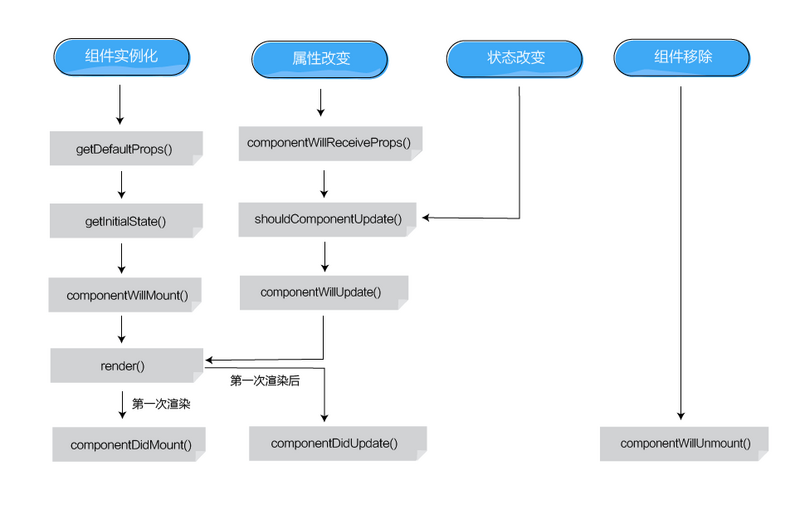
组件生命周期



步骤

第一步：把UI拆分为一个组件的层级

第二步：用React创建一个静态版本

第三步：确定最小（但完备）的 UI state 表达

第四步：确定你的 state 应该存在于哪里

我们需要确定哪个组件可以改变，或者 拥有 这个state.

确定哪些组件要基于 state 来渲染内容。

找到一个共同的拥有者组件（在所有需要这个state组件的层次之上，找出共有的单一组件）。

要么是共同拥有者，要么是其他在层级里更高级的组件应该拥有这个state。

如果你不能找到一个组件让其可以有意义地拥有这个 state，可以简单地创建一个新的组件 hold 住这个state，并把它添加到比共同拥有者组件更高的层级上。

第五步：添加反向数据流

组件

组件定义（组件名大驼峰）

stateless(Function) 优先于 Es6 Class 优先于 React.createClass；

1. function Component(){ ..... }; ////使用时为props.name无状态和周期hook
2. class Component extends React.component{

constructor( props ){

super( props );

this.state = { ...... };

this.handleClick = this.handleClick.bind(this);

}

render(){ ...... }

}

1. let Component = React.createClass({

render(){ ...... }

});

组件通信

（父》子，子》父）我们就实现了一个模态弹窗的基本逻辑，通过传递props，父组件可以控制子组件的状态，而子组件也通过调用父组件传进来的props方法，来通知父元素改变状态来达到影响自己的目的（比如关闭弹窗）。

（子》子）子组件之间的通讯也是一样，一个子组件想作用另一个子组件，可以通过父组件来做中间调度，对于子组件来说，父组件就是一个公共环境，它的一切资源，想调用的话，直接通过props接收就可以了。

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。

1.它是通过props从父级传递来的吗？如果是，它可能不是 state。

2.它随时间变化吗？如果不是,它可能不是 state。

3.你能基于其他任何组件里的 state 或者 props 计算出它吗？如果是,它可能不是state.

4.大部分组件的工作应该是从 props 里取数据并渲染出来。但是，有时需要对用户输入、服务器请求或者时间变化等作出响应，这时才需要使用 State。

5.常用的模式是创建多个只负责渲染数据的无状态（stateless）组件，在它们的上层创建一个有状态（stateful）组件并把它的状态通过 props 传给子级。这个有状态的组件封装了所有用户的交互逻辑，而这些无状态组件则负责声明式地渲染数据。

对应的默认值设置方法：

getInitialState(){ return { ...... } };

getDefaultProps(){ return { ...... } };

state 和 props配合:

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。

组件API setState(object nextState[, function callback])

设置或替换状态----setState/replaceState

设置或替换属性----setProps/replaceProps

强制更新----forceUpdate--forceUpdate([function callback])

