目录

[api 2](#_Toc531356582)

[this.props.children 2](#_Toc531356583)

[顶层api 2](#_Toc531356584)

[React.Children 2](#_Toc531356585)

[React.Children.map 3](#_Toc531356586)

[React.Children.forEach 4](#_Toc531356587)

[React.Children.count 4](#_Toc531356588)

[React.Children.only 4](#_Toc531356589)

[React.createClass 4](#_Toc531356590)

[React.createElement 4](#_Toc531356591)

[React.cloneElement 5](#_Toc531356592)

[React.createFactory 6](#_Toc531356593)

[ReactDOM.render 6](#_Toc531356594)

[React.unmountComponentAtNode 7](#_Toc531356595)

[React.renderToString 7](#_Toc531356596)

[React.renderToStaticMarkup 7](#_Toc531356597)

[React.isValidElement 8](#_Toc531356598)

[React.DOM.tag 8](#_Toc531356599)

[React.PropTypes 9](#_Toc531356600)

[React.initializeTouchEvents 9](#_Toc531356601)

[createRef 10](#_Toc531356602)

[组件api 10](#_Toc531356603)

[setState 10](#_Toc531356604)

[replaceState 10](#_Toc531356605)

[forceUpdate 10](#_Toc531356606)

[getDOMNode() 10](#_Toc531356607)

[isMounted() 11](#_Toc531356608)

[setProps 11](#_Toc531356609)

[replaceProps 11](#_Toc531356610)

[Refs 11](#_Toc531356611)

[React.forwardRef（） 13](#_Toc531356612)

[组件属性 13](#_Toc531356613)

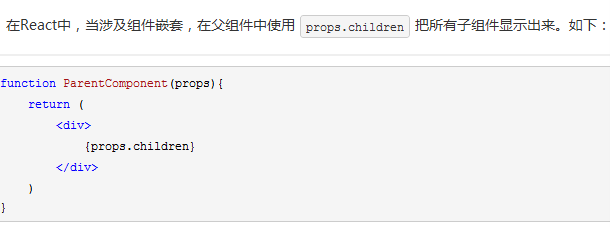
[react-dom 14](#_Toc531356614)

[findDOMNode 14](#_Toc531356615)

# api

## this.props.children

this.props.children 的值有三种可能：如果当前组件没有子节点，它就是 undefined ;如果有一个子节点，数据类型是 object ；如果有多个子节点，数据类型就是 array 。



# 顶层api

## React.Children

Children是组件的集合，想修改属性可以用cloneElement

### React.Children.map

object React.Children.map(object children, function fn [, object context])

React 提供一个工具方法 React.Children 来处理 this.props.children 。我们可以用 **React.Children.map** 来遍历子节点，而不用担心 this.props.children 的数据类型是 undefined 还是 object。

<script type="text/jsx">

var NotesList = React.createClass({

render: function() {

return (

<ol>

{

React.Children.map(this.props.children, function (child) {

**return <li>{child}</li>;**

})

}

</ol>

);

}

});

React.render(

<NotesList>

<span>hello</span>

<span>hello</span>

</NotesList>,

document.body

);

</script>

### React.Children.forEach

**React.Children.forEach**(object children, function fn [, object context])

类似于 React.Children.map()，但是不返回对象。

### React.Children.count

React.Children.count(object children)

返回子元素的总数：

var nums = React.Children.count(this.props.children);

### React.Children.only

React.Children.only(object children)

返回仅有的一个子元素，否则（没有子元素或超过一个子元素）报错且不渲染任何东西：

<div>Hello {React.Children.only(this.props.children)}</div>

## React.createClass

**已经被移除**

在该方法里面，所有的 this 都会在最终调用时自动地晚绑定到当前组件的构造器上

已过时

最新 Component

## React.createElement

**参数：TYPE（string/ReactClass），[PROPS（object）]，[CHILDREN（ReactElement）]**

创建一个指定类型的React元素，注意第三个参数CHILDREN可以是任意个React元素：

React.render(

React.createElement('div', null,

React.createElement( 'p', null,

React.createElement('span', null, 'Hello,'),

React.createElement('span', null, 'world,'),

React.createElement( Component, {a : 1})

