Antd UI 组件：

1. Form表单

1.1：栅栏是24格。

1.2：两种排列方式：

1.2.1：水平排列，可以实现 label 标签和表单控件的水平排列;

<Form horizontal onSubmit={this.handleSubmit}>

1.2.2：行内排列，使其表现为 inline-block 级别的控件。

<Form inline onSubmit={this.handleSubmit}>

1.3：使用Form.create 来获取和更新表单提交的数值。

let Demo = React.createClass({});

Demo = Form.create()(Demo);

1.4：表单控件定义了三种校验状态，为 <FormItem> 定义 validateStatus 属性即可。

validateStatus: ['success', 'warning', 'error', 'validating']

1.5：另外为输入框添加反馈图标，设置 <FormItem> 的 hasFeedback 属性值 为 true 即可。注意: 反馈图标只对 <Input /> 有效。

1.6：如果想要input和你想要的图标组合的话，需要用到Input.Group

1.7：<formItem>组件常用的属性label、labelCol、wrapperCol。

Label：标签名；

labelCol：标签名的样式布局；

wrapperCol：输入控件的样式布局。



创建reactApp的几种方式：

1. create-react-app filename 适用于npm6及以下。
2. npm init react-app filename 适用于npm6以上。
3. npx create-react-app filename 适用于npm5.2以上。

利用脚手架创建vue项目的几种方式：

1. vue create filename 适用于vue-cli3.0以上。
2. vue init webpack filename 适用于 vue-cli2。

React之reflux：

1. 创建action和store：

1.1：reflux.createAction()： 创建一个action；返回值是一个函数，调用这个函数就

会触发相应的事件，在store中监听这个函数，并作相 应的处理。

reflux.createActions([])：创建多个action；返回值是一个对象，包含多个函数。

1.2：reflux.createStore({}); 创建store实例，返回一个对象。

1.3：reflux.ActionMethods.pck = function(){}; 创建action方法的公用方法。

单个action使用公用方法：action.pck();

多个action使用公用方法：action.add.pck();

1.4：reflux.StoreMethods.pck = function(){}; 创建Store的公用方法。

1.5：拓展store的公用方法：

1.5.1：reflux.StoreMethods.pck = function(){}; //pck是定义的方法名。

1.5.2：let mixin = { pck: function(){} };

let store = reflux.createStore({ mixins:[mixin], ...});

1. Store监听action：

let actions = reflux.createActions([ ‘add’, ‘delete’, ‘check’]);

let actions1 = reflux.createAction();

let store = reflux.createStore({ init: function(){}, ...});

2.1：this.listenTo( actions.add, “add”); 第一个参数是actions对象的方法。第二个参

数是store对象监听action的方法调用的方法名。如果是actions1的话，那么 写法就是this.listenTo( actions1, “actions1”); first和second参数名不需要一致。

2.2：this.listenToMany(actions); 参数actions必须是json对象。

处理方法的写法只需让action的标识首字母大写并加上on就可以了。

例如：add→onAdd。

2.3：listenables: [actions]; 属性值actions必须是json对象。

处理方法的写法只需让action的标识首字母大写并加上on就可以了。

例如：add→onAdd。

注意：2.1和2.2都是写在init函数内部，2.3方法是属性写法，listenables是配置 属性。并且2.2和2.3的参数或者属性值必须是createActions创建的action。

1. 异步action ：let asyncAction = Reflux.createAction({asyncResult: true});



1. Action hooks：Reflux为每个action都提供了两个hook方法。

4.1：preEmit(params) ，action emit之前调用，参数是action传递过来的，返回值 会传递给shouldEmit()；

4.2：shouldEmit(params) ，action emit之前调用，参数默认是action传递，如果 preEmit有返回值，则是preEmit返回值；shouldEmit()的返回值决定是否 触发action事件。

4.3：注意，如果preEmit有返回值，则该返回值会作为action事件的参数；否则就 是action的初始参数；不论shouldEmit的返回值是什么，它只是决定是否触 发action事件。

1. 同组件结合：

5.1：store.listen(func); func：被监听的函数。返回值是解除store监听的一个函数。

5.1.1：当组件的生命周期结束时需要解除对Store的监听。

5.1.2： 当Store调用trigger时，才会执行func函数，所以每次Store 更新时，需要手动调用trigger函数。

5.1.3：并且store.trigger(params); 的参数将是func的参数（如果有）。

5.2：mixins: [Reflux.ListenerMixin] + store.listen(func);

