState运行后，相应的绑定事件也会被触发，我们就实现了一个基本的前端路由功能。

react-router为我们提供了一些预置的可以实现前端路由功能的组件。使用方法与React的组件保持一致，只需要在我们的应用中添加一些JSX标签，就可以直接为我们的应用添加上前端路由功能。

react-router这个库比较特别，到现在为止已经发布了4个主要版本，而且不同的版本之间差异比较大，尤其是这次发布的4.0版本完全采用了新的理念来设计，这次最新发布的react-router4.0一共包含了3个主要的库：

- react-router-dom (for web)
- react-router-native (for RN)
- react-router (core)

react-router-dom和react-router-native相当于在之前的react-router之上又抽象封装了一层，需要我们关注来手动配置的部分更少了，如果是开发Web应用的话，在绝大多数情况下，我们只需要引入react-router-dom就可以了，并且不需要像之前一样手动调整history等配置。

引入相应的组件，然后添加在我们开发的应用中合适的位置，一切就大功告成啦。

这节课我们先来体验一下react-router的功能，随后的课程当中我们会详细地介绍每个组件的使用方法的。

我们从react-router-dom当中引入3个组件BrowserRouter, Route, Link，这3个组件都是react组件，react-router设计的非常好，它不是通过配置啊绑定啊这种语法来使用，而是直接可以像使用其他react组件一样，我们为其传入props参数来使用。

我们还是直接来看代码，首先我们把BrowserRouter放到最外层，这是给我们的应用添加路由功能的容器组件。之后我们可以直接在应用里面添加Link组件，Link组件的to属性就是对应到相应的路由地址的。然后就是使用route来添加正式的路由了，每一个路由的path属性也是对应着路由地址，这个和上面的Link如果相同的话，点击对应的Link就可以跳转的对应的路由当中，每个路由还对应着一个component属性，这就是当地址跳转到对应的路由时会被渲染出来显示的组件了。

除此之外呢，我们来看这里的Topic组件，首先肯定的是，路由是可以嵌套使用的，与此同时呢，我们还可以通过例如这里使用的match属性来获取一些和路由相关的信息。

总而言之，react-router的使用是非常方便的，和我们使用其他封装好的react组件几乎没有任何区别，下一节课呢，我们将会正式的介绍react-router当中几个比较重要的组件的使用方法。

```
const { BrowserRouter, Route, Link } = ReactRouterDOM

const BasicExample = () => (
  <BrowserRouter>
    <div>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/topics">Topics</Link></li>
      </ul>

      <hr/>

      <Route exact path="/" component={Home}/>
      <Route path="/about" component={About}/>
      <Route path="/topics" component={Topics}/>
    </div>
  </BrowserRouter>
)

const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
)

const About = () => (
  <div>
```

```
    <h2>About</h2>
  </div>
)

const Topics = ({ match }) => (
  <div>
    <h2>Topics</h2>
    <ul>
      <li>
        <Link to={`${match.url}/rendering`}>
          Rendering with React
        </Link>
      </li>
      <li>
        <Link to={`${match.url}/components`}>
          Components
        </Link>
      </li>
      <li>
        <Link to={`${match.url}/props-v-state`}>
          Props v. State
        </Link>
      </li>
    </ul>

    <Route path={`${match.url}/:topicId`} component={Topic}/>
    <Route exact path={match.url} render={() => (
      <h3>Please select a topic.</h3>
    )}/>
  </div>
)

const Topic = ({ match }) => (
  <div>
    <h3>{match.params.topicId}</h3>
  </div>
)

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

# 4-2-react-router-4.0 配置

- BrowserRouter/Route
- Link/NavLink
- match

## Router/Route

在Web应用中，我们可以使用react-router-dom提供给我们的BrowserRouter组件，这个组件的功效和我们在上节课中演示的前端路由实现相同，主要就是实现页面内容与url同步的功能。

使用的方法也很简单，只需要把我们的应用组件嵌套在BrowserRouter组件当中即可：

```
import { BrowserRouter as Router } from 'react-router-dom'

<Router>
    <App />
</Router>
```

在之前版本的react-router中，只为我们提供了Router组件，这一次全新的BrowserRouter组件已经内置了对浏览器的history支持，几乎不需要我们做任何配置就可以正常使用，而且使用的直接就是很漂亮的浏览器地址，没有#也不会生成随机字符串之类的。

Route也就是我们的路由组件了，一般接受两个参数，path以及component来指定路由对应的url以及要渲染的组件：

同时也有我们经常会需要配置的一个属性，exact，比方说在下面这个例子当中，如果我们不声明exact属性，第一个路由只要是后面带有斜杠的地址全部都会匹配到，而加上了exact呢，它就只会匹配到根地址了。

