react源码中的生命周期和事件系统

flyzz177 LV.4

2023年01月02日 11:34 · 阅读 1202

这一章我想跟大家探讨的是 `React` 的 `生命周期` 与 `事件系统` 。

jsx的编译结果

```
"use strict";

const App = () => {
  const [count, setCount] = useState(0);
  const handleValue = () => {
    setCount(1);
  }
  return (
    <>
      <div
        onclick={handleValue}
        className={'div'}
        CustomProperties={'i love React'}
        >
        a
      </div>
    </>
  )
}
```

```
"use strict";

var _jsxRuntime = require("react/jsx-runtime");

const App = () => {
  const [count, setCount] = useState(0);

  const handleValue = () => {
    setCount(1);
  };

  return /*#__PURE__*/(0, _jsxRuntime.jsx)(_jsxRuntime.Fragment, {
    children: /*#__PURE__*/(0, _jsxRuntime.jsx)("div", {
      onclick: handleValue,
      className: 'div',
      CustomProperties: 'i love React',
      children: "a"
    })
  });
};
```


因为前面也讲到 `jsx` 在 `v17` 中的编译结果，除了 `标签名` ，其他的挂在标签上的 `属性` （比如 `class` ）， `事件` （比如 `click` 事件），都是放在 `_jsxRuntime.jsx` 函数的第二参数上。表现为 `key:value` 的形式，这里我们就会产生几个问题。

- `react` 是怎么知道函数体（事件处理函数）是什么的呢?
- `react` 又是在什么阶段去处理这些事件的呢?

这里我们先卖个关子，我们先来看看一个完整的 `React` 应用的完整的生命周期是怎么样的，我们都知道 `React` 分为 `类组件` 与 `函数组件` ，两种组件的部分生命周期函数 `发生了一些变化` ，在这里我会分别对两种组件的生命周期做讲解。

React组件的生命周期

组件挂载的时候的执行顺序

因为在 `_jsxRuntime.jsx` 编译 `jsx` 对象的时候，我们会去做处理 `defaultProps` 和 `propType` 静态类型检查。所以这也算是一个生命周期吧。 `Class` 组件具有单独的 `constructor` ，在 `mount`

阶段会去执行这个构造函数，我曾经做了部分研究，这个 `constructor` 是类组件独有的，还是 `class` 独有的？后来发现这个 `constructor` 是 `class` 独有的，怎么理解这句话呢？

- 在《重学ES6》这本书中提到：`ES6` 中新增了类的概念，一个类必须要有 `constructor` 方法，如果在类中没有显示定义，则一个空的 `constructor` 方法会被默认添加。对于 `ReactClassComponent` 来讲需要 `constructor` 的作用就是用来初始化 `state` 和绑定事件，另外一点就是声明了 `constructor` ，就必须要调用 `super` ，我们一般用来接收 `props` 传递。假如你不写 `constructor` ，那就没法用 `props` 了，当然了要在 `constructor` 中使用 `props` ，也必须用 `super` 接收才行。
- 所以对于类组件来讲的话， `constructor` 也算是一个生命周期钩子。

`getDerivedStateFromProps` 会在调用 render 方法之前调用，并且在初始挂载及后续更新时都会被调用。它应返回一个对象来更新 state，如果返回 `null` 则不更新任何内容。

`render` 被调用时，它会检查 `this.props` 和 `this.state` 的变化并返回以下类型之一：

- **React 元素。**通常通过 JSX 创建。例如， `<div />` 会被 React 渲染为 DOM 节点， `<MyComponent />` 会被 React 渲染为自定义组件，无论是 `<div />` 还是 `<MyComponent />` 均为 React 元素。
- **数组或 fragments。** 使得 render 方法可以返回多个元素。
- **Portals。**可以渲染子节点到不同的 DOM 子树中。
- **字符串或数值类型。**它们在 DOM 中会被渲染为文本节点。
- **布尔类型或 `null` 。**什么都不渲染。（主要用于支持返回 `test && <Child />` 的模式，其中 test 为布尔类型。）

