[React 源码] useState | useReducer | 相关面试题 [2.4k 字 - 阅读时长5min]

> 代码都来自 React 18 源码， 大家可以放心食用

附加面试题：

- 为什么不能在条件和循环里使用Hooks?
- 为什么不能在函数组件外部使用Hooks?
- React Hooks的状态保存在了哪里?

useReducer 原理

问题: 下面这段代码，从挂载到更新发生了什么？怎么挂载的？怎么更新的。

```javascript
const reducer = (state, action) => {
  if (action.type === "add") return state + 1;
  else return state;
};

function Counter() {
  const [number, dispatch0] = useReducer(reducer, 0);
  const [number1, dispatch1] = useReducer(reducer, 0);
  return (
    <div
      onClick={() => {
        dispatch({ type: "add" });
        dispatch1({ type: "add" });
      }}
    >
      {number}
    </div>
  );
}
```

> 这里，我们直接进入到 reconciler 阶段，默认已经通过深度优先调度到了 Counter 函数组件的 Fiber节点

useReducer mount 挂载阶段

第一：判断是函数节点的 tag 之后，调用 renderWithHooks.

```javascript
/*
workInProgress: 当前工作的 Fiber 节点
Componet: Counter 函数组件
_current: 老 Fiber 节点 也就是 workInProgress.alternate
*/
let value = renderWithHooks(_current,workInProgress,Component);
```

第二：在 renderWithHooks 当中调用 Counter 函数

```javascript
let children = Component();
```

第三： 调用 Counter 函数 的 useReducer 函数

```javascript
export function useReducer(reducer, initialArg) {
  return ReactCurrentDispatcher.current.useReducer(reducer, initialArg);
}
```

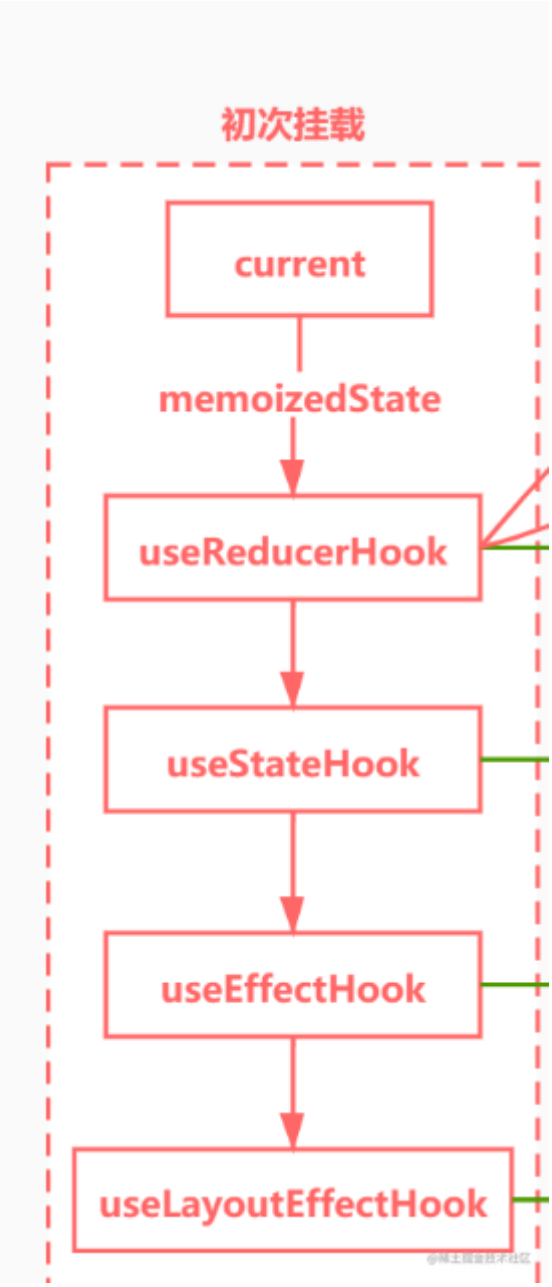
第四：挂载阶段 `ReactCurrentDispatcher.current.useReducer` 实则是调用了 `mountReducer`，

