

17 | 特别的事件系统：React 事件与 DOM 事件有何不同？

2020/12/07 修言



59.7M

00:00/22:53



看视频

相信不少小伙伴在进阶 React 的过程中都会或多或少地了解到这样一件事情：**React 有着自成一派的事件系统**，它和 DOM 原生事件系统不一样。但到底哪里不一样，却很少有人能够一五一十地说清楚。

开篇我们曾经说过，对于不同的知识，需要采取不同的学习策略。就 React 事件系统来说，它涉及的源码量不算小，相关逻辑也不够内聚，整体的理解成本相对较高，可能不少人都被劝退过。

幸运的是，无论是在面试场景下，还是在实际的开发中，React 事件相关的问题都更倾向于考验我们对事件工作流、事件特征等**逻辑层面问题**的理解，而非对源码细节的把握。而事件工作流、事件特征等逻辑层面的“主要矛盾”，正是我们本讲探讨的重点。

不管你曾经被 React 源码劝退过多少次，我想只要能好好把握住这一讲，拿下事件系统对你来说仍将是小菜一碟。所以说大家不要怕，跟着我的思路走就完了。

作为团队前端框架方向的负责人，我曾经在自研框架的初期，从 React 事件系统相关的设计思想中受益良多。在这一讲，我将基于自己对源码的理解，为你介绍 React 事件系统的工作逻辑。

注：本文逻辑提取自 React 16.13.x。随着 React 版本的更迭，事件系统的实现细节难免有调整，但其设计思想总是一脉相承的，你只要把握住核心逻辑即可。

回顾原生 DOM 下的事件流

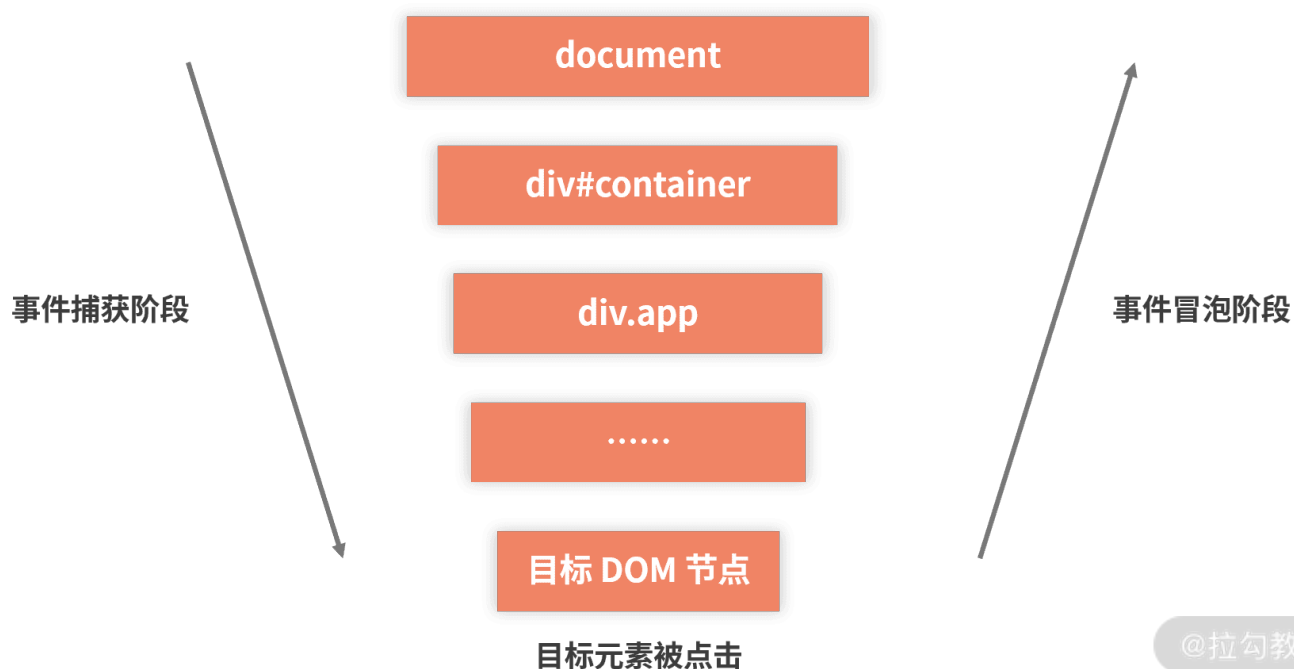
这些年在考察候选人的过程中，我发现了一件非常有趣的事情：一些同学提起前端框架时能够滔滔不绝，可说到 DOM 基础时却开始胡言乱语。这或许只有在当下这个前端发展阶段才会有的魔幻现实主义现象，但要想理解好 **React 事件机制**，就必须对原生 **DOM 事件流**有扎实的掌握。因此在文章的开篇，我们先来简单复习一下 DOM 事件流是如何工作的。

