**useState()**

React 16.8 and onward, it enables developers to add stateful logic to functional component .useState is a hook that lets us add state to our functional component. It returns a array with two elements

* the current state value
* a function that allows us to update the state value

import React, {useState} from "react"

const feature = () => {

// declare state in functional component

const [state, setState] = useState(initialState)

  return (

    <div>feature</div>

  )

}

export default feature

**state**

The current state value

**setState**

a function to update the state value.

**initialState**

the initial state value, which can be any type

Problem before hooks

Before hooks, state management in react was exclusively handled by the class component. This posed several issue

* boilerplate code

Class component require more boilerplate code, such as constructor and method bindings

* component complexity
* reusability

**useState works internally**

1. **initialization**

when a component first renders, useState hook is called during the initial render to set up the state for the component

* **initial value**

the argument passed to useState is used as the initial state value.

* **state queue**

react keeps a queue of state updates for each component. When useState is called, react checks if there are any pending state updates in this queue to determine the current state.

1. **rendering and reconciliation**

* **virtual dom**

React renders the component and generates a virtual dom based on the current state.

* **reconciliation**

react compares the virtual DOM with the previous version to determine what changes are needed. This process is known as reconciliation.

1. **state updates**

* triggering rerenders

when setState is called, it doesn’t immediately update state. Instead, it schedules a state update. React then trigger a rerenders of the component to apply the new state.

* batching

react may batch multiple state updates together to improve performance. If multiple setState calls occur in a single event handler, react might batch them into a single rerender.

1. **rendering with updated state**

* **component re-renders**

after the state is updated, react rerenders the component with the new state value. This updated state is passed to the virtual DOM, and reconciliation happens again to update the real DOM.

* **dom update**

only the parts of the DOM that need to be changed are updated based on the difference between the previous and new virtual DOM.

1. **unmounting and cleanup**

* **unmounting**

when a component is removed from the DOM, react cleans up the state and any associated side effects.

* **garbage collection**

once a component is unmounted, the associated state and effects are eligible for garbage collection, freeing up memory.

**When does it updates the DOM**

* **Initial render**

On the first render, react constructs the DOM based on the initial state.

* **State change**

When setState is called, react schedules a rerender. During this rerender, react reconciles the changes and updates the DOM accordingly.

**When does it create and remove state**

* **creation**

state is created during the initial render of the component, when useState is first called.

* **Removal**

State is removed when the component is unmounted, and react performs cleanup

**useEffect()**

**useLayoutEffect()**

It is similar to useEffect in that it allows us to perform side effects in function components. However, it differs in when its executed. useLayoutEffect runs synchronously after all DOM mutations but before the browser updates the screen, making it suitable for tasks that requires immediate DOM measurements / mutations.

If we update the DOM within useLayoutEffect the user will don’t see the intermediate state between the DOM update and the next paint.

Features

**Synchronous execution**, unlike useEffect, which runs asynchronously and after the paint, useLayoutEffect runs synchronously after the DOM is updated but before the paint.

**Blocking,** because it runs synchronously, it can block the browser from painting if it takes a long tim, potentially leading to performance issues if don’t used carefully.

**Access to DOM,** useLayoutEffect is ideal when we need to perform DOM measurements / make changes to the DOM before the browser paints the screen.

**Usecases**

**Reading layout information,** if we need to measure the size / position of elements after a render and then immediately make changes happen before the browser paints.

**Updating DOM based on measurements,** when a component appearance needs to be updated based on its dimension / position, useLayoutEffect can ensure that these updates are applied without the user seeing any flicker / incomplete rendering

**Synchronous animation setup,** if we need to prepare animations / other visual updates that need precise tim, ‘useLayoutEffect’ can ensure everything is set before the frame is painted.

**Working**

**Render phase,** react renders the component and creates the virtual DOM.

**DOM update** the dom is updated with any changes from the render.

**useLayoutEffect execution,** after the DOM update but before the browser paints the screen, useLayoutEffect runs synchronously. This allows for any necessary DOM manipulation, such as reading layout information / applying styles based on the updated DOM.

**Paint,** the browser paints the screen, showing the updated UI to the user.

**when to use useLayoutEffect**

Critical DOM updates, when we need to make DOM updates that must be visible without any delay, such as adjusting layout / applying styles based on measurements.

**Preventing flicker,** when performing operations that, if done in useEffect would cause a flicker because they would be applied after the paint.

**Handling synchronous animations,** when setting up animation that need precise control over tim relative to rendering and DOM updates.

Performance consideration

Blocking point, because it runs synchronously, it can block the browser from painting, leading to potential performance issues.

Don’t for side effects, use useEffect for non ui related side effects to avoid blocking the paint