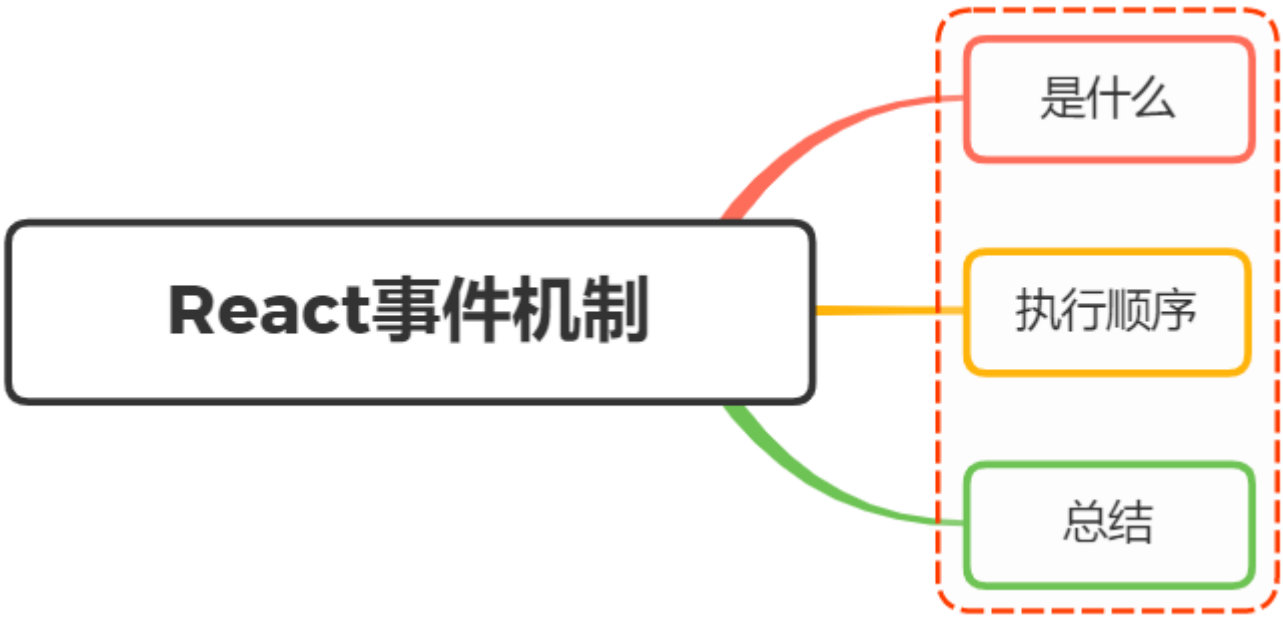
面试官：说说React的事件机制？



一、是什么

React基于浏览器的事件机制自身实现了一套事件机制，包括事件注册、事件的合成、事件冒泡、事件派发等

在React中这套事件机制被称之为合成事件

合成事件（SyntheticEvent）

合成事件是 React 模拟原生 DOM 事件所有能力的一个事件对象，即浏览器原生事件的跨浏览器包装器

根据 W3C 规范来定义合成事件，兼容所有浏览器，拥有与浏览器原生事件相同的接口，例如：

```
const button = <button onClick={handleClick}>按钮</button>
```

如果想要获得原生DOM事件，可以通过e.nativeEvent属性获取

```
const handleClick = (e) => console.log(e.nativeEvent);;
const button = <button onClick={handleClick}>按钮</button>
```

从上面可以看到React事件和原生事件也非常的相似，但也有一定的区别：

- 事件名称命名方式不同

```
// 原生事件绑定方式
<button onclick="handleClick()">按钮命名</button>
```

```
// React 合成事件绑定方式
const button = <button onClick={handleClick}>按钮命名</button>
```

- 事件处理函数书写不同

```
// 原生事件 事件处理函数写法
<button onclick="handleClick()">按钮命名</button>

// React 合成事件 事件处理函数写法
const button = <button onClick={handleClick}>按钮命名</button>
```

虽然onclick看似绑定到DOM元素上，但实际并不会把事件代理函数直接绑定到真实的节点上，而是把所有的事件绑定到结构的最外层，使用一个统一的事件去监听

这个事件监听器上维持了一个映射来保存所有组件内部的事件监听和处理函数。当组件挂载或卸载时，只是在这个统一的事件监听器上插入或删除一些对象

当事件发生时，首先被这个统一的事件监听器处理，然后在映射里找到真正的事件处理函数并调用。这样做简化了事件处理和回收机制，效率也有很大提升

二、执行顺序

关于React 合成事件与原生事件执行顺序，可以看看下面一个例子：

```
import  React  from 'react';
class App extends React.Component{

  constructor(props) {
    super(props);
    this.parentRef = React.createRef();
    this.childRef = React.createRef();
  }
  componentDidMount() {
    console.log("React componentDidMount！");
    this.parentRef.current?.addEventListener("click", () => {
      console.log("原生事件：父元素 DOM 事件监听！");
    });
    this.childRef.current?.addEventListener("click", () => {
      console.log("原生事件：子元素 DOM 事件监听！");
    });
    document.addEventListener("click", (e) => {
      console.log("原生事件：document DOM 事件监听！");
    });
  }
  parentClickFun = () => {
    console.log("React 事件：父元素事件监听！");
  };
  childClickFun = () => {
    console.log("React 事件：子元素事件监听！");
```

```
  };
  render() {
    return (
      <div ref={this.parentRef} onClick={this.parentClickFun}>
        <div ref={this.childRef} onClick={this.childClickFun}>
          分析事件执行顺序
        </div>
      </div>
    );
  }
}
export default App;
```

输出顺序为：

```
原生事件：子元素 DOM 事件监听！
原生事件：父元素 DOM 事件监听！
React 事件：子元素事件监听！
React 事件：父元素事件监听！
原生事件：document DOM 事件监听！
```

可以得出以下结论：

- React 所有事件都挂载在 document 对象上
- 当真实 DOM 元素触发事件，会冒泡到 document 对象后，再处理 React 事件
- 所以会先执行原生事件，然后处理 React 事件
- 最后真正执行 document 上挂载的事件

对应过程如图所示：

所以想要阻止不同时间段的冒泡行为，对应使用不同的方法，对应如下：

- 阻止合成事件间的冒泡，用e.stopPropagation()
- 阻止合成事件与最外层 document 上的事件间的冒泡，用e.nativeEvent.stopImmediatePropagation()
- 阻止合成事件与除最外层document上的原生事件上的冒泡，通过判断e.target来避免

```
document.body.addEventListener('click', e => {
    if (e.target && e.target.matches('div.code')) {
        return;
    }
    this.setState({   active: false,    });   });
}
```

三、总结

React事件机制总结如下：

- React 上注册的事件最终会绑定在document这个 DOM 上，而不是 React 组件对应的 DOM(减少内存开销就是因为所有的事件都绑定在 document 上，其他节点没有绑定事件)
- React 自身实现了一套事件冒泡机制，所以这也就是为什么我们 event.stopPropagation()无效的原因。
- React 通过队列的形式，从触发的组件向父组件回溯，然后调用他们 JSX 中定义的 callback
- React 有一套自己的合成事件 SyntheticEvent

参考文献

- https://zh-hans.reactjs.org/docs/events.html
- https://segmentfault.com/a/1190000015725214?utm_source=sf-similar-article
- https://segmentfault.com/a/1190000038251163