**[细说 JavaScript 七种数据类型](https://www.cnblogs.com/onepixel/p/5140944.html)**

在 JavaScript 规范中，共定义了七种数据类型，分为 “基本类型” 和 “引用类型” 两大类，如下所示：

**基本类型：String、Number、Boolean、Symbol、Undefined、Null**

**引用类型：Object**

下面将详细介绍这七种数据类型的一些特性。

**1、String 类型**

String 类型用于表示由零或多个 16 位 Unicode 字符组成的字符序列，即字符串。

**1.1 存储结构**

由于计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。在计算机中，1个字节（byte）由 8个比特（bit）组成，所以 1 个字节能表示的最大整数就是 255，如果想表示更大整数，就必须用更多的字节，比如 2 个字节可以表示的最大整数为 65535 。最早，只有 127 个字符被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为 ASCII编码，比如大写字母 A 的编码是 65，小写字母 z 的编码是122。

但如果要表示中文字符，显然一个字节是不够的，至少需要两个字节。所以，中国制定了 GB2312 编码，用来表示中文字符。基于同样的原因，各个国家都制定了自己的编码规则 。这样就会出现一个问题，即在多语言混合的文本中，不同的编码会出现冲突，导致乱码出现 。

为了解决这个问题，Unicode 编码应运而生，它把所有的语言都统一到一套编码中，采用 2 个字节表示一个字符，即最多可以表示 65535 个字符，这样基本上可以覆盖世界上常用的文字，如果要表示更多的文字，也可以采用 4 个字节进行编码，这是一种通用的编码规范 。

因此，JavaScript 中的字符也采用 Unicode 来编码 ，也就是说，JavaScript 中的英文字符和中文字符都会占用 2 个字节的空间大小 ，这种多字节字符，通常被称为宽字符。

**1.2 基本包装类型**

在 JavaScript 中，字符串是基本数据类型，本身不存任何操作方法 。为了方便的对字符串进行操作，ECMAScript 提供了一个基本包装类型：String 对象 。它是一种特殊的引用类型，JS引擎每当读取一个字符串的时候，就会在内部创建一个对应的 String 对象，该对象提供了很多操作字符的方法，这就是为什么能对字符串调用方法的原因。

|  |  |
| --- | --- |
| 1  2 | var name ='JavaScript';  var value = name.substr(2,1); |

当第二行代码访问变量 str 时，访问过程处于一种读取模式，也就是要从内存中读取这个字符串的值。而在读取模式中访问字符串时，引擎内部会自动完成下列处理：

* 创建 String 类型的一个实例
* 在实例上调用指定的方法
* 销毁这个实例

用伪代码形象的模拟以上三个步骤：

|  |  |
| --- | --- |
| 1  2  3 | var obj =new String('JavaScript');  var value = obj.substr(2,1);  name =null; |

可以看出，基本包装类型是一种特殊的引用类型。它和普通引用类型有一个很重要的区别，就是对象的生存期不同 。使用 new 操作符创建的引用类型的实例，在执行流离开当前作用域之前都一直保存在内存中。而自动创建的基本包装类型的对象，则只存在于一行代码的执行瞬间，然后立即被销毁。在 JavaScript 中，类似的基本包装类型还有 Number、Boolean 对象 。

**1.3 常用操作方法**

作为字符串的基本包装类型，String 对象提供了以下几类方法，用以操作字符串：

* 字符操作：charAt，charCodeAt，fromCharCode
* 字符串提取：substr，substring ，slice
* 位置索引：indexOf ，lastIndexOf
* 大小写转换：toLowerCase，toUpperCase
* 模式匹配：match，search，replace，split
* 其他操作：concat，trim，localeCompare

charCodeAt 的作用是获取字符的 Unicode 编码，俗称 “Unicode 码点”。fromCharCode 是 String 对象上的静态方法，作用是根据 Unicode 编码返回对应的字符。

|  |  |
| --- | --- |
| 1  2  3  4  5 | var a ='a';  // 获取Unicode编码  var code = a.charCodeAt(0);// 97  // 根据Unicode编码获取字符  String.fromCharCode(code); // a |

通过 charCodeAt 获取字符的 Unicode 编码，然后再把这个编码转化成二进制，就可以得到该字符的二进制表示。

|  |  |
| --- | --- |
| 1  2  3 | var a ='a';  var code = a.charCodeAt(0);// 97  code.toString(2);// 1100001 |

对于字符串的提取操作，有三个相类似的方法，分别如下：

|  |  |
| --- | --- |
| 1  2  3 | substr(start [, length])  substring(start [, end])  slice(start [, end]) |

