

# H5编程

---

第二部分：JAVASCRIPT语言基础

杨强

# 课程环境

---

**Safari浏览器：**主要的测试与体验工具。

■同时安装**Firefox**浏览器，其他浏览器可选：**Chrome**与**IE**浏览器。



参考站点：<https://developer.mozilla.org/zh-CN/docs/Web/JavaScript>

# 08脚本基础

目的：

- 1、掌握Javascript基本语言结构。
- 2、掌握Javascript的数据类型。
- 3、掌握Javascript的运算。
- 4、掌握Javascript的流程控制。

1.语言结构

2.基本数据

3.运算符与表达式

4.流程控制

# 在html中使用脚本

---

在html中使用脚本有三种方式：

- `<script>` 标签中使用。
- 使用 `<script>` 标签引入外部脚本。
- 在标签的事件属性中使用脚本。

注意：

- 在JavaScript中还提供了一种独特的语法，使用 `javascript:void(一个语句)` 执行脚本。主要用于 `a` 标签的 `href` 属性中，其中 `void()` 可以省略，直接使用脚本。

# 使用JavaScript:脚本语句

---

在某些标签中可以使用JavaScript:执行脚本，语法：

## ■JavaScript:脚本语句

■注意：void中的脚本只能一个语句。多个语句使用逗号分隔，最后不要使用分号。

```
<!doctype>
<html>
  <head>
    <title>javascript语言</title>
    <meta charset="utf-8">
  </head>
  <body>
    <a href= “javascript:alert('脚本执行'),location.href='http://www.baidu.com'”>链接</a>
  </body>
</html>
```

# 浏览器对脚本不支持的处理

---

尽管现在浏览器很少有不支持javascript脚本的，但如果浏览器不支持脚本可以使用`<!-- -->`来注释`<javascript>`标签，不支持的浏览器会把该标签当成注释来处理。

# JavaScript语言结构

---

JavaScript语言尽管与java之间有很深的渊源，但却是两个独立的语言。其语言的基本结构与C++/Java/C#等语言完全一致，主要包含三个基本的语法单元：

- 字（**word**）：空格分隔字。
- 句（**statement**）：字构成语句，分号分隔语句。
- 块（**block**）：{ } 分隔块。语句构成块。

# 字的分类

---

在JavaScript中字分成2大类：

- 关键字：浏览器解释模块内置规定的字，比如：**var**，**while**等。
- 标识字：用户在编写JavaScript代码的时候命名的字。

注意：

- 这里对标识字容易产生识别错误，比如会把**String**与**Number**误认为关键字，但他们却是关键字。这些字是由实现**String**与**Number**类型库的程序员命名的、不是浏览器解释模块命名的字。如果要使用其他程序员实现的系统标准库与第三方库，就必须遵循他们的命名，但这些命名不是关键字。
- 除了标识字与关键字，还有一些特殊的符号也可以当成字，比如：运算符号，字面值“**hello**”、**20**等。



# 语句的分类

---

语句由字构成，JavaScript语言的语句大致分成2类：

- 数据语句：用来声明数据或者定义数据类型。
- 功能语句：用来处理数据（赋值/取值/运算）。

注意：

- 在JavaScript语言中还存在一种特殊的语句：注释语句。
  - //行注释：//同行且在后面的字符都当成注释，浏览器不解释执行。
  - /\*\*/块注释：/\*\*/中间的字符都当成注释，浏览器不解释执行。

# 代码块的分类

---

代码语句构成块，在JavaScript语言中有不同的块来负责不同的语法功能，大致可以分成如下几类：

- 匿名块：用来改变变量的作用域。
- 控制块：用来改变语句的执行流程。
- 函数块：用来封装一组语句，便于重复调用，复用代码，用来更加结构良好地组织代码。
- with块：用来指定默认作用对象。
- try、catch、finally块：用来处理异常，当异常发生时用来控制程序流程。
- 字典块/类块：用来封装数据与类。

# 语言结构的基本规则

---

**javascript**语言首先遵循基本的语言结构规则：

- 字的命名规则：字开头字符必须\_、\$、字母，字组成规则必须\_、\$、字母与数字。
- 语句规则：语句按照顺序执行。
- 块规则：**JavaScript**块按照一定规则嵌套（绝大部份可以随意嵌套），却块不能交错。（**java**中的某些块是不能随意的嵌套的，比如函数块，但在**JavaScript**中连函数块都是可以嵌套的）。

# JavaScript的关键字

---

JavaScript一共28个关键字：

- break/continue/return (3)
- for/do/while/if/switch/case/else/in/default (9)
- catch/try/finally/throw (4)
- false/true/void/null (4)
- typeof/instanceof/new/delete/function/this/var (7)
- with (1)

# 字面值

---

在JavaScript中有一种特殊的字，用来直接表示数据，这就是字面值（在很多语言中没有作为字来说明，因为他们有的不是使用固定的字符组成，却遵循一定规则），字面分成如下几类：

- 空值：**null**。使用关键字。
- 逻辑值：**true** | **false**。使用关键字。
- 整数值：**12**。用户使用数字与正负符号自己构成。
- 小数值：**12.56**。用户使用数字、小数点、**e**字符与正负符号自己构成（包含科学计数法）。
- 字符串值：**"字符串"**或者**'字符串'**。使用 双引号**""**或者单引号**"**界定，内容用户自己构成。
- 序列值（集合值：数组，复合值：字典）：**[值1, 值2, ...]**，**{key1 : value1, ...}**
- 特殊值：**Infinity**(无穷大)，**NaN**(非数字)，**undefined**(没有定义)（不是关键字，是系统库定义的特殊值）

# 空语句

直接使用；表示空语句。

- 空语句不做任何执行。
- 但空语句在某些情况下是有用的，因为它表示存在一个语句。比如：空循环，可以使用；表示存在一个语句，并省略 {} 。如果不使用；表示没有语句，则为了循环的语法完整性，必须使用 {} 。

```
<script>
    //这个死循环导致页面一直解释。
    while(true); //表示空语句。

    while(true){
        ;
    }

    while(true){
    }
    console.log("不会有输出");
</script>
```

# var语句

---

## var语句的作用:

- 申请内存空间。

## 语法:

- **var** 空间名/变量名;
- **var** 变量名1,变量名2,.....;

## 注意:

- 没有赋值的变量，其值默认是**undefined**。不建议使用。

```
<script>  
    var a;  
    console.log(a);//打印值: undefined  
    console.log(a);//使用变量a  
    var b,c;//定义多个变量  
</script>
```

# 赋值语句

语句作用：

- 用来把数据放入内存空间。

语法：

- 变量名=值;
- 变量名=变量名;

注意：

- 可以在申请空间的同时赋值：**var** 变量名=值,变量名=变量名,...; 而且**var**可以省略。
- 一旦变量赋值后，就可以直接使用变量名当成数据一样使用。

```
<script>
  var a;
  console.log(a);//打印值： undefined
  a=20;//赋值
  console.log(a);//使用变量a
  var b,c;//定义多个变量
  var d=10,e,f=30,g=40;//定义变量的时候赋值
</script>
```



# 输出语句

---

输出内容到页面：

- `document . write(数据);` //不会输出换行
- `document . writeln(数据);` //输出完毕，会输出换行

控制台调试输出：

- `console . log(数据);` //输出到控制台

注意：

- 有的浏览器的某些版本不支持console功能。

# 函数调用语句

---

函数调用语句的语法:

■ 变量=函数(参数,.....);

■ 注意:

□ 函数可以没有返回值。也可以没有参数。

□ 调用函数的时候，需要明白返回的是什么数据，参数的作用。

```
<script>  
    var re=eval("45+55");  
    console.log(re);  
</script>
```

# 对象使用语句

---

在JavaScript中一个很重要的概念就是对象。一般对象使用可以由如下语句完成：

- 创建对象语句：
- 对象的数据（也称对象属性）访问语句：
- 对象的功能访问语句：
- 释放对象语句：

# 创建与释放对象

---

创建对象:

■ 对象变量 = new 对象类型(对象初始化数据);

释放对象

■ 对象变量 = null;

```
<script>
    var s=new String("Hello");
    console.log(s);
    s=null;
    console.log(s);    //对象空值
</script>
```

# 访问对象属性

获取对象属性值:

■对象变量.属性;

修改对象属性值:

■对象变量.属性=值;

■注意: 有得只读属性, 是不能修改对象的属性的。

释放对象属性:

■delete 对象变量.属性;

```
<script>
    var s=new String( "Hello" );
    console.log(s.length);
    s.length=45;
    console.log(s.length);//仍然是原来的长度, 不是45
    s=null;
    console.log(s);
</script>
```

# 访问对象函数

---

访问对象函数的语法：

■ 变量=对象变量.函数(输入参数, .....);

```
<script>
    //创建对象
    var s=new String( "Hello" );
    //访问函数
    var re=s.substr(1,3);    //指定获取的子字符串，从位置1开始，取的子串长度是3。
    console.log(re);//返回ell。
    s=null;
    console.log(s);
</script>
```

前面输出的输出语句就是调用document对象与console对象的功能。

# 数据与数据类型

JavaScript语言中，在语法上，与其他语言一样，数据分成**基本类型**（**primary type**）与**用户类型**（由基本类型复合而成的类型）。不管基本类型，还是复合类型都是使用一样的使用方式，基本类型也可以像对象类型一样调用成员属性与成员函数。基本类型有如下几种：逻辑类型，数字类型，字符串类型，还有两个特殊的基本类型**null**与**undefined**。

```
<script>
  var i=20;
  console.log(i.toString());           //变量i是对象
  console.log(20.toString());         //字面值也是对象
</script>
```

JavaScript中的用户类型有如下几种：逻辑类型(**Boolean**)/数字类型(**Number**)/字符串类型(**String**)/数组类型(**Array**)/日期类型(**Date**)，以及其他用户定义类型。

**注意：**每个构成用户类型对象的基本类型的值称为**原始值**。原始值不可改变，对象值是引用。

# 确定变量类型

---

确定一个变量的类型有两种

■方式一：隐式数据类型，由值决定变量类型。

**var 变量=初始值;** 比如: **var i=20;**

■方式二：显式数据类型，创建变量的时候指定类型。

**var 变量=new 类型(初始值);** 比如: **var a=new Number("20");**

■注意：使用第二种方式，可以使用字符串等不同方式初始化；第一种就不行。

```
var a=new Number( "20" );    //而语句var a="20";得到的变量a是字符串类型。  
var b=new Number(20);
```



# 获取变量与值的类型

---

返回一个字的类型：

■ 语法1:        **typeof** 值;                      //返回字符串

```
var i=20;  
var d=new Date();  
console.log(typeof i);//number  
console.log(typeof d);//object  
console.log(typeof true);//boolean  
console.log(typeof {});//object  
console.log(typeof []);//object  
console.log(typeof null);//object  
console.log(typeof undefined);//undefined
```

# 逻辑类型

基本类型:

■ `var b=true;//false`

```
var b1=false;
var b2=new Boolean(b1);// “true” | “hello” | 20
console.log(b1);
console.log(b2);
```

对象类型

■ `var b=new Boolean(...);`//任何类型//`"true"|"hello"|20`

提示: 对象类型的如下初始值返回**false**

■ 对象类型的如下初始值返回**false**: `0`、`-0`、`null`、`""`、`false`、`undefined` 或者 `NaN`; 否则返回 **true**。

■ 对象类型的初始值如果是一个**"false"**逻辑变量, 也返回**true**

# 逻辑对象的操作

---

把逻辑值转换为字符串：

■ **toString();**

把逻辑对象转换为基本类型：

■ **valueOf();**

```
var b1=false;  
var b2=new Boolean(b1);  
console.log(b2.valueOf());  
console.log(b2.toString());
```

# 数字类型

---

**JavaScript**中数字类型不分整数与小数，所有的数字都采用浮点存储表示，字节数是**8**字节（**64**位），注意：数组下标范围采用**4**字节（**32**位）。

- 表示范围：最大 **$\pm 1.7976931348623157 \times 10^{308}$** 。最小 **$\pm 5 \times 10^{-324}$** 。
- 有效位数范围：  **$\pm 9\,007\,199\,254\,740\,992$** （**2**的**53**次方），超过就四舍五入。
- 识别的字面值方式：**8**进制（**07777**），**10**进制（**9999**），**16**进制（**0xFFFF**），普通小数（**99.9999**），科学计数法小数（**999.999E99.99**）

语法：

- 基本类型：**`var i=20;`**
- 对象类型：**`var i=new Number("234e-2");`** //可以使用字符串

# 两个特殊的数字值

---

为了表示无穷与非数字，JavaScript引入了两个特殊的全局值：

■ **Infinity**：无穷； **-Infinity**负无穷大， **+Infinity**正无穷大。

■ **NaN**：非数值；

```
var i=077;//8进制，10进制，16进制，普通表示，科学技术法  
console.log(i);  
console.log(typeof Infinity);//无穷：number  
console.log(typeof NaN);//非数值：number
```

# 数值类型的全局常量属性

---

**Number**类型中提供了几个全局常量属性：

- **MAX\_VALUE**：最大数值
- **MIN\_VALUE**：最小数值
- **NEGATIVE\_INFINITY**：负无穷大
- **POSITIVE\_INFINITY**：正无穷大
- **NaN**：非数值。

```
console.log(Number.MAX_VALUE);  
console.log(Number.MIN_VALUE);  
//NaN可以不使用Number作用域前缀  
console.log(Number.NaN);  
console.log(Number.NEGATIVE_INFINITY);  
console.log(Number.POSITIVE_INFINITY);
```

# 数值类型的格式转换

---

- 把数值转换为字符串，可以指定基数（二进制）：`toString(基数)`；默认10(2 ~ 36)
- 把数值转换为本地格式的字符串：`toLocaleString()`
- 把数值转换为有固定位数小数的字符串：`toFixed(小数位数)`；默认0(0-20)
- 把数值转换为有最小长度的字符串：`toPrecision(有效位数)`；默认1(1-21)
- 把数值转换为科学计数法的字符串：`toExponential(小数位数)`；默认0(0-20)
- 返回数值对象的基本类型：`valueOf`

注意：

- 其中的参数不设置，则取默认值。

```
console.log(a.toString(2));  
console.log(b.toString());  
console.log(a.toLocaleString());  
console.log(a.toFixed(10));  
console.log(a.toExponential(10));  
console.log(a.toPrecision(10));
```

# 数值类型与对象类型的转换

---

Javascript提供了一个全局函数**Number**可以把对象转换为数值类型（返回的是基本类型，不是对象类型），语法：

■ **Number(对象);**

例子：

```
var test1= new Boolean(true);  
var test2= new Boolean(false);  
var test3= new Date();  
var test4= new String("999");  
var test5= new String("999 888");  
console.log(Number(test1)); //1  
console.log(Number(test2)); //0  
console.log(Number(test3)); //1469591089391  
console.log(Number(test4)); //999  
console.log(Number(test5)); //NaN
```



