link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=137 sentences=409, words=4459

1. DOM事件

1.1 事件

- 事件是用户或浏览器自身执行的某种动作，而响应某个事件的函数叫做事件处理程序

1.2 DOM事件流

- 事件流包含三个阶段
    - 事件捕获阶段
    - 处于目标阶段
    - 事件冒泡阶段
- 首先发生的是事件捕获，然后是实际的目标接收到事件，最后阶段是冒泡阶段

1.2.1 事件捕获

- 是先由最上一级的节点先接收事件,然后向下传播到具体的节点 `document->body->div->button`

1.2.2 目标阶段

- 在目标节点上触发,称为目标阶段
- 事件目标是真正触发事件的对象

```
let target = event.target || event.srcElement;
```

1.2.3 事件冒泡

- 事件开始时由最具体的元素(文档中嵌套层次最深的那个节点)接收,然后逐级向上传播 `button->div->body->document`

1.3 addEventListener

- 任何发生在W3C事件模型中的事件，首是进入捕获阶段，直到达到目标元素，再进入冒泡阶段
- 可以选择是在捕获阶段还是冒泡阶段绑定事件处理函数
- `useCapture`参数是 `true`，则在捕获阶段绑定函数，反之 `false`，在冒泡阶段绑定函数

```
element.addEventListener(event, function, useCapture)
```

1.4 阻止冒泡

- 如果想要阻止事件的传播
    - 在微软的模型中你必须设置事件的 `cancelBubble`的属性为**true**
    - 在W3C模型中你必须调用事件的 `stopPropagation()`方法

```
function stopPropagation(event) {
    if (!event) {
        window.event.cancelBubble = true;
    }
    if (event.stopPropagation) {
        event.stopPropagation();
    }
}
```

1.5 阻止默认行为

- 取消默认事件

```
function preventDefault(event) {
    if (!event) {
        window.event.returnValue  = false;
    }
    if (event.preventDefault) {
        event.preventDefault();
    }
}
```

1.6 事件代理

- 事件代理又称之为事件委托
- 事件代理是把原本需要绑定在 `&#x5B50;&#x5143;&#x7D20;`的事件委托给 `&#x7236;&#x5143;&#x7D20;`，让父元素负责事件监听
- 事件代理是利用 `&#x4E8B;&#x4EF6;&#x5192;&#x6CE1;`来实现的
- 优点
    - 可以大量节省内存占用，减少事件注册
    - 当新增子对象时无需再次对其绑定

```
<body>
    <ul id="list" onclick="show(event)">
        <li>item 1li>
        <li>item 2li>
        <li>item 3li>
        <li>item nli>
    ul>
    <script>
        function show(event){
          alert(event.target.innerHTML);
        }
    script>
body>
```

2.React事件系统

- 合成事件是围绕浏览器原生事件充当跨浏览器包装器的对象,它们将不同浏览器的行为合并为一个 API,这样做是为了确保事件在不同浏览器中显示一致的属性

2.1 面试题

- 为什么不能使用 `return false`来阻止事件默认行为?
- 请说一下React合成事件的工作原理?
- 可以给函数组件绑定事件吗?为什么?
- ......

2.2 使用

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';
class App extends React.Component {
  parentRef=React.createRef();
  childRef=React.createRef();
  componentDidMount() {
    this.parentRef.current.addEventListener("click", () => {
      console.log("父元素原生捕获");
    },true);
    this.parentRef.current.addEventListener("click", () => {
      console.log("父元素原生冒泡");
    });
    this.childRef.current.addEventListener("click", () => {
      console.log("子元素原生捕获");
    },true);
    this.childRef.current.addEventListener("click", () => {
      console.log("子元素原生冒泡");
    });
    document.addEventListener('click',()=>{
        console.log("document原生捕获");
    },true);
    document.addEventListener('click',()=>{
        console.log("document原生冒泡");
    });
  }
  parentBubble = () => {
    console.log("父元素React事件冒泡");
  };
  childBubble = () => {
    console.log("子元素React事件冒泡");
  };
  parentCapture = () => {
    console.log("父元素React事件捕获");
  };
  childCapture = () => {
    console.log("子元素React事件捕获");
  };
  render() {
    return (
      <div ref={this.parentRef} onClick={this.parentBubble} onClickCapture={this.parentCapture}>
        <p ref={this.childRef} onClick={this.childBubble} onClickCapture={this.childCapture}>
          事件执行顺序
        p>
      div>
    );
  }
}
ReactDOM.render(<App />, document.getElementById('root'));
```

2.3 简易实现

```
<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>eventtitle>
head>
<body>
    <div id="root">
        <div id="parent">
            <div id="child">
                点击
            div>
        div>
    div>
    <script>
        let root = document.getElementById('root');
        let parent = document.getElementById('parent');
        let child = document.getElementById('child');

        root.addEventListener('click',event=>dispatchEvent(event,true),true);

        root.addEventListener('click',event=>dispatchEvent(event,false),false);
        function dispatchEvent(event,isCapture){

            let paths = [];
            let currentTarget = event.target;
            while(currentTarget){
                paths.push(currentTarget);
                currentTarget=currentTarget.parentNode;
            }
            if(isCapture){
                for(let i=paths.length-1;i>=0;i--){
                    let handler = paths[i].onClickCapture;
                    handler&&handler();
                }
            }else{
                for(let i=0;ilet handler = paths[i].onClick;
                    handler&&handler();
                }
            }

        }
        root.addEventListener('click',event=>console.log('根元素原生事件捕获'),true);
        root.addEventListener('click',event=>console.log('根元素原生事件冒泡'),false);
        parent.addEventListener('click',()=>{
         console.log('父元素原生事件捕获');
        },true);
        parent.addEventListener('click',()=>{
         console.log('父元素原生事件冒泡');
        },false);
        child.addEventListener('click',()=>{
         console.log('子元素原生事件捕获');
        },true);
        child.addEventListener('click',()=>{
         console.log('子元素原生事件冒泡');
        },false);
        parent.onClick = ()=>{
            console.log('React:父元素React事件冒泡');
        }
        parent.onClickCapture = ()=>{
            console.log('React:父元素React事件捕获');
        }
        child.onClick = ()=>{
            console.log('React:子元素React事件冒泡');
        }
        child.onClickCapture = ()=>{
            console.log('React:子元素React事件捕获');
        }
    script>
body>
html>
```

2.4 源码实现

- 事件注册
- 事件绑定
- 事件触发

3. 初次渲染

3.1 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
let rootContainerElement = document.getElementById('root');
const handleDivClick = (event)=>{
  console.log('handleDivClick');
}
const handleDivClickCapture = (event)=>{
  console.log('handleDivClickCapture');

}
const handleButtonClick = (event)=>{
  console.log('handleButtonClick');
}
const handleButtonClickCapture = (event)=>{
  console.log('handleButtonClickCapture');
}
let element = (
  <div onClick={handleDivClick} onClickCapture={handleDivClickCapture}>
    <button onClick={handleButtonClick} onClickCapture={handleButtonClickCapture}>+button>
  div>
)
console.log(element);
ReactDOM.render(
  element,
  rootContainerElement
);
```

