目　录

致谢

当前文档 《你不懂JS: 异步与性能（You Dont Know JS）》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-02-09。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/You-Dont-Know-JS-async-performance

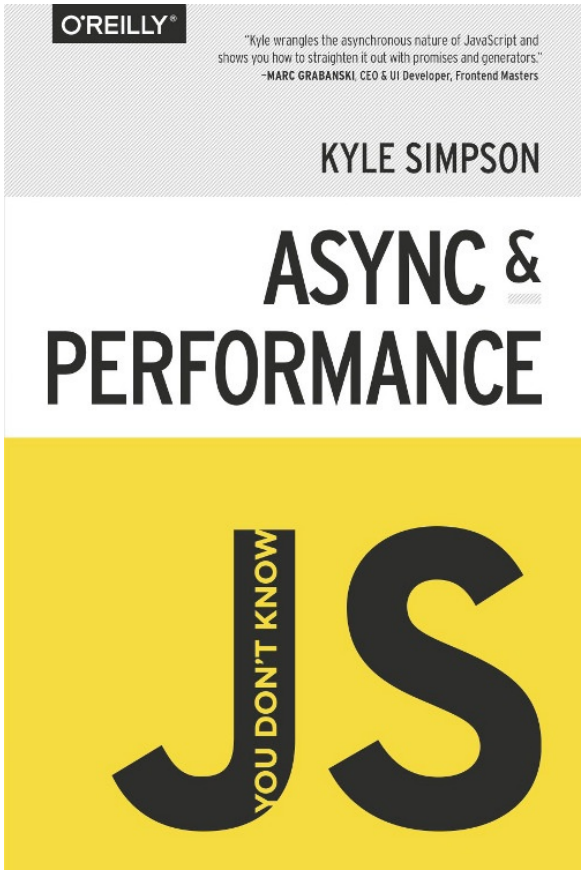
书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

阅前必读

你不懂JS： 异步与性能



从O'Reilly购买数字/印刷版

序

序

多年以前，我的雇主十分信任我来让我进行面试。如果我们要找某些拥有JavaScript技能的人，我的问卷的第一行是…实际上这不是真的，我首先会问问应聘者是否需要上个卫生间或者喝些饮料，因为平静是很重要的，但是一旦我确信可以和应聘者进行流畅的交流，我就要开始考察这位应聘者是否懂得JavaScript，还是只懂得jQuery。

并不是jQuery有什么错。它使你不必真的懂得JavaScript就可以做很多事，这是一个特性而不是一个bug。但是如果这份工作需要关于JavaScript性能和可维护性上的高级技能，你就需要一些懂得jQuery这样的库是如何组装在一起的人。你需要能够像他们一样操控JavaScript的核心。

如果我想对某人的核心JavaScript技能取得一些了解，我最感兴趣就是他们如何使用闭包（你已经读过这个系列的那本书了，对吧？），以及如何最大限度地利用异步性，而这就是这本书带给我们的。

对于初学者，你将被带领着学习回调，它是异步编程的面包和黄油。当然，面包和黄油并不能做一顿特别令人满意的午餐，但是下一课满是非常美味的promise！

如果你不懂得promise，现在是学习的时候了。现在在JavaScript和DOM中，Promise是提供异步返回值的官方方法。所有未来的异步DOM API都将使用它们，而且有许多已经这样做了，所以做好准备！在本次写作时，Promise已经在大多数主流浏览器中获得了支持，IE也很快会支持。一旦你完成了这一课，我希望你离开教室去学习下一刻，Generator。

Generator不声不响地溜进了Chrome和Firefox的稳定版本，因为，老实说，它们的复杂程度要比有趣程度大多了。或者说，直到我看到它们与promise组合起来之前我都是这么认为的。在此，它们成为了增强可读性和可维护性的重要工具。

至于甜点，好吧，我不会把惊喜放坏了，准备好凝视JavaScript的未来吧！许多特性在并发性和异步性上给了你越来越多的控制权。

好吧，我不会继续挡着你享受这本书了，让好戏开始吧！如果你已经在读这篇序之前度过了这本书的一些部分，给你10点异步加分！你值得拥有！

Jake Archibald

jakearchibald.com, @jaffathecake

Google Chrome 技术推广部

第一章: 异步:现在与稍后

第一章： 异步： 现在与稍后

在像JavaScript这样的语言中最重要但经常被误解的编程技术之一，就是如何表达和操作跨越一段时间的程序行为。

这不仅仅是关于从 `for` 循环开始到 `for` 循环结束之间发生的事情，当然它确实要花 一些时间（几微秒到几毫秒）才能完成。它是关于你的程序 现在 运行的部分，和你的程序 稍后 运行的另一部分之间发生的事情—现在 和 稍后 之间有一个间隙，在这个间隙中你的程序没有活跃地执行。

几乎所有被编写过的（特别是用JS）大型程序都不得不用这样或那样的方法来管理这个间隙，不管是等待用户输入，从数据库或文件系统请求数据，通过网络发送数据并等待应答，还是在规定的时间间隔重复某些任务（比如动画）。在所有这些各种方法中，你的程序都不得不跨越时间间隙管理状态。就像在伦敦众所周知的一句话（地铁门与月台间的缝隙）：“小心间隙。”

实际上，你程序中 现在 与 稍后 的部分之间的关系，就是异步编程的核心。

可以确定的是，异步编程在JS的最开始就出现了。但是大多数开发者从没认真地考虑过它到底是如何，为什么出现在他们的程序中的，也没有探索过 其他 处理异步的方式。足够好 的方法总是老实巴交的回调函数。今天还有许多人坚持认为回调就绰绰有余了。

但是JS在使用范围和复杂性上不停地生长，作为运行在浏览器，服务器和每种可能的设备上的头等编程语言，为了适应它不断扩大的要求，我们在管理异步上感受到的痛苦日趋严重，人们迫切地需要一种更强大更合理的处理方法。

虽然眼前这一切看起来很抽象，但我保证，随着我们通读这本书你会更完整且坚实地解决它。在接下来的几章中我们将会探索各种异步JavaScript编程的新兴技术。

但在接触它们之前，我们将不得不更深刻地理解异步是什么，以及它在JS中如何运行。

链接

- 块儿（Chunks）中的程序
- 事件轮询（Event Loop）
- 并行线程
- 并发
- Jobs
- 语句排序

- 复习

块儿（Chunks）中的程序

块儿（Chunks）中的程序

你可能将你的JS程序写在一个 *.js* 文件中，但几乎可以确定你的程序是由几个代码块儿构成的，仅有其中的一个将会在 现在 执行，而其他的将会在 稍后 执行。最常见的 代码块儿 单位是 `function` 。

大多数刚接触JS的开发者都可能会有的问题是，稍后 并不严格且立即地在 现在 之后发生。换句话说，根据定义，现在 不能完成的任务将会异步地完成，而且我们因此不会有你可能在直觉上期望或想要的阻塞行为。

考虑这段代码：

```
1. // ajax(..)是某个包中任意的Ajax函数
2. var data = ajax( "http://some.url.1" );
3.
4. console.log( data );
5. // 噢！`data`一般不会有Ajax的结果
```

你可能意识到Ajax请求不会同步地完成，这意味着 `ajax(..)` 函数还没有任何返回的值可以赋值给变量 `data` 。如果 `ajax(..)` 在应答返回之前 能够 阻塞，那么 `data = ..` 赋值将会正常工作。

但那不是我们使用Ajax的方式。我们 现在 制造一个异步的Ajax请求，直到 稍后 我们才会得到结果。

从 现在 “等到” 稍后 最简单的（但绝对不是唯一的，或最好的）方法，通常称为回调函数：

```
1. // ajax(..) 是某个包中任意的Ajax函数
2. ajax( "http://some.url.1", function myCallbackFunction(data){
3.
4.     console.log( data ); // Yay, 我得到了一些`data`!
5.
6. } );
```

警告： 你可能听说过发起同步的Ajax请求是可能的。虽然在技术上是这样的，但你永远，永远不应该在任何情况下这样做，因为它将锁定浏览器的UI（按钮，菜单，滚动条，等等）而且阻止用户与任何东西互动。这是一个非常差劲的主意，你应当永远回避它。

在你提出抗议之前，不，你渴望避免混乱的回调不是使用阻塞的，同步的Ajax的正当理由。

举个例子，考虑下面的代码：

```
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
```

这个程序中有两个代码块儿：现在 将会运行的东西，和 稍后 将会运行的东西。这两个代码块分别是什么应当十分明显，但还是让我们以最明确的方式指出来：

现在：

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

function later() { .. }

var answer = now();

setTimeout( later, 1000 );
```

稍后：

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

你的程序一执行，现在 代码块儿就会立即运行。但 `setTimeout(..)` 还设置了一个 稍后 会发生的事件（一个超时事件），所以 `later()` 函数的内容将会在一段时间后（从现在开始1000毫秒）被执行。

每当你将一部分代码包进 `function` 并且规定它应当为了响应某些事件而执行（定时器，鼠标点击，Ajax应答等等），你就创建了一个 稍后 代码块儿，也因此在你的程序中引入了异步。

异步控制台

关于 `console.*` 方法如何工作，没有相应的语言规范或一组需求—它们不是JavaScript官方的一部分，而是由 宿主环境 添加到JS上的（见本丛书的 类型与文法）。

所以，不同的浏览器和JS环境各自为战，这有时会导致令人困惑的行为。

特别地，有些浏览器和某些条件下， `console.log(..)` 实际上不会立即输出它得到的东西。这个现象的主要原因可能是因为I/O处理很慢，而且是许多程序的阻塞部分（不仅是JS）。所以，对一个浏览器来说，可能的性能更好的处理方式是（从网页/UI的角度看），在后台异步地处理 `console` I/O，而你也许根本不知道它发生了。

虽然不是很常见，但是一种可能被观察到（不是从代码本身，而是从外部）的场景是：

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

// 稍后
console.log( a ); // ??

// 再稍后
a.index++;
```

我们一般希望看到的是，就在 `console.log(..)` 语句被执行的那一刻，对象 `a` 被取得一个快照，打印出如 `{ index: 1 }` 的内容，如此在下一个语句 `a.index++` 执行时，它修改不同于 `a` 的输出，或者严格的在 `a` 的输出之后的某些东西。

大多数时候，上面的代码将会在你的开发者工具控制台中产生一个你期望的对象表现形式。但是同样的代码也可能运行在这样的情况下：浏览器告诉后台它需要推迟控制台I/O，这时，在对象在控制台中被表示的那个时间点， `a.index++` 已经执行了，所以它将显示 `{ index: 2 }` 。

到底在什么条件下 `console` I/O将被推迟是不确定的，甚至它能不能被观察到都是不确定的。只能当你在调试过程中遇到问题时—对象在 `console.log(..)` 语句之后被修改，但你却意外地看到了修改后的内容—意识到I/O的这种可能的异步性。

注意： 如果你遇到了这种罕见的情况，最好的选择是使用JS调试器的断点，而不是依赖 `console` 的输出。第二好的选择是通过将目标对象序列化为一个 `string` 强制取得一个它的快照，比如用 `JSON.stringify(..)` 。

事件轮询（Event Loop）

事件轮询（Event Loop）

让我们来做一个（也许是令人震惊的）声明：尽管明确地允许异步JS代码（就像我们刚看到的超时），但是实际上，直到最近（ES6）为止，JavaScript本身从来没有任何内建的异步概念。

什么！？ 这听起来简直是疯了，对吧？事实上，它是真的。JS引擎本身除了在某个在被要求的时刻执行你程序的一个单独的代码块外，没有做过任何其他的事情。

“被’谁’要求”？这才是重要的部分！

JS引擎没有运行在隔离的区域。它运行在一个 宿主环境 中，对大多数开发者来说这个宿主环境就是浏览器。在过去的几年中（但不特指这几年），JS超越了浏览器的界限进入到了其他环境中，比如服务器，通过Node.js这样的东西。其实，今天JavaScript已经被嵌入到所有种类的设备中，从机器人到电灯泡儿。

所有这些环境的一个共通的“线程”（一个“不那么微妙”的异步玩笑，不管怎样）是，他们都有一种机制：在每次调用JS引擎时，可以 随着时间的推移 执行你的程序的多个代码块儿，这称为“事件轮询（Event Loop）”。

换句话说，JS引擎对 时间 没有天生的感觉，反而是一个任意JS代码段的按需执行环境。是它周围的环境在不停地安排“事件”（JS代码的执行）。

那么，举例来说，当你的JS程序发起一个从服务器取得数据的Ajax请求时，你在一个函数（通常称为回调）中建立好“应答”代码，然后JS引擎就会告诉宿主环境，“嘿，我就要暂时停止执行了，但不管你什么时候完成了这个网络请求，而且你还得到一些数据的话，请 回来调 这个函数。”

然后浏览器就会为网络的应答设置一个监听器，当它有东西要交给你的时候，它会通过将回调函数插入 事件轮询 来安排它的执行。

那么什么是 事件轮询？

让我们先通过一些假想代码来对它形成一个概念：

```
// `eventLoop`是一个像队列一样的数组（先进先出）
var eventLoop = [ ];
var event;

// “永远”执行
while (true) {
    // 执行一个"tick"
```

```
    if (eventLoop.length > 0) {
        // 在队列中取得下一个事件
        event = eventLoop.shift();

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

当然，这只是一个用来展示概念的大幅简化的假想代码。但是对于帮助我们建立更好的理解来说应该够了。

如你所见，有一个通过 `while` 循环来表现的持续不断的循环，这个循环的每一次迭代称为一个“tick”。在每一个“tick”中，如果队列中有一个事件在等待，它就会被取出执行。这些事件就是你的函数回调。

很重要并需要注意的是， `setTimeout(..)` 不会将你的回调放在事件轮询队列上。它设置一个定时器；当这个定时器超时的时候，环境才会把你的回调放进事件轮询，这样在某个未来的tick中它将会被取出执行。

如果在那时事件轮询队列中已经有了20个事件会怎么样？你的回调要等待。它会排到队列最后—没有一般的方法可以插队和跳到队列的最前方。这就解释了为什么 `setTimeout(..)` 计时器可能不会完美地按照预计时间触发。你得到一个保证（粗略地说）：你的回调不会再你指定的时间间隔之前被触发，但是可能会在这个时间间隔之后被触发，具体要看事件队列的状态。

换句话说，你的程序通常被打断成许多小的代码块儿，它们一个接一个地在事件轮询队列中执行。而且从技术上说，其他与你的程序没有直接关系的事件也可以穿插在队列中。

注意： 我们提到了“直到最近”，暗示着ES6改变了事件轮询队列在何处被管理的性质。这主要是一个正式的技术规范，ES6现在明确地指出了事件轮询应当如何工作，这意味着它技术上属于JS引擎应当关心的范畴内，而不仅仅是 宿主环境。这么做的一个主要原因是为了引入ES6的Promises（我们将在第三章讨论），因为人们需要有能力对事件轮询队列的排队操作进行直接，细粒度的控制（参见“协作”一节中关于 `setTimeout(..0)` 的讨论）。

并行线程

并行线程

“异步”与“并行”两个词经常被混为一谈，但它们实际上是十分不同的。记住，异步是关于 现在 与 稍后 之间的间隙。但并行是关于可以同时发生的事情。

关于并行计算最常见的工具就是进程与线程。进程和线程独立地，可能同时地执行：在不同的处理器上，甚至在不同的计算机上，而多个线程可以共享一个进程的内存资源。

相比之下，一个事件轮询将它的工作打碎成一系列任务并串行地执行它们，不允许并行访问和更改共享的内存。并行与“串行”可能以在不同线程上的事件轮询协作的形式共存。

并行线程执行的穿插，与异步事件的穿插发生在完全不同的粒度等级上：

比如：

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

虽然 `later()` 的整个内容将被当做一个事件轮询队列的实体，但当考虑到将要执行这段代码的线程时，实际上也许会有许多不同的底层操作。比如， `answer = answer * 2` 首先需要读取当前 `answer` 的值，再把 `2` 放在某个地方，然后进行乘法计算，最后把结果存回到 `answer` 。

在一个单线程环境中，线程队列中的内容都是底层操作真的无关紧要，因为没有什么可以打断线程。但如果你有一个并行系统，在同一个程序中有两个不同的线程，你很可能会得到无法预测的行为：

考虑这段代码：

```
var a = 20;

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

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

```
10.
11. // ajax(..) 是一个给定的库中的随意Ajax函数
12. ajax( "http://some.url.1", foo );
13. ajax( "http://some.url.2", bar );
```

在JavaScript的单线程行为下，如果 `foo()` 在 `bar()` 之前执行，结果 `a` 是 `42` ，但如果 `bar()` 在 `foo()` 之前执行，结果 `a` 将是 `41` 。

如果JS事件共享相同的并列执行数据，问题将会变得微妙得多。考虑这两个假想代码段，它们分别描述了运行 `foo()` 和 `bar()` 中代码的线程将要执行的任务，并考虑如果它们在完全相同的时刻运行会发生什么：

线程1（ `X` 和 `Y` 是临时的内存位置）：

```
1. foo():
2.     a. 将`a`的值读取到`X`
3.     b. 将`1`存入`Y`
4.     c. 把`X`和`Y`相加，将结果存入`X`
5.   d. 将`X`的值存入`a`
```

线程2（ `X` 和 `Y` 是临时的内存位置）：

```
1. bar():
2.   a. 将`a`的值读取到`X`
3.     b. 将`2`存入`Y`
4.     c. 把`X`和`Y`相乘，将结果存入`X`
5.     d. 将`X`的值存入`a`
```

现在，让我们假定这两个线程在并行执行。你可能发现了问题，对吧？它们在临时的步骤中使用共享的内存位置 `X` 和 `Y` 。

如果步骤像这样发生， `a` 的最终结果什么？

```
1. 1a  (将`a`的值读取到`X`   ==> `20`)
2. 2a  (将`a`的值读取到`X`   ==> `20`)
3. 1b  (将`1`存入`Y`   ==> `1`)
4. 2b  (将`2`存入`Y`   ==> `2`)
5. 1c  (把`X`和`Y`相加，将结果存入`X`   ==> `22`)
6. 1d  (将`X`的值存入`a`   ==> `22`)
7. 2c  (把`X`和`Y`相乘，将结果存入`X`   ==> `44`)
8. 2d  (将`X`的值存入`a`   ==> `44`)
```

`a` 中的结果将是 `44` 。那么这种顺序呢？

```
1a  (将`a`的值读取到`X`   ==> `20`)
2a  (将`a`的值读取到`X`   ==> `20`)
2b  (将`2`存入`Y`   ==> `2`)
1b  (将`1`存入`Y`   ==> `1`)
2c  (把`X`和`Y`相乘，将结果存入`X`   ==> `20`)
1c  (把`X`和`Y`相加，将结果存入`X`   ==> `21`)
1d  (将`X`的值存入`a`   ==> `21`)
2d  (将`X`的值存入`a`   ==> `21`)
```

`a` 中的结果将是 `21` 。

所以，关于线程的编程十分刁钻，因为如果你不采取特殊的步骤来防止这样的干扰/穿插，你会得到令人非常诧异的，不确定的行为。这通常让人头疼。

JavaScript从不跨线程共享数据，这意味着不必关心这一层的不确定性。但这并不意味着JS总是确定性的。记得前面 `foo()` 和 `bar()` 的相对顺序产生两个不同的结果吗（ `41` 或 `42` ）？

注意： 可能还不明显，但不是所有的不确定性都是坏的。有时候它无关紧要，有时候它是故意的。我们会在本章和后续几章中看到更多的例子。

运行至完成

因为JavaScript是单线程的， `foo()` （和 `bar()` ）中的代码是原子性的，这意味着一旦 `foo()` 开始运行，它的全部代码都会在 `bar()` 中的任何代码可以运行之前执行完成，反之亦然。这称为“运行至完成”行为。

事实上，运行至完成的语义会在 `foo()` 与 `bar()` 中有更多的代码时更明显，比如：

```
var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

因为 `foo()` 不能被 `bar()` 打断，而且 `bar()` 不能被 `foo()` 打断，所以这个程序根据哪一个先执行只有两种可能的结果——如果线程存在， `foo()` 和 `bar()` 中的每一个语句都可能被穿插，可能的结果数量将会极大地增长！

代码块儿1是同步的（现在 发生），但代码块儿2和3是异步的（稍后 发生），这意味着它们的执行将会被时间的间隙分开。

代码块儿1：

```
var a = 1;
var b = 2;
```

代码块儿2（ `foo()` ）：

```
a++;
b = b * a;
a = b + 3;
```

代码块儿3（ `bar()` ）：

```
b--;
a = 8 + b;
b = a * 2;
```

代码块儿2和3哪一个都有可能先执行，所以这个程序有两个可能的结果，正如这里展示的：

结果1：

