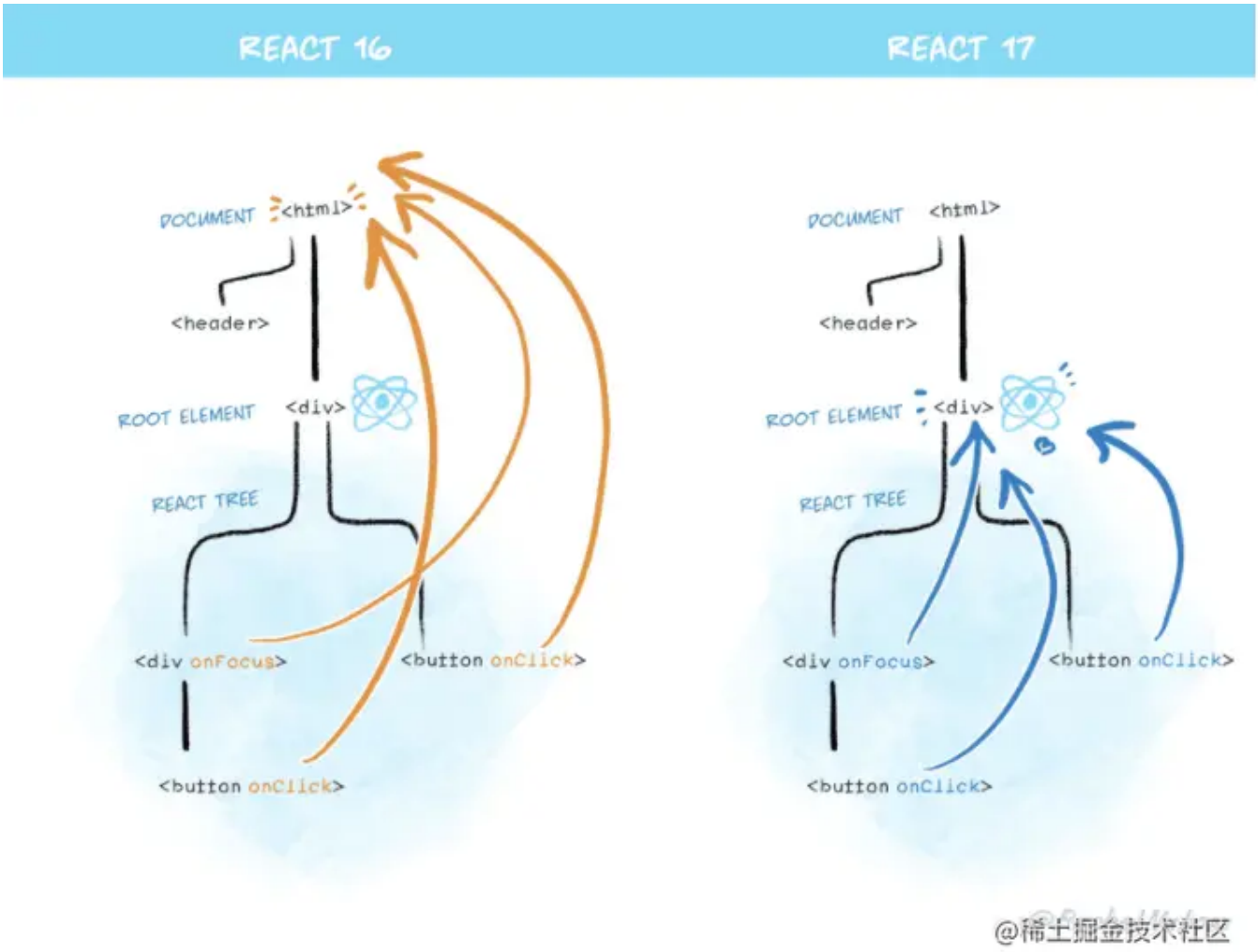
深度分析React源码中的合成事件

热身准备

明确几个概念

在 `React@17.0.3` 版本中：

- 所有事件都是委托在 `id = root` 的DOM元素中（网上很多说是在 `document` 中， `17` 版本不是了）；
- 在应用中所有节点的事件监听其实都是在 `id = root` 的DOM元素中触发；
- `React` 自身实现了一套事件冒泡捕获机制；
- `React` 实现了合成事件 `SyntheticEvent`；
- `React` 在 `17` 版本不再使用事件池了（网上很多说使用了对象池来管理合成事件对象的创建销毁，那是 `16` 版本及之前）；
- 事件一旦在 `id = root` 的DOM元素中委托，其实是一直在触发的，只是没有绑定对应的回调函数；



