

# React入门

## 概念

### 单页面应用

单页面应用(single-page application)，是一个应用程序，它可以加载单个 HTML 页面，以及运行应用程序所需的所有必要资源（例如 JavaScript 和 CSS）。与页面或后续页面的任何交互，都不再需要往返 server 加载资源，即页面不会重新加载。

你可以使用 React 来构建单页应用程序，但不是必须如此。React 还可用于增强现有网站的小部分，使其增加额外交互。用 React 编写的代码，可以与服务器端渲染的标记（例如 PHP）或其他客户端库和平共处。实际上，这也正是 Facebook 内部使用 React 的方式。

### Compiler（编译器）

JavaScript compiler 接收 JavaScript 代码，然后对其进行转换，最终返回不同格式的 JavaScript 代码。最为常见的使用示例是，接收 ES6 语法，然后将其转换为旧版本浏览器能够解释执行的语法。[Babel](#) 是 React 最常用的 compiler。

### JSX

JSX 是一个 JavaScript 语法扩展。它类似于模板语言，但它具有 JavaScript 的全部能力。JSX 最终会被编译为 `React.createElement()` 函数调用，返回称为“React 元素”的普通 JavaScript 对象。通过[查看这篇文档](#)获取 JSX 语法的基本介绍，在[这篇文档]中可以找到 JSX 语法的更多深入教程。

React DOM 使用 camelCase（驼峰式命名）来定义属性的名称，而不使用 HTML 属性名称的命名约定。例如，HTML 的 `tabindex` 属性变成了 JSX 的 `tabIndex`。而 `class` 属性则变为 `className`，这是因为 `class` 是 JavaScript 中的保留字：

```
1  const name = 'Clementine';
2  ReactDOM.render(
3    <h1 className="hello">My name is {name}!</h1>,
4    document.getElementById('root')
5  );
```

### 元素

React 元素是构成 React 应用的基础砖块。人们可能会把元素与广为人知的“组件”概念相互混淆。元素描述了你在屏幕上想看到的内容。React 元素是不可变对象。

```
1  const element = <h1>Hello, world</h1>;
```

通常我们不会直接使用元素，而是从组件中返回元素。

### 组件

React 组件是可复用的小的代码片段，它们返回要在页面中渲染的 React 元素。React 组件的最简版本是，一个返回 React 元素的普通 JavaScript 函数：

```

1 function Welcome(props) {
2   return <h1>Hello, {props.name}</h1>;
3 }

```

组件也可以使用 ES6 的 class 编写：

```

1 class Welcome extends React.Component {
2   render() {
3     return <h1>Hello, {this.props.name}</h1>;
4   }
5 }

```

组件可被拆分为不同的功能片段，这些片段可以在其他组件中使用。组件可以返回其他组件、数组、字符串和数字。根据经验来看，如果 UI 中有一部分被多次使用（Button, Panel, Avatar），或者组件本身就足够复杂（App, FeedStory, Comment），那么它就是一个可复用组件的候选项。组件名称应该始终以大写字母开头（`<Wrapper/>` **而不是** `<wrapper/>`）。有关渲染组件的更多信息，请参阅[这篇文档](#)。

**props**

**props** 是 React 组件的输入。它们是从父组件向下传递给子组件的数据。

记住，**props** 是只读的。不应以任何方式修改它们：

```

1 // 错误做法！
2 props.number = 42;

```

如果你想要修改某些值，以响应用户输入或网络响应，请使用 **state** 来作为替代。

**props.children**

每个组件都可以获取到 **props.children**。它包含组件的开始标签和结束标签之间的内容。例如：

```

1 <Welcome>Hello world!</Welcome>

```

在 `Welcome` 组件中获取 **props.children**，就可以得到字符串 `Hello world!`：

```

1 function Welcome(props) {
2   return <p>{props.children}</p>;
3 }

```

对于 class 组件，请使用 **this.props.children** 来获取：

```

1 class Welcome extends React.Component {
2   render() {
3     return <p>{this.props.children}</p>;
4   }
5 }

```

**state**

当组件中的一些数据在某些时刻发生变化时，这时就需要使用 **state** 来跟踪状态。例如，`Checkbox` 组件可能需要 `isChecked` 状态，而 `NewsFeed` 组件可能需要跟踪 `fetchPosts` 状态。

**state** 和 **props** 之间最重要的区别是：**props** 由父组件传入，而 **state** 由组件本身管理。组件不能修改 **props**，但它可以修改 **state**。

对于所有变化数据中的每个特定部分，只应该由一个组件在其 state 中“持有”它。不要试图同步来自于两个不同组件的 state。相反，应当将其 **提升** 到最近共同祖先组件中，并将这个 state 作为 props 传递到两个子组件。

## key

“key”是在创建元素数组时，需要用到的一个特殊字符串属性。key 帮助 React 识别出被修改、添加或删除的 item。应当给数组内的每个元素都设定 key，以使元素具有固定身份标识。

只需要保证，在同一个数组中的兄弟元素之间的 key 是唯一的。而不需要在整个应用程序甚至单个组件中保持唯一。

不要将 `Math.random()` 之类的值传递给 key。重要的是，在前后两次渲染之间的 key 要具有“固定身份标识”的特点，以便 React 可以在添加、删除或重新排序 item 时，前后对应起来。理想情况下，key 应该从数据中获取，对应着唯一且固定的标识符，例如 `post.id`。

## Hook

<https://react.docschina.org/docs/hooks-intro.html>

## Hook 简介

Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。

```
1  import React, { useState } from 'react';
2
3  function Example() {
4    // 声明一个新的叫做“count”的 state 变量
5    const [count, setCount] = useState(0);
6
7    return (
8      <div>
9        <p>You clicked {count} times</p>
10       <button onClick={() => setCount(count + 1)}>
11         Click me
12       </button>
13     </div>
14   );
15 }
```

`useState` 是我们要学习的第一个“Hook”，这个例子是简单演示。如果不理解也不用担心。

## 框架简介

## ReactJS 简介

- React 起源于 Facebook 的内部项目，因为该公司对市场上所有 JavaScript MVC 框架，都不满意，就决定自己写一套，用来架设 Instagram 的网站。做出来以后，发现这套东西很好用，**就在2013年5月开源了**。
- 由于 React 的设计思想极其独特，属于革命性创新，性能出众，代码逻辑却非常简单。所以，越来越多的人开始关注和使用，认为它可能是将来 Web 开发的主流工具。

- library
- Framework

## 前端三大主流框架

- Angular.js: 出来最早的前端框架, 学习曲线比较陡, NG1学起来比较麻烦, NG2开始, 进行了一系列的改革, 也开始启用组件化了; 在NG中, 也支持使用TS (TypeScript) 进行编程;
- Vue.js: 最火的一门前端框架, 它是中国人开发的, 对我们来说, 文档要友好一些;
- React.js: 最流行的一门框架, 因为它的设计很优秀;
- windowsPhone 7 7.5 8 10

## React与vue.js的对比

### 组件化方面

1. 什么是模块化: 从 **代码** 的角度, 去分析问题, 把我们编程时候的业务逻辑, 分割到不同的模块中来进行开发, 这样能够 **方便代码的重用**;
  2. 什么是组件化: 从 **UI** 的角度, 去分析问题, 把一个页面, 拆分为一些互不相干的小组件, 随着我们项目的开发, 我们手里的组件会越来越多, 最后, 我们如果要实现一个页面, 可能直接把现有的组件拿过来进行拼接, 就能快速得到一个完整的页面, 这样方 **便了UI元素的重用**; **组件是元素的集合体**;
  3. 组件化的好处:
  4. Vue是如何实现组件化的: .vue 组件模板文件, 浏览器不识别这样的.vue文件, 所以, 在运行前, 会把 .vue 预先编译成真正的组件;
- template: UI结构
  - script: 业务逻辑和数据
  - style: UI的样式
5. React如何实现组件化: 在React中实现组件化的时候, 根本没有像 .vue 这样的模板文件, 而是, 直接使用JS代码的形式, 去创建任何你想要的组件;
- React中的组件, 都是直接在 js 文件中定义的;
  - React的组件, 并没有把一个组件拆分为三部分 (结构、样式、业务逻辑), 而是全部使用JS来实现一个组件的; (也就是说: 结构、样式、业务逻辑是混合在JS里面一起编写出来的)

