🡪

React采用于HTML DOM事件处理相似的方式，在组件上注册事件监听器。当用户与组件交互，触发事件的时候被注册的事件监听器捕获进行处理。

🡪

React事件与DOM事件命名方式基本相同，不过React以驼峰命名法命名。例如DOM点击事件为onclick, 而React为onClick

🡪

React组件注册事件监听器可以为行内风格，也可以额外定义方法处理。

例如：

<div className="m-2">  
 <div className="h4 bg-primary text-white text-center p-2">  
 {this.state.message}  
 </div>  
 <div className="text-center">  
 <button className="btn btn-primary" onClick={() => this.setState({message:"Clicked!"})}>Click Me</button> //采用行内风格指定  
 </div>  
</div>

<div className="m-2">  
 <div className="h4 bg-primary text-white text-center p-2">  
 {this.state.message}  
 </div>  
 <div className="text-center">  
 <button className="btn btn-primary" onClick={this.handleEvent}>Click Me</button> //采用额外定义方法处理  
 </div>  
</div>

🡪

为React组件的事件监听器指定this关键字，使处理函数可以通过this获取组件内部状态。我们有两种风格进行指定this。

错误写法，例如组件不指定this关键字的情况：

export default class App extends Component{  
 constructor(props) {  
 super(props);  
 this.state = {  
 message: "Ready"  
 }  
 }  
  
 handleEvent(){  
 this.setState({  
 message: "Clicked"  
 });  
 }  
  
 render() {  
 return (  
 <div className="m-2">  
 <div className="h4 bg-primary text-white text-center p-2">  
 {this.state.message}  
 </div>  
 <div className="text-center">  
 <button className="btn btn-primary" onClick={this.handleEvent}>Click Me</button>  
 </div>  
 </div>  
 )  
 }  
}

当点击按钮以后，浏览器报错无法找到setState方法。原因是我们没有指定this.

第一种风格，在组件定义方法时候指定this。本质是当onClick事件触发时直接调用监听方法。:

handleEvent = () => { // 在组件内部定义监听方法时候指定this  
 console.log(this);  
 this.setState({  
 message: "Clicked"  
 });  
}

render () {  
 return (  
 <div className="m-2">  
 <div className="h4 bg-primary text-white text-center p-2">  
 {this.state.message}  
 </div>  
 <div className="text-center">

//   
 <button className="btn btn-primary" onClick={this.handleEvent}>Click Me</button>  
 </div>  
 </div>  
 )  
}

第二种风格，在html元素注册事件监听器的时候指定this。本质是当onClick事件触发时先调用一个箭头匿名函数，然后再这个箭头匿名函数中再调用事件监听方法。**这种方法可以传递除React自带event事件参数以外额外的参数到事件监听方法中。**

handleEvent(){ // 在组件内部定义监听方法时候指定this  
 console.log(this);  
 this.setState({  
 message: "Clicked"  
 });  
}

render () {  
 return (  
 <div className="m-2">  
 <div className="h4 bg-primary text-white text-center p-2">  
 {this.state.message}  
 </div>  
 <div className="text-center">

// 在html元素注册事件监听器的时候指定this  
 <button className="btn btn-primary" onClick={ () => this.handleEvent()}>Click Me</button>  
 </div>  
 </div>  
 )  
}

🡪

指定事件监听器的实质的定义一个回调函数作为事件监听器。当事件触发以后，React会调用我们定义的回调函数作为事件响应。React调用的形式总是回调函数 (React Event对象)

例子：

huiTest(){  
 ***console***.log(arguments[0].nativeEvent);  
}  
  
huiTest2 = (event) =>{  
 ***console***.log(event.nativeEvent);  
};

render() {  
 return (  
 <div className="m-2">  
 <div className="text-center">

//点击按钮后，React会调用我们定义的huiTest作为回调函数，并且以一个event参数调用, 由于我们定义的huiTest不接受参数，所以JS会把多余的参数放入arguments中。Arguments[0]便是React event对象。  
 <button className="btn btn-danger" onClick={this.huiTest}>  
 huiTest  
 </button>

//点击按钮后，React会调用我们定义的huiTest2作为回调函数，并且以一个event参数调用,由于我们定义的huiTest接受一个参数，所以huiTest2的event参数将是React调用的event参数

<button className="btn btn-danger" onClick={this.huiTest2}>  
 huiTest2  
 </button>  
 </div>  
 </div>  
 )  
}

🡪

React的事件对象(Synthetic Event)

当事件触发的时候，React将产生一个事件对象实例，它的类型是SyntheticEvent而非原始DOM中的Event事件对象。SyntheticEvent是原始DOM Event对象的包裹，提供了原始DOM Event的功能并且在其之上加入了额外的特性。

