

React Internals

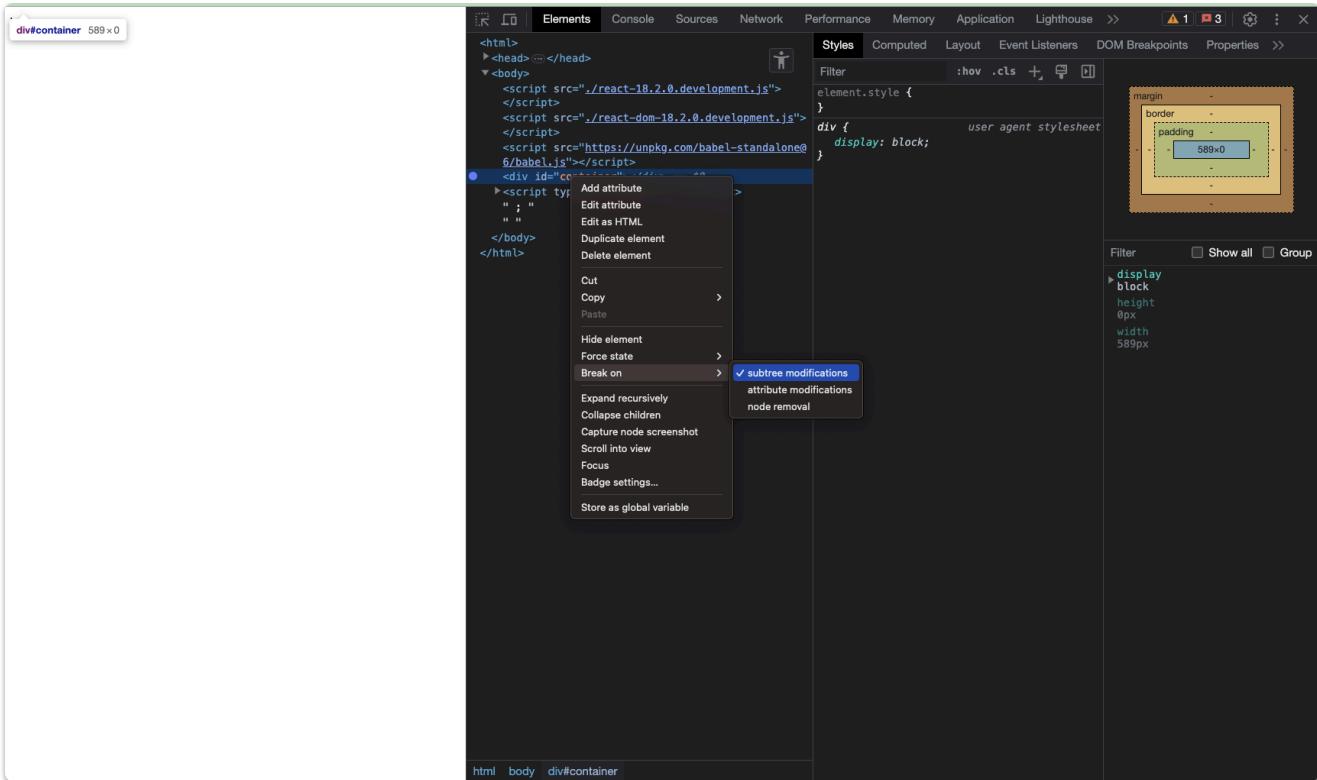
How does React work under the hood ? The Overview of React internals?

Since React is a UI library, one important task is to find the code where DOM is manipulated and then we can read the call stack to figure out what is happening. Here I can just create a DOM breakpoint on the DOM container, like below.