# 数值类型的判定

---

JavaScript提供两个函数来判定数值是否有意义：有穷，非数值。返回逻辑值。

判定数值是否是**有穷大**：

■ isFinite(数值)

判定对象是否非数值：

■ isNaN(数值)

```
var test1= new Boolean(true);  
var test2= new Boolean(false);  
var test3= new Date();  
var test4= new String( "999" );  
var test5= new String( "999 888" );  
console.log(isNaN(test3));    //false  
console.log(isFinite(test3)); //true
```

# 把字符串解析成数值类型

---

**JavaScript**提供两个函数来把字符串解析成数值：一个按照整数格式解析，一个按照小数格式解析。

■按照整数解析： **parseInt(字符串)**

■按照小数解析： **parseFloat(字符串)**

```
console.log(parseInt( "0xff" ));      //255  
console.log(parseFloat( "23.45e3" )); //23450  
console.log(parseFloat( "NaN" ));     //NaN  
console.log(parseFloat( "ABC" ));     //NaN  
console.log(parseFloat( "123abc" ));  //123
```

# 字符串类型

定义字符串对象有两种方式：

- 基本类型： `var s1=""`
- 对象类型： `var s2=new String(.....);`
- 注意：字符串表示也可以使用单引号

可以直接把其他对象转换为字符串

- `var` 字符串对象=`String(对象);`

```
var s1="Hello";  
var s2=new String(0xFFFF);  
console.log(s1);  
console.log(s2);
```

📄 Hello

📄 String {0: "6", 1: "5", 2: "5", 3: "3", 4: "5", length: 5}

基本类型与对象类型使用`console.log`输出有差异。

```
var s=String(d);  
console.log(s);
```

# 字符串中字符表示方式

---

除了正常的字符表达方式，因为某些字符本身在JavaScript中被浏览器解释器作为内置特殊的作用：比如双引号，为了解决这种字符的歧义，字符串中字符引入了多种表示方式：

■ 字符的转义表示（转义符）：

\0(空字符)      \b(退格符)      \t(水平制表符)      \n(换行符)      \v(垂直制表符)

\f(换页符)      \r(回车符)      \"(双引号)      \'(单引号)      \\(反斜杠)

\xXX(字符的1字节ASCII码16进制表示)      \uXXXX(字符的2字节Unicode码16进制表示)

■ 使用Unicode创建字符串的两种方式：直接构造 / 构造函数构造 / 全局函数构造。

```
var s1= “\u4e2d\u56fd\u4eba” ;//中国人  
var s2=String.fromCharCode(0x4e2d,0x56fd,0x4eba);
```

# 字符串的长度属性

---

可以使用**length**属性直接返回字符串长度。

## ■.length

```
var s1= “Hello” ;  
var s2=new String(0xFFFF);  
console.log(s1.length);  
console.log(s2.length);
```

# 字符串的字符操作

由于JavaScript中没有字符这一语法要素，所以字符的存储形势还是字符串，只是长度length为1而已。字符串提供了2个函数返回值字符串中字符：

- `charAt(位置)` 返回在指定位置的字符，返回类型字符串
- `charCodeAt(位置)` 返回在指定的位置的字符的 Unicode 编码。返回类型整数。
- 注意：字符串的位置从0开始到length-1。同时Javascript目前对扩展Unicode支持得不好（3字节Unicode）

```
var s1="Hello";  
//汉字目前都是2字节（不超过65535）  
var s2=new String("我们都是好老头"); //后面1个字符都是超过2字节的字符（unicode的扩展）  
console.log(s1.charAt(2)); //返回：l  
console.log(s2.charAt(0)); //返回：我  
console.log(s1.charCodeAt(2)); //返回：108  
console.log(s2.charCodeAt(0)); //返回：25105  
console.log(s2.charAt(s2.length-1)); //返回：空  
//0x10400=66560(Javascript表示不对)  
console.log(s2.charCodeAt(s2.length-1)); //返回：56320  
console.log(String.fromCharCode(66560));  
document.write(String.fromCharCode(66560));
```

0的unicode码0x10400，10进制是66560。超过65535，在Javascript中的处理明显是错误的。而且把unicode码转换成字符，在控制台与浏览器页面都无法正常显示。

# 字符串的检索

---

可以根据一个已知字符串，找出它在某个字符串中的位置，也称字符串搜索：

- **indexOf**(被检索字符串, 开始检索位置); 返回字符串的开始位置，没有检索到返回-1，类型整数。
- **lastIndexOf**(被检索字符串, 开始检索位置); 与**indexOf**相同，只是检索方向从后往前，同样返回字符串开始位置。

```
var s= "this is a clever dog!";  
var idx=s.indexOf( "is" ,4); //返回的位置从字符串开始位置计  
console.log(idx);    //5  
idx=s.lastIndexOf( "is" ,4);  
console.log(idx);//2,如果从s.length-1开始，则返回5  
  
idx=s.indexOf("fox");  
console.log(idx);//返回-1
```

# 字符串取子串操作(1)

**String**提供4两个函数取已有字符串的子串：

- **substr**(开始位置, 子串长度);返回子串, 类型字符串。
- **substring**(开始位置, 结束位置);不含结束位置字符。

注意：

- 如果位置为负数, 则从取值0.
- 如果**substring** 开始位置 比 结束位置大, 那么该方法在提取子串之前会先交换这两个参数。
- 长度取负数, 等于长度为0.

```
var s= "this is a clever dog" ;  
var s1=s.substr(2,5);//从2开始取5个字符:is is  
console.log(s1);  
s1=s.substr(50,5);//越界的情况  
console.log(s1); //返回空  
var s2=s.substr(2);//从2开始取后面所有字符  
console.log(s2);//is is a clever dog  
var s3=s.substring(2,5);  
console.log(s3);  
s3=s.substring(-2,6);//越界的情况  
console.log(s3);//this  
s3=s.substring(s.length,-5);//越界的情况  
console.log(s3);
```



# 字符串取子串操作(2)

**String**提供了一个与**substring**功能完全相同，但更加灵活的方法：

■ **slice**(开始位置，结束位置)；不含结束位置的字符

注意：

■ **slice**与**substring**的区别在于支持负数作为参数，当使用负数表示从最后一个位置往前计数。**-1**表示倒数第一字符，**-2**表示倒数第二个字符以此类推。

■ **slice**的开始位置大于结束位置，返回空。

```
var s= “abcde” ;
```

```
var s1=s.slice(-3,-1);  
console.log(s1);//cd
```

```
s1=s.slice(-1,-2);  
console.log(s1);//空
```

```
var s2=s.slice(0,-1);  
console.log(s2);//abcd
```

```
s2=s.slice(3,2);  
console.log(s2);//空
```

# 字符串合并与拆分

---

字符串合并使用**concat**函数

- **concat**(字符串1,.....,字符串n);返回合并后字符串。

字符串拆分使用**split**函数

- **split**(分割符,返回数组长度);返回分割后的字符串数组。
- 如果不指定数组长度，则长度无限制。

```
var s1="Hello";  
var s2=new String(" Javascript");  
var s3=s1.concat(s2," HTML"," CSS");  
console.log(s3);//Hello Javascript HTML CSS  
var a1=s3.split(" ");  
console.log(a1);  
//[ "Hello", "Javascript", "HTML", "CSS"]
```

# 字符串转换

---

字符串可以转换成大小写：

- `toLocaleLowerCase()` 把字符串转换为小写。
- `toLocaleUpperCase()` 把字符串转换为大写。
- `toLowerCase()` 把字符串转换为小写。
- `toUpperCase()` 把字符串转换为大写。

返回基本类型的字符串

- `valueOf()` 返回某个字符串对象的原始值。

# 字符串HTML格式化

---

可以在已有字符串两端加上HTML标签。

- **anchor()**          创建 HTML 锚。
- **big()/small()**    用大/小号字体显示字符串。
- **blink()**            显示闪动字符串。
- **bold()/ italics**    使用粗体/斜体显示字符串。
- **fixed()**            以打字机文本显示字符串。
- **fontcolor()/ fontsize()**    使用指定的颜色/尺寸来显示字符串。
- **link()**             将字符串显示为链接。
- **strike()**            使用删除线来显示字符串。
- **sub()/sup()**        把字符串显示为下标/上标。

# 字符串的正则表达式操作

---

尽管JavaScript提供了专门的正则表达式操作，但字符串中也体统了三个正则表达式操作函数：

## ■ 正则表达式检索：

- `match`(检索字符串或者正则表达式) 找到一个或多个正则表达式的匹配，返回匹配的字符串数组。
- `search`(正则表达式) 检索与正则表达式相匹配的值。返回匹配的位置。没有找到返回-1。

## ■ 正则表达式替换：

- `replace`(被替换的子串 / 正则表达式, 替代字符串); 替换与正则表达式匹配的子串。返回替换后的结果字符串。

## ■ 正则表达式拆分：

- `split`(正则表达式, 数组长度限制);

**提示：** 正则表达式在下面有专门的主题讲解。

# 字符串中正则表达式例子

---

```
var s= “this is a clever dog!” ;
```

```
var r=s.match( “is” );//字符串  
console.log(r);//[ “is” ] 第二个is
```

```
var p=/\w+/g; //不要使用双引号表示字符串  
r=s.match(p);//正则表达式  
console.log(r);//[ “this” , “is” , “a” , “clever” , “dog” ]
```

```
r=s.match(/\w+/g);//直接使用  
console.log(r);//[ "this", "is", "a", "clever", "dog"]
```

# 一维数组定义

---

构造一维数组对象：

■方式一：

□var arr=[.....,.....,.....];

■方式二：

□var arr=new Array();

□var arr=new Array(数组长度);

□var arr=new Array(元素1, 元素2, .....);

```
var arr1=["H","e","l","l","o"];  
var arr2=new Array();  
var arr3=new Array(5);  
var arr4=new Array("H","e","l","l","o");
```

# 访问数组

---

访问数组的语法:

■ 数组对象[小标];

读数组元素:

■ 直接使用数组

写数组元素:

■ 使用=运算符

**注意:** 数组下标从0开始

```
//读数组  
var a=arr1[0];  
console.log(a);//H  
console.log(arr1[1]);//e
```

```
//写数组  
arr1[3]=20;//可以使用其他类型的值  
console.log(arr1);//["H", "e", "l", 20, "o"];
```



# 数组长度与越界访问

---

获取数组长度使用length属性:

- 数组对象.length

数组的越界访问:

- 当越界读数组的值, 则返回undefined
- 当越界写数组的值, 则数组动态增加。
- 注意: 我们称这种越界写值产生的数组为稀疏数组, 稀疏数组中间的元素是undefined。

```
console.log(arr1.length);
```

```
console.log(arr1[10]);//越界读值  
arr1[10]=30;//越界写值
```

```
console.log(arr1.length);//11  
//越界写值后的结果, 数组扩容
```

```
console.log(arr1[9]);  
//稀疏部分依然是undefined
```

# 数组合并与子数组

---

把多个数组合并成一个数组：

- 数组对象.concat(数组对象1, .....);返回数组
- 数组对象.concat(数组元素,...,);
- **提示：** 也可以数组对象与元素混合

把数组合并成一个字符串：

- 数组对象.join(分割符号);返回字符串

取数组的子数组：

- 数组对象.slice(开始位置, 结束位置);返回数组

```
var arr=arr1.concat(arr3,arr4);  
console.log(arr);
```

```
var str=arr1.join(":");  
console.log(str);//H:e:l:l:o
```

```
str=arr1.join();//默认逗号  
console.log(str);//H,e,l,l,o
```

```
var a=arr1.slice(1,3);  
console.log(a);//["e", "l"]
```

# 数组的排序与逆序

---

## 数组排序：

- 数组对象.sort(排序规则函数);//返回排序好的数组
- 提示：排序函数必须传递两个参数，返回逻辑值。  
返回true，则第一个参数排序在前，否则在后。
- 提示：在原数组排序，不生成副本

## 数组逆序：

- 数组对象.reverse();//返回顺序相反的数组
- 提示：在原数组逆序，不生成副本。

```
var arr=[8,4,2,7,3,5];  
console.log(arr);//排序前  
function sortrule(a,b){  
    return a-b;//从小到大  
}  
var a=arr.sort(sortrule);  
console.log(a);//排序后  
//在原数组上排序，不生成副本  
console.log(arr);//输出与a相同a==arr  
console.log(arr.reverse());//逆序
```

# 数组元素的操作

- 删除并返回数组的最后一个元素：**pop()**;

```
var arr=[ “Tom” , “Jack” , “Rose” ];  
console.log(arr.pop()); //Rose  
console.log(arr); //["Tom","Jack"]
```

- 向数组的末尾添加一个或更多元素，并返回新的长度：**push(元素1, .....)**;

- 删除并返回数组的第一个元素：**shift()**

```
console.log(arr.unshift( “Obama” , “Clinton” )); //6  
console.log(arr); //["Obama” , “Clinton” , “Tom” , “Jack” ]
```

- 向数组的开头添加一个或更多元素，并返回新的长度：**unshift(元素1, .....)**

- 删除元素，并向数组添加新元素，返回被删除的元素数组：**splice(操作位置，删除个数，添加元素1, ...)**

```
var a=arr.splice(1,2, “Killer” );  
console.log(a); //["Clinton” , “Tom” ]  
console.log(arr); //["Obama", "Killer", "Clinton", "Tom", "Jack"]
```

- 提示：都在原数组上操作。

# 二维数组与多维数组

---

多维数组就是一种嵌套定义：

- 二维数组就是数组的元素是数组。
- 三维数组就是数组的元素是二维数组。
- .....
- $n$ 维数组就是数组的元素是 $n-1$ 维数组。

多维数组的操作：

- 多维数组的操作就是嵌套的反向取值操作。

```
//多维数组的嵌套定义
var arr1=[1,2,3];
var arr2=[[4,5,6],arr1,[7,8]];
console.log(arr2);
//多维数组的嵌套使用
console.log(arr2[1]);//第2个元素是一维数组
console.log(arr2[1][1]);//第2个元素的第2个元素
console.log(arr2.length);//二维数组的长度
console.log(arr2[1].length);//第2个元素的长度
```

# 日期对象

创建日期对象，使用构造函数：

- **var 日期对象=new Date();**//构造函数还可以使用参数
- 提示：参数是从**1970-1-1 0:0:0**开始的毫秒为单位的值。

可以把日期对象转换为字符串：

- 日期对象.**toString()**           转换为字符串。
- 日期对象.**getTimeString()**   转换为时间字符串。
- 日期对象.**toDateString()**   转换为日期字符串。
- 日期对象.**toGMTString()**   转换为GMT格式。
- 日期对象.**toUTCString()**   转换为UTC格式。
- 日期对象.**toLocaleString()** / **toLocaleTimeString()** / **toLocaleDateString()**

