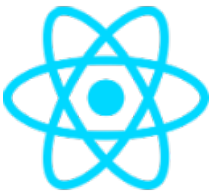
React 教程



React 是一个用于构建用户界面的 JAVASCRIPT 库。

React主要用于构建UI，很多人认为 React 是 MVC 中的 V（视图）。

React 起源于 Facebook 的内部项目，用来架设 Instagram 的网站，并于 2013 年 5 月开源。

React 拥有较高的性能，代码逻辑非常简单，越来越多的人已开始关注和使用它。

React 特点

1.声明式设计 −React采用声明范式，可以轻松描述应用。

2.高效 −React通过对DOM的模拟，最大限度地减少与DOM的交互。

3.灵活 −React可以与已知的库或框架很好地配合。

4.JSX − JSX 是 JavaScript 语法的扩展。React 开发不一定使用 JSX，但我们建议使用它。

5.组件 − 通过 React 构建组件，使得代码更加容易得到复用，能够很好的应用在大项目的开发中。

6.单向响应的数据流 − React 实现了单向响应的数据流，从而减少了重复代码，这也是它为什么比传统数据绑定更简单。

阅读本教程前，您需要了解的知识：

在开始学习 React 之前，您需要具备以下基础知识：

HTML5

CSS

JavaScript

React 第一个实例

在每个章节中，您可以在线编辑实例，然后点击按钮查看结果。

本教程使用了 React 的版本为 16.4.0，你可以在官网 https://reactjs.org/ 下载最新版。

React 实例

```
<div id="example"></div>
<script type="text/babel">
ReactDOM.render(
<h1>Hello, world!</h1>,
document.getElementById('example')
);
</script>
```

尝试一下 »

React 安装

React 可以直接下载使用，下载包中也提供了很多学习的实例。

本教程使用了 React 的版本为 16.4.0，你可以在官网 https://reactjs.org/ 下载最新版。

你也可以直接使用 BootCDN 的 React CDN 库，地址如下：

```
<script src="https://cdn.bootcss.com/react/16.4.0/umd/react.development.js"></script>
<script src="https://cdn.bootcss.com/react-dom/16.4.0/umd/react-dom.development.js"></script>
<!-- 生产环境中不建议使用 -->
<script src="https://cdn.bootcss.com/babel-standalone/6.26.0/babel.min.js"></script>
```

官方提供的 CDN 地址：

```
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
<!-- 生产环境中不建议使用 -->
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
```

注意：在浏览器中使用 Babel 来编译 JSX 效率是非常低的。

使用实例

以下实例输出了 Hello, world!

React 实例

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Hello React!</title>
<script src="https://cdn.bootcss.com/react/16.4.0/umd/react.development.js"></script>
<script src="https://cdn.bootcss.com/react-dom/16.4.0/umd/react-dom.development.js"></script>
<script src="https://cdn.bootcss.com/babel-standalone/6.26.0/babel.min.js"></script>
</head>
<body>
<div id="example"></div>
<script type="text/babel">
ReactDOM.render(
<h1>Hello, world!</h1>,
document.getElementById('example')
);
</script>
</body>
</html>
```

尝试一下 »

实例解析：

实例中我们引入了三个库： react.min.js 、react-dom.min.js 和 babel.min.js：

react.min.js - React 的核心库

react-dom.min.js - 提供与 DOM 相关的功能

babel.min.js - Babel 可以将 ES6 代码转为 ES5 代码，这样我们就能在目前不支持 ES6 浏览器上执行 React 代码。Babel 内嵌了对 JSX 的支持。通过将 Babel 和 babel-sublime 包（package）一同使用可以让源码的语法渲染上升到一个全新的水平。

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

以上代码将一个 h1 标题，插入 id="example" 节点中。

> ***注意：***
>
> *如果我们需要使用 JSX，则 <script> 标签的 type 属性需要设置为 text/babel。*

通过 npm 使用 React

如果你的系统还不支持 Node.js 及 NPM 可以参考我们的 Node.js 教程。

我们建议在 React 中使用 CommonJS 模块系统，比如 browserify 或 webpack，本教程使用 webpack。

国内使用 npm 速度很慢，你可以使用淘宝定制的 cnpm (gzip 压缩支持) 命令行工具代替默认的 npm:

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

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

这样就可以使用 cnpm 命令来安装模块了：

```
$ cnpm install [name]
```

更多信息可以查阅：http://npm.taobao.org/。

使用 create-react-app 快速构建 React 开发环境

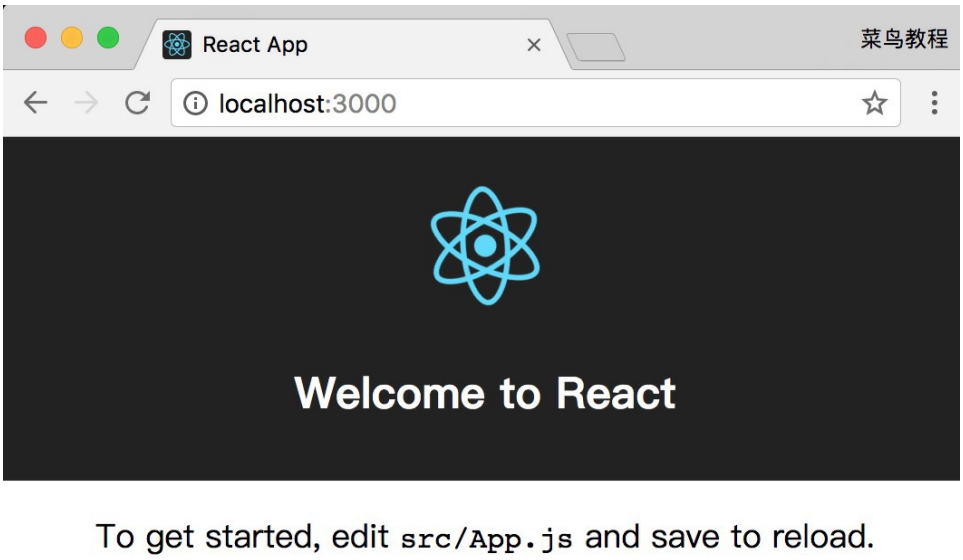
create-react-app 是来自于 Facebook，通过该命令我们无需配置就能快速构建 React 开发环境。

create-react-app 自动创建的项目是基于 Webpack + ES6 。

执行以下命令创建项目：

```
$ cnpm install -g create-react-app

$ create-react-app my-app

$ cd my-app/

$ npm start
```

在浏览器中打开 **http://localhost:3000/** ，结果如下图所示：



项目的目录结构如下：

```
my-app/
  README.md
  node_modules/
  package.json
  .gitignore
  public/
    favicon.ico
    index.html
    manifest.json
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
```

manifest.json 指定了开始页面 index.html，一切的开始都从这里开始，所以这个是代码执行的源头。

尝试修改 src/App.js 文件代码：

src/App.js

