```
    })();
})();
```

可能你会认为最内层函数中的引用 foo 需要进行三层作用域查找。本系列的《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分讨论了词法作用域是如何工作的。事实上，编译器通常会缓存这样的查找结果，使得从不同的作用域引用 foo 实际上并没有任何额外的花费。

但是，还有一些更深入的问题需要思考。如果编译器意识到这个 foo 只在一个位置被引用而别处没有任何引用，并且注意到这个值只是 41 而从来不会变成其他值呢？

JavaScript 可能决定完全去掉 foo 变量，将其值*在线化*，这不是很可能发生也可以接受的吗？就像下面这样：

```
(function(){
    (function(){
        (function(baz){
            var bar = 41 + baz;
            // ..
        })(1);
    })();
})();
```

当然，这里编译器有可能也对 baz 进行类似的分析和重写。

当你把 JavaScript 代码看作对引擎要做什么的提示和建议，而不是逐字逐句的要求时，你就会意识到，对于具体语法细节的很多执着迷恋已经烟消云散了。

另一个例子：

```
function factorial(n) {
    if (n < 2) return 1;
    return n * factorial( n - 1 );
}

factorial( 5 );     // 120
```

啊，很不错的老式阶乘算法！你可能认为 JavaScript 就像代码这样运行。但说实话，只是可能，我真的也不确定。

但作为一件趣事，同样的代码用 C 编写并用高级优化编译的结果是，编译器意识到调用 factorial(5) 可以直接用常量值 120 来代替，完全消除了函数的调用！

另外，有些引擎会进行名为*递归展开*的动作，在这里，它能够意识到你表达的递归其实可以用循环更简单地实现（即优化）。JavaScript 引擎有可能会把前面的代码重写如下来运行：

```
function factorial(n) {
    if (n < 2) return 1;

    var res = 1;
    for (var i=n; i>1; i--) {
        res *= i;
    }
    return res;
}

factorial( 5 );     // 120
```

现在，我们设想一下，在前面的代码片段中你还担心 `n * factorial(n-1)` 和 `n *= factorial(--n)` 哪个运行更快。甚至可能你还进行了性能测试来确定哪个更好。但你忽略了这个事实：在更大的上下文中，引擎可能并不会运行其中任何一行代码，因为它可能会进行递归展开！

说到 `--`，`--n` 对比 `n--` 经常被作为那些通过选择 `--n` 版本来优化的情况进行引用，因为从理论上说，在汇编语言级上它需要处理的工作更少。

对现代 JavaScript 来说，这一类执迷基本上毫无意义。这就属于你应该让引擎来关心的那一类问题。你应该编写意义最明确的代码。比较下面的三个 `for` 循环：

```
// 选择1
for (var i=0; i<10; i++) {
    console.log( i );
}

// 选择2
for (var i=0; i<10; ++i) {
    console.log( i );
}

// 选择3
for (var i=-1; ++i<10; ) {
    console.log( i );
}
```

即使你认为理论上第二个或第三个选择要比第一个选择性能高那么一点点，这也是值得怀疑的。第三个循环更令人迷惑，因为使用了 `++i` 先递增运算，你就不得不把 `i` 从 `-1` 开始计算。而第一个和第二个选择之间的区别实际上完全无关紧要。

完全有可能一个 JavaScript 引擎看到了一个使用 `i++` 的位置，并意识到它可能将其安全地替换为等价的 `++i`，这意味着你花费在决定采用哪一种方案上的时间完全被浪费了，而且产出还毫无意义。

这里是另一个常见的愚蠢的执迷于微观性能的例子：

```
var x = [ .. ];

// 选择1
for (var i=0; i < x.length; i++) {
    // ..
}

// 选择2
for (var i=0, len = x.length; i < len; i++) {
    // ..
}
```