转换为本地格式。

```
var d=new Date();
console.log(d);           //Fri Jul 29 2016 11:04:38 GMT+0800 (CST)
console.log(d.toString());//Fri Jul 29 2016 11:04:38 GMT+0800 (CST)
console.log(d.getTimeString());//11:04:38 GMT+0800 (CST)
console.log(d.toDateString());//Fri Jul 29 2016
console.log(d.toGMTString());//Fri, 29 Jul 2016 03:04:38 GMT
console.log(d.toUTCString());//Fri, 29 Jul 2016 03:04:38 GMT
console.log(d.toLocaleString());//2016年7月29日 GMT+8 上午11:04:38
console.log(d.toLocaleTimeString());//GMT+8 上午11:04:38
console.log(d.toLocaleDateString());//2016年7月29日
```

提示：默认调用**toString()**函数转换格式。

# 日期有关的概念

---

## UTC：世界标准时间

- 全称： **Coordinated Universal Time**（协调世界时）
- 说明：协调世界时是原子时（以秒为单位）与世界时（以毫秒为单位）的精度折衷协调解决方案。

## GMT：格林威治标准时间

- 全称： **Greenwich Mean Time**（世界时）
- 说明：格林威治所在地的标准时间，以地球自转为基础的时间计量系统。由于地球自转速度变化的影响，它不是一种均匀的时间系统。后来世界时先后被历书时（牛顿时，均匀时间）和原子时（原子内部发射的电磁振荡频率为基准的时间计量系统）所取代。

## UTC与GMT时间格式的默认表示：

- 星期, 日月简写 年 时:分:秒 GMT。比如： **Fri, 29 Jul 2016 03:04:38 GMT**。

# 日期的字段属性

---

**Date**实现一组属性函数，可以通过日期的字段属性来获取 / 设置日期的相关字段：

- 年： `setFullYear/getFullYear`，返回4位数字的年
- 月： `setMonth/getMonth`，返回0-11
- 每月第几天： `setDate/getDate`，返回1-31
- 星期几： `setDay/getDay`，返回0-6
- 时： `setHours/getHours`，返回0-23
- 分： `setMinutes/getMinutes`，返回0-59
- 秒： `setSeconds/getSeconds`，返回0-59
- 毫秒： `setMilliseconds/getMilliseconds`，返回0-999



# 日期的计量参照点与时区

---

JavaScript日期的计算参照点是：

- 1970年1月1日0时0分0秒

计量单位是：

- 毫秒

日期对象可以使用两种方式返回从1970-1-1 0:0:0到目前的毫秒：

- `getTime()`

- `Date.UTC(年, 月, 日, 时, 分, 秒, 毫秒)` 根据世界时返回。提示：不需要对象访问。

可以获取时区：

- `getTimezoneOffset()`。返回的是分钟格式。

```
console.log(d.getTimezoneOffset());//返回480=60*8
```

# 日期的格式分析

---

**Date**提供一个把字符串转换为日期表示的函数：

■ **Date.parse**(日期字符串);

■ 提示：字符串的格式要正确，否则解析错误，返回**NaN**。

```
console.log(Date.parse(“2016年7月29日”)); //NaN  
console.log(Date.parse(“Jul 29 2016 11:11:11”)); //1469761871000  
console.log(Date.parse(“Jul 29 2016”)); //1469721600000  
console.log(Date.parse(“Fri Jul 29 2016”)); //1469721600000  
console.log(Date.parse(“Fri Jul 29 2016 GMT+0800”)); //1469721600000  
console.log(Date.parse("11:11:11")); //NaN
```

■ 解析好的日期格式是整数，可以转换成**Date**对象进行更加方便的处理。

```
var t=Date.parse("Jul 8 2015");  
var dt=new Date(t); //可以通过构造函数把整数类型的时间转换为Date对象。  
console.log(dt.toLocaleDateString()); //2015年7月8日
```

# Object类型

实际上所有的JavaScript的对象类型都是Object类型的。对象类型是相对于基本类型而言的。

```
var d=new Date();  
var s=new String(999);  
var n=new Number("888");  
var b=new Boolean("false");  
var a=new Array("Tom","Jack","Rose");  
var o=new Object();  
console.log(typeof d);//object  
console.log(typeof s);//object  
console.log(typeof n);//object  
console.log(typeof b);//object  
console.log(typeof a);//object  
console.log(typeof o);//object
```

对象类型

```
var pb=true;  
var pn=20;  
var ps="Hello";  
var pa=[1,2,3,4];  
var po={};  
console.log(typeof pb);//boolean  
console.log(typeof pn);//number  
console.log(typeof ps);//string  
console.log(typeof pa);//object  
console.log(typeof po);//object
```

基本类型与对象类型

原始类型转换为对象类型后，都是对象类型，使用typeof无法获取准确的类型信息。

# 对象类型的原始类型值

其实在JavaScript中所有类型都是对象方式存储，基本类型也可以访问对象类型的属性与函数。我们有几个概念一定要与其它语言区分：基本类型 / 用户类型（复合类型）；原始值 / 包装值。

得到对象的原始类型值有两种方式：

- 对象.valueOf();
- 类型(对象); // 强制转换成原始值

可以把原始类型值包装称用户类型。

- 具体类型包装：var 变量=new Number(初始值);
- 直接使用Object包装：var 变量=new Object(原始类型值);

```
var ob=new Boolean(false);
console.log(typeof ob);//object
var pb1=Boolean(ob);
console.log(pb1);//true
console.log(typeof pb1);//boolean
var pb2=ob.valueOf();
console.log(pb2);//false
console.log(typeof pb2);//boolean
```

原始值与类型

```
var on1=new Number("20");
var on2=new Object("20");
var on3=new Object(20);
console.log(typeof on1);//object
console.log(typeof on2);//object
console.log(typeof on3);//object
console.log(typeof on1.valueOf());//number
console.log(typeof on2.valueOf());//string
console.log(typeof on3.valueOf());//number
```

对象值与类型

# 各种对象类型的原始值类型

---

下面通过一个例子直观认识各种对象类型的原始值类型：

```
var d=new Date();
var s=new String(999);
var n=new Number("888");
var b=new Boolean("false");
var a=new Array("Tom","Jack","Rose");
var o=new Object();
console.log(typeof d.valueOf()); //number
console.log(typeof s.valueOf()); //string
console.log(typeof n.valueOf()); //number
console.log(typeof b.valueOf()); //boolean
console.log(typeof a.valueOf()); //object
console.log(typeof o.valueOf()); //object
```

总结：

- 1、每个对象都有原始值。
- 2、基本类型的值可以返回类型，对象类型不管它原始值类型是什么，都返回object类型。

# 对象类型的其它公共特征

对象的公共属性:

- 返回构造函数: 对象.**constructor**; //返回对构造函数的引用
- 返回原型: 对象.**prototype**; //可以通过该属性添加用户属性与函数

```
var d=new Date();  
var c=d.constructor;  
var c1=new c();  
console.log(c1.getFullYear());
```

对象的基本功能:

- 返回源代码: 对象.**toSource()**;
- 返回字符串描述: 对象.**toString()**;

```
var d=new Date();  
console.log(d.prototype);//undefined  
console.log(Date.prototype);//Invalid Date  
function A(){}  
console.log(A.prototype);//A {}  
console.log({}.prototype);//undefined
```

# null与 undefined值

**null:** (关键字)

- 表示空对象，使用**typeof**返回的字符串是**object**。
- **null**可以表示数值、字符串、对象是“无值”的。

**undefined:** (非关键字，是定义的全局变量)

- 是**null**更近一层表示空值。
- **null**专门用来说明变量，说明变量没有初始化 / 未定义。
- 使用**typeof**返回的字符串是**undefined**。

**注意:**

**null**与**undefined**可以互换的。  
使用**==**运算符返回**true**。

如果想严格区分它们，使用**===**运算符。

```
console.log(null==undefined);//true  
console.log(null===undefined);//false
```

# 原始值与对象值

---

理解原始值与对象值的区别：

■1、原始值不可变，对象值可变。

□20变成21就不是20了。但var d=new Date();中时间可以改变，但还是同一个对象。

■2、原始值比较是值直接比较，对象比较是引用的比较。

□原始值是直接存放，对象存放的是存放数据的空间的地址。

```
var a=20;
var b=20;
var a1=new Number(20);
var b1=new Number(20);
console.log(a==b);//true:原始值比较
console.log(a===b);//true:原始值比较
console.log(a==a1);//true:原始值比较
console.log(a1==b1);//false:对象比较
```



# 掌握运算符符号的4个核心点

---

掌握运算符从4个核心点掌握：

■1.理解运算符功能。

□大部分运算符符合大部分人的数学常识。

■2.掌握参与运算的数据个数与数据类型。

□JavaScript的类型简单，这个环节使用简单，但容易出问题。

■3.掌握运算结果的类型。

□JavaScript的数据类型比较宽松，加上类型自动转换，返回的结果有的时候超出预期结果。

■4.掌握运算过程中参数类型的转换规则。

# 运算符的几个术语与注意点

---

左值：

- 左值的字面含义表示只能出现在赋值运算符左侧的表达式，实际就是表示允许被赋值的表达式；变量、对象属性、数组元素都是属于左值，甚至函数的返回值都可以是左值，但这只允许内置函数，用户函数的返回值不允许是左值的。

运算顺序、结合顺序（方向）与运算符优先级：

- 运算符优先级就是在多个运算符组成的表达式中，有的运算符被优先执行。如果对同等优先级的运算符，按照从**左到右**或者从**右到左**的顺序执行。

对程序执行的影响：

- 所谓对程序的影响是指影响变量或者内存，大部分运算符是对程序没有影响的，但=、++、+=、new等对程序是有影响的，有的影响变量的值，有的产生新的内存分配。

# 二目算术运算

---

运算符号：

■ +(加) -(减)      \*(乘)      /(除)      %(求余)。

参数个数、类型与结果类型：

■ number, number -> number

结合顺序：

■ 从左到右。

注意：

■ %对小数也可以求余。

```
console.log(5/3);//除法对整数不会产生取整操作： 1.6666666666666667  
console.log(10.2%5);//小数也可以求余： 0.19999999999999993
```

■ /对整数不会产生取整的操作（在其他严格类型的语言中存在，比如Java，C等语言）。

■ 其中+ 也是字符串连接运算符。如果运算符中出现字符串，则+优先作为字符串连接符号。

# 字符串->数值类型转换规则

## ■字符串类型转换为数值类型，其中数值类型指的原始值：

- +作为字符串连接运算符，则运算参数中出现字符串数值，数值转换为字符串后再运算。
- 其他算术运算符，则运算参数中出现字符串数值，字符串数值转换为数值后运算。
- 如果想使用确定的数值类型运算，请使用Number函数转换。

```
console.log( "hello" + " " + "world" ); // "hello world" 作为字符串元、链接符号。
console.log( "1" + "2" );                // "12" 字符串链接符号
console.log(1+ "2" );                    // "12" 字符串链接符号

console.log(1+2);                        //3 算术运算符
console.log( "2" - "1" );                //1 算术运算符
console.log( "2" -1);                    //1
console.log( "5" /3);                    // "5" 转换为5

console.log(Number("1")+Number("3"));    //4，使用Number函数强制转换成数值类型。
```

# 对象->数值类型转换规则

---

算术运算符号对对象数值类型是直接支持的：

- 返回结果，直接从对象类型转换为原始值。

```
var a=new Number( "2" );  
var b=new Number( "3" );  
console.log(a+b);//5  
console.log(typeof (a+b));//number, 不是返回object
```

```
var c=4;  
console.log(a+c);//6,返回number类型  
console.log(typeof (a+c));//typeof优先级高于+, 使用（）改变优先级
```

- 如果对象类型与字符串类型参与运算，则对象类型转换为字符串，并作为字符串参与运算。

# 算术运算中参数非法的规则

算术运算的参与参数都要求是数值类型，如果不是数值类型，则会执行类型自动转换，但有的时候会转换失败。一般是下面几种情况：

- `undefined`参与运算。
- 不能转换成数值的字符串参与运算。
- 不能转换成数值的对象参与运算。

结论：

- 转换失败的结果都返回NaN，NaN参与的算术运算结果都是NaN。
- `null`转换为0。
- 日期转换成毫秒。（对象转换是对`valueOf`的结果转换）

```
console.log(2/ "hello" );//NaN
console.log( "3" - "Hi" );//NaN
console.log(null-2);//-2
console.log(undefined-2);//NaN
console.log(2/undefined);//NaN
console.log( "23ABC" -20);//NaN
console.log(NaN-6);//NaN
var d=new Date();
console.log(d-100);//1470034197399
```

# 对象类型转换与valueOf函数

其实所有类型转换前都是先调用**valueOf**函数，并使用它的返回值进行转换，换句话说就是对对象转换为其他类型都是现使用**valueOf**的到原始值在进行转换。

```
var o={  
  a:20,  
  valueOf:function(){  
    return 30;  
  }  
  toString:function(){  
    return "cd";  
  }  
};  
console.log(40-o); //结果10  
console.log(40+o); //70  
console.log("ab"+o); //ab30, 转换为字符串，调用的也不是toString
```

# 算术运算中的Infinity运算规则

---

在算术运算中除了直接使用无穷以外，只有除0可能产生无穷。

■ 无穷不能做除数。

■ 无穷不能求余。

```
console.log(20/0);           //Infinity
console.log(Infinity+Infinity); //Infinity
console.log(Infinity+20);    //Infinity
console.log(Infinity*Infinity); //Infinity
console.log(Infinity*20);    //Infinity
console.log(Infinity/Infinity); //NaN
console.log(Infinity/20);    //Infinity
console.log(Infinity%Infinity); //NaN
console.log(Infinity%20);      //NaN
```



# 一元算术运算符

---

一元算术运算符包含：

- +(正号)                  -(负号)                  ++(自增加) --(自增减)

参与类型与结果类型

- +与-: `number->number;`
- ++与--: 左值->`number;`

结合顺序：

- 从右到左。

注意：

- 其中类型转换遵循字符串与对象到数值类型的转换规则。

# ++与--运算符

---

++与--要求参与运算的表达式必须是左值：

- 这意味着++与--运算只作用于变量、对象属性、数组元素。

++、--与参数表达式的位置关系：

- ++、--在参数表达式前：前增量

- 对表达式进行增量后返回增量后的值。

- ++、--在参数表达式后：后增量

- 对表达式进行增量，返回增量前的值。

```
//console.log(20++);非法，20不是左值。运行错误
var a=20;
var b=new Number(30);
console.log(a++);20打印的是返回值，而不是a的值。
console.log(b++);30
```

```
console.log(++a);22
console.log(++b);32
```

