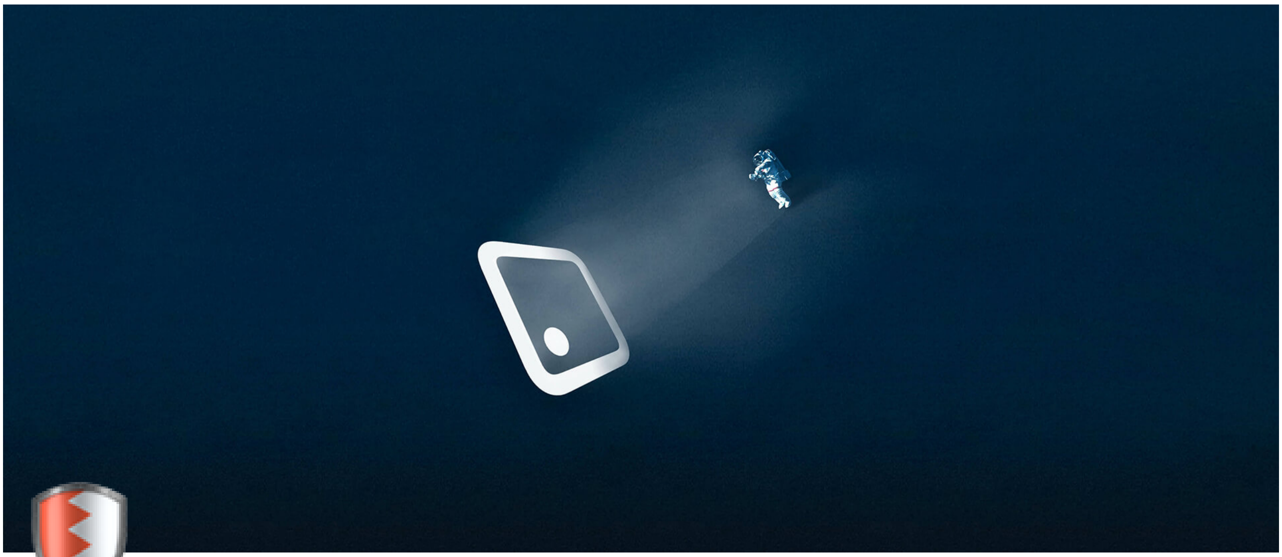


8-3 update的数据结构与算法

本节是类组件与 createRoot(domNode).render()产生的 update，函数组件的 update 在其它篇章单独讲解。

类似 fiber，update queues 也是成对出现的，一个已经完成的即对应目前页面，一个正在工作中的。

`Update`、`SharedQueue`、`UpdateQueue` 类型定义

react/packages/react-reconciler/src/ReactFiberClassUpdateQueue.js

JavaScript

```
export type Update<State> = {
  lane: Lane,

  tag: 0 | 1 | 2 | 3,
  payload: any,
  callback: (() => mixed) | null,

  next: Update<State> | null,
};
```

```
export type SharedQueue<State> = {
  pending: Update<State> | null, // 单向循环链表，尾节点->头结点
  lanes: Lanes,
  // 如果类组件是Activity(以前叫OffScreen)的后代组件，需要延迟执行的其setState
  // Activity目前还是unstable，了解即可~
  hiddenCallbacks: Array<() => mixed> | null,
};

export type UpdateQueue<State> = {
  baseState: State,
  // 单链表 firstBaseUpdate->...->lastBaseUpdate
  firstBaseUpdate: Update<State> | null,
  lastBaseUpdate: Update<State> | null,
  shared: SharedQueue<State>,
  callbacks: Array<() => mixed> | null,
};

export const UpdateState = 0;
export const ReplaceState = 1;
export const ForceUpdate = 2;
export const CaptureUpdate = 3;
```

1. 初始化 `fiber.updateQueue`

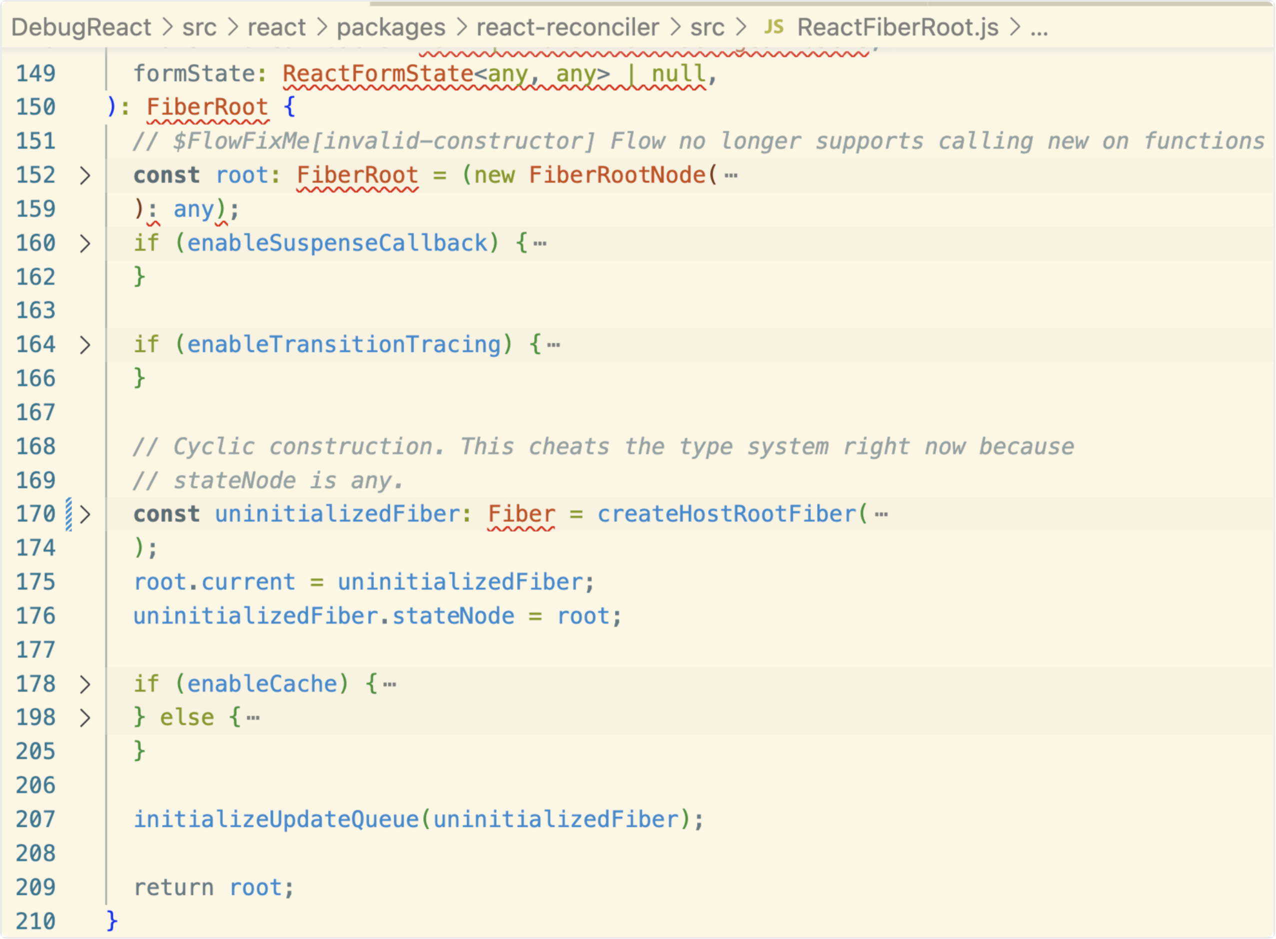
初次渲染页面和**类组件初次挂载**的时候，调用函数 `initializeUpdateQueue` 来初始化 `fiber.updateQueue`。

