一. 概念和发展

1. 什么是ES?

ECMAScript是根据MCMA-262标准，实现的通用脚本语言规范
1997年，JavaScript 1.1 作为一个草案提交给欧洲计算机制造商协会（ECMA）。第39技术委员会(TC39)被委派来 “标准化一个通用、跨平台、中立于厂商的脚本语言的语法和语义”，锤炼出了ECMA-262第一版，定义了名为 ECMAScript 的全新脚本语言。
2009年，ECMA-262第五版(ES5)发布。
2015年，ECMA-262第六版(ES6或者叫ES 2015语言规范)

2. ECMAScript和JavaScript的关系与区别

ECMAScript 和 JavaScript 的关系是，前者是后者的规格，后者是前者的一种实现（另外的 ECMAScript 方言还有 JScript 和 ActionScript）。在日常场合，这两个词是可以互换的

1996 年 11 月，JavaScript 的创造者 Netscape 公司，决定将 JavaScript 提交给标准化组织 ECMA，希望这种语言能够成为国际标准。次年，ECMA 发布 262 号标准文件（ECMA-262）的第一版，规定了浏览器脚本语言的标准，并将这种语言称为 ECMAScript。

JavaScript比ECMA-262的含义多得多，一个完整的JavaScript实现应该由以下三个部分组成：
1.核心（ECMAScript）：提供核心语言功能，基于ES规范。
它规定了语言的组成部分：语法、类型、语句、关键字、保留字、操作符、对象
2.文档对象模型（DOM）：提供访问和操作网页内容的方法和接口。
DOM：Document Object Model的缩写，DOM把整个网页映射为一个多层节点结构。
HTML页面中的每一个组成部分都是某种类型的节点。这些节点又包含着不同类型的数据。
3.浏览器对象模型（BOM）：提供与浏览器交互的方法和接口。
BOM：Browser Object Model的缩写。
浏览器提供了可访问和操作浏览器窗口的浏览器对象模型。

二. let命令

1. 基本用法

ES6新增了let命令，用来声明变量。它的用法类似于var，但由它声明的变量，只在let命令所在的代码块内有效。

```
let a;
let a,b,c;
let a = 1,b = 'hello',c = [];
```

2. 特点

1. let命令只在声明所在的块级作用域内有效。

```
{
    var a = 1;
    let b = 10;
}
console.log("a",a); // a 1
console.log("b",b); // b is not defined  -- Error
```

2. let命令声明的常量不存在变量提升，存在暂时性死区，必需先声明、后使用。

```
var tmp = new Date();
function f() {
    console.log("111",tmp); // tmp is not defined  -- Error
    let tmp = "hello world";  // var tmp = "hello world";
    console.log("222",tmp);
}
f();
console.log("333",tmp);
```

3. let声明的变量不可重复声明。

```
var message = "Hello!";
var age = 25;

// 以下两行都会报错
let message = "Hi!";
const age = 30;
```

3. 块级作用域

ES5 只有全局作用域和函数作用域，没有块级作用域。这带来很多不合理的场景。

1. 内层变量可能会覆盖外层变量

```
var tmp = new Date();
function f() {
  console.log(tmp);
  if (false) {
    var tmp = 'hello world';
  }
}
f(); // undefined
```

2. 用来计数的循环变量泄露为全局变量

```
var s = 'hello';
for(var i = 0; i< s.length; i++){
    console.log("i",i);
}
console.log("i",i);
```

ES6中，let实际上为 JavaScript 新增了块级作用域。ES6 允许块级作用域的任意嵌套。

```
// 示例一
function f1() {
  let n = 5;
  if (true) {
    let n = 10;
  }
  console.log(n);  // 5
}
f1();

// 实例二
var a = [];
```

```
for (var i = 0; i < 10; i++) {
  a[i] = function () {
    console.log("msg1",i);
  };
}
a[6]();  // msg1 10
console.log("msg2",i);  // msg2 10


var a = [];
for (let i = 0; i < 10; i++) {
  a[i] = function () {
    console.log("msg1",i);
  };
}
a[6]();  // msg1 6
console.log("msg2",i);  // i is not defined -- Error
```