SysthenticEvent对象的一些基本属性

|  |  |
| --- | --- |
| nativeEvent | 返回HTML DOM事件对象 |
| target | 返回事件源，为哪个React组件 |
| timestamp | 返回事件发生的时间戳 |
| type | 返回事件的类型 |
| isTrusted | 如果为true,代表由browser触发。为false则代表手动触发 |
| preventDefault() | 阻止事件的默认行为 |
| defaultPrevented | 如果为true则表示preventDefault()被调用过，否则为false |
| persist() | 阻止React重用事件对象 |

🡪

当事件触发的时候，React将产生一个事件对象实例，React的事件监听函数可以接收。

例如

export default class App extends Component{  
 constructor(props) {  
 super(props);  
 this.state = {  
 message: "Ready"  
 }  
 }  
  
 handleEvent = (e) => { //指定的第一个参数为React事件对象SystenticEvent。  
 console.log(e.target);  
 this.setState({  
 message: `Event: ${e.type}`  
 });  
 };  
  
 render() {  
 return (  
 <div className="m-2">  
 <div className="h4 bg-primary text-white text-center p-2">  
 {this.state.message}  
 </div>  
 <div className="text-center">

// onClick指定一个事件监听器函数（回调函数）引用， 无需表明接受的参数列表。React会使用该函数在事件触发时候，React自行调用并且传入一个类型为Synthetic Event的事件对象参数。  
 <button className="btn btn-primary" onClick={ this.handleEvent }>Click Me</button>  
 </div>  
 </div>  
 )  
 }  
}

🡪

React事件persist()方法。当事件被接受处理以后，React默认会重用事件对象，从而重置所有事件对象（从而event.target为null）。如果在异步操作当中，想在异步操作中再次获取事件中属性则为空。需要调用persist()方法，阻止事件对象重置，从而在异步操作当中也可以获取事件的属性。

例如：

export default class App extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 message: "Ready",  
 counter: 0  
 }  
 }  
  
 handleEvent = (event) => {

event.persist() // 阻止React重置事件对象  
 this.setState({  
 counter: this.state.counter +1  
 }, () =>{  
 this.setState({  
 message: `${event.type}: ${this.state.counter}` //异步操作，如果没有调用event.persist()方法则进行此异步操作的时候，事件对象可能已经被重置无法获取  
 })  
 })  
 };  
  
 render() {  
 return (  
 <div className="m-2">  
 <div className="h4 bg-primary text-white text-center p-2">  
 {this.state.message}  
 </div>  
 <div className="text-center">  
 <button className="btn btn-primary" onClick={this.handleEvent}>Click Me</button>  
 </div>  
 </div>  
 )  
 }  
}

🡪

React中为事件监听器传入自定义的额外参数。除了React会传入的event对象外，我们有时候还需要为事件监听器传入额外的参数。

例如：

handleEvent = (event, newTheme) => {  
 event.persist();  
 this.setState({  
 counter: this.state.counter +1,  
 theme: newTheme  
 }, () =>{  
 this.setState({  
 message: `${event.type}: ${this.state.counter}`  
 })  
 })  
};  
  
render() {  
 return (  
 <div className="m-2">  
 <div className={`h4 bg-${this.state.theme} text-white text-center p-2`}>  
 {this.state.message}  
 </div>  
 <div className="text-center">  
 <button className="btn btn-primary" onClick={ (event) => this.handleEvent(event, "primary")}>

//需要额外的参数newTheme,如果直接写onClick={this.handle}则handleEvent函数只能获取event对象，无法获取额外的自定义参数。所以我们需要先定义一个匿名箭头函数接受React的event对象，然后在这个函数中调用我们的事件监听器，为事件监听器传入event对象以及自定义参数  
 Normal  
 </button>  
 <button className="btn btn-danger" onClick={(event) => this.handleEvent(event, "danger")}>  
 Danger  
 </button>  
 </div>  
 </div>  
 )  
}