### 开发团队方面

- React是由FaceBook前端官方团队进行维护和更新的; 因此, React的维护开发团队, 技术实力比较雄厚;
- Vue: 第一版, 主要是有作者 尤雨溪 专门进行维护的, 当 Vue更新到 2.x 版本后, 也有了一个小团队进行相关的维护和开发;

### 社区方面

- 在社区方面, React由于诞生的较早, 所以社区比较强大, 一些常见的问题、坑、最优解决方案, 文档、博客在社区中都是可以很方便就能找到的;
- Vue是近两年才诞生开源出来的, 所以, 它的社区相对于React来说, 要小巧一些, 所以, 可能有一些坑, 没人踩过;

### 移动APP开发体验方面

- Vue, 结合 Weex 这门技术, 提供了 迁移到 移动端App开发的体验 (Weex, 目前只是一个小的玩具, 并没有很成功的大案例; )

- React, 结合 ReactNative, 也提供了无缝迁移到 移动App的开发体验 (RN用的最多, 也是最火最流行的);

## 为什么要学习React

1. 设计很优秀, 是基于组件化的, 方便我们UI代码的重用;
2. 开发团队实力强悍, 不必担心短更的情况;
3. 社区强大, 很多问题都能找到对应的解决方案;
4. 提供了无缝转到 ReactNative 上的开发体验, 让我们技术能力得到了拓展; 增强了我们的核心竞争力

## React中几个核心的概念

### 虚拟DOM (Virtual Document Object Model)

VDOM, 也叫虚拟DOM, 并不是什么高大上的新事物, 它是仅存于内存中的DOM, 因为还未展示到页面中, 所以称为VDOM。

- DOM的本质是什么: 就是用JS表示的UI元素
- DOM和虚拟DOM的区别:
  - DOM是由浏览器中的JS提供功能, 所以我们只能人为的使用 浏览器提供的固定的API来操作DOM对象;
  - 虚拟DOM: 并不是由浏览器提供的, 而是我们程序员手动模拟实现的, 类似于浏览器中的DOM, 但是有着本质的区别;
- 为什么要实现虚拟DOM:
- 什么是React中的虚拟DOM:
- 虚拟DOM的目的:

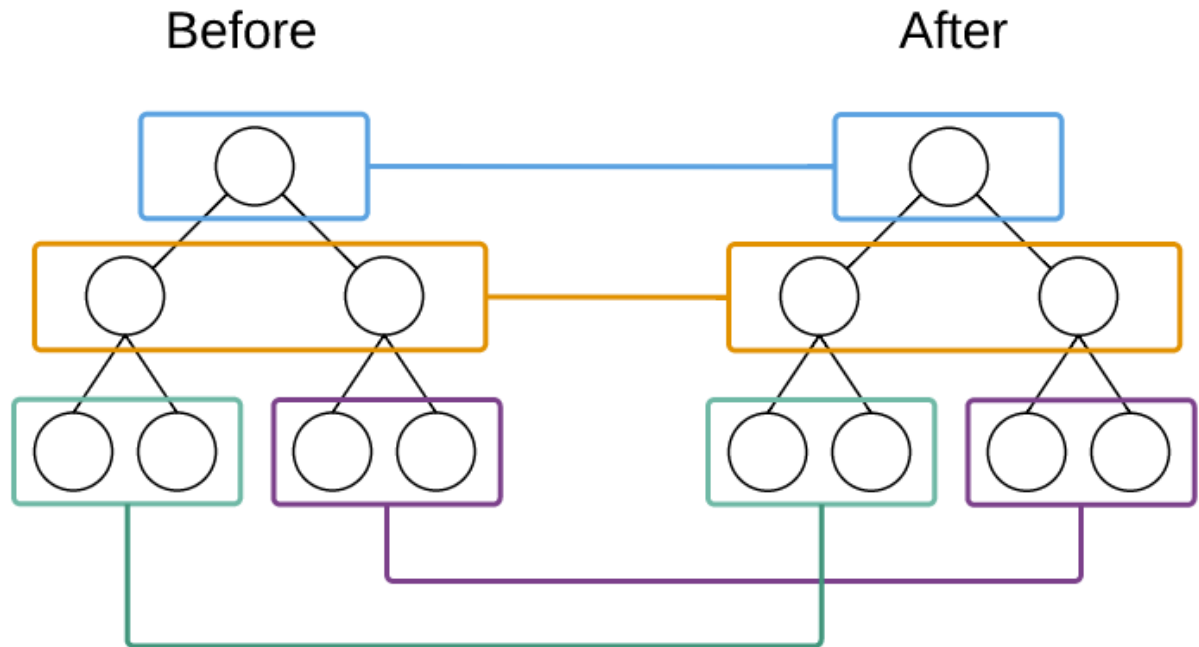
时间 ↕	新关注人数 ↕	取消关注人数 ↕	净增关注人数 ↕	累积关注人数 ↕
2015-11-21	0	0	0	0
2015-11-20	0	0	0	285
2015-11-19	1	0	1	285
2015-11-18	0	1	-1	284
2015-11-17	2	0	2	284
2015-11-16	1	3	-2	282
2015-11-15	0	1	-1	285
2015-11-14	0	0	0	286

### Diff算法

- tree diff: 新旧DOM树, 逐层对比的方式, 就叫做 tree diff; 每当我们从前到后, 把所有层的节点对比完后, 必然能够找到那些 需要被更新的元素;
- component diff: 在对比每一层的时候, 组件之间的对比, 叫做 component diff; 当对比组件的时候, 如果两个组件的类型相同, 则暂时认为这个组件不需要被更新, 如果组件的类型不同, 则立即将旧组件移除, 新建一个组

件，替换到被移除的位置；

- element diff:在组件中，每个元素之间也要进行对比，那么，元素级别的对比，叫做 element diff；
- key: key这个属性，可以把 页面上的 DOM节点 和 虚拟DOM中的对象，做一层关联关系；



时间	新关注人数	取消关注人数	净增关注人数	累积关注人数
2015-11-21	0	0	0	0
2015-11-20	0	0	0	285
2015-11-19	1	0	1	285
2015-11-18	0	1	-1	284
2015-11-17	2	0	2	284
2015-11-16	1	3	-2	282
2015-11-15	0	1	-1	285
2015-11-14	0	0	0	286

需求：实现一个点击 表头，完成 数据根据点击的 列进行排序的功能；

分析步骤：

首先，我们这些所有的列表中渲染出来的数据，都在 浏览器的内存中，以 一个数组的形式存在；

当点击每一个表头的时候，为这每一个表头绑定点击事件，在事件的处理函数中：根据点击的列名，对数组中的每一项进行排序，最终，排序好的新数组，就是我们将要展示到页面上的数据；（这一步完成后，数组是最新的，页面老的，并没有被刷新）

当得到排序好的数组之后，应该把最新的数组，更新到页面中去：使用原生的for循环手动拼接每个tr字符串，append到页面中，使用Jquery、使用 前端模板引擎；当我们拼接好字符串后，就可以把最新的HTML字符串替换到页面上，这样，就完成了我们的需求；

我们上述的实现思路，有没有什么性能的问题？？？

问题：哪怕我们的数据，发生了一丢丢的变化，也会被强制重建整颗DOM树；我们这么做，涉及到很多元素的重绘和重排，导致性能浪费严重；

如何解决上述的这个问题呢？？？

解决方案：只要实现按需更新页面上的元素即可，只需要把 修改的数据，所对应的DOM元素重新构建一下，其它没有变化的数据，所对应的DOM节点不需要被强制更新；

问题：如何实现上述的解决方案：（问题就是：如何按需更新页面上的元素）

要实现上述的解决方案，只需拿到 页面被更新前的 内存中的 DOM树，同时，再拿到 页面更新前，新渲染出来的 内存DOM树，然后，对比这两棵新旧DOM树，只需要找到那些需要被重新创建和修改的元素即可，这样，就能实现DOM的按需更新；

问题：如何拿到这两棵新旧DOM树呢？？？（拆分问题：如何从浏览器的内存中获取到 浏览器自己私有的那两棵DOM树？？）

如果要拿浏览器中的DOM树，那么，我们必须调用浏览器提供的相关JS 的 API 才可以，问题来了（浏览器有提供这样的API吗？）

得出结果：浏览器并没有提供这样的API；

既然浏览器没有提供这样的API，那么我们是不是可以 自己 模拟这两棵 新旧 DOM树呢？？？

问题：如何自己模拟新旧两棵DOM树呢？？？（拆分问题：如何自己模拟一个 DOM节点 `<p title="真的">花哥好帅啊~~</p>`）

手动模拟DOM树的原理：使用 JS 创建一个 对象，用这个对象，来模拟 每个DOM节点，然后在每个DOM节点中，又提供了 类似于 children 这样的属性，来描述当前DOM的子节点，这样，当 DOM节点形成了嵌套关系，就模拟出了一颗 DOM 树；

