# 函数组件Hook

## 参考网址：[React 参考总览 – React 中文文档](https://zh-hans.react.dev/reference/react)

在react16.8版本之前，函数组件是没有状态和没有生命周期的，所以它又被称为“无状态组件”。16.8版本新增了Hook特性。Hook可以让无状态组件拥有状态组件才拥有的能力，解决了函数组件没有状态和生命周期的问题。Hooks 是 React 函数组件中的一类特殊函数，通常以 use 开头，比如 useRef，useState，useReducer 等。通常在我们写 React 组件的时候，如果这个组件比较复杂，拥有自己的生命周期或者 state，就将其写成 class 组件；如果这个组件仅仅用来展示，就将其写成 function 组件。

React 中内置的 Hook API

* [基础 Hook](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#basic-hooks)
  + [useState](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#usestate)
  + [useEffect](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#useeffect)
  + [useContext](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#usecontext)
* [额外的 Hook](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#additional-hooks)
  + [useReducer](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#usereducer)
  + [useCallback](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#usecallback)
  + [useMemo](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#usememo)
  + [useRef](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#useref)
  + [useImperativeHandle](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#useimperativehandle)
  + [useLayoutEffect](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#uselayouteffect)
  + [useDebugValue](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#usedebugvalue)
  + [useDeferredValue](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#usedeferredvalue)
  + [useTransition](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#usetransition)
  + [useId](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#useid)
* [Library Hooks](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#library-hooks)
  + [useSyncExternalStore](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#usesyncexternalstore)
  + [useInsertionEffect](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#useinsertioneffect)

## 常用的Hook

|  |  |
| --- | --- |
| Hook | 说明 |
| useState | 定义组件内部数据 |
| useEffect | 模拟组件内部的生命周期函数 |
| useRef | 用来获取节点或者组件 |
| [useMemo](https://zh-hans.legacy.reactjs.org/docs/hooks-reference.html#usememo) | 用来缓存数据，模拟就是计算属性 |
| useContext | 用来实现组件间传值 |

# useState

## 还是上面的reactdemo3项目，新建一个文件夹-07react-hook，然后新建一个组件文件FunctionState.jsx

## 然后使用快捷键rsc回车新建一个函数组件骨架

|  |
| --- |
| **import *React* from 'react'**;  **const** *FunctionState* = () => {  **return** (  <**div**>    </**div**>  ); };  **export default** *FunctionState*; |

## 在App.jsx中引用该组件

|  |
| --- |
| **import *React* from 'react'**;  **const** *FunctionState* = () => {  **return** (  <**div**>  <**h3**>FunctionState</**h3**>  </**div**>  ); };  **export default** *FunctionState*; |

### useState的用法：let [属性名，set属性名] = useState(初始值)，注意：他返回的是有两个元素的数组，第一个元素是属性名，第二个元素是这个元素的设置方法

### 基本使用，我们添加两个按钮，一个+，一个-分别给他们添加数据处理

|  |
| --- |
| **import *React***, {*useState*} **from 'react'**;  **const** *FunctionState* = () => {  **let** [count,setCount] = *useState*(0);  **return** (  <**div**>   <**h3**>当前计数：{count}</**h3**>  <**button onClick=**{()=>setCount(++count)}>+</**button**><**button onClick=**{()=>setCount(--count)}>-</**button**>  </**div**>  ); };  **export default** *FunctionState*; |

### 效果，点击+，count就会增大，点击-，count就会减少

|  |
| --- |
|  |
|  |

### 在事件处理函数里面传递参数，新建2给按钮，定义一个处理方法setTimesCount(times)然后在事件处理里面调用这个方法并且传递参数代码如下

|  |
| --- |
| **import *React***, {*useState*} **from 'react'**;  **const** *FunctionState* = () => {   **let** [count,setCount] = *useState*(0);  **let** setTimesCount=(times)=>{ *//自定义方法可以在事件处理箭头函数里面调用* setCount(count\*times)  }  **return** (  <**div**>   <**h3**>当前计数：{count}</**h3**>  <**button onClick=**{()=>setCount(++count)}>+</**button**><**button onClick=**{()=>setCount(--count)}>-</**button**>  <**button onClick=**{()=>setTimesCount(3)}>3times</**button**><**button onClick=**{()=>setTimesCount(0.25)}>0.25times</**button**>  </**div**>  ); };  **export default** *FunctionState*; |

### 效果

|  |
| --- |
|  |
|  |

# useEffect：传递不同的参数会有不同的效果

## useEffect是用来在函数组件里面模拟生命周期函数的

## 我们在src目录下面新建一个08react-Hook-useEffect目录，然后在里面新建一个FunctionLifeCycle.jsx函数组件文件，用快捷键rsc回车创建骨架

|  |
| --- |
|  |

## 在App.jsx中添加对这个组件的引用

|  |
| --- |
| **import *React* from 'react'**;  **const** *FunctionLifeCycle* = () => {  **return** (  <**div**>  <**h3**>FunctionLifeCycle</**h3**>  </**div**>  ); };  **export default** *FunctionLifeCycle*; |

## useEffect有4种用法

### 1）用法1，我们在FunctionLifeCycle组件里面添加如下代码

|  |
| --- |
| **import *React***, {*useEffect*, *useState*} **from 'react'**;  **const** *FunctionLifeCycle* = () => {  **let** [count,setCount] = *useState*(1)  **let** [name,setName] = *useState*(**'Mary'**)  *useEffect*(()=>{  ***console***.log(**'useEffect'**)  })   **return** (  <**div**>  <**h3**>{count}</**h3**>  <**h3**>{name}</**h3**>  <**button onClick=**{()=>setCount(++count)}>count++</**button**>  <**button onClick=**{()=>setName(**'Jenifer'**)}>set name</**button**>  </**div**>  ); };  **export default** *FunctionLifeCycle*; |

