第1章：ECMASript 相关介绍

1.1. 什么是ECMA

ECMA（European Computer Manufacturers Association）中文名为欧洲计算机制造商协会，这个组织的目标是评估、开发和认可电信和计算机标准。1994年后该组织改名为Ecma国际。

1.2. 什么是ECMAScript

ECMAScript 是由Ecma国际通过ECMA-262标准化的脚本程序设计语言。

1.3. 什么是ECMA-262

Ecma国际制定了许多标准，而ECMA-262只其中的一个，所有标准列表查看

http://www.ecma-international.org/publications/standards/Standard.htm

1.4. ECMA-262历史

ECMA-262（ECMAScript）历史版本查看网址

http://www.ecma-international.org/publications/standards/Ecma-262-arch.htm

第1版	**1997年**	**制定了语言的基本语法**
第2版	1998年	较小改动
第3版	1999年	引入正则、异常处理、格式化输出等。IE开始支持
第4版	2007年	过于激进，未发布
第5版	2009年	引入严格模式、JSON，扩展对象、数组、原型、字符串、日期方法
第6版	**2015年**	模块化、面向对象语法、Promise、箭头函数、let、const、数组解构赋值等等
第7版	2016	幂运算符、数组扩展、Async/await关键字
第8版	2017年	Async/await、字符串扩展
第9版	2018年	对象解构赋值、正则扩展
第10版	2019年	扩展对象、数组方法
ES.next	动态指向下一个版本	

注：从ES6开始，每年发布一个版本，版本号比年份最后一位大1

1.5. 谁在维护ECMA-262

TC39（Technical Committee 39）是推进 ECMAScript 发展的委员会。其会员都是公司（其中主要是浏览器厂商，有苹果、谷歌、微软、因特尔等）。TC39 定期召开会议，会议由会员公司的代表与特邀专家出席

1.6. 为什么要学习ES6

l ES6的版本变动内容最多，具有里程碑意义

l ES6加入许多新的语法特性，编程实现更简单、高效

l ES6是前端发展趋势，就业必备技能

1.7. ES6兼容性

http://kangax.github.io/compat-table/es6/ 可查看兼容性

2. 第2章： ECMASript6新特性

2.1. let关键字

let关键字用来声明变量，使用let声明的变量有几个特点：

1) 不允许重复声明

```
var a = 1;
    var a = 2
    console.log(a);
    //1  let 不能重复声明
    let b = 1
    console.log(b);
    //let b = 2
    //console.log(b);
```

2) 块儿级作用域

```
if(true){
    let a =10;
}
console.log(a)//a is not defined
```

3) 不存在变量提升

```
console.log(a)//a is not defined
let a=10;
```

4) 防止循环变量变成全局变量

```
for (var i = 0; i < 4; i++) {
        console.log(i); //0 1 2 3
    }
    console.log(i); //4   i 是全局变量

    for (let j = 0; j < 4; j++) {
        console.log(j); //0 1 2 3
    }
    console.log(j);//j is not defined
```

5）暂时性死区

```
var tmp = 123;
if(true){
    console.log(tmp)
    let tmp
}
```

```
var arr = []
for(var i = 0;i<2;i++){
    arr[i]=function (){
        console.log(i)
    }
}
arr[0]();//2
arr[1]();//2
```

```
let arr = []
for(let i = 0;i<2;i++){
    arr[i]=function (){
        console.log(i)
    }
}
arr[0]();
arr[1]();
```

应用场景：以后声明变量使用let就对了

2.2. const关键字

const 关键字用来声明常量，（内存地址不更改的变量）const声明有以下特点

1) 声明必须要赋初始值

```
const PI
console.log(PI)//Missing initializer in const declaration
```

3) 不允许重复声明

```
const PI  =3.1415
const PI  =3.141589//Identifier 'PI' has already been declared
```

4) 不允许修改(内存地址)

```
const a = 100
a = 200 //报错
const arr = [1,2]
arr[0] = 3 // 没有更改内存地址
console.log(arr) //[3,2]
arr = [3,4]//报错
```

5) 块儿级作用域

```
if(true){
    const a =10;
    if(true){
        const a =20;
        console.log(a)
    }
    console.log(a)
}
console.log(a)
```