```
var a = 1;
var b = 2;

// foo()
a++;
b = b * a;
a = b + 3;

// bar()
b--;
a = 8 + b;
b = a * 2;

a; // 11
b; // 22
```

结果2：

```
var a = 1;
var b = 2;

// bar()
b--;
a = 8 + b;
b = a * 2;

// foo()
a++;
b = b * a;
a = b + 3;

a; // 183
b; // 180
```

同一段代码有两种结果仍然意味着不确定性！但是这是在函数（事件）顺序的水平上，而不是在使用线程时语句顺序的水平上（或者说，实际上是表达式操作的顺序上）。换句话说，他比线程更具有 确定性。

当套用到JavaScript行为时，这种函数顺序的不确定性通常称为“竞合状态”，因为 `foo()` 和 `bar()` 在互相竞争看谁会先运行。明确地说，它是一个“竞合状态”因为你不能可靠地预测 `a` 与 `b` 将如何产生。

注意： 如果在JS中不知怎的有一个函数没有运行至完成的行为，我们会有更多可能的结果，对吧？ES6中引入一个这样的东西（见第四章“生成器”），但现在不要担心，我们会回头讨论它。

并发

并发

让我们想象一个网站，它显示一个随着用户向下滚动而逐步加载的状态更新列表（就像社交网络的新消息）。要使这样的特性正确工作，（至少）需要两个分离的“进程” 同时 执行（在同一个时间跨度内，但没必要是同一个时间点）。

注意： 我们在这里使用带引号的“进程”，因为它们不是计算机科学意义上的真正的操作系统级别的进程。它们是虚拟进程，或者说任务，表示一组逻辑上关联，串行顺序的操作。我们将简单地使用“进程”而非“任务”，因为在术语层面它与我们讨论的概念的定义相匹配。

第一个“进程”将响应当用户向下滚动页面时触发的 `onscroll` 事件（发起取得新内容的Ajax请求）。第二个“进程”将接收返回的Ajax应答（将内容绘制在页面上）。

显然，如果用户向下滚动的足够快，你也许会看到在第一个应答返回并处理期间，有两个或更多的 `onscroll` 事件被触发，因此你将使 `onscroll` 事件和Ajax应答事件迅速触发，互相穿插在一起。

并发是当两个或多个“进程”在同一时间段内同时执行，无论构成它们的各个操作是否 并行地（在同一时刻不同的处理器或内核）发生。你可以认为并发是“进程”级别的（或任务级别）的并行机制，而不是操作级别的并行机制（分割进程的线程）。

注意： 并发还引入了这些“进程”间彼此互动的概念。我们稍后会讨论它。

在一个给定的时间跨度内（用户可以滚动的那几秒），让我们将每个独立的“进程”作为一系列事件/操作描绘出来：

“线程”1（ `onscroll` 事件）：

```
1. onscroll, request 1
2. onscroll, request 2
3. onscroll, request 3
4. onscroll, request 4
5. onscroll, request 5
6. onscroll, request 6
7. onscroll, request 7
```

“线程”2（Ajax应答事件）：

```
response 1
response 2
response 3
response 4
response 5
response 6
response 7
```

一个 `onscroll` 事件与一个Ajax应答事件很有可能在同一个 时刻 都准备好被处理了。比如我们在一个时间线上描绘一下这些事件的话：

```
onscroll, request 1
onscroll, request 2          response 1
onscroll, request 3          response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6          response 4
onscroll, request 7
response 6
response 5
response 7
```

但是，回到本章前面的事件轮询概念，JS一次只能处理一个事件，所以不是 `onscroll, request 2` 首先发生就是 `response 1` 首先发生，但是他们不可能完全在同一时刻发生。就像学校食堂的孩子们一样，不管他们在门口挤成什么样，他们最后都不得不排成一个队来打饭！

让我们来描绘一下所有这些事件在事件轮询队列上穿插的情况：

事件轮询队列：

```
onscroll, request 1   <--- 进程1开始
onscroll, request 2
response 1            <--- 进程2开始
onscroll, request 3
response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
response 4
onscroll, request 7   <--- 进程1结束
response 6
response 5
```

```
response 7            <--- 进程2结束
```

“进程1”和“进程2”并发地运行（任务级别的并行），但是它们的个别事件在事件轮询队列上顺序地运行。

顺便说一句，注意到 `response 6` 和 `response 5` 没有按照预想的顺序应答吗？

单线程事件轮询是并发的一种表达（当然还有其他的表达，我们稍后讨论）。

非互动

在同一个程序中两个或更多的“进程”在穿插它们的步骤/事件时，如果它们的任务之间没有联系，那么他们就没必要互动。如果它们不互动，不确定性就是完全可以接受的。

举个例子：

```
var res = {};

function foo(results) {
    res.foo = results;
}

function bar(results) {
    res.bar = results;
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

`foo()` 和 `bar()` 是两个并发的“进程”，而且它们被触发的顺序是不确定的。但对我们的程序的结构来讲它们的触发顺序无关紧要，因为它们的行为相互独立所以不需要互动。

这不是一个“竞合状态”Bug，因为这段代码总能够正确工作，与顺序无关。

互动

更常见的是，通过作用域和/或DOM，并发的“进程”将有必要间接地互动。当这样的互动将要发生时，你需要协调这些互动行为来防止前面讲述的“竞合状态”。

这里是两个由于隐含的顺序而互动的并发“进程”的例子，它 有时会出错：

```
var res = [];

function response(data) {
```

```
    res.push( data );
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

并发的“进程”是那两个将要处理Ajax应答的 `response()` 调用。它们谁都有可能先发生。

假定我们期望的行为是 `res[0]` 拥有 `"http://some.url.1"` 调用的结果，而 `res[1]` 拥有 `"http://some.url.2"` 调用的结果。有时候结果确实是这样，而有时候则相反，要看哪一个调用首先完成。很有可能，这种不确定性是一个“竞合状态”Bug。

注意： 在这些情况下要极其警惕你可能做出的主观臆测。比如这样的情况就没什么不寻常：一个开发者观察到 `"http://some.url.2"` 的应答“总是”比 `"http://some.url.1"` 要慢得多，也许有赖于它们所做的任务（比如，一个执行数据库任务而另一个只是取得静态文件），所以观察到的顺序看起来总是所期望的。就算两个请求都发到同一个服务器，而且它故意以确定的顺序应答，也不能 真正 保证应答回到浏览器的顺序。

所以，为了解决这样的竞合状态，你可以协调互动的顺序：

```
var res = [];

function response(data) {
    if (data.url == "http://some.url.1") {
        res[0] = data;
    }
    else if (data.url == "http://some.url.2") {
        res[1] = data;
    }
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

无论哪个Ajax应答首先返回，我们都考察它的 `data.url`（当然，假设这样的数据会从服务器返回）来找到应答数据应当在 `res` 数组中占有的位置。 `res[0]` 将总是持有 `"http://some.url.1"` 的结果，而 `res[1]` 将总是持有 `"http://some.url.2"` 的结果。通过简单的协调，我们消除了“竞合状态”的不确定性。

这个场景的同样道理可以适用于这样的情况：多个并发的函数调用通过共享的DOM互动，比如一个在更新 `<div>` 的内容而另一个在更新 `<div>` 的样式或属性（比如一旦DOM元素拥有内容就使它变得可见）。你可能不想在DOM元素拥有内容之前显示它，所以协调工作就必须保证正确顺序的互动。

没有协调的互动，有些并发的场景 总是出错（不仅仅是 有时）。考虑下面的代码：

```
var a, b;

function foo(x) {
    a = x * 2;
    baz();
}

function bar(y) {
    b = y * 2;
    baz();
}

function baz() {
    console.log(a + b);
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

在这个例子中，不管 `foo()` 和 `bar()` 谁先触发，总是会使 `baz()` 运行的太早了（ `a` 和 `b` 之一还是空的时候），但是第二个 `baz()` 调用将可以工作，因为 `a` 和 `b` 将都是可用的。

有许多不同的方法可以解决这个状态。这是简单的一种：

```
var a, b;

function foo(x) {
    a = x * 2;
    if (a && b) {
        baz();
    }
}

function bar(y) {
    b = y * 2;
    if (a && b) {
        baz();
    }
}

function baz() {
    console.log( a + b );
}

```

```
// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

`baz()` 调用周围的 `if (a && b)` 条件通常称为“大门”，因为我们不能确定 `a` 和 `b` 到来的顺序，但在打开大门（调用 `baz()` ）之前我们等待它们全部到达。

另一种你可能会遇到的并发互动状态有时称为“竞争”，但更准确地说应该叫“门闩”。它的行为特点是“先到者胜”。在这里不确定性是可以接受的，因为你明确指出“竞争”的终点线上只有一个胜利者。

考虑这段有问题的代码：

```
var a;

function foo(x) {
    a = x * 2;
    baz();
}

function bar(x) {
    a = x / 2;
    baz();
}

function baz() {
    console.log( a );
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

不管哪一个函数最后触发（ `foo()` 或 `bar()` ），它不仅会覆盖前一个函数对 `a` 的赋值，还会重复调用 `baz()` （不太可能是期望的）。

所以，我们可以用一个简单的门闩来协调互动，仅让第一个过去：

```
var a;

function foo(x) {
    if (a == undefined) {
        a = x * 2;
        baz();
    }
}

```

```
function bar(x) {
    if (a == undefined) {
        a = x / 2;
        baz();
    }
}

function baz() {
    console.log( a );
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

`if (a == undefined)` 条件仅会让 `foo()` 或 `bar()` 中的第一个通过，而第二个（以及后续所有的）调用将会被忽略。第二名什么也得不到！

注意： 在所有这些场景中，为了简化说明的目的我们都用了全局变量，这里我们没有任何理由需要这么做。只要我们讨论中的函数可以访问变量（通过作用域），它们就可以正常工作。依赖于词法作用域变量（参见本丛书的 作用域与闭包 ），和这些例子中实质上的全局变量，是这种并发协调形式的一个明显的缺点。在以后的几章中，我们会看到其他的在这方面干净得多的协调方法。

协作

另一种并发协调的表达称为“协作并发”，它并不那么看重在作用域中通过共享值互动（虽然这依然是允许的！）。它的目标是将一个长时间运行的“进程”打断为许多步骤或批处理，以至于其他的并发“进程”有机会将它们的操作穿插进事件轮询队列。

举个例子，考虑一个Ajax应答处理器，它需要遍历一个很长的结果列表来将值变形。我们将使用 `Array#map(..)` 来让代码短一些：

```
var res = [];

// `response(..)`从Ajax调用收到一个结果数组
function response(data) {
    // 连接到既存的`res`数组上
    res = res.concat(
        // 制造一个新的变形过的数组，所有的`data`值都翻倍
        data.map( function(val){
            return val * 2;
        } )
    );
}

// ajax(..) 是某个包中任意的Ajax函数
```

```
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

如果 `"http://some.url.1"` 首先返回它的结果，整个结果列表将会一次性映射进 `res` 。如果只有几千或更少的结果记录，一般来说不是什么大事。但假如有1千万个记录，那么就可能会花一段时间运行（在强大的笔记本电脑上花几秒钟，在移动设备上花的时间长得多，等等）。

当这样的“处理”运行时，页面上没有任何事情可以发生，包括不能有另一个 `response(..)` 调用，不能有UI更新，甚至不能有用户事件比如滚动，打字，按钮点击等。非常痛苦。

所以，为了制造协作性更强、更友好而且不独占事件轮询队列的并发系统，你可以在一个异步批处理中处理这些结果，在批处理的每一步都“让出”事件轮询来让其他等待的事件发生。

这是一个非常简单的方法：

```
var res = [];

// `response(..)`从Ajax调用收到一个结果数组
function response(data) {
    // 我们一次只处理1000件
    var chunk = data.splice( 0, 1000 );

    // 连接到既存的`res`数组上
    res = res.concat(
        // 制造一个新的变形过的数组，所有的`data`值都翻倍
        chunk.map( function(val){
            return val * 2;
        } )
    );

    // 还有东西要处理吗？
    if (data.length > 0) {
        // 异步规划下一个批处理
        setTimeout( function(){
            response( data );
        }, 0 );
    }
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

我们以每次最大1000件作为一个块儿处理数据。这样，我们保证每个“进程”都是短时间运行的，即便这意味着会有许多后续的“进程”，在事件轮询队列上的穿插将会给我们一个响应性（性能）强得多的网站/应用程序。

当然，我们没有对任何这些“进程”的顺序进行互动协调，所以在 `res` 中的结果的顺序是不可预知的。如果要求顺序，你需要使用我们之前讨论的互动技术，或者在本书后续章节中介绍的其他技术。

我们使用 `setTimeout(..0)` （黑科技）来异步排程，基本上它的意思是“将这个函数贴在事件轮询队列的末尾”。

注意： 从技术上讲， `setTimeout(..0)` 没有直接将一条记录插入事件轮询队列。计时器将会在下一个运行机会将事件插入。比如，两个连续的 `setTimeout(..0)` 调用不会严格保证以调用的顺序被处理，所以我们可能看到各种时间偏移的情况，使这样的事件的顺序是不可预知的。在Node.js中，一个相似的方式是 `process.nextTick(..)` 。不管那将会有多方便（而且通常性能更好），（还）没有一个直接的方法可以横跨所有环境来保证异步事件顺序。我们会在下一节详细讨论这个话题。

Jobs

Jobs

在ES6中，在事件轮询队列之上引入了一层新概念，称为“工作队列（Job queue）”。你最有可能接触它的地方是在Promises（见第三章）的异步行为中。

不幸的是，它目前是一个没有公开API的机制，因此要演示它有些兜圈子。我们不得不仅仅在概念上描述它，这样当我们在第三章中讨论异步行为时，你将会理解那些动作行为是如何排程与处理的。

那么，我能找到的考虑它的最佳方式是：“工作队列”是一个挂靠在事件轮询队列的每个tick末尾的队列。在事件轮询的一个tick期间内，某些可能发生的隐含异步动作的行为将不会导致一个全新的事件加入事件轮询队列，而是在当前tick的工作队列的末尾加入一个新的记录（也就是一个Job）。

它好像是在说，“哦，另一件需要我 稍后 去做的事儿，但是保证它在其他任何事情发生之间发生。”

或者，用一个比喻：事件轮询队列就像一个游乐园项目，一旦你乘坐完一次，你就不得不去队尾排队来乘坐下一次。而工作队列就像乘坐完后，立即插队乘坐下一次。

一个Job还可能会导致更多的Job被加入同一个队列的末尾。所以，一个在理论上可能的情况是，Job“轮询”（一个Job持续不断地加入其他Job等）会无限地转下去，从而拖住程序不能移动到一下一个事件轮询tick。这与在你的代码中表达一个长时间运行或无限循环（比如 `while (true) ..` ）在概念上几乎是一样的。

Job的精神有点儿像 `setTimeout(..0)` 黑科技，但以一种定义明确得多的方式实现，而且保证顺序：稍后，但尽快。

让我们想象一个用于Job排程的API，并叫它 `schedule(..)` 。考虑如下代码：

```
1. console.log( "A" );
2.
3. setTimeout( function(){
4.     console.log( "B" );
5. }, 0 );
6.
7. // 理论上的 "Job API"
8. schedule( function(){
9.     console.log( "C" );
10.
11.     schedule( function(){
12.         console.log( "D" );
13.     } );
14. } );
```

你肯能会期望它打印出 `A B C D` ，但是它将会打出 `A C D B` ，因为Job发生在当前的事件轮询tick的末尾，而定时器会在 下一个 事件轮询tick（如果可用的话！）触发排程。

在第三章中，我们会看到Promises的异步行为是基于Job的，所以搞明白它与事件轮询行为的联系是很重要的。

语句排序

语句排序

我们在代码中表达语句的顺序没有必要与JS引擎执行它们的顺序相同。这可能看起来像是个奇怪的论断，所以我们简单地探索一下。

但在我们开始之前，我们应当对一些事情十分清楚：从程序的角度看，语言的规则/文法（参见本丛书的 类型与文法）为语句的顺序决定了一个非常可预知、可靠的行为。所以我们将要讨论的是在你的JS程序中 应当永远观察不到的东西。

警告： 如果你曾经 观察到 过我们将要描述的编译器语句重排，那明显是违反了语言规范，而且无疑是那个JS引擎的Bug—它应当被报告并且修复！但是更常见的是你 怀疑 JS引擎里发生了什么疯狂的事，而事实上它只是你自己代码中的一个Bug（可能是一个“竞合状态”）—所以先检查那里，多检查几遍。在JS调试器使用断点并一行一行地步过你的代码，将是帮你在 你的代码 中找出这样的Bug的最强大的工具。

考虑下面的代码：

```
var a, b;

a = 10;
b = 30;

a = a + 1;
b = b + 1;

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

这段代码没有任何异步表达（除了早先讨论的罕见的 `console` 异步I/O），所以最有可能的推测是它会一行一行地、从上到下地处理。

但是，JS引擎 有可能，在编译完这段代码后（是的，JS是被编译的—见本丛书的 作用域与闭包）发现有机会通过（安全地）重新安排这些语句的顺序来使你的代码运行得更快。实质上，只要你观察不到重排，一切都是合理的。

举个例子，引擎可能会发现如果实际上这样执行代码会更快：

```
var a, b;

a = 10;
a++;
```

```
5.
6. b = 30;
7. b++;
8.
9. console.log( a + b ); // 42
```

或者是这样：

```
1. var a, b;
2.
3. a = 11;
4. b = 31;
5.
6. console.log( a + b ); // 42
```

或者甚至是：

```
1. // 因为`a`和`b`都不再被使用，我们可以内联而且根本不需要它们！
2. console.log( 42 ); // 42
```

在所有这些情况下，JS引擎在它的编译期间进行着安全的优化，而最终的 可观察到 的结果将是相同的。

但也有一个场景，这些特殊的优化是不安全的，因而也是不被允许的（当然，不是说它一点儿都没优化）：

```
1. var a, b;
2.
3. a = 10;
4. b = 30;
5.
6. // 我们需要`a`和`b`递增之前的状态！
7. console.log( a * b ); // 300
8.
9. a = a + 1;
10. b = b + 1;
11.
12. console.log( a + b ); // 42
```

编译器重排会造成可观测的副作用（因此绝不会被允许）的其他例子，包括任何带有副作用的函数调用（特别是getter函数），或者ES6的Proxy对象（参见本丛书的 *ES6与未来*）。

考虑如下代码：

```
function foo() {
    console.log( b );
    return 1;
}

var a, b, c;

// ES5.1 getter 字面语法
c = {
    get bar() {
        console.log( a );
        return 1;
    }
};

a = 10;
b = 30;

a += foo();               // 30
b += c.bar;               // 11

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

如果不是为了这个代码段中的 `console.log(..)` 语句（只是作为这个例子中观察副作用的方便形式），JS引擎将会更加自由，如果它想（谁知道它想不想！？），它会重排这段代码：

```
// ...

a = 10 + foo();
b = 30 + c.bar;

// ...
```

多亏JS语义，我们不会观测到看起来很危险的编译器语句重排，但是理解源代码被编写的方式（从上到下）与它在编译后运行的方式之间的联系是多么微弱，依然是很重要的。

编译器语句重排几乎是并发与互动的微型比喻。作为一个一般概念，这样的意识可以帮你更好地理解异步JS代码流问题。

复习

复习

一个JavaScript程序总是被打断为两个或更多的代码块儿，第一个代码块儿 现在 运行，下一个代码块儿 稍后 运行，来响应一个事件。虽然程序是一块儿一块儿地被执行的，但它们都共享相同的程序作用域和状态，所以对状态的每次修改都是在前一个状态之上的。

不论何时有事件要运行，事件轮询 将运行至队列为空。事件轮询的每次迭代称为一个“tick”。用户交互，IO，和定时器会将事件在事件队列中排队。

在任意给定的时刻，一次只有一个队列中的事件可以被处理。当事件执行时，他可以直接或间接地导致一个或更多的后续事件。

并发是当两个或多个事件链条随着事件相互穿插，因此从高层的角度来看，它们在 同时 运行（即便在给定的某一时刻只有一个事件在被处理）。

在这些并发“进程”之间进行某种形式的互动协调通常是有必要的，比如保证顺序或防止“竞合状态”。这些“进程”还可以 协作：通过将它们自己打断为小的代码块儿来允许其他“进程”穿插。

第二章: 回调

第二章： 回调

在第一章中，我们探讨了JavaScript中关于异步编程的术语和概念。我们的焦点是理解驱动所有“事件”（异步函数调用）的单线程（一次一个）事件轮询队列。我们还探讨了各种解释 同时 运行的事件链，或“进程”（任务， 函数调用等）间的关系的并发模式。

