ES6-ES11学习笔记

ES6

一、let var const

1. let

1. 变量不能重复声明

```
    // let star = '罗志祥';
    // let star = '小猪'; //报错
```

2. 块级作用域

```
    {
      let girl = '周扬青';
    }

    // console.log(girl); // caught ReferenceError: girl is not defined
```

3. 不存在变量提升

```
    // console.log(song); //caught ReferenceError: Cannot access 'song' before
initialization
    // let song = '恋爱达人';
```

4. 不影响作用链

```
    {
      let school = '尚硅谷';
      function fn () {
        console.log(school);

      }
      fn();
    }
```

5. 实践案例

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>点击 DIV 换色</title>
    <link crossorigin="anonymous" href="https://cdn.bootcss.com/twitter-
bootstrap/3.3.7/css/bootstrap.min.css"
        rel="stylesheet">
    <style>
        .item {
```

```
            width: 100px;
            height: 50px;
            border: solid 1px rgb(42, 156, 156);
            float: left;
            margin-right: 10px;
        }
    </style>
</head>

<body>
    <div class="container">
        <h2 class="page-header">点击切换颜色</h2>
        <div class="item"></div>
        <div class="item"></div>
        <div class="item"></div>
    </div>
    <script>
        //获取div元素对象
        let items = document.getElementsByClassName('item');

        //遍历并绑定事件
        for(let i = 0;i<items.length;i++){
            items[i].onclick = function(){
                //修改当前元素的背景颜色
                // this.style.background = 'pink';
                items[i].style.background = 'pink';
            }
        }

    </script>
</body>

</html>
```

2. const

1. 一定要赋初始值，否则报错

```
const A; // 报错
```

2. 一般常量使用大写(潜规则)

```
cosnt B = 100;
```

3. 常量不能修改

```
const NAME = '小康';
NAME = '李白'; // 报错
```

4. 和 `let` 一样有块级作用域
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','MXLG','Ming','Letme'];
// TEAM.push('Meiko');

console.log(TEAM);
```

二、模板字符串

1. 声明

```
let str = `我也是一个字符串哦!`;
console.log(str, typeof str);
```

2. 内容中可以直接出现换行符

```
    let str = `<ul>
                <li>沈腾</li>
                <li>玛丽</li>
                <li>魏翔</li>
                <li>艾伦</li>
                </ul>`;
```

3.变量拼接

```
let lovest = '魏翔';
let out = `${lovest}是我心目中最搞笑的演员!!`;
console.log(out);
```

三、简化对象的写法

> 说明：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。

```
// 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
let name = '小康';
let change = function () {
  console.log('我们可以改变你! ');
}

const school = {
  name,
  change,
  improve () {
    console.log('我们可以提高你的技能');
  }
}

console.log(school);
```

四、变量的解构赋值

> 说明：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。

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, liu, zhao, song);
```

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');
  }
}

// 解构赋值
// let {name, age, xiaopin} = zhao;
```

五、箭头函数

> 说明：ES6 允许使用 箭头 (`=>`) 定义函数。
>
> 1. 箭头函数适合与 this 无关的回调 如定时器、数组的方法回调。
> 2. 箭头函数不适合与 this 有关的回调 如事件回调、对象的方法。

1. this指向问题

- this 是静态的 this 始终指向函数声明时所在作用域下的 this 的值

```
function getName () {
   console.log(this.name);
 }

 let getName2 = () => {
   console.log(this.name);

 }

// 设置 window 对象的 name 属性
window.name = '小康';
const school = {
  name: 'B-website'
}

// 直接调用
getName();
getName2();

// call 方法调用
getName.call(school); // B-website
getName2.call(school); // 小康
```

2. 不能作为构造实例化对象

```
let Person = (name, age) => {
  this.name = name;
  this.age = age;
}

// let me = new Person('xiao', 30);
// console.log(me); // 报错
```

3. 不能使用 arguments 变量

```
let fn = () => {
  console.log(arguments); // 报错
}
fn(1, 2, 3);
```

