# React

ES6 class回顾

1.当出现extend继承且子类里有constructor函数，必须调用super()

1. constructor中的this指向类的实例
2. Class中的方法保存在class类的prototype上

## constructor supe(props) 的用法

通常是用在要自定义state或者绑定函数this时使用，构造器是否需要接受Props，super传入props，取决于是否希望在构造器中使用this.props,不然会报错undefined

## 给组件自身添加一些元素，可以用 static name =1 的形式

export default class Child extends Component {

    // 直接赋值的情况，等于默认给Home组件实例默认state赋值了一个对象

    constructor(props) {

        super(props)

    }

    // 组件props进行类型限制

    static propTypes = {

        reduceArr: PropTypes.array,

        reduceArr: PropTypes.array.isRequired

    }

    static defaultProps = {

        reduceArr: [1, 2, 3, 4]

    }

    render() {

        const { reduceArr } = this.props

        console.log(reduceArr)

        let ctx = reduceArr.reduce((preValue, currentValue) => {

            return preValue + currentValue

        })

        let arr = {

            name: "1", age: 20, obj: {

                gender: "girl"

            }

        }

        let arr2 = { ...arr }

        arr2.name = "2"

        arr2.obj.gender = "boy"

        console.log(arr)

        console.log(arr2)

        return (

            <div>

                {ctx}

                {/\* {name} \*/}

            </div>

        )

    }

}

## React ref的使用

字符串类型的ref，存在效率问题。目前通过createRef()来实现。

1. 给组件节点添加ref=”xxx“,通过this.refs.xxx获取
2. 通过回调函数 <input type="text" ref={(input)=>{this.textInput=input}}

注：回调形式render再次更新时会调用两次。第一次为null，第二次为dom节点（可以改用事件绑定的形式，解决这个两次的问题）

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

1. 通过createRef()函数调用

  constructor(props) {

        super(props)

        this.input1 = createRef()

        this.state = {}

}

<input type="text" ref={this.input1} />

## 受控和非受控组件

1. 受控组件则是将value值维护到state中，从state中获取最新的value值
2. 非受控组件通过ref来实现现用现取

## 高阶函数和函数柯里化

1如果一个函数接受的参数一个函数如promise，settimeout，arr.map等或者调用的返回值依然是一个函数，就是高阶函数

 <input onChange={this.saveFormdata("username")} type="text" name="username" />

这种写法是将saveFormData的函数的返回值传给onChange

saveFormdata(type) {

        console.log(type) // 此处为传递的参数

        return (event) => { // 此回调

            console.group(type, event.target.value)

        }

    }

// 不用柯里化的内联写法。一般用于在函数调用时二次传自定义参数用

<input onChange={(event)=>{this.saveFormdata("username",event)}} type="text" name="username" />

1. 柯里化：通过函数调用继续返回函数的形式，实现多次接受参数最后统一处理的过程。

Function sum(A){

return b=>{

return c=>{

return a+b+c

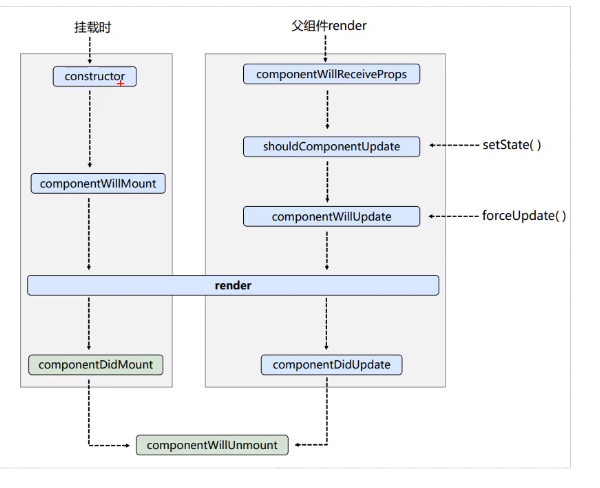
}

}

}

sum(a)(b)(c)

# 生命周期



1. 总体分为初始化阶段，更新阶段，卸载阶段。 初始化阶段中，由Reactdom.render触发，首先执行构造器，接着componentwillMount render didmount

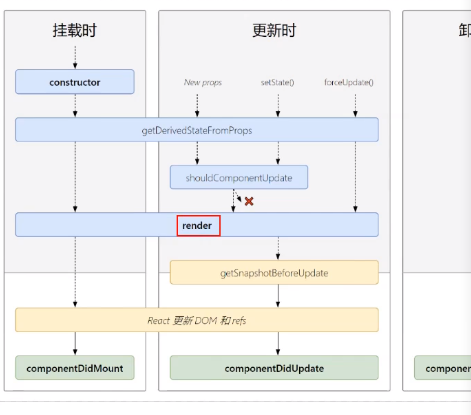
2 . 更新阶段由setState或父组件render触发。 当props修改，componentwillreceiveprops执行，当 shouldcomponentupdate当为true时，执行componentwillupdate, render didupdate。

