**EMCAScript 2020（ES11）**

1. 变量声明

1、js var声明变量

var x = 1;//是在同个函数里面，是一个局部变量

if (x === 1) {

var x = 2;

console.log(x);

// expected output: 2

}

console.log(x);

// expected output: 2

ECMAScript 的变量是松散类型的，所谓松散类型就是可以用来保存任何类型的数据。

如果你重新声明一个 JavaScript 变量，它将不会丢失其值（前提是该变量是在同个函数中）。

var message;

这行代码定义了一个名为 message 的变量，该变量可以用来保存任何值（像这样未经过初始化的 变量，会保存一个特殊的值—— undefined）。

在严格模式下，不能定义名为 eval 或 arguments 的变量，否则会导致语法错误。

var message = "hi";

在此，变量 message 中保存了一个字符串值 “hi” 。

像这样初始化变量并不会把它标记为字符串类型；

初始化的过程就是给变量赋一个值那么简单。

function test(){

var message = "hi"; // 局部变量

}

test();

alert(message); // 错误！

如果在函数中使用 var 定义一个变量，那么这个变量在函数退出后就会被销毁。

变量 message 是在函数中使用 var 定义的。

当函数被调用时，就会创建该变量并为其赋值。

而在此之后，这个变量又会立即被销毁，因此例子中的下一行代码就会导致错误。

2、变量提升

将赋值给未声明变量的值在执行赋值时将其隐式地创建为**全局变量**（它将成为全局对象的属性）。

声明和未声明变量之间的差异是：

（1）声明变量的作用域限制在其声明位置的上下文中，而**非声明变量**总是全局的。

function x() {

y = 1; // 在严格模式（strict mode）下会抛出 ReferenceError 异常

var z = 2;

}

x();

console.log(y); // 打印 "1"

console.log(z); // 抛出 ReferenceError: z 未在 x 外部声明

（2）声明变量在任何代码执行前创建，而非声明变量只有在执行赋值操作的时候才会被创建。

console.log(a); // 抛出ReferenceError。

console.log('still going...'); // 永不执行。

var a;

console.log(a); // 打印"undefined"（chrome是显示这个）或""（不同浏览器实现不同）。

console.log('still going...'); // 打印"still going..."。

（3）声明变量是它所在上下文环境的不可配置属性？？？，非声明变量是可配置的（如非声明变量可以被删除）。

var a = 1;

b = 2;

delete this.a; // 在严格模式（strict mode）下抛出TypeError，其他情况下执行失败并无任何提示。

delete this.b;

console.log(a, b); // 抛出ReferenceError。

// 'b'属性已经被删除。

由于这三个差异，未能声明变量将很可能导致意想不到的结果。

因此，建议始终声明变量，无论它们是在函数还是全局作用域内。

而且，在 ECMAScript 5 严格模式下，分配给未声明的变量会引发错误。

由于变量声明（以及其他声明）总是在任意代码执行之前处理的，所以在代码中的任意位置声明变量总是等效于在代码开头声明。

这意味着变量可以在声明之前使用，这个行为叫做“hoisting”。“hoisting”就像是把所有的变量声明移动到函数或者全局代码的开头位置。

bla = 2

var bla;

// ...

// 可以隐式地（implicitly）将以上代码理解为：

var bla;

bla = 2;

因此，建议始终在作用域顶部声明变量（全局代码的顶部和函数代码的顶部），这可以清楚知道哪些变量是函数作用域（本地），哪些变量在作用域链上解决。

重要的是，提升将影响变量声明，而不会影响其值的初始化。当到达赋值语句时，该值将确实被分配：

function do\_something() {

console.log(bar); // undefined

var bar = 111;

console.log(bar); // 111

}

// is implicitly understood as:

function do\_something() {

var bar;

console.log(bar); // undefined

bar = 111;

console.log(bar); // 111

}

3、例子

(1)、给两个变量赋值成字符串值：

var a = "A";

var b = a;

// 等效于：

var a, b = a = "A";

留意其中的顺序：

var x = y, y = 'A';

console.log(x + y); // undefinedA

在这里，x 和 y 在代码执行前就已经创建了，而赋值操作发生在创建之后。当”x = y”执行时，y 已经存在，所以不抛出 ReferenceError，并且它的值是’undefined’。

所以 x 被赋予 undefined 值。然后，y 被赋予’A’。于是，在执行完第一行之后，x === undefined && y === ‘A’ 才出现了这样的结果。

(2)、多个变量的初始化

var x = 0;//全局变量

function f(){

var x = y = 1; // x在函数内部声明，y不是！

}

f();

console.log(x, y); // 0, 1

// x 是全局变量。

// y 是隐式声明的全局变量。

(3)、隐式全局变量和外部函数作用域

看起来**像是(其实不是的)**隐式全局作用域的变量也有可能是其外部函数变量的引用。

var x = 0; // x是全局变量，并且赋值为0。

console.log(typeof z); // undefined，因为z还不存在。

function a() { // 当a被调用时，

var y = 2; // y被声明成函数a作用域的变量，然后赋值成2。

console.log(x, y); // 0 2

function b() { // 当b被调用时，

x = 3; // 全局变量x被赋值为3，不生成全局变量。

y = 4; // 已存在的外部函数的y变量被赋值为4，不生成新的全局变量。

z = 5; // 创建新的全局变量z，并且给z赋值为5。

} // (在严格模式下（strict mode）抛出ReferenceError)

b(); // 调用b时创建了全局变量z。

console.log(x, y, z); // 3 4 5

}

a(); // 调用a时同时调用了b。

console.log(x, z); // 3 5

console.log(typeof y); // undefined，因为y是a函数的本地（local）变量。

1、js let

let 语句声明一个**块级作用域**的本地变量？？？，并且可选的将其初始化为一个值。

let 演示

let x = 1;

if (x === 1) {

let x = 2;

console.log(x);

