# react

# 脚手架创建项目

$ npx create-react-app 项目文件夹名

$ npx create-react-app 项目文件夹名 --template typescript

$yarn add react-router-dom axios

$yarn add redux react-redux redux-devtools-extension redux-thunk

$ yarn add sass -D

$yarn add customize-cra react-app-rewired babel-plugin-import -D //按需加载

$yarn add react-quill

$yarn add formik yup

$yarn add lodash

$yarn add postcss-px-to-viewport -D

$yarn add amfe-flexible+postcss-pxtorem -D

$yarn add js-cookie

$yarn add socket.io-client // 长连接

$yarn add dompurify // 防御xss攻击

$yarn add highlight.js // 高亮

$yarn add dayjs

$yarn add react-virtualized //虚拟列表

# react特点

1.声明式UI

2.组件化

3.学习一次,随处使用 native开发移动端原生应用

虚拟DOM可以摆脱浏览器的束缚 通过虚拟DOM可以跨平台开发

开发高效 : 组件化,可复用

性能高效 : 虚拟dom+diff算法,最小化页面重绘

# JSX

JSX是JavaScript XML的简写，表示了在Javascript代码中写XML(HTML)格式的代码

JSX 不是标准的 JS 语法，是 JS 的语法扩展。脚手架中内置的 <@babel/plugin-transform-react-jsx> 包，用来解析该语法。 JSX 创建虚拟DOM更方便 本质是React.createElement()方法的语法糖

优势：声明式语法更加直观，与HTML结构相同，降低学习成本，提高开发效率。

createElement缺点:

- 繁琐不简洁

- 不直观，无法一眼看出所描述的结构

- 不优雅，开发体验不好

语法规则:

// 1. 定义虚拟DOM不要用引号

// 2. js表达式用花括号包裹 表达式? 有输出结果 最简单的判断条件 能被console.log打印的都是表达式

// 3. class类名要替换成className 因为class是es6 关键字

// 4. style内联样式 要写成style = {{key:value}}形式 短横线的要转成小驼峰形式 fontSize

// 5. 只能有一个根标签 <></> 或 <React-Fragment></React-Fragment>

// 6. 标签必须闭合

// 7. 标签首字母,

// 7.1 小写开头,则将该标签转为html同名元素 ,html无对应标签会报错

// 7.1 大写开头,react会去渲染对应的组件,组件没有定义则报错

// 9. for属性要用 htmlFor 如: <label htmlFor="box"></label>

// 10. 多行JSX 用()包裹 , 防止自动加;的bug

# 类组件

// 要继承React.Component父类

// 要有render方法 并且有返回值 表示该组件的结构

// 1. 创建类组件

class Teach extends React.Component {

// 2.render()上面定义state

state={

....

}

// 3.定义事件函数

render() {

// 4.return上面定义数据

....

const str = 'jxk'

// 1.定义jsx

return <div>类组件</div>

}

}

## 组件状态

状态即组件的私有数据,数据变了会自动同步到视图 , 这样我们只要操作数据即可,不用操作dom

# 受控组件

常见的受控组件

文本框、文本域、下拉框（操作value属性）

复选框（操作checked属性）

使用步骤:

1. 在state中添加一个状态，作为表单元素的value值（控制表单元素的值）

2. 给表单元素添加change事件，设置state的值为表单元素的值（控制值的变化）

多表单元素:

+ 给表单元素添加name属性，名称与state属性名相同

+ 根据表单元素类型获取对应的值

+ 在事件处理程序中通过`[name]`修改对应的state

# **非受控组件**

// 非受控组件借助ref, 可以获取到原生DOM或者组件中的属性和方法

textRef = React.createRef()

<input type="text" ref={this.textRef} />

<button onClick={this.Fn}></button>

Fn=()=>{

// 获取到原生DOM

console.log(this.textRef.current)

}

# **组件通讯**

组件是封闭的,想要组件之间的数据,状态进行传递,就要用到组件通讯

## **父传子**

// 父组件给子组件传递 在父组件的子组件标签中使用属性传递

// 可以给组件传递任意类型的数据 props是只读的，不允许修改props的数据

## **子传父**

思路：利用回调函数，父组件提供回调，子组件调用，将要传递的数据作为回调函数的参数。

1. 父组件提供一个回调函数（用于接收数据）

2. 将该函数作为属性的值，传递给子组件

3. 子组件通过 props 调用回调函数

4. 将子组件的数据作为参数传递给回调函数

## **createContext跨级通讯**

跨级组件通讯createContext

如果组件嵌套较深使用状态提升的方式 一层层组件往下传递比较繁琐

这时可以使用 React. createContext() 创建 Provider（提供数据） 和 Consumer（消费数据） 两个组件。

import React, { Component } from 'react'

import ReactDOM from 'react-dom'