3.卸载阶段，执行componentwillunmount hook中执行一些逻辑

4.setState之后，先走是否更新。ForceUpdate不需要，强制更新。 willupdate => render =>didupdate

5. 常用hook函数： 1.componentDidMount 当dom节点加载完成后做一些逻辑，比如ajax请求，初始化操作。 2.componentwillunmount 组件即将销毁时，一般用于清理定时器等操作，收尾操作。

1. 新版本reactjs中，异步渲染的推出， componentwillmount componentwillreceiveprops componentwillupdate需要新家UNSAFE\_前缀，不然会报警告。 新生命周期中即将弃用willmount，willreceiveprops，willupdate这三个周期。



#### 以 getDerivedStateFromProps、getSnapshotBeforeUpdate来替换。

使用getDerivedStateFromProps(nextProps, prevState)的原因：

旧的React中componentWillReceiveProps方法是用来判断前后两个 props 是否相同，如果不同，则将新的 props 更新到相应的 state 上去。在这个过程中我们实际上是可以访问到当前props的，这样我们可能会对this.props做一些奇奇怪怪的操作，很可能会破坏 state 数据的单一数据源，导致组件状态变得不可预测。

而在 getDerivedStateFromProps 中禁止了组件去访问 this.props，强制让开发者去比较 nextProps 与 prevState 中的值，以确保当开发者用到 getDerivedStateFromProps 这个生命周期函数时，就是在根据当前的 props 来更新组件的 state，而不是去访问this.props并做其他一些让组件自身状态变得更加不可预测的事情。

 /\*\*

     \* 当state的值任何时候都取决于props，就可以用这个生命周期。

     \* @param {\*} nextProps  传入的props值

     \* @param {\*} preState   原始的state值

     \* @returns

     \*/

    static getDerivedStateFromProps(nextProps, preState) {

        console.log(nextProps, preState)

        console.log("getDerivedStateFromProps")

        return {}   // null 或 状态对象 => this.state , 若返回的值初始state中存在，则无法再修改state的值UI不更新

    }

使用getSnapshotBeforeUpdate(prevProps, prevState)的原因：

在 React 开启异步渲染模式后，在执行函数时读到的 DOM 元素状态并不总是渲染时相同，这就导致在 componentDidUpdate 中使用 componentWillUpdate 中读取到的 DOM 元素状态是不安全的，因为这时的值很有可能已经失效了。

而getSnapshotBeforeUpdate 会在最终的 render 之前被调用，也就是说在 getSnapshotBeforeUpdate 中读取到的 DOM 元素状态是可以保证与componentDidUpdate 中一致的。

一般用于页面渲染时定位当前聊天位置即滚动条的位置所在这种场景

   // 页面即将输出真实dom时，进行一些操作

    // 组件更新之前来个快照 getSnapshotBeforeUpdate 中读取到的 DOM 元素状态是可以保证与componentDidUpdate 中一致的。

    // 如需要以特殊方式处理滚动位置的聊天线程 应返回 snapshot 的值（或 null）

    getSnapshotBeforeUpdate(prevProps, prevState) {

        console.log("getSnapshotBeforeUpdate")

        return this.list.current.scrollHeight // 此hook返回值都将作为参数传给componentDidupdate

    }

    componentDidUpdate(preProps, preState, value) {

        console.log("componentDidUpdate")

        this.list.current.scrollTop += this.list.current.scrollHeight - value

    }

    componentDidMount() {

        setInterval(() => {

            const { newsArr } = this.state

            const news = '新闻' + (newsArr.length + 1)

            this.setState({

                newsArr: [news, ...newsArr]

            })

        }, 1000);

    }

    render() {

        console.log("render")

        return (

            <div>

                {/\* <h1>

                    当前值：{ }

                </h1>

                <button onClick={this.add}>加1</button>

                <button onClick={this.minus}>减1</button> \*/}

                <ul className="list" ref={this.list}>

                    {this.state.newsArr.map(item => {

                        return <li className="list\_" key={item}>{item}</li>

                    })}

                </ul>

            </div>

        )

