# 信息保存：

## Svn--账号密码：

MLJ

m123456

地址：

http://WIN-1211111916:9081/svn/5DBIM/bimWebServer/2.x/webapp2.x

## 服务器代理地址：

<http://192.168.1.203:9080>

接口地址：IMG_256http://192.168.1.234:9080/swagger-ui.html#/inventory-controller/xml2INUsingPOST\_3

http://192.168.1.104:9080/swagger-ui.html

清单维护工具路由：

http://localhost:3000/listMaintain/%7B%22PrjID%22:%22ADF95C8C7FD5B4DACA0F1B63F70289E02%22%7D

## 调试--账号密码：

管理员：admin 9999

梁：oak160 0000

我：KFMLJ 0000

胡：SJHYH  1234

权：quan 1234

胡楷：hkhk1 1236

龙：mylong 0000 代理服务器：192.168.0.22:9080

旧页面github地址：http://192.168.0.34:3000/wugh/bimWebAppSpringboot.git

## 禅道--账号密码：

http://121.8.157.232:82/zentao/

账户：1962500337

密码：mlj1962500337

## 蓝湖--网址

https://lanhuapp.com/web/#/item/

## 旧页面运行步骤

1. 打开IDEA，此时是master分支，先更新master分支。
2. 找到要改bug的分支，列如version分支，本地创建后切换，更新，然后在这个分支上创建自己改bug的分支

运行项目点击：

项目运行了之后，在浏览器里输入localhost：9080，就能运行项目

每次更改后，点击锤子，再更新浏览器页面，

如果浏览器不改变，用无痕窗口，或找到那个js文件（刷新页面，network里找js）清除浏览器缓存clear borwser cache

1. 改完之后，commit+push临时分支
2. 切换version分支+pull代码 + 合并临时分支 + push
3. 切换master分支+pull代码 + 合并临时分支 + push

## **旧页面链接新页面**

1. 先打包webstorm：npm run build
2. 旧页面文件夹里找到：



用记事本打开Service-worker.js,删除if(e)return e;代码

1. 把旧页面的static复制到webstorm的build文件夹；
2. 打开localhost:9080;

注意：若只是改旧项目的代码，不改新项目，则第一次打包后就不用打包，直接运行旧项目的代码即可，若新项目有改动，就要重新打包新项目代码重新执行上面的操作。

# React-通用方法：

## 生命周期函数：

1. 默认属性

static defaultProps = {  
 nav: [],  
 list: []  
};

1. **constructor()**:构造函数

组件被加载前最先调用，并且仅调用一次

constructor(props) {

super(props);

this.state = {

content:null,

}

}

1. **componentWillMount()**

组件初始渲染（render()被调用前）前调用，并且仅调用一次；

如果在这个函数中调用setState改变某些状态机，react会等待setState完成后再渲染组件

1. **render()**

渲染挂载组件；

触发条件：（1）初始化加载页面（2）状态机改变setState ( 3 ) 接收到新的props（父组件更新）

里面不能使用setState，否则陷入死循环

1. **componentDidMount()**

render之后被调用，并且仅调用一次，渲染挂载组件；可以使用refs；

一般用于页面一进去就会执行的方法，列如请求页面数据回显等等。

componentDidMount() {  
 this.getHeight();  
 this.getXml(1)  
}

1. **componentWillReceiveProps(nextProps)：**

组件渲染后，当组件接收到新的props时被调用

这里可以使用setState,因为setState触发render重新渲染时，不会触发函数本身

1. **shouldComponentUpdate(nextProps, nextState)：**

组件挂载后（即执行完render），接收到新的state或props时被调用，即每次执行setstate都会执行该函数，来判断是否重新render组件，默认返回true；

接收两个参数：第一个是新的props，第二个是新的state。

里面不能使用setState，否则陷入死循环

1. **componentWillUpdate()：**

在接收到新的props 或者 state，重新渲染 之前立刻调用，在初始化渲染的时候该方法不会被调用

里面不能使用setState，否则陷入死循环

1. **componentDidUpdate()：**

在接收到新的props 或者 state，重新渲染后调用，在初始化渲染的时候该方法不会被调用

里面不能使用setState，否则陷入死循环

1. **componentWillUnmount()：**

组件被卸载前调用

## **数据类型**

<https://www.cnblogs.com/xuniannian/p/7452086.html>

## 数组--方法

<https://www.cnblogs.com/obel/p/7016414.html>

JavaScript的 **Array** 对象是用于构造数组的全局对象，数组是类似于列表的高阶对象。

创建数组：

创建数组：

var fruits = new Array('Apple', 'Banana');

或：

var fruits = ['Apple', 'Banana'];

console.log(fruits.length);// 2

通过索引访问元素：

如果指定的索引是一个无效值，JavaScript数组并不会报错，而是会返回undefined

var first = fruits[0];// Apple

var last = fruits[fruits.length - 1];// Banana

也可以用字符串表示索引：

var first = fruits['0'];// Apple

var first = fruits['01'];// undefined

遍历数组：

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

console.log(item, index);

});// Apple 0// Banana 1

let *arr* = ['a','b','c','b','a','c','d'];  
for(let *i*=0;*i*<*arr*.length;*i*++){  
 console.log(*i*)  
}//0,1,2,3,4,5,6

倒叙：

let *arr* = ['a','b','c','b','a','c','d'];  
for(let *i* = *arr*.length-1;*i* > -1;*i*--){  
 console.log(*i*)

}

//6,5,4,3,2,1,0

若没有对数组的删除添加操作，只是数据间的对比，就可直接用底下的正序的方式，不需要用倒序

//去重错误示范：这种删除会漏掉删除的那个的索引。例如此时i指向第三个，删除第三个后，后面都往前移一位，往后，i指向第四个，要循环第四个，但此时的第四个是原来的第五个，原来的第五个已经成为了第四个，因此就漏掉了原来的第四个（该索引删除后，该索引就会被漏掉）  
let *arr* = ['a','b','c','b','a','c','a','a'];  
for(let *i* = 0;*i*<*arr*.length;*i*++){  
 for(let *j*= *i*+1;*j*<*arr*.length;*j*++){  
 if(*arr*[*i*] === *arr*[*j*]){  
 *arr*.splice(*j*,1)  
 }  
 }  
}

//去重正确示范：第一层用正序，第二层用倒序，这样就不会出现索引遗漏。例如从第八个开始跟第一个对比，删除第八个后，开始第七个和第一个对比，索引j也是指向第七个，完美  
let *arr* = ['a','b','c','b','a','c','a','a'];  
for(let *i* = 0;*i*<*arr*.length;*i*++){  
 for(let *j*= *arr*.length - 1;*j* > *i*;*j*--){  
 if(*arr*[*i*] === *arr*[*j*]){  
 *arr*.splice(*j*,1)  
 }  
 }  
}

循环解析：

//i的起始位置处，就算是一个变量，但是也只是取最初始的那个值，此后每一次循环的i的变化跟num都没有关系，无论num如何变化，i都按照自己的方式递加下去；  
//但i的终止处，会受变量的影响，如下，循环过程里arr的长度改变了，则i的终止长度也改变了  
let *num* = 0;  
for(let *i* = *num*;*i*<*arr*.length;*i*++){  
 *arr*.shift();  
 *num*--;  
 console.log(*num*,*i*,*arr*.length)  
}

//-1 0 7

//-2 1 6

//-3 2 5

//-4 3 4

### Array方法

#### Array.from()

Array.from() 可以通过以下方式来创建数组对象：

伪数组对象（拥有一个 length 属性和若干索引属性的任意对象）

[可迭代对象](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/iterable)（可以获取对象中的元素,如 Map和 Set 等）

Array.from() 方法有一个可选参数 mapFn，让你可以在最后生成的数组上再执行一次 [map](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/map" \o "map() 方法创建一个新数组，其结果是该数组中的每个元素都调用一个提供的函数后返回的结果。) 方法后再返回。也就是说 Array.from(obj, mapFn, thisArg) 就相当于 Array.from(obj).map(mapFn, thisArg), 除非创建的不是可用的中间数组。

from() 的 length 属性为 1 ，即Array.from.length === 1。

字符串转成数组：

Array.from('foo'); // ["f", "o", "o"]

set对象转成数组：

let s = new Set(['foo', window]);

Array.from(s); // ["foo", window]

Map对象转成数组：

let m = new Map([[1, 2], [2, 4], [4, 8]]);

Array.from(m); // [[1, 2], [2, 4], [4, 8]]

将函数的传参转成数组：

function f() {

return Array.from(arguments);}

f(1, 2, 3); //[1,2,3]

在Array.from()里使用箭头函数

Array.from([1, 2, 3], x => x + x);

// 箭头的意思表示从当前数组中取出一个值现在箭头后面的操作，并将返回的结果添加到新数组中

生成一个数字系列：

Array.from({length: 5}, (v, i) => i);// [0, 1, 2, 3, 4]

数组去重：

let arr = [1,1,2,2,3,3]

Array.from(new Set(arr));//[1, 2, 3]

#### **Array.isArray()**

用于确定传递的值是否是一个 [Array](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Array" \o "REDIRECT Array)

// 下面的函数调用都返回 true

Array.isArray([]);

Array.isArray([1]);

Array.isArray(new Array());// 鲜为人知的事实：其实 Array.prototype 也是一个数组。

Array.isArray(Array.prototype);

// 下面的函数调用都返回 false

Array.isArray();

Array.isArray({});

Array.isArray(null);

Array.isArray(undefined);

Array.isArray(17);

Array.isArray('Array');

Array.isArray(true);

Array.isArray({ \_\_proto\_\_: Array.prototype });

#### Array.of()

创建一个具有可变数量参数的新数组实例，而不考虑参数的数量或类型。

 Array.of() 和 Array 构造函数之间的区别在于处理**整数参数**：Array.of(7) 创建一个具有单个元素 7 的数组，而 Array(7) 创建一个长度为7的空数组（注意：这是指一个有7个空位(empty)的数组，而不是由7个undefined组成的数组）。

Array.of(7); // [7]

Array.of(1, 2, 3); // [1, 2, 3]

Array(7); // [ , , , , , , ]

Array(1, 2, 3); // [1, 2, 3]

**[Array.prototype.length](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/length" \o "length 是Array的实例属性。返回或设置一个数组中的元素个数。该值是一个无符号 32-bit 整数，并且总是大于数组最高项的下标。)**

上面说了，因为 Array.prototype 也是个数组，所以它也有 length 属性，这个值为 0，因为它是个空数组。

额外·补充：

数组可以通过设置length长度来删减长度；

let *fruits* = [1,2,3,4]  
*fruits*.length = 2;  
console.log(*fruits*); // [1, 2]

let *fruits* = [1,2,3,4]  
*fruits*.length = 7;  
*fruits*[5] = '000'  
console.log(*fruits*); // [1, 2, 3, 4, empty, "000", empty]  
console.log(*fruits*[4]);//undefined

### **数组实例-修改器方法：**

**下面的这些方法会改变调用它们的对象自身的值**

添加元素到数组的末尾：

var newLength = fruits.push('Orange');

// newLength:3; fruits: ["Apple", "Banana", "Orange"]

删除数组末尾的元素此方法更改数组的长度

var last = fruits.pop();

// last: "Orange"; fruits: ["Apple", "Banana"];

从数组中删除**第一个**元素，并返回该元素的值。此方法更改数组的长度。

var first = fruits.shift();

// remove Apple from the front// first: "Apple"; fruits: ["Banana"];

给数组的元素添加头部：

var newLength = fruits.unshift('Strawberry')

// add to the front// ["Strawberry", "Banana"];

将数组中元素的位置颠倒，并返回该数组。该方法会改变原数组

var sourceArray = ['one', 'two', 'three'];var reverseArray = sourceArray.reverse();

console.log(sourceArray ) // ['three', 'two', 'one']

console.log(sourceArray === reverseArray); // tru

找出某个元素在数组中的索引：

fruits.push('Mango');// ["Strawberry", "Banana", "Mango"]

var pos = fruits.indexOf('Banana');// 1

通过索引删除某个元素：

fruits.push('Mango');// ["Strawberry", "Banana", "Mango"]

var pos = fruits.indexOf('Banana');// 1

从一个索引位置删除多个元素：

var vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot'];

var removedItems = vegetables.splice(1, 2);

console.log(vegetables); // ["Cabbage", "Carrot"]

console.log(removedItems); // ["Turnip", "Radish"]

复制一个数组：(深复制）

var shallowCopy = fruits.slice(); ["Strawberry", "Mango"]

Console.log(shallowCopy); // ["Apple", "Banana", "aaa"]

shallowCopy [2] = "aaa";//对数组进行修改后。ShallowCopy已经被修改了。因此无论在哪里打印出来都是修改后的数据

Console.log(shallowCopy); // ["Apple", "Banana", "aaa"]

Console.log(fruits); // ["Apple", "Banana"]（这两个变量已经是不同的存储地址，因此互不影响。Fruits还是原来的）

若上面没有slice（），则fruits和shallowCopy是同一个存储地址，因此三次打印都是一样的，都是修改后的数组

#### sort()：

对数组元素进行排序，并返回当前数组

**函数格式：**

function compare(a, b) {

if (a < b ) {

return -1; //升序

}

if (a > b ) {

return 1; //降序

}

//相同

return 0;//位置不变

}

**sort方法可以使用函数表达式方便地书写：**

var numbers = [4, 2, 5, 1, 3];

numbers.sort(function(a, b) {

return a - b;});

console.log(numbers);// [1, 2, 3, 4, 5]

**也可以写成：**

var numbers = [4, 2, 5, 1, 3];

numbers.sort((a, b) => a - b);

console.log(numbers);// [1, 2, 3, 4, 5]

**对象可以按照某个属性排序**

var items = [

{ name: 'Edward', value: 21 },

{ name: 'Sharpe', value: 37 },

{ name: 'And', value: 45 },

];

**// 根据value进行升序排练（前一个减后一个）**

**items.sort((a, b)=> a.value - b.value) ;**

**// 根据value进行降序排练（后一个减前一个）**

**items.sort((a, b)=> b.value - a.value) ;**

// 根据名称排序

items.sort(function(a, b) {

var nameA = a.name.toUpperCase(); // ignore upper and lowercase

var nameB = b.name.toUpperCase(); // ignore upper and lowercase

if (nameA < nameB) {

return -1;

}

if (nameA > nameB) {

return 1;

}

// 名称相同

return 0;

});

### **数组实例-访问方法：**

下面的这些方法绝对不会改变调用它们的对象的值，只会返回一个新的数组或者返回一个其它的期望值。

#### **[Array.prototype.concat()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/concat" \o "concat() 方法用于合并两个或多个数组。此方法不会更改现有数组，而是返回一个新数组。)**

返回一个由当前数组和其它若干个数组或者若干个非数组值组合而成的新数组。

****注意：****数组/值在连接时保持不变。此外，对于新数组的任何操作（仅当元素不是对象引用时）都不会对原始数组产生影响，反之亦然。

连接两个数组[节](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/concat" \l "Example_Concatenating_two_arrays)

以下代码将两个数组合并为一个新数组：

var alpha = ['a', 'b', 'c'];var numeric = [1, 2, 3];

alpha.concat(numeric);// result in ['a', 'b', 'c', 1, 2, 3]

连接三个数组[节](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/concat" \l "Example_Concatenating_three_arrays)

以下代码将三个数组合并为一个新数组：

var num1 = [1, 2, 3],

num2 = [4, 5, 6],

num3 = [7, 8, 9];

var nums = num1.concat(num2, num3);

console.log(nums); // results in [1, 2, 3, 4, 5, 6, 7, 8, 9]

将值连接到数组[节](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/concat" \l "Example_Concatenating_values_to_an_array)

以下代码将三个值连接到数组：

var alpha = ['a', 'b', 'c'];

var alphaNumeric = alpha.concat(1, [2, 3]);或

var alphaNumeric = alpha.concat(1, 2, 3]);或

var alphaNumeric = alpha.concat([1, 2, 3]);

都是一样的效果

console.log(alphaNumeric); // results in ['a', 'b', 'c', 1, 2, 3]

合并嵌套数组[节](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/concat" \l "%E5%90%88%E5%B9%B6%E5%B5%8C%E5%A5%97%E6%95%B0%E7%BB%84)

以下代码合并数组并保留引用：

var num1 = [[1]]，num2 = [2, [3]]，num3=[5,[6]];

var nums = num1.concat(num2);

console.log(nums);// [[1], 2, [3]]

var nums2=num1.concat(4,num3);

console.log(nums2)// results is [[1], 4, 5,[6]]

// 修改num1的第一个元素，因为第一个元素为array，因此改变数组后，最初的num1也会改变，nums也随之改变

num1[0].push(4);

console.log(nums);// results is [[1, 4], 2, [3]]

num1[1] = 4;此时虽然改变了num1，但不会影响上面的num1，是两个不同的num1

#### **[Array.prototype.includes()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/includes" \o "includes() 方法用来判断一个数组是否包含一个指定的值，根据情况，如果包含则返回 true，否则返回false。)**

判断当前数组是否包含某指定的值（字符串区分大小写），如果是返回 true，否则返回 false。

注意：对象数组不能使用includes方法来检测。

第一个参数：要查找的字符，第二个参数：开始查找的位置

[1, 2, 3].includes(2); // true

[1, 2, 3].includes(4); // false

[1, 2, 3].includes(3, 3); // false（若是开始位置大于数长度，则该数组不会被搜索）

[1, 2, 3].includes(3, -1); // true（为负数，则从左到右开始查找）

[1, 2, NaN].includes(NaN); // true

判断传输的一系列参数中是否含某个值（只能用于函数的单个单个的参数，不能是数组对象）

let event = function() {  
 let *a* = [].includes.call(arguments, 'a');  
 console.log(*a*) //true  
 let *d* = [].includes.call(arguments, 'd');  
 console.log(*d*); // false  
};  
  
event('a','b','c');

#### **[Array.prototype.join()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/join" \o "join() 方法将一个数组（或一个类数组对象）的所有元素连接成一个字符串并返回这个字符串。如果数组只有一个项目，那么将返回该项目而不使用分隔符。)**

**join()** 方法将一个数组（或一个[类数组对象](https://developer.mozilla.org/zh-CN//docs/Web/JavaScript/Guide/Indexed_collections" \l "Working_with_array-like_objects)）的所有元素连接成一个字符串并返回这个字符串。如果数组只有一个项目，那么将返回该项目而不使用分隔符。

var a = ['Wind', 'Rain', 'Fire'];

var b = [];

var c = [undefined,null];//这两个值会返回空字符串

var myVar1 = a.join(); // myVar1的值变为"Wind,Rain,Fire" 如果省略()，数组元素用逗号分隔。默认为 ","

var myVar2 = a.join(', '); // myVar2的值变为"Wind, Rain, Fire"

var myVar3 = a.join(' + '); // myVar3的值变为"Wind + Rain + Fire"

var myVar4 = a.join(''); // myVar4的值变为"WindRainFire"

var myVar5 = b.join(); //myVar5的值变为''

var myVar6 = c.join(); //myVar5的值变为','

连接类数组对象：

let event = function () {

var s = Array.prototype.join.call(arguments);

console.log(s); // '1,a,true'

}

event(1, 'a', true);

#### **[Array.prototype.slice()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/slice" \o "slice() 方法返回一个新的数组对象，这一对象是一个由 begin 和 end 决定的原数组的浅拷贝（包括 begin，不包括end）。原始数组不会被改变。)**

**slice()** 方法抽取当前数组中的一段元素组合成一个新数组，这一对象是一个由 begin 和 end 决定的原数组的**浅拷贝**（包括 begin，不包括end）。原始数组不会被改变。

**[Array.prototype.toSource()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/toSource" \o "返回一个字符串,代表该数组的源代码.)**

返回一个表示当前数组字面量的字符串。遮蔽了原型链上的 [Object.prototype.toSource()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/toSource" \o "toSource()方法返回一个表示对象源代码的字符串。) 方法。

**[Array.prototype.toString()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/toString" \o "toString() 返回一个字符串，表示指定的数组及其元素。)**

返回一个由所有数组元素组合而成的字符串。遮蔽了原型链上的 [Object.prototype.toString()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/toString" \o "toString() 方法返回一个表示该对象的字符串。) 方法。

**[Array.prototype.toLocaleString()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/toLocaleString" \o "toLocaleString() 返回一个字符串表示数组中的元素。数组中的元素将使用各自的 toLocaleString 方法转成字符串，这些字符串将使用一个特定语言环境的字符串（例如一个逗号 \",\"）隔开。)**

返回一个由所有数组元素组合而成的本地化后的字符串。遮蔽了原型链上的 [Object.prototype.toLocaleString()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/toLocaleString" \o "toLocaleString() 方法返回一个该对象的字符串表示。此方法被用于派生对象为了特定语言环境的目的（locale-specific purposes）而重载使用。) 方法。

**[Array.prototype.indexOf()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf" \o "indexOf()方法返回在数组中可以找到一个给定元素的第一个索引，如果不存在，则返回-1。)**

返回数组中第一个与指定值相等的元素的索引，如果找不到这样的元素，则返回 -1。

**[Array.prototype.lastIndexOf()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/lastIndexOf" \o "lastIndexOf() 方法返回指定元素（也即有效的 JavaScript 值或变量）在数组中的最后一个的索引，如果不存在则返回 -1。从数组的后面向前查找，从 fromIndex 处开始。)**

返回数组中最后一个（从右边数第一个）与指定值相等的元素的索引，如果找不到这样的元素，则返回 -1。

### 数组实例-迭代方法：

**[Array.prototype.forEach()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach" \o "forEach() 方法对数组的每个元素执行一次提供的函数。)**

为数组中的每个元素执行一次回调函数。

**[Array.prototype.entries()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/entries" \o "entries() 方法返回一个新的Array Iterator对象，该对象包含数组中每个索引的键/值对。)**

返回一个数组迭代器对象，该迭代器会包含所有数组元素的键值对。

**[Array.prototype.every()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/every" \o "every() 方法测试一个数组内的所有元素是否都能通过某个指定函数的测试。它返回一个布尔值。)**

如果数组中的每个元素都满足测试函数，则返回 true，否则返回 false。

**[Array.prototype.some()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/some" \o "some() 方法测试是否至少有一个元素可以通过被提供的函数方法。该方法返回一个Boolean类型的值。)**

如果数组中至少有一个元素满足测试函数，则返回 true，否则返回 false。

**[Array.prototype.filter()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/filter" \o "filter() 方法创建一个新数组, 其包含通过所提供函数实现的测试的所有元素。)**

将所有在过滤函数中返回 true 的数组元素放进一个新数组中并返回。

**[Array.prototype.find()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/find" \o "find() 方法返回数组中满足提供的测试函数的第一个元素的值。否则返回 undefined。)**

找到第一个满足测试函数的元素并返回那个元素的值，如果找不到，则返回 undefined。

**[Array.prototype.findIndex()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex" \o "findIndex()方法返回数组中满足提供的测试函数的第一个元素的索引。否则返回-1。)**

找到第一个满足测试函数的元素并返回那个元素的索引，如果找不到，则返回 -1。

**[Array.prototype.keys()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/keys" \o "keys() 方法返回一个包含数组中每个索引键的Array Iterator对象。)**

返回一个数组迭代器对象，该迭代器会包含所有数组元素的键。

**[Array.prototype.map()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/map" \o "map() 方法创建一个新数组，其结果是该数组中的每个元素都调用一个提供的函数后返回的结果。)**

返回一个由回调函数的返回值组成的新数组。

**[Array.prototype.reduce()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce" \o "reduce() 方法对数组中的每个元素执行一个由您提供的reducer函数(升序执行)，将其结果汇总为单个返回值。)**

从左到右为每个数组元素执行一次回调函数，并把上次回调函数的返回值放在一个暂存器中传给下次回调函数，并返回最后一次回调函数的返回值。

**[Array.prototype.reduceRight()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight" \o "reduceRight() 方法接受一个函数作为累加器（accumulator）和数组的每个值（从右到左）将其减少为单个值。)**

从右到左为每个数组元素执行一次回调函数，并把上次回调函数的返回值放在一个暂存器中传给下次回调函数，并返回最后一次回调函数的返回值。

**[Array.prototype.values()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/values" \o "values() 方法返回一个新的 Array Iterator 对象，该对象包含数组每个索引的值)**

返回一个数组迭代器对象，该迭代器会包含所有数组元素的值。

**[Array.prototype[@@iterator]()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/@@iterator" \o "@@iterator 属性和 Array.prototype.values() 属性的初始值均为同一个函数对象。)**

和上面的 values() 方法是同一个函数。

## 字符串--方法

### 指定下标输出字符

举例：

aaa:'gtwgav asgrgva',  
bbb:'孟丽娇',

**单个字符：**

**str.charAt(0) ---根据下标输出字符**

**let** aa = aaa.charAt(0); //g  
**let** bb = bbb.charAt(0); //孟

**str.charCodeAt(0)---根据下标输出字符值编码**

**let** aa = aaa.charCodeAt(0); //103  
**let** bb = bbb.charCodeAt(0); //23391

**截取：**

**str.substr(); //参数1：指定下标，参数2：指定长度（若无参数2，则取到结尾为止）**

**let** aa = aaa.substr(0,3); //gtw  
**let** bb = bbb.substr(0,2); //孟丽

**let** bb = bbb.substr(0); //孟丽娇

**str.substring(0); //参数1：指定下标，参数2：指定长度（若无参数2，则取到结尾为止）**

**let** aa = aaa.substring(0,3); //gtw

**let** bb = bbb.substring(1,5); //丽娇  
**let** bb = bbb.substring(0); //孟丽娇

**str.slice(); //参数1：开始下标，参数2：结束下标（注意：不包括结束位置的字符）**

**let** aa = aaa.slice(1,5); //twga  
**let** bb = bbb.slice(1,5); //丽娇

### 指定字符输出下标

举例：

aaa:'gtwgav asgrgva',  
bbb:'孟丽娇',

**Str.indexOf()---查找指定字符串，输出下标（‘5’为开始查找位置的下标）**

**let** aa = aaa.indexOf('a',5) //7

**let** bb = bbb.indexOf('娇') //2

**str.lastIndexOf()---从后开始向前查找**

**let** aa = aaa.lastIndexOf('a'); //13  
**let** bb = bbb.lastIndexOf('娇'); //2

**str.search()---根据下标输出字符（不支持第二个参数（开始位置））**

**let** aa = aaa.search('a'); //4  
**let** bb = bbb.search('孟'); //0

### 字符串拼接

举例：

aaa:'gtwgav asgrgva',  
bbb:'孟丽娇',

**str.concat()---拼接**

**let** aa = aaa.concat('哈哈哈'); //gtwgav asgrgva哈哈哈  
**let** bb = bbb.concat('哈哈哈'); //孟丽娇哈哈哈

**str.length---字符串长度**

**let** aa = aaa.length; //14  
**let** bb = bbb.length; //3

### 字符串->数组

**str****.split(); //输出字符串以指定字符分割为数组**

**let** aa = aaa.split(' '); //["gtwgav", "asgrgva"]

**let** aa = aaa.split('a'); //["gtwg", "v ", "sgrgv", ""]

**let** aa = aaa.split('/'); //["gtwgav asgrgva"]

**let** bb = bbb.split(); //["孟丽娇"]

**let** bb = bbb.split(''); //["孟", "丽", "娇"]

### 大小写\编码转换

**str.toLowerCase(); //输出转换为小写字母的字符**

**str.toUpperCase(); //输出转换为大写字母的字符**

**let** aa = aaa.toUpperCase(); //GTWGAV ASGRGVA

**String.fromCharCode(10000); //把Unicode编码转换成对应的文字**

**let** bb = String.fromCharCode(23391); //孟

**[js字符串与Unicode编码互相转换](https://www.cnblogs.com/lwwen/p/6862080.html)：**

'好'.charCodeAt(0).toString(16) //"597d"

'好哦'.charCodeAt(0).toString(16) //"597d"

'好哦'.charCodeAt(1).toString(16) //"54e6"

### 其他类型->字符串

**str.toString(); //把其他类型的数据输出为字符串类型的数据**

**let** arr = ['a','b']  
**let** bb = arr.toString(); //a,b

**去除字符串里所有的空格：**

**let** reg = /([^\s])/g;  
**let** arr = value.match(reg).join('');  
console.log(arr);

**日期排序：**

(a, b) => (  
 (a.fa.split('-')[0]\*31536000+a.fa.split('-')[1]\*2592000+a.fa.split('-')[2]\*86400)-(b.fa.split('-')[0]\*31536000+b.fa.split('-')[1]\*2592000+b.fa.split('-')[2]\*86400)  
),

拼接字符串注意：

数组或对象，如果拼接了字符串。就不显示内容。所以要打印数组或字符串，就不要拼接字符串。

### **toString()和JSON.stringify()的区别**

let *a* = [1,2,3]  
let *data* = *a*.toString()  
console.log(*data*) //'1,2,3'

let *a* = [1,2,3]  
let *data* = JSON.stringify(*a*)  
console.log(*data*) //'[1,2,3]'

* 哪些数据类型可以用toString（）

页面报错：

null.toString()、undefined.toString()

可以用的数据类型：

number：112.toString() //"112"

string：‘aab’.toString() //"aab"

boolen:true.toString() //"true"

array:[1,2,3].toString() //"1,2,3"

多维array:[[1,2],3,[[6]]].toString() //"1,2,3,6"

NaN:NaN.toString() //"NaN"

不可用：

json对象：[{a:1},{b:2}].toString(),//"[object Object],[object Object]" (不可用)

object：{a:1,b:2}.toString() //"[object Object]"

* JSON.stringify()

哪些数据可使用JSON.stringify()

数组

let *text* = [true,1,'c',undefined,null,NaN];  
let *data* = JSON.stringify(*text*) //"[true,1,"c",null,null,null]"

let *text* = "[true,1,\"c\",null,null,null]"  
let *data* = JSON.parse(*text*) //[true, 1, "c", null, null, null]

数组对象，多维数组

let *text* = [{a:1},{b:true,c:'ece',d:null},[true,1,'e']];  
let *data* = JSON.stringify(*text*) //"[{"a":1},{"b":true,"c":"ece","d":null},[true,1,"e"]]"

let *text* = "[{\"a\":1},{\"b\":true,\"c\":\"ece\",\"d\":null},[true,1,\"e\"]]"  
let *data* = JSON.parse(*text*) //[{a:1},{b:true,c:'ece',d:null},[true,1,'e']]

对象：

let *text* = {a:true,b:1,c:'cece',d:[2,3],e:{e\_1:'dsdsd'}}  
let *data* = JSON.stringify(*text*) //"{"a":true,"b":1,"c":"cece","d":[2,3],"e":{"e\_1":"dsdsd"}}"

let *text* = "{\"a\":true,\"b\":1,\"c\":\"cece\",\"d\":[2,3],\"e\":{\"e\_1\":\"dsdsd\"}}"  
let *data* = JSON.parse(*text*) //{a:true,b:1,c:'cece',d:[2,3],e:{e\_1:'dsdsd'}}

基本数据类型：

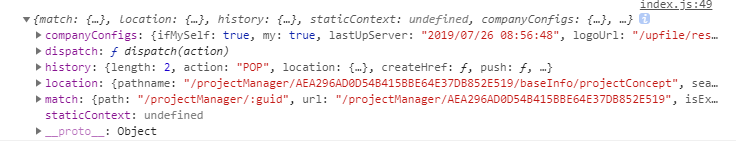
let *text* = 'aaa' 1 true undefined null NaN  
let *data* = JSON.stringify(*text*) // ""aaa"" "1" "true" undefined "null" "null"

let *text* = ""aaa"" "1" "true" undefined "null" "NaN"  
let *data* = JSON.parse(*text*) //"aaa" 1 true 报错 null 报错

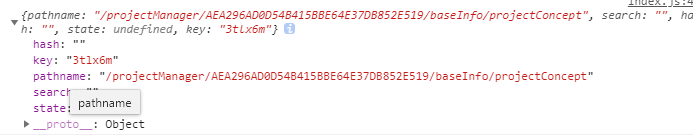
## 对象--方法：

### 解构：

console.log(this.props)



const {*location*} = this.props;  
console.log(*location*);



const { location: { *pathname* }} = this.props  
console.log(*pathname*);



const { location: { *pathname* }, *history* } = this.props

console.log(location,*pathname*,*history*); /三个都打印出对应的属性值

const { match: { params: { *guid* } } } = this.props;

相当于：

const *match* = this.props.match;  
const *params* = this.props.match.params;  
const *guid* = this.props.match.params.guid;

### 合并多个对象：

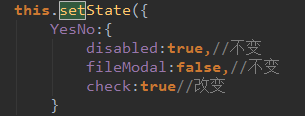
var o1 = { a: 1 };  
var o2 = { b: 2 };  
var o3 = { c: 3 };

var obj = Object.assign(o1, o2, o3);

### 与setState有关

状态为对象时：

当初始状态里的属性为一个对象时，要修改这个对象里的属性时，必须对象里的全部属性一起修改，就算只改变其中的一个，也必须全部写出来。否则没写出来的就没了。



setState方法里放的本身就是一个对象，因此可以先在外面设置好对象直接将整个对象丢进去

let *obj* = {  
 file: {},  
 img: []  
};  
this.setState(*obj*);

等同于：

this.setState({  
 file: {},  
 img: []  
});

### 对象的其他应用：

把一个赋值了字符串的变量直接给一个对象里，会生成：变量为对象的属性，字符串为属性值。

let *link* = 'VNAHFGIIOAWGBVK';   
console.log({*link*})



let *link* = 123;  
console.log({*link*})



let *link* = [1,2];  
console.log({*link*})



let *link* = true;  
console.log({*link*})



let *link* = '123';  
*link* = {*link*};  
console.log({*link*})



如果对象的属性中有数字，或者该属性是一个变量，则必须用[]来调用。

在 JavaScript 中，以数字开头的属性不能用点号引用，必须用方括号

var *robotA* = { 'd2': "Bender" };  
var *robotB* = { '2d':"Flexo" };  
console.log(*robotA*.d2) //"Bender"  
console.log(*robotB*['2d'])//"Flexo"

let *key* = '2d';  
console.log(*robotB*[*key*])//"Flexo"

类似地，如果对象的属性名称是保留字（最好不要这么做！），那么就只能通过字符串的形式用方括号来访问（从 firefox 40.0a2 开始也支持用点号访问了）：

var promise = {

'var' : 'text',

'array': [1, 2, 3, 4]};

console.log(promise['var']);

## **解构赋值**

### 数组的解构：

let *arr* = [1,'22',true];  
let [*a*,*b*,*c*] = *arr*;  
console.log(*a*,*b*,*c*) //1 "22" true

对任意深度的嵌套数组进行解构

var [*foo*, [[*bar*], *baz*]] = [1, [[2], 3]];  
console.log(*foo*);  
// 1  
console.log(*bar*);  
// 2  
console.log(*baz*);  
// 3

留空来跳过被解构数组中的某些元素

var [,,*third*] = ["foo", "bar", "baz"];  
console.log(*third*);

// "baz"

