器 JavaScript 中呢？

对 ES6 中所有的语法扩展来说，都有工具（最常见的术语是 transpiler，指 trans-compiler，翻译编译器）用于接收 ES6 语法并将其翻译为等价（但是显然要丑陋一些！）的前 ES6 代码。因此，生成器可以被翻译为具有同样功能但可以工作于 ES5 及之前的代码。

可怎么实现呢？显然 `yield` 的“魔法”看起来并不那么容易翻译。实际上，我们之前在讨论基于闭包的迭代器时已经暗示了一种解决方案。

4.8.1 手工变换

在讨论 transpiler 之前，先来推导一下对生成器来说手工变换是如何实现的。这不只是一个理论上的练习，因为这个练习实际上可以帮助我们更深入理解其工作原理。

考虑：

```
// request(..)是一个支持Promise的Ajax工具

function *foo(url) {
    try {
        console.log( "requesting:", url );
        var val = yield request( url );
        console.log( val );
    }
    catch (err) {
        console.log( "Oops:", err );
        return false;
    }
}

var it = foo( "http://some.url.1" );
```

首先要观察到的是，我们仍然需要一个可以调用的普通函数 `foo()`，它仍然需要返回一个迭代器。因此，先把非生成器变换的轮廓刻画出来：

```
function foo(url) {

    // ..

    // 构造并返回一个迭代器
    return {
        next: function(v) {
            // ..
        },
        throw: function(e) {
            // ..
        }
    };
}
```

```
var it = foo( "http://some.url.1" );
```

接下来要观察到的是，生成器是通过暂停自己的作用域 / 状态实现它的“魔法”的。可以通过函数闭包（参见本系列的《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分）来模拟这一点。为了理解这样的代码是如何编写的，我们先给生成器的各个部分标注上状态值：

```
// request(..)是一个支持Promise的Ajax工具

function *foo(url) {
    // 状态1

    try {
        console.log( "requesting:", url );
        var TMP1 = request( url );

        // 状态2
        var val = yield TMP1;
        console.log( val );
    }
    catch (err) {
        // 状态3
        console.log( "Oops:", err );
        return false;
    }
}
```

为了更精确地展示，我们使用临时变量 TMP1 把 `val = yield request..` 语句分成了两个部分。`request(..)` 在状态 1 发生，其完成值赋给 `val` 发生在状态 2。当我们把代码转换成其非生成器等价时，会去掉这个中间变量 TMP1。

换句话说，1 是起始状态，2 是 `request(..)` 成功后的状态，3 是 `request(..)` 失败的状态。你大概能够想象出如何把任何额外的 `yield` 步骤编码为更多的状态。

回到我们翻译的生成器，让我们在闭包中定义一个变量 `state` 用于跟踪状态：

```
function foo(url) {
    // 管理生成器状态
    var state;

    // ..
}
```

现在在闭包内定义一个内层函数，称为 `process(..)`，使用 `switch` 语句处理每个状态：

```
// request(..)是一个支持Promise的Ajax工具

function foo(url) {
    // 管理生成器状态
```

```
    var state;

    // 生成器范围变量声明
    var val;

    function process(v) {
        switch (state) {
            case 1:
                console.log( "requesting:", url );
                return request( url );
            case 2:
                val = v;
                console.log( val );
                return;
            case 3:
                var err = v;
                console.log( "Oops:", err );
                return false;
        }
    }

    // ..
}
```

我们生成器的每个状态都在 switch 语句中由自己的 case 表示。每次需要处理一个新状态的时候就会调用 process(..)。稍后我们将会回来介绍这是如何工作的。

对于每个生成器级的变量声明（val），我们都把它移动为 process(..) 外的一个 val 声明，这样它们就可以在多个 process(..) 调用之间存活。不过块作用域的变量 err 只在状态 3 中需要使用，所以把它留在原来的位置。

