你不知道的 JavaScript

第 1 章 类型

1.1 类型

1.2 内置类型

七种内置类型：

- 空值（null）
- 未定义（undefined）
- 布尔值（Boolean）
- 数字（number）
- 字符串（string）
- 对象（object）
- 符号（symbol，ES6 中新增）

注：除对象之外，其它统称“基本类型”

1.3 值和类型

JavaScript 中的变量是没有类型的，只有值才有。变量可以随时持有任何类型的值。

1.3.1 undefined 和 undeclared

变量在未持有值的时候为 undefined。此时 typeof 返回“undefined”

还没有在作用域中声明过的变量为 undeclared.

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。

第二章 值

2.1 数组

数组可以容纳任何类型的值，可以是字符串、数字、对象、甚至是其它数组。

对数组声明后即可向其中加入值，不需要预先设定大小

使用 delete 运算符可以将单元从数组中删除，但单元删除后，数组的 length 属性并不会发生变化。

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，但这与将其显示赋值为 undefine 还是有区别的。

数组通过数字进行索引，他们也是对象，所以也可以包含字符串键值和属性。

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。

2.2 字符串

字符串经常被当成字符串数组。但字符串与数组并不是一回事。他们是类数组，都有 length 属性以及 indexOf 和 concat（）方法。

字符串是不可变的，数组是可变的

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。

2.3.1 数字的语法

数字常量一般用十进制表示。

数字前面的 0 可以省略。

小数点后小数部分最后面的 0 也可以省略。

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。

由于数字值可以使用 number 对象进行封装，因此数字值可以调用 Number.prototype 中的方法。

tofixed()方法可指定小数部分的显示位数

toPrecision（）方法用来指定有效位数的显示位数

2.3.2 较小的数值

在处理带有小数的数字时需要特别注意。很多程序只需要处理整数，最大不超过百万或百亿，此时使用 JavaScript 的数字类型是绝对安全的。

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。

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，也可以为 ES6 之前的版本 polyfill Number.isInteger(..) 方法。

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，可以为 ES6 之前的版本 polyfill Number.isSafeInteger(..) 方法。

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，此时会对它们执行虚拟操作（abstract operation）ToInt32，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 不是值得值

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！）。

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！

永远不要重新定义 undefined。

void 运算符 undefined 是一个内置标识符，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。

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。

2.3.4 特殊的数字

1. 不是数字的数字

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。

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”。

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

2. 无穷数

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。

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）。

3. 零值

JavaScript 有一个常规的 0（也叫作 +0）和一个 -0。

-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"：

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：

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
值     |     27
-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
```

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，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 === 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

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。

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。

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。

第四章　强制类型转换

4.1 值类型转换

将值从一种类型转换为另一种类型通常称为类型转换（type casting），这是显式的情况；隐式的情况称为强制类型转换（coercion）。

JavaScript 中的强制类型转换总是返回标量基本类型值，如字 符串、数字和布尔值不会返回对象和函数。

也可以这样来区分：类型转换发生在静态类型语言的编译阶段，而强制类型转换则发生在动态类型语言的运行时（runtime）。

然而在 JavaScript 中通常将它们统称为强制类型转换，我个人则倾向于用“隐式强制类型 转换”（implicit coercion）和“显式强制类型转换”（explicit coercion）来区分。

二者的区别显而易见：我们能够从代码中看出哪些地方是显式强制类型转换，而隐式强制 类型转换则不那么明显，通常是某些操作产生的副作用。

4.2 抽象值操作

介绍显式和隐式强制类型转换之前，我们需要掌握字符串、数字和布尔值之间类型转换的基本规则。ES5 规范第 9 节中定义了一些“抽象操作”（即“仅供内部使用的操作”）和转 换规则。这里我们着重介绍 ToString、ToNumber 和 ToBoolean，附带讲一讲 ToPrimitive。

4.2.1 ToString

它负责处理非字符串到字符串的强制类型转换。

基本类型值的字符串化规则为：null 转换为 "null"，undefined 转换为 "undefined"，true 转换为 "true"。数字的字符串化则遵循通用规则，不过第 2 章中讲过的那些极小和极大的 数字使用指数形式。

对普通对象来说，除非自行定义，否则 toString()（Object.prototype.toString()）返回 内部属性 [[Class]] 的值，如 "[object Object]"。如果对象有自己的 toString() 方法，字符串化时就会调用该方法并 使用其返回值。

数组的默认 toString() 方法经过了重新定义，将所有单元字符串化以后再用 "," 连接起 来：