捕获数组中的所有尾随元素

var [*head*, ...*tail*] = [1, 2, 3, 4];  
console.log(*head*);  
//1  
console.log(*tail*);  
// [2, 3, 4]

当访问空数组或越界访问数组时，对其解构与对其索引的行为一致，最终得到的结果都是：undefined

console.log([][0]);  
// undefined  
var [*missing*] = [];  
console.log(*missing*);  
// undefined

解构实现数组的合并：

let *a* = [1,2,3];  
console.log(...*a*);  
//1 2 3  
let *b* = ['a','b','c'];  
let *c* = [...*a*,...*b*];  
//[1, 2, 3, "a", "b", "c"]  
let *d* = [*a*,*b*];  
console.log(*d*)  
//[[1, 2, 3], ["a", "b", "c"]]

### 对象的解构赋值：

这种解构，前面的第一个参数是属性，必须和对象里的属性一样，

后面是参数，可随意写，赋值上对应的属性值（注意，这里的name1和name2不是变量，如果直接打印就会报错）

var *robotA* = { name1: "Bender" };  
var *robotB* = { name2: "Flexo" };  
var { name1: first } = *robotA*;  
var { name2: secode } = *robotB*;  
console.log(first);// "Bender"  
console.log(secode);// "Flexo"

属性名与变量名一致：

这种解构，里面是变量，必须和对象里的属性一样，此时这个变量就被赋值了与它相同的属性对应的属性值

var { name1 } = *robotA*;  
var { name2 } = *robotB*;  
console.log(name1);// "Bender"  
console.log(name2);// "Flexo"

随意嵌套并进一步组合对象解构：

var *complicatedObj* = {  
 arrayProp: ["Zapp", { second: "Brannigan" }]  
};  
var { arrayProp: [*first*, { *second* }] } = *complicatedObj*;  
console.log(*first*);  
// "Zapp"  
console.log(*second*);  
// "Brannigan"

当你解构一个未定义的属性时，得到的值为undefined：

var { *missing* } = {};  
console.log(*missing*);  
// undefined

### 默认值

当你要解构的属性未定义时你可以提供一个默认值

var [*missing* = true] = ['111'];  
console.log(*missing*);//'111'

var [*missing* = true] = [];  
console.log(*missing*);// true  
  
var { message: *msg* = "Something went wrong" } = {};  
console.log(*msg*);// "Something went wrong"

var { message: msg = "Something went wrong" } = {message:'1234'};  
console.log(msg);//'1234'  
  
var { *x* = 3 } = {};  
console.log(*x*);// 3

var { x = 3 } = {x:'aaa'};  
console.log(x);// 'aaa'

## **JSON.stringify()和JSON.parse**

### JSON.stringify()

JSON.stringify()将值转换为相应的JSON格式：

JSON.stringify() 方法是将一个JavaScript值(对象或者数组)转换为一个 JSON字符串，如果指定了replacer是一个函数，则可以选择性的替换值，或者如果指定了replacer是一个数组，可选择性的仅包含数组指定的属性。

* 转换值如果有toJSON()方法，该方法定义什么值将被序列化。
* undefined、任意的函数以及 symbol 值，在序列化过程中会被忽略（出现在非数组对象的属性值中时）或者被转换成 null（出现在数组中时）。
* 函数、undefined被单独转换时，会返回undefined，如JSON.stringify(function(){}) or JSON.stringify(undefined).
* 所有以 symbol 为属性键的属性都会被完全忽略掉，即便 replacer 参数中强制指定包含了它们。
* NaN和Infinity格式的数值及null都会被当做null。

JSON.stringify({}); // '{}'

JSON.stringify(true); // 'true'

JSON.stringify("foo"); // '"foo"'

JSON.stringify([1, "false", false]); // '[1,"false",false]'

JSON.stringify({ x: 5 }); // '{"x":5}'

JSON.stringify({x: 5, y: 6}); // "{"x":5,"y":6}"

JSON.stringify([new Number(1), new String("false"), new Boolean(false)]); // '[1,"false",false]'

JSON.stringify({x: undefined, y: Object, z: Symbol("")}); // '{}'

JSON.stringify([undefined, Object, Symbol("")]); // '[null,null,null]'

JSON.stringify({[Symbol("foo")]: "foo"}); // '{}'

JSON.stringify({[Symbol.for("foo")]: "foo"}, [Symbol.for("foo")]);// '{}'

JSON.stringify(

{[Symbol.for("foo")]: "foo"},

function (k, v) {

if (typeof k === "symbol"){

return "a symbol";

}

});

// undefined

// 不可枚举的属性默认会被忽略：JSON.stringify(

Object.create(

null,

{

x: { value: 'x', enumerable: false },

y: { value: 'y', enumerable: true }

}

));

// "{"y":"y"}"

replacer参数:

replacer参数可以是一个函数或者一个数组。作为函数，它有两个参数，键(key)值(value)都会被序列化。

函数：

//若前面为对象：key是属性，value是对应的属性值  
//若前面为数组：key是索引，value是对应的元素

//以对象为例：  
//返回以下几种数据类型：(只能在条件筛选里面返回，最终一定要返回出value)  
// number:a的值替换为该数字  
// string:a的值替换为该字符串  
// Boolean:a的值替换为该布尔值  
// array:a的值替换为该数组  
// object:a的值替换为该对象  
// null:a的值替换为该null  
// undefined：删除a属性  
  
replacer=(key, value)=> {  
 console.log(key,value)  
 if (key === 'a') {  
 return '111';  
 }  
 return value;  
 // return undefined  
}

let *foo* = {  
 a: "aaa",  
 b: "bbb",  
 c: 45,  
 d: "ddd",  
 e: 7  
};  
let *data* = JSON.parse(JSON.stringify(*foo*, this.replacer))

数组：

如果replacer是一个数组，数组的值代表将被序列化成JSON字符串的属性名

let *foo* = {  
 a: "aaa",  
 b: "bbb",  
 c: 45,  
 d: "ddd",  
 e: 7  
};  
let *data* = JSON.parse(JSON.stringify(*foo*, ['a','b'])) //{a: "aaa", b: "bbb"} (只保留a和b)

### JSON.parse()

JSON.parse('{}'); // {}

JSON.parse('true'); // true

JSON.parse('"foo"'); // "foo"

JSON.parse('[1, 5, "false"]'); // [1, 5, "false"]

JSON.parse('null'); // null

JSON.parse('1'); // 1

## **有关true和false的运算**

### true和false的判断

空对象(object)：{}-**true**

空数组(array)：[]-true

空格字符串(string)：" "-true

除0之外的数字(number)：1、-1……-true

空字符串(string)：''、""-false

数字0(number):0、0.0、0.00-false

Null(object):null-false

NaN(number):NaN-false

Undefined(undefined):undefined-false

### 与或非

首先js 会将 && 和 || 和！两边的值按照上面的规则转成Boolean 类型，然后再进行逻辑关系运算

**或**：有一个值为true，整个运算就为true;

var a = "" || null || 3 || 4;

相当于：

var a = false || false || true || true;

短路：第一个false，要去找下一个，第二个同是false，继续往下找。第三个是ture返回，不再检查第四。

结果为true 则返回第一个true,即是3；

从左到右，找第一个为true的值，取该值，若没有true，则取最后一个false

**与**：

var b = 4 && 5 && null && 0;

相当于：

var b = true && true && false && false;

一个为假则为假。当找到第三个时为false，返回。

结果是false 则返回第一个false 即是null .

从左到右，找第一个为false的值，取该值，若没有false，则取最后一个true

应用：

检测是否定义，未定义则赋初始值

// 判断一个变量是否存在，如果没有就给他一个初始值

var attr = attr || "";

function Foo(option) {

var defaultValue = option || {};

}

Foo();

利用返回值的特性:

分析：首先判断第一个(arg == 1 && '类型1')，arg == 1为true，'类型1'为true，没有false，因此取最后一个true，所以这里等于'类型1'；

整个是一个或运算，第一个已经为true，就不需要再往下判断了，直接取第一个的值

function getType(arg) {

var num = (arg == 1 && '类型1') || (arg == 2 && '类型2') || 0;

return num;

}

console.log(getType(1)) //'类型1'

优化：

分析：传递进去的参数为对象的属性，在对象里面查找，有该属性，则num就等于该属性的属性值，没有，num就等于0

function getValue(arg) {

var num = {'5':1,'10':2,'12':3,'15':4}[arg] || 0;

return num;

}

console.log(getType(5)) //1

### **取值为空时**

数组里取不到值：undefined

let *text* = [];  
console.log(*text*[0]) //undefined

对象里取不到值:undefined

let *text* = {};

或：

let *text* = {a:undefined,b:1};

console.log(*text*.a) // undefined

数组与对象不能做比较：

console.log([] === []) //false

console.log({}=== {}) //false

console.log(undefined == null) // true

console.log(undefined === null)//false

### **如何判断对象object为空**

* 将json对象转化为json字符串，再判断该字符串是否为"{}"

let *text* = {};  
console.log(JSON.stringify(*text*) === '{}') //true

* 使用ES6的Object.keys()方法(返回值是对象中属性名组成的数组)

let *text* = {a:1,b:2};  
console.log(Object.keys(*text*)) // ["a", "b"]

### **判断一个数组是否含有某个值**

方法一：  arr.indexOf() 有:返回索引值，没有:-1

if (arr.indexOf(2) != -1){

console.log("数组含有2")

}else {

console.log("数组不含2")

}

  方法二：  for循环结合if判断

for (let i = 0;i < arr.length;i++){

if (arr[i] === 2){

console.log("数组含有2")

　　}

}

方法三： arr.find(callback)   遍历数组

arr.find(value => {

if (value === 2){

console.log("数组含有2")

　　}

})

方法四： arr.includes()   有:返回true，没有:返回false。ES6新方法。

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

if(arr.includes(2)){

console.log("数组中有2");

}else{

console.log("数组中无2");

}

## **Try catch**

<https://www.cnblogs.com/yangheng/p/6018224.html>

## **Map对象**

### 描述

**Map**对象保存键/值对，是键/值对的集合。任何值(对象或者[原始值](https://developer.mozilla.org/en-US/docs/Glossary/Primitive" \t "https://www.jianshu.com/p/_blank)) 都可以作为一个键或一个值。

**Object**结构提供了“字符串—值”的对应，**Map**结构提供了“值—值”的对应。

Object 和 Map 对象的对比：

相似点： 都允许你按键存取一个值、删除键、检测一个键是否绑定了值。

不同点：

1. 一个对象的键只能是字符串或者 Symbols，但一个 Map 的键可以是任意值，包括函数、对象、基本类型。
2. Map 中的键值是有序的，而添加到对象中的键则不是。因此，当对它进行遍历时，Map对象是按插入的顺序返回键值。
3. 通过 size 属性直接获取一个 Map 的键值对个数，而 Object 的键值对个数只能手动计算。
4. Map 是可迭代的，而 Object 的迭代需要先获取它的键数组然后再进行迭代。
5. Object 都有自己的原型，所以原型链上的键名有可能和对象上的键名产生冲突。虽然 ES5 开始可以用 map = Object.create(null) 来创建一个没有原型的对象，但是这种用法不太常见。
6. Map 在频繁增删键值对的场景下会有些性能优势。

Map实例：var myMap = new Map();

Let myMap = new Map();

myMap.set(‘0’,’foo’);

myMap.set(1,’bar’);

myMap.set({a:1},’baz’);

Console.log(myMap);

Let myMap1 = new map([[‘0’,’foo’],[1,’bar’],[{a:1},’baz’]])

Console.log(myMap);

上面两种表述是相同的

### 键的比较：

键的比较是基于 "SameValueZero" 算法：NaN 是与 NaN 相等的（虽然 NaN !== NaN），剩下所有其它的值是根据 === 运算符的结果判断是否相等。

在目前的ECMAScript规范中，-0和+0被认为是相等的

map对象的方法里可直接输入的key值数据类型有：null，undefined，NaN，number，string，Boolean（这几种数据类型都是可以用===来判断是否相等的（NaN特殊除外））

let *original* = new Map([[null, NaN]]);  
console.log(*original*.get(null))//NaN

let *original* = new Map([[undefined, NaN]]);  
console.log(*original*.get(undefined))//NaN

let *original* = new Map([[NaN, NaN]]);  
console.log(*original*.get(NaN))//NaN

。。。。。Number，string、bool

不可输入的key值类型：object，array（引用数据类型不能用===来判断，因此不能直接在方法里输入引入类型作为key值）

var *myMap* = new Map();  
*myMap*.set([], "111");  
*myMap*.set([1,2], "222");  
  
console.log(*myMap*.get([])) //undefined

以下是正确的取值方式：

方法里只能用变量来代替引用类型

var *myMap* = new Map();  
let *arr* = [{a:1}];  
let *arr1* = [1,2];  
*myMap*.set(*arr*, "111");  
*myMap*.set(*arr1*, "222");  
console.log(*myMap*.get(*arr*)) //’111’

console.log(*myMap*.get(*arr1*))//’222’

所有的 Map 对象实例都会继承 [Map.prototype](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/prototype" \o "Map.prototype 属性表示 Map构造函数的原型对象。)

属性：

**Map.prototype.constructor**

返回一个函数，它创建了实例的原型。默认是[Map](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Map" \o "此页面仍未被本地化, 期待您的翻译!)函数。

**[Map.prototype.size](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/size" \o "size 是可访问属性，用于返回 一个Map 对象的成员数量。)**

返回Map对象的键/值对的数量。

方法：

### **[Map.prototype.clear()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/clear" \o "clear()方法会移除Map对象中的所有元素。)**

clear()方法移除Map对象的所有键/值对 。

var myMap = new Map();

myMap.set("bar", "baz");

myMap.set(1, "foo");

myMap.size; // 2

myMap.has("bar"); // true

**myMap.clear()**;

myMap.size; // 0

myMap.has("bar") // false

### **[Map.prototype.delete(key)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/delete" \o " delete() 方法用于移除 Map 对象中指定的元素。)**

**delete()**方法:如果 Map 对象中存在该元素，则移除它并返回 true；否则如果该元素不存在则返回 false

var myMap = new Map();

myMap.set("bar", "foo");

**myMap.delete("bar")**; // 返回 true。成功地移除元素

myMap.has("bar"); // 返回 false。"bar" 元素将不再存在于 Map 实例中

### **[Map.prototype.entries()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/entries" \o "entries() ?方法返回一个新的包含 [key, value] ?对的 Iterator ?对象，返回的迭代器的迭代顺序与 Map 对象的插入顺序相同。)**

**entries()** 方法返回一个新的 Iterator 对象，它按插入顺序包含了Map对象中每个元素的 [key, value]**数组**。

var myMap = new Map();

myMap.set("0", "foo");

myMap.set(1, "bar");

var mapIter = myMap.entries();

console.log(mapIter.next().value); // ["0", "foo"]

console.log(mapIter.next().value); // [1, "bar"]

理解：这个方法根据map对象造就一个MapIterator对象。这个对象有两个键值对，然后每次根据next().value打印都只打印第一个键值对，执行后同时就删除第一个键值对，此时页面就只剩第二个键值对。再次执行next.value打印出来后，此时这个对象里就为空了。

但MapIterator里无论怎么变化，都跟原来的map对象没有关系，myMap是不会变的。

如果用for of来遍历MapIterator对象，遍历后就将该对象清空了。

for(let item of mapIter){

Console.log(item)

}

// ["0", "foo"]

// [1, "bar"]

**此方法一般用不上，因为如果要遍历出键值对，直接给map对象进行遍历即可,遍历出来的每一个键值对是数组格式**

**for (let item of myMap){**

**Console.log(item)**

**}**

**// ["0", "foo"]**

**// [1, "bar"]**

### **[Map.prototype.forEach(callbackFn[, thisArg])](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/forEach" \o "forEach() 方法将会以插入顺序对 Map 对象中的每一个键值对执行一次参数中提供的回调函数。)**

**forEach()** 方法将会以插入顺序对 Map 对象中的每一个键值对执行一次参数中提供的回调函数。

callback 函数有三个参数:

* value - 元素的值
* key - 元素的键
* Map - 当前正在被遍历的对象

let *map* = new Map([["foo", 3], ["bar", {}], ["baz", undefined]]);  
*map*.forEach((value, key, map)=>{  
 console.log(value, key, map)  
});

3 "foo" Map(3) {"foo" => 3, "bar" => {…}, "baz" => undefined}

{} "bar" Map(3) {"foo" => 3, "bar" => {…}, "baz" => undefined}

undefined "baz" Map(3) {"foo" => 3, "bar" => {…}, "baz" => undefined}

### **[Map.prototype.set(key, value)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/set" \o "set() 方法为 Map 对象添加或更新一个指定了键（key）和值（value）的（新）键值对。)**

设置Map对象中键的值。返回该Map对象。

var myMap = new Map();

// 将一个新元素添加到 Map 对象

myMap.set("bar", "foo");

myMap.set(1, "foobar");

// 在Map对象中更新某个元素的值

myMap.set("bar", "baz");

还可以链式使用：

**myMap.set('bar', 'foo')**

**.set(1, 'foobar')**

**.set(2, 'baz');**

### **[Map.prototype.get(key)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/get" \o "get() 方法返回某个 Map 对象中的一个指定元素。)**

**get()** 方法返回某个 Map 对象中的一个指定元素。如果不存在，则返回undefined。

var myMap = new Map();

myMap.set("bar", "foo");

**myMap.get("bar"); // 返回 "foo"**

myMap.get("baz"); // 返回 undefined

### **[Map.prototype.has(key)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/has" \o "方法has() 返回一个bool值，用来表明map 中是否存在指定元素.)**

方法**has()** 返回一个bool值，用来表明map 中是否存在指定元素。

var myMap = new Map();

myMap.set("bar", "foo");

**myMap.has("bar");**  // returns true

myMap.has("baz"); // returns false

### **[Map.prototype.keys()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/keys" \o "keys() 返回一个新的 Iterator 对象。它包含按照顺序插入 Map 对象中每个元素的key值。)**

返回一个新的 Iterator对象， 它按插入顺序包含了Map对象中每个元素的**键**。

var myMap = new Map();

myMap.set("0", "foo");

myMap.set(1, "bar");

myMap.set({}, "baz");

var mapIter = myMap.keys();//MapIterator {"0", 1, {…}}

console.log(mapIter.next().value); // "0"

console.log(mapIter.next().value); // 1

console.log(mapIter.next().value); // {}

同上的MapIterator对象，该对象是包含key值的对象，也是执行一个删除一个。

可直接用for of遍历MapIterator对象，就可一次性遍历出所有的key值；（遍历后MapIterator对象为空对象）

For(let it of mapIter){

Console.log(it)

}

// "0" // 1 // {}

或用for of直接遍历map对象，遍历出数组，取每个数组的第一个值

或用forEach直接遍历map对象，获取所有的key；

### **[Map.prototype.values()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/values" \o "一个新的 Map 可迭代对象.)**

**values()** 方法返回一个新的Iterator对象，它按插入顺序包含了Map对象中每个元素的**值** 。

var myMap = new Map();

myMap.set("0", "foo");

myMap.set(1, "bar");

myMap.set({}, "baz");

var mapIter = myMap.values();//MapIterator {"foo", "bar", "baz"}

console.log(mapIter.next().value); // "foo"

console.log(mapIter.next().value); // "bar"

console.log(mapIter.next().value); // "baz"

同上的MapIterator对象，该对象是包含value值的对象，也是执行一个删除一个。

可直接用for of遍历MapIterator对象，就可一次性遍历出所有的value值；（遍历后MapIterator对象为空对象）

For(let it of mapIter){

Console.log(it)

}

// "0" // 1 // {}

或用for of直接遍历map对象，遍历出数组，取每个数组的第二个值

或用forEach直接遍历map对象，获取所有的key；

### **[Map.prototype[@@iterator]()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map/@@iterator" \o "@@iterator 属性的初始值与 entries 属性的初始值是同一个函数对象。)**

返回一个新的Iterator对象，它按插入顺序包含了Map对象中每个元素的 [key, value]**数组**。

### 将NaN作为Map的键

NaN 也可以作为Map对象的键。虽然 NaN 和任何值甚至和自己都不相等(NaN !== NaN 返回true)，但下面的例子表明，NaN作为Map的键来说是没有区别的:

var myMap = new Map();

myMap.set(NaN, "not a number");

myMap.get(NaN); // "not a number"

var otherNaN = Number("foo");

myMap.get(otherNaN); // "not a number"

### 迭代map对象-for of

Map可以使用for..of循环来实现迭代：

let *myMap* = new Map();  
*myMap*.set('0', 'foo');  
*myMap*.set(1, 'bar');  
*myMap*.set({}, 'baz');  
  
for (let *entry* of *myMap*) {  
 console.log(*entry*);  
}

// ["0", "foo"]

// [1, "bar"]

// [{}, "b

var myMap = new Map();

myMap.set(0, "zero");

myMap.set(1, "one");

for (var [key, value] of myMap) {

console.log(key + " = " + value);}

// 0 = zero

// 1 = one

for (var key of myMap.keys()) {

console.log(key);}

// 0

// 1

for (var value of myMap.values()) {

console.log(value);}

// zero

// one

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

console.log(key + " = " + value);}

// 0 = zero

// 1 = one

### [Map 与数组的关系:](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[var kvArray = [["key1", "value1"], ["key2", "value2"]];](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[// 使用常规的Map构造函数可以将一个二维键值对数组转换成一个Map对象](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[var myMap = new Map(kvArray);](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[myMap.get("key1"); // 返回值为 "value1"](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[// 使用Array.from函数可以将一个Map对象转换成一个二维键值对数组](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[console.log(Array.from(myMap)); // 输出[["key1", "value1"], ["key2", "value2"]]](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[// 或者在键或者值的迭代器上使用Array.from，进而得到只含有键或者值的数组](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[console.log(Array.from(myMap.keys())); // 输出 ["key1", "key2"]](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[console.log(Array.from(myMap.values())); // 输出 ["value1", "value2"]](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[console.log(Array.from(myMap.values())); // 输出[["key1", "value1"], ["key2", "value2"]]](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

### [复制或合并 Maps:](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "%E5%A4%8D%E5%88%B6%E6%88%96%E5%90%88%E5%B9%B6_Maps)

[Map 能像数组一样被复制：](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "%E5%A4%8D%E5%88%B6%E6%88%96%E5%90%88%E5%B9%B6_Maps)

[var original = new Map([[1, 'one']]);](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "%E5%A4%8D%E5%88%B6%E6%88%96%E5%90%88%E5%B9%B6_Maps)

[var clone = new Map(original);](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "%E5%A4%8D%E5%88%B6%E6%88%96%E5%90%88%E5%B9%B6_Maps)

[console.log(original ); // Map(1) {1 => "one"}](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "%E5%A4%8D%E5%88%B6%E6%88%96%E5%90%88%E5%B9%B6_Maps)

[console.log(clone); // Map(1) {1 => "one"}](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "%E5%A4%8D%E5%88%B6%E6%88%96%E5%90%88%E5%B9%B6_Maps)

[console.log(original === clone); // false](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "%E5%A4%8D%E5%88%B6%E6%88%96%E5%90%88%E5%B9%B6_Maps)

[Map对象间可以进行合并，但是会保持键的唯一性](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "%E5%A4%8D%E5%88%B6%E6%88%96%E5%90%88%E5%B9%B6_Maps)

[var first = new Map([](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[[1, 'one'],](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[[2, 'two'],](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[[3, 'three'],]);](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[var second = new Map([](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[[1, 'uno'],](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[[2, 'dos']]);](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[// 合并两个Map对象时，如果有重复的键值，则后面的会覆盖前面的。// 展开运算符本质上是将Map对象转换成数组。var merged = new Map([...first, ...second]);](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[console.log(merged.get(1)); // uno](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[console.log(merged.get(2)); // dos](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[console.log(merged.get(3)); // three](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[Map对象也能与数组合并](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[var first = new Map([](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[[1, 'one'],](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[[2, 'two'],](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[[3, 'three'],]);](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[var second = new Map([](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[[1, 'uno'],](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[[2, 'dos']]);](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[// Map对象同数组进行合并时，如果有重复的键值，则后面的会覆盖前面的。var merged = new Map([...first, ...second, [1, 'eins']]);](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[console.log(merged.get(1)); // eins](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[console.log(merged.get(2)); // dos](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

[console.log(merged.get(3)); // three](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Map" \l "Map_%E4%B8%8E%E6%95%B0%E7%BB%84%E7%9A%84%E5%85%B3%E7%B3%BB)

## 分页的序号：

用于点击每页都重新请求当页数据的情况：

startNum 为起始编号，pageNumber 为第几页，在循环渲染列表中，序号就可以为num,因为每一次循环都+1

以每页20条为例：

let startNum = (res.pageNumber - 1) \* 20;

let num = startNum;

for (let i in res.rows) {

num+=1;

若一次性把所有数据都请求回来了，只需要在数据里添加页码属性。

## 更改状态后的回调：

状态更改后---逗号---箭头函数，调用函数，且将刚更改的状态作为参数传递

this.setState({

currentPage:current

},()=>{

this.getTable(true);

}

)

注意：当回调函数里有用到currentPage时，则不需要这样写，直接将current传输给函数即可：

this.setState({

currentPage:current

}

this.getTable(current);

)

若函数里不能用传参，则：

this.setState({

currentPage:current

},()=>{

this.getTable();

}

)

应用：

方法setState执行完后，执行onCheckData,传递参数,使其为calkData的第二个参数；

onCheckData里调用calkData，设置calkData的第一个参数；

在calkData方法里，就可以根据calkData的两个参数来区分执行的是筛选还是刷新。



## 子组件与父组件之间的参数传递

各种方法调用的时候，都可传递参数

### 子组件调用父组件方法，并传参给父组件

getDates为父组件的方法，传递给子组件后，子组件可以这样调用并传参，父组件的这个方法里就可以接收这个参数；

子组件里：

onClick={()=>**this**.submitData(**this**.props.n)}

点击函数不用写括号，但这样的话，就没办法传参，因此用箭头函数，在箭头函数里再执行父组件传递下来的函数，这时是需要加括号的，就实现传参。

或者：

submitData(){

**this**.submitData(**this**.props.n) //传参给父组件

**let** data = **this**.props.getDates(**this**.props.n);//传参给父组件，并获取到父组件getDates方法里return出的数据

**}**

### 子组件调用父组件方法后，父组件方法里再调用子组件方法

当子组件调用父组件的时候，若父组件需要进行一段条件判断或接口请求操作后才需要执行子组件里的另一个方法，就可以直接先在子组件调用的时候，将this.onChangeEditStatus为editValue的参数传递给父组件。然后再父组件的editValue里再执行它

子组件里：

onClick={()=>*editValue*(storeContent, this.onChangeEditStatus)}

父组件里：

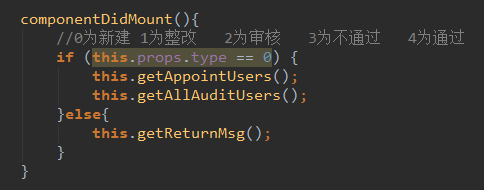
editValue=(obj, callback)=>{

。。。。。。

}

## 页面初始数据的获取：

刚进入页面要请求各种数据，可以在componentDidMount里面，render渲染之后。



例子---接口请求：

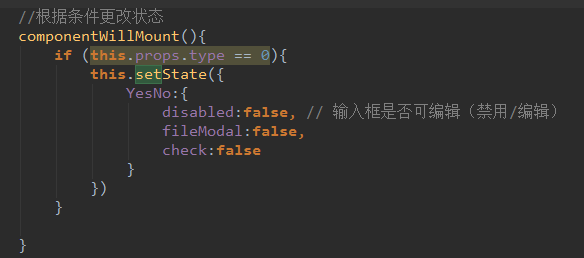
getAllAuditUsers = () => {  
 **let** data = {  
 PrjID:**this**.props.prjGuid,  
 action: "MyMission",  
 op: "allAuditUsers"  
 };  
 // console.log(data);  
 fetch('/servlet/WebReactRequestServlet', {  
 method: 'post',  
 credentials: 'include',  
 headers: {  
 "Content-Type": "application/raw"  
 },  
 body: JSON.stringify(data)  
 }).then(res => {  
 **if** (res.ok) {  
 **return** res.json()  
 .then(res => {  
 console.log(res);  
 **this**.setState({  
 // treeData: res.rows,  
 modalArr: res.modalMap,  
 allAuditUsers: res.allAuditUsers  
 });  
 }  
 );  
 }  
 }).catch(e => {  
 message.error("信息查询失败");  
 })  
};

接口例子----数据为拼接：

getReturnMsg(){  
 **const** url = `/checkUp/postQuerySingleParticulars?guid=` + **this**.props.guid;  
 fetch(url,{  
 method: 'post',  
 credentials: 'include',  
 headers: {  
 "Content-Type": "application/raw"  
 }  
 }).then(res => {  
 **if** (res.ok) {  
 **return** res.json()  
 .then(res => {  
 // console.log(res);  
 **this**.setState({  
 data:res  
 })  
 }  
 });  
 }  
}).catch(e => {  
 message.error('信息查询失败');  
 })  
}

## 若this.props影响初次渲染组件

若初次渲染组件就要根据其他组件传递下来的props进行渲染，可在componentWillMount方法里进行props处理，根据props更改一些组件的状态，再渲染组件。



若不能确定，父组件的参数是在子组件渲染的什么时候传下来，（可能是渲染时一起传下来，也可能在渲染之后才传下来），此时就必须在以下两个钩子函数里都要进行对此参数的接收。总有一个能接收到。

componentWillMount(){  
 **if**(**this**.props.rectifyContent){//不为null  
 **this**.setState({  
 rectifyContent:**this**.props.rectifyContent  
 })  
 }  
}  
componentWillReceiveProps(props){  
 **if**(props.rectifyContent){  
 **this**.setState({  
 rectifyContent:props.rectifyContent  
 })  
 }  
}

使用输入框等组件--

通常使用各种输入框组件时，会有个例如onChange的方法，里面写回调函数，例如这里的updateAuditUsers ，在这种函数里，通常第一个参数就是用户输入的值，此时就可以获取到输入框的值，添加到状态上，再将此状态赋值给输入框的value属性，输入框的值才会跟着变化，这样当清空的时候，输入框的值也能被清空。

注意：这里使用value的时候必须配合onChange一起使用，不然会报错，且不能用defaultValue，因为defaultValue只是初始的时候渲染一次，尽管输入框的内容改变了，onChange变化了，也添加到状态上了，但输入框显示的值永远是初始值，不会随着状态的变化而变化，所以必须用value，输入框才会实时显示改变的内容。

updateAuditUsers = (auditUsers) => {  
 **this**.setState({  
 auditUsers: auditUsers,  
 });  
};

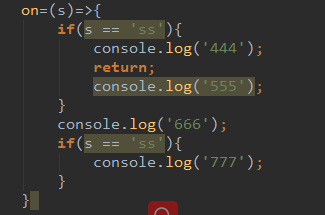
## 10.this指向

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 无（）：this.on | 加（）:this.on() |
| 普通函数： | 代码 |  |  |
| 执行 |  |  |
| 效果 | 点击执行，但函数里没有this | 刷新页面立即执行，不等点击，有this |
| 函数：箭头函数 | 代码 |  |  |
| 执行 |  |  |
| 效果 | 点击执行，有this | 刷新页面立即执行，有this |
| 调用：箭头函数 | 代码 |  |  |
| 执行 | 不带括号，点击不执行 |  |
| 效果 | 不执行 | 只要调用用箭头函数，返回函数，且加括号，无论此函数定义时是普通函数，还是箭头函数，都正常执行 |
| 结论 | | 1. 函数定义时用箭头函数，调用：this.on或（）=>this.on(),都可正常执行，且有this 2. 函数调用时用箭头函数（必须加括号），无论定义时是on()或on=()=>{}，都可正常执行，且有this | |

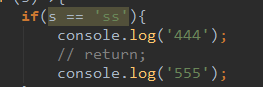
## 11.Return

reture会阻断代码的执行，return语句将终止当前函数并返回当前函数的值；

这里只打印出‘444’，下面的代码均被return阻断



如果被return去掉，就不影响代码的执行



结论：一个函数中，只要中途有了return满足条件执行了，下面的代码就不会执行了。

return只会阻断当前函数，如果on函数被其他地方调用，就算执行了return false；在其他地方也只是一个返回值为false，不会影响代码执行。

## 新增与修改页面，表单的逻辑

**新增：**

默认状态添加textValue值为undefined或为空，将此状态添加到表单的value属性上；

* 表单里输入数据时，一般表单的属性函数又onChange回调函数，第一个参数就为输入的内容。可以获取里面的值，然后setState修改textValue，在清空时，将textValue再设置回undefined或空；
* 提交时，要判断各个输入框是否为空，可判断状态里是否有值，没有则弹出各种提示，代码要用return阻断，直到所有都有值，才能进行下一步操作；
* 下一步操作为从状态里取值，请求接口提交数据；
* 注意为了能够回显编辑，这里最好把能显示内容的所有信息都提交到数据库，这样回显时才能根据数据显示出内容

**编辑：**

* 默认状态添加textValue值为undefined或为空，将此状态添加到表单的value属性上；
* 写接口获取回显数据，根据回显数据 更改textValue,把数据都显示到页面上；（只在初次渲染页面时候回显数据，此接口只请求一次）
* 若有修改的地方，在表单的oncChange回调函数里，通过setState修改对应的textValue状态，在清空时，将textValue设置回undefined或空；
* 提交时，从状态里取值，无论修改是否，都有值，如果清空了没有值，提交这里会判断，然后请求接口提交数据
* 注意，这里从状态里取值，若有修改，则是修改后的值，若无修改，则是回显的数据。

## 找错

找错从数据源开始找，控制台的network

componentDidMount(){  
 **this**.init();  
}  
componentWillReceiveProps(){  
 **this**.init();  
}

//接收回显数据的富文本信息，将其转成对象写到状态上，如果状态上已经有文本了，就不需要再把接收的写到状态上了

init = ()=>{  
 //如果不是读取，则初始化。  
 **this**.focus();  
 setTimeout(()=>{  
 **this**.store()  
 },1000)  
};

//使编辑器获得焦点

focus = () => {  
 **this**.editor.focus()  
};

//接收到富文本的值，将其和原来的值为对比，将最新的值传输到父组件textbox里面，  
store = (content)=>{  
 **let** contentState = content?content:**this**.state.contentState;  
 **this**.props.storeContent(JSON.stringify(contentState));  
};

//输入时获取到内容，添加到状态上，并执行store将内容上传到父组件

handleChange = (content) => {  
 **this**.setState({  
 contentState:content  
 });  
 **this**.store(content);  
};

//获取图片的地址和名称，将其上传

uploadFn = (param) => {  
 };

//将状态里的富文本内容里的路径转换为真实路径

infoGet = ()=>{  
 **if**(trueUrl){  
 **this**.imagesInfosChange();  
 }  
 **return** {contentState}  
};

imagesInfosChange(content){  
 }

## 组件属性用对象传递

**const** textBox = {  
 type:type,  
 ref:textbox=>**this**.textbox = textbox,  
 questionContent:**this**.state.questionContent  
};