我们在第一章的所有例子中，将函数作为独立的，不可分割的操作单位使用，在这些函数内部语句按照可预知的顺序运行（在编译器水平之上！），但是在函数顺序水平上，事件（也就是异步函数调用）可以以各种顺序发生。

在所有这些情况中，函数都是一个“回调”。因为无论什么时候事件轮询队列中的事件被处理时，这个函数都作为事件轮询“调用并返回”程序的目标。

正如你观察到的，在JS程序中，回调是到目前为止最常见的表达和管理异步的方式。确实，在JavaScript语言中回调是最基础的异步模式。

无数的JS程序，即便是最精巧最复杂的程序，都曾经除了回调外不依靠任何其他异步模式而编写（当然，和我们在第一章中探讨的并发互动模式一起）。回调函数是JavaScript的异步苦工，而且它工作得相当好。

除了……回调并不是没有缺点。许多开发者都对 *Promises* 提供的更好的异步模式感到兴奋不已。但是如果你不明白它在抽象什么，和为什么抽象，是不可能有效利用任何抽象机制的。

在本章中，我们将深入探讨这些话题，来说明为什么更精巧的异步模式（在本书的后续章节中探讨）是必要和被期望的。

链接

- 延续
- 顺序的大脑
- 信任问题
- 尝试拯救回调
- 复习

延续

延续

让我们回到在第一章中开始的异步回调的例子，但让我稍微修改它一下来画出重点：

```
// A
ajax( "..", function(..){
    // C
} );
// B
```

`// A` 和 `// B` 代表程序的前半部分（也就是 现在），`// C` 标识了程序的后半部分（也就是 稍后）。前半部分立即执行，然后会出现一个不知多久的“暂停”。在未来某个时刻，如果Ajax调用完成了，那么程序会回到它刚才离开的地方，并 继续 执行后半部分。

换句话说，回调函数包装或封装了程序的 延续。

让我们把代码弄得更简单一些：

```
// A
setTimeout( function(){
    // C
}, 1000 );
// B
```

稍停片刻然后问你自己，你将如何描述（给一个不那么懂JS工作方式的人）这个程序的行为。来吧，大声说出来。这个很好的练习将使我的下一个观点更鲜明。

现在大多数读者可能在想或说着这样的话：“做A，然后设置一个等待1000毫秒的定时器，一旦它触发，就做C”。与你的版本有多接近？

你可能已经发觉了不对劲儿的地方，给了自己一个修正版：“做A，设置一个1000毫秒的定时器，然后做B，然后在超时事件触发后，做C”。这比第一个版本更准确。你能发现不同之处吗？

虽然第二个版本更准确，但是对于以一种将我们的大脑匹配代码，代码匹配JS引擎的方式讲解这段代码来说，这两个版本都是不足的。这里的鸿沟既是微小的也是巨大的，而且是理解回调作为异步表达和管理的缺点的关键。

只要我们以回调函数的方式引入一个延续（或者像许多程序员那样引入几十个！），我们就允许了一个分歧在我们的大脑如何工作和代码将运行的方式之间形成。当这两者背离时，我们的代码就不可避免地陷入这样的境地：更难理解，更难推理，更难调试，和更难维护。

顺序的大脑

顺序的大脑

我相信大多数读者都曾经听某个人说过（甚至你自己就曾这么说），“我能一心多用”。试图表现得一心多用的效果包含幽默（孩子们的拍头揉肚子游戏），平常的行为（边走边嚼口香糖），和彻头彻尾的危险（开车时发微信）。

但我们是一心多用的人吗？我们真的能执行两个意识，有意地一起行动并在完全同一时刻思考/推理它们两个吗？我们最高级的大脑功能有并行的多线程功能吗？

答案可能令你吃惊：可能不是这样。

我们的大脑其实就不是这样构成的。我们中大多数人（特别是A型人格！）都是自己不情愿承认的一个一心一用者。其实我们只能在任一给定的时刻考虑一件事情。

我不是说我们所有的下意识，潜意识，大脑的自动功能，比如心跳，呼吸，和眨眼。那些都是我们延续生命的重要任务，我们不会有意识地给它们分配大脑的能量。谢天谢地，当我们在3分钟内第15次刷朋友圈时，我们的大脑在后台（线程！）继续着这些重要任务。

相反我们讨论的是在某时刻我们的意识最前线的任务。对我来说，是现在正在写这本书。我还在这完全同一个时刻做其他高级的大脑活动吗？不，没有。我很快而且容易分心—在这最后的几段中有几十次了！

当我们 模拟 一心多用时，比如试着在打字的同时和朋友或家人通电话，实际上我们表现得更像一个快速环境切换器。换句话说，我们快速交替地在两个或更多任务间来回切换，在微小，快速的区块中同时 处理每个任务。我们做的是如此之快，以至于从外界看开我们在 平行地 做这些事情。

难道这听起来不像异步事件并发吗（就像JS中发生的那样）？！如果不，回去再读一遍第一章！

事实上，将庞大复杂的神经内科世界简化为我希望可以在这里讨论的东西的一个方法是，我们的大脑工作起来有点儿像事件轮询队列。

如果你把我打得每一个字（或词）当做一个单独的异步事件，那么现在这一句话上就有十几处地方，可以让我的大脑被其他的事件打断，比如我的感觉，甚至只是我随机的想法。

我不会在每个可能的地方被打断并被拉到其他的“处理”上去（谢天谢地—要不这本书永远也写不完了！）。但是它发生得也足够频繁，以至于我感到我的大脑几乎持续不断地切换到各种不同的环境（也就是“进程”）。而且这和JS引擎可能会感觉到的十分相像。

执行与计划

好了，这么说来我们的大脑可以被认为是运行在一个单线程事件轮询队列中，就像JS引擎那样。这听起来是个不错的匹配。

但是我们需要比我们刚才分析的更加细致入微。在我们如何计划各种任务，和我们的大脑实际如何运行这些任务之间，有一个巨大，明显的不同。

再一次，回到这篇文章的写作的比拟上来。在我心里的粗略计划轮廓是继续写啊写，顺序地经过一系列在我思想中定好的点。我没有在这次写作期间计划任何的打扰或非线性的活动。但无论如何，我的大脑依然一直不停地切换。

即便在操作级别上我们的大脑是异步事件的，但我们还是用一种顺序的，同步的方式计划任务。“我得去商店，然后买些牛奶，然后去干洗店”。

你会注意到这种高级思维（规划）方式看起来不是那么“异步”。事实上，我们几乎很少会故意只用事件的形式思考。相反，我们小心，顺序地（A然后B然后C）计划，而且我们假设一个区间有某种临时的阻塞迫使B等待A，使C等待B。

当开发者编写代码时，他们规划一组将要发生的动作。如果他们是合格的开发者，他们会 小心地规划。比如“我需要将 `z` 的值设为 `x` 的值，然后将 `x` 的值设为 `y` 的值”。

当我们编写同步代码时，一个语句接一个语句，它工作起来就像我们的跑腿todo清单：

```
1. // 交换`x`与`y`(通过临时变量`z`)
2. z = x;
3. x = y;
4. y = z;
```

这三个赋值语句是同步的，所以 `x=y` 会等待 `z=x` 完成，而 `y=z` 会相应地等待 `x=y` 完成。另一种说法是这三个语句临时地按照特定的顺序绑在一起执行，一个接一个。幸好我们不必在这里关心任何异步事件的细节。如果我们关心，代码很快就会变得非常复杂！

如果同步的大脑规划和同步的代码语句匹配的很好，那么我们的大脑能把异步代码规划得多好呢？

事实证明，我们在代码中表达异步的方式（用回调）和我们同步的大脑规划行为根本匹配的不是很好。

你能实际想象一下像这样规划你的跑腿todo清单的思维线索吗？

> “我得去趟商店，但是我确信在路上我会接到一个电话，于是‘嗨，妈妈’，然后她开始讲话，我会在*GPS*上搜索商店的位置，但那会花几分钟加载，所以我把收音机音量调小以便听到妈妈讲话，然后我发现我忘了穿夹克而且外面很冷，但没关系，继续开车并和妈妈说话，然后安全带警报提醒我要系好，于是‘是的，妈，我系着安全带呢，我总是系着安全带！’。啊，*GPS*终于得到方向了，现在……”

虽然作为我们如何度过自己的一天，思考以什么顺序做什么事的规划听起来很荒唐，但这正是我们大脑在功能层面运行的方式。记住，这不是一心多用，而只是快速的环境切换。

我们这些开发者编写异步事件代码困难的原因，特别是当我们只有回调手段可用时，就是意识思考/规划的流动对我们大多数人是不自然的。

我们用一步接一步的方式思考，但是一旦我们从同步走向异步，在代码中可以用的工具（回调）不是以一步接一步的方式表达的。

而且这就是为什么正确编写和推理使用回调的异步JS代码是如此困难：因为它不是我们的大脑进行规划的工作方式。

注意： 唯一比不知道为什么代码不好用更糟糕的是，从一开始就不知道为什么代码好用！这是一种经典的“纸牌屋”心理：“它好用，但不知为什，所以大家都别碰！”你可能听说过，“他人即地狱”（萨特），而程序员们模仿这种说法，“他人的代码即地狱”。我相信：“不明白我自己的代码才是地狱。”而回调正是肇事者之一。

嵌套/链接的回调

考虑下面的代码：

```
listen( "click", function handler(evt){
    setTimeout( function request(){
        ajax( "http://some.url.1", function response(text){
            if (text == "hello") {
                handler();
            }
            else if (text == "world") {
                request();
            }
        } );
    }, 500) ;
} );
```

你很可能一眼就能认出这样的代码。我们得到了三个嵌套在一起的函数链，每一个函数都代表异步序列（任务，“进程”）的一个步骤。

这样的代码常被称为“回调地狱（callback hell）”，有时也被称为“末日金字塔（pyramid of doom）”（由于嵌套的缩进使它看起来像一个放倒的三角形）。

但是“回调地狱”实际上与嵌套/缩进几乎无关。它是一个深刻得多的问题。我们将继续在本章剩下的部分看到它为什么和如何成为一个问题。

首先，我们等待“click”事件，然后我们等待定时器触发，然后我们等待Ajax应答回来，就在这时它可能会将所有这些再做一遍。

猛地一看，这段代码的异步性质可能看起来与顺序的大脑规划相匹配。

首先（现在），我们：

```
listen( "..", function handler(..){
    // ..
} );
```

稍后，我们：

```
setTimeout( function request(..){
    // ..
}, 500) ;
```

再 稍后，我们：

```
ajax( "..", function response(..){
    // ..
} );
```

最后（最 稍后），我们：

```
if ( .. ) {
    // ..
}
else ..
```

不过用这样的方式线性推导这段代码有几个问题。

首先，这个例子中我们的步骤在一条顺序的线上（1，2，3，和4……）是一个巧合。在真实的异步JS程序中，经常会有很多噪音把事情搞乱，在我们从一个函数跳到下一个函数时不得不在大脑中把这些噪音快速地演练一遍。理解这样满载回调的异步流程不是不可能，但绝不自然或容易，即使是经历了很多练习后。

而且，有些更深层的，只是在这段代码中不明显的东西搞错了。让我们建立另一个场景（假想代码）来展示它：

```
doA( function(){
    doB();

    doC( function(){
        doD();
    } )

    doE();
} );
```

```
10.
11. doF();
```

虽然根据经验你将正确地指出这些操作的真实顺序，但我打赌它第一眼看上去有些使人糊涂，而且需要一些协调的思维周期才能搞明白。这些操作将会以这种顺序发生：

- `doA()`
- `doF()`
- `doB()`
- `doC()`
- `doE()`
- `doD()`

你是在第一次浏览这段代码就看明白的吗？

好吧，你们肯定有些人在想我在函数的命名上不公平，故意引导你误入歧途。我发誓我只是按照从上到下出现的顺序命名的。不过让我再试一次：

```
1. doA( function(){
2.     doC();
3.
4.     doD( function(){
5.         doF();
6.     } )
7.
8.     doE();
9. } );
10.
11. doB();
```

现在，我以他们实际执行的顺序用字母命名了。但我依然要打赌，即便是现在对这个场景有经验的情况下，大多数读者追踪 `A -> B -> C -> D -> E -> F` 的顺序并不是自然而然的。你的眼睛肯定在这段代码中上上下下跳了许多次，对吧？

就算它对你来说都是自然的，这里依然还有一个可能肆虐的灾难。你能发现它是什么吗？

如果 `doA(..)` 或 `doD(..)` 实际上不是如我们明显地假设的那样，不是异步的呢？嗯，现在顺序不同了。如果它们都是同步的（也许仅仅有时是这样，根据当时程序所处的条件而定），现在的顺序是 `A -> C -> D -> F -> E -> B` 。

你在背景中隐约听到的声音，正是成千上万双手掩面的JS开发者的叹息。

嵌套是问题吗？是它使追踪异步流程变得这么困难吗？当然，有一部分是。

但是让我不用嵌套重写一遍前面事件/超时/Ajax嵌套的例子：

```
listen( "click", handler );

function handler() {
    setTimeout( request, 500 );
}

function request(){
    ajax( "http://some.url.1", response );
}

function response(text){
    if (text == "hello") {
        handler();
    }
    else if (text == "world") {
        request();
    }
}
```

这样的代码组织形式几乎看不出来有前一种形式的嵌套/缩进困境，但它的每一处依然容易受到“回调地狱”的影响。为什么呢？

当我们线性地（顺序地）推理这段代码，我们不得不从一个函数跳到下一个函数，再跳到下一个函数，并在代码中弹来弹去以“看到”顺序流。并且要记住，这个简化的代码风格是某种最佳情况。我们都知道真实的JS程序代码经常更加神奇地错综复杂，使这样量级的顺序推理更加困难。

另一件需要注意的事是：为了将第2，3，4步链接在一起使他们相继发生，回调独自给我们的启示是将第2步硬编码在第1步中，将第3步硬编码在第2步中，将第4步硬编码在第3步中，如此继续。硬编码不一定是一件坏事，如果第2步应当总是在第3步之前真的是一个固定条件。

不过硬编码绝对会使代码变得更脆弱，因为它不考虑任何可能使在步骤前行的过程中出现偏差的异常情况。举个例子，如果第2步失败了，第3步永远不会到达，第2步也不会重试，或者移动到一个错误处理流程上，等等。

所有这些问题你都 可以 手动硬编码在每一步中，但那样的代码总是重复性的，而且不能在其他步骤或你程序的其他异步流程中复用。

即便我们的大脑可能以顺序的方式规划一系列任务（这个，然后这个，然后这个），但我们大脑运行的事件的性质，使恢复/重试/分流这样的流程控制几乎毫不费力。如果你出去购物，而且你发现你把购物单忘在家里了，这并不会因为你没有提前计划这种情况而结束这一天。你的大脑会很容易地绕过这个小问题：你回家，取购物单，然后回头去商店。

但是手动硬编码的回调（甚至带有硬编码的错误处理）的脆弱本性通常不那么优雅。一旦你最终指明了（也就是提前规划好了）所有各种可能性/路径，代码就会变得如此复杂以至于几乎不能维护或更新。

这 才是“回调地狱”想表达的！嵌套/缩进基本上一个余兴表演，转移注意力的东西。

如果以上这些还不够，我们还没有触及两个或更多这些回调延续的链条 同时 发生会怎么样，或者当第三步分叉成为带有大门或门闩的“并行”回调，或者……我的天哪，我脑子疼，你呢？

你抓住这里的重点了吗？我们顺序的，阻塞的大脑规划行为和面向回调的异步代码不能很好地匹配。这就是需要清楚地阐明的关于回调的首要缺陷：它们在代码中表达异步的方式，是需要我们的大脑不得不斗争才能保持一致的。

信任问题

信任问题

在顺序的大脑规划和JS代码中回调驱动的异步处理间的不匹配只是关于回调的问题的一部分。还有一些更深刻的问题值得担忧。

让我们再一次重温这个概念—回调函数是我们程序的延续（也就是程序的第二部分）：

```
1. // A
2. ajax( "..", function(..){
3.     // C
4. } );
5. // B
```

`// A` 和 `// B` 现在 发生，在JS主程序的直接控制之下。但是 `// C` 被推迟到 稍后 再发生，并且在另一部分的控制之下—这里是 `ajax(..)` 函数。在基本的感觉上，这样的控制交接一般不会让程序产生很多问题。

但是不要被这种控制切换不是什么大事的罕见情况欺骗了。事实上，它是回调驱动的设计的最可怕的（也是最微妙的）问题。这个问题围绕着一个想法展开：有时 `ajax(..)` （或者说你向之提交回调的部分）不是你写的函数，或者不是你可以直接控制的函数。很多时候它是一个由第三方提供的工具。

当你把你程序的一部分拿出来并把它执行的控制权移交给另一个第三方时，我们称这种情况为“控制倒转”。在你的代码和第三方工具之间有一个没有明言的“契约”—一组你期望被维护的东西。

五个回调的故事

为什么这件事情很重要可能不是那么明显。让我们来构建一个夸张的场景来生动地描绘一下信任危机。

想象你是一个开发者，正在建造一个贩卖昂贵电视的网站的结算系统。你已经将结算系统的各种页面顺利地制造完成。在最后一个页面，当用户点解“确定”购买电视时，你需要调用一个第三方函数（假如由一个跟踪分析公司提供），以便使这笔交易能够被追踪。

你注意到它们提供的是某种异步追踪工具，也许是为了最佳的性能，这意味着你需要传递一个回调函数。在你传入的这个程序的延续中，有你最后的代码—划客人的信用卡并显示一个感谢页面。

这段代码可能看起来像这样：

```
analytics.trackPurchase( purchaseData, function(){
    chargeCreditCard();
    displayThankyouPage();
} );
```

足够简单，对吧？你写好代码，测试它，一切正常，然后你把它部署到生产环境。大家都很开心！

6个月过去了，没有任何问题。你几乎已经忘了你曾写过的代码。一天早上，工作之前你先在咖啡店坐坐，悠闲地享用着你的拿铁，直到你接到老板慌张的电话要求你立即扔掉咖啡并冲进办公室。

当你到达时，你发现一位高端客户为了买同一台电视信用卡被划了5次，而且可以理解，他不高兴。客服已经道了歉并开始办理退款。但你的老板要求知道这是怎么发生的。“我们没有测试过这样的情况吗！？”

你甚至不记得你写过的代码了。但你还是往回挖掘试着找出是什么出错了。

在分析过一些日志之后，你得出的结论是，唯一的解释是分析工具不知怎么的，由于某些原因，将你的回调函数调用了5次而非一次。他们的文档中没有任何东西提到此事。

十分令人沮丧，你联系了客户支持，当然他们和你一样惊讶。他们同意将此事向上提交至开发者，并许诺给你回复。第二天，你收到一封很长的邮件解释他们发现了什么，然后你将它转发给了你的老板。

看起来，分析公司的开发者曾经制作了一些实验性的代码，在一定条件下，将会每秒重试一次收到的回调，在超时之前共计5秒。他们从没想要把这部分推到生产环境，但不知怎地他们这样做了，而且他们感到十分难堪而且抱歉。然后是许多他们如何定位错误的细节，和他们将要如何做以保证此事不再发生。等等，等等。

后来呢？

你找你的老板谈了此事，但是他对事情的状态不是感觉特别舒服。他坚持，而且你也勉强地同意，你不能再相信 他们 了（咬到你的东西），而你将需要指出如何保护放出的代码，使它们不再受这样的漏洞威胁。

修修补补之后，你实现了一些如下的特殊逻辑代码，团队中的每个人看起来都挺喜欢：

```
var tracked = false;

analytics.trackPurchase( purchaseData, function(){
    if (!tracked) {
        tracked = true;
        chargeCreditCard();
        displayThankyouPage();
    }
```

```
} );
```

注意： 对读过第一章的你来说这应当很熟悉，因为我们实质上创建了一个门闩来处理我们的回调被并发调用多次的情况。

但一个QA的工程师问，“如果他们没调你的回调怎么办？” 噢。谁也没想过。

你开始布下天罗地网，考虑在他们调用你的回调时所有出错的可能性。这里是你得到的分析工具可能不正常运行的方式的大致列表：

- 调用回调过早（在它开始追踪之前）
- 调用回调过晚 （或不调）
- 调用回调太少或太多次（就像你遇到的问题！）
- 没能向你的回调传递必要的环境/参数
- 吞掉了可能发生的错误/异常
- …