在状态 1，没有了 yield resolve(..)，我们所做的是 return resolve(..)。在终止状态 2，没有显式的 return，所以我们只做一个 return，这等价于 return undefined。在终止状态 3，有一个 return false，因此就保留这一句。

现在需要定义迭代器函数的代码，使这些函数正确调用 process(..)：

```
function foo(url) {
    // 管理生成器状态
    var state;

    // 生成器变量范围声明
    var val;

function process(v) {
    switch (state) {
        case 1:
            console.log( "requesting:", url );
            return request( url );
        case 2:
            val = v;
            console.log( val );
```

```
                return;
            case 3:
                var err = v;
                console.log( "Oops:", err );
                return false;
        }
    }

    // 构造并返回一个生成器
    return {
        next: function(v) {
            // 初始状态
            if (!state) {
                state = 1;
                return {
                    done: false,
                    value: process()
                };
            }
            // yield成功恢复
            else if (state == 1) {
                state = 2;
                return {
                    done: true,
                    value: process( v )
                };
            }
            // 生成器已经完成
            else {
                return {
                    done: true,
                    value: undefined
                };
            }
        },
        "throw": function(e) {
            // 唯一的显式错误处理在状态1
            if (state == 1) {
                state = 3;
                return {
                    done: true,
                    value: process( e )
                    };
                }
                // 否则错误就不会处理,所以只把它抛回
                else {
                    throw e;
                }
            }
        };
    }
```

这段代码是如何工作的呢？

(1) 对迭代器的 next() 的第一个调用会把生成器从未初始化状态转移到状态 1，然后调用 process() 来处理这个状态。request(..) 的返回值是对应 Ajax 响应的 promise，作为 value 属性从 next() 调用返回。

(2) 如果 Ajax 请求成功，第二个 next(..) 调用应该发送 Ajax 响应值进来，这会把状态转移到状态 2。再次调用 process(..)（这次包括传入的 Ajax 响应值），从 next(..) 返回的 value 属性将是 undefined。

(3) 然而，如果 Ajax 请求失败的话，就会使用错误调用 throw(..)，这会把状态从 1 转移到 3（而非 2）。再次调用 process(..)，这一次包含错误值。这个 case 返回 false，被作为 throw(..) 调用返回的 value 属性。

从外部来看（也就是说，只与迭代器交互），这个普通函数 foo(..) 与生成器 *foo(..) 的工作几乎完全一样。所以我们已经成功地把 ES6 生成器转为了前 ES6 兼容代码！

然后就可以手工实例化生成器并控制它的迭代器了，调用 var it = foo("..") 和 it.next(..) 等。甚至更好的是，我们可以把它传给前面定义的工具 run(..)，就像 run(foo,"..")。

4.8.2 自动转换

前面的 ES6 生成器到前 ES6 等价代码的手工推导练习，向我们教授了概念上生成器是如何工作的。但是，这个变换非常复杂，并且对于代码中的其他生成器而言也是不可移植的。这部分工作通过手工实现十分不实际，会完全抵消生成器的一切优势。

但幸运的是，已经有一些工具可以自动把 ES6 生成器转化为前面小节中我们推导出来的结果那样的代码。它们不仅会为我们完成这些笨重的工作，还会处理我们忽略的几个枝节问题。

regenerator 就是这样的一个工具（http://facebook.github.io/regenerator/），出自 Facebook 的几个聪明人。

如果使用 regenerator 来转换前面的生成器的话，以下是产生的代码（本书写作之时）：

```
// request(..)是一个支持Promise的Ajax工具

var foo = regeneratorRuntime.mark(function foo(url) {
    var val;

    return regeneratorRuntime.wrap(function foo$(context$1$0) {
        while (1) switch (context$1$0.prev = context$1$0.next) {
        case 0:
            context$1$0.prev = 0;
            console.log( "requesting:", url );
```

```
                context$1$0.next = 4;
                return request( url );
            case 4:
                val = context$1$0.sent;
                console.log( val );
                context$1$0.next = 12;
                break;
            case 8:
                context$1$0.prev = 8;
                context$1$0.t0 = context$1$0.catch(0);
                console.log("Oops:", context$1$0.t0);
                return context$1$0.abrupt("return", false);
            case 12:
            case "end":
                return context$1$0.stop();
            }
        }, foo, this, [[0, 8]]);
    });
```

