目　录

致谢

当前文档 《你不懂JS：类型与文法（You Dont Know JS）》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-02-10。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/You-Dont-Know-JS-types-grammar

书栈官网：http://www.bookstack.cn

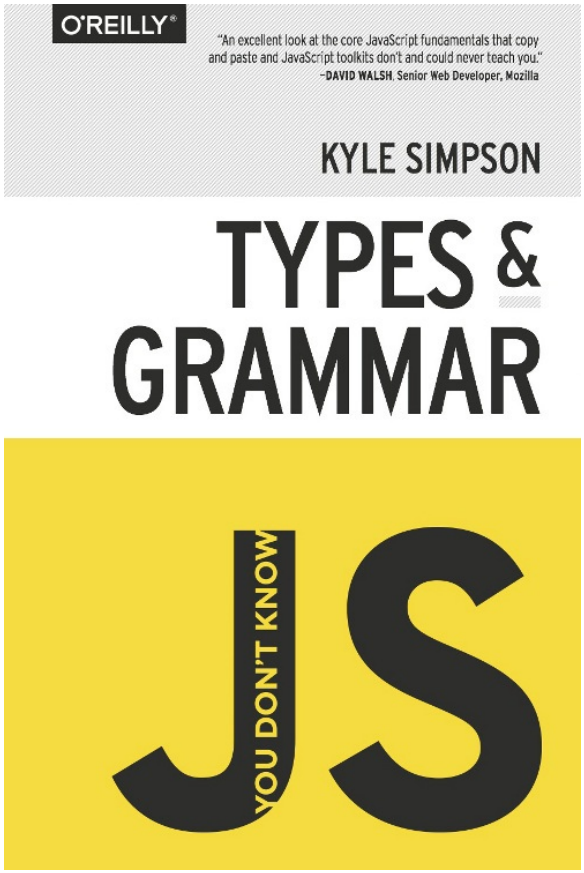
书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

阅前必读

- 你不懂JS：类型与文法

你不懂JS：类型与文法



从 **O'Reilly** 购买数字/印刷版

序

你不懂JS：类型与文法

序

人们曾说，“JavaScript 是唯一一种开发者在学会之前就使用的语言。”

我每次听到这句话都会笑出来，因为对我来说这是真的，而且我怀疑对于许多其他开发者也是。JavaScript，甚至可能还有 CSS 和 HTML，在因特网出现的早期都不是大学中教授的核心计算机语言，所以个人开发很大程度上都是基于开发者的搜索和“看源代码”的能力来将这些基本的 web 语言拼凑在一起。

我还记得我的第一个高中网站项目。它的任务是创建任意类型的网上商店，而我作为一个 James Bond 的粉丝，决定创建一个黄金眼商店。它有一切东西：黄金眼的迷笛主题音乐在背景中播放，一个用 JavaScript 制作的瞄准十字在屏幕上跟踪着鼠标，还有在每次点击时播放一次枪响的音效。Q 本应该会为这个网站中的杰作感到骄傲的。

我说这个故事是因为在那时我确实做了许多开发者今天在做的事情：我在我的项目中拷贝粘贴了大块儿的 JavaScript 代码，而根本不知道究竟发生了什么。像 jQuery 那样的工具包的广泛使用，以它们微不足道的方式，延续了这种不学习核心 JavaScript 的模式。

我不是在贬低 JavaScript 工具包的使用；毕竟，我还是 MooToo，ls JavaScript 团队的一员！但是 JavaScript 工具包如此强大的原因是因为它们的开发者了解基础，和它们的“坑”，并出色地施用了它们。和这些工具包的有用之处一样，了解这门语言的基础依然是极其重要的，而且有了 Kyle Simpson 的 你不懂*JS* 系列这样的书，没有借口不学习它们。

类型与文法，这个系列的第三部，是学习核心 JavaScript 基础的杰出教材，这些基础是拷贝粘贴和 JavaScript 工具包没有和绝不会教你的。强制转换和它的陷阱，原生类型与构造器，和 JavaScript 基本的全部方面都使用专门的代码示例进行了彻底地讲解。和这个系列的其他书籍一样，Kyle 总是一针见血：没有作秀和文字游戏 —— 这正是我喜爱类型的技术书籍。

享受类型与文法而且不要让它离你的桌子太远！

David Walsh

http://davidwalsh.name, @davidwalshblog

高级 Web 开发者，Mozilla

第一章：类型

第一章：类型

大多数开发者会说，动态语言（就像 JS）没有 类型。让我们看看 ES5.1 语言规范（http://www.ecma-international.org/ecma-262/5.1/）在这个问题上是怎么说的：

> 在本语言规范中的算法所操作的每一个值都有一种关联的类型。可能的值的类型就是那些在本条款中定义的类型。类型还进一步被分为 *ECMAScript* 语言类型和语言规范类型
>
> 一个 *ECMAScript* 语言类型对应于 *ECMAScript* 程序员使用 *ECMAScript* 语言直接操作的值。*ECMAScript* 语言类型有 *Undefined*, *Null*, *Boolean*, *String*, *Number*, 和 *Object*。

现在，如果你是一个强类型（静态类型的）语言的爱好者，你可能会反对“类型”一词的用法。在那些语言中，“类型”的含义要比它在 JS 这里的含义丰富得 多。

有些人说 JS 不应该声称拥有“类型”，它们应被称为“标签”或者“子类型”。

去他的！我们将使用这个粗糙的定义（看起来和语言规范的定义相同，只是改变了措辞）：一个 类型是一组固有的，内建的性质，对于引擎 和开发者 来说，它独一无二地标识了一个特定的值的行为，并将它与其他值区分开。

换句话说，如果引擎和开发者看待值 `42`（数字）与看待值 `"42"`（字符串）的方式不同，那么这两个值就拥有不同的 类型 — 分别是 `number` 和 `string`。当你使用 `42` 时，你就在 试图 做一些数字的事情，比如计算。但当你使用 `"42"` 时，你就在 试图 做一些字符串的事情，比如输出到页面上，等等。这两个值有着不同的类型。

这绝不是一个完美的定义。但是对于这里的讨论足够好了。而且它与 JS 描述它的方式并不矛盾。

链接

类型的重要意义

抛开学术上关于定义的分歧，为什么 JavaScript 有或者没有 类型 那么重要？

对每一种 类型 和它的固有行为有一个正确的理解，对于理解如何正确和准确地转换两个不同类型的值来说是绝对必要的（参见第四章，强制转换）。几乎每一个被编写过的 JS 程序都需要以某种形式处理类型的强制转换，所以，你能负责任、有信心地这么做是很重要的。

如果你有一个 `number` 值 `42`，但你想像一个 `string` 那样对待它，比如从位置 `1` 中将 `"2"` 作为一个字符抽取出来，那么显然你需要首先将值从 `number` （强制）转换成一个 `string`。

这看起来十分简单。

但是这样的强制转换可能以许多不同的方式发生。其中有些方式是明确的，很容易推理的，和可靠的。但是如果你不小心，强制转换就可能以非常奇怪的，令人吃惊的方式发生。

对强制转换的困惑可能是 JavaScript 开发者所经历的最深刻的挫败感之一。它曾经总是因为如此危险 而为人所诟病，被认为是一个语言设计上的缺陷而应当被回避。

带着对 JavaScript 类型的全面理解，我们将要阐明为什么强制转换的 坏名声 是言过其实的，而且是有些冤枉的 — 以此来反转你的视角，来看清强制转换的力量和用处。但首先，我们必须更好地把握值与类型。

内建类型

内建类型

JavaScript 定义了七种内建类型：

- `null`
- `undefined`
- `boolean`
- `number`
- `string`
- `object`
- `symbol` – 在 ES6 中被加入的！

注意： 除了 `object` 所有这些类型都被称为“基本类型（primitives）”。

`typeof` 操作符可以检测给定值的类型，而且总是返回七种字符串值中的一种 – 令人吃惊的是，对于我们刚刚列出的七中内建类型，它没有一个恰好的一对一匹配。

```
typeof undefined     === "undefined"; // true
typeof true          === "boolean";   // true
typeof 42            === "number";    // true
typeof "42"          === "string";    // true
typeof { life: 42 }  === "object";    // true

// 在 ES6 中被加入的！
typeof Symbol()      === "symbol";    // true
```

如上所示，这六种列出来的类型拥有相应类型的值，并返回一个与类型名称相同的字符串值。 `Symbol` 是 ES6 的新数据类型，我们将在第三章中讨论它。

正如你可能已经注意到的，我在上面的列表中剔除了 `null` 。它是 特殊的 – 特殊在它与 `typeof` 操作符组合时是有 bug 的。

```
typeof null === "object"; // true
```

要是它返回 `"null"` 就好了（而且是正确的！），但是这个原有的 bug 已经存在了近二十年，而且好像永远也不会被修复了，因为有太多已经存在的 web 的内容依存着这个 bug 的行为，“修复”这个 bug 将会 制造 更多的“bug”并毁掉许多 web 软件。

如果你想要使用 `null` 类型来测试 `null` 值，你需要一个复合条件：

```
var a = null;

(!a && typeof a === "object"); // true
```

`null` 是唯一一个“falsy”（也叫类 false；见第四章），但是在 `typeof` 检查中返回 `"object"` 的基本类型。

那么 `typeof` 可以返回的第七种字符串值是什么？

```
typeof function a(){ /* .. */ } === "function"; // true
```

很容易认为在 JS 中 `function` 是一种顶层的内建类型，特别是看到 `typeof` 操作符的这种行为时。然而，如果你阅读语言规范，你会看到它实际上是对象（object）的“子类型”。特别地，一个函数（function）被称为“可调用对象” —— 一个拥有 `[[Call]]` 内部属性、允许被调用的对象。

函数实际上是对象这一事实十分有用。最重要的是，它们可以拥有属性。例如：

```
function a(b,c) {
    /* .. */
}
```

这个函数对象拥有一个 `length` 属性，它被设置为函数被声明时的形式参数的数量。

```
a.length; // 2
```

因为你使用了两个正式命名的参数（ `b` 和 `c` ）声明了函数，所以“函数的长度”是 `2` 。

那么数组呢？它们是 JS 原生的，所以它们是一个特殊的类型咯？

```
typeof [1,2,3] === "object"; // true
```

不，它们仅仅是对象。考虑它们的最恰当的方法是，也将它们认为是对象的“子类型”（见第三章），带有被数字索引的附加性质（与仅仅使用字符串键的普通对象相反），并维护着一个自动更新的 `.length` 属性。

值作为类型

值作为类型

在 JavaScript 中，变量没有类型 – 值才有类型。变量可以在任何时候，持有任何值。

另一种考虑 JS 类型的方式是，JS 没有“类型强制”，也就是引擎不坚持认为一个 变量 总是持有与它开始存在时相同的 初始类型 的值。在一个赋值语句中，一个变量可以持有一个 `string`，而在下一个赋值语句中持有一个 `nubmer`，如此类推。

值 `42` 有固有的类型 `number`，而且它的 类型 是不能被改变的。另一个值，比如 `string` 类型的 `"42"`，可以通过一个称为 强制转换 的处理从 `number` 类型的值 `42` 中创建出来（见第四章）。

如果你对一个变量使用 `typeof`，它不会像表面上看起来那样询问“这个变量的类型是什么？”，因为 JS 变量是没有类型的。取而代之的是，它会询问“在这个变量里的值的类型是什么？”

```
1. var a = 42;
2. typeof a; // "number"
3.
4. a = true;
5. typeof a; // "boolean"
```

`typeof` 操作符总是返回字符串。所以：

```
1. typeof typeof 42; // "string"
```

第一个 `typeof 42` 返回 `"number"`，而 `typeof "number"` 是 `"string"`。

`undefined` vs “undeclared”

当前 还不拥有值的变量，实际上拥有 `undefined` 值。对这样的变量调用 `typeof` 将会返回 `"undefined"`：

```
1. var a;
2.
3. typeof a; // "undefined"
4.
5. var b = 42;
6. var c;
7.
```

```
// 稍后
b = c;

typeof b; // "undefined"
typeof c; // "undefined"
```

大多数开发者考虑“undefined”这个词的方式会诱使他们认为它是“undeclared（未声明）”的同义词。然而在 JS 中，这两个概念十分不同。

一个“undefined”变量是在可访问的作用域中已经被声明过的，但是在 这个时刻 它里面没有任何值。相比之下，一个“undeclared”变量是在可访问的作用域中还没有被正式声明的。

考虑这段代码：

```
var a;

a; // undefined
b; // ReferenceError: b is not defined
```

一个恼人的困惑是浏览器给这种情形分配的错误消息。正如你所看到的，这个消息是“b is not defined”，这当然很容易而且很合理地使人将它与“b is undefined.”搞混。需要重申的是，“undefined”和“is not defined”是非常不同的东西。要是浏览器能告诉我们类似于“b is not found”或者“b is not declared”之类的东西就好了，那会减少这种困惑！

还有一种 `typeof` 与未声明变量关联的特殊行为，进一步增强了这种困惑。考虑这段代码：

```
var a;

typeof a; // "undefined"

typeof b; // "undefined"
```

`typeof` 操作符甚至为“undeclared”（或“not defined”）变量返回 `"undefined"`。要注意的是，当我们执行 `typeof b` 时，即使 `b` 是一个未声明变量，也不会有错误被抛出。这是 `typeof` 的一种特殊的安全防卫行为。

和上面类似地，要是 `typeof` 与未声明变量一起使用时返回“undeclared”就好了，而不是将其结果值与不同的“undefined”情况混为一谈。

`typeof` Undeclared

不管怎样，当在浏览器中处理 JavaScript 时这种安全防卫是一种有用的特性，因为浏览器中多个脚本文件会将变量加载到共享的全局名称空间。

注意：许多开发者相信，在全局名称空间中绝不应该有任何变量，而且所有东西应当被包含在模块和私有/隔离的名称空间中。这在理论上很伟大但在实践中几乎是不可能的；但它仍然是一个值得的努力方向！幸运的是，ES6 为模块加入了头等支持，这终于使这一理论变得可行的多了。

作为一个简单的例子，想象在你的程序中有一个“调试模式”，它是通过一个称为 `DEBUG` 的全局变量（标志）来控制的。在实施类似于在控制台上输出一条日志消息这样的调试任务之前，你想要检查这个变量是否被声明了。一个顶层的全局 `var DEBUG = true` 声明只包含在一个“debug.js”文件中，这个文件仅在你开发/测试时才被加载到浏览器中，而在生产环境中则不会。

然而，在你其他的程序代码中，你不得不小心你是如何检查这个全局的 `DEBUG` 变量的，这样你才不会抛出一个 `ReferenceError`。这种情况下 `typeof` 上的安全防卫就是我们的朋友。

```
// 噢，这将抛出一个错误！
if (DEBUG) {
    console.log( "Debugging is starting" );
}

// 这是一个安全的存在性检查
if (typeof DEBUG !== "undefined") {
    console.log( "Debugging is starting" );
}
```

即便你不是在对付用户定义的变量（比如 `DEBUG`），这种检查也是很有用的。如果你为一个内建的 API 做特性检查，你也会发现不抛出错误的检查很有帮助：

```
if (typeof atob === "undefined") {
    atob = function() { /*..*/ };
}
```

注意：如果你在为一个还不存在的特性定义一个“填补”，你可能想要避免使用 `var` 来声明 `atob`。如果你在 `if` 语句内部声明 `var atob`，即使这个 `if` 条件没有通过（因为全局的 `atob` 已经存在），这个声明也会被提升（参见本系列的 作用域与闭包）到作用域的顶端。在某些浏览器中，对一些特殊类型的内建全局变量（常被称为“宿主对象”），这种重复声明也许会抛出错误。忽略 `var` 可以防止这种提升声明。

另一种不带有 `typeof` 的安全防卫特性，而对全局变量进行这些检查的方法是，将所有的全局变量作为全局对象的属性来观察，在浏览器中这个全局对象基本上是 `window` 对象。所以，上面的检查可以（十分安全地）这样做：

```
if (window.DEBUG) {
    // ..
}

if (!window.atob) {
```

```
    // ..
}
```

和引用未声明变量不同的是，在你试着访问一个不存在的对象属性时（即便是在全局的 `window` 对象上），不会有 `ReferenceError` 被抛出。

另一方面，一些开发者偏好避免手动使用 `window` 引用全局变量，特别是当你的代码需要运行在多种 JS 环境中时（例如不仅是在浏览器中，还在服务器端的 node.js 中），全局变量可能不总是称为 `window`。

技术上讲，这种 `typeof` 上的安全防卫即使在你不使用全局变量时也很有用，虽然这些情况不那么常见，而且一些开发者也许发现这种设计方式不那么理想。想象一个你想要其他人复制-粘贴到他们程序中或模块中的工具函数，在它里面你想要检查包含它的程序是否已经定义了一个特定的变量（以便于你可以使用它）：

```
function doSomethingCool() {
    var helper =
        (typeof FeatureXYZ !== "undefined") ?
        FeatureXYZ :
        function() { /*.. 默认的特性 ..*/ };

    var val = helper();
    // ..
}
```

`doSomethingCool()` 对称为 `FeatureXYZ` 变量进行检查，如果找到，就使用它，如果没找到，使用它自己的。现在，如果某个人在他的模块/程序中引入了这个工具，它会安全地检查我们是否已经定义了 `FeatureXYZ`：

```
// 一个 IIFE（参见本系列的 *作用域与闭包* 中的“立即被调用的函数表达式”）
(function(){
    function FeatureXYZ() { /*.. my XYZ feature ..*/ }

    // 引入 `doSomethingCool(..)`
    function doSomethingCool() {
        var helper =
            (typeof FeatureXYZ !== "undefined") ?
            FeatureXYZ :
            function() { /*.. 默认的特性 ..*/ };

        var val = helper();
        // ..
    }

    doSomethingCool();
```

```
})();
```

这里，`FeatureXYZ` 根本不是一个全局变量，但我们仍然使用 `typeof` 的安全防卫来使检查变得安全。而且重要的是，我们在这里 没有 可以用于检查的对象（就像我们使用 `window.___` 对全局变量做的那样），所以 `typeof` 十分有帮助。

另一些开发者偏好一种称为“依赖注入”的设计模式，与 `doSomethingCool()` 隐含地检查 `FeatureXYZ` 是否在它外部/周围被定义过不同的是，它需要依赖明确地传递进来，就像这样：

```
function doSomethingCool(FeatureXYZ) {
    var helper = FeatureXYZ ||
        function() { /*.. 默认的特性 ..*/ };

    var val = helper();
    // ..
}
```

在设计这样的功能时有许多选择。这些模式里没有“正确”或“错误” — 每种方式都有各种权衡。但总的来说，`typeof` 的未声明安全防卫给了我们更多选项，这还是很不错的。

复习

复习

JavaScript 有七种内建 类型：`null`、`undefined`、`boolean`、`number`、`string`、`object`、`symbol`。它们可以被 `typeof` 操作符识别。

变量没有类型，但是值有类型。这些类型定义了值的固有行为。

许多开发者会认为“undefined”和“undeclared”大体上是同一个东西，但是在 JavaScript 中，它们是十分不同的。`undefined` 是一个可以由被声明的变量持有的值。“未声明”意味着一个变量从来没有被声明过。

JavaScript 很不幸地将这两个词在某种程度上混为了一谈，不仅体现在它的错误消息上（“ReferenceError: a is not defined”），也体现在 `typeof` 的返回值上：对于两者它都返回 `"undefined"`。

然而，当对一个未声明的变量使用 `typeof` 时，`typeof` 上的安全防卫机制（防止一个错误）可以在特定的情况下非常有用。

第二章：值

第二章：值

`array`、`string`、和 `number` 是任何程序的最基础构建块，但是 JavaScript 在这些类型上有一些或使你惊喜或使你惊讶的独特性质。

让我们来看几种 JS 内建的值类型，并探讨一下我们如何才能更加全面地理解并正确地利用它们的行为。

链接

- Arrays
- Strings
- Numbers
- 特殊值
- 值与引用
- 复习

Arrays

Array

和其他强制类型的语言相比，JavaScript 的 `array` 只是值的容器，而这些值可以是任何类型：`string` 或者 `number` 或者 `object`，甚至是另一个 `array`（这也是你得到多维数组的方法）。

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

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

你不需要预先指定 `array` 的大小，你可以仅声明它们并加入你觉得合适的值：

```
var a = [ ];

a.length;    // 0

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

a.length;    // 3
```

警告： 在一个 `array` 值上使用 `delete` 将会从这个 `array` 上移除一个值槽，但就算你移除了最后一个元素，它也 不会 更新 `length` 属性，所以多加小心！我们会在第五章讨论 `delete` 操作符的更多细节。

要小心创建“稀散”的 `array`（留下或创建空的/丢失的值槽）：

```
var a = [ ];

a[0] = 1;
// 这里没有设置值槽 `a[1]`
a[2] = [ 3 ];

a[1];        // undefined

a.length;    // 3
```

虽然这可以工作，但你留下的“空值槽”可能会导致一些令人困惑的行为。虽然这样的值槽看起来拥有 `undefined` 值，但是它不会像被明确设置（ `a[1] = undefined` ）的值槽那样动作。更多信息可以参见第三章的“Array”。

`array` 是被数字索引的（正如你所想的那样），但微妙的是它们也是对象，可以在它们上面添加 `string` 键/属性（但是这些属性不会计算在 `array` 的 `length` 中）：

```
var a = [ ];

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

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

然而，一个需要小心的坑是，如果一个可以被强制转换为10进制 `number` 的 `string` 值被用作键的话，它会认为你想使用 `number` 索引而不是一个 `string` 键！

```
var a = [ ];

a["13"] = 42;

a.length; // 14
```

一般来说，向 `array` 添加 `string` 键/属性不是一个好主意。最好使用 `object` 来持有键/属性形式的值，而将 `array` 专用于严格地数字索引的值。

类 Array

偶尔你需要将一个类 `array` 值（一个数字索引的值的集合）转换为一个真正的 `array`，通常你可以对这些值的集合调用数组的工具函数（比如 `indexOf(..)`、`concat(..)`、`forEach(..)` 等等）。

举个例子，各种 DOM 查询操作会返回一个 DOM 元素的列表，对于我们转换的目的来说，这些列表不是真正的 `array` 但是也足够类似 `array`。另一个常见的例子是，函数为了像列表一样访问它的参数值，而暴露了 `arugumens` 对象（类 `array`，在 ES6 中被废弃了）。

一个进行这种转换的很常见的方法是对这个值借用 `slice(..)` 工具：

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

```
}

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

如果 `slice()` 没有用其他额外的参数调用，就像上面的代码段那样，它的参数的默认值会使它具有复制这个 `array` （或者，在这个例子中，是一个类 `array` ）的效果。

在 ES6 中，还有一种称为 `Array.from(..)` 的内建工具可以执行相同的任务：

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

注意： `Array.from(..)` 拥有其他几种强大的能力，我们将在本系列的 *ES6 与未来* 中涵盖它的细节。

Strings

String

一个很常见的想法是，`string` 实质上只是字符的 `array`。虽然内部的实现可能是也可能不是 `array`，但重要的是要理解 JavaScript 的 `string` 与字符的 `array` 确实不一样。它们的相似性几乎只是表面上的。

举个例子，让我们考虑这两个值：

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

String 确实与 `array` 有很肤浅的相似性 — 也就是上面说的，类 `array` — 举例来说，它们都有一个 `length` 属性，一个 `indexOf(..)` 方法（在 ES5 中仅有 `array` 版本），和一个 `concat(..)` 方法：

```
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"]
```

那么，它们基本上都仅仅是“字符的数组”，对吧？ 不确切：

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

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

JavaScript 的 `string` 是不可变的，而 `array` 是相当可变的。另外，在 JavaScript 中

用位置访问字符的 `a[1]` 形式不总是广泛合法的。老版本的 IE 就不允许这种语法（但是它们现在允许了）。相反，正确的方式是 `a.charAt(1)`。

`string` 不可变性的进一步的后果是，`string` 上没有一个方法是可以原地修改它的内容的，而是创建并返回一个新的 `string`。与之相对的是，许多改变 `array` 内容的方法实际上是原地修改的。

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

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

另外，许多 `array` 方法在处理 `string` 时非常有用，虽然这些方法不属于 `string`，但我们可以对我们的 `string` “借用”非变化的 `array` 方法：

```
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."
```

让我们来看另一个例子：翻转一个 `string`（顺带一提，这是一个 JavaScript 面试中常见的细节问题！）。`array` 拥有一个原地的 `reverse()` 修改器方法，但是 `string` 没有：

```
a.reverse;        // undefined

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

不幸的是，这种“借用” `array` 修改器不起作用，因为 `string` 是不可变的，因此它不能被原地修改：

```
Array.prototype.reverse.call( a );
// 仍然返回一个“foo”的 String 对象包装器（见第三章） :(
```

另一种迂回的做法（也是黑科技）是，将 `string` 转换为一个 `array`，实施我们想做的操作，

然后将它转回 `string` 。

```
var c = a
    // 将 `a` 切分成一个字符的数组
    .split( "" )
    // 翻转字符的数组
    .reverse()
    // 将字符的数组连接回一个字符串
    .join( "" );

c; // "oof"
```

如果你觉得这很难看，没错。不管怎样，对于简单的 `string` 它 好用，所以如果你需要某些快速但是“脏”的东西，像这样的方式经常能满足你。

警告： 小心！这种方法对含有复杂（unicode）字符（星型字符、多字节字符等）的 `string` 不起作用。你需要支持 unicode 的更精巧的工具库来准确地处理这种操作。在这个问题上可以咨询 Mathias Bynens 的作品：*Esrever*（https://github.com/mathiasbynens/esrever）。

另外一种考虑这个问题的方式是：如果你更经常地将你的“string”基本上作为 字符的数组 来执行一些任务的话，也许就将它们作为 `array` 而不是作为 `string` 存储更好。你可能会因此省去很多每次都将 `string` 转换为 `array` 的麻烦。无论何时你确实需要 `string` 的表现形式的话，你总是可以调用 字符的 `array` 的 `join("")` 方法。

Numbers

Number

JavaScript 只有一种数字类型：`number`。这种类型包含“整数”值和小数值。我说“整数”时加了引号，因为 JS 的一个长久以来为人诟病的原因是，和其他语言不同，JS 没有真正的整数。这可能在未来某个时候会改变，但是目前，我们只有 `number` 可用。

所以，在 JS 中，一个“整数”只是一个没有小数部分的小数值。也就是说，`42.0` 和 `42` 一样是“整数”。

像大多数现代计算机语言，以及几乎所有的脚本语言一样，JavaScript 的 `number` 的实现基于“IEEE 754”标准，通常被称为“浮点”。JavaScript 明确地使用了这个标准的“双精度”（也就是“64位二进制”）格式。

在网络上有许多了不起的文章都在介绍二进制浮点数如何在内存中存储的细节，以及选择这些做法的意义。因为对于理解如何在 JS 中正确使用 `number` 来说，理解内存中的位模式不是必须的，所以我们将这个话题作为练习留给那些想要进一步挖掘 IEEE 754 的细节的读者。

数字的语法

在 JavaScript 中字面数字一般用十进制小数表达。例如：

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

小数的整数部分如果是 `0`，是可选的：

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

相似地，一个小数在 `.` 之后的小数部分如果是 `0`，是可选的：

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

警告：`42.` 是极不常见的，如果你正在努力避免别人阅读你的代码时感到困惑，它可能不是一个好主意。但不管怎样，它是合法的。

默认情况下，大多数 `number` 将会以十进制小数的形式输出，并去掉末尾小数部分的 `0`。所

以：

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

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

非常大或非常小的 `number` 将默认以指数形式输出，与 `toExponential()` 方法的输出一样，比如：

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

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

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

因为 `number` 值可以用 `Number` 对象包装器封装（见第三章），所以 `number` 值可以访问内建在 `Number.prototype` 上的方法（见第三章）。举个例子，`toFixed(..)` 方法允许你指定一个值在被表示时，带有多少位小数：

```
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"
```

要注意的是，它的输出实际上是一个 `number` 的 `string` 表现形式，而且如果你指定的位数多于值持有的小数位数时，会在右侧补 `0`。

`toPrecision(..)` 很相似，但它指定的是有多少 有效数字 用来表示这个值：

```
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"
```

你不必非得使用持有这个值的变量来访问这些方法；你可以直接在 `number` 的字面上访问这些方法。但你不得不小心 `.` 操作符。因为 `.` 是一个合法数字字符，如果可能的话，它会首先被翻译为 `number` 字面的一部分，而不是被翻译为属性访问操作符。

```
// 不合法的语法：
42.toFixed( 3 );    // SyntaxError

// 这些都是合法的：
(42).toFixed( 3 );    // "42.000"
0.42.toFixed( 3 );    // "0.420"
42..toFixed( 3 );    // "42.000"
```

`42.toFixed(3)` 是不合法的语法，因为 `.` 作为 `42.` 字面（这是合法的 — 参见上面的讨论！）的一部分被吞掉了，因此没有 `.` 属性操作符来表示 `.toFixed` 访问。

`42..toFixed(3)` 可以工作，因为第一个 `.` 是 `number` 的一部分，而第二个 `.` 是属性操作符。但它可能看起来很古怪，而且确实在实际的 JavaScript 代码中很少会看到这样的东西。实际上，在任何基本类型上直接访问方法是十分不常见的。但是不常见并不意味着 坏 或者 错。

注意： 有一些库扩展了内建的 `Number.prototype`（见第三章），使用 `number` 或在 `number` 上提供了额外的操作，所以在这些情况下，像使用 `10..makeItRain()` 来设定一个十秒钟的下钱雨的动画，或者其他诸如此类的傻事是完全合法的。

在技术上讲，这也是合法的（注意那个空格）：

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

但是，尤其是对 `number` 字面量来说，这是特别使人糊涂的代码风格，而且除了使其他开发者（和未来的你）糊涂以外没有任何用处。避免它。

`number` 还可以使用科学计数法的形式指定，这在表示很大的 `number` 时很常见，比如：

```
var onethousand = 1E3;                    // 代表 1 * 10^3
var onemilliononehundredthousand = 1.1E6;    // 代表 1.1 * 10^6
```

`number` 字面量还可以使用其他进制表达，比如二进制，八进制，和十六进制。

这些格式是可以在当前版本的 JavaScript 中使用的：

```
0xf3; // 十六进制的: 243
0Xf3; // 同上

```

```
0363; // 八进制的: 243
```

注意： 从 ES6 + `strict` 模式开始，不再允许 `0363` 这样的的八进制形式（新的形式参见后面的讨论）。 `0363` 在非 `strict` 模式下依然是允许的，但是不管怎样你应当停止使用它，来拥抱未来（而且因为你现在应当在使用 `strict` 模式了！）。

至于 ES6，下面的新形式也是合法的：

```
0o363;        // 八进制的: 243
0O363;        // 同上

0b11110011;    // 二进制的: 243
0B11110011; // 同上
```

请为你的开发者同胞们做件好事：绝不要使用 `0O363` 形式。把 `0` 放在大写的 `O` 旁边就是在制造困惑。保持使用小写的谓词 `0x` 、 `0b` 、和 `0o` 。

小数值

使用二进制浮点数的最出名（臭名昭著）的副作用是（记住，这是对 所有 使用 IEEE 754 的语言都成立的 —— 不是许多人认为/假装 仅 在 JavaScript 中存在的问题）：

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

从数学的意义上，我们知道这个语句应当为 `true` 。为什么它是 `false` ？

简单地说， `0.1` 和 `0.2` 的二进制表示形式是不精确的，所以它们相加时，结果不是精确地 `0.3` 。而是 非常 接近的值： `0.30000000000000004` ，但是如果你的比较失败了，“接近”是无关紧要的。

注意： JavaScript 应当切换到可以精确表达所有值的一个不同的 `number` 实现吗？有些人认为应该。多年以来有许多选项出现过。但是没有一个被采纳，而且也许永远也不会。它看起来就像挥挥手然后说“已经改好那个 bug 了!”那么简单，但根本不是那么回事儿。如果真有这么简单，它绝对在很久以前就被改掉了。

现在的问题是，如果一些 `number` 不能被 信任 为精确的，这不是意味着我们根本不能使用 `number` 吗？ 当然不是。

在一些应用程序中你需要多加小心，特别是在对付小数的时候。还有许多（也许是大多数？）应用程序只处理整数，而且，最大只处理到几百万到几万亿。这些应用程序使用 JS 中的数字操作是，而且将总是，非常安全 的。

要是我们 确实 需要比较两个 `number` ，就像 `0.1 + 0.2` 与 `0.3` ，而且知道这个简单的相等

测试将会失败呢？

可以接受的最常见的做法是使用一个很小的“错误舍入”值作为比较的 容差。这个很小的值经常被称为“机械极小值（machine epsilon）”，对于 JavaScript 来说这种 `number` 通常为 `2^-52` （ `2.220446049250313e-16` ）。

在 ES6 中，使用这个容差值预定义了 `Number.EPSILON` ，所以你将会想要使用它，你也可以在前 ES6 中安全地填补这个定义：

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

我们可以使用这个 `Number.EPSILON` 来比较两个 `number` 的“等价性”（带有错误舍入的容差）：