// expected output: 2

}

console.log(x);

// expected output: 1

2、描述

let 允许你声明一个作用域被限制在 块级中的变量、语句或者表达式。与 var 关键字不同的是， var 声明的变量只能是全局或者整个函数块的。 var 和 let 的不同之处在于后者是在编译时才初始化（见下面）。

就像 const 一样，let 不会在全局声明时（在最顶部的范围）创建 window 对象的属性？？？这是为什么。

（1）作用域规则

let 声明的变量只在其声明的块或子块中可用，这一点，与 var 相似。二者之间最主要的区别在于 var 声明的变量的作用域是整个封闭函数。

function varTest() {

var x = 1;

{

var x = 2; // 同样的变量!

console.log(x); // 2

}

console.log(x); // 2

}

function letTest() {

let x = 1;

{

let x = 2; // 不同的变量

console.log(x); // 2

}

console.log(x); // 1

}

在程序和方法的最顶端，let 不像 var 一样，let 不会在全局对象里新建一个属性。比如：

位于函数或代码顶部的 var 声明会给全局对象新增属性？？？无法理解这句话含义，而 let 不会。例如：

var x = 'global';

let y = 'global';

console.log(this.x); // "global"

console.log(this.y); // undefined

（2）模仿私有成员---？？？完全看不懂

（3）重复声明

在同一个函数或块作用域中重复声明同一个变量会引起 SyntaxError。

if (x) {

let axihe;

let axihe; // SyntaxError thrown.已经重复的声明

}

在 switch 语句中只有一个块，你可能因此而遇到错误。

let x = 1;

switch(x) {

case 0:

let axihe;

break;

case 1:

let axihe; // SyntaxError for redeclaration.

break;

}

然而，需要特别指出的是，一个嵌套在 case 子句中的块会创建一个新的块作用域的词法环境，就不会产生上诉重复声明的错误。这主要是使用了 块语句

let x = 1;

switch(x) {

case 0: {

let foo;

break;

}

case 1: {

let foo;//多了{},所以就没事了

break;

}

}

（4）暂存死区

与通过 var 声明的有初始化值 undefined 的变量不同，通过 let 声明的变量**直到它们的定义被执行时才初始化**。在变量初始化前访问该变量会导致 ReferenceError。该变量处在一个自块顶部到初始化处理的“暂存死区”中。

function do\_axihe() {

console.log(bar); // undefined

console.log(foo); // ReferenceError

var bar = 1;

let foo = 2;

}

do\_axihe()

（5）暂存死区与typeof

与通过 var 声明的变量，有初始化值 undefined 和只是未声明的变量不同的是，如果使用 typeof 检测在暂存死区中的变量，会抛出 ReferenceError 异常：

// prints out 'undefined'

console.log(typeof undeclaredVariable);

// results in a 'ReferenceError'

console.log(typeof i);

let i = 10;

（6）暂存死区和静态作用域 / 詞法作用域的相关例子

由于词法作用域，表达式 (foo + 55) 内的标识符 foo 被认为是 if (这里应该是错了，反正就是说在某个块中)块的 foo 变量，而不是值为 33 的块外面的变量 foo。

在同一行，这个 if 块中的 foo 已经在词法环境中被创建了，但是还没有到达（或者终止）它的初始化（这是语句本身的一部分）。

这个 if 块里的 foo 还依旧在暂存死区里。

function test(){

var foo = 33;

{

let foo = (foo + 55); // ReferenceError

}

}

test();

在以下情况下，这种现象可能会使您感到困惑。 let n of n.a 已经在 for 循环块的私有范围内。因此，标识符 n.a 被解析为位于指令本身 (“let n”) 中的“ n”对象的属性“ a”。

在没有执行到它的初始化语句之前，它仍旧存在于暂存死区中。

function go(n) {

// n here is defined!

console.log(n); // Object {a: [1,2,3]}

for (let n of n.a) { // ReferenceError

console.log(n);

}

}

go({a: [1, 2, 3]});？？？这里可能是一个json的数据格式。

（7）其他情况

用在块级作用域中时，let 将变量的作用域限制在块内，而 var 声明的变量的作用域是在函数内。

var a = 1;

var b = 2;

if (a === 1) {

var a = 11; // the scope is global

let b = 22; // the scope is inside the if-block

console.log(a); // 11

console.log(b); // 22

}

console.log(a); // 11

console.log(b); // 2

而这种 var 与 let 合并的声明方式会报 SyntaxError 错误，因为 var 会将变量提升至块的顶部，这会导致隐式地重复声明变量。这种情况下应该是在一个函数内部。

let x = 1;

{

var x = 2; // SyntaxError for re-declaration

}

---------------------------------------------------------

function a() { //

let x = 0;

console.log(x); // 0

if(1)

{

var x=1;//SyntaxError for re-declaration

console.log(x);

}

}

a();

