与这段代码最开始的 `var a` 这样还未赋值的变量是一样的。变量能够以几种不同的方式进入这样“未定义”值的状态，其中包括没有返回值的函数和使用 `void` 运算符。

2.1.1 对象

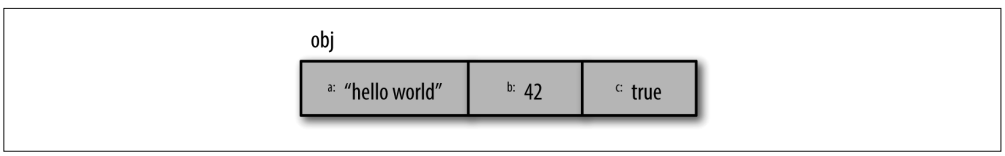
对象类型是指一个组合值，你可以为其设置属性（命名的位置），每个属性可以持有属于自己的任意类型的值。这也许是 JavaScript 中所有的值类型中最有用的一个：

```
var obj = {
    a: "hello world",
    b: 42,
    c: true
};

obj.a;      // "hello world"
obj.b;      // 42
obj.c;      // true

obj["a"];   // "hello world"
obj["b"];   // 42
obj["c"];   // true
```

可以将这个 `obj` 值想象成以下这个可视化的状态，这样更便于理解：



可以通过**点号**（如 `obj.a` 所示）或者**中括号**（如 `obj["a"]` 所示）来访问属性。点号更简短易读，因而尽量使用这种方式。

如果属性名中有特殊字符的话，那么中括号表示法就会很有用，如 `obj["hello world!"]`，在通过中括号表示法访问时，我们通常将这样的属性称为**键值**。`[ ]` 表示法接受变量（后面将会解释）或者字符串字面值（需要使用 `".."` 或 `'..'` 包裹）。

当然，如果需要访问的某个属性 / 键值的名称保存在另一个变量中时，括号表示法也很有用，如下所示：

```
var obj = {
    a: "hello world",
    b: 42
};

var b = "a";

obj[b];       // "hello world"
obj["b"];     // 42
```

有关 JavaScript 对象的更多信息，参见本系列《你不知道的 JavaScript（上卷）》第二部分，特别是第 3 章。

在 JavaScript 程序中，你还需要经常和其他两个值类型打交道：**数组**和**函数**。但你更应该将它们看作是对象类型的特殊子类型，而不是内置类型。

1. 数组

数组是一个持有（任意类型）值的对象，这些值不是通过命名属性 / 键值索引，而是通过数字索引位置。如下所示：

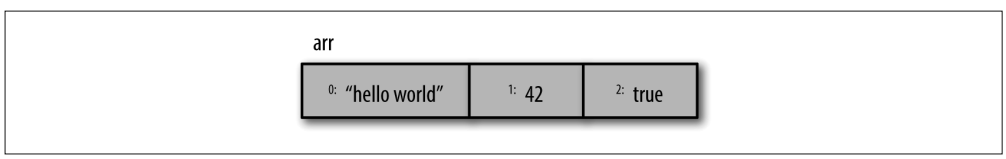
```
var arr = [
    "hello world",
    42,
    true
];

arr[0];          // "hello world"
arr[1];          // 42
arr[2];          // true
arr.length;      // 3

typeof arr;      // "object"
```

像 JavaScript 这样从零开始计数的语言，会使用 `0` 作为数组中第一个元素的索引。

可以将这个 `arr` 值想象成以下这种可视化的状态，这样更便于理解：



因为数组是特殊的对象（正如 `typeof` 所暗示的那样），所以它们也可以有属性，其中包括自动更新的 `length` 属性。

从理论上来说，你可以将数组当作普通的对象来使用，为其添加自己的命名属性，或者你也可以只为一个对象提供数字属性（`0`、`1` 等），就像数组一样使用它。但一般来说，这样使用这些类型是不合理的。

最好的同时也是最自然的方法就是使用数字位置索引数组，通过命名属性使用对象。

2. 函数

在 JavaScript 程序中，另一个常用的对象子类型是函数：

```
function foo() {
    return 42;
}

foo.bar = "hello world";

typeof foo;         // "function"
typeof foo();       // "number"
typeof foo.bar;     // "string"
```

函数也同样是对象的一个子类型，因为 typeof 返回 "function"，这意味着 function 是一个主类型，因此，function 可以拥有属性，但通常只在很少的情况下才会使用函数的对象属性（如 foo.bar）。

