ECMAScript 6-11

1. ECMAScript 相关介绍

1.1. 什么是 ECMA?

ECMA（European Computer Manufacturs Association）中文名称为欧洲计算机制造商协会，这个组织的目标就是评估、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?

EMCASript 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 只是其中一个。

查看所有标准：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

版本	时间	更新内容
第一版	1997年	制定了语言的基本语法
第二版	1998年	较小改动
第三版	1999年	引入正则、异常处理、格式化输出等。IE开始支持
第四版	2007年	过于激进，未发布
第五版	2009年	引入严格模式、JSON、扩展对象、数组、原型、字符串、日期方法
第六版	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
第七版	2016年	幂运算符、数组扩展、async/await关键字
第八版	2017年	async/await、字符串扩展
第九版	2018年	对象结构赋值、正则扩展
第十版	2019年	扩展对象、数组fang'fa

1.5. 谁在维护 ECMA-262

TC39（Technical Committee 39）是推进ECMA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，会议由会员公司的待变与特约专家出席

1.6. 为什么要学习 ES6

- ES6 的版本变化内容最多，具有里程碑意义
- ES6 加入了许多新的语法特性，编程实现更简单、高效
- 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. ECAMScript 6 新特性

2.1. 概述

1. let 关键字
    - 声明具有块级作用域的变量
2. const 关键字
    - 声明常量
3. 变量和对象的结构赋值
    - 简化变量声明
4. 模板字符串
    - 声明自带格式的字符串
5. 简化对象和函数写法
6. 箭头函数
    - 简化函数写法
7. ES6 中函数参数的默认值
    - 给函数的参数设置默认值
8. rest 参数
    - 拿到实参
9. 扩展运算符
    - 将一个数组转为哟还能逗号分隔的参数序列
10. Symbol
    - 表示独一无二的值
11. 迭代器
    - 用来遍历集合、数组等
12. 生成器
    - 是一种异步编程解决方案
13. Promise
    - 非常强大的异步编程的新解决方案
14. Set 集合
    - 类似数组，氮元素不重复的集合
15. Map 集合
    - 键值对集合
16. class 类
    - 像 java 实体类一样声明 js 类
17. 数值扩展
    - 增加了一些数值相关的方法
18. 对象扩展
    - 增加了一些对象相关的方法
19. 模块化
    - 模块化编程、组件化编程
20. Babel 对 ES6 模块化代码转换
    - 为了适配浏览器，将更新的ES规范转成 ES5 规范
21. ES6 模块化引起 npm 包
    - 像导入模块一样导入 npm 包

2.2. ES6 - let 关键字

特性:

let 关键字跟 var 一样，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
2. 块级作用域
3. 不存在变量提升
4. 不影响作用域链

let 创建变量代码示例:

```
let a // 单个声明
let b, c, d // 批量声明
let e = 100 // 单个声明并赋值
let f = 521, g = 'iloveyou', h = [] // 批量声明并赋值
```

1. 不允许重复声明:

```
let dog = 'jojo'
let dog = 'jojo'
// 报错:Uncaught SyntaxError: Identifier 'dog' has already been declared
```

2. 块级作用域（局部变量）

代码实现:

```
{
  let cat = 'mimi'
  console.log(cat)
}
console.log(cat)
// 报错: Uncaught ReferencError: cat is not defined
```

3. 不存在变量提升:

什么是变量提升:

就是在变量创建之前使用 （比如输出: 输出的是 undefined）， let 不存在， var 存在

代码示例:

```
console.log(p1) // 可输出默认值 => undefined
console.log(p2) // 报错: Uncaught ReferencError: p2 is not defined

var p1 = 'noe' // 存在变量提升
let p2 = 'nana' // 不存在变量提升
```

4. 不影响作用域链:

什么是作用域链：很简单，久事代码块内有代码块，跟常规编程语言一样，上级代码块中的局部变量下级可用

代码示例:

```
{
  let p = 'dd'
  function fn() {
    console.log(p) // 这里是可以用的
  }
  fn ()
}
console.log(p)  // 报错: Uncaught ReferencError: p is not defined
```

2.3. ES6 - const 关键字

特性:

const 关键字用来声明常量，const 声明有以下特点：

1. 声明必须赋初始值;
2. 标识符一般为大写 （默认）；
3. 不允许重复声明;
4. 值不允许修改;
5. 块级作用域 （局部变量）

代码示例：

```
const DOG = '大黄'
console.log(DOG)
```

1. 声明必须赋初始值：

```
const CAT  // 报错
const GOO = 'god'
```

2. 不允许重复声明：

```
const CAT = 'miew'
const CAT = 'mi'
// 报错
```

3. 值不允许修改：

```
const CAT = 'miew'
CAT = 'mimi'  // 报错
const ARR = ['dd', 'df', 'do']
ARR[0] = 'oo' // 这里是可以的
```

4. 块级作用域 （局部变量）：

```
{
  const CAT = 'mewo'
  console.log(CAT)
}
console.log(CAT) // 报错
```

5. 对于数组和对象的元素修改，不算做对常量的修改，不会报错

```
const TEAM = ['UZI', 'MLXG', 'Ming', 'letme']
TEAM.push('Meiko')
```

const 应用场景：声明对象类型数据使用 const， 非对象类型数据使用 let

2.4. ES6 - 变量的解构赋值

什么是解构赋值：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；

2.4.1. 数组的解构

```
const F4 = ['小沈阳', '刘能', '赵四', '宋小宝']
let [xiao,  liu,  zhao,  song] = F4
console.log(xiao)  // => 小沈阳
console.log(liu)  // => 刘能
console.log(zhao)  // => 赵四
console.log(song)  // => 宋小宝
```

2.4.2. 对象的解构

```
const zhao = {
  name: '赵本山',
  age: '不详', ],
  xiaopin: function () {
    console.log('我可以演小品')
  }
}

let {name, age, xiaopin} = zhao
console.log(name) // => 赵本山
console.log(age) // => 不详
console.log(xiaopin) // => function () {}
```

结构赋值的应用场景： 频繁使用方法、数组元素，就可以使用结构赋值形式

2.5. ES6 - 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。有以下两个特点

1. 字符串中可以出现换行符；

```
// 1、字符串中可以出现换行符
let str =
`<ul>
<li>大哥</li>
<li>二哥</li>
<li>三哥</li>
<li>四哥</li>
</ul>`
console.log(str)
```

2. 可以使用 ${xxx} 形式引用变量，进行变量拼接；

```
let s = "大哥"
let out = `${s}是我最大的榜样！`
console.log(out) // => 大哥是我最大的榜样！
```

模板字符串应用场景：当遇到字符串与变量拼接的情况使用

2.6. ES6 - 对象的简化写法

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；

代码示例：

```
let name = "滴滴"
let change = function(){
console.log("活着就是为了改变世界！")
}
//创建对象
const school = {
    // 完整写法
    // name:name,
    // change:change
    // 简化写法
    name,
    change,
    // 声明方法的简化
    say(){
        console.log("言行一致！")
    }
}
school.change()
school.say()
```

2.7. ES6 - 箭头函数

ES6允许使用箭头（=>）定义函数，箭头函数提供了一种更加简洁的函数书写方式，箭头函数多用于匿名函数的定义。

箭头函数注意点：

1. 如果形参只有一个，则小括号是可以省略的；
2. 函数体如果只有一条语句，则花括号是可以省略的，函数的返回值为该条语句的执行结果；
3. 箭头函数的this是静态的，始终指向函数声明时所在作用域下的this的值
4. 箭头函数不能作为构造函数实例化；
5. 箭头函数不能使用 arguments。

