+ 作为数字加法操作是可互换的，即 2 + 3 等同于 3 + 2。作为字符串拼接操作则不行，但对空字符串 "" 来说，a + "" 和 "" + a 结果一样。

a + "" 这样的隐式转换十分常见，一些对隐式强制类型转换持批评态度的人也不能免俗。

这本身就很能说明问题，无论怎样被人诟病，*隐式强制类型转换*仍然有其用武之地。

a + ""（隐式）和前面的 String(a)（显式）之间有一个细微的差别需要注意。根据 ToPrimitive 抽象操作规则，a + "" 会对 a 调用 valueOf() 方法，然后通过 ToString 抽象操作将返回值转换为字符串。而 String(a) 则是直接调用 ToString()。

它们最后返回的都是字符串，但如果 a 是对象而非数字结果可能会不一样！

例如：

```
var a = {
    valueOf: function() { return 42; },
    toString: function() { return 4; }
};

a + "";         // "42"

String( a );    // "4"
```

你一般不太可能会遇到这个问题，除非你的代码中真的有这些匪夷所思的数据结构和操作。在定制 valueOf() 和 toString() 方法时需要特别小心，因为这会影响强制类型转换的结果。

再来看看从字符串强制类型转换为数字的情况。

```
var a = "3.14";
var b = a - 0;

b; // 3.14
```

- 是数字减法运算符，因此 a - 0 会将 a 强制类型转换为数字。也可以使用 a * 1 和 a / 1，因为这两个运算符也只适用于数字，只不过这样的用法不太常见。

对象的 - 操作与 + 类似：

```
var a = [3];
var b = [1];

a - b; // 2
```

为了执行减法运算，a 和 b 都需要被转换为数字，它们首先被转换为字符串（通过

toString()），然后再转换为数字。

字符串和数字之间的隐式强制类型转换真如人们所说的那样糟糕吗？我个人不这么看。

b = String(a)（显式）和 b = a + ""（隐式）各有优点，b = a + "" 更常见一些。虽然饱受诟病，但隐式强制类型转换仍然有它的用处。

4.4.3 布尔值到数字的隐式强制类型转换

在将某些复杂的布尔逻辑转换为数字加法的时候，隐式强制类型转换能派上大用场。当然这种情况并不多见，属于特殊情况特殊处理。

例如：

```
function onlyOne(a,b,c) {
    return !!((a && !b && !c) ||
        (!a && b && !c) || (!a && !b && c));
}

var a = true;
var b = false;

onlyOne( a, b, b ); // true
onlyOne( b, a, b ); // true

onlyOne( a, b, a ); // false
```

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。

但如果有多个参数时（4 个、5 个，甚至 20 个），用上面的代码就很难处理了。这时就可以使用从布尔值到数字（0 或 1）的强制类型转换：

```
function onlyOne() {
    var sum = 0;
    for (var i=0; i < arguments.length; i++) {
        // 跳过假值,和处理0一样,但是避免了NaN
        if (arguments[i]) {
            sum += arguments[i];
        }
    }
    return sum == 1;
}

var a = true;
var b = false;

onlyOne( b, a );                // true
onlyOne( b, a, b, b, b );       // true
```

```
onlyOne( b, b );                // false
onlyOne( b, a, b, b, b, a );    // false
```

在 onlyOne(..) 中除了使用 for 循环，还可以使用 ES5 规范中的 reduce(..) 函数。

通过 sum += arguments[i] 中的隐式强制类型转换，将真值（true/truthy）转换为 1 并进行累加。如果有且仅有一个参数为 true，则结果为 1；否则不等于 1，sum == 1 条件不成立。

同样的功能也可以通过显式强制类型转换来实现：

```
function onlyOne() {
    var sum = 0;
    for (var i=0; i < arguments.length; i++) {
        sum += Number( !!arguments[i] );
    }
    return sum === 1;
}
```

!!arguments[i] 首先将参数转换为 true 或 false。因此非布尔值参数在这里也是可以的，比如：onlyOne("42", 0)（否则的话，字符串会执行拼接操作，这样结果就不对了）。

转换为布尔值以后，再通过 Number(..) 显式强制类型转换为 0 或 1。

这里使用显式强制类型转换会不会更好一些？注释里说这样的确能够避免 NaN 带来的问题，不过最终是看我们自己的需要。我个人觉得前者，即隐式强制类型转换，更为简洁（前提是不会传递 undefined 和 NaN 这样的值），而显式强制类型转换则会带来一些代码冗余。