<Textbox text={'整改回复'} {...textBox} />

## 宽度自适应

要用宽度自适应页面宽度，可设置宽度为父元素的百分比，并设置最小宽度；

同时，父元素一定要设置最小宽度，在子元素最小宽度之上，不然子元素的宽度小到最小宽度后，停止缩小，父元素还一直在缩小，子元素就会下榻；父元素设置了最小宽度后，刚才的情况就会使页面出现滚动条。

## 高度自适应

（1）弹性盒

当设置了弹性盒后，主轴为纵轴，若子元素1设置了height:50px;子元素2设置了flex:1,正常情况下，子元素2默认填充剩余高度；

但如果子元素2设置了最小高度min-height,子元素1的高度就变成0了;此时的flex以最小高度为准，所以子元素1也要设置最小高度跟高度一样才有效。这样后，只有页面低于两个子元素的最小高度纸之和就会·出现滚动条。

**让子元素2填充剩余高度：**

无论子元素1设不设置高度，只要将子元素：height:100%;或flex:1;子元素2都能填充除去子元素1的剩余高度。

（2）高度百分比

<div class=’one’ style=”height:100%”>

<div class=’two’>

<p class=’three’ style=”height:100%”></p>

</div>

</div>

高度的百分比继承上一级父元素，这里one设置了高度（无论为百分比和像素），但是two没有设置高度（默认由子元素撑开），three设置高度100%是没有作用的。所以要想three的高度有作用，two就必须设高度（无论为百分比和像素）。

计算：  
div{ height: calc(~"100%-100px") }

1. 屏幕缩小后留白

<div class=‘p’ style=‘height:100%;’>

<p class=‘p1’></p>

<p class=‘p2’></p>

</div>

上面p宽和高默认为100%,就是没有固定的高和宽，跟随页面大小变化。

为了避免页面宽度缩小后，p1和p2会变形，p1和p2要么固定了宽度，要么设置了最小宽度，p1和p2的宽度加起来的和就是p的宽度，当页面缩小到小于p的宽度时，页面就会出现滚动条；此时，因为p的宽度为默认的100%，此时p的100%宽度已经小于p的子元素宽度的和，此时页面就只显示页面的100%宽度的内容，滚动条往右拉，100%宽度之外的内容是空白的。（说明子元素的宽度之和：让页面出现了滚动条，使之不变形，但并没有完全显示在页面上；页面100%宽度：这个才是显示内容的宽度，这个宽度为多少，就显示多少内容，其他为空。）

解决：

給p设置最小宽度>=p1、p2宽度之和，这样页面出现滚动条的宽度就会以p的最小宽度为准，当页面100%小于min-width时，页面显示内容的宽度由页面100%变为min-width，就能显示出全部的内容，不会有空白。

高度也是一样的：

为了避免页面高度缩小后，p1和p2会变形，p1和p2要么有高度（无论设置还是由子元素撑开），要么设置了最小高度，p1和p2的高度加起来的和就是p的内容高度，当页面缩小到小于p的内容高度时，页面就会出现滚动条；此时，因为p的高度为默认的100%，此时p的100%高度已经小于p的内容高度，此时就显示页面的100%高度的内容，滚动条往下拉，100%高度之外的内容是空白的。（说明子元素的高度之和：让页面出现了滚动条，使之不变形，但并没有完全显示在页面上；页面100%高度：这个才是显示内容的高度，这个高度为多少，就显示多少内容，其他为空。）

解决：給p设置最小高度>=p1、p2高度之和，这样页面出现滚动条的高度就会以p的最小高度为准，当页面100%小于min-width时，页面显示内容的高度由页面100%变为min-width，就能显示出全部的内容，不会有空白。

## 接口

### 项目封装httpFetch

传一个对象，在接口上拼接对象的属性值

*/\*\*  
 \* 获取用户权限  
 \** ***@param*** *guid 项目id  
 \** ***@returns*** *{\*}  
 \*/*getRight({guid}) {  
 **return** httpFetch("post", `/checkUp/postIsTrueOrFalse?guid=${guid}`);  
},

**import** api **from** 'src/App.api.js';

api.checkUp.getRight({guid:record.guid}).then(res => {

console.log(res);  
 }).catch(e => {  
 message.error('信息查询失败' + e);  
});

传一个对象，

*/\*\*  
 \* 获取列表页数据  
 \** ***@data*** *object 查找数据的条件  
 \*/*getTable(data){  
 **return** httpFetch("post", `/checkUp/postQuerySafetyList`,data);  
},

**import** api **from** 'src/App.api.js';

**let** datas={  
 pageNumber:**this**.state.currentPage,  
 pageSize:**this**.state.pageSize,  
 projectGuid:**this**.props.projectObj.value  
}

api.checkUp.getTable(datas).then(res=>{

console.log(res);  
 }).catch(e => {  
 message.error('信息查询失败' + e);  
});

或者：

getAppointUsers(params={PrjID,action,op}){  
 **return** httpFetch("post", `/servlet/WebReactRequestServlet`,params);  
},

**import** api **from** 'src/App.api.js';

**let** data = {  
 PrjID:**this**.props.prjGuid,  
 action: "MyMission",  
 op: "getAppointUsers"  
};  
api.checkUp.getAppointUsers(data).then(res=>{

console.log(res);  
 });

### 表单请求：

let *param* = {  
 columnName: 'summaryText',  
 PrjId: *guid*,  
 value: obj,  
 organizationType: *tabKey*};  
fetch('/servlet/WebPrjServlet?action=project&op=resetZKGLGH',{  
 method:'post',  
 credentials: 'include',  
 headers: {  
 'Content-Type': 'application/x-www-form-urlencoded'  
 },  
 body:JSON.stringify(*param*)  
})  
 .then(res=>{  
 if(res.ok){  
 return res.json()  
 .then(res=>{  
 if(res.op==='suc'){  
 *message*.success('编辑成功');  
 this.props.informationQuery();  
 callback()  
 }else{  
 *message*.error('保存编辑内容错误： ' + res.msg)  
 }  
 });  
 }else{  
 *message*.error('保存编辑内容错误,错误代码： ' + res.status);  
 this.setState({  
 logining:false,  
 });  
 }  
 })  
 .catch(err=>{  
 this.setState({  
 logining:false  
 });  
 })

## 日期转换

将moment格式的时间转换成字符串YYYY-MM-DD格式：

**import** moment **from** 'moment';

moment(value).format('YYYY-MM-DD');

将字符串YYYY-MM-DD的时间转换成moment格式：可用于给DatePicker组件回显

moment(**this**.state.applyTime, "YYYY-MM-DD")

将字符串毫秒转换成moment格式：可用于给DatePicker组件回显

moment(Number(**this**.state.declareTime))

字符串转成毫秒：

**let** time = moment('2018-01-01').valueOf();

moment格式转换成毫秒：

moment(text1).valueOf()

moment格式转换成秒：

moment(text1).unix()

毫秒转换成字符串

moment(443646365).format('YYYY-MM-DD')

moment(Number(text)).format('YYYY-MM-DD')

秒转换成字符串

moment(Number(text)\*1000).format('YYYY-MM-DD')

获取当前时间，并转换成字符串格式：

Time: moment(**new** Date().getTime()).format('YYYY-MM-DD')

或：(**new** Date()).Format('yyyy-MM-dd')

指定日期之前的日期禁用：

18号之前禁用：

<DatePicker  
 format="YYYY-MM-DD"  
 disabledDate={(current)=>current && current< moment('2019-2-18').subtract(1, 'days').endOf('day')}  
/>

18之前（包括18号）禁用：

disabledDate={(current)=>current && current< moment('2019-2-18').endOf('day')}

今天之前禁用：

disabledDate={(current)=>current && current< moment().subtract(1, 'days').endOf('day')}

今天之前（包括今天）禁用：

disabledDate={(current)=>current && current< moment().endOf('day')}

## 获取sessionStorage值和cookie值：

**import** webapp **from** 'Res/js/webapp';

constructor(props) {  
 **super**(props);  
 **this**.state ={

LoginID: JSON.parse(sessionStorage.userData)['LoginID'],

}  
 **this**.cookies = webapp.cookieGetAll();  
}

console.log(unescape(**this**.cookies['ShortName']));

## 验证富文本：

富文本为空：

**if**(content === 'null') {  
 message.warn("富文本内容为空");  
 **return**;  
}

不为空，验证空格：

//验证富文本内容为否为空格  
richText = (tu,word)=>{  
 **let** a = '';  
 **if**(JSON.stringify(tu) === '{}'){  
 word.forEach(item=>{  
 //用正则验证：除空格外的字符  
 **let** reg = /([^\s])/g;  
 **let** arr = item.text.match(reg);  
 **if**(arr !== **null**){//提取出有内容的text  
 **if**(arr.length > 0){  
 a = 1;  
 }  
 }  
 });  
 **return** a;  
 }**else**{  
 a = 1;  
 **return** a;  
 }  
}

//验证富文本的空格  
**let** obj = JSON.parse(content);  
**let** word = obj.blocks;  
**let** tu = obj.entityMap;  
**let** fan = **this**.richText(tu,word);  
**if**(fan !== 1){  
 message.warn("请输入有效的富文本内容");  
 **return**;  
}

## async...await

//async...await:将异步转变成同步，首先在 function 前面加 async 用来说明这个函数是一个异步函数，  
// 当然，async是要和await配合使用的，await 意思是等一下，等着getBase64（）这个函数执行完毕，得到值后再赋值给file.preview，  
// 然后再用file.preview的值进行下一步操作，就是这么简单！  
handlePreview = **async** file => {  
 **if** (!file.url && !file.preview) {  
 file.preview = **await** getBase64(file.originFileObj);  
 }  
  
 **this**.setState({  
 previewImage: file.url || file.preview,  
 previewVisible: **true**,  
 });  
};

# Ant design方法

## 回显数据

（1）DatePicker日期回显

getInput=(value,option,state)=>{  
 **this**.setState({  
 appTime:option  
 });

<DatePicker  
 locale={zh\_CN}  
 placeholder={'请选择日期'}  
 format="YYYY-MM-DD"  
 onChange={(date,dateString)=>**this**.getInput(date,dateString,3)}  
 value={**this**.state.appTime.length === 0? **null**:moment(**this**.state.appTime, "YYYY-MM-DD")}  
  
/>

（2）Select选择框回显

getInput=(value,option,state)=>{  
 **let** Obj1 = [];  
 Obj1.push({'name': option.props.children, 'guid': option.key});  
 **this**.setState({  
 proPoserObj: Obj1,  
 proPoser: value//回显数据  
 });

<Select  
 getPopupContainer={triggerNode => triggerNode.parentNode}  
 placeholder="请选择申请人"  
 onChange={(value,option)=>**this**.getInput(value,option,1)}  
 onDeselect={(value,option)=>**this**.onDeselectIn(value,option,1)}  
 value={**this**.state.proPoser}  
>  
 {**this**.props.children}  
</Select>

getPopupContainer为修改下拉区域的相对谁进行的定位

（3）input组件回显

Input组件获取输入的值：

<Input  
 placeholder="请输入关键字"  
 onChange={**this**.MaterialSelect}  
 style={{ width:310,height:28}}

value={**this**.state.materialSelec}  
 allowClear  
/>

//物资列表查询--文本获取  
MaterialSelect=(value)=>{  
 **let** text = value.target.value;  
 **this**.setState({  
 materialSelect:text  
 })  
}

1. 表格图片回显

render:(text)=>(  
 <div className={style.imgBox}>  
 {JSON.parse(text) == ''?'暂无图片':  
 <img  
 src={JSON.parse(text)[0].url}  
 alt=""  
 className={style.img}  
 />}  
 </div>  
)

res.auditUsers为数组元素为对象格式，下面代码可以直接得到数组遍历后取特定的值组成的新数组

**const** auditUsers = res.auditUsers.map(i=>i.name)

要写选择下拉框，获取下拉数据后，可以通过以下方式遍历添加选项

**for** (**let** i = 0; i < userList.length; i++) {  
 children.push(<Option key={userList[i].userguid}>{userList[i].username}</Option>);  
}

## Modal组件

Modal组件，一旦使用，代码位置都在该页面最根部代码的下面显示，标题固定，但身体可以另外加载组件，例如下面的div>AddUpPurchase,然后就单独在AddUpPurchase里面写结构、功能、样式。

<Modal  
 style={{top:50,paddingBottom:0,overflow:'hidden',minWidth:1210}}  
 title={'新增采购申请'}  
 visible={**this**.state.addPurchase}  
 width={'100%'}  
 bodyStyle={{padding:0,width:"100%"}}  
 onCancel={**this**.addPurchaseClose}  
 destroyOnClose={**true**}  
 footer={**null**}  
 wrapClassName={'applyFor'}  
>  
 <div style={{height:'100%'}}>  
 <AddUpPurchase  
 PrjID={**this**.props.PrjID}  
 onCancel={**this**.addPurchaseClose}  
 type={**this**.props.type}  
 guid={**this**.state.guid}  
 children={**this**.state.children}  
 />  
 </div>  
</Modal>

但标题的样式只能在最根部的js里改，这里可以加一个类名，此时的类名跟ant的类名类似，因此通过global修改，同时也区分不同modal组件，例如：

:global{  
 .applyFor{  
 .ant-modal-content{  
 .ant-modal-header{  
 background: #2A2C32;  
 text-align:center;  
 .ant-modal-title{  
 color:#fff;  
 }  
 }  
 .ant-modal-close{  
 color:#fff;  
 }  
 }  
 }  
}

### modal占页面的100%：

constructor(){  
 **super**();  
 **this**.state = {  
 visibleModal:**false** }  
}  
open = ()=>{  
 **this**.setState({  
 visibleModal:**true** })  
};  
onCancel = ()=>{  
 **this**.setState({  
 visibleModal:**false** })  
};  
//style和width的样式，都是给ant-modal添加样式，因此也可以给ant-modal里加这些样式;top最好在style离加，因为sameStyle是公共样式。  
//bodyStyle的样式是给ant-modal-body加样式，因此这里面的样式也可以给ant-modal-body加样式；  
//这里的top，是整个modal是与顶部的距离；  
//sameStyle是modal最顶层，宽高均为页面100%，包含阴影部分，第二层为ant-modal，为内容分部，不包含阴影；  
//要让内容部分百分百于页面，先让ant-modal继承上级高度，若ant-modal设置了top则要减去top的高度；  
//然后ant-modal-content继承ant-modal百分百，ant-modal-content下面是为兄弟关系的×，Hender，body，header设置了高度后，在body里要100%减去hender的高度，再给body设置背景色为#fff,此时就实现了modal占据整个页面。  
//若该页面是可超过100%的，则ant-modal-body的高度就用min-height  
//Modal组件里的元素或组件，都是在ant-modal-body里的，如果是调用另一个外部组件，则样式写在那个组件里即可，若只是简单的html，则样式可直接用'类名'，而不用style.类名，这样就可以写在ant-modal-body里。  
//sameStyle相当于公共样式，给其他需要100%与页面的加这个类名即可，然后再加一个类名，写属于该modal自己的样式。  
//maskClosable={false}:点击阴影层不会关闭modal  
//destroyOnClose：关闭modal时卸载里面的元素，这样就可以显示每次打开modal都重新加载modal里面所有的元素  
//footer={null}：不要底部按钮  
render() {  
 **return** (  
 <div className='box'>  
 <Button onClick={**this**.open}>打开</Button>  
 <Modal  
 title={"变更单关联模型"}  
 visible={**this**.state.visibleModal}  
 width={'100%'}  
 style={{top:30,minWidth:500}}  
 bodyStyle={{}}  
 onCancel={**this**.onCancel}  
 maskClosable={**false**}  
 destroyOnClose  
 footer={**null**}  
 wrapClassName={'RelatedModal sameStyle'}  
 >  
 这是内容部分  
 </Modal>  
 </div>  
 );  
}

:global{  
 .sameStyle{  
 .ant-modal{  
 padding-bottom:0;  
 height:calc(100% - 30px);  
 .ant-modal-content{  
 height:100%;  
 border-radius: 0;  
 .ant-modal-close{  
 color:#fff;  
 .ant-modal-close-x{  
 height: 30px;  
 width:30px;  
 line-height: 30px;  
 }  
 }  
 .ant-modal-header{  
 padding:4px 0;  
 height: 30px;  
 line-height:30px;  
 background: #2A99E2;  
 text-align:center;  
 border-radius: 0;  
 font-size: 14px;  
 border-bottom:0;  
 .ant-modal-title{  
 color:#fff;  
 }  
 }  
 .ant-modal-body{  
 min-height:calc(100% - 30px);  
 padding: 0;  
 background-color: #61dafb;  
 }  
 }  
 }  
 }  
}

### Modal在页面居中的小弹窗：

<Modal  
 title="新增变更标注"  
 visible={**this**.state.visibleAddLable}  
 onCancel={**this**.closeAddLable}  
 maskClosable={**false**}  
 footer={**null**}  
 centered  
 width={446}  
 style={{height:265}}  
 wrapClassName={'AddLable sameStyle'}  
>  
 <div className={"content"}></div>  
 <div className={"btn"}>  
 <Button>确定</Button>  
 <Button>取消</Button>  
 </div>  
</Modal>

//当上面已经设置了width和height后，就只需要在ant-modal里再设置背景色就可以，上面style里的height就是这里ant-modal的高。然后在ant-modal-body里写内容的样式。注意：这里如果内容是写在本页面的，则类名用“”，不用style，就可以直接写在ant-modal-body里，如果是调用了其他页面，则在其他页面用style写样式。

:global{  
 .AddLable{  
 .ant-modal{  
 background-color: #fff;  
 .ant-modal-body{  
 }  
 }  
 }  
 }  
}

:global{  
 .sameStyle{  
 .ant-modal{  
 padding-bottom:0;  
 .ant-modal-content{  
 border-radius: 0;

height: 100%;  
 .ant-modal-header{  
 background: #2A99E2;  
 text-align:center;  
 border-radius: 0;  
 font-size: 14px;  
 border-bottom:0;  
 .ant-modal-title{  
 color:#fff;  
 }  
 }  
 .ant-modal-close{  
 color:#fff;  
 }  
 .ant-modal-body{  
 padding: 0;  
 }  
 }  
 }  
 }  
}

**class** one **extends** React.Component {  
 constructor(){  
 **super**();  
 **this**.state = {  
 visibleModal:**true** }  
 }  
 open = ()=>{  
 **this**.setState({  
 visibleModal:**true** })  
 };  
 onCancel = ()=>{  
 **this**.setState({  
 visibleModal:**false** })  
 };  
 //style和width的样式，都是给ant-modal添加样式，因此也可以给ant-modal里加这些样式;top最好在style里加，因为sameStyle是公共样式。  
 //bodyStyle的样式是给ant-modal-body加样式，因此这里面的样式也可以给ant-modal-body加样式；  
 //这里的top，是整个modal是与顶部的距离；  
 //sameStyle是modal最顶层，宽高均为页面100%，包含阴影部分，第二层为ant-modal，为内容分部，不包含阴影；  
 //要让内容部分百分百于页面，先让ant-modal继承上级高度，若ant-modal设置了top则要减去top的高度；  
 //然后ant-modal-content继承ant-modal百分百，ant-modal-content下面是为兄弟关系的×，Hender，body，header设置了高度后，在body里要100%减去hender的高度，再给body设置背景色为#fff,此时就实现了modal占据整个页面。  
 //若该页面是可超过100%的，则ant-modal-body的高度就用min-height  
 //Modal组件里的元素或组件，都是在ant-modal-body里的，如果是调用另一个外部组件，则样式写在那个组件里即可，若只是简单的html，则样式可直接用'类名'，而不用style.类名，这样就可以写在ant-modal-body里。  
 //sameStyle相当于公共样式，给其他需要100%与页面的加这个类名即可，然后再加一个类名，写属于该modal自己的样式。  
 //maskClosable={false}:点击阴影层不会关闭modal  
 //destroyOnClose：关闭modal时卸载里面的元素，这样就可以显示每次打开modal都重新加载modal里面所有的元素  
 //footer={null}：不要底部按钮  
 render() {  
 **return** (  
 <div className='box'>  
 <Button onClick={**this**.open}>打开</Button>  
 <Modal  
 title="新增变更标注"  
 visible={**this**.state.visibleModal}  
 onCancel={**this**.onCancel}  
 maskClosable={**false**}  
 footer={**null**}  
 centered  
 width={446}  
 style={{height:265}}  
 wrapClassName={'AddLable sameStyle'}  
 >  
 <div className={"content"}></div>  
 <div className={"btn"}>  
 <Button>确定</Button>  
 <Button>取消</Button>  
 </div>  
 </Modal>  
 </div>  
 );  
 }  
}  
  
**export default** one;

### **打开和关闭**

<Modal  
 title={**this**.state.applyTitle}  
 visible={**this**.state.visibleApply}  
 width={'100%'}  
 style={{minWidth:1040}}  
 onCancel={**this**.closeApply}  
 maskClosable={**false**}  
 mask={**false**}  
 destroyOnClose  
 footer={**null**}  
 wrapClassName={'applyAudit sameStyle'}  
>  
 <Apply  
 {...**this**.props}  
 closeApply={**this**.closeApply}  
 applyType={**this**.state.applyType}  
 rowArr={**this**.state.rowArr}  
 />  
</Modal>

//打开申请审核apply：点击哪个按钮需要打开modal，就在哪里调用这个方法，把要传输下去的参数在这里进行设置后赋值给状态，然后把这些状态传参下去  
visibleApply = (text,row)=>{  
 **let** tit = **this**.props.applyTitle.filter(item=>{  
 **return** item.id === text  
 });  
 **this**.setState({  
 applyType:text,  
 visibleApply:**true**,  
 rowArr:row,  
 applyTitle:tit[0].text,  
 })  
};  
  
//关闭申请审核apply：把这个方法首先给modal的onCancel，用于点击X关闭；在赋值给body里的组件，用于点击‘取消’按钮关闭；

不需要清空的情况：若传下去的那些参数，每次打开modal都会重新赋值，而且其他地方没有用到这些参数，则关闭时不需要清空这些参数；（这里就是属于这种，不需要清空）

需要清空：对这些参数的赋值不是在打开modal的时候，而是打开modal之前的操作进行赋值的，那么关闭需要清空，然后在再次打开modal需要验证这些参数是否为空，为空则先提示进行那个赋值的操作后再打开modal。否则再次打开modal时，那些参数都不会变；

//  
closeApply = (bool)=>{  
 **this**.setState({  
 visibleApply:**false** });  
 //提交成功后的关闭要重新请求数据  
 **if**(bool === **true**){  
 **this**.home.getDate()  
 }  
};

### **Modal的js方法：（提示后点击确定）**

title为提示框的提示语，点击确定后要执行的代码写在onOk里

Modal.*confirm*({  
 title: '真的要禁用吗？',  
 content: '',  
 onOk: () => {  
 let *obj* = { id: record.id, userStatus: 3 };  
 *api*.user.updateUserStatus(record.id, *obj*).then((res) => {  
 if (res.code === 0) {  
 *message*.success('禁用成功');  
 const { *current*, *pageSize* } = this.state;  
 let *obj* = Object.assign({ pageNumber: *current*, *pageSize* }, this.search);  
 this.getUserList(*obj*);  
 } else {  
 *message*.error('禁用失败，' + res.msg);  
 }  
 });  
 }  
});

## table组件

table组件里，每行数据需要有唯一的key值，若数据可以自己设置，则每条数据要有key属性，若数据已固定，则用以下方式：

<Table  
 columns={columns}  
 dataSource={**this**.state.dataList}  
 rowKey={record =>record.guid}

onRow={**this**.onRow}  
/>

用每一条数据的guid作为key值（record为每一条数据）

### 圆角与颜色：

:global{  
 .ant-input{  
 border-radius:0;  
 }  
 .ant-btn{  
 border-radius:0;  
 }  
 .ant-select-selection{  
 border-radius:0;  
 }  
 .ant-btn-primary{  
 background-color:#2A99E2;  
 }  
}

固定列时：要设置每列的宽度，然后x的值要大于所有列宽之和，否则会出现重复的列

scroll={{ x:1880,y:**this**.props.height}}

### 滚动部分的高度跟随页面大小变化：

若不需要自适应页面高度，直接定死，则直接在body里设置具体的height即可。

//在第一次计算出表格的高度后，要从上往下一层层的高度100%传递下去，直到ant-table-body，就可以根据100%的高度减去表头的高度得到表体的固定高度。

//无固定列，滚动，只有一层表头：

:global{  
 .listMeasure{  
 height:100%;  
 .ant-spin-nested-loading{  
 height: 100%;  
 .ant-spin-container{  
 height: 100%;  
 .ant-table-small{  
 border-radius:0;  
 }  
 .ant-table{  
 height: 100%;  
 .ant-table-content{  
 height: 100%;  
 .ant-table-scroll{  
 height: 100%;  
 .ant-table-header{  
 margin-bottom: -17px **!important**;  
 table{  
 .ant-table-thead{  
 tr{  
 height: 30px;  
 th{  
 background-color: #E7F1FA;  
 font-weight: bold;  
 font-size:12px;  
 padding:0 0 0 10px;  
 border-bottom: 1px solid #fff;  
 }  
 }  
 }  
 }  
 }  
 .ant-table-body{  
 font-size:12px;  
 height:calc(100% - 30px);  
 }  
 }  
 }  
 }  
 }  
 }  
 .ant-table-bordered .ant-table-tbody>tr>td, .ant-table-bordered .ant-table-thead>tr>th,.ant-table-small.ant-table-bordered .ant-table-fixed-left .ant-table-tbody>tr>td:last-child,.ant-table-small.ant-table-bordered .ant-table-fixed-left .ant-table-thead>tr>th:last-child{  
 border-right:1px solid #fff;  
 }  
 }  
}

有固定列，滚动，且两层表头

、、、、、、、、、、、、、、、、

:global{  
 .listTableStyle{  
 height: 100%;  
 .ant-spin-nested-loading{  
 height: 100%;  
 .ant-spin-container{  
 height: 100%;  
 .ant-table-small{  
 border-radius:0;  
 }  
 .ant-table{  
 height: 100%;  
 .ant-table-content{  
 height: 100%;  
 .ant-table-fixed-left,.ant-table-scroll{  
 .ant-table-header{  
 table{  
 .ant-table-thead{  
 tr{  
 //height: 30px;  
 th{  
 background-color: #E7F1FA;  
 font-weight: bold;  
 font-size:12px;  
 padding:0 0 0 10px;  
 border-bottom: 1px solid #fff;  
 }  
 }  
 }  
 }  
 }  
 .ant-table-body,.ant-table-body-outer{  
 font-size:12px;  
 .ant-table-tbody{  
 tr{  
 }  
 .clickRowSty{  
 background-color:#ddd;  
 }  
 }  
 }  
 }  
 .ant-table-fixed-left{  
 height:calc(100% - 17px);  
 .ant-table-header{  
 table{  
 .ant-table-thead{  
 th{  
 //height: 100% !important;  
 }  
 }  
 }  
 }  
 .ant-table-body-outer{  
 height:calc(100% - 61px + 17px);  
 font-size:12px;  
 }  
 }  
 .ant-table-scroll{  
 height: 100%;  
 .ant-table-header{  
 margin-bottom: -17px **!important**;  
 tr{  
 height: 30px **!important**;  
 }  
 }  
 .ant-table-body{  
 height:calc(100% - 61px);  
 }  
 }  
 }  
 }  
 .ant-table-bordered .ant-table-tbody>tr>td, .ant-table-bordered .ant-table-thead>tr>th,.ant-table-small.ant-table-bordered .ant-table-fixed-left .ant-table-tbody>tr>td:last-child,.ant-table-small.ant-table-bordered .ant-table-fixed-left .ant-table-thead>tr>th:last-child,.ant-table-small.ant-table-bordered .ant-table-thead>tr>th:last-child{  
 border-right:1px solid #fff;  
 }  
 }  
 }  
 }  
}

有滚动有定位，二层标题的表格样式：

//注意：如果设置的高度已经执行了，但是页面的tr的高度并不是设置的高度，可能是因为padding问题

//注意maxheight和height的区别：

只有Max-height时。页面高度由内容决定，页面内容大于max-height就会出现滚动条，

而如果max-height和height同时出现时。若height=200小于max-height=300，页面高度会以height=200为主，若height=1000，大于了max-height=300.则页面高度会以max-height=300为主。但如果子元素的max-height为100%，而父元素只有height，那么子元素的max-height就会继承父元素height。

这里的Table属性y:是max-height，要以max-height受css控制，可将y为100%，即max-height为100%，此时的max-height是body和body-inner的，但是只要在body和body-outer分别计算出相同的高度，一定要相同，否则表格会错位。就实现了自适应。

若有固定和滚动的情况，记住要将ant-table-body的父元素.ant-table-scroll的高度为页面100%，.ant-table-fixed-left的高度为100% - 17px，因为左边的fixed-left要隐藏滚动条。所以这里要减去滚动条，为了保证最终的max-height一致，再在子元素里加上滚动条。

:global{  
 .ant-table-content{

//第一步：先设置fixed-left和scroll所有共同的样式  
 .ant-table-fixed-left,.ant-table-scroll{  
 .ant-table-header{  
 table{  
 .ant-table-thead{  
 tr{  
 th{  
 background-color: #E7F1FA;  
 font-weight: bold;  
 font-size:12px;  
 padding:0 8px;  
 border-bottom: 1px solid #fff;  
 }  
 }  
 }  
 }  
 }

//outer是左边固定了的部分，body是右边滚动的body  
 .ant-table-body,.ant-table-body-outer{  
 font-size:12px;  
 .ant-table-tbody{  
 tr{  
 height: 42px **!important**;//body和body-outer的行高都要设置，不然两边表格有错位  
 td{  
 padding:0 0 0 5px;  
 }  
 }  
 }  
 }

}

//第二步：单独设置fixed-left的样式：

//这里fixed已经自带overflow:hidden，所以将这个父组件的高度减去滚动条，子元素再加上滚动条，但是被父元素溢出隐藏了。这样就可以保持fixed-left的高度与scroll的一致，又隐藏了滚动条。  
 .ant-table-fixed-left{  
 height:calc(100% - 17px);  
 .ant-table-header{  
 table{  
 .ant-table-thead{  
 tr{  
 height: 60px **!important**;  
 }  
 }  
 }  
 }  
 .ant-table-body-outer{  
 height:calc(100% - 60px + 17px);//60为表头的高度，17为滚动条  
 font-size:12px;  
 }  
}

.ant-table-scroll{  
 height: 100%;  
 .ant-table-header{  
 tr{  
 height:30px **!important**;  
 }  
 }  
 .ant-table-body{  
 height:calc(100% - 60px);  
 }  
 }

}  
 .ant-table-bordered .ant-table-tbody>tr>td, .ant-table-bordered .ant-table-thead>tr>th,.ant-table-small.ant-table-bordered .ant-table-fixed-left .ant-table-tbody>tr>td:last-child,.ant-table-small.ant-table-bordered .ant-table-fixed-left .ant-table-thead>tr>th:last-child,.ant-table-small.ant-table-bordered .ant-table-thead>tr>th:last-child{  
 border-right:1px solid #fff;  
 }  
}

无数据时，需要重新计算body高度：

//暂无数据那个div是body的兄弟，而不是body-outer，所以得改body的高度减去53，才能保证和body-outer的一致。可以在包裹table的div盒子里判断数据是否为空，动态加载table3这个类名，而且尽量写在上面自适应样式的下面，便于这里的body高度可以覆盖上面body的高度。

.table3{  
 :global{  
 .ant-table-wrapper {  
 height: 100%;  
 .ant-spin-nested-loading {  
 height: 100%;  
 .ant-spin-container {  
 height: 100%;  
 .ant-table {  
 height: 100%;  
 .ant-table-content {  
 height: 100%;  
 .ant-table-scroll {  
 height: 100%;  
 .ant-table-body {  
 height: calc(100% - 60px - 53px);  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
}

点击每行执行函数：

onRow=(record) => {  
 **return** {  
 onClick: () => {  
 console.log(record)  
 }  
 }  
};

### 根据不同数据对应渲染不同内容：

//第一个参数text是该字段内容，第二个参数record是该行数据

{  
 title: '类别',  
 dataIndex: 'NodeType',  
 key: 'NodeType',  
 width: 70,  
 fixed: 'left',  
 render:(text,record)=>{  
 **switch** (text) {  
 **case** 1:  
 **return** '分部';  
 **case** 2:  
 **return** '子分部';  
 **case** 3:  
 **return** '清单';  
 **case** 4:  
 **return** '工作内容';  
 **case** -100:  
 **return** '直接条目';  
 **case** 5:  
 **return** '定额';  
 }  
 }  
},

{  
 title: '量差%',  
 dataIndex: 'QtyDifPer',  
 key: 'QtyDifPer',  
 width:100,  
 align:'center',  
 render:(text, row)=>{  
 **if**(row.ListQty === 0){  
 **return** 0.00  
 }**else**{  
 **return** (row.Result - row.ListQty)/row.ListQty}  
 }  
}

或者：

render:(text,row)=> {  
 **switch** (text) {  
 **case** 0:  
 **return** <a  
 className={"myAnticon anticon-bianji"}  
 onClick={()=> **this**.props.visibleApply(text,row)}  
 />;  
 **case** 1:  
 **return** <div>已提交</div>;  
 **case** 6:  
 **return** <div>审核完成</div>;  
 **case** -1:  
 **return** <div>审核不通过</div>;  
 **default**:  
 **return** <div>审核中</div>  
 }  
}

是否渲染图标：

render:(text,row)=>{  
 **return** text === 1?<a onClick={()=>**this**.unlocking(text,row)}>可解锁</a>:""  
}

### 点击行增加高亮

<Table

onRow={**this**.onClickRow}  
 rowClassName={**this**.setRowClassName}

>

//点击表格行-保存行数据  
onClickRow = (record) => {  
 **return** {  
 onClick: () => {  
 **if**(**this**.state.rowId === '' || **this**.state.rowId !== record.guid){  
 **this**.setState({  
 rowId: record.guid,  
 rowobj:record  
 });  
 }**else if**(**this**.state.rowId === record.guid){  
 **this**.setState({  
 rowId: '',  
 rowobj:**undefined** });  
 }  
 },  
 };  
};  
  
//给表格行添加类名  
setRowClassName = (record) => {  
 **return** record.guid === **this**.state.rowId ? 'clickRowSty' : '';  
};

css样式：

.ant-table-body,.ant-table-body-outer{  
 .ant-table-tbody{  
 tr{  
 }  
 .clickRowSty{  
 background-color:#eee;  
 }  
 }  
}

### **单元格属性**

//单元格属性-打开标注详情  
onCell = (record) => {  
 **return** {  
 onClick: (e) => {  
 //阻止冒泡  
 e.stopPropagation();  
 e.nativeEvent.stopImmediatePropagation();  
 **this**.openAddLable('details',record);  
 }  
 }  
};

## Button

在modal写button时的常用样式：

.btn{  
 text-align: center;  
 height: 44px;  
 line-height: 44px;  
 .ant-btn{  
 border-radius: 0;  
 padding: 0;  
 width: 50px;  
 height: 24px;  
 margin-right:10px;  
 }  
}