盗用一张官方图，按官方解释，之所以会将事件委托从 `document` 中移到 `id = root` 的DOM元素，是为了**可以更加安全地进行新旧版本 React 树的嵌套。**

感兴趣的可以访问：React中文网站 。

事件系统角色划分

- 事件注册： `registerEvents` ；
- 事件监听： `listenToAllSupportedEvents` ；
- 事件合成： `SyntheticBaseEvent` ；
- 事件派发： `dispatchEvent` ；

事件注册

事件注册是自执行的，也就是 `React` 自身进行调用的：

```javascript
// 注册React事件
registerSimpleEvents();
```

```
registerEvents$2();
registerEvents$1();
registerEvents$3();
registerEvents();
```

`React`事件就是在组件中调用的`onClick`这种写法的事件。上面分为5个函数写，主要是区分不同的事件注册逻辑，但是**最后都会添加到`allNativeEvents`的`Set`数据结构中。**

registerSimpleEvents

这里会注册大部分事件，它们在`React`被定义为顶级事件。

它们分为三类：

- 离散事件：`discreteEvent`，常见的如：`click, keyup, change`；
- 用户阻塞事件：`userBlocking`，常见的如：`dragEnter, mouseMove, scroll`；
- 连续事件：`continuous`，常见的如：`error, progress, load,` ； 它们的优先级排序：

0：离散事件， 1：用户阻塞事件， 2：连续事件

它们会注册冒泡和捕获阶段两个事件。

registerEvents$2

注册类似`onMouseEnter`， `onMouseLeave`单阶段事件，只注册冒泡阶段事件。

registerEvents$1

注册`onChange`相关事件，注册冒泡和捕获阶段两个事件。

registerEvents$3

注册`onSelect`相关事件，注册冒泡和捕获阶段两个事件。

registerEvents

注册`onBeforeInput`， `onCompositionUpdate`等相关事件，注册冒泡和捕获阶段两个事件。

事件监听

在React源码系列之二：React的渲染机制曾提到过，`React`在开始渲染前，会为应用创建一个`fiberRoot`作为应用的根节点。在创建`fiberRoot`还会做一件事，就是

```javascript 复制代码
listenToAllSupportedEvents(rootContainerElement);
```

从字面就能理解这个函数是做事件监听的，其中`rootContainerElement`参数就是应用中的`id = root`的DOM元素。

该函数主要遍历上面事件注册添加到`allNativeEvents`的事件，按照一定规则，区分冒泡阶段，捕获阶段，区分有无副作用进行监听，监听的api还是`addEventListener`：

```javascript 复制代码
// 监听冒泡阶段事件
function addEventBubbleListener(target, eventType, listener) {
  target.addEventListener(eventType, listener, false);
  return listener;
}
// 监听捕获阶段事件
function addEventCaptureListener(target, eventType, listener) {
  target.addEventListener(eventType, listener, true);
  return listener;
}
```

代码中的`target`就是`id = root`的DOM元素。

注意，上面监听的`listener`是一个事件派发器，并不是真实的浏览器事件或你写的事件回调函数。 不要搞混淆了。

事件派发

上面提到，事件一旦在`id = root`的DOM元素中委托，其实是一直在触发的，只是没有绑定对应的回调函数。

意思是，当我们把鼠标移入我们的应用页面中时，这时就在派发事件了，因为页面的DOM元素是有监听`mousemove`之类的事件的。

那问题来了，`React`是如何得知我们给事件绑定了回调函数并触发对应的回调函数的?

