```
var d = Boolean( a && b && c );

d; // true
```

d 为 true，说明 a、b、c 都为 true。

请注意，这里 Boolean(..) 对 a && b && c 进行了封装，有人可能会问为什么。我们暂且记下，稍后会作说明。你可以试试不用 Boolean(..) 的话 d = a && b && c 会产生什么结果。

如果假值对象并非封装了假值的对象，那它究竟是什么？

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。

document.all 并不是一个标准用法，早就被废止了。

有人也许会问：“既然这样的话，浏览器能否将它彻底去掉？”这个想法是好的，只不过仍然有很多 JavaScript 程序在使用它。

那为什么它要是假值呢？因为我们经常通过将 document.all 强制类型转换为布尔值（比如在 if 语句中）来判断浏览器是否是老版本的 IE。IE 自诞生之日起就始终遵循浏览器标准，较其他浏览器更为有力地推动了 Web 的发展。

if(document.all) { /* it's IE */ } 依然存在于许多程序中，也许会一直存在下去，这对 IE 的用户体验来说不是一件好事。

虽然我们无法彻底摆脱 document.all，但为了让新版本更符合标准，IE 并不打算继续支持 if (document.all) { .. }。

“那我们应该怎么办？”

“也许可以修改 JavaScript 的类型机制，将 document.all 作为假值来处理！”

这并不是一个好办法。大多数 JavaScript 开发人员对这个坑了解得不多，不过更糟糕的还是对其置若罔闻的态度。

3. 真值（truthy value）

真值就是假值列表之外的值。

例如：

```
var a = "false";
var b = "0";
var c = "''";

var d = Boolean( a && b && c );

d;
```

这里 d 应该是 true 还是 false 呢？

答案是 true。上例的字符串看似假值，但所有字符串都是真值。不过 "" 除外，因为它是假值列表中唯一的字符串。

再如：

```
var a = [];             // 空数组——是真值还是假值？
var b = {};             // 空对象——是真值还是假值？
var c = function(){};   // 空函数——是真值还是假值？

var d = Boolean( a && b && c );

d;
```

d 依然是 true。还是同样的道理，[]、{} 和 function(){} 都不在假值列表中，因此它们都是真值。

也就是说真值列表可以无限长，无法一一列举，所以我们只能用假值列表作为参考。

你可以花五分钟时间将假值列表写出来贴在显示器上，或者记在脑子里，在需要判断真 / 假值的时候就可以派上用场。

掌握真 / 假值的重点在于理解布尔强制类型转换（显式和隐式），在此基础上我们就能对强制类型转换示例进行深入介绍。

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。

对显式强制类型转换几乎不存在非议，它类似于静态语言中的类型转换，已被广泛接受，不会有什么坑。我们后面会再讨论这个话题。

4.3.1 字符串和数字之间的显式转换

我们从最常见的字符串和数字之间的强制类型转换开始。

字符串和数字之间的转换是通过 `String(..)` 和 `Number(..)` 这两个内建函数（原生构造函数，参见第 3 章）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。

下面是两者之间的显式强制类型转换：

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

var c = "3.14";
var d = Number( c );

b; // "42"
d; // 3.14
```

`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。

它们和静态语言中的类型转换很像，一目了然，所以我们将它们归为显式强制类型转换。

例如，在 C/C++ 中可以使用 `(int)x` 或 `int(x)` 将 `x` 转换为整数。大部分人倾向于后者，因为它看起来更像函数调用。JavaScript 中的 `Number(x)` 与此十分类似，至于它是否真是一个函数并不重要。

除了 `String(..)` 和 `Number(..)` 以外，还有其他方法可以实现字符串和数字之间的显式转换：

