一.小技巧

a标签点击不跳转

```
<a href="javascript:"></a>
```

typeof 检查标签

返回的结果是字符串形式

instanceof 检查标签

检查一个对象是否是一个类的实例

```
console.log(person instance of Person) //true
console.log(dog instance of Person) //false
console.log(person instance of Object) //true 任何对象都是object的后代
```

输出unicode编码

1. js中使用 \u2620 (16进制编码)
2. html中使用 ☠ ☠(转换为10进制编码)

二.数据类型

1. String字符串
2. Number数值
3. Boolean布尔值
4. Null空值
5. Undefined未定义
6. Object 对象

> 其中前五种属于基本数据类型，Object属于引用数据类型。

2.1 Number

1. 正无穷值 a=Infinity;
2. 负无穷值 a=-Infinity;
3. 正最大值 a=Number.MAX_VALUE
4. 正最小值 a=Number.MIN_VALUE
5. `NaN` a="abc" * "bcd" `NaN`也是一个`Number`类型
6. var c=0.1+0.2 可能得到一个不精确的结果，不要用js进行精确运算。

2.2 Null

1. var a=null;
2. `null`值用来表示一个空的对象
3. `typeof`检查一个`null`值时，会返回`Object`

2.3 Undefined

1. var a;
2. `undefined`值用来表示未定义的对象
3. `typeof`检查，返回`undefined`

2.4 强制类型转换(String)

1. 调用被转换数据类型的`toString()`

> **var b=a.toString()**
>
> null和undefined没有toString()方法。

2. 调用`String()`,

> **a=String(a)**
> null和undefined会转换为"null"和"undefined"

2.5强制类型转换(Number)

1.Number()方法

1. 字符串转正常数字，转为Number
2. 字符串不是数字，转为NaN
3. 字符串是空串，转为0
4. null，转为0
5. undefined，转为NaN

> var b=Number(a)

2.parseInt()

只针对字符串使用，从前往后读取数字,读到非法就停止

> var b=parseInt(a)
>
> '123px' -> 123
>
> '123.456' -> 123
>
> '123px456' -> 123

> null/undefined ->"null"/"undefined" -> NaN

3.parseFloat()

> var b=parseFloat(a)
>
> '123.456.789' -> 123.456
>
> null/undefined -> NaN

4.+

> 通过在String类型前增加一个加号来转变为Number类型
>
> var a="1"
>
> a=+a

2.6不同进制数

1. 十六进制 0x??
2. 八进制 0??
3. 二进制 0b??

> parseInt('070')
>
> 有的浏览器会当成八进制解析 -> 56
>
> 有的浏览器会当成十进制解析 -> 70
>
> 解决方案 parseInt('070',10)

2.7强制类型转换(Boolean)

1. **数字**转布尔，除了`0`和`NaN`，其余都是`true`
2. **字符串**转布尔，除了空串(""),其余都是`true`
3. `null`和`undefined`转布尔，结果是`false`
4. **对象**转布尔，结果是`true`

2.8算数运算符

1. true=1,false=0,null=0,undefined=NaN
2. 1+NaN=NaN
3. "abc"+"def"="abcdef"
4. 123+"1"="1231"
5. 1+2+'3'='33'

6. '1'+2+3='123'
7. true+"hello"="truehello"
8. 100-'1'=99
9. 2*'8'=16
10. 2* null =0

> 快速转为String.
>
> var c=123;
>
> c=c+"";
>
> 快速转为Number.
>
> var c='123';
>
> c=c-0;

2.9非布尔值的与或运算

- 与运算

1. 第一个值为true，返回第二个值
2. 第一个值为false，则返回第一个值

> 5 && 6 -> 6
>
> 0 && 2 -> 0
>
> 2 && 0 -> 0
>
> NaN && 0 -> NaN
>
> 0 && NaN -> 0

- 或运算

1. 第一个值为true，返回第一个值
2. 第一个值为false，返回第二个值

> 2 || 1 -> 2
>
> 2 || NaN -> 2
>
> 2 || 0 -> 2
>
> NaN || 0 -> 0

> NaN || 1 -> 1

2.10 关系运算符

1. 任何值和NaN做任何比较都是false

> 11>'hello' -> false
>
> 11<'hello'-> false

2. 比较两个字符串时比较的是Unicode编码

> '11'<'5' -> true
> 'a'<'b' -> true
> 'abb'>'a' -> true

2.11 相等运算符

==

会做类型转换

1. "1"==1 -> true
2. true=='1' -> true
3. null == 0 -> false(需要记住)
4. undefined == null -> true (undefined衍生自null)
5. NaN == NaN -> false (NaN不和任何值相等)