4. 箭头函数的简写

- 省略小括号, 当形参有且只有一个的时候

```
let add = n => {
  return n + n;
}

console.log(add(9));
```

- 省略花括号, 当代码体只有一条语句的时候, 此时 return 必须省略， 而且语句的执行结果就是函数的返回值

```
let pow = n => n * n;
console.log(pow(8));
```

5. 实践案例

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>箭头函数实践</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 () {
      // 保存 this 的值
      // let_this = this;
      // 定时器
      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);

    // 箭头函数适合与 this 无关的回调 如定时器、数组的方法回调
    // 箭头函数不适合与 this 有关的回调 如事件回调、对象的方法


  </script>
</body>

</html>
```

六、参数默认值

1. 形参初始值 具有默认值的的参数, 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);
```

2. 与解构赋值结合

```
function connect ({ host, username, password, port }) {
  console.log(host);
  console.log(username);
  console.log(password);
  console.log(port);
}

connect({
  host: 'codeslive.top',
  username: 'root',
  password: 'root',
  port: 3306
});
```

七、rest 参数

> 说明：ES6 引入 rest 参数, 用于获取函数的实参, 用来代替 arguments。

1. 获取实参的方式

```
function date2 (...args) {
  console.log(args); // filter some every map
}
date2('小白', '小红', '小黑');
```

2. 参数必须要放到参数最后

```
function fn (a, b, ...args) {
  console.log(a);
  console.log(b);
  console.log(args);
}
fn(1, 2, 3, 4, 5, 6, 7);
```

八、扩展运算符

> 说明：扩展运算符能将『数组』转换为逗号分隔的『参数序列』

1. 基本使用

```
// 声明一个数组
const boys = ['猪猪侠', '熊大', '熊二'];
// => '猪猪侠','熊大', '熊二'

// 声明一个函数
function chunwan () {
  console.log(arguments);

}
chunwan(...boys); // '猪猪侠','熊大', '熊二'
```

2. 应用场景

- 数组的合并

```
const food = ['包子', '馒头', '豆浆'];
const goods = ['米饭', '白粥'];
const foodGoods = [...food, ...goods];
console.log(foodGoods);
```

- 数组的克隆

```
const phone = ['meizu', 'xiaomi', 'huawei'];
const phoneT = [...phone];
console.log(phoneT);
```

- 将伪数组转为真正的数组

```
const divs = document.querySelectorAll('div');
const divArr = [...divs];
console.log(divArr); // arguments
```

九、Symbol

1. 基本使用

```
        //创建Symbol
        let s = Symbol();
        // console.log(s, typeof s);
        let s2 = Symbol('尚硅谷');
        let s3 = Symbol('尚硅谷');
        //Symbol.for 创建
        let s4 = Symbol.for('尚硅谷');
        let s5 = Symbol.for('尚硅谷');

        //不能与其他数据进行运算
        //    let result = s + 100;
        //    let result = s > 100;
        //    let result = s + s;

        // USONB  you are so niubility
        // u  undefined
        // s  string  symbol
        // o  object
        // n  null number
        // b  boolean
```

2.Symbol创建对象属性

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

// 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);

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);
```

十、迭代器

1. 基本使用

```
// 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()); // 唐僧
console.log(iterator.next()); // 孙悟空
console.log(iterator.next()); // 猪八戒
console.log(iterator.next()); // 沙僧
console.log(iterator.next()); // undefined
```

2. 自定义遍历对象

```
// 声明一个对象
const banji = {
  name: '终极一班',
  stus: [
    'xiaoming',
    'xiaobao',
```

```
        'xiaokang',
        '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);

    }
```

十一、生成器

> 说明：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

1. 基本使用

```
    function* gen () {
      console.log(111);
      yield '一只没有耳朵';
      console.log(222);
      yield '一只没有尾巴';
      console.log(333);
      yield '真奇怪';
      console.log(444);
    }

    let interator = gen();
    console.log(interator.next());
    console.log(interator.next());
    console.log(interator.next());
    console.log(interator.next());
```

2. 生成器函数参数

```
function* gen (arg) {
  console.log(arg); // BBB
  let one = yield 111;
  console.log(one);
  let two = yield 222;
  console.log(two);
  let three = yield 333;
  console.log(three);

