你有时可能会编写出一系列的 if..else..if 语句，如下所示：

```
if (a == 2) {
    // 做某件事情
}
else if (a == 10) {
    // 做另一件事情
}
else if (a == 42) {
    // 做另一件事情
}
else {
    // 反馈到这儿
}
```

这样的结构可以运行，但是有点繁复，因为你需要为每种情况都指定一个测试。switch 语句是另一种选择：

```
switch (a) {
    case 2:
        // 做某件事情
        break;
    case 10:
        // 做另一件事情
        break;
    case 42:
        // 做另一件事情
        break;
    default:
        // 反馈到这儿
}
```

如果只想要运行某个 case 下的语句，那么 break 是很重要的。如果某个 case 省略了 break，而这个 case 匹配或运行的话，那么会一直执行到下一个 case 的语句，不管那个 case 是否匹配。这种所谓的“通过（fall through）”有时是很有用的。

```
switch (a) {
    case 2:
    case 10:
        // 某个很棒的东西
        break;
    case 42:
        // 其他东西
        break;
    default:
        // 反馈
}
```

在上述示例中，如果 2 或者 10 匹配的话，那么就会执行“某个很棒的东西”代码语句。

在 JavaScript 中，条件判断的另一种形式是“条件运算符”，通常被称为“三进制运算符”。它更像是单个 if..else 语句的紧凑版，如下所示：

```
var a = 42;

var b = (a > 41) ? "hello" : "world";

// 与以下类似:

// if (a > 41) {
//    b = "hello";
// }
// else {
//    b = "world";
// }
```

如果条件表达式（这里是 `a > 41`）求值为真，那么就会返回第一个子句（`"hello"`），否则，结果就是第二个子句（`"world"`），不论结果是什么，都会赋给 `b`。

条件运算符并不一定要用在赋值上，但这肯定是最常见的用法。

有关 `switch` 和 `? :` 的测试条件及其他模式的更多信息，参见本系列中的《你不知道的 JavaScript（中卷）》第一部分。

2.4 严格模式

ES5 为这个语言新增了“严格模式”，严格限制了某些行为的规则。一般来说，这些限制可以将代码保持在一个更安全、更适当的规范集合之内。另外，遵循严格模式也更容易让引擎优化你的代码。严格模式是代码的一次重大突破，你应该在自己的程序中一直使用。

根据严格模式编译指示放置的位置，你可以选择使用单独的函数或者整个文件来遵循严格模式：

```
function foo() {
    "use strict";

    // 这个代码是严格模式

    function bar() {
        // 这个代码是严格模式
    }
}
// 这个代码不是严格模式
```

对比：

```
"use strict";

function foo() {
```

```
    // 这个代码是严格模式

    function bar() {
        // 这个代码是严格模式
    }
}

// 这个代码是严格模式
```

使用严格模式的一个关键区别（改进！）是，不允许省略 var 的隐式自动全局变量声明：

```
function foo() {
    "use strict";    // 开启严格模式
    a = 1;           // 省略var，出现ReferenceError错误
}

foo();
```

如果你在代码中打开严格模式，但代码报错或者开始出现 bug，这可能会诱使你避开严格模式。但这个本能是一个坏习惯。如果严格模式导致程序出现问题，几乎可以确定这标志着你的程序中有些东西应该进行修复。

严格模式不只会让你的代码更加安全或者更易于优化，更代表了这门语言未来的发展方向。现在就要开始习惯严格模式，而不是一直往后推，这对你来说更简单一些，因为转变更晚只会更难！

有关严格模式的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分的第 5 章。

2.5 作为值的函数

到目前为止，我们已经介绍了 JavaScript 中作为主要作用域机制的函数。回忆一下典型的函数声明语法是怎样的：

```
function foo() {
    // ..
}
```

虽然从这个语法上看可能不是很明显，但 foo 基本上就是一个外层作用域中的一个变量，这个作用域赋予被声明函数一个引用。也就是说，这个函数本身是一个值，就像 42 或者 [1,2,3] 一样。

这个概念乍听起来可能很奇怪，你需要花点时间来理解它。不仅你可以向函数传入值（参数），**函数本身也可以作为值**赋给变量或者向其他函数传入，又或者从其他函数传出。

