# Agenda

## Overview

## Architecture

## Components

## Handling Event

## Forms

## Performance

**Homework**

# Overview

简而言之，React就是一个JavaScript的一个library / framework。

Example：

|  |
| --- |
| var Hello = React.createClass({  render: function(){  return <div>  <h1> Hello at {this.props.now}</h1>  </div>;  }  });  ReactDOM.render(<Hello now={new Date().toString()} />,  document.getElementById("root")); |

### Tools

CodePen：

可以在线玩玩React，使用CodePen：

<http://codepen.io/gaearon/pen/rrpgNB?editors=1010>

Babel：编译ES6用的，可以直接嵌入在网页：

|  |
| --- |
| <script src="https://unpkg.com/react@15/dist/react.min.js"></script> |

React和React-DOM package：

Develop版本

|  |
| --- |
| <script src="https://unpkg.com/react@15/dist/react.js"></script> <script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script> |

或者使用Product版本：

|  |
| --- |
| <script src="https://unpkg.com/react@15/dist/react.min.js"></script> <script src="https://unpkg.com/react-dom@15/dist/react-dom.min.js"></script> |

插件：

Chrome：<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en>

Firefox：

<https://addons.mozilla.org/en-US/firefox/addon/react-devtools/>

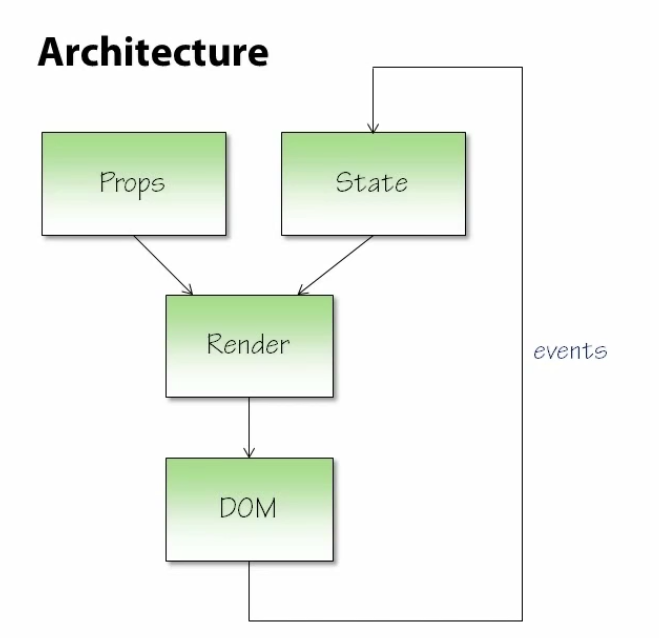
**Template**

该文档之后所有的example都是代码片段，这些代码片段放在以下模板中都是可运行的：

|  |
| --- |
| <!DOCTYPE html> <html>   <head>     <meta **charset="UTF-8"** />     <title>Template</title>     <script **src="./lib/react.js"**></script>     <script **src="./lib/react-dom.js"**></script>     <script **src="./lib/babel.min.js"**></script>   </head>   <body>     <div **id="root"**></div>     <script **type="text/babel"**> {Place code snippet here}     </script>   </body> </html> |

这里需要将对应的React.js, react-dom.js和Babel.min.js下载到对应的lib文件夹下。

# Architecture



一个React程序就是Component的集合，通过Render将Component转成对应的DOM element。因为DOM是树状，所以对应的Component也可以是树状，数据从上到下可以通过Props传递，Component也可以维护自己的状态，通过State。

**Components**

### Introducing JSX

JSX 就是一种JavaScript的语法扩展，例如：

|  |
| --- |
| **const** element = <h1>Hello World!</h1> |

在[Babel](https://babeljs.io/repl/#?babili=false&evaluate=true&lineWrap=false&presets=es2015%2Creact%2Cstage-2&targets=&browsers=&builtIns=false&code=%3CHello%20%2F%3E)官方网页上可以看到这段代码转成的真正JavaScript：

|  |
| --- |
| var element = React.createElement(  "h1",  null,  "Hello World!"  ); |

