# 

# React入门

[React入门 1](#_Toc20937)

[一React 1](#_Toc5887)

[1.1 为什么使用React？ 1](#_Toc16671)

[1.2 React中的基本概念 2](#_Toc3880)

[1.3 React 特点 2](#_Toc21189)

[1.4 React适用范围 4](#_Toc1526)

[1.5 React jsx 4](#_Toc7354)

[1.6 React组件 8](#_Toc2069)

[1.7 如何定义一个的组件（ES6语法，以后都是这种创建方式） 10](#_Toc13568)

[1.8 React组件属性props 12](#_Toc10109)

[1.9 React状态state 19](#_Toc20928)

[2.0 组件生命周期相关 24](#_Toc17783)

[2.1事件和DOM操作 26](#_Toc27581)

[2.2 redux,router基础，redux进阶 30](#_Toc10714)

[2.3 redux中间件 30](#_Toc32041)

[2.4 redux异步操作 30](#_Toc14580)

[2.5 router深入 30](#_Toc2140)

[参考资料： 30](#_Toc26111)

### 

### 1.1 为什么使用React？

狭义来讲 React 是 Facebook 内部开源出来的一个前端 UI 开发框架，广义来讲 React 不仅仅是 js 框架本身，更是一套完整的前端开发生态体系，这套体系包括：

1、React.js

2、ReactRenders: ReactDOM / ReactServer / ReactCanvas

3、Flux 模式及其实现（Redux , Fluxxor）

4、React 开源组件

5、React Native

6、其他

学习 React 并不仅仅是学习 React 本身，而是学习这套开发体系，整个技术栈。

React 是为了解决一个问题：构建随着时间数据不断变化的大规模应用程序。

当底层的数据变了，React 会自动处理所有用户界面的更新，数据变化后，React 概念上与点击“刷新”按钮类似，但仅会更新变化的部分。

React 都是关于构建可复用的组件。事实上，通过 React 你唯一要做的事情就是构建组件。得益于其良好的封装性，组件使代码复用、测试和关注分离（separation of concerns）更加简单。

### 1.2 React中的基本概念

**React.js**

React.js 是 React 的核心库，在应用中必须先加载核心库。

**ReactDOM.js**

ReactDOM.js 是 React 的 DOM 渲染器，React 将核心库和渲染器分离开了，为了在 web 页面中显示开发的组件，需要调用 ReactDOM.render 方法， 第一个参数是 React 组件，第二个参数为 HTMLElement。

**JSX**

JSX 是 React 自定义的语法，最终 JSX 会转化为 JS 运行于页面当中。

**组件**

组件是 React 中的核心概念，页面当中的所有元素都是通过 React 组件来表达， 我们将要写的 React 代码绝大部分都是在做 React 组件的开发。

**Virtual DOM**

React 抽象出来的虚拟 DOM 树，虚拟树是 React 高性能的关键。

**单向数据流**：one-way reactive data flow

React 应用的**核心设计模式**，数据流向自顶向下



看一个简单的案例（官方的）

响应式更新 (Reactive Updates)

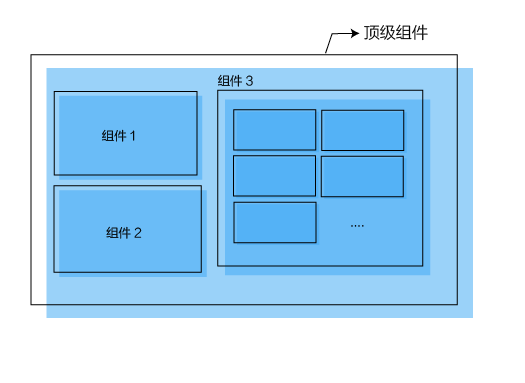
在浏览器中打开 1.html ，在输入框输入你的名字。你会发现 React 在用户界面中只改变了时间，你在输入框的输入内容会保留着，即使你没有写任何代码来完成这个功能。React 也为你解决了这个问题，做了正确的事。

我们想到的解决方案是React 是不会去操作 DOM 的，除非不得不操作 DOM 。它用一种更快的内置仿造的 DOM 来操作差异，为你计算出效率最高的 DOM 改变。

这个组件的输入被称为 props - "properties"的缩写。它们通过 JSX 语法进行参数传递。你必须知道，在组件里这些属性是不可直接改变的，也就是说 this.props 是只读的