判断组件挂载状态----isMounted--返回值：true或false，表示组件是否已挂载到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。

组件生命周期

组件的生命周期可分成三个状态：

Mounting:

getInitialState()

componentWillMount()

componentDidMount()

Updating:正在被重新渲染

componentWillReceiveProps(object nextProps)、

shouldComponentUpdate(object nextProps, object nextState)

componentWillUpdate(object nextProps, object nextState)

componentDidUpdate(object prevProps, object prevState)

Unmounting:

componentWillUnmount()

React 为每个状态都提供了两种处理函数，will 函数在进入状态之前调用，did 函数在进入状态之后调用，三种状态共计五种处理函数。

VOID　componentWillMount()

作用：该方法在整个组件生命周期只会被调用一次，所以可以利用该方法做一些组件内部的初始化工作

VOID　componentDidMount()----ajax、setTimeout、setInterval在这里处理。

作用：DOM 的操作或者其他的一些如请求数据，和第三方库整合的操作。此时可以获取子组件对应的 DOM。

VOID　componentWillUpdate(object nextProps, object nextState)

作用：这个时候可以确定一定会更新组件，可以执行更新前的操作，不可使用setState

VOID componentDidUpdate(object prevProps, object prevState)

作用：可以执行组件更新过后的操作

VOID componentWillUnmount()----当组件被移除时，释放组件已经占用的资源是非常重要的。

作用：主要作清理工作，如清理计时器

React 还提供两种特殊状态的处理函数:

VOID componentWillReceiveProps(object nextProps)：已加载组件收到新的参数时调用

作用：这个时候可以根据新的属性来修改组件状态

BOOLEAN shouldComponentUpdate(object nextProps, object nextState)：组件判断是否重新渲染时调用

作用：该方法让我们有机会决定是否重渲染组件，如果返回 false，那么不会重渲染组件，借此可以优化应用性能（在组件很多的情况）。

Props验证

propTypes

propTypes: {

// js基本类型

optionalArray: React.PropTypes.array, //array\bool\func\number\object\string\any

// 节点

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,

// 自定义验证器

customProp: function(props, propName, componentName) {

if (!/matchme/.test(props[propName])) { return new Error('Validation failed!'); }

}

}

React与DOM

获取DOM元素

ReactDOM.findDOMNode() 与 Refs

只能在生命周期componentDidMount、componentDidUpdate中使用

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。

如：render中<input ref="myInput" />，使用var input = this.refs.myInput;

DOM事件

异步事件需要用----event.persist()

事件捕获和冒泡----

React 中通常我们注册的事件为冒泡事件，如果要注册捕获阶段的事件，可以在事件名称后加 Capture ；如；onClicl----onClickCapture

事件列表

粘贴板事件 {

事件名称：onCopy onCut onPaste

属性：DOMDataTransfer clipboardData

}

编辑事件 {

事件名称：onCompositionEnd onCompositionStart onCompositionUpdate

属性：string data

}

键盘事件 {

事件名称：onKeyDown onKeyPress onKeyUp

属性： {

boolean altKey

number charCode

boolean ctrlKey

boolean getModifierState(key)

string key

number keyCode

string locale

number location

boolean metaKey

boolean repeat

boolean shiftKey

number which

}

}

焦点事件 {

名称：onFocus onBlur

属性：DOMEventTarget relatedTarget

}

表单事件 { 名称：onChange onInput onSubmit}

鼠标事件 {

名称：{

onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave onMouseMove onMouseOut onMouseOver onMouseUp

}

属性：{

boolean altKey

number button

number buttons

number clientX

number clientY

boolean ctrlKey

boolean getModifierState(key)

boolean metaKey

number pageX

number pageY

DOMEventTarget relatedTarget

number screenX

number screenY

boolean shiftKey

}

}

选择事件 {

名称：onSelect

}

触摸事件 {

名称：onTouchCancel onTouchEnd onTouchMove onTouchStart

属性：{

boolean altKey

DOMTouchList changedTouches

boolean ctrlKey

boolean getModifierState(key)

boolean metaKey

boolean shiftKey

DOMTouchList targetTouches

DOMTouchList touches

}

}

UI 事件 {

名称：onScroll

属性：{

number detail

DOMAbstractView view

}

}

滚轮事件 {

名称：onWheel

属性：{

number deltaMode

number deltaX

number deltaY

number deltaZ

}

}

媒体事件 {

名称：{

onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend onTimeUpdate onVolumeChange onWaiting