第五：在 `mountReducer` 中调用， `mountWorkInProgressHook` 函数，创建 `useReducer` 的 `Hook` 对象，构建 `fiber.memoizedState` 也就是 Hook 链表, 然后将 `dispatchAction` bind 绑定之后传入 `queue` 和 `fiber` 为参数，这点很重要。并且初始化 Hook 的更新对象的队列 queue。

```javascript
function mountReducer<S, I, A>(
  reducer: (S, A) => S,
  initialArg: I,
  init?: I => S,
): [S, Dispatch<A>] {
  const hook = mountWorkInProgressHook();
  let initialState;
  if (init !== undefined) {
    initialState = init(initialArg);
```

```
  } else {
    initialState = ((initialArg: any): S);
  }
  hook.memoizedState = hook.baseState = initialState;
  const queue: UpdateQueue<S, A> = {
    pending: null,
    lanes: NoLanes,
    dispatch: null,
    lastRenderedReducer: reducer,
    lastRenderedState: (initialState: any),
  };
  hook.queue = queue;
  const dispatch: Dispatch<A> = (queue.dispatch = (dispatchReducerAction.bind(
    null,
    currentlyRenderingFiber,
    queue,
  ): any));
  return [hook.memoizedState, dispatch];
}
```

如果之后再有 useState useReducer，最终mout阶段的成果是



自此 useReducer 挂载阶段执行完毕

---

useReducer update 更新阶段

第一：通过 dispatch(action) 触发更新， `dispatch` 就是 `dispatchAction.bind(null, fiber, queue)` 返回的绑定函数。所以相当于调用了 `dispatchAction(fiber, queue, action)` 。

第二： 更新时 dispatchAction 调用 `scheduleUpdateOnFiber` ，`enqueueConcurrentHookUpdate`

```javascript
function dispatchReducerAction<S, A>(
  fiber: Fiber,
  queue: UpdateQueue<S, A>,
```

```
  action: A,
): void {
  const lane = requestUpdateLane(fiber);
  const update: Update<S, A> = {
    lane,
    action,
    hasEagerState: false,
    eagerState: null,
    next: (null: any),
  };

  if (isRenderPhaseUpdate(fiber)) {
    enqueueRenderPhaseUpdate(queue, update);
  } else {
    const root = enqueueConcurrentHookUpdate(fiber, queue, update, lane);
    if (root !== null) {
      const eventTime = requestEventTime();
      scheduleUpdateOnFiber(root, fiber, lane, eventTime);
      entangleTransitionUpdate(root, queue, lane);
    }
  }
}
```

`enqueueConcurrentHookUpdate` 函数 中的 `enqueueUpdate` 将 `hook` 更新时产生的对象 update，放入 `queue.pending` 当中，例如 一个 reducer 的 多次 dispatch, `update` 会组成队列。

javascript 复制代码

```
export function enqueueConcurrentHookUpdate<S, A>(
  fiber: Fiber,
  queue: HookQueue<S, A>,
  update: HookUpdate<S, A>,
  lane: Lane,
): FiberRoot | null {
  const concurrentQueue: ConcurrentQueue = (queue: any);
  const concurrentUpdate: ConcurrentUpdate = (update: any);
  enqueueUpdate(fiber, concurrentQueue, concurrentUpdate, lane);
  return getRootForUpdatedFiber(fiber);
}
}
```

`scheduleUpdateOnFiber` 函数，从根节点出发，重新开始调度更新。

第三：我们直接进入到 `reconciler` 阶段，默认已经通过深度优先更新调度到了 `Counter` 函数组件的 `Fiber` 节点

第四：判断是函数节点的 tag 之后，调用 renderWithHooks.

```javascript
/*
workInProgress: 当前工作的 Fiber 节点
Componet：Counter 函数组件
_current: 老 Fiber 节点 也就是 workInProgress.alternate
*/
let value = renderWithHooks(_current,workInProgress,Component);
```

第五：在 renderWithHooks 当中调用 Counter 函数

```javascript
let children = Component();
```

第六： 调用 Counter 函数 的 useReducer 函数

```javascript
export function useReducer(reducer, initialArg) {
  return ReactCurrentDispatcher.current.useReducer(reducer, initialArg);
}
```

第七：更新阶段 `ReactCurrentDispatcher.current.useReducer` 实则是调用了 `updateReducer`

第八：在 `updateReducer` 中调用 `updateWorkInProgressHook` 函数，在此函数中最重要的就是通过 alternate 指针复用 `currentFiber` （老 Fiber） 的 memorizedState, 也就是 Hook 链表，并且按照严格的对应顺序来复用 `currentFiber` （老 Fiber） Hook 链表当中的 Hook（通过 currentHook 指针结合链表来实现），一个比较重要的复用就是去复用老 Hook 的更新队列 `queue`，因为 `dispatchAction.bind` 绑定的就是 `currentFiber` （老 Fiber），通过尽可能的复用来创建新的 Hook 对象，构建 `fiber.memoizedState` 也就是 Hook 链表。