### 经过测试，我们发现当页面第一次渲染会调用useEffect，然后当count的值改变也会触发useEffect当name改变的时候也会触发useEffect没有改变就不会触发

|  |
| --- |
|  |

### 第一种用法就是只给useEffect传递一个回调函数作为参数，没有给第二个参数(参考上面的代码)，他会在组件挂载完成时和组件数据更新完成时执行，他模拟了类组件的componentDidMount和componentDidUpdate生命周期函数

### 2)用法2，是常用的

### 第二种用法，是useEffect的第二个参数传递一个空数组，参考如下代码

|  |
| --- |
| **import *React***, {*useEffect*, *useState*} **from 'react'**;  **const** *FunctionLifeCycle* = () => {  **let** [count,setCount] = *useState*(1)  **let** [name,setName] = *useState*(**'Mary'**)  *// useEffect(()=>{ //第一种用法  // console.log('useEffect')  // })  useEffect*(()=>{  ***console***.log(**'useEffect'**)  },[])   **return** (  <**div**>  <**h3**>{count}</**h3**>  <**h3**>{name}</**h3**>  <**button onClick=**{()=>setCount(++count)}>count++</**button**>  <**button onClick=**{()=>setName(**'Jenifer'**)}>set name</**button**>  </**div**>  ); };  **export default** *FunctionLifeCycle*; |

### 此时他只有在组件挂载完成时才会调用，组件更新时不会被调用。此时它模拟了类组件中的componentDidMount生命周期函数，这种调用方式通常用来发送网络请求

### 3）用法3，也比较常用

### 第三种用法是第二个参数是非空数组，参考下面代码，我们在数组中只放count

|  |
| --- |
| **import *React***, {*useEffect*, *useState*} **from 'react'**;  **const** *FunctionLifeCycle* = () => {  **let** [count,setCount] = *useState*(1)  **let** [name,setName] = *useState*(**'Mary'**)  *// useEffect(()=>{ //第一种用法  // console.log('useEffect')  // })  // useEffect(()=>{ //第二种用法  // console.log('useEffect')  // },[])  useEffect*(()=>{  ***console***.log(**'useEffect'**)  },[count])   **return** (  <**div**>  <**h3**>{count}</**h3**>  <**h3**>{name}</**h3**>  <**button onClick=**{()=>setCount(++count)}>count++</**button**>  <**button onClick=**{()=>setName(**'Jenifer'**)}>set name</**button**>  </**div**>  ); };  **export default** *FunctionLifeCycle*; |

#### 效果：因为我们只在数组里面放了count，所以在组件挂载完成和count改变了，useEffect会执行，name改变了，useEffect不会执行，此时他有点像vue里面的watch功能，如果把name和count都放进数组，它就和第一种情况一样了

### 4）用法4

### 第四种用法就是回调函数返回一个函数，忽略第二个参数，返回的函数会在组件卸载前执行，代码参考下面

|  |
| --- |
| **import *React***, {*useEffect*, *useState*} **from 'react'**;  **const** *FunctionLifeCycle* = () => {  **let** [count,setCount] = *useState*(1)  **let** [name,setName] = *useState*(**'Mary'**)  *// useEffect(()=>{ //第一种用法  // console.log('useEffect')  // })  // useEffect(()=>{ //第二种用法  // console.log('useEffect')  // },[])  // useEffect(()=>{ //第三种用法  // console.log('useEffect')  // },[count])  useEffect*(()=>{ *//第四种用法* ***console***.log(**'useEffect'**)  **return** ()=>{  ***console***.log(**'useEffect clean up'**)  }  })   **return** (  <**div**>  <**h3**>{count}</**h3**>  <**h3**>{name}</**h3**>  <**button onClick=**{()=>setCount(++count)}>count++</**button**>  <**button onClick=**{()=>setName(**'Jenifer'**)}>set name</**button**>  </**div**>  ); };  **export default** *FunctionLifeCycle*; |

### 它模拟了类组件的componentWillUnmount生命周期函数

### 效果:我们发现可以把这个返回的函数作为一个清理函数这样子每当开始下一次渲染之前，就会清理上一次的渲染结果。

|  |
| --- |
|  |
|  |

### 注意：这四种用法中最常用的是第二种，然后就是第三种。

# useMemo

## 这个Hook是用来在函数组件里面模拟vue的计算属性的，我们还是通过一个例子来学习

## 还是reactdemo3项目，在src下面新建一个目录：09react-Hook-useMemo,然后在这个文件夹下面新建一个FunctionComputed.jsx组件文件，用快捷键rsc回车创建组件骨架

|  |
| --- |
| **import *React* from 'react'**;  **const** *FunctionComputed* = () => {  **return** (  <**div**>  <**h3**>FunctionComputed</**h3**>  </**div**>  ); };  **export default** *FunctionComputed*; |

