

## js中的helloworld

document.write()可以向body中输出一个内容

console.log()向控制台输出一个内容

## js代码编写的位置

①可以写在标签的属性中，比如：onclick、href中，但是他们属于结构与行为耦合，不方便维护，不推荐使用。

```
<button onclick="alert('讨厌，你点我干嘛~~');">点我一下</button>

<a href="javascript:alert('让你点你就点！！！');">你也点我一下</a>
<a href="javascript:;"/>你也点我一下
```

②写在script标签中写

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title></title>
    <script type="text/javascript">

      alert("我是script标签中的代码！！");

    </script>
  </head>
  <body>
```

③写在外部的js文件中，通过script标签引入，将文件的路径写在src后面

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title></title>

    <!--可以将js代码编写到外部js文件中，然后通过script标签引入-->
    <script type="text/javascript" src="js/script.js"></script>
```

某个script标签已经引入外部文件后，就不能再在标签中编写js代码了，即使编写了浏览器也会忽略，如果想写，就在编写一个script标签

# js基本语法

①注释

/\* \*/多行注释

//单行注释

②js严格区分大小写

③每一句以分号结尾

如果不写分号，浏览器会自动添加，但是会消耗一些系统资源，而且有时候会加错分号

## 字面量和变量

js中用var来声明一个变量

```
// 声明变量  
// 在js中使用var关键字来声明一个变量  
var a;  
  
// 为变量赋值  
a = 123;  
a = 456;  
a = 123124223423424;  
  
// 声明和赋值同时进行  
var b = 789;|
```

## 标识符

驼峰法命名：首字母 小写，每个单词的开头字母大写，其余小写

helloworld xxxYyyZzz

js底层保存标识符采用的是Unicode编码，理论上讲所有utf-8中的内容都可以作为标识符

## 数据类型

String字符串

Number数值

Boolean布尔值

Null空值

Undefined未定义

Object对象

---

前5种属于**基本数据类型**，Object属于**引用数据类型**

typeof检查一个变量类型

```
console.log(typeof b);
```

## 强制的类型转换

### string的转换

①调用被转换数据类型的**toString()方法**, 该方法不会影响原变量, 只会将结果返回, null和undefined这两个值没有toString方法

```
var a = 123;  
  
//调用a的toString()方法  
//调用xxx的yyy()方法, 就是xxx.yyy()  
var b = a.toString();
```

②调用string()函数来将参数转换为字符串

对于number和Boolean底层就是调用toString

### number的准换

①使用number()函数

纯数字的字符串可准换为number

非纯数字准换为NaN

空串转换为0

布尔准换为1/0

null准换为0

undefined准换为NaN

②**parseInt()函数**把一个字符串准换为整数, 从左到右的整数部分保存下来

**parseFloat()函数**把一个字符串转换为浮点数, 从左到右的浮点数部分保存下来

```
a = "123.456.789px";  
a = parseFloat(a);
```

```
"123.456"
```

对于非字符串值先转换为字符串然后进行操作

```
a = 198.23;
a = parseInt(a);

console.log(typeof a);
console.log(a);
```

```
控制台]，点右上角工具条相应按钮可切换控制台
: ] "number" /day06/06.转换为Number.htm
: ] "198" /day06/06.转换为Number.html (68)
```

可以在parseInt()中传递第二个参数，来指定数字的进制

```
a = "070";

//可以在parseInt()中传递一个第二个参数，来指定数字的进制
a = parseInt(a,10);
```

### Boolean值的转换

Boolean()函数

数字：除了0和NaN，其他的都是true

字符串：除了空串，其他的都是true

null、undefined：转为false

对象：true

---

### String字符串

一些特殊符号可以使用\进行转义

```
str = "我说:\\"今天天气真不错! \\"";
```

\"	表示 "
\'	表示 '
\n	表示换行
\t	制表符
\\\	表示\

### Number数值

js中可以表示的数字最大值 Number.MAX\_VALUE，超过之后会返回一个Infinity（无穷）

Number.MIN\_VALUE 零以上的最小值

NaN 特殊的数字， Not a Number,typeof也是一个数字

使用js进行浮点数运算，可能得到一个不精确的结果，这是所有语言的通病

0x表示16进制的数字

0开头表示8进制的数字

0b开头表示2进制的数字，但并不是所有的浏览器支持

**Boolean布尔值**

**Null空值**

**Undefined未定义**

**Object对象**

## 运算符

也叫操作符

①无引号的非number值进行算数运算，先将值转换为number再运算，任何值与NaN做运算都得NaN

②对两个字符串加法操作，进行拼串操作

③任何值对字符串进行加法操作，先转换为字符串，再做拼串操作（利用这一特殊点，可以将任意数据类型转换为空串）

```
var c = 123;  
c = c + "";  
  
//console.log(result);  
console.log(typeof c);  
!!!  
  
, 点右上角工具条相应按钮可切换控制台  
"string" /day06/09.运算符.  
"123" /day06/09.运算符.html (67)
```

常见输出：

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

```
"c = 123"
```

④若是-/\*操作，将两个值都转为number进行操作即可（利用这一特点看可以做隐式的类型转换：-0 \*1 /1）

## 一元运算符

- + 对于非number类型的值，先将其转换为number然后在运算（利用这一特性，可以在其他数据类型前面用+将其转换为number）

## 非布尔值的与或运算

会先将其转换为布尔值，然后在运算，然后再返回原值

&&与运算：第一个值为true，直接返回第二个值

第一个值为false，直接返回第一个值（结合串联电路）

||或运算：第一个值为true，直接返回第一个值

第一个值为false，直接返回第二个值（结合并联电路）

## 关系运算符

关系运算符比较两个值之间的大小关系，关系成立会返回true，不成立返回false

有非数值的话会先将其化为数字，然后再运算。

如果符号两侧都是字符串，不会转化为数字，直接比较两个字符串的字符编码Unicode，一位一位进行比较，如果两位一样则直接比较下一位（可以借此用来对英文进行排序）

如果想要比较两个字符串型的数字时，一定要转型（如利用加号进行转型）（想利用用户输入值进行大小比较的情景）

```
console.log("11123123123123123123" < +"5");
```

任何值和NaN作比较都是false

## 编码输出

\u四位Unicode编码（默认16进制）：代表此Unicode编码表示的字符

（html中用的是&#Unicode编码，但此处编码需要转为十进制）

## 相等运算符

如果值的类型不同，会自动转换为相同的类型然后比较，大部分是转换为数字

**undefined==null**

**NaN不和任何值相等**

检查一个值是不是NaN用isNaN()函数来判断

**全等运算符**

==== 和 == 类似，但是不会做类型的转换，如果类型不一样，直接返回false

!== 和!=类似，不会做类型的转换，类型不一样，直接返回true

## 条件表达式

也叫三元运算符

语法：条件表达式?语句1:语句2

条件表达式为true，执行语句1，返回执行结果

条件表达式为false，执行语句2，返回执行结果

条件表达式的结果为一个非布尔值，先转换为布尔值，然后在运算

```
var a = 10;
var b = 20;

a > b ? alert("a大"):alert("b大");| //获取a和b中的最大值
var max = a > b ? a : b;

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

var max = a > b ? (a > c ? a :c) : (b > c ? b : c);
console.log("max = "+max);|
```

## 运算符优先级

### 运算符的优先级

- `、[]、new
- 0
- ++、--
- !、~、+(单目)、-(单目)、typeof、void、delete
- %、\*、/
- +(双目)、-(双目)
- <<、>>、>>>
- <、<=、>、>=
- ==、!=、=====
- &
- ^
- |
- &&
- ||
- ?:
- =、+=、-=、\*=、/=、%==、<<=、>>=、>>>=、&=、^=、|=
- ,

## 代码块

用{}来表示代码块，他只具有分组功能，代码块中的内容对外完全可见

## 条件分支语句

也叫switch语句，switch条件表达式和case 表达式如果匹配，从当前的case开始执行，后面的case也会跟着执行

语法：switch (条件表达式) {

case 表达式:

语句

break;

case 表达式:

语句

break;

default:

语句

break;

}

```
switch(parseInt(score/10)){
    case 10:
    case 9:
    case 8:
    case 7:
    case 6:
        console.log("合格");
        break;
    default:
        console.log("不合格");
        break;}
```

js中除法会得到一个浮点数（与其他语言不同），要得到整数商还要进行取整操作

## break和continue

break

只有一个break会立即终止离他最近的那个循环语句

可以为循环语句创建一个label，来标识当前的循环

语法为label:循环语句，这样可以结束指定循环

```
outer:  
for(var i=0 ; i<5 ; i++){  
    console.log("@外层循环"+i)  
    for(var j=0 ; j<5; j++){  
        break outer;  
        console.log("内层循环:"+j);  
    }  
}
```

continue**终止当次的循环**，也可以像break一样使用标签

## 对象

对象分类：

**内建对象**：ES标准定义的对象，在任何的ES的实现中都可以使用，如**Math, String, Number, Boolean, Function, Object**

**宿主对象**：JS运行环境提供的对象，主要是指浏览器中的对象，如BOM（浏览器对象模型）和DOM（文档对象模型）这两组对象，像console和document

**自定义对象**：由开发人员定义

## 自定义对象

使用**new关键字**调用的函数，是构造函数constructor，**构造函数是专门用来创建对象的函数**。如  
`var obj = new Object();`

也可以直接使用字面量来创建一个对象：`var obj = {};`

```
//创建一个对象  
//var obj = new Object();  
  
/*  
 * 使用对象字面量来创建一个对象  
 */  
var obj = {};
```

向对象中添加属性，语法：对象.属性名:属性值

```
//向obj中添加一个name属性  
obj.name = "孙悟空";  
//向obj中添加一个gender属性  
obj.gender = "男";  
//向obj中添加一个age属性  
obj.age = 18;|
```

读取对象中的属性，语法：对象.属性名

修改对象中的属性，语法：对象.属性名:新值

删除对象的属性，语法：delete 对象.属性名

## 属性名和属性值

属性名不强制要求遵守标识符的规范，想用啥用啥，按使用时尽量按照规范

如果需要特殊的属性名，创建和读取都要采用另一种方式：

语法：对象["属性名"]=属性值

使用[]这种方式读属性时更加灵活，[]中的属性名可以用一个变量表示，

```
obj["123"] = 789;  
obj["nihao"] = "你好";  
  
var n = "123";  
  
console.log(obj[n]);|
```

属性值可以使任一种数据类型，也可以是另外一个对象、函数

```
//创建一个对象  
var obj2 = new Object();  
obj2.name = "猪八戒";  
  
//将obj2设置为obj的属性  
obj.test = obj2;  
  
console.log(obj.test.name);  
  
obj.sayName = function(){  
    console.log(obj.name);|  
};|
```

如果一个函数作为一个对象的属性保存，那么我们称这个函数是这个对象的方法

**in运算符**：通过该运算符可以检查一个对象中是否含有指定的属性，有则返回true，没有返回false

语法："属性名" in 对象

```
console.log("test" in obj);|
```

## 基本数据类型和引用数据类型

基本数据类型：

```
16         var a = 123;
17         var b = a;
18         a++;
19
20         console.log("a = "+a);
21         console.log("b = "+b);
22
23
24     </script>
```

控制台

前是 [边改边看控制台]，点右上角工具条相应按钮可切换控制台

Web浏览器] "a = 124" /day09/08.基本和引用数据类型.html (20)  
Web浏览器] "b = 123" /day09/08.基本和引用数据类型.html (21)

引用数据类型：

```
23         var obj = new Object();
24         obj.name = "孙悟空";
25
26         var obj2 = obj;
27
28         //修改obj的name属性
29         obj.name = "猪八戒";
30
31         console.log(obj.name);
32         console.log(obj2.name);
33
34
35     </script>
36 </head>
```

控制台

当前是 [边改边看控制台]，点右上角工具条相应按钮可切换控制台

[Web浏览器] "猪八戒" /day09/08.基本和引用数据类型.html (31)  
[Web浏览器] "猪八戒" /day09/08.基本和引用数据类型.html (32)

js中的变量都是保存在栈内存中

基本数据类型的值直接在栈内存汇总存储，值与值之间是独立存在，修改一个不会影响另一个

```
var a = 123;  
var b = a;
```

变量	值
b	123
a	123

栈内存



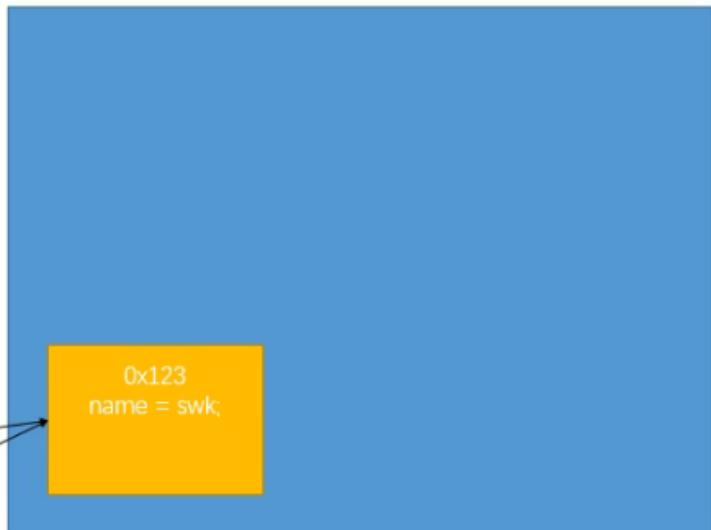
堆内存

对象是保存在**堆内存**中，每创建一个新的对象，就会在堆内存中开辟一个新的内存空间，这时变量保存的是对象的地址（对象的引用）

```
var obj = new Object();  
obj.name = swk;  
var obj2 = obj;]
```

变量	值
obj2	0x123
obj	0x123

栈内存

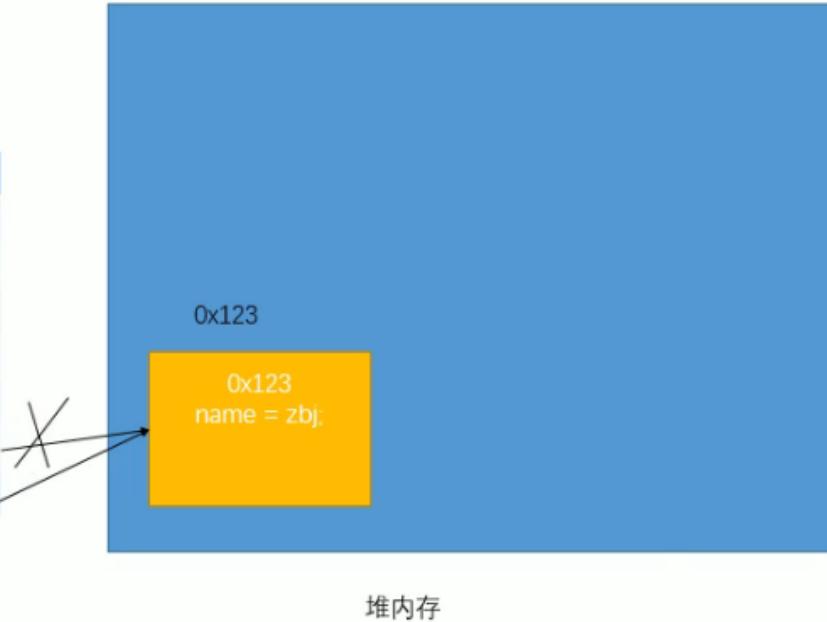


堆内存

当修改变量值的时候，对其他变量不会产生影响，只会使当前对象的引用断开

```
var obj = new Object();
obj.name = "swk";
var obj2 = obj;
obj.name = "zbj"
obj2 = null;
```

变量	值
obj2	null
obj	0x123



当比较两个基本类型数据时，是比较值

当比较两个引用数据类型值，是比较对象的**内存地址**，哪怕值是一样的，地址不同，也会返回false

## 对象字面量

```
/*
 * 使用对象字面量来创建一个对象
 */
var obj = {};
```

```
var obj2 = {
    name: "猪八戒",
    age: 28,
    gender: "男",
    test: {name: "沙和尚"}
};
```

定义的时候就可以指定属性

属性名可以加引号也可以不加，**如果想使用特殊的属性名，则必须加引号**

# 函数

函数也是一个对象，函数中可以封装一些功能，也具有普通对象的功能，也可以进行属性的定义等功能。

①

```
var fun = new Function("代码");//代码需要用双引号括起来
```

调用函数：fun();

但是，实际开发中很少使用到构造函数来创建一个函数。

②

一般使用**函数声明**来创建函数

语法：

```
function 函数名([形参1,形参2,...]) {
```

语句

```
} //最后不用写分号
```

```
function fun2(){  
    console.log("这是我的第二个函数~~~");  
}
```

③

或者，用**函数表达式**来创建一个函数

语法：

```
var 函数名 = function([形参1,形参2,...]) {
```

语句

```
};
```

```
var fun3 = function(){  
    console.log("我是匿名函数中封装的代码");  
};
```

//就相当于使用一个匿名函数，然后在赋予一个函数名，与函数声明法没太大区别

## 函数的参数

函数的解析器不会检查实参的类型，可以是任意类型，可以是对象、函数

```
fun(mianji(10));
```

```
/* ●
 * mianji()
 * - 调用函数
 * mianji
 * - 函数对象
 */
```

参数过多时，可以将所有参数封装成一个对象传进去

```
function sayHello(o){
    //console.log("o = "+o);
    console.log("我是"+o.name+",今年我"+o.age+"岁了,"+"我是一个"+o
}

//sayHello("猪八戒",28,"高老庄","男");
//创建一个对象
var obj = {
    name:"孙悟空",
    age:18,
    gender:"男",
    address:"花果山"
};

sayHello(obj);
```

也不会检查实参的数量，多余的实参不会被赋值

如果实参的数量少于形参的数量，则没有实参对应的形参将会是undefined

## 函数的返回值

```
function sum(a , b , c){
    //alert(a + b +c);
```

```
    var d = a + b + c;
```

```
    return d;
}
```

```
//调用函数
```

```
//变量result的值就是函数的执行结果|
```

```
//函数返回什么result的值就是什么
```

```
var result = sum(4,7,8);
```

return后的语句都不会执行

例子：

```
function isOu(num){  
    return num % 2 == 0;  
}  
  
var result = isOu(15);  
  
console.log("result = "+result);
```

返回值可以是任意一个数据类型，可以是一个对象、函数

## 立即执行函数

```
* 立即执行函数  
* |  
*/  
(function(){  
    alert("我是一个匿名函数~~~");  
})();  
  
(function(a,b){  
    console.log("a = "+a);  
    console.log("b = "+b);  
})(123,456);
```