虚拟DOM的本质：就是使用JS对象来模拟DOM树；

虚拟DOM的目的：为了实现DOM节点的高效更新；

虚拟DOM的概念

React已经在内部帮我们实现了虚拟DOM

## React项目的创建

1. 运行 `cnpm i react react-dom -S` 安装包
2. 在项目中导入两个相关的包：

```
1 // 1. 在 React 学习中, 需要安装 两个包 react react-dom
2 // 1.1 react 这个包, 是专门用来创建React组件、组件生命周期等这些东西;
3 // 1.2 react-dom 里面主要封装了和 DOM 操作相关的包, 比如, 要把 组件渲染到页面上
4 import React from 'react'
5 import ReactDOM from 'react-dom'
```

### 3. 使用JS的创建虚拟DOM节点:

```
1 // 2. 在 react 中, 如要创建 DOM 元素了, 只能使用 React 提供的 JS API 来创建, 不能【直接】像 Vue
  中那样, 手写 HTML 元素
2 // React.createElement() 方法, 用于创建 虚拟DOM 对象, 它接收 3个及以上的参数
3 // 参数1: 是个字符串类型的参数, 表示要创建的元素类型
4 // 参数2: 是一个属性对象, 表示 创建的这个元素上, 有哪些属性
5 // 参数3: 从第三个参数的位置开始, 后面可以放好多的虚拟DOM对象, 这写参数, 表示当前元素的子节点
6 // <div title="this is a div" id="mydiv">这是一个div</div>
7
8 var myH1 = React.createElement('h1', null, '这是一个大大的H1')
9
10 var myDiv = React.createElement('div', { title: 'this is a div', id: 'mydiv' }, '这是一个
  div', myH1)
```

### 4. 使用 ReactDOM 把元素渲染到页面指定的容器中:

```
1 // ReactDOM.render('要渲染的虚拟DOM元素', '要渲染到页面上的哪个位置中')
2 // 注意: ReactDOM.render() 方法的第二个参数, 和vue不一样, 不接受 "#app" 这样的字符串, 而是需要传递
  一个 原生的 DOM 对象
3 ReactDOM.render(myDiv, document.getElementById('app'))
```

## JSX语法

1. 如要使用 JSX 语法, 必须先运行 `cnpm i babel-preset-react -D`, 然后再 `.babelrc` 中添加 语法配置;
2. JSX语法的本质: 还是以 `React.createElement` 的形式来实现的, 并没有直接把 用户写的 HTML代码, 渲染到页面上;
3. 如果要在 JSX 语法内部, 书写 JS 代码了, 那么, 所有的JS代码, 必须写到 `{}` 内部;
4. 当 编译引擎, 在编译JSX代码的时候, 如果遇到了 `<` 那么就把它当作 HTML代码去编译, 如果遇到了 `{}` 就把 花括号内部的代码当作 普通JS代码去编译;
5. 在`{}`内部, 可以写任何符合JS规范的代码;
6. 在JSX中, 如果要为元素添加 `class` 属性了, 那么, 必须写成 `className`, 因为 `class` 在ES6中是一个关键字; 和 `class` 类似, label标签的 `for` 属性需要替换为 `htmlFor` .
7. 在JSX创建DOM的时候, 所有的节点, 必须有唯一的根元素进行包裹;
8. 如果要写注释了, 注释必须放到 `{}` 内部

## 创建组件

### 第一种基本组件的创建方式

```

import React, {Component} from "react";

class Home extends Component { 继承Component
  constructor(props) {
    super(props);
    this.state = {
      name : "xianjs", age : 26,
    };
  }

  render() {
    return (<div className="xianjs">
      <h1>Hello Word!</h1>
      <p>name : { this.state.name }</p>
      <p>name : { this.state.age }</p>
    </div>);
  }
}

export default Home;

```

state定义数据

模板展示数据

导出组件

父组件向子组件传递数据

属性扩散

将组件封装到单独的文件中

## 第二种创建组件的方式

了解ES6中class关键字的使用

基于class关键字创建组件

- 使用 class 关键字来创建组件



```
1 class Person extends React.Component{
2     // 通过报错提示得知：在class创建的组件中，必须定义一个render函数
3     render(){
4         // 在render函数中，必须返回一个null或者符合规范的虚拟DOM元素
5         return <div>
6             <h1>这是用 class 关键字创建的组件! </h1>
7         </div>;
8     }
9 }
```

## 两种创建组件方式的对比

1. 用构造函数创建出来的组件：专业的名字叫做“无状态组件”
2. 用class关键字创建出来的组件：专业的名字叫做“有状态组件”

用构造函数创建出来的组件，和用class创建出来的组件，这两种不同的组件之间的**本质区别就是**：有无state属性！！！有状态组件和无状态组件之间的本质区别就是：有无state属性！

## React基础知识

---

### 目录结构分析

### 创建组件

```

import React from 'react';      导入react

class News extends React.Component{
  constructor(props){          继承
    super(props);
    this.state = {
      user : [ '李四', 26, '男' ]  定义数据
    };
  }

  render(){
    return (<div className='news'>  一个根节点
      <ul>
        <li>{ this.state.user[ 0 ] }</li>  模板语法
      </ul>
    </div>);
  }
}

export default News;          导出组件

```

## JSX语法

<https://www.jianshu.com/p/813aa32555b8>

### 求值表达式

要使用 JavaScript 表达式作为属性值，只需把这个表达式用一对大括号 ({} ) 包起来，不要用引号 ( “ ” )。在编写 JSX 时，在 {} 中不能使用语句 (if 语句、for 语句等等)，但可以使用求值表达式，这本身与 JSX 没有多大关系，是 JS 中的特性，它是会返回值的表达式。我们不能直接使用语句，但可以把语句包裹在函数求值表达式中运用。

### 条件判断的写法

你没法在 JSX 中使用 if-else 语句，因为 JSX 只是函数调用和对象创建的语法糖。在 {} 中使用，是不合法的 JS 代码，不过可以采用三元操作表达式

```

1  var HelloMessage = React.createClass({
2    render: function() {
3      return <div>Hello {this.props.name ? this.props.name : "World"}</div>;
4    }
5  });
6  ReactDOM.render(<HelloMessage name="xiaowang" />, document.body);

```

可以使用比较运算符“||”来书写，如果左边的值为真，则直接返回左边的值，否则返回右边的值，与if的效果相同。

```

1  var HelloMessage = React.createClass({
2    render: function() {
3      return <div>Hello {this.props.name || "World"}</div>;
4    }
5  });

```

## 函数表达式

( ) 有强制运算的作用

```

1  var HelloMessage = React.createClass({
2    render: function() {
3      return <div>Hello {
4        (function(obj){
5          if(obj.props.name)
6            return obj.props.name
7          else
8            return "World"
9        })(this)
10     }</div>;
11   }
12 });
13 ReactDOM.render(<HelloMessage name="xiaowang" />, document.body);

```

外括号“)”放在外面和里面都可以执行。唯一的区别是括号放里面执行完毕拿到的是函数的引用，然后再调用“function(){}(this)()”; 括号放在外面的时候拿到的事返回值。

## 组件的生命周期

组件的生命周期分成三个状态：

```

1  * Mounting: 已插入真实 DOM
2  * Updating: 正在被重新渲染
3  * Unmounting: 已移出真实 DOM

```

React 为每个状态都提供了两种处理函数，will 函数在进入状态之前调用，did 函数在进入状态之后调用，三种状态共计五种处理函数。

```

1  * componentWillMount()
2  * componentDidMount()
3  * componentWillUpdate(object nextProps, object nextState)
4  * componentDidUpdate(object prevProps, object prevState)
5  * componentWillUnmount()

```

此外，React 还提供两种特殊状态的处理函数。

```
1 * componentWillReceiveProps(object nextProps): 已加载组件收到新的参数时调用
2 * shouldComponentUpdate(object nextProps, object nextState): 组件判断是否重新渲染时调用
```