react/packages/react-reconciler/src/ReactFiberClassUpdateQueue.js

```javascript
// 这里初始化fiber.updateQueue。在beginWork阶段，updateHostRoot中使用proces
export function initializeUpdateQueue<State>(fiber: Fiber): void {
  const queue: UpdateQueue<State> = {
    baseState: fiber.memoizedState,
    // 单向循环链表
    firstBaseUpdate: null,
    lastBaseUpdate: null,
    shared: {
      pending: null,
      lanes: NoLanes,
      hiddenCallbacks: null,
    },
    callbacks: null,
  };
  fiber.updateQueue = queue;
}
```

初次渲染页面

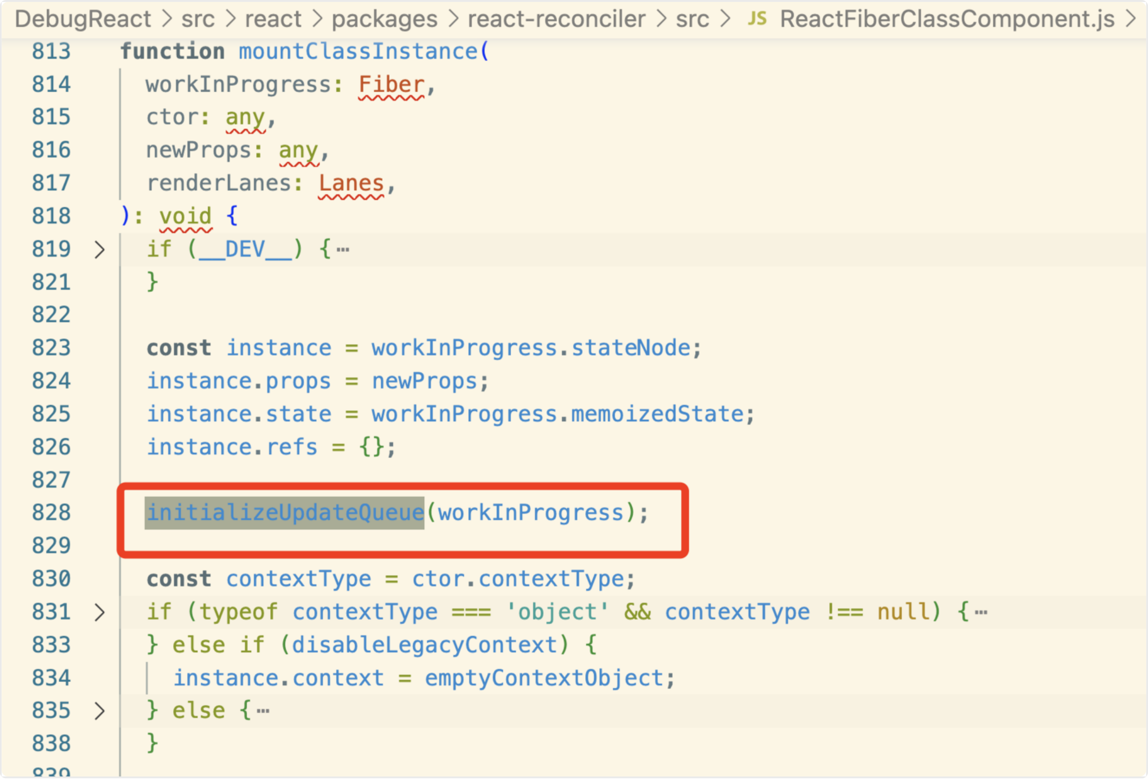
`createRoot` 阶段，创建并返回 FiberRoot：



DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberRoot.js > ...

```
  formState: ReactFormState<any, any> | null,
): FiberRoot {
  // $FlowFixMe[invalid-constructor] Flow no longer supports calling new on functions
  const root: FiberRoot = (new FiberRootNode(⋯
  ): any);
  if (enableSuspenseCallback) {⋯
  }

  if (enableTransitionTracing) {⋯
  }

  // Cyclic construction. This cheats the type system right now because
  // stateNode is any.
  const uninitializedFiber: Fiber = createHostRootFiber(⋯
  );
  root.current = uninitializedFiber;
  uninitializedFiber.stateNode = root;

  if (enableCache) {⋯
  } else {⋯
  }

  initializeUpdateQueue(uninitializedFiber);

  return root;
}
```

类组件初次挂载



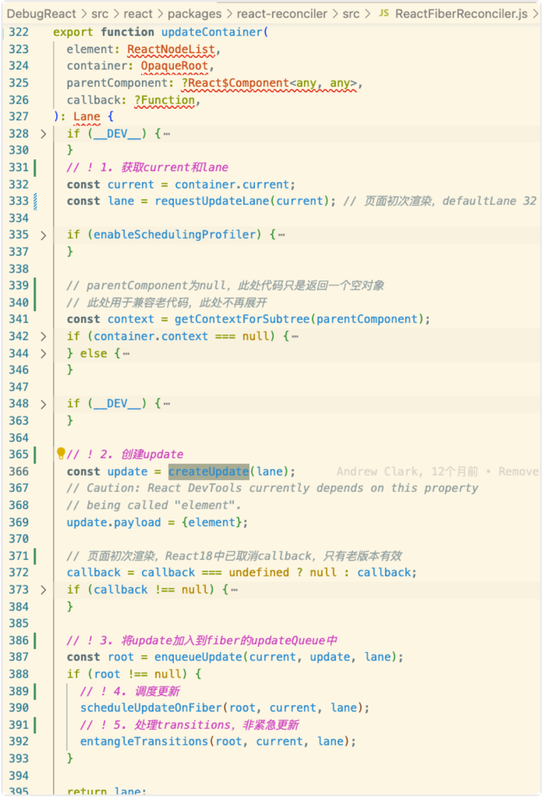
DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberClassComponent.js >

```
function mountClassInstance(
  workInProgress: Fiber,
  ctor: any,
  newProps: any,
  renderLanes: Lanes,
): void {
  if (__DEV__) {…
  }

  const instance = workInProgress.stateNode;
  instance.props = newProps;
  instance.state = workInProgress.memoizedState;
  instance.refs = {};

  initializeUpdateQueue(workInProgress);

  const contextType = ctor.contextType;
  if (typeof contextType === 'object' && contextType !== null) {…
  } else if (disableLegacyContext) {
    instance.context = emptyContextObject;
  } else {…
  }
```

