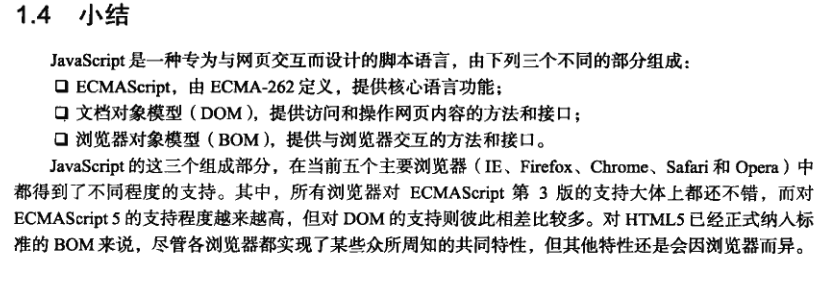
# 2019.3.20**( ES5 )**

## JavaScript的组成



# 2019.3.21

### 关于<script>的引用与属性

**<head>**

**<title> script脚本测试 </title>**

**<script defer="defer" type="text/javascript" src="test.js" ></script> <!-- 外部引入式写法 -->**

**<!--**

**关于script属性:**

**defer="defer": 等加载完毕html文档在解析js脚本( 外部引用有效 )**

**async="async": 异步解析js脚本,并加载html页面( 外部引用有效,有此属性不要修改DOM )**

**type="text/javascript": 约定必备**

**src="js路径": 引入外部js脚本,可本地,可远程( 有src属性时,内部嵌入内容会忽略,只执行外部引入内容 )**

**-->**

**<script type="text/javascript" > // 内部写入方式**

**alert("233");**

**</script>**

**</head>**

#### 关于CDATA避免XML解释器在“<”和”&”出现错误

**<!-- <![CDATA[ 代码内容 ]]> 防止"<"和"&"在XML解释器,内入式js代码出现错误 -->**

**<!-- 如果浏览器不支持<![CDATA[ ]]>,则注释掉如下( 即可支持所有浏览器 ): -->**

**<script type="text/javascript" >**

**//<![CDATA[**

**function abs(a,b){**

**if(a < b){**

**alert("hello");**

**}**

**else{**

**alert("NONO");**

**}**

**}**

**abs(1,2);**

**//]]>**

**</script>**

### 如果浏览器不支持js

#### 隐藏js代码，如果浏览器不支持js则会显示出内入式的js代码，影响html的内容排序

**<title> script脚本测试 </title>**

**<!-- 如果浏览器不支持js,隐藏js代码方式为( 用html方式注释掉 ): -->**

**<script type="text/javascript" >**

**//<!--**

**function abs(a,b){**

**if(a < b){**

**alert("hello");**

**}**

**}**

**abs(1,2);**

**//-->**

**</script>**

#### <noscript>不支持/禁用 - js进行提示

**<!--**

**关于<noscript></noscript>**

**如果浏览器:**

**1. 不支持js**

**2. 禁用js**

**满足以上1点,则<noscript>可以让用户看到一条消息**

**正常情况下,用户是无法看到noscript信息的**

**-->**

**<noscript>**

**<p>**

**此页面需要js支持,(启用)js.**

**</p>**

**</noscript>**

# 2019.3.22

### // 关于js的严格模式

**function test0(){**

**"use strict" // 严格模式,尊循ES3**

**var massage; // 可以赋给任何类型数据**

**alert(massage);**

**}**

### 关于全局变量与局部变量

**// 全局变量与局部变量**

**var mas0; // 全局变量**

**function test(){ // function定义函数**

**var mas1=101,mas2="hello"; // 局部变量**

**mas3 = 100; // 全局变量**

**}**

**test(); // 函数调用**

### typeof操作符( 非函数 ),可以判断数据类型

**// typeof操作符( 非函数 ),可以判断数据类型**

**/\***

**\* undefined: 未定义值,变量为赋值**

**\* boolean: 布尔值**

**\* string: 字符串类型**

**\* number: 阿拉伯数字**

**\* object: 对象或null**

**\* function: 函数类型**

**\*/**

**var a = "he",b;**

**alert(typeof a); // typeof是操作符,非函数,所以不加括号也可以**

**alert(typeof(a)); // 当然加括号也可以**

**alert( typeof null ); // object类型( Safari 5/Chrome7 之前浏览器会返回function,其它浏览器返回object )**

**alert( typeof b ); // undefined类型**

### // 关于null与undefined的关系

**// null为一个空指针**

**alert(typeof null); // 返回object对象**

**alert(null == undefined); // true,注意虽然他们有这个关系,但是应用方向却大不同**

#### // 布尔类型，Boolean()函数可以将变量转为布尔类型值

**var mes = "hello";**

**alert("---")**

**alert( Boolean(mes) ); //** **Boolean()函数可以将变量转为布尔类型值**

**/\***

**\* 注意:布尔类型 "true/false" 不等于 "1/0"**

**\* object true/任何对象 false/null**

**\* undefined 无真,只与false**

**\*/**

**if( mes ){ // 可以这样使用**

**alert("mes为真");**

**}**

#### // Number类型

**// 关于8进制**

**/\***

**\* 8进制以0开头,如果超过(0~7)则变为10进制,会忽略开头的0**

**\* 在严格模式下,8进制写法会报错**

**\*/**

**var octalNum0 = 06; // 8进制,0开头**

**var octalNum1 = 08; // 10进制,因为超出(0~7)**

**// 关于16进制**

**/\***

**\* 16进制0x开头(0~9/A~F)**

**\* 字母不区分大小写**

**\*/**

**var hexNum0 = 0xA; // 16进制,0x开头**

**var hexNum1 = 0xa; // 二者相同,不区分大小写**

#### // 浮点数类型

**/\***

**\* 如果点后,为0则依然认为是整数类型**

**\* 特别注意: 0.1+0.2 == 0.30000000000000004 // 这是个舍入误差,很多编程语言都存在这个问题**

**\* 0.12+0.3 == 0.42 // 这个到是没有问题**

**\*/**

**var floatNum0 = 10.0; // 依然会被认定为int类型"10"**

**// 不推荐写法**

**var floatNum1 = 1.;**

**var floatNum2 = .1;**

**alert(floatNum2); // 0.1**

**// 关于e/E计算方法**

**/\***

**\* e与E相同,不区分大小写**

**\* 例: 3.12e4 == 3.1200.0**

**\* 3.12e4 相当于: 3.12\*10^4 == 3.1200.0**

**\*/**

**var floatNum3 = 3.12e4; // 3.1200.0**

**var floatNum4 = 3e-3; // 数学计算, 3\*10^(-3) == 0.003**

**alert( floatNum4 ); // 0,003**

### // 数值范围，isFinite()查看值是否在最大最小值内

**/\***

**\* 最大值 Number.MAX\_VALUE == 1.7976931348623157e+308**

**\* 最小值 Number.MIN\_VALUE == 5e-324**

**\* isFinite() 查看数是否在最大最小范围内,返回布尔类型**

**\* 如果值超出范围,则数值变为正/负Infinity( 正负无穷 )**

**\*/**

**var result = Number.MAX\_VALUE + Number.MAX\_VALUE; // Number.MAX\_VALUE保存着最大值**

**alert( isFinite(result) );** **// isFinite()查看值是否在最大最小值内**

**alert(result);**

**alert( Number.NEGATIVE\_INFINITY ); // Number.NEGATIVE\_INFINITY/Number.POSITIVE\_INFINITY == -Infinity/+Infinity**

### // 非数值NaN

**/\***

**\* NaN为非数值**

**\* 数值/0 返回NaN,保证不出错**

**\* NaN/10 == NaN // NaN与如何数计算都等于NaN**

**\* NaN != NaN // 甚至它都不等于他本身**

**\*/**

**alert("----");**

**var a=10,b=0;**

**alert(a/b); // 默认情况是NaN,Chrome则为Infinity**

**alert(NaN/10); // 依然为NaN,NaN除任何数都为NaN**

# 2019.3.28

### // isNaN() 判断是否为非数值,返回( 非数值为真,数值为假 )

**var a = 10, b = true, c = "abv",d = "123" ;**

**alert(isNaN(a)); // false**

**alert(isNaN(b)); // false: 布尔值,可以转换为1/0**

**alert(isNaN(c)); // true: 字母或符号字符串,非数值**

**alert( isNaN(d) ); // false: 字符串数字,则也可以转换为数值**

### // 数值转换,Number(),parseInt(),parseFloat() 转换数字类型

**var num1 = true, num2 = null;**

**var str0 = "", str1 = "012", str2 = "0xa", str3 = "13abc", str4 = "abc";**

**var str5 = 12.3,str6 = 1.2e3;**

#### **// Number()转换数字,有很多规矩,一般不推荐使用**

**// 非字符串类型**

**// 可以字符串-转-整数类型-或-浮点类型**

**alert(Number(num1)); // 1: 布尔类型转换为1/0**

**alert(Number(num2)); // 0: null空类型返回0**

**// 字符串类型**

**alert(Number(str0)) // 0:空字符串返回0**

**alert(Number(str1)) // 12:会忽略前数字的0**

**alert(Number(str2)) // 10:会16进制值-转换- 对应的10进制值**

**alert(Number(str3)) // NaN:非纯数字会认定为非数值**

**alert(Number(str4)) // NaN:非数值**

#### **// parseInt( 数据,进制类型 ) 转换为整数类型,推荐使用,更加合理化**

**var num1 = true, num2 = null;**

**var str0 = "", str1 = "012", str2 = "0xa", str3 = "13abc", str4 = "abc";**

**var str5 = 12.3,str6 = 1.2e3;**

**// 1. 无数字,都为NaN**

**// 2. 开头数字解析,非数字停止解析**

**// 非字符串类型**

**alert(parseInt(num1)) // NaN:非数值**

**alert(parseInt(num2)) // NaN**

**// 字符串类型**

**alert(parseInt(str0)) // NaN**

**alert(parseInt(str1,10)) // 12:认定内容为10进制,忽略0,输出10进制数值**

**alert(parseInt(str2,16)) // 10:认定内容为16进制-转换-10进制数值**

**alert(parseInt(str3,10)) // 13:认定为10进制,开头遇到数字则继续解析,遇到非数字则停止解析,输出10进制数值**

**alert(parseInt(str4)) // NaN:非数值**

#### **// parseFloat( 数据 ) 转换浮点类型**

**// 1. 与parseInt()规矩差不多,只不过无指定进制参数**

**alert(parseFloat(str5)) // 12.3**

**alert(parseFloat(str6)) // 1200 // 科学计算法也没有问题**

### **// Object类型**

**// 目前解析: 一组数据和功能的集合**

**var a = new Object(); // 创建Object类型**

#### **// 操作符**

**// 目前解析: 操作数据,如+-\*/,关系操作符**

**// 操作对象会调用valueOf()/toString()方法,方便获取值( valueOf()此方法不太理解 )**

### **// 字面量**

**// '\b': 空格**

**// '\r': 回车**

**// '\x': 16进制数**

**// '\u': Unicode字符**

**var str0 = "\u03a3"; // Unicode字符码**

**alert(str0);**

#### **// .length字符串长度,注意: 如果有双字节字符,长度测量会不准确**

**var str1 = "abc";**

**alert(str1.length); // 3:字符串长度数量**

#### **// 字符串相加**

**var str2 = "java";**

**str2 += "script"; // 字符串相加**

**alert(str2);**

### **// 数据.toString( 指定转换进制 )方法, String( 数据 )函数,字符串转换**

**// toString()方法可以指定转换进制**

**// 1.null类型时,toString()会报错**

**var str3 = 10, str4 = null, str5 = true;**

**alert(str3.toString()); // "10":直接转字符串类型**

**alert(str3.toString(16)); // "a":数据-转换-为指定进制,转变为字符串类型**

#### **// String()靠谱的字符串转换,**

**// 他与toString()区别在与,**

**// 1. 没有办法指定进制转换,**

**// 2. 但是String()兼容性更强,他可以兼容非实数类型转换成字符串类型**

**alert(String(str4)); // "null":字符串类型**

**alert(String(str3)); // "10":直接转不解释**

# **2019.3.30**

### // 一元操作符

// 只能操作一个值,叫一元操作符

**// --age/++age形式**

**var age = 29;**

**var other = --age + 2; // 先执行age = age -1; 在执行 other = age + 2;**

**alert( other ); // 30**

**alert( age ); // 28**

**// age++/age--形式**

**age = 29;**

**other = age-- + 2; // 先执行 other = age + 2; 在执行 age = age -1**

**alert( other ); // 31**

**alert( age ); // 28**

#### **// 一元操作符(++/--),javascript不严谨性计算( 个人强烈不推荐计算 )**

**// 1. 字符串类型++ ( 字符串转换成实数进行数值实数( 有字母或其它符号,则返回NaN类型 ) )**

**// 2. 布尔类型++ ( 会把布尔类型当成1/0进行数值计算 )**

**// 3. 浮点型++ ( 正常的数值计算,不过注意舍入BUG )**

**// 4. Object类型++**

**// 例:**

**// 字符串++;**

**var a1 = "2";**

**a1++;**

**alert(a1); // 3: 被转成实数类型**

**alert(typeof a1);**

**// 布尔类型++**

**var a2 = true;**

**a2++;**

**alert(a2);**

**// Object类型++**

**var o = {**

**valueOf: function(){**

**return -1;**

**}**

**}**

**o++;**

**alert(o);**

#### // 一元加减符(+/-)说白了就是数学里的正负号

**// 1. 一样有上方一元操作符计算不严谨性**

**// 2. 区别在与遇到实数时,就相当于+/-号处理**

**//例:**

**var a3 = "1";**

**alert(+a3); // 3:被转换成了实数**

**alert(typeof +a3);**

# 2019.4.2

### // 32位存储-转-二进制码

// 默认32位,有一位表示正/负

// 0表示正数,1表示负数,所以32位真正存储数字的有31位

**var num = -18;**

**alert( num.toString(2) ); // 更加人性化的转换显示**

### **// 按位对比变换**

**// 总体围绕32位二进制数进行推演**

#### **// 1. 按位非(NOT)**

**// 1变0,0变1**

**// 如:**

**var num2 = 25; // 二进制: 00000000 00000000 0000000 00011001**

**var num3 = ~num2 // -26: 二进制: 11111111 11111111 11111111 1100110**

**alert(num3);**

#### **// 2. 按位与(AND)**

**// 变换规则:**

**// 1 : 1 = 1**

**// 1 : 0 = 0**

**// 0 : 0 = 0**

**// 如:**

**var num4 = 25 & 3;**

**alert(num4);**

**// 25: 00000000 00000000 00000000 00011001**

**// 3: 00000000 00000000 00000000 00000011**

**// 对比后: 1: 00000000 00000000 00000000 00000001**

#### **// 3. 按位或(OR)**

**// 变换规则:**

**// 1 : 1 = 1**

**// 0 : 1 = 1**

**// 0 : 0 = 0**

**// 与上对比变换类似**

**Var num5 = 25 | 3;**

#### **// 4. 按位异或(XOR)**

**// 变换规则:**

**// 1 : 1 = 0**

**// 0 : 1 = 1**

**// 0 : 0 = 0**

**// 规则,二位比较中只有一个1时才返回1**

**var num6 = 25 ^ 3;**

**alert(num6);**

# **2019.4.3**

### // 位移符号( 直接操纵二进制位数 )

// 左移(<<),右移(>>):

// 注意: <</>>对正负没有影响,因为他们真正只操控31位,正负位不进行操纵

// 无符左移(<<<),无符右移(>>>):

// 注意: <<</>>>直接操控32位,因此会影响负数差距会非常大,不过无符位数用负数,也是他喵的神经病

**// 左移( << )**

**var test = 123;**

**alert(test.toString(2)); // 123 : 0 0000000 00000000 00000000 01111011**

**var test1 = test << 5; // 左位移**

**alert(test1.toString(2)); // 3936 : 0 0000000 00000000 00001111 01100000 :可以发现,在"左移"在末尾插入了5位0**

**alert(test1); // 3936**

**// 右移( >> )**

**var test2 = 12;**

**alert(test2.toString(2)); // 12 : 0 0000000 000000000 000000000 00001100**

**var test3 = test2 >> 3; // 右位移**

**alert(test3.toString(2)); // 1 : 0 0000000 000000000 000000000 00000001 :可以发现,在"右移"在开头插入3个0**

**alert(test3); // 1**

#### **// 无符左/右移动,直接操控32位,原理于上方类似**

**// 无符左移<<<**

**// 无符右移>>>**

**Var test4 = 123;**

**Var test5 = test4 >>> 3; // 直接操控32位，在开通加3零在结尾去除超出位数，即是位移结果**

### // 布尔操作符

#### // 逻辑非 ! --- 其实就是真变假,假变真

其实就是真变假,假变真

#### // 逻辑或 ||

// a || b

// 1. 如果第一个数为真则不会继续执行,与C语言逻辑或几乎相同

#### // 逻辑与 &&

// a && b

// 1. 如果第一个数为假则不会继续执行,与C语言逻辑与几乎相同

#### // 乘/除性操作符( 其实就是" \*,/ "数学的乘号 )

// 1. 正常数值计算相乘/除

// 2. 如果一个操作数是非数值,则在后台调用Number()函数转换为数值后在进行计算

// 3. 关于Infinity细节看原书p66

#### // 模( % )

// 1. 左边值 < 右边值 返回 左边值

**alert( 5%23 ); // 5**

// 2. 其实求的是余数( 与C语言的%相同 i)

**alert( 23%5 ); // 3**

// 3. 遇到非数值先使用Number()转换为数值,在进行计算

// 4. 关于Infinity细节看原书p67

#### // 加性操作符（优先转为字符串进行拼接）

// 1. 字符串+字符串 = 进行字符串拼接

// 2. 非实数字符串+数值 = 数值转换字符串，进行字符串拼接

// 3. 如果操作的是对象,或者布尔类型等非数值,则后台会进行toString()/String()转换为字符串在进行字符串拼接

// 4. 实数字符串+数值 = 字符串转换为数值进行计算

**// 如:**

**var a = 1,b = 'b',c='c',d=true,e=’2’;**

**alert(b+c); // "bc": 字符串拼接**

**alert(a+b); // "1b": 先将数值转换为字符串在进行字符串拼接**

**alert(d+b); // "trueb": 非数值类型先进行转换,在进行字符串拼接**

**alert(a+e); // 3: 实数字符串+数值 = 字符串转换为数值进行计算**

#### // 减性操作符（优先转为数值进行计算）

**看实例：**

**var a = 'a',b='2',c='1',d=5;**

**alert(a-b); // NaN: 非数值**

**alert(c-b); // -1: 字符串转数值进行计算**

**alert(d-b); // 3: 字符串转数值进行计算**

# **2019.4.7**

### **// 关系操作符( >,>=,<,<= )**

**// 关系操作符返回布尔值**

**// 并且拥有自动转换数据类型功能**

**// 比较对象时,对象先调用valueOf()或toString()函数进行数值转换,在进行比较**

**// 例:**

**var a = 1,b = 2,c = '3',d = 's',e = 'A';**

**a < b // true: 数值正常比较**

**d > e // true: 双方是字符串时,比较ascii码值大小**

**b > c // false: 数值与实数字符串比较,字符串先转为数值在进行比较**

**a < d // false: 数值与非实数字符串比较,字符串转为NaN类型( 与NaN比较,无论什么情况返回false )**

### **// toLowerCase()方法,字符串,大写转为小写形式**

****alert("ABCaaa".toLowerCase()); // "abcaaa"****

### **// toUpperCase()小写转大写字母**

**alert("ABCaaa". toUpperCase ()); // "ABCAAA"**

### **// 相等操作符( !=,==,===,!== )**

**// 相等操作符同样返回布尔类型**

**// 并且拥有,自动转换数据类型的功能( 除===,!== )**

**// 注意:**

**// NaN与如何数据类型比较都返回false**

**// 布尔类型会转为1/0**

**// 在进行比较时强烈推荐使用===或!==( 推荐使用 )**

**// 例:**

**alert( "5"==5 ); // true: 自动转换类型**

**alert( "5"===5 ); // false: "==="比较数值和类型是否一直,显然二数类型并不一致**

**alert( "5"!=5 ); // false: 因为字符串先自动转换为数值在进行比较**

**alert( "5"!==5 ); // true: 类型不一样**

### **// 条件操作符( 其实就是"C语言里的三元表达式" )**

**var b1,b2=true;**

**b1 = b2 ? 2 : 3; // 如果b2为真,2赋值给b1,否则3赋值给b1**

**alert(b1); // 2**

### **// 赋值操作符**

**// 快捷写法:**

**// 这种写法适用与:**

**// \*=**

**// /=**

**// %=**

**// -=**

**// +=**

**// <<=**

**// <<<=**

**var num = 10;**

**num += 5; // num = num + 5;**

**alert(num); // 15:**

### **// 逗号操作符**

**var a=1,b=2; // 逗号表达式可用于多个变量赋值**

**var num = (1,2,3,4); // 4: 逗号表达式会将最后值赋给变量**

**alert(num); // 4**

## **// 语句( 和C语法几乎差不多 )**

### **// if语句( 老盆友了不多说 )**

**var i=0;**

**// 单行写法( 不推荐可读性差 )**

**// if( i>0 ) alert("yes") else if(i==0) alert("0") else alert("no")**

**// 规范写法**

**if( i>0 ){**

**alert("yes");**

**}**

**else if( i==0 ){**

**alert("0");**

**}**

**else{**

**alert("no");**

**}**

### **// do while语句( 与C几乎差不多 )**

**var s = 0;**

**do{**

**s++;**

**}while( s<10 );**

**alert(s);**

### **// for语句( 老朋友不多说 )**

**var i,count = 3;**

#### **// 规范写法**

**for(i=0;i<count;i++){**

**alert(i);**

**}**

#### **// 无限循环写法**

**/\***

**for(;;){**

**// 内容**

**}**

**\*/**

#### **// for也可以代替while写法**

**i = 0;**

**for(;i<conut;){**

**i++;**

**}**

#### **// for-in 用法( 可以迭代出对象属性,用法有点像python的for-in语句 )**

**// 1. 对象属性是无序的,因此每一次迭代出的属性是随机幻化的,但每一个属性都会被迭代出一次**

**// 2. 如果迭代出的属性变量值是null或undefined,则会被报错,ES5标准则会停止迭代**

**/\***

**for( var propName in window ){ // 迭代出window对象所有属性名,并赋值给propName变量**

**document.write(propName); // 在浏览器中写出变量内容**

**document.write("<br>"); // 写换行符**

**}**

**\*/**

#### **// label语句( 跳出循环使用 )**

**// 通常与for循环配合**

**// break label 或 continue label方法使用**

**// 注意: break label 直接结束目标for的循环**

**// continue label 跳到标签点继续执行**

**// 例:**

**/\***

**var i,a,b;**

**labelEnd: for(i=0;i<5;i++){**

**alert("1-1-1");**

**labelState: for(a=0;a<3;a++){ // label标签用法( 标签点 )**

**alert("2-2-2");**

**for(b=0;b<2;b++){**

**alert("3-3-3");**

**continue labelState; // 直接跳出到标签点,然后继续执行**

**}**

**}**

**break labelEnd; // 结束labelEnd标签点匹配的for循环**

**}**

**// labelEnd 结束后,其它语句开始执行位置**

**alert("end!");**

**\*/**

**// break,continue 功能与C语言语法功能一样**

### **// with语句**

**// 主要作用是将,作用域设置到指定对象中**

**// with会使代码性能降低,不推荐使用**

**// with也不方便调试,并且with无法在严格模式使用,会直接报错**

# 2019.5.11

### 调用[Unicode](https://baike.baidu.com/item/Unicode" \t "_blank) 码

var a = "\u00e8"; // 调用u码

alert(a);

### // 关于switch的用法

**// 1. js的switch于c的功能用法几乎差不多,**

**// 唯一的不不同点在于,它可以判断,更多的数据类型**

**// 2. 可以代替多个if else if判断**

**// 3. 二个case之间如果不写break一定要写注释**

**// 4. switch使用的===全等符判断，所以不会转换数据类型( 我喜欢 )**

**var test = "233";**

**var test2 = 233;**

**switch( 123 ){**

**case "123":**

**alert("123");**

**// 这里故意不写break**

**case test2: // 支持多种类型判断,甚至是变量**

**alert(test2);**

**break**

**default: // 经证实,default没有break依然会往下执行**

**alert( "都没有" );**

**case "233":**

**alert(test);**

**break**

**}**

### **// 关于函数的用法**

**// 1. 老朋友不多解释**

**// 2. 要么有返回值，要么不写返回值**

**function abs( name,mes ){**

**alert(name+mes);**

**return false;**

**}**

**abs("zhao","233");**

#### **// 函数的参数调用**

**// 1. js因为没有函数签名,所以重载,只能进行模仿,**

**// 如检测传入的参数数量,进行判断执行**

**// 2. 其实这些插入的参数是以数组形式传送的,所以即使超出所需值，也可以通过arguments调用**

**// 3. 非严格模式下,通过arguments数组可以改变参数,严格模式下将无法改变,默认值为undefined**

**// 4. 核心: arguments**

**function argu( name,phone){**

**alert( arguments[0]+arguments[1] ); // 索引对应验证**

**arguments[2] = "OK!"; // 非严格模式甚至可以改变传入的参数**

**alert(arguments[2]);**

**alert( arguments.length ); // 查看输入参数数量( 使用length方法测取数组长度 )**

**}**

**argu("zhao",1385,1069);**

#### **// 重载模拟**

**// 1. js是默认是没有重载的，不过可以模拟出来**

**// 2. 何为重载?**

**// 重载，简单说，就是函数或者方法有相同的名称，但是参数列表不相同的情形，这样的同名不同参数的函数或者方法之间，互相称之为重载函数或者方法。**

**function cz( ){**

**if( arguments.length == 1 ){ // 传入1个参数时调用**

**alert( arguments[0]/2 );**

**}**

**else if( arguments.length == 2 ){ // 传入2个参数时调用**

**var result;**

**result = arguments[0] + arguments[1];**

**alert( result );**

**}**

**}**

**cz(22);**

**cz(22,22)**

# **2019.5.14**

## **// 变量与作用域和内存问题**

**// 0. 基本类型: undefined,null,boolean,number,string 5个基本类型**

**// 1. 引用类型: object(),引用类型不允许直接访问内存,在操作过程中为引用访问**

**// 2. new Object() 创建对象**

### **// 0. 引用类型可以被赋予属性,而非引用类型则不可以**

**var person = new Object(); // 引用类型**

**person.name = "Zhao";**

**var dog = "Li"; // 基本类型**

**dog.color = "pink"; // 虽然不会出错,但是确实没什么用处**

**alert( person.name ); // "Zhao"**

**alert( dog.color ); // "undefined"**

#### **// 1. 基本类型,变量复制( javascript特立独行 )**

**var test1 = 250;**

**var test2 = test1; // 在javascript中,这样已经算是复制了！！！( C语言可与它不一样 )**

**alert(test2); // 250**

**test1 = 300; // 上放因并非指针指向,已经属于复制**

**alert(test2); // 250**

#### **// 2. 引用类型,变量复制**

**var test3 = new Object();**

**test3.num = 15;**

**var test4 = test3; // 其实并没有真正的复制,只是指针指向**

**alert(test4.num);**

### **// 3. 传递参数**

**// 0. 按值传递( 真正的复制 )**

**// 1. 引用类型值传递( 指针引用 )**

**// 2. 参数只能'按值传递'**

#### **// 4. 按值传递与局部变量**

**function addnum( num ){**

**num += 10; // 局部变量( 老相识，不必多解释 )**

**return num;**

**}**

**var num = 10;**

**var result = addnum( num );**

**alert(num); // 10**

**alert(result); // 20**

#### **// 5. 伪引用类型值传递，全局变量,在局部变量的反应**

**// 0. 其实,全局变量可以在局部中使用，**

**// 1. 但是局部变量不可以在全局使用,函数一旦结束执行，局部变量也会立刻销毁**

**function setName( obj ){**

**obj.name = "TENG"; // 函数的局部环境可以使用'全局变量'**

**obj = new Object(); // 其实创建的是局部变量**

**obj.name = "TTTT"; // 因为局部变量，在函数结束执行后会立刻销毁**

**}**

**var obj = new Object();**

**setName(obj);**

**alert(obj.name); // "TENG"**

### **// 6. 检测,引用类型/基本类型**

**// 0. instanceof 专门用来检测引用类型,能精确到引用类型什么形式**

**// 1. 引用类型返回true**

**// 2. 基本类型返回false**

**var test5 = new Object();**

**var test6 = 15;**

**var test7;**

**alert( test5 instanceof Object ) // true**

**alert( test6 instanceof Object ) // false**

**alert( test7 instanceof Array ) // false**

# 2019.7.29

## Udemy-js-学习

### // 0. js-使用键盘输入字符串 - prompt()默认键入为string类型

var Q = prompt( 'what is his last name?' );

alert( typeof(Q) ); // 键入值为String类型

console.log( Q );

### // 1. 大于小于符号演示演示

var a=1,b=2;

console.log( a>b );

console.log( typeof( a>b ) );

### // 2. if 语句练习

age = 12;

if( age<18 ){

console.log(" 禁止访问 ");

}

else if ( age>=18 && age <30 ){

console.log(" 访问成功 ");

}

else{

console.log(" 查无此页面 ");

}

### /\*\* 3元表达式 \*\*/

// 一遍用于,简单的判断赋值，不适于复杂的判断

// 使用方法:

// 条件 ? 真,执行此语句 : 假,执行此语句;

var drinks;

// var age = prompt(" 你的年龄? ");

age >= 18 ? drinks = "wine" : drinks = "joice";

console.log( "ZTaer drinks " + drinks + "!" );

### /\*\* switch \*\*/

// 适用于,避免使用多个if else,使程序简洁性

// js的switch很自由,case的条件按照if的判断条件来就好

var man;

switch( true ){

case age < 18 && age >= 16:

man="青少年";

break; // 没有break后果很严重,会不经case处理就执行其他语句

case age >= 18:

man = "成年人";

break;

case age == "21": // case 也可以多条配合,来执行同一条语句

case age == "二十一":

man="21岁";

break;

default: // 如果条件都不符合会执行此语句

man = "你猜一猜";

break;

}

console.log( "ZTaer 已经是 " + man + "!" );

### 获取数组长度，.length

for( i=0; i < join.length i++ ){ // 获取数组长度

var ave = sum\_ave( join[i] )[1]; // 获取数组平均值

if ( ave > winner ){

winner = ave;

pos = i; // 记录最大数位置

}

}

### /\*\* 关于数组 \*\*/

#### // 创建数组

var name = ['abc','sss'];

var year = new Array( '2018','2019' );

#### // 操控数组元素,添加数组元素,删除数组元素,确定数组元素位置

year[ year.length ] = '2020'; // 在末尾添加元素,length可以确定数组中的数量

year.push('8848'); // 在数组末尾添加

year.unshift( 'ztaer' ); // 在数组开头添加元素

console.log(year);

year.pop(); // 删除数组末尾元素

year.shift(); // 删除数组开头元素

console.log(year);

console.log( year.indexOf('2019') ); // 确定数组元素位置，如果数组中无目标元素,则返回"数字类型-1"

### /\*\*\* Object对象 \*\*\*/

#### // 创建对象 - 方法1

var ztaer = {

year: 18,

name: 'oo7',

skills: [ 'h5/css3','js','python3','linux' ],

gf: false

};

var oo7 = new Object(); // 创建对象 - 方法 2

#### // 索引对象属性

console.log( ztaer.year );

console.log( ztaer[ 'name' ] );

console.log( ztaer.skills[1] );

#### // 修改对象属性

ztaer.year = 20;

ztaer['name'] = '\_\_OO7\_\_';

console.log( ztaer );

#### // 添加属性给对象

oo7.name = 'ztaer';

oo7['sex'] = 'man';

console.log( oo7 );

### /\*\*\*\*\* 函数 \*\*\*\*\*/

#### // 函数的创建 - 第一种

function test( x ){

return x;

}

#### // 函数的创建 - 第二种（ 函数表达式 ）

test2 = function( y ){

return y;

}

console.log( test(2) + ' ' + test2(3) );

#### /\*\*\*\*\* object关于this使用，以及对象中函数创建 \*\*\*\*\*/

// object的函数创建其实依据第二种函数创建方法

// this使用:

// a) this.变量，可以调用自身对象属性

// b) this.变量。 也可以创建自身属性

var boy = {

birthYear: 1998,

calcAge: function( year ){ // 创建函数

// this 可以调用自身已有属性，也可以创建新属性

this.age = year - this.birthYear; // 创建新属性

return this.age;

}

}

// 对象函数调用

console.log( boy.calcAge( 2019 ) );

### 关于.toFixed()保留小数,返回字符串类型

// .toFixed(2) 四舍五入保留2位小数,返回字符串类型结果，所以计算时需在转换为数字类型

var userMark = {

name:'Mark',

cale: function( mass,height ){

this.mass = mass;

this.height = height;

this.BMI = +( mass / height \*\* 2).toFixed(2) ; // .toFixed()保留二位小数返回字符串结果，+号字符串转为数字类型数据

return this.BMI;

}

}

/\*\*\*\*\*\*\* MAIN \*/

var winner,BMI;

BMI\_Mark = userMark.cale( 72,1.8 ); // 调用对象属性函数

B

## 融合，数组，对象，最近所学函数实战看，2019.7.28/home.js

## /\*\*\*\* 几种循环，以及continue,break \*/

### // for循环/while与C无异

for( var i=0; i<10; i += 2 ){ // 花式输出偶函数

if( i > 5 ) break; // break与C无异

console.log(i);