# 位算术运算

---

位算术运算包含：

- `&`(且), `|`(或), `^`(异或), `~`(取反)

备注：

- 位运算就是二进制运算，本质上所有运算都是基于二进制运算的，只不过提供稍微复杂的运算规则而已。位运算因为基于二进制运算，同等效果下，采用位运算的速度要优于其他类型的运算。
- `~`是单目运算。

参数类型与结果类型：

- `&`、`|`、`^` 是： `int`, `int->int`。    `~`是： `int->int`

结合顺序：

- `&`、`|`、`^`结合顺序从左到右，`~`的结合顺序是从右到左。

# 理解位运算

## ■&运算

□对应的位全部是1，则取1，否则取0。

00010100  
00000111  
结果: 00000100

## ■|运算

□对应的位全部是0，则取0，否则取1。

00010100  
00000111  
结果: 00010111

## ■^运算

□对应的位不同则取1，否则取0

00010100  
00000111  
结果: 00010011

## ■~运算

□对每个位操作，是1则取0，是0则取1。

00010100  
结果: 11101011

```
console.log(20.toString(2));  
console.log(7.toString(2));  
console.log((20&7).toString(2));//100  
console.log(20.toString(2));  
console.log(7.toString(2));  
console.log((20|7).toString(2));//10111  
console.log(20.toString(2));  
console.log(7.toString(2));  
console.log((20^7).toString(2));//10011  
console.log(20.toString(2));  
console.log(~20.toString(2));//-10101
```

其中11101011是负数，需要取反加1，的到结果是-10101

**注意：**这里需要整数的二进制表示的一些知识，尤其是负数的二进制表示方法。

# 位移运算符

JavaScript提供三个位移运算符

- <<(左移), >>(带符号右移), >>>(右移)

参数类型与结果类型:

- int , int->int

结合顺序:

- 从左到右。

注意:

- 位移的第二目参数在位移前会先与**32**求余, 使用余数作为位移的位数。

```
console.log(20<<2);//80
console.log(20>>2);//5
console.log(20>>>2);//5
console.log(-20>>2);//-5
console.log(-20>>>2);//1073741819
console.log(20>>34);//5,等价于位移2
```

# 理解位移运算符

---

## ■ << 位移运算符

- 第一个参数整体向**左**位移第二个参数指定的位数。
- 左边越界的位，自动截断；右边空出的位，自动填充0。

## ■ >> 位移运算符

- 第一个参数整体向**右**位移第二个参数指定的位数。
- 左边越界的位，自动截断；右边空出的位，负数填充1，整数填充0。这样整数位移后为整数，负数位移后依然是负数。所以称带符号位移。

## ■ >>> 位移运算符

- 第一个参数整体向左位移第二个参数指定的位数。
- 左边越界的位，自动截断。右边空出的位，自动填充0。这样不管什么数位移后，首位都变0，成为正数。

# 位运算符中的类型转换

下面通过例子来说明转换规则：

- 小数的转换：自动截断
- 字符串的转换：能转的自动转，不能转的都转成NaN。
- 对象的转换。（调用valueOf返回原始值再转换）

下面类型参与运算都返回0

- null。
- undefined。
- Infinity。
- NaN。

```
console.log(20.5>>2);//5，没有四舍五入
console.log("20">>2);//5
console.log(null>>2);//0
console.log(undefined>>2);//0
console.log(Infinity>>2);//0
console.log(NaN>>2);//0
console.log("20"&7); //4
console.log("20"&"7"); //4
console.log(Infinity&7);//0
console.log(20&Infinity);//0
console.log(20&null);//0
console.log(undefined&7);//0
console.log(NaN&7);//0
```

# 关系运算符

---

JavaScript的常规关系运算符:

■ >(大于), >=(大于等于), <(小于), <=(小于等于), !=(不等), ==(相等)

参数类型与结果类型:

■ number,number->boolean: 数字比较

■ string,string->boolean: 字母表顺序比较

结合顺序

■ 从左到右

```
console.log(2>4);//false  
console.log( "hello" <= "world" );//true  
console.log(new Number(20)<new Number(10));//false
```



# 恒等与非恒等关系运算符符号

---

JavaScript特殊的关系运算符：

- `===`(恒等), `!==`(非恒等)

参与类型与结果类型：

- `any, any -> boolean`。

结合顺序：

- 从左到右。

# 关系运算符中的类型转换

对关系运算符`<`,`<=`,`>`,`>=`中参数的类型分成如下几种情况：`==`与`!=`单独讨论

## ■ 1.数值类型与数值类型:

□ 数值比较

## ■ 2.字符串类型与字符串类型:

□ 字符串比较

## ■ 3.数值类型与字符串类型:

□ 数值比较，如果字符串转换成数值失败，则永远返回`false`。

## ■ 对象类型参与关系比较:

□ 对象使用`valueOf`的返回值进行比较，并遵循上面的三种规则。

```
console.log(077<10);//false: 数值比较
console.log("077"<10);//false:数值比较
console.log("077"<"10");//true: 字符串比较
console.log(077<"10");//false: 数值比较
```

```
console.log(10>"1a");//false: 字符串转换失败
console.log(10<="1a");//false: 字符串转换失败
```

```
var o={
  a:999,
  valueOf:function(){//对象比较被调用
    console.log("valueOf");
    return "abc";
  }
};
console.log(10<o); //false
console.log("abd">o);//true
```

# 恒等关系元算符号中的类型转换

恒等比较中不存在类型转换的情况（不存在`valueOf`调用的情况），主要用来比较两个对象是否是同一对象的引用。其遵循如下比较规则：

- 如果两个类型不相等，则返回`false`。
- 如果两个值都是`null`，或者`undefined`，则返回`true`。
- 如果其中有一个`NaN`，或者两个都是`NaN`，则返回`false`。
- 如果两个参数都是数值，且数值相等，则返回`true`，否则`false`。
- 如果两个参数都是字符串，且每个字符的16个位完全相同，则返回`true`。
- 如果两个参数都是对象，对象指向同一对象，则相等，否则计算属性完全一样也返回`false`。一个不是对象，则肯定返回`false`。
- 如果两个参数都是`false`或者`true`，则返回`true`。

```
var a1=new Number(10);
var a2=new Number(10);
console.log("hello"==="hello");//true
console.log(20===20);//true
console.log(null===null);//true
console.log(undefined===undefined);//true
console.log(null===undefined);//false
console.log(NaN===NaN);//false
console.log(Infinity===Infinity);//true
console.log(a1===a2);//false
console.log(new Number(30)===30);//false
```

# ==关系运算符的比较

==与!=关系运算符与>,>=,<,<=四个运算有点差异，它遵循如下比较规则：

- 如果是数值与数值，则比较值。
- 如果是字符串与字符串，则比较字母顺序。
- 如果是数值与字符串，则比较数值，字符串转换为数值。
- 如果是数值与对象，则比较数值，对对象的valueOf结果转换后比较。
- 如果是字符串与对象，则比较字符串，对对象的valueOf结果转换后比较。
- 如果是对象与对象，则与恒等规则一样。
- 如果有逻辑值，则转换为0或者1比较。

```
var a1=new Number(10);
var a2=new Number(10);
console.log( "hello" == "hello" );//true
console.log(20==20);//true
console.log(null==null);//true
console.log(undefined==undefined);//true
console.log(null==undefined);//true
console.log(NaN==NaN);//false
console.log(Infinity==Infinity);//true
console.log(a1==a2);//false
console.log(new Number(30)==30);//true
```

# in关系运算符

JavaScript提供了一个对象的属性关系判定运算符in.

- 属性字符串 in 对象;

参数类型与结果类型:

- string,any->boolean

结合顺序:

- 从左到右

注意:

- 属性建议使用字符串, 如果是数字, 会自动转换成字符串。
- 适用于成员属性, 成员函数, 数组下标。

## //4.用户自定义类型

```
var o={
    1:20,
    a:30,
    "b":40
};
var a="a";
//使用字符串, 则报错。
console.log(a in o);//true不
console.log(1 in o);//true
```

```
//"Hello"不是对象,不能使用in
//但能访问length属性。
//console.log("length" in "Hello");
var s=new String("Hello");
//1.属性判定
console.log(length in s);//true
console.log("length" in s);//true
//2.函数判定
console.log("toString" in s);//true
console.log(toString in s);//false
```

```
var arr=[7,8,9];
//3.数组
console.log(0 in arr);//true
console.log(3 in arr);//false
```

# instanceof与对象的类型关系判断

---

语法:

■ 对象 instanceof 类型

参数类型与结果类型

■ object, function-> boolean

结合顺序

■ 从左到右

注意:

■ 第一个参数必须是对象，第二必须是构造器函数名。

■ 因为使用typeof对对象只能返回object，使用instanceof可以判定对象的具体类型。

```
console.log("Hello" instanceof String);//false:"Hello"不是对象
var s=new String("Hello");
console.log(s instanceof String);//true;
//判定对象类型的标准模式
console.log(typeof s=="object" && s instanceof String);
```

# 逻辑运算符

---

JavaScript的逻辑运算符与大部分语言一样，支持三大逻辑运算：

- **&&(且), ||(或), !(否)**

参数类型与结果类型：

- **&&与||**： **boolean,boolean->boolean**

- **!**： **boolean->boolean**

结合顺序：

- **&&与||**： 从左到右。

- **!**：从右到左。

```
console.log(true && false); // false
var b1 = new Boolean("true");
var b2 = new Boolean(false); // "false" 会转换为 true。
console.log((b1 || b2).valueOf()); // true: 对象类型与对象类型，返回对象类型。
console.log(b2 || true); // 返回对象类型
console.log(true || b2); // 返回原始类型
```

# 逻辑运算符的转换规则

因为在逻辑运算中要求的运算参数必须是逻辑类型，如果参数不是逻辑类型，则都会自动转换为逻辑类型，其转换规则与Boolean构造器的转换规则一致：

- 数值0与-0，转换为false，其他转换为true；
- 字符串""转换为false，其他转换为true；
- null，undefined，NaN返回false；

注意：

- 其中&&与||的运算规则有点差异。
- &&与||存在运算短路的情况。
- 其中仅管返回值不是true或者false，但不影响。
- ! 总是返回true或者false。

```
console.log((0&&true));//数值:0
console.log((100&&true));//数值:true
console.log((""&&true));//字符串:""
console.log(("A"&&true));//字符串:true
console.log((null&&true));//null:null
console.log((undefined&&true));//undefined:undefined
console.log((Infinity&&true));//Infinity:true
console.log((NaN&&true));//NaN:NaN
var o={
    valueOf:function(){
        console.log("valueOf");
        return true;
    }
};
//&&: true, 没有调用valueOf, ||:调用了valueOf,返回对象
console.log((o||true));
```



# 赋值复合运算符

赋值复合运算符就是与赋值运算符结合的运算符：

- `+=`, `-=`, `*=`, `/=`, `%=`

- `<<=`, `>>=`, `>>>=`

- `&=`, `|=`, `^=`, `~=`

```
var a=20;  
a+= "hello" ;  
console.log(a);//20hello, 字符串优先
```

说明：

- 所有赋值复合运算符：左值`op`=表达式；等价于：左值=左值`op`表达式。

结合顺序：

- 从右到左。

类型转换规格：与前面对应的运算符一样。

# 三目运算符

---

语法:

■ 逻辑表达式? 值1:值2;

参数类型与结果类型:

■ boolean,any,any->any

结合顺序:

■ 从右到左。

注意:

■ 第二个与第三个参数根据第一个参数决定是否运算。

```
var a=20;  
console.log(true?"hello":++a);//hello  
console.log(a);//20, ++a没有运算。
```

# 其他运算符

在JavaScript中提供几个运算符：

- **delete** 对象.属性与函数：删除对象的属性与函数或者数组元素。(对内置对象无效)
- **typeof** 对象或者**typeof(对象)**：返回变量类型。
- **void** 表达式：总是返回**undefined**，但表达式照样计算。(经常使用的a标签的href属性中:javascript:void 表达式);
- **逗号**：表达式1，表达式2；先计算左操作数，然后计算右操作数，返回右操作数结果。

```
//1.void运算符  
var a=20;  
var b=void ++a;  
console.log(a);//21  
console.log(b);//undefined
```

```
//2.1.delete运算符  
var o={v:20,  
      m:function(){}  
}  
console.log("v" in o);//true  
delete o.v;  
console.log("v" in o);//false
```

```
//3.逗号  
console.log((a,++a));//22
```

```
//2.2.delete数组  
var arr=[1,2,3];  
console.log(0 in arr);  
delete arr[0];  
console.log(0 in arr);
```

# Math数学运算

---

JavaScript还提供了一个内置对象Math用来实现常见数学运算：

■两个常量：

□Math.PI：圆周率

□Math.E：自然数

■常见数学运算：

□开方运算。

□三角 / 反三角函数运算。

□指数 / 对数函数运算。

□随机数函数运算。

□四舍五入，取整，取绝对值，最大最小，角度弧度转换函数运算。

# 常见Math数学运算例子

<code>console.log(Math.PI);</code>	<code>//圆周率</code>
<code>console.log(Math.E);</code>	<code>//自然数</code>
<code>console.log(Math.sin(Math.PI/6));</code>	<code>//正弦</code>
<code>console.log(Math.atan(1));</code>	<code>//反正切</code>
<code>console.log(Math.pow(3,3));</code>	<code>//指数</code>
<code>console.log(Math.log(10));</code>	<code>//对数</code>
<code>console.log(Math.log(100)/Math.LN10);</code>	<code>//对数：10为底</code>
<code>console.log(Math.log(100)/Math.LN2);</code>	<code>//对数：2为底</code>
<code>console.log(Math.exp(3));</code>	<code>//指数或者幂</code>
<code>console.log(Math.random());</code>	<code>//随机</code>
<code>console.log(Math.round(0.51));</code>	<code>//四舍五入</code>
<code>console.log(Math.floor(0.51));</code>	<code>//取整</code>
<code>console.log(Math.ceil(0.51));</code>	<code>//取整</code>
<code>console.log(Math.abs(-10));</code>	<code>//绝对值</code>
<code>console.log(Math.max(10,30));</code>	<code>//最大</code>
<code>console.log(Math.min(10,30));</code>	<code>//最小</code>
<code>console.log(Math.sqrt(4));</code>	<code>//平方根</code>

# 关于运算中的浮点数问题

---

任何语言中都存在浮点数的精确表示问题，大家在应用的时候需要注意，尤其是小数比较运算的时候：

```
var a=0.3-0.2;  
var b=0.2-0.1;  
console.log(a==b);//返回false，这是精度造成的问题
```

