--

[react.js 1](#_Toc28460)

[理论方面 1](#_Toc26445)

[JSX 2](#_Toc11371)

[JSX 2](#_Toc14276)

[jsx 转换 2](#_Toc7184)

[React.createElement 2](#_Toc22740)

[map 2](#_Toc12155)

[map 2](#_Toc26335)

[展开属性 2](#_Toc10267)

[HTML 实体 3](#_Toc8669)

[转义 3](#_Toc24873)

[注释 3](#_Toc11964)

[表达式 3](#_Toc12298)

[API(state props) 3](#_Toc2236)

[demo 4](#_Toc22723)

[点击标签来使得点击计数器加 1 4](#_Toc19076)

[componentDidMount 中启动定时器 4](#_Toc16179)

[组件的复用 4](#_Toc29956)

[用户输入同步 4](#_Toc743)

[结合使用Ajax 5](#_Toc19051)

[把一个Promise对象传入组件 5](#_Toc25286)

## react.js

理论方面

|  |  |
| --- | --- |
| 学习网址 | react  <https://facebook.github.io/react/docs/hello-world.html>  中文文档地址:  <https://doc.react-china.org>  <http://www.css88.com/react/>  react china:（社区）  <http://react-china.org/>  React.js 小书  <http://huziketang.com/books/react/>  IT笔录 = [首页](https://itbilu.com/)[JavaScript](https://itbilu.com/javascript)React  <https://itbilu.com/javascript/react>  阮一峰 React 技术栈系列教程  <http://www.ruanyifeng.com/blog/2016/09/react-technology-stack.html>  material-ui 来自react（A Set of React Components that ImplementGoogle's Material Design）  <http://www.material-ui.com/#/get-started/usage> |
|  |  |
|  | CSS预处理器LESS、SASS使用教程  <https://itbilu.com/other/relate/E1PlpkRL-.html> |
|  | [一看就懂的ReactJs入门教程-精华版](http://www.cnblogs.com/yunfeifei/p/4486125.html)  <http://www.cnblogs.com/yunfeifei/p/4486125.html>  React基础——JSX语法  <http://www.jianshu.com/p/7e872afeae42> |
|  |  |
| 基础概念 | > 简介  facebook出品的reactjs核心是：封装UI；  体现：封装定义一大段JS代码，一句话进行输出，产生一个UI组件  过程：通过写jsx文件，编译生成一段js文件，运行这段JS；  > 特点  react不是一个完整的MVC，MVVM框架；  react跟WEB Components不冲突；  react 的特点是轻  存在组件化开发的思路---封装具有独立功能的UI组件，小组件可以通过组合、嵌套以构成大组件，意味着这些组件可以高度可重用  > 引用场景  复杂场景，同时要求高性能  组件是高度可重用的 |
|  |
|  |  |
|  |  |
| 小结 | 首先，对于React，有一些认识误区，这里先总结一下：  React不是一个完整的MVC框架，最多可以认为是MVC中的V（View），甚至React并不非常认可MVC开发模式；  React的服务器端Render能力只能算是一个锦上添花的功能，并不是其核心出发点，事实上React官方站点几乎没有提及其在服务器端的应用；  有人拿React和Web Component相提并论，但两者并不是完全的竞争关系，你完全可以用React去开发一个真正的Web Component；  React不是一个新的模板语言，JSX只是一个表象，没有JSX的React也能工作。  1、ReactJS的背景和原理  　　在Web开发中，我们总需要将变化的数据实时反应到UI上，这时就需要对DOM进行操作。