在浏览器中，我们通过事件监听来实现 JS 和 HTML 之间的交互。一个页面往往会被绑定许许多多的事件，而页面接收事件的顺序，就是**事件流**。

W3C 标准约定了一个事件的传播过程要经过以下 3 个阶段：

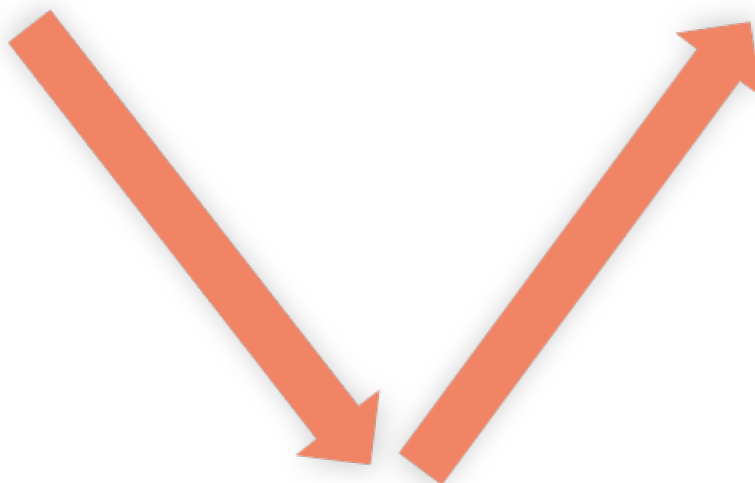
1. 事件捕获阶段
2. 目标阶段
3. 事件冒泡阶段

理解这个过程最好的方式就是读图了，下图是一棵 DOM 树的结构简图，图中的箭头就代表着事件的“穿梭”路径。



当事件被触发时，首先经历的是一个捕获过程：事件会从最外层的元素开始“穿梭”，逐层“穿梭”到最内层元素，这个过程会持续到事件抵达它目标的元素（也就是真正触发这个事件的元素）为止；此时事件流就切换到了“目标阶段”——事件被目标元素所接收；然后事件会被“回弹”，进入到冒泡阶段——它会沿着来时的路“逆流而上”，一层一层再走回去。

这个过程很像我们小时候玩的蹦床：从高处下落，触达蹦床后再弹起、回到高处，整个过程呈一个对称的“V”字形。



@拉勾教育

DOM 事件流下的性能优化思路：事件委托

在原生 DOM 中，事件委托（也叫事件代理）是一种重要的性能优化手段。这里我通过一道面试题，来快速地帮你回忆相关的知识。

请看下面这段代码：

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4.   <meta charset="UTF-8">
5.   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6.   <meta http-equiv="X-UA-Compatible" content="ie=edge">
7.   <title>Document</title>
8. </head>
9. <body>
10.  <ul id="poem">
11.    <li>床前明月光</li>
12.    <li>疑是地上霜</li>
13.    <li>举头望明月</li>
14.    <li>低头思故乡</li>
15.    <li>锄禾日当午</li>
16.    <li>汗滴禾下土</li>
17.    <li>谁知盘中餐</li>
18.    <li>粒粒皆辛苦</li>
19.    <li>背不动了</li>
20.    <li>我背不动了</li>
21.  </ul>
```

■ 复制代码

```
22. </body>
23. </html>
```

问：在这段 HTML 渲染出的界面里，我希望做到点击每一个 li 元素，都能输出它内在的文本内容。你会怎么做？

一个比较直观的思路是让每一个 li 元素都去监听一个点击动作，按照这个思路写出来的代码是这样的：

```
1. <script>
2.   // 获取 li 列表
3.   var liList = document.getElementsByTagName('li')
4.   // 逐个安装监听函数
5.   for (var i = 0; i < liList.length; i++) {
6.     liList[i].addEventListener('click', function (e) {
7.       console.log(e.target.innerHTML)
8.     })
9.   }
10. </script>
```

[复制代码](#)

我们当然可以像这样给 10 个 li 安装 10 次监听函数，但这样不仅累，开销也大。10 个监听函数做的还都是一模一样的事情，也不够优雅。怎么办呢？**事件冒泡**！

对于这 10 个 li 来说，无论点击动作发生在哪个 li 上，点击事件最终都会被冒泡到它们共同的父亲——ul 元素上去，所以我们完全可以让 ul 来帮忙感知这个点击事件。

既然 ul 可以帮忙感知事件，那它能不能帮忙处理事件呢？答案是能，因为我们有 **e.target**。ul 元素可以通过事件对象中的 target 属性，拿到实际触发事件的那个元素，针对这个元素分发事件处理的逻辑，做到真正的“委托”。

按照这个思路，我们就可以丢掉 for 循环来写代码了，以下是用事件代理来实现同样效果的代码：

```
1. var ul = document.getElementById('poem')
2. ul.addEventListener('click', function(e){
3.   console.log(e.target.innerHTML)
4. })
```

[复制代码](#)

这里再强调一下 e.target 这个属性，它指的是触发事件的具体目标，它记录着**事件的源头**。所以说，不管咱们的监听函数在哪一层执行，只要我拿到这个 e.target，就相当于拿到了真正触发事件的那个元素。拿到这个元素后，我们完全可以模拟出它的行为，实现无差别的监听效果。

像这样利用事件的冒泡特性，把多个子元素的同一类型的监听逻辑，合并到父元素上通过一个监听函数来管理的行为，就是**事件委托**。通过事件委托，我们可以减少内存开销、简化注册步骤，大大提高开发

效率。

这绝妙的事件委托，正是 React 合成事件的灵感源泉。

React 事件系统是如何工作的

React 的事件系统沿袭了事件委托的思想。在 React 中，除了少数特殊的不可冒泡的事件（比如媒体类型的事件）无法被事件系统处理外，绝大部分的事件都不会被绑定在具体的元素上，而是统一被绑定在页面的 document 上。当事件在具体的 DOM 节点上被触发后，最终都会冒泡到 document 上，document 上所绑定的统一事件处理程序会将事件分发到具体的组件实例。

在分发事件之前，React 首先会对事件进行包装，把原生 DOM 事件包装成合成事件。

认识 React 合成事件

合成事件是 React 自定义的事件对象，它符合 W3C 规范，在底层抹平了不同浏览器的差异，在上层面向开发者暴露统一的、稳定的、与 DOM 原生事件相同的事件接口。开发者们由此便不必再关注烦琐的兼容性问题，可以专注于业务逻辑的开发。

虽然合成事件并不是原生 DOM 事件，但它保存了原生 DOM 事件的引用。当你需要访问原生 DOM 事件对象时，可以通过合成事件对象的 `e.nativeEvent` 属性获取到它，如下图所示：



e.nativeEvent 将会输出 MouseEvent 这个原生事件，如下图所示：

原生 DOM 事件是