将匿名函数用括号括起来表示一个整体，然后后面再加一个括号进行立即使用

立即执行函数只会使用一次

## 方法

如果一个**函数**作为一个**对象的属性**保存，那么我们称这个函数是这个对象的方法

(只是名称上的区别，没有本质区别)

```

obj.sayName = function(){
    console.log(obj.name);
};

function fun(){
    console.log(obj.name);
};

//console.log(obj.sayName);
//调方法
obj.sayName();
//调函数
fun();
```

## 对象中属性的枚举

语法:

```
for(var 变量 in 对象){
}
```

对象中有几次就循环几次，每次循环就会将对象中的一个属性赋值给变量

```

* for...in语句 对象中有几个属性，循环体就会执行几次
* 每次执行时，会将对象中的一个属性的名字赋值给变量
*/
for(var n in obj){
    console.log("属性名："+n);

    console.log("属性值："+obj[n]);
}
```

## 全局作用域

全局作用域:

- 直接编写在script标签中的js代码，都在全局作用域
- 它在页面打开时创建，页面关闭时销毁
- 全局作用域中有一个全局对象window，它代表的是一个浏览器的窗口，他由浏览器创建我们可以直接使用
- 在全局作用域中，创建的变量都会作为**window的属性保存**，创建的函数都会作为**window的方法保存**
- 全局作用域中定义的变量是全局变量，在页面的任意位置都可以访问到

```
var a = 10;  
var b = 20;  
  
console.log(window.b);
```

## 变量的声明提前

使用var关键字的变量，会在所有的代码执行之前被声明

```
var a;  
  
/*  
 * 变量的声明提前  
 * - 使用var关键字声明的变量，会在所有的代码执行之前被声明  
 */  
console.log("a = "+a);  
  
a = 123;
```

a会被指明undefined，但不会被报错

如果声明变量没用var，就会报错

## 函数的声明提前

(形式上感觉跟变量的声明提前刚好相反)

使用**函数声明**形式创建的函数**function 函数名(){}**，它会在所有的代码执行之前就被创建，此种形式的函数想在哪调用在哪调用

使用**函数表达式**创建的函数不会被声明提前，不能提前调用

函数声明：

```
fun();  
  
function fun(){  
    console.log("我是一个fun函数");  
}  
  
var fun2 = function(){  
    console.log("我是fun2函数");  
};  
  
//fun2();
```

函数表达式：

```
fun2();  
  
function fun(){  
    console.log("我是一个fun函数");  
}  
  
var fun2 = function(){  
    console.log("我是fun2函数");  
};
```

台]，点右上角工具条相应按钮可切换控制台

"我是一个fun函数" /day10/08.变量的声明提前.html (21)

"Uncaught TypeError: undefined is not a function"

## 函数作用域

在函数作用域中操作一个变量时，现在自身作用域中找，没有则在上一级作用域中找，直到找到全局作用域中。

如果想直接访问全局作用域的变量，可以直接 `window.变量`

函数作用域中也存在 **变量的声明提前** 和 **函数声明的提前**

**在函数中，不使用var声明的变量都会成为全局变量，同样相当于window.变量 的省略形式，变成window中的属性**

**例子：**

```
var c = 33;

function fun5(){
    console.log("c = "+c);
    c = 10;
}

fun5();

//在全局输出c
console.log("c = "+c);
```

看控制台]，点右上角工具条相应按钮可切换控制台  
器] "c = 33" /day10/09.函数作用域.html (67)  
器] "c = 10" /day10/09.函数作用域.html (74)

例子：

```
var a = 123;
function fun() {
    alert(a);
    a = 456;
}
fun();
alert(a);
```

undefined

456

```
var a = 123;
function fun(a) {
    alert(a);
    a = 456;
}
fun();
alert(a);
```

undefined

123

## this

解析器在调用函数每次都会向函数内部传递一个隐含的参数，这个隐含的参数就是this，this指向的是一个对象，我们称为函数执行的上下文对象

根据函数**调用方式**的不同，this会指向不同的对象

1.以**函数形式**调用时，this就是window（其实也就是window.fun()的省略形式）

2.以**方法的形式**调用，this就是使用这个调用方法的对象

```
<title></title>
</head>
<body>
    <script type="text/javascript" >
        function fun(){
            console.log(this);
        }
        fun(123,456);
        var obj={
            name:"孙悟空",
            sayname:fun
        };

        obj.sayname(); //以**函数形式**调用时，this就是window
        fun(); //以**方法的形式**调用，this就是使用这个调用方法的对象

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

3.在**构造函数**中可以使用this来引用**新建的对象**（在vue中经常用到）

补充：**在响应函数中，响应函数是给谁绑定的，this就是指谁** 并且极个别有一些例外，不用太在意，如IE8中为事件绑定响应函数的方法attachEvent()中this就代表的window

补充例子：

```
var obj={
    name:"obj",
    sayname:function(){
        console.log(this.name);
    }
};
```

obj

new\_file.html:14

注意：例子中第一种形式，就是在定义一个对象的方法，以**方法的形式**调用，this就是使用这个调用方法的对象

教程[https://www.bilibili.com/video/BV1YW411T7GX?p=62&spm\\_id\\_from=pageDriver](https://www.bilibili.com/video/BV1YW411T7GX?p=62&spm_id_from=pageDriver)

## 使用工厂方法创建对象

工厂法：

```
function createPerson(name , age ,gender){  
    //创建一个新的对象  
    var obj = new Object();  
    //向对象中添加属性  
    obj.name = name;  
    obj.age = age;  
    obj.gender = gender;  
    obj.sayName = function(){  
        alert(this.name);  
    };  
    //将新的对象返回  
    return obj;  
}  
  
var obj2 = createPerson("猪八戒",28,"男");  
var obj3 = createPerson("白骨精",16,"女");  
var obj4 = createPerson("蜘蛛精",18,"女");
```

## 创建构造函数

构造函数就是一个专门创建某一类对象的函数，创建方式和普通函数没有区别

构造函数习惯上首字母大写

构造函数和普通函数区别在于**调用方式**的不同，普通函数直接调用，构造函数用new关键字调用（也就是当一个函数被调用的时候用的是new调用的，那么此时他就成为了一个构造函数）

调用构造函数执行流程：

立即创建一个新的对象

将新建的**对象设置为函数中的this**，在构造函数中可以使用this来引用新建的对象

逐行执行函数中的代码

将新建的对象作为返回值返回（所以一般构造函数不像普通函数需要return）

使用同一个构造函数创建的对象，我们称之为**一类对象**，也将一个构造函数成为一个类，通过创建函数创建的对象成为该类的**实例**

```
function Person(name , age , gender){  
    this.name = name;  
    this.age = age;  
    this.gender = gender;  
    this.sayName = function(){  
        alert(this.name);  
    };  
}  
  
var per = new Person("孙悟空",18,"男");  
var per2 = new Person("玉兔精",16,"女");  
var per3 = new Person("奔波霸",38,"男");
```

**instanceof**: 使用**instanceof**可一检查一个对象是否是一个类的实例

```
/*  
 * 使用instanceof可以检查一个对象是否是一个类的实例  
 */  
console.log(per instanceof Person);  
T
```

this的情况总结

**this**的情况:

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

## 构造函数的修改

```
function Person(name , age , gender){  
    this.name = name;  
    this.age = age;  
    this.gender = gender;  
    //向对象中添加一个方法  
    this.sayName = function(){  
        alert("Hello大家好，我是："+this.name);  
    };  
}  
  
//创建一个Person的实例  
var per = new Person("孙悟空",18,"男");  
  
per.sayName();|
```

创建一个person构造函数，函数中有一个sayname方法，一开始是在函数内部创建，但是这样的话每次调用构造函数都会产生一个新的对象、新的实例、新的方法，所有实例的sayname都是不同的

改进：

```
function Person(name , age , gender){  
    this.name = name;  
    this.age = age;  
    this.gender = gender;  
    //向对象中添加一个方法  
    this.sayName = fun;  
}  
  
//将sayName方法在全局作用域中定义  
function fun(){  
    alert("Hello大家好，我是："+this.name);  
};  
  
//创建一个Person的实例  
var per = new Person("孙悟空",18,"男");
```

将方法放到全局变量中去定义，这样所有实例的方法只用创建一次，不用每次都创建

# 原型对象

上面讲到的，在全局作用域中定义sayname方法，这么做有不足：①污染了全局作用域的命名空间（就是可能出现命名冲突的情况）②放在全局作用域中很不安全

## 原型prototype：

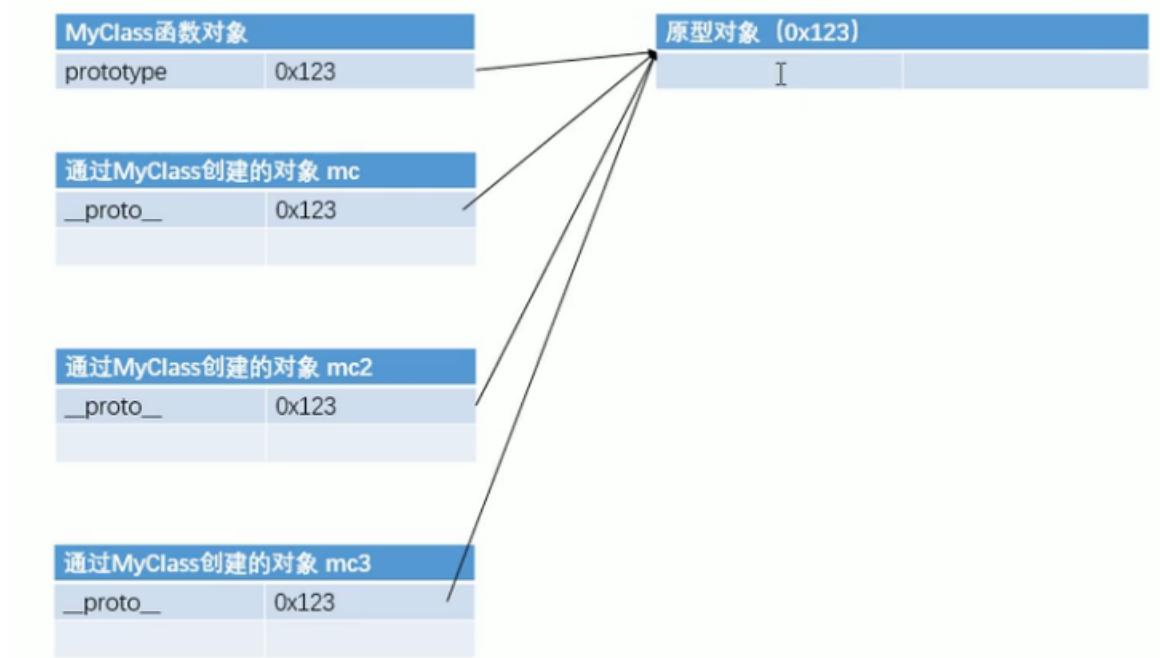
我们创建每一个函数（无论是函数还是构造函数），解析器都会向函数中添加一个**属性 prototype**，这个属性对应一个对象——**原型对象**

如果函数作为普通函数调用，prototype没用任何作用

如果函数作为构造函数调用，所创建的对象都会用一个**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

原型对象相当于一个公共的区域，所有同一个类的实例都可以访问这个原型对象（**同一个类的实例包括那个构造函数的prototype就是同一个**）

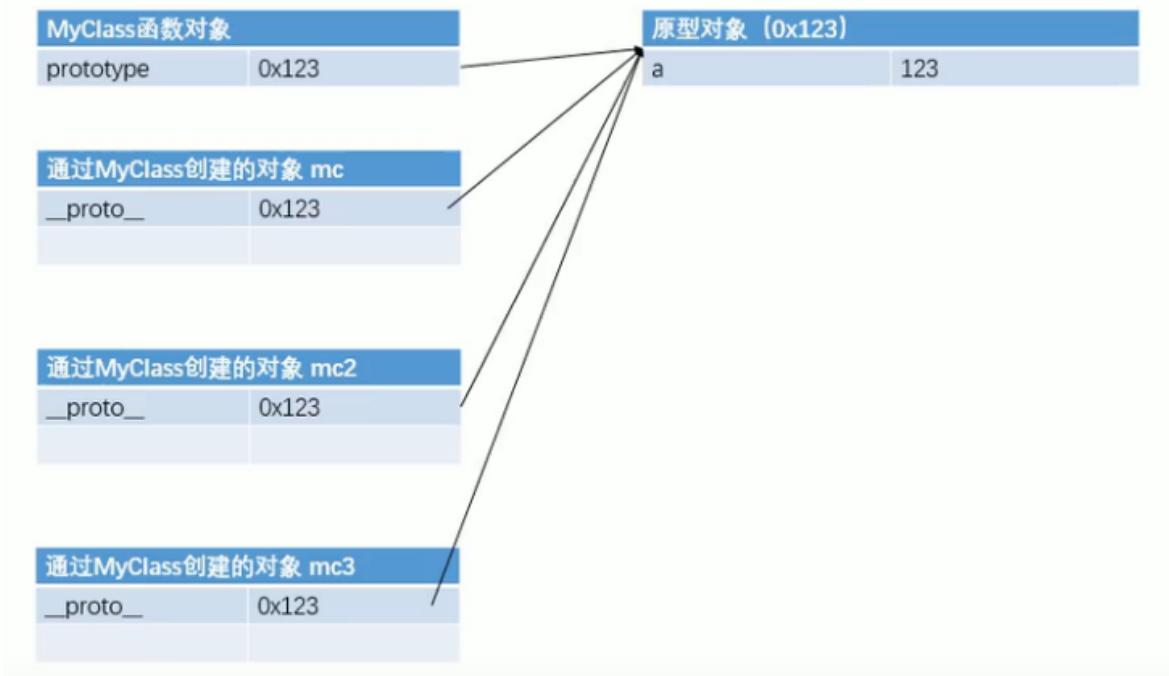
```
function MyClass(){  
}  
  
var mc = new MyClass();  
  
var mc2 = new MyClass();
```



当我们访问一个对象的**属性或方法**时，他先会在自身中找，没找到去**原型对象**中找，如下图

```
//向MyClass的原型中添加属性a  
MyClass.prototype.a = 123;
```

```
console.log(mc.a);  
控制台]，点右上角工具条相应按钮可切换控制台  
[ "123" /day11/03.原型.html (34) ]
```



上一节中的例子可以修改为下面：

```
Person.prototype.sayName = function(){  
    alert("Hello大家好，我是："+this.name);  
};
```

我们可以将公共的内容，添加到原型对象中这样也不会影响到全局作用域

检查原型中是否有某个属性：

用 **in** 检查：对象中没有，原型中有，也会返回true

```
1 |     function MyClass(){}
2 |
3 |
4 |
5 |     //向MyClass的原型中添加一个name属性
6 |     MyClass.prototype.name = "我是原型中的名字";
7 |
8 |     var mc = new MyClass();
9 |
10|     //console.log(mc.name);
11|
12|     //使用in检查对象中是否含有某个属性时，如果对象中没有但是原型中有，也会返回true
13|     console.log("name" in mc);
14|
15|
16|     </script>
17| 
```

控制台

浏览器] "true" /day11/04.原型.html (22)

用**hasOwnProperty()**方法来检查对象中有没有该属性：对象本身中有才会返回true，只有原型中有返回false

```
//可以使用对象的hasOwnProperty()来检查对象自身中是否含有该属性
console.log(mc.hasOwnProperty("name"));
```

控制台

浏览器] "false" /day11/04.原型.html (26)

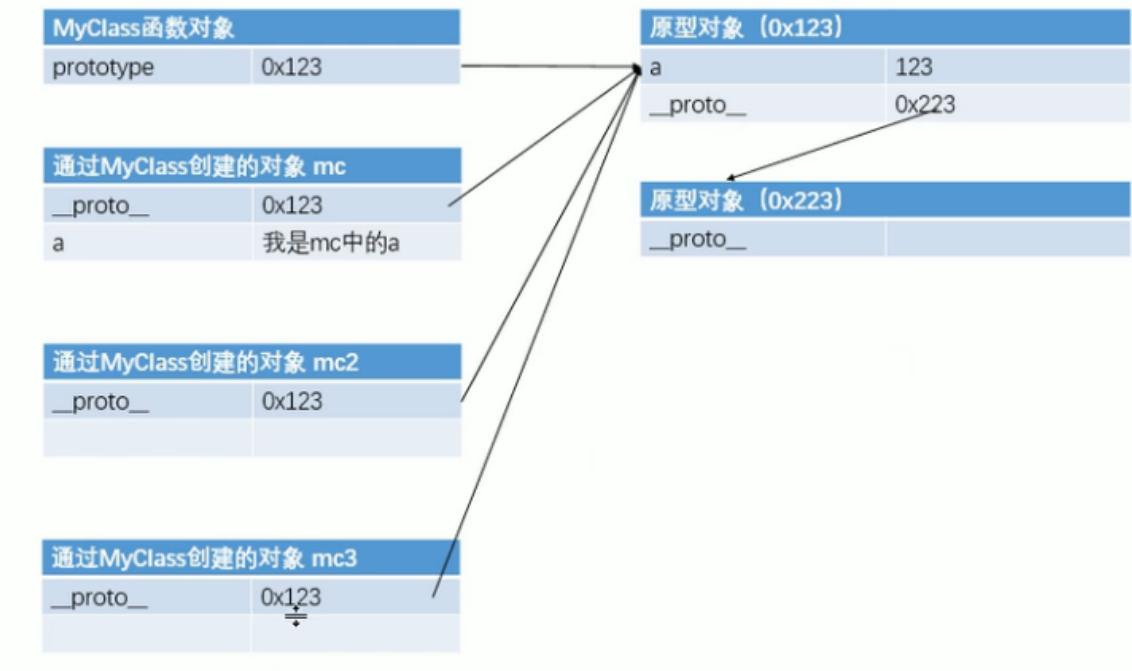
hasOwnProperty()方法没有放在对象的原型中，而是放在**这个对象的原型对象的原型**中

```
console.log(mc.__proto__.hasOwnProperty("hasOwnProperty"));

</script>
</head>
```

控制台

浏览器] "false" /day11/04.原型.html (30)



原型对象也是对象，所以它也有原型

使用一个对象的属性或方法时，会先在自身中寻找

如果没有则去原型对象中寻找，如果原型对象中没有

则去原型的原型中寻找

**没有原型的原型的原型**

如下图：

```

41
42 console.log(mc.__proto__.__proto__.hasOwnProperty("hasOwnProperty"));
43
44
45 </script>
46 <ad>

```

控制台

当前是 [边栏]，点右上角工具条图标可切换控制台

[Web浏览器] "true" /day11/04.原型.html (42)

## toString

当我们直接在页面中打印一个对象时，实际上是输出的对象的 **toString()**方法的返回值，他保存在**原型的原型**中

