<https://www.jianshu.com/p/8b7def32740f>

<https://blog.csdn.net/scu_cindy/article/details/82914547> //脚手架搭建

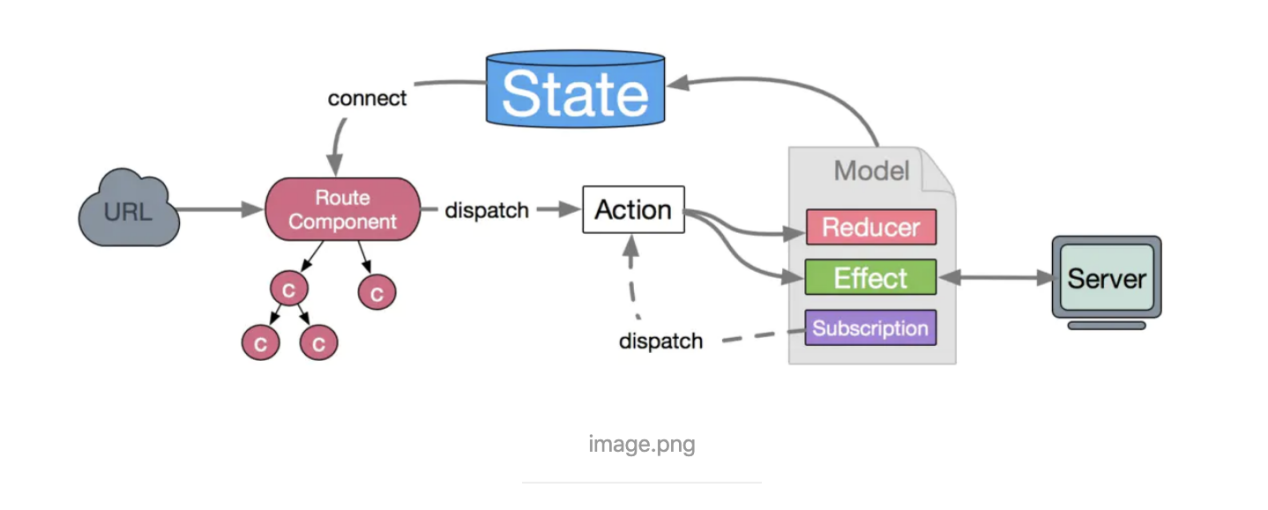
yarn create umi

<https://www.cnblogs.com/weschen/p/12190913.html> 安装完umi之后，输入 umi ui可以可视化构建脚手架

例子：裕华pc 、1dong 、dva练习

数据的改变发生通常是通过用户交互行为或者浏览器行为（如路由跳转等）触发的，当此类行为会改变数据的时候可以通过 dispatch 发起一个 action，如果是同步行为会直接通过 Reducers 改变 State ，如果是异步行为（副作用）会先触发 Effects 然后流向 Reducers 最终改变 State

dva和umi一起使用



项目总结：

1.是否让pages里面的文件自动识别为路由，需要修改.umirc.js文件。删掉routes属性，就可以自动识别pages文件了。

2.入口文件是layouts文件夹里的index.js,

<Content

style={{ background: '#fff', margin: 0, minHeight: 280, }}>

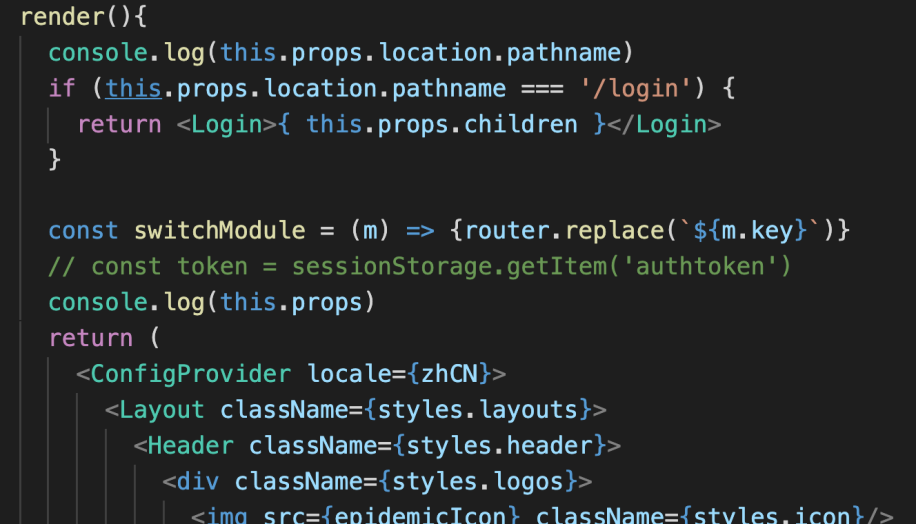
{this.props.children}

</Content>

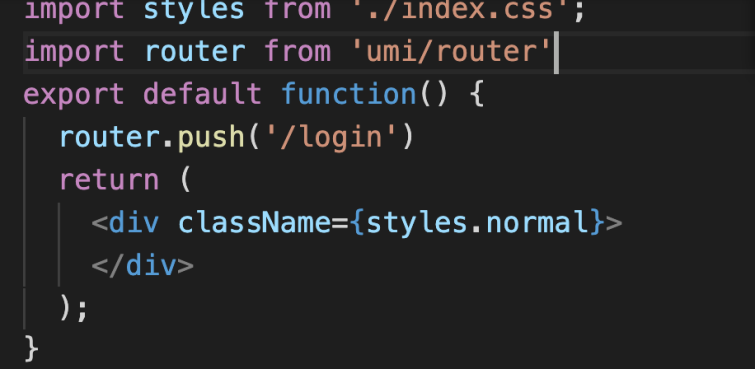
This.Props.children是当路由匹配成功时组件渲染的地方。

Index.js进行路由跳转设置。匹配到的路由直接找到pages文件夹里的文件，文件名就是路由名，进行渲染。

1. 项目有登陆页面的/login页面。在layouts的index.js文件里进行拦截login



在pages里面的index.js文件进行 router.push('/login')