## 注释

JSX 里添加注释很容易；它们只是 JS 表达式而已。你只需要在一个标签的子节点内(非最外层)小心地用 {} 包围要注释的部分。

```
1 { /* 一般注释，用 {} 包围 */ }
```

## 绑定数据

{}

## 绑定对象

## 绑定属性( 绑定class 绑定style)

class绑定需要使用className

for绑定需要使用htmlFor

```
1 <li>绑定属性 : <a href={ this.state.url }>百度</a></li>
2 <li>class绑定 <div className='red' className={ this.state.bg }>green</div></li>
3 <li title={ this.state.xx || '未知数据' }>title绑定</li>
4 <li>
5     for绑定需要使用htmlFor
6     <label htmlFor="userName">userName</label>
7     <input type="text" id="userName"/>
8 </li>
```

style绑定

```
1 style : {
2     background : '#666',
3 },
```

```
1 <li style={ { 'color' : 'red' } } style={ this.state.style }>行内样式</li>
```

## 引入图片

```
1 import img1 from '../assets/img/1.jpg';
```

```
1 <img src={ img1 } width="120" height="120" alt=""/>
2 <img src={ require('../assets/img/1.jpg') } width="120" height="120" alt=""/>
```

## 循环数组渲染数据

循环渲染数据需要绑定key

```
1 list : [ {
2   name : '李四',
3   age : 26,
4   sex : '男',
5 }, {
6   name : '大牛',
7   age : 22,
8   sex : '男',
9 }, {
10  name : '小丽',
11  age : 14,
12  sex : '女',
13 } ],
```

```
1 { this.state.list.map((value, index) => {
2   return (<li key={ index }>{ value.name } - { value.age } - { value.sex }</li>);
3 }) }
```

注意的点： 1、所有的模板要被一个根节点包含起来 2、模板元素不要加引号 3、用大括号{}绑定数据 4、循环数据要加key 5、img要加alt 6、绑定属性 class 要变成 className for 要变成 htmlFor style属性写法

```
1 <div style={{'color': 'red'}}>行内样式1双大括号</div>
2 <div style={this.state.style}>行内样式2 style="{ '{this.state.style}' }"</div>
```

## 事件 方法

### 获取state的值

```
1 import React from 'react';
2
3 class Handler extends React.Component{
4   constructor(props){
5     super(props);
6     this.state = {
7       list : [ 1, 2, 3 ],
8       name : 'xianJs',
9       age : '26岁',
10    };
11  }
```

```

11     this.getName = this.getName.bind(this);
12 }
13
14 getList(params){
15     console.log('params', params);
16 }
17
18 //箭头函数
19 getAge = () => {
20     console.log(this.state.age);
21 };
22
23 //在构造函数绑定this
24 getName(){
25     console.log(this.state.name);
26 }
27
28 render(){
29     return (<div>
30         /*第一种绑定this*/
31         <button onClick={ this.getList.bind(this) }>点击事件</button>
32         <button onClick={ this.getAge }>获取age</button>
33         <button onClick={ this.getName }>获取name</button>
34     </div>);
35 }
36
37 }
38
39 export default Handler;

```

```

1  绑定事件处理函数this的几种方法:
2  第一种方法:
3      run(){
4
5          alert(this.state.name)
6      }
7      <button onClick={this.run.bind(this)}>按钮</button>
8
9
10
11  第二种方法:
12      构造函数中改变
13
14      this.run = this.run.bind(this);
15
16
17      run(){
18
19          alert(this.state.name)
20      }
21      <button onClick={this.run}>按钮</button>
22

```

```

23
24
25 第三种方法:
26      run={() => {
27          alert(this.state.name)
28      }
29
30      <button onClick={this.run}>按钮</button>

```

## 点击事件传递参数

```

1  //修改state的值,传递参数
2  setName = (e, name) => {
3      console.log(name);
4      this.setState({
5          name : name,
6          list : [ 1, 5, 6 ],
7      });
8  };
9  //传递参数2
10 setAge = (age) => {
11     return (e) => {
12         this.setState({
13             age : age,
14         });
15         console.log(age);
16     };
17 };
18 //绑定this的方式传递参数
19 setList = (list) => {
20     console.log(list);
21 };
22
23 render(){
24     return (<div>
25         <p>name = { this.state.name } <br/> age={ this.state.age }</p>
26         <button onClick={ (e) => this.setName(e, '北京天安门') }>传递参数1
27         </button>
28         <button onClick={ this.setAge(22) }>传递参数2</button>
29         <button onClick={ this.setList.bind(this, [ 4, 3, 2, 1 ]) }>传递参数3</button>
30     </div>);
31 }

```

## React定义方法

```

1  constructor(props){
2      super(props);
3      this.state = {

```

```

4     list : [ 1, 2, 3 ],
5     name : 'xianJs',
6     age : '26岁',
7   };
8   this.getName = this.getName.bind(this);
9 }
10
11 getList(params){
12   console.log('params', params);
13 }
14
15 //箭头函数
16 getAge = () => {
17   console.log(this.state.age);
18 };
19
20 //在构造函数绑定this
21 getName(){
22   console.log(this.state.name);
23 }

```

## 获取数据

## 改变数据

```

1 //修改state的值
2 setName = () => {
3   this.setState({
4     name : '北京',
5     list : [ 1, 5, 6 ],
6   });
7 };
8
9 render(){
10   return (<div>
11     <p>name = { this.state.name } <br/> list={ this.state.list }</p>
12     <button onClick={ this.setName }>修改name值</button>
13   </div>);
14 }

```

## 执行方法传值

```

1 class Handler extends React.Component{
2   constructor(props){
3     super(props);
4     this.state = {
5       list : [ 1, 2, 3 ],

```



```

6      name : 'xianJs',
7      age : '26岁',
8  };
9  }
10
11  //修改state的值,传递参数
12  setName = (e, name) => {
13      console.log(name);
14      this.setState({
15          name : name,
16          list : [ 1, 5, 6 ],
17      });
18  };
19  //传递参数2
20  setAge = (age) => {
21      return (e) => {
22          this.setState({
23              age : age,
24          });
25          console.log(age);
26      };
27  };
28  //绑定this的方式传递参数
29  setList = (list) => {
30      console.log(list);
31  };
32
33  render(){
34      return (<div>
35          <p>name = { this.state.name } <br/> age={ this.state.age }</p>
36          <button onClick={ (e) => this.setName(e, '北京天安门') }>传递参数1
37          </button>
38          <button onClick={ this.setAge(22) }>传递参数2</button>
39          <button onClick={ this.setList.bind(this, [ 4, 3, 2, 1 ]) }>传递参数3</button>
40      </div>);
41  }
42
43  }

```

## 事件对象

```

1  setAge = (e,age) => {
2      console.log(e,age);
3      /*buttonDOM节点*/
4      console.log(e.target.getAttribute('uid'));//获取自定义属性
5      e.target.style.background = 'red';
6  };
7
8  render(){
9      return (<div>
10          <button uid="666" onClick={ (e) => {this.setAge(e,16);} }>事件对象获取</button>
11      </div>);
12  }

```

## 表单事件

```
1 <p>userName = { this.state.userName }</p>
2 <input type="text" placeholder="userName" onChange={ this.inputChange }/>
3 <button uid="666" onClick={ this.getName }>获取表单值</button>
4
5 inputChange = (e) => {
6   //将获取到的表单value值赋值给this.state.userName
7   this.setState({
8     userName : e.target.value,
9   });
10   console.log(e.target.value);
11 };
```

## ref双向数据绑定

```
1 <input type="text" ref="userAge" onChange={ this.inputAge }/>
2
3 inputAge = (e) => {
4   //refs对象获取DOM节点
5   console.log(this.refs.userAge.value);
6 };
```

## 键盘事件

```
1 <input type="text" onKeyDown={ this.inputKeyDown }/>
2
3 inputKeyDown = (e) => {
4   console.log(e.keyCode);
5 };
```

## 表单

```
1 约束性和非约束性组件：
2
3 非约束性组:<input type="text" defaultValue="a" /> 这个 defaultValue 其实就是原生DOM中的
value 属性。
4
5 这样写出的来的组件，其value值就是用户输入的内容，React完全不管输入的过程。
6
7
```

```
8      约束性组件: <input value={this.state.username} type="text" onChange=
{this.handleChange} />
```

9  
10 这里, value属性不再是一个写死的值, 他是 this.state.username, this.state.username 是由 this.handleChange 负责管理的。