注意：对象属性修改和数组元素变化不会出发const错误

应用场景：声明对象类型使用const，非对象类型声明选择let

let、const、var 的区别

1. 使用 var 声明的变量，其作用域为该语句所在的函数内，且存在变量提升现象。
2. 使用 let 声明的变量，其作用域为该语句所在的代码块内，不存在变量提升。
3. 使用 const 声明的是常量，在后面出现的代码中不能再修改该常量的值。

var	let	const
函数级作用域	块级作用域	块级作用域
变量提升	不存在变量提升	不存在变量提升
值可更改	值可更改	值不可更改

2.3. 变量的解构

ES6允许按照一定模式从数组和对象中提取值，对变量进行赋值，这被称为解构赋值。

```
//数组的解构赋值
const arr = ['张学友', '刘德华', '黎明', '郭富城'];
let [zhang, liu, li, guo] = arr;
```

```
可嵌套
let [a, [[b], c]] = [1, [[2], 3]];
// a = 1
// b = 2
// c = 3

//对象的解构赋值 默认  let {} 变量名必须和 obj里的key 保持一致
    const obj = {
        name: 'zhangsan',
        age: 18
    }
    let {
        name,
        age
    } = obj
    console.log(name, age);

    //别名赋值
    let {
        name: myname,
        age: myage
    } = obj
    console.log(myname, myage);

//可忽略
let [a, , b] = [1, 2, 3];
// a = 1
// b = 3

剩余参数运算符
let [a, ...b] = [1, 2, 3];
//a = 1
//b = [2, 3]

//字符串等
let [a, b, c, d, e] = 'hello';
// a = 'h'
// b = 'e'
// c = 'l'
// d = 'l'
// e = 'o'
//解构默认值
let [a = 2] = [undefined]; // a = 2
当解构模式有匹配结果，且匹配结果是 undefined 时，会触发默认值作为返回结果。
let [a = 3, b = a] = [];     // a = 3, b = 3
let [a = 3, b = a] = [1];    // a = 1, b = 1
let [a = 3, b = a] = [1, 2]; // a = 1, b = 2


//对象的解构赋值
const lin = {
    name: '林志颖',
    tags: ['车手', '歌手', '小旋风', '演员']
};
let {name, tags} = lin;

//复杂解构
let wangfei = {
```

```
    name: '王菲',
    age: 18,
    songs: ['红豆', '流年', '暧昧', '传奇'],
    history: [
        {name: '窦唯'},
        {name: '李亚鹏'},
        {name: '谢霆锋'}
    ]
};
let {songs: [one, two, three], history: [first, second, third]} = wangfei;
```

注意：频繁使用对象方法、数组元素，就可以使用解构赋值形式

剩余参数

```
const fn = (...args)=>{
    console.log(args)
}
fn(1,2,3)
```

2.4. 模板字符串

模板字符串（template string）是增强版的字符串，用反引号（`）标识，特点：

1) 字符串中可以出现换行符

2) 可以使用 ${xxx} 形式输出变量

3) 可以调用函数

```
// 定义字符串
let str = `<ul>
        <li>沈腾</li>
        <li>玛丽</li>
        <li>魏翔</li>
        <li>艾伦</li>
    </ul>`;
function uppername(name){
    return name.toUpperCase()
}
// 变量拼接
let star = '王宁';
let result = `${star}在前几年离开了开心麻花 ${uppername('hello')}`;
```

注意：当遇到字符串与变量拼接的情况使用模板字符串

2.5. 简化对象写法

ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

```
//以前写法：
```

```
let obj = {
    name:'nihao',
    slogon:'永远追求行业更高标准',
    improve:function(){
    },
    change:function() {
        console.log('可以改变你')
    }
};

//es6简化写法
let name = 'nihao';
let slogon = '永远追求行业更高标准';
let improve = function () {
    console.log('可以提高你的技能');
}
//属性和方法简写
let atguigu = {
    name,
    slogon,
    improve,
    change() {
        console.log('可以改变你')
    }
};
```

注意：对象简写形式简化了代码，所以以后用简写就对了

2.6. 箭头函数

ES6 允许使用「箭头」（=>）定义函数。

```
 //非箭头函数写法
 let fn =function (arg1, arg2, arg3)  {
        return arg1 + arg2 + arg3;
 }
//箭头函数写法 ，把function去掉，在函数（）后面加 =>
 let fn =(arg1, arg2, arg3) => {
        return arg1 + arg2 + arg3;
 }
```

箭头函数的注意点:

1) 如果形参只有一个，则小括号可以省略

2) 函数体如果只有一条语句，则花括号可以省略，函数的返回值为该条语句的执行结果 可以省略 return

3) 箭头函数this指向 声明时 所在作用域下 this 的值

4) 箭头函数不能作为构造函数实例化 不能new

5) 箭头函数不能使用 arguments

```
//2. 省略小括号的情况
      let fn2 = num => {
        return num * 10;
      };

//3. 省略花括号的情况*/
      let fn3 = score => score * 20;

//4. this指向声明时所在作用域中 this 的值 this
      let fn4 = () => {
        console.log(this); //window
      }

let school = {
    name: '张三',
    getName(){
        let fn5 = () => {
            console.log(this);
        }
        fn5();
    }
};
```