3.2 react.js

src\react.js

```
function createElement(type, config, children) {
    delete config.__source;
    delete config.__self;
    delete config.ref;
    delete config.key;
    let props = { ...config };
    if (arguments.length > 3) {
        props.children = Array.prototype.slice.call(arguments, 2);
    } else {
        props.children = children;
    }
    return {
        type,
        props
    }
}
const React = {
    createElement
};
export default React;
```

3.3 react-dom.js

src\react-dom.js

```
function render(vdom, container) {
    mount(vdom, container);
}
export function mount(vdom, container) {
    let newDOM = createDOM(vdom, container);
    container.appendChild(newDOM)
}
export function createDOM(vdom, parentDOM) {
    let { type, props } = vdom;
    let dom;
    if (typeof vdom === 'string' || typeof vdom === 'number') {
        dom = document.createTextNode(vdom);
    } else {
        dom = document.createElement(type);
    }
    if (props) {
        updateProps(dom, {}, props);
        if (Array.isArray(props.children)) {
            reconcileChildren(props.children, dom);
        } else if (props.children) {
            mount(props.children, dom);
        }
    }
    return dom;
}

function updateProps(dom, oldProps, newProps) {
    for (let key in newProps) {
        if (key === 'children') { continue; }
        if (key === 'style') {
            let style = newProps[key];
            for (let attr in style) {
                dom.style[attr] = style[attr]
            }
        } else {
            dom[key] = newProps[key];
        }
    }
}
function reconcileChildren(childrenVdom, parentDOM) {
    for (let i = 0; i < childrenVdom.length; i++) {
        let childVdom = childrenVdom[i];
        mount(childVdom, parentDOM);
    }
}
const ReactDOM = {
    render
};
export default ReactDOM;
```

4. 事件注册

4.1 react-dom.js

src\react-dom.js

```
+import { listenToAllSupportedEvents } from './DOMPluginEventSystem';
function render(vdom, container) {
+   listenToAllSupportedEvents(container);
    mount(vdom, container);
}
export function mount(vdom, container) {
    let newDOM = createDOM(vdom, container);
    container.appendChild(newDOM)
}
export function createDOM(vdom, parentDOM) {
    let { type, props } = vdom;
    let dom;
    if (typeof vdom
        dom = document.createTextNode(vdom);
    } else {
        dom = document.createElement(type);
    }
    if (props) {
        updateProps(dom, {}, props);
        if (Array.isArray(props.children)) {
            reconcileChildren(props.children, dom);
        } else if (props.children) {
            mount(props.children, dom);
        }
    }
    return dom;
}

function updateProps(dom, oldProps, newProps) {
    for (let key in newProps) {
        if (key
        if (key
            let style = newProps[key];
            for (let attr in style) {
                dom.style[attr] = style[attr]
            }
        } else {
            dom[key] = newProps[key];
        }
    }
}
function reconcileChildren(childrenVdom, parentDOM) {
    for (let i = 0; i < childrenVdom.length; i++) {
        let childVdom = childrenVdom[i];
        mount(childVdom, parentDOM);
    }
}
const ReactDOM = {
    render
};
export default ReactDOM;
```

4.2 EventRegistry.js

src\EventRegistry.js

```
export const allNativeEvents = new Set();

export function registerTwoPhaseEvent(registrationName, dependencies) {
    registerDirectEvent(registrationName, dependencies);
    registerDirectEvent(registrationName + 'Capture', dependencies);
}

export function registerDirectEvent(registrationName,dependencies,) {

    for (let i = 0; i < dependencies.length; i++) {
        allNativeEvents.add(dependencies[i]);
    }
}
```

4.3 DOMEventProperties.js

src\DOMEventProperties.js

```
import { registerTwoPhaseEvent } from './EventRegistry';

const discreteEventPairsForSimpleEventPlugin = [
    'click', 'click',
    'dblclick', 'doubleClick'];

export const topLevelEventsToReactNames = new Map();

export function registerSimpleEvents() {
    for (let i = 0; i < discreteEventPairsForSimpleEventPlugin.length; i += 2) {
        const topEvent = discreteEventPairsForSimpleEventPlugin[i];
        const event = discreteEventPairsForSimpleEventPlugin[i + 1];
        const capitalizedEvent = event[0].toUpperCase() + event.slice(1);
        const reactName = 'on' + capitalizedEvent;
        topLevelEventsToReactNames.set(topEvent, reactName);
        registerTwoPhaseEvent(reactName, [topEvent]);
    }
}
```

4.4 SimpleEventPlugin.js

src\SimpleEventPlugin.js

```
import { registerSimpleEvents } from './DOMEventProperties';
export { registerSimpleEvents as registerEvents };
```

4.5 DOMPluginEventSystem.js

src\DOMPluginEventSystem.js

```
import { allNativeEvents } from './EventRegistry';
import * as SimpleEventPlugin from './SimpleEventPlugin';

SimpleEventPlugin.registerEvents();
export function listenToAllSupportedEvents(rootContainerElement) {
    allNativeEvents.forEach(domEventName => {
       console.log('dom事件名',domEventName);
    });
}
```

