# 1-javaScript基础

## 1.1 基本介绍

1、组成部分

| 组成部分 | 作用 |
| --- | --- |
| `DOM` | `Document Object Model` 文档对象模型，用于操作网页中各种元素和标签 |
| `BOM` | `Browser Object Model` 浏览器对象模型，用于操作浏览器中各种对象。如：`window` |
| `ECMA Script` | 脚本语言规则，制定JS脚本的核心基础 |

2、相关联系

```
html   超文本标记语言
css    层叠样式表
js     脚本语言

结构表现和行为
html   结构层
css    表现层
js     行为层
```

3、script标签的说明

- 标签个数：每个网页可以导入多个外部脚本，而且都会依次。
- 位置：在网页的任何位置都可以执行，甚至在`HTML`标签的外面。
- 语句后分号不是必须的，建议加上。

```
<script type="text/javascript" src="引入外部的JS文件"></script>
```

4、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>第一个JS代码</title>
</head>
<body>
<!--
输出5个Hello World
所有的JS代码写在script标签中
-->
<script type="text/javascript">
    for (var i = 0; i < 5; i++) {
        //在文档上写
        document.write("<h2>Hello World</h2>");
```

```
    }
</script>
</body>
</html>
```

5、输出语句

`alert` 、 `document.write` 、 `console.log` 的区别。

- `alert()` == 系统弹出框。
- `document.write()` == 文档输入内容。
- `console.log()` === 控制台中的console显示内容

6、注释

- // 单行注释
- /* 多行注释*/

# 1.2 标识符

变量、函数、属性的名字，或者函数的参数。

## 1.2.1 JS与Java的区别

| 特点 | Java | JavaScript |
| --- | --- | --- |
| **面向对象** | 面向对象 | 基于对象，不完全面向对象。<br>面向对象的某些特性是没有的 |
| **运行方式** | 编译型语言，生成字节码文件 | 解释型语言，不会生成中间文件。<br>运行一部分解析一部分。 |
| **跨平台** | 通过虚拟机运行在不同的操作系统上 | 运行在浏览器，只要系统有浏览器就可以运行 |
| **数据类型** | 强类型，不同数据类型严格区分<br>如：`String str = 123;` //错 | 弱类型，不同数据类型之间可以赋值 |
| **大小写** | 区分大小写 | 区分大小写 |

## 1.2.2 标识符的命名规则

- 由字母、数字、下划线 (_) 或美元符号 ($) 组 成。
- 不能以数字开头
- 不能使用关键字、保留字等作为标识符。

# 1.3 变量的定义

| 数据类型 | Java中定义变量 | JS中定义变量 |
| --- | --- | --- |
| **整数** | int i = 5; | var |
| **浮点数** | float f = 3.14; 或 double d=3.14; | var |
| **布尔** | boolean b = true; | var |
| **字符** | char c = 'a'; | var |
| **字符串** | String str = "abc"; | var |

### 1.3.1 注意事项

1、关于弱类型

- 同一个变量可以赋值为不同的数据类型。

2、在`JS`中的字符和字符串引号

- 在`JS`中没有字符和字符串区分，只有字符串，字符串既可以使用单引号，也可以使用双引号。

3、`var`定义变量的特点

- `var`不是必须的，但建议加上。
- 在`JS`同一个变量可以定义多次。
- 方法外面的大括号不能限制变量的作用范围。

4、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>变量定义</title>
</head>
<body>
    <script type="text/javascript">
        //定义变量
        i=8;
        document.write("整数：" + i + "<br/>");

        var f=3.14;
        document.write("浮点数：" + f + "<br/>");

        var b=true;
        document.write("布尔：" + b + "<br/>");

        //在JS中没有字符和字符串区分，只有字符串，字符串既可以使用单引号，也可以使用双引号。
        var str ="abc";
        document.write("字符串：" + str + "<br/>");

        {
            var str = 'xyz123';
        }
        document.write("字符串：" + str + "<hr/>");

        //弱类型
        b = 100;
```

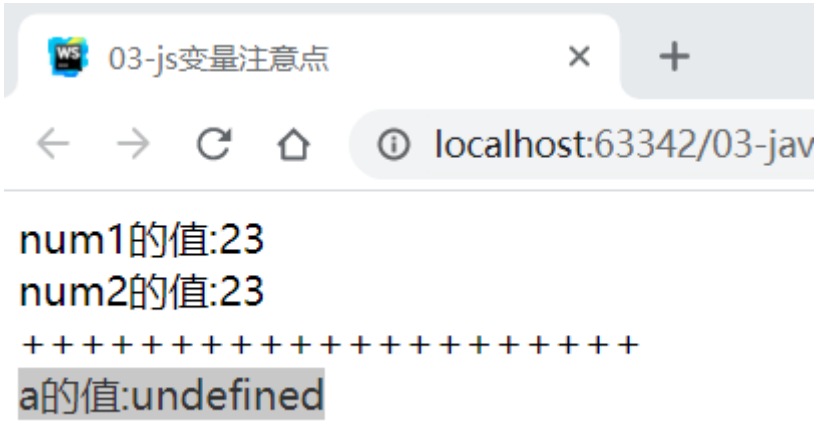
```
        document.write("整数类型: " + b + "<br/>");

    </script>
</body>
</html>
```

### 1.3.2 js变量注意点

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>03-js变量注意点</title>
  <script type="text/javascript">
    // 定义变量
    var num1;
    var num2;
    num2 = 23;
    num1 = num2; // 将num2中的值拷贝一份给num1
    document.write("num1的值:" + num1+"<br/>");
    document.write("num2的值:" + num2+"<br/>");

    document.write("++++++++++++++++++++++" + "<br/>");
    /*预处理之后的代码*/
    var a;
    document.write("a的值:" + a); // a的值:undefined
    a = 123;
  </script>
</head>
<body>
    <!--
        1.在JavaScript中变量之间是可以相互赋值的。如果定义了同名的变量，那么后定义的变量会覆盖先定义的变量。
        3.在老版本的标准的(ES6之前)JavaScript中可以先使用变量，再定义变量，并不会报错
        由于JavaScript是一门解释型的语言，会边解析边执行，浏览器在解析JavaScript代码之前还会进行一个操作"预解析(预处理)"
        预解析(预处理)步骤:将当前JavaScript代码中所有变量的定义和函数的定义放到所有代码的最前面
    -->
</body>
</html>
```

2、执行结果




# 2- 数据类型

## 2.1 基本数据类型

| 关键字 | 说明 |
| --- | --- |
| **number** | 数值型，包含：整数和浮点数 |
| **boolean** | 布尔型，包含：`true`或`false` |
| **string** | 字符串类型，包含字符和字符串 |
| **object** | 对象类型，包含自定义的对象，系统内置对象 |
| **undefined** | 未定义的数据类型，没有初始化的变量。 |

## 2.1 typeof操作符

作用：判断一个变量的数据类型，系统自带函数

| 作用 | 判断一个变量的数据类型，系统自带函数 |
| --- | --- |
| 语法 | `typeof`变量名或者`typeof`(变量名) |
| 返回值 | `string`类型，有可能是:`string`、`number`、`boolean`、`object`、`undefined`、`function` |

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>变量定义</title>
</head>
<body>
<script type="text/javascript">
    //定义变量
    var i=8;
    document.write("整数：" + i + "<br/>");
    document.write("类型：" + typeof(i) + "<hr/>");
```

```
    var f=3.14;
    document.write("浮点数：" + f + "<br/>");
    document.write("类型：" + typeof(f) + "<hr/>");

    var b=true;
    document.write("布尔：" + b + "<br/>");
    document.write("类型：" + typeof(b) + "<hr/>");

    var str ="abc";
    document.write("字符串：" + str + "<br/>");
    document.write("类型：" + typeof(str) + "<hr/>");
    document.write("类型：" + typeof('a') + "<hr/>");

    var obj = new Date();   //内置的日期对象
    document.write(obj + "<br/>");
    document.write("类型：" + typeof(obj) + "<hr/>");

    var u;  //不知道什么类型
    document.write("类型：" + typeof(u) + "<hr/>");

    var n = null;  //是一个对象类型，但对象没有值。
    document.write("类型：" + typeof(n) + "<hr/>");
</script>
</body>
</html>
```

## 2.3 数值型数据类型

1、null与undefined区别

| null与undefined的区别 | 说明 |
| --- | --- |
| **null** | 对象类型，但是对象没有值 |
| **undefined** | 未定义的类型，不知道的类型 |

2、基本介绍

`Number`：表示整数和浮点数。
`NaN`：即非数值`(Not a Number)`是一个特殊的数值。

3、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>06-转换为数值类型</title>
  <script type="text/javascript">
    // 1.将String类型转换为数值类型
    // 1.1 定义str变量
    let str1 = "123";
    console.log("str1的值:" + str1);
    /*如果字符串中都是数值，那么就正常转换*/
    console.log("str1值类型:" + typeof str1);
    // 转换
    let num1 = Number(str1);
```

```
    console.log("num的值:" + num1);
    console.log("---------");

    // 1.2 定义str变量
    let str2 =" ";
    let num2 = Number(str2);
    /*如果字符串是一个空串"  ", 那么转换之后是0*/
    console.log("num2的值:" + num2);
    console.log("num2值类型:" + typeof num2);
    console.log("============");

    // 1.3 定义变量
    let a1 = "36px";
    let a2 = Number(a1);
    /*如果字符串中不仅仅是数字, 那么转换之后是NaN*/
    console.log("a2的值:" + a2);
    console.log("a2值类型:" + typeof a2);
    console.log("++++++++++++++++++++");

    // 2.将Boolean类型转换为数值类型
    let flag = true;
    let a3 = Number(flag);
    /*如果是布尔类型的true, 那么转换之后的结果是1*/
    console.log("a3的值:" + a3);
    console.log("a3值类型:" + typeof a3);
    console.log("********************");

    let flag1 = false;
    let a4 = Number(flag1);
    /*如果是布尔类型的true, 那么转换之后的结果是1*/
    console.log("a4的值:" + a4);
    console.log("a4值类型:" + typeof a4);
    console.log("********************");

    // 3.如果是空类型, 那么转换之后的结果是0.
    let a5 = null;
    let a6 = Number(a5);
    console.log("a6的值:" + a6);
    console.log("a6值类型:" + typeof a6);
    console.log("********************");

    // 4.如果是未定义类型, 那么转换之后的结果是NaN
    let value2 = undefined;
    let value3 = Number(value2);
    console.log("value3的值:" + value3);
    console.log("value3值类型:" + typeof value3);
  </script>
</head>
<body>

</body>
</html>
```

4、执行结果


### 2.3.1 isNaN()方法

1、基本介绍

- 功能：检测`n`是否是"非数值" 返回值：`boolean`。
- 参数：参数`n`可以是任何类型。

**2、注意点**

- 任何涉及NaN的操作(例如NaN/10)都会返回`NaN`。
- `NaN`与任何值都不相等，包括`NaN`本身。
- `isNaN()`在接收到一个值之后，会尝试将这个值转换为数值。某些不是数值的值会直接转换为数值。

3、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <script type="text/javascript">
        var name_01="kobe",age=18,email="hxy1625309592@aliyun.com";

        var distance=12.67;

        var id="16";
```

```
        // 在控制台中打印
        console.log(typeof(distance))

        console.log(typeof(age-"abc"));

        console.log(isNaN(email));

        console.log(isNaN(id));

        id=Number(id);

        name_01=Number(name_01);

        console.log(typeof id);
        // NaN
        console.log(name_01);
    </script>
</body>
</html>
```

### 2.3.2 parseInt()方法

1、基本介绍

忽略字符串前面的空格，直至找到第一个非空格字符。

2、注意点

- `parseInt()`：转换空字符串返回NaN。
- `parseInt()`这个函数提供第二个参数：转换时使用的基数【即多少进制】

### 2.3.3 parseFloat()方法

1、基本介绍

- 从第一个字符开始解析每个字符，直至遇见一个无效的浮点数字符为止。

2、注意点

- 除了第一个小数点有效外，`parseFloat()`与`parseInt()`的第二个区别在于它始终都会忽略前导的零。

3、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>parseFloat()方法</title>
</head>
<body>
    <script type="text/javascript">
        var topval=parseInt("28px");

        var c="abc58"
        // 28
        console.log(topval);
        // NAN
        console.log(parseInt(c));
```

```
        // 15
        console.log(parseInt("0xf",16));

        var d=parseFloat("12.34.56px");
        // 12.34
        console.log(d);

        var e=parseFloat("0.123abc");
        // 0.123
        console.log(e);
    </script>
</body>
</html>
```

## 2.4 String数据类型

1、基本介绍

`String`类型用于表示由零或多个`16`位`Unicode`字符组成的字符序列，即字符串。字符串可以由双引号(`" "`)或单引号(`''`)表示。

2、转换方法

- 对于`Number`类型和`Boolean`类型来说, 可以通过 变量名称`.toString()`的方式来转换。
- 可以通过`String`(常量`or`变量);转换为字符串。
- 还可以通过 变量`or`常量 + "" / 变量`or`常量 + ''转换为字符串。

3、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>05-转换为字符串类型</title>
  <script type="text/javascript">
    // 定义变量
    let value = 123;
    document.write("value的值: " + value + "<br/>");
    document.write(typeof value + "<br/>");
    document.write("++++++++++++++" +"<br/>");
    // 将value变量中存储的数据拷贝一份, 然后将拷贝的数据转换为字符串之后返回.
    let str = value.toString();
    document.write("str的值:" + str + "<br/>");
    document.write(typeof str + "<br/>");
    document.write("++++++++++++++" +"<br/>");
    // 不能使用常量直接调用toString方法, 因为常量是不能改变的.
    // let st2 = 123.toString();

    // String(常量or变量), 因为是根据传入的值重新生成一个新的值, 并不是修改原有的值
    let st1 = 123;
    document.write("st1的值: " + st1 + "<br/>");
    let st2 = String(st1);
    document.write("st2的值: " + st2 + "<br/>");
    document.write(typeof st2 + "<br/>");
    document.write("++++++++++++++" +"<br/>");

    let str3 = 123;
```

```
    let str4 = str3 + "";
    document.write("str4的值：" + str4 + "<br/>");
    document.write(typeof str4 + "<br/>");
  </script>
</head>
<body>
</body>
</html>
```

4、执行结果


## 2.5 Boolean数据类型

1、基本介绍

用于表示真假的类型，即`true`表示真，`false`表示假。

2、类型转换

- 除0之外的所有数字，转换为布尔型都为`true`。
- 除`" "`之外的所有字符，转换为布尔型都为`true`。
- `null`和`undefined`转换为布尔型为`false`

3、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>数据类型</title>
</head>
<body>
    <script type="text/javascript">
        var ids=78965;
        var idstr=ids.toString();
        console.log(typeof idstr);

        var m;
```

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

        var isChild=false;
        console.log(isChild.toString());

        var x=0;
        console.log(Boolean(x));
        var strings=" ";
        console.log(Boolean(strings));
        var y;
        console.log(Boolean(y));
        var timer=null;
        console.log(Boolean(timer));
    </script>
</body>
</html>
```

# 3- 数组

## 3.1 数组定义

- 数组可以把一组相关的数据一起存放，并提供方便的访问(获取)方式。
- 数组是指**一组数据的集合**，其中的每个数据被称作**元素**，在数组中可以**存放任意类型的元素**。
- 数组是一种将一组数据存储在单个变量名下的优雅方式。

## 3.2 创建数组

**JS**中创建数组有两种方式

1、利用 `new` 创建数组

```
注意 Array ()，A 要大写
var 数组名 = new Array() ;
// 创建一个新的空数组
var arr = new Array();
```

2、利用数组字面量创建数组

```
//1. 使用数组字面量方式创建空的数组
var  数组名 = [];
//2. 使用数组字面量方式创建带初始值的数组
var  数组名 = ['kebe','jmaes',10,'11'];
```

**注意点**

- 数组的字面量是方括号`[ ]`
- 声明数组并赋值称为数组的初始化
- 这种字面量方式也是我们以后最多使用的方式

3、数组元素的类型

数组中可以存放任意类型的数据，例如字符串，数字，布尔值等。

```
var arrStus = ['Rondo',12,true,28.9];
```

## 3.3 数组解析赋值

1、基本介绍

解构赋值是ES6中新增的一种赋值方式。

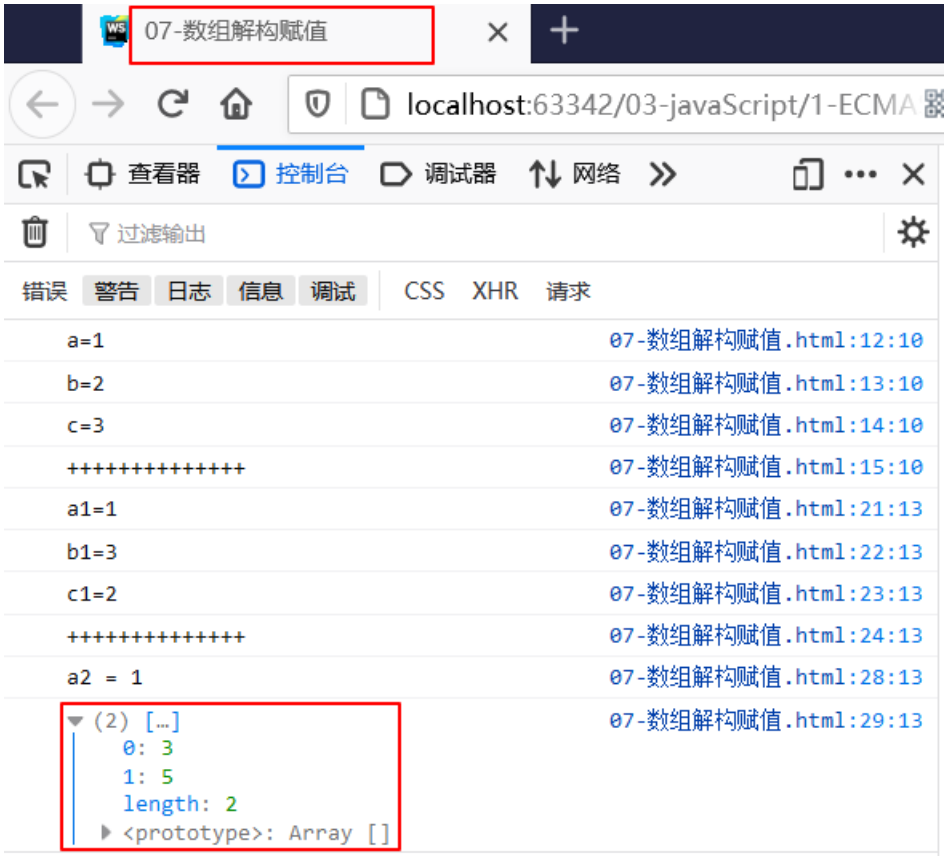
2、解析赋值特点

- 在数组的解构赋值中, 左边的个数可以和右边的个数不一样。
- 如果右边的个数和左边的个数不一样, 那么可以给左边指定默认值。

3、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>07-数组解构赋值</title>
  <script type="text/javascript">
    // 定义数组
    let array = [1,2,3,5];
    // 赋值操作
    let [a, b, c] = array;
    // 输出数组元素
    console.log("a=" + a);
    console.log("b=" + b);
    console.log("c=" + c);
    console.log("+++++++++++++++");

    // 2.等号左边的格式必须和等号右边的格式一模一样, 才能完全解构
    // 定义数组
    let [a1, b1, [c1, d1]] = [1, 3, [2, 4]];
    // 输出数组元素
    console.log("a1=" + a1);
    console.log("b1=" + b1);
    console.log("c1=" + c1);
    console.log("+++++++++++++++");

    // 3.ES6中新增的扩展运算符: ...
    let [a2, ...b2] = [1, 3, 5];
    console.log("a2 = " + a2);
    console.log(b2);
  </script>
</head>
<body>

</body>
</html>
```

4、执行结果




## 3.4 获取数组元素

1、什么是索引

**索引 [下标]：** 用来访问数组元素的序号【数组下标从`0`开始】，如果访问时数组没有和索引值对应的元素，则得到的值是`undefined`。

```
var arr = ['kobe', '22',33,'GuardCode']
    索引号：    0      1    2    3
```

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>数组的使用</title>
</head>
<body>
    <script type="text/javascript">
        // 利用new 创建数组
        var arr = new Array(); // 创建了一个空的数组

        // 利用数组字面量创建数组 []
        var arr = []; // 创建了一个空的数组
        var arr1 = [1, 2, 'pink', true];
        // 获取数组元素  格式 数组名[索引号]  索引号从 0开始
        console.log(arr1);
        console.log(arr1[2]); // pink
        console.log(arr1[3]); // true

        // 定义一个新数组
```

```
        var arr2 = ['11', '吉安特', 'string'];
        console.log(arr2[0]);
        console.log(arr2[1]);
        console.log(arr2[2]);
        console.log(arr2[3]); // 因为没有这个数组元素 所以输出的结果是 undefined
    </script>
</body>
</html>
```

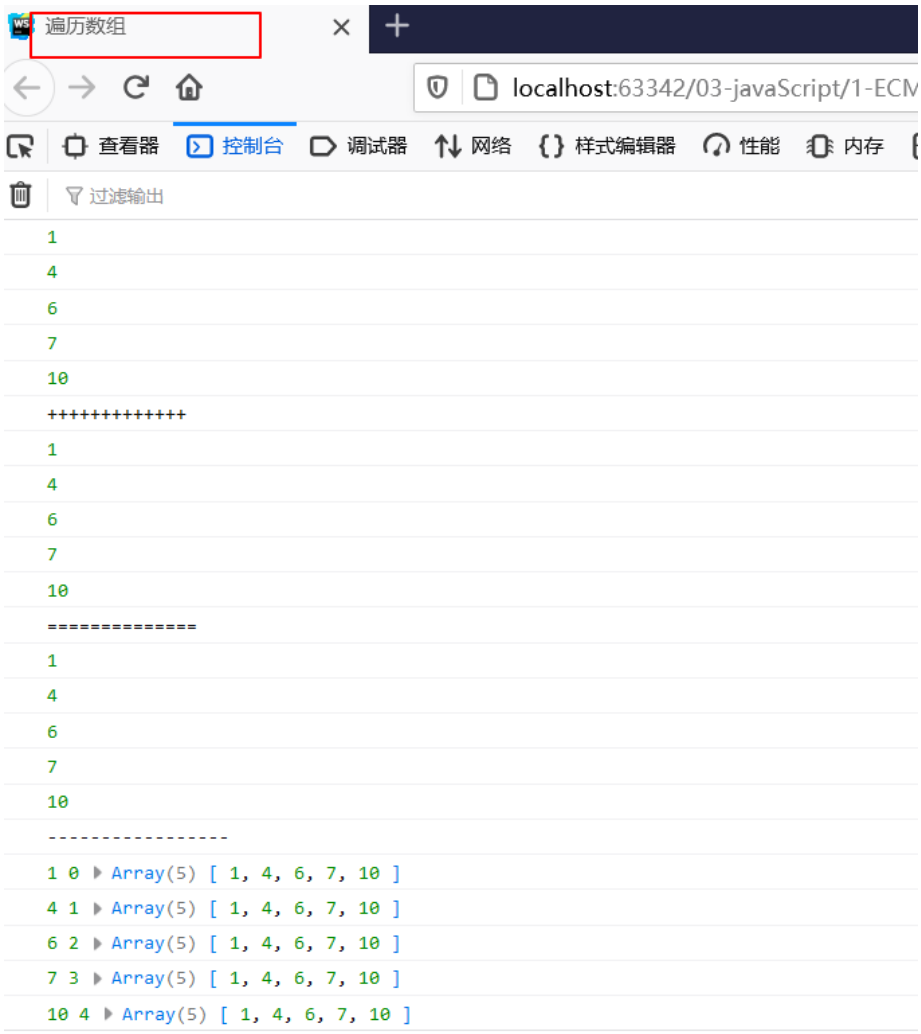
## 3.4 遍历数组

把数组中的每个元素从头到尾都访问一次，可以通过`for`循环索引遍历数组中的每一项。

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>遍历数组</title>
  <script type="text/javascript">
    // 1.利用传统循环来遍历数组
    let arr1 = [1, 4, 6, 7, 10];
    // 遍历数组
    for (let i=0; i<arr1.length; i++){
      console.log(arr1[i]);
    }

    console.log("++++++++++++++");
    // 2.利用ES6中推出的for of循环来遍历数组
    for(let value of arr1){
      console.log(value);
    }

    console.log("===============");
    // 3.用Array对象的forEach方法来遍历数组
    arr1.forEach(function (currentValue, currentIndex, currentArray){
      console.log(currentValue);
    });

    console.log("-----------------");
    // forEach实现
    Array.prototype.myForEach = function (func){
      // this === [1, 4, 6, 7, 10];
      for(let i=0; i<this.length; i++){
        func(this[i], i, this);
      }
    };
    arr1.myForEach(function (currentValue, currentIndex, currentArray){
      console.log(currentValue, currentIndex, currentArray);
    });
  </script>
</head>
<body>

</body>
</html>
```

2、执行结果




## 3.5 数组案例

### 3.5.1 数组的平均值

1、案例实现

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>计算平均值</title>
</head>
<body>
    <script>
        // 定义数组
        var arr = [2, 6, 1, 7, 4];
        // 声明一个求和变量 sum。
        var sum = 0;
        // 声明一个平均值变量averge
        var average = 0;
        for (var i = 0; i < arr.length; i++) {
            // 遍历这个数组，把里面每个数组元素加到 sum 里面。
            sum += arr[i];
        }
        // 用求和变量 sum 除以数组的长度就可以得到数组的平均值。
        average = sum / arr.length;
```

```
        console.log(sum, average);
    </script>
</body>
</html>
```

### 3.5.2 数组新增元素

数组中可以通过以下方式在数组的末尾插入新元素：数组[数组`.length`]=新数据

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>新增数组元素</title>
</head>
<body>
    <script type="text/javascript">
        // 1. 新增数组元素 修改length长度
        var arr = ['red', 'green', 'blue'];
        console.log(arr.length);
        arr.length = 5; // 把我们数组的长度修改为了5里面应该有5个元素
        console.log(arr);
        console.log(arr[3]); // undefined
        console.log(arr[4]); // undefined

        // 2. 新增数组元素 修改索引号 追加数组元素
        var arr1 = ['red', 'green', 'blue'];
        arr1[3] = 'pink';
        console.log(arr1);
        arr1[4] = 'hotpink';
        console.log(arr1);
        arr1[0] = 'yellow'; // 这里是替换原来的数组元素
        console.log(arr1);
        arr1 = '有点意思';
        console.log(arr1); // 不要直接给 数组名赋值 否则里面的数组元素都没有了
    </script>
</body>
</html>
```

### 3.5.3 筛选数组

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>筛选数组</title>
</head>
<body>
    <script type="text/javascript">
        /*
        将数组 [2, 0, 6, 1, 77, 0, 52, 0, 25, 7] 中大于等于 10 的元素选出来，放入新数
组。
```

```
        声明一个新的数组用于存放新数据newArr。遍历原来的旧数组， 找出大于等于 10 的元素。依次追加给新数组 newArr。
        */

        // 方法1
        var arr = [2, 0, 6, 1, 77, 0, 52, 0, 25, 7];
        var newArr = [];
        var j = 0;
        for (var i = 0; i < arr.length; i++) {
            if (arr[i] >= 10) {
                // 新数组索引号应该从0开始 依次递增
                newArr[j] = arr[i];
                j++;
            }
        }
        console.log(newArr);

        // 方法2
        var arr = [2, 0, 6, 1, 77, 0, 52, 0, 25, 7];
        var newArr = [];
        // 刚开始 newArr.length 就是 0
        for (var i = 0; i < arr.length; i++) {
            if (arr[i] >= 10) {
                // 新数组索引号应该从0开始 依次递增
                newArr[newArr.length] = arr[i];
            }
        }
        console.log(newArr);
    </script>
</body>
</html>
```

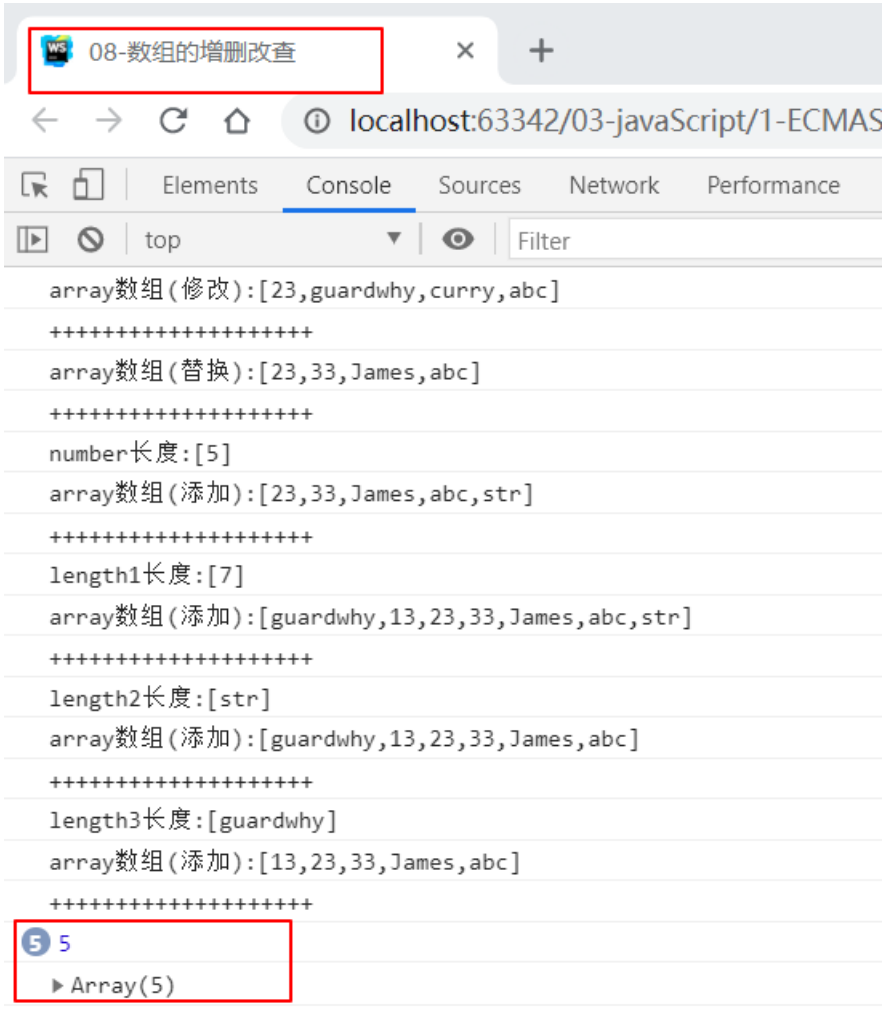
### 3.5.4 数组的增删改查

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>08-数组的增删改查</title>
  <script type="text/javascript">
    // 1.1定义数组
    let array = ["23", "kobe", "curry", "abc"];
    // 1.2 将索引为1的数据修改为guardwhy
    array[1] = "guardwhy";
    console.log("array数组(修改):" + "[" + array + "]");
    console.log("++++++++++++++++++++");

    // 2.1 替换数组元素
    array.splice(1, 2, "33", "James");
    console.log("array数组(替换):" + "[" + array + "]");
    console.log("++++++++++++++++++++");

    // 3.在数组最后添加一条数据
    let number = array.push("str");
    console.log("number长度:" + "[" + number + "]");
```

```
    console.log("array数组(添加):" + "[" + array + "]");
    console.log("++++++++++++++++++++");

    // 4.在数组最前面添加两条数据
    // unshift方法和push方法一样，会将新增内容之后当前数组的长度返回.
    let length1 = array.unshift("guardwhy","13");
    console.log("length1长度:"+ "[" + length1 + "]");
    console.log("array数组(添加):" + "[" + array + "]");
    console.log("++++++++++++++++++++");