代码示例：

```
// ES6允许使用箭头（=>）定义函数
// 传统写法：无参数
var say = function(){
    console.log("hello! ");
}
say();
// ES写法2：无参数
let speak = () => console.log("hello 哈哈! ");
speak();
// 传统写法：一个参数
var hello = function(name){
return "hello " + name;
}
console.log(hello("dd"));
// ES6箭头函数：一个参数
let hi = name => "hi " + name;
运行结果：
console.log(hi("did"));
// 传统写法：多个参数
var sum = function(a,b,c){
return a + b + c;
}
console.log(sum(1,2,3));
// ES6箭头函数：多个参数
let he = (a,b,c) => a + b + c;
console.log(he(1,2,3));
// 特性
// 1、箭头函数的this是静态的，始终指向函数声明时所在作用域下的this的值
const school = {
    name : "大哥",
}
// 传统函数
function getName(){
    console.log("getName：" + this.name);
}
// 箭头函数
getName1 = () => console.log("getName1：" + this.name);
window.name = "滴滴";
// 直接调用
getName();
getName1();
// 使用call调用
getName.call(school);
getName1.call(school);
// 结论：箭头函数的this是静态的，始终指向函数声明时所在作用域下的this的值
// 2、不能作为构造实例化对象
// let Persion = (name,age) => {
// this.name = name;
// this.age = age;
```

```
// }
// let me = new Persion("滴滴",24);
// console.log(me);
// 报错：Uncaught TypeError: Persion is not a constructor
// 3、不能使用 arguments 变量
// let fn = () => console.log(arguments);
// fn(1,2,3);
// 报错：Uncaught ReferenceError: arguments is not defined
```

2.7.1. 箭头函数的实践与应用场景

需求-1：点击 div 2s 后颜色变成『粉色』：

传统写法：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>箭头函数的实践和应用场景</title>
        <style>
            div {
                width: 200px;
                height: 200px;
                background: #58a;
            }
        </style>
    </head>
    <body>
            <div id="ad"></div>
            <script>
                // 需求-1 点击 div 2s 后颜色变成『粉色』
                // 获取元素
                let ad = document.getElementById('ad');
                // 绑定事件
                ad.addEventListener("click", function(){
                    // 传统写法
                    // 定时器：参数1：回调函数；参数2：时间；
                    setTimeout(function(){
                        console.log(this);
                        this.style.background = 'pink';
                    },2000);
                    // 报错Cannot set property 'background' of undefined
                });
        </script>
    </body>
</html>

```

ES6 写法：

```
// 需求-1 点击 div 2s 后颜色变成『粉色』
// 获取元素
let ad = document.getElementById('ad');
// 绑定事件：这也是错误写法，这里的this还是window
// ad.addEventListener("click", () => {
// // ES6写法
```

```
// // 定时器：参数1：回调函数；参数2：时间；
// setTimeout(() => this.style.background = 'pink',2000);
// }
// )
// 绑定事件
ad.addEventListener("click", function(){
    // ES6写法
    // 定时器：参数1：回调函数；参数2：时间；
    // 这个this才是ad
    setTimeout(() => this.style.background = 'pink',2000);
    }
)

//需求-2 从数组中返回偶数的元素
const arr = [1, 6, 9, 10, 100, 25];
// const result = arr.filter(function(item){
// if(item % 2 === 0){
// return true;
// }else{
// return false;
// }
// });
const result = arr.filter(item => item % 2 === 0);
console.log(result);
```

箭头函数总结：

- 箭头函数适合与 this 无关的回调. 定时器, 数组的方法回调
- 箭头函数不适合与 this 有关的回调. 事件回调, 对象的方法

2.8. ES6 - 函数参数的默认值设置

ES6 允许给函数参数赋初始值

```
//1. 形参初始值 具有默认值的参数, 一般位置要靠后(潜规则)
function add(a,b,c=10) {
return a + b + c;
}
let result = add(1,2);
console.log(result); // 13
//2. 与解构赋值结合
// 注意这里参数是一个对象
function connect({host="127.0.0.1", username,password, port}){
    console.log(host)
    console.log(username)
    console.log(password)
    console.log(port)
}
connect({
    host: 'atguigu.com',
    username: 'root',
    password: 'root',
    port: 3306
})

```

2.9. ES6 - rest 参数

ES6 引入 rest 参数，用于获取函数的实参，用来代替 arguments；

> 参考文章：https://www.jianshu.com/p/50bcb376a419

代码示例：

```
function data(){
    console.log(arguments);
}
data("大哥","二哥","三哥","四哥");
// ES6的rest参数...args，rest参数必须放在最后面
function data(...args){
    console.log(args); // fliter some every map
}
data("大哥","二哥","三哥","四哥");
```

2.10. ES6 - 扩展运算符

`...`扩展运算符能将数组转换为逗号分隔的参数序列；

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；

```
// ... 扩展运算符能将数组转换为逗号分隔的参数序列
//声明一个数组 ...
const tfboys = ['易烊千玺', '王源', '王俊凯'];
// => '易烊千玺','王源','王俊凯'
// 声明一个函数
function chunwan() {
    console.log(arguments);
}
chunwan(...tfboys); // chunwan('易烊千玺','王源','王俊凯')

```

2.10.1. 扩展运算符的应用

```
//1. 数组的合并 情圣 误杀 唐探
const kuaizi = ['王太利','肖央'];
const fenghuang = ['曾毅','玲花'];
// 传统的合并方式
// const zuixuanxiaopingguo = kuaizi.concat(fenghuang);
const zuixuanxiaopingguo = [...kuaizi, ...fenghuang];
console.log(zuixuanxiaopingguo);
//2. 数组的克隆
const sanzhihua = ['E','G','M'];
const sanyecao = [...sanzhihua];// ['E','G','M']
console.log(sanyecao);
//3. 将伪数组转为真正的数组
const divs = document.querySelectorAll('div');
const divArr = [...divs];

```

```
15 console.log(divArr); // arguments 也可以转
```

2.11. ES6 - Symbol

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；

> 参考文章：https://blog.csdn.net/fesfsefgs/article/details/108354248

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；

```
1  //创建Symbol
2  let s = Symbol();
3  // console.log(s, typeof s);
4  let s2 = Symbol('尚硅谷');
5  let s3 = Symbol('尚硅谷');
6  console.log(s2==s3); // false
7  //Symbol.for 创建
8  let s4 = Symbol.for('尚硅谷');
9  let s5 = Symbol.for('尚硅谷');
10 console.log(s4==s5); // true
11 //不能与其他数据进行运算
12 // let result = s + 100;
13 // let result = s > 100;
14 // let result = s + s;
15 //JavaScript 数据类型
16 // USONB you are so niubility
17 // u undefined
18 // s string symbol
19 // o object
20 // n null number
21 // b boolean
```

JavaScript 数据类型：
USONB you are so niubility
u undefined
s string symbol
o object
n null number
b boolean

2.11.1. 对象添加 Symbol 类型的属性

```
1 // 向对象中添加方法 up down
2 let game = {
3     name:'俄罗斯方块',
4     up: function(){},
5     down: function(){}
6 };
```

```

// 我们要往game对象里面添加方法，但是怕game对象已经存在
// 同名方法，所以我们这时使用到了Symbol

// 方式一
// 声明一个对象
let methods = {
    up: Symbol(),
    down: Symbol()
};
game[methods.up] = function(){
console.log("我可以改变形状");
}
game[methods.down] = function(){
    console.log("我可以快速下降!!");
}

console.log(game);

// 方式二
let youxi = {
    name:"狼人杀",
    [Symbol('say')]: function(){
        console.log("我可以发言")
    },
    [Symbol('zibao')]: function(){
        console.log('我可以自爆');
    }
}

console.log(youxi);