# eval表达式运算

在Javascript提供了一个函数eval用来计算字符串类型的表达式，只支持一个参数：

- 如果参数是字符串，则编译并执行字符串中代码，并返回最后一个表达式的结果，如果最后表达式没有返回值，则返回undefined。
- 如果参数不是字符串，则直接返回。
- 注意：eval的计算被上下文的环境变量影响。等于使用字符串代替源代码。

```
var a=10,b=20;
console.log(eval(a+b));           //直接返回： 30
console.log(eval( "a+b" ));       //编译执行： 30
console.log(eval(a+b,a++,b-a));   //多个表达式，直接识别第一个参数： 30
console.log(eval( "a+b,a++,b-a" )); //返回最后一个： 8
//console.log(eval(a+b;a++;b-a)); //非法的参数，分号语句结束，产生歧义
console.log(eval( "a+b;a++;b-a" )); //多个语句： 返回最后语句返回值： 7
console.log(eval( "alert( 'Hello' )" )); //返回undefined
```

# 复合语句与空语句

---

在编程语言中，有两种特殊的语句：

- 空语句：只有一个分号的语句，称为空语句。空语句在某些时候是非常有用的。
- 复合语句：使用 { } 把多个语句联合在一起，称为复合语句。复合语句是使用最频繁的语法。

```
//复合语句
{
    var a=20;
    document.write(a);
}
//空语句
;
```

- 注意：复合语句影响着其中定义的变量，函数，类的作用域（范围）。



# if语句

---

使用if语句可以控制复合语句是否被执行，其语法如下：

■ **语法1：** `if (逻辑表达式) {复合语句};`

□ 当逻辑表达式为true或者等价于true，则复合语句得到执行。

□ 当逻辑表达式为false，则跳过复合语句。

■ **语法2：** `if (逻辑表达式) {复合语句1} else{复合语句2};`

□ 当逻辑表达式为true，则执行第一个复合语句并跳过第二个复合语句；

□ 当逻辑表达式为false，则跳过第一个复合语句并执行第二个复合语句。

■ **注意：** 在if语句中，当复合语句中只有一个语句，复合语句的 { } 可以省略。复合语句也可以使用空语句。

# if语句例子

---

## //if语句

//例子1

```
if(true){  
    document.write(“语句被执行” );  
}
```

//例子2

```
if(2>3){  
    document.write(“语句没有被执行” );  
}
```

## //空语句在if中的使用

```
if(true);
```

## //if/else语句

```
if (true) {  
    document.write(“复合语句1” );  
} else{  
    document.write(“复合语句2” );  
};
```

## //空语句在if/else中的使用

```
if (false);  
else{  
    document.write(“空语句” );  
};
```

# if/else的嵌套使用

```
//第一层  
if (false) {  
}  
else{
```

```
//第二层  
if (false) {  
}  
else{
```

```
//第三层  
if (false) {  
}  
else{  
};
```

```
};
```

嵌套排版变形



```
//第一层  
if (false) {  
}  
else
```

```
//第二层  
if (false) {  
}  
else
```

```
//第三层  
if (false) {  
}  
else{  
};
```

嵌套排版整理



```
if (false) {  
}  
else if (false) {
```

```
}  
else if (false) {
```

```
}  
else{  
};
```

复合语句只有一个语句省略 { }

# switch语句

语法:

空switch语句，中不能放语句

■ 空switch语句:

```
switch(表达式){  
  
}
```

■ 分段switch语句:

```
switch(表达式A){  
    case 表达式1:  
        语句;  
    case 表达式2:  
        ....  
    default:  
        语句;  
}
```

**执行流程:**

1. 计算表达式A的值，并用表达式的返回值与switch的每个case的值进行比较，如果比较结果为true，则从该case段中的语句开始执行。

如果要中断switch语句的执行，使用break语句。如果没有break语句，则会依次执行之后的所有case段中语句。

2. 如果所有的case都比较返回false，则从default开始执行。

1. switch使用case与default分成多段，语句不能直接放在switch块中，必须放在case或者default段中。

2. case与default段都是可选的。

# switch语句例子

没有break的情况:

```
var a=10;
var b=20;
switch(a){
  case 10:
    document.write("10"+"<br>");
  case 20:
    document.write("20"+"<br>");
  case b+10:
    document.write("30"+"<br>");
  default:
    document.write("other"+"<br>");
}
```

10  
20  
30  
other

使用break的情况:

```
var a=10;
var b=20;
switch(a){
  case 10:
    document.write("10"+"<br>");
    break;
  case 20:
    document.write("20"+"<br>");
    break;
  case b+10:
    document.write("30"+"<br>");
    break;
  default:
    document.write("other"+"<br>");
    break;
}
```

# 理解switch中比较

---

在switch中表达式的比较使用的是===，而不是==，下面例子可以说明：

```
switch(null){  
  case undefined:  
    document.write( “==” );  
    break;  
  default:  
    document.write( “===” );  
    break;  
}
```

执行结果：输出===。

# while循环语句

---

**while**语句提供一种语法结构，可以让一个复合语句块反复执行。

语法：

```
while(逻辑表达式){  
    语句;  
    ....  
};
```

复合语句可以是空语句：

```
while(false);
```

提示：

- 当复合语句只有一个语句，则可以省略 { }

**执行流程：**

1. 计算逻辑表达式的值。
2. 如果返回值为**true**，则执行复合语句块。
3. 执行复合语句块后，重新回到步骤1开始。
4. 如果返回值**false**，则跳过复合语句块。

# while语句的执行过程例子

```
console.log("1.循环开始");  
var i=0;  
while(console.log("2.辑表达式计算"),i<3){  
    console.log("3.循环语句");  
    i++;  
}  
console.log("4.循环结束");
```

1.循环开始

2.辑表达式计算

3.循环语句

2.辑表达式计算

3.循环语句

2.辑表达式计算

3.循环语句

2.辑表达式计算

4.循环结束

循环重复执行了3次。

逻辑值为false，结束循环。

`console.log(“2.辑表达式计算” ),i<3`是一个语句，返回最后表达式的值。



# do/while循环语句

语法:

```
do{  
    语句;  
    ....  
}while(逻辑表达式);
```

执行流程:

- 1.从do开始执行;
- 2.执行复合语句;
- 3.复合语句执行完毕, 计算逻辑表达式;
- 4.如果表达式值返回true, 重新回到1;
- 5.如果表达式值返回false, 则继续往下执行。

提示:

- 其中复合语句只有一个语句, 则 {} 可以省略。
- do与while的区别, while先判定, do是至少执行一次。

```
do                                     //省略 {}  
    document.write( “do/while” );  
while(false);
```

# do/while循环语句执行过程例子

```
console.log( "1.循环开始" );  
var i=0;  
do{  
    console.log( "2.循环语句(至少一次)" );  
    i++;  
}while(console.log( "3.辑表达式计算" ),i<3);  
console.log("4.循环结束")
```

1.循环开始

2.循环语句(至少一次)

3.辑表达式计算

2.循环语句(至少一次)

3.辑表达式计算

2.循环语句(至少一次)

3.辑表达式计算

4.循环结束

循环重复执行了3次。

逻辑值为false，结束循环。

# for循环

---

语法:

```
for(语句1;逻辑表达式2;语句4){  
    语句3;  
    ....  
};  
语句5;
```

提示:

■ **for**中的复合语句块中只有一个语句，可以省略 { }

执行流程:

- 1.for开始，首先执行语句1；
- 2.计算逻辑表达式的值；
- 3.逻辑表达式返回**false**，则直接跳过复合语句块，执行语句5；
- 4.逻辑表达式返回**true**，则执行复合语句块中语句；
- 5.复合语句执行完毕，然后执行语句4；
- 6.语句4执行完毕，重新从2开始。

# for循环执行过程例子

```
console.log("1.开始循环");  
var i=0;  
for(console.log("2.语句1");console.log("3.语句2"),i<3;console.log("5.语句4")){  
    console.log("4.语句3");  
    i++;  
}  
console.log("6.循环结束");
```

1.开始循环

2.语句1

3.语句2

4.语句3

5.语句4

3.语句2

4.语句3

5.语句4

3.语句2

4.语句3

5.语句4

3.语句2

6.循环结束

循环  
3  
次

# for/in循环语句

for/in语法:

```
for(左值 in 对象){  
    语句;  
    .....  
}
```

提示:

- 如果复合语句块中只有一个语句，{ } 可以省略。
- 其中左值，一般要求是变量，不能是常量与常值。
- 如果in的后面是数值，则直接跳出循环。
- 日期对象本质也是数值类型的。

执行流程:

1. 首先判定对象;
2. 如果对象是null或者undefined，则跳过复合语句;
3. 如果对象不是null与undefined，则判定是否原始值;
4. 原始值是字符串，则循环字符串字符下标;
5. 原始值是数值，则跳过复合语句块。
6. 如果是对象，则循环取对象的属性，赋值给左值，然后执行复合语句块，直到属性取完后跳过复合语句块。

```
var a=20;  
for(var i in a){    //不会循环  
    console.log(i);  
}
```

# for/in遍历数组与字符串对象

---

```
//遍历数组  
var arr=[ “Jack” , “Tom” , “Rose” ];  
for(i in arr){  
    console.log(i);  
}  
//输出0, 1, 2
```

数组的本质是对象，其属性就是下标。

```
var s=“Hello”;  
var o2=new String("Hello");  
for(i in s){  
    console.log(i);  
}
```

字符串的本质是字符数组。属性是下标。

# for/in遍历其它对象

---

```
//遍历console对象
for(i in console){
    console.log(i);
}
//遍历window对象
for(i in window){
    console.log(i);
}
```

可以使用这种方法来罗列某对象支持的属性或者特征。

# 中断语句

---

中断语句可以中断当前执行点，跳转到另外一个执行点，用来控制程序的执行流程。

JavaScript提供三种中断方式：

- **continue;** : 必须用在循环中，用来从**continue**所在位置跳转到循环开始位置。
- **break;** : 必须用在循环中，用来从**break**所在位置跳转到循环结束的位置。
- **return**或者**return 值;** : 必须用在函数中，用来结束函数过程，并返回一个值（如果**return**带值的话）。

提示：

- **continue**与**break**用在其他地方浏览器会抛出异常（**break**还有用在**switch**中的特殊情况）。



# break与continue的循环跳转说明例子

```
while(console.log("开始点"),true){  
    console.log("被循环的语句");  
    break;  
}  
console.log("循环结束点");
```

输出结果：  
开始点  
被循环的语句  
循环结束点

```
var isOn=true;  
while(console.log("开始点"),isOn){  
    console.log("被循环的语句");  
    isOn=!isOn;  
    continue;  
}  
console.log("循环结束点");
```

输出结果：  
开始点  
被循环的语句  
**开始点**  
循环结束点

## 结论：

break是直接跳出循环。  
continue是跳到循环开始处。

# 正则表达式运算

在目前很多语言中都引入了模式匹配运算：正则表达式运算。这种运算用来在字符串中检索变得非常简便，JavaScript提供了一个内置对象`RegExp`来实现正则表达式运算。

语法：

- 对象值表示法：`var reg=new RegExp(模式字符串, 模式标记字符串);`
- 字面值表示法：`var reg=/模式/模式标记;` [注意：不要使用字符串表示，直接表示]

说明：

```
var r1=/is/g;  
var r2=new RegExp("is","ig");
```

- 模式用来指定匹配方式，可以是正则表达式，也可以是内容字符串。
- 模式标记用来修饰匹配方式，只对正则表达式有效，对内容字符串无效。
- 模式标记可以指定三个字符串：`"g"`(匹配所有，而非第一个终止)`"i"`(不区分大小写)`"m"`(多行匹配)

# 正则表达式的方法使用

使用正则表达式： (RegExp核心只有三个方法)

- 正则表达式对象.compile(模式字符串, 标记字符串);
  - 返回正则表达式字符串表示。用来修改匹配模式与标记。
- 正则表达式对象.test(检索字符串);
  - 返回逻辑值true / false。用来检测正则表达式是否能检索成功。
- 正则表达式对象.exec(检索字符串);
  - 返回被找到的字符串数组, 没有找到返回null。用来执行检索。

```
var str="this is a regulat express test  
for a very long long string.IS this right?";  
//检索is  
var r1=/is/g;  
// 检索is, 不区分大小写  
var r2=new RegExp("is","ig");  
console.log(r1.test(str));  
console.log(r2.test(str));  
console.log(r1.exec(str));  
console.log(r2.exec(str));
```

输出:

true
true
["is"] (1)
["is"] (1)

# 设置/获取检索开始位置

**RegExp**中有一个可写/可读属性用来设置正则表达式的检索开始位置，同时返回上次检索后的第一个字符位置，只对**g**标记的检索有效。

■ 正则表达式对象**lastIndex**=整数；

□ 如果**lastIndex**大于检索字符串长度，则检索结果返回**null**，**lastIndex**设置为0。

□ 如果正则表达式再也检索不到内容，则检索结果返回**null**，**lastIndex**设置为0。

```
var str="this is a regulat express test for a very long long string.IS this right?";
var r1=/is/g;           //检索is
var r2=new RegExp("is","ig");//检索is，区分大小写
var re;
do{
    re=r2.exec(str);
    console.log("检索结果:"+re+";位置:"+r2.lastIndex);
}while(re!=null);
```

输出：

检索结果:is;位置:4

检索结果:is;位置:7

检索结果:IS;位置:61

检索结果:is;位置:66

检索结果:null;位置:0

# 正则表达式对象的几个只读属性

---

RegExp对象还有几个只读的属性（不能写）：

- **global**: 逻辑类型，RegExp 对象是否具有标志 **g**。
- **ignoreCase**: 逻辑类型，RegExp 对象是否具有标志 **i**。
- **multiline**: 逻辑类型，RegExp 对象是否具有标志 **m**。
- ~~**unicode**: 逻辑类型，RegExp 对象是否具有标志 **u**。~~
- ~~**sticky**: 逻辑类型，RegExp 对象是否具有标志 **y**。从lastIndex指定的值开始检索。~~
  
- **unicode**与**sticky**有的浏览器可能不支持

# 正则表达式的几个静态属性

---

在`RegExp`类中提供静态的属性来获取检索的上下文：

- 输入： `RegExp.input`或者`RegExp.$_`
- 已检索的内容： `RegExp.leftContext`或者`RegExp["$'"]`(就是~下的字符)
- 未检索的内容： `RegExp.rightContext`或者`RegExp["$'"]`(单引号)
- 分组匹配的内容： `RegExp.$1`， ....., `RegExp.$9`
- 最近一次匹配的内容： `RegExp.lastMatch`或者`RegExp["$&"]`
- 最近一次括号选择匹配的内容： `RegExp.lastParen`或者`RegExp["$+"]`