`componentDidMount()` 会在组件挂载后（插入 DOM 树中）立即调用。依赖于 DOM 节点的初始化应该放在这里。在这里适合去发送异步请求。

组件更新的时候的执行顺序

`getDerivedStateFromProps` => `shouldComponentUpdate()` => `render()` => `getSnapshotBeforeUpdate()` => `componentDidUpdate()`

- 其中 `shouldComponentUpdate` 也被称作为性能优化的一种钩子，其作用在于比较两次更新的 `state` 或 `props` 是否发生变化，决定是否更新当前组件，比较的方式是 `浅比较` ，以前讲过这里不再复述。
- 而 `getSnapshotBeforeUpdate` 函数在最近一次渲染输出（提交到 `DOM` 节点）之前调用。它使得组件能在发生更改之前从 `DOM` 中捕获一些信息。此生命周期方法的任何返回值将作为参

数传递给 `componentDidUpdate()`。

- `componentDidUpdate()` 会在更新后会被立即调用。首次渲染 `不会执行` 此方法。

组件卸载的时候执行顺序

`componentWillUnmount()` 会在组件 `卸载` 及 `销毁` 之前直接调用。在此方法中执行必要的清理操作，例如，清除 `timer`， `取消网络请求` 等等。

组件在发生错误的时候执行顺序

`getDerivedStateFromError` => `componentDidCatch` 关于这两个钩子，同学们可自行移步官网。

当然上面的只是 `ClassComponent` 的生命周期执行顺序，而在新版本的React中已经删除掉了 `componentDidMount`、`componentDidUpdate`、`componentWillUnMount`，取而代之的是 `useEffect`、`useLayoutEffect`。那究竟是谁代替了他们三个呢？这个问题我已经在React源码解析系列(八) -- 深入hooks的原理 中阐述过了，这里不再复述。

现在来回答第一个问题：**react是怎么知道函数体是什么的呢？** 这个问题其实问的非常好，`babel` 解析后的 `jsx` 本身只会去关注 `{事件名:函数名}`，但是每一个事件都是需要被注册、绑定的，然后通过事件触发，来执行绑定函数的函数体。解释这种问题还是得要去看一下源码里面的具体实现。

listenToAllSupportedEvents

我们在React源码解析系列(二) -- 初始化组件的创建更新流程中提到 `rootFiber` 与 `FiberRoot` 的创建，创建完毕之后我们就需要去创建事件，创建事件的入口函数为 `listenToAllSupportedEvents`。

```javascript 复制代码
// packages/react-dom/src/events/DOMPluginEventSystem.js
export function listenToAllSupportedEvents(rootContainerElement: EventTarget) {
  if (enableEagerRootListeners) { // enableEagerRootListeners默认值为false

    // listeningMarker就是一个随机数+字符串，作为唯一值
    if (rootContainerElement[listeningMarker]) {
      ...
      return;
    }
    rootContainerElement[listeningMarker] = true;
```

```
    // 遍历allNativeEvents的所有事件
    allNativeEvents.forEach(domEventName => {

      // 如果不是委托事件，没有冒泡阶段
      // nonDelegatedEvents全部媒体事件，
      if (!nonDelegatedEvents.has(domEventName)) {
        listenToNativeEvent(
          domEventName,
          false,
          ((rootContainerElement: any): Element),
          null,
        );
      }
      // 有冒泡阶段
      listenToNativeEvent(
        domEventName,
        true,
        ((rootContainerElement: any): Element),
        null,
      );
    });
  }
}

//listeningMarker
// 唯一标识
const listeningMarker =
  '_reactListening' +
  Math.random()
    .toString(36)
    .slice(2);
```

我们在这里必须要关注一下 `allNativeEvents` 是什么东西， `allNativeEvents` 在源码里体现为一个存储着事件名的 `Set` 结构：

```
export const allNativeEvents: Set<DOMEventName> = new Set();
```
javascript 复制代码