    // 5.删除数组最后一条数据。
    let length2 = array.pop();
    console.log("length2长度:"+ "[" + length2 + "]");
    console.log("array数组(添加):" + "[" + array + "]");
    console.log("++++++++++++++++++++");

    // 6.删除数组最前面一条数据
    let length3 = array.shift();
    console.log("length3长度:"+ "[" + length3 + "]");
    console.log("array数组(添加):" + "[" + array + "]");

    // 7.遍历的同时删除数组中所有元素
    for(let i=0; i<array.length; i++){
      console.log(array.length);
      // 注意点: 通过delete来删除数组中的元素，数组的length属性不会发生变化
      delete array[i];
    }
    // 输出数组
    console.log(array);
  </script>
</head>
<body>

</body>
</html>
```

2、执行结果




### 3.5.5 数组基本方法

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>数组常用方法</title>
  <script type="text/javascript">
    // 定义数组
    let array1 = [1,2,3,4,5];
    // 1.清空数组
    array1.splice(0, array1.length);
    console.log("array1数组:" + "[" + array1 + "]");
    console.log("+++++++++++++++");

    // 2.数组转换为字符串
    let array2 = [1,2,3,4,5];
    let str = array2.toString();
    console.log("str字符串:" + str);
    console.log("str数据类型:" + typeof str);
    console.log("+++++++++++++++");

    // 3.将两个数组拼接为一个数组
    // 3.1 方式一
    let num1 = [1,3,4,5];
    let num2 = [1,3,6,9];
    // 用加号进行拼接会先转换成字符串再拼接.
```

```
    let res1 = num1.concat(num2);
    console.log("res1数组:" + res1);
    console.log("res1数组数据类型:" + typeof res1);
    console.log("====================");

    // 3.2 方式二
    let array3 = [1,10,4,6];
    let array4 = [1,5,6,11];
    /*
    扩展运算符在解构赋值中表示将剩余的数据打包成一个新的数组.
    扩展运算符在等号右边, 表示将数组中所有的数据解开, 放到所在的位置.
    */
    let res2 = [...array3, ...array4];
    console.log("res2数组:" + res1);
    console.log("res2数组数据类型:" + typeof res1);
    console.log("====================");

    // 4.对数组中的内容进行反转
    let num3 = [1,2,3,4,5];
    let res3 = num3.reverse();
    console.log("res3数组:" + res3);
    console.log("res3数组数据类型:" + typeof res3);
    console.log("====================");

    // 5.截取数组中指定范围内容
    let num4 = [1,2,3,4,5];
    // slice方法是包头不包尾(包含起始位置, 不包含结束的位置)
    let res4 = num4.slice(1,3);
    console.log("res4数组:" + res4);  // res4数组:2,3
    console.log("num4数组:" + num4);
    console.log("====================");

    // 6.查找元素在数组中的位置
    let num5 = [1, 2, 3, 4, 5, 3];
    /*
     indexOf方法如果找到了指定的元素, 就会返回元素对应的位置,如果没有找到指定的元素, 就会返
回-1.
     注意点: indexOf方法默认是从左至右的查找, 一旦找到就会立即停止查找。
    */
    let res5 = num5.indexOf(3);
    console.log("元素索引:" + res5);
    // 6.1 从什么位置开始查找
    let res6 = num5.indexOf(3, 4);
    console.log("元素索引:" + res6);    // 元素索引:5
    // 6.2 lastIndexOf方法默认是从右至左的查找, 一旦找到就会立即停止查找
    let res7 = num5.lastIndexOf(3, 4);
    console.log("元素索引:" + res7);
  </script>
</head>
<body>

</body>
</html>
```

2、执行结果

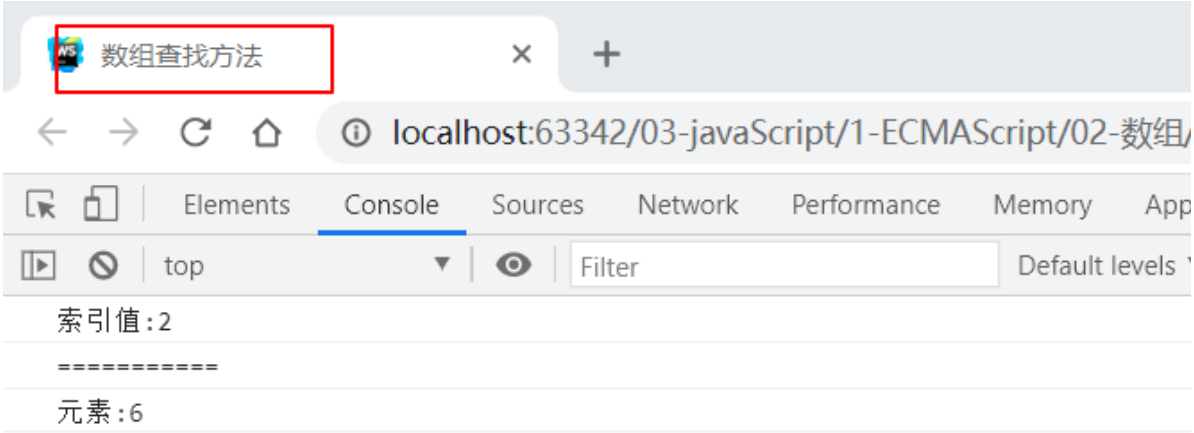

### 3.5.6 数组查找元素

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>数组查找方法</title>
  <script type="text/javascript">
    // 定义数组
    let arr = [3, 2, 6, 7, 6];
    // 1.数组的findIndex方法
    // findIndex方法: 定制版的indexOf, 找到返回索引, 找不到返回-1
    let index = arr.findIndex(function (currentValue, currentIndex,
currentArray){
      if(currentValue === 6){
        return true;
      }
    });
    console.log("索引值:" + index);

    console.log("============");
```

```
    // 2.数组的find方法
    /*
    find方法返回索引，find方法返回找到的元素。
    find方法如果找到了就返回找到的元素，如果找不到就返回undefined。
    */
    let value = arr.find(function (currentValue, currentIndex, currentArray){
      // 条件判断
      if(currentValue === 6){
        return true;
      }
    });

    // 输出结果
    console.log("元素:" + value);

  </script>
</head>
<body>

</body>
</html>
```

2、执行结果



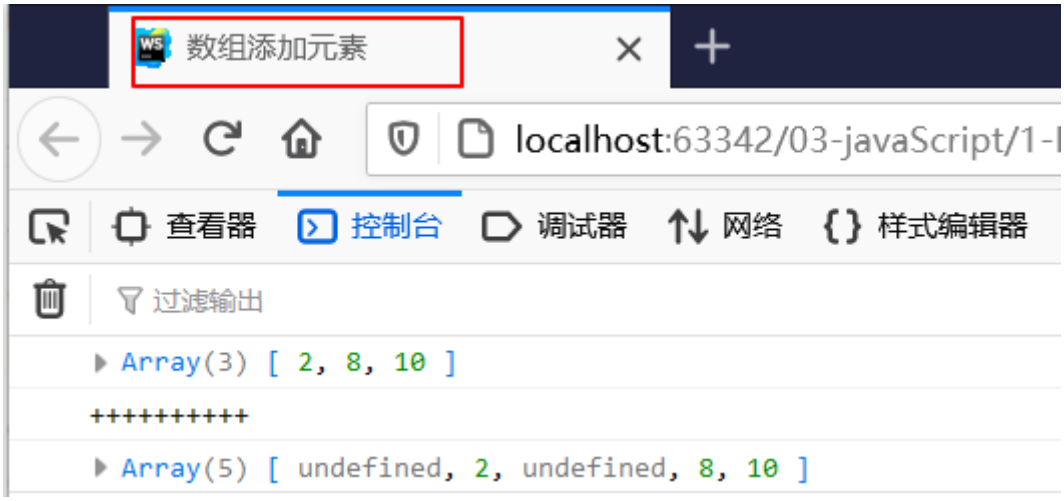

### 3.5.7 添加元素到数组

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>数组添加元素</title>
  <script type="text/javascript">
    // 定义arr数组
    let arr = [1,2,3,8,10];

    // 1.数组的filter方法:将满足条件的元素添加到一个新的数组中
    let newArray1 = arr.filter(function (currentValue, currentIndex,
currentArray){
      // 条件判断
      if(currentValue % 2 === 0){
        return true;
```

```
    }
  });
  // 输出新数组
  console.log(newArray1);

  console.log("++++++++++");

  // 2.数组的map方法:将满足条件的元素映射到一个新的数组中.
  let newArray2 = arr.map(function (currentValue, currentIndex, currentArray){
    // 条件判断
    if(currentValue % 2 === 0){
      return currentValue;
    }
  });
  // 输出新数组
  console.log(newArray2);
</script>
</head>
<body>

</body>
</html>
```

2、执行结果




## 3.6 二维数组

基本介绍

二维数组就是数组的每一个元素又是一个数组, 称之为二维数组。

### 3.6.1 操作二维数组

从二维数组中获取数据

- 数组名称`[二维数组索引]`得到一个一维数组
- 数组名称`[二维数组索引][一维数组索引]`得到一维数组中的元素

往二维数组中存储数据

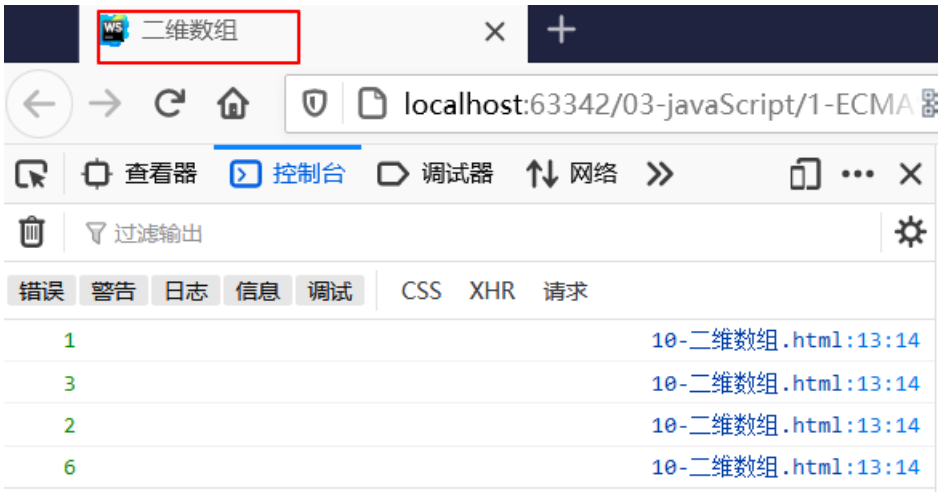
- 数组名称[二维数组索引] = 一维数组;
- 数组名称`[二维数组索引][一维数组索引]` = 值;

### 3.6.2 遍历二维数组

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>二维数组</title>
  <script type="text/javascript">
    // 1.定义数组
    let array = [[1,3], [2,6]];
    // 2.遍历二维数组
    for(let i=0; i<array.length; i++){
      let element = array[i];
      for (let j=0; j<element.length; j++){
        console.log(element[j]);
      }
    }
  </script>
</head>
<body>

</body>
</html>
```

2、执行结果




## 3.7 数组排序

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>数组排序</title>
  <script type="text/javascript">
    // 1.定义数组
    let arr = ["c", "a", "b", "d"];
    // 排序
    arr.sort();
```

```
    console.log(arr);
    console.log("++++++++++");

    // 2.定义数组
    let arr2 = [3, 4, 2, 5, 1];
    // 排序
    arr2.sort(function (a, b) {
      /*
      排序规律:如果数组中的元素是数值类型
      如果需要升序排序, 那么就返回a - b;
      如果需要降序排序, 那么就返回b - a;
      */
      // return a - b;
       return b - a;
    });
    console.log(arr2);
    console.log("==========");

    // 3.定义数组
    let arr3 = ["34", "21", "54321", "123", "6"];
    // 排序操作
    arr3.sort(function (str1, str2){
      // 升序排序
      return str1.length - str2.length;
      // return str2.length - str1.length;
    });
    console.log(arr3);
    console.log("----------");

    // 4.定义数组
    let students = [
      {name:"curry", age:10},
      {name:"kobe",age: 33},
      {name:"James", age: 36},
      {name:"Rondo",age: 34},
    ]
    // 排序操作
    students.sort(function (o1, o2){
      // 降序排序
      return o2.age - o1.age;
    });
    // 输出结果
    console.log(students);
  </script>
</head>
<body>

</body>
</html>
```

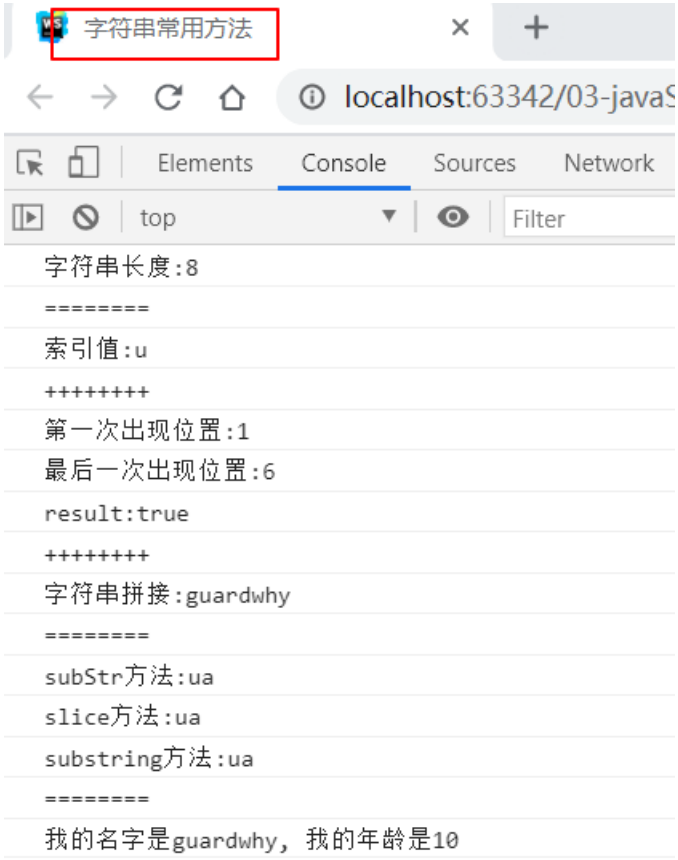
## 3.8 字符串常用方法

1、代码示例

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

```
  <meta charset="UTF-8">
  <title>字符串常用方法</title>
  <script type="text/javascript">
    // 1.获取字符串长度
    let str1 = "guardwhy";
    console.log("字符串长度:" + str1.length);
    console.log("========");

    // 2.获取某个字符 [索引]
    let str2 = "guardwhy";
    // 获取字符
    let ch = str2.charAt(1);
    console.log("索引值:" + ch);
    console.log("++++++++");

    // 3.字符串查找 indexOf / lastIndexOf / includes
    let str3 = "guardwuhy";
    let index1 = str3.indexOf("u");
    console.log("第一次出现位置:" + index1);
    let index2 = str3.lastIndexOf("u");
    console.log("最后一次出现位置:" + index2);
    let result = str3.includes("a");
    console.log("result:" + result);
    console.log("++++++++");

    // 4.字符串拼接
    let str4 = "guard";
    let str5 = "why";
    let str6 = str4 + str5;
    console.log("字符串拼接:" + str6);
    console.log("========");

    // 5.截取子串 slice / substring / substr
    let str = "guardwhy";
    let subStr1 = str.slice(1,3);
    console.log("subStr方法:"+ subStr1);
    let subStr2 = str.slice(1,3);
    console.log("slice方法:"+ subStr2);
    let subStr3 = str.substring(1,3);
    console.log("substring方法:"+ subStr3);
    console.log("========");

    // 6.字符串模板 ES6
    let name = "guardwhy";
    let age = 10;
    // 拼接
    // let myStr = "我的名字是:" + name + ",我的年龄是:" + age;
    let myStr = `我的名字是${name}, 我的年龄是${age}`;
    console.log(myStr);
  </script>
</head>
<body>

</body>
</html>
```

2、执行结果




# 4- 函数

## 4.1 函数定义

1、基本介绍

函数就是封装了一段可被重复调用执行的代码块。通过此代码块可以实现大量代码的重复使用。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>函数定义</title>
</head>
<body>
    <script>
        // 1. 求 1~100的累加和
        var sum = 0;
        for (var i = 1; i <= 100; i++) {
            sum += i;
        }
        console.log(sum);

        // 2. 求 10~50的累加和
        var sum = 0;
        for (var i = 10; i <= 50; i++) {
            sum += i;
        }
        console.log(sum);

        // 3. 函数就是封装了一段可以被重复执行调用的代码块 目的： 就是让大量代码重复使用
        function getSum(num1, num2) {
```

```
            var sum = 0;
            for (var i = num1; i <= num2; i++) {
                sum += i;
            }
            console.log(sum);
        }
        getSum(1, 100);
        getSum(10, 50);
    </script>
</body>
</html>
```

## 4.2 函数的使用

1、声明函数

`function`是声明函数的关键字,必须小写。

```
// 声明函数
function 函数名() {
    //函数体代码
}
```

2、调用函数

基本语法

```
// 调用函数
函数名();  // 通过调用函数名来执行函数体代码
```

注意点:

- 调用的时候千万不要忘记添加小括号
- 函数不调用，自己不执行。声明函数本身并不会执行代码，只有调用函数时才会执行函数体代码。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>函数的使用</title>
</head>
<body>
    <script>
        // 函数使用分为两步： 声明函数 和 调用函数
        // 1. 声明函数
        function sayHi() {
            console.log('hi~~');
        }
        // 2. 调用函数
        sayHi();
    </script>
</body>
</html>
```

3、函数的封装

函数的封装是把一个或者多个功能通过函数的方式封装起来，对外只提供一个简单的函数接口。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>函数封装</title>
</head>
<body>
    <script>
        // 利用函数计算1-100之间的累加和
        // 1. 声明函数
        function getSum() {
            var sum = 0;
            for (var i = 1; i <= 100; i++) {
                sum += i;
            }
            console.log(sum);

        }
        // 2. 调用函数
        getSum();
    </script>
</body>
</html>
```

## 4.3 函数的参数

1、函数参数语法

| 参数 | 说明 |
| --- | --- |
| 形参 | 形式上的参数，**函数定义**的时候传递的参数，当前不知道是什么，函数定义时设置接收调用时传入。 |
| 实参 | 实际上的参数，函数调用的时候传递的参数，实参是传递给形参的。函数调用时传入小括号内的真实数据。 |

2、参数的作用

在函数内部某些值不能固定，我们可以通过参数在调用函数时传递不同的值进去。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>函数参数</title>
</head>
<body>
    <script>
        // 1. 利用函数求任意两个数的和
        function getSum(num1, num2) {
            console.log(num1 + num2);

        }
```

```
        getSum(1, 3);
        getSum(3, 8);

        // 2. 利用函数求任意两个数之间的和
        function getSums(start, end) {
            var sum = 0;
            for (var i = start; i <= end; i++) {
                sum += i;
            }
            console.log(sum);

        }
        getSums(1, 100);
        getSums(1, 10);
        // 3. 注意点
        // (1) 多个参数之间用逗号隔开
        // (2) 形参可以看做是不用声明的变量
    </script>
</body>
</html>
```

3、形参和数量不匹配

| 参数个数 | 说明 |
| --- | --- |
| 实参个数等于形参个数 | 输出正确的结果 |
| 实参个数大于形参个数 | 只能取到形参的个数 |
| 实参个数小于形参个数 | 多的形参定义为undefined，结果是NaN |

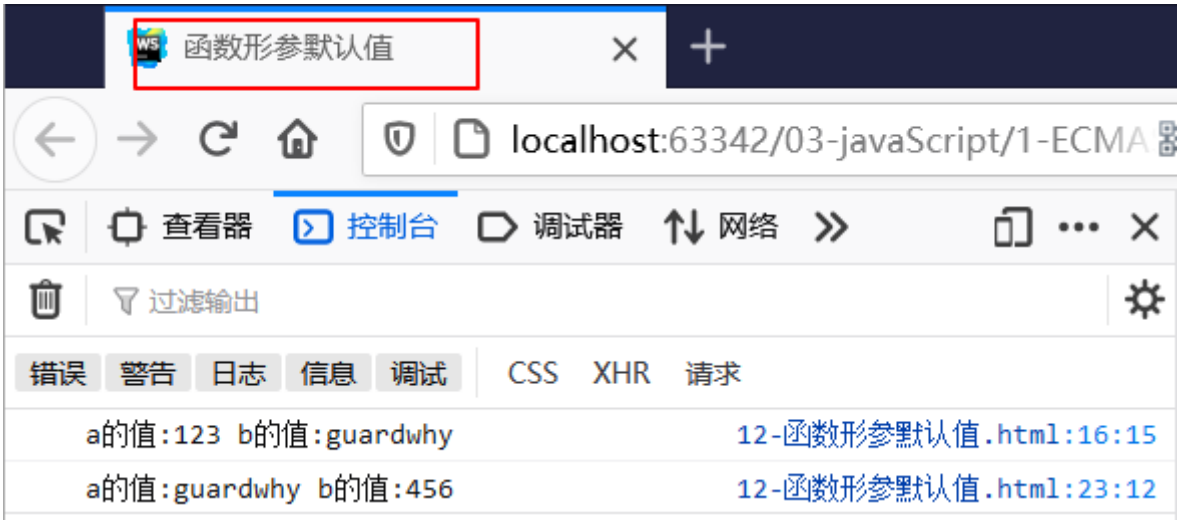
4、函数形参默认值

**代码示例**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>函数形参默认值</title>
  <script type="text/javascript">
    // 1.在ES6之前可以通过逻辑运算符来给形参指定默认值
    function getFun(a,b){
      /*
        格式: 条件A || 条件B
        如果条件A成立, 那么就返回条件A
        如果条件A不成立, 无论条件B是否成立, 都会返回条件B
      */
      a = a || "abc";
      b = b || "James";
      console.log("a的值:" + a, "b的值:" + b);
    }
    // 调用函数
    getFun(123, "guardwhy");

    // 2. 从ES6开始, 可以直接在形参后面通过=指定默认值
    function getFun2(a="guardwhy", b=getDefault()){
```

```
      console.log("a的值:"+ a, "b的值:"+b);
    }
    // 调用函数
    getFun2();
    function getDefault(){
      return "456"
    }
  </script>
</head>
<body>
</body>
</html>
```

**执行结果**




5、总结

- 函数可以带参数也可以不带参数
- 声明函数的时候，函数名括号里面的是形参，形参的默认值为 undefined
- 调用函数的时候，函数名括号里面的是实参
- 多个参数中间用逗号分隔。形参的个数可以和实参个数不匹配，但是结果不可预计，应该尽量要匹配。

**代码示例**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>函数形参实参个数匹配</title>
</head>
<body>
    <script>
        // 函数形参实参个数匹配
        function getSum(num1, num2) {
            console.log(num1 + num2);
        }
        // 1. 如果实参的个数和形参的个数一致 则正常输出结果
        getSum(1, 2);
        // 2. 如果实参的个数多于形参的个数   会取到形参的个数
        getSum(1, 2, 3);
```

```
        // 3. 如果实参的个数小于形参的个数   多于的形参定义为undefined   最终的结果就是 NaN
        // 形参可以看做是不用声明的变量   num2 是一个变量但是没有接受值   结果就是undefined
        getSum(1); // NaN
    </script>
</body>
</html>
```

## 4.4 函数的返回值

1、return 语句

返回值：函数调用整体代表的数据，函数执行完成后可以通过`return`语句将指定数据返回 。

```
// 声明函数
function 函数名(){
    ...
    return   需要返回的值;
}
// 调用函数
函数名();    // 此时调用函数就可以得到函数体内return 后面的值
```

注意点：

- 在使用`return`语句时，函数会停止执行，并返回指定的值。
- 如果函数没有`return`，返回的值是`undefined`。

2、关键字的区别

- `break`：结束当前的循环体`(如 for、while)`。
- `continue`：跳出本次循环，继续执行下次循环`(如 for、while)`。
- `return`：不仅可以退出循环，还能够返回`return`语句中的值，同时还可以结束当前的函数体内的代码。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>函数返回值</title>
</head>
<body>
    <script>
        // 利用函数求数组 [5,2,99,101,67,77] 中的最大数值。
        function getArrMax(arr) {
            // 定义变量
            var max = arr[0];
            for (var i = 1; i <= arr.length; i++) {
                if (arr[i] > max) {
                    max = arr[i];
                }
            }
            return max;
        }
        // 定义数组
        var re = getArrMax([3, 77, 44, 99, 143]);
        console.log(re);
    </script>
</body>
```

```
</html>
```

3、函数扩展运算符

将传递给函数的所有实参打包到一个数组中，注意点: 和在等号左边一样, 也只能写在形参列表的最后。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>ES6函数拓展运算符</title>
  <script type="text/javascript">
    /*
    扩展运算符在等号右边，将数组中的数据解开
        let arr1 = [1, 6, 5];
        let arr2 = [11, 4, 8];
        let array = [...arr1, ...arr2]; let array = [1, 6, 5, 11, 4, 8];
    */
    function getSum(...values){
      // 定义sum值
      let sum = 0;
      for(let i=0; i<values.length; i++){
        let element = values[i];
        sum += element;
      }
      return sum;
    }
    // 调用函数
    let array = getSum(10, 20, 30, 40);
    console.log("array元素总和:" + array); // array元素总和:100
  </script>
</head>
<body>

</body>
</html>
```

## 4.5 arguments对象

1、基本介绍

- 所有函数都内置了一个 `arguments` 对象，`arguments` 对象中存储了传递的所有实参。
- 具有 `length` 属性，按索引方式储存数据，`arguments` 展示形式是一个伪数组，因此可以进行遍历。
- 不具有数组的 `push` ，`pop` 等方法，在函数内部使用该对象，用此对象获取函数调用时传的实参。

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>arguments使用</title>
</head>
<body>
    <script>
```

```
        // 利用函数求任意个数的最大值
        function getMax() {
            // 定义最大值
            var max = arguments[0];
            for (var i = 1; i < arguments.length; i++) {
                if (arguments[i] > max) {
                    max = arguments[i];
                }
            }
            return max;
        }

        // 输出结果
        console.log(getMax(1, 2, 3));
        console.log(getMax(1, 2, 3, 4, 5));
        console.log(getMax(11, 2, 34, 444, 5, 100));
    </script>
</body>
</html>
```

## 4.6 函数声明方式

1、命名函数

利用函数关键字`function`自定义函数方式。

注意：调用函数的代码既可以放到声明函数的前面，也可以放在声明函数的后面。

```
function 函数名(参数) {
   方法体;
   [return 返回值]
}
```

代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>命名函数</title>
</head>
<body>
<script type="text/javascript">
    /*//定义函数
    function hello() {
        alert(1);
        alert(2);
    }

    //函数不调用是不执行的
    hello();
    alert(3);*/

    //定义一个函数：实现2个数的和
    function sum(a, b) {  //形参
        var result = a + b;
        return result;
```

```
    }

    //调用函数：使用的是实参
    var result = sum(3,5);
    document.write("计算结果是：" + result + "<br/>");
</script>
</body>
</html>
```

2、匿名函数

匿名函数就是没有名称的函数

```
//定义匿名函数
var 变量名 = function(形参) {
   return 返回值;
}
```

匿名函数特点

- 函数表达式方式原理跟声明变量方式是一致的，函数调用的代码必须写到函数体后面。
- 函数以后主要用在事件处理函数中，通常不能重用。使用一次，如果需要重用建议使用命名函数。如果要使用匿名函数，也可以将函数赋值给一个变量。

代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>匿名函数使用</title>
</head>
<body>
<script type="text/javascript">
    //将一个匿名的函数赋值为一个变量
    var sum = function (a,b) {
        var result = a + b;
        return result;
    };

    //通过变量名来调用函数
    var result = sum(2,7);
    document.write(result + "<br/>");
</script>
</body>
</html>
```

3、递归函数

基本介绍

- 递归函数就是在函数中自己调用自己,递归函数在一定程度上可以实现循环的功能。
- 每次调用递归函数都会开辟一块新的存储空间, 所以性能不是很好。

代码示例

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>递归函数</title>
  <script type="text/javascript">
    // 声明函数
    function login(){
      // 1.接收用户输入密码
      let pwd = prompt("请输入密码");
      // 2.判断密码是否正确
      if(pwd != "guardwhy"){
        login();
      }
      alert("欢迎再次登录");
    }

    // 调用函数
    login();
  </script>
</head>
<body>

</body>
</html>
```

## 4.7 函数实现案例

### 1、平均值示例

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>求平均值</title>
</head>
<body>
    <script>
       // 求任意一组数的平均值
       function getAvg(){
          // 多所有参数进行求和
          var sum=0,len=arguments.length,i;
          for(i=0;i<len;i++){
             sum+=arguments[i];
          }
          return sum/len;
       }
       var avg=getAvg(5,66,45,32,88,24,40,199,3900);
       console.log(avg);
    </script>
</body>
</html>
```

### 2、函数翻转数组

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

```
<head>
    <meta charset="UTF-8">
    <title>数组反转</title>
</head>
<body>
    <script>
        // 利用函数翻转任意数组 reverse 翻转
        function reverse(arr) {
            // 创建一个空数组
            var newArr = [];
            for (var i = arr.length - 1; i >= 0; i--) {
                newArr[newArr.length] = arr[i];
            }
            return newArr;
        }
        // 定义arr1数组
        var arr1 = reverse([1, 3, 4, 6, 9]);
        console.log(arr1);
        // 定义arr2数组
        var arr2 = reverse(['red', 'pink', 'blue']);
        console.log(arr2);
    </script>
</body>
</html>
```

### 3、函数冒泡排序

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>数组排序</title>
</head>
<body>
    <script>
        // 利用函数冒泡排序 sort 排序
        function sort(arr) {
            for (var i = 0; i < arr.length - 1; i++) {
                for (var j = 0; j < arr.length - i - 1; j++) {
                    if (arr[j] > arr[j + 1]) {
                        var temp = arr[j];
                        arr[j] = arr[j + 1];
                        arr[j + 1] = temp;
                    }
                }
            }
            return arr;
        }
        // 定义数组arr1
        var arr1 = sort([1, 4, 2, 9]);
        console.log(arr1);
        // 定义数组arr2
        var arr2 = sort([11, 7, 22, 999]);
        console.log(arr2);
    </script>
</body>
</html>
```

## 4.8 变量作用域

就是代码名字(变量)在某个范围内起作用和效果,目的是为了提高程序的可靠性更重要的是减少命名冲突。

**1、全局作用域**

作用于所有代码执行的环境(整个script标签内部)或独立的js文件。

**2、局部作用域**

- 作用于函数内的代码环境，就是局部作用域。
- 在`JavaScript`中函数后面`{ }`中的的作用域, 就是局部作用域。

**3、块级作用域**

在`ES6`中只要`{ }`没有和函数结合在一起, 那么应该"块级作用域"。

**4、块级作用域和局部作用域区别**

- 在块级作用域中通过`var`定义的变量是全局变量。
- 在局部作用域中通过`var`定义的变量是局部变量。
- 无论是在块级作用域还是在局部作用域, 省略变量前面的`let`或者`var`就会变成一个全局变量。

**5、代码示例**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>变量作用域</title>
</head>
<body>
    <script>
        /*
        在JavaScript中定义变量有两种方式
            ES6之前: var 变量名称;
            ES6开始: let 变量名称;
        */

        // 1.全局作用域：整个script标签 或者是一个单独的js文件
        var num = 10;
        var num = 30;
        console.log(num);

        // 2.局部作用域（函数作用域） 在函数内部就是局部作用域 这个代码的名字只在函数内部起效果和作用
        function fn() {
            // 局部作用域
            var num = 20;
            console.log(num);

        }
        fn();

        // 3.块级作用域
        {
         var num3 = 11;
        }
```

```
        console.log("num3的值:" + num3);

        // 4.let定义的变量放到一个单独的{}里面，是一个局部变量
        {
          let num4 = 123;
        }
        // console.log("num4的值:" + num4); // 局部变量，报错
    </script>
</body>
</html>
```

### 4.8.1 全局变量

在全局作用域下声明的变量叫做全局变量【在函数外部定义的变量】

- 全局变量在代码的任何位置都可以使用。
- 在全局作用域下`var`声明的变量 是全局变量。
- 特殊情况下，在函数内不使用`var`声明的变量也是全局变量。

### 4.8.2 局部变量

在局部作用域下声明的变量叫做局部变量【在函数内部定义的变量】

- 局部变量只能在该函数内部使用。
- 在函数内部`var`声明的变量是局部变量。
- 函数的形参实际上就是局部变量。

> **两者区别**

- 全局变量：在任何一个地方都可以使用，只有在浏览器关闭时才会被销毁，因此比较占内存。
- 局部变量：只在函数内部使用，当其所在的代码块被执行时，会被初始化，当代码块运行结束后就会被销毁，因此更节省内存空间。

**代码示例**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>变量的作用域</title>
</head>
<body>
    <script>
        // 1. 全局变量：在全局作用域下的变量,在全局下都可以使用
        // 注意 如果在函数内部 没有声明直接赋值的变量也属于全局变量
        var num = 10; // num就是一个全局变量
        console.log(num);

        function fn() {
            console.log(num);

        }
        fn();

        // 2. 局部变量    在局部作用域下的变量    后者在函数内部的变量就是 局部变量
        // 注意:函数的形参也可以看做是局部变量
        function fun(aru) {
            var num1 = 10; // num1就是局部变量 只能在函数内部使用
```

```
            num2 = 20;  // 全局变量
        }
        fun();
        // 3. 从执行效率来看全局变量和局部变量
        // (1) 全局变量只有浏览器关闭的时候才会销毁，比较占内存资源
        // (2) 局部变量 当我们程序执行完毕就会销毁， 比较节约内存资源
    </script>
</body>
</html>
```

## 4.9 作用域链

内部函数访问外部函数的变量，采取的是链式查找的方式来决定取那个值这种结构。

**代码示例**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>作用域链</title>
  <script type="text/javascript">
    /*
        1.1 ES6之前定义变量通过var
        1.2 ES6之前没有块级作用域，只有全局作用域和局部作用域
        1.3 ES6之前函数大括号外的都是全局作用域
        1.4 ES6之前函数大括号中的都是局部作用域

      2.ES6之前作用域链
        2.1.全局作用域我们又称之为0级作用域
        2.2.定义函数开启的作用域就是1级/2级/3级/...作用域
        2.3.JavaScript会将这些作用域链接在一起形成一个链条，这个链条就是作用域链
              0  --->  1 ---->  2  ---->  3 ----> 4
        2.4.除0级作用域以外，当前作用域级别等于上一级+1

      3.变量在作用域链查找规则
        3.1先在当前找，找到就使用当前作用域找到的
        3.2如果当前作用域中没有找到，就去上一级作用域中查找
        3.3以此类推直到0级为止，如果0级作用域还没找到，就报错
     */

    // 1.ES6之前
    // 定义变量
    var num1 = 23;  // 全局作用域 / 0级作用域

    // 定义fun1函数
    function fun1(){
      // 1级作用域
      var num1 = 456;
      function test(){
        // 2级作用域
        var num1 = 33;
        console.log("num值:" + num1);    // num值:33
      }
      // 调用函数
      test();
    }
```

```
    // 调用fun1函数
    fun1();


    // 2.ES6
    /*
      1.1 ES6定义变量通过let.
      1.2 ES6除了全局作用域、局部作用域以外，还新增了块级作用域.
      1.3 ES6虽然新增了块级作用域，但是通过let定义变量并无差异(都是局部变量).

