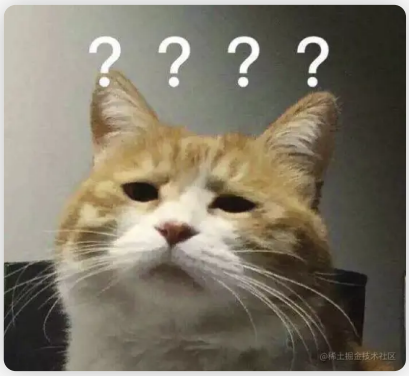
一道面试题引发对React合成事件的深入思考（一）

写在前面

事情是这样的，前几天遇到了一个关于React合成事件的面试题，看完答案的我一脸问号。



所以也是深入的了解了一下React的合成事件，写了一篇总结。

面试题

面试题代理如下：

javascript 复制代码

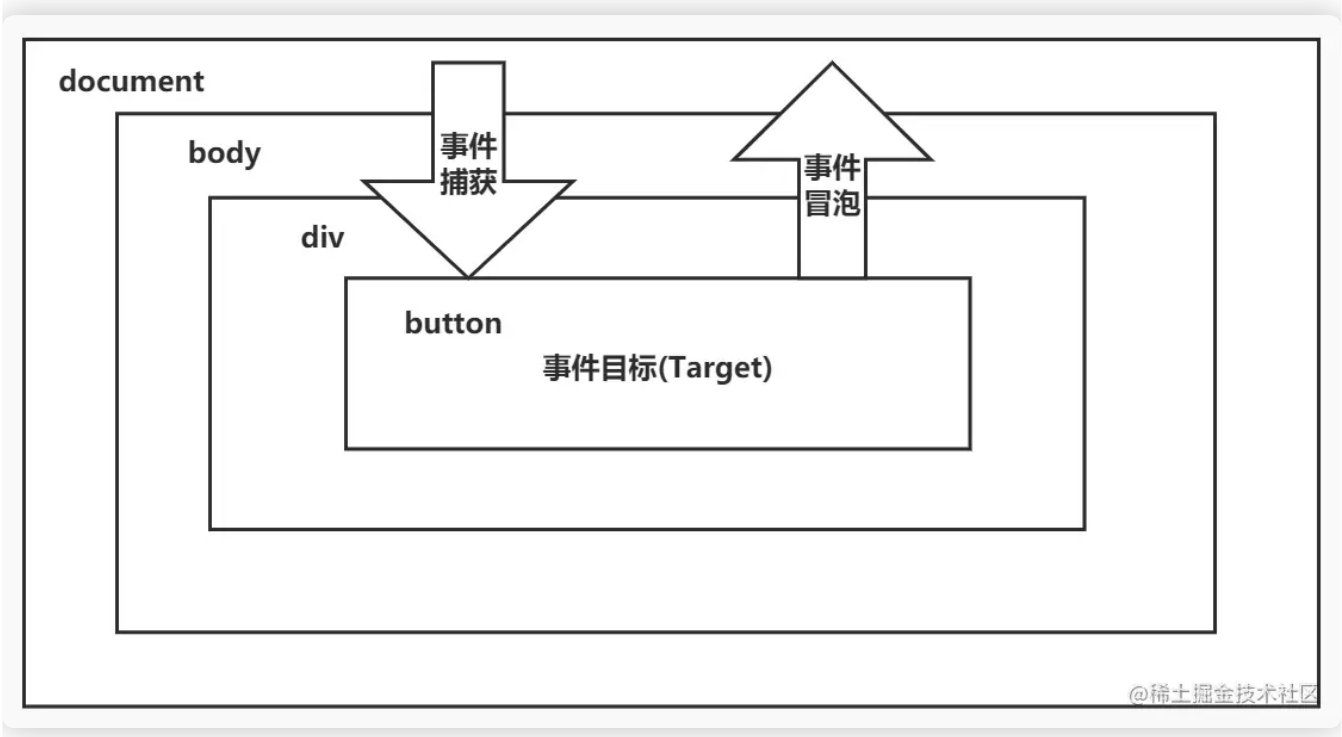
```
import React, { useEffect } from 'react';

function Test() {
  useEffect(() => {
    const parent = document.getElementById('parent');
    const child = document.getElementById('child');

    parent?.addEventListener(
      'click',
      function () {
        console.log('dom parent capture');
      },
      true,
    );
    parent?.addEventListener(
```

```
      'click',
      function () {
        console.log('dom parent bubble');
      },
      false,
    );

    child?.addEventListener(
      'click',
      function () {
        console.log('dom child capture');
      },
      true,
    );
    child?.addEventListener(
      'click',
      function () {
        console.log('dom child bubble');
      },
      false,
    );
  }, []);

  return (
    <div
      id="parent"
      onClick={() => {
        console.log('react parent bubble');
      }}
      onClickCapture={() => {
        console.log('react parent capture');
      }}
    >
      <p
        id="child"
        onClick={() => {
          console.log('react child bubble');
        }}
        onClickCapture={() => {
          console.log('react child capture');
        }}
      >
        click me
      </p>
    </div>
  );
}
```

