# react hooks

## react hooks 解决了什么问题？

先设想一下，如果没有 Hooks，函数组件能够做的只是接受 Props、渲染 UI ，以及触发父组件传过来的事件。所有的处理逻辑都要在类组件中写，这样会使 class 类组件内部错综复杂，每一个类组件都有一套独特的状态，相互之间不能复用，即便是 React 之前出现过 mixin 等复用方式，但是伴随出 mixin 模式下隐式依赖，代码冲突覆盖等问题，也不能成为 React 的中流砥柱的逻辑复用方案。所以 React 放弃 mixin 这种方式。

类组件是一种面向对象思想的体现，类组件之间的状态会随着功能增强而变得越来越臃肿，代码维护成本也比较高，而且不利于后期 tree shaking。所以有必要做出一套函数组件代替类组件的方案，于是 Hooks 也就理所当然的诞生了。

### 所以 Hooks 出现本质上原因是：

* 让函数组件也能做类组件的事，有自己的状态，可以处理一些副作用，能获取 ref ，也能做数据缓存。
* 解决逻辑复用难的问题。
* 放弃面向对象编程，拥抱函数式编程。

### 为什么要使用自定义 Hooks ？

自定义 hooks 是在 React Hooks 基础上的一个拓展，可以根据业务需求制定满足业务需要的组合 hooks ，更注重的是逻辑单元。通过业务场景不同，到底需要React Hooks 做什么，怎么样把一段逻辑封装起来，做到复用，这是自定义 hooks 产生的初衷。

自定义 hooks 也可以说是 React Hooks 聚合产物，其内部有一个或者多个 React Hooks 组成，用于解决一些复杂逻辑。

## 常用的hooks

### useState

允许在函数组件中使用状态。使用useState() 声明一个状态变量，并使用它来存储组件的状态。每次更改状态时，组件将重新渲染。

const [number, setNumber] = useState(0)

### useReducer

是useState() hook的替代品，用于管理更复杂的状态。它使用Reducer函数来管理组件状态，Reducer函数接收当前状态和要进行的操作，然后返回新状态。

const [ number , dispatchNumbner ] = useReducer((state,action)=>{

const { payload , name } = action

/\* return的值为新的state \*/

switch(name){

case 'add':

return state + 1

case 'sub':

return state - 1

case 'reset':

return payload

}

return state

},0)

dispatchNumbner({ name:'add' })

dispatchNumbner({ name:'sub' })

dispatchNumbner({ name:'reset' ,payload:666 })

### useSyncExternalStore

能够让 React 组件在 concurrent 模式下安全地有效地读取外接数据源，在组件渲染过程中能够检测到变化，并且在数据源发生变化的时候，能够调度更新。当读取到外部状态发生了变化，会触发一个强制更新，来保证结果的一致性。

① subscribe 为订阅函数，当数据改变的时候，会触发 subscribe，在 useSyncExternalStore 会通过带有记忆性的 getSnapshot 来判别数据是否发生变化，如果发生变化，那么会强制更新数据。

② getSnapshot 可以理解成一个带有记忆功能的选择器。当 store 变化的时候，会通过 getSnapshot 生成新的状态值，这个状态值可提供给组件作为数据源使用，getSnapshot 可以检查订阅的值是否改变，改变的话那么会触发更新。

③ getServerSnapshot 用于 hydration 模式下的 getSnapshot。

useSyncExternalStore(

subscribe,

getSnapshot,

getServerSnapshot

)

/\* 订阅外部数据源 \*/

const userInfo = useSyncExternalStore(store.userInfo,() => store.getState().userInfo)

### useTransition

在 React v18 中，有一种新概念叫做过渡任务，这种任务是对比立即更新任务而产生的，通常一些影响用户交互直观响应的任务，例如按键，点击，输入等，这些任务需要视图上立即响应，所以可以称之为立即更新的任务，但是有一些更新不是那么急迫，比如页面从一个状态过渡到另外一个状态，这些任务就叫做过渡任务。 打个比方当点击 tab 从 tab1 切换到 tab2 的时候，本质上产生了两个更新任务。

1. 第一个就是 hover 状态由 tab1 变成 tab2。
2. 第二个就是内容区域由 tab1 内容变换到 tab2 内容。

这两个任务，用户肯定希望 hover 状态的响应更迅速，而内容的响应有可能还需要请求数据等操作，所以更新状态并不是立马生效，通常还会有一些 loading 效果。所以**第一个任务作为立即执行任务，而第二个任务就可以视为过渡任务**。

useTransition 执行返回一个数组。数组有两个状态值：

* 第一个是，当处于过渡状态的标志——isPending。
* 第二个是一个方法，可以理解为上述的 startTransition。可以把里面的更新任务变成过渡任务。

import { useTransition } from 'react'

/\* 使用 \*/

const [ isPending , startTransition ] = useTransition ()