这与我们手工推导的结果有一些明显的相似之处，比如那些 `switch/case` 语句，而且我们甚至看到了移出闭包的 `val`，就像我们做的一样。

当然，一个不同之处是，regenerator 的变换需要一个辅助库 `regeneratorRuntime`，其中包含了管理通用生成器和迭代器的所有可复用逻辑。这些重复代码中有很多和我们的版本不同，但即使这样，很多概念还是可以看到的，比如 `context$1$0.next = 4` 记录生成器的下一个状态。

主要的收获是，生成器不再局限于只能在 ES6+ 环境中使用。一旦理解了这些概念，就可以在代码中使用，然后使用工具将其变换为与旧环境兼容的代码。

这比仅仅将修改后的 `Promise` API 用作前 ES6 Promise 所做的工作要多得多，但是，付出的代价是值得的，因为在实现以合理的、明智的、看似同步的、顺序的方式表达异步流程方面，生成器的优势太多了。

一旦迷上了生成器，就再也不会想回到那一团乱麻的异步回调地狱中了。

4.9 小结

生成器是 ES6 的一个新的函数类型，它并不像普通函数那样总是运行到结束。取而代之的是，生成器可以在运行当中（完全保持其状态）暂停，并且将来再从暂停的地方恢复运行。

这种交替的暂停和恢复是合作性的而不是抢占式的，这意味着生成器具有独一无二的能力来暂停自身，这是通过关键字 `yield` 实现的。不过，只有控制生成器的迭代器具有恢复生成器的能力（通过 `next(..)`）。

yield/next(..) 这一对不只是一种控制机制，实际上也是一种双向消息传递机制。yield ..表达式本质上是暂停下来等待某个值，接下来的 next(..) 调用会向被暂停的 yield 表达式传回一个值（或者是隐式的 undefined）。

在异步控制流程方面，生成器的关键优点是：生成器内部的代码是以自然的同步 / 顺序方式表达任务的一系列步骤。其技巧在于，我们把可能的异步隐藏在了关键字 yield 的后面，把异步移动到控制生成器的迭代器的代码部分。

换句话说，生成器为异步代码保持了顺序、同步、阻塞的代码模式，这使得大脑可以更自然地追踪代码，解决了基于回调的异步的两个关键缺陷之一。

第 5 章

程序性能

到目前为止，本书的内容都是关于如何更加有效地利用异步模式。但是，我们并没有直接阐述为什么异步对 JavaScript 来说真的很重要。最显而易见的原因就是性能。

举例来说，如果要发出两个 Ajax 请求，并且它们之间是彼此独立的，但是需要等待两个请求都完成才能执行下一步的任务，那么为这个交互建模有两种选择：顺序与并发。

可以先发出第一个请求，然后等待第一个请求结束，之后发出第二个请求。或者，就像我们在 promise 和生成器部分看到的那样，也可以并行发出两个请求，然后用门模式来等待两个请求完成，之后再继续。