```
var a = [1,2,3];
a.toString(); // "1,2,3"
```

toString() 可以被显式调用，或者在需要字符串化时自动调用。

JSON 字符串化

工具函数 JSON.stringify(..) 在将 JSON 对象序列化为字符串时也用到了 ToString。

请注意，JSON 字符串化并非严格意义上的强制类型转换，因为其中也涉及 ToString 的相 关规则，所以这里顺带介绍一下。

对大多数简单值来说，JSON 字符串化和 toString() 的效果基本相同，只不过序列化的结 果总是字符串：

```
JSON.stringify( 42 );    // "42"
JSON.stringify( "42" ); // ""42"" （含有双引号的字符串）
 JSON.stringify( null ); // "null"
 JSON.stringify( true ); // "true"
```

所有安全的 JSON 值（JSON-safe）都可以使用 JSON.stringify(..) 字符串化。安全的 JSON 值是指能够呈现为有效 JSON 格式的值。

JSON.stringify(..) 在对象中遇到 undefined、function 和 symbol 时会自动将其忽略，在 数组中则会返回 null（以保证单元位置不变）。

对包含循环引用的对象执行 JSON.stringify(..) 会出错。

如果对象中定义了 toJSON() 方法，JSON 字符串化时会首先调用该方法，然后用它的返回 值来进行序列化。

如果要对含有非法 JSON 值的对象做字符串化，或者对象中的某些值无法被序列化时，就 需要定义 toJSON() 方法来返回一个安全的 JSON 值。

toJSON() 应该“返回一个能够被字符串化的安全的 JSON 值”，而不是“返回 一个 JSON 字符串”。

4.2.2 ToNumber

ToNumber 对字符串的处理基本遵循数字常量的相关规则 / 语法。处理失败 时返回 NaN（处理数字常量失败时会产生语法错误）。不同之处是 ToNumber 对以 0 开头的 十六进制数并不按十六进制处理。

对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型 值，则再遵循以上规则将其强制转换为数字。

为了将值转换为相应的基本类型值，抽象操作 ToPrimitive。会首先 （通过内部操作 DefaultValue,）检查该值是否有 valueOf() 方法。 如果有并且返回基本类型值，就使用该值进行强制类型转换。如果没有就使用 toString() 的返回值（如果存在）来进行强制类型转换。

如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误。

4.2.3 ToBoolean

，JavaScript 中有两个关键词 true 和 false，分别代表布尔类型 中的真和假。我们常误以为数值 1 和 0 分别等同于 true 和 false。在有些语言中可能是这 样，但在 JavaScript 中布尔值和数字是不一样的。虽然我们可以将 1 强制类型转换为 true， 将 0 强制类型转换为 false，反之亦然，但它们并不是一回事。

1. 假值（falsy value）

我们再来看看其他值是如何被强制类型转换为布尔值的。

JavaScript 中的值可以分为以下两类:

(1) 可以被强制类型转换为 false 的值

(2) 其他（被强制类型转换为 true 的值）

以下这些是假值:

- undefined
- null
- false
- +0、-0 和 NaN
- ""

2.假值对象（falsy object）

值得注意的是，虽然 JavaScript 代码中会出现假值对象，但它实际上并不属于 JavaScript 语 言的范畴。

浏览器在某些特定情况下，在常规 JavaScript 语法基础上自己创建了一些外来（exotic） 值，这些就是“假值对象”。

假值对象看起来和普通对象并无二致（都有属性，等等），但将它们强制类型转换为布尔 值时结果为 false。

最常见的例子是 document.all，它是一个类数组对象，包含了页面上的所有元素，由 DOM（而不是 JavaScript 引擎）提供给 JavaScript 程序使用。它以前曾是一个真正意义上 的对象，布尔强制类型转换结果为 true，不过现在它是一个假值对象。

3.真值（truthy value）

真值就是假值列表之外的值。

4.3 显式强制类型转换

显式强制类型转换是那些显而易见的类型转换，很多类型转换都属于此列。

我们在编码时应尽可能地将类型转换表达清楚，以免给别人留坑。类型转换越清晰，代码可读性越高，更容易理解。

4.3.1 字符串和数字之间的显式转换

字符串和数字之间的转换是通过 String(..) 和 Number(..) 这两个内建函数来实现的，请注意它们前面没有 new 关键字，并不创

建封装对象。

String(..) 遵循前面讲过的 ToString 规则，将值转换为字符串基本类型。Number(..) 遵循 前面讲过的 ToNumber 规则，将值转换为数字基本类型。

一元运算符 - 和 + 一样，并且它还会反转数字的符号位。由于 -- 会被当作递减运算符来处 理，所以我们不能使用 -- 来撤销反转，而应该像 - -"3.14" 这样，在中间加一个空格，才 能得到正确结果 3.14。

1.日期显式转换为数字

一元运算符 + 的另一个常见用途是将日期 (Date) 对象强制类型转换为数字，返回结果为 Unix 时间戳，以微秒为单位 (从 1970 年 1 月 1 日 00:00:00 UTC 到当前时间)：