理论上说，这里应该在变量 `len` 中缓存 `x` 数组的长度，因为表面上它不会改变，来避免在每个循环迭代中计算 `x.length` 的代价。

如果运行性能测试来比较使用 `x.length` 和将其缓存到 `len` 变量中的方案，你会发现尽管理论听起来没错，但实际的可测差别在统计上是完全无关紧要的。

实际上，在某些像 v8 这样的引擎中，可以看到（http://mrale.ph/blog/2014/12/24/array-length-caching.html），预先缓存长度而不是让引擎为你做这件事情，会使性能稍微下降一点。不要试图和 JavaScript 引擎比谁聪明。对性能优化来说，你很可能会输。

6.5.1 不是所有的引擎都类似

各种浏览器中的不同 JavaScript 引擎可以都是“符合规范的”，但其处理代码的方法却完全不同。JavaScript 规范并没有任何性能相关的要求，好吧，除了 ES6 的“尾调用优化”，这部分将在 6.6 节介绍。

引擎可以自由决定一个运算是否需要优化，可能进行权衡，替换掉运算次要性能。对一个运算来说，很难找到一种方法使其在所有浏览器中都运行得较快。

在一些 JavaScript 开发社区有一场运动，特别是在那些使用 Node.js 工作的开发者中间。这场运动是要分析 v8 JavaScript 引擎的特定内部实现细节，决定编写裁剪过的 JavaScript 代码来最大程度地利用 v8 的工作模式。通过这样的努力，你可能会获得令人吃惊的高度性能优化。因此，这种努力的回报可能会很高。

如下是 v8 的一些经常提到的例子（https://github.com/petkaantonov/bluebird/wiki/Optimization-killers）。

- 不要从一个函数到另外一个函数传递 `arguments` 变量，因为这样的泄漏会降低函数实现速度。
- 把 `try..catch` 分离到单独的函数里。浏览器对任何有 `try..catch` 的函数实行优化都有一些困难，所以把这部分移到独立的函数中意味着你控制了反优化的害处，并让其包含

的代码可以优化。

不过，与其关注这些具体技巧，倒不如让我们在通用的意义上对 v8 独有的优化方法进行一次完整性检查。

确实要编写只需在一个 JavaScript 引擎上运行的代码吗？即使你的代码目前只需要 Node.js，假定使用的 JavaScript 引擎永远是 v8 是否可靠呢？有没有可能某一天，几年以后，会有 Node.js 之外的另一种服务器端 JavaScript 平台被选中运行你的代码呢？如果你之前的优化如今对新引擎而言成了一种运行很慢的方法，要怎么办呢？

或者如果从现在开始你的代码总是保持运行在 v8 上，但是 v8 决定在某些方面修改其运算的工作方式，过去运行很快的方式现在很慢，或者相反，那又该怎么办？

这些场景并不仅仅只是理论。过去把多个字符串值放在一个数组中，然后在数组上调用 `join("")` 来连接这些值比直接用 + 连接这些值要快。这一点的历史原因是微妙的，涉及字符串值在内存中如何存储和管理这样的内部实现细节。

因此，那时的工业界广泛传播的最佳实践建议是：开发者应总是使用数组的 `join(..)` 方法。很多人遵从了这一建议。

但随着时间的发展，JavaScript 引擎改变了内部管理字符串的方法，特别对 + 连接进行了优化。它们并没有降低 `join(..)` 本身的效率，而是花了更多精力提高 + 的使用，因为 `join` 仍然是广泛使用的。

主要基于某些方法当前的广泛使用来标准化或优化这些特定方法的实践通常称为（比喻意义上的）“给已被牛踏出的路铺砖”。

一旦新的处理字符串和连接的方法确定下来，很遗憾，所有那些使用数组 `join(..)` 来连接字符串的代码就成次优的了。

另一个例子：曾几何时，Opera 浏览器在如何处理原生封装的对象的封箱 / 开箱上与其他浏览器不同（参见本书的“类型和语法”部分）。同样，他们对开发者的建议是：如果需要访问 `length` 这样的属性或 `charAt(..)` 这样的方法，应使用 `String` 对象而不是原生字符串值。这个建议对那时候的 Opera 来说可能是正确的，但是它完全与同时代的其他主流浏览器背道而驰，因为后者都对原生字符串有特殊的优化而不是对其对象封装。

