React学习记录

Virgil233 于 2023-12-01 20:25:02 发布

目录

一、什么是React

（1）React是一个用于构建web和原生交互界面的库

（2）使用create-react-app快速搭建开发环境：npx create-react-app react-basic

注：npx是Node.js工具命令，查找并执行后续的包命令；create-react-app是核心包（固定写法），用于创建React项目；react-basic是React项目的名称

二、JSX

（1）JSX是JS和XML(HTML)的缩写，表示在JS代码中编写HTML模板结构，是React中编写UI模板的方式
注：JSX不是标准JS语法，是JS语法扩展，浏览器本身不能识别，需要通过解析工具解析后才能在浏览器中运行
（2）通过逻辑与运算符或三元表达式实现基础条件渲染
（3）复杂条件渲染：自定义函数+if判断语句

三、基础事件绑定

on事件名称={事件处理程序}

```
function App(){
    const clickHandler = () => {
        console.log('点击按钮')
    }
    return(
        <button onClick = {clickHandler}></button>
    )
}
```

四、useState

（1）useState是一个React Hook（函数），可以向组件添加一个状态变量，影响组件渲染结果
语法：const [count, setCount] = useState(0)
useState是一个函数，返回值是一个数组
第一个参数是状态变量，第二个参数是set函数用来修改状态变量
useState参数作为count初始值
（2）修改状态规则——状态不可变：状态是只读的，应该替换它而不是修改它，直接修改状态不能引发视图更新

五、组件基础样式方案

（1）行内样式（不推荐）

```
<div style = {{color: red}}>this is div</div>
```


（2）class类名控制

```
//index.css
.foo{
    color: red;
}

//App.js
import './index.css'

function App(){
    return(
        <div>
            <span className = 'foo'>this is span</span>
        </div>
    )
}
```

(3) classnames优化类名控制：通过条件动态控制class类名显示（要先安装）

```
import classNames from 'classnames'

//nav-item是静态类名；动态类名key表示要控制的类名，value表示条件为true时类名显示
className = {classNames('nav-item', {active: type === item.type})}
```

六、受控表单绑定

使用useState控制表单状态

```
//准备一个React状态值
const [value, setValue] = useState('')

//通过value属性绑定状态，通过onChange属性绑定状态同步的函数
<input
    type = "text"
    value = {value}
    onChange = {(e) => setValue(e.target.value)}
```

```
/>
```

七、获取DOM

使用useRef钩子函数

```
//使用useRef创建ref对象，并与JSX绑定
const inputRef = useRef(null)
<input type = "text" ref = {inputRef} />

//DOM可用时，通过inputRef.current拿到DOM对象
console.log(inputRef.current)
```

八、组件通信

（1）父传子基础实现：在子组件标签上绑定属性（父组件传递数据）→子组件通过props参数接收数据（子组件接收数据）

注：props是只读对象，子组件只能读取props中的数据，不能直接修改，父组件数据只能由父组件修改

（2）父传子——prop children：将内容嵌套在子组件标签中时，父组件会自动在名为children的prop属性中接受该内容

（3）子传父：子组件调用父组件中的函数并传递参数

```
//父组件
function App(){
    const getMsg = (msg) => console.log(msg)
    return(
        <div>
            <Son onGetMsg = {getMsg} />
        </div>
    }
}

//子组件
function Son({onGetMsg}){
    const sonMsg = 'this is msg'
    return(
        <div>
            <button onClick = {() => onGetMsg(sonMsg)}>send</button>
```

```
        </div>
    )
}
```

（4）兄弟组件通信——状态提升：A组件先通过子传父的方式把数据传给父组件，父组件拿到数据后通过父传子的方式再传递给B组件
（5）跨层级组件通信：使用createContext方法创建一个上下文对象Ctx→在顶层组件中通过Ctx.Provider组件提供数据→在底层组件中通过useContext钩子函数获取消费数据

九、useEffect

（1）useEffect是一个React Hook函数，用来创建不是由事件引起而是由渲染本身引起的操作（初始化），如发送AJAX请求，更改DOM等
　　语法：useEffect(() => {}, [])，参数1是一个函数，叫副作用函数，在函数内部可以放置要执行的操作；参数2是一个数组（可选），在数组里放置依赖项，不同依赖项影响参数1函数的执行