简单的描述一下，就是给定一个React的函数组件，分别给 `div#parent` 和 `p#child` 在捕获阶段和冒泡阶段，为两个元素注册了React事件（通过 `onClick` 和 `onClickCapture` 这种react封装的方式），和原生事件（通过 `addEventListener` 这种原生方式），问点击一下触发顺序。

我想这还不是手到擒来，不就是先捕捉后冒泡嘛，不对，React事件先触发还是原生事件先触发呢？想了半天，好悬cpu没给我干烧。所以就有了这篇文章。

基础知识



事件捕获

最早是由 网景公司 提出，在捕获阶段：

- 浏览器检查元素的最外层的祖先 `<html>` ，是否在捕获阶段中注册了一个事件处理程序，如果有，就运行它。
- 然后，它移动到 `<html>` 中单机元素的下一个祖先元素，并执行相同的操作，然后是触发事件再下一个祖先元素，以此类推，直到到达实际触发的元素。

官方的解释都是比较晦涩的，实际上就是从顶级元素开始，依次执行注册事件，直到实际触发元素。

事件冒泡

最早是由微软的IE团队提出，在冒泡阶段：

- 浏览器检查实际触发的元素是否在冒泡阶段中注册了一个事件处理程序，如果是，则运行它
- 然后它移动到下一个直接的祖先元素，并做同样的事情，然后是下一个，等等，直到它到达元素。

冒泡阶段是和捕获阶段触发的过程是相反的，是从实际触发的元素开始，直到顶级元素。

我们从名字就可以猜出大概，把一个元素的响应函数委托给另一个元素。假如说我们有一些类似处理方式处理元素，比如说 `<ul>` 下面会有几个处理相同的 `<li>` ，我们就可以把这个响应函数 委托 给这个 `<ul>` 元素，让他统一去处理。

事件委托 的实现正是利用了浏览器的事件触发机制，无论是事件捕获还是事件冒泡都会经过触发元素的所有祖先祖先元素。而正是利用委托这个一个优点，我们减少了事件监听，从而减小了内存消耗，提升我们的效率。