> 注意：这里也就是为什么不能再循环和判断当中使用 Hook 的重要原因。 一句话：要保持严格的顺序一致。
>
> 读到这儿我们发现，无论是 `Mout Hook` 还是 `updateHook` 都有严格的顺序，如果顺序乱了，更新阶段就不会正确复用到在 `currentFiber` Hook 链表当中的 Hook 的更新队列 queue,也就不能通过更新得到正确的 `state` . 再严重些，useState的更新逻辑，对应的 currentHook 是 useEffect Hook, 无法兼容复用，导致报错。

```javascript
function updateWorkInProgressHook(): Hook {
    const newHook: Hook = {
```

```
      memoizedState: currentHook.memoizedState,
      baseState: currentHook.baseState,
      baseQueue: currentHook.baseQueue,
      queue: currentHook.queue,
      next: null,
    };

  return workInProgressHook;
}
```

第九：在 `updateReducer` 中调用，遍历整个更新队列 `queue.pending`,取出 `update` 对象的 action 通过 `newState = reducer(newState, action)`；返回 新状态 和 queue.dispatch（还是 `dispatchAction.bind(null, Currentfiber, queue)`。

javascript 复制代码

```
function updateReducer<S, I, A>(
  reducer: (S, A) => S,
  initialArg: I,
  init?: I => S,
): [S, Dispatch<A>] {
  const hook = updateWorkInProgressHook();
  const queue = hook.queue;
  queue.lastRenderedReducer = reducer;
  const current: Hook = (currentHook: any);
  let baseQueue = current.baseQueue;
  // The last pending update that hasn't been processed yet.
  const pendingQueue = queue.pending;

  if (baseQueue !== null) {
    // We have a queue to process.
    const first = baseQueue.next;
    let newState = current.baseState;

    let newBaseState = null;
    let newBaseQueueFirst = null;
    let newBaseQueueLast = null;
    let update = first;
    do {
        // Process this update.
        const action = update.action;
        if (update.hasEagerState) {
          // If this update is a state update (not a reducer) and was processed eagerly,
          // we can use the eagerly computed state
          newState = ((update.eagerState: any): S);
        } else {
          newState = reducer(newState, action);
        }
      }
```

```
      update = update.next;
    } while (update !== null && update !== first);

    if (newBaseQueueLast === null) {
      newBaseState = newState;
    } else {
      newBaseQueueLast.next = (newBaseQueueFirst: any);
    }

    hook.memoizedState = newState;
    hook.baseState = newBaseState;
    hook.baseQueue = newBaseQueueLast;
    queue.lastRenderedState = newState;
  }
```

自此 useReducer 更新完毕

---

useState 原理

问题: 下面这段代码，从挂载到更新发生了什么？怎么挂载的？怎么更新的。

```javascript 复制代码
function Counter() {
  const [number, setNumber] = useState(reducer, 0);
  return (
    <div
      onClick={() => {
        setNumber({ type: "add" });
      }}
    >
      {number}
    </div>
  );
}
```

