```
"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
```

以上 24 种情况中有 17 种比较好理解。比如我们都知道 `""` 和 `NaN` 不相等，`"0"` 和 `0` 相等。

然而有 7 种我们注释了“晕！”，因为它们属于假阳（false positive）的情况，里面坑很多。`""` 和 `0` 明显是两个不同的值，它们之间的强制类型转换很容易搞错。请注意这里不存在假阴（false negative）的情况。

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！

事实并非如此。让我们看看 ! 运算符都做了些什么？根据 `ToBoolean` 规则，它会进行布尔值的显式强制类型转换（同时反转奇偶校验位）。所以 `[] == ![]` 变成了 `[] == false`。前面我们讲过 `false == []`，最后的结果就顺理成章了。

再来看看其他情况：

```
2 == [2];       // true
"" == [null];   // true
```

介绍 ToNumber 时我们讲过，== 右边的值 [2] 和 [null] 会进行 ToPrimitive 强制类型转换，以便能够和左边的基本类型值（2 和 ""）进行比较。因为数组的 valueOf() 返回数组本身，所以强制类型转换过程中数组会进行字符串化。

第一行中的 [2] 会转换为 "2"，然后通过 ToNumber 转换为 2。第二行中的 [null] 会直接转换为 ""。

所以最后的结果就是 2 == 2 和 "" == ""。

如果还是觉得头大，那么你的困惑可能并非来自强制类型转换，而是 ToPrimitive 将数组转换为字符串这一过程。也许你认为 [2].toString() 返回的不是 "2"，[null].toString() 返回的也不是 ""。

但是如果不这样处理的话又能怎样呢？我实在想不出其他更好的办法。或许应该将 [2] 转换为 "[2]"，但这样的话在别的地方又显得很奇怪。

有人也许会觉得既然 String(null) 返回 "null"，所以 String([null]) 也应该返回 "null"。确实有道理，但这就是问题所在。

隐式强制类型转换本身不是问题的根源，因为 [null] 在显式强制类型转换中也是转换为 ""。问题在于将数组转换为字符串是否合理，具体该如何处理。所以实际上这是 String([..]) 规则的问题。又或者根本就不应该将数组转换为字符串？但这样一来又会导致很多其他问题。

还有一个坑常常被提到：

```
0 == "\n";  // true
```

前面介绍过，""、"\n"（或者 " " 等其他空格组合）等空字符串被 ToNumber 强制类型转换为 0。这样处理总没有问题了吧，不然你要怎么办？

或许可以将空字符串和空格转换为 NaN，这样 " " == NaN 就为 false 了，然而这并没有从根本上解决问题。

0 == "\n" 导致程序出错的几率小之又小，很容易避免。

类型转换总会出现一些特殊情况，并非只有强制类型转换，任何编程语言都是如此。问题出在我们的臆断（有时或许碰巧猜对了？！），但这并不能成为诟病强制类型转换机制的理由。

上述 7 种情况基本涵盖了所有我们可能遇到的坑（除修改 valueOf() 和 toStrign() 的情况

以外）。

与前面 24 种情况列表相对应的是下面这个列表：

```
42 == "43";                     // false
"foo" == 42;                    // false
"true" == true;                 // false

42 == "42";                     // true
"foo" == [ "foo" ];             // true
```

这些是非假值的常规情况（实际上还可以加上无穷大数字的相等比较），其中涉及的强制类型转换是安全的，也比较好理解。

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!
```

正常情况下我们应该不会这样来写代码。我们应该不太可能会用 `== []` 来做条件判断，而是用 `== ""` 或者 `== 0`，如：

```
function doSomething(a) {
    if (a == "") {
        // ..
```

```
        }
    }
```

如果不小心碰到 `doSomething(0)` 和 `doSomething([])` 这样的情况，结果会让你大吃一惊。又如：