有关 JavaScript 值和类型的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分的前两章。

2.1.2 内置类型方法

我们刚刚讨论过的内置类型和子类型拥有作为属性和方法暴露出来的行为，这是非常强大有力的功能。

举例来说：

```
var a = "hello world";
var b = 3.14159;

a.length;           // 11
a.toUpperCase();    // "HELLO WORLD"
b.toFixed(4);       // "3.1416"
```

能够调用 a.toUpperCase() 的原理要比值上存在的方法这种解释复杂得多。

简单地说，存在一个 String（S 大写）对象封装形式，通常称为“原生的”，与基本 string 类型相对应；这个对象封装在其原型中定义了 toUpperCase() 方法。

像前面的代码片段中那样将 "hello world" 这样的原生值作为对象使用时，在引用其属性和方法时（比如 toUpperCase()），JavaScript 会（暗自）自动地将这个值“封箱”为其对应的对象封装。

字符串值可以封装为 String 对象，数字可以封装为 Number 对象，布尔型值可以封装为 Boolean 对象。在多数情况下，你不需要思考直接使用这样的值的对象封装形式，所有情

况下都使用原生值形式，让 JavaScript 来负责其余的事情吧。

有关 JavaScript 原生类型和“封箱”的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分的第 3 章。要想更好地理解一个对象的原型，参见《你不知道的 JavaScript（上卷）》第二部分的第 5 章。

2.1.3 值的比较

JavaScript 程序中有两种主要的值比较：**相等**与**不等**。不管比较的类型是什么，任何比较的结果都是严格的布尔值（`true` 或者 `false`）。

1. 类型转换

我们在第 1 章中简单地讨论了类型转换，现在再深入讨论一下。

JavaScript 中有两种类型转换：**显式的**类型转换与**隐式的**类型转换。显式的类型转换就是你可以在代码中看到的类型由一种转换到另一种，而隐式的类型转换多是某些其他运算可能存在的隐式副作用而引发的类型转换。

你可能听过“类型转换是邪恶的”这种说法，这显然是因为有些情况下的类型转换确实会产生一些出人意料的结果。最能够激怒程序员的事情就是语言发生出乎意料的变化。

类型转换并不是邪恶的，也并不一定是出人意料的。实际上，使用类型转换的多数情况都是非常容易理解的，甚至可以提高代码的可读性。我们不再深入这个有争议的话题，本系列《你不知道的 JavaScript（中卷）》第一部分的第 4 章覆盖了这个主题的方方面面。

以下是显式类型转换的一个示例：

```
var a = "42";

var b = Number( a );

a;              // "42"
b;              // 42--数字!
```

以下是隐式类型转换的一个示例：

```
var a = "42";

var b = a * 1;  // 这里"42"隐式地转换为了42

a;              // "42"
b;              // 42--数字!
```

2. 真与假

我们在第 1 章中简单地提到了值的“真”与“假”的特性：当非布尔型的值被强制转换为

布尔型时，结果是 true 还是 false 呢？

JavaScript 中“假”值的详细列表如下：

- ""（空字符串）
- 0、-0、NaN（无效数字）
- null、undefined
- false

任何不在“假”值列表中的值都是“真”值。以下是一些示例：

- "hello"
- 42
- true
- [ ]、[1, "2", 3]（数组）
- { }、{ a: 42 }（对象）
- function foo() { .. }（函数）

你需要记住非常重要的一点，只有在非布尔型值强制转换为布尔型值时才会遵从这个“真”/“假”转换规则。看起来是将一个值转换成布尔型，而实际上并没有，这样的情况还是很容易令人迷惑的。

3. 相等

相等运算符有四种：==、===、!= 和 !==。! 形式显然是相应的“不等”版本；不要混淆了**不等关系**和**不相等**。

== 和 === 的区别在于，== 检查值相等，而 === 检查值和类型相等。但这么说并不精确。正确的说法是，== 检查的是允许类型转换情况下的值的相等性，而 === 检查不允许类型转换情况下的值的相等性；因此，=== 经常被称为“严格相等”。

思考以下的情况，== 的粗略相等比较允许隐式的类型转换，而严格相等比较 === 则不允许：

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

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

在比较 a == b 的过程中，JavaScript 注意到这两个值的类型不匹配，于是它会按照一系列的顺序步骤将其中一个或两个值的类型转换到其他类型，直到类型匹配，然后进行简单的值相等检查。

