建议学习路线：

React -> npm -> javascript打包工具 -> ES6 -> Routing(react-router) -> Flux(Redux)

1.JSX语法编译为js语法，第一步：npm install -g react-tools,第二部：进入文件目录，jsx --watch src/ build/(将src目录下的文件转译并输出到build目录下)。

注：编译后无需加载JSXTransformer等js文件，引入的外部文件无需使用type="text/babel"的格式。

## 2. State 工作原理：常用的通知 React 数据变化的方法是调用 setState(data, callback)。这个方法会合并（merge） data 到 this.state，并重新渲染组件。渲染完成后，调用可选的 callback 回调。大部分情况下不需要提供 callback，因为 React 会负责把界面更新到最新状态。

## 3. 哪些 应该 作为 State？ State 应该包括那些可能被组件的事件处理器改变并触发用户界面更新的数据。真实的应用中这种数据一般都很小且能被 JSON 序列化。当创建一个状态化的组件时，想象一下表示它的状态最少需要哪些数据，并只把这些数据存入 this.state。在render() 里再根据 state 来计算你需要的其它数据。你会发现以这种方式思考和开发程序最终往往是正确的，因为如果在 state 里添加冗余数据或计算所得数据，需要你经常手动保持数据同步，不能让 React 来帮你处理。

4. 如果子组件位置会改变（如在搜索结果中）或者有新组件添加到列表开头（如在流中）情况会变得更加复杂。如果子级要在多个渲染阶段保持自己的特征和状态，在这种情况下，你可以通过给子级设置惟一标识的 key 来区分。

render: function() {

var results = this.props.results;

return (

<ol>

{results.map(function(result) {

return <li key={result.id}>{result.text}</li>;

})}

</ol>

);

}

5. 当 React 校正带有 key 的子级时，它会确保它们被重新排序（而不是破坏）或者删除（而不是重用）。 务必 把 key 添加到子级数组里组件本身上，而不是每个子级内部最外层 HTML 上：

// 错误！

var ListItemWrapper = React.createClass({

render: function() {

return <li key={this.props.data.id}>{this.props.data.text}</li>;

}

});

var MyComponent = React.createClass({

render: function() {

return (

<ul>

{this.props.results.map(function(result) {

return <ListItemWrapper data={result}/>;

})}

</ul>

);

}

});

// 正确 :)

var ListItemWrapper = React.createClass({

render: function() {

return <li>{this.props.data.text}</li>;

}

});

var MyComponent = React.createClass({

render: function() {

return (

<ul>

{this.props.results.map(function(result) {

return <ListItemWrapper key={result.id} data={result}/>;

})}

</ul>

);

}

});

## 6. 数据流:React 里，数据通过上面介绍过的 props 从拥有者流向归属者。这就是高效的单向数据绑定(one-way data binding)：拥有者通过它的 props 或 state 计算出一些值，并把这些值绑定到它们拥有的组件的 props 上。因为这个过程会递归地调用，所以数据变化会自动在所有被使用的地方反映出来。

7.验证prop格式，不符合则报错

React.createClass({

propTypes: {

// 可以声明 prop 为指定的 JS 基本类型。默认

// 情况下，这些 prop 都是可传可不传的。

optionalArray: React.PropTypes.array,

optionalBool: React.PropTypes.bool,

optionalFunc: React.PropTypes.func,

optionalNumber: React.PropTypes.number,

optionalObject: React.PropTypes.object,

optionalString: React.PropTypes.string,

// 所有可以被渲染的对象：数字，

// 字符串，DOM 元素或包含这些类型的数组。

optionalNode: React.PropTypes.node,

// React 元素

optionalElement: React.PropTypes.element,

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

optionalMessage: React.PropTypes.instanceOf(Message),

// 用 enum 来限制 prop 只接受指定的值。

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

// 指定的多个对象类型中的一个

optionalUnion: React.PropTypes.oneOfType([

React.PropTypes.string,

React.PropTypes.number,

React.PropTypes.instanceOf(Message)

]),

// 指定类型组成的数组

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

// 指定类型的属性构成的对象

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

// 特定形状参数的对象

optionalObjectWithShape: React.PropTypes.shape({

color: React.PropTypes.string,

fontSize: React.PropTypes.number

}),

// 以后任意类型加上 `isRequired` 来使 prop 不可空。

requiredFunc: React.PropTypes.func.isRequired,

// 不可空的任意类型

requiredAny: React.PropTypes.any.isRequired,

// 自定义验证器。如果验证失败需要返回一个 Error 对象。不要直接

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

customProp: function(props, propName, componentName) {

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

return new Error('Validation failed!');

}

}

},

/\* ... \*/

});

8. React 支持以声明式的方式来定义 props 的默认值。

var ComponentWithDefaultProps = React.createClass({

getDefaultProps: function() {

return {

value: 'default value'

};

}

/\* ... \*/

});

当父级没有传入 props 时，getDefaultProps() 可以保证 this.props.value 有默认值，注意 getDefaultProps 的结果会被 缓存。得益于此，你可以直接使用 props，而不必写手动编写一些重复或无意义的代码。

## 小技巧：

var CheckLink = React.createClass({

render: function() {

// 这样会把 CheckLink 所有的 props 复制到 <a>

return <a {...this.props}>{'√ '}{this.props.children}</a>;

}

});

React.render(

<CheckLink href="/checked.html">

Click here!

