很多 JavaScript 程序都可能存在 NaN 方面的问题，所以我们应该尽量使用 Number.isNaN(..) 这样可靠的方法，无论是系统内置还是 polyfill。

如果你仍在代码中使用 isNaN(..)，那么你的程序迟早会出现 bug。

2. 无穷数

熟悉传统编译型语言（如 C）的开发人员可能都遇到过编译错误（compiler error）或者运行时错误（runtime exception），例如“除以 0”：

```
var a = 1 / 0;
```

然而在 JavaScript 中上例的结果为 Infinity（即 Number.POSITIVE_INfiNITY）。同样：

```
var a = 1 / 0;  // Infinity
var b = -1 / 0; // -Infinity
```

如果除法运算中的一个操作数为负数，则结果为 -Infinity（即 Number.NEGATIVE_INfiNITY）。

JavaScript 使用有限数字表示法（finite numeric representation，即之前介绍过的 IEEE 754 浮点数），所以和纯粹的数学运算不同，JavaScript 的运算结果有可能溢出，此时结果为 Infinity 或者 -Infinity。

例如：

```
var a = Number.MAX_VALUE;   // 1.7976931348623157e+308
a + a;                      // Infinity
a + Math.pow( 2, 970 );     // Infinity
a + Math.pow( 2, 969 );     // 1.7976931348623157e+308
```

规范规定，如果数学运算（如加法）的结果超出处理范围，则由 IEEE 754 规范中的“就近取整”（round-to-nearest）模式来决定最后的结果。例如，相对于 Infinity，Number.MAX_VALUE + Math.pow(2, 969) 与 Number.MAX_VALUE 更为接近，因此它被“向下取整”（round down）；而 Number.MAX_VALUE + Math.pow(2, 970) 与 Infinity 更为接近，所以它被“向上取整”（round up）。

这个问题想多了容易头疼，还是就此打住吧。

计算结果一旦溢出为无穷数（infinity）就无法再得到有穷数。换句话说，就是你可以从有穷走向无穷，但无法从无穷回到有穷。

有人也许会问：“那么无穷除以无穷会得到什么结果呢？”我们的第一反应可能会是“1”或者“无穷”，可惜都不是。因为从数学运算和 JavaScript 语言的角度来说，Infinity/Infinity 是一个未定义操作，结果为 NaN。

那么有穷正数除以 Infinity 呢？很简单，结果是 0。有穷负数除以 Infinity 呢？这里留个悬念，后面将作介绍。

3. 零值

这部分内容对于习惯数学思维的读者可能会带来困惑，JavaScript 有一个常规的 0（也叫作 +0）和一个 -0。在解释为什么会有 -0 之前，我们先来看看 JavaScript 是如何来处理它的。

-0 除了可以用作常量以外，也可以是某些数学运算的返回值。例如：

```
var a = 0 / -3; // -0
var b = 0 * -3; // -0
```

加法和减法运算不会得到负零（negative zero）。

负零在开发调试控制台中通常显示为 -0，但在一些老版本的浏览器中仍然会显示为 0。

根据规范，对负零进行字符串化会返回 "0"：

```
var a = 0 / -3;

// 至少在某些浏览器的控制台中显示是正确的
a                               // -0

// 但是规范定义的返回结果是这样!
a.toString();                   // "0"
a + "";                         // "0"
String( a );                    // "0"

// JSON也如此,很奇怪
JSON.stringify( a );    // "0"
```

有意思的是，如果反过来将其从字符串转换为数字，得到的结果是准确的：

```
+"-0";                 // -0
Number( "-0" );        // -0
JSON.parse( "-0" ); // -0
```

JSON.stringify(-0) 返回 "0"，而 JSON.parse("-0") 返回 -0。

负零转换为字符串的结果令人费解，它的比较操作也是如此：

```
var a = 0;
var b = 0 / -3;

a == b;     // true
```

```
-0 == 0;    // true

a === b;    // true
-0 === 0;   // true

0 > -0;     // false
a > b;      // false
```