这感觉像是一个麻烦清单，因为它就是。你可能慢慢开始理解，你将要不得不为 每一个传递到你不能信任的工具中的回调 都创造一大堆的特殊逻辑。

现在你更全面地理解了“回调地狱”有多地狱。

不仅是其他人的代码

现在有些人可能会怀疑事情到底是不是如我所宣扬的这么大条。也许你根本就不和真正的第三方工具互动。也许你用的是进行了版本控制的API，或者自己保管的库，因此它的行为不会在你不知晓的情况下改变。

那么，好好思考这个问题：你能 真正 信任你理论上控制（在你的代码库中）的工具吗？

这样考虑：我们大多数人都同意，至少在某个区间内我们应当带着一些防御性的输入参数检查制造我们自己的内部函数，来减少/防止以外的问题。

过于相信输入：

```
function addNumbers(x,y) {
    // + 操作符使用强制转换重载为字符串连接
    // 所以根据传入参数的不同，这个操作不是严格的安全。
    return x + y;
}

addNumbers( 21, 21 );    // 42
addNumbers( 21, "21" );    // "2121"
```

防御不信任的输入：

```
function addNumbers(x,y) {
    // 保证数字输入
    if (typeof x != "number" || typeof y != "number") {
        throw Error( "Bad parameters" );
    }

    // 如果我们到达这里，+ 就可以安全地做数字加法
    return x + y;
}

addNumbers( 21, 21 );    // 42
addNumbers( 21, "21" );    // Error: "Bad parameters"
```

或者也许依然安全但更友好：

```
function addNumbers(x,y) {
    // 保证数字输入
    x = Number( x );
    y = Number( y );

    // + 将会安全地执行数字加法
    return x + y;
}

addNumbers( 21, 21 );    // 42
addNumbers( 21, "21" );    // 42
```

不管你怎么做，这类函数参数的检查/规范化是相当常见的，即便是我们理论上完全信任的代码。用一个粗俗的说法，编程好像是地缘政治学的“信任但验证”原则的等价物。

那么，这不是要推论出我们应当对异步函数回调的编写做相同的事，而且不仅是针对真正的外部代码，甚至要对一般认为是“在我们控制之下”的代码？我们当然应该。

但是回调没有给我们提供任何协助。我们不得不自己构建所有的装置，而且这通常最终成为许多我们要在每个异步回调中重复的模板/负担。

有关于回调的最麻烦的问题就是 控制反转 导致所有这些信任完全崩溃。

如果你有代码用到回调，特别是但不特指第三方工具，而且你还没有为所有这些 控制反转 的信任问题实施某些缓和逻辑，那么你的代码现在就 有 bug，虽然它们还没咬到你。将来的bug依然是bug。

确实是地狱。

尝试拯救回调

尝试拯救回调

有几种回调的设计试图解决一些（不是全部！）我们刚才看到的信任问题。这是一种将回调模式从它自己的崩溃中拯救出来的勇敢，但注定失败的努力。

举个例子，为了更平静地处理错误，有些API设计提供了分离的回调（一个用作成功的通知，一个用作错误的通知）：

```
function success(data) {
    console.log( data );
}

function failure(err) {
    console.error( err );
}

ajax( "http://some.url.1", success, failure );
```

在这种设计的API中，`failure()`错误处理器通常是可选的，而且如果不提供的话它会假定你想让错误被吞掉。呃。

注意： ES6的Promises的API使用的就是这种分离回调设计。我们将在下一章中详尽地讨论ES6的Promises。

另一种常见的回调设计模式称为“错误优先风格”（有时称为“Node风格”，因为它几乎在所有的Node.js的API中作为惯例使用），一个回调的第一个参数为一个错误对象保留（如果有的话）。如果成功，这个参数将会是空/falsy（而其他后续的参数将是成功的数据），但如果出现了错误的结果，这第一个参数就会被设置/truthy（而且通常没有其他东西会被传递了）：

```
function response(err,data) {
    // 有错？
    if (err) {
        console.error( err );
    }
    // 否则，认为成功
    else {
        console.log( data );
    }
}
```

```
11.
12. ajax( "http://some.url.1", response );
```

这两种方法都有几件事情应当注意。

首先，它们没有像看起来那样真正解决主要的信任问题。在这两个回调中没有关于防止或过滤意外的重复调用的东西。而且，事情现在更糟糕了，因为你可能同时得到成功和失败信号，或者都得不到，你仍然不得不围绕着这两种情况写代码。

还有，不要忘了这样的事实：虽然它们是你可以引用的标准模式，但它们绝对更加繁冗，而且是不太可能复用的模板代码，所以你将会对在你应用程序的每一个回调中敲出它们感到厌倦。

回调从不被调用的信任问题怎么解决？如果这要紧（而且它可能应当要紧！），你可能需要设置一个超时来取消事件。你可以制作一个工具来帮你：

```
1. function timeoutify(fn,delay) {
2.     var intv = setTimeout( function(){
3.             intv = null;
4.             fn( new Error( "Timeout!" ) );
5.         }, delay )
6.     ;
7.
8.     return function() {
9.         // 超时还没有发生？
10.        if (intv) {
11.            clearTimeout( intv );
12.            fn.apply( this, [ null ].concat( [].slice.call( arguments ) ) );
13.        }
14.    };
15. }
```

这是你如何使用它：

```
1. // 使用“错误优先”风格的回调设计
2. function foo(err,data) {
3.     if (err) {
4.         console.error( err );
5.     }
6.     else {
7.         console.log( data );
8.     }
9. }
10.
11. ajax( "http://some.url.1", timeoutify( foo, 500 ) );
```

另一个信任问题是被调用的“过早”。在应用程序规范上讲，这可能涉及在某些重要的任务完成之前被调用。但更一般地，在那些即可以 现在（同步地），也可以在 稍后（异步地）调用你提供的回调的工具中这个问题更明显。

这种围绕着同步或异步行为的不确定性，几乎总是导致非常难追踪的Bug。在某些圈子中，一个名叫Zalgo的可以导致人精神错乱的虚构怪物被用来描述这种同步/异步的噩梦。经常能听到人们喊“别放出Zalgo！”，而且它引出了一个非常响亮的建议：总是异步地调用回调，即便它是“立即”在事件轮询的下一个迭代中，这样所有的回调都是可预见的异步。

注意： 更多关于Zalgo的信息，参见Oren Golan的“Don’t Release Zalgo!（不要释放Zalgo！）”(https://github.com/oren/oren.github.io/blob/master/posts/zalgo.md)和Isaac Z. Schlueter的“Designing APIs for Asynchrony（异步API设计）”(http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony)。

考虑下面的代码：

```
function result(data) {
    console.log( a );
}

var a = 0;

ajax( "..pre-cached-url..", result );
a++;
```

这段代码是打印 `0`（同步回调调用）还是打印 `1`（异步回调调用）？这……要看情况。

你可以看到Zalgo的不可预见性能有多快地威胁你的JS程序。所以听起来傻呼呼的“别放出Zalgo”实际上是一个不可思议地常见且实在的建议——总是保持异步。

如果你不知道当前的API是否会总是异步地执行呢？你可以制造一个像 `asyncify(..)` 这样的工具：

```
function asyncify(fn) {
    var orig_fn = fn,
        intv = setTimeout( function(){
            intv = null;
            if (fn) fn();
        }, 0 )
    ;

    fn = null;

    return function() {
        // 触发太快，在`intv`计时器触发来
        // 表示异步回合已经过去之前？
        if (intv) {
```

```
            fn = orig_fn.bind.apply(
                orig_fn,
                // 将包装函数的`this`加入`bind(..)`调用的
                // 参数，同时currying其他所有的传入参数
                [this].concat( [].slice.call( arguments ) )
            );
        }
        // 已经是异步
        else {
            // 调用原版的函数
            orig_fn.apply( this, arguments );
        }
    };
}
```

你像这样使用 `asyncify(..)` ：

```
function result(data) {
    console.log( a );
}

var a = 0;

ajax( "..pre-cached-url..", asyncify( result ) );
a++;
```

不管Ajax请求是由于存在于缓存中而解析为立即调用回调，还是它必须走过网线去取得数据而异步地稍后完成，这段代码总是输出 `1` 而不是 `0` —— `result(..)` 总是被异步地调用，这意味着 `a++` 有机会在 `result(..)` 之前运行。

噢耶，又一个信任问题被“解决了”！但它很低效，而且又有更多臃肿的模板代码让你的项目变得沉重。

这只是关于回调一遍又一遍地发生的故事。它们几乎可以做任何你想做的事，但你不得不努力工作来达到目的，而且大多数时候这种努力比你应当在推理这样的代码上所付出的多得多。

你可能发现自己希望有一些内建的API或语言机制来解决这些问题。终于ES6带着一个伟大的答案到来了，所以继续读下去！

复习

复习

回调是JS中异步的基础单位。但是随着JS的成熟，它们对于异步编程的演化趋势来讲显得不够。

首先，我们的大脑用顺序的，阻塞的，单线程的语义方式规划事情，但是回调使用非线性，非顺序的方式表达异步流程，这使我们正确推理这样的代码变得非常困难。不好推理的代码是导致不好的Bug的不好的代码。

我们需要一个种方法，以更同步化，顺序化，阻塞的方式来表达异步，正如我们的大脑那样。

第二，而且是更重要的，回调遭受着 控制反转 的蹂躏，它们隐含地将控制权交给第三方（通常第三方工具不受你控制！）来调用你程序的 延续。这种控制权的转移使我们得到一张信任问题的令人不安的列表，比如回调是否会比我们期望的被调用更多次。

制造特殊的逻辑来解决这些信任问题是可能的，但是它比它应有的难度高多了，还会产生更笨重和更难维护的代码，而且在bug实际咬到你的时候代码会显得在这些危险上被保护的不够。

我们需要一个 所有这些信任问题 的一般化解决方案。一个可以被所有我们制造的回调复用，而且没有多余的模板代码负担的方案。

我们需要比回调更好的东西。目前为止它们做的不错，但JavaScript的 未来 要求更精巧和强大的异步模式。本书的后续章节将会深入这些新兴的发展变化。

第三章: Promise

第三章： Promises

在第二章中，我们定位了在使用回调表达程序异步性和管理并发的两个主要类别的不足：缺乏顺序性和缺乏可靠性。现在我们更亲近地理解了问题，是时候将我们的注意力转向解决它们的模式了。

我们首先想要解决的是 控制倒转 问题，信任是如此脆弱而且是如此的容易丢失。

回想一下，我们将我们的程序的延续包装进一个回调函数中，将这个回调交给另一个团体（甚至是潜在的外部代码），并双手合十祈祷它会做正确的事情并调用这个回调。

我们这么做是因为我们想说，“这是 稍后 将要发生的事，在当前的步骤完成之后。”

但是如果我们能够反向倒转这种 控制倒转 呢？如果不是将我们程序的延续交给另一个团体，而是希望它返回给我们一个可以知道它何时完成的能力，然后我们的代码可以决定下一步做什么呢？

这种规范被称为 **Promise**。

Promise正在像风暴一样席卷JS世界，因为开发者和语言规范作者之流拼命地想要在他们的代码/设计中结束回调地狱的疯狂。事实上，大多数新被加入JS/DOM平台的异步API都是建立在Promise之上的。所以深入学习它们可能是个好主意，你不这么认为吗？

注意： “立即”这个词将在本章频繁使用，一般来说它指代一些Promise解析行为。然而，本质上在所有情况下，“立即”意味着就工作队列行为（参见第一章）而言，不是严格同步的 现在 的感觉。

链接

什么是 Promise？

什么是Promise？

当开发者们决定要学习一种新技术或模式的时候，他们的第一步总是“给我看代码！”。摸着石头过河对我们来讲是十分自然的。

但事实上仅仅考察API丢失了一些抽象过程。Promise是这样一种工具：它能非常明显地看出使用者是否理解了它是为什么和关于什么，还是仅仅学习和使用API。

所以在我展示Promise的代码之前，我想在概念上完整地解释一下Promise到底是什么。我希望这能更好地指引你探索如何将Promise理论整合到你自己的异步流程中。

带着这样的想法，让我们来看两种类比，来解释Promise是什么。

未来的值

想象这样的场景：我走到快餐店的柜台前，点了一个起士汉堡。并交了1.47美元的现金。通过点餐和付款，我为得到一个 值（起士汉堡）制造了一个请求。我发起了一个事务。

但是通常来说，起士汉堡不会立即到我手中。收银员交给一些东西代替我的起士汉堡：一个带有点餐排队号的收据。这个点餐号是一个“我欠你”的许诺（Promise），它保证我最终会得到我的起士汉堡。

于是我就拿着我的收据和点餐号。我知道它代表我的 未来的起士汉堡，所以我无需再担心它—除了挨饿！

在我等待的时候，我可以做其他的事情，比如给我的朋友发微信说，“嘿，一块儿吃午餐吗？我要吃起士汉堡”。

我已经在用我的 未来的起士汉堡 进行推理了，即便它还没有到我手中。我的大脑可以这么做是因为它将点餐号作为起士汉堡的占位符号。这个占位符号实质上使这个值 与时间无关。它是一个 未来的值。

最终，我听到，“113号！”。于是我愉快地拿着收据走回柜台前。我把收据递给收银员，拿回我的起

士汉堡。

换句话说，一旦我的 未来的值 准备好，我就用我的许诺值换回值本身。

但还有另外一种可能的输出。它们叫我的号，但当我去取起士汉堡时，收银员遗憾地告诉我，“对不起，看起来我们的起士汉堡卖光了。”把这种场景下顾客有多沮丧放在一边，我们可以看到 未来的值 的一个重要性质：它们既可以表示成功也可以表示失败。

每次我点起士汉堡时，我都知道我要么最终得到一个起士汉堡，要么得到起士汉堡卖光的坏消息，并且不得不考虑中午吃点儿别的东西。

注意： 在代码中，事情没有这么简单，因为还隐含着一种点餐号永远也不会被叫到的情况，这时我们就被搁置在了一种无限等待的未解析状态。我们待会儿再回头处理这种情况。

现在和稍后的值

这一切也许听起来在思维上太过抽象而不能实施在你的代码中。那么，让我们更具体一些。

然而，在我们能介绍Promise是如何以这种方式工作之前，我们先看看我们已经明白的代码——回调！——是如何处理这些 未来值 的。

在你写代码来推导一个值时，比如在一个 `number` 上进行数学操作，不论你是否理解，对于这个值你已经假设了某些非常基础的事实——这个值已经是一个实在的 现在 值：

```
1. var x, y = 2;
2.
3. console.log( x + y ); // NaN  <-- 因为`x`还没有被赋值
```

`x + y` 操作假定 `x` 和 `y` 都已经被设定好了。用我们一会将要阐述的术语来讲，我们假定 `x` 和 `y` 的值已经被 解析（*resovle*） 了。

期盼 `+` 操作符本身能够魔法般地检测并等待 `x` 和 `y` 的值被解析（也就是准备好），然后仅在那之后才进行操作是没道理的。如果不同的语句 现在 完成而其他的 稍后 完成，这就会在程序中造成混乱，对吧？

如果两个语句中的一个（或两者同时）可能还没有完成，你如何才能推断它们的关系呢？如果语句2要依赖语句1的完成，那么这里仅有两种输出：不是语句1 现在 立即完成而且一切处理正常进行，就是语句1还没有完成，所以语句2将会失败。

如果这些东西听起来很像第一章的内容，很好！

回到我们的 `x + y` 的数学操作。想象有一种方法可以说，“将 `x` 和 `y` 相加，但如果它们中任意一个还没有被设置，就等到它们都被设置。尽快将它们相加。”

你的大脑也许刚刚跳进回调。好吧，那么…

```
function add(getX,getY,cb) {
    var x, y;
    getX( function(xVal){
        x = xVal;
        // 两者都准备好了？
        if (y != undefined) {
            cb( x + y );    // 发送加法的结果
        }
    } );
    getY( function(yVal){
        y = yVal;
        // 两者都准备好了？
        if (x != undefined) {
            cb( x + y );    // 发送加法的结果
        }
    } );
}

// `fetchX()`和`fetchY()`是同步或异步的函数
add( fetchX, fetchY, function(sum){
    console.log( sum ); // 很简单吧？
} );
```

花点儿时间来感受一下这段代码的美妙（或者丑陋），我耐心地等你。

虽然丑陋是无法否认的，但是关于这种异步模式有一些非常重要的事情需要注意。

在这段代码中，我们将 `x` 和 `y` 作为未来的值对待，我们将 `add(..)` 操作表达为：（从外部看来）它并不关心 `x` 或 `y` 或它们两者现在是否可用。换句话所，它泛化了 现在 和 稍后，如此我们可以信赖 `add(..)` 操作的一个可预测的结果。

通过使用一个临时一致的 `add(..)` ——它跨越 现在 和 稍后 的行为是相同的——异步代码的推理变得容易的多了。

更直白地说：为了一致地处理 现在 和 稍后，我们将它们都作为 稍后：所有的操作都变成异步的。

当然，这种粗略的基于回调的方法留下了许多提升的空间。为了理解在不用关心 未来的值 在时间上什么时候变得可用的情况下推理它而带来的好处，这仅仅是迈出的一小步。

Promise值

我们绝对会在本章的后面深入更多关于Promise的细节——所以如果这让你犯糊涂，不要担心——但让我们先简单地看一下我们如何通过 `Promise` 来表达 `x + y` 的例子：

```
function add(xPromise,yPromise) {
    // `Promise.all([ .. ])`接收一个Promise的数组,
```

```
    // 并返回一个等待它们全部完成的新Promise
    return Promise.all( [xPromise, yPromise] )

    // 当这个Promise被解析后，我们拿起收到的`X`和`Y`的值，并把它们相加
    .then( function(values){
        // `values`是一个从先前被解析的Promise那里收到的消息数组
        return values[0] + values[1];
    } );
}

// `fetchX()`和`fetchY()`分别为它们的值返回一个Promise，
// 这些值可能在 *现在* 或 *稍后* 准备好
add( fetchX(), fetchY() )

// 为了将两个数字相加，我们得到一个Promise。
// 现在我们链式地调用`then(..)`来等待返回的Promise被解析
.then( function(sum){
    console.log( sum ); // 这容易多了！
} );
```

在这个代码段中有两层Promise。

`fetchX()` 和 `fetchY()` 被直接调用，它们的返回值（promise！）被传入 `add(..)`。这些promise表示的值将在 现在 或 稍后 准备好，但是每个promise都将行为泛化为与时间无关。我们以一种时间无关的方式来推理 `X` 和 `Y` 的值。它们是 未来值。

第二层是由 `add(..)` 创建（通过 `Promise.all([ .. ])` ）并返回的promise，我们通过调用 `then(..)` 来等待它。当 `add(..)` 操作完成后，我们的 `sum` 未来值 就准备好并可以打印了。我们将等待 `X` 和 `Y` 的 未来值 的逻辑隐藏在 `add(..)` 内部。

注意： 在 `add(..)` 内部。 `Promise.all([ .. ])` 调用创建了一个promise（它在等待 `promiseX` 和 `promiseY` 被解析）。链式调用 `.then(..)` 创建了另一个promise，它的 `return values[0] + values[1]` 这一行会被立即解析（使用加法的结果）。这样，我们链接在 `add(..)` 调用末尾的 `then(..)` 调用——在代码段最后——实际上是在第二个被返回的promise上进行操作，而非被 `Promise.all([ .. ])` 创建的第一个promise。另外，虽然我们没有在这第二个 `then(..)` 的末尾链接任何操作，它也已经创建了另一个promise，我们可以选择监听/使用它。这类Promise链的细节将会在本章后面进行讲解。

就像点一个起士汉堡，Promise的解析可能是一个拒绝（rejection）而非完成（fulfillment）。不同的是，被完成的Promise的值总是程序化的，而一个拒绝值——通常被称为“拒绝理由”——既可以被程序逻辑设置，也可以被运行时异常隐含地设置。

使用Promise， `then(..)` 调用实际上可以接受两个函数，第一个用作完成（正如刚才所示），而第二个用作拒绝：

```
add( fetchX(), fetchY() )
```

```
.then(
    // 完成处理器
    function(sum) {
        console.log( sum );
    },
    // 拒绝处理器
    function(err) {
        console.error( err ); // 倒霉！
    }
);
```

如果在取得 `X` 或 `Y` 时出现了错误，或在加法操作时某些事情不知怎地失败了， `add(..)` 返回的promise就被拒绝了，传入 `then(..)` 的第二个错误处理回调函数会从promise那里收到拒绝的值。

因为Promise包装了时间相关的状态—等待当前值的完成或拒绝—从外部看来，Promise本身是时间无关的，如此Promise就可以用可预测的方式组合，而不用关心时间或底层的结果。

