Boolean类型的字面值只有两个，但其他所有类型的值都有与这两个Boolean值等价的值

|  |  |  |
| --- | --- | --- |
| 数据类型 | 转换true | 转换false |
| Boolean | true | false |
| String | 任何非空字符串 | “”(空字符串) |
| Number | 任何非零数字值 | 0、NaN |
| Object | 任何对象 | null |
| Undefined | n/a | undefined |

NaN(not a Number)：非数值是一个特殊的数值，表示本来要返回数值的操作数未返回数值的情况（这样不会抛出错误，因此不会影响其他代码的执行）

NaN与任何值都不相等，包括NaN本身

isNaN()：确定参数是否“不是数值”

console.log(isNaN(NaN)); //true

console.log(isNaN(10)); //false

console.log(isNaN(“10”)); //false(可以被转换成数字)

console.log(isNaN(“blue”)); //true

console.log(isNaN(true)); //false(可以被转换成1)

数值转换：Number() 、parseInt() 、parseFloat()

Number()转换规则：

true = 1 false = 0

null = 0

undefined = NaN

“123456” = 123456

“zzzz” = NaN

“” = 0

String类型：字符串是不可变的，要改变某个变量保存的字符串，首先要销毁原来的字符串，然后再用另一个包含新值的字符串填充该变量。

Object类型：一组数据和功能的集合。

Object的每个实例都有下列属性和方法：

Constructor：保存着（指向）创建当前对象的函数

hasOwnProperty()：

isPrototypeOf()

…………

操作符：

递增、递减

++10 --10 先操作本身++ --,再操作其他（优先级高）

10++ 10—先操作其他运算符，再本身++ --

var num1 = 10;

var num2 = 10;

var num3 = ++num1 + num2; //21

var num4 = num1 + num2; //21

var num1 = 10;

var num2 = 10;

var num3 = num1++ + num2; //20

var num4 = num1 + num2; //21

布尔操作符

逻辑非（！）：首先会将它的参数转换为一个布尔值，然后再对其求反。

!object = false

!”xxx” = false

!123456 = false

!0 = true

!false = true

!”” = true

!NaN = true

逻辑与（&&）：真真才能是真，其他都是假

逻辑或（||）：假假才能是假，其他都是真

相等、不相等：在比较之前先转换操作数

var res = (“123” == 123); //true

全等、不全等：在比较之前不转换操作数

var res = (“123” === 123); //false

for语句：一种前测试循环语句，但它具有在执行循环之前初始化变量和定义循环后要执行的代码的能力。

for(initialization;expression;post-loop-expression) {

statement

}

initialization：计数器初始化，这个变量为全局变量

expression：循环条件检测

post-loop-expression：计数器变量的更新

等价于下面while循环：

initialization; //1、初始化，只在循环开始之前执行一次

while(expression) { //2、每次执行循环之前会执行并判断表达式的结果来决定是否执行循环体

statement; //3、条件为真则执行循环体内的语句

post-loop-expression; //4、最后执行计数器变量更新语句

} //5、跳转到2继续执行

函数：

return语句也可以不带有任何返回值，在这种情况下，函数在停止执行后将返回undefined

函数中的参数：函数不介意传递进来多少个参数，也不在乎传进来的参数是什么数据类型，因为JS中的参数在内部是用一个数组来表示的，函数接收到的始终都是这个数组，而不关心数组中包含哪些参数，有或者没有参数都无所谓。arguments对象来访问这个参数数组，从而获取传递给函数的每一个参数。arguments对象只是与数组类似，可以使用方括号语法访问它的每一个元素arguments[0], arguments[1]……..，使用length属性来确定传递进来多少个参数。

arguments有一个属性：callee，它是一个指针，指向拥有这个arguments对象的函数

传递参数：JS中所有的函数的参数都是按值传递的

执行环境：是JS中最为重要的一个概念。执行环境定义了变量或函数有权访问的其他数据，决定了它们各自的行为，每个执行环境都有一个与之关联的变量对象，环境中定义的所有变更和函数都保存在这个对象中。

全局执行环境是最外围的一个执行环境，每个函数都有自己的执行环境，当执行流进入一个函数时，函数的环境就会被推入一个环境栈中，函数执行之后，栈将其环境弹出，把控制权返回给之前的执行环境。

当代码在一个环境中执行时，会创建变量对象的一个作用域链，作用域链的用途是保证对执行环境有权访问的所有变量和函数的有序访问，作用域链的前端始终都是当前执行的代码所在环境的变量对象，如果这个环境是函数，则将其活动对象作为变量对象，活动对象在最开始时只包含一个变量，即arguments对象，作用域链中的下一个变量对象来自包含（外部）环境，而再下一个变量对象则来自下一个包含环境，这样一直延续到全局执行环境。

window

var color1 = ‘red’;

function changecolor(){

color1, changecolor()

var color2 = ‘blue’;

color2, swapcolor()

function swapcolor(){

var color3 = color2;

color3

color2 = color1;

color1 = color3;