```
function Person(name , age , gender){  
    this.name = name;  
    this.age = age;  
    this.gender = gender;  
}  
  
//创建一个Person实例  
var per = new Person("孙悟空",18,"男");  
  
//当我们直接在页面中打印一个对象时，事件上是输出的对象的toString()方法的返回值  
  
var result = per.toString();  
console.log("result =" + result);  
  
</script>
```

控制台]，点右上角工具条相应按钮可切换控制台  
] "result =[object Object]" /day11/05.toString.html (20)

如果我们希望在输出对象时不输出[ object object],可以修改原型的原型的toString()方法

```
//修改Person原型的toString  
Person.prototype.toString = function(){  
    return "Person[name='"+this.name+"',age='"+this.age+"',gender='"+this.gender+"']";  
};  
  
34     console.log(per2);  
35     console.log(per);  
36  
37  
38     </script>  
39     </head>  
40     <body>  
41     </body>  
42 </html>  
43
```

控制台]，点右上角工具条相应按钮可切换控制台  
[Web浏览器] "Person[name=猪八戒,age=28,gender=男]" /day11/05.toString.html (34)  
[Web浏览器] "Person[name=孙悟空,age=18,gender=男]" /day11/05.toString.html (35)

## 垃圾回收

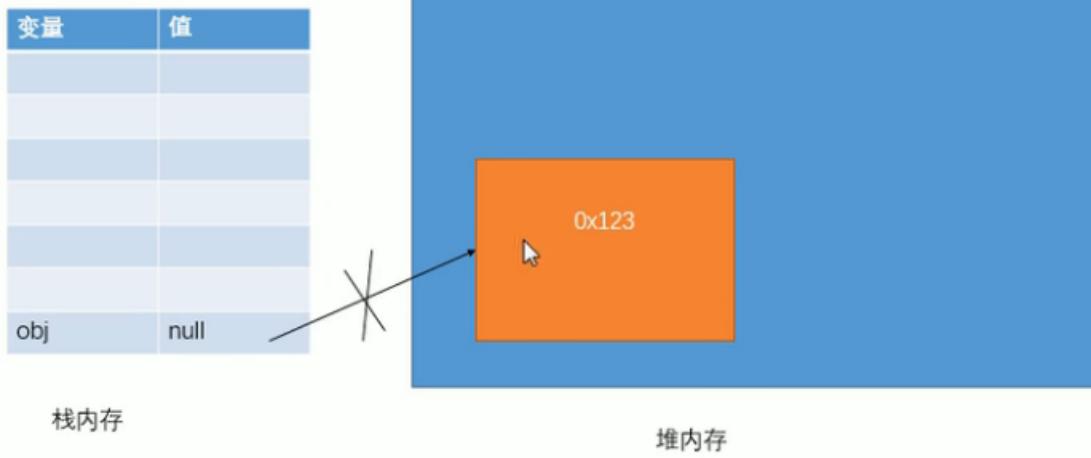
程序运行过程中会产生垃圾

垃圾积攒过多，会导致程序运行的速度过慢，

当一个对象没有任何的变量或属性对它进行引用，此时我们将永远无法操作该对象

这种对象就是一个**垃圾**，这种对象过多会占用大量的内存空间，导致程序运行变慢

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



当一个对象如: `obj=null`, 代表上图对象名与分配的堆内存空间断开连接

在JS中拥有自动的垃圾回收机制, 会自动将这些垃圾对象从内存中销毁, 我们不需要也不能进行垃圾回收的操作

当我们不再需要对一个对象进行操作后, 可以用`obj=null`, 标明此对象没用了, 浏览器自动会清理

## 数组

创建一个数组, 数组都用数字索引来替代普通对象的属性值,

```
// 创建数组对象
var arr = new Array();
```

数组中的值可以是任意的数据类型, 也可以是对象、数组 (二维数组、三位数组....)

```
//数组中的元素可以是任意的数据类型
arr = ["hello", 1, true, null, undefined];

//也可以是对象
var obj = {name:"孙悟空"};
arr[arr.length] = obj;
arr = [{name:"孙悟空"}, {name:"沙和尚"}, {name:"猪八戒"}];

console.log(arr);

</script>

```

控制台，点右上角工具条相应按钮可切换控制台

```
] "[object Object],[object Object],[object Object]" /day
```

### length属性

```
/*
console.log(arr.length);
console.log(arr);

/*
 * 修改length
 */
arr.length = 10;

//向数组的最后一个位置添加元素
arr[arr.length] = 70;
arr[arr.length] = 80;
arr[arr.length] = 90;
```

### 数组的创建

#### ① 创建数组对象

```
//创建数组对象
var arr = new Array();
```

创建时可以指定元素

```
//使用构造函数创建数组时，也可以同时添加元素，将要添加的元素作为构造函数的参数传递
//元素之间使用，隔开
var arr2 = new Array(10, 20, 30);
```

#### ② 使用字面量来创建数组

```
//使用字面量来创建数组
```

```
//语法：[]|
```

```
var arr = [];
```

创建时也可以指定元素

```
//使用字面量创建数组时，可以在创建时就指定数组中的元素
```

```
var arr = [1,2,3,4,5,10];
```

二者均可创建时指定元素，但是区别如下：

```
//创建一个数组数组中只有一个元素10
```

```
arr = [10];
```

```
//创建一个长度为10的数组
```

```
arr2 = new Array(10);
```

```
console.log(arr2.length);
```

控制台]，点右上角工具条相应按钮可切换控制台

```
控制台] ",,," /day11/08.数组.html (32)
```

```
控制台] "10" /day11/08.数组.html (32)
```

## 数组的方法

### push方法

```
var result = arr.push("唐僧","蜘蛛精","白骨精");
```

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

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

```
</script>
```

```
</head>
```

查看控制台]，点右上角工具条相应按钮可切换控制台

```
控制台] "孙悟空,猪八戒,沙和尚,唐僧,蜘蛛精,白骨精" /day11/09.数组的方法.html (20)
```

```
控制台] "result = 6" /day11/09.数组的方法.html (21)
```

## pop方法

```
/*
 * pop()
 * - 该方法可以删除数组的最后一个元素，并将被删除的元素作为返回值返回
 */
result = arr.pop();
console.log(arr);
console.log("result = "+result);
</script>
</head>
<body>
</body>
html>
```

控制台]，点右上角工具条相应按钮可切换控制台  
[控制台] "孙悟空,猪八戒,沙和尚,唐僧,蜘蛛精,白骨精" /day11/09.数组的方法.html (22)  
[控制台] "孙悟空,猪八戒,沙和尚,唐僧,蜘蛛精" /day11/09.数组的方法.html (31)  
[控制台] "result = 白骨精" /day11/09.数组的方法.html (32)

## unshift方法

```
/*
 * unshift()
 * - 向数组开头添加一个或多个元素，并返回新的数组长度
 * - 向前边插入元素以后，其他的元素索引会依次调整
 */
console.log(arr);

arr.unshift("牛魔王","二郎神");

console.log(arr);
```

控制台]，点右上角工具条相应按钮可切换控制台  
[控制台] "孙悟空,猪八戒,沙和尚,唐僧,蜘蛛精,白骨精" /day11/09.数组的方法.html (37)  
[控制台] "牛魔王,二郎神,孙悟空,猪八戒,沙和尚,唐僧,蜘蛛精,白骨精" /day11/09.数组的方法.html (38)

## shift方法

```
/*
 * shift()
 * - 可以删除数组的第一个元素，并将被删除的元素作为返回值返回
 */
result = arr.shift();]
result = arr.shift();]

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

控制台]，点右上角工具条相应按钮可切换控制台  
[控制台] "牛魔王,二郎神,孙悟空,猪八戒,沙和尚,唐僧,蜘蛛精,白骨精" /day11/09.数组的方法.html (50)  
[控制台] "孙悟空,猪八戒,沙和尚,唐僧,蜘蛛精,白骨精" /day11/09.数组的方法.html (51)  
[控制台] "result = 二郎神" /day11/09.数组的方法.html (52)

## foreach方法

一般是用for来遍历数组，js提供一个方法来遍历数组：foreach

foreach只支持IE8以上的浏览器

foreach需要一个函数作为参数

像这种函数，由我们创建但是不由我们调用的，我们称为**回调函数**

数组中有几个元素**函数就会执行几次**，每次执行时，浏览器会将遍历到的元素**以实参的形式传递进来**，我们可以来定义形参，来读取这些内容

```
var arr = ["孙悟空", "猪八戒", "沙和尚", "唐僧"];  
/*  
 * forEach()方法需要一个函数作为参数  
 * - 像这种函数，由我们创建但是不由我们调用的，我们称为回调函数  
 */  
arr.forEach(function(){  
    console.log("hello");  
});
```

控制台输出：

```
[浏览器] "hello" /day11/12.forEach.html (23)  
[浏览器] "hello" /day11/12.forEach.html (23)  
[浏览器] "hello" /day11/12.forEach.html (23)  
[浏览器] "hello" /day11/12.forEach.html (23)
```

浏览器会在回调函数中传递**三个参数**：

第一个参数，就是当前正在遍历的**元素**

第二个参数，就是当前正在遍历的元素的**索引**

第三个参数，就是正在遍历的**数组**

```
arr.forEach(function(value , b , c){  
    console.log("value = "+value);  
});
```

控制台输出：

```
[控制台] "value = 孙悟空" /day11/12.forEach.html (28)  
[控制台] "value = 猪八戒" /day11/12.forEach.html (28)  
[控制台] "value = 沙和尚" /day11/12.forEach.html (28)  
[控制台] "value = 唐僧" /day11/12.forEach.html (28)  
[控制台] "value = 白骨精" /day11/12.forEach.html (28)
```

```

        arr.forEach(function(value , index , c){
            console.log("index = "+index);
        });
    
```

浏览器] "index = 0" /day11/12.forEach.html (29)  
 浏览器] "index = 1" /day11/12.forEach.html (29)  
 浏览器] "index = 2" /day11/12.forEach.html (29)  
 浏览器] "index = 3" /day11/12.forEach.html (29)  
 浏览器] "index = 4" /day11/12.forEach.html (29)

```

8=         arr.forEach(function(value , index , c){
9          console.log(c);
0      });
1
2
    
```

控制台

浏览器] "孙悟空,猪八戒,沙和尚,唐僧,白骨精" /day11/12.forEach.html (29)  
 浏览器] "孙悟空,猪八戒,沙和尚,唐僧,白骨精" /day11/12.forEach.html (29)  
 浏览器] "孙悟空,猪八戒,沙和尚,唐僧,白骨精" /day11/12.forEach.html (29)  
 浏览器] "孙悟空,猪八戒,沙和尚,唐僧,白骨精" /day11/12.forEach.html (29)  
 浏览器] "孙悟空,猪八戒,沙和尚,唐僧,白骨精" /day11/12.forEach.html (29)

## slice方法

可以用来从数组提取指定元素，**该方法不会改变元素数组**，而是将截取到的元素封装到一个**新数组**中返回

两个参数：

1. 截取开始的位置的索引，**包含开始索引**

2. 截取结束的位置的索引，**不包含结束索引**，第二个可以省略不写，此时会截取从开始索引往后的所有元素

索引可以传递一个负值，如果传递一个负值，则从后往前计算

```
var arr = ["孙悟空","猪八戒","沙和尚","唐僧","白骨精"];
```

```
var result = arr.slice(1,4);
```

```
console.log(result);
```

看控制台] 点右上角工具条相应按钮可切换控制台

器] "猪八戒,沙和尚,唐僧" /day11/13.数组的方法

## splice方法

可以用于删除数组中的指定元素

使用splice () **会影响到原数组**, **会将指定元素从原数组中删除**, 并将被删除的元素作为返回值返回

参数:

第一个, 表示开始位置的索引

第二个, 表示删除的数量

第三个及以后, 可以传递一些新的元素, 这些元素将会自动插入到开始位置索引 (即**第一个参数**) 前边 (**可以利用此特点在指定位置添加元素**)

```
arr = ["孙悟空", "猪八戒", "沙和尚", "唐僧", "白骨精"];
var result = arr.splice(1, 2);

//console.log(arr);
console.log(result);

</script>
```

看控制台, 点右上角工具条相应按钮可切换控制台  
器] "猪八戒,沙和尚" /day11/13.数组的方法.html (46)

```
arr = ["孙悟空", "猪八戒", "沙和尚", "唐僧", "白骨精"];
var result = arr.splice(1, 0, "牛魔王", "铁扇公主", "红孩儿");

console.log(arr);
//console.log(result);
```

制台] 点右上角工具条相应按钮可切换控制台  
] "孙悟空,牛魔王,铁扇公主,红孩儿,猪八戒,沙和尚,唐僧,白骨精" /day11/13.数组

## concat方法

**可以连接两个或多个数组, 也可以直接输入元素进行连接**, 并将新的数组返回, 该方法**不会对原数组产生影响**

```
var arr = ["孙悟空", "猪八戒", "沙和尚"];
var arr2 = ["白骨精", "玉兔精", "蜘蛛精"];
var arr3 = ["二郎神", "太上老君", "玉皇大帝"];

/*
 * concat() 可以连接两个或多个数组，并将新的数组返回
 * - 该方法不会对原数组产生影响
 */
var result = arr.concat(arr2, arr3);

console.log(result);
```

X

台]，点右上角工具条相应按钮可切换控制台

"孙悟空,猪八戒,沙和尚,白骨精,玉兔精,蜘蛛精,二郎神,太上老君,玉皇大帝"

```
var result = arr.concat(arr2, arr3, "牛魔王", "铁扇公主");
```

## join方法

该方法可以将数组转换为一个字符串

该方法不会对原数组产生影响，而是将转换后的字符串作为结果返回

在join () 中可以指定一个字符串作为参数，这个字符串将会成为数组中元素的**连接符**，如果不指定连接符，则**默认用逗号**

```
arr = ["孙悟空", "猪八戒", "沙和尚"];
result = arr.join();
console.log(result);

</script>
```

控制台]，点右上角工具条相应按钮可切换控制台

器] "孙悟空,猪八戒,沙和尚"

/day12/01.数组的剩余方法

```
arr = ["孙悟空", "猪八戒", "沙和尚"];
result = arr.join("@-@");
console.log(result);
```

控制台]，点右上角工具条相应按钮可切换控制台

器] "孙悟空@-@猪八戒@-@沙和尚"

/day12/01.数组的剩余方法

## reverse方法

该方法用来反转数组(前边的去后边，后边的去前边)

该方法会直接修改原数组

## sort方法

可以用来对数组中的元素进行排序

也会影响原数组，默认会按照Unicode编码进行排序

对数字进行编排时，按照Unicode编码可能会得到错误的结果

The screenshot shows a browser's developer tools console. The code entered is:

```
arr = ["b", "d", "e", "a", "c"];
/*
 * sort()
 * - 可以用来对数组中的元素进行排序
 * - 也会影响原数组，默认会按照Unicode编码进行排序
 */
arr.sort();
arr.reverse();
```

The console output shows the array after sorting and reversing:

```
浏览器] "a,b,c,d,e" /day12/01.数组的剩余方法.html (46)
```

我们可以自己指定排序的结果，在sort()添加一个**回调函数**，来指定排序规则

回调函数中需要定义两个形参，如图

使用哪个元素调用不确定，但是肯定的是在**数组中a一定在b前边**

浏览器会根据回调函数的返回值来定元素的顺序：

如果返回一个**大于0的值**，则元素会**交换位置**

如果返回一个**小于0的值**，则元素**位置不变**

如果返回一个0，则认为两个元素相等，也不交换位置

```
arr = [5,4,2,1,3,6,8,7];

arr.sort(function(a,b){

    //前边的大
    if(a > b){
        return 1;
    }else if(a < b){
        return -1;
    }else{
        return 0;
    }

});
```

浏览器] "1,2,3,4,5,6,7,8" /day12/01.数组的剩余方法.html (77)

改进：

如果需要升序排列，则返回a-b

如果需要降序排列，则返回b-a

```
arr = [5,4,2,1,3,6,8,7];
arr.sort(function(a,b){
    //升序排列
    return a-b;
    //降序排列
    //return b-a;
});
console.log(arr);
```

## 函数的方法

call和apply都是**函数对象**的方法，需要通过函数对象来调用（**函数对象是函数名，不能在函数后面加括号**）

当对函数调用call() 和 apply() 都会调用**函数执行**

在调用call()和apply()可以将一个**对象指定为第一个参数**，此时这个对象将会成为函数执行时的**this**

如图，正常调用函数时，函数内的this都会指定为**Window**，但是将某个对象作为参数后，**this就变成了这个对象**。用这种方法可以修改函数内this的指向

```
function fun(){
    console.log(this);
}
var obj={};
|fun();
fun.apply(obj);
```

new\_file.html

▶ Window {postMessage: f, blur: f, focus: f, close: f, frames:  
▶ {}

例子2：

```
var obj={
    name:"obj",
    sayname:function(){
        console.log(this.name);
    }
};
var obj2={
    name:"obj2"
};
obj.sayname();
obj.sayname.apply(obj2);
```

new\_file.html:14

obj  
obj2

apply和call不同点是：

二者可以在对象的这个参数后面再加上函数需要的实参，call将参数直接放到对象之后，apply需要将实参封装到一个数组汇总统一传递

如图：

```

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

var obj={
    name:"obj",
    sayname:function(){
        console.log(this.name);
    }
};

fun.apply(obj,[2,3]);
fun.call(obj,2,3);

```

a=2	<a href="#">new_file.html</a>
b=3	<a href="#">new_file.html</a>
a=2	<a href="#">new_file.html</a>
b=3	<a href="#">new_file.html</a>

## arguments

在调用函数时，浏览器每次都会传递进两个**隐藏的参数**：

- 1.函数的上下文对象this
- 2.封装实参的对象 arguments

**arguments**是一个类数组对象，它也可以通过索引来操作数据，也可以获取长度，在调用函数时，**我们所传的实参都会在 arguments中保存**

arguments.length可以用来获取实参的长度

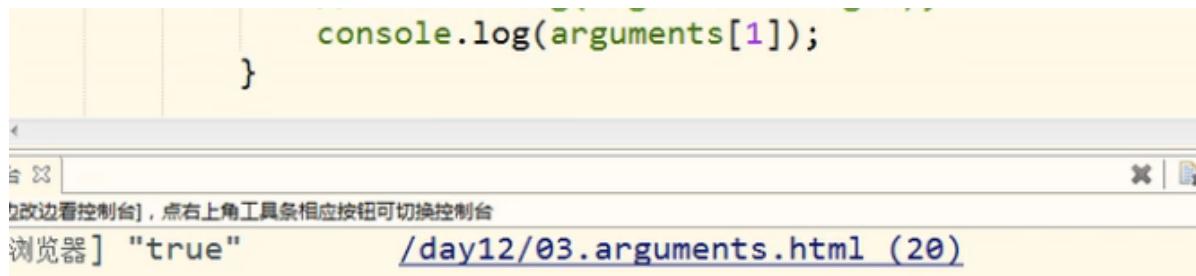
```
function fun(){
    //console.log(arguments instanceof Array);
    //console.log(Array.isArray(arguments));
    console.log(arguments.length);
}

fun("hello",true);

</script>
</head>
```