而复杂或频繁的DOM操作通常是性能瓶颈产生的原因（如何进行高性能的复杂DOM操作通常是衡量一个前端开发人员技能的重要指标）。React为此引入了虚拟DOM（Virtual DOM）的机制：在浏览器端用Javascript实现了一套DOM API。基于React进行开发时所有的DOM构造都是通过虚拟DOM进行，每当数据变化时，React都会重新构建整个DOM树，然后React将当前整个DOM树和上一次的DOM树进行对比，得到DOM结构的区别，然后仅仅将需要变化的部分进行实际的浏览器DOM更新。而且React能够批处理虚拟DOM的刷新，在一个事件循环（Event Loop）内的两次数据变化会被合并，例如你连续的先将节点内容从A变成B，然后又从B变成A，React会认为UI不发生任何变化，而如果通过手动控制，这种逻辑通常是极其复杂的。尽管每一次都需要构造完整的虚拟DOM树，但是因为虚拟DOM是内存数据，性能是极高的，而对实际DOM进行操作的仅仅是Diff部分，因而能达到提高性能的目的。这样，在保证性能的同时，开发者将不再需要关注某个数据的变化如何更新到一个或多个具体的DOM元素，而只需要关心在任意一个数据状态下，整个界面是如何Render的。  如果你像在90年代那样写过服务器端Render的纯Web页面那么应该知道，服务器端所要做的就是根据数据Render出HTML送到浏览器端。如果这时因为用户的一个点击需要改变某个状态文字，那么也是通过刷新整个页面来完成的。服务器端并不需要知道是哪一小段HTML发生了变化，而只需要根据数据刷新整个页面。换句话说，任何UI的变化都是通过整体刷新来完成的。而React将这种开发模式以高性能的方式带到了前端，每做一点界面的更新，你都可以认为刷新了整个页面。至于如何进行局部更新以保证性能，则是React框架要完成的事情。  　　借用Facebook介绍React的视频中聊天应用的例子，当一条新的消息过来时，传统开发的思路如上图，你的开发过程需要知道哪条数据过来了，如何将新的DOM结点添加到当前DOM树上；而基于React的开发思路如下图，你永远只需要关心数据整体，两次数据之间的UI如何变化，则完全交给框架去做。可以看到，使用React大大降低了逻辑复杂性，意味着开发难度降低，可能产生Bug的机会也更少。    2、组件化  　　虚拟DOM(virtual-dom)不仅带来了简单的UI开发逻辑，同时也带来了组件化开发的思想，所谓组件，即封装起来的具有独立功能的UI部件。React推荐以组件的方式去重新思考UI构成，将UI上每一个功能相对独立的模块定义成组件，然后将小的组件通过组合或者嵌套的方式构成大的组件，最终完成整体UI的构建。例如，Facebook的instagram.com整站都采用了React来开发，整个页面就是一个大的组件，其中包含了嵌套的大量其它组件，大家有兴趣可以看下它背后的代码。  如果说MVC的思想让你做到视图-数据-控制器的分离，那么组件化的思考方式则是带来了UI功能模块之间的分离。我们通过一个典型的Blog评论界面来看MVC和组件化开发思路的区别。  对于MVC开发模式来说，开发者将三者定义成不同的类，实现了表现，数据，控制的分离。开发者更多的是从技术的角度来对UI进行拆分，实现松耦合。  对于React而言，则完全是一个新的思路，开发者从功能的角度出发，将UI分成不同的组件，每个组件都独立封装。  在React中，你按照界面模块自然划分的方式来组织和编写你的代码，对于评论界面而言，整个UI是一个通过小组件构成的大组件，每个组件只关心自己部分的逻辑，彼此独立。  IMG_256  React认为一个组件应该具有如下特征：  （1）可组合（Composeable）：一个组件易于和其它组件一起使用，或者嵌套在另一个组件内部。如果一个组件内部创建了另一个组件，那么说父组件拥有（own）它创建的子组件，通过这个特性，一个复杂的UI可以拆分成多个简单的UI组件；  （2）可重用（Reusable）：每个组件都是具有独立功能的，它可以被使用在多个UI场景；  （3）可维护（Maintainable）：每个小的组件仅仅包含自身的逻辑，更容易被理解和维护； |
| 为什么使用 React | 我们创造 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。为了达到这个目标，React 采用下面两个主要的思想。  > 简单  仅仅只要表达出你的应用程序在任一个时间点应该长的样子，然后当底层的数据变了，React 会自动处理所有用户界面的更新。  > 声明式 (Declarative)  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。  - 构建可组合的组件  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。 |