1、js const 声明变量

(1) const 功能

常量是**块级**作用域，很像使用 **let** 语句定义的变量。

常量的值不能通过重新赋值来改变，并且不能重新声明。

(2) 描述

此声明创建一个常量，其作用域可以是全局或本地声明的块。 与 var 变量不同，全局常量不会变为**窗口对象**的属性。需要一个常数的初始化器；也就是说，您必须在声明的同一语句中**指定它的值**（这是有道理的，因为以后不能更改）。

const 声明创建一个值的只读引用。但这并不意味着它所持有的值是不可变的，只是变量标识符不能重新分配。例如，在引用内容是**对象**的情况下，这意味着可以改变对象的内容（例如，其参数）。

关于“暂存死区”的所有讨论都适用于 let 和 const。

一个常量不能和它所在作用域内的其他变量或函数拥有**相同的名称**。

// 注意: 常量在声明的时候可以使用大小写，但通常情况下全部用大写字母。

// 定义常量MY\_FAV并赋值7

const MY\_FAV = 7;

// 报错

MY\_FAV = 20;

// 输出 7

console.log("my favorite number is: " + MY\_FAV);

// 尝试重新声明会报错

const MY\_FAV = 20;

// MY\_FAV 保留给上面的常量，这个操作会失败

var MY\_FAV = 20;

// 也会报错

let MY\_FAV = 20;

// 注意块范围的性质很重要

if (MY\_FAV === 7) {

// 没问题，并且创建了一个块作用域变量 MY\_FAV，会把全局变量覆盖了，因为这是属于块的区域

// (works equally well with let to declare a block scoped non const variable)

let MY\_FAV = 20;

// MY\_FAV 现在为 20

console.log('my favorite number is ' + MY\_FAV);

// 这被提升到全局上下文并引发错误

var MY\_FAV = 20;

}

// MY\_FAV 依旧为7

console.log("my favorite number is " + MY\_FAV);

// 常量要求一个初始值

const FOO; // SyntaxError: missing = in const declaration

// 常量可以定义成对象

const MY\_OBJECT = {"key": "value"};

// 重写对象和上面一样会失败

MY\_OBJECT = {"OTHER\_KEY": "value"};

// 对象属性并不在保护的范围内，下面这个声明会成功执行

MY\_OBJECT.key = "otherValue";

// 也可以用来定义数组

const MY\_ARRAY = [];

// It's possible to push items into the array

// 可以向数组填充数据

MY\_ARRAY.push('A'); // ["A"]

// 但是，将一个新数组赋给变量会引发错误

MY\_ARRAY = ['B']

1、JS 基本的语法概念

(1) 严格模式

ECMAScript 5 引入了严格模式（strict mode）的概念。

严格模式是为 JavaScript 定义了一种不同的解析与执行模型。在严格模式下，ECMAScript 3 中的一些不确定的行为将得到处理，而且对某些不安全的操作也会抛出错误。

要在整个脚本中启用严格模式，可以在顶部添加如下代码：

"use strict";

这行代码看起来像是字符串，而且也没有赋值给任何变量，但其实它是一个编译指示（pragma），用于告诉支持的 JavaScript 引擎切换到严格模式。

这是为不破坏 ECMAScript 3 语法而特意选定的语法。

在函数内部的上方包含这条编译指示，也可以指定函数在严格模式下执行。

function doSomething(){

"use strict";

//函数体

}

严格模式下，JavaScript 的执行结果会有很大不同，因此本书将会随时指出严格模式下的区别。

支持严格模式的浏览器包括 IE10+、Firefox 4+、Safari 5.1+、Opera 12+和 Chrome。

(2) 语句

ECMAScript 中的语句以一个分号结尾；如果省略分号，则由解析器确定语句的结尾，如下例所示：

var sum = a + b // 即使没有分号也是有效的语句——不推荐

var diff = a - b; // 有效的语句——推荐

虽然语句结尾的分号不是必需的，但我们建议任何时候都不要省略它。因为加上这个分号可以避免很多错误（例如不完整的输入），

开发人员也可以放心地通过删除多余的空格来压缩 ECMAScript 代码（代码行结尾处没有分号会导致压缩错误）。

另外，加上分号也会在某些情况下增进**代码的性能**，因为这样解析器就不**必再花时间推测**应该在哪里插入分号了。

1. JS 数据类型整理

1、typeof 检测

typeof 就 是负责提供这方面信息的操作符。对一个值使用 typeof 操作符可能返回下列某个字符串：

“undefined” ——如果这个值未定义；

“boolean” ——如果这个值是布尔值；

“string” ——如果这个值是字符串；

“number” ——如果这个值是数值；

“object” ——如果这个值是对象或 null ；

“function” ——如果这个值是函数。

下面是几个使用 typeof 操作符的例子：

var message = "some string";

alert(typeof message); // "string"

alert(typeof(message)); // "string"

alert(typeof 95); // "number"

这几个例子说明， typeof 操作符的操作数可以是变量（ message ），也可以是数值字面量。

注意，typeof 是一个操作符而不是函数，因此例子中的圆括号尽管可以使用，但不是必需的。

有些时候， typeof 操作符会返回一些令人迷惑但技术上却正确的值。比如，调用 typeof null 会返回 “object” ，因为特殊值 null 被认为是一个空的对象引用。

Safari 5 及之前版本、Chrome 7 及之前版本在对正则表达式调用 typeof 操作符时会返回 “function” ，而其他浏览器在这种情况下会返回”object” 。