上面用到的函数签名就是React.createElement(component, props, ...children)。

使用JSX可以使得编写的代码更易读。

JSX大致要点可以总结为：

1. 可以内嵌表达式
2. JSX本身也是表达式
3. 支持Attributes
4. 支持子节点
5. ReactDOM会Escape JSX中的值
6. JSX中的comments需要用{/\*\*/}， 例如{/\*Comments\*/}

例如：

|  |
| --- |
| 内嵌表达式，Attribute (BasicExample.html) |
| var Hello = React.createClass({  render: function(){  return <div>  <h1> Hello at {this.props.now}</h1>  </div>;  }  });  ReactDOM.render(<Hello now={new Date().toString()} />,  document.getElementById("root")); |

|  |
| --- |
| Non-JSX (Non-Jsx.html) |
| var Hello = React.createClass({  displayName: "Hello",  render: function render() {  return React.createElement(  "div",  null,  React.createElement(  "h1",  null,  " Hello at ",  this.props.now  )  );  }  });  ReactDOM.render(React.createElement(Hello, { now: new Date().toString() }), document.getElementById("root")); |

|  |
| --- |
| 子节点，作为表达式(NestedElement.html) |
| class Inner extends React.Component{  render(){  return (  <p>inner text: {this.props.subTitle}</p>  );  };  }    var Outter = React.createClass({  render: function(){  return <div>  <h1> Hello at {this.props.now}  <br/>  <Inner subTitle="First Title" />  {this.props.innerTitles.map((t, i) => {  return <Inner subTitle={t} key={i} />  })}  </h1>  </div>;  }  });    ReactDOM.render(  <Outter now={new Date().toString()}  innerTitles = {["Title1", "Title2", "Title3"]} />,  document.getElementById("root")); |

|  |
| --- |
| Escape (EscapeProp.html) |
| var DangerContainer = React.createClass({  render: function () {  return <p dangerouslySetInnerHTML={{\_\_html: this.props.dangerous}}></p>;  // return <p>{this.props.dangerous}</p>;  }  });  ReactDOM.render(<DangerContainer dangerous="<strong>HELLO</strong>" />, document.getElementById("root")); |

### Rendering Elements

这里的Element就是指上文提到的React.createElement，element是React App的最小组成单位，Component也是element组成的，ReactDOM也是根据React Element去更新对应DOM中的element。React Element是可不更改的，创建Element非常廉价。

|  |
| --- |
| Rendering Element To DOM (RenderingElement.html) |
| var hello = <h1> Hello world!</h1>;  ReactDOM.render(hello,  document.getElementById("root")); |

### Component, Props, Children and State

Component构建React App的基础，一个Component对应DOM中的一个element(可以通过findDOMNote(component)函数找到对应的DOM element)，利用Component可以渲染DOM element中的内容，同时也能处理DOM element中的事件。Components之间可以相互组合和嵌套（不能连成环），对应的也就有组合和嵌套的DOM element。

|  |
| --- |
| Functional Component (FunctionComponent.html) |
| function Hello(props) {  return (<div>  <h1> Hello at {props.now}</h1>  </div>);  }  ReactDOM.render(<Hello now={new Date().toString()} />,  document.getElementById("root")); |

|  |
| --- |
| Class Component (BasicExample1.html) |
| class Hello extends React.Component{  render(){  return (<div>  <h1>Hello at {this.props.now}.</h1>  </div>);  }  }  ReactDOM.render(<Hello now={new Date().toString()} />,  document.getElementById("root")); |