11  
12  
13 这个时候实际上 input 的 value 根本不是用户输入的内容。而是onChange 事件触发之后, 由于 this.setState 导致了一次重新渲染。不过React会优化这个渲染过程。看上去有点类似双休数据绑定

<https://www.cnblogs.com/yuyujuan/p/10124025.html>

```
1  import React,{ Component } from 'react';
2
3  class Form extends Component{
4      constructor(props){
5          super(props);
6          this.state = {
7              userData : {
8                  name : '',
9                  age : '',
10                 sex : '',
11                 hobby : [
12                     { id : 0,label : '篮球', }, { id : 1,label : '游泳', },
13                     { id : 2,label : '蹴鞠', }, { id : 3,label : '棒球', } ],
14                 city : [
15                     { id : 0,city : '南京', }, { id : 1,city : '北京', },
16                     { id : 2,city : '海南', }, { id : 3,city : '新疆', } ],
17             },
18         };
19     }
20
21     render(){
22         return (<div>
23             <form action="" onSubmit={ this.handleSubmit }>
24                 <p>姓名: <input type="text" onChange={ this.changName } value={
this.state.userData.name }/></p>
25                 <p>年龄: <input type="text" onChange={ this.changAge } value={
this.state.userData.age }/></p>
26                 <p>性别:
27                     <input type="radio" name="sex" value='1' onChange={ this.changSex }/> 男 <br/>
28                     <input type="radio" name="sex" value='0' onChange={ this.changSex }/> 女 <br/>
29                 </p>
30                 <p>
31                     城市: <select value={ this.state.userData.selectCity } onChange={ this.changCity }>
32                         { this.state.userData.city.map(item => {
33                             return <option key={ item.id }>{ item.city }</option>;
34                         }) }
35                     </select>
36                 </p>
37                 <p><input type="submit" value="数据提交保存" /></p>
```

```
38     </form>
39   </div>;
40 }
41
42 changName = (e) => {
43   console.log(e.target.value);
44   let data = Object.assign({}, this.state.userData, {
45     name : e.target.value,
46   });
47   this.setState({
48     userData : data,
49   });
50 };
51
52 changAge = (e) => {
53   let data = Object.assign({}, this.state.userData, {
54     age : e.target.value,
55   });
56   this.setState({
57     userData : data,
58   });
59 };
60
61 handleSubmit = (e) => {
62   e.preventDefault();//阻止表单的默认提交事件
63   console.log(this.state.userData);
64 };
65
66 changSex = (e) => {
67   let data = Object.assign({}, this.state.userData, {
68     sex : e.target.value,
69   });
70   this.setState({
71     userData : data,
72   });
73 };
74
75 changCity = (e) => {
76   let data = Object.assign({}, this.state.userData, {
77     selectCity : e.target.value,
78   });
79   this.setState({
80     userData : data,
81   });
82 };
83 }
84
85
86 export default Form;
```

```

{
  this.state.hobby.map((value,key)=>{
    return (
      <span key={key}>
        {value.do}
        <input type="checkbox" checked={value.checked} onChange={this.handleHobby.bind(this,key)} />
      </span>
    )
  })
}

handleHobby=(key)=>{
  var hobby=this.state.hobby;
  hobby[key].checked=!hobby[key].checked;
  this.setState({
    hobby:hobby
  })
  console.log(this.state.hobby);
}

```

另外就是setState只能对数据整体进行操作，而不能直接对数组和对象的项进行操作，所以这里需要使用一个中间变量进行过渡。

<https://www.cnblogs.com/yuyujuan/category/1329267.html>

---

姓名:

年龄:

性别: ☐ 男  
☐ 女

城市:  ▼

## 组件通信

<https://www.cnblogs.com/yuyujuan/p/10125283.html>

### 什么是组件？

React通过组件的思想，将界面拆分成一个个可复用的模块，每一个模块就是一个React 组件。一个React 应用由若干组件组合而成，一个复杂组件也可以由若干简单组件组合而成。

React 组件可以用好几种方式声明，可以是一个包含 render() 方法的类，也可以是一个简单的函数，不管怎样，它都是以props作为输入，返回 React 元素作为输出。

### 组件的作用

React组件最核心的作用是返回React元素。

这里你也许会问：React元素不应该是由React.createElement() 返回的吗？

其实React组件就是调用React.createElement()，返回React元素，供React内部将其渲染成最终的页面DOM。

```

1 //函数式
2 function Welcome(props) {
3   return <h1>Hello, {props.name}</h1>;
4 }

```

```

1 //类定义
2 class Welcome extends React.Component {
3   //render函数并不做实际的渲染动作，他只是返回一个JSX
4   render() {
5     return <h1>Hello, {this.props.name}</h1>;
6   }
7 }

```

无论是函数式组件，还是类定义组件，最终组件return的都是React元素，而return的React元素(JSX)则又调用了React.createElement()

从return React元素到组件被实际渲染 挂到DOM树上 中间还有很复杂的过程。比如：在类组件中render函数被调用完之后，componentDidMount函数并不是会被立刻调用。componentDidMount被调用的时候，render函数返回的东西已经引发了渲染，组件已经被『装载』到了DOM树上

其实，使用类定义的组件，render方法是唯一必需的方法，其他组件的生命周期方法都只不过是给render服务而已，都不是必需的。

## 组件创建方式

目前，React支持三种方式来定义一个组件，分别是：

- ES5的React.createClass方式；
- ES6的React.Component方式；
- 无状态的函数组件方式。

组件是由元素构成的。元素数据结构是普通对象，而组件数据结构是类或纯函数。

- 类 组件：class extends React.Component()
- js函数式组件
- React.createClass()

1、类 组件 相比于函数式组件功能更强大。它有state，以及不同的生命周期方法，可以让开发者能够在组件的不同阶段（挂载、更新、卸载），对组件做更多的控制。

类组件可能是无状态组件，也可能是有状态组件。详见：[组件分类](#)

```

1 //组件名首字母必须大写
2 class Welcome extends React.Component{
3   //添加其他事件函数这么加：myWay(){...}
4   render(){
5     return (<div>my name is {this.props.name},{this.props.age}</div>);
6   }
7 };

```

2、函数式组件 JavaScript 函数构建的组件一定是无状态组件。它能传入props和context两个参数，没有state，除了render()，没有其它生命周期方法。

但正是这样，函数组件才更加专注和单一，它只是一个返回React 元素的函数，只关注对应UI的展现。函数组件接收外部传入的props，返回对应UI的DOM描述。

```
1 //组件名称总是以大写字母开始。
2 //定义模版的函数名首字母必须大写
3 function Welcome(props){
4     //添加其他事件函数这么加: function otherFun(){.....}
5     return <div>my name is {props.name},{props.age}</div>;
6 }
```

### 3、React.createClass()

```
1 /*组件首字母必须大写, eg:Greeting的G*/
2 var Greeting = React.createClass({
3     render: function() {
4         return <h1>Hello, {this.props.name}</h1>;
5     }
6 });
```

创建组件时需注意： 1、给组件命名时，组件首字母必须大写 2、render()方法的return 里，只能有一个html父标签，所以涉及到多个html标签时，必须外面再嵌套个父标签

## React中的组件

### 父子组件

### React props父组件给子组件传值

<https://www.cnblogs.com/yuyujuan/p/10125283.html>

```
1 父组件
2 import Parent from './Parent';
3
4 class Content extends Component{
5     constructor(props){
6         super(props);
7         this.state = {
8             url : {
9                 url : 'http://www.baidu.com',
10                 name : '百度',
11             },
12         };
13     }
14
15     render(){
```

```

16     return (<div>
17         /*父组件*/
18         <Parent url={ this.state.url }></Parent>
19     </div>);
20 }
21 }
22
23 //子组件
24 class Parent extends Component{
25     constructor(props){
26         super(props);
27         this.state = {};
28     }
29
30     render(){
31         return (<div>
32             <p><a href={ this.props.url.url }>{ this.props.url.name }</a></p>
33         </div>);
34     }
35 }
36

```

## 子组件调用父组件方法

```

1 //父组件
2 render(){
3     return (<div>
4         /*父组件*/
5         <Parent say={ this.sayHello }></Parent>
6     </div>);
7 }
8
9 sayHello = (params) => {
10     //父组件接受子组件传递的数据
11     console.log('父组件的sayHello方法');
12     console.log(params);
13 };
14
15 //子组件
16 render(){
17     return (<div>
18         /*<p><a href={ this.props.url.url }>{ this.props.url.name }</a></p>*/
19         <button onClick={ this.props.say.bind(this, 'xianjs') }>子组件调用父组件的方法</button>
20     </div>);
21 }