```
var a = 42;
var b = a.toString();

var c = "3.14";
var d = +c;

b; // "42"
d; // 3.14
```

`a.toString()` 是显式的（“toString”意为“to a string”），不过其中涉及隐式转换。因为 `toString()` 对 `42` 这样的基本类型值不适用，所以 JavaScript 引擎会自动为 `42` 创建一个封装对象（参见第 3 章），然后对该对象调用 `toString()`。这里显式转换中含有隐式转换。

上例中 +c 是 + 运算符的一元（unary）形式（即只有一个操作数）。+ 运算符显式地将 c 转换为数字，而非数字加法运算（也不是字符串拼接，见下）。

+c 是显式还是隐式，取决于你自己的理解和经验。如果你已然知道一元运算符 + 会将操作数显式强制类型转换为数字，那它就是显式的。如果不明就里的话，它就是隐式强制类型转换，让你摸不着头脑。

在 JavaScript 开源社区中，一元运算 + 被普遍认为是显式强制类型转换。

不过这样有时候也容易产生误会。例如：

```
var c = "3.14";
var d = 5+ +c;

d; // 8.14
```

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。

运算符的一元和二元形式的组合你也许能够想到很多种情况，下面是一个疯狂的例子：

```
1 + - + + + - + 1;  // 2
```

尽量不要把一元运算符 +（还有 -）和其他运算符放在一起使用。上面的代码可以运行，但非常糟糕。此外 d = +c（还有 d =+ c）也容易和 d += c 搞混，两者天壤之别。

一元运算符 + 紧挨着 ++ 和 -- 也很容易引起混淆。例如 a +++b、a + ++b 和 a + + +b。关于 ++，请参见 5.1.2 节。

我们的目的是让代码更清晰、更易懂，而非适得其反。

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
```

我们常用下面的方法来获得当前的时间戳，例如：

```
var timestamp = +new Date();
```

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（ES5 规范 9.5 节）。

`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。

例如 | 运算符（字位操作“或”）的空操作（no-op）`0 | x`，它仅执行 `ToInt32` 转换（第 2 章中介绍过）：

```
0 | -0;         // 0
0 | NaN;        // 0
0 | Infinity;   // 0
0 | -Infinity;  // 0
```

以上这些特殊数字无法以 32 位格式呈现（因为它们来自 64 位 IEEE 754 标准，参见第 2 章），因此 `ToInt32` 返回 `0`。

关于 `0 | ___` 是显式还是隐式仍存在争议。从规范的角度来说它无疑是显式的，但如果对字位运算符没有这样深入的理解，它可能就是隐式的。为了前后保持一致，我们这里将其视为显式。

再回到 `~`。它首先将值强制类型转换为 32 位数字，然后执行字位操作“非”（对每一个字位进行反转）。

这与 `!` 很相像，不仅将值强制类型转换为布尔值 `<`，还对其做字位反转（参见 4.3.3 节）。

字位反转是个很晦涩的主题，JavaScript 开发人员一般很少需要关心到字位级别。

对 `~` 还可以有另外一种诠释，源自早期的计算机科学和离散数学：`~` 返回 2 的补码。这样一来问题就清楚多了！

`~x` 大致等同于 `-(x+1)`。很奇怪，但相对更容易说明问题：

```
~42;    // -(42+1) ==> -43
```

也许你还是没有完全弄明白 `~` 到底是什么玩意？为什么把它放在强制类型转换一章中介绍？稍安勿躁。

在 `-(x+1)` 中唯一能够得到 `0`（或者严格说是 `-0`）的 `x` 值是 `-1`。也就是说如果 `x` 为 `-1` 时，`~` 和一些数字值在一起会返回假值 `0`，其他情况则返回真值。

然而这与我们讨论的内容有什么关系呢？

`-1` 是一个“哨位值”，哨位值是那些在各个类型中（这里是数字）被赋予了特殊含义的值。在 C 语言中我们用 `-1` 来代表函数执行失败，用大于等于 `0` 的值来代表函数执行成功。

JavaScript 中字符串的 indexOf(..) 方法也遵循这一惯例，该方法在字符串中搜索指定的子字符串，如果找到就返回子字符串所在的位置（从 0 开始），否则返回 -1。

indexOf(..) 不仅能够得到子字符串的位置，还可以用来检查字符串中是否包含指定的子字符串，相当于一个条件判断。例如：

```
var a = "Hello World";

if (a.indexOf( "lo" ) >= 0) {   // true
    // 找到匹配!
}
if (a.indexOf( "lo" ) != -1) {  // true
    // 找到匹配!
}

if (a.indexOf( "ol" ) < 0) {    // true
    // 没有找到匹配!
}
if (a.indexOf( "ol" ) == -1) {  // true
    // 没有找到匹配!
}
```

