这些JS系列博客是记录我阅读 阮一峰老师的 《JavaScript 标准参考教程（alpha）》的札记，受益良多，所以就把我学到的，感悟到的一些东西记录下来，不会复制粘贴，只记录重点和感悟。

书里有句话，我非常喜欢，“我想写这本书，主要原因是自己需要。遇到问题，我首先查自己的笔记，如果笔记里没有，再到网上查，最后回过头把笔记补全。终于有一天，我意识到可以把笔记做成书，这就是这本书的由来。”

前言

1995年5月，Brendan Eich只用了10天，就设计完成了这种语言的第一版。它是一个大杂烩，语法有多个来源：

> 基本语法：借鉴C语言和Java语言。(JavaScript这个名字的原意是“很像Java的脚本语言”。) 数据结构：借鉴Java语言，包括将值分成原始值和对象两大类。 函数的用法：借鉴Scheme语言和Awk语言，将函数当作第一等公民，并引入闭包。 原型继承模型：借鉴Self语言（Smalltalk的一种变种）。 正则表达式：借鉴Perl语言。 字符串和数组处理：借鉴Python语言。

JavaScript 是一种轻量级的，嵌入式（embedded）的脚本语言。它本身不提供任何与 I/O（输入/输出）相关的 API，都要靠宿主环境（host）提供。它是目前唯一一种通用的浏览器脚本语言，所有浏览器都支持。

宿主环境有两种： 浏览器 和 Node 项目(服务器环境)。

JS 的灵活性体现在： **JavaScript 并不是纯粹的“面向对象语言”，还支持其他编程范式（比如函数式编程）**，既能面向对象编程，比如es6里引入的经典的 `class`等标志性的语法，也支持类似 C 语言清晰的过程式编程，也支持灵活的函数式编程。

JavaScript 语言本身，虽然是一种 解释型语言，但是在现代浏览器中，JavaScript 都是 编译后运行。程序会被高度优化，运行效率接近二进制程序。而且，JavaScript 引擎正在快速发展，性能将越来越好。（JS v8 引擎，有兴趣的可以去知乎上看一下 justjavac 的相关回答）

语法

JavaScript 的所有值都是对象。

这里只记录比较重要的语法点，想看详细的可以去看阮老师的教程。

标识符命名规则

很多人的 JS 里变量名随便起，其实是有问题的，我的导师教过我一句话：“一个好的命名效果是，即使一个完全不懂程序的人去看你的代码，也知道那个变量指的是什么意思。”

基本规则：

1. 第一个字符，可以是任意Unicode字母（包括英文字母和其他语言的字母），以及美元符号（$）和下划线（_）。不能是数字！
2. 第二个字符及后面的字符，除了Unicode字母、美元符号和下划线，还可以用数字0-9。
3. 标识符里不能有 `* + / - ^ % # & @`等等特殊字符。
4. **要有意义！**.

5. 中文也可以做为标识符（我不建议使用，最好使用英文~）
6. 这三个词虽然不是保留字，但是因为具有特别含义，也不应该用作标识符：`Infinity`、`NaN`、`undefined`。
7. 一些保留字不能用作标识符，如下：

> JavaScript保留字：arguments、break、case、catch、class、const、continue、debugger、default、delete、do、else、enum、eval、export、extends、false、finally、for、function、if、implements、import、in、instanceof、interface、let、new、null、package、private、protected、public、return、static、super、switch、this、throw、true、try、typeof、var、void、while、with、yield。

建议总是使用 var/let 命令声明变量。如果不声明却直接赋值，不仅不利于表达意图，而且不知不觉就会创建一个全局变量。

> 严格地说，var a = 1 与 a = 1，这两条语句的效果不完全一样，主要体现在delete命令无法删除前者。不过，绝大多数情况下，这种差异是可以忽略的。

变量提升

JavaScript引擎的工作方式是，先解析代码，获取所有被声明的变量，然后再一行一行地运行。于是所有的变量的声明语句，都会被提升到代码的头部。

注意：变量提升只对 `var` 命令声明的变量有效，如果一个变量不是用`var`命令声明的，就不会发生变量提升。

js 里的作用域

这里有一个很重要的概念——区块，JavaScript使用大括号，将多个相关的语句组合在一起，称为“区块”（block）。

区块中的变量与区块外的变量，属于同一个作用域。这说明区块不构成单独的作用域，与 不使用区块的情况没有任何区别。区块往往用来构成其他更复杂的语法结构，比如`for`、`if`、`while`、`function`等。

建议总是在 if 语句中使用大括号，因为这样方便插入语句。

标签（label）

JavaScript语言允许，语句的前面有标签（label），相当于定位符，用于跳转到程序的任意位置

```
top1:
  for (var i = 0; i < 3; i++) {
    for (var j = 0; j < 3; j++) {
      if (i === 1 && j === 1)
        break top1;
      console.log('i=' + i + ', j=' + j);
    }
  }
// i=0, j=0
// i=0, j=1
// i=0, j=2
// i=1, j=0
```

break命令后面加上了top标签（注意，top不用加引号），满足条件时，直接跳出双层循环。如果break语句后面不使用标签，则只能跳出内层循环，进入下一次的外层循环。

```
top2:
  for (var i = 0; i < 3; i++) {
    for (var j = 0; j < 3; j++){
      if (i === 1 && j === 1)
        continue top2;
      console.log('i=' + i + ', j=' + j);
    }
  }
// i=0, j=0
// i=0, j=1
// i=0, j=2
// i=1, j=0
// i=2, j=0
// i=2, j=1
// i=2, j=2
```

continue命令后面有一个标签名，满足条件时，会跳过当前循环，直接进入下一轮外层循环。如果continue语句后面不使用标签，则只能进入下一轮的内层循环。

其他

注意：优先采用“严格相等运算符”（===），而不是“相等运算符”（==）。

switch 语句后面的表达式与 case1 语句后面的表示式，在比较运行结果时，采用的是严格相等运算符（===），而不是相等运算符（==），这意味着比较时不会发生类型转换。

不管条件是否为真，do..while 循环至少运行一次，这是这种结构最大的特点。另外，while 语句后面的分号不能省略。

著名程序员 Jeff Atwood 提出了一条“Atwood 定律”：

> “所有可以用 JavaScript 编写的程序，最终都会出现 JavaScript 的版本。”(Any application that can be written in JavaScript will eventually be written in JavaScript.)

数据类型

1. 数值（number）：整数和小数（比如1和3.14）。
2. 字符串（string）：字符组成的文本（比如”Hello World”）。
3. 布尔值（boolean）：true（真）和false（假）两个特定值。
4. undefined：表示“未定义”或不存在，即由于目前没有定义，所以此处暂时没有任何值。
5. null：表示无值，即此处的值就是“无”的状态。
6. 对象（object）：各种值组成的集合。
7. Symbol： 原始数据类型，表示独一无二的值。

1，2，3，7都是原始类型（原子性，不能再分），6 成为合成类型（由多页元原子类型组合而成），4，5 是 6 的特殊值.

如何判断呢？

1. typeof运算符
2. instanceof运算符
3. Object.prototype.toString方法

typeof运算符

1. 原始类型： 数值、字符串、布尔值分别返回 number、string、boolean。
2. 函数： 函数返回 function。
3. undefined： undefined 返回 undefined。
4. 除此以外，其他情况都返回 object。

```
typeof window // "object"
typeof {} // "object"
typeof [] // "object", 说明数组本质上只是一种特殊的对象。
typeof null // "object"
typeof NaN // "Number"
```

null 为什么也是 object 呢？（看了教程才知道，原来是这样）

null 的类型也是 object，这是由于历史原因造成的。1995年 JavaScript语言的第一版，所有值都设计成32位，其中最低的3位用来表述数据类型，object 对应的值是 000 。当时，只设计了五种数据类型（对象、整数、浮点数、字符串和布尔值），完全没考虑 null，只把它当作 object 的一种特殊值，32位 全部为 0。这是 typeof null 返回 object 的根本原因。

