```
function doSomethingCool(FeatureXYZ) {
    var helper = FeatureXYZ ||
        function() { /*.. default feature ..*/ };
    var val = helper();
    // ..
}
```

上述种种选择和方法各有利弊。好在 typeof 的安全防范机制为我们提供了更多选择。

1.4 小结

JavaScript 有七种内置类型：null、undefined、boolean、number、string、object 和 symbol，可以使用 typeof 运算符来查看。

变量没有类型，但它们持有的值有类型。类型定义了值的行为特征。

很多开发人员将 undefined 和 undeclared 混为一谈，但在 JavaScript 中它们是两码事。undefined 是值的一种。undeclared 则表示变量还没有被声明过。

遗憾的是，JavaScript 却将它们混为一谈，在我们试图访问 "undeclared" 变量时这样报错：ReferenceError: a is not defined，并且 typeof 对 undefined 和 undeclared 变量都返回 "undefined"。

然而，通过 typeof 的安全防范机制（阻止报错）来检查 undeclared 变量，有时是个不错的办法。

第 2 章

值

数组（array）、字符串（string）和数字（number）是一个程序最基本的组成部分，但在 JavaScript 中，它们可谓让人喜忧掺半。

本章将介绍 JavaScript 中的几个内置值类型，让读者深入了解和合理运用它们。

2.1 数组

和其他强类型语言不同，在 JavaScript 中，数组可以容纳任何类型的值，可以是字符串、数字、对象（object），甚至是其他数组（多维数组就是通过这种方式来实现的）：

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

a.length;       // 3
a[0] === 1;     // true
a[2][0] === 3;  // true
```

对数组声明后即可向其中加入值，不需要预先设定大小（参见 3.4.1 节）：

```
var a = [ ];

a.length;   // 0

a[0] = 1;
a[1] = "2";
a[2] = [ 3 ];

a.length;   // 3
```

使用 delete 运算符可以将单元从数组中删除，但是请注意，单元删除后，数组的 length 属性并不会发生变化。第 5 章将详细介绍 delete 运算符。

在创建“稀疏”数组（sparse array，即含有空白或空缺单元的数组）时要特别注意：

```
var a = [ ];

a[0] = 1;
// 此处没有设置a[1]单元
a[2] = [ 3 ];

a[1];       // undefined

a.length;   // 3
```

上面的代码可以正常运行，但其中的“空白单元”（empty slot）可能会导致出人意料的结果。a[1] 的值为 undefined，但这与将其显式赋值为 undefined（a[1] = undefined）还是有所区别。详情请参见 3.4.1 节。

数组通过数字进行索引，但有趣的是它们也是对象，所以也可以包含字符串键值和属性（但这些并不计算在数组长度内）：

```
var a = [ ];

a[0] = 1;
a["foobar"] = 2;

a.length;       // 1
a["foobar"];    // 2
a.foobar;       // 2
```

这里有个问题需要特别注意，如果字符串键值能够被强制类型转换为十进制数字的话，它就会被当作数字索引来处理。

```
var a = [ ];

a["13"] = 42;

a.length; // 14
```

在数组中加入字符串键值 / 属性并不是一个好主意。建议使用对象来存放键值 / 属性值，用数组来存放数字索引值。

类数组

有时需要将类数组（一组通过数字索引的值）转换为真正的数组，这一般通过数组工具函

数（如 indexOf(..)、concat(..)、forEach(..) 等）来实现。

例如，一些 DOM 查询操作会返回 DOM 元素列表，它们并非真正意义上的数组，但十分类似。另一个例子是通过 arguments 对象（类数组）将函数的参数当作列表来访问（从 ES6 开始已废止）。

工具函数 slice(..) 经常被用于这类转换：

```
function foo() {
    var arr = Array.prototype.slice.call( arguments );
    arr.push( "bam" );
    console.log( arr );
}

foo( "bar", "baz" ); // ["bar","baz","bam"]
```

如上所示，slice() 返回参数列表（上例中是一个类数组）的一个数组复本。

用 ES6 中的内置工具函数 Array.from(..) 也能实现同样的功能：