|  |
| --- |
| Nested Element |
| class Inner extends React.Component{  render(){  return (  <p>inner text: {this.props.subTitle}</p>  );  };  }    class Outter extends React.Component {  render(){  return <div>  <h1> Hello at {this.props.now}  <br/>  <Inner subTitle="First Title" />  {this.props.innerTitles.map((t, i) => {  return <Inner subTitle={t} key={i} />  })}  </h1>  </div>;  }  }    ReactDOM.render(  <Outter now={new Date().toString()}  innerTitles = {["Title1", "Title2", "Title3"]} />,  document.getElementById("root")); |

|  |
| --- |
| Children Component (JSXChildExpression.html) |
| var First = React.createClass({  render: function () {  return <p>1st</p>;  }  });  var Second = React.createClass({  render: function () {  return <p>2nd</p>;  }  });  var Appender = React.createClass({  render: function () {  return <div className="appender">{this.props.children}</div>;  }  });  ReactDOM.render(  <Appender>  <First />  <Second />  </Appender>, document.getElementById("root")); |

|  |
| --- |
| State (Status.html) |
| /\* class Hello extends React.Component{  constructor(){  super();  this.state = {  now: 1,  };  }    changeStatus(){  this.setState({now: this.state.now + 1});  }    render(){  return (  <div>  <div onClick={() => this.changeStatus()}> ChangeStatus</div>  <h1>Hello at {this.state.now}</h1>  </div>);  }  } \*/    var Hello = React.createClass({  getInitialState: function(){  return {now: 1};  },    changeStatus: function(){  this.setState({now: this.state.now + 1});  },    render: function(){  return (  <div>  <div onClick={() => this.changeStatus()}> ChangeStatus</div>  <h1>Hello at {this.state.now}</h1>  </div>);  }  });    ReactDOM.render(<Hello />,  document.getElementById("root")); |

|  |
| --- |
| Typechecking with PropTypes (Props\_validating.html) |
| /\*  var Child = React.createClass({  propTypes: {  now: function(props, propName, componentName){  if (props[propName] > 5){  return new Error("now property cannot be greater than 5.");  }  },    text: React.PropTypes.string.isRequired,  },    render: function(){  return (  <div>  <h1>Hello {this.props.text}</h1>  <h2>at {this.props.now}</h2>  </div>);  }  }); \*/    class Child extends React.Component{    static propTypes = {  now: function(props, propName, componentName){  if (props[propName] > 5){  return new Error("now property cannot be greater than 5.");  }  },    text: React.PropTypes.string.isRequired,  }    render(){  return (  <div>  <h1>Hello {this.props.text}</h1>  <h2>at {this.props.now}</h2>  </div>);  }  }    class Parent extends React.Component{  constructor(){  super();  this.state = {  now: 1,  text: 'World!'  }  }    changeStatus() {  this.setState({now: this.state.now + 1});  }    render(){  return (  <div onClick={() => this.changeStatus()}> ChangeStatus  <Child now={this.state.now} text={this.state.text} />  </div>    );  }  }    ReactDOM.render(<Parent />,  document.getElementById("root")); |

|  |
| --- |
| Choosing Type at Runtime (ChosingType.html) |
| class Linux extends React.Component {  render (){  return <div>start {this.props.value} in linux.</div>  }  }  class Windows extends React.Component {  render () {  return <div>start {this.props.value} in windows</div>  }  }  const operatorSystems = {  linux: Linux,  windows: Windows  };  const SpecifiedOperatorSystem = operatorSystems['linux'];  ReactDOM.render(<SpecifiedOperatorSystem value={'react'} />,  document.getElementById("root"));  /\* const specifiedOperatorSystem = operatorSystems['linux'];  ReactDOM.render(<specifiedOperatorSystem value={'react'} />,  document.getElementById("root")); \*/ |