三. const命令

const声明一个只读的常量。一旦声明，常量的值就不能改变。

```
const PI = 3.1415;
console.log(PI); // 3.1415
PI = 3;          // Assignment to constant variable.
```

特点：

1. const一旦声明变量，就必须立即初始化，不能在后续赋值。对于const 常量来说，只声明不赋值，就会报错

   ```
   if (true) {
       const MAX = 5;
   }
   console.log(MAX); // MAX is not defined
   ```

2. 对于复合类型的变量，变量名不指向数据，而是指向数据所在的地址。const命令只保证变量名指向的地址不变，并不保证该地址的数据不变。

   ```
   // 数组
   const a = [];
   a.push('Hello');
   a.push('World');
   console.log(a)     // ["Hello", "World"]
   a = ['YinHai'];    // Assignment to constant variable.

   // 对象
   const foo = {};
   foo.prop = 123;
   console.log(foo.prop) // 123
   foo = {};             // Assignment to constant variable.
   ```

四. 解构赋值

1. 引入

ES6允许按照一定模式，从数组，Object对象，String...中提取值，这被称为解构（Destructuring）；然后将提取出来的值赋值给变量，这个过程被称为解构赋值。

```
// ES6以前，为变量赋值，只能直接指定值。
var a = 1;
var b = 2;
var c = 3;

var a = 1,b = 2,c = 3;

// ES6允许通过相同的结构，解构赋值
// “模式匹配”，只要等号两边的模式、结构相同，左边的变量就会被赋予对应的值。
var [a, b, c] = [1, 2, 3];
```

2. 数组的解构赋值

```
// 一维数组
let [a, , c] = [1, 2, 3];
a // 1
c // 3

// 多维数组
let [a, [[b], c]] = [1, [[2], 3]];
a // 1
b // 2
c // 3

// 解构失败（等号两边的模式不同）：
let [a, [b]] = [1];
console.log(a);
console.log(b);
```

1.不完全解构

不完全结构：等号两边的模式相同，但变量的数量和解构的值的数量不同）：

a. 需要被赋值变量的数量 > 解构的值的数量

```
let [a] = [];
console.log(a);

let [a, b] = [1];
console.log(a);
console.log(b);

// 没有数据可以赋值的变量，默认值为undefined。
```

b. 需要赋值变量的数量 < 解构的值的数量

```
let [x, y] = [1, 2, 3];
x // 1
y // 2

let [a, [b], d] = [1, [2, 3], 4];
a // 1
b // 2
d // 4
```

2.带默认值

解构赋值允许变量指定默认值。

```
var [x = 'a'] = [];
console.log(x); // x='a'

[x, y = 'b'] = ['a',]; // x='a', y='b'
console.log(x);
console.log(y);

[x, y = 'b'] = ['a', undefined]; // x='a', y='b'
console.log(x);
console.log(y);

// 如果解构的值严格等于undefined，变量的默认值生效。
// 如果解构的值不严格等于undefined，变量的值为指定的值。
// ES6严格相等运算符（===）
[x, y = 'b'] = ['a', null]; // x='a', y='null'
console.log(x);
console.log(y);
```

3.对象的解构赋值

```
// 根据对象的key对应
let { username: username, password : password} = { username: "aaa", password:
"bbb" };
let { username, password } = { username: "aaa", password: "bbb" };
console.log(username);
console.log(password);

// 对象的解构与数组有一个重要的不同 数组的元素是按次序排列的，变量的取值由它的位置决定；
let { password, username } = { username: "aaa", password: "bbb" };
console.log(username);
console.log(password);

// 变量名和属性名不一致
let { user, pass } = { username: "aaa", password: "bbb" };
console.log(user);
console.log(pass);
/**
    对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。
    真正被赋值的是后者，而不是前者。
**/
var res = {
    'ok':1,
    'data':{
```

```
        'top':[1,2,3],
        'center':[2,3,4]
    },
    'mis':'请求成功'
};
let {ok,data,mis} = res;

const node = {
    loc:{
        start:{
            line: 1,
            column: 5
        }
    }
};
let {loc, loc : { start }, loc : { start: { line }}} = node; // key主要用于中间匹配用
```