```
import React, { Component } from 'react';
import logo from './logo.svg';
```

```
import './App.css';
class App extends Component {
render() {
return (
<div className="App">
<div className="App-header">
<img src={logo} className="App-logo" alt="logo" />
<h2>欢迎来到菜鸟教程</h2>
</div>
<p className="App-intro">
你可以在 <code>src/App.js</code> 文件中修改。
</p>
</div>
);
}
}
export default App;
```

修改后，打开 http://localhost:3000/ （一般自动刷新），输出结果如下：

在 Cloud Studio 中运行 React

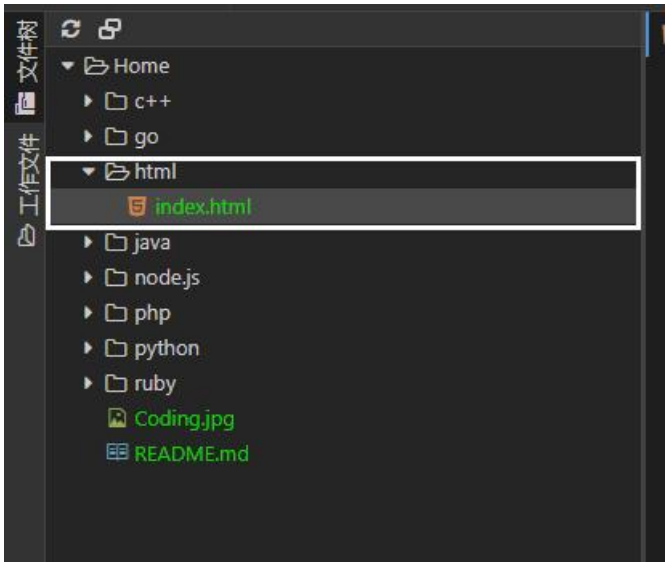
下面我们介绍如何在 Cloud Studio 中安装、使用 React：

step1：访问Cloud Studio，注册/登录账户。

step2：在右侧的运行环境菜单选择 "ubuntu"：

alt=" />

step3：在左侧代码目录中新建 html 目录，编写你的HTML代码，例如 index.html



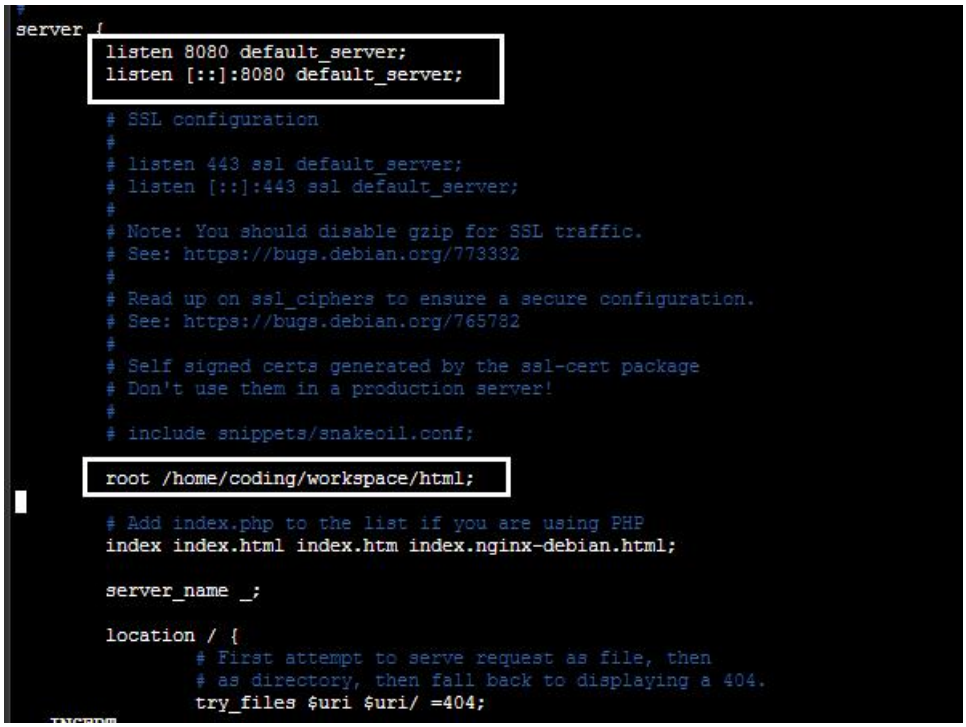
step4: 你可以在官网 https://reactjs.org/ 下载最新版。你也可以直接使用 BootCDN 的 React CDN 库，地址如下：

```
<script src="https://cdn.bootcss.com/react/16.4.0/umd/react.development.js"></script>

<script src="https://cdn.bootcss.com/react-dom/16.4.0/umd/react-dom.development.js"></script>

<!-- 生产环境中不建议使用 -->

<script src="https://cdn.bootcss.com/babel-standalone/6.26.0/babel.min.js"></script>
```

官方提供的 CDN 地址：

```
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>

<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>

<!-- 生产环境中不建议使用 -->

<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
```

step5：在终端中输入命令 `sudo vim /etc/nginx/site-enable/default`。将配置文件红框部分修改为如下图所示，然后输入命令：`sudo nginx restart` 重启 nginx 服务（ nginx 安装完成并启动后默认会监听80端口。我们需要将 nginx 的站点目录以及监听的端口号改为我们需要的）



step6：点击最右侧的【访问链接】选项卡，在访问链接面板中填写端口号为：8080（和刚才 nginx 配置文件中的端口号一致），点击创建链接，即可点击生成的链接访问我们刚刚编写的代码，查看 React 效果。

□ React 教程 React JSX □

2 篇笔记
#2

□写笔记

create-react-app 执行慢的解决方法：
在使用 **create-react-app my-app** 来创建一个新的React应用，在拉取各种资源时,往往会非常慢,一直卡在那：

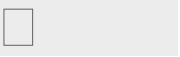
```
fetchMetadata: sill mapToRegistry uri http://registry.npmjs.org/whatwg-fetch
```

可以看到资源还是使用了 npmjs.org，解决方法是换成淘宝的资源：

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

-- 配置后可通过下面方式来验证是否成功

$ npm config get registry

-- 或 npm info express
```

tianqixin2年前 (2017-02-14)

#1



React 代码的书写格式和以前的 JS 有很大的不同，下面通过对这段代码进行分析了解一下他。
以前使用JS 定义一个变量使用 var 现在用 const：

```
const div = document.createElement('div');
```

ReactDOM.render(...) 是渲染方法，所有的 js,html 都可通过它进行渲染绘制，他又两个参数，内容和渲染目标 js 对象。
内容就是要在渲染目标中显示的东西，可以是一个React 部件，也可以是一段HTML或TEXT文本。渲染目标JS对象，就是一个DIV或TABEL,或TD 等HTML的节点对象。

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

`unmountComponentAtNode()` 这个方法是解除渲染挂载，作用和 render 刚好相反，也就清空一个渲染目标中的 React 部件或 html 内容。

```
ReactDOM.unmountComponentAtNode(div);
```

xjjuser2个月前 (08-09)

反馈/建议

React Refs

React 事件处理

React 元素渲染

元素是构成 React 应用的最小单位，它用于描述屏幕上输出的内容。

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

与浏览器的 DOM 元素不同，React 当中的元素事实上是普通的对象，React DOM 可以确保 浏览器 DOM 的数据内容与 React 元素保持一致。

将元素渲染到 DOM 中

首先我们在一个 HTML 页面中添加一个 `id="example"` 的 `<div>`:

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

在此 div 中的所有内容都将由 React DOM 来管理，所以我们将其称之为 "根" DOM 节点。

我们用 React 开发应用时一般只会定义一个根节点。但如果你是在一个已有的项目当中引入 React 的话，你可能会需要在不同的部分单独定义 React 根节点。

要将React元素渲染到根DOM节点中，我们通过把它们都传递给 `ReactDOM.render()` 的方法来将其渲染到页面上：

实例

```
const element = <h1>Hello, world!</h1>;
ReactDOM.render(
element,
document.getElementById('example')
);
```

尝试一下 »

更新元素渲染

React 元素都是不可变的。当元素被创建之后，你是无法改变其内容或属性的。

目前更新界面的唯一办法是创建一个新的元素，然后将它传入 ReactDOM.render() 方法：

来看一下这个计时器的例子:

实例

```
function tick() {
const element = (
<div>
<h1>Hello, world!</h1>
<h2>现在是 {new Date().toLocaleTimeString()}.</h2>
</div>
);
ReactDOM.render(
element,
document.getElementById('example')
);
}
setInterval(tick, 1000);
```

尝试一下 »

以上实例通过 setInterval() 方法，每秒钟调用一次 ReactDOM.render()。

我们可以将要展示的部分封装起来，以下实例用一个函数来表示：

实例

```
function Clock(props) {
return (
<div>
<h1>Hello, world!</h1>
<h2>现在是 {props.date.toLocaleTimeString()}.</h2>
</div>
);
}
function tick() {
ReactDOM.render(
<Clock date={new Date()} />,
document.getElementById('example')
);
}
setInterval(tick, 1000);
```

尝试一下 »

除了函数外我们还可以创建一个 React.Component 的 ES6 类，该类封装了要展示的元素，需要注意的是在 render() 方法中，需要使用 `this.props` 替换 `props`：

实例

```
class Clock extends React.Component {
render() {
return (
<div>
<h1>Hello, world!</h1>
<h2>现在是 {this.props.date.toLocaleTimeString()}.</h2>
</div>
);
}
}
function tick() {
ReactDOM.render(
<Clock date={new Date()} />,
document.getElementById('example')
);
}
setInterval(tick, 1000);
```

尝试一下 »

***React** 只会更新必要的部分*

值得注意的是 React DOM 首先会比较元素内容先后的不同，而在渲染过程中只会更新改变了的部分。

React Refs

React 事件处理

点我分享笔记

反馈/建议

React JSX

React 使用 JSX 来替代常规的 JavaScript。

JSX 是一个看起来很像 XML 的 JavaScript 语法扩展。

我们不需要一定使用 JSX，但它有以下优点：

JSX 执行更快，因为它在编译为 JavaScript 代码后进行了优化。

它是类型安全的，在编译过程中就能发现错误。

使用 JSX 编写模板更加简单快速。

使用 JSX

JSX 看起来类似 HTML，我们可以看下实例:

```
ReactDOM.render(
<h1>Hello, world!</h1>,
document.getElementById('example')
```

```
);
```

我们可以在以上代码中嵌套多个 HTML 标签，需要使用一个 div 元素包裹它，实例中的 p 元素添加了自定义属性 **data-myattribute**，添加自定义属性需要使用 **data-** 前缀。

React 实例

```
ReactDOM.render(
<div>
<h1>菜鸟教程</h1>
<h2>欢迎学习 React</h2>
<p data-myattribute = "somevalue">这是一个很不错的 JavaScript 库!</p>
</div>
,
document.getElementById('example')
);
```

尝试一下 »

独立文件

你的 React JSX 代码可以放在一个独立文件上，例如我们创建一个 helloworld_react.js 文件，代码如下：

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

然后在 HTML 文件中引入该 JS 文件：

React 实例

```
<body>
<div id="example"></div>
<script type="text/babel" src="helloworld_react.js"></script>
</body>
```

尝试一下 »

JavaScript 表达式

我们可以在 JSX 中使用 JavaScript 表达式。表达式写在花括号 **{}** 中。实例如下：

React 实例

```
ReactDOM.render(
<div>
<h1>{1+1}</h1>
</div>
,
document.getElementById('example')
);
```

尝试一下 »

在 JSX 中不能使用 **if else** 语句，但可以使用 **conditional (三元运算)** 表达式来替代。以下实例中如果变量 **i** 等于 **1** 浏览器将输出 **true**, 如果修改 i 的值，则会输出 **false**.