分享给大家一段实用的判断类型方法

**let \_typeof = function (data) {**

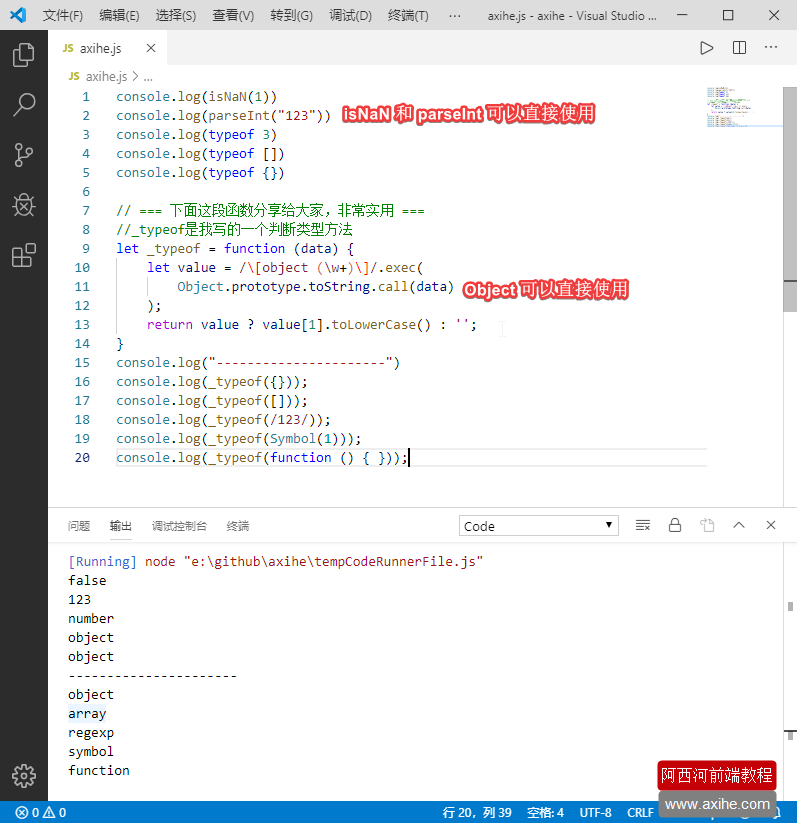
**let value = /\[object (\w+)\]/.exec(**

**Object.prototype.toString.call(data)**

**);**

**return value ? value[1].toLowerCase() : '';**

**}**



1. Undefined 类型

**Undefined 类型只有一个值，即特殊的 undefined**。

1. 在使用 var 声明变量但未对其加以初始化时，这个变量的值就是 undefined ，例如：

var message;

alert(message == undefined); //true

这个例子只声明了变量 message ，但未对其进行初始化。比较这个变量与 undefined 字面量，结果表明它们是相等的。这个例子与下面的例子是等价的：

var message = undefined;

alert(message == undefined); //true

这个例子使用 undefined 值显式初始化了变量 message 。但我们**没有必要**这么做，因为未经初始 化的值默认就会取得 undefined 值。

1. 不过，包含 undefined 值的变量与尚未定义的变量还是不一样的。看看下面这个例子：

var message; // 这个变量声明之后默认取得了 undefined 值

// 下面这个变量并没有声明

// var age

alert(message); // "undefined"

alert(age); // 产生错误

运行以上代码，第一个警告框会显示变量 message 的值，即 “undefined” 。

而第二个警告框——由于传递给 alert() 函数的是尚未声明的变量 age ——则会导致一个错误。

1. 对于尚未声明过的变量，只能执行一项操作，即使用 typeof 操作符检测其数据类型（对未经声明的变量调用 delete 不会导致错误，但这样做没什么实际意义，而且在严格模式下确实会导致错误）。

然而：对**未初始化的变量**执行 typeof 操作符会返回 undefined 值，而对**未声明的变量**执行 typeof 操作符同样也会返回 undefined 值。

**未声明的变量，只能被用typeof，除了 typeof 以外的任何使用都是会报错的！！！！**

1. var message; // 这个变量声明之后默认取得了 undefined 值

// 下面这个变量并没有声明

// var age

alert(typeof message); // "undefined"

alert(typeof age); // "undefined"

结果表明，对**未初始化和未声明的**变量执行 typeof 操作符都返回了 undefined 值；

这个结果有其逻辑上的合理性。

因为虽然这两种变量从技术角度看有本质区别，但实际上无论对哪种变量也不可能执行真正的操作。

即便未初始化的变量会自动被赋予 undefined 值，但显式地初始化变量依然是明智的选择。

如果能够做到这一点(指的就是：显式地初始化变量依然是明智的选择)，那么当 typeof 操作符返回 “undefined” 值时，我们就知道被检测的变量还没有被声明，而不是尚未初始化。

1. Null 类型

**Null 类型只有一个值的数据类型，这个特殊的值是 null 。**

1. 从逻辑角度来看， null 值表示一个空对象指针，而这也正是使用 typeof 操作符检测 null 值时会返回 “object” 的原因。

如下面的例子所示：

var car = null;

alert(typeof car); // "object"

