cm补丁版本

https://192.0.0.110/bigdata/technology/product/bigdata/branches/solutions/hbm/hbm-3.10/ui

项目地址：C:\Users\dongjiacheng\cm-patch

启动项目：先 `npm start` ，再开一个命令行 `npm run build`

服务器：暂时不知道

提交日志规范：`BUG` , `CHG` , `ADD`

cm重构版本

https://192.0.0.110/bigdata/technology/product/bigdata/branches/solutions/serim/1.0.0/serim/ui

项目地址：C:\Users\dongjiacheng\cm-refactor

启动项目：`npm start`

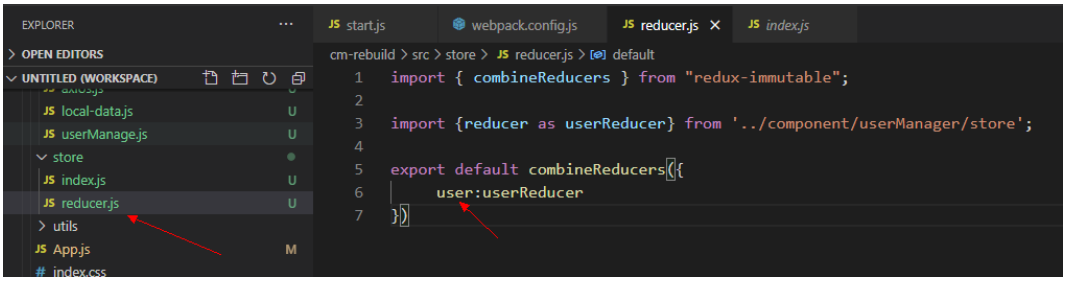
服务器：10.3.71.104:2343，root，hik12345/*（前端文件放在/ycl下）

上去看一下tomcal的sever.xml和web.xml配置，发布就替换一下前端文件

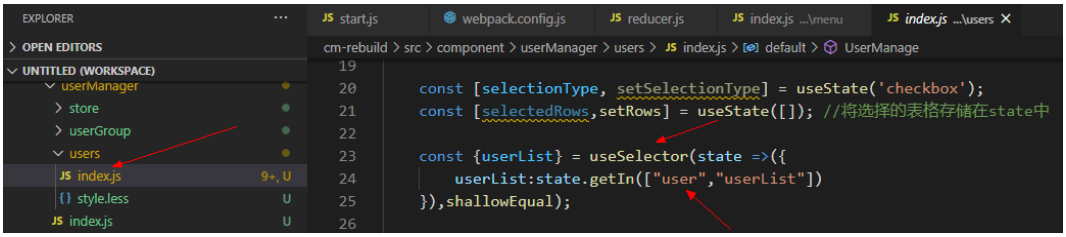
Git 仓库地址：ssh://git@10.3.71.102:2343/home/git/dongjiacheng/cm-rebuild.git

注意事项

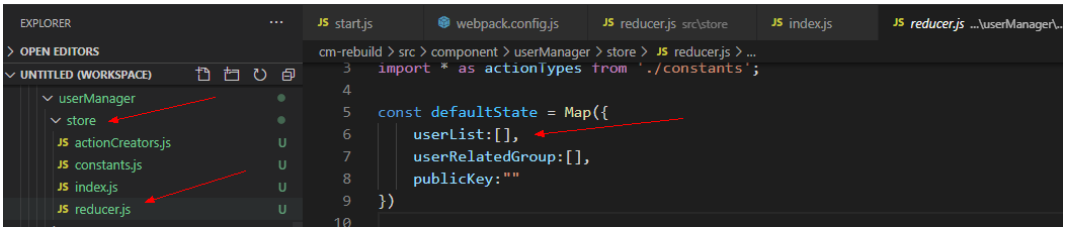
1. react 17 和浏览器调试插件冲突，在启动项目时需要禁用 react-dev-tools
2. 当前版本回退到 react 16，但是redux-persist还是有冲突
    - 不行的话只能手写hooks来存localstorage了
    - 之前的重构版本是16.13，redux-persist没问题，可以使用
3. antd升到高版本后，传入的数据中一定要给item加key，要不复选框会乱掉
4. axios 如果没有指定请求方式，默认请求是 GET
5. redux 使用
    - 这个地方



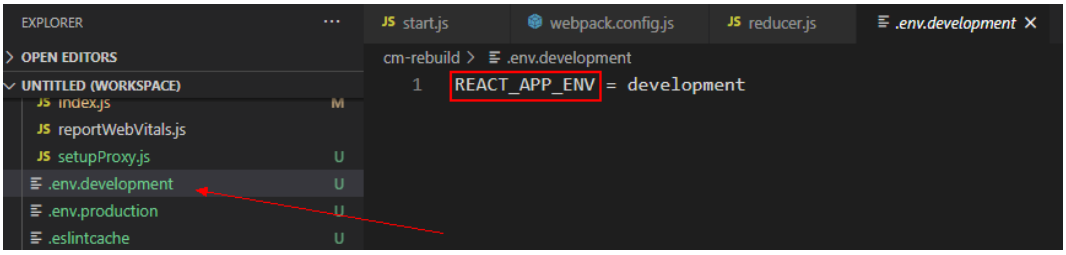
对应的是userReducer



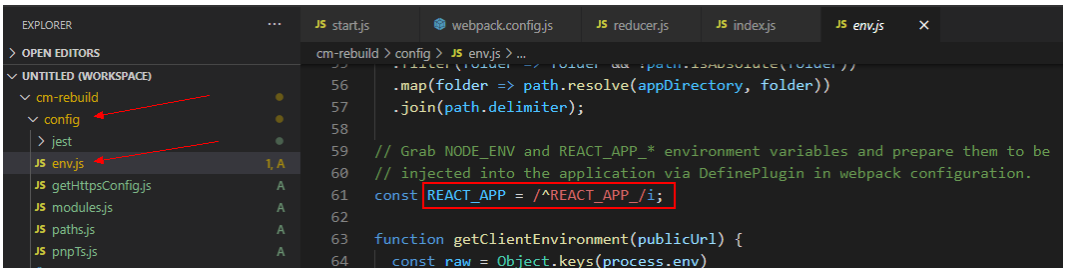
该句的意思就是获取userReducer中的userList值



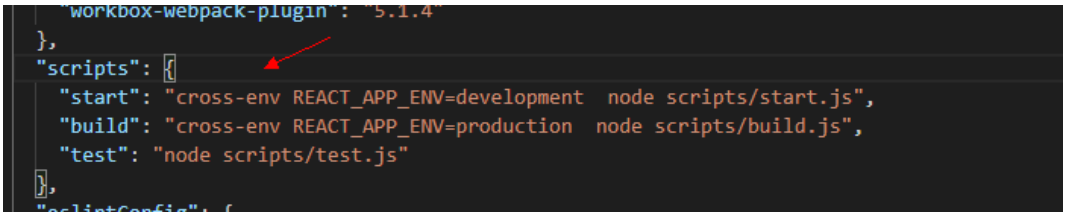
6. build下默认process.env.NODE_ENV是production，start下默认是development
    - 如果在根目录下使用.env，命令要注意是以 REACT_APP 开头