接下来看看 `listenToNativeEvent` 究竟干了些什么。

相关参考视频讲解：进入学习

listenToNativeEvent

javascript 复制代码

```javascript
export function listenToNativeEvent(
  domEventName: DOMEventName,// 事件名
  isCapturePhaseListener: boolean, // 根据上个函数，这里应该是确定是是能够冒泡的事件
  rootContainerElement: EventTarget,  targetElement: Element | null,  eventSystemFlags?: EventSystem
): void {

  let target = rootContainerElement;

  //如果是selectionchange事件，加到dom上
  if (
    domEventName === 'selectionchange' &&
    (rootContainerElement: any).nodeType !== DOCUMENT_NODE
  ) {
    target = (rootContainerElement: any).ownerDocument;
  }

  if (
    targetElement !== null &&
    !isCapturePhaseListener &&
    nonDelegatedEvents.has(domEventName) // 非冒泡事件
  ) {
   ...

    //滚动事件不冒泡
    if (domEventName !== 'scroll') {
      return;
    }

    eventSystemFlags |= IS_NON_DELEGATED; // is_non_delegated 不是委托事件
    target = targetElement;
  }
  //获取dom上绑定的事件名数组 Set[] ||
  const listenerSet = getEventListenerSet(target);
  // 处理事件名为捕获阶段与冒泡阶段 Set[click_bubble]
  const listenerSetKey = getListenerSetKey(
    domEventName,
    isCapturePhaseListener,
  );

  // 把没有打过的IS_CAPTURE_PHASE的符合条件的事件，打上标签
  if (!listenerSet.has(listenerSetKey)) {
    if (isCapturePhaseListener) {
      // 打上捕获的标签
      eventSystemFlags |= IS_CAPTURE_PHASE;
    }

    // 往节点上添加事件绑定
    addTrappedEventListener(
      target,
```

```
      domEventName,
      eventSystemFlags,
      isCapturePhaseListener,
    );

    // 往listenerSet中添加事件名
    listenerSet.add(listenerSetKey);
  }
}

//getEventListenerSet
export function getEventListenerSet(node: EventTarget): Set<string> {
  let elementListenerSet = (node: any)[internalEventHandlersKey];

  if (elementListenerSet === undefined) {
    // 创建一个Set来存放事件名
    elementListenerSet = (node: any)[internalEventHandlersKey] = new Set();
  }
  return elementListenerSet;
}

// getListenerSetKey
export function getListenerSetKey(
  domEventName: DOMEventName,  capture: boolean,
): string {
  // capture捕获，bubble冒泡
  return `${domEventName}__${capture ? 'capture' : 'bubble'}`;
}

// addTrappedEventListener
function addTrappedEventListener(
  targetContainer: EventTarget, // 容器
  domEventName: DOMEventName, // 事件名
  eventSystemFlags: EventSystemFlags, //事件名标识
  isCapturePhaseListener: boolean, // 事件委托
  isDeferredListenerForLegacyFBSupport?: boolean,
) {

  // 创建具有优先级的事件监听函数，返回值为function
  let listener = createEventListenerWrapperWithPriority(
    targetContainer,
    domEventName,
    eventSystemFlags,
  );

  ...

  targetContainer =
    enableLegacyFBSupport && isDeferredListenerForLegacyFBSupport
```

```
      ? (targetContainer: any).ownerDocument
      : targetContainer;

  let unsubscribeListener;
  ...

  // 区分捕获、冒泡 通过node.addEventListener绑定事件到节点上
  if (isCapturePhaseListener) {
    if (isPassiveListener !== undefined) {
      unsubscribeListener = addEventCaptureListenerWithPassiveFlag(
        targetContainer,
        domEventName,
        listener,
        isPassiveListener,
      );
    } else {
      unsubscribeListener = addEventCaptureListener(
        targetContainer,
        domEventName,
        listener,
      );
    }
  } else {
    if (isPassiveListener !== undefined) {
      unsubscribeListener = addEventBubbleListenerWithPassiveFlag(
        targetContainer,
        domEventName,
        listener,
        isPassiveListener,
      );
    } else {
      unsubscribeListener = addEventBubbleListener(
        targetContainer,
        domEventName,
        listener,
      );
    }
  }
}

// createEventListenerWrapperWithPriority
export function createEventListenerWrapperWithPriority(
  targetContainer: EventTarget, // 容器
  domEventName: DOMEventName, // 事件名
  eventSystemFlags: EventSystemFlags, //标识
): Function {

  // 获取事件Map里面已经标记好的优先级
  const eventPriority = getEventPriorityForPluginSystem(domEventName);
```

```
  let listenerWrapper;
  // 根据优先级不同绑定不同的执行函数
  switch (eventPriority) {
    //离散事件
    case DiscreteEvent:
      listenerWrapper = dispatchDiscreteEvent;
      break;
    // 用户交互阻塞渲染的事件
    case UserBlockingEvent:
      listenerWrapper = dispatchUserBlockingUpdate;
      break;
    // 其他事件
    case ContinuousEvent:
    // 默认事件
    default:
      listenerWrapper = dispatchEvent;
      break;
  }

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

在这里我们关注一下获取优先级 `getEventPriorityForPluginSystem` 这里，你会不会产生一个疑问， `React` 内部事件我们知道 `React` 本身一定会给优先级的，但是非 `React` 事件呢，比如 `原生事件` ，他们的优先级是怎么确定的呢？不要急，我们看一看就知道了。

getEventPriorityForPluginSystem

javascript 复制代码