```
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
```

可以被表示的最大的浮点值大概是 `1.798e+308` （它真的非常，非常，非常大！），它为你预定义为 `Number.MAX_VALUE` 。在极小的一端， `Number.MIN_VALUE` 大概是 `5e-324` ，它不是负数但是非常接近于0！

安全整数范围

由于 `number` 的表示方式，对完全是 `number` 的“整数”而言有一个“安全”的值的范围，而且它要比 `Number.MAX_VALUE` 小得多。

可以“安全地”被表示的最大整数（也就是说，可以保证被表示的值是实际可以无误地表示的）是 `2^53 - 1` ，也就是 `9007199254740991` ，如果你插入一些数字分隔符，可以看到它刚好超过9万亿。所以对于 `number` 能表示的上限来说它确实是够TM大的。

在ES6中这个值实际上是自动预定义的，它是 `Number.MAX_SAFE_INTEGER` 。意料之中的是，还有一个最小值， `-9007199254740991` ，它在ES6中定义为 `Number.MIN_SAFE_INTEGER` 。

JS 程序面临处理这样大的数字的主要情况是，处理数据库中的64位 ID 等等。64位数字不能使用 `number` 类型准确表达，所以在 JavaScript 中必须使用 `string` 表现形式存储（和传递）。

谢天谢地，在这样的大 ID `number` 值上的数字操作（除了比较，它使用 `string` 也没问题）并不很常见。但是如果你 确实 需要在这些非常大的值上实施数学操作，目前来讲你需要使用一个大数字 工具。在未来版本的 JavaScript 中，大数字也许会得到官方支持。

测试整数

测试一个值是否是整数，你可以使用 ES6 定义的 `Number.isInteger(..)`：

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

可以为前 ES6 填补 `Number.isInteger(..)`：

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

要测试一个值是否是 安全整数，使用 ES6 定义的 `Number.isSafeInteger(..)`：

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

可以为前 ES6 浏览器填补 `Number.isSafeInteger(..)`：

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

32位（有符号）整数

虽然整数可以安全地最大达到约九万亿（53比特），但有一些数字操作（比如位操作符）是仅仅为32位 `number` 定义的，所以对于被这样使用的 `number` 来说，“安全范围”一定会小得多。

这个范围是从 `Math.pow(-2,31)`（`-2147483648`，大约-21亿）到 `Math.pow(2,31)-1`（`2147483647`，大约+21亿）。

要强制 `a` 中的 `number` 值是32位有符号整数，使用 `a | 0`。这可以工作是因为 `|` 位操作符仅仅对32位值起作用（意味着它可以只关注32位，而其他的位将被丢掉）。而且，和 `0` 进行“或”的位操作实质上是什么也不做。

注意： 特定的特殊值（我们将在下一节讨论），比如 `NaN` 和 `Infinity` 不是“32位安全”的，当这些值被传入位操作符时将会通过一个抽象操作 `ToInt32`（见第四章）并为了位操作而简单地变成 `+0` 值。

特殊值

特殊值

在各种类型中散布着一些特殊值，需要 警惕 的 JS 开发者小心，并正确使用。

不是值的值

对于 `undefined` 类型来说，有且仅有一个值： `undefined` 。对于 `null` 类型来说，有且仅有一个值： `null` 。所以对它们而言，这些文字既是它们的类型也是它们的值。

`undefined` 和 `null` 作为“空”值或者“没有”值，经常被认为是可以互换的。另一些开发者偏好于使用微妙的区别将它们区分开。举例来讲：

- `null` 是一个空值
- `undefined` 是一个丢失的值

或者：

- `undefined` 还没有值
- `null` 曾经有过值但现在没有

不管你选择如何“定义”和使用这两个值， `null` 是一个特殊的关键字，不是一个标识符，因此你不能将它作为一个变量对待来给它赋值（为什么你要给它赋值呢？！）。然而， `undefined` （不幸地）是 一个标识符。噢。

Undefined

在非 `strict` 模式下，给在全局上提供的 `undefined` 标识符赋一个值实际上是可能的（虽然这是一个非常不好的做法！）：

```
function foo() {
    undefined = 2; // 非常差劲儿的主意！
}

foo();
```

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

```
5.
6. foo();
```

但是，在非 `strict` 模式和 `strict` 模式下，你可以创建一个名叫 `undefined` 局部变量。但这又是一个很差劲儿的主意！

```
1. function foo() {
2.     "use strict";
3.     var undefined = 2;
4.     console.log( undefined ); // 2
5. }
6.
7. foo();
```

朋友永远不让朋友覆盖 **`undefined`** 。

`void` 操作符

虽然 `undefined` 是一个持有内建的值 `undefined` 的内建标识符（除非被修改 —— 见上面的讨论！），另一个得到这个值的方法是 `void` 操作符。

表达式 `void __` 会“躲开”任何值，所以这个表达式的结果总是值 `undefined` 。它不会修改任何已经存在的值；只是确保不会有值从操作符表达式中返回来。

```
1. var a = 42;
2.
3. console.log( void a, a ); // undefined 42
```

从惯例上讲（大约是从 C 语言编程中发展而来），要通过使用 `void` 来独立表现值 `undefined` ，你可以使用 `void 0` （虽然，很明显， `void true` 或者任何其他的 `void` 表达式都做同样的事情）。在 `void 0` 、 `void 1` 和 `undefined` 之间没有实际上的区别。

但是在几种其他的环境下 `void` 操作符可以十分有用：如果你需要确保一个表达式没有结果值（即便它有副作用）。

举个例子：

```
1. function doSomething() {
2.     // 注意：`APP.ready` 是由我们的应用程序提供的
3.     if (!APP.ready) {
4.         // 稍后再试一次
5.         return void setTimeout( doSomething, 100 );
6.     }
7.
8.     var result;
```

```
9.
10.     // 做其他一些事情
11.     return result;
12. }
13.
14. // 我们能立即执行吗？
15. if (doSomething()) {
16.     // 马上处理其他任务
17. }
```

这里，`setTimeout(..)` 函数返回一个数字值（时间间隔定时器的唯一标识符，用于取消它自己），但是我们想 `void` 它，这样我们函数的返回值不会在 `if` 语句上给出一个成立的误报。

许多开发者宁愿将这些动作分开，这样的效用相同但不使用 `void` 操作符：

```
1. if (!APP.ready) {
2.     // 稍后再试一次
3.     setTimeout( doSomething, 100 );
4.     return;
5. }
```

一般来说，如果有那么一个地方，有一个值存在（来自某个表达式）而你发现这个值如果是 `undefined` 才有用，就使用 `void` 操作符。这可能在你的程序中不是非常常见，但如果在一些稀有的情况下你需要它，它就十分有用。

特殊的数字

`number` 类型包含几种特殊值。我们将会仔细考察每一种。

不是数字的数字

如果你不使用同为 `number`（或者可以被翻译为十进制或十六进制的普通 `number` 的值）的两个操作数进行任何算数操作，那么操作的结果将失败而产生一个不合法的 `number`，在这种情况下你将得到 `NaN` 值。

`NaN` 在字面上代表“不是一个 `number`（Not a Number）”，但是正如我们即将看到的，这种文字描述十分失败而且容易误导人。将 `NaN` 考虑为“不合法数字”，“失败的数字”，甚至是“坏掉的数字”都要比“不是一个数字”准确得多。

举例来说：

```
1. var a = 2 / "foo";        // NaN
2.
3. typeof a === "number";    // true
```

换句话说："'不是一个数字'的类型是'数字'"！为这使人糊涂的名字和语义欢呼吧。

`NaN` 是一种"哨兵值"（一个被赋予了特殊意义的普通的值），它代表 `number` 集合内的一种特殊的错误情况。这种错误情况实质上是："我试着进行数学操作但是失败了，而这就是失败的 `number` 结果。"

那么，如果你有一个值存在某个变量中，而且你想要测试它是否是这个特殊的失败数字 `NaN`，你也许认为你可以直接将它与 `NaN` 本身比较，就像你能对其它的值做的那样，比如 `null` 和 `undefined`。不是这样。

```
1. var a = 2 / "foo";
2.
3. a == NaN;    // false
4. a === NaN;    // false
```

`NaN` 是一个非常特殊的值，它从来不会等于另一个 `NaN` 值（也就是，它从来不等于它自己）。实际上，它是唯一一个不具有反射性的值（没有恒等性 `x === x`）。所以，`NaN !== NaN`。有点奇怪，对吧？

那么，如果不能与 `NaN` 进行比较（因为这种比较将总是失败），我们该如何测试它呢？

```
1. var a = 2 / "foo";
2.
3. isNaN( a ); // true
```

够简单的吧？我们使用称为 `isNaN(..)` 的内建全局工具，它告诉我们这个值是否是 `NaN`。问题解决了！

别高兴得太早。

`isNaN(..)` 工具有一个重大缺陷。它似乎过于按照字面的意思（"不是一个数字"）去理解 `NaN` 的含义了 —— 它的工作基本上是："测试这个传进来的东西是否不是一个 `number` 或者是一个 `number`"。但这不是十分准确。

```
1. var a = 2 / "foo";
2. var b = "foo";
3.
4. a; // NaN
5. b; // "foo"
6.
7. window.isNaN( a ); // true
8. window.isNaN( b ); // true -- 噢!
```

很明显，`"foo"` 根本不是一个 *number*，但它也绝不是一个 `NaN` 值！这个 bug 从最开始

的时候就存在于 JS 中了（存在超过了十九年的坑）。

在 ES6 中，终于提供了一个替代它的工具：`Number.isNaN(..)`。有一个简单的填补，可以让你即使是在前 ES6 的浏览器中安全地检查 `NaN` 值：

```
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 -- 咻!
```

实际上，通过利用 `NaN` 与它自己不相等这个特殊的事实，我们可以更简单地实现 `Number.isNaN(..)` 的填补。在整个语言中 `NaN` 是唯一一个这样的值；其他的值都总是 等于它自己。

所以：

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

怪吧？但是好用！

不管有意还是无意，在许多真实世界的 JS 程序中 `NaN` 可能是一个现实的问题。使用 `Number.isNaN(..)`（或者它的填补）这样的可靠测试来正确地识别它们是一个非常好的主意。

如果你正在程序中仅使用 `isNaN(..)`，悲惨的现实是你的程序 有 *bug*，即便是你还没有被它咬到！

无穷

来自于像 C 这样的传统编译型语言的开发者，可能习惯于看到编译器错误或者是运行时异常，比如对这样一个操作给出的“除数为 0”：

```
var a = 1 / 0;
```

然而在 JS 中，这个操作是明确定义的，而且它的结果是值 `Infinity`（也就是 `Number.POSITIVE_INFINITY`）。意料之中的是：

```
var a = 1 / 0;    // Infinity
var b = -1 / 0;    // -Infinity
```

如你所见，`-Infinity`（也就是 `Number.NEGATIVE_INFINITY`）是从任一个被除数为负（不是两个都是负数！）的除 0 操作得来的。

JS 使用有限的数字表现形式（IEEE 754 浮点，我们早先讨论过），所以和单纯的数学相比，它看起来甚至在做加法和减法这样的操作时都有可能溢出，这样的情况下你将会得到 `Infinity` 或 `-Infinity`。

例如：

```
var a = Number.MAX_VALUE;    // 1.7976931348623157e+308
a + a;                       // Infinity
a + Math.pow( 2, 970 );      // Infinity
a + Math.pow( 2, 969 );      // 1.7976931348623157e+308
```

根据语言规范，如果一个像加法这样的操作得到一个太大而不能表示的值，IEEE 754 “就近舍入”模式将会指明结果应该是什么。所以粗略的意义上，`Number.MAX_VALUE + Math.pow( 2, 969 )` 比起 `Infinity` 更接近于 `Number.MAX_VALUE`，所以它“向下舍入”，而 `Number.MAX_VALUE + Math.pow( 2, 970 )` 距离 `Infinity` 更近，所以它“向上舍入”。

如果你对此考虑的太多，它会使你头疼的。所以别想了。我是认真的，停！

一旦你溢出了任意一个 无限值，那么，就没有回头路了。换句最有诗意的话说，你可以从有限迈向无限，但不能从无限回归有限。

“无限除以无限等于什么”，这简直是一个哲学问题。我们幼稚的大脑可能会说“1”或“无限”。事实表明它们都不对。在数学上和在 JavaScript 中，`Infinity / Infinity` 不是一个有定义的操作。在 JS 中，它的结果为 `NaN`。

一个有限的正 `number` 除以 `Infinity` 呢？简单！`0`。那一个有限的负 `number` 处理 `Infinity` 呢？接着往下读！

零

虽然这可能使有数学头脑的读者困惑，但 JavaScript 拥有普通的零 `0`（也称为正零 `+0`）和 一个负零 `-0`。在我们讲解为什么 `-0` 存在之前，我们应该考察 JS 如何处理它，因为它

可能十分令人困惑。

除了使用字面量 `-0` 指定，负的零还可以从特定的数学操作中得出。比如：

```
var a = 0 / -3; // -0
var b = 0 * -3; // -0
```

加法和减法无法得出负零。

在开发者控制台中考察一个负的零，经常显示为 `-0`，然而直到最近这才是一个常见情况，所以一些你可能遇到的老版本浏览器也许依然将它报告为 `0`。

但是根据语言规范，如果你试着将一个负零转换为字符串，它将总会被报告为 `"0"`。

```
var a = 0 / -3;

// 至少（有些浏览器）控制台是对的
a;                              // -0

// 但是语言规范坚持要向你撒谎！
a.toString();                // "0"
a + "";                        // "0"
String( a );                // "0"

// 奇怪的是，就连 JSON 也加入了骗局之中
JSON.stringify( a );        // "0"
```

有趣的是，反向操作（从 `string` 到 `number`）不会撒谎：

```
+"-0";                // -0
Number( "-0" );        // -0
JSON.parse( "-0" );    // -0
```

警告：当你观察的时候，`JSON.stringify( -0 )` 产生 `"0"` 显得特别奇怪，因为它与反向操作不符：`JSON.parse( "-0" )` 将像你期望地那样报告 `-0`。

除了一个负零的字符串化会欺骗性地隐藏它实际的值外，比较操作符也被设定为（有意地）要说谎。

```
var a = 0;
var b = 0 / -3;

a == b;        // true
-0 == 0;    // true

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

```
-0 === 0;    // true

0 > -0;        // false
a > b;        // false
```

很明显，如果你想在你的代码中区分 `-0` 和 `0`，你就不能仅依靠开发者控制台的输出，你必须更聪明一些：

```
function isNegZero(n) {
    n = Number( n );
    return (n === 0) && (1 / n === -Infinity);
}

isNegZero( -0 );        // true
isNegZero( 0 / -3 );    // true
isNegZero( 0 );            // false
```

那么，除了学院派的细节以外，我们为什么需要一个负零呢？

在一些应用程序中，开发者使用值的大小来表示一部分信息（比如动画中每一帧的速度），而这个 `number` 的符号来表示另一部分信息（比如移动的方向）。

在这些应用程序中，举例来说，如果一个变量的值变成了 0，而它丢失了符号，那么你就丢失了它是从哪个方向移动到 0 的信息。保留零的符号避免了潜在的意外信息丢失。

特殊等价

正如我们上面看到的，当使用等价性比较时，值 `NaN` 和值 `-0` 拥有特殊的行为。`NaN` 永远不会和自己相等，所以你不得不使用 ES6 的 `Number.isNaN(..)`（或者它的填补）。相似地，`-0` 撒谎并假装它和普通的正零相等（即使使用 `===` 严格等价 —— 见第四章），所以你不得不使用我们上面建议的某些 `isNegZero(..)` 黑科技工具。

在 ES6 中，有一个新工具可以用于测试两个值的绝对等价性，而没有任何这些例外。它称为 `Object.is(..)`：

```
var a = 2 / "foo";
var b = -3 * 0;

Object.is( a, NaN );    // true
Object.is( b, -0 );        // true

Object.is( b, 0 );        // false
```

对于前 ES6 环境，这是一个相当简单的 `Object.is(..)` 填补：

```
if (!Object.is) {
    Object.is = function(v1, v2) {
        // 测试 `-0`
        if (v1 === 0 && v2 === 0) {
            return 1 / v1 === 1 / v2;
        }
        // 测试 `NaN`
        if (v1 !== v1) {
            return v2 !== v2;
        }
        // 其他情况
        return v1 === v2;
    };
}
```

`Object.is(..)` 可能不应当用于那些 `==` 或 `===` 已知 安全 的情况（见第四章“强制转换”），因为这些操作符可能高效得多，并且更惯用/常见。 `Object.is(..)` 很大程度上是为这些特殊的等价情况准备的。

值与引用

值与引用

在其他许多语言中，根据你使用的语法，值可以通过值拷贝，也可以通过引用拷贝来赋予/传递。

比如，在 C++ 中如果你想要把一个 `number` 变量传递进一个函数，并使这个变量的值被更新，你可以用 `int& myNum` 这样的东西来声明函数参数，当你传入一个变量 `x` 时，`myNum` 将是一个 指向 `x` 的引用；引用就像一个特殊形式的指针，你得到的是一个指向另一个变量的指针（像一个 别名（*alias*）） 。如果你没有声明一个引用参数，被传入的值将 总是 被拷贝的，就算它是一个复杂的对象。

在 JavaScript 中，没有指针，并且引用的工作方式有一点儿不同。你不能拥有一个从一个 JS 变量到另一个 JS 变量的引用。这是完全不可能的。

JS 中的引用指向一个（共享的） 值，所以如果你有十个不同的引用，它们都总是同一个共享值的不同引用；它们没有一个是另一个的引用/指针。

另外，在 JavaScript 中，没有语法上的提示可以控制值和引用的赋值/传递。取而代之的是，值的 类型 用来 唯一 控制值是通过值拷贝，还是引用拷贝来赋予。

让我们来展示一下：

```
var a = 2;
var b = a; // `b` 总是 `a` 中的值的拷贝
b++;
a; // 2
b; // 3

var c = [1,2,3];
var d = c; // `d` 是共享值 `[1,2,3]` 的引用
d.push( 4 );
c; // [1,2,3,4]
d; // [1,2,3,4]
```

简单值（也叫基本标量） 总是 通过值拷贝来赋予/传递：`null`、`undefined`、`string`、`number`、`boolean`、以及 ES6 的 `symbol`。

复合值 —— `object`（包括 `array`，和所有的对象包装器 —— 见第三章）和 `function` —— 总是 在赋值或传递时创建一个引用的拷贝。

在上面的代码段中，因为 `2` 是一个基本标量，`a` 持有一个这个值的初始拷贝，而 `b` 被赋

予了这个值的另一个拷贝。当改变 `b` 时，你根本没有在改变 `a` 中的值。

但 `c` 和 `d` 两个都 是同一个共享的值 `[1,2,3]` 的分离的引用。重要的是， `c` 和 `d` 对值 `[1,2,3]` 的“拥有”程度上是一样的 —— 它们只是同一个值的对等引用。所以，不管使用哪一个引用去修改（ `.push(4)` ）实际上共享的 `array` 值本身，影响的仅仅是这一个共享值，而且这两个引用将会指向新修改的值 `[1,2,3,4]` 。

因为引用指向的是值本身而不是变量，你不能使用一个引用来改变另一个引用所指向的值：

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

// 稍后
b = [4,5,6];
a; // [1,2,3]
b; // [4,5,6]
```

当我们做赋值操作 `b = [4,5,6]` 时，我们做的事情绝对不会对 `a` 所指向的 位置（ `[1,2,3]` ）造成任何影响。如果那可能的话， `b` 就会是 `a` 的指针而不是这个 `array` 的引用 —— 但是这样的能力在 JS 中是不存在的！

这样的困惑最常见于函数参数：

```
function foo(x) {
    x.push( 4 );
    x; // [1,2,3,4]

    // 稍后
    x = [4,5,6];
    x.push( 7 );
    x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );

a; // [1,2,3,4] 不是 [4,5,6,7]
```

当我们传入参数 `a` 时，它将一份 `a` 引用的拷贝赋值给 `x` 。 `x` 和 `a` 是指向相同的 `[1,2,3]` 的不同引用。现在，在函数内部，我们可以使用这个引用来改变值本身（ `push(4)` ）。但是当我们进行赋值操作 `x = [4,5,6]` 时，不可能影响原来的引用 `a` 所指向的东西 —— 它仍然指向（已经被修改了的）值 `[1,2,3,4]` 。

没有办法可以使用 `x` 引用来改变 `a` 指向哪里。我们只能修改 `a` 和 `x` 共通指向的那个共享值的内容。

要想改变 `a` 来使它拥有内容为 `[4,5,6,7]` 的值，你不能创建一个新的 `array` 并赋值 —— 你必须修改现存的 `array` 值：

```
function foo(x) {
    x.push( 4 );
    x; // [1,2,3,4]

    // 稍后
    x.length = 0; // 原地清空既存的数组
    x.push( 4, 5, 6, 7 );
    x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );

a; // [4,5,6,7] 不是 [1,2,3,4]
```

正如你看到的，`x.length = 0` 和 `x.push(4,5,6,7)` 没有创建一个新的 `array`，但是修改了现存的共享 `array`。所以理所当然地，`a` 引用了新的内容 `[4,5,6,7]`。

记住：你不能直接控制/覆盖值拷贝和引用拷贝的行为 —— 这些语义是完全由当前值的类型来控制的。

为了实质上地通过值拷贝传递一个复合值（比如一个 `array`），你需要手动制造一个它的拷贝，使被传递的引用不指向原来的值。比如：

```
foo( a.slice() );
```

不带参数的 `slice(..)` 方法默认地为这个 `array` 制造一个全新的（浅）拷贝。所以，我们传入的引用仅指向拷贝的 `array`，这样 `foo(..)` 不会影响 `a` 的内容。

反之 —— 传递一个基本标量值，使它的值的变化可见，就像引用那样 —— 你不得不将这个值包装在另一个可以通过引用拷贝来传递的复合值中（`object`、`array` 等等）：

```
function foo(wrapper) {
    wrapper.a = 42;
}

var obj = {
    a: 2
```

```
};

foo( obj );

obj.a; // 42
```

这里，`obj` 作为基本标量属性 `a` 的包装。当传递给 `foo(..)` 时，一个 `obj` 引用的拷贝被传入并设置给 `wrapper` 参数。我们现在可以使用 `wrapper` 引用来访问这个共享的对象，并更新它的值。在函数完成时，`obj.a` 将被更新为值 `42`。

你可能会遇到这样的情况，如果你想要传入一个像 `2` 这样的基本标量值的引用，你可以将这个值包装在它的 `Number` 对象包装器中（见第三章）。

这个 `Number` 对象的引用的拷贝 将 会被传递给函数是事实，但不幸的是，和你可能期望的不同，拥有一个共享独享的引用不会给你修改这个共享的基本值的能力：

```
function foo(x) {
    x = x + 1;
    x; // 3
}

var a = 2;
var b = new Number( a ); // 或等价的 `Object(a)`

foo( b );
console.log( b ); // 2, 不是 3
```

这里的问题是，底层的基本标量值是 不可变的（`String` 和 `Boolean` 也一样）。如果一个 `Number` 对象持有一个基本标量值 `2`，那么这个 `Number` 对象就永远不能再持有另一个值；你只能用一个不同的值创建一个全新的 `Number` 对象。

当 `x` 用于表达式 `x + 1` 时，底层的基本标量值 `2` 被自动地从 `Number` 对象中开箱（抽出），所以 `x = x + 1` 这一行很微妙地将 `x` 从一个共享的 `Number` 对象的引用，改变为仅持有加法操作 `2 + 1` 的结果 `3` 的基本标量值。因此，外面的 `b` 仍然引用原来的未被改变/不可变的，持有 `2` 的 `Number` 对象。

你 可以 在 `Number` 对象上添加属性（只是不要改变它内部的基本值），所以你可间接地通过这些额外的属性交换信息。

不过，这可不太常见；对大多数开发者来说这可能不是一个好的做法。

与其这样使用 `Number` 包装器对象，使用早先的代码段中那样的手动对象包装器（`obj`）要好得多。这不是说像 `Number` 这样包装好的对象包装器没有用处 —— 而是说在大多数情况下，你可能应该优先使用基本标量值的形式。

引用十分强大，但是有时候它们碍你的事儿，而有时你会在它们不存在时需要它们。你唯一可以用来控制引用与值拷贝的东西是值本身的类型，所以你必须通过你选用的值的类型来间接地影响赋值/传递行为。

复习

复习

在 JavaScript 中，`array` 仅仅是数字索引的集合，可以容纳任何类型的值。`string` 是某种“类 `array`”，但它们有着不同的行为，如果你想要将它们作为 `array` 对待的话，必须要小心。JavaScript 中的数字既包括“整数”也包括浮点数。

几种特殊值被定义在基本类型内部。

`null` 类型只有一个值 `null`，`undefined` 类型同样地只有 `undefined` 值。对于任何没有值存在的变量或属性，`undefined` 基本上是默认值。`void` 操作符允许你从任意另一个值中创建 `undefined` 值。

`number` 包含几种特殊值，比如 `NaN`（意为“不是一个数字”，但称为“非法数字”更合适）；`+Infinity` 和 `-Infinity`；还有 `-0`。

简单基本标量（`string`、`number` 等）通过值拷贝进行赋值/传递，而复合值（`object` 等）通过引用拷贝进行赋值/传递。引用与其他语言中的引用/指针不同 —— 它们从不指向其他的变量/引用，而仅指向底层的值。

第三章：原生类型

第三章：原生类型

在第一和第二章中，我们几次提到了各种内建类型，通常称为“原生类型”，比如 `String` 和 `Number`。现在让我们来仔细检视它们。

这是最常用的原生类型的一览：

- `String()`
- `Number()`
- `Boolean()`
- `Array()`
- `Object()`
- `Function()`
- `RegExp()`
- `Date()`
- `Error()`
- `Symbol()` —— 在 ES6 中被加入的！

如你所见，这些原生类型实际上是内建函数。

如果你拥有像 Java 语言那样的背景，JavaScript 的 `String()` 看起来像是你曾经用来创建字符串值的 `String(..)` 构造器。所以，你很快就会观察到你可以做这样的事情：

```
var s = new String( "Hello World!" );

console.log( s.toString() ); // "Hello World!"
```

这些原生类型的每一种确实可以被用作一个原生类型的构造器。但是被构建的东西可能与你想象的不同：

```
var a = new String( "abc" );

typeof a; // "object" ... 不是 "String"

a instanceof String; // true

Object.prototype.toString.call( a ); // "[object String]"
```

创建值的构造器形式（`new String("abc")`）的结果是一个基本类型值（`"abc"`）的包装器对象。

重要的是，`typeof` 显示这些对象不是它们自己的特殊 类型，而是 `object` 类型的子类型。

这个包装器对象可以被进一步观察，像这样：

```
1. console.log( a );
```

这个语句的输出会根据你使用的浏览器变化，因为对于开发者的查看，开发者控制台可以自由选择它认为合适的方式来序列化对象。

注意： 在写作本书时，最新版的 Chrome 打印出这样的东西：`String {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"}`。但是老版本的 Chrome 曾经只打印出这些：`String {0: "a", 1: "b", 2: "c"}`。当前最新版的 Firefox 打印 `String ["a","b","c"]`，但它曾经以斜体字打印 `"abc"`，点击它可以打开对象查看器。当然，这些结果是总频繁变更的，而且你的体验也许不同。

重点是，`new String("abc")` 为 `"abc"` 创建了一个字符串包装器对象，而不仅是基本类型值 `"abc"` 本身。

链接

- 内部 [[Class]]
- 封箱包装器
- 开箱
- 原生类型作为构造器
- 复习

内部 [[Class]]

内部 `[[Class]]`

`typeof` 的结果为 `"object"` 的值（比如数组）被额外地打上了一个内部的标签属性 `[[Class]]`（请把它考虑为一个内部的分类方法，而非与传统的面向对象编码的类有关）。这个属性不能直接地被访问，但通常可以间接地通过在这个值上借用默认的 `Object.prototype.toString(..)` 方法调用来展示。举例来说：

```
1. Object.prototype.toString.call( [1,2,3] );            // "[object Array]"
2.
3. Object.prototype.toString.call( /regex-literal/i );    // "[object RegExp]"
```

所以，对于这个例子中的数组来说，内部的 `[[Class]]` 值是 `"Array"`，而对于正则表达式，它是 `"RegExp"`。在大多数情况下，这个内部的 `[[Class]]` 值对应于关联这个值的内建的原生类型构造器（见下面的讨论），但事实却不总是这样。

基本类型呢？首先，`null` 和 `undefined`：

```
1. Object.prototype.toString.call( null );            // "[object Null]"
2. Object.prototype.toString.call( undefined );    // "[object Undefined]"
```

你会注意到，不存在 `Null()` 和 `Undefined()` 原生类型构造器，但不管怎样 `"Null"` 和 `"Undefined"` 是被暴露出来的内部 `[[Class]]` 值。

但是对于像 `string`、`number`、和 `boolean` 这样的简单基本类型，实际上会启动另一种行为，通常称为“封箱（boxing）”（见下一节“封箱包装器”）：

```
1. Object.prototype.toString.call( "abc" );    // "[object String]"
2. Object.prototype.toString.call( 42 );        // "[object Number]"
3. Object.prototype.toString.call( true );        // "[object Boolean]"
```

在这个代码段中，每一个简单基本类型都自动地被它们分别对应的对象包装器封箱，这就是为什么 `"String"`、`"Number"`、和 `"Boolean"` 分别被显示为内部 `[[Class]]` 值。

注意： 从 ES5 发展到 ES6 的过程中，这里展示的 `toString()` 和 `[[Class]]` 的行为发生了一点儿改变，但我们会在本系列的 *ES6* 与未来 一书中讲解它们的细节。

封箱包装器

封箱包装器

这些对象包装器服务于一个非常重要的目的。基本类型值没有属性或方法，所以为了访问 `.length` 或 `.toString()` 你需要这个值的对象包装器。值得庆幸的是，JS 将会自动地 封箱（也就是包装）基本类型值来满足这样的访问。

```
var a = "abc";

a.length; // 3
a.toUpperCase(); // "ABC"
```

那么，如果你想以通常的方式访问这些字符串值上的属性/方法，比如一个 `for` 循环的 `i < a.length` 条件，这么做看起来很有道理：一开始就得到一个这个值的对象形式，于是 JS 引擎就不需要隐含地为你创建一个。

但事实证明这是一个坏主意。浏览器们长久以来就对 `.length` 这样的常见情况进行性能优化，这意味着如果你试着直接使用对象形式（它们没有被优化过）进行“提前优化”，那么实际上你的程序将会 变慢。

一般来说，基本上没有理由直接使用对象形式。让封箱在需要的地方隐含地发生会更好。换句话说，永远也不要做 `new String("abc")` 、 `new Number(42)` 这样的事情 —— 应当总是偏向于使用基本类型字面量 `"abc"` 和 `42` 。

对象包装器的坑

如果你 确实 选择要直接使用对象包装器，那么有几个坑你应该注意。

举个例子，考虑 `Boolean` 包装的值：

```
var a = new Boolean( false );

if (!a) {
    console.log( "Oops" ); // 永远不会运行
}
```

这里的问题是，虽然你为值 `false` 创建了一个对象包装器，但是对象本身是“truthy”（见第四章），所以使用对象的效果是与使用底层的值 `false` 本身相反的，这与通常的期望十分不同。

如果你想手动封箱一个基本类型值，你可以使用 `Object(..)` 函数（没有 `new` 关键字）：

```
var a = "abc";
var b = new String( a );
var c = Object( a );

typeof a; // "string"
typeof b; // "object"
typeof c; // "object"

b instanceof String; // true
c instanceof String; // true

Object.prototype.toString.call( b ); // "[object String]"
Object.prototype.toString.call( c ); // "[object String]"
```

再说一遍，通常不鼓励直接使用封箱的包装器对象（比如上面的 `b` 和 `c`），但你可能会遇到一些它们有用的罕见情况。

开箱

开箱

如果你有一个包装器对象，而你想要取出底层的基本类型值，你可以使用 `valueOf()` 方法：

```
var a = new String( "abc" );
var b = new Number( 42 );
var c = new Boolean( true );

a.valueOf(); // "abc"
b.valueOf(); // 42
c.valueOf(); // true
```

当以一种查询基本类型值的方式使用对象包装器时，开箱也会隐含地发生。这个处理的过程（强制转换）将会在第四章中更详细地讲解，但简单地说：

```
var a = new String( "abc" );
var b = a + ""; // `b` 拥有开箱后的基本类型值"abc"

typeof a; // "object"
typeof b; // "string"
```

原生类型作为构造器

原生类型作为构造器

对于 `array` 、 `object` 、 `function` 和正则表达式值来说，使用字面形式来创建它们的值几乎总是更好的选择，而且字面形式与构造器形式所创建的值是同一种对象（也就是，没有非包装的值）。

正如我们刚刚在上面看到的其他原生类型，除非你真的知道你需要这些构造器形式，一般来说应当避免使用它们，这主要是因为它们会带来一些你可能不会想要对付的异常和陷阱。

`Array(..)`