注意：箭头函数不会更改this指向，所以非常适合设置与this无关的回调，比如数组回调、定时器回调，不适合事件回调与对象方法。

2.7. rest参数

ES6引入rest参数，用于 获取 函数的实参，用来代替arguments，分割序列=>数组

1、用于获取函数的实参,替代argments

2、用法： ...参数名

3、获取剩余的实参

4、rest参数必须放到最后

```
//作用与 arguments 类似
   function add(...args){
   console.log(args);
   }
   add(1,2,3,4,5);

// rest 参数必须是最后一个形参
   function minus(a,b,...args){
      console.log(a,b,args);
   }
   minus(100,1,2,3,4,5,19);
```

注意：rest参数非常适合不定个数参数函数的场景

2.8. 扩展运算符

扩展运算符（spread）也是三个点（...）。它好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列，对数组进行解包。

```
let arr = [1,2,3]
...arr //1,2,3
console.log(...arr)// 1 2 3
```

```
//展开数组

   let tfboys = ['德玛西亚之力','德玛西亚之翼','德玛西亚皇子'];
   function fn(){
   console.log(arguments);
   }
   fn(...tfboys)
  // fn('德玛西亚之力','德玛西亚之翼','德玛西亚皇子')

//展开对象

   let skillOne = {
    q: '致命打击',
   };
   let skillTwo = {
    w: '勇气'
   };
   let skillThree = {
    e: '审判'
   };
   let skillFour = {
    r: '德玛西亚正义'
   };

let gailun = {...skillOne, ...skillTwo,...skillThree,...skillFour};
```

扩展运算符的应用

1. 合并数组

```
// ES5
[1, 2].concat(more)
// ES6
[1, 2, ...more]
var arr1 = ['a', 'b'];
var arr2 = ['c'];
var arr3 = ['d', 'e'];
// ES5 的合并数组
arr1.concat(arr2, arr3);
// [ 'a', 'b', 'c', 'd', 'e' ]
// ES6 的合并数组
[...arr1, ...arr2, ...arr3]
// [ 'a', 'b', 'c', 'd', 'e' ]
```

2. 与解构赋值结合

   如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。

```
// ES5
let list = [1,2,3]
a = list[0], rest = list.slice(1)//a = 1 rest = [2,3]
// ES6
[a, ...rest] = list //a = 1 rest = [2,3]
//下面是另外一些例子。
const [first, ...rest] = [1, 2, 3, 4, 5];
first // 1
rest // [2, 3, 4, 5]
const [first, ...rest] = [];
first // undefined
rest // []:
const [first, ...rest] = ["foo"];
first // "foo"
rest // []
```

3. 字符串

   扩展运算符还可以将字符串转为真正的数组。

```
[...'hello']
// [ "h", "e", "l", "l", "o" ]
```

4. 实现了 Iterator 接口的对象

   任何 Iterator 接口的对象，都可以用扩展运算符转为真正的数组

```
//伪数组转换为数组
var nodeList = document.querySelectorAll('div');
var array = [...nodeList];
```

扩展运算符和rest运算符是逆运算

扩展运算符：数组=>分割序列

rest运算符：分割序列=>数组

Array的内置扩展方法

1 Array.from() 将类数组或者可遍历对象转换为真正的数组

```
let arraylike = {
    '0':'1',
    '1':'2',
    length:3
}
let array = Array.from(arraylike);
```

2 Array.includes() 数组内是否包含某个值 返回布尔值

```
let array = [1,3,5,7]
let res = array.includes(3);
```

3 Array.find() 查找第一个符合条件的数组成员，如果没有找到返回undefined

```
let arr = [{
    id:1,
    name:'zhangsan'
},{
    id:2,
    name:'lisi'
}]
let res= arr.find((item,index)=>item.id==2);
```

4 Array.findIndex()查找第一个符合条件的数组成员的索引位置，没有则返回-1

```
let array = [1,3,5,7]
let res = array.findIndex((item,index)=>item>3)
```

5 Array.some() 针对数组中的每一个元素，但是这个方法是，只要有一个元素比对结果为true，返回结果就为true，反之要所有的元素比对结果为false才为false

```
const arr = [10,20.50,60,70,80]
const res = arr.some(item => item <0)
console.log(res) //false
const res = arr.some(item => item >20)
console.log(res) //true
```

6 array.every()针对数组中的每一个元素进行比对，只要有一个元素比对结果为false则返回false，反之要所有的元素比对结果为true才为true

```
every()
```

every() 方法使用指定函数检测数组中的所有元素：

如果数组中检测到有一个元素不满足，则整个表达式返回 false ，且剩余的元素不会再进行检测。
如果所有元素都满足条件，则返回 true。

7 Array.filter()检测数值元素，并返回符合条件所有元素的数组。

8 array.map() 方法返回一个新数组，数组中的元素为原始数组元素调用函数处理后的值。
map() 方法按照原始数组元素顺序依次处理元素。

注意：(1)map()方法不会对空数组进行检测 (2)map()方法不会改变原始数组

```
const arr = [88,90,100,20,50]
const res = arr.map(item => item * 0.8)
console.log(res);//[70.4, 72, 80, 16, 40]
```