判断是否是NaN

1. b==NaN 一定为false **不能判断**
2. isNaN(b) **可以判断**

===

不会做类型转换,如果类型不同，直接返回false

1. "123" === 123 -> false
2. null === undefined -> false

!==

不会做类型转换,如果类型不同，直接返回true

1. 123 !== "123" -> true

三.对象

3.1 对象的分类

1. **内建对象**

- 由ES标准中自定义的对象，在任何的ES的实现中都可以使用
- 比如：Math String Number Boolean Function Object……

2. **宿主对象**

- 由JS的运行环境提供的对象，目前来讲主要指由浏览器提供的对象
- 比如BOM，DOM

3. **自定义对象**

- 由开发人员自己创建的对象

3.2 自定义对象

1. 创建一个对象
2. 向对象中添加属性
3. 查找一个属性
4. 删除一个属性

> 注意: 对象的属性名就是一个字符串

```
    var obj=new Object()
    // var obj={} //两者等价

    obj.a=1
    obj.["1234"]=2

    console.log(obj.a)
    console.log(obj["1234"])

    delete obj.a
```

3.3 检查对象中是否有某个属性

```
    var obj=new Object();
    obj.test=123;
    console.log("test" in obj);
```

3.4 枚举对象的属性

```
for (var k in obj){
    console.log(k) //k是属性名
    console.log(obj[k])//obj[k]是属性值
    console.log(obj.k)//->输出undefined k不是一个属性名
}
```

3.5 对象的比较

1. 当比较两个基本数据类型的值时，就是比较值。
2. 而比较两个引用数据类型时，比较的是内存地址。

```
var obj1=new Object();
obj1.name="abc";

var obj2=obj1
obj2=null //只是把obj2设置为null，obj1记录的地址不变

console.log(obj1) //-> Object
```

四.函数

4.1 创建一个函数对象

1. 使用构造函数方式返回一个函数对象(几乎不用)

```
var fun=new Function("console.log(123)")
```

2. 使用**函数声明**来创建一个函数

```
function fun1(){
    ...
}
```

3. 使用**函数表达式**来创建一个函数

```
// 将一个匿名函数 赋值给变量
var fun2=function(){
    ...
}
```

4.2 函数的参数

1. 函数的实参可以是任意的数据类型，如果有可能需要对参数进行类型的检查。
2. 调用函数时，解析器不会检查实参的数量。
3. 多余的实参不会被赋值。
4. 实参数量少于形参的数量，则没有对应实参的形参将是undefined

4.3 立即执行函数

只会执行一次

```
(function(){
    alert("123");
})()
```

五.作用域

5.1 全局作用域

- 在全局作用域中有一个全局对象window
- 在全局作用域中：

1. 创建的变量会作为window对象的属性保存
2. 创建的函数会作为window对象的方法保存

```
    var a=10;
    function fun(){
        ...
    }
    console.log(window.a)
    window.fun()
    window.alert(123)
```

5.2 变量的声明提前

- 使用var关键字生命的变量，会在所有的代码执行之前被声明(但是不会被赋值)。
- 但是如果声明变量时不使用var关键字，则变量不会被声明提前

```
    console.log(a) -> undefined
    var a=123

    等价于
    var a;
    console.log(a) -> undefined
    a=123
```

但如果

```
    console.log(a) -> undefined //-> ERROR a is not defined
    a=123
```

5.3 函数的声明提前

- 使用函数声明形式创建的函数 `function 函数(){}`,它会在所有的代码执行之前就被创建，所以我们可以在函数声明前来调用函数
- 使用函数表达式创建的函数，不会被声明提前，所以不能在声明前调用。

```
fun1() //能被正常调用
function fun1(){
    ...
}
```

```
fun2() //ERROR -> undefined is not a function
var fun2=function(){//fun2虽然声明被提前，但没有被赋值，不能被调用
    ...
};
```

5.4 局部作用域

- 当在函数作用域操作一个变量时，它会先在自身作用域中寻找，如果有就直接使用；如果没有则向上一级作用域中寻找，直到找到全局作用域。
- 如果在函数中想访问全局作用域可以使用window.xxx
- 在函数中也有声明提前的特性

```
var a=1
function fun1(){
    var a=2
    function fun2(){
        console.log(a);// -> undefined var声明提前
        var a=3;
    }
}
```

- 在函数中,不使用var声明的变量都会成为全局变量

```
var a=1
function fun3(){
    a=10 //a没有使用var关键字，则会设置为全局变量
}
console.log(a)// -> 10
```