```
var a = new Array( 1, 2, 3 );
a; // [1, 2, 3]

var b = [1, 2, 3];
b; // [1, 2, 3]
```

注意： `Array(..)` 构造器不要求在它前面使用 `new` 关键字。如果你省略它，它也会像你已经使用了一样动作。所以 `Array(1,2,3)` 和 `new Array(1,2,3)` 的结果是一样的。

`Array` 构造器有一种特殊形式，如果它仅仅被传入一个 `number` 参数，与将这个值作为数组的内容 不同，它会被认为是用来“预定数组大小”（嗯，某种意义上）用的长度。

这是个可怕的主意。首先，你会意外地用错这种形式，因为它很容易忘记。

但更重要的是，其实没有预定数组大小这样的东西。你所创建的是一个空数组，并将这个数组的 `length` 属性设置为那个指定的数字值。

一个数组在它的值槽上没有明确的值，但是有一个 `length` 属性意味着这些值槽是存在的，在 JS 中这是一个诡异的数据结构，它带有一些非常奇怪且令人困惑的行为。可以创建这样的值的能力，完全源自于老旧的、已经废弃的、仅具有历史意义的功能（比如 `arguments` 这样的“类数组对象”）。

注意： 带有至少一个“空值槽”的数组经常被称为“稀散数组”。

这是另外一个例子，展示浏览器的开发者控制台在如何表示这样的对象上有所不同，它产生了更多的困惑。

举例来说：

```
var a = new Array( 3 );

a.length; // 3
```

```
a;
```

在 Chrome 中 `a` 的序列化表达是（在本书写作时）：`[ undefined x 3 ]`。这真的很不幸。它暗示着在这个数组的值槽中有三个 `undefined` 值，而事实上这样的值槽是不存在的（所谓的“空值槽（empty slots）” —— 也是一个烂名字！）。

要观察这种不同，试试这段代码：

```
var a = new Array( 3 );
var b = [ undefined, undefined, undefined ];
var c = [];
c.length = 3;

a;
b;
c;
```

注意： 正如你在这个例子中看到的 `c`，数组中的空值槽可以在数组的创建之后发生。将数组的 `length` 改变为超过它实际定义的槽值的数目，你就隐含地引入了空值槽。事实上，你甚至可以在上面的代码段中调用 `delete b[1]`，而这么做将会在 `b` 的中间引入一个空值槽。

对于 `b`（在当前的 Chrome 中），你会发现它的序列化表现为 `[ undefined, undefined, undefined ]`，与之相对的是 `a` 和 `c` 的 `[ undefined x 3 ]`。糊涂了吧？是的，大家都糊涂了。

更糟糕的是，在写作本书时，Firefox 对 `a` 和 `c` 报告 `[ , , , ]`。你发现为什么这使人犯糊涂了吗？仔细看。三个逗号表示有四个值槽，不是我们期望的三个值槽。

什么！？ Firefox 在它们的序列化表达的末尾放了一个额外的 `,`，因为在 ES5 中，列表（数组值，属性列表等等）末尾的逗号是允许的（被砍掉并忽略）。所以如果你在你的程序或控制台中敲入 `[ , , , ]` 值，你实际上得到的是一个底层为 `[ , , ]` 的值（也就是，一个带有三个空值槽的数组）。这种选择，虽然在阅读开发者控制台时使人困惑，但是因为它使拷贝粘贴的时候准确，所以被留了下来。

如果你现在在摇头或翻白眼儿，你并不孤单！（耸肩）

不幸的是，事情越来越糟。比在控制台的输出产生的困惑更糟的是，上面代码段中的 `a` 和 `b` 实际上在有些情况下相同，但在另一些情况下不同：

```
a.join( "-" ); // "--"
b.join( "-" ); // "--"

a.map(function(v,i){ return i; }); // [ undefined x 3 ]
b.map(function(v,i){ return i; }); // [ 0, 1, 2 ]
```

呃。

`a.map(..)` 调用会 失败 是因为值槽根本就不实际存在，所以 `map(..)` 没有东西可以迭代。 `join(..)` 的工作方式不同，基本上我们可以认为它是像这样被实现的：

```
function fakeJoin(arr,connector) {
    var str = "";
    for (var i = 0; i < arr.length; i++) {
        if (i > 0) {
            str += connector;
        }
        if (arr[i] !== undefined) {
            str += arr[i];
        }
    }
    return str;
}

var a = new Array( 3 );
fakeJoin( a, "-" ); // "--"
```

如你所见， `join(..)` 好用仅仅是因为它 认为 值槽存在，并循环至 `length` 值。不管 `map(..)` 内部是在做什么，它（显然）没有做出这样的假设，所以源自于奇怪的“空值槽”数组的结果出人意料，而且好像是失败了。

那么，如果你想要 确实 创建一个实际的 `undefined` 值的数组（不只是“空值槽”），你如何才能做到呢（除了手动以外）？

```
var a = Array.apply( null, { length: 3 } );
a; // [ undefined, undefined, undefined ]
```

糊涂了吧？是的。这里是它大概的工作方式。

`apply(..)` 是一个对所有函数可用的工具方法，它以一种特殊方式调用这个使用它的函数。

第一个参数是一个 `this` 对象绑定（在本系列的 *this* 与对象原型 中有详细讲解），在这里我们不关心它，所以我们将它设置为 `null` 。第二个参数应该是一个数组（或 像 数组的东西 —— 也就是“类数组对象”）。这个“数组”的内容作为这个函数的参数“扩散”开来。

所以， `Array.apply(..)` 在调用 `Array(..)` 函数，并将一个值（ `{ length: 3 }` 对象值）作为它的参数值扩散开。

在 `apply(..)` 内部，我们可以预见这里有另一个 `for` 循环（有些像上面的 `join(..)` ），它从 `0` 开始上升但不包含至 `length` （这个例子中是 `3` ）。

对于每一个索引，它从对象中取得相应的键。所以如果这个数组对象参数在 `apply(..)` 内部被命名为 `arr`，那么这种属性访问实质上是 `arr[0]`、`arr[1]` 和 `arr[2]`。当然，没有一个属性是在 `{ length: 3 }` 对象值上存在的，所以这三个属性访问都将返回值 `undefined`。

换句话说，调用 `Array(..)` 的结局基本上是这样：`Array(undefined,undefined,undefined)`，这就是我们如何得到一个填满 `undefined` 值的数组的，而非仅仅是一些（疯狂的）空值槽。

虽然对于创建一个填满 `undefined` 值的数组来说，`Array.apply( null, { length: 3 } )` 是一个奇怪而且繁冗的方法，但是它要比使用砸自己的脚似的 `Array(3)` 空值槽要可靠和好得 太多了。

底线：你 在任何情况下，永远不，也不应该有意地创建并使用诡异的空值槽数组。就别这么干。它们是怪胎。

`Object(..)`、`Function(..)` 和 `RegExp(..)`

`Object(..)` / `Function(..)` / `RegExp(..)` 构造器一般来说也是可选的（因此除非是特别的目的，应当避免使用）：

```
var c = new Object();
c.foo = "bar";
c; // { foo: "bar" }

var d = { foo: "bar" };
d; // { foo: "bar" }

var e = new Function( "a", "return a * 2;" );
var f = function(a) { return a * 2; };
function g(a) { return a * 2; }

var h = new RegExp( "^a*b+", "g" );
var i = /^a*b+/g;
```

几乎没有理由使用 `new Object()` 构造器形式，尤其因为它强迫你一个一个地添加属性，而不是像对象的字面形式那样一次添加许多。

`Function` 构造器仅在最最罕见的情况下有用，也就是你需要动态地定义一个函数的参数和/或它的函数体。不要将 `Function(..)` 仅仅作为另一种形式的 `eval(..)`。你几乎永远不会需要用这种方式动态定义一个函数。

用字面量形式（`/^a*b+/g`）定义正则表达式是被大力采用的，不仅因为语法简单，而且还有性能的原因 —— JS 引擎会在代码执行前预编译并缓存它们。和我们迄今看到的其他构造器形式不同，`RegExp(..)` 有一些合理的用途：用来动态定义一个正则表达式的范例。

```
var name = "Kyle";
var namePattern = new RegExp( "\\b(?:" + name + ")+\\b", "ig" );
```

```
3.
4. var matches = someText.match( namePattern );
```

这样的场景在 JS 程序中一次又一次地合法出现，所以你有需要使用 `new RegExp("pattern","flags")` 形式。

`Date(..)` 和 `Error(..)`

`Date(..)` 和 `Error(..)` 原生类型构造器要比其他种类的原生类型有用得多，因为它们没有字面量形式。

要创建一个日期对象值，你必须使用 `new Date()` 。 `Date(..)` 构造器接收可选参数值来指定要使用的日期/时间，但是如果省略的话，就会使用当前的日期/时间。

目前你构建一个日期对象的最常见的理由是要得到当前的时间戳（一个有符号整数，从1970年1月1日开始算起的毫秒数）。你可以在一个日期对象实例上调用 `getTime()` 得到它。

但是在 ES5 中，一个更简单的方法是调用定义为 `Date.now()` 的静态帮助函数。而且在前 ES5 中填补它很容易：

```
1. if (!Date.now) {
2.     Date.now = function(){
3.         return (new Date()).getTime();
4.     };
5. }
```

注意： 如果你不带 `new` 调用 `Date()` ，你将会得到一个那个时刻的日期/时间的字符串表达。在语言规范中没有规定这个表达的确切形式，虽然各个浏览器趋向于赞同使用这样的东西： `"Fri Jul 18 2014 00:31:02 GMT-0500 (CDT)"` 。

`Error(..)` 构造器（很像上面的 `Array()` ）在有 `new` 与没有 `new` 时的行为是相同的。

你想要创建 error 对象的主要原因是，它会将当前的执行栈上下文捕捉进对象中（在大多数 JS 引擎中，在创建后使用只读的 `.stack` 属性表示）。这个栈上下文包含函数调用栈和 error 对象被创建时的行号，这使调试这个错误更简单。

典型地，你将与 `throw` 操作符一起使用这样的 error 对象：

```
1. function foo(x) {
2.     if (!x) {
3.         throw new Error( "x wasn't provided" );
4.     }
5.     // ..
6. }
```

`Error` 对象实例一般拥有至少一个 `message` 属性，有时还有其他属性（你应当将它们作为只读的），比如 `type`。然而，与其检视上面提到的 `stack` 属性，最好是在 `error` 对象上调用 `toString()`（明确地调用，或者是通过强制转换隐含地调用 —— 见第四章）来得到一个格式友好的错误消息。

提示： 技术上讲，除了一般的 `Error(..)` 原生类型以外，还有几种特定错误的原生类型：`EvalError(..)`、`RangeError(..)`、`ReferenceError(..)`、`SyntaxError(..)`、`TypeError(..)` 和 `URIError(..)`。但是手动使用这些特定错误原生类型十分少见。如果你的程序确实遭受了一个真实的异常，它们是会自动地被使用的（比如引用一个未声明的变量而得到一个 `ReferenceError` 错误）。

`Symbol(..)`

在 ES6 中，新增了一个基本值类型，称为“Symbol”。Symbol 是一种特殊的“独一无二”（不是严格保证的！）的值，可以作为对象上的属性使用而几乎不必担心任何冲突。它们主要是为特殊的 ES6 结构的内建行为设计的，但你也可以定义你自己的 symbol。

Symbol 可以用做属性名，但是你不能从你的程序中看到或访问一个 symbol 的实际值，从开发者控制台也不行。例如，如果你在开发者控制台中对一个 Symbol 求值，将会显示 `Symbol(Symbol.create)` 之类的东西。

在 ES6 中有几种预定义的 Symbol，做为 `Symbol` 函数对象的静态属性访问，比如 `Symbol.create`，`Symbol.iterator` 等等。要使用它们，可以这样做：

```
obj[Symbol.iterator] = function(){ /*..*/ };
```

要定义你自己的 Symbol，使用 `Symbol(..)` 原生类型。`Symbol(..)` 原生类型“构造器”很独特，因为它不允许你将 `new` 与它一起使用，这么做会抛出一个错误。

```
var mysym = Symbol( "my own symbol" );
mysym;                // Symbol(my own symbol)
mysym.toString();     // "Symbol(my own symbol)"
typeof mysym;         // "symbol"

var a = { };
a[mysym] = "foobar";

Object.getOwnPropertySymbols( a );
// [ Symbol(my own symbol) ]
```

虽然 Symbol 实际上不是私有的（在对象上使用 `Object.getOwnPropertySymbols(..)` 反射，揭示了 Symbol 其实是相当公开的），但是它们的主要用途可能是私有属性，或者类似的特殊属性。对于大多数开发者，他们也许会在属性名上加入 `_` 下划线前缀，这在经常在惯例上表示：“这是一个私有的/特殊的/内部的属性，别碰！”

注意： `Symbol` 不是 `object`，它们是简单的基本标量。

原生类型原型

每一个内建的原生构造器都拥有它自己的 `.prototype` 对象 —— `Array.prototype`，`String.prototype` 等等。

对于它们特定的对象子类型，这些对象含有独特的行为。

例如，所有的字符串对象，和 `string` 基本值的扩展（通过封箱），都可以访问在 `String.prototype` 对象上做为方法定义的默认行为。

注意： 做为文档惯例，`String.prototype.XYZ` 会被缩写为 `String#XYZ`，对于其它所有 `.prototype` 的属性都是如此。

- `String#indexOf(..)`：在一个字符串中找出一个子串的位置
- `String#charAt(..)`：访问一个字符串中某个位置的字符
- `String#substr(..)`、`String#substring(..)` 和 `String#slice(..)`：将字符串的一部分抽取为一个新字符串
- `String#toUpperCase()` 和 `String#toLowerCase()`：创建一个转换为大写或小写的新字符串
- `String#trim()`：创建一个截去开头或结尾空格的新字符串。

这些方法中没有一个是在 原地 修改字符串的。修改（比如大小写变换或去空格）会根据当前的值来创建一个新的值。

有赖于原型委托（见本系列的 *this* 与对象原型），任何字符串值都可以访问这些方法：

```
var a = " abc ";

a.indexOf( "c" ); // 3
a.toUpperCase(); // " ABC "
a.trim(); // "abc"
```

其他构造器的原型包含适用于它们类型的行为，比如 `Number#toFixed(..)`（将一个数字转换为一个固定小数位的字符串）和 `Array#concat(..)`（混合数组）。所有这些函数都可以访问 `apply(..)`、`call(..)` 和 `bind(..)`，因为 `Function.prototype` 定义了它们。

但是，一些原生类型的原型不 仅仅 是单纯的对象：

```
typeof Function.prototype;            // "function"
Function.prototype();                 // 它是一个空函数！

RegExp.prototype.toString();          // "/(?:)/" —— 空的正则表达式
"abc".match( RegExp.prototype );      // [""]
```

一个特别差劲儿的主意是，你甚至可以修改这些原生类型的原型（不仅仅是你可能熟悉的添加属性）：

```
Array.isArray( Array.prototype );    // true
Array.prototype.push( 1, 2, 3 );    // 3
Array.prototype;                    // [1,2,3]

// 别这么留着它，要不就等着怪事发生吧！
// 将`Array.prototype`重置为空
Array.prototype.length = 0;
```

如你所见，`Function.prototype` 是一个函数，`RegExp.prototype` 是一个正则表达式，而 `Array.prototype` 是一个数组。有趣吧？酷吧？

原型作为默认值

`Function.prototype` 是一个空函数，`RegExp.prototype` 是一个“空”正则表达式（也就是不匹配任何东西），而 `Array.prototype` 是一个空数组，这使它们成了可以赋值给变量的，很好的“默认”值 —— 如果这些类型的变量还没有值。

例如：

```
function isThisCool(vals,fn,rx) {
    vals = vals || Array.prototype;
    fn = fn || Function.prototype;
    rx = rx || RegExp.prototype;

    return rx.test(
        vals.map( fn ).join( "" )
    );
}

isThisCool();        // true

isThisCool(
    ["a","b","c"],
    function(v){ return v.toUpperCase(); },
    /D/
);                   // false
```

注意： 在 ES6 中，我们不再需要使用 `vals = vals || ..` 这样的默认值语法技巧了（见第四章），因为在函数声明中可以通过原生语法为参数设定默认值（见第五章）。

这个方式的一个微小的副作用是，`.prototype` 已经被创建了，而且是内建的，因此它仅被创建一次。相比之下，使用 `[]`、`function(){}` 和 `/(?:)/` 这些值本身作为默认值，将会（很可

能，要看引擎如何实现）在每次调用 `isThisCool(..)` 时重新创建这些值（而且稍可能要回收它们）。这可能会消耗内存/CPU。

另外，要非常小心不要对 后续要被修改的值 使用 `Array.prototype` 做为默认值。在这个例子中， `vals` 是只读的，但如果你要在原地对 `vals` 进行修改，那你实际上修改的是 `Array.prototype` 本身，这将把你引到刚才提到的坑里！

注意： 虽然我们指出了这些原生类型的原型和一些用处，但是依赖它们的时候要小心，更要小心以任何形式修改它们。更多的讨论见附录A“原生原型”。

复习

复习

JavaScript 为基本类型提供了对象包装器，被称为原生类型（ `String` 、 `Number` 、 `Boolean` 等等）。这些对象包装器使这些值可以访问每种对象子类型的恰当行为（ `String#trim()` 和 `Array#concat(..)` ）。

如果你有一个像 `"abc"` 这样的简单基本类型标量，而且你想要访问它的 `length` 属性或某些 `String.prototype` 方法，JS 会自动地“封箱”这个值（用它所对应种类的对象包装器把它包起来），以满足这样的属性/方法访问。

第四章：强制转换

你不懂JS：类型与文法

第四章：强制转换

现在我们更全面地了解了 JavaScript 的类型和值，我们将注意力转向一个极具争议的话题：强制转换。

正如我们在第一章中提到的，关于强制转换到底是一个有用的特性，还是一个语言设计上的缺陷（或介于两者之间！），早就开始就争论不休了。如果你读过关于 JS 的其他书籍，你就会知道流行在世面上那种淹没一切的 声音：强制转换是魔法，是邪恶的，令人困惑的，而且就是彻头彻尾的坏主意。

本着这个系列丛书的总体精神，我认为你应当直面你不理解的东西并设法更全面地 搞懂它。而不是因为大家都这样做，或是你曾经被一些怪东西咬到就逃避强制转换。

我们的目标是全面地探索强制转换的优点和缺点（是的，它们 有 优点！），这样你就能在程序中对它是否合适做出明智的决定。

链接

转换值

转换值

将一个值从一个类型明确地转换到另一个类型通常称为“类型转换（type casting）”，当这个操作隐含地完成时称为“强制转换（coercion）”（根据一个值如何被使用的规则来强制它变换类型）。

注意：这可能不明显，但是 JavaScript 强制转换总是得到基本标量值的一种，比如 `string`、`number`、或 `boolean`。没有强制转换可以得到像 `object` 和 `function` 这样的复杂值。第三章讲解了“封箱”，它将一个基本类型标量值包装在它们相应的 `object` 中，但在准确的意义上这不是真正的强制转换。

另一种区别这些术语的常见方法是：“类型转换（type casting/conversion）”发生在静态类型语言的编译时，而“类型强制转换（type coercion）”是动态类型语言的运行时转换。

然而，在 JavaScript 中，大多数人将所有这些类型的转换都称为 强制转换（*coercion*），所以我偏好的区别方式是使用“隐含强制转换（implicit coercion）”与“明确强制转换（explicit coercion）”。

其中的区别应当是很明显的：在观察代码时如果一个类型转换明显是有意为之的，那么它就是“明确强制转换”，而如果这个类型转换是做为其他操作的不那么明显的副作用发生的，那么它就是“隐含强制转换”。

例如，考虑这两种强制转换的方式：

```
1. var a = 42;
2.
3. var b = a + "";           // 隐含强制转换
4.
5. var c = String( a );    // 明确强制转换
```

对于 `b` 来说，强制转换是隐含地发生的，因为如果与 `+` 操作符组合的操作数之一是一个 `string` 值（`""`），这将使 `+` 操作成为一个 `string` 连接（将两个字符串加在一起），而 `string` 连接的一个（隐藏的）副作用将 `a` 中的值 `42` 强制转换为它的 `string` 等价物：`"42"`。

相比之下，`String(..)` 函数使一切相当明显，它明确地取得 `a` 中的值，并把它强制转换为一个 `string` 表现形式。

两种方式都能达到相同的效果：从 `42` 变成 `"42"`。但它们如何达到这种效果，才是关于 JavaScript 强制转换的热烈争论的核心。

注意： 技术上讲，这里有一些在语法形式区别之上的，行为上的微妙区别。我们将在本章稍后，“隐含：Strings <–> Numbers”一节中仔细讲解。

“明确地”、“隐含地”、或“明显地”和“隐藏的副作用”这些术语，是 相对的。

如果你确切地知道 `a + ""` 是在做什么，并且你有意地这么做来强制转换一个 `string`，你可能感觉这个操作已经足够“明确”了。相反，如果你从没见过 `String(..)` 函数被用于 `string` 强制转换，那么对你来说它的行为可能看起来太过隐蔽而让你感到“隐含”。

但我们是基于一个 大众的，充分了解，但不是专家或 *JS* 规范爱好者的 开发者的观点来讨论“明确”与“隐含”的。无论你的程度如何，或是没有在这个范畴内准确地找到自己，你都需要根据我们在这里的观察方式，相应地调整你的角度。

记住：我们自己写代码而也只有我们自己会读它，通常是很少见的。即便你是一个精通 JS 里里外外的专家，也要考虑一个经验没那么丰富的队友在读你的代码时感受如何。对于他们和对于你来说，“明确”或“隐含”的意义相同吗？

抽象值操作

抽象值操作

在我们可以探究 明确 与 隐含 强制转换之前，我们需要学习一些基本规则，是它们控制着值如何变成 一个 `string` 、 `number` 、或 `boolean` 的。ES5 语言规范的第九部分用值的变形规则定义了几种“抽象操作”（“仅供内部使用的操作”的高大上说法）。我们将特别关注于： `ToString` 、 `ToNumber` 、和 `ToBoolean` ，并稍稍关注一下 `ToPrimitive` 。

`ToString`

当任何一个非 `string` 值被强制转换为一个 `string` 表现形式时，这个转换的过程是由语言规范的 9.8 部分的 `ToString` 抽象操作处理的。

内建的基本类型值拥有自然的字符串化形式： `null` 变为 `"null"` ， `undefined` 变为 `"undefined"` ， `true` 变为 `"true"` 。 `number` 一般会以你期望的自然方式表达，但正如我们在第二章中讨论的，非常小或非常大的 `number` 将会以指数形式表达：

```
// `1.07`乘以`1000`, 7次
var a = 1.07 * 1000 * 1000 * 1000 * 1000 * 1000 * 1000 * 1000;

// 7次乘以3位 => 21位
a.toString(); // "1.07e21"
```

对于普通的对象，除非你指定你自己的，默认的 `toString()` （可以在 `Object.prototype.toString()` 找到）将返回 内部 `[[Class]]` （见第三章），例如 `"[object Object]"` 。

但正如早先所展示的，如果一个对象上拥有它自己的 `toString()` 方法，而你又以一种类似 `string` 的方式使用这个对象，那么它的 `toString()` 将会被自动调用，而且这个调用的 `string` 结果将被使用。

注意： 技术上讲，一个对象被强制转换为一个 `string` 要通过 `ToPrimitive` 抽象操作（ES5

语言规范，9.1 部分），但是那其中的微妙细节将会在本章稍后的 `ToNumber` 部分中讲解，所以我们在这里先跳过它。

数组拥有一个覆盖版本的默认 `toString()`，将数组字符串化为它所有的值（每个都字符串化）的（字符串）连接，并用 `","` 分割每个值。

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

a.toString(); // "1,2,3"
```

重申一次，`toString()` 可以明确地被调用，也可以通过在一个需要 `string` 的上下文环境中使用一个非 `string` 来自动地被调用。

JSON 字符串化

另一种看起来与 `ToString` 密切相关的操作是，使用 `JSON.stringify(..)` 工具将一个值序列化为一个 JSON 兼容的 `string` 值。

重要的是要注意，这种字符串化与强制转换并不完全是同一种东西。但是因为它与上面讲的 `ToString` 规则有关联，我们将在这里稍微转移一下话题，来讲解 JSON 字符串化行为。

对于最简单的值，JSON 字符串化行为基本上和 `toString()` 转换是相同的，除了序列化的结果总是一个 *`string`*：

```
JSON.stringify( 42 );    // "42"
JSON.stringify( "42" );    // ""42"" （一个包含双引号的字符串）
JSON.stringify( null );    // "null"
JSON.stringify( true );    // "true"
```

任何 *JSON* 安全 的值都可以被 `JSON.stringify(..)` 字符串化。但是什么是 *JSON* 安全的？任何可以用 JSON 表现形式合法表达的值。

考虑 JSON 不 安全的值可能更容易一些。一些例子是：`undefined`、`function`、（ES6+）`symbol`、和带有循环引用的 `object`（一个对象结构中的属性互相引用而造成了一个永不终结的循环）。对于标准的 JSON 结构来说这些都是非法的值，主要是因为它们不能移植到消费 JSON 值的其他语言中。

`JSON.stringify(..)` 工具在遇到 `undefined`、`function`、和 `symbol` 时将会自动地忽略它们。如果在一个 `array` 中遇到这样的值，它会被替换为 `null`（这样数组的位置信息就不会改变）。如果在一个 `object` 的属性中遇到这样的值，这个属性会被简单地剔除掉。

考虑下面的代码：

```
JSON.stringify( undefined );                    // undefined
```

```
JSON.stringify( function(){} );                  // undefined

JSON.stringify( [1,undefined,function(){},4] );    // "[1,null,null,4]"
JSON.stringify( { a:2, b:function(){} } );        // "{"a":2}"
```

但如果你试着 `JSON.stringify(..)` 一个带有循环引用的 `object`，就会抛出一个错误。

JSON 字符串化有一个特殊行为，如果一个 `object` 值定义了一个 `toJSON()` 方法，这个方法将会被首先调用，以取得用于序列化的值。

如果你打算 JSON 字符串化一个可能含有非法 JSON 值的对象，或者如果这个对象中正好有不适于序列化的值，那么你就应当为它定义一个 `toJSON()` 方法，返回这个 `object` 的一个 *JSON* 安全 版本。

例如：

```
var o = { };

var a = {
    b: 42,
    c: o,
    d: function(){}
};

// 在 `a` 内部制造一个循环引用
o.e = a;

// 这会因循环引用而抛出一个错误
// JSON.stringify( a );

// 自定义一个 JSON 值序列化
a.toJSON = function() {
    // 序列化仅包含属性 `b`
    return { b: this.b };
};

JSON.stringify( a ); // "{"b":42}"
```

一个很常见的误解是，`toJSON()` 应当返回一个 JSON 字符串化的表现形式。这可能是不正确的，除非你事实上想要字符串化 `string` 本身（通常不会！）。`toJSON()` 应当返回合适的实际普通值（无论什么类型），而 `JSON.stringify(..)` 自己会处理字符串化。

换句话说，`toJSON()` 应当被翻译为：“变为一个适用于字符串化的 JSON 安全的值”，而不是像许多开发者错误认为的那样，“变为一个 JSON 字符串”。

考虑下面的代码：

```
var a = {
    val: [1,2,3],

    // 可能正确！
    toJSON: function(){
        return this.val.slice( 1 );
    }
};

var b = {
    val: [1,2,3],

    // 可能不正确！
    toJSON: function(){
        return "[" +
            this.val.slice( 1 ).join() +
        "]";
    }
};

JSON.stringify( a ); // "[2,3]"

JSON.stringify( b ); // ""[2,3]""
```

在第二个调用中，我们字符串化了返回的 `string` 而不是 `array` 本身，这可能不是我们想要做的。

既然我们说到了 `JSON.stringify(..)`，那么就让我们来讨论一些不那么广为人知，但是仍然很有用的功能吧。

`JSON.stringify(..)` 的第二个参数值是可选的，它称为 替换器（*replacer*）。这个参数值既可以是一个 `array` 也可以是一个 `function`。与 `toJSON()` 为序列化准备一个值的方式类似，它提供一种过滤机制，指出一个 `object` 的哪一个属性应该或不应该被包含在序列化形式中，来自定义这个 `object` 的递归序列化行为。

如果 替换器 是一个 `array`，那么它应当是一个 `string` 的 `array`，它的每一个元素指定了允许被包含在这个 `object` 的序列化形式中的属性名称。如果一个属性不存在于这个列表中，那么它就会被跳过。

如果 替换器 是一个 `function`，那么它会为 `object` 本身而被调用一次，并且为这个 `object` 中的每个属性都被调用一次，而且每次都被传入两个参数值，*key* 和 *value*。要在序列化中跳过一个 *key*，可以返回 `undefined`。否则，就返回被提供的 *value*。

```
var a = {
    b: 42,
    c: "42",
```

```
    d: [1,2,3]
};

JSON.stringify( a, ["b","c"] ); // "{"b":42,"c":"42"}"

JSON.stringify( a, function(k,v){
    if (k !== "c") return v;
} );
// "{"b":42,"d":[1,2,3]}"
```

注意： 在 `function` 替换器 的情况下，第一次调用时 key 参数 `k` 是 `undefined`（而对象 `a` 本身会被传入）。 `if` 语句会 过滤掉 名称为 `c` 的属性。字符串化是递归的，所以数组 `[1,2,3]` 会将它的每一个值（ `1` 、 `2` 、和 `3` ）都作为 `v` 传递给 替换器，并将索引值（ `0` 、 `1` 、和 `2` ）作为 `k` 。

`JSON.stringify(..)` 还可以接收第三个可选参数值，称为 填充符（*space*），在对人类友好的输出中它被用做缩进。填充符 可以是一个正整数，用来指示每一级缩进中应当使用多少个空格字符。或者，填充符 可以是一个 `string` ，这时每一级缩进将会使用它的前十个字符。

```
var a = {
    b: 42,
    c: "42",
    d: [1,2,3]
};

JSON.stringify( a, null, 3 );
// "{
//    "b": 42,
//    "c": "42",
//    "d": [
//       1,
//       2,
//       3
//    ]
// }"

JSON.stringify( a, null, "-----" );
// "{
// -----"b": 42,
// -----"c": "42",
// -----"d": [
// ----------1,
// ----------2,
// ----------3
// -----]
// }"
```

记住，`JSON.stringify(..)` 并不直接是一种强制转换的形式。但是，我们在这里讨论它，是由于两个与 `ToString` 强制转换有关联的行为：

1. `string`、`number`、`boolean`、和 `null` 值在 JSON 字符串化时，与它们通过 `ToString` 抽象操作的规则强制转换为 `string` 值的方式基本上是相同的。
2. 如果传递一个 `object` 值给 `JSON.stringify(..)`，而这个 `object` 上拥有一个 `toJSON()` 方法，那么在字符串化之前，`toJSON()` 就会被自动调用来将这个值（某种意义上）“强制转换”为 *JSON 安全* 的。

`ToNumber`

如果任何非 `number` 值，以一种要求它是 `number` 的方式被使用，比如数学操作，就会发生 ES5 语言规范在 9.3 部分定义的 `ToNumber` 抽象操作。

例如，`true` 变为 `1` 而 `false` 变为 `0`。`undefined` 变为 `NaN`，而（奇怪的是）`null` 变为 `0`。

对于一个 `string` 值来说，`ToNumber` 工作起来很大程度上与数字字面量的规则/语法很相似（见第三章）。如果它失败了，结果将是 `NaN`（而不是 `number` 字面量中会出现的语法错误）。一个不同之处的例子是，在这个操作中 `0` 前缀的八进制数不会被作为八进制数来处理（而仅作为普通的十进制小数），虽然这样的八进制数作为 `number` 字面量是合法的。

注意：`number` 字面量文法与用于 `string` 值的 `ToNumber` 间的区别极其微妙，在这里就不进一步讲解了。更多的信息可以参考 ES 语言规范的 9.3.1 部分。

对象（以及数组）将会首先被转换为它们的基本类型值的等价物，而后这个结果值（如果它还不是一个 `number` 基本类型）会根据刚才提到的 `ToNumber` 规则被强制转换为一个 `number`。

为了转换为基本类型值的等价物，`ToPrimitive` 抽象操作（ES5 语言规范，9.1 部分）将会查询这个值（使用内部的 `DefaultValue` 操作 —— ES5 语言规范，8.12.8 部分），看它有没有 `valueOf()` 方法。如果 `valueOf()` 可用并且它返回一个基本类型值，那么 这个 值就将用于强制转换。如果不是这样，但 `toString()` 可用，那么就由它来提供用于强制转换的值。

如果这两种操作都没提供一个基本类型值，就会抛出一个 `TypeError`。

在 ES5 中，你可以创建这样一个不可强制转换的对象 —— 没有 `valueOf()` 和 `toString()` —— 如果它的 `[[Prototype]]` 的值为 `null`，这通常是通过 `Object.create(null)` 来创建的。关于 `[[Prototype]]` 的详细信息参见本系列的 *this* 与对象原型。