```
▼ MouseEvent {isTrusted: true, screenX: 106, screenY: 177, clientX: 47, clientY: 60, ...} ⓘ
  altKey: false
  bubbles: true
  button: 0
  buttons: 0
  cancelBubble: false
  cancelable: true
  clientX: 47
  clientY: 60
  composed: true
  ctrlKey: false
  currentTarget: null
  defaultPrevented: false
  detail: 1
  eventPhase: 0
  fromElement: null
  isTrusted: true
  layerX: 47
  layerY: 60
  metaKey: false
```

@拉勾教育

到这里，大家就对 React 事件系统的基本原理，包括合成事件的基本概念有了一定的了解。接下来，我们将在此基础上结合 React 源码和调用栈，对事件系统的工作流进行深入的拆解。

React 事件系统工作流拆解

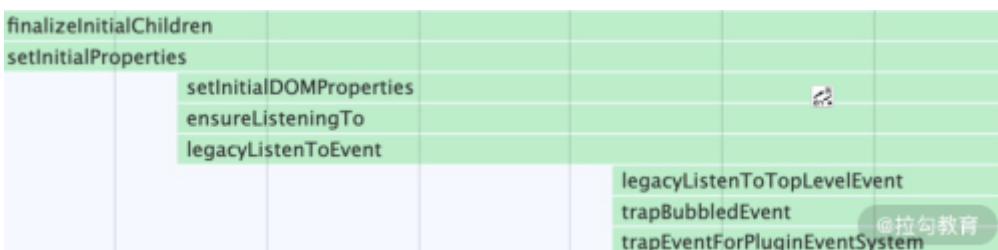
既然是事件系统，那就逃不出“事件绑定”和“事件触发”这两个关键动作。首先让我们一起来看看事件的绑定是如何实现的。

事件的绑定

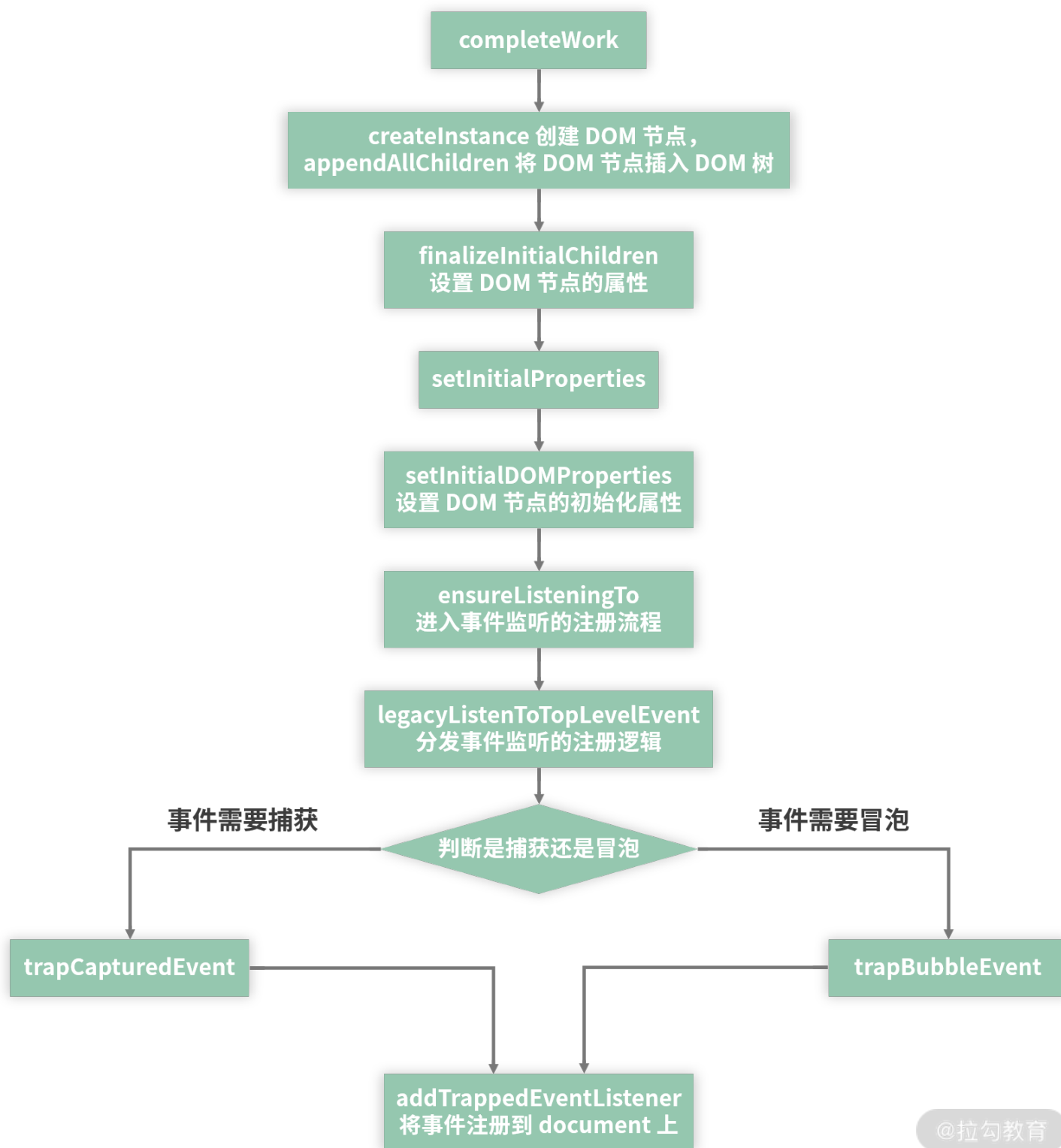
事件的绑定是在组件的挂载过程中完成的，具体来说，是在 **completeWork** 中完成的。关于 completeWork，我们已经在第 15 讲中学习过它的工作原理，这里需要你回忆起来的是 completeWork 中的以下三个动作：

completeWork 内部有三个关键动作：**创建 DOM 节点**（createInstance）、**将 DOM 节点插入到 DOM 树中**（appendAllChildren）、**为 DOM 节点设置属性**（finalizeInitialChildren）。

其中“为 DOM 节点**设置属性**”这个环节，会遍历 FiberNode 的 props key。当遍历到事件相关的 props 时，就会触发事件的注册链路。整个过程涉及的函数调用栈如下图所示：



这些函数之间是如何各司其职、打好“配合”的呢？请看下面这张工作流大图：



@拉勾教育

从图中可以看出，事件的注册过程是由 **ensureListeningTo** 函数开启的。在 **ensureListeningTo** 中，会尝试获取当前 DOM 结构中的根节点（这里指的是 **document** 对象），然后通过调用 **legacyListenToEvent**，将统一的事件监听函数注册到 **document** 上面。

在 `legacyListenToEvent` 中，实际上是通过调用 `legacyListenToTopLevelEvent` 来处理事件和 `document` 之间的关系的。`legacyListenToTopLevelEvent` 直译过来是“监听顶层的事件”，这里的“顶层”就可以理解为事件委托的最上层，也就是 `document` 节点。在 `legacyListenToTopLevelEvent` 中，有这样一段逻辑值得我们注意，请看下图：