}

swapcolor();

}

changecolor();

向上搜索作用域链，以查询变量和函数，在搜索过程中，如果存在一个局部变量的定义，则搜索会自动停止，不再进入另一个变量对象，换名话说，如果局部环境中存在着同名标识符，就不会使用位于父环境中的标识符。

垃圾收集：JS具有自动垃圾收集机制，执行环境会负责管理代码执行过程中使用的内存。垃圾收集器会按照固定的时间间隔周期性的找出那些不再继续使用的变更，然后释放其占用的内存。

实例化对象

new Object() new Function()

实例化的过程就是为对象分配内存空间的过程，此时，对象才成为引用类型的实例

实例化了就有自己内存空间，没有实例化就和引用的对象内在空间地址相同，指向同一空间

类 对象 实例

类：所有的人

对象：抽象的结构，人大致的外形，会说会听会跑会跳，有男人、女人

实例：具体化的结构。陈飞彪是一个实实在在存在的人（实例都是对象，对象不全是实例）

function Fn() {

this.hello = function () {

console.log('实例方法');

}

}

Fn.hello = function() {

console.log('静态方法');

}

Fn.prototype.hello = function () {

console.log('实例共享方法(对象方法)');

}

RegExp类型：支持正则表达式

var reg = /pattern/flags;

pattern：任何简单或复杂的正则表达式，可以包含字符类、限定符、分组、向前查找、反向引用

flags：g：全局模式，即模式将被应用于所有字符串，而非发现第一个匹配项时立即停止

i：不区分大小写

m：多行模式，即在到达一行文本末尾时还会继续查找下一行中是否存在与模式匹配的项

var reg = /at/g; //匹配字符串中所有“at”的实例

var reg2 = /[bc]at/i; //匹配第一个“bat”或“cat”，不区分不小定

var reg3 = /.at/gi; //匹配所有以“at”结尾的3个字符的组合，不区分在小写

Function类型

每个函数都是Function类型的实例，由于函数是对象，因此函数名实际上也是一个指向函数对象的指针，不会与某个函数绑定

访问函数的指针：fun

调用执行函数：fun()

由于函数名仅仅是指向函数的指针，因此函数名与包含对象指针的其他变量没有什么不同，换句话说，一个函数可能会有多个不同的名字

function sum(num1,num2){

return num1 + num2;

}

console.log(sum(1,2));

var count = sum;

console.log(count(2,2));

sum = null;

console.log(count(2,2));

函数声明、函数表达式区别：在解析器执行加载的代码时的不同，先在执行任何代码前读取函数声明并赋值，而函数表达式必须等到解析器执行到它所在的代码行时，才会被解释执行。

作为值的函数，函数名本身是变更，所以函数也可以作为值来使用，不公可以像传递参数一样把一个函数传递给另一个函数，而且可以将一个函数作为另一个函数的结果返回

function callfun(func,arg){

return func(arg);

}

function add(num){

return num + 10;

}

var res = callfun(add,10);

console.log(res);

function say(name){

return “hello, ” + name;

}

console.log(callfun(say,”pshcen”));

函数内部属性：callee

this；由于在调用函数之前this的值并不确定，因此this可能会在代码执行过程中引用不同的对象，

每个函数都包含两个非继承而来的方法：call() apply()

这两个方法都是在特定的作用域中调用函数，实际上等于设置函数体内this对象的值

apply()方法接收两个对数，一个是在其中运行函数的作用域，另一个是参数数组（可以是Array的实例，也可以是arguments对象）

function sum(num1,num2){

return num1 + num2;

}

function callsum1(num1,num2){

return sum.apply(this,arguments);

}

function callsum2(num1,num2){

return sum.apply(this,[num1,num2]);

}

function callsum2(num1,num2){

return sum.call(this,num1,num2);

}

console.log(callsum1(1,2));

console.log(callsum2(1,2));

这两个方法真正强大的地方是能够扩充函数赖以运行的作用域。

color = “red”;

var obj = {color:”blue”};

function sayColor(){

console.log(this.color);

}

sayColor(); //red

sayColor.call(this); //red

sayColor.call(obj); //blue

obj.callcolor = sayColor; //obj.color = sayColor;重写了color属性

obj.callcolor(); //blue

bind()：创建一个函数的实例，其this值会绑定到传给bind()函数的值

color = “red”;

var obj = {color:”blue”};

function saycolor(){

console.log(this.color);

}

var callcolor = saycolor.bind(obj); //创建了callclor()函数，this绑定到obj

callcolor();