// 如何调用方法？？？  讲师没讲，这是弹幕说的方法
let say = Symbol('say');
let youxi1 = {
    name:"狼人杀",
    [say]: function(){
        console.log("我可以发言")
    },
    [Symbol('zibao')]: function(){
        console.log('我可以自爆');
    }
}
youxi1[say]();
```

2.11.2. Symbol 的内置属性

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。可以称这些方法为魔术方法，因为它们会在特定的场景下自动执行；

属性/方法	描述
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.isConcatSpreadabal	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.unscopables	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.resplace	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。
Symbol.split	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。

备注：Symbol内置值的使用，都是作为某个对象类型的属性去使用；

代码演示：

```
class Person{
    static [Symbol.hasInstance](param){
        console.log(param);
        console.log("我被用来检测类型了");
        return false;
    }
}
let o = {};
console.log(o instanceof Person);
const arr = [1,2,3];
const arr2 = [4,5,6];
// 合并数组：false数组不可展开，true可展开
arr2[Symbol.isConcatSpreadable] = false;
console.log(arr.concat(arr2));
```

2.12. ES6 - 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了 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. 原生具备 Iterator 接口的数据（可用 for...of 遍历）
    - Array
    - Arguments
    - Set
    - Map
    - String
    - TypedArray
    - NodeList
3. 工作原理
    - 创建一个指针对象，指向当前数据结构的起始位置
    - 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
    - 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
    - 每调用 next 方法返回一个包含 value 和 done 属性的对象

注意：需要自定义遍历数据的时候，应该想到迭代器

代码示例：

```
// 声明一个数组
const xiyou = ['唐僧', '孙悟空', '猪八戒', '沙僧'];
// 使用 for...of 遍历数组
for(let v of xiyou){
console.log(v);
}
let iterator = xiyou[Symbol.iterator]();
// 调用对象的next方法
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
// 重新初始化对象，指针也会重新回到最前面
let iterator1 = xiyou[Symbol.iterator]();
console.log(iterator1.next());

```

2.12.1. 迭代器应用 - 自定义遍历对象

```
// 声明一个对象
const banji = {
    name: "终极一班",
    stus: [
        'xiaoming',
        'xiaoning',
        'xiaotian',
        'knight'
    ],
    [Symbol.iterator]() {

```

```
    // 索引变量
    let index = 0;
    // 保存this
    let _this = this;
    return {
        next: function() {
            if (index < _this.stus.length) {
                const result = {
                    value: _this.stus[index],
                    done: false
                };
                // 下标自增
                index++;
                // 返回结果
                return result;
            } else {
                    return {
                        value: undefined,
                        done: true
                    };
                }
            }
        };
    }
}
// 遍历这个对象
for (let v of banji) {
    console.log(v);
}
/*
* // 运行结果
* xiaoming
* xiaoning
* xiaotian
* knight
*/
```

2.13. ES6 - 生成器函数声明与调用

生成器函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同；

声明：

```
function * gen() {
  console.log(111)
}
```

代码示例：

```
// 生成器其实就是一个特殊的函数
// 异步编程 纯回调函数 node fs ajax mongodb
// yield：函数代码的分隔符
function * gen() {
    console.log(111);
    yield '一只没有耳朵';
```

```
    console.log(222);
    yield '一只没有尾部';
    console.log(333);
    yield '真奇怪';
    console.log(444);
}
let iterator = gen();
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log("遍历：");
//遍历
for(let v of gen()){
    console.log(v); // 每一次遍历打印 yield 语句后面的表达式结果或者字面量值
}
```

2.13.1. 生成器函数的参数传递

next()方法是可以传入参数的，传入的参数作为上一条 yield 语句的返回结果

```
function * gen(arg){
    console.log(arg);
    let one = yield 111;
    console.log(one);
    let two = yield 222;
    console.log(two);
    let three = yield 333;
    console.log(three);
}
let iterator = gen("AAA");
console.log(iterator.next()); // 会执行yield 111;
// next()方法是可以传入参数的，传入的参数作为第一条(上一条)语句yield 111的返回结果
console.log(iterator.next("BBB")); // 会执行yield 222;
console.log(iterator.next("CCC")); // 会执行yield 333;
console.log(iterator.next("DDD")); // 继续往后走，未定义;

```

2.13.2. 生成器函数实例1

```
// 异步编程 文件操作 网络操作（ajax，request） 数据库操作
// 需求：1s后控制台输出111 再过2s后控制台输出222 再过3s后控制台输出333
// 一种做法：回调地狱
// setTimeout(()=>{
//  console.log(111);
//  setTimeout(()=>{
//      console.log(222);
//          setTimeout(()=>{
//              console.log(333);
//          },3000)
//      },2000)
// },1000)
// 另一种做法
function one(){
    setTimeout(()=>{
        console.log(111);
```

```
        iterator.next();
    },1000)
}
function two(){
    setTimeout(()=>{
        console.log(222);
        iterator.next();
    },1000)
}
function three(){
    setTimeout(()=>{
        console.log(333);
        iterator.next();
    },1000)
}
function * gen(){
    yield one();
    yield two();
    yield three();
}
// 调用生成器函数
let iterator = gen();
iterator.next();

```

2.13.3. 生成器函数实例2

```
// 模拟获取: 用户数据 订单数据 商品数据
function getUsers(){
    setTimeout(()=>{
        let data = "用户数据";
        // 第二次调用next，传入参数，作为第一个的返回值
        iterator.next(data); // 这里将data传入
    },1000);
}
function getOrders(){
    setTimeout(()=>{
        let data = "订单数据";
        iterator.next(data); // 这里将data传入
    },1000);
}
function getGoods(){
    setTimeout(()=>{
        let data = "商品数据";
        iterator.next(data); // 这里将data传入
    },1000);
}
function * gen(){
    let users = yield getUsers();
    console.log(users);
    let orders = yield getOrders();
    console.log(orders);
    let goods = yield getGoods();
    console.log(goods); // 这种操作有点秀啊!
}
let iterator = gen();
iterator.next();
```

2.14. ES6 - Promise

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。

1. Promise 构造函数: Promise (excutor) {};
2. Promise.prototype.then 方法
3. Promise.prototype.catch 方法;

基本使用:

```
// 实例化 Promise 对象
// Promise 对象三种状态：初始化、成功、失败
const p = new Promise(function(resolve,reject){
    setTimeout(function(){
        // 成功
        // let data = "数据";
        // 调用resolve，这个Promise 对象的状态就会变成成功
        // resolve(data);
        // 失败
        let err = "失败了！";
        reject(err);
    },1000);
});

// 成功
// 调用 Promise 对象的then方法，两个参数为函数
p.then(function(value){ // 成功
    console.log(value);
}, function(season){ // 失败
    console.log(season);
});

```

2.14.1. Promise 封装读取文件

非 Promise 写法:

```
// 1、引入 fs 模块
const fs = require("fs");
// 2、调用方法，读取文件
fs.readFile("resources/text.txt",(err,data)=>{
    // 如果失败则抛出错误
    if(err) throw err;
    // 如果没有出错，则输出内容
console.log(data.toString());
});

```

Promise 封装方法:

```
// 1、引入 fs 模块
const fs = require("fs");
```

```
3
4  // 2、调用方法，读取文件
5  // fs.readFile("resources/text.txt",(err,data)=>{
6  // // 如果失败则抛出错误
7  // if(err) throw err;
8  // // 如果没有出错，则输出内容
9  // console.log(data.toString());
10 // });
11
12 // 3、使用Promise封装
13 const p = new Promise(function(resolve,data){
14     fs.readFile("resources/text.txt",(err,data)=>{
15         // 判断如果失败
16         if(err) reject(err);
17         // 如果成功
18         resolve(data);
19     });
20 });
21
22 p.then(function(value){
23     console.log(value.toString());
24 },function(reason){
25     console.log(reason); // 读取失败
26 })
```

2.14.2. Promise 封装 AJAX 请求

原生 AJAX 请求：

