# 01 JS 高 值转换 类型转换 ES5数组

js

###概述

ECMAScript高级的三座大山

1.作用域&作用域链(衍生运用)

2.原型&原型链　(衍生运用)

3.异步&单线程　(衍生运用)

## 1.javascript性能问题

javascript本身就只是一个非常不起眼的语言，用来处理非常小众的问题。所以从设计之初它的性能就没有被重点考虑过。由于web时代的来临和Html5的兴起，这一切让javascript前所未有地成为焦点。自然它的功能和性能在大家的努力下都有了长足的进步。

一方面javascript是解释性语言，运行时编译。另一方面，它是一种无类型语言（动态语言）。相比较而言，静态语言C++ Java。他们在编译阶段就能够知道变量的类型。但是javascript语言的动态性没有办法在编译的时候就知道变量的类型。只能在运行时才能确定。在运算时计算和决定类型导致了javascript语言的运行效率比静态语言C++ Java都要低的很多。

java:

```
class Object{
    int x;
    int y;
}
int add(Object obj1,Object obj2){
    return obj1.x*obj1.y + obj2.x*obj2.y;
}
```

在编译(.java --> .class(JVM上运行！))阶段，编译器发现有两个对象obj1,obj2.

在内存中分配地址

ooo --->obj1　　xxx --->obj2

ooo为obj1的基地址　　xxx为obj2的基地址

int类型占据四个字节　　int类型占据四个字节

x---> ooo往后推四个字节　　x---> xxx往后推四个字节

y---> x的地址值再往后四个字节　　y---> x的地址值再往后四个字节

在C++ Java等语言中，已事先知道存取的成员变量(属性)的类型，所以语言解释系统只要利用数组和位移来存取这些变量和方法的地址。位移信息使它只要几个机器语言指令，就可以存取变量，找出变量或执行其他任务。

js:

```
function add(obj1,obj2){
    return obj1.x*obj1.y + obj2.x*obj2.y;
}
```

对于传统的javascript解释器，所有这一切都是解释执行。所以效率不会高到哪去我们也将javascript代码的处理分成两个阶段。

---编译阶段。(做不了太多事情，javascript的编译一般就是在执行前的几微秒内进行)

---执行阶段。

对于javascript引擎来说，因为没有C++或者java这样的强类型语言的类型信息。所以javascript引擎的通常做法就是在分配基地址后使用键值对的形式来存储每一个对象。(典型的做法就是采用字符串)。之后访问对象的属性值时需要通过属性名的匹配来获取相应的值。

(这一切都是在执行阶段处理的)

归根结底,js性能的瓶颈在于如何获取对象属性值的具体位置?

---相对于对象基地址的偏移?　　//拿不到

---分配基地址后通过匹配属性名查找属性值?　　//不能用基地址去找，匹配属性名查找.

JS中所有的传递都是值传递。　值传递断一根线。

栈：　先进后出。　出后只有一个

队列：先进先出。

基本数据类型存在栈中，容量有限。（变量存在栈中 ）

引用类型：存在堆中。

值是最核心的。　（存储和获取值，最核心的功能）

# 2 . 值,数据类型,变量(ES5)

a.值;计算机程序的运行需要对值进行操作,在javascript中值被分成了两大类:基本数据类型,引用数据类型

b.数据类型;数据类型是值的内部特征,它定义了值的行为,以使其区别于其他的值。

| 基本数据类型 | (对应的值) |
| --- | --- |
| 空 | null |
| 未定义 | undefined |
| 布尔类型 | true/false |
| 数字 | 1,2,3.... |
| 字符串 | "1","2","3",'a','b','c'.... |
| 引用数据类型 | (对应的值) |
| 对象 | 数组,函数,{}.... |

我们可以用 typeof 运算符来查看值的类型

```
typeof undefined     === "undefined";       // true
typeof true          === "boolean";         // true
typeof 42            === "number";          // true
typeof "42"          === "string";          // true
typeof { life: 42 } === "object";          // true
```

```
/*基本数据类性*/
console.log(typeof undefined  === "undefined")// true
console.log(typeof null   === "object") // true
console.log(typeof true    === "boolean") // true
console.log(typeof 42     === "number") // true
console.log(typeof "42"    === "string") // true
```

```
/*引用数据类型*/
console.log(typeof {}    === "object") // true
console.log(typeof []   === "object") // true
console.log(typeof function(){} === "function") // true
```

函数是javascript中的一等公民

以上五种类型均有同名的字符串值与之对应,你可能注意到 null 类型不在此列。它比较 特殊， typeof 对它的处理有问题:

1.typeof null === "object"; // true

正确的返回结果应该是 "null" ，但这个 bug 由来已久，在 javascript 中已经存在了将近二十年，

也许永远也不会修复，因为这牵涉到太多的 Web 系统，修复 它会 产生 更多的 bug ，令许多系统无法正常工作

2.typeof function a(){ /* .. */ } === "function"; // true

function是 object 的一个 子类型 。具体来说，函数是 可调用对象

函数是javascript中的一等公民

3.typeof 安全机制 (讲述左右查询时再来理解)

typeof undefined === "undefined"; // true

typeof 未定义的变量 === "undefined"; // true

c.变量;当程序需要将值保存起来以备将来使用时,便将其赋值给一个变量。变量是一个值的符号名称。

可以通过名称来获得对值的引用。

d.注意点!

