**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。

链接：https://pan.baidu.com/s/1ly4ItXe4btMh7dHWoROatg 密码：pnrn （angular2视频）

**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

}

**ES6之Set：**

概述：ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有 重复的值。

1. var set = new Set();

Set 函数可以接受一个数组（或者具有 iterable 接口的其他数据结构）作为参数， 用来初始化。

var s = new Set([1,2,3]);

var e = new Set(document.getElementsByTagName(‘div’));

1. Set实例的属性：

2.1：constructor 构造函数，默认就是Set函数。

2.2：size 返回Set实例的成员总数，类似于数组的length属性。

1. Set实例的方法：

3.1：操作方法

var set = new Set();

3.1.1：add(value); 添加某个值（值类型不会发生转换），返回Set实例本身。

set.add(1).add(2).add(3);

3.1.2：delete(value); 删除某个值，返回一个布尔值，表示删除是否成功。

set.delete(1);

3.1.3：has(value); 返回一个布尔值，表示value是否是该Set实例的成员。

set.has(3);

3.1.4：clear(); 清除Set实例的所有成员，没有返回值。

set.clear();

3.2：遍历方法（Set的遍历顺序就是插入顺序）

3.2.1：keys(); 返回键名的遍历器。

3.2.2：values(); 返回键值的遍历器。

3.2.3：entries(); 返回键值对的遍历器。

3.2.4：forEach(); 使用回调函数遍历每个成员。

（function, this(回调函数内部this)）

3.2.5：keys()、values()、entries()方法返回的都是遍历器。

set结构没有键名，只有键值，所以keys()和values()的行为完全一致。

3.2.6：Set 结构的实例默认可遍历，它的默认遍历器生成函数就是它 的values方法。

Set.prototype[Symbol.iterator] === Set.prototype.values // true

这意味着，可以省略values方法，直接用for...of循环遍历 Set。

3.2.7：... 扩展运算符内部使用的是 for...of循环。所以也可以用于 set结构。

3.2.8：在遍历操作中，目前没有直接的方法可以同步改变原来的 Set 结构。

**ES6之WeakSet：**

1. 含义：WeakSet 结构与 Set 类似，也是不重复的值的集合。
2. 与set结构的区别：  
    2.1：WeakSet的成员只能是对象，而不能是其他类型的值。  
    2.2：WeakSet中的对象都是弱引用。
3. 弱引用：即垃圾回收机制不考虑 WeakSet 对该对象的引用，也就 是说，如果其他对象都不再引用该对象，那么垃圾回收机 制会自动回收该对象所占用的内存，不考虑该对象还存在 于 WeakSet 之中。
4. WeakSet的成员有多少室友回收机制有没有运行来决定的。有可能运行前 后成员不一致，所以es6规定WeakSet不可遍历。
5. 垃圾回收机制：垃圾回收机制依赖引用计数，如果一个值的引用次数不为 0，垃圾回收机制就不会释放这块内存。
6. WeakSet 里面的引用，都不计入垃圾回收机制。
7. 创建WeakSet实例：  
    7.1：var ws = new WeakSet();  
    7.2：var ws = new WeakSet([a,b]); a,b必须都是对象，不然会报错。
8. WeakSet实例的方法：  
    8.1：add(obj); 向WeakSet实例添加一个新的成员（对象）。返回本身。  
    8.2：delete(obj); 清除WeakSet实例的指定成员。返回布尔值。  
    8.3：has(obj); 返回一个布尔值，表示该值是否是WeakSet实例的成员。
9. 用处：用于防止内存泄漏。

**ES6之Map：**

**含义：**它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串， 各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提 供了“字符串—值”的对应，Map 结构提供了“值—值”的对应，是 一种更完善的 Hash 结构实现。

1. 创建Map实例：

var map = new Map([[key, value],[key, value]]);

任何具有 Iterator 接口、且每个成员都是一个双元素的数组的数据 结构都可以当作Map构造函数的参数。

该参数类似于Set实例的entries()遍历方法返回的结果。

1. get()/set()

