children

The Section component currently renders its children elements.

```
export default function Section({ children }) {
  return (
    <section className="section">
      {children}
    </section>
  );
}
```

Other

import React, {**Component, PureComponent，FC, useState, userEffect, createRef, PropTypes, Fragment, Dispatch, ReactNode, ErrorBoundary** } from 'react'

PureComponent： setState 将原来的值重新赋值，不会触发 render)

Fragment 相当于空的<> </> 聚合子标签，并且不会渲染在 dom 上

Suspense 配置组件加载前干的事 <Suspense fallback={<div>Loading...</div>}> <OtherComponent /> </Suspense>

ReactNode JSX.Element 接口---返回一个 React.createElement 对象 ， ReactNode 接口---组件所有可能返回值的集合

FC 函数组件使用的泛型 const ampleModel: FC<{}> = () =>{ }

useEffect 无状态组件内部使用，第一次渲染之后和每次更新之后都会执行，给无状态组件提供了类似生命周期函数的功能

useEffect(() => { });

useEffect(()=>{}, [type]) 数组中所依赖的值发生了改变，useEffect 才会被重新执行，类似 componentDidUpdate

useContext

useReducer

useCallback

useMemo

useRef

useImperativeHandle

useLayoutEffect

useDebugValue

React.crerateElement('img', {id:'xx'}或 null , '123' , [mydiv]) 创建 jsx 虚拟 dom 对象（元素名，属性，子节点，其他子节点）

React.render(jsxobj, document.getElementById('root')) == ReactDOM.render(<App />, document.getElementById('root')); 将 jsx 对象 渲染成虚拟 dom 挂载到页面

React.createRef() let formRef = React.createRef() <Form ref={this.formRef}> const node = this.myRef.current; 获取 dom 对象

import {Button,Radio} from 'antd' 框架组件

import PageHeaderWrapper from '@/components/PageHeaderWrapper'; 自定义组件

import { SetCacheValue, GetCacheValue, RemoveCacheValue } from '@/utils/cache'; 自定义工具

import { connect } from 'dva'; 状态管理

import moment from 'moment'; 插件

import logo from './images/logo.png'　　图片路径（动态）

import styles from './index.scss'　　css 文件

import { Consumer } from "./App.js"　　父组件内部的 Consumer 接收数据

const myimg = xx　　JSX 对象　（使用 Babel JSX 插件 将 HTML 转换为 React.createElement 返回的 JS 对象）

const arr=[ab , cd]　　JSX 对象数组

Function Component

pages/Test.tsx >>

```
import { prependOnceListener } from "process";
import React, { useState, useEffect,useLayoutEffect, useMemo,useContext,useCallback, Suspense, Children, createContext } from "react";
const Test=React.lazy(()=>import("@pages/Test"))    //懒加载，只有使用到 Test 组件时才会加载，必须配合 Suspense 使用
const context1=createContext({name:""});      //上下文容器，减少 props 传递层级
const context2=createContext({age:23});       //上下文容器，减少 props 传递层级

export interface IApp {             //定义从父类接收的对象类型
  role: string;
  isCps?: boolean;
  onUpdateUser?: (params: any) => void;
}

const Grandson: React.FC<any>=()=>{
  const context1Val=useContext(context1);

  // return <div>grandson - <context1.Consumer>{value=>value.name}</context1.Consumer></div>      //传统方式的消费上下文
  return <div>grandson -{context1Val.name}</div>          //函数式组件可以直接使用上下文
}

const Child: React.FC<any>=React.memo((props)=>{          // momo 是一个高阶组件，作用和 PureComponent 相同
  console.log("child render",props.children)
  props.onMsg()
  return <>
    <div>child - {props.children.a} - {props.children.b}</div> 插槽
    <Grandson />
  </>
})

const App: React.FC<IApp> = ({ role, isCps }) => {                              // 定义函数组件

  const [showDetails, setShowDetails] = useState<boolean | number>(false);       //  定义内部数据 和 修改函数  // setShowDetails(true)
  const sdkRef = useRef<HTMLDivElement>(null);       //  定义一个引用，可以通过 sdkRef 访问对应的 dom 对象

  useEffect(() => {}, [showDetails]);       // 监听数据变化，数据发生变化才会执行函数【当执行{...obj}这种操作时会改变对象，使用 JSON.stringify(obj)可以避免频繁触发】
  useEffect(() => {}, []);                  // 第一次渲染之后 和 每次更新之后都会执行
  useLayoutEffect(() => {}, []);            // 第一次渲染之前 执行，可以防止闪烁


  const funcTestCallback=useCallback(()=>{    //useCallback 是缓存函数，需要和 memo 高阶组件一起使用，缓存第一个函数，不会执行，直接返回，避免组件重新渲染
    console.log("callback changed")
```

```
  },[])                                          //useCallback 监听值改变时，更新缓存函数，重新渲染
  const funcTestMemo=useMemo(()=>{               //useMemo 是缓存返回值，需要和 memo 高阶组件一起使用，避免组件重新渲染
    return {
      a:<span>123</span>,
      b:<span>456</span>
    }
  },[])                                        //useMemeo 监听值改变时，更新返回值，重新渲染
  console.log("App render")
  return <div>
    <Suspense fallback={<div>loading...</div>}>    {/* 加载时显示的组件 */}
      <button onClick={()=>{setShowDetails(!showDetails)}}>ShowDetails</button>
      <div>{String(showDetails)}</div>
      <div ref={sdkRef}>test ref<div>              {/* 使用 ref 引用 */}
      <Test />
      <context1.Provider value={{name:"zhangsan"}}>
        <Child onMsg={funcTestCallback}>{/* 直接传入新对象会让 memo 失效（插槽里的任何内容都会成为 children，即使是空格） */}
        {funcTestMemo}
        </Child>
      </context1.Provider>
    </Suspense>
  </div>
};
export default App
```

import Context, { Provider, Consumer } from React.createContext('defaultval')　　创建上下文

export default function Father(props**){**　　创建无状态组件（父级 props 只读，没有私有数据和生命周期函数）

　return <Provider value={ name:'99', kk:88 }>　　<Son name={name}></Son>　　</Provider>　　Provider 给内部所有组件传递数据

　return　<Son tempf="xx" func={ ()=>{ console.log(this.state.aa) } } arr={ [1,2,3] } key={1} >　　</Son>　　自定义属性传递值（key 值为便利时给子组件添加的，子组件不能接收 key）

}

function Xx(){

　return <Consumer>　　{ name =>{ {name} }}　　</Consumer>　　直接访问 Provider 内的 value 对象内部数据

}

Class Component

核心组件

@PageWrap({})　　外部嵌套高阶组件 export defaults　props ==> WrapComponent => { render(){ return <PageContainer> <WrapComponent> } }

export default class Son extends Component {　　【组件首字母大写】 创建有状态组件 （父级 props 只读，含有私有数据 this.state 实时刷新组件内属性值，含有生命周期函数）

　state={}　　初始化 state

　constructor(props){

　　super(props)

　　this.state={msg:'yy'}　　初始化 state 数据

　　this.increment = this.increment.bind(this);　　使内部函数能够访问 this.state 和 this.props

　}

　handleClick=(e)=>{　　定义事件，阻止冒泡

　　e.preventDefault()

e.nativeEvent.stopImmediatePropagation()

this.setState()

}

componentWillMount 初始化渲染之前调用一次 （方法内调用 setState，render 函数将会仅执行一次，无论 state 是否改变）】

render 创建虚拟 Dom 并挂载到页面 （根据虚拟 DOM 渲染成真实 DOM）

componentDidMount 初始化渲染之后调用一次 （在初始化渲染时不会调用 组件运行阶段的生命周期函数）】

componentWillReceiveProps(nextprops){} 组件接收到新的 **props** 的时候调用 （nextprops 为接受到的 props 值，state 没有此类方法）】

shouldComponentUpdate(nextprops, nextstate){} 组件接收到新的 **props** 或 **state** 之后调用（一般在这一步进行渲染劫持阻止更新，不能在内使用 setState，会无限循环更新组件）

componentWillUpdate(nextprops, nextstate){} 组件接收到新的 props 或 state 的时候调用，更新组件之前 （return false 将不会更新组件）

render 将创建更新的虚拟 Dom 挂载到页面 （diff 算法比较新旧 DOM，再修改）

componentDidUpdate(prevprops, prevstate) 组件更新之后调用（prevprops，prevstate 为之前更新组件的 props state 值）【在对象内部赋值 props 内部成员时，在追加到 state 中可能造成不更新情况】

componentWillUnmount 组件从 DOM 中移除之前调用

forceUpdate 强制更新（部分组件或全局）

render () {

if(xx) return <xx> return 内部不能使用标志符 if else 等,，return 外部可用

this.props.children this.props.text 获取双标签内部标签或文本（单标签无法获取）

this.setState({msg:'123'}，()=>{}) 修改并更新页面中的数据，不会覆盖其他数据（禁止在 render 内部调用，会造成死循环）

setState 调用时，将要修改的 state 放入队列，将一作用域内的多次 setState 的状态修改合并成一次状态修改，最终只执行一次组件重绘

return

<div> { a + 2 + bo? 'a' : 'b' } { myimg } </div> { }内解析为 js 代码（解析变量或 JSX 对象）

<div>{ **this.state.arr.map(** (val, ind) =>{ return <h1>{val}</h1>} **)** }</div> 遍历普通数组 => JSX 对象数组 （一个 JSX 对象只有一个根标签）

<div key={item} ref='dong' > </div> key 属性，遍历时保持愿状态（其他属性变化：class-->className，for-->htmlFor，style-->style={{ fontSize: '24px' }} ）

}

高阶组件：

const HOC = (Son) =>{ 高阶组件

class WrapperComponent extends Component {

render() {

return <Son {...this.props} {...newProps}/>； 获得 Son 获得的 props，在中间层改变 props 的值

}

}

}

export default HOC(Son)

属性——

Son.defaultProps={ 定义默认的 props

```
  items:0
}
Son.propTypes = {
  handleClick: PropTypes.func.isRequired   限制属性类型
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
  optionalSymbol: PropTypes.symbol,
}
```

Node / Vue

Concept

双向数据绑定

vue 是单向数据流，不是双向绑定的（双向绑定实现： 数据劫持+订阅发布）

数据劫持 Object.defineProperty(obj, key, {})

订阅发布/观察者模式

VUE 【this.$refs.mypagelist_foot.offsetTop 元素距离顶部的高度】 【window.screen.height 获取屏幕高度】【window.scrollY 获取滚动长度】 【this.$refs.myindex.offsetHeight 获取高度】

框架

框架（不容易切换） 库/插件（容易更换）

前端 MVVM **V** 视图（DOM） **VM** 中间观察者 **M** 数据（js 对象） 【视图和数据不能直接通信，通过 VM 实现双向数据绑定】

后端 MVC **C** 业务逻辑 **V** 视图（DOM） **M** 数据（js 对象） 【前端发生数据交互会刷新整个页面】

vue.config.js

根目录创建此文件会被 devserver 默认加载

```
module.exports={
  devServer:{
      open: true,  /*自动打开浏览器*/
      port: 3000,    /* 配置端口号 */
      proxy: {
          '/api': {
              target: 'https://c.com/mv',  /* 请求/api路径时，自动代理到指定网址  */   配置devserver的代理--解决跨域问题（上线使用nginx实现代理）
              pathRewrite: { '^api': '' }   /*pathRewrite 拼接地址自动替换api开头：  'https://c.com/mv/api/list  -> https://c.com/mv/api/list */ } } }
}
```

项目文件

main.js

```
import { createApp } from 'vue';
import VueRouter from 'vue-router';
import Vuex from 'vuex';
import VueLazyload from 'vue-lazyload';

Vue.use(VueRouter);  使用路由
Vue.use(Vuex);  使用 vuex 仓库
```

```
Vue.use(VueLazyload, {       使用 vue 懒加载
  attempt: 1,
  loading:   'https://otosaas.pek3a.qingstor.com/old/pro/mms/7e74c99f9b004cada6e9bdfb48459539.svg'
});
new Vue({
    el: '#app'                                                    实例控制的页面区域（不能给 body 加）
    data: { msg:'xxx', aa:new Data() }                            实例数据（可设置临时双向绑定数据，触发时添加到 list 列表里）
    methods: { show:function(){  this.msg  this.aa();  },  }     实例方法【事件触发函数】
    filters: { fname:function(data, xxx){}  }                                  私有过滤器（与全局同名时，私有优先）
    directives: { 'focus':{ bind: function(){} },  'test': function(){}  }     私有指令（省略对象默认将函数添加到 bind 和 update 里）
    components:{  v-xxx:{ name:'x'  v-xxx:'#templ', data(){return {}}, methods:{}, prop:['pmsg'] }, }      私有组件，可多个（需要手动添加 template 模板）【当前页面使用<v-xxx></v-xxx>】
    watch: {  msg:function(newVal, oldVal ){},  $route.path:function(){} }      【数据监听函数，msg 数据改变立即执行函数（vm 实例的路由数据等）】
    computed: { msg:function(){ return this.a+this.b }, }                       【计算属性，函数内部数据发生变化立即重新计算并返回，赋值给 msg 数据（v-model="msg"）  】
    render: function( createElements ){ return createElements({template:'<h1>xxx<h1>'}) }   将组件模板转换为 虚拟 dom（包含 data method 等）    替换 vue 实例 el 元素内所有内容

    router: routerObj    添加路由组件实现监听地址
    store: store          挂载 vuex 仓库
    beforeCreate (){}              实例创建之前---只含内置属性
    created (){}                   实例创建之后          （data 和 methods 初始化之后 ---适合发 ajax）
    beforeMount (){}               页面挂载之前          （元素模板在内存中编译完毕，页面只有空的模板页面）
    mounted (){}                   页面挂载之后          （页面中渲染表达式填充完毕，可获取 Dom 节点了 ---适合外界 UI 组件初始化）
        beforeUpdate (){}          data 数据更新到页面之前      （此时 data 数据已经改变，页面还是老数据）
        updated (){}               data 数据更新到页面之后
    beforeDestroy (){}             销毁之前（data 和 methods 等可用）
    destroyed (){}                 销毁之后（实例不可用）
}).$mount('#app')      挂载到页面的 app 容器内

Vue.component('myLogin', { template: '<h1>xx</h1>', data(){ return {msg:'xx',} }, methods:{} } )   全局组件，组件可以有自己的私有数据函数---必须返回数据对象使每个组件数据不共享
        Vue.component('myLogin', Vue.extend( { template: '<h1>xx</h1>' } ) )
        Vue.component('myLogin', { template:'#templ' })
        <template id="templ"> </template>                组件模板---所有组件模板只能有唯一的根节点 （template:''和<template>中元素都可以直接使用组件中的数据和函数）
        <component :is="'myLogin'"></component>          组件占位符，只显指定 id 的组件
        <my-login></my-login>                            组件在页面中使用（含大写需加-）
Vue.filter('fname', function(data 默认的管道前数据, msg){ return newdata; } )      全局过滤器          {{ name | fname | f2 }}  或 {{ name | fname(msg) | f2 }}
Vue.config.keyCodes.f2=113                                                  全局自定义键码      @keyup.f2="add"
Vue.directive('focus', {    bind:function(el, binding, vnode, oldVnode){}, }  )      全局自定义指令，    加 v-使用   <p v-focus></p>
        Vue.directive('focus', function(el, binding, vnode, oldVnode){} )        省略对象默认将函数添加到 bind 和 update 里
```

