从源码的角度回答 React Hooks 面试题

杭州已经不看健康码了，某种程度上来说YQ已经结束了，但某种程度上YQ又刚刚开始。无论是谁在这个冬天都要保护好自己，明年春暖花开的时候，肯定就是真正的春天到了

闲言少叙，开始正文

前言

上篇文章《都React V18了，还不会正确使用React Hooks吗，万字长文解析Hooks的常见问题》说到了React Hooks的一些常见问题，本文让我们继续深入react hooks，结合几个常见的面试题，从源码角度分析一下react hooks，从源码的角度回答一下这些问题。

本文的React源码基于最新的版本 V18.2.0

带着问题看源码

从React V16.8 发布以来 Hooks API 以来，社区出现了很多针对Hooks源码分析的文章。但是源码阅读本就是一个比较枯燥的事情，而且React源码也相对复杂，涉及了诸如Fiber、双缓冲之类相对不好整体理解的新概念，源码阅读起来没有那么顺畅。React Hooks又是面试会经常被问到的一个点，本文笔者不会长篇铺上代码，会结合几个面试中常见的问题，对 React Hooks 的源码进行分析，带着问题去分析源码，尽可能的使用更规范的答案，来回答这些面试题，同时对React Hooks源码有更深入的理解。

常见的React Hooks面试题

为什么可以在Function Component中使用使用私有状态

Function Component 和 Class Component 不同，Function Component 不能实例化，所有就不可以将私有状态挂在到实例上，组件每次重新渲染都会把Function重新执行一遍，函数内的变量就会重新初始化，这些状态在React内是怎么维护的呢

```
// 组件re-render的时候name，setName保存在哪里
const Comp = () => {
  const [name, setName] = uesState('');
  return <>{name}</>
}
```

问题分析

这个问题应该是所有人在使用hooks api的第一个疑惑，为什么React Function Component每次渲染之后都能返回的之前的值，毋庸置疑的是React 肯定通过某种机制记录下来了更新操作，并将其结果记录并返回给下一次渲染了。但是这些状态记录在哪里了呢？这里先卖个关子，先看看下一个React Hooks的问题

为什么不要在循环，条件或嵌套函数中调用hooks

React官方文档中明确说明了这条Hooks调用顺序的限制

> 不要在循环，条件或嵌套函数中调用 Hook， 确保总是在你的 React 函数的最顶层以及任何 return 之前调用Hooks

假如我们这样使用了就会遇到eslint的提醒，比如下面的  :

```
const Comp = (props) => {
  if(props.disable) {
    return null;
  }
  const [name, setName] = useState('');
  return <>{name}</>
}
```

上面的例子不出意外的会出现一个警告 `React Hook "useState" is called conditionally. React Hooks muse be called in the exact same order in every render`

这种用法也很常见，但是为什么React Hooks就能不兼容呢

问题分析

这个问题Rudi Yardley有个回答笔者感觉特别好：React hooks: not magic, just arrays；因为hooks维护了一系列的数组，假如在不同的render返回的返回的hooks不同，内部游标就无法做到正确的匹配。

这个问题也刚好匹配到了第一个问题，函数组件的状态就是维护到了这些数组中，根据组件内书写的hooks的顺序，按顺序返回，这就解释了为什么Function Component 可以记录私有变量。

下面从React源码的角度看看这个问题

源码解读

首先我们不关注不相关的部分，react的源码逻辑很深，假如过于关注某些细节很容易陷入其中，找不到重点，我们直接跳到关键位置

```js
// react-reconciler/src/ReactFiberHooks.js
// HooksDispatcherOnMount，挂载阶段的useState
function mountState<S>() {
  const hook = mountWorkInProgressHook();
  // ...
  hook.memoizedState = hook.baseState = initialState;
  const queue: UpdateQueue<S, BasicStateAction<S>> = {
    // ...
  };
  const dispatch = dispatchSetState.bind();
  return [hook.memoizedState, dispatch];
}

// HooksDispatcherOnUpdate，更新阶段的useState
function updateState<S>() {
  return updateReducer(basicStateReducer, (initialState: any));
  function updateReducer(){
      const hook = updateWorkInProgressHook();
      const queue = hook.queue;
      const dispatch: Dispatch<A> = (queue.dispatch: any);
      return [hook.memoizedState, dispatch];
    }
}

// 关键点指向mountWorkInProgressHook
function mountWorkInProgressHook(): Hook {
  // 创建一个节点
  const hook: Hook = {
    memoizedState: null,
```

```
    baseState: null,
    baseQueue: null,
    queue: null,

    next: null,
  };
  // 判断这是不是链表的第一个节点
  if (workInProgressHook === null) {
    // 若这是链表的第一个节点，将链表指针memoizedState指向当前hook
    // currentlyRenderingFiber 是 workInProgress 指向的 fiber 节点，workInProgress本文就不做分解了，后续
    currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
  } else {
    // 若这不是链表的第一个节点，则直接添加节点放到列表的最后
    workInProgressHook = workInProgressHook.next = hook;
  }
  return workInProgressHook;
}
```