2. 创建 update

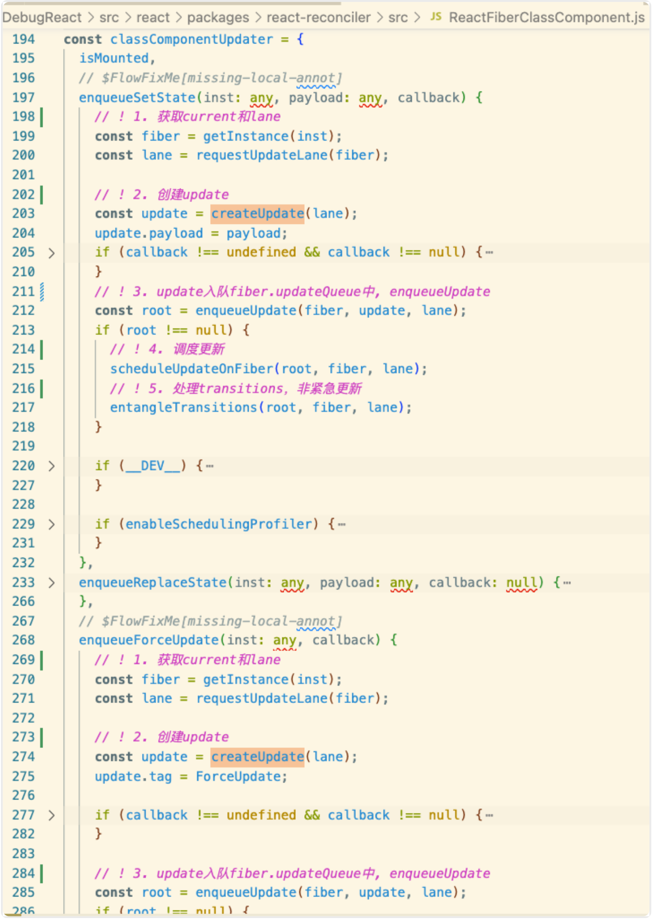
`createRoot(root).render()` 阶段与类组件的 `setState`、`forceUpdate` 阶段均会创建 update：

src/react/packages/react-reconciler/src/ReactFiberReconciler.js 与 react/packages/react-reconciler/src/ReactFiberClassComponent.js



DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberReconciler.js >

```
export function updateContainer(
  element: ReactNodeList,
  container: OpaqueRoot,
  parentComponent: ?React$Component<any, any>,
  callback: ?Function,
): Lane {
  if (__DEV__) {…
  }
  // ! 1. 获取current和lane
  const current = container.current;
  const lane = requestUpdateLane(current); // 页面初次渲染, defaultLane 32

  if (enableSchedulingProfiler) {…
  }

  // parentComponent为null, 此处代码只是返回一个空对象
  // 此处用于兼容老代码, 此处不再展开
  const context = getContextForSubtree(parentComponent);
  if (container.context === null) {…
  } else {…
  }

  if (__DEV__) {…
  }

  // ! 2. 创建update
  const update = createUpdate(lane);       Andrew Clark, 12个月前 • Remove
  // Caution: React DevTools currently depends on this property
  // being called "element".
  update.payload = {element};

  // 页面初次渲染, React18中已取消callback, 只有老版本有效
  callback = callback === undefined ? null : callback;
  if (callback !== null) {…
  }

  // ! 3. 将update加入到fiber的updateQueue中
  const root = enqueueUpdate(current, update, lane);
  if (root !== null) {
    // ! 4. 调度更新
    scheduleUpdateOnFiber(root, current, lane);
    // ! 5. 处理transitions, 非紧急更新
    entangleTransitions(root, current, lane);
  }

  return lane;
```



DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberClassComponent.js

```
const classComponentUpdater = {
  isMounted,
  // $FlowFixMe[missing-local-annot]
  enqueueSetState(inst: any, payload: any, callback) {
    // ! 1. 获取current和lane
    const fiber = getInstance(inst);
    const lane = requestUpdateLane(fiber);

    // ! 2. 创建update
    const update = createUpdate(lane);
    update.payload = payload;
    if (callback !== undefined && callback !== null) {…
    }
    // ! 3. update入队fiber.updateQueue中, enqueueUpdate
    const root = enqueueUpdate(fiber, update, lane);
    if (root !== null) {
      // ! 4. 调度更新
      scheduleUpdateOnFiber(root, fiber, lane);
      // ! 5. 处理transitions, 非紧急更新
      entangleTransitions(root, fiber, lane);
    }

    if (__DEV__) {…
    }

    if (enableSchedulingProfiler) {…
    }
  },
  enqueueReplaceState(inst: any, payload: any, callback: null) {…
  },
  // $FlowFixMe[missing-local-annot]
  enqueueForceUpdate(inst: any, callback) {
    // ! 1. 获取current和lane
    const fiber = getInstance(inst);
    const lane = requestUpdateLane(fiber);

    // ! 2. 创建update
    const update = createUpdate(lane);
    update.tag = ForceUpdate;

    if (callback !== undefined && callback !== null) {…
    }

    // ! 3. update入队fiber.updateQueue中, enqueueUpdate
    const root = enqueueUpdate(fiber, update, lane);
    if (root !== null) {
```