2.1：map.get(key); 返回value。

如果读取一个未知的键，则返回undefined。

2.2：map.set(key, value); 添加成员，返回整个map实例，key不限 类型。

注意：

只有对同一个对象的引用，Map 结构才将其视为同一个键。这一点 要非常小心。

1. 原理：

Map 的键实际上是跟内存地址绑定的，只要内存地址不一样，就视 为两个键。这就解决了同名属性碰撞（clash）的问题，我们扩展别 人的库的时候，如果使用对象作为键名，就不用担心自己的属性与 原作者的属性同名。

1. 如果 Map 的键是一个简单类型的值（数字、字符串、布尔值），则只要 两个值严格相等（类型和值都相等，NaN除外），Map 将其视为一个键。

6. 属性：

6.1：map.size 返回map实例的成员总数。

7. 方法：

7.1：操作方法

7.1.1：get(key)/set(key, value)

7.1.2：has(key) 返回一个布尔值，表示某个键是否在当前map中。

7.1.3：delete(key) 删除某个键，返回布尔值表示删除是否成功。

7.1.4：clear() 清除当前map所有成员，无返回值。

7.2：遍历方法（Map 的遍历顺序就是插入顺序，和Set类似。）

7.2.1：map.keys() 返回键名的遍历器。

7.2.2：map.values() 返回键值的遍历器。

7.2.3：map.entries() 返回所有成员的遍历器。

7.2.4：map.forEach() 遍历map的所有成员。

7.3：转换数据类型

7.3.1：Map => Array [...map]

7.3.2：Array => Map new Map(Array)

7.3.3：Map => Object for ... of

7.3.4：Object => Map new Map(Object.entries(obj))

7.3.5：Map => JSON

1. JSON.stringify(Map => Object)

Map 的键名都是字符串，这时可以选择转为对象 JSON。

1. JSON.stringify(...Map)

Map 的键名有非字符串，这时可以选择转为数组 JSON。

7.3.6：JSON => Map

1. new Map(Object.entries(objjson))
2. new Map(ArrayJson)

JSON是有3种形式：数组json、对象json、混合json。

**ES6之WeakMap：**

含义：WeakMap结构与Map结构类似，也是用于生成键值对的集合。

1. WeakMap与Map的区别有两点：

1.1：WeakMap只接受对象作为键名（null除外），不接受其他类型的值 作为键名。

1.2：WeakMap的键名所指向的对象，不计入垃圾回收机制。

**注意：WeakMap 弱引用的只是键名，而不是键值。键值依然是正常引用。**

1. WeakMap的专用场合就是，它的键所对应的对象，可能会在将来消失。 WeakMap结构有助于防止内存泄漏。
2. WeakMap只有四个方法可以用：

3.1：wm.get(key); 返回键值。没有就返回undefined。

3.2：wm.set(obj, value); 添加成员，返回当前weakmap实例。

3.3：wm.has(key); 返回布尔值，表示是否有该成员。

3.4：wm.delete(key); 返回布尔值，表示删除是否成功。

3.5：没有遍历操作和size属性以及清除方法clear();

1. 与WeakSet一样是弱引用，主要用于防止内存泄漏。
2. var wm = new WeakMap([[obj,value],[obj,value]]);  
    var wm = new WeakMap();

**ES6之proxy：**

1. Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象  
    的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的  
    访问进行过滤和改写。Proxy 这个词的原意是代理，用在这里表示由它  
    来“代理”某些操作，可以译为“代理器”。
2. Proxy 用于修改某些操作的默认行为，等同于在语言层面做出修改，所  
    以属于一种“元编程”（meta programming），即对编程语言进行编程。
3. 创建proxy：

3.1：var proxy = new Proxy(target,handle);

3.1.1：target，需要代理的目标对象。

3.1.2：handle，配置对象。对于每一个被代理的操作，需要提供一 个对应的处理函数，该函数将拦截对应的操作。

注意：要使得Proxy起作用，必须针对Proxy实例进行操作，而不 是针对被代理的目标对象进行操作。

3.2：如果handle参数是空对象（{}）的话，那么访问proxy对象就等 于访问target对象。对proxy操作，就相当于对target进行操 作。但是他们不相等。也就是说他们不指向同一个内存地址。