提示：

- 上述静态属性都是非标准属性，在正式的web开发中不要使用，但目前主流浏览器都支持。

# 正则表达式模式

---

正则表达式模式是：由一组字符序列组成。其中字符分成两类：

- 直接语义字符：按照字面含义匹配。比如a，就是检索字符a。
- 特殊语义字符：有特殊的匹配含义。比如\d，就是检索数字；也称元字符。还有其他一些符号（^，\$，{}，[]，()，/，\，.，:，!，|，=等），特殊语义字符分成如下几种作用：
  - 用来指定匹配内容，也称字符类；
  - 用来指定匹配的重复次数，也称量词；
  - 用来指定匹配的位置。
  - 用来对匹配模式分组 / 引用 / 选择。

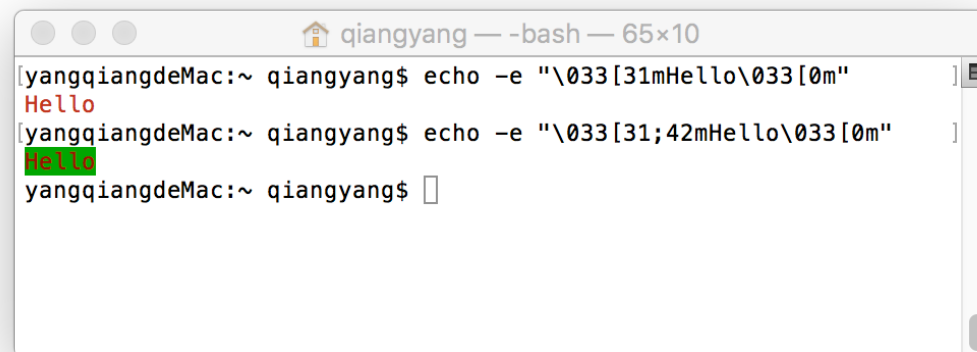
# 直接语义字符

直接语义字符包含如下情况：

■ 字母与数字 匹配自身；

■ 转义字符：

- \0 匹配空字符
- \t 匹配制表符
- \n 匹配换行符
- \v 匹配垂直制表符
- \f 匹配换页符
- \r 匹配回车符
- \xnn 匹配16进制的ASCII字符
- \unnnn 匹配16进制的Unicode字符
- \cx 匹配控制字符（虚拟终端下常用的控制输出方式的字符）



```
qiangyang — -bash — 65x10
[ yangqiangdeMac:~ qiangyang$ echo -e "\033[31mHello\033[0m"
Hello
[ yangqiangdeMac:~ qiangyang$ echo -e "\033[31;42mHello\033[0m"
Hello
[ yangqiangdeMac:~ qiangyang$
```

控制终端输出颜色的就是控制字符。  
（**window**的控制台不支持控制字符）



# 直接语义字符例子

---

```
var str="中国汉字(025-12345678)与English.JavaScript and Java is a    good web \
programming language! Java includes JavaSE,JavaME and JavaEE";
var reg=new RegExp("Java","g");      //1.测试ASCII字母数字直接量
//var reg=new RegExp("中国","g");    //2.测试Unicode字母数字直接量
//var reg=new RegExp("中\u56fd","g"); //3.测试Unicode直接量:匹配"中国"
//var reg=new RegExp(" \\t ", " g"); //4.字符串可以使用续航符，注意： 新行有三个\t
var r;//返回结果
do{
    r=reg.exec(str);
    console.log(reg.lastIndex+ “:” +r);
    console.log(RegExp.rightContext);//用来直观观察检索的过程
    //console.log(RegExp.input);    //匹配test返回true才改变
    console.log(RegExp.lastMatch);
}while(r);//r转换成逻辑值， null转换为false。
```

# 分类字符

---

- `[.....]` : 可以使用枚举`[abcdef]`, 与范围表示`[a-f]`, 匹配一个指定范围内的字符。
- `[^.....]` : 匹配一个指定范围以外的字符。
- `.` : 匹配除换行符以外的任意字符。
- `\w` : 任何**ASCII**字符组成的单词, 等价于`[a-zA-Z0-9]`
- `\W` : 任何非**ASCII**字符组成的单词, 等价于`[^a-zA-Z0-9]`
- `\s` : 任何**Unicode**空白符。**ASCII**空白字符也算。
- `\S` : 任何非**Unicode**空白符的字符。
- `\d` : 任何**ASCII**数字字符, 等价于`[0-9]`
- `\D` : 除**ASCII**数字字符以外的任何字符, 等价于`[^0-9]`
- `[\b]` : 退格

**提示:** 在字符串中\要使用\\转义。

# 分类字符例子

var str= “Hello我的电话号码是025-12345678，我的手机是 13388888888，请记住我的E文名:Louis Young。 ” ;

```
//var reg=new RegExp( “[0-9][0-9]” , “g” );//[]02,12,....
```

```
//var reg=new RegExp( “[^a-z]” , “g” );
```

```
var reg=new RegExp( “\\S” , “g” );//记得在字符串中转义。
```

```
//var reg=/\w/g;//这个不需要转义
```

```
//var reg=new RegExp(“\\s”,“g”);
```

```
//var reg=new RegExp(“\\w”,“g”);
```

```
//var reg=new RegExp(“\\W”,“g”);
```

```
//var reg=new RegExp(“\\d”,“g”);
```

```
//var reg=new RegExp(“\\D”,“g”); var r;
```

```
do{
```

```
    r=reg.exec(str);
```

```
    console.log(r);
```

```
}while(r);
```

# 匹配次数

---

- $\{m,n\}$ :  $m \leq \text{匹配次数} \leq n$
- $\{m,\}$  :  $m \leq \text{匹配次数}$
- $\{m\}$  :  $m == \text{匹配次数}$
- $?$  : 匹配次数为0次, 或者1次, 等价于  $\{0,1\}$
- $+$  : 匹配次数至少1次, 等价于  $\{1,\}$
- $*$  : 匹配次数可以任意次, 等价于  $\{0,\}$
- 注意: 匹配次数使用在匹配字符后。比如:  $\backslash w\{1,3\}/$ ,  $/[0-1]\{5\}/$

# 匹配次数例子

```
var str= “我的电话号码是025-12345678， a 我的手机是 13388888888， 请记住我的E文名:Louis Young。” ;  
//var reg=new RegExp( “[0-9][0-9]” , “g” );//[02,12,...  
//var reg=new RegExp( “[^a-z]” , “g” );  
//var reg=new RegExp( “\\W*” , “g” );//记得在字符串中转义。汉字  
//var reg=/\w*/g;//这个不需要转义  
//var reg=new RegExp( “\\w*” , “g” ); //西文  
var reg=new RegExp( “\\W{2,4}” , “g” ); //汉字  
var reg=new RegExp( “\\W{2,4}?” , “g” );//汉字  
var r;  
do{  
    r=reg.exec(str);  
    console.log(r);  
    console.log(reg.lastIndex);  
}while(r && r!="");
```

## 注意：

- 1、次数匹配默认是贪婪匹配，就是尽可能多的匹配，比如{2,4}就优先匹配4。
- 2、如果在次数后缀?，则非贪婪匹配，就是尽可能少的匹配。比如{2,4}就优先匹配2。

# 匹配选择

---

## 选择1:

- 匹配模式1 | 匹配模式2 | .....
- 匹配规则：左边优先，当左边匹配成功后，右边的就停止匹配。

## 选择2:

- 匹配模式1(匹配模式2)
- 匹配规则：匹配模式2可以作为匹配，也可以不作为匹配。
- 注意： ()的用途除了用作匹配选择，还可以用作分组，并且引用。

# 匹配选择例子

---

//匹配选择

```
var str= “我们学习正则表达式很难吗？请拨打电话025-12345678, Studying regular express is very difficult! please mail to:38395870@qq.com to study” ;
```

//选择1

```
//var reg=new RegExp( “[s|S]tudy” , “g” );//Study study
```

//选择2 ( ) 的模式也会作为数组返回。

```
var reg=new RegExp( “[s|S]tudy(ing)?” , “g” );//[ “Studying” , “ing” ]
```

```
do{
```

```
    r=reg.exec(str);
```

```
    console.log(r);
```

```
    console.log(reg.lastIndex);
```

```
}while(r && r!="");
```

# 匹配分组与引用

---

- `(.....)` : 将几个匹配项组合成一个单元, 可以使用`*`,`+`,`?`,`|`修饰, 并可以记住与这个组合相匹配的字符串供后面引用。
- `(?:.....)` : 将几个匹配项组合成一个单元, 不提供匹配字符串的引用。
- `\n` : 和第`n`个分组第一次匹配的字符串匹配。组是`()`定义的分组。



# 匹配分组与引用例子

```
var str= “Studying regular express is very difficult! please mail to: ‘38395870@qq.com’ to study” ;  
//分组选择  
//1.(.....)  
//var reg=new RegExp( “[s|S]tudy(ing)?” , “g” );//[ “Studying” , “ing” ]  
//2.(?:.....)  
//var reg=new RegExp( “[s|S]tudy(?:ing)?” , “g” );//[ “Studying” ]  
//3.\n, 从1开始  
var reg=new RegExp( “( ‘)[^’ ]*\1” , “g” );//检索 ‘ ’ 界定的字符串, 记得使用转义符号  
//匹配结果: ‘38395870@qq.com’  
do{  
    r=reg.exec(str);  
    console.log(r);  
    console.log(reg.lastIndex);  
    console.log(RegExp.$1);//返回第一个匹配的模式字符串  
}while(r && r!="");
```

**提示:** 只能支持\$1到\$9共9个引用。

# 匹配位置

---

也称匹配位置为匹配的锚：用来指定匹配发生的合法位置与特定位置。

- `^` : 匹配字符串的开头位置，在多行检索中，匹配一行的开头位置。使用在前面（前缀修饰）。
- `$` : 匹配字符串的结尾位置，在多行检索中，匹配一行的结尾位置。使用在后面（后缀修饰）。
- `\b` : 匹配单词的边界位置，就是`\w`(或者`\W`)之间的位置。
- `\B` : 匹配非单词边界的位置
- `(?=p)` : 要求后继字符与`p`匹配，但不包含匹配`p`的字符。使用在后面（后缀修饰）
- `(?!p)` : 要求后继字符不与`p`匹配。使用在后面（后缀修饰）

# 匹配位置例子

---

```
var str= “Studying regular express is very difficult! please mail to:38395870@qq.com to study” ;  
//^  
//var reg=new RegExp( “^Studying” , “g” );//检索行头的Studying字符串.  
//$  
//var reg=new RegExp( “study$” , “g” );//检索行尾的study.  
//\b  
//var reg=new RegExp( “\b\w{1,}\b” , “g” );//检索单词  
//(?=p)  
var reg=new RegExp( “\w*(?=@)” , “g” );//检索@前的字符串  
do{  
    r=reg.exec(str);  
    console.log(r);  
    console.log(reg.lastIndex);  
}while(r && r!="");
```

# 在字符串中的正则表达式使用

---

在字符串中有如下几个函数可以使用正则表达式操作：

- 整数索引=字符串.**search**(正则表达式);
- 匹配的数组结果=字符串.**match**(正则表达式);
- 新字符串= 字符串.**replace**(正则表达式|子串, 替换子串|替换函数);
- 分割后的数组结果=字符串.**split**([分割符号或者正则表达式 [, 数组限制]]);

# 使用正则表达式搜索字符串

**String**对象的**search**函数整数索引=字符串.**search**(正则表达式) ) 的搜索规则:

- 根据参数指定的正则表达式搜索，搜索到第一个匹配字符串，则返回匹配字符串的第一个字符索引（整数）。
- 如果搜索失败，则返回-1。
- 提示：该函数适合提供匹配的位置定位。无法确定内容与内容的长度。

```
var str= “this is a good boy! is not bad boy!” ;  
var reg=/\bis\b/g;      //单词is  
var r, s1=str;  
while((r=s1.search(reg))!=-1){  
    console.log(r);  
    s1=s1.substring(r+ “is” .length); //定位在匹配的第一个字符，偏移查找的字符  
}
```

# 使用正则表达式检索匹配结果

---

**String**对象的**match**（匹配的数组结果=字符串.**match**(正则表达式)）可以直接返回匹配的结果字符串，结果可能是多个，使用数组返回。

■ 如果匹配失败则返回**null**。

```
//字符串匹配match  
var str= “this is a good boy! is not bad boy!” ;  
var reg=/\b\w*\b/g;    //匹配任意长度的单词  
var r;  
r=str.match(reg);  
console.log(r);
```

# 使用正则表达式替换字符串

---

**String**对象的**replace**函数（新字符串= 字符串.**replace**(正则表达式|子串, 替换子串|替换函数)）使用正则表达式替换。

- 返回的结果是替换后的字符串；如果没有匹配成功的，返回原字符串。
- 替换的内容可以直接使用子串表示，也可以使用控制函数返回结果替换。

//字符串替换

```
var str="this is a good boy! is not bad boy!";
```

```
var reg=/\b\w{1,}\b/g; //单词匹配
```

```
var r;
```

```
r=str.replace(reg,function(val){
```

```
    return "["+val+";
```

```
});
```

```
console.log(r);      //返回结果: [this] [is] [a] [good] [boy]! [is] [not] [bad] [boy]!
```

# 使用正则表达式切分字符串

---

**String**对象的**split**函数（分割后的数组结果=字符串.**split**([分割符号或者正则表达式[, 数组限制]])）也可以使用正则表达式分隔。

■ 返回分隔后的字符串数组；如果没有分隔，则返回整个字符串。

```
var str= “this is a good boy;that is not bad boy!I am a girl.” ;  
var reg=/[!;.]/g;    //句子匹配  
var r;  
r=str.split(reg);  
console.log(r); //返回！；.分隔的句子，最后返回一个空字符串。
```



# 常用的匹配正则表达式

---

- 输入长度检测: `/.{6,}/`
- 电话号码检测: `/\d{3}-\d{8}|\d{4}-\d{7,8}/`
- 电子邮件检测: `/^\w+([-+.] \w+)*@\w+([-.] \w+)*\.\w+([-.] \w+)*$/`
- HTTP URL检测: `/^http://([\w-]+\.)+[\w-]+(/[\w-./?%&=]*)?$/`
- QQ 号码检测: `/[1-9][0-9]{4,}/` //从10000开始
- 身份号码检测: `/^[1-9]\d{5}[1-9]\d{3}((0\d)|(1[0-2]))(((0|1|2)\d)|3[0-1])\d{3}(\d|x|X)$/`
- 邮政编码检测: `/^[1-9]\d{5}$/`
- IP 地址检测: `/\b(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b/`
- 日期输入检测: `/^\d{4}-\d{1,2}-\d{1,2}/`