```

## 父组件中通过refs获取子组件属性和方法

```

1 //父组件

```



```

2  render(){
3      return (<div>
4          <Parent ref="parent"></Parent>
5          /*父组件*/ }
6          <button onClick={ this.getParentData }>执行父组件的方法和获取子组件的数据</button>
7      </div>);
8  }
9
10  getParentData = () => {
11      //执行子组件的方法
12      this.refs.parent.getParentName();
13      //获取子组件的数据
14      console.log(this.refs.parent.state.title);
15  };
16
17
18  //子组件
19  getParentName = () => {
20      console.log(this.state);
21  };
22
23  render(){
24      return (<div>
25          <h1>子组件 parent</h1>
26      </div>);
27  }

```

```

▶ {title: "子组件数组"}

```

```

子组件数组

```

```

>

```

## 在子组件中获取整个父组件

通过在父组件中获取整个子组件的实例，从而获取了组件的数据和方法，其实，在子组件中，也可以获取整个父组件的实例，从而获取父组件的数据和方法。

首要，父组件中定义数据和方法，并在调用子组件的时候，定义一个属性，传入this，即当前组件。

```
JS News.js x
1  import React,{Component} from 'react'
2  import Header from './Header'
3  class News extends Component{
4      constructor(props){
5          super(props);
6          this.state={
7              title:'我是父组件数据'
8          }
9      }
10     run=()=>{
11         console.log('我是父组件方法')
12     }
13
14     render(){
15         return(
16             <div>
17                 <h2>我是News.js，是一个父组件</h2>
18                 <Header news={this}/>
19             </div>
20         )
21     }
22
23 }
24 export default News;
```

然后在子组件中，可以直接使用这些数据和方法

```

1  import React,{Component} from 'react'
2  class Header extends Component{
3      constructor(props){
4          super(props);
5          this.state={
6              title:"我是子组件的数据"
7          }
8      }
9      getNews()=>{
10         this.props.news.run();
11         this.setState({
12             title:this.props.news.state.title
13         })
14     }
15     render(){
16         return(
17             <div>
18                 <h3>我是子组件</h3>
19                 <div>{this.state.title}</div>
20                 <button onClick={this.props.news.run}>获取父组件的方法</button>
21                 <br />
22                 <button onClick={this.getNews}>获取父组件的数据和方法</button>
23             </div>
24         )
25     }
26 }
27 export default Header;

```

当然了，这种情况下也可以很方便的将子组件的数据传递到父组件了，而不再需要通过在父组件中获取整个子组件了。

```

JS Header.js x
1  import React,{Component} from 'react'
2  class Header extends Component{
3      constructor(props){
4          super(props);
5          this.state={
6              title:"我是子组件的数据"
7          }
8      }
9      render(){
10         return(
11             <div>
12                 <h3>我是子组件</h3>
13                 <button onClick={this.props.news.getChildData.bind(this,this.state.title)}>子组件给父组件传值</button>
14             </div>
15         )
16     }
17 }
18 export default Header;

```

```

JS News.js x
1  import React,{Component} from 'react'
2  import Header from './Header'
3  class News extends Component{
4      constructor(props){
5          super(props);
6          this.state={
7              title:'我是父组件数据'
8          }
9      }
10     getChildData=(result)=>{
11         this.setState({
12             title:result
13         })
14     }
15     render(){
16         return(
17             <div>
18                 <h2>我是News.js，是一个父组件</h2>
19                 <div>{this.state.title}</div>
20                 <Header news={this}/>
21             </div>
22         )
23     }
24
25 }
26 export default News;

```

子组件中div里面的数据依赖于父组件传递过来的数据，那么当父组件没有给子组件传递数据时，子组件div里面就没有了数据了，这显然也不符合我们的预期，我们希望给子组件一个默认值，当父组件传递了数据过来时，就显示父组件传递的数据，当父组件没有传递数据时，子组件也能显示自己的默认值，这就时今天要说的defaultProps。

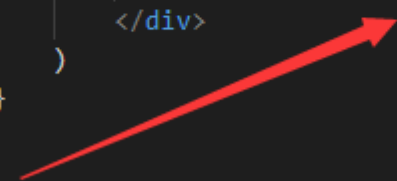
## defaultProps

defaultProps的用法就是，在父子组件传值中，如果父组件调用子组件的时候不给子组件传值，则可以在子组件中使用defaultProps定义的默认值。具体使用方法如下：

```

JS Header.js x
1  import React,{Component} from 'react'
2  class Header extends Component{
3      constructor(props){
4          super(props);
5          this.state={
6
7      }
8  }
9  render(){
10     return(
11         <div>
12             <h3>我是子组件</h3>
13             <div>{this.props.title}</div>
14         </div>
15     )
16 }
17
18 }
19 Header.defaultProps={
20     title:"默认值"
21 }
22 export default Header;

```



当父组件中没有传递数据时，显示的就是默认值，

```

<div>
  <h2>我是News.js，是一个父组件</h2>
  <Header />
</div>

```

**我是子组件**

默认值

当父组件中传递了数据时，显示的就是传递进来的数据值。

```

<div>
  <h2>我是News.js，是一个父组件</h2>
  <Header title={this.state.title}/>
</div>

```

**我是子组件**

我是父组件数据

## propTypes

在父子组件数据传递中，propTypes也经常用到，用于在子组件中限定子组件希望得到的数据类型。

在使用的时候，首先需要引入，然后再定义相关数据的类型：

```
JS Header.js x
1  import React,{Component} from 'react'
2  import propTypes from 'prop-types'
3  class Header extends Component{
4      constructor(props){
5          super(props);
6          this.state={}
7      }
8      render(){
9          return(
10             <div>
11                 <h3>我是子组件</h3>
12                 <div>{this.props.num}</div>
13             </div>
14         )
15     }
16 }
17
18 Header.propTypes={
19     num propTypes.number
20 }
21 export default Header;
```

那么当父组件传递的数据不是被期待的数据类型时，数据依然会显示，但是会给出一个警告：

```

JS News.js x
1  import React,{Component} from 'react'
2  import Header from './Header'
3  class News extends Component{
4      constructor(props){
5          super(props);
6          this.state={
7              num: '123'
8          }
9      }
10
11     render(){
12         return(
13             <div>
14                 <h2>我是News.js，是一个父组件</h2>
15                 <Header num={this.state.num}/>
16             </div>
17         )
18     }
19
20 }
21 export default News;

```

✖ Warning: Failed prop type: Invalid prop `num` of type `string` supplied to `Header`, expected `number`.  
 index.js:1452  
 in Header (at News.js:15)  
 in News (at App.js:11)  
 in div (at App.js:8)  
 in App (at src/index.js:7)

<https://www.cnblogs.com/yuyujuan/p/10125392.html>

## React生命周期函数

<https://www.cnblogs.com/yuyujuan/p/10125547.html>

在react中，生命周期函数指的是组件在加载前，加载后，以及组件更新数据和组件销毁时触发的一系列方法。通常分为以下几类：

1. 组件加载的时候触发的函数：constructor、componentWillMount、render、componentDidMount
2. 组件数据更新的时候触发的函数：shouldComponentUpdate、componentWillUpdate、render、componentDidUpdate
3. 在父组件里面改变props传值的时候触发的函数：componentWillReceiveProps
4. 组件销毁的时候触发的函数：componentWillUnmount

## 组件加载

```

JS Lifecycle.js x
1  import React , {Component} from 'react'
2  class Lifecycle extends Component{
3      constructor(props){
4          console.log('01构造函数');
5          super(props);
6          this.state={};
7      }
8      componentWillMount(){
9          console.log('02组件将要挂载');
10     }
11     componentDidMount(){
12         console.log('04组件挂载完成');
13     }
14
15     render(){
16         console.log('03数据渲染render');
17         return (
18             <div></div>
19         )
20     }
21 }
22 export default Lifecycle;

```

Download the React DevTools
01构造函数
02组件将要挂载
03数据渲染render
04组件挂载完成

当然了，在这个里面，构造函数和render并不属于生命周期函数部分，这里将它们放在一起，只是为了更好的展示函数的执行顺序。

需要注意的是，componentDidMount是组件挂在完成的时候触发的生命周期函数，所以通常**将DOM操作和数据请求都放在componentDidMount里面**。