带着这个问题我们来研究下**事件派发。**

要讲事件派发，还得提下事件监听阶段监听的 `listener`，它实际是下面这玩意：

```javascript 复制代码
function createEventListenerWrapperWithPriority(targetContainer, domEventName, eventSystemFlags) {
  var eventPriority = getEventPriorityForPluginSystem(domEventName);
  var listenerWrapper;

  switch (eventPriority) {
    case DiscreteEvent:
      listenerWrapper = dispatchDiscreteEvent;
      break;

    case UserBlockingEvent:
      listenerWrapper = dispatchUserBlockingUpdate;
      break;

    case ContinuousEvent:
    default:
      listenerWrapper = dispatchEvent;
      break;
  }

  return listenerWrapper.bind(null, domEventName, eventSystemFlags, targetContainer);
}
```

和事件注册一样，`listener` 也分为 `dispatchDiscreteEvent, dispatchUserBlockingUpdate, dispatchEvent` 三种。它们之间的主要区别是执行优先级，还有 `discreteEvent` 涉及到要清除之前的 `discreteEvent` 问题，所以做了区分。但是它们最后都会调用 `dispatchEvent`。相关参考视频讲解：进入学习

所以事件派发的角色应该是 `dispatchEvent`

```javascript 复制代码
function dispatchEvent(domEventName, eventSystemFlags, targetContainer, nativeEvent) {

  var allowReplay = true;

  allowReplay = (eventSystemFlags & IS_CAPTURE_PHASE) === 0;
  // 如果有离散事件正在执行，会排队，顺序执行
  if (allowReplay && hasQueuedDiscreteEvents() && isReplayableDiscreteEvent(domEventName)) {
    domEventName, eventSystemFlags, targetContainer, nativeEvent);
    return;
  }
  // 尝试事件派发，如果成功，就不用执行下面的代码了
```

```
  var blockedOn = attemptToDispatchEvent(domEventName, eventSystemFlags, targetContainer, nativeEver
  // 尝试事件派发成功
  if (blockedOn === null) {
    if (allowReplay) {
      // 清除连续事件队列
      clearIfContinuousEvent(domEventName, nativeEvent);
    }

    return;
  }

  if (allowReplay) {
    if (isReplayableDiscreteEvent(domEventName)) {

      queueDiscreteEvent(blockedOn, domEventName, eventSystemFlags, targetContainer, nativeEvent);
      return;
    }

    if (queueIfContinuousEvent(blockedOn, domEventName, eventSystemFlags, targetContainer, nativeEve
      return;
    }

    clearIfContinuousEvent(domEventName, nativeEvent);
  }

  dispatchEventForPluginEventSystem(domEventName, eventSystemFlags, nativeEvent, null, targetContair
}
```

介绍下 `dispatchEvent` 的几个参数：

- `domEventName`： DOM事件名称，如：`click`，不是`onClick`；
- `eventSystemFlags`：事件系统标记；
- `targetContainer`：`id=root`的DOM元素；
- `nativeEvent`：原生事件（来自`addEventListener`）；

在`attemptToDispatchEvent`中， 根据`nativeEvent.target`找到真正触发事件的DOM元素，并根据DOM元素找到对应的`fiber`节点，判断`fiber`节点的**类型**以及**是否已渲染**来决定是否要派发事件。

在一系列判断通过后，就开始真正的事件处理了：

```
                                                                                javascript 复制代码
function dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent, targetInst, targetCor
  // 获取触发事件的DOM元素
  var nativeEventTarget = getEventTarget(nativeEvent);
```

```
  // 初始化事件派发队列
  var dispatchQueue = [];
  // 合成事件
  extractEvents$5(dispatchQueue, domEventName, targetInst, nativeEvent, nativeEventTarget, eventSyst
  // 按事件派发队列执行事件派发
  processDispatchQueue(dispatchQueue, eventSystemFlags);
}
```

在 `extractEvents$5` 中会进行事件合成，放在下面单独讲。

在 `processDispatchQueue` 会根据事件阶段（冒泡或捕获）来决定是正序还是倒序遍历合成事件中的 `listeners` 。

接下来就比较简单了。 遍历 `listeners` 执行上面的 `listener` 。

合成事件

在合成事件中，会根据 `domEventName` 来决定使用哪种类型的合成事件。

以 `click` 为例，当我们点击页面的某个元素时， `React` 会根据原生事件 `nativeEvent` 找到触发事件的DOM元素和对应的 `fiber` 节点。并以该节点为孩子节点往上查找，找到包括该节点及以上所有的 `click` 回调函数创建 `dispatchListener` ，并添加到 `listeners` 数组中。

arduino 复制代码

```
// dispatchListener
{
    instance: instance,  // 事件所在的fiber节点
    listener: listener,  // 事件回调函数
    currentTarget: currentTarget  // 事件对应的DOM元素
  }
```

