**react**



react-route-dom



redux



三大核心

1.单一数据源：整个应用的state被存储在一棵object tree中

2.state是只读的，唯一改变state的方法就是触发action

3.使用纯函数来执行修改，为了描述action是如何改变state tree 的需要编写reducer



组成



一：state--状态，为我们传递的数据



简单分类

1.DomainDate:可以理解为服务器端的数据，比如token

2.UI State:决定当前UI展示的状态

3.App State: APP级别的状态，比如路由状态信息



二：Action



特点

1.本质是一个对象,包含两个属性：type:标识属性，多个action用type区分，payload:表示动作携带的参数

2.包含一个type属性表示要执行的动作

3.项目中常常创建一个函数返回action对象

三：reducer:函数，接收两个参数：参数一：sate,参数二：action,返回值被store接收  
const  reducer  = (state = 0, action ) => {  
    switch (action.type) {  
     case 'add':  return  state +1  
     case 'addN':  return state + action.payload  
     default:  return state  
    }  
}



四：Store:对象，将action和reducer联系在一起

维护应用的状态，获取状态  getState()获取state

状态更新时:需要分发action，dispatch()发送action

通过subscribe()来注册监听状态变化：  
store.subscribe()订阅者，当state发生变化时store会执行它

通过subscribe()返回值来注销监听状态变化

使用：import  { createStore } from  'redux'  
          const   store =  createStore(传递reducer)



react-redux

provider:包裹在根组件的最外层，使所有的子组件都可以拿到state  
,接收store作为props,然后通过context往下传递，这样react中任何组件都可以通过conext来获取store

connect:provider内部组件如果想要使用state中的数据，就必须要通过connect包裹

使用：1.在app.js中引入provider,  
import {provider} from 'react-redux'  
import store  from './store'  
<Provider   store={store}>根组件</Provider>  
2.导入connect方法  
import  {connect} from '

代码执行过程：  
创建store --->  redux调用一次reducer  ---> （action的type是一个随机值）获取state的初始值  --->  该初始值作为下次调用store.getState()的prestate  
更新状态   ---->  store.dispatch(action)  ---->调用reducer(传入上一次的状态和action),  --->  计算出新的状态并返回   --->  reducer执行完毕将最新的状态交给store，store替换掉旧的状态



类组件



state状态

定义方法一：  
const App extends Component{  
    state =｛｝  
}

定义方法二：  
const App extends Component{  
    constructor ()  {  
        super()  
        this.state = {  
             username: 'lisa'  
        }  
    }  
}

兄弟组件：状态提升，将被修改的状态定义在父组件中，



父子之间



props

children:只要组件有子组件，就会自动作为children  
如：<App>文本子组件</App>  
class  App  extends  Component{  
   console.log(this.props.children)  
}

子传父：  
父组件定义一个回调函数F，用于接收数据  
将该函数作为属性值传递给子组件



antd

表单：  
（1）表单域的获取  
        class:通过React.createRef()  
        function: 通过Form.useForm()



组件通讯

props:接收传递给组件的数据

父组件传递数据给子组件，props:接收传递给子组件的数据  
类组件：constructor (props) {  super(props)  }  
 函数组件： function  (props) {}

子传父：  
class  Parent extends  React.Component{  
    getChildMsg = ( msg ) => {  
         console.log('接收的子组件的数据', msg)  
    }  
    render  () {  
         return (<div><child  getMsg={this.getChildMsg}></div>)  
     }  
}

自由主题

自由主题



props和高阶组件



组件复用



方式一：props模式

问题一：关于复用的state参数的获取：  
在使用时添加一个属性，属性值为函数，函数的参数即为  
<Mouse render="{(state)=>{}}'></Mouse>



方式二：高阶组件，采用包装模式实现逻辑复用

使用步骤：  
1.创建一个函数，名称约定以with开头  
2.指定函数参数，参数应该以大写字母开头  
3.在函数内部创建一个类组件，提供复用的状态逻辑  
4.在该组件中，渲染参数组件，同时将状态通过props传递给参数组件  
function  withMouse(WrapperComponent) {  
    class  Mouse  extends  React.Component()  
    return  Mouse  
}  
  
// Mouse组件内的render方法：  
return     <wrapperComponent   {...this.state}></WrapperComponent>



设置displayName

为了解决组件复用后得到的各组件名称相同的问题，因为默认情况下，React使用组件名称作为displayName

设置方法:  
组件名.displayName = '高阶组件函数名${getDisplayName(高阶组件函数参数)}'  
Mouse.displayName = 'WithMouse${getDisplayNmae(WrapperComponent)}}'  
function  getDisplayName(WrapperComponent)  {  
    return   WrappedComponent.displayName  ||  WrappedComponent.name  ||  'Component'  
}



传递props

传递方式：<WrappedComponent {...this.state}  {...this.props} ></WrappedComponent>



Hooks

概念：Hooks是一些可以让你在函数组件内使用状态和生命周期等特性的函数，



hook规则

不要在循环，条件或者嵌套函数中调用hook,因为需要确保Hook在每一次渲染中都能按照同样的顺序被调用。  
为什么需要保证相同的顺序呢？因为当存在多个hook时，比如多个useState时，组件是如何保证修改的是同一个状态呢？答案是hook的书写顺序。  
```js  
import React, {useState,useEffect, useRef} from 'react'  
const Study = () => {  
  const [name, setName] = useState('lisa')  
  if (name !== '') {  useEffect(()=> {  console.log(name)  })  }  
  setName('')  
  const [curname, setCurname] = useState('rose')  
  useEffect(() => {  console.log(name + '  ' + curname)  })  
  console.log('更新')  
  return ( <div><div onCli></div></div> )  
}  
export default Study  
```



useState

1.导入useState函数  
2.调用useState函数，传入初始值，返回值为状态和修改状态的函数  
useState导致函数更新，每次时使用的state都是最新的state,而非初始state