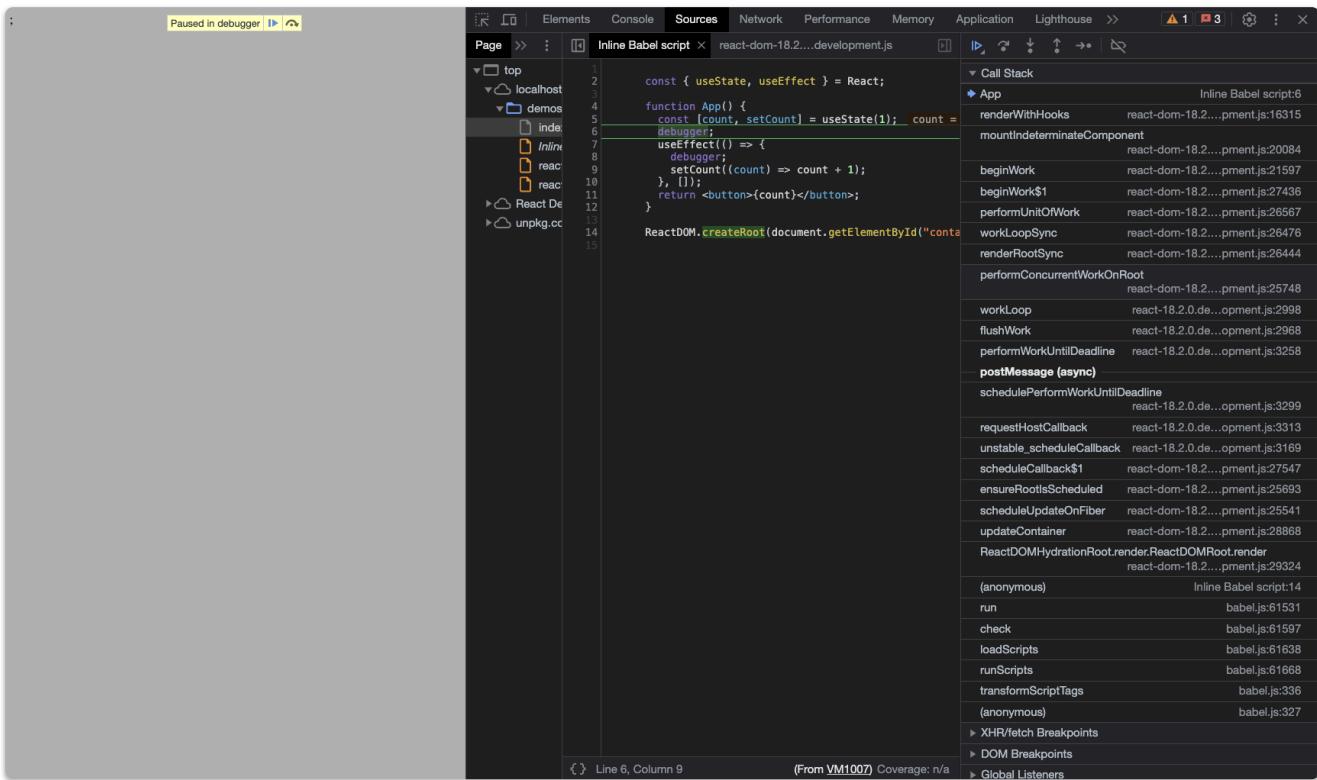
code Snippets

```
1 import { useEffect, useState } from "react";
2
3 function App(){
4   const [count, setCount] = useState(1);
5   debugger;
6   useEffect(() => {
7     debugger;
8
9     setCount((prev) => prev + 1);
0   }, []);
1   return <button>{count}</button>;
2
3 }
4 export default App
```

Inspect view



First pause at rendering the component



Below are some important functions from call stack.

1. `ReactDOM.render()` → this is the user-side code we write, we first `createRoot()` and then `render()`
2. `scheduleUpdateOnFiber()` → this tells React where to render, for initial mount there is no previous version so it is called on root.
3. `ensureRootIsScheduled()` → this important call is to “ensure” `performConcurrentWorkOnRoot()` is scheduled.

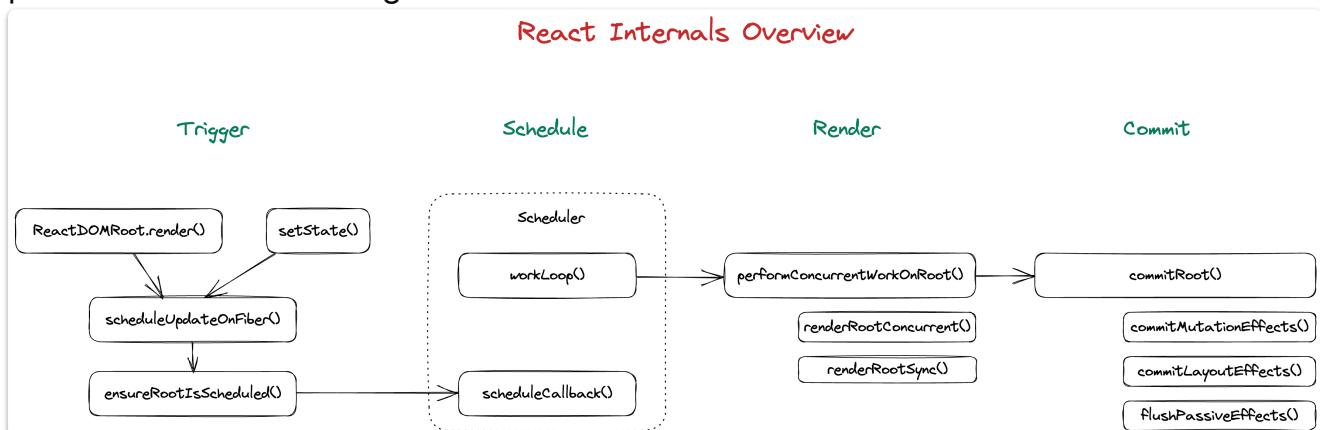
4. `scheduleCallback()` → the actual scheduling, which is part of [React Scheduler](#), see from screenshot that it is async by `postMessage()`.
5. `workLoop()` → how [React Scheduler](#) processes the tasks.
6. `performConcurrentWorkOnRoot()` → the scheduled task is now being run, in which our component is actually rendered.