}

### // while,按照C正常使用即可，continue/break同样如此

var group = [ '1','2',3,'4','5' ];

var i = 0;

console.log( typeof( group[2] ) )

while( i < group.length ){

if ( typeof group[i] !== 'number' ){ // typeof 验证数据类型并返回的是字符串类型数据

i++;

continue

}

else{

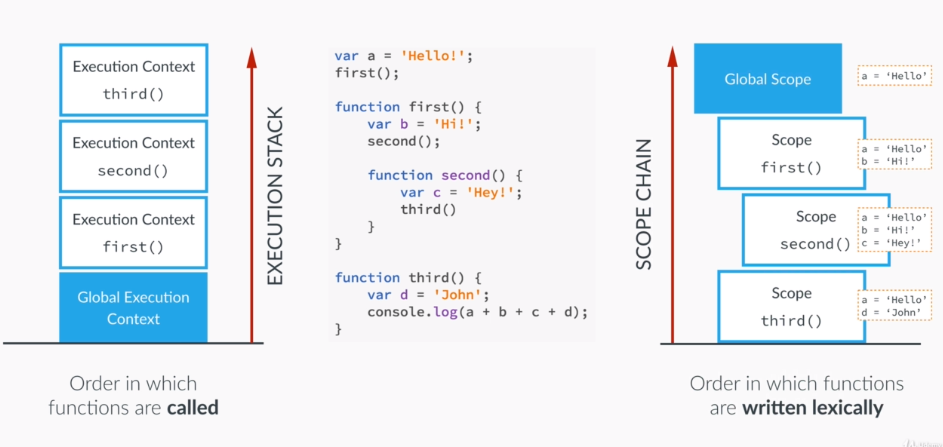
console.log( group[i] );

break;

}

}

## 堆栈的工作顺序



# 2019.8.2

## // 0. 函数与函数表达式 - 工作方式

// a) 函数与函数表达式工作方式是有区别的

// b) 函数声明会直接存放在堆栈中，是全文可以调用，但函数表达式，必须创建之后使用

### // 函数工作方式:

// 函数声明会直接存放在堆栈中，是全文可以调用，但函数表达式，必须创建之后使用

console.log( abs(5) );

function abs( x ){

return x\*\*2;

}

### // 函数表达式工作方式:

// console.log( avg(1,2) ) // 不可用，因为“变量”影响工作方式必须按照顺序执行

var avg = function( x,y ){

var result = ( x+y )/2;

return result;

}

## // 1. 变量工作方式

// a) 变量在全文中有定义但是顺序不正确，默认会赋值变量为undeifne，

// b) 如果是全文中都没用定义, 去调用则会出现错误

console.log(s); // 虽然顺序不正确，但确实有定义此变量，所以会返回undefined

var s = 5;

// console.log(bb); // 此变量没有定义，后返回错误的信息

## // 2. 关于this的细节

// a) 在对像中，this默认无法进行但级函数访问属性，最多只能一级函数访问，如下案列

// b) 其它变量 = this; this因规则受限无法多级函数使用，但是把他赋值给其它函数即可进行多级属性访问,如下案列

jone = {

name:'jone',

tel: 123456,

cale: function(){

console.log( this.name ); // 可正常访问

var that = this; // 将this赋值给that及时闭包依然可以代替this可以进行多级函数，直接属性访问

function test2(){

console.log( this.name ); // 无法访问,因为this的调用是有限制的，this只能进行一级访问。

console.log( that.tel ); // 正常访问，因为that变量单独已被this赋值，this是受默认规则的影响无法进行多级函数属性访问,但其它变量不受限。

}

test2();

}

}

jone.cale();

### // 3. 关于对象的细节，对象间借用函数

var zhao = {

name:'zhao',

cale: function(){

console.log(this.name);

}

}

var li = {

name:'zhao',

cale: zhao.cale() // 内借用

}

li.cale2 = zhao.cale; // 外借用

li.cale; // 调用借来的函数( 正常调用 )

## DOM操作 – 2019.7.29实战的游戏而总结

### /\*\*\*\*\*\*\*\*\*\*\*\* es5-生成随机数，以及转为整数

#### // 0. 生成0~1随机数( 与乘法配合生成其它随机数 )

Math.random();

#### // 1. 非四舍五入取整数

Math.floor();

// 例如:

var ran = 0;

ran = Math.floor( Math.random() \* 6 + 1 ); // 生成1~6的随机数

console.log(ran);

\*/

### /\*\*\*\*\*\*\*\*\*\*\*\* es5-document.querySelector()HTML标签操控

#### // 0. 选中标签

document.querySelector(' #score-0 '); // 以CSS选择器选中即可

#### // 1. 改变标签内容

// a) 改变标签文本内容: <a id="score-0" > 文本内容 <a>

document.querySelector(' #score-0 ').textContent = ran ;

// b) 在标签中插入HTML代码: <a id="score-0" > <b> $ran </b> </a>

document.querySelector('#score-0').innerHTML= '<b>' + ran + '</b>' ;

#### // 2. 获取标签内容

// a) 获取标签中的文本内容( 字符串类型 )

var test0 = document.querySelector('#score-0').textContent;

console.log( test0 );

#### // b) 获取标签中，HTML标签

var test1 = document.querySelector('.player-0-panel').innerHTML;

console.log( test1 );

#### // 3. 改变标签中CSS样式

// a) 其实核心document.querySelector('CSS选择器').style.css属性 = 'css样式参数';

document.querySelector('.dice').style.display = 'none';

#### // 4. 改变标签属性 - 改变图片实例

// a) 选中的标签.属性 = '修改值';

var choose\_img = 3;

var dice\_img = document.querySelector('.dice');

dice\_img.src = 'dice-' + choose\_img + '.png'; // 核心

\*/

### /\*\*\*\*\*\*\*\*\*\*\*\*\* es5-.addEventListener()监听事件，并执行函数

// 监听事件参数参考网站: <https://developer.mozilla.org/zh-CN/docs/Web/Events/>

// 目前发现，他只能监听按钮

#### // 0. 正常调用函数 - 监听单机事件,执行test0()函数

function test0(){

var dice = Math.floor( Math.random() \* 6 + 1 ); // 随机数

document.getElementById('score-0').textContent = dice; // 抓取ID标签

}

var btn\_roll = document.querySelector('.btn-roll').addEventListener('click',test0);

#### // 1. 调用匿名函数( 匿名函数就是没有变量名称的函数 ) - 监听单击时间，并执行函数

// a) .addEventListener('监听事件', function(){ 函数内容 } );

var btn\_roll = document.querySelector('.btn-roll').addEventListener('click', function(){

var dice = Math.floor( Math.random() \* 6 + 1 ); // 随机数

document.getElementById('score-0').textContent = dice;

} );

\*/

### /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* es5-.getElementById('test')抓取ID ，选中ID

// 0. 抓取ID, 抓取速度比.querySelector()更快，但是功能单一不方便

document.getElementById('test');

\*/

### /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* es5-.classList标签多类名操控，删除，添加，切换HTML标签类名

#### // 删除指定类名,多个类名用逗号隔开

document.querySelector('.player-0-panel').classList.remove('active','test','test1');

#### // 添加类名,多个类名同样用逗号隔开

document.querySelector( '.player-0-panel' ).classList.add('active','name');

#### // 切换类名，所为切换: 标签含有类名时删除,无类名时添加

document.querySelector('.player-0-panel').classList.toggle('active');

\*/

## Js-初始化规则

Js的初始化参数一遍用init()函数来命名

# 2019.8.5、

## // 继承的目的，以及使用概念（ 理论概念 ）

// 0. 目的是减少重复代码

// 1. 使用方法

// a) 创建一个蓝图函数，命名开头大写，易于区分蓝图对象于普通对象

// b) 蓝图的索引用调用，让新建的对象继承蓝图中的属性

## // 让对象拥有继承函数/属性成为可能的prototype( js的基本继承思路 )

// 原因( js中除原语外，万物皆对象 ):

// 0. 每一个建立的Object对象/函数都拥有prototype属性,所以我们才能在对象中调用一些函数和方法

// 1. 同理如果我们自定义的函数也加入到prototype中，在对象中同样可以调用我们自定义的函数

// 2. 对象在调用函数时，首先会在自身中搜索函数，如果没用则会去prototype中查找函数并调用，如果prototype中也没用则返回undefine

// 用于:

// 0. 常用于，与蓝图配合增加函数

// 1. 当然也可以增加属性，但是并不常用

## /\*\*\*\*\*\*\*\*\*\* 蓝图 - 对象继承使用方法,以及prototype使用方法

### // 创建蓝图,命名开头大写,易于于函数区分

var Parsen = function( name, age, address ){

this.name = name; // 注意加this.

this.age = age;

this.address = address;

}

#### // 使用蓝图创建对象

// 原理:

// a) new 运算符的作用是: 创建一个空对象

// b) 而蓝图为函数，加入this的变量被new操作为对象属性.

var zhao = new Parsen('zhao','21','大学嘉园');

var teng = new Parsen('teng','21','赵楼');

console.log( zhao,teng );

### // 使用prototype给蓝图对象添加自定义函数

#### // 可用于继承的原因:

// 0. prototype在每一个对象或者函数默认拥有的属性( 核心 )

// 1. prototype中包含官方自带的函数，所以我们才能在对象中使用

// 2. 同理如果将我们创建的自定义函数加入prototype属性中，那么也就可以在对象中调用我们的自定义函数

#### // 常用于:

// 0. 给对象增加函数,如蓝图

// 1. 也可以添加属性，但是并不常用

Parsen.prototype.cale = function(){

console.log( 100 - this.age );

}

zhao.cale(); // 调用函数

teng.cale();

\*/

## /\*\*\*\*\*\*\*\*\*\* 原型链细节补充，以及证明js皆对象

// 0. 在chrome中可以清楚的看到，js万物皆对象，除“ int,float,null,undefine,true/false ”数据类型为源语

// 1. 他们都有原型链 \_\_proto\_\_ 目的是方便继承，调用不同的函数或者方法。

// 2. 如创建的对象与蓝图中的原型链:

// a) zhao.\_\_proto\_\_ === Parsen.prototype // true 成立

// 3. 因为有\_\_proto\_\_(原型链)的存在，才可实现不同数据间对函数的灵活调用，且他默认包含官方的默认函数在\_\_proto\_\_中的\_\_proto\_\_可找到，利用谷歌控制器

console.log( zhao.\_\_proto\_\_ === Parsen.prototype ); // true

\*/

## /\*\*\*\*\*\*\*\*\*\* 在chrome控制器调试小技巧-查看原型链对象构成

### // .hasOwnProperty()可判断对象中是否有指定属性

zhao.hasOwnProperty( name );

### // 判断对象蓝图归属

zhao instanceof Parsen // true

// 查看数据的对象构成细节，如数组的对象细节

sss = [1,2];

console.info(sss);

// 如下图:

\*/

## /\*\*\*\*\*\*\*\*\*\* 使用Object.creat()打造继承-模板对象蓝图

// 前面我们演示了，用函数来缔造蓝图，这次用对象来缔造

// 使用Object.creat()，缔造对象模板蓝图

// 缺点: 比起使用函数来缔造对象蓝图，Object.creat其实只搞些普通的属性确实很麻烦

// 优点: 对于函数类型的属性，能够更加容易引用且可读性更高

// 推荐使用方法2

### // 方法1( 其实这个方法比较灵活 ):

//创建蓝图

var personProto = {

cale: function(){

console.log( 2019 - age );

}

}

// 继承蓝图

var zhao = Object.create( personProto );

zhao.name = 'zhao'; // 继承蓝图后在增加属性

zhao.age = '21';

### // 方法2( 这个写法是按照js底层的工作方式来写的value代表属性的内容 ):

var teng = Object.create( personProto, {

// 这里涉及的js的底层工作方式，在书中有提到,本人已解

name: { value: 'teng' },

age: { value: '21' }

} );

\*/

## /\*\*\*\*\*\*\*\*\*\* 关于js的赋值 \*/

### // 0. 普通变量间的赋值

// a) 不同与C，在JS中普通变量间的赋值是真正的复制

// b) 指定的不是内存的同一个位置，但内容相同

var a = 20,b;

b = a;

a = 18;

console.log(a,b); // 18,20

### // 1. 对象之间的赋值

// a) 二者之间并非真正的复制，而是指向同一个内存位置

var test1 = {

a : 'abc',

b : '2000'

}

var test2 = test1;

test1.a = '233';

console.log( test1.a,test2.a ); // 233,233

#### 对象之间的比较

二个对象之间比较大小，其实比较的是对象内存位置是否相同

# 2019.8.5

## /\*\*\*\*\*\*\*\*\*\* 函数，接受函数 - 极为强大方便

// js的函数可以接受函数

### // 原因:

// 0. 函数可以当作变量进行传递

// 如: function a(){}; b = a;

// 那么b()具有a的同样功能

### // 目的:

// 0. 演示接受函数的便捷性

// 1. 实战时减少冗余代码，增加一个骚操作

// 2. js确实灵活

### // 示例

var year = [ 1998,2005,1995 ];

function cale( year,fuc ){ // 接受函数

var result = [];

for( var i = 0; i < year.length; i++ ){

result[i] = fuc( year[i] ); // 注意这里，经过接受的函数处理后在加入数组

}

return result;

}

// 功能1: 计算年龄

function age( year ){

return 2019 - year;

}

// 功能2: 判断是否未成年

function ifOld( year ){

var old = age(year);

return ( old >= 18 ? true : false );

}

console.log( cale(year,age) ); // 输出计算后的年龄

console.log( cale(year,ifOld) ); // 输出是否未成年

\*/

## /\*\*\*\*\*\*\*\*\*\* 函数，返回函数 - 极为强大

### // 目的:

// 可以根据条件灵活的变更函数，以处理不同的数据

### // 返回函数的原理:

// 其实这种返回函数的方法基于“函数表达式”构成的

// var a = function(x);

// 然后可以使用 a(x);

function interView( job ){

if( job == 'running' ){

return function( name ){

console.log( '你的一名运动员?' + name );

}

}

else if( job == 'code' ){

return function( name ){

console.log( name + '哈哈和我一样是玩代码的');

}

}

else{

return function( name ){

console.log( '你到底是做什么工作的?' + name );

}

}

}

#### // 普通调用

var qa = interView( 'game' );

qa('\_\_OO7\_\_');

#### // 便捷调用:

// 原因:

// 0. interView('code')('ZTaer');的调用相当于匿名函数function('ZTaer'){}函数被调用;

// 1. js是从"左到右"来处理代码的

interView( 'code' )( 'ZTaer' );

interView( 'running' )( 'OO7' );

\*/

## /\*\*\*\*\*\*\*\*\*\* IIFF函数表达式，专门创建隐私安全的变量

### // IIFF使用目的:

// 0. 创建极据安全的隐私变量，不影响全局变量，又能在全局中使用，也不需要用函数去命名

// 1. 使用IIFF创建的变量，是一个极具安全的变量

// 2. 可以对一些敏感数据变量进行化函数处理

// 3. 使我们的代码不受其它代码的影响,如变量名称相同时。因为有时一个js项目有多个人合作,可能难免因为变量名称的问题互相产生影响,造成程序混乱.

### // IIFF原理:

// 0. 在函数中外部无法访问函数内的变量

// 1. ()括号内为表达式状态将直接被执行, 其实就是欺骗编译器的一种手段

// 2. 如果无括号()那么只是对函数的声明，并不会被执行

### // IIFF原型:

// ( function(x){} )(x);

( function( money ){

console.log( money + 999 );

})( 1 ); // 直接被执行输出

\*/

## /\*\*\*\*\*\*\*\*\*\* 闭包 - 理解闭包,将能成为一个非常非常厉害的javascript的开发人员,而现在我已经掌握

### // 原因:

// 0. 函数在调用后，正常情况下,局部变量应在内存中清空

// 1. 但是在某些情况，如函数返回函数时，前函数"未闭包"他的局部变量依然在内存中，可以访问.

// 2. 我们利用这一特性，从而创造出更加便利的函数来使用

// 3. 至于"闭包"这不用担心，js将自动闭包

### // 目的:

// 1. 就是在返回函数的过程种调用前函数的变量来借用,以达到类似于“返回函数”便利的效果

// 2. 闭包特性利用的目的在于便捷性

### // 如下案列:

function iv( name ){

return function( job ){ // iv返回函数后，我们依然可以调用局部变量name值，此时就是未闭包

if( job == 'designer' ){

console.log( name + ' explain what UX design is ?' );

}

else if( job == 'teacher' ){

console.log( 'what sbject do you teach ' + name + ' ?' );

}

else{

console.log('hello ' + name + ' what do you do?');

}

}

}

iv('ZTaer')('designer'); // 其实这里利用时，函数返回函数的调用方式( js中函数是可以当一个变量来赋值的 )

iv('ZHAO')('code');

\*/

## /\*\*\*\*\*\*\*\*\* 函数的预设参数实验( 可行 )

function a( name = 'zhao', age = '19' ){

console.log(name,age);

}

a();

a( 'li','21' );

\*/

## /\*\*\*\*\*\*\*\*\*\* .call()方法借用,函数替换

### // 原理:

// 创建一个副本来进行操作

### // 0. 方法引用

// a) a.cale.call(b,x,y...); 模板

// b) call前为借用的方法，call的第一个参数为目标变量，其它为方法需要的参数

### // 1. 函数替换( 感觉并不常用 )

// a) test1.call(test2,x...); 模板

// b) 借用test1函数,相当于test1(x);

### // 0. 方法引用

var john = {

name:'john',

age:12,

cale: function( size ){

console.log( this.name + ' - ' + this.age + ' - ' + size );

}

}

var zhao = {

name:'zhao',

age:23

}

john.cale( 'young' );

john.cale.call(zhao,'old'); // 二对象之间引用方法，此时zhao对象引用john的方法

### // 1. 函数替换

function test1( x ){

console.log( x-100 );

}

function test2( x ){

console.log( x+100 );

}

test1(50);

test2.call( test1,50 ); // 借用test2函数

\*/

## /\*\*\*\*\*\*\*\*\*\*\* .apply()方法借用，功能于call()相同，只是传输参数的方法不同

john.cale.apply( zhao,['old'] ); // 用数组传输参数,其它功能于call一样

\*/

## /\*\*\*\*\*\*\*\*\*\*\* .bind()方法借用，预设参数，返回函数 \*/

### // 注意:

// bind()返回的是一个函数( 复制一个副本 )

### // 功能:

// 借用方法: var test = john.cale.bind( zhao,'big' ); // 返回的为一个函数

// 预设参数: test('code') // 因为有一个变量为预设参数，所以这里传输一个即可

// 返回一个复制的副本函数,以便后续使用

var john = {

name:'john',

age:12,

cale: function( size,job ){

console.log( this.name + ' - ' + this.age + ' - ' + size + ' ' + job );

}

}

var zhao = {

name:'zhao',

age:23

}

var bindTest = john.cale.bind( zhao,'big' ); // bind()具有借用方法功能，预设参数功能，返回的是一个函数

bindTest('code');

#### // 或者

john.cale.bind( zhao,'big' )('code');

## 对2019.8.5面向对象实战总结

1. 老师把所用到的函数都通过prototype的方式加入到了对象中，以方便调用
2. IIFF的目的使，我们的代码不受其它代码的影响,如变量名称相同时。因为有时一个js项目有多个人合作,可能难免因为变量名称的问题互相产生影响,造成程序混乱.

# 2019.8.9

## IIFF+模块化+闭包特性，小项目安全骨架参考：

请参考2019.8.8 – 小应用安全骨架参考 – app.js

// Init()函数，放置初始化函数

// 监听有自己函数

// 抓取前端有自己的函数

// 大模块 + API思维方式

# 2019.8.11

## 预算项目实战笔记

### // 监听全局"按键"并返回对象给event

/\*\*

\* 0. 监听返回一个对象，对象中包含一些参数可以在浏览器控制台中查看

\* 1. 故知Enter键按下返回值为event.keyCode:13;

\* 2. event.which:13;为老旧的浏览器准备，同样存在与返回的对象中

\*/

document.addEventListener('keypress',function(event){

if( event.keyCode == 13 || event.which == 13 ){

control\_add\_items();

}

});

## 抓取表单select参数

<select class="add\_\_type">

<option value="inc" selected>+</option>

<option value="exp">-</option>

</select>

#### Select如下图:



#### 抓取方式:

iType: document.querySelector( .add\_\_type ).value;

iType = inc; // 如果用户选择+则返回’inc’相反则返回’exp’

# 2019.8.12

### Math.round( x )四舍五入

Math.round( 2.5 ) // 为3

# 2019.8.14

## Js变量保存HTML代码

#### // 引用HTML给变量时，不能有空格间隙,并要以字符串的形式引用

html = '<div class="item clearfix" id="income-%id%"></div>';

### 替换字符串内容.replace();

newsHtml = html.replace('%id%',test); // 将html的字符串”%id%”替换为test内容

## 插入HTML代码，.insertAdjacentHTML( position, text );

1. 关于position参数
   1. Beforebegin // 在选中元素之前
   2. Afterbegin // 在选中标签内容中,上部插入
   3. Beforeend // 在选中标签内容中,下部插入
   4. Afterend // 在选中标签内容外部的底部插入内容



1. 实列:

// 插入HTML到前端

newsHtml = ‘<div class=”py-5”></div>’

document.querySelector(choose).insertAdjacentHTML('beforeend',newsHtml);

## // .querySelectorAll( )选中所有元素，返回NodeList对象

/\*\*

### \* 0. 选中指定所有元素，

并返回一个NodeList对象,对象中包括标签中所有被选中的元素( NodeList包含所有的CSS样式 )

### \* 1. 小技巧:

\* a) .querySelectorAll('p').length 可以查看HTML选中元素个数

\* b) 因为返回的是一个对象，对象中包括所有被选中的元素，所以需要迭代出所有元素进行操作

\* c) 但是如果想精确的选中元素进化操作，则需要转换为数组进行操作，因为objct类型数据为无序，数组为有序

### \* 2. 转换数组:

\* a) 借用数组的slice函数: Array.prototype.slice

\* b) 使用call()借用方法: Array.prototype.slice.call(example)即可返回一个数组;

\*/

// 此时为多选

var input = document.querySelectorAll(DOM\_strings.addDescroption + ',' + DOM\_strings.addValue);

input = Array.prototype.slice.call(input);

## // .forEach()方法迭代数组元素，并将元素传递给回调函数。

/\*\*

\* 0. forEach()默认会返回给函数3个元素

\* 1. array.forEach(function(currentValue, index, array), thisValue)

\* a) currentValue: 当前元素

\* b) index: 当前元素索引值

\* c) array: 当前所属数组对象

\* 2. 在函数中操控元素

\*/

input.forEach( function( item,index,array ){

item.value = ""; // 改变每一个迭代出的值

} );

## // .focus()成为焦点元素，表单选中状态

document.getElementById("myAnchor").focus();

# 2019.8.16

## // 以下参数可以在prototype控制台中查看

### // 精确选中获取标签

// .target: 获取当前标签

// .parentNode: 获取父类标签

// .parentElement: 获取父类标签( 与parentNode功能相同 )

// .id 获取标签id

var itemsID = event.target.parentNode.parentNode.parentNode.parentNode.id;

## // .split('x')指定符号,切割字符串并返回数组

splitID = ‘inc-1’;

splitID = itemsID.split('-'); // 返回数组[ ‘inc’,1 ]

itype = splitID[0];

ID = parseInt( splitID[1] );

## //array.map() 遍历数组元素，并进化函数处理，处理完则返回一个新的数组

/\*\*

\* 0.默认会返回3个参数，数组元素为必须，其它2个为可选

\* array.map( function( 数组元素,索引值,当前数组 ){} );

\* 1. 遍历完数组元素后，进行函数处理，处理完则返回一个新的数组，

\* 2. 排序保持不变,且不会改变原数组

\* \*/

// data.allItems[ itype ] 为一个数组

ids = data.allItems[ itype ].map( function( current ){ // 通过map函数获取数组的id值

return current.id;

} ); // ids = [1,2,3]

indexID = ids.indexOf(ID) // 利用map返回数组排序不变，配合indexOf判断目标索引值

## //利用splice()修改数组( 注意split()与splice()区别 )

// array.splice()添加/删除数组( 注意，会直接影响原数组 )

/\*\*\*

\* 0. array.splice( 目标索引值，删除数量，添加的元素0,添加的元素1.... );

\* a) 注意splice将直接改变原数组内容，无需赋值操作

\* b) 将返回删除的数组

\* 1. 删除:

\* 返回删除的数组，且原数组也将改变

\* var fruits = ["Banana", "Orange", "Apple", "Mango"];

\* fruits.splice(2,2); // Banana,Orange

\* 2. 添加( 指定插入数组内容 ):

\* var fruits = ["Banana", "Orange", "Apple", "Mango"];

\* fruits.splice(2,0,"Lemon","Kiwi"); // Banana,Orange,Lemon,Kiwi,Apple,Mango

\* 3. 替换添加:

\* fruits.splice(2,1,"Lemon","Kiwi"); // Banana,Orange,Lemon,Kiwi,Mango

\*

\*/

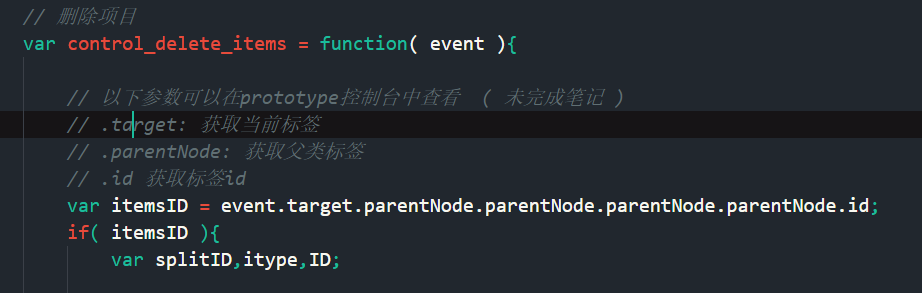
## // 监听标签内DIV内的所有活动，并针对某活动进行函数处理

<div class=”container” ></div>

// 监听删除按钮

document.querySelector(getDOM.container).addEventListener( 'click', control\_delete\_items );

// 在配合其它函数获取特定的动态，达到指定特征监听活动效果



## //.removeChild( ) 删除父类中所有子类HTML标签内容

// 目标父类.REMOVECHILD( 目标子类 ) 删除子标签

// a) 因为js，.removechild()只能通过父类来删除子类，进行删除目标( 目前没用方法直接删除目标标签 )

target = document.getElementById( targetID ); // 删除目标

target.parentNode.removeChild( target ); // 目标的父类.removeChild( 目标 )

# 2019.8.21

## 注意: split(),splice(),slice()不要搞混喽

1. Splict()用于切割字符串并返回一个数组
   1. 如: ‘abc’.Split(‘b’); // [‘a’,’c’]
2. Splice()用于，添加/删除数组元素
   1. 请看上方详细解析
3. Slice()用于, 数组切片功能
   1. B = [1,2,3,4];
   2. B.slice( 1,3 ); // [ 2,3 ]

## 回调函数callback()

### // 回调函数实列 - 创建nodelist的forEach()函数（ 迭代对象进化函数处理 ）

// 回调函数: 返回参数给其它调用的父类函数

// callback() 回调是一个函数，它作为参数传递给另一个函数，并在其父函数完成后执行。( 未完成笔记 )

var nodeListForEach = function( list, callback ){

for( var i=0; i<list.length; i++ ){

callback( list[i],i );

}

}

#### nodeListForEach()使用方法:

nodeListForEach( nodeList, function( cur,index ){

cur // 迭代出的对象元素

index // 元素在nodeList中的索引值

} )；

## Math.abs()绝对值

// 绝对值,保证数字为正数

resultNum = Math.abs(num);

## substr()自定义字符串切片

// substr()自定义字符串切片,不影响原字符串

// substr( 开始位置,结束位置 );

// 实际切去位置，与python的计数方式一样( 开始位置 ~ 结束位置-1 )

如: a = '4000', a.substr( 0,1 ) == '4';

a.substr( 1,a.length ); == ‘0000’

## // Date()获取当前时间

// a) getFullYear(): 获取当前年

// b) getMonth(): 获取当前月

// c) 更多参考: <https://www.runoob.com/jsref/jsref-obj-date.html>

#### 获取时间实列:

time = new Date(); // 获取时间

tYear = time.getFullYear();

tMouth = time.getMonth();

## 监听变化事件,”change”

### 监听变化事件实列( 预算应用项目，改变+/-时在改变表单颜色 ):

### // 1. 监听+/-切换选项，得以操控样式

// 监听事件 - 监听改变状态，一旦标签发生改变则执行此函数

document.querySelector(getDOM.addType).addEventListener('change', ui.changeFocus );

### // 2. 切换表单焦点样式

changeFocus: function(){

// 创建一个对象选中多个元素（ 类名 ）

var tar = document.querySelectorAll(

DOM\_strings.addType + ',' +

DOM\_strings.addDescroption + ',' +

DOM\_strings.addValue

);

// 配合nodelistForEach(可以迭代对象的函数)进行多个toggle的方法进行元素操控( 补充: nodeListForEach()本质，以及要入'黄金模块' )

nodeListForEach( tar, function( cur ){

cur.classList.toggle('red-focus');

} );

// 给按钮切换焦点样式

document.querySelector(DOM\_strings.addBtn).classList.toggle('red');

}

# 2019.8.26( ES6 )

## 开始学习ES6

/\*\*

## // 0. ES6-创建变量的二种方式

// a) let: 创建可改变变量

// b) const: 创建不可改变变量

let actv = 123; // 可改变

const noactv = 456; // 不可改变

// noactv = 123; // 会报错

## // 1. ES6-变量的作用范围

// a) var 工作范围为一个作用域

// b) let / const 工作范围为一个“块范围”，出了大括号即可被内存释放

// 0. 关于变量的使用范围，

// 1. 老版本能使用在相应的作用域，而现版本则为"块"的概念，工作方式不同，

// 2. 新版本的变量只能在" 大括号内，使用，包括IF/FOR等

// 3, ( 有点像function的局部变量，但是要比function要求更加严格,因为在if/for中出了大括号变量一样无法使用 )

let a = 1;

if( a > 1 ){

console.log(s); // 如果为ES5的var，此时会返回undefine,在ES6中的let/const则会报错

let s = 'ok';

}

console.log(s); // 会报错，因为变量不存在

## // 2. ES6-新版IIFF

// a) 创建新版IIFF只需"{}"一对大括号,在配合let/const

// b) 切忌不要使用var创建变量,因为ES6变量为块范围，var为作用域范围

{

let abc = 1;

}

\*\*/

/\*\*

## // 3. ES6-新版输出字符串方式( 便捷式输出字符串 )

// a) 直接将字符串``写在此符号中( 数字1键左侧 )

// b) 将变量/函数写在此括号中 ${ \* } 可以正常的调用函数/变量

// c) 模板: ` ${ } `

// 演示:

let first = 'zhao';

let last = 'teng';

let calc = function( num ){

return num - 1998;

}

console.log( `name: ${first} ${last}\nyear: ${ calc(2019) }` ); // 可直接调用变量，和函数

## // 4. 验证字符串是否包含指定字符串

// a) .startsWith(\*): 验证字符串开头

// b) .endsWith(\*): 验证字符串结尾

// c) .includes(\*): 验证是否包含字符串

// d) 是: 返回true -- 否: 返回false

// 演示:

let name = 'ztaer';

console.log( name.startsWith('zt') ); // true - 验证开头是否为指定字符串

console.log( name.endsWith('aer') ); // true - 验证是否结尾为指定字符串

console.log( name.includes('taer') ); // true - 验证是否包含字符串

### Array.includes( 目标值, 位置 )也可以用于数组验证

Const test = [ 1,2,3,4 ];

正常验证: test.includes( 2 ); // true

指定位置验证: test.includes( 2,3 ) // false

## // 5. 循环叠加字符串

// a) .repeat( num ): 重复叠加字符串

let ho = 'ho';

ho = ho.repeat(6); // 重复叠加6次字符串

console.log( ho );

\*\*/

/\*\*

## // 6. 3种箭头函数使用方法(=>),用来代替function函数

// a) 第一种,单参数单语句写法: 参数 => 表达式；

// b) 第二种,多个参数当语句写法: ( 参数，参数 ) => 表达式;

// c) 第三种,完整的=>函数写法: ( 参数,参数 ) => { 表达式; 表达式; return 表达式 };

// 0. 有大括号需要return来返回数据, 无大括号时可直接返回数据

### // 第一种

let a = [ 1,2,3,4 ];

let b = a.map( cur => cur+1 );

console.log(b);

### // 第二种

let c = a.map( ( cur,index ) => cur - index );

console.log(c);

### // 第三种法

let sss = ( num ) => {

return num + 1;

};

console.log( sss(5) );

\*\*/

/\*

## // 7. 对象的this变量多级处理

### // 0. ES5,this变量多级处理方式

var test = {

name:'zhao',

age:18,

sex:'man',

calc: function(){

var self = this; // 因为this只能进行一级调用,有它 可以进行多级的数据处理

document.querySelector('.b-g').addEventListener('click',function(){

self.bro = 2019 - self.age;

console.log(test);

});

}

}

test.calc();

\*/

/\*\*

### // 1. ES6,this变量多级处理方式( 说白了,function嵌套=>;才能使=>直接读取this变量值 )

// a) function(){ ()=>{} } : 在对象中,function中嵌套=> 则=>可以直接读取function的变量,包括this变量

// b) 因为=> 会获取父类环境的所有变量

// c) 注意: () => { ()=>{} }: 如果=>嵌套=>与function嵌套function结果一样

var test = {

name:'zhao',

age:18,

sex:'man',

calc: function(){

document.querySelector('.b-g').addEventListener('click',() => {

this.bro = 2019 - this.age;

console.log(test);

});

}

}

test.calc();

## // 8. 对象模板中处理多级变量

### // ES5 - 对象模板处理多级this - 使用bind()处理多级this变量的骚操作

var Model = function( name ){

this.name = name;

this.age = '21';

}

Model.prototype.calc = function( friends ){

var newArr = friends.map( function( cur ){

return this.name + '是' + cur + '的朋友';

}.bind(this) ); // 函数.bind(this) 借用函数处理this变量

console.log(newArr);

}

var friends = [ 'abc','cad','ccc' ];

new Model( 'ztaer' ).calc( friends );

\*\*/

### // ES6 - 对象模板处理多级this,原理就是function 嵌套 => ;使 => 可以读取使用所有的父类function变量,包括this

let Model = function( name ){

this.name = name;

this.age = '21';