bind 仅元素绑定指令时调用一次---属性操作，此时没有插入 Dom 树 （先解析每个元素的各个属性到内存，再统一放到 dom 树中）

inserted 仅绑定元素插入到 DOM 调用一次---行为操作

updated 所有组件的 Vnode 更新时

componentUpdated 所有组件的 VNode 和组件的子 VNode 更新时

unbind 仅解绑时调用一次

el 绑定元素的原生的 js 对象

binding 一个对象 **name** 指令名，不含 v-前缀 **value** 绑定值的计算结果 v-xxx='1+1' ==> 值为 2

expression 绑定值的字符串形式

oldValue 指令绑定的前一个值 arg modifiers

App.vue

```
<template>
    <div id="app">
        <p>  11 {{  msg }} 22  </p>        页面渲染
        <p  ref="mydom"> </p>                          获取普通 Dom 元素        【this.$refs.mydom.innerText 】
       <son  ref="mycomp">  </son>                     接收子组件数据和方法      【  this.$refs.mycomp.show();  】
        <son :pmsg="msg">    </son>                    传递给子组件 pmsg 数据
       <son @pfun="show">    </son>                    传递给子组件 pfun 方法（可以用 show 接受$emit 的数据）
   <son><span>ssssssss</span></son>                内部内容    传入到 son 的默认坑中<slot>  </slot>
    <son>
     <template slot="myslot" slot-scope="scope">        内部内容    传入到 son 的指定坑中（scope 为子组件 slot 所在位置传入的数据 对象）
        <div v-for="item in scope.data">{{ item }}</div>
     </template>
   </son>

    </div>
</template>
<script>
   import HelloWorld from './components/HelloWorld'
   export default {
     methods:{
         show( sdata ){}                    接收子组件$emit 数据
      }
      components:{   v-helloworld: HelloWorld }    v-helloworld 才是这个组件最终的名字
   }
</script>
```

xx.vue

```
<template>                  内部只能有一个根标签，防止和原生组件冲突 template 应该命名为<v-xxx>
      <slot></slot>                                 默认坑（接收内容）
     <slot name="myslot" :data="list"></slot>     指定坑（接收指定内容，传入 msg 数据）
    <son :pmsg="msg"></son>     {template:'<br/>', props: ['pmsg'] }                          子组件使用父组件数据 props 属性数据---只读   data---读写
     <son @pfun="show"></son>   { template:'<br/>', methods:{ a(){  this.$emit('pfun', val 参数)  }}}   子组件调用父组件方法 调用父组件方法---可给父组件传递数据 this.sonmsg
     <xx :msg-val="msg">  接收自组件传递的值
```

```
</template>

<template id="templ">      可以多个模板，作为暴露组件 或私有组件
  <div></div>
</template>

<script>
    export default {       暴露数据对象（自动导入 template 内容，不需要手动写 template:""属性）
        name: 'v-xxx',      只能在当前文件使用，递归调用，自己调用自己<v-xxx></v-xxx>
        data(){  return{msg:'Hello'  },        内部数据（页面引号内使用  :pmsg="msg"，其他方法内使用  this.msg）
        methods:{
            funa(){
              this.$emit('pfun', val1,val2)      接收并调用 调用父组件方法 pfun
              this.$emit('sdata', 'xxx')          传递给父组件数据 sdata
            }
         },
        prop:['pmsg'],                                          接收父组件传递的属性值 pmsg
        prop:{
            pmsg: {
                type: Array,      指定接收类型（如果类型不对，会警告）
                default: [0,0,0]   这样可以指定默认的值
            }
       }
        aa:{name:'x', template:'#templ', data(){}, methods:{}, prop:['pmsg'] },      暴露组件对象，在其他文件使用（import { aa } from 'xx.vue')
    }
</script>
<style scoped 组件显示时激活样式 lang="scss"使用的语言>       最外层 div 用 css 属性选择器实现 scoped（默认为全局样式）【img 样式失效时可将 scoped 去除】
</style>
```

命令
html 指令

```
<p v-cloak> 11 {{ msg }} 22  </p>       防止渲染表达式闪烁，防止遍历内部 引号内变量造成图片闪烁     [v-cloak] { display:none }
<p v-text="msg"></p>                    覆盖双标签内容          【vue 可以直接在属性字符串里访问到变量】
<p v-html="msg"></p>                    覆盖双标签 HTML 内容
<p v-if="true" v-if="index<3"></p>        重新创建或彻底删除元素（从虚拟 dom 上，不能和 v-for 一起使用）
<p v-else-if="true" ></p>                 语句（限兄弟元素使用）
<p v-else></p>                            语句（限兄弟元素使用）

<p v-show="toupflag"></p>               切换元素 display: none 样式（适合频繁的切换，能直接写 data 变量）
<input v-model="msg">                   仅用于表单元素双向数据绑定， 表单元素内部值和实例数据实现双向同步
<img v-bind:title="msg + '111' ">      <img :title="msg + '111' "  >                                双向数据绑定后可以解析字符串内变量， 解析为 js 代码-表达式
<img v-on:click="a('xx')">   <img v-on='{click: a, mouseleave: b}'>    <img @click="show">    @click="a(), b()"     绑定事件使用实例的方法    【函数加小括号可传参】
                        <img @click.once.stop ="show">        事件修饰符    // 前面功能的先实现
```

.stop 阻止冒泡 .prevent 阻止默认事件 .capture 使用事件捕获机制 .self 阻止自身事件被其他元素触发 .once 事件只触发一次 .native 监听原生事件

<input @keyup.enter="show"> 按键修饰符 // 监听指定按键

.113---直接填写键码 .enter .tab .delete .esc .space .up .down .left .right

<button @click="Eventfunc($event)"> </button> 使用 event 对象 event.path[0]== document event.path[1] ==window

<p v-for="val, ind in list" :key="ind">{{ val }} 遍历数组： 值 索引 【(val, ind) in list 可以加括号】【key 保存原数据的状态，表单在上面添加数据，勾选的盒子位置会不变】

<p v-for="val, key, ind in list" :key="ind">{{ val }} 遍历对象： 值 键 索引 【(val, key, ind) in list 可以加括号】【item in search(msg) 可直接访问 data 里的方法数据实现查询 】

识别样式

<img :class="['red', 'thin', flag?'classa':'']"> 可用三元表达式

<img :class="['red', 'thin', {'classa':flag}]"> 对象代替三元表达式

<img :class="classobj"> data:{ classobj: {a:true, d:true, c:false, } } 类名可省略引号

<img :style="styleobj"> data:{ styleobj: {color:'red','font-weight':200 } } 含 - 需加引号

<img :style="[styleobj1, styleobj2]"> data:{ styleobj1: {color:'red','font-weight':200 } } 可使用数组

动画

v-enter 进入开始状态 v-enter-to 进入结束状态 (v-enter-active 进入时间段)

v-leave 离开开始状态 v-leave-to 离开结束状态 (v-leave-active 离开时间段)

<transition name='tn' mode="out-in"> <transition> 包裹动画元素

.v-enter, .v-leave-to { opacity: 0; transform: translateX(150px); } 全局动画样式

.v-enter-active, .v-leave-active{ transition: all 0.4s ease; }

.v-move{ transition: all 0.6s ease; } 固定写法---实现列表后续元素渐渐飘上来的效果

.v-leave-actve {position: absolute; }

.tn-enter { opacity: 0; transform: translateX(100%); } .tn -leave-to { opacity: 0; transform: translateX(-100%); position: absolute} 指定动画样式（右侧进来左侧消失）

.tn-enter-active, .tn-leave-active{ transition: all 0.4s ease; }

<transition enter-active-class='animated bounceIn' leave-active-class='animated bounceOut' 进入离开样式

:duration="400" :duration="{enter:200, leave:400}" 进入离开时间---分别设置

@before-enter="beforeEnter" 设置开始样式 // beforeEnter(el 执行动画的原生 dom 元素){}

@enter="enter" 设置结束样式和过渡 // enter(el 执行动画的原生 dom 元素, done 无延迟调用 afterEnter){el.offsetTop 刷新动画 el.style.transition='' done(); }

@after-enter="afterEnter" 动画完成之后（this.flag=!this.flag 跳过后半场动画---flag 前半场状态变化对应前半场动画，不能执行后半场动画）

@enter-cancelled="enterCancelled"

@before-leave="beforeLeave" 设置离开样式

@leave="leave"

@after-leave="afterLeave"

@leave-cancelled="leaveCancelled">

 可直接给元素指定基本样式

</transition>

<transition-group appear 列表入场效果 tag="ul" 指定 transition-group 渲染成的元素---默认渲染为 span> v-for 渲染元素添加动画（必须添加 key 属性）

<p v-for="item in list" :key="item.id">{{ item }}

</transition-group>
<transition mode="out-in"> <component :is="'myLogin'"></component> </transition> 包裹切换组件动画
<transition mode="out-in"> <router-view> </router-view> </transition> 包裹路由占位符动画

第三方包

vue-resource

ajax 请求
Vue.http.options.root='http://www.baidu.com/' 设置请求根路径---配制后请求必须为相对路径
Vue.http.options.emulateJSON=true; 全局 options 配置---各个请求的 options
this.$http.get('url', [{options}]).then(function(res){}, [function(err){}]) this 为 methods 各个方法里的 this
post('url', [{body}], [{options}]) .then(function(res){}, [function(err){}]) 手动发送的 post 请求默认没有表单格式，有的服务器处理不了
options emulateJSON:true (以 application/x-www-form-urlenoded 表单默认格式发送请求体)
jsonp('url', [{options}]).then(function(res){}, [function(err){}])
res.body 获取返回数据

vue-router

官方前端路由（不利于 SEO，虚构界面不能被搜索，浏览器无法前进后退点击只会重新请求，单页面无法记住滚动位置,）
router > index.js

```
import Vue from 'vue'   import Router from 'vue-router'   import son from '@/views/son'   导入各个组件配置到路由实例中
Vue.use(Router)
const router= new Router(
{   mode: 'history',                                  路由模式默认为哈希（#开头），history 为直接用 url 地址
    routes: [ {  path:'/father',                                  拦截路由 （动态地址 /login:id:name',   使用<router-link to="/login/12/pna")
                 name: '店铺'  },                                 路由名-便于 js 跳转
                 component: son.                                  匹配组件
                children:[   {path:'son',  component: {template:'<>'}  ]    },      子路由（ < router-link to="/login/son"></router-link> )
                { path:'/login',   components: { 'default': head, 'left': leftBox, } },     多坑显示多组件 （<router-view/> <router-view name="left"/>  )
                { path:'/',          redirect: '/login' },                                  重定向到另一个路由
               ],
    linkActiveClass: 'myactive'              自定义 css 中 .router-link-active 的名称（router-link 激活时的样式）
 } )
router.beforeEach(( to, from, next )=>{. }).     访问路径时过滤路由【to 即将跳转的路由实例，next()直接跳转】
export default router
```

App.vue <router-link to="/login?id=9" tag="span" > xxx </router-link> 路由链接 【tag 指定渲染成的元素-含默认点击显示组件事件】
<router-link v-bind:to="{ name:'cart', params:{id:123} }"> 路由链接（根据路由名跳转）
<router-view/> 路由坑（根据路由链接显示相应的组件）
.router-link-active{} 路由链接激活样式 <router-link class='router-link-active' >
js 使用路由 (this.$route 参数对象 this.router 导航对象-路由前进后退跳转)
this.$router.push({ name:'user', params:{id} }) 根据路由名跳转
this.$router.push('/home/goodsinfo') 直接跳转
this.$router.push({path: '/backend/order', query: {selected: "2"}})
this.$router.go(1) 前进后退

this.$router.options.routes　　router 示例的内部参数

this.$route.path　　请求路径　【 {{ $route.query.id }}　页面模板中直接使用　】

this.$route.query　　请求参数的键值对对象

this.$route.params.id　　动态地址:id 匹配到的数据

xx.vue　　< router-link to="/login/son"></router-link>　　点击显示子路由组件

<router-view></router-view>　　子路由坑

main.js　　import router from './router'

new Vue({ el: '#app', **router**, components: { App }, template: '<App/>' })

vuex

公共数据管理工具（共享数据保存到 vuex 中，各个组件都可以获取和修改其内的数据）

store.js　　import Vue form 'vue'

import Vuex form 'vuex'　　Vue.use(Vuex)

export default new Vuex.Store({ **state:**{a:0},　　保存数据

mutations:{ funa(state){state.a++} } 保存函数 操作 state 里的数据

mutations 里的函数最多支持两个形参

fun**(**state, {a:3, b:4} **){}**　　（state 对象和 commit 提交过来的数据）

getters:{ funa(state){ return 'xxxx'+ state.a; } }　对外提供数据不修改数据　})

创建完将对象挂载在 vm 根实例上　store: store,

actions => 像一个装饰器，包裹 mutations，使之可以异步。

modules => 模块化 Vuex

main.js　　import Vue from 'vue'

import store from './vuex/store'

new Vue({el: '#app', router, **store,** })　　挂载仓库

index.vue　　this.$store.state.count 调用仓库数据　this.$store.commit('fun', aa) 调用仓库方法　　组件 methods 内部函数里访问 vuex 全局仓库（不推荐组件中单独定义方法操作仓库数据 ）

<input v-model="$store.state.count ">

state => 基本数据(数据源存放地)

getters => 从基本数据派生出来的数据

mutations => 提交更改数据的方法，同步！

actions => 像一个装饰器，包裹 mutations，使之可以异步。

modules => 模块化 Vuex

postcss-pxtorem
lib-flexible

main.js > import 'lib-flexible' 自动配置移动端 meta 和设置适配各种设备屏幕的 rem（需要将 index.html 中的<meta name='viewport'>删除）

新建 postcss.config.js >

```
module.exports = {
  plugins: {  'autoprefixer': {     browsers: ['Android >= 4.0', 'iOS >= 7']   },
            'postcss-pxtorem': {   rootValue: 37.5,//     propList: ['*']   /* 设置哪些属性可以从 px 变为 rem。“!”表示不匹配，
  “ !font* ” 表示不匹配字体相关属性 */ } }
        //适配 375 屏幕, 设计图 750 中量出来的尺寸要/2 ， 配置成 37.5 是为了兼容没有适配 rem 布局的第三方 ui 库
```

页面上直接使用 px ， 会自动转换为 rem（在宽度 375 的屏幕上是 1:1 比例）

vant

Icon　　icon="setting-0" 或" https://b.yzcdn.cn/vant/icon-demo-1126.png"

TabBar　　　　icon.cat_inactive 可修改成对应的激活图片

Swipe　　data(){}　　images:[require('@/assets/s1.jpg'),]　　vant 框架内图片路径需要加上 require

　标签后直接加样式不会被 postcss-pxtorem 适配修改成 rem（给 my-swipe 添加背景图片会出错）

不适配样式:　　 行内样式　　background: url("a.png") no-repeat 15px 150px/20px; 背景图片大小

路由跳转图片消失:　　van-swipe-item 有默认背景色，定位浮动 z-index 为 0 时被后方覆盖　---->　background-color: transparent

覆盖样式　　在 App.vue 中定义全局样式覆盖

其他

axios.js

moment.js　格式化时间插件　　moment(datastr).format(pattern)

vue-preview　图片缩略图插件

Weex　　迁移到移动端 App 开发

Node / @types/react

export = React;

export as namespace React;

declare namespace React

FC

type **FC**<P = {}> = FunctionComponent<P>;

StrictMode

const **StrictMode**: ExoticComponent<{ children?: ReactNode | undefined }>;

lets you find common bugs in your components early during development.