----取得并使用值是所有程序设计中的要点。在javascript中变量是没有类型的，只有值才持有类型。！！

也就是说javascript不做类型强制语言引擎不要求变量总是持有与其初始值同类型的值。一个变 量可以现在被赋值为字符串类型值，随后又被赋值为数字类型值。这是javascript作为动态语言的一个优势。----在对变量执行 typeof 操作时，得到的结果并不是该变量的类型，而是该变量持有的值的类型，因为 JavaScript 中的变量没有类型。

e.值传递&引用传递

**变量**是一个值的符号名称。值;计算机程序的运行需要对值进行操作,在javascript中值被分成了两大类:基本数据类型,引用数据类型

变量是没有类型的，只有值才持有类型。！！

基本数据类型总是通过值复制的方法来赋值/传递。

引用数据类型总是通过引用复制来完成赋值/传递。

JS中所有的传递都是值传递 ，没有引用传递。

注意：引用指向的是值而非变量，所以一个引用无法更改另一个引用的指向，

但是可以更改不同变量共同指向的值!

```
var a=3;
function test(a){
 a=a+1;
```

```
}
test(a);
console.log(a);//3


    function foo(wrapper){
 wrapper.a=2;
 console.log(wrapper.a);//2
}
var wrapper={
 a:1
}
foo(wrapper);
console.log(wrapper.a);//2
```

两种 数据类型：

1.基本数据类型： number , string , null , boolean undefined;

2.引用数据类型：Object , 数组 ， function ; 所有开头字母大写Number ， String , Array ,都是引用类型。因为是 new 出来的。 他们的typeof都是object.

## 3 . 内存与垃圾回收机制

JavaScript 具有自动垃圾收集机制，也就是说，执行环境会负责管理代码执行过程中使用的内存。
而在 C 和 C++之类的语言中，开发人员的一项基本任务就是手工跟踪内存的使用情况，这是造成许多问题的一个根源。在编写 JavaScript 程序时，开发人员不用再关心内存使用问题，所需内存的分配以及无用内存的回收完全实现了自动管理。这种垃圾收集机制的原理其实很简单：找出那些不再继续使用的变量，然后释放其占用的内存。为此，垃圾收集器会按照固定的时间间隔周期性地执行这一操作。
下面我们来分析一下函数中局部变量的正常生命周期。局部变量只在函数执行的过程中存在。而在这个过程中，会为局部变量在栈（或堆）内存上分配相应的空间，以便存储它们的值。然后在函数中使用这些变量，直至函数执行结束。此时，局部变量就没有存在的必要了，因此可以释放它们的内存以供将来使用。在这种情况下，很容易判断变量是否还有存在的必要；但并非所有情况下都这么容易就能得出结论。垃圾收集器必须跟踪哪个变量有用哪个变量没用，对于不再有用的变量打上标记，以备将来收回其占用的内存。用于标识无用变量的策略可能会因实现而异，但具体到浏览器中的实现，则通常有两个策略。标记清除，引用计数。

###全局变量
关闭浏览器页面的时候才会释放

###区分变量的释放 与 垃圾的回收
变量释放是瞬间的
垃圾回收是每隔一段时间进行侦测的，它会回收缓于变量释放

总结：
函数中的变量释放，在他调用后就被释放。（变量释放是瞬间的，垃圾回收是周期性的。）
垃圾回收：标记清除：立flag
引用计数: 引用几次，引用为0时，垃圾回收。

## 4.引用数据类型(数组)

### 1.数组(ES5)

**a.如何创建一个数组**

和其他强类型语言不同，在 JavaScript中，数组可以容纳任何类型的值，可以是字符串、数字、对象（ object ），甚至是其他数组（多维数组就是通过这种方式来实现的）

```
var arrayObj = new Array();  //创建一个数组
var arrayObj = new Array([size]);  //创建一个数组并指定长度，注意不是上限，是长度
var arrayObj = new Array([element0[, element1[, ...[, elementN]]]]);  //创建一个数组并赋值
```

Array()

**Array(1,2,3) 和 new Array(1,2,3)效果一样,即Array的普通调用效果等于构造调用。**

我们可以粗暴的认为Array就是构造函数

Array 构造函数只带一个数字参数的时候，该参数会被作为数组的预设长度（ length ），而非只充当数组中的一个元素。

这实非明智之举：一是容易忘记，二是容易出错。

更为关键的是，数组并没有预设长度这个概念。这样创建出来的只是一个空数组，只不过它的 length 属性被设置成了指定的值。

如若一个数组没有任何单元，但它的 length 属性中却显示有单元数量，这样奇特的数据结构会导致一些怪异的行为,我们将包含至少一个 “ 空单元 ” 的数组称为 “ 稀疏数组 ” .

