18-3 实现事件派发

dispatchDiscreteEvent

适用事件

click、drop、input、drop 等

packages/react-dom-bindings/src/events/ReactDOMEventListener.js

JavaScript

```javascript
// Used by SimpleEventPlugin:
    case 'cancel':
    case 'click':
    case 'close':
    case 'contextmenu':
    case 'copy':
    case 'cut':
    case 'auxclick':
    case 'dblclick':
    case 'dragend':
    case 'dragstart':
    case 'drop':
    case 'focusin':
```

```
    case 'focusout':
    case 'input':
    case 'invalid':
    case 'keydown':
    case 'keypress':
    case 'keyup':
    case 'mousedown':
    case 'mouseup':
    case 'paste':
    case 'pause':
    case 'play':
    case 'pointercancel':
    case 'pointerdown':
    case 'pointerup':
    case 'ratechange':
    case 'reset':
    case 'resize':
    case 'seeked':
    case 'submit':
    case 'touchcancel':
    case 'touchend':
    case 'touchstart':
    case 'volumechange':
    // Used by polyfills: (fall through)
    case 'change':
    case 'selectionchange':
    case 'textInput':
    case 'compositionstart':
    case 'compositionend':
    case 'compositionupdate':
    // Only enableCreateEventHandleAPI: (fall through)
    case 'beforeblur':
    case 'afterblur':
    // Not used by React but could be by user code: (fall through)
    case 'beforeinput':
    case 'blur':
    case 'fullscreenchange':
    case 'focus':
    case 'hashchange':
    case 'popstate':
    case 'select':
```

```
    case 'selectstart':

     case "message": {
      // 我们可能在调度器回调中。
      // 最终，这种机制将被替换为检查本机调度器上的当前优先级。
      const schedulerPriority = Scheduler.getCurrentPriorityLevel();
      switch (schedulerPriority) {
        case ImmediatePriority:
          return DiscreteEventPriority;
        case UserBlockingPriority:
          return ContinuousEventPriority;
        case NormalPriority:
        case LowPriority:
          return DefaultEventPriority;
        case IdlePriority:
          return IdleEventPriority;
        default:
          return DefaultEventPriority;
      }
    }
```

派发事件源码

packages/react-dom-bindings/src/events/ReactDOMEventListener.js

JavaScript

```javascript
function dispatchDiscreteEvent(
  domEventName: DOMEventName,
  eventSystemFlags: EventSystemFlags,
  container: EventTarget,
  nativeEvent: AnyNativeEvent
) {
  // ! 1. 记录上一次的事件优先级
  const previousPriority = getCurrentUpdatePriority();
  try {
    // !4. 设置当前事件优先级为DiscreteEventPriority
    setCurrentUpdatePriority(DiscreteEventPriority);
    // !5. 调用dispatchEvent，执行事件
    dispatchEvent(domEventName, eventSystemFlags, container, nativeEve
  } finally {
    // !6. 恢复
    setCurrentUpdatePriority(previousPriority);
  }
}
```

事件优先级记录

packages/react-reconciler/src/ReactEventPriorities.js

```JavaScript
export opaque type EventPriority = Lane;

export const DiscreteEventPriority: EventPriority = SyncLane;
export const ContinuousEventPriority: EventPriority = InputContinuousL
export const DefaultEventPriority: EventPriority = DefaultLane; // 页面
export const IdleEventPriority: EventPriority = IdleLane;

let currentUpdatePriority: EventPriority = NoLane;

export function getCurrentUpdatePriority(): EventPriority {
  return currentUpdatePriority;
}

export function setCurrentUpdatePriority(newPriority: EventPriority) {
  currentUpdatePriority = newPriority;
}
```

dispatchContinuousEvent

适用事件

packages/react-dom-bindings/src/events/ReactDOMEventListener.js

JavaScript

```javascript
case 'drag':
case 'dragenter':
case 'dragexit':
case 'dragleave':
case 'dragover':
case 'mousemove':
case 'mouseout':
case 'mouseover':
case 'pointermove':
case 'pointerout':
case 'pointerover':
case 'scroll':
case 'toggle':
case 'touchmove':
case 'wheel':
// Not used by React but could be by user code: (fall through)
case 'mouseenter':
case 'mouseleave':
case 'pointerenter':
case 'pointerleave':

 case "message": {
  // 我们可能在调度器回调中。
  // 最终，这种机制将被替换为检查本机调度器上的当前优先级。
  const schedulerPriority = Scheduler.getCurrentPriorityLevel();
  switch (schedulerPriority) {
    case ImmediatePriority:
      return DiscreteEventPriority;
    case UserBlockingPriority:
      return ContinuousEventPriority;
    case NormalPriority:
    case LowPriority:
      return DefaultEventPriority;
    case IdlePriority:
      return IdleEventPriority;
    default:
      return DefaultEventPriority;
  }
}
```