## **表单from**

### 双向数据绑定

**import** { Form} **from** 'antd'

**const** FormItem = Form.*Item*

*给表单赋值：*

**class** NormalLoginForm **extends** React.Component {

*}*

componentDidMount(){  
 **let** orj = {  
 userName:’jjjjjj’,  
 oldpassWord:**’122223’**  
 };  
 **this**.props.form.setFieldsValue(orj);  
}

*获取表单数据：*

*Summit=()=>{*

**this**.props.form.validateFields((err, values) => {

if(!err){

Console.log(values.userName,values.oldpassWord)

**this**.props.form.resetFields();//重置

}

*}*

*}*

*设置数据：*

**let** orj = {  
 userName:**this**.props.userName,  
 oldpassWord:**this**.props.passWord  
};  
**this**.props.form.setFieldsValue(orj);

*清空数据：*

**this**.props.form.resetFields(['newpassWord','repassWord']);

*清空所有数据：*

**this**.props.form.resetFields();

<FormItem  
 // label="用户名"  
>  
 {getFieldDecorator('userName', {  
 rules: [{ required: **true**, message: '请输入正确的用户名' }],  
 })(  
 <Input size='large' prefix={<Icon type="user" style={{ fontSize: 13 }} />} placeholder="键入您的用户名" />  
 )}  
</FormItem>

<FormItem  
 // label="旧密码"  
>  
 {getFieldDecorator('oldpassWord', {  
 rules: [{ required: **true**, message: '请输入密码' }],  
 })(  
 <Input size='large' prefix={<Icon type="lock" style={{ fontSize: 13 }} />} type="password" placeholder="键入您的旧密码" />  
 )}  
</FormItem>

**const** WrappedNormalLoginForm = Form.*create*()(NormalLoginForm);

### **校验的坑**

如果在一个js文件写了多个表单，尽管页面只渲染一个，但只要required为true，校验的时候还是全部都会进行校验：

eleType = (type) => {  
 const { *getFieldDecorator* } = this.props.form;  
 let *ele* = {  
 account: (  
 <Form onSubmit={(e) => this.onSubmit(e, 'account')} className={style.loginForm} onFocus={this.onFocus}>  
 <FormItem>  
 {*getFieldDecorator*('userName1', {  
 rules: [{ required: true, message: '请输入正确的用户名' }]  
 })(<Input size="large" placeholder="账号名称" />)}  
 </FormItem>  
 <FormItem>  
 {*getFieldDecorator*('passWord1', {  
 rules: [{ required: true, message: '请输入正确的密码' }]  
 })(<Input size="large" type="password" placeholder="密码" />)}  
 </FormItem>  
 <FormItem>  
 <Button type="primary" htmlType="submit" className="login-form-button" block>  
 登录  
 </Button>  
 </FormItem>  
 </Form>  
 ),  
 phone: (  
 <Form onSubmit={(e) => this.onSubmit(e, 'phone')} className={style.loginForm} onFocus={this.onFocus}>  
 <FormItem>  
 {*getFieldDecorator*('userName2', {  
 rules: [{ required: true, message: '请输入正确的用户名' }]  
 })(<Input size="large" placeholder="手机号" />)}  
 </FormItem>  
 <FormItem>  
 {*getFieldDecorator*('passWord2', {  
 rules: [{ required: true, message: '请输入正确的验证码' }]  
 })(  
 <Input  
 size="large"  
 type="password"  
 placeholder="验证码"  
 addonAfter={<a onClick={this.getCode}>获取验证码</a>}  
 />  
 )}  
 </FormItem>  
 <FormItem>  
 <Button type="primary" htmlType="submit" className="login-form-button" block>  
 登录  
 </Button>  
 </FormItem>  
 </Form>  
 )  
 };  
 return *ele*[type];  
};

导致err一直有值，而在这里获取不到值：(此时的value，若还没填写，则为undefined，若填写了再删除，为空字符串）

this.props.form.validateFields((err, values) => {  
 if (!err) {

Console.log(values);  
 }  
});

但如果把required的状态用变量来表示的话，当状态改变的时候，功能会有延迟：状态改变了，但是还没有作用到输入框上，此时输入框校验的状态变成了都不校验，一个输入框都不校验了，必须输入后，改变后的变量才有作用。因此此方法不可行。

换一个方法，还是用上面的变量，但在改变变量的时候，给该输入框用validateFields：

this.setState(  
 {  
 checkNick: e.target.checked,  
 },  
 () => {  
 this.props.form.validateFields(['userName'], { force: true });  
 },  
);

但这个方法表示立即对该输入框进行校验，哪怕还没有输入内容，然后就会出现校验的错误提示，但错误提示需要在点击提交的时候才验证。因此这个方法也不可行。

正确方法：

把所有表单的required都为true，然后再提交获取值时，再传入表单名进行选择性校验：（获取值时的验证是在required为true的基础上的）

static *defaultProps* = {  
 arr: {  
 account: ['userName1', 'passWord1'],  
 phone: ['userName2', 'passWord2']  
 }  
};  
  
//校验成功-登录  
onSubmit = (e, type) => {  
 const { *arr* } = this.props;  
 e.preventDefault();  
 this.props.form.validateFields(*arr*[type], (err, values) => {  
 if (!err) {  
 this.login(values[*arr*[type][0]], values[*arr*[type][1]]);  
 }  
 });  
};  
  
//登录  
login = (userName, passWord) => {  
 console.log(userName, passWord);  
};

## **Input输入框**

### 获取多个输入框的值

页面有多个input，要获取输入的值，可只写一个onChange，调用时传递不同参数进行区分。

当相同的方法要被多个地方调用时，可只写一个，然后根据传递的参数进行区分。

<Input  
 value={drawingcode}  
 onChange={(e)=>**this**.applyChange(e,1)}  
 disabled={visibleAudit}  
/>

applyChange = (text,num)=>{  
 **switch** (num) {  
 **case** 1://图号  
 **this**.setState({  
 drawingcode:text.target.value  
 });  
 **break**;  
 **case** 2://变更原因  
 **this**.setState({  
 changereason:text.target.value  
 });  
 **break**;  
 **case** 3://变更内容  
 **this**.setState({  
 changecontent:text.target.value  
 });  
 **break**;  
 **case** 4://申请意见  
 **this**.setState({  
 applyopinion:text.target.value  
 });  
 **break**;  
 **case** 5://一审意见  
 **this**.setState({  
 auditopinion1:text.target.value  
 });  
 **break**;  
 **default**:  
 **break**;  
 }  
};

### TextArea

要使文本框拖拉不动，只需设置相同的minRows和maxRows即可。

获取值与回显与input一致。

<TextArea  
 autosize={{ minRows:5, maxRows:5}}  
 value={auditopinion1}  
 onChange={(e)=>**this**.applyChange(e,5)}  
 disabled={visibleAudit1}  
/>

## **Loading**

**把要loading的区域用spin包裹起来，要加载时，给loading为true，加载结束时，给loading为false**

**import** {Spin} **from** 'antd'

<Spin spinning={**this**.state.loading}>

.....  
</Spin>

## **Radio单选**

### 多个radio实现单选效果

//单选框功能  
clickRadio = (text,row)=>{  
 **if**(**this**.state.radioGuid === ''){  
 **this**.setState({  
 radioType:1,  
 radioGuid:row.guid  
 })  
 }**else if**(row.guid === **this**.state.radioGuid){  
 **this**.setState({  
 radioType:0,  
 radioGuid:''  
 })  
 }**else if**(row.guid !== **this**.state.radioGuid){  
 **this**.setState({  
 radioType:1,  
 radioGuid:row.guid  
 })  
 }  
};

**这里的多个radio里，为1表示默认选中的，渲染底下的radio组件；**

**为0表示默认不选中，渲染上面的radio；初始数据里只有一个radio为1；**

**switch** (text) {  
 **case** 0:  
 **return** <Radio  
 checked={**this**.state.radioType === 1 && row.guid === **this**.state.radioGuid}  
 onClick={()=>**this**.clickRadio(text,row)}  
 />;  
 **case** 1:  
 **return** <Radio  
 checked={text === 1}  
 onClick={()=>**this**.clickRadio(text,row)}  
 />;  
 **default**:  
 **return** ''  
}

## **Tree树**

**import** { Tree ,Icon} **from** 'antd'  
**const** { *TreeNode* } = Tree

**export default class** Admin **extends** Component{  
 state = {  
 expandedKeys: [],  
 autoExpandParent: **true**,  
 checkedKeys: [],  
 selectedKeys: [],  
 };  
  
 //点击展开图标时触发：在原来数组的基础上，将点击到的那条数据的key值添加或删除给数组  
 //这里会出现一个问题，就是若数组里初始时有了最底层的key，而这个方法里不可能添加或删除到最底层的key，然后这个key值就会一直存在于数组里，导致展开到这个key的层级都失去效果而一直展开着，因此在初始时不能设置最底层的key，只能设置有子级的key给数据。  
 //用了这个属性，就不能隔层收缩，必须一层一层往外收缩，因为每点击一次只能删除一个key，点击最外层的key删除后，但里层的key还在。  
 //['0-0-0','0-0-1']:若第三层有条数据的key值为0-0-1，只需要0-0-1给这个数组，就可以展开到这条数据，而不需要将它的所有父元素都给数组。若0-0-1有子元素，则表示展开到这条数据的所有元素  
 onExpand = expandedKeys => {  
 **this**.setState({  
 expandedKeys,  
 autoExpandParent: **false**,  
 });  
 };  
  
 //点击复选框时触发：  
 //页面页面上所有打了勾的key都会添加到数组里，因为·默认父子节点是关联的，若点击了一个父级，则父级的key和这个父级所有子级的key都会添加到数组里，当所有子节点都被选中后，其父节点也会被选中。删除也是一样的。也就是说数组里的key都会打上勾。  
 onCheck = checkedKeys => {  
 // console.log('onCheck', checkedKeys);  
 **this**.setState({ checkedKeys });  
 };  
  
 //点击树节点时触发：  
 //数组里只能有一个key，每次将点击到的key替换数组里原先的key，页面样式为这条数据高亮，表示被鼠标选中。子父节点间没有任何关联  
 onSelect = (selectedKeys, info) => {  
 // console.log('onSelect', selectedKeys);  
 **this**.setState({ selectedKeys });  
 };  
  
 //根据数据递归加载节点  
 renderTreeNodes = data =>  
 data.map(item => {  
 **if** (item.children) {  
 **return** (  
 <TreeNode title={item.title} key={item.key} dataRef={item}>  
 {**this**.renderTreeNodes(item.children)}  
 </TreeNode>  
 );  
 }  
 **return** <TreeNode {...item} />;  
 });  
  
 render() {  
 **return** (  
 <Tree  
 //自动展开所有节点，但有时候没作用，所有还是用expandedKeys和onExpand  
 autoExpandParent  
  
 //展开：  
 onExpand={**this**.onExpand}  
 expandedKeys={**this**.state.expandedKeys}  
 //默认展开所有树节点：必须得在expandedKeys不使用的情况下才能使用defaultExpandAll，否则会以expandedKeys为主  
 defaultExpandAll  
 //默认展开指定的树节点： 若有expandedKeys，以若有expandedKeys为主；若无若有expandedKeys，但有defaultExpandAll，则展开所有节点，若也无但有defaultExpandAll，才执行defaultExpandedKeys  
 defaultExpandedKeys={['0-0-0']}  
  
 //复选框：  
 //节点前加复选框  
 checkable  
 onCheck={**this**.onCheck}  
 checkedKeys={**this**.state.checkedKeys}  
 //默认选中复选框的树节点：必须得在checkedKeys不使用的情况下才能使用defaultCheckedKeys，否则会以checkedKeys为主  
 defaultCheckedKeys={['0-0-0']}  
 //这个属性表示父节点和子节点断开关系，不会关联  
 checkStrictly  
  
 //鼠标点击：  
 onSelect={**this**.onSelect}  
 selectedKeys={**this**.state.selectedKeys}  
 //默认选中的树节点:必须得在selectedKeys不使用的情况下才能使用defaultSelectedKeys，否则会以selectedKeys为主  
 defaultSelectedKeys={['0-0-0']}  
 //支持点击多个节点  
 multiple  
  
 //将树禁用：可以展开，不可以选择和选中高亮  
 disabled  
  
 //自定义折叠图标  
 showIcon  
 switcherIcon={<Icon type="down-square" />}  
  
 //是否展开连线，若使用了这个属性，则自定义折叠图标就失效了  
 showLine  
 >  
 {**this**.renderTreeNodes(treeData)}  
 </Tree>  
 );  
 }  
}

## Layout布局

## **tabs组件共用组件引起组件多次渲染**

import React,{ Component } from 'react';  
import { Tabs,Button } from 'antd';  
import Content from './Content/index'  
import './css.less';  
  
const { *TabPane* } = Tabs;  
  
export default class Tab extends Component{  
 constructor(){  
 super();  
 this.state = {  
 key:'1',  
 value:9999  
 }  
 }  
  
 //如果在修改状态后又进行一次修盖状态，则此render渲染两次，content就渲染6次  
 onChange = (key)=>{  
 this.setState({  
 key  
 })  
 };  
  
 //只要这个组件里用了setState，触发此render重新渲染，render里用了三次content组件，因此这个组件都会被触发三次，无论是否传递参数下去  
 changeValue = ()=>{  
 this.setState({  
 value:Math.random().toString().substr(0,5)  
 })  
 };  
  
 //初始时，只渲染了第一个content，后面两个还没有渲染，因此tab的componentDidMount，和第一个content的componentDidMount，和第一个content里面的one的componentDidMount执行。  
 //点击第二个tab后，此时第一个也会渲染，但此时第一个是content的componentWillReceiveProps和第一个的one的componentWillReceiveProps执行，而第二个content是第一次渲染，因此第二个content是componentDidMount，和第二个content里面的one的componentDidMount执行  
 //第三个同样，第三次是第一次渲染，而第一个content是第三次渲染，第二个content是第二次渲染  
 render(){  
 const {*key*,*value*} = this.state;  
 return (  
 <div>  
 <Button onClick={this.changeValue}>爷组件：改变一个状态传递下去</Button>  
 <Tabs activeKey={*key*} onChange={this.onChange}>  
 <TabPane tab="Tab 1" key="1">  
 <Content value={*value* +'父组件1'}/>  
 </TabPane>  
 <TabPane tab="Tab 2" key="2">  
 <Content value={*value*+'父组件2'}/>  
 </TabPane>  
 <TabPane tab="Tab 3" key="3">  
 <Content value={*value*+'父组件3'}/>  
 </TabPane>  
 </Tabs>  
 </div>  
 )  
 }  
}

content子组件：

import React,{ Component } from 'react';  
import { Input } from 'antd';  
import One from './One/index';  
import './css.less';  
  
export default class Content extends Component{  
 constructor(props){  
 super(props);  
 //这里只会在初始状态下执行一次，因此，第一次props传下来的value是什么，text就是什么，后面props再传下来不同的value参数，这里就不会接收了。  
 // 要想更新text值，这时只能在下面的componentWillReceiveProps里面可以接收到新的value属性，然后用setState赋值给text。  
 this.state = {  
 text:props.value  
 }  
 }  
 //初始化时不会执行，但是父组件里用了setState后，子组件重新渲染后，同时这个钩子函数也会执行，无论是否有参数传递下来,  
 //也无论传递的参数是否相同  
 //若父组件的value改变了，则this.props.value和props.value不相同，但这个钩子函数还是会一模一样的执行三遍  
 //但三次渲染传下来的参数互不影响  
 // 注意：当父组件第二次渲染时，这个钩子函数执行后，render才执行，此时在里面用setState修改状态时，render会等状态修改完就猜执行，因此render只执行一次吗，而不会执行两次。因此跟直接在render里用this.props.value的效果是一样的。两种可选择一种方式即可。一般如果参数是回显用，则执行在render用this.props.value,其他的就在这里钩子函数里更新状态。  
 componentWillReceiveProps(props){  
 console.log('componentWillReceiveProps')  
 if(this.props.value !== props.value){  
 // this.setState({  
 // text:props.value  
 // })  
 }  
 }  
  
 onChange=(text)=>{  
 this.setState({  
 text:text.target.value  
 })  
 };  
 render(){  
 return (  
 <div>  
 <Input value={this.state.text} onChange={this.onChange}/>  
 <One/>  
 </div>  
 )  
 }  
}

## **Upload**

import React, { Component } from 'react';  
import { Upload, *Icon*, Modal } from 'antd';  
import './css.less';  
  
//异步读取文件  
function getBase64(file) {  
 return new Promise((resolve, reject) => {  
 const *reader* = new FileReader();  
 *reader*.onload = () => resolve(*reader*.result);  
 *reader*.onerror = error => reject(error);  
 });  
}  
  
export default class App extends Component {  
 constructor(){  
 super();  
 this.state = {  
 previewVisible: false,//控制预览modal的  
 previewImage: '',  
 fileList: [  
 {  
 uid: '-1',  
 name: 'image.png',  
 status: 'done',  
 url: 'https://zos.alipayobjects.com/rmsportal/jkjgkEfvpUPVyRjUImniVslZfWPnJuuZ.png',  
 }  
 ],  
 };  
 }  
 handleCancel = () => this.setState({ previewVisible: false });  
  
 //打开图片预览  
 handlePreview = async file => {  
 //如果图片参数里url和preview这两个参数都没有的话，就取参数originFileObj将其转变为base64编码  
 if (!file.url && !file.preview) {  
 file.preview = await getBase64(file.originFileObj);  
 }  
 console.log(file.originFileObj)  
 // getBase64(file.originFileObj)  
 //有file.url1,去该值，没有，则取file.preview  
 this.setState({  
 previewImage: file.url || file.preview,  
 previewVisible: true,  
 });  
 };  
  
 //图片变化时的回调；上传，删除时执行  
 // response为上传接口的响应  
 //上传时执行三次，状态info.file.status分别为uploading,uploading,done.前两次没有response，最后一次上传成功后，状态变成done，也有了response  
 //重点：每一次执行都必须获取fileList，传递给upload标签。若第一次没有获取传递进去，则第二次就不执行了。就无法上传成功  
 handleChange = (info) =>{  
 console.log(info.file);  
 let *fileList* = info.fileList;  
 this.setState({ *fileList* })  
 };  
  
  
 render() {  
 const { *previewVisible*, *previewImage*, *fileList* } = this.state;  
 //一般图片上传组件需要两个标签。一个是Upload上传标签，一个是图片预览modal。  
 // Upload：  
 //action:上传接口  
 // fileList：所有文件列表  
 //onPreview：点击了图片的预览图标的回调  
 //onChange：图片上传，删除，都会调用这个方法  
 return (  
 <div className="clearfix">  
 <Upload  
 action="https://www.mocky.io/v2/5cc8019d300000980a055e76"  
 listType="picture-card"  
 fileList={*fileList*}  
 onPreview={this.handlePreview}  
 onChange={this.handleChange}  
 >  
 {*fileList*.length >= 3 ? null : (  
 <div>  
 <Icon type="plus" />  
 <div className="ant-upload-text">Upload</div>  
 </div>  
 )}  
 </Upload>  
 <Modal  
 visible={*previewVisible*}  
 footer={null}  
 onCancel={this.handleCancel}  
 >  
 <img  
 alt="example"  
 style={{ width: '100%' }}  
 src={*previewImage*}  
 />  
 </Modal>  
 </div>  
 );  
}  
}

这里是一个错误示范：第一次执行时将fileList清空了，此后就不执行了。因此状态一直是unloading

handleChange = info => {

let fileList = info.fileList;

fileList.filter(file => file.status === 'done');

this.setState({fileList});

}

受控情况下，在onChange回调里把fileList 设置为空，Upload组件内部的fileList就更新为空数组了，它内部的onSuccess，onProgress等逻辑里会从fileList找触发事件的file，找不到就不触发onChange回调了

# **bizcharts**

## 导出为各类图片

**let** menu = (  
 <Menu>  
 <Menu.Item key="1" onClick={() => {**this**.showTo(1)}}>打印图表</Menu.Item>  
 <Menu.Divider />  
 <Menu.Item key="2" onClick={() => {**this**.showTo(2)}}>导出 PNG 图片</Menu.Item>  
 <Menu.Item key="3" onClick={() => {**this**.showTo(3)}}>导出 JPG 图片</Menu.Item>  
 <Menu.Item key="4" onClick={() => {**this**.showTo(4)}}>导出 PDF 文档</Menu.Item>  
 <Menu.Item key="5" onClick={() => {**this**.showTo(5)}}>导出 SVG 图片</Menu.Item>  
 </Menu>  
);

onVisibleChange = (visible,type)=>{  
 **this**.setState({  
 type,  
 })  
};

//区分图表  
showTo = (type)=>{  
 **let** {chartIns1,  
 chartIns2,  
 chartIns3,  
 chartIns4,  
 chartIns5,  
 chartIns6,  
 chartInsSvg1,  
 chartInsSvg2,  
 chartInsSvg3,  
 chartInsSvg4,  
 chartInsSvg5,  
 chartInsSvg6,  
 title  
 } = **this**.state;  
 **switch** (**this**.state.type) {  
 **case** 'small1':  
 **this**.exportImg(type,chartIns1,chartInsSvg1,'审核进程对比');  
 **break**;  
 **case** 'small2':  
 **this**.exportImg(type,chartIns2,chartInsSvg2,`${title}数量对比`);  
 **break**;  
 **case** 'small3':  
 **this**.exportImg(type,chartIns3,chartInsSvg3,`${title}费用汇总对比`);  
 **break**;  
 **case** 'large1':  
 **this**.exportImg(type,chartIns4,chartInsSvg4,`月${title}数统计`);  
 **break**;  
 **case** 'large2':  
 **this**.exportImg(type,chartIns5,chartInsSvg5,`月${title}费用统计`);  
 **break**;  
 **case** 'large3':  
 **this**.exportImg(type,chartIns6,chartInsSvg6,`${title}费用汇总分析图`);  
 **break**;  
 **default**:  
 **break**;  
 }  
};  
  
//区分导出图片类型  
exportImg = (type,chartIns,chartInsSvg,name)=>{  
 **switch** (type) {  
 **case** 1://打印图表  
 **this**.print(chartIns,"image/png",name);  
 **break**;  
 **case** 2://导出 PNG 图片  
 **this**.image(chartIns,"image/png",name);  
 **break**;  
 **case** 3://导出 JPG 图片  
 **this**.image(chartIns,"image/jpeg",name);  
 **break**;  
 **case** 4://导出 PDF 文档  
 **this**.importPdf(chartIns,"image/jpeg",name);  
 **break**;  
 **case** 5://导出 SVG 图片  
 **this**.svgImg(chartIns,'image/svg+xml',chartInsSvg,name);  
 **break**;  
 **default**:  
 **break**;  
 }  
};  
  
//打印  
print = (chartIns,imageTpye,name)=>{  
 **let** image3 = chartIns.get('canvas').\_cfg.el.toDataURL(imageTpye);  
 **let** blankPage = window.open();  
 **let** img = document.createElement('img');  
 img.setAttribute("src", image3);  
 blankPage.document.body.appendChild(img);  
 setTimeout(()=>{  
 blankPage.print();  
 }, 500);  
};  
  
//导出png,jpg图片  
image = (chartIns,imageTpye,name)=>{  
 **let** image = chartIns.get('canvas').\_cfg.el.toDataURL(imageTpye);  
 **let** ele = document.createElement('a');  
 ele.setAttribute("href", image);  
 ele.setAttribute("download", name);  
 ele.click();  
};  
  
//导出为svg  
svgImg = (chartIns,imageTpye,chartInsSvg,name)=>{  
 chartInsSvg.downloadImage()  
};  
  
//导出pdf  
importPdf = (chartIns,imageTpye,name)=>{  
 console.log('pdf');  
 **let** ele = **new** jsPDF('p','mm','a4');  
 **let** image = chartIns.get('canvas').\_cfg.el.toDataURL(imageTpye);  
 ele.addImage(image, 'JPEG', 10,10,180,100);  
 ele.save(name + ".pdf")  
};

导出svg前提是图表由svg渲染

# 项目方法

## 项目封装方法：

### 设置两个类名

import classNames from 'classnames'；

<div className={classNames(style.item,style.post)}>

或者：（拼接一个空格）

<div className={style.item + ‘ ’ + style.post)}>

可以用条件判断：

classNames(style.a,{[style.b]:bool})

### 保留两位小数：

把数字放进去：

**import** webapp **from** 'Res/js/webapp';

webapp.floatNumFormatted(text)

### 修改ant样式

:global(.DraftEditor-root){

padding:15px;

height:50%;

}

或:

:global{

.DraftEditor-root{}

}

### 颜色保存

蓝色：#2A99E2

边框色：#E8E8E8

### 引入接口

**import** api **from** 'src/App.api.js'

api.changeVisaClaim.listUpdate(obj,PrjID).then(res=>{})

### 字体图标

<i className={"myAnticon anticon-chuyidong"}/>

<a className={"myAnticon anticon-bianji"}/>

### **Webapp使用**

#### 保留两位小数

**import** webapp **from** 'Res/js/webapp'

webapp.floatNumFormatted(text)

### **模型ModelViewer**

#### 模型回显

父元素将dataJson传递下来，因此在这个周期函数里开始加载模型，模型只需要加载一次，因此第一次传下来不等于undefined的dataJson时，加载模型，且把dataJson保存起，等再传下来dataJson时，已经有这个值了，就不需要再次加载模型

componentWillReceiveProps(props){  
 **if**(**this**.state.dataJson === **undefined**) {  
 **this**.loadModel(props);  
 }  
}

加载前开启loading，判断dataJson不为undefined后，保存dataJson，循环加载里面的每一个json文件

//加载模型json数据  
loadModel = (props)=>{  
 **const** {dataJson} = props;  
 **let** that = **this**;  
 **this**.setState({ loading: **true** });  
 **if**(dataJson !== **undefined**) {  
 **this**.setState({dataJson,});  
 dataJson.rows.forEach(row => {  
 **let** count = 0;  
 that.addJson(row.WebFiles, count)  
 });  
 }  
};

这里用到递归，因为必须在加载完前一个json文件后才能加载下一个json文件，可先设定索引为0，加载完一个依次加1，直到索引与json文件个数相等时，相当于所有json文件都加载结束，此时就设置loadFinish和setView

//加载json文件  
addJson = (files,count)=>{  
 **let** PrjID = **this**.props.PrjID;  
 **let** that = **this**;  
 **if**(count === files.length){  
 **this**.finishCall();  
 **return** }  
 LoadUnify.loadModelJsonFile(files[count]).then(res=>{  
 **this**.setState({  
 LoadModel:{  
 DataJson: res,  
 DataTime:Math.random(),  
 para:{  
 PrjID:PrjID  
 },  
 OnFinishCall:()=>{  
 that.addJson(files,count+1);  
 }  
 },  
 })  
 })  
};  
  
//最后一个json加载完成调用（必须设置loadFinish和setView）  
finishCall = ()=>{  
 **this**.setState({  
 LoadFinish:{  
 DataTime:Math.random(),  
 Version:3,  
 },  
 SetView:{  
 Mode:'Front',  
 DataTime:Math.random(),  
 Version:3,  
 },  
 loading:**false** },()=>{  
 **this**.handleShowLabel()  
 })  
};

这些状态初始都可为undefined；

Spin为要loading的区域；

<Spin spinning={**this**.state.loading}>  
 <TopMenu  
 onChoiceView={**this**.handleChoiceView}  
 />  
 {**this**.pointWord()}  
 <ModelViewer  
 LoadModel={**this**.state.LoadModel}  
 LoadFinish={**this**.state.LoadFinish}  
 SetView={**this**.state.SetView}  
 OnPickCall={(data)=>**this**.onPickCall(data,0)}  
 HighModel={**this**.state.HighModel}  
 LabelData={**this**.state.LabelData}  
 />  
</Spin>

#### 选择方位

**调用这个组件，返回出方位信息，然后将方位信息赋值给SetView的type就可以实现方位的变化了。**

**import** TopMenu **from** 'src/Components/BimModule/SimpleViewer/TopMenu/index'

<TopMenu  
 onChoiceView={**this**.handleChoiceView}  
/>

//选择方位  
handleChoiceView = (type)=>{  
 **this**.setState({  
 SetView:{  
 Mode:type,  
 DataTime:Math.random(),  
 Version:3,  
 }  
 })  
};

#### 点击高亮模型

这里有三个地方调用该高亮方法，第一个是点击构建，也就是num为0时，onPickCall返回出该构建信息，要保存起来，因为这里有个操作就是要点击构建后才能进行添加标注。

把data里面的数据添加给HighModel状态，data.ObjGUID为构建guid。

near为0时表示不拉近距离，为1表示拉近距离。

clear为true时表示清除高亮

//高亮模型构建  
onPickCall = (data,num)=>{  
 **let** objData = **undefined**;  
 **let** bool = **false**;  
 **switch** (num) {  
 **case** 0://点击构件  
 objData = data;  
 bool = **false**;  
 **break**;  
 **case** 1://点击标注  
 bool = **false**;  
 **break**;  
 **case** 2://新增成功，清除高亮  
 bool = **true**;  
 **break**;  
 **default**:  
 **break**;  
 }  
 **this**.setState({  
 HighModel:{  
 high:{  
 rows:[{  
 PrjID:data.PrjID,  
 ObjGUIDs:[data.ObjGUID]  
 }],  
 para:{  
 near:num,  
 }  
 },  
 clear:bool,  
 DataTime:Math.random(),  
 Version:3,  
 },  
 objData,  
 });  
};

<ModelViewer  
 LoadModel={**this**.state.LoadModel}  
 LoadFinish={**this**.state.LoadFinish}  
 SetView={**this**.state.SetView}  
 OnPickCall={(data)=>**this**.onPickCall(data,0)}  
 HighModel={**this**.state.HighModel}  
 LabelData={**this**.state.LabelData}  
/>

#### 模型标注

子组件：

显示模型标注的状态为LabelData,在模型加载完后调用，数据用父组件传递下来的数据。

当添加或删除标注后要重新渲染标注，因此要重新调用该方法，在重新请求标注信息成功后调用，数据可调用时直接用参数传递回来。

处理后端返回的所有标注信息，赋值给LabelData里的rows；就可在模型里看到标注红点；

//显示标注（模型方位完成后调用and重新获取标注列表后调用，用有无参数区分）  
handleShowLabel=(data)=>{  
 **let** TopicList = data;  
 **if**(!data){  
 TopicList = **this**.props.TopicList  
 }  
 **let** rows = [];  
 **if**(TopicList.length > 0){  
 TopicList.forEach(item=>{  
 **let** viewinfo = JSON.parse(item.viewinfo);  
 **let** obj = {  
 AddNote:item.addnote,  
 Title:item.title,  
 key:item.guid,  
 ViewInfo:viewinfo  
 };  
 obj.ViewInfo.position = viewinfo.camera.position;  
 rows.push(obj)  
 });  
 **this**.setState({  
 LabelData:{  
 show:{  
 rows:rows,  
 OnPointTo2DCall:**this**.handleOnPointTo2DCall,  
 },  
 DataTime:Math.random(),  
 Version:3,  
 }  
 });  
 }  
};

OnPointTo2DCall可设置回调函数，回调函数在鼠标按住拖动页面的时候就会执行，返回出来的是标注的位置信息，只要标注有移动，就会不断的返回最新的标注位置信息  
  
//返回标注位置信息（鼠标按住移动就会执行）  
handleOnPointTo2DCall = (call)=>{  
 **this**.setState({  
 point:call  
 });  
};

在父组件里添加或删除单个标注时成功后，要先清除所有标注；清除完后再重新请求标注列表信息（getTopicList为重新请求标注列表信息）  
  
//清除所有标注  
deletePoint = ()=>{  
 **this**.setState({  
 LabelData:{  
 clear:**true**,  
 DataTime:Math.random(),  
 Version:3,  
 },  
 point:**undefined** },()=>{  
 **this**.props.getTopicList(**true**)  
 });  
};

把标注用a标签渲染，定位到页面中，标注位置移动，point改变就重新渲染，因此a标签才能跟着动。  
  
//渲染所有的标注  
pointWord = ()=>{  
 **let** {point} = **this**.state;  
 **if**(point !== **undefined**){  
 **let** arr = [];  
 **for**(**let** key **in** point){  
 arr.push(point[key]);  
 }  
 **return** arr.map(item=>{  
 **return** <a  
 style={{position:'absolute',top:item.Point2D.y,left:item.Point2D.x}}  
 className={style.point}  
 key={item.GUID}  
 onClick={()=>**this**.clickPoint(item)}  
 >{item.Title}  
 </a>  
 })  
 }  
};

//点击标注-高亮、靠近  
clickPoint = (item)=>{  
 **this**.onPickCall(item.ViewInfo,1)  
};

{**this**.pointWord()}

父组件：

根据有无参数判断是页面初始请求还是清除标注后的重新请求；

重新请求标注列表信息后，就要重新渲染标注，this.fixModal.handleShowLabel为模型标注渲染

//获取变更标注列表  
getTopicList = (bool)=>{  
 **let** PrjID = **this**.state.PrjID;  
 **let** pcsId = **this**.state.rowArr.guid;  
 api.changeVisaClaim.gettopiclist(PrjID,pcsId).then(res=>{  
 **if**(res.op === "suc"){  
 **this**.setState({  
 TopicList:res.data  
 });  
 **if**(bool){  
 **this**.fixModal.handleShowLabel(res.data)  
 }  
 }  
 })  
};

## 快捷键：

清除缓存：ctrl +F5

打开控制台：F12

WebStorm编辑器--打开查询：Ctrl + F

## 编码规范

（1）对于JSX属性值总是使用双引号("), 其他均使用单引号(')

<Foo bar="bar" />

<Foo style={{ left: '20px' }} />

1. 属性缩进

// 若能在一行中显示, 直接写成一行，一行不用加括号，可直接写在return后面

<Foo bar="bar" />

// good, 有多行属性的话, 新建一行关闭标签，且要用括号包裹

<Foo

superLongParam="bar"

anotherSuperLongParam="baz"

/>

1. 空格

总是在自动关闭的标签前加一个空格

<Foo />

不要在JSX {} 引用括号里两边加空格

<Foo bar={baz} />

能用三个等===，就不要用两个等

if(num === 0){}

如果后端返回的不是一个数字，要么让后端改返回的数据，int，或者在接收数据时自己转为数字。

1. 属性

JSX属性名使用骆驼式风格

<Foo

userName="hello"

phoneNumber={12345678}

/>

如果属性值为 true, 可以直接省略

// bad

<Foo

hidden={true}

/>

// good

<Foo

hidden

/>

// good

<Foo hidden />

避免使用数组的index来作为属性key的值，推荐使用唯一ID

{todos.map(todo => (

<Todo

{...todo}

key={todo.id}

/>

))}

在清楚明白扩展对象时才使用扩展运算符

export default function Foo {

const props = {

text: '',

isPublished: false

}

return (<div {...props} />);

}

尽可能地筛选出不必要的属性

render() {

const { irrelevantProp, ...relevantProps } = this.props;

return <WrappedComponent {...relevantProps} />

}

1. Refs

总是在Refs里使用回调函数

// bad

<Foo

ref="myRef"

