如果某个 script 中定义了函数 foo()，后面的 script 代码就可以访问并调用 foo()，就像 foo() 在其内部被声明过一样。

但是全局变量作用域的提升机制（hoisting，参见《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分）在这些边界中不适用，因此无论是 <script> .. </script> 还是 <script src=..></script>，下面的代码都无法运行（因为 foo() 还未被声明）。

```
<script>foo();</script>

<script>
  function foo() { .. }
</script>
```

但是下面的两段代码则没问题：

```
<script>
  foo();
  function foo() { .. }
</script>
```

和：

```
<script>
  function foo() { .. }
</script>

<script>foo();</script>
```

如果 script 中的代码（无论是内联代码还是外部代码）发生错误，它会像独立的 JavaScript 程序那样停止，但是后续的 script 中的代码（仍然共享 global）依然会接着运行，不会受影响。

你可以使用代码来动态创建 script，将其加入到页面的 DOM 中，效果是一样的：

```
var greeting = "Hello World";

var el = document.createElement( "script" );

el.text = "function foo(){ alert( greeting );\
 } setTimeout( foo, 1000 );";

document.body.appendChild( el );
```

如果将 el.src 的值设置为一个文件 URL，就可以通过 <scrscript=..></script> 动态加载外部文件。

内联代码和外部文件中的代码之间有一个区别，即在内联代码中不可以出现 `</script>` 字符串，一旦出现即被视为代码块结束。因此对于下面这样的代码需要非常小心：

```
<script>
  var code = "<script>alert( 'Hello World' )</script>";
</script>
```

上述代码看似没什么问题，但是字符串常量中的 `</script>` 将会被当作结束标签来处理，因此会导致错误。常用的变通方法是：

```
"</sc" + "ript>";
```

另外需要注意的一点是，我们是根据代码文件的字符集属性（UTF-8、ISO-8859-8 等）来解析外部文件中的代码（或者默认字符集），而内联代码则使用其所在页面文件的字符集（或者默认字符集）。

内联代码的 `script` 标签没有 `charset` 属性。

`script` 标签的一个已废止的用法是在内联代码中包含 HTML 或 XHTML 格式的注释，如：

```
<script>
<!--
alert( "Hello" );
//-->
</script>

<script>
<!--//--><![CDATA[//><!--
alert( "World" );
//--><!]]>
</script>
```

现在我们已经不需要这样做了，所以不要再继续使用它们。

`<!--` 和 `-->`（HTML 格式的注释）在 JavaScript 中被定义为合法的单行注释分隔符（`var x = 2;` 是另一行合法注释），这是老的技术导致的（详见 A.1 节的“Web ECMAScript”部分），切勿再使用它们。

A.6 保留字

ES5 规范在 7.6.1 节中定义了一些“保留字”，我们不能将它们用作变量名。这些保留字有

四类：“关键字”“预留关键字”、null 常量和 true/false 布尔常量。

像 function 和 switch 都是关键字。预留关键字包括 enum 等，它们中很多已经在 ES6 中被用到（如 class、extend 等）。另外还有一些在严格模式中使用的保留字，如 interface。

一个名为“art4theSould”的 StackOverflow 用户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.