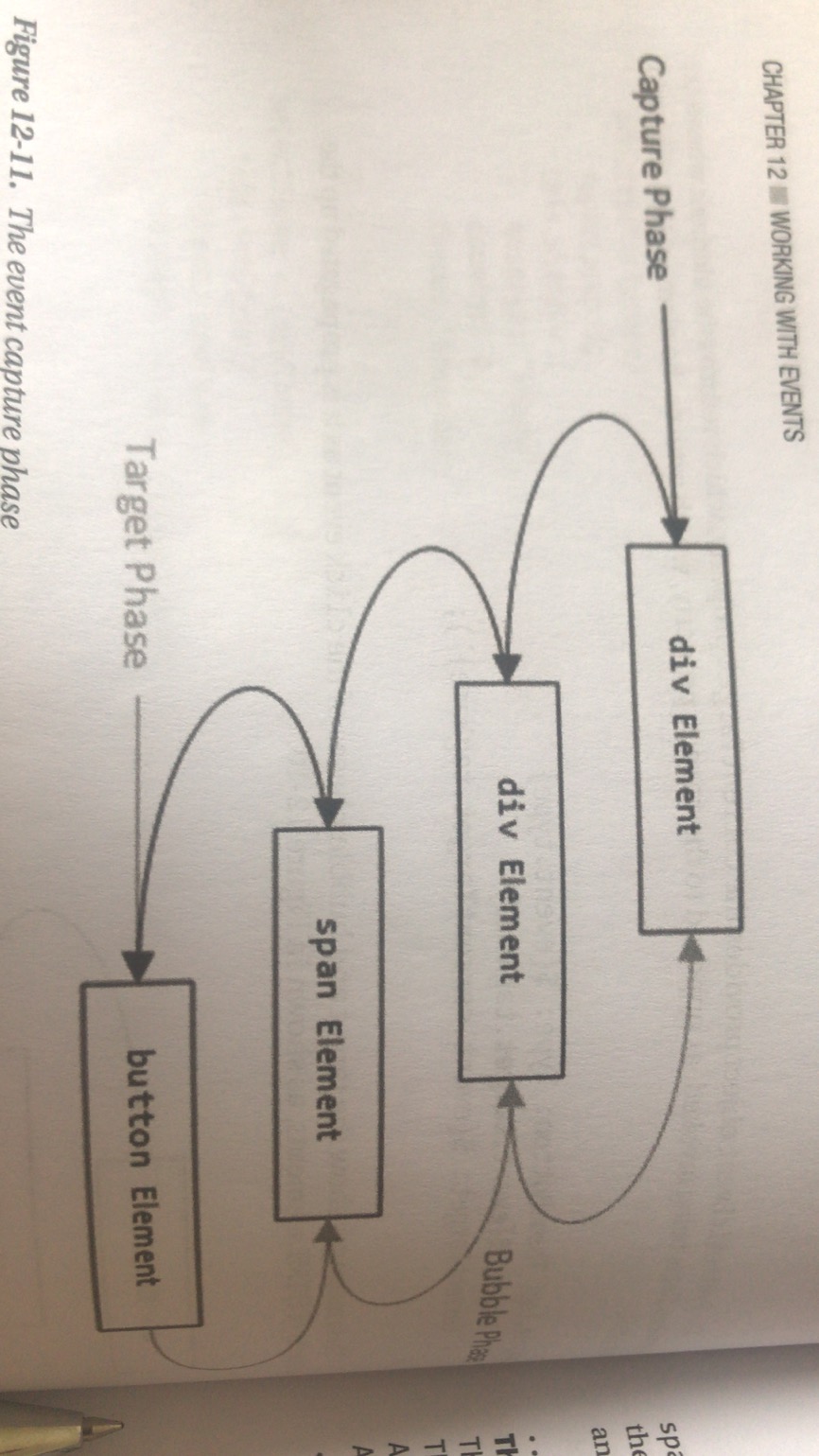
🡪

Reactjs的事件传播模型

Reactjs事件对象关于传播的属性以及方法

|  |  |
| --- | --- |
| eventPhase | 返回该事件对象所在的传播阶段***（由于React特殊性，此属性永远为bubble 阶段。）***  ***如果需要根据事件所在的阶段执行不同的逻辑，则***   1. ***分辨是否存在于capturePhase阶段。由于capturePhase阶段需要capture作为后缀注册事件监听器，所以我们可以专门定义方法作为capture阶段的监听器。或者给监听器传入额外的参数*** 2. ***分辨是否存在于target或者bubble阶段。由于target或者bubble阶段都是使用相同关键字注册事件监听器，所以我们判读方法是event.target是否等于event.currentTarget，如果相等则为处于target phase如果不相等则处于bubble phase.*** |
| bubbles | 返回该事件对象是否存在bubble阶段 |
| target | 返回事件对象的发生源 |
| currentTarget | 返回该事件对象正在被什么元素的事件监听器处理 |
| stopPropgagation() | 停止该事件对象的传播 |
| isPropagationStopped() | 返回true则代表该事件对象已经调用过stopPropagation()方法 |

Reactjs的事件传播模型



点击按钮以后，首先进入Capture Phase从上层祖父组件传递，一层层往下直到点击组件。传播到点击组件时候进入Target Phase,到了Target Phase以后从下往上传播进入Bubble Phase阶段。

P.s Target Phase以及Bubble Phase注册事件监听器关键字为onClick,onXxxx等。如果要注册Capture Phase的事件监听器，需要加入后缀Capture.例如onClickCapture.

P.s 并不是所有事件都存在bubble阶段。例如onFocus事件不会有bubble阶段。可以通过调用event.bubbles来查看是否存在bubble阶段。

P.s onClick等事件监听器注册只能用于HTML元素标签中，不能用于自定义的组件标签上。如果用于自定义的组件标签，例如<Component1 onClick={this.handleClick} /> 不会报错但是没有任何作用，也不会触发任何方法。