- 定义形参就相当于在函数作用域中声明了变量

```
var e=23;
function fun6(e){
    console.log(e);// -> undefined
}
fun6();
```

5.5 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this,this指向的是一个对象，这个对象我们称为函数执行的**上下文对象**。

1. 以**函数**的形式调用时,this永远都是window。

2. 以**方法**的形式调用时，this就是调用方法的那个对象。
3. 当以构造函数的形式调用时，this就是新创建的那个对象。

```
var name='全局作用域的name';
function fun(){
    console.log(this.name);
}
var obj={
    name:"zhangsan";
    sayName:fun
}

fun() // -> '全局作用域的name' 即this=window
obj.sayName() // -> 'zhangsan' 即this=obj
```

5.6 使用工厂方法创建对象

```
function createPerson(name,age){
    // 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
    // 将新的对象返回
    return obj
}

var obj1=createPerson('zhangsan',18)
```

5.7 构造函数

- 构造函数和普通函数的区别就是调用方式的不同
- 普通函数就是直接调用，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(this)作为返回值返回
- 使用同一个构造函数创建的对象，我们称为一类对象，也将一个构造函数称为一个类。
- 我们将通过一个构造函数创建的对象，称为是该类的实例

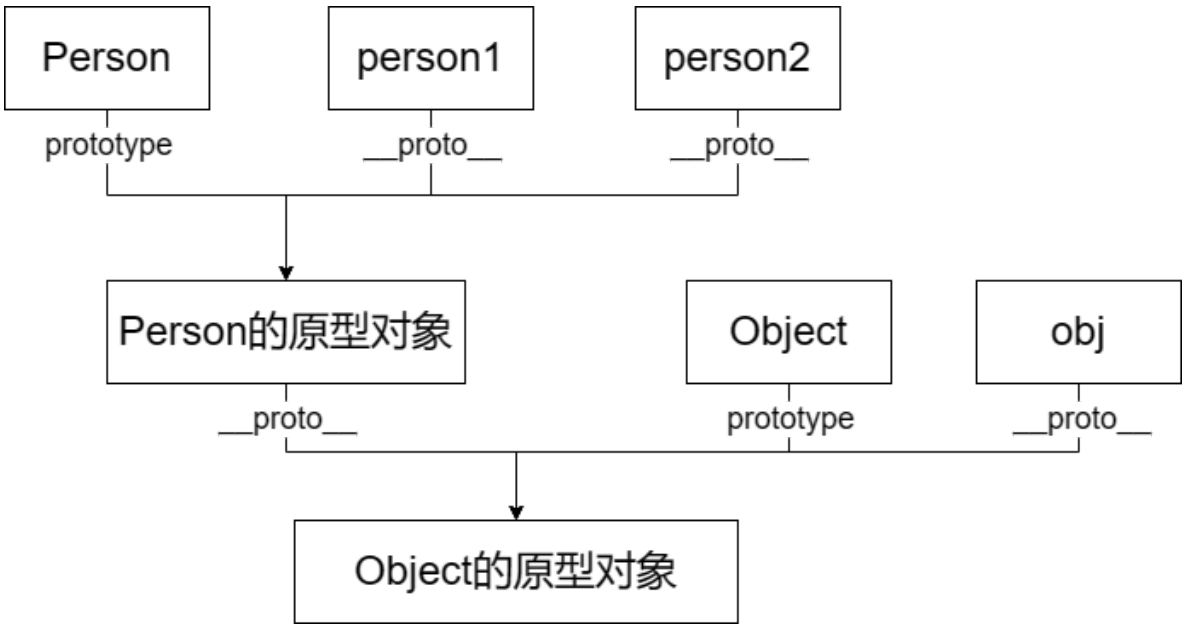
```
function Person(name,age){
    this.name=name
    this.age=age
}

var person=new Person()
console.log(person instanceof Person)
```

5.8 原型

5.8.1 prototype和proto

- 我们所创建的每一个函数，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。
- 如果函数作为普通函数调用`prototype`没有任何作用
- 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，我们可以通过`__proto__`来访问该属性。
- 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，我们可以将对象中共有的内容，统一设置到原型对象中。
- 当我们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。
- 直到找到Object的原型对象




```
// prototype 和 __proto__
function Person(name,age){
    this.name=name
    this.age=age
}

// 向Person的原型中添加一个方法
```

```
Person.prototype.sayName=function(){
    console.log(this.name)
}

// 创建一个实例
var p1=new Person('zhangsan',18)
var p2=new Person('lisi',22)

// 调用实例的方法，会在person的__proto__中寻找（即Person的prototype）
p1.sayName()
p2.sayName()

console.log(p1.__proto__ == Person.prototype) // -> True
```