5. 事件绑定

5.1 DOMPluginEventSystem.js

src\DOMPluginEventSystem.js

```
import { allNativeEvents } from './EventRegistry';
import * as SimpleEventPlugin from './SimpleEventPlugin';
+import { addEventCaptureListener, addEventBubbleListener } from './EventListener';
+import { getEventListenerSet } from './ReactDOMComponentTree';
+import { IS_CAPTURE_PHASE } from './EventSystemFlags';
+import { dispatchEvent } from './ReactDOMEventListener';
//注册插件 其实就是收集原生事件名称
SimpleEventPlugin.registerEvents();
+export const nonDelegatedEvents = new Set(['scroll'])
export function listenToAllSupportedEvents(rootContainerElement) {
    allNativeEvents.forEach(domEventName => {
+       //注册冒泡阶段
+       if (!nonDelegatedEvents.has(domEventName)) {
+           listenToNativeEvent(
+               domEventName,//click
+               false,//isCapturePhaseListener=false
+               rootContainerElement
+           );
+       }
+       //注册捕获阶段
+       listenToNativeEvent(
+           domEventName,//click
+           true,//isCapturePhaseListener=false
+           rootContainerElement//容器DOM元素
+       );
    });
}
+/**
+ *
+ * @param {*} domEventName DOM事件 click
+ * @param {*} isCapturePhaseListener 是否是捕获事件监听
+ * @param {*} rootContainerElement 根容器
+ * @param {*} targetElement 目标元素
+ * @param {*} eventSystemFlags 事件系统标志
+ */
+ export function listenToNativeEvent(domEventName, isCapturePhaseListener,rootContainerElement, eventSystemFlags = 0) {
+    const listenerSet = getEventListenerSet(rootContainerElement);//[]
+    //click__bubble click__capture
+    const listenerSetKey = getListenerSetKey(domEventName, isCapturePhaseListener);
+    if (!listenerSet.has(listenerSetKey)) {
+        if (isCapturePhaseListener) {
+            eventSystemFlags |= IS_CAPTURE_PHASE;//4
+        }
+        addTrappedEventListener(
+            rootContainerElement,
+            domEventName,
+            eventSystemFlags,
+            isCapturePhaseListener
+        );
+        listenerSet.add(listenerSetKey);
+    }
+}
+/**
+ * 根据事件名称和是否捕获阶段得到监听的key
+ * @param {*} domEventName 事件名称 click
+ * @param {*} capture  是否是捕获阶段
+ * @returns 监听事件的key
+ */
+ export function getListenerSetKey(domEventName, capture) {
+    return `${domEventName}__${capture ? 'capture' : 'bubble'}`;
+}
+/**
+ * 注册监听函数
+ * @param {*} targetContainer 绑定的目标容器
+ * @param {*} domEventName  事件名称
+ * @param {*} eventSystemFlags  事件系统标识
+ * @param {*} isCapturePhaseListener  是否是捕获阶段
+ */
+ function addTrappedEventListener(targetContainer, domEventName, eventSystemFlags, isCapturePhaseListener) {
+    let listener = dispatchEvent.bind(null, domEventName, eventSystemFlags, targetContainer);
+    if (isCapturePhaseListener) {
+        addEventCaptureListener(targetContainer, domEventName, listener);
+    } else {
+        addEventBubbleListener(targetContainer, domEventName, listener);
+    }
+}
```

5.2 EventSystemFlags.js

src\EventSystemFlags.js

```
export const IS_CAPTURE_PHASE = 1 << 2;
```

5.3 EventListener.js

src\EventListener.js

```
export function addEventCaptureListener(target,eventType,listener) {
    target.addEventListener(eventType, listener, true);
    return listener;
}

export function addEventBubbleListener(target, eventType, listener) {
    target.addEventListener(eventType, listener, false);
    return listener;
}
```

5.4 ReactDOMComponentTree.js

src\ReactDOMComponentTree.js

```
const randomKey = Math.random().toString(36).slice(2);
export const internalEventHandlersKey = '__reactEvents{{content}}#x27; + randomKey;
export function getEventListenerSet(node) {
  let elementListenerSet = node[internalEventHandlersKey];
  if (elementListenerSet === undefined) {
    elementListenerSet = node[internalEventHandlersKey] = new Set();
  }
  return elementListenerSet;
}
```

5.5 ReactDOMEventListener.js

src\ReactDOMEventListener.js

```

export function dispatchEvent(domEventName,eventSystemFlags,targetContainer,nativeEvent) {
    const nativeEventTarget = nativeEvent.target;
    console.log('domEventName',domEventName,'eventSystemFlags',eventSystemFlags,'nativeEventTarget',nativeEventTarget);
}
```

6 事件触发

6.1 构建fiber树

6.1.1 react-dom.js

src\react-dom.js

```
import { listenToAllSupportedEvents } from './DOMPluginEventSystem';
+import { internalInstanceKey, internalPropsKey } from './ReactDOMComponentTree';
+import { HostComponent } from './ReactWorkTags';
function render(vdom, container) {
    listenToAllSupportedEvents(container);
    mount(vdom, container);
}
export function mount(vdom, container) {
    let newDOM = createDOM(vdom, container);
    container.appendChild(newDOM)
}
export function createDOM(vdom, parentDOM) {
    let { type, props } = vdom;
    let dom;
    if (typeof vdom
        dom = document.createTextNode(vdom);
    } else {
        dom = document.createElement(type);
    }
+   let returnFiber = parentDOM[internalInstanceKey] || null;
+   let fiber = { tag: HostComponent, type, stateNode: dom, return: returnFiber };
+   dom[internalInstanceKey] = fiber;
+   dom[internalPropsKey] = props;
    if (props) {
        updateProps(dom, {}, props);
        if (Array.isArray(props.children)) {
            reconcileChildren(props.children, dom);
        } else if (props.children) {
            mount(props.children, dom);
        }
    }
    return dom;
}

function updateProps(dom, oldProps, newProps) {
    for (let key in newProps) {
        if (key
        if (key
            let style = newProps[key];
            for (let attr in style) {
                dom.style[attr] = style[attr]
            }
        } else {
            dom[key] = newProps[key];
        }
    }
}
function reconcileChildren(childrenVdom, parentDOM) {
    for (let i = 0; i < childrenVdom.length; i++) {
        let childVdom = childrenVdom[i];
        mount(childVdom, parentDOM);
    }
}
const ReactDOM = {
    render
};
export default ReactDOM;
```

src\ReactWorkTags.js

```
export const HostComponent = 5;
```

6.1.3 ReactDOMComponentTree.js

src\ReactDOMComponentTree.js

```
const randomKey = Math.random().toString(36).slice(2);
export const internalEventHandlersKey = '__reactEvents{{content}}#x27; + randomKey;//dom上的事件绑定集合
+export const internalInstanceKey = '__reactFiber{{content}}#x27; + randomKey;//dom上的fiber节点
+export const internalPropsKey = '__reactProps{{content}}#x27; + randomKey;//dom上的属性对象
export function getEventListenerSet(node) {
  let elementListenerSet = node[internalEventHandlersKey];
  if (elementListenerSet
    elementListenerSet = node[internalEventHandlersKey] = new Set();
  }
  return elementListenerSet;
}
+/**
+ * 根据DOM元素获取对应的fiber对象
+ * @param {*} targetNode DOM元素
+ * @returns fiber对象
+ */
+ export function getClosestInstanceFromNode(targetNode) {
+  let targetInst = targetNode[internalInstanceKey];
+  if (targetInst) {
+    return targetInst;
+  }
+}
+export function getFiberCurrentPropsFromNode(node) {
+  return node[internalPropsKey] || null;
+}
```

