目录

作业讲评

按照商品名称查询:

id	产品名称	价格
1	小米9	3999
2	oppo	999
3	荣耀V20	1699
4	华为	1999
5	华为nova	1999
6	华为P30	2999
7	华为Mate	5999
8	小米10	4999
9	荣耀V30	3599

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


<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <style>
        table {
            width: 400px;
            border: 1px solid #000;
            border-collapse: collapse;
            margin: 0 auto;
        }

        td,
        th {
            border: 1px solid #000;
            text-align: center;
        }
```

```
        .search {
            width: 600px;
            margin: 20px auto;
        }
    </style>
</head>


<body>
    <div class="search">
        按照商品名称查询: <input type="text" class="product">
    </div>
    <table>
        <thead>
            <tr>
                <th>id</th>
                <th>产品名称</th>
                <th>价格</th>
            </tr>
        </thead>
        <tbody>
        </tbody>
    </table>
    <script>
        // 利用新增数组方法操作数据
        var data = [{
            id: 1,
            pname: '小米9',
            price: 3999
        }, {
            id: 2,
            pname: 'oppo',
            price: 999
        }, {
            id: 3,
            pname: '荣耀V20',
            price: 1699
        }, {
            id: 4,
            pname: '华为',
            price: 1999
        }, {
            id: 5,
            pname: '华为nova',
            price: 1999
        },{
            id: 6,
            pname: '华为P30',
            price: 2999
        },{
            id: 7,
            pname: '华为Mate',
            price: 5999
        },{
            id: 8,
            pname: '小米10',
            price: 4999
        },{
            id: 9,
            pname: '荣耀V30',
            price: 3599
        },];


        // 获取tbody标签
        var tbody = document.querySelector("tbody");

        // 封装一个显示数据的函数
        function setData(myData){
            // 清除原来tbody里面的内容
            tbody.innerHTML = "";

            myData.filter(function(value){
                // 创建tr行
```

```
                // 创建tr行
                var tr = document.createElement("tr");
                // 设置tr行的内容
                tr.innerHTML = "<td>"+value.id+"</td><td>"+value.pname+"</td><td>"+value.price+"</td>";
                // 把tr添加到tbody标签中
                tbody.appendChild( tr );
            })
        }


        setData( data );


        // 给input文本框绑定键盘弹起事件
        var product = document.querySelector(".product");
        product.onkeyup = function(){
            // 获取文本框中的值
            var search = this.value;


            // 从一个数组里面找到满足条件的一些元素  filter返回是一个新数组
            var newArr = data.filter(function( val ){
                // 判断字符串中是否含有子字符串  可以使用indexOf方法
                return val.pname.indexOf( search ) != -1 ;
            });


            // 遍历数据
            setData( newArr );
        }
    </script>
</body>


</html>
```

学习目标

- 能够使用构造函数创建对象
- 能够说出原型的作用
- 能够说出访问对象成员的规则

forEach,filter和some区别

- 如果查询数组中唯一的元素, 用some方法更合适,在some 里面 遇到 return true, 因为return true就代表找到了元素, 所以会终止遍历 迭代效率更高
- 在forEach 里面 return 不会终止迭代
- 在filter 里面 return 不会终止迭代

- forEach,some和filter都可以对数组进行遍历,但是forEach没有返回值,some用于查找数组中是否至少有一个元素满足条件,返回一个布尔值,filter筛选满足条件的所有元素,返回一个新数组

举例:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // forEach,filter和some区别
        // 1. 相同点:forEach和filter和some都可以遍历数组中的元素,遍历就是把数组中的每个元素都访问一遍
        // 2. 不同点:  forEach没有返回值   filter返回值是满足条件元素组成的新数组   some返回值是布尔值,判断是否有一个满足条件,只要有一个满足条件就返回true;如果所有元素都不满足条件就返回false
        // 3. 不同点: forEach主要用于遍历数组,类似之前使用for遍历数组;   filter筛选满足条件的数组元素,回调函数中需要return布尔值或者条件表达式;   some判断是否至少有一个元素满足条件,回调函数中需要return布尔值或者条件表达式
        // 4. 不同点: forEach遇到return以后,不会终止遍历;  filter遇到return以后,也不会终止遍历   some效率比较高,找到满足条件以后的元素,也就是return true以后,就会终止遍历


        var arr1 = [10,20,30];
        var result1 = arr1.forEach(function(value){
            console.log( value );
            return false;
        });
        console.log( result1 );
        console.log("");

        var arr2 = [5,7,8,9,52,67];
        var result2 = arr2.filter(function(value){
            // console.log( value );


            // 返回大于10的元素
            return value > 10;


            // console.log( 111 );
            // return false;
        });
        console.log( result2 );
        console.log("");

        var arr3 = ["张五","李四","王五五五","赵六六"];
        var result3 = arr3.some(function(value){
            // console.log( value );


            // 其实就是找每个元素的值是否有等于张三的
            // return value == "张三";
            // console.log( value.length );


            console.log( 2222 );
            return value.length > 3;


            // console.log( 111  );
            // return true;
        });
        // console.log( result3 );
    </script>
</body>
</html>
```

构造函数和原型

概述

在典型的 OOP 的语言中（如 Java），都存在类的概念，类就是对象的模板，对象就是类的实例，但在 ES6之前，JS 中并没用引入类的概念。

ES6， 全称 ECMAScript 6.0 ， 2015.06 发版。但是目前浏览器的 JavaScript 是 ES5 版本，大多数高版本的浏览器也支持 ES6，不过只实现了 ES6 的部分特性和功能。

在 ES6之前 ， 对象不是基于类创建的，而是用一种称为**构造函数**的特殊函数来定义对象和它们的特征。

ES6定义类的语法

```
class 类名{
 // 定义属性
 constructor(参数1,参数2...){
 this.属性名1 = 参数1;
 this.属性名2 = 参数2;
 ...
 }
 // 定义方法 方法之间不需要使用逗号隔开,方法也不需要写function关键字
 方法名1(){
  方法体;
  }
  方法名2(){
  方法体;
  }
  ...
}
```