/>

// good

<Foo

ref={(ref) => { this.myRef = ref; }}

/>

1. 将多行的JSX标签写在 ()里

// good

render() {

return (

<MyComponent className="long body" foo="bar">

<MyChild />

</MyComponent>

);

}

// good, 单行可以不需要

render() {

const body = <div>hello</div>;

return <MyComponent>{body}</MyComponent>;

}

1. 对于没有子元素的标签来说总是自己关闭标签

<Foo className="stuff" />

1. 如果模块有多行的属性， 关闭标签时新建一行

<Foo

bar="bar"

baz="baz"

/>

1. 使用箭头函数来获取本地变量.

function ItemList(props) {

return (

<ul>

{props.items.map((item, index) => (

<Item

key={item.key}

onClick={() => doSomethingWith(item.name, index)}

/>

))}

</ul>

);

}

1. 箭头函数

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。

var f = () => 5;

var sum = (num1, num2) => num1 + num2;

如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用return语句返回（重要）

var sum = (num1, num2) => { return num1 + num2; }

函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象由于箭头函数没有自己的this，所以也不能用call()、apply()、bind()这些方法去改变this的指向。

Js有个坑，就在当调用函数用：函数名（ ），这时里面的this就会丢失，再在函数里使用this，是为空的。

箭头函数在定义时就默认执行了bind（this）

## 导航栏-点击添加高亮

**//这里导航栏是遍历数组进行渲染，在类名这里，给高亮的类名进行判断，把此时的dataclass与id进行判断；点击的时候把该项的id传递过去，在cut方法里将id赋值给dataclass状态，因此此时的dataclass和id是相同的，判断为true，高亮就会显现。**

**this**.props.nav.map(item => {  
 **return** <li key={item.id}>  
 <a  
 className={classNames(style.single, {[style.singleClick]: **this**.state.dataClass === item.id})}  
 onClick={() => **this**.cut(item.id)}>{item.text}  
 </a>  
 </li>  
})  
}

//导航栏切换  
cut = (id) => {  
 **let** bool = **true**;  
 **if** (id === 1 || id === 2) {  
 bool = **true**;  
 } **else** {  
 bool = **false**;  
 }  
 **this**.getXml(id);  
 **this**.setState({  
 dataClass: id,  
 tableBool: bool  
 }, () => **this**.getHeight())  
};

//这里如果页面中，导航的每个页面之间有些组件显示或者不显示，也可以在切换导航栏的cut方法里进行控制。例如上面的tableBool状态；

若页面需要获取高度，也可以在cut方法里进行获取高度，但要在导航栏切换之后才能获取，因此这里获取高度的函数写在setState的后面，用箭头函数再调用获取高度的函数，否则它不会再setState方法结束后立马执行。

## 获取页面高度

//有些组件需要输入固定的一个数值作为高度，就需要获取页面的高度，可以用clientHeight ：

getHeight = () => {  
 **let** h = document.querySelector(".listHeight").clientHeight - 49 - 62;  
 **this**.setState({height: h})  
};

conWidth: window.innerWidth, //容器宽度  
conHeight: window.innerHeight //容器高度

//在标签里计算宽高

<div

className={styles.task}

style={{minHeight: conHeight,width: `calc(100% - ${diffWidth}px)`}}

>

//在css.less里计算高度

height:calc(100% - 90px);

## 图标控制组件的显示或隐藏

//manBool 的状态控制了组件TableMan的显示或隐藏。因此点击图标的时候要控制manBool 的状态；

//当点击下拉图标的时候，传递0，表示要将组件隐藏，hidden方法里就根据参数为0时，修改状态为manBool为false，此时组件TableMan就隐藏了，同时manBool又控制着图标，此时图标是上拉图标，点击上拉图标后，传递1，manBool变为true，组件又显示，图标又变为下拉图标。

{  
 **this**.state.manBool === **true** ?  
 <i  
 className={style.tu + ' ' + "myAnticon anticon-xia1"}  
 onClick={() => **this**.hidden(0)}  
 /> :  
 <i  
 className={style.tu + ' ' + "myAnticon anticon-shang"}  
 onClick={() => **this**.hidden(1)}  
 />  
}

{  
 **this**.state.manBool === **true** ?  
 <TableMan  
 height={**this**.state.height}  
 dataMan={**this**.state.dataMan}  
 /> : ''  
}

hidden = (num) => {  
 **switch** (num) {  
 **case** 0:  
 **this**.setState({manBool: **false**}, () => **this**.getHeight());  
 **break**;  
 **case** 1:  
 **this**.setState({manBool: **true**}, () => **this**.getHeight());  
 **break**;  
 }  
};

//这里有计算高度，因此在修改完状态后要执行此方法

## 下拉Select使用

//简单使用：只需要设置默认值和defaultValue，选项里必须有属性value，选中时，handleChange里可以获取到执行的选中的那个选项的value值，然后在这个函数里可以根据不同的value值，进行不同的操作，这里就是改变了flag值，而且改变完之后重新获取页面的数据。

<Select  
 defaultValue="biao"  
 onChange={**this**.handleChange}  
>  
 <Option value="biao">标底清单</Option>  
 <Option value="he">合同清单</Option>  
 <Option value="mong">模型清单</Option>  
</Select>

//清单选择  
handleChange = (value) => {  
 **let** flag = 1;  
 **switch** (value) {  
 **case** 'biao':  
 flag = 1;  
 **break**;  
 **case** 'he':  
 flag = 2;  
 **break**;  
 **case** 'mong':  
 flag = 3;  
 **break**;  
 }  
 **this**.setState({flag: flag}, () => {  
 **this**.getXml(**this**.state.dataClass)  
 });  
};

## 上传组件upload--先临时目录-后转真实目录

**//用**Upload 标签把需要点击实现上传的按钮包裹起来，设置参数fileProps，这里用函数返回的方式，action属性为上传的地址，accept为限定上传的文件后缀，onChange为上传后的回调，可以在里面获取上传后文件的信息，showUploadList为禁止显示列表。

//onChange里可以获取到上传成功后返回的数据。可以将需要的信息保存给状态。这里是有一个确认框，要上传到真实目录，因此这里上传成功后，打开一个确认框，点击确认的时候，执行propsXmlTrue 方法，关闭确认框，发送上传请求，此时数据就可以从状态里获取。如果不需要确认框的话，就可以在onChange里直接执行propsXmlTrue ，数据用传参。

**let** fileProps = **this**.propsXml();//xml2.0上传参数

<Upload {...fileProps}>  
 <i className={'myAnticon anticon-daochu'} style={{color: 'orange'}}/>  
 导入XML2.0  
</Upload>

*/\*\*上传至临时目录--------xml2.0\*/*propsXml = () => {  
 **return** {  
 name: 'file',  
 action: '/api/upload/temp',  
 headers: {authorization: 'authorization-text'},  
 accept: '.XML',  
 onChange: (info) => **this**.onChange(info),  
 showUploadList: **false**//禁止显示列表  
 }  
};  
  
//上传验证  
onChange = (info) => {  
 **if** (info.file.status !== 'uploading') {  
 }  
 **if** (info.file.status === 'done') {  
 **let** url = info.file.response.data.url;  
 **this**.setState({  
 url: url,  
 visibleXML: **true** });  
 } **else if** (info.file.status === 'error') {  
 message.error(`${info.file.name} 文件上传失败`);  
 }  
};  
  
*/\*\*临时目录->真实目录---------xml2.0\*/*propsXmlTrue = () => {  
 **this**.setState({visibleXML: **false**});  
 **let** data = {  
 filepath: **this**.state.url,  
 flag: **this**.state.flag,  
 PrjID: JSON.parse(**this**.props.match.params.obj).PrjID,  
 selectIdx: 0  
 };  
 api.detaileList.xmlUpload(data).then(res => {  
 **if** (res.op === 'suc') {  
 message.success('文件上传成功');  
 **this**.getXml(**this**.state.dataClass)  
 }  
 })  
};

## 上传组件upload--一步上传，自定义分成三步

//这里是要将一步上传改成三步上传，需要用到beforeUpload和customRequest。因为第一步点击选择文件后，需要回显数据，因此要在beforeUpload（上传前）获取到文件的名称（用来回显）和文件大小（用来后面的上传），将其保存在状态里。

//然后会接着执行customRequest（上传时），在这里获取到要上传的参数和文件，同样将其保存在状态里，因为这里是要点击另外一个按钮才将其上传，因此先保存在状态里， 然后点击上传按钮的时候，执行excelClick，先验证文件有没有被选中，文件信息是否都具备了，如果已经选择了文件了，则打开确认框，最后在点击确认的时候，执行propsExcelTrue，关闭确认框，发起请求将文件上传，并且将保存的文件信息都清空还原。若点击取消，暂时不用清空这些数据，这样就再次点击上传按钮再次上传。但如果选择了文件后，没有上传，在关闭这个modal的时候，最好还是将保存文件的所有数据都清空，避免关闭后再打开modal，里面的数据还在。

**import** request **from** 'Res/js/request';

*/\*\*上传excel\*/*

propsExcel = () => {

**let** data = {  
 flag: **this**.state.flag,  
 op: op  
 };  
 **return** {  
 name: 'file',  
 action: '/servlet/WebUploadServlet',  
 data: {  
 action: 'importExcel',  
 md5code: **this**.state.excelSize,  
 data: JSON.stringify(data),  
 PrjID: JSON.parse(**this**.props.match.params.obj).PrjID,  
 },  
 headers: {authorization: 'authorization-text'},  
 accept: '.xlsx,.xls',  
 beforeUpload: **this**.beforeUpload,  
 customRequest:**this**.customRequest,  
 showUploadList: **false**//禁止显示列表  
 }  
};  
  
//上传前：获取文件的size和name  
beforeUpload = (file) => {  
 console.log(file);  
 **this**.setState({  
 excelSize: file.size,  
 excelName: file.name  
 });  
};  
  
  
//上传时：获取上传所需的路径和参数  
customRequest = ({action,data,file})=> {  
 console.log(action);  
 console.log(data);  
 **const** formData = **new** FormData();  
 formData.append('action',data.action);  
 formData.append('excel',file);  
 formData.append('md5code',data.md5code);  
 formData.append('data',data.data);  
 formData.append('PrjID',data.PrjID);  
 **this**.setState({  
 action: action,  
 formData: formData  
 });  
};  
  
//点击-上传清单：打开确认对话框  
excelClick = () => {  
 **if** (**this**.state.excelName === **undefined**) {  
 message.warn('请先选择微软的Excel文件,即xls或者xlsx后缀的文件')  
 } **else** {  
 **this**.setState({visibleExcel: **true**})  
 }  
};  
  
//点击ok:上传excel  
propsExcelTrue = () => {  
 console.log('这里是上传excel的接口');  
 **let** {action,formData} = **this**.state;  
 request(action,{method:"post",body:formData},(res)=>{  
 console.log(res);  
 **if**(res.op === 'suc'){  
 message.success('上传成功');  
 **this**.getXml(**this**.state.dataClass)  
 }  
 },(err)=>{  
 console.log(err);  
 message.error('上传失败')  
 });  
 console.log('上传结束');  
 **this**.setState({  
 visibleExcel: **false**,  
 action: **undefined**,  
 formData: **undefined**,  
 excelSize: **undefined**,  
 excelName: **undefined** })  
  
};

这里注意把关闭modal的方法放在请求外面，或者放在请求上面，不要放在里面，这样就可以刚点击的时候就关闭modal，不用等请求成功后才关。

## 下载excel

//下载excel，首先要请求接口，获取到下载文件的信息，再自己设置文件名称，因为中间有个下载小窗，而不是一步下载，所以需要把那些数据全都保存在状态里。

//这里下载有个小窗口，在上一步请求成功，打开小窗口，回显这个文件的信息，然后在点击小窗里的下载按钮的时候，执行download ，关闭小窗，创建a标签，将保存在状态里的文件地址和文件名称，赋值给a标签的属性'href'和'download'，方法如下：

*/\*\*下载模板-excel\*/*template = () => {  
 **let** data = {  
 action: **this**.state.tableBool ? 'fbfx' : 'PCSOtherList',  
 op: "download"  
 };  
 **let** data1 = {  
 DataClass: **this**.state.dataClass,  
 typec: "excel",  
 temp: **true** };  
 api.detaileList.template(data, data1).then(res => {  
 console.log(res);  
 **let** navType = '';//dataClass  
 **this**.props.nav.forEach(item => {  
 **if** (item.id === **this**.state.dataClass) {  
 navType = item.Menu  
 }  
 });  
 **let** name = `${navType}-Excel模板.xlsx`;  
 **this**.setState({  
 downloadTemplate: **true**,  
 webUrl: res.webUrl,//地址  
 fileName: name,//名称  
 fileSize: (res.length / 1024).toFixed(2)//大小  
 });  
 })  
};  
  
*/\*\*点击下载excel\*/*download = () => {  
 **this**.setState({  
 visibleUpLoad: **false**  
});  
 **const** ele = document.createElement('a');  
 //代码上传，把下面的域名去掉  
 ele.setAttribute('href', `http://192.168.1.234:9080${**this**.state.webUrl}`); //设置下载文件的url地址  
 ele.setAttribute('download', **this**.state.fileName);//用于设置下载文件的文件名  
 document.body.appendChild(ele);  
 ele.click();  
 document.body.removeChild(ele);  
 message.success('下载成功');  
 setTimeout(() => {  
 **this**.getHeight()  
 }, 300)  
};

//若页面样式有变，需要重新获取页面样式。

## 组件间数据传递

//该组件要获取到TableOne组件里面的数据。先把一个方法getManCaiJi 传递给TableOne组件，然后在TableOne组件里，当需要的数据获取到之后，执行**this**.props.getManCaiJi(data)，此时父组件的getManCaiJi 方法里就可以获取到data，然后就可以保存在父组件的状态里。这里还可以通过传递[],父组件可以判断获取到的数据是否为空，从而赋值给状态不同的值。

<TableOne  
 getManCaiJi={**this**.getManCaiJi}  
 ref={(ref) => {  
 **this**.one = ref  
 }}  
/>

//获取人材机数据  
getManCaiJi = (data) => {  
 **if** (data.length > 0) {  
 **this**.setState({  
 dataMan: data  
 })  
 } **else** {  
 **this**.setState({  
 dataMan: **undefined** })  
 }  
};

TableOne:

//点击定额获取人材机  
**if**(record.NodeType === 5){  
 **let** data={  
 DataClass:record.DataClass,  
 fbfxID:record.GUID,  
 prjID:**this**.props.PrjID  
 };  
 api.detaileList.getMan(data).then(res=>{  
 **let** data = JSON.parse(res.data);  
 **this**.props.getManCaiJi(data)  
 })  
}**else**{  
 **this**.props.getManCaiJi([])  
}

## 动态加载组件

//导航切换改变状态dataclass的值，这里根据dataClass加载不同组件，若组件只占一行，不需要括号；但如果组件占了多行，需要加括号，且组件换行显示。

//这里的ref格式如下，父组件就可以通过this.ref使用子组件的状态和方法

//根据导航栏动态加载组件  
tableLarge = () => {  
 **switch** (**this**.state.dataClass) {  
 **case** 1:  
 **return** <TableOne ref={(ref) => {**this**.one = ref}}/>  
 );  
 **case** 2:  
 **return** (  
 <TableTwo  
 getXml={**this**.getXml}  
 ref={(ref) => {**this**.two = ref}}  
 />  
 );  
 **case** 5:  
 **return** (  
 <TableFour  
 height={**this**.state.height}  
 data={**this**.state.data5}  
 />  
 );  
 **default**:  
 **return** <TableOne height={**this**.state.height}/>;  
 }  
};

## 递归：

### 例一：计算子组件数据的和，回显在父组件里

//当要计算子组件里的数据时，要先判断有数据，且不为空，这里数据是从父组件传递下去的，所以直接在这里判断

//再判断子组件已经渲染，再调用里面的方法。

<TableThree ref={(ref) => {**this**.Three = ref}}/>

//获取后面三项的合计金额  
getAll = () => {  
 **if** (**this**.state.dataClass === 3 && **this**.state.data3 !== **undefined**) {  
 **if** (**this**.state.data3.length > 0) {  
 **let** all = 0;  
 //确保组件three已经渲染  
 **if** (**this**.Three !== **null**) {  
 **this**.state.data3.forEach(item => {  
 all += **this**.Three.setAmount(item);  
 });  
 }  
 **return** all  
 }  
 }

TableThree :

//要计算每一个数据Amount都为children的Amount之和。数据里，有些有children，有些没有，而children也有为空的，因此在判断的时候就得判断有children且不为空。

//传进去每一条数据（一行数据），判断出有children且不为空后，先申明一个变量：Amount = 0，然后遍历children数组，将children所有的子元素的Amount累计相加，在遍历外，要return出累计相加后的值。但在累计相加之前，加个判断，若判断出children的这个子元素还有children的话，则需要自己调用自己，将这个children的Amount 等于这个children的children里的每一个子元素的和。然后依次类推下去。直到再没有满足这个判断的，然后就开始执行累加。此时就是先累加最底层，然后一步步往上累加，最终返回出来的值就是底下所有的children的Amount 之和。

//用递归计算合计金额  
setAmount = (row)=>{  
 //有children的计算出children的和  
 **if**(row.children !== **undefined** && row.children.length > 0){  
 **let** Amount = 0;  
 **for**(**let** i=0;i<row.children.length;i++){  
 //若底下还有children,调用自己，获取返回值再求和，依此循环下去  
 **if**(row.children[i].children !== **undefined** && row.children[i].children.length > 0){  
 row.children[i].Amount = **this**.setAmount(row.children[i]);  
 }  
 Amount += row.children[i].Amount;  
 }  
 **return** Amount  
 }  
 //无children直接返回金额  
 **else**{  
 **return** row.Amount  
 }  
};

### **例二：动态渲染节点（不知层级）**

renderTreeNodes = data =>  
 data.map(item => {  
 **if** (item.children) {  
 **return** (  
 <TreeNode title={item.title} key={item.key} dataRef={item}>  
 {**this**.renderTreeNodes(item.children)}  
 </TreeNode>  
 );  
 }  
 **return** <TreeNode {...item} />;  
 });

### **例三：数组对象去重，去重前进行数据计算后再去重**

去重前：

this.data = [  
 {  
 key:'1',  
 amount:111,  
 used:10,  
 children:[  
 {key:'1-1',amount:21,used:9},  
 {key:'1-2',amount:22,used:11}  
 ]  
 },  
 {  
 key:'1',  
 amount:44,  
 used:4,  
 children:[  
 {key:'1-1',amount:21,used:9},  
 {key:'2-2',amount:13,used:7}  
 ]  
 }  
]

去重：

componentWillMount(){  
 console.log(this.event(this.data,1))  
}  
//递归计算-第一层，相同key值的，将amount相加，赋值给前一个amount，将used相加，也是赋值给第一个，然后将后一个删除，在删除前，若相同key有children，则将两个children合并（一个有一个没有，则给没有的那个赋值空数组），则要将合并后children再进行上述的操作，因此再次调用自己，将children传给自己。这里used相加只需要在第一层，因此这里可以用count参数来判断是第几层；  
// 注意，最后要return出需要的数据，这样最终才能获取到处理完的数据  
event = (data,count)=>{  
 for(let *i* = 0;*i*<data.length;*i*++){  
 for(let *j* = data.length - 1;*j* > *i*;*j*--){  
 if(data[*i*].key === data[*j*].key){  
 data[*i*].amount = data[*i*].amount + data[*j*].amount;  
 if(count === 1){  
 data[*i*].used = data[*i*].used + data[*j*].used;  
 }  
 if(data[*j*].children && data[*j*].children.length > 0){  
 if(!data[*i*].children){  
 data[*i*].children = []  
 }  
 data[*i*].children = [...data[*i*].children,...data[*j*].children];  
 if(data[*i*].children.length > 0){  
 this.event(data[*i*].children,count+1)  
 }  
 }  
 data.splice(*j*,1)  
 }  
 }  
 }  
 return data  
};

去重后：

[{

amount: 155,

key: "1",

used: 14,

children:[

{key: "1-1", amount: 42, used: 9},

{key: "1-2", amount: 22, used: 11},

{key: "2-2", amount: 13, used: 7}

]

]

### **例四：树形数据加key值，第一层计算一下数据**

//在第一层：给每个对象加key值，按照type分类，计算不同的数据，当有children时，调用自己，但第二层只需要给每个对象加key值即可，不需要再进行数据计算  
handleUser = (data,type,count)=>{  
 if(data.length > 0){  
 data.forEach(item=>{  
 item.key = this.mathRande();  
 if(count === 1){  
 switch (type) {  
 case 1://分部  
 case 2://子分部  
 case 4://工作内容  
 item.Useage = item.UserAmount;  
 break;  
 case 3://清单  
 case -100://直接条目  
 item.Useage = item.Useage/100;  
 break;  
 default:  
 return null  
 }  
 }  
 if(item.children && item.children.length > 0){  
 this.handleUser(item.children,type,count+1)  
 }  
  
 });  
 return data  
 }  
}

## 手动控制table树形结构的展开-按层级

//按照层级来展开，点击按钮1，展开第一层，也就是不展开，点击按钮2，展开3展开3层...这里演示到第五层：

//首先要将每一层的所有有children的元素全部提取出来组成新的数组，例如data1 ，data2在data1的基础上，首先用forEach遍历数组data1，每一项item.children也是一个数组，用filter过滤每一个item.children，筛选出item.children底下所有有children的元素组成新的数组a,有多少个item.children就有多少个a，把每一个a用concat连接起来，这样就把第二层的所有带children的元素组成了一个新的数组。以此类推.....

//最后取这些数组的key，用map遍历这些数组，获取到每一层的所有带children的key值。每一层的key获取完之后，若要展开2层，就把一层和二层key数组连接，若要展开4层，则把一层、二层、三层、四层的数组都连接，再赋值给状态arr.

//最后：expandedRowKeys={**this**.state.arr}

//onExpandedRowsChange={**this**.change}，这个属性设置后，可以手动展开，方法change里的参数就是手动展开后，所需的全部key数组，因此就要在这里将手动展开后返回的key数组赋给状态arr。

<div>  
 <Button onClick={()=>{**this**.event(1)}}>第一层</Button>  
 <Button onClick={()=>{**this**.event(2)}}>第二层</Button>  
 <Button onClick={()=>{**this**.event(3)}}>第三层</Button>  
 <Button onClick={()=>{**this**.event(4)}}>第四层</Button>  
 <Button onClick={()=>{**this**.event(5)}}>第五层</Button>  
 <div ref={'box'}>  
 <Table  
 columns={columns}  
 dataSource={**this**.state.data}  
 expandedRowKeys={**this**.state.arr}  
 onExpandedRowsChange={**this**.change}  
 indentSize={5}  
 />  
 </div>  
 </div>

event = (num)=>{  
 **let** {data} = **this**.state;  
  
 //所有带‘children’属性的第一层组成新数组  
 **let** data1 = data.filter((item)=>{  
 **return** item.children !== **undefined** });  
  
 //所有带‘children’属性的第二层组成新的数组  
 **let** data2 = [];  
 data1.forEach((item)=>{  
 **let** a = item.children.filter((it)=>{  
 **return** it.children !== **undefined**;  
 });  
 data2 = data2.concat(a);  
 });  
  
 //所有带‘children’属性的第三层组成新的数组  
 **let** data3 = [];  
 data2.forEach((item)=>{  
 **let** a = item.children.filter((it)=>{  
 **return** it.children !== **undefined**;  
 });  
 data3 = data3.concat(a);  
 });  
  
 //所有带‘children’属性的第四层组成新的数组  
 **let** data4 = [];  
 data3.forEach((item)=>{  
 **let** a = item.children.filter((it)=>{  
 **return** it.children !== **undefined**;  
 });  
 data4 = data4.concat(a);  
 });  
  
  
 //第一层key数组（展开二层）  
 **let** arr1 = data1.map((item)=>{  
 **return** item.key  
 });  
 //第二层key数组（展开三层）  
 **let** arr2 = data2.map((item)=>{  
 **return** item.key  
 });  
 //第三层key数组（展开四层）  
 **let** arr3 = data3.map((item)=>{  
 **return** item.key  
 });  
 //第四层key数组（展开五层）  
 **let** arr4 = data4.map((item)=>{  
 **return** item.key  
 });  
  
 **switch** (num) {  
 **case** 1:  
 **this**.setState({  
 arr:[]  
 });  
 **break**;  
 **case** 2://  
 **this**.setState({  
 arr:arr1  
 });  
 **break**;  
 **case** 3://展开三层  
 **this**.setState({  
 arr:arr1.concat(arr2)  
 });  
 **break**;  
 **case** 4://展开四层  
 **this**.setState({  
 arr:arr1.concat(arr2,arr3)  
 });  
 **break**;  
 **case** 5://展开五层  
 **this**.setState({  
 arr:arr1.concat(arr2,arr3,arr4)  
 });  
 **break**;  
 **default**://默认展开一层  
 **this**.setState({  
 arr:[]  
 });  
 **break**;  
 }  
 };  
 change = (expandedRows)=>{  
 **this**.setState({  
 arr:expandedRows  
 });  
 };

## 手动控制table树形结构的展开-按指定数据

//点击的时候执行方法showTo，传递参数1：

<Menu.Item key="1" onClick={() => {**this**.showTo(1)}}>分部</Menu.Item>

...

//方法showTo接收到参数后，调用组件TableOne的方法showTO，并将参数传递过去。这里在调用子组件方法时，同样需判断数据不为空，且子组件已经渲染。  
showTo = (num) => {  
 **if** (**this**.state.dataClass === 1 && **this**.state.data1 !== **undefined**) {  
 //确保组件one已经渲染  
 **if** (**this**.one !== **null**) {  
 **this**.one.showTO(num);  
 }  
 }  
 };

<TableOne  
 data={**this**.state.data1}  
 ref={(ref) => {**this**.one = ref}}  
/>

TableOne

//属性expandedRowKeys，将要展开的key数组传进去，表格会展开所有的key的数据，

//属性onExpandedRowsChange，当手动将其展开会收缩时，change方法里的参数就为手动后的展开的key数组。要将其赋值给状态arr

<Table  
 columns={columnsaa}  
 dataSource={**this**.props.data}  
 pagination={**false**}  
 rowKey={record =>record.GUID}  
 onRow={**this**.onRow}  
 expandedRowKeys={**this**.state.arr}  
 onExpandedRowsChange={**this**.change}  
/>

//手动展开收缩时，动态修改arr  
change = (expandedRows)=>{  
 **this**.setState({  
 arr:expandedRows  
 });  
};

//接收到参数之后，根据需求传递不同的参数给方法show，获取到所需要的key数组后，将其赋值给状态arr，变能展开对应的key数据

showTO = (num)=>{  
 **let** data = **this**.props.data;  
 **let** showArr = [];  
 **switch** (num) {  
 **case** 1://展开分部和子分部  
 showArr = **this**.show(data,1,2);  
 **break**;  
 **case** 2://展开清单和直接条目  
 showArr = **this**.show(data,3,-100);  
 **break**;  
 **case** 3://展开工作内容  
 showArr = **this**.show(data,4,0);  
 **if**(showArr.length === 0){  
 showArr = **this**.show(data,3,-100);  
 }  
 **break**;  
 **case** 4://展开定额  
 showArr = **this**.show(data,5,0);  
 **break**;  
 }  
 **this**.setState({  
 arr:showArr  
 });  
};

//因为数据中每一层数据的结构都是一样的，可以用递归来查找：首先要传递的参数为data，因为每一次调用自己都是对不同数据进行查找，因此每次都需要调用都必须传入data，所以这个参数是必须要有的；后面的num1和num2是不变的，是查找的条件，因此调用自己的时候也就把num1和num2传递进去即可。

//首先声明一个空数组arr，判断数据不为空后，这里用for循环遍历数组，筛选出有children的元素，然后再次再嵌套一个循环，遍历有children元素的children的属性值（数组）。此时data[i].children[j]是第二层的数据，然后要进行条件判NodeType

断，如果符合条件，则要把第一层的数据guid取出来：arr.push(data[i].GUID)（因为要展开二层，只需要获取到一层的key即可）

//这里在二层循环里要写个break，因为一层有M个带children的元素，就有M个二层循环，这个二层循环，每一次循环，只要找到第一个满足条件，就可以跳出循环了，因为要验证的一层数据的二层有没有满足条件的，二层满足条件，取的是一层的guid,因此只要二层有满足的，就不用再找了，已经确认这个一层的guid是可以取的。此时就可以跳出这个循环了。然后开始第二个二层循环。同样满足条件就跳出循环，开始第三个二层循环...

//这里，在找第二层的数据是否有满足NodeType的条件之前，若第二层的数据还有children属性（有第三层数据）,此时就可以自己调用自己，将每一个有第三层数据的整个二层数据（1/M,上面的一个循环的数据）都传递给自己，这样直到最底层已经没有children为止，然后就继续往下执行，往下就是条件判断取guid,而且会一层层往上判断，只要满足条件的guid,都将被取出来。再通过concat连接，最终整个数据完成后传递出来的数据就是所有满足条件的guid数组。

//上面得到的guid数组不完善，还需要获取到每一个guid的所有父元素的guid,因此递归show 里需要调用另一个递归来获取每一个guid所有的父元素的guid

//递归:找出children符合NodeType的父元素的GUID  
show = (data,num1,num2)=>{  
 **let** arr=[];  
 **if**(data !== **undefined**){  
 **for**(**let** i=0;i<data.length;i++){  
 **if**(data[i].children !== **undefined** && data[i].children.length > 0){  
 **for**(**let** j=0;j<data[i].children.length;j++){  
 **if**(data[i].children[j].children !== **undefined** && data[i].children[j].children.length > 0){  
 **let** str = **this**.show(data[i].children,num1,num2);  
 arr = arr.concat(str);  
 }  
 **if**(data[i].children[j].NodeType === num1 || data[i].children[j].NodeType === num2){  
 arr.push(data[i].GUID);//这里：发现孩子满足条件，则直接取父元素的guid  
 **if**(data[i].ParentGUID !== '-1'){  
 arr.push(data[i].ParentGUID);//将guid的父组件再给arr，然后再查找更上一级的父组件  
 **let** parentGuid = **this**.parent(**this**.props.data,data[i].ParentGUID);  
 arr = arr.concat(parentGuid);  
 }  
 **break** }  
 }  
 }  
 }  
 **let** arr1 = Array.from(**new** Set(arr));  
 **return** arr1  
 }  
};

//这里有两个变量，每次需要传入不同的data和不同的guid,因此两个参数都是必要要有的。首先就是遍历传进去的一层数组，判断筛选出有children（二层数组）的一层数组，然后将这些一层数组的children（二层数组）再次传递给自己，然后每次判断有children，就把children传给自己，直到最底层没有children，然后开始往下执行，且此时一层层往上执行。

//下面的代码从最底层开始一层层往上执行，满足条件判断，aaa.push(data[i].GUID)，当它有ParentGUID的时候，再把ParentGUID传给自己，然后再所有数据里再进行查找这个guid,找到则取出guid，然后再把ParentGUID再传给自己，直到最上面的那层，就停止调用自己，然后不断连接返回出来的数据。最终得到的就是最初的guid的所有父元素的guid数组

//递归：先找到上面传入的guid，再找出它所有的父元素  
parent = (data,guid)=>{  
 **let** aaa = [];  
 **for**(**let** i=0;i<data.length;i++){  
 //如果有孩子，把孩子数组给自己，再进行查找  
 **if**(data[i].children !== **undefined** && data[i].children.length > 0){  
 **let** st = **this**.parent(data[i].children,guid);  
 aaa = aaa.concat(st);  
 }  
 //找到后，如果这个有父元素，将父元素guid传给自己在所有数据中进行查找  
 **if**(data[i].GUID === guid){  
 aaa.push(data[i].GUID);  
 **if**(data[i].ParentGUID !== '-1'){  
 aaa.push(data[i].ParentGUID);  
 **let** st2 = **this**.parent(**this**.props.data,data[i].ParentGUID);  
 aaa = aaa.concat(st2);  
 }  
 }  
 }  
 **return** aaa  
};

## 下拉菜单的使用

**//此时的效果为：点击**Dropdown 里面的元素，Menu出现。

**let** menu2 = (  
 <Menu>  
 <Menu.Item key="1" onClick={() => {**this**.showTo(1)}}>分部</Menu.Item>  
 <Menu.Item key="2" onClick={() => {**this**.showTo(2)}}>清单</Menu.Item>  
 <Menu.Item key="3" onClick={() => {**this**.showTo(3)}}>工作内容</Menu.Item>  
 <Menu.Item key="4" onClick={() => {**this**.showTo(4)}}>定额</Menu.Item>  
 </Menu>  
);

<Dropdown overlay={menu2} trigger={['click']}>  
 <a className="ant-dropdown-link" href="#">  
 <span className={style.word}>显示至</span>  
 <Icon type="caret-down" style={{fontSize: '10px', color: '#999'}}/>  
 </a>  
</Dropdown>

## 在render里遍历出jsx

{**this**.props.nav.map(item => {  
 **return** <li key={item.id}>  
 <a  
 className={classNames(style.single, {[style.singleClick]: **this**.state.dataClass === item.id})}  
 onClick={() => **this**.cut(item.id)}>{item.text}  
 </a>  
 </li>  
})  
}

## render里三元判断

{  
 **this**.state.tableBool === **true** ?  
 <div className={style.select}>  
 </div> : ''  
}

## 滚动事件

//这里注意：给节点绑定滚动事件时，这个节点若为window，则是整个页面滚动才有效果，所以如果要让页面下的某一个盒子有滚动事件，就必须绑定在节点上，例如下面的.box。然后要在卸载组件前将事件移除。

handleScroll 方法就能在滚动的时候执行

//滚动事件  
<div className="box">  
 <div className="one">顶部</div>  
 <div className="two">内容</div>  
</div>

ComponentDidMount(){  
 document.querySelector(".box").addEventListener('scroll', **this**.handleScroll.bind(**this**));  
  
}  
componentWillUnmount(){  
 document.querySelector(".box").removeEventListener('scroll', **this**.handleScroll.bind(**this**))  
}  
  
handleScroll = ()=>{  
 **let** scrollTop = document.querySelector(".two").getBoundingClientRect().top;  
 console.log(scrollTop);  
};

//下面的需求是：将绝对定位的元素跟随页面进行滚动。scrollTop可以获取到绑定了滚动事件元素的滚动距离，然后就可以动态设置要跟随滚动的top值，前提是这个元素得有定位。

//备注：overflow：hidden对于决定定位的元素是没有作用的，超出后不能将其隐藏。

//滚动事件  
componentDidMount(){  
 document.querySelector(".newPage").addEventListener('scroll', **this**.handleScroll.bind(**this**));  
  