}

// 原理就是function 嵌套 => ;使 => 可以读取使用所有的父类function变量,包括this

Model.prototype.calc = function( friends ){

let newArr = friends.map( cur => {

return this.name + '是' + cur + '的朋友';

});

console.log(newArr);

}

let friends = [ 'abc','cad','ccc' ];

new Model( 'ztaer' ).calc( friends );

## // ES6-关于解构,目的是方便操控数据

/\*

### // 1. 数组解构

// 便捷式给数组元素命名多个变量，方便索引调用

const masg = ['zhaoteng','21','1998'];

const [ name,age,bro ] = masg; // 数组解构

console.log(name,age,bro);

\*/

/\*

### // 2. 对象解构

// 0. 解构的目的是，可以直接索引到对象数据

// 1. 大括号内的变量名称要与对象中变量名称相同对应

// 2. 当然也可以重命名对象索引的变量名称,如下演示

const masg2 = {

name : 'ztaer',

age : '18',

bro : '2001'

};

#### // 对象解构

const { name,age,bro } = masg2;

console.log(name,age,bro);

#### // 对象解构重命名

const { name: mz, age: nl,bro: cs } = masg2;

console.log( mz,nl,cs );

### // 3. 函数返回值解构( 原理为数组结构 )

function abs(num){

let inc = num + 100;

let exp = num - 100;

return [ inc,exp ]; // 函数返回数组

}

const [ numInc,numExp ] = abs(99); // 其实就相当于数组解构

console.log( numInc,numExp );

\*/

## // 0. ES6-Array.form()快速将nodeList对象转换数组

list = document.querySelectorAll('.box');

arr = Array.from(list); // nodeList对象转为数组( 核心 )

console.log(arr);

## // 1. ES6-新的for迭代方式 - className+includes可以精确选中标签

// a) for( cur of array ){ cur为迭代出的元素 }:

// b) 关于.className: 获取标签所有类名

// 0. 返回标签所有的class类( 字符串类型 )，不同类名用空格隔开

// c) 关于includes( str ): 验证字符串中是否包含指定字符串

for( cur of arr ){ // arr为一个数组

if( cur.className.includes('blue') ){ //通过配合精确选中指定标签

continue;

}

else{

cur.textContent = '2333!!!!';

cur.style.backgroundColor = 'pink';

}

}

## // 2. ES6-array.find( => )快速查找数组中符合条件的元素,返回元素值

// a) 只能查找一个符合条件的元素

// b) 找到: 返回元素值

// c) 没找到: 返回-1

// d) array.find( (cur,index,thisArray) => ... ); 一般只需要cur足矣

const test = [12,22,21,2,4];

let b = test.find( cur => cur >= 18 ); // ( 核心 )

console.log(b); // 22: 返回第一个符合条件的元素

## // 3. ES6-array.findIndex( => )快速查找数组中符合条件的元素,返回索引值

// a) 只能查找一个符合条件的元素

// b) 找到: 返回元素索引值

// c) 没找到: 返回-1

// d) array.findIndex( (cur,index,thisArray) => ... ); 一般只需要cur足矣

let c = test.findIndex( cur => cur >= 18 );// ( 核心 )

console.log(c);

## // 4. 函数直接使用数组元素作为参数

//0. ES5-使用.apply()

// a) function.apply(null,Array)舍弃替换目标,直接传输参数到借用的函数中

//1. ES6-“...”扩展符

// b) 扩展符可以使用在“数组”，"nodeList"中

### //ES5

function add( a, b, c, d ){

return a+b+c+d;

}

var ages = [ 2,32,21,23 ];

var result = add.apply(null,ages); //( 核心 - function.apply(null,Array)舍弃替换目标,直接传输参数到借用的函数中 )

console.log(result);

### //ES6

result = add( ...ages ); // 核心

console.log(result);

## // 5. 关于"..."(扩展符)的使用

// 0. 扩展符可以使用在“数组”，"nodeList"中

// 1. 扩展符释放所有 数组/nodeList 元素

// a) [...nodeList,...array,...other] 这种扩展符使用小技巧非常重要

### // 0. 数组使用扩展符

// a) 可以直接使用在函数的参数中

// b) 利用扩展符数组组合 [...array,...array,other]

aTest = ['zhao','teng'];

bTest = ['ztaer','killer'];

cResult = [ ...aTest,'OO7',...bTest ]; // 利用扩展符数组组合

console.log(cResult); //  ["zhao", "teng", "OO7", "ztaer", "killer"]

### // 1. nodeList 使用扩展符

// a) 利用扩展符数组组合 [...nodeList,...nodeList,other]

// b) [...nodeList]: 此方法nodeList转数组的功能

const h = document.querySelector('h1');

const box = document.querySelectorAll('.box'); // nodeList

const all = [h,...box]; // 核心: 扩展符组合，其实这里已经将nodeList转换为数组啦

all.forEach( cur => cur.style.color = 'red' );

# 2019.9.2

## // 0. ES6-函数处理多个参数方法

#### //0. ES5-在函数中使用变量: arguments可以获取函数使有传输的参数

// a) 注意: arguments并非数组,迭代使用时需转换

// b) Array.prototype.slice.call(arguments,1) // 可切出所需参数

#### //1. ES6-方便快捷,使用扩展符转换传输的函数为数组

// a) 传输多个参数和单个参数,的同时，用逗号隔开也极为方便分辨传输单个参数和多个参数

#### //2. Map()方法汇总

// 创建Map()数据类型

// let test = new Map();

// 获取Map存储值

// Map.get(key);

// 增删查改:

// 键值已经存在,则set()为修改

// Map.set( key, value); Map.delete( key ); Map.has( key ); Map.set( key, value );

// 获取map所有元素( 常与解构配合进行迭代 )

// Map.entries();

// 清空map所有元素

// Map.clear();

### //ES5

function add5(){

// arguments变量包含传输的所有参数

//a) arguments并非数组,只是张得像,迭代时需转换( 核心 )

console.log(arguments);

var argu = Array.prototype.slice.call(arguments);

argu.forEach( function( cur ){

console.log(cur);

} );

}

add5(1,2,3,4,45,2);

### // ES6

// ...year使用扩展符将传输的参数转换为数组( 核心 )

function add6(limit,...year){

for( cur of year ){

if(cur >= limit){

console.log(true);

}

else{

console.log(false);

}

}

}

add6(18, 12,23,22,11,23,44,1,3,7,22);

## // 1. ES6-函数设置默认参数

// 0. ES5-设置默认参数需3元表达式配合

// a) 过程极为繁杂

// 1. ES6-设置默认参数直接使用=号即可，与python相似

### //ES5

function Msg( name, age, country ){

// 设定函数默认值

country == undefined ? country = 'cn' : country = country;

this.name = name;

this.age = age;

this.country = country;

}

var oo7 = new Msg('zhao','18','usa');

var ztaer = new Msg('killer','21'); // country没有赋值则按默认值设定

console.log(oo7,ztaer);

### //ES6

// 直接使用等号设定默认值

function Msg6( name, age, country = 'cn', lang ='zh'){

this.name = name;

this.age = age;

this.country = country;

this.lang = lang;

}

let zhao = new Msg6( 'teng',22 );

## // 2. ES6-Map()数据类型

#### // 0. 使用Map()数据类型的理由:

// a) 可迭代，有序存储

// b) 方便'增删查改'

// c) 可以存储任何数据类型

// d) 可以通过Map.size()函数获取准确大小

#### // 1. Map()数据结构:

// { key => value, key => value ... }

// 左边为键值,右边为存储值

### // 0. Map()数据,增删添改,使用实列

// new Map(): 创建map

const question = new Map();

// Map.set( key ): map增加map元素,可以保存任何数据类型

question.set('Q','你喜欢那个版本的JavaScript?');

question.set(1,'ES5');

question.set(2,'ES6');

question.set(3,'ES7');

question.set('A',2);

question.set(true,'YES!!!');

question.set(false,'NO!!!');

// Map.has( key ): map验证指定键值是否存在map中

if( question.has(3) ){

// Map.delete( key ): map删除指定元素

question.delete(3);

}

console.log(question);

// Map.get( key ): map获取指定元素

question.get('A');

// Map.set(key): map若键值已存在,则直接修改内容

question.set(2,'ES2015');

console.log(question);

// Map.size: map判断map大小

console.log( question.size );

// Map.clear(): 清空map元素

// console.log( question.clear() );

### // 1. map数据迭代

// a) forEach()方法

// b) for方法

#### // 0. forEach()方法迭代map

// a）注意,参数值为反方向

question.forEach( ( value,key ) => {

console.log(` ${key} --- ${value} `);

} );

#### // 1. for()方法迭代map

// a) Map.entries() 获取map全部值

// b) []通过解构的方法获取迭代出的对应值

for( [ key,value ] of question.entries() ){

console.log(` ${key} --- ${value} `);

}

### // 2. map数据结构选择题实例

const ans = parseInt( prompt(question.get('Q')) );

console.log(

question.get( ans == question.get('A') ) // 中心返回一个布尔值( 在通过布尔值返回对应数据 )

);

# 2019.9.5

/\*

## // ES6-使用class来写蓝图的方式

// 0. 起本质哈是prototype继承

// 1. class不能直接添加属性，但因为继承属性的原因，最终也是属性的聚焦

// 2. 通过函数方法的形式足以创建所需属性，所以class也无需需要直接添加属性的功能，起函数方法足矣

class Person6{

msg( name, bro, job ){

this.name = name;

this.bro = bro;

this.job = job;

}

calcAge(){

let age = new Date().getFullYear() - this.bro;

console.log(age);

}

static print(){

console.log( '定义静态函数，可直接通过蓝图调用' );

}

}

### // ES6的蓝图的使用方式相同( 特别注意: 使用class一定要先NEW才能正常使用 )

let zhao = new Person6();

zhao.msg( 'teng', '1998', 'hacker' );

zhao.calcAge();

### // 调用静态函数 -可直接通过蓝图名称来调用相应函数( 并不常用 )

Person6.print();

\*/

## // ES6-蓝图合并的方法

// 0. 其本质工作方式在ES5更能体现出( 记得有时间整理理解下 )

// 1. class默认自带constructor()函数

// 2. 在extends扩展蓝图后

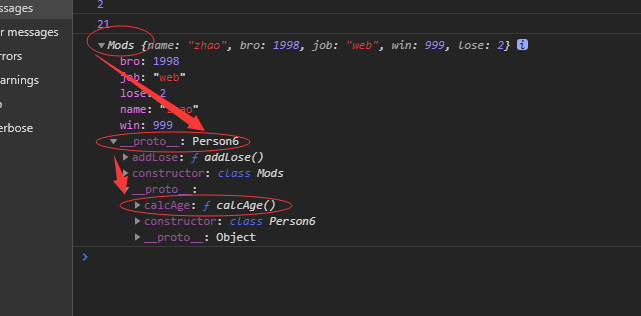
// a) super()函数只能用于constructor()中引用蓝图属性变量

// 3. 蓝图( 参数 ): 默认直接参数传输给constructor()函数

// a) 传入形式: let test = new Person6(0,'test');

// 4. 融合蓝图后, class 新类 extends 原类 { }； 相当于‘原类’成为‘新类’的子类，如图：

原因： 所以‘新类’可以调用‘原类’函数



### // 使用 extends使二者蓝图融合扩展

class Person6{

// class默认就有constructor()函数

constructor( name, bro, job ){

this.name = name;

this.bro = bro;

this.job = job;

}

calcAge(){

let age = new Date().getFullYear() - this.bro;

console.log(age);

}

}

class Mods extends Person6{

constructor( name, bro, job, win, lose ){ // 注意变量名称要对应

// 在constructor中才能使用super函数引用属性

super( name, bro, job ); // 注意变量名称要对应

this.win = win;

this.lose = lose;

}

addLose(){

this.lose++;

console.log(this.lose)

}

}

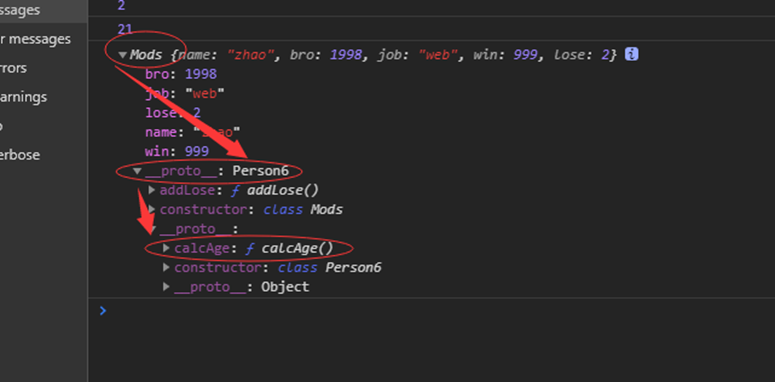
#### // 蓝图( 参数 ): 默认直接参数传输给constructor()函数

const oo7 = new Mods('zhao', 1998, 'web', 999, 1);

#### // 融合蓝图后,二者蓝图的函数都可调用

oo7.addLose();

oo7.calcAge();



### 关于super()继承蓝图属性的正常用法

// super()继承蓝图属性

class Person6{

constructor(){

this.name = 'name';

this.bro = 'bro';

}

}

class Mods extends Person6{

constructor(){

super(); // super()默认继承蓝图全部属性( 蓝图继承必写 )

this.win = 2;

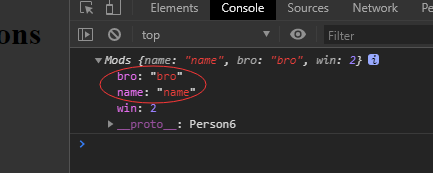
}

}

const oo7 = new Mods();

console.log( oo7 );

// 如图：蓝图的属性，全部被继承过来



## // class内部函数调用

// 0. 调用方式: this.test();即可调用内部函数

// 1. 调用函数不分上下, 如下演示, 上方函数可以直接调用下方函数

class Ptest{

a = () => this.b(); // 可正常调用

b = () => console.log( 'bbb' );

}

const test = new Ptest();

test.b();

## // .reduce()数组数字计算函数

// array.reduce( function(total, currentValue, currentIndex, arr) , initialValue)

// total: 总和

// currentValue: 当前数组元素值

// arr: 数组对像( 不常用 )

// initialValue: 传入的默认参数( 从左边到右进化函数参数的默认值设定，所以此时设定的默认值为total )

### // 实列:

let a = [1,2,3,4];

#### // reduce()使用的函数分析:

// total + cur : 相当于 return total + cur; 以递归的方式计算总和

// reduce( f( total,cur,index ),0 ): 至于0为传入函数的默认参数

let result = a.reduce( ( total, cur, index ) => total + cur, 0 );

console.log( result );

# 2019.9.10

## ES6-异步实列

### // setTimeout()异步函数之一,设定指定时间异步执行此事件

let test1 = () => {

console.log('hello');

}

let test2 = () => {

console.log('world');

#### // setTimeout()异步函数之一,设定指定时间异步执行此事件

// 原模型: setTimeout( ()=>{},times );

// 使用此函数后，不会影响后续代码的执行，这就是异步的好处

setTimeout( () => {

test1();

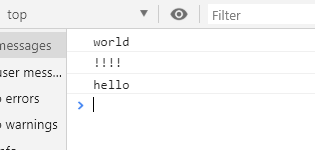
},3000 ); // 3秒后执行此事件

console.log('!!!!');

}

test2();

执行结果:



## // ES6-js异步底层工作方式

### 先看以下Js工作组成

执法堆栈 – 1. WEB APIS – 2. 消息队列 – 3. 事件监听循环

1. 如下图:



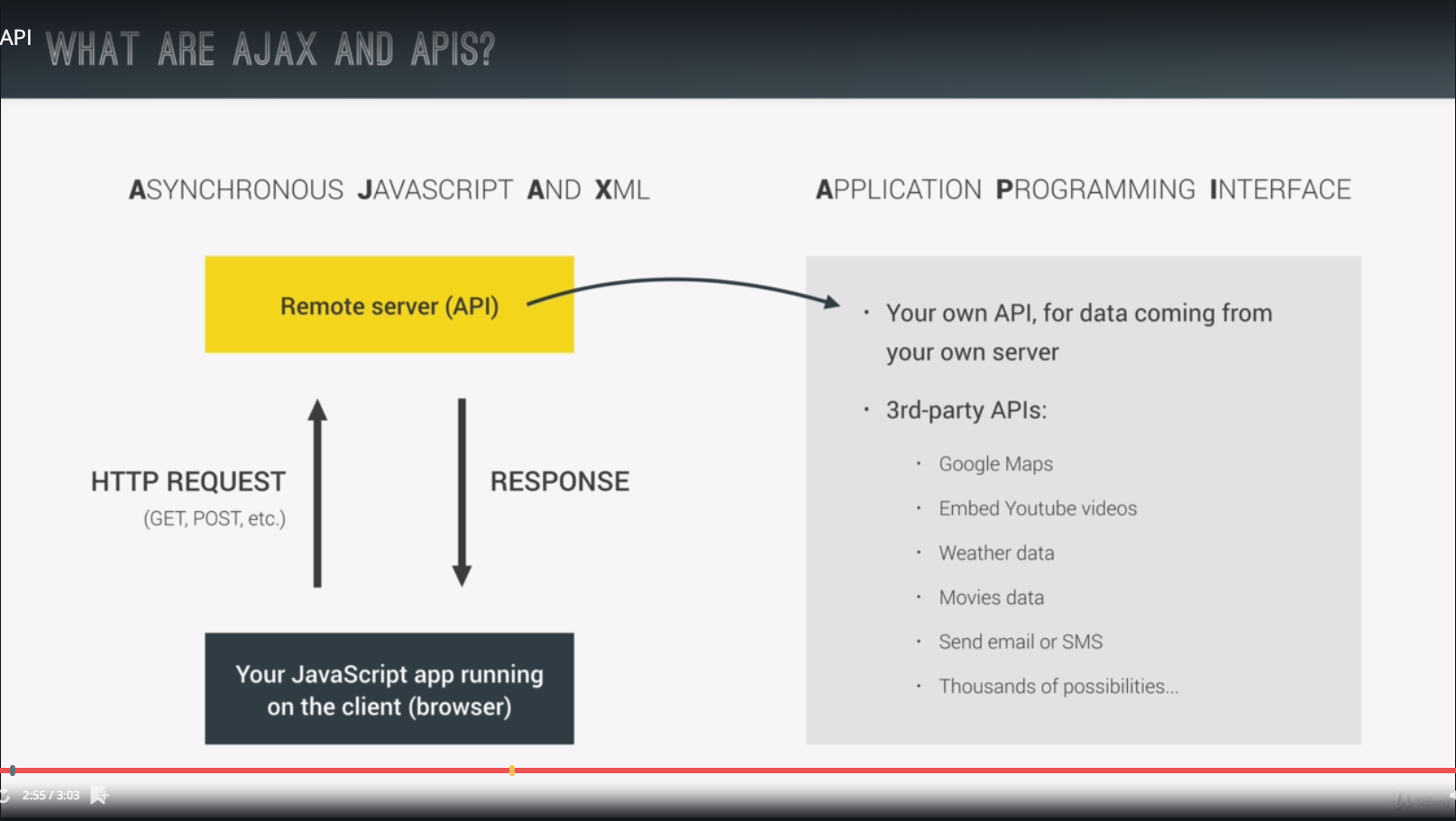
### 异步工作步骤方式:

1. 执行到异步函数，将事件转移到web apis ( web apis 属于浏览器 )，其它代码正常执行
2. 等待到执行时间后，加入到”消息队列”
3. 由”event loog”事件监听循环，监听到执行堆栈为空时
4. 堆栈为空时将”消息队列”中的事件加入到执行堆栈中
5. 完成异步

# 2019.9.14

## AJAX工作方式

// AJAX解析 - Asynchronous Javascript And Xml - 异步js和xml



/\*

## // ES6-异步

### // setTimeout()异步函数之一,设定指定时间异步执行此事件

// 原模型: setTimeout( ()=>{},times );

// 使用此函数后，不会影响后续代码的执行，这就是异步的好处

let test1 = () => {

console.log('hello');

}

let test2 = () => {

console.log('world');

setTimeout( () => {

test1();

},3000 );

console.log('!!!!');

}

test2();

\*/

## // 多种版本模拟服务器获取信息异步方法

### // 0. ES5 - callback hell 回调地狱三角异步方法( 难维护,不方便阅读 )

// 0. 模型: setTimeout( ()=>{}, times, 默认参数 )

// 1. setTimeout()可以互相嵌套

function getRecipe5(params) {

setTimeout( () => {

const ids = [01,02,03,04,05];

console.log( ids );

setTimeout( id => {

const recipe = {

id:02,

title:'美味鲜美的番茄披萨',

description:'12英寸美味披萨，精烹3小时,来自亚洲张家界优质番茄,最美味的味道最匹配最好的你!',

publisher:'ZTaer',

}

console.log(`ID: ${id} - ${recipe.title}`);

setTimeout( publisher => {

console.log( `由${publisher}大厨烹饪!` );

}, 1500, recipe.publisher );

},2000,ids[1] );

}, 2000 );

}

console.log('BGN!!!');

// getRecipe5();

### // 1. ES6 - Promises对象异步方法( 较易维护,方便可读,但是依然较为麻烦 )

#### // 0. 创建Promises函数

// a) 模型new Promise( (resolve, reject) => {} );

// b) 不过new Promise()通常情况下与setTimeout()配合

// 0. new Promises( ( resolve, reject ) => {

// setTimeout( ()=>{ resolve('返回内容') },1500 );

// } );

// c) 在Promises中:

// 0. resolve()获取数据"成功"时返回的数据

// 1. reject()获取数据"失败"时返回的数据

// d) 注意: new Promise( (resolve, reject)=>{} );其中resolve,reject为必备参数

#### // 1. 处理异步数据方法.then( cur=>{} ).catch( error=>{} );

// a) 具体使用方式请看下方实列演示

// b) Promise.then( ()=>{} ).then( ()=>{} ).catch( ()=>{} );

// 0. then()可进行多级嵌套，并持续处理return值

// a) 如: then( ()=>{return a} ).then( a=>{ console.log(a) } )

// b) a为同一值,因为then()的return是传递给下一个then的表现

// c) then()用于专门等待Promise异步数据输出来处理

// 1. catch( error=>{} )当数据发生错误执行此函数中的语句

### // a) 模拟创建异步获取数据

const getIDs = new Promise( ( resolve, reject ) => {

setTimeout( ()=>{

const ids = [10,20,30,40,50];

resolve(ids); // 成功获取数据后 - 返回获取到的值

reject('error'); // 获取数据失败后 - 返回的值

},1500 );

} );

const getRecipe = id => {

return new Promise( ( resolve, reject ) => {

setTimeout( id => {

const recipe = {

id:02,

title:'美味鲜美的番茄披萨',

description:'12英寸美味披萨，精烹3小时,来自亚洲张家界优质番茄,最美味的味道最匹配最好的你!',

publisher:'ZTaer',

}

resolve(`ID: ${id} - ${recipe.title}`);

}, 1500, id );

} )

}

const getRelated = publisher => {

return new Promise( ( resolve, reject ) => {

setTimeout( publisher => {

resolve( `由${publisher}大厨烹饪!` );

},1500, publisher );

} );

}

/\*

#### // b) 模拟使用异步函数获取服务器信息

getIDs.then( IDs => {

console.log(IDs);

return getRecipe( IDs[1] ); // .then()具有可传递性数据处理

} )

.then( recipe => {

console.log( recipe );

return getRelated( 'ZTaer' );

} )

.then( publisher => {

console.log( publisher );

} )

.catch( error => {

console.log( error );

} );

\*/

## // 2. ES8 - async/await处理异步数据( 更科学,更易维护 )

// 0. 创建异步的方法依然使用 new Promise()

// 1. async只是处理异步数据的方法,代替了then()/catch()

// 2. await解析:

// a) await只能使用在async函数中

// b) await 为等待异步数据输出,在接受处理结果

// 3. 异步数据处理步骤:

// a) 0. new Promise创建异步函数 -> 1. async处理Promise输出的异步数据 -> 2. then处理async的异步数据

async function getRecipe8(){

const IDs = await getIDs; // await处理Promise写法

console.log(IDs);

const recipe = await IDs[1];

console.log(recipe);

const publisher = await getRelated('ZTaer');

console.log(publisher);

return recipe;

}

// 注意: 这样是无法处理async函数的输出值的,因为async为异步状态

// const get8 = getRecipe8();

// console.log(get8);

// 处理async异步数据方法需要于then配合

getRecipe8().then( cur => {

console.log(` async配合then处理异步数据 `);

} );

## // 关于fetch()函数获取API数据

// 0. 用于获取API的数据

// 1. 用法: fetch(`API链接`);

/\*

## // 0. then实战请求天气数据 - Fetch/Promises/then方法( 不推荐使用then,因为维护性不强，推荐使用async与then配合使用 )

let cityWather = fetch('https://cors-anywhere.herokuapp.com/https://www.metaweather.com/api/location/2151330/') // 请求API数据 - 在没有域名的情况下本地需要cros代理才能正常的获取API数据

cityWather.then( cur => {

return cur.json(); // 将json数据转换为对象类型

} )

.then( data =>{

console.log(data);

console.log('------');

const today = data.consolidated\_weather[0];

console.log(`${data.title} - ${today.applicable\_date} - 气温${today.the\_temp} - 天气${today.weather\_state\_name}`);

} )

.catch( error => {

console.log(error);

} );

\*/

## // 1. async实战请求天气数据 - Fetch/Promises/async( 推荐使用async与then配合 )

// 0. async检错机制

// try{ ...代码内容 }catch(error){ ...代码内容 }

async function getWether( cityNum ){

try{

let data = await fetch(`https://cors-anywhere.herokuapp.com/https://www.metaweather.com/api/location/${cityNum}/`);

data = await data.json();

const today = data.consolidated\_weather[0];

console.log(`${data.title} - ${today.applicable\_date} - 气温${today.the\_temp} - 天气${today.weather\_state\_name}`);

return data;

}

catch( error ){

console.log(error);

}

}

getWether(2151330); // async函数调用

#### // async与then配合 - 处理async函数的异步数据

getWether(44418).then( data => {

console.log(data);

} )

.catch( error => console.log(error) );

# 2019.9.16

## 大型项目准备

### 必备3工具

NPM： 方便管理第三方库和Node.js部署，方便下载使用第三方库，也方便共享自己的库

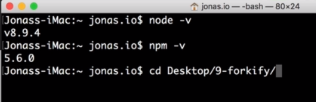
Babel: 将新版本的代码( 如: ES6/7/8 )转换为ES5代码以提高代码的兼容性

Webpack: 打包js/css/scss等,将多个库融合在一起,不过json老师推荐融合js足矣

## 此行的目的是为了让NPM – Babel – Webpack 配合起来( 实战例子为: 2019.9.16 )

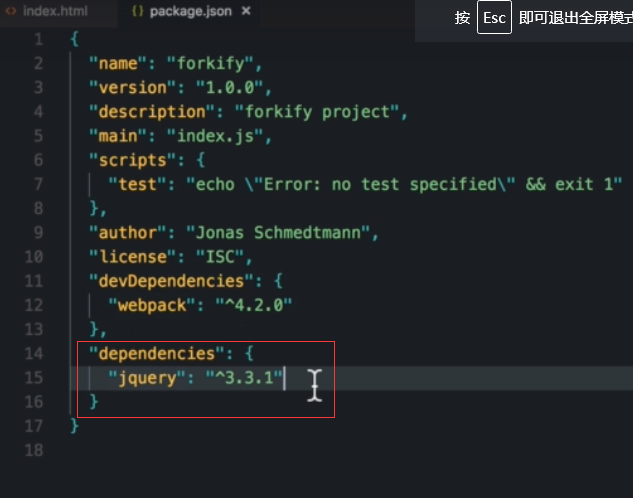
### NPM的安装

1. 安装NPM – 因为Node.js与Npm是绑定的所以直接去官网安装Node.js即可
   1. 官网: <https://nodejs.org/en/>
   2. 直接下载对应版本安装即可
   3. 在CMD命令中 npm –v node –v 检测是否安装正常( 通过查询版本号确定是否安装 )



1. Npm init创建npm管理目录( 找到目标文件夹 )
   1. 根据提示生成package.json( 创建完毕也可编辑 )

#### Npm安装与卸载第三方库方法

1. 如: npm安装jquery
   1. npm安装jquery命令Npm install jquery - -save
   2. 
   3. 卸载jquery方法 npm uninstall jquery - -save
   4. 注意安装的分支: 如果安装时在- -save –dev 那么卸载时也要指定

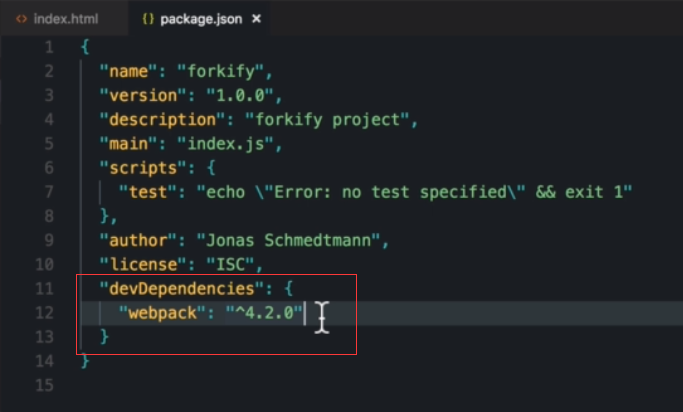
#### Npm中生成的package.json重要性

1. Npm在安装第三方插件后也会生成“node\_modules”文件夹
   1. 里面包含所有的环境依赖文件
2. 但是即使删除node\_modules文件夹也是可以的，只要保留package.json文件
   1. 通过命令 npm install 他将会根据package.json的配置自动生成所需配置文件
3. Package.json也为Npm核心配置文件

#### NPM全局安装 live-server短暂的成为”实时服务器”提供js等

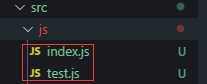
1. Live-server是能使一个文件夹短暂的成为一个”实时服务器”
   1. 其目的是: 实时查看开发的网页或项目效果。
2. 安装命令 npm install live-server - -global （- -global可以使用-g代替）
3. 直接到指定文件夹如js/css存放位置: 输入命令 live-server即可开启实时共享
   1. 他能实时监听文件是否有所改变
4. 在命令行中Ctrol + C即可终止共享

### WEBPACK安装

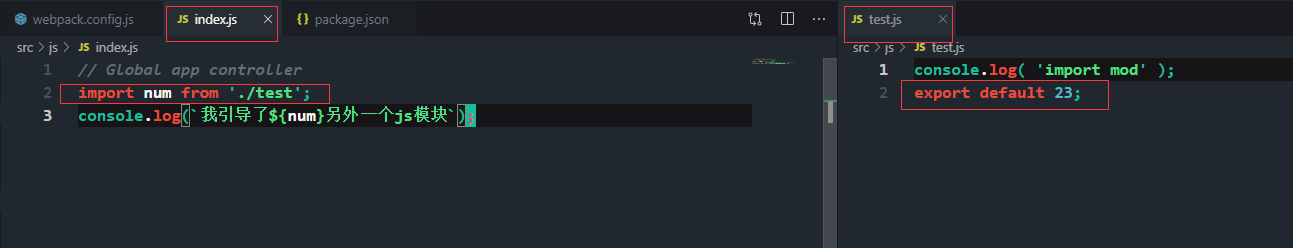
1. 在cmd中输入： npm install webpack - -save–dev // 其- - save –dev 的意思为保存的 dev环境分支中
   1. 
2. 创建”webpack.config.js”文件为webpack配置文件
   1. Webpack有四大核心: 入口点, 输出点, 加载器, 插件
   2. webpack.config.js总体配置:
3. ***// 设定当前文件所在位置为根目录***
4. **const path = require('path');**
5. **module.exports = {**
6. ***// 切入点***
7. ***// a) 设定监控文件 ( "./"为当前目录的意思 )***
8. ***// b) 注意: 多js文件绑定时需 import/export配合***
9. **entry: './src/js/index.js',**
11. ***// 输出点***
12. ***// a) path: 为设定生成路径***
13. ***// a) \_\_dirname：为生成文件的设定的权限***
14. ***// b) ‘dist/js’: 为生成的js文件路径***
15. ***// b) filename: 为设定生成文件名称***
16. **output:{**
17. **path: path.resolve( \_\_dirname, 'dist/js' ),**
18. **filename: 'bundle.js'**
19. **},**
20. **};**

#### webpack多js文件打包配置

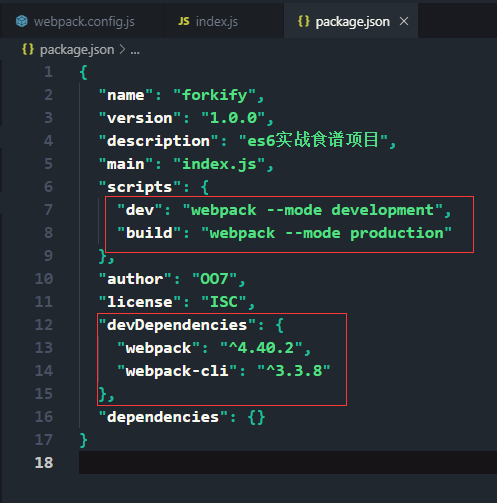
* 1. 在同一目录内



* 1. 至少二者文件之间要有 import/export联系，否则webpack将无法进化多文件打包

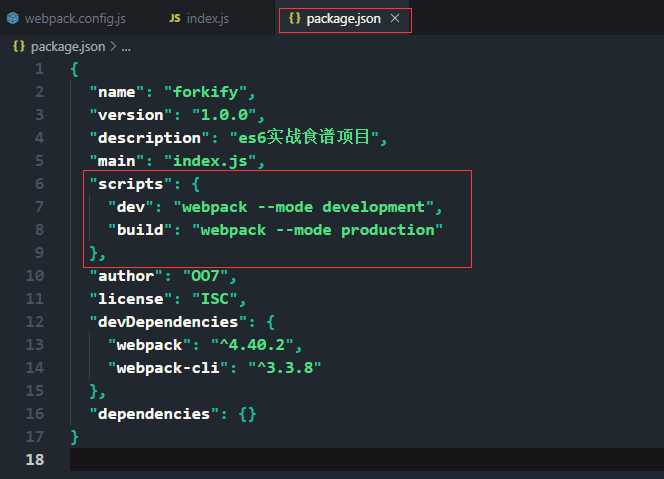


#### 安装Webpack-cli命令行操控

1. 安装命令npm install webpack-cli - -save–dev 让webpack可以进行命令操控
2. Package.json总体配置
   1. 
   2. Scripts: 为创建2条环境分支
      1. Dev: 为测试环境使用,代码未压缩,方便阅读
      2. Build: 为生成环境所需,代码被压缩,不方便阅读
   3. DevDependencies: 显示的为已安装模块，如现在已经安装“webpack”和”webpack-cil”