React 实例

```
ReactDOM.render(
<div>
<h1>{i == 1 ? 'True!' : 'False'}</h1>
</div>
,
document.getElementById('example')
);
```

尝试一下 »

样式

React 推荐使用内联样式。我们可以使用 **camelCase** 语法来设置内联样式. React 会在指定元素数字后自动添加 **px** 。以下实例演示了为 **h1** 元素添加 **myStyle** 内联样式：

React 实例

```
var myStyle = {
fontSize: 100,
color: '#FF0000'
};
ReactDOM.render(
<h1 style = {myStyle}>菜鸟教程</h1>,
document.getElementById('example')
);
```

尝试一下 »

注释

注释需要写在花括号中，实例如下：

React 实例

```
ReactDOM.render(
<div>
<h1>菜鸟教程</h1>
{/*注释...*/}
</div>,
document.getElementById('example')
);
```

尝试一下 »

数组

JSX 允许在模板中插入数组，数组会自动展开所有成员：

React 实例

```
var arr = [
<h1>菜鸟教程</h1>,
<h2>学的不仅是技术，更是梦想！</h2>,
];
ReactDOM.render(
<div>{arr}</div>,
document.getElementById('example')
);
```

尝试一下 »

HTML 标签 vs. React 组件

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

要渲染 HTML 标签，只需在 JSX 里使用小写字母的标签名。

```
var myDivElement = <div className="foo" />;
ReactDOM.render(myDivElement, document.getElementById('example'));
```

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} />;
ReactDOM.render(myElement, document.getElementById('example'));
```

React 的 JSX 使用大、小写的约定来区分本地组件的类和 HTML 标签。

> *注意:*
>
> *由于 JSX 就是 JavaScript，一些标识符像 `class` 和 `for` 不建议作为 XML 属性名。作为替代，React DOM 使用 `className` 和 `htmlFor` 来做对应的属性。*

React 组件

本章节我们将讨论如何使用组件使得我们的应用更容易来管理。

接下来我们封装一个输出 "Hello World！" 的组件，组件名为 HelloMessage：

React 实例

```
function HelloMessage(props) {
return <h1>Hello World!</h1>;
}
const element = <HelloMessage />;
ReactDOM.render(
element,
document.getElementById('example')
);
```

尝试一下 »

实例解析：

1、我们可以使用函数定义了一个组件：

```
function HelloMessage(props) {

    return <h1>Hello World!</h1>;

}
```

你也可以使用 ES6 class 来定义一个组件:

```
class Welcome extends React.Component {

  render() {
```

```
    return <h1>Hello World!</h1>;

  }

}
```

2、**const element = <HelloMessage />** 为用户自定义的组件。

> *注意，原生 HTML 元素名以小写字母开头，而自定义的 React 类名以大写字母开头，比如 HelloMessage 不能写成 helloMessage 。除此之外还需要注意组件类只能包含一个顶层标签，否则也会报错。*

如果我们需要向组件传递参数，可以使用 **this.props** 对象,实例如下：

React 实例

```
function HelloMessage(props) {
return <h1>Hello {props.name}!</h1>;
}
const element = <HelloMessage name="Runoob"/>;
ReactDOM.render(
element,
document.getElementById('example')
);
```

尝试一下 »

以上实例中 **name** 属性通过 `props.name` 来获取。

> *注意，在添加属性时， class 属性需要写成 className ， for 属性需要写成 htmlFor ， 这是因为 class 和 for 是 JavaScript 的保留字。*

复合组件

我们可以通过创建多个组件来合成一个组件，即把组件的不同功能点进行分离。

以下实例我们实现了输出网站名字和网址的组件：

React 实例

```
function Name(props) {
return <h1>网站名称：{props.name}</h1>;
}
function Url(props) {
return <h1>网站地址：{props.url}</h1>;
}
function Nickname(props) {
return <h1>网站小名：{props.nickname}</h1>;
}
function App() {
return (
<div>
<Name name="菜鸟教程" />
<Url url="http://www.runoob.com" />
<Nickname nickname="Runoob" />
</div>
);
}
ReactDOM.render(
<App />,
document.getElementById('example')
);
```

尝试一下 »

实例中 App 组件使用了 Name、Url 和 Nickname 组件来输出对应的信息。

4 篇笔记

写笔记

反馈/建议

React State(状态)

React 把组件看成是一个状态机（State Machines）。通过与用户的交互，实现不同状态，然后渲染 UI，让用户界面和数据保持一致。

React 里，只需更新组件的 state，然后根据新的 state 重新渲染用户界面（不要操作 DOM）。

以下实例创建一个名称扩展为 React.Component 的 ES6 类，在 render() 方法中使用 this.state 来修改当前的时间。

添加一个类构造函数来初始化状态 this.state，类组件应始终使用 props 调用基础构造函数。

React 实例

```
class Clock extends React.Component {
constructor(props) {
super(props);
this.state = {date: new Date()};
}
render() {
return (
<div>
<h1>Hello, world!</h1>
<h2>现在是 {this.state.date.toLocaleTimeString()}.</h2>
</div>
);
}
}
ReactDOM.render(
<Clock />,
document.getElementById('example')
);
```

尝试一下 »

接下来，我们将使Clock设置自己的计时器并每秒更新一次。

将生命周期方法添加到类中

在具有许多组件的应用程序中，在销毁时释放组件所占用的资源非常重要。

每当 Clock 组件第一次加载到 DOM 中的时候，我们都想生成定时器，这在 React 中被称为**挂载**。

同样，每当 Clock 生成的这个 DOM 被移除的时候，我们也会想要清除定时器，这在 React 中被称为**卸载**。

我们可以在组件类上声明特殊的方法，当组件挂载或卸载时，来运行一些代码：

React 实例

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

```
this.state = {date: new Date()};
}
componentDidMount() {
this.timerID = setInterval(
() => this.tick(),
1000
);
}
componentWillUnmount() {
clearInterval(this.timerID);
}
tick() {
this.setState({
date: new Date()
});
}
render() {
return (
<div>
<h1>Hello, world!</h1>
<h2>现在是 {this.state.date.toLocaleTimeString()}.</h2>
</div>
);
}
}
ReactDOM.render(
<Clock />,
document.getElementById('example')
);
```

尝试一下 »

实例解析：

`componentDidMount()` 与 `componentWillUnmount()` 方法被称作生命周期钩子。

在组件输出到 DOM 后会执行 `componentDidMount()` 钩子，我们就可以在这个钩子上设置一个定时器。

this.timerID 为计算器的 ID，我们可以在 `componentWillUnmount()` 钩子中卸载计算器。

代码执行顺序：

1. 当 `<Clock />` 被传递给 `ReactDOM.render()` 时，React 调用 `Clock` 组件的构造函数。 由于 `Clock` 需要显示当前时间，所以使用包含当前时间的对象来初始化 `this.state` 。 我们稍后会更新此状态。
2. React 然后调用 `Clock` 组件的 `render()` 方法。这是 React 了解屏幕上应该显示什么内容，然后 React 更新 DOM 以匹配 `Clock` 的渲染输出。
3. 当 `Clock` 的输出插入到 DOM 中时，React 调用 `componentDidMount()` 生命周期钩子。 在其中，`Clock` 组件要求浏览器设置一个定时器，每秒钟调用一次 `tick()`。
4. 浏览器每秒钟调用 `tick()` 方法。 在其中，`Clock` 组件通过使用包含当前时间的对象调用 `setState()` 来调度UI更新。 通过调用 `setState()` ，React 知道状态已经改变，并再次调用 `render()` 方法来确定屏幕上应当显示什么。 这一次，`render()` 方法中的 `this.state.date` 将不同，所以渲染输出将包含更新的时间，并相应地更新 DOM。
5. 一旦 `Clock` 组件被从 DOM 中移除，React 会调用 `componentWillUnmount()` 这个钩子函数，定时器也就会被清除。

数据自顶向下流动

父组件或子组件都不能知道某个组件是有状态还是无状态，并且它们不应该关心某组件是被定义为一个函数还是一个类。

这就是为什么状态通常被称为局部或封装。 除了拥有并设置它的组件外，其它组件不可访问。

以下实例中 FormattedDate 组件将在其属性中接收到 date 值，并且不知道它是来自 Clock 状态、还是来自 Clock 的属性、亦或手工输入：

React 实例