注意：我们会在本章稍后讲解如何强制转换至 `number`，但对于下面的代码段，想象 `Number(..)` 函数就是那样做的。

考虑如下代码：

```
1. var a = {
2.     valueOf: function(){
```

```
        return "42";
    }
};

var b = {
    toString: function(){
        return "42";
    }
};

var c = [4,2];
c.toString = function(){
    return this.join( "" );    // "42"
};

Number( a );            // 42
Number( b );            // 42
Number( c );            // 42
Number( "" );            // 0
Number( [] );            // 0
Number( [ "abc" ] );    // NaN
```

`ToBoolean`

下面，让我们聊一聊在 JS 中 `boolean` 如何动作。世面上关于这个话题有 许多的困惑和误解，所以集中注意力！

首先而且最重要的是，JS 实际上拥有 `true` 和 `false` 关键字，而且它们的行为正如你所期望的 `boolean` 值一样。一个常见的误解是，值 `1` 和 `0` 与 `true` / `false` 是相同的。虽然这可能在其他语言中是成立的，但在 JS 中 `number` 就是 `number`，而 `boolean` 就是 `boolean`。你可以将 `1` 强制转换为 `true`（或反之），或将 `0` 强制转换为 `false`（或反之）。但它们不是相同的。

Falsy 值

但这还不是故事的结尾。我们需要讨论一下，除了这两个 `boolean` 值以外，当你把其他值强制转换为它们的 `boolean` 等价物时如何动作。

所有的 JavaScript 值都可以被划分进两个类别：

1. 如果被强制转换为 `boolean`，将成为 `false` 的值
2. 其它的一切值（很明显将变为 `true`）

我不是在出洋相。JS 语言规范给那些在强制转换为 `boolean` 值时将会变为 `false` 的值定义了一个明确的，小范围的列表。

我们如何才能知道这个列表中的值是什么？在 ES5 语言规范中，9.2 部分定义了一个 `ToBoolean` 抽象操作，它讲述了对所有可能的值而言，当你试着强制转换它们为 boolean 时究竟会发生什么。

从这个表格中，我们得到了下面所谓的“falsy”值列表：

- `undefined`
- `null`
- `false`
- `+0`, `-0`, and `NaN`
- `""`

就是这些。如果一个值在这个列表中，它就是一个“falsy”值，而且当你在它上面进行 `boolean` 强制转换时它会转换为 `false`。

通过逻辑上的推论，如果一个值 不 在这个列表中，那么它一定在 另一个列表 中，也就是我们称为“truthy”值的列表。但是 JS 没有真正定义一个“truthy”列表。它给出了一些例子，比如它说所有的对象都是 truthy，但是语言规范大致上暗示着：任何没有明确地存在于 **falsy** 列表中的东西，都是 **truthy**。

Falsy 对象

等一下，这一节的标题听起来简直是矛盾的。我 刚刚才说过 语言规范将所有对象称为 truthy，对吧？应该没有“falsy 对象”这样的东西。

这会是什么意思呢？

它可能诱使你认为它意味着一个包装了 falsy 值（比如 `""`、`0` 或 `false`）的对象包装器（见第三章）。但别掉到这个 陷阱 中。

注意： 这个可能是一个语言规范的微妙笑话。

考虑下面的代码：

```
1. var a = new Boolean( false );
2. var b = new Number( 0 );
3. var c = new String( "" );
```

我们知道这三个值都是包装了明显是 falsy 值的对象（见第三章）。但这些对象是作为 `true` 还是作为 `false` 动作呢？这很容易回答：

```
1. var d = Boolean( a && b && c );
2.
3. d; // true
```

所以，三个都作为 `true` 动作，这是唯一能使 `d` 得到 `true` 的方法。

提示： 注意包在 `a && b && c` 表达式外面的 `Boolean( .. )` —— 你可能想知道为什么它在这儿。我们会在本章稍后回到这个话题，所以先做个心理准备。为了先睹为快，你可以自己试试如果没有 `Boolean( .. )` 调用而只有 `d = a && b && c` 时 `d` 是什么。

那么，如果“falsy 对象” 不是包装着 **falsy** 值的对象，它们是什么鬼东西？

刁钻的地方在于，它们可以出现在你的 JS 程序中，但它们实际上不是 JavaScript 本身的一部分。

什么！？

有些特定的情况，在普通的 JS 语义之上，浏览器已经创建了它们自己的某种 外来 值的行为，也就是这种“falsy 对象”的想法。

一个“falsy 对象”看起来和动起来都像一个普通对象（属性，等等）的值，但是当你强制转换它为一个 `boolean` 时，它会变为一个 `false` 值。

为什么！？

最著名的例子是 `document.all`：一个 由 *DOM*（不是 JS 引擎本身） 给你的 JS 程序提供的类数组（对象），它向你的 JS 程序暴露你页面上的元素。它 曾经 像一个普通对象那样动作 —— 是一个 truthy。但不再是了。

`document.all` 本身从来就不是“标准的”，而且从很早以前就被废弃/抛弃了。

“那他们就不能删掉它吗？” 对不起，想得不错。但愿它们能。但是世面上有太多的遗产 JS 代码库依赖于它。

那么，为什么使它像 falsy 一样动作？因为从 `document.all` 到 `boolean` 的强制转换（比如在 `if` 语句中）几乎总是用来检测老的，非标准的 IE。

IE 从很早以前就开始顺应规范了，而且在许多情况下它在推动 web 向前发展的作用和其他浏览器一样多，甚至更多。但是所有那些老旧的 `if (document.all) { /* it's IE */ }` 代码依然留在世面上，而且大多数可能永远都不会消失。所有这些遗产代码依然假设它们运行在那些给 IE 用户带来差劲儿的浏览体验的，几十年前的老 IE 上，

所以，我们不能完全移除 `document.all`，但是 IE 不再想让 `if (document.all) { .. }` 代码继续工作了，这样现代 IE 的用户就能得到新的，符合标准的代码逻辑。

“我们应当怎么做？” “我知道了！让我们黑进 JS 的类型系统并假装 `document.all` 是 falsy！”

呃。这很烂。这是一个大多数 JS 开发者们都不理解的疯狂的坑。但是其它的替代方案（对上面两败俱伤的问题什么都不做）还要烂得 多那么一点点。

所以……这就是我们得到的：由浏览器给 JavaScript 添加的疯狂、非标准的“falsy 对象”。耶！

Truthy 值

回到 truthy 列表。到底什么是 truthy 值？记住：如果一个值不在 **falsy** 列表中，它就是 **truthy**。

考虑下面代码：

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

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

d;
```

你期望这里的 `d` 是什么值？它要么是 `true` 要么是 `false`。

它是 `true`。为什么？因为尽管这些 `string` 值的内容看起来是falsy值，但是 `string` 值本身都是truthy，而这是因为在falsy列表中 `""` 是唯一的 `string` 值。

那么这些呢？

```
var a = [];                // 空数组 -- truthy 还是 falsy?
var b = {};                // 空对象 -- truthy 还是 falsy?
var c = function(){};    // 空函数 -- truthy 还是 falsy?

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

d;
```

是的，你猜到了，这里的 `d` 依然是 `true`。为什么？和前面的原因一样。尽管它们看起来像，但是 `[]`，`{}`，和 `function(){}` 不在 falsy列表中，因此它们是truthy值。

换句话说，truthy列表是无限长的。不可能制成一个这样的列表。你只能制造一个falsy列表并查询它。

花五分钟，把falsy列表写在便利贴上，然后粘在你的电脑显示器上，或者如果你愿意就记住它。不管哪种方法，你都可以在自己需要的时候通过简单地查询一个值是否在falsy列表中，来构建一个虚拟的truthy列表。

truthy和falsy的重要性在于，理解如果一个值在被（明确地或隐含地）强制转换为 `boolean` 值的话，它将如何动作。现在你的大脑中有了这两个列表，我们可以深入强制转换的例子本身了。

明确的强制转换

明确的强制转换

明确的 强制转换指的是明显且明确的类型转换。对于大多数开发者来说，有很多类型转换的用法可以清楚地归类于这种 明确的 强制转换。

我们在这里的目标是，在我们的代码中指明一些模式，在这些模式中我们可以清楚明白地将一个值从一种类型转换至另一种类型，以确保不给未来将读到这段代码的开发者留下任何坑。我们越明确，后来的人就越容易读懂我们的代码，也不必费太多的力气去理解我们的意图。

关于 明确的 强制转换可能很难找到什么主要的不同意见，因为它与被广泛接受的静态类型语言中的类型转换的工作方式非常接近。因此，我们理所当然地认为（暂且） 明确的 强制转换可以被认同为不是邪恶的，或没有争议的。虽然我们稍后会回到这个话题。

明确地：Strings <—> Numbers

我们将从最简单，也许是最常见强制转换操作开始：将值在 string 和 number 表现形式之间进行强制转换。

为了在 string 和 number 之间进行强制转换，我们使用内建的 String(..) 和 Number(..) 函数（我们在第三章中所指的“原生构造器”），但 非常重要的是，我们不在它们前面使用 new 关键字。这样，我们就不是在创建对象包装器。

取而代之的是，我们实际上在两种类型之间进行 明确地强制转换：

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

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

b; // "42"
```

```
d; // 3.14
```

`String(..)` 使用早先讨论的 `ToString` 操作的规则，将任意其它的值强制转换为一个基本类型的 `string` 值。 `Number(..)` 使用早先讨论过的 `ToNumber` 操作的规则，将任意其他的值强制转换为一个基本类型的 `number` 值。

我称此为 明确的 强制转换是因为，一般对于大多数开发者来说这是十分明显的：这些操作的最终结果是适当的类型转换。

实际上，这种用法看起来与其他的静态类型语言中的用法非常相像。

举个例子，在C/C++中，你既可以说 `(int)x` 也可以说 `int(x)` ，而且它们都将 `x` 中的值转换为一个整数。两种形式都是合法的，但是许多人偏向于后者，它看起来有点儿像一个函数调用。在JavaScript中，当你说 `Number(x)` 时，它看起来极其相似。在JS中它实际上是一个函数调用这个事实重要吗？并非如此。

除了 `String(..)` 和 `Number(..)` ，还有其他的方法可以把这些值在 `string` 和 `number` 之间进行“明确地”转换：

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

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

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

调用 `a.toString()` 在表面上是明确的（“toString”意味着“变成一个字符串”是很明白的），但是这里有一些藏起来的隐含性。 `toString()` 不能在像 `42` 这样的 基本类型 值上调用。所以JS会自动地将 `42` “封箱”在一个对象包装器中（见第三章），这样 `toString()` 就可以针对这个对象调用。换句话讲，你可能会叫它“明确的隐含”。

这里的 `+c` 是 `+` 操作符的 一元操作符（操作符只有一个操作数）形式。取代进行数学加法（或字符串连接 —— 见下面的讨论）的是，一元的 `+` 明确地将它的操作数（ `c` ）强制转换为一个 `number` 值。

`+c` 是 明确的 强制转换吗？这要看你的经验和角度。如果你知道（现在你知道了！）一元 `+` 明确地意味着 `number` 强制转换，那么它就是相当明确和明显的。但是，如果你以前从没见过它，那么它看起来就极其困惑，晦涩，带有隐含的副作用，等等。

注意： 在开源的JS社区中一般被接受的观点是，一元 `+` 是一个 明确的 强制转换形式。

即使你真的喜欢 `+c` 这种形式，它绝对会在有的地方看起来非常令人困惑。考虑下面的代码：

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

d; // 8.14
```

一元 `-` 操作符也像 `+` 一样进行强制转换，但它还会翻转数字的符号。但是你不能放两个减号 `--` 来使符号翻转回来，因为那将被解释为递减操作符。取代它的是，你需要这么做： `- -"3.14"` ，在两个减号之间加入空格，这将会使强制转换的结果为 `3.14` 。

你可能会想到所有种类的可怕组合 —— 一个二元操作符挨着另一个操作符的一元形式。这里有另一个疯狂的例子：

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

当一个一元 `+` （或 `-` ）紧邻其他操作符时，你应当强烈地考虑避免使用它。虽然上面的代码可以工作，但几乎全世界都认为它是一个坏主意。即使是 `d = +c` （或者 `d =+ c` ！）都太容易与 `d += c` 像混淆了，而后者完全是不同的东西！

注意： 一元 `+` 的另一个极端使人困惑的地方是，被用于紧挨着另一个将要作为 `++` 递增操作符和 `--` 递减操作符的操作数。例如： `a +++b` ， `a + ++b` ，和 `a + + +b` 。更多关于 `++` 的信息，参见第五章的“表达式副作用”。

记住，我们正努力变得明确并 减少 困惑，不是把事情弄得更糟！

从 `Date` 到 `number`

另一个一元 `+` 操作符的常见用法是将一个 `Date` 对象强制转换为一个 `number` ，其结果是这个日期/时间值的unix时间戳（从世界协调时间的1970年1月1日0点开始计算，经过的毫秒数）表现形式：

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

+d; // 1408369986000
```

这种习惯性用法经常用于取得当前的 现在 时刻的时间戳，比如：

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

注意： 一些开发者知道一个JavaScript中的特别的语法“技巧”，就是在构造器调用（一个带有 `new` 的函数调用）中如果没有参数值要传递的话， `()` 是 可选的。所以你可能遇到 `var timestamp = +new Date;` 形式。然而，不是所有的开发者都同意忽略 `()` 可以增强可读性，因为它是一种不寻常的语法特例，只能适用于 `new fn()` 调用形式，而不能用于普通的 `fn()` 调用形式。

但强制转换不是从 `Date` 对象中取得时间戳的唯一方法。一个不使用强制转换的方式可能更好，因为它更加明确：

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

但是一个 更更好的 不使用强制转换的选择是使用ES5加入的 `Date.now()` 静态函数：

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

而且如果你想要为老版本的浏览器填补 `Date.now()` 的话，也十分简单：

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

我推荐跳过与日期有关的强制转换形式。使用 `Date.now()` 来取得当前 现在 的时间戳，而使用 `new Date( .. ).getTime()` 来取得一个需要你指定的 非现在 日期/时间的时间戳。

奇异的 `~`

一个经常被忽视并通常让人糊涂的JS强制操作符是波浪线 `~` 操作符（也叫“按位取反”，“比特非”）。许多理解它在做什么的人也总是想要避开它。但是为了坚持我们在本书和本系列中的精神，让我们深入并找出 `~` 是否有一些对我们有用的东西。

在第二章的“32位（有符号）整数”一节，我们讲解了在JS中位操作符是如何仅为32位操作定义的，这意味着我们强制它们的操作数遵循32位值的表现形式。这个规则如何发生是由 `ToInt32` 抽象操作（ES5语言规范，9.5部分）控制的。

`ToInt32` 首先进行 `ToNumber` 强制转换，这就是说如果值是 `"123"` ，它在 `ToInt32` 规则实施之前会首先变成 `123` 。

虽然它本身没有 技术上进行 强制转换（因为类型没有改变），但对一些特定的特殊 `number` 值使用位操作符（比如 `|` 或 `~` ）会产生一种强制转换效果，这种效果的结果是一个不同的 `number` 值。

举例来说，让我们首先考虑惯用的空操作 `0 | x` （在第二种章有展示）中使用的 `|` “比特或”操作符，它实质上仅仅进行 `ToInt32` 转换：

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

```
4. 0 | -Infinity;    // 0
```

这些特殊的数字是不可用32位表现的（因为它们源自64位的IEEE 754标准 —— 见第二章），所以 `ToInt32` 将这些值的结果指定为 `0` 。

有争议的是， `0 | __` 是否是一种 `ToInt32` 强制转换操作的 明确的 形式，还是更倾向于 隐含。从语言规范的角度来说，毫无疑问是 明确的，但是如果你没有在这样的层次上理解位操作，它就可能看起来有点像 隐含的 魔法。不管怎样，为了与本章中其他的断言保持一致，我们称它为 明确的。

那么，让我们把注意力转回 `~` 。 `~` 操作符首先将值“强制转换”为一个32位 `number` 值，然后实施按位取反（翻转每一个比特位）。

注意： 这与 `!` 不仅强制转换它的值为 `boolean` 而且还翻转它的每一位很相似（见后面关于“一元 `!` ”的讨论）。

但是……什么！？为什么我们要关心被翻转的比特位？这是一些相当特殊的，微妙的东西。JS开发者需要推理个别比特位是十分少见的。

另一种考虑 `~` 定义的方法是， `~` 源自学校中的计算机科学/离散数学： `~` 进行二进制取补操作。太好了，谢谢，我完全明白了！

我们再试一次： `~x` 大致与 `-(x+1)` 相同。这很奇怪，但是稍微容易推理一些。所以：

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

你可能还在想 `~` 这个鬼东西到底和什么有关，或者对于强制转换的讨论它究竟有什么要紧。让我们快速进入要点。

考虑一下 `-(x+1)` 。通过进行这个操作，能够产生结果 `0` （或者从技术上说 `-0` ！）的唯一的值是什么？ `-1` 。换句话说， `~` 用于一个范围的 `number` 值时，将会为输入值 `-1` 产生一个falsy（很容易强制转换为 `false` ）的 `0` ，而为任意其他的输入产生truthy的 `number` 。

为什么这要紧？

`-1` 通常称为一个“哨兵值”，它基本上意味着一个在同类型值（ `number` ）的更大的集合中被赋予了任意的语义。在C语言中许多函数使用哨兵值 `-1` ，它们返回 `>= 0` 的值表示“成功”，返回 `-1` 表示“失败”。

JavaScript在定义 `string` 操作 `indexOf(..)` 时采纳了这种先例，它搜索一个子字符串，如果找到就返回它从0开始计算的索引位置，没有找到的话就返回 `-1` 。

这样的情况很常见：不仅仅将 `indexOf(..)` 作为取得位置的操作，而且作为检查一个子字符串存在/不存在于另一个 `string` 中的 `boolean` 值。这就是开发者们通常如何进行这样的检查：

```
1. var a = "Hello World";
```

```
2.
3. if (a.indexOf( "lo" ) >= 0) {    // true
4.     // 找到了！
5. }
6. if (a.indexOf( "lo" ) != -1) {    // true
7.     // 找到了
8. }
9.
10. if (a.indexOf( "ol" ) < 0) {    // true
11.     // 没找到！
12. }
13. if (a.indexOf( "ol" ) == -1) {    // true
14.     // 没找到！
15. }
```

我感觉看着 `>= 0` 或 `== -1` 有些恶心。它基本上是一种“抽象泄漏”，这里它将底层的实现行为 —— 使用哨兵值 `-1` 表示“失败” —— 泄漏到我的代码中。我倒是乐意隐藏这样的细节。

现在，我们终于看到为什 `~` 可以帮到我们了！将 `~` 和 `indexOf()` 一起使用可以将值“强制转换”（实际上只是变形）为 可以适当地强制转换为 `boolean` 的值：

```
1. var a = "Hello World";
2.
3. ~a.indexOf( "lo" );            // -4   <-- truthy!
4.
5. if (~a.indexOf( "lo" )) {    // true
6.     // 找到了！
7. }
8.
9. ~a.indexOf( "ol" );            // 0    <-- falsy!
10. !~a.indexOf( "ol" );        // true
11.
12. if (!~a.indexOf( "ol" )) {    // true
13.     // 没找到！
14. }
```

`~` 拿到 `indexOf(..)` 的返回值并将它变形：对于“失败”的 `-1` 我们得到falsy的 `0` ，而其他的值都是truthy。

注意： `~` 的假想算法 `-(x+1)` 暗示着 `~-1` 是 `-0` ，但是实际上它产生 `0` ，因为底层的操作其实是按位的，不是数学操作。

技术上讲， `if (~a.indexOf(..))` 仍然依靠 隐含的 强制转换将它的结果 `0` 变为 `false` 或非零变为 `true` 。但总的来说，对我而言 `~` 更像一种 明确的 强制转换机制，只要你知道在这种惯用法中它的意图是什么。

我感觉这样的代码要比前面凌乱的 `>= 0` / `== -1` 更干净。

截断比特位

在你遇到的代码中，还有一个地方可能出现 `~`：一些开发者使用双波浪线 `~~` 来截断一个 `number` 的小数部分（也就是，将它“强制转换”为一个“整数”）。这通常（虽然是错误的）被说成与调用 `Math.floor(..)` 的结果相同。

`~ ~` 的工作方式是，第一个 `~` 实施 `ToInt32` “强制转换”并进行按位取反，然后第二个 `~` 进行另一次按位取反，将每一个比特位都翻转回原来的状态。于是最终的结果就是 `ToInt32` “强制转换”（也叫截断）。

注意： `~~` 的按位双翻转，与双否定 `!!` 的行为非常相似，它将在稍后的“明确地：* –> Boolean”一节中讲解。

然而， `~~` 需要一些注意/澄清。首先，它仅在32位值上可以可靠地工作。但更重要的是，它在负数上工作的方式与 `Math.floor(..)` 不同！

```
1. Math.floor( -49.6 );    // -50
2. ~~-49.6;                // -49
```

把 `Math.floor(..)` 的不同放在一边， `~~x` 可以将值截断为一个（32位）整数。但是 `x | 0` 也可以，而且看起来还（稍微）省事儿 一些。

那么，为什么你可能会选择 `~~x` 而不是 `x | 0` ？操作符优先权（见第五章）：

```
1. ~~1E20 / 10;        // 166199296
2.
3. 1E20 | 0 / 10;        // 1661992960
4. (1E20 | 0) / 10;    // 166199296
```

正如这里给出的其他建议一样，仅在读/写这样的代码的每一个人都知道这些操作符如何工作的情况下，才将 `~` 和 `~~` 作为“强制转换”和将值变形的明确机制。

明确地：解析数字字符串

将一个 `string` 强制转换为一个 `number` 的类似结果，可以通过从 `string` 的字符内容中解析（parsing）出一个 `number` 得到。然而在这种解析和我们上面讲解的类型转换之间存在着区别。

考虑下面的代码：

```
1. var a = "42";
2. var b = "42px";
3.
```

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

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

从一个字符串中解析出一个数字是 容忍 非数字字符的 —— 从左到右，如果遇到非数字字符就停止解析 —— 而强制转换是 不容忍 并且会失败而得出值 `NaN` 。

解析不应当被视为强制转换的替代品。这两种任务虽然相似，但是有着不同的目的。当你不知道/不关心右手边可能有什么其他的非数字字符时，你可以将一个 `string` 作为 `number` 解析。当只有数字才是可接受的值，而且像 `"42px"` 这样的东西作为数字应当被排除时，就强制转换一个 `string` （变为一个 `number` ）。

提示： `parseInt(..)` 有一个孪生兄弟， `parseFloat(..)` ，它（听起来）从一个字符串中拉出一个浮点数。

不要忘了 `parseInt(..)` 工作在 `string` 值上。向 `parseInt(..)` 传递一个 `number` 绝对没有任何意义。传递其他任何类型也都没有意义，比如 `true` ， `function(){..}` 或 `[1,2,3]` 。

如果你传入一个非 `string` ，你所传入的值首先将自动地被强制转换为一个 `string` （见早先的“ `ToString` ”），这很明显是一种隐藏的 隐含 强制转换。在你的程序中依赖这样的行为真的是一个坏主意，所以永远也不要将 `parseInt(..)` 与非 `string` 值一起使用。

在ES5之前， `parseInt(..)` 还存在另外一个坑，这曾是许多JS程序的bug的根源。如果你不传递第二个参数来指定使用哪种进制（也叫基数）来翻译数字的 `string` 内容， `parseInt(..)` 将会根据开头的字符进行猜测。

如果开头的两个字符是 `"0x"` 或 `"0X"` ，那么猜测（根据惯例）将是你想要将这个 `string` 翻译为一个16进制 `number` 。否则，如果第一个字符是 `"0"` ，那么猜测（也是根据惯例）将是你想要将这个 `string` 翻译成8进制 `number` 。

16进制的 `string` （以 `0x` 或 `0X` 开头）没那么容易搞混。但是事实证明8进制数字的猜测过于常见了。比如：

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

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

看起来无害，对吧？试着在小时上选择 `08` 在分钟上选择 `09` 。你会得到 `0:0` 。为什么？因为 `8` 和 `9` 都不是合法的8进制数。

ES5之前的修改很简单，但是很容易忘：总是在第二个参数值上传递 `10` 。这完全是安全的：

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

在ES5中， `parseInt(..)` 不再猜测八进制数了。除非你指定，否则它会假定为10进制（或者为 `"0x"` 前缀猜测16进制数）。这好多了。只是要小心，如果你的代码不得不运行在前ES5环境中，你仍然需要为基数传递 `10` 。

解析非字符串

几年以前有一个挖苦JS的玩笑，使一个关于 `parseInt(..)` 行为的一个臭名昭著的例子备受关注，它取笑JS的这个行为：

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

这里面设想（但完全不合法）的断言是，“如果我传入一个无限大，并从中解析出一个整数的话，我应该得到一个无限大，不是18”。没错，JS一定是疯了才得出这个结果，对吧？

虽然这是个明显故意造成的，不真实的例子，但是让我们放纵这种疯狂一小会儿，来检视一下JS是否真的那么疯狂。

首先，这其中最明显的原罪是将一个非 `string` 传入了 `parseInt(..)` 。这是不对的。这么做是自找麻烦。但就算你这么做了，JS也会礼貌地将你传入的东西强制转换为它可以解析的 `string` 。

有些人可能会争论说这是一种不合理的行为， `parseInt(..)` 应当拒绝在一个非 `string` 值上操作。它应该抛出一个错误吗？坦白地说，像Java那样。但是一想到JS应当开始在满世界抛出错误，以至于几乎每一行代码都需要用 `try..catch` 围起来，我就不寒而栗。

它应当返回 `NaN` 吗？也许。但是……要是这样呢：

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

这也应当失败吗？它是一个非 `string` 值啊。如果你想让 `String` 对象包装器被开箱成 `"42"` ，那么 `42` 先变成 `"42"` ，以使 `42` 可以被解析回来就那么不寻常吗？

我会争论说，这种可能发生的半 明确 半 隐含 的强制转换经常可以成为非常有用的东西。比如：

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

parseInt( a ); // 42
```

事实上 `parseInt(..)` 将它的值强制转换为 `string` 来实施解析是十分合理的。如果你传垃圾进去，那么你就会得到垃圾，不要责备垃圾桶 —— 它只是忠实地尽自己的责任。

那么，如果你传入像 `Infinity` （很明显是 `1 / 0` 的结果）这样的值，对于它的强制转换来说哪种 `string` 表现形式最有道理呢？我脑中只有两种合理的选择： `"Infinity"` 和 `"∞"` 。JS选择了 `"Infinity"` 。我很高兴它这么选。

我认为在JS中 所有的值 都有某种默认的 `string` 表现形式是一件好事，这样它们就不是我们不能调试和推理的神秘黑箱了。

现在，关于19进制呢？很明显，这完全是伪命题和造作。没有真实的JS程序使用19进制。那太荒谬了。但是，让我们再一次放任这种荒谬。在19进制中，合法的数字字符是 `0` - `9` 和 `a` - `i` （大小写无关）。

那么，回到我们的 `parseInt( 1/0, 19 )` 例子。它实质上是 `parseInt( "Infinity", 19 )` 。它如何解析？第一个字符是 `"I"` ，在愚蠢的19进制中是值 `18` 。第二个字符 `"n"` 不再合法的数字字符集内，所以这样的解析就礼貌地停止了，就像它在 `"42px"` 中遇到 `"p"` 那样。

结果呢？ `18` 。正如它应该的那样。对JS来说，并非一个错误或者 `Infinity` 本身，而是将我们带到这里的一系列的行为才是 非常重要 的，不应当那么简单地被丢弃。

其他关于 `parseInt(..)` 行为的，令人吃惊但又十分合理的例子还包括：

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

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

其实 `parseInt(..)` 在它的行为上是相当可预见和一致的。如果你正确地使用它，你就能得到合理的结果。如果你不正确地使用它，那么你得到的疯狂结果并不是JavaScript的错。

明确地：* -> Boolean

现在，我们来检视从任意的非 `boolean` 值到一个 `boolean` 值的强制转换。

正如上面的 `String(..)` 和 `Number(..)` ， `Boolean(..)` （当然，不带 `new` ！）是强制进行 `ToBoolean` 转换的明确方法：

```
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(..)` 是非常明确的，但是它并不常见也不为人所惯用。

正如一元 `+` 操作符将一个值强制转换为一个 `number` （参见上面的讨论），一元的 `!` 否定操作符可以将一个值明确地强制转换为一个 `boolean` 。问题 是它还将值从truthy翻转为falsy，或反之。所以，大多数JS开发者使用 `!!` 双否定操作符进行 `boolean` 强制转换，因为第二个 `!` 将会把它翻转回原本的true或false：

```
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
```

没有 `Boolean(..)` 或 `!!` 的话，任何这些 `ToBoolean` 强制转换都将 隐含地 发生，比如在一个 `if (..) ..` 语句这样使用 `boolean` 的上下文中。但这里的目标是，明确地强制一个值成为 `boolean` 来使 `ToBoolean` 强制转换的意图显得明明白白。

另一个 `ToBoolean` 强制转换的用例是，如果你想在数据结构的JSON序列化中强制转换一个 `true` / `false` ：

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

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

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

如果你是从Java来到JavaScript的话，你可能会认得这个惯用法：

```
var a = 42;

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

`? :` 三元操作符将会测试 `a` 的真假，然后根据这个测试的结果相应地将 `true` 或 `false` 赋值给 `b` 。

表面上，这个惯用法看起来是一种 明确的 `ToBoolean` 类型强制转换形式，因为很明显它操作的结果要么是 `true` 要么是 `false` 。

然而，这里有一个隐藏的 隐含 强制转换，就是表达式 `a` 不得不首先被强制转换为 `boolean` 来进行真假测试。我称这种惯用法为“明确地隐含”。另外，我建议你在JavaScript中 完全避免这种惯用法。它不会提供真正的好处，而且会让事情变得更糟。

对于 明确的 强制转换 `Boolean(a)` 和 `!!a` 是好得多的选项。

隐含的强制转换

隐含的强制转换

隐含的 强制转换是指这样的类型转换：它们是隐藏的，由于其他的动作隐含地发生的不明显的副作用。换句话说，任何（对你）不明显的类型转换都是 隐含的强制转换。

虽然 明确的 强制转换的目的很明白，但是这可能 太过 明显 —— 隐含的 强制转换拥有相反的目的：使代码更难理解。

从表面上来看，我相信这就是许多关于强制转换的愤怒的源头。绝大多数关于“JavaScript强制转换”的抱怨实际上都指向了（不管他们是否理解它） 隐含的 强制转换。

注意： Douglas Crockford，*“JavaScript: The Good Parts”* 的作者，在许多会议和他的作品中声称应当避免JavaScript强制转换。但看起来他的意思是 隐含的 强制转换是不好的（以他的意见）。然而，如果你读他自己的代码的话，你会发现相当多的强制转换的例子，明确 和 隐含 都有！事实上，他的担忧主要在于 `==` 操作，但正如你将在本章中看到的，那只是强制转换机制的一部分。

那么，隐含强制转换 是邪恶的吗？它很危险吗？它是JavaScript设计上的缺陷吗？我们应该尽一切力量避免它吗？

我打赌大多数读者都倾向于踊跃地欢呼，“是的！”

别那么着急。听我把话说完。

让我们在 隐含的 强制转换是什么，和可以是什么这个问题上采取一个不同的角度，而不是仅仅说它是“好的明确强制转换的反面”。这太过狭隘，而且忽视了一个重要的微妙细节。

让我们将 隐含的 强制转换的目的定义为：减少搞乱我们代码的繁冗，模板代码，和/或不必要的实现细节，不使它们的噪音掩盖更重要的意图。

用于简化的隐含

在我们进入JavaScript以前，我建议使用某个理论上是强类型的语言的假想代码来说明一下：

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

在这个例子中，我在 `y` 中有一些任意类型的值，想把它转换为 `SomeType` 类型。问题是，这种语言不能从当前 `y` 的类型直接走到 `SomeType` 。它需要一个中间步骤，它首先转换为 `AnotherType` ，然后从 `AnotherType` 转换到 `SomeType` 。

现在，要是这种语言（或者你可用这种语言创建自己的定义）允许你这么说呢：