并不是说 null 就属于对象，本质上null 是一个类似于 undefined 的特殊值。区分null 和 object? 使用 instanceof

typeof对数组（array）和对象（object）的显示结果都是object，那么怎么区分它们呢？还是可以使用 instanceof.

```
null instanceof Object // false

var o = {};
var a = [];

o instanceof Array // false
a instanceof Array // true
```

```
// 错误的写法
if (v) {
  // ...
```

```
}
// ReferenceError: v is not defined

// 正确的写法
if (typeof v === "undefined") {
  // ...
}
```

null 和 undefined

null 与 undefined 都可以表示“没有”，含义非常相似。

```
if (!undefined) {
  console.log('undefined is false');
}
// undefined is false

if (!null) {
  console.log('null is false');
}
// null is false

undefined == null
// true
```

目前 null 和 undefined 基本是同义的，只有一些细微的差别。区别：

```
typeof null // "object"
typeof undefined // "undefined"
```

null 被设计成可以自动转为0，而undefined 会转为undefined

```
Number(null) // 0
5 + null // 5

Number(undefined) // NaN
5 + undefined // NaN
```

1. null 表示空值，即该处的值现在为空。调用函数时，某个参数未设置任何值，这时就可以传入 null。书中有个例子举得特别好：比如，某个函数接受引擎抛出的错误作为参数，如果运行过程中未出错，那么这个参数就会传入null，表示未发生错误。
2. undefined表示“未定义”

```
// 1.变量声明了，但没有赋值
var i;
i // undefined

// 2.调用函数时，应该提供的参数没有提供，该参数等于undefined
function f(x) {
  return x;
}
f() // undefined

// 3.对象没有赋值的属性
var o = new Object();
o.p // undefined

// 4.函数没有返回值时，默认返回undefined
function f() {}
f() // undefined
```

数据类型的转换

JavaScript 是一种动态类型语言，变量没有类型限制，可以随时赋予任意值。但是数据本身和各种运算符是有类型的。如果运算符发现，数据的类型与预期不符，就会自动转换类型。

```
'4' - '3' // 1
```

布尔值的转换

1. 两元逻辑运算符： && (And)，|| (Or)
2. 前置逻辑运算符： ! (Not)
3. 相等运算符：===，!==，==，!=
4. 比较运算符：>，>=，<，<=

转换规则是除了下面六个值被转为 false，其他值都视为 true。

undefined null false 0 NaN ""或''（空字符串）

需要特别注意的是，空数组（[]）和空对象（{}）对应的布尔值，都是 true。

强制转换

Number()

将任意类型的值转化成数值。

原始类型值的转换规则

```
// 数值：转换后还是原来的值
Number(324) // 324

// 字符串：如果可以被解析为数值，则转换为相应的数值
Number('324') // 324

// 字符串：如果不可以被解析为数值，返回 NaN
Number('324abc') // NaN

// 空字符串转为0
Number('') // 0

// 布尔值：true 转成 1，false 转成 0
Number(true) // 1
Number(false) // 0

// undefined：转成 NaN
Number(undefined) // NaN

// null：转成0
Number(null) // 0

Number({}) // NaN
```

Number函数将字符串转为数值，要比parseInt函数严格很多。基本上，只要有一个字符无法转成数值，整个字符串就会被转为NaN。

```
parseInt('42 cats') // 42
Number('42 cats') // NaN
```

Number函数会自动过滤一个字符串前导和后缀的空格。

```
Number('\t\v\r12.34\n') // 12.34
```

补充：parseInt and parsefloat

parseInt

parseInt方法用于将字符串转为整数。parseInt的返回值只有两种可能，不是一个十进制整数，就是NaN。

```
parseInt('123') // 123

// 如果字符串头部有空格，空格会被自动去除。
parseInt('   81') // 81

// 如果parseInt的参数不是字符串，则会先转为字符串再转换。
```

```
parseInt(1.23) // 1
// 等同于
parseInt('1.23') // 1

// 字符串转为整数的时候，是一个个字符依次转换，如果遇到不能转为数字的字符，就不再进行下去，返回已经转好的部分。
parseInt('8a') // 8
parseInt('12**') // 12
parseInt('12.34') // 12
parseInt('15e2') // 15
parseInt('15px') // 15

// 如果字符串的第一个字符不能转化为数字（后面跟着数字的正负号除外），返回NaN。
parseInt('abc') // NaN
parseInt('.3') // NaN
parseInt('') // NaN
parseInt('+') // NaN
parseInt('+1') // 1

// 如果字符串以0x或0X开头，parseInt会将其按照十六进制数解析。
parseInt('0x10') // 16

// 如果字符串以0开头，将其按照10进制解析。
parseInt('011') // 11

// 对于那些会自动转为科学计数法的数字，parseInt会将科学计数法的表示方法视为字符串，因此导致一些奇怪的结果。
parseInt(1000000000000000000000.5) // 1
// 等同于
parseInt('1e+21') // 1

parseInt(0.0000008) // 8
// 等同于
parseInt('8e-7') // 8
```

parseInt方法还可以接受第二个参数（2到36之间），表示被解析的值的进制，返回该值对应的十进制数。默认情况下，parseInt的第二个参数为10，即默认是十进制转十进制。

如果第二个参数不是数值，会被自动转为一个整数。这个整数只有在2到36之间，才能得到有意义的结果，超出这个范围，则返回NaN。如果第二个参数是0、undefined和null，则直接忽略。

```
parseInt('1000') // 1000
// 等同于
parseInt('1000', 10) // 1000

parseInt('1000', 2) // 8
parseInt('1000', 6) // 216
parseInt('1000', 8) // 512

parseInt('10', 37) // NaN
parseInt('10', 1) // NaN
```

```
parseInt('10', 0) // 10
parseInt('10', null) // 10
parseInt('10', undefined) // 10
```

parseFloat()

parseFloat方法用于将一个字符串转为浮点数。尤其值得注意，parseFloat会将空字符串转为NaN。

这些特点使得parseFloat的转换结果不同于Number函数。

```
parseFloat('3.14') // 3.14

parseFloat('314e-2') // 3.14
parseFloat('0.0314E+2') // 3.14

parseFloat('3.14more non-digit characters') // 3.14

parseFloat('\t\v\r12.34\n ') // 12.34

// 如果参数不是字符串，或者字符串的第一个字符不能转化为浮点数，则返回NaN。
parseFloat([]) // NaN
parseFloat('FF2') // NaN
parseFloat('') // NaN

parseFloat(true)  // NaN
Number(true) // 1

parseFloat(null) // NaN
Number(null) // 0

parseFloat('') // NaN
Number('') // 0

parseFloat('123.45#') // 123.45
Number('123.45#') // NaN
```

对**象的**转换规则

`Number` 方法的参数是对象时，将返回 `NaN` ，除非是包含单个数值的数组。

```
Number({a: 1}) // NaN
Number([1, 2, 3]) // NaN
Number([5]) // 5
```

`Number` 背后的转换规则比较复杂

1. 调用对象自身的 valueOf 方法。如果返回原始类型的值，则直接对该值使用Number 函数，不再进行后续步骤。
2. 如果 valueOf 方法返回的还是对象，则改为调用对象自身的 toString 方法。如果 toString 方法返回原始类型的值，则对该值使用 Number 函数，不再进行后续步骤。
3. 如果 toString 方法返回的是对象，就报错。

```
var obj = {x: 1};
Number(obj) // NaN

// 等同于
if (typeof obj.valueOf() === 'object') {
  Number(obj.toString());
} else {
  Number(obj.valueOf());
}
```

String()

将任意类型的值转化成字符串。

原始类型值的转换规则String

1. 数值：转为相应的字符串。
2. 字符串：转换后还是原来的值。
3. 布尔值：true 转为 "true"，false 转为 "false"。
4. undefined：转为 "undefined"。
5. null：转为 "null"。

```
String(123) // "123"
String('abc') // "abc"
String(true) // "true"
String(undefined) // "undefined"
String(null) // "null"
```

对象的转换规则String

String方法的参数如果是对象，返回一个类型字符串；如果是数组，返回该数组的字符串形式。

```
String({a: 1}) // "[object Object]"
String([1, 2, 3]) // "1,2,3"
```

1. String方法背后的转换规则，与Number方法基本相同，只是互换了valueOf方法和toString方法的执行顺序。先调用对象自身的toString方法。如果返回原始类型的值，则对该值使用String函数，不再进行以下步骤。

2. 如果toString方法返回的是对象，再调用原对象的valueOf方法。如果valueOf方法返回原始类型的值，则对该值使用String函数，不再进行以下步骤。
3. 如果valueOf方法返回的是对象，就报错。

Boolean()