|  |
| --- |
| Conditional Rendering (ConditionalRendering.html) |
| /\* class Mail extends React.Component {  render() {  return (  <div>  <h1> Hello you,</h1>    {this.props.unreadMessages.length > 0 &&  <h2>  You have {this.props.unreadMessages.length} unread message(s).  </h2>  }  </div>  );  }  } \*/  /\* class Mail extends React.Component {  render() {  if (this.props.unreadMessages.length > 0){  return (  <div>  <h1> Hello you,</h1>  <h2>  You have {this.props.unreadMessages.length} unread message(s).  </h2>  </div>  );  } else {  return (<div>  <h1> Hello you,</h1>  </div>);  }  }  } \*/    class Mail extends React.Component {  render() {  return (  <div>  <h1> Hello you,</h1>    {this.props.unreadMessages.length > 0 ?  (<h2>  You have {this.props.unreadMessages.length} unread message(s).  </h2>) : null  }  </div>  );  }  }    ReactDOM.render(<Mail unreadMessages={["mail1", "mail2"]} />,  document.getElementById("root")); |

要点：

1. Component的name都是首字母大写，这样便于区分DOM的tag和自定义的tag，例如<div />代表DOM tag，<Hello />代表一个Component
2. setState的效果为：



1. 不要直接更改state，例如

|  |
| --- |
| // Wrong  this.state.now = 1;  //Correct  this.setState({now: 1}); |

1. setState不要依赖当前的state和props，例如：

|  |
| --- |
| // Wrong  this.setState({now: this.state.now + this.props.increment}) |

因为React可能会因为性能的原因，将多个setState语句放到一起。可以使用setState的另一个结构代替：

|  |
| --- |
| // Correct  this.setState((prevState, props) => {  now: prevState.now + props.increment,  }); |

1. State 的赋值只能在构造函数中或者getInitialState函数中
2. State只在Component中可见，对其他的Component不可见，当然可以通过props传值的方式将State赋值给嵌入的component。所以这种数据流都是从上到下的方式
3. propTypes只会在develop模式下生效
4. Boolean, Null, Undefined会被ignore，以下的标签是一样的效果

|  |
| --- |
| <div />  <div>{true}</div>  <div>{undefined}<div> |

1. 可以在render函数中返回null，这样就不会render到DOM中
2. List 需要key

**LifeCycle**

**Mounting**

constructor()

componentWillMount()

render()

componentDidMount()

**Updating**

componentWillReceiveProps()：指收到新的props

shouldComponentUpdate()

componentWillUpdate()

render()

componentDidUpdate()

**UnMounting**

componentWillUnmount()

|  |
| --- |
| LifeCycle (Lifecycle.html) |
| class Clock extends React.Component{  constructor(){  super();  this.state = {date: new Date()};  }    componentDidMount(){  this.timerId = setInterval(  () => this.tick(),  1000  )  }    tick(){  this.setState({date: new Date()})  }    componentWillUnmount(){  clearInterval(this.timerId);  }    render(){  return (<div>  <h1>Hello at {this.state.date.toString()}.</h1>  </div>);  }  }  ReactDOM.render(<Clock />,  document.getElementById("root")); |

要点：

1. Functional component没有lifecycle

**Not recommended**

[Mixins](https://facebook.github.io/react/docs/react-without-es6.html#mixins)

[Legacy API: String Refs](https://facebook.github.io/react/docs/refs-and-the-dom.html#legacy-api-string-refs)

# Handling Event

**Differences**

React中handle event跟DOM中的handle event非常类似，区别如下：

1. React使用驼峰（camelCase）命名法，DOM中都是小写
2. JSX中传入的都是function，DOM传入的是string（也就是表达式）

例如：

|  |  |
| --- | --- |
| DOM HTML | React (HandleClick.html) |
| <button  onclick="handleClick()">  Click Me  </button> | <button  onClick="() => this.handleClick()">  Click Me  </button> |

1. React中通过调用preventDefault函数来重写默认的behavior，在DOM中通过return false：

|  |  |
| --- | --- |
| DOM HTML | React([event](https://www.w3.org/TR/uievents/#event-type-click), preventDefault.html) |
| <a href="#" target="\_blank"  onclick="console.log('clicked'); return false;">  click me  </a> | class ClickMe extends React.Component {  render(){  return (<a href="#" target="\_blank"  onClick={(e) => this.handleClick(e)}>  click me  </a>);  }  handleClick(e){  console.log('clicked');  e.preventDefault();  }  }  ReactDOM.render(<ClickMe />,  document.getElementById("root")); |