### 1.3 React 特点

**1、组件的组合模式（web component）**



基于组合模式的优点：

构建可重用的组件：组件的开发能够形成公司的组件库，每个业务的开发都能积累可重用的组件。

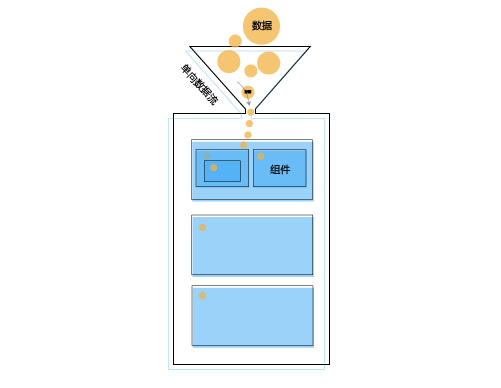
无学习障碍：天然符合 HTML 的结构， 对前端开发者来说几乎没有学习障碍。

具有弹性的架构：组合模式很简单却很有效，能够构建简单的页面也能构建大型的前端应用。

源码高可维护性：开发只是工作中的一部分，应用的上线才是噩梦的开始，很多大型应用因为复制的业务逻辑导致无法快速响应业务需求，可维护性低。

**2、单向数据流的设计**（属性数据从父组件传递）

React 的单向数据流的设计让前端 bug 定位变得简单， 页面的 UI 和数据的对应是唯一的



3、**高效的性能**（虚拟Dom,更新时只更新变化的部分）。

4、**分离的设计**

React.js 现在的版本已经将源码分开为 ReactDOM 和 React.js . 这就意味着 React 不仅仅能够在 web 端工作， 甚至可以在服务端（nodejs），Native 端运行。

与此同时， 我们可以自定义自己的渲染器， 实现比如 Three.js， Pixi.js， D3.js 的 React 方式渲染。

### 1.4 React适用范围

1、web 端应用

2、原生应用 - IOS、Android、Native 应用

3、Node.Js 服务端渲染

### 1.5 React jsx

JSX 让你可以用 HTML 语法去写 JavaScript 函数调用。 为了在 React 生成一个链接，通过纯 JavaScript 你可以这么写：

React.createElement('a', {href: 'http://facebook.github.io/react/'}, 'Hello React!')

通过 JSX 这就变成了

<a href="http://facebook.github.io/react/">Hello React!</a>

我们发现这会使搭建 React 应用更加简单，设计师也偏向用这种语法，但是每个人都有自己的工作流，所以JSX 并不强制必须使用的。

1. **HTML标签与React组件对比**

React 可以渲染 HTML 标签 (strings) 或 React 组件 (classes)。

要渲染 HTML 标签，只需在 JSX 里使用小写字母开头的标签名。

var myDivElement = <div className="foo" />;

React.render(myDivElement, document.body);

要渲染 React 组件，只需创建一个大写字母开头的本地变量。

var MyComponent = React.createClass({/\*...\*/});

var myElement = <MyComponent someProperty={true} />;

React.render(myElement, document.body);

React 的 JSX 里约定分别使用首字母大、小写来区分本地组件的类和 HTML 标签。

JSX把类XML 的语法转成纯粹JavaScript，XML元素、属性和子节点被转换成 React.createElement 的参数。

var Nav;

// 输入 (JSX):

var app = <Nav color="blue" />;

// 输出 (JS):

var app = React.createElement(Nav, {color:"blue"});

**2、JavaScript 表达式**

属性表达式

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 ("")。

// 输入 (JSX):

var person = <Person name={window.isLoggedIn ? window.name : ''} />;

// 输出 (JS):

var person = React.createElement(Person,{name: window.isLoggedIn ? window.name : ''});

**子节点表达式**

同样地，JavaScript 表达式可用于描述子结点：

// 输入 (JSX):

var content = <Container>{window.isLoggedIn ? <Nav /> : <Login />}</Container>;

// 输出 (JS):

var content = React.createElement(

Container,

null,

window.isLoggedIn ? React.createElement(Nav) : React.createElement(Login)

);

**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。

var content = (

<Nav>

{/\* 一般注释, 用 {} 包围 \*/}

<Person

/\* 多

行

注释 \*/

name={window.isLoggedIn ? window.name : ''} // 行尾注释

/>

</Nav>

);

**简单例子 #**

var LikeButton = React.createClass({

getInitialState: function() {

return {liked: false};

},

handleClick: function(event) {

this.setState({liked: !this.state.liked});

},

render: function() {

var text = this.state.liked ? 'like' : 'haven\'t liked';

return (

<p onClick={this.handleClick}>

You {text} this. Click to toggle.