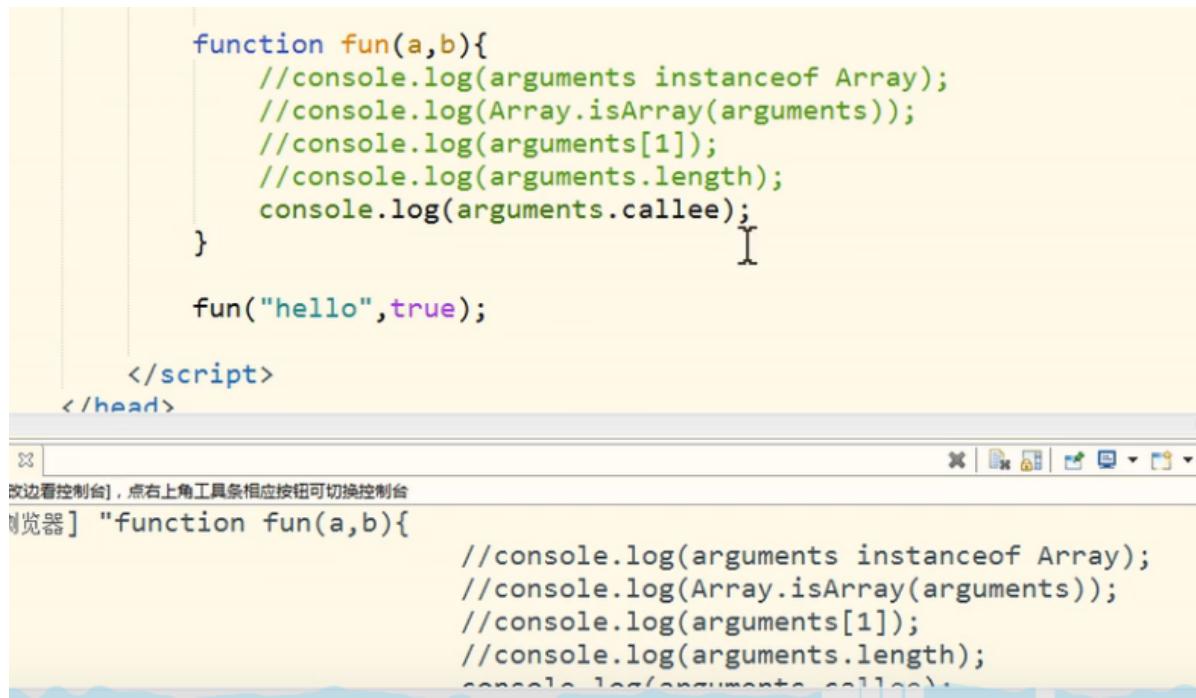


根据arguments特性，即使不定义形参我们也可以通过arguments来使用实参



arguments里边有一个属性叫做**callee**

这个属性对应一个函数对象，就是当前正在指向的函数的对象的内容



# Date对象

使用Date对象来表示一个时间

## 创建一个默认的时间对象

如果直接使用**构造函数**创建一个Date对象，则会封装为当前代码执行的时间

```
//创建一个Date对象
var d = new Date();

console.log(d);

</script>
```

控制台]，点右上角工具条相应按钮可切换控制台  
[控制台] "Fri Dec 02 2016 11:04:03 GMT+0800 (中国标准时间)" /day12/0

## 创建一个指定的时间对象

需要在构造函数中传递一个表示时间的字符串作为参数

日期的格式：月份/日/年 时:分:秒

```
//创建一个指定的时间对象
//需要在构造函数中传递一个表示时间的字符串作为参数
//日期的格式 月份/日/年 时:分:秒
var d2 = new Date("12/03/2011 11:10:30");

console.log(d2);

</script>
```

控制台]，点右上角工具条相应按钮可切换控制台  
[控制台] "Sat Dec 03 2016 11:10:30 GMT+0800 (中国标准时间)" /day12/0  
[控制台] "Sat Dec 03 2011 11:10:30 GMT+0800 (中国标准时间)" /day12/0

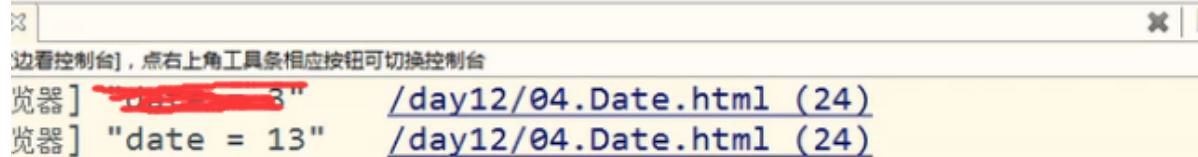
## Date的一些方法

getDate获取当前日期是当月的第几日

```
var d2 = new Date("12/13/2011 11:10:30");

/*
 * getDate()
 * - 获取当前日期对象是几日
 */
var date = d2.getDate();

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

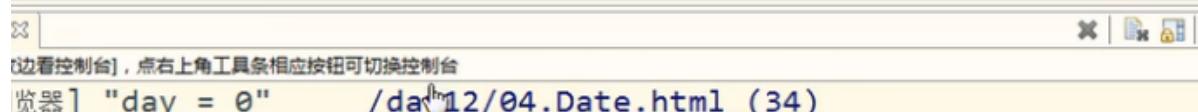


getDate获取当前日期是周几，返回0-6，周日-周六

```
var d2 = new Date("12/18/2011 11:10:30");

/*
 * getDate()
 * - 获取当前日期对象是几日
 */
var date = d2.getDate();
/*
 * getDay()
 * - 获取当前日期对象时周几
 */
var day = d2.getDay();

//console.log("date = "+date);
console.log("day = "+day);
```



getMonth获取当前的月份，返回0-11，代表1-12月

getFullYear获取当前日期对象的年份

**getTime**获取当前时间对象的**时间戳**：指的是从格林威治标准时间的1970年1月1日，0时0分0秒到当前日期所花费的**毫秒数**

计算机在底层保存的都是时间戳

**now()方法** 获取当前的时间戳

利用时间戳来测试代码的执行的性能（之前用过一个**console.time** 和 **console.timeEnd** 来计算）

# Math

Math和其他的对象不同，它不是一个构造函数

它属于一个工具类，不用创建对象，直接用就行了。它里边封装了数学运算相关的属性（一些常量）和方法。

例子：

The screenshot shows a browser developer tools console window. At the top, there is a code editor with two snippets of JavaScript. The first snippet is a comment explaining the `abs()` method and a call to it:

```
/*
 * abs() 可以用来计算一个数的绝对值
 */
console.log(Math.abs(-1));
```

The second snippet is another comment explaining the `ceil()` and `floor()` methods, followed by their respective calls:

```
* Math.ceil()
* - 可以对一个数进行向上取整，小数位只要有值就自动进1
* Math.floor()
* - 可以对一个数进行向下取整
*/
//console.log(Math.ceil(1.1));
console.log(Math.floor(1.99));
```

Below the code editor, the console output area shows the results of the executed commands:

```
[转到控制台]，点右上角工具条相应按钮可切换控制台
[转到控制台] "1" /day12/05.Math.html (20)
[转到控制台] "1" /day12/05.Math.html (20)
```

At the bottom of the console output, there is a note: "[转到控制台]，点右上角工具条相应按钮可切换控制台" and the URL "[转到控制台] "1" /day12/05.Math.html (29)".

```
* Math.random()
*   - 可以用来生成一个0-1之间的随机数
*   - 生成一个0-10的随机数
*   - 生成一个0-x之间的随机数
*       Math.round(Math.random()*x)
*/
for(var i=0 ; i<100 ; i++){
    //console.log(Math.round(Math.random()*10));
    console.log(Math.round(Math.random()*20));
```

看控制台，点右上角工具条相应按钮可切换控制台

```
器] "11" /day12/05.Math.html (43)
器] "8" /day12/05.Math.html (43)
器] "8" /day12/05.Math.html (43)
器] "6" /day12/05.Math.html (43)
```

```
*   - 生成一个1-10
*/
for(var i=0 ; i<100 ; i++){
    //console.log(Math.round(Math.random()*10));
    //console.log(Math.round(Math.random()*20));

    console.log(Math.round(Math.random()*9)+1);
}
```

看控制台，点右上角工具条相应按钮可切换控制台

```
器] "7" /day12/05.Math.html (47)
器] "6" /day12/05.Math.html (47)
器] "7" /day12/05.Math.html (47)
器] "10" /day12/05.Math.html (47)
器] "6" /day12/05.Math.html (47)
```

```
*   - 生成一个x-y之间的随机数
*       Math.round(Math.random()*(y-x)+x)
*/
```

## 包装类

```
* 基本数据类型
* String Number Boolean Null Undefined
* 引用数据类型
* Object
*
* 在JS中为我们提供了三个包装类，通过这三个包装类可以将基本数据类型的数据转换为对象
* String() 
  - 可以将基本数据类型字符串转换为String对象
* Number()
  - 可以将基本数据类型的数字转换为Number对象
* Boolean()
  - 可以将基本数据类型的布尔值转换为Boolean对象
```

例子：

The screenshot shows three examples of creating objects from primitive values using the constructor functions.

Example 1 (Number constructor):

```
// 创建一个Number类型的对象
//num = 3;
var num = new Number(3);
console.log(typeof num);
```

Output in the console:

```
object
```

Example 2 (String constructor):

```
var str = new String("hello");
```

Example 3 (Boolean constructor):

```
var bool = new Boolean(true);
```

Output in the console (for both examples):

```
object
```

但是在开发中我们绝对不会像上面这么用（就比如创建一个bool对象，他类型为object，用它来做判断，首先进行类型转换，object转换为布尔值true，就没有意义了）

我们平时再用`.toString()`等等这些方法时，直接用在基本数据类型上面，但是基本数据类型本来不能调用方法和属性

对一些基本数据类型的值去调用属性和方法时，浏览器会临时使用**包装类**将其转换为**对象**，然后在调用对象的属性和方法。调用完以后，自动将其转换为**基本数据类型**（所以像str.hello="你好"这种向字符串中添加属性的操作**并不会报错**，但是没有作用）

## string对象的方法

字符串在**底层**是以**字符数组**的形式保存的

```
/*
 * 在底层字符串是以字符数组的形式保存的
 * ["H", "e", "l"]
 */
```

也能使用数组的一些方法，也能使用索引

```
3     var str = "Hello Atguigu";
4
5     /*
6      * 在底层字符串是以字符数组的形式保存的
7      * ["H", "e", "l"]
8      */
9
10    console.log(str.length);
```

控制台 [边改边看控制台]，点右上角工具条相应按钮可切换控制台  
浏览器 ] "13" /day12/07.字符串的相关方法.html (15)

## charAt()方法

返回字符串指定位置字符，当然也可以直接用中括号

```
str = "Hello Atguigu";  
  
var result = str.charAt(0);  
  
console.log(result);  
  
</script>  
改边看控制台]，点右上角工具条相应按钮可切换控制台  
浏览器] "H" /day12/07.字符串的相关方法.html (30)  
  
str = "Hello Atguigu";  
  
var result = str[6];  
  
console.log(result);  
  
, 点右上角工具条相应按钮可切换控制台  
'A" /day12/07.字符串的相关方法.html (30)
```

## charCodeAt()方法

返回字符串指定位置字符的Unicode编码

```
str = "Hello Atguigu";  
  
var result = str.charCodeAt(6);  
  
/*  
 * charCodeAt()  
 */  
  
result = str.charCodeAt(0);  
  
console.log(result);  
  
改边看控制台]，点右上角工具条相应按钮可切换控制台  
浏览器] "72" /day12/07.字符串的相关方法.html (37)
```

## fromCharCode()方法

String.fromCharCode (编码) 根据Unicode字符编码去获取字符

这的String是内建对象

```
* String.fromCharCode()
* - 可以根据字符编码去获取字符
*/
result = String.fromCharCode(74);

console.log(result);
```

改边看控制台，点右上角工具条相应按钮可切换控制台

浏览器] "20013" /day12/07.字符串的相关方法.html (37)  
浏览器] "H" /day12/07.字符串的相关方法.html (44)  
浏览器] "I" /day12/07.字符串的相关方法.html (44)  
浏览器] "J" /day12/07.字符串的相关方法.html (44)

## concat()

用来连接两个或者多个字符串，作用和+号一样

```
result = str.concat("你好", "再见");
```

## indexOf()方法

该方法可以检索一个字符串中是否含有指定内容

如果字符串中含有该内容，则会返回其第一次出现的索引

如果没有找到指定的内容，则返回-1

可以指定第二个参数，指定开始查找的位置

```
str = "hello atguigu";
result = str.indexOf("e");
console.log(result);
```

看控制台，点右上角工具条相应按钮可切换控制台  
[控制台] "0" /day12/07.字符串的相关方法.html (60)  
[控制台] "1" /day12/07.字符串的相关方法.html (60)

```
str = "hello hatguigu";
result = str.indexOf("h", 1);
console.log(result);

</script>
```

看控制台，点右上角工具条相应按钮可切换控制台  
[控制台] "6" /day12/07.字符串的相关方法.html (62)  
[控制台] "0" /day12/07.字符串的相关方法.html (62)

## lastIndexOf()方法

该方法的用法和 indexOf () 一样，也能指定查找位置

不同的是indexOf是从前往后找，

而lastIndexOf是从后往前找

```
str = "hello hatguigu";
result = str.indexOf("h", 1);
result = str.lastIndexOf("h");
console.log(result);
```

控制台，点右上角工具条相应按钮可切换控制台  
[控制台] "6" /day12/07.字符串的相关方法.html (67)

## slice()方法

可以从字符串中截取指定的内容，不会影响原字符串，而是将截取到内容返回(数组的方法中也有同样的方法)

参数：

第一个，开始位置的索引（**包括开始位置**）

第二个，结束位置的索引（**不包括结束位置**）省略第二个参数，则表明会截取后面所有的

传递负数的话标明从后面开始计算

```
str = "abcdefghijklm";
result = str.slice(0, 2);
console.log(result);
```

看控制台]，点右上角工具条相应按钮可切换控制台  
器] "ab" /day12/07.字符串的相关方法.html (80)

## substring()方法

基本上跟slice一样，不同的是这个方法**不能接受负值**作为参数，如果传递了一个负值，则默认使用0

而且他还自动调整参数的位置，**如果第二个参数小于第一个，则自动交换**

## substr()方法

```
* substr()  
* - 用来截取字符串  
* - 参数:  
*   1. 截取开始位置的索引  
*   2. 截取的长度  
*/
```

```
str = "abcdefg";  
      ^ ^  
result = str.substr(3,2);  
  
console.log(result);
```

台

[力改边看控制台]，点右上角工具条相应按钮可切换控制台

浏览器] "de" /day12/07.字符串的相关方法.html (108)

## split()方法

可以将一个字符串拆分成一个数组

需要一个字符串作为参数，将会根据该字符串去拆分数组

```
str = "abc,bcd,efg,hij";  
      +  
result = str.split(",");  
  
console.log(result);
```

```
</script>  
</head>  
<body>  
</body>
```

[力改边看控制台]，点右上角工具条相应按钮可切换控制台

浏览器] "abc,bcd,efg,hij" /day12/07.字符串的相关方法.html (108)

# 正则表达式简介

创建正则表达式对象

构造函数创建语法：

```
* 语法:  
* var 变量 = new RegExp("正则表达式", "匹配模式");  
*/
```

字面量创建语法：

```
* 使用字面量来创建正则表达式  
* 语法: var 变量 = /正则表达式/匹配模式
```

以下两种写法等同

```
//var reg = new RegExp("a","i");  
reg = /a/i;
```

但是构造函数方法更加灵活，因为参数可以使用变量

**两种匹配模式**: i: 忽略大小写模式

g: 全局匹配模式

**test()方法**

此方法是**正则表达式对象的方法**

使用这个方法可以用来检查一个字符串是否符合正则表达式的规则，如果符合则返回true，否则返回false

```
var reg = new RegExp("a");  
  
var str = "a";  
  
/*  
 * 正则表达式的方法:  
 * test()  
 * - 使用这个方法可以用来检查一个字符串是否符合正则表达式的规则,  
 *   如果符合则返回true, 否则返回false  
 */  
var result = reg.test(str);  
console.log(result);
```

看控制台]，点右上角工具条相应按钮可切换控制台

器] "true"      [/day12/08.正则表达式.html \(39\)](#)

## 支持正则表达式的string对象

split()方法：将字符串拆分成一个数组，可以利用正则表达式

```
var str = "1a2b3c4d5e6f7";
/*
 * split()
 * - 可以将一个字符串拆分为一个数组
 * - 方法中可以传递一个正则表达式作为参数，这样方法将会根据正则表达式去拆分字符串
 */

/*
 * 根据任意字母来将字符串拆分
 */
var result = str.split(/[A-z]/); []

console.log(result);

</script>
```

控制台]，点右上角工具条相应按钮可切换控制台  
器] "1,2,3,4,5,6,7" /day12/10.字符串和正则相关的方法.html (20)

search()方法：搜索字符串中是否有指定内容，可以接受正则表达式作为索引，返回第一次出现的索引

```
21      * search()
22      * - 可以搜索字符串中是否含有指定内容
23      * - 如果搜索到指定内容，则会返回第一次出现的索引，如果没有搜索到返回 -1
24      * - 它可以接受一个正则表达式作为参数，然后会根据正则表达式去检索字符串
25      */
26      str = "hello abc hello aec afc";
27      /*
28      * 搜索字符串中是否含有abc 或 aec 或 afc
29      */
30      result = str.search(/a[bc]efc/);
31
32      console.log(result);
33
34      </script>
```

控制台 [前是 [边改边看控制台]，点右上角工具条相应按钮可切换控制台  
Web浏览器] "6" /day12/10.字符串和正则相关的方法.html (32)

match()方法：可以根据正则表达式，从一个字符串中将符合条件的内容提取出来，会将匹配到的内容封装到一个数组中返回，即使只查询到一个结果

默认情况下match只会找到第一个符合条件的内容然后就会停止检索，可以设置正则表达式为全局匹配模式，这样就会匹配到所有内容

可以为一个正则表达式设置多个匹配模式，且顺序无所谓