## 在App.jsx中添加引用

|  |
| --- |
| **import** { *useState* } **from 'react'** *// import reactLogo from './assets/react.svg' // import viteLogo from '/vite.svg' // import './App.css'* **import** *FuncComponent* **from "./01-compoment-basic/FuncComponent"**; **import** ClassComponent **from "./01-compoment-basic/ClassComponent"**; **import** ClassEvent **from "./02-class-component-events/ClassEvent"**; **import** ClassState **from "./03class-component-state/ClassState"**; **import** ClassComputed **from "./04class-computed-property/ClassComputed" import** ClassLifeCycle **from "./05class-comp-life-cycle/ClassLifeCycle"**; *// import SideMenu from "./side-menu/SideMenu";* **import** SideMenu2 **from "./side-menu-v2/SideMenu2"**; **import** ComponentStyle **from "./06component-style/ComponentStyle"**; **import** *FunctionState* **from "./07readct-hook-useState/FunctionState"**; **import** *FunctionLifeCycle* **from "./08react-Hook-useEffect/FunctionLifeCycle"**; **import** *FunctionComputed* **from "./09react-Hook-useMemo/FunctionComputed"**;  **function** *App*() {   **return** (  <>  {*/\*<h1>React demo3</h1>\*/*}  {*/\*<FuncComponent/>\*/*}  {*/\*<ClassComponent/>\*/*}  {*/\*<ClassEvent/>\*/*}  {*/\* <ClassState></ClassState>\*/*}  {*/\* <ClassComputed></ClassComputed>\*/*}  {*/\* <ClassLifeCycle></ClassLifeCycle>\*/*}  {*/\* <SideMenu></SideMenu>\*/*}  {*/\* <SideMenu2/>\*/*}  {*/\* <ComponentStyle></ComponentStyle>\*/*}  {*/\* <FunctionState></FunctionState>\*/*}  {*/\* <FunctionLifeCycle></FunctionLifeCycle>\*/*}  <**FunctionComputed**></**FunctionComputed**>  </>  ) }  **export default** *App* |

## useMemo的使用方法：useMemo(有返回值的回调函数，[依赖项]),这个回调函数没有参数而且可以返回任意值，如果初次调用后，依赖项没有变化，就直接返回回相同值，否则，将会再次调用 calculateValue 并返回最新结果，然后缓存该结果以便下次重复使用。

## 我们在FunctionComputed.jsx这添加一些代码

|  |
| --- |
| **import *React***, {*useMemo*, *useState*} **from 'react'**;  **const** *FunctionComputed* = () => {  **let** [count,setCount] = *useState*(1)  **var** result = *useMemo*(()=>{  ***console***.log(**'useMemo'**)  **return** count+10/2+count  },[count]);    **return** (  <**div**>  <**h3**>{result}</**h3**>  <**h3**>{result}</**h3**>  <**h3**>{result}</**h3**>   </**div**>  ); };  **export default** *FunctionComputed*; |

## 效果如下

|  |
| --- |
|  |

### 我们可以把上面的代码改一下，更好的来体验计算属性的作用

#### FunctionComputed.jsx

|  |
| --- |
| **import *React***, {*useMemo*, *useRef*, *useState*} **from 'react'**;  **const** *FunctionComputed* = () => {  **let** [nums,setNums] = *useState*([10,20,30,40])  **let** inputRef = *useRef*(**null**)  //使用useMemo来模拟vue的计算属性，当依赖项改变了，它会重新计算，我们用下面的点击按钮来给nums添加元素，注意类型转换  **let** max = *useMemo*(()=>{  **let** biggest = nums[0]  nums.forEach(item=>{  **if**(item>biggest){  biggest = item  }  })  **return** biggest  },[nums])  **return** (  <**div**>  <**ul**>  {nums.map(item=><**li key=**{item}>{item}</**li**>)}  </**ul**>  <**p**>max value:{max}</**p**>  new value: <**input type="text" ref=**{inputRef} /> <**button onClick=**{()=>setNums([  ...nums,***Number***(inputRef.**current**.**value**)  ])}>add</**button**>  </**div**>   ); };  **export default** *FunctionComputed*; |

#### 效果

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

### 可以看到需要用一个变量将useMemo的返回值保存起来。供后面使用

## useMemo和useCallback的区别是什么？

useCallback是将某个函数“放入到react底层原型链上，并返回该函数的索引”，而useMemo是将某个函数返回值“放入到react底层原型链上，并返回该返回值的索引”。一个是针对函数，一个是针对函数返回值。

网上有些人的文章里，会提到：useCallback(fn, deps) 相当于 useMemo(() => fn, deps)。

这句话似乎是没有问题，但是他隐藏或者说忽略了几个重要关键点：  
1、不是所有fn(函数)都适用的，必须是该函数有返回值，即函数有 return xx 才可以。  
2、虽然都是fn，但是函数体内代码内容却相差很大，useCallback中的fn主要用来处理各种操作事务的代码，例如修改某变量值或加载数据等。而useMemo中的fn主要用来处理各种计算事务的代码。  
3、useCallback和useMemo都是为了提升组件性能，但是他们两个的适用场景却不相同，不是谁是谁的替代品或谁是谁的简化版。

再次强调一遍，useCallback中的函数是侧重“操作事务”，useMemo中的函数是侧重“计算结果”，永远不要在useMemo的函数中添加修改数据之类的代码。

## useMemo是来解决什么问题的？

答：useMemo的目的是“减少组件重新渲染时不必要的函数计算”。  
useMemo可以将某些函数的计算结果(返回值)挂载到react底层原型链上，并返回该函数返回值的索引。当组件重新渲染时，如果useMemo依赖的数据变量未发生变化，那么直接使用原型链上保存的该函数计算结果，跳过本次无意义的重新计算，达到提高组件性能的目的。

补充说明：  
1、useMemo并不需要子组件必须使用React.memo。  
2、“不必要的函数计算”中的函数计算必须是有一定复杂度的，例如需要1000个for循环才能计算出的某个值。如果计算量本身很简单，例如1+2，那完全没有必要使用useMemo，就直接每次重新计算一遍也无所谓。