}

// 执行获取迭代器对象
let iterator = gen('AAA');
console.log(iterator.next()); // AAA

// next 方法可以传入实参
console.log(iterator.next('BBB'));
console.log(iterator.next('CCC'));
console.log(iterator.next('DDD'));
```

3.生成器函数实例

- 实例1

```
// 异步编程 文件操作 网络操作 (ajax, request) 数据库操作
// 1s 后控制台输出 111  2s后输出 222  3s后输出 333

// 回调地狱
// setTimeout(() => {
//   console.log(111);
//   setTimeout(() => {
//     console.log(222);
//     setTimeout(() => {
//       console.log(333);
//     }, 3000);
//   }, 2000);
// }, 1000);

function one () {
  setTimeout(() => {
    console.log(111);
    iterator.next();
  }, 1000);
}

function two () {
  setTimeout(() => {
    console.log(222);
    iterator.next();
  }, 2000);
}

function three () {
  setTimeout(() => {
```

```
    console.log(333);
    iterator.next();
  }, 3000);
}

function* gen () {
  yield one();
  yield two();
  yield three();
}

// 调用生成器函数
let iterator = gen();
iterator.next();
```

- 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';
    // 调用 next 方法, 并且将数据传入
    iterator.next(data);
  }, 1000);
}

function getOrders () {
  setTimeout(() => {
    let data = '订单数据';
    // 调用 next 方法, 并且将数据传入
    iterator.next(data);
  }, 1000);
}

function getGoods () {
  setTimeout(() => {
    let data = '商品数据';
    // 调用 next 方法, 并且将数据传入
    iterator.next(data);
  }, 1000);
}


function* gen () {
  let users = yield getUsers();
  console.log(users);
  let orders = yield getOrders();
  console.log(orders);
  let goods = yield getGoods();
  console.log(goods);
}

// 调用生成器函数
let iterator = gen();
iterator.next();
```

十二、Promise

1. 基本语法

```
        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("出错啦!");
            }, 1000)
        });

        // p.then(function(value){}, function(reason){
        //     console.error(reason);
        // });

        p.catch(function(reason){
            console.warn(reason);
        });
```

2. 封装读取文件

```
//1. 引入 fs 模块
const fs = require('fs');

//2. 调用方法读取文件
// fs.readFile('./resources/为学.md', (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());
// });

//3. 使用 Promise 封装
const p = new Promise(function(resolve, reject){
    fs.readFile("./resources/为学.mda", (err, data)=>{
        //判断如果失败
        if(err) reject(err);
        //如果成功
        resolve(data);
    });
});

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

3. 封装AJAX

```
        // 接口地址: https://api.apiopen.top/getJoke
        const p = new Promise((resolve, reject) => {
            //1. 创建对象
            const xhr = new XMLHttpRequest();
```

```
        //2. 初始化
        xhr.open("GET", "https://api.apiopen.top/getJ");

        //3. 发送
        xhr.send();

        //4. 绑定事件, 处理响应结果
        xhr.onreadystatechange = function () {
            //判断
            if (xhr.readyState === 4) {
                //判断响应状态码 200-299
                if (xhr.status >= 200 && xhr.status < 300) {
                    //表示成功
                    resolve(xhr.response);
                } else {
                    //如果失败
                    reject(xhr.status);
                }
            }
        }
    })

    //指定回调
    p.then(function(value){
        console.log(value);
    }, function(reason){
        console.error(reason);
    });
```

4. then方法

```
    //创建 promise 对象
    const p = new Promise((resolve, reject)=>{
        setTimeout(()=>{
            resolve('用户数据');
            // reject('出错啦');
        }, 1000)
    });

    //调用 then 方法  then方法的返回结果是 Promise 对象, 对象状态由回调函数的执行结果决定
    //1. 如果回调函数中返回的结果是 非 promise 类型的属性, 状态为成功, 返回值为对象的成功的值