)

), document.body

);

React.createElement 是一个语法糖：

var reactElement = React.createElement(type, props, children);

//等价于下面两行：

var div = React.createFactory('div');

var reactDivElement = div(props, children);

## React.cloneElement

**参数：TYPE（ReactElement），[PROPS（object）]，[CHILDREN（ReactElement）]**

克隆并返回一个新的 ReactElement （内部子元素也会跟着克隆），新返回的元素会保留有旧元素的 props、ref、key，也会集成新的 props（只要在第二个参数中有定义）

**createElement 的第一个参数必须是字符串或 ReactClass，而在 cloneElement 里第一个参数应该是 ReactElement：**

(理解：reactClass是申明好的组件，reactElement是已经创建好的组件，已经传入参数创建出来了)(component和<component/>)

## React.createFactory

**参数：TYPE（string/ReactClass）**

返回一个某种类型的ReactElement工厂函数，可以利用返回的函数来创建一个ReactElement（配置 props 和 children）：

p = React.createFactory(Component),

ReactElementP = **p({a:1})**,

div = React.createFactory('div'),

ReactElementDiv = **div(null, ReactElementP);**

## React.unmountComponentAtNode

可以已经移到reactDOM里面了

**参数：CONTAINER（DOMElement）**

从 container 指定的 DOM 中移除已经挂载的 React 组件，清除相应的事件处理器和 state。如果在 container 内没有组件挂载，这个函数将什么都不做。如果组件成功移除，则返回 true；如果没有组件被移除，则返回 false：

var isUnmount = React.unmountComponentAtNode(document.body);

console.log(isUnmount); //打印出true

## React.renderToString

**参数：REACTELEMENT（ReactElement）**

React为服务端提供的一个方法，可以直接输出 ReactElement 为 HTML 字符串，将这些标记发送（比如 res.write(HTMLString)）给客户端，可以获得更快的页面加载速度，并且有利于搜索引擎抓取页面，方便做 SEO（主要是百度不争气，谷歌早可以从内存中去抓最终生成的HTML内容了）：

comHTML = React.renderToString(com);

console.log(comHTML); //输出“<p data-reactid=".0" data-react-checksum="-2122315716">123</p>”

## React.renderToStaticMarkup

**参数：REACTELEMENT（ReactElement）**

类似 React.renderToString ，但只生成纯粹的HTML标记字符串，不会包含类似 data-reactid 之类的React属性，从而节省字节数：

comHTML = React.renderToStaticMarkup(com);

console.log(comHTML); //输出“<p>123</p>”

## React.isValidElement

**参数：SOMETHING**

判断参数是否一个合法的 ReactElement，并返回 Boolean 值：

var com = <Component/>,

com2 = '<Component/>';

console.log(React.isValidElement(com)); //true

console.log(React.isValidElement(com2)); //false

## ****React.PropTypes****

用于组件内部验证传入 Props 的类型，如果传入的类型不匹配，React 会打印出警告：**isRequired表示必须传入**

var Component = React.createClass({

propTypes : {

a : React.PropTypes.number.**isRequired**, //**必须**传入一个名为“a”、类型为number的props

callback : React.PropTypes.func //如果传入了名为“callback”的props，其类型必须是函数

},

render : function() {

return this.props.a==1 ? <p onClick={this.props.callback}>123</p> : null

}

});

## ****React.initializeTouchEvents****

**参数：SholdUserTouch（boolean）**

开启或关闭 React 的触摸事件机制，传入参数 true 使 React 能处理移动设备的触摸（ touch ）事件：

**React.initializeTouchEvents(true);**

var Component = React.createClass({

render : function() {

return <p onTouchStart={this.props.callback}>123</p>

}

});

var cb = function(){

alert('touch!')