Use StrictMode to enable additional development behaviors and warnings for the component tree inside

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'));
root.render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

useState

function useState<S>(initialState: S | (() => S)): [S, Dispatch<SetStateAction<S>>];

useState is a React Hook that lets you add a state variable to your component.

```
import { useState } from 'react';

function MyComponent() {
  const [age, setAge] = useState(28);
  const [name, setName] = useState('Taylor');
  const [todos, setTodos] = useState(() => createTodos());
  // ...
```

useEffect

function **useEffect**(effect: EffectCallback, deps?: DependencyList): void;

useEffect is a React Hook that lets you synchronize a component with an external system.

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

createRef

function **createRef**<T>(): RefObject<T>;

Call createRef to declare a ref inside a class component.

```
import { Component, createRef } from 'react';

export default class Form extends Component {
  inputRef = createRef();

  handleClick = () => {
    this.inputRef.current.focus();
  }

  render() {
    return (
      <>
        <input ref={this.inputRef} />
        <button onClick={this.handleClick}>
          Focus the input
        </button>
      </>
    );
  }
}
```

useRef

function **useRef**<T>(initialValue: T): MutableRefObject<T>;

useRef is a React Hook that lets you reference a value that's not needed for rendering.

```
import { useRef } from 'react';

export default function Counter() {
  let ref = useRef(0);

  function handleClick() {
    ref.current = ref.current + 1;
    alert('You clicked ' + ref.current + ' times!');
  }

  return (
    <button onClick={handleClick}>
      Click me!
    </button>
  );
}


import React from 'react';
```

```
import PropTypes from 'prop-types';
const Canvas = ({draw, height, width}) => {
  const canvas = React.useRef();
  React.useEffect(() => {
    const context = canvas.current.getContext('2d');
    draw(context);
  });
return (
    <canvas ref={canvas} height={height} width={width} />
  );
};
Canvas.propTypes = {
  draw: PropTypes.func.isRequired,
  height: PropTypes.number.isRequired,
  width: PropTypes.number.isRequired,
};
export default Canvas;
```

createContext

function createContext<T>(
// If you thought this should be optional, see
// https://github.com/DefinitelyTyped/DefinitelyTyped/pull/24509#issuecomment-382213106
defaultValue: T,
): Context<T>;

When using the props.child, It is used to pass some parameters to child components.
The use of context is usually not visible in external components that use BasicLayout components.

```
import { createContext } from 'react';
const ThemeContext = createContext('light');
```

useContext

function useContext<T>(context: Context<T>/*, (not public API) observedBits?: number|boolean */): T;

useContext is a React Hook that lets you read and subscribe to context from your component.

```
function Button() {
  const theme = useContext(ThemeContext);
  return <button className={theme} />;
}
```

Suspense

const **Suspense**: ExoticComponent<SuspenseProps>;

<Suspense> lets you display a fallback until its children have finished loading.

```
<Suspense fallback={<Loading />}>
  <SomeComponent />
</Suspense>
```

Node / typescript/lib

HTMLCanvasElement

interface **HTMLCanvasElement** extends HTMLElement You can get “canvas” tag directly like “document” tag

width: number; Canvas element width
height: number; Canvas element height
getContext(contextId: "2d", options?: CanvasRenderingContext2DSettings): CanvasRenderingContext2D | null;
getContext(contextId: "bitmaprenderer", options?: ImageBitmapRenderingContextSettings): ImageBitmapRenderingContext | null;
getContext(contextId: "webgl", options?: WebGLContextAttributes): WebGLRenderingContext | null;

getContext(contextId: "webgl2", options?: WebGLContextAttributes): WebGL2RenderingContext | null;
getContext(contextId: string, options?: any): RenderingContext | null;
toDataURL(
type?: string,
quality?: any A Number between 0 and 1 indicating the image quality to be used when creating images using file formats that support lossy compression (such as image/jpeg or image/webp).
): string; Returns the content of the current canvas as an image base64 URL

const fullQuality = canvas.toDataURL("image/jpeg", 1.0);
DataUrl: .....

CanvasRenderingContext2D

interface **CanvasRenderingContext2D** extends CanvasCompositing, CanvasDrawImage,
CanvasDrawPath, CanvasFillStrokeStyles, CanvasFilters, CanvasImageData, CanvasImageSmoothing,
CanvasPath, CanvasPathDrawingStyles, CanvasRect, CanvasShadowStyles, CanvasState, CanvasText,
CanvasTextDrawingStyles, CanvasTransform, CanvasUserInterface {
readonly canvas: HTMLCanvasElement;
getContextAttributes(): CanvasRenderingContext2DSettings;
}

CanvasImageData

getImageData(sx: number, sy: number, sw: number, sh: number, settings?: ImageDataSettings): ImageData;

CanvasPath

closePath(): void;
moveTo(x: number, y: number): void; Move the brush to the target coordinates.
lineTo(x: number, y: number): void; Draw a straight path with x, y as the destination coordinates

CanvasDrawPath

interface **CanvasDrawPath**
beginPath(): void; Open a new path (it needs to be stroked again later)
stroke(): void; Stroke the path (both tracing and filling can be done simultaneously)
stroke(path: Path2D): void;
fill(fillRule?: CanvasFillRule): void;
fill(path: Path2D, fillRule?: CanvasFillRule): void;

Node / Configuration

package.json

"**name**": "testvue",
The unique name for your package.
"**author**": "dd",
The author of the project.
"**version**": "1.0.0",
Defines the version of the package in the format major.minor.patch (e.g., 1.0.0). Follows Semantic Versioning.
"**description**": "ff",
A brief description of what the package does.
"**license**": "ISC",

Specifies the licensing for the project. Common licenses include "MIT", "Apache-2.0", and "GPL-3.0".

"private": true,　　禁止 npm 发布（npm publish）

"main": "./index.js",

Defines the entry point of the application or module (usually the main file, like index.js).

When someone imports your package, this file will be the default import.

"scripts":

A set of custom commands to automate tasks. You can define scripts like start, test, build, and more。

You can run scripts with npm run <script-name>.

{

"dev": "webpack-dev-server --port 80 -process",　　（可以 set PORT=3033 && webpack 使用 windows 脚本语法定义临时环境变量）

"build": "webpack --config ./config/webpack.config.js"

},

"dependencies":

Lists all the packages required for your project to run.

Dependencies are installed using npm install <package-name> and are essential for the application at runtime.

They appear as key-value pairs, where the key is the package name, and the value specifies the version (e.g., "express": "^4.17.1").

【~2.1.2:　安装 2.1.2 以上的版本，但是不安装 2.2.0　　^2.1.2:　安装 2.1.2 以上的版本，但是不安装 3.0.0　】

{

"css-loader": "^5.0.2",

}

"devDependencies": {　　本地依赖包，不会被打包

"autoprefixer": "^7.1.2",

"babel-core": "^6.22.1",

}

"overrides": {　　确保软件包 foo 始终安装为 1.0.0 版本，无论您的依赖项依赖什么版本

"foo": "1.0.0"

}

"repository" :

Defines where the source code is hosted, useful for linking to the code repository:

{

"type" : "git",

"url" : "http://github.com/saidake/testvue.git"

},

"bin": { "lan": "bin/lan" }

Defines executable files, useful when creating a CLI tool in the path "node_modules/.bin/"

"homepage": "https://github.com/saidake/dongfeng-create#readme",　发布包时的主页链接

"keywords": ["dongfeng-create"],

An array of keywords related to the project, useful for searchability on npm if the package is published.

"files": ["src", "dist/*.js", "types/*.d.ts"],

An array of files to include when publishing a package.

"config": {　　设置一些用于 npm 包的脚本命令会用到的配置参数

"commitizen": { "path": "./node_modules/cz-conventional-changelog" }

}

```
"bugs": {  "url": "https://github.com/vuejs/vue/issues"  },      bug 地址
"gitHooks": {                                                    代码质量检查
    "pre-commit": "lint-staged",
    "commit-msg": "node scripts/verify-commit-msg.js"
},
"lint-staged": {   "*.js": [  "eslint --fix",  "git add" ]    },          代码检查

"browserslist": [                浏览器兼容设置---Array
    "> 1%",                  兼容全球超过 1%人使用的浏览器
    "last 2 versions",       所有浏览器兼容到最新的两个版本
    "not ie <= 8"            不兼容的版本
],
"engines":
    Specifies compatible Node or Npm versions for the project:
{
    "node": ">= 6.0.0",
    "npm": ">= 3.0.0"
},
"os" : [ "darwin", "linux", "!win32" ]        指定模块运行的操作系统---Array（！为黑名单）
"cpu" : [ "x64", "ia32", "!arm", "!mips" ]    指定模块运行的 cpu 架构---Array（！为黑名单）
```

webpack.config.js

ERROR　　dev-server.js　　最新版本的 vue-cli 已经放弃 dev-server.js，只需在 webpack.dev.conf.js 配置就行

```
const path = require('path');
const { VueLoaderPlugin } = require('vue-loader');              加载 vue-loader 的插件
```

———————————————————————————— 基础

```
module.exports = {
  mode: "production",                          打包模式   ("production"  "development"  "none")
  context: path.resolve(__dirname, '../'),     项目根目录   (entry 和 module.rules.loader 选项都相对于此目录解析)
  entry: './src/main.js'                       打包入口文件  (entry 有三种定义方式)
  entry: [    './js/base.js',          './js/app.js'          ]                从前往后执行，生成一个代码块
  entry: {   app: './src/main.js',     hello: './src/hello.js',   ddd:'jquery'  },     入口文件，可生成多个 js 代码块和第三方代码块---chunk  (导入文件依赖的所有模块，生成 chunk 代码块)
  output: {
    path:path.resolve(__dirname, "dist"),          输出文件目录（必须是绝对路径）
    filename: 'js/[name][hash:8].js',              编译生成的 js 文件存放到根目录下面的 js 目录下面,如果 js 目录不存在则自动创建（必须是绝对路径）
    chunkFilename:  js/[name].[chunkhash].js       非入口 chunk 文件的名称
    publicPath: process.env.NODE_ENV === 'production' ? config.build.assetsPublicPath: config.dev.assetsPublicPath   生产模式时更新 css、html 文件里的 url 值
                                                【publicPath:'https://someCDN/'        background-image: url('./test.png'); --> background-image: url('https://someCDN/test.png'); 】
  }
```

———————————————————————————— 模块

```
  resolve: {
    extensions: ['.js', '.vue', '.json'],                导入时可以省略的后缀
    alias: {
      'vue$': 'vue/dist/vue.esm.js',
      '@': resolve('src'),
    }
  },
  module: {
    rules: [
      {
        test: /\.vue$/,
        use:[ "style-loader", "css-loader" ],     匹配文件后从后向前加载解析器
        enforce: 'pre'                                  匹配到相同的 test 时，优先加载当前解析器
        exclude: /node_modules/,                    不匹配 node_modules 下的文件
        include: [resolve('src'), resolve('test'), resolve('node_modules/webpack-dev-server/client')],      仅匹配 src 目录 下的 js 文件   【 babel-loader   (/\.js$/,) 】
        options: vueLoaderConfig                  解析器选项  【 vue-loader 】
        options: {
          limit: 10000,                                小于 10k 时自动处理为 base64 图片
          name: utils.assetsPath('img/[name].[hash:7].[ext]')          【 url-loader ( /\.(png|jpe?g|gif|svg)(\?.*)?$/, )  】
          name: utils.assetsPath('media/[name].[hash:7].[ext]')      【   url-loader (/\.(mp4|webm|ogg|mp3|wav|flac|aac)(\?.*)?$/, )    】
          name: utils.assetsPath('fonts/[name].[hash:7].[ext]')        【   url-loader  (/\.(woff2?|eot|ttf|otf)(\?.*)?$/,)    】
        },
        use: [{                                  use 内匹配文件后从后向前加载解析器
          loader: "style-loader"
        }, {
          loader: "css-loader"
        }, {
          loader: "less-loader",
          options: {                          【   /\.less$/,  】
            strictMath: true,
            noIeCompat: true
          }
        }]
      },
      { oneOf: [ { test: /\.css/,   loader: 'url-loader', },  { test: /\.css/,  use:[{ loader:'file-loader'}, {loader:'css-loader'} ]   } ]},      oneOf 内的加载器只会匹配一个，一般单独用
    ]
  },
  externals: {              防止将某些 import 的包(package)打包到 bundle 中，而是在运行时(runtime)再去从外部网址获取这些扩展依赖(external dependencies)。
    jquery: 'jQuery',
  },
```