将任意类型的变量转为布尔值。 除了以下六个值的转换结果为false，其他的值全部为true。

undefined null -0 0或+0 NaN ''（空字符串）

```
Boolean(undefined) // false
Boolean(null) // false
Boolean(0) // false
Boolean(NaN) // false
Boolean('') // false
```

所有对象的布尔值都是true

注意，所有对象（包括空对象）的转换结果都是 `true` ，甚至连 `false` 对应的布尔对象 `new Boolean(false)` 也是 `true` 。

```
Boolean({}) // true
Boolean([]) // true
Boolean(new Boolean(false)) // true
```

自动转换类型

规则：预期什么类型的值，就调用该类型的转换函数。自动转换具有不确定性，而且不易除错，建议在预期为布尔值、数值、字符串 的地方，全部使用 Boolean、Number和String 函数进行显式转换。

我挑选了一些比较常见的例子，一看就懂

```
// 1. 不同类型的数据互相运算
123 + 'abc' // "123abc"
'5' + 1 // '51'
'5' + true // "5true"
'5' + false // "5false"
'5' + {} // "5[object Object]"
'5' + [] // "5"
'5' + function (){} // "5function (){}"
'5' + undefined // "5undefined"
'5' + null // "5null"

// 这一种很容易出错
var obj = {
  width: '100'
};
```

```
// 开发者可能期望返回120，但是由于自动转换，实际上返回了一个字符10020。
obj.width + 20 // "10020"

'5' - '2' // 3
'5' * '2' // 10
true - 1  // 0
false - 1 // -1
'1' - 1   // 0
'5' * []    // 0
false / '5' // 0
'abc' - 1   // NaN

// 2. 对非布尔值类型的数据求布尔值
if ('abc') {
  console.log('hello')
}  // "hello"

// 3. 对非数值类型的数据使用一元运算符（即“+”和“-”）
+ {foo: 'bar'} // NaN
- [1, 2, 3] // NaN
+'abc' // NaN
-'abc' // NaN
+true // 1
-false // 0

// 4. 将一个表达式转为布尔值
// 写法一
expression ? true : false

// 写法二
!! expression
// 结果为 expression || false

//因为 !!null == false, !!undefined == false
```

整数和浮点数

JavaScript 内部，所有数字都是以64位浮点数形式储存，即使整数也是如此。所以，1 与 1.0 是相同的，是同一个数。

```
1 === 1.0 // true
```

JavaScript 语言的底层根本没有整数，所有数字都是小数（64位浮点数）,所以，如果遇到只需要整数的情况：JavaScript 会自动把64位浮点数，转成32位整数，然后再进行运算.于是，会出现下面一些奇怪的情况：

```
0.1 + 0.2 === 0.3
// false
```

```
0.3 / 0.1
// 2.9999999999999996

(0.3 - 0.2) === (0.2 - 0.1)
// false
```

数据的精度

根据国际标准 IEEE 754，JavaScript 浮点数的64个二进制位，从最左边开始，是这样组成的。

第1位：符号位，0表示正数，1表示负数 第2位到第12位：指数部分 第13位到第64位：小数部分（即有效数字） 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。

IEEE 754 规定，有效数字第一位默认总是 1 ，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，最长可能为 52位。因此，JavaScript 提供的有效数字最长为 53个 二进制位。

```
(-1)^符号位 * 1.xx...xx * 2^指数位
```

精度最多只能到53个二进制位，这意味着，绝对值小于2的53次方的整数，即-(2^53-1)到2^53-1，都可以精确表示。

```
Math.pow(2, 53)
// 9007199254740992

Math.pow(2, 53) + 1
// 9007199254740992

Math.pow(2, 53) + 2
// 9007199254740994

Math.pow(2, 53) + 3
// 9007199254740996

Math.pow(2, 53) + 4
// 9007199254740996
```

从上面示例可以看到，大于2的53次方以后，整数运算的结果开始出现错误。所以，大于等于2的53次方的数值，都无法保持精度。

```
Math.pow(2, 53)
// 9007199254740992

// 多出的三个有效数字，将无法保存
9007199254740992111
// 9007199254740992000
```

上面示例表明，大于2的53次方以后，多出来的有效数字（最后三位的111）都会无法保存，变成0。

数值范围

64位浮点数的指数部分的长度是11个二进制位，意味着指数部分的最大值是2047（2的11次方减1）。即，JavaScript 能够表示的数值范围为 2^1024 到 2^(-1023)（开区间），超出这个范围的数无法表示。

如果指数部分等于或超过最大正值1024，JavaScript 会返回Infinity（关于Infinity的介绍参见下文），这称为“正向溢出”；如果等于或超过最小负值-1023（即非常接近0），JavaScript 会直接把这个数转为0，这称为“负向溢出”。

```
var x = 0.5;

for(var i = 0; i < 25; i++) {
  x = x * x;
}

x // 0
```

具体的最大值和最小值，JavaScript 提供Number对象的 MAX_VALUE和MIN_VALUE属性表示

```
Number.MAX_VALUE // 1.7976931348623157e+308
Number.MIN_VALUE // 5e-324
```

数值的表示法

JavaScript 的数值有多种表示方法，可以用字面形式直接表示，比如 35（十进制）和 0xFF（十六进制）。

数值也可以采用科学计数法表示，下面是几个科学计数法的例子。

```
123e3 // 123000
123e-3 // 0.123
-3.1E+12
.1e-23
```

以下两种情况，JavaScript 会自动将数值转为科学计数法表示，其他情况都采用字面形式直接表示。 （1）小数点前的数字多于21位。

```
1234567890123456789012
// 1.2345678901234568e+21

123456789012345678901
// 1234567890123456800000
```

（2）小数点后的零多于5个。

```
// 小数点后紧跟5个以上的零，
// 就自动转为科学计数法
0.0000003 // 3e-7

// 否则，就保持原来的字面形式
0.000003 // 0.000003
```

数值的进制

使用字面量（literal）时，JavaScript 对整数提供四种进制的表示方法：十进制、十六进制、八进制、2进制。

十进制：没有前导0的数值。 八进制：有前缀0o或0O的数值，或者有前导0、且只用到0-7的八个阿拉伯数字的数值。 十六进制：有前缀0x或0X的数值。 二进制：有前缀0b或0B的数值。

如果八进制、十六进制、二进制的数值里面，出现不属于该进制的数字，就会报错。

通常来说，有前导0的数值会被视为八进制，但是如果前导0后面有数字8和9，则该数值被视为十进制。

```
0888 // 888
0777 // 511
```

前导0表示八进制，处理时很容易造成混乱。ES5的严格模式和ES6，已经废除了这种表示法，但是浏览器目前还支持。

特殊数值

正零和负零

```
-0 === +0 // true
0 === -0 // true
0 === +0 // true

//唯一有区别的场合是，+0或-0当作分母，返回的值是不相等的。
(1 / +0) === (1 / -0) // false
```

因为除以正零得到 `+Infinity`，除以负零得到 `-Infinity`，这两者是不相等的

NaN

`NaN` 是 JavaScript 的特殊值，表示“非数字”（Not a Number），主要出现在将字符串解析成数字出错的场合。

```
5 - 'x' // NaN
```

上面代码运行时，会自动将字符串x转为数值，但是由于x不是数值，所以最后得到结果为 NaN，表示它是“非数字”（NaN）。

另外，一些数学函数的运算结果会出现 NaN。

```
Math.acos(2) // NaN
Math.log(-1) // NaN
Math.sqrt(-1) // NaN
// 0除以0也会得到NaN。

0 / 0 // NaN
```

需要注意的是，NaN不是一种独立的数据类型，而是一种特殊数值，它的数据类型依然属于Number，使用typeof运算符可以看得很清楚。

```
typeof NaN // 'number'

// NaN不等于任何值，包括它本身。
NaN === NaN // false
Boolean(NaN) // false

// NaN与任何数（包括它自己）的运算，得到的都是NaN。
NaN + 32 // NaN
NaN - 32 // NaN
NaN * 32 // NaN
NaN / 32 // NaN
```

isNaN方法可以用来判断一个值是否为NaN。不过，使用isNaN之前，最好判断一下数据类型。isNaN只对数值有效，如果传入其他值，会被先转成数值。比如，传入字符串的时候，字符串会被先转成 NaN，所以最后返回true，这一点要特别引起注意。也就是说，isNaN为true的值，有可能不是NaN，而是一个字符串。

```
isNaN(NaN) // true
isNaN(123) // false

isNaN('Hello') // true
// 相当于
isNaN(Number('Hello')) // true

// 同理
isNaN({}) // true
// 等同于
isNaN(Number({})) // true

isNaN(['xzy']) // true
```

```
// 等同于
isNaN(Number(['xzy'])) // true
```