// 1.调用 React. createContext() 创建 Provider（提供数据） 和 Consumer（消费数据） 两个组件。

const { Provider, Consumer } = React.createContext()

class App extends Component {

state = {

color: 'red'

}

render() {

const { color } = this.state

return (

// 2.使用 Provider 组件作为父节点。

// 3.设置 value 属性，表示要传递的数据 传递多个数据用对象表示

<Provider value={color}>

<div>

<h1>App组件</h1>

{/\* 4.调用Consumer组件 组件内用箭头函数形式表示 参数接收Provider中value的数据 \*/}

<Consumer>

{color => (

<div>

<h5 style={{ color }}>Chilr组件</h5>

</div>

)}

</Consumer>

</div>

</Provider>

)

}

}

ReactDOM.render(<App />, document.getElementById('root'))

# **children属性**

children属性：表示该组件的子节点，只要组件有子节点，props就有该属性

children 属性与普通的props一样，值可以是任意值（文本、React元素、组件，甚至是函数）

# props校验/默认值

约束规则

1. 常见类型：array、bool、func、number、object、string

2. React元素类型：element

3. 必填项：isRequired

4. 特定结构的对象：shape({ })

示例:

import { Component } from 'react'

import ReactDOM from 'react-dom'

// 1. 引入

import PropTypes from 'prop-types'

class App extends Component {

// 状态

state = {

list: [{ name: 'hah ' }],

car: {

name: 'name',

type: 1

}

}

render() {

const { list, car } = this.state

return (

<div>

<Son list={list} car={car}></Son>

</div>

)

}

}

class Son extends Component {

// 给Son组件增加校验

// static要用babel转义才能用 脚手架已安装不需要额外配置

static propTypes = {

list: PropTypes.array,

car: PropTypes.shape({

name: PropTypes.string.isRequired,

type: PropTypes.number.isRequired

})

}

// 设置默认值

static defaultProps = {

list: [{ name: '哈哈' }],

car: {

name: '奔驰',

type: 3000

}

}

render() {

const { list, car } = this.props

return (

<div>

<ul>

{list.map((item, index) => (

<li key={index}>{item.name + car.name + car.type}</li>

))}

</ul>

</div>

)

}

}

ReactDOM.render(<App />, document.getElementById('root'))

# 生命周期

constructor // 创建组件时最先执行

作用: 1.初始化state 2.创建Ref

render // 每次组件渲染都会触发

作用: 渲染UI

componentDidMount // 组件挂载后

作用: 1.发送网络请求 2.DOM操作

componentDidUpdate // 组件更新后

作用: DOM操作,可以获取到更新后的DOM

挂载阶段

执行时机: 组件创建时 (页面加载时)

执行顺序: constructor() => render() => componentDidMount()

更新阶段

执行时机: 1. setState() 2. forceUpdate() //强制刷新 3. 组件接收到新的props

执行顺序: render() => componentDidUpdate()

卸载阶段

componentWillUnmount()

# **setState**

// 第一个参数是对象 或者函数 第二个参数是回调函数

// 第一个参数

// 如果是对象,会将多个对象进行合并

this.setState({ count: this.state.count + 1 })

// 第一个参数

// 如果是函数,会依次执行 如果后面的setState需要依赖前面的setState 就要用函数形式

// 第二个参数是回调函数,因为setState是异步的 在setState第二个参数中可以获取修改过的值

# 避免不必要的渲染

## **shouldComponentUpdate**

// 状态变化时(render之前)触发

shouldComponentUpdate(nextProps, nextState) {

if (this.state.current === nextState.current) {

// 状态没有发生改变 return false 就不触发render()

return false

} else {

// return true 触发render()

return true

}

}

## **PureComponent 纯组件**

import React, { PureComponent } from 'react'

import ReactDOM from 'react-dom'

// state 和props的值(地址)没有变化 就不会重新渲染

class App extends PureComponent {

render() {

return <div></div>

}

}

// 渲染组件

ReactDOM.render(<App />, document.getElementById('root'))

# 路由

// 底层Router

import { createHashRouter } from 'history'

HashRouter = <Router history={createHashRouter()} />

// 1. 安装依赖包

yarn add react-router-dom

// 2. 导入核心组件

import { HashRouter as Router, Link, Route,Switch,Redirect } from 'react-router-dom'