>= 0 和 == -1 这样的写法不是很好，称为“抽象渗漏”，意思是在代码中暴露了底层的实现细节，这里是指用 -1 作为失败时的返回值，这些细节应该被屏蔽掉。

现在我们终于明白 ~ 有什么用处了！ ~ 和 indexOf() 一起可以将结果强制类型转换（实际上仅仅是转换）为真 / 假值：

```
var a = "Hello World";

~a.indexOf( "lo" );         // -4   <-- 真值!

if (~a.indexOf( "lo" )) {   // true
    // 找到匹配!
}

~a.indexOf( "ol" );         // 0    <-- 假值!
!~a.indexOf( "ol" );        // true

if (!~a.indexOf( "ol" )) {  // true
    // 没有找到匹配!
}
```

如果 indexOf(..) 返回 -1，~ 将其转换为假值 0，其他情况一律转换为真值。

由 -(x+1) 推断 ~-1 的结果应该是 -0，然而实际上结果是 0，因为它是字位操作而非数学运算。

从技术角度来说，if (~a.indexOf(..)) 仍然是对 indexOf(..) 的返回结果进行隐式强制类型转换，0 转换为 false，其他情况转换为 true。但我觉得 ~ 更像显式强制类型转换，前提是我对它有充分的理解。

个人认为 ~ 比 >= 0 和 == -1 更简洁。

3. 字位截除

一些开发人员使用 ~~ 来截除数字值的小数部分，以为这和 Math.floor(..) 的效果一样，实际上并非如此。

~~ 中的第一个 ~ 执行 ToInt32 并反转字位，然后第二个 ~ 再进行一次字位反转，即将所有字位反转回原值，最后得到的仍然是 ToInt32 的结果。

~~ 和 !! 很相似，我们将在 4.3.3 节中介绍。

对 ~~ 我们要多加注意。首先它只适用于 32 位数字，更重要的是它对负数的处理与 Math.floor(..) 不同。

```
Math.floor( -49.6 );    // -50
~~-49.6;                // -49
```

~~x 能将值截除为一个 32 位整数，x | 0 也可以，而且看起来还更简洁。

出于对运算符优先级（详见第 5 章）的考虑，我们可能更倾向于使用 ~~x：

```
~~1E20 / 10;        // 166199296

1E20 | 0 / 10;      // 1661992960
(1E20 | 0) / 10;    // 166199296
```

我们在使用 ~ 和 ~~ 进行此类转换时需要确保其他人也能够看得懂。

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。

例如：

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

```
Number( a );    // 42
parseInt( a );  // 42

Number( b );    // NaN
parseInt( b );  // 42
```

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（参见 4.2.1 节），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。

以 x 和 X 开头的十六进制相对来说还不太容易搞错，而八进制则不然。例如：

```
var hour = parseInt( selectedHour.value );
var minute = parseInt( selectedMinute.value );

console.log(
   "The time you selected was: " + hour + ":" + minute
);
```

上面的代码看似没有问题，但是当小时为 08、分钟为 09 时，结果是 0:0，因为 8 和 9 都不是有效的八进制数。

将第二个参数设置为 10，即可避免这个问题：

```
var hour = parseInt( selectedHour.value, 10 );
var minute = parseInt( selectedMiniute.value, 10 );
```

从 ES5 开始 `parseInt(..)` 默认转换为十进制数，除非另外指定。如果你的代码需要在 ES5 之前的环境运行，请记得将第二个参数设置为 `10`。

解析非字符串

曾经有人发帖吐槽过 `parseInt(..)` 的一个坑：

```
parseInt( 1/0, 19 ); // 18
```

很多人想当然地以为（实际上大错特错）“如果第一个参数值为 `Infinity`，解析结果也应该是 `Infinity`”，返回 `18` 也太无厘头了。

尽管这个例子纯属虚构，我们还是来看看 JavaScript 是否真的这样无厘头。

其中第一个错误是向 `parseInt(..)` 传递非字符串，这完全是在自找麻烦。此时 JavaScript 会将参数强制类型转换为它能够处理的字符串。

有人可能会觉得这不合理，`parseInt(..)` 应该拒绝接受非字符串参数。但如果这样的话，它是否应该抛出一个错误？这是 Java 的做法。一想到 JavaScript 代码中到处是抛出的错误，要在每个地方加上 `try..catch`，我整个人都不好了。

