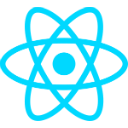
# React 教程



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。

在开始学习 React 之前，您需要具备以下基础知识：

* HTML5
* CSS
* JavaScript

# React 安装

React 可以直接下载使用，官方也提供了很多学习的实例。

### **使用实例**

以下实例输出了 Hello, world!

<!DOCTYPE html>  
<**html** lang=**"en"**>  
<**head**>  
 <**meta** charset=**"UTF-8"**>  
 <**title**>Hello, world!</**title**>  
 <**script** src=**"../react/react-15.4.0/build/react.min.js"**></**script**>  
 <**script** src=**"../react/react-15.4.0/build/react-dom.min.js"**></**script**>  
 *<!--让浏览器支持babel语法-->* <**script** src=**"../react/react-15.4.0/build/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**>

**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** - 用于将 JSX 语法转为 JavaScript 语法

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（自行了解）

### 自动编译

安装离线转换工具，在项目根目录下执行命令：

依赖npm

**cnpm install -g react-tools**

将含有JSX格式的文件转换成标准的JavaScript文件，命令：

**jsx --watch src/ build/**

一旦修改src目录下的文件，build目录下会自动生成对应的文件。

下面对比一下编译前和编译后的JS文件：

编译前：

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

编译后：

***ReactDOM***.**render**(  
 ***React***.**createElement**(**"h1"**, **null**, **"Hello, world!"**),  
 **document**.getElementById(**'example'**)  
);

# 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。

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

*注意：多个标签情况下外边一定要用一个div（或者其他元素）包裹，否则会报错！！！*

## 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：

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

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**.

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

## 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：

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

## 注释

注释需要写在花括号中，实例如下：

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

## 数组

JSX 允许在模板中插入数组，数组会自动展开所有成员：

**var** arr = [  
 <h1>小张</h1>,  
 <h2>小李</h2>,  
];  
***ReactDOM***.**render**(  
<div>{***arr***}</div>,  
 **document**.getElementById(**'example'**)  
);

## 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：

**var** HelloMessage = ***React***.**createClass**({  
 render: **function**() {  
 **return** <h1>Hello World！</h1>;  
 }  
});  
  
***ReactDOM***.**render**(  
<***HelloMessage*** />,  
 **document**.getElementById(**'example'**)  
);

## 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：

**var** WebSite = ***React***.**createClass**({  
 render: **function**() {  
 **return** (  
 <div>  
 <***Name*** />  
 <***Link*** />  
 </div>  
 );  
 }  
});  
  
**var** Name = ***React***.**createClass**({  
 render: **function**() {  
 **return** (  
 <h1>百度</h1>  
 );  
 }  
});  
  
**var** Link = ***React***.**createClass**({  
 render: **function**() {  
 **return** (  
 <**a href**=**'http://www.baidu.com/'**>官方网址</**a**>  
 );  
 }  
});  
  
***ReactDOM***.**render**(  
<***WebSite*** />,  
 **document**.getElementById(**'example'**)  
);

# 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）。

以下实例中创建了 **LikeButton** 组件，getInitialState 方法用于定义初始状态，也就是一个对象，这个对象可以通过 this.state 属性读取。当用户点击组件，导致状态变化，this.setState 方法就修改状态值，每次修改以后，自动调用 this.render 方法，再次渲染组件。

**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** ? **'喜欢'** : **'不喜欢'**;  
 **return** (  
 <**p onClick**={**this**.handleClick}>  
 你<**b**>{text}</**b**>我。点我切换状态。  
 </**p**>  
 );  
 }  
});  
  
***ReactDOM***.**render**(  
<***LikeButton*** />,  
 **document**.getElementById(**'example'**)  
);

# 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：

**var** HelloMessage = ***React***.**createClass**({  
 render: **function**() {  
 **return** <h1>Hello {**this**.**props**.**name**}</h1>;  
 }  
});  
  