我想，即使是对于今天的代码，这些陷阱至少是可能出现的，如果不是很容易发生的话。因此，我对在我的代码中单纯根据引擎实现细节进行的广泛性能优化非常小心，特别是如果这些细节只对于单个引擎成立的话。

反过来的情形也需要慎重：你不应该修改一段代码以通过高性能运行一段代码，进而绕过一个引擎的困难之处。

从历史上看，IE 一直是这类问题的主要源头。因为在很多场景下，老版的 IE 都挣扎于许多性能方面的问题，而同时期的其他主流浏览器却似乎没什么问题。实际上我们刚才讨论的字符串连接问题在 IE6 和 IE7 时期是一个真实的问题，那时候通过 `join(..)` 可能会得到比 + 更好的性能。

但是，如果只有一个浏览器出现性能问题，就建议使用可能在其他所有浏览器都是次优的代码方案，可能会带来麻烦。即使这个浏览器在你网站用户中占据最大的市场份额也是如此，可能更实际的方法是编写合适的代码，并依赖浏览器以更好的优化来更新自己。

“没有比临时 hack 更持久的了”。很有可能你现在编写的用来绕过一些性能 bug 的代码可能比浏览器的性能问题本身存在得更长久。

在浏览器每五年才更新一次的时候，这是个很难作出的抉择。但是到了现在，浏览器更新的速度要快得多（尽管移动世界显然还落在后面），它们都彼此竞争着对 Web 功能进行越来越好的优化。

如果你遇到这样的情形，即一个浏览器有性能问题而其他浏览器没有，那就要确保通过随便什么可用的渠道把这个问题报告其开发者。多数浏览器都提供了开放的 bug 跟踪工具用于此处。

我建议只有在浏览器的性能问题确实引发彻底的中断性故障时才去绕过它，不要仅仅因为它让人讨厌就那么做。我也会非常小心地检查，以确定性能 hack 在其他浏览器上不会有显著的消极副作用。

6.5.2 大局

我们应该关注优化的大局，而不是担心这些微观性能的细微差别。

怎么知道什么是大局呢？首先要了解你的代码是否运行在关键路径上。如果不在关键路径上，你的优化就很可能得不到很大的收益。

有没有听过“这是过早优化”这样的警告？这来自于高德纳著名的一句话：“过早优化是万恶之源。”很多开发者都会引用这句话来说明多数优化都是“过早的”，因此是白费力气。和通常情况一样，事实要更加微妙一些。

这里是高德纳的原话及上下文（http://web.archive.org/web/20130731202547/http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf）（重点强调）：

> 程序员们浪费了大量的时间用于思考，或担心他们程序中非关键部分的速度，这些针对效率的努力在调试和维护方面带来了强烈的负面效果。我们应该在，比如说 97% 的时间里，忘掉小处的效率：过早优化是万恶之源。但我们不应该错过关键的 3% 中的机会。
>
> ——计算访谈 6（1974 年 12 月）

我相信这么解释高德纳的意思是合理的："非关键路径上的优化是万恶之源。"所以，关键是确定你的代码是否在关键路径上——如果在的话，就应该优化！

甚至可以更进一步这么说：花费在优化关键路径上的时间不是浪费，不管节省的时间多么少；而花在非关键路径优化上的时间都不值得，不管节省的时间多么多。

如果你的代码在关键路径上，比如是一段将要反复运行多次的"热"代码，或者在用户会注意到的 UX 关键位置上，如动画循环或 CSS 风格更新，那你就不应该吝惜精力去采用有意义的、可测量的有效优化。