```
function FormattedDate(props) {
return <h2>现在是 {props.date.toLocaleTimeString()}.</h2>;
}
class Clock extends React.Component {
constructor(props) {
```

```
super(props);
this.state = {date: new Date()};
}
componentDidMount() {
this.timerID = setInterval(
() => this.tick(),
1000
);
}
componentWillUnmount() {
clearInterval(this.timerID);
}
tick() {
this.setState({
date: new Date()
});
}
render() {
return (
<div>
<h1>Hello, world!</h1>
<FormattedDate date={this.state.date} />
</div>
);
}
}
ReactDOM.render(
<Clock />,
document.getElementById('example')
);
```

尝试一下 »

这通常被称为自顶向下或单向数据流。 任何状态始终由某些特定组件所有，并且从该状态导出的任何数据或 UI 只能影响树中下方的组件。

如果你想象一个组件树作为属性的瀑布，每个组件的状态就像一个额外的水源，它连接在一个任意点，但也流下来。

为了表明所有组件都是真正隔离的，我们可以创建一个 App 组件，它渲染三个Clock：

React 实例

```
function FormattedDate(props) {
return <h2>现在是 {props.date.toLocaleTimeString()}.</h2>;
}
class Clock extends React.Component {
constructor(props) {
super(props);
this.state = {date: new Date()};
}
componentDidMount() {
this.timerID = setInterval(
() => this.tick(),
1000
);
}
componentWillUnmount() {
clearInterval(this.timerID);
}
tick() {
this.setState({
date: new Date()
});
}
render() {
return (
<div>
<h1>Hello, world!</h1>
<FormattedDate date={this.state.date} />
</div>
);
}
}
function App() {
```

```
return (
<div>
<Clock />
<Clock />
<Clock />
</div>
);
}
ReactDOM.render(<App />, document.getElementById('example'));
```

尝试一下 »

以上实例中每个 Clock 组件都建立了自己的定时器并且独立更新。

在 React 应用程序中，组件是有状态还是无状态被认为是可能随时间而变化的组件的实现细节。

我们可以在有状态组件中使用无状态组件，也可以在无状态组件中使用有状态组件。

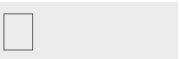
□ React 组件 React Props □

□

1 篇笔记

#1

□写笔记



关于挂载时的 setInterval 中调用 tick() 的方式 ()=>this.tick()：

1、()=>this.tick()

`()=>this.tick()` 是 ES6 中声明函数的一种方式，叫做箭头函数表达式，引入箭头函数有两个方面的作用：更简短的函数并且不绑定 this。

```
var f = ([参数]) => 表达式（单一）

// 等价于以下写法

var f = function([参数]){

   return 表达式;

}
```

箭头函数的基本语法如下：

```
(参数1, 参数2, …, 参数N) => { 函数声明 }

(参数1, 参数2, …, 参数N) => 表达式（单一）

//相当于：(参数1, 参数2, …, 参数N) =>{ return 表达式; }



// 当只有一个参数时，圆括号是可选的：

(单一参数) => {函数声明}

单一参数 => {函数声明}



// 没有参数的函数应该写成一对圆括号。

() => {函数声明}
```

根据以上概念，尝试将 setInterval 中调用 tick() 的方式改为通常声明方式：

```
this.timerID = setInterval(function(){

    return this.tick();
```

```
  },1000

);
```

但是会报错，tick() 不是一个方法。

2、this.tick()

this.tick() 中的 this 指代的是 function，而不是我们想要的指代所在的组件类 Clock，所以我们要想办法让 this 能被正常指代。这里我们采用围魏救赵的办法:

```
let that = this;

this.timerID = setInterval(function () {

  return that.tick();

},1000);
```

在闭包函数的外部先用 that 引用组件 Clock 中挂载组件方法 componentDidMount() 中 this 的值，然后在 setInterval 中闭包函数中使用that，that 无法找到声明，就会根据作用域链去上级（上次层）中继承 that，也就是我们引用的组件类 Clock 中的 this。

到此为止，将 () => this.tick()等价代换为了我们熟悉的形式。

ch4o53周前 (09-07)

反馈/建议

React Props

state 和 props 主要的区别在于 **props** 是不可变的，而 state 可以根据与用户交互来改变。这就是为什么有些容器组件需要定义 state 来更新和修改数据。 而子组件只能通过 props 来传递数据。

使用 Props

以下实例演示了如何在组件中使用 props：

React 实例

```
function HelloMessage(props) {
return <h1>Hello {props.name}!</h1>;
}
const element = <HelloMessage name="Runoob"/>;
ReactDOM.render(
element,
document.getElementById('example')
);
```

尝试一下 »

实例中 name 属性通过 this.props.name 来获取。

默认 Props

你可以通过组件类的 defaultProps 属性为 props 设置默认值，实例如下：

React 实例

```
class HelloMessage extends React.Component {
render() {
return (
<h1>Hello, {this.props.name}</h1>
);
}
}
HelloMessage.defaultProps = {
name: 'Runoob'
};
const element = <HelloMessage/>;
ReactDOM.render(
element,
document.getElementById('example')
);
```

尝试一下 »

State 和 Props

以下实例演示了如何在应用中组合使用 state 和 props 。我们可以在父组件中设置 state， 并通过在子组件上使用 props 将其传递到子组件上。在 render 函数中, 我们设置 name 和 site 来获取父组件传递过来的数据。

React 实例

```
class WebSite extends React.Component {
constructor() {
super();
this.state = {
name: "菜鸟教程",
site: "https://www.runoob.com"
}
}
render() {
return (
<div>
<Name name={this.state.name} />
<Link site={this.state.site} />
</div>
);
}
}
class Name extends React.Component {
render() {
return (
<h1>{this.props.name}</h1>
);
}
}
class Link extends React.Component {
render() {
return (
<a href={this.props.site}>
{this.props.site}
</a>
);
}
}
ReactDOM.render(
<WebSite />,
document.getElementById('example')
);
```

尝试一下 »

Props 验证

React.PropTypes 在 React v15.5 版本后已经移到了 prop-types 库。

```
<script src="https://cdn.bootcss.com/prop-types/15.6.1/prop-types.js"></script>
```

Props 验证使用 **propTypes**，它可以保证我们的应用组件被正确使用，React.PropTypes 提供很多验证器 (validator) 来验证传入数据是否有效。当向 props 传入无效数据时，JavaScript 控制台会抛出警告。

以下实例创建一个 Mytitle 组件，属性 title 是必须的且是字符串，非字符串类型会自动转换为字符串 ：

React 16.4 实例

```
var title = "菜鸟教程";
// var title = 123;
class MyTitle extends React.Component {
render() {
return (
<h1>Hello, {this.props.title}</h1>
);
}
}
MyTitle.propTypes = {
title: PropTypes.string
};
ReactDOM.render(
<MyTitle title={title} />,
document.getElementById('example')
);
```

尝试一下 »

React 15.4 实例

```
var title = "菜鸟教程";
// var title = 123;
var MyTitle = React.createClass({
propTypes: {
title: React.PropTypes.string.isRequired,
},
render: function() {
return <h1> {this.props.title} </h1>;
}
});
ReactDOM.render(
<MyTitle title={title} />,
document.getElementById('example')
);
```

尝试一下 »

更多验证器说明如下：

```
MyComponent.propTypes = {
// 可以声明 prop 为指定的 JS 基本数据类型，默认情况，这些数据是可选的
optionalArray: React.PropTypes.array,
optionalBool: React.PropTypes.bool,
optionalFunc: React.PropTypes.func,
optionalNumber: React.PropTypes.number,
optionalObject: React.PropTypes.object,
optionalString: React.PropTypes.string,
// 可以被渲染的对象 numbers, strings, elements 或 array
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']),
// 可以是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 shape 参数的对象
optionalObjectWithShape: React.PropTypes.shape({
color: React.PropTypes.string,
fontSize: React.PropTypes.number
}),
// 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!');
}
}
}
}
```

□ React State(状态)　　React 组件 API □

2 篇笔记

#2

□写笔记

上次在 React 组件看到这篇笔记没看懂，原来是这里的，现在贴过来分享一下。但是自己现在还是不太懂，希望过几天再来看的时候能够明白。
对创建多个组件的代码，做了点小修改，帮助大家理解。
`<WebSite name="菜鸟教程" site=" http://www.runoob.com" />`，这种形式传入的 name 和 url 值，只能在 WebSit 组件中用 this.props.xxx 来使用。虽然原来的代码中，Name 和 Site 组件中也是以同样的形式使用的，但并不是因为这条语句的作用，而是因为 `<Name name={this.props.name} /> <Link site={this.props.site} />` 。所以我特意将这几行代码做了修改，方便大家感受感受！
WebSite 组件中：