6.1.4 ReactDOMEventListener.js

src\ReactDOMEventListener.js

```
+import {getClosestInstanceFromNode,getFiberCurrentPropsFromNode} from './ReactDOMComponentTree';

/**
 * 事件处理函数
 * @param {*} domEventName 事件名 click
 * @param {*} eventSystemFlags 4 事件系统标志
 * @param {*} targetContainer 代理的容器 div#root
 * @param {*} nativeEvent 原生的事件对象 MouseEvent
 */
export function dispatchEvent(domEventName,eventSystemFlags,targetContainer,nativeEvent) {
+    const nativeEventTarget = nativeEvent.target;
+    //获得来源对应的fiber对象
+    const targetInst = getClosestInstanceFromNode(nativeEventTarget);
+    console.log('targetInst',targetInst);
+   //获得来源对应的fiber的属性对象
+    const props = getFiberCurrentPropsFromNode(nativeEventTarget);
+    console.log('props',props);
}
```

6.2 收集事件函数

6.2.1 ReactDOMEventListener.js

src\ReactDOMEventListener.js

```
import {getClosestInstanceFromNode,getFiberCurrentPropsFromNode} from './ReactDOMComponentTree';
+import {dispatchEventsForPlugins} from './DOMPluginEventSystem';
/**
 * 事件处理函数
 * @param {*} domEventName 事件名 click
 * @param {*} eventSystemFlags 4 事件系统标志
 * @param {*} targetContainer 代理的容器 div#root
 * @param {*} nativeEvent 原生的事件对象 MouseEvent
 */
export function dispatchEvent(domEventName,eventSystemFlags,targetContainer,nativeEvent) {
    const nativeEventTarget = nativeEvent.target;
    //获得来源对应的fiber对象
    const targetInst = getClosestInstanceFromNode(nativeEventTarget);
    //console.log('targetInst',targetInst);
    //获得来源对应的fiber对象
    //const props = getFiberCurrentPropsFromNode(nativeEventTarget);
    //console.log('props',props);
+   dispatchEventsForPlugins(
+       domEventName,
+       eventSystemFlags,
+       nativeEvent,
+       targetInst,
+       targetContainer
+   );
}
```

6.2.2 DOMPluginEventSystem.js

src\DOMPluginEventSystem.js

```
import { allNativeEvents } from './EventRegistry';
import * as SimpleEventPlugin from './SimpleEventPlugin';
import { addEventCaptureListener, addEventBubbleListener } from './EventListener';
import { getEventListenerSet } from './ReactDOMComponentTree';
import { IS_CAPTURE_PHASE } from './EventSystemFlags';
import { dispatchEvent } from './ReactDOMEventListener';
+import { HostComponent } from './ReactWorkTags';
+import getListener from './getListener';
//注册插件 其实就是收集原生事件名称
SimpleEventPlugin.registerEvents();
export const nonDelegatedEvents = new Set(['scroll'])
export function listenToAllSupportedEvents(rootContainerElement) {
    allNativeEvents.forEach(domEventName => {
```

```
        //注册冒泡阶段
        if (!nonDelegatedEvents.has(domEventName)) {
            listenToNativeEvent(
                domEventName,//click
                false,//isCapturePhaseListener=false
                rootContainerElement
            );
        }
        //注册捕获阶段
        listenToNativeEvent(
            domEventName,//click
            true,//isCapturePhaseListener=false
            rootContainerElement//容器DOM元素
        );
    });
}
/**
 *
 * @param {*} domEventName DOM事件 click
 * @param {*} isCapturePhaseListener 是否是捕获事件监听
 * @param {*} rootContainerElement 根容器
 * @param {*} targetElement 目标元素
 * @param {*} eventSystemFlags 事件系统标志
 */
export function listenToNativeEvent(domEventName, isCapturePhaseListener, rootContainerElement, eventSystemFlags = 0) {
    const listenerSet = getEventListenerSet(rootContainerElement);//[]
    //click__bubble click__capture
    const listenerSetKey = getListenerSetKey(domEventName, isCapturePhaseListener);
    if (!listenerSet.has(listenerSetKey)) {
        if (isCapturePhaseListener) {
            eventSystemFlags |= IS_CAPTURE_PHASE;//4
        }
        addTrappedEventListener(
            rootContainerElement,
            domEventName,
            eventSystemFlags,
            isCapturePhaseListener
        );
        listenerSet.add(listenerSetKey);
    }
}
/**
 * 根据事件名称和是否捕获阶段得到监听的key
 * @param {*} domEventName 事件名称 click
 * @param {*} capture  是否是捕获阶段
 * @returns 监听事件的key
 */
export function getListenerSetKey(domEventName, capture) {
    return `${domEventName}__${capture ? 'capture' : 'bubble'}`;
}
/**
 * 注册监听函数
 * @param {*} targetContainer 绑定的目标容器
 * @param {*} domEventName  事件名称
 * @param {*} eventSystemFlags  事件系统标识
 * @param {*} isCapturePhaseListener  是否是捕获阶段
 */
function addTrappedEventListener(targetContainer, domEventName, eventSystemFlags, isCapturePhaseListener) {
    let listener = dispatchEvent.bind(null, domEventName, eventSystemFlags, targetContainer);
    if (isCapturePhaseListener) {
        addEventCaptureListener(targetContainer, domEventName, listener);
    } else {
        addEventBubbleListener(targetContainer, domEventName, listener);
    }
}

+/**
+ * 派发事件
+ * @param {*} domEventName 事件名称 event
+ * @param {*} eventSystemFlags 事件标志，0或者4
+ * @param {*} nativeEvent 原生事件对象
+ * @param {*} targetInst fiber实例
+ * @param {*} targetContainer 目标容器
+ */
+export function dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent, targetInst, targetContainer) {
+    const nativeEventTarget = nativeEvent.target;
+    //事件处理函数数组
+    const dispatchQueue = [];
+    //提取监听事件
+    SimpleEventPlugin.extractEvents(
+        dispatchQueue,
+        domEventName,
+        targetInst,
+        nativeEvent,
+        nativeEventTarget,
+        eventSystemFlags,
+        targetContainer
+    );
+    console.log('dispatchQueue',dispatchQueue);
+}
+
+/**
+ * 收集一个阶段的监听
+ * @param {*} targetFiber 对应的fiber {tag:5,type:'button'}
+ * @param {*} reactName 事件名 onClick
+ * @param {*} nativeEventType 原生事件名 click
+ * @param {*} inCapturePhase 是否捕获阶段
+ * @returns
+ */
+export function accumulateSinglePhaseListeners(targetFiber, reactName, nativeEventType, +inCapturePhase) {
+    const captureName = reactName + 'Capture';//onClickCapture
+    //onClick或+onClickCapture
+    const reactEventName = inCapturePhase ? captureName : reactName;
+    const listeners = [];//所有的监听函数
```

```
+    let instance = targetFiber;//当前的fiber
+    let lastHostComponent = null;//上一个原生DOM元素
+    //从当前向上出发，收集所有的Dispatch
+    while (instance !== null) {
+        const { stateNode, tag } = instance;
+        if (tag === HostComponent && stateNode !== null) {
+            lastHostComponent = stateNode;
+            if (reactEventName !== null) {
+                const listener = getListener(instance, reactEventName);
+                if (listener != null) {
+                    listeners.push(createDispatchListener(instance, listener, lastHostComponent));
+                }
+            }
+        }
+        instance = instance.return;
+    }
+    return listeners;
+}
+ /**
+ * 创建Dispatch
+ * @param {*} instance fiber实例
+ * @param {*} listener 监听函数
+ * @param {*} currentTarget 当前的DOM事件对象
+ * @returns Dispatch
+ */
+ function createDispatchListener(instance, listener, currentTarget) {
+    return { instance, listener, currentTarget };
+}
```