```
function legacyListenToTopLevelEvent(topLevelType, mountAt, listenerMap) { top
  if (!listenerMap.has(topLevelType)) {
    switch (topLevelType) {
      case TOP_SCROLL:
        trapCapturedEvent(TOP_SCROLL, mountAt);
        break;

      case TOP_FOCUS:
      case TOP_BLUR:
        trapCapturedEvent(TOP_FOCUS, mountAt);
        trapCapturedEvent(TOP_BLUR, mountAt); // We set the flag for a single d
        // but this ensures we mark both as attached rather than just one.

        listenerMap.set(TOP_BLUR, null);
        listenerMap.set(TOP_FOCUS, null);
        break;

      case TOP_CANCEL:
      case TOP_CLOSE:
        if (isEventSupported(getRawEventName(topLevelType))) {
          trapCapturedEvent(topLevelType, mountAt);
        }

        break;
    }
  }
}
```

@拉勾教育

`listenerMap` 是在 `legacyListenToEvent` 里创建/获取的一个数据结构，它将记录当前 **document** 已经监听了哪些事件。在 `legacyListenToTopLevelEvent` 逻辑的起点，会首先判断 `listenerMap.has(topLevelType)` 这个条件是否为 `true`。

这里插播一个小的前置知识：`topLevelType` 在 `legacyListenToTopLevelEvent` 的函数上下文中代表事件的类型，比如说我尝试监听的是一个点击事件，那么 `topLevelType` 的值就会是 `click`，如下图所示：

```
}
}
function legacyListenToTopLevelEvent(topLevel
  if (!listenerMap.has(topLevelType)) {
    switch (topLevelType) {
      case TOP_SCROLL:
        ...
    }
  }
}
```

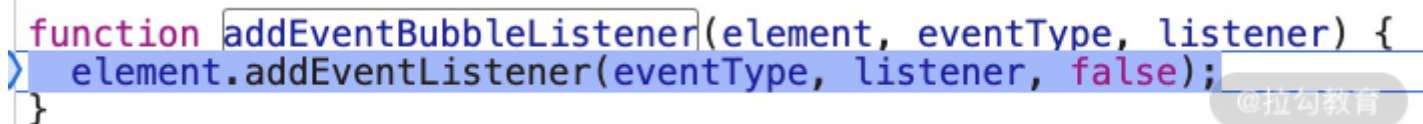
"click"

@拉勾教育

若事件系统识别到 `listenerMap.has(topLevelType)` 为 `true`，也就是当前这个事件 `document` 已经监听了，那么就会直接跳过对这个事件的处理，否则才会进入具体的事件监听逻辑。如此一来，即便我们在 **React** 项目中多次调用了对同一个事件的监听，也只会 **在 document 上触发一次注册**。

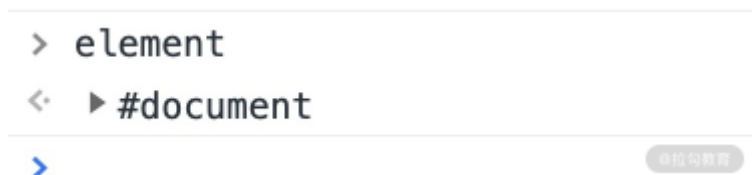
为什么针对同一个事件，即便可能会存在多个回调，**document** 也只需要注册一次监听？因为 React 最终注册到 **document** 上的并不是某一个 DOM 节点上对应的具体回调逻辑，而是一个统一的事件分发函数。这里我将断点打在事件监听函数的绑定动作上，请看下图：