```
1. SomeType x = SomeType( y )
```

难道一般来说你不会同意我们简化了这里的类型转换，降低了中间转换步骤的无谓的“噪音”吗？我的意思是，在这段代码的这一点上，能看到并处理 `y` 先变为 `AnotherType` 然后再变为 `SomeType` 的事实，真的 是很重要的一件事吗？

有些人可能会争辩，至少在某些环境下，是的。但我想我可以做出相同的争辩说，在许多其他的环境下，不管是通过语言本身的还是我们自己的抽象，这样的简化通过抽象或隐藏这些细节 确实增强了代码的可读性。

毫无疑问，在幕后的某些地方，那个中间的步骤依然是发生的。但如果这样的细节在视野中隐藏起来，我们就可以将使 `y` 变为类型 `SomeType` 作为一个泛化操作来推理，并隐藏混乱的细节。

虽然不是一个完美的类比，我要在本章剩余部分争论的是，JS的 隐含的 强制转换可以被认为是给你的代码提供了一个类似的辅助。

但是，很重要的是，这不是一个无边界的，绝对的论断。绝对有许多 邪恶的东西 潜伏在 隐含 强制转换周围，它们对你的代码造成的损害要比任何潜在的可读性改善厉害的多。很清楚，我们不得不学习如何避免这样的结构，使我们不会用各种bug来毒害我们的代码。

许多开发者相信，如果一个机制可以做某些有用的事儿 **A**，但也可以被滥用或误用来做某些可怕的事儿 **Z**，那么我们就应当将这种机制整个儿扔掉，仅仅是为了安全。

我对你的鼓励是：不要安心于此。不要“把孩子跟洗澡水一起泼出去”。不要因为你只见到过它的“坏的一面”就假设 隐含 强制转换都是坏的。我认为这里有“好的一面”，而我想要帮助和启发你们更多的人找到并接纳它们！

隐含地：Strings <—> Numbers

在本章的早先，我们探索了 `string` 和 `number` 值之间的 明确 强制转换。现在，让我们使用 隐含强制转换的方式探索相同的任务。但在我们开始之前，我们不得不检视一些将会 隐含地 发生强制转换的操作的微妙之处。

为了服务于 `number` 的相加和 `string` 的连接两个目的， `+` 操作符被重载了。那么JS如何知道你

想用的是哪一种操作呢？考虑下面的代码：

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

var c = 42;
var d = 0;

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

是什么不同导致了 `"420"` 和 `42` ？一个常见的误解是，这个不同之处在于操作数之一或两者是否是一个 `string` ，这意味着 `+` 将假设 `string` 连接。虽然这有一部分是对的，但实际情况要更复杂。

考虑如下代码：

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

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

两个操作数都不是 `string` ，但很明显它们都被强制转换为 `string` 然后启动了 `string` 连接。那么到底发生了什么？

（警告： 语言规范式的深度细节就要来了，如果这会吓到你就跳过下面两段！）

根据ES5语言规范的11.6.1部分， `+` 的算法是（当一个操作数是 `object` 值时），如果两个操作数之一已经是一个 `string` ，或者下列步骤产生一个 `string` 表达形式， `+` 将会进行连接。所以，当 `+` 的两个操作数之一收到一个 `object` （包括 `array` ）时，它首先在这个值上调用 `ToPrimitive` 抽象操作（9.1部分），而它会带着 `number` 的上下文环境提示来调用 `[[DefaultValue]]` 算法（8.12.8部分）。

如果你仔细观察，你会发现这个操作现在和 `ToNumber` 抽象操作处理 `object` 的过程是一样的（参见早先的“ `ToNumber` ”一节）。在 `array` 上的 `valueOf()` 操作将会在产生一个简单基本类型时失败，于是它退回到一个 `toString()` 表现形式。两个 `array` 因此分别变成了 `"1,2"` 和 `"3,4"` 。现在， `+` 就如你通常期望的那样连接这两个 `string` ： `"1,23,4"` 。

让我们把这些乱七八糟的细节放在一边，回到一个早前的，简化的解释：如果 `+` 的两个操作数之一是一个 `string` （或在上面的步骤中成为一个 `string` ），那么操作就会是 `string` 连接。否则，它总是数字加法。

注意： 关于强制转换，一个经常被引用的坑是 `[] + {}` 和 `{} + []` ，这两个表达式的结果分别

是 `"[object Object]"` 和 `0` 。虽然对此有更多的东西，但是我们将在第五章的“Block”中讲解这其中的细节。

这对 隐含 强制转换意味着什么？

你可以简单地通过将 `number` 和空 `string``""` “相加”来把一个 `number` 强制转换为一个 `string` ：

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

b; // "42"
```

提示： 使用 `+` 操作符的数字加法是可交换的，这意味着 `2 + 3` 与 `3 + 2` 是相同的。使用 `+` 的字符串连接很明显通常不是可交换的，但是 对于 `""` 的特定情况，它实质上是可交换的，因为 `a + ""` 和 `"" + a` 会产生相同的结果。

使用一个 `+ ""` 操作将 `number` （隐含地）强制转换为 `string` 是极其常见/惯用的。事实上，有趣的是，一些在口头上批评 隐含 强制转换得最严厉的人仍然在他们自己的代码中使用这种方式，而不是使用它的 明确的 替代形式。

在 隐含 强制转换的有用形式中，我认为这是一个很棒的例子，尽管这种机制那么频繁地被人诟病！

将 `a + ""` 这种 隐含的 强制转换与我们早先的 `String(a)` 明确的 强制转换的例子相比较，有一个另外的需要小心的奇怪之处。由于 `ToPrimitive` 抽象操作的工作方式， `a + ""` 在值 `a` 上调用 `valueOf()` ，它的返回值再最终通过内部的 `ToString` 抽象操作转换为一个 `string` 。但是 `String(a)` 只直接调用 `toString()` 。

两种方式的最终结果都是一个 `string` ，但如果你使用一个 `object` 而不是一个普通的基本类型 `number` 的值，你可能不一定得到 相同的 `string` 值！

考虑这段代码：

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

a + "";           // "42"

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

一般来说这样的坑不会咬到你，除非你真的试着创建令人困惑的数据结构和操作，但如果你为某些 `object` 同时定义了你自己的 `valueOf()` 和 `toString()` 方法，你就应当小心，因为你强制转换这些值的方式将影响到结果。

那么另外一个方向呢？我们如何将一个 `string` 隐含强制转换 为一个 `number` ？

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

b; // 3.14
```

`-` 操作符是仅为数字减法定义的，所以 `a - 0` 强制 `a` 的值被转换为一个 `number` 。虽然少见得多， `a * 1` 或 `a / 1` 也会得到相同的结果，因为这些操作符也是仅为数字操作定义的。

那么对 `-` 操作符使用 `object` 值会怎样呢？和上面的 `+` 的故事相似：

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

a - b; // 2
```

两个 `array` 值都不得不变为 `number` ，但它们首先会被强制转换为 `string` （使用意料之中的 `toString()` 序列化），然后再强制转换为 `number` ，以便 `-` 减法操作可以实施。

那么， `string` 和 `number` 值之间的 隐含 强制转换还是你总是在恐怖故事当中听到的丑陋怪物吗？我个人不这么认为。

比较 `b = String(a)` （明确的）和 `b = a + ""` （隐含的）。我认为在你的代码中会出现两种方式都有用的情况。当然 `b = a + ""` 在JS程序中更常见一些，不管一般意义上 隐含 强制转换的好处或害处的 感觉 如何，它都提供了自己的用途。

隐含地: Booleans -> Numbers

我认为 隐含 强制转换可以真正闪光的一个情况是，将特定类型的复杂 `boolean` 逻辑简化为简单的数字加法。当然，这不是一个通用的技术，而是一个特定情况的特定解决方法。

考虑如下代码：

```
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
```

这个 `onlyOne(..)` 工具应当仅在正好有一个参数是 `true` /truthy时返回 `true` 。它在truthy的检查上使用 隐含的 强制转换，而在其他的地方使用 明确的 强制转换，包括最后的返回值。

但如果我们需要这个工具能够以相同的方式处理四个，五个，或者二十个标志值呢？很难想象处理所有那些比较的排列组合的代码实现。

但这里是 `boolean` 值到 `number` （很明显， `0` 或 `1` ）的强制转换可以提供巨大帮助的地方：

```
function onlyOne() {
    var sum = 0;
    for (var i=0; i < arguments.length; i++) {
        // 跳过falsy值。与将它们视为0相同，但是避开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
```

注意： 当然，除了在 `onlyOne(..)` 中的 `for` 循环，你可以更简洁地使用ES5的 `reduce(..)` 工具，但我不想因此而模糊概念。

我们在这里做的事情有赖于 `true` /truthy的强制转换结果为 `1` ，并将它们作为数字加起来。 `sum += arguments[i]` 通过 隐含的 强制转换使这发生。如果在 `arguments` 列表中有且仅有一个值为 `true` ，那么这个数字的和将是 `1` ，否则和就不是 `1` 而不能使期望的条件成立。

我们当然本可以使用 明确的 强制转换：

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

我们首先使用 `!!arguments[i]` 来将这个值强制转换为 `true` 或 `false` 。这样你就可以像 `onlyOne( "42", 0 )` 这样传入非 `boolean` 值了，而且它依然可以如意料的那样工作（要不然，你将会得到 `string` 连接，而且逻辑也不正确）。

一旦我们确认它是一个 `boolean` ，我们就使用 `Number(..)` 进行另一个 明确的 强制转换来确保值是 `0` 或 `1` 。

这个工具的 明确 强制转换形式“更好”吗？它确实像代码注释中解释的那样避开了 `NaN` 的陷阱。但是，这最终要看你的需要。我个人认为前一个版本，依赖于 隐含的 强制转换更优雅（如果你不传入 `undefined` 或 `NaN` ），而 明确的 版本是一种不必要的繁冗。

但与我们在这里讨论的几乎所有东西一样，这是一个主观判断。

注意： 不管是 隐含的 还是 明确的 方式，你可以通过将最后的比较从 `1` 改为 `2` 或 `5` ，来分别很容易地制造 `onlyTwo(..)` 或 `onlyFive(..)` 。这要比添加一大堆 `&&` 和 `||` 表达式要简单太多了。所以，一般来说，在这种情况下强制转换非常有用。

隐含地: * -> Boolean

现在，让我们将注意力转向目标为 `boolean` 值的 隐含 强制转换上，这是目前最常见，并且还是目前潜在的最麻烦的一种。

记住，隐含的 强制转换是当你以强制一个值被转换的方式使用这个值时才启动的。对于数字和 `string` 操作，很容易就能看出这种强制转换是如何发生的。

但是，哪个种类的表达式操作（隐含地）要求/强制一个 `boolean` 转换呢？

1. 在一个 `if (..)` 语句中的测试表达式。
2. 在一个 `for ( .. ; .. ; .. )` 头部的测试表达式（第二个子句）。
3. 在 `while (..)` 和 `do..while(..)` 循环中的测试表达式。
4. 在 `? :` 三元表达式中的测试表达式（第一个子句）。
5. `||` （“逻辑或”）和 `&&` （“逻辑与”）操作符左手边的操作数（它用作测试表达式 —— 见下面的讨论！）。

在这些上下文环境中使用的，任何还不是 `boolean` 的值，将通过本章早先讲解的 `ToBoolean` 抽象操作的规则，被 隐含地 强制转换为一个 `boolean` 。

我们来看一些例子：

```
1. var a = 42;
2. var b = "abc";
3. var c;
4. var d = null;
5.
```

```
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
}
```

在所有这些上下文环境中，非 `boolean` 值被 隐含地强制转换 为它们的 `boolean` 等价物，来决定测试的结果。

`||` 和 `&&` 操作符

很可能你已经在你用过的大多数或所有其他语言中见到过 `||` （“逻辑或”）和 `&&` （“逻辑与”）操作符了。所以假设它们在JavaScript中的工作方式和其他类似的语言基本上相同是很自然的。

这里有一个鲜为人知的，但很重要的，微妙细节。

其实，我会争辩这些操作符甚至不应当被称为“逻辑__操作符”，因为这样的名称没有完整地描述它们在做什么。如果让我给它们一个更准确的（也更蹩脚的）名称，我会叫它们“选择器操作符”或更完整的，“操作数选择器操作符”。

为什么？因为在JavaScript中它们实际上不会得出一个 逻辑 值（也就是 `boolean` ），这与它们在其他的语言中的表现不同。

那么它们到底得出什么？它们得出两个操作数中的一个（而且仅有一个）。换句话说，它们在两个操作数的值中选择一个。

引用ES5语言规范的11.11部分：

> 一个&&或||操作符产生的值不见得是Boolean类型。这个产生的值将总是两个操作数表达式其中之一的值。

让我们展示一下：

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

a || b;        // 42
```

```
a && b;        // "abc"

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

等一下，什么！？ 想一想。在像C和PHP这样的语言中，这些表达式结果为 `true` 或 `false` ，而在JS中（就此而言还有Python和Ruby！），结果来自于值本身。

`||` 和 `&&` 操作符都在 第一个操作数（ `a` 或 `c` ） 上进行 `boolean` 测试。如果这个操作数还不是 `boolean` （就像在这里一样），就会发生一次普通的 `ToBoolean` 强制转换，这样测试就可以进行了。

对于 `||` 操作符，如果测试结果为 `true` ， `||` 表达式就将 第一个操作数 的值（ `a` 或 `c` ）作为结果。如果测试结果为 `false` ， `||` 表达式就将 第二个操作数 的值（ `b` ）作为结果。

相反地，对于 `&&` 操作符，如果测试结果为 `true` ， `&&` 表达式将 第二个操作数 的值（ `b` ）作为结果。如果测试结果为 `false` ，那么 `&&` 表达式就将 第一个操作数 的值（ `a` 或 `c` ）作为结果。

`||` 或 `&&` 表达式的结果总是两个操作数之一的底层值，不是（可能是被强制转换来的）测试的结果。在 `c && b` 中， `c` 是 `null` ，因此是falsy。但是 `&&` 表达式本身的结果为 `null` （ `c` 中的值），不是用于测试的强制转换来的 `false` 。

现在你明白这些操作符如何像“操作数选择器”一样工作了吗？

另一种考虑这些操作数的方式是：

```
a || b;
// 大体上等价于：
a ? a : b;

a && b;
// 大体上等价于：
a ? b : a;
```

注意： 我说 `a || b` “大体上等价”于 `a ? a : b` ，是因为虽然结果相同，但是这里有一个微妙的不同。在 `a ? a : b` 中，如果 `a` 是一个更复杂的表达式（例如像调用 `function` 那样可能带有副作用），那么这个表达式 `a` 将有可能被求值两次（如果第一次求值的结果为truthy）。相比之下，对于 `a || b` ，表达式 `a` 仅被求值一次，而且这个值将被同时用于强制转换测试和结果值（如果合适的话）。同样的区别也适用于 `a && b` 和 `a ? b : a` 表达式。

很有可能你在没有完全理解之前你就已经使用了这个行为的一个极其常见，而且很有帮助的用法：

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

```
    b = b || "world";

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

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

这种 `a = a || "hello"` 惯用法（有时被说成C#“null合并操作符”的JavaScript版本）对 `a` 进行测试，如果它没有值（或仅仅是一个不期望的falsy值），就提供一个后备的默认值（ `"hello"` ）。

但是 要小心！

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

看到问题了吗？作为第二个参数的 `""` 是一个falsy值（参见本章早先的 `ToBoolean` ），所以 `b = b || "world"` 测试失败，而默认值 `"world"` 被替换上来，即便本来的意图可能是想让明确传入的 `""` 作为赋给 `b` 的值。

这种 `||` 惯用法极其常见，而且十分有用，但是你不得不只在 所有的*falsy*值 应当被跳过时使用它。不然，你就需要在你的测试中更加具体，而且可能应该使用一个 `? :` 三元操作符。

这种默认值赋值惯用法是如此常见（和有用！），以至于那些公开激烈诽谤JavaScript强制转换的人都经常在它们的代码中使用！

那么 `&&` 呢？

有另一种在手动编写中不那么常见，而在JS压缩器中频繁使用的惯用法。 `&&` 操作符会“选择”第二个操作数，当且仅当第一个操作数测试为truthy，这种用法有时被称为“守护操作符”（参见第五章的“短接”） —— 第一个表达式的测试“守护”着第二个表达式：

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

var a = 42;

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

`foo()` 仅在 `a` 测试为truthy时会被调用。如果这个测试失败，这个 `a && foo()` 表达式语句将会无声地停止 —— 这被称为“短接” —— 而且永远不会调用 `foo()` 。

重申一次，几乎很少有人手动编写这样的东西。通常，他们会写 `if (a) { foo(); }` 。但是JS压缩器选择 `a && foo()` 是因为它短的多。所以，现在，如果你不得不解读这样的代码，你就知道它是在做什么以及为什么了。

好了，那么 `||` 和 `&&` 在它们的功能上有些不错的技巧，只要你乐意让 隐含的 强制转换掺和进来。

注意： `a = b || "something"` 和 `a && b()` 两种惯用法都依赖于短接行为，我们将在第五章中讲述它的细节。

现在，这些操作符实际上不会得出 `true` 和 `false` 的事实可能使你的头脑有点儿混乱。你可能想知道，如果你的 `if` 语句和 `for` 循环包含 `a && (b || c)` 这样的复合的逻辑表达式，它们到底都是怎么工作的。

别担心！天没塌下来。你的代码（可能）没有问题。你只是可能从来没有理解在这个符合表达式被求值 之后，有一个向 `boolean` 隐含的 强制转换发生了。

考虑这段代码：

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

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

这段代码将会像你总是认为的那样工作，除了一个额外的微妙细节。 `a && (b || c)` 的结果 实际上是 `"foo"` ，不是 `true` 。所以，这之后 `if` 语句强制值 `"foo"` 转换为一个 `boolean` ，这理所当然地将是 `true` 。

看到了？没有理由惊慌。你的代码可能依然是安全的。但是现在关于它在做什么和如何做，你知道了更多。

而且现在你理解了这样的代码使用 隐含的 强制转换。如果你依然属于“避开（隐含）强制转换阵营”，那么你就需要退回去并使所有这些测试 明确：

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

祝你好运！...对不起，只是逗个乐儿。

Symbol 强制转换

在此为止，在 明确的 和 隐含的 强制转换之间几乎没有可以观察到的结果上的不同 —— 只有代码的可读性至关重要。

但是ES6的Symbol在强制转换系统中引入了一个我们需要简单讨论的坑。由于一个明显超出了我们将在本书中讨论的范围的原因，从一个 `symbol` 到一个 `string` 的 明确 强制转换是允许的，但是相同的 隐含 强制转换是不被允许的，而且会抛出一个错误。

考虑如下代码：

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

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

`symbol` 值根本不能强制转换为 `number` （不论哪种方式都抛出错误），但奇怪的是它们既可以 明确地 也可以 隐含地 强制转换为 `boolean` （总是 `true` ）。

一致性总是容易学习的，而对付例外从来就不有趣，但是我们只需要在ES6 `symbol` 值和我们如何强制转换它们的问题上多加小心。

好消息：你需要强制转换一个 `symbol` 值的情况可能极其少见。它们典型的被使用的方式（见第三章）可能不会用到强制转换。

宽松等价与严格等价

宽松等价与严格等价

宽松等价是 `==` 操作符，而严格等价是 `===` 操作符。两个操作符都被用于比较两个值的“等价性”，但是“宽松”和“严格”暗示着它们行为之间的一个 非常重要 的不同，特别是在它们如何决定“等价性”上。

关于这两个操作符的一个非常常见的误解是：“ `==` 检查值的等价性，而 `===` 检查值和类型的等价性。”虽然这听起来很好很合理，但是不准确。无数知名的JavaScript书籍和文章都是这么说的，但不幸的是它们都 错了。

正确的描述是：“ `==` 允许在等价性比较中进行强制转换，而 `===` 不允许强制转换”。

等价性的性能

停下来思考一下第一种（不正确的）解释和这第二种（正确的）解释的不同。

在第一种解释中，看起来 `===` 明显的要比 `==` 做更多工作，因为它还必须检查类型。在第二种解释中， `==` 是要 做更多工作 的，因为它不得不在类型不同时走过强制转换的步骤。

不要像许多人那样落入陷阱中，认为这会与性能有任何关系，虽然在这个问题上 `==` 好像要比 `===` 慢一些。强制转换确实要花费 一点点 处理时间，但也就是仅仅几微秒（是的，1微秒就是一秒的百万分之一！）。

如果你比较同类型的两个值， `==` 和 `===` 使用的是相同的算法，所以除了在引擎实现上的一些微小的区别，它们做的应当是相同的工作。

如果你比较两个不同类型的值，性能也不是重要因素。你应当问自己的是：当比较这两个值时，我想要进行强制转换吗？

如果你想要进行强制转换，使用 `==` 宽松等价，但如果你不想进行强制转换，就使用 `===` 严格等价。

注意： 这里暗示 `==` 和 `===` 都会检查它们的操作数的类型。不同之处在于它们在类型不同时如何反应。

抽象等价性

在ES5语言规范的11.9.3部分中， `==` 操作符的行为被定义为“抽象等价性比较算法”。那里列出了一个详尽但简单的算法，它明确地指出了类型的每一种可能的组合，与对于每一种组合强制转化应当如何发生（如果有必要的话）。

警告： 当（隐含的）强制转换被中伤为太过复杂和缺陷过多而不能成为 有用的，好的部分 时，遭到谴责的正是这些“抽象等价”规则。一般上，它们被认为对于开发者来说过于复杂和不直观而不能实际学习和应用，而且在JS程序中，和改善代码的可读性比起来，它倾向于导致更多的bug。我相信这是一种有缺陷的预断 —— 读者都是整天都在写（而且读，理解）算法（也就是代码）的能干的开发者。所以，接下来的是用简单的词语来直白地解读“抽象等价性”。但我恳请你也去读一下ES5规范的11.9.3部分。我想你将会对它是多么合理而感到震惊。

基本上，它的第一个条款（11.9.3.1）是在说，如果两个被比较的值是同一类型，它们就像你期望的那样通过等价性简单自然地比较。比如， `42` 只和 `42` 相等，而 `"abc"` 只和 `"abc"` 相等。

在一般期望的结果中，有一些例外需要小心：

- `NaN` 永远不等于它自己（见第二章）
- `+0` 和 `-0` 是相等的（见第二章）

条款11.9.3.1的最后一个规定是关于 `object` （包括 `function` 和 `array` ）的 `==` 宽松相等性比较。这样的两个值仅在它们引用 完全相同的值 时 相等。这里没有强制转换发生。

注意： `===` 严格等价比较与11.9.3.1的定义一模一样，包括关于两个 `object` 的值的规定。很少有人知道，在两个 `object` 被比较的情况下， `==` 和 `===` 的行为相同！

11.9.3算法中的剩余部分指出，如果你使用 `==` 宽松等价来比较两个不同类型的值，它们两者或其中之一将需要被 隐含地 强制转换。由于这个强制转换，两个值最终归于同一类型，可以使用简单的值的等价性来直接比较它们相等与否。

注意： `!=` 宽松不等价操作是如你预料的那样定义的，它差不多就是 `==` 比较操作完整实施，之后对结果取反。这对于 `!==` 严格不等价操作也是一样的。

比较： `string` 与 `number`

为了展示 `==` 强制转换，首先让我们建立本章中早先的 `string` 和 `number` 的例子：

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

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

我们所预料的， `a === b` 失败了，因为不允许强制转换，而且值 `42` 和 `"42"` 确实是不同的。

然而，第二个比较 `a == b` 使用了宽松等价，这意味着如果类型偶然不同，这个比较算法将会对两个或其中一个值实施 隐含的 强制转换。

那么这里发生的究竟是那种强制转换呢？是 `a` 的值变成了一个 `string` ，还是 `b` 的值 `"42"` 变成了一个 `number` ？

在ES5语言规范中，条款11.9.3.4-5说：

> 1. 如果*Type(x)*是*Number*而*Type(y)*是*String*，
>   返回比较*x == ToNumber(y)*的结果。
> 2. 如果*Type(x)*是*String*而*Type(y)*是*Number*，
>   返回比较*ToNumber(x) == y*的结果。

警告： 语言规范中使用 `Number` 和 `String` 作为类型的正式名称，虽然这本书中偏好使用 `number` 和 `string` 指代基本类型。别让语言规范中首字母大写的 `Number` 与 `Number()` 原生函数把你给搞糊涂了。对于我们的目的来说，类型名称的首字母大写是无关紧要的 —— 它们基本上是同一个意思。

显然，语言规范说为了比较，将值 `"42"` 强制转换为一个 `number` 。这个强制转换如何进行已经在前面将结过了，明确地说就是通过 `ToNumber` 抽象操作。在这种情况下十分明显，两个值 `42` 是相等的。

比较：任何东西与 `boolean`

当你试着将一个值直接与 `true` 或 `false` 相比较时，你会遇到 `==` 宽松等价的 隐含 强制转换中最大的一个坑。

考虑如下代码：

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

a == b;    // false
```

等一下，这里发生了什么！？我们知道 `"42"` 是一个truthy值（见本章早先的部分）。那么它和 `true` 怎么不是 `==` 宽松等价的？

其中的原因既简单又刁钻得使人迷惑。它是如此的容易让人误解，许多JS开发者从来不会花费足够多的精力来完全掌握它。

让我们再次引用语言规范，条款11.9.3.6-7

> 1. 如果*Type(x)*是*Boolean*，
>   返回比较 *ToNumber(x) == y* 的结果。
> 2. 如果*Type(y)*是*Boolean*，
>   返回比较 *x == ToNumber(y)* 的结果。

我们来把它分解。首先：

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

x == y; // false
```

`Type(x)` 确实是 `Boolean` ，所以它会实施 `ToNumber(x)` ，将 `true` 强制转换为 `1` 。现在， `1 == "42"` 会被求值。这里面的类型依然不同，所以（实质上是递归地）我们再次向早先讲解过的算法求解，它将 `"42"` 强制转换为 `42` ，而 `1 == 42` 明显是 `false` 。

反过来，我们任然得到相同的结果：

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

x == y; // false
```

这次 `Type(y)` 是 `Boolean` ，所以 `ToNumber(y)` 给出 `0` 。 `"42" == 0` 递归地变为 `42 == 0` ，这当然是 `false` 。

换句话说，值 **`"42"`** 既不 **`== true`** 也不 **`== false`** 。猛地一看，这看起来像句疯话。一个值怎么可能既不是truthy也不是falsy呢？

但这就是问题所在！你在问一个完全错误的问题。但这确实不是你的错，你的大脑在耍你。

`"42"` 的确是truthy，但是 `"42" == true` 根本就 不是在进行一个**boolean**测试/强制转换，不管你的大脑怎么说， `"42"` 没有 被强制转换为一个 `boolean` （ `true` ），而是 `true` 被强制转换为一个 `1` ，而后 `"42"` 被强制转换为 `42` 。

不管我们喜不喜欢， `ToBoolean` 甚至都没参与到这里，所以 `"42"` 的真假是与 `==` 操作无关的！

而有关的是要理解 `==` 比较算法对所有不同类型组合如何动作。当 `==` 的任意一边是一个 `boolean` 值时， `boolean` 总是首先被强制转换为一个 `number` 。

如果这对你来讲很奇怪，那么你不是一个人。我个人建议永远，永远，不要在任何情况下，使用 `==`

`true` 或 `== false` 。永远。

但时要记住，我在此说的仅与 `==` 有关。 `=== true` 和 `=== false` 不允许强制转换，所以它们没有 `ToNumber` 强制转换，因而是安全的。

考虑如下代码：

```
var a = "42";

// 不好（会失败的！）：
if (a == true) {
    // ..
}

// 也不该（会失败的！）：
if (a === true) {
    // ..
}

// 足够好（隐含地工作）：
if (a) {
    // ..
}

// 更好（明确地工作）：
if (!!a) {
    // ..
}

// 也很好（明确地工作）：
if (Boolean( a )) {
    // ..
}
```

如果你在你的代码中一直避免使用 `== true` 或 `== false` （也就是与 `boolean` 的宽松等价），你将永远不必担心这种真/假的思维陷阱。

比较：`null` 与 `undefined`

另一个 隐含 强制转换的例子可以在 `null` 和 `undefined` 值之间的 `==` 宽松等价中看到。又再一次引述ES5语言规范，条款11.9.3.2-3：

> 1. 如果*x*是*null*而*y*是*undefined*，返回*true*。
> 2. 如果*x*是*undefined*而*y*是*null*，返回*true*。

当使用 `==` 宽松等价比较 `null` 和 `undefined` ，它们是互相等价（也就是互相强制转换）的，而且在整个语言中不会等价于其他值了。

这意味着 `null` 和 `undefined` 对于比较的目的来说，如果你使用 `==` 宽松等价操作符来允许它们互相 隐含地 强制转换的话，它们可以被认为是不可区分的。

```
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` 之间的强制转换是安全且可预见的，而且在这样的检查中没有其他的值会给出测试成立的误判。我推荐使用这种强制转换来允许 `null` 和 `undefined` 是不可区分的，如此将它们作为相同的值对待。

比如：

```
var a = doSomething();

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

`a == null` 检查仅在 `doSomething()` 返回 `null` 或者 `undefined` 时才会通过，而在任何其他值的情况下将会失败，即便是 `0` ， `false` ，和 `""` 这样的falsy值。

这个检查的 明确 形式 —— 不允许任何强制转换 —— （我认为）没有必要地难看太多了（而且性能可能有点儿不好！）：

```
var a = doSomething();

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

在我看来， `a == null` 的形式是另一个用 隐含 强制转换增进了代码可读性的例子，而且是以一种可靠安全的方式。

比较：`object`与非`object`

如果一个 `object` / `function` / `array` 被与一个简单基本标量（ `string` ， `number` ，或 `boolean` ）进行比较，ES5语言规范在条款11.9.3.8-9中这样说道：

> 1. 如果*Type(x)*是一个*String*或者*Number*而*Type(y)*是一个*Object*，
>   返回比较 *x == ToPrimitive(y)* 的结果。
> 2. 如果*Type(x)*是一个*Object*而*Type(y)*是*String*或者*Number*，
>   返回比较 *ToPrimitive(x) == y* 的结果。

注意： 你可能注意到了，这些条款仅提到了 `String` 和 `Number` ，而没有 `Boolean` 。这是因为，正如我们早先引述的，条款11.9.3.6-7首先将任何出现的 `Boolean` 操作数强制转换为一个 `Number` 。

考虑如下代码：

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

a == b;    // true
```

值 `[ 42 ]` 的 `ToPrimitive` 抽象操作（见先前的“抽象值操作”部分）被调用，结果为值 `"42"` 。这里它就变为 `42 == "42"` ，我们已经讲解过这将变为 `42 == 42` ，所以 `a` 和 `b` 被认为是强制转换地等价。

提示： 我们在本章早先讨论过的 `ToPrimitive` 抽象操作的所以奇怪之处（ `toString()` ， `valueOf()` ），都在这里如你期望的那样适用。如果你有一个复杂的数据结构，而且你想在它上面定义一个 `valueOf()` 方法来为等价比较提供一个简单值的话，这将十分有用。

在第三章中，我们讲解了“拆箱”，就是一个基本类型值的 `object` 包装器（例如 `new String("abc")` 这样的形式）被展开，其底层的基本类型值（ `"abc"` ）被返回。这种行为与 `==` 算法中的 `ToPrimitive` 强制转换有关：

```
var a = "abc";
var b = Object( a );    // 与`new String( a )`相同

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

`a == b` 为 `true` 是因为 `b` 通过 `ToPrimitive` 强制转换为它的底层简单基本标量值 `"abc"` ，它与 `a` 中的值是相同的。

然而由于 `==` 算法中的其他覆盖规则，有些值是例外。考虑如下代码：

```
var a = null;
```

```
var b = Object( a );    // 与`Object()`相同
a == b;                 // false

var c = undefined;
var d = Object( c );    // 与`Object()`相同
c == d;                 // false

var e = NaN;
var f = Object( e );    // 与`new Number( e )`相同
e == f;                 // false
```

值 `null` 和 `undefined` 不能被装箱 —— 它们没有等价的对象包装器 —— 所以 `Object(null)` 就像 `Object()` 一样，它们都仅仅产生一个普通对象。

`NaN` 可以被封箱到它等价的 `Number` 对象包装器中，当 `==` 导致拆箱时，比较 `NaN == NaN` 会失败，因为 `NaN` 永远不会它自己相等（见第二章）。

边界情况

现在我们已经彻底检视了 `==` 宽松等价的 隐含 强制转换是如何工作的（从合理与惊讶两个方式），让我们召唤角落中最差劲儿的，最疯狂的情况，这样我们就能看到我们需要避免什么来防止被强制转换的bug咬到。

首先，让我们检视修改内建的原生prototype是如何产生疯狂的结果的：

一个拥有其他值的数字将会……

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

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

警告： `2 == 3` 不会掉到这个陷阱中，这是由于 `2` 和 `3` 都不会调用内建的 `Number.prototype.valueOf()` 方法，因为它们已经是基本 `number` 值，可以直接比较。然而， `new Number(2)` 必须通过 `ToPrimitive` 强制转换，因此调用 `valueOf()` 。

邪恶吧？当然。任何人都不应当做这样的事情。你 可以 这么做，这个事实有时被当成批评强制转换和 `==` 的根据。但这种沮丧是被误导的。JavaScript不会因为你能做这样的事情而 不好，是 做这样的事的开发者 不好。不要陷入“我的编程语言应当保护我不受我自己伤害”的谬论。

接下来，让我们考虑另一个刁钻的例子，它将前一个例子的邪恶带到另一个水平：

```
if (a == 2 && a == 3) {
```

```
    // ..
}
```

你可能认为这是不可能的，因为 `a` 绝不会 同时 等于 `2` 和 `3` 。但是“同时”是不准确的，因为第一个表达式 `a == 2` 严格地发生在 `a == 3` 之前。

那么，要是我们让 `a.valueOf()` 在每次被调用时拥有一种副作用，使它第一次被调用时返回 `2` 而第二次被调用时返回 `3` 呢？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." );
}
```