    }

1. React组件通信 ： 1.父传子之间通过props。 子组件通过父组件传一个回调的形式实现子传父，

Onmouseover onmouseenter 区别： mouseenter不会冒泡。本质上效果一样。

注：

react事件使用： 1.react事件中如果直接传参，会在渲染时调用，不会再触发点击时调用，如果需要传参，可以通过箭头函数的匿名函数形式间接传参

Item 就是要自定义传入的参数，可传多个，event可以不传，event代表的事默认点击事件的回调。

onChange={(event)=>{this.onChangeBox(item,event)}}

// 参数传递放在匿名函数回调里

onChange={()=>{this.onChangeBox(item)}}

生命周期使用小结。 1.constructor componentdidmount只会执行一次，当子组件state依赖于父组件props值的更新时，可以采用componentwillreceiveProps来实现再次setState的方法或者采用新的生命周期getDerivedStateFromProps(nextProps,prevState)来实现，此方法静态方法，无法使用this ，一般需要加一个if判断，判断nextProps.xxx 与 prevState.xxx的值是否不等，不等的话则进行逻辑修改，赋值，最后retrun一个对象，return的对象就是改掉的state的值，会合并到最后的state里，若相等则不用刷新，return一个null 。 如果不加判断，则每次更新交互都会走到这个钩子，导致数据不对，每次都是上一次父组件里的相同数据。2.使用componentwillReceiveProps(nextProps) 时注意，参数nextProps保存着本次更新的最新的props值，若在此钩子里使用this.props，this.props代表的是上次更新的props值，有时可以通过二者的不同，减少不必要的额外的render渲染，需要注意。

React antD使用踩坑：state的值更新了，input框的value值没有更新。因为我们设置的属性是defaultValue={number}，这种可以实现手动编辑，但是不会随着state值变化而更新，我们将属性更改为value={number}之后，发现又没办法输入了，这个时候需要使用input的onChange方法去更新

<Input value={this.state.textValue} onChange={this.onChange} allowClear onPressEnter={this.getCtxValue} placeholder="输入信息，回车键确认" />

 onChange = ({ target }) => {

        this.setState({

            textValue: target.value

        })

    }

##### 对传参进行限制：

// 引入PropTypes 需要大驼峰。作为对象使用

import PropTypes from 'prop-types'

 static propTypes = {

        updateList: PropTypes.func.isRequired,

        updateCheckedList: PropTypes.func.isRequired,

        todoList: PropTypes.array

 }

# React、ajax代理使用

1. 使用axios发送网络请求。 跨域的本质是因为有ajax引擎把响应拦截。代理是没有ajax引擎，同源策略不限制代理，只是作为一个中转的过程。

**// 注： 配置代理之后，并不代表所有请求都会寻找代理服务器地址，首先会在自身资源是否存在若有，则不去代理服务器寻找，若没找到，则会去代理服务器寻找**

1. 配置代理 1.可以在package.json里配置一个proxy选项

  "proxy":"http://xxxxx"

1. 配置代理 2. **src新建 setupProxy.js**

const proxy = require("http-proxy-middleware")

module.exports = function (app) {

    app.use(

        proxy("/api1", { // 匹配到/api1前缀，触发代理

            target: "http://localhost:5000", // 目标地址

            changeOrigin: true, // 如果设置成true：发送请求头中host会设置成target,不然仍是浏览器host

            pathRewrite: { '^/api1': '' } // 重写请求路径，请求到服务器的地址时把地址的/api1替换成""空字符串，不写会报错

        }),

        proxy("/api2", {

            target: "http://localhost:5001",

            changeOrigin: true, // 如果设置成true：发送请求头中host会设置成target,不然仍是浏览里host

            pathRewrite: { '^/api2': '' }

        })

    )

}

# 消息订阅与发布（兄弟组件传值）pubsub.js (publish subscribe发布订阅)

Pubsub.publish("dingyue1","woshishei") // 发布

// msg=>订阅名 ,data =>订阅消息  订阅

componentDidMount() {

        this.token = PubSub.subscribe("dingyue1", (msg, data) => {

            console.log(msg, data)

        })

    }

 componentWillUnmount() {

        PubSub.unsubscribe(this.token);

取消订阅

  }

# React路由

1. **Spa** ： 整个应用只有一个页面，通过ajax局部刷新UI，点击连接不刷新界面。 =》 **单页面多组件的形式。**
2. **前端路由理解。** 整体就是对httpurl路径的匹配以key，value的形式，/home代表path,即key， value就是要展示的组件，即component。 Key代表path，value代表组件
3. **后端路由理解：** 与前端一致，不过value 是一个逻辑函数，用于处理当前路由的逻辑，比如返回数组。
4. 核心依赖： 依赖的是BOM对象的history , createBrowserHistory()方法生成一个**H5推出的history**对象，history.push().go().replace() replace的区别是不向histroy添加记录，返回上一层是上上页。

**浏览器的history记录是一个栈的结构，先进后出**  createHashHistory() 理论上用的也就是**锚点跳转**

1. **React-router ：分为三个部分给web，native和any三个区域。一般前端web使用的是第一种，也称作 router-router-dom**

**使用总结：**

1. **Link标签做跳转，to参数路径， Route标签是展示组件, 这些标签都要包裹在browserRouter或hashRouter里。一般包裹在app.js中用于管理整个项目的路由。**

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

function App() {

  return (

    <div className="App">

      <Link to="/about">Go 1</Link>

      <br />

      <Link to="/home">Go 2</Link>

      <article>

        <Route path="/about" component={About} />

        <Route path="/home" component={Home} />

      </article>

    </div>

  );