举例来说，考虑一下：一个关键路径动画循环需要把一个字符串类型转换到数字。当然有很多种方法可以实现（参见本书的"类型和语法"部分），但是哪一种，如果有的话，是最快的呢？

```
var x = "42";    // 需要数字42

// 选择1:让隐式类型转换自动发生
var y = x / 2;

// 选择2:使用parseInt(..)
var y = parseInt( x, 0 ) / 2;

// 选择3:使用Number(..)
var y = Number( x ) / 2;

// 选择4:使用一元运算符+
var y = +x / 2;

// 选项5:使用一元运算符|
var y = (x | 0) / 2;
```

我将把这个问题留给你作为练习。如果感兴趣的话，可以建立一个测试，检查这些选择之间的性能差异。

在考虑这些不同的选择时，就像别人说的，"其中必有一个是与众不同的"。`parseInt(..)` 可以实现这个功能，但是它也做了更多的工作：它解析字符串而不是近几年进行类型转

换。你很可能会猜测 parseInt(..) 是一个比较慢的选择，应该避免，这是正确的。

当然，如果 x 可能是一个*需要解析*的值，比如 "42px"（比如来自 CSS 风格查找），那 parseInt(..) 就确实是唯一合理的选择了！

Number(..) 也是一个函数调用。从行为角度说，它和一元运算符 + 选择是完全一样的，但实际上它可能更慢一些，要求更多的执行函数的机制。当然，也可能 JavaScript 引擎意识到了行为上的相同性，会帮你把 Number(..) 在线化（即 +x）！

但是，请记住，沉迷于 +x 与 x | 0 的对比在绝大多数情况下都是浪费时间。这是一个微观性能问题，是一个你不应该让其影响程序可读性的问题。

尽管程序关键路径上的性能非常重要，但这并不是唯一要考虑的因素。在性能方面大体相似的几个选择中，可读性应该是另外一个重要的考量因素。

6.6 尾调用优化

正如前面我们提到的，ES6 包含了一个性能领域的特殊要求。这与一个涉及函数调用的特定优化形式相关：*尾调用优化*（Tail Call Optimization，TCO）。

简单地说，尾调用就是一个出现在另一个函数“结尾”处的函数调用。这个调用结束后就没有其余事情要做了（除了可能要返回结果值）。

举例来说，以下是一个非递归的尾调用：

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

function bar(y) {
    return foo( y + 1 );     // 尾调用
}

function baz() {
    return 1 + bar( 40 );    // 非尾调用
}

baz();                       // 42
```

foo(y+1) 是 bar(..) 中的尾调用，因为在 foo(..) 完成后，bar(..) 也完成了，并且只需要返回 foo(..) 调用的结果。然而，bar(40) 不是尾调用，因为在它完成后，它的结果需要加上 1 才能由 baz() 返回。

不详细谈那么多本质细节的话，调用一个新的函数需要额外的一块预留内存来管理调用栈，称为*栈帧*。所以前面的代码一般会同时需要为每个 baz()、bar(..) 和 foo(..) 保留一

个栈帧。

然而，如果支持 TCO 的引擎能够意识到 `foo(y+1)` 调用*位于尾部*，这意味着 `bar(..)` 基本上已经完成了，那么在调用 `foo(..)` 时，它就不需要创建一个新的栈帧，而是可以重用已有的 `bar(..)` 的栈帧。这样不仅速度更快，也更节省内存。

在简单的代码片段中，这类优化算不了什么，但是在处理递归时，这就解决了大问题，特别是如果递归可能会导致成百上千个栈帧的时候。有了 TCO，引擎可以用同一个栈帧执行所有这类调用！

递归是 JavaScript 中一个纷繁复杂的主题。因为如果没有 TCO 的话，引擎需要实现一个随意（还彼此不同！）的限制来界定递归栈的深度，达到了就得停止，以防止内存耗尽。有了 TCO，尾调用的递归函数本质上就可以任意运行，因为再也不需要使用额外的内存！

考虑到前面递归的 `factorial(..)`，这次重写成 TCO 友好的：