ES6实例化对象

```
new 类名(实参1,实参2...);

new以后自动就会调用constructor构造器方法
```

ES6继承的语法

```
class Father{

}

class Son extends Father{

}

如果在子类中存在构造器,需要super关键字调用一次
 如果子类有自己独特的属性也需要使用super关键字
```

对象的三种创建方式--复习

1.利用 字面量{}方式创建对象

```
var obj = {};
```

2.利用 new Object() 关键字创建对象

```
var obj = new Object();
```

3.利用 构造函数方式创建对象

```
function Person(name,age){
 this.name = name;
 this.age = age;
}
var obj = new Person('zs',12);
```

构造函数

在 JS 中，使用构造函数时要注意以下两点：

1.构造函数用于创建某一类对象，其首字母要大写

2.构造函数要和 new 一起使用才有意义

new 在执行时会做四件事情：

① 在内存中创建一个新的空对象。

② 让 this 指向这个新的对象。

③ 执行构造函数里面的代码，给这个新对象添加属性和方法。

④ 返回这个新对象（所以构造函数里面不需要 return ）。

举例:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 在 JS 中，使用构造函数时要注意以下两点：
        // 1.构造函数用于创建某一类对象，其首字母要大写
        // 2.构造函数要和 new 一起使用才有意义


        // 使用构造函数创建一个明星类
        function Star(uname,age,sex){
            // 属性
            this.uname = uname;
            this.age = age;
            this.sex = sex;


            // console.log( this );  this在这里面代表实例化的那个对象


            // 方法
            this.sing = function(){
                console.log( this.uname + "他今年" + this.age +"岁了,还在唱歌");
            }
        }
        // 实例化得到对象  通过new关键字
        // new 在执行时会做四件事情：
        // ① 在内存中创建一个新的空对象。
        // ② 让 this 指向这个新的对象。
        // ③ 执行构造函数里面的代码，给这个新对象添加属性和方法。
        // ④ 返回这个新对象（所以构造函数里面不需要 return ）。
        var ldh = new Star("刘德华",58,"男");
        console.log( ldh );
        var zxy = new Star("张学友",65,"男");
        console.log( zxy );
    </script>
</body>
</html>
```

静态成员和实例成员

JavaScript 的构造函数中可以添加一些成员，可以在构造函数本身上添加，也可以在构造函数内部的 this 上添加。通过这两种方式添加的成员，就分别称为静态成员和实例成员。

- 静态成员：在构造函数本身身上添加的成员称为静态成员，只能由构造函数本身来访问
- 实例成员：在构造函数内部创建的对象成员称为实例成员，只能由实例化的对象来访问

实例成员

实例成员就是构造函数内部通过this添加的成员 如下列代码中uname age sex sing 就是实例成员,实例成员只能通过实例化的对象来访问

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 使用构造函数创建一个明星类
        function Star(uname,age,sex){
            // JavaScript 的构造函数中可以添加一些成员，可以在构造函数本身上添加，也可以在构造函数内部的 this 上添加。通过这两种方式添加的成员，就分别称为静态成员和实例成员。

            // 属性
            this.uname = uname;
            this.age = age;
            this.sex = sex;

            // 方法
            this.sing = function(){
                console.log( this.uname + "他今年" + this.age +"岁了,还在唱歌");
            }
        }

        // 实例成员：在构造函数内部创建的对象成员称为实例成员，只能由实例化的对象来访问


        // 实例成员不能通过构造函数直接调用
        // console.log( Star.uname );// undefined
        // console.log( Star.sing() );// 报错


        // 实例化对象
        var ldh = new Star("黎明",45,"男");
        // 调用实例成员
        console.log( ldh.uname );
        ldh.sing();
    </script>
</body>
</html>
```

静态成员

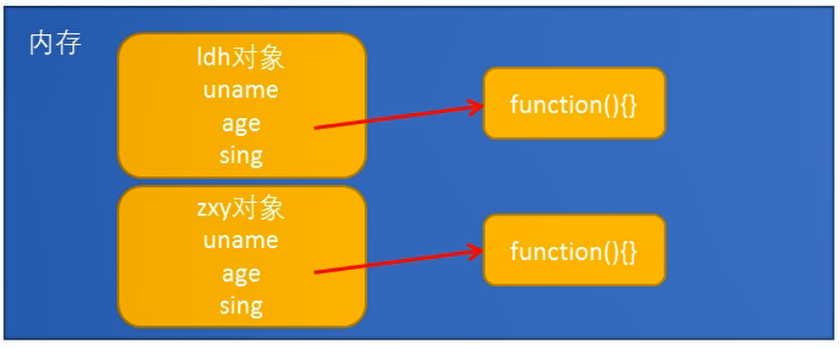
静态成员 在构造函数本身身上添加的成员 如下列代码中 address 跟 show 就是静态成员,静态成员只能通过构造函数来访问,不能通过对象来访问

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 使用构造函数创建一个明星类
        function Star(uname,age,sex){
            // 属性
            this.uname = uname;
            this.age = age;
            this.sex = sex;

            // 方法
            this.sing = function(){
                console.log( this.uname + "他今年" + this.age +"岁了,还在唱歌");
            }
        }
        // JavaScript 的构造函数中可以添加一些成员，可以在构造函数本身上添加，也可以在构造函数内部的 this 上添加。通过这两种方式添加的成员，就分别称为静态成员和实例成员。
        // 静态成员：在构造函数本身身上添加的成员称为静态成员，只能由构造函数本身来访问


        // 给Star构造函数本身身上添加静态成员
        Star.address = "hongkong";
        Star.show = function(){
            console.log("演戏,演电影");
        }


        // console.dir输出比console.log更强大
        console.dir( Star );


        // 访问静态成功  通过构造函数本身
        console.log("明星地址:" + Star.address );
        Star.show();