那是不是应该返回 `NaN`？也许吧，但是下面的情况是否应该运行失败？

```
parseInt( new String( "42") );
```

因为它的参数也是一个非字符串。如果你认为此时应该将 `String` 封装对象拆封（unbox）为 `"42"`，那么将 `42` 先转换为 `"42"` 再解析回 `42` 不也合情合理吗？

这种半显式、半隐式的强制类型转换很多时候非常有用。例如：

```
var a = {
    num: 21,
    toString: function() { return String( this.num * 2 ); }
};

parseInt( a ); // 42
```

`parseInt(..)` 先将参数强制类型转换为字符串再进行解析，这样做没有任何问题。因为传递错误的参数而得到错误的结果，并不能归咎于函数本身。

怎么来处理 `Infinity`（`1/0` 的结果）最合理呢？有两个选择：`"Infinity"` 和 `"∞"`，JavaScript 选择的是 `"Infinity"`。

JavaScript 中所有的值都有一个默认的字符串形式，这很不错，能够方便我们调试。

再回到基数 19，这显然是个玩笑话，在实际的 JavaScript 代码中不会用到基数 19。它的有效数字字符范围是 `0-9` 和 `a-i`（区分大小写）。

parseInt(1/0, 19) 实际上是 parseInt("Infinity", 19)。第一个字符是 "I"，以 19 为基数时值为 18。第二个字符 "n" 不是一个有效的数字字符，解析到此为止，和 "42px" 中的 "p" 一样。

最后的结果是 18，而非 Infinity 或者报错。所以理解其中的工作原理对于我们学习 JavaScript 是非常重要的。

此外还有一些看起来奇怪但实际上解释得通的例子：

```
parseInt( 0.000008 );       // 0   ("0" 来自于 "0.000008")
parseInt( 0.0000008 );      // 8   ("8" 来自于 "8e-7")
parseInt( false, 16 );      // 250 ("fa" 来自于 "false")
parseInt( parseInt, 16 );   // 15  ("f" 来自于 "function..")

parseInt( "0x10" );         // 16
parseInt( "103", 2 );       // 2
```

其实 parseInt(..) 函数是十分靠谱的，只要使用得当就不会有问题。因为使用不当而导致一些莫名其妙的结果，并不能归咎于 JavaScript 本身。

4.3.3 显式转换为布尔值

现在我们来看看从非布尔值强制类型转换为布尔值的情况。

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：

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

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

!!a;    // true
!!b;    // true
!!c;    // true

!!d;    // false
!!e;    // false
!!f;    // false
!!g;    // false
```

在 if(..).. 这样的布尔值上下文中，如果没有使用 Boolean(..) 和 !!，就会自动隐式地进行 ToBoolean 转换。建议使用 Boolean(..) 和 !! 来进行显式转换以便让代码更清晰易读。

显式 ToBoolean 的另外一个用处，是在 JSON 序列化过程中将值强制类型转换为 true 或 false：

```
var a = [
    1,
    function(){ /*..*/ },
    2,
    function(){ /*..*/ }
];

JSON.stringify( a ); // "[1,null,2,null]"

JSON.stringify( a, function(key,val){
    if (typeof val == "function") {
        // 函数的ToBoolean强制类型转换
        return !!val;
    }
    else {
        return val;
    }
} );
// "[1,true,2,true]"
```

下面的语法对于熟悉 Java 的人并不陌生：

