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。

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>hello world</title>  <script src="./build/react.min.js"></script>  <script src="./build/react-dom.min.js"></script>  <script src="./build/browser.min.js"></script>  </head>  <body>  <div id="container"></div>  <script type="text/babel">  ReactDOM.render(  <div>  <h1>hello, world!</h1>  <h2>welcome study</h2>  {/\* 注释:添加自定义属性需要使用 data- 前缀 \*/}  <p data-myattribute= "somevalue">JavaScript</p>  </div>,  document.getElementById('container')  );  </script>  </body>  </html> |

**JavaScript 表达式**

可在 JSX 中使用 JavaScript 表达式。写在花括号 **{}** 中。实例如下：

|  |
| --- |
| ReactDOM.render(  <div>  <h1>{1+1}</h1>  </div>,  document.getElementById('container')  ); |

JSX 中不能使用 if else 语句，但可用三元运算表达式来替代。

以下实例中如果变量 i = 1, 浏览器将输出 true, 如果修改 i 的值，则会输出 false.

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

**样式**

React 推荐使用内联样式。可使用 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 。

|  |
| --- |
| var mystyle = {  fontSize : 100,  color : "#FF0000"  };  ReactDOM.render(  <div>  <h1 style={mystyle}>菜鸟教程</h1>  </div>,  document.getElementById('container')  ); |

**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>react教程</h1>,  <h2>学的不仅是技术，更是梦想！</h2>,  ];  ReactDOM.render(  <div>{arr}</div>,  document.getElementById('container')  ); |

HTML 标签 vs. React 组件

React 可渲染 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('container')  ); |

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('container')  ); |

# React 组件

使用组件使应用更容易来管理。

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('container')  ); |

实例解析：

React.createClass 方法用于生成一个组件类 HelloMessage。

<HelloMessage /> 实例组件类并输出信息。

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。

还需要注意组件类只能包含一个顶层标签，否则也会报错。

如需要向组件传递参数，可使用 this.props 对象,实例如下：

|  |
| --- |
| var HelloMessage = React.createClass({  render : function() {  return <h1>Hello {this.props.name}</h1>;  }  });  ReactDOM.render(  <HelloMessage name="Runoob"/>,  document.getElementById('container')  ); |

结果: Hello Runoob

以上实例中 name 属性通过 this.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 ，因为 class 和 for 是 JavaScript 的保留字。

**复合组件**

可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。

输出网站名字和网址的组件：

|  |
| --- |
| var Website = React.createClass({  render: function() {  return (  <div>  <Name name= {this.props.name} />  <Link site= {this.props.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 name="菜鸟教程" site="http://www.runoob.com" />,  document.getElementById('container')  ); |

# 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() {  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('container')  ); |

# React Props

state 和 props 主要区别在于 props 不可变，而 state 可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="Runoob" />,  document.getElementById('container')  ); |

**默认 Props**

可通过 getDefaultProps() 为 props 设置默认值：

|  |
| --- |
| var HelloMessage = React.createClass({  getDefaultProps : function() {  return {  name : 'Runoob'  };  },  render: function() {  return <h1>Hello {this.props.name}</h1>;  }  });    ReactDOM.render(  <HelloMessage/>,  document.getElementById('container')  ); |

**State 和 Props**

以下实例演示了如何在应用中组合使用 state 和 props 。

可在父组件中设置 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, 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: "菜鸟教程",  site: "http://www.runoob.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')  ); |

**Props 验证**

使用 propTypes，可保证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 ：

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

注:假如title值不为string, 将只显示 Hello

更多验证器:

|  |
| --- |
| React.createClass({  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 组件 API

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，要设置的新状态，会和当前state合并

callback，可选，回调函数。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。

|  |
| --- |
| 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('container')  ); |

**替换状态：replaceState**

replaceState(object nextState[, function callback])

nextState，要设置的新状态，会替换当前的state。

callback，可选，回调函数。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，要设置的新属性，该状态会和当前的props合并

callback，可选，回调函数。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，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，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，可在节点上再次调用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，回调函数。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()。但组件重新渲染时，依然会读取this.props和this.state，如状态没改变，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()

一般应尽量避免用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，this.findDOMNode()也会返回null。

从DOM 中读取值时，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

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：

* componentWillMount 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，可通过this.getDOMNode()来进行访问。 如想和其他JavaScript框架一起使用，可在这个方法中调用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。在初始化render时不会被调用。
* shouldComponentUpdate 返回布尔值。在组件接收到新的props或者state时被调用。在初始化时或使用forceUpdate时不被调用。可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。

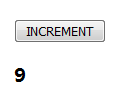
以下实例在 Hello 组件加载以后，通过 componentDidMount 方法设置一个定时器，每隔100毫秒重新设置组件的透明度，并重新渲染：

|  |
| --- |
| 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('container')  ); |

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  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}>INCREMENT</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')  ); |

结果：点一下按钮数字增加1



# React AJAX

React 组件的数据可通过 componentDidMount 方法中的 Ajax 来获取，当从服务端获取数据库可将数据存储在 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="https://api.github.com/users/octocat/gists" />,  mountNode  ); |

# React 表单与事件

一个简单的实例：设置了输入框 input 值value = {this.state.data}。

在输入框值变化时，可更新 state。可使用 onChange 事件来监听 input 的变化，并修改 state。

|  |
| --- |
| var HelloMessage = React.createClass({  getInitialState: function() {  return {value: 'Hello Runoob!'};  },  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 Runoob! 的 input 元素，并通过 onChange 事件响应更新用户输入的值。

**实例 2**

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。

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 Runoob!'};  },  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**

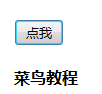
通过 onClick 事件来修改数据：

|  |
| --- |
| var HelloMessage = React.createClass({  getInitialState: function() {  return {value: 'Hello Runoob!'};  },  handleChange: function(event) {  this.setState({value: '菜鸟教程'})  },  render: function() {  var value = this.state.value;  return <div>  <button onClick={this.handleChange}>点我</button>  <h4>{value}</h4>  </div>;  }  });  ReactDOM.render(  <HelloMessage />,  document.getElementById('example')  ); |

原本：



点击按钮后:



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</button>  <h4>{this.props.myDataProp}</h4>  </div>  }  });  var HelloMessage = React.createClass({  getInitialState: function() {  return {value: 'Hello Runoob!'};  },  handleChange: function(event) {  this.setState({value: '菜鸟教程'})  },  render: function() {  var value = this.state.value;  return <div>  <Content myDataProp = {value}  updateStateProp = {this.handleChange}></Content>  </div>;  }  });  ReactDOM.render(  <HelloMessage />,  document.getElementById('example')  ); |

原本：



点击之后：



# React Refs

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

允许引用 render() 返回的相应的支撑实例（ backing instance ）。

就可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

可通过 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')  ); |

原来：



点击按钮后：



实例中，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。

也可使用 getDOMNode()方法获取DOM元素