    - 该部分的处理逻辑，可以去config下的env.js中查看



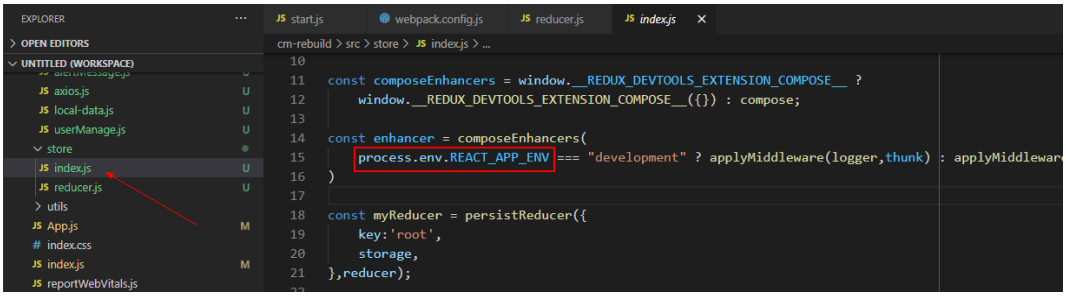
    - 自己定义.env的话，可以比系统自带的development和production多一些
    - 当前的环境，实际上只有两种，开发和生产



引入cross-env，主要是为了让你看下怎么区分环境的

不使用cross-env，也是可以的

这一块可以直接使用process.env.NODE_ENV



7. hooks的使用，也注意下

- 只能在顶层写，不要在条件判断中添加hooks处理

8. npm 执行报错的处理方法
   - npm cache clean -f 命令执行一波再试下
9. 在下面代码中，`()` 和 `{}` 是不一样的。如果使用 `()` 表示箭头函数的函数体只有一行，所以不用 `return` 。如果使用 `{}` 表示箭头函数的函数体有多行，需要使用 `return` ，否则没有 JSX 元素返回。

```
render: () => (
    <button>点我试试</button>
)

render: () => {
    return <button>再点下试试</button>
}
```

10. react-redux 调用关系

    目录结构如下

```
src
|-component
| |-userManage
|   |-store
|   | |-actionCreators.js
|   | |-constant.js
|   | |-index.js
|   | |-reducer.js
|   |-userGroup
|   | |-index.js
|   | |-style.less
|   |-users
|-store
  |-index.js
  |-reducer.js
```

首先在 `userManage` 模块下建一个 `store` ，创建 `index.js` 文件，内容如下：

```
// index.js
import reducer from './reducer';
export {
    reducer
}
```

然后创建 `reducer.js` 文件，内容如下：

```
import {Map} from 'immutable';
import * as actionTypes from './constants';

const defaultState = Map({
    userGroup: [],
    tenantList: []
})

function reducer(state=defaultState, action) {
```

```
    switch(action.type) {
        case actionTypes.SET_USER_GROUP:
            action.payload.forEach(item => item.key = item.cn);
            return state.set("userGroup", action.payload);
        case actionTypes.SET_TENANT_LIST:
            return state.set("tenantList", action.payload);
        default:
            return state;
    }
}

export default reducer;
```

然后创建 `constants.js` 文件，内容如下：

```
export const SET_USER_GROUP = "user/SET_USER_GROUP";
export const SET_TENANT_LIST = "user/SET_TENANT_LIST";
```

然后创建 `actionCreators.js` 文件，内容如下：

```
import * as actionTypes from './constants';
import { getUserGroup, getTenants } from '@/services/userManage';

// 更新用户组
const changeUserGroupAction = (userGroup) => ({
    type: actionTypes.SET_USER_GROUP,
    payload: userGroup
})

// 更新租户
const changeTenantListAction = (tenantList) => ({
    type: actionTypes.SET_TENANT_LIST,
    payload: tenantList
})

// 调接口获取用户组
export const getUserGroupAction = (currentPage, pageSize) => {
    return dispatch => {
        getUserGroup(currentPage, pageSize).then(res => {
            dispatch(changeUserGroupAction(res));
        })
    }
}

// 调接口获取租户
export const getTenantListAction = () => {
    return dispatch => {
        getTenants().then(res => {
            console.log(res);
            dispatch(changeTenantListAction(res));
        })
    }
}
```

接下来在 `userGroup/index.js` 中使用，代码如下：

```
import React, { useEffect } from 'react';
import { useDispatch, useSelector, shallowEqual } from  'react-redux';

export default function UserGroupList() {
    const dispatch = useDispatch();

    const { userGroup, tenantList } = useSelector(state => ({
        userGroup: state.getIn(["user", "userGroup"]),
        tenantList: state.getIn(["user", "tenantList"])
    }), shallowEqual)

    useEffect(() => {
        dispatch(getUserGroupAction());
        dispatch(getTenantListAction());
    }, [])

    return (
        <div>
            { userGroup.map(item => <div key={item}>{item}</div>) }
        </div>
    )
}
```

调用顺序如下：当组件渲染完成，会触发副作用 `useEffect` ，页面组件会分发（dispatch）一个action给到 `actionCreators` 里面的getAction。在getAction里面进行接口调用，然后分发给changeAction。changeAction会把action.type和action.payload传给 `reducer` ，所有状态都保存在这里，然后 `reducer` 匹配相应的action.type进行状态修改。在页面组件中通过 `useSelector` 获取状态，当状态发生变化，这里的值也会跟着改变。这边有一句代码：

```
userGroup: state.getIn(["user", "userGroup"])
```

这里 `"userGroup"` 毫无疑问肯定是在 `reducer` 定义的，那么 `"user"` 是在哪里定义的？在 `src` 目录下的 `reducer.js` 里面，代码如下：

```
import { combineReducers } from "redux-immutable";
import {reducer as userReducer} from '../component/userManager/store';

export default combineReducers({
     user:userReducer
})
```

然后在同级目录下的 `index.js` 文件代码如下：

```
import { createStore, compose, applyMiddleware } from "redux";
import { persistReducer } from 'redux-persist';
import storage from 'redux-persist/lib/storage';
import thunk from "redux-thunk"; // 现在用的是redux-thunk，后面会改成saga
import logger from 'redux-logger';
import reducer from './reducer';

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ?
    window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__({}) : compose;

const enhancer = composeEnhancers(
```

```
    process.env.REACT_APP_ENV === "development" ?
applyMiddleware(logger,thunk) : applyMiddleware(thunk)
)

const myReducer = persistReducer({
    key:'root',
    storage,
},reducer);

const store = createStore(myReducer, enhancer);

export default store;
```

关于接口调用，这边定义在 `actionCreator` 里面，也可以在页面组件的hooks里面调接口，然后把数据dispatch给 `actionCreator` 。

11. 列表渲染需要使用 `key` ，如果是手动循环，`key` 可以加到 JSX 元素上，如果是使用 `<Table>` 组件，`key` 需要加到需要渲染的数据里面
12. 在下面代码中，`useState` 实际上是异步的，调用 `setUserGroupData` 修改 `userGroupData` 之后，并不能立即获取到修改后的值。如果想要实现同步调用，可以使用回调函数、async/await等方式。

```
const [userGroupData, setUserGroupData] = useState({cn: "", description:
""});
```