}  
componentWillUnmount(){  
 document.querySelector(".newPage").removeEventListener('scroll', **this**.handleScroll.bind(**this**))  
}  
  
handleScroll = ()=>{  
 //获取到页面滚动的距离  
 **let** newPage = document.querySelector(".newPage").scrollTop;  
 **let** select = document.querySelector(".ant-select-dropdown");  
 **let** time = document.querySelector(".ant-calendar-picker-container");  
 **if**(select !== **null**){  
 **let** top = 129-newPage;  
 select.style.top = `${top}px`;  
 }  
 **if**(time !== **null**){  
 **let** top1 = 98-newPage;  
 time.style.top = `${top1}px`;  
 }  
};

## 键盘事件

<Button  
 onClick={() => **this**.listLoading(2)}  
 onKeyUp={**this**.doSomething}  
>查询</Button>

componentDidMount(){  
 window.addEventListener('keydown',**this**.doSomething)

};  
componentWillUnmount(){  
 document.removeEventListener("keydown",**this**.doSomething);  
};  
doSomething = (e)=>{  
 **const** code = e.keyCode || e.which || e.charCode;  
 **if**(code === 13){  
 **this**.listLoading(2)  
 }  
};

//jq的键盘事件

$(document).on('keydown', '#verifyDiv', **function**(e) {  
 **if**(e.keyCode === 13){  
 confirePwdBtnClick()  
 }  
});  
$("#confirePwdBtn").click(**function**(){  
 confirePwdBtnClick()  
});



## 请求json文件

//json文件返回的数据并不是标准的返回数据格式，因此这里用不了httpFetch。

fetch('/config/billcode.json',{  
 method:'get',  
 credentials: 'include',  
}).then(res=>{  
 **if**(res.ok){  
 **return** res.json()  
 .then(res=>{

console.log(res)  
 })  
 }  
})

## 查询-自己筛选数据

**//因为是自己进行筛选数据，因此需要一份原始数据和一份筛选后的数据，所以在保存数据给状态的时候，可以保存给两个状态**

**this**.setState({  
 dataCode:aa,  
 dataCodeFil:aa,//用于筛选  
 GUID:row.GUID,  
})

## 路由传参

//路由跳转设置：

<Route path="/ListMaintain/:obj" component={ListMain}/>

**//通过路由传参：传对象**

**let** obj = {  
 PrjID:'xxxx',  
 agreeGUID:'xxxxx'  
};  
window.open(`/ListMaintain/${JSON.stringify(obj)}`)

//获取到路由上面的参数：

Let PrjID=JSON.parse(**this**.props.match.params.obj).PrjID  
let agreeGUID=JSON.parse(**this**.props.match.params.obj).agreeGUID

//若是单独传一个字符串：

`/ListMaintain/${PrjID}`

则获取只需要：

Let PrjID=**this**.props.match.params.obj

## 选择下拉定位（跟随页面滚动）

//getPopupContainer属性：选择定位的父元素，一般该下拉框要跟随哪个容器滚动，就将其作为自己的父元素。且父元素要有position:relative;就能实现下拉框跟随页面滚动

//RangePicker时间用getCalendarContainer属性，方法是一样的。

<Select  
 getPopupContainer={triggerNode => triggerNode.parentNode.parentNode.parentNode}  
>  
 <Option value='0'>全部</Option>

............................  
</Select>

## table动态生成表头

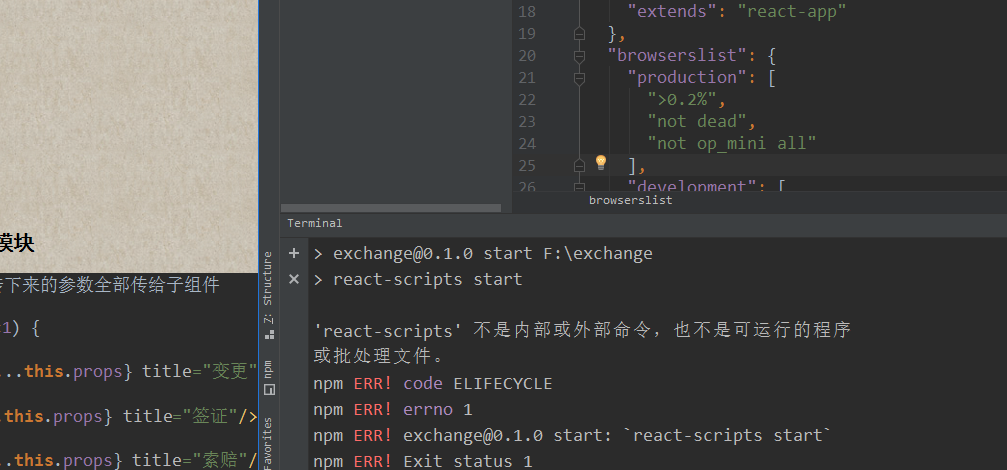
**//要根据返回的数据动态的生成表格的表头：**

**首先要遍历表体数组的一个对象，因为表头的dataIndex要从里面取。将表体数组的一个对象的属性提取出放到数组title里，然后就是遍历hender数据生成columns，这里map里面的ind刚好对应title里的index，因此这里就可以直接用。（前提是顺序不能错）。若是hender对象的排列顺序跟body的对象的字段从上到下的顺序不一样，就得用映射的方法，将hender的每一个对象要对应到body的对象的每一个字段。**

**let** hender =[  
 {title:"姓名1111111",encode:"8748463",content:**null**},  
 {title:"年龄22222",encode:"67464",content:**null**},  
 {title:"手机号333",encode:"6465856",content:**null**}  
 ];  
**let** body = [  
 {  
 "encode":"56146",  
 "name":"小红",  
 "content":"小红的内容",  
 "age":"222"  
 }];  
**let** title = [];  
**for**(**var** i **in** body[0]) {  
 **if**(i === 'encode'){  
 **continue** }  
 title.push(i)  
}  
console.log(title);  
**let** columns = hender.map((item,ind)=>{  
 **let** obj = {  
 title: item.title,  
 dataIndex: title[ind],  
 key: item.encode,  
 };  
 **return** obj  
});  
console.log(columns);

# **报错处理**

## React运行错误：



使用create-react-app创建的React项目时，有时在安装完其他组件后，再次运行 npm start 命令时会报以上错误，让我很郁闷，不过在上网搜了这个错后原来：

是create-react-app有丢包的缺陷，手动安装包后，需要重新npm install一下，这样node\_modules/.bin/目录下才会重新出现react-scripts的文件，这样npm start命令才能正常执行。

解决方案：

npm install react-scripts

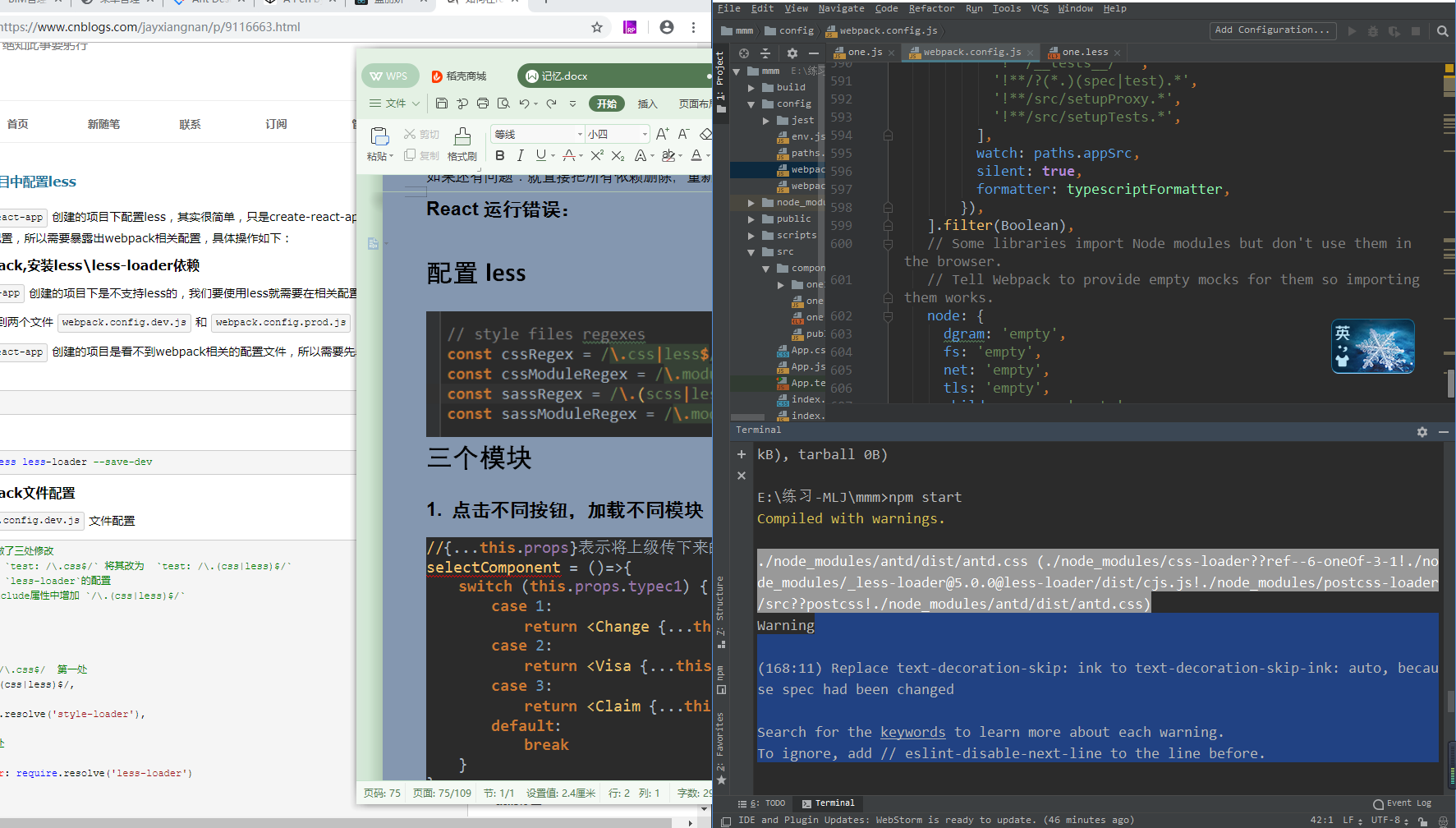
或

npm install

安装完成后再次运行 npm start 即可

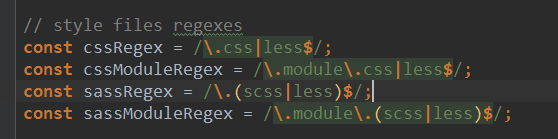
如果还有问题：就直接把所有依赖删除，重新

## **React运行错误：**



解决方法：cnpm install antd --save

# **配置less**



# 三个模块

## 点击不同按钮，加载不同模块

//{...**this**.props}表示将上级传下来的参数全部传给子组件

selectComponent = ()=>{  
 **switch** (**this**.props.typec1) {  
 **case** 1:  
 **return** <Change {...**this**.props} title="变更"/>;  
 **case** 2:  
 **return** <Visa {...**this**.props} title="签证"/>;  
 **case** 3:  
 **return** <Claim {...**this**.props} title="索赔"/>;  
 **default**:  
 **break** }  
};  
render(){  
  
 **return** (  
 <div>  
 {**this**.selectComponent()}  
 </div>  
 )  
}

## 文件夹的安排

//这里的home组件是多个模块共用的，但是在home组件里的apply组件又是不一样的，如果在home组件里对不同apply进行区分的话，home会很累赘，因此这里可以把对不同apply组件的区分跟调用home放在一起。

<Home  
 visibleApply={**this**.visibleApply}  
 {...**this**.props}  
 ref={home=>**this**.home = home}  
/>  
{/\*申请操作\*/}  
<Modal  
 title={**this**.state.applyTitle}  
 visible={**this**.state.visibleApply}  
 width={'100%'}  
 style={{minWidth:1040}}  
 onCancel={**this**.closeApply}  
 maskClosable={**false**}  
 mask={**false**}  
 destroyOnClose  
 footer={**null**}  
 wrapClassName={'applyAudit sameStyle'}  
>  
 <Apply  
 {...**this**.props}  
 closeApply={**this**.closeApply}  
 applyType={**this**.state.applyType}  
 rowArr={**this**.state.rowArr}  
 />  
</Modal>

//一般若顶部有多个按钮，点击不同按钮跳转不同页面时，就把按钮放在父级js里，每一个页面为一个子级js。

## 请求列表数据

//回显表格数据

如果返回值有数据，则把需要用到的数据赋值给状态，无数据，则需要把这些状态清空，因为这个方法需要经常被调用  
getDate = ()=>{  
 **let** PrjID = **this**.state.PrjID;  
 **let** data = {  
 accountType:**this**.state.typec1,//清单类型 1 变更台账 2-签证台账 3-索赔台账  
 projectId:**this**.state.PrjID,  
 };  
 api.changeVisaClaim.getdDate(data,PrjID).then(res=>{  
 **let** tableData = res.data.processChangeManageDOList;  
 **if**(res.op === "suc"){  
 **if**(tableData !== **null**){  
 tableData.forEach((item,index)=>{  
 **return** item.num = index + 1  
 });  
 **this**.setState({  
 tableData,  
 chartCVCVOList:res.data.chartCVCVOList,  
 rowId:'',  
 rowobj:**undefined**,  
 })  
 }**else**{  
 **this**.setState({  
 tableData:**undefined**,  
 chartCVCVOList:**undefined** })  
 }  
 }  
 })  
};

//若modal有输入框，一般在关闭modal的时候，要将其清空。

请求数据进行回显的时候，若字段只是回显，不需要提交，则不需要单独的状态字段，直接把整个数据给一个状态字段，然后从这个字段里取值。

若页面有要进行提交操作的字段，则回显时每一个回显在页面上的字段都需要单独的状态字段来存放。

若该方法被多次调用，且调用后并不是改变页面所有的数据吗，只是改变少数的页面字段，那么这些字段要用单独的字段来回显。在调用时传递参数，然后在方法里，根据参数进行区分是哪种调用，然后改变需要对应的字段即可。

## 操作表格

结构：{**this**.props.handle.map(item=>{  
 **return** <li key={item.id}>  
 <a  
 className={classNames(style.black)}  
 onClick={()=>**this**.click(item.id)}  
 >{item.text}</a>  
  
 </li>  
})}

方法：

//点击顶部按钮：

新增和编辑一般用同一个modal框，新增，直接打开modal框即可，编辑要回显，而且要先判断表格中是否有高亮的行，这里还要判断state状态为0时才有编辑权限，因此在点击行进行高亮的时候，除了保存该条数据的guid,也要保存整条数据，然后将需要回显的数据赋值给状态，回显的地方就回显这些状态。

删除也是一样，

click = (id)=>{  
 **let** {rowId,PrjID,rowobj,title} = **this**.state;  
 **switch** (id) {  
 **case** 1://新增  
 **this**.setState({  
 handleId:id,  
 addText:`新增${title}`,  
 addVisible:**true** });  
 **break**;  
 **case** 2://编辑  
 **if**(rowId === ''){  
 message.warn("请先选择一行")  
 }**else if**(rowobj.state === 0){  
 **this**.setState({  
 handleId:id,  
 addText:'编辑',  
 addVisible:**true**,  
 number:rowobj.code,//单号  
 declareTime:JSON.parse(rowobj.otherdata).declareTime,//申报日期  
 content:rowobj.applycontent,//内容  
 reason:rowobj.applyreason,//理由简述  
 })  
 }**else**{  
 message.warn("非申请状态，不可编辑")  
 }  
 **break**;  
 **case** 3://删除  
 **if**(rowId === ''){  
 message.warn("请先选择一行")  
 }**else if**(rowobj.state === 0 || rowobj.state === 1){  
 api.changeVisaClaim.deleteDrop(PrjID,rowId).then(res=>{  
 **if**(res.op === "suc"){  
 message.success('删除成功');  
 **this**.getDate()  
 }  
 })  
 }**else**{  
 message.warn("当前清单数据已处于受审核中，无法删除")  
 }  
 **break**;  
 **default**:  
 **break**;  
 }  
};

关于提交按钮，新增和编辑用的是同一个页面，因此按钮也是同一个，一般接口也是同一个，这就需要调用提交方法时，传输能区分新增和编辑的参数，然后条件判断不同参数下请求不同接口

## 封装modal确认框

**import** React,{Component} **from** 'react'  
**import** style **from** './css.less'  
**import** {Modal,Button} **from** 'antd'  
  
**export default class** ConfirmBox **extends** Component{  
 **static** *defaultProps* = {  
 visible:**false** };  
 render(){  
 **return** (  
 <div className={style.confirmBox}>  
 <Modal  
 title="确认"  
 visible={**this**.props.visible}  
 onCancel={**this**.props.onCancel}  
 maskClosable={**false**}  
 footer={**null**}  
 centered  
 destroyOnClose  
 width={350}  
 style={{height:174}}  
 wrapClassName={'excel sameStyle'}  
 >  
 <div className={"content"}>{**this**.props.text}</div>  
 <div className={"btn"}>  
 <Button onClick={**this**.props.onOk}>确定</Button>  
 <Button onClick={**this**.props.onCancel}>取消</Button>  
 </div>  
 </Modal>  
 </div>  
 )  
 }  
}

调用：

在需要打开的确认框的地方，将状态confirmBox为true，在需要关闭的时候，设为false；

将确认框的提示的文字传输给text；

然后将点击确定要执行的方法传给onOk；

关闭确认框，modal为false，点击×和取消按钮的时候执行这个方法

{/\*确认框\*/}  
<ConfirmBox  
 visible={**this**.state.confirmBox}  
 text={'确认解锁吗？解锁后可重新对变更进行编辑'}  
 onOk={**this**.unlocking1}  
 onCancel={()=>{**this**.setState({confirmBox: **false**})}}  
/>

备注：一般要调用确认框时，都将一个方法分成两步，因此第一步处理后的数据一般也是要等到第二步点击确认框的确认后才执行。两个方法间又没办法进行参数传递，因此第一步里的处理的数据先保存在状态里，第二步再调用。

若一个提交方法有两种调用方式，一种直接一步调用，另一个要加确认框，分成两步。那么，先区分好一步和两步的计算后，一步的直接将处理的参数传递下去即可；两步的，先处理好的数据保存在状态里，打开确认框，然后在提交方法里取状态里的数据，因此在提交方法里还需要判断是直接调用还是确认框的调用。

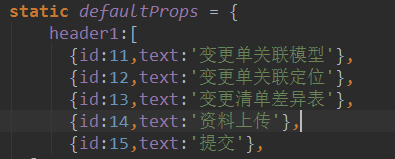
**if**(data.flgPass === 1){  
 **this**.setState({  
 confirmBox: **true**,  
 submitData:data  
 });  
}**else**{  
 **this**.submitPort1(data)  
}

## 方法的位置

假设这是一个tab标签页：

像这种，可渲染成li列表类型的文字，可以是操作按钮，tab标签之类的，都可以先将文字写在静态属性里，用map渲染出来；

若是操作按钮是变动，不同页面的操作按钮文字都不一样，那更应该用静态属性来渲染，选择不同的静态属性进行渲染。



//顶部按钮动态回显  
selectHeader = ()=>{  
 **switch** (**this**.props.applyType) {  
 **case** 0:  
 **return this**.props.header1;  
 **case** 1:  
 **case** 2:  
 **case** 3:  
 **case** 4:  
 **case** 5:  
 **return this**.props.header2;  
 **case** 6:  
 **return this**.props.header3;  
 **default**:  
 **return this**.props.header1;  
 }  
};

若一个js里调用多个子级，多个子级间有公共的方法，则此方法写在父级js里，将此方法传参给子级js即可。

若有请求数据回来进行回显，若父级js有用到数据，或是其他的子级js也有用到，则无论是哪个js进行回显，请求数据的方法都写在父级js里，然后再将请求回来的数据传输给子级js；

在子级js里，只是回显的字段直接写在回显的地方，而需要操作改变的字段，则要先赋值给状态，每一个字段都需要单独的一个状态，这样这些字段才能回显，更改，提交。提交后一般要重新请求数据，将父级js请求数据的方法传输给子级js，然后子级js就可以调用。

一般若请求的数据只是这个子级js里使用，父级js和其他兄弟js都没关系的话，这个请求数据就只需要写在子级js里，无操作，则不需要设置状态，有操作，则要设置状态。

## 父级js获取到子级js的方法

方法一：

给子组件的调用添加ref：

ref={ApplyTable=>**this**.ApplyTable = ApplyTable}

在子组件里处理好数据返回出来：

submitData = (bool)=>{  
 **let** text = **this**.visibleInput();  
 **let** obj = {  
 drawingCode:**this**.state.drawingcode,//图号  
 changeReason:**this**.state.changereason,//变更原因  
 changeContent:**this**.state.changecontent, //变更内容

...  
 };  
 **return** obj  
};

在父组件里调用子组件的方法就可以获取到数据：  
（还可以根据传进去不同的参数，返回出来不同的数据，列入下面的true）

**let** submitData = **this**.ApplyTable.submitData(**true**);

方法二：

在父组件写好一个可以接收参数的方法，将此方法传给子组件，在子组件里需求传参的位置调用这个方法，将数据作为这个方法的参数。此时在父组件的这个方法被触发，参数就是子组件传过来的数据。

## 覆盖浏览器自动填充的input颜色

input:-webkit-autofill {  
 -webkit-box-shadow: 0 0 0px 1000px rgba(255, 0, 0, 0.1) inset **!important**;  
}

## 下载

### 根据url下载

**const** ele = document.createElement('a');  
ele.setAttribute('href',url);  
ele.setAttribute('download','工作变更申请表.docx');  
document.body.appendChild(ele);  
ele.click();  
document.body.removeChild(ele);  
message.success('下载成功')

## 条件判断区分

### 区分多个输入框是否禁用

这个方法被调用两次，首先是在componentWillMount里调用，用于判断输入框是否禁用，其次是在提交时候调用，因为只提交一个输入框的值，因此要判断提交哪一个输入框的值。

由于这里的两次判断，都是根据同一个applyType来判断，且一个是设置组件状态，一个判断后返回出值，互不影响。因此可以将其写在一起。

//判断输入框是否禁用 and 判断提交哪个字段  
visibleInput = ()=>{  
 **let** text = '';  
 **let** {applyType} = **this**.props;  
 **this**.setState({  
 visibleAudit:applyType === 0 ? **false**:**true**,  
 visibleAudit1:applyType === 1 ? **false**:**true**,  
 visibleAudit2:applyType === 2 ? **false**:**true**,  
 visibleAudit3:applyType === 3 ? **false**:**true**,  
 visibleAudit4:applyType === 4 ? **false**:**true**,  
 visibleAudit5:applyType === 5 ? **false**:**true**,  
 });  
 **switch** (**this**.props.applyType) {  
 **case** 0:  
 text = **this**.state.applyopinion;  
 **break**;  
 **case** 1:  
 text = **this**.state.auditopinion1;  
 **break**;  
 **case** 2:  
 text = **this**.state.auditopinion2;  
 **break**;  
 **case** 3:  
 text = **this**.state.auditopinion3;  
 **break**;  
 **case** 4:  
 text = **this**.state.auditopinion4;  
 **break**;  
 **case** 5:  
 text = **this**.state.proprietorauditopinion;  
 **break**;  
 **default**:  
 **break**;  
 }  
 **return** text  
};

### **根据不同状态回显不同数据**

若有很多种同等类型的字段，但这个页面的小的弹窗里，同种类型的数据只显示一种。因此要为这个小的弹窗设置一个专有的状态字段，然后条件判断不同类型下，将要回显的那个类型的值赋值给专有的状态字段。

//打开支付modal，回显支付modal数据  
payChange = (row,num)=>{  
 **let** text1 = '';  
 **let** text2 = '';  
 **let** oldQuality = '';  
 **let** oldUnitprice = '';  
 **let** newQuality = '';  
 **let** newUnitprice = '';  
 **let** contractSum = row.contractresult\*row.contractprice;  
 **switch** (num) {  
 **case** 2:  
 text1 = '一审';  
 text2 = '二审';  
 oldQuality = row.oneresult;  
 oldUnitprice = row.oneprice;  
 newQuality = row.tworesult;  
 newUnitprice = row.twoprice;  
 **break**;  
 **case** 3:  
 text1 = '二审';  
 text2 = '三审';  
 oldQuality = row.tworesult;  
 oldUnitprice = row.twoprice;  
 newQuality = row.threeresult;  
 newUnitprice = row.threeprice;  
 **break**;  
 **default**:  
 **break**;  
 }  
 **this**.setState({  
 visiblePay:**true**,  
 oldAuditText:text1,  
 newAuditText:text2,  
 oldQuality:oldQuality,//old工程量(支付modal)  
 oldUnitprice:oldUnitprice,//old单价(支付modal)  
 newQuality,//工程量  
 newUnitprice,//单价  
 contractSum,//合同合价  
 })  
};

## Html写表单

Table：设置整个表格的宽高；

Tr:设置高度；

Td：

设置宽度，只需要给第一列设，下面的会自动对齐第一列；

设置合并，合并几列就colSpan等于几；

文字居中：align="center"

<table border="1" width="1000" height="400" cellSpacing="0">  
 <tbody>  
 <tr className={style.height1}>  
 <td className={style.width1}>申请单位(公章)</td>  
 <td colSpan="3" className={style.padding}>{companyUser.length > 0?companyUser[0].customer:''}</td>  
 </tr>  
 <tr className={style.height1}>  
 <td align="center">联系人</td>  
 <td className={classNames(style.width2,style.padding)}>{companyUser.length > 0?companyUser[0].name:''}</td>  
 <td className={style.width1}>电话</td>  
 <td className={style.padding}>{applyUserPhone}</td>  
 </tr>  
 <tr className={style.height1}>  
 <td align="center">工程名称</td>  
 <td colSpan="3" className={style.padding}>{projectname}</td>  
 </tr>  
 <tr className={style.height1}>  
 <td align="center">专业</td>  
 <td className={style.padding}>{specialty}</td>  
 <td align="center"><b className={style.asterisk}>\*</b>图号</td>  
 <td>  
 <Input  
 value={drawingcode}  
 onChange={(e)=>**this**.applyChange(e,1)}  
 disabled={visibleAudit}  
 />  
 </td>  
 </tr>  
 <tr className={style.height1}>  
 <td align="center" className={style.padding}>变更预算</td>  
 <td className={style.padding}>{auditprice === ''?'':webapp.floatNumFormatted(auditprice)}</td>  
 <td align="center">增减造价</td>  
 <td className={style.padding}>{pricechange === ''?'':webapp.floatNumFormatted(pricechange)}</td>  
 </tr>  
 </tbody>  
</table>

## 累加

**let** ggg = 0;

data.forEach(row=>{  
 ggg += row.contractresult\*row.contractprice;  
});

## 动态渲染节点

### 根据数组对象数据渲染输入框：

方法一：（用于渲染前有数据操作）

sideMenuItem = () => {

...  
 **return** sideMenu.map((i) => {  
 **return** (  
 <Menu.Item key={i.key} onClick={() => **this**.onMenuClick(i)}>  
 <Icon type={i.icon} />  
 <span>{i.name}</span>  
 </Menu.Item>  
 );  
 });  
};

方法二：（用于没有数据操作，可直接用map来渲染的节点，就直接在xml里渲染）

<div className={style.BaseInformation}>  
 {data.map((item) => {  
 return (  
 <div className={classNames(style.item, { [style.line]: item.line })} key={item.key}>  
 <span className={style.title}>{item.title}：</span>  
 <span className={style.value}>{item.value}</span>  
 </div>  
 );  
 })}  
</div>

### **根据类型变量动态渲染组件**

**渲染组件：先将类型定义为对象属性，将这种类型下要渲染的元素定义为属性内容，然后在return出去的时候，就直接obj[type]，因为类型type也是变量**

showInput = () => {  
 **return this**.props.data.map((item) => {  
 **const** {type,} = item;  
 **const** obj = {  
 input: (  
 <InputField  
 addonBefore={addonBefore}  
 disabled={disabled}  
 required={required}  
 handleChange={(text1, text2) => **this**.handleChange(text1, text2, 'input')}  
 />  
 ),  
 select: (  
 <SelectField  
 selectData={options}  
 required={required}  
 handleChange={(text1, text2) => **this**.handleChange(text1, text2, 'select')}  
 />  
 )  
 };  
 **return** obj[type];  
 });  
};

Render:

{**this**.showInput()}

### **有子组件的递归渲染**

这里是treeNodes 是一个树的总数组对象，里面还有不确定的子数组节点，checkedKeys 是一个key值数组，都包含在总数组里的key范围之内。这里用map渲染节点，用{ children, ...props }解构出总数组里的一个对象。剩下的对象

const generateTree = (treeNodes = [], checkedKeys = []) => {  
 return treeNodes.map(({ children, ...props }) => (  
 <TreeNode {...props} disabled={checkedKeys.includes(props.key)}>  
 {generateTree(children, checkedKeys)}  
 </TreeNode>  
 ));  
};

# 封装公共方法

**const** publicMethod = {  
 */\*\*  
 \* 时间戳转换成xxxx-xx-xx时间格式  
 \*/* timeCycle:(text)=>{  
 **let** value = 0;  
 ......

**return** value  
 },

};  
**export default** publicMethod;

调用：

**import** publicMethod **from** '../publicMethod'

publicMethod.timeCycle(text)

## 时间戳转换成xxxx-xx-xx时间格式

可传入：10位的秒或13位的毫秒的数字或字符串都可以

timeCycle:(text)=>{  
 **let** value = 0;  
 **if**(**typeof** text === 'string'){  
 **if**(text.length === 10){  
 value = moment(parseInt(text)\*1000).format('YYYY-MM-DD')  
 }**else if**(text.length === 13){  
 value = moment(parseInt(text)).format('YYYY-MM-DD')  
 }  
 }  
 **if**(**typeof** text === 'number'){  
 **let** text1 = text.toString();  
 **if**(text1.length === 10){  
 value = moment(text\*1000).format('YYYY-MM-DD')  
 }**else if**(text1.length === 13){  
 value = moment(text).format('YYYY-MM-DD')  
 }  
 }  
 **return** value  
},

## 将字符串毫秒转换成moment格式：可用于给DatePicker组件回显

可传入：10位的秒或13位的毫秒的数字或字符串都可以

toMoment:(text)=>{  
 **let** value = **null**;  
 **if**(**typeof** text === 'string'){  
 **if**(text.length === 10){  
 value = moment(parseInt(text)\*1000)  
 }**else if**(text.length === 13){  
 value = moment(parseInt(text))  
 }  
 }  
 **if**(**typeof** text === 'number'){  
 **let** text1 = text.toString();  
 **if**(text1.length === 10){  
 value = moment(text\*1000)  
 }**else if**(text1.length === 13){  
 value = moment(text)  
 }  
 }  
 **return** value  
},

## 将数字金额转成金额大写

*/\*\*  
 \*将数字金额转成中文大写  
 \*/*chineseCapital:**function**(text){  
 **if**(text === **null** || text === **undefined** || **this**.blank(text) === **null**){  
 **return** ''  
 }  
 //把value通通变成字符串  
 **let** values = text.toString();  
 values = values.replace(/\s\*/g,"");  
 **if**(values === '' || values === '0'){  
 **return** ''  
 }  
 // 区分是整数还是小数  
 //整数：  
 **if**(values.split('.').length === 1){  
 **let** integer = **this**.integer(values) + '整';  
 **return** integer  
 }//小数  
 **else if**(values.split('.').length === 2){  
 **let** integer = **this**.integer(values.split('.')[0]);  
 **let** decimals = **this**.decimals(values.split('.')[1]);  
 **return** integer + decimals  
 }  
},  
  
//整数转成大写  
integer:**function**(value){  
 **let** len=value.length;//统计出长度  
 **let** arr=[];  
 **let** chin\_list=['零','壹','贰','叁','肆','伍','陆','柒','捌','玖'];//所有的数值对应的汉字  
 **let** chin\_lisp=['仟亿','佰亿','拾亿','亿','仟万','佰万','拾万','万','仟','佰','拾'];//进制  
 **for**(**let** i=0;i<len;i++){  
 arr.push(parseInt(value[i]));//存进去数字  
 arr[i]=chin\_list[arr[i]];//将数字作为汉字数组的下标，查找所有数字的汉字存进数组  
 }  
 //给所有汉字后面添加进制单位  
 **for**(**let** i=len-1,j=1;i>0;i--){  
 arr.splice(i,0,chin\_lisp[chin\_lisp.length-j++]) //j=2  
 }  
 //查找中间的‘零‘，删除‘零’后面的进制  
 **for**(**let** i=0;i<arr.length-2;i++){  
 **if**(arr[i] === '零'){  
 arr.splice(i+1,1);  
 }  
 }  
 //删除中间相邻的‘零’  
 **let** arr1 = **this**.deleteZero(arr);  
 //删除最后一个零  
 **if**(arr[arr1.length-1] === '零'){  
 arr.splice(arr.length-1,1)  
 }  
 arr=arr.join('');  
 **if**(len>=1){  
 arr+='元'  
 }  
 **return** arr;  
},  
  
//递归删除相邻的‘零’  
deleteZero:**function**(arr){  
 **for**(**let** i=0;i<arr.length-1;i++){  
 **if**(arr[i] === '零'){  
 **if**(arr[i+1] === '零'){  
 arr.splice(i+1,1);  
 **this**.deleteZero(arr)  
 }  
 }  
 }  
 **return** arr  
},  
  
//两位小数转成大写  
decimals:**function**(text){  
 **let** chin\_list=['零','壹','贰','叁','肆','伍','陆','柒','捌','玖'];  
 **let** arr = [];  
 **for**(**let** i=0;i<text.length;i++){  
 arr.push(parseInt(text[i]));//存进去数字  
 arr[i]=chin\_list[arr[i]];//将数字作为汉字数组的下标，查找所有数字的汉字存进数组  
 }  
 **if**(text.length === 2){  
 **if**(arr[0] === '零' && arr[1] === '零'){  
 **return** ''  
 }**else if**(arr[0] !== '零' && arr[1] === '零'){  
 **let** value = arr[0] + '角';  
 **return** value  
 }**else if**(arr[0] === '零' && arr[1] !== '零'){  
 **let** value = '零' + arr[1] + '分';  
 **return** value  
 }**else if**(arr[0] !== '零' && arr[1] !== '零'){  
 **let** value = arr[0] + '角' + arr[1] + '分';  
 **return** value  
 }**else**{  
 **return** ''  
 }  
 }**else if**(text.length === 1){  
 **if**(arr[0] === '零'){  
 **return** ''  
 }**else**{  
 **return** arr[0] + '角'  
 }  
 }**else**{  
 **return** ''  
 }  
},

## 生成6位随机数

mathRande:**function** () {  
 **let** num="";  
 **for**(**let** i=0;i<6;i++){  
 num+=Math.floor(Math.random()\*10);  
 }  
 **return** num  
},

## 验证空格

*/\*\*  
 \*验证空格：将字符串中去掉空格后转成数组  
 \* 空字符串，空格字符串，arr均为null  
 \*/*

blank:(text)=>{  
 **if**(text !== **undefined**){  
 **if**(text !== **null**){  
 **let** reg = /([^\s])/g;  
 text = text.toString();  
 **let** arr = text.match(reg);  
 **return** arr  
 }**else**{  
 **return null** }  
 }**else**{  
 **return null** }  
},