除了上述切换 tab 场景外，还有很多场景非常适合 useTransition 产生的过渡任务，比如输入内容，实时搜索并展示数据，这本质上也是有两个优先级的任务，第一个任务就是受控表单的实时响应；第二个就是输入内容改变,数据展示的变化。

import React from 'react';

const mockList1 = new Array(10000).fill('tab1').map((item, index) => item + '--' + index)

const mockList2 = new Array(10000).fill('tab2').map((item, index) => item + '--' + index)

const mockList3 = new Array(10000).fill('tab3').map((item, index) => item + '--' + index)

const tab = {

tab1: mockList1,

tab2: mockList2,

tab3: mockList3

}

function TestUseTransition() {

const [active, setActive] = React.useState('tab1') //需要立即响应的任务，立即更新任务

const [renderData, setRenderData] = React.useState(tab[active]) //不需要立即响应的任务，过渡任务

**const [isPending, startTransition] = React.useTransition()**

const handleChangeTab = (activeItem) => {

setActive(activeItem) // 立即更新

**startTransition(() => { // startTransition 里面的任务优先级低**

**setRenderData(tab[activeItem])**

**})**

}

return (

<div className='TestUseTransition'>

<div className='tab' >

{Object.keys(tab).map((item) => <span className={active === item && 'active'} onClick={() => handleChangeTab(item)} >{item}</span>)}

</div>

<ul className='content' >

{**isPending** && <div> loading... </div>}

{renderData.map(item => <li key={item} >{item}</li>)}

</ul>

</div>

)

}

TestUseTransition.displayName = 'TestUseTransition';

export default TestUseTransition;

### useDeferredValue

React 18 提供了 useDeferredValue 可以让状态滞后派生。useDeferredValue 的实现效果也类似于 transtion，当迫切的任务执行后，再得到新的状态，而这个新的状态就称之为 DeferredValue。

#### useDeferredValue 和上述 useTransition 本质上有什么异同呢？

相同点： useDeferredValue 本质上和内部实现与 useTransition 一样都是**标记成了过渡更新任务**。

不同点： useTransition 是把 startTransition 内部的更新任务变成了过渡任务transtion,而 useDeferredValue 是把原值通过过渡任务得到新的值，这个值作为延时状态。 **一个是处理一段逻辑，另一个是生产一个新的状态**。

useDeferredValue 接受一个参数 value ，一般为可变的 state , 返回一个延时状态 deferrredValue。

const deferrredValue = React.useDeferredValue(value)

基础用法:

function TestUseDeferredValue(props: IProps) {

const [active, setActive] = React.useState('tab1') //需要立即响应的任务，立即更新任务

**const deferActive = React.useDeferredValue(active)** // 把状态延时更新，类似于过渡任务

const handleChangeTab = (activeItem) => {

setActive(activeItem) // 立即更新

}

**const renderData = tab[deferActive]** // 使用滞后状态

return <div>

<div className='tab'>

{Object.keys(tab).map((item) => <span className={active === item && 'active'} onClick={() => handleChangeTab(item)} >{item}</span>)}

</div>

<ul className='content' >

{renderData.map(item => <li key={item} >{item}</li>)}

</ul>

</div>

}

TestUseDeferredValue.displayName = 'TestUseDeferredValue';

export default TestUseDeferredValue;

### useEffect

用于处理副作用。副作用指在React组件之外进行的操作，例如从服务器获取数据，处理DOM元素等。使用useEffect() hook，您可以执行此类操作，而无需在类组件中编写生命周期方法。

useEffect(()=>{

return destory

},dep)

### useLayoutEffect

useLayoutEffect 和 useEffect 不同的地方是采用了同步执行，那么和useEffect有什么区别呢？

① 首先 **useLayoutEffect 是在 DOM 更新之后，浏览器绘制之前**，这样可以方便修改 DOM，获取 DOM 信息，这样浏览器只会绘制一次，如果修改 DOM 布局放在 useEffect ，那 useEffect 执行是在浏览器绘制视图之后，接下来又改 DOM ，就可能会导致浏览器再次回流和重绘。而且由于两次绘制，视图上可能会造成闪现突兀的效果。

② useLayoutEffect callback 中代码执行会**阻塞浏览器绘制**。

const DemoUseLayoutEffect = () => {

const target = useRef()

useLayoutEffect(() => {

/\*我们需要在dom绘制之前，移动dom到制定位置\*/

const { x ,y } = getPositon() /\* 获取要移动的 x,y坐标 \*/

animate(target.current,{ x,y })

}, []);

return (

<div >

<span ref={ target } className="animate"></span>

</div>

)

}

### useInsertionEffect

useInsertionEffect 是在 React v18 新添加的 hooks ，它的用法和 useEffect 和 useLayoutEffect 一样。那么这个 hooks 用于什么呢?

在介绍 useInsertionEffect 用途之前，先看一下 useInsertionEffect 的执行时机。

React.useEffect(()=>{

console.log('useEffect 执行')

},[])

React.useLayoutEffect(()=>{

console.log('useLayoutEffect 执行')

},[])

React.useInsertionEffect(()=>{

console.log('useInsertionEffect 执行')

},[])

打印： useInsertionEffect 执行 -> useLayoutEffect 执行 -> useEffect 执行

可以看到 useInsertionEffect 的执行时机要比 useLayoutEffect 提前，useLayoutEffect 执行的时候 DOM 已经更新了，但是在 useInsertionEffect 的执行的时候，DOM 还没有更新。**本质上 useInsertionEffect 主要是解决 CSS-in-JS 在渲染中注入样式的性能问题**。这个 hooks 主要是应用于这个场景，在其他场景下 React 不期望用这个 hooks 。