3.3：在配置任何拦截处理函数时，第一个参数必须是target（最好） 或者{}。不能是基本数据类型（undefined除外）。

3.4：对于可以设置、但没有设置拦截的操作，则直接落在目标对象上， 按照原先的方式产生结果。

1. Proxy支持的13种拦截操作：

4.1：get(target,key,receiver)(目标对象，属性名，操作行为所针对的对象)

拦截对象的读取，例如proxy.name/proxy[‘name’]。

4.2：set(target,key,value,receiver)

拦截对象属性的设置，例如proxy.name/proxy[‘name’] = name;  
 返回一个布尔值。

4.3：deleteProperty(target,key)  
 拦截delete proxy[key]操作，返回一个布尔值。  
 delete不能删除对象的不可配置的属性。

4.4：has(target,key)

拦截key in proxy的操作（for ... in除外），返回一个布尔值。

如果target是不可配置或者禁止扩展，那么拦截  
 （key(存在) in proxy）一定不能返回 false，否则就会报错。

4.5：ownKeys(target)

拦截Object.getOwnPropertyNames(proxy)、 Object.getOwnPropertySymbols(proxy)、 Object.keys(proxy)、 for...in循环，返回一个数组。该方法返 回目标对象所有自身的 属性的属性名，而Object.keys()的返回结果 仅包括目标对象自 身的可遍历属性。

4.6：getOwnPropertyDescriptor(target,key)

拦截Object.getOwnPropertyDescriptor(proxy,key);

返回：{value: "a", writable: true, enumerable: true, configurable: true}  
 {值，可写否，可列举否，可设置否}

该方法的功能是获取对象属性的描述。

4.7：defineProperty(target,key,Desc)(目标对象，键，描述)

拦截Object.defineProperty(proxy,key,Desc)、  
 Object.defineProperties(proxy, propDescs)，返回一个布尔 值，返回自身。  
 propDescs={key1:{...},key2:{...}};  
 Desc = {...};

该方法的功能是，设置成员的描述。

4.8：preventExtensions(target)

拦截Object.preventExtensions(proxy) 返回被阻止的对象。

该方法的功能是阻止对象添加成员，不阻止删减成员。

4.9：isExtensible(target)

拦截Object.isExtensible(proxy);

该方法的功能是检查对象是否可扩展。返回一个布尔值。

4.10：getPrototypeOf(target);

拦截Object.getPrototypeOf(proxy);返回一个对象。

该方法的功能是获取target对象的proto对象。

4.11：setPrototypeOf(target,proto)

拦截Object.setPrototypeOf(proxy,proto);返回一个布尔值。如果 目标对象是函数，那么还有两种额外操作可以拦截。

该方法的功能是设置target对象的proto对象。

4.12：apply(target,object,args)

拦截 Proxy 实例作为函数调用的操作，比如proxy(...args)、 proxy.call(object, ...args)、proxy.apply(...)。

该方法的功能是继承。

4.13：construct(target,args)

拦截 Proxy 实例作为构造函数调用的操作，比如

new proxy(...args)。

1. Proxy.revocable();

5.1：返回一个可取消的proxy实例。

5.2：var prv = Proxy.revocable(target,handle);  
 prv // {proxy: Proxy, revoke: ƒ}  
 proxy: Proxy实例，revoke: 可取消Proxy实例的函数。

5.3：一个使用场景是，目标对象不允许直接访问，必须通过代理访问， 一旦访问结束，就收回代理权，不允许再次访问。

**ES6之Reflect：**

概述：Reflect对象与Proxy对象一样，也是 ES6 为了操作对象而提供的新 API。Reflect对象的设计目的有这样几个。

1. 设计目的：

1.1：将Object对象的一些明显属于语言内部的方法（比如 Object.defineProperty），放到Reflect对象上。

1.2：修改某些Object方法的返回结果，让其变得更合理。

1.3：让Object操作都变成函数行为。

1.3.1：key in obj => Reflect.has(obj,key);  
1.3.2：delete obj[key] => Reflect.deleteProperty(obj,key);