        // 小结：  每个对象都有的属性并且属性值相同,或者相同的方法,就可以考虑使用静态成员
        // 小结：  如果每个对象的属性值都是不一样,那就是使用实例成员
        // 小结：  相对来说,实例成员用的更多
    </script>
</body>
</html>
```

构造函数的问题

构造函数方法很好用，但是存在浪费内存的问题。 构造函数方法每次创建一个实例,就会单独创建一个空间来存放同一个函数,这样就比较浪费内存

```
function Star(uname, age) {
    this.uname = uname;
    this.age = age;
    this.sing = function() {
        console.log('我会唱歌');
}
}
var ldh = new Star('刘德华', 18);
var zxy = new Star('张学友', 19);
```



我们希望所有的对象使用同一个函数，这样就比较节省内存，那么我们要怎样做呢？

举例:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 创建一个明星类
        function Star(uname,age){
            this.uname = uname;
            this.age = age;
            this.sing = function(){
                console.log("我会唱歌");
            }
        }


        // 实例化对象
        var ldh = new Star("刘德华", 18);
        console.log( ldh );
        ldh.sing();
        console.log("");


        var zxy = new Star("张学友", 19);
        console.log( zxy );
        zxy.sing();
        console.log("");


        console.log( ldh.uname );
        console.log( zxy.uname );
        console.log("");


        console.log( ldh.sing );
        console.log( zxy.sing );
        console.log("");

        console.log( ldh.uname == zxy.uname );// false
        console.log( ldh.uname === zxy.uname );// false
        // 构造函数方法每次创建一个实例,就会单独创建一个空间来存放同一个函数,这样就比较浪费内存
        console.log( ldh.sing  == zxy.sing );// false
        console.log( ldh.sing  === zxy.sing );// false
    </script>
</body>
</html>
```

构造函数 原型对象prototype　重点,难点

构造函数通过原型分配的函数是所有对象所共享的。

JavaScript 规定，每一个构造函数都有一个prototype 属性，指向另一个对象。

注意这个prototype就是一个**对象**，这个对象的所有属性和方法，都会被构造函数所拥有。

我们可以把那些不变的方法，直接定义在 prototype 对象上，这样所有对象的实例就可以共享这些方法。

注意:一般情况下,我们的**公共属性定义到构造函数里面,公共的方法我们放到原型对象身上**

举例:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 创建一个明星类
        function Star(uname,age){
            this.uname = uname;
            this.age = age;
            this.sing = function(){
                console.log("我会唱歌");
            }
        }

        // 构造函数通过"原型"分配的函数是所有对象所共享的。
        // JavaScript 规定，每一个"构造函数"都有一个prototype 属性，指向另一个对象。注意这个prototype就是一个对象，"这个对象的所有属性和方法，都会被构造函数所拥有"。
        console.dir( Star );


        // 因为prototype叫原型,然后它是值一个"对象",所以我们通常喜欢叫prototype原型对象
        console.log( Star.prototype );// 得到一个对象


        // 怎么给一个对象添加属性?怎么给一个对象添加方法?
        // 对象.属性名 = 属性值
        Star.prototype.sex = "男";

        // 对象.方法名 = function(){}
        Star.prototype.dance = function(){
            console.log("我会跳舞");
        }


        console.log( Star.prototype );// 得到一个对象
        console.log("");
        console.log("");
        console.log("");


        // 实例化两个对象
        var ldh = new Star("刘德华",18);
        var zxy = new Star("张学友",19);
        // 这个prototype就是一个对象，"这个对象的所有属性和方法，都会被构造函数所拥有"。
        console.log(ldh.uname);
        // 构造函数上没有sex属性,但是构造函数原型对象上有,所以也可以访问
        console.log(ldh.sex);

        console.log(zxy.uname);
        // 构造函数上没有sex属性,但是构造函数原型对象上有,所以也可以访问
        console.log(zxy.sex);
        console.log("");


        ldh.dance();// 我会跳舞
        zxy.dance();// 我会跳舞
        console.log( ldh.dance == zxy.dance );// true
        console.log( ldh.dance === zxy.dance );// true
        console.log( ldh.sing == zxy.sing );// false
        console.log( ldh.sing === zxy.sing );// false