}

Main.js中

ReactDOM.render(

  <React.StrictMode>

    <BrowserRouter>

      <App />

    </BrowserRouter>

  </React.StrictMode>,

  document.getElementById('root')

);

Link 标签没有高亮

可以替换为NavLink标签，进行高亮

Navlink 自带 activeClassName 属性，可以用来设置高亮的class类名

<NavLink activeClassName="active" to="/about">Go 1</NavLink>

<br />

<NavLink activeClassName="active" to="/home">Go 2</NavLink>

1. **路由组件一般要放在工程目录 pages文件夹。一般组件放在components文件夹中。**
2. **封装Navlink组件**

 {/\* 自定义组件标签体内容将通过props.children的形式传给子组件 \*/}

Go 1 在自定义组件中 可以通过 this.props.children的形式获取 也可以通过children的形式传给子组件

<MyNavLink to="/about">Go 1</MyNavLink>

<MyNavLink to="/about" children="Go 1" />

<MyNavLink to="/home">Go 2</MyNavLink>

总结： 不一定非得写闭合标签，如果是自闭和，可以将标签体内容以children的形式传给组件

import React, { Component } from 'react';

import { NavLink } from 'react-router-dom'

class MyNavLink extends Component {

    render() {

        return (

            <div>

                <NavLink className="link" activeClassName="active" {...this.props} />

// react中 {...this.props}的形式代表把对象展开依次传给组件

            </div>

        );

    }

}

export default MyNavLink;

1. **使用Switch ，包裹Switch标签，匹配到路由，则不会继续向下匹配。不用Switch的话则会一直匹配，不管是否匹配到。匹配到几个显示几个。 提高匹配效率**

**题外话：http://localhost:3000 就是wepackdevserver开启的服务，指的就是当前脚手架，项目中public文件夹就是http://localhost:3000（当前脚手架服务器下）的根路径。 如果请求啥也没请求到，会把index.html返回给你。**

1. **一级路由加前缀，在browserRouter下可能会导致样式丢失。如果index.html中的路径是相对路径./xxx/xxx ，如果加了额外匹配。比如项目是http://localhost:3000/api/about , browserRouter会把/api/about路径带到请求路径中去。 解决办法1：index.html换成根路径/xx/xx，或者以./路径的话，可以用hashRouter, hashRouter默认把#后面的内容当成前端资源，不会放入请求路径**
2. **模糊匹配，总体总结：输入path必须包含匹配的path，而且必须在做左侧**

   <nav>

        {/\* 自定义组件标签体内容将通过props.children的形式传给子组件 \*/}

        <MyNavLink to="/about" children="Go 1" />

        <MyNavLink to="/home/a/b">Go 2</MyNavLink>

      </nav>

      <article>

        <Route path="/about" component={About} />

        <Route path="/home" component={Home} />

      </article>

// 此时，路由地址/home/a/b 遵循最左匹配原则，可以匹配到home

 <MyNavLink to="/a/home/b">Go 2</MyNavLink>

// 如果是这种，则匹配不到，因为遵循最左原则，第一个没匹配到，则不匹配

<nav>

        {/\* 自定义组件标签体内容将通过props.children的形式传给子组件 \*/}

        <MyNavLink to="/about" children="Go 1" />

        <MyNavLink to="/home">Go 2</MyNavLink>

      </nav>

      <article>

        <Route path="/about" component={About} />

        <Route path="/home/a/b" component={Home} />

      </article>

、

// 这种情况下无法匹配。模糊匹配的原则是 link的路径必须包含 route的路径

1. **精确匹配 ，必须路径一样，否则匹配不到。**

  <Route path="/about" component={About} exact />

1. **Redirect ：重定向 。 放在Route最后，当所有都没匹配到，则走这个Redirect，或者对已存在的path进行重定向**
2. **嵌套路由，路由注册顺序有先后关系，首先从先注册的进行匹配。当有嵌套路由，嵌套路由地址path需包含父级路由地址如 /home/message 。 当路径是 /home/message，首先从先注册的路由匹配到/home，在进入home页，继续匹配到/home/message 。 注意：存在嵌套路由时，父级路由不能使用exact精确匹配，否则匹配不到。模糊匹配的原则事匹配的link的path要包含route的path**

import React, { Component } from 'react';

import News from './news'

import MessageBox from './message'

import MyNavLink from '../MyNavLink'

import { Link, NavLink, BrowserRouter, Switch, Route, Redirect } from 'react-router-dom'

class Home extends Component {

    render() {

        return (

            <div>

                我这里是Home组件

                ===============

                <MyNavLink to="/home/news" children="Go 1" />

                <MyNavLink to="/home/message">Go 2</MyNavLink>

            ------------   ------------   ------------   ------------

                <div>

                    <Switch>

                        <Route path="/home/news" component={News} />

                        <Route path="/home/message" component={MessageBox} />

                        <Redirect to="/home/news"></Redirect>

                    </Switch>

                </div>

            </div>

        );

    }

}

export default Home;