```
<Name title={this.props.name}/>

// 将this.props.name以title名称传给Name组件，Name通过this.props.title来使用其值

<Url site={this.props.url}/>

// 将this.props.url以site名称传给Url组件，Url通过this.props.site来使用其值
```

Name 组件中：

```
<h1>{this.props.title}</h1>
```

Site 组件中：`<a href={this.props.site}>{this.props.site}</a>`

年轻的C同学2个月前
(08-08)

#1

来补充一下上面那位同学所说的。很多情况下，子控件需要父控件所有的 props 参数，这个时候我们一个一个参数的写会很麻烦，比如：

```
<Name name={this.props.name} url={this.props.url}  .../>
```

那么我们怎么样吧父属性直接赋值给子组件的props参数呢？如下写法即可：

```
<Name props={this.props}/>
```

这样写就非常简洁了，也就子控件和父控件都有了同样数据结构的 props 参数。
很多情况下我们调试页面时，看到的参数名在父控件和子控件中部一样，但是表示的值是同一个，写这段代码的人可能还记得这个参数是转译的，但是其他人阅读时就会摸不着头脑，在效率上是处于弱势的，所以我们一般建议引用父组件参数尽量保持名称不变，以便以后维护。

xjjuser2个月前 (08-10)

反馈/建议

React 事件处理

React 元素的事件处理和 DOM 元素类似。但是有一点语法上的不同:

React 事件绑定属性的命名采用驼峰式写法，而不是小写。

如果采用 JSX 的语法你需要传入一个函数作为事件处理函数，而不是一个字符串(DOM 元素的写法)

HTML 通常写法是：

```
<button onclick="activateLasers()">
  激活按钮
</button>
```

React 中写法为：

```
<button onClick={activateLasers}>
  激活按钮
</button>
```

在 React 中另一个不同是你不能使用返回 **false** 的方式阻止默认行为， 你必须明确的使用 `preventDefault`。
例如，通常我们在 HTML 中阻止链接默认打开一个新页面，可以这样写：

```
<a href="#" onclick="console.log('点击链接'); return false">
  点我
</a>
```

在 React 的写法为：

```
function ActionLink() {
function handleClick(e) {
e.preventDefault();
console.log('链接被点击');
}
return (
<a href="#" onClick={handleClick}>
```

```
点我
</a>
);
}
```

实例中 e 是一个合成事件。

使用 React 的时候通常你不需要使用 addEventListener 为一个已创建的 DOM 元素添加监听器。你仅仅需要在这个元素初始渲染的时候提供一个监听器。

当你使用 ES6 class 语法来定义一个组件的时候，事件处理器会成为类的一个方法。例如，下面的 Toggle 组件渲染一个让用户切换开关状态的按钮：

实例

```
class Toggle extends React.Component {
constructor(props) {
super(props);
this.state = {isToggleOn: true};
// 这边绑定是必要的，这样 `this` 才能在回调函数中使用
this.handleClick = this.handleClick.bind(this);
}
handleClick() {
this.setState(prevState => ({
isToggleOn: !prevState.isToggleOn
}));
}
render() {
return (
<button onClick={this.handleClick}>
{this.state.isToggleOn ? 'ON' : 'OFF'}
</button>
);
}
}
ReactDOM.render(
<Toggle />,
document.getElementById('example')
);
```

尝试一下 »

你必须谨慎对待 JSX 回调函数中的 this，类的方法默认是不会绑定 this 的。如果你忘记绑定 this.handleClick 并把它传入 onClick, 当你调用这个函数的时候 this 的值会是 undefined。

这并不是 React 的特殊行为；它是函数如何在 JavaScript 中运行的一部分。通常情况下，如果你没有在方法后面添加 ()，例如 `onClick={this.handleClick}`，你应该为这个方法绑定 this。

如果使用 bind 让你很烦，这里有两种方式可以解决。如果你正在使用实验性的属性初始化器语法，你可以使用属性初始化器来正确的绑定回调函数：

```
class LoggingButton extends React.Component {
// 这个语法确保了 `this` 绑定在 handleClick 中
// 这里只是一个测试
handleClick = () => {
console.log('this is:', this);
}
render() {
return (
<button onClick={this.handleClick}>
Click me
</button>
);
}
}
```

如果你没有使用属性初始化器语法，你可以在回调函数中使用 箭头函数：

```
class LoggingButton extends React.Component {
handleClick() {
console.log('this is:', this);
}
render() {
```

```
// 这个语法确保了 `this` 绑定在 handleClick 中
return (
<button onClick={(e) => this.handleClick(e)}>
Click me
</button>
);
}
}
```

使用这个语法有个问题就是每次 LoggingButton 渲染的时候都会创建一个不同的回调函数。在大多数情况下，这没有问题。然而如果这个回调函数作为一个属性值传入低阶组件，这些组件可能会进行额外的重新渲染。我们通常建议在构造函数中绑定或使用属性初始化器语法来避免这类性能问题。

向事件处理程序传递参数

通常我们会为事件处理程序传递额外的参数。例如，若是 id 是你要删除那一行的 id，以下两种方式都可以向事件处理程序传递参数：

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

上述两种方式是等价的。

上面两个例子中，参数 e 作为 React 事件对象将会被作为第二个参数进行传递。通过箭头函数的方式，事件对象必须显式的进行传递，但是通过 bind 的方式，事件对象以及更多的参数将会被隐式的进行传递。

值得注意的是，通过 bind 方式向监听函数传参，在类组件中定义的监听函数，事件对象 e 要排在所传递参数的后面，例如:

```
class Popper extends React.Component{
constructor(){
super();
this.state = {name:'Hello world!'};
}
preventPop(name, e){ //事件对象e要放在最后
e.preventDefault();
alert(name);
}
render(){
return (
<div>
<p>hello</p>
{/* 通过 bind() 方法传递参数。 */}
<a href="https://reactjs.org" onClick={this.preventPop.bind(this,this.state.name)}>Click</a>
</div>
);
}
}
```

□ React 元素渲染　　React 条件渲染 □

事件：`this.handleclick.bind(this，要传的参数)`
函数：`handleclick(传过来的参数，event)`
从 **render** 中传递参数到 外部函数 `one()`：

```
class Ex extends React.Component{



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



... ... ...



function one(para){

    console.log('parameter in one Func',para);

}



... ... ...



render(){

    var mm='AABB';

    return (<div><button onClick={this.one.bind(this,mm)}> test</button>

    <button onClick={this.one.bind(this,this.state.name)}> test</button></div>);



}}
```

阿珂2个月前 (07-26)

反馈/建议

React 条件渲染

在 React 中，你可以创建不同的组件来封装各种你需要的行为。然后还可以根据应用的状态变化只渲染其中的一部分。

React 中的条件渲染和 JavaScript 中的一致，使用 JavaScript 操作符 if 或条件运算符来创建表示当前状态的元素，然后让 React 根据它们来更新 UI 。

先来看两个组件:

```
function UserGreeting(props) {

  return <h1>欢迎回来!</h1>;

}



function GuestGreeting(props) {

  return <h1>请先注册。</h1>;

}
```

我们将创建一个 Greeting 组件，它会根据用户是否登录来显示其中之一：

React 实例

```
function Greeting(props) {
const isLoggedIn = props.isLoggedIn;
if (isLoggedIn) {
return <UserGreeting />;
}
return <GuestGreeting />;
}
ReactDOM.render(
// 尝试修改 isLoggedIn={true}:
<Greeting isLoggedIn={false} />,
document.getElementById('example')
);
```

尝试一下 »

元素变量

你可以使用变量来储存元素。它可以帮助你有条件的渲染组件的一部分，而输出的其他部分不会更改。

在下面的例子中，我们将要创建一个名为 LoginControl 的有状态的组件。

它会根据当前的状态来渲染 `<LoginButton />` 或 `<LogoutButton />`，它也将渲染前面例子中的 `<Greeting />`。

React 实例