要区分 -0 和 0，不能仅仅依赖开发调试窗口的显示结果，还需要做一些特殊处理：

```
function isNegZero(n) {
    n = Number( n );
    return (n === 0) && (1 / n === -Infinity);
}

isNegZero( -0 );        // true
isNegZero( 0 / -3 );    // true
isNegZero( 0 );         // false
```

抛开学术上的繁枝褥节不论，我们为什么需要负零呢？

有些应用程序中的数据需要以级数形式来表示（比如动画帧的移动速度），数字的符号位（sign）用来代表其他信息（比如移动的方向）。此时如果一个值为 0 的变量失去了它的符号位，它的方向信息就会丢失。所以保留 0 值的符号位可以防止这类情况发生。

2.4.4 特殊等式

如前所述，NaN 和 -0 在相等比较时的表现有些特别。由于 NaN 和自身不相等，所以必须使用 ES6 中的 Number.isNaN(..)（或者 polyfill）。而 -0 等于 0（对于 === 也是如此，参见第 4 章），因此我们必须使用 isNegZero(..) 这样的工具函数。

ES6 中新加入了一个工具方法 Object.is(..) 来判断两个值是否绝对相等，可以用来处理上述所有的特殊情况：

```
var a = 2 / "foo";
var b = -3 * 0;

Object.is( a, NaN );    // true
Object.is( b, -0 );     // true

Object.is( b, 0 );      // false
```

对于 ES6 之前的版本，Object.is(..) 有一个简单的 polyfill：

```
if (!Object.is) {
    Object.is = function(v1, v2) {
        // 判断是否是-0
        if (v1 === 0 && v2 === 0) {
            return 1 / v1 === 1 / v2;
```

```
        }
        // 判断是否是NaN
        if (v1 !== v1) {
            return v2 !== v2;
        }
        // 其他情况
        return v1 === v2;
    };
}
```

能使用 == 和 ===（参见第 4 章）时就尽量不要使用 Object.is(..)，因为前者效率更高、更为通用。Object.is(..) 主要用来处理那些特殊的相等比较。

2.5 值和引用

在许多编程语言中，赋值和参数传递可以通过值复制（value-copy）或者引用复制（reference-copy）来完成，这取决于我们使用什么语法。

例如，在 C++ 中如果要向函数传递一个数字并在函数中更改它的值，就可以这样来声明参数 int& myNum，即如果传递的变量是 x，myNum 就是指向 x 的引用。引用就像一种特殊的指针，是来指向变量的指针（别名）。如果参数不声明为引用的话，参数值总是通过值复制的方式传递，即便对复杂的对象值也是如此。

JavaScript 中没有指针，引用的工作机制也不尽相同。在 JavaScript 中变量不可能成为指向另一个变量的引用。

JavaScript 引用指向的是值。如果一个值有 10 个引用，这些引用指向的都是同一个值，它们相互之间没有引用 / 指向关系。

JavaScript 对值和引用的赋值 / 传递在语法上没有区别，完全根据值的类型来决定。

下面来看一个例子：

```
var a = 2;
var b = a; // b是a的值的一个副本
b++;
a; // 2
b; // 3

var c = [1,2,3];
var d = c; // d是[1,2,3]的一个引用
d.push( 4 );
c; // [1,2,3,4]
d; // [1,2,3,4]
```

简单值（即标量基本类型值，scalar primitive）总是通过值复制的方式来赋值 / 传递，包括 null、undefined、字符串、数字、布尔和 ES6 中的 symbol。

复合值（compound value）——对象（包括数组和封装对象，参见第 3 章）和函数，则总是通过引用复制的方式来赋值 / 传递。

上例中 2 是一个标量基本类型值，所以变量 a 持有该值的一个复本，b 持有它的另一个复本。b 更改时，a 的值保持不变。

c 和 d 则分别指向同一个复合值 [1,2,3] 的两个不同引用。请注意，c 和 d 仅仅是指向值 [1,2,3]，并非持有。所以它们更改的是同一个值（如调用 .push(4)），随后它们都指向更改后的新值 [1,2,3,4]。