4. 字符串的解构赋值

```
const [a,b,c,d,e] = "hello";
console.log(a,b,c,d,e);

// 数组、类似数组的对象都有一个length属性，
let {length : len01} = [1,2,3,4,5];
console.log(len01);
let {length : len02} = 'hello';
console.log(len02);
```

5. 数值Number和布尔值的解构赋值（了解）

解构赋值的规则是，只要等号右边的值不是对象，就先将其转为对象。

```
// 由于undefined和null无法转为对象，所以对它们进行解构赋值，都会报错。
let { prop: x } = undefined;
// Cannot destructure property `prop` of 'undefined' or 'null'
let { prop: y } = null;
// Cannot destructure property `prop` of 'undefined' or 'null'

// 解构赋值时，如果等号右边是数值和布尔值，则会先转为对象。
// 实际进行的是对象的解构赋值

// 数值
console.log(Number.prototype); // prototype可以获取到当前对象下的属性，包括构造函数 方法

let {toString: s,valueOf: v} = 123;
console.log(s);
console.log(v);

let boo = (s === Number.prototype.toString);
console.log(boo);
boo = (v === Number.prototype.valueOf);
console.log(boo);

// 布尔值
```

```
console.log(Boolean.prototype);

let {toString: s,valueOf: v} = true;
console.log(s);
console.log(v);
```

6. 函数参数的解构赋值

函数的参数也可以使用解构赋值。

```
function add([x, y]){
    return x + y;
}
let arr = [1,2];
console.log(add(arr));// 3
```

函数参数的解构也可以使用默认值。

```
function move({x = 0, y = 0}) {
    return [x, y];
}
console.log(move({x: 3, y: 8})); // [3, 8]
console.log(move({x: 3}));       // [3, 0]
console.log(move({}));           // [0, 0]
console.log(move());             // Cannot destructure property `x` of
'undefined' or 'null'.

function move({x = 0, y = 0}={}) {
  return [x, y];
}
console.log(move());             // [0, 0]
```

7. 解构赋值的用途

1.交换变量

```
let [x, y] = [111,222];
[x, y] = [y, x];
console.log([x, y]); // [222, 111]
```

2.获取函数的多个返回值

```
// 返回一个数组
function example() {
    return [1, 2, 3];
}
let [a, b, c] = example();
console.log([a, b, c]);

// 返回一个对象
function example() {
    return {
        a: 1,
        b: 2
     };
```

```
}
var { a, b } = example();
console.log({ a, b });
```

3.函数参数的定义

解构赋值可以方便地将一组参数与变量名对应起来。

```
// 参数是一组有序的值
function f([x, y, z]) { ... }
f([1, 2, 3]);

// 参数是一组无序的值
function f({x, y, z}) { ... }
f({z: 3, y: 2, x: 1});
```

4.提取JSON数据

解构赋值对提取JSON对象中的数据，尤其有用。

```
let jsonData = {
    id: 42,
    status: "OK",
    data: [867, 5309]
};

let { id, status, data: number } = jsonData;

console.log(id);
console.log(status);
console.log(number);

let [a,b] = number;
console.log(a);
console.log(b);
```

5.函数参数的默认值

指定参数的默认值，就避免在函数体内部再写var foo = config.foo || 'default foo';这样的语句。

```
jQuery.ajax = function (url, {
    async = true,
    beforeSend = function () {},
    cache = true,
    complete = function () {},
    crossDomain = false,
    global = true,
      // ... more config
}) {
      // ... do stuff
};
```

6.遍历Map结构

```
// 任何部署了Iterator接口的对象，都可以用for...of循环遍历。
```

```
// Map结构原生支持Iterator接口，配合变量的解构赋值，获取键名和键值就非常方便。
var map = new Map();
map.set('first', 'hello');
map.set('second', 'world');

for (let [key, value] of map) {
    console.log(key + " is " + value);
}
// first is hello
// second is world

// 如果只想获取键名，或者只想获取键值，可以写成下面这样。
// 获取键名
for (let [key] of map) {
     // ...
}
```