## 实例2，我们把FunctionComputed.jsx复制一份，改名FunctionComputed2.jsx，把代码修改如下

|  |
| --- |
| **import *React***, {*useMemo*, *useState*} **from 'react'**;  **const** *FunctionComputed2* = () => {  **const** [num,setNum] = *useState*(2020);  **const** [random,setRandom] = *useState*(0);   *//通过useMemo将函数内的计算结果(返回值)保存到react底层原型链上  //totalPrimes为react底层原型链上该函数计算结果的引用* **const** totalPrimes = *useMemo*(() => {  ***console***.log(**'begin....'**); *//这里添加一个console.log，方便验证在重新渲染时是否重新执行了一遍计算* **let** total = 0; *//声明质数总和对应的变量   //以下为计算num范围内所有质数个数总和的计算代码，不需要认真阅读，只需要知道这是一段“比较复杂的计算代码”即可* **for**(**let** i = 1; i<=num; i++){  **let** boo = **true**;  **for**(**let** j = 2; j<i; j++){  **if**(i % j === 0){  boo = **false**;  **break**;  }  }  **if**(boo && i!==1){  total ++;  }  }  *//复杂的计算代码到此结束* **return** total;*//将质数总和作为返回值return出去* }, [num]);   **const** clickHandler01 = () => {  setNum(num+1);  }   **const** clickHandler02 = () => {  setRandom(***Math***.floor(***Math***.random()\*100)); *//修改random的值导致整个组件重新渲染* }   **return** (  <**div**>  {num} - {totalPrimes} - {random}  <**button onClick=**{clickHandler01}>num + 1</**button**>  <**button onClick=**{clickHandler02}>random</**button**>  </**div**>  ) };  **export default** *FunctionComputed2*; |

## 在App.jsx中添加对这个组件的引用

|  |
| --- |
| **import** { *useState* } **from 'react'** *// import reactLogo from './assets/react.svg' // import viteLogo from '/vite.svg' // import './App.css'* **import** *FuncComponent* **from "./01-compoment-basic/FuncComponent"**; **import** ClassComponent **from "./01-compoment-basic/ClassComponent"**; **import** ClassEvent **from "./02-class-component-events/ClassEvent"**; **import** ClassState **from "./03class-component-state/ClassState"**; **import** ClassComputed **from "./04class-computed-property/ClassComputed" import** ClassLifeCycle **from "./05class-comp-life-cycle/ClassLifeCycle"**; *// import SideMenu from "./side-menu/SideMenu";* **import** SideMenu2 **from "./side-menu-v2/SideMenu2"**; **import** ComponentStyle **from "./06component-style/ComponentStyle"**; **import** *FunctionState* **from "./07readct-hook-useState/FunctionState"**; **import** *FunctionLifeCycle* **from "./08react-Hook-useEffect/FunctionLifeCycle"**; **import** *FunctionComputed* **from "./09react-Hook-useMemo/FunctionComputed"**; **import** *FunctionComputed2* **from "./09react-Hook-useMemo/FunctionComputed2"**;  **function** *App*() {   **return** (  <>  {*/\*<h1>React demo3</h1>\*/*}  {*/\*<FuncComponent/>\*/*}  {*/\*<ClassComponent/>\*/*}  {*/\*<ClassEvent/>\*/*}  {*/\* <ClassState></ClassState>\*/*}  {*/\* <ClassComputed></ClassComputed>\*/*}  {*/\* <ClassLifeCycle></ClassLifeCycle>\*/*}  {*/\* <SideMenu></SideMenu>\*/*}  {*/\* <SideMenu2/>\*/*}  {*/\* <ComponentStyle></ComponentStyle>\*/*}  {*/\* <FunctionState></FunctionState>\*/*}  {*/\* <FunctionLifeCycle></FunctionLifeCycle>\*/*}  {*/\* <FunctionComputed></FunctionComputed>\*/*}  <**FunctionComputed2**></**FunctionComputed2**>  </>  ) }  **export default** *App* |

### 效果如下：

|  |
| --- |
|  |

### 这个实例的参考网址：<https://github.com/puxiao/react-hook-tutorial/blob/master/11%20useMemo%E5%9F%BA%E7%A1%80%E7%94%A8%E6%B3%95.md>

# useRef

## 这个Hook是用来获取节点的

## 我们在src目录下面新建一个文件夹10react-Hook-useRef,然后新建一个FunctionRef.jsx组件文件，用快捷键rsc（vscode是rafc）创建骨架，

|  |
| --- |
| **import *React* from 'react'**;  **const** *FunctionRef* = () => {  **return** (  <**div**>  <**h3**>FunctionRef</**h3**>  </**div**>  ); };  **export default** *FunctionRef*; |

