Function

ES5函数缺点：

1)arguments是类数组问题，设计时候的缺陷，自己需要手动转换为数组 ES6用res决(ES5 + ES6) []/Array.prototype.slice.call(likeAry)、;Array.from(likeAry) likeAry()函数[自己进行封装]

2)函数的默认参数问题， 默认参数是false六大类的，采用逻辑与(||)运算符会有问题，所以用typeof去检测，这个变量到底传入没有 -->> ES6直接用形参 = 默认值来写了，方便多了

3)函数调用this的问题， 本应该是上级作用域里面的this ，但是在定时器(setTimeout、setInterval)和自执行函数((funciton(){})()、(function(){}()、~function(){}...)里面的this为window对象 ，ES6箭头函数解决(ES6箭头函数没有this，作用域链查找this) --->> ES5变量存储this对象， ES6箭头函数

4)函数表达式函数名的问题， 低版本浏览器获取函数名为空船，高版本浏览器里面是变量名， ES6如果有函数名是函数名，没有则使用是变量

5) XMLHttpRequest 是一个设计粗糙的 API，不符合关注分离（Separation of Concerns）的原则，配置和调用方式非常混乱 ES6中采用promise对象解决， 封装Promise对象，在window对象增加fetch方法[全局方法，内部采用Promise封装的]代替ajax

1.函数参数的默认值

ES5函数默认值的设置

```
//ES5
  let fn = function (x, y) {
  x = x || 10;    //--->>这里有问题， 如果x传入了 那六个false("" NaN false undefined null 0)则都会返回false，取到10
```

// 所以我们ES5中采用 typeof 来检测是否传入参数，去解决这个问题

```
console.log(x);
};
fn(0);
fn(false);
```

```
//ES6
```

1） 函数的参数默认值并不一定是基本类型的值，可以是引用数据类型（数组和对象） `function fn(x, y = getX());`

2） 第一个参数的值，可以默认赋值给第二个参数 `function fn(x, y = x);`

3) 如果参数设置为undefined，则使用默认的参数，因为typeof undefined == undefined,

```
function makeRequest(url, timeout = 2000, callbakc = function() {}) {
        console.log(url, timeout, callbakc);
    }

    makeRequest('./a.json', undefined);  // ==>> ./a.json 2000 ƒ () {}
    makeRequest('./a.json', null);       // ==>> ./a.json null ƒ () {}



function fn(a) {
        if (typeof a !== 'undefined') {
            alert(2);
        }
```

```
        }
fn(undefined);
fn(null);

let fn = function(x = 0, y = false) { // 形参默认值去解决
   console.log(x,y );
};
 //原理:
//   var x = arguments.length > 0 && arguments[0] !== undefined ? arguments[0] : 10;
//   var y = arguments.length > 1 && arguments[1] !== undefined ? arguments[1] : 20;
fn();
fn(1);
fn(false, 0);


 function getValue() {
            return 5;
        }

        function add(first, second = getValue()) {
            console.log(first + second);
        }

 add(1, undefined);
```

注意：**与解构赋值默认值结合使用更加强大**

2.rest 参数(...参数名) 参数名是一个数组，可以使用数组的方法

ES6 引入 rest 参数（形式为...变量名），用于获取函数的多余参数，这样就不需要使用arguments对象了。rest 参数搭配的变量是一个数组，该变量将多余的参数放入数组中。

[函数内置接受形参对象arguments，是一个类数组，设计的缺陷，ES6函数用rest参数(...变量名)<变量是一个数组，可以使用数组的所有方法>来代替他，箭头函数直接把他取消了，箭头函数里面没有arguments对象] 直接用rest参数

注意：

1） 箭头函数里面没有arguments对象

```
let fn = () => console.log(arguments);
```

2） rest 参数之后不能再有其他参数（即只能是最后一个参数），否则会报错

```
let fn = (a, b, ...val, d) => {}
```

3)函数的length属性，不包括 rest 参数。 (length获取实参的个数， arguments获取形参的个数)

```
let fn6 =(a, b, c, d,...val) => 1;
console.log(fn6.length); //4. ...val不包括
```

3.严格模式

4.name 属性：低版本浏览器获取函数名为空串，高版本浏览器里面是变量名， ES6如果有函数名是函数名，没有则使用是变量

5.箭头函数(=>)

规则：

1） 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。let fn =() => {} let fn1 =(x =0, y =0) => {}