总之如本书一贯强调的那样，一切都取决于我们自己的判断和权衡。

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

现在我们来看看到布尔值的隐式强制类型转换，它最为常见也最容易搞错。

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）。

以上情况中，非布尔值会被隐式强制类型转换为布尔值，遵循前面介绍过的 `ToBoolean` 抽象操作规则。

例如：

```
var a = 42;
var b = "abc";
var c;
var d = null;

if (a) {
    console.log( "yep" );       // yep
}

while (c) {
    console.log( "nope, never runs" );
}

c = d ? a : b;
c;                              // "abc"

if ((a && d) || c) {
    console.log( "yep" );       // yep
}
```

上例中的非布尔值会被隐式强制类型转换为布尔值以便执行条件判断。

4.4.5 || 和 &&

逻辑运算符 `||`（或）和 `&&`（与）应该并不陌生，也许正因为如此有人觉得它们在 JavaScript 中的表现也和在其他语言中一样。

这里面有一些非常重要但却不太为人所知的细微差别。

我其实不太赞同将它们称为“逻辑运算符”，因为这不太准确。称它们为“选择器运算符”（selector operators）或者“操作数选择器运算符”（operand selector operators）更恰当些。

为什么？因为和其他语言不同，在 JavaScript 中它们返回的并不是布尔值。

它们的返回值是两个操作数中的一个（且仅一个）。即选择两个操作数中的一个，然后返

回它的值。

引述 ES5 规范 11.11 节：

> && 和 || 运算符的返回值并不一定是布尔类型，而是两个操作数其中一个的值。

例如：

```
var a = 42;
var b = "abc";
var c = null;

a || b;     // 42
a && b;     // "abc"

c || b;     // "abc"
c && b;     // null
```

在 C 和 PHP 中，上例的结果是 true 或 false，在 JavaScript（以及 Python 和 Ruby）中却是某个操作数的值。

|| 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。

现在明白我为什么把它们叫作“操作数选择器”了吧？

换一个角度来理解：

```
a || b;
// 大致相当于(roughly equivalent to):
a ? a : b;

a && b;
// 大致相当于(roughly equivalent to):
a ? b : a;
```

之所以说大致相当，是因为它们返回结果虽然相同但是却有一个细微的差别。在 `a ? a : b` 中，如果 `a` 是一个复杂一些的表达式（比如有副作用的函数调用等），它有可能被执行两次（如果第一次结果为真）。而在 `a || b` 中 `a` 只执行一次，其结果用于条件判断和返回结果（如果适用的话）。`a  b` 和 `a ? b : a` 也是如此。

下面是一个十分常见的 `||` 的用法，也许你已经用过但并未完全理解：

```
function foo(a,b) {
    a = a || "hello";
    b = b || "world";

    console.log( a + " " + b );
}

foo();                  // "hello world"
foo( "yeah", "yeah!" ); // "yeah yeah!"
```

`a = a || "hello"`（又称为 C# 的“空值合并运算符”的 JavaScript 版本）检查变量 `a`，如果还未赋值（或者为假值），就赋予它一个默认值（`"hello"`）。

这里需要注意！

```
foo( "That's it!", "" ); // "That's it! world" <-- 晕!
```

第二个参数 `""` 是一个假值（falsy value，参见 4.2.3 节），因此 `b = b || "world"` 条件不成立，返回默认值 `"world"`。

这种用法很常见，但是其中不能有假值，除非加上更明确的条件判断，或者转而使用 `? :` 三元表达式。

通过这种方式来设置默认值很方便，甚至那些公开诟病 JavaScript 强制类型转换的人也经常使用。

再来看看 `&&`。

有一种用法对开发人员不常见，然而 JavaScript 代码压缩工具常用。就是如果第一个操作数为真值，则 `&&` 运算符“选择”第二个操作数作为返回值，这也叫作“守护运算符”（guard operator，参见 5.2.1 节），即前面的表达式为后面的表达式“把关”：

```
function foo() {
    console.log( a );
}

var a = 42;

a && foo(); // 42
```

`foo()` 只有在条件判断 `a` 通过时才会被调用。如果条件判断未通过，`a && foo()` 就会悄然终止（也叫作“短路”，short circuiting），`foo()` 不会被调用。

这样的用法对开发人员不太常见，开发人员通常使用 `if (a) { foo(); }`。但 JavaScript 代码压缩工具用的是 `a && foo()`，因为更简洁。以后再碰到这样的代码你就知道是怎么回事了。

`||` 和 `&&` 各自有它们的用武之地，前提是我们理解并且愿意在代码中运用隐式强制类型转换。

`a = b || "something"` 和 `a && b()` 用到了“短路”机制，我们将在 5.2.1 节详细介绍。

你大概会有疑问：既然返回的不是 `true` 和 `false`，为什么 `a && (b || c)` 这样的表达式在 `if` 和 `for` 中没出过问题？

这或许并不是代码的问题，问题在于你可能不知道这些条件判断表达式最后还会执行布尔值的隐式强制类型转换。

例如：

```
var a = 42;
var b = null;
var c = "foo";

if (a && (b || c)) {
    console.log( "yep" );
}
```