```
function factorial(n) {
    function fact(n,res) {
        if (n < 2) return res;

        return fact( n - 1, n * res );
    }

    return fact( n, 1 );
}

factorial( 5 );     // 120
```

这个版本的 `factorial(..)` 仍然是递归的，但它也是可以 TCO 优化的，因为内部的两次 `fact(..)` 调用的位置都在结尾处。

有一点很重要，需要注意：TCO 只用于有实际的尾调用的情况。如果你写了一个没有尾调用的递归函数，那么性能还是会回到普通栈帧分配的情形，引擎对这样的递归调用栈的限制也仍然有效。很多递归函数都可以改写，就像刚刚展示的 `factorial(..)` 那样，但是需要认真注意细节。

ES6 之所以要求引擎实现 TCO 而不是将其留给引擎自由决定，一个原因是缺乏 TCO 会导致一些 JavaScript 算法因为害怕调用栈限制而降低了通过递归实现的概率。

如果在所有的情况下引擎缺乏 TCO 只是降低了性能，那它就不会成为 ES6 所要求的东西。但是，由于缺乏 TCO 确实可以使一些程序变得无法实现，所以它就成为了一个重要的语言特性而不是隐藏的实现细节。

ES6 确保了 JavaScript 开发者从现在开始可以在所有符合 ES6+ 的浏览器中依赖这个优化。这对 JavaScript 性能来说是一个胜利。

6.7 小结

对一段代码进行有效的性能测试，特别是与同样代码的另外一个选择对比来看看哪种方案更快，需要认真注意细节。

与其打造你自己的统计有效的性能测试逻辑，不如直接使用 Benchmark.js 库，它已经为你实现了这些。但是，编写测试要小心，因为我们很容易就会构造一个看似有效实际却有缺陷的测试，即使是微小的差异也可能扭曲结果，使其完全不可靠。

从尽可能多的环境中得到尽可能多的测试结果以消除硬件 / 设备的偏差，这一点很重要。jsPerf.com 是很好的网站，用于众包性能测试运行。

遗憾的是，很多常用的性能测试执迷于无关紧要的微观性能细节，比如 x++ 对比 ++x。编写好的测试意味着理解如何关注大局，比如关键路径上的优化以及避免落入类似不同的 JavaScript 实现细节这样的陷阱中。

尾调用优化是 ES6 要求的一种优化方法。它使 JavaScript 中原本不可能的一些递归模式变得实际。TCO 允许一个函数在结尾处调用另外一个函数来执行，不需要任何额外资源。这意味着，对递归算法来说，引擎不再需要限制栈深度。

附录 A

asynquence 库

第 1 章和第 2 章介绍了很多典型异步编程模式的细节，以及通常如何通过回调来实现。但是，我们也看到了为什么回调在功能上有致命的限制性，这进而引出了第 3 章和第 4 章中对 Promise 和生成器的介绍。它们为构建异步提供了更坚固、更可信任、更合理的基础。

本书中多次提到我自己的异步库 asynquence（http://github.com/getify/asynquence）（async+sequence=asynquence）。现在我要简单介绍一下它的工作方式以及为什么其独特的设计是重要和有用的。

附录 B 将探索几种高级的异步模式，但你可能需要一个库才能让这些模式变得足够实用。我们将使用 asynquence 来表达这些模式，所以需要花费一点时间先来了解一下这个库。

当然，asynquence 并不是异步编程的唯一好选择。在这个领域的确有很多非常好的库。但是，通过把所有这些模式中最好的部分组合到单个库中，asynquence 提供了一个独特的视角，而且它还是建立在单个基本抽象之上的：（异步）序列。

我的前提是，高级 JavaScript 编程通常需要把各种不同的异步模式一块块编织在一起，而这通常是完全留给开发者来实现的。asynquence 不再需要引入分别关注异步不同方面的两个或更多异步库，而是把它们统一为序列步骤的不同变体，这样就只需要学习和部署一个核心库。

通过 asynquence 使得 Promise 风格语义的异步流程控制编程完成起来非常简单。我确信这是很有价值的，所以这也就是为什么这里只关注这一个库。

首先，我要解释 asynquence 背后的设计原理，然后通过代码示例展示其 API 的工作方式。

A.1 序列与抽象设计

理解 asynquence 要从理解一个基本的抽象开始：一个任务的一系列步骤，不管各自是同步的还是异步的，都可以整合起来看成一个*序列*（sequence）。换句话说，一个序列代表了一个任务的容器，由完成这个任务的独立（可能是异步）的步骤组成。

序列中的每个步骤在形式上通过一个 Promise（参见第 3 章）控制。也就是说，添加到序列中的每个步骤隐式地创建了一个 Promise 连接到之前序列的尾端。由于 Promise 的语义，序列中每个单个步骤的运行都是异步的，即使是同步完成这个步骤也是如此。

另外，序列通常是从一个步骤到一个步骤线性处理的，也就是说步骤 2 要在步骤 1 完成之后开始，以此类推。

当然，可以从现有的序列分叉（fork）出新的序列，这意味着主序列到达流程中的这个点上就会发生分叉。也可以通过各种方法合并序列，包括在流程中的特定点上让一个序列包含另一个序列。