> `Person`的`prototype`和`person`的`__proto__`均指向**Person的原型对象**
>
> 原型对象中保存一些公共的属性和方法。

5.8.2 原型对象案例

```
function MyClass(){

}
MyClass.prototype.name="我是原型对象中的name";
var mc=new MyClass();

// 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
console.log('name' in MyClass)//-> true
console.log('name' in mc)//-> true

// 可以使用对象的hasOwnProperty()来检查对象自身是否含有该属性
console.log(mc.hasOwnProperty('name')); // -> false
console.log(mc.__proto__.hasOwnProperty('name'));// -> true 原型中含有name属性

// 判断hasOwnProperty方法的位置
// hasOwnProperty在Object的原型上 | Person的原型的原型上
console.log(mc.__proto__.hasOwnProperty('hasOwnProperty')); // -> false
console.log(mc.__proto__.__proto__.hasOwnProperty('hasOwnProperty'));// -> true

// Person的原型的原型的原型为null
console.log(mc.__proto__.__proto__.__proto__); // -> null
```

5.8.3 toString()

- 当我们直接在页面上打印一个对象时，事件上输出的是Object原型对象的toString()方法的返回值。
- 可以通过在原型对象上添加toString方法，来改变输出值。

```
function Person(name,age){
    this.name=name
    this.age=age
}

Person.prototype.toString=function(){
    return "Person[name="+this.age+",age="+this.age+"]"
}

var p1=new Person('zhangsan',18)
console.log(p1)
```

5.9 垃圾回收GC

- 当对象没有引用时就会自动进行垃圾回收

```
var obj=new Object()

obj=null //obj会进行垃圾回收
```

六.数组

- 数组是一个内建对象
- 数组使用数字作为索引操作元素(索引为属性)
- 数组存储效率高

6.1 数组的增删改查

1. 创建一个数组
2. 在数组中添加一个元素
3. 查看数组的长度
4. 修改数组的长度
5. 在末尾添加一个元素

```
// 1.创建一个数组
var arr=new Array();

// 2.在数组中添加元素
arr[1]=1 //[undefined,1]
arr[2]=2 //[undefined,1,2]

// 3.查看数组的长度
console.log(arr.length) // -> length=3

// 4.修改数组的长度
arr.length=5 //[undefined,1,2,undefined,undefined]
```

```
arr.length=2 //[undefined,1]

// 5.在末尾添加一个元素
arr[arr.length]=3 //[undefined,1,3]
```

6.2 数组字面量

1. 使用字面量创建数组
2. 使用构造函数函数创造数组

```
var arr1=[10,20,30,40,50]
var arr2=new Array(10,20,30,40,50)

var arr3=new Array(3) // [undefined,undefined,undefined]
```

6.3 数组的四个增删方法

1. push(a,b,c)

- 向数组最后添加一个或多个元素
- 返回添加后的长度

```
var arr=[1,2,3]
console.log(arr.push(4,5,6,7)) // -> 7
```

2. pop()

- 向数组最后删除一个元素
- 返回被删除的元素

```
var arr=[1,2,3]
console.log(arr.pop()) // -> 3
```

3. unshift(a,b,c)

- 向数组开头添加一个或多个元素
- 返回添加后的长度

```
var arr=[2,3,4]
console.log(arr.unshift(0,1)) // -> 5
// arr [0,1,2,3,4]
```

4. shift()

- 向数组开头删除一个元素
- 返回被删除的元素

```
var arr=[1,2,3]
console.log(arr.shift()) // -> 1
```

6.4 数组的遍历

```
var arr=[1,2,3,4,5]
for(var i=0;i<arr.length;i++){
    console.log(arr[i])
}
```

6.5 forEach(func(value,index,obj))

- forEach()方法需要一个函数作为参数
- 像这种函数，有我们创建但不由我们调用的，我们称为回调函数
- 数组中有几个函数就会执行几次，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
- 浏览器会在回调函数中传递三个参数
    1. 第一个参数value，数组元素
    2. 第二个参数index，数组索引
    3. 第三个参数obj，数组

```
var arr=['x','y','z']

arr.forEach((value,index,obj) => {
    console.log(value);
    console.log(index);
    console.log(obj);
})
```

6.6 slice(start,end)