1. **路由传参**
2. **state传参，不体现在url上，刷新可记录。如果是browserRouter的情况下。因为一直在操作浏览器的history对象。 route不需要配置接受。直接通过this.props.location.state获取**

 <MyNavLink to={{ pathname: "/home/message", state: { id: 1 } }}>message</MyNavLink>

1. **Param传参，以/ xx /xx的形式拼接在url上，刷新可保存，需要通过Route匹配。并通过this.props.match.param来获取**

 <MyNavLink to="/home/message/01">message</MyNavLink>

 <Route path="/home/message/:id" component={MessageBox} />

1. **Search传参，已 ?id=xxx&name=xxx的形式传递，Route不需要引用匹配。直接在页面中this.props.location.search获取，获取到的是urlencoded形式的字符串，需要引用qs包。querystring包。来进行转换**
2. **withRouter使用，普通组件也可以使用路由组件的一些api，获取到其this.props.history等方法，可以进行外部跳转**

import React, { Component } from 'react';

import {withRouter} from 'react-router-dom'

class HeaderBox extends Component {

    getChange = () => {

    }

    render() {

        return (

            <div>

                我这里是header组件

                <div>

                    <button onClick={this.getChange}>获取信息</button>

                </div>

            </div>

        );

    }

}

export default withRouter(HeaderBox);

1. **browserRouter与hashRouter的区别**
2. **browserRouter底层原理用的H5 history API，不兼容IE9以下，hashRouter使用的url的哈希值**
3. **Url体现不同，browser没有#，hash有#,并且hash#后面的默认为前端资源，不发送给服务器**
4. **刷新后对路由参数state的影响，brwoser不会丢失，保存在history上，但hashRouter刷新后会导致state参数的丢失**
5. **AntD样式按需引入，和自定主题，具体看官网。 Vue.config.js = config-overrides.js , 在这里可以配置一些wepback的一些配置，比如antD的样式按需引入和修改主题等操作。**

const { override, fixBabelImports, addLessLoader } = require('customize-cra');

module.exports = override(

    // antd style按需引入

    fixBabelImports('import', {

        libraryName: 'antd',

        libraryDirectory: 'es',

        style: 'css',

    }),

    // 自定义主题

    addLessLoader({

        lessOptions: {

            javascriptEnabled: true,

            modifyVars: { '@primary-color': '#1DA57A' },

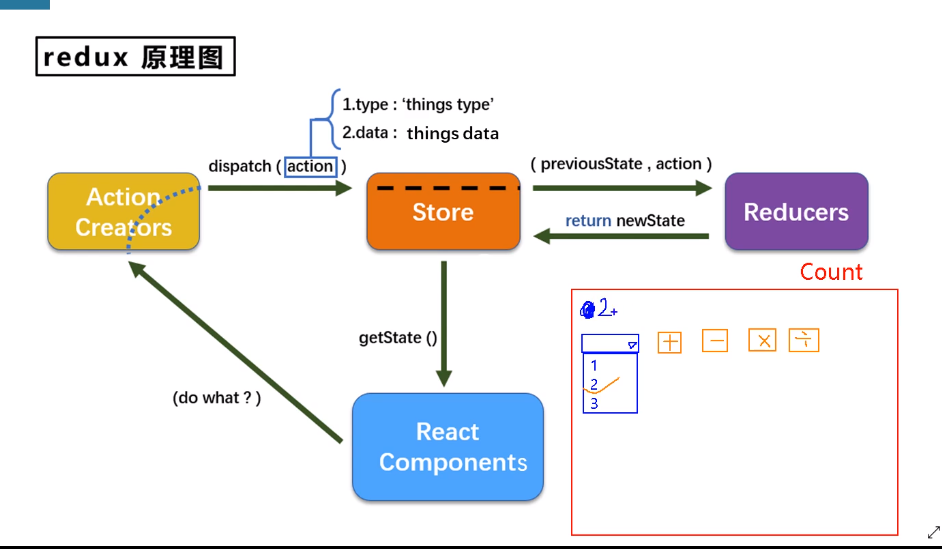
        }

    }),

);

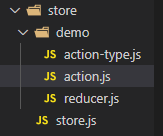
# Reudx

1. Redux，一个数据集中管里的js库，基本与react一起使用。使用场景：1.共享某个组件的状态，共享公共组件的state信息，整体共享。2.组件改变另一个组件的状态。原理类似Vuex store



1. Action Store Reducers Component之间的关系：共享state要放到Store里。 Component发出交互即提交 dispatch action {type:”xx”,data:xxx} 给 store， store收集分发给给单个组件模块服务的reducer，reducers可能有多个为不同的组件服务等。Reducer收集到action，根据actiontype来处理逻辑，返回新的state给store， component组件从state获取最新的state。 Store.getState(),通过store.subscribe()来监听store state对象的变化，刷新UI。初始化时工作在Reducer中。