另外，一旦Promise被解析，它就永远保持那个状态—它在那个时刻变成了一个 不可变的值—而且可以根据需要 被监听 任意多次。

注意： 因为Promise一旦被解析就是外部不可变的，所以现在将这个值传递给任何其他团体都是安全的，而且我们知道它不会被意外或恶意地被修改。这在许多团体监听同一个Promise的解析时特别有用。一个团体去影响另一个团体对Promise解析的监听能力是不可能的。不可变性听起来是一个学院派话题，但它实际上是Promise设计中最基础且最重要的方面之一，因此不能将它随意地跳过。

这是用于理解Promise的最强大且最重要的概念之一。通过大量的工作，你可以仅仅使用丑陋的回调组合来创建相同的效果，但这真的不是一个高效的策略，特别是你不得不一遍一遍地重复它。

Promise是一种用来包装与组合 未来值，并且可以很容易复用的机制。

完成事件

正如我们刚才看到的，一个独立的Promise作为一个 未来值 动作。但还有另外一种方式考虑Promise的解析：在一个异步任务的两个或以上步骤中，作为一种流程控制机制—俗称“这个然后那个”。

让我们想象调用 `foo(..)` 来执行某个任务。我们对它的细节一无所知，我们也不关心。它可能会立即完成任务，也可能会花一段时间完成。

我们仅仅想简单地知道 `foo(..)` 什么时候完成，以便于我们可以移动到下一个任务。换句话说，我们想要一种方法被告知 `foo(..)` 的完成，以便于我们可以 继续。

在典型的JavaScript风格中，如果你需要监听一个通知，你很可能会想到事件（event）。那么我们可以将我们的通知需求重新表述为，监听由 `foo(..)` 发出的 完成（或 继续）事件。

注意： 将它称为一个“完成事件”还是一个“继续事件”取决于你的角度。你是更关心 `foo(..)` 发生的事情，还是更关心 `foo(..)` 完成 之后 发生的事情？两种角度都对而且都有用。事件通知告诉我们 `foo(..)` 已经 完成，但是 继续 到下一个步骤也没问题。的确，你为了事件通知调用而传入的回调函数本身，在前面我们称它为一个 延续。因为 完成事件 更加聚焦于 `foo(..)` ，也就是我们当前注意的东西，所以在这篇文章的其余部分我们稍稍偏向于使用 完成事件。

使用回调，“通知”就是被任务（ `foo(..)` ）调用的我们的回调函数。但是使用Promise，我们将关系扭转过来，我们希望能够监听一个来自于 `foo(..)` 的事件，当我们被通知时，做相应的处理。

首先，考虑一些假想代码：

```
foo(x) {
    // 开始做一些可能会花一段时间的事情
}

foo( 42 )

on (foo "completion") {
    // 现在我们可以做下一步了！
}

on (foo "error") {
    // 噢，在`foo(..)`中有某些事情搞错了
}
```

我们调用 `foo(..)` 然后我们设置两个事件监听器，一个给 `"completion"` ，一个给 `"error"` —— `foo(..)` 调用的两种可能的最终结果。实质上， `foo(..)` 甚至不知道调用它的代码监听了这些事件，这构成了一个非常美妙的 关注分离（*separation of concerns*）。

不幸的是，这样的代码将需要JS环境不具备的一些“魔法”（而且显得有些不切实际）。这里是一种用JS表达它的更自然的方式：

```
function foo(x) {
    // 开始做一些可能会花一段时间的事情

    // 制造一个`listener`事件通知能力并返回

    return listener;
}

var evt = foo( 42 );

evt.on( "completion", function(){
    // 现在我们可以做下一步了！
} );

```

```
evt.on( "failure", function(err){
    // 噢，在`foo(..)`中有某些事情搞错了
} );
```

`foo(..)` 明确地创建并返回了一个事件监听能力，调用方代码接收并在它上面注册了两个事件监听器。

很明显这反转了一般的面向回调代码，而且是有意为之。与将回调传入 `foo(..)` 相反，它返回一个我们称之为 `evt` 的事件能力，它接收回调。

但如果你回想第二章，回调本身代表着一种 控制反转。所以反转回调模式实际上是 反转的反转，或者说是一个 控制非反转—将控制权归还给我们希望保持它的调用方代码，

一个重要的好处是，代码的多个分离部分都可以被赋予事件监听能力，而且它们都可在 `foo(..)` 完成时被独立地通知，来执行后续的步骤：

```
var evt = foo( 42 );

// 让`bar(..)`监听`foo(..)`的完成
bar( evt );

// 同时，让`baz(..)`监听`foo(..)`的完成
baz( evt );
```

控制非反转 导致了更好的 关注分离，也就是 `bar(..)` 和 `baz(..)` 不必卷入 `foo(..)` 是如何被调用的问题。相似地， `foo(..)` 也不必知道或关心 `bar(..)` 和 `baz(..)` 的存在或它们是否在等待 `foo(..)` 完成的通知。

实质上，这个 `evt` 对象是一个中立的第三方团体，在分离的关注点之间进行交涉。

Promise“事件”

正如你可能已经猜到的， `evt` 事件监听能力是一个Promise的类比。

在一个基于Promise的方式中，前面的代码段将会使 `foo(..)` 创建并返回一个 `Promise` 实例，而且这个promise将会被传入 `bar(..)` 和 `baz(..)` 。

注意： 我们监听的Promise解析“事件”并不是严格的事件（虽然它们为了某些目的表现得像事件），而且它们也不经常称为 `"completion"` 或 `"error"` 。相反，我们用 `then(..)` 来注册一个 `"then"` 事件。或者也许更准确地讲， `then(..)` 注册了 `"fulfillment（完成）"` 和/或 `"rejection（拒绝）"` 事件，虽然我们在代码中不会看到这些名词被明确地使用。

考虑：

```
function foo(x) {
```

```
    // 开始做一些可能会花一段时间的事情

    // 构建并返回一个promise
    return new Promise( function(resolve,reject){
        // 最终需要调用`resolve(..)`或`reject(..)`
        // 它们是这个promise的解析回调
    } );
}

var p = foo( 42 );

bar( p );

baz( p );
```

注意： 在 `new Promise( function(..){ .. } )` 中展示的模式通常被称为“揭示构造器（revealing constructor）”。被传入的函数被立即执行（不会被异步推迟，像 `then(..)` 的回调那样），而且它被提供了两个参数，我们叫它们 `resolve` 和 `reject` 。这些是Promise的解析函数。 `resolve(..)` 一般表示完成，而 `reject(..)` 表示拒绝。

你可能猜到了 `bar(..)` 和 `baz(..)` 的内部看起来是什么样子：

```
function bar(fooPromise) {
    // 监听`foo(..)`的完成
    fooPromise.then(
        function(){
            // `foo(..)`现在完成了，那么做`bar(..)`的任务
        },
        function(){
            // 噢，在`foo(..)`中有某些事情搞错了
        }
    );
}

// `baz(..)`同上
```

Promise解析没有必要一定发送消息，就像我们将Promise作为 未来值 考察时那样。它可以仅仅作为一种流程控制信号，就像前面的代码中那样使用。

另一种表达方式是：

```
function bar() {
    // `foo(..)`绝对已经完成了，那么做`bar(..)`的任务
}

function oopsBar() {
```

```
    // 噢，在`foo(..)`中有某些事情搞错了，那么`bar(..)`不会运行
}

// `baz()`和`oopsBaz()`同上

var p = foo( 42 );

p.then( bar, oopsBar );

p.then( baz, oopsBaz );
```

注意： 如果你以前见过基于Promise的代码，你可能会相信这段代码的最后两行应当写做 `p.then( .. ).then( .. )` ，使用链接，而不是 `p.then(..); p.then(..)` 。这将会是两种完全不同的行为，所以要小心！这种区别现在看起来可能不明显，但是它们实际上是我们目前还没有见过的异步模式：分割（splitting）/分叉（forking）。不必担心！本章后面我们会回到这个话题。

与将 `p` promise传入 `bar(..)` 和 `baz(..)` 相反，我们使用promise来控制 `bar(..)` 和 `baz(..)` 何时该运行，如果有这样的时刻。主要区别在于错误处理。

在第一个代码段的方式中，无论 `foo(..)` 是否成功 `bar(..)` 都会被调用，如果被通知 `foo(..)` 失败了的话它提供自己的后备逻辑。显然， `baz(..)` 也是这样做的。

在第二个代码段中， `bar(..)` 仅在 `foo(..)` 成功后才被调用，否则 `oopsBar(..)` 会被调用。 `baz(..)` 也是。

两种方式本身都 对。但会有一些情况使一种优于另一种。

在这两种方式中，从 `foo(..)` 返回的promise `p` 都被用于控制下一步发生什么。

另外，两个代码段都以对同一个promise `p` 调用两次 `then(..)` 结束，这展示了先前的观点，也就是Promise（一旦被解析）会永远保持相同的解析结果（完成或拒绝），而且可以按需要后续地被监听任意多次。

无论何时 `p` 被解析，下一步都将总是相同的，包括 现在 和 稍后。

Thenable 鸭子类型（Duck Typing）

Thenable鸭子类型（Duck Typing）

在Promise的世界中，一个重要的细节是如何确定一个值是否是纯粹的Promise。或者更直接地说，一个值会不会像Promise那样动作？

我们知道Promise是由 `new Promise(..)` 语法构建的，你可能会想 `p instanceof Promise` 将是一个可以接受的检查。但不幸的是，有几个理由表明它不是完全够用。

主要原因是，你可以从其他浏览器窗口中收到Promise值（iframe等），其他的浏览器窗口会拥有自己的不同于当前窗口/frame的Promise，这种检查将会在定位Promise实例时失效。

另外，一个库或框架可能会选择实现自己的Promise而不是用ES6原生的 `Promise` 实现。事实上，你很可能在根本没有Promise的老版本浏览器中通过一个库来使用Promise。

当我们在本章稍后讨论Promise的解析过程时，为什么识别并同化一个非纯种但相似Promise的值仍然很重要会愈发明显。但目前只需要相信我，它是拼图中很重要的一块。

如此，人们决定识别一个Promise（或像Promise一样动作的某些东西）的方法是定义一种称为“thenable”的东西，也就是任何拥有 `then(..)` 方法的对象或函数。这种方法假定任何这样的值都是一个符合Promise的thenable。

根据值的形状（存在什么属性）来推测它的“类型”的“类型检查”有一个一般的名称，称为“鸭子类型检查”——“如果它看起来像一只鸭子，并且叫起来像一只鸭子，那么它一定是一只鸭子”（参见本丛书的 类型与文法）。所以对thenable的鸭子类型检查可能大致是这样：

```
if (
    p !== null &&
    (
        typeof p === "object" ||
        typeof p === "function"
    ) &&
    typeof p.then === "function"
) {
    // 认为它是一个thenable!
}
else {
    // 不是一个thenable
}
```

晕！先把将这种逻辑在各种地方实现有点丑陋的事实放在一边不谈，这里还有更多更深层的麻烦。

如果你试着用一个偶然拥有 `then(..)` 函数的任意对象/函数来完成一个Promise，但你又没想把它当做一个Promise/thenable来对待，你的运气就用光了，因为它会被自动地识别为一个thenable并以特殊的规则来对待（见本章后面的部分）。

如果你不知道一个值上面拥有 `then(..)` 就更是这样。比如：

```
var o = { then: function(){} };

// 使`v`用`[[Prototype]]`链接到`o`
var v = Object.create( o );

v.someStuff = "cool";
v.otherStuff = "not so cool";

v.hasOwnProperty( "then" );        // false
```

`v` 看起来根本不像是一个Promise或thenable。它只是一个拥有一些属性的直白的对象。你可能只是想要把这个值像其他对象那样传递而已。

但你不知道的是，`v` 还 `[[Prototype]]` 连接着（见本丛书的 *this*与对象原型）另一个对象 `o`，在它上面偶然拥有一个 `then(..)`。所以thenable鸭子类型检查将会认为并假定 `v` 是一个thenable。噢。

它甚至不需要直接故意那么做：

```
Object.prototype.then = function(){};
Array.prototype.then = function(){};

var v1 = { hello: "world" };
var v2 = [ "Hello", "World" ];
```

`v1` 和 `v2` 都将被假定为是thenalbe的。你不能控制或预测是否有其他代码偶然或恶意地将 `then(..)` 加到 `Object.prototype`，`Array.prototype`，或其他任何原生原型上。而且如果这个指定的函数并不将它的任何参数作为回调调用，那么任何用这样的值被解析的Promise都将无声地永远挂起！疯狂。

听起来难以置信或不太可能？也许。

要知道，在ES6之前就有几种广为人知的非Promise库在社区中存在了，而且它们已经偶然拥有了称为 `then(..)` 的方法。这些库中的一些选择了重命名它们自己的方法来回避冲突（这很烂！）。另一些则因为它们无法改变来回避冲突，简单地降级为“不兼容基于Promise的代码”的不幸状态。

用来劫持原先非保留的—而且听起来完全是通用的— `then` 属性名称的标准决议是，没有值（或它

的任何委托），无论是过去，现在，还是将来，可以拥有 `then(..)` 函数，不管是有意的还是偶然的，否则这个值将在Promise系统中被混淆为一个thenable，从而可能产生非常难以追踪的Bug。

警告： 我不喜欢我们用thenable的鸭子类型来结束对Promise认知的方式。还有其他的选项，比如“branding”或者甚至是“anti-branding”；我们得到的似乎是一个最差劲儿的妥协。但它并不全是悲观与失望。thenable鸭子类型可以很有用，就像我们马上要看到的。只是要小心，如果thenable鸭子类型将不是Promise的东西误认为是Promise，它就可能成为灾难。

Promise的信任

Promise的信任

我们已经看过了两个强烈的类比，它们解释了Promise可以为我们的异步代码所做的事的不同方面。但如果我们停在这里，我们就可能会错过一个Promise模式建立的最重要的性质：信任。

随着 未来值 和 完成事件 的类别在我们探索的代码模式中的明确展开，有一个问题依然没有完全明确：Promise是为什么，以及如何被设计为来解决所有我们在第二章“信任问题”一节中提出的 控制倒转 的信任问题的。但是只要深挖一点儿，我们就可以发现一些重要的保证，来重建第二章中毁掉的对异步代码的信心！

让我们从复习仅使用回调的代码中的信任问题开始。当你传递一个回调给一个工具 `foo(..)` 的时候，它可能：

- 调用回调太早
- 调用回调太晚（或根本不调）
- 调用回调太少或太多次
- 没能传递必要的环境/参数
- 吞掉了任何可能发生的错误/异常

Promise的性质被有意地设计为给这些顾虑提供有用的，可复用的答案。

调的太早

这种顾虑主要是代码是否会引入类Zalgo效应，也就是一个任务有时会同步完地成，而有时会异步地完成，这将导致竞合状态。

Promise被定义为不能受这种顾虑的影响，因为即便是立即完成的Promise（比如 `new Promise(function(resolve){ resolve(42); })` ）也不可能被同步地 监听。

也就是说，但你在Promise上调用 `then(..)` 的时候，即便这个Promise已经被解析了，你给 `then(..)` 提供的回调也将 总是 被异步地调用（更多关于这里的内容，参照第一章的"Jobs"）。

不必再插入你自己的 `setTimeout(..,0)` 黑科技了。Promise自动地防止了Zalgo效应。

调的太晚

和前一点相似，在 `resolve(..)` 或 `reject(..)` 被Promise创建机制调用时，一个Promise的 `then(..)` 上注册的监听回调将自动地被排程。这些被排程好的回调将在下一个异步时刻被可预测地触发（参照第一章的"Jobs"）。

同步监听是不可能的，所以不可能有一个同步的任务链的运行来"推迟"另一个回调的发生。也就是说，当一个Promise被解析时，所有在 `then(..)` 上注册的回调都将被立即，按顺序地，在下一个异步机会时被调用（再一次，参照第一章的"Jobs"），而且没有任何在这些回调中发生的事情可以影响/推迟其他回调的调用。

举例来说：

```
p.then( function(){
    p.then( function(){
        console.log( "C" );
    } );
    console.log( "A" );
} );
p.then( function(){
    console.log( "B" );
} );
// A B C
```

这里，有赖于Promise如何定义操作， `"C"` 不可能干扰并优先于 `"B"` 。

Promise排程的怪现象

重要并需要注意的是，排程有许多微妙的地方：链接在两个分离的Promise上的回调之间的相对顺序，是不能可靠预测的。

如果两个promise `p1` 和 `p2` 都准备好被解析了，那么 `p1.then(..); p2.then(..)` 应当归结为首先调用 `p1` 的回调，然后调用 `p2` 的。但有一些微妙的情形可能会使这不成立，比如下面这样：

```
var p3 = new Promise( function(resolve,reject){
    resolve( "B" );
} );

var p1 = new Promise( function(resolve,reject){
```

```
    resolve( p3 );
} );

var p2 = new Promise( function(resolve,reject){
    resolve( "A" );
} );

p1.then( function(v){
    console.log( v );
} );

p2.then( function(v){
    console.log( v );
} );

// A B  <-- 不是你可能期望的 B A
```

我们稍后会更多地讲解这个问题，但如你所见， `p1` 不是被一个立即值所解析的，而是由另一个 promise `p3` 所解析，而 `p3` 本身被一个值 `"B"` 所解析。这种指定的行为将 `p3` 展开到 `p1` ，但是是异步地，所以在异步工作队列中 `p1` 的回调位于 `p2` 的回调之后（参照第一章的"Jobs"）。

为了回避这样的微妙的噩梦，你绝不应该依靠任何跨Promise的回调顺序/排程。事实上，一个好的实践方式是在代码中根本不要让多个回调的顺序成为问题。尽可能回避它。

根本不调回调

这是一个很常见的顾虑。Promise用几种方式解决它。

首先，没有任何东西（JS错误都不能）可以阻止一个Promise通知你它的解析（如果它被解析了的话）。如果你在一个Promise上同时注册了完成和拒绝回调，而且这个Promise被解析了，两个回调中的一个总会被调用。

当然，如果你的回调本身有JS错误，你可能不会看到你期望的结果，但是回调事实上已经被调用了。我们待会儿就会讲到如何在你的回调中收到关于一个错误的通知，因为就算是它们也不会被吞掉。

那如果Promise本身不管怎样永远没有被解析呢？即便是这种状态Promise也给出了答案，使用一个称为"竞赛（race）"的高级抽象。

```
// 一个使Promise超时的工具
function timeoutPromise(delay) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            reject( "Timeout!" );
        }, delay );
    } );
```

```
}

// 为`foo()`设置一个超时
Promise.race( [
    foo(),                    // 尝试调用`foo()`
    timeoutPromise( 3000 )    // 给它3秒钟
] )
.then(
    function(){
        // `foo(..)`及时地完成了！
    },
    function(err){
        // `foo()`不是被拒绝了，就是它没有及时完成
        // 那么可以考察`err`来知道是哪种情况
    }
);
```

这种Promise的超时模式有更多的细节需要考虑，但我们待会儿再回头讨论。

重要的是，我们可以确保一个信号作为 `foo(..)` 的结果，来防止它无限地挂起我们的程序。

调太少或太多次

根据定义，对于被调用的回调来讲 一次 是一个合适的次数。“太少”的情况将会是0次，和我们刚刚考察的从不调用是相同的。

“太多”的情况则很容易解释。Promise被定义为只能被解析一次。如果因为某些原因，Promise的创建代码试着调用 `resolve(..)` 或 `reject(..)` 许多次，或者试着同时调用它们俩，Promise将仅接受第一次解析，而无声地忽略后续的尝试。

因为一个Promise仅能被解析一次，所以任何 `then(..)` 上注册的（每个）回调将仅仅被调用一次。

当然，如果你把同一个回调注册多次（比如 `p.then(f); p.then(f);` ），那么它就会被调用注册的那么多次。响应函数仅被调用一次的保证并不能防止你砸自己的脚。

没能传入任何参数/环境

Promise可以拥有最多一个解析值（完成或拒绝）。

如果无论怎样你没有用一个值明确地解析它，它的值就是 `undefined` ，就像JS中常见的那样。但不管是什么值，它总是会被传入所有被注册的（并且适当地：完成或拒绝）回调中，不管是 现在 还是将来。

需要意识到的是：如果你使用多个参数调用 `resolve(..)` 或 `reject(..)` ，所有第一个参数之外的后续参数都会被无声地忽略。虽然这看起来违反了我们刚才描述的保证，但并不确切，因为它构成了一

种Promise机制的无效使用方式。其他的API无效使用方式（比如调用 `resolve(..)` 许多次）也都相似地 被保护，所以Promise的行为在这里是一致的（除了有一点点让人沮丧）。