5.2.1：适用于作为React.createClass({})的配置属性。

例如：let test = React.createClass({ mixins: [Reflux.ListenerMixin] });

5.2.2：接下来的写法和5.1方法一样，不过没有了5.1.1的限制。

5.3：mixins: [Reflux.listenTo(store, “func”)]

5.3.1：适用于作为React.createClass({})的配置属性。

例如：let test = React.createClass({mixins: [Reflux.listenTo(store, “func”)] });

5.3.2：它比5.2更优化的是，没有了store.listen(func); 但是也必须要手动触发 store.trigger();

5.4：minxins: [Reflux.connect(store, ‘list’)]

5.4.1：它相对应5.3来说，优化了一点，也就是不需要函数来更新state里面 的list，但是还是必须手动触发store.trigger()。

5.4.2：注意，这里的‘list’参数是state里面的属性名（被监听的）。

5.5：mixins: [Reflux.connectFilter(store, ‘list’, function(list){})];

5.5.1：相对于5.4来说，只是增加了一个数据过滤器。

5.5.2：第三个参数就是过滤函数。参数名不必和第二个参数名相同。

5.5.3：还是必须要手动触发 store.trigger(); 。

总结：这5种结合的方法，都必须手动触发store.trigger(); ；

5.4方法与5.5方法的第二个参数名必须和state里面的属性（被监听）名一 致（字符串）。

React-mixin:



createClass中常规属性的许多东西现在都是类的静态属性。要使getDefaultProps，

propTypes和getInitialState正常工作，您需要将react-mixin应用于比原型更高的级别：

类本身。

Localstorage:

let storage = window.localStorage;

1. 写入：

1.1：storage.a = 1;

1.2：storage[‘a’] = 1;

1.3：storage.setItem(‘a’, 1);

1. 删除：

2.1：storage.clean(); 清除localstorage的所有内容。

2.2：storage.removeItem(“a”); 清除某个键值对。

1. 读取：

3.1：let a = storage.getItem(‘a’);

3.2：let a = storage.a;

3.3：let a = storate[‘a’];

注意：在将json对象存入localstorage的时候，需要将其转换为json字符串。

ES6 之 symbol：

1. 概述：它是一种新的数据类型。表示独一无二的值。
2. 用法：let a = Symbol(); (不得用成 let a = new Symbol())。
3. let s1 = Symbol(“d”); let s2 = Symbol(“d”); s1 == s2 //false。
4. Symbol 类型不能直接进行运算。 s1 + s2 //Cannot convert a Symbol value to a number
5. 但是可以转换成字符串和boolean值。
6. Symbol 类型作为对象的属性名：（symbol产生的原因）
   1. 独一无二的特性，保证对象属性不覆盖。
   2. 限制：在用symbol作为属性名时，不得使用 . 运算符添加该属性。

所以我们在使用属性的时候，也不能用 . 运算符。

* 1. 添加属性：let a = {}; let s1 = Symbol(); a[s1] = 2;

let b = { [s1]: 2 };

let c = {}; Object.defineProperty(c, s1, {value: 4});

* 1. 个人理解他保证属性独一无二的原因：就是因为调用和添加的方式。
  2. 然后有一个小bug就是，由于symbol的唯一性，所以不会覆盖，于是就会产 生多余的垃圾属性。

let a = Symbol(‘a’);

let obj = {};

Obj[a] = “1”;

a = Symbol(‘a’);

obj[a] = 2;

console.log(obj); // {Symbol(a): 1, Symbol(a): 2}Symbol(a): 1Symbol(a): 2}

* 1. 如果想debug6.5的话，有两种方法：

6.5.1. 在新建属性之前，delete一下。

6.5.2. 使用Symbol.for(key) 方法。

6.5.3. 需要注意的是，Symbol.for为 Symbol 值登记的名字，是全局环境的， 可以在不同的 iframe 或 service worker 中取到同一个值。

6.7. Symbol.keyFor方法返回一个已登记的 Symbol 类型值的key。

let s1 = Symbol.for(“c”);

console.log(Symbol.keyFor(s1)); // “c”

1. 内置的Symbol值：

7.1：Symbol.hasInstance

概述：对象的Symbol.hasInstance属性，指向一个内部方法。当其他对象使用 instanceof运算符，判断是否为该对象的实例时，会调用这个方法。

用法：const Even = {

[Symbol.hasInstance](obj) {

return Number(obj) % 2 === 0;