**b.ES5中关于数组的api**

**c.伪数组**

具有length属性,下标为数字的javascript对象 我们一般称之为伪数组

如何将伪数组转为真数组 （面试）

var arr = Array.prototype.slice.call( 伪数组 );

/ /找最干净的那个slice。

**d.逗号问题**

从 ES5 规范开始就允许在列表（数组值、属性列表等）末尾多加一个逗号（在实际处理中会被忽略不计）。

所以如果你在代码或者调试控制台中输入 [ , , , ] ，实际得到的是 [ , , ] （包含三个空单元的数组）返回的数字是3个空 ---> [empty × 3]

总结：

**1. isArray(obj)**
**2.forEach()**
**3.every()**
**4.some()**
**5.filter()**
**6. sort()**
**7.map()**
**8.slice()**
**9.splice()**
**10.reduce()**

1. **.isArray(obj)** 用于确定传递的值是否是一个 Array

```
    //obj：需要检测的值。
//返回值：如果对象是 Array，则为true; 否则为false。
//是否影响老数组：不影响
//Array.isArray(obj) 用于确定传递的值是否是一个 Array  //控制台
Array.isArray([1, 2, 3]);  // true
```

2. **forEach()** 方法对数组的每个元素执行一次提供的函数。 遍历

```
        //参数1:callback
 /*为数组中每个元素执行的函数，该函数接收三个参数：
  currentValue(当前值):数组中正在处理的当前元素。
  index(索引):数组中正在处理的当前元素的索引。
  array:forEach()方法正在操作的数组。*/
//参数2:thisArg
 /*可选参数。当执行回调 函数时用作this的值。*/

//返回值：undefined

//是否影响老数组：不影响老数组
```

3. **every()** 方法测试数组的**所有**元素是否都通过了指定函数的测试.

```
      //参数1:callback
 /*为数组中每个元素执行的函数，该函数接收三个参数：
  currentValue(当前值):
  index(索引):
  array:forEach()方法正在操作的数组。*/
//参数2:thisArg
 /*可选参数。当执行回调 函数时用作this的值。*/

//返回值：true false

//是否影响老数组：不影响
          eg:var arr =[1,232,3,4,5,6,7,8];
var result = arr.every(function(item,index){
 if(item<9){
```

```
  return true;
 }
})
console.log(result);
```

4. **some()** 方法测试数组中的**某些**元素是否通过由提供的函数实现的测试。

```
        //参数1:callback
 /*为数组中每个元素执行的函数，该函数接收三个参数：
  currentValue(当前值)。
  index(索引):。
  array:forEach()方法正在操作的数组。*/
//参数2:thisArg
 /*可选参数。当执行回调 函数时用作this的值。*/
//返回值：true false
//是否影响老数组：不影响
```

5. **filter()** 方法创建一个新数组, 其包含通过所提供函数实现的测试的所有元素。

```
          var arr =[1,2,3,4,5,6,7,8];
var result = arr.filter(function(item,index){
 if(item > 7){
  return true;
 }
})
console.log(result);

//参数1:callback
 /*为数组中每个元素执行的函数，该函数接收三个参数：
  currentValue(当前值):
  index(索引):
  array:forEach()方法正在操作的数组。*/
//参数2:thisArg
 /*可选参数。当执行回调 函数时用作this的值。*/
//返回值：  一个新的通过测试的元素的集合的数组
//是否影响老数组：不影响
```

6. **sort()** 方法使用in-place算法对数组的元素进行排序，并返回数组。默认排序顺序是根据字符串Unicode码点。

```
            var arr =[1,25,34,14,26,16,47,28];
var result = arr.sort(function(a,b){
 //从小到大
 return a-b;
 //从大到小
//   return b-a;
})
console.log(result);
console.log(arr);

//参数compareFunction 可选。
 /*
  用来指定按某种顺序进行排列的函数。如果省略，
  元素按照转换为的字符串的各个字符的Unicode位点进行排序。
 */

//返回值：返回排序后的数组。

//是否影响老数组： 影响 原数组已经被排序后的数组代替。
```