从定义上看，substring 和 slice 是同类的，参数都是字符串的某个 start 位置到某个 end 位置（但 end 位置的字符不包括在结果中）；而 substr 则是字符串的某个 start 位置起，数 length 个长度的字符才结束。二者的共性是：从 start 开始，如果没有第 2 个参数，都是直到字符串末尾。

substring 和 slice 的区别则是：slice 可以接受 “负数”，表示从字符串尾部开始计数； 而 substring 则把负数或其它无效的数当作 0。

|  |  |
| --- | --- |
| 1  2 | 'hello world!'.slice(-6, -1)// 'world'  'hello world!'.substring("abc", 5)// 'hello' |

substr 的 start 也可接受负数，也表示从字符串尾部计数，这点和 slice 相同；但 substr 的 length 则不能小于 1，否则返回空字符串。

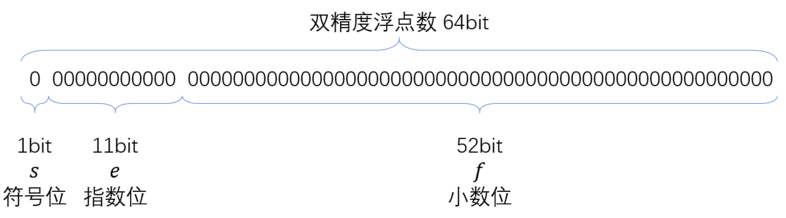
|  |  |
| --- | --- |
| 1  2 | 'hello world!'.substr(-6, 5)// 'world'  'hello world!'.substr(0, -1)// '' |

**2、Number 类型**

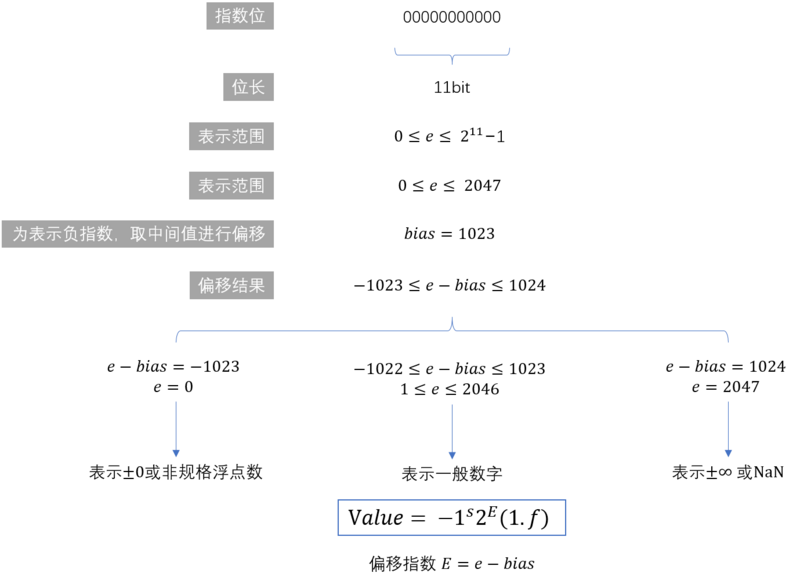
JavaScript 中的数字类型只有 Number 一种，Number 类型采用 IEEE754 标准中的 “双精度浮点数” 来表示一个数字，不区分整数和浮点数 。

**2.1 存储结构**

在 IEEE754 中，双精度浮点数采用 64 位存储，即 8 个字节表示一个浮点数 。其存储结构如下图所示：



指数位可以通过下面的方法转换为使用的指数值：



**2.2 数值范围**

从存储结构中可以看出， 指数部分的长度是11个二进制，即指数部分能表示的最大值是 2047（211-1），取中间值进行偏移，用来表示负指数，也就是说指数的范围是 [-1023,1024] 。因此，这种存储结构能够表示的数值范围为 21024到 2-1023，超出这个范围的数无法表示 。21024 和 2-1023转换为科学计数法如下所示：

1.7976931348623157 × 10308

5 × 10-324

因此，JavaScript 中能表示的最大值是 1.7976931348623157 × 10308，最小值为 5 × 10-324 。

这两个边界值可以分别通过访问 Number 对象的 MAX\_VALUE 属性和 MIN\_VALUE 属性来获取：

|  |  |
| --- | --- |
| 1  2 | Number.MAX\_VALUE;// 1.7976931348623157e+308  Number.MIN\_VALUE;// 5e-324 |

如果数字超过最大值或最小值，JavaScript 将返回一个不正确的值，这称为 “正向溢出(overflow)” 或 “负向溢出(underflow)” 。

|  |  |
| --- | --- |
| 1  2  3  4  5 | Number.MAX\_VALUE+1 == Number.MAX\_VALUE;//true  Number.MAX\_VALUE+1e292;//Infinity  Number.MIN\_VALUE + 1;//1  Number.MIN\_VALUE - 3e-324;//0  Number.MIN\_VALUE - 2e-324;//5e-324 |