显然，通常后一种模式会比前一种更高效。而更高的性能通常也会带来更好的用户体验。

甚至有可能异步（交替执行的并发）只能够提高感觉到的性能，而整体来说，程序完成的时间还是一样的。用户感知的性能和实际可测的性能一样重要，如果不是更重要的话！

现在我们不再局限于局部化的异步模式，而是将在程序级别上讨论更大图景下的性能细节。

你可能想要了解一些微性能问题，比如 `a++` 和 `++a` 哪个更快。这一类性能细节将在第 6 章中讨论。

5.1 Web Worker

如果你有一些处理密集型的任务要执行，但不希望它们都在主线程运行（这可能会减慢浏览器 /UI），可能你就会希望 JavaScript 能够以多线程的方式运行。

在第 1 章里，我们已经详细介绍了 JavaScript 是如何单线程运作的。但是，单线程并不是组织程序执行的唯一方式。

设想一下，把你的程序分为两个部分：一部分运行在主 UI 线程下，另外一部分运行在另一个完全独立的线程中。

这样的架构可能会引出哪些方面的问题呢？

一个就是，你会想要知道在独立的线程运行是否意味着它可以并行运行（在多 CPU/ 核心的系统上），这样第二个线程的长时间运行就不会阻塞程序主线程。否则，相比于 JavaScript 中已有的异步并发，“虚拟多线程”并不会带来多少好处。

你还会想知道程序的这两个部分能否访问共享的作用域和资源。如果可以的话，那你就将遇到多线程语言（Java、C++ 等）要面对的所有问题，比如需要合作式或抢占式的锁机制（mutex 等）。这是相当多的额外工作，不要小看。

还有，如果这两个部分能够共享作用域和资源的话，你会想要知道它们将如何通信。

在我们对 Web 平台 HTML5 的一个叫作 Web Worker 的新增特性的探索过程中，这些都是很好的问题。这是浏览器（即宿主环境）的功能，实际上和 JavaScript 语言本身几乎没什么关系。也就是说，JavaScript 当前并没有任何支持多线程执行的功能。

但是，像你的浏览器这样的环境，很容易提供多个 JavaScript 引擎实例，各自运行在自己的线程上，这样你可以在每个线程上运行不同的程序。程序中每一个这样的独立的多线程部分被称为一个（Web）Worker。这种类型的并行化被称为*任务并行*，因为其重点在于把程序划分为多个块来并发运行。

从 JavaScript 主程序（或另一个 Worker）中，可以这样实例化一个 Worker：

```
var w1 = new Worker( "http://some.url.1/mycoolworker.js" );
```

这个 URL 应该指向一个 JavaScript 文件的位置（而不是一个 HTML 页面！），这个文件将被加载到一个 Worker 中。然后浏览器启动一个独立的线程，让这个文件在这个线程中作为独立的程序运行。

这种通过这样的 URL 创建的 Worker 称为专用 Worker（Dedicated Worker）。除了提供一个指向外部文件的 URL，你还可以通过提供一个 Blob URL（另外一个 HTML5 特性）创建一个在线 Worker（Inline Worker），本质上就是一个存储在单个（二进制）值中的在线文件。不过，Blob 已经超出了我们这里的讨论范围。

Worker 之间以及它们和主程序之间，不会共享任何作用域或资源，那会把所有多线程编程的噩梦带到前端领域，而是通过一个基本的事件消息机制相互联系。

Worker `w1` 对象是一个事件侦听者和触发者，可以通过订阅它来获得这个 Worker 发出的事件以及发送事件给这个 Worker。

以下是如何侦听事件（其实就是固定的 `"message"` 事件）：

```
w1.addEventListener( "message", function(evt){
    // evt.data
} );
```

也可以发送 `"message"` 事件给这个 Worker：

```
w1.postMessage( "something cool to say" );
```

在这个 Worker 内部，收发消息是完全对称的：

```
// "mycoolworker.js"

addEventListener( "message", function(evt){
    // evt.data
} );

postMessage( "a really cool reply" );
```

注意，专用 Worker 和创建它的程序之间是一对一的关系。也就是说，`"message"` 事件没有任何歧义需要消除，因为我们确定它只能来自这个一对一的关系：它要么来自这个 Worker，要么来自主页面。

通常由主页面应用程序创建 Worker，但若是需要的话，Worker 也可以实例化它自己的子 Worker，称为 subworker。有时候，把这样的细节委托给一个“主”Worker，由它来创建其他 Worker 处理部分任务，这样很有用。不幸的是，到写作本书时为止，Chrome 还不支持 subworker，不过 Firefox 支持。