JSX

|  |  |  |
| --- | --- | --- |
| JSX | JSX是React的核心组成部分，它使用XML标记的方式去直接声明界面，界面组件之间可以互相嵌套。可以理解为在JS中编写与XML类似的语言,一种定义带属性树结构（DOM结构）的语法，它的目的不是要在浏览器或者引擎中实现，它的目的是通过各种编译器将这些标记编译成标准的JS语言。  虽然你可以完全不使用JSX语法，只使用JS语法，但还是推荐使用JSX，可以定义包含属性的树状结构的语法，类似HTML标签那样的使用，而且更便于代码的阅读。  使用JSX语法后，你必须要引入babel的JSX解析器，把JSX转化成JS语法，这个工作会由babel自动完成。同时引入babel后，你就可以使用新的es6语法，babel会帮你把es6语法转化成es5语法，兼容更多的浏览器。 | |
| 我们坚信组件是正确方法去做关注分离，而不是通过 模板 和 展示逻辑。我们认为标签和生成它的代码是紧密相连的。此外，展示逻辑通常是很复杂的，通过模板语言实现这些逻辑会产生大量代码。  我们得出解决这个问题最好的方案是通过 JavaScript 直接生成模板，这样你就可以用一个真正语言的所有表达能力去构建用户界面。为了使这变得更简单，我们做了一个非常简单、可选类似 HTML 语法 ，通过函数调用即可生成模板的编译器，称为 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：  var a = React.createElement('a', {href: 'http://facebook.github.io/react/'}, 'Hello React!')  通过 JSX 这就变成了:  var a = <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，设计师也偏向用这用语法，但是每个人可以有它们自己的工作流，所以JSX 不是必须用的。  JSX 非常小；上面“hello, world”的例子使用了 JSX 所有的特性。想要了解更多，请看[深入理解 JSX](http://wiki.jikexueyuan.com/project/react/jsx-depth.html)。或者直接使用在线 JSX 编译器观察变化过程。  JSX 类似于 HTML，但不是完全一样。参考 [JSX 陷阱](http://wiki.jikexueyuan.com/project/react/jsx-gotchas.html) 学习关键区别。  最简单开始学习 JSX 的方法就是使用浏览器端的 JSXTransformer。我们强烈建议你不要在生产环境中使用它。你可以通过我们的命令行工具 [react-tools](http://npmjs.org/package/react-tools) 包来预编译你的代码。(babel)  你完全可以选择是否使用 JSX，并不是 React 必须的。  > JSX的特点：  类XML语法容易接受，结构清晰  增强JS语义  抽象程度高，屏蔽DOM操作，跨平台  代码模块化  > JSX基本语法规则：  JSX本身就和XML语法类似，可以定义属性以及子元素。唯一特殊的是可以用大括号来加入JavaScript表达式。遇到 HTML 标签（以 < 开头），就用 HTML 规则解析；遇到代码块（以 { 开头），就用 JavaScript 规则解析。  > campare  可以使用 [facebook compile](https://babeljs.io/repl/" \t "http://www.jianshu.com/p/_blank) 工具查看比较两者转换  [https://babeljs.io/repl/](https://babeljs.io/repl/" \t "http://www.jianshu.com/p/_blank) | |
| jsx 转换 | HTML 语言直接写在 JavaScript 语言之中，不加任何引号，这就是 JSX 的语法，它允许 HTML 与 JavaScript 的混写：  ReactDOM.render(<h1>Hello, world!</h1>, document.getElementById('example'));  如果转化成纯JavaScript的话，就是:  ReactDOM.render(React.DOM.h1(null, 'hello,world!'), document.getElementById('example')); | |
| <body>  <div id = "container"></div>  </body>  </html>  <script src="react.js"></script>  <script src="react-dom.js"></script>  <script src="browser.min.js"></script>  <script type="text/babel">  ReactDOM.render(  <h1>Hello, world!</h1>,  document.querySelector('#container')  );  </script> | > <script> 标签的 type 属性为 text/babel，这是React 独有的 JSX 语法，跟 JavaScript 不兼容。凡是在页面中直接使用 JSX 的地方，都要加上 type="text/babel"。  > 一共用了三个库： react.js 、react-dom.js 和 browser.min.js ，它们必须首先加载。其中，react.js 是 React 的核心库，react-dom.js 是提供与 DOM 相关的功能， browser.min.js的作用是将 JSX 语法转为 JavaScript 语法。  将 JSX 语法转为 JavaScript 语法，这一步很消耗时间。现在前端项目，都会使用前端工程化，不会直接在html页面中直接写js代码，写好的js代码都会使用工具进行编译压缩等。这样的话，我们的jsx也会通过编译直接转化成js语法，让浏览器直接使用。  > ReactDOM.render 是 React 的最基本方法，ReactDOM.render()将JSX转换成HTML，并且将其插入特定的DOM节点。在0.14版本之前，使用的是React.render()方法，目前还支持，但是React不推荐使用。 |
