# React Hook

## 路由的基本使用

代码示例：

import React, {Component} from "react";

import {Link,Route,Routes} from 'react-router-dom'

import Home from "./components/Home";

import List from "./components/List";

class App extends Component {

  render( ) {

    return (

      <div>

        <div>

          <Link to="Home"> 首页</Link>-------

          <Link to="/List"> 列表</Link>

        </div>

        <div>

          <Routes>

            <Route path="/Home" element={<Home />}   />

            <Route path="/List" element={<List />}  />

            </Routes>

        </div>

      </div>

    )

  }

}

export default App

主要是导入 react-router-dom中的link route routes三个方法

Link 有to属性类似a标签里面的 herf to到的就是指定跳转的链接

Routes要包裹route 里面的内容之前是用component={Home} 现在这个版本使用

<Routes>

            <Route path="/Home" element={<Home />}   />

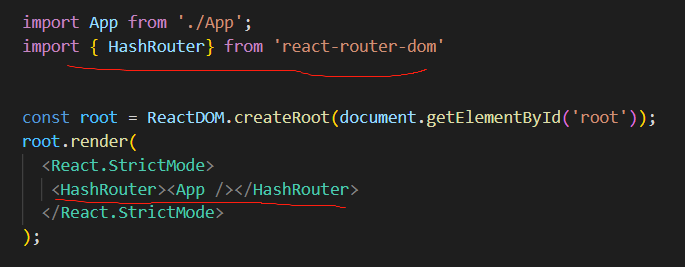
            <Route path="/List" element={<List />}  />

            </Routes>

这种格式

Route里面Path 是匹配规则 其实本质是为了匹配 to里面内容的正则 element属性是匹配到后展示

开启路由用



## Exact属性

在 react-router中，exact属性仍然非常有用。它用于确保路由只匹配精确的 URL，而不是任何子路由。这对于确保您的应用程序在导航时正确呈现组件非常重要。如果您不使用 exact属性，则可能会导致您的路由匹配到不正确的 URL，从而呈现错误的组件。

## 路由匹配规则

Switch 和 Routes是 react-router 中的两个不同的组件，它们用于处理路由匹配和呈现。

Switch 组件用于在多个路由之间进行选择，只呈现第一个匹配的路由。例如，如果您有两个路由 /home 和 /about，并且用户导航到 /home，则只会呈现与 /home匹配的路由。Routes

组件用于定义应用程序的路由层次结构，并将 URL 路径映射到组件。例如，如果您有一个路由 /home，则可以使用 Routes 组件将其映射到 Home组件。因此，Switch 和 Routes是 react-router中的两个不同的组件，它们用于不同的目的

默认模糊匹配 如果要精确匹配使用 exact 从上往下执行 匹配到几个展示几个

使用404 只指定js不用指定 path即可 原理就是所有的都可以匹配上404

react-router 是一个用于构建单页应用程序（SPA）的 JavaScript 库。它提供了一组组件和 API，用于处理应用程序的路由和导航。使用 react-router，您可以将 URL 路径映射到组件，并在用户导航时呈现正确的组件。以下是 react-router 中的一些核心概念：