序列有点类似于 Promise 链。然而，通过 Promise 链没有“句柄”可以拿到整个链的引用。拿到的 Promise 引用只代表链中当前步骤以及后面的其他步骤。本质上说，你无法持有一个 Promise 链的引用，除非你拿到链中第一个 Promise 的引用。

很多情况下，持有到整个序列的引用是非常有用的。其中最重要的就是序列的停止或取消。就像我们在第 3 章扩展讨论过的，Promise 本身永远不应该可以被取消，因为这违背了一个基本的设计规则：外部不可变性。

但对序列来说，并没有这样的不可变设计原则，主要是因为序列不会被作为需要不可变值语义的未来值容器来传递。因此，序列是处理停止或取消行为的正确抽象层级。asynquence 序列可以在任何时间被 `abort()`，序列会在这个时间点停止，不会因为任何理由继续进行下去。

之所以选择在 Promise 之上建立序列抽象用于流程控制的目的，还有很多别的理由。

第一，Promise 链接更多是一个手工过程。一旦开始在大范围的程序内创建和链接 Promise，事情就可能会变得十分乏味。这种麻烦可能会极大阻碍开发者在 Promise 本来十分适用的地方使用 Promise。

抽象的目的是减少重复样板代码和避免乏味，所以序列抽象是针对这个问题的一个很好的解决方案。通过 Promise，你的关注点放在各个步骤上，几乎没有假定链的继续。而如果使用序列的话，情况则正好相反：会假定序列持续，会有更多的步骤无限地附加上来。

当考虑到更高阶的 Promise 模式时（在 `race([..])` 和 `all([..])` 之上），这种抽象复杂性的

降低就格外强大了。

举例来说，在序列当中，你可能想要表达一个在概念上类似 `try..catch` 的步骤，这个步骤总是返回成功，要么是想要的主功能成功决议，要么是一个标识被捕获错误的非错误信号。或者，你可能想要表达一个类似 `retry/until` 的循环，其中会持续重复试验同样的步骤直到成功为止。

如果只使用 Promise 原语表达的话，这些种类的抽象工作量可并不小，在现有的 Promise 链当中实现也并不优美。但是，如果把你的思路抽象为序列，并把步骤当作对 Promise 的封装，那么这样的步骤封装就可以隐藏这些细节，节省你的精力，从而让你以最合理的方式考虑流程控制，不需要为细节所困。

第二，可能也是最重要的一点，以序列中的步骤这样的视角来考量异步流程控制，这样就可以把每个单独步骤涉及的异步类型等细节抽象出去。在此之下，总是由 Promise 来控制着这个步骤，但是表面上，这个步骤看起来要么类似 continuation 回调（最简单的默认情况），要么类似真正的 Promise，要么就类似完整运行的生成器，要么……希望你已经理解了我的意思。

第三，序列很容易被改造，以适应不同的思考模式，比如基于事件、基于流、基于响应的编码。asynquence 提供了一个模式，我称之为*响应序列*（reactive sequence，后面会介绍），是 RxJS（Reactive 扩展）中 reactive observable 思想的一个变体。它利用重复的事件每次启动一个新的序列实例。Promise 只有一次，所以单独使用 Promise 对于表达重复的异步是很笨拙的。

另外，有一种思路在一个被我称为*可迭代序列*的模式中反转了决议和控制功能。不再是每个单独的步骤在内部控制自己的完成（于是有序列前进），事实上，这个序列被反转了，前进控制是通过外部的迭代器，并且可迭代序列中的每个步骤只响应 `next(..)` 迭代器控制。

本附录在后面会介绍这些不同的变体，所以不必担心刚才的讨论过于简略。

需要记住的是，对复杂异步来说，比起只用 Promise（Promise 链）或只用生成器，序列是更强大更合理的抽象。asynquence 的设计目标就是在合适的层级表达这个抽象，使异步编程更容易理解、更有乐趣。

A.2 asynquence API

首先，创建序列（一个 asynquence 实例）的方法是通过函数 `ASQ(..)`。没有参数的 `ASQ()` 调用会创建一个空的初始序列，而向 `ASQ(..)` 函数传递一个或多个值，则会创建一个序列，其中每个参数表示序列中的一个初始步骤。

为了使这里所有的代码示例起见，我将在全局浏览器使用 asynquence 顶级标识符：ASQ。如果你通过模块系统（浏览器或服务器）包含并使用 asynquence 的话，当然可以定义任何你喜欢的符号，asynquence 不会在意！

这里讨论的许多 API 方法是构建在 asynquence 的核心库中的，还有其他一些是通过包含可选的 contrib 插件包提供的。请参考 asynquence 文档（http://github.com/getify/asynquence），确定一个方法是内建的还是通过插件定义的。

A.2.1 步骤

如果一个函数表示序列中的一个普通步骤，那调用这个函数时第一个参数是 continuation 回调，所有后续的参数都是从前一个步骤传递过来的消息。直到这个 continuation 回调被调用后，这个步骤才完成。一旦它被调用，传给它的所有参数将会作为消息传入序列中的下一个步骤。

要向序列中添加额外的普通步骤，可以调用 then(..)（这本质上和 ASQ(..) 调用的语义完全相同）：