```
function doSomething(a,b) {
    if (a == b) {
        // ..
    }
}
```

`doSomething("",0)` 和 `doSomething([],"")` 也会如此。

这些特殊情况会导致各种问题，我们要多加小心，好在它们并不十分常见。

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。

Alex Dorey（GitHub 用户名 @dorey）在 GitHub 上制作了一张图表，列出了各种相等比较的情况，如图 4-1 所示。

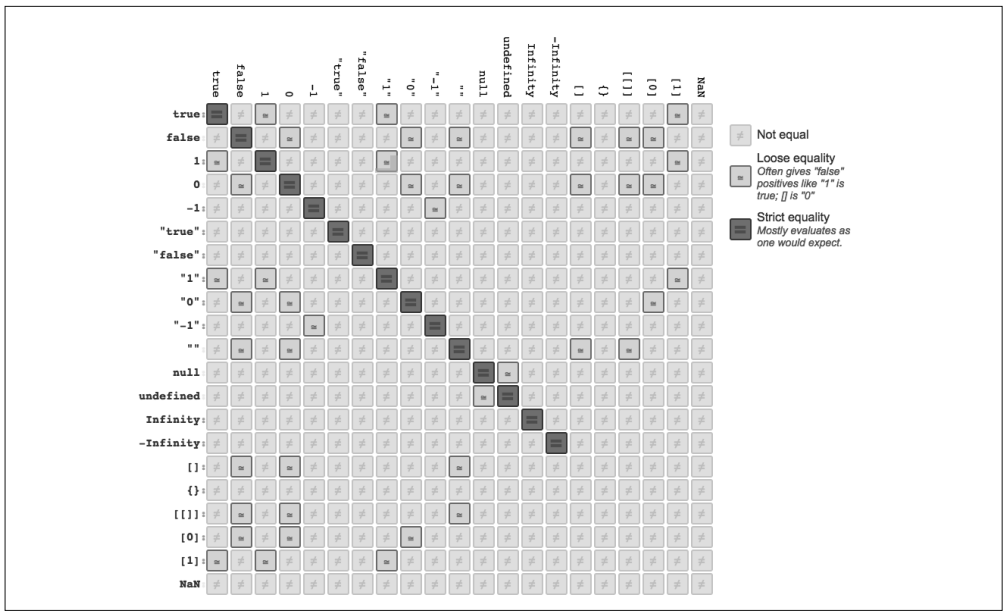

图 4-1：JavaScript 中的相等比较

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。

例如：

```
var a = [ 42 ];
var b = [ "43" ];
```

```
a < b;  // true
b < a;  // false
```

前面介绍过的 -0 和 NaN 的相关规则在这里也适用。

如果比较双方都是字符串，则按字母顺序来进行比较：

```
var a = [ "42" ];
var b = [ "043" ];

a < b;  // false
```

a 和 b 并没有被转换为数字，因为 ToPrimitive 返回的是字符串，所以这里比较的是 "42" 和 "043" 两个字符串，它们分别以 "4" 和 "0" 开头。因为 "0" 在字母顺序上小于 "4"，所以最后结果为 false。

同理：

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

a < b;  // false
```

a 转换为 "4, 2"，b 转换为 "0, 4, 3"，同样是按字母顺序进行比较。

再比如：

```
var a = { b: 42 };
var b = { b: 43 };

a < b;  // ??
```

结果还是 false，因为 a 是 [object Object]，b 也是 [object Object]，所以按照字母顺序 a < b 并不成立。

下面的例子就有些奇怪了：

```
var a = { b: 42 };
var b = { b: 43 };

a < b;  // false
a == b; // false
a > b;  // false

a <= b; // true
a >= b; // true
```

为什么 a == b 的结果不是 true？它们的字符串值相同（同为 "[object Object]"），按道理应该相等才对？实际上不是这样，你可以回忆一下前面讲过的对象的相等比较。

但是如果 a < b 和 a == b 结果为 false，为什么 a <= b 和 a >= b 的结果会是 true 呢？

因为根据规范 a <= b 被处理为 b < a，然后将结果反转。因为 b < a 的结果是 false，所以 a <= b 的结果是 true。

这可能与我们设想的大相径庭，即 <= 应该是“小于或者等于”。实际上 JavaScript 中 <= 是“不大于”的意思（即 !(a > b)，处理为 !(b < a)）。同理 a >= b 处理为 b <= a。