所以，最好判断一下数据类型

```
function myIsNaN(value) {
  return typeof value === 'number' && isNaN(value);
}
```

判断NaN更可靠的方法是，利用NaN是JavaScript之中唯一不等于自身的值这个特点，进行判断。

```
function myIsNaN(value) {
  return value !== value;
}
```

Infinity

Infinity表示“无穷”，用来表示两种场景。一种是一个正的数值太大，或一个负的数值太小，无法表示；另一种是非0数值除以0，得到Infinity。

```
// 场景一
Math.pow(2, Math.pow(2, 100))
// Infinity

// 场景二
0 / 0 // NaN
1 / 0 // Infinity

Infinity === -Infinity // false

1 / -0 // -Infinity
-1 / -0 // Infinity
```

由于数值正向溢出（overflow）、负向溢出（underflow）和被0除，JavaScript都不报错，而是返回Infinity，所以单纯的数学运算几乎没有可能抛出错误。

Infinity大于一切数值（除了NaN），-Infinity小于一切数值（除了NaN）。

```
Infinity > 1000 // true
-Infinity < -1000 // true

// Infinity VS NaN
Infinity > NaN // false
-Infinity > NaN // false
Infinity < NaN // false
```

```
-Infinity < NaN // false

// Infinity VS number
5 * Infinity // Infinity
5 - Infinity // -Infinity
Infinity / 5 // Infinity
5 / Infinity // 0

// Infinity VS 0
0 * Infinity // NaN
0 / Infinity // 0
Infinity / 0 // Infinity

// Infinity VS null
null * Infinity // NaN
null / Infinity // 0
Infinity / null // Infinity

// Infinity VS undefined
undefined + Infinity // NaN
undefined - Infinity // NaN
undefined * Infinity // NaN
undefined / Infinity // NaN
Infinity / undefined // NaN

// Infinity VS Infinity
Infinity + Infinity // Infinity
Infinity * Infinity // Infinity

Infinity - Infinity // NaN
Infinity / Infinity // NaN
```

如何判断呢？使用isFinite函数返回一个布尔值，检查某个值是不是正常数值，而不是Infinity，即：是正常值，返回 true,不是，则返回false。

```
isFinite(Infinity) // false
isFinite(-1) // true
isFinite(true) // true
isFinite(NaN) // false,说明NaN不是一个正常值
```

对象

JavaScript 的所有值都是对象。所谓对象，就是一种无序的数据集合，由若干个“键值对”（key-value）构成。

又可以分为

1. 狭义的对象（object）
2. 数组（array）
3. 函数（function）：处理数据的方法。

对象的生成方法

```
var o1 = {};
var o2 = new Object();
var o3 = Object.create(Object.prototype);
```

关于对象的一些小规则

```
// 1. 对象的所有键名都是字符串，所以加不加引号都可以。

// 2. 如果键名是数值，会被自动转为字符串。
var o ={
  1: 'a',
  3.2: 'b',
  1e2: true,
  1e-2: true,
  .234: true,
  0xFF: true
};

o
// Object {
//   1: "a",
//   3.2: "b",
//   100: true,
//   0.01: true,
//   0.234: true,
//   255: true
// }

// 3. 如果键名不符合标识名的条件（比如第一个字符为数字，或者含有空格或运算符），也不是数字，则必须加上引号，否则会报错。

var o = {
  '1p': "Hello World",
  'h w': "Hello World",
  'p+q': "Hello World"
};

// 4. JavaScript的保留字可以不加引号当作键名。
var obj = {
  for: 1,
  class: 2
};

// 5. 对象的属性之间用逗号分隔，最后一个属性后面可以加逗号（trailing comma），也可以不加。
var o = {
  p: 123,
  m: function () { ... },
```

```
}
```

对象的引用

如果不同的变量名指向同一个对象，那么它们都是这个对象的引用，也就是说指向同一个内存地址。修改其中一个变量，会影响到其他所有变量。

```
var o1 = {};
var o2 = o1;

o1.a = 1;
o2.a // 1

o2.b = 2;
o1.b // 2

// 如果取消某一个变量对于原对象的引用，不会影响到另一个变量。
var o1 = {};
var o2 = o1;

o1 = 1;
o2 // {}
```

但是，这种引用只局限于对象，对于原始类型的数据则是传值引用，也就是说，都是值的拷贝。

```
var x = 1;
var y = x;

x = 2;
y // 1
```

上面的代码中，当x的值发生变化后，y的值并不变，这就表示y和x并不是指向同一个内存地址。

JavaScript规定，如果行首是大括号，一律解释为语句（即代码块）。如果要解释为表达式（即对象），必须在大括号前加上圆括号。

```
({ foo: 123 })
// 这种差异在eval语句中反映得最明显。

eval('{foo: 123}') // 123
eval('({foo: 123})') // {foo: 123}
```

属性的操作

属性的读取

```
var o = {
  p: 'Hello World'
};

o.p // "Hello World"
o['p'] // "Hello World"

// 1. 如果使用方括号运算符，键名必须放在引号里面，否则会被当作变量处理。但是，数字键可以不加引号，因为会被当作字符串处理。
var o = {
  0.7: 'Hello World'
};

o['0.7'] // "Hello World"
o[0.7] // "Hello World"

// 2. 方括号运算符内部可以使用表达式。
o['hello' + ' world']
o[3 + 3]

// 3. 数值键名不能使用点运算符（因为会被当成小数点），只能使用方括号运算符。
```

属性的赋值

JavaScript允许属性的“后绑定”，也就是说，你可以在任意时刻新增属性，没必要在定义对象的时候，就定义好属性。

```
var o = { p: 1 };

// 等价于
var o = {};
o.p = 1;
```

查看一个对象本身的所有属性，可以使用`Object.keys`方法。

```
var o = {
  key1: 1,
  key2: 2
};

Object.keys(o);
// ['key1', 'key2']
```

delete 属性

delete命令用于删除对象的属性，删除成功后返回true。

```
var o = {p: 1};
Object.keys(o) // ["p"]

delete o.p // true
o.p // undefined
Object.keys(o) // []
```

注意，删除一个不存在的属性，delete不报错，而且返回true。 因此，不能根据delete命令的结果，认定某个属性是存在的，只能保证读取这个属性肯定得到undefined。

```
var o = {};
delete o.p // true
```

只有一种情况，delete命令会返回false，那就是该属性存在，且不得删除。

```
var o = Object.defineProperty({}, 'p', {
  value: 123,
  configurable: false
});

o.p // 123
delete o.p // false
```

另外，需要注意的是，delete命令只能删除对象本身的属性，无法删除继承的属性。

```
var o = {};
delete o.toString // true
o.toString // function toString() { [native code] }
```

toString是对象o继承的属性，虽然delete命令返回true，但该属性并没有被删除，依然存在。

最后，delete命令不能删除var命令声明的变量，只能用来删除属性。

```
var p = 1;
delete p // false
delete window.p // false
```

上面命令中，p是var命令声明的变量，delete命令无法删除它，返回false。因为var声明的全局变量都是顶层对象的属性，而且默认不得删除。

in运算符

in运算符用于检查对象是否包含某个属性（注意，检查的是键名，不是键值），如果包含就返回true，否则返回false。

```
var o = { p: 1 };
'p' in o // true
```

in运算符的一个问题是，它不能识别对象继承的属性。

```
var o = new Object();
o.hasOwnProperty('toString') // false

'toString' in o // true
```

toString方法不是对象o自身的属性，而是继承的属性，hasOwnProperty方法可以说明这一点。但是，in运算符不能识别，对继承的属性也返回true。

for...in循环

for...in循环用来遍历一个对象的全部属性。

```
var o = {a: 1, b: 2, c: 3};

for (var i in o) {
  console.log(o[i]);
}
// 1
// 2
// 3
```

下面是一个使用for...in循环，提取对象属性的例子。

```
var obj = {
  x: 1,
  y: 2
};
var props = [];
var i = 0;

for(props[i++] in obj);

props // ['x', 'y']
```

for...in循环有两个使用注意点。

> 它遍历的是对象所有可遍历（enumerable）的属性，会跳过不可遍历的属性。 它不仅遍历对象自身的属性，还遍历继承的属性。

```
// name 是 Person 本身的属性
function Person(name) {
  this.name = name;
}

// describe是Person.prototype的属性
Person.prototype.describe = function () {
  return 'Name: '+this.name;
};

var person = new Person('Jane');

// for...in循环会遍历实例自身的属性（name），
// 以及继承的属性（describe）
for (var key in person) {
  console.log(key);
}
// name
// describe
```

上面代码中，name是对象本身的属性，`describe`是对象继承的属性，for...in循环的遍历会包括这两者。

如果只想遍历对象本身的属性，可以使用`hasOwnProperty`方法，在循环内部判断一下是不是自身的属性。

```
for (var key in person) {
  if (person.hasOwnProperty(key)) {
    console.log(key);
  }
}
// name
```

对象person其实还有其他继承的属性，比如toString。

```
person.toString()
// "[object Object]"
```

这个`toString`属性不会被`for...in`循环遍历到，因为它默认设置为“不可遍历”.

