ES介绍

ES全称ECMAScript ， 是脚本语言的规范，而平时经常编写的JavaScript是ECMAScript的一种实现，所以ES新特性其实指的是JavaScript的新特性

let

```
<script>
    // 声明变量
    let a;
    let b,c,d;
    let e=100;
    let f=123,g='hello,world',h=[];

    // let声明变量的特性：
    //(1) 变量不能重复声明
    // let star='wang';
    // let star='liu';

    //(2) 块级作用域  全局  函数  eval
    // 块级作用域:在代码块里面有效，出代码块无效
    // if else while for
    // {
    //     var girl='Marry';
    // }
    // console.log(girl);   //可以

    // {
    //     let girl='Mary';
    // }
    // console.log(girl);   //不可以

    // (3)不存在变量提升
    console.log(song);
    // let song='Hero';    //不可以

    var song='Hero';    //undefined

    // (4)不影响作用域链
    {
        let school='CUGB';
        function fn(){
            console.log(school);
        };
        fn();
    }
</script>
```

const

```
<script>
    //const 关键字
    // 声明常量
    const SCHOOL='CUGB';
    console.log(SCHOOL);

    // 1. 一定要赋初始值
    // const A;

    // 2. 一般常量使用大写（潜规则）
    const a=100;   //不会报错

    // 3. 常量的值不能修改
    // SCHOOL='MIT';

    // 4. 块级作用域
    {
        const PLAYER='UZI';
    };
    // console.log(PLAYER);

    // 5. 对于数组和对象的元素修改，不算对常量的修改 不会报错
    // 常量指向的地址没有改变
    const COLOR=['PINK','RED','GREEN'];
    COLOR.push('white');

</script>
```

解构赋值

```
<script>
    // 遇到方法频繁被调用，可以考虑使用解构赋值
    // 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

    // 1. 数组的解构
    const COLOR=['RED','PINK','GREEN','BLUE'];
    let [r,p,g,b]=COLOR;
    console.log(r);
    console.log(p);
    console.log(g);
    console.log(b);

    // 2. 对象的解构
    const zhao={
        name:'zhao',
        age:18,
        xiaopin:function(){
            console.log('我可以演小品');
        }
    };

    let{name,age,xiaopin}=zhao;
    console.log(name);
    console.log(age);
    console.log(xiaopin);
    xiaopin();
</script>
```

模板字符串

```
<script>
    // 模板字符串
    // ES6引入新的声明字符串的方式

    // 1.声明
    // let str = `也是一个字符串`;
    // console.log(str, typeof str);

    // 2. 内容中可以直接出现换行符
    let str = `<ul>
        <li>海绵宝宝</li>
        <li>派大星</li>
        </ul>`;

    // 3. 直接进行变量拼接
    let lovest='海绵宝宝';
    let out=`我最喜欢的动画人物是${lovest}`;
    console.log(out);
</script>
```

简化对象写法

```
<script>
    // 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
    // 这样的写法更加简介

    let name = 'CUGB';
    let change = function () {
        console.log('艰苦朴素，求真务实');
    }

    const school = {
        name,
        change,
        improve (){
            console.log('中国地质大学北京');
        }
    }

    console.log(school);
</script>
```

箭头函数

```
<script>
    // ES 6允许使用箭头（=>）定义函数

    // 声明一个函数
    // let fn=function(){

    // }

    // let fn = (a, b) => {
    //     return a + b;
    // };
    // let result = fn(1, 2);
    // console.log(result);

    // 箭头函数声明的特性

    // 1.this 是静态的 this始终指向函数声明时所在作用域下的this值

    // 普通函数
    function getName() {
        console.log(this.name);
    }
    // 箭头函数
    let getName2 = () => {
        console.log(this.name);
    }

    // 设置window对象的name属性
    window.name = '王建钦';
    const school = {
        name: 'CUGB'
    };

    // 直接调用:
    getName();
    getName2();

    // call()方法调用
    getName.call(school); //改变为CUGB
    getName2.call(school); //未改变 还是王建钦


    // 2.不能作为构造函数实例化对象
    // let Person = (name, age) => {
    //     this.name = name;
    //     this.age = age;
    // };
    // let me = new Person('xiao', 30);
    // console.log(me);


    // 3. 不能使用arguments变量
    // let fn=()=>{
    //     console.log(arguments);
```

```
        // };
        // fn(1,2,3);


        // 4. 箭头函数的简写
        // 1)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(3));

        // 2)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
        let pow = n => n * n;

        console.log(pow(5));
    </script>
```