相等比较有严格相等，关系比较却没有“严格关系比较”（strict relational comparison）。也就是说如果要避免 a < b 中发生隐式强制类型转换，我们只能确保 a 和 b 为相同的类型，除此之外别无他法。

与 == 和 === 的完整性检查一样，我们应该在必要和安全的情况下使用强制类型转换，如：42 < "43"。换句话说就是为了保证安全，应该对关系比较中的值进行显式强制类型转换：

```
var a = [ 42 ];
var b = "043";

a < b;                      // false -- 字符串比较!
Number( a ) < Number( b );  // true -- 数字比较!
```

4.7 小结

本章介绍了 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。

第 5 章

语法

语法（grammar）是本部分讨论的最后一个重点。也许你觉得自己已经会用 JavaScript 编程了，然而 JavaScript 语法中仍然有很多地方容易产生困惑、造成误解，本章将对此进行深入的介绍。

相比“词法”（syntax），“语法”一词对读者来说可能更陌生一些。很多时候二者是同一个意思，都是语言规则的定义。虽然它们之间有一些微小的差别，但我们这里可以忽略不计。JavaScript 语法定义了词法规则（syntax rule，如运算符和关键词等）是如何构成可运行的程序代码的。换句话说，只看词法不看语法会遗漏掉很多重要的细节。所以准确地说，本章介绍的是语法，虽然和开发人员直接打交道的是词法。

5.1 语句和表达式

开发人员常常将“语句”（statement）和“表达式”（expression）混为一谈，但这里我们要将二者区别开来，因为它们在 JavaScript 中存在一些重要差别。

你应该对英语更熟悉，这里我们就借用它的术语来说明问题。

“句子”（sentence）是完整表达某个意思的一组词，由一个或多个“短语”（phrase）组成，它们之间由标点符号或连接词（and 和 or 等）连接起来。短语可以由更小的短语组成。有些短语是不完整的，不能独立表达意思；有些短语则相对完整，并且能够独立表达某个意思。这些规则就是英语的*语法*。

JavaScript 的语法也是如此。语句相当于句子，表达式相当于短语，运算符则相当于标点符号和连接词。

JavaScript 中表达式可以返回一个结果值。例如：

```
var a = 3 * 6;
var b = a;
b;
```

这里，`3 * 6` 是一个表达式（结果为 `18`）。第二行的 `a` 也是一个表达式，第三行的 `b` 也是。表达式 `a` 和 `b` 的结果值都是 `18`。

这三行代码都是包含表达式的语句。`var a = 3 * 6` 和 `var b = a` 称为“声明语句”（declaration statement），因为它们声明了变量（还可以为其赋值）。`a = 3 * 6` 和 `b = a`（不带 `var`）叫作“赋值表达式”。

第三行代码中只有一个表达式 `b`，同时它也是一个语句（虽然没有太大意义）。这样的情况通常叫作“表达式语句”（expression statement）。

5.1.1 语句的结果值

很多人不知道，语句都有一个结果值（statement completion value，`undefined` 也算）。

获得结果值最直接的方法是在浏览器开发控制台中输入语句，默认情况下控制台会显示所执行的最后一条语句的结果值。

以赋值表达式 `b = a` 为例，其结果值是赋给 `b` 的值（`18`），但规范定义 `var` 的结果值是 `undefined`。如果在控制台中输入 `var a = 42` 会得到结果值 `undefined`，而非 `42`。

从技术角度来解释要更复杂一些。ES5 规范 12.2 节中的变量声明（`VariableDeclaration`）算法实际上有一个返回值（是一个包含所声明变量名称的字符串，很奇特吧？），但是这个值被变量语句（`VariableStatement`）算法屏蔽掉了（`for..in` 循环除外），最后返回结果为空（`undefined`）。

如果你用开发控制台（或者 JavaScript REPL——read/evaluate/print/loop 工具）调试过代码，应该会看到很多语句的返回值显示为 `undefined`，只是你可能从未探究过其中的原因。其实控制台中显示的就是语句的结果值。