```
var d = new Date( "Mon, 18 Aug 2014 08:53:06 CDT" );

+d; // 1408369986000
```

JavaScript 有一处奇特的语法，即构造函数没有参数时可以不用带 ()。于是 我们可能会碰到 var timestamp = +new Date; 这样的写法。这样能否提高代 码可读性还存在争议，因为这仅用于 new fn()，对一般的函数调用 fn() 并 不适用。

将日期对象转换为时间戳并非只有强制类型转换这一种方法，或许使用更显式的方法会更 好一些：

```
var timestamp = new Date().getTime();
// var timestamp = (new Date()).getTime();
// var timestamp = (new Date).getTime();
```

不过最好还是使用 ES5 中新加入的静态方法 Date.now()：

```
var timestamp = Date.now();
```

为老版本浏览器提供 Date.now() 的 polyfill 也很简单：

```
if (!Date.now) {
        Date.now = function() {
                return +new Date();        };
}
```

我们不建议对日期类型使用强制类型转换，应该使用 Date.now() 来获得当前的时间戳，使 用 new Date(..).getTime() 来获得指定时间的时间戳。

2.奇特的 ~ 运算符

一个常被人忽视的地方是 ~ 运算符（即字位操作“非”）相关的强制类型转换，它很让人 费解，以至于了解它的开发人员也常常对其敬而远之。秉承本书的一贯宗旨，我们在此深 入探讨一下 ~ 有哪些用处。

在 2.3.5 节中，我们讲过字位运算符只适用于 32 位整数，运算符会强制操作数使用 32 位 格式。这是通过抽象操作 ToInt32 来实现的。

ToInt32 首先执行 ToNumber 强制类型转换，比如"123" 会先被转换为 123，然后再执行 ToInt32。

虽然严格说来并非强制类型转换（因为返回值类型并没有发生变化），但字位运算符（如 | 和 ~）和某些特殊数字一起使用时会产生类似强制类型转换的效果，返回另外一个数字。

4.3.2 显式解析数字字符串

解析字符串中的数字和将字符串强制类型转换为数字的返回结果都是数字。但解析和转换 两者之间还是有明显的差别。

解析允许字符串中含有非数字字符，解析按从左到右的顺序，如果遇到非数字字符就停 止。而转换不允许出现非数字字符，否则会失败并返回 NaN。

解析和转换之间不是相互替代的关系。它们虽然类似，但各有各的用途。如果字符串右边 的非数字字符不影响结果，就可以使用解析。而转换要求字符串中所有的字符都是数字， 像 "42px" 这样的字符串就不行。

解析字符串中的浮点数可以使用 parseFloat(..) 函数。

不要忘了 parseInt(..) 针对的是字符串值。向 parseInt(..) 传递数字和其他类型的参数是 没有用的，比如 true、function(){...} 和 [1,2,3]。

非字符串参数会首先被强制类型转换为字符串，依赖这样的隐式强制类型 转换并非上策，应该避免向 parseInt(..) 传递非字符

串参数。

ES5 之前的 parseInt(..) 有一个坑导致了很多 bug。即如果没有第二个参数来指定转换的 基数（又称为 radix）， parseInt(..) 会根据字符串的第一个字符来自行决定基数。

如果第一个字符是 x 或 X，则转换为十六进制数字。如果是 0，则转换为八进制数字。

4.3.3 显式转换为布尔值

与前面的 String(..) 和 Number(..) 一样，Boolean(..)（不带 new）是显式的 ToBoolean 强 制类型转换：

```
var a = "0";
var b = [];
var c = {};

var d = "";
var e = 0;
var f = null;
var g;