        // 我们可以把那些不变的方法，直接定义在 prototype 对象上，这样所有对象的实例就可以共享这些方法。


        // 注意:一般情况下,我们的"公共属性"定义到"构造函数"里面,"公共的方法"我们放到"原型对象"身上
    </script>
</body>
</html>
```

问答?

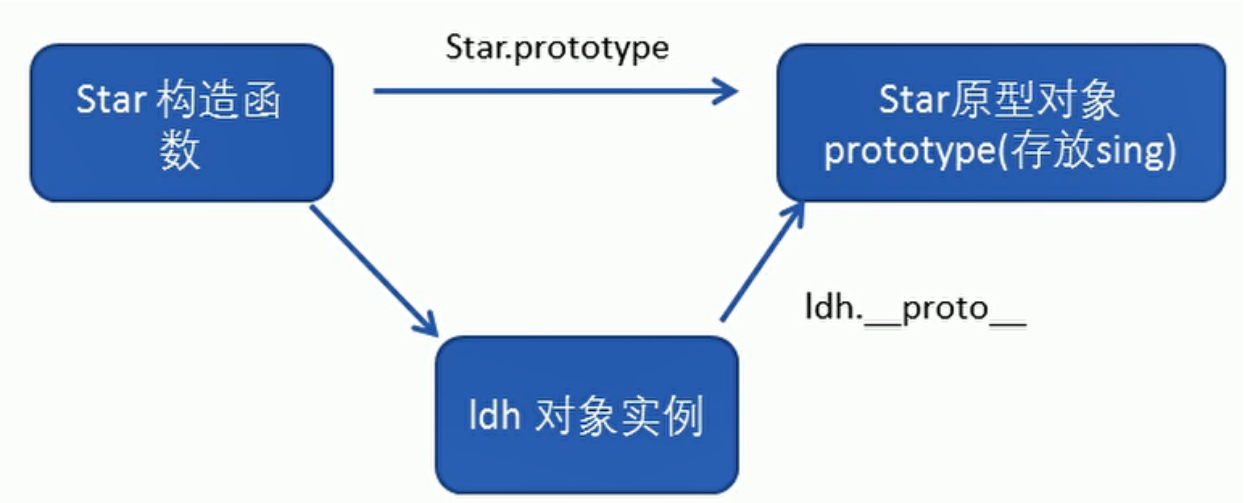
1. 原型是什么？ 一个对象，我们也称为 prototype 为原型对象。

2. 原型的作用是什么？ 共享方法。

对象原型

对象都会有一个属性 `__proto__` 指向构造函数的 prototype 原型对象，之所以我们对象可以使用构造函数 prototype 原型对象的属性和方法，就是因为对象有 `__proto__` 原型的存在。

- `__proto__`对象原型和原型对象 prototype 是等价的
- `__proto__`对象原型的意义就在于为对象的查找机制提供一个方向，或者说一条路线，但是它是一个非标准属性，因此实际开发中，不可以使用这个属性，它只是内部指向原型对象 prototype



方法的查找规则:

1. 首先先看ldh 对象身上是否有 sing 方法,如果有就执行这个对象上的sing
2. 如果没有sing 这个方法,因为有`__proto__` 的存在,就去构造函数原型对象 prototype身上去查找sing这个方法

举例1:　__proto__ 对象原型

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 定义了一个构造函数
        function Star(uname,age){
            // 公共属性放在构造函数中
            this.uname = uname;
            this.age = age;
        }


        // 公共方法放在原型对象中
        Star.prototype.sing = function(){
            console.log(this.uname + "我会唱歌" );
        }
        Star.prototype.dance = function(){
            console.log(this.uname + "我会跳舞" );
        }
        console.dir( Star );
        console.log( Star.prototype );
        console.log("");


        // 每个构造函数都有一个原型prototype,这个原型的值是一个对象,所以我经常称之为原型对象  原型对象的作用就是共享方法
        // 每个实例对象上也有一个属性叫__proto__,因为在对象上,所以叫"对象原型",它的作用就是"指向构造函数的原型对象"  开发的时候,比较少用到 __proto__这个属性,所以大家了解即可



        // 创建多个实例化对象
        var zs = new Star("张三",23);
        console.log( zs );
        zs.sing();
        zs.dance();
        // 对象原型是指向构造函数的原型对象
        console.log( zs.__proto__ === Star.prototype );// true
        console.log("");

        var ls = new Star("李四",24);
        console.log( ls );
        ls.sing();
        ls.dance();
        // 对象原型是指向构造函数的原型对象
        console.log( ls.__proto__ === Star.prototype );// true
        console.log("");
    </script>
</body>
</html>
```

```
▶ Star {uname: "张三", age: 23}
张三我会唱歌
张三我会跳舞
true

▶ Star {uname: "李四", age: 24}
李四我会唱歌
李四我会跳舞
true
```

举例2: 方法的查找规则

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 方法的查找规则:
        // 首先先看ldh 对象身上是否有 sing 方法,如果有就执行这个对象上的sing
        // 如果没有sing 这个方法,因为有__proto__ 的存在,就去构造函数原型对象prototype身上去查找sing这个方法


        function Star(uname,age){
            this.uname = uname;
            this.age = age;


            /* this.sing = function(){
                console.log("在构造函数中定义的sing方法");
            } */
        }


        // 通过原型对象添加公共方法
        Star.prototype.sing = function(){
            console.log("在原型对象中定义的sing方法");
        }
        Star.prototype.dance = function(){
            console.log("在原型对象中定义的dance方法");
        }


        var ldh = new Star("刘德华",23);
        console.log( ldh );
        // 构造函数有的,就用构造函数里面定义的;如果没有,就会去原型对象中找
        ldh.sing();
        ldh.dance();
    </script>
</body>
</html>
```

```
▶ Star {uname: "刘德华", age: 23}
在原型对象中定义的sing方法
在原型对象中定义的dance方法
```

constructor构造函数

对象原型（ `__proto__` ）和构造函数（prototype）原型对象里面都有一个属性 constructor 属性， constructor 我们称为构造函数，因为**它指回构造函数本身**。

举例:

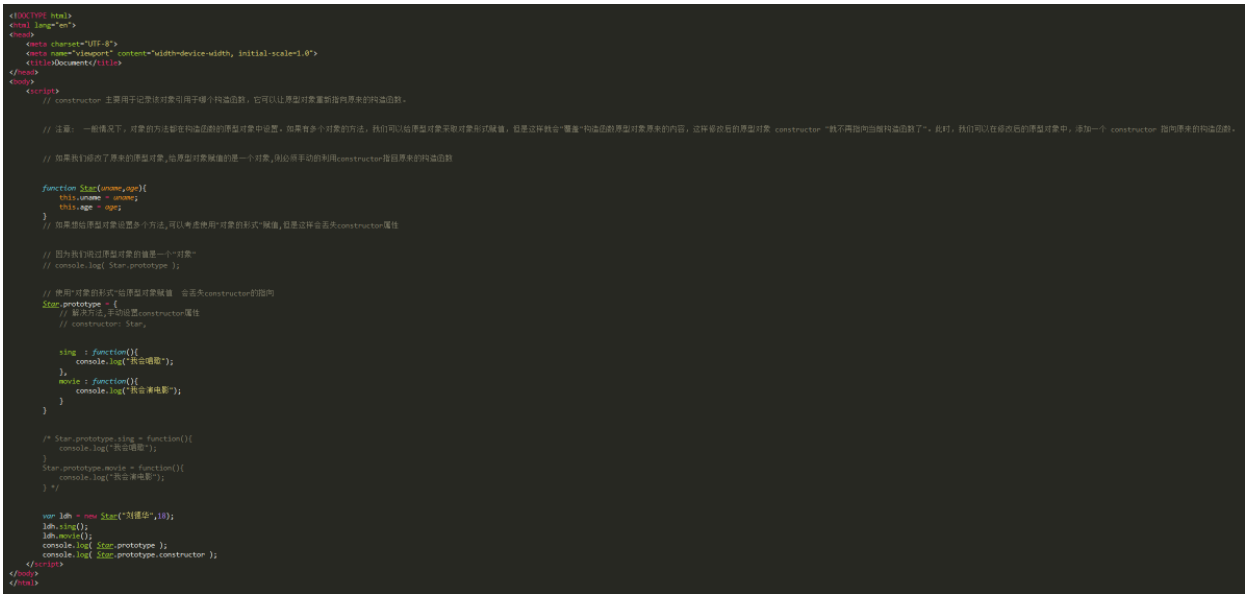
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        function Star(uname,age){
            this.uname = uname;
            this.age = age;
            this.sing = function(){
                console.log( "我会唱歌" );
            }
        }
        Star.prototype.dance = function(){
            console.log( "我会跳舞" );
        }



        // 实例化对象
        var ldh = new Star("刘德华",18);
        var zxy = new Star("张学友",19);

        // constructor构造函数
        // 对象原型（ __proto__）和构造函数（prototype）原型对象里面都有一个属性 constructor 属性 ，constructor 我们称为构造函数，因为它指回构造函数本身。


        // 输出对象原型
        console.log("对象原型");
        console.log( ldh.__proto__ );
        console.log( zxy.__proto__ );
        console.log("");

        // 输出原型对象
        console.log("原型对象");
        console.log( Star.prototype );
        console.log("");


        // 对象原型就指向原型对象，对象上没有的东西,就会去原型对象上寻找
        console.log( ldh.__proto__ === Star.prototype );
        console.log( zxy.__proto__ === Star.prototype );
        console.log("");

        // 输出对象原型身上的constructor属性,也叫构造函数,因为它指回构造函数本身。
        console.log( ldh.__proto__.constructor );
        console.log( zxy.__proto__.constructor );
        // 输出原型对象身上的constructor属性,也叫构造函数,因为它指回构造函数本身。
        console.log( Star.prototype.constructor );
        console.log("");


        // 判断对象原型的constructor属性 跟 原型对象的的constructor属性 是否全等
        console.log( ldh.__proto__.constructor === Star.prototype.constructor);
        console.log( zxy.__proto__.constructor === Star.prototype.constructor);
    </script>
</body>
</html>
```