```
...
var arr = Array.from( arguments );
...
```

Array.from(..) 有一些非常强大的功能，将在本系列的《你不知道的 JavaScript（下卷）》的“ES6 & Beyond”部分详细介绍。

2.2 字符串

字符串经常被当成字符数组。字符串的内部实现究竟有没有使用数组并不好说，但 JavaScript 中的字符串和字符数组并不是一回事，最多只是看上去相似而已。

例如下面两个值：

```
var a = "foo";
var b = ["f","o","o"];
```

字符串和数组的确很相似，它们都是类数组，都有 length 属性以及 indexOf(..)（从 ES5 开始数组支持此方法）和 concat(..) 方法：

```
[source,js]

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

```
a.indexOf( "o" );                   // 1
b.indexOf( "o" );                   // 1

var c = a.concat( "bar" );          // "foobar"
var d = b.concat( ["b","a","r"] );  // ["f","o","o","b","a","r"]

a === c;                            // false
b === d;                            // false

a;                                  // "foo"
b;                                  // ["f","o","o"]
```

但这并不意味着它们都是“字符数组”，比如：

```
a[1] = "O";
b[1] = "O";

a; // "foo"
b; // ["f","O","o"]
```

JavaScript 中字符串是不可变的，而数组是可变的。并且 `a[1]` 在 JavaScript 中并非总是合法语法，在老版本的 IE 中就不被允许（现在可以了）。正确的方法应该是 `a.charAt(1)`。

字符串不可变是指字符串的成员函数不会改变其原始值，而是创建并返回一个新的字符串。而数组的成员函数都是在其原始值上进行操作。

```
c = a.toUpperCase();
a === c;    // false
a;          // "foo"
c;          // "FOO"

b.push( "!" );
b;          // ["f","O","o","!"]
```

许多数组函数用来处理字符串很方便。虽然字符串没有这些函数，但可以通过“借用”数组的非变更方法来处理字符串：

```
a.join;         // undefined
a.map;          // undefined

var c = Array.prototype.join.call( a, "-" );
var d = Array.prototype.map.call( a, function(v){
    return v.toUpperCase() + ".";
} ).join( "" );

c;              // "f-o-o"
d;              // "F.O.O."
```

另一个不同点在于字符串反转（JavaScript 面试常见问题）。数组有一个字符串没有的可变

更成员函数 reverse()：

```
a.reverse;      // undefined

b.reverse();    // ["!","o","O","f"]
b;              // ["f","O","o","!"]
```

可惜我们无法“借用”数组的可变更成员函数，因为字符串是不可变的：

```
Array.prototype.reverse.call( a );
// 返回值仍然是字符串"foo"的一个封装对象(参见第3章):(
```

一个变通（破解）的办法是先将字符串转换为数组，待处理完后再将结果转换回字符串：

```
var c = a
    // 将a的值转换为字符数组
    .split( "" )
    // 将数组中的字符进行倒转
    .reverse()
    // 将数组中的字符拼接回字符串
    .join( "" );

c; // "oof"
```

这种方法的确简单粗暴，但对简单的字符串却完全适用。

请注意！上述方法对于包含复杂字符（Unicode，如星号、多字节字符等）的字符串并不适用。这时则需要功能更加完备、能够处理 Unicode 的工具库。可以参考 Mathias Bynen 的 Esrever（https://github.com/mathiasbynents/esrever）。

如果需要经常以字符数组的方式来处理字符串的话，倒不如直接使用数组。这样就不用在字符串和数组之间来回折腾。可以在需要时使用 join("") 将字符数组转换为字符串。

2.3 数字

JavaScript 只有一种数值类型：number（数字），包括“整数”和带小数的十进制数。此处“整数”之所以加引号是因为和其他语言不同，JavaScript 没有真正意义上的整数，这也是它一直以来为人诟病的地方。这种情况在将来或许会有所改观，但目前只有数字类型。