      2.ES6作用域链
        2.1.全局作用域我们又称之为0级作用域
        2.2.定义函数或者代码块都会开启的作用域就是1级/2级/3级/...作用域
        2.3.JavaScript会将这些作用域链接在一起形成一个链条，这个链条就是作用域链
          0  --->  1 ---->  2  ---->  3 ----> 4
        2.4.除0级作用域以外，当前作用域级别等于上一级+1

        3.变量在作用域链查找规则
        3.1先在当前找，找到就使用当前作用域找到的
        3.2如果当前作用域中没有找到，就去上一级作用域中查找
        3.3以此类推直到0级为止，如果0级作用域还没找到，就报错
     */

    // 全局作用域 / 0级作用域
    let num2 = 123;
    {
      // 1 级作用域
      let num2 = 45;
      function test2(){
        // 2 级作用域
        let num2 = 6;
        console.log("num2值:" + num2);
      }
      // 调用函数
      test2();
    }
  </script>
</head>
<body>

</body>
</html>
```

## 4.10 预解析

> **什么是预解析**

浏览器在执行`JS`代码的时候会分成两部分操作：预解析以及逐行执行代码，也就是说浏览器不会直接执行代码, 而是加工处理之后再执行。

> **预解析**

在当前作用域下, JS 代码执行之前，浏览器会默认把带有`var( let)`和`function`声明的变量在内存中进行提前声明或者定义。
预解析也叫做变量、函数提升。

> **代码执行**

从上到下执行 JS 语句。
注意：预解析会把变量和函数的声明在代码执行之前执行完成。

### 4.10.1变量预解析

变量的声明会被提升到当前作用域的最上面，变量的赋值不会提升。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>预解析</title>
  <script type="text/javascript">
    console.log(num);  // undefined,变量提升只提升声明，不提升赋值
    let num = 10;

     // 通过let定义的变量不会被提升(不会被预解析)
      // console.log(num); // 报错
      let num = 456;
  </script>
</head>
<body>

</body>
</html>
```

### 4.10.2 函数预解析

函数的声明会被提升到当前作用域的最上面，但是不会调用函数。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>预解析</title>
  <script type="text/javascript">
    /*
    1.ES6之前定义函数的格式
    */
    // 1.1 命名函数
    func1();
    // ES6之前的这种定义函数的格式，是会被预解析的，所以可以提前调用
    function func1(){
      console.log("guardwhy is mvp!!!");
    }

    /*
    // 预解析之后的代码
    function func1(){
      console.log("guardwhy is mvp!!!");
    }

    func1();
    */

    // 2.匿名函数
    // func2(); // fun2 is not a function
```

```
    // 如果将函数赋值给一个var定义的变量，那么函数不会被预解析，只有变量会被预解析
    var func2 = function (){
      console.log("kobe is mvp!!!");
    }

    /*
      var func2;    //undefined
      func2();
      func2 = function (){
        console.log("kobe is mvp!!!");
      }
    */

    // 3.ES6定义函数的格式
    func3();    // func3 is not defined
    let func3 = () =>{
      console.log("james is mvp!!!");
    }

  </script>
</head>
<body>

</body>
</html>
```

### 4.10.3 函数表达式声明函数

函数表达式创建函数，会执行变量提升！！！

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>预解析</title>
  <script type="text/javascript">
    func();
    var  func = function() {
      console.log('我太难了');  // 结果：报错提示 "func is not a function"
    }
  </script>
</head>
<body>
    <!--该段代码执行之前，会做变量声明提升，fn在提升之后的值是undefined；
        而fn调用是在fn被赋值为函数体之前，此时fn的值是undefined，所以无法正确调用。
    -->
</body>
</html>
```

# 5- 面向对象

在 `JavaScript` 中，对象是一组无序的相关属性和方法的集合，所有的事物都是对象，例如字符串、数值、数组、函数等。

## 5.1 创建对象

对象是由属性和方法组成的。

- 属性：事物的特征，在对象中用属性来表示。
- 方法：事物的行为，在对象中用方法来表示。

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>创建默认对象</title>
  <script type="text/javascript">
    /*
    1.JavaScript中提供了一个默认的类Object，可以通过这个类来创建对象
    2.由于是使用系统默认的类创建的对象，所以系统不知道想要什么属性和行为，所以必须手动的添加想要的属性和行为
    3.如何给一个对象添加属性对象名称.属性名称 = 值;
    4.如何给一个对象添加行为对象名称.行为名称 = 函数;
    */

    // 1.创建对象(方式一)
    let obj1 = new Object();
    obj1.name = "guardwhy";
    obj1.age = 21;
    obj1.func1 = function (){
      console.log("属性:" + "javaScript最牛逼");
    }
    // 输出属性
    console.log("姓名:"+ obj1.name);
    console.log("年龄:" + obj1.age);
    // 调用方法
    obj1.func1();

    console.log("++++++++++++++++++++");

    // 2.创建对象(方式2)
    let obj2 = {};
    obj2.name = "guardwhy";
    obj2.age = 21;
    obj2.func2 = function (){
      console.log("属性:" + "javaScript最牛逼");
    }
    // 输出属性
    console.log("姓名:"+ obj2.name);
    console.log("年龄:" + obj2.age);
    // 调用方法
    obj2.func2();

    console.log("========================");

    // 3.创建对象(方式3)
    // 注意点: 属性名称和取值之间用冒号隔开，属性和属性之间用逗号隔开
    let obj3 = {
      name:"guardwhy",
```

```
      age: 21,
      func3: function (){
        console.log("属性:" + "javaScript最牛逼");
      }
    };
    // 输出属性
    console.log("姓名:"+ obj3.name);
    console.log("年龄:" + obj3.age);
    // 调用方法
    obj3.func3();
  </script>
</head>
<body>

</body>
</html>
```

## 5.2 函数和方法区别

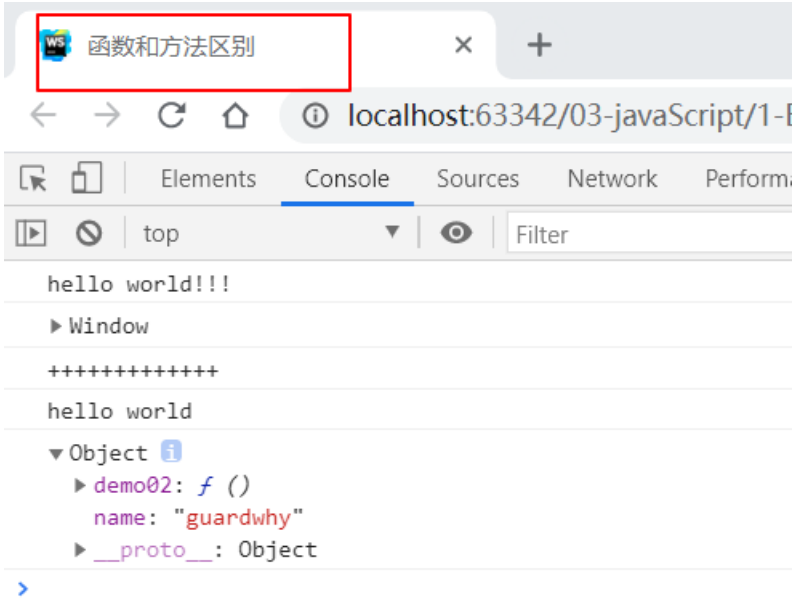
函数可以直接调用, 但是方法不能直接调用, 只能通过对象来调用，函数内部的`this`输出的是`window`，方法内部的`this`输出的是当前调用的那个对象。

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>函数和方法区别</title>
  <script type="text/javascript">
    /*
    1.什么是函数?
    函数就是没有和其它的类显示的绑定在一起的, 我们就称之为函数

    2.什么是方法?
    方法就是显示的和其它的类绑定在一起的, 我们就称之为方法。

    4.无论是函数还是方法, 内部都有一个this
    this是什么? 谁调用了当前的函数或者方法, 那么当前的this就是谁
    */
    // 1.函数
    function demo01(){
      console.log("hello world!!!")
      console.log(this);
    }
    // 调用函数
    window.demo01();

    console.log("++++++++++++++");

    // 2.方法
    let obj = {
      name:"guardwhy",
      demo02: function (){
        console.log("hello world");
        console.log(this);
      }
```

```
    };
    // 对象调用方法
    obj.demo02();
  </script>
</head>
<body>

</body>
</html>
```

2、执行结果




## 5.3 工厂函数

1、基本介绍

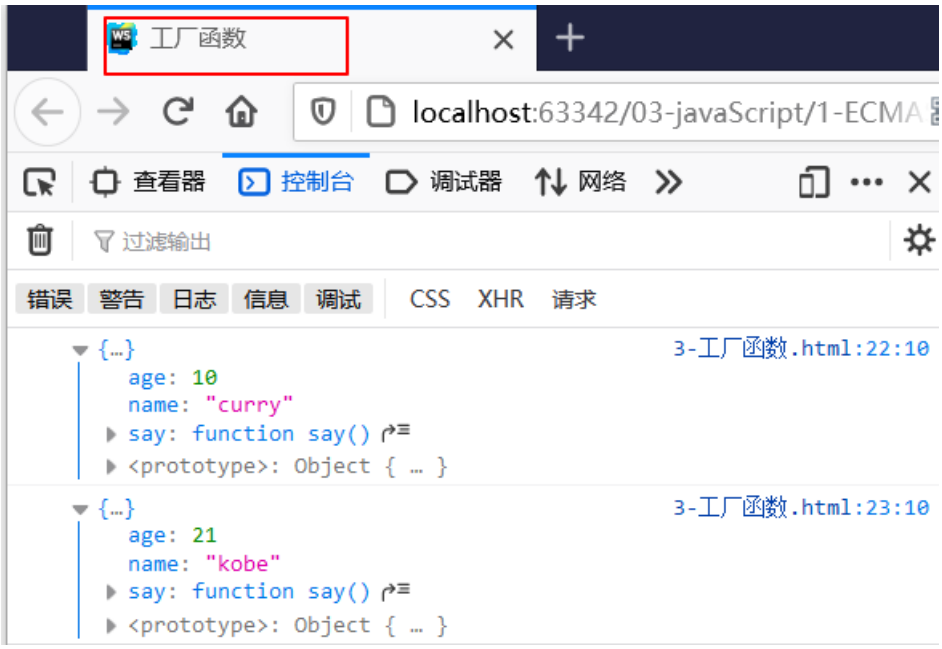
工厂函数就是专门用于创建对象的函数

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>工厂函数</title>
  <script type="text/javascript">
    function Person(Name, Age){
      // 创建对象
      let obj = new Object();
      obj.name = Name;
      obj.age = Age;
      obj.say = function (){
        console.log("hello javaScript");
      }
      return obj;
    }

    // 创建对象
    let obj1 = new Person("curry", 10);
    let obj2 = new Person("kobe", 21);
```

```
    // 输出对象
    console.log(obj1);
    console.log(obj2);
  </script>
</head>
<body>

</body>
</html>
```

3、执行结果




# 5.3 构造函数

## 5.3.1 什么是构造函数

1、基本概念

主要用来初始化对象，即为对象成员变量赋初始值，它总与 new 运算符一起使用。

可以把对象中一些公共的属性和方法抽取出来，然后封装到这个函数里面。**构造函数本质上是工厂函数的简写。**

```
function 构造函数名(形参1,形参2,形参3) {
      this.属性名1 = 参数1;
      this.属性名2 = 参数2;
      this.属性名3 = 参数3;
      this.方法名 = 函数体;
}
```

构造函数的调用格式

```
var obj = new 构造函数名(实参1，实参2，实参3)
```

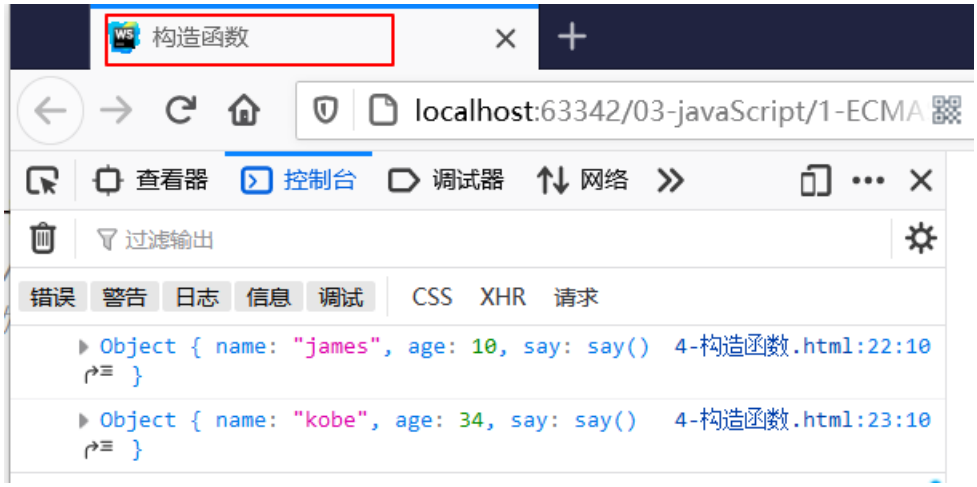
2、构造函数和工厂函数的区别

构造函数的函数名称首字母必须大写,构造函数只能够通过new来调用。

代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>构造函数创建对象</title>
  <script type="text/javascript">
    // 构造函数的函数名称首字母必须大写
    function Person(myName, myAge){
      // let obj = new Object();  // 系统自动添加的
      // let this = obj; // 系统自动添加的
      this.name = myName;
      this.age= myAge;
      this.say = function (){
        console.log("hello javaScript!!!")
      }
    }

    // 构造函数只能够通过new来调用
    let obj1 = new Person("james", 10);
    let obj2 = new Person("kobe", 34);
    // 输出属性
    console.log(obj1);
    console.log(obj2);
  </script>
</head>
<body>
    <!--
        构造函数就是一个普通的函数，创建方式和普通函数没有区别，
        不同的是构造函数习惯上首字母大写，构造函数和普通函数的区别就是调用方式的不同。
        普通函数是直接调用，而构造函数需要使用new关键字来调用。

        构造函数的执行流程：
          1.立刻创建一个新的对象
          2.将新建的对象设置为函数中this,在构造函数中可以使用this来引用新建的对象
          3.逐行执行函数中的代码
          4.将新建的对象作为返回值返回

    使用同一个构造函数创建的对象，称为一类对象，也将一个构造函数称为一个类。通过一个构造函数创建的对象，称为是该类的实例.
    -->
</body>
</html>
```

3、执行结果




### 5.3.2 new关键字

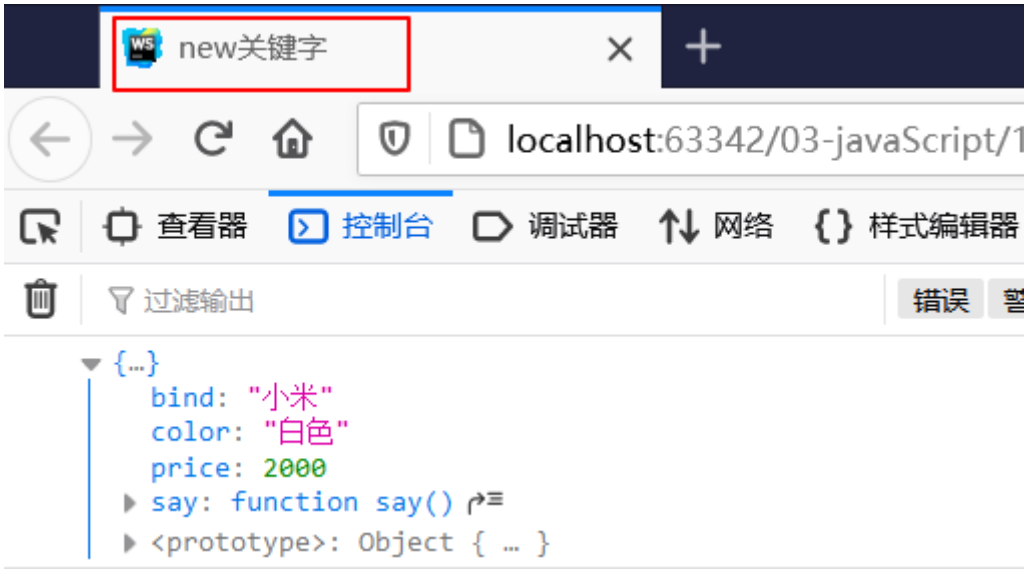
1、基本介绍

`new` 构造函数可以在内存中创建了一个空的对象， `this` 就会指向刚才创建的空对象。执行构造函数里面的代码 给这个空对象添加属性和方法， 返回这个对象。

2、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>new关键字</title>
  <script type="text/javascript">
    function Phone(bind, price, color) {
      this.bind = bind;
      this.price = price;
      this.color = color;
      this.say = function () {
        console.log("小米牛逼！！！");
      }
    }
    // 创建对象
    let obj = new Phone('小米', 2000, '白色');
    console.log(obj);
  </script>
</head>
<body>

</body>
</html>
```

3、执行结果




### 5.3.4 遍历对象

1、基本语法

```
// for...in 语句用于对数组或者对象的属性进行循环操作。
for (var k in obj) {
    console.log(k);      // 这里的 k 是属性名
    console.log(obj[k]); // 这里的 obj[k] 是属性值
}
```

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>构造函数遍历</title>
  <script type="text/javascript">
    // 遍历对象
    let obj = {
      name: 'Rondo老师',
      age: 18,
      sex: '男',
      fn: function() {}
    }
    // for (变量 in 对象) {

    // }
    for (let k in obj) {
      /*
      console.log(k); // k 变量 输出  得到的是 属性名
      console.log(obj[k]); // obj[k] 得到是 属性值
      */
      console.log(k +":" + obj[k]);
    }
  </script>
</head>
<body>

</body>
```

```
</html>
```

### 5.3.5 静态成员和实例成员

> 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 sing` 就是实例成员,实例成员只能通过实例化的对象来访问。

```
 function Person(uname, age) {
     this.uname = uname;
     this.age = age;
     this.play = function() {
     console.log('我会打球');
    }
}
var obj = new Person('kobe', 18);
console.log(obj.uname);//实例成员只能通过实例化的对象来访问
```

> 静态成员

静态成员在构造函数本身上添加的成员,如下列代码中 sex 就是静态成员,静态成员只能通过构造函数来访问。

```
 function Star(uname, age) {
     this.uname = uname;
     this.age = age;
     this.sing = function() {
     console.log('我会打球');
    }
}
Star.sex = '男';
var ldh = new Star('kobe', 18);
console.log(Star.sex);//静态成员只能通过构造函数来访问
```

### 5.3.6 构造函数缺点

1、存在问题

使用构造函数带来的最大的好处就是创建对象更方便了，但是其本身也存在一个浪费内存的问题。

```
function Person (name, age) {
  this.name = name
  this.age = age
  this.type = 'human'
  this.sayHello = function () {
    console.log('hello ' + this.name)
  }
}

var p1 = new Person('lpz', 18)
var p2 = new Person('Jack', 16)
console.log(p1.sayHello === p2.sayHello) // => false
```

2、解决方案

通过`prototype`属性进行优化。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>构造函数的优化</title>
  <script type="text/javascript">
    function Person (uName, uAge) {
      this.name = uName
      this.age = uAge
    }

    Person.prototype = {
      sayHello: function (){
        console.log('hello ' + this.name)
      }
    }

    let p1 = new Person('curry', 10)
    let p2 = new Person('james', 16)
    console.log(p1.sayHello === p2.sayHello) // => true
  </script>
</head>
<body>

</body>
</html>
```

# 6-函数对象

## 6.1 原型对象

1、基本介绍

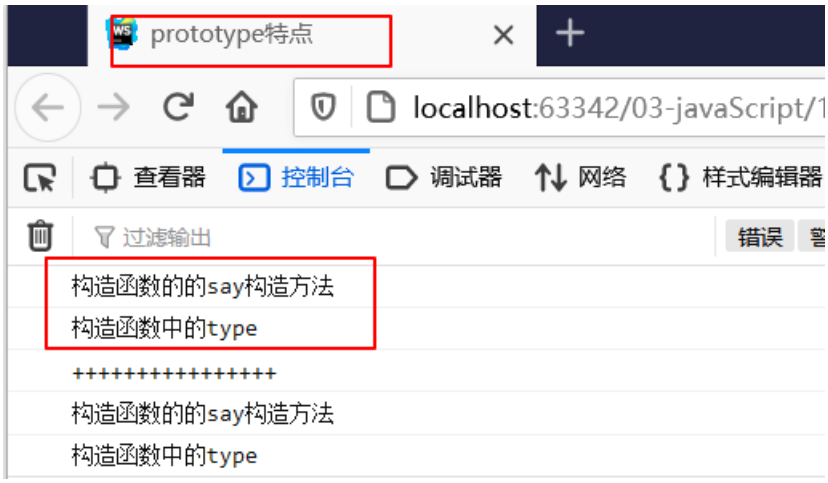
- `prototype`为原型对象。每一个构造函数都有一个`prototype`属性，指向另一个对象。
- `prototype`就是一个对象，这个对象的所有属性和方法，都会被构造函数所拥有。
- 原型对象的作用是共享方法。

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>prototype特点</title>
  <script type="text/javascript">
    function Person(myName, myAge){
      this.name = myName;
      this.age = myAge;
      this.currentType = "构造函数中的type";
      this.say = function (){
        console.log("构造函数的的say构造方法");
      }
    }
    Person.prototype = {
```

```
      currentType:"hello javaScript!!!",
      say: function (){
        console.log("hello world");
      }
    }
    // 创建对象
    let obj1 = new Person("curry", 10);
    obj1.say();
    console.log(obj1.currentType);
    console.log("+++++++++++++++++");
    let obj2 = new Person("james", 35);
    obj2.say();
    console.log(obj2.currentType);
  </script>
</head>
<body>
    <!--
    原型 prototype:
        1.每当创建的每一个函数,解析器都会向函数中添加一个属性prototype。
        这个属性对应着一个对象，这个对象就是我们所谓的原型对象。函数作为普通函数调用prototype
没有任何作用。

        2.当函数以构造函数的形式调用时，它所创建的对象中都会有一个隐含的属性，指向该构造函数的
原型对象。
          可以通过__proto__来访问该属性。

        3.原型对象就相当于一个公共的区域，所有同一个类的实例都可以访问到这个原型对象，可以将对
象中共有的内容，统一设置到原型对象中。

        4.当访问对象的一个属性或方法时，它会先在对象自身中寻找，如果有则直接使用，如果没有则会
去原型对象中寻找，如果找到则直接使用。

        5.以后创建构造函数时，可以将这些对象共有的属性和方法，统一添加到构造函数的原型对象中。
          这样不用分别为每一个对象添加，也不会影响到全局作用域，就可以使每个对象都具有这些属性
和方法了。
    -->
</body>
</html>
```

3、执行结果



# 6.2 对象原型

## 6.2.1 基本介绍

- 每个对象都会有一个属性`__ proto __`指向构造函数的`prototype`原型对象。
- 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 __`对象原型的意义为对象的查找机制提供一个方向，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`。

## 6.2.2 Constructor属性

1、基本介绍

对象原型（`__ proto __`）和构造函数原型对象（`prototype`）里面都有一个属性`constructor`属性`constructor`称为构造函数，因为它指回构造函数本身。

2、注意事项

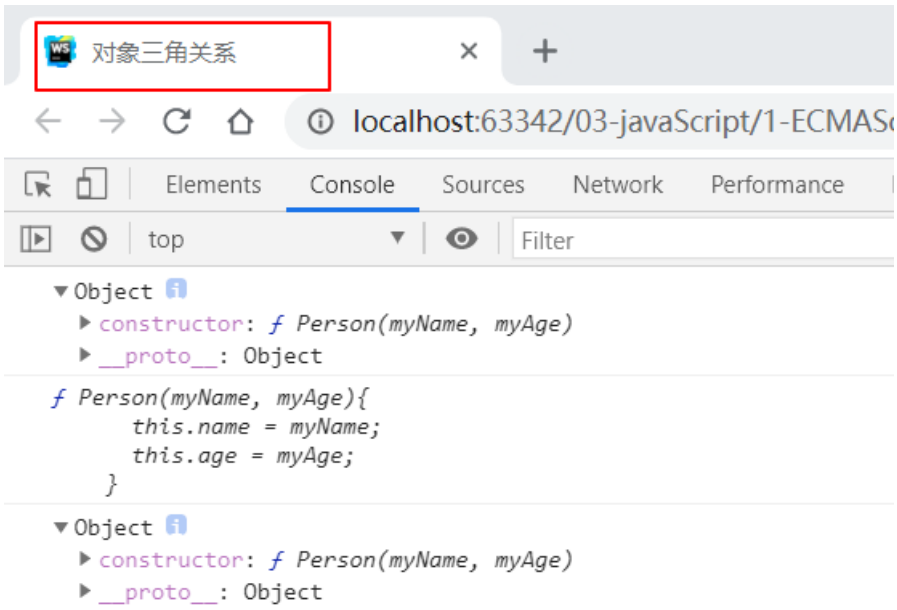
- `constructor`主要用于记录该对象引用于哪个构造函数，它可以让原型对象重新指向原来的构造函数。
- 一般情况下，对象的方法都在构造函数的原型对象中设置。如果有多个对象的方法，可以给原型对象采取对象形式赋值。但是这样就会覆盖构造函数原型对象原来的内容，这样修改后的原型对象`constructor`就不再指向当前构造函数了。
- 此时，可以在修改后的原型对象中，添加一个`constructor`指向原来的构造函数。

3、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>对象三角关系</title>
  <script type="text/javascript">
    /*
        1.每个"构造函数"中都有一个默认的属性，叫做prototype
          prototype属性保存着一个对象，这个对象我们称之为"原型对象"
        2.每个"原型对象"中都有一个默认的属性，叫做constructor
          constructor指向当前原型对象对应的那个"构造函数"
        3.通过构造函数创建出来的对象我们称之为"实例对象"
          每个"实例对象"中都有一个默认的属性，叫做__proto__
          __proto__指向创建它的那个构造函数的"原型对象"
     */

    // 1.创建Person构造函数
    function Person(myName, myAge){
      this.name = myName;
      this.age = myAge;
    }

    // 示例化对象
    let obj = new Person("guardwhy", 26);

    // 输出结果
    console.log(Person.prototype);
    console.log(Person.prototype.constructor);
```

```
    console.log(obj.__proto__);
  </script>
</head>
<body>

</body>
</html>
```

4、执行结果




### 6.2.3 Function函数

#### 1、基本特点

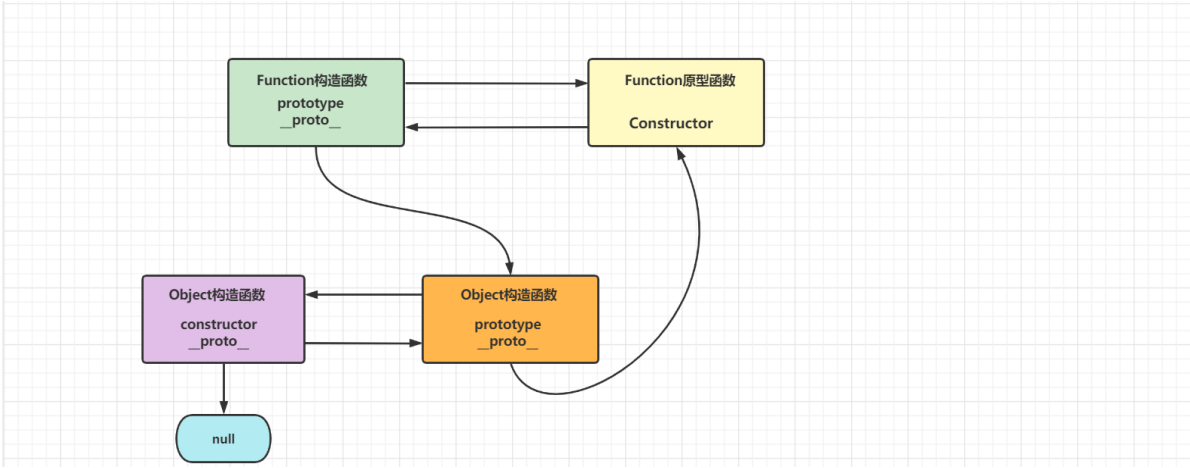
- JavaScript中函数是引用类型【对象类型】既然是对象，所以也是通过构造函数创建出来的。所有函数都是通过`Function`构造函数创建出来的对象。
- 只要是函数就有`prototype`属性，`Function`函数的`prototype`属性指向`Function`原型对象。
- `JavaScript`中只要原型对象就有`constructor`属性，`Function`原型对象的`constructor`指向它对应的构造函数。

#### 2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Function函数</title>
  <script type="text/javascript">
    /*
     Person构造函数是Function构造函数的实例对象，所以也有__proto__属性
     Person构造函数的__proto__属性指向"Function原型对象"
    */

    // 1.构造函数
    function Person(myName, myAge){
      this.name = myName;
      this.age = myAge;
    }
    // 2.实例化对象
    let obj = new Person("guardwhy", 26);
```

```
    // 3.输出结果
    console.log(Function);
    console.log(Function.prototype);
    console.log(Function.prototype.constructor);
    console.log("++++++++++++++");

    console.log(Function == Function.prototype.constructor); // true
    console.log("==============");
    console.log(Person.__proto__);
    console.log(Person.__proto__ == Function.prototype);    // true
  </script>
</head>
<body>

</body>
</html>
```