## 验证数字（保留两位小数）

*/\*\*  
 \*验证数字：先验证不能为0开头，再验证只能保存两位小数  
 \* 空字符串，空格字符串，arr均为null  
 \*/*testReg:(text)=>{  
 **let** str1 = /^(0|[1-9][0-9]\*)+(.[0-9]{1,2})?$/;//不能为0开头/^([1-9][0-9]\*)+(.[0-9]{1,2})?$/  
 **let** str2 = /^(0|[1-9][0-9]|[0-9]+(.[0-9]{2})\*)$/;  
 text = text.toString();  
 text = text.replace(/\s\*/g,"");  
 **if**(text === '0' || text === '0.00' || Number(text) === 0){  
 message.warn('请输入大于零的数字');  
 **return false** }**else if**(text.match(str1) === **null**){  
 message.warn('请输入非零开头的数字，小数点后只能保留两位');  
 **return false** }**else if**(text.length > 1 && Number(text) > 1 && text.charAt(0) === '0'){  
 message.warn('请输入非零开头的数字');  
 **return false** }**else if**(text.charAt(0) === '0' && text.charAt(1) === '0'){  
 message.warn('请输入非零开头的数字');  
 **return false** }**else**{  
 **return true** }  
}

# 封装组件

## 基本信息栏

调用：

import BaseInformation from 'src/Components/Public/BaseInformation/index';  
  
*/\*\*  
 \* line：是否占一整行，true：占一行；false/不传：三个一行  
 \* \*/*export default class BaseInformationTest extends Component {  
 constructor() {  
 super();  
 this.state = {  
 baseData: [  
 {  
 title: '名称',  
 value: 'kkkk',  
 key: '001'  
 },  
 {  
 title: '项目组长',  
 value: '嘎嘎嘎个',  
 key: 'aaa',  
 btn: [  
 { key: 'add', text: '添加', onClick: this.callback, show: true },  
 { key: 'cut', text: '切换', onClick: this.callback, show: false },  
 { key: 'remove', text: '移除', onClick: this.callback, show: false }  
 ]  
 },  
 {  
 title: '项目组名称',  
 value: '嘎嘎嘎',  
 key: '005'  
 },  
 {  
 title: '项目组组长',  
 value: '咯咯咯',  
 key: '006',  
 line: true //是否占一整行  
 },  
 {  
 title: '描述',  
 value: '哈哈哈哈',  
 key: '007'  
 }  
 ]  
 };  
 }

**针对有按钮的那个btn数据，因为需求为：单独一个‘添加’，或‘切换’和‘移除’一起。而里面的show就是控制按钮是否显示的，因此‘切换’与‘移除’的show值应该一致与‘添加’相反。然后要将这两组按钮进行切换时，首先遍历大数组，找到有btn的那条数据，再遍历btn数组，将里面的所有的show值都变为相反的布尔值，然后在里层的遍历里要返回el，在外层的遍历再返回item。这样就把整个数组里的按钮的状态都改变了，按钮也就随之切换了** */\*\*  
 \* 切换按钮  
 \*/* cutBtn = () => {  
 const { baseData } = this.state;  
 let *data* = baseData.map((item) => {  
 if (item.btn) {  
 item.btn.map((el) => {  
 el.show = !el.show;  
 return el;  
 });  
 }  
 return item;  
 });  
 this.setState({  
 baseData: *data* });  
 };

**当按钮由数组对象渲染出来，且点击每个按钮都有回调函数时，可以直接在数组数据的每个按钮对象后面加属性onClick，属性值为点击要执行的回调函数。可以将其设置为同一个函数，通过返回不同的参数加以区分点击的是哪个按钮。也可以设置不同的回调。**

**当然前提是：在渲染按钮时，就要把回调函数渲染在按钮节点的onClick属性里，点击才能执行。**

**注意：这里因为按钮是渲染的，代码只有一次，因此在回调函数传参时，所有的回调函数传递的都是相同的参数**  
 */\*\*  
 \* 按钮回调  
 \*/* callback = (type) => {  
 switch (type) {  
 case 'add':  
 console.log('添加');  
 break;  
 case 'cut':  
 console.log('切换');  
 break;  
 case 'remove':  
 console.log('移除');  
 break;  
 default:  
 break;  
 }  
 };

**要改变里面的数据的某一条数据内容，首先遍历其数组，找到要改变某一个属性值的那个对象，将要改变属性值的属性直接赋值改变，最后return出整个item，再将整个数组保存在状态里触发页面重新渲染** */\*\*  
 \* 添加数据-切换按钮  
 \*/* addData = () => {  
 let *a* = 'aaassddfdf';  
 const { baseData } = this.state;  
 let *data* = baseData.map((item) => {  
 if (item.key === 'aaa') {  
 item.value = *a*;  
 }  
 return item;  
 });  
 this.setState({  
 baseData: *data* });  
 this.cutBtn();  
 };  
**与上面的原理相同：** */\*\*  
 \* 清空数据  
 \*/* celarData = () => {  
 const { baseData } = this.state;  
 let *data* = baseData.map((item) => {  
 if (item.key === 'aaa') {  
 item.value = '';  
 }  
 return item;  
 });  
 this.setState({  
 baseData: *data* });  
 this.cutBtn();  
 };  
  
 render() {  
 const { baseData } = this.state;  
 return (  
 <div>  
 <BaseInformation data={baseData} />  
 <Button onClick={this.cutBtn} style={{ margin: 5 }}>  
 切换按钮  
 </Button>  
 <Button onClick={this.addData} style={{ margin: 5 }}>  
 添加数据  
 </Button>  
 <Button onClick={this.celarData} style={{ margin: 5 }}>  
 清空数据  
 </Button>  
 </div>  
 );  
 }  
}

组件：

import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import style from './css.less';  
import classNames from 'classnames';  
  
export default class BaseInformation extends Component {

**给传递下来的数据加类型判断** static *propTypes* = {  
 data: PropTypes.array //输入框数组对象  
 };  
**传递下来的属性加默认数据** static *defaultProps* = {  
 data: []  
 };

**如果有btn属性，就渲染，此方法是直接用在render里的，所以可直接return出整个map遍历的数组。**

**注意：要将对象的onClick属性渲染在节点的onClick属性，并返回用以区分的参数** btnShow = (data) => {  
 return data.map((item) => {  
 return item.show ? (  
 <span key={item.key} className={style.btn} onClick={() => item.onClick(item.key)}>  
 {item.text}  
 </span>  
 ) : null;  
 });  
 };  
 render() {  
 const { data } = this.props;  
 return (  
 <div className={style.BaseInformation}>  
 {data.map((item) => {  
 return (  
 <div className={classNames(style.item, { [style.line]: item.line })} key={item.key}>  
 <span className={style.title}>{item.title}：</span>  
 <span className={style.value}>{item.value}</span>  
 {item.btn ? this.btnShow(item.btn) : null}  
 </div>  
 );  
 })}  
 </div>  
 );  
 }  
}

Css:

.BaseInformation {  
 display: flex;  
 flex-wrap: wrap;  
 font-size: 14px;  
 .item {  
 width: 32%;  
 height: 40px;  
 line-height: 40px;  
 .title {  
 color: #000;  
 }  
 .value {  
 color: #999;  
 }  
 .btn {  
 cursor: pointer;  
 color: #2a99e2;  
 margin: 0 10px;  
 }  
 }  
 .line {  
 width: 100%;  
 }  
}

## 搜索栏（查询和重置）

### 调用（传输数据）：

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import SearchBox from 'src/Components/Public/SearchBox/index';  
  
*/\*\*  
 \* 注意：总的有三种输入框,  
 \* (1)若只是一般的输入框input，则必填项为：{key,name,type};  
 \* (2)若为一般的选择器，必填项为:{key,name,type,choice,selectData},  
 \* (3)若为与远程数据结合的选择器，必填项为：{key,name,type,choice,Interface,callBack}  
 \* 调用时，在callBase可以接收到接口请求回来的数据后，需提取数据为[{text,value},...]格式，并return出去；  
 \* width和line二选一，占一整行就不需要再传入宽度  
 \*/*export default class SearchBoxTest extends Component {  
 constructor() {  
 super();  
 this.state = {  
 num: 0,  
 dataBox: {  
 openClose: true, //是否显示展开和收起按钮（必填）  
 className: '', //控制容器的类型  
 inquireBtn: (data) => {  
 console.log(data);  
 }, //点击查询后返回输入的数据对象（必填）  
 data: [  
 //（必填）  
 {  
 key: '001', //（必填）  
 label: '名称/编号', //输入框的标题，若没有，则输入''  
 name: 'aaa', //输入框的名称，用于查询获取到每个输入框内容的标识（必填）  
 type: 'input', //input：普通输入框；select：普通选择器；seekSelect：与远程数据相结合的选择器，根据输入内容实时请求远程数据（必填）  
 placeholder: '请输入', //占位提示内容  
 defaultValue: '555', //初始值  
 width: 230 //输入框的宽度（number）  
 },  
 {  
 key: '05454303', //（必填）  
 label: '呵呵',  
 name: 'hfhf', //（必填）  
 type: 'select', //（必填）  
 placeholder: '请输入',  
 defaultValue: '01', //若为单选，直接填入selectData数据里的key值（string），若为多选，填入数组来装多个key值  
 choice: 'radio', //radio:单选，multiple:多选（必填）  
 width: 230, //输入框的宽度（number）  
 line: true, //是否占一整行,  
 selectData: [{ key: '01', value: '删除' }, { key: '02', value: '创建' }, { key: '03', value: '修改' }] //（必填）  
 },  
 {  
 key: '003', //（必填）  
 label: '安安',  
 name: 'ddd', //（必填）  
 type: 'select', //（必填）  
 placeholder: '请输入',  
 defaultValue: ['01', '02'], //若为单选，直接填入selectData数据里的key值（string），若为多选，填入数组来装多个key值  
 choice: 'multiple', //radio:单选，multiple:多选（必填）  
 width: 230, //输入框的宽度（number）  
 selectData: [  
 { key: '01', value: '权限-删除' },  
 { key: '02', value: '权限-创建' },  
 { key: '03', value: '权限-修改' }  
 ] //（必填）  
 },  
 {  
 key: 'roleIds', //（必填）  
 label: '角色',  
 name: 'roleIds', //（必填）  
 type: 'seekSelect', //（必填）  
 Interface: {  
 //（必填）  
 action: '/api/upms/manage/roles/name?context=',  
 method: 'get'  
 }, //请求远程数据接口  
 placeholder: '请输入关键字',  
 defaultValue: [],  
 width: 230, //输入框的宽度（number）  
 choice: 'multiple' //（必填）  
 },  
 {  
 key: 'permissionIds', //（必填）  
 label: '权限',  
 name: 'permissionIds', //（必填）  
 type: 'seekSelect', //（必填）  
 Interface: {  
 //（必填）  
 action: '/api/upms/manage/permissions/name?context=',  
 method: 'get'  
 }, //请求远程数据接口  
 placeholder: '请输入关键字',  
 defaultValue: [],  
 width: 230, //输入框的宽度（number）  
 choice: 'multiple' //（必填）  
 },  
 {  
 key: '454355',  
 label: '所属',  
 name: 'belong',  
 type: 'preInput',  
 placeholder: '请输入',  
 prePlaceholder: '归属类型',  
 defaultValue: '',  
 preDefaultValue: '02',  
 preData: [  
 { key: '01', value: '无', disabled: false },  
 { key: '02', value: '平台', disabled: true },  
 { key: '03', value: '公司', disabled: false }  
 ],  
 width: 230  
 },  
 {  
 key: '65565',  
 label: '项目组长',  
 name: 'leader',  
 type: 'clickInput',  
 placeholder: '请选择',  
 defaultValue: [],  
 // choice: 'radio', //radio:单选，multiple:多选（必填）  
 errText: '添加失败，值已存在', //若为单选，添加第二个值时的提示语  
 width: 230,  
 callback: this.event  
 }  
 ]  
 }  
 };  
 }  
**该方法用在有点击回调的那个输入框，点击后缀图标的时候调用这个方法，返回出这条对象数据的key和name** //点击后缀图标回调  
 event = (key, name) => {  
 console.log('点击弹出模态框' + key, name);  
 };  
**该方法用在给有点击的那个输入框传值** value = () => {  
 let *num* = this.state.num;  
 *num*++;  
 this.setState({ *num* });  
 //给输入框添加值，给SearchBox加ref，然后调用这句代码  
 this.searchBox.setValue({ key: '65565', value: *num*, id: 'FHISDFHAFBVSHVISHISFHSH' });  
 };  
 render() {  
 return (  
 <div>  
 <SearchBox dataBox={this.state.dataBox} ref={(ref) => (this.searchBox = ref)} />  
 <div onClick={this.value}>给项目组长添加value</div>  
 </div>  
 );  
 }  
}

### 父组件index：

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import classNames from 'classnames';  
import { Button } from 'antd';  
import InputBox from 'src/Components/Public/SearchBox/InputBox/index'; //普通输入框  
import SelectBox from 'src/Components/Public/SearchBox/SelectBox/index'; //普通选择器  
import SeekSelectBox from 'src/Components/Public/SearchBox/SeekSelectBox/index'; //与远程数据相结合的选择器  
import PreInput from 'src/Components/Public/SearchBox/PreInput/index'; //前缀输入框  
import ClickInput from 'src/Components/Public/SearchBox/ClickInput/index'; //点击输入框  
import style from './css.less';  
*/\*\*  
 \* 注意：总的有四种输入框,  
 \* (1)若只是一般的输入框input，则必填项为：{key,name,type};  
 \* (2)若为一般的选择器，必填项为:{key,name,type,choice,selectData},  
 \* (3)若为与远程数据结合的选择器，必填项为：{key,name,type,choice,Interface,callBack}  
 \* 调用时，在callBase可以接收到接口请求回来的数据后，需提取数据为[{text,value},...]格式，并return出去  
 \*/*export default class SearchBox extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 open: false, //控制展开与收起  
 obj: {}, //输入时保存输入的数据  
 data: props.dataBox.data  
 };  
 }  
**验证对象里面的每一个属性的数据类型** static *propTypes* = {  
 //输入框数组对象  
 dataBox: PropTypes.shape({  
 openClose: PropTypes.bool.isRequired,  
 className: PropTypes.string,  
 inquireBtn: PropTypes.func.isRequired,  
 data: PropTypes.array.isRequired  
 })  
 };  
 static *defaultProps* = {  
 dataBox: {  
 openClose: false,  
 className: '',  
 inquireBtn: () => {},  
 data: []  
 }  
 };  
 componentWillMount() {  
 this.getDefault();  
 }

**设置默认对象：在这里先设置好点击查询返回的数据对象，对象的属性为name,属性值为defaultValue，保存在对象里，也可以保存在构造函数里，这样改变输入值时就不会渲染页面。**

**注意：这里类型为‘**preInput**’为一个对象，因此需要单独设置里面每个属性的初始值** */\*\*  
 \* 获取初始数据  
 \*/* getDefault = () => {  
 let { data } = this.state;  
 let *obj* = {};  
 data.forEach((item) => {  
 *obj*[item.name] = item.defaultValue;  
 if (item.type === 'preInput') {  
 *obj*[item.name] = {};  
 *obj*[item.name].preSelect = item.preDefaultValue;  
 *obj*[item.name].preValue = item.defaultValue;  
 }  
 });  
 this.setState({  
 *obj* });  
 };

**当每次输入框值发生变化的时候都会执行该函数，然后直接在初始对象里根据返回的name，找到该属性，赋值text；**

**这样的话，就算不输入，直接点击查询，返回的就为初始值，输入过就为输入的值。**

**这里不进行渲染页面，因为这里进行了浅复制，改变obj的同时，也改变了this.state.obj，此时这个方法里的obj与this.state.obj是相等的。而这里又将obj return出去了，因此在点击查询获取值的，直接取这个方法的obj或者去this.state.obj都是一样的。** */\*\*  
 \*获取输入的数据(浅复制)  
 \** ***@param*** *{string}text-输入框内容 name-输入框名称 type-输入框类型  
 \** ***@return*** *{object}  
 \*/* handleChange = (text, name, type) => {  
 let *obj* = this.state.obj;  
 if (type === 'common') {  
 *obj*[name] = text;  
 } else {  
 switch (type) {  
 case 'preSelect':  
 *obj*[name].preSelect = text;  
 break;  
 case 'preValue':  
 *obj*[name].preValue = text;  
 break;  
 default:  
 break;  
 }  
 }  
 return *obj*;  
 };

**重置的时候，将内容对象和底下的输入框清空：**

**要触发每一个输入框的重置方法，因此在渲染的时候，需要渲染不同的ref给每个输入框，这里面才能用ref调用到每一个输入框。**

**多选和点击输入框的数据类型为：array**

**前缀输入框的属性值为对象，对象的属性值为字符串，因此需要单独设置；**

**其他的数据类型都是：字符串** */\*\*  
 \* 点击重置  
 \*/* resetData = () => {  
 let *data* = this.props.dataBox.data;  
 const { obj } = this.state;  
 *data*.forEach((item) => {  
 this[item.key].reset();  
 if (item.choice === 'multiple' || item.type === 'clickInput') {  
 obj[item.name] = [];  
 } else if (item.type === 'preInput') {  
 obj[item.name].preSelect = '';  
 obj[item.name].preValue = '';  
 } else {  
 obj[item.name] = '';  
 }  
 });  
 this.setState({ obj });  
 };  
**点击查询，返回出输入时保存的内容对象，可直接取this.state.obj，也可取上面this.handleChange的返回值** */\*\*  
 \*点击查询  
 \*/* inquire = () => {  
 let *obj* = this.handleChange();  
 this.props.dataBox.inquireBtn(*obj*);  
 };

**该方法针对于clickInput是输入框，是一个选择器，但是没有下拉选择项，值由外面的方法传入。**

**因此在外面要这个输入框里传值的时候，就在外面用ref调用下面这个方法。**

**这个方法的作用：获取到外面传入的key值，根据key用ref调用到对应的输入框，然后将剩下的对象再传输给找到的输入框的方法里** */\*\*  
 \* 设置点击选择框的值  
 \*/* setValue = ({ key, ...obj } = data) => {  
 this[key].setValue({ ...obj });  
 };  
**渲染方法：可以用switch，也可以用对象法，将type为属性，对应的要渲染的节点为属性值，最后返回出obj[type]。都是一样的**

**handleChange方法里的第三个参数，用于区分该输入框的内容为一个值，还是多个值（对象），common为一个值，直接将值赋值给内容内容的属性即可，但PreInput节点，里面是有多个值的，在内容对象里对应的属性值也是一个对象，有两个属性，因此要进行区分是哪一个的值，类型就由子节点传回来**

**重点：根据不同key值给不同子节点加ref：ref={(ref) => {this[item.key] = ref;}}** */\*\*  
 \* 根据类型渲染不同组件  
 \* input:一般输入框；select：一般选择框；seekSelect：与远程数据相结合选择框  
 \*/* drawInput = () => {  
 let *data* = this.props.dataBox.data;  
 return *data*.map((item) => {  
 switch (item.type) {  
 case 'input':  
 return (  
 <InputBox  
 ref={(ref) => {  
 this[item.key] = ref;  
 }}  
 key={item.key}  
 label={item.label}  
 name={item.name}  
 placeholder={item.placeholder}  
 defaultValue={item.defaultValue}  
 width={item.width}  
 line={item.line}  
 handleChange={(text1, text2) => this.handleChange(text1, text2, 'common')}  
 />  
 );  
 case 'select':  
 return (  
 <SelectBox  
 ref={(ref) => {  
 this[item.key] = ref;  
 }}  
 key={item.key}  
 label={item.label}  
 name={item.name}  
 placeholder={item.placeholder}  
 defaultValue={item.defaultValue}  
 choice={item.choice}  
 width={item.width}  
 line={item.line}  
 selectData={item.selectData}  
 handleChange={(text1, text2) => this.handleChange(text1, text2, 'common')}  
 />  
 );  
 case 'seekSelect':  
 return (  
 <SeekSelectBox  
 ref={(ref) => {  
 this[item.key] = ref;  
 }}  
 key={item.key}  
 label={item.label}  
 name={item.name}  
 Interface={item.Interface}  
 placeholder={item.placeholder}  
 defaultValue={item.defaultValue}  
 choice={item.choice}  
 callBack={item.callBack}  
 width={item.width}  
 line={item.line}  
 handleChange={(text1, text2) => this.handleChange(text1, text2, 'common')}  
 />  
 );  
 case 'preInput':  
 return (  
 <PreInput  
 ref={(ref) => {  
 this[item.key] = ref;  
 }}  
 key={item.key}  
 label={item.label}  
 name={item.name}  
 placeholder={item.placeholder}  
 prePlaceholder={item.prePlaceholder}  
 defaultValue={item.defaultValue}  
 preDefaultValue={item.preDefaultValue}  
 width={item.width}  
 line={item.line}  
 preData={item.preData}  
 handleChange={(text1, text2, type1) => this.handleChange(text1, text2, type1)}  
 />  
 );  
 case 'clickInput':  
 return (  
 <ClickInput  
 ref={(ref) => {  
 this[item.key] = ref;  
 }}  
 key={item.key}  
 label={item.label}  
 name={item.name}  
 placeholder={item.placeholder}  
 defaultValue={item.defaultValue}  
 width={item.width}  
 line={item.line}  
 choice={item.choice}  
 errText={item.errText}  
 callback={(name) => item.callback(name, item.key)}  
 handleChange={(text1, text2) => this.handleChange(text1, text2, 'common')}  
 />  
 );  
 default:  
 return null;  
 }  
 });  
 };  
**外层可以再加入一个className，给这个组件的最上层，可以再外面单独改样式；**

**通过openClose 判断是否有展开和收起按钮；按钮只有一个，通过open判断显示‘展开’还是‘收起’，点击的时候将open取反，按钮文字也随之改变** render() {  
 const { *className*, dataBox } = this.props;  
 const { open } = this.state;  
 return (  
 <div className={classNames(style.searchBox, *className*, { [style.searchBox1]: open })}>  
 <div className={style.input}>{this.drawInput()}</div>  
 <div className={style.btns}>  
 {dataBox.openClose ? (  
 <div className={style.open}>  
 <Button onClick={() => this.setState({ open: !open })}>{open ? '展开' : '收起'}</Button>  
 </div>  
 ) : (  
 ''  
 )}  
 <div className={style.find}>  
 <Button type="primary" onClick={this.inquire}>  
 查询  
 </Button>  
 <Button onClick={this.resetData}>重置</Button>  
 </div>  
 </div>  
 </div>  
 );  
 }  
}

### **父组件css:**

/\*\* @format \*/  
  
.searchBox {  
 //background-color: #ccc;  
 display: flex;  
 :global {  
 .ant-input {  
 width: 80%;  
 }  
 }  
 .input {  
 width: calc(100% - 148px);  
 display: flex;  
 flex-wrap: wrap;  
 .item {  
 margin-right: 10px;  
 width: 32%;  
 height: 45px;  
 line-height: 45px;  
 display: flex;  
 align-items: center;  
 }  
 }  
 .btns {  
 width: 148px;  
 text-align: right;  
 display: flex;  
 flex-wrap: wrap;  
 .open {  
 text-align: right;  
 height: 45px;  
 line-height: 45px;  
 width: 100%;  
 }

**这里，align-self: flex-end;使得find这个节点始终在父盒子的底端对齐** .find {  
 height: 45px;  
 line-height: 45px;  
 align-self: flex-end;  
 }  
 :global(.ant-btn) {  
 margin-right: 10px;  
 }  
 }  
}  
.searchBox1 {  
 height: 45px;  
 overflow: hidden;  
}

### 子组件input：

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import { Input } from 'antd';  
import style from './css.less';  
import classNames from 'classnames';

**input里有value属性的话，defaultValue就不管用了，因此要将父组件传递下来的defaultValue赋值给value进行回显初始值**export default class InputBox extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 value: props.defaultValue  
 };  
 }  
 static *propTypes* = {  
 label: PropTypes.string, //输入框的标题，若没有，则输入''  
 name: PropTypes.string.isRequired, //输入框的名称，用于查询获取到每个输入框内容的标识  
 placeholder: PropTypes.string, //占位提示内容  
 defaultValue: PropTypes.oneOfType([PropTypes.string, PropTypes.array]), //初始值  
 width: PropTypes.number, //输入框的宽度（number）  
 handleChange: PropTypes.func.isRequired //获取输入内容的回调  
 };  
 static *defaultProps* = {  
 label: '',  
 name: '',  
 placeholder: '请输入内容',  
 defaultValue: '',  
 width: 230,  
 handleChange: () => {}  
 };  
**这个一般输入框是最简单的，只有两个功能：**

**输入的时候获取内容，调用父组件传递下来的handleChange,将值和该组件的名称传递回去；**

**点击到重置的时候，父组件会通过ref调用到这个reset方法，该方法将value清空。** */\*\*  
 \* 输入时获取内容  
 \*/* onChange = (e, name) => {  
 this.setState({ value: e.target.value });  
 this.props.handleChange(e.target.value, name);  
 };  
  
 */\*\*  
 \* 重置  
 \*/* reset = () => {  
 this.setState({  
 value: ''  
 });  
 };  
**title：表示属性移上去的时候回显出所有的值** render() {  
 const { *placeholder*, *label*, *width*, *line*, *name* } = this.props;  
 return (  
 <div className={classNames(style.item, { [style.item1]: *line* })}>  
 <span className={style.label}>{*label*.length > 0 ? `${*label*}：` : ''}</span>  
 <Input  
 placeholder={*placeholder*}  
 value={this.state.value}  
 title={this.state.value}  
 style={{ width: *width* }}  
 onChange={(e) => this.onChange(e, *name*)}  
 />  
 </div>  
 );  
 }  
}

### 子组件select：

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import classNames from 'classnames';  
import httpFetch from 'src/util/httpFetch';  
import { Select } from 'antd';  
import style from './css.less';  
const { *Option* } = Select;  
  
export default class InputBox extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 value: props.defaultValue,  
 options: props.selectData  
 };  
 }  
 static *propTypes* = {  
 label: PropTypes.string, //输入框的标题，若没有，则输入''  
 name: PropTypes.string.isRequired, //输入框的名称，用于查询获取到每个输入框内容的标识  
 placeholder: PropTypes.string, //占位提示内容  
 defaultValue: PropTypes.oneOfType([PropTypes.string, PropTypes.array]), //初始值  
 choice: PropTypes.string.isRequired, //radio:单选，multiple:多选（必填）  
 width: PropTypes.number, //输入框的宽度（number）  
 selectData: PropTypes.array.isRequired,  
 handleChange: PropTypes.func.isRequired,  
 Interface: PropTypes.object, //接口的配置与callBack一起使用  
 callBack: PropTypes.func //有接口时的回调函数与Interface一起使用  
 };  
 static *defaultProps* = {  
 placeholder: '请输入内容',  
 label: '',  
 name: '',  
 defaultValue: [],  
 selectData: [],  
 width: 230,  
 handleChange: () => {}  
 };

**这里若为单选，e为key字符串，若为多选，e为key数组**

*/\*\*  
 \* 输入时获取内容  
 \*/* onChange = (e, name) => {  
 this.setState({ value: e });  
 this.props.handleChange(e, name);  
 };

**重置里区分出是单选还是多选，单选-空字符串，多选-空数组** */\*\*  
 \* 重置  
 \*/* reset = () => {  
 let { *choice* } = this.props;  
 let *text*;  
 if (*choice* === 'multiple') {  
 *text* = [];  
 } else if (*choice* === 'radio') {  
 *text* = '';  
 }  
 this.setState({  
 value: *text* });  
 };  
  
 render() {  
 let { *placeholder*, *label*, *choice*, *width*, *line*, *name* } = this.props;  
 const { options } = this.state;  
 return (  
 <div className={classNames(style.item, { [style.item1]: *line* })}>  
 <span className={style.label}>{*label*.length > 0 ? `${*label*}：` : ''}</span>  
 <Select  
 placeholder={*placeholder*}  
 style={{ minWidth: *width* }}  
 mode={*choice*}  
 value={this.state.value}  
 title={this.state.value}  
 onChange={(e) => this.onChange(e, *name*)}  
 >  
 {options.map((el) => {  
 return (  
 <Option value={el.key} key={el.key}>  
 {el.value}  
 </Option>  
 );  
 })}  
 </Select>  
 </div>  
 );  
 }  
}

### 子组件seekselect：

*/\*\** ***@format*** *\*/*

***要进行防抖处理，首先要引入*lodash/debounce**import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import { Select, Spin } from 'antd';  
import debounce from 'lodash/debounce';  
import style from './css.less';  
import classNames from 'classnames';  
const { *Option* } = Select;  
  
*/\*\*  
 \* 选项与远程数据相结合  
 \* \*/*export default class InputBox extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 value: props.defaultValue,  
 data: [],  
 fetching: false //控制加载圈圈  
 };  
 this.lastFetchId = 0;

**这个构造函数表示，将this.fetchUser延迟800毫秒执行，如果在800之内，这个函数执行了第二次，那以第二次执行重新开始计时，若800毫秒之内没有再执行了，800毫秒之后才就开始执行，这样就减少了执行次数** this.fetchUser = debounce(this.fetchUser, 800);  
 }  
  
 static *propTypes* = {  
 label: PropTypes.string, //输入框的标题，若没有，则输入''  
 name: PropTypes.string, //输入框的名称，用于查询获取到每个输入框内容的标识（必填）  
 Interface: PropTypes.shape({  
 action: PropTypes.string, //请求远程数据接口  
 method: PropTypes.string, //请求方式  
 query: PropTypes.string, //承载输入框value值的字段名称  
 rowKey: PropTypes.string, //回显的key值字段  
 rowTitle: PropTypes.string //回显的名称字段  
 }),  
 placeholder: PropTypes.string,  
 defaultValue: PropTypes.oneOfType([PropTypes.string, PropTypes.array]), //若为单选，直接填入selectData数据里的key值（string），若为多选，填入数组来装多个key值  
 choice: PropTypes.string, //radio单选，multiple多选  
 callBack: PropTypes.func, //获取到调用组件后处理后的数据  
 width: PropTypes.number, //输入框的宽度（number）  
 handleChange: PropTypes.func  
 };  
 static *defaultProps* = {  
 label: '',  
 name: '',  
 Interface: {  
 action: 'https://randomuser.me/api/?results=5',  
 method: 'get',  
 query: 'results', //承载输入框value值的字段名称  
 rowKey: 'id', //回显的key值字段  
 rowTitle: 'name' //回显的名称字段  
 },  
 placeholder: '请输入内容',  
 defaultValue: '',  
 choice: '',  
 callBack: () => {},  
 width: 230,  
 handleChange: () => {}  
 };

**根据输入内容请求数据回来保存在状态里，请求前打开加载中，请求结果后保存选项时把加载中关闭；**

**这个方法是在往输入框里手动输入内容时执行** */\*\*  
 \* 根据输入的信息请求接口  
 \** ***@param*** *{string} value 输入框内容  
 \** ***@return*** *{\*}  
 \*/*fetchUser = (value) => {  
 let { *Interface* } = this.props;  
 const { *action*, *method*, *query* } = *Interface*;  
 this.setState({ data: [], fetching: true });  
 this.lastFetchId += 1;  
 const *fetchId* = this.lastFetchId;  
 let *url* = *action*;  
 if (*method* === 'get') {  
 *url* = *action* + value;  
 }  
 fetch(*url*, { method: *method* })  
 .then((response) => response.json())  
 .then((body) => {  
 if (body.op === 'suc') {  
 if (*fetchId* !== this.lastFetchId) {  
 return;  
 }  
 this.setState({  
 data: body.data,  
 fetching: false  
 });  
 }  
 });  
};

*/\*\*  
 \* 输入时获取内容  
 \*/* onChange = (e, name) => {  
 this.setState({  
 value: e,  
 fetching: false  
 });  
 this.props.handleChange(e, name);  
 };  
 */\*\*  
 \* 获得焦点时的回调  
 \*/* // onFocus = () => {  
 // this.setState({  
 // data: []  
 // });  
 // };  
 */\*\*  
 \* 重置  
 \*/* reset = () => {  
 let { *choice* } = this.props;  
 let *text*;  
 if (*choice* === 'multiple') {  
 *text* = [];  
 } else if (*choice* === 'radio') {  
 *text* = '';  
 }  
 this.setState({  
 value: *text* });  
 };  
  
 render() {  
 const { fetching, data } = this.state;  
 const { *label*, *placeholder*, *width*, *line*, *choice*, *name*, *Interface* } = this.props;  
  
 return (  
 <div className={classNames(style.item, { [style.item1]: *line* })}>  
 <span className={style.label}>{*label*.length > 0 ? `${*label*}：` : ''}</span>  
 <Select  
 showSearch={*choice* === 'radio' ? true : false}  
 mode={*choice*}  
 placeholder={*placeholder*}  
 notFoundContent={fetching ? <Spin size="small" /> : null}  
 filterOption={false}  
 value={this.state.value}  
 onSearch={this.fetchUser}  
 onChange={(e) => this.onChange(e, *name*)}  
 // onFocus={this.onFocus}  
 style={{ minWidth: *width* }}  
 >  
 {data.map((item) => (  
 <Option key={item[*Interface*.rowKey]}>{item[*Interface*.rowTitle]}</Option>  
 ))}  
 </Select>  
 </div>  
 );  
 }  
}