JavaScript 中的“整数”就是没有小数的十进制数。所以 42.0 即等同于“整数”42。

与大部分现代编程语言（包括几乎所有的脚本语言）一样，JavaScript 中的数字类型是基于 IEEE 754 标准来实现的，该标准通常也被称为“浮点数”。JavaScript 使用的是“双精度”格式（即 64 位二进制）。

网上的很多文章详细介绍了二进制浮点数在内存中的存储方式，以及不同方式各自的考量。要想正确使用 JavaScript 中的数字类型，并非一定要了解数位（bit）在内存中的存储方式，所以本书对此不多作介绍，有兴趣的读者可以参见 IEEE 754 的相关细节。

2.3.1 数字的语法

JavaScript 中的数字常量一般用十进制表示。例如：

```
var a = 42;
var b = 42.3;
```

数字前面的 0 可以省略：

```
var a = 0.42;
var b = .42;
```

小数点后小数部分最后面的 `0` 也可以省略：

```
var a = 42.0;
var b = 42.;
```

`42.` 这种写法没问题，只是不常见，但从代码的可读性考虑，不建议这样写。

默认情况下大部分数字都以十进制显示，小数部分最后面的 `0` 被省略，如：

```
var a = 42.300;
var b = 42.0;

a; // 42.3
b; // 42
```

特别大和特别小的数字默认用指数格式显示，与 `toExponential()` 函数的输出结果相同。例如：

```
var a = 5E10;
a;                  // 50000000000
a.toExponential();  // "5e+10"

var b = a * a;
b;                  // 2.5e+21

var c = 1 / a;
c;                  // 2e-11
```

由于数字值可以使用 `Number` 对象进行封装（参见第 3 章），因此数字值可以调用 `Number.`

prototype 中的方法（参见第 3 章）。例如，tofixed(..) 方法可指定小数部分的显示位数：

```
var a = 42.59;

a.toFixed( 0 ); // "43"
a.toFixed( 1 ); // "42.6"
a.toFixed( 2 ); // "42.59"
a.toFixed( 3 ); // "42.590"
a.toFixed( 4 ); // "42.5900"
```

请注意，上例中的输出结果实际上是给定数字的字符串形式，如果指定的小数部分的显示位数多于实际位数就用 0 补齐。

toPrecision(..) 方法用来指定有效数位的显示位数：

```
var a = 42.59;

a.toPrecision( 1 ); // "4e+1"
a.toPrecision( 2 ); // "43"
a.toPrecision( 3 ); // "42.6"
a.toPrecision( 4 ); // "42.59"
a.toPrecision( 5 ); // "42.590"
a.toPrecision( 6 ); // "42.5900"
```

上面的方法不仅适用于数字变量，也适用于数字常量。不过对于 . 运算符需要给予特别注意，因为它是一个有效的数字字符，会被优先识别为数字常量的一部分，然后才是对象属性访问运算符。

```
// 无效语法：
42.toFixed( 3 );    // SyntaxError

// 下面的语法都有效：
(42).toFixed( 3 );  // "42.000"
0.42.toFixed( 3 );  // "0.420"
42..toFixed( 3 );   // "42.000"
```

42.tofixed(3) 是无效语法，因为 . 被视为常量 42. 的一部分（如前所述），所以没有 . 属性访问运算符来调用 tofixed 方法。

42..tofixed(3) 则没有问题，因为第一个 . 被视为 number 的一部分，第二个 . 是属性访问运算符。只是这样看着奇怪，实际情况中也很少见。在基本类型值上直接调用的方法并不多见，不过这并不代表不好或不对。

一些工具库扩展了 Number.prototype 的内置方法（参见第 3 章）以提供更多的数值操作，比如用 10..makeItRain() 方法来实现十秒钟金钱雨动画等效果。

下面的语法也是有效的（请注意其中的空格）：

```
42 .toFixed(3); // "42.000"
```

然而对数字常量而言，这样的语法很容易引起误会，不建议使用。

我们还可以用指数形式来表示较大的数字，如：

```
var onethousand = 1E3;                        // 即 1 * 10^3
var onemillononehundredthousand = 1.1E6;   // 即 1.1 * 10^6
```