**2.3 数值精度**

在 64 位的二进制中，符号位决定了一个数的正负，指数部分决定了数值的大小，小数部分决定了数值的精度。

IEEE754 规定，有效数字第一位默认总是1 。因此，在表示精度的位数前面，还存在一个 “隐藏位” ，固定为 1 ，但它不保存在 64 位浮点数之中。也就是说，有效数字总是 1.xx...xx 的形式，其中 xx..xx 的部分保存在 64 位浮点数之中，最长为52位 。所以，JavaScript 提供的有效数字最长为 53 个二进制位，其内部实际的表现形式为：

(-1)^符号位 \* 1.xx...xx \* 2^指数位

这意味着，JavaScript 能表示并进行精确算术运算的整数范围为：[-253-1，253-1]，即从最小值 -9007199254740991 到最大值 9007199254740991 之间的范围 。

|  |  |
| --- | --- |
| 1  2 | Math.pow(2, 53)-1 ;// 9007199254740991  -Math.pow(2, 53)-1 ;// -9007199254740991 |

可以通过 Number.MAX\_SAFE\_INTEGER 和  Number.MIN\_SAFE\_INTEGER 来分别获取这个最大值和最小值。

|  |  |
| --- | --- |
| 1  2 | console.log(Number.MAX\_SAFE\_INTEGER) ;// 9007199254740991  console.log(Number.MIN\_SAFE\_INTEGER) ;// -9007199254740991 |

对于超过这个范围的整数，JavaScript 依旧可以进行运算，但却不保证运算结果的精度。

|  |  |
| --- | --- |
| 1  2  3  4  5 | Math.pow(2, 53) ;// 9007199254740992  Math.pow(2, 53) + 1;// 9007199254740992  9007199254740993;//9007199254740992  90071992547409921;//90071992547409920  0.923456789012345678;//0.9234567890123456 |

**2.4 精度丢失**

计算机中的数字都是以二进制存储的，如果要计算 0.1 + 0.2 的结果，计算机会先把 0.1 和 0.2 分别转化成二进制，然后相加，最后再把相加得到的结果转为十进制 。

但有一些浮点数在转化为二进制时，会出现无限循环 。比如， 十进制的 0.1 转化为二进制，会得到如下结果：

0.0001 1001 1001 1001 1001 1001 1001 1001 …（1001无限循环）

而存储结构中的尾数部分最多只能表示 53 位。为了能表示 0.1，只能模仿十进制进行四舍五入了，但二进制只有 0 和 1 ， 于是变为 0 舍 1 入 。 因此，0.1 在计算机里的二进制表示形式如下：

0.0001100110011001100110011001100110011001100110011001101

用标准计数法表示如下：

(−1)0× 2−4× (1.1001100110011001100110011001100110011001100110011010)2

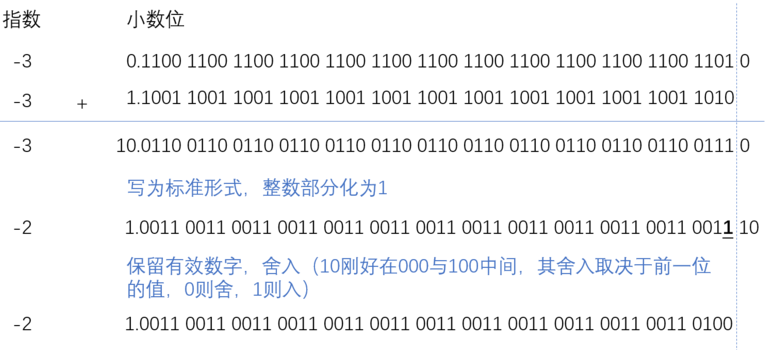
同样，0.2 的二进制也可以表示为：

(−1)0× 2−3× (1.1001100110011001100110011001100110011001100110011010)2

在计算浮点数相加时，需要先进行 “对位”，将较小的指数化为较大的指数，并将小数部分相应右移：

0.1→ (−1)0× 2−3× (0.11001100110011001100110011001100110011001100110011010)2  
0.2→ (−1)0× 2−3× (1.1001100110011001100110011001100110011001100110011010)2

最终，“0.1 + 0.2” 在计算机里的计算过程如下：



经过上面的计算过程，0.1 + 0.2 得到的结果也可以表示为：

(−1)0× 2−2× (1.0011001100110011001100110011001100110011001100110100)2

然后，通过 JS 将这个二进制结果转化为十进制表示：

|  |  |
| --- | --- |
| 1  2 | (-1)\*\*0 \* 2\*\*-2 \* (0b10011001100110011001100110011001100110011001100110100 \* 2\*\*-52);//0.30000000000000004  console.log(0.1 + 0.2) ;// 0.30000000000000004 |

这是一个典型的精度丢失案例，从上面的计算过程可以看出，0.1 和 0.2 在转换为二进制时就发生了一次精度丢失，而对于计算后的二进制又有一次精度丢失 。因此，得到的结果是不准确的。