但我们在代码中是没有办法获得这个结果值的，具体解决方法比较复杂，首先得弄清楚为什么要获得语句的结果值。

先来看看其他语句的结果值。比如代码块 `{ .. }` 的结果值是其最后一个语句 / 表达式的

结果。

例如：

```
var b;

if (true) {
    b = 4 + 38;
}
```

在控制台 /REPL 中输入以上代码应该会显示 42，即最后一个语句 / 表达式 `b = 4 + 38` 的结果值。

换句话说，代码块的结果值就如同一个*隐式的返回*，即返回最后一个语句的结果值。

与此类似，CoffeeScript 中的函数也会隐式地返回最后一个语句的结果值。

但下面这样的代码无法运行：

```
var a, b;

a = if (true) {
    b = 4 + 38;
};
```

因为语法不允许我们获得语句的结果值并将其赋值给另一个变量（至少目前不行）。

那应该怎样获得语句的结果值呢？

以下代码仅为演示，切勿在实际开发中这样操作！

可以使用万恶的 `eval(..)`（又读作“evil”）来获得结果值：

```
var a, b;

a = eval( "if (true) { b = 4 + 38; }" );

a;  // 42
```

这并不是个好办法，但确实管用。

ES7 规范有一项“do 表达式”（do expression）提案，类似下面这样：

```
var a, b;

a = do {
    if (true) {
        b = 4 + 38;
    }
};

a;  // 42
```

上例中，do { .. } 表达式执行一个代码块（包含一个或多个语句），并且返回其中最后一个语句的结果值，然后赋值给变量 a。

其目的是将语句当作表达式来处理（语句中可以包含其他语句），从而不需要将语句封装为函数再调用 return 来返回值。

虽然目前语句的结果值还无关紧要，但随着 JavaScript 语言的演进，它可能会扮演越来越重要的角色。希望 do { .. } 表达式的引入能够减少对 eval(..) 这类方法的使用。

再次强调：不要使用 eval(..)。详情请参见《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分。

5.1.2 表达式的副作用

大部分表达式没有副作用。例如：

```
var a = 2;
var b = a + 3;
```

表达式 a + 3 本身没有副作用（比如改变 a 的值）。它的结果值为 5，通过 b = a + 3 赋值给变量 b。

最常见的有副作用（也可能没有）的表达式是函数调用：

```
function foo() {
    a = a + 1;
}

var a = 1;
foo();      // 结果值：undefined。副作用：a的值被改变
```

其他一些表达式也有副作用，比如：

```
var a = 42;
var b = a++;
```

a++ 首先返回变量 a 的当前值 42（再将该值赋给 b），然后将 a 的值加 1：

```
var a = 42;
var b = a++;

a;  // 43
b;  // 42
```

很多开发人员误以为变量 b 和 a 的值都是 43，这是因为没有完全理解 ++ 运算符的副作用何时产生。

递增运算符 ++ 和递减运算符 -- 都是一元运算符（参见第 4 章），它们既可以用在操作数的前面，也可以用在后面：

```
var a = 42;

a++;    // 42
a;      // 43

++a;    // 44
a;      // 44
```

++ 在前面时，如 ++a，它的副作用（将 a 递增）产生在表达式返回结果值之前，而 a++ 的副作用则产生在之后。

++a++ 会产生 ReferenceError 错误，因为运算符需要将产生的副作用赋值给一个变量。以 ++a++ 为例，它首先执行 a++（根据运算符优先级，如下），返回 42，然后执行 ++42，这时会产生 ReferenceError 错误，因为 ++ 无法直接在 42 这样的值上产生副作用。

常有人误以为可以用括号 ( ) 将 a++ 的副作用封装起来，例如：

```
var a = 42;
var b = (a++);

a;  // 43
b;  // 42
```

事实并非如此。( ) 本身并不是一个封装表达式，不会在表达式 a++ 产生副作用之后执行。即便可以，a++ 会首先返回 42，除非有表达式在 ++ 之后再次对 a 进行运算，否则还是不会得到 43，也就不能将 43 赋值给 b。