#### Webpack绑定生成文件

注意必备参数:



1. 生成未压缩代码,开发环境时使用: npm run dev
2. 生成压缩代码,生产环境时使用: npm run build

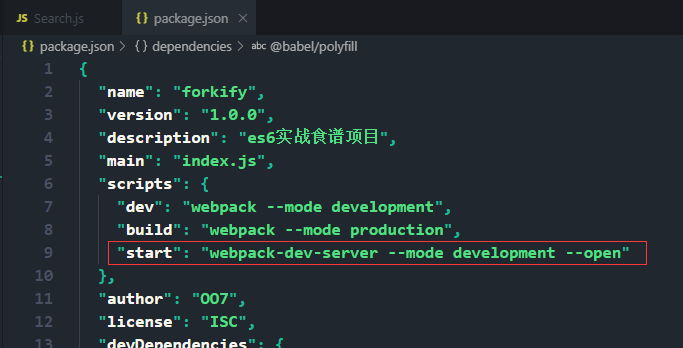
# 2019.9.19

## Web-dev-server安装( 实时浏览 )

### 安装指令 - 实时服务器, 文件发生修改时能直接显示在浏览器中

Npm install webpack-dev-server - -save-dev

### 配置package.json文件 –open代表一直保持运行状态



### 配置webpack.config.js文件

// 设置动态监听目录与web-dev-server配合

// 监听到js文件变动会直接显示在浏览器中



## Plugin安装( 自动生成HTML )

### 安装命令

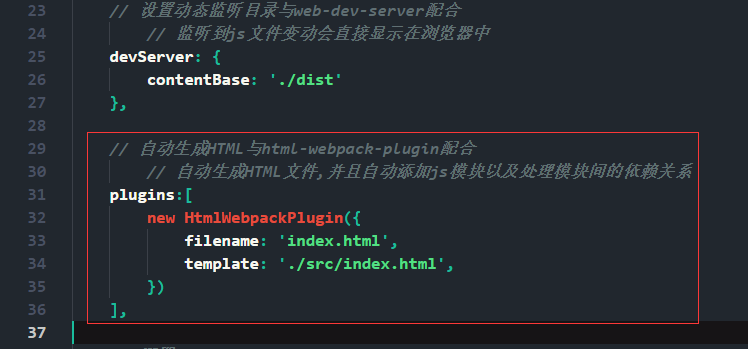
Npm install html-webpack-plugin - -save-dev

### 配置 webpack.config.js

// 自动生成HTML与html-webpack-plugin配合

// 自动生成HTML文件,并且自动添加js模块以及处理模块间的依赖关系

Template: 确定文件位置



### 使用plugin注意:

因为plugin不会将文件保存到硬盘中，所以需要npm run dev来生成的HTML写入到硬盘

## 安装Babel（将ES6/7/8..转换为ES5）

### 安装命令

npm install --save-dev @babel/core @babel/preset-env babel-loader

npm install --save @babel/polyfill

### 配置webpack.config.js

#### babel抓取js文件配置

***// 配置babel***

**module: {**

**rules:[**

**{**

**test: /\.js$/, *// 使用正则抓取JS文件***

**exclude: /node\_modules/, *// 不包含抓取node\_modules文件夹中的内容***

**use: {**

**loader: 'babel-loader' *// 使用babel-loader加载器***

**}**

**}**

**]**

**},**

### 创建并配置“.babelrc”文件

1. 创建文件”.babelrc”
2. 写入配置，支持IE8浏览器以上
3. **{**
4. **"presets": [**
5. **["@babel/env", {**
6. **"targets": {**
7. **"browsers": [**
8. **"last 5 versions",**
9. **"ie >= 8"**
10. **]**
11. **}**
12. **}]**
13. **]**
14. **}**

## 安装polyfill（使babel更好的支持es6到es5的转换）

### 安装命令:

npm install babel-polyfill - -save

### 配置webpack.config.js入口点配置:

Webpack是可以配置多个切入点的

entry: ['@babel/polyfill', './src/js/index.js'],

## 关于weboack安装模块引用方式( axios安装为例 )

1. 如安装axios npm install axios - -save ( axios低头fetch函数，比fetch函数更优秀牛逼 )
2. 安装完毕直接使用 import axios from ‘axios’ 导入即可, webpack将自动匹配已安装模块的路径

## webpack配置总结: 模板位置( 2019.9.16 )

1. Npm安装: npm –v node -v Npm init
2. Live-server安装（可以使文件夹成为临时服务器）: npm install live-server - -global
3. Webpack安装（打包js文件）: npm install webpack - -save–dev
4. Webpack-cli（可以使用命令行来操控webpack）: npm install webpack-cli - -save–dev
5. Web-dev-server（实时预览浏览器）: Npm install webpack-dev-server - -save-dev
6. Plugin（生成HTML文件）：Npm install html-webpack-plugin - -save-dev
7. Babel（高版本ES代码转ES5）:
   1. npm install --save-dev @babel/core @babel/preset-env babel-loader
   2. npm install --save @babel/polyfill
8. polyfill（使BABEL更好的支持ES6到ES5的转换）: npm install babel-polyfill - -save
9. axios（代替fetch函数,比fetch函数更加优秀方便）: npm install axios - -save
10. webpack配置注意事项： HTML文件不要手动导入JS文件，这会使JS代码重复执行

## 关于MVC模式

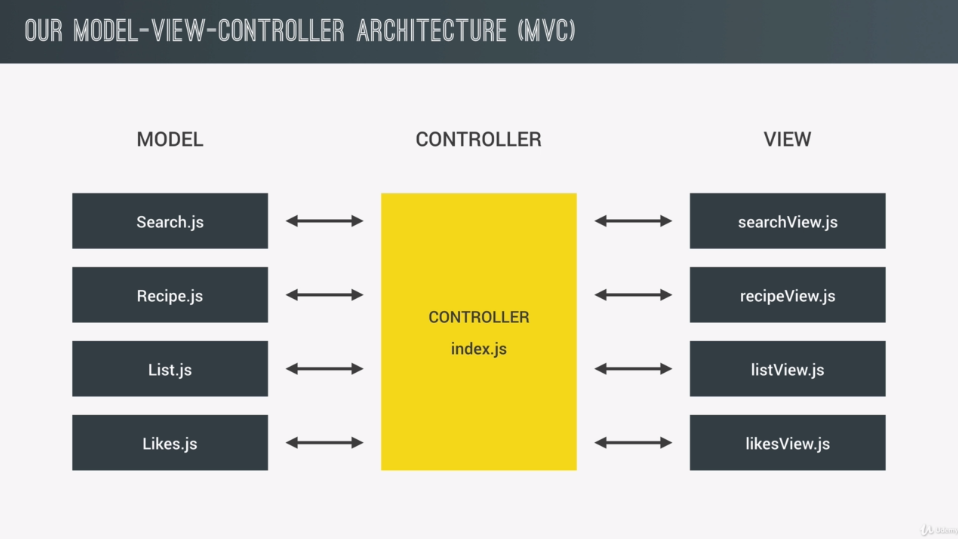
M – MODEL – 模块: 处理数据的功能模块

V – VIEW – 视图: 对于UI界面的交互显示

C – CONTROLLER – 主控: 程序主要逻辑操控

#### 参考图:

**注意:** Model的功能模块名称”开头大写”。



## ES6-关于import/export导入/导出的使用

### 注意：import/export使用条件

JS本不存在import/export 所以必须在webpack打包下只能正常使用此语法，因为webpack会将他们打包成同一文件

### Import/export导入/导出”单个数据”时

#### // 0. import 接受单个数据

// import test from './models/Search';

#### // 0. export default 导出单个数据

// export default '这是一个测试';

### Import/export导入/导出”多个数据”时

#### // 1. import 接受多个数据

// a) 括号法: 注意变量名称要对应

// as: 可以重命名变量名称

import { add, ID as oo7 } from './models/Search';

// b) 星号法: 接受全部返回值

// 引用方式: 像对象一样引用如: Search.ID

import \* as Search from './models/Search';

console.log(` ${Search.add(5,5)},${Search.ID} `);

// import多次调用，使用逗号隔开

import React, { Component } from ‘React’;

\*/

#### // 1. export导出多个数据方法

export const add = (a,b) => a+b;

export const ID = 7;

### // 2. export导出class或者function或者对象

#### // a) export导出class

export default class Search{

// 内容

}

#### // b) export导出function

export const Search = () => {

// 内容

}

## Export与export default的区别

**0、在一个文件或模块中，export、import可以有多个，export default仅有一个**

1、通过export方式导出，在导入时要加{ }，export default则不需要

这样来说其实很多时候export与export default可以实现同样的目的，只是用法有些区别。注意第四条，通过export方式导出，在导入时要加{ }，export default则不需要。使用export default命令，为模块指定默认输出，这样就不需要知道所要加载模块的变量名。

### 如：

var name="李四";

export { name }

//import { name } from "/.a.js"

可以写成：

var name="李四";

export default name

//import name from "/.a.js" 这里name不需要大括号

# 2019.9.21

## Forkify项目 – 食谱WEB程序实战

### 文件结构介绍，请查看2019.9.16/Forkify/Forkify文件结构.emmx 此文件

### 禁用默认效果 – .preventDefault();

Doucument.queryselector.search.addEventListener( 'submit', e => {

e.preventDefault(); // 禁用默认效果 } );

# 2019.9.22

## ES6-Array.join( 分隔符 )数组转换字符串，也可以设定分隔符

a = [ 12,22,33,44 ];

a.join(); // 生成字符串“12,22,33,44”

a.join(‘’); // 生成字符串”12223344”

a.join(‘-’); // 生成字符串”12-22-33-44”

# 2019.9.30

## Es6-closest()指定选择元素

### // 0. .closest()符合抓取标签

// 是: 返回标签

// 否: 返回NULL

### // 1. closest()指定抓取3种用法:

// a) .closest( '元素名称' ): 指定抓取返回标签,否则返回Null

// b) .closest( 'div' ): 默认多标签时抓最底层子类

// c) .closest( 'section > div' ): 会抓取最近子类div

// d) .closest( ':not(div)' ): 不抓取div元素,抓取其它元素

// 2. 通常与document.querySelector('xxx').target.closest('yyy');来合作抓取指定标签

const btn = document.querysSelector.target.closest('.btn-inline');

## // .dataset用法,标签data属性读取( 等待笔记 )

// 注意: 返回值为"字符串格式"

// 例:

// HTML: <a data-goto='1'></a>

// JS: document.querySlector('a').dataset.goto; == '1';

// 0. 其实.dataset将返回一个对象

// a) 此时的goto为对象中的变量，

// b) 索引.dataset.goto得到属性值

## 获取浏览器HASH值,监听URL变化( 浏览器#号后面的内容 )

const id = window.location.hash.replace('#',''); // 通常用来监听URL是否发生变化，配合鼠标事件单击元素的ID值

## 监听window事件触发函数

### 监听到URL的Hash值变化促发函数

Window.addEventListener( ‘hashchange’, function(){} );

### 监听到”加载页面”时促发函数

Window.addEventListener( ‘load’, function(){} );

## window全局变量

如: window.r = 100;

因为window为全局对象，所以增加的属性变量是可以全局访问的

## Eval()函数，让字符串成为执行语句

### // eval()函数, 让字符串成为执行语句

// 0. 如: eval( " a = () => { console.log('233') }; a(); " ) 成立;

// 1. 不过通常用来字符串计算使用

// a) 如: eval( "1+1" );

## Es6-对象属性便捷式写法

如:

cur = {

// 对象属性便捷式写法

num, // num, 相当于 num = num,

unit,

ingredients,

}

# 2019.10.5

## Math.ceil( num )向上取整

如:

Math.ceil( 0.95 ); // 1

Math.ceil( 1.2 ); // 2

# 2019.10.15

Css3属性选择器 [ 属性=”值” ]( 便捷式抓取标签ID与js配合 )

1. 抓取标签属性及属性值

<style>

// 0. 只抓取所有标签含有test值的标签

[class="test"] {

background: #ffff00;

}

// 1. 抓取所有标签类中含有test的标签 ，包括class=” xxx\_test ”

[class\*="test"] {

background: #ffff00;

}

// 2. 抓取含有对应属性的标签

[data-itemid=’123’]{

// 及可抓取对应标签

}

</style>

<div class="test" data-itemid = ‘123’ >The third div element.</div>

<p class="test" data-itemid = ‘222’>This is some text in a paragraph.</p>

## // .matches( css选择 )判断是否为指定的CSS选择标签, 返回True/False

// a) 可使用css多选如: el.traget.matches( ' btn-dec, btn-dec \* ' ) 常用与选中按钮( 因为matches返回布尔类型，所以可以与” btn-dec \* ”配合 )

// b) '.btn-dec \*': 的意思为选中所有子类标签

### matches在HTML中进行指定选择按钮方式，指定选中按钮

Document.querySelector.addEventListener( 'click', cur => {

if (cur.target.matches(' .btn-dec, .btn-dec \* ') ){

state.recipe.reviseServings( 'dec' );

recipeView.showServings( state.recipe );

}

else if( cur.target.matches(' .btn-inc, .btn-inc \* ' ) ){

state.recipe.reviseServings( 'inc' );

recipeView.showServings( state.recipe );

}

## 注意: 创建new时的事项

// 主控程序注意全局状态new创建

// 0. 一定要验证如果new已经创建，就没有在创建的必要。

if( !state.likes ) state.likes = new Likes();

## .findIndex()快速查找数组中元素内容，并返回索引值

### // 0. 通常findIndex() 与 splice() 配合

// a) findIndex()找出索引值

// 0. findIndex() 找到目标返回索引值, 否则返回-1

// b) splice()处理数组

### FINDINDEX() 与 SPLICE() 配合实战：

Const id = 666;

Like = [ class1, class2, class3 ];

Const pos = Like.findIndex( cur => cur.id == id ); // 找到符合条件的对象元素，并返回索引值，否则返回-1

If( pos >= 0 ) { like.splice( pos, 1 ) }; // 索引值大于负数，说明目标存在，在使用splice()处理数组

## // .setAttribute( '属性名称','属性值' )改变标签属性值( 等待笔记 )

document.querySelector( `.recipe\_\_love use` ).setAttribute('href',`img/icons.svg#${string}`);

## // map()配合join能轻松将数据转换多个HTML模板

### 使用map生成数组元素，每一个元素为HTML模板

const modelHTML = items.map( cur => {

cur = `

<div class=”test” >${x}</div>

`;

// 别忘记map必须要有return才能返回值

return cur;

} );

#### 在使用.join()将数组转换为字符串，在渲染到HTML

element.shoppingList.insertAdjacentHTML('beforeend', modelHTML.join('') );

## css3-number类型的input表单，step属性可以控制表单的单次加减数

<input type="number" value="${cur.num}" step="${cur.num}"> // step=””可以设置默认值翻倍数

# 2019.10.16( 第三方库总计 – 添加所有遇到的库 )

## 目录:

### 前端模块：

1. fractional | 分子分母转换库 | npm install fractional --save
2. axios | 请求API数据函数 | npm install axios - -save
3. uniqid | 自动生成ID库 | npm install uniqid - -save
4. scrollwatch | 滚动监听库, 当标签在用户视图时发生反馈,多用与动画效果配合 | npm install scrollwatch - -save
5. google font | 各式各样的的字体样式: <https://fonts.google.com/?subset=chinese-simplified&selection.family>
6. gh-pages | 在github上部署静态页面(React环境下) | yarn add gh-pages
7. node-sass | 这个是大坑，目的是webpack识别scss文件进行编译 | 具体搭建方法请看下方
8. bootstrap4 | 在create-react-app安装使用 | yarn add bootstrap
9. clipboard | 复制粘贴函数库 | 就是单击一下按钮就复制目标文本
10. slick | 滑动播放效果
11. ekko-lightbox | 灯箱图片展示效果
12. react-router-dom | react前端路由必备模块
13. redux | 代替react的state，他具有更加科学结构化模块化的state管理与使用方式
14. reselect | 与redux配合，保证只有在redux数据改变时对应的react组件才会重新渲染组件，提高react性能
15. react-modal | 强大的自定义弹窗
16. redux-persist | 配合redux浏览器本地存储变量数据，代替window.localStorage
17. styled-components | 为了解决css3全局问题，防止css3在未来扩展时出现冲突
18. redux-thunk | 让异步获取数据方式在redux中, 通常用于获取后端API数据

### 后端模块：

1. fs | 读取本地文件库 | const fs = require(‘fs’);
2. http | 基本网络模块 | const http = require(‘http’);
3. url | 基本网络模块 | const url = require(‘url’);

## 0. FRNUM = new Fraction( num ); 分子分母转换库

// 返回对象，分子，分母

// frNum.numerator 获取分子

// frNum.denominator 获取分母

### 实列

frNum = new Fraction(0.5);

num = `${frNum.numerator} / ${frNum.denominator}`; // 1/2

## axios | 请求API数据函数

优点: 返回的直接为对象数据，能精确的报错

### 实战

Async test(){

try{

const data = await axios(` 获取API数据的链接 `);

}catch( error ){

Console.log( error );

}

}

## uniqid | 自动生成ID库

Const id = uniqid();即可生成随机ID

## Scrollwatch | 标签监听库

### HTML: 监听滚动初始化

示例: <section data-scroll-watch id="banner" ></section>

### // JS: 监听滚动初始化

let sw = new watchTab({

watchOnce: true, // 是否开启不重复监听( 关闭则动画将来回执行 )

infiniteOffset: -10, // 偏移量

scrollThrottle: 200, // 延迟监听(ms)

// inViewClass: 'play', // 给当前视图增加类

onElementInView: function(data) { // 监听当前视图的标签

let activeId = data.el.id;

console.log( activeId );

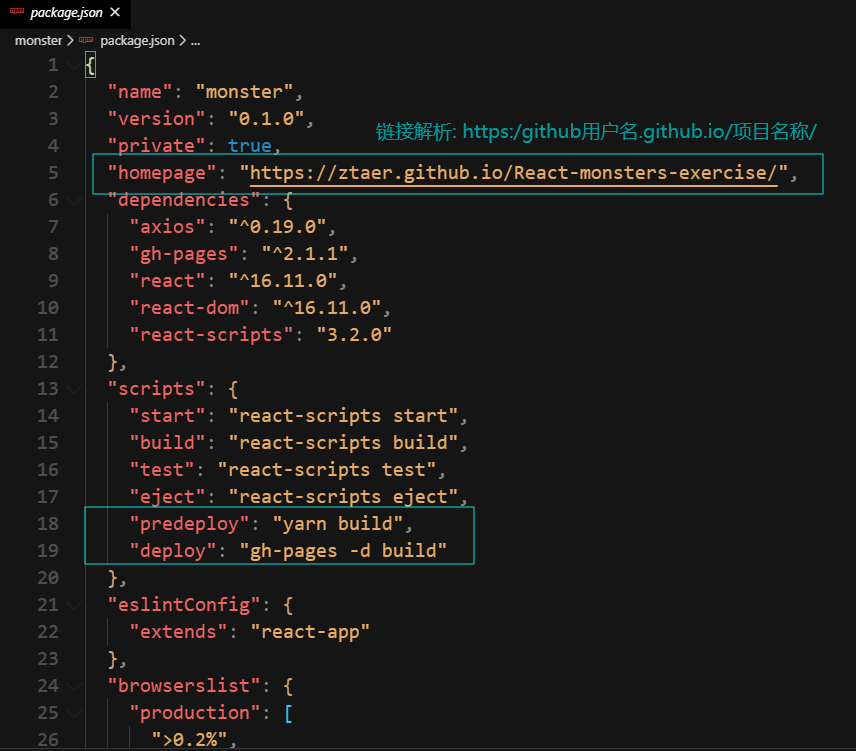
}

});

## 5.gh-pages | 在github上部署静态页面(react环境下)

0. 安装gh-pages: yarn add gh-pages

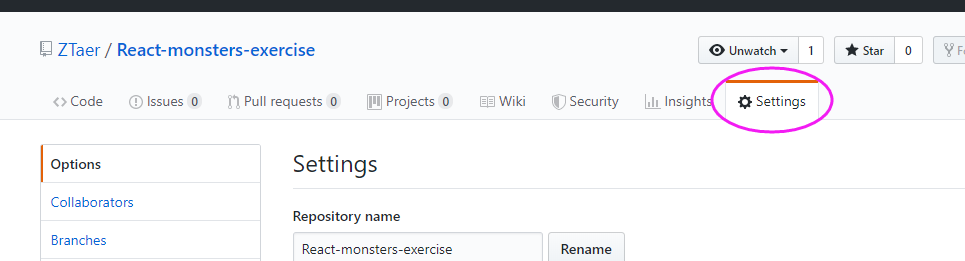
1. 配置package.json文件



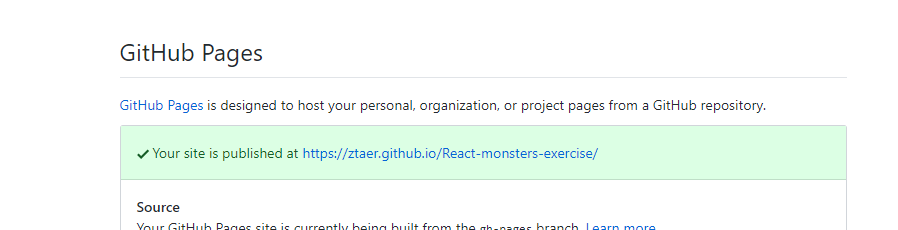
1. 运行命令: yarn deploy

#### 上传到GitHub，并查看链接

* 1. 找到对应项目的settings选项



* 1. 在下方即可找到部署的静态页面链接



## 6. node-sass | 这个是大坑，目的是webpack识别scss文件进行编译

### 注意所有需要使用的文件安装包在：“2019.12.3/node-sass环境安装包/” 目录下

1. Win10要想使node-sass能正常使用，需要基本条件如下：
   1. **环境： python2.7 + Visual Studio 2015, 2017 ,2019, 2013 全部安装上，预防万一**
      1. Python2.7环境构建最简单方法：
         1. 就是在安装node.js时，**直接勾选自动配置环境选项框**
         2. 2019最稳定node版本为v11.15.0：<https://nodejs.org/dist/v11.15.0/> (node-v11.15.0-x64.msi)
      2. Vs2013~2015下载：<https://support.microsoft.com/en-us/help/2977003/the-latest-supported-visual-c-downloads>
      3. Microsoft 生成工具 2015：<https://www.microsoft.com/zh-CN/download/details.aspx?id=48159>
   2. **安装PowerShell方便使用管理员权限命令（二选一）:** 
      1. Npm: npm install --global --production windows-build-tools ( 需管理员权限 )
      2. Yarn: yarn global add windows-build-tools
   3. **清除yarn/npm的缓存( 非常重要 )：**
      1. Npm清除方法:
         1. 删除项目中node\_modules 文件夹
         2. 执行: npm cache clean -f
      2. Yarn清除缓存:
         1. 删除项目中node\_modules 文件夹
         2. 执行：yarn cache clean
   4. **安装node-sass：**
      1. Npm要使用淘宝源( 及使能翻墙也要用它 ):
         1. npm install -g mirror-config-china --registry=http://registry.npm.taobao.org
         2. npm install node-sass
      2. yarn: 直接 yarn add node-sass

### node-sass环境搭建总结

1. Python2.7环境,使用node安装包，勾选自动配置环境选项
2. Vs2013~2015手动去微软下载
3. PowerShell：npm install --global --production windows-build-tools
4. npm和yarn都要清除缓存文件
5. npm使用淘宝源安装/yarn正常安装:yarn add node-sass

## 7. bootstrap4 | 在create-react-app安装使用 | yarn add bootstrap

0. 安装:

Npm: npm install --save bootstrap

Yarn: yarn add bootstrap

1. 使用：
   1. 比如在index.js文件下直接写入：import 'bootstrap/dist/css/bootstrap.css';

## 8. clipboard | 复制粘贴函数库 | 就是单击一下按钮就复制目标文本

0. 说明文档: <https://clipboardjs.com>

1. 注意事项目：**复制按钮.btn必须要有**

        <section id="menu" > 复制内容到剪切板 </section>

        <button data-clipboard-action="copy" data-clipboard-target="#menu" class="btn"   >

           <i class="far fa-copy"></i>

        </button>

<script>

// 复制初始化

            var clipboard = new ClipboardJS('.btn', {

                container: document.getElementById('modal') // bootstrap专用

            });

            clipboard.on('success', function(e) {

                // 当前活动 console.info('Action:', e.action);

                // 复制的文本 console.info('Text:', e.text);

                // 目标标签 console.info('Trigger:', e.trigger);

                e.clearSelection(); // 不显示选中效果

            });

            clipboard.on('error', function(e) {

                console.error('Action:', e.action);

                console.error('Trigger:', e.trigger);

            });

 </script>

## slick | 滑动播放效果

具体查看官网说明文档：<http://kenwheeler.github.io/slick/>

## 10. ekko-lightbox | 灯箱图片展示效果

具体查看官网说明文档：<https://ashleydw.github.io/lightbox/>

属性：**data-gallery=”xxx”**相同属性多个灯图可以相互切换

<a href="./assets/images/exp/cn.png"  data-toggle="lightbox" data-title="在菜单下方即可看到此选项"

data-gallery="img-gallery" data-width="1200" class="btn"> 中文翻译 </a>

<script>

// 灯箱初始化

    $(document).on('click', '[data-toggle="lightbox"]', function(event) {

        event.preventDefault();

        $(this).ekkoLightbox();

});

  </script>

## 11. react-router-dom | react路由必备模块( 是的你没有看错，路由是写在前端的 )

具体使用方式请查看下方: 2019.12.8（ 注意：请将遇到与此库有关的函数记录在其中，方便统一查询 ）

## 12. redux | 代替react的state，他具有更加科学结构化模块化的state管理与使用方式

具体使用方式请查看下方: 2019.12.15（ 注意：请将遇到与此库有关的函数记录在其中，方便统一查询 ）

## 13. reselect | 与redux配合，保证只有在redux数据改变时对应的react组件才会重新渲染组件，提高react性能

具体使用方式请查看下方: 2019.12.29（ 注意：请将遇到与此库有关的函数记录在其中，方便统一查询 ）

## 14. react-modal | 强大的自定义弹窗

安装：yarn add react-modal

在线实列：<https://codepen.io/ztaer/pen/wvBzvmg>

具体使用方式介绍：<http://reactcommunity.org/react-modal/>

### 个人使用分析（实战）

实战实列：文件JavaScript\_Study\2019.12.3\shopping-react\src\components\custom-modal

#### 弹窗component构建

import React from 'react';

import './custom-modal.style.scss';

import Modal from 'react-modal'; **// 弹窗必备导入**

import {connect} from 'react-redux';

import { handleCloseModal,handleOpenModal } from '../../redux/modal/modal.actions';

**// 锁定显示的父类标签(必备)**

var appElement = document.querySelector('body');

Modal.setAppElement(appElement);

class CustomModal extends React.Component {

    render(){

        const { handleCloseModal,handleOpenModal,showModal,text } = this.props;

        return(

            <div className="custom-modal" >

                <Modal

                contentLabel="onRequestClose Example" // 弹窗说明

                isOpen={showModal} // 布尔函数决定是否开启弹窗

                onRequestClose={handleCloseModal} // 关闭弹窗函数用于初始化布尔值

                closeTimeoutMS={0} // 弹窗打开时等待时间

                className="Modal" // 自定弹窗本身css名称,默认css名称.ReactModal\_\_Content

                overlayClassName="Overlay" // 自定弹窗背景css名称,默认css名称.ReactModal\_\_Overlay

                // onAfterOpen = {在弹窗打开后将运行的函数}

                // onAfterClose = {在弹窗关闭后将运行的函数}

                >

                    <div className="modal-title">

                        <button className="modal-btn" onClick={handleCloseModal}>

                            X

                        </button>

                    </div>

                    <div className="modal-content">

                        <p>

                            { text }

                        </p>

                    </div>

                </Modal>

            </div>

        );

    };

}

const mapStateToProps = ({ modal: { showModal,text } }) => ({

    showModal,

    text,

});

const mapDispatchToProps = dispatch => ({

    handleOpenModal: text => dispatch(handleOpenModal(text)),

    handleCloseModal: () => dispatch(handleCloseModal()),

});

export default connect(mapStateToProps,mapDispatchToProps)(CustomModal) ;

#### 弹窗redux构建

位置：E:\Study\JavaScript\_Study\2019.12.3\shopping-react\src\redux\modal

import ModalTypes from "./modal.types";

const INITIAL\_STATE = {

    showModal: false,**// 由他控制弹窗的打开于关闭**

    text: '后加内容，方便调用, 可选',

}

const modalReducer = ( state=INITIAL\_STATE, action ) => {

    switch(action.type){

        case ModalTypes.HandleOpenModal:

            return {

                ...state,

                showModal: true,

                text: action.payload,

            }

        case ModalTypes.HandleCloseModal:

            return {

                ...state,

                showModal: false,

            }

        default:

            return state;

    }

}

export default modalReducer;

## 15. redux-persist | 配合redux浏览器本地存储变量数据，代替window.localStorage

具体使用方式请查看下方: 2019.1.9（ 注意：请将遇到与此库有关的函数记录在其中，方便统一查询 ）

## 16.styled-components | 为了解决css3全局问题，防止css3在未来扩展时出现冲突

具体使用方式请查看下方: 2019.1.13（ 注意：请将遇到与此库有关的函数记录在其中，方便统一查询 ）

## 17. redux-thunk | 让异步获取数据方式在redux中, 通常用于获取后端API数据

具体使用方式请查看下方: 2019.1.28（ 注意：请将遇到与此库有关的函数记录在其中，方便统一查询 ）

## a) fs | 读取本地文件库

### // 2. \_\_dirname获取文件当前位置

a) console.log( **\_\_dirname** );

### // node.js-本地读取文件关于.readFile()与.readFileSync()的区别

// 0. riadFile()需要回调函数

// a) .readFile( 路径, 编码, 回调函数 );

// 1. readFileSync()无需回调函数

// a) .readFileSync( 路径, 编码 );

#### Fs.ReadFile:实战：

fs.readFile( `${\_\_dirname}/template-laptop.html`,'utf-8',( err, date ) => {

const laptop = laptopDate[id];

const result = replaceHTML( date, laptop );

res.end( result );

} );

#### FS.READFILESync:实战：

const laptopJson = fs.readFileSync( `${\_\_dirname}/data/data.json`, 'utf-8' );

# 2019.10.19

## JS-本地存储localStorage对象

1. 经测试localStorage为特殊的object对象
   1. 他能将数据存储到浏览器中，及使网页刷新也不会消散数据
   2. 但是他只能存储字符串类型，必须要通过第三方手段转换他存储的数据
2. localStorage语法
   1. 加入元素: localStorage.setItem( ‘test’,’test’ );
   2. 获取存储: localStorage.getItem( ‘test’ );
   3. 删除元素: localStorage.removeItem( ‘test’ );
   4. 获取存储数量: localStorage.length;
3. 经测试，同域名时为相通的localStorage对象，不同域名则不相通。

## JSON.stringify()超级字符串转换，常与localStorage()配合使用

1. JSON.stringify()可以将对象，数组，数字等转换为字符串
   1. 他的特殊之处在于，数组和对象保留格式

如：a = [ 1,2,3 ] -> JSON.stringify( a ) -> “ [ 1,2,3 ] ”

#### JSON.stringify()与localStorage()配合实列:

通常情况数组存储进localStorage.setItem( ‘test’, JSON.stringify( a ) );经过加工存储，这样方便数据还原

## JSON.parse()还原JSON.stringify()转换的数据，还原成原始类型数据

实列：

A = [ 1,2,3,4 ];

B = JSON.stringify( A );

JSON.parse( B ) === A;

# 2019.10.27

## Node.js浅浅的尝试

### 本地运行js文件

#### 手动运行文件

Node xx.js

#### nodemon自动运行js文件

1. 首先安装nodemon: npm install nodemon -g
2. 启动nodemon: 直接输入nodemon命令即可开启自动运行JS文件

### Json文件转换为对象类型

// 读取JSON文件

const laptopJson = fs.readFileSync( `${\_\_dirname}/data/data.json`, 'utf-8' );

// json格式转object格式

const laptopDate = JSON.parse( laptopJson );

## node.js创建路由

### // 创建路由( 等待笔记 )

// a) req: 为用户请求信息

// b) res: 为服务器返回信息

const server = http.createServer( ( req, res ) => {

### // 获取URL链接( 等待笔记 )

// 0. req.url如: 127.0.0.1:1337/test

// a) 得: req.url == 'test'

// 1. url.parse( req.url, true );解析url链接为对象

// a) 解析完后会生成一个对象，对象里有很多参数，这个以后深入学习node.js会用到

// b) .pathname: 访问的路径名称

// c) .query:

// 0. 链接: 127.0.0.1/laptop?id=123 -> 那么query的内容为 .query == { id: '123' };

// 1. 其实 ?x=y 相当于创建对象 { x: y };

// 2. ?x=y&?xx=yy 相当于创建对象 { x: y, xx: yy };

如:

const urlDate = url.parse( req.url, true );

const pathName = urlDate.pathname;

const query = urlDate.query;

## // 发送信息到前端

### 发送字符串类型包

// a) .writeHead( 标识码, { 'Content-type', '文件类型;charset=编码格式' } )