工具

```
devServer: {              配置 webpack-dev-server
    contentBase: path.join(__dirname, 'dist'),      静态资源目录
    compress: true,          是否压缩
    host: 'localhost'        服务器 IP 地址
    port: 9000,              端口
    open:true                自动打开页面
    before(app) {
      apiMocker(app, mockFilesPathArr)
    },
stats: {                 “minimal”          关闭输出提示
    colors: false,
    hash: false,
    version: false,
    timings: false,
    assets: false,
    chunks: false,       关于 chunk 的信息
    modules: false,    是否添加关于构建模块的信息。[built]
   times:false,
   builtAt:false,
    reasons: false,
    children: false,
    source: false,
    errors: true,
    errorDetails: true,
    warnings: false,
    publicPath: false
  }
  },
 devtool: 'cheap-module-eval-source-map'     提高编译效率
  plugins:[                插件选项---Array
 new webpack.ProvidePlugin({       自动向使用此变量的模块内导入 jquery
       $:'jquery'
 }),
  new VueLoaderPlugin()
  ],
node: {
  setImmediate: false,
  dgram: 'empty',
  fs: 'empty',
  net: 'empty',
  tls: 'empty',
  child_process: 'empty'
}
 optimization: {
     splitChunks: {
         chunks: 'async',  'all'    'initial'              //默认只作用于异步模块，为`all`时对所有模块生效,`initial`对同步模块有效
         minSize: 30000,                                  //合并前模块文件的体积
```

```
            minChunks: 1,                           //最少被引用次数
            maxAsyncRequests: 5,
            maxInitialRequests: 3,
            automaticNameDelimiter: "~",
            cacheGroups: {
                vendors: {
                    test: /node_modules/,
                    minChunks: 1,
                    priority: -10  },
                default: {
                     test: /src/,
                    minChunks: 2,
                    priority: -20,
                    reuseExistingChunk: true }
            }
}
  }
```

tsconfig.json

TypeScript Documentation: https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

```
{
    "compilerOptions": {
        "target": "esnext",       Compilation ES version
                ECMAScript：ECMA script language standard
                Ecma has been named by year since 2016, while older versions of es were named by numbers, such as ES5 (2009) and ES6 (2015), but can still be named by numbers
                        es2009 Or es5    June 2009
                        es2015 Or es6    June 2015
                        es2016 Or es7    June 2016
                        es2017 Or es8    June 2017
                        es2018 Or es9    June 2018
                        es2019 Or es10  June 2019
                        es2020 Or es11  June 2020
                        es2021 Or es12  June 2021
                        es2022 Or es13  June 2022
                        esnext              latest next version
        "module": "esnext",     Module Standard 【'none', 'commonjs', 'amd', 'system', 'umd', 'es6', 'es2015', 'es2020', 'es2022', 'esnext', 'node16', 'nodenext'】
        "lib": ["DOM"],         lib 用来指定项目中使用的语法库
            //  'es5', 'es6', 'es2015', 'es7', 'es2016', 'es2017', 'es2018',
            // 'es2019', 'es2020', 'es2021', 'es2022', 'esnext', 'dom', 'dom.iterable', 'webworker',
            //  'webworker.importscripts', 'webworker.iterable', 'scripthost', 'es2015.core', 'es2015.collection',
            // 'es2015.generator', 'es2015.iterable', 'es2015.promise', 'es2015.proxy', 'es2015.reflect', 'es2015.symbol',
            //  'es2015.symbol.wellknown', 'es2016.array.include', 'es2017.object', 'es2017.sharedmemory', 'es2017.string',
            //  'es2017.intl', 'es2017.typedarrays', 'es2018.asyncgenerator', 'es2018.asynciterable', 'es2018.intl',
            // 'es2018.promise', 'es2018.regexp', 'es2019.array', 'es2019.object', 'es2019.string', 'es2019.symbol',
            // 'es2020.bigint', 'es2020.date', 'es2020.promise', 'es2020.sharedmemory', 'es2020.string',
      'es2020.symbol.wellknown',
```

```
        //  'es2020.intl', 'es2020.number', 'es2021.promise', 'es2021.string', 'es2021.weakref', 'es2021.intl', 'es2022.array',
        // 'es2022.error', 'es2022.intl', 'es2022.object', 'es2022.sharedmemory', 'es2022.string', 'esnext.array',
  'esnext.symbol',
        // 'esnext.asynciterable', 'esnext.intl', 'esnext.bigint', 'esnext.string', 'esnext.promise', 'esnext.weakref'.
    "outDir": "./dist",                  指定编译后所存放的目录文件
    "outFile": "./dist/content.js",      outfile 用来将全局作用域代码进行合并
    "allowJs": true,                     是否编译 js 文件，默认为 false
    "checkJs": false,                    是否检验 js 代码符合 ts 语法规范
    "removeComments": true,      是否移除文本注释
    "noEmit": false,                     不生成编译后的文件
    "noEmitOnError": true,     当语法有错误时不生成编译文件
    "skipLibCheck": true,          跳过 lib 库检查
    "allowSyntheticDefaultImports": true,   当没有 default 导出时，允许合成默认导入   // import * as React from "react";
    "moduleResolution": "node",              用 node 解析
    "importHelpers": true,                   开启后, 不在插入具体的辅助方法的代码到对应的位置，而是通过模块导入来引用
    typescript 的辅助方法
    "esModuleInterop": true,         通过导入内容创建命名空间，实现 CommonJS 和 ES 模块之间的互操作性
    "sourceMap": true,               socuceMap 用来指定编译时是否生成.map 文件， 主要用于调试
      “isolatedModules” : false,        如果某个 ts 文件中没有一个 import or export 时，ts 则认为这个模块不是一个 ES
    Module 模块，它被认为是一个全局的脚本，如果是 ts 文件会报错
    "jsx": "react-jsx",                      TS 的三种 JSX 模式，这些模式只在代码生成阶段起作用，类型检查并不受影响
    【preserve，react，react-native】
                                                 preserve:      在 preserve 模式下生成代码中会保留 JSX 以供后续的转换操
                                                 作使用，输出文件会带有.jsx 扩展名
                                                 react:          react 模式会生成 React.createElement，在使用前不需要再
                                                 进行转换操作了，输出文件的扩展名为.js
                                                 react-native：react-native 相当于 preserve，它也保留了所有的 JSX，但
                                                 是输出文件的扩展名是.js

    "strict": true,              启用所有严格检查（noImplicitAny，noImplicitThis, alwaysStrict， strictNullChecks，
    strictFunctionTypes， strictPropertyInitialization）
    "alwaysStrict": true,      设置编译后的文件是否使用严格模式
    "noImplicitAny": true,         不允许使用隐式 any，any 等同于关闭了 ts 数据类型检测，所以不推荐使用 any
    "noImplicitThis": true,       不允许不明确类型的 this
    "strictNullChecks": true,                    严格检查空值

    "noUnusedLocals": true,                  用于检查是否有定义了但是没有使用变量 false 不检查
    "noUnusedParameters": true,              用于检测是否在函数中没有使用的参数 false 默认不检查
    "noFallthroughCasesInSwitch": true,      用于检查 switch 中是否有 case 没有使用 break 跳出 switch，默认为 false
    "forceConsistentCasingInFileNames": true     引入文件名必须大小写一致
}

"baseUrl": ".",
"paths": {                          定义路径别名（必须先定义 baseUrl，paths 都是相对 baseUrl 的路径）
    "@/*": ["src/*"],               路径相对于当前配置文件 tsconfig.json
    "@@/*": ["src/.umi/*"]
},
```

"typeRoots": [类型根路径
"./node_modules/@types"
],
"include": ["src/**/*", "tests/**/*"] Specifies an array of filenames or patterns to include in the program. These filenames are resolved relative to the directory containing the tsconfig.json file.
}

Node / Npm Package

Build Project

css

style-loader 可以将 css 文件变成 style 标签插入 head 中（放在最后）
css-loader 打包时转换 css 文件中的 url 路径为正确的路径，并将 css 文件变成一个模块
sass-loade node-sass 编译 sass

webpack-bundle-analyzer

打包分析工具
vue> vue.config.js npm run serve

html-webpack-plugin

在 html 源文件上 使用 script 引入最新打包后的 js 文件，然后在新目录生成新的 html 文件（搭配 webpack-dev-server）
plugins:[
new HtmlWebpackPlugin({
title: 'My App', 页面标题
template: "./src/index.html", 由此源文件生成 HTML
filename: 'index.html' 生成的 HTML 文件名，可以指定目录（"./dist/index.html"）
hash:ture, 在 js 文件里添加查询字符串避免缓存
inject: "body" script 标签插入的位置 true "body" 插入 body 底部 "head" 插入 head 标签底部 false 不插入
chunks: ["common",'index',"ddd"] 指定引入的代码块，可以导入第三方 ddd 代码块（webpack.config.js 中 entry 为对象时，生成的多个 js 代码块）
minify:{ removeAttributeQuotes:true } 移除 js 文件 src 内的引号
})]
<title><%= htmlWebpackPlugin.options.title %></title> 在源文件中使用自定义属性

clean-webpack-plugin

打包生成新的 js 文件时，会将之前的未使用 js 文件删除
plugins:[new CleanWebpackPlugin()]

blueimp-md5

提供 md5 加密函数 //body.password=md5(md5(password+'itscast'))

commander

The commander module is a popular library in Node.js for building command-line interfaces (CLI) with options and arguments.

Install commander

npm install commander

Setting up a Simple CLI Command

Use the command and description methods to define the command. Let's add a greet command that accepts a name argument:

```
#!/usr/bin/env node
```

This is known as a "shebang" line. It tells the operating system which interpreter to use to execute the file.

```
"use strict"
```

This line enables "strict mode" in JavaScript, which is a way to opt in to a restricted variant of JavaScript.

It helps catch common coding mistakes and "unsafe" actions such as:

```
const { Command } = require('commander');
const program = new Command();
program
  .name('my-cli')
  .description('A simple CLI application')
  .version('1.0.0')
  .option('-d, --debug', 'Enable debug mode');

program
  .command('greet <name>')
  .description('Greet a person by name')
  .option('-e, --excited', 'Make the greeting excited')
  .action((name, options) => {
    if (program.opts().debug) console.log('Debug mode enabled');
    const greeting = options.excited ? `Hello, ${name}!!!` : `Hello, ${name}.`;
    console.log(greeting);
  });
program
    .command('info')
    .description('Provide information about a person')
    .arguments("<name> [age]") // Required argument `name` and optional argument `age`
    .option('-e, --excited', 'Make the greeting excited') // Option
    .action((name, age, options) => {
        // Use the name and age provided by the user
        let response = `Name: ${name}`;
        if (age) {
            response += `, Age: ${age}`;
        }
        if (options.excited) {
            response += '!!!';
        }
        console.log(response);
    });


program.command('build', 'Build the project', { executableFile: './build' });
```

In this example, if the user runs my-cli build, it will execute the logic found in the ./build-cli script.

The executableFile option cannot be added directly as a parameter in the same way as the description for a command.

```
program.parse(process.argv);
```

Processes the arguments provided when the script is run.

commander

```
var program = require('commander');

program
  .version('0.0.1', '-v, --vers', 'output the current version')  版本选项
  .command('update <sonexearg>', 'update installed packages', { executableFile: 'myexe'})     子命令，自动执行 myexe.js 文件内容
   .arguments('<command> [options]')                                                          命令参数描述（--help 显示 <xxx> 表示一个必选项 xxx 是描述， ）
  .description('xxx')                                                                         命令描述（--help 显示）
  .alias('i')
  .option('-d, --ddd --no-ddd <p1> [p2]', 'description', 'defaultvalue')      <>为必填项    []为可填项 （--no--xxx 为不加此项为 true）
  .action((sonexearg, cmdobj) => {                                           子命令参数  和 cmd 对象（cmd 可以获取 option 的
```

值：cmdobj.ddd）
    console.log('Hello World %s', source)　　支持占位符
  })
  . allowUnknownOption()　　阻止缺失参数时 的默认错误，手动添加参数验证。
  .addHelpText('after', 'xxx');　　自定义帮助信息，在--help 最后方显示 xxx
  .on('--help',function(){ })　　监听--help 传入事件
  .on('options:info',function(){ })　　监听--help 传入事件
  .parse(process.argv);　　结尾
program.opts()　　所有的 options
program.ddd　　访问 option 内填写的值，没有为 undefined（只有加了<> 或 []才有值）

download-git-repo

download('github:saidake/test#master', dir , {clone:true}, function (err) {　　git 路径，载入路径，回调函数
  console.log(err ? 'Error' : 'Success')
})

validate-npm-package-name

 validate("123numeric")

rimraf

rimraf.sync(dir,{})　删除文件夹

fs-extra

fs.ensureDirSync(projectName);　创建目录（存在就不创建，返回 undefined，成功返回 D:\Desktop\DevWeb\lef\dongfeng）

chalk

console.log(chalk.green('success') chalk.red('success'))

ora

loading 插件
ora.start('xx')
ora.stop()
ora.succeed('xxx')
ora.fail('xxx')

semver

semver.valid('1.2.3') // '1.2.3'
semver.valid('a.b.c') // null
semver.clean(' =v1.2.3 ') // '1.2.3'
semver.satisfies('1.2.3', '1.x || >=2.5.0 || 5.0.0 - 7.2.3') // true
semver.gt('1.2.3', '9.8.7') // false
semver.lt('1.2.3', '9.8.7') // true
semver.minVersion('>=1.0.0') // '1.0.0'
semver.valid(semver.coerce('v2')) // '2.0.0'
semver.valid(semver.coerce('42.6.7.9.3-alpha')) // '42.6.7'

fs-extra

异步:　emptydir()　　清空目录，没有则新建
同步:　emptyDirSync(), emptydirSync()

envinfo

envinfo　　输出指定系统环境信息
    .run(
     {
       System: ["OS", "CPU"],
       Binaries: ["Node", "npm", "Yarn"],

```
        Browsers: [
          "Chrome",
          "Edge",
          "Internet Explorer",
          "Firefox",
          "Safari",
        ],
        // npmPackages: ['react', 'react-dom', 'react-scripts'],
        // npmGlobalPackages: ['lef'],
      },
      {
        duplicates: true,
        showNotFound: true,
      }
    )
    .then(console.log);
```

cross-spawn

执行 spawn 命令（在使用 spawn 在 windows 上会出问题）

```
spawn.sync('npm',['config','list']).output.join('');      获取命令输出
var vvv=spawn(command, args, { stdio: 'inherit' });
 vvv.stdout.on('data', (data) => {
     console.log(`stdout: ${data}`);
   });
   vvv.stderr.on('data', (data) => {
     console.error(`stderr: ${data}`);
   });
   vvv.on('close', (code) => {
     console.log(`子进程退出，退出码 ${code}`);
   });
```

Inquirer

```
let inquirer = require('inquirer')
inquirer.prompt([
  {
    type: 'confirm',
    name: 'handsome',
    message: '我是世界上最帅的男人吗?',
    default: true
  }
]).then((answers) => {
  console.log(answers)
})
```

Charts

amcharts

Stock

root

As with any chart type in amCharts 5, we'll need to start with creation of the Root element.

```
    let root = am5.Root.new("chartdiv");
```

```
let stockChart = root.container.children.push(
  am5stock.StockChart.new(root, {})
);
```

panel

A "panel" in a stock chart is an instance of a StockPanel which in turn extends XYChart pushed into panels list.

```
let mainPanel = stockChart.panels.push(am5stock.StockPanel.new(root, {
  wheelY: "zoomX",
  panX: true,
  panY: true
}));
```

axes

Since an XY chart requires at least two axes (X and Y) to function, we'll need to add those to panel's (chart), too.

```
let valueAxis = mainPanel.yAxes.push(am5xy.ValueAxis.new(root, {
  renderer: am5xy.AxisRendererY.new(root, {})
}));

let dateAxis = mainPanel.xAxes.push(am5xy.DateAxis.new(root, {
  baseInterval: {
    timeUnit: "day",
    count: 1
  },
  renderer: am5xy.AxisRendererX.new(root, {})
}));
```

Series

Series in stock chart are added to series list of its panels.

```
let valueSeries = mainPanel.series.push(am5xy.LineSeries.new(root, {
  name: "STCK",
  valueXField: "Date",
  valueYField: "Close",
  xAxis: dateAxis,
  yAxis: valueAxis,
}));

valueSeries.data.setAll(data);
```

Stock chart goes beyond regular XY charts by providing a lot of additional analytical tools like indicators, trend drawing, as well as comparisons of different indexes.