由于引用指向的是值本身而非变量，所以一个引用无法更改另一个引用的指向。

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

// 然后
b = [4,5,6];
a; // [1,2,3]
b; // [4,5,6]
```

b=[4,5,6] 并不影响 a 指向值 [1,2,3]，除非 b 不是指向数组的引用，而是指向 a 的指针，但在 JavaScript 中不存在这种情况！

函数参数就经常让人产生这样的困惑：

```
function foo(x) {
    x.push( 4 );
    x; // [1,2,3,4]

    // 然后
    x = [4,5,6];
    x.push( 7 );
    x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );

a; // 是[1,2,3,4],不是[4,5,6,7]
```

我们向函数传递 a 的时候，实际是将引用 a 的一个复本赋值给 x，而 a 仍然指向 [1,2,3]。在函数中我们可以通过引用 x 来更改数组的值（push(4) 之后变为 [1,2,3,4]）。但 x = [4,5,6] 并不影响 a 的指向，所以 a 仍然指向 [1,2,3,4]。

我们不能通过引用 x 来更改引用 a 的指向，只能更改 a 和 x 共同指向的值。

如果要将 a 的值变为 [4,5,6,7]，必须更改 x 指向的数组，而不是为 x 赋值一个新的数组。

```
function foo(x) {
    x.push( 4 );
    x; // [1,2,3,4]

    // 然后
    x.length = 0; // 清空数组
    x.push( 4, 5, 6, 7 );
    x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );

a; // 是[4,5,6,7],不是[1,2,3,4]
```

从上例可以看出，x.length = 0 和 x.push(4,5,6,7) 并没有创建一个新的数组，而是更改了当前的数组。于是 a 指向的值变成了 [4,5,6,7]。

请记住：我们无法自行决定使用值复制还是引用复制，一切由值的类型来决定。

如果通过值复制的方式来传递复合值（如数组），就需要为其创建一个复本，这样传递的就不再是原始值。例如：

```
foo( a.slice() );
```

slice(..) 不带参数会返回当前数组的一个浅复本（shallow copy）。由于传递给函数的是指向该复本的引用，所以 foo(..) 中的操作不会影响 a 指向的数组。

相反，如果要将标量基本类型值传递到函数内并进行更改，就需要将该值封装到一个复合值（对象、数组等）中，然后通过引用复制的方式传递。

```
function foo(wrapper) {
    wrapper.a = 42;
}

var obj = {
    a: 2
};

foo( obj );

obj.a; // 42
```

这里 obj 是一个封装了标量基本类型值 a 的封装对象。obj 引用的一个复本作为参数 wrapper 被传递到 foo(..) 中。这样我们就可以通过 wrapper 来访问该对象并更改它的属性。函数执行结束后 obj.a 将变成 42。

这样看来，如果需要传递指向标量基本类型值（比如 2）的引用，就可以将其封装到对应的数字封装对象中（参见第 3 章）。

与预期不同的是，虽然传递的是指向数字对象的引用复本，但我们并不能通过它来更改其中的基本类型值：