7. **map()** 方法创建一个新数组，其结果是该数组中的每个元素都调用一个提供的函数后返回的结果。

```
        //参数1:callback
 /*为数组中每个元素执行的函数，该函数接收三个参数：
  currentValue(当前值):
  index(索引):
  array:forEach()方法正在操作的数组。*/
//参数2:thisArg
 /*可选参数。当执行回调 函数时用作this的值。*/
//返回值：一个新数组，每个元素都是回调函数的结果。
//是否影响老数组：不影响.
          eg://格式化数组中的对象
// reformattedArray 数组为： [{1: 10}, {2: 20}, {3: 30}],
var kvArray = [{key: 1, value: 10}, {key: 2, value: 20}, {key: 3, value: 30}];
var reformattedArray = kvArray.map(function(obj) {
  var rObj = {};
  rObj[obj.key] = obj.value;
  return rObj;
});
console.log(reformattedArray)
console.log(kvArray)
  //什么都不传的时候返回的是一个新数组。（相当于复制了一份）
```

8. **slice()** 方法返回一个从开始到结束（不包括结束，左闭右开）选择的数组的一部分前拷贝到一个新数组对象.

```
        //参数
 /*
  begin 可选
  end
  slice(begin,end):[begin,end)
 */
//返回值：一个含有提取元素的新数组
//是否影响老数组：不影响
          eg:  var arr =[1,25,34,14,26,16,47,28];
console.log(arr.slice());
console.log(arr.slice(0,2))
```

9. **splice()** 方法通过删除现有元素和/或添加新元素来更改一个数组的内容

```
        //参数
  start   从start开始删
  deleteCount  删几个
  item列表   删完之后在相同位置处加哪几个
//返回值：由被删除的元素组成的一个数组
//是否影响老数组：影响
     eg:var arr =[1,25,34,14,26,16,47,28];
//arr.splice(0)//全删
arr.splice(0,2)  //删前两个

//arr.splice(0,2,1,2,3) //删前两个 ，把1,2,3添加到删除的位置
console.log(arr);
```

10. **reduce()** 方法对累加器和数组中的每个元素（从左到右）应用一个函数，将其减少为单个值。

```
      //参数1:callback
为数组中每个元素执行的函数，该函数接收三个参数:
      accumulator(当前值):累加器累加回调的返回值; 它是上一次调用回调时   返回的累积值,
      currentValue:数组中正在处理的元素。
currentIndex可选:数组中正在处理的当前元素的索引。 如果提供了initialValue，则索   .                    引号为0，否则为索引为1。
array可选:  调用reduce的数组。
//参数2:initialValue
      用作第一个调用 callback的第一个参数的值。 如果没有提供初始值，则将使用数组中的第一个元素。  在没有初始值的空数组上调用 reduce 将报错。
//返回值:   最终累加的结果
```

```
//是否影响老数组：不影响
        eg :var arr =[2, 1, 2, 3];
var sum = arr.reduce(function (a, b) {
  return a + b;   //类加，把上一次回调的值传给a。
},0);
console.log(sum);
console.log(arr);
      //数组里所有值的和
```

方法 3.4.5，是一套的。 3,4 返回的是boolear； 5 返回的是数组。

2.数组去重 （2种方法 sort()+map()、sort()+reduce() ）

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

console.log(qucong(arr));
      function qucong(arr){

 var resulet=[];
 arr.sort().map(function(item,index){
  if(resulet.length===0 || item !==arr[index-1]){
   resulet.push(item);
  }
 });
 return resulet;
}
function qucong(arr){
 return arr.sort().reduce(function(result,item,index){
  if(result.length===0 || item!==arr[index-1]){
   result.push(item);
  }
  return result;
 },[]);
}
```

# 5．引用数据类型(函数)

**a.原生函数**

**c.自定义函数**

**d.回调函数**

**e.IIFE 自调**

函数是javascript中的一等公民。javascript中不分普通函数和构造函数。

只存在函数的普通调用和构造调用。如果函数被普通调用，那即为普通函数。
如果函数被构造调用，那即为构造函数。

**a.原生函数**

String()
Number()
Boolean()
Array()
Object()
Function()
RegExp() //正则
---在实际情况中没有必要使用 new Object() 来创建对象，因为这样就无法像常量形式那样一次设定多个属性，而必须逐一设定
---构造函数 Function 只在极少数情况下很有用，比如动态定义函数参数和函数体的时候，你基本上不会通过这种方式来定义函数
---强烈建议使用常量形式（如 /^a*b+/g ）来定义正则表达式，这样不仅语法简单，执行效率也更高，因为 JavaScript 引擎在代码执行前会对它们进行预编译和缓存。与前面的构造函数不同， RegExp(..) 有时还是很有用的，比如动态定义正则表达式
Date()
Error() //报错
---相较于其他原生构造函数， Date(..) 和 Error(..) 的用处要大很多，因为没有对应的常量形式来作为它们的替代。
创建日期对象必须使用 new Date()
创建错误对象