```
var a = 42;

var b = a ? true : false;
```

三元运算符 ? : 判断 a 是否为真，如果是则将变量 b 赋值为 true，否则赋值为 false。

表面上这是一个显式的 ToBoolean 强制类型转换，因为返回结果是 true 或者 false。

然而这里涉及隐式强制类型转换，因为 a 要首先被强制类型转换为布尔值才能进行条件判断。这种情况称为“显式的隐式”，有百害而无一益，我们应彻底杜绝。

建议使用 Boolean(a) 和 !!a 来进行显式强制类型转换。

4.4 隐式强制类型转换

*隐式强制类型转换*指的是那些隐蔽的强制类型转换，副作用也不是很明显。换句话说，你自己觉得不够明显的强制类型转换都可以算作隐式强制类型转换。

显式强制类型转换旨在让代码更加清晰易读，而隐式强制类型转换看起来就像是它的对立面，会让代码变得晦涩难懂。

对强制类型转换的诟病大多是针对*隐式*强制类型转换。

《JavaScript 语言精粹》的作者 Douglas Crockford 在许多场合和文章中都主张不要使用强制类型转换，认为其非常糟糕。然而他的代码中也大量使用了隐式和显式强制类型转换。实际上他的吐槽大部分是针对 == 运算符，但读完本章你会发现这只是强制类型转换的冰山一角。

问题是，隐式强制类型转换真是如此不堪吗？它是不是 JavaScript 语言的设计缺陷？我们是否应该对其退避三舍？

估计大多数读者会回答“是的”。其实不然，请容我细细道来。

让我们从另一个角度来看待隐式强制类型转换，看看它究竟为何物、该如何使用，不要简单地把它当作“显式强制类型转换的对立面”，因为这样理解过于狭隘，忽略了它们之间一个细微却十分重要的区别。

隐式强制类型转换的作用是减少冗余，让代码更简洁。

4.4.1 隐式地简化

我们先来看一个例子，它不是 JavaScript 代码，而是强类型语言的伪代码：

```
SomeType x = SomeType( AnotherType( y ) )
```

其中变量 y 的值被转换为 SomeType 类型。问题是语言本身不允许直接将 y 转换为

SomeType 类型。于是我们需要一个中间步骤，先将 y 转换为 AnotherType 类型，然后再从 AnotherType 转换为 SomeType。

如果能够这样：

```
SomeType x = SomeType( y )
```

省去了中间步骤以后，类型转换变得更简洁了。这些无关紧要的中间步骤可以也应该被隐藏。

也许有些情况下这些中间步骤还是必要的，但是我觉得通过语言机制或定制方法来简化代码，抽象和隐藏那些细枝末节，有助于提高代码的可读性。

当然这些中间步骤仍然会发生在某处。通过隐藏这些细节，我们就可以专注于问题本身，这里是将变量 y 转换为 SomeType 类型。

虽然这并非是个十分恰当的隐式强制类型转换的例子，但我想说明的问题是，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

前面我们讲了字符串和数字之间的显式强制类型转换，现在介绍它们之间的隐式强制类型转换。先来看一些会产生隐式强制类型转换的操作。

通过重载，+ 运算符即能用于数字加法，也能用于字符串拼接。JavaScript 怎样来判断我们要执行的是哪个操作？例如：

```
var a = "42";
var b = "0";

var c = 42;
var d = 0;

a + b; // "420"
c + d; // 42
```

这里为什么会得到 "420" 和 42 两个不同的结果呢？通常的理解是，因为某一个或者两个操作数都是字符串，所以 + 执行的是字符串拼接操作。这样解释只对了一半，实际情况要复杂得多。

例如：

```
var a = [1,2];
var b = [3,4];

a + b; // "1,23,4"
```

a 和 b 都不是字符串，但是它们都被强制转换为字符串然后进行拼接。原因何在？

下面两段内容与规范有关，如果太难理解可以跳过。

根据 ES5 规范 11.6.1 节，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（规范 9.1 节），该抽象操作再调用 [[DefaultValue]]（规范 8.12.8 节），以数字作为上下文。

你或许注意到这与 ToNumber 抽象操作处理对象的方式一样（参见 4.2.2 节）。因为数组的 valueOf() 操作无法得到简单基本类型值，于是它转而调用 toString()。因此上例中的两个数组变成了 "1,2" 和 "3,4"。+ 将它们拼接后返回 "1,23,4"。

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。

有一个坑常常被提到，即 [] + {} 和 {} + []，它们返回不同的结果，分别是 "[object Object]" 和 0。我们将在 5.1.3 节详细介绍。

对隐式强制类型转换来说，这意味着什么？

我们可以将数字和空字符串 "" 相 + 来将其转换为字符串：

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

b; // "42"
```