### useContext

允许您在React中使用上下文。上下文是一种在组件树中传递数据的方法，可以避免通过Props一层层传递数据。使用useContext() hook，您可以访问整个应用程序中定义的上下文对象。

### useRef

useRef 可以用来**获取元素，缓存状态**，接受一个状态 initState 作为初始值，返回一个 ref 对象 cur, cur 上有一个 current 属性就是 ref 对象需要获取的内容。

### useImperativeHandle

useImperativeHandle 可以配合 forwardRef 自定义暴露给父组件的实例值。这个很有用，我们知道，对于子组件，如果是 class 类组件，我们可以通过 ref 获取类组件的实例，但是在子组件是函数组件的情况，如果我们不能直接通过 ref 的，那么此时 useImperativeHandle 和 forwardRef 配合就能达到效果。

useImperativeHandle 接受三个参数：

1. 第一个参数ref: 接受 forWardRef 传递过来的 ref。
2. 第二个参数 createHandle ：处理函数，返回值作为暴露给父组件的 ref 对象。
3. 第三个参数 deps : 依赖项 deps ，依赖项更改形成新的 ref 对象。

### useMemo

**派生新状态,缓存计算结果**

用于缓存计算结果，以避免在每次渲染时重新计算。这非常有用，特别是当计算成本很高时。

useMemo 可以在函数组件 render 上下文中同步执行一个函数逻辑，这个函数的返回值可以作为一个新的状态缓存起来。那么这个 hooks 的作用就显而易见了：

场景一：在一些场景下，需要在函数组件中进行大量的逻辑计算，那么我们不期望每一次函数组件渲染都执行这些复杂的计算逻辑，所以就需要在 useMemo 的回调函数中执行这些逻辑，然后把得到的产物（计算结果）缓存起来就可以了。

场景二：React 在整个更新流程中，diff 起到了决定性的作用，比如 Context 中的 provider 通过 diff value 来判断是否更新

### useCallback

useMemo 和 useCallback 接收的参数都是一样，都是在其依赖项发生变化后才执行，都是返回缓存的值，区别在于 useMemo 返回的是函数运行的结果，useCallback 返回的是函数，这个回调函数是经过处理后的也就是说父组件传递一个函数给子组件的时候，由于是无状态组件每一次都会重新生成新的 props 函数，这样就使得每一次传递给子组件的函数都发生了变化，这时候就会触发子组件的更新，这些更新是没有必要的，此时我们就可以通过 usecallback 来处理此函数，然后作为 props 传递给子组件。

### useDebugValue

useDebugValue 可用于在 React 开发者工具中显示自定义 hook 的标签。这个hooks目的就是检查自定义hooks。

function useFriendStatus(friendID) {

const [isOnline, setIsOnline] = useState(null);

// ...

// 在开发者工具中的这个 Hook 旁边显示标签

// e.g. "FriendStatus: Online"

useDebugValue(isOnline ? 'Online' : 'Offline');

return isOnline;

}

### useId

useId 也是 React v18 产生的新的 hooks , 它可以在 client 和 server 生成唯一的 id , 解决了在服务器渲染中，服务端和客户端产生 id 不一致的问题，更重要的是保障了 React v18 中 streaming renderer （流式渲染） 中 id 的稳定性。

**低版本 React ssr 存在的问题：**

比如在一些项目或者是开源库中用 Math.random() 作为 ID 的时候，可以会有一些随机生成 id 的场景：

const rid = Math.random() + '\_id\_' /\* 生成一个随机id \*/

function Demo (){

// 使用 rid

return <div id={rid} ></div>

}

这在纯客户端渲染中没有问题，但是在服务端渲染的时候，传统模式下需要走如下流程：

服务器获取数据 -> 服务器渲染到html模板 -> 客户端加载html等代码 -> hydrate

在这个过程中，当服务端渲染到 html 和 hydrate 过程分别在服务端和客户端进行，但是会走两遍 id 的生成流程，这样就会造成 id不一致的情况发生。useId 的出现能有效的解决这个问题。

基本语法

function Demo (){

const rid = useId() // 生成稳定的 id

return <div id={rid} ></div>

}

## useInsertionEffect,useLayoutEffect,useEffect执行顺序

useInsertionEffect 执行在DOM 还没有更新前

**本质上 useInsertionEffect 主要是解决 CSS-in-JS 在渲染中注入样式的性能问题**

useLayoutEffect 执行在dom更新后,浏览器渲染之前

* 1.首先 **useLayoutEffect 是在 DOM 更新之后，浏览器绘制之前**，这样可以方便修改 DOM，获取 DOM 信息，这样浏览器只会绘制一次，如果修改 DOM 布局放在 useEffect ，那 useEffect 执行是在浏览器绘制视图之后，接下来又改 DOM ，就可能会导致浏览器再次回流和重绘。而且由于两次绘制，视图上可能会造成闪现突兀的效果。
* 2.useLayoutEffect callback 中代码执行会**阻塞浏览器绘制**。

useEffect 执行是在浏览器绘制视图之后

## 什么是useCallback和useMemo？

useCallback 和 useMemo 都是react可用于性能优化的内置hooks。

两者的区别在于：useCallback缓存的是一个函数，而useMemo缓存的是计算结果。

### 为什么使用useCallback和useMemo?

在函数式组件中，每次UI的变化，都是通过重新执行整个函数来完成的，这和传统的类组件有很大区别：函数组件中并没有一个直接的方式在多次渲染之间维持一个状态。

在重新执行整个函数组件的过程中，其中的函数和引用类型的变量会创建新的（指向新的引用），导致函数组件在re-render前后，其中函数和引用类型变量是不相等的，这又会导致其他非必要的re-render。