```
ASQ(
    // 步骤1
    function(done){
        setTimeout( function(){
            done( "Hello" );
        }, 100 );
    },
    // 步骤2
    function(done,greeting) {
        setTimeout( function(){
            done( greeting + " World" );
        }, 100 );
    }
)
// 步骤3
.then( function(done,msg){
    setTimeout( function(){
        done( msg.toUpperCase() );
    }, 100 );
} )
// 步骤4
.then( function(done,msg){
    console.log( msg );         // HELLO WORLD
} );
```

尽管 then(..) 和原生 Promise API 名称相同，但是这个 then(..) 是不一样的。你可以向 then(..) 传递任意多个函数或值，其中每一个都会作为一个独立步骤。其中并不涉及两个回调的完成 / 拒绝语义。

和 Promise 不同的一点是：在 Promise 中，如果你要把一个 Promise 链接到下一个，需要创建这个 Promise 并通过 then(..) 完成回调函数返回这个 Promise；而使用 asynquence，你需要做的就是调用 continuation 回调——我一直称之为 done()，但你可以随便给它取什么名字——并可选择性将完成消息传递给它作为参数。

通过 then(..) 定义的每个步骤都被假定为异步的。如果你有一个同步的步骤，那你可以直接调用 done(..)，也可以使用更简单的步骤辅助函数 val(..)。