```
1  // 请求地址：https://api.apiopen.top/getJoke
2  // 原生请求
3  // 1、创建对象
4  const xhr = new XMLHttpRequest();
5  // 2、初始化
6  xhr.open("GET","https://api.apiopen.top/getJoke");
7  // 3、发送
8  xhr.send();
9  // 4、绑定事件，处理响应结果
10 xhr.onreadystatechange = function(){
11     // 判断状态
12     if(xhr.readyState == 4){
13         // 判断响应状态码 200-299
14         if(xhr.status>=200 && xhr.status<=299){
15             // 成功
16             console.log(xhr.response);
17         }else{
18             // 失败
19             console.error(xhr.status);
20         }
21     }
22 }
23
```

Promise 封装 AJAX 请求：

```
1  // 请求地址：https://api.apiopen.top/getJoke
```

```
const p = new Promise(function(resolve,reason){
    // 原生请求
    // 1、创建对象
    const xhr = new XMLHttpRequest();
    // 2、初始化
    xhr.open("GET","https://api.apiopen.top/getJoke");
    // 3、发送
    xhr.send();
    // 4、绑定事件，处理响应结果
    xhr.onreadystatechange = function(){
        // 判断状态
        if(xhr.readyState == 4){
            // 判断响应状态码 200-299
            if(xhr.status>=200 && xhr.status<=299){
                // 成功
                resolve(xhr.response);
            }else{
                // 失败
                reason(xhr.status);
            }
        }
    }
});

p.then(function(value){
    console.log(value.toString());
},function(reason){
    console.log(reason); // 读取失败
})

```

2.14.3. Promise.prototype.then

代码示例：

```
// 创建 Promise 对象
const p = new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve("用户数据");
    },1000);
});
// 调用then方法，then方法的返回结果是promise对象，
// 对象的状态有回调函数的结果决定；
const result = p.then(value => {
    console.log(value);
    // 1、如果回调函数中返回的结果是 非promise 类型的数据，
    // 状态为成功，返回值为对象的成功值resolved
    // [[PromiseStatus]]:"resolved"
    // [[PromiseValue]]:123
    // return 123;
    // 2、如果...是promise类型的数据
    // 此Promise对象的状态决定上面Promise对象p的状态
    // return new Promise((resolve,reject)=>{
    //  // resolve("ok"); // resolved
    //       reject("ok"); // rejected
    // });
    // 3、抛出错误
```

```
    // throw new Error("失败啦！");
    // 状态：rejected
    // value：失败啦！
    },reason => {
        console.error(reason);
})
// 链式调用
// then里面两个函数参数，可以只写一个
p.then(value=>{},reason=>{}).then(value=>{},reason=>{});
console.log(result);
```

2.14.4. Promise实践 - 读取多个文件

普通方法（回调地狱）：

```
// 1、引入 fs 模块
const fs = require("fs");
// 2、调用方法，读取文件
fs.readFile("resources/text.txt",(err,data1)=>{
    fs.readFile("resources/test1.txt",(err,data2)=>{
        fs.readFile("resources/test2.txt",(err,data3)=>{
            let result = data1 + data2 + data3;
            console.log(result);
        });
    });
});
```

Promise 实现：

```
// 1、引入 fs 模块
const fs = require("fs");

// 3、使用Promise实现
const p = new Promise((resolve,reject)=>{
    fs.readFile("resources/text.txt",(err,data)=>{
        resolve(data);
    });
});
p.then(value=>{
    return new Promise((resolve,reject)=>{
        fs.readFile("resources/test1.txt",(err,data)=>{
            resolve([value,data]);
        });
    })
}).then(value=>{
    return new Promise((resolve,reject)=>{
        fs.readFile("resources/test2.txt",(err,data)=>{
            // 存入数组
            value.push(data);
            resolve(value);
        });
    })
}).then(value=>{
    console.log(value.join("\r\n"));
})
```

2.14.5. Promise 对象 catch 方法

代码示例：

```
// Promise对象catch方法
const p = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        // 设置p对象的状态为失败，并设置失败的值
        reject("失败啦~! ");
    },1000);
})
// p.then(value=>{
// console.log(value);
// },reason=>{
// console.warn(reason);
// });
p.catch(reason=>{
    console.warn(reason);
});
```

2.15. ES6 - Set 集合

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**，集合实现了 iterator 接口，所以可以使用『扩展运算符』和『for...of...』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；

2.15.1. 基本使用

```
// Set集合
let s = new Set();
console.log(s,typeof s);
let s1 = new Set(["大哥","二哥","三哥","四哥","三哥"]);
console.log(s1); // 自动去重
// 1. size 返回集合的元素个数；
console.log(s1.size);
// 2. add 增加一个新元素，返回当前集合；
s1.add("大姐");
console.log(s1);
// 3. delete 删除元素，返回 boolean 值；
let result = s1.delete("三哥");
console.log(result);
console.log(s1);
// 4. has 检测集合中是否包含某个元素，返回 boolean 值；
let r1 = s1.has("二姐");
console.log(r1);
// 5. clear 清空集合，返回 undefined；
s1.clear();
console.log(s1);
```

2.15.2. Set 集合实践

代码示例：

```
// Set集合实践
let arr = [1,2,3,4,5,4,3,2,1];
//1. 数组去重
let s1 = [...new Set(arr)];
console.log(s1);
// 2. 交集
let arr2 = [3,4,5,6,5,4,3];
let result = [...new Set(arr)].filter(item => {
  let s2 = new Set(arr2)
  if (s2.has(item)) {
      return ture
  } else {
      return false
  }
})
// 交集实现代码简化
let result = [...new Set(arr)].filter(item=>new
Set(arr2).has(item));
console.log(result);
// 3. 并集
// ... 为扩展运算符，将数组转化为逗号分隔的序列
let union = [...new Set([...arr,...arr2])];
console.log(union);
// 4. 差集：比如集合1和集合2求差集，就是1里面有的，2里面没的
let result1 = [...new Set(arr)].filter(item=>!(new
Set(arr2).has(item)));
console.log(result1);
```

2.16. ES6 - Map 数据结构

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() 删除元素，返回 boolean 值；
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；

代码示例：

```
// Map集合
// 创建一个空 map
let m = new Map();
// 创建一个非空 map
let m2 = new Map([
['name','尚硅谷'],
```

```
['slogon','不断提高行业标准']
]);
// 1. size 返回 Map 的元素个数;
console.log(m2.size);
// 2. set 增加一个新元素，返回当前 Map;
m.set("皇帝","大哥");
m.set("丞相","二哥");
console.log(m);
// 3. get 返回键名对象的键值;
console.log(m.get("皇帝"));
// 4. has 检测 Map 中是否包含某个元素，返回 boolean 值;
console.log(m.has("皇帝"));
// 5. clear 清空集合，返回 undefined;
m.clear();
console.log(m);
```

2.17. ES6 - class 类

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;

2.17.1. class 初体验

```
// 手机 ES5写法
// function Phone(brand,price){
//      this.brand = brand;
//      this.price = price;
// }
// // 添加方法
// Phone.prototype.call = function(){
//      console.log("我可以打电话! ");
// }
// // 实例化对象
运行结果:
class静态成员:
代码实现:
// let HuaWei = new Phone("华为",5999);
// HuaWei.call();
// console.log(HuaWei);
// ES6写法
class Phone{
    // 构造方法，名字是固定的
    constructor(brand,price) {
        this.brand = brand;
        this.price = price;
```