### 3、Function关系图




## 6.2.4 Object函数

1、基本概念

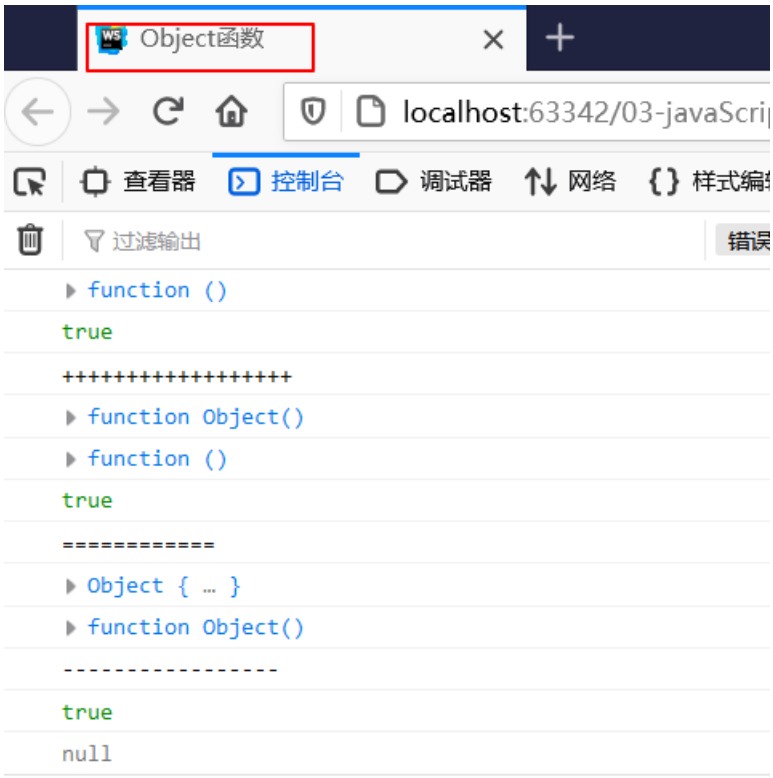
- `JavaScript`中还有一个系统提供的构造函数叫做`Object`，只要是函数都是`Function`构造函数的实例对象。
- 只要是对象就有`__proto__`属性，所以`Object`构造函数也有`__proto__`属性。`Object`构造函数的`__proto__`属性指向创建它那个构造函数的原型对象。

2、代码示例

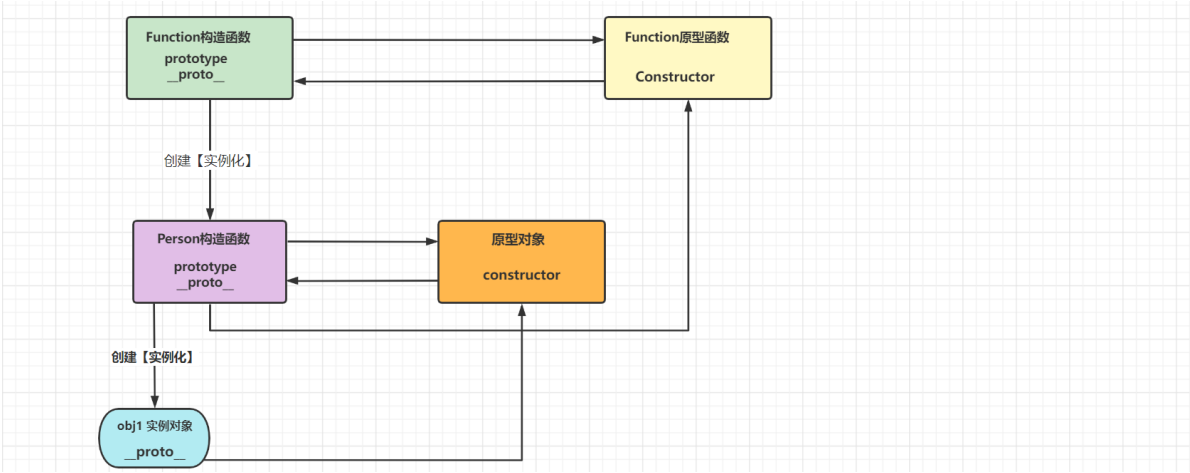
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Object函数</title>
  <script type="text/javascript">
    /*
    1.只要是构造函数都有一个默认的属性，叫做prototype，prototype属性保存着一个对象，这个对象我们称之为原型对象。
    2.只要是原型对象都有一个默认的属性，叫做constructor，constructor指向当前原型对象对应的那个构造函数。
    */

    // 1.创建Person构造函数
```

```
    function Person(myName, myAge){
      this.name = myName;
      this.age = myAge;
    }
    // 实例化对象
    let obj1 = new Person("guardwhy", 26);
    // 输出结果
    console.log(Function.__proto__);
    console.log(Function.__proto__ === Function.prototype); // true
    console.log("++++++++++++++++++");

    console.log(Object);
    console.log(Object.__proto__);
    console.log(Object.__proto__ === Function.prototype); // true
    console.log("============");

    console.log(Object.prototype);
    console.log(Object.prototype.constructor);
    console.log("-----------------");

    console.log(Object.prototype.constructor == Object);    // true
    console.log(Object.prototype.__proto__);    // null
  </script>
</head>
<body>

</body>
</html>
```

3、执行结果




**4、Object函数关系图**




## 6.3 函数对象关系

1、函数对象特点

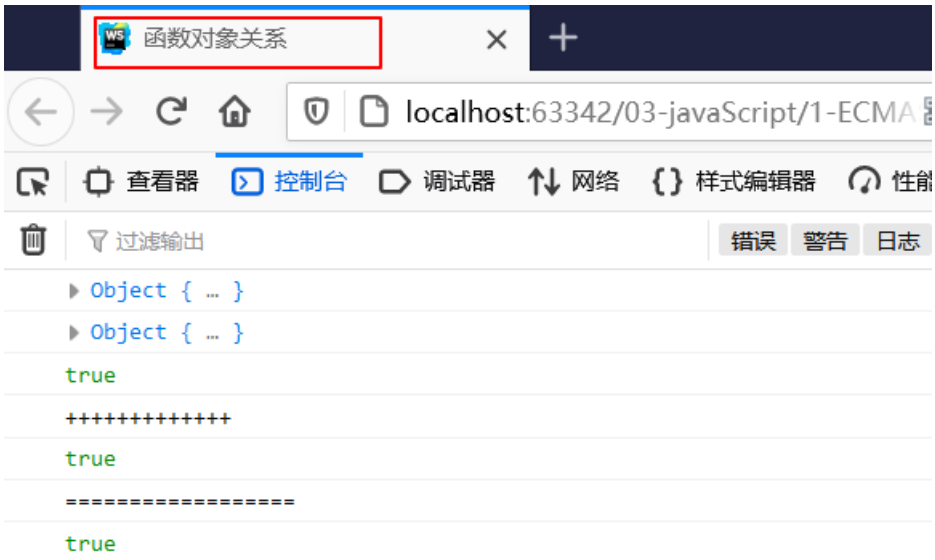
- `Function`函数是所有函数的祖先函数。所有构造函数都有一个`prototype`属性。
- 所有原型函数对象都有一个`constructor`属性，所有函数都是对象,所有对象都有一个`__proto__`属性。

2、代码示例

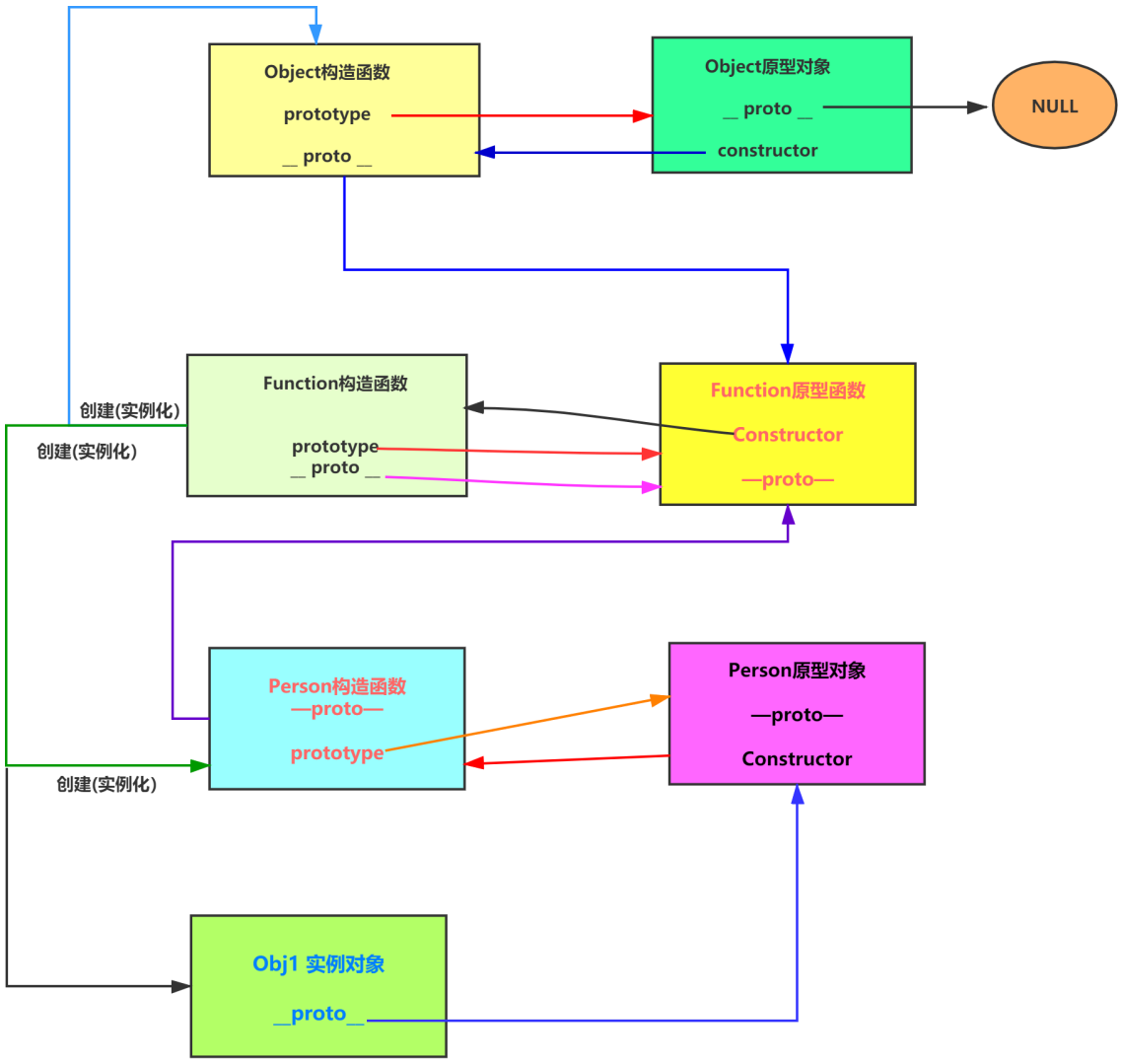
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>函数对象关系</title>
  <script type="text/javascript">
    /*
      1.所有的构造函数都有一个prototype属性，所有prototype属性都指向自己的原型对象
      2,所有的原型对象都有一个constructor属性，所有constructor属性都指向自己的构造函数
      3.所有函数都是Function构造函数的实例对象
      4.所有函数都是对象，包括Function构造函数
      5.所有对象都有__proto__属性
      6.普通对象的__proto__属性指向创建它的那个构造函数对应的"原型对象"
      7.所有对象的__proto__属性最终都会指向"Object原型对象"
      8."Object原型对象"的__proto__属性指向NULL
    */

    // 构造函数Person
    function Person(myName, myAge){
      this.name = myName;
      this.age = myAge;
    }

    // 实例化对象
    let obj1 = new Person("guardwhy", 21);

    console.log(Function.prototype.__proto__);
    console.log(Person.prototype.__proto__);
    console.log(Function.prototype.__proto__ === Person.prototype.__proto__); //
true
    console.log("++++++++++++++");
    console.log(Function.prototype.__proto__ === Object.prototype); // true
    console.log("==================");
```

```
    console.log(Person.prototype.__proto__ === Object.prototype); // true
  </script>
</head>
<body>

</body>
</html>
```

3、执行结果



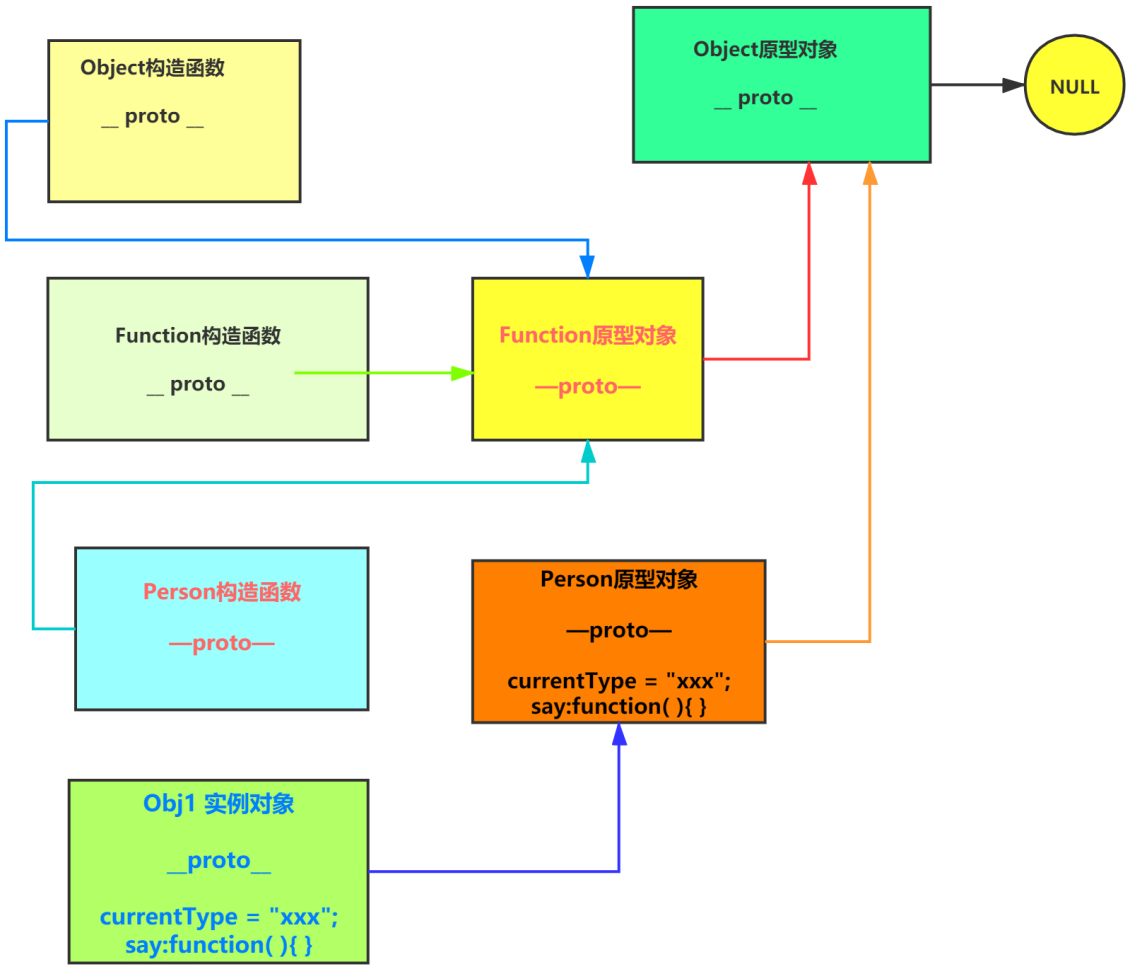

4、**函数对象图示**




## 6.4 原型链

### 6.4.1 基本概念

- 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。

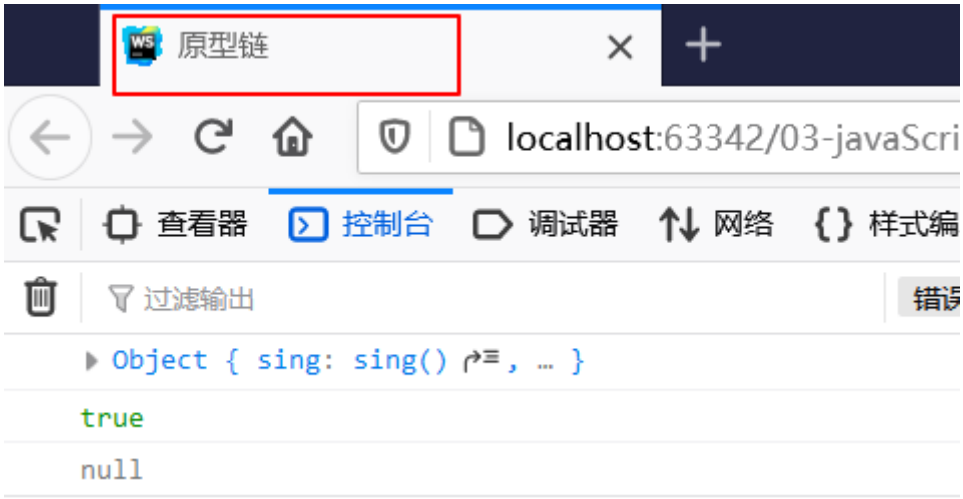



2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>原型链</title>
</head>
<body>
<script>
  function Star(uname, age) {
    this.name = uname;
    this.age = age;
  }
  Star.prototype.sing = function() {
    console.log('我会唱歌');
  };

  let obj = new Star('蔡徐坤', 18);
  // 1. 只要是对象就有__proto__ 原型, 指向原型对象
  console.log(Star.prototype);
  console.log(Star.prototype.__proto__ === Object.prototype);   // true
  // 2.我们Star原型对象里面的__proto__原型指向的是 Object.prototype
  console.log(Object.prototype.__proto__);  // null
  // 3. 我们Object.prototype原型对象里面的__proto__原型  指向为 null
</script>
</body>
```

```
</html>
```

3、执行结果




### 6.4.2 查找机制注意点

- 当访问对象的属性【包括方法】时，首先查找这个对象自身有没有该属性。如果没有就查找它的原型【也就是 `__proto__` 指向的 `prototype` 原型对象】
- 如果还没有就查找原型对象的原型【`Object` 的原型对象】。依此类推一直找到 `Object` 为止【`null`】。
- `__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。

### 6.3.3 属性注意点

- 在给一个对象不存在的属性设置值的时候, 不会去原型对象中查找。
- 如果当前对象没有就会给当前对象新增一个不存在的属性。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>属性注意点</title>
  <script type="text/javascript">
    // 1.构造函数Person
    function Person(myName, myAge){
      this.name = myName;
      this.age = myAge;
    }

    Person.prototype = {
      constructor: Person,
      currentType: "javaScript hello!!!",
      say: function (){
        console.log("属性注意点~~~");
      }
    }
    // 2.创建对象
    let obj = new Person("guardwhy", 26);
    obj.currentType = "javaScript处理跳转..";
    console.log(obj.currentType);   // javaScript处理跳转..
    console.log(obj.__proto__.currentType); // javaScript hello!!!
  </script>
```

```
</head>
<body>

</body>
</html>
```

## 6.5 函数特性

### 6.5.1 基本介绍

1、实例属性/实例方法 静态属性/静态方法

- 通过实例对象访问的属性, 称之为实例属性。通过实例对象调用的方法, 称之为实例方法。
- 通过构造函数访问的属性, 称之为静态属性。通过构造函数调用的方法, 称之为静态方法。

2、局部变量和局部函数

- 无论是`ES6`之前还是`ES6`, 只要定义一个函数就会开启一个新的作用域。
- 只要在这个新的作用域中, 通过`let/var`定义的变量就是局部变量，只要在这个新的作用域中, 定义的函数就是局部函数。

3、私有变量和函数

- 默认情况下对象中的属性和方法都是公有的, 只要拿到对象就能操作对象的属性和方法。
- 外界不能直接访问的变量和函数就是私有变量和是有函数。
- 构造函数的本质也是一个函数, 所以也会开启一个新的作用域, 所以在构造函数中定义的变量和函数就是私有和函数。

### 6.5.2 什么是封装

1、基本概念

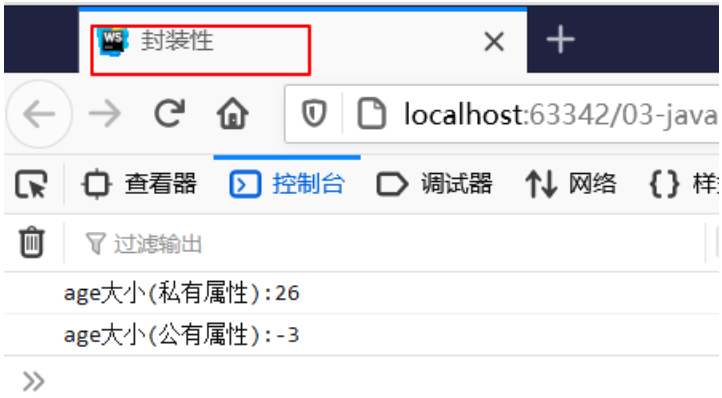
- 封装性就是隐藏实现细节，仅对外公开接口。
- 当一个类把自己的成员变量暴露给外部的时候,那么该类就失去对属性的管理权，可以任意的修改你的属性。
- 封装就是将数据隐藏起来,只能用此类的方法才可以读取或者设置数据,不可被外部任意修改。
- 封装是面向对象设计本质【将变化隔离】，这样降低了数据被误用的可能 【提高安全性和灵活性】。

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>封装性</title>
  <script type="text/javascript">
    function Person(){
      this.name = "guardwhy";
      // 定义age变量
      let age = 26;
      this.setAge = function (myAge){
        if(myAge >= 0){
          age = myAge;
        }
      }

      this.getAge = function (){
```

```
        return age;
      }
    }
    // 创建obj对象
    let obj = new Person();
    // 操作的是私有属性(局部变量)
    obj.setAge(-10);
    // 输出结果
    console.log("age大小(私有属性):" + obj.getAge());

    /*
    注意点:
    在给一个对象不存在的属性设置值的时候，不会去原型对象中查找，如果当前对象没有就会给当前对象新增一个不存在的属性。
    由于私有属性的本质就是一个局部变量，并不是真正的属性，如果通过 对象.xxx的方式是找不到私有属性的。
    */

    // 操作公有属性
    obj.age = -3;
    console.log("age大小(公有属性):" + obj.age);
  </script>
</head>
<body>

</body>
</html>
```

3、执行结果




## 6.6 this关键字

### 6.6.1 基本特点

- 当以函数的形式调用时，`this`是`window`。
- 当以方法的形式调用时，谁调用方法`this`就是谁。
- 当以构造函数的形式调用时，`this`就是新创建的那个对象。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>this关键字</title>
  <script type="text/javascript">
    // 1.创建Person构造函数
```

```
    function Person(myName, myAge) {
      this.name = myName;
      this.age = myAge;
      this.say = function () {
        // 方法中的this谁调用就是谁, 所以当前是obj1调用, 所以当前的this就是obj1
        console.log(this.name, this.age);
      }
      // return this; // 系统自动添加的
    }
    // 创建obj1对象
    let obj1 = new Person("guardwhy", 26);
    // console.log(obj1.name);
    // console.log(obj1.age);

    // 对象调用方法
    obj1.say();
  </script>
</head>
<body>

</body>
</html>
```

### 6.6.2 函数内部this指向

这些 `this` 的指向，当调用函数的时候确定的。调用方式的不同决定了 `this` 的指向不同， `this` 一般指向调用者。

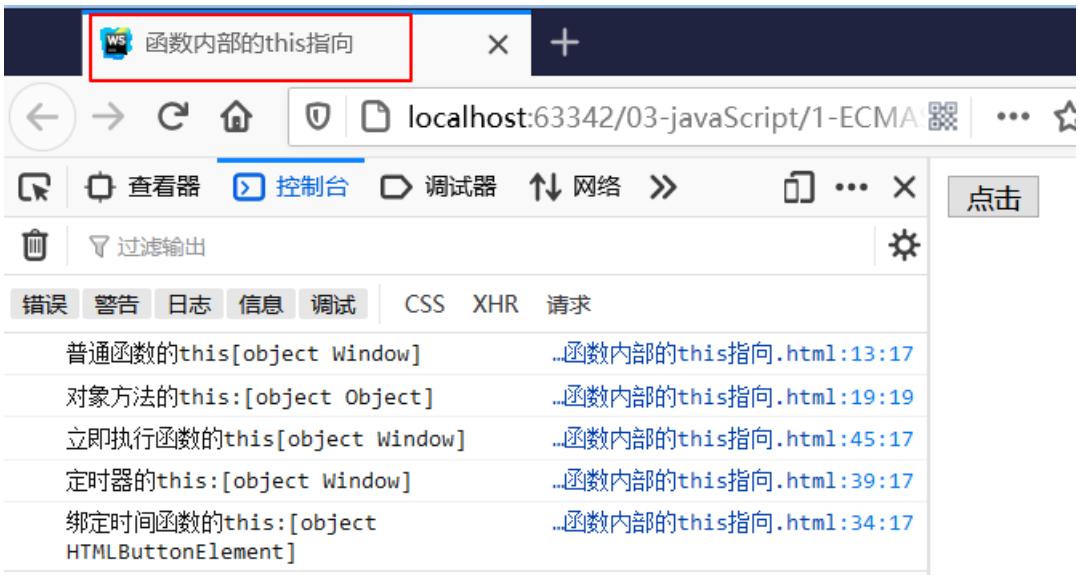
| 调用方式 | this指向 |
| --- | --- |
| 普通函数调用 | `window` |
| 构造函数调用 | 实例对象,原型对象里面的方法也指向实例对象 |
| 对象方法调用 | 该方法所属对象 |
| 事件绑定方法 | 绑定事件对象 |
| 定时器函数 | `window` |
| 立即执行函数 | `window` |

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>函数内部的this指向</title>
</head>
<body>
    <button>点击</button>
    <script type="text/javascript">
      // 函数的不同调用方式决定了this 的指向不同
      // 1. 普通函数 this 指向window
      function fn() {
        console.log('普通函数的this' + this);
```

```
      }
      window.fn();
      // 2. 对象的方法 this指向的是对象 o
      let obj1 = {
        sayHi: function() {
          console.log('对象方法的this:' + this);
        }
      }
      obj1.sayHi();

      // 3. 构造函数 this 指向 ldh 这个实例对象 原型对象里面的this 指向的也是 ldh这个实例对象
      function Person() {};
      Person.prototype.sing = function() {

      }
      let obj2 = new Person();

      // 4. 绑定事件函数 this 指向的是函数的调用者 btn这个按钮对象
      let btn = document.querySelector('button');
      btn.onclick = function() {
        console.log('绑定时间函数的this:' + this);
      };

      // 5. 定时器函数 this 指向的也是window
      window.setTimeout(function() {
        console.log('定时器的this:' + this);

      }, 1000);

      // 6. 立即执行函数 this还是指向window
      (function() {
        console.log('立即执行函数的this' + this);
      })();
    </script>
</body>
</html>
```

2、执行结果



### 6.6.3 改变函数内部this指向

> call方法

1、`call()`方法调用一个对象。简单理解为调用函数的方式，但是它可以改变函数的`this`指向。

**应用场景:** 经常做继承。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>call方法</title>
  <script type="text/javascript">
    let obj = {
      name: 'guardwhy'
    }

    function func(a, b) {
      console.log(this);
      console.log("a+b=" + a + b);
    }

    // call 第一个可以调用函数 第二个可以改变函数内的this指向
    func.call(obj, 1, 2);


    function Father(myName, myAge, mySex) {
      this.name = myName;
      this.age = myAge;
      this.sex = mySex;
    }

    function Son(myName, myAge, mySex) {
      // call 的主要作用可以实现继承
      Father.call(this, myName, myAge, mySex);
    }
    let son = new Son('小明', 18, '男');
    console.log(son);
  </script>
</head>
<body>

</body>
</html>
```

2、执行结果

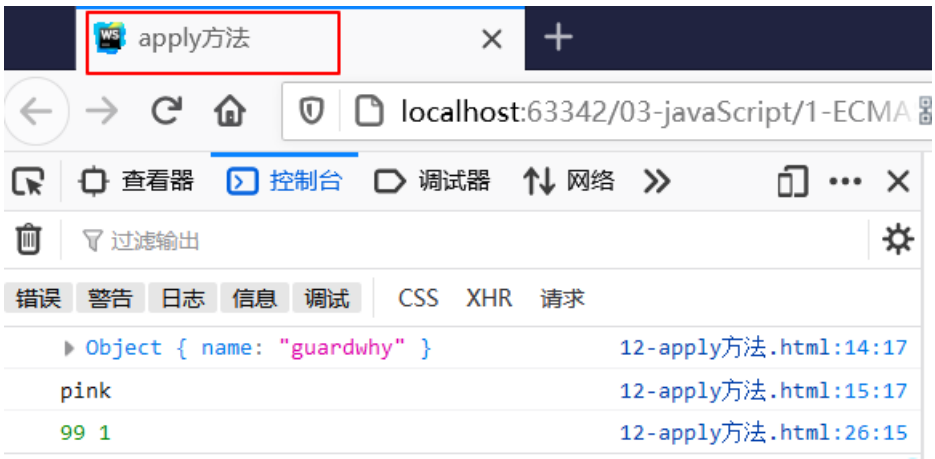

### apply方法

1、`apply()` 方法调用一个函数。简单理解为调用函数的方式，但是它可以改变函数的 `this` 指向。

**应用场景:** 经常跟数组有关系。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>apply方法</title>
</head>
<body>
    <script type="text/javascript">
      let obj = {
        name: 'guardwhy'
      };

      function fn(array1) {
        console.log(this);
        console.log(array1); // 'pink'