BrowserRouter：用于将 URL 路径映射到组件的根组件。

Route：用于将 URL 路径映射到组件。

Switch：用于在多个路由之间进行选择，并只呈现第一个匹配的路由。

Link：用于在应用程序中导航到不同的 URL。

NavLink：类似于 Link，但可以根据当前 URL 自动添加活动类。

Redirect：用于将用户重定向到不同的 URL。

useParams：用于从 URL 中提取参数。

useLocation：用于访问当前 URL 的位置对象。

useHistory：用于访问浏览器历史记录。

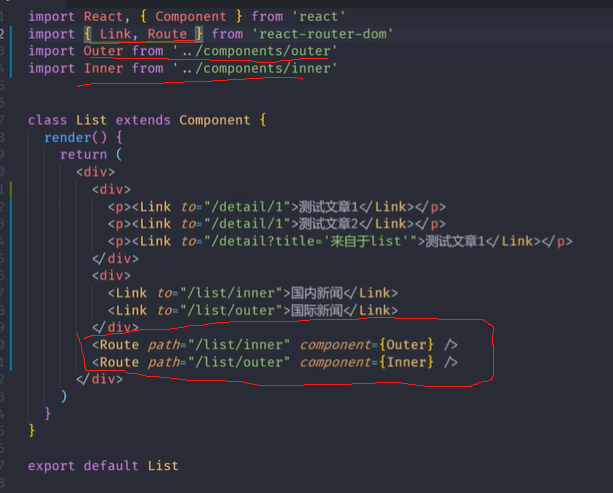
## 路由的动态传参

1路由参数占位符

2触发操作的时候传递具体的参数

3在具体的组件当中使用传递的参数

## 路由嵌套

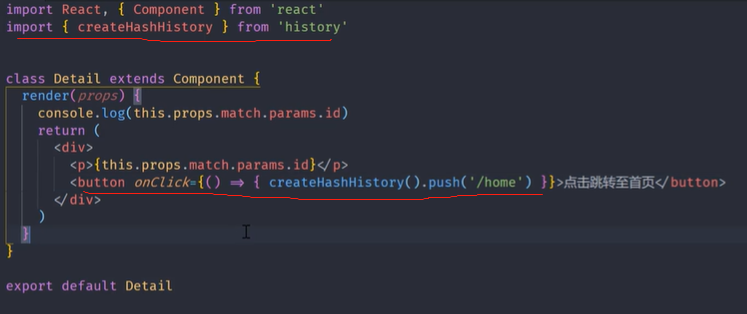




上面的写法对比 下面的这种写法是用绝对路径来描述 to代表的url和path代表的匹配规则path 这样的话嵌套它的组件无论怎么改名字都不会影响inner和outer的使用