### 什么时候使用useCallback和useMemo?

#### useCallback

当子组件接收一个函数props时，一般会使用useCallback来缓存这个函数，减少不必要的re-render。以下例子：向子组件传递一个函数，在父组件每次re-render的时候，函数会重新创建新的，这会导致使用这个函数props的子组件也re-render，但这是不必要的，可以用useCallback来解决。

#### useMemo

useMemo常用在以下两种场景的优化中：

1. 引用类型的变量
2. 需要大量时间执行的计算函数。

useMemo 和 useCallback 接收的参数都是一样,第一个参数为回调 第二个参数为要依赖的数据

### 异同点

共同作用：

1.仅仅 依赖数据 发生变化, 才会重新计算结果，也就是起到缓存的作用。

两者区别：

1.useMemo 计算结果是 return 回来的值, 主要用于 缓存计算结果的值 ，应用场景如： 需要 计算的状态

2.useCallback 计算结果是 函数, 主要用于 缓存函数，应用场景如: 需要缓存的函数，因为函数式组件每次任何一个 state 的变化 整个组件 都会被重新刷新，一些函数是没有必要被重新刷新的，此时就应该缓存起来，提高性能，和减少资源浪费。

注意： 不要滥用会造成性能浪费，react中减少render就能提高性能，所以这个仅仅只针对缓存能减少重复渲染时使用和缓存计算结果。

### 总结

useMemo返回的是一个变量的值，useCallback返回的是一个函数。对应到function组件最后要return的html代码部分，useMemo就是作为一个值来使用的，而useCallback则是被绑定到onClick上，作为要执行的函数。这就是它俩的本质区别。

useCallback(fn, deps) 等价于 useMemo(() => fn, deps).

## useRef有什么作用？

1. 获取DOM。
2. 保存变量，存储上一次渲染的值，可以用useRef创建一个对象来存储setState前的旧值。

## useState和useReducer

都是用于**管理组件内部状态的Hooks**，它们的不同点主要在于使用场景、复杂性和可复用性等方面：

useState：

1. 用途：适合于处理简单的独立状态值。

2. 声明与使用：它接受一个初始状态值作为参数，并返回一个包含当前状态和一个更新函数（setter）的数组。每次调用这个更新函数时，会触发组件重新渲染。 const [state, setState] = useState(initialState);

3. 更新逻辑：setState函数可以直接接收新的状态值来替换旧的状态。

4. 局限性：当需要管理多个相互关联的状态或者状态更新逻辑变得复杂时，直接使用useState可能使代码难以维护。

useReducer：

1. 用途：适用于处理复杂的、有多个相关状态或具有更复杂状态更新逻辑的情况。

2. 声明与使用：它接受一个reducer函数（类似于Redux中的reducer），以及一个初始状态值（可选的初始化函数）。reducer函数负责根据action和当前状态计算出新的状态。 const [state, dispatch] = useReducer(reducer, initialState, initFunction);

3. 更新逻辑：通过dispatch函数分发actions，这些actions被reducer函数消费以生成新的状态。这种模式使得状态转换逻辑集中且易于测试。

4. 优点：支持更复杂的状态管理和逻辑复用，可以模拟Redux的行为，有利于大规模应用的状态管理，同时也便于实现异步操作和中间件。

总结来说：

useState更适合简单、独立的状态更新。

useReducer适合处理复杂的、全局性的或需要明确动作驱动的状态管理场景。

此外，useReducer还可以方便地与useContext结合使用，从而实现跨组件的状态共享和管理。

## React hooks之useCallback的使用场景及其深度解读

1、useCallBack不是每个函数都需要使用！

2、useCallBack在什么情况下使用？

### useCallBack不是每个函数都需要使用

1.为什么不用useCallBack把每个函数都包一下呢?

2.useCallBack不是缓存工具吗？

3.将每个函数都缓存不是可以更好提升性能吗？

useCallBack是一个缓存工具没错。但实际上他并不能阻止函数都重现构建。

//Com组件

const Com = () => {

//示例1包裹了useCallBack的函数

const fun1 = useCallBack(() => {

console.log('示例一函数');

},[])

//示例2没有包裹useCallBack的函数

const fun2 = () => {

console.log('示例二函数');

}

return <div></div>

}

上方这种结构的组件,Com组件中包含了fun1和fun2两个函数。

是不是认为当Com组件重新渲染的时候，只有fun2（没有使用useCallBack的函数）函数会被重新构建，而fun1（使用了useCallBack的函数）函数不会被重新构建。