```
function foo(x) {
    if (!x) {
        throw new Error( "x wasn't provided" );
    }
}
```

注意:

```
var a = new String( "abc" );
typeof a; // 是 "object" ，不是 "String"  //实例，
```

**b.[[class]]**

**所有 typeof 返回值为 object的对象都包含一个内部属性 [[Class]];**

这个属性无法直接访问，一般通过 Object.prototype.toString()方法来查看 。

多数情况下，对象的内部的[[Class]] 属性和创建该对象的原生构造函数相对应

```
Object.prototype.toString.call( [1,2,3] );       // "[object Array]"
Object.prototype.toString.call( /a/i );          // "[object RegExp]"
```

注意：基本数据类型居然也有对应的[[class]]

```
Object.prototype.toString.call( null );       // "[object Null]"
Object.prototype.toString.call( undefined );// "[object Undefined]"
Object.prototype.toString.call( "abc" );    // "[object String]"
Object.prototype.toString.call( 42 );       // "[object Number]"
Object.prototype.toString.call( true );       // "[object Boolean]"
Object.prototype.toString.call(function(){})// "[object Function]"
```

// Object.prototype.toString

**c.基本数据类型的包装类----装包**

由于**基本类型值没有 .length 和 .toString() 这样的属性和方法，**

需要通过封装对象才能访问，此时 JavaScript 会自动为基本类型值 包装 （ box或者 wrap）一个封装对象

```
var a = "abc";
a.length; // 3
a.toUpperCase(); // "ABC"
```

如果需要经常用到这些字符串属性和方法，比如在 for 循环中使用 i < a.length ，那么从一开始就创建一个封装对象也许更为方便，这样 JavaScript 引擎就不用每次都自动创建了

但实际证明这并不是一个好办法，因为浏览器已经为 .length 这样的常见情况做了性能优化，直接使用封装对象来 “提前优化” 代码 反而会降低执行效率 。

一般情况下，我们不需要直接使用封装对象。最好的办法是让 JavaScript 引擎自己决定什么时候应该使用封装对象。换句话说，就是应该优先考虑使用 "abc" 和 42 这样的基本类型值，而非 new String("abc") 和 new Number(42) 。

```
eg:var a = 1;
//组包  基本数据类型--->包装类 Number
//var a = new Number(1);
console.log(a.toString());
console.log(typeof a);//number  组包只是一个过程，过程是封装在Js引擎内部的。
//不是所有的类型都有包装类， 数字、字符串、布尔 值有包装类。
//基本数据类型 ，在栈中，没有引用没有方法，只有值
```

**d.基本数据类型的包装类----拆包**

如果想要得到封装对象中的基本类型值，可以使用 valueOf() 函数;

```
var a = new String( "abc" );
var b = new Number( 42 );
var c = new Boolean( true );
a.valueOf(); // "abc"
b.valueOf(); // 42
c.valueOf(); // true
```

valueOf

JavaScript 调用 valueOf() 方法用来把对象转换成原始类型的值（数值、字符串和布尔值）。

你很少需要自己调用此函数; JavaScript 会自动调用此函数当需要转换成一个原始值时。

默认情况下, valueOf() 会被每个对象Object继承。每一个内置对象都会覆盖这个方法为了返回一个合理的值,

如果对象没有原始值，valueOf() 就会返回对象自己

**c.自定义函数**

**d.回调函数**

**e.IIFE:Immediately Invoked Function Expression 立即可执行函数表达式。**

```
(function(){})()
    //表达式:      不是function开的，前面不管是(！都是表达式
//函数声明方式: function 开头
```

###概述

在很多语言中其实我们是会区分: 类型转换 & 强制类型转换的

类型转换发生在静态类型语言的编译阶段，

强制类型转换则发生在动态类型语言的运行时.

由于在javascript中，编译过程体现的不是很明显,创建作用域差不多够了。所以我们一般都会认为javascript中 只存在强制类型转换。

**小结:**

1. //  函数在javascript中是一等公民；