如果你想传递多个值，你必须将它们包装在另一个单独的值中，比如一个 `array` 或一个 `object` 。

至于环境，JS中的函数总是保持他们被定义时所在作用域的闭包（见本系列的 作用域与闭包），所以它们理所当然地可以继续访问你提供的环境状态。当然，这对仅使用回调的设计来讲也是对的，所以这不能算是Promise带来的增益—但尽管如此，它依然是我们可以依赖的保证。

吞掉所有错误/异常

在基本的感觉上，这是前一点的重述。如果你用一个 理由（也就是错误消息）拒绝一个Promise，这个值就会被传入拒绝回调。

但是这里有一个更重要的事情。如果在Promise的创建过程中的任意一点，或者在监听它的解析的过程中，一个JS异常错误发生的话，比如 `TypeError` 或 `ReferenceError` ，这个异常将会被捕获，并且强制当前的Promise变为拒绝。

举例来说：

```
var p = new Promise( function(resolve,reject){
    foo.bar();    // `foo`没有定义，所以这是一个错误！
    resolve( 42 );    // 永远不会跑到这里 :(
} );

p.then(
    function fulfilled(){
        // 永远不会跑到这里 :(
    },
    function rejected(err){
        // `err`将是一个来自`foo.bar()`那一行的`TypeError`异常对象
    }
);
```

在 `foo.bar()` 上发生的JS异常变成了一个你可以捕获并响应的Promise拒绝。

这是一个重要的细节，因为它有效地解决了另一种潜在的Zalgo时刻，也就是错误可能会产生一个同步的反应，而没有错误的部分还是异步的。Promise甚至将JS异常都转化为异步行为，因此极大地降低了发生竞合状态的可能性。

但是如果Promise完成了，但是在监听过程中（在一个 `then(..)` 上注册的回调上）出现了JS异常错误会怎样呢？即便是那些也不会丢失，但你可能会发现处理它们的方式有些令人诧异，除非你深挖一些：

```
var p = new Promise( function(resolve,reject){
```

```
    resolve( 42 );
} );

p.then(
    function fulfilled(msg){
        foo.bar();
        console.log( msg );    // 永远不会跑到这里 :(
    },
    function rejected(err){
        // 也永远不会跑到这里 :(
    }
);
```

等一下，这看起来 `foo.bar()` 发生的异常确实被吞掉了。不要害怕，它没有。但更深层次的东西出问题了，也就是我们没能成功地监听他。 `p.then(..)` 调用本身返回另一个promise，是 那个promise将会被 `TypeError` 异常拒绝。

为什么它不能调用我们在这里定义的错误处理器呢？表面上看起来是一个符合逻辑的行为。但它会违反Promise一旦被解析就 不可变 的基本原则。 `p` 已经完成为值 `42` ，所以它不能因为在监听 `p` 的解析时发生了错误，而在稍后变成一个拒绝。

除了违反原则，这样的行为还可能造成破坏，假如说有多个在promise `p` 上注册的 `then(..)` 回调，因为有些会被调用而有些不会，而且至于为什么是很明显的。

可信的Promise？

为了基于Promise模式建立信任，还有最后一个细节需要考察。

无疑你已经注意到了，Promise根本没有摆脱回调。它们只是改变了回调传递的位置。与将一个回调传入 `foo(..)` 相反，我们从 `foo(..)` 那里拿回 某些东西 （表面上是一个纯粹的Promise），然后我们将回调传入这个 东西。

但为什么这要比仅使用回调的方式更可靠呢？我们如何确信我们拿回来的 某些东西 事实上是一个可信的Promise？这难道不是说我们相信它仅仅因为我们已经相信它了吗？

一个Promise经常被忽视，但是最重要的细节之一，就是它也为这个问题给出了解决方案。包含在原生的ES6 `Promise` 实现中，它就是 `Promise.resolve(..)` 。

如果你传递一个立即的，非Promise的，非thenable的值给 `Promise.resolve(..)` ，你会得到一个用这个值完成的promise。换句话说，下面两个promise `p1` 和 `p2` 的行为基本上完全相同：

```
var p1 = new Promise( function(resolve,reject){
    resolve( 42 );
} );

```

```
var p2 = Promise.resolve( 42 );
```

但如果你传递一个纯粹的Promise给 `Promise.resolve(..)` ，你会得到这个完全相同的promise：

```
var p1 = Promise.resolve( 42 );

var p2 = Promise.resolve( p1 );

p1 === p2; // true
```

更重要的是，如果你传递一个非Promise的thenable值给 `Promise.resolve(..)` ，它会试着将这个值展开，而且直到抽出一个最终具体的非Promise值之前，展开操作将会一直继续下去。

还记得我们先前讨论的thenable吗？

考虑这段代码：

```
var p = {
    then: function(cb) {
        cb( 42 );
    }
};

// 这工作起来没问题，但要靠运气
p
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // 永远不会跑到这里
    }
);
```

这个 `p` 是一个thenable，但它不是一个纯粹的Promise。很走运，它是合理的，正如大多数情况那样。但是如果你得到的是看起来像这样的东西：

```
var p = {
    then: function(cb,errcb) {
        cb( 42 );
        errcb( "evil laugh" );
    }
};

p
```

```
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // 噢，这里本不该运行
        console.log( err ); // evil laugh
    }
);
```

这个 `p` 是一个thenable，但它不是表现良好的promise。它是恶意的吗？或者它只是不知道Promise应当如何工作？老实说，这不重要。不管哪种情况，它都不那么可靠。

尽管如此，我们可以将这两个版本的 `p` 传入 `Promise.resolve(..)` ，而且我们将会得到一个我们期望的泛化，安全的结果：

```
Promise.resolve( p )
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // 永远不会跑到这里
    }
);
```

`Promise.resolve(..)` 会接受任何thenable，而且将它展开直至非thenable值。但你会从 `Promise.resolve(..)` 那里得到一个真正的，纯粹的Promise，一个你可以信任的东西。如果你传入的东西已经是一个纯粹的Promise了，那么你会单纯地将它拿回来，所以通过 `Promise.resolve(..)` 过滤来得到信任没有任何坏处。

那么我们假定，我们在调用一个 `foo(..)` 工具，而且不能确定我们能相信它的返回值是一个行为规范的Promise，但我们知道它至少是一个thenable。 `Promise.resolve(..)` 将会给我们一个可靠的Promise包装器来进行链式调用：

```
// 不要只是这么做：
foo( 42 )
.then( function(v){
    console.log( v );
} );

// 相反，这样做：
Promise.resolve( foo( 42 ) )
.then( function(v){
    console.log( v );
} );
```

注意： 将任意函数的返回值（thenable或不是thenable）包装在 `Promise.resolve(..)` 中的另一个好的副作用是，它可以很容易地将函数调用泛化为一个行为规范的异步任务。如果 `foo(42)` 有时返回一个立即值，而其他时候返回一个Promise， `Promise.resolve(foo(42))` ，将确保它总是返回Promise。并且使代码成为回避Zalgo效应的更好的代码。

信任建立了

希望前面的讨论使你现在完全理解了Promise是可靠的，而且更为重要的是，为什么信任对于建造强壮，可维护的软件来说是如此关键。

没有信任，你能用JS编写异步代码吗？你当然能。我们JS开发者在除了回调以外没有任何东西的情况下，写了将近20年的异步代码了。

但是一旦你开始质疑你到底能够以多大的程度相信你的底层机制，它实际上多么可预见，多么可靠，你就会开始理解回调的信任基础多么的摇摇欲坠。

Promise是一个用可靠语义来增强回调的模式，所以它的行为更合理更可靠。通过将回调的 控制倒转 反置过来，我们将控制交给一个可靠的系统（Promise），它是为了将你的异步处理进行清晰的表达而特意设计的。

链式流程

链式流程

我们已经被暗示过几次，但Promise不仅仅是一个单步的 这个然后那个 操作机制。当然，那是构建块儿，但事实证明我们可以将多个Promise串联在一起来表达一系列的异步步骤。

使这一切能够工作的关键，是Promise的两个固有行为：

- 每次你在一个Promise上调用`then(..)`的时候，它都创建并返回一个新的Promise，我们可以在它上面进行 链接。
- 无论你从`then(..)`调用的完成回调中（第一个参数）返回什么值，它都做为被链接的Promise的完成。

我们首先来说明一下这是什么意思，然后我们将会延伸出它是如何帮助我们创建异步顺序的控制流程的。考虑下面的代码：

```
var p = Promise.resolve( 21 );

var p2 = p.then( function(v){
    console.log( v );    // 21

    // 使用值`42`完成`p2`
    return v * 2;
} );

// 在`p2`后链接
p2.then( function(v){
    console.log( v );    // 42
} );
```

通过返回 `v * 2` （也就是 `42` ），我们完成了由第一个 `then(..)` 调用创建并返回的 `p2` promise。当 `p2` 的 `then(..)` 调用运行时，它从 `return v * 2` 语句那里收到完成信号。当然， `p2.then(..)` 还会创建另一个promise，我们将它存储在变量 `p3` 中。

但是不得不创建临时变量 `p2` （或 `p3` 等）有点儿恼人。幸运的是，我们可以简单地将这些链接在一起：

```
var p = Promise.resolve( 21 );

```

```
p
.then( function(v){
    console.log( v );    // 21

    // 使用值`42`完成被链接的promise
    return v * 2;
} )
// 这里是被链接的promise
.then( function(v){
    console.log( v );    // 42
} );
```

那么现在第一个 `then(..)` 是异步序列的第一步，而第二个 `then(..)` 就是第二步。它可以根据你的需要延伸至任意长。只要持续不断地用每个自动创建的Promise在前一个 `then(..)` 末尾进行连接即可。

但是这里错过了某些东西。要是我们想让第2步等待第1步去做一些异步的事情呢？我们使用的是一个立即的 `return` 语句，它立即完成了链接中的promise。

使Promise序列在每一步上都是真正异步的关键，需要回忆一下当你向 `Promise.resolve(..)` 传递一个Promise或thenable而非一个最终值时它如何执行。 `Promise.resolve(..)` 会直接返回收到的纯粹Promise，或者它会展开收到的thenable的值——并且它会递归地持续展开thenable。

如果你从完成（或拒绝）处理器中返回一个thenable或Promise，同样的展开操作也会发生。考虑这段代码：

```
var p = Promise.resolve( 21 );

p.then( function(v){
    console.log( v );    // 21

    // 创建一个promise并返回它
    return new Promise( function(resolve,reject){
        // 使用值`42`完成
        resolve( v * 2 );
    } );
} )
.then( function(v){
    console.log( v );    // 42
} );
```

即便我们把 `42` 包装在一个我们返回的promise中，它依然会被展开并作为下一个被链接的promise的解析，如此第二个 `then(..)` 仍然收到 `42` 。如果我们在这个包装promise中引入异步，一切还是会同样正常的工作：

```
var p = Promise.resolve( 21 );

p.then( function(v){
    console.log( v );    // 21

    // 创建一个promise并返回
    return new Promise( function(resolve,reject){
        // 引入异步！
        setTimeout( function(){
            // 使用值`42`完成
            resolve( v * 2 );
        }, 100 );
    } );
} )
.then( function(v){
    // 在上一步中的100毫秒延迟之后运行
    console.log( v );    // 42
} );
```

这真是不可思议的强大！现在我们可以构建一个序列，它可以有我们想要的任意多的步骤，而且每一步都可以按照需要来推迟下一步（或者不推迟）。

当然，在这些例子中一步一步向下传递的值是可选的。如果你没有返回一个明确的值，那么它假定一个隐含的 `undefined`，而且promise依然会以同样的方式链接在一起。如此，每个Promise的解析只不过是进行至下一步的信号。

为了演示更长的链接，让我们把推迟Promise的创建（没有解析信息）泛化为一个我们可以在多个步骤中复用的工具：

```
function delay(time) {
    return new Promise( function(resolve,reject){
        setTimeout( resolve, time );
    } );
}

delay( 100 ) // step 1
.then( function STEP2(){
    console.log( "step 2 (after 100ms)" );
    return delay( 200 );
} )
.then( function STEP3(){
    console.log( "step 3 (after another 200ms)" );
} )
.then( function STEP4(){
    console.log( "step 4 (next Job)" );
    return delay( 50 );
```

```
} )
.then( function STEP5(){
    console.log( "step 5 (after another 50ms)" );
} )
...
```

调用 `delay(200)` 创建了一个将在200毫秒内完成的promise，然后我们在第一个 `then(..)` 的完成回调中返回它，这将使第二个 `then(..)` 的promise等待这个200毫秒的promise。

注意： 正如刚才描述的，技术上讲在这个交替中有两个promise：一个200毫秒延迟的promise，和一个被第二个 `then(..)` 链接的promise。但你可能会发现将这两个promise组合在一起更容易思考，因为Promise机制帮你把它们的状态自动地混合到了一起。从这个角度讲，你可以认为 `return delay(200)` 创建了一个promise来取代早前一个返回的被链接的promise。

老实说，没有任何消息进行传递的一系列延迟作为Promise流程控制的例子不是很有用。让我们来看一个更加实在的场景：

与计时器不同，让我们考虑发起Ajax请求：

```
// 假定一个`ajax( {url}, {callback} )`工具

// 带有Promise的ajax
function request(url) {
    return new Promise( function(resolve,reject){
        // `ajax(..)`的回调应当是我们的promise的`resolve(..)`函数
        ajax( url, resolve );
    } );
}
```

我们首先定义一个 `request(..)` 工具，它构建一个promise表示 `ajax(..)` 调用的完成：

```
request( "http://some.url.1/" )
.then( function(response1){
    return request( "http://some.url.2/?v=" + response1 );
} )
.then( function(response2){
    console.log( response2 );
} );
```

注意： 开发者们通常遭遇的一种情况是，他们想用本身不支持Promise的工具（就像这里的 `ajax(..)` ，它期待一个回调）进行Promise式的异步流程控制。虽然ES6原生的 `Promise` 机制不会自动帮我们解决这种模式，但是在实践中所有的Promise库会帮我们这么做。它们通常称这种处理为“提升（lifting）”或“promise化”或其他的什么名词。我们稍后再回头讨论这种技术。

使用返回Promise的 `request(..)` ，通过用第一个URL调用它我们在链条中隐式地创建了第一步，然

后我们用第一个 `then(..)` 在返回的promise末尾进行连接。

一旦 `response1` 返回，我们用它的值来构建第二个URL，并且发起第二个 `request(..)` 调用。这第二个 `promise` 是 `return` 的，所以我们的异步流程控制的第三步将会等待这个Ajax调用完成。最终，一旦 `response2` 返回，我们就打印它。

我们构建的Promise链不仅是一个表达多步骤异步序列的流程控制，它还扮演者将消息从一步传递到下一步的消息管道。

要是Promise链中的某一步出错了会怎样呢？一个错误/异常是基于每个Promise的，意味着在链条的任意一点捕获这些错误是可能的，而且这些捕获操作在那一点上将链条“重置”，使它回到正常的操作上来：

```
// 步骤 1:
request( "http://some.url.1/" )

// 步骤 2:
.then( function(response1){
    foo.bar(); // 没有定义，错误！

    // 永远不会跑到这里
    return request( "http://some.url.2/?v=" + response1 );
} )

// 步骤 3:
.then(
    function fulfilled(response2){
        // 永远不会跑到这里
    },
    // 拒绝处理器捕捉错误
    function rejected(err){
        console.log( err );    // 来自 `foo.bar()` 的 `TypeError` 错误
        return 42;
    }
)

// 步骤 4:
.then( function(msg){
    console.log( msg );        // 42
} );
```

当错误在第2步中发生时，第3步的拒绝处理器将它捕获。拒绝处理器的返回值（在这个代码段里是 `42` ），如果有的话，将会完成下一步（第4步）的promise，如此整个链条又回到完成的状态。

注意： 就像我们刚才讨论过的，当我们从一个完成处理器中返回一个promise时，它会被展开并有可能推迟下一步。这对从拒绝处理器中返回的promise也是成立的，这样如果我们在第3步返回一个

promise而不是 `return 42` ，那么这个promise就可能会推迟第4步。不管是在 `then(..)` 的完成还是拒绝处理器中，一个被抛出的异常都将导致下一个（链接着的）promise立即用这个异常拒绝。

如果你在一个promise上调用 `then(..)` ，而且你只向它传递了一个完成处理器，一个假定的拒绝处理器会取而代之：

```
var p = new Promise( function(resolve,reject){
    reject( "Oops" );
} );

var p2 = p.then(
    function fulfilled(){
        // 永远不会跑到这里
    }
    // 如果忽略或者传入任何非函数的值,
    // 会有假定有一个这样的拒绝处理器
    // function(err) {
    //     throw err;
    // }
);
```

如你所见，这个假定的拒绝处理器仅仅简单地重新抛出错误，它最终强制 `p2` （链接着的promise）用同样的错误进行拒绝。实质上，它允许错误持续地在Promise链上传播，直到遇到一个明确定义的拒绝处理器。

注意： 稍后我们会讲到更多关于使用Promise进行错误处理的细节，因为会有更多微妙的细节需要关心。

如果没有一个恰当的合法的函数作为 `then(..)` 的完成处理器参数，也会有一个默认的处理器取而代之：

```
var p = Promise.resolve( 42 );

p.then(
    // 如果忽略或者传入任何非函数的值,
    // 会有假定有一个这样的完成处理器
    // function(v) {
    //     return v;
    // }
    null,
    function rejected(err){
        // 永远不会跑到这里
    }
);
```

如你所见，默认的完成处理器简单地将它收到的任何值传递给下一步（Promise）。

注意： `then(null,function(err){ .. })` 这种模式——仅处理拒绝（如果发生的话）但让成功通过——有一个缩写的API： `catch(function(err){ .. })` 。我们会在下一节中更全面地涵盖 `catch(..)` 。

让我们简要地复习一下使链式流程控制成为可能的Promise固有行为：

- 在一个Promise上的 `then(..)` 调用会自动生成一个新的Promise并返回。
- 在完成/拒绝处理器内部，如果你返回一个值或抛出一个异常，新返回的Promise（可以被链接的）将会相应地被解析。
- 如果完成或拒绝处理器返回一个Promise，它会被展开，所以无论它被解析为什么值，这个值都将变成从当前的 `then(..)` 返回的被链接的Promise的解析。

虽然链式流程控制很有用，但是将它认为是Promise的组合方式的副作用可能最准确，而不是它的主要意图。正如我们已经详细讨论过许多次的，Promise泛化了异步处理并且包装了与时间相关的值和状态，这才是让我们以这种有用的方式将它们链接在一起的原因。

当然，相对于我们在第二章中看到的一堆混乱的回调，这种链条的顺序表达是一个巨大的改进。但是仍然要蹚过相当多的模板代码（ `then(..)` and `function(){ .. }` ）。在下一章中，我们将看到一种极大美化顺序流程控制的表达模式，生成器（generators）。

术语：Resolve（解析），Fulfill（完成），和Reject（拒绝）

在你更多深入地学习Promise之前，在“解析（resolve）”，“完成（fulfill）”，和“拒绝（reject）”这些名词之间还有一些我们需要辨明的小困惑。首先让我们考虑一下 `Promise(..)` 构造器：

```
var p = new Promise( function(X,Y){
    // X() 给 fulfillment（完成）
    // Y() 给 rejection（拒绝）
} );
```

如你所见，有两个回调（标识为 `X` 和 `Y` ）被提供了。第一个 通常 用于表示Promise完成了，而第二个 总是 表示Promise拒绝了。但“通常”是什么意思？它对这些参数的正确命名暗示着什么呢？

最终，这只是你的用户代码，和将被引擎翻译为没有任何含义的东西的标识符，所以在 技术上 它无紧要； `foo(..)` 和 `bar(..)` 在功能性上是相等的。但是你用的词不仅会影响你如何考虑这段代码，还会影响你所在团队的其他开发者如何考虑它。将精心策划的异步代码错误地考虑，几乎可以说要比面条一般的回调还要差劲儿。

所以，某种意义上你如何称呼它们很关键。

第二个参数很容易决定。几乎所有的文献都使用 `reject(..)` 做为它的名称，因为这正是它（唯一！）要做的，对于命名来说这是一个很好的选择。我也强烈推荐你一直使用 `reject(..)` 。

但是关于第一个参数还是有些带有歧义，它在许多关于Promise的文献中常被标识为 `resolve(..)` 。这个词明显地是与“resolution（解析）”有关，它在所有的文献中（包括本书）广泛用于描述给Promise设定一个最终的值/状态。我们已经使用“解析Promise（resolve the Promise）”许多次来意味Promise的完成（fulfilling）或拒绝（rejecting）。