(2) 如果定义的变量准备在将来用于保存对象，那么最好将该变量初始化为 null 而不是其他值。

这样一来，只要直接检查 null 值就可以知道相应的变量是否已经保存了一个对象的引用，如下面的例子所示：

if (car != null){

// 对 car 对象执行某些操作

}

(3) 实际上， undefined 值是派生自 null 值的，因此 ECMA-262 **规定对它们的相等性**测试要返回 true ：

alert(null == undefined); //true

alert(null === undefined); //false，它们是两种类型。

这里，位于 null 和 undefined 之间的相等操作符（ == ）总是返回 true ，不过要注意的是，这个操作符出于比较的目的会转换其操作数。

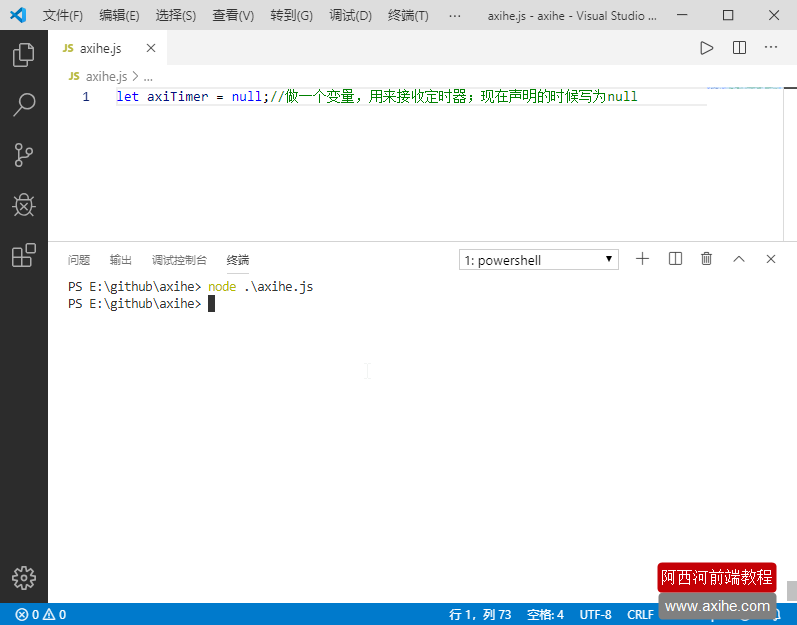
尽管 null 和 undefined 有这样的关系，但它们的用途完全不同。

如前所述，无论在什么情况下都没有必要把一个变量的值显式地设置为 undefined ，可是同样的规则对 null 却不适用。

换句话说，只要意在保存对象的变量**还没有真正保存对象**，就应该明确地让该变量保存 null 值。

这样做不仅可以体现 null 作为空对象指针的惯例，而且也有助于进一步区分 null 和 undefined 。

比如定时器我一般都是写为 null。



1. Boolean 类型

Boolean 类型是 ECMAScript 中使用得最多的一种类型，该类型**只有两个字面值： true 和 false 。**

这两个值与数字值不是一回事，因此 true 不一定等于 1，而 false 也不一定等于 0。

以下是为变量赋 Boolean 类型值的例子：

var found = true;

var lost = false;

需要注意的是， Boolean 类型的字面值 true 和 false 是区分大小写的。也就是说， True 和 False（以及其他的混合大小写形式）都不是 Boolean 值，只是标识符。

虽然 Boolean 类型的字面值只有两个，但 ECMAScript 中**所有类型的值**都有与这两个 Boolean 值等价的值。

要将一个值转换为其对应的 Boolean 值，可以调用转型函数 Boolean() ，如下例所示：

var message = "Hello world!";

var messageAsBoolean = Boolean(message);

在这个例子中，字符串 message 被转换成了一个 Boolean 值，该值被保存在 messageAsBoolean 变量中。

可以对任何数据类型的值调用 Boolean() 函数，而且总会返回一个 Boolean 值。至于返回的这个值是 true 还是 false ，取决于要转换值的数据类型及其实际值。

下表给出了各种数据类型及其对应的转换规则。

真假数据类型

| **数据类型** | **转换为 true 的值** | **转换为 false 的值** |
| --- | --- | --- |
| Boolean | true | false |
| String | 任何非空字符串 | ”” （空字符串） |
| Number | 任何非零数字值（包括无穷大） | 0 和 NaN |
| Object | 任何对象 | null |
| Undefined | n/a | undefined |

n/a（或 N/A），是 not applicable 的缩写，意思是“不适用”。

**注意上面这些为 false 的值一定一定要记牢固！非常重要！！！**

var message = "Hello world!";

if (message){

alert("Value is true");

}

运行这个示例，就会显示一个警告框，因为字符串 message 被自动转换成了对应的 Boolean 值（ true ）。

由于存在这种自动执行的 Boolean 转换，因此确切地知道在流控制语句中使用的是什么变量至关重要。

错误地使用一个对象而不是一个 Boolean 值，就有可能彻底改变应用程序的流程。

1. Number 类型

Number 类型应该是 ECMAScript 中最令人关注的数据类型了，这种类型使用 IEEE754 格式来表示整数和浮点数值（浮点数值在某些语言中也被称为双精度数值）。