```
    //在javascript中没有构造函数，只有函数的构造调用
/*fn()  ---> 普通函数
```

new Fn() ---> 构造函数*/

//实例 ：new 构造函数
var xfz = new Array(1,2); //也可以说new Arrayde 孩子 是xfz . xfz 是实例。
// 实例 ： new 后面 首字母大写 ，都是实例 。typeof 的值都是Object.都是引用类型。
2.判断是否是一个数组两种方法:
1.Array.is Array() //true
2.return Object.prototype.toString.call(arr) //[object Array]
javascript中的强制类型转换总是返回基本数据类型，不会返回对象或函数。所以组包从严格意义上来讲，不能算作强制类型转换。

强制类型转换:
隐式强制类型转换,显式强制类型转换
var a = 42;
var b = a + ""; // 隐式强制类型转换
var c = String( a ); // 显式强制类型转换
对变量 b 而言，强制类型转换是隐式的；由于 + 运算符的其中一个操作数是字符串，所以是字符串拼接操作，
结果是数字 42被强制类型转换为相应的字符串 "42" 。
而String( a )则是将 a 显式强制类型转换为字符串

**a.ToPrimitive , ToString , ToNumber , ToBoolean**

**--ToPrimitive:它负责处理对象的基本化**
检查该值是否有 valueOf() 方法。如果有并且返回基本类型值，就使用该值进
行强制类型转换。如果没有就使用 toString() 的返回值（如果存在）来进行强制类型转换。
如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误

valueOf()
JavaScript 调用 valueOf() 方法用来把对象转换成原始类型的值（数值、字符串和布尔值）。
你很少需要自己调用此函数; JavaScript 会自动调用此函数当需要转换成一个原始值时。
默认情况下, valueOf() 会被每个对象Object继承。每一个内置对象都会覆盖这个方法为了返回一个合理的值,
如果对象没有原始值，valueOf() 就会返回对象自己

toString():
对普通对象来说，除非自行定义，否则 toString() 返回 内部 属性[[Class]] 的值
即调用Object.prototype.toString()
如果对象有自己的 toString() 方法，字符串化时就会调用该方法并使用其返回值
比如数组的默认 toString() 方法经过了重新定义，将所有单元字符串化以后
再用 "," 连接起来:

**--ToString:它负责处理非字符串转化为字符串**
基本类型值的字符串化规则为:
String()
对象 (ToPrimitive)
oPrimitive 转成基本数据类型再String()

String() 方法
null 转换为 "null"
undefined 转换为 "undefined"
Boolean：true 转换为 "true" ;false 转换为 "false"
数字的字符串化则遵循通用规则
1 转换为 "1"（当然可能会有指数的形式 1.07e21)

--ToNumber:它负责处理非数字化为数字
基本数据类型数值化规则:
Number()
对象 ()
ToPrimitive 转成基本数据类型再Number()

Number方法
Null==>0
Undefined==>NaN
Boolean：true==>1 false==>0

字符串
就是把字符串两边的空白字符去掉，然后把两边的引号去掉，看它能否组成一个合法的数字。
如果是，转化结果就是这个数字；否则，结果是NaN。
当然也有例外，比如空白字符串转化为数字的结果是0。

--ToBoolean：它负责非布尔化为布尔
假值 --->false
真值 ---> true

以下这些是假值:
undefined
null
false
+0 、 -0 和 NaN
""

真值就是假值列表之外的值。

eg:

```
1.将其他基本类型转换成字符串类型     //   String() 方法
   var  str = String(ture);
    console.log(str, typeof str) ;        //  true string
2.将对象转成字符串类型。
    //将对象基本化。
 1.toString()
var a ={
   toString :function(){
         return  1234;
   },
  valueOf :function(){
         return  123;
   }
}
var str = String(a);
console.log(str , typeof str);
/*--ToPrimitive:它负责处理对象的基本化
  *
  * 检查该值是否有 toString() 方法。如果有并且返回基本类型值，就使用该值进行强制类型转换
  *
  * 如果没有返回值不是基本数据类型
  *    检查该值是否有 valueOf() 方法，如果有并且返回基本类型值，就使用该值进行强制类型转换
  *
  * 如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误


  //  只有包装类的ValueOf()的方法，才会返回包装类的数据类型，其他返回原对象。
  //Deta  或自定义方法，来调时，先调toString()的方法，再调valueOf()的方法。
 2.toNumber()方法。
 eg:var a = {
   valueOf:function(){
    return {};
   },
   toString:function(){
    return "   123a    ";
   }
  }
  var mun = Number(a);

  console.log(mun,typeof mun);  //返回的是 ： NaN "number"

                        //  NaN ： Number 之后，返回的是：object;
         //Number（）方法就是把字符串两边的空白字符去掉，然后把两边的引号去                    .                    看它能否组成一个合法的数字
如果是，转化结果就是这个数字；.                    .                    否则，结果是NaN。