```
export function getEventPriorityForPluginSystem(
  domEventName: DOMEventName,
): EventPriority {
  // 通过事件名获取优先级
  const priority = eventPriorities.get(domEventName);

  // ContinuousEvent为默认优先级
  return priority === undefined ? ContinuousEvent : priority;
}
```

```
//eventPriorities
const eventPriorities = new Map();
```

`eventPriorities`本身是一个Map结构，我们可以发现两个地方进行了`eventPriorities.set()`的操作。

```javascript 复制代码
// packages/react-dom/src/events/DOMEventProperties.js
function setEventPriorities(
  eventTypes: Array<DOMEventName>,  priority: EventPriority,
): void {
  for (let i = 0; i < eventTypes.length; i++) {
    // 往eventPriorities添加优先级
    eventPriorities.set(eventTypes[i], priority);
  }
}

//registerSimplePluginEventsAndSetTheirPriorities
function registerSimplePluginEventsAndSetTheirPriorities(
  eventTypes: Array<DOMEventName | string>,  priority: EventPriority,
): void {
  for (let i = 0; i < eventTypes.length; i += 2) {
    const topEvent = ((eventTypes[i]: any): DOMEventName);
    const event = ((eventTypes[i + 1]: any): string);
    const capitalizedEvent = event[0].toUpperCase() + event.slice(1);
    // 改变事件名 click => onClick
    const reactName = 'on' + capitalizedEvent;
    // 往eventPriorities添加优先级
    eventPriorities.set(topEvent, priority);
    topLevelEventsToReactNames.set(topEvent, reactName);

    // 注册捕获阶段，冒泡阶段的事件
    registerTwoPhaseEvent(reactName, [topEvent]);
  }
}
```

这就说明，在这两个函数里面已经做好了优先级的处理，那我们可以去看一下在哪里调用的这两个函数，我们发现在函数`registerSimpleEvents`中，执行了这两个函数，往`eventPriorities`里面添加优先级。

```javascript 复制代码
// packages/react-dom/src/events/DOMEventProperties.js
export function registerSimpleEvents() {
  // 处理离散事件优先级
  registerSimplePluginEventsAndSetTheirPriorities(
    discreteEventPairsForSimpleEventPlugin,
    DiscreteEvent,
```