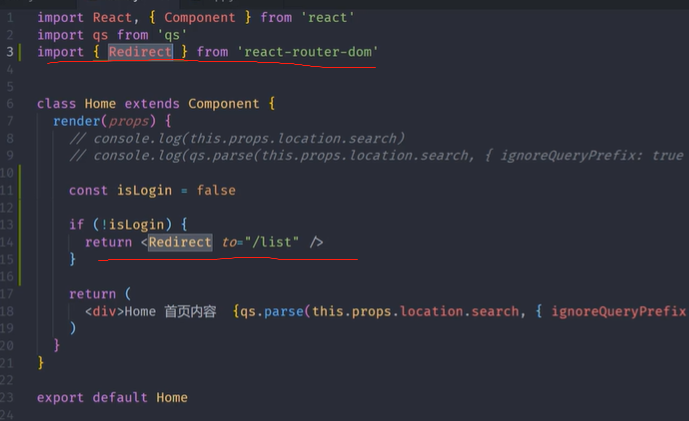
## 路由跳转和重定向

路由跳转



原生js原理 用createHashHistory（）方法进行跳转页面 该方法下有个push（）方法 里面传入想要跳转的地方 在push()放入想去的url

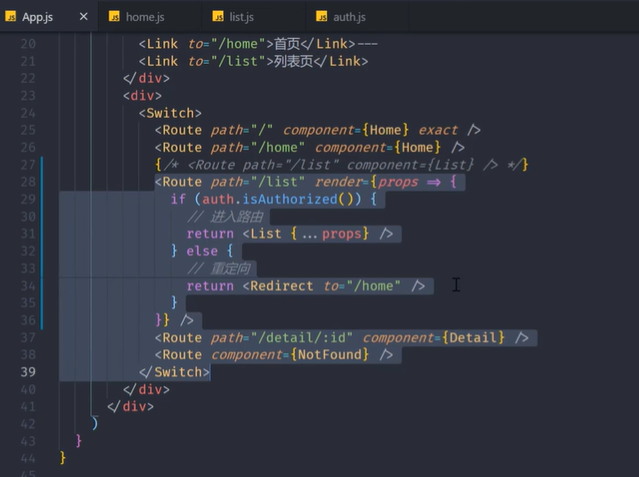
Redirect重定向方法



用redirect to重定向到指定url

## 路由守卫

通过什么前置条件 才能访问指定的路由



路由设置在父组件设置

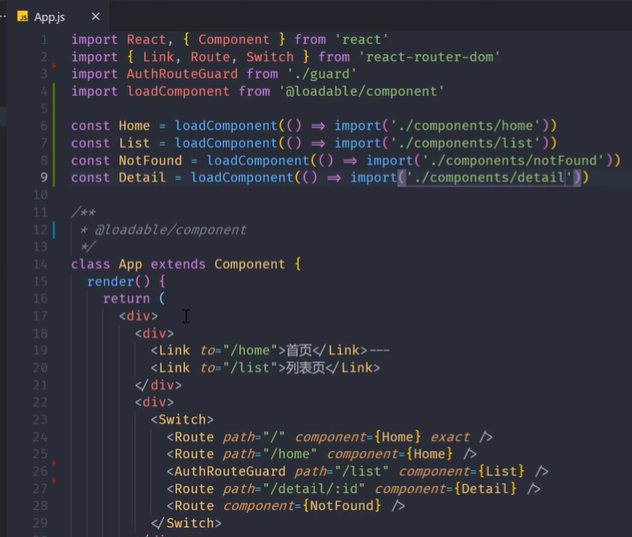
以组件的形式封装路由守卫



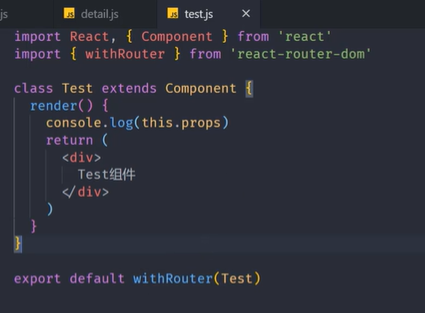


## 懒加载

懒加载原理是将路由组件拆分到一个文件当中 ，只加载当前要用到的路由，提高网站首次加载速度



## 非路由组件传递路由信息



WithRouter 用来传递非路由信息

## React Hook

Hook是一个特殊的函数，让函数式组件具有类组件的特性。

Hook出现的原因：1、学习成本较低 2、数据共享不需要redux包 3、多个业务逻辑代码可能会存在于同一个生命周期函数中

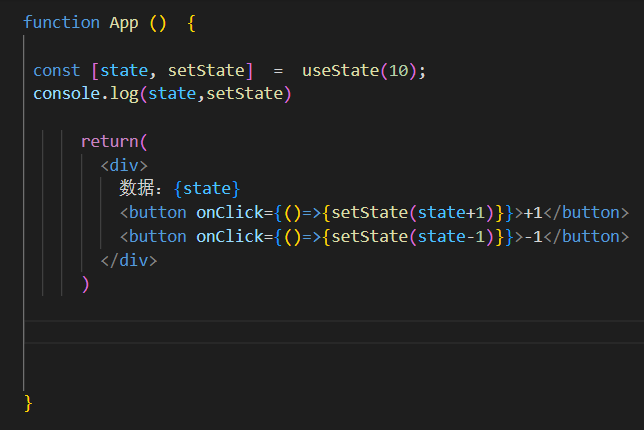
Hook只能在函数式组件中使用，并且写在函数组件的最外层，并且不能放在语句中

## useState

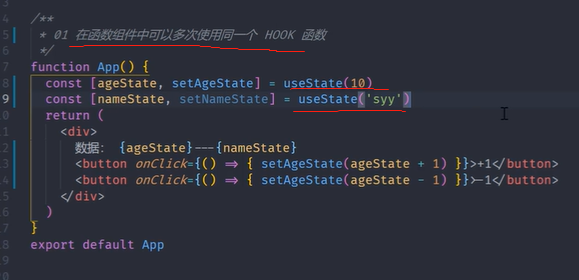
一个可以在函数组件中保存状态的hook函数

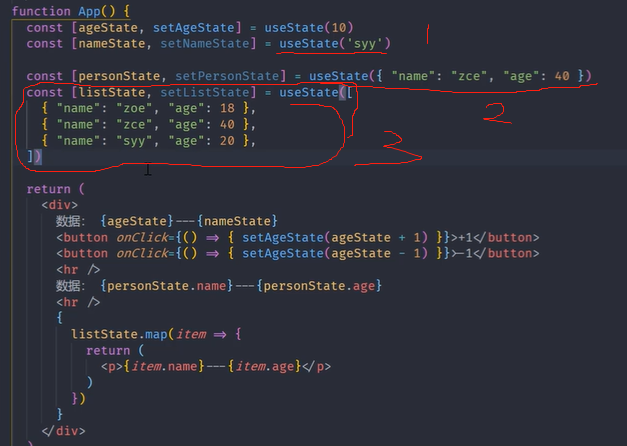