</p>

);

}

});

React.render(

<LikeButton />,

document.getElementById('example')

);

创建 JSX 语法的本质目的是为了使用基于 xml 的方式表达组件的嵌套，保持和 HTML 一致的结构，语法上除了在描述组件上比较特别以外，其它和普通的 Javascript 没有区别。 并且最终所有的 JSX 都会编译为原生 Javascript。

标签可以任意的嵌套

function render() {

return <p>

text content

<ul>

<li>....</li>

<li>....</li>

</ul>

</p>

}

**标签闭合** 标签必须严格闭合，否则无法编译通过

function render() {

return <input type="text"/>

}

function render() {

return <p>....</p>

}

**定义一个****自定义组件**

var CustomComponnet = React.createClass({

render: function() {

return <div> custom component </div>

}

});

**使用自定义组件**

function render() {

return <p> <CustomComponent/> </p>

}

**组件属性**

和 html 一样，JSX 中组件也有属性，传递属性的方式也相同

对于 HTML 组件

function render() {

return <p title="title" >hello, React, world </p>

}

如果是 React 组件可以定义自定义属性，传递自定义属性的方式

function render() {

return <p> <CustomComponent customProps="data"/> </p>

}

}

function render() {

var data = {a: 1, b:2};

return <p> <CustomComponent customProps={data}/> </p>

}

需要注意的地方上，属性的写法上和 HTML 存在区别，在写 JSX 的时候，所有的属性都是驼峰式的写法，如：

function render() {

return <div className="...">

<label htmlFor="..."></label>

<input onChange="..."/>

</div>

}

### 1.6 React组件

一个React应用就是构建在 React组件之上的。

1、组件有两个核心概念：

props

state

一个组件就是通过这两个属性的值在render方法里面生成这个组件对应的 HTML 结构。

注意：组件生成的HTML结构只能有一个单一的根节点。

2、props

props就是组件的属性，由外部通过 JSX 属性传入设置，一旦初始设置完成，就可以认为this.props是不可更改的，所以不要轻易更改设置this.props里面的值（虽然对于一个JS对象你可以做任何事）。

3、state

state 是组件的当前状态，可以把组件简单看成一个“状态机”，根据状态state呈现不同的UI展示。

一旦状态（数据）更改，组件就会自动调用render 重新渲染UI，这个更改的动作会通过this.setState方法来触发。

使用组件的目的就是通过构建模块化的组件，相互组合组件最后组装成一个复杂的应用。在React组件中要包含其他组件作为子组件，只需要把组件当作一个DOM元素引 入就可以了。

4、无状态组件

你也可以用纯粹的函数来定义无状态的组件(stateless function)，这种组件没有状 态，没有生命周期，只是简单的接受props渲染生成DOM结构。无状态组件非常 简单，开销很低，如果可能的话尽量使用无状态组件。比如使用箭头函数定义：

const HelloMessage = (props) => <div> Hello {props.name}</div>; render(<HelloMessage name="John" />, mountNode);

因为无状态组件只是函数，所以它没有实例返回。

案例：



5、组件命名空间

React.createClass 生成的组件类为一个 Javascript 对象。 当我们想设置命名空间组件时，可以在组件下面添加子组件：

eg:

MyComponent.SubComponent = React.createClass({...});

MyComponent.SubComponent.Sub = React.createClass({....});

在组件较多的情况下，可以借助命名空间优化组件维护结构以及解决组件名称冲突问题。

### 1.7 如何定义一个的组件（ES6语法，以后都是这种创建方式）

**组件组成**

1.定义

一般来说，一个组件类由 extends Component 创建，并且提供一个 render 方法以及其他可选的生命周期函数、组件相关的事件或方法来定义。

2.一个标准组件的组织结构

一般来说一个结构良好好维护的组件应该遵循下面这样的结构和顺序

1 class definition

1.1 constructor //构造函数

1.1.1 event handlers // 定义 eventHandler

1.2 'component' lifecycle events //生命周期函数

1.3 getters //一些获取属性或者状态的get方法

1.4 render //render方法

2 defaultProps //定义组件默认的属性

3 proptypes //组件属性校验

* 举例

（ES5的继承，实质是先创造子类的实例对象this，然后再将父类的方法添加到this上面（Parent.apply(this)）。ES6的继承机制完全不同，实质是先创造父类的实例对象this（所以必须先调用super方法），然后再用子类的构造函数修改this。）

//简单的代码举例

class Person extends React.Component {

//构造函数

constructor (props) {

super(props);

this.state = { smiling: false };

this.handleClick = () => {

this.setState({smiling: !this.state.smiling});

};

// 定义 eventHandler

this.handleSave = this.handleSave.bind(this);

}

//生命周期函数

componentWillMount () {

// add event listeners (Flux Store, WebSocket, document, etc.)

}

//生命周期函数

componentDidMount () {

// React.getDOMNode()

}

//生命周期函数

componentWillUnmount () {

// remove event listeners (Flux Store, WebSocket, document, etc.)