**2.5 特殊数值**

JavaScript 提供了几个特殊数值，用于判断数字的边界和其他特性 。如下所示：

* Number.MAX\_VALUE：JavaScript 中的最大值
* Number.MIN\_VALUE：JavaScript 中的最小值
* Number.MAX\_SAFE\_INTEGER：最大安全整数，为 253-1
* Number.MIN\_SAFE\_INTEGER：最小安全整数，为 -(253-1)
* Number.POSITIVE\_INFINITY：对应 Infinity，代表正无穷
* Number.NEGATIVE\_INFINITY：对应 -Infinity，代表负无穷
* Number.EPSILON：是一个极小的值，用于检测计算结果是否在误差范围内
* Number.NaN：表示非数字，NaN与任何值都不相等，包括NaN本身
* Infinity：表示无穷大，分 正无穷 Infinity 和 负无穷 -Infinity

**2.6 数值转换**

有 3 个函数可以把非数值转换为数值，分别如下：

|  |  |
| --- | --- |
| 1  2  3 | Number(value)  parseInt(string [, radix])  parseFloat(string) |

Number() 可以用于任何数据类型，而另两个函数则专门用于把字符串转换成数值。

对于字符串而言，Number() 只能对字符串进行整体转换，而 parseInt() 和 parseFloat() 可以对字符串进行部分转换，即只转换第一个无效字符之前的字符。

对于不同数据类型的转换，Number() 的处理也不尽相同，其转换规则如下:

【1】如果是 Boolean 值，true 和 false 将分别被转换为 1 和 0。

【2】如果是数字值，只是简单的传入和返回。

【3】如果是 null 值，返回 0。

【4】如果是 undefined，返回 NaN。

【5】如果是字符串，遵循下列规则：

* 如果字符串中只包含数字(包括前面带正号或负号的情况)，则将其转换为十进制数值;
* 如果字符串中包含有效的浮点格式，则将其转换为对应的浮点数值;
* 如果字符串中包含有效的十六进制格式，则将其转换为相同大小的十进制整数值;
* 如果字符串是空的(不包含任何字符)，则将其转换为 0;
* 如果字符串中包含除上述格式之外的字符，则将其转换为 NaN。

【6】如果是对象，则调用对象的 valueOf() 方法，然后依照前面的规则转换返回的值。如果转换的结果是 NaN，则调用对象的 toString() 方法，然后再次依照前面的规则转换返回的字符串值。

需要注意的是：

一元加操作符加号 “+” 和 Number() 具有同样的作用。

在 ECMAScript 2015 规范中，为了实现全局模块化，Number 对象重写了 parseInt 和 parseFloat 方法，但和对应的全局方法并无区别。

|  |  |
| --- | --- |
| 1  2 | Number.parseInt === parseInt;// true  Number.parseFloat === parseFloat;// true |

**2.7 位运算**

位运算作用于最基本的层次上，即按内存中表示数值的位来操作数值。ECMAScript 中的数值以64位双精度浮点数存储，但位运算只能作用于整数，因此要先将 64 位的浮点数转换成 32 位的整数，然后再进行位运算，最后再将计算结果转换成64位浮点数存储。常见的位运算有以下几种：

* 按位非（NOT）：~
* 按位与（AND）：&
* 按位或（OR）： |
* 按位异或（XOR）：^
* 左移：<<
* 有符号右移：>>
* 无符号右移：>>>

需要注意的是：

“有符号右移” 和 “无符号右移” 只在计算负数的情况下存在差异，>> 在符号位的右侧补0，不移动符号位；而 >>> 是在符号位的左侧补0，符号位发生移动和改变。

**2.8 四舍五入**

JavaScript 对数字进行四舍五入操作的 API 有 ceil，floor，round，toFixed，toPrecision 等，详细介绍请参考：[JavaScript 中的四舍五入](http://www.cnblogs.com/onepixel/p/5141566.html)

**3、Boolean 类型**

Boolean 类型只有两个字面值：true 和 false 。 在 JavaScript 中，所有类型的值都可以转化为与 Boolean 等价的值 。转化规则如下：

* 所有对象都被当作 true
* 空字符串被当作 false
* null 和 undefined 被当作 false
* 数字 0 和 NaN 被当作 false

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Boolean([]);//true  Boolean({});//true  Boolean(undefined);//false  Boolean(null);//false  Boolean(''); //false  Boolean(0);  //false  Boolean(NaN);//false |