***ReactDOM***.**render**(  
<***HelloMessage* name**=**"尚学堂"** />,  
 **document**.getElementById(**'example'**)  
);

实例中 name 属性通过 this.props.name 来获取。

## 默认 Props

你可以通过 getDefaultProps() 方法为 props 设置默认值，实例如下：

**var** HelloMessage = ***React***.**createClass**({  
 getDefaultProps: **function**() {  
 **return** {  
 **name**: **'尚学堂'** };  
 },  
 render: **function**() {  
 **return** <h1>Hello {**this**.**props**.**name**}</h1>;  
 }  
});  
  
***ReactDOM***.**render**(  
<***HelloMessage*** />,  
 **document**.getElementById(**'example'**)  
);

## 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。

**var** WebSite = ***React***.**createClass**({  
 getInitialState: **function**() {  
 **return** {  
 **name**: **"百度"**,  
 **site**: **"http://www.baidu.com"** };  
 },  
  
 render: **function**() {  
 **return** (  
 <div>  
 <***Name* name**={**this**.state.name} />  
 <***Link* site**={**this**.state.site} />  
 </div>  
 );  
 }  
});  
  
**var** Name = ***React***.**createClass**({  
 render: **function**() {  
 **return** (  
 <h1>{**this**.props.name}</h1>  
 );  
 }  
});  
  
**var** Link = ***React***.**createClass**({  
 render: **function**() {  
 **return** (  
 <**a href**={**this**.props.site}>  
 {**this**.**props**.**site**}  
 </**a**>  
 );  
 }  
});  
  
***ReactDOM***.render(  
<***WebSite*** />,  
 **document**.getElementById(**'example'**)  
);

# 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**

**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(**'example'**)  
);

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

## 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。

# 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(防止异部操作阻塞UI)。
* **componentWillReceiveProps** 在组件接收到一个新的prop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 中移除的时候立刻被调用。

这些方法的详细说明，可以参考[官方文档](http://facebook.github.io/react/docs/component-specs.html" \l "lifecycle-methods" \t "http://www.runoob.com/react/_blank)。

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：

**var** Hello = ***React***.**createClass**({  
 getInitialState: **function** () {  
 **return** {  
 **opacity**: 1.0  
 };  
 },  
  
 componentDidMount: **function** () {  
 **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: **function** () {  
 **return** (  
 <div **style**={{**opacity**: **this**.**state**.**opacity**}}>  
 ***Hello*** {**this**.**props**.**name**}  
 </div>  
 );  
 }  
});  
  
***ReactDOM***.render(  
<***Hello* name**=**"world"**/>,  
 **document**.getElementById(**'example'**)  
);

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。

**var** Button = ***React***.**createClass**({  
 getInitialState: **function**() {  
 **return** {  
 **data**:0  
 };  
 },  
 setNewNumber: **function**() {  
 **this**.setState({**data**: **this**.**state**.**data** + 1})  
 },  
 render: **function** () {  
 **return** (  
 <div>  
 <button **onClick** = {**this**.setNewNumber}>增加</button>  
 <***Content* myNumber** = {**this**.state.data}></***Content***>  
 </div>  
 );  
 }  
})  
**var** Content = ***React***.**createClass**({  
 componentWillMount:**function**() {  
 **console**.log(**'Component WILL MOUNT!'**)  
 },  
 componentDidMount:**function**() {  
 **console**.log(**'Component DID MOUNT!'**)  
 },  
 componentWillReceiveProps:**function**(newProps) {  
 **console**.log(**'Component WILL RECEIVE PROPS!'**)  
 },  
 shouldComponentUpdate:**function**(newProps, newState) {  
 **return true**;  
 },  
 componentWillUpdate:**function**(nextProps, nextState) {  
 **console**.log(**'Component WILL UPDATE!'**);  
 },  
 componentDidUpdate:**function**(prevProps, prevState) {  
 **console**.log(**'Component DID UPDATE!'**)  
 },  
 componentWillUnmount:**function**() {  
 **console**.log(**'Component WILL UNMOUNT!'**)  
 },  
  
 render: **function** () {  
 **return** (  
 <div>  
 <h3>{**this**.props.myNumber}</h3>  
 </div>  
 );  
 }  
});  
***ReactDOM***.render(  
<div>  
<***Button*** />  
</div>,  
 **document**.getElementById(**'example'**)  
);