```
function foo(x) {
    x = x + 1;
    x; // 3
}

var a = 2;
var b = new Number( a ); // Object(a)也一样

foo( b );
console.log( b ); // 是2,不是3
```

原因是标量基本类型值是*不可更改*的（字符串和布尔也是如此）。如果一个数字对象的标量基本类型值是 2，那么该值就不能更改，除非创建一个包含新值的数字对象。

`x = x + 1` 中，`x` 中的标量基本类型值 2 从数字对象中拆封（或者提取）出来后，`x` 就神不知鬼不觉地从引用变成了数字对象，它的值为 `2 + 1` 等于 3。然而函数外的 `b` 仍然指向原来那个值为 2 的数字对象。

我们还可以为数字对象添加属性（只要不更改其内部的基本类型值即可），通过它们间接地进行数据交换。

不过这种做法不太常见，大多数开发人员可能都觉得这不是一个好办法。

相对而言，前面用 `obj` 作为封装对象的办法可能更好一些。这并不是说数字等封装对象没有什么用，只是多数情况下我们应该优先考虑使用标量基本类型。

引用的功能很强大，但有时也难免成为阻碍。赋值 / 参数传递是通过引用还是值复制完全由值的类型来决定，所以使用哪种类型也间接决定了赋值 / 参数传递的方式。

2.6 小结

JavaScript 中的数组是通过数字索引的一组任意类型的值。字符串和数组类似，但是它们的行为特征不同，在将字符作为数组来处理时需要特别小心。JavaScript 中的数字包括“整数”和“浮点型”。

基本类型中定义了几个特殊的值。

`null` 类型只有一个值 `null`，`undefined` 类型也只有一个值 `undefined`。所有变量在赋值之前默认值都是 `undefined`。`void` 运算符返回 `undefined`。

数字类型有几个特殊值，包括 `NaN`（意指“not a number”，更确切地说是“invalid number”）、`+Infinity`、`-Infinity` 和 `-0`。

简单标量基本类型值（字符串和数字等）通过值复制来赋值 / 传递，而复合值（对象等）通过引用复制来赋值 / 传递。JavaScript 中的引用和其他语言中的引用 / 指针不同，它们不能指向别的变量 / 引用，只能指向值。

第 3 章

原生函数

第 1 章和第 2 章曾提到 JavaScript 的内建函数（built-in function），也叫原生函数（native function），如 `String` 和 `Number`。本章将详细介绍它们。

常用的原生函数有：

- `String()`
- `Number()`
- `Boolean()`
- `Array()`
- `Object()`
- `Function()`
- `RegExp()`
- `Date()`
- `Error()`
- `Symbol()`——ES6 中新加入的！

实际上，它们就是内建函数。

熟悉 Java 语言的人会发现，JavaScript 中的 `String()` 和 Java 中的字符串构造函数 `String(..)` 非常相似，可以这样来用：

```
var s = new String( "Hello World!" );

console.log( s.toString() ); // "Hello World!"
```

原生函数可以被当作构造函数来使用，但其构造出来的对象可能会和我们设想的有所出入：

```
var a = new String( "abc" );

typeof a;                               // 是"object",不是"String"

a instanceof String;                    // true

Object.prototype.toString.call( a ); // "[object String]"
```

通过构造函数（如 `new String("abc")`）创建出来的是封装了基本类型值（如 `"abc"`）的封装对象。

请注意：`typeof` 在这里返回的是对象类型的子类型。

可以这样来查看封装对象：

```
console.log( a );
```

由于不同浏览器在开发控制台中显示对象的方式不同（对象序列化，object serialization），所以上面的输出结果也不尽相同。

在本书写作期间，Chrome 的最新版本是这样显示的：`String {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"}`，而老版本这样显示：`String {0: "a", 1: "b", 2: "c"}`。最新版本的 Firefox 这样显示：`String ["a","b","c"]`；老版本这样显示：`"abc"`，并且可以点击打开对象查看器。这些输出结果随着浏览器的演进不断变化，也带给人们不同的体验。

再次强调，`new String("abc")` 创建的是字符串 `"abc"` 的封装对象，而非基本类型值 `"abc"`。

3.1 内部属性 [[Class]]

所有 `typeof` 返回值为 `"object"` 的对象（如数组）都包含一个内部属性 `[[Class]]`（我们可以把它看作一个内部的分类，而非传统的面向对象意义上的类）。这个属性无法直接访问，一般通过 `Object.prototype.toString(..)` 来查看。例如：

```
Object.prototype.toString.call( [1,2,3] );
// "[object Array]"

Object.prototype.toString.call( /regex-literal/i );
// "[object RegExp]"
```

上例中，数组的内部 `[[Class]]` 属性值是 `"Array"`，正则表达式的值是 `"RegExp"`。多数情况

下，对象的内部 [[Class]] 属性和创建该对象的内建原生构造函数相对应（如下），但并非总是如此。

那么基本类型值呢？下面先来看看 null 和 undefined：