**Bind ‘this’**

在class中的函数，this并没有显式的绑定，所以在使用的时候，函数被调用的时候this有可能为null。有三种方式可以进行绑定：

第一种在preventDefault的demo中有看到。另外能在构造函数中绑定：

|  |
| --- |
| class ClickMe extends React.Component {  constructor (){  super();  this.handClick = this.handleClick.bind(this);  }  render(){  return (<a href="#" target="\_blank"  onClick={this.handleClick}>  click me  </a>);  }  handleClick(e){  console.log('clicked');  e.preventDefault();  }  }  ReactDOM.render(<ClickMe />,  document.getElementById("root")); |

剩下的一种是Experimental的语法，这种称为[property initializer syntax](https://babeljs.io/docs/plugins/transform-class-properties/)

|  |
| --- |
| class ClickMe extends React.Component {  render(){  return (<a href="#" target="\_blank"  onClick={this.handleClick}>  click me  </a>);  }  handleClick = (e) => {  console.log('clicked');  e.preventDefault();  }  }  ReactDOM.render(<ClickMe />,  document.getElementById("root")); |

**Ref**

通过ref attribute可以访问到父Component，这时候数据的流动是从下到上的。

ref attribute会接受一个callback函数，在component在挂载或者卸载的时候会被执行。

|  |
| --- |
| Ref (Refs.html) |
| class CustomTestInput extends React.Component {  constructor (){  super();  this.focus = this.focus.bind(this);  }  focus() {  this.textInput.focus();  }  render(){  return (<div>  <input type="text" ref={(input) => {this.textInput = input;}} />  <input type="text" />  <input type="button" value="Focus" onClick={this.focus} />  </div>);  }  }  ReactDOM.render(<CustomTestInput />,  document.getElementById("root")); |

要点：

1. Ref中的callback函数参数，在mount的时候传进去的是mount的component，在上面的例子中就是第一个input，在unmount的时候传进去的是null
2. 不将ref使用在functional components中，因为functional component没有lifecycle
3. this.refs 已经不推荐使用了
4. 不要滥用ref

**Forms**

**Controlled Components**

先来看一个DOM中的form：

|  |
| --- |
| <form>  <label>  Name:  <input type="text" name="name" />  </label>  <input type="submit" value="submit" />  </form> |

但是上面那些提交的表单信息都不归React控制，如果说由React中的Component render出来的Form，并且这个form是有component自主控制的，也就是通过state来管理用户的输入，那么这就是Controlled Component

|  |
| --- |
| Controlled Component |
| class ControlledForm extends React.Component {  constructor(){  super();  this.state = {  name: '',  };  this.handleChanges = this.handleChanges.bind(this);  this.handleSubmit = this.handleSubmit.bind(this);  }  handleSubmit(e){  console.log('submited: {name: ' + this.state.name + '}');  e.preventDefault();  }  handleChanges(e){  this.setState({  name: e.target.value  });  }  render(){  return (  <form onSubmit={this.handleSubmit}>  <label>  Name:  <input type="text" name="name" value={this.state.value} onChange={this.handleChanges} />  {/\* <input type="text" name="name" value="init" /> \*/}  </label>  <input type="submit" value="submit" />  </form>  );  }  handleClick = (e) => {  console.log('clicked');  e.preventDefault();  }  }  ReactDOM.render(<ControlledForm />,  document.getElementById("root")); |

要点：

1. 在DOM中textarea 是将内容放入\_text中，对应在React中是textarea的value属性：

|  |  |
| --- | --- |
| DOM | React |
| <textarea>  this is content.  </textarea> | var element = <textarea value='this is content.' />  ReactDOM.render(element,  document.getElementById("root")); |