除 Boolean() 方法可以返回布尔值外，以下 4 种类型的操作，也会返回布尔值。

**【1】关系操作符：>，>=，<，<=**

当关系操作符的操作数使用了非数值时，要进行数据转换或完成某些奇怪的操作。

* 如果两个操作数都是数值，则执行数值比较。
* 如果两个操作数都是字符串，则逐个比较两者对应的字符编码(charCode)，直到分出大小为止 。
* 如果操作数是其他基本类型，则调用Number() 将其转化为数值，然后进行比较。
* NaN 与任何值比较，均返回 false 。
* 如果操作数是对象，则调用对象的 valueOf 方法（如果没有 valueOf ，就调用 toString 方法），最后用得到的结果，根据前面的规则执行比较。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 'a' >'b';// false, 即 'a'.charCodeAt(0) > 'b'.charCodeAt(0)  2 >'1'; // true, 即 Number('1') = 1  true > 0;//true, 即 Number(true) = 1  undefined > 0;//false, Number(undefined) = NaN  null < 0;//false, Number(null) = NaN  new Date > 100;//true , 即 new Date().valueOf() |

**【2】相等操作符： ==，!=，===，!==**

== 和 != 操作符都会先转换操作数，然后再比较它们的相等性。在转换不同的数据类型时，需要遵循下列基本规则：

* 如果有一个操作数是布尔值，则在比较相等性之前，先调用 Number() 将其转换为数值；
* 如果一个操作数是字符串，另一个操作数是数值，在比较相等性之前，先调用 Number() 将字符串转换为数值；
* 如果一个操作数是对象，另一个操作数不是，则调用对象的 valueOf() 方法，用得到的基本类型值按照前面的规则进行比较；
* null 和 undefined 是相等的。在比较相等性之前，不能将 null 和 undefined 转换成其他任何值。
* 如果有一个操作数是 NaN，则相等操作符返回 false，而不相等操作符返回 true；
* 如果两个操作数都是对象，则比较它们的指针地址。如果都指向同一个对象，则相等操作符返回 true；否则，返回 false。

=== 和 !== 操作符最大的特点是，在比较之前不转换操作数 。它们的操作规则如下：

* ===： 类型相同，并且值相等，才返回 true ，否则返回 false 。
* !== ： 类型不同，或者值不相等，就返回 true，否则返回 false 。

|  |  |
| --- | --- |
| 1  2  3  4  5 | null === undefined;//false, 类型不同，直接返回 false  [] === [];//false ,类型相同，值不相同，指针地址不同  a=[],b=a,a===b;//true, 类型相同，值也相等  1 !=='1' ;// true , 值相等，但类型不同  [] !== [] ; // true, 类型相同，但值不相等 |

**【3】布尔操作符：!**

布尔操作符属于一元操作符，即只有一个分项。其求值过程如下：

* 对分项求值，得到一个任意类型值；
* 使用 Boolean() 把该值转换为布尔值 true 或 false；
* 对布尔值取反，即 true 变 false，false 变 true

|  |  |
| --- | --- |
| 1  2  3 | !(2+3) ;// false  !(function(){});//false  !([] &&null &&'');//true |

利用 ! 的取反的特点，使用 !! 可以很方便的将一个任意类型值转换为布尔值：

|  |  |
| --- | --- |
| 1  2  3  4 | console.log(!!0);//false  console.log(!!'');//false  console.log(!!(2+3));//true  console.log(!!([] &&null &&''));//false |

需要注意的是：

逻辑与 “&&” 和 逻辑或 “||” 返回的不一定是布尔值，而是包含布尔值在内的任意类型值。

如下所示：

|  |  |
| --- | --- |
| 1  2  3  4 | [] && 1;//1  null && undefined;//null  [] || 1;//[]  null || 1;//1 |

逻辑操作符属于短路操作符 。在进行计算之前，会先通过 Boolean() 方法将两边的分项转换为布尔值，然后分别遵循下列规则进行计算：

* 逻辑与：从左到右检测每一个分项，返回第一个布尔值为 false 的分项，并停止检测 。如果没有检测到 false 项，则返回最后一个分项 。
* 逻辑或：从左到右检测每一个分项，返回第一个布尔值为 true 的分项，并停止检测 。如果没有检测到 true 项，则返回最后一个分项 。

|  |  |
| --- | --- |
| 1  2  3  4 | [] && {} && null && 1;//null  [] && {} &&  !null  &&  1 ;//1  null || undefined || 1 || 0;//1  undefined || 0 || function(){};//function(){} |

**【4】条件语句：if，while，?**

条件语句通过计算表达式返回一个布尔值，然后再根据布尔值的真假，来执行对应的代码。其计算过程如下：

* 对表达式求值，得到一个任意类型值
* 使用 Boolean() 将得到的值转换为布尔值 true 或 false

|  |  |
| --- | --- |
| 1  2  3 | if(arr.length) { }  obj && obj.name ?'obj.name' :''  while(arr.length){ } |