```
  );
  // 处理用户阻塞事件优先级
  registerSimplePluginEventsAndSetTheirPriorities(
    userBlockingPairsForSimpleEventPlugin,
    UserBlockingEvent,
  );
  // 处理默认事件优先级
  registerSimplePluginEventsAndSetTheirPriorities(
    continuousPairsForSimpleEventPlugin,
    ContinuousEvent,
  );
  // 处理其他事件优先级
  setEventPriorities(otherDiscreteEvents, DiscreteEvent);
}
```

上述代码中可以看到有非常多的`Plugin`，代码如下：

javascript 复制代码

```
const discreteEventPairsForSimpleEventPlugin = [
  ('cancel': DOMEventName), 'cancel',
  ('click': DOMEventName), 'click',
  ('close': DOMEventName), 'close',
  ('contextmenu': DOMEventName), 'contextMenu',
  ('copy': DOMEventName), 'copy',
  ('cut': DOMEventName), 'cut',
  ('auxclick': DOMEventName), 'auxClick',
  ('dblclick': DOMEventName), 'doubleClick', // Careful!
  ('dragend': DOMEventName), 'dragEnd',
  ('dragstart': DOMEventName), 'dragStart',
  ('drop': DOMEventName), 'drop',
  ('focusin': DOMEventName), 'focus', // Careful!
  ('focusout': DOMEventName), 'blur', // Careful!
  ('input': DOMEventName), 'input',
  ('invalid': DOMEventName), 'invalid',
  ('keydown': DOMEventName), 'keyDown',
  ('keypress': DOMEventName), 'keyPress',
  ('keyup': DOMEventName), 'keyUp',
  ('mousedown': DOMEventName), 'mouseDown',
  ('mouseup': DOMEventName), 'mouseUp',
  ('paste': DOMEventName), 'paste',
  ('pause': DOMEventName), 'pause',
  ('play': DOMEventName), 'play',
  ('pointercancel': DOMEventName), 'pointerCancel',
  ('pointerdown': DOMEventName), 'pointerDown',
  ('pointerup': DOMEventName), 'pointerUp',
  ('ratechange': DOMEventName), 'rateChange',
  ('reset': DOMEventName), 'reset',
  ('seeked': DOMEventName), 'seeked',
```

```
  ('submit': DOMEventName), 'submit',
  ('touchcancel': DOMEventName), 'touchCancel',
  ('touchend': DOMEventName), 'touchEnd',
  ('touchstart': DOMEventName), 'touchStart',
  ('volumechange': DOMEventName), 'volumeChange',
];

const otherDiscreteEvents: Array<DOMEventName> = [
  'change',
  'selectionchange',
  'textInput',
  'compositionstart',
  'compositionend',
  'compositionupdate',
];

const userBlockingPairsForSimpleEventPlugin: Array<string | DOMEventName> = [
  ('drag': DOMEventName), 'drag',
  ('dragenter': DOMEventName), 'dragEnter',
  ('dragexit': DOMEventName), 'dragExit',
  ('dragleave': DOMEventName), 'dragLeave',
  ('dragover': DOMEventName), 'dragOver',
  ('mousemove': DOMEventName), 'mouseMove',
  ('mouseout': DOMEventName), 'mouseOut',
  ('mouseover': DOMEventName), 'mouseOver',
  ('pointermove': DOMEventName), 'pointerMove',
  ('pointerout': DOMEventName), 'pointerOut',
  ('pointerover': DOMEventName), 'pointerOver',
  ('scroll': DOMEventName), 'scroll',
  ('toggle': DOMEventName), 'toggle',
  ('touchmove': DOMEventName), 'touchMove',
  ('wheel': DOMEventName), 'wheel',
];