```
class LoginControl extends React.Component {
constructor(props) {
super(props);
this.handleLoginClick = this.handleLoginClick.bind(this);
this.handleLogoutClick = this.handleLogoutClick.bind(this);
this.state = {isLoggedIn: false};
}
handleLoginClick() {
this.setState({isLoggedIn: true});
}
handleLogoutClick() {
this.setState({isLoggedIn: false});
}
render() {
const isLoggedIn = this.state.isLoggedIn;
let button = null;
if (isLoggedIn) {
button = <LogoutButton onClick={this.handleLogoutClick} />;
} else {
button = <LoginButton onClick={this.handleLoginClick} />;
}
return (
<div>
<Greeting isLoggedIn={isLoggedIn} />
```

```
{button}
</div>
);
}
}
ReactDOM.render(
<LoginControl />,
document.getElementById('example')
);
```

尝试一下 »

与运算符 &&

你可以通过用花括号包裹代码在 JSX 中嵌入任何表达式 ，也包括 JavaScript 的逻辑与 &&，它可以方便地条件渲染一个元素。

React 实例

```
function Mailbox(props) {
const unreadMessages = props.unreadMessages;
return (
<div>
<h1>Hello!</h1>
{unreadMessages.length > 0 &&
<h2>
您有 {unreadMessages.length} 条未读信息。
</h2>
}
</div>
);
}
const messages = ['React', 'Re: React', 'Re:Re: React'];
ReactDOM.render(
<Mailbox unreadMessages={messages} />,
document.getElementById('example')
);
```

尝试一下 »

在 JavaScript 中，`true && expression` 总是返回 **expression**，而 `false && expression` 总是返回 **false**。

因此，如果条件是 **true**，`&&` 右侧的元素就会被渲染，如果是 **false**，React 会忽略并跳过它。

三目运算符

条件渲染的另一种方法是使用 JavaScript 的条件运算符:

```
condition ? true : false。
```

在下面的例子中，我们用它来有条件的渲染一小段文本。

render() { const isLoggedIn = this.state.isLoggedIn; return (The user is **{isLoggedIn ? 'currently' : 'not'}** logged in.); } 同样它也可以用在较大的表达式中，虽然不太直观：

```
render() {

  const isLoggedIn = this.state.isLoggedIn;

  return (

    <div>

      {isLoggedIn ? (

        <LogoutButton onClick={this.handleLogoutClick} />

      ) : (
```

```
        <LoginButton onClick={this.handleLoginClick} />

      )}

    </div>

  );

}
```

阻止组件渲染

在极少数情况下，你可能希望隐藏组件，即使它被其他组件渲染。让 render 方法返回 null 而不是它的渲染结果即可实现。

在下面的例子中，`<WarningBanner />` 根据属性 warn 的值条件渲染。如果 warn 的值是 false，则组件不会渲染：

React 实例

```
function WarningBanner(props) {
if (!props.warn) {
return null;
}
return (
<div className="warning">
警告!
</div>
);
}
class Page extends React.Component {
constructor(props) {
super(props);
this.state = {showWarning: true}
this.handleToggleClick = this.handleToggleClick.bind(this);
}
handleToggleClick() {
this.setState(prevState => ({
showWarning: !prevState.showWarning
}));
}
render() {
return (
<div>
<WarningBanner warn={this.state.showWarning} />
<button onClick={this.handleToggleClick}>
{this.state.showWarning ? '隐藏' : '显示'}
</button>
</div>
);
}
}
ReactDOM.render(
<Page />,
document.getElementById('example')
);
```

尝试一下 »

组件的 render 方法返回 null 并不会影响该组件生命周期方法的回调。例如，componentWillUpdate 和 componentDidUpdate 依然可以被调用。

□ React 事件处理　　React 列表 & Keys □

1 篇笔记

#1

□写笔记

通过条件渲染页面：

首先建一个函数，来根据不同的情况返回不同的值，然后建一个类组建，先进行变量的初始化，对变量进行操作，在组件内进行小的渲染，最后通过 **ReactDOM.render()**

渲染到页面上。

为什么要进行变量的初始化？

一个软件所分配到的空间中极可能存在着以前其他软件使用过后的残留数据，这些数据被称之为垃圾数据。所以通常情况下我们为一个变量，为一个数组，分配好存储空间之后都要对该内存空间初始化。

简单来说就是清理残留数据。

亮仔2个月前 (07-22)

反馈/建议

React 列表 & Keys

我们可以使用 JavaScript 的 map() 方法来创建列表。

React 实例

使用 map() 方法遍历数组生成了一个 1 到 5 的数字列表:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((numbers) =>
<li>{numbers}</li>
);
ReactDOM.render(
<ul>{listItems}</ul>,
document.getElementById('example')
);
```

尝试一下 »

我们可以将以上实例重构成一个组件，组件接收数组参数，每个列表元素分配一个 key，不然会出现警告 `a key should be provided for list items`，意思就是需要包含 key：

React 实例

```
function NumberList(props) {
const numbers = props.numbers;
const listItems = numbers.map((number) =>
<li key={number.toString()}>
{number}
</li>
);
return (
<ul>{listItems}</ul>
);
}
const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
<NumberList numbers={numbers} />,
document.getElementById('example')
);
```

尝试一下 »

Keys

Keys 可以在 DOM 中的某些元素被增加或删除的时候帮助 React 识别哪些元素发生了变化。因此你应当给数组中的每一个元素赋予一个确定的标识。

```
const numbers = [1, 2, 3, 4, 5];

const listItems = numbers.map((number) =>

  <li key={number.toString()}>

    {number}

  </li>

);
```

一个元素的 key 最好是这个元素在列表中拥有的一个独一无二的字符串。通常，我们使用来自数据的 id 作为元素的 key:

```
const todoItems = todos.map((todo) =>

  <li key={todo.id}>

    {todo.text}

  </li>

);
```

当元素没有确定的 id 时，你可以使用他的序列号索引 index 作为 key：

```
const todoItems = todos.map((todo, index) =>

  // 只有在没有确定的 id 时使用

  <li key={index}>

    {todo.text}

  </li>

);
```

如果列表可以重新排序，我们不建议使用索引来进行排序，因为这会导致渲染变得很慢。

用keys提取组件

元素的 key 只有在它和它的兄弟节点对比时才有意义。

比方说，如果你提取出一个 ListItem 组件，你应该把 key 保存在数组中的这个 `<ListItem />` 元素上，而不是放在 ListItem 组件中的 **<li>** 元素上。

错误的示范

```
function ListItem(props) {

  const value = props.value;

  return (

    // 错啦！你不需要在这里指定key:

    <li key={value.toString()}>

      {value}
```

```
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    //错啦！元素的key应该在这里指定：
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('example')
);
```

key的正确使用方式

React 实例

```
function ListItem(props) {
// 对啦！这里不需要指定key:
return <li>{props.value}</li>;
}
function NumberList(props) {
const numbers = props.numbers;
const listItems = numbers.map((number) =>
// 又对啦！key应该在数组的上下文中被指定
<ListItem key={number.toString()}
value={number} />
);
return (
<ul>
{listItems}
</ul>
);
}
const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
```

```
<NumberList numbers={numbers} />,
document.getElementById('example')
);
```

尝试一下 »

当你在 map() 方法的内部调用元素时，你最好随时记得为每一个元素加上一个独一无二的 key。

元素的 key 在他的兄弟元素之间应该唯一

数组元素中使用的 key 在其兄弟之间应该是独一无二的。然而，它们不需要是全局唯一的。当我们生成两个不同的数组时，我们可以使用相同的键。

React 实例

```
function Blog(props) {
const sidebar = (
<ul>
{props.posts.map((post) =>
<li key={post.id}>
{post.title}
</li>
)}
</ul>
);
const content = props.posts.map((post) =>
<div key={post.id}>
<h3>{post.title}</h3>
<p>{post.content}</p>
</div>
);
return (
<div>
{sidebar}
<hr />
{content}
</div>
);
}
const posts = [
{id: 1, title: 'Hello World', content: 'Welcome to learning React!'},
{id: 2, title: 'Installation', content: 'You can install React from npm.'}
];
ReactDOM.render(
<Blog posts={posts} />,
document.getElementById('example')
);
```

尝试一下 »

key 会作为给 React 的提示，但不会传递给你的组件。如果您的组件中需要使用和 key 相同的值，请将其作为属性传递：

```
const content = posts.map((post) =>

  <Post

    key={post.id}

    id={post.id}

    title={post.title} />

);
```

上面例子中，Post 组件可以读出 props.id，但是不能读出 props.key。

在 jsx 中嵌入 map()

在上面的例子中，我们声明了一个单独的 listItems 变量并将其包含在 JSX 中：

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()}
              value={number} />


  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

JSX 允许在大括号中嵌入任何表达式，所以我们可以在 map() 中这样使用：

React 实例

```
function NumberList(props) {
const numbers = props.numbers;
return (
<ul>
{numbers.map((number) =>
<ListItem key={number.toString()}
value={number} />
)}
</ul>
);
}
```

尝试一下 »