为支持各种数值类型，ECMA-262 定义了不同的数值字面量格式。

最基本的数值字面量格式是十进制整数，十进制整数可以像下面这样直接在代码中输入：

var intNum = 55; // 整数

除了以十进制表示外，整数还可以通过八进制（以 8 为基数）或十六进制（以 16 为基数）的字面值来表示。

八进制字面值的第一位必须是**零（0）**，然后是八进制数字序列（0～7）。

如果字面值中的数值超出了范围，那么前导零将被忽略，后面的数值将被当作十进制数值解析。

请看下面的例子：

var octalNum1 = 070; // 八进制的 56

var octalNum2 = 079; // 无效的八进制数值——解析为 79

var octalNum3 = 08; // 无效的八进制数值——解析为 8

八进制字面量在严格模式下是无效的，会导致支持的 JavaScript 引擎抛出错误。

十六进制字面值的前两位必须是 0x，后跟任何十六进制数字（0～9 及 A～F）。其中，字母 A～F 可以大写，也可以小写。

如下面的例子所示：

var hexNum1 = 0xA; // 十六进制的 10

var hexNum2 = 0x1f; // 十六进制的 31

在进行算术计算时，所有以八进制和十六进制表示的数值最终都将被转换成十进制数值。

鉴于 JavaScript 中保存数值的方式，可以保存正零（+0）和负零（- 0）。

正零和负零被认为相等，但为了读者更好地理解上下文，这里特别做此说明。

1. 浮点数值

所谓浮点数值，就是该数值中必须包含一个小数点，并且小数点后面必须至少有一位数字。

虽然小数点前面可以没有整数，但我们不推荐这种写法。以下是浮点数值的几个例子：

var floatNum1 = 1.1;

var floatNum2 = 0.1;

var floatNum3 = .1; // 有效，但不推荐

由于保存浮点数值需要的内存空间是保存整**数值的两倍**，因此 ECMAScript 会不失时机地将浮点数值转换为整数值。

显然，如果小数点后面没有跟任何数字，那么这个数值就可以作为整数值来保存。同样地，如果浮点数值本身表示的就是一个整数（如 1.0），那么该值也会被转换为整数，如下面的例子所示：

var floatNum1 = 1.; // 小数点后面没有数字——解析为 1

var floatNum2 = 10.0; // 整数——解析为 10

对于那些极大或极小的数值，可以用 e 表示法（即科学计数法）表示的浮点数值表示。

用 e 表示法表示的数值等于 e 前面的数值乘以 10 的指数次幂。ECMAScript 中 e 表示法的格式也是如此，即前面是一个数值（可以是整数也可以是浮点数），中间是一个大写或小写的字母 E，后面是 10 的幂中的指数，该幂值将用来与前面的数相乘。

下面是一个使用 e 表示法表示数值的例子：

var floatNum = 3.125e7; // 等于 31250000

在这个例子中，使用 e 表示法表示的变量 floatNum 的形式虽然简洁，但它的实际值则是 31250000。在此，e 表示法的实际含义就是“3.125 乘以 10 7 ”。

也可以使用 e 表示法表示极小的数值，如 0.00000000000000003，这个数值可以使用更简洁的 3e-17 表示。在默认情况下，ECMASctipt 会将那些小数点后面带有 6 个零以上的浮点数值转换为以 e 表示法表示的数值（例如，0.0000003 会被转换成 3e7）。

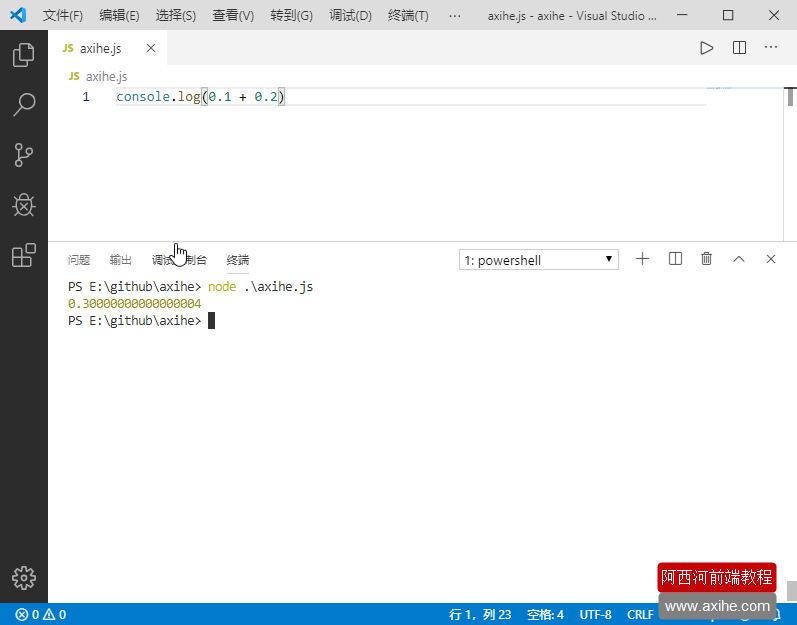
浮点数值的最高精度是 17 位小数，但在进行算术计算时其精确度远远不如整数。例如，0.1 加 0.2 的结果不是 0.3，而是 0.30000000000000004。这个小小的舍入误差会导致无法测试特定的浮点数值。

例如：

if (a + b == 0.3){ // 不要做这样的测试！

console.log("You got 0.3.");

}



在这个例子中，我们测试的是两个数的和是不是等于 0.3。如果这**两个数是 0.05 和 0.25，或者是 0.15 和 0.15 都不会有问题???**。而如前所述，如果这两个数是 0.1 和 0.2，那么测试将无法通过。

**因此，永远不要测试某个特定的浮点数值。**

关于浮点数值计算会产生舍入误差的问题，有一点需要明确：这是使用基于 IEEE754 数值的浮点计算的通病，ECMAScript 并非独此一家，这并不是 JS 这门语言的问题！

其他使用相同数值格式的语言也存在这个问题。

1. 数值范围

由于内存的限制，ECMAScript 并不能保存世界上所有的数值。ECMAScript 能够表示的最小数值保存在 Number.MIN\_VALUE 中——在大多数浏览器中，这个值是 5e-324；能够表示的最大数值保存在 Number.MAX\_VALUE 中——在大多数浏览器中，这个值是 1.7976931348623157e+308。