> 一般情况下，都是只想遍历对象自身的属性，所以不推荐使用for...in循环。

with语句

它的作用是操作同一个对象的多个属性时，提供一些书写的方便。

```
// 例一
with (o) {
  p1 = 1;
  p2 = 2;
}
// 等同于
o.p1 = 1;
o.p2 = 2;

// 例二
with (document.links[0]){
  console.log(href);
  console.log(title);
  console.log(style);
}
// 等同于
console.log(document.links[0].href);
console.log(document.links[0].title);
console.log(document.links[0].style);
```

注意，with区块内部的变量，必须是当前对象已经存在的属性，否则会创造一个当前作用域的全局变量。这是因为with区块没有改变作用域，它的内部依然是当前作用域。

```
var o = {};

with (o) {
  x = "abc";
}

o.x // undefined
x // "abc"
```

这是with语句的一个很大的弊病，就是绑定对象不明确。建议少用。不过，with语句少数有用场合之一，就是替换模板变量。

```
var o = {
  name: 'Alice'
};

var p = [];

with (o) {
  p.push('Hello ', name, '!');
};

p.join('') // "Hello Alice!"
```

数组（array）是按次序排列的一组值。每个值的位置都有编号（从0开始）任何类型的数据，都可以放入数组。

```
var arr = [
  {a: 1},
  [1, 2, 3],
  function() {return true;}
];

arr[0] // Object {a: 1}
arr[1] // [1, 2, 3]
arr[2] // function (){return true;}

var a = [[1, 2], [3, 4]];
a[0][1] // 2
a[1][1] // 4
```

本质上，数组属于一种特殊的对象。 typeof运算符会返回数组的类型是object。所以，也可以使用Object.keys() 获取键名

```
var arr = ['a', 'b', 'c'];

Object.keys(arr)
// ["0", "1", "2"]

var arr = ['a', 'b', 'c'];

arr['0'] // 'a'
arr[0] // 'a',原因是数值键名被自动转为了字符串。

var a = [];

a['1000'] = 'abc';
a[1000] // 'abc'

a[1.00] = 6;
a[1] // 6
```

基础

length属性

数组的length属性，返回数组的成员数量。

```
['a', 'b', 'c'].length // 3
```

JavaScript使用一个32位整数，保存数组的元素个数。这意味着，数组成员最多只有4294967295个（2^32 - 1）个，也就是说length属性的最大值就是4294967295。

只要是数组，就一定有length属性。该属性是一个动态的值，等于键名中的最大整数加上1。

```
var arr = ['a', 'b'];
arr.length // 2

arr[9] = 'd';
arr.length // 10

arr[1000] = 'e';
arr.length // 1001
```

另外，这也表明数组是一种动态的数据结构，可以随时增减数组的成员。

length属性是可写的。如果人为设置一个小于当前成员个数的值，该数组的成员会自动减少到length设置的值。

```
var arr = [ 'a', 'b', 'c' ];
arr.length // 3

arr.length = 2;
arr // ["a", "b"]

// 所以，可以利用这一属性，清空数组
arr.length = 0;
arr // []

// 反之
var a = ['a'];

a.length = 3;
a[1] // undefined
```

虽然 length 可以人为设置，但是也有约束，不合理就会报错

```
// 设置负值
[].length = -1
// RangeError: Invalid array length

// 数组元素个数大于等于2的32次方
[].length = Math.pow(2, 32)
// RangeError: Invalid array length

// 设置字符串
```

```
[].length = 'abc'
// RangeError: Invalid array length
```

注意！由于数组本质上是对象的一种，所以我们可以为数组添加属性，但是这不影响length属性的值。

```
var a = [];

a['p'] = 'abc';
a.length // 0

a[2.1] = 'abc';
a.length // 0
```

因为，length属性的值就是等于最大的数字键加1，而这个数组没有整数键，所以length属性保持为0。

类数组对象

如果一个对象的所有键名都是正整数或零，并且有length属性，那么这个对象就很像数组，语法上称为“类似数组的对象”（array-like object）。

```
var obj = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};

obj[0] // 'a'
obj[1] // 'b'
obj.length // 3
// obj对象没有数组的push方法，使用该方法就会报错。
obj.push('d') // TypeError: obj.push is not a function
```

“类似数组的对象”的根本特征，就是具有length属性。但是，这种length属性不是动态值，不会随着成员的变化而变化。

典型的“类似数组的对象”是函数的arguments对象，以及大多数 DOM 元素集，还有字符串。

```
// arguments对象
function args() { return arguments }
var arrayLike = args('a', 'b');

arrayLike[0] // 'a'
arrayLike.length // 2
arrayLike instanceof Array // false

// DOM元素集
```

```
var elts = document.getElementsByTagName('h3');
elts.length // 3
elts instanceof Array // false

// 字符串
'abc'[1] // 'b'
'abc'.length // 3
'abc' instanceof Array // false
```

数组的 slice 方法可以将 类数组对象 变成真正的数组。

```
var arr = Array.prototype.slice.call(arrayLike);
```

除了转为真正的数组，“类似数组的对象”还有一个办法可以使用数组的方法，就是通过call()把数组的方法放到对象上面。

```
// forEach 方法
function logArgs() {
  Array.prototype.forEach.call(arguments, function (elem, i) {
    console.log(i+'. '+elem);
  });
}

// 等同于 for 循环
function logArgs() {
  for (var i = 0; i < arguments.length; i++) {
    console.log(i + '. ' + arguments[i]);
  }
}

// 字符串也可以
Array.prototype.forEach.call('abc', function (chr) {
  console.log(chr);
});
// a
// b
// c
```

这种方法比·直接使用数组原生的forEach要慢·，所以最好还是先将“类似数组的对象”转为真正的数组，然后再直接调用数组的forEach方法。

```
var arr = Array.prototype.slice.call('abc');
arr.forEach(function (chr) {
  console.log(chr);
});
// a
```

```
// b
// c
```

遍历数组

由于数组也是一中特殊的对象，因此可以用 for...in, 但是，它不仅会遍历数组所有的数字键，还会遍历非数字键。

```
var a = [1, 2, 3];
a.foo = true;

for (var key in a) {
  console.log(key);
}
// 0
// 1
// 2
// foo
```

所以，不推荐使用for...in遍历数组。

数组的遍历可以考虑使用for循环或while循环。

```
var a = [1, 2, 3];

// for循环
for(var i = 0; i < a.length; i++) {
  console.log(a[i]);
}

// while循环
var i = 0;
while (i < a.length) {
  console.log(a[i]);
  i++;
}

var l = a.length;
while (l--) {
  console.log(a[l]);
}

var colors = ['red', 'green', 'blue'];
colors.forEach(function (color) {
  console.log(color);
});
```

数组的空位

当数组的某个位置是空元素，即两个逗号之间没有任何值，我们称该数组存在空位（hole）。

```
var a = [1, , 1];
a.length // 3
```

上面代码表明，数组的空位不影响length属性。

需要注意的是，如果最后一个元素后面有逗号，并不会产生空位。也就是说，有没有这个逗号，结果都是一样的。

```
var a = [1, 2, 3,];

a.length // 3
a // [1, 2, 3]

var b = [, , ,];
b[1] // undefined
```