    // const result = p.then(value => {
    //     console.log(value);
    //     //1. 非 promise 类型的属性
    //     // return 'iloveyou';
    //     //2. 是 promise 对象
    //     // return new Promise((resolve, reject)=>{
    //     //     // resolve('ok');
    //     //     reject('error');
    //     // });
    //     //3. 抛出错误
    //     // throw new Error('出错啦!');
    //     throw '出错啦!';
    // }, reason=>{
    //     console.warn(reason);
```

```
        // });

        //链式调用
        p.then(value=>{

        }).then(value=>{

        });
```

5. 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("出错啦!");
            }, 1000)
        });

        // p.then(function(value){}, function(reason){
        //     console.error(reason);
        // });

        p.catch(function(reason){
            console.warn(reason);
        });
```

6. promise实践

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

// fs.readFile('./resources/为学.md', (err, data1)=>{
//     fs.readFile('./resources/插秧诗.md', (err, data2)=>{
//         fs.readFile('./resources/观书有感.md', (err, data3)=>{
//             let result = data1 + '\r\n' +data2  +'\r\n'+ data3;
//             console.log(result);
//         });
//     });
// });

//使用 promise 实现
const p = new Promise((resolve, reject) => {
    fs.readFile("./resources/为学.md", (err, data) => {
        resolve(data);
    });
});

p.then(value => {
    return new Promise((resolve, reject) => {
        fs.readFile("./resources/插秧诗.md", (err, data) => {
            resolve([value, data]);
        });
    });
}).then(value => {
    return new Promise((resolve, reject) => {
        fs.readFile("./resources/观书有感.md", (err, data) => {
```

```
            //压入
            value.push(data);
            resolve(value);
        });
    })
}).then(value => {
    console.log(value.join('\r\n'));
});
```

十三、集合

1. Set

- 声明一个Set

```
        let s = new Set();
        let s2 = new Set(['大事儿','小事儿','好事儿','坏事儿','小事儿']);
```

- Set API

    1. 元素个数

    ```
    console.log(s2.size);
    ```

    2. 添加新的元素

    ```
    s2.add('喜事儿');
    ```

    3. 删除元素

    ```
    s2.delete('坏事儿');
    ```

    4. 检测元素是否存在，存在返回 `true` ，不存在返回 `false`

    ```
    console.log(s2.has('糟心事'));
    ```

    5. 清空

    ```
    s2.clear();
    ```

- 集合实践

    1. 数组去重

    ```
    let arr = [1, 1, 34, 56, 66, 78, 100];
    // 数组去重
    let result = [...new Set(arr)];
    console.log(result);
    ```

    2. 交集

    ```
    let arr = [1, 1, 34, 56, 66, 78, 100];
    let arr2 = [1, 1, 1, 4, , 56, 78, 100, 9, 8];
    // 交集
    let result = [...new Set(arr)].filter(item => {
      let s2 = new Set(arr2);
      if (s2.has(item)) {
        return true;
    ```

```
    } else {
      return false;
    }
  });

  console.log(result);
  // 链式写法
  let result = [...new Set(arr)].filter(item => new Set(arr2).has(item));
```

2. Map

- 基本使用

```
// 声明 Map
let m = new Map();
// 添加元素
m.set('name', '小康');
m.set('change', function () {
  console.log('我们可以改变你！！');
});
let key = {
  home: 'LUODING'
};
m.set(key, ['广州', '深圳']);

// size
console.log(m);
```

- Map API

  1. 元素个数

```
console.log(m.size);
```

  2. 删除元素

```
m.delete('name');
```

  3. 获取元素

```
console.log(m.get('change'));
console.log(m.get(key));
```

  4. 清空

```
m.clear();
```

十四、类

1. class类

```
// 手机
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
let Huawei = new Phone('华为', 5999);
Huawei.call();
console.log(Huawei);

// class
class Showji {
  // 构造方法 名字不能修改
  constructor(brand, price) {
    this.brand = brand;
    this.price = price;
  }

  // 方法必须使用该语法, 不能使用 ES5 的对象完整形式
  call () {
    console.log('我可以打电话!!');
  }
}

let onePlus = new Showji('1+', 1999);
console.log(onePlus);
```

2. class类的静态成员

```
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); // undefined
console.log(Phone.name); // 手机
```

3. class类继承

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