加载顺序是：先加载layouts里的文件，匹配到 / 的index.js，加载pages里的index.js的时候。触发 router.push('/login'),再次加载layouts的index.js，进行if (this.props.location.pathname === '/login') 的判断，进行加载login页面。

每次push到新的路由都会重新加载layouts里面的index.js.执行对应的路由匹配，去加载匹配到的（pages里）文件。

umi的使用

第一种

import Link from 'umi/link';

<Link to="/list?a=b">Go to list page</Link>

import router from 'umi/router';

// 带参数

第二种

router.push('/list?a=b');

第三种

router.push({

pathname: '/list',

query: {

a: 'b',

},

});

获取

this.props.location.query

umi约定式嵌套路由



Modul

Subscription

Subscription 语义是订阅，用于订阅一个数据源，然后根据条件 dispatch 需要的 action。数据源可以是当前的时间、服务器的 websocket 连接、keyboard 输入、geolocation 变化、history 路由变化等等。

subscriptions: {

setup({ dispatch, history }) {

return history.listen(({ pathname, query }) => {

//本地使用

if (pathname === '/') {

if (query.appName) {

dispatch({type:'getToken', payload:query }).then(res=>{

if(res.status===0){

dispatch({ type: 'login', payload: {appName:query.appName,token:res.data}});

}

})

}

Effect

Effect 被称为副作用，在我们的应用中，最常见的就是异步操作。它来自于函数编程的概念，之所以叫副作用是因为它使得我们的函数变得不纯，同样的输入不一定获得同样的输出。  
dva 为了控制副作用的操作，底层引入了redux-sagas做异步流程控制，由于采用了generator的相关概念，所以将异步转成同步写法，从而将effects转为纯函数

put如果调用同步的reducers是阻塞的．call执行异步任务，select是获取state.

const { pageSize} = yield select(s => s.fullStaff)

fullStaff是model的namespace

1

\* getTreeCounting ({ payload }, { put, call, select }) {

const { formSelect } = yield select(s => s.fullStaff)

const res = yield call(fullstaff.getCounting, { params: { ...payload,...formSelect} })

const { data, code } = handleResponse(res)

//'update'是reduce里的方法

yield put({ type: 'update', payload: { categoryCounting: data.result.categoryCounting,peopleCounting:data.result.peopleCounting}})

},

2

\*getlog({ payload: { page,size } }, { call, put }) {

const { data } = yield call(usersService.getLog, page,size);

yield put({

type: 'getlist',//'getlist'是reduce里的方法

payload: {

data: data

},

});

},

Reducer（而 dipatch 可以看作是触发这个行为的方式，而 Reducer 则是描述如何改变数据的。）

通过 actions 中传入的值，与当前 reducers 中的值进行运算获得新的值（也就是新的 state），需要注意的是 Reducer 必须是纯函数，所以同样的输入必然得到同样的输出，它们不应该产生任何副作用

saveType(state, { payload: { type } }) {

return { ...state, type };

},

State

State 表示 Model 的状态数据，通常表现为一个 javascript 对象（当然它可以是任何值）；操作的时候每次都要当作不可变数据（immutable data）来对待，保证每次都是全新对象，没有引用关系，这样才能保证 State 的独立性，便于测试和追踪变化

namespace: 'users',

state: {

list: [],

total: null,

page: null,

startTime: null,

}

Action

Action 是一个普通 javascript 对象，它是改变 State 的唯一途径。无论是从 UI 事件、网络回调，还是 WebSocket 等数据源所获得的数据，最终都会通过 dispatch 函数调用一个 action，从而改变对应的数据

需要注意的是 dispatch 是在组件 connect Models以后，通过 props 传入的

dispatch({ type: 'initList' })

后面也可以跟then()，但是effects里面需要返回值。

this.props.dispatch({type: 'fullStaff/getHouseList', payload: { id } }).then(res =>{

if(res.code === 2000) {

this.setState({houseList:res.data.result})

}

})

effects里

\* getHouseList ({ payload },{ put, call }) {

const res = yield call(fullstaff.getHouseList, { params: payload })

const response= handleResponse(res)//handleResponse处理一下返回值

return response; //response为接口返回值

},

Connect

import { connect } from 'dva'

通过connect将modul中的元素作为props的方式传递给component

export default connect (({fullStaff}) => ({

tableList: fullStaff.tableList,

streetId: fullStaff.streetId,

communityId: fullStaff.communityId,

typeList: fullStaff.typeList,

categoryCounting: fullStaff.categoryCounting,

}))(Index)//Index组件名