9 array.forEach()

2.9. Symbol

2.9.1. Symbol基本使用

ES6 引入了一种新的原始数据类型Symbol，表示独一无二的值。它是 JavaScript 语言的第七种数据类型，是一种类似于字符串的数据类型。

Symbol特点

1) Symbol的值是唯一的，用来解决命名冲突的问题

2) Symbol值不能与其他数据进行运算

3) Symbol定义的对象属性不能使用for...in循环遍历，但是可以使用Reflect.ownKeys来获取对象的所有键名

```
//创建 Symbol
let s1 = Symbol();
console.log(s1, typeof s1);

//添加标识的 Symbol
let s2 = Symbol('张三');
let s2_2 = Symbol('张三');
console.log(s2 === s2_2);//false

//使用 Symbol for 定义
let s3 = Symbol.for('张三');
let s3_2 = Symbol.for('张三');
console.log(s3 === s3_2);//true  内存地址一样
```

使用场景
1、作为属性名
用法
由于每一个 Symbol 的值都是不相等的，所以 Symbol 作为对象的属性名，可以保证属性不重名。

```
let sy = Symbol("key1");

// 写法1
let syObject = {};
syObject[sy] = "kk";
console.log(syObject);    // {Symbol(key1): "kk"}

// 写法2
let syObject = {
[sy]: "kk"
};
console.log(syObject);    // {Symbol(key1): "kk"}

// 写法3
let syObject = {};
Object.defineProperty(syObject, sy, {value: "kk"});
console.log(syObject);   // {Symbol(key1): "kk"}
```

Symbol 作为对象属性名时不能用.运算符，要用方括号。因为.运算符后面是字符串，所以取到的是字符串 sy 属性，而不是 Symbol 值 sy 属性。

```
let syObject = {};

 syObject[sy] = "kk";

syObject[sy];  // "kk"

 syObject.sy;   // undefined
```

注:* *Symbol类型唯一合理的用法是用变量存储 symbol的值，然后使用存储的值创建对象属性

Symbol.for()
Symbol.for() 类似单例模式，首先会在全局搜索被登记的 Symbol 中是否有该字符串参数作为名称的 Symbol 值，如果有即返回该 Symbol 值，若没有则新建并返回一个以该字符串参数为名称的 Symbol 值，并登记在全局环境中供搜索。

//Symbol.for('张三') 有 则返回 没有则创建

```
let yellow = Symbol("Yellow");
let yellow1 = Symbol.for("Yellow");
yellow === yellow1;      // false

let yellow2 = Symbol.for("Yellow");
yellow1 === yellow2;     // true
```

Symbol.keyFor()

Symbol.keyFor() 返回一个已登记的 Symbol 类型值的 key ，用来检测该字符串参数作为名称的 Symbol 值是否已被登记。

```
let yellow1 = Symbol.for("Yellow");
Symbol.keyFor(yellow1);    // "Yellow"
```

2.9.2. Symbol内置值

除了定义自己使用的 Symbol 值以外，ES6 还提供了11个内置的Symbol值，指向语言内部使用的方法。

Symbol.hasInstance	**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**
Symbol.isConcatSpreadable	对象的Symbol.isConcatSpreadable属性等于的是一个布尔值，表示该对象用于Array.prototype.concat()时，是否可以展开。
Symbol.species	创建衍生对象时，会使用该属性
Symbol.match	当执行str.match(myObject) 时，如果该属性存在，会调用它，返回该方法的返回值。
Symbol.replace	当该对象被str.replace(myObject)方法调用时，会返回该方法的返回值。
Symbol.search	当该对象被str. search (myObject)方法调用时，会返回该方法的返回值

	的返回值。
Symbol.hasInstance Symbol.split	**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** 当该对象被str.split(myObject)方法调用时，会返回该方法的返回值。
Symbol.iterator	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
Symbol.toPrimitive	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。
Symbol. toStringTag	在该对象上面调用toString方法时，返回该方法的返回值
Symbol. unscopables	该对象指定了使用with关键字时，哪些属性会被with环境排除。