这首诗中包含了 ES3 中的保留字（byte、long 等），它们在 ES5 中已经不再是保留字。

在 ES5 之前，保留字也不能用来作为对象常量中的属性名称或者键值，但是现在已经没有这个限制。

例如，下面的情况是不允许的：

```
var import = "42";
```

但是下面的情况是允许的：

```
var obj = { import: "42" };
console.log( obj.import );
```

需要注意的是，在一些版本较老的浏览器中（主要是 IE），这些规则并不完全适用，有时候将保留字作为对象属性还是会出错。所以需要在所有要支持的浏览器中仔细测试。

A.7 实现中的限制

JavaScript 规范对于函数中参数的个数，以及字符串常量的长度等并没有限制；但是由于 JavaScript 引擎实现各异，规范在某些地方有一些限制。

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
```

在一些 JavaScript 引擎中你会得到正确答案 `499950000`，而另外一些引擎（如 Safari 6.x）中则会产生错误“RangeError: Maximum call stack size exceeded”。

下面列出一些已知的限制：

- 字符串常量中允许的最大字符数（并非只是针对字符串值）；
- 可以作为参数传递到函数中的数据大小（也称为栈大小，以字节为单位）；
- 函数声明中的参数个数；
- 未经优化的调用栈（例如递归）的最大层数，即函数调用链的最大长度；
- JavaScript 程序以阻塞方式在浏览器中运行的最长时间（秒）；
- 变量名的最大长度。

我们不会经常碰到这些限制，但应该对它们有所了解，特别是不同的 JavaScript 引擎的限制各异。

A.8 小结

JavaScript 语言本身有一个统一的标准，在所有浏览器 / 引擎中的实现也是可靠的。这是好事！

但是 JavaScript 很少独立运行。通常运行环境中还有第三方代码，有时代码甚至会运行在浏览器之外的引擎 / 环境中。

只要我们对这些问题多加注意，就能够提高代码的可靠性和健壮性。

第二部分

异步和性能

单 业 译

序

多年以来，老板信任我，让我负责面试。如果我们在寻找一个具备 JavaScript 技能的雇员，我的第一个问题就是……好吧，其实就是问问面试者是否需要上厕所或者喝点什么，因为舒适很重要。不过一旦面试者完成了这个液体输入或输出的过程，我就要开始判断他是否了解 JavaScript，还是只了解 jQuery。

不是说 jQuery 哪里不好。jQuery 让你不需要真正了解 JavaScript 就可以做很多事情，这是功能，而不是 bug。但是，如果工作内容需要 JavaScript 性能和维护方面的高级技能，那么应聘的人就应了解如何把像 jQuery 这样的库组合起来。你得能够像这些库一样驾驭 JavaScript 的核心。

如果要整体了解一个人的核心 JavaScript 技能，我最感兴趣的是他们会如何使用闭包（你已经读过这一系列中的《你不知道的 JavaScript（上卷）》了吧？）以及如何充分利用异步，这一点把我们引向了这本书。

首先第一道菜是回调，这是异步编程的面包和黄油（基础）。当然，靠面包和黄油并不足以完成令人非常满意的大餐，接下来的课程就是美味异常的 Promise！

如果还不了解 Promise，现在正是学习的时候。Promise 现在已经是 JavaScript 和 DOM 提供异步返回值的正式方法。所有未来的异步 DOM API 都会使用它们，而且很多已经这么做了，所以做好准备吧！写作本文的时候，多数主流浏览器中已经发布了 Promise，IE 很快也会提供支持。享用 Promise 之后，希望你的胃里还有空间留给下一道美食——生成器。

没有大张旗鼓的宣传，生成器就已经悄悄进入了 Chrome 和 Firefox 的稳定版本，这是因为，坦白地说，它们的复杂性要高于其趣味性。或者说，在看到它们与 Promise 合作之前，我一直都是这么认为的。可之后呢，它们成了提高可读性和可维护性的重要工具。

甜品是……嗯，我不想破坏了惊喜，但是，准备好展望 JavaScript 的未来吧！这本书涵盖的功能会让你对并发和异步有越来越多的控制。

好吧，我不再妨碍你享用这本书了——让精彩继续吧！如果在看此序之前你已经阅读了本书的部分内容，那么请给你自己加 10 个异步分！这 10 分是你应得的。

——Jake Archibald（http://jakearchibald.com，@jaffathecake），
Google Chrome 开发大使

第 1 章

异步：现在与将来

使用像 JavaScript 这样的语言编程时，很重要但常常被误解的一点是，如何表达和控制持续一段时间的程序行为。

这不仅仅是指从 `for` 循环开始到结束的过程，当然这也需要持续一段时间（几微秒或几毫秒）才能完成。它是指程序的一部分*现在*运行，而另一部分则在*将来*运行——*现在*和*将来*之间有段间隙，在这段间隙中，程序没有活跃执行。

实际上，所有重要的程序（特别是 JavaScript 程序）都需要通过这样或那样的方法来管理这段时间间隙，这时可能是在等待用户输入、从数据库或文件系统中请求数据、通过网络发送数据并等待响应，或者是在以固定时间间隔执行重复任务（比如动画）。在诸如此类的场景中，程序都需要管理这段时间间隙的状态。地铁门上不也总是贴着一句警示语——“小心空隙”（指地铁门与站台之间的空隙）。

事实上，程序中*现在*运行的部分和*将来*运行的部分之间的关系就是异步编程的核心。

毫无疑问，从一开始，JavaScript 就涉及异步编程。但是，多数 JavaScript 开发者从来没有认真思考过自己程序中的异步到底是如何出现的，以及其为什么会出现，也没有探索过处理异步的其他方法。一直以来，低调的回调函数就算足够好的方法了。目前为止，还有很多人坚持认为回调函数完全够用。

但是，作为在浏览器、服务器以及其他能够想到的任何设备上运行的一流编程语言，JavaScript 面临的需求日益扩大。为了满足这些需求，JavaScript 的规模和复杂性也在持续增长，对异步的管理也越来越令人痛苦，这一切都迫切需要更强大、更合理的异步方法。

目前为止，所有这些讨论看起来都还比较抽象，不过我向你保证，随着本书内容的推进，对这个问题的讨论会越来越完整和具体。在接下来的几章中，我们会探讨各种新出现的JavaScript 异步编程技术。

但是，在此之前，我们首先需要深入理解异步的概念及其在 JavaScript 中的运作模式。

1.1 分块的程序

可以把 JavaScript 程序写在单个 .js 文件中，但是这个程序几乎一定是由多个块构成的。这些块中只有一个是*现在*执行，其余的则会在*将来*执行。最常见的块单位是函数。

大多数 JavaScript 新手程序员都会遇到的问题是：程序中*将来*执行的部分并不一定在*现在*运行的部分执行完之后就立即执行。换句话说，*现在*无法完成的任务将会异步完成，因此并不会出现人们本能地认为会出现的或希望出现的阻塞行为。

考虑：