箭头函数案例

```
    <style>
        div {
            width: 200px;
            height: 200px;
            background: yellow;
        }
    </style>
</head>

<body>
    <div class="ad"></div>
    <script>
        // 需求1：点击div 2秒后颜色变为粉色
        // 获取元素
        let ad = document.querySelector('.ad');
        // 绑定事件
        ad.addEventListener('click', function () {
            // 定时器
            setTimeout(() => {
                // 修改背景颜色 this
                this.style.background = 'pink';

            }, 3000)
        });

        // 箭头函数this是静态的， 指向在声明时所在作用域下的this值

        // 需求2：从数组中返回偶数的元素

        const arr = [1, 45, 8745, 77, 88, 9, 3, 45, 6, 4, 2, 7, 7888888, 5, 7, 1, 335, 6, 7887
        // 原先的方式
        // const result = arr.filter(function (item) {
        //     if (item % 2 == 0) {
        //         return true;
        //     } else {
        //         return false;
        //     };
        // });
        // console.log(result);

        // 箭头函数 的方式
        const result = arr.filter(item => item % 2 === 0);
        console.log(result);


        // 总结：
        // 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

        // {
        //     name: '海绵宝宝',
        //     getName: () => {
        //         this.name;
        //     }
```

```
    // }
</script>
```

参数默认值

```
<script>
    // ES 6参数默认值
    // ES6允许给函数参数赋初始值

    // 1. 形参的初始值  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);


    // 2. 默认值可以与解构赋值结合使用
    function connect({
        host='127.0.0.1',
        username,
        password,
        port
    }) {
        console.log(host);
        console.log(username);
        console.log(password);
        console.log(port);

    }
    connect({
        // host: 'localhost',
        username: 'root',
        password: 'password',
        port: 8087
    })
</script>
```

rest 参数

```
<script>
    // ES 6引入rest参数，用于获取函数的实参，用来替代arguments

    // ES5获取实参的方式
    // function date(){
    //     console.log(arguments);
    // };

    // date('red','blue','green','yellow');
    // 获取过来的是一个对象

    // ES6的rest参数
    function date(...args) {
        console.log(args);
    };
    date('red','blue','green','yellow');
    // 获取过来的是一个数组，可以对获取的实参使用数组的API方法 filter some every map等  ,可以据


    // 注意：
    // rest参数必须放到参数最后
    function fn(a,b,...args){
        console.log(a);
        console.log(b);
        console.log(args);
    }
    fn(1,2,3,4,5,6);
</script>
```

spread扩展运算符

```
    <script>
        // ... 扩展运算符可以将数组转换为逗号分隔的参数序列

        // 声明一个数组
        const tfboys=['a','b','c'];

        // 声明一个函数
        function chunwan(){
            console.log(arguments);
        }
        chunwan(...tfboys);
        // rest 参数放在函数声明的形参里面
        // spread扩展运算符放在函数调用的实参里面


        // ... 扩展运算符可以将数组转换为逗号分隔的参数序列

        // 声明一个数组
        const tfboys=['a','b','c'];

        // 声明一个函数
        function chunwan(){
            console.log(arguments);
        }
        chunwan(...tfboys);
        // rest 参数放在函数声明的形参里面
        // spread扩展运算符放在函数调用的实参里面


        // 扩展运算符的应用
        // 1.数组的合并
        const kuaizi =['a','b'];
        const fenghuang=['c','d'];
        const hebing=[...kuaizi,...fenghuang];
        console.log(hebing);


        // 2.数组的克隆
        const original=['a','b','c'];
        const copy=[...original];
        console.log(copy);


        // 3. 将伪数组转换为真正的数组
        const divs=document.querySelectorAll('div');
        console.log(divs);
        const divArr=[...divs];
        console.log(divArr);
</script>
```