}

//自定义的get函数

get smilingMessage () {

return (this.state.smiling) ? "is smiling" : "";

}

//render

render () {

return (

<div onClick={this.handleClick}>

{this.props.name} {this.smilingMessage}

</div>

);

}

}

//定义组件默认的属性值

Person.defaultProps = {

name: 'Guest'

};

//校验从父组件传递的属性值是否符合

Person.propTypes = {

name: React.PropTypes.string

};

### 1.8 React组件属性props

* 基础概念

**React属性应该关注的点**

this.props对象的属性与组件的属性一一对应，但是有一个例外，就是this.props.children 属性。它表示组件的所有子节点.

###### **this.props.children 的值有三种可能：**

* + 如果当前组件没有子节点，它就是undefined ;
  + 如果有一个子节点，数据类型是object；
  + 如果有多个子节点，数据类型就是array；  
    所以，处理 this.props.children 的时候要小心。

React 提供一个工具方法 [React.Children([https://facebook.github.io/react/docs/top-level-api.html#react.children](https://facebook.github.io/react/docs/top-level-api.html" \l "react.children)) 来处理 this.props.children 。我们可以用 React.Children.map 来遍历子节点

而不用担心 this.props.children 的数据类型是 undefined 还是 object。更多的 React.Children 的方法。

#### **PropTypes**

组件的属性可以接受任意值，字符串、对象、函数等等都可以。有时，我们需要一种机制，验证别人使用组件时，提供的参数是否符合要求。

组件类的PropTypes属性，就是用来验证组件实例的属性是否符合要求，从组件定义模板里能发现如何使用

propTypes: {

title: React.PropTypes.string.isRequired,

}

基本使用方法：

propTypes: {

myArray: React.PropTypes.array,

myBool: React.PropTypes.bool,

myFunc: React.PropTypes.func,

myNumber: React.PropTypes.number,

myString: React.PropTypes.string，

// You can chain any of the above with `isRequired` to make sure a warning

// is shown if the prop isn't provided.

requiredFunc: React.PropTypes.func.isRequired

}

假如我们props不是以上类型，而是拥有复杂结构的对象怎么办？比如下面这个：

{

text: 'hello world',

numbers: [5, 2, 7, 9],

}

当然，我们可以直接用React.PropTypes.object,但是对象内部的数据我们却无法验证。

propTypes: {

myObject: React.PropTypes.object,

}

进阶使用方法：shape()和arrayOf()

propTypes: {

myObject: React.PropTypes.shape({

text: React.PropTypes.string,

numbers: React.PropTypes.arrayOf(React.PropTypes.number),

})

}

下面是一个更复杂的Props：

[

{

name: 'Zachary He',

age: 13,

married: true,

},

{

name: 'Alice Yo',

name: 17,

},

{

name: 'Jonyu Me',

age: 20,

married: false,

}

]

综合上面，写起来应该就不难了：

propTypes: {

myArray: React.PropTypes.arrayOf(

React.propTypes.shape({

name: React.propTypes.string.isRequired,

age: React.propTypes.number.isRequired,

married: React.propTypes.bool

})

)

}

* 详细的类型校验

var MyComponent = React.createClass({

propTypes: {

optionalArray: React.PropTypes.array,

optionalBool: React.PropTypes.bool,

optionalFunc: React.PropTypes.func,

optionalNumber: React.PropTypes.number,

optionalObject: React.PropTypes.object,

optionalString: React.PropTypes.string,

// 任何可以被渲染的包括，数字，字符串，组件，或者数组

optionalNode: React.PropTypes.node,

// React 元素

optionalElement: React.PropTypes.element,

// 枚举

optionalEnum: React.PropTypes.oneOf(['News', 'Photos']),

// 任意一种类型

optionalUnion: React.PropTypes.oneOfType([

React.PropTypes.string,

React.PropTypes.number,

React.PropTypes.instanceOf(Message)

]),

// 具体类型的数组

optionalArrayOf: React.PropTypes.arrayOf(React.PropTypes.number),

// 具体类型的对象

optionalObjectOf: React.PropTypes.objectOf(React.PropTypes.number),

// 符合定义的对象

optionalObjectWithShape: React.PropTypes.shape({

color: React.PropTypes.string,

fontSize: React.PropTypes.number

}),

requiredFunc: React.PropTypes.func.isRequired,

requiredAny: React.PropTypes.any.isRequired,

// 自定义校验

customProp: function(props, propName, componentName) {}

}

});