// 0. 200 - 访问正常

// 1. 404 - 访问失败

// b) .end( '返回内容' ); 后续代码将不在执行，类似于return

res.writeHead( 200, { 'Content-type': 'text/html;charset=utf-8' } );

end(‘ 一切正常 ’);

### 发送图片类型包

1. 首先使用fs.readFile打开图片
2. 然后使用.end在发送

fs.readFile( `${\_\_dirname}/data/img${pathName}`,( err, cur ) => {

res.writeHead( 200, { 'Content-type': 'image/jpg' } );

res.end( cur );

} );

## 正则表达式-抓取图片

(/\.(jpg|jpeg|png|gif)$/i).test( pathName ) // 验证字符串是否结尾是指定内容，是则返回true否则返回false

## 正则表达式-抓取全文指定内容，配合.replace()字符串替换函数

result.replace( /{%SCREEN%}/g, data.screen ); // “/xxx/g”抓取全文的xxx内容

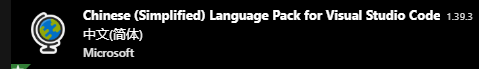
# 2019.10.31

## VSCODE前端必备插件

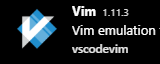
### 常用必备类

字体：Consolas, 'Courier New', monospace

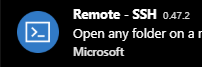
中文简体：Chinese (Simplified) Language Pack for Visual Studio Code



Vim快捷键: vim

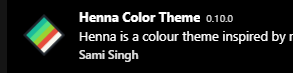


Ssh远程代码编辑：Remote – SSH

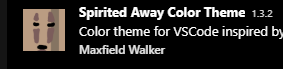


### 皮肤类

橘红皮肤(\*\*\*\*\*): Henna Color Theme

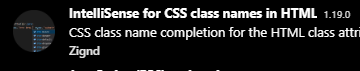


千与千寻皮肤(\*\*\*\*\*)：Spirited Away Color Theme

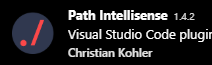


### 代码补全类

代码补全：IntelliSense for CSS class names in HTML



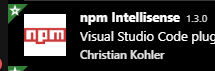
智能路径补全：Path Intellisense



Typescript代码补全：Auto Import



Npm模块名称补全：npm Intellisense

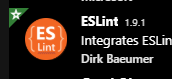


Es6代码补全：JavaScript (ES6) code snippets

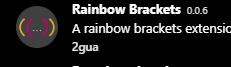


### 辅助书写布局类

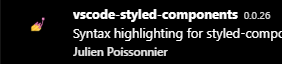
提示语法错误: ESLint



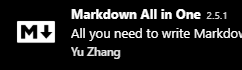
彩虹括号：Rainbow Brackets



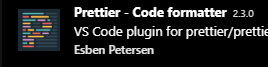
语法突出显示：vscode-styled-components



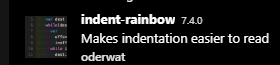
快捷键辅助: Markdown Support for Visual Studio Code



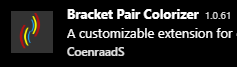
更漂亮的语法高亮：Prettier - Code formatter



提高对齐代码可读性：Indent-Rainbow

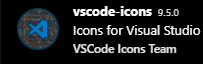


函数代码块区分：Bracket Pair Colorizer



### 图标类：

文件图标：vscode-icons

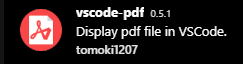


### 第三方文件查看类：

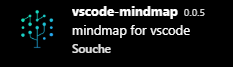
Svg查看：SVG Viewer



Pdf文件查看: vscode-pdf

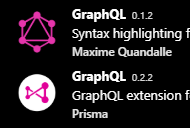


思路图插件：vscode-mindmap

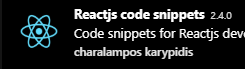


### React环境配置类

GraphQL：



React代码提示：Reactjs code snippets



### 代码测试类

右键直接运行代码(包括90%的语言): Code Runner



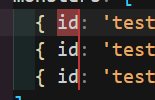
### 小技巧

#### vim插件安装后：多选快捷键修改：

如果安装vim插件则ctrl+d将有冲突

设置 --> ctrl+d --> ctrl+alt+d 在V视图即可正常使用

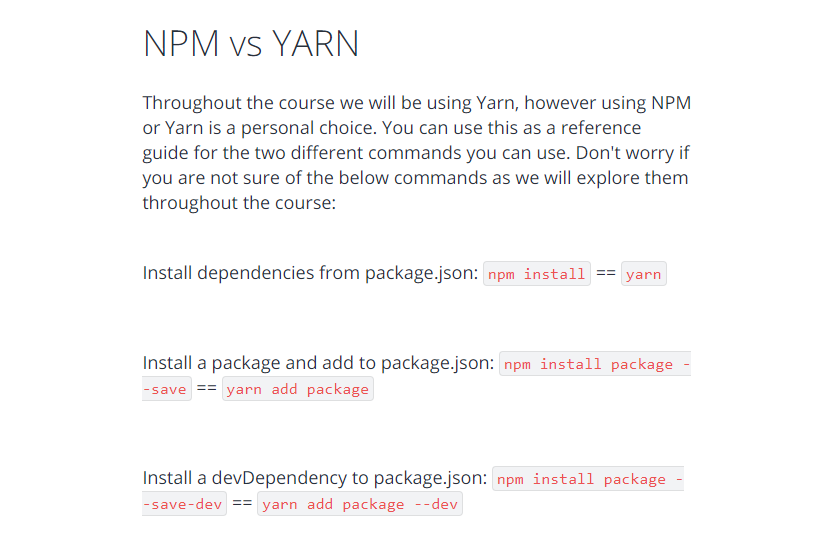




# 2019.11.3( React )

## 开始学习React

### Npm与yarn常用命令



### Yarn升级库

0. 查看当前已安装库版本: yarn list xxx yyy

1. package.json文件中如果版本号"^16.0.3"代表始终保持最新版本

2. yarn upgrade 进行版本升级

3. 在运行yarn install 确保更新即可

3. yarn list xxx查看库目前版本

### Npm升级库

1. npm list xxx yyy产看当前库版本
2. 同样的方法修改package.json
3. npm update即可升级
4. npm install 确保更新即可
5. npm audit fix **修复涉嫌有安全风险的库**

## React环境搭建

### 零：安装node

1. Node官网：<https://nodejs.org/en/>
2. 下载安装包根据提示,下一步下一步单击即可
3. 命令行中node -v测试是否正常

### 一: 安装yarn,代替npm

1. Yarn官网: <https://www.yarnpkg.com/zh-Hans/>
2. 下载安装包根据提示,下一步下一步单击即可
3. 命令行中测试 yarn -v 验证是否安装成功

### 二：安装creact-react-app快速搭建react基本环境

**他继承了webpack，babel兼容，以及react，使用户快速方便的搭建最佳的react环境**

#### Npm安装create-raect-app( 如果不支持npx安装时 )

参考链接：<https://react-legacy.netlify.com/docs/installation.html>

npm install -g create-react-app // 安装

create-react-app xxx // 创建react项目

cd xxx

npm start

#### create-react-app创建react项目

参考链接：<https://reactjs.org/docs/create-a-new-react-app.html>

npx create-react-app xxx

cd xxx

yarn start/npm start

## React自定义标签

### // 自定义标签-总体情况

// 0. 因为class开头大写, 自定义标签名称与class名相同

// 1. 可以使用自定义标签的情况:

// a) 类情况: class Xxx extends Component{ render( return( JSX ) ); }

// b) 函数情况:

// 0. const Xxx ()=>( JSX ): 小括号其实具有欺骗,编译器的效果,使编译器直接执行/返回小括号内容

// 1. const Xxx ()=>{ return (JSX) }

// 2. function Xxx(){ return (JSX) }

// 3. <Xxx/>自定义标签转换过程

// a) 自定义标签( JSX -> 自定义标签 -> HTML ):

// 4. JSX名称要开头大写( 否则报错 )

// 5. Export方都要有import React from ‘react’;

// 6. 注意: return( 只能有一个JSX顶级标签，也就是说多个标签时只能嵌套在一个div中 );

### // 自定义标签-函数形式写法,属性传输,以及重用性

/\*\*

#### import方:

import { CardList } from '...';

<Cardlist xxx="zhao" >

<h1> \_\_oo7\_\_ </h1>

</Cardlist>

<Cardlist name="zhao" /> // 可以多次重用,像函数一样不受其它影响

#### export方:

a) props.children: 是自定义标签中的JSX内容

b) props.xxx: 为自定义标签中的属性内容

c) props为对象类型, 存储着自定义标签的参数

d) 注意: 必须要引入react库: import 'React' from 'react';

import 'React' from 'react';

export const CardList = props => {

console.log( props );

return (

<div>

{ props.children }

<h1> { props.name } </h1>

</div>

);

};

### // 自定义标签-对象属性传输(非props方法 )

1. **目的:** 在不借助props变量情况直接获取传递来的属性
2. **注意:** 二者时间**属性名称要一致**

#### IMPORT方:

import { CardList } from '...';

<SearchBox

placeholder='搜索怪物名称-测试实时搜索功能'

searchChangeEvent={ this.searchChangeEvent }

/>

#### EXPORT方( 接受属性数据 )

export const SearchBox = ( { placeholder, searchChangeEvent } ) => {

return (

<input

className='search'

type='search'

placeholder={ placeholder }

onChange={ searchChangeEvent }

/>

);

}

### 自定义标签-类形式写法

#### // Component.render()渲染JSX到前端

// 0. 函数本身为Component类中的函数, 经过蓝图继承, 可直接调用蓝图函数.

// 1. 使用方法:

// a) 前提条件是在Component类下

// b) render(){ return( JSX内容 ) };

#### 自定义标签-类形式总体源码架构：

import React,{ Component } from 'react';

class App extends Component {

constructor(){

super(); // 继承蓝图属性

this.state = {

string: '这是一个测试',

};

}

render(){

return(

JSX内容

);

}

## 关于JSX

### // JSX是React模仿HTML的内容( 等待笔记 )

// 0. JSX大致于HTML相似，只不过更加方便改变参数在JS中

// 1. 注意className == class: 目的是为了区分JS的class

// 2. JSX中大括号内才可以写JS语法: 如src={ logo }

// 3. JSX导入css(react导入css): import 'xx.css';注意JSX,className对应html的class名称

### // JSX-关于onClick={ function }用法( 等待笔记 )

// 0. 在JSX中添加此方法后，当用户单击时，将重新执行render()函数，也就是前端界面将重新渲染

// 1. 经常使用此方法来改变某个变量参数, 来反应到前端

// **如:** onClick={ ()=>{ this.setState({ string: 'change this' }) } }

### 关于this.setState( {} )便捷式改变this.state属性参数

#### // this.setState({ string: 'test' }); 相当于 this.state.string = 'test';

// 0. 此方法能轻松的改变对象属性参数

// 1. 注意: 只能使用在this.state对象中, 其它名称对象不行

### JSX与this.setState实战源码

<div className="App">

<header className="App-header">

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

<p>

{ this.state.string }

</p>

<button onClick={ () => { this.setState({ string: '测试onClick函数，以及this.setState({})直接访问对象属性的写法' }) } } >

Click Change Text

</button>

</header>

</div>

## // Component-关于componentDidMount()函数

// 0. 经常用于请求数据,改变this.state数据,以至于react又要调用render()渲染界面。

// 1. 执行步骤：componet 🡪 render() 🡪 componentDidMount() 🡪 render()

#### 具体源码：

import React,{ Component } from 'react';

class App extends Component{

constructor(){

super();

this.state = {

monsters: [],

}

}

componentDidMount(){

// async异步请求数据

// 具体实战代码请查看: 2019.10.27/monsters-rolodex/src/App.js

}

render(){

return(

// JSX内容

);

}

}

## // map()函数渲染JSX方法使用( 用于循环数组, 只不过在这里是的内容是JSX )

// 0. 使用map函数的优势:

// a) react非常的聪明, 如果map循环的数据中有变化, 那么他只会渲染发生变化的内容, 不会全部重新渲染

// b) 这样的方法大大提高了效率

// 1. JSX中map函数使用注意事项:

// a) JSX中大括号中才能写JS语法: { JS语法 }

// b) key属性，非常重要react用于标识不同的标签,

// 0. 有map()的地方一定要有key配合

// 1. key一般存储id, 放置于JSX标签属性

// c) 如果有cur => { return xxx; }别忘记return, map()才能重组数组;

// 2. map()无return写法

// { this.state.monsters.map( cur => <h1 key={ cur.id } > { cur.name } </h1> ) }

// { this.state.monsters.map( cur => (<h1 key={ cur.id } > { cur.name } </h1>) ) }

// 关于对象+解构+map请看“2019.12.3\shopping-react\src\components\directory-menu\jsx文件”

// 3. map()有return写法, 如下:

export const CardList = props => {

return (

<div className="card-list" >

{

props.monsters.map( cur => { return( <Card key={cur.id} monster={cur} /> ); } )

}

</div>

);

};

## 指定位置渲染自定义标签REACTDOM.RENDER()

### // ReactDOM.render()指定ID标签,渲染JSX内容到前端

// 0. <App />: 自定义标签

// 1. 使用方法: ReactDOM.render( <自定义标签 />, 渲染位置 );

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

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

## //React合理的文件结构 – react标准文件结构

标准的react文件结构，查看实列：****JavaScript\_Study\2019.12.3\shopping-react\src****

// 0. webpack最终将会把他们打包一起

// 1. 关于JSX文件结构化:

// a) jsx文件起名: “类名.文件夹.jsx”

// b) 示例:

/\*\* components文件夹:

\* --card文件夹

\* ----card.components.jsx

\* ----card.style.css

\*

\* --search-box文件夹

\* ----search.components.jsx

\* ----search.style.css

\*/

## API: 有趣的API接口

### // 索引怪物图片API:

**${props.monster.id}: 随机ID即可**

**const monsterImg = `https://robohash.org/${props.monster.id}?set=set2&=180x180`;**

# 2019.11.4

## Array.filter( function )数组过滤器, 返回一个符合条件的数组，原数组不变

**True 返回数组元素，False 不返回数组元素**

Const test = [ 2,3,4,5,6 ];

test.filter( e => e > 4 ); **// 返回的数组为: [ 5, 6 ]**

test.filter( ( cur, index )=>{xxx} ); **// cur为传递的数组元素，index为当前元素在数组中的’索引值’**

# 2019.11.9

/\*\*

## react中class函数传递( class类中可以使用'箭头函数方便函数传递 )

\*/

// 0. 原因: 在react中如果将class函数传递给其它作用域，普通class函数因无this会报错必须借用this.xxx.bind(this);才能正常传递使用

// 1. 作用: class中如果使用箭头函数，直接可以传递使用this.xxx; 在export放

// 2. 解析: 因为'箭头函数'可以继承父类的所有方法属性.因为他们同属于一个作用域

// 3. 实战:

/\*\*

\* import React,{Component} from 'react';

\*

\* class App() extends Component{

\* constructor(){

\* super();

\* this.state = {

\* xx: '',

\* };

\*

\* this.tt.bind( this ); // 普通方法, 有他才能正常传递使用

\* }

\*

\* // 普通方法

\* tt(){

\* this.setState({ xx: 'xxx' });

\* }

\*

\* // 箭头函数方法

\* bb = ()=>{

\* this.setState({ xx: 'xxx' });

\* }

\*

\* render(

\* <DivCard1 onChangeEvent={ this.tt } /> // 普通方法, 需bind()配合

\* <DivCard2 onChangeEvent={ this.bb } /> // 箭头函数方法, 直接调用

\* );

\* }

\*/

## /\*\*\*this.setState异步与回调函数

\* 0. this.setState( { 改变值 }, 改变后执行的函数 ) -> this.setState( { xxx: xxx }, ()=>{} );

\* 1. 因为this.setState( {}, function )为异步运行的，所以要在同setState下的，另写一个function确定修改值的准确性。异步可以保证其它js正常运行，可以直接应用到异步需求的搜索栏

\*

使用方式：this.setState( { searchField: e.target.value }, ()=>{console.log(this.state)} )

\*/

## /\*\*\*JSX属性onChange={}表单内容发生改变触发函数

\* 0. onChange = { e => {} }

\* 1. e: 包含表单标签的属性，常用e.target.value获取表单输入值

\* 2. 如: <input type='search' onChange={ e => { 注意: 改变this.state值会重新执行render函数 } } />

\*/

# 2019.11.13

## 关于React如何聪明高效的证实

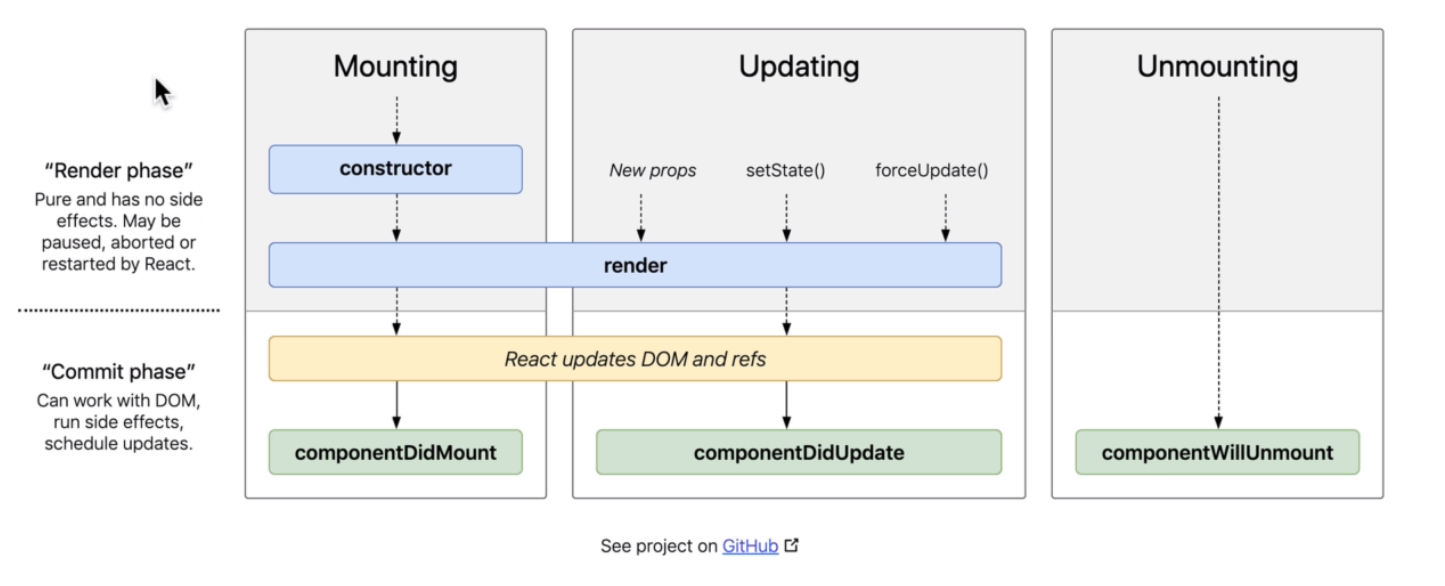
### 谷歌浏览器 - 查看界面动态刷新：

1. 调试器下拉菜单中 -> more tools -> Rendering ->
2. 勾选Paint flashing即可看到页面的动态变化
3. 开启此功能后在React代码中即可开始测试

### React高效的原因：

就是为了证明React是非常高效的：**他只渲染改变的内容，不变的内容是不会进行多余的重复渲染的**

## React核心: 组件的生命周期流程



原文链接： <http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>

**实战项目在：**2019.11.13/lifecycle/components/lifecycleTest/**lifecycleTest.component.jsx**

### React组件生命周期分为3个大部分:

/\*\*

 \* React组件生命周期分为3个大部分:

 \* 0. Mounting挂载 -> Updating更新 -> Unmounting卸载

 \* 1. Mounting挂载:

 \*   a) 运行顺序: constructor() -> componentWillMount() -> render() -> componentDidMount()

 \*      0. 函数解析:

 \*          a) componentWillMount() - 准备首次渲染组件: 常用于请求数据

 \*          b) componentDidMount() - 组件渲染完成: 常用于请求数据

 \* 2. Updating更新:

 \*   a) 促发条件: this.props/this.state发生改变，forceUpdate()强制更新

 \*   b) 运行顺序: shouldComponetUpdate() -> render() -> componentDidUpdate()

 \*      0. 函数解析:

 \*        a) shouldComponentUpdate() - 决定是否渲染组件: 常用于判断是否数据真正的发生改变,来决定是否渲染

 \* 3. Unmounting卸载:

 \*   a) 促发条件: 当释放掉此组件时, 由父类级决定不使用目前组件时

 \*   b) 运行顺序: componentWillUnmount()

 \*      0. 函数解析:

 \*          a) componentWillUnmount() 标注着: 组件代码将被内存释放掉, 这是React高效的表现, 且无需使用的组件不会占用系统资源

 \*/

import React from 'react';

class LifeCycle extends React.Component{

    constructor(){

        super();

        console.log( '@constructor' );

    }

    componentDidMount(){

        console.log( '@componentDidMount' );

    }

    componentDidUpdate(){

        console.log( '@componentDidUpdate' );

    }

    componentWillMount(){

        console.log( '@componentWillMount' );

    }

### shouldComponentUpdate()决定是否渲染组件

    /\*\*

     \* shouldComponentUpdate()决定是否渲染组件

     \*/

    // 0. 模型:  shouldComponentUpdate( nextProps, nextState ){ return true/false };

        // a) 二个参数:

            // 0. nextProps: 发生改变的this.props属性

            // 1. nextState: 发生改变的this.state属性

        // b) return true代表可以渲染组件

        // c) return false代表不可以渲染组件

    shouldComponentUpdate( nextProps, nextState ){

        console.log( '@shouldComponentUpdate');

        console.log(nextProps,nextState);

        return true;

    }

    componentWillUnmount(){

        console.log( '@componentWillUnmount' );

    }

    render(){

        console.log( '@render' );

        return (

            <div className="lifeCycle" >

                <h1>生命周期测试</h1>

                <p> {this.props.state.text} </p>

            </div>

        );

    }

}

export default LifeCycle;

# 2019.11.14

## class中箭头函数和普通函数调用实战

  /\*\*

   \* class中箭头函数和普通函数调用实战

  \*/

  // 0. 箭头函数调用更加便利

  // 1. 普通函数调用较为麻烦

  // 0. 箭头函数

  clickOnChangeAdd = () => {

    this.setState({ age: this.state.age+1 });

  }

  // 1. 普通函数

  clickOnChangeLess(){

    this.setState({ age: this.state.age-1 });

  }

  render(){

    return (

        <div className="App">

          <header className="App-header">

            <p>

              { this.state.age }

            </p>

### // 箭头函数调用

// 箭头函数调用

            <button onClick={ this.clickOnChangeAdd } >

              增加

            </button>

### // 普通函数调用方法

// 普通函数调用方法

            <button onClick={ **() => this.clickOnChangeLess()**} >

              减少

            </button>

          </header>

        </div>

     );

  }

## REACT中this.setState()的本质用法

class App extends React.Component{

  constructor(){

    super();

    this.state = {

      age: 48,

    };

  }

  clickOnChangeAdd = () => {

    /\*\*

     \* this.setState()的本质

     \*/

    // 0. this.setState()为异步函数, 属于Component组件

    // 1. 本质用法: this.setState( ( prevState, prevProps )=>{ return {} }, ()=>{ '衔接异步数据处理' } );

        // a) 其实他有二个回调参数:

           // 0. prevState: 相当于复制this.state数据

           // 1. prevProps: 相当于复制this.props数据

           // 2. prevState/prevProps名称可变

        // b) 只所以异步是因为效率, React能根据最佳时机来改变this.state的数据

        // c) 正是因为有prevState,prevProps二个回调参数为对象类型, 他才诞生了‘正常用法必须使用对象的形式‘

        // d) 注意: this.state函数返回中有return: return { age: 'xxx' }依旧为对象类型

    // 2. 正常用法: this.setState( { xxx: 'yyy' }, ()=>{ '衔接异步数据处理' } );

    // 3. 关于this.props

        // a) this.props属于 Component组件的默认属性

        // b) 它包含的是'自定义标签'传递来的标签属性

            // 0. 如: < APP test="123" string="hahah" / > 那么this.props == {test:'123', string='hahah'};

    this.setState(

**(** prevState, prevProps**) => {**

          console.log( prevState , prevProps ); // {age:48} , {test:123}

          return { age: prevState.age + prevProps.addNum  };

**},**

      () => console.log( this.props ) // {test:123}

    );

  }

}

## class简洁版 - 依据ES6,class

import React from 'react';

/\*\*

 \* class简洁版 - 依据ES6,class

 \*/

// 0. 无需constructor,因为他在class默认存在

// 1. 属性直接写即可，他会自动认定数据进入constructor下

// 2. 函数无变化

class App2 extends React.Component{

**state = {**

**age: 12,**

**}**

  addAge = ()=>{

    this.setState( { age: this.state.age + 1 } );

  }

  render(){

    return (

        <div className="App2">

            <p>

              { this.state.age }

            </p>

            <button onClick={ this.addAge } >

              增加

            </button>

        </div>

     );

  }

}

## 在没有WebPack情况下使用React

<!DOCTYPE html>

<html lang="zh">

<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">

    <title> 在没有webpack的情况下使用react </title>

</head>

<body>

    <div id="main-body">

    </div>

### CDN-React索引必备

    <script crossorigin src="https://unpkg.com/react@16/umd/react.production.min.js"></script>

<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>

    <script>

### // React.createElement()创建标签函数; - 无webpack

 // React.createElement()创建标签函数; - 无webpack

            // 0. 三个参数React.createElement( '标签名称', {标签属性}, '标签内容' );

                // a) 标签名称: 按照H5标准/或者React的自定义标签名称

                // b) 标签属性: 按照JSX标准来

                    // 0. 例: { className: 'text-light display-2' } -> class="text-light display-2"

                // c) 标签内容:

                    // 0. 可以再次嵌套React.createElement()

                    // 1. 字符串/或变量都可以

                    // 2. 多个嵌套: React.createElement( 'div', { className: 'test' }, [ xxx, xxx ] );

        // 实例: React.createElement()正常使用

        const App = ()=>{

            return React.createElement( 'h1', {}, '测试React.createElement()函数使用'  );

        }

        // 实例: React.createElement()嵌套使用

        const App1 = () =>{

            return React.createElement( 'div', {}, React.createElement( 'h2', { className: 'text-light display-2' } ), 'React.createElement()函数可以嵌套使用' );

        }

        // 实例: React.createElement()多个嵌套使用

        const App2 = ()=>{

            return React.createElement( 'div', {}, [ React.createElement(App),React.createElement(App1),React.createElement(App1), ] );

        }

### function组件写法 - 无webpack

        /\*\*

         \* function组件写法 - 无webpack

        \*/

            // 0. 只是以前return JSX的地方让React.createElement()代替

        const Ftest = () => {

            return React.createElement( 'p', {}, 'return JSX的地方, 都使用React.createElement()代替' );

        }

### \* class组件写法 - 无webpack

        /\*\*

\* class组件写法 - 无webpack

        \*/

            // 0. 只是以前return JSX的地方让React.createElement()代替

        class App3 extends React.Component {

            constructor(){

                super();

                this.state = {

                    string: '无webpack, class组件写法',

                };

            }

            render(){

                return React.createElement( 'h3', {}, this.state.string );

            }

        }

### \* 渲染组件ReactDOM.render()多条自定义标签渲染 - 无webpack

/\*\*

 \* 渲染组件ReactDOM.render()多条自定义标签渲染 - 无webpack

        \*/

            // 0. 在没有webpack情况下: ReactDOM.render( 待渲染标签数组, 渲染位置 );

                // a) 待渲染标签数组：

                    // 0. 需要React.createElement()配合使用

                    // 1. 多个标签渲染: ReactDOM.render( [ xxx, xxx ], 位置 );

                // b) 渲染位置: 起使用方式并没有变化

        ReactDOM.render( [React.createElement(App), React.createElement(App1), React.createElement(App2), React.createElement(App3) ], document.getElementById('main-body') );

    </script>

</body>

</html>

# 2019.12.6

## JSX-关于标签属性style: 其实为对象类型，它的css属性需要按照对象的规则来写入

### JSX:

<div style={ {

            backgroundImage: `url(${imageUrl})`  // jsx中标签属性style为对象类型

        } } className="background-image">

     </div>

## JSX-className多类名写法

**// 0.** jsx需要{}大括号内才可以使用js语法

**// 1.** 使用ES6` ${} `字符串处理套装

<div className={ ` menu-item ${size} ` }>

# 2019.12.8( react-router-dom )

## react-router-dom | react前端,路由必备模块

### React路由模块安装

安装模块：yarn add react-router-dom

如果出现安装问题，则直接，删除项目中的缓存，重新yarn install。

### BrowserRouter - 函数路由初始化

1. 在ReactDOM.render()下要渲染的标签使用 <BrowserRouter>标签
2. 这样在此标签下的其它路由函数才能生效

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter } from 'react-router-dom';

import './index.css';

import App from './App';

ReactDOM.render(

    // 展示路由功能的基础标签

    <BrowserRouter>

        <App /> // 待渲染自定义标签

    </BrowserRouter>,

    document.getElementById('root')

);

### Route/SWITCH – 函数的使用

import { Switch,Route } from 'react-router-dom';

// 测试路由 - 0

const HatsPage = () => ( <div> <h2 className="display-2" > TEST - Router </h2></div> );

const TestPage = () => ( <div> <h2 className="display-2" > TEST2 - Router </h2></div> );

function App() {

  return (

    // Route标签使用:

      // 0. 模型: <Route exact={true/false} path="/" component={JSX} >默认3个参数

      // 1. exact属性:

        // a) exact为是否开启精确路由访问

        // b) 如果Route标签不写exact: 则默认exact为false否则为true

        // c) 关于exact属性的使用:

          // 0. 在<Switch>内，

            // a) 不开启exact: 则只能访问'/'路由页面

            // b) 开启exact: 则可以访问精确路由页面

          // 1. 不在<Switch>则进行叠加JSX渲染,就是根目录页面+指定路由页面渲染在同页

      // 2. path属性:

        // a) path="/"按照web路径来写

      // 3. component属性:

        // a) component={JSX}: 放置渲染的JSX变量( 原自定义标签名称 )

    <div className="App">

      <Switch>

        <Route exact={ true } path='/' component={HomePage} />

        <Route path='/hats' component={HatsPage} />

        <Route path='/test' component={TestPage} />

      </Switch>

    </div>

  );

}

export default App;

### 关于Route标签传递的参数props -当前页面路由信息

### Link - 函数使用及Link标签的使用

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

// 测试路由 - 1

// 关于Route标签传递的参数props

  // 0. match属性对象:

    // a) url: 保存‘当前’在浏览器中的url链接( 仅限于保存符合路由规则的url,想要完整的url信息请看props.location.pathname属性解析 )

      // 0. 常用于创建动态路径( 具体看下方实战代码 )

    // b) path: 保存在Route标签中的path属性内容

      // 0. 如: <Route path='/pro/:proID' component={ProTextTest} /> 那么 props.match.path == '/pro/:proID';

    // c) isExact: 当url符合path规定的路由路径时为True, 否则为false

    // d) params: 常用于它的参数来配置'动态路由'页面跳转

      // 0. 常与Route标签中 path='/:xxx'下的':'符号配合, 因为它能让’:xxx’变为params的属性

      // 1. 如:

        /\*\*

         \* 路由配置: <Route path='/pro/:proID' component={ProTextTest} />

         \* url访问: xxxx/pro/123

         \* 那么params对象属性有: params = { proID: 123 }

         \* \*\*/

  // 1. history属性对象:

    // a) history下有很多函数属性,目前最常用的为push函数,用于跳转指定url页面,与Link标签功能相似

    // b) push(): 跳转指定url

      // 0. 如: <button onClick={ ()=> props.history.push('/pro') } > 产品页面 </button>

  // 2. location属性对象:

    // a) pathname: 保存完整的‘当前url信息’

      // 0. 如:

        /\*\*

         \* 路由规则: <Route path='/pro/:proID' component={ProTextTest} />

         \* 访问域名: http://localhost:3000/pro/123123/123123

         \*/

        // a) props.location.pathname = "/pro/123123/123123" ;

        // c) props.match.url = "/pro/123123";

        // b) 与match.url的区别: 在于pathname保留完整的url信息，但math.url只保存符合路由规则的url信息

// Link标签库：直接跳转到指定url

  // 0. 如: <Link to="/" > 返回主页 </Link>

  // 1. 在前端界面将被渲染为‘a标签’

const HomePageTest = props => {

  console.log( '1',props );

  return (

    <div>

      <h2 className="display-1" > 主页 </h2>

      <button onClick={ ()=> props.history.push('/pro') } > 产品页面 </button>

    </div>

  );

}

const ProTest = props => {

  console.log( '2',props );

  return (

      <div>

        <h2 className="display-1" > 产品页 </h2>

        <Link to="/" > 返回主页 </Link>

        <Link to={ `${props.match.url}/13` } > 产品13 - match.url实战 </Link>

        <Link to={ `${props.match.url}/14` } > 产品14 - match.url实战 </Link>

      </div>

    );

}

const ProTextTest = props => {

  console.log( '3',props );

  return ( <div> <h2 className="display-1" > 产品详情页ID: { props.match.params.proID } </h2></div> );

}

function App() {

  return (

    <div className="App">

      <Switch>

        <Route exact path='/' component={HomePageTest} />

        <Route exact path='/pro' component={ProTest} />

        <Route path='/pro/:proID' component={ProTextTest} />

      </Switch>

    </div>

  );

}

### withRouter函数 - 使当前页面下的组件, 可以直接访问"当前页面路由信息"

路由信息就是前面提到的，什么history，match，location等

#### withRouter函数用法

import React from 'react';

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

import "./menu-item.style.scss";

/\*\*

 \* 路由withRouter函数: 使当前页面下的组件, 可以直接访问"当前页面路由信息"

 \* 0. 引入: import { withRouter } from 'react-router-dom';

 \* 1. 输出: export default withRouter( 组件名 );

 \* 2. 使用: 路由属性名称 -> 直接传递给组件, 如下实战

 \*/

const MenuItem = ({ title,imageUrl,size,linkUrl,history,match }) => (

    <div

    onClick={ ()=>history.push( `${match.url}${linkUrl}` ) }

    className={ ` menu-item ${size} ` }

    >

        <div style={ {

            backgroundImage: `url(${imageUrl})`

        } } className="background-image">

        </div>

        <div className="content">

            <h1 className="title">

                {title.toUpperCase()  }

            </h1>

            <span className="subtitle">

                SHOP NOW

            </span>

        </div>

    </div>

);

export default withRouter(MenuItem);

#### withRouter对象用法

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

class SignIn extends React.Component {

**// withRouter在class中的用法**

         // 通过this.props就能访问当前页面的路由信息

         // 如: this.props.history; this.props.match;

// 注意: super(props)确保this.props的存在

    constructor(props){

        super(props);

        this.state = {

            email:'',

            password:'',

        };

}

}

### route高级路由 – 在子级组件中创建路由

import React from 'react';

import "./shopage.styles.scss";

import { Route } from 'react-router-dom';

import CollectionOverView  from '../../components/collection-overview/collection-overview.component';

import CollectionPage from '../collectionpage/collectionpage.component';

const ShopPage = ({ match }) => {

**// 高级路由**

        // 0. 子级组件中使用路由,就叫高级路由，😀

        // 1. 注意事项:

            // a) 在主路由中去除exact(当前主路由位置为APP.js)

                // <Route path='/shop' component={ShopPage} />

            // b) 在当前组件中路由要有exact,match.path为当前url位置

                // 0. 当前页面(必备): <Route exact path={`${match.path}`} component={CollectionOverView} />

                // 1. 子页面: <Route path={`${match.path}/:collectionId`} component={CollectionPage} />

        // 2. 通常情况下是由match配置动态路由

    return (

        <div className="shop-page">

            <Route exact path={`${match.path}`} component={CollectionOverView} />

            <Route path={`${match.path}/:collectionId`} component={CollectionPage} />

        </div>

    );

}

export default ShopPage;

## // firebase创建用户函数

handleSubmit = async cur => {

        cur.preventDefault();

        const { displayName, email, password, confirmPassword } = this.state;

        if( password != confirmPassword ){

            alert('二者密码不相同,请重新输入');

            return;

        }

        try{

**// firebase创建用户函数**

                // 0. auth.createUserWithEmailAndPassword( email, password )创建用户

                // 1. 成功: 则返回创建用户信息，并在firbase用户管理出现账户信息，不会在数据库中保留

                // 2. 失败: 则返回失败原因

                // 3. 注意: 要在Firebase控制台开启'允许用户使用电子邮件和密码注册'

            let createUser = await auth.createUserWithEmailAndPassword( email, password );

// 在数据库中创建用户信息

            await createUserProfileDocument( createUser.user, {displayName} ); // 吐血~这个createUser,应该返回createUser.user!!!!无语!!!!

            alert('注册成功');

            // 跳转到主页

            this.props.history.push('/');

        }

        catch( err ){

**// 在控制台显示红色错误信息**

            console.error(err);

            alert( err.message );

        }

// 注册完成state要初始化

        this.setState({

            email: '',

            password: '',

            displayName: '',

            confirmPassword: '',

        });

    }

## // firebase登陆用户函数

import { signInWithGoogle, auth } from "../../firebase/firebase.config";

handleSubmit = async cur => {

        cur.preventDefault();

        try{

**// firebase登陆用户函数**

                 // 0. auth.signInWithEmailAndPassword(email,password)

            await auth.signInWithEmailAndPassword( this.state.email, this.state.password );

            this.setState( { email:'', password:'' } ); // 登陆完,要初始化

            //跳转到主页

            await this.test;

            this.props.history.push('/');

        }

        catch(err){

            alert(err.message);

        }

    }

## // React-解构法自定义标签对象数据传输

### 传入方

**// React-解构法自定义标签对象数据传输**

// 0. 解构法传递，对象属性. 并具有自动识别，未传输参数来进行传输。

// 1. 接收方: 依然保持对应的对象名称来接受

注意假设: this.state.section = [

{title:xxx,imaUrl:xxx,size:xxx,id:xxx,linkUrl:xxx} ,

{title:xxx,imaUrl:xxx,size:xxx,id:xxx,linkUrl:xxx} ,

];

this.state.sections.map( ( { id, ...otherData } ) => (

                        <MenuItem key={id} {...otherData} ></MenuItem>

                     ) )

### 接受方

**注意：接收方变量名称，要于对象中属性名称一致**

const MenuItem = ({ title,imageUrl,size,linkUrl,history,match }) => (

  xxxxxxxx…..