思考一下这个过程，有两种类型转换情况可以导致 `a == b` 结果为真。最终比较的要么是 `42 == 42`，要么是 `"42" == "42"`，那么到底是哪一个呢？

答案是，`"42"` 被转化为了 42，使得最终的比较为 `42 == 42`。在这个简单的示例中，过程似乎是无关紧要的，因为最终结果都是一样的。而在一些更为复杂的示例中，不只是最终的比较结果很重要，转换过程也会产生影响。

`a === b` 为假，因为类型转换不被允许，所以简单的值比较的结果显然为假。很多开发者认为 `===` 更可预测，所以他们支持一直使用 `===` 而避免使用 `==`。我认为这种观点是很短视的。在我看来，`==` 是一个强大的工具，**如果花时间来学习其工作原理的话**，那么对程序是很有益的。

我们不打算面面俱到地覆盖 `==` 比较中类型转换的所有工作细节。多数情况都是比较容易理解的，但也有一些重要的特例需要小心对待。你可以查看 ES5 规范（http://www.ecma-international.org/ecma-262/5.1/）的 11.9.3 节来了解精确的规则，与围绕着 `==` 的负面传闻相比，你可能会吃惊于这套机制看起来是多么直观。

下面我将列出几条简单的规则，将所有这些大量的细节归结为简单的条目，以帮助你了解在不同情况下应该使用 `==` 还是 `===`。

- 如果要比较的两个值的任意一个（即一边）可能是 `true` 或者 `false` 值，那么要避免使用 `==`，而使用 `===`。
- 如果要比较的两个值中的任意一个可能是特定值（`0`、`""` 或者 `[]`——空数组），那么避免使用 `==`，而使用 `===`。
- 在所有其他情况下，使用 `==` 都是安全的。不仅仅只是安全而已，这在很多情况下也会简化代码，提高代码的可读性。

提炼出的这几条规则要求你认真思考自己的代码，思考要比较相等性的变量的可能值有哪些。如果你能够确定这些值，并且 `==` 是安全的，那么就可以使用它！如果不能确定其值，那么就使用 `===`。就是这么简单。

不等 `!=` 与 `==` 对应，`!==` 与 `===` 对应。前面讨论过的所有规则和观察对这些不等比较也都是适用的。

如果是比较两个非原生值的话，比如对象（包括函数和数组），那么你需要特殊注意 `==` 与 `===` 这些比较规则。因为这些值通常是通过引用访问的，所以 `==` 和 `===` 比较只是简单地检查这些引用是否匹配，而完全不关心其引用的值是什么。

举例来说，通过简单地在元素之间插入逗号（`,`），数组在默认情况下会转换为字符串。你可能会认为内容相同的两个数组也会 `==` 相等，但并非如此：

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

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

有关 == 相等比较规则的更多信息，参见 ES5 规范（11.9.3 节）以及本系列《你不知道的 JavaScript（中卷）》第一部分的第 4 章；有关值与引用的更多信息，参见《你不知道的 JavaScript（中卷）》第一部分的第 2 章。

4. 不等关系

运算符 <,>、<= 和 >= 用于表示不等关系，在规范中被称为“关系比较”。通常来说，它们用于比较有序值，比如数字。`3 < 4` 这样的比较是很容易理解的。

也可以比较 JavaScript 中的字符串值的不等关系，这是按照常见的字母表规则来比较的（`"bar" < "foo"`）。

类型转换呢？与 == 比较规则类似（尽管是不完全相同！）的规则可以应用于不等关系运算符。注意，并没有与“严格相等”=== 类似的、不允许类型转换的“严格不等关系”运算符。

考虑：

```
var a = 41;
var b = "42";
var c = "43";

a < b;       // true
b < c;       // true
```

上述代码发生了什么？ ES5 规范中的 11.8.5 节中提到，如果 < 比较的两个值都是字符串，就像在 `b < c` 中那样，那么比较按照字典顺序（即字典中的字母表顺序）进行。如果其中一边或两边都不是字符串，就像在 `a < b` 中那样，那么这两个值的类型都转换为数字，然后进行普通的数字比较。

针对类型可能不同的值之间的比较，请记住，没有“严格不等”形式可用。你最需要了解的一点是，当其中一个值无法转换为有效数字时的情形，如下所示：

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

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

为什么这三个比较结果都为假呢？这是因为 < 和 > 比较中的值 b 都被类型转换为了“无效数字值”NaN，规范设定 NaN 既不大于也不小于任何其他值。

== 比较的结果为假的原因则不同。不论解释为 42 == NaN 还是 "42" == "foo"，都会使得 a == b 结果为假——我们已经在前文中介绍过，这种情况属于前者。