```
function addEventBubbleListener(element, eventType, listener) {  
  element.addEventListener(eventType, listener, false);  
}
```



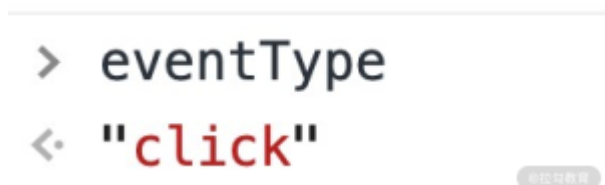
在这段逻辑中，**element** 就是 **document** 这个 DOM 元素，如下图所示，它在 **legacyListenToEvent** 阶段被获取后，又被层层逻辑传递到了这个位置。

```
> element  
  < ▶ #document  
  >
```



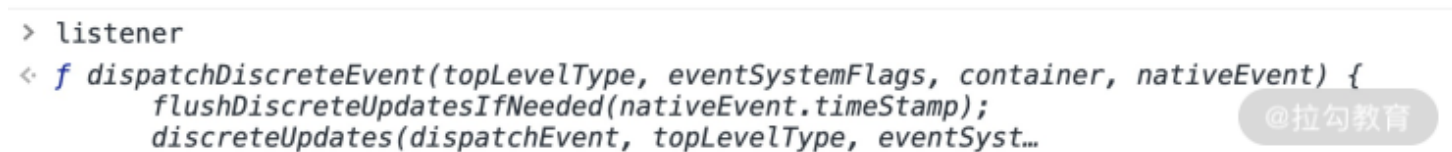
addEventListener 就更不用多说了，它是[原生 DOM 里专门用来注册事件监听器的接口](#)。我们真正需要关注的是图中这个函数的前两个入参，首先看 **eventType**，它表示事件的类型，这里我监听的是一个点击事件，因此 **eventType** 就是 **click**（见下图的运行时输出结果）。

```
> eventType  
  < "click"
```



重点在 **listener** 上，前面刚说过，最终注册到 **document** 上的是一个**统一的事件分发函数**，这个函数到底长啥样？我们来看看，以下是运行时的 **listener** 输出结果：