);

export default withRouter(MenuItem);

# 2019.12.9

## React黄金三原则 – 领悟到了( 第二条 )

**目的：**是让代码更加的高效简约

原则:

1. Decide on Components – 解析: 打造可重用组件，决定什么情况使用组件
2. Decide the State and where it lives – 解析: 因为this.state的改变会影响React决定‘渲染组件’，所以this.state的位置要有讲究。( 实列可参考 – 2019.12.3注册/登陆页面,组件代码 )
3. What changes when state changes

# 2019.12.10

## // 在Switch标签外不受路由影响 - 无论页面如何变化,组件依然显示存在

**// 在Switch标签外不受路由影响 - 无论页面如何变化,组件依然显示存在**

*// 0.**把导航栏放在switch之外，这样导航栏将一直存在。不会受路由的控制*

*// 1. 这是一个非常非常重要的功能 - 不受页面刷新影响方法*

function App() {

  return (

    <div className="App">

      <Header /> // 把导航栏放在switch之外，这样导航栏将一直存在。不会受路由的控制

      <Switch>

        <Route exact path='/' component={HomePage} />

        <Route exact path='/shop' component={ShopPage} />

        <Route path='/shop/:proName' component={TestPage} />

        <Route path='/sign' component={SignPage} />

      </Switch>

    </div>

  );

}

## // React传输函数给组件

**传送方:**

<FormInput

    label="邮箱"

    handleChange={this.handleChange}

    type="email"

    name="email"

    value={this.state.email} required

/>

**接受方:**

**// React传输函数给组件**

*// 0. 注意：react是可以传输，函数的。比如现在的handleChange函数*

const FormInput = ({ label, handleChange, ...otherProps }) => {

    return(

        <div className="group">

            <input

            onChange={handleChange}

            className="form-input"

            {...otherProps}

            />

        </div>

    );

}

## // React-React导入svg文件

**// React-React导入svg文件**

*// 0. import { ReactComponent as Logo } from "../../assets/crown.svg";*

*// 1. 因为文件为jsx文件格式，所以需要借用 ReactComponent 这是规则，*

*// 2. { ReactComponent as Logo }修改名称为logo*

*// 3. 这样就可以使用自定义标签来代表svg图像<Logo />*

*// 4. SVG的优势: svg图标，为矢量图，并且很小*

import { ReactComponent as Logo } from "../../assets/crown.svg"; // 导入svg

const Header = () => {

    return(

           <Logo /> // 使用svg图像

    );

}

## // 解构法,面对标签时创建赋值变量

## // 变量值变为对象的键值

## // 感悟: 感觉REACT组件,就像是一个一个函数一样

## REACT处理FORM表单( 实战示例 )

### // React处理from提交数据

### // React处理input改变数据

import React from "react";

import "./sign-in.style.scss";

import FormInput from "../form-input/form-input.component";

import CustomButton from "../custom-button/custom-button.component";

class SignIn extends React.Component {

    constructor(){

        super();

        this.state = {

            email:'',

            password:'',

        }

    }

**// React处理from提交数据**

*// 0. form准备: <form onSubmit={this.handleSubmit}></form>*

*// 1. 注意: cur.preventDefault()函数的本质为,阻止默认事件发生,好由React来处理form数据*

*// 2. 函数准备: 如下 - cur为接受的form数据*

    handleSubmit = cur => {

        cur.preventDefault();

        this.setState( { email:'', password:'' } );

    }

**// React处理input改变数据**

*// 0. React中,input必须要有onChange={xxx}来处理表单数据,否则会报错*

*// 1. input准备: <input type="email" onChange={this.handleChange} />*

*// 2. 函数准备: 如下 - cur为接受的input数据*

    handleChange = cur => {

**// 解构法,面对标签时创建赋值变量**

*// 0. 因为其js"抓取的标签.target"本质也为对象*

*// 1. 所以可以使用解构法创建变量, 但注意名字要一致*

        const { value, name } = cur.target; // 相当于 cosnt value = cur.target.vaule, name = cur.target.name;

**// 变量值变为对象的键值**

*// 0. this.setState( { [name]: value } );*

*// 1. 其中[name]的意思为，将name的变量值作为键值,方便键值动态变化。*

*// 2. 比如name="password"那么相当于this.setState( password: value );*

        this.setState( {[name]: value} );

    }

**// 感悟: 感觉React组件,就像是一个一个函数一样**

    render(){

        return(

            <div className="sign-in">

                <h2>

                    登陆用户

                </h2>

                <span>

                    请使用邮箱和密码登陆

                </span>

                <form onSubmit={this.handleSubmit}>

                    <FormInput

                        label="邮箱"

                        handleChange={this.handleChange}

                        type="email"

                        name="email"

                        value={this.state.email} required

                    />

                    <FormInput

                        label="密码"

                        handleChange={this.handleChange}

                        type="password"

                        name="password"

                        value={this.state.password} required

                    />

                    <CustomButton type="submit" >

                        登陆

                    </CustomButton>

                </form>

            </div>

        );

    }

}

export default SignIn;

## // 解构变量属性读取

const FormInput = ({ label, handleChange, ...otherProps }) => {

    return(

**// 解构变量属性读取**

*// 0. '...otherProps' 当为解构的对象时*

*// 1. otherProps.xxx可以直接读取其中的属性,当然名称要一致*

                    <label className={ `${ otherProps.value.length ? 'shrink' : null } form-input-label` } >

                        {label}

                    </label>

    );

}

export default FormInput

## // 对象解构法 - 创建对象属性变量并赋值

**// 对象解构法 - 创建对象属性变量并赋值**

*// 0. const { collectionShop } = this.state; 相当于 const collectionShop = this.state.collectionShop;*

*// 1. 注意属性名称与变量名称一致，以及注意大括号。*

        const { collectionShop } = this.state;

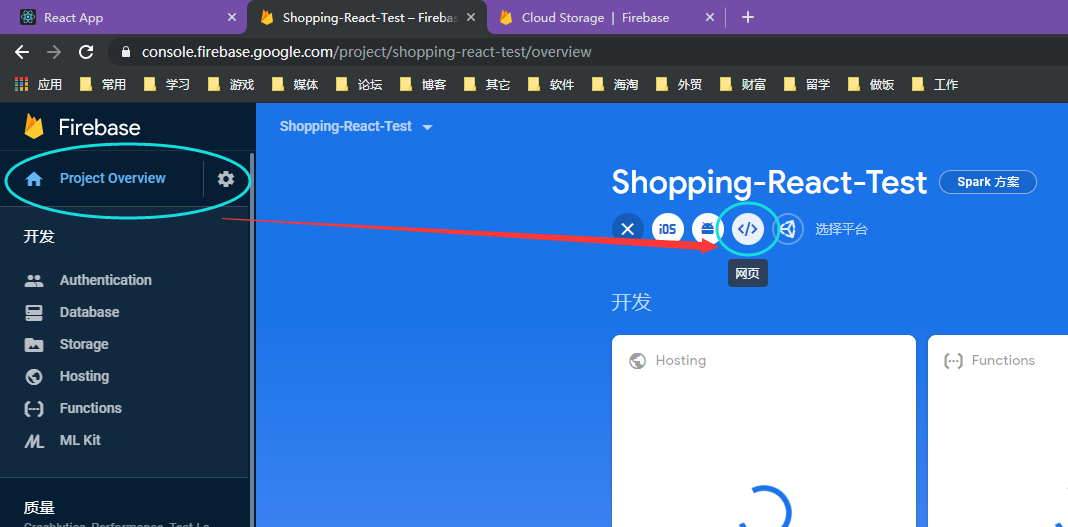
# 2019.12.10( Firebase )

## 使用Firebase创建谷歌用户验证

### 创建项目

控制台创建项目： <https://console.firebase.google.com/>

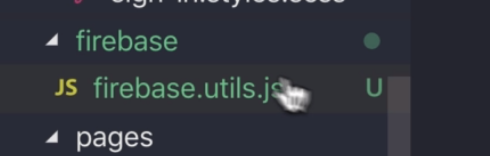








### 安装firebase库

1. **保留上方选中的配置信息**
2. **安装：**yarn add firebase
3. 
4. Config密钥配置信息，如上放图，下方为firebase.utils.js配置信息

## Firebase配置Google登陆( 详细过程 )

**\* Firebase谷歌登陆配置**

 \*/

import firebase from 'firebase/app'; // 获取firebase/app我们只需要它就够用拉

import 'firebase/firestore'; // 获取应用函数库

import 'firebase/auth'; // 获取认证函数库

**// 认证签名信息( 上方图有说明 )**

const config = {

    apiKey: "AIzaSyB2ypDqAys\_PEcg2kSW4bZwAJh0PxnjNtw",

    authDomain: "shopping-react-test.firebaseapp.com",

    databaseURL: "https://shopping-react-test.firebaseio.com",

    projectId: "shopping-react-test",

    storageBucket: "shopping-react-test.appspot.com",

    messagingSenderId: "448778125411",

    appId: "1:448778125411:web:887aaf1acf4e8723c23856",

    measurementId: "G-DQ5KCRGZB0"

}

firebase.initializeApp(config); // 提交认证前面信息

export const auth = firebase.auth(); // 认证函数库

export const firestore = firebase.firestore();

const provider = new firebase.auth.GoogleAuthProvider(); // 初始化谷歌登陆

provider.setCustomParameters({ prompt: 'select\_account' }); // 开启弹窗选择账号

export const signInWithGoogle = () => auth.signInWithPopup(provider); // 验证用户

export default firebase; // 默认导出

/\*

**//****Firebase配置Google登陆,核心函数总结：**

**// Firebase配置:**

        // 0. 准备好需要用的库:

            import firebase from 'firebase/app'; // 获取firebase/app我们只需要它就够用拉

            import 'firebase/firestore'; // 获取应用函数库

            import 'firebase/auth'; // 获取认证函数库

        // 1. 得到firbase的认证:

            // a) firebase.initializeApp(config);

            // b) config信息在firbase中获得

        // 2. 第三方登陆认证库: firbase.auth()

        // 3. firbase程序包/扩展功能小背包: firebase.firestore()

**// 谷歌登陆配置:**

        // 0. 初始化谷歌登陆: const provider = new firebase.auth.GoogleAuthProvider();

        // 1. 开启弹窗登陆: provider.setCustomParameters({ prompt: 'select\_account' });

        // 2. 此函数开启谷歌登陆窗口: export const signInWithGoogle = () => auth.signInWithPopup(provider);

            // a) 开启谷歌登陆窗口: signInWithGoogle();

            // b) 获取登陆者信息( 返回一个对象,**别忘记防内存泄漏** ): auth.onAuthStateChanged( user => console.log(user) );

            // c) 退出登陆: auth.signOut()

\*/

## 登陆信息防止内存泄漏方法

## class中赋值骚操作,class变量也可以以函数执行的方式初始化

import { auth } from './firebase/firebase.config';

class App extends React.Component {

  constructor(){

    super();

    this.state = {

      currentUser: null,

    }

  }

  unsubscribeFromAuth = null; **// Google登陆验证防内存泄漏**

**// 获取登陆用户信息**

// 注意: 生命周期位置

  componentDidMount(){

    this.unsubscribeFromAuth = auth.onAuthStateChanged( user => {

      this.setState({ currentUser: user });

      console.log( user );

    } );

  }

**// 卸载组件时,退出登陆**

// 注意: 生命周期位置

  componentWillUnmount(){

    this.unsubscribeFromAuth();**//****class中赋值骚操作,相当于this.unsubscribeFromAuth = null;**

  }

}

## JS-oninput实时监听表单数据变化

Html:

<input id="test1" type="text" placeholder="请输入文档主题" >

JS:

Document.queryselector(‘#test1’).oninput = props => {xxx};

## JS-获取指定标签宽度-用JS实现页面跳转

document.body.clientWidth > 1200 ? location.href = "http://www.baidu.com" : null;

xxx.clientWidth - 获取指定标签宽度

location.href = <http://www.baidu.com> – 实现跳转页面

## 使用firebase添加数据, 以及获取数据( 来源于电商实战项目firebase.config.js )

### // firestore测试ref对象的数据获取

**// firestore测试ref对象的数据获取**

        const collectionRef = await firestore.collection(`/users`);

        const collectionSnapshot = await collectionRef.get();

        console.log( 'test',collectionSnapshot.docs.map( doc => doc.data() ) ); // 注意.docs为数组,不能直接data()处理, 需迭代出内容

### // 构建: firebase添加数据,使用firestore.batch批处理传送

### // 获取商品数据从firebase

### // reduce骚操作构建对象类型数据

### encodeURI( 字符串 ); 将字符串转换为URL编码, 方便浏览器处理

**// 构建: firebase添加数据,使用firestore.batch批处理传送**

    // a) 只所以使用batch来传输数据, 是为了保证传输数据的完整性, 因数据传输时会分批传输, 若有一方失败, 则否定全部数据传输失败, 保证传输成功的数据完整性, 可控性.

    // b) 获取batch函数: const batch = firestore.batch()

    // c) 设定待传送的数据: batch.set( 空文档, 传送数据 );

    // d) 上传数据: batch.commit()

    // f) 注意: 异步上传数据

export const addCollectionAndDocuments = async ( collectionKey, objectsToAdd ) => {

    const collectionRef = firestore.collection( collectionKey ); // 先collection创建新集合

    const batch = firestore.batch();

    objectsToAdd.forEach( obj => {

        const newDocRef = collectionRef.doc(); //  在doc 创建新文档, 新对象数据

        batch.set( newDocRef, obj ); // batch.set准备发射, 准备需要传送的数据

    } );

    return await batch.commit(); // batch.commit 发射, 数据提交

};

**// 获取商品数据从firebase**

    // 0. 在这里加工处理商品数据

    // 1. encodeURI( 字符串 ); 将字符串转换为URL编码, 方便浏览器处理

export const convertCollectionsSnapshotToMap = collections => {

    const transfromedCollection = collections.docs.map( doc => {

        const { title, items } = doc.data();

        return {

            routeName: encodeURI( title.toLowerCase() ),

            id: doc.id,

            title,

            items,

        }

    } )

**// reduce骚操作构建对象类型数据**

        // 0. 核心: { [key]: value }: 将变量变为对象的键值，变量转对象键值

        // 1. reduce()不仅可以计算统计数字,也可以用于构建数据

    return transfromedCollection.reduce( ( total, cur )=>{

        total[ cur.title.toLowerCase() ] = cur;

        return total;

    },{} );

}

## React组件使用onSnapshot动态监听后端数据

import { firestore,convertCollectionsSnapshotToMap } from '../../firebase/firebase.config';

import { connect } from 'react-redux';

import { updateCollections } from '../../redux/shop/shop.action';

    componentDidMount(){

        const { updateCollections } = this.props;

**// 从firebase获取动态数据**

            // 0. onSnapshot() 监听ref对象数据变化

        const collectionsRef = firestore.collection('collections');

        collectionsRef.onSnapshot( async snapshot => {

            updateCollections(convertCollectionsSnapshotToMap( snapshot ));

        } )

    }

# 2019.12.12

## 关于firestore专有名词:

// a) Ref对象: 由.doc()和.collection()获取的对象数据

// b) 快照对象: 由.get()获取的对象数据

// c) 存储对象: 从快照对象中获取数据.data()来获取

## // Firebase获取Ref对象方式 - 请求数据库内容

**// Firebase获取Ref对象方式 - firebase请求数据库内容**

    // 0. 普通方法:

    const test = await firestore.collection('users').doc(`/users/${user.uid}`).collection('boughtItems').doc(`XipnGtHbDjWGIfngyj5F`);

    // 1. 便捷式方法:

    // a) 获取Ref对象合集:

    const test2 = await firestore.collection(`/users/aN6rU93VOpAlywi9p01N/boughtItems`);

    // b) 获取Ref对象文档:

const test3 = await firestore.doc(`/users/aN6rU93VOpAlywi9p01N/boughtItems/XipnGtHbDjWGIfngyj5F`);

## 将谷歌登陆信息存储道Firebase数据库

### 如果谷歌登陆,数据库无用户信息,则近行创建用户信息,保存道users合集中

### // firestore.doc('数据库路径') 获取指定路径文档Ref对象

### // FIRESTORE.collection('数据库路径') 获取合集Ref对象

### // JS获取当前时间

**// firestore.doc('数据库路径')** **获取指定路径文档Ref对象**

**// firestore.collection('数据库路径')** **获取合集Ref对象**

**// 基于ref对象才能使用的函数: get(),set(),update(),delete(),add()**

            // 0. 基于.doc()才能使用:

                // a) get()获取文档快照对象:

                    // exists: 如果目标在数据库中则为true否则为false,常用于验证用户是否在数据库中

                // b) set()编辑数据:

                    // firstore.doc('xxx').set({ xxx:yyy, xxx2:yyy2 });

                // c) update()更新数据

                // d) delete()删除数据

                // e) .onSnapshot( props => {} )监听文档,文档快照发生变化时方便更新数据。

            // 1. 基于.collection()才能使用

                // a) get() 获取合集快照

                // b) add() 在合集中添加内容,并且自动创建文档ID

                    // firestore.collection('xxx').add({ xxx:yyy });

export const createUserProfileDocument = async ( userAuth, otherData ) =>{

    try{

        if( !userAuth ) return; // 如果没有发生登陆,则什么都不干

        const userRef = await firestore.doc(`/users/${userAuth.uid}`);

        const snapShot = await userRef.get();

        if( !snapShot.exists ){ // exists验证是否已在数据库中

            const { displayName, email, photoURL  } = userAuth;

            const createTime = new Date(); **// 获取当前时间**

            try{

                await userRef.set({

                    displayName,

                    email,

                    photoURL,

                    createTime,

                    ...otherData,

                });

            }

            catch(err){

                console.log( err );

            }

        }

        return await userRef; // 传递获取的用户信息方便进行其它操作

    }

    catch(err){

        console.log(err);

    }

};

### 当前用户数据存储在本地

### firebase-onSnapshot()监听文档对象(快照对象)方便数据更新

### React生命周期组件内,可以使用await等待异步数据

### firestore.doc(‘xxx’).data()获取快照对象存储在数据库中的数据

componentDidMount(){

    this.unsubscribeFromAuth = auth.onAuthStateChanged( async user => {

      // 如果用户登陆

      if( user ){

**// firebase-onSnapshot()监听文档对象(快照对象)方便数据更新( 等待笔记 )**

          // 0. onSnapshot(props=>{xx}) 用于监听快照对象,如果数据发生变化,方便数据变化时实时更新

          // 1. props用于传递快照对象的数据,于监听快照对象无疑

        const userRef = await createUserProfileDocument( user );**// React生命周期组件内,可以使用await等待异步数据**

        userRef.onSnapshot( props => { // 注意userRef为Ref对象

          this.setState({

            id: props.id,

            ...props.data() **// 解构当前对象内容;** **// firestore.doc(‘xxx’).data()获取快照对象存储在数据库中的数据**

          });

        } );

      }

      // 注意:此乃谷歌原始配置信息

      this.setState({ currentUser: user });

    } );

  }

# 2019.12.15( Redux )

## Redux核心原则

0. single source of truth( )

1. state is read only （对象不会被修改）

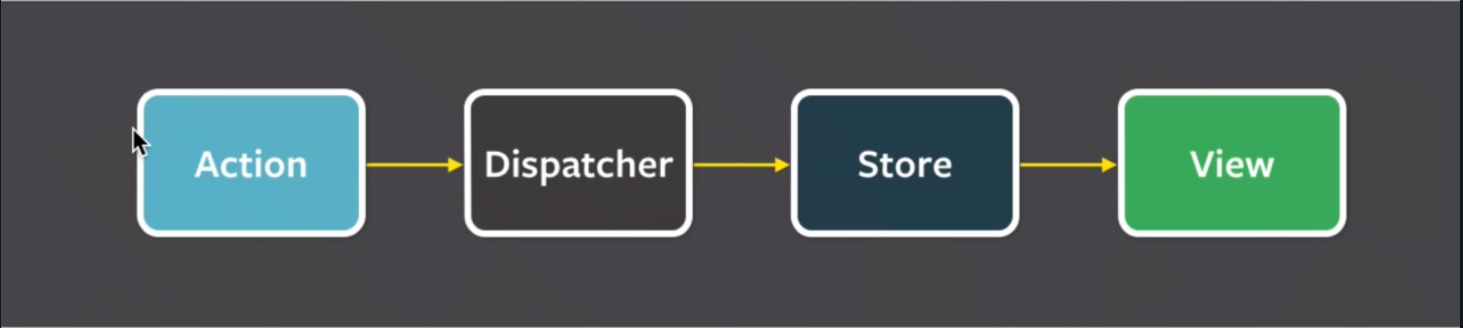
2. changes using pure functions （只使用，函数进行修改）