      }
      fn.apply(obj, ['pink']);
      // 1. 也是调用函数 第二个可以改变函数内部的this指向
      // 2. 但是他的参数必须是数组(伪数组)
      let array1 = [1, 66, 3, 99, 4];
      let array2 = ['red', 'pink'];
      // var max = Math.max.apply(null, arr);
      let max = Math.max.apply(Math, array1);
      let min = Math.min.apply(Math, array1);
      console.log(max, min);
    </script>
</body>
</html>
```

2、执行结果



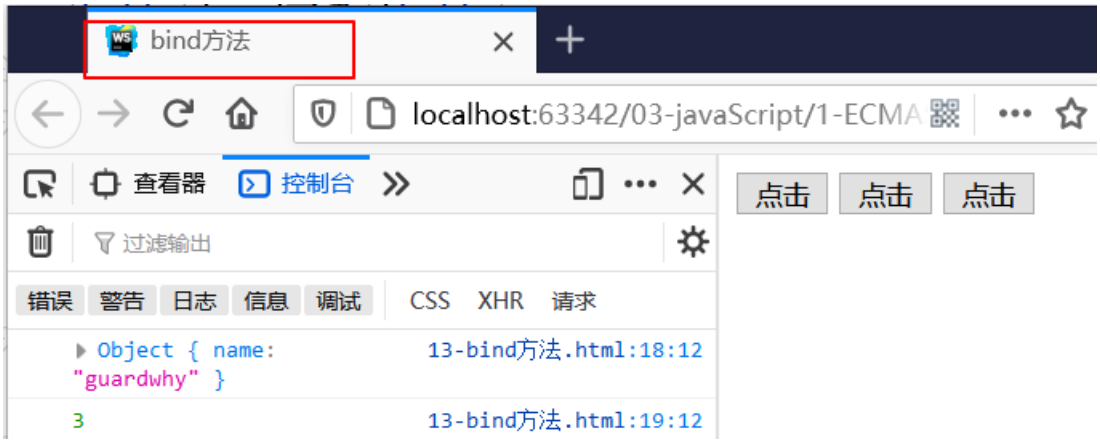

### bind方法

不会调用原来的函数，可以改变原来函数内部的this 指向，返回的是原函数改变this之后产生的新函数。

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>bind方法</title>
</head>
<body>
  <button>点击</button>
  <button>点击</button>
  <button>点击</button>
  <script>
    // bind()  绑定 捆绑的意思
    let obj = {
      name: 'guardwhy'
    };

    function func(a, b) {
      console.log(this);
      console.log(a + b);
    }
    let fs = func.bind(obj, 1, 2);
    fs();
    // 1. 如果有的函数我们不需要立即调用,但是又想改变这个函数内部的this指向此时用bind
    // 2. 我们有一个按钮,当我们点击了之后,就禁用这个按钮,2秒钟之后开启这个按钮
    let btns = document.querySelectorAll('button');
    for (let i = 0; i < btns.length; i++) {
      btns[i].onclick = function() {
        // 点击就禁用
        this.disabled = true;

        setTimeout(function() {
          this.disabled = false;
        }.bind(this), 2000);
      }
    }
  </script>
</body>
</html>
```

2、代码实现




### 6.6.4 方法区别

1、区别

共同点

- 都可以改变this指向

不同点

- `call` 和 `apply` 会调用函数, 并且改变函数内部 `this` 指向。
- `call` 和 `apply` 传递的参数不一样, `call` 传递参数使用逗号隔开, `apply` 使用数组传递。
- `bind` 不会调用函数, 可以改变函数内部 `this` 指向。

2、应用场景

- `call` 经常做继承。
- `apply` 经常跟数组有关系，比如借助于数学对象实现数组最大值最小值
- `bind` 不调用函数,但是还想改变 `this` 指向，比如改变定时器内部的 `this` 指向。

## 6.7 对象特性

### 6.7.1 JS 继承

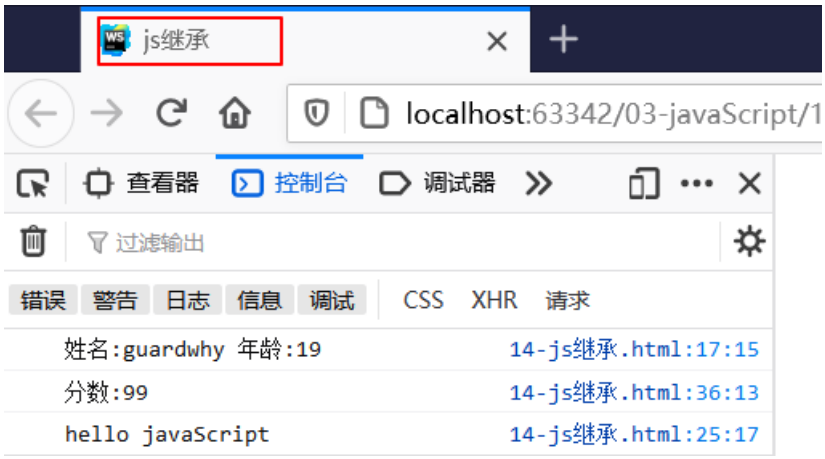
子类继承父类的属性

1、代码示例

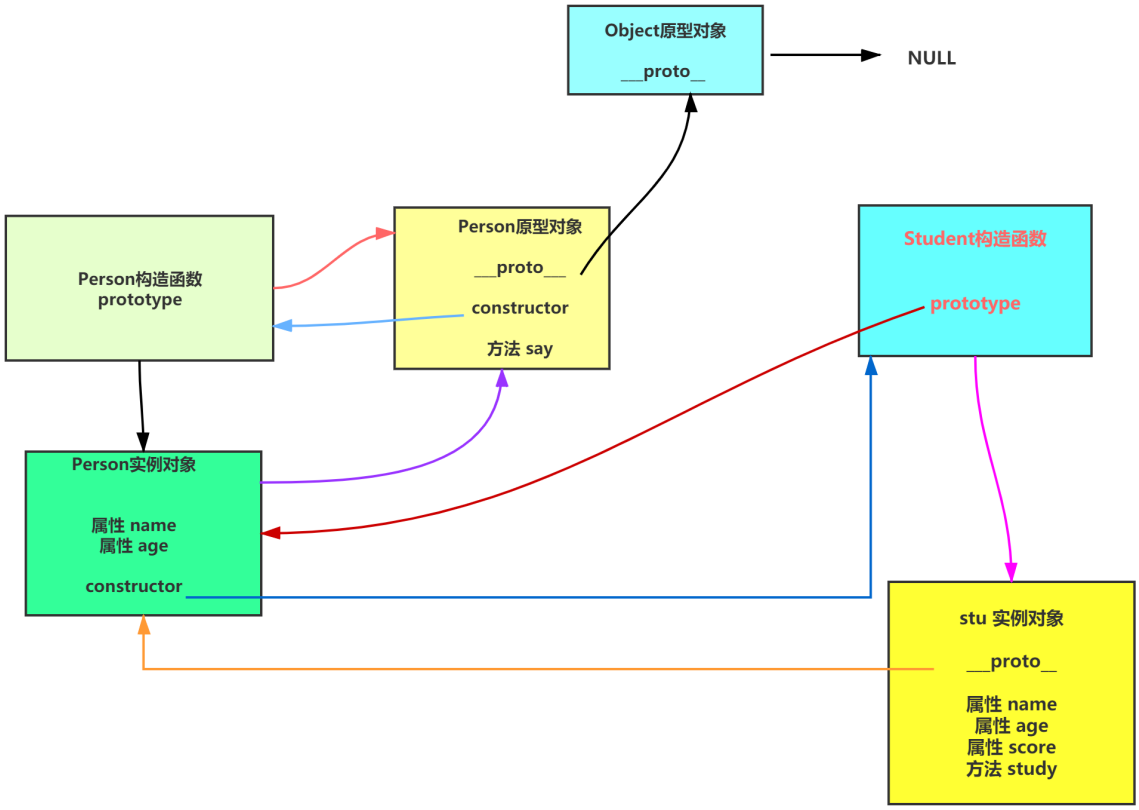
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>js继承</title>
  <script type="text/javascript">
    function Person(myName, myAge) {
      // let per = new Object();
      // let this = per;
      // this = stu;
      this.name = myName; // stu.name = myName;
      this.age = myAge; // stu.age = myAge;
      // return this;
    }
```

```
    Person.prototype.message = function () {
      console.log("姓名:" + this.name, "年龄:" + this.age);
    }

    function Student(myName, myAge, myScore) {
      // 在子类的构造函数中通过call借助父类的构造函数。
      Person.call(this, myName, myAge);
      this.score = myScore;
      this.study = function () {
        console.log("hello javaScript");
      }
    }

    Student.prototype = new Person();
    // 将子类的原型对象修改为父类的实例对象。
    Student.prototype.constructor = Student;

    // 实例化对象
    let stu = new Student("guardwhy", 19, 99);
    stu.message();
    console.log( "分数:" + stu.score);
    // 调用方法
    stu.study();
  </script>
</head>
<body>
</body>
</html>
```

2、执行结果




3、继承图示




## 6.7.2 对象属性

> 判断对象属性

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>判断对象属性</title>
</head>
<body>
    <script type="text/javascript">
      class Person{
        name = null;
        age = 0;
        height = 1.80;
      }

      // 1.创建对象obj
      let obj = new Person();
      // in的特点: 只要类中或者原型对象中有, 就会返回true
      console.log("name" in obj);   // true
      console.log("width" in obj);  // false
      console.log("height" in obj); // true

      // hasOwnProperty:判断某一个对象自身是否拥有某一个属性
      // 特点: 只会去类中查找有没有, 不会去原型对象中查找
      console.log(obj.hasOwnProperty("name")); // true
      console.log(obj.hasOwnProperty("score")); // false
    </script>
```

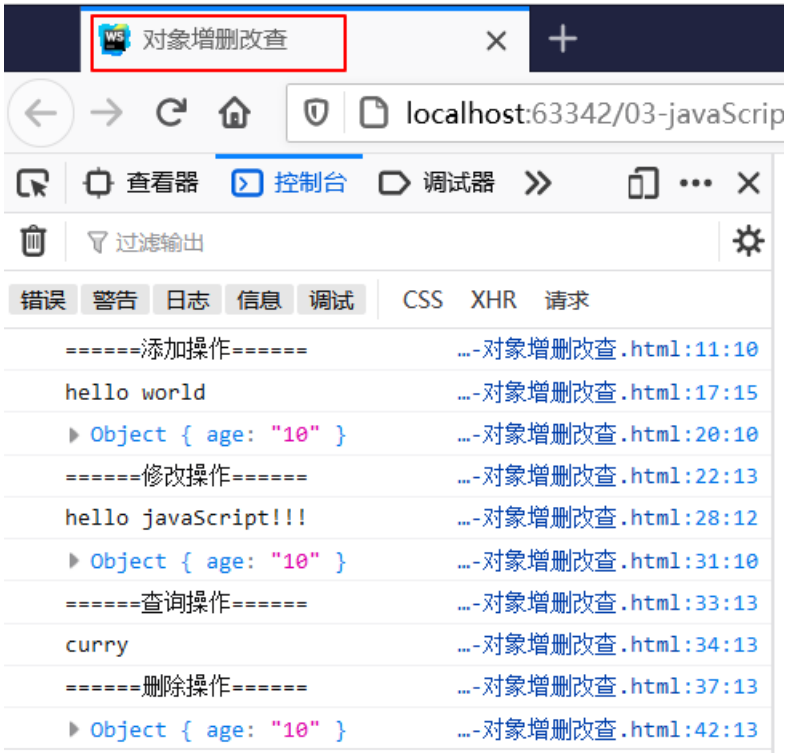
```
</body>
</html>
```

### 对象增删改查

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>对象增删改查</title>
  <script type="text/javascript">
    // 创建Person类
    class Person{}
    // 创建obj对象
    let obj = new Person();
    console.log("======添加操作======");
    // 1.添加属性
    obj["name"] = "guardwhy";
    obj["age"] = "18";
    // 2.添加方法
    obj["say"] = function (){
      console.log("hello world");
    }
    obj.say();
    console.log(obj);

    console.log("======修改操作======");
    // 3.修改属性
    obj["name"] = "curry";
    obj["age"] = "10";
    // 4.修改方法
    obj["say"] = function (){
      console.log("hello javaScript!!!");
    }
    obj.say();
    console.log(obj);

    console.log("======查询操作======");
    console.log(obj["name"]);


    console.log("======删除操作======");
    // .删除属性
    delete obj["name"];
    // 4.删除方法
    delete obj["say"];
    console.log(obj);

  </script>
</head>
<body>

</body>
</html>
```

2、执行结果




### 对象的遍历

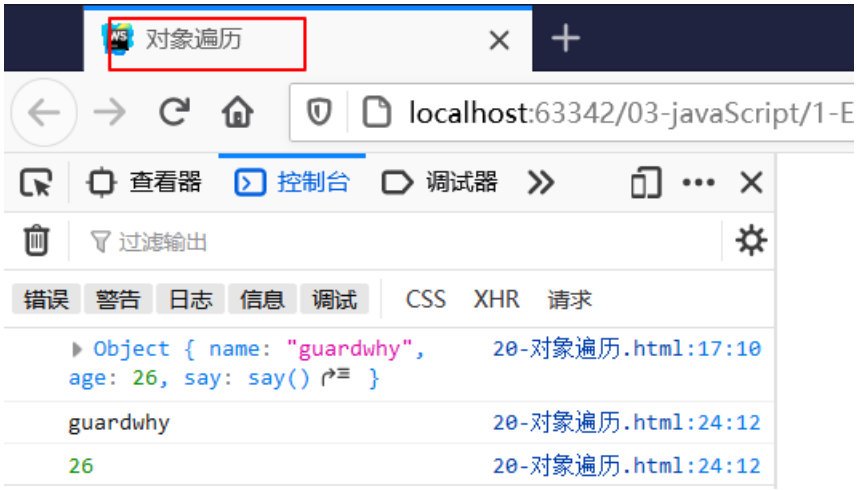
对象的遍历就是依次取出对象中所有的属性和方法。

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>对象遍历</title>
  <script type="text/javascript">
    // 1.构造函数
    function Person(myName, myAge){
      this.name = myName;
      this.age = myAge;
      this.say = function (){
        console.log(this.name, this.age);
      }
    }
    // 创建obj对象
    let obj = new Person("guardwhy", 26);
    console.log(obj);
    // 条件遍历
    for (let key in obj){
      if(obj[key] instanceof Function){
        continue;
      }
      // 注意点:取出obj对象中名称叫做当前遍历到的名称的属性或者方法的取值
      console.log(obj[key]);
    }
  </script>
</head>
<body>
```

```
</body>
</html>
```

2、执行结果



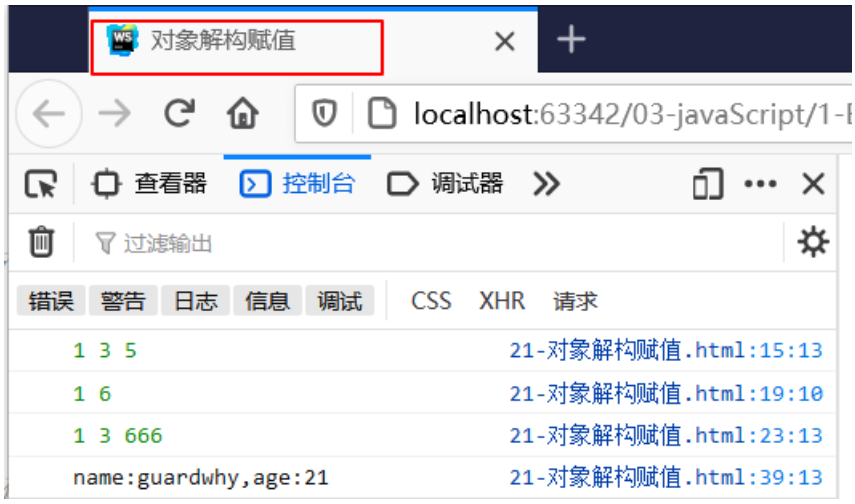

### 对象解构赋值

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>对象解构赋值</title>
  <script type="text/javascript">
    /*
      注意点:
      对象的解构赋值和数组的解构赋值 除了符号不一样, 其它的一模一样
      数组解构使用[]
      对象解构使用{}
      */
    // 1.在数组的解构赋值中, 等号左边的格式必须和等号右边的格式一模一样, 才能完全解构
    let [a1,b1,c1] = [1,3,5];
    console.log(a1, b1, c1);    // 1 3 5

    // 2.在数组的解构赋值中, 两边的个数可以不一样
    let [a2, b2] = [1,6,9];
    console.log(a2, b2);    // 1 6

    // 3.在数组的解构赋值中,如果右边少于左边, 可以左边指定默认值
    let [a3, b3, c3 = 666] = [1, 3];
    console.log(a3, b3, c3); // 1 3 666

    // 4.注意点: 在对象解构赋值中, 左边的变量名称必须和对象的属性名称一致, 才能解构出数据
    /*
      let obj = {
          name: "lnj",
          age: 34
      }
      let name = obj.name;
      let age = obj.age;

      console.log("name:" + name + ",age:" +age);
    */
```

```
    // 结构赋值
    let {name, age} = {name:"guardwhy", age:21};
    console.log("name:" + name + ",age:" +age);
  </script>
</head>
<body>

</body>
</html>
```

2、执行结果




### 6.7.3 拷贝

1、基本介绍

> **浅拷贝**

修改新变量的值会影响原有的变量的值，默认情况下引用类型都是浅拷贝。

> **深拷贝**

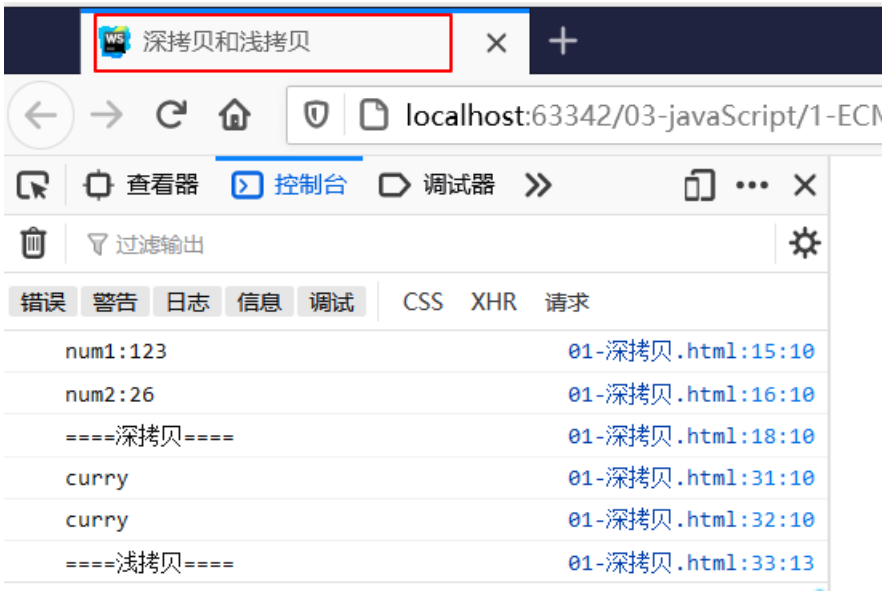
修改新变量的值不会影响原有变量的值，默认情况下基本数据类型都是深拷贝。

2、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>深拷贝和浅拷贝</title>
  <script type="text/javascript">

    // 1.深拷贝
    // 定义变量
    let num1 = 123;
    let num2 = num1;
    // 修改形变量的值
    num2 = 26;
    // 输出结果
    console.log("num1:" + num1);
    console.log("num2:" + num2);

    console.log("====深拷贝====");
```

```
    // 2.浅拷贝
    class Person{
      name="guardwhy";
      age = 26;
    }
    // 创建obj1对象
    let obj1 = new Person();
    let obj2 = obj1;
    // 修改变量的值
    obj2.name = "curry";
    // 输出结果
    console.log(obj1.name);
    console.log(obj2.name);
    console.log("====浅拷贝====");
  </script>
</head>
<body>

</body>
</html>
```

3、执行结果



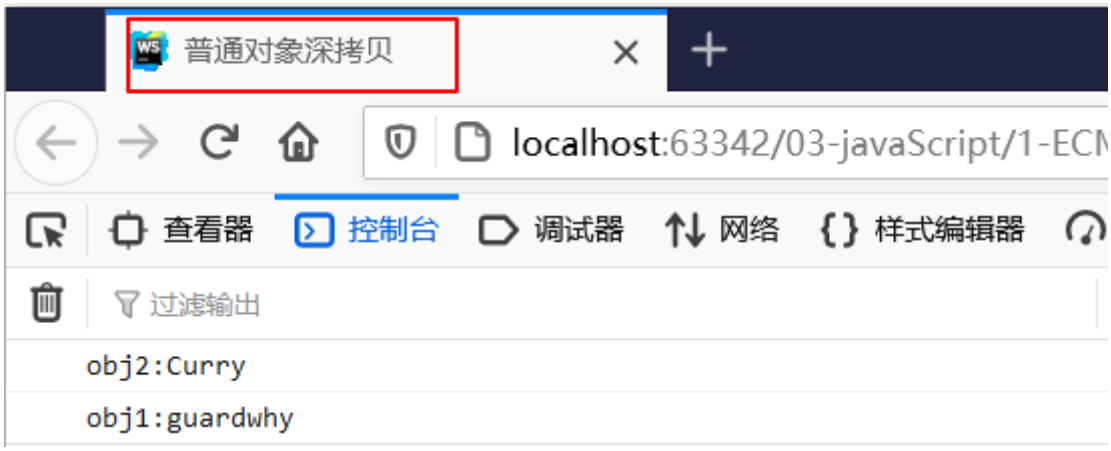

### 普通对象深拷贝

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>普通对象深拷贝</title>
  <script type="text/javascript">
    // 创建Person类
    class Person{
      name="guardwhy";
      age = 27;
    }

    // 创建obj1对象
```

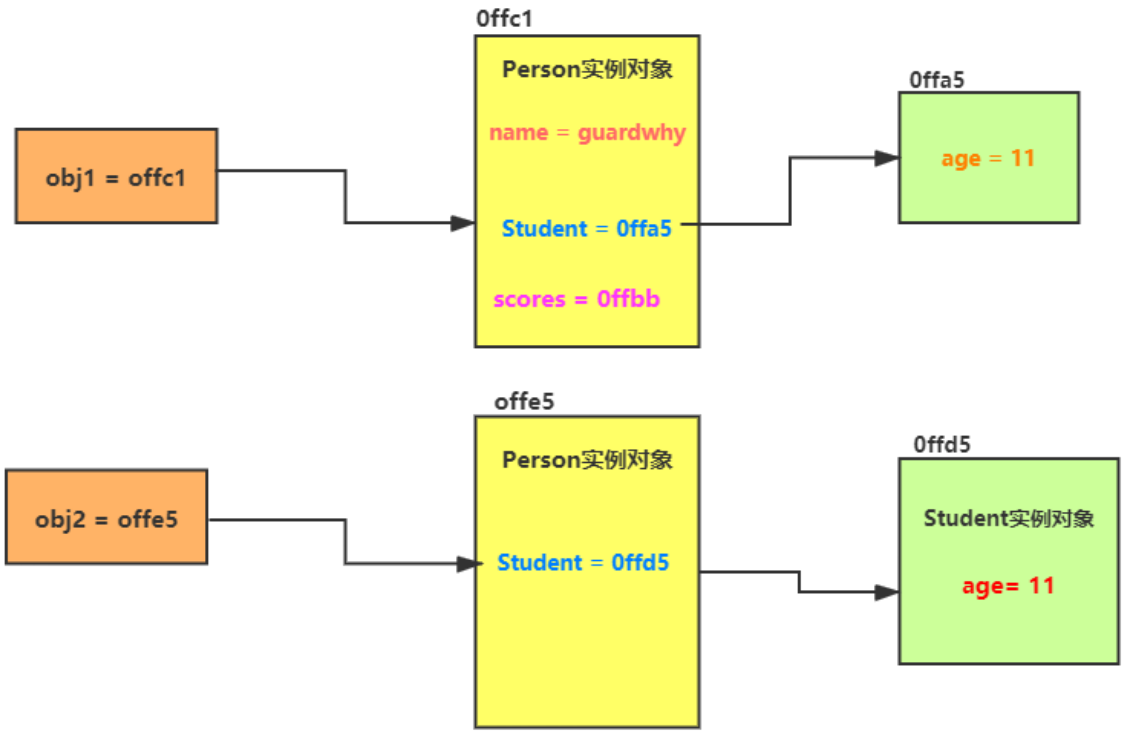
```
    let obj1 = new Person();
    // 深拷贝
    let obj2 = new Object();
    // assign方法可以将第二个参数的对象的属性和方法拷贝到第一个参数的对象中
    Object.assign(obj2, obj1);
    obj2.name = "Curry";
    console.log("obj2:"+ obj2.name);
    console.log("obj1:" + obj1.name);
  </script>
</head>
<body>

</body>
</html>
```

2、执行结果




### 对象深拷贝

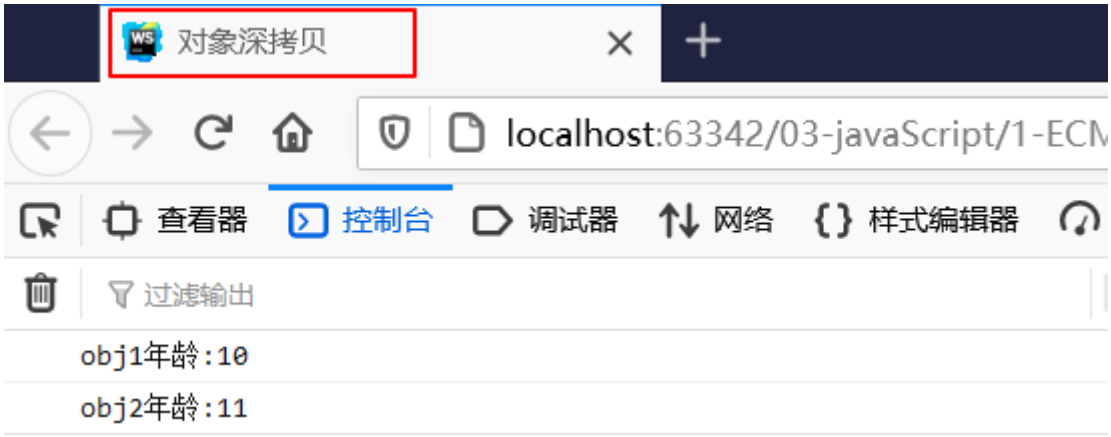
1、对象深拷贝图解



2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>对象深拷贝</title>
  <script type="text/javascript">
    // 创建Person类
    class Person{
      name="Curry";
      Student = {
        age: 10
      };
      scores = [45, 68, 89];
    }
    // 创建obj1对象
    let obj1 = new Person();
    let obj2 = new Object();

    // 调用函数
    depCopy(obj2, obj1);
    obj2.Student.age = 11;
    // 输出结果
    console.log("obj1年龄:" + obj1.Student.age);
    console.log("obj2年龄:" + obj2.Student.age);

    function depCopy(target, source){
      // 1.通过遍历拿到source中所有的属性
      for(let key in source){
        // 2.取出当前遍历到的属性对应的取值
        let sourceValue = source[key];
        // 3.判断当前的取值是否是引用数据类型
        if(sourceValue instanceof  Object){
          let subTarget = new sourceValue.constructor;
          target[key] = subTarget;
          depCopy(subTarget, sourceValue);
        }else {
          target[key] = sourceValue;
        }
      }
    }
  </script>
</head>
<body>

</body>
</html>
```

3、执行结果

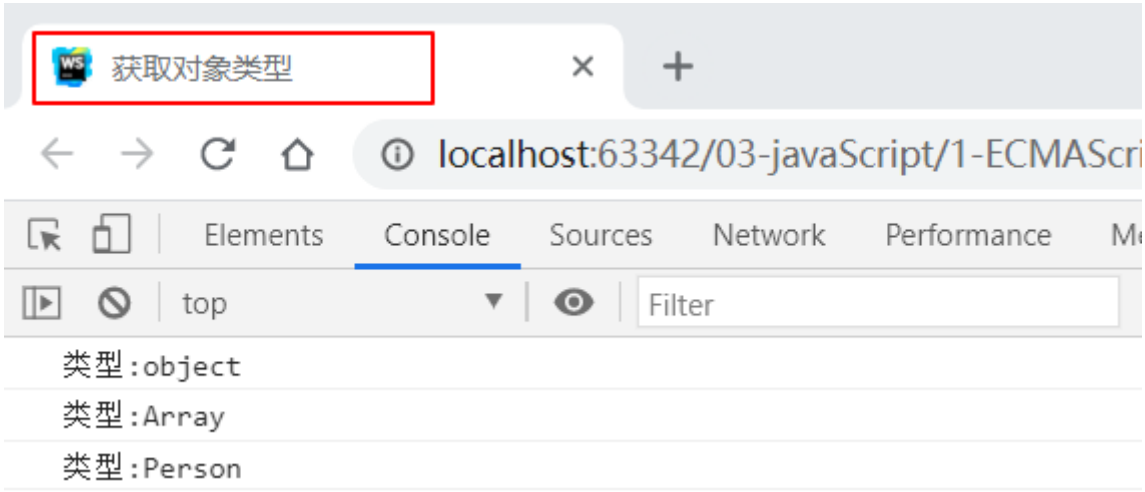



## 6.8 获取对象类型

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>获取对象类型</title>
  <script type="text/javascript">
    // 1.创建obj对象
    let obj1 = new Object();
    console.log("类型:" + typeof obj1);   // 类型:Object

    // 2.定义数组类型
    let array = new Array();
    console.log("类型:"+ array.constructor.name); // 类型:Array

    // 3.对象类型
    function Person(){
      this.name = "guardwhy";
      this.age = 21;
      this.message = function (){
        console.log("姓名:" + this.name, "年龄:" + this.age);
      }
    }

    // 实例化对象
    let obj2 = new Person();
    console.log("类型:" + obj2.constructor.name); // 类型:Person
  </script>
</head>
<body>

</body>
</html>
```

2、执行结果




## 6.9 关键字

### 6.9.1 instanceof

1、基本概念

- `instanceof` 用于判断 **"对象"** 是否是指定构造函数的 **"实例"**。
- 只要构造函数的原型对象出现在实例对象的原型链中都会返回 `true`。

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>instanceof关键字</title>
</head>
<body>
    <script type="text/javascript">
      function Person(myName){
        this.name = myName;
      }
      function Student(myName, myScore){
        Person.call(this, myName);
        this.score = myScore;
      }

      Student.prototype = new Person();
      Student.prototype.constructor = Student;

      // 实例化对象
      let obj = new Student();
      console.log(obj instanceof Person);   // true
    </script>
</body>
</html>
```

### 6.9.2 isPrototypeOf

1、基本概念

- `isPrototypeOf`用于判断 一个对象是否是另一个对象的原型。
- 只要调用者在传入对象的原型链上都会返回`true`。

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>isPrototypeOf属性</title>
</head>
<body>
    <script type="text/javascript">
      function Person(myName){
        this.name = myName;
      }
      function Student(myName, myScore){
        Person.call(this, myName);
        this.score = myScore;
      }

      Student.prototype = new Person();
      Student.prototype.constructor = Student;