## 在App.jsx中添加这个组件的引用

|  |
| --- |
| **import** { *useState* } **from 'react'** *// import reactLogo from './assets/react.svg' // import viteLogo from '/vite.svg' // import './App.css'* **import** *FuncComponent* **from "./01-compoment-basic/FuncComponent"**; **import** ClassComponent **from "./01-compoment-basic/ClassComponent"**; **import** ClassEvent **from "./02-class-component-events/ClassEvent"**; **import** ClassState **from "./03class-component-state/ClassState"**; **import** ClassComputed **from "./04class-computed-property/ClassComputed" import** ClassLifeCycle **from "./05class-comp-life-cycle/ClassLifeCycle"**; *// import SideMenu from "./side-menu/SideMenu";* **import** SideMenu2 **from "./side-menu-v2/SideMenu2"**; **import** ComponentStyle **from "./06component-style/ComponentStyle"**; **import** *FunctionState* **from "./07readct-hook-useState/FunctionState"**; **import** *FunctionLifeCycle* **from "./08react-Hook-useEffect/FunctionLifeCycle"**; **import** *FunctionComputed* **from "./09react-Hook-useMemo/FunctionComputed"**; **import** *FunctionComputed2* **from "./09react-Hook-useMemo/FunctionComputed2"**; **import** *FunctionRef* **from "./10react-Hook-useRef/FunctionRef"**;  **function** *App*() {   **return** (  <>  {*/\*<h1>React demo3</h1>\*/*}  {*/\*<FuncComponent/>\*/*}  {*/\*<ClassComponent/>\*/*}  {*/\*<ClassEvent/>\*/*}  {*/\* <ClassState></ClassState>\*/*}  {*/\* <ClassComputed></ClassComputed>\*/*}  {*/\* <ClassLifeCycle></ClassLifeCycle>\*/*}  {*/\* <SideMenu></SideMenu>\*/*}  {*/\* <SideMenu2/>\*/*}  {*/\* <ComponentStyle></ComponentStyle>\*/*}  {*/\* <FunctionState></FunctionState>\*/*}  {*/\* <FunctionLifeCycle></FunctionLifeCycle>\*/*}  {*/\* <FunctionComputed></FunctionComputed>\*/*}  {*/\* <FunctionComputed2></FunctionComputed2>\*/*}  <**FunctionRef**/>   </>  ) }  **export default** *App* |

### 测试正常

|  |
| --- |
|  |

### 我们添加一些代码，我们需要实现的功能是当输入框失去焦点的时候，将他的值赋给count

|  |
| --- |
| **import *React***, {*useRef*, *useState*} **from 'react'**;  **const** *FunctionRef* = () => {  **let** [count,setCount] = *useState*(0)  **let** inputRef = *useRef*(**null**)  **return** (  <**div**>  <**h3**>count:{count}</**h3**>  {*/\*<input type="text" onChange={(e)=>{\*/*}  {*/\* setCount(e.target.value)\*/*}  {*/\*}}/>\*/*}  <**input type="text" ref=**{inputRef} **onBlur=**{(e)=>{  setCount(inputRef.**current**.**value**)  *// console.log(inputRef.current.value)* }}/>  </**div**>  ); };  **export default** *FunctionRef*; |

### 效果如下

|  |
| --- |
|  |

### 我们把FunctionRef.jsx的代码改一下，实现另外一个案例

|  |
| --- |
| **import *React***, {*useRef*} **from 'react'**;  **const** *FunctionRef* = () => {  **let** divRef = *useRef*(**null**)  **let** inputRef = *useRef*(**null**)  **return** (  <**div**>  <**div ref=**{divRef}>hello</**div**>  <**button onClick=**{()=>{  ***console***.log(divRef.**current**.**innerHTML**)  inputRef.**current**.**value** = divRef.**current**.**innerHTML** }}>get text</**button**>  <**input type="text" ref=**{inputRef}/>  </**div**>  ); };  **export default** *FunctionRef*; |

#### 效果：

|  |
| --- |
|  |
|  |

#### 注意：获取到的节点需要根据类型来获取值如：div需要使用divRef.current.innerHTML来获取文本，input标签需要使用inputRef.current.value

### 使用useRef的步骤，

#### 1.用一个变量来接受uesRef函数的返回值，

#### 2.在一个标签在利用ref属性把她和返回值绑定，然后就可以获取返回值里面的属性值

## 我们还可以在onBluer事件里面把失去焦点的文本框的内容清空

|  |
| --- |
| **import *React***, {*useRef*, *useState*} **from 'react'**;  **const** *FunctionRef* = () => {  **let** [count,setCount] = *useState*(0)  **let** inputRef = *useRef*(**null**)  **return** (  <**div**>  <**h3**>count:{count}</**h3**>  {*/\*<input type="text" onChange={(e)=>{\*/*}  {*/\* setCount(e.target.value)\*/*}  {*/\*}}/>\*/*}  <**input type="text" ref=**{inputRef} **onBlur=**{(e)=>{  setCount(***Number***(inputRef.**current**.**value**));//这样好一点  inputRef.**current**.**value** = **''**;  }}/>  </**div**>  ); };  **export default** *FunctionRef*; |

### 效果

#### 失去焦点前

|  |
| --- |
|  |

#### 失去焦点后

|  |
| --- |
|  |

# useCallBack，可以把函数缓存起来，避免不必要的渲染从而优化组件

## 我们创建一个实例

### 在src文件夹下面新建一个文件夹：14func-component-optimize，然后在它下面新建一个FatherMemo.jsx和一个ChildMemo.jsx组件文件，用快捷键rsc创建骨架在父组件里面引用子组件

#### FatherMemo.jsx

|  |
| --- |
| **import *React* from 'react'**; **import** *ChildMemo* **from "./ChildMemo"**;  **const** *FatherMemo* = () => {  **return** (  <**div**>  <**ChildMemo**/>  </**div**>  ); };  **export default** *FatherMemo*; |

#### ChildMemo.jsx

|  |
| --- |
| **import *React* from 'react'**;  **const** *ChildMemo* = () => {  **return** (  <**div**>    </**div**>  ); };  **export default** *ChildMemo*; |