`createUpdate`

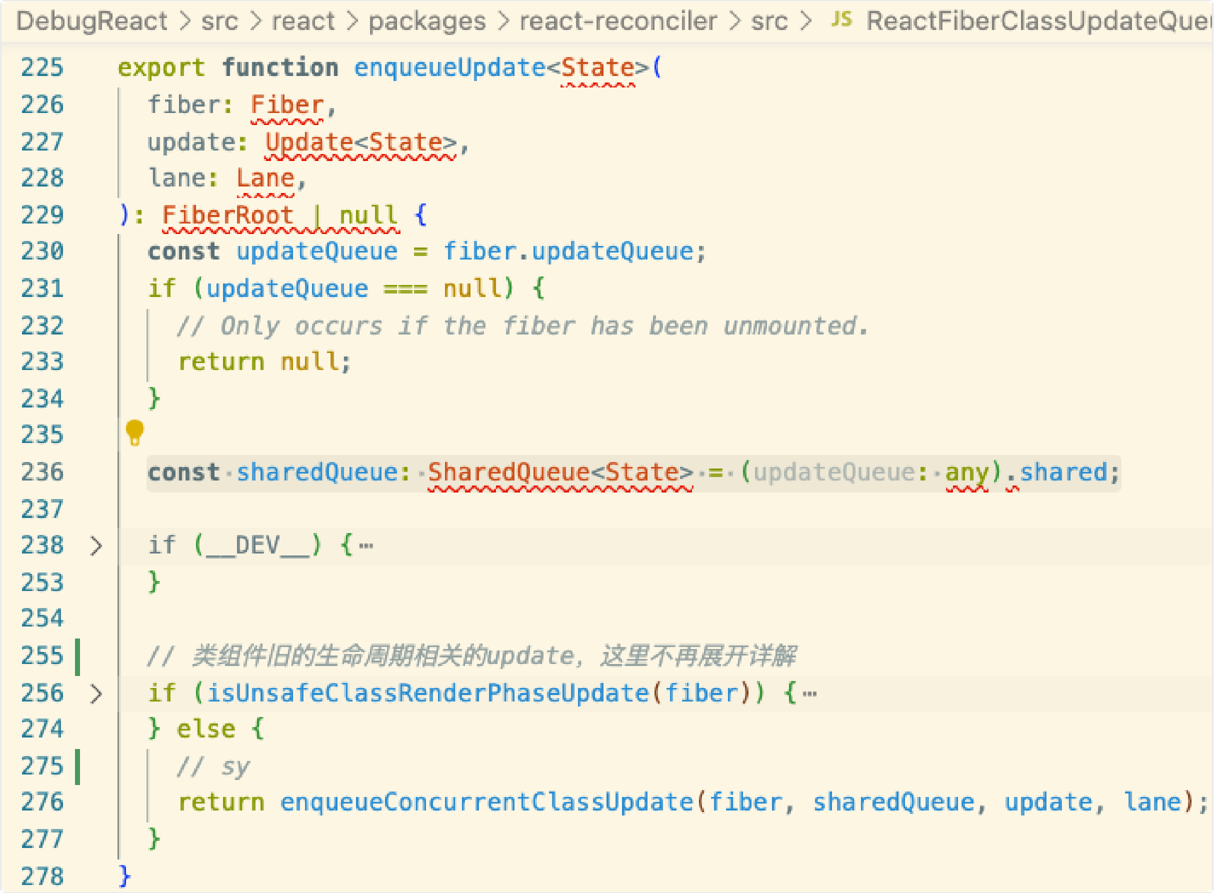
创建 update

react/packages/react-reconciler/src/ReactFiberClassUpdateQueue.js

```javascript
export const UpdateState = 0;

export function createUpdate(lane: Lane): Update<mixed> {
  const update: Update<mixed> = {
    lane,

    tag: UpdateState,
    payload: null,
    callback: null,

    next: null,
  };
  return update;
}
```

3. update 入队

1. `createRoot(root).render()` 阶段与类组件的 `setState`、`forceUpdate` 阶段最开始调用的是 ReactFiberClassUpdateQueue.js 中的 `enqueueUpdate`，源码如下：



DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberClassUpdateQue

```
export function enqueueUpdate<State>(
  fiber: Fiber,
  update: Update<State>,
  lane: Lane,
): FiberRoot | null {
  const updateQueue = fiber.updateQueue;
  if (updateQueue === null) {
    // Only occurs if the fiber has been unmounted.
    return null;
  }

  const sharedQueue: SharedQueue<State> = (updateQueue: any).shared;

  if (__DEV__) {…
  }

  // 类组件旧的生命周期相关的update，这里不再展开详解
  if (isUnsafeClassRenderPhaseUpdate(fiber)) {…
  } else {
    // sy
    return enqueueConcurrentClassUpdate(fiber, sharedQueue, update, lane);
  }
}
```

`createRoot(root).render()` 阶段与类组件的 `setState`、`forceUpdate` 阶段均会创建 update：

把 update 存储到 concurrentQueues 中。

react/packages/react-reconciler/src/ReactFiberConcurrentUpdates.js

部分代码

JavaScript

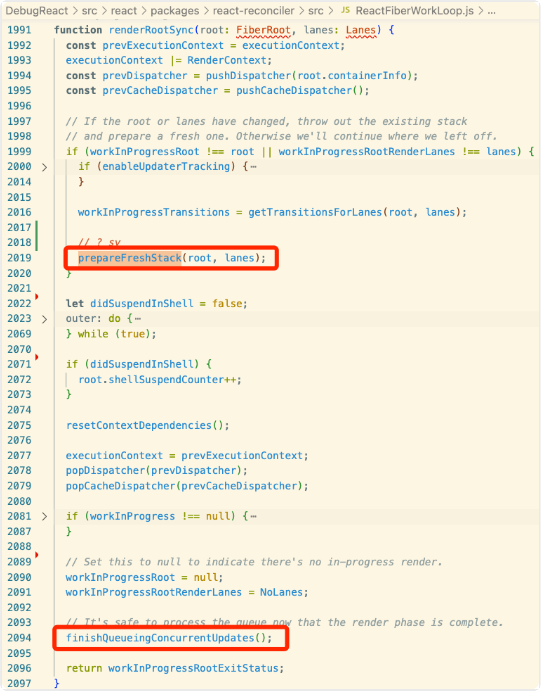
```
// ClassUpdate | HookUpdate;
export type ConcurrentUpdate = {
  next: ConcurrentUpdate,
  lane: Lane,
};
// ClassQueue | HookQueue;
type ConcurrentQueue = {
  pending: ConcurrentUpdate | null,
};

// 如果渲染正在进行中，并且收到来自并发事件的更新,我们会等到当前的渲染结束（无论是完
// 将其推送到这个数组中，这样我们以后就可以访问queue、fiber、update等。
const concurrentQueues: Array<any> = [];
let concurrentQueuesIndex = 0;

let concurrentlyUpdatedLanes: Lanes = NoLanes;

export function enqueueConcurrentClassUpdate<State>(
  fiber: Fiber,
  queue: ClassQueue<State>,
  update: ClassUpdate<State>,
  lane: Lane,
): FiberRoot | null {
  const concurrentQueue: ConcurrentQueue = (queue: any);
  const concurrentUpdate: ConcurrentUpdate = (update: any);
  // ! 1. update入队
  enqueueUpdate(fiber, concurrentQueue, concurrentUpdate, lane);
  // ! 2. 返回FiberRoot
  return getRootForUpdatedFiber(fiber);
}

function enqueueUpdate(
  fiber: Fiber,
  queue: ConcurrentQueue | null,
  update: ConcurrentUpdate | null,
  lane: Lane,
) {
  concurrentQueues[concurrentQueuesIndex++] = fiber;
  concurrentQueues[concurrentQueuesIndex++] = queue;
  concurrentQueues[concurrentQueuesIndex++] = update;
  concurrentQueues[concurrentQueuesIndex++] = lane;
  concurrentlyUpdatedLanes = mergeLanes(concurrentlyUpdatedLanes, lane

  fiber.lanes = mergeLanes(fiber.lanes, lane);
  const alternate = fiber.alternate;
  if (alternate !== null) {
    alternate.lanes = mergeLanes(alternate.lanes, lane);
  }
}
```