没有依赖项	组件初始渲染一次+组件更新时执行
空数组依赖	只在组件初始渲染时执行一次
添加特定依赖项	组件初始渲染一次+特定依赖项变化时执行

（2）清除副作用：useEffect中的操作也常叫做副作用操作，如在useEffect中开启了一个定时器，在组件卸载时将定时器清理掉，就是一个清理副作用的过程。最常见的执行时机是在组件卸载时自动执行

```
useEffect(() => {
    return() => {
        //清理副作用逻辑
    }
}, [])
```

十、自定义Hook函数

（1）声明一个以use开头的函数
（2）在函数体内封装可复用的逻辑
（3）将组件中用到的状态或回调return出去（对象或数组）

（4）哪个组件中用到此逻辑，就执行这个函数，将状态和回调解构出来进行使用
（5）ReactHooks使用规则：只能在组件中或其他自定义Hook函数中调用；只能在组件的顶层调用，不能嵌套在if、for、其他函数中

十一、Redux

（1）Redux是React最常用的集中状态管理工具，可以独立于框架运行
（2）使用纯Redux实现计数器：定义一个reducer函数（根据当前想要做的修改返回一个新状态）

```
//state：管理的数据初始状态
//action：对象type标记当前想要做什么样的修改
function reducer(state = {count: 0}, action){
    if(action.type === 'INCREMENT'){
        return {count: state.count + 1}
    }
}
```

→使用createStore方法传入reducer函数，生成一个store实例对象→使用store实例的subscribe方法订阅数据变化（数据一旦变化，可以得到通知）→使用store实例的dispatch方法提交action对象，触发数据变化（告诉reducer要怎么修改数据）

```
store.dispatch({
    type: 'INCREMENT'
})
```

→使用store实例的getState方法获取最新状态数据更新到视图中
（3）Redux结合React配套工具：npm i @reduxjs/toolkit react-redux
（4）Redux+React实现计数器：
使用React Toolkit创建counterStore：

```
/*counterStore.js*/

import {createSlice} from "@reduxjs/toolkit"

const counterStore = createSlice({
    name: 'counter',
    //初始状态数据
```

```
    initialState:{
        count: 0
    },
    //修改数据的同步方法
    reducers:{
        increment(state){
            state.count++
        },
        decrement(state){
            state.count--
        },
    }
})

//解构出创建action对象的函数
const {increment, decrement} = counterStore.actions
//获取reducer函数
const counterReducer = counterStore.reducer
//导出创建action对象的函数和reducer函数
export {increment, decrement}
export default counterReducer
```

```
/*store/index.js*/

import {configureStore} from "@reduxjs/toolkit"
import counterReducer from "./modules/counterStore"

//创建根store组合子模块
const store = configureStore({
    reducer:{
        counter: counterReducer
    }
})

export default store
```

为React注入store：react-redux内置Provider组件通过store参数把创建好的store实例注入到应用中，链接正式建立

```
import store from './store'
import {Provider} from 'react-redux'

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(
    <Provider store = {store}>
        <APP/>
    </Provider>
)
```

React组件获取store中的数据：使用useSelector将store中的数据映射到组件中
React组件修改store中的数据：使用useDispatch生成提交action对象的dispatch函数

```
import {useDispatch, useSelector} from 'react-redux'
//导入创建action对象的方法
import {decrement, increment} from './store/modules/couterStore'

function App() {
  const {count} = useSelector(state => state.counter)
  //得到dispatch函数
  const dispatch = useDispatch()
  return (
    <div className = "App">
      {/* 调用dispatch提交action对象 */}
      <button onClick = {() => dispatch(decrement())}> - </button>
      <span>{count}</span>
      <button onClick = {() => dispatch(increment())}> + </button>
    </div>
  );
}
```