五. 字符串模板

```
// 传统的JavaScript语言，输出模板通常是这样写的。
$('#result').append(
    'There are <b>' + basket.count + '</b> ' +
    'items in your basket, ' +
    '<em>' + basket.onSale +
    '</em> are on sale!'
);
// 这种写法相当繁琐不方便，各种引号 加号 转义符号 效率低

// ES6引入了模板字符串解决这个问题。
$('#result').append(`
    There are <b>${basket.count}</b> items
    in your basket, <em>${basket.onSale}</em>
    are on sale!
`);
```

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

它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

```
// 普通字符串
`In JavaScript '\n' is a line-feed.`

// 多行字符串
`In JavaScript this is
not legal.`
console.log(`string text line 1
string text line 2`);

// 字符串中嵌入变量
var name = "Bob", time = "today";
`Hello ${name}, how are you ${time}?`

// 如果在模板字符串中需要使用反引号，则前面要用反斜杠转义。
var greeting = `\`Yo\` World!`;
```

六. 函数参数的默认值

在ES6之前，不能直接为函数的参数指定默认值，
ES6允许为函数的参数设置默认值，即直接写在参数定义的后面。

```
// 示例一
function log(x, y = 'World') {
    console.log(x, y);
}

log('Hello') // Hello World
log('Hello', 'China') // Hello China
log('Hello', '') // Hello

// 示例二
function Point(x = 0, y = 0) {
    this.x = x;
    this.y = y;
}

let p = new Point();
console.log(p);// 0,0

p = new Point(1,2);
console.log(p);1,2
```

七. rest参数

```
// ES6引入rest参数（形式为“...变量名”），用于获取函数的多个参数，可以不再使用arguments对象。
// arguments对象：可以在函数内访问所有的实参

function f01() {
   console.log(arguments[0]);
   console.log(arguments[1]);
   console.log(arguments[2]);
}

f01(12, 23, 33);   //12 23 33

// 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。
function f02(...values) {
   let sum = 0;
   for (var val of values) {
       sum += val;
   }
   return sum;
}

let result = f02(2, 5, 8) // 15
console.log(result);

// 注意：rest参数之后不能再有其他参数（即rest参数只能是最后一个参数），否则会报错。
function f(a, ...b, c) {
    // ...
```

```
}

// 函数的length属性代表函数的参数个数。
// length属性，不包括已指定默认值的参数
// length属性，不包括rest参数。

let len = (function(a) {}).length;  // 1
console.log(len);

len = (function (a = 5) {}).length; // 0
console.log(len);

len =(function (a, b, c = 5) {}).length; // 2
console.log(len);

len =(function(...a) {}).length;  // 0
console.log(len);

len = (function(a, ...b) {}).length ; // 1
console.log(len);
```

八. 扩展运算符

```
// 扩展运算符（spread）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。
console.log(...[1, 2, 3]);// 1 2 3

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

console.log([...document.querySelectorAll('body')]);// [<body>]

// 该运算符主要用于向函数传递实际参数。
// （rest参数定义函数的形式参数;扩展运算符提供实际参数）
function f02(...values) {
   let sum = 0;
   for (var val of values) {
       sum += val;
   }
   return sum;
}
//普通参数
let result = f02(2, 5, 8) // 15
console.log(result);

//扩展运算符参数
result = f02(...[3,6,9]) // 18
console.log(result);

//普通参数与扩展运算符参数结合使用
result = f02(666, ...[1,4,7]) // 678
console.log(result);
```

应用：

1. 复制数组

```
const a1 = [1, 2];
// 写法一
const a2 = [...a1];
// 写法二
const [...a3] = a1;
```

2. 合并数组

```
// 扩展运算符提供了数组合并的新写法。
let arr1 = ['a', 'b'];
let arr2 = ['c'];
let arr3 = ['d', 'e'];

// ES5的合并数组
let result = arr1.concat(arr2, arr3);
// [ 'a', 'b', 'c', 'd', 'e' ]

// ES6的合并数组
result = [...arr1, ...arr2, ...arr3]
// [ 'a', 'b', 'c', 'd', 'e' ]
```