因此，应该将函数值视为一个表达式，与其他的值或者表达式类似。

考虑：

```
var foo = function() {
    // ..
};

var x = function bar(){
    // ..
};
```

第一个赋给变量 foo 的函数表达式被称为是**匿名的**，因为这个函数表达式没有名称。

第二个函数表达式是已命名的（bar），即使它的引用赋值给了变量 x。虽然**匿名函数表达式**的使用仍然极为广泛，但通常更需要**已命名函数表达式**。

要想获取更多信息，参见本系列中的《你不知道的 JavaScript（上卷）》第一部分。

2.5.1 立即调用函数表达式

在前面的代码片段中，两个函数表达式都没有运行——如果加上 foo() 或者 x()，那么就可以执行了。

还有另一种方法可以执行函数表达式，这种方法通常被称为**立即调用函数表达式**（immediately invoked function expression，IIFE）：

```
(function IIFE(){
    console.log( "Hello!" );
})();
// "Hello!"
```

(function IIFE(){ .. }) 函数表达式外面的 (..) 就是 JavaScript 语法能够防止其成为普通函数声明的部分。

表达式最后的 ()（即 })(); 这一行）实际上就表示立即执行前面给出的函数表达式。

这看起来可能有点奇怪，但实际上并不像初看上去那么诡异。思考 foo 和这里的 IIFE 的类似之处：

```
function foo() { .. }

// foo函数引用表达式，然后()执行它
foo();

// IIFE函数表达式，然后()执行它
(function IIFE(){ .. })();
```

正如你看到的，在运行 () 前列出 (function IIFE(){ .. }) 本质上和执行 () 之前的 foo 是一样的；两种情况都是使用 () 执行了在它之前的函数引用。

因为 IIFE 就是一个函数，而且函数会创建新的变量**作用域**，所以使用 IIFE 的这种风格也常用于声明不会影响 IIFE 外代码的变量：

```
var a = 42;

(function IIFE(){
    var a = 10;
    console.log( a );    // 10
})();

console.log( a );        // 42
```

IIFE 也可以有返回值：

```
var x = (function IIFE(){
    return 42;
})();

x;  // 42
```

以上执行的 IIFE 命名函数返回了值 42，并被赋给了 x。

2.5.2 闭包

闭包是 JavaScript 中一个非常重要，且经常被误解的概念。这里不作深入介绍，可以参见本系列中的《你不知道的 JavaScript（上卷）》第一部分。但我将会解释相关的几个要点，以帮助你理解一般概念。这将会是你 JavaScript 技巧集中最重要的技术之一。

你可以将闭包看作“记忆”并在函数运行完毕后继续访问这个函数作用域（其变量）的一种方法。

考虑：

```
function makeAdder(x) {
    // 参数x是一个内层变量

    // 内层函数add()使用x，所以它外围有一个“闭包”
    function add(y) {
        return y + x;
    };

    return add;
}
```

每次调用外层 makeAdder(..) 返回的、指向内层 add(..) 函数的引用能够记忆传入 makeAdder(..) 的 x 值。现在，我们来使用 makeAdder(..)：

```
// plusOne获得指向内层add(..)的一个引用
// 带有闭包的函数在外层makeAdder(..)的x参数上
var plusOne = makeAdder( 1 );

// plusTen获得指向内层add(..)的一个引用
// 带有闭包的函数在外层makeAdder(..)的x参数上
var plusTen = makeAdder( 10 );

plusOne( 3 );        // 4 <-- 1 + 3
plusOne( 41 );       // 42 <-- 1 + 41

plusTen( 13 );       // 23 <-- 10 + 13
```

我们来详细说明一下这段代码是如何执行的。

(1) 调用 `makeAdder(1)` 时得到了内层 `add(..)` 的一个引用，它会将 `x` 记为 `1`。我们将这个函数引用命名为 `plusOne()`。
(2) 调用 `makeAdder(10)` 时得到了内层 `add(..)` 的另一个引用，它会将 `x` 记为 `10`，我们将这个函数引用命名为 `plusTen()`。
(3) 调用 `plusOne(3)` 时，它会向 `1`（记住的 `x`）加上 `3`（内层 `y`），从而得到结果 `4`。
(4) 调用 `plusTen(13)` 时，它会向 `10`（记住的 `x`）加上 `13`（内层 `y`），从而得到结果 `23`。