在mount阶段，每调用一次创建 hook 的函数，不论是什么 hook，只要是在这个函数组件内定义的，都会添加到workInProgressHook链表中，**一个Function Component中定义的所有的hooks节点都会放到链表内，并存放在通过指针memoizedState保持引用，可以从该属性中获取链表的指针。**这些 hooks 的调用顺序，其实就是其添加在链表上的顺序。在re-render时，也会按照添加的顺序来执行，所以需要hooks在函数顶部声明，同时不能在判断语句中声明，否则可能会引起hooks的错乱

思考

组件在挂在阶段是通过mountWorkInProgressHook来生成hooks并挂载到链表上的，我们都知道组件的生命周期内，还包含更新，在mounted阶段，初始节点已经把所有的 hooks 都挂载在链表中了，在update阶段的更新操作是怎么处理的呢，这里不做详细说明了，可以去看看 `updateWorkInProgressHook` 。

hooks的依赖跟踪是怎么实现的

你写代码的时候会不会遇到过在Effect中依赖了某个state，但是忘记将其加入到依赖中了，比如：

js 复制代码

```
const Comp = () => {
  const [keyword, setKeyword] = useState('');
  const [pageNumber, setPageNumber] = useState(0);
```

```
  useEffect(() => {doFetch(keyword, pageNumber)}, [pageNumber])

  return <>{keyword}</>
}
```

上述的代码会造成useEffect在keyword变化的时候不会执行fetch，代码执行的时机或者执行结果会出现异常。那为什么useEffect中依赖的state必须加到依赖项中呢

其实useEffect(), useMemo(), useCallback() 等hooks中，第2个参数是依赖项，react是怎么实现依赖跟踪的呢

源码分析

我们还是直接前往关键代码位置

js 复制代码

```
// /packages/react-reconciler/src/ReactFiberHooks.js
function updateEffectImpl(
  // ...
  create: () => (() => void) | void,
  deps: Array<mixed> | void | null,
) {
  const hook = updateWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
    const prevEffect = currentHook.memoizedState;
    if (nextDeps !== null) {
      const prevDeps = prevEffect.deps;
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        hook.memoizedState = pushEffect(hookFlags, create, destroy, nextDeps);
        return;
      }
    }

  // ...
  currentlyRenderingFiber.flags |= fiberFlags;
  hook.memoizedState = pushEffect(hookFlags, create, destroy, nextDeps);
}
```

从上面的代码可以看出，假如没有设置依赖项，或设置的依赖项为 null，则该 hook 每次渲染时都会执行；若依赖项任何一项都没有变化，使用上一次渲染的结果。依赖项目的判断是怎么做的呢

```js 复制代码
function areHookInputsEqual(
  nextDeps: Array<mixed>,
  prevDeps: Array<mixed> | null,
): boolean {
  if (prevDeps === null) {
    return false;
  }

  // $FlowFixMe[incompatible-use] found when upgrading Flow
  for (let i = 0; i < prevDeps.length && i < nextDeps.length; i++) {
    // $FlowFixMe[incompatible-use] found when upgrading Flow
    if (is(nextDeps[i], prevDeps[i])) {
      continue;
    }
    return false;
  }
  return true;
}
```

在HooksDispatcherOnUpdate阶段，上面的代码会比较两个依赖项中的每一项是否有变化，若任意的一项变化，就返回 false，表示两个依赖项不相等, hooks会重新执行；若都一样，则hooks还使用之前缓存的数据，而不会去执行effect，这就是hooks依赖跟踪机制

useCallback和useMemo的区别是什么

首先useCallback我们经常用在包裹一个函数，避免组件re-render的时候生成一个新的函数，引起子组件re-render；一般我们会将useCallback和React.memo结合使用，可以参考一下上篇文章中关于《用useCallback肯定能提升性能吗？》的部分。而useMemo一般用于复杂的计算，优化一些不必要的计算，比如计算的相关参数都没有变化的时候，直接返回上一次的计算结果就好了，避免出现在当前组件中进行冗余的计算

可以看看这个例子

```js 复制代码
const Children = React.memo((props) => <button onClick={props.doSubmit}>提交</button>);

export default function Demo() {
  const [number, setNumber] = useState(0);

  const doSubmit = useCallback(() => {
    console.log(`Number: ${number}`); // 每次输出都是初始值
  }, [number]); // 把`text`写在依赖数组里

  const memoValue = React.useMemo(() => {
```

```
    return Array(100000).fill('').map(v => /*数据计算*/ v);
  }, [number]);

  return (
    <>
      <input value={number} onChange={(e) => setNumber(e.target.value)} />
      <Children doSubmit={doSubmit} />
    </>
  );
}
```