4. 管理更新队列 `finishQueueingConcurrentUpdates`

`finishQueueingConcurrentUpdates` 把 `concurrentQueues` 的内容添加到 fiber 的 queue 中。

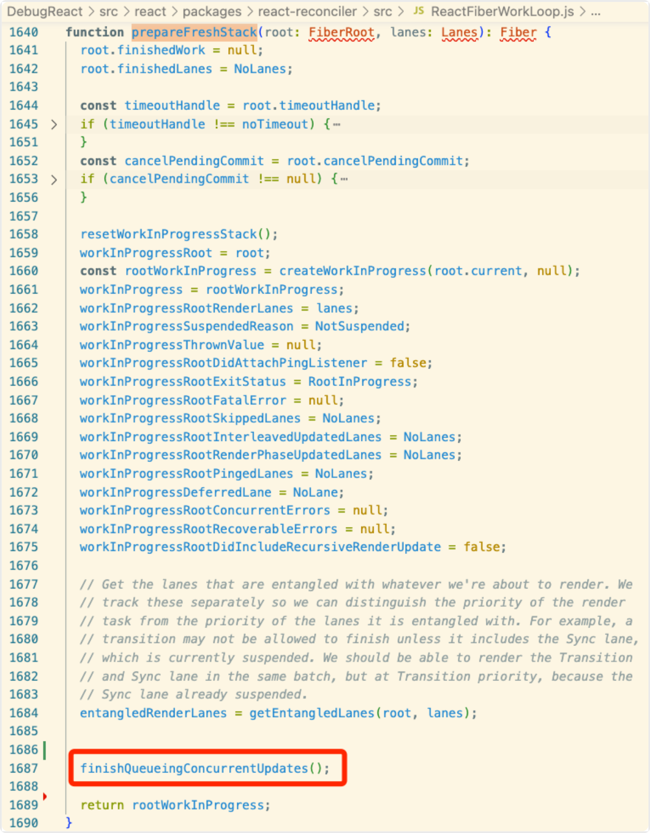
在 render 阶段，有两处调用 `finishQueueingConcurrentUpdates`，分别是 1. render 开始的时候，在 `prepareFreshStack` 函数中； 2， 在 render 结束的时候，最后再调用一遍。

和， 先把 concurrentQueues 的内容添加到 fiber 的 queue 中。后续才是根据 VDOM 更新 DOM。此函数的调用在 `prepareFreshStack` 中：



DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberWorkLoop.js > ...

```
function renderRootSync(root: FiberRoot, lanes: Lanes) {
  const prevExecutionContext = executionContext;
  executionContext |= RenderContext;
  const prevDispatcher = pushDispatcher(root.containerInfo);
  const prevCacheDispatcher = pushCacheDispatcher();

  // If the root or lanes have changed, throw out the existing stack
  // and prepare a fresh one. Otherwise we'll continue where we left off.
  if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
    if (enableUpdaterTracking) {⋯
    }

    workInProgressTransitions = getTransitionsForLanes(root, lanes);

    // ? sy
    prepareFreshStack(root, lanes);
  }

  let didSuspendInShell = false;
  outer: do {⋯
  } while (true);

  if (didSuspendInShell) {
    root.shellSuspendCounter++;
  }

  resetContextDependencies();

  executionContext = prevExecutionContext;
  popDispatcher(prevDispatcher);
  popCacheDispatcher(prevCacheDispatcher);

  if (workInProgress !== null) {⋯
  }

  // Set this to null to indicate there's no in-progress render.
  workInProgressRoot = null;
  workInProgressRootRenderLanes = NoLanes;

  // It's safe to process the queue now that the render phase is complete.
  finishQueueingConcurrentUpdates();

  return workInProgressRootExitStatus;
}
```



DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberWorkLoop.js > ...

```
function prepareFreshStack(root: FiberRoot, lanes: Lanes): Fiber {
  root.finishedWork = null;
  root.finishedLanes = NoLanes;

  const timeoutHandle = root.timeoutHandle;
  if (timeoutHandle !== noTimeout) {⋯
  }
  const cancelPendingCommit = root.cancelPendingCommit;
  if (cancelPendingCommit !== null) {⋯
  }

  resetWorkInProgressStack();
  workInProgressRoot = root;
  const rootWorkInProgress = createWorkInProgress(root.current, null);
  workInProgress = rootWorkInProgress;
  workInProgressRootRenderLanes = lanes;
  workInProgressSuspendedReason = NotSuspended;
  workInProgressThrownValue = null;
  workInProgressRootDidAttachPingListener = false;
  workInProgressRootExitStatus = RootInProgress;
  workInProgressRootFatalError = null;
  workInProgressRootSkippedLanes = NoLanes;
  workInProgressRootInterleavedUpdatedLanes = NoLanes;
  workInProgressRootRenderPhaseUpdatedLanes = NoLanes;
  workInProgressRootPingedLanes = NoLanes;
  workInProgressDeferredLane = NoLane;
  workInProgressRootConcurrentErrors = null;
  workInProgressRootRecoverableErrors = null;
  workInProgressRootDidIncludeRecursiveRenderUpdate = false;

  // Get the lanes that are entangled with whatever we're about to render. We
  // track these separately so we can distinguish the priority of the render
  // task from the priority of the lanes it is entangled with. For example, a
  // transition may not be allowed to finish unless it includes the Sync lane,
  // which is currently suspended. We should be able to render the Transition
  // and Sync lane in the same batch, but at Transition priority, because the
  // Sync lane already suspended.
  entangledRenderLanes = getEntangledLanes(root, lanes);

  finishQueueingConcurrentUpdates();

  return rootWorkInProgress;
}
```