```
    }
    // 打电话，方法必须使用该方式写
    call(){
        console.log("我可以打电话！");
    }
}
let HuaWei = new Phone("华为",5999);
HuaWei.call();
console.log(HuaWei);
```

2.17.2. class 静态成员

代码示例：

```
// class静态成员
// ES5写法
// function Phone(){}
// Phone.name = "手机";
// Phone.change = function(){
//      console.log("我可以改变世界！");
// }
// let nokia = new Phone();
// console.log(nokia.name); // undefined
// // nokia.change();
// // 报错：Uncaught TypeError: nokia.change is not a function
// Phone.prototype.color = "黑色";
// console.log(nokia.color); // 黑色
// console.log(Phone.name);
// Phone.change();
// 注意：实例对象和函数对象的属性是不相通的
// ES6写法
class Phone{
    // 静态属性
    static name = "手机";
    static change(){
        console.log("我可以改变世界！");
    }
}
let nokia = new Phone();
console.log(nokia.name);
console.log(Phone.name);  // => undefined
Phone.change();  // change is not defined

```

2.17.3. 继承实现

2.17.3.1. ES5 构造函数实现继承

```
// ES5构造函数继承
// 手机
function Phone(brand,price){
    this.brand = brand;
    this.price = price;
}
Phone.prototype.call = function(){
```

```
    console.log("我可以打电话！");
}
// 智能手机
function SmartPhone(brand,price,color,size){
    Phone.call(this,brand,price);
    this.color = color;
    this.size = size;
}
// 设置子级构造函数的原型
SmartPhone.prototype = new Phone;
SmartPhone.prototype.constructor = SmartPhone;
// 声明子类的方法
SmartPhone.prototype.photo = function(){
    console.log("我可以拍照！");
}
SmartPhone.prototype.game = function(){
    console.log("我可以玩游戏！");
}
const chuizi = new SmartPhone("锤子",2499,"黑色","5.5inch");
console.log(chuizi);
chuizi.call();
chuizi.photo();
chuizi.game();

```

2.17.3.2. ES6 class 类继承

代码示例：

```
// ES6class类继承
class Phone{
    constructor(brand,price) {
        this.brand = brand;
        this.price = price;
    }
    call(){
        console.log("我可以打电话！");
    }
}
class SmartPhone extends Phone{
    // 构造函数
    constructor(brand,price,color,size) {
        super(brand,price); // 调用父类构造函数
        this.color = color;
        this.size = size;
    }
    photo(){
        console.log("我可以拍照！");
    }
    game(){
        console.log("我可以玩游戏！");
    }
}
const chuizi = new SmartPhone("小米",1999,"黑色","5.15inch");
console.log(chuizi);
chuizi.call();
chuizi.photo();
```

```
chuizi.game();
```

2.17.4. 子类对父类方法的重写

```
// ES6class类继承
class Phone{
    constructor(brand,price) {
        this.brand = brand;
        this.price = price;
    }
    call(){
        console.log("我可以打电话！");
    }
}
class SmartPhone extends Phone{
    // 构造函数
    constructor(brand,price,color,size) {
        super(brand,price); // 调用父类构造函数
        this.color = color;
        this.size = size;
    }
    // 子类对父类方法重写
    // 直接写，直接覆盖
    // 注意：子类无法调用父类同名方法
    call(){
        console.log("我可以进行视频通话！");
    }
    photo(){
        console.log("我可以拍照！");
    }
    game(){
        console.log("我可以玩游戏！");
    }
}
const chuizi = new SmartPhone("小米",1999,"黑色","5.15inch");
console.log(chuizi);
chuizi.call();
chuizi.photo();
chuizi.game();
```

2.17.5. class 中的 getter 和 setter 设置

```
// class中的getter和setter设置
class Phone{
    get price(){
        console.log("价格属性被读取了！");
        // 返回值
        return 123;
    }
    set price(value){
        console.log("价格属性被修改了！");
    }
}
// 实例化对象
let s = new Phone();
console.log(s.price); // 返回值
```

```
s.price = 2999;
```

get 通常对对象的动态属性进行封装，所求的值是随着数据的变化而变化，这时候我们用 get

set 可以添加更多的控制和判断：比如设置的值是否是我们规定的值

2.18. ES6 - 数值扩展

1. Number.EPSILON

   Number.EPSILON 是 JavaScript 表示的最小精度；

   EPSILON 属性的值接近于 2.220446049250313080847263336181 6E-16。

2. 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。

3. 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。

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。

6. Math.sign

   判断一个数到底是正数、复数还是 0（返回1 -1 0）

代码示例：

```
// 数值扩展
// 0. Number.EPSILON 是 JavaScript 表示的最小精度
// EPSILON 属性的值接近于 2.220446049250313080847263336181 6E-16
// function equal(a, b){
//      return Math.abs(a-b) < Number.EPSILON;
// }
console.log("0、Number.EPSILON 是 JavaScript 表示的最小精度");
// 箭头函数简化写法
equal = (a, b) => Math.abs(a-b) < Number.EPSILON;
console.log(0.1 + 0.2);
console.log(0.1 + 0.2 === 0.3); // false
console.log(equal(0.1 + 0.2, 0.3)); // true

// 1. 二进制和八进制
console.log("1、二进制和八进制");
let b = 0b1010;
let o = 0o777;
let d = 100;
let x = 0xff;
console.log(x);

// 2. Number.isFinite 检测一个数值是否为有限数
console.log("2、Number.isFinite 检测一个数值是否为有限数");
console.log(Number.isFinite(100));
console.log(Number.isFinite(100/0));
```

```
console.log(Number.isFinite(Infinity));

// 3. Number.isNaN 检测一个数值是否为 NaN
console.log("3. Number.isNaN 检测一个数值是否为 NaN");
console.log(Number.isNaN(123));

// 4. Number.parseInt Number.parseFloat字符串转整数
console.log("4. Number.parseInt Number.parseFloat字符串转整数");
console.log(Number.parseInt('5211314love'));
console.log(Number.parseFloat('3.1415926神奇'));

// 5. Number.isInteger 判断一个数是否为整数
console.log("5. Number.isInteger 判断一个数是否为整数");
console.log(Number.isInteger(5));
console.log(Number.isInteger(2.5));

// 6. Math.trunc 将数字的小数部分抹掉
console.log("6. Math.trunc 将数字的小数部分抹掉 ");
console.log(Math.trunc(3.5));

// 7. Math.sign 判断一个数到底为正数 负数 还是零
console.log("7. Math.sign 判断一个数到底为正数 负数 还是零");
console.log(Math.sign(100));
console.log(Math.sign(0));
console.log(Math.sign(-20000));

```

2.19. ES6的的对象方法扩展

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 与 `===` 判断不相同）；
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（设置）、 getPrototypeOf（获取） 可以直接设置对象的原型；

代码示例：

```
// 对象扩展
// 1. Object.is 比较两个值是否严格相等，与『===』行为基本一致（+0 与 NaN）；
console.log(Object.is(120,120)); // ===
// 注意下面的区别
console.log(Object.is(NaN,NaN));
console.log(NaN === NaN);
// NaN与任何数值做===比较都是false，跟他自己也如此！
// 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；
const config1 = {
    host : "localhost",
    port : 3306,
    name : "root",
    pass : "root",
    test : "test" // 唯一存在
}
const config2 = {
    host : "http://zibo.com",
```