6.2.3 SimpleEventPlugin.js

src\SimpleEventPlugin.js

```
+import { registerSimpleEvents ,topLevelEventsToReactNames} from './DOMEventProperties';
+import { IS_CAPTURE_PHASE } from './EventSystemFlags';
+import { SyntheticMouseEvent } from './SyntheticEvent';
+import { accumulateSinglePhaseListeners } from './DOMPluginEventSystem';
+/**
+ * 提取事件处理函数
+ * @param {*} dispatchQueue 队列
+ * @param {*} domEventName 事件名称 click
+ * @param {*} targetInst fiber实例
+ * @param {*} nativeEvent 原生事件
+ * @param {*} nativeEventTarget 原生事件对象
+ * @param {*} eventSystemFlags 事件标志
+ */
+ function extractEvents(dispatchQueue,domEventName,targetInst,nativeEvent,nativeEventTarget,eventSystemFlags) {
+    const reactName = topLevelEventsToReactNames.get(domEventName);//click=>onClick
+    let SyntheticEventCtor;
+    let reactEventType = domEventName;//click
+    switch (domEventName) {
+        case 'click':
+            SyntheticEventCtor = SyntheticMouseEvent;
+            break;
+        default:
+            break;
+    }
+    const inCapturePhase = (eventSystemFlags & IS_CAPTURE_PHASE) !== 0;
+    const listeners = accumulateSinglePhaseListeners(targetInst,reactName,nativeEvent.type,inCapturePhase);
+    if (listeners.length > 0) {
+        const event = new SyntheticEventCtor(
+            reactName,
+            reactEventType,
+            targetInst,
+            nativeEvent,
+            nativeEventTarget
+        );
+        dispatchQueue.push({ event, listeners });
+    }
+}
+
+export { registerSimpleEvents as registerEvents ,extractEvents};
```

6.2.4 getListener.js

src\getListener.js

```
import {getFiberCurrentPropsFromNode} from './ReactDOMComponentTree';

export default function getListener(inst,registrationName) {
    const stateNode = inst.stateNode;
    const props = getFiberCurrentPropsFromNode(stateNode);
    const listener = props[registrationName];
    return listener;
}
```

6.2.5 SyntheticEvent.js

src\SyntheticEvent.js

```
function functionThatReturnsTrue() {
  return true;
}
function functionThatReturnsFalse() {
  return false;
}

function createSyntheticEvent(Interface) {
  function SyntheticBaseEvent(reactName, reactEventType, targetInst, nativeEvent, nativeEventTarget) {
    this._reactName = reactName;
    this.type = reactEventType;
    this._targetInst = targetInst;
    this.nativeEvent = nativeEvent;
    this.target = nativeEventTarget;
    this.currentTarget = null;
    for (const propName in Interface) {
      this[propName] = nativeEvent[propName];
    }
    this.isDefaultPrevented = functionThatReturnsFalse;
    this.isPropagationStopped = functionThatReturnsFalse;
    return this;
  }
  Object.assign(SyntheticBaseEvent.prototype, {
    preventDefault: function () {
      this.defaultPrevented = true;
      const event = this.nativeEvent;
      if (event.preventDefault) {
        event.preventDefault();
      } else {
        event.returnValue = false;
      }
      this.isDefaultPrevented = functionThatReturnsTrue;
    },
    stopPropagation: function () {
      const event = this.nativeEvent;
      if (event.stopPropagation) {
        event.stopPropagation();
      } else {
        event.cancelBubble = true;
      }
      this.isPropagationStopped = functionThatReturnsTrue;
    }
  });
  return SyntheticBaseEvent;
}
const MouseEventInterface = {
  clientX: 0,
  clientY: 0
}
export const SyntheticMouseEvent = createSyntheticEvent(MouseEventInterface);
```