1.4：Reflect对象的方法与Proxy对象的方法一一对应，只要是Proxy 对象的方法，就能在Reflect对象上找到对应的方法。  
  
 这就让Proxy对象可以方便地调用对应的Reflect方法，完成默 认行为，作为修改行为的基础。

1. 静态方法：Reflect对象一共有13个静态方法。

2.1：Reflect.get(target,name,reciver)  
 Reflect.get方法查找并返回target对象的name属性，如果没有 该属性，则返回undefined。

注意：如果name属性设置了读取函数（getter），则读取函数的 this绑定receiver。

2.2：Reflect.set(target,name,value,reciver)  
 Reflect.set方法设置target对象的name属性的值等于value。  
  
 注意：如果name属性设置了赋值函数（setter），则赋值函数的 this绑定receiver。target不是对象会报错。

如果 Proxy 对象和 Reflect 对象联合使用，前者拦截赋值 操作，后者完成赋值的默认行为，而且传入了receiver，那 么Reflect.set会触发Proxy.defineProperty拦截。（当 然，proxy对象设置了defineProperty拦截，且传入了 receiver参数）

2.3：Reflect.has(target,key)  
 Reflect.has方法对应name in obj里面的in运算符。

2.4：Reflect.deleteProperty(target,key)  
  
 Reflect.deleteProperty方法等同于delete obj[name]，用于删 除对象的属性。  
 **该方法返回一个布尔值。**如果删除成功，或者被删除的属性不存在， 返回true；删除失败，被删除的属性依然存在，返回false。

2.5：Reflect.ownKeys(target)

Reflect.ownKeys方法用于返回对象的所有属性，基本等同于 Object.getOwnPropertyNames与Object.getOwnPropertySymbols 之和。

2.6：Reflect.defineProperty(target,name,descObj)

Reflect.defineProperty方法基本等同于  
 Object.defineProperty，用来为对象定义属性。未来，后者会被 逐渐废除，请从现在开始就使用Reflect.defineProperty代替它。   
 如果Reflect.defineProperty的第一个参数不是对象，就会抛出 错误。

2.7：Reflect.getOwnPropertyDescriptor(target,key)

Reflect.getOwnPropertyDescriptor基本等同于  
 Object.getOwnPropertyDescriptor，用于得到指定属性的描述对 象，将来会替代掉后者。

区别就是在于target如果不是对象时的处理。Object不会报错，会返回undefined，Reflect会直接报错。

2.8：Reflect.preventExtensions(target)

Reflect.preventExtensions对应Object.preventExtensions方 法，用于让一个对象变为不可扩展。它返回一个布尔值，表示是 否操作成功。

如果target不是对象，Object在es5环境会报错，在es6环境 会返回target；然而Reflect会直接报错。

2.9：Reflect.isExtensiable(target)

Reflect.isExtensible方法对应Object.isExtensible，返回一 个布尔值，表示当前对象是否可扩展。

如果target不是对象，Object返回false，Reflect会报错。

2.10：Reflect.getPrototypeOf(target)

Reflect.getPrototypeOf方法用于读取对象的\_\_proto\_\_属性， 对应Object.getPrototypeOf(obj)。

区别：如果target不是对象，Object会将target转换成对象之后 在运行；Reflect会直接报错。

2.11：Reflect.setPrototypeOf(target,newProto)

Reflect.setPrototypeOf方法用于设置目标对象的原型   
 （prototype），对应Object.setPrototypeOf(obj, newProto) 方法。它返回一个布尔值，表示是否设置成功。

区别：**如果**target不是对象，Object会返回target；Reflect会 报错。  
 **如果**target是undefined或者是null，Object和Reflect 都会报错。  
 **如果**无法设置目标对象的原型（比如，目标对象禁止扩展）， Reflect.setPrototypeOf方法返回false；Object直 接报错。

2.12：Reflect.apply(func,funcThis,args)  
 Reflect.apply方法等同于   
 Function.prototype.apply.call(func, thisArg, args)，用于 绑定this对象后执行给定函数。

一般来说，如果要绑定一个函数的this对象，可以这样写 **fn.apply(obj, args)**，但是如果函数**定义了自己的apply方法**， 就只能写成**unction.prototype.apply.call(fn,obj,args)**， 采用**Reflect**对象可以简化这种操作。