1. 对于dropdownlist，DOM和React也有区别：

|  |  |
| --- | --- |
| DOM | React |
| <select>  <option value="one">1</option>  <option selected value="two">2</option>  <option value="three">3</option>  </select> | var element = (<select value='two'>  <option value="one">1</option>  <option value="two">2</option>  <option value="three">3</option>  </select>);  ReactDOM.render(element,  document.getElementById("root")); |

1. 一个小技巧，针对多个需要用户输入的fields，我们可以通过name或者其他的属性写一个handleInput函数就可以将所有的用户输入和对应的state关联起来，例如：

|  |
| --- |
| Handle Multiple Inputs |
| class ControlledForm extends React.Component {  constructor(){  super();  this.state = {  name: '',  value: '',  };  this.handleChanges = this.handleChanges.bind(this);  this.handleSubmit = this.handleSubmit.bind(this);  }  handleSubmit(e){  console.log('submited: {name: ' + this.state.name + ', value: ' + this.state.value + '}');  e.preventDefault();  }  handleChanges(e){  const name = e.target.name;  this.setState({  [name]: e.target.value  });  }  render(){  return (  <form onSubmit={this.handleSubmit}>  <label>  Name:  <input type="text" name="name" value={this.state.name} onChange={this.handleChanges} />  Value:  <input type="text" name="value" value={this.state.value} onChange={this.handleChanges} />  </label>  <input type="submit" value="submit" />  </form>  );  }  handleClick = (e) => {  console.log('clicked');  e.preventDefault();  }  }  ReactDOM.render(<ControlledForm />,  document.getElementById("root")); |

**UnControlled Components**

UnControlled Components就是指用户输入的fields并不是由React控制的，虽然React不通过state控制用户的输入，但是我们可以使用ref来得到form中的数据：

|  |
| --- |
| Uncontrolled Components |
| class UncontrolledForm extends React.Component {  constructor(){  super();  this.state = {  name: '',  };  this.handleSubmit = this.handleSubmit.bind(this);  }  handleSubmit(e){  console.log('submited: {name: ' + this.textInput.value + '}');  e.preventDefault();  }  render(){  return (  <form onSubmit={this.handleSubmit}>  <label>  Name:  {/\*<input type="text" value="init" ref={(input) => {this.textInput = input;}} />\*/}  <input type="text" defaultValue="init" ref={(input) => {this.textInput = input;}} />  </label>  <input type="submit" value="submit" />  </form>  );  }  handleClick = (e) => {  console.log('clicked');  e.preventDefault();  }  }  ReactDOM.render(<UncontrolledForm />,  document.getElementById("root")); |

# Performance

**Chome Timeline**

1. 打开Chrome加载网页的时候加上参数?react\_perf，例如：

<http://localhost:8080/Lifecycle.html?react_perf>

1. 打开开发人员工具，选择Timeline tab
2. 选择record（不要超过20s，不然容易hang死）
3. 停止record，这时候能看到一个UserTime的label，这里面React相关的事件信息

**Avoid Reconciliation**

当我们使用React render UI的时候，在React的内部有对应的“virtual DOM”进行管理，React就根据这些virtual DOM对必要的UI进行重新render，从而避免创建不必要的DOM Nodes。这些对应的对象可以认为是render函数生成的React Elements，可以认为render函数就是为了创建一个由React Elements组成的树形结构。当我们在更新props或者state的时候，那么render就返回一个不同的树。这时候React就需要考虑怎么有效的更新DOM中的element。

首先React有必要比较生成的tree的的不同，React使用的算法基于以下两点：

1. 两个不同类型的Elements会产生不同的树状结构
2. Children Elements可以由开发人员指定的identity

算法大致如下：

1. 比较render函数生成的树，首先会比较根节点
2. 如果节点的类型不一样，那么原来的React Element对应的DOM element会全部被回收掉，重新构建，这时候生命周期中的componentWillUnmount， componentWillMount和componentDidMount会被触发，如：

|  |  |
| --- | --- |
| <div>  <Counter />  </div> | <span>  <Counter />  </span> |