```
str = "1a2b3c4d5e6f7A8B9C";  
result = str.match(/[a-z]/ig);  
console.log(result);
```

边看控制台，点右上角工具条相应按钮可切换控制台  
览器] "a,b,c,d,e,f,A,B,C" /day12/10.字符串和正则相关的方法.html (42)  
览器] "a,b,c,d,e,f,A,B,C" /day12/10.字符串和正则相关的方法.html (42)

(i g这两个先后顺序不要求 )

**replace()**方法：可以将字符串中指定内容替换为新的内容，**默认只会替换第一个**，可以设置正则表达式为**全局匹配模式**，匹配到所有内容（单个字符一个个的相匹配）

两个参数：

1.被替换的内容，可以接受一个正则表达式作为参数

2.新的内容

```
//result = str.replace(/[a-z]/gi , "@_@");  
result = str.replace(/[a-z]/gi , "");  
console.log(result);
```

制台，点右上角工具条相应按钮可切换控制台  
| "123456789" /day12/10.字符串和正则相关的方法.html (59)

## 正则表达式

例子：

```
* 手机号的规则：  
* 1 3 567890123 (11位)  
*  
* 1. 以1开头  
* 2. 第二位3-9任意数字  
* 3. 三位以后任意数字9个  
*  
* ^1 [3-9] [0-9]{9}$  
*  
*/
```

```
var phoneStr = "13067890123";  
|  
var phoneReg = /^1[3-9][0-9]{9}$/;  
  
console.log(phoneReg.test(phoneStr));
```

## [ ]的使用

[ ]是定义匹配的字符范围，单个字符进行匹配

比如[a-zA-Z0-9]表示相应位置的字符要匹配英文字符和数字。

## ^的使用

^在[]中表示除了，^在[]之外表示以什么开头

## \的使用

```
/*
 * 检查一个字符串中是否含有 .
 * . 表示任意字符
 * 在正则表达式中使用\作为转义字符
 * \. 来表示 .
 * \\ 表示\
 *
 * 注意：使用构造函数时，由于它的参数是一个字符串，而\是字符串中转义字符，
 * 如果要使用\则需要使用\\来代替
 */
```

字面量创建正则表达式，使用转义符\，直接用就可以了，因为字面量这种方法不带双引号

使用构造函数创建正则表达式，使用转义符\，**需要用\\来代替**，因为这种方法有**双引号**，\也是字符串中的转义符

## 非打印字符

非打印字符也可以是正则表达式的组成部分。下表列出了表示非打印字符的转义序列：

字符	描述
\cx	匹配由x指明的控制字符。例如，\cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	<b>匹配任何空白字符，包括空格、制表符、换页符等等。</b> 等价于 [ \f\n\r\t\v]。注意 Unicode 正则表达式会匹配全角空格符。
\S	<b>匹配任何非空白字符。</b> 等价于 [^\f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。

**\s的例子：**

去掉所有的空格：

```
var str = "      hello      ";
//去除掉字符串中的空格
//去除空格就是使用""来替换空格
console.log(str);

str = str.replace(/\s/g , "");

console.log(str);
```

控制台输出：

```
[边改边看控制台]，点右上角工具条相应按钮可切换控制台
浏览器] "      hello      " /day13/03.正则表达式.html (68)
浏览器] "hello" /day13/03.正则表达式.html (72)
```

去掉开头结尾的空格：

```
//去除掉字符串中的前后的空格
//去除空格就是使用""来替换空格
console.log(str);

//str = str.replace(/\s/g , "");

//去除开头的空格
//str = str.replace(/^\s*/ , "");
//去除结尾的空格
//str = str.replace(/\s*$/ , "");
// /\s*/ /\s*$/ 
str = str.replace(/^\s*|\s*$/g,"");

console.log(str);
```

控制台输出：

```
[边改边看控制台]，点右上角工具条相应按钮可切换控制台
浏览器] "      he      llo      " /day13/03.正则表达式.html (80)
浏览器] "he      llo" /day13/03.正则表达式.html (80)
```

|代表或，只用|就可能只去掉开头的空格，后面加上g，就全局模式匹配，前后空格都去掉了

## 特殊字符

所谓特殊字符，就是一些有特殊含义的字符，如上面说的 runo\*b 中的 \*，简单的说就是表示任何字符串的意思。如果要查找字符串中的 \* 符号，则需要对 \* 进行转义，即在其前加一个 \：runo\*ob 匹配 runo\*ob。

许多元字符要求在试图匹配它们时特别对待。若要匹配这些特殊字符，必须首先使字符“转义”，即，将反斜杠字符\放在它们前面。下表列出了正则表达式中的特殊字符：

特别字符	描述
\$	匹配输入字符串的 <b>结尾位置</b> 。如果设置了 RegExp 对象的 Multiline 属性，则 \$ 也匹配 '\n' 或 '\r'。要匹配 \$ 字符本身，请使用 \\$。
( )	标记一个 <b>子表达式</b> 的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 ( 和 )。
*	匹配前面的子表达式零次或多次。要匹配 * 字符，请使用 \*。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。
.	匹配 <b>除换行符 \n 之外的任何单字符</b> 。要匹配 .，请使用 \.。
[ ]	标记一个中括号表达式的开始。要匹配 [，请使用 \[。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如，'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "", 而 '\"' 则匹配 ""。转义符
^	匹配输入字符串的 <b>开始位置</b> ，除非在方括号表达式中使用，当该符号在方括号表达式中使用时，表示不接受该方括号表达式中的字符集合。要匹配 ^ 字符本身，请使用 \^。
{ }	标记限定符表达式的开始。要匹配 {，请使用 \{。
	指明两项之间的一个选择。要匹配  ，请使用 \ 。

^a\$符合的条件只有一个a (a它自己即使开头又是结尾)



```
* 如果在正则表达式中同时使用^ $则要求字符串必须完全符合正则表达式
*/
reg = /^a$/;

console.log(reg.test("a"));

改边看控制台，点右上角工具条相应按钮可切换控制台
浏览器] "true"      /day13/02.正则表达式.html (54)
```

## 限定符

限定符用来指定正则表达式的一个给定组件必须要出现多少次才能满足匹配。有 \* 或 + 或 ? 或 {n} 或 {n,} 或 {n,m} 共6种。

正则表达式的限定符有：

字符	描述
*	匹配前面的子表达式零次或多次。例如，'zo*' 能匹配 "z" 以及 "zoo"。* 等价于 {0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do"、"does" 中的 "does"、"doxy" 中的 "do"。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配 n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数，其中 n <= m。最少匹配 n 次且最多匹配 m 次。例如，'o{1,3}' 将匹配 "foooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。
\w	任意字母、数字、_ (下划线)，相当于 [A-z0-9_]
\W	除了任意字母、数字、_ (下划线)，相当于 [^A-z0-9_]
\d	任意数字 [0-9]
\D	除了数字 [^0-9]

## 定位符

定位符使您能够将正则表达式固定到行首或行尾。它们还使您能够创建这样的正则表达式，这些正则表达式出现在一个单词内、在一个单词的开头或者一个单词的结尾。

定位符用来描述字符串或单词的边界，^ 和 \$ 分别指字符串的开始与结束，\b 描述单词的前或后边界，\B 表示非单词边界。

正则表达式的定位符有：

字符	描述
^	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与 \n 或 \r 之后的位置匹配。
\$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，\$ 还会与 \n 或 \r 之前的位置匹配。
\b	匹配一个单词边界，即匹配一个单词，单词前或者后不能有别的字符，必须是空格。
\B	非单词边界匹配，单词前面或者后面不能是空格，在单词的前或者后面用，相应位置不能是空格，必须连起来

```
reg = /\bchild\b/;
```

```
console.log(reg.test("hello child "));
```

**注意：**不能将限定符与定位符一起使用。由于在紧靠换行或者单词边界的前面或后面不能有一个以上位置，因此不允许诸如 `^*` 之类的表达式。

例子：

The screenshot shows a browser's developer tools console window. The code in the editor pane is:

```
/*
 * 电子邮件
 * hello .nihao      @      abc .com.cn
 *
 * 任意字母数字下划线 .任意字母数字下划线 @  任意字母数字 .任意字母（2-5位） .任意字母（2-5位）
 *
 * \w{3,} (\.\w+)* @ [A-z0-9]+ (\.[A-z]{2,5}){1,2}
 */
17 var emailReg = /^\\w{3,}(\\.\\w+)*@[A-z0-9]+(\\.[A-z]{2,5}){1,2}$/;
18
19 var email = "123abc.hello@163.com";
20
21 console.log(emailReg.test(email));
22
23
24 <script>
25 |>
```

The console output below shows the result of the regex test:

```
控制台
前是 [边改边看控制台]，点右上角工具条相应按钮可切换控制台
Web浏览器 ] "true" /day13/04.邮件的正则.html (21)
```

(此例有错误，A-z包括下划线)

## 宿主对象

**对象一共分三种：内建对象、宿主对象、自定义对象**

宿主对象：由浏览器，或者说有运行环境给我们提供的对象

## DOM (文档对象模型)

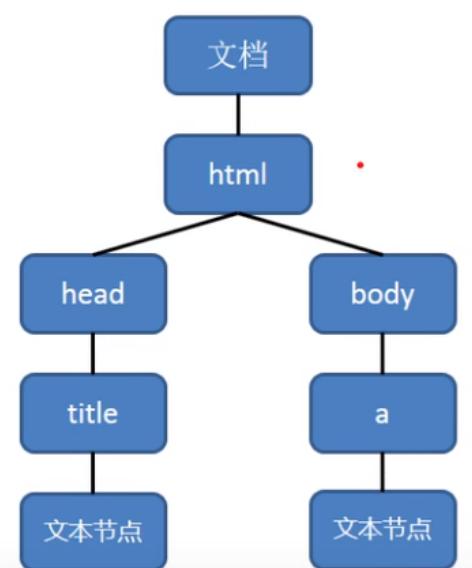
# 什么是DOM

- DOM，全称Document Object Model文档对象模型。
- JS中通过DOM来对HTML文档进行操作。只要理解了DOM就可以随心所欲的操作WEB页面。
- 文档
  - 文档表示的就是整个的HTML网页文档
- 对象
  - 对象表示将网页中的每一个部分都转换为了一个对象。
- 模型
  - 使用模型来表示对象之间的关系，这样方便我们获取对象。

## 模型

1.html

```
<html>
  <head>
    <title>网页的标题</title>
  </head>
  <body>
    <a href="1.html" >超连接</a>
  </body>
</html>
```



节点

# 节点

- 节点Node，是构成我们网页的最基本的组成部分，网页中的每一个部分都可以称为是一个节点。
- 比如：html标签、属性、文本、注释、整个文档等都是一个节点。
- 虽然都是节点，但是实际上他们的具体类型是不同的。
- 比如：标签我们称为元素节点、属性称为属性节点、文本称为文本节点、文档称为文档节点。
- 节点的类型不同，属性和方法也都不尽相同。

节点：Node——构成html文档最基本的单元

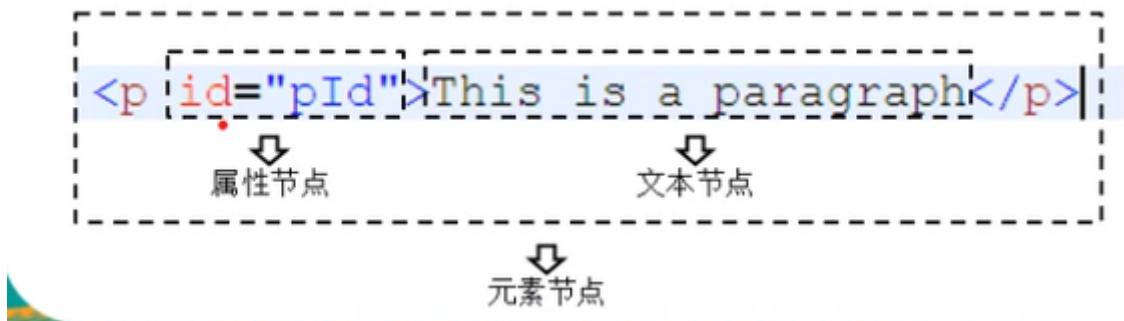
常用的节点分为四类：

**文档节点**：整个html文档

**元素节点**：HTML文档中的标签

**属性节点**：元素的属性

**文本节点**：html标签中的文本内容



## 节点的属性

每一个节点都有**nodeName**、**nodeType**、**nodeValue**三个属性

# 节点的属性

	nodeName	nodeType	nodeValue
文档节点	#document	9	null
元素节点	标签名	1	null
属性节点	属性名	2	属性值
文本节点	#text	3	★文本内容

## 文档节点对象

浏览器已经为我们提供文档节点对象这个对象是 window的属性——**document**

可以在页面中直接使用，文档节点代表的是整个网页

例子：document对象的getElementById()方法来寻找元素节点（文档节点->元素节点）

```
//获取到button对象
var btn = document.getElementById("btn");

console.log(btn);
```

浏览器] "[object HTMLButtonElement]" /day13/05.DOM.html (20)

**innerHTML属性**, btn.innerHTML代表btn元素（也就是button标签）的内容

```
<body>
  <button id="btn">我是一个按钮</button>
  <script type="text/javascript">

    /*
     * 浏览器已经为我们提供 文档节点 对象这个对象是window属性
     * 可以在页面中直接使用，文档节点代表的是整个网页
     */
    //console.log(document);

    //获取到button对象
    var btn = document.getElementById("btn");

    //修改按钮的文字
    btn.innerHTML = "I'm Button";
```

# 事件

- 事件，就是文档或浏览器窗口中发生的一些特定的交互瞬间。
- JavaScript 与 HTML 之间的交互是通过事件实现的。
- 对于 Web 应用来说，有下面这些代表性的事件：点击某个元素、将鼠标移动至某个元素上方、按下键盘上某个键，等等。

我们可以在**事件对应的属性**中设置一些js代码，

这样当事件被触发时，这些代码将会执行

js书写位置：

```
<button id="btn" onmousemove="alert('讨厌，你点我干嘛！');">我是一个按钮</button>
```

可以直接在其他的标签中当做属性写，但是这种称为结构与行为耦合，不方便维护不推荐使用

应该写在body 标签中，并且靠下面的位置

浏览器在加载一个页面时，是按照**自上向下的**顺序加载的，读取到一行就运行一行，如果将 script 标签写到页面的上边，在代码执行时，页面还没有加载

如果非要写在 body标签前面，则需要用到onload事件，**window.onload**代表等到页面加载完成之后再执行

支持该事件的 JavaScript 对象：

```
image, layer, window
```

然后把代码都写到这个事件中去，确保相应的js代码执行时，所有的DOM对象都应经加载完毕

```
* onload事件会在整个页面加载完成之后才触发
* 为window绑定一个onload事件
*/
window.onload = function(){
    alert("hello");
};
```

但是这种方法属于先加载后执行，效率不高，但用的比较常见

### 事件绑定的基本操作：

因此，将js代码写在script标签中，然后通过document.getElementById()方法进行对象的获取，然后在进行事件的绑定（已经定义好了很多事件）

事件相当于属性，属性等于的是一个函数，函数当事件被触发的时候执行。如下图

```
25 //获取按钮对象
26 var btn = document.getElementById("btn");
27
28/*
29 * 可以为按钮的对应事件绑定处理函数的形式来响应事件
30 * 这样当事件被触发时，其对应的函数将会被调用
31 */
32
33 //绑定一个单击事件
34btn.onclick = function(){
35     alert("你还点~~~");
36 };
37
```

## DOM查询

### 获取元素节点

document下获取元素节点对象的方法

# 获取元素节点

- 通过document对象调用
  1. getElementById()
    - 通过**id**属性获取一个元素节点对象
  2. getElementsByTagName()
    - 通过**标签名**获取一组元素节点对象
  3. getElementsByName()
    - 通过**name**属性获取一组元素节点对象

**getElementById()**

通过**id**属性获取一个元素节点对象

**getElementsByTagName()**

通过**标签名**获取一组元素节点对象 (比如div标签, 通过div来获取所有的div标签)

**getElementsByName()**

通过**name**属性获取一组元素节点对象

(**id**和**name**的区别: **id**唯一, 只能具体某个标签使用, 而**name**则可以重复多个标签使用, 但都属于标签的属性)

**innerHTML**属性: 通过这个属性可以获取元素内部的html代码

以上三个方法的实践:

通过**id**属性来获取某个标签

```
//为id为btn01的按钮绑定一个单击响应函数
var btn01 = document.getElementById("btn01");
btn01.onclick = function(){
    //查找#bj节点
    var bj = document.getElementById("bj");
    //打印bj
    //innerHTML 通过这个属性可以获取到元素内部的html代码
    alert(bj.innerHTML);
};
```

通过标签名来获取所有指定标签：

```
//为id为btn02的按钮绑定一个单击响应函数
var btn02 = document.getElementById("btn02");
btn02.onclick = function(){
    //查找所有li节点
    //getElementsByTagName()可以根据标签名来获取一组元素节点对象
    //这个方法会给我们返回一个类数组对象，所有查询到的元素都会封装到对象中
    //即使查询到的元素只有一个，也会封装到数组中返回
    var lis = document.getElementsByTagName("li");

    //打印lis
    //alert(lis.length);

    //变量lis
    for(var i=0 ; i<lis.length ; i++){
        alert(lis[i].innerHTML);
    }
};
```

通过name属性来获取所有指定的标签：

```
//为id为btn03的按钮绑定一个单击响应函数
var btn03 = document.getElementById("btn03");
btn03.onclick = function(){
    //查找name=gender的所有节点
    var inputs = document.getElementsByName("gender");

    //alert(inputs.length);

    for(var i=0 ; i<inputs.length ; i++){
        /*
         * innerHTML用于获取元素内部的HTML代码的
         * 对于自结束标签，这个属性没有意义
         */
        //alert(inputs[i].innerHTML);
        /*
         * 如果需要读取元素节点属性，
         * 直接使用元素.属性名
         * 例子：元素.id 元素.name 元素.value
         * 注意：class属性不能采用这种方式，
         *       读取class属性时需要使用元素.className
         */
        alert(inputs[i].className);
    }
};
```

例子：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title></title>
        <style type="text/css">
```

```
*{
    margin:0;
    padding:0;
}
#outer{
    width: 500px;
    margin: 50px auto;
    padding: 10px;
    background-color: yellowgreen;
    text-align: center;
}
</style>

<script type="text/javascript">
window.onload =function(){
    // 点击按钮切换图片
    var prev =document.getElementById("prev");
    var next =document.getElementById("next");
    var imgArr=["1.jpg","2.jpg","3.jpg","4.jpg"];
    var img=document.getElementsByTagName("img")[0];
    var index = 0;
    prev.onclick = function(){
        index--;
        if(index<0){
            index = 0;
        }
        img.src=imgArr[index];
    };
    next.onclick = function(){
        index++;
        if(index>imgArr.length-1){
            index =imgArr.length-1;
        }
        img.src=imgArr[index];
    };
}
</script>

</head>
<body>
    <div id="outer">
        
        <button id="prev">上一张</button>
        <button id="next">下一张</button>
    </div>

    <script type="text/javascript" >
    </script>
</body>
</html>
```