React.createClass({

propTypes: {

// 可以声明 prop 为指定的 JS 基本类型。默认

// 情况下，这些 prop 都是可传可不传的。

optionalArray: React.PropTypes.array,

optionalBool: React.PropTypes.bool,

optionalFunc: React.PropTypes.func,

optionalNumber: React.PropTypes.number,

optionalObject: React.PropTypes.object,

optionalString: React.PropTypes.string,

// 所有可以被渲染的对象：数字，

// 字符串，DOM 元素或包含这些类型的数组。

optionalNode: React.PropTypes.node,

// React 元素

optionalElement: React.PropTypes.element,

// 用 JS 的 instanceof 操作符声明 prop 为类的实例。

optionalMessage: React.PropTypes.instanceOf(Message),

// 用 enum 来限制 prop 只接受指定的值。

optionalEnum: React.PropTypes.oneOf(['News', 'Photos']),

// 指定的多个对象类型中的一个

optionalUnion: React.PropTypes.oneOfType([

React.PropTypes.string,

React.PropTypes.number,

React.PropTypes.instanceOf(Message)

]),

// 指定类型组成的数组

optionalArrayOf: React.PropTypes.arrayOf(React.PropTypes.number),

// 指定类型的属性构成的对象

optionalObjectOf: React.PropTypes.objectOf(React.PropTypes.number),

// 特定形状参数的对象

optionalObjectWithShape: React.PropTypes.shape({

color: React.PropTypes.string,

fontSize: React.PropTypes.number

}),

// 以后任意类型加上 `isRequired` 来使 prop 不可空。

requiredFunc: React.PropTypes.func.isRequired,

// 不可空的任意类型

requiredAny: React.PropTypes.any.isRequired,

// 自定义验证器。如果验证失败需要返回一个 Error 对象。不要直接

// 使用 `console.warn` 或抛异常，因为这样 `oneOfType` 会失效。

customProp: function(props, propName, componentName) {

if (!/matchme/.test(props[propName])) {

return new Error('Validation failed!');

}

}

},

/\* ... \*/

});

#### **getDefaultProps 定义默认属性**

只在组件创建时调用一次并缓存返回的对象（即在 React.createClass 之后就会调用）。

因为这个方法在实例初始化之前调用，所以在这个方法里面不能依赖 this 获取到这个组件的实例。

在组件装载之后，这个方法缓存的结果会用来保证访问 this.props 的属性时，当这个属性没有在父组件中传入（在这个组件的 JSX 属性里设置），也总是有值的。

getDefaultProps : function () {

return {

title : 'Hello World'

};

}

如果是使用 ES6 语法，可以直接定义 defaultProps 这个类属性来替代，这样能更直观的知道 default props 是预先定义好的对象值：

Counter.defaultProps = { initialCount: 0 };

### 1.9 React状态state

#### **基础**

#### 

#### ****关注点****

* React 的渲染结果是由组件属性和状态共同决定的，状态和属性的区别是，状态维护在组件内部，属性是由外部控制，我们先介绍组件状态相关细节：

控制状态的 API 为：  
1. this.state：组件的当前状态

(React.component方式不能用)getInitialState：获取组件的初始状态，在组件加载的时候会被调用一次，返回值赋予 ****this.state**** 作为初始值

this.setState：

组件状态改变时，可以通过 this.setState 修改状态

setState 方法支持按需修改，如 state 有两个字段，仅当 setState 传入的对象包含字段 key 才会修改属性

每次调用 setState 会导致重渲染调用 render 方法

直接修改 state 不会重渲染组件

**关系到状态改变和显示逻辑判断的点**

把计算和条件判断都交给 render()

##### **1. 组件的state中不能出现props**

// 不好的写法:

constructor (props) {

this.state = {

fullName: `${props.firstName} ${props.lastName}`

};

}

render () {

var fullName = this.state.fullName;

return (

<div>

<h2>{fullName}</h2>

</div>

);

}

// 好的写法:

render () {

var fullName = `${this.props.firstName} ${this.props.lastName}`;

}

当然，复杂的display logic也应该避免全堆放在render()中，因为那样可能导致整个render()方法变得臃肿，不优雅。我们可以把一些复杂的逻辑通过helper function移出去。

// GOOD: helper function

renderFullName () {

return `${this.props.firstName} ${this.props.lastName}`;

}

render () {

var fullName = this.renderFullName();

}

##### **2. 保持state的简洁，不要出现计算得来的state**

// 错误的写法，不要这么写:

constructor (props) {

this.state = {

listItems: [1, 2, 3, 4, 5, 6],

//属性通过数组长度计算出来的

itemsNum: this.state.listItems.length

};

}

render() {

return (

<div>

<span>{this.state.itemsNum}</span>

</div>

)