参数：保存状态的初始值

返回值： 数组 【第一个元素当前保存的状态，修改当前保存状态的方法】、

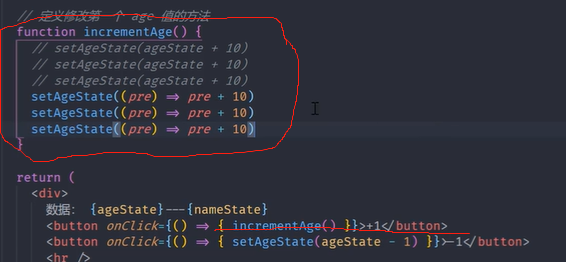


useState（）括号里面的参数 保存的是初始值 const 【state，setState】是useState的返回值 是数组形式 第一个返回值是元素当前的保存状态，因为是处理状态的hook 第二个返回值是修改当前保存状态的方法，第二个返回值里面的参数是第一个返回值对应操作





UseState里面可以留有简单类型 对象类型 数组类型 甚至数组里面嵌套也可以 数据里面嵌套复杂类型用map（）方法解析出来就行了 其实都在 就是看怎么展示了



UseState本身就是异步操作 如果不使用回调函数 无论执行几次最后都只会执行最后一次 先进先出原则 需要在其内部使用回调函数callback这样因为回调了 所以每次必须执行

Set操作异步的，不能直接修改state的原始值

修改数据 不要直接改变第一个返回值里面的数据 通过调用第二个返回值的数据来修改第一个数据。

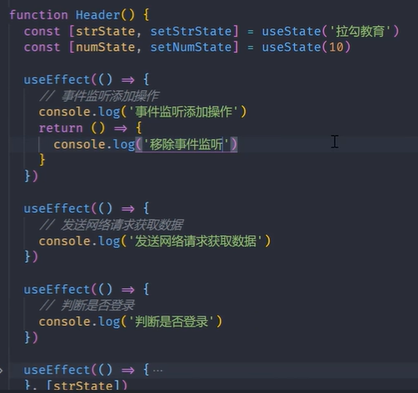
## UseEffect

在挂载 、 更新、 卸载时执行

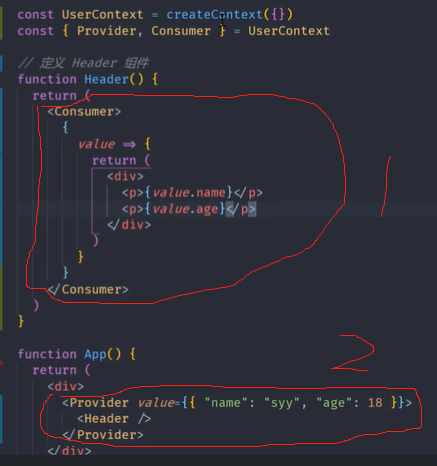


UseEffect(()=>{},[]) useeffect里面有两个参数 第一个参数是个函数体 挂载与更新是在函数执行时， return的返回值 也是个函数体 在卸载的时候执行 第二个参数是个数组【】 里面放依赖，如上【strState】 只有当 strState执行了 函数体才会执行卸载 才会有再次的挂载和更新