const continuousPairsForSimpleEventPlugin: Array<string | DOMEventName> = [
  ('abort': DOMEventName), 'abort',
  (ANIMATION_END: DOMEventName), 'animationEnd',
  (ANIMATION_ITERATION: DOMEventName), 'animationIteration',
  (ANIMATION_START: DOMEventName), 'animationStart',
  ('canplay': DOMEventName), 'canPlay',
  ('canplaythrough': DOMEventName), 'canPlayThrough',
  ('durationchange': DOMEventName), 'durationChange',
  ('emptied': DOMEventName), 'emptied',
  ('encrypted': DOMEventName), 'encrypted',
  ('ended': DOMEventName), 'ended',
  ('error': DOMEventName), 'error',
  ('gotpointercapture': DOMEventName), 'gotPointerCapture',
  ('load': DOMEventName), 'load',
  ('loadeddata': DOMEventName), 'loadedData',
```

```
  ('loadedmetadata': DOMEventName), 'loadedMetadata',
  ('loadstart': DOMEventName), 'loadStart',
  ('lostpointercapture': DOMEventName), 'lostPointerCapture',
  ('playing': DOMEventName), 'playing',
  ('progress': DOMEventName), 'progress',
  ('seeking': DOMEventName), 'seeking',
  ('stalled': DOMEventName), 'stalled',
  ('suspend': DOMEventName), 'suspend',
  ('timeupdate': DOMEventName), 'timeUpdate',
  (TRANSITION_END: DOMEventName), 'transitionEnd',
  ('waiting': DOMEventName), 'waiting',
];
```

而在`registerSimplePluginEventsAndSetTheirPriorities`函数里面，我们发现了注册事件`registerTwoPhaseEvent`，我们继续去深究一下，究竟是怎么注册的。

registerTwoPhaseEvent

javascript 复制代码

```
export function registerTwoPhaseEvent(
  registrationName: string, // 注册事件reactName
  dependencies: Array<DOMEventName>, // 依赖
): void {
  registerDirectEvent(registrationName, dependencies);
  registerDirectEvent(registrationName + 'Capture', dependencies);
}
```

registerDirectEvent

javascript 复制代码

```
// Mapping from registration name to event name
export const registrationNameDependencies = {};

export function registerDirectEvent(
  registrationName: string, //react事件名onClick
  dependencies: Array<DOMEventName>, // 依赖
) {
  ...

  // 以react事件名为key，dependencies为value的map对象
  registrationNameDependencies[registrationName] = dependencies;

  if (__DEV__) {
    ...
  }
```

```
  // 遍历依赖，把每一项加入到allNativeEvents中去
  for (let i = 0; i < dependencies.length; i++) {
    allNativeEvents.add(dependencies[i]);
  }
}
```

前面说 `allNativeEvents` 是一个存储事件名的 `Set` ，这里往里面添加 `事件名` ，就完成了 `事件注册` 。还没有完，上面说过了事件注册，与事件绑定，但是用户点击的时候，应该怎么去触发呢？上面的代码，在获取了优先级之后，每个事件会根据当前优先级生成一个 `listenerWrapper` ，这个 `listenerWrapper` 也就是对应的事件触发绑定的函数。 `dispatchDiscreteEvent` 、 `dispatchUserBlockingUpdate` 、 `dispatchEvent` 三个函数都通过bind执行，我们都知道bind绑定的函数，会返回一个新函数，并不会立即执行。所以我们也必须看看他的入参是什么。

- `this` ： `null`
- `argments` ： `domEventName` ：事件名， `eventSystemFlags` ：事件类型标记， `targetContainer` ：目标容器。

dispatchEvent

因为不管是 `dispatchDiscreteEvent` 、 `dispatchUserBlockingUpdate` 最后都会去执行 `dispatchEvent` ，所以我们可以看看他的实现。

```
javascript 复制代码
// packages/react-dom/src/events/ReactDOMEventListener.js
export function dispatchEvent(
  domEventName: DOMEventName, // 事件名
  eventSystemFlags: EventSystemFlags, // 事件类型标记
  targetContainer: EventTarget, // 目标容器
  nativeEvent: AnyNativeEvent, // native事件
): void {
  ...
  // 如果被阻塞了，尝试调度事件 并返回挂载的实例或者容器
  const blockedOn = attemptToDispatchEvent(
    domEventName,
    eventSystemFlags,
    targetContainer,
    nativeEvent,
  );

  if (blockedOn === null) {
    // We successfully dispatched this event.
    ...
```

```
    return;
  }

  ...