};

React.render(

<Component callback={cb} />, document.body

)

## createRef

# reactDOM

## render

**参数：reactElement（ReactElement），container（DOMElement），[CALLBACK（function）]**

渲染一个 ReactElement 到 container 指定的 DOM 中，并返回一个到该组件的引用。如果提供了可选的回调函数，则该函数将会在组件渲染或者更新之后调用:

如果我们希望在组件外部获取到组件内部（能通过 this 访问）的东西，可以将 React.render 的返回值赋予一个变量，在后续调用该变量即可。

var p = React.render(

<Component a="1" />, document.body, function(){

console.log('OK')

}

);

setTimeout(function(){

console.log(**p.props.a**); //打印出“1”

}, 2000)

## findDOMNode

findDOMNode获取到的永远是html的node，ref根据绑定的情况可能是vnode可能是htmNode。

## React.DOM.tag

**参数：attribute（object/null），children（string/ReactElement）**

常规是用于在非 JSX 下来创建 ReactElement，tag 表示相应的DOM类型（比如“div”、“p”）。另外首个参数可以定制相关的 DOM 属性（比如“name”），第二个参数表示 DOM 内的内容：

var div = React.DOM.div({name : 'div1'}, 'HELLO ', React.DOM.span(null, <em>WORLD</em>));

React.render(

div, document.body

)



# 组件api

## setState

***参数： nextState（object）, [callback（function）]***

## replaceState

***参数： nextState（object）, [callback（function）]***

对整个state进行完全替换

## forceUpdate

***参数： [callback（function）]***

顾名思义就是在任何调用的时刻强制渲染组件，例如即使 shouldComponentUpdate 返回了false：

## getDOMNode()

返回组件/ReactElement挂载到页面上所对应的DOM元素：

若 render 返回 null 或者 false 的时候，this.getDOMNode() 将返回 null。

**这个方法已经被移除了。使用ReactDOM.findDOMNode(this.refs.textarea)**

**（0.15就已经移除了）**

## isMounted()

返回一个 Boolean 值，如果组件挂载到了 DOM 中，isMounted() 返回 true。

其适用于异步请求的情景下：

## setProps

***参数： nextProps（object）, [callback（function）]***

## replaceProps

***参数： nextProps（object）, [callback（function）]***

## Refs

https://segmentfault.com/a/1190000015113359

结合ref，获得的是react组件，可以调用对应的方法，比如getDOMNode

大多数情况下不需要寻找原生节点，很多方法react节点就有

16以前:函数式

ref={ele => { this.textInput = ele}}

16以后： reactApi

this.textInput = React.createRef();

ref={this.textInput}

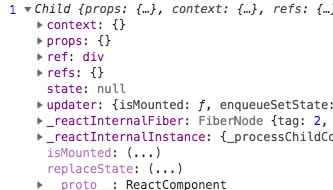
在V16版本前，我们可以直接通过变量访问元素的方法，在V16后，我们需要通过 this.textInput.current，即真实的元素是通过current属性来引用的。

如果是DOM元素，那引用的就是DOM元素。

那么引用的就是组件实例，引用的就是react组件对象。

函数式有一个问题：

每次组件重新渲染（组件更新时）的时候，行内函数都会执行两次，第一次的ele的值为空，第二次才为真正的DOM对象。因为在每次渲染中React都会创建一个新的函数实例。因此，在更新时，React 需要清理旧的 ref 并且设置新的。通过将 ref 的回调函数定义成类的绑定函数的方式可以避免上述问题，

****

****

### React.forwardRef（）

用于输出组件的ref，正在父组件拿到的ref是子组件的reac对象，不是dom元素对象。停过forwardRef包裹子组件后，父组件可以直接拿到子组件的dom元素对象。

### 组件属性

正常情况下，组件属性是不会出现在渲染好的dom节点中的，只有html原生支持的属性会绑定到dom节点上(比如data-)

# react-dom

## findDOMNode