### 子组件PreInput ：

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import { Input, Select } from 'antd';  
import style from './css.less';  
import classNames from 'classnames';  
const { *Option* } = Select;  
  
export default class PreInput extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 value: props.defaultValue,  
 selectValue: props.preDefaultValue,  
 disabled: false  
 };  
 }  
 static *propTypes* = {  
 label: PropTypes.string, //输入框的标题，若没有，则输入''  
 name: PropTypes.string.isRequired, //输入框的名称，用于查询获取到每个输入框内容的标识  
 placeholder: PropTypes.string, //占位提示内容  
 defaultValue: PropTypes.string, //输入框初始值  
 preDefaultValue: PropTypes.string, //前缀初始值  
 prePlaceholder: PropTypes.string, //前缀占位符  
 preData: PropTypes.array.isRequired, //前缀下来列表  
 width: PropTypes.number, //输入框的宽度（number）  
 handleChange: PropTypes.func.isRequired //获取输入内容的回调  
 };  
 static *defaultProps* = {  
 label: '',  
 name: '',  
 placeholder: '请输入内容',  
 prePlaceholder: '归属类型',  
 defaultValue: '',  
 preDefaultValue: '',  
 preData: [],  
 width: 230,  
 handleChange: () => {}  
 };  
 componentWillMount() {  
 //初始化disable  
 const { *preDefaultValue*, *preData* } = this.props;  
 *preData*.forEach((item) => {  
 if (item.key === *preDefaultValue*) {  
 this.setState({  
 disabled: item.disabled  
 });  
 }  
 });  
 }  
 */\*\*  
 \* 输入时获取内容  
 \*/* onChange = (e, name) => {  
 this.setState({ value: e.target.value });  
 this.props.handleChange(e.target.value, name, 'preValue');  
 };  
 */\*\*  
 \* 前缀变化时获取内容  
 \*/* selectChange = (value, text, name) => {  
 let *boolText* = text.key.split(',')[1];  
 let *bool*;  
 if (*boolText* === 'true') {  
 *bool* = true;  
 } else if (*boolText* === 'false') {  
 *bool* = false;  
 }  
 this.setState({  
 disabled: *bool*,  
 value: '',  
 selectValue: value  
 });  
 this.props.handleChange(value, name, 'preSelect');  
 this.props.handleChange('', name, 'preValue');  
 };  
 */\*\*  
 \* 重置  
 \*/* reset = () => {  
 this.setState({  
 value: '',  
 selectValue: ''  
 });  
 };  
 render() {  
 const { *placeholder*, *prePlaceholder*, *label*, *width*, *line*, *name*, *preData*, *preDefaultValue* } = this.props;  
 const { *disabled*, *selectValue*, *value* } = this.state;  
 const *selectBefore* = (  
 <Select  
 style={{ width: 100 }}  
 placeholder={*prePlaceholder*}  
 value={*selectValue*}  
 defaultValue={*preDefaultValue*}  
 onChange={(value, text) => this.selectChange(value, text, *name*)}  
 >  
 {*preData*.map((el) => {  
 return (  
 <Option value={el.key} key={el.key + ',' + el.disabled}>  
 {el.value}  
 </Option>  
 );  
 })}  
 </Select>  
 );  
 return (  
 <div className={classNames(style.item, { [style.item1]: *line* })}>  
 <span className={style.label}>{*label*.length > 0 ? `${*label*}：` : ''}</span>  
 <Input  
 addonBefore={*selectBefore*}  
 placeholder={*placeholder*}  
 value={*value*}  
 title={*value*}  
 style={{ width: *width* }}  
 onChange={(e) => this.onChange(e, *name*)}  
 disabled={*disabled*}  
 />  
 </div>  
 );  
 }  
}

### 子组件ClickInput ：

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import { *Icon*, Select, *message* } from 'antd';  
import style from './css.less';  
import classNames from 'classnames';  
  
export default class ClickInput extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 value: JSON.stringify(props.defaultValue),  
 defaultValue: props.defaultValue  
 };  
 }  
 static *propTypes* = {  
 label: PropTypes.string, //输入框的标题，若没有，则输入''  
 name: PropTypes.string.isRequired, //输入框的名称，用于查询获取到每个输入框内容的标识  
 placeholder: PropTypes.string, //占位提示内容  
 defaultValue: PropTypes.array, //初始值  
 width: PropTypes.number, //输入框的宽度（number）  
 callback: PropTypes.func.isRequired, //点击图标的回调  
 handleChange: PropTypes.func.isRequired //获取输入内容的回调  
 };  
 static *defaultProps* = {  
 label: '',  
 name: '',  
 placeholder: '请输入内容',  
 defaultValue: '',  
 width: 230,  
 callback: () => {},  
 handleChange: () => {}  
 };  
  
 */\*\*  
 \* 获取值  
 \*/* setValue = (obj) => {  
 const { *value* } = this.state;  
 const { *choice*, *errText* } = this.props;  
 let *valueArr* = JSON.parse(*value*);  
 if (*choice* === 'radio') {  
 if (*valueArr*.length > 0) {  
 *message*.error(*errText*);  
 return;  
 }  
 }  
 const { *name* } = this.props;  
 *valueArr*.push(obj);  
 this.setState({ value: JSON.stringify(*valueArr*) });  
 this.props.handleChange(*valueArr*, *name*);  
 };  
 */\*\*  
 \* 重置  
 \*/* reset = () => {  
 this.setState({ value: JSON.stringify([]) });  
 };  
 */\*\*  
 \* 取消选种时回调  
 \*/* onDeselect = (text) => {  
 const { *value* } = this.state;  
 const { *name* } = this.props;  
 let *valueArr* = JSON.parse(*value*);  
 *valueArr*.forEach((item, ind) => {  
 if (item.value === text) {  
 *valueArr*.splice(ind, 1);  
 }  
 });  
 this.props.handleChange(*valueArr*, *name*);  
 this.setState({  
 value: JSON.stringify(*valueArr*)  
 });  
 };  
 */\*\*  
 \* 回显value  
 \*/* showValue = (valueStr) => {  
 return JSON.parse(valueStr).map((item) => {  
 return item.value;  
 });  
 };  
 render() {  
 const { *placeholder*, *label*, *width*, *line*, *name*, *callback* } = this.props;  
 const { *value* } = this.state;  
 return (  
 <div className={classNames(style.item, { [style.item1]: *line* })}>  
 <span className={style.label}>{*label*.length > 0 ? `${*label*}：` : ''}</span>  
 <Select  
 placeholder={*placeholder*}  
 mode={'multiple'}  
 value={this.showValue(*value*)}  
 title={this.showValue(*value*)}  
 style={{ width: *width* }}  
 onDeselect={this.onDeselect}  
 />  
 <Icon type="menu-fold" style={{ color: '#2a99e2' }} onClick={() => *callback*(*name*)} />  
 </div>  
 );  
 }  
}

## 创建栏（输入时同时输出）

### 调用：

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import FormItem from 'src/Components/Public/FormItem/index';  
  
*/\*\*  
 \*  
 \* 跟小谭组件传入对象的data属性的数据是一样的  
 \*/*export default class FormItemTest extends Component {  
 constructor() {  
 super();  
 this.state = {  
 num: 0  
 };  
 }  
 change = (data) => {  
 console.log(data);  
 };  
 value = () => {  
 let *num* = this.state.num;  
 *num*++;  
 this.setState({ *num* });  
 //给输入框添加值，给SearchBox加ref，然后调用这句代码  
 this.formItem.setValue({ key: '556546455', value: *num*, id: 'CNKFHKFJNVKNVKNVKJFJFJFJFJVN' });  
 };  
 render() {  
 const *data* = [  
 {  
 key: '000', //(必填)  
 title: '名称', //输入框的标题，若没有，则输入''  
 type: 'input', //input：普通输入框；select：多选或单选下拉框（必填）  
 name: 'aa', //输入框的名称，用于获取到每个输入框内容的标识（必填）  
 placeholder: '请输入名称', //占位提示内容  
 defaultValue: '小花花', //初始值  
 required: true, //输入框是否比填红色小标识  
 iconType: '', //输入框后面的icon按钮（如预览按钮,提示符按钮）  
 disabled: false, //输入框可编辑状态  
 addonBefore: '10%', //输入框前缀内容  
 tip: '提示\_符号（例如权限名称\_ADMIN）', //输入框的提示符  
 width: '28%' //自定义输入框的宽度  
 },  
 {  
 key: '001', //(必填)  
 title: 'URL',  
 type: 'input',  
 name: 'bb',  
 placeholder: '请输入URL',  
 defaultValue: '我是url',  
 required: true,  
 iconType: '',  
 disabled: false,  
 tip: '',  
 width: '28%'  
 },  
 {  
 key: '002', //(必填)  
 title: '父节点名称',  
 type: 'input',  
 name: 'cc',  
 placeholder: '请输入父节点名称',  
 defaultValue: '平台角色',  
 required: false,  
 iconType: '',  
 disabled: true,  
 tip: '',  
 width: '28%'  
 },  
 {  
 key: '003', //(必填)  
 title: 'ICON',  
 type: 'input',  
 name: 'dd',  
 placeholder: '请输入名称',  
 defaultValue: 'yuytyu',  
 required: false,  
 iconType: 'preview',  
 disabled: false,  
 tip: '',  
 width: '28%'  
 },  
 {  
 key: '004', //(必填)  
 title: '描述',  
 type: 'input',  
 name: 'ee',  
 placeholder: '请输入描述内容',  
 defaultValue: '我是猫咪',  
 required: false,  
 iconType: '',  
 disabled: false,  
 tip: '',  
 width: '100%'  
 },  
 {  
 key: '005', //（必填）  
 title: '类型',  
 name: 'ff', //（必填）  
 type: 'select', //（必填）  
 iconType: '',  
 placeholder: '请输入',  
 defaultValue: ['集团', '公司'], //若为单选，直接填入selectData数据里的key值（string），若为多选，填入数组来装多个key值  
 choice: 'radio', //radio:单选，multiple:多选（必填）  
 width: '100%',  
 tip: '',  
 options: [  
 { key: '0001', value: '集团' },  
 { key: '0002', value: '公司' },  
 { key: '0003', value: '部门' },  
 { key: '0004', value: '小组' }  
 ] //选择下拉框里面的选项内容  
 },  
 {  
 key: '006', //（必填）  
 title: '请求方式',  
 name: 'ff', //（必填）  
 type: 'select', //（必填）  
 iconType: '',  
 placeholder: '请选择',  
 defaultValue: ['全方法', 'GET'], //若为单选，直接填入selectData数据里的key值（string），若为多选，填入数组来装多个key值  
 choice: 'multiple', //radio:单选，multiple:多选（必填）  
 width: '100%',  
 tip: '',  
 options: [  
 { key: '0001', value: '全方法' },  
 { key: '0002', value: 'GET' },  
 { key: '0003', value: 'POST' },  
 { key: '0004', value: 'PUT' },  
 { key: '0005', value: 'DELETE' },  
 { key: '0006', value: 'PATCH' }  
 ]  
 },  
 {  
 key: '454355',  
 title: '所属',  
 name: 'belong',  
 type: 'preInput',  
 iconType: '',  
 placeholder: '请输入',  
 required: false, //小红点  
 prePlaceholder: '归属类型',  
 defaultValue: '',  
 preDefaultValue: '02',  
 preData: [  
 { key: '01', value: '无', disabled: false },  
 { key: '02', value: '平台', disabled: true },  
 { key: '03', value: '公司', disabled: false }  
 ],  
 width: 230  
 },  
 {  
 key: '556546455',  
 title: '点击回调',  
 name: 'leader',  
 type: 'clickInput',  
 placeholder: '请选择',  
 required: true, //小红点  
 defaultValue: [], //必须为数组，不可为字符串  
 // choice: 'radio', //radio:单选，multiple:多选（必填）  
 // errText: '添加失败，值已存在', //若为单选，添加第二个值时的提示语  
 width: 230,  
 callback: (key, name) => {  
 console.log('点击弹出模态框' + key, name);  
 }  
 }  
 ];  
 return (  
 <div>  
 <FormItem data={*data*} handleChange={this.change} ref={(ref) => (this.formItem = ref)} />  
 <div onClick={this.value}>给点击回调添加value</div>  
 </div>  
 );  
 }  
}

### 父组件index：

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import classNames from 'classnames';  
import InputField from 'src/Components/Public/CreateEditBar/InputField/index'; //输入框组件  
import SelectField from 'src/Components/Public/CreateEditBar/SelectField/index'; //多选或单选下拉框组件  
import PreInput from 'src/Components/Public/FormItem/PreInput/index';  
import ClickInput from 'src/Components/Public/FormItem/ClickInput/index';  
import style from './css.less';  
  
export default class FormItem extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 data: '',  
 yanData: props.data,  
 defaultObj: {}, //默认值对象  
 obj: {} //输入值对象  
 };  
 }  
 static *propTypes* = {  
 data: PropTypes.array.isRequired,  
 handleChange: PropTypes.func.isRequired  
 };  
 static *defaultProps* = {  
 data: [],  
 handleChange: () => {}  
 };  
 componentWillMount() {  
 this.defaultValue(this.props);  
 }  
  
 componentWillReceiveProps(props) {  
 if (JSON.stringify(props.data) !== this.state.data) {  
 this.defaultValue(props);  
 }  
 }  
  
 shouldComponentUpdate(nextProps, nextState) {  
 if (JSON.stringify(nextProps.data) !== this.state.data) {  
 return true;  
 } else {  
 return false;  
 }  
 }  
 */\*\*  
 \* 获取初始数据  
 \** ***@param*** *{object} props  
 \** ***@return*** *{\*}  
 \*/* defaultValue = (props) => {  
 if (props.data.length > 0) {  
 let *defaultObj* = {};  
 props.data.forEach((item) => {  
 *defaultObj*[item.name] = item.defaultValue;  
 if (item.type === 'preInput') {  
 *defaultObj*[item.name] = {};  
 *defaultObj*[item.name].preSelect = item.preDefaultValue;  
 *defaultObj*[item.name].preValue = item.defaultValue;  
 }  
 });  
 let *info* = JSON.stringify(props.data);  
 this.setState({  
 *defaultObj*,  
 data: *info* });  
 }  
 };  
 */\*\*  
 \* 获取输入的数据  
 \** ***@param*** *{string} text  
 \** ***@param*** *{string} name  
 \** ***@param*** *{type} type input=输入框，select=下拉框  
 \** ***@return*** *{\*}  
 \*/* handleChange = (text, name, type) => {  
 let *obj* = this.state.defaultObj;  
 if (type === 'common') {  
 *obj*[name] = text;  
 } else {  
 switch (type) {  
 case 'preSelect':  
 *obj*[name].preSelect = text;  
 break;  
 case 'preValue':  
 *obj*[name].preValue = text;  
 break;  
 default:  
 break;  
 }  
 }  
 this.props.handleChange(*obj*);  
 };  
 */\*\*  
 \* 设置点击选择框的值  
 \*/* setValue = ({ key, ...obj } = data) => {  
 this[key].setValue({ ...obj });  
 };  
 */\*\*  
 \* 动态组件渲染  
 \** ***@return*** *{\*}  
 \*/* showInput = () => {  
 const { data } = this.props;  
 return data.map((item) => {  
 const {  
 *type*,  
 *key*,  
 *index*,  
 *title*,  
 *iconType*,  
 *name*,  
 *placeholder*,  
 *width*,  
 *defaultValue*,  
 *addonBefore*,  
 *disabled*,  
 *required*,  
 *options*,  
 *choice*,  
 *Interface*,  
 *callBack* } = item;  
  
 const *obj* = {  
 input: (  
 <InputField  
 key={*key*}  
 index={*index*}  
 label={*title*}  
 iconType={*iconType*}  
 name={*name*}  
 placeholder={*placeholder*}  
 width={*width*}  
 defaultValue={*defaultValue*}  
 addonBefore={*addonBefore*}  
 disabled={*disabled*}  
 required={*required*}  
 handleChange={(text1, text2) => this.handleChange(text1, text2, 'common')}  
 />  
 ),  
 select: (  
 <SelectField  
 key={*key*}  
 label={*title*}  
 name={*name*}  
 placeholder={*placeholder*}  
 defaultValue={*defaultValue*}  
 choice={*choice*}  
 width={*width*}  
 selectData={*options*}  
 disabled={*disabled*}  
 required={*required*}  
 handleChange={(text1, text2) => this.handleChange(text1, text2, 'common')}  
 Interface={*Interface*}  
 callBack={*callBack*}  
 />  
 ),  
 preInput: (  
 <PreInput  
 key={*key*}  
 label={*title*}  
 name={*name*}  
 placeholder={*placeholder*}  
 prePlaceholder={item.prePlaceholder}  
 defaultValue={*defaultValue*}  
 preDefaultValue={item.preDefaultValue}  
 width={*width*}  
 iconType={*iconType*}  
 required={*required*}  
 preData={item.preData}  
 handleChange={(text1, text2, type1) => this.handleChange(text1, text2, type1)}  
 />  
 ),  
 clickInput: (  
 <ClickInput  
 ref={(ref) => (this[*key*] = ref)}  
 key={*key*}  
 label={*title*}  
 name={*name*}  
 placeholder={*placeholder*}  
 defaultValue={*defaultValue*}  
 width={*width*}  
 iconType={*iconType*}  
 required={*required*}  
 choice={*choice*}  
 errText={item.errText}  
 callback={item.callback}  
 handleChange={(text1, text2) => this.handleChange(text1, text2, 'common')}  
 />  
 )  
 };  
 return *obj*[*type*];  
 });  
 };  
 render() {  
 const { *className*, data } = this.props;  
 return <div className={classNames(style.init, *className*)}>{this.showInput()}</div>;  
 }  
}

### 子组件InputField

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import { Button, *Icon*, Input } from 'antd';  
import style from './css.less';  
  
export default class InputField extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 value: props.defaultValue  
 };  
 }  
 static *propTypes* = {  
 label: PropTypes.string, //输入框的标题，若没有，则输入''  
 name: PropTypes.string.isRequired, //输入框的名称，用于查询获取到每个输入框内容的标识  
 placeholder: PropTypes.string, //占位提示内容  
 defaultValue: PropTypes.oneOfType([PropTypes.string, PropTypes.number]), //初始值  
 width: PropTypes.oneOfType([PropTypes.string, PropTypes.number]), //一个输入框的宽度，包括label，可输入百分比（string）和数字（number）  
 handleChange: PropTypes.func.isRequired //获取输入内容的回调  
 };  
 static *defaultProps* = {  
 label: '',  
 name: '',  
 placeholder: '请输入内容',  
 defaultValue: '',  
 width: '28%',  
 handleChange: () => {}  
 };  
 componentWillReceiveProps(props) {  
 const { value } = this.state;  
 if (value !== props.defaultValue) {  
 this.setState({  
 value: props.defaultValue  
 });  
 }  
 }  
  
 //输入时获取内容  
 onChange = (e, name) => {  
 this.setState({ value: e.target.value });  
 this.props.handleChange(e.target.value, name);  
 };  
  
 //图标按钮  
 iconShow = (iconType) => {  
 switch (iconType) {  
 case 'preview':  
 return (  
 <div className={style.iconDiv}>  
 <Button type="primary" className={style.iconButton}>  
 预览  
 </Button>  
 </div>  
 );  
 case 'info-circle':  
 return (  
 <div className={style.iconDiv}>  
 <Icon  
 type={iconType}  
 className={style.iconPoint}  
 // style={{ marginLeft: '10px', cursor: 'pointer', color: '#2a99e2' }}  
 // onClick={(key) => this.iconTip(item)}  
 />  
 </div>  
 );  
 default:  
 return '';  
 }  
 };  
  
 render() {  
 let { placeholder, label, width, name, *disabled*, *addonBefore*, *required*, *iconType*, *index*, defaultValue } = this.props;  
 return (  
 <div className={style.item} style={{ width: width }}>  
 {label.length > 0 ? (  
 <div className={style.itemTitle}>  
 {*required* == true ? <span className={style.point}>\*</span> : ''}  
 {label}：  
 </div>  
 ) : (  
 ''  
 )}  
 <div className={style.inputDiv}>  
 <Input  
 placeholder={placeholder}  
 value={this.state.value}  
 addonBefore={*addonBefore*}  
 style={{ flex: 1 }}  
 disabled={*disabled*}  
 onChange={(e) => this.onChange(e, name)}  
 />  
 {this.iconShow(*iconType*)}  
 </div>  
 </div>  
 );  
 }  
}

### 子组件SelectField

*/\*\** ***@format*** *\*/*import React, { findDOMNode, Component } from 'react';  
import PropTypes from 'prop-types';  
import { Select } from 'antd';  
import style from './css.less';  
import httpFetch from '../../../../util/httpFetch';  
const { *Option* } = Select;  
  
export default class SelectField extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 value: props.defaultValue,  
 defaultValue: props.defaultValue,  
 options: props.selectData  
 };  
 }  
 static *propTypes* = {  
 label: PropTypes.string, //输入框的标题，若没有，则输入''  
 name: PropTypes.string.isRequired, //输入框的名称，用于查询获取到每个输入框内容的标识  
 placeholder: PropTypes.string, //占位提示内容  
 defaultValue: PropTypes.oneOfType([PropTypes.string, PropTypes.array]), //初始值  
 choice: PropTypes.string.isRequired, //radio:单选，multiple:多选（必填）  
 width: PropTypes.oneOfType([PropTypes.string, PropTypes.number]),  
 selectData: PropTypes.array.isRequired,  
 handleChange: PropTypes.func.isRequired,  
 Interface: PropTypes.object, //接口的配置与callBack一起使用  
 callBack: PropTypes.func //有接口时的回调函数与Interface一起使用  
 };  
 static *defaultProps* = {  
 placeholder: '请输入内容',  
 label: '',  
 name: '',  
 defaultValue: [],  
 selectData: [],  
 width: '32%',  
 choice: 'radio',  
 handleChange: () => {}  
 };  
 componentWillMount() {  
 if (this.props.Interface) {  
 this.getOptions();  
 }  
 }  
 componentWillReceiveProps(props) {  
 const { defaultValue } = this.state;  
 if (defaultValue !== props.defaultValue) {  
 this.setState({  
 value: props.defaultValue  
 });  
 }  
 }  
 */\*\*  
 \* 如果有接口，那么就需要动态获取option  
 \** ***@return*** *{\*}  
 \*/* getOptions = () => {  
 const {  
 Interface: { *url*, *method*, *param*, *config* }  
 } = this.props;  
 httpFetch(*method*, *url*, *param*, *config*).then((res) => {  
 let *data* = this.props.callBack(res);  
 this.setState({ options: *data* });  
 });  
 };  
 //输入时获取内容  
 onChange = (e, name) => {  
 this.setState({ value: e });  
 this.props.handleChange(e, name);  
 };  
  
 render() {  
 let { placeholder, label, defaultValue, choice, width, name, *required*, *disabled* } = this.props;  
 const { options } = this.state;  
 return (  
 <div className={style.item} style={{ width: width }}>  
 {label.length > 0 ? (  
 <div className={style.itemTitle}>  
 {*required* === true ? <span className={style.point}>\*</span> : ''}  
 {label}：  
 </div>  
 ) : (  
 ''  
 )}  
 <Select  
 disabled={*disabled*}  
 placeholder={placeholder}  
 style={{ flex: 1 }}  
 defaultValue={defaultValue}  
 mode={choice}  
 value={this.state.value}  
 onChange={(e) => this.onChange(e, name)}  
 >  
 {options.map((el) => {  
 return (  
 <Option value={el.key} key={el.key}>  
 {el.value}  
 </Option>  
 );  
 })}  
 </Select>  
 </div>  
 );  
 }  
}

### 子组件PreInput

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import { Input, Select, Button, *Icon* } from 'antd';  
import style from './css.less';  
import classNames from 'classnames';  
const { *Option* } = Select;  
  
export default class PreInput extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 value: props.defaultValue,  
 selectValue: props.preDefaultValue,  
 disabled: false  
 };  
 }  
 static *propTypes* = {  
 label: PropTypes.string, //输入框的标题，若没有，则输入''  
 name: PropTypes.string.isRequired, //输入框的名称，用于查询获取到每个输入框内容的标识  
 placeholder: PropTypes.string, //占位提示内容  
 defaultValue: PropTypes.oneOfType([PropTypes.string, PropTypes.array]), //输入框初始值  
 preDefaultValue: PropTypes.string, //前缀初始值  
 prePlaceholder: PropTypes.string, //前缀占位符  
 preData: PropTypes.array.isRequired, //前缀下来列表  
 width: PropTypes.oneOfType([PropTypes.string, PropTypes.number]), //输入框的宽度（number）  
 handleChange: PropTypes.func.isRequired //获取输入内容的回调  
 };  
 static *defaultProps* = {  
 label: '',  
 name: '',  
 placeholder: '请输入内容',  
 prePlaceholder: '归属类型',  
 defaultValue: '',  
 preDefaultValue: '',  
 preData: [],  
 width: 230,  
 handleChange: () => {}  
 };  
 componentWillMount() {  
 this.inDisable(this.props);  
 }  
 componentWillReceiveProps(props) {  
 const { *value*, *selectValue* } = this.state;  
 if (*value* !== props.defaultValue || *selectValue* !== props.preDefaultValue) {  
 this.setState({  
 value: props.defaultValue,  
 selectValue: props.preDefaultValue  
 });  
 }  
 this.inDisable(props);  
 }  
 */\*\*  
 \* 初始化disable  
 \*/* inDisable = (props) => {  
 const { *preDefaultValue*, *preData* } = props;  
 *preData*.forEach((item) => {  
 if (item.key === *preDefaultValue*) {  
 this.setState({  
 disabled: item.disabled  
 });  
 }  
 });  
 };  
 */\*\*  
 \* 输入时获取内容  
 \*/* onChange = (e, name) => {  
 this.setState({ value: e.target.value });  
 this.props.handleChange(e.target.value, name, 'preValue');  
 };  
 */\*\*  
 \* 前缀变化时获取内容  
 \*/* selectChange = (value, text, name) => {  
 let *boolText* = text.key.split(',')[1];  
 let *bool*;  
 if (*boolText* === 'true') {  
 *bool* = true;  
 } else if (*boolText* === 'false') {  
 *bool* = false;  
 }  
 this.setState({  
 disabled: *bool*,  
 value: '',  
 selectValue: value  
 });  
 this.props.handleChange(value, name, 'preSelect');  
 this.props.handleChange('', name, 'preValue');  
 };  
 */\*\*  
 \* 图标按钮  
 \*/* iconShow = (iconType) => {  
 switch (iconType) {  
 case 'preview':  
 return (  
 <div className={style.iconDiv}>  
 <Button type="primary" className={style.iconButton}>  
 预览  
 </Button>  
 </div>  
 );  
 case 'info-circle':  
 return (  
 <div className={style.iconDiv}>  
 <Icon  
 type={iconType}  
 className={style.iconPoint}  
 // style={{ marginLeft: '10px', cursor: 'pointer', color: '#2a99e2' }}  
 // onClick={(key) => this.iconTip(item)}  
 />  
 </div>  
 );  
 default:  
 return '';  
 }  
 };  
 render() {  
 const {  
 *placeholder*,  
 *prePlaceholder*,  
 *label*,  
 *width*,  
 *line*,  
 *name*,  
 *preData*,  
 *preDefaultValue*,  
 *required*,  
 *iconType* } = this.props;  
 const { *disabled*, *selectValue*, *value* } = this.state;  
 const *selectBefore* = (  
 <Select  
 style={{ width: 100 }}  
 placeholder={*prePlaceholder*}  
 value={*selectValue*}  
 defaultValue={*preDefaultValue*}  
 onChange={(value, text) => this.selectChange(value, text, *name*)}  
 >  
 {*preData*.map((el) => {  
 return (  
 <Option value={el.key} key={el.key + ',' + el.disabled}>  
 {el.value}  
 </Option>  
 );  
 })}  
 </Select>  
 );  
 return (  
 <div className={style.item} style={{ width: *width* }}>  
 {*label*.length > 0 ? (  
 <div className={style.itemTitle}>  
 {*required* == true ? <span className={style.point}>\*</span> : ''}  
 {*label*}：  
 </div>  
 ) : (  
 ''  
 )}  
 <div className={style.inputDiv}>  
 <Input  
 addonBefore={*selectBefore*}  
 placeholder={*placeholder*}  
 value={*value*}  
 title={*value*}  
 style={{ flex: 1 }}  
 onChange={(e) => this.onChange(e, *name*)}  
 disabled={*disabled*}  
 />  
 {this.iconShow(*iconType*)}  
 </div>  
 </div>  
 );  
 }  
}

### 子组件ClickInput

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import { *Icon*, Select, *message* } from 'antd';  
import style from './css.less';  
import classNames from 'classnames';  
  
export default class ClickInput extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 value: JSON.stringify(props.defaultValue),  
 defaultValue: props.defaultValue  
 };  
 }  
 static *propTypes* = {  
 label: PropTypes.string, //输入框的标题，若没有，则输入''  
 name: PropTypes.string.isRequired, //输入框的名称，用于查询获取到每个输入框内容的标识  
 placeholder: PropTypes.string, //占位提示内容  
 defaultValue: PropTypes.array, //初始值  
 width: PropTypes.oneOfType([PropTypes.string, PropTypes.number]), //输入框的宽度（number）  
 callback: PropTypes.func.isRequired, //点击图标的回调  
 handleChange: PropTypes.func.isRequired //获取输入内容的回调  
 };  
 static *defaultProps* = {  
 label: '',  
 name: '',  
 placeholder: '请输入内容',  
 defaultValue: '',  
 width: 230,  
 callback: () => {},  
 handleChange: () => {}  
 };  
  
 */\*\*  
 \* 获取值  
 \*/* setValue = (obj) => {  
 const { *value* } = this.state;  
 const { *choice*, *errText* } = this.props;  
 let *valueArr* = JSON.parse(*value*);  
 if (*choice* === 'radio') {  
 if (*valueArr*.length > 0) {  
 *message*.error(*errText*);  
 return;  
 }  
 }  
 const { *name* } = this.props;  
 *valueArr*.push(obj);  
 this.setState({ value: JSON.stringify(*valueArr*) });  
 this.props.handleChange(*valueArr*, *name*);  
 };  
 */\*\*  
 \* 重置  
 \*/* reset = () => {  
 this.setState({ value: JSON.stringify([]) });  
 };  
 */\*\*  
 \* 取消选种时回调  
 \*/* onDeselect = (text) => {  
 const { *value* } = this.state;  
 const { *name* } = this.props;  
 let *valueArr* = JSON.parse(*value*);  
 *valueArr*.forEach((item, ind) => {  
 if (item.value === text) {  
 *valueArr*.splice(ind, 1);  
 }  
 });  
 this.props.handleChange(*valueArr*, *name*);  
 this.setState({  
 value: JSON.stringify(*valueArr*)  
 });  
 };  
 */\*\*  
 \* 回显value  
 \*/* showValue = (valueStr) => {  
 return JSON.parse(valueStr).map((item) => {  
 return item.value;  
 });  
 };  
 render() {  
 const { *placeholder*, *label*, *width*, *line*, *name*, *callback*, *required* } = this.props;  
 const { *value* } = this.state;  
 return (  
 <div className={classNames(style.item, { [style.item1]: *line* })}>  
 {*label*.length > 0 ? (  
 <div className={style.itemTitle}>  
 {*required* === true ? <span className={style.point}>\*</span> : ''}  
 {*label*}：  
 </div>  
 ) : (  
 ''  
 )}  
 <div className={style.inputDiv}>  
 <Select  
 placeholder={*placeholder*}  
 mode={'multiple'}  
 value={this.showValue(*value*)}  
 title={this.showValue(*value*)}  
 style={{ width: *width* }}  
 onDeselect={this.onDeselect}  
 />  
 <Icon type="menu-fold" style={{ color: '#2a99e2' }} onClick={() => *callback*(*name*)} />  
 </div>  
 </div>  
 );  
 }  
}

## 关键字搜索

### 调用：

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import KeyInput from 'src/Components/Public/KeyInput/index';  
import *api* from '../../../App.api';  
  
export default class TableModalTest extends Component {  
 render() {  
 //必传的只有clickAdd和callback两个回调函数，其他可不传，有默认值  
 const *data* = {  
 label: '', //标题（无-可不传，或传空字符串）  
 placeholder: '请输入关键字',  
 width: 300,  
 //点击下拉框的‘添加按钮’的回调  
 clickAdd: (id, userName) => {  
 console.log(id, userName);  
 },  
 //text为输入的内容，根据该内容自行请求接口，将请求回的数据处理成[{id:xx,userName:xx}]的格式，调用this.keyInput.selectData(info)，将数组传递进去（要给组件先加ref）  
 callback: (text) => {  
 *api*.userGroup.findUser(text).then((res) => {  
 if (res.op === 'suc') {  
 let *info* = res.data.map((item) => {  
 let *obj* = {  
 id: item.id,  
 userName: item.nickname  
 };  
 return *obj*;  
 });  
 this.keyInput.selectData(*info*);  
 }  
 });  
 }  
 };  
 return (  
 <div>  
 <KeyInput {...*data*} ref={(ref) => (this.keyInput = ref)} />  
 </div>  
 );  
 }  
}

### 组件：

*/\*\** ***@format*** *\*/*import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
import { Input, Spin, Button } from 'antd';  
import debounce from 'lodash/debounce';  
import style from './css.less';  
  
export default class TableModal extends Component {  
 constructor() {  
 super();  
 this.state = {  
 value: [],  
 data: [],  
 visible: false, //控制整个下拉框是否出现  
 fetching: false //控制加载圈圈  
 };  
 this.fetchUser = debounce(this.fetchUser, 800);  
 }  
 static *propTypes* = {  
 label: PropTypes.string,  
 placeholder: PropTypes.string,  
 width: PropTypes.number, //输入框的宽度（number）  
 clickAdd: PropTypes.func,  
 callback: PropTypes.func  
 };  
 static *defaultProps* = {  
 label: '',  
 placeholder: '请输入关键字',  
 width: 300, //输入框的宽度（number）  
 clickAdd: PropTypes.func,  
 callback: PropTypes.func  
 };  
 */\*\*  
 \* 根据输入的信息请求接口  
 \** ***@param*** *{string} value 输入框内容  
 \** ***@return*** *{\*}  
 \*/* fetchUser = (value) => {  
 if (value !== '') {  
 this.setState({  
 data: [],  
 fetching: true,  
 visible: true  
 });  
 this.props.callback(value);  
 }  
 };  
 */\*\*  
 \* 获取搜索到的数据  
 \*/* selectData = (data) => {  
 this.setState({  
 data,  
 fetching: false  
 });  
 };  
 */\*\*  
 \* 获取焦点时的回调  
 \*/* onFocus = () => {  
 this.setState({  
 data: [],  
 visible: false  
 });  
 };  
 */\*\*  
 \* 点击添加的回调  
 \*/* clickAdd = (id, userName) => {  
 this.props.clickAdd(id, userName);  
 };  
  
 render() {  
 const { data, fetching, visible } = this.state;  
 const { width, placeholder, *Interface*, label } = this.props;  
 return (  
 <div className={style.item}>  
 {label.length > 0 ? <span className={style.label}>{label}：</span> : null}  
 <Input  
 placeholder={placeholder}  
 onChange={(e) => this.fetchUser(e.target.value)}  
 style={{ width: width }}  
 onFocus={this.onFocus}  
 />  
 {visible ? (  
 <Spin size="small" spinning={fetching}>  
 <ul style={{ width: width - 2 }}>  
 {data.length > 0 ? (  
 data.map((item) => {  
 return (  
 <li key={item.id} className="lis">  
 <span>{item.userName}</span>  
 <Button onClick={() => this.clickAdd(item.id, item.userName)}>添加</Button>  
 </li>  
 );  
 })  
 ) : (  
 <li className="lis">查询无果</li>  
 )}  
 </ul>  
 </Spin>  
 ) : null}  
 </div>  
 );  
 }  
}

### Css:

/\*\* @format \*/  
.item {  
 margin-right: 10px;  
 height: 45px;  
 display: flex;  
 align-items: center;  
 position: relative;  
  
 .label {  
 width: 84px;  
 text-align: right;  
 }  
 :global {  
 .ant-spin-nested-loading {  
 z-index: 100;  
 background-color: #fff;  
 border-left: 1px solid #000;  
 border-right: 1px solid #000;  
 position: absolute;  
 top: 40px;  
 left: 0;  
 ul {  
 list-style-type: none;  
 padding: 0;  
 margin: 0;  
 .lis {  
 height: 40px;  
 display: flex;  
 padding: 0 5px;  
 justify-content: space-between;  
 align-items: center;  
 border-bottom: 1px solid #ccc;  
 .ant-btn {  
 width: 56px;  
 height: 28px;  
 }  
 }  
 }  
 }  
 }  
}