Second pause at DOM manipulation

The screenshot shows a browser developer tools debugger with the "Call Stack" tab selected. The stack trace starts with `appendChildToContainer` in `react-dom-18.2...pment.js:11079`. It then branches into `insertOrAppendPlacementNodeIntoContainer` and `commitPlacement`, both in `react-dom-18.2...pment.js:23885`. This leads to `commitReconciliationEffects` in `react-dom-18.2...pment.js:24601`, `commitMutationEffectsOnFiber` in `react-dom-18.2...pment.js:24304`, and `recursivelyTraverseMutationEffects` in `react-dom-18.2...pment.js:24283`. Finally, it reaches `commitMutationEffectsOnFiber` in `react-dom-18.2...pment.js:24442`, `commitMutationEffects` in `react-dom-18.2...pment.js:24253`, `commitRootImpl` in `react-dom-18.2...pment.js:26820`, `commitRoot` in `react-dom-18.2...pment.js:26692`, `finalConcurrentRender` in `react-dom-18.2...pment.js:25991`, `performConcurrentWorkOnRoot` in `react-dom-18.2...pment.js:25819`, `workLoop` in `react-18.2.0.de...pment.js:2998`, `flushWork` in `react-18.2.0.de...pment.js:2968`, `performWorkUntilDeadline` in `react-18.2.0.de...pment.js:3258`, and `postMessage (async)`. The main code being debugged is from `react-dom-18.2...pment.js`, specifically the `commitTextUpdate` and `appendChild` methods.

In our `useEffect()` we call `setState()` to trigger re-render, from the call stack we can see that the whole re-render is quite similar to the first breakpoint pause we got, except that inside `performConcurrentWorkOnRoot()`, it is `updateFunctionComponent()` rather than `mountIndeterminateComponent()`.

In React source code, `mount` means the initial render, since in initial render there is no previous version to diff against.



Trigger

I gave it the name “Trigger” since all work is kicked off here. No matter it is initial mount or re-render caused by state hook. In this phase we are telling React runtime which part of the app needs to be rendered(`scheduleUpdateOnFiber()`) and how it should be done.

We can think of this phase as “create a task”, `ensureRootIsScheduled()` is the last step of creating such task and then task is sent to Scheduler by `scheduleCallback()`.

Schedule

This is the React Scheduler, it is basically a [Priority Queue](#) that processes the tasks by priority. `scheduleCallback()` is called in runtime code to schedule tasks like rendering or running effects. `workLoop()` inside Scheduler is how tasks are actually run.

Render

Render is the task scheduled (`performConcurrentWorkOnRoot()`), it means calculating the new Fiber Tree and figure out what updates are needed to apply to host DOM. We don’t need to know details of Fiber Tree here, it is basically an internal tree-like structure that represents current state of our app. It was called Virtual DOM before, but it is not only for DOM now and React team doesn’t call it Virtual DOM any more. So `performConcurrentWorkOnRoot()` is created in Trigger phase, prioritized in Scheduler, and then actually run here. Think of it as if there is a little man, who walks around the Fiber Tree and checks if they need to re-render and figures out the necessary updates on host DOM.

Because of concurrent mode, “Render” phase might be interrupted and resumed, which makes it a quite complex step.

Commit

After new Fiber Tree is constructed and the minimum updates are derived, it is time to “commit” the updates to host DOM.<https://jser.dev/2023-06-19-how-does-usestate-work/>

Of course there are more than just manipulating the DOM(`commitMutationEffects()`). For example, all kinds of Effects are processed here as well (`flushPassiveEffects()` , `commitLayoutEffects()`).

<https://jser.dev/2023-06-19-how-does-usestate-work/>

later on