</CheckLink>,

document.getElementById('example')

);

**9.mixin**组件是 React 里复用代码最佳方式，但是有时一些复杂的组件间也需要共用一些功能。有时会被称为 [跨切面关注点](http://en.wikipedia.org/wiki/Cross-cutting_concern)。React 使用 mixins 来解决这类问题。

一个通用的场景是：一个组件需要定期更新。用 setInterval() 做很容易，但当不需要它的时候取消定时器来节省内存是非常重要的。React 提供 [生命周期方法](https://github.com/reactjs-cn/react-docs/blob/master/react/docs/working-with-the-browser.html#component-lifecycle) 来告知组件创建或销毁的时间。下面来做一个简单的 mixin，使用 setInterval() 并保证在组件销毁时清理定时器。

var SetIntervalMixin = {

componentWillMount: function() {

this.intervals = [];

},

setInterval: function() {

this.intervals.push(setInterval.apply(null, arguments));

},

componentWillUnmount: function() {

this.intervals.map(clearInterval);

}

};

var TickTock = React.createClass({

mixins: [SetIntervalMixin], // 引用 mixin

getInitialState: function() {

return {seconds: 0};

},

componentDidMount: function() {

this.setInterval(this.tick, 1000); // 调用 mixin 的方法

},

tick: function() {

this.setState({seconds: this.state.seconds + 1});

},

render: function() {

return (

<p>

React has been running for {this.state.seconds} seconds.

</p>

);

}

});

React.render(

<TickTock />,

document.getElementById('example')

);

## 10.交互组件

## （1）受限组件

## 设置了 value 的 <input> 是一个*受限*组件。 对于受限的 <input>，渲染出来的 HTML 元素始终保持 value 属性的值。例如：

render: function() {

return <input type="text" value="Hello!" />;

}

上面的代码将渲染出一个值为 Hello! 的 input 元素。用户在渲染出来的元素里输入任何值都不起作用，因为 React 已经赋值为 Hello!。如果想响应更新用户输入的值，就得使用 onChange 事件：

getInitialState: function() {

return {value: 'Hello!'};

},

handleChange: function(event) {

this.setState({value: event.target.value});

},

render: function() {

var value = this.state.value;

return <input type="text" value={value} onChange={this.handleChange} />;

}

上面的代码中，React 将用户输入的值更新到 <input> 组件的 value 属性。这样实现响应或者验证用户输入的界面就很容易了。例如：

handleChange: function(event) {

this.setState({value: event.target.value.substr(0, 140)});

}

上面的代码接受用户输入，并截取前 140 个字符。

## （2）不受限组件

没有设置 value(或者设为 null) 的 <input> 组件是一个不受限组件。对于不受限的 <input> 组件，渲染出来的元素直接反应用户输入。例如：

render: function() {

return <input type="text" />;

}

上面的代码将渲染出一个空值的输入框，用户输入将立即反应到元素上。和受限元素一样，使用 onChange 事件可以监听值的变化。

如果想给组件设置一个非空的初始值，可以使用 defaultValue 属性。例如：

render: function() {

return <input type="text" defaultValue="Hello!" />;

}

上面的代码渲染出来的元素和**受限组件**一样有一个初始值，但这个值用户可以改变并会反应到界面上。

同样地， 类型为 radio、checkbox 的<input> 支持 defaultChecked 属性， <select> 支持 defaultValue 属性。

render: function() {

return (

<div>

<input type="radio" name="opt" defaultChecked /> Option 1

<input type="radio" name="opt" /> Option 2

<select defaultValue="C">

<option value="A">Apple</option>

<option value="B">Banana</option>

<option value="C">Cranberry</option>

</select>

</div>

);

}

## （3）textarea

## 在 HTML 中， <textarea> 的值通常使用子节点设置：

<!-- 反例：在 React 中不要这样使用！ -->

<textarea name="description">This is the description.</textarea>

对 HTML 而言，让开发者设置多行的值很容易。但是，React 是 JavaScript，没有字符限制，可以使用 \n 实现换行。简言之，React 已经有 value、defaultValue 属性，</textarea> 组件的子节点扮演什么角色就有点模棱两可了。基于此， 设置<textarea> 值时不应该使用子节点：

<textarea name="description" value="This is a description." />

如果非要\*使用子节点，效果和使用 defaultValue 一样。

## (4)select

HTML 中 <select> 通常使用 <option> 的 selected 属性设置选中状态；React 为了更方便的控制组件，采用以下方式代替：

<select value="B">

<option value="A">Apple</option>

<option value="B">Banana</option>

<option value="C">Cranberry</option>

</select>

如果是不受限组件，则使用 defaultValue。

注意：给 value 属性传递一个数组，可以选中多个选项：<select multiple={true} value={['B', 'C']}>。