```
  }
}

class SmartPhone extends Phone {
  // 构造方法
  constructor(brand, price, color, size) {
    super(brand, price);
    this.color = color;
    this.size = size;
  }

  photo () {
    console.log('拍照');

  }
  playGame () {
    console.log('玩游戏');

  }
  call () {
    console.log('我可以进行视频通话');
  }
}

const xiaomi = new SmartPhone('小米', 799, '黑色', '4.7inch');
xiaomi.call(); // 我可以进行视频通话
xiaomi.photo(); // 拍照
xiaomi.playGame(); // 玩游戏
```

4. class的get和set

```
class Phone {
  get price () {
    console.log('价格属性被获取了');
    return 'iloveyou';
  }

  set price (newVal) {
    console.log('价格属性被修改了');
  }
}

// 实例化对象
let s = new Phone();
console.log(s.price);

s.price = 'free';
```

十五、数值扩展

```
// 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) {
  if (Math.abs(a - b) < Number.EPSILON) {
```

```
    return true;
  } else {
    return false;
  }
}

// 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
// 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(Number.isFinite(100));
// console.log(Number.isFinite(100 / 0));
// console.log(Number.isFinite(Infinity));

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

// 4. Number.parseInt Number.parseFloat字符串转整数
// console.log(Number.parseInt(`521love`)); // 521
// console.log(Number.parseFloat(`12345加油`)); // 12345

// 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(Math.trunc(3.5));

// 7. Math.sign 判断一个数到底是正数 负数 还是零
console.log(Math.sign(100)); // 1
console.log(Math.sign(0)); // 0
console.log(Math.sign(-20000)); // -1
```

十六、对象方法扩展

```
// 1. Object.is 判断两个值是否完全相等
// console.log(Object.is(120, 120)); // true
// console.log(Object.is(NaN, NaN)); // true
// console.log(NaN === NaN); // false

// 2. Object.assign 对象的合并
const config1 = {
  host: 'localhost',
  port: 3306,
  name: 'root',
  pass: 'root',
  test: 'test'
};
```

```
    const config2 = {
      host: 'http://codeslive.top',
      port: '33060',
      name: 'xiaokang',
      pass: 'iloveyou',
      test2: 'test2'
    }

    console.log(Object.assign(config1, config2));

    // 3. Object.setPrototypeOf 设置原型对象 Object.getPrototypeof
    const school = {
      name: '小康'
    }

    const cities = {
      xiaoqu: ['beijing', 'shanghai', 'shenzhen']
    }

    Object.setPrototypeOf(school, cities);
    console.log(Object.getPrototypeOf(school));
    console.log(school);
```

十七、模块化

1. 文件引入

```
        //1. 通用的导入方式
        //引入 m1.js 模块内容
        // import * as m1 from "./src/js/m1.js";
        // //引入 m2.js 模块内容
        // import * as m2 from "./src/js/m2.js";
        // //引入 m3.js
        // import * as m3 from "./src/js/m3.js";

        //2. 解构赋值形式
        // import {school, teach} from "./src/js/m1.js";
        // import {school as guigu, findJob} from "./src/js/m2.js";
        // import {default as m3} from "./src/js/m3.js";

        //3. 简便形式  针对默认暴露
        // import m3 from "./src/js/m3.js";
        // console.log(m3);
```

2. babel转换

```
<body>
    <!--
        1. 安装工具 npm i babel-cli babel-preset-env browserify(webpack) -D
        2. 编译 npx babel src/js -d dist/js --presets=babel-preset-env
        3. 打包 npx browserify dist/js/app.js -o dist/bundle.js
     -->
     <script src="dist/bundle.js"></script>
</body>
```

ES7

一、includes

> 说明：`includes()` 是 ES7 中新增的数组方法，用于判断数组中是否包含某个值，并返回一个布尔值。

```
// includes
const mingzhu = ['西游记', '红楼梦', '三国演义', '水浒传'];