2.10. 迭代器

迭代器（Iterator）是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作。

1) ES6创造了一种新的遍历命令for...of循环，Iterator接口主要供for...of消费

2) 原生数据类型 具备iterator接口的数据(可用for of遍历)

a) Array

b) Arguments

c) Set

d) Map

e) String

f) TypedArray //类型数组

g) NodeList //是DOM操作取出的集合（实际上是基于DOM结构动态查询的结果）

```
//遍历数组
let arr = ['张三','李四','王五'];
for(let v of arr){
  console.log(v);
}

//遍历arguments
        function fn(){

            for(let a of arguments){
                console.log(a);
            }

        }
  fn(1,2,3);
  //遍历字符串
  let str="八月十五中秋节";
   for(let v of str){
       console.log(v);
  }

  //遍历nodeList
```

```
   const divs=  document.querySelectorAll("div");
   for(let d of divs){
    console.log(d.innerHTML);
  }
```

3) 工作原理

a) 创建一个指针对象，指向当前数据结构的起始位置

b) 第一次调用对象的next方法，指针自动指向数据结构的第一个成员

c) 接下来不断调用next方法，指针一直往后移动，直到指向最后一个成员

d) 每调用next方法返回一个包含value和done属性的对象

```
let iter=arr[Symbol.iterator]();
 console.log(iter.next());
 console.log(iter.next());
```

注: 需要自定义遍历数据的时候，要想到迭代器。

2.11. Promise

处理异步有几种方式?
1、回调函数
2、promise
3、async await

Promise是ES6引入的异步编程的新解决方案。语法上Promise是一个构造函数，用来封装异步操作并可以获取其成功或失败的结果。

Promise 的含义
Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。它由社区最早提出和实现，ES6 将其写进了语言标准，统一了用法，原生提供了Promise对象。

Promise 异步操作有三种状态：pending（进行中）、fulfilled（已成功）和 rejected（已失败）
两种结果：成功 resolve、失败reject

回调地狱

在实际的使用中，有非常多的应用场景我们不能立即知道应该如何继续往下执行。最常见的一个场景就是ajax请求。通俗来说，由于网速的不同，可能你得到返回值的时间也是不同的，这个时候我们就需要等待，结果出来了之后才知道怎么样继续下去。在ajax的原生实现中，利用了onreadystatechange事件，当该事件触发并且符合一定条件时，才能拿到想要的数据，之后才能开始处理数据。

这样做看上去并没有什么麻烦，但如果这个时候，我们还需要另外一个ajax请求，这个新ajax请求的其中一个参数，得从上一个ajax请求中获取，这个时候我们就不得不等待上一个接口请求完成之后，再请求后一个接口。如下:

```
var url = 'http://api.2106.com/data?name=刘博金';
var result;

var XHR = new XMLHttpRequest();
XHR.open('GET', url, true);
XHR.send();

XHR.onreadystatechange = function() {
    if (XHR.readyState == 4 && XHR.status == 200) {
```

```
        result = XHR.response;
        console.log(result);

        // 伪代码
        var url2 = 'http://api.2106.com/list?page=' + result.someParams;
        var XHR2 = new XMLHttpRequest();
        XHR2.open('GET', url, true);
        XHR2.send();
        XHR2.onreadystatechange = function() {
            ...
        }
    }
}


//jquery ajax
var url = 'http://api.2106.com/data';
var url_1 = 'http://api.2106.com/data1';
$.ajax({
    'type':'get',
    'url':url
    'data':'',
    sucess:function(res){
        if(res.code=='00000'){
            $.ajax({
                'type':'get',
                'url':url_1
                'data':'id='+res.id,
                sucess:function(res){
                    if(res.code=='00000'){

                    }
                }
            })
        }
    }
})
```

当出现第三个ajax(甚至更多)仍然依赖上一个请求时，我们的代码就变成了一场灾难。这场灾难，往往也被称为**回调地狱。**

因此我们需要一个叫做Promise的东西，来解决这个问题。

当然，除了回调地狱之外，还有一个非常重要的需求：**为了代码更加具有可读性和可维护性，我们需要将数据请求与数据处理明确的区分开来。**上面的写法，是完全没有区分开，当数据变得复杂时，也许我们自己都无法轻松维护自己的代码了。这也是模块化过程中，必须要掌握的一个重要技能，请一定重视。

基本用法
ES6 规定，Promise对象是一个构造函数，用来生成Promise实例。

下面代码创造了一个Promise实例。

```
//promise用法
   const p= new Promise((resolve,reject)=>{
    if(true){resolve()} //返回成功
    if(false) {reject()}//返回失败
   });
  p.then(res=>{
      console.log(res);
      console.log("成功");
  },fail=>{
      console.log(fail);
      console.log("失败");
  });
```

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。

resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”（即从 pending 变为 resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”（即从 pending 变为 rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

Promise实例生成以后，可以用then方法分别指定resolved状态和rejected状态的回调函数。

```
promise.then(function(value) {
  // success
}, function(error) {
  // failure
});
```

then方法可以接受两个回调函数作为参数。第一个回调函数是Promise对象的状态变为resolved时调用，第二个回调函数是Promise对象的状态变为rejected时调用。这两个函数都是可选的，不一定要提供。它们都接受Promise对象传出的值作为参数。