  // 调度事件，触发事件
  dispatchEventForPluginEventSystem(
    domEventName,
    eventSystemFlags,
    nativeEvent,
    null,
    targetContainer,
  );
}

// dispatchEventForPluginEventSystem
export function dispatchEventForPluginEventSystem(
  domEventName: DOMEventName,  eventSystemFlags: EventSystemFlags,  nativeEvent: AnyNativeEvent,  ta
): void {
  ...
  //批量更新事件
  batchedEventUpdates(() =>
    dispatchEventsForPlugins(
      domEventName,
      eventSystemFlags,
      nativeEvent,
      ancestorInst,
      targetContainer,
    ),
  );
}

// batchedEventUpdates
export function batchedEventUpdates(fn, a, b) {
  ...
  isBatchingEventUpdates = true;
  try {
    // fn : ()=>dispatchEventsForPlugins
    //(domEventName,eventSystemFlags,ativeEvent,ancestorInst,targetContainer,),
    // a: undefined
    // b: undefined
    return batchedEventUpdatesImpl(fn, a, b);
    // batchedEventUpdatesImpl(fn, a, b) =>
    // Defaults
    // let batchedUpdatesImpl = function(fn, bookkeeping) {
    // return fn(bookkeeping); 执行dispatchEventsForPlugins
};
  } finally {
    ...
```

```
  }
}
```

dispatchEventsForPlugins

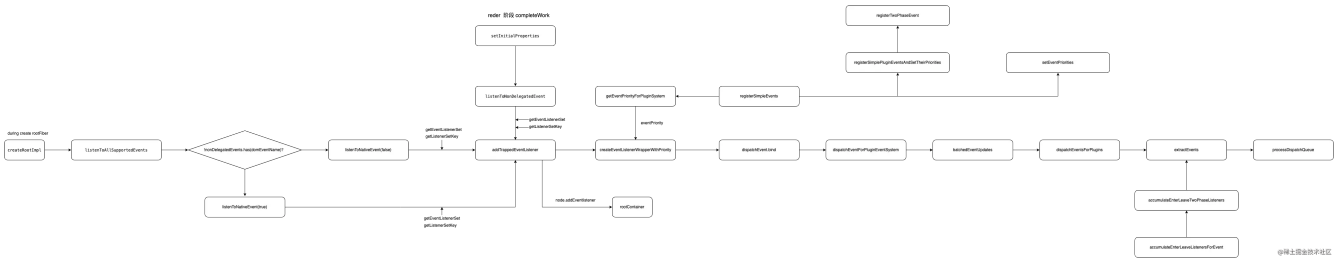
```javascript 复制代码
function dispatchEventsForPlugins(
  domEventName: DOMEventName,  eventSystemFlags: EventSystemFlags,  nativeEvent: AnyNativeEvent,  ta
): void {
  const nativeEventTarget = getEventTarget(nativeEvent);
  const dispatchQueue: DispatchQueue = [];
  //创建合成事件，遍历fiber链表，将会触发的事件加入到dispatchQueue中
  extractEvents(
    dispatchQueue,
    domEventName,
    targetInst,
    nativeEvent,
    nativeEventTarget,
    eventSystemFlags,
    targetContainer,
  );
  //触发时间队列，执行事件
  processDispatchQueue(dispatchQueue, eventSystemFlags);
}

//extractEvents
function extractEvents(
  dispatchQueue: DispatchQueue,  domEventName: DOMEventName,  targetInst: null | Fiber,  nativeEvent
) {
  ...
  let from;
  let to;
  ...