# 09编程思想

目的：

- 1、掌握面相过程的编程思想。
- 2、掌握面相对象的编程思想。

1、函数

2、类与构造函数

3、继承与原型

4、**object**中内置约定的属性

5、**eval**函数与**JSON**对象

# 函数的定义

---

函数本质一段可以反复调用的复合语句，提供输入参数与返回值，其语法如下：

■ **function** 函数名(参数1,.....){语句; .....};

说明：

■ 参数可以在函数中使用，用来在调用的时候传入值，函数中的参数就是调用者传入的值。

■ 如果要输出数据给调用者，则使用**return** 数据值；函数可以没有返回值，则不需要使用**return**。

```
function myfunc(a,b){  
    console.log(“用户传入的阐述” +a+ “:” +b);  
    return a+b;//把调用者传递的数据，进行求和处理，返回给调用者  
}
```

# 调用函数

---

调用函数语法:

■ 函数名(数据, 数据, .....);

如果有返回数据, 可以使用变量接收存储

■ 变量=函数名(数据, .....);

```
function myfunc(a,b){  
    console.log("用户传入的阐述"+a+":"+b);  
    return a+b;//把调用者传递的数据, 进行求和处理, 返回给调用者  
}  
var re=myfun(45,55);           //函数调用  
console.log(re);
```

# 理解函数的参数

---

函数的参数的作用域：

- 函数的参数的作用域只在函数范围内有效，称为临时变量。函数调用结束就被释放。

函数参数的传递类型：

- 传值：参数是原始值（拷贝：函数内参数与调用者传入的参数是不同的存储空间）。
- 传引用：参数是对象值（引用：函数内参数与调用者传入的参数是同一个参数）。

```
//传对象
function func_obj(a){
    if(typeof(a)=="object" && a instanceof Date){
        console.log("日期对象值");
        a.setTime(a.getTime()+7*24*60*60*1000);//增加7天
    }
}
var vd=new Date();
console.log(vd);
func_obj(vd);
console.log(vd);//前后输出不同。
```

```
//传原始值
function func_val(a){
    if(typeof(a)=="number"){
        console.log("原始值");
        a=30;
    }
}
var va=10;
console.log(va);
func_val(va);
console.log(va); //前后输出相同
```

# 函数的返回值

函数的返回值有下面几点需要注意：

- **JavaScript**要求用户函数的返回值都是右值，内置函数有的是左值的。
- 函数如果没有返回值，也可以使用变量接收，返回的是**undefined**。

```
function myfunc(){  
    console.log("没有返回值");  
}  
var re=myfunc();  
console.log(re);//undefined
```

```
function myfunc(){  
    var a=20;  
    return a;  
}  
myfunc()=30;//不能是左值
```

把函数作为左值，浏览器抛出的错误：

! ▶ ReferenceError: Left side of assignment is not a reference.

# 函数变量

在Javascript中函数也是一种类型：

- 1.是原始值，使用**typeof**返回“function”；
- 2.可以把函数赋值给一个变量，该变量的类型也是“function”；
- 3.函数变量与函数一样调用。
- 4.函数可以使用**object**包装，包装以后的函数还是**function**类型，并可以调用；
- 5.函数可以作为参数传递，并使用。

```
//2.函数变量
var f=myfunc;
console.log(typeof(f));//function
//3.函数变量使用
var re=f();
console.log(re);
```

```
//4.函数的包装类型
var of=new Object(myfunc);
console.log(typeof(of));
//调用函数对象
re=of();
console.log(re);
```

```
function myfunc(){
    var a=20;
    return a;
}
//1.函数的类型
console.log(typeof(myfunc));//function
```

```
//5.myf是函数参数
function func(myf){
    if(typeof(myf)==="function"){
        var r=myf();
        console.log("通过参数调用函数");
    }
}
func(f); //传递函数给函数的参数。
```

# 函数数字面值与匿名函数

在JavaScript中，不仅可以把定义好的函数赋值给函数变量，还可以直接把函数在定义的同时赋值给函数变量：

■ `var 函数变量=function(参数){.....};`

注意：

- 函数不需要名字，称为匿名函数或者函数数字面值。
- 匿名函数可以直接作为参数传递。

```
//匿名函数，函数数字面值  
var f=function(a,b){  
    return a+b;  
};  
//调用函数  
var r=f(45,55);  
console.log(r);
```

```
//函数直接作为参数传递  
var arr=[3,1,5,7,2,4,6];  
arr.sort(function(a,b){return a-b;});  
console.log(arr);    //[1,2,3,4,5,6,7]
```



# 在定义函数的时候直接调用

---

可以在定义函数的时候直接调用：

■ 有名调用

■ 匿名调用

```
//1.有名定义调用  
var r1=function myfunc(a,b){  
    return a+b;  
}(45,55);  
console.log(r1);
```

```
//2.匿名定义调用  
var r2=function(a,b){  
    return a+b;  
}(45,55);  
console.log(r2);
```

# 函数嵌套

---

Javascript与Java语言有一个不同的地方，就是函数的定义可以嵌套：

- 在函数中定义函数，也称局部函数。
- 嵌套函数只能在定义的范围内调用。

```
//函数嵌套
function outter_func(){
    //局部定义函数
    function inner_func(){
        console.log(“内部函数”);
    }
    //局部函数调用
    inner_func();
}
//调用外部函数
outter_func();
//不能在外部调用内部函数
//inner_func();//浏览器报错，函数没有定义
```

# 嵌套函数作为返回值

---

在外部无法调用函数内部定义的函数，但嵌套函数可以通过外部函数的返回值，在外部调用。

```
function outter_func(){  
    function inner_func(){  
        console.log("嵌套函数");  
    }  
    return inner_func;  
}  
//直接调用  
outter_func()();  
//使用函数变量调用  
var f=outter_func();  
f();
```

# 变量的作用域

在JavaScript中，一个 { } 块就表示一个作用域，变量的作用域遵循如下规则：

- 1、内部可以访问外部，外部不能访问内部；
- 2、如果变量名内部与外部同名，局部优先。
- 提示：这个规则也适用于if/while等控制块。
- 提示：<script>下直接定义的变量为全局变量。

```
var a=20;
function outter_func(){
    var b=30;      //只在当前与嵌套的{}有效
    function inner_func(){
        var b=50;
        var c=40;
        console.log(a);
        console.log(b); //当前作用域的b优先
        console.log(c);
    }
    //不能访问c
    inner_func();
}
//不能访问b
outter_func();
```

# 嵌套函数中变量的问题

## 问题描述:

### ■前提:

- 1.内部函数访问 / 修改外部函数中变量。
- 2.外部函数把内部函数最为返回值返回。
- 3.调用外部函数返回内部函数，并调用内部函数。

### ■问题:

- 当调用内部函数时，外部函数已经结束，其定义的变量已经释放，调用内部函数时，内部函数访问的这些外部函数定义的变量会出现什么问题？

```
//函数嵌套中的变量作用域问题
function outter_func(){
    var v=20;//外部函数定义的变量
    function inner_func(){
        v++;//1.访问外部的函数定义的变量
        console.log(v);
    }
    return inner_func;//2.返回内部函数
}
var f=outter_func();//返回内部函数
//3.调用内部函数
f(); //21
f(); //22 给人感觉v成了全局变量
```

# 函数闭包

---

- 嵌套函数中外部函数中局部变量因为内部函数的访问，不会在外部函数调用结束后释放（变量从分配到释放称为一个生命周期）。
- 嵌套函数中外部函数中局部变量的生命周期因为内部函数的访问而发生改变的现象，就是一种叫闭包的技术造成。
- 闭包：每个函数是一个对象，每个对象函数的执行依赖一个作用域或者作用域链。函数内部的变量保存在作用域内，这个作用域就称为闭包。只要函数还在调用，这个作用域与作用域链就会起作用。也就是函数定义时闭包（作用域）在函数调用时依然有效。

# 函数的参数与arguments对象

在每个函数中，每个参数声明都是可选的（可以不用声明，但调用的时候，可以传递参数），传递的参数可以通过arguments对象访问，arguments对象与数组非常相似：

- 访问参数个数：arguments.length;
- 访问参数：arguments[下标]; //下标从0开始
- 还可以通过arguments判定期望的参数个数：arguments.callee.length（期望参数在定义函数声明的参数）。

```
function myfunc(a){//期望参数1个
    console.log(“参数实际个数: ” +arguments.length);
    console.log(“参数期望个数: ” +arguments.callee.length);
    if(arguments.length>0){
        console.log(arguments[0]);//输出第一个参数
    }
}
myfunc(1,2); //实际参数2
myfunc();//实际参数0
myfunc(3,4,5,6,7);//实际参数5
```

# 函数对象与函数的其他调用方式

---

所有的函数都是函数对象，所有函数对象有两个方法：

- 函数对象.call(函数所属对象, 参数1, .....);
- 函数对象.apply(函数所属对象, 参数数组);
- 提示：在JavaScript中每个函数都是成员函数，全局函数也有一个所属对象，就是全局对象。全局对象没有名字，但可以在全局作用区域通过关键字this访问。

```
function myfunc(a,b){  
    console.log( “call” );  
    return a+b;  
}  
var r1=myfunc.call(this,45,55);  
console.log(r1);  
var r2=myfunc.apply(this,[45,55]);  
console.log(r2);
```



# 字典

---

在JavaScript中可以定义一种key-value格式的集合数据，传统意义上称为字典，JavaScript称为对象，key称为属性：

- 定义字典： `var 变量={key:value, ..... , key:value};`
- 使用字典：字典变量[key];
- 字典是传统的一种数据结构，提供数据的增删修查操作。

```
var d={1: "Jack" ,2: "Tom" ,3: "Rose" ,age:20};
console.log(d[ "age" ]);//字符串需要使用“”或者‘’防治与变量歧义
//字典访问
console.log(d[1]);      //获取
d[1]= "Killer" ;        //修改
d[0]= "Clinton" ;       //添加
delete d[1];            //删除
for(v in d){            //遍历
    console.log(v);
}
```

# 对象

---

在JavaScript中采用字典集合的语法来表示对象，并提供对象的访问语法：

- 对象.属性
- 提示：建议属性定义都采用字符串。
- 提示：访问属性不能使用字符串。

```
var o={  
    "name":"Louis",  
    "age":20,  
    "score":99.50,  
    "grow":function(){  
        console.log("grow called!");  
    }  
};  
console.log(o.name);    //获取  
o.name= "Jack" ;       //修改  
o.favor= "Swim" ;      //新增  
delete o.score;        //删除  
for(v in o)  
    console.log(v);     //遍历  
o.grow();               //调用方法
```

# JavaScript的面向对象

JavaScript的面向对象与其他语言有比较大的差别，其他语言采用类块的语法来描述与实现对象，JavaScript采用的语法是：

- 构造器函数（创建对象）
- 原型（实现继承）

提示：

- 内置对象的原型都是返回**undefined**。
- 内置对象不能被继承/重写。

**//1.不要通过类型访问**

```
console.log(Date.constructor);//function Function(){}  
console.log(Date.prototype);//Invalid Date
```

**//2.通过对象访问**

```
var dt=new Date();  
console.log(dt.constructor);//function Date()  
console.log(dt.prototype);//undefined
```

**//使用构造函数创建对象**

**//1.直接使用**

```
var d1=new Date();
```

**//2.间接使用**

```
var c=d1.constructor;  
var d2=new c();  
console.log(d2);
```

# 构造函数与类

---

定义构造函数的语法：（与函数定义没有差异）

```
function 构造函数名(参数列表,.....){  
    //定义成员  
}
```

提示：

- 构造函数可以不带参数，也可以带参数。
- 构造函数定义函数也称为：类

使用构造函数创建对象：

```
var 对象变量=new 构造函数(参数, .....);
```

```
//无参构造函数  
function Animal(){  
    }  
//带参构造函数  
function Human(type,nation){  
    }  
//构建对象  
var o1=new Animal();  
var o2=new Human("亚洲","中国");  
    console.log(o1,o2);
```

# 成员变量与初始化

在字面值对象中，可以使用字典的语法定义成员变量，在构造函数中定义成员变量，必须与局部变量区分，使用“**this.**”前缀区分,不使用**var**:

■ **this.**成员变量;

```
function Animal(){
    this.type; //默认初始化undefined
    this.nation= “日本” ;
}
function Human(type,nation){
    this.type=type;    //使用参数初始化
    this.nation=nation;
}
```

```
//成员调用
var o1=new Animal();
var o2=new Human( “亚洲” , “中国” );
console.log(o1,o2);
console.log(o1.type);//uncdefined
console.log(o1.nation);
```

# 成员函数

---

成员函数与成员变量一样，只是其值是一个函数：

```
function Human(type,nation){  
    this.type=type;    //使用参数初始化  
    this.nation=nation;  
    this.change=function(nation){           //定义成员函数  
        this.nation=nation;               //访问成员变量记得使用this。  
    }  
}
```

```
var o2=new Human( “亚洲” , “中国” );  
o2.change( “中国梦” );           //调用成员函数  
console.log(o2.nation);
```

# 类成员与实例成员

---

- 在构造函数通过**this**定义的变量与函数，都必须通过**new**创建的对象调用，这称为实例变量与函数，也称为成员。
- 可以定义直接使用类调用的变量与函数，称为类变量与类函数，比如：**Math.PI**与**Math**中的运算函数。类变量与函数实现在构造函数外部实现。

```
//在构造函数外定义成员
Human.mym=function(a,b){
    return a+b;
}
var o2=new Human("亚洲","中国");
var r=o2.mym(45,55);//调用错误，无法识别函数
console.log(Human.mym(45,55));
```