优点：代码函数化 一个代码做一件事 可以分开写 其实本质就是所有的监听都可以同步进行 一件事一件事的搞就完事了



## UseContext



在类组件里 通过provider包裹 子组件的方式来形成 父子组件的关系 再由父子件向子组件传递参数 App是生产者 在子组件Header里 通过consumer标签来消费父组件app传来的数据 但是如果 消费者过多 就会出现嵌套的这种情况 书写太复杂

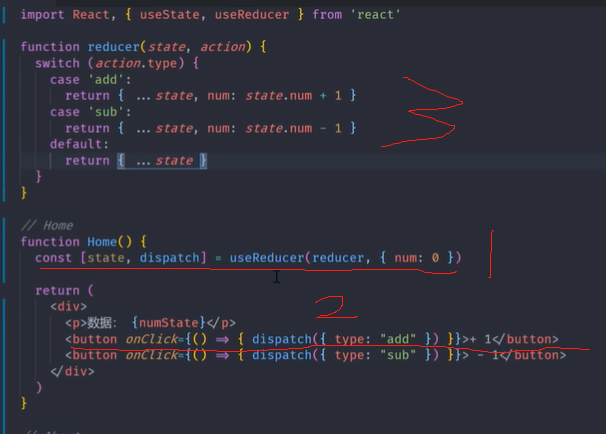
所以用useContext钩子来处理这种情况



## UseReducer（）

UseReducer是为了简化useState的重复操作

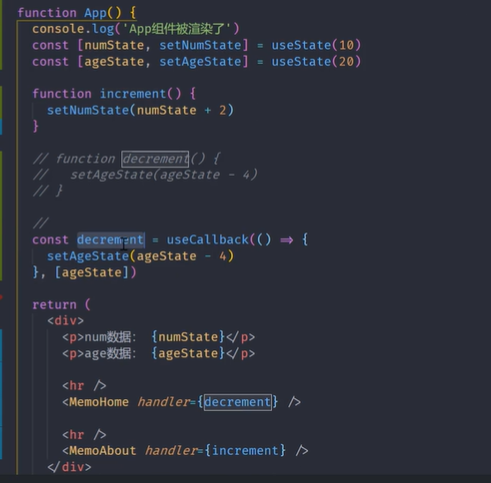
Const [state,dispatch（指令触发）] =reducer(reducer,{num:0}) reducer是将来对数据操作的业务逻辑代码 第二个是状态的初始数据



## UseCallback

用于优化代码，考虑渲染的性能问题





以上述代码为例 About组件中调用了decrement方法 如果用useCallback（）钩子 ，传入两个参数，第一个为函数型 负责阐述状态变化 第二个是数组【】里面填入依赖。只有当父组件中的ageState变化 About才会渲染

## UseMemo

useMemo和useCallback都是React中的hooks，它们的主要区别在于它们的返回值。useMemo返回一个memoized值，而useCallback返回一个memoized的函数。useMemo的使用场景是当你有一个昂贵的计算，你不想在每次渲染时都重新计算它时，你可以使用useMemo。useMemo接收两个参数：一个函数和一个依赖项数组。当依赖项数组中的任何一个值发生变化时，useMemo将重新计算函数并返回新的memoized值。否则，它将返回上一次计算的memoized值。

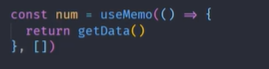
useCallback的使用场景是当你需要将一个回调函数传递给子组件时，你可以使用useCallback来避免在每次渲染时都创建一个新的函数。useCallback接收两个参数：一个回调函数和一个依赖项数组。当依赖项数组中的任何一个值发生变化时，useCallback将返回一个新的memoized函数。否则，它将返回上一次memoized的函数。