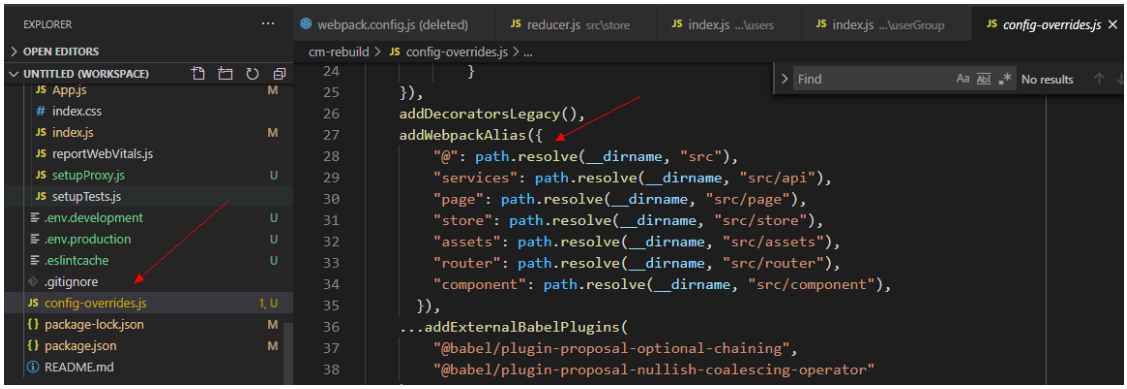
实际测试发现，上面的几种方法好像也没有转成同步，倒是 `Object.assign(userGroupData, newValue)` 以浅拷贝形式可以进行同步调用。

13. 在 `useEffect` 使用了 `dispatch` ，需要把 `dispatch` 也放到依赖项里面

```
useEffect(() => {
    dispatch(getTenantListAction());
}, [dispatch]);
```

> 注意：如果没有依赖项，也要传个空数组。如果不传，只要状态改变就会触发回调函数，触发回调又改变状态，可能会陷入死循环！

14. 导入路径设置别名



15. antd 组件语言设置

如下，引入 `ConfigProvider` 和 `zhCN` ，用 `ConfigProvider` 将组件包裹住

```
import { Table, ConfigProvider } from 'antd';
import zhCN from "antd/es/locale/zh_CN";
```

```
export default function TableList() {
    return (
        <ConfigProvider locale={zhCN}>
            <Table
                className="user-group-table"
                columns={columns}
                dataSource={userGroup}
            />
        </ConfigProvider>
    )
}
```

16. react-router-config 的用法

    这是一个辅助react-router的插件，主要是使用配置文件集中式管理路由

    如果不使用这个插件，嵌套路由的定义就比较麻烦。如下，Main组件中有 `/` 和 `/page1` 两个路由，而Page1组件中还有 `/page1/page2` 和 `/page1/page3` 两个路由。可以看到，路由是定义在页面组件里面的。

```
import React from 'react';
import { BrowserRouter as Router,Route} from 'react-router-dom';
import Home from './Home';
import Page2 from './Page2';
import Page3 from './Page3';

function Page1() {
    return (
        <>
            <div>这是Page1组件</div>
            <Router>
                <Route path="/page1/page2" component={Page2} />
                <Route path="/page1/page3" component={Page3} />
            </Router>
        </>
    )
}

function Main() {
    return(
        <>
            <div>这是Main组件</div>
            <Router>
                <Route path="/" exact component={Home} />
                <Route path="/page1" component={Page1} />
            </Router>
        </>
    )
}
export default Main;
```

    使用插件之后，所有的路由可以集中放到一个配置文件里面：

```
import React from 'react'
import { Redirect } from 'react-router-dom'
```

```
export default [
    {
        path:"/",
        component: React.lazy(_ => import("../application/Home")),
        routes:[
            {
                path:"/",
                exact:true,
                render:() => (
                    <Redirect to={"/recommend"}/>
                )
            },
            {
                path:"/recommend",
                component: React.lazy(_ =>
import("../application/Recommend")),
            },
            {
                path:"/singers",
                component: React.lazy(_ =>
import("../application/Singers")),
            },
            {
                path:"/rank",
                component: React.lazy(_ => import("../application/Rank")),
            }
        ]
    }
]
```

在页面组件中通过 `renderRoutes` 渲染配置文件中的组件，一次只能渲染一层，即上面的Home组件

```
import React, { Suspense } from 'react';
import routes from './routes/index'; // 引入配置路由
import { renderRoutes } from 'react-router-config'
import { Layout } from 'antd';
//redux
import { Provider } from 'react-redux'
import store from './store/index'

function App() {
    const { Header, Sider, Content } = Layout;
    return (
        <Provider store={store}>
            <Layout>
                <Header />
                <Layout>
                    <Sider />
                    <Content>
                        <Suspense fallback={<div>loading</div>}>
                            {renderRoutes(routes)}
                        </Suspense>
                    </Content>
                </Layout>
            </Layout>
        </Provider>
```

```
    )
}
```

`renderRoutes(routes)` 函数会把子路由routers作为props传入到Home组件中，在Home组件中使用相同的方法渲染它的子路由

```
import React from 'react';
import { renderRoutes } from "react-router-config"; // 这边不用再引入配置，参数会传进来
import { NavLink } from 'react-router-dom'

function Home(props) {
    const { route } = props;
    return (
        <div className="main">
            <div className="title">This is Home component!</div>
            <div className="tabs">
                <NavLink to="/recommend">
                    <span>推荐</span>
                </NavLink>
                <NavLink to="/singers">
                    <span>歌手</span>
                </NavLink>
                <NavLink to="/rank">
                    <span>排行榜</span>
                </NavLink>
            </div>
            <div className="sub-pages">
                {renderRoutes(route.routes)}
            </div>
        </div>
    )
}

export default React.memo(Home);
```

17. Link 和 NavLink 的区别

    需要点击按钮进行页面跳转，如果用 `<a>` 标签实现，在每次点击时，页面被重新加载。React Router提供了 `<Link>` 组件用来避免这种状况发生。当 你点击 `<Link>` 时，url会更新，组件会被重新渲染，但是页面不会重新加载

```
// to为string
<Link to="/about">关于</Link>

// to为obj
<Link to={{
  pathname: '/courses',
  search: '?sort=name',
  hash: '#the-hash',
  state: { fromDashboard: true }
}}/>

// replace
<Link to="/courses" replace />
```

`<Link>` 使用to参数来描述需要定位的页面。它的值既可是字符串，也可以是location对象（包含pathname、search、hash、与state属性）如果其值为字符串，将会被转换为location对象

replace(bool)：为 true 时，点击链接后将使用新地址替换掉访问历史记录里面的原地址；为 false 时，点击链接后将在原有访问历史记录的基础上添加一个新的纪录。默认为 false；

Link 组件最终会渲染为 HTML 标签 `<a>` ，它的 to、query、hash 属性会被组合在一起并渲染为 href 属性。虽然 Link 被渲染为超链接，但在内部实现上使用脚本拦截了浏览器的默认行为，然后调用了history.pushState 方法

`<NavLink>` 是 `<Link>` 的一个特定版本，会在匹配上当前的url的时候给已经渲染的元素添加参数

```
// activeClassName选中时样式为selected
<NavLink
  to="/faq"
  activeClassName="selected"
>FAQs</NavLink>

// 选中时样式为activeStyle的样式设置
<NavLink
  to="/faq"
  activeStyle={{
    fontWeight: 'bold',
    color: 'red'
   }}
>FAQs</NavLink>
```

18. props两种使用方法

```
import React from 'react';

// props 解构赋值
function MyComponent(props) {
    const { user, role } = props;
    return <div>用户：{user}，角色：{role}</div>
}

// 直接在形参里面解构
function MyComponent({user, role}) {
    return <div>用户：{user}，角色：{role}</div>
}

function Main() {
    const role = "admin"
    return (
        <MyComponent user="zhangsan" role={role}/>
    )
}

export default Main
```

19. 路径拼接注意事项

```
background: url("~@/assets/img/login-bg.png")
```

20. React 实现路由跳转

除了上面提到的 `<Link>` 标签，react-router-dom 内置的 useHistory hook 也可以实现跳转

```
import React from 'react'
import { useHistory, Redirect } from 'react-router-dom'

const MyComponent = () => {
    const history = useHistory(); // 使用 useHistory hook
    const openUser = () => {
        history.push("/userManage");
    }
    return <a onClick={openUser}>打开用户管理页面</a>
}
```