### ps:一点小问题

某个例子：在for循环中绑定一个单击响应函数，这是for循环中的i使用就出现了问题

```
//获取所有额超链接
var allA = document.getElementsByTagName("a");

//为每个超链接都绑定一个单击响应函数
for(var i=0 ; i < allA.length ; i++){
    allA[i].onclick = function(){
        alert(allA[i]);
        return false;
    };
}
```

for循环在页面加载结束一开始执行完了，所有的响应函数也绑定完了，之后再点击超链接，再次执行allA[i]，这里面的i仍然是变量，循环时并没有被具体数字替换，这时候i已经是循环结束后的那个值，且一直是那个值

用this代替allA[i]就没有问题了

### 获取元素节点的子节点

## 获取元素节点的子节点

- 通过具体的元素节点调用
1. getElementsByTagName()
    - 方法，返回当前节点的指定标签名后代节点
  2. childNodes
    - 属性，表示当前节点的所有子节点
  3. firstChild
    - 属性，表示当前节点的第一个子节点
  4. lastChild
    - 属性，表示当前节点的最后一个子节点

getElementsByName()方法在**文档对象** (document) 和**元素对象** (标签) 都有

通过标签名来获取当前标签后代节点 (后代标签)

getElementsByName()方法

```
//为id为btn04的按钮绑定一个单击响应函数
var btn04 = document.getElementById("btn04");
btn04.onclick = function(){

    //获取id为city的元素
    var city = document.getElementById("city");

    //查找#city下所有li节点
    var lis = city.getElementsByTagName("li");

    for(var i=0 ; i<lis.length ; i++){
        alert(lis[i].innerHTML);
    }
}
```

返回当前节点的所有子节点

childNodes属性

```
//为id为btn05的按钮绑定一个单击响应函数
var btn05 = document.getElementById("btn05");
btn05.onclick = function(){
    //获取id为city的节点
    var city = document.getElementById("city");
    //返回#city的所有子节点
    /*
     * childNodes属性会获取包括文本节点在内的所有节点
     * 根据DOM标签标签间空白也会当成文本节点
     */
    var cns = city.childNodes;

    alert(cns.length);

    /*for(var i=0 ; i<cns.length ; i++){
        alert(cns[i]);
    }*/
};
```

所有子节点也包括文本节点，一处标签间的空白也会作为一个文本节点（但是在IE8中不会将空白当成文本节点）

返回当前节点的所有子元素

children属性

```
* children属性可以获取当前元素的所有子元素
```

```
*/
```

```
var cns2 = city.children;  
alert(cns2.length);
```

此时返回的内容中就不包括了空白文本节点

只会返回标签

返回当前节点的第一个子节点

firstChild

```
//为id为btn06的按钮绑定一个单击响应函数  
var btn06 = document.getElementById("btn06");  
btn06.onclick = function(){  
    //获取id为phone的元素  
    var phone = document.getElementById("phone");  
    //返回#phone的第一个子节点  
    //phone.childNodes[0];  
    //firstChild可以获取到当前元素的第一个子节点（包括空白文本节点）  
    var fir = phone.firstChild;  
  
    alert(fir);  
};
```

返回节点也包括文本节点

返回当前节点的第一个子元素

firstElementChild

```
//firstElementChild获取当前元素的第一个子元素  
fir = phone.firstElementChild;
```

不包括空白文本等等文本节点（但是不支持IE8及以下的浏览器）

获取元素节点的父亲节点和兄弟节点

PS: 前后这几节教程，每次都要获取事件，绑定参数，为方便，直接定义一个函数来简化步骤

```
* 定义一个函数，专门用来为指定元素绑定单击响应函数
* 参数：
*   idStr 要绑定单击响应函数的对象的id属性值
*   fun 事件的回调函数，当单击元素时，该函数将会被触发
*/
function myClick(idStr , fun){
    var btn = document.getElementById(idStr);
    btn.onclick = fun;
}
```

innerText属性，该属性可以获取元素内部的文本内容，他和innerHTML类似，不同的是他会自动地将HTML标签去除，只保留文本内容。

## 获取父节点和兄弟节点

- 通过具体的节点调用

### 1. parentNode

- 属性，表示当前节点的父节点

### 2. previousSibling

- 属性，表示当前节点的前一个兄弟节点

### 3. nextSibling

- 属性，表示当前节点的后一个兄弟节点

返回当前节点的父节点

parentNode属性

```
//为id为btn07的按钮绑定一个单击响应函数
myClick("btn07",function(){

    //获取id为bj的节点
    var bj = document.getElementById("bj");

    //返回#bj的父节点
    var pn = bj.parentNode;
```

某个元素的父节点是唯一的，就不可能是空白之类的文本节点了

返回当前节点的前一个兄弟节点

previousSibling属性

```
//为id为btn08的按钮绑定一个单击响应函数
myClick("btn08",function(){

    //获取id为android的元素
    var and = document.getElementById("android");

    //返回#android的前一个兄弟节点
    var ps = and.previousSibling;】

    alert(ps);

});
```

也会获取空白、换行之类的文本节点

返回当前节点的前一个兄弟元素

previousElementSibling

```
var pe = and.previousElementSibling;
```

不会返回文本节点，返回的只能是元素 (IE8及以下版本不支持)

读取当前节点的value属性值

就相当于某个对象的属性，文本框里面写的什么独到的就是什么

```
//读取#username的value属性值
myClick("btn09",function(){
    //获取id为username的元素
    var um = document.getElementById("username");
    //读取um的value属性值
    //文本框的value属性值，就是文本框中填写的内容】
    alert(um.value);
});
```

nodeValue属性

获取某个节点的文本内容，大部分情况下做到的效果跟innerText一样，但是比innerText麻烦

```
//获取bj中的文本节点
/*var fc = bj.firstChild;
alert(fc.nodeValue);*/

alert(bj.firstChild.nodeValue);
```

## DOM查询的剩余方法

①

要查询body标签的话，可以用getElementsByName()方法

```
//获取body标签  
var body = document.getElementsByTagName("body")[0];
```

document中也有一个属性body，专门用来引用body标签的，返回的就不是类数组了，返回的就是直接是body标签

```
/*  
 * 在document中有一个属性body，它保存的是body的引用  
 */  
var body = document.body;
```

②

document中有一个属性documentElement，用来获取<html>根标签

```
/*  
 * document.documentElement保存的是html根标签  
 */  
var html = document.documentElement;
```

③

document.all代表页面中所有的元素（即所有的标签）

```
/*  
 * document.all代表页面中所有的元素  
 */  
var all = document.all;  
  
//console.log(all.length);  
  
for(var i=0 ; i<all.length ; i++){  
    console.log(all[i]);  
}
```

④

`document.getElementsByTagName("*");`这样代表的也是页面中的所有元素，作用跟`document.all`一样

```
all = document.getElementsByTagName("*");
```

⑤

`document.getElementsByClassName()`方法用来根据`class`属性值来获取一组元素节点对象，但是IE8及以下不支持

⑥

```
document.querySelector()
```

需要一个css选择器的字符串作为参数，可以根据一个CSS选择器来查询一个元素节点对象，使用该方法总会返回唯一的一个元素，如果满足条件的元素有多个，那么它只会返回第一个

虽然IE8中没有`getElementsByClassName()`但是可以使用`querySelector()`代替

```
/*
 * document.querySelector()
 * - 需要一个选择器的字符串作为参数，可以根据一个CSS选择器来查询一个元素节点对象
 * - 虽然IE8中没有getElementsByClassName()但是可以使用querySelector()代替
 * - 使用该方法总会返回唯一的一个元素，如果满足条件的元素有多个，那么它只会返回第一个
 */
var div = document.querySelector(".box1 div");

var box1 = document.querySelector(".box1")
```

⑦

```
document.querySelectorAll()
```

该方法和`querySelector()`用法类似，不同的是它会将符合条件的元素封装到一个数组中返回  
即使符合条件的元素只有一个，它也会返回数组

```
/*
 * document.querySelectorAll()
 * - 该方法和querySelector()用法类似，不同的是它会将符合条件的元素封装到一个数组中返回
 * - 即使符合条件的元素只有一个，它也会返回数组
 */
box1 = document.querySelectorAll(".box1");
box1 = document.querySelectorAll("#box2");
console.log(box1);
```

ps：例子中的`.box1`查找的是`class`名

`#box2`查找的是`id`名

## DOM增删改

创建元素节点（即创建标签）

`document.createElement()`方法

```
* document.createElement()  
* 可以用于创建一个元素节点对象,  
* 它需要一个标签名作为参数, 将会根据该标签名创建元素节点对象,  
* 并将创建好的对象作为返回值返回  
*/  
var li = document.createElement("li");
```

### 创建文本节点

document.createTextNode()方法

```
* document.createTextNode()  
* 可以用来创建一个文本节点对象  
* 需要一个文本内容作为参数, 将会根据该内容创建文本节点, 并将新的节点返回  
*/  
var gzText = document.createTextNode("广州");
```

### 把子节点添加到指定节点中

父节点.appendChild()方法

```
* appendChild()  
* - 向一个父节点中添加一个新的子节点  
* - 用法: 父节点.appendChild(子节点);  
*/  
li.appendChild(gzText);
```

### 在指定的子节点前插入新的子节点

父节点.insertBefore(新节点,旧节点)

```
* insertBefore()  
* - 可以在指定的子节点前插入新的子节点  
* - 语法:  
*      父节点.insertBefore(新节点,旧节点);  
*/  
city.insertBefore(li , bj);
```

### 使用指定节点替换已有的节点

父节点.replaceChild(新节点,旧节点)

```
/*
 * replaceChild()
 * - 可以使用指定的子节点替换已有的子节点
 * - 语法: 父节点.replaceChild(新节点,旧节点);
 */
city.replaceChild(li , bj);
```

### 删除指定节点的子节点

父节点.removeChild(子节点)

```
/*
 * removeChild()
 * - 可以删除一个子节点
 * - 语法: 父节点.removeChild(子节点);
 */
city.removeChild(bj);
```

但有时候不知道父节点是谁，所以常用操作：

子节点.parentNode.removeChild(子节点)

```
/*
 * removeChild()
 * - 可以删除一个子节点
 * - 语法: 父节点.removeChild(子节点);
 *
 *         |子节点.parentNode.removeChild(子节点);
 */
//city.removeChild(bj);

bj.parentNode.removeChild(bj);
});
```

### 读取或设置元素内的HTML代码

节点.innerHTML

```
//读取#city内的HTML代码
myClick("btn05",function(){
    //获取city
    var city = document.getElementById("city");

    alert(city.innerHTML);
});

//设置#bj内的HTML代码
myClick("btn06" , function(){
    //获取bj
    var bj = document.getElementById("bj");
    bj.innerHTML = "昌平";
```

用innerHTML也能实现指定节点添加到已有节点、把子节点添加到指定节点中去等操作

```
//向city中添加广州
var city = document.getElementById("city");

city.innerHTML += "<li>广州</li>";
```

但这种方法动静太大，修改了父节点所有的子节点，如果之前的子节点中绑定了事件，修改后就没了

一般使用折中的方式：

按原来的方法，有以下操作：

创建一个元素节点

创建一个文本节点

将文本节点添加到创建的元素节点中去

将这个创建的元素节点再添加到指定的父节点中去

折中后的操作如下：

```
//创建一个li
var li = document.createElement("li");
//向li中设置文本
li.innerHTML = "广州";
//将li添加到city中
city.appendChild(li);
```

## 使用DOM操作css

修改或者读取元素的内联样式

语法：

元素.style.样式名=样式值

注意：如果css的样式名中含有“-”（减号），这样写是不合法的，需要将样式名修改为驼峰命名法

如：background-color改为backgroundColor

```
box1.style.width = "300px";
box1.style.height = "300px";
box1.style.backgroundColor = "yellow";
```

```
//点击按钮2以后，读取元素的样式
var btn02 = document.getElementById("btn02");
btn02.onclick = function(){
    //读取box1的样式
    /*
     * 语法：元素.style.样式名
     */
    alert(box1.style.width);
};
```

我们通过style属性设置的样式都是内联样式（直接在相应标签中修改的属性），修改和读取都是内联样式，无法读取修改样式表中的样式。

他有较高的优先级，所以通过js修改的样式往往会立即显示

```
<!DOCTYPE html>
<html>
    <head>
        <body>
            <button id="btn01">点我一下</button>
            <br>
            <br>
            <div id="box" style="width: 300px; height: 300px; background-color: yellow;"></div>
        </body>
    </html>
```

但是如果在样式中写了! important，则此时样式会有最高的优先级，

即使通过JS也不能覆盖该样式，此时将会导致JS修改样式失效，所以尽量不要为样式添加!important

读取元素当前显示的样式

currentStyle

缺点是只有IE支持

语法：

元素.currentStyle样式名

内联表或者样式表中都可以读取，谁生效获取谁

如果当前元素没有设置样式值，则获取他的默认值

```
/*
 * 获取元素的当前显示的样式
 * 语法：元素.currentStyle.样式名
 * 它可以用来读取当前元素正在显示的样式
 * 如果当前元素没有设置该样式，则获取它
 */
//alert(box1.currentStyle.width);
alert(box1.currentStyle.backgroundColor);
```

### getComputerdStyle()方法

IE8以上支持

getComputedStyle()这个方法来获取元素当前的样式，效果和currentStyle一样

但①返回的样式参数的格式不太一样，②并且获取的样式如果没有设置，不会获取默认值，而是真实的值，比如，没有设置width，它不会获取到auto，而是一个长度

这个方法是window的方法，可以直接使用

需要两个参数：

第一个要获取样式的元素

第二个可以传递一个伪元素，一般都传null

该方法会返回一个对象，对象中封装了当前元素对应的样式，读取需要像读取属性一样读取

```
/*
 * 在其他浏览器中可以使用
 *      getComputedStyle()这个方法来获取元素当前的样式
 *      这个方法是window的方法，可以直接使用
 * 需要两个参数
 *      第一个：要获取样式的元素
 *      第二个：可以传递一个伪元素，一般都传null
 *
 * 该方法会返回一个对象，对象中封装了当前元素对应的样式
 */
var obj = getComputedStyle(box1,null);

alert(obj.width);
```

currentStyle和getComputerStyle()读取到的样式都只是可读的，只有style属性才能修改样式

为了能在所有的浏览器上面运行，可以自己写一个函数

定义一个函数，用来获取指定元素的当前的样式

### 参数

obj: 要获取样式的元素

name: 要获取的样式名

```
function getStyle(obj , name){  
    if(window.getComputedStyle){  
        //正常浏览器的方式，具有getComputedStyle()方法  
        return getComputedStyle(obj , null)[name];  
    }else{  
        //IE8的方式，没有getComputedStyle()方法  
        return obj.currentStyle[name];  
    }  
}
```

ps:

①name的时使用，不能在函数中直接用.name，因为.name就直接在样式里面找名字为name的样式了，根本不会替换成实参来使用，要使用[name]

②if当中必须写成window.getComputerStyle，如果只写getComputerStyle的话，在函数作用域和全局作用域中寻找getComputerStyle，没找到就直接报错了，写成window的方法的形式，返回的是未定义，判断为false

## 其他样式相关属性

### clientWidth属性

### clientHeight属性

这两个属性可以获取元素的可见宽度和高度

这些属性都是不带px的，返回都是一个数字，可以直接进行计算

会获取元素宽度和高度，包括内容区和内边距，不包括边框border，有滚动条的还要减去滚动条

这些属性都是只读的

### offsetWidth属性

### offsetHeight属性

获取元素的整个宽度和高度，包括内容区和内边距，包括边框border

### offsetParent属性

可以用来获取当前元素的定位父元素

会获取到离当前元素最近的开启了定位的祖先元素

如果所有的祖先元素都没有开启定位，则返回body

PS:position值默认static, relative相对定位, absolute绝对定位,fixed固定定位,sticky粘滞定位。  
position值不是static就开启了定位

### offsetLeft

当前元素相对于其定位父元素（即offsetParent）的水平偏移量

### offsetTop

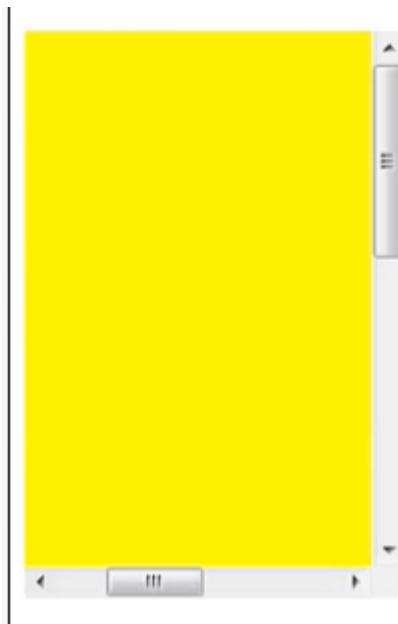
当前元素相对于其定位父元素的垂直偏移量

### scrollWidth

### scrollHeight

可以获取元素整个滚动区域的宽度和高度

有的子元素大小超过了父元素，overflow设置的为auto，会出现滚动条。



如果用clientWidth属性、clientHeight属性这两个属性读取父元素的长度和宽度的话，只会读取可见宽度，如果用scrollWidth、scrollHeight这两个属性的话，则会返回滚动宽度，**返回子元素的实际占的大小**

### scrollLeft

可以获取水平滚动条滚动的距离

### scrollTop

可以获取垂直滚动条滚动的距离

```
//当满足scrollHeight - scrollTop == clientHeight  
//说明垂直滚动条滚动到底了  
|  
//当满足scrollWidth - scrollLeft == clientWidth  
//说明水平滚动条滚动到底  
//alert(box4.scrollHeight - box4.scrollTop); // 600
```

## 事件对象

事件对象

当事件的响应函数**被触发**时，浏览器每次都会将一个**事件对象**作为实参**传递进响应函数**，平时写响应函数，都是不写参数的，传进去的参数就可以作为事件对象

在事件对象中封装了**当前事件相关的一切信息**，比如：鼠标的坐标、键盘哪个按键被按下、鼠标滚轮滚动的方向。。。等等

```
/* 事件对象  
* - 当事件的响应函数被触发时，浏览器每次都会将一个事件对象作为实参传递进响应函数  
areaDiv.onmousemove = function(e){  
    alert(e);  
}
```