constructor 主要用于记录该对象引用于哪个构造函数，它可以让原型对象重新指向原来的构造函数。

注意: 一般情况下，对象的方法都在构造函数的原型对象中设置。如果有多个对象的方法，我们可以给原型对象采取对象形式赋值，但是这样就会"覆盖"构造函数原型对象原来的内容，这样修改后的原型对象 constructor "就不再指向当前构造函数了"。此时，我们可以在修改后的原型对象中，添加一个 constructor 指向原来的构造函数。

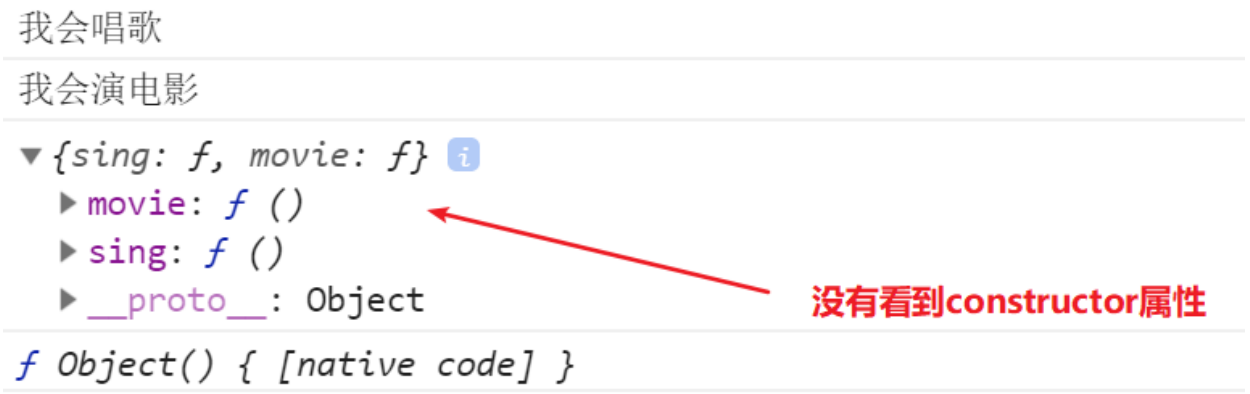
如果我们修改了原来的原型对象,**给原型对象赋值的是一个对象**,则**必须手动的利用constructor指回原来的构造函数**

举例1：　如果需要给原型对象同时设置多个方法,可以采取对象的形式赋值,但是这样会导致constructor不再指向构造函数,所以需要添加一个constructor指向原来的构造函数