finishQueueingConcurrentUpdates

```javascript
// 把concurrentQueues的内容添加到fiber的queue中
export function finishQueueingConcurrentUpdates(): void {
  const endIndex = concurrentQueuesIndex;
  concurrentQueuesIndex = 0; // 重置

  concurrentlyUpdatedLanes = NoLanes; // 重置

  let i = 0;
  while (i < endIndex) {
    const fiber: Fiber = concurrentQueues[i];

    concurrentQueues[i++] = null;
    const queue: ConcurrentQueue = concurrentQueues[i];
    concurrentQueues[i++] = null;
    const update: ConcurrentUpdate = concurrentQueues[i];
    concurrentQueues[i++] = null;
    const lane: Lane = concurrentQueues[i];
    concurrentQueues[i++] = null;

     // 注意：这里构建完之后的fiber.updateQueue.shared.pending数据类型是Upda
    // 所以fiber.updateQueue.shared.pending其实是指最后一个update，它的nex
    if (queue !== null && update !== null) {
      const pending = queue.pending;
      if (pending === null) {
        // This is the first update. Create a circular list.
        update.next = update;
      } else {
        update.next = pending.next;
        pending.next = update;
      }
      queue.pending = update;
    }


    if (lane !== NoLane) {
      // 更新fiber.lanes
      // 从当前节点开始，往上找到根节点，更新childLanes
      markUpdateLaneFromFiberToRoot(fiber, update, lane);
    }
  }
}
```

markUpdateLaneFromFiberToRoot

从 fiber 开始，逐层往上找到根节点，标记 update。如下：

JavaScript

```
function markUpdateLaneFromFiberToRoot(
  sourceFiber: Fiber,
  update: ConcurrentUpdate | null,
  lane: Lane,
): void {
  // 更新 fiber的lanes
  sourceFiber.lanes = mergeLanes(sourceFiber.lanes, lane);
  let alternate = sourceFiber.alternate;
  if (alternate !== null) {
    alternate.lanes = mergeLanes(alternate.lanes, lane);
  }
  // 从当前节点开始，往上找到根节点，更新childLanes
  let parent = sourceFiber.return;
  let node = sourceFiber;
  while (parent !== null) {
    parent.childLanes = mergeLanes(parent.childLanes, lane);
    alternate = parent.alternate;
    if (alternate !== null) {
      alternate.childLanes = mergeLanes(alternate.childLanes, lane);
    }
    node = parent;
    parent = parent.return;
  }
}
```

5. 处理更新队列 `processUpdateQueue`

这个函数用来处理更新队列。

`processUpdateQueue` 在 `beginWork` 阶段会被两个地方调用：

`updateHostRoot`

react/packages/react-reconciler/src/ReactFiberBeginWork.js 来源：

Flow

```
processUpdateQueue(workInProgress, nextProps, null, renderLanes);
```

`updateClassComponent`

在类组件的 mount、resumeMount、更新阶段，均会调用。

src/react/packages/react-reconciler/src/ReactFiberClassComponent.js

Flow

```
processUpdateQueue(workInProgress, newProps, instance, renderLanes);
```

`processUpdateQueue` 源码

react/packages/react-reconciler/src/ReactFiberClassUpdateQueue.js

此处代码较多，详情查看源码文件。

1. 检查是否有 pending update。

如果有，将它们转移到 baseQueue。

pending update 是个单向循环链表，转移到 单链表 firstBaseUpdate->...->lastBaseUpdate 中去。

2.遍历 queue，根据这些 update 计算出最后的结果

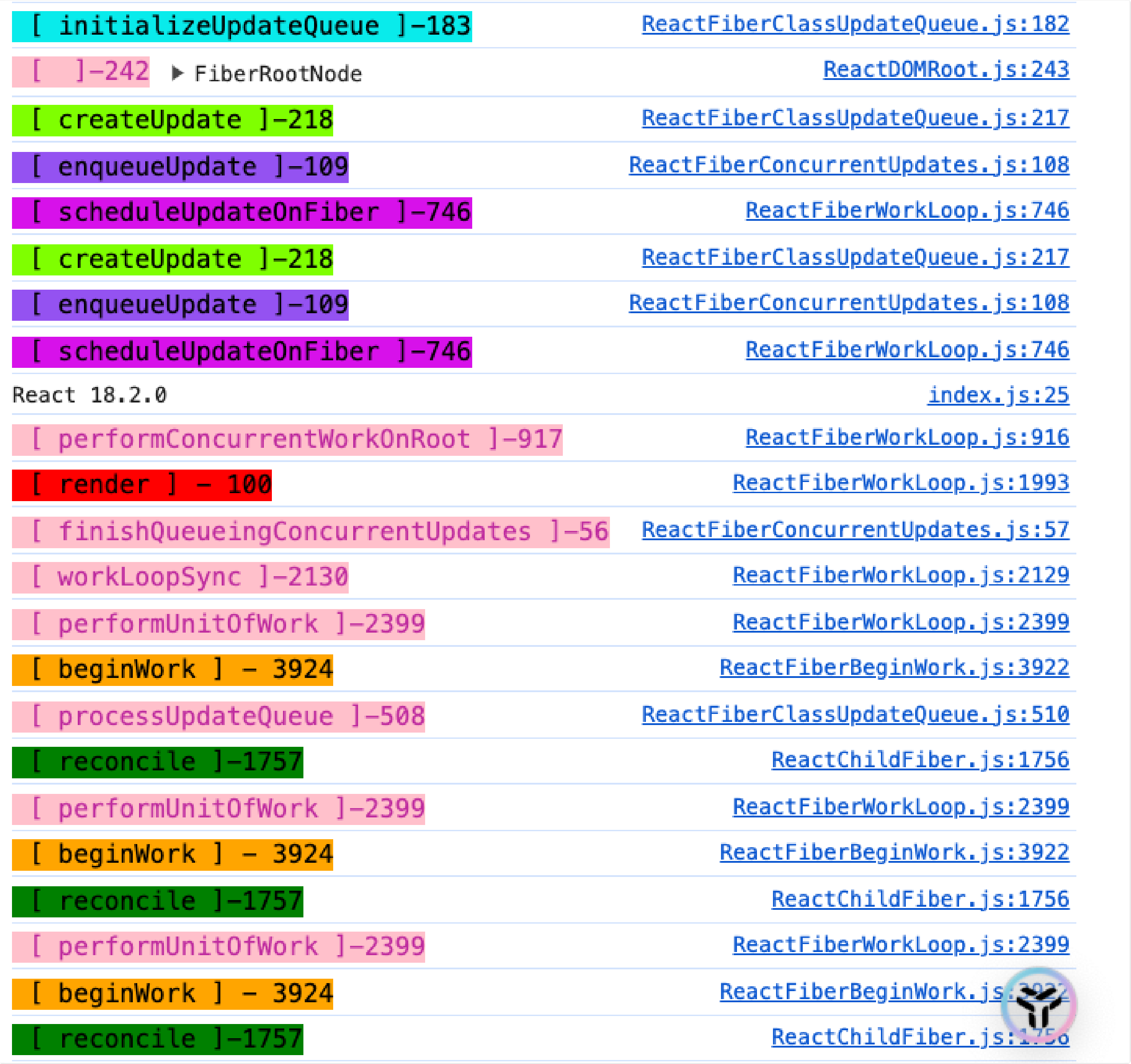
接下来要做的就是遍历 queue，然后根据这些 update，计算出最后的结果。

```Flow
 // 处理这个更新
newState = getStateFromUpdate(
  workInProgress,
  queue,
  update,
  newState,
  props,
  instance,
);
```