 /*--ToPrimitive:它负责处理对象的基本化
   检查该值是否有 valueOf() 方法。如果有并且返回基本类型值，就使用该值进行强制类型转换。、
```

如果没有就使用 toString() 的返回值（如果存在）来进行强制类型转换。

如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误

valueOf()
JavaScript 调用 valueOf() 方法用来把对象转换成原始类型的值（数值、字符串和布尔值）。
你很少需要自己调用此函数; JavaScript 会自动调用此函数当需要转换成一个原始值时。
默认情况下, valueOf() 会被每个对象Object继承。每一个内置对象都会覆盖这个方法为了返回一个合理的值，
如果对象没有原始值，valueOf() 就会返回对象自己

toString():
对普通对象来说，除非自行定义，否则 toString() 返回 内部 属性[[Class]] 的值
即调用Object.prototype.toString()
如果对象有自己的 toString() 方法，字符串化时就会调用该方法并使用其返回值
比如数组的默认 toString() 方法经过了重新定义，将所有单元字符串化以后
再用 "," 连接起来:
*/

###parseInt(value) 与 Number(value) 转换字符串的区别

parseInt转换情况

字符串->只要最高位是数字则会转换成数值；否则转换成NaN

Number转换情况

就是把字符串两边的空白字符去掉，然后把两边的引号去掉，看它能否组成一个合法的数字。
如果是，转化结果就是这个数字；否则，结果是NaN。
当然也有例外，比如空白字符串转化为数字的结果是0。

3.==操作符

==比较符在js中的规则：趋于数字化(不同数据类型之间的比较，排除掉obj与obj)
obj与obj之间的 == 只比较栈中保存的地址值

==操作符解析①
首先把javascript中的数据类型分成两组:
1. String、Number、Boolean和Object(有)
2. Undefined和Null(无)

3. 1与2之间的比较都为false
4. Undefined和Null之间的比较为true
5. NaN 不等于 NaN

null和undefined不得不说的故事(设计者原本的意图)
假如你打算把一个变量赋予对象类型的值，但是现在还没有赋值，
那么你可以用null(关键字)表示此时的状态(证据之一就是typeof null 的结果是'object')；
相反，假如你打算把一个变量赋予原始类型的值，但是现在还没有赋值，
那么你可以用undefined（标识符）表示此时的状态。

统统往数字转！！！

**Array的普通调用效果等于构造调用。**

1. string()被普通调用，强制转化
被构调用，构造出一个实例字符串。

包装类函数的普通调用---> 强制类型转化
包装类函数的构造调用---> 返回包装类所对应实例
包装类函数：String() 、Number() 、Boolean()

2.Array被普通调用，被构造调用返回的是一个实例。

###day01

1.HTML5存储
window.localStorage ：永久存储
getItem
setItem
removeItem
clear
window.sessionStorage: 会话级别(一个页面)
storage事件
当storage里的数据有改动时在共享页面触发

cookie(小甜饼)
session(服务端的存储)

2.js的性能问题

性能瓶颈:运行时编译，在代码被执行前的几微秒进行编译
动态性，弱类型:编译时无法精确分配空间

栈　　: 存储基本数据类型;先进后出
堆　　: 存储结构，一般存储引用数据类型
队列　　: 定时器队列 事件队列;先进先出

3.值 数据类型 变量
js中的变量是没有类型的,值是有类型的,变量拿的是值所持有的类型。
数据类型
基本数据类型: null undefined number boolean string
引用数据类型: object（数组 function {}）
typeof null --> object(bug)
typeof [] ---> object
typeof fn ---> function

数据传递（值传递）
a=b
形参 = 实参
js数据传递都是值传递，基本数据类型传递值,引用数据类型传递基地址值(string)

垃圾回收机制
变量的释放:瞬间的
垃圾的回收:周期性的
标记清除
引用计数

4.引用数据类型
数组（ES5）
api: 该api的意义
参数
返回值
是否影响原数组
衬垫代码
稀疏数组
有空隙的数组，会导致api无法正常工作
伪数组
具有length属性，以数字为属性名的对象
将伪数组转为真数组:
将最干净的那个slice方法的this指向换成那个伪数组
var arr = Array.prototype.slice().call(伪数组)
数组去重
arr.sort().reduce(function(累加器,item,index,arr){},[])
函数
函数是js中的一等公民，js中没有专门的构造函数，
只有函数的普通调用和函数的构造调用

[[class]]
最干净的toString方法
[[class]] = Object.prototype.toString.call(相应的对象);

组包，拆包，ToPrimitive规则
高低类型之间进行的转换,并不是强制类型转换
组包:基本数据类型 --> 包装类(一般都是隐式)
拆包:包装类--->基本数据类型（valueOf）
ToPrimitive规则:对象-->基本数据类型
valueOf方法
包装类:返回基本数据类型
对象:返回原对象
注意valueOf方法可以被重写
toString方法
没有重写过toString方法:[[class]]
重写过toString方法：看重写的逻辑
array toString方法：将数组内的元素用，链接
5.强制类型转换
基本数据类型之间的转化叫强制类型转换
运行时转化叫强制类型转换
编译时的转化叫类型转换

js中只有强制类型转换
显示强制类型转换
隐式强制类型转换

ToString规则( String() )
其他基本数据类型 --> string
null 转换为 "null"
undefined 转换为 "undefined"
Boolean：true 转换为 "true" ;false 转换为 "false"
数字的字符串化则遵循通用规则
1 转换为 "1"（当然可能会有指数的形式 1.07e21）
对象 ---> string
ToPrimitive规则( 先toSring() 再调valueOf()  报错typeerror )--->基本数据类型
其他基本数据类型 --> string

ToNumber规则( Number() )
其他基本数据类型 --> number
Null==>0
Undefined==>NaN
Boolean：true==>1　false==>0
字符串
就是把字符串两边的空白字符去掉，然后把两边的引号去掉，看它能否组成一个合法的数字。
如果是，转化结果就是这个数字；否则，结果是NaN。
当然也有例外，比如空白字符串转化为数字的结果是0。
对象 ---> number
ToPrimitive规则( 先valueOf() 再调toSring()  报错typeerror )--->基本数据类型
其他基本数据类型 --> number

==操作符
1.有的状态  无的状态(null undefined)
2.有和无之间做比较返回false
无跟无之间做比较返回true
NaN 永远 不等于 NaN
对象与对象之间比较的永远是栈中存储的地址值

ToBoolean规则( Boolean() )
假值
undefined
null
false
+0 、 -0 和 NaN
""
真值
其他都是真值

ToBoolean规则（Bloolean()  构造）
var a = Boolean({});
console.log(a);  //  true

**强制转换**

**1.显示强制转换**