6.3 执行事件函数

6.3.1 DOMPluginEventSystem.js

src\DOMPluginEventSystem.js

```
import { allNativeEvents } from './EventRegistry';
import * as SimpleEventPlugin from './SimpleEventPlugin';
import { addEventCaptureListener, addEventBubbleListener } from './EventListener';
import { getEventListenerSet } from './ReactDOMComponentTree';
import { IS_CAPTURE_PHASE } from './EventSystemFlags';
import { dispatchEvent } from './ReactDOMEventListener';
import { HostComponent } from './ReactWorkTags';
import getListener from './getListener';
//注册插件 其实就是收集原生事件名称
SimpleEventPlugin.registerEvents();
export const nonDelegatedEvents = new Set(['scroll'])
export function listenToAllSupportedEvents(rootContainerElement) {
    allNativeEvents.forEach(domEventName => {
        //注册冒泡阶段
        if (!nonDelegatedEvents.has(domEventName)) {
            listenToNativeEvent(
                domEventName,//click
                false,//isCapturePhaseListener=false
                rootContainerElement
            );
        }
        //注册捕获阶段
        listenToNativeEvent(
            domEventName,//click
            true,//isCapturePhaseListener=false
            rootContainerElement//容器DOM元素
        );
    });
}
/**
 *
 * @param {*} domEventName DOM事件 click
 * @param {*} isCapturePhaseListener 是否是捕获事件监听
 * @param {*} rootContainerElement 根容器
 * @param {*} targetElement 目标元素
 * @param {*} eventSystemFlags 事件系统标志
 */
export function listenToNativeEvent(domEventName, isCapturePhaseListener, rootContainerElement, eventSystemFlags = 0) {
    const listenerSet = getEventListenerSet(rootContainerElement);//[]
    //click__bubble click__capture
    const listenerSetKey = getListenerSetKey(domEventName, isCapturePhaseListener);
    if (!listenerSet.has(listenerSetKey)) {
        if (isCapturePhaseListener) {
            eventSystemFlags |= IS_CAPTURE_PHASE;//4
        }
        addTrappedEventListener(
            rootContainerElement,
```

```
            domEventName,
            eventSystemFlags,
            isCapturePhaseListener
        );
        listenerSet.add(listenerSetKey);
    }
}
/**
 * 根据事件名称和是否捕获阶段得到监听的key
 * @param {*} domEventName 事件名称 click
 * @param {*} capture  是否是捕获阶段
 * @returns 监听事件的key
 */
export function getListenerSetKey(domEventName, capture) {
    return `${domEventName}__${capture ? 'capture' : 'bubble'}`;
}
/**
 * 注册监听函数
 * @param {*} targetContainer 绑定的目标容器
 * @param {*} domEventName  事件名称
 * @param {*} eventSystemFlags  事件系统标识
 * @param {*} isCapturePhaseListener  是否是捕获阶段
 */
function addTrappedEventListener(targetContainer, domEventName, eventSystemFlags, isCapturePhaseListener) {
    let listener = dispatchEvent.bind(null, domEventName, eventSystemFlags, targetContainer);
    if (isCapturePhaseListener) {
        addEventCaptureListener(targetContainer, domEventName, listener);
    } else {
        addEventBubbleListener(targetContainer, domEventName, listener);
    }
}

/**
 * 派发事件
 * @param {*} domEventName 事件名称 event
 * @param {*} eventSystemFlags 事件标志，0或者4
 * @param {*} nativeEvent 原生事件对象
 * @param {*} targetInst fiber实例
 * @param {*} targetContainer 目标容器
 */
export function dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent, targetInst, targetContainer) {
    const nativeEventTarget = nativeEvent.target;
    //事件处理函数数组
    const dispatchQueue = [];
    //提取监听事件
    SimpleEventPlugin.extractEvents(
        dispatchQueue,
        domEventName,
        targetInst,
        nativeEvent,
        nativeEventTarget,
        eventSystemFlags,
        targetContainer
    );
    //console.log('dispatchQueue',dispatchQueue);
+   processDispatchQueue(dispatchQueue, eventSystemFlags);
}

+/**
+ * 执行所有的Dispatch
+ * @param {*} dispatchQueue 队列
+ * @param {*} eventSystemFlags 事件标志
+ */
+ export function processDispatchQueue(dispatchQueue, eventSystemFlags) {
+     const inCapturePhase = (eventSystemFlags & IS_CAPTURE_PHASE) !== 0;//是否是捕获阶段
+     for (let i = 0; i < dispatchQueue.length; i++) {
+         const { event, listeners } = dispatchQueue[i];
+         processDispatchQueueItemsInOrder(event, listeners, inCapturePhase);
+     }
+}
+/**
+ * 处理dispatch方法
+ * @param {*} event 合成事件对象
+ * @param {*} dispatchListeners 监听函数
+ * @param {*} inCapturePhase 是否是获取阶段
+ */
+function processDispatchQueueItemsInOrder(event, dispatchListeners, inCapturePhase,) {
+     if (inCapturePhase) {//因为收集的时候是从内往外，所以捕获阶段是倒序执行
+         for (let i = dispatchListeners.length - 1; i >= 0; i--) {
+             const { currentTarget, listener } = dispatchListeners[i];
+             if (event.isPropagationStopped()) {
+                 return;
+             }
+             executeDispatch(event, listener, currentTarget);
+         }
+     } else {
+         for (let i = 0; i < dispatchListeners.length; i++) {
+             const { currentTarget, listener } = dispatchListeners[i];
+             if (event.isPropagationStopped()) {
+                 return;
+             }
+             executeDispatch(event, listener, currentTarget);
+         }
+     }
+}
+/**
+ * 执行监听函数
+ * @param {*} event 合成事件对象
+ * @param {*} listener 监听函数
+ * @param {*} currentTarget 当前的DOM对象
+ */
+ function executeDispatch(event, listener, currentTarget,) {
+     event.currentTarget = currentTarget;
+     listener(event);
```

```
+    event.currentTarget = null;
+}
/**
 * 收集一个阶段的监听
 * @param {*} targetFiber 对应的fiber {tag:5,type:'button'}
 * @param {*} reactName 事件名 onClick
 * @param {*} nativeEventType 原生事件名 click
 * @param {*} inCapturePhase 是否捕获阶段
 * @returns
 */
export function accumulateSinglePhaseListeners(targetFiber, reactName, nativeEventType, inCapturePhase) {
    const captureName = reactName + 'Capture';//onClickCapture
    const reactEventName = inCapturePhase ? captureName : reactName;//onClick或onClickCapture
    const listeners = [];//所有的监听函数
    let instance = targetFiber;//当前的fiber
    let lastHostComponent = null;//上一个原生DOM元素
    //从当前向上出发，收集所有的Dispatch
    while (instance !== null) {
        const { stateNode, tag } = instance;
        if (tag
            lastHostComponent = stateNode;
            if (reactEventName !== null) {
                const listener = getListener(instance, reactEventName);
                if (listener != null) {
                    listeners.push(createDispatchListener(instance, listener, lastHostComponent));
                }
            }
        }
        instance = instance.return;
    }
    return listeners;
}
/**
 * 创建Dispatch
 * @param {*} instance fiber实例
 * @param {*} listener 监听函数
 * @param {*} currentTarget 当前的DOM事件对象
 * @returns Dispatch
 */
function createDispatchListener(instance, listener, currentTarget) {
    return { instance, listener, currentTarget };
}
```

7 批量执行

7.1 ReactDOMUpdateBatching.js

src\ReactDOMUpdateBatching.js

```
export let isBatchingEventUpdates = false;
export function batchedEventUpdates(fn, a, b) {
    isBatchingEventUpdates = true;
    try {
        return fn(a, b);
    } finally {
        isBatchingEventUpdates = false;
    }
}
```