```
> listener  
  < f dispatchDiscreteEvent(topLevelType, eventSystemFlags, container, nativeEvent) {  
    flushDiscreteUpdatesIfNeeded(nativeEvent.timeStamp);  
    discreteUpdates(dispatchEvent, topLevelType, eventSyst...
```



可以看到，**listener** 本体是一个名为 **dispatchDiscreteEvent** 的函数。事实上，根据情况的不同，**listener** 可能是以下 3 个函数中的任意一个：

1. **dispatchDiscreteEvent**
2. **dispatchUserBlockingUpdate**
3. **dispatchEvent**

dispatchDiscreteEvent 和 dispatchUserBlockingUpdate 的不同，主要体现在对优先级的处理上，对事件分发动作倒没什么影响。无论是 dispatchDiscreteEvent 还是 dispatchUserBlockingUpdate，它们最后都是通过调用 dispatchEvent 来执行事件分发的。因此可以认为，最后绑定到 **document** 上的这个统一的事件分发函数，其实就是 **dispatchEvent**。

那么 dispatchEvent 是如何实现事件分发的呢？

事件的触发

事件触发的本质是对 dispatchEvent 函数的调用。由于 dispatchEvent 触发的调用链路较长，中间涉及的要素也过多，因此我们这里不再逐个跟踪函数的调用栈，直接来看核心工作流，请看下图：

事件触发，冒泡至 document

执行 dispatchEvent

创建事件对应的合成事件对象
(SyntheticEvent)

收集事件在**捕获阶段**所波及的
回调函数和对应的节点实例

收集事件在**冒泡阶段**所波及的
回调函数和对应的节点实例

将**前两步收集来的回调**按顺序执行
执行时 SyntheticEvent 会作为入参被传入每个回调

workflow 中前三步我们在前面都有所提及，对你来说相对难以理解的应该是 4、5、6 这三步，这三步也是我们接下来讲解的重点。

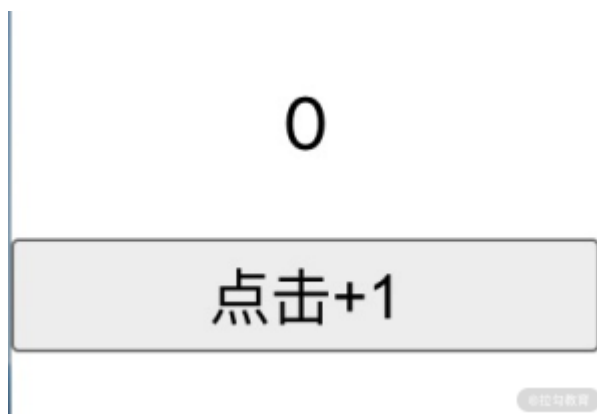
事件回调的收集与执行

我们借助一个 Demo 来理解这个过程，Demo 组件代码如下：

■ 复制代码

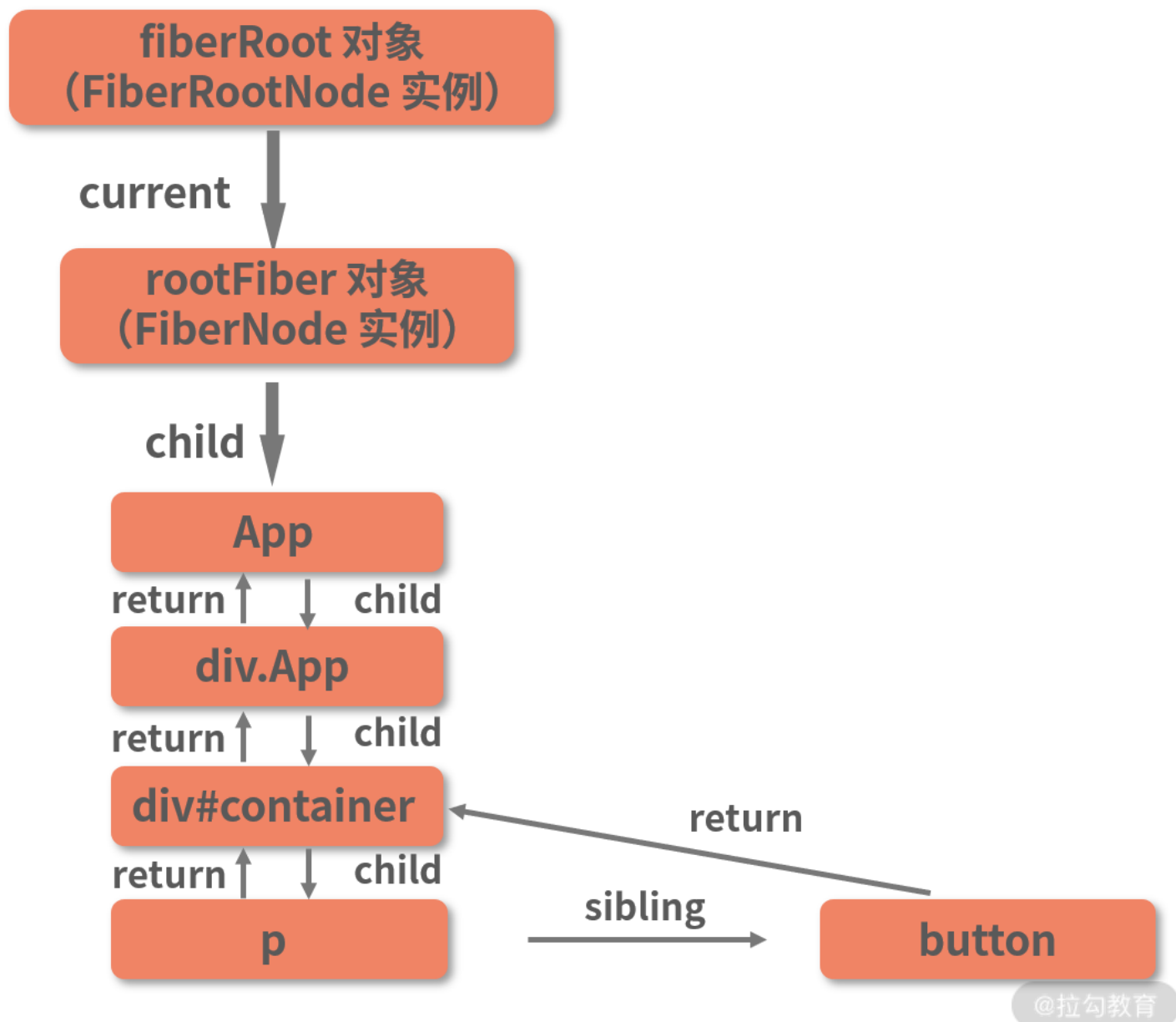
```
1. import React from 'react';
2. import { useState } from 'react'
3. function App() {
4.   const [state, setState] = useState(0)
5.   return (
6.     <div className="App">
7.       <div id="container" onClickCapture={() => console.log('捕获经过 div')} onClick={
8.         <p style={{ width: 128, textAlign: 'center' }}>
9.           {state}
10.        </p>
11.        <button style={{ width: 128 }} onClick={() => { setState(state + 1) }}>
12.        </div>
13.      </div>
14.    );
15.  }
16. export default App;
```

这个组件对应的界面如下图所示：



界面中渲染出来的是一行数字文本和一个按钮，每点击一下按钮，数字文本会 +1。在 JSX 结构中，监听点击事件的除了 button 按钮外，还有 id 为 container 的 div 元素，这个 div 元素同时监听了点击事件的冒泡和捕获。

App 组件对应的 Fiber 树结构如下图所示：



接下来我们借助这张 Fiber 树结构图来理解事件回调的收集过程。

首先我们来看收集过程对应的源码逻辑，这部分逻辑在 `traverseTwoPhase` 函数中，源码如下（解析在注释里）：

```
1. function traverseTwoPhase(inst, fn, arg) {
2.   // 定义一个 path 数组
3.   var path = [];
4.
5.   while (inst) {
6.     // 将当前节点收集进 path 数组
7.     path.push(inst);
8.     // 向上收集 tag===HostComponent 的父节点
9.     inst = getParent(inst);
10.  }
11.  var i;
```

[复制代码](#)