3. 实际开发中，一般需要创建一个主store, action, actionType,reducers四个文件。Store时共有的。每个模块一个文件夹里面有三个action, actionType,reducers，action是页面提交的action的函数内容 , actionType 是用来type用的定义的常量，reducers是模块的加工处理。



action.js

import { INCREMENT } from './count\_actionType'

export const createIncreament = data => {

    return {

        type: INCREMENT,

        data

    }

}

actionType.js

export const INCREMENT = 'INCREMENT'

Reducer.js

/\*\*

 \* 创建一个为组件服务的reducer，本质是一个函数

 \*/

import { INCREMENT } from './count\_actionType'

const initState = 0 // 初始化状态

export default function countReducer(prevState = initState, action) {

    // 初始化时,prevState = undefined, action = @@init@@

    const { type, data } = action

    // 根据type来处理逻辑

    console.log(type)

    switch (type) {

        case INCREMENT:

            return prevState + data

        case 'DECREMENT':

            return prevState - data

        default:  // 初始化在这里

            return prevState

    }

}

1. 异步和同步Action，如果action的是普通obj则为同步，如果是一个函数，则为异步Action

想使用异步Action，需要使用中间件middleWare，引用applyMiddleWare,同时引用redux-thunk。不然会报错 。 若存在多个reducers, 通过combineReducers来实现

combineReducers接受一个对象 combineReducers ( {

A: reducer1,

B: reducer2

} )

组件中映射时,以此处的名字为参数 , state => state.A.count ....

import { createStore, combineReducers, applyMiddleware } from 'redux';

import \* as demo from './demo/reducer';

import thunk from 'redux-thunk'

let store = createStore

// 合并reducers,可能有多个reducer,用过import \* as , 最后通过 {...}来

展开合并

    combineReducers({ ...demo }),

    applyMiddleware(thunk)

);

export default store;

import \* as Type from './action-type'

import { get } from 'axios'

// 同步action

export const savaForm = (payload) => {

    return {

        type: Type.SAVEFORM,

        payload

    }

}

// 异步Action

// 异步Action中一般都会调用同步Action

export const asyncMethod = () => {

    return async dispatch => {

        try {

// 可以在这里调用IO口获取数据

            let { data } = await get("http://jsonplaceholder.typicode.com/posts")

            console.log(data)

            dispatch({

                type: Type.GETLIST,

                payload: data

            })

        } catch (error) {

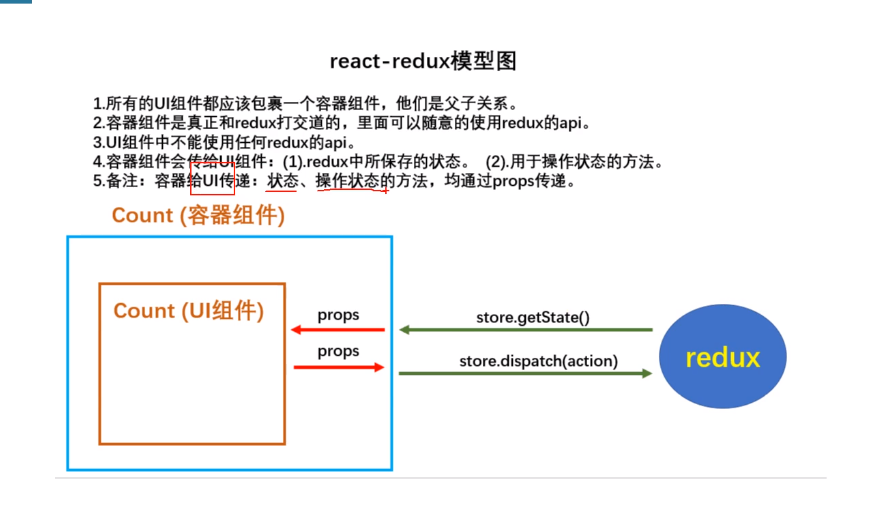
            console.error(error);

        }

    }

}

1. 使用 react-redux ，需要在所有UI组件外包一个容器组件，所有的状态操作都是容器组件和redux来操作。UI组件只通过props来操作方法返回给容器，在返回给Redux



Export default connect(mapStateToProps,mapDispatchToProps)(UI) , mapStateToProps返回一个对象，从redux映射来的state, mapDispatchToProps返回一个对象，从redux映射来的action操作state的方法。



用provider 为项目所有需要的地方，注入store依赖

注!: react,redux中很少使用push,unshift等方法,此与Vue不同, redux中reducers必须是一个纯函数,如果写成数组等push等方法,最后返回这个数组,此时数组确实变化,但最后react更新时进行浅对比发现返回但和之前但数组但引用地址一致,UI不会更新,必须返回一个新引用地址的数组,

Redux 的 reducer 必须是一个 纯函数

纯函数: 同样的输入,得到同样的输出 a(1) = 1 b(a) = a {

1. 不得改写参数,如果用unshift push则改变了redux的preState
2. 不得写网络请求等
3. 不能调用Date.now() Math.random()等不纯的方法