```
var a = 42;
var b = String( a );
var c = "3.14";
var d = Number( c );


var a = 42;
var b = a.toString();
var c = "3.14";
var d = +c;          //一元运算符 + 会将操作数显式强制类型转换为数字

var a = "0";
var b = [];
var c = {};
var d = "";
var e = 0;
var f = null;
var g;
```

```
Boolean( a ); // true
Boolean( b ); // true
Boolean( c ); // true
Boolean( d ); // false
Boolean( e ); // false
Boolean( f ); // false
Boolean( g ); // false

var a = "0";
var b = [];
var c = {};
var d = "";
var e = 0;
var f = null;
var g;
!a
!!a; // true
!!b; // true
!!c; // true
!!d; // false
!!e; // false
!!f; // false
!!g; // false
```

一元运算符 ! 显式地将值强制类型转换为布尔值。但是它同时还将真值反转为假值（或者将假值反转为真值）。所以显式强制类型转换为布尔值最常用的方法是 !!，因为第二个 ! 会将结果反转回原值。

**2.隐式强制转换**

数字与字符串

```
var a = 42;
var b = a + "";
```

隐式强制类型转换为布尔值

(1) if (..) 语句中的条件判断表达式。
(2) for ( .. ; .. ; .. ) 语句中的条件判断表达式（第二个）。
(3) while (..) 和 do..while(..) 循环中的条件判断表达式。
(4) ? : 中的条件判断表达式。
(5) 逻辑运算符 || （逻辑或）和 && （逻辑与）左边的操作数（作为条件判断表达式）

```
var a = 42;
var b = "abc";
var c;
var d = null;
if (a) {
    console.log( "yep" ); // yep
}
while (c) {
    console.log( "nope, never runs" );
}
c = d ? a : b;
c; // "abc"
if ((a && d) || c) {
    console.log( "yep" ); // yep
}
```

### == 操作符

### || 和 &&

```
var a = 42;
var b = "abc";
var c = null;
a || b; // 42
a && b; // "abc"
c || b; // "abc"
c && b; // null
```

|| 和 && 首先会对 第一个操作数 （a 和 c ）执行条件判断，
如果其不是布尔值（如上例）就先进行 ToBoolean 强制类型转换，然后再执行条件判断。
对于 || 来说，如果条件判断结果为 true 就返回第一个操作数（a 和 c ）的值，

如果为 false 就返回第二个操作数（ b ）的值。

对于 && 来说，如果条件判断结果为 true 就返回第二个操作数（ b ）的值，

如果为 false 就返回第一个操作数（ a 和 c ）的值。

|| 和 && 返回它们其中一个操作数的值，而非条件判断的结果（其中可能涉及强制类型转换）。

c && b 中 c 为 null ，是一个假值，因此 && 表达式的结果是 null (即 c 的值)，而非条件判断的结果 false 。

a || b; 大致相当于 a ? a : b;

a && b; 大致相当于 a ? b : a;

短路:

|| 或 第一个进行ToBoolean，如果flase ,看第二个。返回两个。

ture，看第一个。返回一个

&& 与 第一个进行ToBoolean，flase,停止，

第一个 true ,看第二个ToBoolean

```
console.log(NaN||1);    // 1
console.log(NaN&&1);    //NaN
```