这么做有时可以使你的代码更清晰，但有时这种风格也会被滥用。就像在 JavaScript 中一样，何时需要为了可读性提取出一个变量，这完全取决于你。但请记住，如果一个 map() 嵌套了太多层级，那你就可以提取出组件。

React 条件渲染

JSX允许在大括号中嵌入任何表达式，需要注意的事项（请看注释）：

```
var ListItem = (props) => {       //es6中箭头函数
    return <li>{props.value}</li>;
}
```

```
function NumberList(props) {
    var numbers;    //声明在外面是因为 {} 中不能出现var,const,let等这种关键字
    return (
    <ul>
      {
        numbers = props.numbers,   //注意这里要加逗号

        numbers.map((number) =>
        <ListItem key={number}
         value={number} />
      )}
    </ul>
    );
}

var arr = [1,2,3];   //要传递的参数
ReactDOM.render(
    <NumberList numbers={arr}/>,  //这里的numbers就是props下的numbers,即props.numbers
    document.all('example')
);
```

阿凯1个月前 (08-22)

反馈/建议

React 组件 API

在本章节中我们将讨论 React 组件 API。我们将讲解以下7个方法:

设置状态：setState

替换状态：replaceState

设置属性：setProps

替换属性：replaceProps

强制更新：forceUpdate

获取DOM节点：findDOMNode

判断组件挂载状态：isMounted

设置状态:setState

```
setState(object nextState[, function callback])
```

参数说明

nextState，将要设置的新状态，该状态会和当前的**state**合并

callback，可选参数，回调函数。该函数会在**setState**设置成功，且组件重新渲染后调用。

合并nextState和当前state，并重新渲染组件。setState是React事件处理函数中和请求回调函数中触发UI更新的主要方法。

关于setState

不能在组件内部通过this.state修改状态，因为该状态会在调用setState()后被替换。

setState()并不会立即改变this.state，而是创建一个即将处理的state。setState()并不一定是同步的，为了提升性能React会批量执行state和DOM渲染。

setState()总是会触发一次组件重绘，除非在shouldComponentUpdate()中实现了一些条件渲染逻辑。

实例

React 实例

```
class Counter extends React.Component{
constructor(props) {
super(props);
this.state = {clickCount: 0};
this.handleClick = this.handleClick.bind(this);
}
handleClick() {
this.setState(function(state) {
return {clickCount: state.clickCount + 1};
});
}
render () {
return (<h2 onClick={this.handleClick}>点我！点击次数为: {this.state.clickCount}</h2>);
}
}
ReactDOM.render(
<Counter />,
document.getElementById('example')
);
```

尝试一下 »

实例中通过点击 h2 标签来使得点击计数器加 1。

替换状态：replaceState

```
replaceState(object nextState[, function callback])
```

nextState，将要设置的新状态，该状态会替换当前的**state**。

callback，可选参数，回调函数。该函数会在**replaceState**设置成功，且组件重新渲染后调用。

replaceState()方法与**setState()**类似，但是方法只会保留**nextState**中状态，原**state**不在**nextState**中的状态都会被删除。

设置属性：setProps

```
setProps(object nextProps[, function callback])
```

nextProps，将要设置的新属性，该状态会和当前的**props**合并

callback，可选参数，回调函数。该函数会在**setProps**设置成功，且组件重新渲染后调用。

设置组件属性，并重新渲染组件。

props相当于组件的数据流，它总是会从父组件向下传递至所有的子组件中。当和一个外部的JavaScript应用集成时，我们可能会需要向组件传递数据或通知**React.render()**组件需要重新渲染，可以使用**setProps()**。

更新组件，我可以在节点上再次调用**React.render()**，也可以通过**setProps()**方法改变组件属性，触发组件重新渲染。

替换属性：replaceProps

```
replaceProps(object nextProps[, function callback])
```

nextProps，将要设置的新属性，该属性会替换当前的**props**。

callback，可选参数，回调函数。该函数会在**replaceProps**设置成功，且组件重新渲染后调用。

replaceProps()方法与**setProps**类似，但它会删除原有 props。

强制更新：forceUpdate

```
forceUpdate([function callback])
```

参数说明

callback，可选参数，回调函数。该函数会在组件**render()**方法调用后调用。

forceUpdate()方法会使组件调用自身的render()方法重新渲染组件，组件的子组件也会调用自己的render()。但是，组件重新渲染时，依然会读取this.props和this.state，如果状态没有改变，那么React只会更新DOM。

forceUpdate()方法适用于this.props和this.state之外的组件重绘（如：修改了this.state后），通过该方法通知React需要调用render()

一般来说，应该尽量避免使用forceUpdate()，而仅从this.props和this.state中读取状态并由React触发render()调用。

获取DOM节点：findDOMNode

```
DOMElement findDOMNode()
```

返回值：DOM元素DOMElement

如果组件已经挂载到DOM中，该方法返回对应的本地浏览器 DOM 元素。当**render**返回**null** 或 **false**时，**this.findDOMNode()**也会返回**null**。从DOM中读取值的时候，该方法很有用，如：获取表单字段的值和做一些 DOM 操作。

判断组件挂载状态：isMounted

```
bool isMounted()
```

返回值：**true**或**false**，表示组件是否已挂载到DOM中

isMounted()方法用于判断组件是否已挂载到DOM中。可以使用该方法保证了**setState()**和**forceUpdate()**在异步场景下的调用不会出错。

本文参考：http://itbilu.com/javascript/react/EkACBdqKe.html

□ React Props　　React 组件生命周期 □

□ React 组件 API　　React AJAX □

React 组件生命周期

在本章节中我们将讨论 React 组件的生命周期。

组件的生命周期可分成三个状态：

Mounting：已插入真实 DOM

Updating：正在被重新渲染

Unmounting：已移出真实 DOM

生命周期的方法有：

componentWillMount 在渲染前调用,在客户端也在服务端。

componentDidMount : 在第一次渲染后调用，只在客户端。之后组件已经生成了对应的DOM结构，可以通过this.getDOMNode()来进行访问。 如果你想和其他JavaScript框架一起使用，可以在这个方法中调用setTimeout, setInterval或者发送AJAX请求等操作(防止异步操作阻塞UI)。

componentWillReceiveProps 在组件接收到一个新的 prop (更新后)时被调用。这个方法在初始化render时不会被调用。

shouldComponentUpdate 返回一个布尔值。在组件接收到新的props或者state时被调用。在初始化时或者使用forceUpdate时不被调用。可以在你确认不需要更新组件时使用。

componentWillUpdate在组件接收到新的props或者state但还没有render时被调用。在初始化时不会被调用。

componentDidUpdate 在组件完成更新后立即调用。在初始化时不会被调用。

componentWillUnmount在组件从 DOM 中移除之前立刻被调用。

这些方法的详细说明，可以参考官方文档。

以下实例在 Hello 组件加载以后，通过 componentDidMount 方法设置一个定时器，每隔100毫秒重新设置组件的透明度，并重新渲染：

React 实例

```
class Hello extends React.Component {
```

```
constructor(props) {
super(props);
this.state = {opacity: 1.0};
}
componentDidMount() {
this.timer = setInterval(function () {
var opacity = this.state.opacity;
opacity -= .05;
if (opacity < 0.1) {
opacity = 1.0;
}
this.setState({
opacity: opacity
});
}.bind(this), 100);
}
render () {
return (
<div style={{opacity: this.state.opacity}}>
Hello {this.props.name}
</div>
);
}
}
ReactDOM.render(
<Hello name="world"/>,
document.body
);
```

尝试一下 »

以下实例初始化 **state**， **setNewnumber** 用于更新 **state**。所有生命周期在 **Content** 组件中。

React 实例

```
class Button extends React.Component {
constructor(props) {
super(props);
this.state = {data: 0};
this.setNewNumber = this.setNewNumber.bind(this);
}
setNewNumber() {
this.setState({data: this.state.data + 1})
}
render() {
return (
<div>
<button onClick = {this.setNewNumber}>INCREMENT</button>
<Content myNumber = {this.state.data}></Content>
</div>
);
}
}
class Content extends React.Component {
componentWillMount() {
console.log('Component WILL MOUNT!')
}
componentDidMount() {
console.log('Component DID MOUNT!')
}
componentWillReceiveProps(newProps) {
console.log('Component WILL RECEIVE PROPS!')
}
shouldComponentUpdate(newProps, newState) {
return true;
}
componentWillUpdate(nextProps, nextState) {
console.log('Component WILL UPDATE!');
}
componentDidUpdate(prevProps, prevState) {
console.log('Component DID UPDATE!')
}
```

```
componentWillUnmount() {
console.log('Component WILL UNMOUNT!')
}
render() {
return (
<div>
<h3>{this.props.myNumber}</h3>
</div>
);
}
}
ReactDOM.render(
<div>
<Button />
</div>,
document.getElementById('example')
);
```