symbol

```
<script>
    // ES6引入了一种新的数据类型symbol，表示独一无二的值。它是JavaScript语言的第七种数据类型，是

    // 特点：
    // symbol的值是唯一的，用来解决命名冲突的问题
    // Symbol值不能与其他数据进行运算
    // Symbol定义的属性不能使用for... in循环遍历，但是可以使用Reflect.ownKeys来获取对象的说有键

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

    let s2 = Symbol('海绵宝宝');
    let s3 = Symbol('海绵宝宝');
    console.log(s2 == s3); // flase

    // Symbol.for 创建
    let s4 = Symbol.for('海绵宝宝');
    let s5 = Symbol.for('海绵宝宝');
    console.log(s4, typeof s4);
    console.log(s4 == s5); //true


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


    // 总结
    // 七种数据类型  USONB     you are so niubility
    // u   undefined
    // s   string  Symbol
    // o   Object
    // n   null  Number
    // b   boolean


    // Symbol 的使用
    // 向对象添加属性和方法
    let game = {}

    // 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);



        // Symbol 内置值

        class Person {
            static[Symbol.hasInstance]() {
                console.log('我被用来检测类型了');
                return true;
            }
        }

        let o = {};
        console.log(o instanceof Person);

        // Symbol.isConcatSpreadable    布尔类型 用来控制数组是否可以展开
        const arr = [1, 2, 3];
        const arr2 = [4, 5, 7];
        arr2[Symbol.isConcatSpreadable] = false;
        console.log(arr.concat(arr2));
    </script>
```

迭代器

```
<script>
    // 迭代器
    // 迭代器（Iterator）是一种接口，是对象里面的一个属性，这个属性的名字叫做Symbol.iterator
    // 任何数据结构只要部署Iterator接口，就可以完成遍历操作

    // 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
    // 原生具备iterator接口的数据（可用for of循环）
    // Array
    // Set
    // Map
    // String
    // TypedArray
    // NodeList

    // 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，要想到迭代器

    // 声明一个数组
    const xiyou = ['a', 'b', 'c', 'd'];
    // 使用for ...of 遍历数组
    for (let v of xiyou) {
        console.log(v);
    }
    console.log(xiyou);

    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());


    // 迭代器自定义遍历对象案例
    // 声明一个对象
    const banji = {
        name: '三年二班',
        stus: [
            'xiaoming',
            'wang',
            'li',
            'liu'
        ],
        [Symbol.iterator]() {
            // 索引变量
            let index = 0;
            let _this = this;
            return {
```

```
                next: function () {
                    if (index < _this.stus.length) {
                        const result = {
                            value: '_this.stus[i]',
                            done: false
                        };
                        index++;
                        return result;
                    }else{
                        return {value:undefined,done:true};
                    }
                }
            };
        }
    }

    // 遍历这个对象    每次返回的结果是stus数组的一个成员
    // banji.stus.forEach();    这样是可以的，但是不符合面向对象的思想
    for (let v of banji) {
        console.log(v);
    }
</script>
```