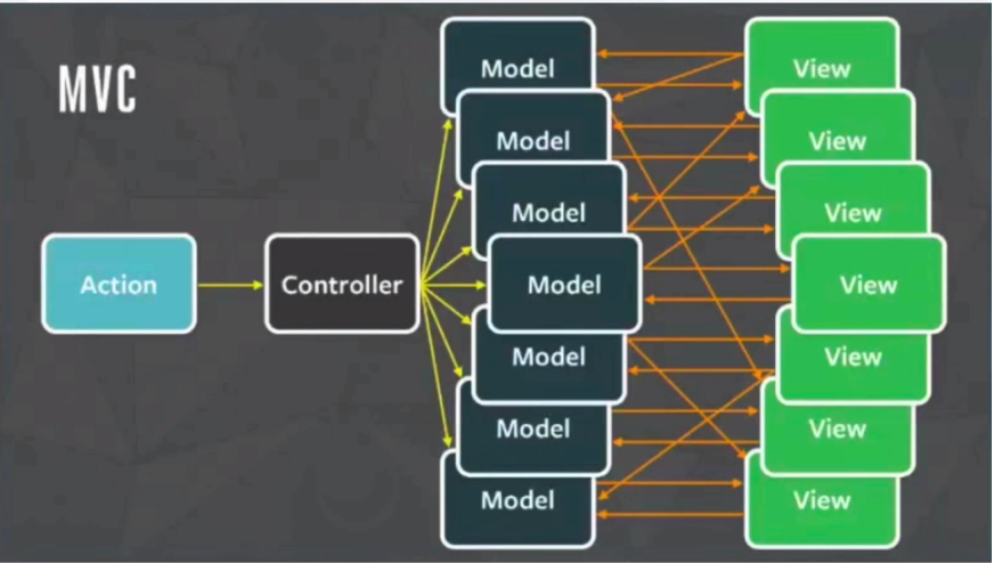
### 操控逻辑：

redux专用名词，以及操控逻辑：

1. **Action ：活动**
2. **dispatcher ：收发**
3. **store ：商店**
4. **view : 视图**

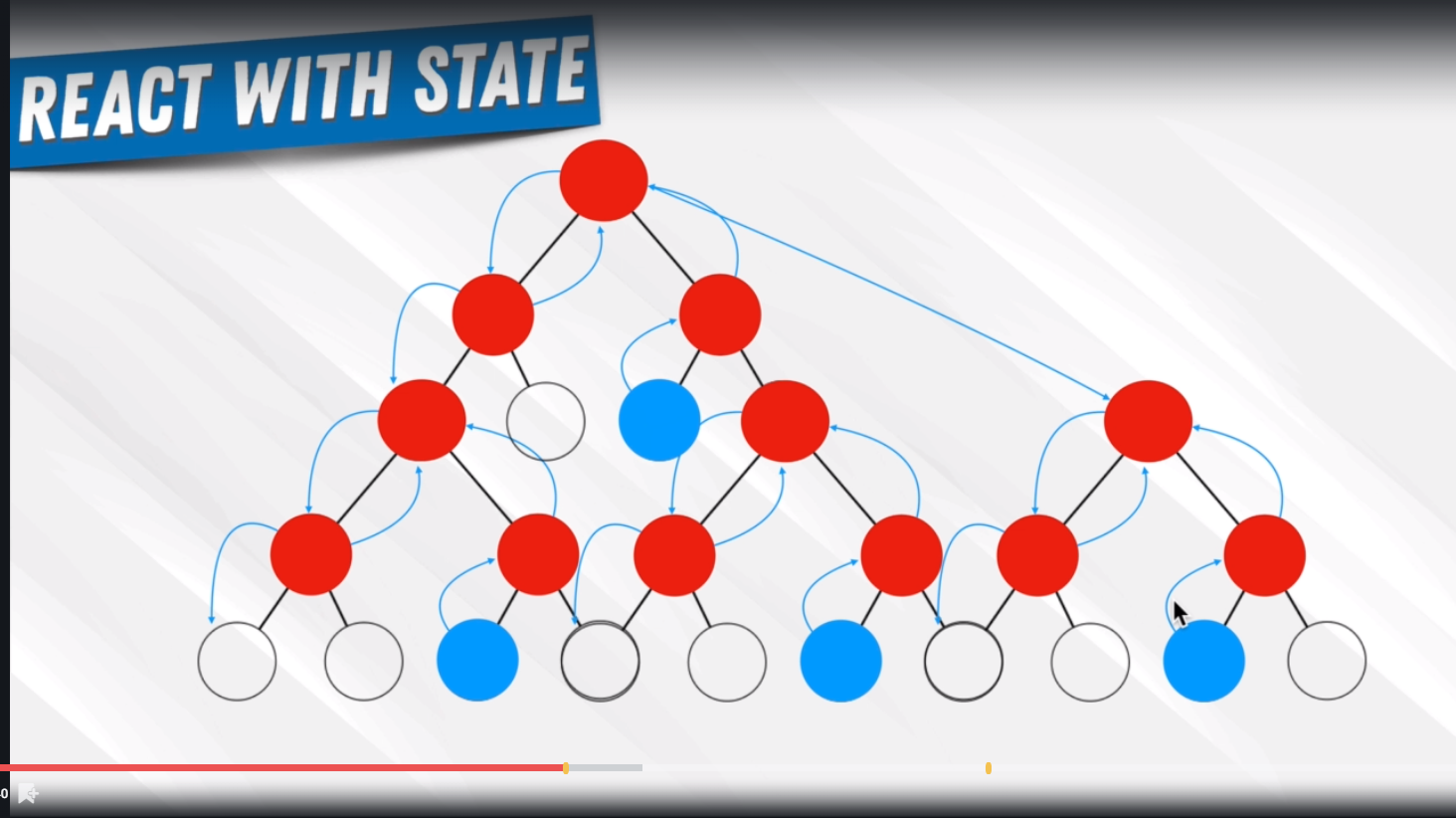


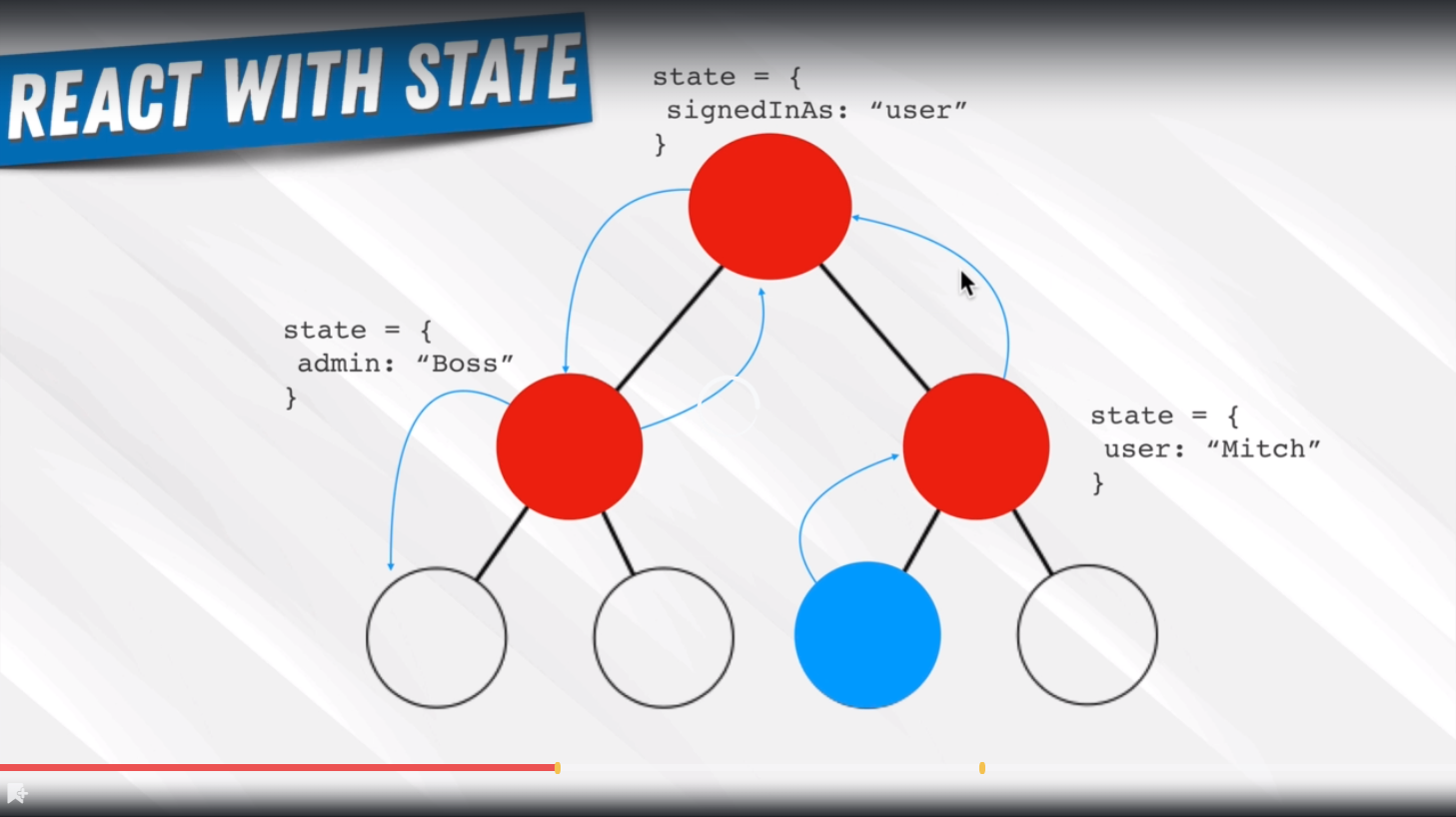
### redux 采用MVC模式，来处理交互数据



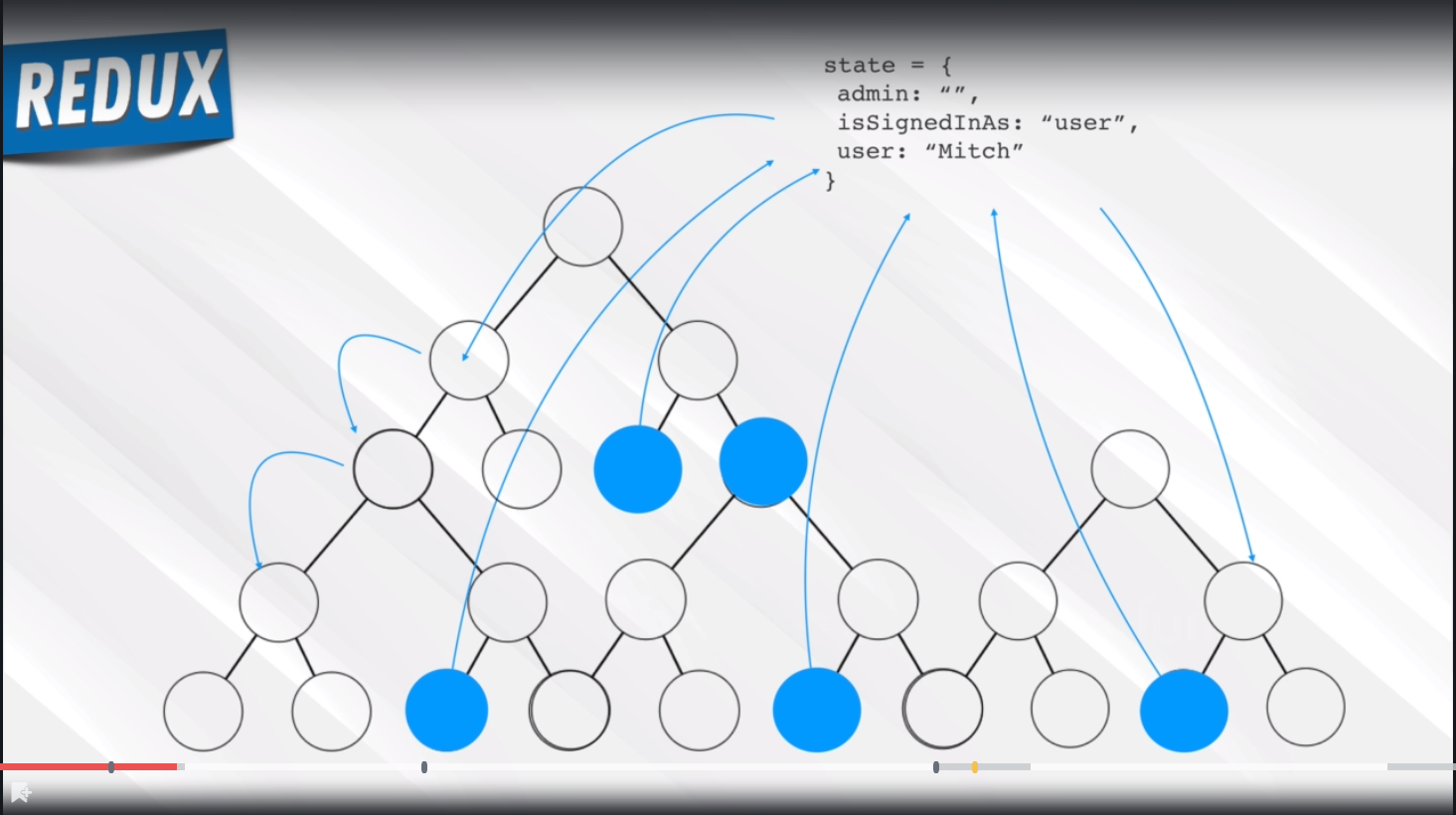
## Redux工作方式，以及与正常react工作方式区别

### 普通的react，state工作方式

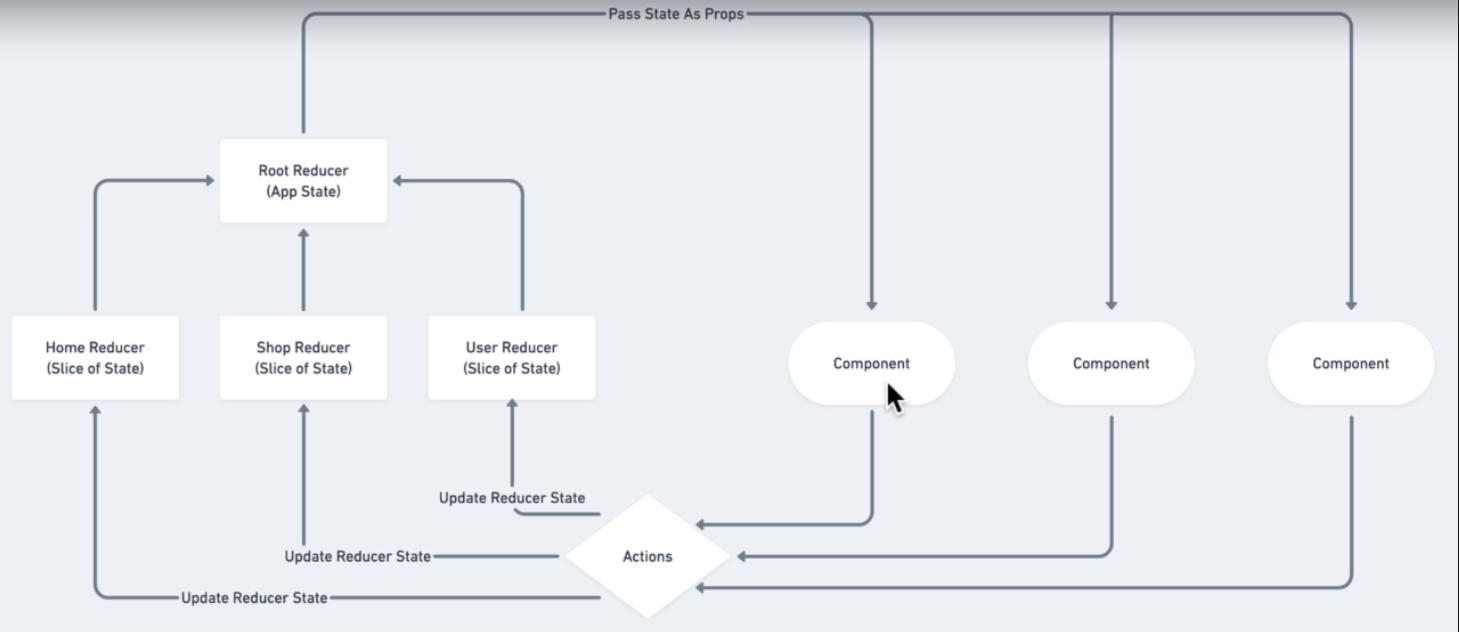




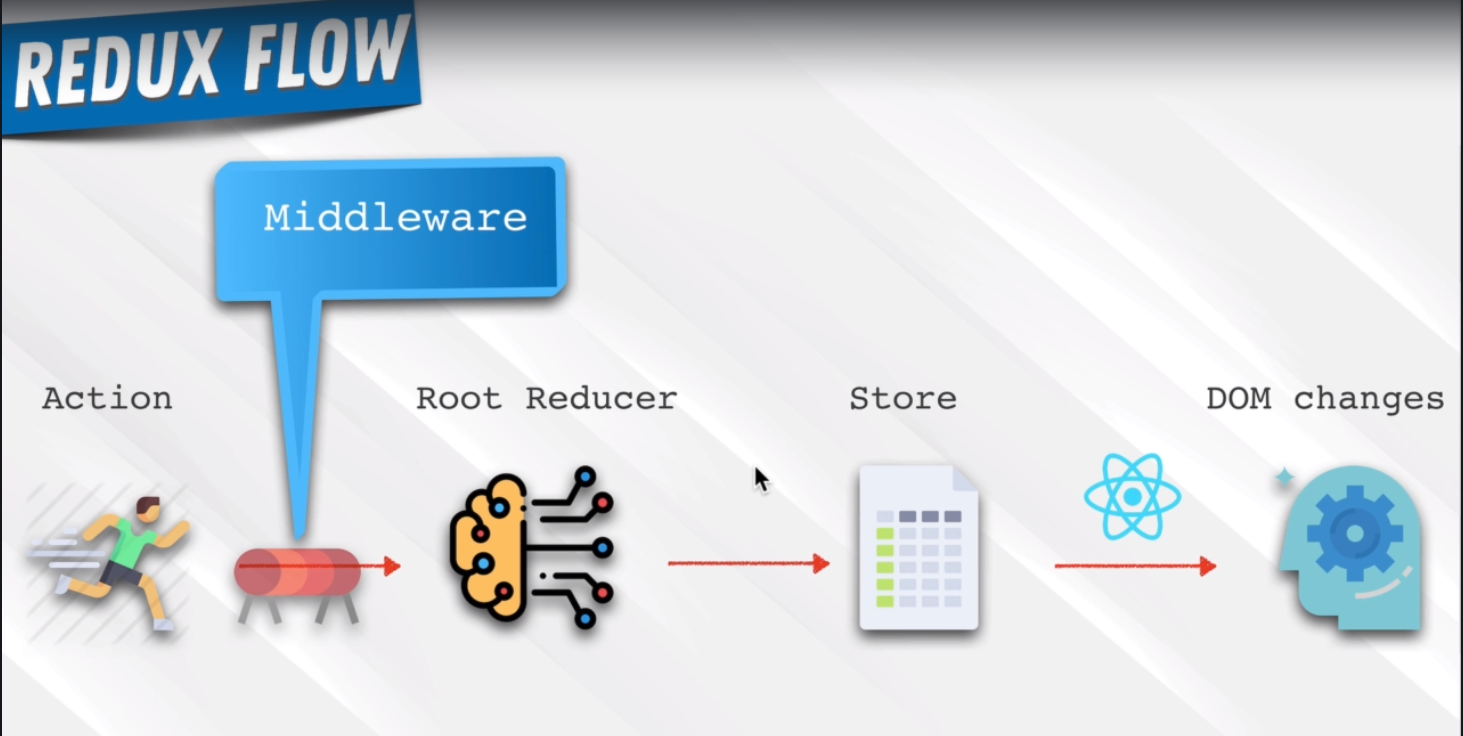
### Redux工作方式



## Redux具体工作流程

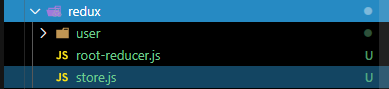
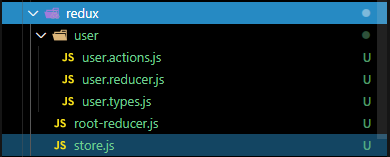


## Redux概念流程图



## 第一部分：Redux文件构建

### 文件结构



### redux文件构建流程

1. 目录：root-reducer.js – 用于统计记录redux组件信息
2. 中间件: store.js – 构建redux商店，方便其它函数使用
3. Redux组件：分3种类型文件
   1. Xxx.reducer.js: 配合actions选择处理方法
   2. Xxx.actions.js: 用于改变redux数据参数
   3. Xxx.types.js: 用于保存type字符串，方便reducer/actions文件调用，也方便字符串统一管理

#### 目录：root-reducer.js – 用于统计记录redux组件信息

/\*\*

**\* 1. reducer统计目录: root-reducer.js**

 \*/

import { combineReducers } from 'redux';

import userReducer from './user/user.reducer'; // 引如redux组件

export default combineReducers({

    user: userReducer,

});

#### 中间件: store.js – 构建redux商店，方便其它函数使用

/\*\*

**\* 2. 构建store.js和中间件**

 \*      a) 创建中间件: createStore( rootReducer, applyMiddleware(...middlewares) );

 \*      b) 解析: 创建中间件时,使用createStore( rootReducer目录，应用中间件函数 );

 \*      c) logger为redux记录器，方便开发查看数据变化

 \*/

import { createStore, applyMiddleware } from 'redux';

import logger from 'redux-logger';

import rootReducer from './root-reducer'; // 引入redux目录

**// 中间件: 因为中间件后期要添加很多,所以需要解构符来配合**

**// process.end.NODE\_ENV可以确定当前项目是否为生产环境**

    // 0. process.env.NODE\_ENV === 'development': 为开发环境

    // 1. process.env.NODE\_ENV === 'production': 为生产环境

const middlewares = [];

if( process.env.NODE\_ENV === 'development' ){

    middlewares.push(logger);

}

const store = createStore( rootReducer, applyMiddleware(...middlewares) );

export default store;

#### Redux组件：分3种类型文件

##### Xxx.reducer.js:

/\*\*

**\* 0. 构建reducer默认写法**

 \*/

import { UserActionTypes } from './user.types';

**// 初始化state**

const INITIAL\_STATE = {

    currentUser: null

}

**// 接受处理action值改变state**

    // 0. 根据action.type选择处理方法,方法执行使用action.payload,处理后返回改进的state

        // a) action.type验证类型

        // b) action.payload传输的数据,可以是对象,函数,数字,字符串,等数据类型。

    // 1. 都不符合则返回默认state

const userReducer = ( state = INITIAL\_STATE, action ) => {

    switch( action.type ){

        case UserActionTypes.SET\_CURRENT\_USER:

            return{

                ...state,

                currentUser: action.payload,

            }

        default:

            return state;

    }

}

export default userReducer;

##### Xxx.actions.js

**// 4. 构建actions写法**

**// a) type和payload必须要有的**

**// 0. type：放置type字符串**

**// 1. payload：action.payload传输的数据,可以是对象,函数,数字,字符串,等数据类型。在没有传输值时payload参数可以忽略不写**

import { UserActionTypes } from './user.types';

export const setCurrentUser = user => ({

    type: UserActionTypes.SET\_CURRENT\_USER,

    payload: user,

});

##### Xxx.types.js

**// 保存type字符串集合，方便调用修改**

export const UserActionTypes = {

    SET\_CURRENT\_USER: 'SET\_CURRENT\_USER',

}

##### xx.utility.js

**//****xx.utility.js专门写入,扩展函数,多用于加工处理reducer传输的数据并返回加工后的结果**

export const  addItemToCart = ( cartItems, additem ) => {

   const door = cartItems.find( cur => cur.id === additem.id );

// 如果购物车已存在物品则数量+1

   if( door ){

        return cartItems.map(

            cur => cur.id === additem.id ? { ...cur, quantity: cur.quantity + 1 } : cur

        );

   }

**// 解构的方法增加对象元素/解构的方法增加数组元素**

// 如果购物车无物品则创建物品,并设定数量为1

   return [ ...cartItems, { ...additem, quantity: 1 } ];

}

**// xx.utility.js使用方式，具体实列请看2019.12.3/shopping-react/redux/cart/内**

import CartActionsType from  "./cart.types";

import { addItemToCart } from "./cart.utility";

const INITIAL\_STATE = {

    hidden: false,

    cartItems: [],

}

const cartReducer = ( state = INITIAL\_STATE, action ) => {

    switch( action.type ){

        case CartActionsType.TOGGLE\_CART\_HIDDEN:

            return {

                ...state,

                hidden: !state.hidden,

            }

        case CartActionsType.ADD\_CART\_ITEMS:

            return {

                ...state,

                cartItems: addItemToCart(state.cartItems, action.payload ),

            }

        default:

            return state;

    }

}

export default cartReducer;

##### xxx.selectors.js

请跳转到笔记下方2019.12.27中对reselect框架的使用

## 第二部分：Redux组件中使用方式

### 初始化redux

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter } from 'react-router-dom';

/\*\*

**\* redux获取组件访问权**

 \*/

import { Provider } from 'react-redux';

import store from './redux/store';

import './index.css';

import App from './App';

**// Redux初始化标签必备,获取组件访问权**

ReactDOM.render(

**// React-Redux: Provider标签一定要传递store,否则容易报错**

    <Provider store={store} >

        <BrowserRouter>

            <App />

        </BrowserRouter>

    </Provider>

    ,

    document.getElementById('root')

);

### 核心：react组件交互redux数据

**// connect函数使react组件可以访问redux存储**

import { connect } from 'react-redux';

**// React-Redux: connect()函数,用于交互redux数据**

    // 0. connect( mapStateToProps, mapDispatchToProps );

**// 1. mapStateToProps: 用于获取redux数据,默认值为null**

        // a) store更新的任何时候mapStateToProps都将被调用,换句话说保持数据更新

        // b) 他返回必须是一个对象类型

        // c) 使用方式：

            // 0. class获取传输的Redux数据:

                    import { connect } from 'react-redux';

                    class xxx {

                        this.props.currentUser; // 通过this.props.xxxx获取

                    }

                    const mapStateToProps = state => ({

                        currentUser: state.user.currentUser,

                    });

                    export default connect(mapStateToProps)(App);

            // 1.函数传输的Redux数据

                    import { connect } from 'react-redux';

                    const Header = ({ currentUser }) => {

                        currentUser; // 需要函数接受一下对应变量名称，即可正常使用

                    }

                    const mapStateToProps = state => ({

                        currentUser: state.user.currentUser,

                    });

                    export default connect(mapStateToProps)(Header);

**// 2. mapDispatchToProps: 用于传输参数并执行Actions从而改变redux数据**

        // a) 调度redux对应组件actions函数

        // b) dispatch( actions函数 )，让redux知道我们要执行actions函数,用于改变数据

        // c) 使用方式: - 与mapStateToProps()相同，函数直接传入actions名称，对象通过this.props传入

                import { connect } from 'react-redux';

                import { setCurrentUser } from './redux/user/user.actions';

c

                const mapDispatchToProps = dispatch => ({

                    setCurrentUser: user => dispatch( setCurrentUser(user) ),

                });

                export default connect(null, mapDispatchToProps)(App);

### // redux的dispatch()可以直接传递使用-便捷式使用dispatch

### // withRouter与redux的connect嵌套并不冲突

import React from 'react';

import "./cart-dropdown.styles.scss";

import { createStructuredSelector } from 'reselect';

import { selectCartItems } from '../../redux/cart/cart.selectors';

import CustomButton from '../custom-button/custom-button.component';

import CartItem from '../cart-item/cart-item.component';

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

import {connect} from 'react-redux';

import { toggleCartHidden } from '../../redux/cart/cart.actions';

**// redux的dispatch()可以直接传递使用-便捷式使用dispatch**

    // 0. redux的dispatch()可以直接传递使用

    // 1. 并且在只有mapStateToProps()的情况下

    // 2. dispatch()函数可以直接执行actions中的函数，非常方便

const CartDropdown = ({ cartItems, history, dispatch }) => {

    return (

        <div className="cart-dropdown">

            <div className="cart-items">

                {

                    cartItems.length ?

                    cartItems.map( cur => (<CartItem key={cur.id} item={cur} />) ) :

                    (<span className="cart-items-alt" >你购物车是空的!</span>)

                }

            </div>

            <CustomButton

            onClick={ ()=>{

                dispatch(toggleCartHidden()); **// 直接执行redux actions 函数**

                history.push('/checkout');

            } }

            selfCss={

                'cart-dropdown-btn'

            } >

                结算

            </CustomButton>

        </div>

    );

}

const mapStateToprops = createStructuredSelector({

    cartItems: selectCartItems,

})

**// withRouter与redux的connect嵌套并不冲突**

    // 0. 正常使用即可

export default withRouter(connect(mapStateToprops)(CartDropdown));

### // 关于路由render属性的功能

### // 关于路由Redirect的使用,重定向指定页面

{

**// 关于路由render属性的功能，决定渲染组件**

              // 0. 用法: <Route exact path render={()=> <自定义标签 />} >

              // 1. 有render属性则不能有component属性

              // 2. 必需要有exact属性

**// 关于路由Redirect的使用,重定向指定页面**

              // 0. 导入Redirect: import { Switch,Route,Link,Redirect } from 'react-router-dom';

              // 1. 使用方式<Redirect to='路由位置'>,直要被渲染将直接跳转指定路由位置

              // 2. 实列( 当用户登陆后,则将无法访问注册/登陆页面 ):

          }

          <Route exact path='/sign' render={ ()=> this.props.currentUser ? <Redirect to='/' /> : <SignPage />  } />

# 2019.12.16( codesandbox )

## Codesandbox 在线编辑器的使用

官网：<https://codesandbox.io/>

常用配置：

1. 开启VIM模式
   1. 在右上角图标处->设置->Editer->开启vim模式
2. 更换皮肤
   1. 按照vscood设置即可
3. 隐藏右侧预览
   1. Ctrl+shift+D
4. 切换窗口拆分方向
   1. Alt+Shift+0

# 2019.12.29( reselect )

## // 接受单个对象的，解构方法，真的骚

**// 接受单个对象的解构方法，真的骚**

const CartItem = ({ item: { imageUrl, name, price, quantity } }) => (

    <div className="cart-item">

        <img src={imageUrl} alt={name}/>

    </div>

);

**// 相当于**

const CartItem = (item) => (

    <div className="cart-item">

        <img src={item.imageUrl} alt={item.name}/>

    </div>

);

## 关于reselect用法 – 专门处理接受后的state数据( mapStateToProps()传递来的state )

## xxx.selectors.js文件构建，与redux配合提高react性能

安装：yarn add reselect

/\*\*\*

**\*****xxx.selectors.js文件构建，与redux配合提高react性能**

**注意：select只与mapStateToProps传递来的数据加工有关系，对于redux的action执行,无任何配置关联**

**0. 目的: 保证只有在redux数据改变时，才能重新渲染组件，提高react性能**

   1. 总体来说select分，输入/输出

   2. 构建流程：

        a) select分成二步，输入与输出

        b) 输入: 来选择引入来的数据:

            0. const selectXXX = state => stateXXX;

        c) 输出：数据流只要经过select输出，就能保证只有redux数据方式改变时对应的组件才能重新渲染，提高了react性能。

            0. export const xxx = createSelector(

                [selectXXX], // 导入数据

                xxx => xxx.yyy;

            );

            1. 输出处理核心:

            import {createSelector} from 'reselect';

            createSelector([xxx,yyy],xxx=>xxx.xxx);

    3. 使用流程:

        a) 普通方法使用select

            0. 实例代码

import { selectCartItemsCount } from '../../redux/cart/cart.selectors';

const mapStateToProps = state => ({

     itemCount: selectCartItemsCount(state),

});

        b) 便捷式方法使用select

            0. 核心：createStructuredSelector({ xx:yy })

            1. 实例代码

import { createStructuredSelector } from 'reselect';

import { selectCartItems } from '../../redux/cart/cart.selectors';

const mapStateToprops = createStructuredSelector({

     cartItems: selectCartItems,

})

 \*/

import { createSelector } from 'reselect';

**// select - 输入部分 - BGN**

const selectCart = state => state.cart;

**// select - 输出部分 - END**

**// 1. select单个数据处理输出使用方式**

export const selectCartItems = createSelector(

    [selectCart],

    cart => cart.cartItems,

);

export const selectCartHidden = createSelector(

    [selectCart],

    cart => cart.hidden,

);

export const selectCartItemsCount = createSelector(

    [selectCartItems],

    cartItems => cartItems.reduce( (total,cur)=>total+cur.quantity,0 ),

);

**// 2. select多个数据处理输出使用方式**

const selectCart = state => state.cart;

const selectUser = state => state.user;

export const selectCartItems = createSelector(

    [selectCart, selectUser],

    (cart, user) => user.currentUser,

);

## reSelect传入参数的函数构建方式

## 对象转数组

## 高效索引数据-直接索引对象数据最最高效的, 比filuter,find都要高效

import { createSelector } from 'reselect';

const selectShop = state => state.shop;

export const selectCollectionShop = createSelector(

    [selectShop],

    shop => shop.collectionShop,

);

export const selectCollectionShopArray = createSelector(

    [selectCollectionShop],

**//****对象转数组类型**

        // 0. 核心:

            // a) Object.keys( 对象数据 )返回数组对象的键值

                // 0. test = { a:123, b: 334 } --> Object.keys(test) --> [ 'a', 'b' ]

            // b) 键值数组.map( 创建新数组 )通过键值迭代索引对象内容,在通过map创建新数组

        // 1. 实例: collectionShop为对象数据

    collectionShop => Object.keys(collectionShop).map( e=>collectionShop[e] ) ,

);

**// 构建: reselect接受参数函数处理写法**

export const selectCollectionItem = collectionUrlProps => createSelector(

    [selectCollectionShop],

    collectionShop => collectionShop[ collectionUrlProps ]  **//****直接索引对象数据最最高效的, 比filuter,find都要高效: 原理https://www.kirupa.com/html5/hashtables\_vs\_arrays.htm**

);

### 使用: reselect传递参数函数处理写法( 注意与上方”构建: reselect接受参数函数处理写法”配套使用 )

**// 使用: reselect传递参数函数处理写法**

    // 0. 传入参数解析:

        // a) state: 为redux默认传入的对象数据

        // b) ownProps(任意名称): 为当前组件中的this.props对象数据

    // 1. select函数使用方式:

        // b) selectXXX( 自定义传入参数 )( Redux数据 )

const mapStateToProps = (state, ownProps) => ({

    collectionItem: selectCollectionItem(ownProps.match.params.collectionId)(state),

})

# 2020.1.3

## Currying： 函数多级嵌套用法

**// curring: 函数多级嵌套用法**

    // 创建:

    const curringFunction = (a) => (b) => (a\*b);

    // 使用方法一:

    curringFunction(5)(5) // 25

    // 使用方法二:

    const curringFunction5 = curringFunction(5);

    curringFunction5(5) // 25

# 2020.1.4

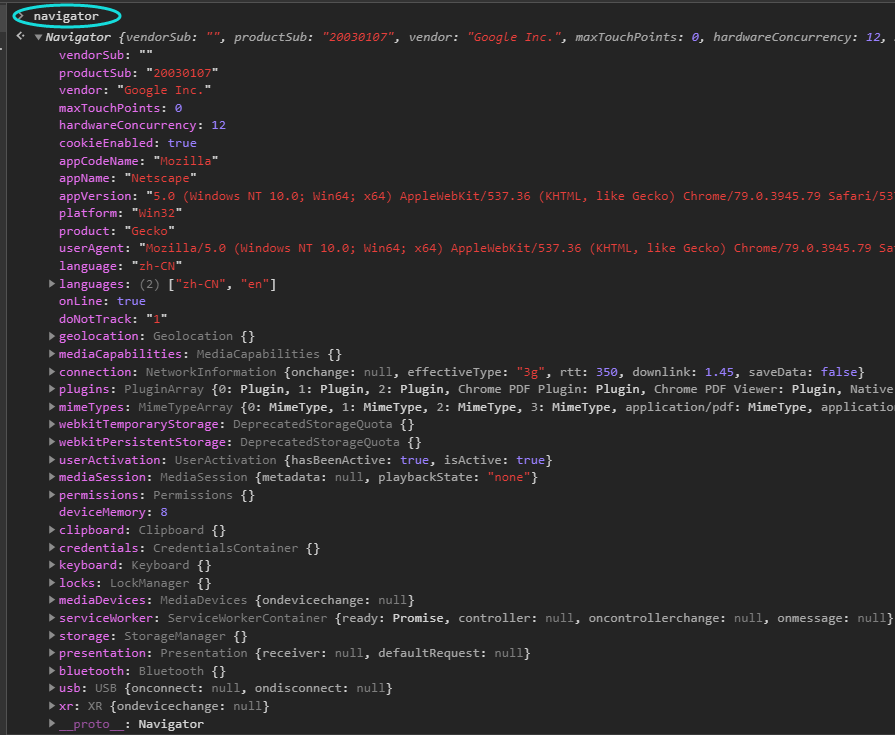
## JS本地存储的二个函数：

1. Window.localStorage: 关闭浏览器时清除
2. Window.sessionStorage: 关闭当前页面时清除
   1. 关于window.sessionStorage可以参考：<https://developer.mozilla.org/zh-CN/docs/Web/API/Window/sessionStorage>

# 2019.1.9( redux-persist )

## Js-获取访问站点的设备信息

浏览器默认对象navigator包含访问用户的信息

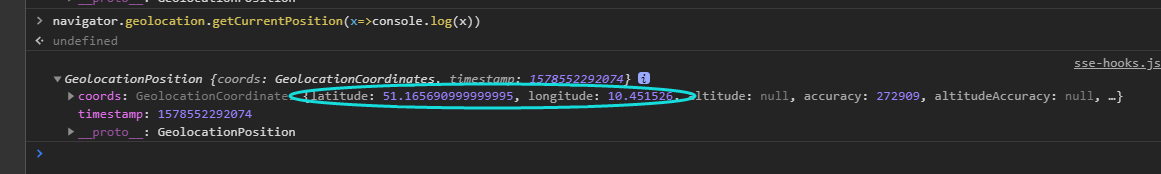


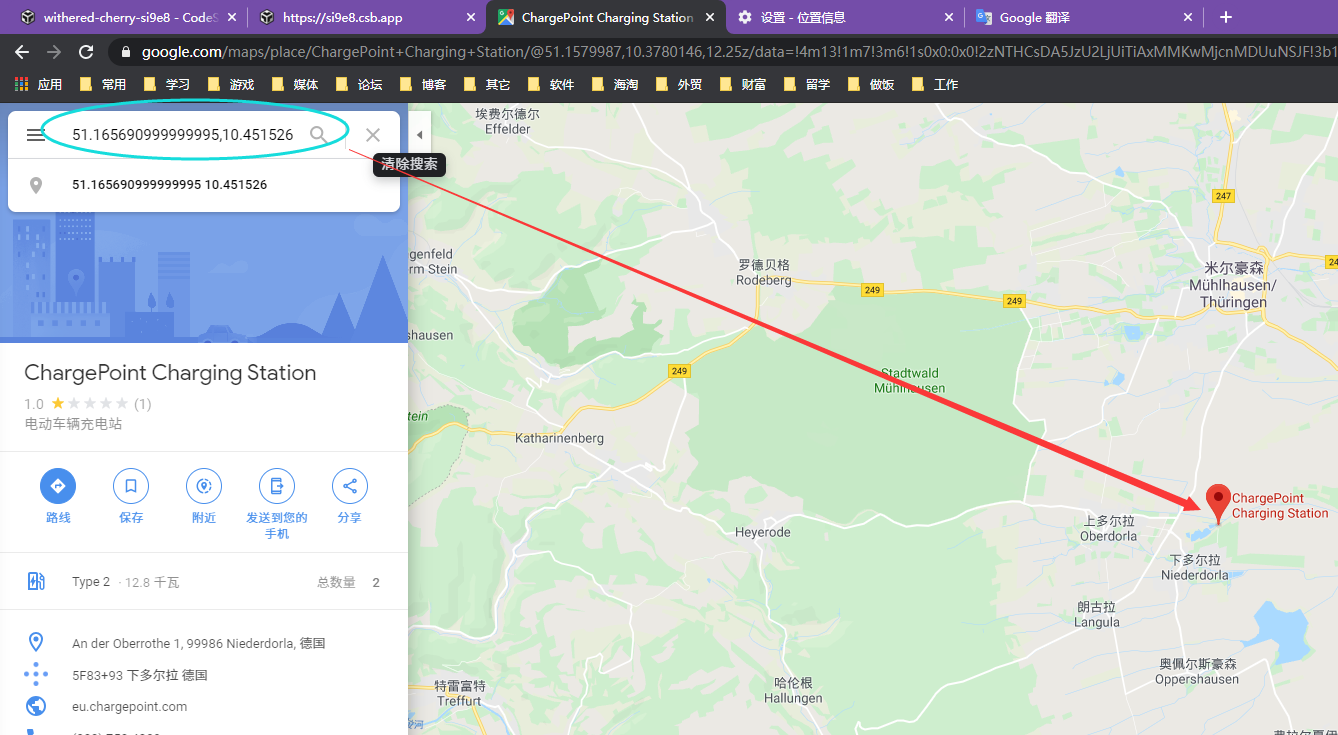
### Js-获取用户地理位置(需要用户同意)

navigator.geolocation.getCurrentPosition(x=>console.log(x)); **// 获取地理位置（需要权限同意）**

navigator.geolocation.watchPosition(e=>console.log(e)) **// 监听地理位置（需要权限同意）**

**返回一个对象：(latitude,longitude)放到谷歌地图即可查看位置**





## redux-persist | 配合redux浏览器本地存储变量数据，代替window.localStorage

安装：yarn add redux-persist

### redux-persist的store.js配置

import { createStore, applyMiddleware } from 'redux';

import logger from 'redux-logger';

**// redux-persist的store.js配置，导入必须的库**

import { persistStore } from 'redux-persist';

import rootReducer from './root-reducer';

const middlewares = [ logger ];

const store = createStore( rootReducer, applyMiddleware(...middlewares) );

**// redux-persist的store.js配置**

const persistor = persistStore(store); // 处理store

export { store, persistor }; // 导出persistor

### redux-persist的root-reducer.js配置

import { combineReducers } from 'redux';

**// redux-persist的root-reducer.js配置，导入必须的库**

import { persistReducer } from 'redux-persist';

import storage from 'redux-persist/lib/storage';

import userReducer from './user/user.reducer';

import cartReducer from './cart/cart.reducer';

import modalReducer from './modal/modal.reducer';

import direReducer from './dire/dire.reducer';

import shopReducer from './shop/shop.reducer';