2.13：Reflect.construct(target, args)  
 Reflect.construct方法等同于new target(...args)，这提供 了一种不使用new，来调用构造函数的方法。  
 **Function G(name) { this.name = name; }  
 var g= new G('张三');// new 的写法  
 var g=Reflect.construct(G, ['张三']);// Reflect的写法**

**ES6之Generator：**

1. Generator 函数是 ES6 提供的一种异步编程解决方案。
2. 从语法上理解，可以理解成它是一个状态机，封装了多个内部状态。
3. 形式上，Generator函数是一个普通函数，但是与普通函数有几点区别。

3.1：function 关键字与函数名之间有一个 \* 符号。Function\* f(){}

3.2：函数内部使用yield表达式，定义不听的内部状态。

3.3：Generator函数执行之后不会执行内部的语句，会返回一个遍历器 对象(指向内部状态的指针对象)；通过该对象去执封装的内部状 态。例如：var g = Generator(); g.next();

1. 写法 function\* func() { ... };
2. 这里涉及到了 yield表达式，以及next()方法。

5.1：yield，可以理解成一个暂停标志。当遇到它的时候，就暂停运行。 并将紧跟在yield之后的表达式的值当做返回对象的value的值。  
   
 如：{value: val,done: Boolean}//Boolean?遍历完毕:还有遍历

注意：yield没有返回值，只有next()函数带了参数了，yield才 有返回值，且第一次的next()函数带参数是没有用的，**因为  
 next()函数带的参数会当做上一个yield的返回值。  
 并且yield只能在Generator函数里面运行，不能再普通函数内 部运行，否则就会报错。**

5.2：next()，可以理解成开始运行，与yield暂停相反；就相当于解锁 yield，当然一次运行，只能解锁一个yield。

5.3：return语句，只执行一次，且将紧跟在return之后的表达式的值 当做返回对象的value的值。不过此是返回对象的done的值就肯 定为true了。也就是说，之后的语句不会再执行。

如果执行到最后一个yield了，没有遇到return语句的话，那么 就返回{value: undefined, done: true}，在这之前，done都为 false。就算只有yield语句也是一样。  
  
 6. For...of 循环，可以直接遍历generator遍历器。且是直接取到 紧跟yield之后的表达式的值，不用next()函数执行。且不会拿紧 跟return之后的表达式的值。

7. Generator.prototype.throw()

7.1：每一个Generator函数返回的遍历器对象，都有一个throw方法， 可以在函数体外抛出错误，然后在Generator函数内部捕获。

7.2：Generator函数内部的每一个catch只能捕获一个遍历器对象抛出 的错误。

注意：在遍历器实例调用throw时，在函数内部找不到catch就会 报错，然后遍历器指针会**直接跳到状态机的最末语句**。此时 的done值已经为true，所以再执行next方法的时候会返回  
 {value: undefined, done: true}。

7.3：throw方法相当于next() + catch()，遇到yield还是会暂停。但 是在遇到yield之前必须遇到catch()，否则报错，指针直接跳到 最后。

7.4：throw方法的参数会被catch接收，建议抛出Error对象的实例。

7.5：该throw方法是Generator函数返回的遍历器对象的方法，并不是 **全局的throw**命令；全局的throw命令抛出的错误只能被函数体外 的catch捕获。

8. Generator.prototype.return()

8.1：Generator 函数返回的遍历器对象，还有一个**return方法**，可以 返回给定的值，并且终结遍历 Generator 函数。

8.2：return的参数会成为value的值，并且done为true。（一般情况）

8.3：如果说Generator函数内部有try...finally语句，那么可以分为 以下几种情况：

8.3.1：未进入try...finally语句内部调用return(sk)；此时就会 终结函数，并返回{value: sk,done: true}。

8.3.2：进入了try调用return(sk)，那么会跳过return之后的所 有语句，直接进入finally内部调用next()方法，此时会暂 时恢复正常状态，等待finally内部的所有语句执行完毕之 后再次next()就会返回{value: sk,done: true}。

8.3.3：进入finally语句里面调用return(sk)，那么会直接终结 Generator函数，并返回{value: sk,done: true}；如果在 try里面也有调用return(sk)，那么finally语句的return 会覆盖try的return。