2） 如果一个参数直接写就可以，多个用（）包起来

3） 如果一条语句，可以不写{} ，也不用写return,自动返回 let fn6 = (x = 1, y = 2, z = 3) => x + y + z; //多个参数

4） 如果一个参数，可以直接写，不用写小括号 let f = v => v;

5） 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。

箭头函数注意点：

（1）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。< call和apply不好使，无效 > [解决ES5this问题,用变量去保保存当前this]

（2）不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。 没有构造函数constructor,prototype原型属性， 箭头函数里面没有this关键字

//原因： this指向的固定化，并不是因为箭头函数内部有绑定this的机制，实际原因是箭头函数根本没有自己的this，导致内部的this就是外层代码块的this。正是因为它没有 this，所以也就不能用作构造函数。

// ---->>除了this，以下三个变量在箭头函数之中也是不存在的，指向外层函数的对应变量：arguments、super、new.target。 外层没有报错

（3）不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。（...变量名） [解决ES5类数组问题]

（4）不可以使用yield命令，因此箭头函数不能用作 Generator 函数。

prototype， constructor(不能new,里面没有this)， arguments(没有， rest参数)， super(没有类，肯定没有继承)， new.target， call/apply/bind(不可以使用，没有this)

上面四点中，第一点尤其值得注意。this对象的指向是可变的，但是在箭头函数中，它是固定的。

6.绑定 this

this的情况总结：

1)自执行函数调用 this是window

2)普通函数调用(点(.)前面是谁this就是谁，没有的话，默认是window，严格模式是undefined)

3)作为方法来调用

4)作为构造函数来调用(new A() this是当前类的一个实例)

5)定时器里面的this是window(setTimeout, setInterval)

6)使用apply/call方法来调用

7)Function.prototype.bind方法

8)es6箭头函数(没有作用域链查找)

9)给某一个元素绑定事件，事件触发，里面的this为当前元素

10)函数执行返回一个匿名函数，在让这个匿名函数执行，里面的this为window

11) 在模块顶层使用this关键字，是无意义的,this为undefined

```
var bind = function() {


};
```

7.尾调用优化

8.函数参数的尾逗号

面试题：

1. **类数组转为数组**

1) `Array/[].prototype.slice.call(arguments)`

2)
```
let toArray = function(likeAry) {
    let ary = [];
    try {
        ary  Array.prototype.slice.call(likeAry);
    } catch(e) {
        for (let i=0,len=likeAry.length;i<len;i++) {
            ary[i] = likeAry[i];
        }
    }

 return ary;
```

```
};
```

3) Array.from(arrayLike)

4) 函数参数使用rest参数(...val/ [...arguments]);

2.数组去重

1):

```
Array.prototype.unique = function () {
    let obj = {};
    for (let i=0;i<this.length;i++) {
        let cur = this[i];
      if (obj[cur] === cur) {
          this.splice(i, 1);
          i--;
          continue;
      }
      obj[cur] = cur;
    }
    obj = null;
    return this;
};
```

2):

```
new Set(likeAry); // ===>> set数据类型
```

3):

```
Array.prototype.unique2 = function(){
 this.sort(); //先排序
 var res = [this[0]];
 for(var i = 1; i < this.length; i++){
  if(this[i] !== res[res.length - 1]){
   res.push(this[i]);
```

```
        }
      }
      return res;
    }
```

2.this考察

```
A1);
   var name = '我是全局的name';
   let obj3 = {
   name: 'clh',
   getName1: function () {
       window.setTimeout(() =>
   console.log(this.name),1000); //-->> 'clh'  this-->>
   obj3 上级作用域里面的this
   },
   getName2: () => {
       window.setTimeout(() =>
   console.log(this.name),2000); //-->>  '我是全局的name';
   this-->> window 当前作用域里面的this
   },
   getName3(){
       window.setTimeout(() =>
   console.log(this.name),3000); //-->>  //-->> 'clh'
   this-->> obj3 上级作用域里面的this
   },

   getName4(){
       window.setTimeout(() =>
   console.log(this.name),4000); //-->>  //-->> 'clh'
   this-->> obj4 上级作用域里面的this
```

```
    },
    getName5() {
        return () => {
            console.log(this.name);   //-->> 'clh'   this-->> obj4 上级作用域里面的this
        }
    }
    };
    obj3.getName1();
    obj3.getName2();
    obj3.getName3();


    obj3.getName4();
    obj3.getName5.call(window)();
A2);
    let obj1 = {
            name: true,
            getName: () => {
                console.log(this); //-->> //-->>this上级作用域里面this  <<作用域1>>
                return () => {
                  console.log(this);  //-->>this上级作用域里面this  <<作用域0>>
                };
            }
        };
    obj1.getName()();


A3);
```

```
var name = 10;
let obj = {
   name: null,
   getName: function () {
       console.log(this);
      /*return function () {
          console.log(this);

      };*/

   return () => console.log(this);
   }
};
```

