Boilerplate , use case, bottlenecks, verbose, overhead,bail out , compose,lazy initializing, shallow rendering

Use state, use Effect,useMemo,useReducer, useCallback,React.lazy():

Roadmap

New context API

New router Navi

immer

Profiler:

it can be handy to track performance regressions over time in larger apps

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](https://reactjs.org/docs/hooks-faq.html#how-to-memoize-calculations).

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](https://reactjs.org/docs/profiler.html" \l "onrender-callback) (function) which React calls any time a component within the tree “commits” an update.

render(

<Profiler id="application" onRender={onRenderCallback}>

<App>

<Navigation {...props} />

<Main {...props} />

</App>

</Profiler>

);

Profiling adds some additional overhead, so **it is disabled in**[**the production build**](https://reactjs.org/docs/optimizing-performance.html#use-the-production-build).

React cache

New Lifecycle method:

getDerivedStateFromProps

suspense: (data fetching, code splitting and other async data dependencies): lazy loading with suspense

* pause any state update until the data is ready
* add async data to any component without plumbing
* on a fast network, render after the whole tree is ready
* on a slow network , precisely control the loading states.

Time splicing:

* React doesn’t block thread while rendering
* Feels synchronous if the device is fast
* Feels synchronous if the device is slow
* Only the final rendered state is displayed

React 16.5 version Profiler in dev tools:

What’s happening in the app and optimize them.

Hooks:

eslint-plugin-react-hooks

**State Updates May Be Asynchronous**

React may batch multiple setState() calls into a single update for performance.

Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.

For example, this code may fail to update the counter:

// Wrong

this.setState({

counter: this.state.counter + this.props.increment,

});

To fix it, use a second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

// Correct

this.setState((state, props) => ({

counter: state.counter + props.increment

}));

### Pure functions:

function sum(a, b) {

return a + b;

}

Such functions are called [“pure”](https://en.wikipedia.org/wiki/Pure_function) because they do not attempt to change their inputs, and always return the same result for the same inputs.