## 组件数据更新



```

JS Lifecycle.js x
1  import React , {Component} from 'react'
2  class Lifecycle extends Component{
3      constructor(props){
4          super(props);
5          this.state={
6              msg: '我是改变前的msg的数据'
7          };
8      }
9      setMsg={()=>{
10         this.setState({
11             msg: '我是改变后的msg的数据'
12         })
13     }}
14     shouldComponentUpdate(){
15         console.log('01是否要更新数据');
16         return true;
17     }
18     componentWillUpdate(){
19         console.log('02组件将要更新');
20     }
21     componentDidUpdate(){
22         console.log('04组件数据更新完成');
23     }
24     render(){
25         console.log('03数据渲染render');
26         return (
27             <div>
28                 {this.state.msg}
29                 <br />
30                 <button onClick={this.setMsg}>更新msg的数据</button>
31             </div>
32         )
33     }
34 }

```

当我们点击按钮，更改组件数据时，会依次触发上面的函数。

01是否要更新数据
02组件将要更新
03数据渲染render
04组件数据更新完成

这个里面组要注意的是，shouldComponentUpdate表示是否更新数据，只有当返回true的时候才会执行更新数据的操作。

## 组件销毁

要控制组件的销毁，可以在父组件中，通过一个标志数据的布尔值来控制是否加载该组件，然后通过点击事件改变该标志数据的值，从而控制组件的加载和销毁。

```

5  class App extends Component {
6    constructor(props) {
7      super(props);
8      this.state = {
9        flag: true
10     }
11   }
12   setFlag = () => {
13     this.setState({
14       flag: !this.state.flag
15     })
16   }
17   render() {
18     return (
19       <div className="App">
20         这里是根组件
21         <br/>
22         <br/>
23         {
24           this.state.flag ? <Lifecycle /> : ""
25         }
26         <br/>
27         <br/>
28         <button onClick={this.setFlag}> 挂载和销毁声明周期函数组件 </button>
29       </div>
30     );
31   }
32 }

```

然后就可以在子组件中监听组件的加载和销毁了。

```

1  import React , {Component} from 'react'
2  class Lifecycle extends Component{
3      constructor(props){
4          console.log('01构造函数');
5          super(props);
6          this.state={
7              msg: '我是改变前的msg的数据'
8          };
9      }
10     componentWillMount(){
11         console.log('02组件将要挂载');
12     }
13     componentDidMount(){
14         console.log('04组件将要挂载');
15     }
16     componentWillUnmount(){
17         console.log('05组件销毁了');
18     }
19     render(){
20         console.log('03数据渲染render');
21         return (
22             <div>
23                 {this.state.msg}
24             </div>
25         )
26     }
27 }
28 export default Lifecycle;

```

01构造函数
02组件将要挂载
03数据渲染render
04组件将要挂载
05组件销毁了
01构造函数
02组件将要挂载
03数据渲染render
04组件将要挂载

## 父组件改变传值

父组件改变传值的时候，会触发相应的生命周期函数，因为数据的改变，也会触发组件数据更新的相关函数。

```
5  class App extends Component {
6    constructor(props){
7      super(props);
8      this.state={
9        title:'我是父组件的title'
10     }
11   }
12   setTitle={()=>{
13     this.setState({
14       title:'我是父组件改变后的title'
15     })
16   }
17   render() {
18     return (
19       <div className="App">
20         这里是根组件
21         <br/>
22         <br/>
23         <Lifecycle title={this.state.title} />
24         <br/>
25         <br/>
26         <button onClick={this.setTitle}>改变父组件title的值</button>
27       </div>
28     );
29   }
30 }
```

```

JS Lifecycle.js x
1  import React , {Component} from 'react'
2  class Lifecycle extends Component{
3      constructor(props){
4          console.log('01构造函数');
5          super(props);
6          this.state={
7              title: '我是app组件的title',
8          };
9      }
10     shouldComponentUpdate(nextProps, nextState){
11         console.log('01是否要更新数据');
12         console.log(nextProps);
13         console.log(nextState);
14         return true;
15     }
16     componentWillUpdate(){
17         console.log('02组件将要更新');
18     }
19     componentDidUpdate(){
20         console.log('04组件数据更新完成');
21     }
22     componentWillReceiveProps(){
23         console.log('父子组件传值，父组件里面改变了props的值触发的方法')
24     }
25     render(){
26         console.log('03数据渲染render');
27         return (
28             <div>
29                 {this.props.title}
30             </div>
31         )
32     }
33 }
34 export default Lifecycle;

```

这里是根组件

我是父组件的title

改变父组件title的值

Elements

top

Download the React D

01构造函数

03数据渲染render

>

当我们点击按钮改变父组件的传值的时候，相关函数的触发顺序如下：

父子组件传值，父组件里面改变了props的值触发的方法
01是否要更新数据
▶ {title: "我是父组件改变后的title"}
▶ {title: "我是app组件的title"}
02组件将要更新
03数据渲染render
04组件数据更新完成

这里需要说明的是shouldComponentUpdate这个函数，它有两个参数，当在组件内部改变数据的时候，第二个参数是改变后的数据值，当在父组件中改变数据的时候，第一个参数是改变后的值。

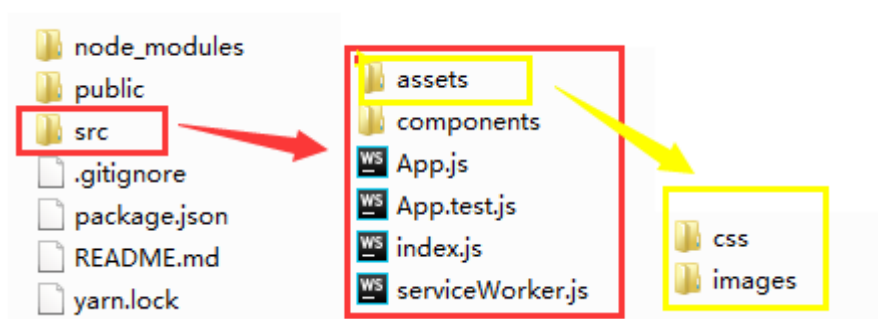
## react-router 4.x

<https://www.cnblogs.com/yuyujuan/p/10128703.html>

本次主要总结react中的路由的使用，实现让根组件根据用户访问的地址动态挂载不同的组件。

### 1, 创建项目

首先使用命令 `npx create-react-app react-router` 创建项目，然后 `npm install` 下载相关依赖，再按照之前的文件目录整理 `src` 文件夹，最后再 `components` 文件夹下面新建两个组件 `Home.js` 和 `News.js`。



```

1  import React, { Component } from 'react';
2
3  class Home extends Component {
4      constructor(props) {
5          super(props);
6          this.state = {  };
7      }
8      render() {
9          return (
10             <div>
11                 我是home组件
12             </div>
13          );
14      }
15  }
16
17  export default Home;

```

```

1  import React, { Component } from 'react';
2
3  class News extends Component {
4      constructor(props) {
5          super(props);
6          this.state = {  };
7      }
8      render() {
9          return (
10             <div>
11                 我是新闻组件
12             </div>
13          );
14      }
15  }
16
17  export default News;

```

## 2, 安装和引入路由

安装和引入路由可以分为以下几步：

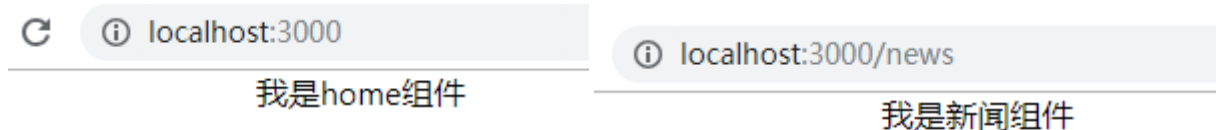
1. 安装路由：在项目根目录执行命令：`npm install react-router-dom -save`进行安装
2. 根组件进行引入 `import { BrowserRouter as Router, Route, Link } from "react-router-dom"`
3. 修改根组件文件App.js：在根元素中使用标签对路由组件进行包裹，然后使用组件。

```

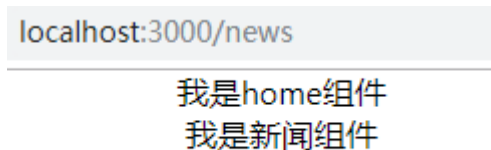
1  import React, { Component } from 'react';
2  import './assets/css/App.css';
3  import { BrowserRouter as Router, Route, Link } from "react-router-dom";
4  import Home from './components/Home';
5  import News from './components/News';
6
7  class App extends Component {
8    render() {
9      return (
10        <div className="App">
11          <Router>
12            <div>
13              <Route exact path="/" component={Home} />
14              <Route path="/news" component={News} />
15            </div>
16          </Router>
17        </div>
18      );
19    }
20  }
21
22  export default App;