// 判断
console.log(mingzhu.includes('西游记')); // true
console.log(mingzhu.includes('金瓶梅')); // false
```

二、双星号（**）

> 说明：在 ES7 中，`双星号（**）`是指数运算符。它可以用于计算某个数的幂次方

```
console.log(2 ** 10); //1024
```

ES8

一、async函数

```
// async 函数
async function fn () {
  // 返回一个字符串
  // return 'xiaokang';

  // 返回的结果不是一个 Promise 类型的对象, 返回的结果就是成功 Promise 对象

  // return;  // fulfilled undefined

  // 抛出错误, 返回结果是一个失败的 Promise
  // throw new Error('出错了'); // 报错并返回失败的promise对象

  // 返回的结果如果是一个 Promise 对象
  return new Promise((resolve, reject) => {
    resolve('成功的数据');
  });


}

const result = fn();
console.log(result);

// 调用 then 方法
result.then(value => {
```

```
    console.log(value);
  }, reason => {
    console.warn(reason);
  });
```

二、await函数

```
  // 创建 promise 对象
  const p = new Promise((resolve, reject) => {
    resolve("用户数据");
  });

  // await 要放在 async 函数中
  async function main () {
    try {
      let result = await p;
      console.log(result);

    } catch (e) {
      console.log(e);
    }
  }

  // 调用函数
  main();
```

三、async和await结合读取文件

```
// 1. 引入 fs 模块
const fs = require('fs');
const { resolve } = require('path');

// 读取 为学
function readWeiXue () {
  return new Promise((resolve, reject) => {
    fs.readFile('./resources/为学.md', (err, data) => {
      // 如果失败
      if (err) reject(err);
      // 如果成功
      resolve(data);
    });
  });
}


function readChaYangShi () {
  return new Promise((resolve, reject) => {
    fs.readFile('./resources/插秧诗.md', (err, data) => {
      // 如果失败
      if (err) reject(err);
      // 如果成功
      resolve(data);
    });
  });
}
```

```
function readGuanShu () {
  return new Promise((resolve, reject) => {
    fs.readFile('./resources/观书有感.md', (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 weixue = await readWeiXue();
  // 获取插秧诗的内容
  let chayang = await readChaYangShi();
  // 获取观书有感
  let guanshu = await readGuanShu();

  console.log(weixue.toString());
  console.log(chayang.toString());
  console.log(guanshu.toString());

}

// 调用函数
main();
```

四、async与await封装AJAX

```
  <script>
    // 发送 AJAX 请求, 返回的结果是 Promise 对象
    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 < 300) {
              // 成功
              resolve(x.response);
            } else {
              // 如果失败
```

```
            reject(x.status);
          }
        }
      }
    });
}

// async 与 await 测试
async function main () {
  // 发送 AJAX 请求
  let result = await sendAJAX('https://b.codeslive.top/api/public/book/findFav5');
  console.log(result);
}

main();
```

五、对象方法扩展

```
// 声明对象
const school = {
  name: 'xiaokang',
  cities: ['beijing', 'shanghai', 'shenzhen'],
  xueke: ['前端', 'Java', '大数据', '运维']
};

// 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));
// entries
console.log(Object.entries(school));
// 创建 Map
const m = new Map(Object.entries(school));
console.log(m.get('cities'));

// 对象属性描述对象
console.log(Object.getOwnPropertyDescriptors(school));
```

ES9

一、对象展开

```
/*
  Rest 参数 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'
});

// 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'
}

const mangseng = { ...skillOne, skillTwo, skillThree, skillFour };
console.log(mangseng);
```

二、正则命名分组

```
// 声明一个字符串
// let str = `<a href="https://codeslive.top">小康</a>`;
// // 提前 url 与 标签文本
// const reg = /<a href="(.*)">(.*)<\/a>/;
// // 执行
// const result = reg.exec(str);
// console.log(result);

let str = `<a href="https://codeslive.top">小康</a>`;
// 分组命名
const reg = /<a href="(?<url>.*)">(?<text>.*)<\/a>/;
const result = reg.exec(str);
console.log(result.groups.url);
console.log(result.groups.text);
```

三、正则断言