重申一次，这些都是邪恶的技巧。不要这么做。也不要用它们来抱怨强制转换。潜在地滥用一种机制并不是谴责这种机制的充分证据。避开这些疯狂的技巧，并坚持强制转换的合法与合理的用法就好了。

False-y 比较

关于 `==` 比较中 隐含 强制转换的最常见的抱怨，来自于falsy值互相比较时它们如何令人吃惊地动作。

为了展示，让我们看一个关于falsy值比较的极端例子的列表，来瞧瞧哪一个是合理的，哪一个是麻烦的：

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

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

```
false == [];            // true -- 噢！
false == {};            // false

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

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

在这24个比较的类表中，17个是十分合理和可预见的。比如，我们知道 `""` 和 `"NaN"` 是根本不可能相等的值，并且它们确实不会强制转换以成为宽松等价的，而 `"0"` 和 `0` 是合理等价的，而且确实强制转换为宽松等价。

然而，这些比较中的7个被标上了“噢！”。作为误判的成立，它们更像是会将你陷进去的坑。 `""` 和 `0` 绝对是有区别的不同的值，而且你很少会将它们作为等价的，所以它们的互相强制转换是一种麻烦。注意这里没有任何误判的不成立。

疯狂的情况

但是我们不必停留在此。我们可以继续寻找更能引起麻烦的强制转换：

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

噢，这看起来像是更高层次的疯狂，对吧！？你的大脑可能会欺骗你说，你在将一个truthy和falsy值比较，所以结果 `true` 是令人吃惊的，因为我们知道一个值不可能同时为truthy和falsy！

但这不是实际发生的事情。让我们把它分解一下。我们了解 `!` 一元操作符吧？它明确地使用 `ToBoolean` 规则将操作数强制转换为一个 `boolean` （而且它还会翻转真假性）。所以在 `[] == ![]` 执行之前，它实际上已经被翻译为了 `[] == false` 。我们已将在上面的列表中见过了这种形式（ `false == []` ），所以它的令人吃惊的结果对我们来说并不 新鲜。

其它的极端情况呢？

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

在关于 `ToNumber` 的讨论中我们说过，右手边的 `[2]` 和 `[null]` 值将会通过一个 `ToPrimitive` 强

制转换，以使我们可以方便地与左手边的简单基本类型值进行比较。因为 `array` 值的 `valueOf()` 只是返回 `array` 本身，强制转换会退到 `array` 的字符串化上。

对于第一个比较的右手边的值来说， `[2]` 将变为 `"2"` ，然后它会 `ToNumber` 强制转换为 `2` 。 `[null]` 就直接变成 `""` 。

那么， `2 == 2` 和 `"" == ""` 是完全可以理解的。

如果你的直觉依然不喜欢这个结果，那么你的沮丧实际上与你可能认为的强制转换无关。这其实是在抱怨 `array` 值在强制转换为 `string` 值时的默认 `ToPrimitive` 行为。很可能，你只是希望 `[2].toString()` 不返回 `"2"` ，或者 `[null].toString()` 不返回 `""` 。

但是这些 `string` 强制转换到底 应该 得出什么结果？对于 `[2]` 的 `string` 强制转换，除了 `"2"` 我确实想不出来其他合适的结果，也许是 `"[2]"` —— 但这可能会在其他的上下文中很奇怪！

你可以正确地制造另一个例子：因为 `String(null)` 变成了 `"null"` ，那么 `String([null])` 也应当变成 `"null"` 。这是个合理的断言。所以，它才是真正的犯人。

隐含 强制转换在这里并不邪恶。即使一个从 `[null]` 到 `string` 结果为 `""` 的 明确 强制转换也不。真正奇怪的是， `array` 值字符串化为它们内容的等价物是否有道理，和它是如何发生的。所以，应当将你沮丧的原因指向 `String( [..] )` 的规则，因为这里才是疯狂起源的地方。也许根本就不应该有 `array` 的字符串化强制转换？但这会在语言的其他部分造成许多的缺点。

另一个常被引用的著名的坑是：

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

正如我们早先讨论的空 `""` ， `"\n"` （或 `" "` ，或其他任何空格的组合）是通过 `ToNumber` 强制转换的，而且结果为 `0` 。你还希望空格被转换为其他的什么 `number` 值呢？明确的 `Number()` 给出 `0` 会困扰你吗？

空字符串和空格字符串可以转换为的，另一个真正唯一合理的 `number` 值是 `NaN` 。但这 真的 会更好吗？ `" " == NaN` 的比较当然会失败，但是不清楚我们是否真的 修正 了任何底层的问题。

真实世界中的JS程序由于 `0 == "\n"` 而失败的几率非常之低，而且这样的极端用例很容比避免。

在任何语言中，类型转换 总是 有极端用例 —— 强制转换也不例外。这里讨论的是特定的一组极端用例的马后炮，但不是针对强制转换整体而言的争论。

底线：你可能遇到的几乎所有 普通值 间的疯狂强制转换（除了像早先那样有意而为的 `valueOf()` 或 `toString()` 黑科技），都能归结为我们在上面指出的7中情况的短列表。

对比这24个疑似强制转换的坑，考虑另一个像这样的列表：

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

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

在这些非falsy，非极端的用例中（而且我们简直可以向这个列表中添加无限多个比较），强制转换完全是安全，合理，和可解释的。

可行性检查

好的，当我们深入观察 隐含的 强制转换时，我确实找到了一些疯狂的东西。难怪大多数开发者声称强制转换是邪恶而且应该避开的，对吧？

但是让我们退一步并做一下可行性检查。

通过大量比较，我们得到了一张7个麻烦的，坑人的强制转换的列表，但我们还得到了另一张（至少17个，但实际上有无限多个）完全正常和可以解释的强制转换的列表。

如果你在寻找一本“把孩子和洗澡水一起泼出去”的教科书，这就是了：由于一个仅有7个坑的列表，而抛弃整个强制转换（安全且有效的行为的无限大列表）。

一个更谨慎的反应是问，“我如何使用强制转换的 好的部分，而避开这几个 坏的部分 呢？”

让我们再看一次这个 坏 列表：

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

这个列表中7个项目的4个与 `== false` 比较有关，我们早先说过你应当 总是，总是 避免的。

现在这个列表缩小到了3个项目。

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

这些是你在一般的JavaScript程序中使用的合理的强制转换吗？在什么条件下它们会发生？

我不认为你在程序里有很大的可能要在一个 `boolean` 测试中使用 `== []` ，至少在你知道自己在做什么的情况下。你可能会使用 `== ""` 或 `== 0` ，比如：

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

如果你偶然调用了 `doSomething(0)` 或 `doSomething([])` ，你就会吓一跳。另一个例子：

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

再一次，如果你调用 `doSomething("",0)` 或 `doSomething([],"")` 时，它们会失败。

所以，虽然这些强制转换会咬到你的情况 可能 存在，而且你会小心地处理它们，但是它们可能不会在你的代码库中超级常见。

安全地使用隐含强制转换

我能给你的最重要的建议是：检查你的程序，并推理什么样的值会出现在 `==` 比较两边。为了避免这样的比较中的问题，这里有一些可以遵循的启发性规则：

1. 如果比较的任意一边可能出现 `true` 或者 `false` 值，那么就永远，永远不要使用 `==` 。
2. 如果比较的任意一边可能出现 `[]` ， `""` ，或 `0` 这些值，那么认真地考虑不使用 `==` 。

在这些场景中，为了避免不希望的强制转换，几乎可以确定使用 `===` 要比使用 `==` 好。遵循这两个简单的规则，可以有效地避免几乎所有可能会伤害你的强制转换的坑。

在这些情况下，使用更加明确/繁冗的方式会减少很多使你头疼的东西。

`==` 与 `===` 的问题其实可以更加恰当地表述为：你是否应当在比较中允许强制转换？

在许多情况下这样的强制转换会很有用，允许你更简练地表述一些比较逻辑（例如， `null` 和 `undefined` ）。

对于整体来说，相对有几个 隐含 强制转换会真的很危险的情况。但是在这些地方，为了安全起见，绝对要使用 `===` 。

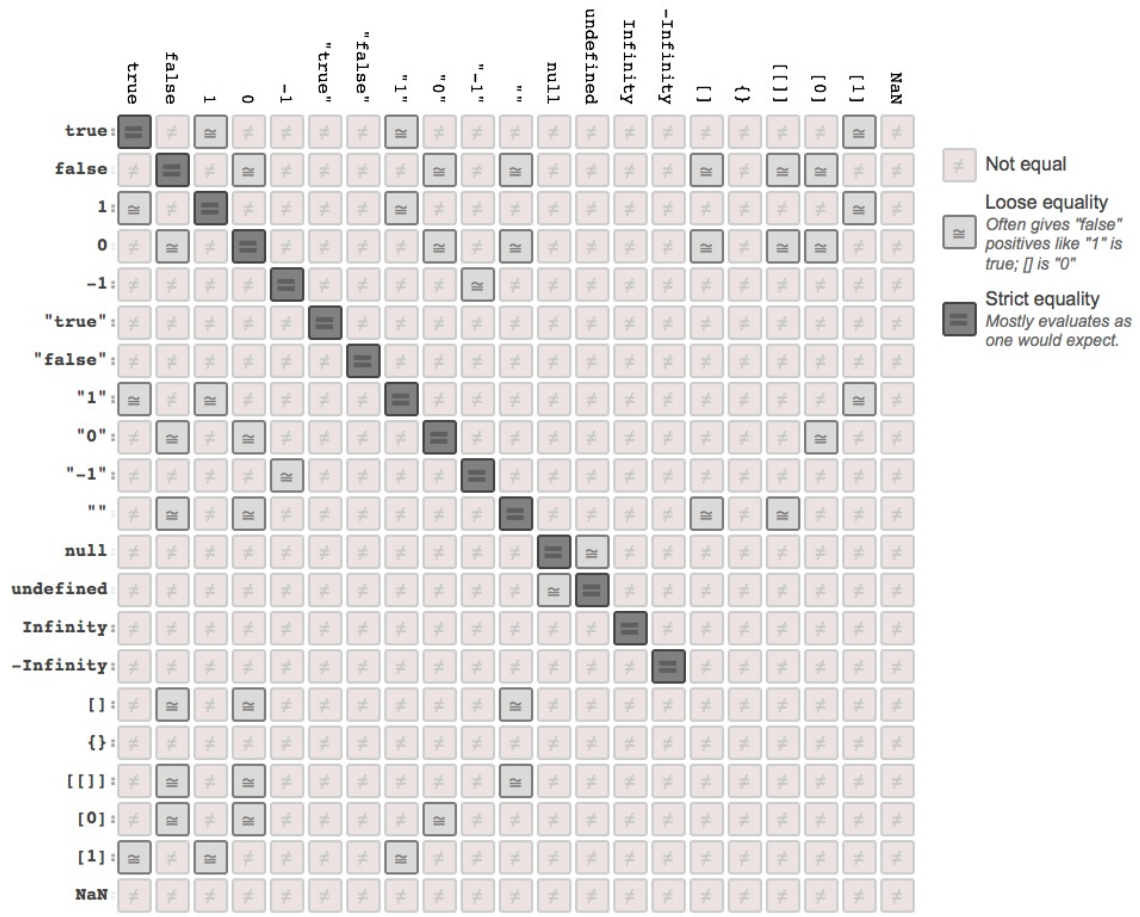
提示： 另一个强制转换保证 不会 咬到你的地方是 `typeof` 操作符。 `typeof` 总是将返回给你7中字符串之一（见第一章），它们中没有一个是空 `""` 字符串。这样，检查某个值的类型时不会有任何

情况与 隐含 强制转换相冲突。 `typeof x == "function"` 就像 `typeof x === "function"` 一样100%安全可靠。从字面意义上将，语言规范说这种情况下它们的算法是相同的。所以，不要只是因为你的代码工具告诉你这么做，或者（最差劲儿的）在某本书中有人告诉你 不要考虑它，而盲目地到处使用 `===` 。你掌管着你的代码的质量。

隐含 强制转换是邪恶和危险的吗？在几个情况下，是的，但总体说来，不是。

做一个负责任和成熟的开发者。学习如何有效并安全地使用强制转换（明确的 和 隐含的 两者）的力量。并教你周围的人也这么做。

这里是由Alex Dorey (@dorey on GitHub)制作的一个方便的表格，将各种比较进行了可视化：



出处：https://github.com/dorey/JavaScript-Equality-Table

抽象关系比较

抽象关系比较

虽然这部分的 隐含 强制转换经常不为人所注意，但无论如何考虑比较 `a < b` 时发生了什么是很重要的（和我们如何深入检视 `a == b` 类似）。

在ES5语言规范的11.8.5部分的“抽象关系型比较”算法，实质上把自己分成了两个部分：如果比较涉及两个 `string` 值要做什么（后半部分），和除此之外的其他值要做什么（前半部分）。

注意： 这个算法仅仅定义了 `a < b` 。所以， `a > b` 作为 `b < a` 处理。

这个算法首先在两个值上调用 `ToPrimitive` 强制转换，如果两个调用的返回值之一不是 `string` ，那么就使用 `ToNumber` 操作规则将这两个值强制转换为 `number` 值，并进行数字的比较。

举例来说：

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

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

注意： 早先讨论的关于 `-0` 和 `NaN` 在 `==` 算法中的类似注意事项也适用于这里。

然而，如果 `<` 比较的两个值都是 `string` 的话，就会在字符上进行简单的字典顺序（自然的字母顺序）比较：

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

a < b;    // false
```

`a` 和 `b` 不会 被强制转换为 `number` ，因为它们会在两个 `array` 的 `ToPrimitive` 强制转换后成为 `string` 。所以， `"42"` 将会与 `"043"` 一个字符一个字符地进行比较，从第一个字符开始，分别是 `"4"` 和 `"0"` 。因为 `"0"` 在字典顺序上 小于 `"4"` ，所以这个比较返回 `false` 。

完全相同的行为和推理也适用于：

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

```

```
4. a < b;    // false
```

这里， `a` 变成了 `"4,2"` 而 `b` 变成了 `"0,4,3"` ，而字典顺序比较和前一个代码段一模一样。

那么这个怎么样：

```
1. var a = { b: 42 };
2. var b = { b: 43 };
3.
4. a < b;    // ??
```

`a < b` 也是 `false` ，因为 `a` 变成了 `[object Object]` 而 `b` 变成了 `[object Object]` ，所以明显地 `a` 在字典顺序上不小于 `b` 。

但奇怪的是：

```
1. var a = { b: 42 };
2. var b = { b: 43 };
3.
4. a < b;    // false
5. a == b;    // false
6. a > b;    // false
7.
8. a <= b;    // true
9. a >= b;    // true
```

为什么 `a == b` 不是 `true` ？它们是相同的 `string` 值（ `"[object Object]"` ），所以看起来它们应当相等，对吧？不。回忆一下前面关于 `==` 如何与 `object` 引用进行工作的讨论。

那么为什么 `a <= b` 和 `a >= b` 的结果为 `true` ，如果 `a < b` 和 `a == b` 和 `a > b` 都是 `false` ？

因为语言规范说，对于 `a <= b` ，它实际上首先对 `b < a` 求值，然后反转那个结果。因为 `b < a` 也是 `false` ，所以 `a <= b` 的结果为 `true` 。

到目前为止你解释 `<=` 在做什么的方式可能是：“小于 或 等于”。而这可能完全相反，JS更准确地将 `<=` 考虑为“不大于”（ `!(a > b)` ，JS将它作为 `(!b < a)` ）。另外， `a >= b` 被解释为它首先被考虑为 `b <= a` ，然后实施相同的推理。

不幸的是，没有像等价那样的“严格的关系型比较”。换句话说，没有办法防止 `a < b` 这样的关系型比较发生 隐含的 强制转换，除非在进行比较之前就明确地确保 `a` 和 `b` 是同种类型。

使用与我们早先 `==` 与 `===` 合理性检查的讨论相同的推理方法。如果强制转换有帮助并且合理安全，比如比较 `42 < "43"` ，就使用它。另一方面，如果你需要在关系型比较上获得安全性，那么在使用 `<` （或 `>` ）之前，就首先 明确地强制转换 这些值。

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

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

复习

复习

在这一章中，我们将注意力转向了JavaScript类型转换如何发生，也叫 强制转换，按性质来说它要么是 明确的 要么是 隐含的。

强制转换的名声很坏，但它实际上在许多情况下很有帮助。对于负责任的JS开发者来说，一个重要的任务就是花时间去学习强制转换的里里外外，来决定哪一部分将帮助他们改进代码，哪一部分他们真的应该回避。

明确的 强制转换时这样一种代码，它很明显地有意将一个值从一种类型转换到另一种类型。它的益处是通过减少困惑来增强了代码的可读性和可维护性。

隐含的 强制转换是作为一些其他操作的“隐藏的”副作用而存在的，将要发生的类型转换并不明显。虽然看起来 隐含的 强制转换是 明确的 反面，而且因此是不好的（确实，很多人这么认为！），但是实际上 隐含的 强制转换也是为了增强代码的可读性。

特别是对于 隐含的，强制转换必须被负责地，有意识地使用。懂得为什么你在写你正在写的代码，和它是如何工作的。同时也要努力编写其他人容易学习和理解的代码。

第五章：文法

第五章：文法

我们想要解决的最后一个主要话题是JavaScript的语法如何工作（也称为它的文法）。你可能认为你懂得如何编写JS，但是语言文法的各个部分中有太多微妙的地方导致了困惑和误解，所以我们想要深入这些部分并搞清楚一些事情。

注意： 对于读者们来说，“文法（grammar）”一词不像“语法（syntax）”一词那么为人熟知。在许多意义上，它们是相似的词，描述语言如何工作的 规则。它们有一些微妙的不同，但是大部分对于我们在这里的讨论无关紧要。JavaScript的文法是一种结构化的方式，来描述语法（操作符，关键字，等等）如何组合在一起形成结构良好，合法的程序。换句话说，抛开文法来讨论语法将会忽略许多重要的细节。所以我们在本章中注目的内容的最准确的描述是 文法，尽管语言中的纯语法才是开发者们直接交互的。

链接

语句与表达式

语句与表达式

一个很常见的现象是，开发者们假定“语句（statement）”和“表达式（expression）”是大致等价的。但是这里我们需要区分它们俩，因为在我们的JS程序中它们有一些非常重要的区别。

为了描述这种区别，让我们借用一下你可能更熟悉的术语：英语。

一个“句子（sentence）”是一个表达想法的词汇的完整构造。它由一个或多个“短语（phrase）”组成，它们每一个都可以用标点符号或连词（“和”，“或”等等）连接。一个短语本身可以由更小的短语组成。一些短语是不完整的，而且本身没有太多含义，而另一些短语可以自成一句。这些规则总体地称为英语的 文法。

JavaScript文法也类似。语句就是句子，表达式就是短语，而操作符就是连词/标点。

JS中的每一个表达式都可以被求值而成为一个单独的，具体的结果值。举例来说：

```
1. var a = 3 * 6;
2. var b = a;
3. b;
```

在这个代码段中，`3 * 6` 是一个表达式（求值得值 `18`）。而第二行的 `a` 也是一个表达式，第三行的 `b` 也一样。对表达式 `a` 和 `b` 求值都会得到在那一时刻存储在这些变量中的值，也就偶然是 `18`。

另外，这三行的每一行都是一个包含表达式的语句。`var a = 3 * 6` 和 `var b = a` 称为“声明语句（declaration statments）”因为它们每一个都声明了一个变量（并选择性地给它赋值）。赋值 `a = 3 * 6` 和 `b = a`（除去 `var`）被称为赋值表达式（assignment expressions）。

第三行仅仅含有一个表达式 `b`，但是它本身也是一个语句（虽然不是非常有趣的一个！）。这一般

称为一个“表达式语句（expression statement）”。

语句完成值

一个鲜为人知的事实是，所有语句都有完成值（即使这个值只是 `undefined` ）。

你要如何做才能看到一个语句的完成值呢？

最明显的答案是把语句敲进你的浏览器开发者控制台，因为当你运行它时，默认地控制台会报告最近一次执行的语句的完成值。

让我们考虑一下 `var b = a` 。这个语句的完成值是什么？

`b = a` 赋值表达式给出的结果是被赋予的值（上面的 `18` ），但是 `var` 语句本身给出的结果是 `undefined` 。为什么？因为在语言规范中 `var` 语句就是这么定义的。如果你在你的控制台中敲入 `var a = 42` ，你会看到 `undefined` 被报告而不是 `42` 。

注意： 技术上讲，事情要比这复杂一些。在ES5语言规范，12.2部分的“变量语句”中， `VariableDeclaration` 算法实际上返回了一个值（一个包含被声明变量的名称的 `string` —— 诡异吧！？），但是这个值基本上被 `VariableStatement` 算法吞掉了（除了在 `for..in` 循环中使用），而这强制产生一个空的（也就是 `undefined` ）完成值。

事实上，如果你曾在你的控制台上（或者一个JavaScript环境的REPL —— read/evaluate/print/loop工具）做过很多的代码实验的话，你可能看到过许多不同的语句都报告 `undefined` ，而且你也许从来没理解它是什么和为什么。简单地说，控制台仅仅报告语句的完成值。

但是控制台打印出的完成值并不是我们可以在程序中使用的东西。那么我们该如何捕获完成值呢？

这是个更加复杂的任务。在我们解释 如何 之前，让我们先探索一下 为什么 你想这样做。

我们需要考虑其他类型的语句的完成值。例如，任何普通的 `{ .. }` 块儿都有一个完成值，即它所包含的最后一个语句/表达式的完成值。

考虑如下代码：

```
var b;

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

如果你将这段代码敲入你的控制台/REPL，你可能会看到它报告 `42` ，因为 `42` 是 `if` 块儿的完成值，它取自 `if` 的最后一个复制表达式语句 `b = 4 + 38` 。

换句话说，一个块儿的完成值就像 隐含地返回 块儿中最后一个语句的值。

注意： 这在概念上与CoffeeScript这样的语言很类似，它们隐含地从 `function` 中 `return` 值，这些值与函数中最后一个语句的值是相同的。

但这里有一个明显的问题。这样的代码是不工作的：

```
var a, b;

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

我们不能以任何简单的语法/文法来捕获一个语句的完成值并将它赋值给另一个变量（至少是还不能！）。

那么，我们能做什么？

警告： 仅用于演示的目的 —— 不要实际地在你的真实代码中做如下内容！

我们可以使用臭名昭著的 `eval(..)` （有时读成“evil”）函数来捕获这个完成值。

```
var a, b;

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

a;    // 42
```

啊呀呀。这太难看了。但是这好用！而且它展示了语句的完成值是一个真实的东西，不仅仅是在控制台中，还可以在我们的程序中被捕获。

有一个称为“do表达式”的ES7提案。这是它可能工作的方式：

```
var a, b;

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

a;    // 42
```

`do { .. }` 表达式执行一个块儿（其中有一个或多个语句），这个块儿中的最后一个语句的完成值将成为 `do` 表达式的完成值，它可以像展示的那样被赋值给 `a` 。

这里的大意是能够将语句作为表达式对待 —— 他们可以出现在其他语句内部 —— 而不必将它们包装在一个内联的函数表达式中，并实施一个明确的 `return ..` 。

到目前为止，语句的完成值不过是一些琐碎的事情。不过随着JS的进化它们的重要性可能会进一步提高，而且很有希望的是 `do { .. }` 表达式将会降低使用 `eval(..)` 这样的东西的冲动。

警告： 重复我刚才的训诫：避开 `eval(..)` 。真的。更多解释参见本系列的 作用域与闭包 一书。

表达式副作用

大多数表达式没有副作用。例如：

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

表达式 `a + 3` 本身并没有副作用，例如改变 `a` 。它有一个结果，就是 `5` ，而且这个结果在语句 `b = a + 3` 中被赋值给 `b` 。

一个最常见的（可能）带有副作用的表达式的例子是函数调用表达式：

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

var a = 1;
foo();        // 结果：`undefined`，副作用：改变 `a`
```

还有其他的副作用表达式。例如：

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

表达式 `a++` 有两个分离的行为。首先，它返回 `a` 的当前值，也就是 `42` （然后它被赋值给 `b` ）。但 接下来，它改变 `a` 本身的值，将它增加1。

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

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

许多开发者错误的认为 `b` 和 `a` 一样拥有值 `43` 。这种困惑源自没有完全考虑 `++` 操作符的副作用在 什么时候 发生。

`++` 递增操作符和 `--` 递减操作符都是一元操作符（见第四章），它们既可以用于后缀（“后面”）位置也可用于前缀（“前面”）位置。

```
var a = 42;

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

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

当 `++` 像 `++a` 这样用于前缀位置时，它的副作用（递增 `a` ）发生在值从表达式中返回 之前，而不是 `a++` 那样发生在 之后。

注意： 你认为 `++a++` 是一个合法的语法吗？如果你试一下，你将会得到一个 `ReferenceError` 错误，但为什么？因为有副作用的操作符 要求一个变量引用 来作为它们副作用的目标。对于 `++a++` 来说， `a++` 这部分会首先被求值（因为操作符优先级 —— 参见下面的讨论），它会给出 `a` 在递增 之前 的值。但然后它试着对 `++42` 求值，这将（如果你试一下）会给出相同的 `ReferenceError` 错误，因为 `++` 不能直接在 `42` 这样的值上施加副作用。

有时它会被错误地认为，你可以通过将 `a++` 包进一个 `( )` 中来封装它的 后 副作用，比如：

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

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

不幸的是， `( )` 本身不会像我们希望的那样，定义一个新的被包装的表达式，而它会在 `a++` 表达式的 后副作用 之 后 求值。事实上，就算它能， `a++` 也会首先返回 `42` ，而且除非你有另一个表达式在 `++` 的副作用之后对 `a` 再次求值，你也不会从这个表达式中得到 `43` ，于是 `b` 不会被赋值为 `43` 。

虽然，有另一种选择： `,` 语句序列逗号操作符。这个操作符允许你将多个独立的表达式语句连成一个单独的语句：

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

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

注意： `a++, a` 周围的 `( .. )` 是必需的。其原因的操作符优先级，我们将在本章后面讨论。

表达式 `a++, a` 意味着第二个 `a` 语句表达式会在第一个 `a++` 语句表达式的 后副作用 之 后 进行求值，这表明它为 `b` 的赋值返回 `43` 。

另一个副作用操作符的例子是 `delete` 。正如我们在第二章中展示的， `delete` 用于从一个 `object` 或一个 `array` 值槽中移除一个属性。但它经常作为一个独立语句被调用：

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

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

如果被请求的操作是合法/可允许的， `delete` 操作符的结果值为 `true` ，否则结果为 `false` 。但是这个操作符的副作用是它移除了属性（或数组值槽）。

注意： 我们说合法/可允许是什么意思？不存在的属性，或存在且可配置的属性（见本系列 *this与对象原型* 的第三章）将会从 `delete` 操作符中返回 `true` 。否则，其结果将是 `false` 或者一个错误。

副作用操作符的最后一个例子，可能既是明显的也是不明显的，是 `=` 赋值操作符。

考虑如下代码：

```
var a;

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

对于这个表达式来说， `a = 42` 中的 `=` 看起来似乎不是一个副作用操作符。但如果我们检视语句 `a = 42` 的结果值，会发现它就是刚刚被赋予的值（ `42` ），所以向 `a` 赋予的相同的值实质上是一种副作用。

提示： 相同的原因也适用于 `+=` ， `-=` 这样的复合赋值操作符的副作用。例如， `a = b += 2` 被处理为首先进行 `b += 2` （也就是 `b = b + 2` ），然后这个赋值的结果被赋予 `a` 。

这种赋值表达式（语句）得出被赋予的值的行为，主要在链式赋值上十分有用，就像这样：

```
var a, b, c;

a = b = c = 42;
```

这里， `c = 42` 被求值得出 `42` （带有将 `42` 赋值给 `c` 的副作用），然后 `b = 42` 被求值得

出 `42` （带有将 `42` 赋值给 `b` 的副作用），而最后 `a = 42` 被求值（带有将 `42` 赋值给 `a` 的副作用）。

警告： 一个开发者们常犯的错误是将链式赋值写成 `var a = b = 42` 这样。虽然这看起来是相同的东西，但它不是。如果这个语句发生在没有另外分离的 `var b` （在作用域的某处）来正式声明它的情况下，那么 `var a = b = 42` 将不会直接声明 `b` 。根据 `strict` 模式的状态，它要么抛出一个错误，要么无意中创建一个全局变量（参见本系列的 作用域与闭包）。

另一个要考虑的场景是：

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

    if (str) {
        // 找出所有的元音字母
        matches = str.match( /[aeiou]/g );

        if (matches) {
            return matches;
        }
    }
}

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

这可以工作，而且许多开发者喜欢这么做。但是使用一个我们可以利用赋值副作用的惯用法，可以通过将两个 `if` 语句组合为一个来进行简化：

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

    // 找出所有的元音字母
    if (str && (matches = str.match( /[aeiou]/g ))) {
        return matches;
    }
}

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

注意： `matches = str.match..` 周围的 `( .. )` 是必需的。其原因是操作符优先级，我们将在本章稍后的“操作符优先级”一节中讨论。

我偏好这种短一些的风格，因为我认为它明白地表示了两个条件其实是有关联的，而非分离的。但是与大多数JS中的风格选择一样，哪一种 更好 纯粹是个人意见。

上下文规则

在JavaScript文法规则中有好几个地方，同样的语法根据它们被使用的地方/方式不同意味着不同的东西。这样的东西可能，孤立的看，导致相当多的困惑。

我们不会在这里详尽地罗列所有这些情况，而只是指出常见的几个。

`{ .. }` 大括号

在你的代码中一对 `{ .. }` 大括号将主要出现在两种地方（随着JS的进化会有更多！）。让我们来看看它们每一种。

对象字面量

首先，作为一个 `object` 字面量：

```
// 假定有一个函数`bar()`的定义

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

我们怎么知道这是一个 `object` 字面量？因为 `{ .. }` 是一个被赋予给 `a` 的值。

注意： `a` 这个引用被称为一个“l-值”（也称为左手边的值）因为它是赋值的目标。 `{ .. }` 是一个“r-值”（也称为右手边的值）因为它仅被作为一个值使用（在这里作为赋值的源）。

标签

如果我们移除上面代码的 `var a =` 部分会发生什么？

```
// 假定有一个函数`bar()`的定义

{
    foo: bar()
}
```

许多开发者臆测 `{ .. }` 只是一个独立的没有被赋值给任何地方的 `object` 字面量。但事实上完全不同。

这里， `{ .. }` 只是一个普通的代码块儿。在JavaScript中拥有一个这样的独立 `{ .. }` 块儿并不是一个很惯用的形式（在其他语言中要常见得多！），但它是完美合法的JS文法。当与 `let` 块儿作用域声明组合使用时非常有用（见本系列的 作用域与闭包）。

这里的 `{ .. }` 代码块儿在功能上差不多与附着在一些语句后面的代码块儿是相同的，比

如 `for` / `while` 循环， `if` 条件，等等。

但如果它是一个一般代码块儿，那么那个看起来异乎寻常的 `foo: bar()` 语法是什么？它怎么会是合法的呢？

这是因为一个鲜为人知的（而且，坦白地说，不鼓励使用的）称为“打标签的语句”的JavaScript特性。 `foo` 是语句 `bar()` （这个语句省略了末尾的 `;` —— 见本章稍后的“自动分号”）的标签。但一个打了标签的语句有何意义？

如果JavaScript有一个 `goto` 语句，那么在理论上你就可以说 `goto foo` 并使程序的执行跳转到代码中的那个位置。 `goto` 通常被认为是一种糟糕的编码惯用形式，因为它们使代码更难于理解（也称为“面条代码”），所以JavaScript没有一般的 `goto` 语句是一件 非常好的事情。

然而，JS的确支持一种有限的，特殊形式的 `goto` ：标签跳转。 `continue` 和 `break` 语句都可以选择性地接受一个指定的标签，在这种情况下程序流会有些像 `goto` 一样“跳转”。考虑一下代码：

```
// 用`foo`标记的循环
foo: for (var i=0; i<4; i++) {
    for (var j=0; j<4; j++) {
        // 每当循环相遇，就继续外层循环
        if (j == i) {
            // 跳到被`foo`标记的循环的下一次迭代
            continue foo;
        }

        // 跳过奇数的乘积
        if ((j * i) % 2 == 1) {
            // 内层循环的普通（没有被标记的） `continue`
            continue;
        }

        console.log( i, j );
    }
}
// 1 0
// 2 0
// 2 1
// 3 0
// 3 2
```

注意： `continue foo` 不意味着“走到标记为‘foo’的位置并继续”，而是，“继续标记为‘foo’的循环，并进行下一次迭代”。所以，它不是一个 真正的 随意的 `goto` 。

如你所见，我们跳过了乘积为奇数的 `3 1` 迭代，而且被打了标签的循环跳转还跳过了 `1 1` 和 `2 2` 的迭代。

也许标签跳转的一个稍稍更有用的形式是，使用 `break __` 从一个内部循环里面跳出外部循环。没有带标签的 `break` ，同样的逻辑有时写起来非常尴尬：