21. 行内事件监听器绑定函数的两种写法

```
import React from 'react'

// 如果不用进行额外传参，直接写函数名称
const MyComponent = () => {
    const handleClick = (e) => console.log(e.target)
    // 这样写会进行隐式传参，例如事件对象
    return <a onClick={handleClick}></a>
}
// 需要传参，外面再嵌套一层函数 () => func()
comst MyComponent = () => {
    const handleClick = (param) => console.log(param)
    return <a onClick={() => handleClick("Clicked!")}></a>
}
// 下面两种写法是错误的
<a onClick={handleClick()}></a>
<a onClick={() => handleClick}></a>
```

> 确保传进去的是函数表达式，而不是执行结果

22. 列表渲染记得加 key

- andt Table 的 columns 配置项
- antd Table 的 dataSource
- antd Table 在 columns 的 render 函数中进行列表渲染

```
const columns = [
    {
        title: '角色实例',
        dataIndex: 'roles',
        key: 'roles', // 有dataIndex可不用加key
        render: (text, record) => {
            return text.map(item => <span key={item}>{item}</span>)
        }
    }
]
```

23. 使用antd的form组件，不需要进行数据绑定，表单校验的时候就能获取到数据

```
const handleOk = async () => {
    try {
        // 表单校验 && 获取数据
        const values = await form.validateFields();
        console.log(values)
    } catch (error) {
        console.log('form failed', error)
    }
}
```

> 如果使用数据绑定，需要好几个 `useState` ，还要考虑什么时候清空数据

24. useCallback 的使用场景

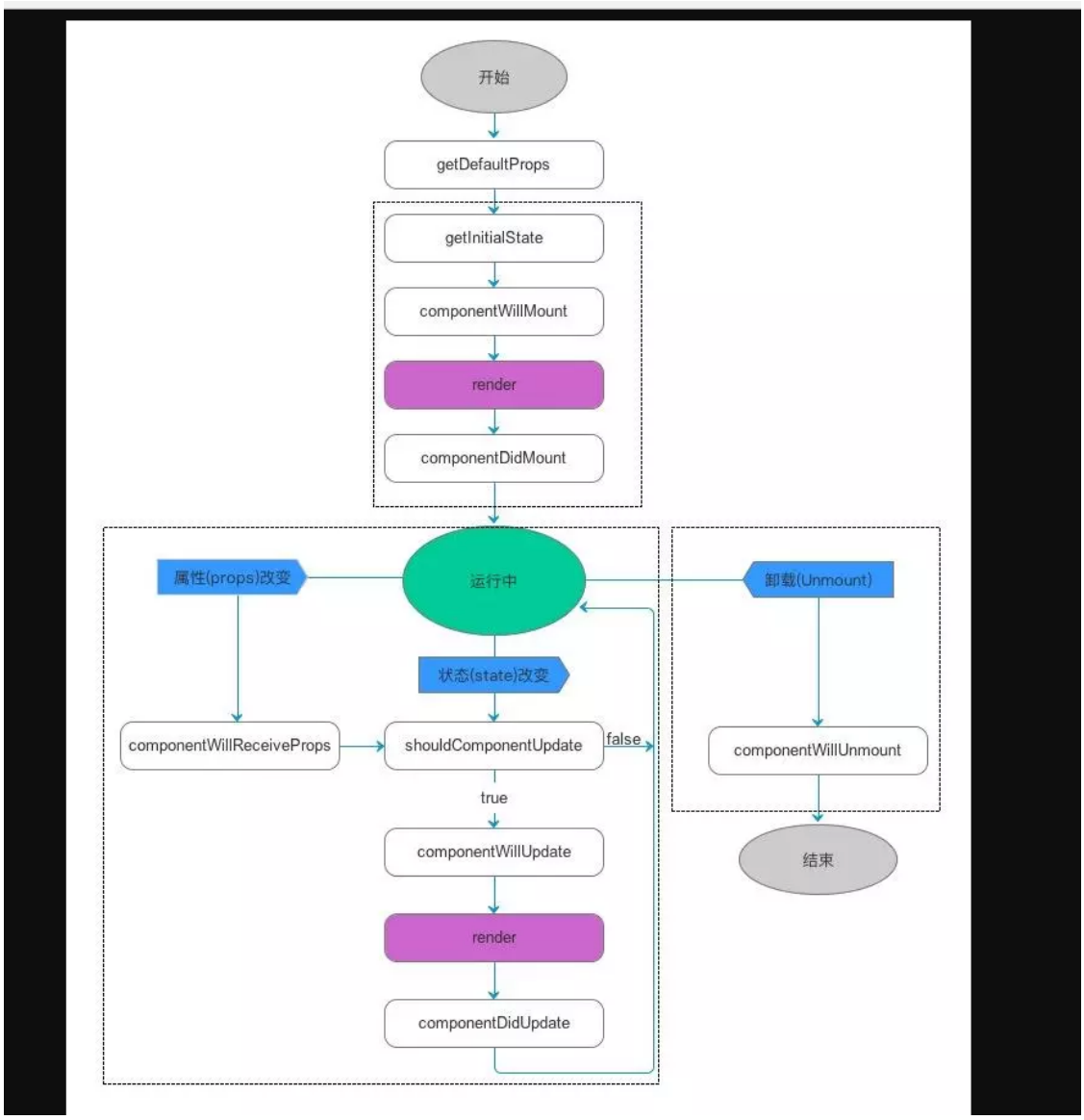
用于缓存组件渲染会触发的方法，防止状态更新后组件重新渲染导致不必要的执行

```
import React, { useState, useCallback } from 'react'

function MyComponent() {
    const [count, setCount] = useState(0);
    const [otherState, setOtherState] = useState(0)

    // 这个函数跟JSX是绑定的，在组件第一次渲染的时候会执行
    // 函数组件状态改变的时候（例如useState）会重新渲染
    // 因此组件重新渲染的时候又会触发这个函数
    // 我们只希望在 count 更新的时候执行以下，otherState 更新不执行
    const computeClickTimes = () => {
       // do something
       return `You clicked ${count} times!`;
    }
    // 使用 useCallback 包装之后，函数只会在依赖项发生变化的时候执行
    const computeClickTimes = useCallback(() => {
        // do something
       return `You clicked ${count} times!`;
    }, [count])

    return (
        <React.Fragment>
            <div>{computeClickTimes()}</div>
            <button onClick={() => setCount(count++)}>
                Click Me
            </button>
            <button onClick={() => setOtherState(otherState++)}>
                Other Button
            </button>
        </React.Fragment>
    )
}
```

> useCallback 用于缓存函数，useMemo 用于缓存计算结果的值
>
> `useCallback(fn, deps)` 相当于 `useMemo(() => fn, deps)`
>
> useCallback 可以使用，useMemo 不建议作为最佳实践

25. Class 组件生命周期

Class 组件 `state` 或者 `props` 修改后，如果 `shouldComponentUpdate` 里面没有判断，那么就会触发整个组件更新，同时触发 `componentDidUpdate` 钩子