**实际上，被useCallBack包裹了的函数也会被重新构建并当成useCallBack函数的实参传入。**

**useCallBack的本质工作不是在依赖不变的情况下阻止函数创建，而是在依赖不变的情况下不返回新的函数地址而返回旧的函数地址。不论是否使用useCallBack都无法阻止组件render时函数的重新创建！！**

每一个被useCallBack的函数都将被加入useCallBack内部的管理队列。而当我们大量使用useCallBack的时候，管理队列中的函数会非常之多，任何一个使用了useCallBack的组件重新渲染的时候都需要去便利useCallBack内部所有被管理的函数找到需要校验依赖是否改变的函数并进行校验。

在以上这个过程中，寻找指定函数需要性能，校验也需要性能。**所以，滥用useCallBack不但不能阻止函数重新构建还会增加“寻找指定函数和校验依赖是否改变”这两个功能，为项目增添不必要的负担。**

### useCallBack在什么情况下使用？

在往子组件传入了一个函数并且子组件被React.momo缓存了的时候使用

useCallBack的作用不是阻止函数创建，而是在依赖不变的情况下返回旧函数地址（保持地址不变）。

React.memo()，是一种缓存技术。能看到这里的笔友我想都不需要我详细解释React.memo是干什么的。

简单说，React.memo()是通过校验props中的数据是否改变的来决定组件是否需要重新渲染的一种缓存技术，具体点说React.memo()其实是通过校验Props中的数据的内存地址是否改变来决定组件是否重新渲染组件的一种技术。

假设我们往子组件（假设子组件为Child组件）传入一个函数呢？当父组件的其他State（与Child组件无关的state）改变的时候。那么，因为状态的改变，父组件需要重新渲染，那被React.memo保护的子组件（Child组件）是否会被重新构建？

import {useCallBack,memo} from 'react';

/\*\*父组件\*\*/

const Parent = () => {

const [parentState,setParentState] = useState(0); //父组件的state

//需要传入子组件的函数

const toChildFun = () => {

console.log("需要传入子组件的函数");

}

return (<div>

<Button onClick={() => setParentState(val => val+1)}>

点击我改变父组件中与Child组件无关的state

</Button>

//将父组件的函数传入子组件

<Child fun={toChildFun}></Child>

<div>)

}

/\*\*被memo保护的子组件\*\*/

const Child = memo(() => {

consolo.log("我被打印了就说明子组件重新构建了")

return <div><div>

})

当我点击父组件中的Button改变父组件中的state。子组件会不会重新渲染。乍一看，改变的是parentState这个变量，和子组件半毛钱关系没有，子组件还被React.memo保护着，好像是不会被重新渲染。但这里的问题是，你要传个其他变量进去这也就走的通了。但是传入的是函数，不行，走不通。会重新渲染。

React.memo检测的是props中数据的栈地址是否改变。而父组件重新构建的时候，会重新构建父组件中的所有函数（旧函数销毁，新函数创建，等于更新了函数地址）,新的函数地址传入到子组件中被props检测到栈地址更新。也就引发了子组件的重新渲染。

所以，在上面的代码示例里面，子组件是要被重新渲染的。

### 那么如何才能让子组件不进行重新渲染呢？useCallBack的正确使用方法来了。

使用useCallBack包一下需要传入子组件的那个函数。那样的话，父组件重新渲染，子组件中的函数就会因为被useCallBack保护而返回旧的函数地址，子组件就不会检测成地址变化，也就不会重选渲染。

import {useCallBack,memo} from 'react';

/\*\*父组件\*\*/

const Parent = () => {

const [parentState,setParentState] = useState(0); //父组件的state

//需要传入子组件的函数

//只有这里和上一个示例不一样！！

const toChildFun = useCallBack(() => {

console.log("需要传入子组件的函数");

...

},[])

return (<div>

<Button onClick={() => setParentState(val => val+1)}>

点击我改变父组件中与Child组件无关的state

</Button>

//将父组件的函数传入子组件

<Child fun={toChildFun}></Child>

<div>)

}

/\*\*被memo保护的子组件\*\*/

const Child = memo(() => {

consolo.log("我被打印了就说明子组件重新构建了")

return <div><div>

})

这样，子组件就不会被重新渲染了。

代码示例一和代码示例二中的区别只有被传入的子组件的函数（toChildFun函数）是否被useCallBack保护。

我们只需要使用useCallBack保护一下父组件中传入子组件的那个函数（toChildFun函数）保证它不会在没有必要的情况下返回一个新的内存地址就好了。

### 总结

* **useCallBack不要每个函数都包一下，否则就会变成反向优化，useCallBack本身就是需要一定性能的**
* **useCallBack并不能阻止函数重新创建,它只能通过依赖决定返回新的函数还是旧的函数,从而在依赖不变的情况下保证函数地址不变**
* **useCallBack需要配合React.memo使用**

## hooks useState setState的过程

使用 useState 返回的 setter 函数时，它不会立刻改变 state，而是发起一个状态更新过程，React 会将这次更新放入队列，并在下一个事件循环中进行处理。