}

// 正确的写法，定义在render方法中使用:

render () {

var itemsNum = this.state.listItems.length;

}

##### **3. 能用三元判断符，就不用If，直接放在render()里**

// 不好的写法，太啰嗦:

renderSmilingStatement () {

if (this.state.isSmiling) {

return <span>is smiling</span>;

}else {

return '';

}

},

render () {

return <div>{this.props.name}{this.renderSmilingStatement()}</div>;

}

// 简介明了的写法:

render () {

return (

<div>

{this.props.name}

{(this.state.smiling)

? <span>is smiling</span>

: null

}

</div>

);

}

##### **布尔值都不能搞定的，交给IIFE（即时调用函数）**

return (

<section>

<h1>Color</h1>

<h3>Name</h3>

<p>{this.state.color || "white"}</p>

<h3>Hex</h3>

<p>

{(() => {

switch (this.state.color) {

case "red": return "#FF0000";

case "green": return "#00FF00";

case "blue": return "#0000FF";

default: return "#FFFFFF";

}

})()}

</p>

</section>

);

##### **5. 不要把display logic（和显示相关的逻辑判断）写在componentWillReceiveProps或componentWillMount中，把它们都移到render()中去。**

### 2.0 组件生命周期相关

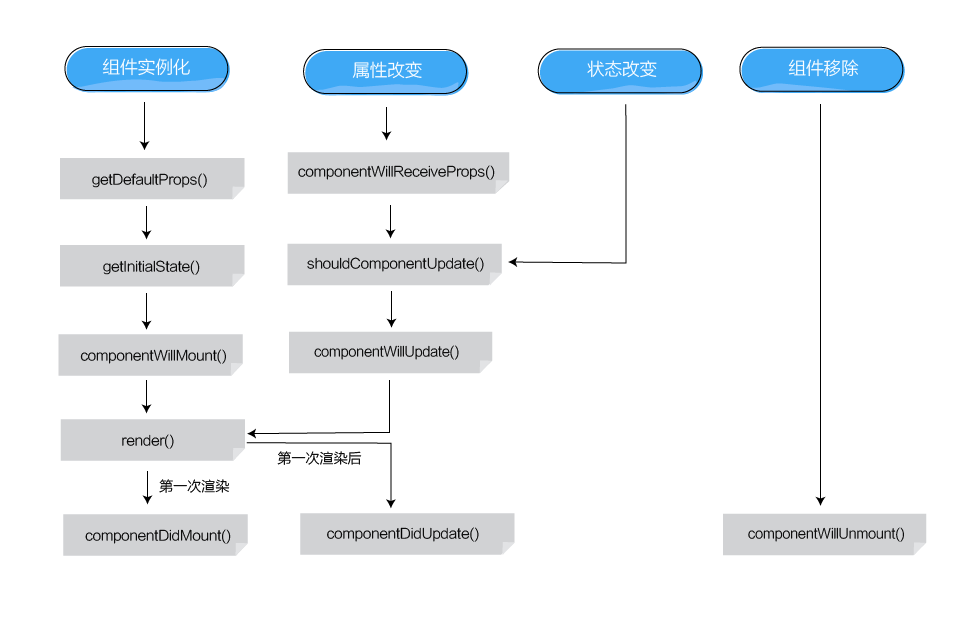
组件的生命周期的变化，包含的主要几种情况：

组件被实例化的时候

组件属性改变的时候

组件状态被改变的时候

组件被销毁的时候

详细请看下图：  
[](https://sfault-image.b0.upaiyun.com/251/432/2514324564-5743bce54f496)

**componentWillMount**

void componentWillMount()

****条件****：第一次渲染阶段在调用 render 方法前会被调用  
****作用****：该方法在整个组件生命周期只会被调用一次，所以可以利用该方法做一些组件内部的初始化工作

**componentDidMount**

void componentDidMount()

****条件****：第一次渲染成功过后，组件对应的 DOM 已经添加到页面后调用  
****作用****：这个阶段表示组件对应的 DOM 已经存在，我们可以在这个时候做一些依赖 DOM 的操作或者其他的一些如请求数据，和第三方库整合的操作。如果嵌套了子组件，子组件会比父组件优先渲染，所以这个时候可以获取子组件对应的 DOM。

**componentWillReceiveProps(newProps)**

void componentWillReceiveProps(

object nextProps

)

****条件：**** 当组件获取新属性的时候，第一次渲染不会调用  
****用处：**** 这个时候可以根据新的属性来修改组件状态

**shouldComponentUpdate(nextProps, nextState)**

boolean shouldComponentUpdate(

object nextProps, object nextState

)