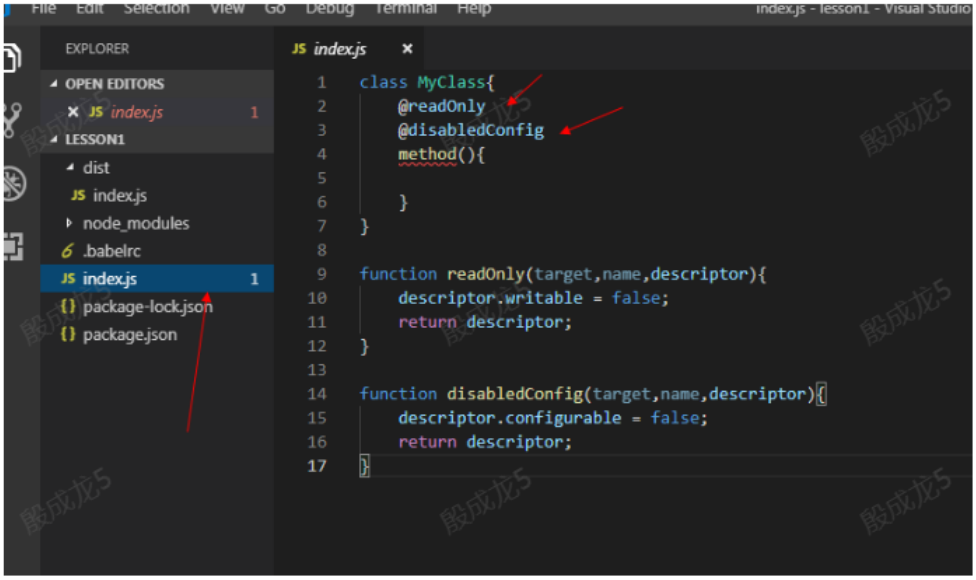


> 类似函数组件 useState 修改之后会触发整个组件更新

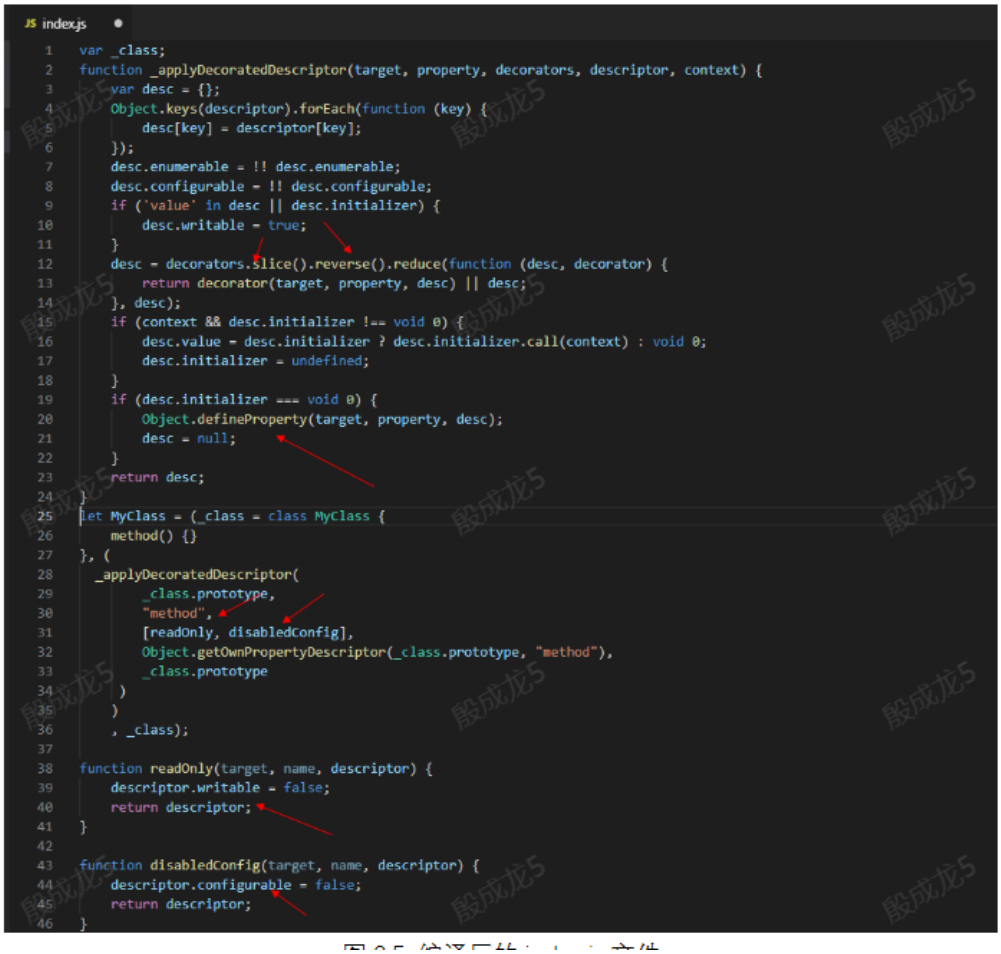
> useEffect 可以看作 componentDidMounted ， componentDidUpdate，componentDidUnmounted 三个函数的组合

26. 装饰器实现原理

    修饰器（Decorator）是一个函数，用来修改类的行为。修饰器对类的行为的改变，是代码编译时发生的，而不是在运行时。这意味着，**修饰器能在编译阶段运行代码**



这是编译后的代码



27. useEffect中使用定时器

    这里 `clearInterval` 的执行时机是什么

```
useEffect(() =>{
    const interval = startRefresh && setInterval(()=>{
        searchLogDetail();
    },5000);
    return () => clearInterval(interval); // 通过闭包缓存interval引用
},[startRefresh])

let interval; // 变量声明放在里面会导致引用丢失，定时器无法停止
useEffect(() => {
    if(!interval) clearInterval(interval);
    interval = startRefresh && setInterval(searchLogDetail, 5000);
}, [startRefresh])
```

28. react-redux 中 `@connect` 装饰器的用法

```
import React from 'react'
import { connect } from 'react-redux' // 从 react-redux 模块中引入 connect
import { addAppVar,changeVarRow,deleteVarRecord } from
'@/store/scriptManageRedux'

// mapStateToProps 是一个方法，返回 state 中的状态
// connect 内部会给 mapStateToProps 传参
const mapStateToProps = state => ({
    urrentOpen: state.scripts.currentOpen,
    curStepIndex: state.scripts.currentOpen.curStepIndex
})
// mapDispatchToProps 是一个对象，里面包含修改状态的方法
const mapDispatchToProps = { addAppVar, changeVarRow, deleteVarRecord }
// connect 会装饰离他最近的那个class
@connect(mapStateToProps, mapDispatchToProps)
class MyComponent extends React.Component{
    constuructor(props){
        super(props);
        this.state = {
            // 访问 connect 传进来的状态
            // 就跟正常的 props 一样访问
            this.urrentOpen = this.props.urrentOpen;
            ...
        }
        this.tableColumns = [
            ...
        ]
    }
    myMethod = () => {
        // 访问 connect 传进来的方法
        const { addAppVar } = props;
        addAppVar();
    }
    render() {
        {/* 访问 connect 传进来的状态 */}
        return <div>{this.props.curStepIndex}</div>
    }
}

export default MyComponent
```

> 关于 class 组件的说明
>
> 如果不需要定义state，constructor可以省略
>
> 如果用了constructor就必须要有super，否则访问 this 会报错

29. 组件内部如何定义多个子组件

    例如一个 `<BtnList/>` 组件，里面有各种不同类型的按钮

```
import React from 'react'
import { Button } from 'antd'

class Valid extends React.Component{
    render() {
        return <Button>校验按钮</Button>
    }
}
class Submit extends React.Component{
    render() {
        return <Button>提交按钮</Button>
    }
}
class NextStep extends React.Component{
    render() {
        return <Button>下一步按钮</Button>
    }
}

export default { Valid, Submit, NextStep }
```