要在创建 Worker 的程序中终止 Worker，可以调用 Worker 对象（就像前面代码中的 `w1`）上的 `terminate()`。突然终止 Worker 线程不会给它任何机会完成它的工作或者清理任何资源。这就类似于通过关闭浏览器标签页来关闭页面。

如果浏览器中有两个或多个页面（或同一页上的多个 tab！）试图从同一个文件 URL 创建 Worker，那么最终得到的实际上是完全独立的 Worker。后面我们会简单介绍如何共享 Worker。

看起来似乎恶意或无知的 JavaScript 程序只要在一个系统中生成上百个 Worker，让每个 Worker 运行在低级独立的线程上，就能够以此制造拒绝服务攻击。尽管这确实从某种程度上保证了每个 Worker 将运行在自己的独立线程上，但是这个保证并不是毫无限度的。系统能够决定可以创建多少个实际的线程 /CPU/ 核心。没有办法预测或保证你能够访问多少个可用线程，尽管很多人假定至少可以达到 CPU/ 核心的数量。我认为最安全的假定就是在主 UI 线程之外至少还有一个线程，就是这样。

5.1.1 Worker 环境

在 Worker 内部是无法访问主程序的任何资源的。这意味着你不能访问它的任何全局变量，也不能访问页面的 DOM 或者其他资源。记住，这是一个完全独立的线程。

但是，你可以执行网络操作（Ajax、WebSockets）以及设定定时器。还有，Worker 可以访问几个重要的全局变量和功能的本地复本，包括 `navigator`、`location`、`JSON` 和 `applicationCache`。

你还可以通过 `importScripts(..)` 向 Worker 加载额外的 JavaScript 脚本：

```
// 在Worker内部
importScripts( "foo.js", "bar.js" );
```

这些脚本加载是同步的。也就是说，`importScripts(..)` 调用会阻塞余下 Worker 的执行，直到文件加载和执行完成。

另外，已经有一些讨论涉及把 `<canvas>`API 暴露给 Worker，以及把 canvas 变为 Transferable（参见 5.1.2 节），这将使 Worker 可以执行更高级的 off-thread 图形处理，这对于高性能游戏（WebGL）和其他类似的应用是很有用的。尽管目前的浏览器中还不存在这种支持，但很可能不远的将来就会有。

Web Worker 通常应用于哪些方面呢？

- 处理密集型数学计算
- 大数据集排序
- 数据处理（压缩、音频分析、图像处理等）
- 高流量网络通信

5.1.2 数据传递

你可能已经注意到这些应用中的大多数有一个共性，就是需要在线程之间通过事件机制传递大量的信息，可能是双向的。

在早期的 Worker 中，唯一的选择就是把所有数据序列化到一个字符串值中。除了双向序列化导致的速度损失之外，另一个主要的负面因素是数据需要被复制，这意味着两倍的内存使用（及其引起的垃圾收集方面的波动）。

谢天谢地，现在已经有了一些更好的选择。

如果要传递一个对象，可以使用结构化克隆算法（structured clone algorithm）（https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/The_structured_clone_algorithm）把这个对象复制到另一边。这个算法非常高级，甚至可以处理要复制的对象有循环引用的情况。这样就不用付出 to-string 和 from-string 的性能损失了，但是这种方案还是要使用双倍的内存。IE10 及更高版本以及所有其他主流浏览器都支持这种方案。

还有一个更好的选择，特别是对于大数据集而言，就是使用 Transferable 对象（http://updates.html5rocks.com/2011/12/Transferable-Objects-Lightning-Fast）。这时发生的是对象所有权的转移，数据本身并没有移动。一旦你把对象传递到一个 Worker 中，在原来的位置上，它就变为空的或者是不可访问的，这样就消除了多线程编程作用域共享带来的混乱。当然，所有权传递是可以双向进行的。