```
// ajax(..)是某个库中提供的某个Ajax函数
var data = ajax( "http://some.url.1" );

console.log( data );
// 啊哦！data通常不会包含Ajax结果
```

你可能已经了解，标准 Ajax 请求不是同步完成的，这意味着 `ajax(..)` 函数还没有返回任何值可以赋给变量 `data`。如果 `ajax(..)` 能够阻塞到响应返回，那么 `data = ..` 赋值就会正确工作。

但我们并不是这么使用 Ajax 的。*现在*我们发出一个异步 Ajax 请求，然后在*将来*才能得到返回的结果。

从*现在*到*将来*的“等待”，最简单的方法（但绝对不是唯一的，甚至也不是最好的！）是使用一个通常称为*回调函数*的函数：

```
// ajax(..)是某个库中提供的某个Ajax函数
ajax( "http://some.url.1", function myCallbackFunction(data){

    console.log( data ); // 耶！这里得到了一些数据！

} );
```

可能你已经听说过，可以发送同步 Ajax 请求。尽管技术上说是这样，但是，在任何情况下都不应该使用这种方式，因为它会锁定浏览器 UI（按钮、菜单、滚动条等），并阻塞所有的用户交互。这是一个可怕的想法，一定要避免。

你有不同的意见？我知道，但为了避免回调函数引起的混乱并不足以成为使用阻塞式同步 Ajax 的理由。

举例来说，考虑一下下面这段代码：

```
function now() {
    return 21;
}

function later() {
    answer = answer * 2;
    console.log( "Meaning of life:", answer );
}

var answer = now();

setTimeout( later, 1000 ); // Meaning of life: 42
```

这个程序有两个块：*现在*执行的部分，以及*将来*执行的部分。这两块的内容很明显，但这里我们还是要明确指出来。

现在：

```
function now() {
    return 21;
}

function later() { .. }

var answer = now();