这里 `a && (b || c)` 的结果实际上是 `"foo"` 而非 `true`，然后再由 `if` 将 `foo` 强制类型转换为布尔值，所以最后结果为 `true`。

现在明白了吧，这里发生了隐式强制类型转换。如果要避免隐式强制类型转换就得这样：

```
if (!!a && (!!b || !!c)) {
    console.log( "yep" );
}
```

4.4.6 符号的强制类型转换

目前我们介绍的显式和隐式强制类型转换结果是一样的，它们之间的差异仅仅体现在代码可读性方面。

但 ES6 中引入了符号类型，它的强制类型转换有一个坑，在这里有必要提一下。ES6 允许从符号到字符串的显式强制类型转换，然而隐式强制类型转换会产生错误，具体的原因不在本书讨论范围之内。

例如：

```
var s1 = Symbol( "cool" );
String( s1 );     // "Symbol(cool)"

var s2 = Symbol( "not cool" );
s2 + "";      // TypeError
```

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（参见第 3 章），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

我们来看一看两种解释的区别。

根据第一种解释（不准确的版本），=== 似乎比 == 做的事情更多，因为它还要检查值的类型。第二种解释中 == 的工作量更大一些，因为如果值的类型不同还需要进行强制类型转换。

有人觉得 == 会比 === 慢，实际上虽然强制类型转换确实要多花点时间，但仅仅是微秒级（百万分之一秒）的差别而已。

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

ES5 规范 11.9.3 节的“抽象相等比较算法”定义了 == 运算符的行为。该算法简单而又全面，涵盖了所有可能出现的类型组合，以及它们进行强制类型转换的方式。

“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。

其中第一段（11.9.3.1）规定如果两个值的类型相同，就仅比较它们是否相等。例如，`42` 等于 `42`，`"abc"` 等于 `"abc"`。

有几个非常规的情况需要注意。

- `NaN` 不等于 `NaN`（参见第 2 章）。
- `+0` 等于 `-0`（参见第 2 章）。

11.9.3.1 的最后定义了对象（包括函数和数组）的宽松相等 ==。两个对象指向同一个值时即视为相等，不发生强制类型转换。

=== 的定义和 11.9.3.1 一样，包括对象的情况。实际上在比较两个对象的时候，== 和 === 的工作原理是一样的。

11.9.3 节中还规定，== 在比较两个不同类型的值时会发生*隐式*强制类型转换，会将其中之一或两者都转换为相同的类型后再进行比较。

宽松不相等（loose not-equality）!= 就是 == 的相反值，!== 同理。

1. 字符串和数字之间的相等比较

我们沿用本章前面字符串和数字的例子来解释 == 中的强制类型转换：

```
var a = 42;
var b = "42";

a === b;    // false
a == b;     // true
```

因为没有强制类型转换，所以 a === b 为 false，42 和 "42" 不相等。

而 a == b 是宽松相等，即如果两个值的类型不同，则对其中之一或两者都进行强制类型转换。

具体怎么转换？是 a 从 42 转换为字符串，还是 b 从 "42" 转换为数字？

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。

根据规范，"42" 应该被强制类型转换为数字以便进行相等比较。相关规则，特别是 ToNumber 抽象操作的规则前面已经介绍过。本例中两个值相等，均为 42。

2. 其他类型和布尔类型之间的相等比较

== 最容易出错的一个地方是 true 和 false 与其他类型之间的相等比较。

例如：

```
var a = "42";
var b = true;

a == b; // false
```

我们都知道 "42" 是一个真值（见本章前面部分），为什么 == 的结果不是 true 呢？原因既简单又复杂，让人很容易掉坑里，很多 JavaScript 开发人员对这个地方并未引起足够的重视。

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。

仔细分析例子，首先：

```
var x = true;
var y = "42";

x == y; // false
```

Type(x) 是布尔值，所以 ToNumber(x) 将 true 强制类型转换为 1，变成 1 == "42"，二者的类型仍然不同，"42" 根据规则被强制类型转换为 42，最后变成 1 == 42，结果为 false。

反过来也一样：

```
var x = "42";
var y = false;

x == y; // false
```

Type(y) 是布尔值，所以 ToNumber(y) 将 false 强制类型转换为 0，然后 "42" == 0 再变成 42 == 0，结果为 false。

也就是说，字符串 "42" 既不等于 true，也不等于 false。一个值怎么可以既非真值也非假值，这也太奇怪了吧？

这个问题本身就是错误的，我们被自己的大脑欺骗了。

"42" 是一个真值没错，但 "42" == true 中并没有发生布尔值的比较和强制类型转换。这里不是 "42" 转换为布尔值（true），而是 true 转换为 1，"42" 转换为 42。