有关不等比较规则的更多信息，参见 ES5 标准的 11.8.5 节以及本系列《你不知道的 JavaScript（中卷）》第一部分的第 4 章。

2.2 变量

在 JavaScript 中，变量的名称（包括函数名称）必须是有效的**标识符**。考虑到 Unicode 这样的非传统字符的情况，标识符中有效字符的严格完整规则有点复杂。如果只是考虑常用的 ASCII 字母数字的话，那么规则是非常简单的。

标识符必须由 a~z、A~Z、$ 或 _ 开始。它可以包含前面所有这些字符以及数字 0~9。

一般来说，变量标识符的这个规则对属性命名也同样适用。但是，有些单词不能用作变量名，但可以作为属性名。这些单词被称为“保留词”，其中包括 JavaScript 关键字（for、in、if 等）以及 null、true 和 false。

有关保留字的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分的附录 A。

函数作用域

如果使用关键字 var 声明一个变量，那么这个变量就属于当前的函数作用域，如果声明是发生在任何函数外的顶层声明，那么这个变量则属于全局作用域。

1. 提升

无论 var 出现在一个作用域中的哪个位置，这个声明都属于整个作用域，在其中到处都是可以访问的。

这一行为被比喻地称为**提升**（hoisting），var 声明概念上“移动”到了其所在作用域的最前面。从技术上来说，可以通过如何编译代码更精确地解释这个过程，我们暂时先不赘述这些细节。

考虑：

```
var a = 2;
```

```
foo();                  // 因为foo()而运行
                        // 声明是“被提升的”

function foo() {
    a = 3;

    console.log( a );   // 3

    var a;              // 声明是“被提升的”
                        // 到foo()的顶端
}

console.log( a );       // 2
```

在变量声明出现之前，依靠变量**提升**在其作用域使用这个变量并不常见，也并不是一个好的想法；这样的代码可能会令人非常迷惑。相比之下，使用**提升后的**函数声明则要常见得多，就像我们在 foo() 的正式声明出现前就调用了它。

2. 嵌套作用域

声明后的变量在这个作用域内是随处可以访问的，包括所有低层 / 内层的作用域。举例来说：

```
function foo() {
    var a = 1;

    function bar() {
        var b = 2;

        function baz() {
            var c = 3;

            console.log( a, b, c );  // 1 2 3
        }

        baz();
        console.log( a, b );         // 1 2
    }

    bar();
    console.log( a );                // 1
}

foo();
```

注意，在上述示例中，c 在 bar() 的内部是不可访问的，因为它只声明在内层 baz() 作用域，b 在 foo() 中是不可访问的，也是同样的原因。

如果试图在一个作用域中访问一个不可访问的变量，那么就会抛出 ReferenceError。如果

试图设定尚未声明的变量，那么就会导致在顶层全局作用域创建这个变量（不好！）或者出现错误，这要根据是否处于“严格模式”而定（参见 2.4 节）。我们来看一下：

```
function foo() {
    a = 1;  // a没有正式声明
}

foo();
a;          // 1--哎呀，自动全局变量 :(
```

这是一个很差的实践。不要这么做！一定要正式声明你的变量。

除了在函数层级声明变量，ES6 还支持通过 `let` 关键字声明属于单独块（`{ .. }` 对）的变量。除了细节上有一些微小差别，这里的作用域规则和我们在函数中看到的基本上是相同的：

```
function foo() {
    var a = 1;

    if (a >= 1) {
        let b = 2;

        while (b < 5) {
            let c = b * 2;
            b++;

            console.log( a + c );
        }
    }
}

foo();
// 5 7 9
```

因为使用了 `let` 而不是 `var`，所以 `b` 只属于 `if` 语句，不属于整个 `foo()` 函数的作用域。与此类似，`c` 只属于 `while` 循环。块作用域非常有助于更细化地管理变量作用域，从而更容易随着时间的发展而维护代码。

有关作用域的更多信息，参见本系列中的《你不知道的 JavaScript（上卷）》第一部分。有关 `let` 块作用域的更多信息，参见本书第二部分。

2.3 条件判断

除了第 1 章中简单介绍过的 `if` 语句，JavaScript 还提供了几种其他条件判断机制，我们也应该了解一下。