> 这里，我们直接进入到 reconciler 阶段，默认已经通过深度优先调度到了 Counter 函数组件的 Fiber节点

---

useState mount 挂载阶段

```javascript
let children = Component();
```

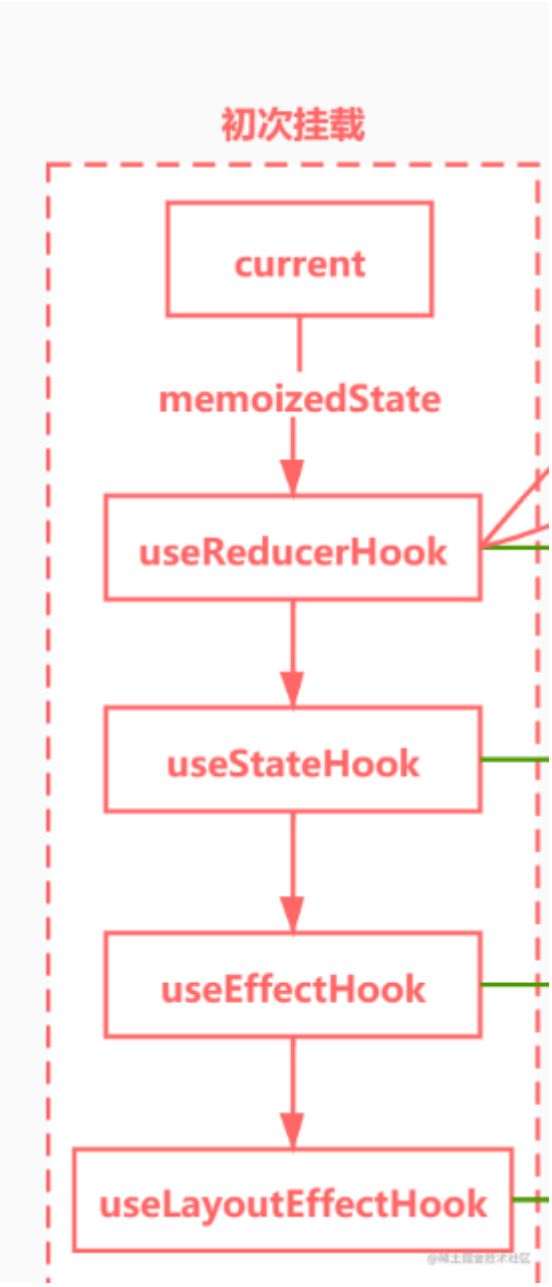
第三： 调用 Counter 函数的第一个 useState 函数

```javascript
export function useEffect(create, deps) {
  return ReactCurrentDispatcher.current.useState(create, deps);
}
```

第四：挂载阶段 `ReactCurrentDispatcher.current.useState` 实则是调用了 `moutState`，`moutState` 中调用， `mountWorkInProgressHook` 函数，创建 `useState` 的 `Hook` 对象，构建 `fiber.memoizedState` 也就是 Hook 链表, 然后将 `dispatchAction` bind 绑定之后传入 `queue` 和 `fiber` 为参数，这点很重要。并且初始化 Hook 的更新对象的队列 queue。

```javascript
function mountState<S>(
  initialState: (() => S) | S,
): [S, Dispatch<BasicStateAction<S>>] {
  const hook = mountWorkInProgressHook();
  if (typeof initialState === 'function') {
    // $FlowFixMe: Flow doesn't like mixed types
    initialState = initialState();
  }
  hook.memoizedState = hook.baseState = initialState;
  const queue: UpdateQueue<S, BasicStateAction<S>> = {
    pending: null,
    lanes: NoLanes,
    dispatch: null,
    lastRenderedReducer: basicStateReducer,
    lastRenderedState: (initialState: any),
  };
  hook.queue = queue;
  const dispatch: Dispatch<
    BasicStateAction<S>,
  > = (queue.dispatch = (dispatchSetState.bind(
    null,
    currentlyRenderingFiber,
    queue,
  ): any));
  return [hook.memoizedState, dispatch];
}
```

如果之后再有 useState useReducer，最终mount阶段的成果如下图：



自此 useState 挂载阶段执行完毕

---

useState update 更新阶段

第一：通过 dispatch(action) 触发更新，`dispatch` 就是 `dispatchAction.bind(null, fiber, queue)` 返回的绑定函数。所以相当于调用了 `dispatchAction(fiber, queue, action)`。

第二： 更新时 dispatchAction 调用 `scheduleUpdateOnFiber`，`enqueueConcurrentHookUpdate`

```javascript
function dispatchReducerAction<S, A>(
  fiber: Fiber,
  queue: UpdateQueue<S, A>,
```

```
  action: A,
): void {
  const lane = requestUpdateLane(fiber);
  const update: Update<S, A> = {
    lane,
    action,
    hasEagerState: false,
    eagerState: null,
    next: (null: any),
  };

  if (isRenderPhaseUpdate(fiber)) {
    enqueueRenderPhaseUpdate(queue, update);
  } else {
    const root = enqueueConcurrentHookUpdate(fiber, queue, update, lane);
    if (root !== null) {
      const eventTime = requestEventTime();
      scheduleUpdateOnFiber(root, fiber, lane, eventTime);
      entangleTransitionUpdate(root, queue, lane);
    }
  }
}
```

`enqueueConcurrentHookUpdate` 函数 中的 `enqueueUpdate` 将 `hook` 更新时产生的对象 update，放入 `queue.pending` 当中，例如 一个 useState 的多次 setStae, `update` 会组成队列。

javascript 复制代码

```
export function enqueueConcurrentHookUpdate<S, A>(
  fiber: Fiber,
  queue: HookQueue<S, A>,
  update: HookUpdate<S, A>,
  lane: Lane,
): FiberRoot | null {
  const concurrentQueue: ConcurrentQueue = (queue: any);
  const concurrentUpdate: ConcurrentUpdate = (update: any);
  enqueueUpdate(fiber, concurrentQueue, concurrentUpdate, lane);
  return getRootForUpdatedFiber(fiber);
}
}
```