****条件：****接收到新属性或者新状态的时候在 render 前会被调用（除了调用 forceUpdate 和初始化渲染以外）  
****用处：**** 该方法让我们有机会决定是否重渲染组件，如果返回 false，那么不会重渲染组件，借此可以优化应用性能（在组件很多的情况）。  
**componentWillUpdate**

void componentWillUpdate(

object nextProps, object nextState

)

****条件****：当组件确定要更新，在 render 之前调用  
****用处****：这个时候可以确定一定会更新组件，可以执行更新前的操作  
****注意****：方法中不能使用 setState ，setState 的操作应该在 componentWillReceiveProps 方法中调用

**componentDidUpdate**

void componentDidUpdate(

object prevProps, object prevState

)

****条件****：更新被应用到 DOM 之后  
****用处****：可以执行组件更新过后的操作

### 2.1事件和DOM操作

**1、操作DOM**

DOM 真正被添加到 HTML 中的 hook 为

componentDidMount

componentDidUpdate

在这两个 hook 函数中， 我们可以获取真正的DOM元素。

findDOMNode()

通过 ReactDOM 提供的 findDOMNode 方法，传入参数

var MyComponent = React.createClass({

render: function() {

return <div> .... </div>

},

componentDidMount: function() {

var $root = ReactDOM.findDOMNode(this);

console.log($root);

}

})

需要注意的是此方法不能应用到无状态组件上

上面的方法只能获取到 root 元素，那如果我的 DOM 有很多层级，我想获取一个子级的元素呢？React 提供了 ref 属性来实现这种需求。

每个组件实例都有一个 this.refs 属性，会自动引用所有包含 ref 属性组件的 DOM

var MyComponent = React.createClass({

render: function() {

return <div>

<button ref="btn">...</button>

<a href="" ref="link"></a>

</div>

},

componentDidMount: function() {

var $btn = this.refs.btn;

var $link = this.refs.link;

console.log($btn, $link);

}

})

**2、绑定事件**

在 React 中绑定事件的方式很简单，只需要在元素中添加事件名称的属性已经对应的处理函数，如：

var MyComponent = React.creatClass({

render: function() {

return <div>

<button onClick={this.onClick}>Click Me</button>

</div>

},

onClick: function() {

console.log('click me');

}

});

事件名称和其他属性名称一样，服从驼峰式命名。

onChange 事件

和普通 HTML 中的 onChange 事件不同， 在原生组件中，只有 input 元素失去焦点才会触发 onChange 事件， 在 React 中，只要元素的值被修改就会触发 onChange 事件。

var MyComponent = React.createClass({

getInitialState: function() {

return {

value: ''

}

},

render: function() {

return <div onChange={this.onChangeBubble}>

<input value={this.state.value} onChange={this.onChange}/>

</div>

},

onChange: function(ev) {

console.log('change: ' + ev.target.value);

this.setState({

value: ev.target.value

});

},

// onChange 事件支持所有组件，可以被用于监听冒泡事件

onChangeBubble: function(ev) {

console.log('bubble onChange event', + ev.target.value);

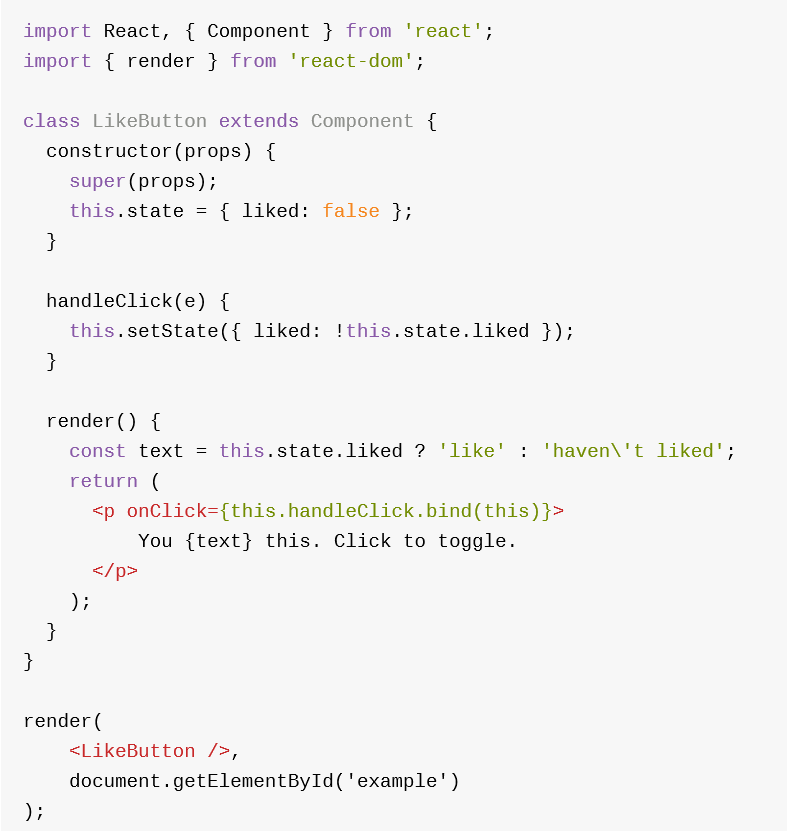
}

})

参数传递

给事件处理函数传递额外参数的方式：bind(this,arg1,arg2,...)