  const leave = new SyntheticEventCtor(
    leaveEventType,
    eventTypePrefix + 'leave',
    from,
    nativeEvent,
    nativeEventTarget,
  );
  leave.target = fromNode;
  leave.relatedTarget = toNode;

  let enter: KnownReactSyntheticEvent | null = null;
  ...
  accumulateEnterLeaveTwoPhaseListeners(dispatchQueue, leave, enter, from, to);
```

```
}

//accumulateEnterLeaveTwoPhaseListeners
export function accumulateEnterLeaveTwoPhaseListeners(
  dispatchQueue: DispatchQueue,  leaveEvent: KnownReactSyntheticEvent,  enterEvent: null | KnownReac
): void {
  const common = from && to ? getLowestCommonAncestor(from, to) : null;

  if (from !== null) {
    accumulateEnterLeaveListenersForEvent(
      dispatchQueue,
      leaveEvent,
      from,
      common,
      false,
    );
  }
  if (to !== null && enterEvent !== null) {
    accumulateEnterLeaveListenersForEvent(
      dispatchQueue,
      enterEvent,
      to,
      common,
      true,
    );
  }
}

// accumulateEnterLeaveListenersForEvent
function accumulateEnterLeaveListenersForEvent(
  dispatchQueue: DispatchQueue,  event: KnownReactSyntheticEvent,  target: Fiber,  common: Fiber | r
): void {
  // 获取注册的事件名
  const registrationName = event._reactName;
  // 事件处理函数容器
  const listeners: Array<DispatchListener> = [];
  //节点实例
  let instance = target;

  // 遍历fiber，获取fiber上的事件对应的事件处理函数
  while (instance !== null) {

    if (instance === common) {
      break;
    }

    const {alternate, stateNode, tag} = instance;
    if (alternate !== null && alternate === common) {
      break;
```

```
    }

    if (tag === HostComponent && stateNode !== null) {
      const currentTarget = stateNode;

      // 根据捕获阶段，还是冒泡阶段处理不同的函数逻辑
      if (inCapturePhase) {
        const captureListener = getListener(instance, registrationName);
        if (captureListener != null) {

          // 加入到listeners中
          // instance:当前fiebr实例
          // currentTarget:当前dom
          listeners.unshift(
            createDispatchListener(instance, captureListener, currentTarget),
          );
        }
      } else if (!inCapturePhase) {
        // 冒泡
        const bubbleListener = getListener(instance, registrationName);
        if (bubbleListener != null) {
          // 加入到listeners中
          listeners.push(
            createDispatchListener(instance, bubbleListener, currentTarget),
          );
        }
      }
    }
    // 当前fiber实例的父级
    instance = instance.return;
  }
  if (listeners.length !== 0) {
    // 把事件、事件处理函数全部推到dispatchQueue中
    dispatchQueue.push({event, listeners});
  }
}
// processDispatchQueue
export function processDispatchQueue(
  dispatchQueue: DispatchQueue, // 事件队列
  eventSystemFlags: EventSystemFlags, // 事件类型标记
): void {
  const inCapturePhase = (eventSystemFlags & IS_CAPTURE_PHASE) !== 0;

  for (let i = 0; i < dispatchQueue.length; i++) {
    const {event, listeners} = dispatchQueue[i];
    // 执行事件，完成触发
    processDispatchQueueItemsInOrder(event, listeners, inCapturePhase);
    //  event system doesn't use pooling.
  }
```

```
    // This would be a good time to rethrow if any of the event handlers threw.
    rethrowCaughtError();
  }
```

所以到这里，`React` 的事件系统就解析完了，在这里上面的问题就很好解答了，`React` 对事件名与事件处理函数对做了绑定，并在 `创建rootFiber` 的时候就做了 `事件注册`、`事件绑定`、`事件调度`。那么他们的执行流程大致如下：



总结

这一章主要是介绍组件在 `mount`、`update`、`destroy` 阶段的生命周期执行顺序与 `React` 事件系统的注册，绑定，调度更新等