如果某次计算的结果得到了一个超出 JavaScript 数值范围的值，那么这个数值将被自动转换成特殊的 Infinity 值。具体来说，如果这个数值是负数，则会被转换成 -Infinity （负无穷），如果这个数值是正数，则会被转换成 Infinity （正无穷）。

如上所述，如果某次计算返回了正或负的 Infinity 值，那么该值将无法继续参与下一次的计算，因为 Infinity 不是能够参与计算的数值。要想确定一个数值是不是有穷的（换句话说，是不是位于最小和最大的数值之间），可以使用 isFinite() 函数。这个函数在参数位于最小与最大数值之间时会返回 true ，如下面的例子所示：

var result = Number.MAX\_VALUE + Number.MAX\_VALUE;

console.log(isFinite(result)); //false

尽管在计算中很少出现某些值超出表示范围的情况，但在执行极小或极大数值的计算时，检测监控这些值是可能的，也是必需的。

访问 Number.NEGATIVE\_INFINITY 和 Number.POSITIVE\_INFINITY 也可以得到负和正 Infinity 的值。可以想见，这两个属性中分别保存着 -Infinity 和 Infinity 。

(3) NaN

NaN ，即非数值（Not a Number）是一个特殊的数值，这个数值用于表示一个本来要返回数值的操作数未返回数值的情况（这样就不会抛出错误了）。例如，在其他编程语言中，任何数值除以 0 都会导致错误，从而停止代码执行。但在 ECMAScript 中，任何数值除以 0 会返回 NaN，因此不会影响其他代码的执行。

NaN 本身有两个非同寻常的特点。

首先，任何涉及 NaN 的操作（例如 NaN /10）都会返回 NaN ，这个特点在多步计算中有可能导致问题。

其次， NaN 与任何值都不相等，包括 NaN 本身。例如，下面的代码会返回 false ：

console.log(NaN == NaN); //false

针对 NaN 的这两个特点，ECMAScript 定义了 isNaN() 函数。这个函数接受一个参数，该参数可以是任何类型，而函数会帮我们确定这个参数是否“不是数值”。

isNaN() 在接收到一个值之后，会尝试将这个值转换为数值。某些不是数值的值会直接转换为数值，例如字符串 “10” 或 Boolean 值。而任何不能被转换为数值的值都会导致这个函数返回 true 。请看下面的例子：

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

console.log(isNaN(10)); //false（10 是一个数值）

console.log(isNaN("10")); //false（可以被转换成数值 10）

console.log(isNaN("blue")); //true（不能转换成数值）

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

这个例子测试了 5 个不同的值。测试的第一个值是 NaN 本身，结果当然会返回 true 。

然后分别测试了数值 10 和字符串 “10” ，结果这两个测试都返回了 false ，因为前者本身就是数值，而后者可以被转换成数值。

但是，字符串 “blue” 不能被转换成数值，因此函数返回了 true 。由于 Boolean 值 true 可以转换成数值 1，因此函数返回 false 。

尽管有点儿不可思议，但 isNaN() 确实也适用于对象。在基于对象调用 isNaN() 函数时，会首先调用对象的 valueOf() 方法？？？这个valueof是什么意思，然后确定该方法返回的值是否可以转换为数值。

如果不能，则基于这个返回值再调用 toString() 方法，再测试返回值。

而这个过程也是 ECMAScript 中内置函数和操作符的一般执行流程；

(4) 数值转换

有 3 个函数可以把非数值转换为数值： Number() 、 parseInt() 和 parseFloat() 。第一个函数，即转型函数 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。这 3 个函数对于同样的输入会有返回不同的结果。

**Number() 函数的转换规则如下。**

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

如果是数字值，只是简单的传入和返回。

如果是 null 值，返回 0。

如果是 undefined ，返回 NaN 。

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，即 “1”会变成 1， “123” 会变成 123，而 “011” 会变成 11（注意：前导的零被忽略了）；

如果字符串中包含有效的浮点格式，如 “1.1” ，则将其转换为对应的浮点数值（同样，也会忽略前导零）；

如果字符串中包含有效的十六进制格式，例如开头是 “0xf” ，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；parseInt转换为 NaN。

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 。

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。

根据这么多的规则使用 Number() 把各种数据类型转换为数值确实有点复杂。下面还是给出几个具 体的例子吧。

var num1 = Number("Hello world!"); //NaN

var num2 = Number(""); //0

var num3 = Number("000011"); //11

var num4 = Number(true); //1

首先，字符串 “Hello world!” 会被转换为 NaN ，因为其中不包含任何有意义的数字值。空字符串 会被转换为 0。字符串 “000011” 会被转换为 11，因为忽略了其前导的零。最后， true 值被转换为 1。