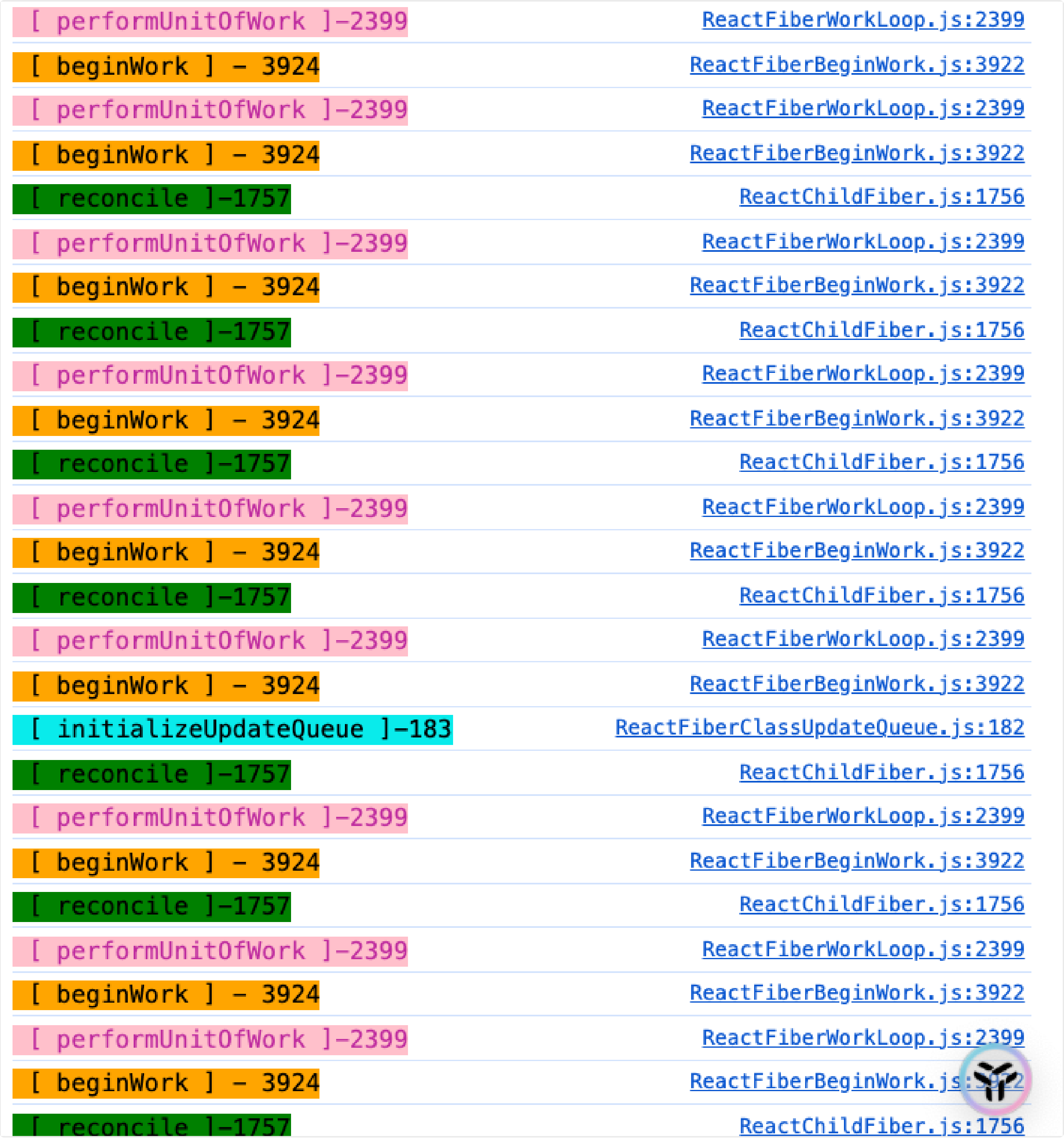
3. 更新到 fiber 上

```Flow
workInProgress.lanes = newLanes;
workInProgress.memoizedState = newState; // 更新状态
```