其他案例：



React里面绑定事件的方式和在HTML中绑定事件类似，使用驼峰式命名指定要绑定的onClick属性为组件定义的一个方法 {this.handleClick.bind(this)}

注意要显式调用bind(this)将事件函数上下文绑定要组件实例上。

另外一种绑定方式

render() {

return (

<input onChange={() => this.handleChange()}

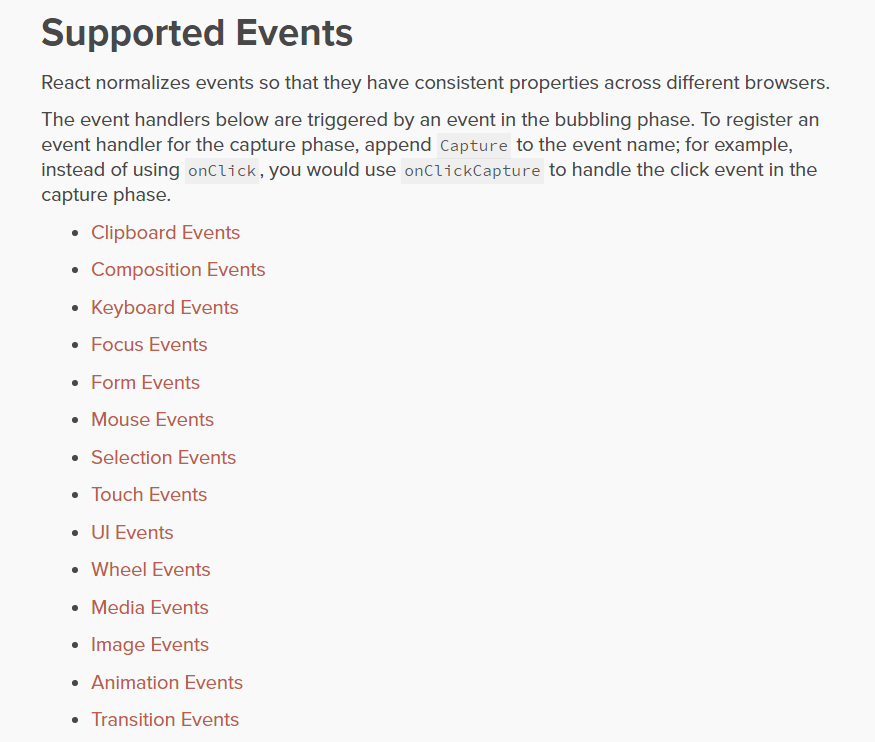
value={this.state.text} />

);

合成事件（SyntheticEvent）

在 React 中，事件的处理由其内部自己实现的事件系统完成，触发的事件都叫做 合成事件（SyntheticEvent），事件系统对浏览器做了兼容，其提供的 API 与原生的事件无异。

官方事件文档 http://facebook.github.io/react/docs/events.html



**3、****受控组件和非受控组件（pdf里看，不重复写了）**

**4、****style，样式设置**

基本的样式设置

render: function() {

var style = {

backgroundColor: 'red',

height: 100,

width: 100

}

return <div style={style}></div>

}

实际使用中，一般css也单独出来作为模块导入，在DOM中通过className使用样式。

### 2.2 redux,router基础，redux进阶

Redux和router基础查看目录下的react-tutorial.pdf

Redux进阶请看 redux进阶.pdf

### 2.3 redux中间件

打开 理解 redux 中间件.pdf

### 2.4 redux异步操作

打开 掌握redux异步操作.pdf

### 2.5 router深入

打开 react-router的实现原理 .pdf

### 参考资料：

<http://www.alloyteam.com/2015/09/react-redux/>

<https://segmentfault.com/a/1190000005136764#articleHeader2>

[http://cn.redux.js.org//docs/introduction/ThreePrinciples.html](http://cn.redux.js.org/docs/introduction/ThreePrinciples.html)

https://react-guide.github.io/react-router-cn/

2016/11/21 高明尧