}

}

图像事件 {

名称：onLoad onError

}

动画事件 {

名称：onAnimationStart onAnimationEnd onAnimationIteration

属性：{

string animationName

string pseudoElement

float elapsedTime

}

}

渐变事件 {

名称：onTransitionEnd

属性： {

string propertyName

string pseudoElement

float elapsedTime

}

}

表单：

交互属性：value----<input> <select> <textarea>

checked---- <input type="checkbox|radio">

Selected----option如多选，multiple=｛true｝value=｛array｝

Style属性----为对象。<div style={style}></div>style为object，React推崇inline样式

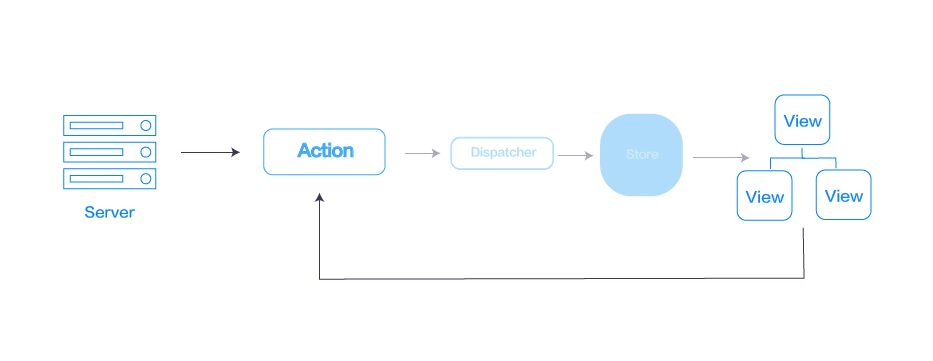
浏览器差异兼容：var divStyle = {

WebkitTransition: 'all', // 'W' 是大写

msTransition: 'all' // 'ms' 为小写

};

Flux



React规范

ES6class组件定义规范：

1.static方法

2.constructor

3.getChildContext

4.componentWillMount

5.componentDidMount

6.componentWillReceiverProps

7.shouldComponentUpdate

8.componentWillUpdate

9.componentDidUpdate

10.componentWillUnmount

1. getter or setter

12.clickHandlers+eventHandlers 如 onClickSubmit() 或 onChangeDescription()

13.render methods 如 renderNavigation() 或 renderProfilePicture()

14.render

命名规范：

组件名称：大驼峰

属性名称：小驼峰

事件处理函数：handleSomething

自定义事件属性名称：onSomething={this.handleSomething}

key: 不能使用数组 index ，构造或使用唯一的 id

组件方法名称：避免使用下划线开头的命名

如：jsx规范

<Foo

superLongParam="bar"

anotherSuperLongParam="baz"

/>

可以配置插件：eslint-plugin-react

container 组件负责获取状态数据，然后传递给与之对应的 normal component，对应表示两个组件的名称对应container 组件是 smart 组件，normal 组件是 dummy 组件，这样的责任分离让 normal 组件更加独立，不需要知道状态数据。

TodoListContainer => TodoList

FooterContainer => Footer

使用 Composition 替代 mixins （good）

小知识点：

ReactDOM.render() 实例化根组件，启动框架，注入标记到原始的 DOM 元素中，作为第二个参数提供。

this.props.children----它表示组件的所有子节点

在React组件中style参数必须为Object ---- style={{color:'red'}}

Mixins----有时一些不同的组件间也需要共用一些功能（为array类型）

key属性----React 会提示对于循环输出的组件，需要有一个唯一的 key 属性----实现渲染优化。

statics----object

statics可以定义组件的类方法,区别与实例方法，实例方法可以访问 this。

displayName----string

为了显示调试信息，每个组件都会有一个名称，JSX 在转为 JS 的时候自动的设置 displayName,也可以自定义。