      // 实例化对象
      let obj = new Student();
      console.log(Person.prototype.isPrototypeOf(obj)); // true
    </script>
</body>
</html>
```

# 7 - JS DOM

## 7.1 DOM介绍

### 7.1.1 window概念

`window`：是一个全局对象, 代表浏览器中一个打开的窗口, 每个窗口都是一个`window`对象。

### 7.1.2 document概念

`document`是`window`的一个属性, 这个属性是一个对象。

`document`代表当前窗口中的整个网页, `document`对象保存了网页上所有的内容, 通过`document`对象就可以操作网页上的内容。

### 7.1.3 DOM介绍

1、基本介绍

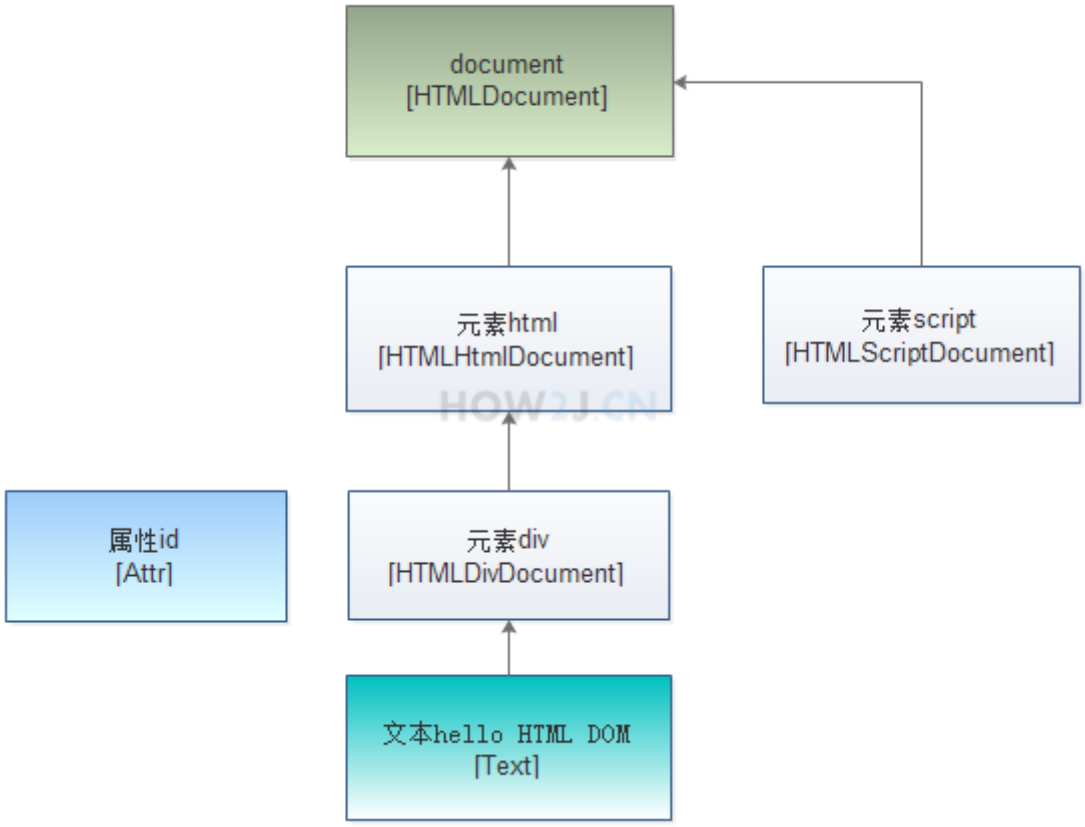
文档对象模型【`DOM, Document Object Model`】主要用于对 `HTML` 文档的内容进行操作。

`DOM` 把 `HTML` 文档表达成一个节点树，通过对节点进行操作，实现对文档内容的添加、删除、修改、查找等功能。

`DOM` 节点就是 `HTML` 上所有的内容 包括 :

- 文档节点
- 元素节点(标签)
- 元素属性节点
- 文本节点
- 注释节点

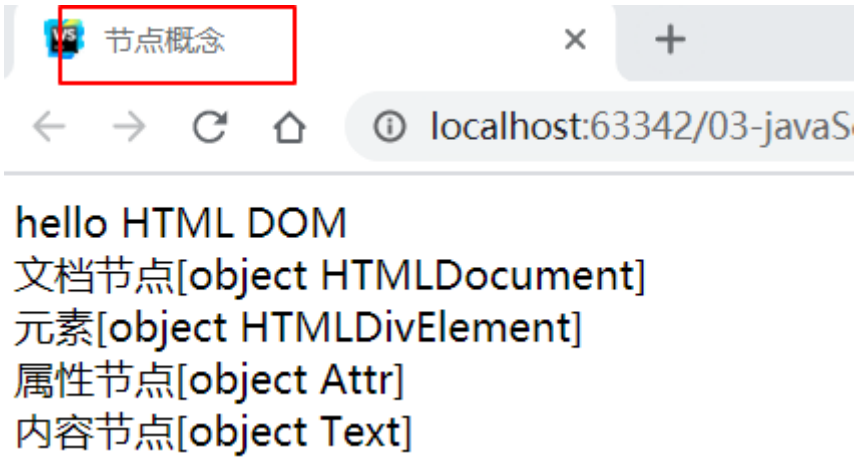
2、节点图示




3、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>节点概念</title>
</head>
<body>
<div id="d1">hello HTML DOM</div>
</body>
<script>
  function p(s){
    document.write(s);
    document.write("<br>");
  }
  let div1 = document.getElementById("d1");
  p("文档节点"+document);
  p("元素"+div1);
  p("属性节点"+div1.attributes[0]);
  p("内容节点"+div1.childNodes[0]);
</script>
</html>
```

4、执行结果




## 7.2 DOM相关操作

### 7.2.1 获取DOM元素

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>获取DOM元素</title>
</head>
<body>
<!--
    1.在JavaScript中HTML标签也称之为DOM元素。
    2.使用document的时候前面不用加window。
-->
<div class="father">
  <form>
    <input type="text" name="test">
    <input type="password" name="test">
  </form>

  <div class="father" id="box">这是一个div</div>

  <script type="text/javascript">
    /*
    1.通过id获取指定元素
    由于id不可以重复，所以找到了就会将找到的标签包装成一个对象返回给我们，找不到就返回Null
    注意点：DOM操作返回的是一个对象，这个对象是宿主类型对象(浏览器提供的对象)
    */
    console.log("====通过id获取指定元素====");
    let div1 = document.getElementById("box");
    // 输出结果
    console.log(div1);
    console.log(typeof div1);   // obj类型

    /*
    2.通过class名称获取
    由于class可以重复，所以找到了就返回一个存储了标签对象的数组，找不到就返回一个空数组
    */
    console.log("====通过class名称获取====");
```

```
    let div2 = document.getElementsByClassName("father");
    console.log(div2);

    /*
     3.通过name名称获取
     由于name可以重复，所以找到了就返回一个存储了标签对象的数组，找不到就返回一个空数组。
   */
    console.log("++++通过name名称获取++++");
    let div3 = document.getElementsByName("test");
    console.log(div3);

    /*
   4.通过标签名称获取
   由于标签名称可以重复，所以找到了就返回一个存储了标签对象的数组，找不到就返回一个空数组
   */
    console.log("++++通过标签名称获取++++");
    let div4 = document.getElementsByTagName("div");
    console.log(div4);

    /*
    5.通过选择器获取
    querySelector只会返回根据指定选择器找到的第一个元素
    */
    console.log("====通过选择器获取====");
    let div5 = document.querySelector("#box");
    console.log(div5);

    let div6 = document.querySelector(".father");
    console.log(div6);

    let div7 = document.querySelector("div>form");
    console.log(div7);

    /*
    6.通过选择器获取
    querySelectorAll会返回指定选择器找到的所有元素
    */
    console.log("====返回指定选择器找到的所有元素====");
    let elements = document.querySelectorAll(".father");
    console.log(elements);
  </script>
</div>
</body>
</html>
```

2、执行结果


### 7.2.2 获取节点

DOM 对象 (document) 这个对象以树的形式保存了界面上所有的内容， HTML 页面每一部分都是节点【标签(元素),文本，属性】。

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>获取DOM元素</title>
</head>
<body>
    <div>
      <h3>hello javaScript! ! ! </h3>
      <h3>hello Vue.js!!!</h3>
      <p class="special">Spring Boot! ! ! </p>
      <p>html</p>
      <span>CSS改变文本颜色.</span>
    </div>
    <script type="text/javascript">
      // 1.获取指定元素所有的子元素
      console.log("====获取指定元素所有的子元素====");
      let elements = document.querySelector("div");
      // 1.1 children属性获取到的是指定元素中所有的子元素
      console.log(elements.children);
      // 1.2 childNodes属性获取到的是指定元素中所有的节点
```

```
        console.log("====获取到的是指定元素中所有的节点===")
        console.log(elements.childNodes);
        // 1.3 遍历节点
        console.log("===遍历节点====")
        for(let node of elements.childNodes){
          // 条件判断
          if(node.nodeType === Node.ELEMENT_NODE){
            console.log(node);
          }
        }

        // 2.获取指定节点中的第一个子节点
        console.log("===获取指定节点第一个子节点==");
        let div2 = document.querySelector("div");
        console.log(div2.firstChild);

        //3.获取指定元素中的第一个子元素
        console.log("===获取指定元素的第一个子元素===");
        console.log(div2.firstElementChild);

        // 4.获取指定节点中最后一个子节点
        console.log("===获取指定节点最后一个子节点===");
        console.log(div2.lastChild);

        // 5.获取指定元素中最后一个子元素
        console.log("===获取指定元素中最后一个子元素===");
        console.log(div2.lastElementChild);

        // 6.通过子元素获取父元素/父节点
        console.log("===通过子元素获取父元素/父节点===")
        let div3 = document.querySelector(".special");
        let parentEle = div3.parentElement || div3.parentNode;
        // 输出结果
        console.log(parentEle);

        // 7.获取相邻上一个节点
        console.log("====获取相邻上一个节点===");
        console.log(div3.previousSibling);

        // 8.获取相邻的上一个元素
        console.log("====获取相邻的上一个元素===");
        console.log(div3.previousElementSibling);

        // 9.获取相邻的下一个节点
        console.log("====获取相邻的下一个节点===");
        console.log(div3.nextSibling);
        console.log("===获取相邻下一个元素===");
        console.log(div3.nextElementSibling);
    </script>
</body>
</html>
```

2、执行结果


### 7.2.3 节点的增删改查

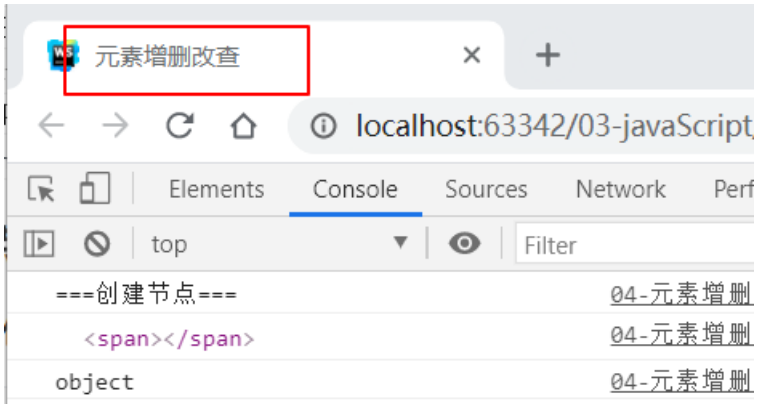
> **创建节点**

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>元素增删改查</title>
</head>
<body>
<div>
  <h3>javaScript DOM</h3>
  <p>DOM元素增删改查</p>
```

```
</div>
<script type="text/javascript">
  // 1.创建节点
  console.log("===创建节点===");
  let mySpan = document.createElement("span");
  console.log(mySpan);
  console.log(typeof mySpan);
</script>
</body>
</html>
```

2、执行结果

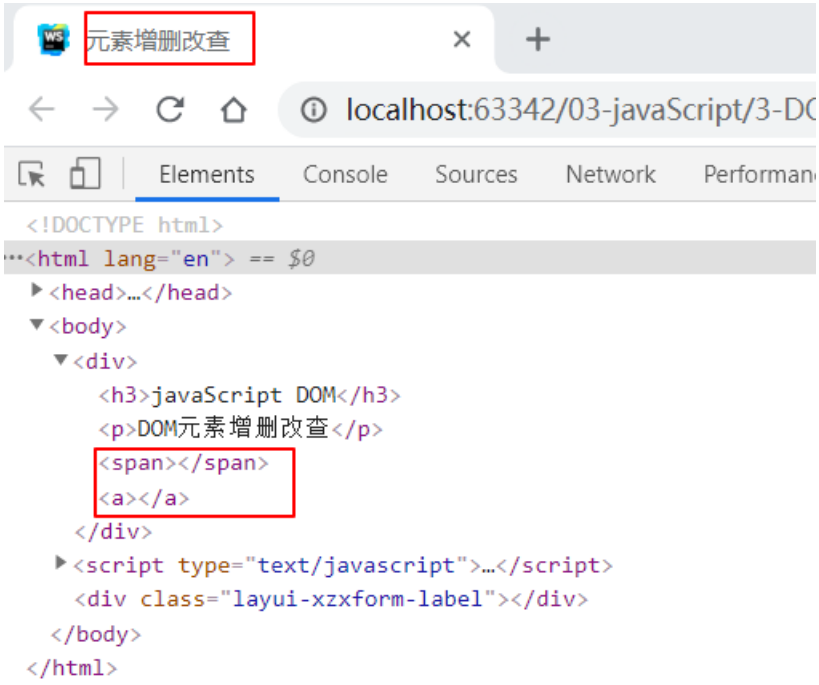



### 添加节点

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>元素增删改查</title>
</head>
<body>
<div>
  <h3>javaScript DOM</h3>
  <p>DOM元素增删改查</p>
</div>
<script type="text/javascript">

  // 2.添加节点
  // 注意点: appendChild方法会将指定的元素添加到最后
  let mySpan = document.createElement("span");
  let myDiv = document.querySelector("div");
  myDiv.appendChild(mySpan);
  let myA = document.createElement("a");
  myDiv.appendChild(myA);
</script>
</body>
</html>
```

2、执行结果

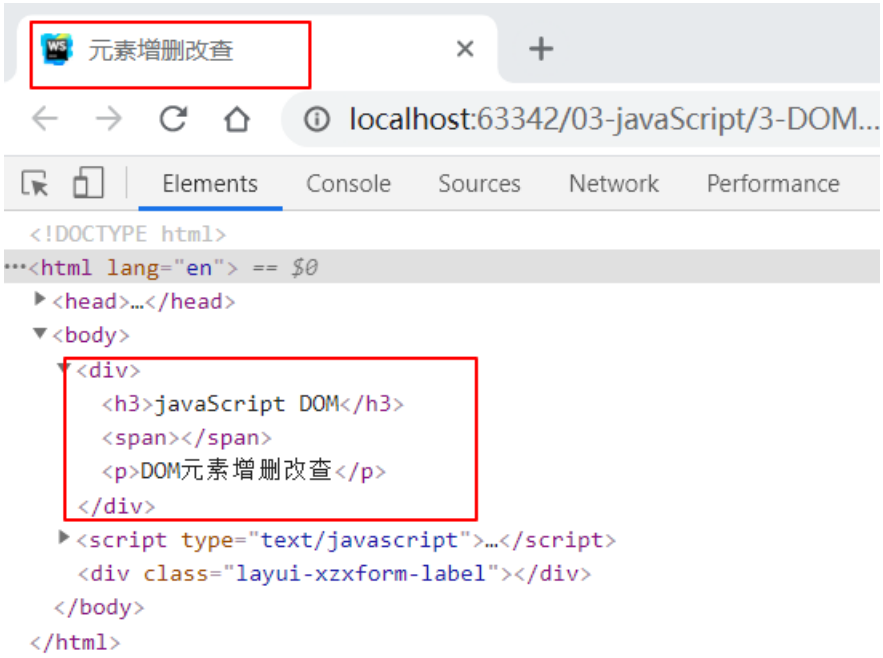



### 插入节点

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>元素增删改查</title>
</head>
<body>
<div>
  <h3>javaScript DOM</h3>
  <p>DOM元素增删改查</p>
</div>
<script type="text/javascript">
  // 3.插入节点
  let mySpan = document.createElement("span");
  let myDiv = document.querySelector("div");
  let myH3 = document.querySelector("h3");
  myDiv.insertBefore(mySpan, myH3);
  let myP = document.querySelector("p");
  myDiv.insertBefore(mySpan, myP);
</script>
</body>
</html>
```

2、执行结果

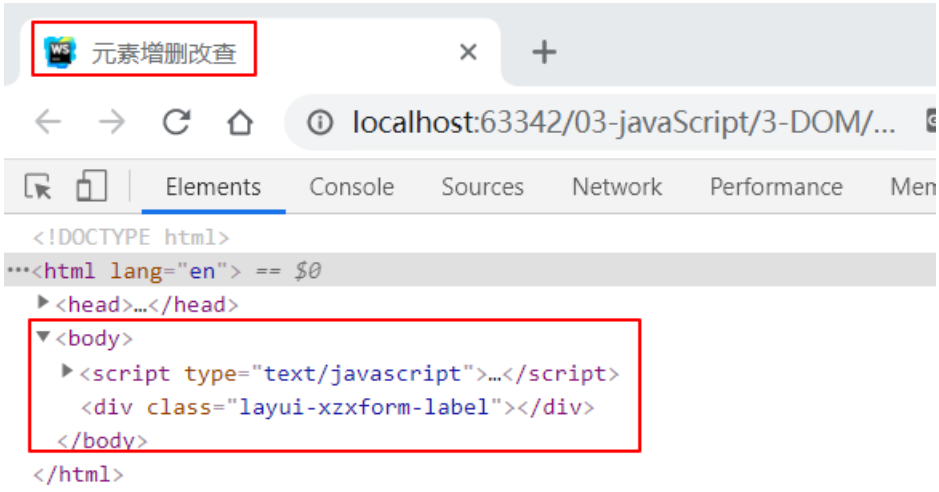



### 删除节点

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>元素增删改查</title>
</head>
<body>
<div>
  <h3>javaScript DOM</h3>
  <p>DOM元素增删改查</p>
</div>
<script type="text/javascript">

  // 3.插入节点
  let mySpan = document.createElement("span");
  let myDiv = document.querySelector("div");
  let myH3 = document.querySelector("h3");
  myDiv.insertBefore(mySpan, myH3);
  let myP = document.querySelector("p");
  myDiv.insertBefore(mySpan, myP);

  // 4.删除节点
  // 注意:在js中如果想要删除某一个元素, 只能通过对应的父元素来删除.
  mySpan.parentNode.removeChild(mySpan);
  myDiv.parentNode.removeChild(myDiv);
</script>
</body>
</html>
```

2、执行结果

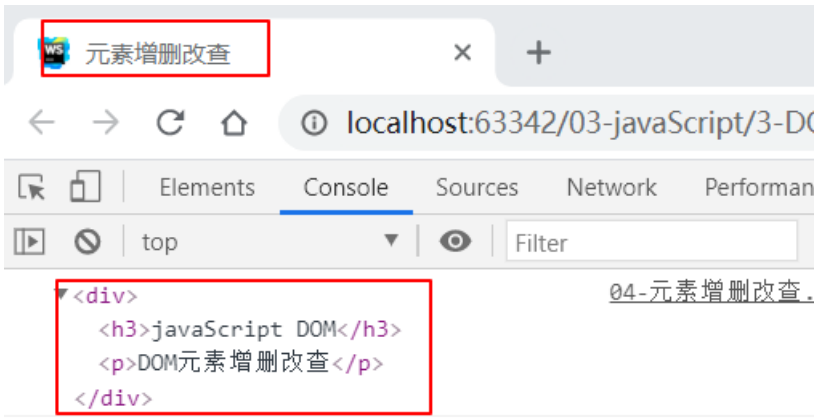



### 克隆节点

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>元素增删改查</title>
</head>
<body>
<div>
  <h3>javaScript DOM</h3>
  <p>DOM元素增删改查</p>
</div>
<script type="text/javascript">
  // 5.克隆节点
  let myDiv = document.querySelector("div");
  let newDiv = myDiv.cloneNode(true);
  console.log(newDiv);
</script>
</body>
</html>
```

2、执行结果



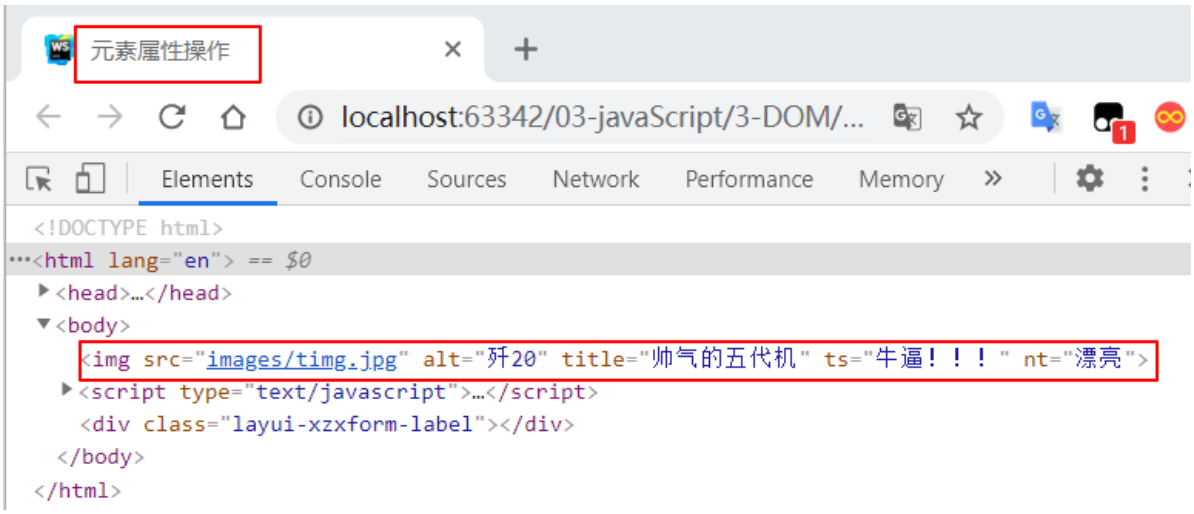
### 7.2.4 元素属性操作

通过对象`.`属性名称的方式无法获取到自定义属性的取值，通过`getAttribute`方法可以获取到自定义属性的取值。

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>元素属性操作</title>
</head>
<body>
<img src="images/timg.jpg" alt="歼20" title="第五代隐形战斗机" ts="123">
<script type="text/javascript">
  // 1.获取元素属性
  let myImg1 = document.querySelector("img");
  // 输出结果
  console.log("alt属性:" + myImg1.getAttribute("alt"));
  // 自定义属性
  console.log("TS属性:" + myImg1.getAttribute("TS"));

  // 2.修改元素属性
  let myImg2 = document.querySelector("img");
  // 设置属性
  myImg2.setAttribute("title", "帅气的五代机");
  myImg2.setAttribute("ts", "牛逼！！！");

  // 3.新增元素属性
  let myImg3 = document.querySelector("img");
  myImg3.setAttribute("nb", "隐形机");
  myImg3.setAttribute("nt", "漂亮");

  // 4.删除元素属性
  let myImg4 = document.querySelector("img");
  myImg4.removeAttribute("nb");
</script>
</body>
</html>
```

2、执行结果



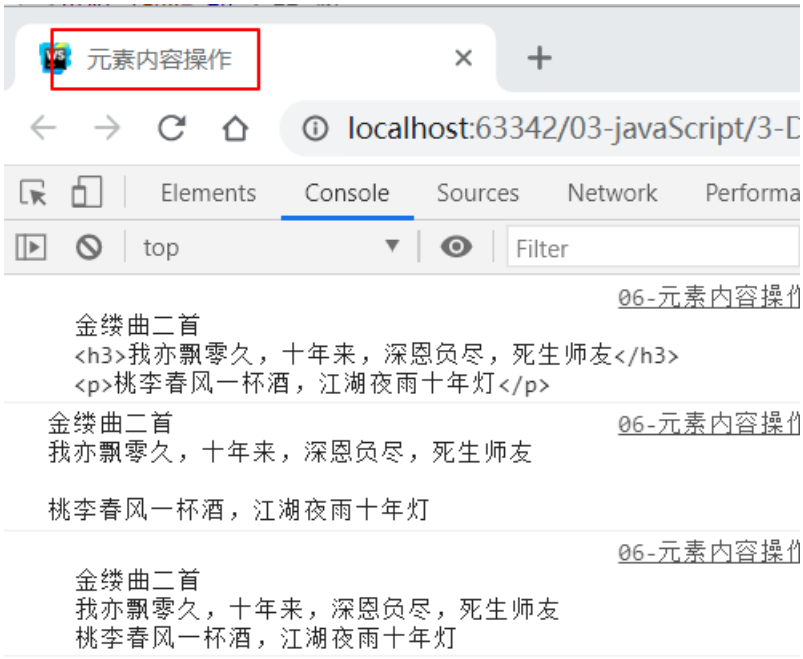

## 7.2.5 元素内容操作

> **获取元素内容**

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>元素内容操作</title>
</head>
<body>
<div>
  金缕曲二首
  <h3>我亦飘零久，十年来，深恩负尽，死生师友</h3>
  <p>桃李春风一杯酒，江湖夜雨十年灯</p>
</div>
<script type="text/javascript">
  // 1.获取元素内容
  /*
   1.innerHTML获取的内容包含标签，innerText/textContent获取的内容不包含标签
   2.innerHTML/textContent获取的内容不会去除两端的空格，innerText获取的内容会去除两端的空格
   */
  let myDiv1 = document.querySelector("div");
  console.log(myDiv1.innerHTML);
  console.log(myDiv1.innerText);
  console.log(myDiv1.textContent);
</script>
</body>
</html>
```

2、执行结果



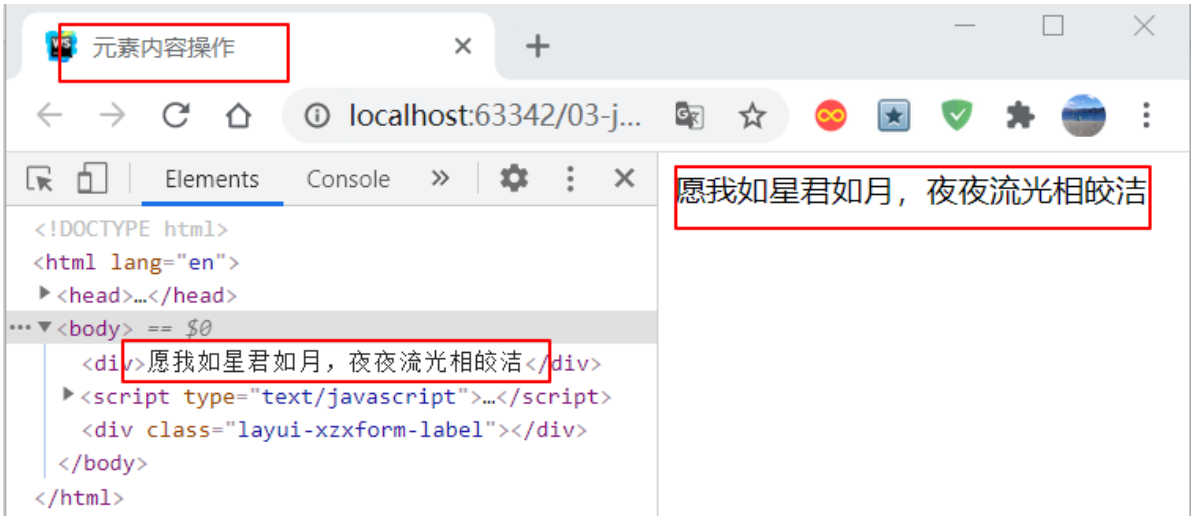

> 设置元素内容

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>元素内容操作</title>
</head>
<body>
<div>
  金缕曲二首
  <h3>我亦飘零久，十年来，深恩负尽，死生师友</h3>
  <p>桃李春风一杯酒，江湖夜雨十年灯</p>
</div>
<script type="text/javascript">

  // 2.设置元素内容
  /*
  特点:
  无论通过innerHTML/innerText/textContent设置内容，新的内容都会覆盖原有的内容。
  区别:
  如果通过innerHTML设置数据，数据中包含标签，会转换成标签之后再添加。
  如果通过innerText/textContent设置数据，数据中包含标签，不会转换成标签，会当做一个字符串直接设置。
  */
  let myDiv2 = document.querySelector("div");
  // myDiv2.innerHTML = "123";
  // myDiv2.innerText = "456";
  // myDiv2.textContent = "789";
  // myDiv2.innerHTML = "<span>愿我如星君如月，夜夜流光相皎洁</span>";
  // myDiv2.innerText = "<span>愿我如星君如月，夜夜流光相皎洁</span>";
  // myDiv2.textContent = "<span>愿我如星君如月，夜夜流光相皎洁</span>";

  setText(myDiv2, "愿我如星君如月，夜夜流光相皎洁");
  function setText(obj, text) {
    if("textContent" in obj){
      obj.textContent = text;
```

```
    }else{
      obj.innerText = text;
    }
  }
</script>
</body>
</html>
```

2、执行结果




## 7.2.6 操作元素样式

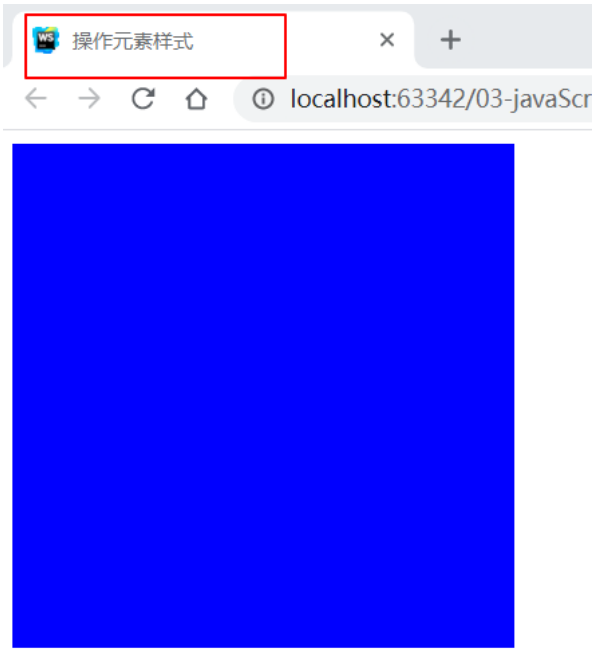
> **设置元素样式**

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>操作元素样式</title>
  <style>
    .box{
        width: 200px;
        height: 200px;
        background-color: blueviolet;
    }
  </style>
</head>
<body>
<div class="box"></div>
<script type="text/javascript">
  // 1.设置元素样式
  let myDiv1 = document.querySelector("div");
  // 方式一:由于class在JS中是一个关键字, 所以叫做className.
  myDiv1.className = "box";

  // 第二种方式
  // 注意点: 通过JS添加的样式都是行内样式, 会覆盖掉同名的CSS样式.
  myDiv1.style.width = "300px";
  myDiv1.style.height = "300px";
  myDiv1.style.backgroundColor = "blue";
```

```
</script>
</body>
</html>
```

2、执行结果

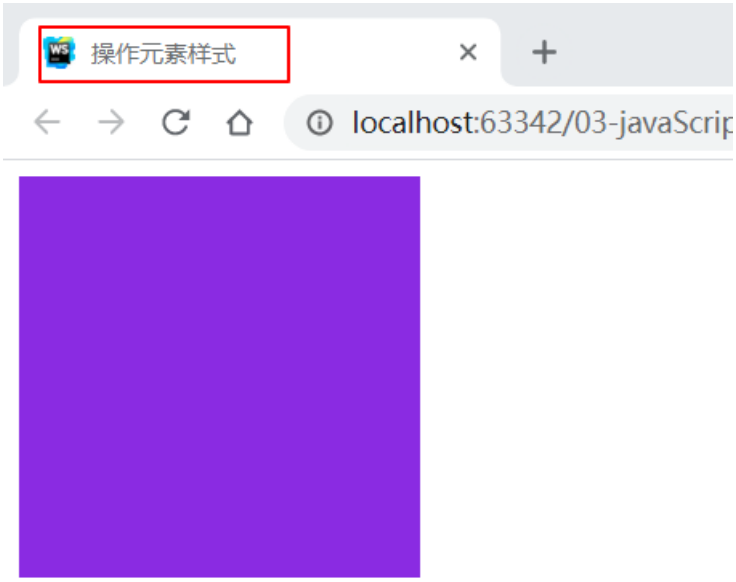



### 获取元素样式

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>操作元素样式</title>
  <style>
    .box{
        width: 200px;
        height: 200px;
        background-color: blueviolet;
    }
  </style>
</head>
<body>
<div class="box"></div>
<script type="text/javascript">
  // 2.获取元素样式
  let myDiv2 = document.querySelector("div");
  // 取到CSS设置的属性值,必须通过getComputedStyle方法来获取
  let style = window.getComputedStyle(myDiv2);
  console.log(style.width);
</script>
</body>
</html>
```

2、执行结果




## 7.3 DOM事件

### 7.3.1 事件基本定义

- 在网页中操作的时候，会激活各种事件，可以通过`JS`代码来对这些事件编程，当激活这些事件的时候，实现相应的功能。
- 可以让网页"活"起来，与用户有交互的功能。

### 7.3.2 设置事件方式

> 命名函数

```
<input type="button" onclick="clickMe()" id="btn">

function clickMe() {
   //事件处理函数
}
```