```
// 步骤1(同步)
ASQ( function(done){
    done( "Hello" );    // 手工同步
} )
// 步骤2(同步)
.val( function(greeting){
    return greeting + " World";
} )
// 步骤3(异步)
.then( function(done,msg){
    setTimeout( function(){
        done( msg.toUpperCase() );
    }, 100 );
} )
// 步骤4(同步)
.val( function(msg){
    console.log( msg );
} );
```

可以看到，通过 val(..) 调用的步骤并不接受 continuation 回调，因为这一部分已经为你假定了，结果就是参数列表没那么凌乱！如果要给下一个步骤发送消息的话，只需要使用 return。

可以把 val(..) 看作一个表示同步的“只有值”的步骤，可以用于同步值运算、日志记录及其他类似的操作。

A.2.2 错误

与 Promise 相比，asynquence 一个重要的不同之处就是错误处理。

通过 Promise，链中每个独立的 Promise（步骤）都可以有自己独立的错误，接下来的每个步骤都能处理（或者不处理）这个错误。这个语义的主要原因（再次）来自于对单独 Promise 的关注而不是将链（序列）作为整体。

我相信，多数时候，序列中某个部分的错误通常是不可恢复的，所以序列中后续的步骤也就没有意义了，应该跳过。因此，在默认情况下，一个序列中任何一个步骤出错都会把整个序列抛入出错模式中，剩余的普通步骤会被忽略。

如果你确实需要一个错误可恢复的步骤，有几种不同的 API 方法可以实现，比如 `try(..)`（前面作为一种 `try..catch` 步骤提到过）或者 `until(..)`（一个重试循环，会尝试步骤直到成功或者你手工使用 `break()`）。asynquence 甚至还有 `pThen(..)` 和 `pCatch(..)` 方法，它们和普通的 Promise `then(..)` 和 `catch(..)` 的工作方式完全一样（参见第 3 章）。因此，如果你愿意的话，可以定制序列当中的错误处理。

关键在于，你有两种选择，但根据我的经验，更常用的是默认的那个。通过 Promise，为了使一个步骤链在出错时忽略所有步骤，你需要小心地避免在任意步骤中注册拒绝处理函数。否则的话，这个错误就会因被当作已经处理的而被吞掉，同时这个序列可能会继续（很可能是出乎意料的）。要正确可靠地处理这一类需求有点棘手。

asynquence 为注册一个序列错误通知处理函数提供了一个 `or(..)` 序列方法。这个方法还有一个别名，`onerror(..)`。你可以在序列的任何地方调用这个方法，也可以注册任意多个处理函数。这很容易实现多个不同的消费者在同一个序列上侦听，以得知它有没有失败。从这个角度来说，它有点类似错误事件处理函数。

和使用 Promise 类似，所有的 JavaScript 异常都成为了序列错误，或者你也可以编写代码来发送一个序列错误信号：

```
var sq = ASQ( function(done){
    setTimeout( function(){
        // 为序列发送出错信号
        done.fail( "Oops" );
    }, 100 );
} )
.then( function(done){
    // 不会到达这里
} )
.or( function(err){
    console.log( err );         // Oops
} )
.then( function(done){
    // 也不会到达这里
} );

// 之后

sq.or( function(err){
    console.log( err );         // Oops
} );
```

asynquence 的错误处理和原生 Promise 还有一个非常重要的区别，就是默认状态下未处理异常的行为。正如在第 3 章中讨论过的，没有注册拒绝处理函数的被拒绝 Promise 就会默默地持有（即吞掉）这个错误。你需要记得总要在链的尾端添加一个最后的 `catch(..)`。

而在 asynquence 中，这个假定是相反的。