3. 解构赋值结合

```
// 扩展运算符可以与解构赋值结合起来，用于生成数组。
// 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。
const [...butLast, last] = [1, 2, 3, 4, 5];
console.log([...butLast, last]);


const [first, ...middle, last] = [1, 2, 3, 4, 5];
console.log([first, ...middle, last]);


const [first, ...rest] = [1, 2, 3, 4, 5];
console.log([first, ...rest]);
```

4. 字符串

```
// 扩展运算符还可以将字符串转为真正的数组。
let arr = [...'hello']
console.log(arr);
// [ "h", "e", "l", "l", "o" ]
```

5. 转为真正的数组

任何Iterator接口的对象，都可以用扩展运算符转为真正数组。

```
var nodeList = document.querySelectorAll('body');
var array = [...nodeList];
console.log(array);
// 上面代码中，querySelectorAll方法返回的是一个nodeList对象。
// 它不是数组，而是一个类似数组的对象。
// 这时，扩展运算符可以将其转为真正的数组，原因就在于NodeList对象实现了Iterator接口。
```

```
// 对于那些没有部署Iterator接口的类似数组的对象，扩展运算符就无法将其转为真正的数组。
let arrayLike = {
    '0': 'a',
    '1': 'b',
    '2': 'c',
    length: 3
};

let array = [...arrayLike];
// object is not iterable (cannot read property Symbol(Symbol.iterator))
console.log(array);
// 上面代码中，arrayLike是一个类似数组的对象，但是没有部署Iterator接口，扩展运算符就会报错。
// 这时，可以改为使用Array.from方法将arrayLike转为真正的数组。
```

6. Map和Set结构

```
// 扩展运算符内部调用的是数据结构的Iterator接口，
// 因此只要具有Iterator接口的对象，都可以使用扩展运算符

// Map结构：
    let map = new Map([
        [1, 'one'],
        [2, 'two'],
        [3, 'three'],
    ]);

    let keyArray = [...map.keys()];
    console.log(keyArray);

    let valueArray = [...map.values()];
    console.log(valueArray);]
```

九. 箭头函数

```
// ES6允许使用“箭头”（=>）定义函数。
v => v;

let f = v => v;
// 该函数等同于：
let f = function(v) {
    return v;
};


// 通过对比发现，上述定义的箭头函数是拥有一个参数、一个返回语句的函数
// 箭头函数是函数式编程的一种体现，将更多的关注点放在输入和输出的关系，省去过程中的一些因素。
// 箭头函数相当于匿名函数。

// 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。
let f = () => 5;
// 等同于
let f = function () { return 5 };

let sum = (num1, num2) => num1 + num2;
// 等同于
```

```
let sum = function(num1, num2) {
    return num1 + num2;
};

// 如果箭头函数的代码块不仅只有return语句，就要使用语句块将它们括起来。
let sum = (num1, num2) => {
    console.log(num1 + num2);
    return num1 + num2;
};

// 由于大括号在箭头函数中被解释为代码块，
// 所以如果箭头函数返回一个对象，必须在对象外面加上括号。
let getTempItem = id => ({ id: id, name: "Temp" });
// 等同于
let getTempItem = function (id) { return { id: id, name: "Temp" } };

// 箭头函数可以与解构赋值结合使用。
const full = ({ first, last }) => first + ' ' + last;
// 等同于
function full(person) {
    return person.first + ' ' + person.last;
}

// 箭头函数使得表达更加简洁。
    const isEven = n => n % 2 == 0;
    const square = n => n * n;
// 上面代码只用了两行，就定义了两个简单的工具函数。
// 如果不用箭头函数，可能就要占用多行，而且还不如现在这样写醒目。


// 箭头函数简化回调函数。
// 正常函数写法
[1,2,3].map(function (x) {
    return x * x;
});
[1,4,9]

// 箭头函数写法
[1,2,3].map(x => x * x);

// rest参数、扩展运算符、箭头函数结合。

const numbers = (...nums) => nums;
let result = numbers(1, 2, 3, 4, 5);
console.log(result);
// [1,2,3,4,5]

result = numbers(...[6, 7, 8, 9, 0]);
console.log(result);
// [6, 7, 8, 9, 0]

const headAndTail = (head, ...tail) => [head, tail];
result = headAndTail(1, 2, 3, 4, 5);
console.log(result);
// [1,[2,3,4,5]]
```