数字常量还可以用其他格式来表示，如二进制、八进制和十六进制。

当前的 JavaScript 版本都支持这些格式：

```
0xf3; // 243的十六进制
0Xf3; // 同上

0363; // 243的八进制
```

从 ES6 开始，严格模式（strict mode）不再支持 `0363` 八进制格式（新格式如下）。`0363` 格式在非严格模式（non-strict mode）中仍然受支持，但是考虑到将来的兼容性，最好不要再使用（我们现在使用的应该是严格模式）。

ES6 支持以下新格式：

```
0o363;      // 243的八进制
0O363;      // 同上

0b11110011; // 243的二进制
0B11110011; // 同上
```

考虑到代码的易读性，不推荐使用 `0O363` 格式，因为 `0` 和大写字母 `O` 在一起容易混淆。建议尽量使用小写的 `0x`、`0b` 和 `0o`。

2.3.2 较小的数值

二进制浮点数最大的问题（不仅 JavaScript，所有遵循 IEEE 754 规范的语言都是如此），是会出现如下情况：

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

从数学角度来说，上面的条件判断应该为 `true`，可结果为什么是 `false` 呢？

简单来说，二进制浮点数中的 `0.1` 和 `0.2` 并不是十分精确，它们相加的结果并非刚好等于 `0.3`，而是一个比较接近的数字 `0.30000000000000004`，所以条件判断结果为 `false`。

有人认为，JavaScript 应该采用一种可以精确呈现数字的实现方式。一直以来出现过很多替代方案，只是都没能成为标准，以后大概也不会。这个问题看似简单，实则不然，否则早就解决了。

问题是，如果一些数字无法做到完全精确，是否意味着数字类型毫无用处呢？答案当然是否定的。

在处理带有小数的数字时需要特别注意。很多（也许是绝大多数）程序只需要处理整数，最大不超过百万或者万亿，此时使用 JavaScript 的数字类型是绝对安全的。

那么应该怎样来判断 `0.1 + 0.2` 和 `0.3` 是否相等呢？

最常见的方法是设置一个误差范围值，通常称为“机器精度”（machine epsilon），对 JavaScript 的数字来说，这个值通常是 `2^-52 (2.220446049250313e-16)`。

从 ES6 开始，该值定义在 `Number.EPSILON` 中，我们可以直接拿来用，也可以为 ES6 之前的版本写 polyfill：

```
if (!Number.EPSILON) {
    Number.EPSILON = Math.pow(2,-52);
}
```

可以使用 `Number.EPSILON` 来比较两个数字是否相等（在指定的误差范围内）：

```
function numbersCloseEnoughToEqual(n1,n2) {
    return Math.abs( n1 - n2 ) < Number.EPSILON;
}

var a = 0.1 + 0.2;
var b = 0.3;

numbersCloseEnoughToEqual( a, b );                  // true
numbersCloseEnoughToEqual( 0.0000001, 0.0000002 );  // false
```

能够呈现的最大浮点数大约是 `1.798e+308`（这是一个相当大的数字），它定义在 `Number.MAX_VALUE` 中。最小浮点数定义在 `Number.MIN_VALUE` 中，大约是 `5e-324`，它不是负数，但无限接近于 0！

2.3.3 整数的安全范围

数字的呈现方式决定了“整数”的安全值范围远远小于 `Number.MAX_VALUE`。

能够被“安全”呈现的最大整数是 `2^53 - 1`，即 `9007199254740991`，在 ES6 中被定义为 `Number.MAX_SAFE_INTEGER`。最小整数是 `-9007199254740991`，在 ES6 中被定义为 `Number.MIN_SAFE_INTEGER`。

有时 JavaScript 程序需要处理一些比较大的数字，如数据库中的 64 位 ID 等。由于 JavaScript 的数字类型无法精确呈现 64 位数值，所以必须将它们保存（转换）为字符串。