```
    port : 300300600,
    name : "root4444",
    pass : "root4444",
    test2 : "test2"
}
// 如果前边有后边没有会添加，如果前后都有，后面的会覆盖前面的
console.log(Object.assign(config1,config2));
// 3. __proto__、setPrototypeOf、 getPrototypeOf 可以直接设置对象的原型；
const school = {
    name : "尚硅谷"
}
const cities = {
    xiaoqu : ['北京','上海','深圳']
}
// 并不建议这么做
Object.setPrototypeOf(school,cities);
console.log(Object.getPrototypeOf(school));
console.log(school);
```

2.20. ES6 - 模块化

模块化是指将一个大的程序文件，拆分成许多小的文件，然后将小文件组合起来。

2.20.1. 模块化的好处

模块化的优势有以下几点：

1. 防止命名冲突
2. 代码复用
3. 高维护性

2.20.2. 模块化规范产品

ES6 之前的模块化规范有：

1. CommonJS => NodeJS、Browserify
2. AMD => requireJS
3. CMD => seaJS

2.20.3. ES6 模块化语法

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（导出模块）
- dimport 命令用于输入其他模块提供的功能（导入模块）；

简单使用：

```
// m.js 导出模块
export let school = "尚硅谷"

export function teach(){
    console.log("我们可以教你开发技术！")
}
```

```
<-- 模块化.html（导入和使用模块）-->
```

```
!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>模块化</title>
    </head>
    <body>
        <script type="module">
            // 引入m.js模块内容
            import * as m from "./js/m.js";
            console.log(m);
            console.log(m.school);
            m.teach();
        </script>
    </body>
</html>
```

2.20.4. ES6 模块暴露数据语法汇总

m.js（逐个导出模块）：

```
// 分别暴露（导出）
export let school = "尚硅谷";

export function teach(){
    console.log("我们可以教你开发技术！");
}
```

n.js（统一导出模块）：

```
// 统一暴露（导出）
let school = "尚硅谷";

function findJob(){
    console.log("我们可以帮你找到好工作！");
}

export {school,findJob}
```

o.js（默认导出模块）：

```
// 默认暴露（导出）
export default{
    school : "尚硅谷",
    change : function(){
        console.log("我们可以帮你改变人生！");
    }
}
```

模块化.html（引入和使用模块）：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
```

```
        <title>模块化</title>
    </head>
    <body>
        <script type="module">
            // 引入m.js模块内容
            import * as m from "./js/m.js";
            console.log(m);
            console.log(m.school);
            m.teach();
            // 引入n.js模块内容
            import * as n from "./js/n.js";
            console.log(n);
            console.log(n.school);
            n.findJob();
            // 引入o.js模块内容
            import * as o from "./js/o.js";
            console.log(o);
            // 注意这里调用方法的时候需要加上default
            console.log(o.default.school);
            o.default.change();
        </script>
    </body>
</html>
```

2.20.5. ES6 导入模块语法汇总

模块化.html：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>模块化</title>
    </head>
    <body>
        <script type="module">
            // 1. 通用方式
            // 引入m.js模块内容
            import * as m from "./js/m.js";
            console.log(m);
            console.log(m.school);
            m.teach();
            // 引入n.js模块内容
            import * as n from "./js/n.js";
            console.log(n);
            console.log(n.school);
            n.findJob();
            // 引入o.js模块内容
            import * as o from "./js/o.js";
            console.log(o);
            // 注意这里调用方法的时候需要加上default
            console.log(o.default.school);
            o.default.change();

            // 2. 解构赋值形式
            import {school,teach} from "./js/m.js";
            // 重名的可以使用别名
```

```
            import {school as xuexiao,findJob} from "./js/n.js";
            // 导入默认导出的模块，必须使用别名
            import {default as one} from "./js/o.js";
            // 直接可以使用
            console.log(school);
            teach();
            console.log(xuexiao);
            console.log(one);
            console.log(one.school);
            one.change();

            // 3. 简便形式，只支持默认导出
            import oh from "./js/o.js";
            console.log(oh);
            console.log(oh.school);
            oh.change();
        </script>
    </body>
</html>

```

2.20.6. 浏览器使用 ES6 模块化的方式二

将js语法整合到一个文件 app.js(入口文件) 中，在 html 文件中使用 script 标签引入 app.js：

```
// 通用方式
// 引入m.js模块内容
import * as m from "./m.js";
console.log(m);
console.log(m.school);
m.teach();
```

使用模块化的另一种方式.html：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>使用模块化的另一种方式</title>
    </head>
    <body>
            <script src="./js/app.js" type="module"></script>
    </body>
</html>

```

2.20.7. Babel 对 ES6 模块化代码转化

- Babel 是一个 JavaScript 编译器；
- Babel 能够将新的ES规范语法转换成ES5的语法；
- 因为不是所有的浏览器都支持最新的ES规范，所以，一般项目中都需要使用Babel进行转换；
- 步骤：使用Babel转换JS代码——打包成一个文件——使用时引入即可。

步骤：

第一步：安装工具：babel-cli（命令行工具）、babel-preset-env（ES转换工具）、browserify（打包工具，项目中使用的是webpack）。

第二步：初始化项目

```
npm init -y
```

第三步：安装

```
npm i babel-cli babel-preset-env browserify
```

第四步：使用 babel 转换

```
npx babel js(未转换的js 目录) -d dist/js(转化后的js目录) --presets=babel-preset-env
```

第五步：打包

```
npx browserify dist/js/app.js -o dist/bundle.js
```

第六步：在使用时引入 bundle.js

```
<script src="./js/bundle.js" type="module"></script>
```

转换前后对比：

转换前：

```
//分别暴露
export let school = '尚硅谷';
export function teach() {
    console.log("我们可以教给你开发技能");
}
```

转换后：

```
"use strict";
Object.defineProperty(exports, "__esModule", {
    value: true
});
exports.teach = teach;
//分别暴露
var school = exports.school = '尚硅谷';
function teach() {
    console.log("我们可以教给你开发技能");
}
```

2.20.8. ES6 模块化引入 npm 包

第一步：安装jQuery：

```
npm i jquery
```

第二步：在 app.js 使用 jQuery

```
//入口文件
//修改背景颜色为粉色
import $ from 'jquery';// 相当于const $ = require("jquery");
$('body').css('background','pink');
```

---

3. ECMAScript 7 新特性

3.1. Array.prototype.includes

include 方法用来检测数组中是否包含某个元素，返回布尔类型值。判断数组中是否包含某元素，语法：arr.includes(元素值)。

代码示例：

```
// includes
let arr = [1,2,3,4,5];
console.log(arr.includes(1)); // => true
```

3.2. 指数操作符（**）

在 ES7 中引入指数运算符「**」，用来实现幂运算，功能与 Math.pow 结果相同；

幂运算的简化写法，例如：2的10次方：2**10；

代码示例：

```
// 指数操作符
console.log(Math.pow(2,10))
console.log(2**10);
```

---

4. ECMAScript8 新特性

4.1. async 和 await

async 和 await 两种语法结合可以让异步代码像同步代码一样

4.1.1. async 函数

1. async 函数的返回值为 promise 对象。
2. promise 对象的结果有 async 函数执行的返回值决定。

代码示例：

```
// async函数：异步函数
async function fn(){
    // return 123; // 返回普通数据
    // 若报错，则返回的Promise对象也是错误的
    // throw new Error("出错啦！");
    // 若返回的是Promise对象，那么返回的结果就是Promise对象的结果
    return new Promise((resolve,reject)=>{
        // resolve("成功啦！");
        reject("失败啦！");
    })
}
const result = fn();
// console.log(result); // 返回的结果是一个Promise对象
// 调用then方法
result.then(value => {
    console.log(value);
},reason => {
    console.warn(reason);
});