    在其他组件可以这样使用

```
import React from 'react'
import BtnList from './BtnList'

class MyComponent extends React.Component{
    render() {
        return (
            <BtnList.Valid />
            <BtnList.Submit />
            <BtnList.NextStep />
        )
    }
}
```

30. class 组件的 this 绑定

    一种是通过 bind 函数实现

```
import React from 'react'
import { Button } from 'antd'

class MyComponent extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
            ...
```

```
        }
        // JS中class的实例方法默认不会绑定this
        // 在回调中使用this必须进行绑定
        this.myMethod = this.myMethod.bind(this);
    }
    myMethod() {
        console.log("打印内容", this)
    }
    render() {
        return <Button onClick={this.myMethod}>Click me!</Button>
    }
}
```

另一种是通过箭头函数实现

```
import React from 'react'
import { Button } from 'antd'

class MyComponent extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
            ...
        }
    }
    myMethod() {
        console.log("打印内容", this)
    }
    render() {
        // 在函数组件中如果不用额外传参，可以直接这样写
        // 事件对象会被隐式传递
        // <Button onClick={myMethod}>Click me!</Button>
        return <Button onClick={(e) => this.myMethod(e)}>Click me</Button>
    }
}
```

31. React.Children的用法

    React 中的 props 是单向数据流，但是有一个例外 `this.props.children` ，可以获取到当前组件的子节点

    `this.props.children` 的值有三种可能：

    - 如果当前组件没有子节点，它就是 undefined ;
    - 如果有一个子节点，数据类型是 Object;
    - 如果有多个子节点，数据类型就是 Array。

    > 所以，处理 this.props.children 的时候要小心

    React 提供一个工具方法 React.Children 来处理 `this.props.children` 。可以用 React.Children.map 来遍历子节点，而不用担心 `this.props.children` 的数据类型是 undefined 还是 Object Array

```
import React from 'react'
import { Form } from 'antd'

class MyComponent extends React.Components{
    render() {
```

```
        const { children } = this.props;
        return (
            <Form layout="horizonal">
                <Form.Item label="姓名">
                    <Input />
                </Form.Item>
                {
                    React.Children.map(children,child=>{
                        return React.cloneElement(child,
{form:this.props.form})
                    })
                }
            </Form>
        )
    }
}
```

> React.Children 只是一个方法的集合，提供了 map/forEach 等方法用来处理子节点

32. ReactDOM.render
33. koa的洋葱模型
34. redux里combineReducer，手写
35. connect的实现原理

参考资料

react-router-config 大致用法

react-redux

redux-persist

immutable.js

常用的几个 hooks

React 项目引入 less

安装 `less` 和 `less-loader` 依赖

在 `webpack.config.js` 中针对 `.less` 文件启用 `less-loader`

```
const lessRegex = /\.less$/;
const lessModuleRegex = /\.module\.less$/;

{
    test: lessRegex,
    exclude: lessModuleRegex,
    use: getStyleLoaders(
        {
            importLoaders: 2,
            sourceMap: isEnvProduction
            ? shouldUseSourceMap
            : isEnvDevelopment,
```

```
        },
        'less-loader'
    ),
    // Don't consider CSS imports dead code even if the
    // containing package claims to have no side effects.
    // Remove this when webpack adds a warning or an error for this.
    // See https://github.com/webpack/webpack/issues/6571
    sideEffects: true,
},
```

Service 目录

`axios.js` 主要是 `axios` 的封装，请求拦截器

`local-data.js` 是左侧导航栏的内容

`userManage.js` 现在里面定义了用户管理相关接口

Router 目录

路由配置

setupProxy.js

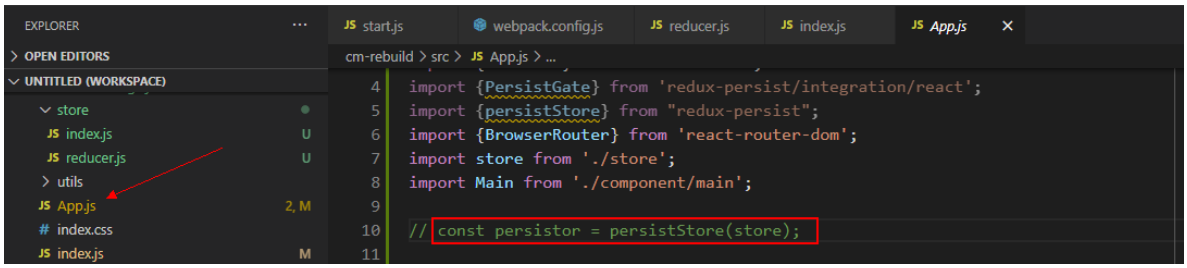
代理配置

.env 开头的文件

环境变量配置

App.js

里面包含 `redux-persist` 配置，导入的时候有报错，目前不清楚问题在哪里



重构任务

1. 引入 async/await

```
async function getPublicKeyAction(record){
    let result = await getPublicKey();
    const encrypt = new JSEncrypt();
    encrypt.setPublicKey(result.publicKey);
    // const encrypted = encrypt.encrypt(data);
    if(Object.keys(result).length>0){
        let modifyFlag= await modifyUserPassword(record.username,record);
        if(modifyFlag){
            Modal.success({
                content:"用户密码修改成功！"
            })
        }
        return dispatch =>{
                dispatch(setPublicKeyAction(result.publicKey));
            }
        }
}
```

action这里要是一个对象，不能是函数处理

```
const setPublicKeyAction = (keyData) =>({
    type:actionTypes.SET_PUBLIC_KEY,
    payload:keyData
})
```

海康大数据管理平台

服务器密码

> 10.3.69.93-95集群是我们组的调试环境
>
> 10.3.69.108-110环境是大组的集成测试环境，连上去的话没有问题，不进行危险操作就行

http://10.3.69.93:8877/cm/

账号：admin

密码：hikvision12345+

服务器：root, 2343, vision123456HIK*

http://10.3.69.94:8877/cm/

账号：admin

密码：hik12345+

http://10.3.71.102:8877/cm/

账号：admin

密码：hik12345-*/

服务器：root，2343，hika1b2c3+

http://10.3.71.104:8877/cm/

账号：admin

密码：hik12345-*/

服务器：root，2343，hika1b2c3+

http://10.3.71.106:8877/cm/

账号：admin

密码：hik12345-*/

服务器：root，2343，hika1b2c3+

这个可以用：

http://10.3.68.139:8877/cm/

账号：admin

密码：hik12345+

服务器：10.3.68.139:2343，root，hik12345+

http://10.3.69.108:8877/cm/

账号：admin

密码：hikvision12345+

http://10.3.69.110:8877/cm/

账号：admin

密码：hikvision12345+

http://10.3.72.160:8877/cm/

账号：admin

密码：hikyy12345+

http://10.3.75.212:8877/cm/

账号：admin

密码：hikyy12345+

http://10.3.72.155:8877/cm/

账号：admin

密码：hik12345+

服务器：10.3.72.155:2343，root，vision123456HIK*

服务器管理平台

http://rdknown.hikvision.com/rd/doc/knowledge/view?id=8f29292e-c031-4666-b615-7b4a8c741501&classname=com.hikvision.knowledge

服务器更换文件流程

1. 停止 lark server，命令如下：

```
$ /usr/lib/cloudmanager/sbin/stop-cloudmanager.sh
```

2. 更换文件
3. 重启 lark server，命令如下：

```
$ /usr/lib/cloudmanager/sbin/start-cloudmanager.sh
```