生成器

```
<script>
    // 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

    // 声明方式
    // function* gen() {
    //     // console.log('hello,generator');
    //     // yield可以算作函数代码的分隔符 把函数代码分割成几块 由next方法执行
    //     // console.log(1);
    //     yield '一只没有耳朵';
    //     // console.log(2);

    //     yield '一只没有尾巴';
    //     // console.log(3);

    //     yield '真奇怪';
    //     // console.log(4);

    // };

    // 执行
    // 直接调用不显示 使用next方法调用
    // let iterator = gen();
    // // console.log(iterator);
    // console.log(iterator.next());
    // console.log(iterator.next());
    // console.log(iterator.next());
    // console.log(iterator.next());
    // (图：生成器输出)

    // 遍历
    // // 每一次调用的返回结果是yield后面的表达式
    // for (let v of gen()){
    //     console.log(v);
    // }


    // 生成器函数参数
    // function* gen(arg) {
    //     console.log(arg);
    //     let one=yield 111;
    //     console.log(one);
    //     yield 222;
    //     yield 333;
    //     yield 444;
    // };

    // // 执行获取迭代器对象
    // let iterator = gen('abc');
    // console.log(iterator.next());
    // // next 方法可以传入实参  传入的参数可以作为上一个语句的整体返回结果
    // console.log(iterator.next('bbb'));
    // // console.log(iterator.next());
    // // console.log(iterator.next());
```

```
// // console.log(iterator.next());



// 生成器函数实例1
// 异步编程    如:文件操作,网络操作(ajax,request) 数据库操作

// 定时器案例
// 1秒后控制台输出111 2s后输出222 3s 后输出333

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


// 生成器函数做法
// 生成三个函数 分别完成3个异步任务
// 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);
// };

// // 生成生成器函数
// function* gen() {
//     yield one();
//     yield two();
//     yield three();
// };

// // 调用生成器函数
```

```
    // let iterator = gen();
    // iterator.next();



    // 生成器案例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';
            iterator.next(data);
        }, 1000)
    };

    function getGoods() {
        setTimeout(() => {
            let data = '商品数据';
            iterator.next(data);
        }, 1000)
    };

    // 声明生成器函数
    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();
</script>
```

Promise

```
<script>
    // Promise
    // Promise是ES6引入的异步编程的新解决方法。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

    // 实例化Promise对象
    // const P=new Promise(function(resolve,reject){
    //     setTimeout(function(){
    // let data='数据库中的用户数据';
    // resolve
    // resolve(data);
    // 调用完resolve函数 之后，Promise 对象P的状态就会变为成功 （三个状态：初始化 成功 失败）
    // let err='数据读取失败';
    // reject(err);
    // 调用完reject函数之后，promise 对象P的状态就会变为失败 失败后会调用then 方法 的第二个函数

    //     },1000);
    // });

    // 调用promise对象中的then 方法
    // then 有两个参数 都是函数  函数分别有一个形参，成功的形参一般叫value 失败的形参一般叫 rea
    // P.then(function(value){
    //     console.log(value);
    // },function(reason){
    //     console.log(reason);
    // })



    // Promise读取文件内容
    // （这部分涉及到node.js，先略过）



    // Promise封装AJAX

    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/getJoke');

        // 3.发送
        xhr.send();

        // 4. 绑定事件处理响应结果
        xhr.onreadystatechange = function () {
            if (xhr.readyState === 4) {
                //判断响应状态码 200~299之间
                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);
    })
</script>
```

Promise then方法

```
<script>
    // 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，对象状态由回调函数的执行结果决定，如果回调
    const result = P.then(value => {
        console.log(value);
        // 1. 非promise类型的属性
        // return "hi";

        // 2. 是promise对象
        // return new Promise((resolve, reject) => {
            // resolve('ok');
            // reject('error');
        // });

        // 3.抛出错误
    //     throw new Error('error');
    // }, reason => {
    //     console.warn(reason);
    // });


    // 链式调用
    // P.then(value=>{},reason=>{}).then(value=>{},reason=>{})

    // console.log(result);
</script>
```