好在大数值操作并不常见（它们的比较操作可以通过字符串来实现）。如果确实需要对大数值进行数学运算，目前还是需要借助相关的工具库。将来 JavaScript 也许会加入对大数值的支持。

2.3.4 整数检测

要检测一个值是否是整数，可以使用 ES6 中的 `Number.isInteger(..)` 方法：

```
Number.isInteger( 42 );     // true
Number.isInteger( 42.000 ); // true
Number.isInteger( 42.3 );   // false
```

也可以为 ES6 之前的版本 polyfill `Number.isInteger(..)` 方法：

```
if (!Number.isInteger) {
    Number.isInteger = function(num) {
        return typeof num == "number" && num % 1 == 0;
    };
}
```

要检测一个值是否是安全的整数，可以使用 ES6 中的 `Number.isSafeInteger(..)` 方法：

```
Number.isSafeInteger( Number.MAX_SAFE_INTEGER );    // true
Number.isSafeInteger( Math.pow( 2, 53 ) );          // false
Number.isSafeInteger( Math.pow( 2, 53 ) - 1 );      // true
```

可以为 ES6 之前的版本 polyfill `Number.isSafeInteger(..)` 方法：

```
if (!Number.isSafeInteger) {
    Number.isSafeInteger = function(num) {
        return Number.isInteger( num ) &&
            Math.abs( num ) <= Number.MAX_SAFE_INTEGER;
    };
}
```

2.3.5 32 位有符号整数

虽然整数最大能够达到 53 位，但是有些数字操作（如数位操作）只适用于 32 位数字，所以这些操作中数字的安全范围就要小很多，变成从 `Math.pow(-2,31)`（`-2147483648`，约−21 亿）到 `Math.pow(2,31) - 1`（`2147483647`，约 21 亿）。

`a | 0` 可以将变量 `a` 中的数值转换为 32 位有符号整数，因为数位运算符 | 只适用于 32 位整数（它只关心 32 位以内的值，其他的数位将被忽略）。因此与 `0` 进行操作即可截取 `a` 中的 32 位数位。

某些特殊的值并不是 32 位安全范围的，如 NaN 和 Infinity（下节将作相关介绍），此时会对它们执行虚拟操作（abstract operation）ToInt32（参见第 4 章），以便转换为符合数位运算符要求的 +0 值。

2.4 特殊数值

JavaScript 数据类型中有几个特殊的值需要开发人员特别注意和小心使用。

2.4.1 不是值的值

undefined 类型只有一个值，即 undefined。null 类型也只有一个值，即 null。它们的名称既是类型也是值。

undefined 和 null 常被用来表示“空的”值或“不是值”的值。二者之间有一些细微的差别。例如：

- null 指空值（empty value）
- undefined 指没有值（missing value）

或者：

- undefined 指从未赋值
- null 指曾赋过值，但是目前没有值

null 是一个特殊关键字，不是标识符，我们不能将其当作变量来使用和赋值。然而 undefined 却是一个标识符，可以被当作变量来使用和赋值。

2.4.2 undefined

在非严格模式下，我们可以为全局标识符 undefined 赋值（这样的设计实在是欠考虑！）：

```
function foo() {
    undefined = 2; // 非常糟糕的做法!
}

foo();

function foo() {
    "use strict";
    undefined = 2; // TypeError!
}

foo();
```

在非严格和严格两种模式下，我们可以声明一个名为 undefined 的局部变量。再次强调最好不要这样做！

```
function foo() {
    "use strict";
    var undefined = 2;
    console.log( undefined ); // 2
}

foo();
```

永远不要重新定义 undefined。

void 运算符

undefined 是一个内置标识符（除非被重新定义，见前面的介绍），它的值为 undefined，通过 void 运算符即可得到该值。

表达式 void ___ 没有返回值，因此返回结果是 undefined。void 并不改变表达式的结果，只是让表达式不返回值：

```
var a = 42;

console.log( void a, a ); // undefined 42
```