`scheduleUpdateOnFiber` 函数，从根节点出发，重新开始调度更新。

第三：我们直接进入到 `reconciler` 阶段，默认已经通过深度优先更新调度到了 `Counter` 函数组件的 `Fiber` 节点

第四：判断是函数节点的 tag 之后，调用 renderWithHooks.

```javascript
/*
workInProgress: 当前工作的 Fiber 节点
Componet：Counter 函数组件
_current: 老 Fiber 节点 也就是 workInProgress.alternate
*/
let value = renderWithHooks(_current,workInProgress,Component);
```

第五：在 renderWithHooks 当中调用 Counter 函数

```javascript
let children = Component();
```

第六： 调用 Counter 函数 的 useState 函数

```javascript
export function useState(reducer, initialArg) {
  return ReactCurrentDispatcher.current.useState(reducer, initialArg);
}
```

第七：更新阶段 `ReactCurrentDispatcher.current.useState` 实则是调用了 `updateReducer`

> useState 更新的时候，调用的还是 updateReducer, 说明 useState 本质就是 useReducer. 也是 useReducer 的语法糖。 将 basicStateRecuer 作为 reducer 函数。

```javascript

function updateState<S>() {
  return updateReducer(basicStateReducer, (initialState: any));
}

function basicStateReducer<S>(state: S, action: BasicStateAction<S>): S {
  // $FlowFixMe: Flow doesn't like mixed types
  return typeof action === 'function' ? action(state) : action;
}
```

第八：在 `updateReducer` 中调用 `updateWorkInProgressHook` 函数，在此函数中最重要的就是通过 alternate 指针复用 `currentFiber` （老 Fiber） 的 memorizedState, 也就是 Hook 链表，并且按照严格的对应顺序来复用 `currentFiber` （老 Fiber） Hook 链表当中的 Hook（通过 currentHook 指针结合链表来实现），一个比较重要的复用就是复用老 Hook 的更新队列

`queue`，因为 `dispatchAction.bind` 绑定的就是 `currentFiber`（老 Fiber），通过尽可能的复用来创建新的 Hook 对象，构建 `fiber.memoizedState` 也就是 Hook 链表。

javascript 复制代码

```
function updateWorkInProgressHook(): Hook {
    const newHook: Hook = {
      memoizedState: currentHook.memoizedState,
      baseState: currentHook.baseState,
      baseQueue: currentHook.baseQueue,
      queue: currentHook.queue,
      next: null,
    };

  return workInProgressHook;
}
```

第九：在 `updateReducer` 中调用，遍历整个更新队列 `queue.pending`,取出 `update` 对象的 action 通过 `newState = reducer(newState, action)`；返回 新状态 和 queue.dispatch（还是 `dispatchAction.bind(null, Currentfiber, queue)`。

javascript 复制代码

```
function updateReducer<S, I, A>(
  reducer: (S, A) => S,
  initialArg: I,
  init?: I => S,
): [S, Dispatch<A>] {
  const hook = updateWorkInProgressHook();
  const queue = hook.queue;
  queue.lastRenderedReducer = reducer;
  const current: Hook = (currentHook: any);
  let baseQueue = current.baseQueue;
  // The last pending update that hasn't been processed yet.
  const pendingQueue = queue.pending;

  if (baseQueue !== null) {
    // We have a queue to process.
    const first = baseQueue.next;
    let newState = current.baseState;

    let newBaseState = null;
    let newBaseQueueFirst = null;
    let newBaseQueueLast = null;
    let update = first;
    do {
        // Process this update.
        const action = update.action;
        if (update.hasEagerState) {
```

```
          // If this update is a state update (not a reducer) and was processed eagerly,
          // we can use the eagerly computed state
          newState = ((update.eagerState: any): S);
        } else {
          newState = reducer(newState, action);
        }
      }
      update = update.next;
    } while (update !== null && update !== first);

    if (newBaseQueueLast === null) {
      newBaseState = newState;
    } else {
      newBaseQueueLast.next = (newBaseQueueFirst: any);
    }

    hook.memoizedState = newState;
    hook.baseState = newBaseState;
    hook.baseQueue = newBaseQueueLast;
    queue.lastRenderedState = newState;
  }
```

自此 useState 更新完毕