（5）提交action传参：在reducer的同步修改方法中添加action对象参数，在调用actionCreater时传递参数，参数被传递到action对象的payload属性上

```
/*counterStore.js*/

const counterStore = createSlice({
    name: 'counter',
    //初始状态数据
    initialState:{
        count: 0
    },
    //修改数据的同步方法
    reducers:{
        increment(state){
            state.count++
        },
        decrement(state){
            state.count--
        },
        addToNum(state, action){
            state.count = action.payload
        }
    }
})

export {increment, decrement, addToNum}
```

```
import {decrement, increment, addToNum} from './store/modules/couterStore'

function App() {
  const {count} = useSelector(state => state.counter)
  //得到dispatch函数
  const dispatch = useDispatch()
  return (
    <div className = "App">
      {/* 调用dispatch提交action对象 */}
      <button onClick = {() => dispatch(decrement())}> - </button>
      <span>{count}</span>
```

```
      <button onClick = {() => dispatch(increment())}> + </button>
    <button onClick = {() => dispatch(addToNum(10))}> add to 10 </button>
    <button onClick = {() => dispatch(addToNum(20))}> add to 20 </button>
  );
}
```

(6) 异步状态操作：

创建store写法不变，配置同步修改状态方法：

```
const counterStore = createSlice({
    name: 'channel',
    //初始状态数据
    initialState:{
        channelList: []
    },
    //修改数据的同步方法
    reducers:{
        setChannels(state, action){
            state.channelList = action.payload
        }
    }
})
```

单独封装一个函数，函数内return一个新函数，新函数中封装异步请求获取数据；调用同步actionCreater传入异步数据生成action对象，用dispatch提交：

```
const {setChannels} = ChannelStore.actions
const url = ''
const fetchChannelList = () => {
    return async (dispatch) => {
        const res = await axios.get(url)
        dispatch(setChannels(res.data.data.channels))
    }
}

```

```
export {fetchChannelList}
```

组件中dispatch写法不变：

```
const dispatch = useDispatch()
useEffect(() => {
    dispatch(fetchChannelList())
}, [dispatch])
```

十二、ReactRouter

（1）前端路由：一个路径path对应一个组件component，当在浏览器中访问一个path时，path对应的组件会在页面中渲染
（2）安装ReactRouter包：npm i react-router-dom
（3）实现：

```
import ReactDOM from "react-dom/client"
import {createBrowserRouter, RouterProvider} from "react-router-dom"

const router = createBrowserRouter([
    {
        path: "/",
        element: <div> hello </div>,
    },
]};

ReactDOM.createRoot(document.getElementById("root")).render(
    <RouterProvider router = {router}/>
);
```

（4）路由导航：路由系统中多个路由之间进行路由跳转，在跳转时可能需要传递参数进行通信
声明式导航：<Link to = "/article">文章</Link>，通过to属性指定要跳转到路由path，组件被渲染为浏览器支持的a链接，如需要传参则直接通过字符串拼接的方式拼接参数
编程式导航：通过useNavigate钩子得到导航方法，通过调用方法以命令式形式进行路由跳转


```
import {useNavigate} from "react-router-dom"
```

```
const login = () => {
    const navigate = useNavigate()
    return(
        <div>
            登录
            <button onClick = {() => navigate('/article')}> 跳转 </button>
        </div>
    )
}
```

(5) 路由导航传参：

searchParams传参：

```
navigate('/article?id=100&name=jack')

const [params] = useSearchParams()
let id = params.get('id')
```

params传参：

```
navigate('/article/100')

const params = useParams()
let id = params.id
```

注：要在对应router里添加占位符，path:'/article/:id'

(6) 嵌套路由：使用children属性配置路由嵌套关系→使用<Outlet/>组件配置二级路由渲染位置

```
{
    path:'/',
    element:<Layout/>,
    children:[
        {
```

```
            path:'board',
            element:<Board/>,
        },
        {
            path:'about',
            element:<About/>,
        },
    ],
},


const Layout = () => {
    return(
        <div>
            <div>我是Layout</div>
            <Link to = "/board>面板</Link>
            <Link to = "/about>关于</Link>

            {//二级路由出口}
            <Outlet/>
        </div>
    )
}
```

(7) 默认二级路由：在二级路由位置去掉path，添加index属性设置为true
(8) 404路由：准备一个NotFound组件→在路由表数组末尾，以*作为路由path配置路由
(9) 两种路由模式：

路由模式	url	底层原理	是否需要后端支持	创建函数
history	url/login	history对象+pushState事件	需要	createBrowerRouter