但是如果这个参数看起来被用于特指Promise的完成，为什么我们不更准确地叫它 `fulfill(..)` ，而是用 `resolve(..)` 呢？要回答这个问题，让我们看一下 `Promise` 的两个API方法：

```
var fulfilledPr = Promise.resolve( 42 );

var rejectedPr = Promise.reject( "Oops" );
```

`Promise.resolve(..)` 创建了一个Promise，它被解析为它被给予的值。在这个例子中， `42` 是一个一般的，非Promise，非thenable的值，所以完成的promise `fulfilledPr` 是为值 `42` 创建的。 `Promise.reject("Oops")` 为了原因 `"Oops"` 创建的拒绝的promise `rejectedPr` 。

现在让我们来解释为什么如果“resolve”这个词（正如 `Promise.resolve(..)` 里的）被明确用于一个既可能完成也可能拒绝的环境时，它没有歧义，反而更加准确：

```
var rejectedTh = {
    then: function(resolved,rejected) {
        rejected( "Oops" );
    }
};

var rejectedPr = Promise.resolve( rejectedTh );
```

就像我们在本章前面讨论的， `Promise.resolve(..)` 将会直接返回收到的纯粹的Promise，或者将收到的thenable展开。如果展开这个thenable之后是一个拒绝状态，那么从 `Promise.resolve(..)` 返回的Promise事实上是相同的拒绝状态。

所以对于这个API方法来说， `Promise.resolve(..)` 是一个好的，准确的名称，因为它实际上既可以得到完成的结果，也可以得到拒绝的结果。

`Promise(..)` 构造器的第一个回调参数既可以展开一个thenable（与 `Promise.resolve(..)` 相同），也可以展开一个Promise：

```
var rejectedPr = new Promise( function(resolve,reject){
    // 用一个被拒绝的promise来解析这个promise
    resolve( Promise.reject( "Oops" ) );
} );

```

```
rejectedPr.then(
    function fulfilled(){
        // 永远不会跑到这里
    },
    function rejected(err){
        console.log( err );    // "Oops"
    }
);
```

现在应当清楚了，对于 `Promise(..)` 构造器的第一个参数来说 `resolve(..)` 是一个合适的名称。

警告： 前面提到的 `reject(..)` 不会 像 `resolve(..)` 那样进行展开。如果你向 `reject(..)` 传递一个Promise/thenable值，这个没有被碰过的值将作为拒绝的理由。一个后续的拒绝处理器将会受到你传递给 `reject(..)` 的实际的Promise/thenable，而不是它底层的立即值。

现在让我们将注意力转向提供给 `then(..)` 的回调。它们应当叫什么（在文献和代码中）？我的建议是 `fulfilled(..)` 和 `rejected(..)` ：

```
function fulfilled(msg) {
    console.log( msg );
}

function rejected(err) {
    console.error( err );
}

p.then(
    fulfilled,
    rejected
);
```

对于 `then(..)` 的第一个参数的情况，它没有歧义地总是完成状态，所以没有必要使用带有双重意义的“resolve”术语。另一方面，ES6语言规范中使用 `onFulfilled(..)` 和 `onRejected(..)` 来标识这两个回调，所以它们是准确的术语。

错误处理

错误处理

我们已经看过几个例子，Promise拒绝—既可以通过有意调用 `reject(..)` ，也可以通过意外的JS异常—是如何在异步编程中允许清晰的错误处理的。让我们兜个圈子回去，将我们一带而过的一些细节弄清楚。

对大多数开发者来说，最自然的错误处理形式是同步的 `try..catch` 结构。不幸的是，它仅能用于同步状态，所以在异步代码模式中它帮不上什么忙：

```
function foo() {
    setTimeout( function(){
        baz.bar();
    }, 100 );
}

try {
    foo();
    // 稍后会从`baz.bar()`抛出全局错误
}
catch (err) {
    // 永远不会到这里
}
```

能有 `try..catch` 当然很好，但除非有某些附加的环境支持，它无法与异步操作一起工作。我们将会在第四章中讨论generator时回到这个话题。

在回调中，对于错误处理的模式已经有了一些新兴的模式，最有名的就是“错误优先回调”风格：

```
function foo(cb) {
    setTimeout( function(){
        try {
            var x = baz.bar();
            cb( null, x ); // 成功！
        }
        catch (err) {
            cb( err );
```

```
        }
    }, 100 );
}

foo( function(err,val){
    if (err) {
        console.error( err ); // 倒霉 :(
    }
    else {
        console.log( val );
    }
} );
```

注意： 这里的 `try..catch` 仅在 `baz.bar()` 调用立即地，同步地成功或失败时才能工作。如果 `baz.bar()` 本身是一个异步完成的函数，它内部的任何异步错误都不能被捕获。

我们传递给 `foo(..)` 的回调期望通过预留的 `err` 参数收到一个表示错误的信号。如果存在，就假定出错。如果不存在，就假定成功。

这类错误处理在技术上是 异步兼容的，但它根本组织的不好。用无处不在的 `if` 语句检查将多层错误优先回调编织在一起，将不可避免地将你置于回调地狱的危险之中（见第二章）。

那么我们回到Promise的错误处理，使用传递给 `then(..)` 的拒绝处理器。Promise不使用流行的“错误优先回调”设计风格，反而使用“分割回调”的风格；一个回调给完成，一个回调给拒绝：

```
var p = Promise.reject( "Oops" );

p.then(
    function fulfilled(){
        // 永远不会到这里
    },
    function rejected(err){
        console.log( err ); // "Oops"
    }
);
```

虽然这种模式表面上看起来十分有道理，但是Promise错误处理的微妙之处经常使它有点儿相当难以全面把握。

考虑下面的代码：

```
var p = Promise.resolve( 42 );

p.then(
    function fulfilled(msg){
        // 数字没有字符串方法,
```

```
        // 所以这里抛出一个错误
        console.log( msg.toLowerCase() );
    },
    function rejected(err){
        // 永远不会到这里
    }
);
```

如果 `msg.toLowerCase()` 合法地抛出一个错误（它会的！），为什么我们的错误处理器没有得到通知？正如我们早先解释的，这是因为 这个 错误处理器是为 `p` promise准备的，也就是已经被值 `42` 完成的那个promise。 `p` promise是不可变的，所以唯一可以得到错误通知的promise是由 `p.then(..)` 返回的那个，而在这里我们没有捕获它。

这应当解释了：为什么Promise的错误处理是易错的。错误太容易被吞掉了，而这很少是你有意这么做的。

警告： 如果你以一种不合法的方式使用Promise API，而且有错误阻止正常的Promise构建，其结果将是一个立即被抛出的异常，而不是一个拒绝**Promise**。这是一些导致Promise构建失败的错误用法： `new Promise(null)` ， `Promise.all()` ， `Promise.race(42)` 等等。如果你没有足够合法地使用Promise API来首先实际构建一个Promise，你就不能得到一个拒绝Promise！

绝望的深渊

几年前Jeff Atwood曾经写到：编程语言总是默认地以这样的方式建立，开发者们会掉入“绝望的深渊”（http://blog.codinghorror.com/falling-into-the-pit-of-success/ ）—在这里意外会被惩罚—而你不得不更努力地使它正确。他恳求我们相反地创建“成功的深渊”，就是你会默认地掉入期望的（成功的）行为，而如此你不得不更努力地去失败。

毫无疑问，Promise的错误处理是一种“绝望的深渊”的设计。默认情况下，它假定你想让所有的错误都被Promise的状态吞掉，而且如果你忘记监听这个状态，错误就会默默地凋零/死去—通常是绝望的。

为了回避把一个被遗忘/抛弃的Promise的错误无声地丢失，一些开发者宣称Promise链的“最佳实践”是，总是将你的链条以 `catch(..)` 终结，就像这样：

```
var p = Promise.resolve( 42 );

p.then(
    function fulfilled(msg){
        // 数字没有字符串方法,
        // 所以这里抛出一个错误
        console.log( msg.toLowerCase() );
    }
)
.catch( handleErrors );
```

因为我们没有给 `then(..)` 传递拒绝处理器，默认的处理器会顶替上来，它仅仅简单地将错误传播到链条的下一个promise中。如此，在 `p` 中发生的错误，与在 `p` 之后的解析中（比如 `msg.toLowerCase()` ）发生的错误都将会过滤到最后的 `handleErrors(..)` 中。

问题解决了，对吧？没那么容易！

要是 `handleErrors(..)` 本身也有错误呢？谁来捕获它？这里还有一个没人注意的promise: `catch(..)` 返回的promise，我们没有对它进行捕获，也没注册拒绝处理器。

你不能仅仅将另一个 `catch(..)` 贴在链条末尾，因为它也可能失败。Promise链的最后一步，无论它是什么，总有可能，即便这种可能性逐渐减少，悬挂着一个困在未被监听的Promise中的，未被捕获的错误。

听起来像一个不可解的迷吧？

处理未被捕获的错误

这不是一个很容易就能完全解决的问题。但是有些接近于解决的方法，或者说 更好的方法。

一些Promise库有一些附加的方法，可以注册某些类似于“全局的未处理拒绝”的处理器，全局上不会抛出错误，而是调用它。但是他们识别一个错误是“未被捕获的错误”的方案是，使用一个任意长的计时器，比如说3秒，从拒绝的那一刻开始计时。如果一个Promise被拒绝但没有错误处理在计时器被触发前注册，那么它就假定你不会注册监听器了，所以它是“未被捕获的”。

实践中，这个方法在许多库中工作的很好，因为大多数用法不会在Promise拒绝和监听这个拒绝之间有很明显的延迟。但是这个模式有点儿麻烦，因为3秒实在太随意了（即便它是实证过的），还因为确实有些情况你想让一个Promise在一段不确定的时间内持有它的拒绝状态，而且你不希望你的“未捕获错误”处理器因为这些误报（还没处理的“未捕获错误”）而被调用。

另一种常见的建议是，Promise应当增加一个 `done(..)` 方法，它实质上标志着Promise链的“终结”。 `done(..)` 不会创建并返回一个Promise，所以传递给 `done(..)` 的回调很明显地不会链接上一个不存在的Promise链，并向它报告问题。

那么接下来会发什么？正如你通常在未处理错误状态下希望的那样，在 `done(..)` 的拒绝处理器内部的任何异常都作为全局的未捕获错误抛出（基本上扔到开发者控制台）：

```
var p = Promise.resolve( 42 );

p.then(
    function fulfilled(msg){
        // 数字没有字符串方法,
        // 所以这里抛出一个错误
        console.log( msg.toLowerCase() );
    }
```

```
)
.done( null, handleErrors );

// 如果`handleErrors(..)`自身发生异常，它会在这里被抛出到全局
```

这听起来要比永不终结的链条或随意的超时要吸引人。但最大的问题是，它不是ES6标准，所以不管听起来多么好，它成为一个可靠而普遍的解决方案还有很长的距离。

那我们就卡在这里了？不完全是。

浏览器有一个我们的代码没有的能力：它们可以追踪并确定一个对象什么时候被废弃并可以作为垃圾回收。所以，浏览器可以追踪Promise对象，当它们被当做垃圾回收时，如果在它们内部存在一个拒绝状态，浏览器就可以确信这是一个合法的“未捕获错误”，它可以信心十足地知道应当在开发者控制台上报告这一情况。

注意： 在写作本书的时候，Chrome和Firefox都早已试图实现这种“未捕获拒绝”的能力，虽然至多也就是支持的不完整。

然而，如果一个Promise不被垃圾回收—通过许多不同的代码模式，这极其容易不经意地发生—浏览器的垃圾回收检测不会帮你知道或诊断你有一个拒绝的Promise静静地躺在附近。

还有其他选项吗？有。

成功的深渊

以下讲的仅仅是理论上，Promise 可能 在某一天变成什么样的行为。我相信那会比我们现在拥有的优越许多。而且我想这种改变可能会发生在后ES6时代，因为我不认为它会破坏Web的兼容性。另外，如果你小心行事，它是可以被填补（polyfilled）/预填补（prollyfilled）的。让我们来看一下：

- Promise可以默认为是报告(向开发者控制台)一切拒绝的，就在下一个Job或事件轮询tick，如果就在这时Promise上没有注册任何错误处理器。
- 如果你希望拒绝的Promise在被监听前，将其拒绝状态保持一段不确定的时间。你可以调用`defer()`，它会压制这个Promise自动报告错误。

如果一个Promise被拒绝，默认地它会吵吵闹闹地向开发者控制台报告这个情况（而不是默认不出声）。你既可以选择隐式地处理这个报告（通过在拒绝之前注册错误处理器），也可以选择明确地处理这个报告（使用 `defer()` ）。无论哪种情况，你 都控制着这种误报。

考虑下面的代码：

```
var p = Promise.reject( "Oops" ).defer();

// `foo(..)`返回Promise
foo( 42 )
```

```
.then(
    function fulfilled(){
        return p;
    },
    function rejected(err){
        // 处理`foo(..)`的错误
    }
);
...
```

我们创建了 `p` ，我们知道我们会为了使用/监听它的拒绝而等待一会儿，所以我们调用 `defer()` —如此就不会有全局的报告。 `defer()` 单纯地返回同一个promise，为了链接的目的。

从 `foo(..)` 返回的promise 当即 就添附了一个错误处理器，所以这隐含地跳出了默认行为，而且不会有全局的关于错误的报告。

但是从 `then(..)` 调用返回的promise没有 `defer()` 或添附错误处理器，所以如果它被拒绝（从它内部的任意一个解析处理器中），那么它就会向开发者控制台报告一个未捕获错误。

这种设计称为成功的深渊。默认情况下，所有的错误不是被处理就是被报告—这几乎是所有开发者在几乎所有情况下所期望的。你要么不得不注册一个监听器，要么不得不有意什么都不做，并指示你要将错误处理推迟到 稍后；你仅为这种特定情况选择承担额外的责任。

这种方式唯一真正的危险是，你 `defer()` 了一个Promise但是实际上没有监听/处理它的拒绝。

但你不得不有意地调用 `defer()` 来选择进入绝望深渊—默认是成功深渊—所以对于从你自己的错误中拯救你这件事来说，我们能做的不多。

我觉得对于Promise的错误处理还有希望（在后ES6时代）。我希望上层人物将会重新思考这种情况并考虑选用这种方式。同时，你可以自己实现这种方式（给读者们的挑战练习！），或使用一个 聪明的Promise库来为你这么做。

注意： 这种错误处理/报告的确切的模型已经在我的 *asynquence* Promise抽象库中实现，我们会在本书的附录A中讨论它。

Promise 模式

Promise模式

我们已经隐含地看到了使用Promise链的顺序模式（这个-然后-这个-然后-那个的流程控制），但是我们还可以在Promise的基础上抽象出许多其他种类的异步模式。这些模式用于简化异步流程控制的的表达—它可以使我们的代码更易于推理并且更易于维护—即便是我们程序中最复杂的部分。

有两个这样的模式被直接编码在ES6原生的 `Promise` 实现中，所以我们免费的得到了它们，来作为我们其他模式的构建块儿。

Promise.all([..])

在一个异步序列（Promise链）中，在任何给定的时刻都只有一个异步任务在被协调—第2步严格地接着第1步，而第3步严格地接着第2步。但要是并发（也叫“并行地”）地去做两个或以上的步骤呢？

用经典的编程术语，一个“门（gate）”是一种等待两个或更多并行/并发任务都执行完再继续的机制。它们完成的顺序无关紧要，只是它们不得不都完成才能让门打开，继而让流程控制通过。

在Promise API中，我们称这种模式为 `all([ .. ])` 。

比方说你想同时发起两个Ajax请求，在发起第三个Ajax请求发起之前，等待它们都完成，而不管它们的顺序。考虑这段代码：

```
1.  // `request(..)`是一个兼容Promise的Ajax工具
2.  // 就像我们在本章早前定义的
3.
4.  var p1 = request( "http://some.url.1/" );
5.  var p2 = request( "http://some.url.2/" );
6.
7.  Promise.all( [p1,p2] )
8.  .then( function(msgs){
9.      // `p1`和`p2`都已完成，这里将它们的消息传入
10.     return request(
11.         "http://some.url.3/?v=" + msgs.join(",")
```

```
    );
} )
.then( function(msg){
    console.log( msg );
} );
```

`Promise.all([ .. ])` 期待一个单独的参数，一个 `array` ，一般由Promise的实例组成。从 `Promise.all([ .. ])` 返回的promise将会收到完成的消息（在这段代码中是 `msgs` ），它是一个由所有被传入的promise的完成消息按照被传入的顺序构成的 `array` （与完成的顺序无关）。

注意： 技术上讲，被传入 `Promise.all([ .. ])` 的 `array` 的值可以包括Promise，thenable，甚至是立即值。这个列表中的每一个值都实质上通过 `Promise.resolve(..)` 来确保它是一个可以被等待的纯粹的Promise，所以一个立即值将被范化为这个值的一个Promise。如果这个 `array` 是空的，主Promise将会立即完成。

从 `Promise.resolve(..)` 返回的主Promise将会在所有组成它的promise完成之后才会被完成。如果其中任意一个promise被拒绝， `Promise.all([ .. ])` 的主Promise将立即被拒绝，并放弃所有其他promise的结果。

要记得总是给每个promise添加拒绝/错误处理器，即使和特别是那个从 `Promise.all([ .. ])` 返回的promise。

Promise.race([..])

虽然 `Promise.all([ .. ])` 并发地协调多个Promise并假定它们都需要被完成，但是有时候你只想应答“冲过终点的第一个Promise”，而让其他的Promise被丢弃。

这种模式经典地被称为“闩”，但在Promise中它被称为一个“竞合（race）”。

警告： 虽然“只有第一个冲过终点的算赢”是一个非常合适被比喻，但不幸的是“竞合（race）”是一个被占用的词，因为“竞合状态（race conditions）”通常被认为是程序中的Bug（见第一章）。不要把 `Promise.race([ .. ])` 与“竞合状态（race conditions）”搞混了。

“竞合状态（race conditions）”也期待一个单独的 `array` 参数，含有一个或多个Promise，thenable，或立即值。与立即值进行竞合并没有多大实际意义，因为很明显列表中的第一个会胜出—就像赛跑时有一个选手在终点线上起跑！

和 `Promise.all([ .. ])` 相似， `Promise.race([ .. ])` 将会在任意一个Promise解析为完成时完成，而且它会在任意一个Promise解析为拒绝时拒绝。

注意： 一个“竞合（race）”需要至少一个“选手”，所以如果你传入一个空的 `array` ， `race([..])` 的主Promise将不会立即解析，反而是永远不会被解析。这是砸自己的脚！ES6应当将它规范为要么完成，要么拒绝，或者要么抛出某种同步错误。不幸的是，因为在ES6的 `Promise` 之前的Promise库的优先权高，他们不得不把这个坑留在这儿，所以要小心绝不要传入

一个空 `array` 。

让我们重温刚才的并发Ajax的例子，但是在 `p1` 和 `p2` 竞合的环境下：

```
// `request(..)`是一个兼容Promise的Ajax工具
// 就像我们在本章早前定义的

var p1 = request( "http://some.url.1/" );
var p2 = request( "http://some.url.2/" );

Promise.race( [p1,p2] )
.then( function(msg){
    // `p1`或`p2`会赢得竞合
    return request(
        "http://some.url.3/?v=" + msg
    );
} )
.then( function(msg){
    console.log( msg );
} );
```

因为只有一个Promise会胜出，所以完成的值是一个单独的消息，而不是一个像 `Promise.all([ .. ])` 中那样的 `array` 。

超时竞合

我们早先看过这个例子，描述 `Promise.race([ .. ])` 如何能够用于表达“promise超时”模式：

```
// `foo()`是一个兼容Promise

// `timeoutPromise(..)`在早前定义过，
// 返回一个在指定延迟之后会被拒绝的Promise

// 为`foo()`设置一个超时
Promise.race( [
    foo(),                  // 尝试`foo()`
    timeoutPromise( 3000 )  // 给它3秒钟
] )
.then(
    function(){
        // `foo(..)`及时地完成了！
    },
    function(err){
        // `foo()`要么是被拒绝了，要么就是没有及时完成
        // 可以考察`err`来知道是哪一个原因
    }
```

```
19. );
```

这种超时模式在绝大多数情况下工作的很好。但这里有一些微妙的细节要考虑，而且坦率的说它们对于 `Promise.race([ .. ])` 和 `Promise.all([ .. ])` 都同样需要考虑。

“Finally”