}

export const changeCount = (prevState = initalState, action) => {

console.log(action)

switch (action.type) {

case ADD\_COUNT:

return { // 纯函数写法

...initalState, count: action.data + prevState.count

};

case MIN\_COUNT: // 这种写法不是纯函数写法,返回的prevState和参数的 prevState是同一个引用地址,并且它改变了prevState数组,React更新时浅对比发现引用地址相同,即使数组更新了,UI界面也不会刷新!!

prevState.push(1)

return prevState

// 可以用这种写法

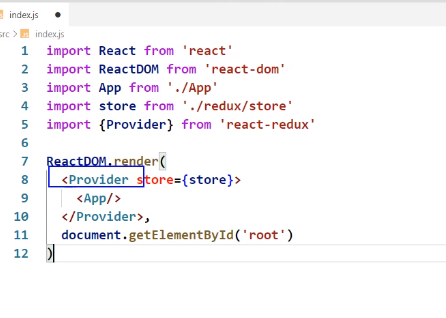
return [data,...prevState] // 返回一个新数组,纯函数写法

default: // 初始化

return initalState

}

}



withRouter 和 connect 混用

// 连接UI组件和容器UI

export default connect(

state => ({

count: state.changeCount.count

}),

{

addCount,

minCount

}

)(withRouter(App))

# 七 打包

打包成功后,可全局安装serve库,来实现文件夹的快速部署服务,静态资源即可查看

# 八 总结扩展

##### setState , 可以分为对象式写法和函数式写法,第二个参数为可选参数,是一个回调函数 , setState为同步,但更新为异步更新,为了合并更多的state更新

this.setState({

count: val + 1

}, () => {

//

})

this.setState((state, props) => {

return {

count: state.count + 1

}

}, () => {

// 回调

})

如果新状态不依赖原状态,直接用对象写法,如果依赖props和原来state,可以用函数式写法,对象式写法就是函数式写法但简写,因为函数式写法最后也要返回一个对象式的对象值

异步的情况：

由React控制内的事件处理函数(onChange、onClick等合成事件)，以及生命周期函数调用setState时表现为异步 。

同步的情况：

React控制之外的事件中调用setState是同步更新的。比如原生js绑定的事件，setTimeout/setInterval，ajax，promise.then内等 React 无法掌控的 API情况下，setState是同步更新state的。

##### 路由懒加载

// 路由懒加载

import React, { Suspense, lazy } from 'react'

// const component1 = lazy(() => import('./FormDemo'))

suspense类似一个过度,当请求懒加载组件过程中展示这个

// < Suspense fallback={ component2 }>

// <Route>

// </Route>

// </Suspense >

##### 16.8 之后 React,新增 Hooks

import React, { useState, useRef, useEffect } from 'react'

// import logo from './logo.svg';

import './App.css';

// import StateComponent from './components/stateComponents'

// import LifeCycle from './components/LifeCycle'

// import SetStateDemo from './components/setStateDemo'

// import FlagRender from './components/FlagRender'

// import MapList from './components/MapList'

// import FormDemo from './components/FormDemo'

import { connect } from 'react-redux'

import { addCount, minCount } from './store/count\_/action'

// class App extends React.Component {

// constructor() {

// super()

// this.state = {

// title: "x"

// }

// }

// changTitile = () => {

// this.setState({

// title: "this is not title"

// })

// }

// callback = msg => {

// console.log(msg)

// }

// changeButton = () => {

// console.log()

// this.props.addCount(1)

// }

// render() {

// console.log(this.props)

// return (

// <div className="App">

// <header className="App-header">

// <img src={logo} className="App-logo" alt="logo" />

// <FormDemo />

// {this.props.count}

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

// 加 1

// </button>

// {/\* <LifeCycle callback={this.callback} title={this.state.title} />

// <button onClick={this.changTitile}>chang titile</button> \*/}

// </header>

// </div>

// )

// }

// }

function App( props ) {

// 所有的类组件的this.props = 此处的 props

// count 接收值, setCount接收改变count的回调函数

const [count, setCount] = React.useState(0) // 返回两个,一个是值0,一个是改变值的方法

const [info, setInfo] = useState([{

id: "001",

name: "paul"

}])

let myref = useRef()

function addCount() {

setCount(count => count + 1)

// setCount(2)

console.log(info)

}

// 使用生命周期

useEffect(() => {

// 此处是哪个周期根据第二个参数来,如果第二个参数不传,则代表监听所有数据的改变 componentDidupdate

// 如果第二个参数传了数组,比如[count],则代表只监听count的变化来更新UI

// 如果第二个参数传了一个[]空数组,则代表不监听任何变化,等同于componentDidMount

console.log(info)

return () => {

// 此处return的回调函数代表 componentwillUnomount

console.log("compp")

}

}, [count, info])

function addList() {

let obj = {

id: "2",

name: "123"