这里并不涉及 ToBoolean，所以 "42" 是真值还是假值与 == 本身没有关系！

重点是我们要搞清楚 == 对不同的类型组合怎样处理。== 两边的布尔值会被强制类型转换为数字。

很奇怪吧？我个人建议无论什么情况下都不要使用 == true 和 == false。

请注意，这里说的只是 ==，=== true 和 === false 不允许强制类型转换，所以并不涉及 ToNumber。

例如：

```
var a = "42";
```

```
// 不要这样用,条件判断不成立:
if (a == true) {
    // ..
}

// 也不要这样用,条件判断不成立:
if (a === true) {
    // ..
}

// 这样的显式用法没问题:
if (a) {
    // ..
}

// 这样的显式用法更好:
if (!!a) {
    // ..
}

// 这样的显式用法也很好:
if (Boolean( a )) {
    // ..
}
```

避免了 == true 和 == false（也叫作布尔值的宽松相等）之后我们就不用担心这些坑了。

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。

这也就是说在 == 中 null 和 undefined 是一回事，可以相互进行隐式强制类型转换：

```
var a = null;
var b;

a == b;     // true
a == null;  // true
b == null;  // true

a == false; // false
b == false; // false
a == "";    // false
b == "";    // false
a == 0;     // false
b == 0;     // false
```

null 和 undefined 之间的强制类型转换是安全可靠的，上例中除 null 和 undefined 以外的其他值均无法得到假阳（false positive）结果。个人认为通过这种方式将 null 和 undefined 作为等价值来处理比较好。

例如：

```
var a = doSomething();

if (a == null) {
    // ..
}
```

条件判断 a == null 仅在 doSomething() 返回非 null 和 undefined 时才成立，除此之外其他值都不成立，包括 0、false 和 "" 这样的假值。

下面是显式的做法，其中不涉及强制类型转换，个人感觉更繁琐一些（大概执行效率也会更低）：

```
var a = doSomething();

if (a === undefined || a === null) {
    // ..
}
```

我认为 a == null 这样的隐式强制类型转换在保证安全性的同时还能提高代码可读性。

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。

例如：

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

a == b; // true
```

[42] 首先调用 ToPromitive 抽象操作（参见 4.2 节），返回 "42"，变成 "42" == 42，然后又变成 42 == 42，最后二者相等。

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。

在第 3 章中，我们介绍过“拆封”，即“打开”封装对象（如 new String("abc")），返回其中的基本数据类型值（"abc"）。== 中的 ToPromitive 强制类型转换也会发生这样的情况：

```
var a = "abc";
var b = Object( a );    // 和new String( a )一样

a === b;                // false
a == b;                 // true
```

a == b 结果为 true，因为 b 通过 ToPromitive 进行强制类型转换（也称为“拆封”，英文为 unboxed 或者 unwrapped），并返回标量基本类型值 "abc"，与 a 相等。

但有一些值不这样，原因是 == 算法中其他优先级更高的规则。例如：

```
var a = null;
var b = Object( a );    // 和Object()一样
a == b;                 // false

var c = undefined;
var d = Object( c );    // 和Object()一样
c == d;                 // false

var e = NaN;
var f = Object( e );    // 和new Number( e )一样
e == f;                 // false
```

因为没有对应的封装对象，所以 null 和 undefined 不能够被封装（boxed），Object(null) 和 Object() 均返回一个常规对象。

NaN 能够被封装为数字封装对象，但拆封之后 NaN == NaN 返回 false，因为 NaN 不等于 NaN（参见第 2 章）。

4.5.3 比较少见的情况

我们已经全面介绍了 == 中的隐式强制类型转换（常规和非常规的情况），现在来看一下那些需要特别注意和避免的比较少见的情况。

首先来看看更改内置原生原型会导致哪些奇怪的结果。

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()。

真是让人头大。这也是强制类型转换和 == 被诟病的原因之一。但问题并非出自 JavaScript，而是我们自己。不要有这样的想法，觉得“编程语言应该阻止我们犯错误”。

还有更奇怪的情况：

```
if (a == 2 && a == 3) {
    // ..
}
```

你也许觉得这不可能，因为 a 不会同时等于 2 和 3。但“同时”一词并不准确，因为 a == 2 在 a == 3 之前执行。

如果让 a.valueOf() 每次调用都产生副作用，比如第一次返回 2，第二次返回 3，就会出现这样的情况。这实现起来很简单：

```
var i = 2;

Number.prototype.valueOf = function() {
    return i++;
};

var a = new Number( 42 );

if (a == 2 && a == 3) {
    console.log( "Yep, this happened." );
}
```

再次强调，千万不要这样，也不要因此而抱怨强制类型转换。对一种机制的滥用并不能成为诟病它的借口。我们应该正确合理地运用强制类型转换，避免这些极端的情况。

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：