```

4.1.2. await 表达式

1. await 必须写在 async 函数中；
2. await 右侧的表达式一般为 promise 对象；
3. await 返回的是 promise 成功的值；
4. await 的 promise 失败了，就会抛出异常，需要通过 try...catch 捕获处理。

代码示例：

```
// async函数 + await表达式：异步函数
// 创建Prmise对象
const p = new Promise((resolve,reject)=>{
    resolve("成功啦！");
})
async function fn(){
    // await 返回的是 promise 成功的值
    let result = await p;
    console.log(result); // 成功啦！
}
fn();
```

4.1.3. async 和 await 读取文件案例

```
// 导入模块
const fs = require("fs");

// 读取
function readText() {
    return new Promise((resolve, reject) => {
        fs.readFile("../resources/text.txt", (err, data) => {
            //如果失败
            if (err) reject(err);
            //如果成功
            resolve(data);
        })
    })
}

function readTest1() {
    return new Promise((resolve, reject) => {
        fs.readFile("../resources/test1.txt", (err, data) => {
            //如果失败
            if (err) reject(err);
            //如果成功
            resolve(data);
        })
    })
}

function readTest2() {
    return new Promise((resolve, reject) => {
        fs.readFile("../resources/test2.txt", (err, data) => {
            //如果失败
            if (err) reject(err);
            //如果成功
            resolve(data);
        })
    })
}

//声明一个 async 函数
async function main(){
    //获取为学内容
    let t0 = await readText();
    //获取插秧诗内容
    let t1 = await readTest1();
    // 获取观书有感
    let t2 = await readTest2();
    console.log(t0.toString());
    console.log(t1.toString());
    console.log(t2.toString());
}
main();

```

4.1.4. async 和 await 结合发送 AJAX 请求

```
// async 和 await 结合发送ajax请求
function sendAjax(url){
    return new Promise((resolve,reject)=>{
        // 1、创建对象
        const x = new XMLHttpRequest();
        // 2、初始化
        x.open("GET",url);
        // 3、发送
        x.send();
        // 4、事件绑定
        x.onreadystatechange = function(){
            if(x.readyState == 4){
                if(x.status>=200 && x.status<=299){
                    // 成功
                    resolve(x.response);
                }else{
                    // 失败
                    reject(x.status);
                }
            }
        }
    });
}
// 测试
// const result = sendAjax("https://api.apiopen.top/getJoke");
// result.then(value=>{
//      console.log(value);
// },reason=>{
//      console.warn(reason);
// })
// 使用async和await
async function main(){
    let result = await sendAjax("https://api.apiopen.top/getJoke");
    console.log(result);
}
main();
```

4.2. ES8 - 对象方法扩展

4.2.1. Object.values 和 Object.entries

1. Object.values() 方法返回一个给定对象的所有可枚举属性值的数组

```
// 对象方法扩展
let school = {
    name : "滴滴",
    age : 24,
    sex : "男"
}
// 获取对象所有的键
console.log(Object.keys(school));
// 获取对象所有的值
console.log(Object.values(school));
```

2. Object.entries() 方法返回一个给定对象自身可遍历属性 [key, value] 的数组。可以方便创建 Map 数据结构

```
// 对象方法扩展
let school = {
    name : "滴滴",
    age : 24,
    sex : "男"
}// 获取对象的entries
console.log(Object.entries(school));
// 创建map
const map = new Map(Object.entries(school));
console.log(map);
console.log(map.get("name"));
```

4.2.2. Object.getOwnPropertyDescriptors

Object.getOwnPropertyDescriptors() 方法：返回指定对象所有自身属性的描述对象。通过该属性可以进行对象深层次的克隆（各种属性的各种属性都相同）

```
// 返回指定对象所有自身属性的描述对象
console.log(Object.getOwnPropertyDescriptors(school));
// 参考内容：
const obj = Object.create(null,{
    name : {
        // 设置值
        value : "滴滴",
        // 属性特性
        writable : true,
        configuration : true,
        enumerable : true
    }
});

```

5. ECMAScript9 新特性

5.1. ES9 - Rest 参数与 spread 扩展运算符

Rest 参数与 spread 扩展运算符在 ES6 中已经引入，不过 ES6 中只针对于数组，在 ES9 中为对象提供了像数组一样的 rest 参数和扩展运算符。

5.1.1. Rest 参数（剩余参数）

```
// Rest参数与spread扩展运算符
// Rest 参数与 spread 扩展运算符在 ES6 中已经引入，
// 不过 ES6 中只针对于数组，在 ES9 中为对象提供了像
// 数组一样的 rest 参数和扩展运算符；
//rest 参数
function connect({
    host,
    port,
    ...user
}) {
    console.log(host);
    console.log(port);
    console.log(user);
}

connect({
    host: '127.0.0.1',
    port: 3306,
    username: 'root',
    password: 'root',
    type: 'master'
});
```

5.1.2. spread 扩展运算符

```
//对象合并
const skillOne = {
    q: '天音波'
}
const skillTwo = {
    w: '金钟罩'
}
const skillThree = {
    e: '天雷破'
}
const skillFour = {
    r: '猛龙摆尾',
    z: '马氏三角杀'
}
const mangseng = {
    ...skillOne,
    ...skillTwo,
    ...skillThree,
    ...skillFour
};
console.log(mangseng)
// ...skillOne => q: '天音波', w: '金钟罩'
```

5.2. ES9 - 正则扩展

5.2.1. 命名捕获分组

ES9 允许命名捕获组使用符号『?<捕获名>』,这样获取捕获结果可读性更强

```
// 正则扩展：命名捕获分组
// 声明一个字符串
let str = '<a href="http://www.baidu.com">滴滴</a>';
// 需求：提取url和标签内文本
// 之前的写法
const reg = /<a href="(.*)">(.*)<\/a>/;

// 执行
const result = reg.exec(str);
console.log(result);
// 结果是一个数组，第一个元素是所匹配的所有字符串
// 第二个元素是第一个(.*)匹配到的字符串
// 第三个元素是第二个(.*)匹配到的字符串
// 我们将此称之为捕获
console.log(result[1]);
console.log(result[2]);
// 命名捕获分组
const reg1 = /<a href="(?<url>.*)">(?<text>.*)<\/a>/;
const result1 = reg1.exec(str);
console.log(result1);
// 这里的结果多了一个groups
// groups:
// text:"滴滴"
// url:"http://www.baidu.com"
console.log(result1.groups.url);
console.log(result1.groups.text);
```

5.2.2. 反向断言

ES9 支持反向断言，通过对匹配结果前面的内容进行判断，对匹配进行筛选。

```
// 正则扩展：反向断言
// 字符串
let str = "JS5201314你知道么555啦啦啦";
// 需求：我们只想匹配到555
// 正向断言
const reg = /\d+(?=啦)/; // 前面是数字后面是啦
const result = reg.exec(str);
console.log(result);
// 反向断言
const reg1 = /(?<=么)\d+/; // 后面是数字前面是么
const result1 = reg.exec(str);
console.log(result1);

```

5.2.3. dotAll 模式

正则表达式中点 `.` 匹配除回车外的任何单字符，标记『s』改变这种行为，允许行终止符出现。

```
// 正则扩展：dotAll 模式
// dot就是. 元字符，表示除换行符之外的任意单个字符
let str = `
<ul>
    <li>
        <a>肖生克的救赎</a>
        <p>上映日期：1994-09-10</p>
    </li>
    <li>
        <a>阿甘正传</a>
        <p>上映日期：1994-07-06</p>
    </li>
</ul>
`;
// 需求：我们想要将其中的电影名称和对应上映时间提取出来，存到对象
// 之前的写法
// const reg = /<li>\s+<a>(.*?)<\/a>\s+<p>(.*?)<\/p>/;
// dotAll 模式
const reg = /<li>.*?<a>(.*?)<\/a>.*?<p>(.*?)<\/p>/gs;
// const result = reg.exec(str);
// console.log(result);
let result;
let data = [];
while(result = reg.exec(str)){
    console.log(result);
    data.push({title:result[1],time:result[2]});
}
console.log(data);
```