箭头函数有几个使用注意点。
（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。
（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。
（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。

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

var id = 21;

function fn01() {
   setTimeout(function () {
            console.log("-- fn01 id-- : ", this.id);
   }, 100);
}
fn01.call({id: 42});

// 如果是普通函数，执行时this应该指向全局对象window，这时应该输出21。

function fn02() {
   setTimeout(() => {
      console.log("-- fn02 id-- : ", this.id);
   }, 100);
}
fn02.call({id: 42});

// 上面代码中，setTimeout的回调函数是一个箭头函数，
// 这个箭头函数的定义生效是在foo函数生成时，而它的真正执行要等到100毫秒后。
// 箭头函数导致this总是指向函数定义生效时所在的对象（本例是{id: 42}），所以输出的是42。
// 箭头函数可以让setTimeout里面的this，绑定定义时所在的作用域，而不是指向运行时所在的作用域。
```

十. Iterator和for...of循环

Iterator（遍历器）的概念

JavaScript原有的表示“集合”的数据结构，
主要是数组（Array）和对象（Object），
ES6又添加了Map和Set。这样就有了四种数据集合，
用户还可以组合使用它们，定义自己的数据结构，比如数组的成员是Map，Map的成员是对象。
这样就需要一种统一的接口机制，来处理所有不同的数据结构。
遍历器（Iterator）就是这样一种机制。它是一种接口，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（即依次处理该数据结构的所有成员）。

有些数据结构原生具备Iterator接口（比如数组），即不用任何处理，就可以被for...of循环遍历，有些就不行（比如对象）。原因在于，这些**数据结构原生部署了Symbol.iterator属性**，另外一些数据结构没有。凡是部署了Symbol.iterator属性的数据结构，就称为部署了遍历器接口。调用这个接口，就会返回一个遍历器对象。

```
console.log(Array.prototype);
console.log(Object.prototype);
console.log(Map.prototype);
console.log(Set.prototype);

// ES6规定，默认的Iterator接口部署在数据结构的Symbol.iterator属性，
// 或者说，一个数据结构只要具有Symbol.iterator属性，就可以认为是“可遍历的”（iterable）。
```

```
// 调用Symbol.iterator方法，就会得到当前数据结构默认的遍历器生成函数。
// Symbol.iterator本身是一个表达式，返回Symbol对象的iterator属性，这是一个预定义好的、类型为Symbol的特殊值，所以要放在方括号内。

 const arr = ['red', 'green', 'blue'];
 let iterator = arr[Symbol.iterator]();

// for...of循环
// ES6借鉴C++、Java、C#和Python语言，引入了for...of循环，作为遍历所有数据结构的统一的方法。
// 一个数据结构只要部署了Symbol.iterator属性，就被视为具有iterator接口，就可以用for...of循环遍历它的成员。
// 也就是说，for...of循环内部调用的是数据结构的Symbol.iterator方法。
// for...of循环可以使用的范围包括数组、Set和Map结构、某些类似数组的对象（比如arguments对象、DOM NodeList对象）、Generator对象，以及字符串。

// 数组原生具备iterator接口，for...of循环本质上就是调用这个接口产生的遍历器

const arr = ['red', 'green', 'blue'];

let iterator = arr[Symbol.iterator]();
for (let v of iterator) {
    console.log(v);
}

for (let v of arr) {
    console.log(v);
}

// JavaScript原有的for...in循环，遍历获得对象的键名。{name:123,pass:456}
// ES6提供for...of循环，遍历获得键值。

var arr = ['a', 'b', 'c', 'd'];

for (let a in arr) {
    console.log(a);
}

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

// 类似数组的对象

// 类似数组的对象包括好几类。
// 下面是for...of循环用于字符串、DOM NodeList对象、arguments对象的例子。

    // 字符串
    let str = "hello";
    for (let s of str) {
        console.log(s);
    }