如果这在刚开始看上去很奇怪，也令人迷惑的话，不要着急！你需要大量实践才能完全理解这个过程。

不过相信我，一旦你理解了，它就会成为所有编程技术中最为强大有用的技术。它绝对值得你花费一些脑力去理解。在下一节中，我们将针对闭包进行更深入的实践。

模块

在 JavaScript 中，闭包最常见的应用是模块模式。模块允许你定义外部不可见的私有实现细节（变量、函数），同时也可以提供允许从外部访问的公开 API。

考虑：

```
function User(){
    var username, password;

    function doLogin(user,pw) {
        username = user;
        password = pw;

        // 执行剩下的登录工作
    }
    var publicAPI = {
        login: doLogin
    };

    return publicAPI;
```

```
}

// 创建一个User模块实例
var fred = User();

fred.login( "fred", "12Battery34!" );
```

函数 User() 用作外层作用域，持有变量 username 和 password，以及内层的函数 doLogin()；这些都是这个 User 模块私有的内部细节，无法从外部访问。

我们有意没有调用 new User()，尽管这事实上可能对多数读者来说更为熟悉。User() 只是一个函数，而不是需要实例化的类，所以只是正常调用就可以了。使用 new 是不合适的，实际上也是浪费资源。

执行 User() 创建了 User 模块的一个**实例**，这创建了一个新的作用域，因而创建了所有内层变量 / 函数的一个新副本。我们将这个实例赋给 fred。如果再次运行 User()，那么会得到一个不同于 fred 的全新实例。

内层的函数 doLogin() 在 username 和 password 上有一个闭包，这意味着即使在 User() 函数运行完毕之后，函数 doLogin() 也保持着对它们的访问权。

publicAPI 是带有一个属性 / 方法 login 的对象，login 是对内层函数 doLogin() 的一个引用。当我们从 User() 返回 publicAPI 时，它就变成了我们命名为 fred 的那个实例。

此时，外层的函数 User() 已经运行完毕。我们通常认为像 username 和 password 这样的内层变量也就随之消失了。但上述示例并不会这样，因为 login() 函数的内部有一个可以使得它们依然保持活跃的闭包。

这就是我们可以调用 fred.login(..) 的原因，这等同于调用内层 doLogin(..)，并且 fred.login(..) 仍然可以访问内层变量 username 和 password。

这里只是对闭包和模块模式的匆匆一瞥，它们的某些细节很可能还是有点令人迷惑。没关系！让你的大脑完全理解它还需要一些努力。

从现在开始，阅读本系列《你不知道的 JavaScript（上卷）》第一部分，这有助于你更进一步地理解这些知识。

2.6 this 标识符

JavaScript 中另一个被普遍误解的概念是 this 关键字。同样地，本系列《你不知道的 JavaScript（上卷）》第二部分中有几章内容是专门介绍这一技术的，因此，这里只是简单地介绍一下概念。

虽然 this 一般与“面向对象的模式”相关，但 JavaScript 中的 this 则是另外一种机制。

如果一个函数内部有一个 this 引用，那么这个 this 通常指向一个对象。但它指向的是哪个对象要根据这个函数是如何被调用来决定。

this **并不指向**这个函数本身，意识到这一点非常重要，因为这是最常见的误解。

以下是一个简单的说明：

```
function foo() {
    console.log( this.bar );
}

var bar = "global";

var obj1 = {
    bar: "obj1",
    foo: foo
};

var obj2 = {
    bar: "obj2"
};

// --------

foo();              // “全局的”
obj1.foo();         // "obj1"
foo.call( obj2 );   // "obj2"
new foo();          // undefined
```

关于如何设置 this 有 4 条规则，上述代码中的最后 4 行展示了这 4 条规则。

(1) 在非严格模式下，foo() 最后会将 this 设置为全局对象。在严格模式下，这是未定义的行为，在访问 bar 属性时会出错——因此 "global" 是为 this.bar 创建的值。
(2) obj1.foo() 将 this 设置为对象 obj1。
(3) foo.call(obj2) 将 this 设置为对象 obj2。
(4) new foo() 将 this 设置为一个全新的空对象。