---

6. ECMAScript10 新特性

概述：

1. Object.fromEntries
    - 将二维数组或者map转换成对象；
2. trimStart 和 trimEnd
    - 去除字符串前后的空白字符；
3. Array.prototype.flat 与 flatMap
    - 将多维数组降维；
4. Symbol.prototype.description
    - 获取Symbol的字符串描述；

6.1. Object.fromEntries

将二维数组或者map转换成对象；

之前学的Object.entries是将对象转换成二维数组。

代码示例：

```
// Object.fromEntries：将二维数组或者map转换成对象
// 之前学的Object.entries是将对象转换成二维数组
// 此方法接收的是一个二维数组，或者是一个map集合
// 二维数组
const result = Object.fromEntries([
    ["name","訾博"],
    ["age",24],
]);
console.log(result);
const m = new Map();
m.set("name","滴滴");
m.set("age",24);
const result1 = Object.fromEntries(m);
console.log(result1);  // 转换成对象

```

6.2. trimStart 和 trimEnd

去掉字符串前后的空白字符；

代码示例：

```
// trimStart 和 trimEnd
let str = " zibo ";
console.log(str.trimLeft());
console.log(str.trimRight());
console.log(str.trimStart());
console.log(str.trimEnd());
```

6.3. Array.prototype.flat 与 flatMap

将多维数组转换成低维数组；

代码示例：

```
// Array.prototype.flat 与 flatMap
// flat 单词意思为 `平`
// 将多维数组转换成低维数组
// 将二维数组转换成一维数组
const arr = [1,2,3,[4,5],6,7];
console.log(arr.flat());
// 将三维数组转换成二维数组
const arr2 = [1,2,3,[4,5,[6,7]],8,9];
console.log(arr2.flat());
// 将三维数组转换成一维数组
```

```
// 参数为深度，是一个数字，默认值是 1
console.log(arr2.flat(2));
// flatMap
const arr3 = [1,2,3,4,5];
const result = arr3.map(item => [item * 10]);
console.log(result);
// 如果 map 方法返回的是一个数组，则可以用 flatMap 转成一个数组，相当于两个操作的结合（map 和 flat）
const result1 = arr3.flatMap(item => [item * 10]);
console.log(result1);

```

6.4. Symbol.prototype.description

获取Symbol的描述字符串。

```
// Symbol.prototype.description
// 获取Symbol的描述字符串
// 创建Symbol
let s = Symbol("砥博");
console.log(s.description)
```

---

7. ECMAScript11 新特性

概述：

1. 类的私有属性
    - 私有属性外部不可访问直接；
2. Promise.allSettled
    - 获取多个promise执行的结果集；
3. String.prototype.matchAll
    - 用来得到正则批量匹配的结果；
4. 可选链操作符
    - 简化对象存在的判断逻辑；
5. 动态 import 导入
    - 动态导入模块，什么时候使用什么时候导入；
6. BigInt
    - 大整型；
7. globalThis 对象
    - 始终指向全局对象window；

7.1. ES11 - 类的私有属性

私有属性外部不能直接访问。

```
// 类的私有属性
class Person{
    // 公有属性
    name;
    // 私有属性
    #age;
    #weight;
    // 构造方法
    constructor(name, age, weight){
        this.name = name;
        this.#age = age;
        this.#weight = weight;
    }
    intro(){
        console.log(this.name);
        console.log(this.#age);
        console.log(this.#weight);
    }
}
// 实例化
const girl = new Person("小兰",18,"90kg");
console.log(girl);
// 公有属性的访问
console.log(girl.name);
// 私有属性的访问
console.log(girl.age); // undefined
// 报错Private field '#age' must be declared in an enclosing class
// console.log(girl.#age);
girl.intro();
```

7.2. Promise.allSettled([promises])

- 接收一个Promise 数组;
- 返回结果为一个 Primise 对象;
- 返回结果永远为成功状态，成功的值是传入的每一个 promise 对象的状态和结果组成的数组。

代码示例:

```
// Promise.allSettled
// 获取多个promise执行的结果集
// 声明两个promise对象
const p1 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        resolve("商品数据--1");
    },1000);
});

const p2 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        reject("失败啦");
```

```
    },1000);
});
// 调用Promise.allSettled方法
const result = Promise.allSettled([p1,p2]);
console.log(result);
const result1 = Promise.all([p1,p2]); // 注意区别，传入的promise 都成功才成功，只要有一个失败就失败，失败的值就是失败的promise 的值
console.log(result1);

```

Promise.allSettled 和 Promise.all 都用来做一些批量异步任务的处理。

- 如果每一个异步任务都想知道结果就用 Promise.allSettled;
- 如果异步任务要求每一个都成功才能继续往下执行那就用 Promise.all。

7.3. String.prototype.matchAll

用来获取正则批量匹配的结果。

代码示例:

```
// String.prototype.matchAll
// 用来得到正则批量匹配的结果
let str = `<ul>
    <li>
        <a>肖生克的救赎</a>
        <p>上映日期: 1994-09-10</p>
    </li>
    <li>
        <a>阿甘正传</a>
        <p>上映日期: 1994-07-06</p>
    </li>
</ul>`;
// 正则
const reg = /<li>.*?<a>(.*?)<\/a>.*?<p>(.*?)<\/p>/sg;
const result = str.matchAll(reg);
// 返回的是可迭代对象，可用扩展运算符展开
// console.log(...result);
// 使用for...of...遍历
for(let v of result){
    console.log(v);
}
```

7.4. ES11 - 可选链操作符

如果存在则往下走，省略对对象是否传入的层层判断。

```
// 可选链操作符
// ?.
function main(config){
    // 传统写法
    // const dbHost = config && config.db && config.db.host;
    // 可选链操作符写法

```

```
    const dbHost = config?.db?.host;
    console.log(dbHost);
}
main({
    db:{
        host:"192.168.1.100",
        username:"root"
    },
    cache:{
        host:"192.168.1.200",
        username:"admin"
    }
});
```

7.5. ES11 - 动态 import 导入

- 动态导入模块，什么时候使用时候导入。
- import() 函数，返回结果为 promise 对象，该对象成功的值就是被导入模块中暴露出的对象

hello.js：

```
export function hello(){
    alert('Hello');
}
```

app.js：

```
// import * as m1 from "./hello.js"; // 传统静态导入
//获取元素
const btn = document.getElementById('btn');
btn.onclick = function(){
    import('./hello.js').then(module => {
        module.hello();
    });
}
```

动态import加载.html：

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <title>动态 import </title>
    </head>
    <body>
        <button id="btn">点击</button>
        <script src="app.js" type="module"></script>
    </body>
</html>

```

7.6. ES11 - BigInt 类型

- 大整数

```
// BigInt
// 大整型
let n = 100n;
console.log(n,typeof(n));

// 函数：普通整型转大整型
let m = 123;
console.log(BigInt(m));

// 用于更大数值的运算
let max = Number.MAX_SAFE_INTEGER;
console.log(max);
console.log(max+1);
console.log(max+2); // 出错了 到达最大值，没法再往上加了
console.log(BigInt(max));
console.log(BigInt(max)+BigInt(1));
console.log(BigInt(max)+BigInt(2));
```

7.7. ES11 - 绝对全局对象 globalThis

- 始终指向全局对象window。

```
// globalThis 对象 : 始终指向全局对象window
console.log(globalThis);
```