setTimeout( later, 1000 );
```

将来：

```
answer = answer * 2;
console.log( "Meaning of life:", answer );
```

*现在*这一块在程序运行之后就会立即执行。但是，`setTimeout(..)` 还设置了一个事件（定时）在*将来*执行，所以函数 `later()` 的内容会在之后的某个时间（从现在起 1000 毫秒之后）执行。

任何时候，只要把一段代码包装成一个函数，并指定它在响应某个事件（定时器、鼠标点击、Ajax 响应等）时执行，你就是在代码中创建了一个*将来*执行的块，也由此在这个程序中引入了异步机制。

异步控制台

并没有什么规范或一组需求指定 `console.*` 方法族如何工作——它们并不是 JavaScript 正式

的一部分，而是由*宿主环境*（请参考本书的“类型和语法”部分）添加到 JavaScript 中的。

因此，不同的浏览器和 JavaScript 环境可以按照自己的意愿来实现，有时候这会引起混淆。

尤其要提出的是，在某些条件下，某些浏览器的 `console.log(..)` 并不会把传入的内容立即输出。出现这种情况的主要原因是，在许多程序（不只是 JavaScript）中，I/O 是非常低速的阻塞部分。所以，（从页面 /UI 的角度来说）浏览器在后台异步处理控制台 I/O 能够提高性能，这时用户甚至可能根本意识不到其发生。

下面这种情景不是很常见，但也可能发生，从中（不是从代码本身而是从外部）可以观察到这种情况：

```
var a = {
    index: 1
};

// 然后
console.log( a ); // ??

// 再然后
a.index++;
```

我们通常认为恰好在执行到 `console.log(..)` 语句的时候会看到 `a` 对象的快照，打印出类似于 `{ index: 1 }` 这样的内容，然后在下一条语句 `a.index++` 执行时将其修改，这句的执行会严格在 `a` 的输出之后。

多数情况下，前述代码在开发者工具的控制台中输出的对象表示与期望是一致的。但是，这段代码运行的时候，浏览器可能会认为需要把控制台 I/O 延迟到后台，在这种情况下，等到浏览器控制台输出对象内容时，`a.index++` 可能已经执行，因此会显示 `{ index: 2 }`。

到底什么时候控制台 I/O 会延迟，甚至是否能够被观察到，这都是游移不定的。如果在调试的过程中遇到对象在 `console.log(..)` 语句之后被修改，可你却看到了意料之外的结果，要意识到这可能是这种 I/O 的异步化造成的。

如果遇到这种少见的情况，最好的选择是在 JavaScript 调试器中使用断点，而不要依赖控制台输出。次优的方案是把对象序列化到一个字符串中，以强制执行一次“快照”，比如通过 `JSON.stringify(..)`。

1.2 事件循环

现在我们来澄清一件事情（可能令人震惊）：尽管你显然能够编写异步 JavaScript 代码（就像前面我们看到的定时代码），但直到最近（ES6），JavaScript 才真正内建有直接的异步概念。

什么？！这种说法似乎很疯狂，对不对？但事实就是这样。JavaScript 引擎本身所做的只不过是在需要的时候，在给定的任意时刻执行程序中的单个代码块。

“需要”，谁的需要？这正是关键所在！

JavaScript 引擎并不是独立运行的，它运行在*宿主环境*中，对多数开发者来说通常就是 Web 浏览器。经过最近几年（不仅于此）的发展，JavaScript 已经超出了浏览器的范围，进入了其他环境，比如通过像 Node.js 这样的工具进入服务器领域。实际上，JavaScript 现如今已经嵌入到了从机器人到电灯泡等各种各样的设备中。

但是，所有这些环境都有一个共同“点”（thread，也指线程。不论真假与否，这都不算一个很精妙的异步笑话），即它们都提供了一种机制来处理程序中多个块的执行，且执行每块时调用 JavaScript 引擎，这种机制被称为*事件循环*。

换句话说，JavaScript 引擎本身并没有时间的概念，只是一个按需执行 JavaScript 任意代码片段的环境。“事件”（JavaScript 代码执行）调度总是由包含它的环境进行。

所以，举例来说，如果你的 JavaScript 程序发出一个 Ajax 请求，从服务器获取一些数据，那你就在一个函数（通常称为*回调函数*）中设置好响应代码，然后 JavaScript 引擎会通知宿主环境：“嘿，现在我要暂停执行，你一旦完成网络请求，拿到了数据，就请调用这个函数。”

然后浏览器就会设置侦听来自网络的响应，拿到要给你的数据之后，就会把回调函数插入到事件循环，以此实现对这个回调的调度执行。

那么，什么是事件循环？

先通过一段伪代码了解一下这个概念：

```
// eventLoop是一个用作队列的数组
// (先进,先出)
var eventLoop = [ ];
var event;

// “永远”执行
while (true) {
    // 一次tick
    if (eventLoop.length > 0) {
        // 拿到队列中的下一个事件
        event = eventLoop.shift();