```
12. // 从后往前，收集 path 数组中会参与捕获过程的节点与对应回调
13. for (i = path.length; i-- > 0;) {
14.   fn(path[i], 'captured', arg);
15. }
16.
17. // 从前往后，收集 path 数组中会参与冒泡过程的节点与对应回调
18. for (i = 0; i < path.length; i++) {
19.   fn(path[i], 'bubbled', arg);
20. }
21. }
```

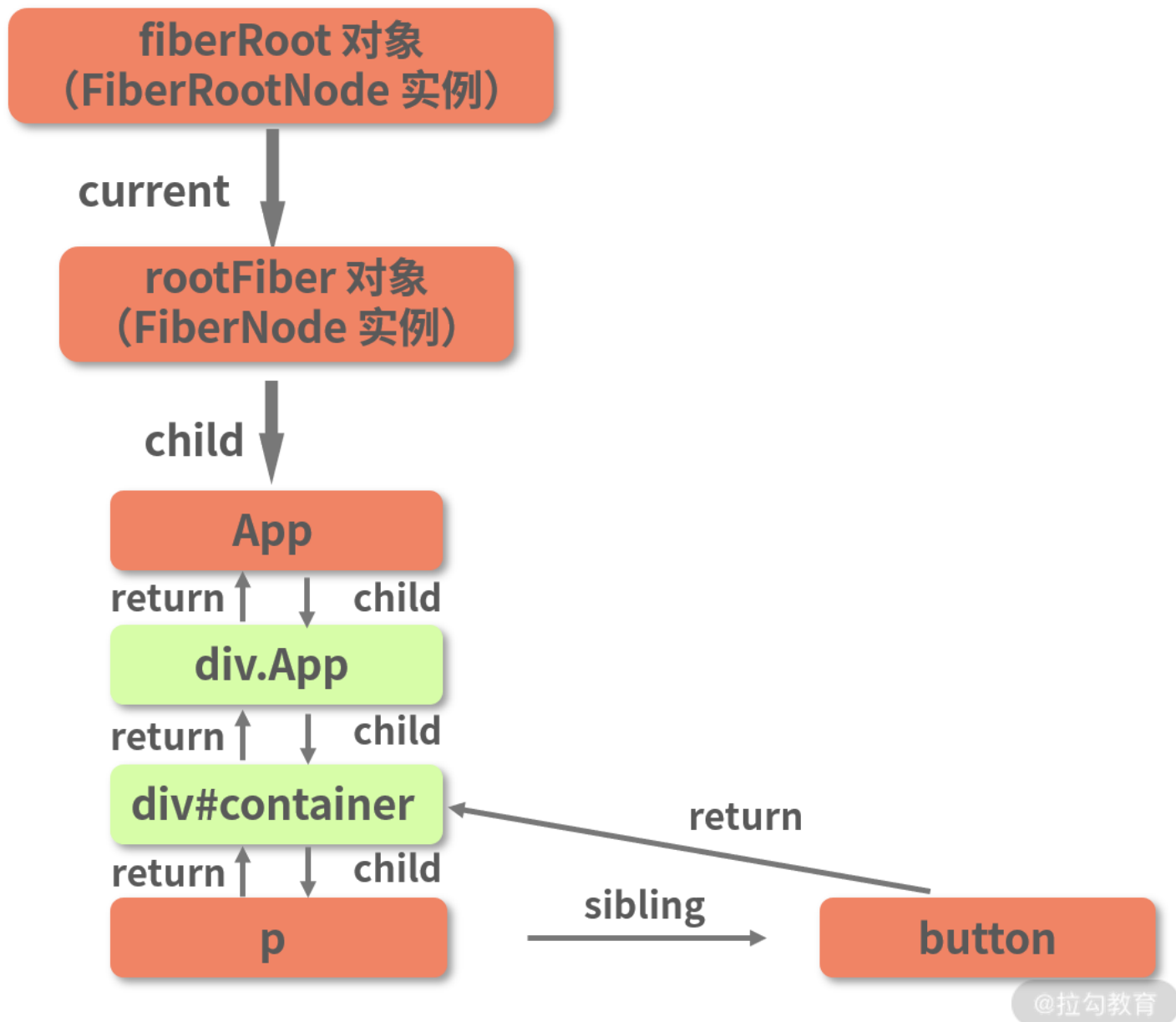
traverseTwoPhase 函数做了以下三件事情。

1. 循环收集符合条件的父节点，存进 path 数组中

traverseTwoPhase 会以当前节点（触发事件的目标节点）为起点，不断向上寻找 **tag===HostComponent** 的父节点，并将这些节点按顺序收集进 path 数组中。其中 tag===HostComponent 这个条件是在 getParent() 函数中管控的。

为什么一定要求 **tag===HostComponent** 呢？前面介绍渲染链路时，我们曾经讲过，**HostComponent** 是 DOM 元素对应的 Fiber 节点类型。此处限制 tag===HostComponent，也就是说只收集 DOM 元素对应的 Fiber 节点。之所以这样做，是因为浏览器只认识 DOM 节点，浏览器事件也只会 DOM 节点之间传播，收集其他节点是没有意义的。

将这个过程对应到 Demo 示例的 Fiber 树中来看，button 节点是事件触发的起点，在它的父节点中，符合 tag===HostComponent 这个条件的只有 div#container 和 div.App（即下图高亮处）。



因此最后收集上来的 path 数组内容就是 div#container、div.App 及 button 节点自身（button 节点别忘了，它是 while 循环的起点，一开始就会被推进 path 数组），如下图所示：