}};

6 instance Even //true

7.2：Symbol.isConcatSpreadable

概述：对象的Symbol.isConcatSpreadable属性等于一个布尔值，表示该对象用 于Array.prototype.concat()时，是否可以展开。

用法：let arr1 = [];

arr1[Symbol.isConcatSpreadable] = true;

注意：数组默认是展开的。

类似数组的对象默认是不展开的。（length必须有，不然设置了也不能 展开）

let arr = [‘a’, ‘b’, ‘c’];

let obj = {length: 1, 0: ‘d’}; // 以索引下标为key

arr.concat(obj);//[‘a’, ‘b’, ‘c’, {...}];

obj[Symbol.isConcatSpreadable] = true;

arr.concat(obj); //[‘a’, ‘b’, ‘c’, ‘d’]

7.3：Symbol.species

概述：对象的Symbol.species属性，指向一个构造函数。创建衍生对象时，会 使用该属性。（为了在某些时候希望返回的实例是基类的实例）

用法：class MyArray extends Array {

static get [Symbol.species]() { return Array; }

}

7.4：Symbol.match

概述：对象的Symbol.match属性，指向一个函数。当执行str.match(myObject) 时，如果该属性（Symbol.match）存在，会调用它，返回该方法的返回 值。

用法：class Sd {

[Symbol.match](string) { ... }

}

‘dae’.match(new Sd());

7.5：Symbol.search

概述：对象的Symbol.search属性，指向一个方法，当该对象被 String.prototype.search方法调用时，会返回该方法的返回值。

用法：class Sd {

constructor(value){

this.value = value;

}

[Symbol.search](string){

...

}

}

‘forever’.search(new Sd(“for”));

7.6：Symbol.replace

概述：对象的Symbol.replace属性，指向一个方法，当该对象被 String.prototype.replace方法调用时，会返回该方法的返回值。

用法：Symbol.replace方法会收到两个参数，第一个参数是replace方法正在作 用的对象（Hello），第二个参数是替换后的值（world）。

class Sd{

[Symbol.replace](...s){ ... }}

}

“Hello”.replace(new Sd(), “world”);

7.7：Symbol.split

概述：对象的Symbol.split属性，指向一个方法，当该对象被String.prototype.split 方法调用时，会返回该方法的返回值。

用法：class Sd{

[Symbol.split](str){ ... }

}

‘abcdefg’.split(new Sd(str));

7.8：Symbol.iterator

概述：对象的Symbol.iterator属性，指向该对象的默认遍历器方法。

对象进行for...of循环时，会调用Symbol.iterator方法，返回该对象的默 认遍历器。

用法：class Sd(){

[Symbol.iterator](){ ... }

}

let myCollection = new Sd();

myCollection[0] = 1;

myCollection[1] = 2;

for (let value of myCollection) { ... };

7.9：Symbol.toPrimitive

概述：对象的Symbol.toPrimitive属性，指向一个方法。该对象被转 为原始类型的值时，会调用这个方法，返回该对象对应的原 始类型值。

用法：Symbol.toPrimitive被调用时，会接受一个字符串参数，表示 当前运算的模式，一共有三种模式。

Number：该场合需要转成数值

String：该场合需要转成字符串

Default：该场合可以转成数值，也可以转成字符串

7.10：Symbol.toStringTag

概述：对象的Symbol.toStringTag属性，指向一个方法。在该对象上面调用 Object.prototype.toString方法时，如果这个属性存在，它的返回值会出 现在toString方法返回的字符串之中，表示对象的类型。也就是说，这 个属性可以用来定制[object Object]或[object Array]中object后面的那个 字符串。

7.11：Symbol.unscopables

概述：对象的Symbol.unscopables属性，指向一个对象。该对象指定了使用 with关键字时，哪些属性会被with环境排除。

用法：

// 没有 unscopables 时

class MyClass {

foo() { return 1; }

}

var foo = function () { return 2; };

with (MyClass.prototype) {

foo(); // 1

}

// 有 unscopables 时

class MyClass {

foo() { return 1; }

get [Symbol.unscopables]() {

return { foo: true };

}

}

var foo = function () { return 2; };

with (MyClass.prototype) {

foo(); // 2

}