Boolean( a ); // true
Boolean( b ); // true
Boolean( c ); // true

Boolean( d ); // false
Boolean( e ); // false
Boolean( f ); // false
Boolean( g ); // false
```

虽然 Boolean(..) 是显式的，但并不常用。

和前面讲过的 + 类似，一元运算符 ！显式地将值强制类型转换为布尔值。但是它同时还将 真值反转为假值（或者将假值反转为真值）。所以显式强制类型转换为布尔值最常用的方法是 !!，因为第二个 ！会将结果反转回原值。

4.4 隐式强制类型转换

4.4.1 隐式地简化

隐式强制类 型转换同样可以用来提高代码可读性。

然而隐式强制类型转换也会带来一些负面影响，有时甚至是弊大于利。因此我们更应该学 习怎样去其糟粕，取其精华。

很多开发人员认为如果某个机制有优点 A 但同时又有缺点 Z，为了保险起见不如全部弃之 不用。

我不赞同这种“因噎废食”的做法。不要因为只看到了隐式强制类型转换的缺点就想当然 地认为它一无是处。它也有好的方面，希望越来越多的开发人员能加以发现和运用。

4.4.2 字符串和数字之间的隐式强制类型转换

通过重载，+ 运算符即能用于数字加法，也能用于字符串拼接。

如果某个操作数是字符串或者能够通过以下步骤转换为字符串 的话，+ 将进行拼接操作。如果其中一个操作数是对象（包括数组），则首先对其调用 ToPrimitive 抽象操作，该抽象操作再调用 [[DefaultValue]]，以数字作为上下文。

简单来说就是，如果 + 的其中一个操作数是字符串（或者通过以上步骤可以得到字符串）， 则执行字符串拼接；否则执行数字加法。

4.4.3 布尔值到数字的隐式强制类型转换

如果其中有且仅有一个参数为 true，则 onlyOne(..) 返回 true。其在条件判断中使用了隐 式强制类型转换，其他地方则是显式的，包括最后的返回值。

无论使用隐式还是显式，我们都能通过修改 onlyTwo(..) 或者 onlyFive(..) 来处理更复杂的情况，只需要将最后的条件判断从 1

改为 2 或 5。这比加入 一大堆 && 和 || 表达式简洁得多。所以强制类型转换在这里还是很有用的。

4.4.4 隐式强制类型转换为布尔值

相对布尔值，数字和字符串操作中的隐式强制类型转换还算比较明显。下面的情况会发生 布尔值隐式强制类型转换。

(1) if (..) 语句中的条件判断表达式。

(2) for (.. ; .. ; ..) 语句中的条件判断表达式（第二个）。

(3) while (..) 和 do..while(..) 循环中的条件判断表达式。

(4) ? : 中的条件判断表达式。

(5) 逻辑运算符 ||（逻辑或）和 &&（逻辑与）左边的操作数（作为条件判断表达式）。

4.4.5 || 和 &&

|| 和 && 首先会对第一个操作数（a 和 c）执行条件判断，如果其不是布尔值（如上例）就 先进行 ToBoolean 强制类型转换，然后再执行条件判断。

对于|| 来说，如果条件判断结果为 true 就返回第一个操作数（a 和 c）的值，如果为 false 就返回第二个操作数（b）的值。

&& 则相反，如果条件判断结果为 true 就返回第二个操作数（b）的值，如果为 false 就返 回第一个操作数（a 和 c）的值。

|| 和 && 返回它们其中一个操作数的值，而非条件判断的结果（其中可能涉及强制类型转 换）。 c && b 中 c 为 null，是一个假值，因此 && 表达式的结果是 null（即 c 的值），而非 条件判断的结果 false。

4.4.6 符号的强制类型转换

符号不能够被强制类型转换为数字（显式和隐式都会产生错误），但可以被强制类型转换 为布尔值（显式和隐式结果都是 true）。

由于规则缺乏一致性，我们要对 ES6 中符号的强制类型转换多加小心。