派发事件源码

packages/react-dom-bindings/src/events/ReactDOMEventListener.js

JavaScript
```javascript
function dispatchContinuousEvent(
  domEventName: DOMEventName,
  eventSystemFlags: EventSystemFlags,
  container: EventTarget,
  nativeEvent: AnyNativeEvent
) {
  const previousPriority = getCurrentUpdatePriority();
  try {
    setCurrentUpdatePriority(ContinuousEventPriority);
    dispatchEvent(domEventName, eventSystemFlags, container, nativeEve
  } finally {
    setCurrentUpdatePriority(previousPriority);
  }
}
```

dispatchEvent

packages/react-dom-bindings/src/events/ReactDOMEventListener.js

JavaScript
```javascript
export function dispatchEvent(
  domEventName: DOMEventName,
  eventSystemFlags: number,
  targetContainer: EventTarget,
  nativeEvent: AnyNativeEvent
): void {
  // 有些场景下是禁止事件的，比如在commit阶段
  if (domEventName === "click") {
    const nativeEventTarget = nativeEvent.target;
    return_targetInst = getClosestInstanceFromNode(nativeEventTarget);

    const dispatchQueue: DispatchQueue = [];
    extractEvents(
      dispatchQueue,
      domEventName,
      return_targetInst,
      nativeEvent,
      nativeEventTarget,
      eventSystemFlags,
      targetContainer
    );

    processDispatchQueue(dispatchQueue, eventSystemFlags);
  }
}
```

extractEvents

packages/react-dom-bindings/src/events/DOMPluginEventSystem.ts

以 SimpleEvent 为例：

```TypeScript
export type AnyNativeEvent = Event | KeyboardEvent | MouseEvent | Touc

export type DispatchListener = {
  instance: null | Fiber;
  listener: Function;
  currentTarget: EventTarget;
};

type DispatchEntry = {
  event: AnyNativeEvent;
  listeners: Array<DispatchListener>;
};

export type DispatchQueue = Array<DispatchEntry>;

export function extractEvents(
  dispatchQueue: DispatchQueue,
  domEventName: DOMEventName,
  targetInst: null | Fiber,
  nativeEvent: AnyNativeEvent,
  nativeEventTarget: null | EventTarget,
  eventSystemFlags: EventSystemFlags,
  targetContainer: EventTarget
) {
  SimpleEventPlugin.extractEvents(
    dispatchQueue,
    domEventName,
    targetInst,
    nativeEvent,
    nativeEventTarget,
    eventSystemFlags,
    targetContainer
  );
}
```

packages/react-dom-bindings/src/events/plugins/SimpleEventPlugin.ts

```TypeScript
import {
  registerSimpleEvents,
  topLevelEventsToReactNames,
} from "../DOMEventProperties";
import { DOMEventName } from "../DOMEventNames";
import { Fiber } from "react-reconciler/src/ReactInternalTypes";
import {
  AnyNativeEvent,
  DispatchQueue,
  accumulateSinglePhaseListeners,
} from "../DOMPluginEventSystem";

import { IS_CAPTURE_PHASE, type EventSystemFlags } from "../EventSyste

function extractEvents(
  dispatchQueue: DispatchQueue,
  domEventName: DOMEventName,
  targetInst: null | Fiber,
  nativeEvent: AnyNativeEvent,
  nativeEventTarget: null | EventTarget,
  eventSystemFlags: EventSystemFlags,
  targetContainer: EventTarget
): void {
  // click->onClick
  const reactName = topLevelEventsToReactNames.get(domEventName);
  if (reactName === undefined) {
    return;
  }

  const inCapturePhase = (eventSystemFlags & IS_CAPTURE_PHASE) !== 0;
  // 如果是 scroll 事件，或者是 scrollend 事件，那么只会在冒泡阶段触发
  const accumulateTargetOnly =
    !inCapturePhase &&
    (domEventName === "scroll" || domEventName === "scrollend");

  const listeners = accumulateSinglePhaseListeners(
    targetInst,
    reactName,
    nativeEvent.type,
    inCapturePhase,
    accumulateTargetOnly,
    nativeEvent
  );

  if (listeners.length > 0) {
    dispatchQueue.push({ event: nativeEvent, listeners });
  }
}

export { registerSimpleEvents as registerEvents, extractEvents };
```