使用delete命令删除一个数组成员，会形成空位，并且不会影响length属性。

```
var a = [1, 2, 3];
delete a[1];

a[1] // undefined
a.length // 3
```

length属性不过滤空位。所以，使用length属性进行数组遍历，一定要非常小心。

空位和 undefined的区别

数组的某个位置是空位，与某个位置是undefined，是不一样的。如果是空位，使用数组的forEach方法、for...in结构、以及Object.keys方法进行遍历，空位都会被跳过。

因为，空位就是数组没有这个元素，所以不会被遍历到，而undefined则表示数组有这个元素，值是undefined，所以遍历不会跳过。

```
var a = [, , ,];

a.forEach(function (x, i) {
  console.log(i + '. ' + x);
})
// 不产生任何输出
```

```
for (var i in a) {
  console.log(i);
}
// 不产生任何输出

Object.keys(a)
// []
```

如果某个位置是undefined，遍历的时候就不会被跳过。

```
var a = [undefined, undefined, undefined];

a.forEach(function (x, i) {
  console.log(i + '. ' + x);
});
// 0. undefined
// 1. undefined
// 2. undefined

for (var i in a) {
  console.log(i);
}
// 0
// 1
// 2

Object.keys(a)
// ['0', '1', '2']
```

常用函数

在写 JS 的过程中，经常会遇到控制台报错，这期我们就来探索一下 JS 里的错误类型和机制。

JavaScript解析或执行时，一旦发生错误，引擎就会抛出一个错误对象。然后整个程序就中断在发生错误的地方，不再往下执行。

JavaScript原生提供一个Error构造函数，所有抛出的错误都是这个构造函数的实例。

Error对象的实例必须有message属性，表示出错时的提示信息，其他属性则没有提及。大多数JavaScript引擎，对Error实例还提供name和stack属性，分别表示错误的名称和错误的堆栈，但它们是非标准的，不是每种实现都有。

message：错误提示信息 name：错误名称（非标准属性） stack：错误的堆栈（非标准属性）

```
function throwit() {
  throw new Error('');
}
```

```
function catchit() {
  try {
    throwit();
  } catch(e) {
    console.log(e.stack); // print stack trace
  }
}

catchit()
// Error
//    at throwit (~/examples/throwcatch.js:9:11)
//    at catchit (~/examples/throwcatch.js:3:9)
//    at repl:1:5
```

错误类型

SyntaxError

SyntaxError是 解析代码时发生的语法错误。

```
// 变量名错误
var 1a;

// 缺少括号
console.log 'hello');
```

ReferenceError

ReferenceError是引用一个不存在的变量时发生的错误。

```
unknownVariable
// ReferenceError: unknownVariable is not defined
```

另一种触发场景是，将一个值分配给无法分配的对象，比如对函数的运行结果或者this赋值。

```
console.log() = 1
// ReferenceError: Invalid left-hand side in assignment

this = 1
// ReferenceError: Invalid left-hand side in assignment
```

上面代码对函数console.log的运行结果和this赋值，结果都引发了ReferenceError错误。

RangeError

RangeError是当一个值超出有效范围时发生的错误。主要有几种情况，一是数组长度为负数，二是Number对象的方法参数超出范围，以及函数堆栈超过最大值。

```
new Array(-1)
// RangeError: Invalid array length

(1234).toExponential(21)
// RangeError: toExponential() argument must be between 0 and 20
```

TypeError

TypeError是变量或参数不是预期类型时发生的错误。比如，对字符串、布尔值、数值等原始类型的值使用new命令，就会抛出这种错误，因为new命令的参数应该是一个构造函数。

```
new 123
//TypeError: number is not a func

var obj = {};
obj.unknownMethod()
// TypeError: obj.unknownMethod is not a function
```

上面代码的第二种情况，调用对象不存在的方法，会抛出TypeError错误。

URIError

URIError是URI相关函数的参数不正确时抛出的错误，主要涉及encodeURI()、decodeURI()、encodeURIComponent()、decodeURIComponent()、escape()和unescape()这六个函数。

```
decodeURI('%2')
// URIError: URI malformed
```

EvalError

eval函数没有被正确执行时，会抛出EvalError错误。该错误类型已经不再在ES5中出现了，只是为了保证与以前代码兼容，才继续保留。

以上这6种派生错误，连同原始的Error对象，都是构造函数。开发者可以使用它们，人为生成错误对象的实例。

```
new Error('出错了！');
new RangeError('出错了，变量超出有效范围！');
new TypeError('出错了，变量类型无效！');
```

上面代码新建错误对象的实例，实质就是手动抛出错误。可以看到，错误对象的构造函数接受一个参数，代表错误提示信息（message）。

自定义错误

```
function UserError(message) {
  this.message = message || '默认信息';
  this.name = 'UserError';
}

UserError.prototype = new Error();
UserError.prototype.constructor = UserError;

new UserError('自定义错误! ');
```

throw 语句

throw可以接受各种值作为参数。JavaScript引擎一旦遇到throw语句，就会停止执行后面的语句，并将throw语句的参数值，返回给用户。

```
// 抛出一个字符串
throw "Error! ";

// 抛出一个数值
throw 42;

// 抛出一个布尔值
throw true;

// 抛出一个对象
throw {toString: function() { return "Error!"; }
```

如果只是简单的错误，返回一条出错信息就可以了，但是如果遇到复杂的情况，就需要在出错以后进一步处理。这时最好的做法是使用throw语句手动抛出一个Error对象。

```
throw new Error('出错了!');

// 或者
function UserError(message) {
  this.message = message || "默认信息";
  this.name = "UserError";
}

UserError.prototype.toString = function() {
  return this.name + ': "' + this.message + '"';
}
```

```
throw new UserError("出错了！");

// 可以通过自定义一个assert函数，规范化throw抛出的信息。

function assert(expression, message) {
  if (!expression)
    throw {name: 'Assertion Exception', message: message};
}

assert(typeof myVar != 'undefined', 'myVar is undefined!');
```

try…catch结构

对错误进行处理，需要使用try...catch结构。

```
try {
  throw new Error('出错了!');
} catch (e) {
  console.log(e.name + ": " + e.message);
  console.log(e.stack);
}
// Error: 出错了!
//    at <anonymous>:3:9
//    ...
```

catch接受一个参数，表示try代码块抛出的值。

```
function throwIt(exception) {
  try {
    throw exception;
  } catch (e) {
    console.log('Caught: ' + e);
  }
}

throwIt(3);
// Caught: 3
throwIt('hello');
// Caught: hello
throwIt(new Error('An error happened'));
// Caught: Error: An error happened
```

catch代码块捕获错误之后，程序不会中断，会按照正常流程继续执行下去。

catch代码块之中，还可以再抛出错误，甚至使用嵌套的try...catch结构。

```
var n = 100;

try {
  throw n;
} catch (e) {
  if (e <= 50) {
    // ...
  } else {
    throw e;
  }
}
```

为了捕捉不同类型的错误，catch代码块之中可以加入判断语句。

```
try {
  foo.bar();
} catch (e) {
  if (e instanceof EvalError) {
    console.log(e.name + ": " + e.message);
  } else if (e instanceof RangeError) {
    console.log(e.name + ": " + e.message);
  }
  // ...
}
```

这种结构多多少少是对结构化编程原则一种破坏，处理不当就会变成类似goto语句的效果，应该谨慎使用。

finally代码块

try...catch结构允许在最后添加一个finally代码块，表示不管是否出现错误，都必需在最后运行的语句。

即使有return语句在前，finally代码块依然会得到执行，且在其执行完毕后，才会显示return语句的值。

```
function idle(x) {
  try {
    console.log(x);
    return 'result';
  } finally {
    console.log("FINALLY");
  }
}

idle('hello')
// hello
// FINALLY
// "result"
```

注意，return的执行是排在finally代码之前，只是等finally代码执行完毕后才返回。

```
var count = 0;
function countUp() {
  try {
    return count;
  } finally {
    count++;
  }
}

countUp()
// 0
count
// 1
```

经典用法

```
openFile();

try {
  writeFile(Data);
} catch(e) {
  handleError(e);
} finally {
  closeFile();
}
```

1. 首先打开一个文件，然后在try代码块中写入文件，如果没有发生错误，则运行finally代码块关闭文件；
2. 一旦发生错误，则先使用catch代码块处理错误，再使用finally代码块关闭文件。