事件对象的**clientX属性**可以获取鼠标指针的水平坐标

事件对象的**clientY属性**可以获取鼠标指针的垂直坐标

```
/*  
* onmousemove  
* - 该事件将会在鼠标在元素中移动时被触发  
*  
* 事件对象  
* - 当事件的响应函数被触发时，浏览器每次都会将一个事件对象作为实参传递进响应函数，  
*     在事件对象中封装了当前事件相关的一切信息，比如：鼠标的坐标 键盘哪个按键被按下 鼠标滚轮滚动的方向。。。  
areaDiv.onmousemove = function(event){  
  
    /*  
     * clientX可以获取鼠标指针的水平坐标  
     * clientY可以获取鼠标指针的垂直坐标  
    */  
    var x = event.clientX;  
    var y = event.clientY;  
  
    alert("x = "+x + " , y = "+y);  
  
    //在showMsg中显示鼠标的坐标  
};
```

在IE8中，响应函数被触发时，浏览器不会传事件对象

在IE8及以下的浏览器中，是将事件对象作为 window对象的属性保存的

用window.event.clientX便可以读取

```
var x = window.event.clientX;  
var y = window.event.clientY;
```

用下面这两种方法可以解决不同浏览器事件对象的兼容性问题

```
/*if(!event){  
    event = window.event;  
}*/  
  
event = event || window.event;  
/*
```

## 事件的冒泡

事件的冒泡(Bubble)

所谓的冒泡指的就是事件的向上传导，当后代元素上的事件被触发时，其祖先元素的相同事件也会被触发

大部分情况冒泡都是有用的，也是自然而然可以理解的

如果不希望发生事件冒泡，可以用**事件对象**来取消冒泡

取消冒泡：**cancelBubble属性**

语法：

**事件.cancelBubble=true**

```
//为s1绑定一个单击响应函数  
var s1 = document.getElementById("s1");  
s1.onclick = function(event){  
    event = event || window.event;  
    alert("我是span的单击响应函数");  
  
    //取消冒泡  
    //可以将事件对象的cancelBubble设置为true，即可取消冒泡  
    event.cancelBubble = true;  
};
```

## 事件的委派

事件的委派

指将事件统一给元素的**共同的祖先元素**，这样当后代元素上的事件触发时，会一直冒泡到祖先元素

从而通过祖先元素的响应函数来处理事件。

事件委派是利用了冒泡，**通过委派可以减少事件绑定的次数，提高程序的性能**

**事件的target属性**，表示触发事件的那个元素对象，并不是响应函数绑定的对象，比如响应事件绑定在ul中，ul中还有li等标签，响应事是在li中触发的，就会返回li

语法：事件.target

```
//为ul绑定一个单击响应函数
u1.onclick = function(event){
    event = event || window.event;

    /*
     * target
     * - event中的target表示的触发事件的对象
     */
    alert(event.target);

    //如果触发事件的对象是我们期望的元素，则执行否则不执行
    //alert("我是ul的单击响应函数");
};
```

## 事件的绑定

一个事件绑定多个响应函数

**addEventListener()方法**。通过这个方法可以同时为一个元素的相同事件同时定多个响应函数，但是**IE8及以下浏览器不支持，此方法中this指向的是元素**

这样当事件被触发时，响应函数将会按照函数的绑定顺序执行

### 三个参数

- 1.事件的字符串，**不要on**
- 2.回调函数，**当事件触发时该函数会被调用**
- 3.是否在捕获阶段触发事件，需要一个布尔值，**一般都传 False**

例子：

```

/*
 * - 通过这个方法也可以为元素绑定响应函数
 * - 参数:
 *   1.事件的字符串, 不要on
 *   2.回调函数, 当事件触发时该函数会被调用
 *   3.是否在捕获阶段触发事件, 需要一个布尔值, 一般都传false
 *
 * 使用addEventListener()可以同时为一个元素的相同事件同时绑定多个响应函数,
 * 这样当事件被触发时, 响应函数将会按照函数的绑定顺序执行|
 */
btn01.addEventListener("click",function(){
    alert(1);
},false);

btn01.addEventListener("click",function(){
    alert(2);
},false);

btn01.addEventListener("click",function(){
    alert(3);
},false);

```

### attachEvent()方法

在IE5-10中可以使用 attachEvent () 来绑定多个事件

不同的是, **他是后绑定先执行**, 和addEventListener()方法顺序相反, **且此方法中this指向的是window**

#### **两个参数:**

1.事件的字符串, **要加上on**

2.回调函数

```

/*
 * attachEvent()
 * - 在IE8中可以使用attachEvent()来绑定事件
 * - 参数:
 *   1.事件的字符串, 要on
 *   2.回调函数
 */
btn01.attachEvent("onclick",function(){
    alert(1);|
});
```

上面这两中方法, 为了实现兼容, 自定义一个**bind函数**

```

    * 参数:
    * obj 要绑定事件的对象
    * eventStr 事件的字符串(不要on)
    * callback 回调函数
    */
function bind(obj , eventStr , callback){

    if(obj.addEventListener){
        //大部分浏览器兼容的方式
        obj.addEventListener(eventStr , callback , false);
    }else{
        //IE8及以下
        obj.attachEvent("on"+eventStr , callback);
    }
}

```

虽然能用了，但是上面所说的两种方法的差异韩式存在

**改进：**消除this的差异，改成都指向元素

```

    * 参数:
    * obj 要绑定事件的对象
    * eventStr 事件的字符串(不要on)
    * callback 回调函数
    */
function bind(obj , eventStr , callback){
    if(obj.addEventListener){
        //大部分浏览器兼容的方式
        obj.addEventListener(eventStr , callback , false);
    }else{
        /*
        * this是谁由调用方式决定
        * callback.call(obj)
        */
        //IE8及以下
        obj.attachEvent("on"+eventStr , function(){
            //在匿名函数中调用回调函数
            callback.call(obj);
        });
    }
}

```

attachEvent()方法中，**this返回的是window，说明它底层采用的是函数型调用**，函数有个方法是：`函数.call(obj)`，来修改函数的this指向。

**直接添加callback.call(obj)不行。因为call特点是函数就直接将相应的函数运行，例子中不允许这么做**

事件的传播

```

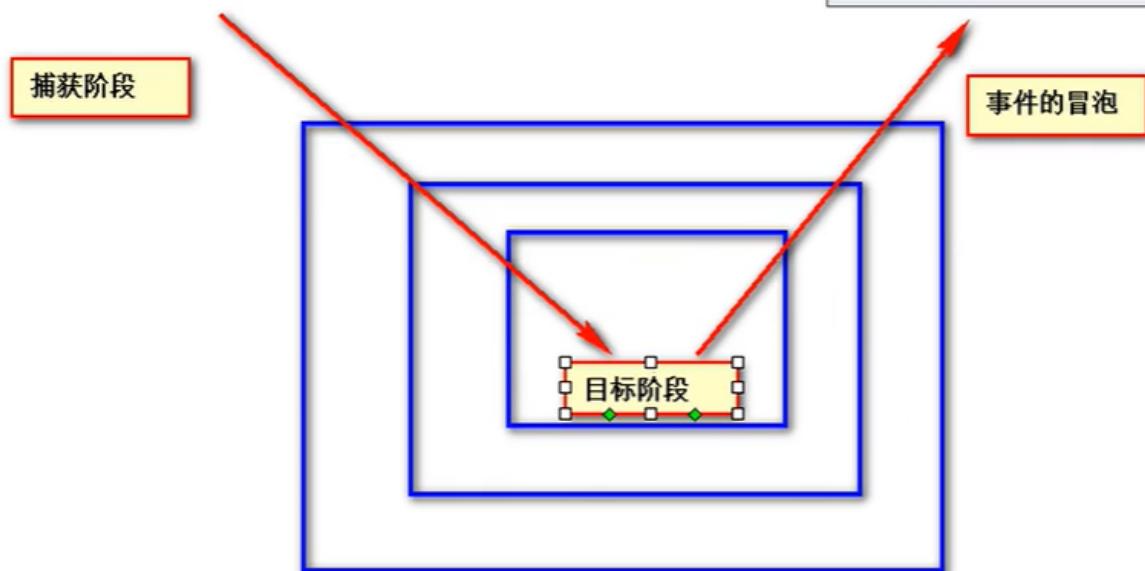
/*
 * 事件的传播
 * - 关于事件的传播网景公司和微软公司有不同的理解
 * - 微软公司认为事件应该是由内向外传播，也就是当事件触发时，应该先触发当前元素上的事件，  

 * 然后再向当前元素的祖先元素上传播，也就是说事件应该在冒泡阶段执行。
 * - 网景公司认为事件应该是由外向内传播的，也就是当前事件触发时，应该先触发当前元素的最外层的祖先元素的事件，  

 * 然后在向内传播给后代元素
 * - W3C综合了两个公司的方案，将事件传播分成了三个阶段
 * 1.捕获阶段
 * - 在捕获阶段时从最外层的祖先元素，向目标元素进行事件的捕获，但是默认此时不会触发事件
 * 2.目标阶段
 * - 事件捕获到目标元素，捕获结束开始在目标元素上触发事件
 * 3.冒泡阶段
 * - 事件从目标元素向他的祖先元素传递，依次触发祖先元素上的事件
 *
 * - 如果希望在捕获阶段就触发事件，可以将addEventListen()的第三个参数设置为true  

 * 一般情况下我们不会希望在捕获阶段触发事件，所以这个参数一般都是false
 *
 * - IE8及以下的浏览器中没有捕获阶段

```



## 滚轮事件

**onmousewheel** 滚轮事件，此事件在**火狐**中不兼容

在**火狐**中，使用**DOMMouseScroll**

但是，**火狐**中该事件的绑定必须使用**addEventListener**

**ps：更新，现在火狐、谷歌都可以用onwheel，ie用onmousewheel**

**事件.wheelDelta**，鼠标滚动方向，**火狐**中不支持

```

//判断鼠标滚轮滚动的方向
//event.wheelDelta 可以获取鼠标滚轮滚动的方向
//向上滚 120 向下滚 -120
//wheelDelta这个值我们不看大小，只看正负

```

```

//alert(event.wheelDelta);

```

在**火狐**中，用**事件.detail**获取鼠标滚动事件

```
//wheelDelta这个属性火狐中不支持  
//在火狐中使用event.detail来获取滚动的方向  
//向上滚 -3 向下滚 3  
//alert(event.detail);
```

实现兼容

```
//判断鼠标滚轮滚动的方向  
if(event.wheelDelta > 0 || event.detail < 0){  
    alert("向上滚");  
}else{  
    alert("向下滚");  
}
```

有时候自定义的div块中定义的鼠标滚动响应事件，浏览器也在页面滚动

**取消默认行为，要写在相应的元素中的滚动事件的响应函数中去**

```
/*  
 * 使用addEventListener()方法绑定响应函数，取消默认行为时不能使用return false  
 * 需要使用event来取消默认行为event.preventDefault();  
 * 但是IE8不支持event.preventDefault();这个玩意，如果直接调用会报错  
 */  
event.preventDefault && event.preventDefault();  
  
/*  
 * 当滚轮滚动时，如果浏览器有滚动条，滚动条会随之滚动，  
 * 这是浏览器的默认行为，如果不希望发生，则可以取消默认行为  
 */  
return false;
```

使用 addEventListener()方法绑定响应函数，**取消默认行为时不能使用 return false**

此时，可以用事件对象的preventDefault()方法来取消默认行为，但是ie8不支持

例子中的event.preventDefault()&&event.preventDefault()作用是让IE浏览器来判断，ie两个都不能用，就相当于相与为假，执行下面的代码，不会报错，单写一个event.preventDefault()就报错了

## 键盘事件

**onkeydown** 键盘按下

如果一直按着某个事件不松手，会连续触发

当连续触发时，第一次和第二次会有一个小的停顿，之后的会非常快，这是为了**防止误操作**

**onkeyup** 键盘松开

键盘事件一般都绑定给可以**获取焦点**的事件，或者**document**

**按键事件.keyCode**获取按键的Unicode编码

```
document.onkeydown = function(event){  
    event = event || window.event;  
  
    /*  
     * 可以通过keyCode来获取按键的编码  
     * 通过它可以判断哪个按键被按下  
     */  
  
    //判断一个y是否被按下  
    if(event.keyCode === 89){  
        console.log("y被按下了");  
    }  
  
};
```

判断哪个键被按下

**altKey**

**ctrlKey**

**shiftKey**

```
//判断y和ctrl是否同时被按下  
if(event.keyCode === 89 && event.ctrlKey){  
    console.log("ctrl和y都被按下了");  
}
```

例子

```
input.onkeydown = function(){  
    console.log("按键被按下了");  
  
    //在文本框中输入内容，属于onkeydown的默认行为  
    //如果在onkeydown中取消了默认行为，则输入的内容，不会出现在文本框中  
    return false;  
};
```

在文本框中不允许输入数字：

```
//获取input
var input = document.getElementsByTagName("input")[0];

input.onkeydown = function(event){
    event = event || window.event;

    //console.log(event.keyCode);
    //数字 48 - 57
    //使文本框中不能输入数字
    if(event.keyCode >= 48 && event.keyCode <= 57){
        //在文本框中输入内容，属于onkeydown的默认行为
        //如果在onkeydown中取消了默认行为，则输入的内容，不会出现在文本框中
        return false;
    }
}
```

## BOM (浏览器对象模型)

BOM可以使我们通过JS来操作浏览器

BOM对象 调用时首字母都小写

**window**

代表的是整个浏览器的窗口，同时 window也是网页中的全局对象

**navigator**

代表的当前浏览器的信息，通过该对象可以识别不同的浏览器

**location**

代表当前浏览器的地址栏信息，通过 Location可以获取地址栏信息，或者操作浏览器跳转页面

**history**

代表浏览器的历史记录，可以通过该对象来操作浏览器的历史记录

由于隐私原因，该对象不能获取到具体的历史记录，只能操作浏览器向前或向后翻页

而且该操作只在当次访回时有效

**screen**

代表用户的屏幕的信息，通过该对象可以获取到用户的显示器的相关信息

这些BOM对象在浏览器中都是作为 **window对象的属性保存的**，

可以通过 **window对象来使用**，也可以直接使用

```
32
33         //console.log(navigator);
34         //console.log(location);
35         console.log(history);
36
37
38     </script>
```

控制台

```
[当前是 [边改边看控制台], 点右上角工具条相应按钮可切换控制台]
[Web浏览器] "[object Navigator]" /day17/01.BOM.html (33)
[Web浏览器] "http://127.0.0.1:8020/day17/01.BOM.html" /day17/01.BOM.html (34)
[Web浏览器] "[object History]" /day17/01.BOM.html (35)
```

## navigator

代表的当前浏览器的信息，通过该对象可以来识别不同的浏览器

**由于历史原因， Navigator对象中的大部分属性都已经不能帮助我们识别浏览器了**

一般我们只会使用 **useragent** 来判断浏览器的信息，

useragent是一个字符串，这个字符串中包含有用来描述浏览器信息的内容，不同的浏览器会有不同的 useragent

```
火狐的userAgent
Mozilla/5.0 (Windows NT 6.1; WOW64; rv:50.0) Gecko/20100101 Firefox/50.0

Chrome的userAgent
Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.274

IE8
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/7.0; SLCC2; .NET CLR 2.0.5072)

IE9
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/7.0; SLCC2; .NET CLR 2.0.5072)

IE10
Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; WOW64; Trident/7.0; SLCC2; .NET CLR 2.0.5072)

IE11
Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729;
- 在IE11中已经将微软和IE相关的标识都已经去除了，所以我们基本已经不能通过UserAgent来识别一个浏览器是否是IE了
```

如图所示，如果通过 userAgent 不能判断 IE11，还可以通过一些浏览器中特有的对象，来判断浏览器的信息

比如：**Activexobject** 是 IE 中独有的，通过判断 **Activexobject in window** 真假来判断是否是 IE 浏览器（但不能用 if(window.ActiveXObject)，因为 IE 鸡贼的把这个值强行改为 false）

**通过 userAgent 的信息来区分浏览器：**

```
var ua = navigator.userAgent;  
  
console.log(ua);  
  
if(/firefox/i.test(ua)){  
    alert("你是火狐！！！");  
}else if(/chrome/i.test(ua)){  
    alert("你是Chrome");  
}else if(/msie/i.test(ua)){  
    alert("你是IE浏览器~~~");  
}else if("ActiveXObject" in window){  
    alert("你是IE11, 枪毙了你~~~");  
}  
}
```

## history

**history.length属性** 获取当前访问的链接数量

**back()方法**

可以用来回退到上一个页面，作用和浏览器的回退按钮一样

**forward()方法**

可以跳转下一个页面，作用和浏览器的前进按钮一样

ps: back和forward这两个是方法，作用到点击事件的响应函数中的话，就会直接跳转

**go()方法**，可以用来跳转到指定的页面

它需要一个整数作为参数：

1：表示向前跳转一个页面相当于 forward()

2：表示向前跳转两个页面

-1：表示向后跳转一个页面

-2：表示向后跳转两个页面

## location

如果直接打印location，则可以获取到地址栏的信息（当前页面的完整路径）

如果直接将location属性修改为一个完整的路径，或相对路径，则页面会自动跳转到该路径，放到 onclick 的响应函数中去，点击就跳转了。并且会生成相应的历史记录

```
btn.onclick = function(){

    //如果直接打印location，则可以获取到地址栏的信息（当前页面的完整路径）
    //alert(location);

    /*
     * 如果直接将location属性修改为一个完整的路径，或相对路径
     * 则我们页面会自动跳转到该路径，并且会生成相应的历史记录
     */
    //location = "http://www.baidu.com";
    //location = "01.BOM.html";
```

**location.assign()方法**，作用和location一样，只不过将地址作为参数传给方法

```
/*
 * assign()
 * - 用来跳转到其他的页面，作用和直接修改location一样
 */
location.assign("http://www.baidu.com");
```

**location.reload()方法**

用于重新加载当前页面，作用和刷新按钮一样

如果在方法中传递一个true,作为参数，则会强制清空缓存刷新页面

```
/*
 * reload()
 * - 用于重新加载当前页面，作用和刷新按钮一样
 * - 如果在方法中传递一个true, 作为参数，则会强制清空缓存刷新页面
 */
location.reload(true);
```

**location.replace()方法**

可以使用一个新的页面替换当前页面，调用完毕也会跳转页面

不会生成历史记录，不能使用回退按钮

```
/*
 * replace()
 * - 可以使用一个新的页面替换当前页面，调用完毕也会跳转页面
 * 不会生成历史记录，不能使用回退按钮回退
 */
location.replace("01.BOM.html");
```