```
// 声明字符串
let str = 'JS5211314你知道么555啦啦啦';
// 正向断言
const reg = /\d+(?=啦)/;
const result = reg.exec(str);
console.log(result);

// 反向断言
const reg2 = /(?<=么)\d+/;
const result2 = reg2.exec(str);
console.log(result2);
```

四、正则dotAll模式

```
    //dot   .  元字符  除换行符以外的任意单个字符
    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>`;
    //声明正则
    // const reg = /<li>\s+<a>(.*?)<\/a>\s+<p>(.*?)<\/p>/;
    const reg = /<li>.*?<a>(.*?)<\/a>.*?<p>(.*?)<\/p>/gs;
    //执行匹配
    // const result = reg.exec(str);
    let result;
    let data = [];
    while(result = reg.exec(str)){
        data.push({title: result[1], time: result[2]});
    }
    //输出结果
    console.log(data);
```

ES10

一、Object.fromEntries

```
// 二维数组 Object.fromEntries：数组转对象
const result = Object.fromEntries([
  ['name', 'xiaokang'],
  ['xueke', 'java,web']
]);

console.log(result);

// 对象转数组 Object.entries ES8
```

```
const arr = Object.entries({
  name: 'xiaokang'
});

console.log(arr); // 对象转数字
```

二、trimStart与trimEnd

```
// trim
let str = '    iloveyou      ';
console.log(str);
console.log(str.trimStart()); // 去除开始的空格
console.log(str.trimEnd()); // 去除结尾的空格
```

三、Array.prototype.flat与flatMap

```
// flat 平
// 将多维数组转换为低维数组
const arr = [1, 2, [5, 6]];
const arr2 = [1, 2, 3, 4, [7, 8, 9]];
// 参数为深度 是一个数字
console.log(arr.flat(2));

// flatMap
const arr3 = [1, 2, 3, 4];
const result = arr3.flatMap(item => [item * 10]);
console.log(result);
```

四、Symbol.prototype.description

```
// 创建 Symbol
let s = Symbol('小康');
console.log(s.description); // 小康
```

ES11

一、私有属性

```
class Person {
  // 共有属性
  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('小康', 18, '45kg');

console.log(girl.name);
// console.log(girl.#age); // 私有属性不可调用
girl.intro();
```

二、Promise.allSettled

```
// 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(() => {
    resolve('商品数据 - 2');
  }, 1000);
});

// 调用 allsettled 方法
const result = Promise.allSettled([p1, p2]); // 返回状态和值
const res = Promise.all([p1, p2]); // 返回值
console.log(res);
console.log(result);
```

三、String.prototype.matchAll

```
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>`;

// 声明正则
const reg = /<li>.*?<a>(.*?)<\/a>.*?<p>(.*?)<\/p>/sg;

// 调用方法
const result = str.matchAll(reg);
```

```
// for(let v of result){
//   console.log(v);
// }

const arr = [...result];
console.log(arr);
```

四、可选链操作符

```
// ?.
function main (config) {
  // const dbHost = config && config.db && config.db.host;
  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'
  }
});
```

五、动态import加载

- html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>5-动态import加载</title>
</head>

<body>
  <button id="btn">点击</button>
  <script src="./js/app.js"></script>
</body>

</html>
```

- app.js

```
// import * as m1 from "./hello.js";
//获取元素
const btn = document.getElementById('btn');

btn.onclick = function(){
    import('./hello.js').then(module => {
        module.hello();
    });
}
```

- hello.js

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

六、BigInt

```
        //大整形
        // let n = 521n;
        // console.log(n, typeof(n));

        //函数
        // let n = 123;
        // console.log(BigInt(n));
        // console.log(BigInt(1.2));

        //大数值运算
        let max = Number.MAX_SAFE_INTEGER;
        console.log(max);
        console.log(max + 1);
        console.log(max + 2);

        console.log(BigInt(max))
        console.log(BigInt(max) + BigInt(1))
        console.log(BigInt(max) + BigInt(2))
```

七、globalThis

```
console.log(globalThis);
```