**4、Symbol 类型**

Symbol 是 ES6 新增的一种原始数据类型，它的字面意思是：符号、标记。代表独一无二的值 。

在 ES6 之前，对象的属性名只能是字符串，这样会导致一个问题，当通过 mixin 模式为对象注入新属性的时候，就可能会和原来的属性名产生冲突 。而在 ES6 中，Symbol 类型也可以作为对象属性名，凡是属性名是 Symbol 类型的，就都是独一无二的，可以保证不会与其他属性名产生冲突。

Symbol 值通过函数生成，如下所示：

|  |  |
| --- | --- |
| 1  2 | let s = Symbol();//s是独一无二的值  typeof s ;// symbol |

和其他基本类型不同的是，Symbol 作为基本类型，没有对应的包装类型，也就是说 Symbol 本身不是对象，而是一个函数。因此，在生成 Symbol 类型值的时候，不能使用 new 操作符 。

Symbol 函数可以接受一个字符串作为参数，表示对 Symbol 值的描述，当有多个 Symbol 值时，比较容易区分。

|  |  |
| --- | --- |
| 1  2  3 | var s1 = Symbol('s1');  var s2 = Symbol('s2');  console.log(s1,s2);// Symbol(s1) Symbol(s2) |

注意，Symbol 函数的参数只是表示对当前 Symbol 值的描述，因此，相同参数的 Symbol 函数的返回值也是不相等的。

用 Symbol 作为对象的属性名时，不能直接通过**点**的方式访问属性和设置属性值。因为正常情况下，引擎会把点后面的属性名解析成字符串。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var s = Symbol();  var obj = {};  obj.s ='Jack';  console.log(obj);// {s: "Jack"}  obj[s] ='Jack';  console.log(obj) ;//{Symbol(): "Jack"} |

Symbol 作为属性名，该属性不会出现在 for...in、for...of 循环中，也不会被 Object.keys()、Object.getOwnPropertyNames()、JSON.stringify() 返回。但是，它也不是私有属性，有一个 Object.getOwnPropertySymbols() 方法，专门获取指定对象的所有 Symbol 属性名。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var obj = {};  var s1 = Symbol('s1');  var s2 = Symbol('s2')；  obj[s1] ='Jack';  obj[s2] ='Tom';  Object.keys(obj);//[]  for(var iin obj){      console.log(i);//无输出  }  Object.getOwnPropertySymbols(obj);//[Symbol(s1), Symbol(s2)] |

另一个新的API，Reflect.ownKeys 方法可以返回所有类型的键名，包括常规键名和 Symbol 键名。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var obj = {};  var s1 = Symbol('s1');  var s2 = Symbol('s2')；  obj[s1] ='Jack';  obj[s2] ='Tom';  obj.name ='Nick';  Reflect.ownKeys(obj);//[Symbol(s1), Symbol(s2),"name"] |

有时，我们希望重新使用同一个 Symbol 值，Symbol.for 方法可以做到这一点。它接受一个字符串作为参数，然后全局搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则就新建并返回一个以该字符串为名称的 Symbol 值。

|  |  |
| --- | --- |
| 1  2  3 | var s1 = Symbol.for('foo');  var s2 = Symbol.for('foo');  s1 === s2//true |

Symbol.for() 也可以生成 Symbol 值，它 和 Symbol() 的区别是：

* Symbol.for() 首先会在全局环境中查找给定的 key 是否存在，如果存在就返回，否则就创建一个以 key 为标识的 Symbol 值
* Symbol.for() 生成的 Symbol 会登记在全局环境中供搜索，而 Symbol() 不会。

因此，Symbol.for()  永远搜索不到 用 Symbol() 产生的值。

|  |  |
| --- | --- |
| 1  2  3 | var s = Symbol('foo');  var s1 = Symbol.for('foo');  s === s1;// false |

Symbol.keyFor() 方法返回一个已在全局环境中登记的 Symbol 类型值的 key 。

|  |  |
| --- | --- |
| 1  2  3  4 | var s1 = Symbol.for('s1');  Symbol.keyFor(s1);//foo  var s2 = Symbol('s2');//未登记的 Symbol 值  Symbol.keyFor(s2);//undefined |

**5、Undefined 类型**

Undefined 是 Javascript 中特殊的原始数据类型，它只有一个值，即 undefined，字面意思是：未定义的值 。它的语义是，希望**表示一个变量最原始的状态，而非人为操作的结果 。** 这种原始状态会在以下 4 种场景中出现：

**【1】声明了一个变量，但没有赋值**

|  |  |
| --- | --- |
| 1  2 | var foo;  console.log(foo);//undefined |

访问 foo，返回了 undefined，表示这个变量自从声明了以后，就从来没有使用过，也没有定义过任何有效的值，即处于一种原始而不可用的状态。

**【2】访问对象上不存在的属性**

|  |  |
| --- | --- |
| 1  2  3 | console.log(Object.foo);// undefined  var arr = [];  console.log(arr[0]);// undefined |