- 数组切片
- 与python [start:end] 相同
- 参数

1. start(必需):截取开始位置的索引
2. end(可选):截取结束位置的索引,可选，可传负值

- [start,end)左闭右开
- 不写end相当于 [start:]

6.7 splice(start,length,insert_items,····) -> 影响自身

- 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：
    1. 第一个参数(必需)：表示开始位置的索引
    2. 第二个参数(可选)：表示删除的个数
    3. 第三个参数及以后(可选)：将自动插入到start前的位置
- 有多种使用功能

1. 一个参数:删除包括start后的所有元素

```
var arr=[0,1,2,3,4,5]
arr.splice(2) // 删除了[2,3,4,5]
console.log(arr) // 剩余[0,1]
```

2. 两个参数:删除包括start后的length个元素

```
var arr=[0,1,2,3,4,5]
arr.splice(2,2) // 删除了[2,3]
console.log(arr) // 剩余[0,1,4,5]
```

3. 三个参数:删除包括start后的length个元素，并在start处添加元素

```
var arr=[0,1,2,3,4,5]
arr.splice(2,2,7,8) // 删除了[2,3]
console.log(arr) // 剩余[0,1,7,8,4,5]
```

4. 删除某些元素

```
var arr=[0,1,2,3,4,5]
arr.splice(2,2) // 删除了[2,3]
console.log(arr) // 剩余[0,1,4,5]
```

5. 添加某些元素(删除0个元素)

```
var arr=[0,1,2,3,4,5]
arr.splice(2,0,7,8) // 不删除元素
console.log(arr) // [0,1,7,8,2,3,4,5]
```

6. 替换某些元素

```
var arr=[0,1,2,3,4,5]
arr.splice(2,2,7,8) // 删除了[2,3] 添加了[7,8]
console.log(arr) // 剩余[0,1,7,8,4,5]
```

6.8 concat(arr2，arr3,element1,element2…)

- 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 arr1=[0,1,2]
var arr2=[3]
var arr3=[4,5]
console.log(arr1.concat(arr2,arr3,6,7,8)) //[0,1,2,3,4,5,6,7,8]
```

6.9 join(separator)

- 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转作为结果返回
- 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，则默认使用，作为连接符

```
var arr=[1,2,3]
console.log(arr.join())// -> "1,2,3"
console.log(arr.join(""))// -> "123"
console.log(arr.join("-"))// -> "1-2-3"
```

6.10 reverse() -> 影响自身

- 该方法用来反转数组
- 该方法会直接修改原数组

```
var arr=[1,2,3]
arr.reverse()
console.log(arr)//-> [3,2,1]
```

6.11 sort() -> 影响自身

- 可以用来对数组中的元素进行排序
- 影响原数组，默认会按照Unicode编码进行排序
- 对数字排序时，可能会得到错误的结果。
- 可以在sort()添加一个回调函数，来制定排序规则

> 回调函数中需要定义两个形参
>
> 浏览器将会使用数组中的元素作为实参去调用回调函数
>
> 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前面
>
> - 浏览器会根据回调函数的返回值来决定元素的顺序
>
> 1. 如果返回一个大于0的值，则元素会交换位置
> 2. 如果返回一个小于0的值，则元素位置不变
> 3. 如果返回一个0，则认为两个元素相等，也不交换位置

> - 如果需要升序排序，则返回a-b
> - 如果需要降序排序，则返回b-a

```
// 对数字排序时，可能会得到错误的结果。
var arr=[5,11,6]
arr.sort()
console.log(arr) // -> [11,5,6] 根据Unicode编码
```

```
// 对数字升序排序方法
var arr=[5,11,6]
arr.sort(function(a,b){
    if(a<b)return -1 //a一定在b左面 如果a<b 则不交换位置 即默认升序排序
    if(a>b)return 1
    return 0
})
console.log(arr) // -> [5，6，11]
```

```
// 对数字升序简写方式
var arr=[5,11,6]
arr.sort((a,b)=>a-b)
console.log(arr) // -> [5，6，11]

// 对数字降序简写方式
var arr=[5,11,6]
arr.sort((a,b)=>b-a)
console.log(arr) // -> [11，6，5]
```

七.函数相关

7.1 函数对象的方法call(obj,a,b…)和apply(obj,[a,b，…])

- 这两个方法都是函数对象的方法，需要通过函数对象来调用
- 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

区别

- call()方法可以将实参在对象之后一次传递
- apply()方法需要将实参封装到一个数组中统一传递

> **总结：this的情况：**
>
> 1. 以函数形式调用时，this永远都是window
> 2. 以方法的形式调用时，this是调用方法的对象(实例)
> 3. 以构造函数的形式调用时，this是新创建的那个对象
> 4. 使用call和apply调用时，this是指定的那个对象