Promise.then()
Promise 实例具有then方法，有两个回调函数，第一个参数是resolved状态的回调函数，第二个参数是rejected状态的回调函数，它们都是可选的。

then方法返回的是一个新的Promise实例（注意，不是原来那个Promise实例）。因此可以采用链式写法，即then方法后面再调用另一个then方法。

```
promise.then(function(json) {
  return json.post;
}).then(function(post) {
  // ...
});
```

Promise.catch()
Promise.prototype.catch()方法是.then(null, rejection)或.then(undefined, rejection)的别名，用于指定发生错误时的回调函数。

```
promise.then(function(posts) {
  // ...
}).catch(function(error) {
  // 处理 getJSON 和 前一个回调函数运行时发生的错误
  console.log('发生错误! ', error);
});
```

Promise.all()方法

all()方法执行多个Promise对象，只有全部成功，才执行then里的resolve回调函数，否则执行reject回调函数

```
const p1=new Promise((resolve,reject)=>{

    setTimeout(() => {
        resolve(100);
    }, 1000);

});
const p2=new Promise((resolve,reject)=>{

    setTimeout(() => {
        reject(200);
    }, 2000);

});
const p3=new Promise((resolve,reject)=>{

    setTimeout(() => {
        reject(404);
    }, 500);

});

Promise.all([p1,p2,p3]).then(res=>{

    console.log(res);
    console.log("成功");

},fail=>{
    console.log(fail);//404
    console.log("失败");

});
```

Promise.race()方法

race()方法执行多个Promise对象， Promise.race([p1, p2, p3])里面哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。

```
const p1=new Promise((resolve,reject)=>{

    setTimeout(() => {
```

```
        resolve(100);
    }, 1000);

});
const p2=new Promise((resolve,reject)=>{

    setTimeout(() => {
        reject(200);
    }, 2000);

});
const p3=new Promise((resolve,reject)=>{

    setTimeout(() => {
        reject(404);
    }, 500);

});

Promise.race([p1,p2,p3]).then(res=>{

    console.log("成功");
    console.log(res);

},fail=>{

    console.log("失败");
    console.log(fail);

});
```

对最开始的ajax的例子进行一个简单的封装。看看会是什么样子。

```
var url = 'http://api.2106.com/data';

// 封装一个get请求的方法
function getJSON(url) {
    return new Promise(function(resolve, reject) {
        var XHR = new XMLHttpRequest();
        XHR.open('GET', url, true);
        XHR.send();

        XHR.onreadystatechange = function() {
            if (XHR.readyState == 4) {
                if (XHR.status == 200) {
                    try {
                        var response = JSON.parse(XHR.responseText);
                        resolve(response);
                    } catch (e) {
                        reject(e);
                    }
                } else {
                    reject(new Error(XHR.statusText));
                }
            }
        }
    })
```

```
}

getJSON(url).then(resp => console.log(resp));
//Promise.all的用法
var url = 'http://api.2106.com/data1';
var url1 = 'http://api.2106.com/data2';

function renderAll() {
    return Promise.all([getJSON(url), getJSON(url1)]);
}

renderAll().then(function(value) {
    // 建议大家在浏览器中看看这里的value值
    console.log(value);
})

//Promise.race的用法
function renderRace() {
    return Promise.race([getJSON(url), getJSON(url1)]);
}

renderRace().then(function(value) {
    console.log(value);
})
```

2.12. async 函数

Async/await 是Javascript编写异步程序的新方法。

语法

```
function test () {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(200);
    }, 2000);
  });
}
 //await 等待
async function hello() {
  var x = await test;
  return x;

}
hello ();
//hello ().then(v=>{
   //console.log(v);
//})
```

返回值

async 函数返回一个 Promise 对象，可以使用 then 方法添加回调函数。

await后面针对所跟不同表达式的处理方式：

Promise 对象：await 会暂停执行，等待 Promise 对象 resolve，然后恢复 async 函数的执行并返回解析值。
非 Promise 对象：直接返回对应的值。

```
function test(){
  return "test"
}
async function hello(){
  let rows= await test();
  console.log(rows);
}
```

```
function test(){
  return new Promise((resolve,reject) => {
    setTimeout(function(){
    resolve(200);
    }, 1000);
  });
}
async function hello(){
 let rows= await test();
  console.log(rows);
}
```

2.13. Set

ES6 提供了新的数据结构 Set（集合）。它类似于数组，『但成员的值都是唯一』的,实现方式类似构造函数需要搭配new 关键字使用，集合实现了iterator接口，所以可以使用『扩展运算符』和『for...of...』 foreach 进行遍历，集合的属性和方法：