3) 箭头函数：

arrowFunction1);

```
let name = '常连海';
let fn3 = {
     name: '王东',
     getName: function () {
         return function () {
             console.log(this.name);
         };
     }
   };
```

fn3.getName()(); //空值， 因为我是用let声明的变量，没有绑定到window对象上讲name变量

arrowFunction2)

```
;~function() {
    console.log(this); // ===>> 自执行函数里面的this为window，没有执行的主体，严格模式下为undefined
}();

;(() => {
 console.log(this); //====>> 箭头函数里面没有this， 作用域链向上级查找找到全局this为window
})();
```

arrowFunction3)

```
let obj1 = {name: 'clh', age: 23};
let fn5 = function () {
    console.log(this);  //fn5函数里面就有一个this，下面的都是查找到这个this
    return () => {
        console.log(this);
        return () => {
            console.log(this);
            return () => {
                console.log(this);
            }
        }
    }
};
fn5.call(obj1)()()();; //空值， 因为我是用let声明的变量，没有绑定到window对象上讲name变量
```

arrowFunction4)

```
let name = 'wd';
```

```
let obj = {
 name: 'clh',
 getName1() {
   window.setTimeout(() => {
      console.log(this.name);
   });
 },

getName2() {
   window.setTimeout(function() {
      console.log(this.name);
   });
 }
};

obj.getName1(); // ===>> 'clh'
obj.getName2(); // ===>> 空值
```

arrowFunction5)

```
let name = '王东';
let obj = {
name: 'clh',
getName: function () {
    console.log(this.name);
}
};
let fn9 = () => {
console.log(arguments);
console.log(this);
};
```

```
fn9(obj); // ===>> arguments is not defined
fn9.call(obj); //===>> window对象(作用域链)
```

4) rest参数

//类数组改为数组 和箭头函数对象写法

```
// ==>> ES5
let fn4_1 = function () {
    return [].slice.call(arguments).sort(function
(a, b) {
        return b - a;
    });
};

// ==>> ES6
let fn4_2 = (...val) => val.sort((a, b) => b - a);
```

案例:

```
let fn1 = function (...values) {
    console.log(values.sort());
    console.log(arguments instanceof Array);
//false
    console.log(values instanceof Array); //true
};
fn1(1, 3, 4, 8, 0);
```

5) 默认参数

```
let fn = function(x ,y) {
   x = x || 0;
};
fn(0, false);
let fn1 = function(x, y) {
```

```
    if (typeof x === 'undefined') {
        x = 0;
    }
};
fn1(0, false);

let fn2 = function(x =0, y =0) {

};
fn2(0, false);
```

ES6 的函数考虑了 JS 开发者多年的抱怨和要求，向前大步迈进，于是便在 ES5 函数之上实现了不少增量改进，让 JS 的编程错误更少并且更加强大。

- 带参数默认值的函数
  - 在 ES5 中模拟参数默认值
  - ES6 中的参数默认值
  - 参数默认值如何影响 arguments 对象
  - 参数默认值表达式
  - 参数默认值的暂时性死区
- 使用不具名参数
  - ES5 中的不具名参数
  - 剩余参数
    - 剩余参数的限制条件
    - 剩余参数如何影响 arguments 对象
- 函数构造器的增强能力
- 扩展运算符
- ES6 的名称属性
  - 选择合适的名称
  - 名称属性的特殊情况
- 明确函数的双重用途
  - 在 ES5 中判断函数如何被调用
  - new.target 元属性
- 块级函数
  - 决定何时使用块级函数
  - 非严格模式的块级函数
- 箭头函数
  - 箭头函数语法
  - 创建立即调用函数表达式
  - 没有 this 绑定
  - 箭头函数与数组
  - 没有 arguments 绑定
  - 识别箭头函数
- 尾调用优化