路由模式	url	底层原理	是否需要后端支持	创建函数
hash	url/#/login	监听hashChange事件	不需要	createHashRouter

十三、useReducer

（1） 与useState作用类似，用来管理相对复杂的状态数据

（2）基础用法：

定义一个reducer函数（根据不同的action返回不同的新状态）

```
function reducer(state, action){
    //根据不同action type返回新的state
    switch(action.type){
        case 'INC':
            return state + 1
        case 'DEC':
            return state - 1
        //分配action时传参
        case 'SET':
            return action.payload
        default:
            return state
    }
}
```

在组件中调用useReducer，并传入reducer函数和state的初始值

```
const [state, dispatch] = useReducer(reducer, 0)
```

事件发生时，通过dispatch函数分配一个action对象（通知reducer要返回哪个新状态并渲染UI）


```
dispatch({
```

```
    type: 'INC',
                3       //分配action时传参，点击直接变成100
    payload: 100
})
```

十四、useMemo

（1）在组件每次重新渲染时缓存计算的结果
（2）基础语法：组件内有两个依赖项，count1和count2，要求只有count1变化时重新计算

```
useMemo(() => {
    //根据count1返回计算结果
}, [count1])
```

十五、React.memo

（1）允许组件在props没有改变的情况下跳过渲染（React组件默认渲染机制：只要父组件重新渲染，子组件也会跟着重新渲染）
（2）基础语法：

```
const MemoComponent = memo(function ChildComponent(props){
    ...
})
```

（3）props比较机制：使用memo缓存组件后，React会对每一个prop使用Object.js比较新值和老值，返回true表示没有变化
prop是简单类型时：Object.js(3,3) → true 没有变化
prop是引用类型（对象、数组）时：Object.js([],[]) → false 有变化，只关心引用是否变化

十六、useCallback

（1）在组件多次重新渲染时缓存函数
（2）基础语法：

```
const 函数名 = useCallback(函数体, [])
```

十七、React.forwardRef

（1）使用ref暴露DOM节点给父组件

（2）基本语法：

```
//子组件
const Input = forwardRef((props, ref) => {
    return <input type = "text" ref = {ref}/>
})

//父组件
function App(){
    const inputRef = useRef(null)
    return(
        <>
            <Input ref = {inputRef}/>
        </>
    )
}
```

十八、useInperativeHandle

（1）通过ref暴露子组件中的方法

（2）基础语法：

```
//父组件不变
//子组件
const Input = forwardRef((props, ref) => {
    const inputRef = useRef(null)
    //实现聚焦逻辑函数
    const focusHandler = () =>{
        inputRef.current.focus()
    }
    //暴露函数给父组件调用
    useImperativeHandle(ref, () => {
        return{
            foucusHandler
```

```
        }
    })
    return <input type = "text" ref = {inputRef}/>
})
```

十九、Class API类组件

（1）类组件就是通过JS中的类来组织组件的代码
（2）步骤：通过类属性state定义状态数据→通过setState方法修改状态数据→通过render写UI
（3）生命周期函数：组件从创建到销毁的各个阶段自动执行的函数
componentDidMount：异步数据获取，组件挂载完毕自动执行
componentWillUnmount：清理副作用，组件卸载时自动执行
（4）组件通信：
父传子：通过prop绑定数据
子传父：通过prop绑定父组件中的函数，子组件调用
兄弟通信：状态提升，通过父组件做桥接

二十、zustand

（1）基础用法：

```
import {create} from 'zustand'

//创建store
const useStore = create((set) => {
    return{
        //状态数据
        count: 0,
        //修改状态数据的方法
        inc: () => {
            //语法1：参数是函数，需要用到老数据的场景
            set((state) => ({count: state.count + 1}))
            //语法2：参数直接是一个对象
            set({count: 100})
        }
    }
```

```
})

//绑定store到组件
function App(){
    const {count, inc} = useStore()
    return(
        <>
            <button onClick = {inc}> {count} </button>
        </>
    )
}
```