```
function fun(){
    console.log(this);
}

var obj1={}

fun() // this -> window

// 当对函数调用call()和apply()都会调用函数执行
fun.call() // this -> window

// 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
fun.call(obj1) // this -> obj1
fun.apply(obj1) // this -> obj1
```

```
function fun(a,b){
    console.log(this.name);
    console.log(a);
    console.log(b);
}

var obj1={name:'zhangsan'}
var obj2={name:'lisi'}

// call()方法可以将实参在对象之后一次传递
// apply()方法需要将实参封装到一个数组中统一传递

fun.call(obj1,2,3) // zhangsan 2 3
fun.apply(obj2,[4,5]) // lisi 4 5
```

7.2 arguments

在调用函数时，浏览器每次都会传递进两个隐含的参数：

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（不是数组），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，所传递的实参都会在arguments中保存
- arguments.length可以用来获取实参的长度
- 我们即使不定义形参，也可以通过arguments来使用实参
- 例如:arguments[0] 表示第一个实参
- arguments[1] 表示第二个实参
- arguments中有一个属性叫做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。

```
function fun(){
    console.log(typeof arguments) // object
    console.log(arguments instanceof Array) //false
    console.log(arguments.length) // 2
    console.log(arguments[0]) // 11
    console.log(arguments[1]) // 22
    console.log(arguments.callee==fun) // true
}
fun(11,22)
```

7.3 Date对象

```
var d=new Date()
var d2=new Date("12/03/2016 11:10:30")

d.getDate()
d.getDay()
d.getMonth()
d.getFullYear()
d.getTime() // 返回1970年1月1日至今的毫秒数（时间戳）

var start=Date.now() //（时间戳）
//do something
var end=Date.now() //（时间戳）
```

7.4 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，它里面封装了数学运算相关的属性和方法

```
Math.random() //随机生成一个0到1的小数
Math.floor(Math.random()*(y-x+1)+x) //随机生成一个x到y的整数
```

7.5 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()

- Number()
- Boolean()

> 注意：
>
> 在实际应用中不会使用基本数据类型的对象，如果使用基本数据类型的对象，在做一些比较时可能会带来一些不可预期的结果

```
var num=new Number(3);
var num2=new Number(3);
var str=new String('hello');
var bool=new Boolean(true);
var bool2=true;

//类型为Object
console.log(typeof num) //Object

//向num中添加一个属性
num.hello="abcdefg";

//进行比较
console.log(num==num2) // -> false 对象比较地址
console.log(bool==bool2) // -> true bool进行了类型转换
console.log(bool===bool2)// -> false 类型不同
```

- 方法和属性能添加给对象，不能添加给基本数据类型
- 当我们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。

```
var s=123;
s=s.toString();//s是一个基本数据类型Number，没有toString方法，但没有报错，因为会临时使用包装类。Number包装类中有toString方法。

console.log(s) //'123'
console.log(typeof s) //String

s.abc=128 //s会转为Number包装类，添加属性，再转回Number
console.log(s.abc) //undefined
```

7.6 String的方法

1. charAt(idx)
    - 可以返回字符串中指定位置的字符
    - 根据索引获取指定的字符
2. charCodeAt(idx)
    - 获取指定位置字符的字符编码(Unicode编码)
3. String.fromCharCode(unicode)

- 可以根据字符编码去获取字符

4. concat(str1,str2····)
   - 可以用来连接两个或多个字符串
   - 作用和+一样
5. indexOf(ch,startidx)
   - 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
6. lastIndexOf(ch,startidx)
   - 该方法的用法和indexOf()一样，不同的是lastIndexOf是从后往前找
   - 也可以指定开始查找的位置
7. slice(start,end)
   - 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
   - 第一个参数(必须)：开始位置的索引（包括开始的位置）
   - 第二个参数(可选):结束位置的索引（不包括结束位置）
8. substring(start,end)
   - 可以用来截取一个字符串，与slice()类似
   - 参数：
   - 第一个参数（必须）：开始截取位置的索引（包括开始的位置）
   - 第二个参数（可选）：结束位置的索引（不包括结束位置）

> 与slice的区别：
>
> 1. 第二个参数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
> 2. 如果第二个参数小于第一个参数，会自动调整参数的位置，会调换两参数的位置

9. substr()
   - 用来截取字符串(不推荐使用这个方法)
   - 参数：
   - 1. 截取开始位置的索引
   - 2. 截取的长度
10. split()
    - 可以将一个字符串拆分为一个数组
    - 参数
    - 需要一个字符串作为参数，将会根据该字符串去拆分数组
11. toUpperCase()

- 将一个字符串转换为大写并返回

12. toLowerCase()
    - 将一个字符串转换为小写并返回

八.正则

8.1 正则表达式的简介

- 创建正则表达式的对象
- 语法：
- var 变量= new RegExp("正则表达式"，"匹配模式");
- 匹配模式 `i`(ignore):不区分大小写,`g`(global):开启全局模式(匹配一个->所有)
- - 可以为一个正则表达式设置多个匹配模式，且顺序无所谓。
- 使用typeof检查正则对象，会返回object

正则表达式的方法test()

- 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