> 匿名函数

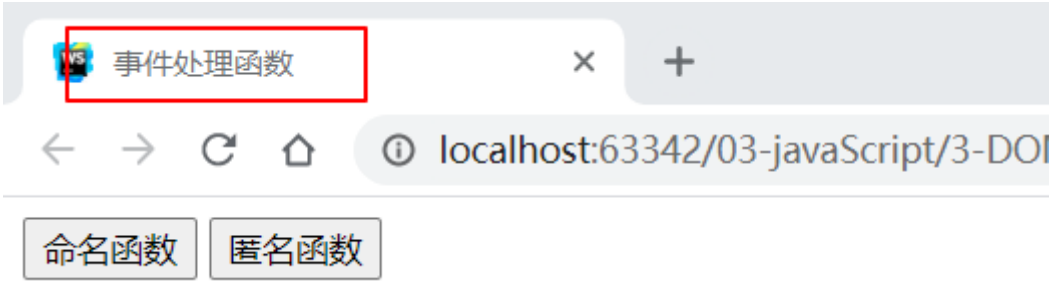
```
document.querySelector("btn").onclick = function() {
  //事件处理函数
}
```

**代码示例**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>事件处理函数</title>
</head>
<body>
<input type="button" value="命名函数" id="b1" onclick="clickMe()">
<input type="button" value="匿名函数" id="b2">

<script type="text/javascript">
  //命名函数
  function clickMe() {
```

```
    alert("命名函数");
  }

  //匿名函数要写在按钮元素的后面
  document.getElementById("b2").onclick = function () {
    alert("匿名函数");
  }
</script>
</body>
</html>
```

**执行结果**




### 7.3.4 Event(事件)

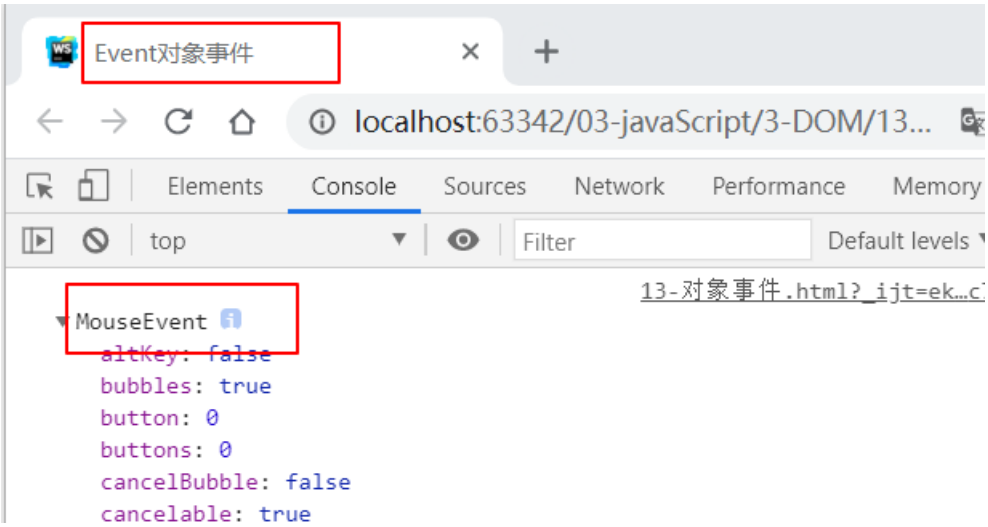
1、基本介绍

- `Event`对象代表事件的状态，比如事件在其中发生的元素、键盘按键的状态、鼠标的位置、鼠标按钮的状态。
- 事件通常与函数结合使用，函数不会在事件发生前被执行!
- 事件对象是和当前事件有关系。

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Event对象事件</title>
  <style type="text/css">
      body,html{height: 100%;}
  </style>
</head>
<body>
<span id='span'></span>
<script>
  onclick = function(event){
    console.log(event)
  };
</script>
</body>
</html>
```

3、执行结果




### 7.3.5 事件流

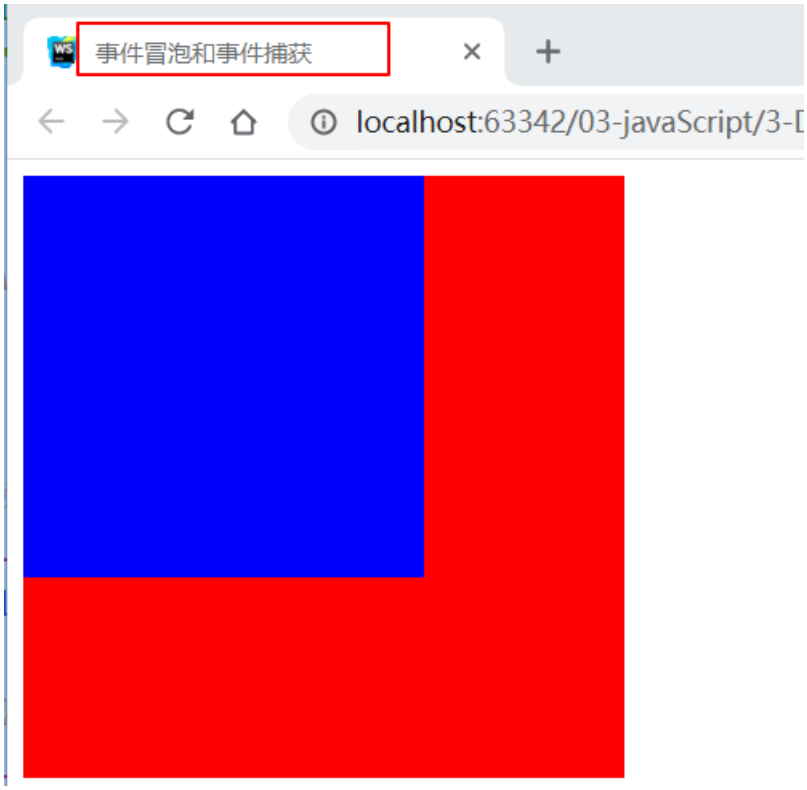
1、相关介绍

- **事件冒泡**：从内到外
- **事件捕获**：从外到内

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>事件冒泡和事件捕获</title>
</head>
<body>
<div id="div1" style="background: red; width: 300px; height: 300px">
  <div id="div2" style="background: blue;width: 200px;height: 200px"></div>
</div>

<script>
  // 获取div元素
  let div1 = document.getElementById("div1");
  let div2 = document.getElementById("div2");

  // 事件捕获
  div1.onclick = function(){
    alert(1);
  }
  // 事件冒泡
  div2.onclick = function(){

    alert(2);

  }
</script>
</body>
</html>
```

3、执行结果



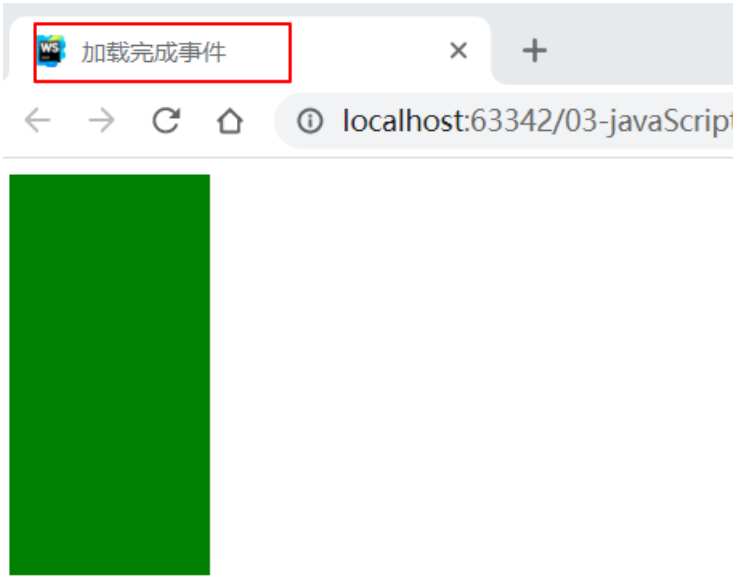

### 7.3.6 加载完成事件

当整个文档加载成功，或者一个图片加载成功，会触发加载事件 。

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>加载完成事件</title>
  <style type="text/css">
      #myDiv{width: 100px;
          height: 100px;
          background: green;}
  </style>
</head>
<body>
<div id=myDiv></div>
<script>
  // 函数一开始是不执行的：整个浏览器窗口加载完毕以后才执行
  window.onload = function(){
    setTimeout(function(){
      document.querySelector("#myDiv").style.height = '200px';
    },1000);
  };
</script>
</body>
</html>
```

2、执行结果



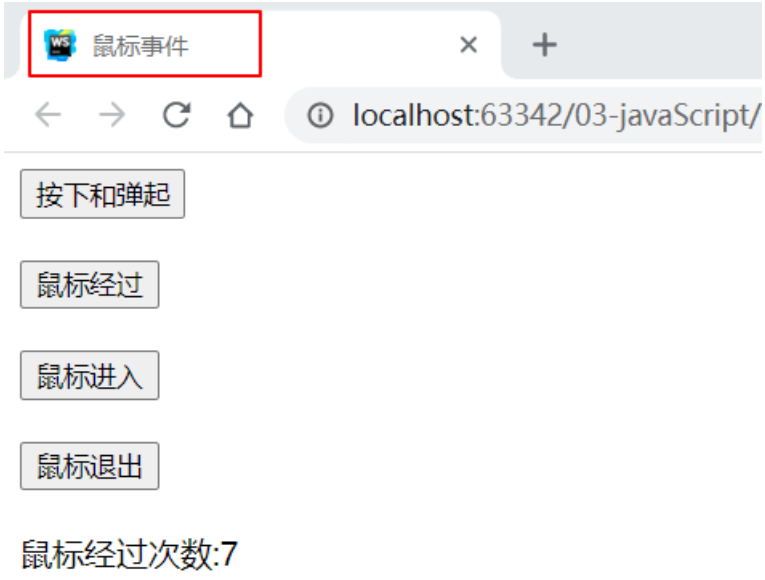

### 7.3.7 鼠标事件

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>鼠标事件</title>
</head>
<body>
<input type="button" onmousedown="down()" onmouseup="up()" value="按下和弹起" >
<br/><br/>

<input type="button" onmousemove="move()"  value="鼠标经过" ><br/><br/>

<input type="button" onmouseover="over()" value="鼠标进入"><br/><br/>

<input type="button" onmouseout="out()" value="鼠标退出" ><br/><br/>

<div id="message"></div>

<script type="text/javascript">
  // 定义number变量
  let num = 0;
  /*鼠标按下调用函数*/
  function down(){
    document.querySelector("#message").innerHTML="按下了鼠标";
  }
  /*鼠标谈起调用函数*/
  function up(){
    document.querySelector("#message").innerHTML="弹起了鼠标";
  }
  /*鼠标经过次数调用函数*/
  function move(){
    document.querySelector("#message").innerHTML="鼠标经过次数:"+(++num);
  }
  /*鼠标进入次数调用函数*/
  function over(){
    document.querySelector("#message").innerHTML="鼠标进入次数:"+(++num);
```

```
  }
  /*鼠标退出调用函数*/
  function out(){
    document.querySelector("#message").innerHTML="鼠标退出";
    num = 0;
  }
</script>
</body>
</html>
```

2、执行结果




### 7.3.8 点击事件

1、基本介绍

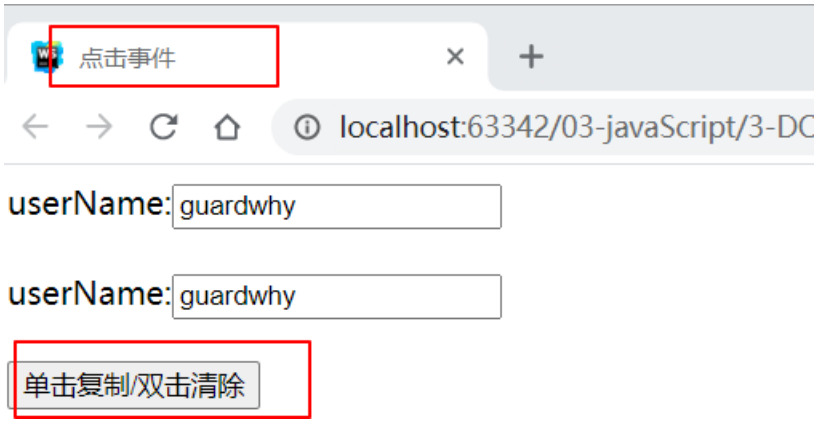
- 点击事件，由单击，双击按两个事件组成。
- 当在组件上**单击**的时候，会触发`onclick`事件，当在组件上**双击**的时候，会触发`ondblclick`事件。

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>点击事件</title>
</head>
<body>
userName:<input type="text" id="t1"> <br/><br/>
userName:<input type="text" id="t2"> <br/><br/>
<input type="button" value="单击复制/双击清除" id="btn">
<script type="text/javascript">
  //单击事件
  document.querySelector("#btn").onclick = function () {
    document.querySelector("#t2").value = document.querySelector("#t1").value;
  }

  //双击事件
  document.getElementById("btn").ondblclick = function () {
    /*清空为0*/
```

```
    document.getElementById("t1").value = "";
    document.getElementById("t2").value = "";
  }
</script>
</body>
</html>
```

3、执行结果

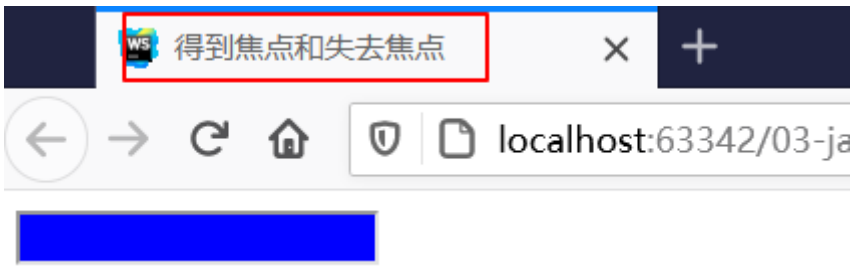



### 7.3.9 焦点事件

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>得到焦点和失去焦点</title>
</head>
<body>
<input type="text" id="txt">

</body>
<script>
  // 获得文本对象
  let txt = document.querySelector("#txt");

  // 得到焦点
  txt.onfocus = function(){

    txt.style.background = "red";
  }

  // 失去焦点
  txt.onblur = function(){

    txt.style.background = "blue";
  }
</script>
</html>
```

2、执行结果

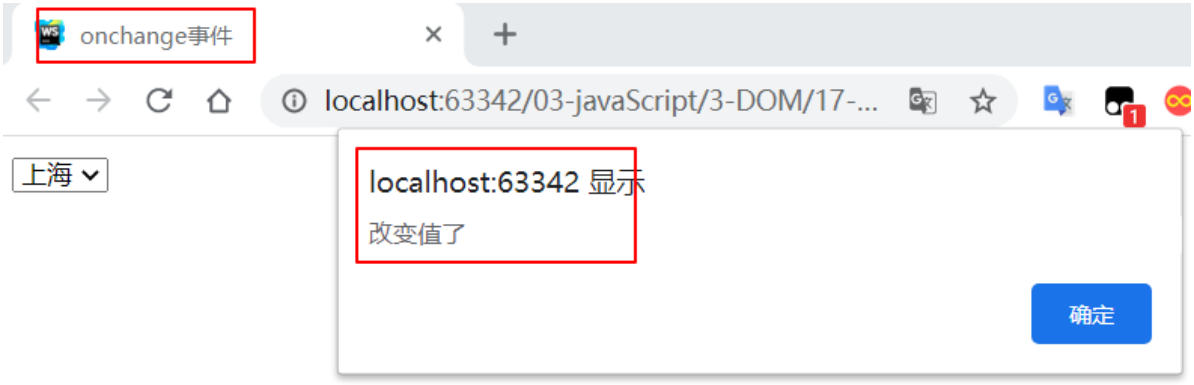



### 7.3.10 改变事件

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>onchange事件</title>
</head>
<body>
<select name="" id="city">
  <option value="">北京</option>
  <option value="">天津</option>
  <option value="">上海</option>
</select>
</body>
<script>
  // 获得文本
  let city = document.getElementById("city");
  // 文本修改值
  city.onchange = function(){
    alert("改变值了");
  }
  // 4.oninput事件可以时时获取到用户修改之后的数据， 只要用户修改了数据就会调用(执行)
  city.oninput = function (){
    console.log(this.value);
  }
</script>
</html>
```

2、执行结果



### 7.3.11 添加事件方式

1、基本介绍

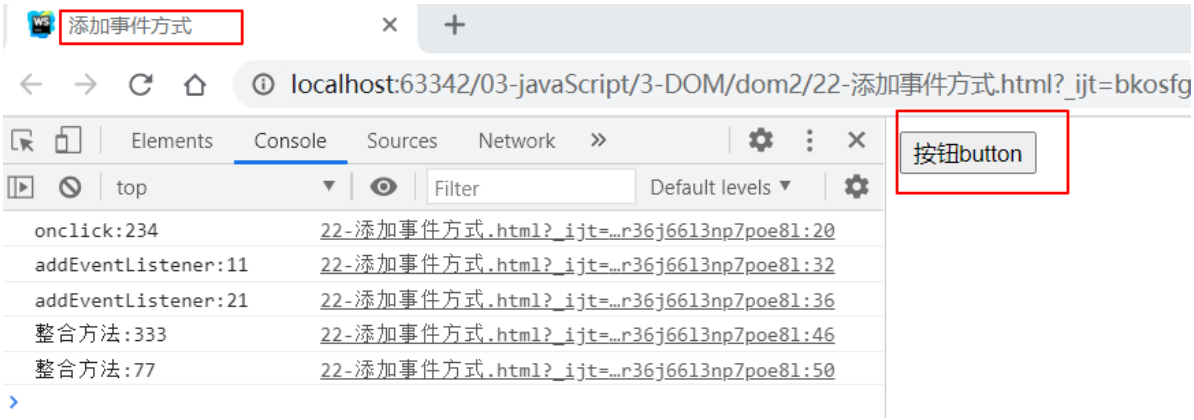
- 通过`onclick`的方式来添加。
- 通过`addEventListener`方法添加。
- 整合`attachEvent`方法添加。

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>添加事件方式</title>
</head>
<body>
<button id="btn">按钮button</button>
<script type="text/javascript">
  let oBtn = document.getElementById("btn");
  /*
   注意点: 由于是给属性赋值, 所以后赋值的会覆盖先赋值.
  */
  oBtn.onclick = function (){
    console.log("onclick:" + 123);
  }
  oBtn.onclick = function (){
    console.log("onclick:" + 234);
  }

  /*
    事件名称不需要添加on，后添加的不会覆盖先添加的,只支持最新的浏览器。
  */

  oBtn.addEventListener("click", function (){
    console.log("addEventListener:" + 11);
  });

  oBtn.addEventListener("click", function (){
    console.log("addEventListener:" + 21);
  });

    /*
    事件名称必须加上on，后添加的不会覆盖先添加的，只支持低版本的浏览器。
    */
  addEvent(oBtn, "click", function (){
    console.log("整合方法:" + 333);
  })

  addEvent(oBtn, "click", function (){
    console.log("整合方法:" + 77);
  })

  // 调用方法
  function addEvent(ele, name, func){
    if(ele.attachEvent){
      ele.attachEvent("on" + name, func);
```

```
    }else {
      ele.addEventListener(name, func);
    }
  }
</script>
</body>
</html>
```

2、执行结果




## 7.3.12 事件对象

1、事件对象基本定义

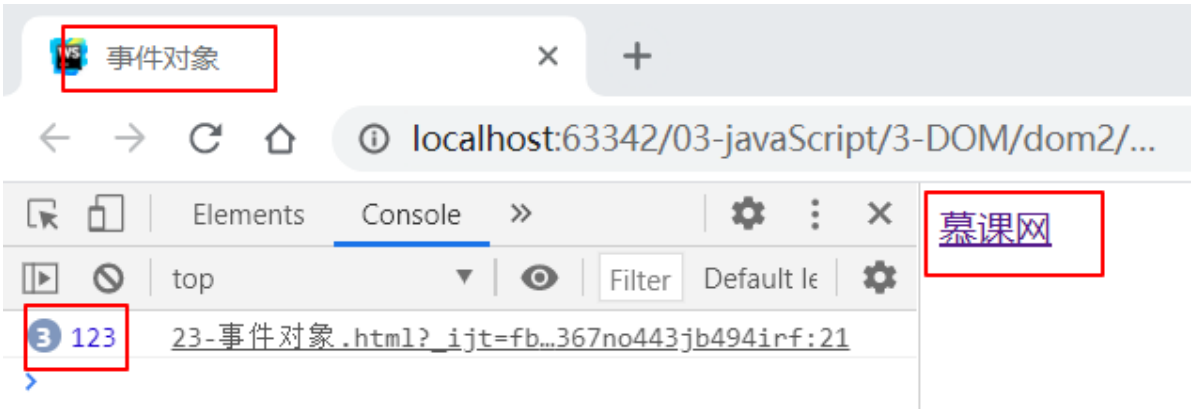
- 事件对象就是一个系统自动创建的一个对象，当注册的事件被触发的时候，系统就会自动创建事件对象。
- 当注册的事件被触发的时候，系统就会自动创建事件对象。

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>事件对象</title>
</head>
<body>
<a href="https://www.imooc.com/">慕课网</a>
<script type="text/javascript">
  /*
  事件对象的注意点：
    在高级版本的浏览器中，会自动将事件对象传递给回到函数
    在低级版本的浏览器中，不会自动将事件对象传递给回调函数
    在低级版本的浏览器中，需要通过window.event来获取事件对象
  */
let oA = document.querySelector("a");
  oA.onclick = function (event){
    // 兼容性实现
    event = event || window.event;
    // 打印
    console.log(123);
    // 阻止默认行为
    return false;
  }
```

```
</script>
</body>
</html>
```

3、执行结果



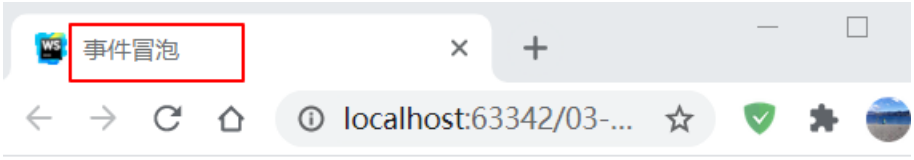

## 7.3.13 事件冒泡

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>事件冒泡</title>
  <style type="text/css">
    *{
        margin: 0;
        padding: 0;
    }
    ul{
        list-style: none;
        width: 400px;
        margin: 100px auto;
        border: 1px solid #000;
    }
    .selected{
        background: green;
    }
  </style>
</head>
<body>
<ul>
  <li class="selected">我亦飘零久，十年来，深恩负尽，死生师友。</li>
  <li>桃李春风一杯酒，江湖夜雨十年灯。</li>
  <li>雨中黄叶树，灯下白头人。</li>
  <li>绿蚁新醅酒，红泥小火炉。晚来天欲雪，能饮一杯无？</li>
  <li>君埋泉下泥销骨，我寄人间雪满头。</li>
</ul>
<script type="text/javascript">
  // 1.获取对象
  let oUl = document.querySelector("ul");
  let oLi = document.querySelector(".selected");
  // 2.点击事件
```

```
  oUl.onclick = function (event){
    event = event || window.event;
    oLi.className = "";
    let item = event.target;
    item.className = "selected";
    oLi = item;
  }
</script>
</body>
</html>
```

2、执行结果




我亦飘零久，十年来，深恩负尽，死生师友。
桃李春风一杯酒，江湖夜雨十年灯。
雨中黄叶树，灯下白头人。
绿蚁新醅酒，红泥小火炉。晚来天欲雪，能饮一杯无?
君埋泉下泥销骨，我寄人间雪满头。

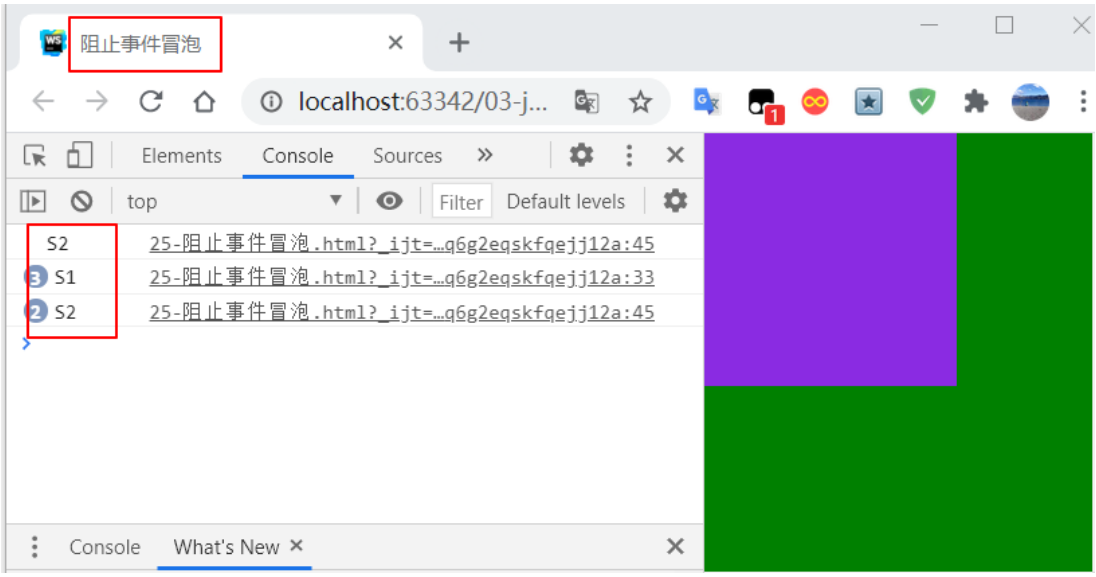
### 7.3.14 阻止事件冒泡

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>阻止事件冒泡</title>
  <style type="text/css">
    *{
        margin: 0;
        padding: 0;
    }
    .div1{
        width: 300px;
        height: 300px;
        background: green;
    }
    .div2{
        width: 150px;
        height: 150px;
        background: blueviolet;
    }
  </style>
</head>
<body>
<div class="div1" id="s1">
```

```
  <div class="div2" id="S2"></div>
</div>
<script type="text/javascript">
  // 1.拿到需要操作的元素
  let oS1 = document.getElementById("S1");
  let oS2 = document.getElementById("S2");
  // 2.注册事件监听
  oS1.onclick = function (){
    console.log("S1");
  }

  oS2.onclick = function (event){
    event = event || window.event;
    // stopPropagation方法只支持高级浏览器，cancelBubble支持低级浏览器
    if(event.cancelBubble){
      event.cancelBubble = true;
    }else {
      event.stopPropagation();
    }
    // 输出s2
    console.log("S2");
  }
</script>
</body>
</html>
```

2、代码示例



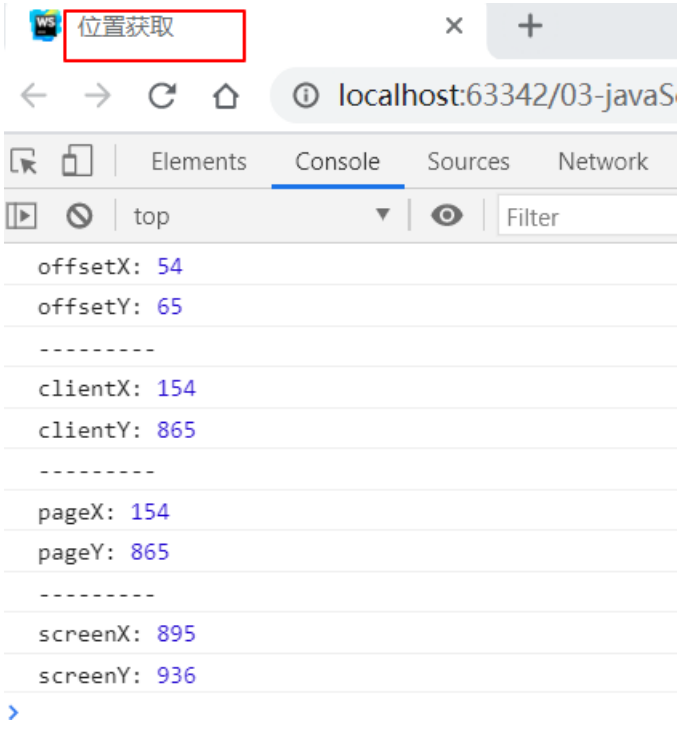

## 7.3.15 位置获取

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>位置获取</title>
  <style>
    *{
        margin: 0;
        padding: 0;
```

```
    }
    div{
        width: 100px;
        height: 100px;
        background: red;
        margin-left: 100px;
        margin-top: 800px;
    }
  </style>
</head>
<body>
<div id="box"></div>
<script type="text/javascript">
  // 获得id值
  let oDiv = document.getElementById("box");
  // 点击事件
  oDiv.onclick = function (event){
    // offsetX/offsetY: 事件触发相对于当前元素自身的位置
    console.log("offsetX:", event.offsetX);
    console.log("offsetY:", event.offsetY);
    console.log("---------");
    // clientX/clientY: 事件触发相对于浏览器可视区域的位置
    console.log("clientX:", event.clientX);
    console.log("clientY:", event.clientY);
    console.log("---------");
    // pageX/pageY:事件触发相对于整个网页的位置
    console.log("pageX:", event.pageX);
    console.log("pageY:", event.pageY);
    // screenX/screenY: 事件触发相对于屏幕的位置
    console.log("---------");
    console.log("screenX:", event.screenX);
    console.log("screenY:", event.screenY);
  }
</script>
</body>
</html>
```

2、执行结果



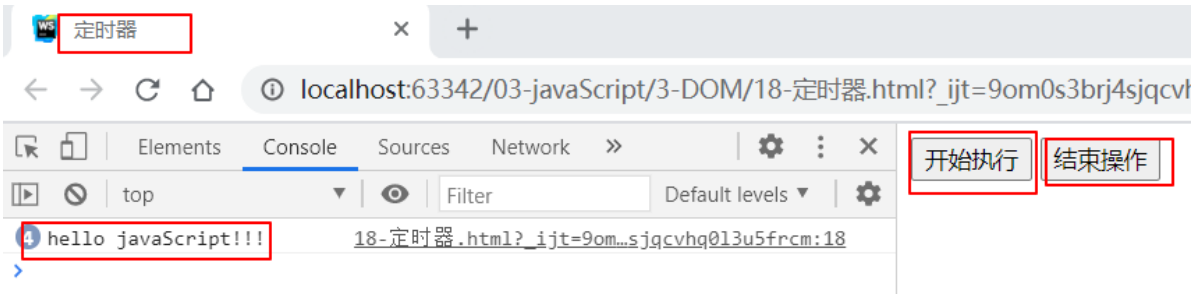

## 7.4 定时器

### 7.4.1 重复执行

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>定时器</title>
</head>
<body>
<button id="start">开始执行</button>
<button id="close">结束操作</button>

<script type="text/javascript">
  // 1.重复执行的定时器
  let start1 = document.querySelector("#start");
  let id = null;
  // 1.1 点击事件
  start1.onclick = function (){
    id = setInterval(function (){
      console.log("hello javaScript!!!");
    }, 1000);
  }
  // 1.2 关闭点击事件
  let close1 = document.querySelector("#close");
  // 停止事件
  close1.onclick = function (){
    clearInterval(id);
  }
</script>
</body>
</html>
```