// Router

1. HashRouter或者HashRouter包裹整个应用，一个项目中只会有一个Router

1.1 HashRouter 使用 URL 的哈希值实现 原理:监听window的hashchange事件

1.2 BrowserRouter 使用 H5 的 history API 实现 原理:监听window的popstate事件来实现

// Link

2.1. Link 指定导航链接替换成a标签

2.2 NavLink

`属性`:

to : 要跳转的url

activeClass : 用于指定高亮的类名，默认`active` // NavLink才有此属性

exact : 精确匹配，表示必须精确匹配类名才生效 // NavLink才有此属性 路径配置了/ ,都需要配置此属性

// Route

3. Route 指定路由规则

`属性`:

path : 匹配路径 // 没有指定 path，那么component对应组件一定会被渲染

component : 匹配路径渲染的组件

// Switch

+ 通常，我们会把`Route`包裹在一个`Switch`组件中

+ 在`Switch`组件中，不管有多少个路由规则匹配到了，都只会渲染第一个匹配的组件

+ 通过`Switch`组件非常容易的就能实现404错误页面的提示

// Redirect

`属性`:

from: 要去的路径

to: 需要重定向的路径 (可以是对象形式 {pathname: ‘ ’, state:{})

exact : 精确匹配

## **路由执行过程**

1. 浏览器url 发生改变

2. React路由监听到url变化

3. react路由内部遍历所有Route组件, puth 与 pathname(hash) 进行匹配

4. 能够匹配就展示该Route组件的内容

## **编程式导航**

// history

通过this.props.history 获取浏览器历史记录的相关信息

this.props.history.push(path):跳转到某个页面 参数path表示要跳转的路径

this.props.history.go(n) : 前进或后退到某个页面 参数n表示前进或后退页面数量

// match

<Route exact path="/user/:id" component={Users}>

通过this.props.match.params.id 获取动态路由参数

// to传的是对象

<Redirect to={{pathname:"/login",search="?id=123",state:{from:Props.loaction.pathname } }} />

通过this.props.loaction 可以拿到search和state

# 样式冲突

推荐使用：CSS Modules （React脚手架已集成，可直接使用）

// CSS Modules会帮我们自动生成的类名，我们只需要提供 classname 即可

生成类名的格式为 [filename]\_[classname]\_[hash]

// 1. 在组件中创建的样式文件名称： 假设此文件为.test类名设置了样式

index.module.css

// 2. 在组件用导入js的形式导入样式文件：

import styles from './index.module.css'

// 3. 通过 styles 对象访问对象中的样式名来设置样式

<div className={styles.test}></div>

// 注意点:

// 1 .user-list的短横线样式 使用 => styles['user-list'] 最好用驼峰 userList命名

// 2 因为CSS Modules会把类名给改掉 如果对字体图标 iconfont 设置类名,字体图标不会生效

// :global(类名) 被:global包裹的类名不会被修改 为全局类名

.aa :global(.iconfont) { color:red }

# **组件复用的技术**

## **mixins(已废弃)**

缺点:

1. 数据来源不明确,不容易维护

2. 方法名冲突

3. 滚雪球般的复杂性

## **render-props**

基本使用:

1. 将公用的状态和方法封装到一个组件中

2. 在组件标签中通过props传递一个回调函数 函数的返回值就是需要渲染的ui ,回调函数的参数就是组件提供的状态

3. 组件的render中调用父组件的props,通常将这个prop命名为render,状态以render参数的形式传递出去

render(){ return this.props.render(this.state)}调用父组件的render属性

示例:

class Mouse extends React.Component {

// … 省略state和操作state的方法

render() {

return this.props.render(this.state)

}

}

<Mouse render={(mouse) => (

<p>鼠标当前位置 {mouse.x}，{mouse.y}</p>

)}/>

//--------------------children代替props------------------------

class Mouse extends React.Component {

// … 省略state和操作state的方法

render() {

return this.props.children(this.state)

}

}

<Mouse>

{(mouse) => (

<p>鼠标当前位置 {mouse.x}，{mouse.y}</p>

)}

<Mouse/>

## **HOC**

一个函数，接收要包装的组件，返回增强后的组件

使用规则

1. 函数名称约定以with开头

2. 参数(传入的组件)只渲染基本的UI

3. 函数内部创建一个类组件,自身提供状态和逻辑, 并将这个类组件的状态传给ui

4.将这个类组件return

// 高阶组件内部创建的类组件：

const WithMouse = (Base) => {

class Mouse extends React.Component {

// 处理鼠标的位置等操作

render() {

return <Base {...this.state} {...this.props} />

}

}

return Mouse

}

// 创建组件

const MousePosition = withMouse(Position)

// 渲染组件

<MousePosition />

# **Hooks**

开发模式

16.8之后 Hooks (提供状态) + 函数组件 (展示内容)

以下问题 函数组件+hooks都可以解决

类组件存在的问题

1. 函数组件和类组件的选择问题

2. 要关注注册事件回调函数的this指向问题以及class的this怎么工作的

3. 代码要拆分到不同的生命周期钩子中,不便于维护

4. 相比于函数组件,不利于代码压缩和优化(tree shaking 树摇),也不利于TS的类型推导

render-props和HOC存在的问题

JSX嵌套地狱问题

重新组织组件结构

`hooks优势`

1. 复用组件状态逻辑,无需改变组件层次结构

2. 根据功能而不是基于生命周期进行代码分割

3. 更好的类型推导

4. tree--shaking 友好,打包会去掉未引用的代码

5. 更好的压缩

// tree shaking 树摇 (类组件使用不了)