如果状态依赖于之前的值，直接在 setter 中引用旧值而不是闭包外的变量，以确保拿到的是最新的状态。

调用 setState 不一定会导致组件完全重新渲染，React 的 Reconciliation 过程会根据虚拟DOM diff算法决定哪些部分需要更新。

## 使用Hooks有踩过哪些坑？

useEffect中没有正确设置依赖数组导致死循环。

useEffect中没有清除副作用导致内存泄漏。

在条件语句和循环中使用Hooks导致报错。

闭包陷阱。

## React16中render和hydrate的区别

### render

* 在提供的 container 里渲染一个 React 元素，并返回对该组件的引用（或者针对无状态组件返回 null）。
* 如果 React 元素之前已经在 container 里渲染过，这将会对其执行更新操作，并仅会在必要时改变 DOM 以映射最新的 React 元素
* 如果提供了可选的回调函数，该回调将在组件被渲染或更新之后被执行。

### hydrate

与 render() 相同，但它用于在 ReactDOMServer 渲染的容器中对 HTML 的内容进行 hydrate 操作。

hydrate基本上用于SSR（服务器端渲染）。 SSR为您提供了从服务器附带的框架或HTML标记，因此，第一次在页面加载时不为空白，搜索引擎机器人可以将其索引为SEO（SSR的一个用例）。 因此，hydrate会将JS添加到您的页面或要应用SSR的节点。 这样您的页面才能响应用户执行的事件。

渲染用于在客户端浏览器Plus上渲染组件，如果尝试将hydrate替换为render，则会收到警告，提示render已弃用，在SSR情况下无法使用。 由于它比水合物慢，因此将其除去。

### 为什么在服务端渲染的时候不采用render？

在react15中，当服务端和客户端渲染不一致时，render会做dom patch，使得最后的渲染内容和客户端一致，否则这会使得客户端代码陷入混乱之中，如下的代码就会挂掉。

import React from 'react';

export default class Admin extends React.Component {

componentDidMount() {

const container = document.querySelector('.client');

container.innerHTML = 'this is client';

}

render() {

const content = \_\_IS\_CLIENT\_\_ ? 'client' : 'server';

return (

<div className={content}>

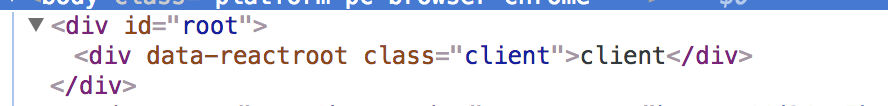
{content}

</div>

);

}

}



### 总结来说：

采用React15的render在服务端渲染的时候如果服务器端与客户端节点不一致，服务器端的节点将完全被废弃掉重新渲染并采用客户端的节点。

采用React16的render在服务端渲染的时候如果服务器端与客户端节点不一致，仅仅是尝试修改不匹配的HTML子树，而不是修改整个HTML树，同时官方给予警告：render() 对服务端渲染容器进行 hydrate 操作的方式已经被废弃，并且会在 React 17 被移除。作为替代，请使用 hydrate()。

这种变化对用户不会有影响，调用render()/hydrate()时React 16不会修改SSR生成的不匹配HTML。这一项性能优化意味着你需要额外确保修复在开发模式下的所有警告。

采用React16的hydrate时，hydrate的策略与render的策略不一样，其并不会对整个dom树做dom patch，其只会对text Content内容做patch

### 服务端和客户端之间的差异

服务端和客户端的运行环境不一样，所支持的语法也不一样。

服务端无法支持图片、css等资源文件。

服务端缺乏BOM和DOM环境，服务端下无法访问window,navigator等对象。

服务端中所有用户公用一个global环境，客户端每个用户都有自己的global环境。

### 为什么要服务器端渲染

相比于浏览器端渲染，服务器端渲染的好处是：

1.可以缩短“第一有意义渲染时间”

如果完全依赖于浏览器端渲染，那么服务器端返回的 HTML 就是一个空荡荡的框架和对 JavaScript 的应用，然后浏览器下载 JavaScript，再根据 JavaScript 中的 AJAX 调用获取服务器端数据，再渲染出 DOM 来填充网页内容，总共需要三个 HTTP 或 HTTPS 请求。

如果使用服务器端渲染，第一个 HTTP/HTTPS 请求返回的 HTML 里就包含可以渲染的内容了，这样用户第一时间就会感觉到“有东西画出来了”，这样的感知性能更好。

2.更好的搜索引擎优化（seo优化）

## react 18+新特性

* 自动批处理state更新
* 支持Suspense的新SSR架构
* Concurrent features

### Automatic batching（批处理）

批处理是指是React能够将多个状态更新合并到单个re-render中一次性更新以获得更好的渲染性能。

function App() {

const [count, setCount] = useState(0);

const [flag, setFlag] = useState(false);

function handleClick() {

setCount(c => c + 1); // 不触发re-render

setFlag(f => !f); // 不触发re-render

// 在调用结束是只触发一次re-render（这就是batching）

}

return <button onClick={handleClick}>Next</button>;

}

但是在React 18以前，对于在异步回调中调用的updates，React Batching将无法生效：

function handleClick() {

fetchSomething().then(() => {

setCount(c => c + 1); // 触发re-render

setFlag(f => !f); // 再次触发re-render

});

}

这是因为在React 18之前，React只在事件处理程序期间批量更新。默认情况下，React不会对promise、setTimeout或任意event事件中的更新进行批处理。