好在鉴于符号的特殊用途，我们不会经常用到它的强制类型转换。

4.5 宽松相等和严格相等

宽松相等（loose equals）== 和严格相等（strict equals）=== 都用来判断两个值是否“相 等”，但是它们之间有一个很重要的区别，特别是在判断条件上。

常见的误区是“== 检查值是否相等，=== 检查值和类型是否相等”。听起来蛮有道理，然而 还不够准确。很多 JavaScript 的书籍和博客也是这样来解释的，但是很遗憾他们都错了。

正确的解释是：“== 允许在相等比较中进行强制类型转换，而 === 不允许。”

4.5.1 相等比较操作的性能

如果进行比较的两个值类型相同，则 == 和 === 使用相同的算法，所以除了 JavaScript 引擎 实现上的细微差别之外，它们之间并没有什么不同。

如果两个值的类型不同，我们就需要考虑有没有强制类型转换的必要，有就用 ==，没有就用 ===，不用在乎性能。

== 和 === 都会检查操作数的类型。区别在于操作数类型不同时它们的处理方 式不同。

4.5.2 抽象相等

“抽象相等”（abstract equality）的这些规则正是隐式强制类型转换被诟病 的原因。开发人员觉得它们太晦涩，很难掌握和运用，弊（导致 bug）大 于利（提高代码可读性）。这种观点我不敢苟同，因为本书的读者都是优秀 的开发人员，整天与算法和代码打交道，“抽象相等”对各位来说只是小菜 一碟。建议大家看一看 ES5 规范 11.9.3 节，你会发现这些规则其实非常简 单明了。

1.字符串和数字之间的相等比较

ES5 规范 11.9.3.4-5 这样定义：

(1) 如果 Type(x) 是数字，Type(y) 是字符串，则返回 x == ToNumber(y) 的结果。

(2) 如果 Type(x) 是字符串，Type(y) 是数字，则返回 ToNumber(x) == y 的结果。

规范使用 Number 和 String 来代表数字和字符串类型，而本书使用的是数字 （number）和字符串（string）。切勿将规范中的 Number 和原生函数 Number() 混为一谈。本书中类型名的首字符大写和小写是一回事。

规范 11.9.3.6-7 是这样说的：

(1) 如果 Type(x) 是布尔类型，则返回 ToNumber(x) == y 的结果；

(2) 如果 Type(y) 是布尔类型，则返回 x == ToNumber(y) 的结果。

3. null 和 undefined 之间的相等比较

null 和 undefined 之间的 == 也涉及隐式强制类型转换。ES5 规范 11.9.3.2-3 规定：

(1) 如果 x 为 null，y 为 undefined，则结果为 true。

(2) 如果 x 为 undefined，y 为 null，则结果为 true。

在 == 中 null 和 undefined 相等（它们也与其自身相等），除此之外其他值都不存在这种 情况。

4. 对象和非对象之间的相等比较 关于对象(对象 / 函数 / 数组)和标量基本类型(字符串 / 数字 / 布尔值)之间的相等比 较，ES5 规范 11.9.3.8-9 做如下规定：

(1) 如果 Type(x) 是字符串或数字，Type(y) 是对象，则返回 x == ToPrimitive(y) 的结果；

(2) 如果 Type(x) 是对象，Type(y) 是字符串或数字，则返回 ToPromitive(x) == y 的结果。

这里只提到了字符串和数字，没有布尔值。原因是我们之前介绍过 11.9.3.6-7 中规定了布尔值会先被强制类型转换为数字。

之前介绍过的 ToPromitive 抽象操作的所有特性（如 toString()、valueOf()） 在这里都适用。如果我们需要自定义 valueOf() 以便从复杂的数据结构返回 一个简单值进行相等比较，这些特性会很有帮助。

4.5.3 比较少见的情况

1. 返回其他数字