# 认识prototype与类

使用字典的方式定义的对象，每个对象都是独立的属性集合。为了共享定义，JavaScript引入了原型的概念，每个实例化对象都从原型继承属性。

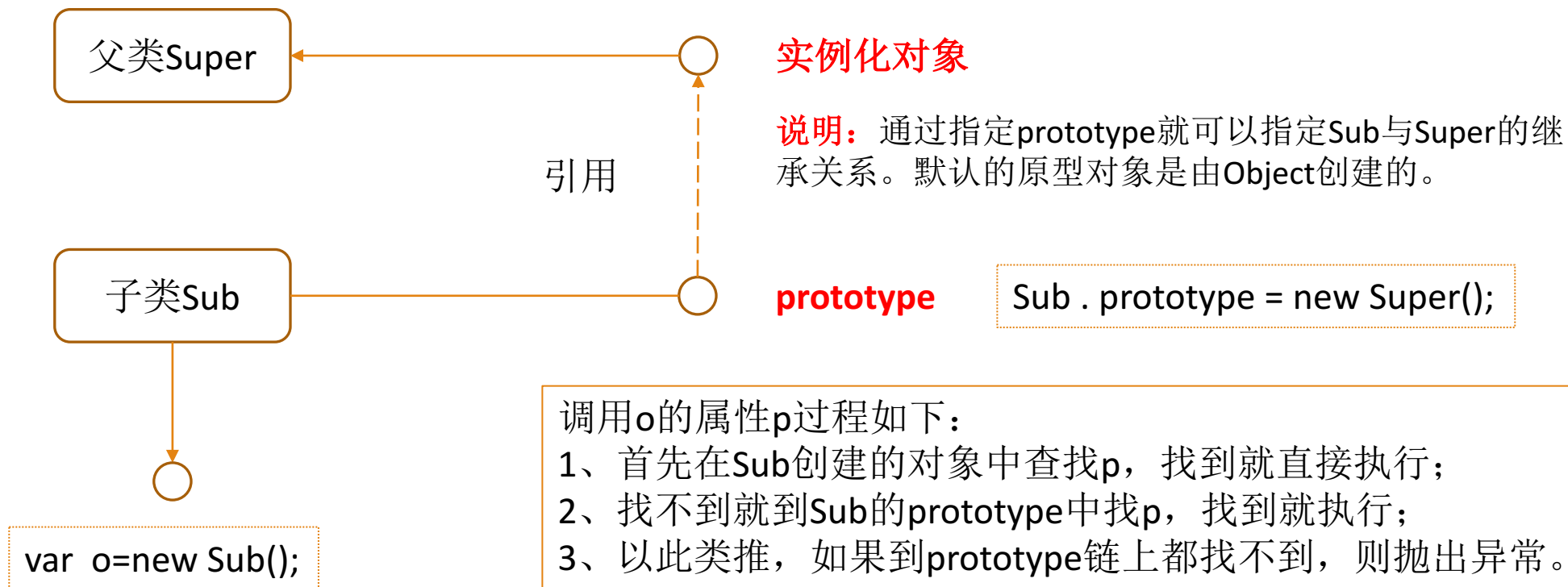
- **prototype**本身是一个对象，是类的核心。
- **prototype**本身是类的一个实例。
- 默认情况下**prototype**就是**Object**类的实例。
- 用户可以定义类的**prototype**。
- 内置类的**prototype**是**undefined**的。

```
function Human(type,nation){
    this.type=type; //使用参数初始化
    this.nation=nation;
    this.change=function(nation){
        this.nation=nation;
    }
}
console.log(Human.prototype);           //Human{}
console.log(typeof(Human.prototype));    //object
console.log("toString" in Human.prototype); //true
console.log("nation" in Human.prototype); //false
```



# prototype与继承

下面用一个图来诠释prototype的工作原理以及继承的原理



# 使用prototype实现继承的例子

## //1、独立的父类

```
function Super(param){  
    this.sup_property=param;  
    this.sup_method=function(){  
        console.log(this.sup_property);  
    }  
}
```

## //2、独立的子类

```
function Sub(param){  
    this.sub_property=param;  
    this.sub_method=function(){  
        console.log(this.sub_property);  
    }  
}
```

**提示：**通过prototype建立父子关系，需要在子类实例化之前。最好是在子类构造函数之前。

## //3、通过prototype建立父子关系

```
Sub.prototype=new Super("hello");
```

```
var sub=new Sub( "World" );  
sub.sub_method();  
//下面语句运行会抛出错误。  
sub.sup_method();
```

指定prototype前

```
var sub=new Sub( "World" );  
sub.sub_method();  
//可以通过sub调用sup的属性  
sub.sup_method();
```

指定prototype后

# prototype解决的两个问题

**prototype**非常巧妙地解决了面向对象中的两个关键点：（等价于Java中的**super**关键字）

- 子类继承父类，子类实例化的时候，父类属性的实例化问题。
- 父类与子类中属性名冲突的时候，通过**prototype**解决了调用歧义问题。

```
function Super(param){  
    this.name= "Louis" ;  
    this.property=param;  
    this.method=function(){  
        console.log(this.property);  
    }  
}
```

```
var sup=new Super( "Hello" );  
//建立父子关系  
Sub.prototype=sup;  
//子类  
function Sub(param){  
    this.property=param;    //与父类同名  
    this.method=function(){ //与父类同名  
        //调用父类属性（冲突）  
        console.log(Sub.prototype.property);  
        //访问父类属性 （不冲突）  
        console.log(this.name);  
    }  
}
```

```
var sub=new Sub( "World" );  
//sub的method方法是父类还是子类？  
sub.method();  
Sub.prototype.method();
```



# 使用with来简化对象属性访问

在JavaScript中引入了with关键字，来简化对对象的属性访问，语法如下：

■ with(对象或者类){ ..... };

■ 说明：

□ 在with块中，访问对象属性，可以省略对象前缀与属性调用符号。

□ 在类定义内部，同样可以对实例的引用this使用。

## 1、类的情况

```
with(Math){  
    console.log(sqrt(4));  
}
```

## 2、对象的情况

```
var o=new Date();  
with(o){  
    console.log(getFullYear());  
}  
with(new Date()){  
    console.log(getFullYear());  
}
```

## 3、类内部this的情况

```
function Cls(){  
    this.name= "Louis" ;  
    this.meth=function(){  
        with(this){  
            name= "Killer" ;  
        }//出现歧义，还得用this  
    }  
}  
var obj=new Cls();  
obj.meth();  
console.log(obj.name);
```

# 利用对象来实现代码模块化

---

代码模块化的最终好处是可以规模化开发，这需要解决两个问题：

- 分散代码管理
- 全局变量冲突（污染）

在JavaScript中，没有引入模块语法，但可以使用对象来技巧实现：

- 构建一个简单对象，并用它作为命名空间。
- 通过对象属性扩展来实现代码。

```
var myspace={}; //命名空间名
myspace.Animal=function(){
    this.name= "Japan" ;
    this.act=function(){
        console.log( "Ho!ho!..." );
    }
}
var o=new myspace.Animal();
console.log(o.name);
o.act();
```

# 判定对象的类型

到目前为止判定对象的类型，可以使用两种方式：

- `instanceof`

- 对象的`constructor`属性

提示：

- `constructor`是构造函数，返回的是函数对象。

```
var myspace={}; //命名空间名
myspace.Animal=function(){
    this.name="Japan";
    this.act=function(){
        console.log("Ho!ho!...");
    }
}
var o=new myspace.Animal();
console.log(o.name);
o.act();
if(o instanceof myspace.Animal){
    console.log("True");
}
if(o.constructor==myspace.Animal){
    console.log("True");
}
```

# 对象序列化与反序列化

---

对象序列化指把对象转换为其他格式，比如字符串与二进制；把其他格式再转换为对象，称为反序列化；一般存在与两个地方：

- 对象存储
- 对象网络传输

在JavaScript中由于有两个API函数的存在，对象的序列化与反序列化可以简单实现：

- JSON内置对象：字符串与对象之间转换。
- 全局eval函数：提供把字符串转换为对象，包含数组。



# 使用eval反序列化对象

使用eval把字符串代码编译执行，返回对象；需要使用”（”与”）”界定：

■ `var 对象=eval("("+字符串+")")`;

■ 提示：字符串必须符合对象与数组字面值格式。

//对象

```
var strObj = '{' ;  
    strObj+=' a:"louis",';  
    strObj+=' b:"Jack";  
    strObj+='}';  
var obj=eval("("+strObj+")");  
console.log(obj.a);
```

//数组

```
var strArr = '[1,2,3,4,5]' ;  
var arr=eval( "(" +strArr+ ")" );  
console.log(arr[0]);
```

**提示：**对象数组，有数组属性的对象都可以转换。

# JSON.stringify说明

---

语法: **JSON.stringify(value [, replacer] [, space]);**

返回: **JSON**格式的字符串 (**JavaScript Object Notation**)

- **value** : 被转换的对象
- **replacer**: 控制转换结果的函数或数组。
- **space**: 类型是数值, 用来添加缩进、空白和换行符, 使转换后文本便于阅读。
  - 如果省略 **space**, 返回值文本生成, 没有任何额外的空白。
  - 如果 **space** 是数字, 则返回值具有空白的文本缩进指定数目在每个级别的。
  - 如果 **space** 大于 10 时, 文本缩进 10 个空白。
  - 如果 **space** 为非空字符串, 如 “\t”, 返回值文本缩进与字符串的字符在每个级别。
  - 如果 **space** 为大于 10 个字符的字符串, 使用前 10 个字符。

# JSON格式字符串规范

---

## 基本规范:

- 数据在名称/值对中
- 数据由逗号分隔
- 花括号保存对象
- 方括号保存数组

```
var employees = [  
  { "firstName":"Bill" , "lastName":"Gates" },  
  { "firstName":"George" , "lastName":"Bush" },  
  { "firstName":"Thomas" , "lastName":"Carter" }  
];
```

## 其中名称 / 值对规范:

- 名称/值对包括字段名称（**在双引号中**），后面写一个冒号，然后是值:

## 属性值的类型规范:

- 数字（整数或浮点数）/字符串（在双引号中）/逻辑值（**true** 或 **false**）/数组（在方括号中）/对象（在花括号中）/null

# 使用JSON默认序列化对象

被序列化的长剑三个类型的对象：

```
var d=new Date();    //内置对象
var o={              //用户对象
    "name":"Louis",
    "age":20,
    "sex":true,
    "grow":function(){
        age++;
    }
};
var a=["Jack","Tom","Rose");//数组对象
```

默认序列化：

```
var strD=JSON.stringify(d);
var strO=JSON.stringify(o);
var strA=JSON.stringify(a);
console.log(strD);//"2016-08-05T01:28:47.480Z"
console.log(strO);//{"name":"Louis","age":20,"sex":true}
console.log(strA);//["Jack","Tom","Rose"]
```

默认序列化：不指定序列化控制函数与数组，不指定拍版。

**注意：**函数是不会序列化的。

# 使用JSON控制序列化输出排版

---

```
var strD=JSON.stringify(d,null,2); //指定2个空格缩格  
var strO=JSON.stringify(o,null,2);  
var strA=JSON.stringify(a,null,2);  
console.log(strD);
```

```
"2016-08-05T01:51:33.851Z"
```

```
console.log(strO);
```

```
{  
  "name": "Louis",  
  "age": 20,  
  "sex": true  
}
```

```
console.log(strA);
```

```
[  
  "Jack",  
  "Tom",  
  "Rose"  
]
```

# 使用数组控制序列化输出内容

■ 设置第二个参数为数组，用数组过滤控制输出内容。

□ 数组中存在的对象属性就序列化输出。

```
var arrString=["name","age"];  
var arrIndex=[0,2];  
var strD=JSON.stringify(d,arrIndex,8);  
var strO=JSON.stringify(o,arrString,8);  
var strA=JSON.stringify(a,arrIndex,8);  
console.log(strD);  
console.log(strO);  
console.log(strA);
```

```
"2016-08-05T02:12:51.133Z"
```

```
{  
  "name": "Louis",  
  "age": 20  
}
```

```
[  
  "Jack",  
  "Tom",  
  "Rose"  
]
```

**提示：**日期与数组没有属性，不受数组的过滤影响。

# 理解控制函数工作原理

控制函数接受两个参数，并且返回一个结果，这个函数的理解的关键点在于知道该函数什么时候被调用，什么时候终止调用。

- 理解控制函数什么时候被调用，首先把对象看成一个文档树，对象本身是root节点，每个属性是一个子节点。
- stringify函数遇见每个节点就调用控制函数，然后根据控制函数返回的结果是否是枚举对象，来决定控制函数的递归调用，如果返回对象不是枚举类型，则终止递归调用。下面使用例子说明

```
var o={  
  "name":"Louis",  
  "age":20,  
  "sex":true,  
  "grow":function(){  
    age++;  
  }  
};
```

- 1、首先o是第一个判定的对象，调用控制函数，key为""字符串，value为对象。
- 2、如果控制函数返回的对象不可枚举，则stringify结束解析。输出前面的结果。
- 3、如果控制函数返回的是可枚举对象，则循环调用控制函数，取枚举属性key: value，传递给控制函数。
- 4、根据控制函数的返回值，重复步骤2与3。

# 使用控制函数控制序列化输出内容

## 处理要求:

- 1.对象与函数采用默认处理。
- 2.如果string, number, boolean, null, undefined值直接添加 [] 。

```
function handle(key,value){
    if(typeof(value)== "object" || typeof(value)== "function" ){
        return value;
    }
    else{
        return "[" +value+ "]" ;
    }
}
var strD=JSON.stringify(d,handle,8);
var strO=JSON.stringify(o,handle,8);
var strA=JSON.stringify(a,handle,8);
console.log(strD);
console.log(strO);
console.log(strA);
```

对对象o，控制函数调用5次，o一次，4个属性各一次。

```
"[2016-08-05T03:49:44.441Z]"
{
  "name": "[Louis]",
  "age": "[20]",
  "sex": "[true]"
}
[
  "[Jack]",
  "[Tom]",
  "[Rose]"
]
```

key用来识别属性，不是属性的，key为""（不是null与undefined）。



# JSON.parse说明

---

语法: `JSON.parse(text [, reviver]);`

返回: JavaScript对象

- **text** : 一个有效的 JSON 格式字符串。
- **reviver** : 一个转换结果的函数。 将为对象的每个成员调用此函数。 参数是属性名与属性值。

# 使用JSON.parse反序列化

反序列化的字符串：JSON字符串

```
var str='';
str+='{'
str+='  "country":"中国",'
str+='  "birth":"OCT 1 1949",'
str+='  "provinces":'
str+='    ['
str+='      {'
str+='        "name":"江苏",'
str+='        "city":"南京",'
str+='        "counties":"[苏州,无锡,常州,徐州,淮安,宿迁]"';
str+='      },'
str+='    {'
str+='      "name":"河北",'
str+='      "city":"石家庄",'
str+='      "counties":"[保定,唐山,邢台]"';
str+='    }'
str+='  ]'
str+='}'
```

```
var o=JSON.parse(str);
//访问对象
console.log(o.country);
console.log(o.birth);
console.log(o.provinces[0].name);
```

输出：

中国

OCT 1 1949

江苏

# 控制反序列化内容

## 控制要求:

把其中的日期转换为Date类型。

```
function handler(key,value){
    if(key== "birth" ){
        var time=Date.parse(value);
        return new Date(time);
    }
    else{
        return value;
    }
}
var o=JSON.parse(str,handler);
//访问对象
console.log(o.country);
console.log(o.birth);
console.log(o.provinces[0].name);
```

## 输出:

---

中国

---

Sat Oct 01 1949 00:00:00 GMT+0800 (CST)

---

江苏

---

可以用于个人学习，如果要用于商业用途，请联系我：

QQ: 38395870

电话: 13338629985