Const a = useMemo（（）=》{

Return（）=》{}}，【】）

通过调用useMemo（）方法返回一个函数, 这个函数被保存在变量a里面，a依赖于【】中的数，数组中的数不变 a 不变

UseMemo会返回更多的类型，对象 数组或者其他类型 useCallback返回的就是函数。两者其实没有本质区别只是useMemo范围更大



## UseRef

可以用于获取元素

还可以保存数据

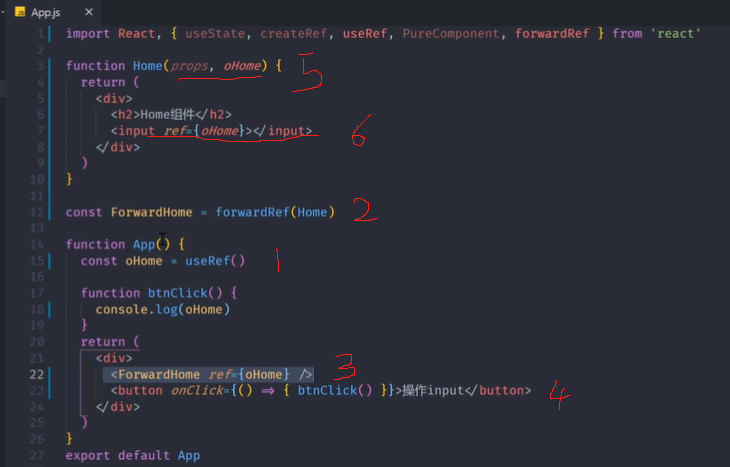
保存的数据除非我们手动修改，否则不会改变

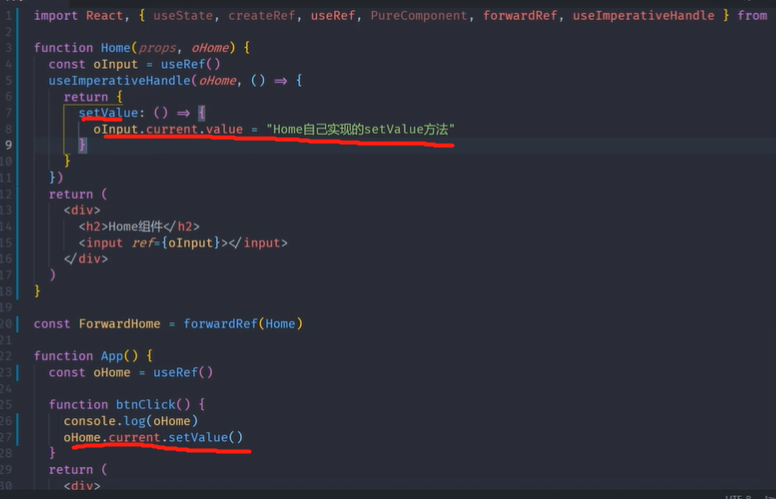


运行结果 {numState}里面的值会随着点击一直增加 但是 obj.current不会

## useImperativeHandle

例子：用父组件获取子组件里面input的内容





useImperativeHandle实际上是拦击器 上述例子父组件要操控子组件 但是子组件并不需要让自己的所有内容都受父组件的操控 useImperativeHandle钩子就是一个拦截器或者说是过滤器，自己可以设置自己想暴露出的内容

useImplerativeHandle( a（被访问的内容）, ()=>{

Return {

想要交给父组件的方法

}

} )

## UseLayoutEffect

UseLayoutEffect比useEffect先执行，后者在渲染时才执行，前者在dom发生个改变时就执行。在对dom元素进行修改时 使用useLayoutEffect

UseLayoutEffect（（）=>{

A return () => { B }

},[]）

## 自定义Hook

UseMoumoumou（）当用use开头就会被当成一个函数式组件 就是个钩子函数