```
// 这个正则表达式可以来检查一个字符串中是否含有a
var reg= new RegExp("a")
var str="bbacd"

console.log(reg.test(str)) // -> true

// 忽略大小写
var reg1= new RegExp("a","i")
var str="bbAcd"

console.log(reg.test(str)) // -> true
```

8.2 正则语法

8.2.1 使用字面量创建正则表达式对象

- 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
- 使用字面量的方式创建更加简单
- 使用构造函数创建更加灵活(可以传参)

```
var reg=new RegExp('a','i')
var reg1=/a/i ;
```

8.2.2 正则表达式 或

- 创建一个正则表达式，检查一个字符串中是否有a或b
- 使用|表示或者的意思
- []里的内容也是或的关系
- /[ab]/== /a|b/

```
var reg=/a|b|c/
var reg=/[abc]/
```

- /[a-z]/ 所有小写字母
- /[A-Z]/ 所有大写字母
- /[A-z]/ 所有字母
- /[0-9]/ 所有数字

```
// 检查一个字符串中是否有 abc 或 adc 或 aec
var reg=/a[bde]c/
```

8.2.3 正则表达式 除了

- /[^ab]/ 除了ab是否还有其他字符

8.2.4 正则表达式 以开始结束

- 检查一个字符串是否以a开头
- ^ 表示开头 /^a/
- $ 表示结尾 /a$/
- 同时开头结尾 /^a$/

检查是否是一个手机号

1. 以1开头
2. 第二位3-9任意数字
3. 三位后任意数字9个

```
var reg=/^1[3-9][0-9]{9}$/

var phoneStr='12345678901'
console.log
```

8.3 支持正则表达式的String对象方法

8.3.1 split(re)

- split(正则表达式)
- 这个方法即使不指定**全局匹配**，也会全部插分

```
var str="1a2b3c4d"
console.log(str.split(/[a-z]/))
```

8.3.2 search(re)

- 可以搜索字符串中是否含有指定内容
- 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
- 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
- search()只会查找第一个，即使设置**全局匹配**也没用

```
var str="hello abc hello aec afc"
console.log(str.search(/a[bef]c/))
```

8.3.3 match(re)

- 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
- 默认情况下我们的match只会找到第一个符合要求的内容，找到以后就停止检索
- 我们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的内容
- match()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

```
var str='1a2b3c4d5e6f7A8B9C'
console.log(str.match(/[a-z]/ig)) //[a,b,c,d,e,f,A,B,C] 忽略大小写且全局匹配
```

8.3.4 replace(re,newstr)

- 可以将字符串中指定内容替换为新的内容返回
- 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
- 默认只会替换第一个

```
var str='1a2b3c4d5e6f'
console.log(str.replace(/[a-z]/,"@")) // - > '1@2b3c4d5e6f'
console.log(str.replace(/[a-z]/g,"@")) // - > '1@2@3@4@5@6@'
```

8.4 量词

量词

- 通过量词可以设置一个内容出现的次数
- 量词只对它前边的一个内容起作用
- {n} 正好出现n次 /ab{3}c/
- {m,n} 出现m到n次
- {m,} 出现m次以上
- 至少一个，相当于{1,} /ab+c/
- 0个或多个，相当于{0,}
- ? 0个或1个，相当于{0,1}

```
var reg=/b{3}/
var str="abbbbc" //true 含有连续的三个b
console.log(reg.test(str))
```

8.5 元字符

8.5.1 通配符 `.`

- `.`表示任意字符
- 在正则表达式中\作为转义字符
    1. \. 来表示.
    2. \\来表示\
- 在构造函数中，由于它的参数是一个字符串，而\是字符串中的转义字符，如果要使用\则需要使用\\来代替

判断一个字符串中是否有点