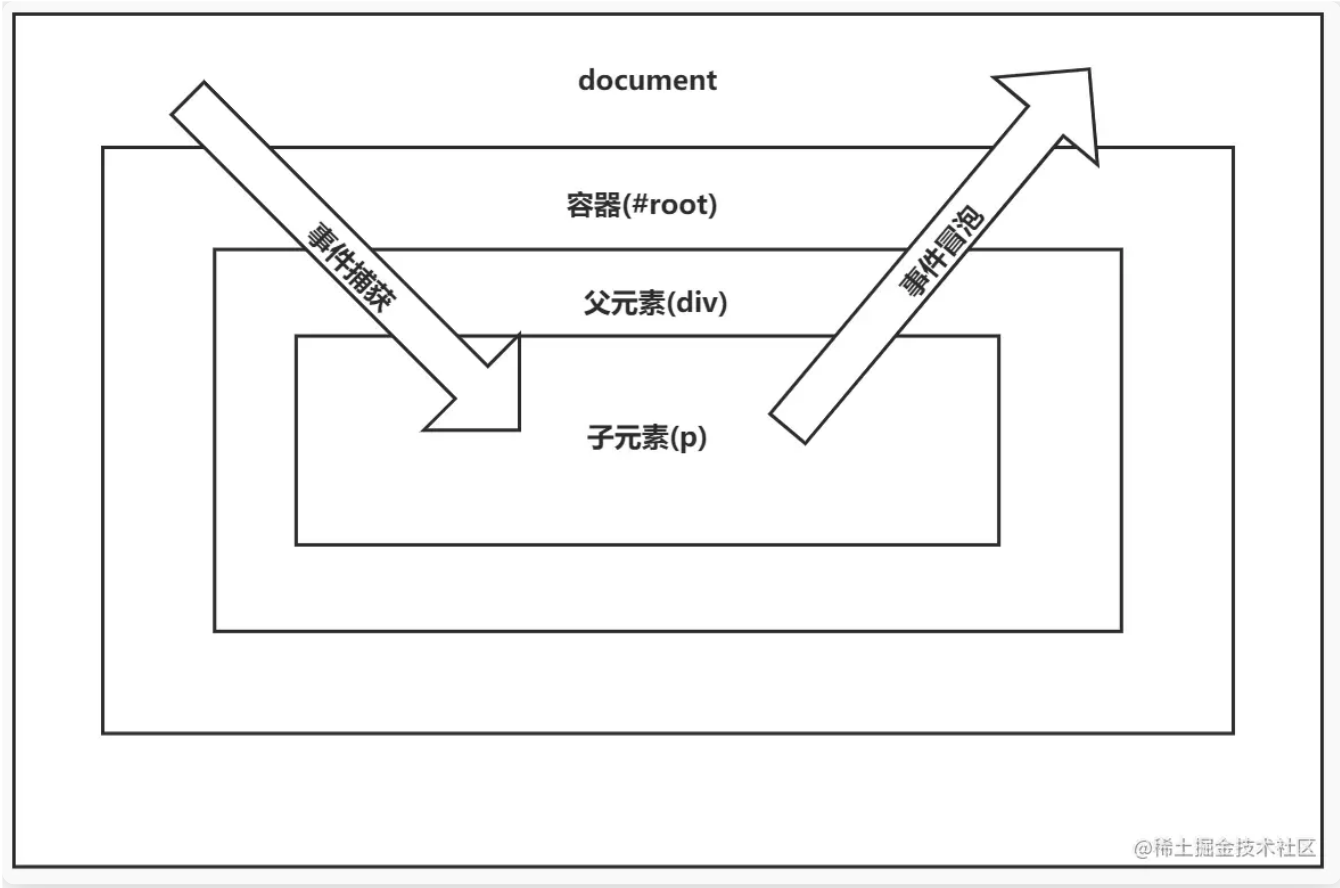
那读到这里问题就来了，我们为什么不把所有的事件都绑在一个页面元素都会经过的元素上面，让他监听所有的事件，那我们的效率不是大大的提升了吗。

实际上React也是这么做的，他把所有的监听都放在了root元素上面。

面试题分析

```javascript 复制代码
// React17
ReactDOM.createRoot(document.getElementById('root')!).render(<App />);

// React18
const root = createRoot(document.getElementById("root"));
```

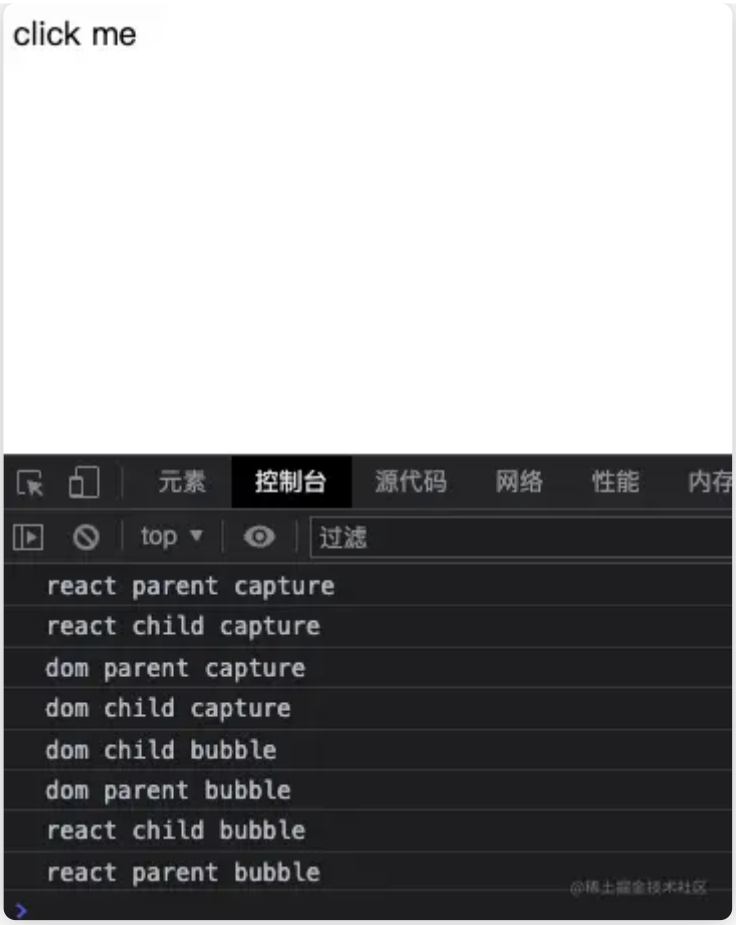
无论是18版本还是之前版本，React都要求我们传入一个根元素作为我们所有DOM的父节点，React就是把所有的事件监听都是放在 `div#root` 上面的，即React把所有事件都委托给了 `div#root` ，并在它上面监听所有触发的事件。以题目中的点击事件为例，React事件都是绑定在 `div#root` 上面的，即只有捕捉或是冒泡过程经过 `div#root` 时，才会触发相应的事件。