```
Object.prototype.toString.call( null );
// "[object Null]"

Object.prototype.toString.call( undefined );
// "[object Undefined]"
```

虽然 Null() 和 Undefined() 这样的原生构造函数并不存在，但是内部 [[Class]] 属性值仍然是 "Null" 和 "Undefined"。

其他基本类型值（如字符串、数字和布尔）的情况有所不同，通常称为“包装”（boxing，参见 3.2 节）：

```
Object.prototype.toString.call( "abc" );
// "[object String]"

Object.prototype.toString.call( 42 );
// "[object Number]"

Object.prototype.toString.call( true );
// "[object Boolean]"
```

上例中基本类型值被各自的封装对象自动包装，所以它们的内部 [[Class]] 属性值分别为 "String"、"Number" 和 "Boolean"。

从 ES5 到 ES6，toString() 和 [[Class]] 的行为发生了一些变化，详情见本系列的《你不知道的 JavaScript（下卷）》的“ES6 & Beyond”部分。

3.2 封装对象包装

封装对象（object wrapper）扮演着十分重要的角色。由于基本类型值没有 .length 和 .toString() 这样的属性和方法，需要通过封装对象才能访问，此时 JavaScript 会自动为基本类型值包装（box 或者 wrap）一个封装对象：

```
var a = "abc";

a.length; // 3
a.toUpperCase(); // "ABC"
```

如果需要经常用到这些字符串属性和方法，比如在 for 循环中使用 i < a.length，那么从一开始就创建一个封装对象也许更为方便，这样 JavaScript 引擎就不用每次都自动创建了。

但实际证明这并不是一个好办法，因为浏览器已经为 .length 这样的常见情况做了性能优化，直接使用封装对象来“提前优化”代码反而会降低执行效率。

一般情况下，我们不需要直接使用封装对象。最好的办法是让 JavaScript 引擎自己决定什么时候应该使用封装对象。换句话说，就是应该优先考虑使用 "abc" 和 42 这样的基本类型值，而非 new String("abc") 和 new Number(42)。

封装对象释疑

使用封装对象时有些地方需要特别注意。

比如 Boolean：

```
var a = new Boolean( false );

if (!a) {
    console.log( "Oops" ); // 执行不到这里
}
```

我们为 false 创建了一个封装对象，然而该对象是真值（“truthy”，即总是返回 true，参见第 4 章），所以这里使用封装对象得到的结果和使用 false 截然相反。

如果想要自行封装基本类型值，可以使用 Object(..) 函数（不带 new 关键字）：

```
var a = "abc";
var b = new String( a );
var c = Object( a );

typeof a; // "string"
typeof b; // "object"
typeof c; // "object"

b instanceof String; // true
c instanceof String; // true

Object.prototype.toString.call( b ); // "[object String]"
Object.prototype.toString.call( c ); // "[object String]"
```

再次强调，一般不推荐直接使用封装对象（如上例中的 b 和 c），但它们偶尔也会派上用场。

3.3 拆封

如果想要得到封装对象中的基本类型值，可以使用 valueOf() 函数：

```
var a = new String( "abc" );
var b = new Number( 42 );
var c = new Boolean( true );

a.valueOf(); // "abc"
b.valueOf(); // 42
c.valueOf(); // true
```

在需要用到封装对象中的基本类型值的地方会发生隐式拆封。具体过程（即强制类型转换）将在第 4 章详细介绍。

```
var a = new String( "abc" );
var b = a + ""; // b的值为"abc"

typeof a;       // "object"
typeof b;       // "string"
```

3.4 原生函数作为构造函数

关于数组（array）、对象（object）、函数（function）和正则表达式，我们通常喜欢以常量的形式来创建它们。实际上，使用常量和使用构造函数的效果是一样的（创建的值都是通过封装对象来包装）。

如前所述，应该尽量避免使用构造函数，除非十分必要，因为它们经常会产生意想不到的结果。

3.4.1 Array(..)