尝试一下 »

React 组件 API　　React AJAX

点我分享笔记

反馈/建议

React 组件生命周期　　React 表单与事件

React AJAX

React 组件的数据可以通过 componentDidMount 方法中的 Ajax 来获取，当从服务端获取数据时可以将数据存储在 state 中，再用 this.setState 方法重新渲染 UI。

当使用异步加载数据时，在组件卸载前使用 componentWillUnmount 来取消未完成的请求。

以下实例演示了获取 Github 用户最新 gist 共享描述:

React 实例

```
class UserGist extends React.Component {
constructor(props) {
super(props);
this.state = {username: '', lastGistUrl: ''};
}
componentDidMount() {
this.serverRequest = $.get(this.props.source, function (result) {
var lastGist = result[0];
this.setState({
username: lastGist.owner.login,
lastGistUrl: lastGist.html_url
});
}.bind(this));
}
componentWillUnmount() {
```

```
this.serverRequest.abort();
}
render() {
return (
<div>
{this.state.username} 用户最新的 Gist 共享地址:
<a href={this.state.lastGistUrl}>{this.state.lastGistUrl}</a>
</div>
);
}
}
ReactDOM.render(
<UserGist source="https://api.github.com/users/octocat/gists" />,
document.getElementById('example')
);
```

尝试一下 »

以上代码使用 jQuery 完成 Ajax 请求。

React 组件生命周期 React 表单与事件

React AJAX React Refs

React 表单与事件

本章节我们将讨论如何在 React 中使用表单。

HTML 表单元素与 React 中的其他 DOM 元素有所不同,因为表单元素生来就保留一些内部状态。

在 HTML 当中，像 <input>, <textarea>, 和 <select> 这类表单元素会维持自身状态，并根据用户输入进行更新。但在React中，可变的状态通常保存在组件的状态属性中，并且只能用 setState() 方法进行更新。

一个简单的实例

在实例中我们设置了输入框 input 值 **value = {this.state.data}**。在输入框值发生变化时我们可以更新 state。我们可以使用 **onChange** 事件来监听 input 的变化，并修改 state。

React 实例

```
class HelloMessage extends React.Component {
constructor(props) {
super(props);
this.state = {value: 'Hello Runoob!'};
this.handleChange = this.handleChange.bind(this);
}
handleChange(event) {
this.setState({value: event.target.value});
}
render() {
var value = this.state.value;
```

```
return <div>
<input type="text" value={value} onChange={this.handleChange} />
<h4>{value}</h4>
</div>;
}
}
ReactDOM.render(
<HelloMessage />,
document.getElementById('example')
);
```

尝试一下 »

上面的代码将渲染出一个值为 Hello Runoob! 的 input 元素，并通过 onChange 事件响应更新用户输入的值。

实例 2

在以下实例中我们将为大家演示如何在子组件上使用表单。 **onChange** 方法将触发 state 的更新并将更新的值传递到子组件的输入框的 **value** 上来重新渲染界面。

你需要在父组件通过创建事件句柄 (**handleChange**)，并作为 prop (**updateStateProp**) 传递到你的子组件上。

React 实例

```
class Content extends React.Component {
render() {
return <div>
<input type="text" value={this.props.myDataProp} onChange={this.props.updateStateProp} />
<h4>{this.props.myDataProp}</h4>
</div>;
}
}
class HelloMessage extends React.Component {
constructor(props) {
super(props);
this.state = {value: 'Hello Runoob!'};
this.handleChange = this.handleChange.bind(this);
}
handleChange(event) {
this.setState({value: event.target.value});
}
render() {
var value = this.state.value;
return <div>
<Content myDataProp = {value}
updateStateProp = {this.handleChange}></Content>
</div>;
}
}
ReactDOM.render(
<HelloMessage />,
document.getElementById('example')
);
```

尝试一下 »

Select 下拉菜单

在 React 中，不使用 selected 属性，而在根 select 标签上用 value 属性来表示选中项。

React 实例

```
class FlavorForm extends React.Component {
constructor(props) {
super(props);
this.state = {value: 'coconut'};
this.handleChange = this.handleChange.bind(this);
this.handleSubmit = this.handleSubmit.bind(this);
}
handleChange(event) {
this.setState({value: event.target.value});
}
```

```
handleSubmit(event) {
alert('Your favorite flavor is: ' + this.state.value);
event.preventDefault();
}
render() {
return (
<form onSubmit={this.handleSubmit}>
<label>
选择您最喜欢的网站
<select value={this.state.value} onChange={this.handleChange}>
<option value="gg">Google</option>
<option value="rn">Runoob</option>
<option value="tb">Taobao</option>
<option value="fb">Facebook</option>
</select>
</label>
<input type="submit" value="提交" />
</form>
);
}
}
ReactDOM.render(
<FlavorForm />,
document.getElementById('example')
);
```

尝试一下 »

多个表单

当你有处理多个 input 元素时，你可以通过给每个元素添加一个 name 属性，来让处理函数根据 `event.target.name` 的值来选择做什么。

React 实例

```
class Reservation extends React.Component {
constructor(props) {
super(props);
this.state = {
isGoing: true,
numberOfGuests: 2
};
this.handleInputChange = this.handleInputChange.bind(this);
}
handleInputChange(event) {
const target = event.target;
const value = target.type === 'checkbox' ? target.checked : target.value;
const name = target.name;
this.setState({
[name]: value
});
}
render() {
return (
<form>
<label>
是否离开:
<input
name="isGoing"
type="checkbox"
checked={this.state.isGoing}
onChange={this.handleInputChange} />
</label>
<br />
<label>
访客数:
<input
name="numberOfGuests"
type="number"
value={this.state.numberOfGuests}
onChange={this.handleInputChange} />
</label>
</form>
```

```
);
}
}
```

尝试一下 »

React 事件

以下实例演示通过 onClick 事件来修改数据：

React 实例

```
class HelloMessage extends React.Component {
constructor(props) {
super(props);
this.state = {value: 'Hello Runoob!'};
this.handleChange = this.handleChange.bind(this);
}
handleChange(event) {
this.setState({value: '菜鸟教程'})
}
render() {
var value = this.state.value;
return <div>
<button onClick={this.handleChange}>点我</button>
<h4>{value}</h4>
</div>;
}
}
ReactDOM.render(
<HelloMessage />,
document.getElementById('example')
);
```

尝试一下 »

当你需要从子组件中更新父组件的 **state** 时，你需要在父组件通过创建事件句柄 (**handleChange**)，并作为 prop (**updateStateProp**) 传递到你的子组件上。实例如下：

React 实例

```
class Content extends React.Component {
render() {
return <div>
<button onClick = {this.props.updateStateProp}>点我</button>
<h4>{this.props.myDataProp}</h4>
</div>
}
}
class HelloMessage extends React.Component {
constructor(props) {
super(props);
this.state = {value: 'Hello Runoob!'};
this.handleChange = this.handleChange.bind(this);
}
handleChange(event) {
this.setState({value: '菜鸟教程'})
}
render() {
var value = this.state.value;
return <div>
<Content myDataProp = {value}
updateStateProp = {this.handleChange}></Content>
</div>;
}
}
ReactDOM.render(
<HelloMessage />,
document.getElementById('example')
);
```

尝试一下 »

React AJAX

React Refs

点我分享笔记

反馈/建议

React Refs

React 支持一种非常特殊的属性 **Ref** ，你可以用来绑定到 render() 输出的任何组件上。

这个特殊的属性允许你引用 render() 返回的相应的支撑实例（ backing instance ）。这样就可以确保在任何时间总是拿到正确的实例。

使用方法

绑定一个 ref 属性到 render 的返回值上：

```
<input ref="myInput" />
```

在其它代码中，通过 this.refs 获取支撑实例:

```
var input = this.refs.myInput;

var inputValue = input.value;

var inputRect = input.getBoundingClientRect();
```

完整实例

你可以通过使用 this 来获取当前 React 组件，或使用 ref 来获取组件的引用，实例如下：

React 实例

```
class MyComponent extends React.Component {
handleClick() {
// 使用原生的 DOM API 获取焦点
this.refs.myInput.focus();
}
render() {
// 当组件插入到 DOM 后，ref 属性添加一个组件的引用于到 this.refs
return (
<div>
<input type="text" ref="myInput" />
```

```
<input
type="button"
value="点我输入框获取焦点"
onClick={this.handleClick.bind(this)}
/>
</div>
);
}
}
ReactDOM.render(
<MyComponent />,
document.getElementById('example')
);
```

尝试一下 »

实例中，我们获取了输入框的支撑实例的引用，子点击按钮后输入框获取焦点。

我们也可以使用 getDOMNode()方法获取DOM元素

□ React 表单与事件　　React 元素渲染 □