|  | |
| var Nav;  var app = <Nav color="blue" />; | var Nav;  var app = React.createElement(Nav, {color:"blue"}); |
| var Nav, Profile;  var app = <Nav color="blue"><Profile>click</Profile></Nav>; | var Nav, Profile;  var app = React.createElement( Nav, {color:"blue"}, React.createElement(Profile, null, "click")); |
| React.render(  <**div**>  <**div**>  <**div**>content</**div**>  </**div**>  </**div**>,  document.getElementById('example') ); | React.render(  React.createElement('div', **null**,  React.createElement('div', **null**,  React.createElement('div', **null**, 'content')  )  ),  document.getElementById('example') ); |
| 我们写一个XML标签，实质上就是在调用React.createElement这个方法，并返回一个ReactElement对象。  ReactElement createElement(  string/ReactClass type,  [object props],  [children ...]  )  这个方法的第一个参数可以是一个字符串，表示是一个HTML标准内的元素，或者是一个ReactClass类型的对象，表示我们之前封装好的自定义组件。第二个参数是一个对象，或者说是字典也可以，它保存了这个元素的所有固有属性（即传入后基本不会改变的值）。从第三个参数开始，之后的参数都被认作是元素的子元素。 | |
| React.createElement | 你可以通过 React.createElement 来创建一个树。第一个参数是标签，第二个参数是一个属性对象，每三个是子节点。  var child = React.createElement('li', null, 'Text Content');  var root = React.createElement('ul', { className: 'my-list' }, child);  React.render(root, document.getElementById('example'));  方便起见，你可以创建基于自定义组件的速记工厂方法。  var Factory = React.createFactory(ComponentClass);  ...  var root = Factory({ custom: 'prop' });  React.render(root, document.getElementById('example'));  React 已经为 HTML 标签提供内置工厂方法。  var root = React.DOM.ul({ className: 'my-list' },  React.DOM.li(null, 'Text Content')  ); | |
| map | //声明了一个names数组，然后遍历在前面加上Hello,输出到DOM中:  var names = ['Jack', 'Tom', 'Alice'];  var doms = <div>{  names.map(function (name) {  return <div>Hello, {name}!</div>  })  }</div>;  render(doms, document.getElementById('root')); | |
| map | //JSX 允许直接在模板插入 JavaScript 变量。  var arr = [<h1>Hello world!</h1>, <h2>React is perfect!</h2>];  // 1  render(<div>{arr}</div>, document.getElementById('container'));  // 2  arr.map(function (n, i) {  render(n, document.getElementById('d' + i));  }) | |
| 展开属性 | > 展开属性  如果你事先知道组件需要的全部 Props（属性），JSX 很容易地这样写：  var component = <Component foo={x} bar={y} />;  如果你不知道要设置哪些 Props，那么现在最好不要设置它：  var component = <Component />;  component.props.foo = x; // 不好  component.props.bar = y; // 不好  这样是反模式，因为 React 不能帮你检查属性类型（propTypes）。这样即使你的 属性类型有错误也不能得到清晰的错误提示。  Props 应该被当作禁止修改的。修改 props 对象可能会导致预料之外的结果，所以最好不要去修改 props 对象。  这样写是错误的，因为我们手动直接添加的属性React后续没办法检查到属性类型错误，也就是说，当我们手动添加的属性发生类型错误时，在控制台是看不到错误信息的。  在React的设定中，初始化完props后，props是不可变的。改变props会引起无法想象的后果。  为了解决这个问题，React引入了属性延伸  > 展开属性（Spread Attributes）  可以使用 JSX 的新特性 - 展开属性；  var props = {};  props.foo = x;  props.bar = y;  var component = <Component {...props} />;  传入对象的属性会被复制到组件内。  它能被多次使用，也可以和其它属性一起用。注意顺序很重要，后面的会覆盖掉前面的。  var props = { foo: 'default' };  var component = <Component {...props} foo={'override'} />;  console.log(component.props.foo); // 'override' | |