一元加操作符的操作与 Number() 函数相同。？？？一元操作符是指哪一个

由于 Number() 函数在转换字符串时比较复杂而且不够合理，因此在处理整数的时候更常用的是 parseInt() 函数。

**parseInt()函数**

转换的结果（使用了带第二个参数）：

如果是纯数字字符(包括十六进制)，则按基数转换后得到十进制数。

如果是英文字符分为

1. 开头是有数字字符按纯数字去解析。
2. 开头不是数字字符，返回 NaN。

*先看带不带参数，带的话再判断是几进制的（注意十六进制，可以不带前面的 “0x”），在一个字符一个字符判断，最后结果显示成十进制的。*

如

parseInt(“10blue”)，返回10，

parseInt(“0x10blue”),返回267

--------------------------------------------------------------

parseInt(“0x10blue”,10),用十进制去解析，返回 0

parseInt(“10blue”,16)，返回267

parseInt(“0x10blue”,16)返回 267

parseInt()函数在转换字符串时，更多的是看其是否符合数值模式。它会忽略字符串前面的空格，直至找到第一个非空格字符。如果第一个字符不是数字字符或者负号， parseInt() 就会返回 NaN ；也就是说，用 parseInt() 转换空字符串会返回 NaN （ Number() 对空字符返回 0）。如果第一个字符是数字字符， parseInt() 会继续解析第二个字符，直到解析完所有后续字符或者遇到了一个非数字字符。

例如， “1234blue” 会被转换为 1234，因为 “blue” 会被完全忽略。类似地， “22.5”会被转换为 22，因为小数点并不是有效的数字字符。 如果字符串中的第一个字符是数字字符， parseInt() 也能够识别出各种整数格式（即前面讨论的十进制、八进制和十六进制数）。也就是说，如果字符串以 “0x” 开头且后跟数字字符(如果强制按十进制解析，识别不了0x,这里没有指定，会可以自动识别)，就会将其当作一个十六进制整数；如果字符串以 “0” 开头且后跟数字字符，则会将其当作一个八进制数来解析。

为了更好地理解 parseInt() 函数的转换规则，下面给出一些例子：

var num1 = parseInt("1234blue"); // 1234

var num2 = parseInt(""); // NaN

var num3 = parseInt("0xA"); // 10（十六进制数）

var num4 = parseInt(22.5); // 22

var num5 = parseInt("070"); // 56（八进制数）

var num6 = parseInt("70"); // 70（十进制数）

var num7 = parseInt("0xf"); // 15（十六进制数）

parseInt(0x11,16)--->输出是23，原因是0x11会转换成17，十六进制17表示十进制23

parseInt(“0x11”,16)--->输出是17

在使用 parseInt() 解析像八进制字面量的字符串时，ECMAScript 3 和 5 存在分歧。例如：

//ECMAScript 3 认为是 56（八进制），ECMAScript 5 认为是 70（十进制）

var num = parseInt("070");

在 ECMAScript 3 JavaScript 引擎中， “070” 被当成八进制字面量，因此转换后的值是十进制的 56。而在 ECMAScript 5 JavaScript 引擎中， parseInt() 已经**不具有解析八进制值**的能力，因此前导的零会被认为无效，从而将这个值当成 “70” ，结果就得到十进制的 70。在 ECMAScript 5 中，即使是在非严格模式下也会如此。

为了消除在使用 parseInt() 函数时可能导致的上述困惑，可以为这个函数提供第二个参数：转换时使用的基数（即多少进制）。如果知道要解析的值是十六进制格式的字符串，那么指定基数 16 作为第二个参数，可以保证得到正确的结果，例如：

var num = parseInt("0xAF", 16); //175

实际上，如果指定了 16 作为第二个参数，字符串可以不带前面的 “0x” ，如下所示：

var num1 = parseInt("AF", 16); //175

var num2 = parseInt("AF"); //NaN

这个例子中的第一个转换成功了，而第二个则失败了。

差别在于第一个转换传入了基数，明确告诉 parseInt() 要解析一个十六进制格式的字符串；而第二个转换发现第一个字符不是数字字符，因此就自动终止了。

指定基数会影响到转换的输出结果。例如：

var num1 = parseInt("10", 2); //2 （按二进制解析）

var num2 = parseInt("10", 8); //8 （按八进制解析）

var num3 = parseInt("10", 10); //10 （按十进制解析）

var num4 = parseInt("10", 16); //16 （按十六进制解析）

不指定基数意味着让 parseInt() 决定如何解析输入的字符串，因此为了避免错误的解析，我们建议无论在什么情况下都明确指定基数。

多数情况下，我们要解析的都是十进制数值，因此始终将 10 作为第二个参数是非常必要的。

与 parseInt() 函数类似。

**parseFloat()函数**

parseFloat() 也是从第一个字符（位置 0）开始解析每个字符。

而且也是一直解析到字符串末尾，或者解析到遇见一个无效的浮点数字字符为止。也就是说，字符串中的第一个小数点是有效的，而第二个小数点就是无效的了，因此它后面的字符串将被忽略。举例来说，”22.34.5” 将会被转换为 22.34。

除了第一个小数点有效之外， parseFloat() 与 parseInt() 的第二个区别在于它始终都会忽略前导的零。

parseFloat() 可以识别前面讨论过的所有浮点数值格式，也包括十进制整数格式。但十六进制格式的字符串则始终会被转换成 0。

由于 parseFloat() 只解析十进制值，因此它没有用第二个参数指定基数的用法。最后还要注意一点：如果字符串包含的是一个可解析为整数的数（没有小数点，或者小数点后都是零）， parseFloat() 会返回整数。以下是使用 parseFloat() 转换数值的几个典型示例。

var num1 = parseFloat("1234blue"); //1234 （整数）

var num2 = parseFloat("0xA"); //0

var num3 = parseFloat("22.5"); //22.5

var num4 = parseFloat("22.34.5"); //22.34

var num5 = parseFloat("0908.5"); //908.5

var num6 = parseFloat("3.125e7"); //31250000