9. yield\* 表达式

9.1：Generator函数里面调用另外一个Generator函数，也就是把另外 一个状态机封装的内部状态放到这个状态机里面来，普通函数调用 是这样的：

function a() { b() }//普通函数调用  
 function\* g() { k() }//Generator函数调用（达不到目的）  
 function\* g() { yield\* k() }//Generator函数标准调用

9.2：它的作用就是在一个 Generator 函数里面执行另一个 Generator 函数。  
 yield\* 表达式可以说成是遍历了Generator函数的实例，将另外 一个状态机封装的内部状态放到了这个状态机里面。

10. Generator函数作为一个对象的属性写法：

10.1：var obj = {  
 \* g() { ... }  
 }

10.2：var obj = {  
 g: function\* () { ... }  
 }

1. Generator函数的this

11.1：该函数返回的是一个遍历器对象（并非this对象），该对象继承 了Generator函数prototype对象上的方法。

11.2：该函数不能与new 命令一起用，new Generator(); 报错。

11.3：可以用call方法来间接实现让Generator函数成为可以使用new 命令的构造函数。

1. Generator函数与上下文  
    **JavaScript 代码运行时，**会产生一个全局的上下文环境（又称运行环境）， 包含了当前所有的变量和对象。然后，执行函数（或块级代码）的时候，又会在当 前上下文环境的上层，产生一个函数运行的上下文，变成当前（active）的上下文， 由此形成一个上下文环境的堆栈（context stack）。  
    这个堆栈是“后进先出”的数据结构，最后产生的上下文环境首先执行完成， 退出堆栈，然后再执行完成它下层的上下文，直至所有代码执行完成，堆栈清空。  
    **Generator 函数执行产生的上下文环境**，一旦遇到yield命令，就会暂时退 出堆栈，但是并不消失，里面的所有变量和对象会冻结在当前状态。等到对它执行 next命令时，这个上下文环境又会重新加入调用栈，冻结的变量和对象恢复执行。

ES6之Generator函数的异步应用：

1. 异步：

1.1：所谓异步，简单说就是一个任务不是连续完成的。

可以理解成把一个任务分成2部分，先完成一部分，然后执行其 他任务，等该任务的第二部分准备好了就立即执行第二部分的任 务。

所谓同步，就是连续执行。

1. 异步编程的方法：

2.1：回调函数。

就是把任务的第二部分单独卸载一个函数内部，等到重新执行这个任务的时候，就直接调用这个函数。

2.2：事件监听。

2.3：发布/订阅。

2.4：Promise对象。

Promise 的最大问题是代码冗余，原来的任务被 Promise 包装了一下，不管什么操作，一眼看去都是一堆then，原来的语义变得很不清楚。

2.5：以上都是传统的方法，现在有了Generator函数异步编程。

2.6：async函数，是一个Generator函数的语法糖。

1. 一个有趣的问题是，为什么 Node 约定，回调函数的第一个参数，必须 是错误对象err（如果没有错误，该参数就是null）？

原因是执行分成两段，第一段执行完以后，任务所在的上下文环境就已 经结束了。在这以后抛出的错误，原来的上下文环境已经无法捕捉，只 能当作参数，传入第二段。

**ES6之async函数：**

1. 它是一个Generator函数的语法糖。
2. 与Generator函数的比较：

2.1：将 \* 换成了async，yield命令换成了await。

2.2：async函数内置了执行器。也就是说，Generator函数的执行需要 依靠执行器，所以有了co模块。而async函数自带执行器，async 函数的执行就如普通函数那样调用就能得到运行结果。

2.3：更好的语义。Async和await比 \* 号和yield的语义更清楚，async 表示函数里面有异步操作，await表示紧跟在后面的表达式需要等 待结果。

2.4：更广的适用性。co模块约定，yield命令后面只能是 Thunk 函数 或 Promise 对象，而async函数的await命令后面，可以是 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时 等同于同步操作）。

2.5：async函数的返回值是一个promise对象。这比 Generator 函数 的返回值是 Iterator 对象方便多了。你可以用then方法指定下 一步的操作。