Those rely on data from main series.

Stock chart recognizes two main series: value and volume.

Some indicators and tools just use "value main series", others rely on "volume main series", while some need both.

We can use stock chart's settings valueSeries and volumeSeries to set which series to use for those analytical tools.

```
stockChart.set("stockSeries", valueSeries);
stockChart.set("volumeSeries", volumeSeries);
```

In case we don't need it displayed, we can create a hidden volume series:

```
let volumeValueAxis = mainPanel.yAxes.push(am5xy.ValueAxis.new(root, {
forceHidden: true,
renderer: am5xy.AxisRendererY.new(root, {})
}));
volumeValueAxis.get("renderer").grid.template.set("forceHidden", true);
volumeValueAxis.get("renderer").labels.template.set("forceHidden", true);
let volumeSeries = mainPanel.series.push(am5xy.ColumnSeries.new(root, {
valueXField: "Date",
valueYField: "Volume",
xAxis: dateAxis,
```

```
yAxis: volumeValueAxis,
forceHidden: true
}));
stockChart.set("volumeSeries", volumeSeries);
```

Positive/negative colors

Main value series (set on stockSeries) and main volume series (set on volumeSeries) will use two colors if they are of type column, candlestick, or OHLC.

They will use "negative" and "positive" colors from the default interface color set.

Negative color will be used for items that have their close value lower than their open.

Positive color will be used for the rest of the series items.

```
let stockChart = root.container.children.push(
  am5stock.StockChart.new(root, {
    stockPositiveColor: am5.color(0x999999),
    stockNegativeColor: am5.color(0x000000),
    volumePositiveColor: am5.color(0x999999),
    volumeNegativeColor: am5.color(0x000000)
  })
);
```

To completely remove built-in coloring, you can use null in place of the actual color:

```
let stockChart = root.container.children.push(
  am5stock.StockChart.new(root, {
    stockPositiveColor: null,
    stockNegativeColor: null,
    volumePositiveColor: null,
    volumeNegativeColor: null
  })
);
```

Legend

In stock chart, we can use both the regular legend that we'd use in an XY chart, as well as advanced special "sock legend". The latter offers enhanced view as well as tools to edit related series/indicators.

```
let valueLegend = mainPanel.plotContainer.children.push(am5stock.StockLegend.new(root, {
  stockChart: stockChart
}));
valueLegend.data.setAll([valueSeries]);
```

The important difference from regular legend is that stock legend is designed to be displayed over the plot area, so we push it into main panel's plotContainer.

Another difference is that because of additional functionality, stock legend requires to know what its stock chart it, hence us setting stockChart in the above code.

If we wanted we could use a regular legend here as well:

```
let valueLegend = valueAxis.axisHeader.children.push(am5.Legend.new(root, {}));
valueLegend.data.setAll([valueSeries]);
```

Cursors

Cursors work the same way as in XY charts: by creating an instance of XYCursor and supplying it to panel's (chart's) cursor setting.

```
mainPanel.set("cursor", am5xy.XYCursor.new(root, {
  yAxis: valueAxis,
  xAxis: dateAxis
}));
```

Scrollbar

Adding a scrollbar is identical how we would do it in an XY chart, except for pushing it into chart/panel's children, we use a special container on a stock chart: toolsContainer:

```
let scrollbar = mainPanel.set("scrollbarX", am5xy.XYChartScrollbar.new(root, {
  orientation: "horizontal",
  height: 50
}));
```

```
stockChart.toolsContainer.children.push(scrollbar);

let sbDateAxis = scrollbar.chart.xAxes.push(am5xy.GaplessDateAxis.new(root, {
  baseInterval: {
    timeUnit: "day",
    count: 1
  },
  renderer: am5xy.AxisRendererX.new(root, {})
}));

let sbValueAxis = scrollbar.chart.yAxes.push(am5xy.ValueAxis.new(root, {
  renderer: am5xy.AxisRendererY.new(root, {})
}));

let sbSeries = scrollbar.chart.series.push(am5xy.LineSeries.new(root, {
  valueYField: "Close",
  valueXField: "Date",
  xAxis: sbDateAxis,
  yAxis: sbValueAxis
}));

sbSeries.fills.template.setAll({
  visible: true,
  fillOpacity: 0.3
});

sbSeries.data.setAll(data);
```

If we'd like to place the scrollbar at the bottom of the panel instead, we can push it into panel's bottomAxesContainer instead:

```
mainPanel.bottomAxesContainer.children.push(scrollbar);
```

Cli Utils
umi

介绍：封装 react react-router-dom 的框架

依赖：yarn create @umijs/umi-app   在当前目录下创建项目，不会新建目录直接生成文件（无需安装@umijs/create-umi-app）

@types/node   node 模块定义

目录：config/config.ts   config/router.config.ts   删除.umirc.ts   新增 config 文件

插件：@umijs/plugin-sass   支持 sass（本地安装@umijs/plugin-sass，typings.d.ts 添加 scss 文件模块）

图标：public/favicon.ico

目录结构——

config

    config.ts   配置项（.umirc 优先级高， 需要先删除）

    router.config.ts   路由

public

    favicon.ico   图标

src

    app.ts   全局错误处理

    global.less   全局样式

    document.ejs   核心模板文件

assets   资源文件

layouts

    LoginLayout.tsx   登录布局文件

LoginLayout.less　登录布局样式

BaseLayout.tsx　　核心布局

pages

Login

index.tsx　　登录页面

index.less　　登录页样式

源码解析——

import { history } from 'umi'

history.push('/users/2')　　　umi3 跳转路由

配置项——

config.ts >

import {defineConfig} from 'umi';

export default defineConfig({

routes: [{

path: '/', **component**: '../layouts/BasicLayout', **routes**: [ ] },　path 请求路径 component 组件路径 routes 子路由（真正的路由在内部 routers）

exact: true, **redirect**: '/list'],　　exact 是否严格匹配　redirect 重定向

publicPath: "/"　　打包时在静态资源路径前加的值　www.baidu.com/umixxxxx.js

ignoreMomentLocale: true　　忽略 moment 的 locale 文件

nodeModulesTransform: { type: none }　node_modules 目录下依赖文件的编译方式

hash: true　　使生成的文件包含 hash 后缀

history: { type: 'browser' }　　history 类型

theme: { '@primary-color': '#1DA57A', },　配置 less 变量

proxy: {　　代理服务器（本地发送的请求 http://localhost:8080/api/xx/xx 请求的路径其实是 http://baidu.com:8899/api/xx/xx）

'/api': {　　访问 /api/users　等同访问 http://baidu.com/api/users，在前方加入网址

'target': 'http://baidu.com:8899',

//'changeOrigin': true,

//'pathRewrite': { '^/api' : '/' },　　重写最终地址的/api （访问 /api/users　等同访问 http://baidu.com/users）

},

},

plugins: ['@umijs/plugin-sass'],　　umi 插件配置（默认有 antd）

dva: { hmr:true },　　配置项的名字通常是插件名去掉 umi-plugin- 或 @umijs/plugin 前缀。

title: “xxx”,

links: [

{ rel: 'icon', href: '图片地址' },　　// href 的图片你可以放在 public 里面，直接./图片名.png 就可以了，也可以是 cdn 链接

],

});

router.config.ts >>

export default [

{

path: '/login',

```
    component: '../layouts/LoginLayout',
    routes: [
      { path: '/login', component: './Login/login' },
    ]
  },
  {
    path: '/',
    component: '../layouts/BasicLayout',
    Routes: ['src/pages/Authorized'],
    routes: [
      { path: '/', redirect: '/overview' },
      {
      path: '/overview',
        name: '数据概览',
        icon: 'pie-chart',
        component: './Data/Business',
        showMenu: true
      },

      {
        path: '/finance',
        name: '22 中心',
        icon: 'calculator',
        key: '1610',
        routes: [
          {
            path: '/finance/mall-settlement',
            name: '33 结算',
            key: '1621',
            routes: [
              {
                path: '/finance/mall-settlement/reconciliation-summary',
                name: '44 表',
                component: './Finance/Reconciliation',
                key: '1622',
                params: { type: 1, businessType: 1 }
              },
            ]
          }
        ]
      }
    ]
  }
];
```

页面——

document.ejs >>

```
<!DOCTYPE html>
```

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link rel="icon" href="/favicon.ico" type="image/x-icon" />
    <title>Boss PaaS</title>
</head>

<body>
    <div id="root"></div>
    <div>lalalla</div>
</body>

</html>
```

BasicLayout.tsx >> 通过显示子路由的 child，实现固定内容不变，child 切换

```
import React, { PureComponent, Dispatch } from 'react';
import { ConfigProvider, Layout, Modal, message, Tabs } from 'antd';
import { withRouter } from 'react-router';
import styles from './BaseLayout.less'
const { Header, Footer, Sider, Content } = Layout;
import logo from "@/assets/daboluo-logo.svg";
export default class BasicLayout extends PureComponent<any, any> {
    public render(): React.ReactNode {

    const { children} = this.props;
        return <Layout style={{ minHeight: '100vh',fontWeight:"400",fontFamily:"Metropolis,Avenir Next,Helvetica
Neue,Arial,sans-serif"}}>
      <Header  style={{ color:
'#fff',backgroundColor:"#00364d",overflow:"hidden",paddingLeft:"30px",height:"60px",lineHeight:"60px"}}>
          <div style={{float:"left",width:"200px"}}>
              <img style={{float:"left",width:"40px",height:"60px",lineHeight:"60px"}} src={logo} alt="daboluo" />
              <div style={{float:"left",paddingLeft:"10px"}}>Daboluo</div>
          </div>
      </Header>
      {children}
    </Layout>
    }
}
```

Login.index.tsx >>

```
import React, { PureComponent, Dispatch } from 'react';
import { FormComponentProps } from 'antd/lib/form';
import { connect } from 'dva';
import { Form, Icon, Input, Button, Tabs, Card } from 'antd';
import REGEX from '@/utils/regex';
```

```
import debounce from 'lodash/debounce';
import styles from './login.scss';

const FormItem = Form.Item;

interface CustomFormProps extends FormComponentProps {
  dispatch: Dispatch<any>
  loading: boolean
}

interface LoginParams {
  loginName: string
  password: number
}

@connect(({ loading }) => ({
  loading: loading.effects['login/login']
}))
class LoginPage extends PureComponent<CustomFormProps, any> {

  public handleSubmit = debounce((event) => {
  }, 500)

    // 输入框失焦处理 去除空格
  handleBlur = (name, value) => {

  }

  public render() {
    const { form: { getFieldDecorator }, loading } = this.props;

    return (
      <div className={styles.login} onSubmit={this.handleSubmit}>
        <Form>
        </Form>
      </div>
    )
  }
};

export default Form.create()(LoginPage);


model.ts >
import { Effect,  Reducer} from 'umi';                    导入 Effect，Reducer 类型
export interface PageState {   }                          限制 state 内容的接口
export interface ExpressModelType {   }                   限制 model 内容（namespace，state，effects，reducers）
const ExpressModel: ExpressModelType= { };
```

```
export default ExpressModel;

index.tsx >
import { IndexModelState, ConnectProps, Loading,connect, history } from 'umi';
class IndexPage extends PureComponent<PageProps,PageState>{ }                    页面组件
export default connect(
({ express, loading }: { express: ExpressState; loading: Loading }) => ({  express,  loading: loading.models.express, }),          限制 express 内容
)(IndexPage)

history.push("#")
history.go(-1)
history.goBack()
```

@vue/cli

```
vue create hello-world     创建一个新项目
vue.config.js  >
const path = require('path')
function resolve(dir) {
  return path.join(__dirname, dir)
}
module.exports = {
    productionSourceMap: false,       不需要生产环境的 source map
    chainWebpack: config => {         通过 webpack-chain 对 webpack 内部配置进行链式操作
    config.resolve.alias          设置别名
      .set('@src', resolve('src'))
    config
      .entry('element-ui')     设置 entry1
      .add('element-ui')       新增入口
      .end()                   结束链式
      .entry('vendor')         设置 entry2
      .add('vue')
      .add('vue-router')
      .add('vuex')
      .add('axios')
      .end()
      .output.chunkFilename(`js/[name].[chunkhash].js`)
      config.optimization.splitChunks({
      chunks: 'all',                    //默认只作用于异步模块，为`all`时对所有模块生效,`initial`对同步模块有效
    })
  },
  lintOnSave: true,          是否在开发环境下通过 eslint-loader 在每次保存时 lint 代码。这个值会在 @vue/cli-plugin-eslint 被安装之后生效。
  devServer: {
    hot: true,
    clientLogLevel: 'warning',
    port: 8080,
    proxy: {                        访问 /api/users   等同访问 http://baidu.com/api/users，在前方加入网址
```

```
    '/api': {
      target: 'http://sccba.yjxyjxyjx.com',
      ws: true,
      changeOrigin: true          允许跨域
      pathRewrite: {
        '^/api': '',              重写最终地址的/api （访问 /api/users  等同访问 http://baidu.com/users）
      },
    }
  }
}
```

Core

webpack

webpack -v --mode --config 对当前目录项目打包（node 包命令执行顺序： /node_modules/.bin global 全局）

【 -v 查看版本 --mode production， development 生产模式或开发模式打包 --config ./config/webpack.config.js 设置配置文件】

webpack serve 运行 webpack-dev-server（需要在 webpack 配置的运行目录手动添加 index.html，并手动添加打包后的文件 bundle.js）

【--open 自动打开浏览器 --mode devlopment 选择开发模式：devlopment production】

dependencies: webpack webpack-cli webpack-dev-server

devDependencies: clean-webpack-plugin html-webpack-plugin style-loader css-loader

adonis 4.1

npm i -g @adonisjs/cli

adonis new mypro 在当前目录创建新目录项目

adonis serve --dev 启动服务

adonis make:controller Admin/User --type http 创建控制器

axios

源码解析^0.21.3

axio.defaults.headers.common["channelId"]="UCP" 自定义 header 属性

react-dom

```
import ReactDOM from "react-dom"
import ReactDOMClient from "react-dom/client"