| HTML 实体 | <div>First · Second</div>  如果想在 JSX 表达式中显示 HTML 实体，可以会遇到二次转义的问题，因为 React 默认会转义所有字符串，为了防止各种 XSS 攻击。  <div>{'First · Second'}</div> // 错误: 会显示 “First · Second”  有多种绕过的方法。最简单的是直接用 Unicode 字符。这时要确保文件是 UTF-8 编码且网页也指定为 UTF-8 编码。  <div>{'First · Second'}</div>  安全的做法是先找到 [实体的 Unicode 编号](http://www.fileformat.info/info/unicode/char/b7/index.htm" \t "http://www.jianshu.com/p/_blank) ，然后在 JavaScript 字符串里使用。  <div>{'First \u00b7 Second'}</div> <div>{'First ' + String.fromCharCode(183) + ' Second'}</div>  可以在数组里混合使用字符串和 JSX 元素。  <div>{['First ', <span>·</span>, ' Second']}</div>  万不得已，可以直接使用原始 HTML。  <div **dangerouslySetInnerHTML**={{'{{'}}\_\_html: 'First · Second'}} />  如果往原生 HTML 元素里传入 HTML 规范里不存在的属性，React 不会显示它们。如果需要使用自定义属性，要加data-前缀。  <div data-custom-attribute="foo" />  以 aria-开头的 [网络无障碍] 属性可以正常使用。  <div aria-hidden={true} /> | |
| 转义 | 比如我们有一些内容是用户输入的富文本，从后台取到数据后展示在页面上，希望展示相应的样式  js 代码:  var content='<strong>content</strong>';    React.render(  <div>{content}</div>,  document.body  );  结果页面直接输出内容了：<strong>content</strong>  React默认会进行HTML的转义，避免XSS攻击，如果要不转义，可以这么写：  var content='<strong>content</strong>';  React.render(  <div **dangerouslySetInnerHTML**={{\_\_html: content}}></div>,  document.body  ); | |
| 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>  ); | |
| 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。  表达式写法  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  render: function() {  return <div>Hello {this.props.name?this.props.name : "World"}</div>;  }  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。  render: function() {  return <div>Hello {this.props.name || "World"}</div>;  }  > 属性表达式  var person = <Person name={window.isLoggedIn ? window.name : ''} />;  > 子节点表达式  var content = <Container>{ **window.isLoggedIn ? <Nav /> : <Login />** }</Container>;  要注意的是，JSX语法只是语法糖，它的背后是调用ReactElement的构造方法React.createElement的，所以类似这样的写法是不可以的：  <div id={if (condition) { 'msg' }}>Hello World!</div>  React.createElement("div", {id: if (condition) { 'msg' }}, "Hello World!");  可以从转化后的Javascript代码中看出明显的语法错误，所以要不用三目运算符，要不就这样写：  if (condition) <div id='msg'>Hello World!</div> else <div>Hello World!</div>  函数返回值写法  class HelloMessage extends Component {  constructor(props) {  super(props);  }  render() {  return (<div>  Hello {  (function (obj) {  if (obj.props.name)  return obj.props.name  else  return "World"  }(this))  }  </div>);  }  };  render(<HelloMessage name="xiaowang" />, document.getElementById("root")); | |