但也不是没有办法，可以使用 , 语句系列逗号运算符（statement-series comma operator）将

多个独立的表达式语句串联成一个语句：

```
var a = 42, b;
b = ( a++, a );

a;  // 43
b;  // 43
```

由于运算符优先级的关系，a++， a 需要放到（ .. ）中。本章后面将会介绍。

a++，a 中第二个表达式 a 在 a++ 之后执行，结果为 43，并被赋值给 b。

再如 delete 运算符。第 2 章讲过，delete 用来删除对象中的属性和数组中的单元。它通常以单独一个语句的形式出现：

```
var obj = {
    a: 42
};

obj.a;          // 42
delete obj.a;   // true
obj.a;          // undefined
```

如果操作成功，delete 返回 true，否则返回 false。其副作用是属性被从对象中删除（或者单元从 array 中删除）。

操作成功是指对于那些不存在或者存在且可配置（configurable，参见《你不知道的 JavaScript（上卷）》的“this 和对象原型”部分的第 3 章）的属性，delete 返回 true，否则返回 false 或者报错。

另一个有趣的例子是 = 赋值运算符。

例如：

```
var a;

a = 42;     // 42
a;          // 42
```

a = 42 中的 = 运算符看起来没有副作用，实际上它的结果值是 42，它的副作用是将 42 赋值给 a。

组合赋值运算符，如 += 和 -= 等也是如此。例如，a = b += 2 首先执行 b += 2（即 b = b + 2），然后结果再被赋值给 a。

多个赋值语句串联时（链式赋值，chained assignment），赋值表达式（和语句）的结果值就能派上用场，比如：

```
var a, b, c;

a = b = c = 42;
```

这里 c = 42 的结果值为 42（副作用是将 c 赋值 42），然后 b = 42 的结果值为 42（副作用是将 b 赋值 42），最后是 a = 42（副作用是将 a 赋值 42）。

链式赋值常常被误用，例如 var a = b = 42，看似和前面的例子差不多，实则不然。如果变量 b 没有在作用域中象 var b 这样声明过，则 var a = b = 42 不会对变量 b 进行声明。在严格模式中这样会产生错误，或者会无意中创建一个全局变量（参见《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分）。

另一个需要注意的问题是：

```
function vowels(str) {
    var matches;

    if (str) {
        // 提取所有元音字母
        matches = str.match( /[aeiou]/g );

        if (matches) {
            return matches;
        }
    }
}

vowels( "Hello World" ); // ["e","o","o"]
```

上面的代码没问题，很多开发人员也喜欢这样做。其实我们可以利用赋值语句的副作用将两个 if 语句合二为一：

```
function vowels(str) {
    var matches;

    // 提取所有元音字母
    if (str && (matches = str.match( /[aeiou]/g ))) {
        return matches;
```

```
    }
}

vowels( "Hello World" ); // ["e","o","o"]
```

将 `matches = str.match..` 放到 `( .. )` 中是必要的，原因请参见 5.2 节。

我更偏向后者，因为它更简洁，能体现两个条件的关联性。不过这只是个人偏好，无关对错。

5.1.3 上下文规则

在 JavaScript 语法规则中，有时候同样的语法在不同的情况下会有不同的解释。这些语法规则孤立起来会很难理解。

这里我们不一一列举，只介绍一些常见情况。

1. 大括号

下面两种情况会用到大括号 `{ .. }`（随着 JavaScript 的演进会出现更多类似的情况）。

(1) 对象常量

用大括号定义对象常量（`object` literal）：

```
// 假定函数bar()已经定义

var a = {
    foo: bar()
};
```

`{ .. }` 被赋值给 `a`，因而它是一个对象常量。

`a` 是赋值的对象，称为“左值”（l-value）。`{ .. }` 是所赋的值（即本例中赋给变量 `a` 的值），称为“右值”（r-value）。

(2) 标签

如果将上例中的 `var a =` 去掉会发生什么情况呢？

```
// 假定函数bar()已经定义
```