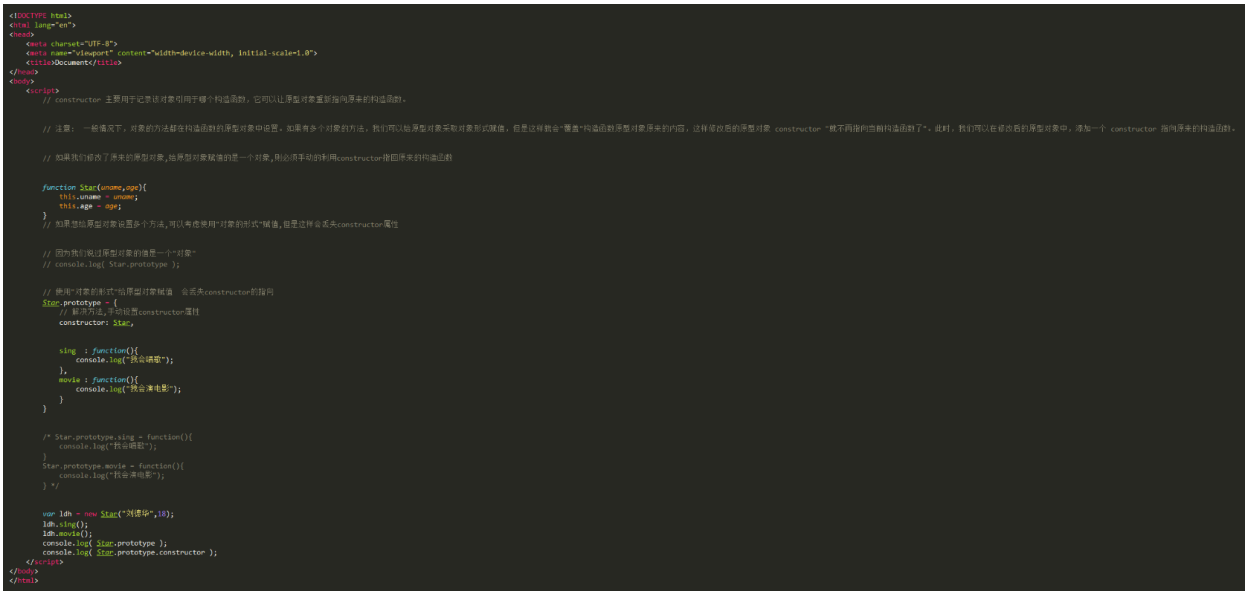


```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // constructor 主要用于记录该对象引用于哪个构造函数，它可以让原型对象重新指向原来的构造函数。

        // 注意：  一般情况下，对象的方法都在构造函数的原型对象中设置，如果有多个对象的方法，我们可以给原型对象采取对象形式赋值，但是这样就会"覆盖"构造函数原型对象原来的内容，这样修改后的原型对象 constructor "就不再指向当前构造函数了"。此时，我们可以在修改后的原型对象中，添加一个 constructor 指向原来的构造函数。

        // 如果我们修改了原来的原型对象,给原型对象赋值的是一个对象,则必须手动的利用constructor指回原来的构造函数


        function Star(uname,age){
            this.uname = uname;
            this.age = age;
        }
        // 如果给原型对象设置多个方法,可以考虑使用"对象的形式"赋值,但是这样会丢失constructor属性

        // 因为我们现在原型对象的值是一个"对象"
        // console.log( Star.prototype );

        // 使用"对象的形式"给原型对象赋值，会丢失constructor的指向
        Star.prototype = {
            // 解决方法,手动设置constructor属性
            // constructor: Star,


            sing : function(){
                console.log("我会唱歌");
            },
            movie : function(){
                console.log("我会演电影");
            }
        }

        /* Star.prototype.sing = function(){
            console.log("我会唱歌");
        }
        Star.prototype.movie = function(){
            console.log("我会演电影");
        } */


        var ldh = new Star("刘德华",18);
        ldh.sing();
        ldh.movie();
        console.log( Star.prototype );
        console.log( Star.prototype.constructor );
    </script>
</body>
</html>
```

以上代码运行结果,输出原型对象,如果未设置constructor属性,如图:



举例2：通过你 constructor: Star 手动设置constructor指向



以上代码运行结果,输出原型对象,如果手动设置constructor属性如图:

原型对象的应用——扩展内置对象方法

我们可以通过原型对象为数组扩展内置方法,对原来的内置对象进行扩展自定义的方法; 比如给数组增加自定义求偶数和的功能

注意：数组和字符串等内置对象不能给原型对象覆盖操作 Array.prototype = {} ，只能是 Array.prototype.xxx = function(){} 的方式。

举例:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 我们学过哪些内置对象？ Array数组  String字符串  Math数学  Date日期
```

```
// 因为我们在创建数组的时候也可以使用new的方式,那么Array也是一个构造函数
/* var arr = new Array(10,20,30,40,50);
// 构造函数就会有原型对象
console.log( Array.prototype );
arr.push( 60 );
console.log( arr );


console.log( String.prototype ); */


// 我们可以通过原型对象给内置对象扩展一些自定的方法


// 比如求数组的偶数和的方法  内置Array对象是没有给我们提供的
Array.prototype.getEvenSum = function(){
    // 在原型对象中this指向实例
    // console.log( this );


    var sum = 0;
    for(var i = 0; i<this.length; i++){
        if( this[i]%2== 0){
            sum += this[i];
        }
    }
    return sum;
}


// 求数组中最大值  内置Array对象是没有给我们提供的
Array.prototype.getMax = function(){
    var max = this[0];
    for(var i = 1;i <this.length;i++){
        if( max < this[i] ){
            max = this[i];
        }
    }
    return max;
}


// 内置的构造函数的原型对象,可以使用对象的方式赋值吗?   不可以,添加不了
// 自定的构造函数的原型对象,就可以使用对象的方式的赋值,但是要注意constructor指回原构造函数
/* Array.prototype = {
    getEvenSum : function(){
        // 在原型对象中this指向实例
        // console.log( this );


        var sum = 0;
        for(var i = 0; i<this.length; i++){
            if( this[i]%2== 0){
                sum += this[i];
            }
        }
        return sum;
    },


    getMax : function(){
        var max = this[0];
        for(var i = 1;i <this.length;i++){
            if( max < this[i] ){
                max = this[i];
            }
        }
        return max;
```

```
        }
    }; */


    // 输出原型对象
    console.log( Array.prototype );
    console.log("");

    // 创建实例并且调用方法
    var arr1 = [10,50,70,1,3];
    var result1 = arr1.getEvenSum();
    console.log( result1 );
    console.log("");


    var arr2 = [8,3,2,12,7];
    var result2 = arr2.getEvenSum();
    console.log( result2 );
    console.log("");


    var arr3 = new Array(3,20,2020,15,44);
    var result3 = arr3.getEvenSum();
    console.log( result3 );
    console.log("");