```

根据上面的路由配置，当启动项目的时候，显示的是Home组件里面的内容，当我们更改地址栏，在其后面加入/news的后缀以后，就会显示News组件的内容。



在Home组件路由配置中，多了一个单词exact，这个意思是严格匹配，如果去掉这个单词，那么当在地址改为 <http://localhost:3000/news> 的时候，就会通过加载两个组件，因为/news也能匹配/这个路径。



### 3，使用路由

我们在引入路由的时候，一共引入了三个组件Router, Route, Link，接下来及来使用这最后一个组件。

在实际运用中，并不会通过手动修改地址栏来进行页面切换，一般都是通过点击事件触发的，在react中，可以借助Link实现a标签进行地址跳转的功能，如下所示，只需要稍微修改根组件App.js就可以了。



```

1  import React, { Component } from 'react';
2  import './assets/css/App.css';
3  import { BrowserRouter as Router, Route, Link } from "react-router-dom";
4  import Home from './components/Home';
5  import News from './components/News';
6
7  class App extends Component {
8    render() {
9      return (
10       <div className="App">
11         <Router>
12           <div>
13             <header className="title">
14               <Link to="/">首页</Link>
15               <Link to="/news">新闻</Link>
16             </header>
17             <Route exact path="/" component={Home} />
18             <Route path="/news" component={News} />
19           </div>
20         </Router>
21       </div>
22     );
23   }
24 }
25
26 export default App;

```

首页 新闻

我是home组件

现在当我们点击不同的标签，就会加载不同的页面了。下面贴出app.js的代码：



```

1  import React, { Component } from 'react';
2  import './assets/css/App.css';
3  import { BrowserRouter as Router, Route, Link } from "react-router-dom";
4  import Home from './components/Home';
5  import News from './components/News';
6
7  class App extends Component {
8    render() {
9      return (
10       <div className="App">
11         <Router>
12           <div>
13             <header className="title">
14               <Link to="/">首页</Link>
15               <Link to="/news">新闻</Link>

```

```
16         </header>
17         <Route exact path="/" component={Home} />
18         <Route path="/news" component={News} />
19     </div>
20 </Router>
21 </div>
22 );
23 }
24 }
25
26 export default App;
```

## 路由传值

---

<https://www.cnblogs.com/yuyujuan/p/10129098.html>

## React数据请求

---

<https://www.cnblogs.com/yuyujuan/p/10134020.html>

## React路由模块化

---

<https://www.cnblogs.com/yuyujuan/p/10146421.html>

## redux & react-redux

---

在vue中，可以使用vuex进行数据管理，在react中，可以使用redux进行数据管理。redux主要由Store、Reducer和Action组成：

- Store：状态载体，访问状态、提交状态、监听状态变更
- Reducer：状态更新具体执行者，纯函数
- Action：存放数据的对象，即消息的载体，只能被别人操作，自己不能进行任何操作

## 简单使用

---

在redux中，首先需要了解的是store，所有的数据都在这一个数据源里面进行管理，具有全局唯一性，但是redux本身和react并没有直接的联系，可以单独使用，复杂的项目才需要redux来管理数据，简单的项目，state+props+context就足够了。

例如，我们想要实现一个简单的累加器，就需要以下几步：

1. 用来存储数据的store，store里面的state是数据放置的位置
2. 通过dispatch一个action来提交对数据的修改
3. 请求提交到reducer函数里，根据传入的action和state，返回新的state

首先新建项目，然后执行命令npm install redux --save安装redux。

其次，在src文件夹下面新建store.js，创建store，然后根据action的不同类型，执行不同的操作：

store.js



```
1  import {createStore} from 'redux';
2
3  const counterReducer = (state = 0, action) => {
4    switch(action.type){
5      case 'add':
6        return state + 1;
7      case 'minus':
8        return state - 1;
9      default:
10       return state;
11    }
12  }
13
14  export default createStore(counterReducer)
```



然后在components文件夹下面新建Test.js组件，并在组件中引入store.js

Test.js



```
1  import React, {Component} from 'react'
2  import store from '../store'
3
4  class Test extends Component{
5    render(){
6      return (
7        <div>
8          <p>{store.getState()}</p>
9          <button onClick={()=>store.dispatch({type:'add'})}> + </button>
10         <button onClick={()=>store.dispatch({type:'minus'})}> - </button>
11        </div>
12      )
13    }
14  }
15
16  export default Test;
```

最后使用组件，使用组件分两步：初始化渲染时，需要请求初始化的数据；后面每次数据改变时，重新加载数据。

index.js

```

1 // 初始化执行
2 ReactDOM.render(<Test />, document.getElementById('root'));
3 // 每次发生变化时执行
4 store.subscribe(()=>{
5     ReactDOM.render(<Test />, document.getElementById('root'));
6 })

```

## react-redux

如果在大型项目中，我们每次都在需要使用地方重新调用render，会十分麻烦，所以，需要使用更简洁的方法：react-redux，react-redux提供了两个api：提供数据的顶级组件Provider和提供数据与方法的高阶组件connect。

首先，要实现react-redux，需要先进行安装：npm install react-redux --save

其次，既然要用到高阶组件，就需要使用高阶组件装饰器：npm install --save-dev babel-plugin-transform-decorators-legacy，具体的可以参考前面的[react高阶组件](#)。

最后，来改写上面的累加器组件。

1，在index.js中引入Provider组件，并进行相应的修改，这样，后面就不需要再每个需要的页面多次引入store.js了，更不用在每次操作了数据以后重新render。

index.js

```

1 import store from './store';
2 import Test from './components/Test';
3 import {Provider} from 'react-redux'
4
5
6 ReactDOM.render((
7     <Provider store={store}>
8         <Test />
9     </Provider>
10 ), document.getElementById('root'));
11
12 serviceWorker.unregister();

```

2，在Test.js页面，重新更改写法，使用高阶组件connect来提供数据和方法：

```

1 import React, {Component} from 'react'
2 import { connect } from "react-redux";
3
4 @connect(
5     state => ({ num: state }), // 状态映射
6     {
7         add: () => ({ type: "add" }),
8         minus: () => ({ type: "minus" })
9     }
10 )

```

```
11
12 class Test extends Component{
13     render(){
14         return (
15             <div>
16                 <p>{this.props.num}</p>
17                 <button onClick={()=>this.props.add()}> + </button>
18                 <button onClick={()=>this.props.minus()}> - </button>
19             </div>
20         )
21     }
22 }
23
24 export default Test;
```

## 一个小案例，巩固有状态组件和无状态组件的使用

### 通过for循环生成多个组件

1. 数据：

```
1  CommentList = [
2    { user: '张三', content: '哈哈，沙发' },
3    { user: '张三2', content: '哈哈，板凳' },
4    { user: '张三3', content: '哈哈，凉席' },
5    { user: '张三4', content: '哈哈，砖头' },
6    { user: '张三5', content: '哈哈，楼下山炮' }
7  ]
```

### style样式

## 总结

理解React中虚拟DOM的概念 理解React中三种Diff算法的概念 使用JS中createElement的方式创建虚拟DOM 使用ReactDOM.render方法 使用JSX语法并理解其本质 掌握创建组件的两种方式 理解有状态组件和无状态组件的本质区别 理解props和state的区别

## 相关文章

- [React数据流和组件间的沟通总结](#)
- [单向数据流和双向绑定各有什么优缺点?](#)
- [怎么更好的理解虚拟DOM?](#)
- [React中文文档 - 版本较低](#)
- [React 源码剖析系列 - 不可思议的 react diff](#)
- [深入浅出React（四）：虚拟DOM Diff算法解析](#)
- [一看就懂的ReactJs入门教程（精华版）](#)
- [CSS Modules 用法教程](#)
- [将Markdown转换为HTML页面](#)
- [win7命令行 端口占用 查询进程号 杀进程](#)