调用顺序

```
function f() {
  try {
    console.log(0);
    throw 'bug';
  } catch(e) {
    console.log(1);
    return true; // 重点：这句原本会延迟到finally代码块结束再执行
    console.log(2); // 不会运行
  } finally {
    console.log(3);
    return false; // 这句会覆盖掉前面那句return，因为是return，所以函数体直接返回false
了，catch里的return没有被执行
    console.log(4); // 不会运行
  }
```

```
  console.log(5); // 不会运行
}

var result = f();
// 0
// 1
// 3

result
// false
```

函数就是一段可以反复调用的代码块。函数还能接受输入的参数，不同的参数会返回不同的值。

基础概念

函数的声明

```
// 1. function命令
function print(s) {
  console.log(s);
}

// 2. 函数表达式：将一个匿名函数赋值给变量。这时，这个匿名函数又称函数表达式（Function Expression）
var print = function(s) {
  console.log(s);
};

// *****************
// 注意：采用函数表达式声明函数时，function命令后面不带有函数名。如果加上函数名，该函数名只在函数体内部有效，在函数体外部无效。
var print = function x(){
  console.log(typeof x);
};    // 注意带上分号，因为时声明语句

x
// ReferenceError: x is not defined
print()
// function

// 这种写法的用处有两个，一是可以在函数体内部调用自身，二是方便除错（除错工具显示函数调用栈时，将显示函数名，而不再显示这里是一个匿名函数）。

//*******************

//3. Function构造函数（不直观，故不推荐使用），可以传递任意数量的参数给Function构造函数，只有最后一个参数会被当做函数体，如果只有一个参数，该参数就是函数体。
var add = new Function(
  'x',
  'y',
  'return x + y'
```

```
);

// 等同于
function add(x, y) {
  return x + y;
}
```

如果同一个函数被多次声明，后面的声明就会覆盖前面的声明。

第一等公民

JavaScript语言将函数看作一种值，与其它值（数值、字符串、布尔值等等）地位相同。凡是可以使用值的地方，就能使用函数。

```
function add(x, y) {
  return x + y;
}

// 将函数赋值给一个变量
var operator = add;

// 将函数作为参数和返回值
function a(op){
  return op;
}
a(add)(1, 1)
// 2
```

函数名的提升

采用function命令声明函数时，整个函数会像变量声明一样，被提升到代码头部。

```
// 1. 使用function声明
f();
function f() {}

// 2. 使用var声明就会报错
f();
var f = function (){};
// TypeError: undefined is not a function
// 等同于
var f;
f();
f = function () {};
```

因此，如果同时采用function命令和赋值语句声明同一个函数，最后总是采用赋值语句的定义。

```
var f = function() {
  console.log('1');
}

function f() {
  console.log('2');
}

f() // 1
```

另外，根据ECMAScript的规范，不得在非函数的代码块中声明函数。

在条件语句中声明函数，可能是无效的，这是非常容易出错的地方。要达到在条件语句中定义函数的目的，只有使用函数表达式。

```
if (false) {
  var f = function () {};
}

f() // undefined
```

函数的属性和方法

name 属性 和 length 属性

name属性返回紧跟在function关键字之后的那个函数名

```
function f1() {}
f1.name // 'f1'

var f2 = function () {};
f2.name // ''

var f3 = function myName() {};
f3.name // 'myName'
// 对于f3来说，返回函数表达式的名字（真正的函数名还是f3，myName这个名字只在函数体内部可用）
```

length属性返回函数预期传入的参数个数，即函数定义之中的参数个数。

```
function f(a, b) {}
f.length // 2
```

length属性提供了一种机制，判断定义时和调用时参数的差异，以便实现面向对象编程的”方法重载“（overload）。

toString()

函数的toString方法返回函数的源码。包括注释。

```
function f() {/*
  这是一个
  多行注释
*/}

f.toString()
// "function f(){/*
//   这是一个
//   多行注释
// */}"
```

利用其实现 多行字符串；

```
var multiline = function(fn) {
  var arr = fn.toString().split('\n');
  return arr.slice(1, arr.length - 1).join('\n');
};

function f() {/*
  这是一个
  多行注释
*/}

multiline(f);
// " 这是一个
//   多行注释"
```

函数作用域问题

作用域（scope）指的是变量存在的范围。

Javascript只有两种作用域：一种是全局作用域，变量在整个程序中一直存在，所有地方都可以读取；另一种是函数作用域，变量只在函数内部存在。

函数内部定义的变量，会在该作用域内覆盖同名全局变量。

```
var v = 1;

function f(){
  var v = 2;
  console.log(v);
}
```

```
f() // 2
v // 1

// 如果改成这样
function f(){
  v = 2;
  console.log(v);
}

f() // 2
v // 2
```

注意，对于var命令来说，局部变量只能在函数内部声明，在其他区块中声明，一律都是全局变量。

```
if (true) {
  var x = 5;
}
console.log(x);  // 5
```

函数内部的变量提升

与全局作用域一样，函数作用域内部也会产生“变量提升”现象。var命令声明的变量，不管在什么位置，变量声明都会被提升到函数体的头部。

函数本身的作用域

函数本身也是一个值，也有自己的作用域。它的作用域与变量一样，就是其声明时所在的作用域，与其运行时所在的作用域无关。

```
var a = 1;
var x = function () {
  console.log(a);
};

function f() {
  var a = 2;
  x();
}

f() // 1
```

很容易犯错的一点是，如果函数A调用函数B，却没考虑到函数B不会引用函数A的内部变量。

```
var x = function () {
  console.log(a);
```

```
};

function y(f) {
  var a = 2;
  f();
}

y(x)
// ReferenceError: a is not defined
```

同样的，函数体内部声明的函数，作用域绑定函数体内部。

```
function foo() {
  var x = 1;
  function bar() {
    console.log(x);
  }
  return bar;
}

var x = 2;
var f = foo();
f() // 1
```

参数

函数运行的时候，有时需要提供外部数据，不同的外部数据会得到不同的结果，这种外部数据就叫参数。

函数参数不是必需的，Javascript允许省略参数。

```
function f(a, b) {
  return a;
}

f(1, 2, 3) // 1
f(1) // 1
f() // undefined，被省略的参数的值就变为undefined

f.length // 2 ,函数的length属性与实际传入的参数个数无关，只反映函数预期传入的参数个数。

// 但是，没有办法只省略靠前的参数，而保留靠后的参数。如果一定要省略靠前的参数，只有显式传入undefined。

f( , 1) // SyntaxError: Unexpected token ,(…)
f(undefined, 1) // undefined
```

可以为函数的参数设置默认值。下面是比较好的一种方式

```
function f(a) {
  // a = a || 1; // 不好，因为只有布尔运算为true时，才会返回a。可是，除了undefined以外，0、空字符、null等的布尔值也是false。明明有参数的情况下，也会返回默认值
  (a !== undefined && a !== null) ? a = a : a = 1;
  return a;
}

f() // 1
f('') // ""
f(0) // 0
```

传递方式

函数参数如果是原始类型的值（数值、字符串、布尔值），传递方式是传值传递（passes by value）。这意味着，在函数体内修改参数值，不会影响到函数外部。

```
var p = 2;

function f(p) {
  p = 3;
}
f(p);

p // 2
```

变量p是一个原始类型的值，传入函数f的方式是传值传递。因此，在函数内部，p的值是原始值的拷贝，无论怎么修改，都不会影响到原始值。

但是，如果函数参数是复合类型的值（数组、对象、其他函数），传递方式是传址传递（pass by reference）。也就是说，传入函数的原始值的地址，因此在函数内部修改参数，将会影响到原始值。

```
var obj = {p: 1};

function f(o) {
  o.p = 2;
}
f(obj);

obj.p // 2
```

注意，如果函数内部修改的，不是参数对象的某个属性，而是替换掉整个参数，这时不会影响到原始值。

```
var obj = [1, 2, 3];

function f(o){
  o = [2, 3, 4];
```

```
}
f(obj);

obj // [1, 2, 3]
```

这是因为，形式参数（o）与实际参数obj 存在一个赋值关系。

```
// 函数f内部
o = obj;
```

对o的修改都会反映在obj身上。但是，如果对o赋予一个新的值，就等于切断了o与obj的联系，导致此后的修改都不会影响到obj了。

某些情况下，如果需要对某个原始类型的变量，获取传址传递的效果，可以将它写成全局对象的属性。

```
var a = 1;

function f(p) {
  window[p] = 2;
}
f('a');

a // 2
```