```
// 用`foo`标记的循环
foo: for (var i=0; i<4; i++) {
    for (var j=0; j<4; j++) {
        if ((i * j) >= 3) {
            console.log( "stopping!", i, j );
            // 跳出被`foo`标记的循环
            break foo;
        }

        console.log( i, j );
    }
}
// 0 0
// 0 1
// 0 2
// 0 3
// 1 0
// 1 1
// 1 2
// stopping! 1 3
```

注意： `break foo` 不意味着“走到‘foo’标记的位置并继续”，而是，“跳出标记为‘foo’的循环/代码块儿，并继续它 后面 的部分”。不是一个传统意义上的 `goto` ，对吧？

对于上面的问题，使用不带标签的 `break` 将可能会牵连一个或多个函数，共享作用域中变量的访问，等等。它很可能要比带标签的 `break` 更令人糊涂，所以在这里使用带标签的 `break` 也许是更好的选择。

一个标签也可以用于一个非循环的块儿，但只有 `break` 可以引用这样的非循环标签。你可以使用带标签的 `break ___` 跳出任何被标记的块儿，但你不能 `continue ___` 一个非循环标签，也不能用一个不带标签的 `break` 跳出一个块儿。

```
function foo() {
    // 用`bar`标记的块儿
    bar: {
        console.log( "Hello" );
        break bar;
        console.log( "never runs" );
    }
    console.log( "World" );
}

foo();
```

```
// Hello
// World
```

带标签的循环/块儿极不常见，而且经常使人皱眉头。最好尽可能地避开它们；比如使用函数调用取代循环跳转。但是也许在一些有限的情况下它们会有用。如果你打算使用标签跳转，那么就确保使用大量注释在文档中记下你在做什么！

一个很常见的想法是，JSON是一个JS的恰当子集，所以一个JSON字符串（比如 `{"a":42}` —— 注意属性名周围的引号是JSON必需的！）被认为是一个合法的JavaScript程序。不是这样的！ 如果你试着把 `{"a":42}` 敲进你的JS控制台，你会得到一个错误。

这是因为语句标签周围不能有引号，所以 `"a"` 不是一个合法的标签，因此 `:` 不能出现在它后面。

所以，JSON确实是JS语法的子集，但是JSON本身不是合法的JS文法。

按照这个路线产生的一个极其常见的误解是，如果你将一个JS文件加载进一个 `<script src=..>` 标签，而它里面仅含有JSON内容的话（就像从API调用中得到那样），这些数据将作为合法的JavaScript被读取，但只是不能从程序中访问。JSON-P（将JSON数据包进一个函数调用的做法，比如 `foo({"a":42})` ）经常被说成是解决了这种不可访问性，通过向你程序中的一个函数发送这些值。

不是这样的！ 实际上完全合法的JSON值 `{"a":42}` 本身将会抛出一个JS错误，因为它被翻译为一个带有非法标签的语句块儿。但是 `foo({"a":42})` 是一个合法的JS，因为在它里面， `{"a":42}` 是一个被传入 `foo(..)` 的 `object` 字面量值。所以，更合适的说法是，**JSON-P使JSON成为合法的JS文法！**

块儿

另一个常为人所诟病的JS坑（与强制转换有关 —— 见第四章）是：

```
[] + {}; // "[object Object]"
{} + []; // 0
```

这看起来暗示着 `+` 操作符会根据第一个操作数是 `[]` 还是 `{}` 而给出不同的结果。但实际上这与它一点儿关系都没有！

在第一行中， `{}` 出现在 `+` 操作符的表达式中，因此被翻译为一个实际的值（一个空 `object` ）。第四章解释过， `[]` 被强制转换为 `""` 因此 `{}` 也会被强制转换为一个 `string` ： `"[object Object]"` 。

但在第二行中， `{}` 被翻译为一个独立的 `{}` 空代码块儿（它什么也不做）。块儿不需要分号来终结它们，所以这里缺少分号不是一个问题。最终， `+ []` 是一个将 `[]` 明确强制转换为 `number` 的表达式，而它的值是 `0` 。

对象解构

从ES6开始，你将看到 `{ .. }` 出现的另一个地方是“解构赋值”（更多信息参见本系列的 *ES6与未来*），确切地说是 `object` 解构。考虑下面的代码：

```
function getData() {
    // ..
    return {
        a: 42,
        b: "foo"
    };
}

var { a, b } = getData();

console.log( a, b ); // 42 "foo"
```

正如你可能看出来的， `var { a , b } = ..` 是ES6解构赋值的一种形式，它大体等价于：

```
var res = getData();
var a = res.a;
var b = res.b;
```

注意： `{ a, b }` 实际上是 `{ a: a, b: b }` 的ES6解构缩写，两者都能工作，但是人们期望短一些的 `{ a, b }` 能成为首选的形式。

使用一个 `{ .. }` 进行对象解构也可用于被命名的函数参数，这时它是同种类的隐含对象属性赋值的语法糖：

```
function foo({ a, b, c }) {
    // 不再需要：
    // var a = obj.a, b = obj.b, c = obj.c
    console.log( a, b, c );
}

foo( {
    c: [1,2,3],
    a: 42,
    b: "foo"
} );    // 42 "foo" [1, 2, 3]
```

所以，我们使用 `{ .. }` 的上下文环境整体上决定了它们的含义，这展示了语法和文法之间的区别。理解这些微妙之处以回避JS引擎进行意外的翻译是很重要的。

`else if` 和可选块儿

一个常见的误解是JavaScript拥有一个 `else if` 子句，因为你可以这么做：

```
if (a) {
    // ..
}
else if (b) {
    // ..
}
else {
    // ..
}
```

但是这里有一个JS文法隐藏的性质：它没有 `else if` 。但是如果附着在 `if` 和 `else` 语句后面的代码块儿仅包含一个语句时， `if` 和 `else` 语句允许省略这些代码块儿周围的 `{ }` 。毫无疑问，你以前已经见过这种现象很多次了：

```
if (a) doSomething( a );
```

许多JS编码风格指引坚持认为，你应当总是在一个单独的语句块儿周围使用 `{ }` ，就像：

```
if (a) { doSomething( a ); }
```

然而，完全相同的文法规则也适用于 `else` 子句，所以你经常编写的 `else if` 形式 实际上 被解析为：

```
if (a) {
    // ..
}
else {
    if (b) {
        // ..
    }
    else {
        // ..
    }
}
```

`if (b) { .. } else { .. }` 是一个紧随着 `else` 的单独的语句，所以你在它周围放不放一个 `{ }` 都可以。换句话说，当你使用 `else if` 的时候，从技术上讲你就打破了那个常见的编码风格指导的规则，而且只是用一个单独的 `if` 语句定义了你的 `else` 。

当然， `else if` 惯用法极其常见，而且减少了一级缩进，所以它很吸引人。无论你用哪种方式，就在你自己的编码风格指导/规则中明确地指出它，并且不要臆测 `else if` 是直接的文法规则。

操作符优先级

操作符优先级

就像我们在第四章中讲解的，JavaScript版本的 `&&` 和 `||` 很有趣，因为它们选择并返回它们的操作数之一，而不是仅仅得出 `true` 或 `false` 的结果。如果只有两个操作数和一个操作符，这很容易推理。

```
1. var a = 42;
2. var b = "foo";
3.
4. a && b;    // "foo"
5. a || b;    // 42
```

但是如果牵扯到两个操作符，和三个操作数呢？

```
1. var a = 42;
2. var b = "foo";
3. var c = [1,2,3];
4.
5. a && b || c; // ???
6. a || b && c; // ???
```

要明白这些表达式产生什么结果，我们就需要理解当在一个表达式中有多于一个操作符时，什么样的规则统治着操作符被处理的方式。

这些规则称为“操作符优先级”。

我打赌大多数读者都觉得自己已经很好地理解了操作符优先级。但是和我们在本系列丛书中讲解的其他一切东西一样，我们将拨弄这种理解来看看它到底有多扎实，并希望能在这个过程中学到一些新东西。

回想上面的例子：

```
1. var a = 42, b;
2. b = ( a++, a );
```

```
3.
4. a;    // 43
5. b;    // 43
```

要是我们移除了 `( )` 会怎样？

```
1. var a = 42, b;
2. b = a++, a;
3.
4. a;    // 43
5. b;    // 42
```

等一下！为什么这改变了赋给 `b` 的值？

因为 `,` 操作符要比 `=` 操作符的优先级低。所以， `b = a++, a` 被翻译为 `(b = a++), a` 。因为（如我们前面讲解的） `a++` 拥有 后副作用，赋值给 `b` 的值就是在 `++` 改变 `a` 之前的值 `42` 。

这只是为了理解操作符优先级所需的一个简单事实。如果你将要把 `,` 作为一个语句序列操作符使用，那么知道它实际上拥有最低的优先级是很重要的。任何其他的操作符都将要比 `,` 结合得更紧密。

现在，回想上面的这个例子：

```
1. if (str && (matches = str.match( /[aeiou]/g ))) {
2.     // ..
3. }
```

我们说过赋值语句周围的 `( )` 是必须的，但为什么？因为 `&&` 拥有的优先级比 `=` 更高，所以如果没有 `( )` 来强制结合，这个表达式将被作为 `(str && matches) = str.match..` 对待。但是这将是个错误，因为 `(str && matches)` 的结果将不是一个变量（在这里是 `undefined` ），而是一个值，因此它不能成为 `=` 赋值的左边！

好了，那么你可能认为你已经搞定操作符优先级了。

让我们移动到更复杂的例子（在本章下面几节中我们将一直使用这个例子），来 真正 测试一下你的理解：

```
1. var a = 42;
2. var b = "foo";
3. var c = false;
4.
5. var d = a && b || c ? c || b ? a : c && b : a;
6.
7. d;        // ??
```

好的，邪恶，我承认。没有人会写这样的表达式串，对吧？也许 不会，但是我们将使用它来检视将多个操作符链接在一起时的各种问题，而链接多个操作符是一个非常常见的任务。

上面的结果是 `42` 。但是这根本没意思，除非我们自己能搞清楚这个答案，而不是将它插进JS程序来让JavaScript搞定它。

让我们深入挖掘一下。

第一个问题 —— 你可能还从来没问过 —— 是，第一个部分（ `a && b || c` ）是像 `(a && b) || c` 那样动作，还是像 `a && (b || c)` 那样动作？你能确定吗？你能说服你自己它们实际上是不同的吗？

```
1. (false && true) || true;    // true
2. false && (true || true);    // false
```

那么，这就是它们不同的证据。但是 `false && true || true` 到底是如何动作的？答案是：

```
1. false && true || true;       // true
2. (false && true) || true;    // true
```

那么我们有了答案。 `&&` 操作符首先被求值，而 `||` 操作符第二被求值。

但这不是因为从左到右的处理顺序吗？让我们把操作符的顺序倒过来：

```
1. true || false && false;       // true
2.
3. (true || false) && false;    // false -- 不
4. true || (false && false);    // true -- 这才是胜利者!
```

现在我们证明了 `&&` 首先被求值，然后才是 `||` ，而且在这个例子中的顺序实际上是与一般希望的从左到右的顺序相反的。

那么什么导致了这种行为？操作符优先级。

每种语言都定义了自己的操作符优先级列表。虽然令人焦虑，但是JS开发者读过JS的列表却不太常见。

如果你熟知它，上面的例子一点儿都不会绊到你，因为你已经知道了 `&&` 要比 `||` 优先级高。但是我打赌有相当一部分读者不得不将它考虑一会。

注意： 不幸的是，JS语言规范没有将它的操作符优先级罗列在一个方便，单独的位置。你不得不通读并理解所有的文法规则。所以我们将试着以一种更方便的格式排列出更常见和更有用的部分。要得到完整的操作符优先级列表，参见MDN网站的“操作符优先级”(*
https://developer.mozilla.org/en-

US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence)。

短接

在第四章中，我们在一个边注中提到了操作符 `&&` 和 `||` 的“短接”性质。让我们更详细地重温它们。

对于 `&&` 和 `||` 两个操作符来说，如果左手边的操作数足够确定操作的结果，那么右手边的操作数将 不会被求值。故而，有了“短接”（如果可能，它就会取捷径退出）这个名字。

例如，说 `a && b` ，如果 `a` 是falsy `b` 就不会被求值，因为 `&&` 操作数的结果已经确定了，所以再去麻烦地检查 `b` 是没有意义的。同样的，说 `a || b` ，如果 `a` 是truthy，那么操作的结果就已经确定了，所以没有理由再去检查 `b` 。

这种短接非常有帮助，而且经常被使用：

```
function doSomething(opts) {
    if (opts && opts.cool) {
        // ..
    }
}
```

`opts && opts.cool` 测试的 `opts` 部分就像某种保护，因为如果 `opts` 没有被赋值（或不是一个 `object` ），那么表达式 `opts.cool` 就将抛出一个错误。 `opts` 测试失败加上短接意味着 `opts.cool` 根本不会被求值，因此没有错误！

相似地，你可以用 `||` 短接：

```
function doSomething(opts) {
    if (opts.cache || primeCache()) {
        // ..
    }
}
```

这里，我们首先检查 `opts.cache` ，如果它存在，我们就不会调用 `primeCache()` 函数，如此避免了潜在的不必要的工作。

更紧密的绑定

让我们把注意力转回前面全是链接的操作符的复杂语句的例子，特别是 `? :` 三元操作符的部分。 `? :` 操作对的优先级与 `&&` 和 `||` 操作符比起来是高还是低？

```
a && b || c ? c || b ? a : c && b : a
```

它是更像这样：

```
1. a && b || (c ? c || (b ? a : c) && b : a)
```

还是这样？

```
1. (a && b || c) ? (c || b) ? a : (c && b) : a
```

答案是第二个。但为什么？

因为 `&&` 优先级比 `||` 高，而 `||` 优先级比 `? :` 高。

所以，表达式 `(a && b || c)` 在 `? :` 参与之前被 首先 求值。另一种常见的解释方式是， `&&` 和 `||` 要比 `? :` “结合的更紧密”。如果倒过来成立的话，那么 `c ? c..` 将结合的更紧密，那么它就会如 `a && b || (c ? c..)` 那样动作（就像第一种选择）。

结合性

所以， `&&` 和 `||` 操作符首先集合，然后是 `? :` 操作符。但是多个同等优先级的操作符呢？它们总是从左到右或是从右到左地处理吗？

一般来说，操作符不是左结合的就是右结合的，这要看 分组是从左边发生还是从右边发生。

至关重要的是，结合性与从左到右或从右到左的处理 不是 同一个东西。

但为什么处理是从左到右或从右到左那么重要？因为表达式可以有副作用，例如函数调用：

```
1. var a = foo() && bar();
```

这里， `foo()` 首先被求值，然后根据表达式 `foo()` 的结果， `bar()` 可能会求值。如果 `bar()` 在 `foo()` 之前被调用绝对会得出不同的程序行为。

但是这个行为就是从左到右的处理（JavaScript中的默认行为！）— 它与 `&&` 的结合性无关。在这个例子中，因为这里只有一个 `&&` 因此没有相关的分组，所以根本谈不上结合性。

但是像 `a && b && c` 这样的表达式，分组将会隐含地发生，意味着不是 `a && b` 就是 `b && c` 会先被求值。

技术上讲， `a && b && c` 将会作为 `(a && b) && c` 处理，因为 `&&` 是左结合的（顺带一提， `||` 也是）。然而，右结合的 `a && (b && c)` 也表现出相同的行为。对于相同的值，相同的表达式是按照相同的顺序求值的。

注意： 如果假设 `&&` 是右结合的，它就会与你手动使用 `( )` 建立 `a && (b && c)` 这样的分组的处理方式一样。但是这仍然 不意味着 `c` 将会在 `b` 之前被处理。右结合性的意思 不是 从右到左

求值，它的意思是从右到左 分组。不管哪种方式，无论分组/结合性怎样，严格的求值顺序将是 `a` ，然后 `b` ，然后 `c` （也就是从左到右）。

因此，除了使我们对它们定义的讨论更准确以外， `&&` 和 `||` 是左结合这件事没有那么重要。

但事情不总是这样。一些操作符根据左结合性与右结合性将会做出不同的行为。

考虑 `? :` （“三元”或“条件”）操作符：

```
a ? b : c ? d : e;
```

`? :` 是右结合的，那么哪种分组表现了它将被处理的方式？

- `a ? b : (c ? d : e)`
- `(a ? b : c) ? d : e`

答案是 `a ? b : (c ? d : e)` 。不像上面的 `&&` 和 `||` ，在这里右结合性很重要，因为对于一些（不是全部！）值的组合来说 `(a ? b : c) ? d : e` 的行为将会不同。

一个这样的例子是：

```
true ? false : true ? true : true;        // false

true ? false : (true ? true : true);    // false
(true ? false : true) ? true : true;    // true
```

在其他的值的组合中潜伏着更加微妙的不同，即便他们的最终结果是相同的。考虑：

```
true ? false : true ? true : false;        // false

true ? false : (true ? true : false);    // false
(true ? false : true) ? true : false;    // false
```

在这个场景中，相同的最终结果暗示着分组是没有实际意义的。然而：

```
var a = true, b = false, c = true, d = true, e = false;

a ? b : (c ? d : e); // false, 仅仅对 `a` 和 `b` 求值
(a ? b : c) ? d : e; // false, 对 `a`, `b` 和 `e` 求值
```

这样，我们就清楚地证明了 `? :` 是右结合的，而且在这个操作符与它自己链接的方式上，右结合性是发挥影响的。

另一个右结合（分组）的例子是 `=` 操作符。回想本章早先的链式赋值的例子：

```
var a, b, c;

a = b = c = 42;
```

我们早先断言过，`a = b = c = 42` 的处理方式是，首先对 `c = 42` 赋值求值，然后是 `b = ..`，最后是 `a = ..`。为什么？因为右结合性，它实际上这样看待这个语句：`a = (b = (c = 42))`。

记得本章前面，我们的复杂赋值表达式的实例吗？

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

var d = a && b || c ? c || b ? a : c && b : a;

d;        // 42
```

随着我们使用优先级和结合性的知识把自己武装起来，我们应当可以像这样把这段代码分解为它的分组行为：

```
((a && b) || c) ? ((c || b) ? a : (c && b)) : a
```

或者，如果这样容易理解的话，可以用缩进表达：

```
(
  (a && b)
    ||
  c
)
  ?
(
  (c || b)
    ?
  a
    :
  (c && b)
)
  :
a
```

让我们解析它：

1. `(a && b)` 是 `"foo"`.
2. `"foo" || c` 是 `"foo"`.

3. 对于第一个 `?` 测试，`"foo"` 是truthy。
4. `(c || b)` 是 `"foo"`。
5. 对于第二个 `?` 测试，`"foo"` 是truthy。
6. `a` 是 `42`。

就是这样，我们搞定了！答案是 `42`，正如我们早先看到的。其实它没那么难，不是吗？

消除歧义

现在你应该对操作符优先级（和结合性）有了更好的把握，并对理解多个链接的操作符如何动作感到更适应了。

但还存在一个重要的问题：我们应当一直编写完美地依赖于操作符优先级/结合性的代码吗？我们应该仅在有必要强制一种不同的处理顺序时使用 `( )` 手动分组吗？

或者，另一方面，我们应当这样认识吗：虽然这样的规则 *实际上* 是可以学懂的，但是太多的坑让我们不得不忽略自动优先级/结合性？如果是这样，我们应当总是使用 `( )` 手动分组并移除对这些自动行为的所有依赖吗？

这种争论是非常主观的，而且和第四章中关于 *隐含* 强制转换的争论是强烈对称的。大多数开发者对这两个争论的感觉是一样的：要么他们同时接受这两种行为并使用它们编码，要么他们同时摒弃两种行为并坚持手动/明确的写法。

当然，在这个问题上，我们不能给出比我在第四章中给出的更绝对的答案。但我向你展示了利弊，并且希望促进了你更深刻的理解，以使你可以做出合理而不是人云亦云的决定。

在我看来，这里有一个重要的中间立场。我们应当将操作符优先级/结合性 *与* `( )` 手动分组两者混合进我们的程序 —— 我在第四章中对于 *隐含的* 强制转换的健康/安全用法做过同样的辩论，但当然不会没有界限地仅仅拥护它。

例如，对我来说 `if (a && b && c) ..` 是完全没问题的，而我不会为了明确表现结合性而写出 `if ((a && b) && c) ..`，因为我认为这过于繁冗了。

另一方面，如果我需要链接两个 `? :` 条件操作符，我会理所当然地使用 `( )` 手动分组来使我意图的逻辑表达的绝对清晰。

因此，我在这里的意见和在第四章中的相似：在操作符优先级/结合性可以使代码更短更干净的地方使用操作符优先级/结合性，在 `( )` 手动分组可以帮你创建更清晰的代码并减少困惑的地方使用 `( )` 手动分组

自动分号

自动分号

当JavaScript认为在你的JS程序中特定的地方有一个 `;` 时，就算你没在那里放一个 `;` ，它就会进行ASI（Automatic Semicolon Insertion —— 自动分号插入）。

为什么它这么做？因为就算你只省略了一个必需的 `;` ，你的程序就会失败。不是非常宽容。ASI允许JS容忍那些通常被认为是不需要 `;` 的特定地方省略 `;` 。

必须注意的是，ASI将仅在换行存在时起作用。分号不会被插入一行的中间。

基本上，如果JS解析器在解析一行时发生了解析错误（缺少一个应有的 `;` ），而且它可以合理的插入一个 `;` ，它就会这么做。什么样的地方对插入是合理的？仅在一个语句和这一行的换行之间除了空格和/或注释没有别的东西时。

考虑如下代码：

```
1. var a = 42, b
2. c;
```

JS应当将下一行的 `c` 作为 `var` 语句的一部分看待吗？如果在 `b` 和 `c` 之间的任意一个地方出现一个 `,` ，它当然会的。但是因为没有，所以JS认为在 `b` 后面有一个隐含的 `;` （在换行处）。如此 `c;` 就剩下来作为一个独立的表达式语句。

类似地：

```
1. var a = 42, b = "foo";
2.
3. a
4. b    // "foo"
```

这仍然是一个没有错误的合法程序，因为表达式语句也接受ASI。

有一些特定的地方ASI很有帮助，例如：

```
1. var a = 42;
2.
3. do {
```

```
    // ..
} while (a)    // <-- 这里需要;!
a;
```

文法要求 `do..while` 循环后面要有一个 `;` ，但是 `while` 或 `for` 循环后面则没有。但是大多数开发者都不记得它！所以ASI帮助性地介入并插入一个。

如我们在本章早先说过的，语句块儿不需要 `;` 终结，所以ASI是不必要的：

```
var a = 42;

while (a) {
    // ..
} // <-- 这里不需要;
a;
```

另一个ASI介入的主要情况是，与 `break` ， `continue` ， `return` ，和（ES6） `yield` 关键字：

```
function foo(a) {
    if (!a) return
    a *= 2;
    // ..
}
```

这个 `return` 语句的作用不会超过换行到 `a *= 2` 表达式，因为ASI认为 `;` 终结了 `return` 语句。当然， `return` 语句 可以 很容易地跨越多行，只要 `return` 后面不是除了换行外什么都没有就行。

```
function foo(a) {
    return (
        a * 2 + 3 / 12
    );
}
```

同样的道理也适用于 `break` ， `continue` ，和 `yield` 。

纠错

在JS社区中斗得最火热的 宗教战争 之一（除了制表与空格以外），就是是否应当严重/唯一地依赖ASI。

大多数，但不是全部的，分号是可选的，但是 `for ( .. ) ..` 循环的头部的两个 `;` 是必须的。

在这场争论的正方，许多开发者相信ASI是一种有用的机制，允许他们通过省略除了必须（很少几

个）以外的所有 `;` 写出更简洁（和更“美观”）的代码。他们经常断言因为ASI使许多 `;` 成为可选的，所以一个 不带它们 而正确编写的程序，与 带着它们 而正确编写的程序没有区别。

在这场争论的反方，许多开发者将断言有 太多 的地方可以成为意想不到的坑了，特别是对那些新来的，缺乏经验的开发者来说，无意间被魔法般插入的 `;` 改变了程序的含义。类似地，一些开发者将会争论如果他们省略了一个分号，这就是一个直白的错误，而且他们希望他们的工具（linter等等）在JS引擎背地里 纠正 它之前就抓住他。

让我分享一下我的观点。仔细阅读语言规范，会发现它暗示ASI是一个 纠错 过程。你可能会问，什么样的错误？明确地讲，是一个 解析器错误。换句话说，为了使解析器失败的少一些，ASI让它更宽容。

但是宽容什么？在我看来，一个 解析器错误 发生的唯一方式是，它被给予了一个不正确/错误的程序去解析。所以虽然ASI在严格地纠正解析器错误，但是它得到这样的错误的唯一方式是，程序首先就写错了 —— 在文法要求使用分号的地方忽略了它们。

所以，更直率地讲，当我听到有人声称他们想要省略“可选的分号”时，我的大脑就将它翻译为“我想尽量编写最能破坏解析器但依然可以工作的程序。”

我发现这种立场很荒唐，而且省几下键盘敲击和更“美观的代码”的观点是软弱无力的。

进一步讲，我不同意这和空格与制表符的争论是同一种东西 —— 那纯粹是表面上的 —— 我宁愿相信这是一个根本问题：是编写遵循文法要求的代码，还是编写依赖于文法异常但仅仅将之忽略不计的代码。

另一种看待这个问题的方式是，依赖ASI实质上将换行视为有意义的“空格”。像Python那样的其他语言中有真正的有意义的空格。但是就今天的JavaScript来说，认为它拥有有意义的换行真的合适吗？

我的意见是：在你知道分号是“必需的”地方使用分号，并且把你对**ASI**的臆测限制到最小。

不要光听我的一面之词。回到2012年，JavaScript的创造者Brendan Eich说过下面的话(http://brendaneich.com/2012/04/the-infernal-semicolon/):

> 这个故事的精神是：*ASI*是一种（正式地说）语法错误纠正过程。如果你在好像有一种普遍的有意义的换行的规则的前提下开始编码，你将会陷入麻烦。
>
> . .
>
> 如果回到*1995*年五月的那十天，我希望我使换行在*JS*中更有意义。
>
> . .
>
> 如果*ASI*好像给了*JS*有意义的换行，那么要小心不要使用它。

错误

错误

JavaScript不仅拥有不同的错误 子类型（ `TypeError` ， `ReferenceError` ， `SyntaxError` 等等），而且和其他在运行时期间发生的错误相比，它的文法还定义了在编译时被强制执行的特定错误。

尤其是，早就有许多明确的情况应当被作为“早期错误”（编译期间）被捕获和报告。任何直接的语法错误都是一个早期错误（例如， `a = ,` ），而且文法还定义了一些语法上合法但是无论怎样都不允许的东西。

因为你的代码还没有开始执行，这些错误不能使用 `try..catch` 捕获；它们只是会在你的程序进行解析/编译时导致失败。

提示： 在语言规范中没有要求浏览器（和开发者工具）到底应当怎样报告错误。所以在下面的错误例子中，对于哪一种错误的子类型会被报告或它包含什么样的错误消息，你可能会在各种浏览器中看到不同的形式，

一个简单的例子是正则表达式字面量中的语法。这里的JS语法没有错误，而是不合法的正则表达式将会抛出一个早期错误：

```
1. var a = /+foo/;        // 错误！
```

一个赋值的目标必须是一个标识符（或者一个产生一个或多个标识符的ES6解构表达式），所以一个像 `42` 这样的值在这个位置上是不合法的，因此可以立即被报告：

```
1. var a;
2. 42 = a;        // 错误！
```

ES5的 `strict` 模式定义了更多的早期错误。例如，在 `strict` 模式中，函数参数的名称不能重复：

```
1. function foo(a,b,a) { }                 // 还好
2.
3. function bar(a,b,a) { "use strict"; }    // 错误！
```

另一种 `strict` 模式的早期错误是，一个对象字面量拥有一个以上的同名属性：

```
(function(){
    "use strict";

    var a = {
        b: 42,
        b: 43
    };            // 错误！
})();
```

注意： 从语义上讲，这样的错误技术上不是 语法 错误，而是 文法 错误 —— 上面的代码段是语法上合法的。但是因为没有 `GrammarError` 类型，一些浏览器使用 `SyntaxError` 代替。

过早使用变量

ES6定义了一个（坦白地说，让人困惑地命名的）新的概念，称为TDZ（“Temporal Dead Zone” —— 时间死区）

TDZ指的是代码中还不能使用变量引用的地方，因为它还没有到完成它所必须的初始化。

对此最明白的例子就是ES6的 `let` 块儿作用域：

```
{
    a = 2;        // ReferenceError!
    let a;
}
```

赋值 `a = 2` 在变量 `a` （它确实是在 `{ .. }` 块儿作用域中）被声明 `let a` 初始化之前就访问它，所以 `a` 位于TDZ中并抛出一个错误。

有趣的是，虽然 `typeof` 有一个例外，它对于未声明的变量是安全的（见第一章），但是对于TDZ引用却没有这样的安全例外：

```
{
    typeof a;    // undefined
    typeof b;    // ReferenceError! (TDZ)
    let b;
}
```

函数参数值

函数参数值

另一个违反TDZ的例子可以在ES6的参数默认值（参见本系列的 *ES6与未来*）中看到：

```
var b = 3;

function foo( a = 42, b = a + b + 5 ) {
    // ..
}
```

在赋值中的 `b` 引用将在参数 `b` 的TDZ中发生（不会被拉到外面的 `b` 引用），所以它会抛出一个错误。然而，赋值中的 `a` 是没有问题的，因为那时参数 `a` 的TDZ已经过去了。

当使用ES6的参数默认值时，如果你省略一个参数，或者你在它的位置上传递一个 `undefined` 值的话，就会应用这个默认值。

```
function foo( a = 42, b = a + 1 ) {
    console.log( a, b );
}

foo();                  // 42 43
foo( undefined );       // 42 43
foo( 5 );               // 5 6
foo( void 0, 7 );       // 42 7
foo( null );            // null 1
```

注意： 在表达式 `a + 1` 中 `null` 被强制转换为值 `0`。更多信息参考第四章。

从ES6参数默认值的角度看，忽略一个参数和传递一个 `undefined` 值之间没有区别。然而，有一个办法可以在一些情况下探测到这种区别：

```
function foo( a = 42, b = a + 1 ) {
    console.log(
        arguments.length, a, b,
        arguments[0], arguments[1]
    );
}

foo();                  // 0 42 43 undefined undefined
foo( 10 );              // 1 10 11 10 undefined
```

```
foo( 10, undefined );    // 2 10 11 10 undefined
foo( 10, null );        // 2 10 null 10 null
```

即便参数默认值被应用到了参数 `a` 和 `b` 上，但是如果没有参数传入这些值槽，数组 `arguments` 也不会有任何元素。

反过来，如果你明确地传入一个 `undefined` 参数，在数组 `argument` 中就会为这个参数存在一个元素，但它将是 `undefined` ，并且与同一值槽中的被命名参数将被提供的默认值不同。

虽然ES6参数默认值会在数组 `arguments` 的值槽和相应的命名参数变量之间造成差异，但是这种脱节也会以诡异的方式发生在ES5中：

```
function foo(a) {
    a = 42;
    console.log( arguments[0] );
}

foo( 2 );    // 42 (链接了)
foo();        // undefined (没链接)
```

如果你传递一个参数， `arguments` 的值槽和命名的参数总是链接到同一个值上。如果你省略这个参数，就没有这样的链接会发生。

但是在 `strict` 模式下，这种链接无论怎样都不存在了：

```
function foo(a) {
    "use strict";
    a = 42;
    console.log( arguments[0] );
}

foo( 2 );    // 2 (没链接)
foo();        // undefined (没链接)
```

依赖于这样的链接几乎可以肯定是一个坏主意，而且事实上这种连接本身是一种抽象泄漏，它暴露了引擎的底层实现细节，而不是一个合适的设计特性。

`arguments` 数组的使用已经废弃了（特别是被ES6 `...` 剩余参数取代以后 —— 参见本系列的 *ES6与未来*），但这不意味着它都是不好的。

在ES6以前，要得到向另一个函数传递的所有参数值的数组， `arguments` 是唯一的办法，它被证实十分有用。你也可以安全地混用被命名参数和 `arguments` 数组，只要你遵循一个简单的规则：绝不同时引用一个被命名参数 和 它相应的 **`arguments`** 值槽。如果你能避开那种错误的实践，你就永远也不会暴露这种易泄漏的链接行为。

```
function foo(a) {
    console.log( a + arguments[1] ); // 安全!
}

foo( 10, 32 );    // 42
```

try..finally

`try..finally`

你可能很熟悉 `try..catch` 块儿是如何工作的。但是你有没有停下来考虑过可以与之成对出现的 `finally` 子句呢？事实上，你有没有意识到 `try` 只要求 `catch` 和 `finally` 两者之一，虽然如果有需要它们可以同时出现。

在 `finally` 子句中的代码 总是 运行的（无论发生什么），而且它总是在 `try` （和 `catch` ，如果存在的话）完成后立即运行，在其他任何代码之前。从一种意义上说，你似乎可以认为 `finally` 子句中的代码是一个回调函数，无论块儿中的其他代码如何动作，它总是被调用。