代码优化 把没用到的方法和变量剔除掉

## **useState**

// useState不能在if/for/swich/函数中使用, 从开发者工具可以看出 useState在Hook里是按顺序查找的

// useState初始值如果要通过复杂的计算获取,可以使用回调函数 return 的形式,只在组件创建加载

## **useEffect**

// 渲染和更新都会触发回调 相当于componentDidMount + 相当于componentDidUpdate

// 处理函数组件除渲染之外的副作用 例如 Ajax , dom操作

// useEffect本身不能使用async修饰 因为返回值是一个函数 async返回的是promise

减少不必要的diff 组件销毁后 就不要执行修改状态的操作了 // 出现场景 在服务器发起请求时还未响应就切换组件

useEffect中定义变量a控制组件的状态 let a = false

在 修改状态上面 if (a) return

useEffect的return里 a = true

// 参数1: 回调函数

// 参数2: 空数组,代表只有初次渲染时触发, 定时器,Ajax (componentDidMount )

// 参数2: 数组指定依赖项,只会初次渲染和依赖状态值改变时触发 (componentDidMount + componentDidUpdate)

// 回调里用到了依赖 就在参数2中添加此依赖

// 参数1的回调中return一个函数称为清理副作用的函数 此函数在组件销毁的时候执行 ( componentWillUnmount )

## **useRef**

## **useContext**

// 数据提供方

export const Context = React.createContext()

const App = () => {

const [color, setColor] = useState('red')

return (

<Context.Provider value={color}>

....

</Context.Provider>

)

}

// 数据接收方

import { Context } from '../App.js'

import { useContext } from 'react'

const Child = () => {

const color = useContext(Context)

return (

<div>

<h5>我是子组件--{color}</h5>

</div>

)

}

## **React.memo()**

减少不必要的渲染 memo是在DOM更新前触发的

// 相当于类组件的PureComponent 或者shouldComponentUpdate

import { memo } from 'react'

function Child (){

return <div></div>

}

export default memo(Child)

## **useCallback**

缓存函数 配合memo (解决每次组件渲染 传入函数的地址都会改变 导致memo不生效)

用法 useCallback ( 缓存函数 , [] )

const changeName = useCallback(newName => setName(newName), [])

<ChildComp name={name} onClick={changeName} />

## **useMemo**

缓存任意类型的数据

如果是一个昂贵的计算, 为了避免每次状态改变导致每次渲染执行 就可以使用useMemo

只在依赖状态改变的时候在执行 类似于vue的计算属性

import { useMemo } from 'react'

useMemo(() => {return 要缓存的数据}, deps)

# redux

redux 主张单向数据流，不能直接更改 store，即需要通过 disptch(action) -> reducer -> store 的方式更改 store

src/store/index.js

import { createStore, applyMiddleware } from 'redux'

import reducers from './reducers'

import thunk from 'redux-thunk'

import { composeWithDevTools } from 'redux-devtools-extension'

import { getTokenInfo } from '@/utils/storage.js'

const store = createStore(

  reducers,

  { login: getTokenInfo() },

  composeWithDevTools(applyMiddleware(thunk))

)

export default store

src/index.js

import ReactDom from 'react-dom'

import App from './App'

// 导入全局样式

import '@scss/index.scss'

// store

import store from '@/store'

import { Provider } from 'react-redux'

ReactDom.render(

  <Provider store={store}>

    <App />

  </Provider>,

  document.getElementById('root')

)

src/store/reducer/index.js

import { combineReducers } from 'redux'

import login from './login'

import profile from './profile'

import home from './home.js'

const reducers = combineReducers({

  login,

  profile,

  home

})

export default reducers

src/store/reducer/login.js

const initValue = {

  token: '',

  refresh\_token: ''

}

export default function reducer(state = initValue, action) {

  if (action.type === 'login/token') {

    return action.payload

  }

  return state

}

src/store/actions/login.js

同步

export const saveToken = payload => {

  return {

    type: 'login/token',

    payload

  }

}

异步

export const login = (data: { mobile: string; code: string }) => {

  return async (dispatch: Dispatch) => {

    const res = await request({

      method: 'post',

      url: '/authorizations',

      data

    })

    // 保存token到redux中

    dispatch(saveToken(res.data))

    // 保存到本地

    setTokenInfo(res.data)

  }

}

组件中使用

import {add} from './store/action.js'

import { useSelector,useDispatch } from 'react-redux'

// 取数据

const name = useSelector((state)=>state.name)

// 改数据

const dispatch = useDispatch()

dispatch(add(100))