## window

很多window方法前面已经讲过了，并且window可以省略

### setInterval()方法

#### 定时调用

可以将一个函数，每隔一段时间执行一次

#### 参数：

1.回调函数，该函数会每隔一段时间被调用一次

2.每次调用间隔的时间，单位是毫秒

```
var num = 1;

setInterval(function(){
    count.innerHTML = num++;
}, 1000);
```

该方法会返回一个**返回值**：Number类型的数据

这个数字用来作为**定时器的唯一标识**

将方法赋值给变量就是将**返回值赋值给变量**的过程，**此时这个定时器仍然开启，响应函数仍然运行**

### clearInterval()方法

可以用来关闭一个定时器

方法中需要一个定时器的**标识作为参数**，也就是上面说的返回值，这样将关闭标识对应的定时器

clearInterval()可以接收任意参数，

如果参数是一个有效的定时器的标识，则停止对应的定时器

如果参数是一个**无效的标识**，则什么也不做，不会报错

例子：

```
var num = 1;

var timer = setInterval(function(){
    count.innerHTML = num++;

    if(num == 11){
        //关闭定时器
        clearInterval(timer);
    }
}, 1000);
```

如果将定时器绑定到一个按钮上，点一下就开启一个定时器，**点多次就开启多个定时器**，如果定时器赋值给某一个变量，后来的定时器**返回值就会覆盖之前的**，用clearInterval(变量)关闭定时器也**只能关掉最后一个**，之前的就再也关不到了，看上去定时器也会越来越快

解决方法：**在开始定时器之前就将要开始的定时器关闭，根据clearInterval()的特点，也不会报错**

```
//在开启定时器之前，需要将上一个定时器关闭
clearInterval(timer);
|
/*
 * 开启一个定时器，来自动切换图片
 */
timer = setInterval(function(){
    //使索引自增
    index++;
    //判断索引是否超过最大索引
    /*if(index >= imgArr.length){
        //则将index设置为0
        index = 0;
    }*/
    index %= imgArr.length;
    //修改img1的src属性
    img1.src = imgArr[index];
},1000);
};
```

### setTimeout()方法

延时调用

延时调用，和定时调用类似，也是有**两个参数**

延时调用一个函数不马上执行，而是隔一段时间以后再执行，**并且只会执行一次**

```
/*
 * 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再执行，而且只会执行一次
 *
 */
setTimeout(function(){
    console.log(num++);
},3000);
```

有一个返回值，用来标识延时调用，**可以将方法赋给一个变量**

```
var timer = setTimeout(function(){
    console.log(num++);
}, 3000);
```

### clearTimeout()方法

用来关闭一个延时调用，参数是延时调用的返回值

```
var timer = setTimeout(function(){
    console.log(num++);
}, 3000);

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

## 类的操作

之前修改元素的样式是直接通过js来修改元素的style属性，如下图：

```
* 通过 style 属性来修改元素的样式，每修改一个样式，浏览器就需要重新渲染一次页面
* 这样的执行的性能是比较差的，而且这种形式当我们要修改多个样式时，也不太方便
*/
●
box.style.width = "200px";
box.style.height = "200px";
box.style.backgroundColor = "yellow";
```

这样修改的样式是直接在标签中修改的，**属于内联样式，结构与行为耦合**

并且**每修改一个样式，浏览器就需要重新渲染一次页面**

这样的执行的**性能是比较差的**，而且这种形式当我们要修改多个样式时，也不太方便

所以不推荐使用

可以通过**className属性**修改元素的类名，更新过的样式写到新的类的css选择器中，这样只用改一下类名，同时修改多个样式，行为与表现也分离了，浏览器也只用执行一次就行

```
.b2{
    width: 300px;
    height: 300px;
    background-color: yellow;
}

box.className = "b2";
```

如果只是想更新一些属性，并不想完全覆盖，可以将类名 + “另一个类名”，将类名进行拼串操作，新的类名前面要有空格，这样元素就有两个类名了，新的类名选择器中写要更新的样式

```
box.className += " b2";
```

改进解决方法：

可以创建一个下面的函数，进行**属性的添加**

```
// 定义一个函数，用来向一个元素中添加指定的class属性值
/*
 * 参数：
 *   obj 要添加class属性的元素
 *   cn 要添加的class值
 *
 */
function addClass(obj , cn){    [
    obj.className += " "+cn;    ]
}
```

但是这样一个缺点，添加之前不仅行类名检查，这样同一个类可以添加多次

可以添加一个**类名检查的函数**，函数两个参数：元素、类名

因为要用到正则表达式，类名作为变量传入，就不能使用字面量来创建正则表达式（不能使用变量），只能用构造函数创建法

```
function hasClass(obj , cn){  
    //判断obj中有没有cn class  
    //创建一个正则表达式  
    //var reg = /\bb2\b/;  
    var reg = new RegExp("\b"+cn+"\b");  
  
    return reg.test(obj.className);  
}
```

将上面创建的两个addClass和hasClass函数结合起来，整合成一个addClass函数。

```
function addClass(obj , cn){  
    //检查obj中是否含有cn  
    if(!hasClass(obj , cn)){  
        obj.className += " "+cn;  
    }  
}
```

还可以写一个**删除指定类选择器的函数**, 函数中使用replace方法, 将指定类名替换为空串

```
/*  
 * 删除一个元素中的指定的class属性  
 */  
function removeClass(obj , cn){  
    //创建一个正则表达式  
    var reg = new RegExp("\b"+cn+"\b");  
  
    //删除class  
    obj.className = obj.className.replace(reg , "");  
}
```

除此之外, 还可以创建一个切换类的函数toggleClass:

如果元素中具有该类, 则删除

如果元素中没有该类, 则添加

```
* toggleClass可以用来切换一个类  
* 如果元素中具有该类, 则删除  
* 如果元素中没有该类, 则添加  
*/  
function toggleClass(obj , cn){  
  
    //判断obj中是否含有cn  
    if(hasClass(obj , cn)){  
        //有, 则删除  
        removeClass(obj , cn);  
    }else{  
        //没有, 则添加  
        addClass(obj , cn);  
    }  
}
```

# JSON

js中的对象只有JS自己认识，其他的语言都不认识

JSON就是一个**特殊格式的字符串**，这个字符串**可以被任意的语言所识别**，

并且**可以转换为任意语言中的对象**，JSON在开发中主要用来**数据的交互**

javascript object notation (js对象表示法)

JSON和JS对象的格式一样，只不过JSON字符串中的**属性名必须加双引号**，其他的语法和js一样

json分类：1.对象，使用{}  
2.数组，使用[]

JSON中只允许的值：

- 1.字符串
- 2.数值
- 3.布尔值
- 4.null
- 5.对象（只是普通的对象，不包括函数）
- 6.数组

下面的都是json对象

```
var obj = '{"name": "孙悟空", "age": 18, "gender": "男"}';
var arr = '[1, 2, 3, "hello", true]';
var obj2 = '{"arr": [1, 2, 3]}';

var arr2 = '[{"name": "孙悟空", "age": 18, "gender": "男"}, {"name": "孙悟空", "age": 18, "gender": "男"}]'
```

在JS中，为我们提供了一个工具类，就叫JSON

这个对象可以帮助我们把一个JSON对象转化为一个JS对象，也可以把一个JS对象转化为一个JSON对象

**JSON-->JS对象**

**JSON.parse(JSON字符串)**

可以将一个JSON字符串转化为JS对象（同样，如果在java中，也可以转换为Java对象）

它需要一个JSON字符串作为参数，会将该字符串转换为**JS对象并返回**

```
4
5     var json = '{"name":"孙悟空","age":18,"gender":"男"
6
7     /*
8      * json --> js对象
9      * JSON.parse()
10     - 可以将以JSON字符串转换为js对象
11     - 它需要一个JSON字符串作为参数，会将该字符串转换为JS对
12   */
13
14     var o = JSON.parse(json);
15
16     console.log(o.gender);

```

控制台

是 [边改边看控制台]，点右上角工具条相应按钮可切换控制台

leb浏览器] "孙悟空" /day18/04.JSON.html (56)

leb浏览器] "18" /day18/04.JSON.html (56)

leb浏览器] "男" /day18/04.JSON.html (56)

```
var o2 = JSON.parse(arr);

//console.log(o.gender);
console.log(o2[1]);
```

控制台

是 [边改边看控制台]，点右上角工具条相应按钮可切换控制台

浏览器] "2" /day18/04.JSON.html (58)

JS对象-->JSON字符串

### JSON.stringify()

可以将一个JS对象转化为一个JSON字符串

需要一个js对象作为参数，会返回一个JSON字符

```
0 var obj3 = {name:"猪八戒" , age:28 , gender:"男"};
1
2
3/*
4 * JS对象 ---> JSON
5 * JSON.stringify()
6 * - 可以将一个JS对象转换为JSON字符串
7 * - 需要一个js对象作为参数，会返回一个JSON字符串
8 */
9
0 var str = JSON.stringify(obj3);
1 console.log(str);
2
3
4
```

控制台

是 [边改边看控制台]，点右上角工具条相应按钮可切换控制台

IE浏览器 ] {"name": "猪八戒", "age": 28, "gender": "男"} /d

真整个变成了字符串，并且属性名也加上了双引号

JSON这个对象在**IE7及以下的浏览器**不能使用，会报错

### eval()函数

这个函数可以用来执行一段字符串形式的JS代码，并将执行结果返回

```
/*
 * eval()
 * - 这个函数可以用来执行一段字符串形式的JS代码，并将执行结果返回
 */

var str2 = "alert('hello');";

eval(str2);
```

如果使用eval()执行的字符串中含有`{}，它会将{}当成是代码块。如果不希望将其当成代码块解析，则需要在字符串前后各加一个\\n。`

```
var str = '{"name": "孙悟空", "age": 18, "gender": "男"}';
```

```
/*
 * eval()
 * - 这个函数可以用来执行一段字符串形式的JS代码，并将执行结果返回
 * - 如果使用eval()执行的字符串中含有{}，它会将{}当成是代码块
 *   如果不希望将其当成代码块解析，则需要在字符串前后各加一个()
 */
```

```
var str2 = "alert('hello');";
```

```
var obj = eval("(" + str + ")");
```

```
console.log(obj);
```

eval()这个函数的功能很强大，可以直接执行一个字符串中的JS代码

但是在开发中尽量不要使用，首先它的执行性能比较差，然后它还具有安全隐患

## PS

**prompt** : prompt() 弹出一个文本框，可以在文本框中输入内容，可以将提示文字作为函数的参数。函数的返回值是**string**类型的

```
var name=prompt("Please enter your name","");

```

需要利用输入的值进行运算时，先进行**类型的转换**（比如在返回值前面加上加号）

```
var num1 = +prompt("请输入第一个数:");
```

document.write()是向网页里面输出，换行的话不能用\n，而应该用标签

```
4 <meta charset="UTF-8">
5 <title></title>
6 <script type="text/javascript">
7
8
9 /*
10  * 向页面中输出连续的数字
11 */
12 document.write(1+<br />);
13 document.write(2+<br />);
14 document.write(3+<br />);
15 document.write(4+<br />);
16
```

---

#### 在html代码中 和空格的区别：

在html代码中每输入一个转义字符 就表示一个空格，输入十个 ，页面中就显示10个空格位置。

而在html代码中输入空格，不管输入多少个空格，最终在页面中显示的空格位置只有一个。

---

html-css:

- **inline:**

1. 使元素变成行内元素，拥有行内元素的特性，即可以与其他行内元素共享一行，不会独占一行。
2. 不能更改元素的height, width的值，大小由内容撑开。
3. 可以使用padding, margin的left和right产生边距效果，但是top和bottom就不行。

- **block:**

1. 使元素变成块级元素，独占一行，在不设置自己的宽度的情况下，块级元素会默认填满父级元素的宽度。
2. 能够改变元素的height, width的值
3. 可以设置padding, margin的各个属性值, top, left, bottom, right都能够产生边距效果。

- **inline-block:**

1. 结合了inline与block的一些特点，结合了上述inline的第一个特点和block的第二个特点。
2. 用通俗的话讲，就是不独占一行的块级元素。

.....

**console.time()**来创建一个计时器，括号里面写计时器的名字

**console.timeEnd()**来终止一个计时器，括号里面写计时器的名字，并且输出脚本执行的时间

.....

**Math.sqrt()**对一个数进行开方

.....

**HTML页面的 <script type="text/javascript">含义。**

在老的HTML版本，如果需要在HTML页面中声明一段js脚本，需要做<script type="text/javascript">的声明，表明在<script type="text/javascript">和</script>中添加的文本是js脚本；

在HTML5以后，HTML5的默认脚本语言就是javascript即js，所以无需显示声明 type="text/javascript"

，直接使用<script> </script>即可。

.....

<meta> 标签是 HTML 语言头部的一个辅助性标签，我们可以定义页面编码语言、搜索引擎优化、自动刷新并指向新的页面、控制页面缓冲、响应式视窗等！

属性	值	描述
charset(H5 新)	character_set	定义文档的字符编码。
content	text	定义与 http-equiv 或 name 属性相关的元信息。
http-equiv	content-type default-style refresh	把 content 属性关联到 HTTP 头部。
name	application-name author description generator keywords	把 content 属性关联到一个名称。
scheme(H5 删除)	format/URI	HTML5不支持。 定义用于翻译 content 属性值的格式。

.....

**instanceof: 使用instanceof可一检查一个对象是否是一个类的实例**

```
/*
 * 使用instanceof可以检查一个对象是否是一个类的实例
 */
console.log(per instanceof Person);
```

.....

**isArray()方法: 判断一个对象是不是数组**

语法:

Array.isArray(对象);

.....

**innerHTML属性**, 该属性可以获取某个元素内的HTML代码

**innerText属性**, 该属性可以获取元素内部的文本内容, 他和innerHTML类似, 不同的是他会自动地将HTML标签去除, 只保留文本内容

## nodeValue属性

获取某个节点的文本内容，大部分情况下做到的效果跟innerText一样，但是比innerText麻烦

```
//获取bj中的文本节点  
/*var fc = bj.firstChild;  
alert(fc.nodeValue);*/  
  
alert(bj.firstChild.nodeValue);
```

checked属性，返回或者设置HTML中<checkbox>类型的标签是否被选中

```
//通过多选框的checked属性可以来获取或设置多选框的选中状态  
//alert(items[i].checked);  
  
items[i].checked = true;
```

HTML的超链接默认点击后都会跳转

不希望出现跳转行为的话，如果为超链接绑定了响应事件，可以在点击事件的响应函数最后return false，来取消默认行为

```
//为每个超链接都绑定一个单击响应函数  
for(var i=0 ; i < allA.length ; i++){  
    allA[i].onclick = function(){  
        alert("hello");  
  
        /*  
         * 点击超链接以后，超链接会跳转页面，这是超链接的默认行为。  
         * 但是此时我们不希望出现默认行为，可以通过在响应函数的最后return false来取消默认行为  
         */  
        return false;  
    },  
}
```

或者，可以直接在标签中href属性中将内容改为javascript:;也可以取消跳转

```
<td><a href="javascript:;">Delete</a></td>
```

.....

confirm是window下的方法，跟alert相似

只不过confirm有确认和取消两个按钮，alert只有确认

如果用户点击确认，则返回true，如果用户点击取消，则返回false

可以在某些操作（如删除操作）的响应函数中使用confirm，来进行行为的确认

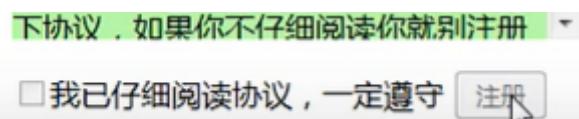
```
var flag = confirm("确认删除吗？");

//如果用户点击确认
if(flag){
    //删除tr
    tr.parentNode.removeChild(tr);
}
```



如果为表单项添加 **disabled="disabled"** 则表单项将变成不可用的状态，相应按钮或者选项将变成不可用的

```
<body>
    <h3>欢迎亲爱的用户注册</h3>
    <p id="info">□
        <!-- 如果为表单项添加disabled="disabled" 则表单项将变成不可用的状态 -->
        <input type="checkbox" disabled="disabled" />我已仔细阅读协议，一定遵守
        <input type="submit" value="注册" disabled="disabled" />
</body>
```



元素中也有disabled属性

```
* disabled属性可以设置一个元素是否禁用,
* 如果设置为true，则元素禁用
* 如果设置为false，则元素可用
*/
inputs[0].disabled = false;
inputs[1].disabled = false;
```

获取滚动条距离顶端的距离，通过 **scrollTop属性**

chrome认为浏览器的滚动条是 **body** 的，可以通过 `document.body.scrollTop` 来获取

火狐等浏览器认为滚动条是 **HTML** 的。可以通过 `document.documentElement.scrollTop`

```
//获取滚动条滚动的距离
/*
 * chrome认为浏览器的滚动条是body的，可以通过body.scrollTop来获取
 * 火狐等浏览器认为浏览器的滚动条是html的，
 */
var st = document.body.scrollTop || document.documentElement.scrollTop;
//var st = document.documentElement.scrollTop;
```

注意：现在 `document.documentElement.scrollTop` 都可以用了

clientX和clientY获取鼠标相对于当前可见的页面坐标

div块的偏移量一般都是用的相对于整个原始页面的偏移量

pageX和PageY获取鼠标相对于整个原始页面的偏移量，但是在IE8中不支持

.....  
**元素.className**, 此属性可以返回标签的class属性的内容

.....  
**事件的target属性**, 表示触发事件的那个元素对象

语法：事件.target

```
//为ul绑定一个单击响应函数
u1.onclick = function(event){
    event = event || window.event;

    /*
     * target
     * - event中的target表示的触发事件的对象
     */
    alert(event.target);

    //如果触发事件的对象是我们期望的元素，则执行否则不执行
    //alert("我是ul的单击响应函数");
};
```

.....  
**bind()方法**

二个作用：

用来修改函数的this指向

用来向调用的函数传递参数

## 与call和apply的联系：

作用和call和apply相似

bind参数的传递规则和call一样，一个个传递，apply参数无论几个都要封装成数组

call() 和apply()，都是立马就调用了对应的函数，而 bind() 不会，bind() 会生成一个新的函数(下面实例)

例子：

```
1 var m = {  
2     "x" : 1  
3 };  
4 function foo(y) {  
5     alert(this.x + y);  
6 }  
7 foo.apply(m, [5]);  
8 foo.call(m, 5);  
9 var foo1 = foo.bind(m, 5);  
10 foo1();
```

详解，看这个就明白了：<https://blog.csdn.net/qiqingjin/article/details/50706087>

## 末尾处