那么如果在 `try` 子句内部有一个 `return` 语句将会怎样？很明显它将返回一个值，对吧？但是调用端代码是在 `finally` 之前还是之后才收到这个值呢？

```
function foo() {
    try {
        return 42;
    }
    finally {
        console.log( "Hello" );
    }

    console.log( "never runs" );
}

console.log( foo() );
// Hello
// 42
```

`return 42` 立即运行，它设置好 `foo()` 调用的完成值。这个动作完成了 `try` 子句而 `finally` 子句接下来立即运行。只有这之后 `foo()` 函数才算完成，所以被返回的完成值交给 `console.log(..)` 语句使用。

对于 `try` 内部的 `throw` 来说，行为是完全相同的：

```
function foo() {
    try {
        throw 42;
    }
    finally {
        console.log( "Hello" );
    }
```

```
8.
9.      console.log( "never runs" );
10. }
11.
12. console.log( foo() );
13. // Hello
14. // Uncaught Exception: 42
```

现在，如果一个异常从 `finally` 子句中被抛出（偶然地或有意地），它将会作为这个函数的主要完成值进行覆盖。如果 `try` 块儿中的前一个 `return` 已经设置好了这个函数的完成值，那么这个值就会被抛弃。

```
1. function foo() {
2.     try {
3.         return 42;
4.     }
5.     finally {
6.         throw "Oops!";
7.     }
8.
9.     console.log( "never runs" );
10. }
11.
12. console.log( foo() );
13. // Uncaught Exception: Oops!
```

其他的诸如 `continue` 和 `break` 这样的非线性控制语句表现出与 `return` 和 `throw` 相似的行为是没什么令人吃惊的：

```
1. for (var i=0; i<10; i++) {
2.     try {
3.         continue;
4.     }
5.     finally {
6.         console.log( i );
7.     }
8. }
9. // 0 1 2 3 4 5 6 7 8 9
```

`console.log(i)` 语句在 `continue` 语句引起的每次循环迭代的末尾运行。然而，它依然是运行在更新语句 `i++` 之前的，这就是为什么打印出的值是 `0..9` 而非 `1..10` 。

注意： ES6在generator（参见本系列的 异步与性能）中增加了 `yield` 语句，generator从某些方面可以看作是中间的 `return` 语句。然而，和 `return` 不同的是，一个 `yield` 在generator被推进前不会完成，这意味着 `try { .. yield .. }` 还没有完成。所以附着在其上的 `finally` 子句将不

会像它和 `return` 一起时那样，在 `yield` 之后立即运行。

一个在 `finally` 内部的 `return` 有着覆盖前一个 `try` 或 `catch` 子句中的 `return` 的特殊能力，但是仅在 `return` 被明确调用的情况下：

```
function foo() {
    try {
        return 42;
    }
    finally {
        // 这里没有 `return ..`, 所以返回值不会被覆盖
    }
}

function bar() {
    try {
        return 42;
    }
    finally {
        // 覆盖前面的 `return 42`
        return;
    }
}

function baz() {
    try {
        return 42;
    }
    finally {
        // 覆盖前面的 `return 42`
        return "Hello";
    }
}

foo();    // 42
bar();    // undefined
baz();    // "Hello"
```

一般来说，在函数中省略 `return` 和 `return;` 或者 `return undefined;` 是相同的，但是在一个 `finally` 块儿内部， `return` 的省略不是用一个 `return undefined` 覆盖；它只是让前一个 `return` 继续生效。

事实上，如果将打了标签的 `break` （在本章早先讨论过）与 `finally` 相组合，我们真的可以制造一种疯狂：

```
function foo() {
```

```
    bar: {
        try {
            return 42;
        }
        finally {
            // 跳出标记为`bar`的块儿
            break bar;
        }
    }

    console.log( "Crazy" );

    return "Hello";
}

console.log( foo() );
// Crazy
// Hello
```

但是……别这么做。说真的。使用一个 `finally` + 打了标签的 `break` 实质上取消了 `return`，这是你在尽最大的努力制造最令人困惑的代码。我打赌没有任何注释可以拯救这段代码。

switch

`switch`

让我们简单探索一下 `switch` 语句，某种 `if..else if..else..` 语句链的语法缩写。

```
switch (a) {
    case 2:
        // 做一些事
        break;
    case 42:
        // 做另一些事
        break;
    default:
        // 这里是后备操作
}
```

如你所见，它对 `a` 求值一次，然后将结果值与每个 `case` 表达式进行匹配（这里只是一些简单的值表达式）。如果找到一个匹配，就会开始执行那个匹配的 `case` ，它将会持续执行直到遇到一个 `break` 或者遇到 `switch` 块儿的末尾。

这些可能不会令你吃惊，但是关于 `switch` ，有几个你以前可能从没注意过的奇怪的地方。

首先，在表达式 `a` 和每一个 `case` 表达式之间的匹配与 `===` 算法（见第四章）是相同的。 `switch` 经常在 `case` 语句中使用绝对值，就像上面展示的，因此严格匹配是恰当的。

然而，你也许希望允许宽松等价（也就是 `==` ，见第四章），而这么做你需要“黑”一下 `switch` 语句：

```
var a = "42";

switch (true) {
    case a == 10:
        console.log( "10 or '10'" );
        break;
    case a == 42:
        console.log( "42 or '42'" );
        break;
    default:
        // 永远不会运行到这里
}
// 42 or '42'
```

这可以工作是因为 `case` 子句可以拥有任何表达式（不仅是简单值），这意味着它将用这个表达式的结果与测试表达式（ `true` ）进行严格匹配。因为这里 `a == 42` 的结果为 `true` ，所以匹配成功。

尽管 `==` ， `switch` 的匹配本身依然是严格的，在这里是 `true` 和 `true` 之间。如果 `case` 表达式得出truthy的结果而不是严格的 `true` ，它就不会工作。例如如果在你的表达式中使用 `||` 或 `&&` 这样的“逻辑操作符”，这就可能咬到你：

```
var a = "hello world";
var b = 10;

switch (true) {
    case (a || b == 10):
        // 永远不会运行到这里
        break;
    default:
        console.log( "Oops" );
}
// Oops
```

因为 `(a || b == 10)` 的结果是 `"hello world"` 而不是 `true` ，所以严格匹配失败了。这种情况下，修改的方法是强制表达式明确成为一个 `true` 或 `false` ，比如 `case !!(a || b == 10):` （见第四章）。

最后， `default` 子句是可选的，而且它不一定非要位于末尾（虽然那是一种强烈的惯例）。即使是在 `default` 子句中，是否遇到 `break` 的规则也是一样的：

```
var a = 10;

switch (a) {
    case 1:
    case 2:
        // 永远不会运行到这里
    default:
        console.log( "default" );
    case 3:
        console.log( "3" );
        break;
    case 4:
        console.log( "4" );
}
// default
// 3
```

注意： 就像我们前面讨论的打标签的 `break` ， `case` 子句内部的 `break` 也可以被打标签。

这段代码的处理方式是，它首先通过所有的 `case` 子句，没有找到匹配，然后它回到 `default` 子句开始执行。因为这里没有 `break` ，它会继续走进已经被跳过的块儿 `case 3` ，在遇到那个 `break` 后才会停止。

虽然这种有些迂回的逻辑在JavaScript中是明显可能的，但是它几乎不可能制造出合理或易懂的代码。要对你自己是否想要创建这种环状的逻辑流程保持怀疑，如果你真的想要这么做，确保你留下了大量的代码注释来解释你要做什么！

复习

复习

JavaScript文法有相当多的微妙之处，我们作为开发者应当比平常多花一点儿时间来关注它。一点儿努力可以帮助你巩固对这个语言更深层次的知识。

语句和表达式在英语中有类似的概念 —— 语句就像句子，而表达式就像短语。表达式可以是纯粹的/自包含的，或者他们可以有副作用。

JavaScript文法层面的语义用法规则（也就是上下文），是在纯粹的语法之上的。例如，用于你程序中不同地方的 `{ }` 可以意味着块儿， `object` 字面量，（ES6）解构语句，或者（ES6）被命名的函数参数。

JavaScript操作符都有严格定义的优先级（哪一个操作符首先结合）和结合性（多个操作符表达式如何隐含地分组）规则。一旦你学会了这些规则，你就可以自己决定优先级/结合性是否是为了它们自己有利而 过于明确，或者它们是否会对编写更短，更干净的代码有所助益。

ASI（自动分号插入）是一种内建在JS引擎找中的解析器纠错机制，它允许JS引擎在特定的环境下，在需要 `;` 但是被省略了的地方，并且插入可以纠正解析错误时，插入一个 `;` 。有一场争论是关于这种行为是否暗示着大多数 `;` 都是可选的（而且为了更干净的代码可以/应当省略），或者是否它意味着省略它们是在制造JS引擎帮你扫清的错误。

JavaScript有几种类型的错误，但很少有人知道它有两种类别的错误：“早期”（编译器抛出的不可捕获的）和“运行时”（可以 `try..catch` 的）。所有在程序运行之前就使它停止的语法错误都明显是早期错误，但也有一些别的错误。

函数参数值与它们正式声明的命名参数之间有一种有趣的联系。明确地说，如果你不小心， `arguments` 数组会有一些泄漏抽象行为的坑。尽可能避开 `arguments` ，但如果你必须使用它，那就设法避免同时使用 `arguments` 中带有位置的值槽，和相同参数的命名参数。

附着在 `try` （或 `try..catch` ）上的 `finall` 在执行处理顺序上提供了一些非常有趣的能力。这些能力中的一些可以很有帮助，但是它也可能制造许多困惑，特别是在与打了标签的块儿组合使用时。像往常一样，为了更好更干净的代码而使用 `finally` ，不是为了显得更聪明或更糊涂。

`switch` 为 `if..else if..` 语句提供了一个不错的缩写形式，但是要小心许多常见的关于它的简化假设。如果你不小心，会有几个奇怪的地方绊倒你，但是 `switch` 手上也有一些隐藏的高招！

附录A：与环境混合的 JavaScript

你不懂JS：类型与文法

附录A：与环境混合的JavaScript

当你的JS代码在真实世界中运行时，除了我们在本书中完整探索过的核心语言机制以外，它还有好几种不同的行为方式。如果JS纯粹地运行在一个引擎中，那么它就会按照语言规范非黑即白地动作，是完全可以预测的。但是JS很可能总是运行在一个宿主环境的上下文中，这将会给你的代码带来某种程度的不可预测性。

例如，当你的代码与源自于其他地方的代码并肩运行时，或者当你的代码在不同种类的JS引擎（不只是浏览器）中运行时，有些事情的行为就可能不同。

我们将简要地探索这些问题中的一些。

Annex B （ECMAScript）

一个鲜为人知的事实是，这门语言的官方名称是ECMAScript（意指管理它的ECMA标准本体）。那么“JavaScript”是什么？JavaScript是这门语言常见的商业名称，当然，更恰当地说，JavaScript基本上是语言规范的浏览器实现。

官方的ECMAScript语言规范包含“Annex B”，它是为了浏览器中JS的兼容性，讨论那些与官方语言规范有偏差的特别部分。

考虑这些偏差部分的恰当方法是，它们仅在你的代码运行在浏览器中时才是确实会出现/合法的。如果

你的代码总是运行在浏览器中，那你就不会看到明显的不同。如果不是（比如它可以运行在node.js、Rhino中，等等），或者你不确定，那么就要小心对待。

兼容性上的主要不同是：

- 八进制数字字面量是允许的，比如在非`strict mode`下的`0123`（小数`83`）。
- `window.escape(..)`和`window.unescape(..)`允许你使用`%`分割的十六进制转义序列来转义或非转义字符串。例如：`window.escape( "?foo=97%&bar=3%" )`产生`"%3Ffoo%3D97%25%26bar%3D3%25"`
- `String.prototype.substr`与`String.prototype.substring`十分相似，除了第二个参数是`length`（要包含的字符数），而非结束（不含）的索引。

Web ECMAScript

Web ECMAScript语言规范(http://javascript.spec.whatwg.org/)涵盖了官方ECMAScript语言规范与当前浏览器中JavaScript实现之间的不同。

换言之，这些项目是浏览器的“必须品”（为了相互兼容），但是（在本书编写时）没有列在官方语言规范的“Annex B”部分是：

- `<!--`和`-->`是合法的单行注释分割符。
- `String.prototype` 拥有返回HTML格式化字符串的附加方法：`anchor(..)`、`big(..)`、`blink(..)`、`bold(..)`、`fixed(..)`、`fontcolor(..)`、`fontsize(..)`、`italics(..)`、`link(..)`、`small(..)`、`strike(..)`、和`sub(..)`。注意： 它们在实际应用中非常罕见，而且一般来说不鼓励使用，而是用其他内建DOM API或用户定义的工具取代。
- `RegExp`扩展：`RegExp.$1` .. `RegExp.$9`（匹配组）和`RegExp.lastMatch`/`RegExp["$&"]`（最近的匹配）。
- `Function.prototype`附加功能：`Function.prototype.arguments`（内部`arguments`对象的别名）和`Function.caller`（内部`arguments.caller`的别名）。注意： `arguments`和`arguments.caller`都被废弃了，所以你应当尽可能避免使用它们。这些别名更是这样 —— 不要使用它们！

注意： 其他的一些微小和罕见的偏差点没有包含在我们这里的列表中。有必要的话，更多详细信息可以参见外部的“Annex B”和“Web ECMAScript”文档。

一般来说，所有这些不同点都很少被使用，所以这些与语言规范有出入的地方不是什么重大问题。只是如果你依赖于其中任何一个的话，要小心。

宿主对象

JS中变量的行为有一些广为人知的例外 —— 当它们是被自动定义，或由持有你代码的环境（浏览器等）创建并提供给JS时 —— 也就是所谓的“宿主对象”（包括`object`和`function`两者）。

例如：

```
var a = document.createElement( "div" );

typeof a;                                   // "object" -- 意料之中的
Object.prototype.toString.call( a );    // "[object HTMLDivElement]"

a.tagName;                                  // "DIV"
```

`a` 不仅是一个 `object` ，而且是一个特殊的宿主对象，因为它是一个DOM元素。它拥有一个不同的内部 `[[Class]]` 值（ `"HTMLDivElement"` ），而且带有预定义的（而且通常是不可更改的）属性。

另一个已经在第四章的“Falsy对象”一节中探讨过的同样的怪异之处是：存在这样一些对象，当被强制转换为 `boolean` 时，它们将（令人糊涂地）被转换为 `false` 而不是预期的 `true` 。

另一些需要小心的宿主对象行为包括：

- 不能访问像 `toString()` 这样的 `object` 内建方法
- 不可覆盖
- 拥有特定的预定义只读属性
- 拥有一些 `this` 不可被重载为其他对象的方法
- 其他……

为了使我们的JS代码与它外围的环境一起工作，宿主对象至关重要。但在你与宿主对象交互时是要特别注意，并且在推测它的行为时要小心，因为它们经常与普通的JS `object` 不符。

一个尽人皆知的你可能经常与之交互的宿主对象的例子，就是 `console` 对象和他的各种函数（ `log(..)` 、 `error(..)` 等等）。 `console` 对象是由 宿主环境 特别提供的，所以你的代码可以与之互动来进行各种开发相关的输出任务。

在浏览器中， `console` 与开发者工具控制台的显示相勾连，因此在node.js和其他服务器端JS环境中， `console` 一般连接着JavaScript环境系统进程的标准输出流（ `stdout` ）和标准错误流（ `stderr` ）。

全局DOM变量

你可能知道，在全局作用域中声明变量（用或者不用 `var` ）不仅会创建一个全局变量，还会创建它的镜像：在 `global` 对象（浏览器中的 `window` ）上的同名属性。

但少为人知的是，（由于浏览器的遗留行为）使用 `id` 属性创建DOM元素会创建同名的全局变量。例如：

```
<div id="foo"></div>
```

和：

```
if (typeof foo == "undefined") {
    foo = 42;        // 永远不会运行
}

console.log( foo );    // HTML元素
```

你可能臆测只有JS代码会创建这样的变量，并习惯于在这样假定的前提下进行全局变量检测（使用 `typeof` 或者 `.. in window` 检查），但是如你所见，你的宿主HTML页面的内容也会创建它们，如果你不小心它们就可以轻而易举地摆脱你的存在性检查。

这就是另一个你为什么应该尽全力避免使用全局变量的原因，如果你不得不这样做，那就使用不太可能冲突的变量名。但是你还是需要确认它不会与HTML的内容以及其他的代码相冲突。

原生原型

最广为人知的，经典的JavaScript 最佳实践 智慧之一是：永远不要扩展原生原型。

当你将方法或属性添加到 `Array.prototype` 时，无论你想出什么样的（还）不存在于 `Array.prototype` 上名称，如果它是有用的、设计良好的、并且被恰当命名的新增功能，那么它就有很大的可能性被最终加入语言规范 —— 这种情况下你的扩展就处于冲突之中。

这里有一个真实地发生在我身上的例子，很好地展示了这一点。

那时我正在为其他网站建造一个可嵌入的控件，而且我的控件依赖于JQuery（虽然任何框架都很可能遭受这样的坑）。它几乎可以在每一个网站上工作，但是我们碰到了一个它会完全崩溃的网站。

经过差不多一周的分析/调试之后，我发现这个出问题的网站有这样一段代码，埋藏在它的一个遗留文件的深处：

```
// Netscape 4 没有 Array.push
Array.prototype.push = function(item) {
    this[this.length] = item;
};
```

除了那疯狂的注释（谁还会关心Netscape 4！？），它看起来很合理，对吧？

问题是，在这段 Netscape 4 时代的代码被编写之后的某个时点，`Array.prototype.push` 被加入了语言规范，但是被加入的东西与这段代码是不兼容的。标准的 `push(..)` 允许一次加入多个项目，而这个黑进来的东西会忽略后续项目。

基本上所有的JS框架都有这样的代码 —— 依赖于带有多个元素的 `push(..)` 。在我的例子中，我在围绕着一个完全被毁坏的CSS选择器引擎进行编码。但是可以料想到还有其他十几处可疑的地方。

一开始编写这个 `push(..)` 黑科技的开发者称它为 `push` ，这种直觉很正确，但是没有预见到添加多个元素。当然他们的初衷是好的，但是也埋下了一个地雷，当我差不多在10年之后路过时才不知不觉地踩上。

这里要吸取几个教训。

第一，不要扩展原生类型，除非你绝对确信你的代码将是运行在那个环境中的唯一代码。如果你不能100%确信，那么扩展原生类型就是危险的。你必须掂量掂量风险。

其次，不要无条件地定义扩展（因为你可能意外地覆盖原生类型）。就这个特定的例子，用代码说话就是：

```
if (!Array.prototype.push) {
    // Netscape 4 没有 Array.push
    Array.prototype.push = function(item) {
        this[this.length] = item;
    };
}
```

`if` 守护语句将会仅在JS环境中不存在 `push()` 时才定义那个 `push()` 黑科技。在我的情况中，这可能就够了。但即便是这种方式也不是没有风险：

1. 如果网站的代码（为了某些疯狂的理由！）有赖于忽略多个项目的`push(..)`，那么几年以后当标准的`push(..)`推出时，那些代码将会坏掉。
2. 如果有其他库被引入，并在这个`if`守护之前就黑进了`push(..)`，而且还是以一种不兼容的方式，那么它就在那一刻毁坏了这个网站。

这里的重点，坦白地讲，是一个没有得到JS开发者们足够重视的有趣问题：如果在你代码运行的环境中，你的代码不是唯一的存在，那么 你应该依赖于任何原生的内建行为吗?

严格的答案是 不，但这非常不切实际。你的代码通常不会为所有它依赖的内建行为重新定义它自己的、不可接触的私有版本。即便你 能，那也是相当的浪费。

那么，你应当为内建行为进行特性测试，以及为了验证它能如你预期的那样工作而进行兼容性测试吗？但如果测试失败了 —— 你的代码应当拒绝运行吗？

```
// 不信任 Array.prototype.push
(function(){
    if (Array.prototype.push) {
        var a = [];
        a.push(1,2);
        if (a[0] === 1 && a[1] === 2) {
            // 测试通过，可以安全使用！
            return;
        }
    }
```

```
11.
12.     throw Error(
13.         "Array#push() is missing/broken!"
14.     );
15. })();
```

理论上，这貌似有些道理，但是为每一个内建方法设计测试还是非常不切实际。

那么，我们应当怎么做？我们应当 信赖但验证（特性测试和兼容性测试）每一件事吗？我们应当假设既存的东西是符合规范的并让（由他人）造成的破坏任意传播吗？

没有太好的答案。可以观察到的唯一事实是，扩展原生原型是这些东西咬到你的唯一方式。

如果你不这么做，而且在你的应用程序中也没有其他人这么做，那么你就是安全的。否则，你就应当多多少少建立一些怀疑的、悲观的机制、并对可能的破坏做好准备。

在所有已知环境中，为你的代码准备一整套单元/回归测试是发现一些前述问题的方法，但是它不会对这些冲突为你做出任何实际的保护。

Shims/Polyfills

人们常说，扩展一个原生类型唯一安全的地方是在一个（不兼容语言规范的）老版本环境中，因为它不太可能再改变了 —— 带有新语言规范特性的新浏览器会取代老版本浏览器，而非改良它们。

如果你能预见未来，而且确信未来的标准将是怎样，比如 `Array.prototype.foobar` ，那么现在就制造你自己的兼容版本来使用就是完全安全的，对吧？

```
1. if (!Array.prototype.foobar) {
2.     // 愚蠢，愚蠢
3.     Array.prototype.foobar = function() {
4.         this.push( "foo", "bar" );
5.     };
6. }
```

如果已经有了 `Array.prototype.foobar` 的规范，而且规定的行为与这个逻辑等价，那么你定义这样的代码段就十分安全，在这种情况下它通常称为一个“polyfill（填补）”（或者“shim（垫片）”）。

在你的代码库中引入这样的代码，对给那些没有更新到最新规范的老版本浏览器环境打“补丁”非常有用。为所有你支持的环境创建可预见的代码，使用填补是非常好的方法。

提示： ES5-Shim (https://github.com/es-shims/es5-shim) 是一个将项目代码桥接至ES5基准线的完整的shims/polyfills集合，相似地，ES6-Shim (https://github.com/es-shims/es6-shim) 提供了ES6新增的新API的shim。虽然API可以被填补，但新的语法通常是不能的。要桥接语法的部分，你将还需要使用一个ES6到ES5的转译器，比如Traceur (https://github.com/google/traceur-compiler/wiki/GettingStarted)。

如果有一个即将到来的标准，而且关于它叫什么名字和它将如何工作的讨论达成了一致，那么为了兼容面向未来的标准提前创建填补，被称为“prollyfill（probably-fill —— 预填补）”。

真正的坑是某些标准行为不能被（完全）填补/预填补。

在开发者社区中有这样一种争论：对于常见的情况一个部分地填补是否是可接受的，或者如果一个填补不能100%地与语言规范兼容是否应当避免它。

许多开发者至少会接受一些常见的部分填补（例如 `Object.create(..)`），因为没有被填补的部分是他们不管怎样都不会用到的。

一些开发者相信，包围着 polyfill/shim 的 `if` 守护语句应当引入某种形式的一致性测试，在既存的方法缺失或者测试失败时取代它。这额外的一层兼容性测试有时被用于将“shim”（兼容性测试）与“polyfill”（存在性测试）区别开。

这里的要点是，没有绝对 正确 的答案。即使是在老版本环境中“安全地”扩展原生类型，也不是100%安全的。在其他人代码存在的情况下依赖于（可能被扩展过的）原生类型也是一样。

在这两种情况下都应当小心地使用防御性的代码，并在文档中大量记录它的风险。

`<script>`

大多数通过浏览器使用的网站/应用程序都将它们的代码包含在一个以上的文件中，在一个页面中含有几个或好几个分别加载这些文件的 `<script src=..></script>` 元素，甚至几个内联的 `<script> .. </script>` 元素也很常见。

但这些分离的文件/代码段是组成分离的程序，还是综合为一个JS程序？

（也许令人吃惊）现实是它们在极大程度上，但不是全部，像独立的JS程序那样动作。

它们所 共享 的一个东西是一个单独的 `global` 对象（在浏览器中是 `window`），这意味着多个文件可以将它们的代码追加到这个共享的名称空间中，而且它们都是可以交互的。

所以，如果一个 `script` 元素定义了一个全局函数 `foo()`，当第二个 `script` 运行时，它就可以访问并调用 `foo()`，就好像它自己已经定义过了这个函数一样。

但是全局变量作用域 提升（参见本系列的 作用域与闭包）不会跨越这些界线发生，所以下面的代码将不能工作（因为 `foo()` 的声明还没有被声明过），无论它们是否是内联的 `<script> .. </script>` 元素还是外部加载的 `<script src=..></script>` 文件：

```
<script>foo();</script>

<script>
  function foo() { .. }
</script>
```

但是这两个都将 可以 工作：

```
<script>
  foo();
  function foo() { .. }
</script>
```

或者：

```
<script>
  function foo() { .. }
</script>

<script>foo();</script>
```

另外，如果在一个 `script` 元素（内联或者外部的）中发生了一个错误，一个分离的独立的JS程序将会失败并停止，但是任何后续的 `script` 都将会（依然在共享的 `global` 中）畅通无阻地运行。

你可以在你的代码中动态地创建 `script` 元素，并将它们插入到页面的DOM中，它们之中的代码基本上将会像从一个分离的文件中普通地加载那样运行：

```
var greeting = "Hello World";

var el = document.createElement( "script" );

el.text = "function foo(){ alert( greeting );\
 } setTimeout( foo, 1000 );";

document.body.appendChild( el );
```

注意： 当然，如果你试一下上面的代码段并将 `el.src` 设置为某些文件的URL，而非将 `el.text` 设置为代码内容，你就会动态地创建一个外部加载的 `<script src=..></script>` 元素。

内联代码块中的代码，与在外部文件中的相同的代码之间的一个不同之处是，在内联的代码块中，字符 `</script>` 的序列不能一起出现，因为（无论它在哪里出现）它将会被翻译为代码块的末尾。所以，小心这样的代码：

```
<script>
  var code = "<script>alert( 'Hello World' )</script>";
</script>
```

它看起来无害，但是在 `string` 字面量中出现的 `</script>` 将会不正常地终结script块，造成一个错误。绕过它最常见的一个方法是：

```
"</sc" + "ript>";
```

另外要小心的是，一个外部文件中的代码将会根据和文件一起被提供（或默认的）的字符集编码（UTF-8、ISO-8859-8等等）来翻译，但在内联在你HTML页面中的一个 `script` 元素中的相同代码将会根据这个页面的（或它默认的）字符集编码来翻译。

警告： `charset` 属性在内联script元素中不能工作。

关于内联 `script` 元素，另一个被废弃的做法是在内联代码的周围引入HTML风格或X(HT)ML风格的注释，就像：

```
<script>
<!--
alert( "Hello" );
//-->
</script>

<script>
<!--//--><![CDATA[//><!--
alert( "World" );
//--><!]]>
</script>
```

这两种东西现在完全是不必要的了，所以如果你还在这么做，停下！

注意： 实际上纯粹是因为这种老技术，JavaScript才把 `<!--` 和 `-->` （HTML风格的注释）两者都被规定为合法的单行注释分隔符（ `var x = 2; <!-- valid comment` 和 `--> another valid line comment` ）。永远不要使用它们。

保留字

ES5语言规范在第7.6.1部分中定义了一套“保留字”，它们不能被用作独立的变量名。技术上讲，有四个类别：“关键字”，“未来保留字”， `null` 字面量，以及 `true` / `false` 布尔字面量。

像 `function` 和 `switch` 这样的关键字是显而易见的。像 `enum` 之类的未来保留字，虽然它们中的许多（ `class` 、 `extends` 等等）现在都已经实际被ES6使用了；但还有另外一些像 `interface` 之类的仅在strict模式下的保留字。

StackOverflow用户“art4theSould”创造性地将这些保留字编成了一首有趣的小诗(http://stackoverflow.com/questions/26255/reserved-keywords-in-javascript/12114140#12114140):

> *Let this long package float,*
> *Goto private class if short.*

> *While protected with debugger case,*
> *Continue volatile interface.*
> *Instanceof super synchronized throw,*
> *Extends final export throws.*
>
> *Try import double enum?*
>
> - *False, boolean, abstract function,*
>   *Implements typeof transient break!*
>   *Void static, default do,*
>   *Switch int native new.*
>   *Else, delete null public var*
>   *In return for const, true, char*
>   *…Finally catch byte.*

注意： 这首诗包含ES3中的保留字（ `byte` 、 `long` 等等），它们在ES5中不再被保留了。

在ES5以前，这些保留字也不能被用于对象字面量中的属性名或键，但这种限制已经不复存在了。

所以，这是不允许的：

```
var import = "42";
```

但这是允许的：

```
var obj = { import: "42" };
console.log( obj.import );
```

你应当小心，有些老版本的浏览器（主要是老IE）没有完全地遵循这些规则，所以有些将保留字用作对象属性名的地方任然会造成问题。小心地测试所有你支持的浏览器环境。

实现的限制

JavaScript语言规范没有在诸如函数参数值的个数，或者字符串字面量的长度上做出随意的限制，但是由于不同引擎的实现细节，无论如何这些限制是存在的。

例如：

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

var nums = [];
```

```
for (var i=1; i < 100000; i++) {
    nums.push(i);
}

addAll( 2, 4, 6 );                // 12
addAll.apply( null, nums );       // 应该是：499950000
```

在某些JS引擎中，你将会得到正确答案 `499950000` ，但在另一些引擎中（比如Safari 6.x），你会得到一个错误：“RangeError: Maximum call stack size exceeded.”

已知存在的其他限制的例子：

- 在字符串字面量（不是一个字符串变量）中允许出现的最大字符个数
- 在一个函数调用的参数值中可以发送的数据的大小（字节数，也称为栈的大小）
- 在一个函数声明中的参数数量
- 没有经过优化的调用栈最大深度（比如，使用递归时）：从一个函数到另一个函数的调用链能有多长
- JS程序可以持续运行并阻塞浏览器的秒数
- 变量名的最大长度
- …

遭遇这些限制不是非常常见，但你应当知道这些限制存在并确实会发生，而且重要的是它们因引擎不同而不同。

复习

我们知道并且可以依赖于这样的事实：JS语言本身拥有一个标准，而且这个标准可预见地被所有现代浏览器/引擎实现了。这是非常好的一件事！

但是JavaScript几乎不会与世隔绝地运行。它会运行在混合了第三方库的环境中运行，而且有时甚至会在不同浏览器中不同的引擎/环境中运行。

对这些问题多加注意，会改进你代码的可靠性和健壮性。

附录B: 鸣谢

你不懂JS：类型与文法

附录B： 鸣谢

为了这本书和整个系列的诞生，我有很多人要感谢。

首先，我必须感谢我的妻子 Christen Simpson，和我的两个孩子 Ethan 和 Emily，忍受着老爹总是在电脑上敲打。即使在没有写书时，我对 JavaScript 的痴迷也将我的眼睛粘在屏幕上太久了。我从家庭那里借来的时间是这些书可以如此深入和完整地向你，读者，解释 JavaScript 的原因。我欠我的家庭一切。

我要感谢我在 O'Reilly 的编辑，他们是 Simon St.Laurent 和 Brian MacDonald，还有其他的编辑和市场员工。和他们一起工作很棒，而且在这种"开源"写作，编辑，和生产的实验期间提供了特别的通融。

感谢许多通过提供编辑意见和订正来参与使这部丛书变得更好的朋友们，他们是 Shelley Powers, Tim Ferro, Evan Borden, Forrest L. Norvell, Jennifer Davis, Jesse Harlin, Kris Kowal, Rick Waldron, Jordan Harband, Benjamin Gruenbaum, Vyacheslav Egorov, David Nolen，和许多其他人。一个巨大感谢送给为本书作序的 David Walsh。

感谢社区中无数的朋友们，包括 TC39 协会的成员，他们和我们分享了那么多的知识，特别是以耐心和细节容忍我无休止的问题和探究。John-David Dalton, Juriy "kangax" Zaytsev, Mathias Bynens, Axel Rauschmayer, Nicholas Zakas, Angus Croll, Reginald Braithwaite, Dave Herman, Brendan Eich, Allen Wirfs-Brock, Bradley Meck, Domenic Denicola, David Walsh, Tim Disney, Peter van der Zee, Andrea Giammarchi, Kit Cambridge, Eric Elliott，和其他许多我甚至不能接触到的人。

你不懂*JS* 系列丛书诞生于 Kickstarter，所以我也要感谢我的所有（将近）500 位慷慨的支持者，没有他们这部丛书不可能诞生：

> *Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas*

> *Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hodsden, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel בר-לבב Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuț Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave ♥♬★ furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziółkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Carlos Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew*

> *Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry, Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard*

这部丛书是以开源的风格书写的，包括编辑和生产。我们感激 GitHub 使这样的事情在社区中成为可能！

再次感谢所有无数的朋友，尽管我不能叫上名字但是我依然亏欠感谢。但愿这部丛书被我们所有人“拥有”，并为增进对 JavaScript 语言的意识和理解做出贡献，成为当下和未来所有社区贡献者的助益。