4. 执行下面的命令，如果两个端口都起来了（LISTEN状态）就正常了

```
$ lsof -i:8877
$ lsof -i:7897
```

租户白名单功能

10.3.72.160/10.3.75.212

admin hik12345+ root vision123456HIK*

接口详见下载里面的文档

支持双网卡

调式地址：http://localhost:3000/#apps/deployManager/dispatchPackage

> 调试环境：10.3.72.155
>
> IP端：10.3.72.155 - 10.3.72.157
>
> ssh端口：2343
>
> root密码：vision123456HIK*

> 调试环境：10.3.69.110
>
> IP端：10.3.69.108 - 10.3.69.115
>
> ssh端口：2343
>
> root密码：hika1b2c3+

> 调试环境：10.3.69.94
>
> IP端：10.3.69.94 - 10.3.69.95
>
> ssh端口：2343
>
> root密码：vision123456HIK*

发请求的时候不用真的发出去，把数据打印出来看看

备份路径：/mnt/disk1/backup

安装第六步：

HIKCLOUD 需要一个同网段 ping 不通的节点，例如：10.3.69.220

FREEIPA 需要配置下

FLINK 需要提供一个ip，使用当前环境的ip

服务实例接口

http://10.3.75.212:7897/api/lark/k8s/pods?serviceId=service-kafka-yyy你直接把请求最后的serviceId变成现在这个样子，返回给你的应该就是一个List。原来是serviceId=service-kafka-yyy-broker-0

把后面的-broker-0去掉。参数只包含“service-服务名-租户名”就行

以前返回去的也是一个List，只不过每个List里都只有一个元素，因为参数级别太细了

你再看一下这个接口前端有没有展示异常pod的detailUrl

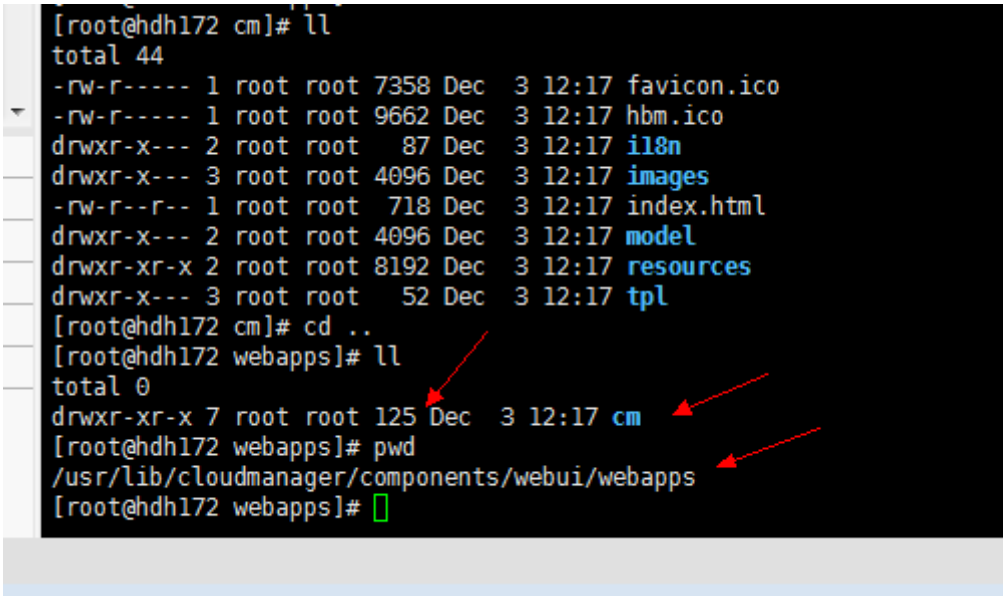
查看线上环境打包的日期

进入 webapps 目录：

```
$ cd /usr/lib/cloudmanager/components/webui/webapps
```

查看打包日期：

```
$ ll
```



待办事宜

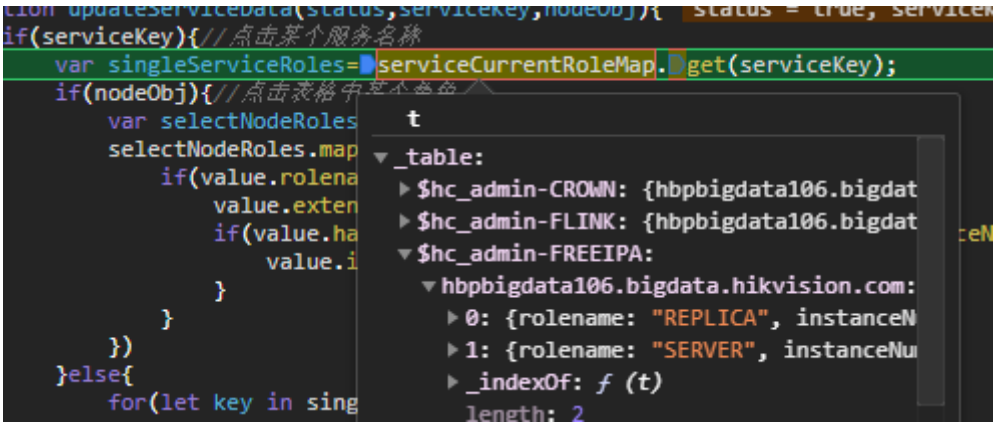
可以登录 OA 系统查看。

http://defect.hikvision.com.cn/approve/document?flowInstId=93879506

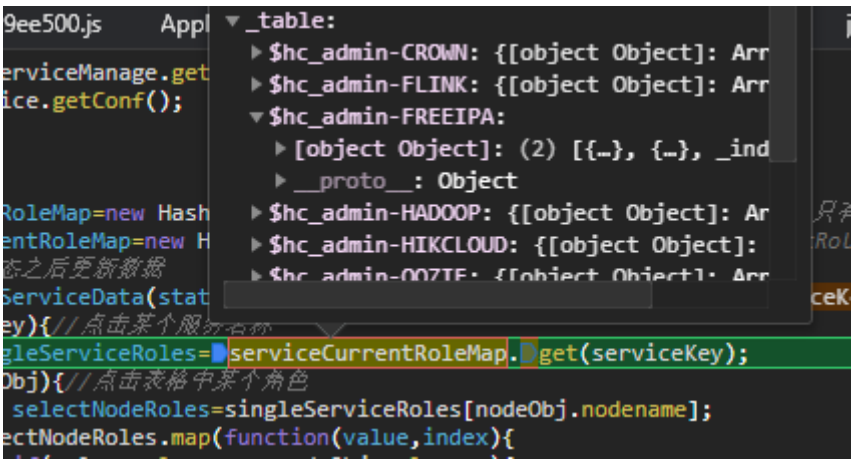
http://defect.hikvision.com.cn/approve/document?flowInstId=93873338

- [x] 千兆网卡默认不选
- [x] 部署节点的节点列表方框放到前面
- [x] 帮助这里鼠标hover应该出现手型、
- [x] 节点管理/集群扩容/添加节点，宽度不够的时候主机名重叠了
- [x] 白名单后端返回map改一下
- [x] 服务部署完成后跳转404
- [x] 环境部署，磁盘支持用户自己添加，如果在已有列表中提示重复
- [x] 刘工、朱乔工遗留问题 http://defect.hikvision.com.cn/approve/document?flowInstId=94174818
- [x] 陈工问题
- [x] 韩工测试问题
- [x] 首页增加集群信息与操作
- [x] 环境部署下一步按钮禁用问题修复
- [x] 一月中旬前所有问题都先solved
- [x] 添加时钟源增加内部/外部选项，前端已经加上，等待后端接口
- [x] 创建集群备份路径是否要加复选框跟韩工确认下
- [x] 集群扩容修改下
- [x] 如果没有nodePort，就展示"本地网络，请访问服务端口"说明
- [x] 贾工、余工的项目缺点单结掉
- [x] 用户文档图片换成老项目的