# 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库可以将数据存储在 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:

**var** UserGist = ***React***.**createClass**({  
 getInitialState: **function**() {  
 **return** {  
 **username**: **''**,  
 **lastGistUrl**: **''** };  
 },  
  
 componentDidMount: **function**() {  
 **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: **function**() {  
 **this**.**serverRequest**.abort();  
 },  
  
 render: **function**() {  
 **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=**"http://api.github.com/users/octocat/gists"** />,  
 document.getElementById(**'example'**)  
);

# React 表单与事件

本章节我们将讨论如何在 React 中使用表单。

### **实例1**

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。

**var** HelloMessage = ***React***.**createClass**({  
 getInitialState: **function**() {  
 **return** {**value**: **'Hello 尚学堂!'**};  
 },  
 handleChange: **function**(event) {  
 **this**.setState({**value**: event.**target**.**value**});  
 },  
 render: **function**() {  
 **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'**)  
);

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

### **实例2**

在以下实例中我么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。

**var** Content = ***React***.**createClass**({  
 render: **function**() {  
 **return** <div>  
 <**input type**=**"text" value**={**this**.props.myDataProp} **onChange**={**this**.props.updateStateProp} />   
 <h4>{**this**.props.myDataProp}</h4>  
 </div>;  
 }  
});  
**var** HelloMessage = ***React***.**createClass**({  
 getInitialState: **function**() {  
 **return** {**value**: **'Hello 尚学堂!'**};  
 },  
 handleChange: **function**(event) {  
 **this**.setState({**value**: event.**target**.**value**});  
 },  
 render: **function**() {  
 **var** value = **this**.**state**.**value**;  
 **return** <div>  
 <***Content* myDataProp** = {value}   
 **updateStateProp** = {**this**.handleChange}></***Content***>  
 </div>;  
 }  
});  
***ReactDOM***.**render**(  
 <***HelloMessage*** />,  
 **document**.getElementById(**'example'**)  
);

## React 事件

以下实例演示通过 onClick 事件来修改数据：

**var** HelloMessage = ***React***.**createClass**({  
 getInitialState: **function**() {  
 **return** {**value**: **'Hello 尚学堂!'**};  
 },  
 handleChange: **function**(event) {  
 **this**.setState({**value**: **'欢迎你！'**})  
 },  
 render: **function**() {  
 **var** value = **this**.**state**.**value**;  
 **return** <div>  
 <button **onClick**={**this**.handleChange}>点我有惊喜</button>  
 <h4>{value}</h4>  
 </div>;  
 }  
});  
***ReactDOM***.render(  
<***HelloMessage*** />,  
 **document**.getElementById(**'example'**)  
);

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：

**var** Content = ***React***.**createClass**({  
 render: **function**() {  
 **return** <div>  
 <button **onClick** = {**this**.props.updateStateProp}>点我改变父级state</button>  
 <h4>{**this**.props.myDataProp}</h4>  
 </div>  
 }  
});  
**var** HelloMessage = ***React***.**createClass**({  
 getInitialState: **function**() {  
 **return** {**value**: **'state：父级初始状态!'**};  
 },  
 handleChange: **function**(event) {  
 **this**.setState({**value**: **'state：父级改变状态！'**})  
 },  
 render: **function**() {  
 **var** value = **this**.**state**.**value**;  
 **return** <div>  
 <***Content* myDataProp** = {value}  
 **updateStateProp** = {**this**.handleChange}></***Content***>  
 </div>;  
 }  
});  
***ReactDOM***.render(  
<***HelloMessage*** />,  
 **document**.getElementById(**'example'**)  
);

# 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：

**var** MyComponent = ***React***.**createClass**({  
 handleClick: **function**() {  
 *// 使用原生的 DOM API 获取焦点* **this**.**refs**.myInput.focus();  
 },  
 render: **function**() {  
 *// 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}  
 />  
 </div>  
 );  
 }  
});  
  
***ReactDOM***.render(  
<***MyComponent*** />,  
 **document**.getElementById(**'example'**)  
);

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。