### 跳过批处理

通常情况下，React 18的批处理是安全的，而且是框架层面自动处理，开发者无需过多关注。但某些情况下可能依赖于在状态更改后立即从 DOM 中读取某些内容。对于这些case，可以使用ReactDOM.flushSync()选择跳过批处理：

import { flushSync } from 'react-dom';

function handleClick() {

flushSync(() => {

setCounter(c => c + 1);

});

// React has updated the DOM by now

flushSync(() => {

setFlag(f => !f);

});

// React has updated the DOM by now

}

### 类组件的breaking change

类组件有个实现上的怪异，它可以同步读取事件内部的状态更新。这意味着可以在setState之间读取到最新的state：

handleClick = () => {

setTimeout(() => {

this.setState(({ count }) => ({ count: count + 1 }));

console.log(this.state.count); // => 1

this.setState(({ flag }) => ({ flag: !flag }));

});

};

在 React 18，情况不再如此。由于即使在setTimeout中的所有更新都是批处理的，因此React不会同步渲染第一个setState的结果——渲染发生在browser nextTick，所以在setState之间读取state将获取不到上一个setState的结果：

handleClick = () => {

setTimeout(() => {

this.setState(({ count }) => ({ count: count + 1 }));

console.log(this.state.count); // 0 依然是初始值

this.setState(({ flag }) => ({ flag: !flag }));

});

};

### unstable\_batchedUpdates

这是一个不太常用的api，通常一些库会使用它来强制对事件之外的setState进行强制批处理：

import { unstable\_batchedUpdates } from 'react-dom';

unstable\_batchedUpdates(() => {

setCount(c => c + 1);

setFlag(f => !f);

});

React 18将保留unstable\_batchedUpdates，但开发者无需再手动调用它了，后续的主版本可能会移除此api。

### 新的Suspense SSR架构

除了Concurrent Mode即将正式随18正式发布之外，SSR功能也将带来全新的架构升级。

#### 传统React SSR

在React 18之前，React SSR架构通常分以下几个环节：

* server端，解析server entry并fetch所需数据（通常的做法是在Component上挂载静态方法，server直接调用）。
* 数据准备好之后，调用ReactDOM/server renderToString api将组件渲染成string形式作为response返回
* client等待2中的response text，渲染到页面上，开始加载entry assets。
* client等entry js加载完成后，执行entry js，合成（hydrate）最终的页面内容（渲染剩余内容、挂载js事件等）。

传统SSR架构里边，每一个环节都依赖上一个的执行结果，如果某一个环节特别慢会拖慢整体的渲染速度。例如首屏内容依赖server端准备好数据，但如果获取数据超时则client端依然会有很长的白屏时间，依然解决不了体验（性能）问题。

### Streaming HTML and Selective Hydration

中文翻译：流式传输HTML + 选择性水合

Suspense解锁了React 18中有两个主要的SSR特性：

* 在server端流式传输HTML：使用全新的pipeToNodeWritable api来搭建从server端到client的流式渲染通道，可以持续不断地向client输出已经ready的html区块，而不需要像传统SSR那样需要整体ready之后通过renderToString一次性输出。
* client端进行选择性水合（hydration）：将组件（或者widget）使用组件进行包裹，这样它们就不会阻塞渲染，在组件内容数据ready之前，框架将渲染通过fallback参数指定的内容。

在新的SSR架构下，能够实现页面上的组件单独render，React无需等待所有组件“ready”后再展现给用户，实现了页面的分片渲染。

以上Features使得传统SSR模式的一些瓶颈得到解决：

* server端无需等待所有数据（和HTML）都ready再响应客户端，相反地，当应用骨架准备好时你就可以发送给客户端显示，剩余的内容将在它们ready之后流式传输给客户端。
* client端来说，不再需要等待所有 JavaScript 加载完毕才能开始渲染。可以结合code spliting与SSR一起使用，在server端的HTML（片段）将被保留（在server），当相关代码加载时，React将对其进行合成。
* 不再需要等待所有组件都加载完成才运行用户交互。相反，可以依靠Selective Hydration 来确定用户与之交互的组件的优先级。

Streaming HTML（对应pipeToNodeWritable）这个特性其实并不是React首创，BigPipe技术，它最早是facebook先提出来的，原理就是通过server端生成html（或者js片段），通过预先内置的runtime sdk解析下载的片段并执行。不过BigPipe技术没有流行起来是因为它依赖了非前端的一些技术栈，没有和前端框架、工程体系结合起来，而React 18的SSR做到了和开发框架的完美融合，在框架层面实现了片段的生成和装载（hydrate），这使得一些需要动态能力的场景多了一种全新的技术选型（比如投放）。

Selective Hydration（对应hydrateRoot以及）表明hydration过程也是逐步进行的，不会一下子执行完所有js导致页面卡顿。正常情况下，框架会根据node tree的顺序执行hydration，但React会监听用户行为事件（如click），提升被用户点击的区块在hydration的优先级。

总结下，Suspense SSR架构的核心原理：渐进式SSR（Progressive SSR） + 部分非阻塞式水合（Partial non-blocking hydration）。它把一张页面分割成了无数个「插槽」区块，然后server端可以持续不断地将已经ready的区块HTML返回给client，client端根据优先级决定合成哪些区块，从而带来相对完善的用户体验效果。

### New root apis

在React 18以前的版本中，root元素被绑定（attached）到了DOM根节点上，它对用户是不可见的：

const container = document.getElementById('app');

// 初始化 render.