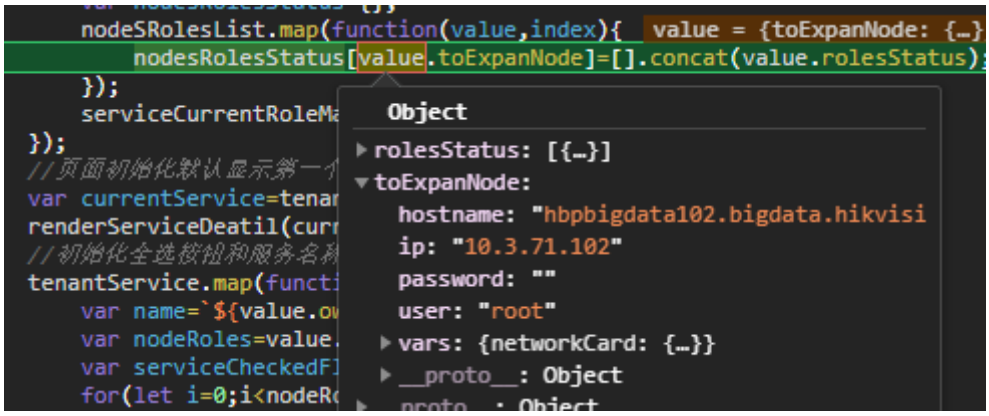
经过添加节点这步，键值是hbpbigdata106.bigdata.hikvision.com



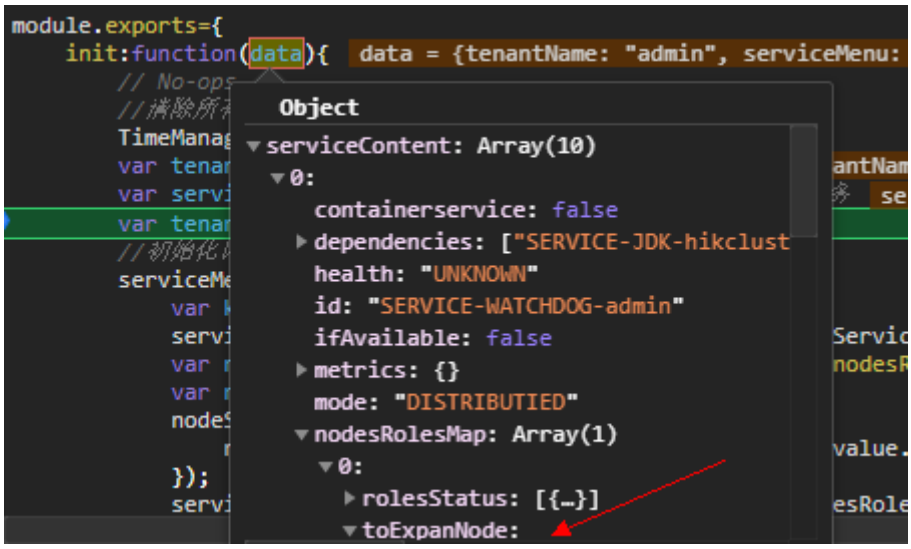
跳过添加节点这步，键值变成了一个对象



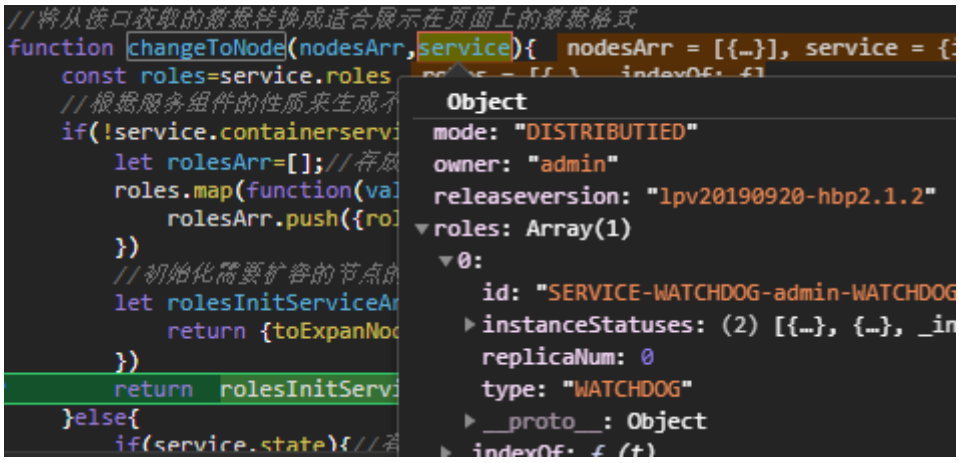
toExpanNode变成对象



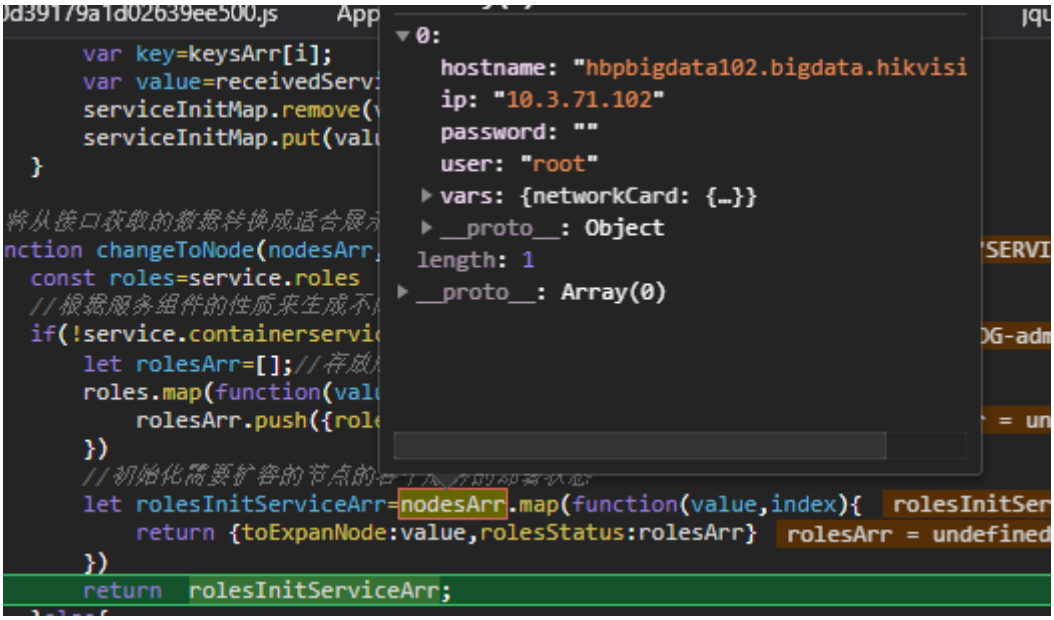
expanService.js传进来的数据，nodesRolesMap里面，每个元素的toExpanNode都是对象



service.roles 里面每个元素不应该是对象



nodesArr 里面每个元素都是对象，导致toExpanNode变成对象



传入changeToNode第一个参数nodesList从localStorage获取：