ReactDOMClient.createRoot( document.getElementById("root") as HTMLElement )  创建根节点
ReactDOM.render(<App/> , document.getElementById("root"))
```

typescript

create-react-app offcial website: https://create-react-app.dev/docs/adding-typescript/

npm install --save typescript @types/node @types/react @types/react-dom @types/jest

rename any file to be a TypeScript file (e.g. src/index.js to src/index.tsx)

Router

react-router-dom

src/App.tsx >>

```
import React,{Suspense} from 'react';
```

```
import {  Link,NavLink, Routes,Navigate,BrowserRouter,
HashRouter,Route,Outlet,useNavigate,useLocation,useParams   } from 'react-router-
dom';       //依赖 react-router
const Test =React.lazy(()=>import("@pages/Test"))

const Home = () => {
  const navigate=useNavigate()    // 编程跳转，只能在 Route 组件内使用
  const location=useLocation()    // 获取路径信息      {pathname: '/base/ssss/333', search:
'', hash: '', state: null, key: 'default'}
  const params=useParams()          // 获取冒号参数信息  {username: 'ssss', lala: '333'}
  console.log("home location",location)
  console.log("home params",params)
  return (
      <div>
          Home
          <button onClick={()=>navigate("/home/redirect/lala/ddd")}>navigate</button>
          <Outlet />   {/* 路由出口，父路由使用，匹配加载后继续向子路由匹配 */}
      </div>
  );
}
const Person = (props:any)=> {
  return (
      <div>
          person
          <Outlet />   {/* 路由出口，父路由使用，匹配加载后继续向子路由匹配 */}
      </div>
  );
}

const App: React.FC<any> = () => {
  return (
    <>
      <BrowserRouter>  {/* BrowserRouter 为无#路由    */}
        <Link to="/base" replace>点击跳转 Home</Link><br/>
        <Link to="/base/second1" replace>点击跳转 second1</Link><br/>      {/* 点击 Link 跳转到
指定路由 ( replace 替换原页面的 URL 历史纪录地址  )   */}
        <Link to="/base/second2" replace>点击跳转 second2</Link><br/>

        <NavLink to="/home/direct">点击跳转 direct</NavLink><br/>
        <Link to="/home/redirect/lala/ddd" replace>点击跳转 redirect</Link><br/>
        <Suspense fallback={<div>loading....</div>}>                    {/* 路由懒加载 */}
          <Routes>                                                        {/* 从上向下匹配,
只匹配一个 Route (等于 5.x.x 的 Switch)  */}
            <Route path='/base/:username/:lala' element={<Home
/>}>                        {/* 匹配路由，父路由不需要匹配 (去掉了 5.x.x 的 component 和 render)
*/}
```

```
                <Route path='second1' element={<div>second1</div>}></Route>      {/* 二级路由
*/}
                <Route path='second2' element={<div>second2</div>}></Route>      {/* 二级路由
*/}
            </Route>
            <Route path='/home/direct' element={<Test/>}></Route>                          {/*
匹配请求路径并显示组件，也可以用编程式跳转 location.href 或点击<a>跳转 */}
            <Route path='/home/redirect/*' element={<Navigate to="/home/direct"
/>}></Route>    {/* 跳转，(等于 5.x.x 的 Redirect 的 from, to)  */}
          </Routes>
        </Suspense>
      </BrowserRouter>
    </>
  );
};

export default App;
```

import { Switch, Route, Router, withRouter } from 'react-router'; react-router-dom 的核心功能
import { HashHistory, Link, Redirect, BrowserRouter, HashRouter } from 'react-router-dom'; 依赖 react-router

xxx.jsx >

export default function App (){ this.props.children 切换路由不会出发 componentDidmount 会触发 componentDidUpdate
  return
   <**Link** to="/about" [replace]>点击跳转</**Link**> 点击 Link 跳转到指定路由（replace 替换原页面的 URL 历史纪录地址 ）
  <**BrowserRouter**> BrowserRouter 为无#路由
  <**Switch**> Switch 从上向下匹配，只匹配一个 Route
    <Route path='/' component={Login} render={ ()=>{ } }></Route>
    <Route path='/admin/dd' component={Admin}></Route> Route 匹配请求路径并显示组件【跳转到其他路由需要手动 location.href 或点击<a>跳转
    <Redirect from='/home/*' to='/home/detail'></Redirect> Redircct Route 匹配将的路由 重定向到另一个网址，没有 Route 直接重定向
    <Redirect to='/home'></Redirect> 如果前面的都不匹配，则匹配最后一个，类似于 from='*'
</Switch>
  </BrowserRouter>
}

BasicLayout.tsx >

withRouter(BasicLayout) 路由相关的方法通过 props 传给它包裹的组件的 props 上（this.props.children==子页面）
this.props.match.params.id 获取路径匹配的参数
this.props.history.goBack(); 跳转页面
this.props.location 获取路由标签传递下来的参数

vue-router

官方前端路由（不利于 SEO，虚构界面不能被搜索，浏览器无法前进后退点击只会重新请求，单页面无法记住滚动位置,）

main.js

import **Vue** from 'vue'
import **Router** from 'vue-router'

import son from '@/views/son'　　导入各个组件配置到路由实例中
Vue.use(Router)
const router= new Router({
mode: 'history',　　路由模式默认为哈希（#开头），history 为直接用 url 地址
　routes: [{ path:'/father',　　拦截路由 （动态地址 /login:id:name',　使用<router-link to="/login/12/pna"）
　　name: '店铺' },　　路由名-便于 js 跳转
　　component: son.　　匹配组件
　　children:[　{path:'son', component: {template:'<>'}]　},　　子路由（ < router-link to="/login/son"></router-link> ）
　　{ path:'/login', components: { 'default': head, 'left': leftBox, } },　　多坑显示多组件 （<router-view/> <router-view name="left"/> ）
　　{ path:'/',　　redirect: '/login' },　　重定向到另一个路由
　　],
　linkActiveClass: 'myactive'　　自定义 css 中 .router-link-active 的名称（router-link 激活时的样式）
 })
router.beforeEach((to, from, next)=>{. }).　　访问路径时过滤路由【to 即将跳转的路由实例，next()直接跳转】
new Vue({ router, store, render: (h) => h(App),
 renderError(h, err) { return h('pre', { style: { color: 'red' } }, err.stack);　}　}).$mount('#app');
App.vue　　<router-link to="/login?id=9" tag="span" > xxx </router-link>　　路由链接 【tag 指定渲染成的元素-含默认点击显示组件事件】
　　<router-link v-bind:to="{ name:'cart', params:{id:123} }">　　路由链接（根据路由名跳转）
　　<router-view/>　　路由坑（根据路由链接显示相应的组件，路由对象挂载后，可以直接拦截路由显示了）
　　.router-link-active{}　路由链接激活样式　<router-link class='router-link-active' >
　　js 使用路由　(this.$route 参数对象　this.router 导航对象-路由前进后退跳转)
　　this.$router.push({ name:'user', params:{id} })　根据路由名跳转
　　this.$router.push('/home/goodsinfo')　直接跳转
　　this.$router.push({path: '/backend/order', query: {selected: "2"}})
　　this.$router.go(1)　前进后退
　　this.$router.options.routes　router 示例的内部参数
　　this.$route.path　请求路径 【 {{ $route.query.id }}　页面模板中直接使用 】
　　this.$route.query　请求参数的键值对对象
　　this.$route.params.id　动态地址:id 匹配到的数据

http-proxy-middleware

Concept

Node.js proxying made simple. Configure proxy middleware with ease for connect, express, next.js and many more. Powered by the popular Nodejitsu http-proxy.

Proxy Options

pathRewrite (object/function)

Rewrite target's url path. Object-keys will be used as RegExp to match paths.

```
// rewrite path
pathRewrite: {'^/old/api' : '/new/api'}

// remove path
pathRewrite: {'^/remove/api' : ''}

// add base path
```

```
pathRewrite: {'^/' : '/basepath/'}

// custom rewriting
pathRewrite: function (path, req) { return path.replace('/api', '/base/api') }

// custom rewriting, returning Promise
pathRewrite: async function (path, req) {
  const should_add_something = await httpRequestToDecideSomething(path);
  if (should_add_something) path += "something";
  return path;
}
```

[http-proxy-middleware]

createProxyMiddleware

export declare function **createProxyMiddleware**(context: Filter | Options, options?: Options): import("./types").RequestHandler;

Proxy /api requests to http://www.example.org

```
// typescript
import * as express from 'express';
import { createProxyMiddleware, Filter, Options, RequestHandler } from 'http-proxy-middleware';

const app = express();

app.use(
  '/api',                    // Avoid using '/' directly, as it may cause static files to be
inaccessible.
  createProxyMiddleware({
    target: 'http://www.example.org/api',
    changeOrigin: true,
  })
);

app.listen(3000);

// proxy and keep the same base path "/api"
// http://127.0.0.1:3000/api/foo/bar -> http://www.example.org/api/foo/bar
```

Data Storage
recoil
[recoil]

RecoilRoot

export const **RecoilRoot**: React.FC<RecoilRootProps>;

Components that use recoil state need RecoilRoot to appear somewhere in the parent tree. A good place to put this is in your root component:

```
import React from 'react';
import {
  RecoilRoot,
  atom,
  selector,
  useRecoilState,
  useRecoilValue,
} from 'recoil';

function App() {
  return (
    <RecoilRoot>
      <CharacterCounter />
    </RecoilRoot>
  );
```

```
}
```

RecoilState

export class **RecoilState**<T> extends AbstractRecoilValue<T> {}

atom

export function **atom**<T>(options: AtomOptions<T>): RecoilState<T>;

An atom represents a piece of state. Atoms can be read from and written to from any component.
Components that read the value of an atom are implicitly subscribed to that atom,
so any atom updates will result in a re-render of all components subscribed to that atom

```
const textState : RecoilState<string> = atom({
  key: 'textState',      // unique ID (with respect to other atoms/selectors)
  default: '',           // default value (aka initial value)
});
```

useRecoilStateLoadable

export function **useRecoilStateLoadable**<T>(recoilState: RecoilState<T>): [Loadable<T>, SetterOrUpdater<T>];

This hook can avoid state loading errors when triggering props.children rendering.
This hook is intended to be used for reading the value of asynchronous selectors.
This hook will implicitly subscribe the component to the given state.

```
function UserInfo({userID}) {
  const [userNameLoadable, setUserName] = useRecoilStateLoadable(userNameQuery(userID));
  switch (userNameLoadable.state) {
    case 'hasValue':
      return <div>{userNameLoadable.contents}</div>;
    case 'loading':
      return <div>Loading...</div>;
    case 'hasError':
      throw userNameLoadable.contents;
  }
}
```

Other

src/store/atoms/DashboardAtom.ts >> 数据存储

```
import {
    atom,
    selector,
} from 'recoil';

const TEXT_STATE = atom({
    key: 'textState',
    default: '',           //不设置默认值会导致原子挂起，页面无法加载，直到被设置值
});
const TEXT_STATE_COMPUTED = selector({
    key: 'textStateChangeValue',
    /**
     * 1.若 get 内进行的是异步请求，则需要把 promise 对象返回出去
     * 2.报错处理:throw ... 抛出错误
     * 3.带参数查询:selector --> selectorFamily
     */
    get: ({ get }) => {
        const text = get(TEXT_STATE);
        return new Promise((resolve, reject) => {
```

```
            resolve(text.length)
        })
    },
    // get: async ({ get }) => {
    //     const text = get(TEXT_STATE);
    //     const response = await new Promise((resolve, reject) => {
    //         resolve(text.length)
    //     })
    //     return response
    // },
})

export default { TEXT_STATE, TEXT_STATE_COMPUTED };
export {
    TEXT_STATE,
    TEXT_STATE_COMPUTED
}
```

src/api/account.ts >> 行为存储

```
import axios from 'axios';

export const fetchAccuntDataSync = async ()=>{
     const res=await axios.get("/sdk-citi/account");
     return res.data;
}
```

src/pages/Test.ts >> 页面使用

```
import React from "react";
import "./tese1.css";
import { TEXT_STATE,TEXT_STATE_COMPUTED } from "../../recoil/index";

import { useRecoilState, useRecoilValue } from "recoil";
import { Button } from 'antd';
function Test1(props) {
  const [text, setText] = useRecoilState(TEXT_STATE);
  const length = useRecoilValue(TEXT_STATE_COMPUTED)
  const [accountRecoilDataLoadable, setAccountRecoilData] =
useRecoilStateLoadable<IAccountData>(accountData);

  const onChange = (event) => {
    setText(event.target.value);
  };

  return (
    <div>
      <input type="text" value={text} onChange={onChange} />
      <br />
      输入值: {text}
      <br />
```

```
      输入值的长度: {length}
      <br />
      <Button type="primary" onClick={() => props.history.push('/test')}/>
    </div>
  );
}

export default Test1;
```

export default Test1;

react-redux

依赖：redux　react-redux （状态管理）　redux-thunk（异步请求）

src/reduxs/action/dashboard/dashboardAction.tsx >>　异步动作 action

```
import { Dispatch } from "react";
import { IUpdateServerParams } from "../../../types";
import axios from "axios";

export const getAccountDataAction=(params: IUpdateServerParams)=>(dispatch:
Dispatch<any>)=>{       //获取数据 并派发到 reducer 里
    dispatch(receiveDashboardLoadingStatus(true))            //设置 loading 状态
    return axios.get("/cashmgmt/account").then((res)=>{
        dispatch({data:res.data,type:"CHANGE_ACCOUNT_DATA"});
    }).catch(err=>{
        dispatch(receiveServerErrorResponse(500,err))
    }).finally(()=>{
        dispatch(receiveDashboardLoadingStatus(false))
    })
}


const receiveDashboardLoadingStatus=(status:any)=>({
    type:"DASHBOARD_LOADING",
    isLoading:status
})
const receiveServerErrorResponse=(status:number,detail:string)=>({
    type:"SERVER_ERROR_RESPONSE",
    status,
    detail
})
```

src/reducer/index.tsx >>　设置 仓库数据 reducer

```
import { combineReducers,AnyAction } from "redux";


const appReducer=combineReducers<any,AnyAction>({
    accountData: function (state={},action:AnyAction){
        console.log("update accountData",action)
        if(action.type==="CHANGE_ACCOUNT_DATA"){
            return {
                ...state,                          //旧数据 accountData 的值
                accountData:action.data            //修改的新数据内容
```