底线：为了搞清楚 this 指向什么，你必须检查相关的函数是如何被调用的。调用方式会是以上 4 种之一，这也会回答“this 是什么”这个问题。

有关 this 的更多信息，参见本系列《你不知道的 JavaScript（上卷）》第二部分中的前两章。

2.7 原型

JavaScript 中的原型机制是非常复杂的。这里我们仅仅浅谈一下。你需要花费很多时间来阅读本系列《你不知道的 JavaScript（上卷）》第二部分的 4~6 章，以了解所有细节。

当引用对象的某个属性时，如果这个属性并不存在，那么 JavaScript 会自动使用对象的内部原型引用找到另外一个对象来寻找这个属性。你可以将这点看作是属性缺失情况的备用模式。

从一个对象到其后备对象的内部原型引用的链接是在创建对象时发生的。展示这一点的最简单的方法就是使用内置工具 `Object.create(..)`。

考虑：

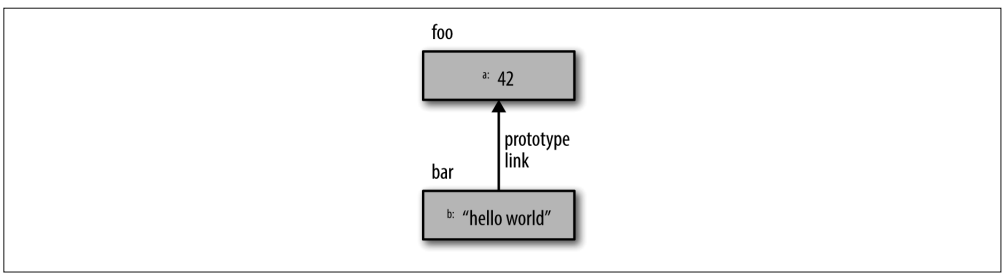
```
var foo = {
    a: 42
};

// 创建bar并将其链接到foo
var bar = Object.create( foo );

bar.b = "hello world";

bar.b;      // "hello world"
bar.a;      // 42 <-- 委托给foo
```

下图直观地展示了对象 `foo`、`bar` 及其相互关系：



对象 `bar` 上的 `a` 属性实际上并不存在，但因为 `bar` 是原型链接到 `foo` 的，所以 JavaScript 自动跳入到 `foo` 对象上搜索 `a` 属性，并且成功找到了。

这种链接看起来像是这个语言的奇怪特性。这个特性最常见的使用（我认为是误用）方式就是模拟 / 伪装带“继承”关系的“类”机制。

更自然应用原型的方式是被称为“行为委托”的模式，其设计意图是，被链接对象能够将其所需要的行为委托给另外一个对象。

有关原型和行为委托的更多信息，参见本系列《你不知道的 JavaScript（上卷）》第二部分的 4~6 章。

2.8 旧与新

在我们已经介绍过的 JavaScript 特性和本系列其余图书将要介绍的更多特性中，有一部分特性是新增的，旧版浏览器不一定会支持这样的特性。实际上，标准中的部分最新特性甚至还没有在哪个稳定版的浏览器中实现。

那么，应该怎样对待这些新特性呢？是不是只能等几年甚至几十年，直到所有这些旧版的浏览器退休呢？

很多人都是这么认为的，但这对 JavaScript 来说实际上是很不健康的一种思路。

你可以使用两种主要的技术，即 polyfilling 和 transpilling，向旧版浏览器“引入”新版的 JavaScript 特性。

2.8.1 polyfilling

单词“polyfill”是由 Remy Sharp 发明的一个新术语（https://remysharp.com/2010/10/08/what-is-a-polyfill），用于表示根据新特性的定义，创建一段与之行为等价但能够在旧的 JavaScript 环境中运行的代码。

举例来说，ES6 定义了一个名为 `Number.isNaN(..)` 的工具，用于提供一个精确无 bug 的 `NaN` 值检查，取代原来的 `isNaN(..)`。但对这个工具进行兼容处理很容易，这样一来，无论终端用户是否使用 ES6 浏览器，你都能够开始使用它。

考虑：

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

`if` 语句防止在已经支持此特性的 ES6 浏览器中应用 polyfill 定义。如果还不存在的话，那我们就定义 `Number.isNaN(..)`。

我们进行的这个检查利用了 `NaN` 值的一个特性，即 `NaN` 是整个语言中唯一和自身不相等的值。因此，`NaN` 是使得 `x != x` 为真的唯一值。