    // DOM NodeList对象
    let container = document.querySelector(".parent");
    // 获取tabs
    let tabs = container.querySelectorAll(".tab");
    for (let tab of tabs) {
```

```
        tab.style.color = "blue";
    }

    // arguments对象
    function printArgs() {
        for (let x of arguments) {
            console.log(x);
        }
    }
    printArgs('a', 'b');
```

十一. Promise对象

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。

所谓Promise，
简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。
从语法上说，Promise是一个对象，从它可以获取异步操作的消息。

Promise对象有以下两个特点。
（1）对象的状态不受外界影响，由异步操作的状态决定。
Promise对象代表一个异步操作，有三种状态：Pending（进行中）、Resolved（已完成）和Rejected（已失败）。
只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。
这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

（2）一旦状态改变，就不会再变，任何时候都可以得到这个结果。
Promise对象的状态改变，只有两种可能：从Pending变为Resolved和从Pending变为Rejected。
只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。
就算改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。
这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

```
// 基本用法
// ES6规定，Promise对象是一个构造函数，用来生成Promise实例。
let promise = new Promise(function (resolve, reject) {
    // ... some code
    if (/*异步操作成功*/ ) {
        resolve(value);
    } else {
        reject(error);
    }
});

// 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Reject状态的回调函数。
```

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

// 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Reject时调用。
// 其中，第二个函数是可选的，不一定要提供。这两个函数都接受Promise对象传出的值作为参数。

    // 写法一
    let promise = new Promise(function(resolve, reject) {
        reject(new Error('test'));
    });
    promise.catch(function(error) {
        console.log(error);
    });

    // 写法二
    let promise = new Promise(function(resolve, reject) {
        try {
            throw new Error('test');
        } catch(e) {
            reject(e);
        }
    });
    promise.catch(function(error) {
        console.log(error);
    });

// 比较上面两种写法，可以发现reject方法的作用，等同于抛出错误。
// promise通过reject方法抛出一个错误，就被catch方法指定的回调函数捕获。


// 一般来说，不要在then方法里面定义Reject状态的回调函数（即then的第二个参数），而是使用catch方法来捕获异常。

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

实例：加载图片文件显示到页面上

```
// 加载图片 return promise
function imgLoad(src) {
    return new Promise(function(resolve, reject) {
        let img = document.createElement('img');
        img.className = 'm-img';
        img.src = src;
```

```
        img.onload = function(){
            resolve(img);
        }
        img.onerror = function(err){
            reject(err);
        }
    });
}
// 将单个图片添加到界面上
function showImg(img){
    let p = document.createElement('p');
    p.appendChild(img);
    document.body.appendChild(p);
}

// 调用，使用then处理结果（应该还使用catch捕获异常，这里省略）
imgLoad('http://qpzo4o916.hn-bkt.clouddn.com/dog.jpg').then(showImg);
imgLoad('http://qpzo4o916.hn-bkt.clouddn.com/ha.jpg').then(showImg);
imgLoad('http://qpzo4o916.hn-bkt.clouddn.com/panda.jpg').then(showImg);
```

扩展：

1. Promise.all()

```
// 以上面的实例为基础，了解Promise.all()的用法

// 添加多个图片到文件
function showImgs(imgs){
    imgs.forEach(img=>{
    document.body.appendChild(img);
    })
}

// 所有图片加载完成后再添加到界面
Promise.all([
    imgLoad('http://qpzo4o916.hn-bkt.clouddn.com/dog.jpg'),
    imgLoad('http://qpzo4o916.hn-bkt.clouddn.com/ha.jpg'),
    imgLoad('http://qpzo4o916.hn-bkt.clouddn.com/panda.jpg')
]).then(showImgs);
```

2. Promise.race()