按惯例我们用 void 0 来获得 undefined（这主要源自 C 语言，当然使用 void true 或其他 void 表达式也是可以的）。void 0、void 1 和 undefined 之间并没有实质上的区别。

void 运算符在其他地方也能派上用场，比如不让表达式返回任何结果（即使其有副作用）。

例如：

```
function doSomething() {
    // 注：APP.ready 由程序自己定义
    if (!APP.ready) {
        // 稍后再试
        return void setTimeout( doSomething,100 );
    }

    var result;

    // 其他
    return result;
}

// 现在可以了吗？
if (doSomething()) {
    // 立即执行下一个任务
}
```

这里 setTimeout(..) 函数返回一个数值（计时器间隔的唯一标识符，用来取消计时），但

是为了确保 if 语句不产生误报（false positive），我们要 void 掉它。

很多开发人员喜欢分开操作，效果都一样，只是没有使用 void 运算符：

```
if (!APP.ready) {
    // 稍后再试
    setTimeout( doSomething,100 );
    return;
}
```

总之，如果要将代码中的值（如表达式的返回值）设为 undefined，就可以使用 void。这种做法并不多见，但在某些情况下却很有用。

2.4.3 特殊的数字

数字类型中有几个特殊的值，下面将详细介绍。

1. 不是数字的数字

如果数学运算的操作数不是数字类型（或者无法解析为常规的十进制或十六进制数字），就无法返回一个有效的数字，这种情况下返回值为 NaN。

NaN 意指“不是一个数字”（not a number），这个名字容易引起误会，后面将会提到。将它理解为“无效数值”“失败数值”或者“坏数值”可能更准确些。

例如：

```
var a = 2 / "foo";      // NaN

typeof a === "number";  // true
```

换句话说，“不是数字的数字”仍然是数字类型。这种说法可能有点绕。

NaN 是一个“警戒值”（sentinel value，有特殊用途的常规值），用于指出数字类型中的错误情况，即“执行数学运算没有成功，这是失败后返回的结果”。

有人也许认为如果要检查变量的值是否为 NaN，可以直接和 NaN 进行比较，就像比较 null 和 undefined 那样。实则不然。

```
var a = 2 / "foo";

a == NaN;   // false
a === NaN;  // false
```

NaN 是一个特殊值，它和自身不相等，是唯一一个非自反（自反，reflexive，即 x === x 不成立）的值。而 NaN != NaN 为 true，很奇怪吧？

既然我们无法对 NaN 进行比较（结果永远为 false），那应该怎样来判断它呢？

```
var a = 2 / "foo";

isNaN( a ); // true
```

很简单，可以使用内建的全局工具函数 isNaN(..) 来判断一个值是否是 NaN。

然而操作起来并非这么容易。isNaN(..) 有一个严重的缺陷，它的检查方式过于死板，就是“检查参数是否不是 NaN，也不是数字”。但是这样做的结果并不太准确：

```
var a = 2 / "foo";
var b = "foo";

a; // NaN
b; "foo"

window.isNaN( a ); // true
window.isNaN( b ); // true——晕!
```

很明显 "foo" 不是一个数字，但是它也不是 NaN。这个 bug 自 JavaScript 问世以来就一直存在，至今已超过 19 年。

从 ES6 开始我们可以使用工具函数 Number.isNaN(..)。ES6 之前的浏览器的 polyfill 如下：

```
if (!Number.isNaN) {
    Number.isNaN = function(n) {
        return (
            typeof n === "number" &&
            window.isNaN( n )
        );
    };
}

var a = 2 / "foo";
var b = "foo";

Number.isNaN( a ); // true
Number.isNaN( b ); // false——好!
```

实际上还有一个更简单的方法，即利用 NaN 不等于自身这个特点。NaN 是 JavaScript 中唯一一个不等于自身的值。

于是我们可以这样：

```
if (!Number.isNaN) {
    Number.isNaN = function(n) {
        return n !== n;
    };
}
```