访问 Object 对象上的 foo 属性，返回 undefined ， 表示Object 上不存在或者没有定义名为 foo 的属性。数组中的元素在内部也属于对象属性，访问下标就等于访问这个属性，返回 undefined ，就表示数组中不存在这个元素。

**【3】函数定义了形参，但没有传递实参**

|  |  |
| --- | --- |
| 1  2  3  4  5 | // 函数定义了形参 a  function fn(a) {      console.log(a);//undefined  }  fn();// 未传递实参 |

函数 fn 定义了形参 a， 但 fn 被调用时没有传递参数，因此，fn 运行时的参数 a 就是一个原始的、未被赋值的变量。

**【4】使用 void 对表达式求值**

|  |  |
| --- | --- |
| 1  2  3  4  5 | void 0 ;// undefined  void false;// undefined  void [];// undefined  void null;// undefined  void function fn(){} ;// undefined |

ECMAScript 明确规定 void 操作符 对任何表达式求值都返回 undefined ，这和函数执行操作后没有返回值的作用是一样的，JavaScript 中的函数都有返回值，当没有 return 操作时，就默认返回一个原始的状态值，这个值就是 undefined，表明函数的返回值未被定义。

因此，undefined 一般都来自于某个表达式最原始的状态值，不是人为操作的结果。当然，你也可以手动给一个变量赋值 undefined，但这样做没有意义，因为一个变量不赋值就是 undefined 。

**6、Null 类型**

Null 是 Javascript 中特殊的原始数据类型，它只有一个值，即 null，字面意思是：“空值”  。它的语义是，希望**表示一个对象被人为的重置为空对象，而非一个变量最原始的状态 。** 在内存里的表示就是，栈中的变量没有指向堆中的内存对象。当一个对象被赋值了 null 以后，原来的对象在内存中就处于游离状态，GC 会择机回收该对象并释放内存。因此，如果需要释放某个对象，就将变量设置为 null，即表示该对象已经被清空，目前无效状态。

null 是原始数据类型 Null 中的唯一一个值，但 typeof 会将 null 误判为 Object 类型 。

|  |  |
| --- | --- |
| 1 | typeof null =='object' |

在 JavaScript 中，数据类型在底层都是以二进制形式表示的，**二进制的前三位为 0 会被 typeof 判定为对象类型**，如下所示：

* 000 - 对象，数据是对象的应用
* 1 - 整型，数据是31位带符号整数
* 010 - 双精度类型，数据是双精度数字
* 100 - 字符串，数据是字符串
* 110 - 布尔类型，数据是布尔值

而 null 值的二进制表示全是 0 ，自然前三位当然也是 000，因此，typeof 会误以为是对象类型。如果想要知道 null 的真实数据类型，可以通过下面的方式来获取。

|  |  |
| --- | --- |
| 1 | Object.prototype.toString.call(null) ;// [object Null] |

**7、Object 类型**

在 ECMAScript 规范中，引用类型除 Object 本身外，Date、Array、RegExp 也属于引用类型 。

引用类型也即对象类型，ECMA262 把对象定义为：**无序属性的集合，其属性可以包含基本值、对象或者函数**。 也就是说，对象是一组没有特定顺序的值 。由于其值的大小会改变，所以不能将其存放在栈中，否则会降低变量查询速度。因此，对象的值存储在堆(heap)中，而存储在变量处的值，是一个指针，指向存储对象的内存处，即按址访问。具备这种存储结构的，都可以称之为引用类型 。

**7.1 对象拷贝**

由于引用类型的变量只存指针，而对象本身存储在堆中 。因此，当把一个对象赋值给多个变量时，就相当于把同一个对象地址赋值给了每个变量指针 。这样，每个变量都指向了同一个对象，当通过一个变量修改对象，其他变量也会同步更新。

|  |  |
| --- | --- |
| 1  2  3  4 | var obj = {name:'Jack'};  var obj2 = obj;  obj2.name ='Tom'  console.log(obj.name,obj2.name);//Tom,Tom |

ES6 提供了一个原生方法用于对象的拷贝，即 Object.assign() 。

|  |  |
| --- | --- |
| 1  2  3  4 | var obj = {name:'Jack'};  var obj2 = Object.assign({},obj);  obj2.name ='Tom'  console.log(obj.name,obj2.name);//Jack Tom |

需要注意的是，Object.assign() 拷贝的是属性值。当属性值是基本类型时，没有什么问题 ，但如果该属性值是一个指向对象的引用，它也只能拷贝那个引用值，而不会拷贝被引用的那个对象。

|  |  |
| --- | --- |
| 1  2  3  4 | var obj = {base:{name:'Jack'}};  var obj2 = Object.assign({},obj);  obj2.base.name ='Tom'  console.log(obj.base.name,obj2.base.name);//Tom Tom |