**// redux-persist基本信息配置**

const persistConfig = {

    key: 'root', // 设定存储的开始位置

    storage, // 导入存储库

    whitelist: ['cart'], // **控制核心：**白名单,放置本地存储的变量目标,以字符串的形式

}

const rootReducer = combineReducers({

    user: userReducer,

    cart: cartReducer,

    modal: modalReducer,

    dire: direReducer,

    shop: shopReducer,

});

**// redux-persist要经过persistReducer( redux-persist配置, redux目录 )**

    // 0. 处理后导出rootReducer

    // 1. 所以说,接收方依然import rootReducer from 'xxx/root-reducer';正常接受即可

export default persistReducer( persistConfig, rootReducer );

### redux-persist在index.js配置

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter } from 'react-router-dom';

import { Provider } from 'react-redux';

**// redux-persist在index.js配置**

    // 0. PersistGate标签

        // a) 将渲染的APP标签嵌套在PersistGate标签中,并将persistor信息传入此标签

        // b) 方便redux-persist抓取目标变量进行处理

import { PersistGate } from 'redux-persist/integration/react';

import { store, persistor } from './redux/store';

import './index.css';

import App from './App';

ReactDOM.render(

    <Provider store={store} >

        <BrowserRouter>

            <PersistGate persistor={persistor} >

                <App />

            </PersistGate>

        </BrowserRouter>

    </Provider>

    ,

    document.getElementById('root')

);

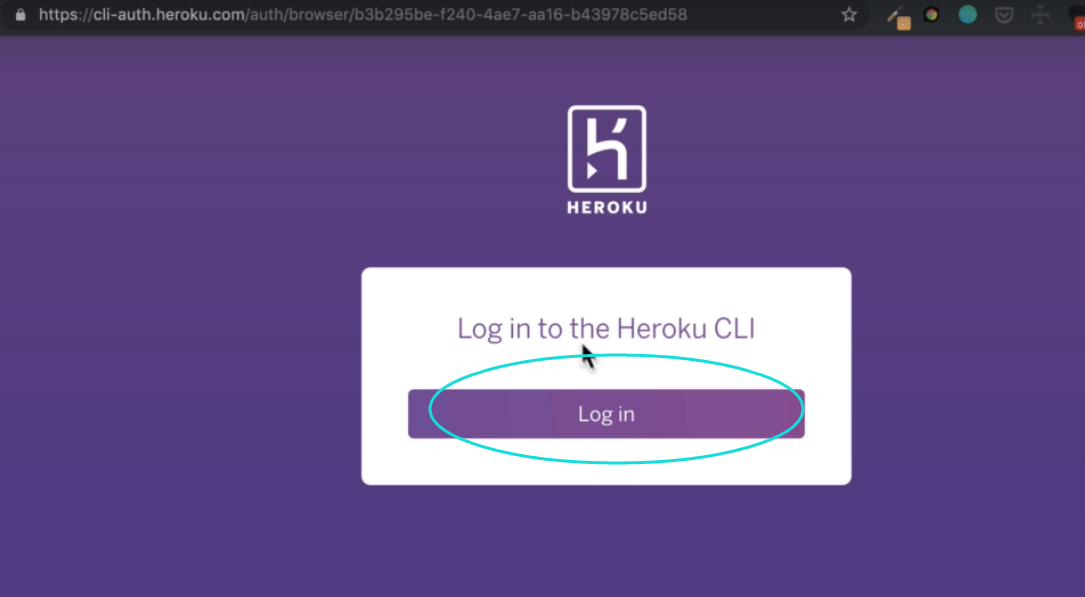
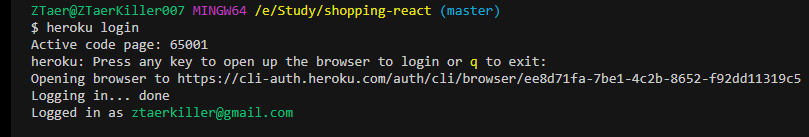
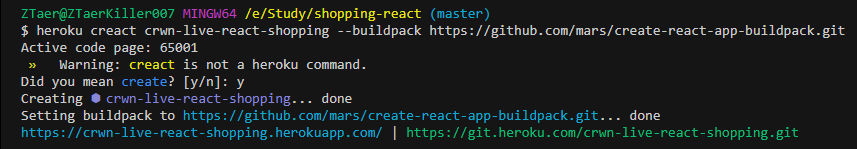
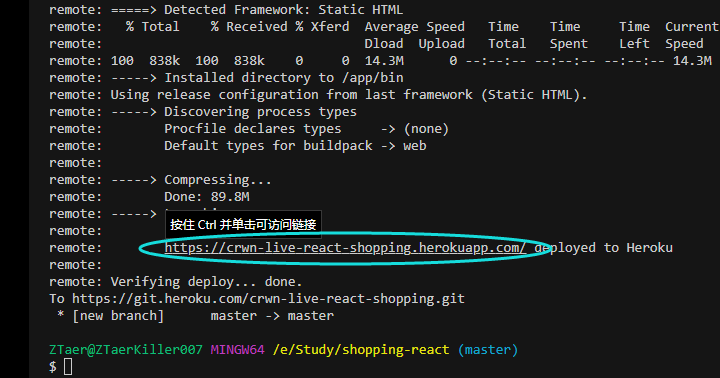
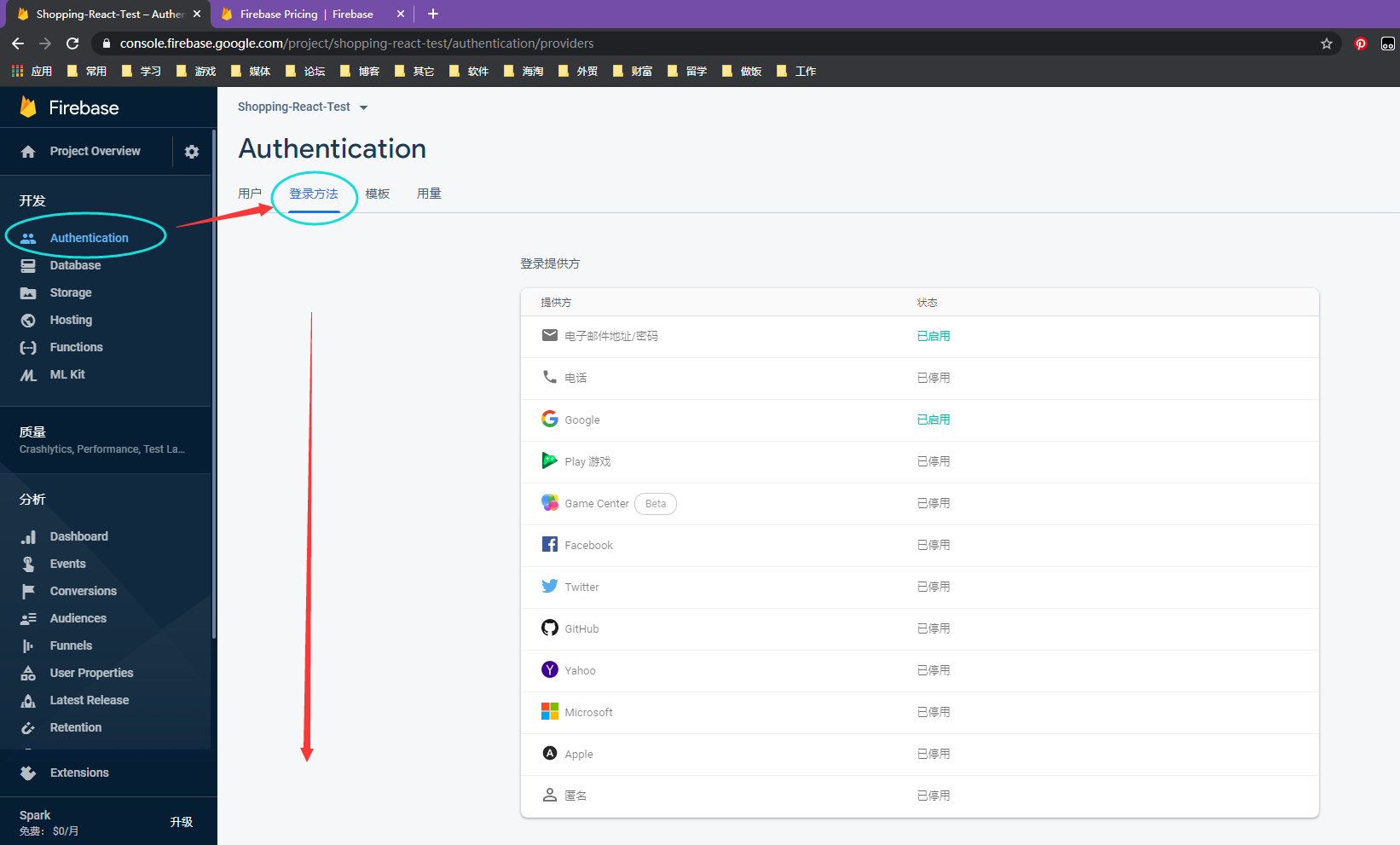
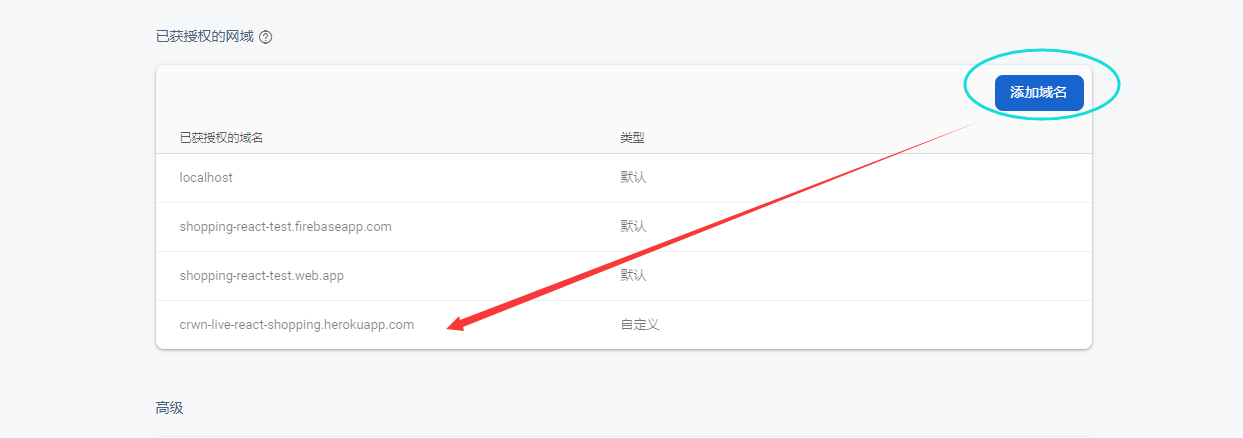
# 2020.1.13( heroku/styled-component )

## Heroku便捷式的部署react项目

Heroku: 云托管公司能快速的部署react项目默认免费

Heroku官网: <https://www.heroku.com/home>

### Heroku安装/heroku部署项目

1. Heroku-cil下载安装: <https://devcenter.heroku.com/articles/getting-started-with-nodejs#set-up>
2. 本地cmd登陆heroku: heroku login
   1. 
   2. 
3. 找到目标项目进行部署( 这里为react构建的电商项目 )：
   1. Cmd指令: heroku creact crwn-live-react-shopping --buildpack <https://github.com/mars/create-react-app-buildpack.git>
   2. 指令解析: heroku 创建项目名称 - -buildpack 从github获取react项目必备组件
   3. 
4. 使用git部署上传到heroku, 类似于git上传到github
   1. Cmd指令: git push heroku master
   2. 
   3. 显示他即可正常访问浏览react项目
5. 添加Firebase域名白名单，保证站点后端功能正常
   1. 
   2. 

### Heroku与github同步更新

参考文档(暂且不推荐)：<https://devcenter.heroku.com/articles/github-integration>

## Styled component | 解决大型项目css冲突的问题

### 注意事项:

考虑项目的最佳方案，并不是越时髦的库就越好，最适合的就是最好的。要考虑到权衡利弊

### 使用此库的原因：

0. 为了解决css全局问题，防止在未来扩展时出现冲突。

1. styled-components能很好的解决这个问题。

2. 而bem库虽然也能解决，但是不够完美，开发人员在扩展查修css时会非常麻烦

3. 安装: yarn add styled-components

### styled-components实列使用方式

/\*\*

**\* styled-components使用方式**

 \*/

import React from 'react';

import styled from 'styled-components';**// 导入此库**

**// 0. styled-components普通构建方式**

const Card = styled.div`

    width:500px;

    height:200px;

    padding:0.5rem;

    display:flex;

    justify-content:center;

    align-items:center;

    background:rgba(0,0,0,0.7);

    border-radius:0.7rem;

    margin:auto;

`;

**// 1. styled-components构建: 配合js构建动态css**

    // a) 核心: `${}` --> `${ ()=>{} }` --> `${ ({ xxx })=>{} }`;

    // b) 在函数中以字符串形式表达css

const Text = styled.p`

    text-align:center;

    letter-spacing:3px;

    /\* 单属性/多属性构建方式 \*/

    color: ${ ({ door }) => door ? 'red' : '#fff' };

    ${ ({door}) => {

        return door ?

       'font-size:2rem; font-weight:bold;' :

       'font-size:3rem; font-weight:lighter;';

    } };

`;

**// 2. styled-components使用: 及使styled标签在不同组件中,也是同样的使用方法**

const TelPage = () => {

    return (

        <Card>

            <Text door={false} >

                styled-components使用动态css, 注意标签中的door的使用

            </Text>

        </Card>

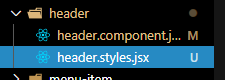
    );

}

export default TelPage;

### STYLED-COMPONENTS在react实战使用方式

#### 首先构建xxx.styles.jsx

* 1. 如

import styled, { css } from 'styled-components';

import { Link } from 'react-router-dom';

const HeaderDiv = styled.div`

    height: 70px;

    width: 95%;

    margin:0px auto;

    display: flex;

    justify-content: space-between;

    margin-bottom: 25px;

    font-family: 'Open Sans Condensed';

`;

**// styled-components自定义标签赋予CSS**

    // 0. 先导入目标自定义标签,此时为Link标签

    // 1. 在使用styled( 目标标签 )

const LogoContainer = styled(Link)`

    height: 100%;

    width: 70px;

    padding: 25px;

`;

const OptionsDiv = styled.div`

    width: 50%;

    height: 100%;

    display: flex;

    align-items: center;

    justify-content: flex-end;

`;

**// styled-component使用css函数构建公共css3属性**

    // 0. 当前为构建公共css

const OptionContainerStyles = css`

    cursor: pointer;

    padding: 10px 15px;

`;

    // 1. 使用公共css

const OptionLink = styled(Link)`

    ${OptionContainerStyles} **/\* 引用公共css3 \*/**

`;

export { HeaderDiv, LogoContainer, OptionsDiv, OptionLink };

#### 至于styled-components使用方式与jsx的自定义标签相同

1. 先import导入，然后可直接书写自定义标签

### styled-components使用as可进行标签类型转换

**// styled-components使用as可进行标签类型转换**

    // a) 使用方式: <Xxx as={'div'} ><Xxx/> --> <div></div>

// b) as可以将组件转换为自定义标签,也可以转换为默认HMTL标签

// 0. 没有as时: 此标签将编译为<a></a>

<OptionLink to='' onClick={ ()=>auth.signOut() } >退出</OptionLink>

// 1. 有as时: 此标签将编译为<div></div>

<OptionLink as='div' to='' onClick={ ()=>auth.signOut() } >退出</OptionLink>

### styled-components构建: 根据组件传输的参数以函数操控css动态变化,以动态按钮为例

/\*\*

**styled-components构建: 根据组件传输的参数以函数操控css动态变化,以动态按钮为例**

 \*      a) 核心: const xxx = props => {}; 其中props可以接受到组件传递来的参数

 \*/

import styled, {css} from 'styled-components';

// 普通登陆界面按钮样式

const signWidthStyles = css`

    width:190px;

`;

// 谷歌按钮样式

let googleColor = '#4284f3';

const googleBtnStyles = css`

    ${signWidthStyles}

    background: ${googleColor};

    border:1px solid ${googleColor};

    &:hover{

        background: #fff;

        color:${googleColor};

        border:1px solid ${googleColor};

    }

`;

// 购物车下拉菜单按钮

const cartDropdownBtnStyles = css`

  margin:0px auto;

  width:100% !important;

`;

// 产品页加入购物车按钮

let animateTimeOnly = '0.1s';

const collectionItemBtnStyles = css`

    width:75%;

    margin-bottom:1rem;

    display: none;

    &:hover{

    background: rgba(255,255,255,0.6);

    }

    &:active{

    transition:all ${animateTimeOnly} linear;

    -moz-transition: all ${animateTimeOnly} linear;

    -webkit-transition: all ${animateTimeOnly} linear;

    background: red;

    color:#fff;

    }

`;

**// 核心: 动态css核心逻辑 - 可接受组件传来的参数**

const getButtonStyles = props => {

    if( props.isGoogleStyles ){

        return googleBtnStyles;

    }

    else if( props.isSignWidthStyles ){

        return signWidthStyles;

    }

    else if( props.isCartDropdownBtnStyles ){

        return cartDropdownBtnStyles;

    }

    else if( props.isCollectionItemBtnStyles ){

        return collectionItemBtnStyles;

    }

};

let animateTime = '0.3s'; // 控制动画时间

const CustomButtonExp = styled.button`

    min-width: 165px;

    width: auto;

    height: 50px;

    letter-spacing: 0.5px;

    line-height: 50px;

    text-align: center;

    font-size: 15px;

    background-color: black;

    color: white;

    text-transform: uppercase;

    font-family: 'Open Sans Condensed';

    font-weight: bolder;

    border: none;

    cursor: pointer;

    letter-spacing: 2px;

    border: 1px solid black;

    transition:all ${animateTime} linear;

    -moz-transition: all ${animateTime} linear;

    -webkit-transition: all ${animateTime} linear;

    &:hover {

        background-color: white;

        color: black;

        border: 1px solid black;

    }

    ${getButtonStyles} **/\*\* 引用逻辑函数, 最好放置最后方便覆盖css样式 \*/**

`;

export { CustomButtonExp };

#### styled-components动态css使用方式

import React from "react";

import CustomButtonExp from "../custom-button-exp/custom-button-exp.component"; **// 导入组件**

class SignIn extends React.Component {

    render(){

        return(

                    <div className="btn-sign">

                        <CustomButtonExp isSignWidthStyles type="submit" >**// 通过参数改变组件按钮样式**

                            登陆

                        </CustomButtonExp>

                        <CustomButtonExp isGoogleStyles onClick={ signInWithGoogle } > **// 通过参数改变组件按钮样式**

                            Google登陆

                        </CustomButtonExp>

                    </div>

        );

    }

}

export default SignIn;

# 2019.1.26

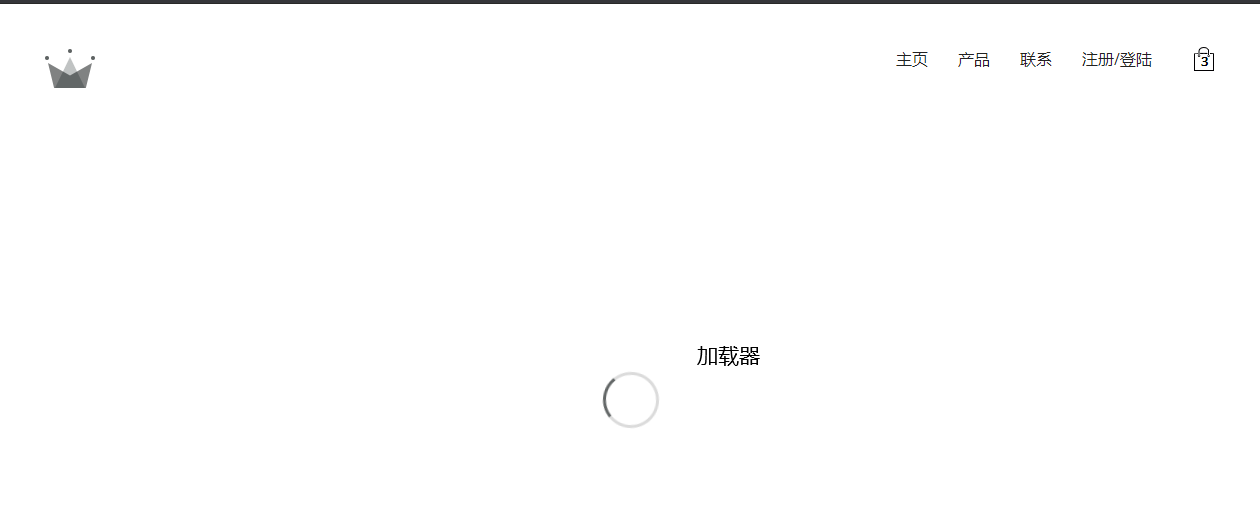
## 取出数组对象中，指定数据

Const collectionsArray = [ {title,items,routename,id}, {title,items,routename,id} ];

collectionsArray.map( ({ title, items })=>({ title, items }) ); // [ {title, items}, {title, items} ]

## // react加载器构建,用于加载异步数据显示加载动画

## // HOCS高级组件使用方式一( 来源: shopping-react\src\components\with-spinner )



### // 构建: react加载器构建,用于加载异步数据显示加载动画

### // hocs高级组件构建方式一

**// 构建: react加载器构建,用于加载异步数据显示加载动画**

import React from 'react';

import { SpinnerContainer, SpinnerOverlay } from './with-spinner.styles';

**// 加载器使用的高级用法**

    // 0. curring函数 - 多个函数嵌套

    // 1. 高阶组件 - 这里使用了, 高阶组件用法

        // a) 传递接受组件, 并加工

const WithSpinner = WrappedComponent => ({ isLoading, ...otherProps }) => {

    return isLoading ? ( // isLoading为真则显示加载器, 否则返回传递来的组件

        <SpinnerOverlay>

            <SpinnerContainer />

        </SpinnerOverlay>

    ) :

    (

        <WrappedComponent { ...otherProps } />

    );

}

export default WithSpinner;

### react加载器在route中的使用

### 有redux时react简化版state写法,可以配合this.setState()使用

### hocs高级组件使用方式一

**// 使用: react加载器的使用**

import WithSpinner from '../../components/with-spinner/with-spinner.component';

import CollectionOverView  from '../../components/collection-overview/collection-overview.component';

import CollectionPage from '../collectionpage/collectionpage.component';

// react加载器: 要使用加载器的组件准备

const CollectionOverViewWithSpinner = WithSpinner( CollectionOverView );

const CollectionPageWithSpinner = WithSpinner( CollectionPage );

class ShopPage extends React.Component {

**// react加载器:****有redux时react简化版state写法,可以配合this.setState()使用**

    state = {

        loadding: true,

    }

    render(){

        const { match } = this.props;

        const { loadding } = this.state;

        return (

**// react加载器在route中的使用**

                // 0. 注意loadding变量来自于this.state

            <div className="shop-page">

                <Route

                    exact

                    path={`${match.path}`}

                    render={

                        props => ( <CollectionOverViewWithSpinner isLoading={ loadding } {...props} /> )

                    }

                />

                <Route

                    path={`${match.path}/:collectionId`}

                    render={

                        props => ( <CollectionPageWithSpinner isLoading={ loadding } {...props} /> )

                    }

                />

            </div>

        );

    }

}

const mapDispatchToProps = dispatch => ({

    updateCollections: collectionsData => dispatch( updateCollections( collectionsData ) ),

});

export default connect(null,mapDispatchToProps)(ShopPage);

## // hocs高级组件构建方式二( 来源: 2020.1.27\hocs-test )

## // 公共帮助不同组件获取异步数据实列

import React from 'react';

import axios from 'axios';

**// hocs高级组件构建方式二: 获取异步数据实列**

    // 0. 所为高级组件, 是接受一个组件进行加工处理

    // 1.**核心:**目标组件使用高级组件后, 高级组件可默认接受目标组件的所有props数据, 以便加工处理

        // a) 所以此时的dataSource可直接使用

    // 2. WrappedComponent: 接受一个组件 - 返回: <WrappedComponent xxx={this.state.xxx} {...this.props} />

        // a) 重点在于,"...this.props"为必备参数

        // b) xxx为当前异步请求的数据

    // 3. 注意: withData( WrappedCommponent, xxx )也可以接受更多的参数, 不要忘记使用时也要加入相应的参数

const withData = (WrappedComponent) => {

    class WithData extends React.Component {

        constructor(props){

            super(props);

            this.state = {

                data:[],

            };

        }

        rqPosts = async () => {

            try{

                const dataBase = await axios(this.props.dataSource);

                this.setState({ data: dataBase.data.slice(0,5) });

            }

            catch{

                alert('请求数据失败!');

            }

        }

        componentDidMount(){

            this.rqPosts();

        }

        render(){

            return this.state.data.length > 1 ?

            ( <WrappedComponent data={this.state.data} {...this.props} /> ) :

            ( <h1>加载中...</h1> );

        }

    }

    return WithData;

}

export default withData;

### // hocs高级组件使用方式二: 使用hocs高级组件

import React from 'react';

import withData from '../with-data/with-data.component';

class UserProfile extends React.Component {

  constructor(props){

    super(props);

  }

  render(){

    const { name, email, data } = this.props;

    return (

      <div className='container'>

        <h1>{name}</h1>

        <h2>{email}</h2>

        测试数据:

        {

          data.map( post => (

            <div key={post.id} className="post">

              <h1>{post.title}</h1>

              <p>{ post.body }</p>

            </div>

          ) )

        }

      </div>

    );

  }

}

**// hocs高级组件使用方式二: 使用hocs高级组件**

export default withData(UserProfile) ;

#### app.js直接使用, 注意”目标组件”的this.props将直接传递到”高阶组件”方便加工处理

import React from 'react';

import './App.scss';

import UserProfile from './component-hocs/user-profile/user-profile.component';

import UserList from './component-hocs/user-list/user-list.component';

function App() {

  return (

    <div className="App">

      <UserProfile

        name={`测试`}

        email={`test@test.com`}

        dataSource='https://jsonplaceholder.typicode.com/posts'

      />

      <UserList

        dataSource='https://jsonplaceholder.typicode.com/users'

      />

    </div>

  );

}

export default App;

# 2020.1.28( redux-thunk )

## 观察者模式/承诺模式

观察者模式 – firebase就为它，主要是监察动态获取的异步数据

承诺模式 – 配合使用redyx thunk 将异步数据写入到redux中

## redux-thunk | 让异步获取数据方式在REDUX中, 通常用于获取后端API数据

安装: yarn add redux-thunk

### // redux-thunk配置store.js

**// redux-thunk配置store.js**

    // 0. 注意中间件加入 const middlewares = [thunk];

import thunk from 'redux-thunk';

// 中间件: 因为中间件后期要添加很多,所以需要解构符来配合

const middlewares = [thunk];

### redux-thunk使用方式: 以获取后端商品数据为例

#### // redux-thunk配置xx.types.js

**// redux-thunk配置xx.types.js**

export const shopActionTypes = {

    AXIOS\_COLLECTIONS\_STATE: 'AXIOS\_COLLECTIONS\_STATE',

    AXIOS\_COLLECTIONS\_SUCCESS: 'AXIOS\_COLLECTIONS\_SUCCESS',

    AXIOS\_COLLECTIONS\_FAILURE: 'AXIOS\_COLLECTIONS\_FAILURE',

}

#### // redux-thunk配置xx.reducer.js

import { shopActionTypes } from './shop.types';

**// redux-thunk配置xx.reducer.js**

    // 0. 注意获取数据的3种状态,此乃redux获取异步数据常用模式( 变量名称自定义 )

        // a) 当前数据获取状态: isAxiosing ( 注意布尔值的变化, 常用于配合加载器 )

        // b) 获取数据成功: collectionShop

        // c) 获取数据失败: errorMsg

const INITIAL\_STATE = {

    collectionShop: null,

    isAxiosing: false, // 获取数据的状态

    errorMsg: undefined,

}

const shopReducer = ( state=INITIAL\_STATE, action ) => {

    switch (action.type) {

        case shopActionTypes.AXIOS\_COLLECTIONS\_STATE:

            return {

                ...state,

                isAxiosing: true,

            }

        case shopActionTypes.AXIOS\_COLLECTIONS\_SUCCESS:

            return{

                ...state,

                isAxiosing: false,

                collectionShop: action.payload,

            }

        case shopActionTypes.AXIOS\_COLLECTIONS\_FAILURE:

            return{

                ...state,

                isAxiosing:false,

                errorMsg: action.payload,

            }

        default:

            return state;

    }

};

export default shopReducer;

#### // redux-thunk配置xx.action.js

import { shopActionTypes } from './shop.types';

**// redux-thunk配置xx.action.js**

import { firestore,convertCollectionsSnapshotToMap } from '../../firebase/firebase.config';

export const axiosCollectionsState = () => ({

    type: shopActionTypes.AXIOS\_COLLECTIONS\_STATE,

});

export const axiosCollectionsFailure = errorMsg => ({

    type: shopActionTypes.AXIOS\_COLLECTIONS\_FAILURE,

    payload: errorMsg,

});

export const axiosCollectionsSuccess = data => ({

    type: shopActionTypes.AXIOS\_COLLECTIONS\_SUCCESS,

    payload: data,

});

**// redux-thunk核心目的: 就是将获取异步数据的函数放在xx.action.js中**

    // 0. redux-thunk已配置在'中间件'中, 所以可以直接使用dispatch()来执行对应的action函数

    // 1. redux thunk 在中间件中会抓取自己力所能及的事件，比如action中的dispatch函数

export const axiosCollectionsStateAsync = () => {

    return dispatch => {

        const collectionsRef = firestore.collection('collections');

        dispatch( axiosCollectionsState() ); // 开启获取数据状态

        const collectionMap = async () => {

            try{

                const snapshot = await collectionsRef.get();

                dispatch( axiosCollectionsSuccess( convertCollectionsSnapshotToMap( snapshot ) ) ); // 注意数据获取成功使用的action

            }

            catch( error ){

                dispatch( axiosCollectionsFailure( error ) ); // 注意数据获取失败使用的action

            };

        }

        collectionMap();

    }

}

#### // REDUX-THUNK配置xx.selectors.js

#### // redux-thunk配置xx.selectors.js核心: 验证异步数据是否存在,配合加载器布尔值,防止出错

#### "!!"符号: 将变量数据转布尔值类型

import { createSelector } from 'reselect';

const selectShop = state => state.shop;

export const selectCollectionShop = createSelector(

    [selectShop],

    shop => shop.collectionShop,

);

export const selectCollectionShopArray = createSelector(

    [selectCollectionShop],

    collectionShop => collectionShop ? Object.keys(collectionShop).map( e=>collectionShop[e] ) : [] , // 防止没有数据时发生错误, 所以返回一个空数组

);

export const selectCollectionItem = collectionUrlProps => createSelector(

    [selectCollectionShop],

    collectionShop => collectionShop ? collectionShop[ collectionUrlProps ] : null

);

**// 选择失败时返回的数据**

export const selectCollectionsFailure = createSelector(

    [selectShop],

    shop => shop.errorMsg,

);

**// 选择异步获取数据的状态**

export const selectCollectionsState = createSelector(

    [selectShop],

    shop => shop.isAxiosing,

);

**// redux-thunk配置xx.selectors.js验证异步数据是否存在,配合加载器布尔值,防止出错**

**// 0.****"!!"符号: 将变量数据转布尔值类型**

        // a) !!"xxx" --> true;

        // b) !!null --> false;

    // 1. 此时当数据存在时返回false隐藏加载器, 否则相反

export const selectCollectionsLoaded = createSelector(

    [selectCollectionShop],

    collectionShop => !( !!collectionShop ),

)

#### // redux-thunk异步数据的使用

import React from 'react';

import "./shopage.styles.scss";

import { Route } from 'react-router-dom';

import CollectionOverView  from '../../components/collection-overview/collection-overview.component';

import CollectionPage from '../collectionpage/collectionpage.component';

import { connect } from 'react-redux';

**// redux-thunk异步数据的使用**

import { axiosCollectionsStateAsync } from '../../redux/shop/shop.action';

import { createStructuredSelector } from 'reselect';

import { selectCollectionsLoaded, selectCollectionsState } from '../../redux/shop/shop.selectors';

// react加载器的使用

import WithSpinner from '../../components/with-spinner/with-spinner.component';

// react加载器: 要使用加载器的组件准备

const CollectionOverViewWithSpinner = WithSpinner( CollectionOverView );

const CollectionPageWithSpinner = WithSpinner( CollectionPage );

class ShopPage extends React.Component {

    componentDidMount(){

       const { axiosCollectionsStateAsync } = this.props;

       axiosCollectionsStateAsync();**// 促发redux异步获取数据( 核心 )**

    }

    render(){

        const { match, isCollectionsAxiosing, isCollectionsLoad } = this.props;

        return (

            <div className="shop-page">

                <Route

                    exact

                    path={`${match.path}`}

                    render={

                        props => ( <CollectionOverViewWithSpinner isLoading={ isCollectionsAxiosing  } {...props} /> )

                    }

                />

                <Route

                    path={`${match.path}/:collectionId`}

                    render={

                        props => ( <CollectionPageWithSpinner isLoading={ isCollectionsLoad } {...props} /> )

                    }

                />

            </div>

        );

    }

}

**// 使用redux基本配置**

const mapStateToProps = createStructuredSelector({

    isCollectionsAxiosing: selectCollectionsState, **// 异步获取数据的状态信息, 配合加载器使用**

    isCollectionsLoad: selectCollectionsLoaded, **// 为更好的配合加载器所用,避免发生错误( 更好的兼容性 )**

});

const mapDispatchToProps = dispatch => ({

    axiosCollectionsStateAsync: ()=>dispatch( axiosCollectionsStateAsync() ),

});

export default connect(mapStateToProps,mapDispatchToProps)(ShopPage);