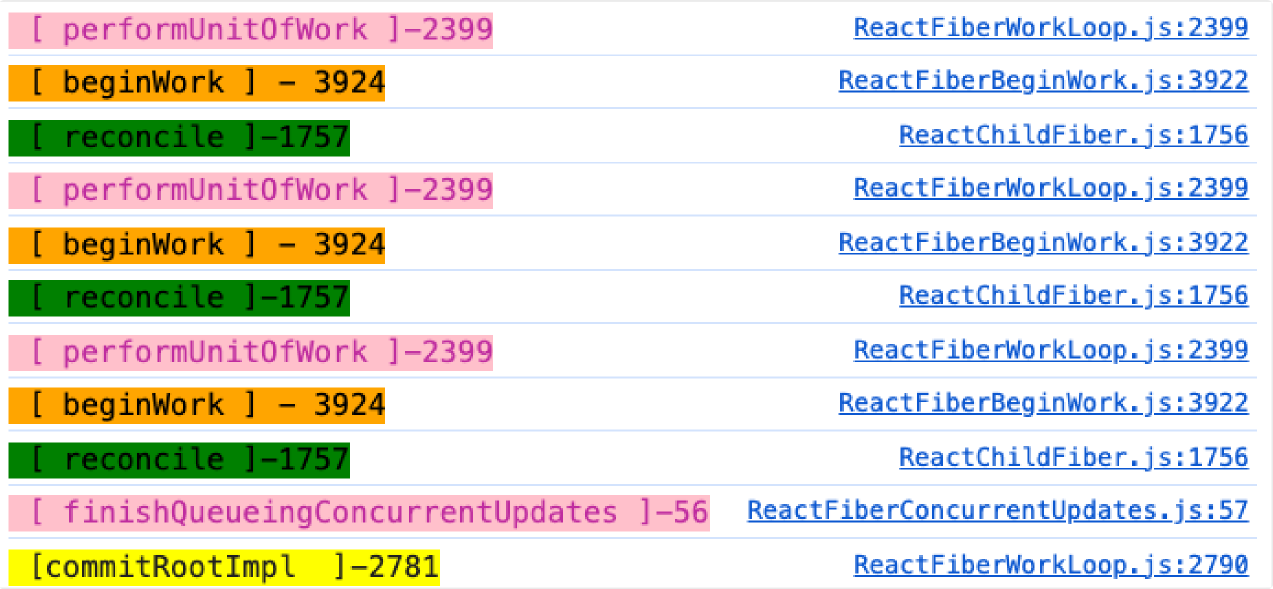
截图



```
[ initializeUpdateQueue ]-183                        ReactFiberClassUpdateQueue.js:182
[  ]-242  ▶ FiberRootNode                                         ReactDOMRoot.js:243
[ createUpdate ]-218                                 ReactFiberClassUpdateQueue.js:217
[ enqueueUpdate ]-109                             ReactFiberConcurrentUpdates.js:108
[ scheduleUpdateOnFiber ]-746                                 ReactFiberWorkLoop.js:746
[ createUpdate ]-218                                 ReactFiberClassUpdateQueue.js:217
[ enqueueUpdate ]-109                             ReactFiberConcurrentUpdates.js:108
[ scheduleUpdateOnFiber ]-746                                 ReactFiberWorkLoop.js:746
React 18.2.0                                                               index.js:25
[ performConcurrentWorkOnRoot ]-917                           ReactFiberWorkLoop.js:916
[ render ] - 100                                             ReactFiberWorkLoop.js:1993
[ finishQueueingConcurrentUpdates ]-56             ReactFiberConcurrentUpdates.js:57
[ workLoopSync ]-2130                                        ReactFiberWorkLoop.js:2129
[ performUnitOfWork ]-2399                                   ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                                        ReactFiberBeginWork.js:3922
[ processUpdateQueue ]-508                           ReactFiberClassUpdateQueue.js:510
[ reconcile ]-1757                                             ReactChildFiber.js:1756
[ performUnitOfWork ]-2399                                   ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                                        ReactFiberBeginWork.js:3922
[ reconcile ]-1757                                             ReactChildFiber.js:1756
[ performUnitOfWork ]-2399                                   ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                                        ReactFiberBeginWork.js:3922
[ reconcile ]-1757                                             ReactChildFiber.js:1756
```



```
[ performUnitOfWork ]-2399                 ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                      ReactFiberBeginWork.js:3922
[ performUnitOfWork ]-2399                 ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                      ReactFiberBeginWork.js:3922
[ reconcile ]-1757                            ReactChildFiber.js:1756
[ performUnitOfWork ]-2399                 ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                      ReactFiberBeginWork.js:3922
[ reconcile ]-1757                            ReactChildFiber.js:1756
[ performUnitOfWork ]-2399                 ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                      ReactFiberBeginWork.js:3922
[ reconcile ]-1757                            ReactChildFiber.js:1756
[ performUnitOfWork ]-2399                 ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                      ReactFiberBeginWork.js:3922
[ reconcile ]-1757                            ReactChildFiber.js:1756
[ performUnitOfWork ]-2399                 ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                      ReactFiberBeginWork.js:3922
[ initializeUpdateQueue ]-183       ReactFiberClassUpdateQueue.js:182
[ reconcile ]-1757                            ReactChildFiber.js:1756
[ performUnitOfWork ]-2399                 ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                      ReactFiberBeginWork.js:3922
[ reconcile ]-1757                            ReactChildFiber.js:1756
[ performUnitOfWork ]-2399                 ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                      ReactFiberBeginWork.js:3922
[ performUnitOfWork ]-2399                 ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                      ReactFiberBeginWork.js:3922
[ reconcile ]-1757                            ReactChildFiber.js:1756
```



```
[ performUnitOfWork ]-2399                          ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                               ReactFiberBeginWork.js:3922
[ reconcile ]-1757                                    ReactChildFiber.js:1756
[ performUnitOfWork ]-2399                          ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                               ReactFiberBeginWork.js:3922
[ reconcile ]-1757                                    ReactChildFiber.js:1756
[ performUnitOfWork ]-2399                          ReactFiberWorkLoop.js:2399
[ beginWork ] - 3924                               ReactFiberBeginWork.js:3922
[ reconcile ]-1757                                    ReactChildFiber.js:1756
[ finishQueueingConcurrentUpdates ]-56   ReactFiberConcurrentUpdates.js:57
[commitRootImpl  ]-2781                             ReactFiberWorkLoop.js:2790
```

index.js

JavaScript

```javascript
import jsx from "./pages/ExamplePage";
import ClassFunctionComponent from "./pages/ClassFunctionComponent";

const root = createRoot(document.getElementById("root"));
root.render(<ClassFunctionComponent />);
root.render(jsx);
```

ExamplePage

React JSX

```jsx
import {
  Component,
  useState,
  useReducer,
  useEffect,
  useLayoutEffect,
} from "../whichReact";

class ClassComponent extends Component {
  state = { count: 0 };
  render() {
    return (
      <div className="class border">
        {this.props.name}
        <button
          onClick={() => {
            this.setState({ count: this.state.count + 1 });
            this.setState({ count: this.state.count + 2 });
          }}
        >
          {this.state.count}
        </button>
      </div>
    );
  }
}

function FunctionComponent(props) {
  const [count1, setCount1] = useReducer((x) => x + 1, 0);

  useEffect(() => {
    return () => {
      console.log("销毁");
    };
  }, []);

  return (
    <div className="border">
      <p>{props.name}</p>
      <button
        onClick={() => {
          setCount1();
        }}
      >
        {count1}
      </button>
    </div>
  );
}

const jsx = (
  <div className="box border">
```

```
    <h1 className="border">omg</h1>
    123
    <FunctionComponent name="函数组件" />
    <ClassComponent name="class组件" />
    <>
      <h1>1</h1>
      <h1>2</h1>
    </>
  </div>
);

export default jsx;

// document.createDocumentFragment

// ! 原生节点 有对应的dom节点
// 1. 原生标签节点 div\span\a等 HostComponent
// 2. 文本节点

// ! 非原生节点 没有对应的dom节点
// 函数组件、类组件、Provider、Consumer、Fragment等
```