图中示例，当捕获阶段经过 `div#root` 时，会依次触发 `react parent capture``react child capture`，到达 `div#parent` 时触发 `dom parent capture`，到达 `p#child` 时触发 `dom child capture`。

在冒泡阶段，从 `p#child` 出发，首先触发 `dom child bubble`，到达 `div#parent` 时，触发 `dom parent bubble`，后到达 `div#root`，依次触发 `react child bubble` 和 `react parent bubble`。

这是我们分析的过程，看一下具体是如何打印的。



最后的结果跟我们的分析一般无二，我们的分析并没有问题，那问题就又来了，我们该如何简单的实现这样一个事件委托呢？

我们根据上面的基础知识，尝试着去实现这样一个事件委托。

xml 复制代码

```xml
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>

  <body>
    <div id="root">
      <div id="parent">
        <p id="child">click</p>
      </div>
    </div>
```

```
</body>
<script>
  var parentBubble = () => {
    console.log('react parent bubble');
  };
  var parentCapture = () => {
    console.log('react parent capture');
  };
  var childBubble = () => {
    console.log('react child bubble');
  };
  var childCapture = () => {
    console.log('react child capture');
  };

  // 获取目标节点
  const root = document.getElementById('root');
  const parent = document.getElementById('parent');
  const child = document.getElementById('child');

  // 添加React监听函数
  parent.onClick = parentBubble;
  parent.onClickCapture = parentCapture;
  child.onClick = childBubble;
  child.onClickCapture = childCapture;
  //模拟React中的事件委托
  root.addEventListener('click', dispatchEvent.bind(null, true), true); // 捕获事件
  root.addEventListener('click', dispatchEvent.bind(null, false), false); // 冒泡事件

  // 派发事件
  function dispatchEvent(isCapture, nativeEvent) {
    let paths = [];

    let currentTarget = nativeEvent.target;
    while (currentTarget) {
      paths.push(currentTarget);
      currentTarget = currentTarget.parentNode;
    }
    console.log(paths); // [p#child, div#parent, div#root, body, html, document]
    if (isCapture) {
      // 根据paths中的元素，倒序触发
      for (let i = paths.length - 1; i >= 0; i--) {
        let handler = paths[i].onClickCapture;
        handler && handler();
      }
    } else {
      for (let i = 0; i < paths.length; i++) {
        let handler = paths[i].onClick;
        handler && handler();
```

```
        }
      }
    }

    // 添加原生监听
    parent.addEventListener(
      'click',
      () => {
        console.log('dom parent capture');
      },
      true,
    );
    parent.addEventListener(
      'click',
      () => {
        console.log('dom parent bubble');
      },
      false,
    );
    child.addEventListener(
      'click',
      () => {
        console.log('dom child capture');
      },
      true,
    );
    child.addEventListener(
      'click',
      () => {
        console.log('dom child bubble');
      },
      false,
    );
  </script>
</html>
```

我们通过事件委托的方式，来模拟了一个React合成事件的触发方式，表现结果跟我们之前通过React实现也是一样的。

这里的代码注释比较详细，感兴趣可以看一下，主要是 `dispatchEvent` 的实现，下面我们进一步思考🤔

- React为什么要这么处理事件；
- 通过简易的实现，React到底是怎么实现的。

为什么使用合成事件

我们之前说的只是React的触发方式，实际上对于 合成事件 中 合成 并没有解释，这个也是打算是放在下一个篇章中结合源码进行讲解，我们可以把使用合成事件归结成下面两个目的

- 抹平浏览器差异，这里主要是针对不同浏览器对于事件的api的兼容，主要是万恶的IE的兼容，好在微软官方已经放弃它了。
- 使用事件委托，监听都在根节点进行，减少了内存开销。