上面代码中，变量a本来是传值传递，但是写成window对象的属性，就达到了传址传递的效果。

注意与不传值的情况进行对比

```
var p = 2;

function f() {
  p = 3;
}
f();

p // 3
```

如果有同名的参数，则取最后出现的那个值。

```
function f(a, a) {
  console.log(a);
}

f(1, 2) // 2
```

```
f(1) // undefined

// 这时，如果要获得第一个a的值，可以使用arguments对象。
f(1) // 1
```

arguments对象

一种机制，可以在函数体内部读取所有参数。

arguments 对象包含了函数运行时的所有参数，arguments[0]就是第一个参数，arguments[1]就是第二个参数，以此类推。这个对象只有在函数体内部，才可以使用。

arguments对象除了可以读取参数，还可以为参数赋值（严格模式不允许这种用法）。

```
var f = function(a, b) {
  arguments[0] = 3;
  arguments[1] = 2;
  return a + b;
}

f(1, 1)
// 5
```

可以通过arguments对象的length属性，判断函数调用时到底带几个参数。

```
function f() {
  return arguments.length;
}

f(1, 2, 3) // 3
f(1) // 1
f() // 0
```

虽然arguments很像数组，但它是一个对象。数组专有的方法（比如slice和forEach），不能在arguments对象上直接使用。

要让arguments对象使用数组方法，真正的解决方法是将arguments转为真正的数组。下面是两种常用的转换方法：slice方法和逐一填入新数组。

```
var args = Array.prototype.slice.call(arguments);

// or
var args = [];
for (var i = 0; i < arguments.length; i++) {
  args.push(arguments[i]);
}
```

arguments对象带有一个callee属性，返回它所对应的原函数。

```
var f = function(one) {
  console.log(arguments.callee === f);
}

f() // true
```

可以通过arguments.callee，达到调用函数自身的目的。这个属性在严格模式里面是禁用的，因此不建议使用。

其他知识点

闭包

JavaScript有两种作用域：全局作用域和函数作用域。函数内部可以直接读取全局变量。在函数外部无法读取函数内部声明的变量。

那么，如果需要得到函数内的局部变量，该怎么办呢？

JavaScript语言特有的"链式作用域"结构（chain scope），子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

```
function f1() {
  var n = 999;
  function f2() {
    console.log(n);
  }
  return f2;
}

var result = f1();
result(); // 999
```

函数 f2 就是闭包，即能够读取其他函数内部变量的函数。在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

闭包的最大用处有两个，一个是可以读取函数内部的变量，另一个就是让这些变量始终保持在内存中，即闭包可以使得它诞生环境一直存在。

```
function createIncrementor(start) {
  return function() {
    return start++;
  };
}

var inc = createIncrementor(5);
```

```
inc() // 5
inc() // 6
inc() // 7
```

闭包的另一个用处，是封装对象的私有属性和私有方法。

```
function Person(name) {
  var _age;
  function setAge(n) {
    _age = n;
  }
  function getAge() {
    return _age;
  }

  return {
    name: name,
    getAge: getAge,
    setAge: setAge
  };
}

var p1 = Person('张三');
p1.setAge(25);
p1.getAge() // 25
```

外层函数每次运行，都会生成一个新的闭包，而这个闭包又会保留外层函数的内部变量，所以内存消耗很大。因此不能滥用闭包，否则会造成网页的性能问题。

立即调用的函数表达式（IIFE）

一对圆括号()是一种运算符，跟在函数名之后，表示调用该函数。比如，print()就表示调用print函数。

不能在函数的定义之后加上圆括号，这会产生语法错误。因为function这个关键字即可以当作语句，也可以当作表达式。

JavaScript引擎规定，如果function关键字出现在行首，一律解释成语句。所以需要用到一些巧妙的办法（不让它出现在行首就行）。

```
function(){ /* code */ }();
// SyntaxError: Unexpected token (

(function(){ /* code */ }());
// 或者
(function(){ /* code */ })(); // 上面两种写法最后的分号都是必须的，因为是表达式

var i = function(){ return 10; }();
true && function(){ /* code */ }();
```

```
0, function(){ /* code */ }();

// 甚至这样也可以
!function(){ /* code */ }();
~function(){ /* code */ }();
-function(){ /* code */ }();
+function(){ /* code */ }();
```

通常情况下，只对匿名函数使用这种“立即执行的函数表达式”。它的目的有两个：一是不必为函数命名，避免了污染全局变量；二是IIFE内部形成了一个单独的作用域，可以封装一些外部无法读取的私有变量。

```
// 写法一
var tmp = newData;
processData(tmp);
storeData(tmp);

// 写法二
(function (){
  var tmp = newData;
  processData(tmp);
  storeData(tmp);
}());
```

写法二比写法一更好，因为完全避免了污染全局变量。

eval命令

eval命令的作用是，将字符串当作语句执行。eval没有自己的作用域，都在当前作用域内执行，因此可能会修改当前作用域的变量的值，造成安全问题。

为了防止这种风险，JavaScript规定，如果使用严格模式，eval内部声明的变量，不会影响到外部作用域。

```
eval('var a = 1;');
a // 1

var b = 1;
eval('b = 2');

b // 2

// 严格模式
(function f() {
  'use strict';
  eval('var foo = 123');
  console.log(foo);  // ReferenceError: foo is not defined
})()

// 但是它依然可以读写当前作用域的变量。
(function f() {
```

```
  'use strict';
  var foo = 1;
  eval('foo = 2');
  console.log(foo);  // 2
})()
```

1. 严格模式下，eval 内部还是改写了外部变量，可见安全风险依然存在。
2. 此外，eval 的命令字符串不会得到 JavaScript引擎 的优化，运行速度较慢。（会执行两次，一个解析成语句，一次执行）

eval 的调用分为 直接调用,间接调用。

1. 直接调用：eval 的作用域就是当前作用域.
2. 间接调用：eval 的作用域总是全局作用域.

只要不是直接调用，都属于间接调用。

```
var a = 1;

function f() {
  var a = 2;
  var e = eval;
  e('console.log(a)');
}

f() // 1

// 间接调用
eval.call(null, '...')
window.eval('...')
(1, eval)('...')
(eval, eval)('...')
```

JavaScript 程序可以采用事件驱动（event-driven）和非阻塞式（non-blocking）设计，在服务器端适合高并发环境，普通的硬件就可以承受很大的访问量。

JS的用途也越来越广泛(节选)：

1. 调用更多系统功能： 随着 HTML5 的出现，浏览器本身的功能越来越强，js 可以操作本地文件、操作图片、调用摄像头和麦克风等等，功能越来越丰富。
2. Node： Node 项目使得 JavaScript 可以用于开发服务器端的大型项目，网站的前后端都用 JavaScript 开发已经成为了现实。甚至可以为嵌入式平台开发应用程序（Raspberry Pi）。
3. 数据库操作： NoSQL 数据库这个概念，本身就是在 JSON 格式的基础上诞生的，大部分 NoSQL 数据库允许 JavaScript 直接操作。（JavaScript Object Notation，JavaScript 对象表示法，第一次知道 JSON 原来是这个意思。。。以前一直以为只是键值对，一种格式而已）
4. 跨移动平台： PhoneGap 项目就是将 JavaScript 和 HTML5 打包在一个容器之中，使得它能同时在 iOS 和安卓上运行。Facebook 公司的 React Native 项目则是将 JavaScript 写的组件，编译成原生

组件，从而使它们具备优秀的性能。Mozilla 基金会的手机操作系统Firefox OS，更是直接将 JavaScript 作为操作系统的平台语言。

5. 内嵌脚本语言： 越来越多的应用程序，将 JavaScript 作为内嵌的脚本语言，比如 Adobe 公司的著名 PDF 阅读器 Acrobat、Linux 桌面环境 GNOME 3。
6. 跨平台的桌面应用程序： Chromium OS、Windows 8 等操作系统直接支持 JavaScript 编写应用程序。Mozilla 的 Open Web Apps 项目、Google 的 Chrome App 项目、Github 的 Electron 项目、以及 TideSDK 项目，都可以用来编写运行于 Windows、Mac OS 和 Android 等多个桌面平台的程序，不依赖浏览器。