}

setInfo(info => {

return [obj, ...info]

})

}

function changeList() {

console.log(myref.current.value)

}

return (

<div>

我这个是函数时组件,这是{count}

<input type="text" ref={myref} />

<button onClick={addCount}>改变count</button>

<button onClick={addList}>新增info</button>

<button onClick={changeList}>改变info</button>

</div>

)

}

// 连接UI组件和容器UI

export default connect(

state => ({

count: state.changeCount.count

}),

{

addCount,

minCount

}

)(App)

1. Fragment , 虚拟废弃DOM , 由于页面必须有一个根节点,一般都是div,但如果我们不想要渲染这个div,可以用 <Fragment ></Fragment> 包裹,不会被渲染 , 当然也可以用一个 <> </> 空标签渲染 , 区别在于 如果此列表是遍历动态渲染需要绑定key的时候, Fragment 可以写一个 key , 但<>空标签啥也不能给
2. context 一般用于实现祖孙组件传值

import Child from './Child'

// 首先需要顶层创建一个context对象

let MyContext = React.createContext()

export const { Provider, Consumer } = MyContext

// 结构出prover和consemer组件来使用

function App() {

const [userInfo, setUserInfo] = React.useState({

name: "123",

age: 20

})

return (

<div>

我是最大A

// 给后代组件提供时,包裹父组件中的一层子组件,那么后代中所有组件通过注册都可以接收到context值 , 类似 react-redux的顶层provider

<Provider value={userInfo}>

<Bpp />

</Provider>

</div>

)

}

function Bpp() {

return (

<div>

我是第二B

<Child />

</div>

)

}

// function Cpp() {

// return (

// <Consumer>

// {value => {

// console.log(value)

// return (

// < div >

// 我是最小C

// </div>

// )

// }}

// </Consumer >

// )

// }

// Child.jsx

import React from 'react'

import { Consumer } from './App'

function Child() {

return (

// 后代组件需要接收context值时,需要用Consumer来接收

// 此为函数式组件接收写法

<Consumer>

{

value => {

console.log(value)

}

}

</Consumer>

)

}

export default Child

// 如果是类式组件

export default class Child extends React.Component {

// 定义一个静态变量来 接收这个 context顶层对象

通过 this.contextChild 来获取值

static contextChild = MyContext\_

render() {

console.log(this.contextChild, "contextChild")

return (

<div>

我是Child最小的

这是值

</div>

)

}

}

6 pureComponent与Component , PureCompent自动浅对比state与props,来告诉组件是否需要更新,一般用于纯展示组件,component需要实现可以通过 shouldComponentUpdate(nextProps,netxState)来判断两次是否变化,来是否更新

1. renderProps, childrenProps 插槽 , 组件中预留一个位置,随时可以插入任何组件任何元素,可以传入一些组件的参数 这个就是render props插槽

普通组件中比如<Bpp> 123 </Bpp> , 此123不会展示,则会保存在Bpp组件中的this.props.children中,可以通过在Bpp组件中调用this.props.children来展示自定义的123内容,这个就是children插槽

function App() {

const [userInfo, setUserInfo] = React.useState({

name: "123",

age: 20

})

return (

<div>

我是最大A

<Provider value={userInfo}>

// render 返回一个函数 ,此name就是 Bpp组件里要开放的参数

<Bpp render={(name) => <Child name={name}></Child>} />

</Provider>

</div>

)

}

function Bpp(props) {

return (

<div>

我是第二B

// 此处需要调用props.render() 可以传入一些参数给插槽用

{props.render("name")}

</div>

)

}

1. 错误边界,捕捉子组件的错误并控制在子组件范围里,并不影响整个组件树的展示

父组件中使用

static getDerivedStateFromError(error) {

return {}

}

只能捕获后代组件生命周期里出现的错误,会把错误带给父组件 error ,此处可以定义变量,通过变量来控制组件是否展示,如果没错误,则展示组件,如果有错误,则展示404组件 , 此方法用于 生产环境 , 开发环境不使用

componentDidCatch () 用于统计子组件中的页面错误量, 可以发送给后台进行反馈

1. 组件通信总结 : 1.父子 props

2. 兄弟 pub-sub,event等消息订阅发布, 如果有同父,则可以子- 父- 子, redux集中管理

3.祖孙 pubsub消息订阅发布, 或者一层一层传, 或者集中式管理 redux ,亦或者可以用context 来实现,消费者订阅者

Context:开发用的不多, 需要用

Const MyContext = React.createContext

取得其 {Provider,Consumer}来,在第一层子组件用Prover包裹 <Provider name={obj}></Provider> 传入 ,则所有后代自组件 ,如果是 函数式 , 通过引用 Consumer 来包裹

如果是类式组件,需要自定义静态变量 static xxx = myContext , 通过this.xxx获取

<Consumer >

{

// value就是 provider传入的参数

Value =>{

Return ‘’’

}

}

</Consumer >