### 在App.jsx中添加对父组件的引用

|  |
| --- |
| **import** { *useState* } **from 'react'** *// import reactLogo from './assets/react.svg' // import viteLogo from '/vite.svg' // import './App.css'* **import** *FuncComponent* **from "./01-compoment-basic/FuncComponent"**; **import** ClassComponent **from "./01-compoment-basic/ClassComponent"**; **import** ClassEvent **from "./02-class-component-events/ClassEvent"**; **import** ClassState **from "./03class-component-state/ClassState"**; **import** ClassComputed **from "./04class-computed-property/ClassComputed" import** ClassLifeCycle **from "./05class-comp-life-cycle/ClassLifeCycle"**; *// import SideMenu from "./side-menu/SideMenu";* **import** SideMenu2 **from "./side-menu-v2/SideMenu2"**; **import** ComponentStyle **from "./06component-style/ComponentStyle"**; **import** *FunctionState* **from "./07readct-hook-useState/FunctionState"**; **import** *FunctionLifeCycle* **from "./08react-Hook-useEffect/FunctionLifeCycle"**; **import** *FunctionComputed* **from "./09react-Hook-useMemo/FunctionComputed"**; **import** *FunctionComputed2* **from "./09react-Hook-useMemo/FunctionComputed2"**; **import** *FunctionRef* **from "./10react-Hook-useRef/FunctionRef"**; **import** *TodoList* **from "./exercise-todoList/TodoList"**; **import** *FunctionProps* **from "./11react-props/FunctionProps"**; **import** *FatherProp* **from "./11react-props/FatherProp"**; **import** *FatherEvent* **from "./12comp-communication/FatherEvent"**; **import** *GrandFather* **from "./13multilayer-comp-communication/GrandFather"**; **import** *FatherMemo* **from "./14func-component-optimize/FatherMemo"**;  **function** *App*() {  **let** props={  **name** :**'jack'** ,  **age**:35,  **gender**:**'male'** ,  **money**:9000000  }  **return** (  <>  {*/\*<h1>React demo3</h1>\*/*}  {*/\*<FuncComponent/>\*/*}  {*/\*<ClassComponent/>\*/*}  {*/\*<ClassEvent/>\*/*}  {*/\* <ClassState></ClassState>\*/*}  {*/\* <ClassComputed></ClassComputed>\*/*}  {*/\* <ClassLifeCycle></ClassLifeCycle>\*/*}  {*/\* <SideMenu></SideMenu>\*/*}  {*/\* <SideMenu2/>\*/*}  {*/\* <ComponentStyle></ComponentStyle>\*/*}  {*/\* <FunctionState></FunctionState>\*/*}  {*/\* <FunctionLifeCycle></FunctionLifeCycle>\*/*}  {*/\* <FunctionComputed></FunctionComputed>\*/*}  {*/\* <FunctionComputed2></FunctionComputed2>\*/*}  {*/\* <FunctionRef/>\*/*}  {*/\*<TodoList/>\*/*}  {*/\* <FunctionProps name = 'jack' age={30} gender='male' money={100000} />\*/*}  {*/\* <FunctionProps {...props} />\*/*}  {*/\* <FatherProp/>\*/*}  {*/\* <FatherEvent/>\*/*}  {*/\* <GrandFather/>\*/*}  <**FatherMemo**/>  </>  ) }  **export default** *App* |

## 我们在父组件里面使用useState生成count和setCount，添加一个按钮，在点击事件里面调用setCount然后渲染到h3标签中

|  |
| --- |
| **//FatherMemo.jsx**  **import *React***, {*useState*} **from 'react'**; **import** *ChildMemo* **from "./ChildMemo"**;  **const** *FatherMemo* = () => {  ***console***.log(**'父组件'**)  **let** [count,setCount] = *useState*(0)  **return** (  <**div**>  <**h3**>{count}</**h3**>  <**button onClick=**{()=>setCount(++count)}>add</**button**>  <**ChildMemo**/>  </**div**>  ); };  **export default** *FatherMemo*; |

## 运行项目，发现当我们点击按钮时，即使子组件数据没有更新，也一起渲染了

|  |
| --- |
|  |

## .解决办法？

## 答：先使用React.memo把组件处理后，再把结果返回而不是直接返回组件

### 我们把上面的子组件的代码修改一下,

|  |
| --- |
| **//ChildMemo.jsx**  **import *React*** ,{*memo*} **from 'react'**;  **const** ChildMemo = () => {  ***console***.log(**'================子组件'**)  **return** (  <**div**>  <**h3**>child memo</**h3**>  </**div**>  ); };  **export default** *memo*(ChildMemo); |

### 先把memo函数解构出来，然后把子组件作为参数传递给memo函数，再把返回值导出

### 然后测试，发现点击按钮就只有父组件在渲染

|  |
| --- |
|  |

### 注意：当子组件的数据发生改变，它还是需要被渲染的，如我们修改一下父组件的代码，个体子组件传递一个count

|  |
| --- |
| **import *React***, {*useState*} **from 'react'**; **import** ChildMemo **from "./ChildMemo"**;  **const** *FatherMemo* = () => {  ***console***.log(**'父组件'**)  **let** [count,setCount] = *useState*(0)  **return** (  <**div**>  <**h3**>{count}</**h3**>  <**button onClick=**{()=>setCount(++count)}>add</**button**>  <**ChildMemo count=**{count}/>  </**div**>  ); };  **export default** *FatherMemo*; |

### 子组件代码做相应的修改

|  |
| --- |
| **import *React*** ,{*memo*} **from 'react'**;  **const** ChildMemo = (props) => {  **let** {count} = props  ***console***.log(**'================子组件count：'**,count)  **return** (  <**div**>  <**h3**>child memo</**h3**>  <**p**>Count:{count}</**p**>  </**div**>  ); };  **export default** *memo*(ChildMemo); |