```
//匹配一个点
var reg=/\./
var reg=RegExp('\\.') // 字符串先转为 -> /\./ 与正则表达式一致

//匹配一个\
var reg1=/\\/
var reg1=RegExp('\\\\') // 字符串先转为 -> /\\/ 与正则表达式一致
```

8.5.2 其他元字符

- \w 任意字母、数字、_ <=> [A-z0-9_]
- \W 除了字母、数字、_ <=> [^A-z0-9_]
- \d 任意的数字 [0-9]
- \D 除了数字 [^0-9]
- \s 空格
- \S 除了空格

- \b 单词边界
- \B 除了单词边界

单词边界演示

```
var reg=/\bchild\b/
console.log(reg.test("hello children")) // false
console.log(reg.test("hello child")) // true
```

8.6 例子

8.6.1 去除前后的空格，不去除中间的

```
var str='   he   llo     '
var reg=/^\s* | \s*$/g //需要开全局模式 否则只会替换一处
console.log(str.replace(reg,""))
```

8.6.2 电子邮件正则

```
/*
    电子邮件
    hello  . nihao  @  abc .com .cn
    任意字母数字下划线(3个以上) . 任意字母数字下划线 @ 任意字母数字 . 任意字母(2-5次) . 任意字母
(2-5次 )
    ^ \w{3,} (\.\w+)* @ [A-zO-9]+ (\.[A-z]{2,5}){1,2} $
*/

var reg=/^\w{3,}(\.\w+)*@[A-zO-9]+(\.[A-z]{2,5}){1,2}$/ //别忘了开始结尾
var email='hello.nihao@abc.com.cn'
console.log(reg.test(email))
```

九.DOM

9.1 DOM

- DOM，全称Document Object Model文档对象模型
- JS通过DOM来对HTML文档进行操作
- 文档： 整个HTML网页文档
- 对象： 每一个节点就是一个对象
- 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

9.2 节点

- 节点是构成网页最基本的组成部分，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、整个文档都是一个节点。

- 标签称为元素节点，属性称为属性节点，文本称为文本节点，文档称为文档节点。
  节点的属性

1. nodeName
2. nodeType
3. nodeValue

9.3 获取元素节点

通过document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

9.4 获取元素节点的子节点

通过具体的元素节点调用

1. getElementsByTagName() 返回当前节点的指定标签名后代节点
2. childNodes 当前节点的所有子节点
3. firstChild 当前节点的第一个子节点
4. lastChild 当前节点的最后一个子节点

9.5 获取元素的子元素

1. children 获取当前元素的所有子元素
2. firstElementChild 获取当前元素的第一个子元素
3. lastElementChild 获取当前元素的最后一个子元素

9.6 获取父节点和兄弟节点

通过具体的节点调用

1. parentNode 表示当前节点的父节点
2. previousSibling 表示当前节点的前一个兄弟节点
3. nextSibling 表示当前节点的后一个兄弟节点

9.7 获取兄弟元素

1. previousElementSibling 获取上一个兄弟元素
2. nextElementSibling 获取下一个兄弟元素

9.8 获取子元素和子节点的区别

- 子元素就是一个html标签，但子节点可能包括文本
- 换行和空格会导致子节点中出现文本节点

```
<ul id="uul">
        <li>a</li>
        <li>b</li>
        <li>c</li>
</ul>
<script>
    var uul=document.getElementById('uul');
    // 获取ul的第一个子节点为#Text
    // #Text的值为空（因为换行导致了多出空文本）
    console.log(uul.firstChild.nodeValue);
</script>
```

9.9 innerHTML和innerText

- 元素节点调用会返回内在的HTML或Text
- 文本节点调用会返回undefined

```
<ul id="uul">
        <li>a</li>
        <li>b</li>
        <li>c</li>
</ul>
<script>
    var uul=document.getElementById('uul');
    console.log(uul.innerHTML);
    /* ->
        <li>a</li>
        <li>b</li>
        <li>c</li>
    */
    console.log(uul.innerText);
    /* ->
        a
        b
        c
    */
</script>
```

9.10 获取某些元素

- 获取body元素

1. document.getElementsByTagName('body')[0]
2. document.body

- 获取html标签

1. html = document.documentElement

- 获取所有元素

1. all = document.all
2. all = document.getElementByTagName("*")

- 根据class获取节点对象

1. document.getElementsByClassName("box1")

- 根据css选择器选择一个元素节点(只会返回唯一的元素，只会返回第一个)

1. document.querySelector(".box1 div")

- 根据css选择器选择所有元素节点(只会返回唯一的元素，只会返回第一个)

1. document.querySelectorAll('.box1')

9.10 DOM增删改