```
            }
        }else{
            return state;
        }

    },
    testData: function (state={},action:AnyAction){
        console.log("update testData",action)
        if(action.type==="CHANGE_TEST_DATA"){
            return {
                ...state,                       //旧数据 testData 的值
                ...action.data                  //修改的新数据内容
            }
        }else{
            return state;
        }
    },
    loadingStatus:function (state={},action:AnyAction){
        console.log("update loadingStatus",action)
        if(action.type==="DASHBOARD_LOADING"){
            return {
                ...state,
                isLoading:action.isLoading,
                detail:action.detail
            }
        }else{
            return state;
        }
    },
    serverResponse:function (state={},action:AnyAction){
        console.log("update serverResponse",action)
        if(action.type==="SERVER_ERROR_RESPONSE"){
            return {
                ...state,
                status:action.status,
                detail:action.detail
            }
        }else{
            return state;
        }
    },
} as any)


const result= (state:any,action: AnyAction)=>{
    return appReducer(state,action);
}

export default result;
```

src/index.tsx >> 渲染首页

```tsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import { Provider } from 'react-redux';
import rootReducer from "./reduxs/reducer"
import {createStore,applyMiddleware} from "redux"
import reportWebVitals from "./reportWebVitals"
import thunk from 'redux-thunk';
export const store = createStore(rootReducer,applyMiddleware(thunk))        //创建 redux 数据仓库

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App role="test"/>
    </Provider>
  </React.StrictMode>
)

reportWebVitals();
```

src/App.tsx >> 首页组件

```tsx
import React,{useEffect} from 'react';
import './App.css';

import { IServerResponseData, IUpdateServerParams } from './types';
import { getAccountDataAction } from './reduxs/action/server/serverResponseAction';
import {connect } from "react-redux";
import {ThunkDispatch } from "redux-thunk";

export interface IApp{
  role?: string;
  serverResponse?: [IServerResponseData];
  testData?:any;
  loadingStatus?:any;
  onUpdateAccountDataResponse?: any;
  onUpdateTestData?: any;
}


const App: React.FC<IApp> =
({role,serverResponse,testData,loadingStatus,onUpdateAccountDataResponse,onUpdateTestData})=>
{
  useEffect(()=>{
    onUpdateAccountDataResponse();
```

```
  },[])
  return <div>
    <button onClick={()=>{ onUpdateAccountDataResponse(); }}>accountData</button>
    <button
onClick={()=>{ onUpdateTestData({type:"CHANGE_TEST_DATA",data:"test"}); }}>testdata</button>

    <div><span>loadingStatus:</span>{JSON.stringify(loadingStatus)}</div>
    <div><span>testData:</span>{JSON.stringify(testData)}</div>
    <div><span>accountData:</span>{JSON.stringify(serverResponse)}</div>
  </div>
}
function mapStateToProps({accountData,testData,loadingStatus}:any){           //从 store 拿出数
据映射到 props 里，可以通过 Props 使用
  return {
    serverResponse:accountData.accountData,
    testData:testData,
    loadingStatus:loadingStatus
  }
}
const mapDispatchToProps=(dispatch: ThunkDispatch<{},{},any>)=>{              //把 action 里的方
法绑定到 props 上，用 Props 修改 store 里的数据
  return {
    onUpdateAccountDataResponse:(params:IUpdateServerParams)=>dispatch(getAccountDataAction(p
arams)),
    onUpdateTestData:(params:any)=>dispatch(params)
  }
}


export default connect(mapStateToProps,mapDispatchToProps)(App);
```

dva

依赖："dva": "^2.4.1"

使用 2.4.1——

model.js >>

```
import { message } from 'antd';
import { queryPortLineList,queryDataviewTemperatureList } from './service';

const Model = {
 namespace: 'overview',  //命名空间
 state: {              //基础数据
  portLineList: [],
  temperatureList:[]
 },
 effects: {            //异步函数
  *queryDataviewTemperatureList({payload},{call,put}){
   const returndata = yield call(queryDataviewTemperatureList, payload);
   if(returndata&&returndata.code=='000'){
```

```
        yield put({
          type:"saveData",
          payload:{
            temperatureList:returndata.data
          }
        })
      }
    }
  },
  reducers: {              //内部函数
    saveData(state, { payload }) {
      console.log('payload: ', payload);
      return { ...state, ...payload };
    },
  },
};
export default Model;
 overview.tsx >>
import React, { PureComponent ,Component} from 'react';
import { Button, Slider, Layout,Spin } from 'antd';
import {
  G2,
  Chart,
} from 'bizcharts';
import {connect} from 'umi';
import styles from './index.scss';

const { Header, Content, Sider, Footer } = Layout;
import BigMap from '@/components/BigMap/BigMap';
interface OverviewProps {
  loading:boolean;
  overview:any;
  dispatch:any;
}

class Overview extends Component<OverviewProps, any> {
  componentDidMount() {
    if (this.props.dispatch) {
      this.props.dispatch({
        type: 'overview/queryPortLineList',    //指定的异步函数
        payload:{},                            //参数
        callback: ()=>{},                      //回调函数
      });
    }
  }
  render() {
    return (
      <div className={styles.pageContainer}>
```

```
      </div>
    );
  }
}

export default connect(
  ({ overview, loading }) => ({
    overview,
    loading: loading.models.overview,
  })
)(Overview);
service.ts >>
export async function queryAppList(params) {                //GET
  return request('/api/v1/boss/app/list', params)
};
export async function sendValidateCaptch(params) {          //POST
  return request('/api/v1/boss/imageValidate/validate', {
    method: 'POST',
    body: params
  });
};
export async function queryCaptchImage(params) {            //IMAGE
  return request('/api/v1/boss/imageValidate/getStreamVerify', {
    method: 'IMAGE',
    params
  })
};

import { routerRedux }  from 'dva/router'            路由功能    routerRedux.push()
import fatch from 'dva/fetch'                        路由功能
@connect(  ( {loading, xxx}  )=>{
  pageLoading: loading.models["authAccount"],                                全局 effect 都获得结果为 false【yield
put({  type:"xxx", payload:response.data })】
  roleLoading: loading.effects["authAccount/getRoleListByTenantId"],        获得结果为 false
  userList: xxx.userList  }  )                                               载入 model 内数据到 props 中
class indexPage extends Component{
    handleClick(){
        dispatch({                        访问 modle 内部 effect 异步函数（dispatch==React.Dispatch  中间派遣访问 effect）
      type: 'todo/addTodo ',        调用异步函数   namespace==todo   effect==addTodo
            payload: { status:1},          参数对象
            callback: function(){}         回调函数
        });
        fetch('www.baidu.com/api/list',  {  method:'POST',   body:JSON.stringify(param) ,   headers: {  'content-type':
'application/json' }   )
       .then(  (response)=>{  response++  }   )
       .then(response => response )
        .catch(err => ({ err }));
```

```
  }
}
```

mobx

store.js >

```
import { observable, action, computed } from 'mobx'

class HomeModel {
@observable a=0
@action async add(){ this.a++ }
@computed geta(){ return this.a }
}
export default new HomeModel()
```

xxx.js>

```
import { inject, observer } from 'mobx-react'.
@Consumer
@inject('HomeModel')
@observer
export default class Home extends React.Component
const { geta, a } = this.props.HomeModel
```

inject 注入 store 仓库 **observer** 封装下一行语句

this.props 从父级调用 store 仓库内的异步函数，之后再获取数据 a

【dva 是直接获取数据，mobx 需要调用函数手动获取数据】

father.jsx >

```
import { Provider } from 'mobx-react'
<Provider { ...store }>  <Router> <Route component={ Loading } </Router> </Provider>
```

Data Utils

lodash

moment

```
moment.locale()                    查看全局语言环境 // en
moment("1995-12-25")                    解析时间字符串，返回 moment 对象
moment("12-25-1995", "MM-DD-YYYY");    解析时间字符串，返回 moment 对象
moment('24/12/2019 09:15:00', "DD MM YYYY hh:mm:ss", true);    解析时间字符串，严格模式，格式不符合会报错，返回
moment 对象 // var day = moment("2022/1/14 22:05","YYYY/M/D H:mm",true)
```

File Processing

html2canvas

Usage

```
import html2canvas from 'html2canvas';
html2canvas(document.body).then(function(canvas) {
  document.body.appendChild(canvas);
});
```

koa

server.js >

```
const Koa = require("koa");                    // 获取 koa
const bodyParser = require("koa-bodyparser");    // body 中间件
const static = require("koa-static");          // 静态服务器中间件
```

```
const path = require("path");
const KoaRouter = require("@koa/router");          // 路由包
const views= require("koa-views");                  // 模板引擎
const userInfo= require("./user/info");             // 嵌套子路由

const appRouter = new KoaRouter();          // 实例化路由

const app = new Koa();                              // 实例化

app.use(appRouter.routes());                    // 使用路由
app.use(appRouter.allowedMethods());


app.use(bodyParser());                                      // 使用 body 中间件

app.use( static(path.resolve(__dirname, "static"))  );      //处理此目录下的 css 和 js 等静态资源请求，不会返回 html 了
app.use(views(path.join(__dirname,"./views"),{             //views 目录下的 ejs 文件全部使用模板渲染
   map:{
      html: "ejs"
   }
}));                // 使用模板引擎

appRouter.get("/form", async (ctx) => {
   await ctx.render("index",{name:"张三"})
});
appRouter.get("/info", async (ctx) => {
   await ctx.render("user");
});

appRouter.post("/form", async (ctx) => {
  ctx.body = ctx.request.body;
})
appRouter.use("/user",otherRouter.routes()); // 使用嵌套路由

app.listen(3000);    //监听端口
```

art-template
app.js > var template=require('art-template'); 直接加载 node_modules 里的包 template.render(data.toString(), {name:'jack'}) 读取整个 HTML 文件为字符串再用对象替换
art-template **express-art-template**
app.js > app.engine('art', require('express-art-template')); 配置使用模板引擎（渲染.art 结尾的文件时默认使用 art-template，配制后可用 res.render 方法) app.set('views','render 默认路径') 修改默认的 views 渲染目录 res.render('xxx.art'或'admin/index.html', {a:b}) 自动去项目根目录的 views 目录渲染文件并发送响应 res.redirect('/') 临时重定向结束响应（服务端只针对同步请求跳转，可以设置 location.href)
body-parser

获取 post 请求数据插件（在 router 挂载之前）

app.js > var bodyParser=require('body-parser')

app.use(bodyParser.urlencoded({ extended:false }))　　app.use(bodyParser.json())　配置 body-parser 后可使用 **req.body** 对象获取 post 请求数据

mongoose

封装 MongoDB 数据库的包（对官方的 mongodb 包进行了封装-效率高）

app.js > var mongoose = require('mongoose');　　加载模块

var Schema = mongoose.Schema　　引入结构类

mongoose.connect('mongodb://localhost/test');　连接指定的数据库

mongoose.Promise=golbal.Promise;

var aaaSchema=new Schema({title: String, comments:[{body:String}], data:{type: Date, default: Data.now}}）新建一个表结构

titile:{**type: Date 或 Number**--JS 类型,**enum:[1, 0,-1]**--多个可选值 **required: true**--必须有, **default: Date.now**--每次创建时动态调用此方法，防止立刻调用为固定值}

var Table = mongoose.model('Table', aaaSchema 或{ name: String});　　创建表类---自动变成小写复数 tables

var a = new Table({ name: 'Silence' });　　新建一个数据（可配合循环语句使用）

a.save(function (err, ret) {　if (err) {} else {}　});　　保存一个数据（ret 为此时插入的数据）

Table.find({name:'xx'}, function (err, ret) {　if (err) {} else {}　})　　查询指定条件的数据（省略条件为所有数据）

{$or:[{a:xx},{b:xx}] }　或条件

Table.findOne({name:'xx'}, function (err, ret) {　if (err) {} else {}　})　　查询匹配的第一条数据

Table.remove({name:'xx'}, function (err, ret) {　if (err) {} else {}　})　　删除对象

Table.findByIdAndUpdate('idnumber', {name:'xx'}—更新内容, function (err, ret) {　if (err) {} else {}　}）更新数据

mysql

封装连接 MySQL 数据库的包

app.js > var mysql=require('mysql')　　加载模块

var connectin=mysql.createConnection({host:'localhost',user:'xx', password:'xx', database:'xx'}）创建连接

connection.connect(); 连接数据库

connection.query('SQL 语句', function(err, res, filelds){ if(err) throw err; console.log(res[0].solution) }）操作数据

connection.end()　关闭连接

express-session

var session =require(' express-session')　　加载模块

app.use(session({ secret:'keyboardcat', ----配置 sessionid 加密字符串，在原有加密基础上加密

resave:false,

saveUninitialized:false } ----存数据再发送 id（true 无论是否使用 sesseion 都发送一个 id）

))

req.session.foo='bar'　req.session.foo　添加和获取 session 数据　（删除 delete req.session.foo）

nodemon

监听 node 执行文件变化自动同步服务器 【-g】

nodemon xxx.js

dotenv

导入.env 内的配置到 process.env 中

require('dotenv').config()

knex

orm 数据库查询

knex.column('title', 'author', 'year').select().from('books')

Loader

css

style-loader 可以将 css 文件变成 style 标签插入 head 中（放在最后）

css-loader 打包时转换 css 文件中的 url 路径为正确的路径，并将 css 文件变成一个模块

sass-loade node-sass 编译 sass （配置 node 镜像地址，NET Framework 下载，全局 windows-build-tools 构建工具，全局 node-sass 环境， python 2.7 环境，和 node 版本一致）

babel

@babel/core : It allows us to run babel from tools like webpack.

babel-loader : Its a webpack plugin. It allows us to teach webpack

@babel/preset-env : Which helps babel to convert ES6, ES7 and ES8 code to ES5.

@babel/preset-react : Which Transforms JSX to JavaScript.

@babel/preset-typescript: TypeScript compiler

@babel/plugin-proposal-class-properties 此插件转换静态类属性以及使用属性初始值设定项语法声明的属性

@babel/plugin-transform-runtime 外部化对帮助程序和内置函数的引用，自动填充您的代码而不会污染全局变量

@types/node

@types/react

@types/react-dom

Project Support

react-copy-to-clipboard

```
import {CopyToClipboard} from 'react-copy-to-clipboard'
<CopyToClipboard text={"fffff"}                    拷贝内容
        onCopy={ Toast.success("sucess！") }>     拷贝成功函数
        <span>show content</span>
  </CopyToClipboard>
```

react-sortable-hoc

```
import {SortableContainer, SortableElement} from 'react-sortable-hoc';
const SortableItem= SortableElement(  (props) => <div className="pagelist">{props.children}</div>  );        子容器
const SortableList= SortableContainer(  (props)=>
    { props.imglist.map( (val, ind) =>
        <SortableItem  index={ind}  key={ind} imgsrc={val} ind={ind}>          ind 为固定的顺序，不能当作索引  index 和 key 必须属性
        </SortableItem>
     )
}
  onSortEnd = ({oldIndex, newIndex}) => {  };                                                                    排序结束函数，需要手动交换 imglist 的 oldindex，newindex 元素
<SortableList   imglist={ this.state.imglist }  axis="x" distance={5} onSortEnd={this.onSortEnd}>       ind 为固定的顺序，不能当作索引  index 和 key 必须属性
</SortableList>
```

qs

qs 是一个用于解析和字符串化的工具库。