```
var a = new Array( 1, 2, 3 );
a; // [1, 2, 3]

var b = [1, 2, 3];
b; // [1, 2, 3]
```

构造函数 Array(..) 不要求必须带 new 关键字。不带时，它会被自动补上。因此 Array(1,2,3) 和 new Array(1,2,3) 的效果是一样的。

Array 构造函数只带一个数字参数的时候，该参数会被作为数组的预设长度（length），而非只充当数组中的一个元素。

这实非明智之举：一是容易忘记，二是容易出错。

更为关键的是，数组并没有预设长度这个概念。这样创建出来的只是一个空数组，只不过

它的 length 属性被设置成了指定的值。

如若一个数组没有任何单元，但它的 length 属性中却显示有单元数量，这样奇特的数据结构会导致一些怪异的行为。而这一切都归咎于已被废止的旧特性（类似 arguments 这样的类数组）。

我们将包含至少一个“空单元”的数组称为“稀疏数组”。

对此，不同浏览器的开发控制台显示的结果也不尽相同，这让问题变得更加复杂。

例如：

```
var a = new Array( 3 );

a.length; // 3
a;
```

a 在 Chrome 中显示为 [undefined x 3]（目前为止），这意味着它有三个值为 undefined 的单元，但实际上单元并不存在（“空单元”这个叫法也同样不准确）。

从下面代码的结果可以看出它们的差别：

```
var a = new Array( 3 );
var b = [ undefined, undefined, undefined ];
var c = [];
c.length = 3;

a;
b;
c;
```

我们可以创建包含空单元的数组，如上例中的 c。只要将 length 属性设置为超过实际单元数的值，就能隐式地制造出空单元。另外还可以通过 delete b[1] 在数组 b 中制造出一个空单元。

b 在当前版本的 Chrome 中显示为 [undefined, undefined, undefined]，而 a 和 c 则显示为 [undefined x 3]。是不是感到很困惑？

更令人费解的是在当前版本的 Firefox 中 a 和 c 显示为 [, , ,]。仔细看来，这其中有三个逗号，代表四个空单元，而不是三个。

Firefox 在输出结果后面多添了一个 ,，原因是从 ES5 规范开始就允许在列表（数组值、属

性列表等）末尾多加一个逗号（在实际处理中会被忽略不计）。所以如果你在代码或者调试控制台中输入 [, , ,]，实际得到的是 [, ,]（包含三个空单元的数组）。这样做虽然在控制台中看似令人费解，实则是为了让复制粘贴结果更为准确。

读到这里你或许已是一头雾水，但没关系，打起精神，你不是一个人在战斗！

针对这种情况，Firefox 将 [, , ,] 改为显示 Array [<3 empty slots>]，这无疑是个很大的提升。

更糟糕的是，上例中 a 和 b 的行为有时相同，有时又大相径庭：

```
a.join( "-" ); // "--"
b.join( "-" ); // "--"

a.map(function(v,i){ return i; }); // [ undefined x 3 ]
b.map(function(v,i){ return i; }); // [ 0, 1, 2 ]
```

a.map(..) 之所以执行失败，是因为数组中并不存在任何单元，所以 map(..) 无从遍历。而 join(..) 却不一样，它的具体实现可参考下面的代码：

```
function fakeJoin(arr,connector) {
    var str = "";
    for (var i = 0; i < arr.length; i++) {
        if (i > 0) {
            str += connector;
        }
        if (arr[i] !== undefined) {
            str += arr[i];
        }
    }
    return str;
}

var a = new Array( 3 );
fakeJoin( a, "-" ); // "--"
```

从中可以看出，join(..) 首先假定数组不为空，然后通过 length 属性值来遍历其中的元素。而 map(..) 并不做这样的假定，因此结果也往往在预期之外，并可能导致失败。

我们可以通过下述方式来创建包含 undefined 单元（而非“空单元”）的数组：

```
var a = Array.apply( null, { length: 3 } );
a; // [ undefined, undefined, undefined ]
```

上述代码或许会引起困惑，下面大致解释一下。