1) size 返回集合的元素个数

2) add　增加一个新元素，返回当前集合

3) delete　删除元素，返回boolean 值

4) has　检测集合中是否包含某个元素，返回boolean值

5) clear　清空集合，返回undefined

```
//创建一个空集合
let s = new Set();
//创建一个非空集合
let s1 = new Set([1,2,3,1,2,3]);

//集合属性与方法
//返回集合的元素个数
console.log(s1.size);
//添加新元素
console.log(s1.add(4));
//删除元素
```

```
console.log(s1.delete(1));
//检测是否存在某个值
console.log(s1.has(2));
//清空集合
console.log(s1.clear());

//实现数组去重（应用场景 如关键词搜索）
const s3 = new Set(['a','b','a','b']);
console.log(s3.size);
//利用扩展运算符实现转换为数组
const arr = [...s3]
console.log(arr);
```

2.14. Map

ES6 提供了 Map 数据结构。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。Map也实现了iterator接口，所以可以使用『扩展运算符』和『for...of...』进行遍历。Map的属性和方法：

1) size 返回Map的元素个数

2) set　　增加一个新元素，返回当前Map

3) get　　返回键名对象的键值

4) delete　删除某个键

5) has　　检测Map中是否包含某个元素，返回boolean值

6) clear　清空集合，返回undefined

```
//创建一个空 map
let m = new Map();
//创建一个非空 map
let m2 = new Map([
    ['name','张三'],
    ['slogon','不断提高行业标准']
]);

//属性和方法
//获取映射元素的个数
console.log(m2.size);
//添加映射值
console.log(m2.set('age', 6));
//获取映射值
console.log(m2.get('age'));
//检测是否有该映射
console.log(m2.has('age'));
//清除
console.log(m2.clear());

  const stus=new Map([
            ["张三",20],
            ["李四",25],
            ["王五",30]
```

```
        ]);
        //遍历map键
        for (const [key] of stus) {
            console.log(key);
        }
        //遍历键和值
        for (const [key,value] of stus) {
            console.log(key+value);
        }
```

2.15. class类

面向对象的核心是对象，对象可以是任意一个事物，编程中使用类来描述现实世界中的元素，类是抽象的。

类的结构：
    属性： 描述对象的特征
    方法： 描述对象的行为功能

ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过class关键字，可以定义类。基本上，ES6 的class可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。

知识点：

1. class声明类
2. constructor定义构造函数初始化
3. extends继承父类
4. super调用父级构造方法
5. static定义静态方法和属性
6. 父类方法可以重写
7. Get和set方法

类的声明

```
class Phone {
    //构造方法
    constructor() {

    }

}
```

constructor 方法
constructor()方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor()方法，如果没有显式定义，一个空的constructor()方法会被默认添加。

```
class Point {
}

// 等同于
class Point {
  constructor() {}
}
```

上面代码中，定义了一个空的类Point，JavaScript 引擎会自动为它添加一个空的constructor()方法。

静态方法
如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类名来调用，这就称为“静态方法”。

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}

Foo.classMethod() // 'hello'

var foo = new Foo();
foo.classMethod()
// TypeError: foo.classMethod is not a function
```

总结：

class里的非静态方法调用，必须实例化类，静态方法调用：类名.方法名,静态方法不能通过this.非静态方法名来调用非静态方法.

实例属性的新写法
实例属性除了定义在constructor()方法里面的this上面，也可以定义在类的最顶层。

```
class IncreasingCounter {
  constructor() {
    this._count = 0;
  }
  increment() {
    this._count++;
  }
}
```

上面代码中，实例属性this._count定义在constructor()方法里面。另一种写法是，这个属性也可以定义在类的最顶层，其他都不变。

```
class IncreasingCounter {
  _count = 0;
  increment() {
    this._count++;
  }
}
```

上面代码中，实例属性_count与取值函数value()和increment()方法，处于同一个层级。这时，不需要在实例属性前面加上this。

这种新写法的好处是，所有实例对象自身的属性都定义在类的头部，看上去比较整齐，一眼就能看出这个类有哪些实例属性。

父类方法可以重写

```
//父类
class Phone {
    //构造方法
    constructor(brand) {
        this.brand = brand;

    }
    //对象方法
    call() {
        console.log('我可以打电话!!!')
    }

}

//子类
class SmartPhone extends Phone {

    constructor(brand, color) {
        super(brand); //super调用父级构造方法
        this.color =color;

    }

    //方法重写
    call(){
        console.log('我可以进行视频通话!!');
    }


}
```

```
//实例化对象
const iPhone6s = new SmartPhone('苹果', '白色');
//调用重写方法
iPhone6s.call();
```

取值函数（getter）和存值函数（setter）

与 ES5 一样，在“类”的内部可以使用get和set关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。对属性进行存取值.