```
var obj = qs.parse('a=c');        结果   { a: 'c' }
var str = qs.stringify(obj);      结果 stringify  '字符串化'   'a=c'
qs.parse('foo[bar]=baz')
qs.parse('a[hasOwnProperty]=b', { plainObjects: true }
```

mock

4)、属性值为 object

```
' name| count' : object ;   //从 object 中随机抽取 count 个属性;
```

' name| min-max' : object; //从 object 中随机抽取 min 到 max 个属性;

5)、属性值为 array

' name| 1': array ; //从 array 中随机选取一个值最为最终值; 【最终值为元素，不是数组了】

' name| +1' : array; //从 array 中顺序选择一个元素，最为最终值;

' name| min-max' : array ; //通过重复 array 生成一个新数组，重复的次数大于等于 min, 小于等于 max ;

' name| count' : array; //通过重复 array 生成一个新数组，重复的次数为 count ;

aliossUploader

阿里云 oss 上传插件

libphonenumber-js

依赖: npm install libphonenumber-js

import { parsePhoneNumber, isPossiblePhoneNumber, isValidPhoneNumber, validatePhoneNumberLength } from 'libphonenumber-js'

isPossiblePhoneNumber('8 (800) 555-35-35', 'RU') === true
isValidPhoneNumber('8 (800) 555-35-35', 'RU') === true

validatePhoneNumberLength('8 (800) 555', 'RU') === 'TOO_SHORT'
validatePhoneNumberLength('8 (800) 555-35-35', 'RU') === undefined

const phoneNumber = parsePhoneNumber('+12133734253') 格式化电话号
phoneNumber.formatInternational() === '+1 213 373 4253'
phoneNumber.formatNational() === '(213) 373-4253'
phoneNumber.getURI() === 'tel:+12133734253'

parsePhoneNumber('+12133734253')

classnames

```
import classnames from 'classnames';
const inputCls = classnames({              根据条件判断是否启用样式
    [styles.input]: true,
    [styles.small]: (size === 'small'),
    [styles.large]: (size === 'large'),
    [styles.default]: (size === 'default'),
  });
```

eslint

eslint Command

eslint [options] file.js [file.js] [dir]
Fixing problems://修正问题
  --fix   Automatically fix problems//自动修复问题
  --fix-dry-run   Automatically fix problems without saving the changes to the file system//自动修复问题而不保存对文件系统的更改

Basic configuration: //基本配置
  --no-eslintrc   Disable use of configuration from .eslintrc.* //禁止使用来自.eslintrc.*的配置文件
  -c, --config path::String   Use this configuration, overriding .eslintrc.* config options if present //如果存在.eslintrc.*，则使用且重写该配置文件

```
  --env [String]                Specify environments //指定环境
  --ext [String]                Specify JavaScript file extensions - default: .js //指定的 JS 文件扩展名，默认：.js
  --global [String]             Define global variables //定义全局变量
  --parser String               Specify the parser to be used //指定使用某种解析器
  --parser-options Object       Specify parser options //指定解析参数

Specifying rules and plugins: //指定规则和插件
  --rulesdir [path::String]     Use additional rules from this directory //从该路径使用额外的规则
  --plugin [String]             Specify plugins //指定插件
  --rule Object                 Specify rules//指定规则


Ignoring files: //忽略文件
  --ignore-path path::String    Specify path of ignore file //指定忽略的文件
  --no-ignore                   Disable use of ignore files and patterns //禁止使用忽略文件和样式
  --ignore-pattern [String]     Pattern of files to ignore (in addition to those in .eslintignore) //要忽略的文件模式（除了
在.eslintignore 中的文件）

Using stdin: //unix]标准输入（设备）文件
  --stdin                       Lint code provided on <STDIN> - default: false
  --stdin-filename String       Specify filename to process STDIN as //指定用于处理 stdin 的文件名

Handling warnings://处理警告
  --quiet                       Report errors only - default: false //仅以错误报告出来
  --max-warnings Int            Number of warnings to trigger nonzero exit code - default: -1 //要触发非零退出代码的警告数-默认
值：-1

Output:
  -o, --output-file path::String  Specify file to write report to //指定输出的文件路径
  -f, --format String           Use a specific output format - default: stylish //使用特定的输出格式-默认：stylish
  --color, --no-color           Force enabling/disabling of color

Inline configuration comments: //内联配置注释
  --no-inline-config            Prevent comments from changing config or rules //阻止注释更改配置或规则
  --report-unused-disable-directives  Adds reported errors for unused eslint-disable directives //为未使用的 eslint disable 指令
添加报告的错误

Caching:
  --cache                       Only check changed files - default: false //仅仅检查改变过的文件
  --cache-file path::String     Path to the cache file. Deprecated: use --cache-location - default: .eslintcache//缓存文件的路径。
已弃用：使用--缓存位置-默认值：.eslintcache
  --cache-location path::String  Path to the cache file or directory//缓存文件或文件夹的路径

Miscellaneous://其他
  --init                        Run config initialization wizard - default: false //运行配置初始化向导-默认值：false
  --debug                       Output debugging information//输出调试信息
  -h, --help                    Show help
```

-v, --version Output the version number
--print-config path::String Print the configuration for the given file//打印给定文件的配置

.eslintignore >>
src/** 检查忽略那些文件

eslint-plugin-react-hooks

Avoid Breaking the Rules of Hooks
You can use the eslint-plugin-react-hooks plugin to catch some of these mistakes.

normalize.css

初始化 css 样式

react-dev-utils

react 调试工具

Server

express

封装 http 模块的项目框架 (**app.js**)
var express = require('express') 加载模块 var **app**=**express**(); 创建一个 Server 实例
app.**listen**(3000, **function**(){}) 设置监听端口
app.all 匹配所有动作
app.**use**(["/abc/"], express.static('./public/')) 开放允许访问目录（请求以/abc/开头时，去./public/寻找文件---省略为直接去./public/寻找文件）
app.use(function(req, res, [next]){ next() }) 匹配任意请求，默认跳过下方所有中间件（调用 next 进入下一能匹配的中间件，可在最后添加进行 404 处理）
app.use('/a', function(req, res, [next]){ next() }) 匹配指定请求路径开头
app.**get**('/b', function(req, res, [next]){ next() }) app.**post**('/b', function(req, res, [next]){ next() }) 匹配指定请求路径和指定方法
app.use(function(err, req, res, next){ res.status(500).send(err.message) }) 错误处理中间件
【 next(err)---next 函数带有参数直接进入错误处理中间件 】
res.**send**('xxx') 结束发送响应体（原生的方法依然存在，支持链式编程）
res.**json**({a:xx}) 将对象转为字符串并发送响应体（项目会发送一些自定义状态码 err_code 和信息 message）
res.**status**(200) 设置状态码
封装路由 **router.js >** var router = express.Router() 创建路由
router.get('/login', function(req,res){}) router.post('/login', function(req,res){}) 处理各种请求
app.js > app.use(router) 路由添加到 app 服务中

UI

antd

import { ConfigProvider } from 'antd' 全局化配置转换为中文
import zh_CN from "antd/es/locale/zh_CN";

```
<ConfigProvider locale={zhCN}>
    <Routes />
</ConfigProvider>
```

getFieldDecorator("ara",{ rules: [{ required:true,message:"xxx", pattern:/\s/ }] })( ) </Form.Item>
getFieldsValue() 获取所有键值，Object.keys 获取所有键

getFieldValue("ara")　获取指定键的值
validateFields((err,values)=>{})　　//在事件内使用 form.setFieldsValue 对值进行预处理　onBlur={e => this.handleBlur("name", e.target.value)}
columns=[{ dataIndex:"name" }]　dataIndex　取出 dataSource 内部指定的属性值　　dataIndex:"ind" 会默认排序
export Form.create({})(UserManagePage)

修改原有样式:

```
.sideBar{
   min-width: 150px!important;
   background-color: #1b2a32;
   color: #fff;
   :global(.ant-menu){
      background-color: pink;
   }
 }
```

antd-mobile

```
 homeList = new ListView.DataSource({                    ListView
   rowHasChanged: (row1, row2) => row1 !== row2,
 })
this.homeList = this.homeList.cloneWithRows(this.list)      数组转 Listview
    <ListView
        ref={el => this.listView = el}
        dataSource={this.listDataSource}
        renderHeader={() => <span>header</span>}
        renderFooter={() => (<div style={{ padding: 30, textAlign: 'center' }}>
          {this.state.isLoading ? 'Loading...' : 'Loaded'}
        </div>)}
        renderBodyComponent={() =><div> this is text </div>}
        style={{
          height: this.state.height,
          overflow: 'auto',
        }}
        renderRow={row}
        pageSize={4}
        onScroll={() => { console.log('scroll'); }}
        scrollRenderAheadDistance={500}
        onEndReached={this.onEndReached}
        onEndReachedThreshold={10}
      />
```

Node / Core

http

var http = require('http')　　http 服务
var app=http.createServer()　　创建一个 Server 实例（其他文件内定义需要执行 app 的函数并导入主文件---主文件调用时传入 app）
app.on('request', function(request, response){})　　客户端请求事件（默认发送 utf8 编码响应体，浏览器不知道响应体类型时用操作系统默认 GBK 解码）
　　req 请求对象　　req.url　　请求路径（默认路径/限制访问路径/自定义访问路径 a.action）

req.**socket.remoteAddress** 远程访问地址 req.**socket.remotePort** 远程访问端口

req.**query** url 请求参数

res 响应对象 res.**write**('xxx') 设置响应体 res.**end**('**xxx**'/**data**) 结束响应并发送（内容只能是字符串或读取文件的二进制数据 data）

res.**statusCode**=302 设置状态码

res.**setHeader**('Content-Type', 'text/plain; charset=utf-8') 设置响应头 res.**json**({"a":"b"}) 将对象解析为 j 串并发送给客户端

app.**listen**(**3000**, **function**(){}) 绑定端口号并启动服务器（可以开启多个不同端口服务，关闭 cmd 则关闭服务）

https

```
https.get('https://www.baidu.com', res=>{
        res.on('data', ()=>{})    //接收流数据
        res.on('end', ()=>{})     //数据接收完毕
}).on('error', (err)=>{})
```

http-proxy

Proxy Server Options

target url string to be parsed with the url module

changeOrigin true/false, Default: false - changes the origin of the host header to the target URL

[@types/http-proxy]

```
declare class Server<TIncomingMessage = http.IncomingMessage, TServerResponse = http.ServerResponse> extends
events.EventEmitter {
static createProxyServer<TIncomingMessage = http.IncomingMessage, TServerResponse = http.ServerResponse>(options?:
Server.ServerOptions): Server<TIncomingMessage, TServerResponse>;
```

Create an HTTP proxy server with an HTTPS target

```
httpProxy.createProxyServer({
  target: {
    protocol: 'https:',
    host: 'my-domain-name',
    port: 443,
    pfx: fs.readFileSync('path/to/certificate.p12'),
    passphrase: 'password',
  },
  changeOrigin: true,
}).listen(8000);
```

fs

var fs = require('fs') 读取文件[异步操作需要配合回调函数使用 callback]

fs.**readFile**('**path**', '**utf-8**', **function**(**error, data**){}) 读取文件（读取成功 data 为二进制数据，读取失败 error 为对象）【'**utf-8**' 自动转换将二进制为字符串，data.toString() 转为字符串】

fs.writeFile('path', 'content', function(error){}) 新建文件

fs.**readdir**('**path**', **function**(**err, files**){}) 读取主目录，返回 files 数组（文件名+目录名）

fs.**existsSync**('path') 路径是否存在

fs.lstatSync('path') 返回文件信息对象 stats

fs.stat("./wenjian.txt", function(err,stats){ }) stats.size 文件的大小； stats.atime 文件最后一次访问的时间对象 stats.birthtime 文件创建的时间

stats.isFile() 是否是文件 stats.isDirectory()是否是目录

os

var os = require('os') 系统

os.**EOL** 定义了操作系统的行尾符的常量

os.**cpus**() 获取当前机器 cpu 信息

os.**totalmen**() 获取当前机器内存大小-byte

path

var path = require('path')　　路径：（各个模块里的文件相对路径是相对执行 node 命令时的终端路径，命令执行位置不同会有误差，需使用拼接绝对路径）

__dirname　当前文件 所属目录的绝对路径---动态获取（常用为__dirname+'path' ）　【导入 path 后可全局直接使用】

__filename　当前文件绝对路径（动态获取）

path.parse('path')　　将路径解析为对象并返回（root 根目录，dir 目录路径，base 含后缀文件名，name 纯文件名，ext 后缀名）

path.extname('xxx')　　获取路径扩展名

path.basename('path', ['.js'])　获取路径文件名含后缀名（可指定去掉的后缀类型）

path.dirname('C:/a/b/xxx.js')　获取路径目录部分（C:/a/b）

path.isAbsolute('path')　　判断是否为绝对路径

path.basename('C:/a/b/xxx.js')　获取基础路径 xxx.js

path.resolve('dist')　　拼接到当前路径 形成绝对路径，多个参数时，会像 cd 一样向前移动　// path.resolve('dist') --> D:\DevWeb\lef\dist

path.join(__dirname 或'C:/a', '/img/so', '/1.jpg') ===　require.resolve('/img/so', '/1.jpg')　拼接多个路径为绝对路径 （自动去掉前面或后面重叠的斜杠）【require.resolve 自动添加__dirname】

url

var url = require('url')　网址

var obj=url.parse('http://ak.com/pinglun?a=11&b=22' ,[true]) 分解请求的内容，true 将查询字符串转为对象

dns

var url = require('dns')　域名服务器

process

var process=require("process")

const spawn = require('child_process').spawn;

process.versions　　node 版本信息　{ node: '12.10.0', v8: '7.6.303.29-node.16', unicode: '12.1'}

process.version　　node 版本　　v12.10.0

process.env　　node 环境信息

process.exit()　　退出进程

process.cwd()　　会返回 Node.js 进程的当前工作目录

process.chdir("path")　变更 node 的工作目录，失败抛出异常

process.execPath　　返回启动 Node.js 进程的可执行文件的绝对路径名　'/usr/local/bin/node '

```
const touch = spawn('touch',['spawn.js']);
touch.stdout.on('data', (data) => {
 console.log(`stdout: ${data}`);
});

touch.stderr.on('data', (data) => {
 console.log(`stderr: ${data}`);
});

touch.on('close', (code) => {
 console.log(`child process exited with code $[code]`);
});
```

child_process

import child_process, { spawn, execSync } from "child_process"　【spawn 推荐使用 cross-spawn，内置执行命令会有错误】

execSync("npm --version", {　　创建同步进程，执行系统命令

stdio: 'ignore' }).toString()	不输出命令信息
Custom Modules	
var exp=**require**('./xxx.js') 行一次，/==C:/表示磁盘根路径)	加载其他 js 文件内容—可省略扩展名（返回其他文件的 exports 对象，其他文件里重复加载时只执