如果选择 Transferable 对象的话，其实不需要做什么。任何实现了 Transferable 接口（http://developer.mozilla.org/en-US/docs/Web/API/Transferable）的数据结构就自动按照这种方式传输（Firefox 和 Chrome 都支持）。

举例来说，像 Uint8Array 这样的带类型的数组（参见本系列的《你不知道的 JavaScript（下卷）》的“ES6 & Beyond”部分）就是 Transferable。下面是如何使用 `postMessage(..)` 发送一个 Transferable 对象：

```
// 比如foo是一个Uint8Array
postMessage( foo.buffer, [ foo.buffer ] );
```

第一个参数是一个原始缓冲区，第二个是一个要传输的内容的列表。

不支持 Transferable 对象的浏览器就降级到结构化克隆，这会带来性能下降而不是彻底的功能失效。

5.1.3 共享 Worker

如果你的站点或 app 允许加载同一个页面的多个 tab（一个常见的功能），那你可能非常希

望通过防止重复专用 Worker 来降低系统的资源使用。在这一方面最常见的有限资源就是 socket 网络连接，因为浏览器限制了到同一个主机的同时连接数目。当然，限制来自于同一客户端的连接数也减轻了你的资源压力。

在这种情况下，创建一个整个站点或 app 的所有页面实例都可以共享的中心 Worker 就非常有用了。

这称为 `SharedWorker`，可通过下面的方式创建（只有 Firefox 和 Chrome 支持这一功能）：

```
var w1 = new SharedWorker( "http://some.url.1/mycoolworker.js" );
```

因为共享 Worker 可以与站点的多个程序实例或多个页面连接，所以这个 Worker 需要通过某种方式来得知消息来自于哪个程序。这个唯一标识符称为*端口*（port），可以类比网络 socket 的端口。因此，调用程序必须使用 Worker 的 `port` 对象用于通信：

```
w1.port.addEventListener( "message", handleMessages );

// ..

w1.port.postMessage( "something cool" );
```

还有，端口连接必须要初始化，形式如下：

```
w1.port.start();
```

在共享 Worker 内部，必须要处理额外的一个事件：`"connect"`。这个事件为这个特定的连接提供了端口对象。保持多个连接独立的最简单办法就是使用 `port` 上的闭包（参见本系列《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分），就像下面的代码一样，把这个链接上的事件侦听和传递定义在 `"connect"` 事件的处理函数内部：

```
// 在共享Worker内部
addEventListener( "connect", function(evt){
    // 这个连接分配的端口
    var port = evt.ports[0];

    port.addEventListener( "message", function(evt){
        // ..

        port.postMessage( .. );

        // ..
    } );

    // 初始化端口连接
    port.start();
} );
```

除了这个区别之外，共享和专用 Worker 在功能和语义方面都是一样的。

如果有某个端口连接终止而其他端口连接仍然活跃，那么共享 Worker 不会终止。而对专用 Worker 来说，只要到实例化它的程序的连接终止，它就会终止。

5.1.4 模拟 Web Worker

从性能的角度来说，将 Web Worker 用于并行运行 JavaScript 程序是非常有吸引力的方案。但是，由于环境所限。你可能需要在缺乏对此支持的更老的浏览器中运行你的代码。因为 Worker 是一种 API 而不是语法，所以我们可以作为扩展来模拟它。

如果浏览器不支持 Worker，那么从性能的角度来说是没法模拟多线程的。通常认为 Iframe 提供了并行环境，但是在所有的现代浏览器中，它们实际上都是和主页面运行在同一个线程中的，所以并不足以模拟并发。

就像我们在第 1 章中详细讨论的，JavaScript 的异步（不是并行）来自于事件循环队列，所以可使用定时器（`setTimeout(..)` 等）强制模拟实现异步的伪 Worker。然后你只需要提供一个 Worker API 的封装。Modernizr GitHub 页面（http://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills#web-workers）上列出了一些实现，但坦白地说，它们看起来都不太好。