```
import {
    BrowserRouter as Router,
    Route
} from 'react-router-dom'

<Router>
  <div>
    <Route exact path="/" component={Home}/>
    <Route path="/blog" component={Blog}/>
  </div>
</Router>
```

# Link/NavLink

有了路由，我们的应用必然也需要导航，这样我们才能在应用的多个页面之间进行切换。react-router当然也很贴心地为我们提供了现成的导航组件。

Link是最基本的导航组件，一般我们只需要提供一个to作为参数指向其连接的路由：

```
import {
    BrowserRouter as Router,
    Route,
    Link
} from 'react-router-dom'

<Router>
  <div>
    <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/blog">Blog</Link></li>
    </ul>

    <Route exact path="/" component={Home}/>
    <Route path="/blog" component={Blog}/>
  </div>
</Router>
```

但是通常我们在应用中，会为我们当前激活的导航设置不同的样式，这时就需要使用到NavLink，事实上NavLink也是在Link组件的基础上实现的，此外还附加了导航激活的相关属性：

比方说在这里，我们为NavLink加上activeStyle呢，在切换到对应路由的时候它就会加粗显示了。

```
import {
    BrowserRouter as Router,
    Route,
    NavLink as Link
} from 'react-router-dom'

<Router>
  <div>
    <ul>
        <li><Link activeStyle={{ fontWeight: 'bold' }} to="/">Home</Link></li>
        <li><Link activeStyle={{ fontWeight: 'bold' }} to="/blog">Blog</Link></li>
    </ul>

    <Route exact path="/" component={Home}/>
    <Route path="/blog" component={Blog}/>
  </div>
</Router>
```

# match

Route组件默认会为其component组件提供match作为props参数，match对象包含以下几项属性：

- params 预设在url中传递的参数
- isExact 当前url是否绝对匹配此路由
- path 路由设定的path值
- url 当前的url

我们可以在路由设定的组件中使用match传递过来的数据，例如下面这个例子：

```
import {
    BrowserRouter as Router,
    Route,
    NavLink as Link
} from 'react-router-dom'

<Router>
  <div>
    <ul>
        <li><Link activeStyle={{ fontWeight: 'bold' }} to="/repo/react">Repo:react</Li
nk></li>
        <li><Link activeStyle={{ fontWeight: 'bold' }} to="/repo/vue">Repo:vue</Link></
li>
        <li><Link activeStyle={{ fontWeight: 'bold' }} to="/repo/bootstrap">Repo:boots
trap</Link></li>
    </ul>

    <Route path="/repo/:repoName" component={Repo} />
  </div>
</Router>

const Repo = ({ match }) => (
    <div>
        <p>{match.params.repoName}</p>
    </div>
    )
```

# optional param

### React Router v1, v2 and v3

在之前的版本当中，路由的可选参数通过下面代码这种形式进行设置。也就是用小括号把我们路由的可选部分括住。

```
<Route path="to/page(/:pathParam)" component={MyPage} />
```

### React Router v4

React Router v4 和之前的 v1-v3 版本有很大不同, 可选参数的写法也没有在文档中明确提出来，我们通过如下这种方式来传入路由可选参数，则需要在参数的结尾加上一个问号，表示它为可选的。如果不加上可选参数的符号，在我们没有传入参数时，以下面这个例子来说，它就不会渲染MyPage组件了。

```
<Route path="/to/page/:pathParam?" component={MyPage} />
```

# withRouter

在react-router中，获取match对象的方法有许多种，除了Route组件的component属性中包含的组件可以获取match之外，我们还可以通过一个叫做withRouter的方法，直接将路由参数传递到我们的目标组件中，这一方法也就解决了类似之前介绍过的React当中context解决的问题。

withRouter也可以同redux一起协同使用：

```
import { connect } from 'react-redux'
import { withRouter } from 'react-router'
import { TodoList } from '../Components/TodoList'
import { getVisibleTodos } from '../reducer'
import { toggleTodo } from '../action'

const mapStateToProps = (state, { match }) => ({
    todos: getVisibleTodos(
      state,
      match.params.filter || 'all'
      )
  })

export const TodoList = withRouter(connect()(TodoList))
```