从结果可以看出，obj 和 obj2 的属性 base 指向了同一个对象的引用。因此，Object.assign 仅仅是拷贝了一份对象指针作为副本 。这种拷贝被称为 “一级拷贝” 或 “浅拷贝”**。**

如果要彻底的拷贝一个对象作为副本，两者之间的操作相互不受影响，则可以通过 JSON 的序列化和反序列化方法来实现 。

|  |  |
| --- | --- |
| 1  2  3  4 | var obj = {base:{name:'Jack'}};  var obj2 = JSON.parse(JSON.stringify(obj))  obj2.base.name ='Tom'  console.log(obj.base.name,obj2.base.name);//Jack Tom |

这种拷贝被称为 “多级拷贝” 或 “深拷贝” 。

**7.2 属性类型**

ECMA-262 第 5 版定义了一些内部特性（attribute），用以描述对象属性（property）的各种特征。ECMA-262 定义这些特性是为了实现 JavaScript 引擎用的，因此在 JavaScript 中不能直接访问它们。为了表示特性是内部值，该规范把它们放在了两对儿方括号中，例如[[Enumerable]]。 这些内部特性可以分为两种：数据属性 和 访问器属性 。

**【1】数据属性**

数据属性包含一个数据值的位置，在这个位置可以读取和写入值 。数据属性有4个描述其行为的内部特性：

* [[Configurable]]：能否通过 delete 删除属性从而重新定义属性，或者能否把属性修改为访问器属性。该默认值为 true。
* [[Enumerable]]：表示能否通过 for-in 循环返回属性。默认值为 true。
* [[Writable]]：能否修改属性的值。默认值为 true。
* [[Value]]：包含这个属性的数据值。读取属性值的时候，从这个位置读；写入属性值的时候，把新值保存在这个位置。默认值为 undefined 。

要修改属性默认的特性，必须使用 ECMAScript 5 的 Object.defineProperty() 方法。这个方法接收三个参数：属性所在的对象、属性的名字和一个描述符对象。其中，描述符（descriptor）对象的属性必须是：configurable、enumerable、writable 和 value。设置其中的一或多个值，可以修改对应的特性值。例如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var person = {};  Object.defineProperty(person,"name", {      writable:false,      value:"Nicholas"  });  console.log(person.name);//"Nicholas"  person.name ="Greg";  console.log(person.name);//"Nicholas" |

在调用 Object.defineProperty() 方法时，如果不指定 configurable、enumerable 和 writable 特性，其默认值都是 false 。

**【2】访问器属性**

访问器属性不包含数据值，它们包含一对 getter 和 setter 函数（不过，这两个函数都不是必需的）。在读取访问器属性时，会调用getter 函数，这个函数负责返回有效的值；在写入访问器属性时，会调用setter 函数并传入新值，这个函数负责决定如何处理数据。访问器属性有如下4 个特性。

* [[Configurable]]：表示能否通过 delete 删除属性从而重新定义属性，或者能否把属性修改为数据属性。默认值为true 。
* [[Enumerable]]：表示能否通过 for-in 循环返回属性。默认值为 true。
* [[Get]]：在读取属性时调用的函数。默认值为 undefined 。
* [[Set]]：在写入属性时调用的函数。默认值为 undefined 。

访问器属性不能直接定义，也必须使用 Object.defineProperty() 来定义。请看下面的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | var book = {      \_year: 2004  };  Object.defineProperty(book,"year", {      get: function () {          return this.\_year;      },      set: function (newValue) {          if (newValue > 2004) {              this.\_year = newValue;              console.log('set new value:'+ newValue)          }      }  });  book.year = 2005;//set new value:2005 |

**7.3  Object 新增 API**

ECMA-262 第 5 版对 Object 对象进行了增强，包括 defineProperty 在内，共定义了 9 个新的 API：

* create(prototype[,descriptors])：用于原型链继承。创建一个对象，并把其 prototype 属性赋值为第一个参数，同时可以设置多个 descriptors 。
* defineProperty(O,Prop,descriptor) ：用于定义对象属性的特性。
* defineProperties(O,descriptors) ：用于同时定义多个属性的特性。
* getOwnPropertyDescriptor(O,property)：获取 defineProperty 方法设置的 property 特性。
* getOwnPropertyNames：获取所有的属性名，不包括 prototy 中的属性，返回一个数组。
* keys()：和 getOwnPropertyNames 方法类似，但是获取所有的可枚举的属性，返回一个数组。
* preventExtensions(O) ：用于锁住对象属性，使其不能够拓展，也就是不能增加新的属性，但是属性的值仍然可以更改，也可以把属性删除。
* Object.seal(O) ：把对象密封，也就是让对象既不可以拓展也不可以删除属性（把每个属性的 configurable 设为 false），单数属性值仍然可以修改。
* Object.freeze(O) ：完全冻结对象，在 seal 的基础上，属性值也不可以修改（每个属性的 wirtable 也被设为 false）。