在这一点上，我也编写了一个模拟 Worker 的概要实现（https://gist.github.com/getify/1b26accb1a09aa53ad25）。它是很基本的，但如果双向消息机制正确工作，并且“`onerror`”处理函数也正确工作，那么它应该可以提供简单的 Worker 支持。如果需要的话，你也可以扩展它，实现更多的功能，比如 `terminate()` 或伪共享 Worker。

因为无法模拟同步阻塞，所以这个封装不支持使用 `importScripts(..)`。对此，一个可能的选择是，解析并转换 Worker 的代码（一旦 Ajax 加载之后）来处理重写为某种异步形式的 `importScripts(..)` 模拟，可能通过支持 promise 的接口。

5.2 SIMD

单指令多数据（SIMD）是一种*数据并行*（data parallelism）方式，与 Web Worker 的*任务并行*（task parallelism）相对，因为这里的重点实际上不再是把程序逻辑分成并行的块，而是并行处理数据的多个位。

通过 SIMD，线程不再提供并行。取而代之的是，现代 CPU 通过数字“向量”（特定类型的数组），以及可以在所有这些数字上并行操作的指令，来提供 SIMD 功能。这是利用低级指令级并行的底层运算。

把 SIMD 功能暴露到 JavaScript 的尝试最初是由 Intel 发起的（https://01.org/node/1495），具体来说就是，Mohammad Haghighat（在本书写作时）与 Firefox 和 Chrome 团队合作。SIMD 目前正在进行早期的标准化，很有机会进入到 JavaScript 的未来版本，比如 ES7。

SIMD JavaScript 计划向 JavaScript 代码暴露短向量类型和 API。在支持 SIMD 的那些系统中，这些运算将会直接映射到等价的 CPU 指令，而在非 SIMD 系统中就会退化回非并行化的运算。

对于数据密集型的应用（信号分析、关于图形的矩阵运算，等等），这样的并行数学处理带来的性能收益是非常明显的！

在本书写作时，早期提案中的 API 形式类似如下：

```
var v1 = SIMD.float32x4( 3.14159, 21.0, 32.3, 55.55 );
var v2 = SIMD.float32x4( 2.1, 3.2, 4.3, 5.4 );

var v3 = SIMD.int32x4( 10, 101, 1001, 10001 );
var v4 = SIMD.int32x4( 10, 20, 30, 40 );

SIMD.float32x4.mul( v1, v2 );
    // [ 6.597339, 67.2, 138.89, 299.97 ]
SIMD.int32x4.add( v3, v4 );
    // [ 20, 121, 1031, 10041 ]
```

这里展示的是两个不同的向量数据类型，32 位浮点数和 32 位整型。可以看到，这些向量大小恰好就是四个 32 位元素，因为这和多数当代 CPU 上支持的 SIMD 向量大小（128 位）匹配。未来还有可能看到这些 API 的 `x8`（或更大！）版本。

除了 `mul()` 和 `add()`，很多其他运算还可以包含在内，比如 `sub()`、`div()`、`abs()`、`neg()`、`sqrt()`、`reciprocal()`、`reciprocalSqrt()`（算术）、`shuffle()`（重新安排向量元素）、`and()`、`or()`、`xor()`、`not()`（逻辑）、`equal()`、`greaterThan()`、`lessThan()`（比较）、`shiftLeft()`、`shiftRightLogical()`、`shiftRightArithmetic()`（移位）、`fromFloat32x4()` 以及 `fromInt32x4()`（转换）。

对于可用的 SIMD 功能（http://github.com/johnmccutchan/ecmascript_simd），有一个官方的（有希望的、值得期待的、面向未来的）prolyfill，它展示了比我们这一节中多得多的计划好的 SIMD 功能。