2、执行结果

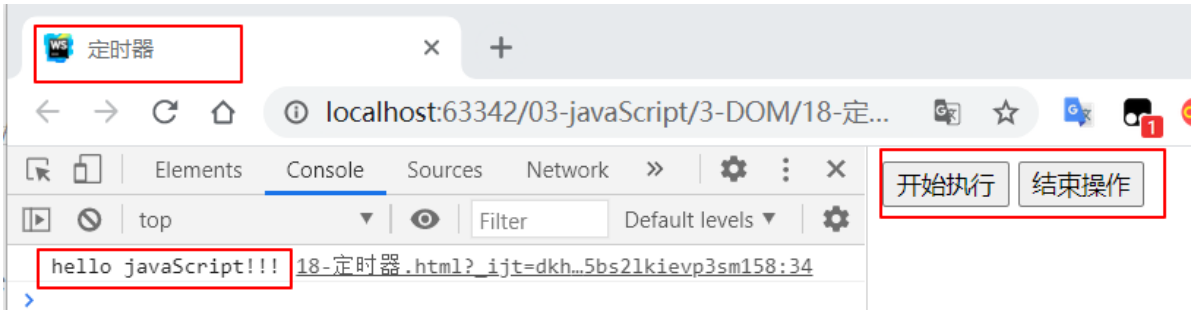



## 7.4.2 执行一次

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>定时器</title>
</head>
<body>
<button id="start">开始执行</button>
<button id="close">结束操作</button>

<script type="text/javascript">
  // 2.执行一次的定时器
  let start2 = document.querySelector("#start");
  let id = null;
  // 2.1 点击事件
  start2.onclick = function (){
    id = window.setTimeout(function (){
      console.log("hello javaScript!!!");
    }, 1000);
  }
  // 2.2 关闭点击事件
  let close2 = document.querySelector("#close");
  // 停止事件
  close2.onclick = function (){
    clearTimeout(id);
  }
</script>
</body>
</html>
```

2、执行结果



### 7.4.3 延时调用

1、相关方法

**SetTimeout()方法**

延时调用一个函数不马上执行，而是隔一段时间以后在执行，而且只会执行一次。

**ClearTimeout()方法**

可取消由`setTimeout()`方法设置的定时操作。

2、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>延时调用</title>
</head>
<body>
    <script type="text/javascript">

        var num = 1;

        /*
         * 延时调用：
         *  延时调用一个函数不马上执行，而是隔一段时间以后在执行，而且只会执行一次
         * 延时调用和定时调用的区别，定时调用会执行多次，而延时调用只会执行一次
         * 延时调用和定时调用实际上是可以互相代替的，在开发中可以根据自己需要去选择
         */
        var timer = setTimeout(function(){
            console.log(num++);
        },3000);

        //使用clearTimeout()来关闭一个延时调用
        clearTimeout(timer);

    </script>
</body>
</html>
```

## 7.5 JS 闭包

1、闭包基本定义

子函数使用父函数变量的行为叫做闭包。

2、生成闭包

当一个内部函数引用了外部函数的数据(变量/函数)时, 那么内部的函数就是闭包，所以只要满足**是函数嵌套、内部函数**引用外部函数数据。

3、闭包的特性

- 只要闭包还在使用外部函数的数据, 那么外部的数据就一直不会被释放，也就是说可以延长外部函数数据的生命周期。
- 在js中函数内部的局部变量只允许其子函数使用。
- 父函数没法使用子函数的局部变量，子函数可以使用父函数的局部变量。

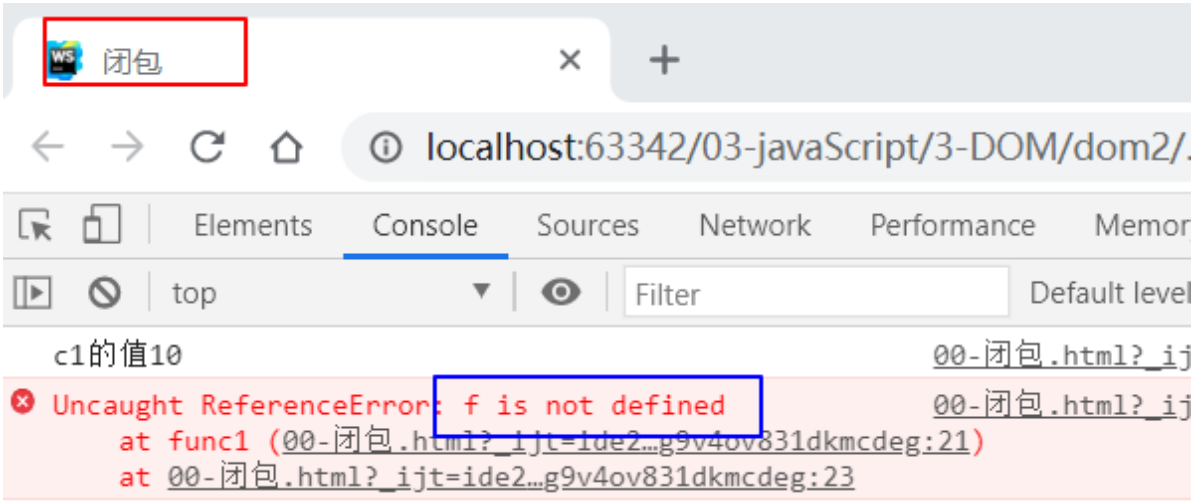
4、闭包的注意点

当后续不需要使用闭包时候, 一定要手动将闭包设置为null, 否则会出现内存泄漏。

5、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>闭包</title>
</head>
<body>
<script type="text/javascript">
  function func1(){
    //c 是局部变量 外面拿不到
    var c = 10;

    function func2(){
      var f = 20;
      console.log("c1的值" + c);
      function func3(){
        console.log("c2的值:" + c);
      }
    }
    func2();
    console.log("f的值:" + f);
  }
  func1();
</script>
</body>
</html>
```

6、执行结果




# 8-正则表达式

## 8.1 基本概念

正则表达式【`Regular Expression`】是用于匹配字符串中字符组合的模式。正则表通常被用来检索、替换那些符合某个模式【规则】的文本。

> **基本规则**

| 符号 | 作用 |
| --- | --- |
| **[a-z]** | 中括号表示`1`个字符，`-`表示一个范围，所有的小写字母 |
| **[xyz]** | `x`或`y`或`z`中任何一个字符 |
| **[^xyz]** | 如果用在中括号中，表示取反，除了`xyz`之外的所有字符 |
| **\d** | `digital` 表示`1`个数字 |
| **\w** | 表示`1`个单词字符，相当于：`[a-zA-Z0-9_]` |
| **.** | 通配符表示任意字符，如果只匹配点号。要转义 \. |
| **()** | 用于分组 |
| **{n}** | 表示前面的字符出现`n`次 |
| **{n,}** | 表示前面的字符出现大于等于`n`次 `>=n` |
| **{n,m}** | 表示前面的字符出现大于等于`n`，小于等于`m` |
| **+** | 表示前面的字符出现`1~n`次 |
| ***** | 表示前面的字符出现`0~n`次 |
| **?** | 表示前面的字符出现`0~1`次 |
| **\|** | 表示或者，几个或几组字符中出现`1`个 |
| **^** | 用于正则表达式开头，表示匹配开始 |
| **$** | 用于正则表达式结尾，表示匹配结束 |

## 8.2 正则基本特点

1、基本介绍

- 灵活性、逻辑性和功能性非常的强。
- 可以迅速地用极简单的方式达到字符串的复杂控制。
- 在`JS`中默认是模糊匹配，只要包含正则表达式就可以。如果要精确匹配，前面应该加上`^`，后面加上`$`。

| 正则表达式 | 匹配字符串 |
| --- | --- |
| **\d{3}** | 包含 3 个数字即可: a123b |
| **^\d{3}** | 以 3 个数字开头: 123b |
| **\d{3}$** | 以 3 个数字结尾: a123 |
| **ab{2}** | a 后面出现 2 次 b ：abb |
| **ab{2，}** | a 后面出现 2 次及以上的 b ： abb 或者 abbb 或者 abbbb |
| **ab{3,5}** | a 后面出现 3~5 次的 b ： abbb 或者abbbb 或者 abbbbb |
| **ab+** | a 后面出现 1~n 次 b ：ab 或者 abb 或者 abbb |
| **ab*** | a 后面出现 0~n 次 b ：ab 或者 abb 或者 abbb |
| **ab?** | a 后面出现 0~1 次 b: a 或者 ab |
| **hi\|hello** | 字符串里面有 hi 或者 hello |
| **(b\|cd)ef** | 表示 bef 或者 cdef |
| **^.{3}$** | 表示有任意三个字符的字符串 |
| **[^a-zA-Z]** | 中括号内部的 ^,表示不出现，既不出现: 大小写字母 |

## 8.3 js中的使用

在 JavaScript 中，可以通过两种方式创建一个正则表达式。

方式一：通过调用 RegExp 对象的构造函数创建

```
var regexp = new RegExp(/123/);
console.log(regexp);
```

方式二：利用字面量创建 正则表达式

```
 var rg = /123/;
```

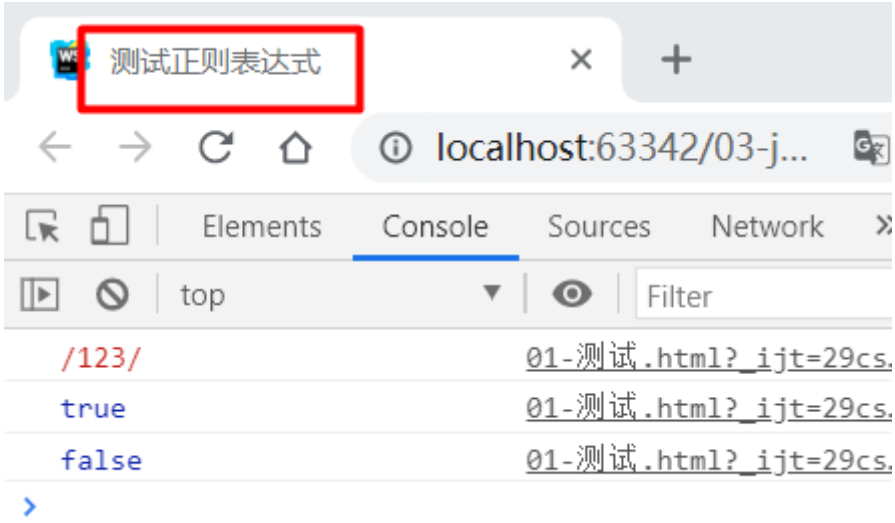
> **测试正则表达式**

1、代码实现

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>测试正则表达式</title>
</head>
<body>
    <script>
        // 1. 利用 RegExp对象来创建 正则表达式
        let regexp = new RegExp(/123/);
        console.log(regexp);
```

```
        // 2. 利用字面量创建 正则表达式
        let rg = /123/;
        // 3.test 方法用来检测字符串是否符合正则表达式要求的规范
        console.log(rg.test(123));
        console.log(rg.test('abc'));
    </script>
</body>
</html>
```

2、执行结果




## 8.4 表达式操作

> **正则表达式匹配**

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>正则表达式匹配</title>
</head>
<body>
<script type="text/javascript">
  // 定义字符串
  let str = "123abc456";
  // 指定匹配的规则
  let reg = new RegExp("A", "i");
  let res = reg.test(str);
  // 输出结果
  console.log(res); // true
</script>
</body>
</html>
```

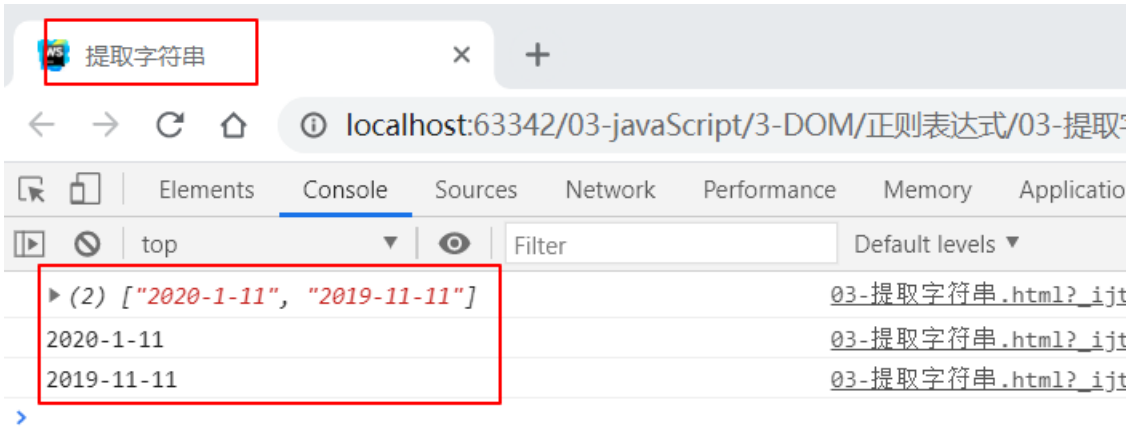
> **提取字符串**

1、代码示例

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

```
<head>
  <meta charset="UTF-8">
  <title>提取字符串</title>
</head>
<body>
<script type="text/javascript">
  // 1.定义字符串
  let str = "abc2020-1-11def2019-11-11fdjsklf";
  // 2.默认情况下在正则表达式中一旦匹配就会停止查找 /g表示查到最后
  let reg = /\d{4}-\d{1,2}-\d{1,2}/g
  let res = str.match(reg);
  // 输出结果
  console.log(res);
  console.log(res[0]);
  console.log(res[1]);
</script>
</body>
</html>
```

2、执行结果

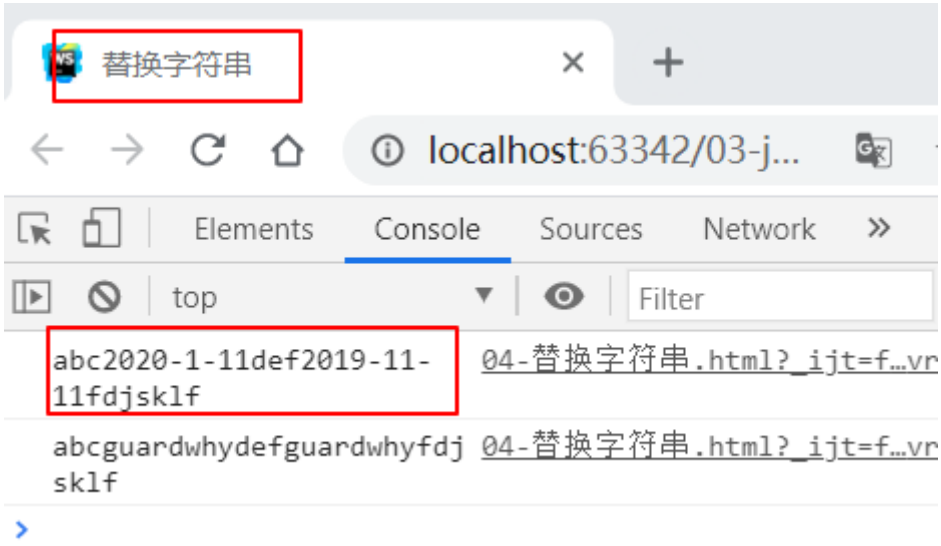



### 替换字符串

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>替换字符串</title>
</head>
<body>
<script type="text/javascript">
  // 定义字符串str
  let str = "abc2020-1-11def2019-11-11fdjsklf";
  let reg = /\d{4}-\d{1,2}-\d{1,2}/g
  let newStr = str.replace(reg, "guardwhy");
  // 输出结果
  console.log(str);
  console.log(newStr);
</script>
</body>
</html>
```

2、执行结果




# 9- JS BOM

## 9.1 BOM介绍

1、基本定义

- `Browser Object Mode` 浏览器对象模型，操作浏览器中各种对象。
- 浏览器对象模型（BOM）使 `JavaScript` 有能力与浏览器"对话"。
- `Window` 对象是 `BOM` 中所有对象的核心，除了是 `BOM` 中所有对象的父对象外，还包含一些窗口控制函数。

2、`Window` 对象

- 所有浏览器都支持 `window` 对象。它表示浏览器窗口。
- 所有 `JavaScript` 全局对象、函数以及变量均自动成为 `window` 对象的成员。
- 全局变量是 `window` 对象的属性。
- 全局函数是 `window` 对象的方法。

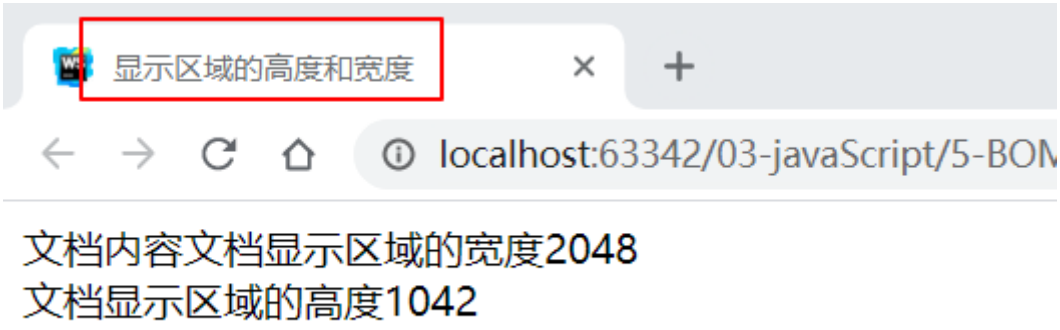
## 9.2 BOM 基本属性

> **显示区域的高度和宽度**

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>显示区域的高度和宽度</title>
</head>
<body>
<script>
  document.write("文档内容");
  document.write("文档显示区域的宽度"+window.innerWidth);
  document.write("<br>");
  document.write("文档显示区域的高度"+window.innerHeight);
</script>
</body>
</html>
```

2、执行结果

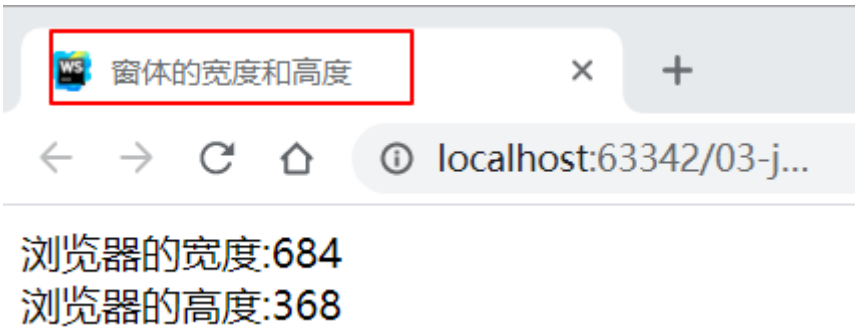



### 窗体的宽度和高度

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>窗体的宽度和高度 </title>
</head>
<body>
<script type="text/javascript">
  document.write("浏览器的宽度:"+window.outerWidth);
  document.write("<br>");
  document.write("浏览器的高度:"+window.outerHeight);
</script>
</body>
</html>
```

2、执行结果



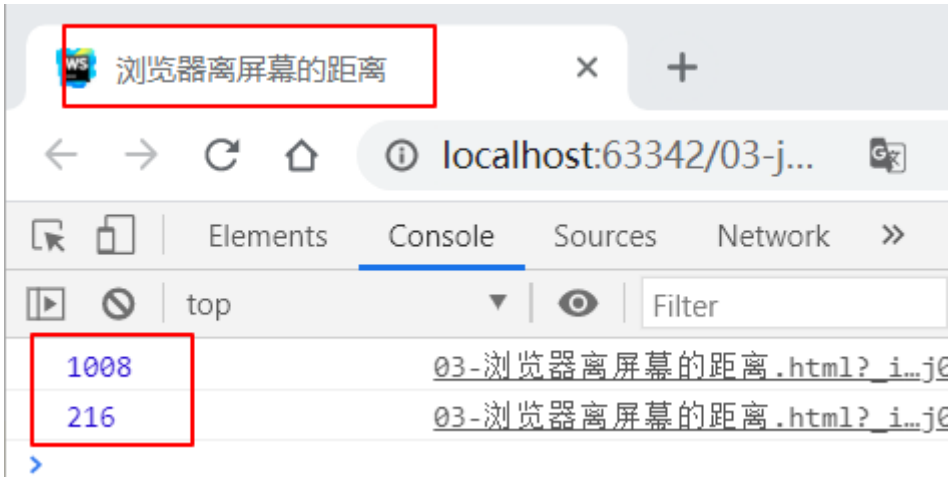

### 浏览器离屏幕的距离

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>浏览器离屏幕的距离</title>
</head>
<body>
<script type="text/javascript">
  // 浏览器离屏幕左边的距离
  console.log(window.screenLeft);
  // 浏览器离屏幕上边的距离
```

```
  console.log(window.screenTop);
</script>
</body>
</html>
```

2、执行结果

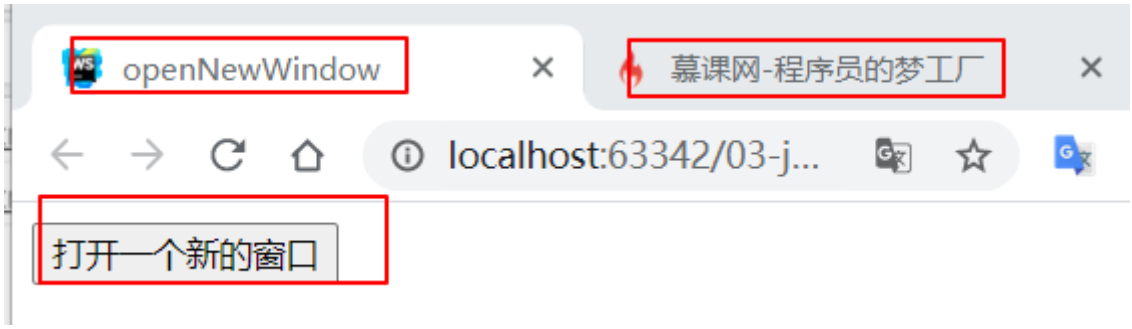



### 打开新窗口

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>openNewWindow</title>
  <script>
    function openNewWindow(){
      myWindow=window.open("https://www.imooc.com/");
    }
  </script>
</head>
<body>
<button onclick="openNewWindow()">打开一个新的窗口</button>
</body>
</html>
```

2、执行结果

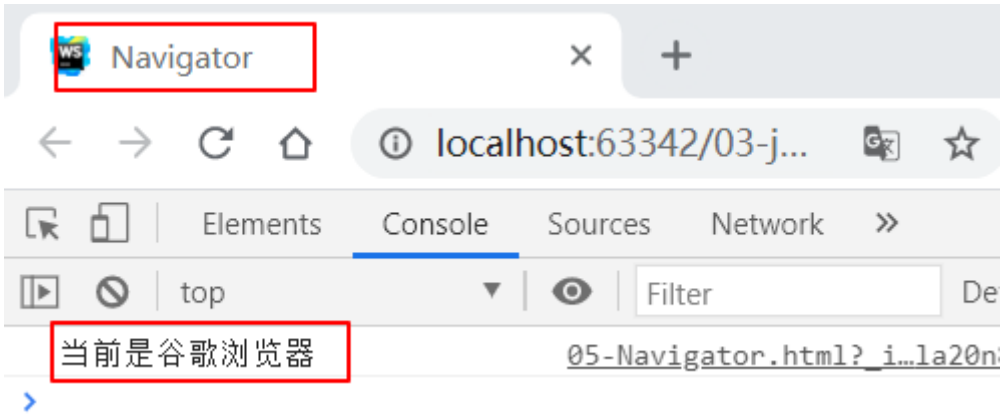


## 9.4 Navigator对象

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Navigator</title>
</head>
<body>
<script type="text/javascript">
  // Navigator: 代表当前浏览器的信息, 通过Navigator我们就能判断用户当前是什么浏览器
  let agent = window.navigator.userAgent;
  // 条件判断
  if(/chrome/i.test(agent)){
    console.log("当前是谷歌浏览器");
  }else if(/firefox/i.test(agent)){
    console.log("当前是火狐浏览器");
  }else if(/msie/i.test(agent)){
    console.log("当前是低级IE浏览器");
  }else if("ActiveXObject" in window){
    console.log("当前是高级IE浏览器");
  }
</script>
</body>
</html>
```

2、执行结果




## 9.5 Location对象

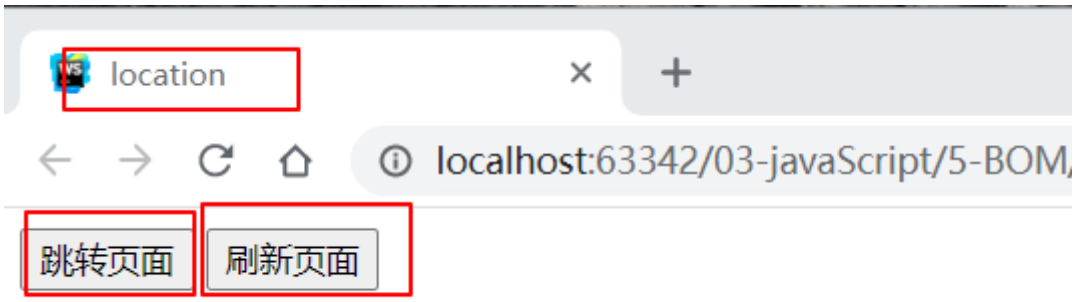
`Location`:代表浏览器地址栏的信息，通过`Location`能设置或者获取当前地址信息。

1、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>location</title>
</head>
<body>
<button id=div1>跳转页面</button>
<button id="div2">刷新页面</button>
```

```
<script type="text/javascript">
    /*跳转页面*/
  div1.onclick = function(){
    location.href = "https://www.baidu.com"
  };
    /*刷新页面*/
  div2.onclick = function(){
    location.reload();
  };
</script>
</body>
</html>
```

2、执行结果




## 9.6 History对象

`History`：代表浏览器的历史信息, 通过`History`来实现刷新/前进/后退。

1、代码示例

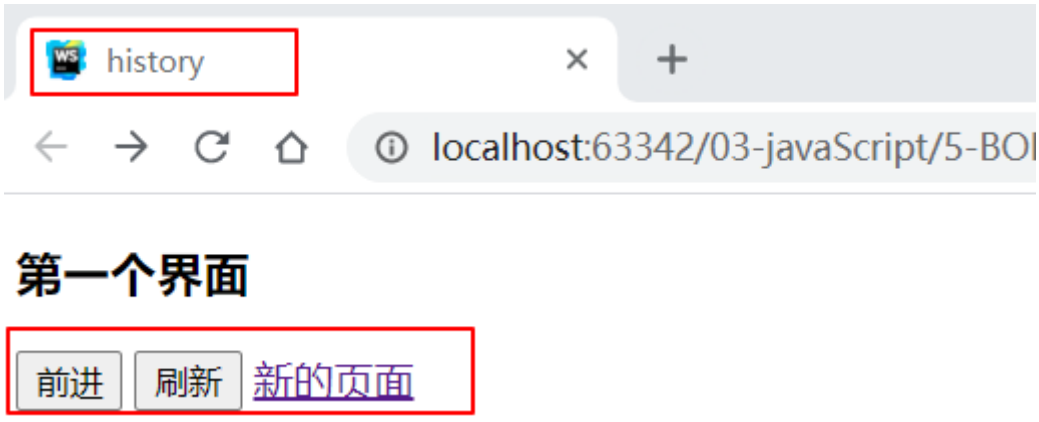
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>history</title>
</head>
<body>
<h3>第一个界面</h3>
<button id="btn1">前进</button>
<button id="btn2">刷新</button>
<a href="09-history.html">新的页面</a>

<script type="text/javascript">
  // History:代表浏览器的历史信息, 通过History来实现刷新/前进/后退
  let oBtn1 = document.querySelector("#btn1");
  let oBtn2 = document.querySelector("#btn2");

  /*
    注意点:
    只有当前访问过其它的界面, 才能通过forward/go方法前进
    如果给go方法传递1, 就代表前进1个界面, 传递2就代表进行2个界面
  */
  oBtn1.onclick = function (){
    // window.history.forward();
    window.history.go(1);
  }
```

```
  // 刷新:如果给go方法传递0,就代表刷新
  oBtn2.onclick = function (){
    window.history.go(0);
  }
</script>
</body>
</html>
```

2、代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>09-history</title>
</head>
<body>
<h3>新的页面</h3>
<button id="btn1">后退</button>
<script type="text/javascript">
  // History:代表浏览器的历史信息,通过History来实现刷新/上一步/下一步
  let oBtn1 = document.querySelector("#btn1");
  /*
   注意点:
   只有当前访问过其它的界面, back/go方法后退
   如果给go方法传递-1, 就代表后退1个界面, 传递-2就代表后退2个界面
   */

  // 后退
  oBtn1.onclick = function (){
    // window.history.back();
    window.history.go(-1);
  }
</script>
</body>
</html>
```

3、执行结果



# 9.7 JavaScript 弹窗

## 9.7.1 警告框(alert)

警告框经常用于确保用户可以得到某些信息。

1、代码实现

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>警告框</title>
</head>
<body>
    <div id='div1'>go</div>
    <script type="text/javascript">
        div1.onclick = function(){
            alert(1)
        };
    </script>
</body>
</html>
```

## 9.7.2 确认框(confirm)

1、基本介绍

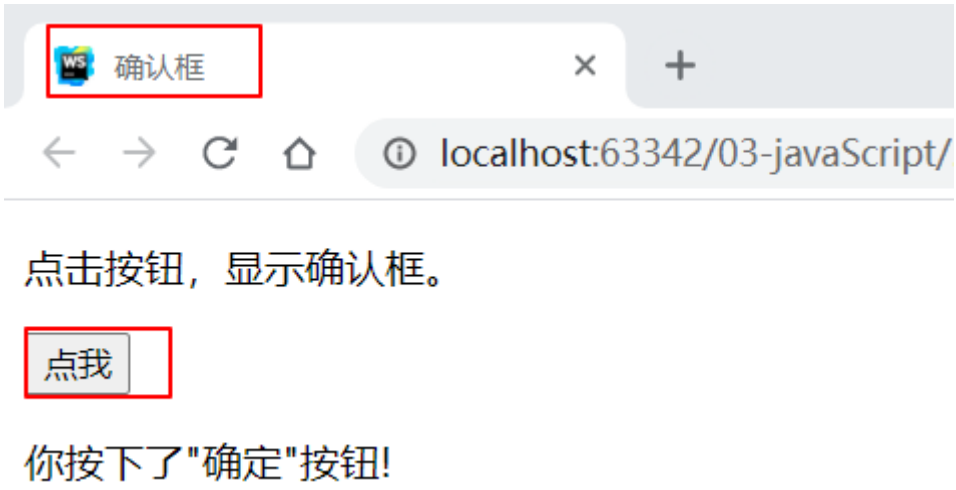
- 确认框通常用于验证是否接受用户操作。
- 当确认卡弹出时，用户可以点击 "确认" 或者 "取消" 来确定用户操作。
- 点击 "确认", 确认框返回 `true`， 如果点击 "取消", 确认框返回 `false`。

2、代码示例

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>确认框</title>
</head>
<body>
    <p>点击按钮，显示确认框。</p>
    <button onclick="myFunction()">点我</button>
    <p id="demo"></p>
    <script>
    function myFunction(){
        var x;
        var r=confirm("按下按钮!");
        if (r==true){
            x="你按下了\"确定\"按钮!";
        }
        else{
            x="你按下了\"取消\"按钮!";
        }
        document.getElementById("demo").innerHTML=x;
    }
    </script>
```

```
</body>
</html>
```

2、执行结果




### 9.7.3 提示框(prompt)

1、基本介绍

- 提示框经常用于提示用户在进入页面前输入某个值。
- 当提示框出现后，用户需要输入某个值，然后点击确认或取消按钮才能继续操纵。
- 如果用户点击确认，那么返回值为输入的值。如果用户点击取消，那么返回值为`null`。

2、代码实现

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>提示框</title>
</head>
<body>
    <p>点击按钮查看输入的对话框。</p>
    <button onclick="myFunction()">点我</button>
    <p id="demo"></p>
    <script>
    function myFunction(){
        var x;
        var person=prompt("请输入你的名字","Harry Potter");
        if (person!=null && person!=""){
            x="你好 " + person + "! 今天感觉如何?";
            document.getElementById("demo").innerHTML=x;
        }
    }
    </script>
</body>
</html>
```