```
// 以上面的实例为基础，了解Promise.race()的用法

// 一个图片加载完成后就添加到界面
Promise.race([
    imgLoad('http://qpzo4o916.hn-bkt.clouddn.com/dog.jpg'),
    imgLoad('http://qpzo4o916.hn-bkt.clouddn.com/ha.jpg'),
    imgLoad('http://qpzo4o916.hn-bkt.clouddn.com/panda.jpg')
]).then(showImg);
```

十二. Module

在 ES6 之前，社区制定了一些模块加载方案，最主要的有 CommonJS 和 AMD 两种。前者用于服务器，后者用于浏览器。ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。ES6 模块不是对象，而是通过`export`命令显式指定输出的代码，再通过`import`命令输入。

模块功能主要由两个命令构成：`export`和`import`。`export`命令用于规定模块的对外接口，`import`命令用于输入其他模块提供的功能。

```
// 在HTML网页中，ES5 脚本的常规用法
    <!-- 页面内嵌JavaScript脚本 -->
    <script type="application/javascript">
      // module code
     </script>

    <!--引入外部JavaScript脚本 -->
    <script type="application/javascript" src="path/myModule.js"></script>
```

export命令&import命令

```
// 1.导出变量
export let A = 123;
// 2.导出方法
export let test = function(){
    console.log('test');
};
// 3.导出对象
export const student = {
    name: 'Megan',
    age: 18
};

// or 先命名变量、方法、对象，再导出
let A = 123;
let test = function(){console.log('test')};
const student = {
    name: 'magan',
    age: 18
}
export {A,test,student};
export {A as x, test as y, student as c};

// ES6模块不是对象，而是通过export命令显式指定输出的代码，输入时也采用静态命令的形式。
// 页面内嵌ES6模块
<script type="module">
    import { stat, exists, readFile } from 'path/myModule';
    // other code
</script>

// 引入变量，方法，对象
import {A,test,student} from '../xxx.js' // 引入的名称和导出的名称要一致

// 引入部分(按需引入)
import {A} from '../xxx.js'

// 引入全部
import * as Test from '../xxx.js'
```

```
// ------------------------------------------------------------------------
--

/*
从前面的例子可以看出，使用import命令的时候，用户需要知道所要加载的变量名或函数名，否则无法加
载。
如果用户希望快速上手，不愿意阅读文档，去了解模块有哪些属性和方法。
为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到export default命令，为模块指定默认
输出。

本质上，export default就是输出一个叫做default的变量或方法，然后系统允许你为它取任意名字。
*/
// 例一
let A = 123;
let test = function(){console.log('test')};
const student = {
    name: 'Megan',
    age: 18
}
export default{
    A,
    test,
    student
}
// 这种写法，引入的时候不需要跟导出的名称一样，它将导出的对象命名的权力交给了引入方
import Test1 from '../xxx.js';
console.log(Test1.A);
console.log(Test1.test);
console.log(Test1.Hello);

// 例二
export default function foo03() {
        console.log(" foo03() ");
}
import x from '../xxx.js'
x();
```

注意:

1. export语句，要写在最顶层，不可写在函数或者代码块内部

```
function foo() {
    console.log(" -- foo() -- ");
    // export {foo};  // -- Error Unexpected token export
}
export {foo};
```

2. 在一个文件或模块中，export、import 可以有多个，export default 仅有一个。通过 export 向外暴露的成员，在导入时需要对应的变量名，并且必需要加{ }，通过 export default 向外暴露的成员，在导入时可以使用任意变量来接收，不能加{ }。

```
let a = 123;
export a; // --Error
// 正确写法
export {a};


export default function foo03() {
    console.log(" foo03() ");
}
import {x} from '../xxx.js' // -- Error 会去模块中查找名称为x的变量
// 正确写法
import x from '../xxx.js'
```

3. import语句可以与export语句写在一起

```
export {foo01, foo02} from '../xxx.js';
/*
    约等于下面两段语句，
    不过上面的导入导出复合写法，模块中没有导入 foo01 与 foo02
    上面代码中，export和import语句结合在一起，写成一行。
     但是从可读性考虑，不建议采用这种写法，而应该采用标准写法。
*/
import {foo01, foo02} from "./xxx.js";
export {foo01, foo02};

export * from '../xxx.js';
// 这里会忽略xxx.js中的default方法
```