accumulateSinglePhaseListeners

packages/react-dom-bindings/src/events/DOMPluginEventSystem.ts

TypeScript

```
export function accumulateSinglePhaseListeners(
  targetFiber: Fiber | null,
  reactName: string | null,
  nativeEventType: string,
  inCapturePhase: boolean,
  accumulateTargetOnly: boolean,
  nativeEvent: AnyNativeEvent
): Array<DispatchListener> {
  const captureName = reactName !== null ? reactName + "Capture" : nul
  const reactEventName = inCapturePhase ? captureName : reactName;
  let listeners: Array<DispatchListener> = [];

  let instance = targetFiber;

  // 通过target -> root累积所有fiber和listeners。
  while (instance !== null) {
    const { stateNode, tag } = instance;
    // 处理位于HostComponents（即 <div> 元素）上的listeners
    if (tag === HostComponent) {
      // 标准 React on* listeners, i.e. onClick or onClickCapture
      const listener = getListener(instance, reactEventName as string)
      if (listener != null) {
        listeners.push({
          instance,
          listener,
          currentTarget: stateNode,
        });
      }
    }
    // 如果只是为target累积事件，那么我们就不会继续通过 React Fiber 树传播以查
    if (accumulateTargetOnly) {
      break;
    }

    instance = instance.return;
  }
  return listeners;
}
```

processDispatchQueue

packages/react-dom-bindings/src/events/ReactDOMEventListener.ts

JavaScript

```javascript
export function processDispatchQueue(
  dispatchQueue: DispatchQueue,
  eventSystemFlags: EventSystemFlags
): void {
  const inCapturePhase = (eventSystemFlags & IS_CAPTURE_PHASE) !== 0;
  for (let i = 0; i < dispatchQueue.length; i++) {
    const { event, listeners } = dispatchQueue[i];

    processDispatchQueueItemsInOrder(event, listeners, inCapturePhase)
  }
}
```

processDispatchQueueItemsInOrder

packages/react-dom-bindings/src/events/ReactDOMEventListener.ts

JavaScript

```javascript
function processDispatchQueueItemsInOrder(
  event: Event,
  dispatchListeners: Array<DispatchListener>,
  inCapturePhase: boolean
): void {
  if (inCapturePhase) {
    // 捕获阶段，从上往下执行
    for (let i = dispatchListeners.length - 1; i >= 0; i--) {
      const { instance, currentTarget, listener } = dispatchListeners[
      executeDispatch(event, listener, currentTarget);
    }
  } else {
    for (let i = 0; i < dispatchListeners.length; i++) {
      const { instance, currentTarget, listener } = dispatchListeners[
      executeDispatch(event, listener, currentTarget);
    }
  }
}
```

执行事件

packages/react-dom-bindings/src/events/ReactDOMEventListener.ts

JavaScript
```javascript
function executeDispatch(
  event: Event,
  listener: Function,
  currentTarget: EventTarget
): void {
  const type = event.type || "unknown-event";
  // event.currentTarget = currentTarget;
  invokeGuardedCallbackAndCatchFirstError(type, listener, undefined, e
  // event.currentTarget = null;
}
```

packages/shared/ReactErrorUtils.js

JavaScript
```javascript
export function invokeGuardedCallbackAndCatchFirstError<
  A,
  B,
  C,
  D,
  E,
  F,
  Context,
>(
  this: mixed,
  name: string | null,
  func: (a: A, b: B, c: C, d: D, e: E, f: F) => void,
  context: Context,
  a: A,
  b: B,
  c: C,
  d: D,
  e: E,
  f: F,
): void {
  invokeGuardedCallback.apply(this, arguments);
  if (hasError) {
    const error = clearCaughtError();
    if (!hasRethrowError) {
      hasRethrowError = true;
      rethrowError = error;
    }
  }
}

export function invokeGuardedCallback<A, B, C, D, E, F, Context>(
  name: string | null,
  func: (a: A, b: B, c: C, d: D, e: E, f: F) => mixed,
  context: Context,
  a: A,
  b: B,
  c: C,
  d: D,
  e: E,
  f: F,
): void {
  hasError = false;
  caughtError = null;
  invokeGuardedCallbackImpl.apply(reporter, arguments);
}
```

packages/shared/invokeGuardedCallbackImpl.js

JavaScript
```
export default function invokeGuardedCallbackImpl<Args: Array<mixed>,
  this: {onError: (error: mixed) => void},
  name: string | null,
  func: (...Args) => mixed,
  context: Context,
): void {
   const funcArgs = Array.prototype.slice.call(arguments, 3);
    try {
      func.apply(context, funcArgs);
    } catch (error) {
      this.onError(error);
    }
}
```