API(state props)

|  |  |
| --- | --- |
|  | 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。 |
|  | 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。 |
|  |  |
|  |  |

demo

|  |  |
| --- | --- |
|  | npm init  npm i webpack -S (Install Globally too)  npm i babel-loader babel-preset-es2015 babel-preset-react babel-core -S  npm i react react-dom -S |
|  | // The ES5 way  var Photo = React.createClass({  handleDoubleTap: function(e) { … },  render: function() { … },  });  // The ES6+ way  class Photo extends React.Component {  handleDoubleTap(e) { … }  render() { … }  } |
| 点击标签来使得点击计数器加 1 | var Counter = React.createClass({  getInitialState: function () {  return {clickCount: 0};  },  handleClick: function () {  this.setState(function (state) {  return {clickCount: state.clickCount + 1};  });  },  render: function () {  return (<h2 onClick={this.handleClick}>点我！点击次数为: {this.state.clickCount}</h2>);  } }); ReactDOM.render(<Counter />, document.getElementById('message')); |
|  | import React, {Component} from 'react'; import {render} from 'react-dom'; import PropTypes from 'prop-types';  export default class Counter extends Component {  static propTypes = { // 限制类型  initialCount: PropTypes.number  }  static defaultProps = { // 默认值  initialCount: 1  }   constructor(props) {// 入参  super(props);// 组件基类  this.state = {count: this.props.initialCount};// 可以作为实例属性写在类内部  this.addOne = this.addOne.bind(this);  }   addOne() {  this.setState({count: ~~this.state.count + 1});  }   render() {  return <div onClick={this.addOne} tyle={{color: 'blue'}}>{this.state.count}</div>;  } }  <Test1 initialCount={2}></Test1> |
| componentDidMount 中启动定时器 | import React, { Component } from 'react';  import { render } from 'react-dom';  class Hello extends Component {  constructor(props) {  super(props);  this.state = {  opacity: 1  }  }  componentDidMount() {  this.deg = 0;  this.timer = setInterval(function () {  this.deg += 10;  this.deg >= 360 && (this.deg = 0);  this.setState({ opacity: Math.cos(this.deg \* Math.PI / 180) });  }.bind(this), 100);  }  render() {  return (  <div style={{ opacity: this.state.opacity }}> Hello {this.props.name} </div>  );  }  };  render(<Hello />, document.getElementById("root")); |
| 组件的复用 | import React, {Component} from 'react'; import {render} from 'react-dom'; import PropTypes from 'prop-types';  class Search extends Component {  constructor(props) {  super(props);  }   render() {  return (  <div>  {this.props.searchType}:<input type="text"/>  <button>Search</button>  </div>  );  } }  /\*应用方式1\*/ render(<div>  <Search searchType="Title"/>  <Search searchType="Content"/> </div>, document.getElementById('root'));  /\*应用方式2\*/ class Page extends Component {  render() {  return (<div>  <Search searchType="Title"/>  <Search searchType="Content"/>  </div>);  } }  render(<Page/>, document.getElementById('root')); |
| 用户输入同步 | import React, {Component} from 'react'; import {render} from 'react-dom';  // 用户在表单填入的内容，属于用户跟组件的互动，所以不能用 this.props 读取； // 而要定义一个 onChange事件的回调函数，通过 event.target.value 读取用户输入的值。textarea 元素、select元素、radio元素都属于这种情况 class Input extends Component {  constructor(props) {  super(props);  this.state = {  value: "Hello"  };  this.handleChange = this.handleChange.bind(this);  }   handleChange(event) {  this.setState({value: event.target.value});  }   render() {  return (  <div>  <input type="text" value={this.state.value} onChange={this.handleChange}/>  <p>{this.state.value}</p>  <p>{this.state.value}</p>  </div>  );  } }; render(<Input/>, document.getElementById("root")); |
| 结合使用Ajax | 组件的数据来源，通常是通过 Ajax 请求从服务器获取，可以使用 componentDidMount 方法设置 Ajax 请求，等到请求成功，再用 this.setState 方法重新渲染 UI （查看 [demo11](https://github.com/ruanyf/react-demos/blob/master/demo11/index.html" \t "http://www.ruanyifeng.com/blog/2015/03/_blank) ）。  **import** React, {**Component**} **from** 'react'; **import** {render} **from** 'react-dom';  **class** UserGist **extends Component** {  constructor(props) {  **super**(props);  **this**.state = {  username: '',  lastGistUrl: ''  }  }   componentDidMount() {  $.get(**this**.props.source, **function** (result) {  **var** lastGist = result[0];  **if** (**this**.isMounted()) {  **this**.setState({  username: lastGist.owner.login,  lastGistUrl: lastGist.html\_url  });  }  }.bind(**this**));  }   render() {  **return** (  <**div**>  {**this**.state.username}'s last gist is  <**a href=**{**this**.state.lastGistUrl}>here</**a**>.  </**div**>  );  } }  render(<**UserGist source="https://api.github.com/users/octocat/gists"**/>, **document**.getElementById('root'));  上面代码使用 jQuery 完成 Ajax 请求，这是为了便于说明。React 本身没有任何依赖，完全可以不用jQuery，而使用其他库。 |
| 把一个Promise对象传入组件 | import React, {Component} from 'react'; import {render} from 'react-dom';  class RepoList extends Component {  constructor(props) {  super(props);  this.state = {  loading: true,  error: null,  data: null  }  }   componentDidMount() {  this.props.getJsonPro.then(  value => this.setState({loading: false, data: value}),  error => this.setState({loading: false, error: error})  );  }   render() {  if (this.state.loading) {  return <span>Loading...</span>;  } else if (this.state.error !== null) {  return <span>Error: {this.state.error.message}</span>;  } else {  var repos = this.state.data.items;  var repoList = repos.map(function (repo, index) {  return (  <li key={index}>  <a href={repo.html\_url}>{repo.name}</a> ({repo.stargazers\_count} stars) <br/> {repo.description}  </li>  );  });  return (  <main>  <h1>Most Popular JavaScript Projects in Github</h1>  <ol>{repoList}</ol>  </main>  );  }  } }  let jsonPromise = $.getJSON('https://api.github.com/search/repositories?q=javascript&sort=stars'); render(<RepoList getJsonPro={jsonPromise}/>, document.getElementById('root'));  上面代码从Github的API抓取数据，然后将Promise对象作为属性，传给RepoList组件。  如果Promise对象正在抓取数据（pending状态），组件显示"正在加载"；如果Promise对象报错（rejected状态），组件显示报错信息；如果Promise对象抓取数据成功（fulfilled状态），组件显示获取的数据。 |
|  |  |