1. 如果类型一样，但是attribute不一样，那么只会更新对应DOM中对应的Attribute，其他相同的attribute不更新。回到Component，其state的更新在render方法维护，如果这时候更新的是子Component的props，那么React除了更新对应Component的props，还会调用对应实例的componentWillReceiveProps和componentWillUpdate函数，render()函数完成后，接下来会递归子节点
2. 如果遍历的子节点是一个DOM节点，会比较子节点，然后对于不同的地方进行更新，如：

|  |
| --- |
| <ul>  <li>first</li>  <li>second</li>  </ul>  <ul>  <li>first</li>  <li>second</li>  <li>third</li>  </ul> |

这时候比较第一个节点和第二个节点都是一样，那么对应的DOM节点不会变化，第三个节点会生成更新在页面上，但是如果碰到这样的情况：

|  |
| --- |
| <ul>  <li>Duke</li>  <li>Villanova</li>  </ul>  <ul>  <li>Connecticut</li>  <li>Duke</li>  <li>Villanova</li>  </ul> |

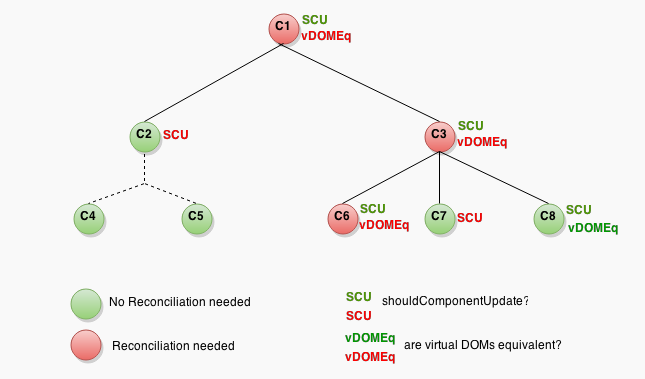
那这时候原来的两个节点都会被替换掉，生成三个新的DOM Node，这也是为什么会React会假设开发会指定Identity/key：

|  |
| --- |
| <ul>  <li key="2015">Duke</li>  <li key="2016">Villanova</li>  </ul>  <ul>  <li key="2014">Connecticut</li>  <li key="2015">Duke</li>  <li key="2016">Villanova</li>  </ul> |

注意:

* key是给React用的，我们通过编程的方式是拿不到这个值的，如果你需要用到这个值，可以放在其他的Attribute中，如ID，可参考[NestedElement](#NestedElement) Example.
* Key应该是stable，可预测的，而不是随机的，不然等于没设
* Key不必是全局的，只要是在sliblings中唯一就可以

如果是Component的props或state更新的时候，会调用该Component生命周期中shouldComponentUpdate函数来确定该Component是否需要重新render，如果不需要那么子节点不需要递归调用，那么对应的DOM node是不会更新的，默认情况下shouldComponentUpdate返回true。但是在极少的情况我们可以重写这个方法来减少更新的情况。关于shouldComponentUpdate，可参考这张图：



**SCU**表示true， **SCU**表示false

**vDOMEq**表示true，**vDOMEq**表示false

因为C2中SCU返回false，所以C4，C5都不需要render，C4和C5的SCU都没有被调用。

C1和C3都SCU返回true，所以React会到子节点中.

C6中SCU返回true，因为virtual DOM确实不相等

C7返回false，不进行比较

C8返回true，即使virtualDOM相等，这时候也会重新调用C8的render方法，完成C8的render方法调用后会比较更新，这时候会发现没必要更新DOM。

图中C6是不可避免需要更新的；C8调用了render并且进行了生成tree的比较；C2，C2的子节点和C7都没有调用render方法

**Homework**

Guide:

https://facebook.github.io/react/tutorial/tutorial.html

Start:

<https://codepen.io/ericnakagawa/pen/vXpjwZ?editors=0010>