当向上查找完成后，会基于 `click` 类型的合成事件类创建事件

csharp 复制代码

```
// 创建合成事件实例
var _event = new SyntheticEventCtor(reactName, reactEventType, null, nativeEvent, nativeEventTarget)
// 事件派发队列添加事件
dispatchQueue.push({
  event: _event,   // 合成事件实例
  listeners: _listeners  // 同类型事件的集合数组
});
```

看下`SyntheticEventCtor`

```javascript 复制代码
// Interface根据事件类型有所不同
function createSyntheticEvent(Interface) {
  // 合成事件构造函数
  function SyntheticBaseEvent(reactName, reactEventType, targetInst, nativeEvent, nativeEventTarget)
    this._reactName = reactName;
    this._targetInst = targetInst;
    this.type = reactEventType;
    this.nativeEvent = nativeEvent;
    this.target = nativeEventTarget;
    this.currentTarget = null;
    // React根据不同事件类型写了对应的属性接口，这里基于接口将原生事件上的属性clone到构造函数中
    for (var _propName in Interface) {... }

    var defaultPrevented = nativeEvent.defaultPrevented != null ? nativeEvent.defaultPrevented : nat

    if (defaultPrevented) {
      this.isDefaultPrevented = functionThatReturnsTrue;
    } else {
      this.isDefaultPrevented = functionThatReturnsFalse;
    }

    this.isPropagationStopped = functionThatReturnsFalse;
    return this;
  }

  _assign(SyntheticBaseEvent.prototype, {
    // 阻止默认事件
    preventDefault: function () {...},
    // 阻止捕获和冒泡阶段中当前事件的进一步传播
    stopPropagation: function () {...},
    // 合成事件不使用对象池了，这个事件是空的，没有意义，保存是为了向下兼容不报错。
    persist: function () {},

    isPersistent: functionThatReturnsTrue
  });

  return SyntheticBaseEvent;
}
```

看到这里，我们基本能弄明白合成事件是个什么东西了。

`React`合成事件是将同类型的事件找出来，基于这个类型的事件，`React`通过代码定义好的类型事件的接口和原生事件创建相应的合成事件实例，并重写了`preventDefault`和

`stopPropagation` 方法。

这样，**同类型的事件会复用同一个合成事件实例对象，节省了单独为每一个事件创建事件实例对象的开销，这就是事件的合成。**

捕获和冒泡

事件派发分为两个阶段执行， 捕获阶段和冒泡阶段。

在上面事件合成中讲过， `React` 会根据事件触发的 `fiber` 节点向上查找，将上面的同类型事件添加到队列中，这样天然就有了一个冒泡的顺序，从最底层向上冒泡。如果倒序过来遍历就是捕获的顺序。

所以， `React` 实现冒泡和捕获就很简单了，只需要根据事件派发的阶段，判断是冒泡阶段还是捕获阶段来决定是正序遍历 `listeners` 还是倒序遍历就行了。

总结

说是讲 `React` 的合成事件，实际上讲了 `React` 的事件系统。做下总结：

`React` 的事件系统分为几个部分：

- 事件注册；
- 事件监听；
- 事件合成；
- 事件派发； 事件系统流程：

1. 在 `React` 代码执行时，内部会自动执行事件的注册；
2. 第一次渲染，创建 `fiberRoot` 时，会进行事件的监听，所有的事件通过 `addEventListener` 委托在 `id=root` 的DOM元素上进行监听；
3. 在我们触发事件时，会进行事件合成，同类型事件复用一个合成事件类实例对象；
4. 最后进行事件的派发，执行我们代码中的事件回调函数；

当然，由于篇幅问题，这里也是对 `React` 事件系统的一个精简剖析，可能忽略了一些地方，欢迎指正。

看完这篇文章， 我们可以弄明白下面这几个问题：

1. `React` 事件委托在哪?
2. `React` 合成事件是什么?
3. `React` 合成事件是怎么实现的?
4. `React` 是怎么实现冒泡和捕获的?
5. `React` 合成事件是使用的原生事件吗?
6. `React` 事件系统分为哪几个部分?