### 此时每点击一次按钮，两个组件都会渲染，因为数据的确发生了变化

|  |
| --- |
|  |

### 好，我们把代码还原。现在不传数据我们给父组件添加一个方法并且传递一个给子组件，

|  |
| --- |
| **import *React***, {*useState*} **from 'react'**; **import** ChildMemo **from "./ChildMemo"**;  **const** *FatherMemo* = () => {  ***console***.log(**'父组件'**)  **let** [count,setCount] = *useState*(0)  **let** sayHello = ()=>{  ***console***.log(**'hello from dad'**)  }  **return** (  <**div**>  <**h3**>{count}</**h3**>  <**button onClick=**{()=>setCount(++count)}>add</**button**>  <**ChildMemo sayHello=**{sayHello} />  </**div**>  ); };  **export default** *FatherMemo*; |

### 我们发现此时点击按钮，即使子组件没有接收父组件的方法，子组件也会一起渲染

|  |
| --- |
|  |

### 这是为什么?我们只修改了count的值并没有修改函数，那它为什么会把子组件也渲染？

### 答：因为我们修改了父组件的数据，父组件就会重新渲染，每一次渲染sayHello函数的地址都不一样，所以传递给子组件的数据其实也是改变了，需要子组件要更新，需要重新渲染。

### 那这个问题这么解决？

### 需要使用一个叫做useCallback的Hook，他是专门用来缓存函数的，我们把父组件的代码修改一下

|  |
| --- |
| **import *React***, {*useCallback*, *useState*} **from 'react'**; **import** ChildMemo **from "./ChildMemo"**;  **const** *FatherMemo* = () => {  ***console***.log(**'父组件'**)  **let** [count,setCount] = *useState*(0)  **let** sayHello = ()=>{  **let** msg = **'hello from dad'**;  ***console***.log(msg)  **return** msg  }  **var** sayHelloCb = *useCallback*(sayHello,[]);  **return** (  <**div**>  <**h3**>{count}</**h3**>  <**button onClick=**{()=>setCount(++count)}>add</**button**>  <**ChildMemo sayHello=**{sayHelloCb} />  </**div**>  ); };  **export default** *FatherMemo*; |

### 在子组件中接受函数并且使用

|  |
| --- |
| **import *React*** ,{*memo*} **from 'react'**;  **const** ChildMemo = (props) => {   ***console***.log(**'================子组件'**)  **return** (  <**div**>  <**h3**>child memo</**h3**>  <**span**>Message:{props.**sayHello**()}</**span**>  </**div**>  ); };  **export default** *memo*(ChildMemo); |

### 注意：我们在父组件中用useCallback这个Hook给函数包装了一层，但是到了子组件里面看不见包装层，只有原来的函数

### 此时的确可以避免不必要的渲染

|  |
| --- |
|  |

# useContext，多次嵌套的组件间传递数据

### 我们在src下面新建一个文件夹13multilayer-comp-communication，在里面新建GrandFather.jsx，Father.jsx和Son.jsx,用快捷键rsc创建骨架

|  |
| --- |
| //GrandFather.jsx  **import *React* from 'react'**; **import** *Father* **from "./Father"**;  **const** *GrandFather* = () => {  **return** (  <**div style=**{{**border**:**'1px solid deeppink'**}}>  <**h3**>GrandFather</**h3**>  <**Father**/>  </**div**>  ); };  **export default** *GrandFather*; |
| //Father.jsx  **import *React* from 'react'**; **import** *Son* **from "./Son"**;  **const** *Father* = () => {  **return** (  <**div style=**{{**border**:**'1px dashed skyblue'**}}>  <**h4**>father</**h4**>  <**Son**/>  </**div**>  ); };  **export default** *Father*; |
| //Son.jsx  **import *React* from 'react'**;  **const** *Son* = () => {  **return** (  <**div style=**{{**background**:**"#cccccc"**,**color**:**'deeppink'**}}>  son  </**div**>  ); };  **export default** *Son*; |