分析

useCallback和useMemo几乎完全一样，唯一的区别就是useCallback根据第二个参数（依赖项）缓存并返回第一个参数（回调函数），useMemo是根据第二个参数（依赖项）缓存并返回第一个参数（回调函数）的执行结果，**两者一个缓存的回调函数，一个缓存的时回调函数的执行结果**

源码分析

在上述的分析中，我们可以发现所有的 hooks 在内部实现时，都会区分mount阶段和update阶段，useCallback和useMemo也是会区分的，我们主要关心一下update阶段的代码，依赖更新部分的能力可以参考上面的分析，这不做过多解释，让我们就直观的看看useCallback和useMemo的区别


```
function updateCallback<T>(callback: T, deps: Array<mixed> | void | null): T {
  const hook = updateWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
  const prevState = hook.memoizedState;
  if (areHookInputsEqual(nextDeps, prevDeps)) {
        return prevState[0];
  }
  // 返回函数
  hook.memoizedState = [callback, nextDeps];
  return callback;
}

function updateMemo<T>(
  nextCreate: () => T,
  deps: Array<mixed> | void | null,
): T {
  const hook = updateWorkInProgressHook();
```

```
  const nextDeps = deps === undefined ? null : deps;
  const prevState = hook.memoizedState;
  if (areHookInputsEqual(nextDeps, prevDeps)) {
    return prevState[0];
  }
  // 计算结果
  if (shouldDoubleInvokeUserFnsInHooksDEV) {
    nextCreate();
  }
  const nextValue = nextCreate();
  hook.memoizedState = [nextValue, nextDeps];
  return nextValue;
}
```

可以直观的看到 `hook.memoizedState` 保存的分别是callback和nextValue，这样就可以解释两者一个缓存的回调函数，一个缓存的时回调函数的执行结果这个结果了

useEffect和useLayoutEffect执行时机有什么区别

先看看官方文档对这两个hooks的描述，先看看useEffect:

> 该 Hook 接收一个包含命令式、且可能有副作用代码的函数。使用 useEffect 完成副作用操作。赋值给 useEffect 的函数会在组件渲染到屏幕之后执行。默认情况下，effect 将在每轮渲染结束后执行，但你可以选择让它在只有某些值改变的时候才执行。

再看看useLayoutEffect:

> 其函数签名与 useEffect 相同，但它会在所有的 DOM 变更之后同步调用 effect。可以使用它来读取 DOM 布局并同步触发重渲染。在浏览器执行绘制之前，useLayoutEffect 内部的更新计划将被同步刷新。

源码分析

下面的代码可以看出，useEffect和useLayoutEffect的调用的函数其实是同一个，那两者的区别是什么呢

js 复制代码

```
function updateLayoutEffect(
  create: () => (() => void) | void,
  deps: Array<mixed> | void | null,
```

```
): void {
  return updateEffectImpl(UpdateEffect, HookLayout, create, deps);
}
```

其实关键点就是updateEffectImpl第一个参数fiberFlags的控制，分别是 `PassiveEffect` 和 `UpdateEffect` ,现在让给我们看一下两者在调度器Scheduler内的执行时机

React的一次状态更新，可以简单概括为两个阶段，构造fiber树（render）和渲染fiber树（commit），为了直观，我们依旧不去关心优先级，不去关心调度器的实现，只关注useEffect和useLayoutEffect回掉函数执行时机，当workInProgressFiber树构建完成，就会进入commit阶段，useEffect的副作用才会开始执行或销毁，commit阶段的内部执行也是很复杂但是不是这个问题重点，让我们直接跳过commit阶段不相关的部分，前往Effect的执行部分

commit的入口函数是 `commitRoot --> commitRootImpl`

js 复制代码

```
function commitRootImpl(
  root: FiberRoot,
  recoverableErrors: null | Array<CapturedValue<mixed>>,
  transitions: Array<Transition> | null,
  renderPriorityLevel: EventPriority,
) {
  // 异步调度effect
  scheduleCallback(NormalSchedulerPriority, () => {
    flushPassiveEffects();
    return null;
  });
  // Before Mutation阶段
  const shouldFireAfterActiveInstanceBlur = commitBeforeMutationEffects(
      root,
      finishedWork,
    );

 // Mutation阶段
  commitMutationEffects(root, finishedWork, lanes);
 // Layout阶段
  commitLayoutEffects(finishedWork, root, lanes);

  flushSyncCallbacks();
}
```

可以明确的是，useLayoutEffect会在commit阶段中同步执行，回调函数会更早一点执行，可以在useLayoutEffects中进行一些可能影响dom的操作，其create中可以获取到最新的dom树

且由于此时浏览器未进行绘制，这个时候操作dom可以避免一起一些浏览器的抖动行为，这部分逻辑就不再深入了，后续可以针对react的更新进行一次彻底的源码分析

思考

这个问题其实还有个延伸问题，useEffect和useLayoutEffect和Class Component中的生命周期如何对应等等，这个可以看看源码思考一下