```
Number.prototype.valueOf = function() {
        return 3;
        };

new Number( 2 ) == 3;    // true
```

2 == 3 不会有这种问题，因为 2 和 3 都是数字基本类型值，不会调用 Number.prototype.valueOf() 方法。而 Number(2) 涉及 ToPrimitive 强制类型 转换，因此会调用 valueOf()。

2.假值的相等比较

== 中的隐式强制类型转换最为人诟病的地方是假值的相等比较。

下面分别列出了常规和非常规的情况：

```
"0" == null;          // false
"0" == undefined;     // false
"0" == false;         // true -- 晕!
"0" == NaN;           // false
"0" == 0;             // true
"0" == "";            // false

false == null;        // false
false == undefined;   // false
false == NaN;         // false
false == 0;           // true -- 晕!
false == "";          // true -- 晕!
false == [];          // true -- 晕!
false == {};          // false

"" == null;           // false
"" == undefined;      // false
"" == NaN;            // false
"" == 0;              // true -- 晕!
"" == [];             // true -- 晕!
"" == {};             // false
```

```
0 == null;                  // false
0 == undefined;             // false
0 == NaN;                   // false
0 == [];                    // true -- 晕
0      == {};                    // false
```

3.极端情况

这还不算完，还有更极端的例子：

```
[] == ![]     // true
```

事情变得越来越疯狂了。看起来这似乎是真值和假值的相等比较，结果不应该是 true，因 为一个值不可能同时既是真值也是假值！

4. 完整性检查 我们深入介绍了隐式强制类型转换中的一些特殊情况。也难怪大多数开发人员都觉得这太 晦涩，唯恐避之不及。现在回过头来做一下完整性检查（sanity check）。

前面列举了相等比较中的强制类型转换的 7 个坑，不过另外还有至少 17 种情况是绝对安 全和容易理解的。

因为 7 棵歪脖树而放弃整片森林似乎有点因噎废食了，所以明智的做法是扬其长避其短。

再来看看那些“短”的地方：

```
"0" == false;               // true -- 晕！
false == 0;                 // true -- 晕！
false == "";                // true -- 晕！
false == [];                // true -- 晕！
"" == 0;                    // true -- 晕！
"" == [];                   // true -- 晕！
0 == [];                    // true -- 晕！
```

其中有 4 种情况涉及 == false，之前我们说过应该避免，应该不难掌握。

现在剩下 3 种：

```
"" == 0;                    // true -- 晕！
"" == [];                   // true -- 晕！
0 == [];                    // true -- 晕！
```

5. 安全运用隐式强制类型转换

我们要对 == 两边的值认真推敲，以下两个原则可以让我们有效地避免出错。

- 如果两边的值中有 true 或者 false，千万不要使用 ==。
- 如果两边的值中有 []、"" 或者 0，尽量不要使用 ==。

这时最好用 === 来避免不经意的强制类型转换。这两个原则可以让我们避开几乎所有强制 类型转换的坑。

这种情况下强制类型转换越显式越好，能省去很多麻烦。

所以 == 和 === 选择哪一个取决于是否允许在相等比较中发生强制类型转换。

强制类型转换在很多地方非常有用，能够让相等比较更简洁（比如 null 和 undefined）。

隐式强制类型转换在部分情况下确实很危险，这时为了安全起见就要使用 ===。

有一种情况下强制类型转换是绝对安全的，那就是 typeof 操作。typeof 总是 返回七个字符串之一（参见第 1 章），其中没有空字符串。所以在类型检查 过程中不会发生隐式强制类型转换。typeof x == "function" 是 100% 安全 的，和 typeof x === "function" 一样。事实上两者在规范中是一回事。所 以既不要盲目听命于代码工具每一处都用 ===，更不要对这个问题置若罔闻。 我们要对自己的代码负责。

隐式强制类型转换真的那么不堪吗？某些情况下是，但总的来说并非如此。

作为一个成熟负责的开发人员，我们应该学会安全有效地运用强制类型转换（显式和隐 式），并对周围的同行言传身教。

4.6 抽象关系比较

a < b 中涉及的隐式强制类型转换不太引人注意，不过还是很有必要深入了解一下。

ES5 规范 11.8.5 节定义了“抽象关系比较”（abstract relational comparison），分为两个部 分：比较双方都是字符串（后半部分）和其他情况（前半部分）。

该算法仅针对 a < b，a="> b 会被处理为 b <>

比较双方首先调用 ToPrimitive，如果结果出现非字符串，就根据 ToNumber 规则将双方强 制类型转换为数字来进行比较

4.7 小结

JavaScript 的数据类型之间的转换，即强制类型转换：包括显式和隐式。

强制类型转换常常为人诟病，但实际上很多时候它们是非常有用的。作为有使命感的 JavaScript 开发人员，我们有必要深入了解强制类型转换，这样就能取其精华，去其糟粕。

显式强制类型转换明确告诉我们哪里发生了类型转换，有助于提高代码可读性和可维 护性。

隐式强制类型转换则没有那么明显，是其他操作的副作用。感觉上好像是显式强制类型转 换的反面，实际上隐式强制类型转换也有助于提高代码的可读性。

在处理强制类型转换的时候要十分小心，尤其是隐式强制类型转换。在编码的时候，要知 其然，还要知其所以然，并努力让代码清晰易读。