### 然后在App.jsx中添加对爷爷组件的引用

|  |
| --- |
| **import** { *useState* } **from 'react'** *// import reactLogo from './assets/react.svg' // import viteLogo from '/vite.svg' // import './App.css'* **import** *FuncComponent* **from "./01-compoment-basic/FuncComponent"**; **import** ClassComponent **from "./01-compoment-basic/ClassComponent"**; **import** ClassEvent **from "./02-class-component-events/ClassEvent"**; **import** ClassState **from "./03class-component-state/ClassState"**; **import** ClassComputed **from "./04class-computed-property/ClassComputed" import** ClassLifeCycle **from "./05class-comp-life-cycle/ClassLifeCycle"**; *// import SideMenu from "./side-menu/SideMenu";* **import** SideMenu2 **from "./side-menu-v2/SideMenu2"**; **import** ComponentStyle **from "./06component-style/ComponentStyle"**; **import** *FunctionState* **from "./07readct-hook-useState/FunctionState"**; **import** *FunctionLifeCycle* **from "./08react-Hook-useEffect/FunctionLifeCycle"**; **import** *FunctionComputed* **from "./09react-Hook-useMemo/FunctionComputed"**; **import** *FunctionComputed2* **from "./09react-Hook-useMemo/FunctionComputed2"**; **import** *FunctionRef* **from "./10react-Hook-useRef/FunctionRef"**; **import** *TodoList* **from "./exercise-todoList/TodoList"**; **import** *FunctionProps* **from "./11react-props/FunctionProps"**; **import** *FatherProp* **from "./11react-props/FatherProp"**; **import** *FatherEvent* **from "./12comp-communication/FatherEvent"**; **import** *GrandFather* **from "./13multilayer-comp-communication/GrandFather"**;  **function** *App*() {  **let** props={  **name** :**'jack'** ,  **age**:35,  **gender**:**'male'** ,  **money**:9000000  }  **return** (  <>  {*/\*<h1>React demo3</h1>\*/*}  {*/\*<FuncComponent/>\*/*}  {*/\*<ClassComponent/>\*/*}  {*/\*<ClassEvent/>\*/*}  {*/\* <ClassState></ClassState>\*/*}  {*/\* <ClassComputed></ClassComputed>\*/*}  {*/\* <ClassLifeCycle></ClassLifeCycle>\*/*}  {*/\* <SideMenu></SideMenu>\*/*}  {*/\* <SideMenu2/>\*/*}  {*/\* <ComponentStyle></ComponentStyle>\*/*}  {*/\* <FunctionState></FunctionState>\*/*}  {*/\* <FunctionLifeCycle></FunctionLifeCycle>\*/*}  {*/\* <FunctionComputed></FunctionComputed>\*/*}  {*/\* <FunctionComputed2></FunctionComputed2>\*/*}  {*/\* <FunctionRef/>\*/*}  {*/\*<TodoList/>\*/*}  {*/\* <FunctionProps name = 'jack' age={30} gender='male' money={100000} />\*/*}  {*/\* <FunctionProps {...props} />\*/*}  {*/\* <FatherProp/>\*/*}  {*/\* <FatherEvent/>\*/*}  <**GrandFather**/>  </>  ) }  **export default** *App* |

### 测试一下，效果如下

|  |
| --- |
|  |

### 需要新建一个叫做context.js的js文件

reactdemo3-components/src/13multilayer-comp-communication/context.js

|  |
| --- |
| **import** {*createContext*} **from "react"**; **export default** *createContext*(**null**) |

### 在爷爷组件里面添加代码把数据传递给爸爸，需要使用<Context.Provider>和</Context.Provider>把爸爸包裹起来，然后给这个Provider添加value属性实现数据传递，注意这个数据是一个对象，可以传递多个状态

|  |
| --- |
| **import *React***, {*useState*} **from 'react'**; **import** *Father* **from "./Father"**; **import** Context **from './context'  const** *GrandFather* = () => {  **let** [count ,setCount] = *useState*(0)  **return** (  <**div style=**{{**border**:**'1px solid deeppink'**}}>  <**h3**>GrandFather</**h3**>  <**Context.Provider value=**{{count,setCount}}>  <**Father**/>  </**Context.Provider**>  </**div**>  ); };  **export default** *GrandFather*; |

### 注意，只需要把father组件作为提供者，标签father组件里面包含son子组件，son就可以 使用Context里面的状态，提供者不能作为Context的使用者，它在页面中不可见

### 我们在Son里面添加接受数据的代码，需要用到一个useContext的Hook

|  |
| --- |
| **import *React***, {*useContext*} **from 'react'**; **import** Context **from "./context"**;  **const** *Son* = () => {  **let** {count,setCount} = *useContext*(Context)  **return** (  <**div style=**{{**background**:**"#cccccc"**,**color**:**'deeppink'**}}>  <**h5**>Son</**h5**>  <**button onClick=**{()=>setCount(--count)}>-</**button**>  <**span** >{count}</**span**>  <**button onClick=**{()=>setCount(++count)}>+</**button**>  </**div**>  ); };  **export default** *Son*; |

#### 效果：点击+会增加计数，点击-会减小计数

|  |
| --- |
|  |

### 我们可以把代码修改一下，再做另外一个案例

#### GrandFather.jsx

|  |
| --- |
| **import *React***, {*useState*} **from 'react'**; **import** Context **from './context.js' import** *Father* **from "./Father"**;  **const** *GrandFather* = () => {  **let** [count,setCount] = *useState*(0)  **let** [num,setNum] = *useState*(0)   **return** (  <**div**>  <**Context.Provider value=**{{count, setCount,num,setNum}}>  <**Father**></**Father**>  </**Context.Provider**>   </**div**>  ); };  **export default** *GrandFather*; |

#### Father.jsx

|  |
| --- |
| **import *React* from 'react'**; **import** *Child1* **from "./Child1"**; **import** *Child2* **from "./Child2"**;  **const** *Father* = () => {  **return** (  <**div**>  <**Child1**></**Child1**>  <**Child2**></**Child2**>  </**div**>  ); };  **export default** *Father*; |

#### Child1.jsx

|  |
| --- |
| **import *React***, {*useContext*} **from 'react'**; **import** Context **from './context'  const** *Child1* = () => {  **let** {num,setNum}= *useContext*(Context)  **return** (  <**div**>   <**button onClick=**{()=>setNum(++num)}>+</**button**>  {num}  <**button onClick=**{()=>setNum(--num)}>-</**button**>  </**div**>  ); };  **export default** *Child1*; |

#### Child2.jsx

|  |
| --- |
| **import *React***, {*useContext*} **from 'react'**; **import** Context **from './context'  const** *Child2* = () => {  **let** {count,setCount} = *useContext*(Context)  **return** (  <**div**>   <**button onClick=**{()=>setCount(++count)}>+</**button**>  {count}  <**button onClick=**{()=>setCount(--count)}>-</**button**>  </**div**>  ); };  **export default** *Child2*; |

# react有3个Hook可以用来缓存东西的

## useMemo用来缓存数值数据

## memo用来缓存组件

## useCallback用来缓存函数