7.2 ReactDOMEventListener.js

src\ReactDOMEventListener.js

```
import {getClosestInstanceFromNode,getFiberCurrentPropsFromNode} from './ReactDOMComponentTree';
import {dispatchEventsForPlugins} from './DOMPluginEventSystem';
+import {batchedEventUpdates} from './ReactDOMUpdateBatching'
/**
 * 事件处理函数
 * @param {*} domEventName 事件名 click
 * @param {*} eventSystemFlags 4 事件系统标志
 * @param {*} targetContainer 代理的容器 div#root
 * @param {*} nativeEvent 原生的事件对象 MouseEvent
 */
export function dispatchEvent(domEventName,eventSystemFlags,targetContainer,nativeEvent) {
    const nativeEventTarget = nativeEvent.target;
    //获得来源对应的fiber对象
    const targetInst = getClosestInstanceFromNode(nativeEventTarget);
    //console.log('targetInst',targetInst);
    //获得来源对应的fiber对象
    //const props = getFiberCurrentPropsFromNode(nativeEventTarget);
    //console.log('props',props);
+   batchedEventUpdates(() =>{
        dispatchEventsForPlugins(
            domEventName,
            eventSystemFlags,
            nativeEvent,
            targetInst,
            targetContainer
+       );
    })
}
```

8 支持onChange事件

8.1 src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
let rootContainerElement = document.getElementById('root');
const handleDivClick = (event)=>{
  console.log('handleDivClick');
}
const handleDivClickCapture = (event)=>{
  console.log('handleDivClickCapture');

}
const handleButtonClick = (event)=>{
  console.log('handleButtonClick');
}
const handleButtonClickCapture = (event)=>{
  console.log('handleButtonClickCapture');
}
+const handleChange = (event)=>{
+  console.log('handleChange',event);
+}
let element = (

+
    +

)
console.log(element);
ReactDOM.render(
  element,
  rootContainerElement
);
```

8.2 SyntheticEvent.js

src\SyntheticEvent.js

```
function functionThatReturnsTrue() {
  return true;
}
function functionThatReturnsFalse() {
  return false;
}
/**
 * 返回事件构建函数
 * @param {*} Interface
 * @returns
 */
function createSyntheticEvent(Interface) {
  function SyntheticBaseEvent(reactName, reactEventType, targetInst, nativeEvent, nativeEventTarget) {
    this._reactName = reactName;
    this.type = reactEventType;
    this._targetInst = targetInst;
    this.nativeEvent = nativeEvent;
    this.target = nativeEventTarget;
    this.currentTarget = null;
    for (const propName in Interface) {
      this[propName] = nativeEvent[propName];
    }
    this.isDefaultPrevented = functionThatReturnsFalse;
    this.isPropagationStopped = functionThatReturnsFalse;
    return this;
  }
  Object.assign(SyntheticBaseEvent.prototype, {
    preventDefault: function () {
      this.defaultPrevented = true;
      const event = this.nativeEvent;
      if (event.preventDefault) {
        event.preventDefault();
      } else {//IE
        event.returnValue = false;
      }
      this.isDefaultPrevented = functionThatReturnsTrue;
    },
    stopPropagation: function () {
      const event = this.nativeEvent;
      if (event.stopPropagation) {
        event.stopPropagation();
      } else {//IE
        event.cancelBubble = true;
      }
      this.isPropagationStopped = functionThatReturnsTrue;
    }
  });
  return SyntheticBaseEvent;
}
const MouseEventInterface = {
  clientX: 0,
  clientY: 0
}
export const SyntheticMouseEvent = createSyntheticEvent(MouseEventInterface);
+export const SyntheticEvent = createSyntheticEvent({});
```

8.3 ChangeEventPlugin.js

src\ChangeEventPlugin.js

```
import { SyntheticEvent } from './SyntheticEvent';
import {registerTwoPhaseEvent} from './EventRegistry';
import {accumulateTwoPhaseListeners} from './DOMPluginEventSystem';

function extractEvents(dispatchQueue,domEventName,targetInst,nativeEvent,nativeEventTarget) {
  if(domEventName === 'input'||domEventName === 'change'){
    const listeners = accumulateTwoPhaseListeners(targetInst, 'onChange');
    if (listeners.length > 0) {
      const event = new SyntheticEvent(
        'onChange',
        'change',
        null,
        nativeEvent,
        nativeEventTarget,
      );
      dispatchQueue.push({event, listeners});
    }
  }
}

function registerEvents() {
    registerTwoPhaseEvent('onChange', [
      'change',
      'input'
    ]);
  }
export { registerEvents, extractEvents };
```

8.4 DOMPluginEventSystem.js

src\DOMPluginEventSystem.js

```
import { allNativeEvents } from './EventRegistry';
import * as SimpleEventPlugin from './SimpleEventPlugin';
+import * as ChangeEventPlugin from './ChangeEventPlugin';
import { addEventCaptureListener, addEventBubbleListener } from './EventListener';
import { getEventListenerSet } from './ReactDOMComponentTree';
import { IS_CAPTURE_PHASE } from './EventSystemFlags';
import { dispatchEvent } from './ReactDOMEventListener';
import { HostComponent } from './ReactWorkTags';
import getListener from './getListener';
//注册插件 其实就是收集原生事件名称
SimpleEventPlugin.registerEvents();
+ChangeEventPlugin.registerEvents();
export const nonDelegatedEvents = new Set(['scroll'])
export function listenToAllSupportedEvents(rootContainerElement) {
    allNativeEvents.forEach(domEventName => {
        //注册冒泡阶段
        if (!nonDelegatedEvents.has(domEventName)) {
            listenToNativeEvent(
                domEventName,//click
                false,//isCapturePhaseListener=false
                rootContainerElement
            );
        }
        //注册捕获阶段
        listenToNativeEvent(
            domEventName,//click
            true,//isCapturePhaseListener=false
            rootContainerElement//容器DOM元素
        );
    });
}
/**
 *
 * @param {*} domEventName DOM事件 click
 * @param {*} isCapturePhaseListener 是否是捕获事件监听
 * @param {*} rootContainerElement 根容器
 * @param {*} targetElement 目标元素
 * @param {*} eventSystemFlags 事件系统标志
 */
export function listenToNativeEvent(domEventName, isCapturePhaseListener, rootContainerElement, eventSystemFlags = 0) {
    const listenerSet = getEventListenerSet(rootContainerElement);//[]
    //click__bubble click__capture
    const listenerSetKey = getListenerSetKey(domEventName, isCapturePhaseListener);
    if (!listenerSet.has(listenerSetKey)) {
        if (isCapturePhaseListener) {
            eventSystemFlags |= IS_CAPTURE_PHASE;//4
        }
        addTrappedEventListener(
            rootContainerElement,
            domEventName,
            eventSystemFlags,
            isCapturePhaseListener
        );
        listenerSet.add(listenerSetKey);
    }
}
/**
 * 根据事件名称和是否捕获阶段得到监听的key
 * @param {*} domEventName 事件名称 click
 * @param {*} capture  是否是捕获阶段
 * @returns 监听事件的key
 */
export function getListenerSetKey(domEventName, capture) {
    return `${domEventName}__${capture ? 'capture' : 'bubble'}`;
}
/**
 * 注册监听函数
 * @param {*} targetContainer 绑定的目标容器
 * @param {*} domEventName  事件名称
 * @param {*} eventSystemFlags  事件系统标识
 * @param {*} isCapturePhaseListener  是否是捕获阶段
 */
```