```
class MyClass {
  constructor() {
    // ...
  }
  get prop() {
    return  this._prop;

  }
```

```
  set prop(value) {
    this._prop=value;

  }
}
let inst = new MyClass();
inst.prop = 123;
// setter: 123
inst.prop
// 123
```

```
//父类
class Phone {
    //构造方法
    constructor(brand, color, price) {
        this.brand = brand;
        this.color = color;
        this.price = price;
    }

    //对象方法
    call() {
        console.log('我可以打电话!!!')
    }

}

//子类
class SmartPhone extends Phone {
    static people='雷军'
    constructor(brand, color, price, screen, pixel) {
        super(brand, color, price);
        this.screen = screen;
        this.pixel = pixel;
    }

    //子类方法
    photo(){
        console.log('我可以拍照!!');
    }

    playGame(){
        console.log('我可以玩游戏!!');
    }

    //方法重写
    call(){
        console.log('我可以进行视频通话!!');
    }

    //静态方法  静态方法不能调用this.*** 属性 只能调用static 定义的属性
    static run(){
        console.log('小米手机的创始人是'+this.people)
        console.log('我可以运行程序')
    }
```

```
    static connect(){
        console.log('我可以建立连接')
    }

}

//实例化对象
const Nokia = new Phone('诺基亚', '灰色', 230);
const iPhone6s = new SmartPhone('苹果', '白色', 6088, '4.7inch','500w');

//调用子类方法
iPhone6s.playGame();
//调用重写方法
iPhone6s.call();
//更改静态属性
SmartPhone.people='董明珠';
//调用静态方法
SmartPhone.run();
```

2.16. ES6 模块

基本用法 export 与 import

模块导入导出各种类型的变量，如字符串，数值，函数，类。

1. 导出的函数声明与类声明必须要有名称（export default 命令另外考虑）。
2. 不仅能导出声明还能导出引用（例如函数）。
3. export 命令可以出现在模块的任何位置。
4. import 命令会提升到整个模块的头部，首先执行。

1、export 用法

```
/*-----export [test.js]-----*/
export let myName = "Tom";
let myAge = 20;
let myfn = function(){
    return "My name is" + myName + "! I'm '" + myAge + "years old."
}
let myClass =  class myClass {
    static a = "yeah!";
}
export {  myAge, myfn, myClass }//按需导出

/*-----import [xxx.js]-----*/
import { myName, myAge, myfn, myClass } from "./test.js"; //按需导入
console.log(myfn());// My name is Tom! I'm 20 years old.
console.log(myAge);// 20
console.log(myName);// Tom
console.log(myClass.a );// yeah!
```

报错 ”Cannot use import statement outside a module“

```
需要在script标签上添加属性type="module"
```

as 的用法

```
/*-----export [test1.js]-----*/
let myName = "Tom";
export { myName }

/*-----export [test2.js]-----*/
let myName = "Jerry";
export { myName }

/*-----import [xxx.js]-----*/
import { myName as name1 } from "./test1.js";
import { myName as name2 } from "./test2.js";

console.log(name1);
console.log(name2);
```

不同模块导出接口名称命名重复， 使用 as 重新定义变量名。

2、export default 用法

```
/*-----export [test.js]-----*/
let name='tom';
const say=function() {
  console.log(this.name)
}
export default {name,say}
```

```
/*-----import [xxx.js]-----*/
import person from "./1.js";
console.log(person.name)
person.say()
```

3、export 和export default的区别

export

- 每个文件中可使用多次export命令
- import时需要知道所加载的变量名或函数名
- import时需要使用{}，或者整体加载方法

export	export default
每个文件中可使用多次export命令	每个文件中只能使用一次export default命令
import时需要知道所加载的变量名或函数名	import时可指定任意名字

2.17. 数值扩展

1.二进制和八进制

ES6 提供了二进制和八进制数值的新的写法，分别用前缀0b和0o表示。

2.Number.isFinite()与Number.isNaN()

Number.isFinite() 用来检查一个数值是否为有限的

Number.isNaN() 用来检查一个值是否为NaN

3.Number.parseInt()与Number.parseFloat()

ES6 将全局方法parseInt和parseFloat，移植到Number对象上面，使用不变。

4.Math.trunc

用于去除一个数的小数部分，返回整数部分。

5.Number.isInteger

Number.isInteger() 用来判断一个数值是否为整数

2.17. 对象扩展

ES6新增了一些Object对象的方法

1) Object.is 比较两个值是否严格相等，与『===』行为基本一致（+0 与 NaN）

2) Object.assign 对象的合并，将源对象的所有可枚举属性，复制到目标对象

3) **proto**、setPrototypeOf、 setPrototypeOf可以直接设置对象的原型