    var arr4 = [10,20,5,70,3];
    console.log( arr4.getMax() );
</script>
</body>
</html>
```

练习:

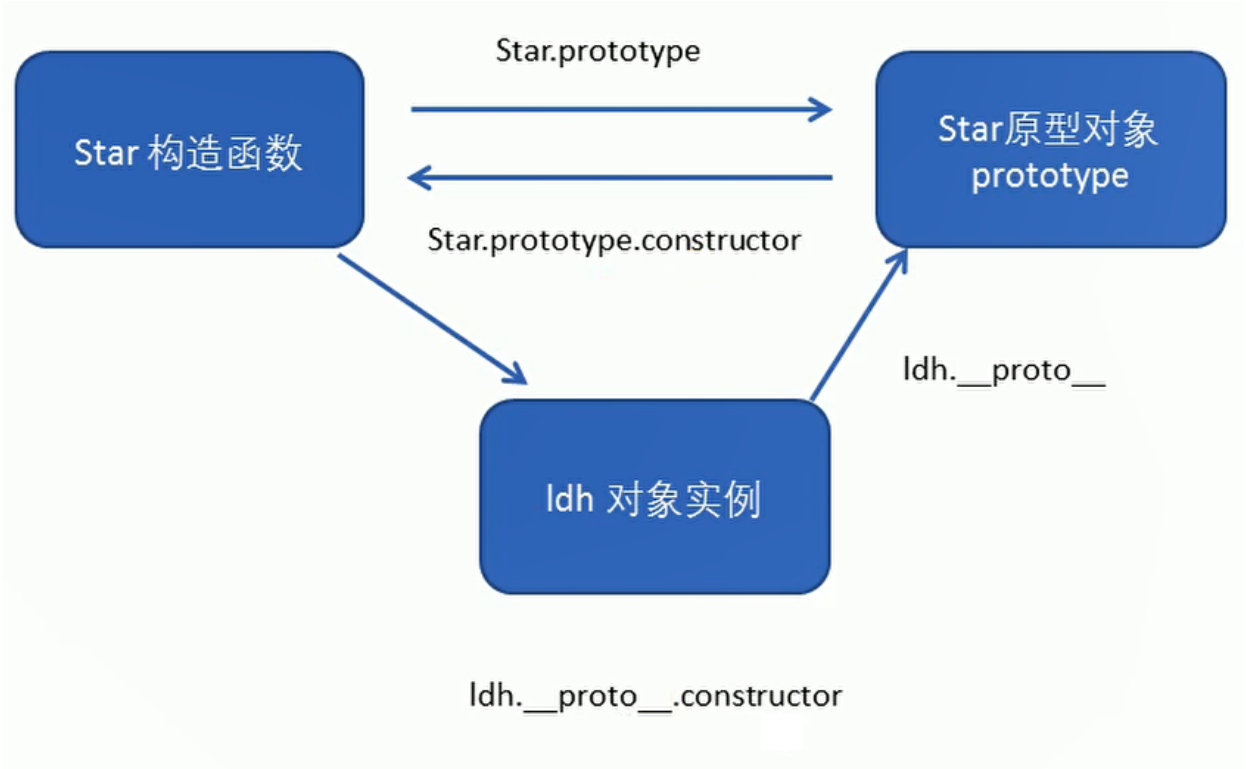
1. 使用构造函数的方式,创建一个汽车Cars类,里面有color,brand属性,
需要通过原型对象方式给个构造函数一个run的方法　　要求大家
把原型对象输出　把对象原型输出　调用run方法

2. 给数组内置对象扩展一个方法:　求数组中是3的倍数的元素的和
再创建几个对象,调用这个方法

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 1.  使用构造函数的方式,创建一个汽车Cars类,里面有color,brand属性,需要通过原型对象方式给个构造函数一个run的方法    要求大家   把原型对象输出  把对象原型输出  调用run方法
        function Cars(color,brand){
            this.color = color;
            this.brand = brand;
        }


        Cars.prototype.run = function(){
            console.log("小汽车行驶中...");
        }


        console.log( Cars.prototype );
        // 实例化对象
        var car = new Cars("粉色","兰博基尼");
        console.log( car.__proto__ );
        car.run();


        // 2.  给数组内置对象扩展一个方法:  求数组中是3的倍数的元素的和   再创建几个对象,调用这个方法
        Array.prototype.getThreeSum = function(){
            var sum = 0;
            for(var i=0;i<this.length;i++){
                if(this[i]%3 == 0){
                    sum += 0;
                }
            }
            return sum;
        }
        var arr1 = [3,10,5,6,7,9];
        console.log( arr1.getThreeSum );
    </script>


</body>
</html>
```

