Profiler API

The `Profiler` measures how often a React application renders and what the "cost" of rendering is. Its purpose is to help identify parts of an application that are slow and may benefit from optimizations such as memoization.

> **Note:**
> Profiling adds some additional overhead, so **it is disabled in the production build**.
>
> To opt into production profiling, React provides a special production build with profiling enabled. Read more about how to use this build at fb.me/react-profiling

Usage

A `Profiler` can be added anywhere in a React tree to measure the cost of rendering that part of the tree. It requires two props: an `id` (string) and an `onRender` callback (function) which React calls any time a component within the tree "commits" an update.

For example, to profile a `Navigation` component and its descendants:

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Main {...props} />
  </App>
);
```

Multiple `Profiler` components can be used to measure different parts of an application.

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Profiler id="Main" onRender={callback}>
      <Main {...props} />
    </Profiler>
  </App>
);
```

`Profiler` components can also be nested to measure different components within the same subtree:

```
render(
  <App>
    <Profiler id="Panel" onRender={callback}>
      <Panel {...props}>
        <Profiler id="Content" onRender={callback}>
          <Content {...props} />
        </Profiler>
        <Profiler id="PreviewPane" onRender={callback}>
          <PreviewPane {...props} />
        </Profiler>
      </Panel>
    </Profiler>
  </App>
);
```

> **Note**
> Although `Profiler` is a light-weight component, it should be used only when necessary; each use adds some CPU and memory overhead to an application.

---

onRender Callback

The `Profiler` requires an `onRender` function as a prop. React calls this function any time a component within the profiled tree "commits" an update. It receives parameters describing what was rendered and how long it took.

```
function onRenderCallback(
  id, // the "id" prop of the Profiler tree that has just committed
  phase, // either "mount" (if the tree just mounted) or "update" (if it re-rendered)
  actualDuration, // time spent rendering the committed update
  baseDuration, // estimated time to render the entire subtree without memoization
  startTime, // when React began rendering this update
  commitTime, // when React committed this update
  interactions // the Set of interactions belonging to this update
) {
  // Aggregate or log render timings...
}
```

Let's take a closer look at each of the props:

- **`id: string`** - The `id` prop of the `Profiler` tree that has just committed. This can be used to identify which part of the tree was committed if you are using multiple profilers.
- **`phase: "mount" | "update"`** - Identifies whether the tree has just been mounted for the first time or re-rendered due to a change in props, state, or hooks.
- **`actualDuration: number`** - Time spent rendering the `Profiler` and its descendants for the current update. This indicates how well the subtree makes use of memoization (e.g. `React.memo`, `useMemo`, `shouldComponentUpdate`). Ideally this value should decrease significantly after the initial mount as many of the descendants will only need to re-render if their specific props change.
- **`baseDuration: number`** - Duration of the most recent `render` time for each individual component within the `Profiler` tree. This value estimates a worst-case cost of rendering (e.g. the initial mount or a tree with no memoization).
- **`startTime: number`** - Timestamp when React began rendering the current update.
- **`commitTime: number`** - Timestamp when React committed the current update. This value is shared between all profilers in a commit, enabling them to be grouped if desirable.
- **`interactions: Set`** - Set of "interactions" that were being traced when the update was scheduled (e.g. when `render` or `setState` were called).

Note

Interactions can be used to identify the cause of an update, although the API for tracing them is still experimental.

Learn more about it at fb.me/react-interaction-tracing

Is this page useful? Edit this page