初始值类型

1.基本类型：直接传入数据  
2.引用类型：使用函数式更新加扩展运算符，不然会导致数据无法更新成功！！！



写法

写法一：const 【 状态，函数】= useState(初始值）

写法二：const  【状态，函数】=useState(回调函数）  
useState内传入的回调函数的入参：初始值，返回值：初始值经过加工后的新值

多个状态：const 【状态，函数】=  useState({状态1，状态2}）



惰性初始化

如果你要更新的state的前后值相同时，react将跳过子组件的渲染并且不会触发effect的执行



useEffect

函数式组件的主作用：根据数据渲染DOM  
副作用：数据请求，开启定时器，事件监听，localStorage存取等

格式：useEffect(参数一，参数二）  
useEffect( () => {  
  // 副作用函数的内容  
  return 清理函数  
}，参数二）  
参数一：副作用函数，该函数必须为同步函数，如果要发送异步请求可以在函数内部再声明一个函数  
// 副作用函数的返回值的作用：清理副作用，场景：启动和清理定时器  
// 副作用函数的返回值清理函数的执行时机：  
组件卸载或者下一次副作用函数执行前  // componentWillUnmount,,,,,,,,因为组件通常会被多次渲染，因为useEffect是根据state状态来发生更新的，所以在执行下一个effect之前，上一个effect要被清除即return的函数要被清除  
若参数二为空：则每次更新时都会执行useEffect函数   //相当于componentDidMount + DidUpdate  
若参数二为空数组[], 执行时机：只执行一次 // 场景ajax请求   //相当于componentDidMount  
若参数二指定了依赖项：执行时机：初始化执行一次，依赖项值变化时再执行一次



useRef

1.调用useRef(初始值)函数，得到引用对象  
2.将引用对象通过ref属性添加给任意元素  
3.通过引用对象.current.value获取值



使用场景二：在多次渲染之间共享数据  
1.const nameRef =  useRef()  
2.nameRef.current = 要被共享的数据

应用场景：倒计时  
const timer = useRef()  
const [cout, setCount] = useState()  
useEffect(() => {  
    timer.current = setInterval(()=>{  
        setCount((count)=>{count-1})  
    },1000)  
  return clearInterval(timer)  
})



自定义hooks

定时器在组件销毁时可能并没有清理干净，需要在useEffect中使用清理函数清理



useContext

使用步骤：  
1.导入并调用createContext方法，得到Context对象，然后再导出context对象  
import  {createContext} from 'react'  
export  const  Context = createContext()  
  
2.使用Provider组件包裹根组件，并通过value属性传递共享数据  
<Context.provider  value={}>  
    根组件  
</Context.Provider>  
  
3.在任意后代组件中，如果希望获取公共数据：  
导入useContext,调用useContext得到value



const [state, dispatch] =  useReducer(reducer, initialArg, init)

1.reducer:处理状态更新的reducer  
2.initialArg:状态初始值  
3.init状态初始化函数



useCallback

useCallback的使用场景：在函数式组件中，定义在组件内函数会随着状态值的更新而重新渲染，函数中定义的函数会被频繁定义，在父组件的通信中这样是非常消耗性能的，使用useCallback结合memo可以有效的减少子组件更新的频率，提高效率  
即：当父组件中的状态发生变化时，子组件也会随之更新，即使子组件么有用到发生改变的状态，所以可以使用React.memo()和useCallback()



useMemo

基本使用：const  memoizedValue = useMemo( () => computedValue(a, b), [a, b])  
第一个参数用于处理耗时计算并返回需要记录的值  
第二个参数为数组，用于指定被记忆函数更新所依赖的值

useMemo和useCallback的区别  
1.useMemo传入的函数内部需要有返回值  
2.useMemo只能声明在函数式组件内部,但是react.memo()是直接包裹在函数式组件的外部，类似于computed

useReducer



react-redux



使用：providere.useSelector,useDispatch来操作store

获取值：  
import  {useSelector  }  from  'react-redux'  
const xx = useSelector(  state  =>  return  XX)  
添加订阅者：  
import  { Provider } from  'react-redux'  
import store from './store/index.js'  
<Provider  store={store}></Provider>  
修改值：  
import {useDispatch } from 'react-redux'  
const  dispatch  =  useDispatch()  
dispatch(action)



state  
(修改state的本质不是直接修改原来的状态，而是创建一个新的状态，  
根据不同的状态类型，我们将修改方式分成了三种  
)

修改state:  
(1)状态为基本数据类型  
        直接赋新值this.setState({title: '登录'})  
(2)状态为数组  
       通过concat，展开运算符，Object.assgin()等,,this.setState( preState => type: preState.type.concat([ 'type1' ]))  
(3)replaceState()

setState真的是异步吗：  
https://juejin.cn/post/6844903636749778958



脑图模式

字数：4/8915

主题数：1/140

主题数限时不限制