1. async函数返回一个 Promise 对象，可以使用then方法添加回调函数。 当函数执行的时候，一旦遇到await就会先返回，等到异步操作完成， 再接着执行函数体内后面的语句。
2. Async 函数内部return语句返回的值，会成为then方法回调函数的参 数。
3. Async函数返回的promise对象，必须等到内部await命令后面的promise 对象执行完，才会发生状态改变，除非遇到return语句或者抛出错误。 也就是说，只有等到所有的异步操作完成，才会执行then方法指定的回 调函数。
4. Await命令后面是一个promise对象。如果不是，会被转换成立即resolve 的promise对象。
5. await命令后面的promise对象如果变为reject状态，那么不管有没有 return语句，该reject的参数会被catch方法的回调函数接收到。
6. 只要有一个await命令后面的promise对象变为reject状态，那么整个 async函数都会中断执行。
7. 普通的 async 函数返回的是一个 Promise 对象，而异步 Generator 函 数返回的是一个异步 Iterator 对象。
8. async 函数和异步 Generator 函数，是封装异步操作的两种方法，都 用来达到同一种目的。区别在于，前者自带执行器，后者通过  
    for await...of执行，或者自己编写执行器。
9. for await...of

11.1：async function f() {

for await (const x of createAsyncIterable(['a', 'b'])) {

console.log(x);

}}

11.2：上面代码中，createAsyncIterable()返回一个拥有异步遍历器 接口的对象，**for...of循环**自动调用这个对象的异步遍历器的 next方法，会得到一个 Promise 对象。await用来处理这个 Promise 对象，一旦resolve，就把得到的值（x）传入for...of 的循环体。

1. 异步Generator函数

12.1：async function\* g() {...};

12.2：var gen = g(); //gen是一个异步Iterator对象。

gen.next() //返回一个promise对象。

如果个g()函数内部抛出错误，会导致gen.next()返回的 promise对象的状态变为reject，然后抛出的错误被catch捕获。

netstat -lnp | grep 8171 查看8171端口 启用情况。

**ES6之Class的继承：**

1. Class的继承时通过extends关键字实现。
2. 在子类的构造函数（constructor）里面，只有调用了super()方法之后 才能使用this关键字。因为子类实例的构建是基于父类的实例的，只有 super()方法才能调用父类实例。
3. 父类的一切都会被子类所继承，就算是静态方法也是一样。
4. super 关键字：

4.1：super 作为函数调用时，代表父类的构造函数；虽然它代表了父类 的构造函数，但是返回的却是子类的实例。(A是父类）

即：super() 相当于 A.prototype.consructor.call(this);

注：super作为函数调用时，只能用于子类的构造函数内部。

4.2：super作为对象调用时，在普通函数中，super指向父类的原型 对象（A.prototype）；在静态函数中，super指向父类( A )。

4.3：ES6规定，在子类的普通函数中通过super调用父类的方法时，方 法内部的this指向当前的子类的实例。

同理，在子类的静态函数中通过super调用父类的静态方法时，方 法（父类的静态方法）内部的this指向当前的子类。

1. 类的prototype属性和\_\_proto\_\_属性：

5.1：子类的\_\_proto\_\_属性，表示构造函数的继承，总是指向父类。

B.\_\_proto\_\_ === A

5.2：子类的prototype属性的\_\_proto\_\_属性，表示方法的继承，总是 指向父类的prototype属性。

B.prototype.\_\_proto\_\_ === A.prototype

5.3：由于子类是继承父类的，所以导致子类原型的原型指向父类的原型。 于是就有：

var a = new A(); // a.\_\_proto\_\_ === A.prototype  
 var b = new V(); // b.\_\_proto\_\_ === B.prototype  
 B.prototype.\_\_proto\_\_ === A.prototype  
 b.\_\_proto\_\_.\_\_proto\_\_ === a.\_\_proto\_\_

1. ES5与ES6生成实例的区别：

6.1：ES5 是先新建子类的实例对象this，再将父类的属性添加到子类 上，由于父类的内部属性无法获取，导致无法继承原生的构造函 数。

6.2：ES6 是先新建父类的实例对象this，然后再用子类的构造函数修 饰this，使得父类的所有行为都可以继承。