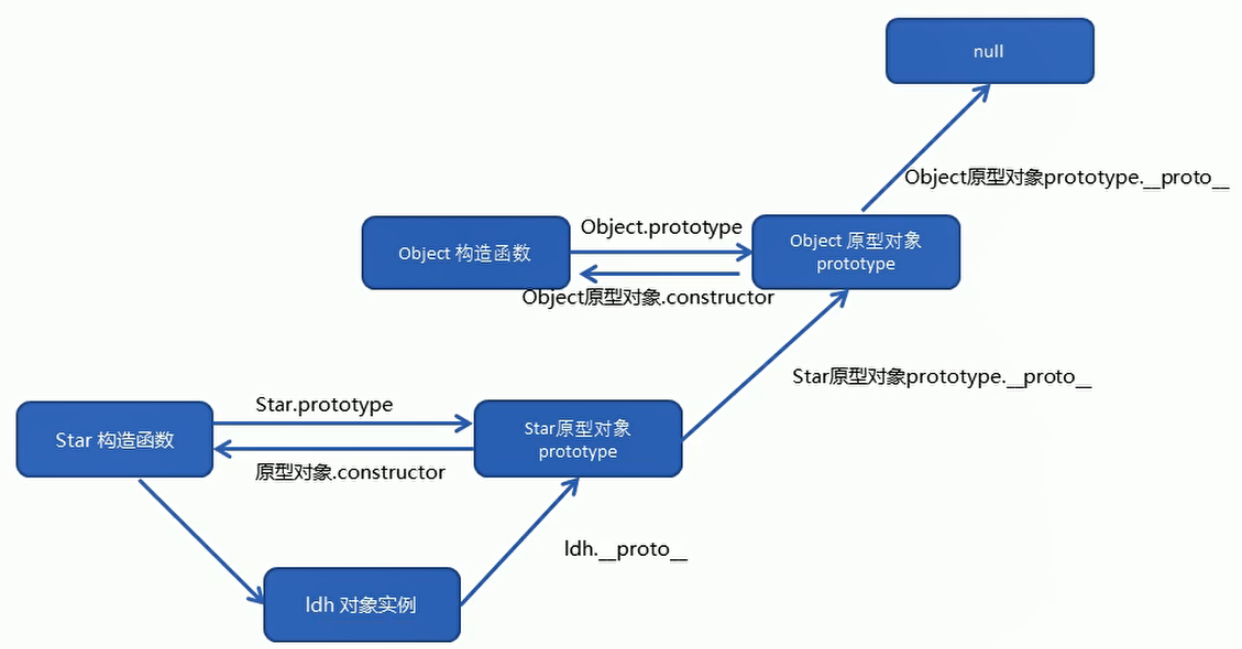
构造函数实例和原型对象三角关系

1. 构造函数的prototype属性指向了构造函数原型对象
2. 实例对象是由构造函数创建的,实例对象的__proto__属性指向了构造函数的原型对象
3. 构造函数的原型对象的constructor属性指向了构造函数,
   实例对象的原型的constructor属性也指向了构造函数



原型链 重点,难点

每一个实例对象又有一个__proto__属性，指向的构造函数的原型对象，**构造函数的原型对象也是一个对象**，所以也有__proto__属性，这样一层一层往上找就形成了原型链。



举例:

```
1. 只要是对象就有__proto__原型,指向原型对象
2. 我们Star原型对象里面的__proto__原型指向的是 Object.prototype
3. 我们Object.prototype原型对象里面的__proto__原型 指向为 null
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 每一个实例对象又有一个__proto__属性，指向的构造函数的原型对象，"构造函数的原型对象也是一个对象"，所以也有"__proto__"属性，这样一层一层往上找就形成了原型链。
        function Star(uname,age){
            this.uname = uname;
            this.age = age;
        }
        Star.prototype.sing = function(){
            console.log("我会唱歌");
        }
        var ldh = new Star("刘德华",18);
        // 1. 只要是对象就有__proto__对象原型,指向原型对象
        console.log( ldh.__proto__ );
        console.log( Star.prototype );
        console.log("");


        // 2. 我们Star原型对象里面的__proto__原型指向的是 Object.prototype
        console.log( Star.prototype.__proto__ );
        console.log( Object.prototype );
        console.log( Star.prototype.__proto__ === Object.prototype );
        console.log("");

        // 3. 我们Object.prototype原型对象里面的__proto__原型 指向为 null
        console.log( Object.prototype.__proto__ );
    </script>
</body>
</html>
```

原型链成员的查找机制 重点

任何对象都有原型对象,也就是prototype属性,任何原型对象也是一个对象,该对象就有__proto__属性,这样一层一层往上找,就形成了一条链,我们称此为原型链;

JavaScript 的成员查找机制(规则):

1. 当访问一个对象的属性（包括方法）时，首先查找这个对象自身有没有该属性。
2. 如果没有就查找它的原型（也就是 __proto__指向的 prototype 原型对象）。
3. 如果还没有就查找原型对象的原型（Object的原型对象）。
4. 依此类推一直找到 Object 为止（null）。
5. __proto__对象原型的意义就在于为对象成员查找机制提供一个方向，或者说一条路线。
6. 如果多个原型对象同时有同一个属性(或者方法),遵循就近原则

举例: 分别给实例对象,Star原型对象,Object原型对象设置sex属性

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 之前我们说过,构造方法里面没有东西,就会原型对象中找
        function Star(uname,age){
            this.uname = uname;
            this.age = age;
        }
        Star.prototype.sing = function(){
            console.log( "我会唱歌" );
        }


        // Star.prototype.sex = "男";


        // 给Object的原型对象设置和方法
        Object.prototype.sex = "保密";

        Object.prototype.hello = function(){
            console.log("我是Ojbect里面hello方法~~你好,你猴,雷猴...");
        }


        // 实例化一个对象
        var ldh = new Star("刘德华" , 18);
        console.log( ldh.sex );
        console.log("");

        // 再实例化一个对象
        var dlrb = new Star("迪丽热巴",16);
        // 对象自己身上有一个sex属性
        dlrb.sex = "女";
        console.log( dlrb.sex );
        console.log("");


        // Star.prototype.__proto__指向的Object的原型对象
        // Star.prototype是原型对象,只要是对象就会有__proto__对象原型
        console.log( Star.prototype.__proto__ );
        console.log( Object.prototype );
        console.log( Star.prototype.__proto__ === Object.prototype );
        console.log( "" );
        // 以上代码总结得到,Star构造函数的上一层是Object构造函数


        var fbb = new Star("方冰冰",15);
        console.log( fbb.sex );
        console.log("");


        // 我们之前说过强制转换字符串有一种方式是  对象.toString()
        var arr = ["张三","男",23,"H5","头发浓密"];
        var str = arr.toString();
        // 输出数组构造函数的原型对象
        console.log( Array.prototype );
        console.log( str );
        console.log("");


        // 虽然arr和fbb对应的构造函数的原型没有hello这个方法,因为原型链的关系,Object.prototype上有hello这个方法  也可以使用
        arr.hello();
        fbb.hello();
    </script>
</body>
</html>
```

原型对象中this指向

构造函数中的this和原型对象的this,都指向我们new出来的实例对象

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 构造函数中的this和原型对象的this,都指向我们new出来的实例对象
        var that;
        var another;
        function Star(uname,age){
            this.uname = uname;
            this.age = age;

            that = this;


            console.log( "构造函数中的this" );
            console.log( this );
            console.log( "" );
        }
        Star.prototype.sing = function(){
            another = this;


            console.log( "我会唱歌" );
            console.log( "原型对象中的this" );
            console.log( this );
            console.log( "" );
        }


        var ldh = new Star("刘德华",18);
        ldh.sing();


        console.log("输出实例对象ldh");
        console.log( ldh );
        console.log("");


        console.log( ldh === that );
        console.log( ldh === another );
        console.log( that === another );
    </script>
</body>
</html>
```

今日总结

xmind要做,下周一交

今日作业

请查看作业文件夹