```
function addTrappedEventListener(targetContainer, domEventName, eventSystemFlags, isCapturePhaseListener) {
    let listener = dispatchEvent.bind(null, domEventName, eventSystemFlags, targetContainer);
    if (isCapturePhaseListener) {
        addEventCaptureListener(targetContainer, domEventName, listener);
    } else {
        addEventBubbleListener(targetContainer, domEventName, listener);
    }
}

/**
 * 派发事件
 * @param {*} domEventName 事件名称 event
 * @param {*} eventSystemFlags 事件标志, 0或者4
 * @param {*} nativeEvent 原生事件对象
 * @param {*} targetInst fiber实例
 * @param {*} targetContainer 目标容器
 */
export function dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent, targetInst, targetContainer) {
    const nativeEventTarget = nativeEvent.target;
    //事件处理函数数组
    const dispatchQueue = [];
    //提取监听事件
    SimpleEventPlugin.extractEvents(
        dispatchQueue,
        domEventName,
        targetInst,
        nativeEvent,
        nativeEventTarget,
        eventSystemFlags,
        targetContainer
    );
+   if (eventSystemFlags !== IS_CAPTURE_PHASE) {
+       ChangeEventPlugin.extractEvents(
+           dispatchQueue,
+           domEventName,
+           targetInst,
+           nativeEvent,
+           nativeEventTarget,
+           eventSystemFlags,
+           targetContainer
+       )
+   }
    processDispatchQueue(dispatchQueue, eventSystemFlags);
}

/**
 * 执行所有的Dispatch
 * @param {*} dispatchQueue 队列
 * @param {*} eventSystemFlags 事件标志
 */
export function processDispatchQueue(dispatchQueue, eventSystemFlags) {
    const inCapturePhase = (eventSystemFlags & IS_CAPTURE_PHASE) !== 0;//是否是捕获阶段
    for (let i = 0; i < dispatchQueue.length; i++) {
        const { event, listeners } = dispatchQueue[i];
        processDispatchQueueItemsInOrder(event, listeners, inCapturePhase);
    }
}
/**
 * 处理dispatch方法
 * @param {*} event 合成事件对象
 * @param {*} dispatchListeners 监听函数
 * @param {*} inCapturePhase 是否是获取阶段
 */
function processDispatchQueueItemsInOrder(event, dispatchListeners, inCapturePhase,) {
    if (inCapturePhase) {//因为收集的时候是从内往外，所以捕获阶段是倒序执行
        for (let i = dispatchListeners.length - 1; i >= 0; i--) {
            const { currentTarget, listener } = dispatchListeners[i];
            if (event.isPropagationStopped()) {
                return;
            }
            executeDispatch(event, listener, currentTarget);
        }
    } else {
        for (let i = 0; i < dispatchListeners.length; i++) {
            const { currentTarget, listener } = dispatchListeners[i];
            if (event.isPropagationStopped()) {
                return;
            }
            executeDispatch(event, listener, currentTarget);
        }
    }
}
/**
 * 执行监听函数
 * @param {*} event 合成事件对象
 * @param {*} listener 监听函数
 * @param {*} currentTarget 当前的DOM对象
 */
function executeDispatch(event, listener, currentTarget,) {
    event.currentTarget = currentTarget;
    listener(event);
    event.currentTarget = null;
}
/**
 * 收集一个阶段的监听
 * @param {*} targetFiber 对应的fiber {tag:5,type:'button'}
 * @param {*} reactName 事件名 onClick
 * @param {*} nativeEventType 原生事件名 click
 * @param {*} inCapturePhase 是否捕获阶段
 * @returns
 */
export function accumulateSinglePhaseListeners(targetFiber, reactName, nativeEventType, inCapturePhase) {
    const captureName = reactName + 'Capture';//onClickCapture
    const reactEventName = inCapturePhase ? captureName : reactName;//onClick或onClickCapture
```

```
    const listeners = [];//所有的监听函数
    let instance = targetFiber;//当前的fiber
    let lastHostComponent = null;//上一个原生DOM元素
    //从当前向上出发，收集所有的Dispatch
    while (instance !== null) {
        const { stateNode, tag } = instance;
        if (tag
            lastHostComponent = stateNode;
            if (reactEventName !== null) {
                const listener = getListener(instance, reactEventName);
                if (listener != null) {
                    listeners.push(createDispatchListener(instance, listener, lastHostComponent));
                }
            }
        }
        instance = instance.return;
    }
    return listeners;
}
/**
 * 创建Dispatch
 * @param {*} instance fiber实例
 * @param {*} listener 监听函数
 * @param {*} currentTarget 当前的DOM事件对象
 * @returns Dispatch
 */
function createDispatchListener(instance, listener, currentTarget) {
    return { instance, listener, currentTarget };
}

+export function accumulateTwoPhaseListeners(targetFiber,reactName) {
+    const captureName = reactName + 'Capture';
+    const listeners = [];
+    let instance = targetFiber;
+    while (instance !== null) {
+        const { stateNode, tag } = instance;
+        if (tag === HostComponent && stateNode !== null) {
+            const currentTarget = stateNode;
+            const captureListener = getListener(instance, captureName);
+            if (captureListener != null) {
+                listeners.unshift(
+                    createDispatchListener(instance, captureListener, currentTarget),
+                );
+            }
+            const bubbleListener = getListener(instance, reactName);
+            if (bubbleListener != null) {
+                listeners.push(
+                    createDispatchListener(instance, bubbleListener, currentTarget),
+                );
+            }
+        }
+        instance = instance.return;
+    }
+    return listeners;
+}
```