Promise catch方法

```
<script>
    // set
    // ES6提供了新的数据结构set（集合）。它类似于数组，但成员的值都是唯一的，集合实现了iterator
    // size    返回集合的元素的个数
    // add  增加一个新元素，返回当前元素
    // delete    删除元素，返回boolean值
    // has      检测集合中是否包含某个元素，返回Boolean值

    // 声明一个set
    let s = new Set();

    // 自动去重
    let s2 = new Set(['大事', '小事', '好事', '坏事', '小事']);
    console.log(s2);

    // 元素个数
    console.log(s2.size); //4

    // 向集合添加新的元素
    s2.add('喜事');
    console.log(s2);

    // 删除元素
    s2.delete('坏事');
    console.log(s2);

    // 检测元素
    console.log(s2.has('好事'));

    // 清空
    // s2.clear();
    // console.log(s2);

    for (let v of s2) {
        console.log(v);
    };


    // set集合实践
    let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 4, 6, 7, 3, 2, 6, 7];

    // 1.数组去重
    // let result = [...new Set(arr)];
    // console.log(result);

    // // 2.交集
    let arr2 = [4, 5, 6, 5, 7, 233];
    // let result = [...new Set(arr)].filter(item => {
    //     let s2 = new Set(arr2); //4 5 6
    //     if (s2.has(item)) {
    //         return true;
    //     } else {
    //         return false;
    //     }
```

```
    // });
    // console.log(result);
    // 简单写法
    let result1 = [...new Set(arr)].filter(item => new Set(arr2).has(item));
    console.log(result1);

    // 3. 并集
    let union = [...new Set([...arr, ...arr2])];
    console.log(union);

    // 4. 差集
    let diff=[...new Set(arr)].filter(item => !(new Set(arr2).has(item)));
    console.log(diff);
</script>
```

Map

```
<script>
    // 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
    // map的属性和方法:
    // size  返回一个map的元素个数
    // set   增加一个新元素，返回当前map
    // get   返回键名对象的键值
    // has    检测map中是否包含某个元素，返回Boolean值
    // clear  清空集合，返回undefined

    // 声明map
    let m = new Map();

    // 添加元素
    m.set('name', '海绵宝宝');
    m.set('say', function () {
        '我准备好了'
    });
    let key = {
        workplace: '蟹黄堡餐厅'
    };
    // 键是对象，值是数组
    m.set(key, ['a', 'b', 'c']);
    console.log(m);


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

    // 删除
    m.delete('name');
    console.log(m);

    // 获取
    console.log(m.get('say'));
    console.log(m.get(key));

    // 清空
    // m.clear();
    // console.log(m);

    // 遍历
    for (var v of m){
        console.log(m);
    };
</script>
```

class 类

```
<script>
    ES6提供了更加接近传统语言的写法，引入了class（类）的概念，作为对象的模板，通过class关键字，

    ES5 构造函数实例化对象
    function Phone(brand, price) {
        this.brand = brand;
        this.price = price;
    };

    // 添加方法：通过原型对象来添加
    Phone.prototype.call = function () {
        console.log('我可以打电话');
    };

    // 实例化对象
    let Huawei = new Phone('Huawei', 5999);
    Huawei.call();
    console.log(Huawei);

    class
    class Phone {
        //构造方法 constructor    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 Phone('oneplus',3999);
    console.log(onePlus);
</script>
```

类的静态成员

```
<script>
    ES6提供了更加接近传统语言的写法，引入了class（类）的概念，作为对象的模板，通过class关键字，

    ES5 构造函数实例化对象
    function Phone(brand, price) {
        this.brand = brand;
        this.price = price;
    };

    // 添加方法：通过原型对象来添加
    Phone.prototype.call = function () {
        console.log('我可以打电话');
    };

    // 实例化对象
    let Huawei = new Phone('Huawei', 5999);
    Huawei.call();
    console.log(Huawei);

    class
    class Phone {
        //构造方法 constructor    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 Phone('oneplus',3999);
    console.log(onePlus);


    类的静态成员
    ES5
    function Phone() {

    }

    Phone.name = '手机';
    Phone.change = function () {
        console.log('我可以改变世界');
    }
    以上两个是属于函数对象的，不属于实例对象    是静态成员

    Phone.prototype.size = '5.5inch';

    实例对象和函数对象的属性和方法是不通的
    实例对象和原型对象的属性和方法是想通的
    let nokia = new Phone();
```

```
        console.log(nokia.name);   //undefined
        console.log(nokia.change());    //报错 nokia is not a function
        console.log(nokia.size);

        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(nokia.change());
</script>
```