ReactDOM.render(<App tab="home" />, container);

// 为了更新app，你不得不保留container的引用

ReactDOM.render(<App tab="profile" />, container);

React 18带来了更加语义化的root api：

// 创建root节点

const root = ReactDOM.createRoot(container);

// 初始化：将组建渲染到root

root.render(<App tab="home" />);

// 更新组件时，无需再关注container的引用了，因为它已经提前绑定在root上

root.render(<App tab="profile" />);

SSR（hydrate）root api：

// 旧的api

ReactDOM.hydrate(<App tab="home" />, container);

// 新的root api

ReactDOM.hydrateRoot(container, <App tab="home" />);

注意，和createRoot不同的是，hydrateRoot的第二个参数就需要传入初始化jsx（而无需调用root.render()），这是因为初始client端渲染是特殊的，需要与server端的节点树匹配。

首次hydration之后如果想更新root，可以调用render：

// You can later update it.

root.render(<App tab="profile" />);

React 18新的root api移除了render的callback，原因是Suspense SSR采用了渐进式SSR以及部分水合，回调的时机可能与用户预期不符，为了避免混淆，React推荐的做法是使用requestIdleCallback、setTimeout或者在root上ref回调：

// 旧的root api，直接通过render参数传入callback

ReactDOM.render(container, <App />, function() {

console.log('rendered').

});

// 新的root api，通过ref传入render callback

const root = ReactDOM.createRoot(container);

root.render(<App callback={() => console.log("renderered")} />);

function App({ callback }) {

// 当div第一次被创建时callback将会被调用

return <div ref={callback}>...</div>;

}

### Concurrent new feature: startTransition

React 18以前，Concurrent Mode一直处于实验性阶段，React将正式发布Concurrent功能。Concurrent Mode简单说就是一种非阻塞UI、可中断式的渲染架构。它可以灵活调度渲染任务以避免js长时间执行导致UI进程无法响应。

React将状态更新分为两类：

* 紧急更新（Urgent updates）：反映直接的交互，如输入、点击、按键按下等等。
* 过渡更新（Transition updates）：将UI从一个视图过渡到另一个视图。

输入、点击、按键按下等需要立即响应以符合人类的物理认知，符合人的直觉。但是过渡更新却不同，用户不会期望看到中间的转换过程（只需要结果），因此可能不需要立即更新视图。

在React 18以前的版本以及React 18默认情况下（为了向前兼容），所有的更新都会认为是紧急更新。而startTransition提供api给用户来手动将某些更新标记为非紧急更新，从而避免浪费时间去渲染不必要的内容。

// 在界面上显示用户输入

setInputValue(input);

// 在界面上呈现查询结果

setSearchQuery(input);

setInputValue会立即更新用户的输入到界面上，属于需要紧急更新的操作。setSearchQuery是根据用户输入，查询相应的内容，用户可以输入很多次，如果一直查询会可能会导致过多的js计算消耗在查询上面（甚至阻塞输入操作），而一般来说用户的期望值是「等待输入完成之后，查询并显示最终的结果」，因此这里的setSearchQuery可以看成是非紧急更新。

通过startTransition包裹setSearchQuery将其标记为非紧急更新：

setInputValue(input);

// 标记为非紧急更新

startTransition(() => {

React.setSearchQuery(input);

});

#### 和setTimeout的区别

就上面setSearchQuery的例子，使用setTimeout（或者debounce or throttle）也能达到相似的目的，那这个startTransition和setTimeout有啥区别？

* 一个重要区别是setTimeout是「延迟」执行，startTransition是立即执行的，传递给startTransition的函数是同步运行，但是其内部的所有更新都会标记为非紧急，React将在稍后处理更新时决定如何render这些updates，这意味着将会比setTimeout中的更新更早地被render。
* 另一个重要区别是用setTimeout包裹的如果是内大面积的更新操作会导致页面阻塞不可交互，直到超时。这时候用户的输入、键盘按下等紧急更新操作将被阻止。而startTransition则不同，由于它所标记的更新都是可中断的，所以不会阻塞UI交互。即使用户输入发生变化，React也不必继续渲染用户不再感兴趣的内容。
* 最后，因为setTimeout是异步执行，哪怕只是展示一个小小的loading也要编写异步代码。而通过transitions，React可以使用hook来追踪transition的执行状态，根据transition的当前状态来更新loading。

### 追踪transition的执行状态

useTransition hook：

function App() {

const [isPending, startTransition] = React.useTransition();

startTransition(() => {

// 标记非紧急更新

});

return isPending ? <Spinner /> : <Content />;

}

总结：可以使用startTransition来wrapper想要移动到后台执行的任何更新。通常，这些类型的更新分为两类：

* 慢渲染（Slow rendering）：那些需要耗费大量的时间的UI转换任务。
* 弱网络（Slow network）：网络请求更新也可以标记为非紧急更新。这与Suspense能力密切集成。

这里有个利用startTransition来优化用户体验的例子，几个视频看下来可以说很直观体现出startTransition的效果了。