注：创建store函数参数必须返回一个对象，对象内部编写状态数据和方法；set是用来修改数据的专门方法

(2) 异步支持：在函数中编写异步逻辑，最后调用set方法传入新状态

```
const useStore = create((set) => {
    return{
        //状态数据
        channelList:[],
        //异步方法
        fetchChannelList:async() => {
            const res = await fetch(URL)
            const jsonData = await res.json()
            //调用set方法更新状态
            set({channelList: jsonData.data.channels})
        }
    }
})
```

(3) 切片模式：当单个store比较大时，可以采用切片模式进行模块拆分组合，类似于模块化

```
//创建counter相关切片
const createCounterStore = (set) => {
    return{
        count: 0,
```

```
        setCount: () => {
                        set(state => ({count: state.count + 1}))
        }
    }
}

//创建channel相关切片
const createChannelStore = (set) => {
    return{
        channelList: [],
        fetchGetList: async() => {
            const res = await fetch(URL)
            const jsonData = await res.json()
            set({channelList: jsonData.data.channels})
        }
    }
}

//组合切片
const useStore = create((...a) => ({
    ...createCounterStore(...a),
    ...createChannelStore(...a)
}))
```

案例总结

组件库：Ant Design

（1）删除评论：拿到点击项id，以id为条件对列表进行filter过滤

（2）tab切换高亮：点击谁就把谁的type（唯一标识）记录下来，然后和遍历时每一项type做匹配，谁匹配成功就给谁添加高亮类名

（3）json-server实现数据Mock：安装npm i -D json-server→准备一个json文件→在package.json中添加启动命令（scripts）"server": "json-server ./server/data.json --port 8888"→访问接口进行测试

（4）antD-mobile主题定制：组件color属性值为primary

全局定制：整个应用范围内组件都生效


```
:root:root{
```

```
    --adm-color-primary:#a062d4; 3 }
```

局部定制：只在某些元素内部的组件生效

```
.组件外层元素类名{
    --adm-color-primary:#a062d4;
}
```

(5) 账单数据按月分组：数据二次处理钩子函数useMemo+按月分组逻辑lodash.groupBy
(6) 封装request请求模块：

```
const request = axios.create({
    //1.根域名配置
    baseURL:'',
    //2.超时时间
    timeout:5000
})

//3.请求拦截器
request.interceptors.request.use((config) => {
    //操作config注入token数据
    //1.获取token
    const token = loaclStorage.getItem('token_key')
    if(token){
        //2.按后端格式要求拼接token
        config.headers.Authorization = `Bearer ${token}`
    }
    return config
}, (error) => {
    return Promise.reject(error)
})

//4.响应拦截器
request.interceptors.response.use((response) => {
    //2xx范围内状态码触发
    return response.data
```

```
}, (error) => {
    //超出2xx范围状态码触发
    //监控401，token失效状态码
    if(error.response.status === 401){
        removeToken()  //清除本地
        router.navigate('/login')
        window.loaction.reload()  //强制刷新，解决不跳转问题
    }
    return Promise.reject(error)
})

export {request}
```

(7) redux管理token:

```
const userStore = createSlice({
    name:'user',
    //数据状态
    initialState:{
        token:''
    },
    //同步修改方法
    reducers:{
        setToken(state, action){
            state.token = action.payoad
        }
    }
})

//解构actionCreater函数
const {setToken} = userStore.actions

//获取reducer函数
cosnt userReducer = userStore.reducer

//异步方法，完成登录获取token
```

```
const fetchLogin = (loginForm) => {
    return async(dispatch) => {
        //发送异步请求
        const res = await request.post('url', loginForm)
        //提交同步action进行token存入
        dispatch(setToken(rs.data.token))
    }
}

export {setToken, fetchLogin}
export default userReducer
```

(8) token持久化：

问题：redux存入token后如果刷新浏览器，token会丢失

原因：redux是基于浏览器内存的存储方式，刷新时状态恢复为初始值initialState:{token:''}

解决：redux+localStorage获取并存token→loaclStorage？loaclStorage：空字符串

(9) 根据token控制路由跳转：

```
export function AuthRoute({children}){
    const token = localStorage.getItem('token_key')
    if(token){
        //有token，正常返回路由组件
        return <>{children}</>
    }else{
        //无token，强制跳回登录页
        return <Navigate to = {'/login'} replace/>
    }
}
```

(10) 图表渲染：Echarts

(11) 富文本编辑器：npm i react-quill@版本