        // 现在,执行下一个事件
        try {
            event();
        }
        catch (err) {
            reportError(err);
```

```
            }
        }
    }
```

这当然是一段极度简化的伪代码，只用来说明概念。不过它应该足以用来帮助大家有更好的理解。

你可以看到，有一个用 `while` 循环实现的持续运行的循环，循环的每一轮称为一个 tick。对每个 tick 而言，如果在队列中有等待事件，那么就会从队列中摘下一个事件并执行。这些事件就是你的回调函数。

一定要清楚，`setTimeout(..)` 并没有把你的回调函数挂在事件循环队列中。它所做的是设定一个定时器。当定时器到时后，环境会把你的回调函数放在事件循环中，这样，在未来某个时刻的 tick 会摘下并执行这个回调。

如果这时候事件循环中已经有 20 个项目了会怎样呢？你的回调就会等待。它得排在其他项目后面——通常没有抢占式的方式支持直接将其排到队首。这也解释了为什么 `setTimeout(..)` 定时器的精度可能不高。大体说来，只能确保你的回调函数不会在指定的时间间隔之前运行，但可能会在那个时刻运行，也可能在那之后运行，要根据事件队列的状态而定。

所以换句话说就是，程序通常分成了很多小块，在事件循环队列中一个接一个地执行。严格地说，和你的程序不直接相关的其他事件也可能会插入到队列中。

前面提到的“直到最近”是指 ES6 从本质上改变了在哪里管理事件循环。本来它几乎已经是一种正式的技术模型了，但现在 ES6 精确指定了事件循环的工作细节，这意味着在技术上将其纳入了 JavaScript 引擎的势力范围，而不是只由宿主环境来管理。这个改变的一个主要原因是 ES6 中 Promise 的引入，因为这项技术要求对事件循环队列的调度运行能够直接进行精细控制（参见 1.4.3 节中对 `setTimeout(..0)` 的讨论），具体内容会在第 3 章中介绍。

1.3 并行线程

术语“异步”和“并行”常常被混为一谈，但实际上它们的意义完全不同。记住，异步是关于*现在*和*将来*的时间间隙，而并行是关于能够同时发生的事情。

并行计算最常见的工具就是*进程*和*线程*。进程和线程独立运行，并可能同时运行：在不同的处理器，甚至不同的计算机上，但多个线程能够共享单个进程的内存。

与之相对的是，事件循环把自身的工作分成一个个任务并顺序执行，不允许对共享内存的并行访问和修改。通过分立线程中彼此合作的事件循环，并行和顺序执行可以共存。

并行线程的交替执行和异步事件的交替调度，其粒度是完全不同的。

举例来说：

```
function later() {
    answer = answer * 2;
    console.log( "Meaning of life:", answer );
}
```

尽管 later() 的所有内容被看作单独的一个事件循环队列表项，但如果考虑到这段代码是运行在一个线程中，实际上可能有很多个不同的底层运算。比如，answer = answer * 2 需要先加载 answer 的当前值，然后把 2 放到某处并执行乘法，取得结果之后保存回 answer 中。

在单线程环境中，线程队列中的这些项目是底层运算确实是无所谓的，因为线程本身不会被中断。但如果是在并行系统中，同一个程序中可能有两个不同的线程在运转，这时很可能就会得到不确定的结果。

考虑：

```
var a = 20;

function foo() {
    a = a + 1;
}

function bar() {
    a = a * 2;
}

// ajax(..)是某个库中提供的某个Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

根据 JavaScript 的单线程运行特性，如果 foo() 运行在 bar() 之前，a 的结果是 42，而如果 bar() 运行在 foo() 之前的话，a 的结果就是 41。

如果共享同一数据的 JavaScript 事件并行执行的话，那么问题就变得更加微妙了。考虑 foo() 和 bar() 中代码运行的线程分别执行的是以下两段伪代码任务，然后思考一下如果它们恰好同时运行的话会出现什么情况。

线程 1（X 和 Y 是临时内存地址）：

```
foo():
  a. 把a的值加载到X
  b. 把1保存在Y
  c. 执行X加Y,结果保存在X
  d. 把X的值保存在a
```

线程 2（X 和 Y 是临时内存地址）：