```
> path
< (3) [FiberNode, FiberNode, FiberNode]
  > 0: FiberNode {tag: 5, key: null, elementType: "button", type: "button", stateNode: button, ...}
  > 1: FiberNode {tag: 5, key: null, elementType: "div", type: "div", stateNode: div.container, ...}
  > 2: FiberNode {tag: 5, key: null, elementType: "div", type: "div", stateNode: div.App, ...}
  length: 3
  __proto__: Array(0)
```

2. 模拟事件在捕获阶段的传播顺序，收集捕获阶段相关的节点实例与回调函数

接下来，`traverseTwoPhase` 会从后往前遍历 `path` 数组，模拟事件的捕获顺序，收集事件在捕获阶段对应的回调与实例。

前面咱们说 path 数组是从子节点出发，向上收集得来的。所以说**path 数组中子节点在前，祖先节点在后**。

从后往前遍历 path 数组，其实就是从父节点往下遍历子节点，直至遍历到目标节点的过程，这个遍历顺序和事件在捕获阶段的传播顺序是一致的。在遍历的过程中，fn 函数会对每个节点的回调情况进行检查，若该节点上对应当前事件的捕获回调不为空，那么节点实例会被收集到合成事件的 `_dispatchInstances` 属性（也就是 `SyntheticEvent._dispatchInstances`）中去，事件回调则会被收集到合成事件的 `_dispatchListeners` 属性（也就是 `SyntheticEvent._dispatchListeners`）中去，等待后续的执行。

3. 模拟事件在冒泡阶段的传播顺序，收集冒泡阶段相关的节点实例与回调函数

捕获阶段的工作完成后，`traverseTwoPhase` 会从后往前遍历 path 数组，模拟事件的冒泡顺序，收集事件在捕获阶段对应的回调与实例。

这个过程和步骤 2 基本是一样的，唯一的区别是对 path 数组的倒序遍历变成了正序遍历。既然倒序遍历模拟的是捕获阶段的事件传播顺序，那么正序遍历自然模拟的就是冒泡阶段的事件传播顺序。在正序遍历的过程中，同样会对每个节点的回调情况进行检查，若该节点上对应当前事件的冒泡回调不为空，那么节点实例和事件回调同样会分别被收集到 `SyntheticEvent._dispatchInstances` 和 `SyntheticEvent._dispatchListeners` 中去。

需要注意的是，当前事件对应的 `SyntheticEvent` 实例有且仅有一个，因此在模拟捕获和模拟冒泡这两个过程中，收集到的实例会被推入同一个 `SyntheticEvent._dispatchInstances`，收集到的事件回调也会被推入同一个 `SyntheticEvent._dispatchListeners`。

这样一来，我们在事件回调的执行阶段，只需要按照顺序执行 `SyntheticEvent._dispatchListeners` 数组中的回调函数，就能够一口气模拟出整个完整的 DOM 事件流，也就是“捕获-目标-冒泡”这三个阶段。

接下来仍然是以 Demo 为例，我们来看看 button 上触发的点击事件对应的 `SyntheticEvent` 对象上的 `_dispatchInstances` 和 `_dispatchListeners` 各是什么内容，请看下图：

```
> event._dispatchInstances
< ▼ (3) [FiberNode, FiberNode, FiberNode] ⓘ
  ▶ 0: FiberNode {tag: 5, key: null, elementType: "div", type: "div", stateNode: div.container, ...}
  ▶ 1: FiberNode {tag: 5, key: null, elementType: "button", type: "button", stateNode: button, ...}
  ▶ 2: FiberNode {tag: 5, key: null, elementType: "div", type: "div", stateNode: div.container, ...}
    length: 3
    ▶ __proto__: Array(0)

> event._dispatchListeners
< ▼ (3) [f, f, f] ⓘ
  ▶ 0: () => console.log('捕获经过 div')
  ▶ 1: () => { setState(state + 1); }
  ▶ 2: () => console.log('冒泡经过 div')
    length: 3
    ▶ __proto__: Array(0)
```

@拉勾教育

可以看出，`_dispatchInstances` 和 `_dispatchListeners` 两个数组中的元素是严格的一一对应关系，这确保了在回调的执行阶段，我们可以简单地通过索引来将实例与监听函数关联起来，实现事件委托的效果。同时，两个数组中元素的排序，完美地契合了 DOM 标准中“捕获-目标-冒泡”这三个阶段的事件传播顺序，真是妙啊！

总结

本讲我们在回顾原生 DOM 事件流的基础上，对 React 事件系统的工作流进行了学习。行文至此，相信你已经对 React 事件机制的实现原理有了通透的理解，此时不妨尝试问自己一个问题：既然到头来不过是基于合成事件在模拟 DOM 事件流，React 为什么不直接使用原生 DOM 提供的事件机制呢？

或者换个问法：**React 事件系统的设计动机是什么？**

这里我结合个人的理解，给出两个思考的角度，希望能给你带来一些启发。

1. 首先一定要说的，也是 React 官方说明过的一点是：合成事件符合 [W3C](#) 规范，在底层抹平了不同浏览器的差异，在上层面向开发者暴露统一的、稳定的、与 DOM 原生事件相同的事件接口。开发者们由此便不必再关注烦琐的底层兼容问题，可以专注于业务逻辑的开发。
2. 此外，自研事件系统使 **React 牢牢把握住了事件处理的主动权**：这一点其实和我们平时造轮子是一样的。我在牵头自研团队前端框架之前，首先问自己的问题也是“为什么需要自研？React 不好用吗？Vue 不香吗？”。我们造轮子，很多时候并不是因为别人家的轮子不好，而是因为别人家的轮子没有办法 **Match** 我们的场景。拿 React 来说，举两个大家都比较熟悉的例子，比如说它想在事件系统中处理 Fiber 相关的优先级概念，或者想把多个事件揉成一个事件（比如 `onChange` 事件），原生 DOM 会帮它做吗？不会，因为原生讲究的就是个通用性。而 React 想要的则是“量体裁衣”，通过自研事件系统，React 能够从很大程度上干预事件的表现，使其符合自身的需求。

我在社区的一些讨论中，曾经见到过“合成事件性能更好”这样的结论，该结论的推导过程往往是这样的：事件委托可以节省内存开销 → React 合成事件承袭了事件委托的思想 → 合成事件性能更好。对于这类观点，个人目前持保留意见。

React 合成事件虽然承袭了事件委托的思想，但它的实现过程比传统的事件委托复杂太多。个人愚见，对 React 来说，事件委托主要的作用应该在于帮助 React **实现了对所有事件的中心化管控**。至于 React 事件是否比不使用事件委托的原生 DOM 事件性能更好？没有严格的对比和大量测试数据做支撑，我们很难下结论，React 官方也从没有给出过类似的说法。严谨起见，这里不推荐大家以性能为切入点去把握合成事件的特征。

关于 React 事件系统，就介绍到这里。从下一讲开始，我们将进入 Redux 的世界。