要问的关键问题是，“那些被丢弃/忽略的promise发生了什么？”我们不是从性能的角度在问这个问题—它们通常最终会变成垃圾回收的合法对象—而是从行为的角度（副作用等等）。Promise不能被取消—而且不应当被取消，因为那会摧毁本章稍后的“Promise不可取消”一节中要讨论的外部不可变性—所以它们只能被无声地忽略。

但如果前面例子中的 `foo()` 占用了某些资源，但超时首先触发而且导致这个promise被忽略了呢？这种模式中存在某种东西可以在超时后主动释放被占用的资源，或者取消任何它可能带来的副作用吗？要是你想做的全部只是记录下 `foo()` 超时的事实呢？

一些开发者提议，Promise需要一个 `finally(..)` 回调注册机制，它总是在Promise解析时被调用，而且允许你制定任何可能的清理操作。在当前的语言规范中它还不存在，但它可能会在ES7+中加入。我们不得不边走边看了。

它看起来可能是这样：

```
1. var p = Promise.resolve( 42 );
2.
3. p.then( something )
4. .finally( cleanup )
5. .then( another )
6. .finally( cleanup );
```

注意： 在各种Promise库中， `finally(..)` 依然会创建并返回一个新的Promise（为了使链条延续下去）。如果 `cleanup(..)` 函数返回一个Promise，它将会链入链条，这意味着你可能还有我们刚才讨论的未处理拒绝的问题。

同时，我们可以制造一个静态的帮助工具来让我们观察（但不干涉）Promise的解析：

```
1. // 填补的安全检查
2. if (!Promise.observe) {
3.     Promise.observe = function(pr,cb) {
4.         // 从侧面观察`pr`的解析
5.         pr.then(
6.             function fulfilled(msg){
7.                 // 异步安排回调（作为Job）
8.                 Promise.resolve( msg ).then( cb );
9.             },
10.            function rejected(err){
```

```
                // 异步安排回调（作为Job）
                Promise.resolve( err ).then( cb );
            }
        );

        // 返回原本的promise
        return pr;
    };
}
```

这是我们在前面的超时例子中如何使用它：

```
Promise.race( [
    Promise.observe(
        foo(),                      // 尝试`foo()`
        function cleanup(msg){
            // 在`foo()`之后进行清理，即便它没有及时完成
        }
    ),
    timeoutPromise( 3000 )    // 给它3秒钟
] )
```

这个 `Promise.observe(..)` 帮助工具只是描述你如何在不干扰Promise的情况下观测它的完成。其他的Promise库有他们自己的解决方案。不论你怎么做，你都将很可能有个地方想用来确认你的Promise没有意外地被无声地忽略掉。

all([..]) 与 race([..]) 的变种

原生的ES6Promise带有内建的 `Promise.all([ .. ])` 和 `Promise.race([ .. ])` ，这里还有几个关于这些语义的其他常用的变种模式：

- `none([ .. ])` 很像 `all([ .. ])` ，但是完成和拒绝被转置了。所有的Promise都需要被拒绝—拒绝变成了完成值，反之亦然。
- `any([ .. ])` 很像 `all([ .. ])` ，但它忽略任何拒绝，所以只有一个需要完成即可，而不是它们所有的。
- `first([ .. ])` 像是一个带有 `any([ .. ])` 的竞合，它忽略任何拒绝，而且一旦有一个Promise完成时，它就立即完成。
- `last([ .. ])` 很像 `first([ .. ])` ，但是只有最后一个完成胜出。

某些Promise抽象工具库提供这些方法，但你也可以用Promise机制的 `race([ .. ])` 和 `all([ .. ])` ，自己定义他们。

比如，这是我们如何定义 `first([..])` ：

```
// 填补的安全检查
if (!Promise.first) {
    Promise.first = function(prs) {
        return new Promise( function(resolve,reject){
            // 迭代所有的promise
            prs.forEach( function(pr){
                // 泛化它的值
                Promise.resolve( pr )
                // 无论哪一个首先成功完成，都由它来解析主promise
                .then( resolve );
            } );
        } );
    };
}
```

注意： 这个 `first(..)` 的实现不会在它所有的promise都被拒绝时拒绝；它会简单地挂起，很像 `Promise.race([])` 。如果需要，你可以添加一些附加逻辑来追踪每个promise的拒绝，而且如果所有的都被拒绝，就在主promise上调用 `reject()` 。我们将此作为练习留给读者。

并发迭代

有时候你想迭代一个Promise的列表，并对它们所有都实施一些任务，就像你可以对同步的 `array` 做的那样（比如， `forEach(..)` ， `map(..)` ， `some(..)` ，和 `every(..)` ）。如果对每个Promise实施的操作根本上是同步的，它们工作的很好，正如我们在前面的代码段中用过的 `forEach(..)` 。

但如果任务在根本上是异步的，或者可以/应当并发地实施，你可以使用许多库提供的异步版本的这些工具方法。

比如，让我们考虑一个异步的 `map(..)` 工具，它接收一个 `array` 值（可以是Promise或任何东西），外加一个对数组中每一个值实施的函数（任务）。 `map(..)` 本身返回一个promise，它的完成值是一个持有每个任务的异步完成值的 `array` （以与映射（mapping）相同的顺序）：

```
if (!Promise.map) {
    Promise.map = function(vals,cb) {
        // 一个等待所有被映射的promise的新promise
        return Promise.all(
            // 注意：普通的数组`map(..)`，
            // 将值的数组变为promise的数组
            vals.map( function(val){
                // 将`val`替换为一个在`val`
                // 异步映射完成后才解析的新promise
                return new Promise( function(resolve){
                    cb( val, resolve );
                } );
```

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

注意： 在这种 `map(..)` 的实现中，你无法表示异步拒绝，但如果一个在映射的回调内部发生一个同步的异常/错误，那么 `Promise.map(..)` 返回的主Promise就会拒绝。

让我们描绘一下对一组Promise（不是简单的值）使用 `map(..)`：

```
var p1 = Promise.resolve( 21 );
var p2 = Promise.resolve( 42 );
var p3 = Promise.reject( "Oops" );

// 将列表中的值翻倍，即便它们在Promise中
Promise.map( [p1,p2,p3], function(pr,done){
    // 确保列表中每一个值都是Promise
    Promise.resolve( pr )
    .then(
        // 将值作为`v`抽取出来
        function(v){
            // 将完成的`v`映射到新的值
            done( v * 2 );
        },
        // 或者，映射到promise的拒绝消息上
        done
    );
} )
.then( function(vals){
    console.log( vals );    // [42,84,"Oops"]
} );
```

Promise API概览

Promise API概览

让我们复习一下我们已经在本章中零散地展开的ES6 `Promise` API。

注意： 下面的API尽管在ES6中是原生的，但也存在一些语言规范兼容的填补（不光是扩展Promise库），它们定义了 `Promise` 和与之相关的所有行为，所以即使是在前ES6时代的浏览器中你也以使用原生的Promise。这类填补的其中之一是“Native Promise Only”(http://github.com/getify/native-promise-only)，我写的！

new Promise(..)构造器

揭示构造器（*revealing constructor*） `Promise(..)` 必须与 `new` 一起使用，而且必须提供一个被同步/立即调用的回调函数。这个函数被传入两个回调函数，它们作为promise的解析能力。我们通常将它们标识为 `resolve(..)` 和 `reject(..)`：

```
var p = new Promise( function(resolve,reject){
    // `resolve(..)`给解析/完成的promise
    // `reject(..)`给拒绝的promise
} );
```

`reject(..)` 简单地拒绝promise，但是 `resolve(..)` 既可以完成promise，也可以拒绝promise，这要看它被传入什么值。如果 `resolve(..)` 被传入一个立即的，非Promise，非thenable的值，那么这个promise将用这个值完成。

但如果 `resolve(..)` 被传入一个Promise或者thenable的值，那么这个值将被递归地展开，而且无论它最终解析结果/状态是什么，都将被promise采用。

Promise.resolve(..) 和 Promise.reject(..)

一个用于创建已被拒绝的Promise的简便方法是 `Promise.reject(..)`，所以这两个promise是等价的：

```
var p1 = new Promise( function(resolve,reject){
```

```
    reject( "Oops" );
} );

var p2 = Promise.reject( "Oops" );
```

与 `Promise.reject(..)` 相似， `Promise.resolve(..)` 通常用来创建一个已完成的Promise。然而， `Promise.resolve(..)` 还会展开thenale值（就像我们已经几次讨论过的）。在这种情况下，返回的Promise将会采用你传入的thenable的解析，它既可能是完成，也可能是拒绝：

```
var fulfilledTh = {
    then: function(cb) { cb( 42 ); }
};
var rejectedTh = {
    then: function(cb,errCb) {
        errCb( "Oops" );
    }
};

var p1 = Promise.resolve( fulfilledTh );
var p2 = Promise.resolve( rejectedTh );

// `p1`将是一个完成的promise
// `p2`将是一个拒绝的promise
```

而且要记住，如果你传入一个纯粹的Promise， `Promise.resolve(..)` 不会做任何事情；它仅仅会直接返回这个值。所以在你不知道其本性的值上调用 `Promise.resolve(..)` 不会有额外的开销，如果它偶然已经是一个纯粹的Promise。

then(..) 和 catch(..)

每个Promise实例（不是 `Promise` API 名称空间）都有 `then(..)` 和 `catch(..)` 方法，它们允许你为Promise注册成功或拒绝处理器。一旦Promise被解析，它们中的一个就会被调用，但不是都会被调用，而且它们总是会被异步地调用（参见第一章的“Jobs”）。

`then(..)` 接收两个参数，第一个用于完成回调，第二个用户拒绝回调。如果它们其中之一被省略，或者被传入一个非函数的值，那么一个默认的回调就会分别顶替上来。默认的完成回调简单地将值向下传递，而默认的拒绝回调简单地重新抛出（传播）收到的拒绝理由。

`catch(..)` 仅仅接收一个拒绝回调作为参数，而且会自动的顶替一个默认的成功回调，就像我们讨论过的。换句话说，它等价于 `then(null,..)` ：

```
p.then( fulfilled );

p.then( fulfilled, rejected );
```

```
5. p.catch( rejected ); // 或者`p.then( null, rejected )`
```

`then(..)` 和 `catch(..)` 也会创建并返回一个新的promise，它可以用来表达Promise链式流程控制。如果完成或拒绝回调有异常被抛出，这个返回的promise就会被拒绝。如果这两个回调之一返回一个立即，非Promise，非thenable值，那么这个值就会作为被返回的promise的完成。如果完成处理器指定地返回一个promise或thenable值这个值就会被展开而且变成被返回的promise的解析。

Promise.all([..]) 和 Promise.race([..])

在ES6的 `Promise` API的静态帮助方法 `Promise.all([ .. ])` 和 `Promise.race([ .. ])` 都创建一个Promise作为它们的返回值。这个promise的解析完全由你传入的promise数组控制。

对于 `Promise.all([ .. ])` ，为了被返回的promise完成，所有你传入的promise都必须完成。如果其中任意一个被拒绝，返回的主promise也会立即被拒绝（丢弃其他所有promise的结果）。至于完成状态，你会收到一个含有所有被传入的promise的完成值的 `array` 。至于拒绝状态，你仅会收到第一个promise拒绝的理由值。这种模式通常称为“门”：在门打开前所有人都必须到达。

对于 `Promise.race([ .. ])` ，只有第一个解析（成功或拒绝）的promise会“胜出”，而且不论解析的结果是什么，都会成为被返回的promise的解析结果。这种模式通常成为“闩”：第一个打开门闩的人才能进来。考虑这段代码：

```
1. var p1 = Promise.resolve( 42 );
2. var p2 = Promise.resolve( "Hello World" );
3. var p3 = Promise.reject( "Oops" );
4.
5. Promise.race( [p1,p2,p3] )
6. .then( function(msg){
7.     console.log( msg );        // 42
8. } );
9.
10. Promise.all( [p1,p2,p3] )
11. .catch( function(err){
12.     console.error( err );    // "Oops"
13. } );
14.
15. Promise.all( [p1,p2] )
16. .then( function(msgs){
17.     console.log( msgs );    // [42,"Hello World"]
18. } );
```

警告： 要小心！如果一个空的 `array` 被传入 `Promise.all([ .. ])` ，它会立即完成，但 `Promise.race([ .. ])` 却会永远挂起，永远不会解析。

ES6的 `Promise` API十分简单和直接。对服务于大多数基本的异步情况来说它足够好了，而且当你要把你的代码从回调地狱变为某些更好的东西时，它是一个开始的好地方。

但是依然还有许多应用程序所要求的精巧的异步处理，由于Promise本身所受的限制而不能解决。在下一节中，为了有效利用Promise库，我们将深入检视这些限制。

Promise 的限制

Promise的限制

本节中我们将要讨论的许多细节已经在这一章中被提及了，但我们将明确地复习这些限制。

顺序的错误处理

我们在本章前面的部分详细讲解了Promise风格的错误处理。Promise的设计方式—特别是他们如何链接—所产生的限制，创建了一个非常容易掉进去的陷阱，Promise链中的错误会被意外地无声地忽略掉。

但关于Promise的错误还有一些其他事情要考虑。因为Promise链只不过是将组成它的Promise连在一起，没有一个实体可以用来将整个链条表达为一个单独的 东西，这意味着没有外部的方法能够监听可能发生的任何错误。

如果你构建一个不包含错误处理器的Promise链，这个链条的任意位置发生的任何错误都将沿着链条向下无限传播，直到被监听为止（通过在某一步上注册拒绝处理器）。所以，在这种特定情况下，拥有链条的最后一个promise的引用就够了（下面代码段中的 `p` ），因为你可以在这里注册拒绝处理器，而且它会被所有传播的错误通知：

```
// `foo(..)`, `STEP2(..)` 和 `STEP3(..)`
// 都是promise兼容的工具

var p = foo( 42 )
.then( STEP2 )
.then( STEP3 );
```

虽然这看起来有点儿小糊涂，但是这里的 `p` 没有指向链条中的第一个promise（ `foo(42)` 调用中来的那一个），而是指向了最后一个promise，来自于 `then(STEP3)` 调用的那一个。

另外，这个promise链条上看不到一个步骤做了自己的错误处理。这意味着你可以在 `p` 上注册一个拒绝处理器，如果在链条的任意位置发生了错误，它就会被通知。

```
p.catch( handleErrors );
```

但如果这个链条中的某一步事实上做了自己的错误处理（也许是隐藏/抽象出去了，所以你看不到），那么你的 `handleErrors(..)` 就不会被通知。这可能是你想要的—它毕竟是一个“被处理过的拒绝”—但它也可能 不 是你想要的。完全缺乏被通知的能力（被“已处理过的”拒绝错误通知）是一个在某些用法中约束功能的一种限制。

它基本上和 `try..catch` 中存在的限制是相同的，它可以捕获一个异常并简单地吞掉。所以这不是一个 **Promise特有** 的问题，但它确实是一个我们希望绕过的限制。

不幸的是，许多时候Promise链序列的中间步骤不会被留下引用，所以没有这些引用，你就不能添加错误处理器来可靠地监听错误。

单独的值

根据定义，Promise只能有一个单独的完成值或一个单独的拒绝理由。在简单的例子中，这没什么大不了的，但在更精巧的场景下，你可能发现这个限制。

通常的建议是构建一个包装值（比如 `object` 或 `array` ）来包含这些多个消息。这个方法好用，但是在你的Promise链的每一步上把消息包装再拆开显得十分尴尬和烦人。

分割值

有时你可以将这种情况当做一个信号，表示你可以/应当将问题拆分为两个或更多的Promise。

想象你有一个工具 `foo(..)` ，它异步地产生两个值（ `x` 和 `y` ）：

```
function getY(x) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            resolve( (3 * x) - 1 );
        }, 100 );
    } );
}

function foo(bar,baz) {
    var x = bar * baz;

    return getY( x )
    .then( function(y){
        // 将两个值包装近一个容器
        return [x,y];
```

```
    } );
}

foo( 10, 20 )
.then( function(msgs){
    var x = msgs[0];
    var y = msgs[1];

    console.log( x, y );    // 200 599
} );
```

首先，让我们重新安排一下 `foo(..)` 返回的东西，以便于我们不必再将 `x` 和 `y` 包装进一个单独的 `array` 值中来传送给一个Promise。相反，我们将每一个值包装进它自己的promise：

```
function foo(bar,baz) {
    var x = bar * baz;

    // 将两个promise返回
    return [
        Promise.resolve( x ),
        getY( x )
    ];
}

Promise.all(
    foo( 10, 20 )
)
.then( function(msgs){
    var x = msgs[0];
    var y = msgs[1];

    console.log( x, y );
} );
```

一个promise的 `array` 真的要比传递给一个单独的Promise的值的 `array` 要好吗？语法上，它没有太多改进。

但是这种方式更加接近于Promise的设计原理。现在它更易于在未来将 `x` 与 `y` 的计算分开，重构进两个分离的函数中。它更清晰，也允许调用端代码更灵活地安排这两个promise——这里使用了 `Promise.all([ .. ])` ，但它当然不是唯一的选择——而不是将这样的细节在 `foo(..)` 内部进行抽象。

展开/散开参数

`var x = ..` 和 `var y = ..` 的赋值依然是一个尴尬的负担。我们可以在一个帮助工具中利用一些函数式技巧（向Reginald Braithwaite致敬，在推特上 `@raganwald` ）：

```
function spread(fn) {
    return Function.apply.bind( fn, null );
}

Promise.all(
    foo( 10, 20 )
)
.then(
    spread( function(x,y){
        console.log( x, y );    // 200 599
    } )
)
```

看起来好些了！当然，你可以内联这个函数式魔法来避免额外的帮助函数：

```
Promise.all(
    foo( 10, 20 )
)
.then( Function.apply.bind(
    function(x,y){
        console.log( x, y );    // 200 599
    },
    null
) );
```

这个技巧可能很整洁，但是ES6给了我们一个更好的答案：解构（destructuring）。数组的解构赋值形式看起来像这样：

```
Promise.all(
    foo( 10, 20 )
)
.then( function(msgs){
    var [x,y] = msgs;

    console.log( x, y );    // 200 599
} );
```

最棒的是，ES6提供了数组参数解构形式：

```
Promise.all(
    foo( 10, 20 )
)
.then( function([x,y]){
    console.log( x, y );    // 200 599
} );
```

我们现在已经接受了“每个Promise一个值”的准则，继续让我们把模板代码最小化！

注意： 更多关于ES6解构形式的信息，参阅本系列的 *ES6*与未来。

单次解析

Promise的一个最固有的行为之一就是，一个Promise只能被解析一次（成功或拒绝）。对于多数异步用例来说，你仅仅取用这个值一次，所以这工作的很好。

但也有许多异步情况适用于一个不同的模型—更类似于事件和/或数据流。表面上看不清Promise能对这种用例适应的多好，如果能的话。没有基于Promise的重大抽象过程，它们完全缺乏对多个值解析的处理。

想象这样一个场景，你可能想要为响应一个刺激（比如事件）触发一系列异步处理步骤，而这实际上将会发生多次，比如按钮点击。

这可能不会像你想的那样工作：

```
// `click(..)` 绑定了一个DOM元素的 `"click"` 事件
// `request(..)` 是先前定义的支持Promise的Ajax

var p = new Promise( function(resolve,reject){
    click( "#mybtn", resolve );
} );

p.then( function(evt){
    var btnID = evt.currentTarget.id;
    return request( "http://some.url.1/?id=" + btnID );
} )
.then( function(text){
    console.log( text );
} );
```

这里的行为仅能在你的应用程序只让按钮被点击一次的情况下工作。如果按钮被点击第二次，promise `p` 已经被解析了，所以第二个 `resolve(..)` 将被忽略。

相反的，你可能需要将模式反过来，在每次事件触发时创建一个全新的Promise链：

```
click( "#mybtn", function(evt){
    var btnID = evt.currentTarget.id;

    request( "http://some.url.1/?id=" + btnID )
    .then( function(text){
        console.log( text );
    } );
```

```
} );
```

这种方式会 好用，为每个按钮上的 `"click"` 事件发起一个全新的Promise序列。

但是除了在事件处理器内部定义一整套Promise链看起来很丑以外，这样的设计在某种意义上违背了关注/能力分离原则（SoC）。你可能非常想在一个你的代码不同的地方定义事件处理器：你定义对事件的 响应（Promise链）的地方。如果没有帮助机制，在这种模式下这么做很尴尬。

注意： 这种限制的另一种表述方法是，如果我们能够构建某种能在它上面进行Promise链监听的“可监听对象（observable）”就好了。有一些库已经建立这些抽象（比如RxJS—— http://rxjs.codeplex.com/ ），但是这种抽象看起来是如此的重，以至于你甚至再也看不到Promise的性质。这样的重抽象带来一个重要的问题：这些机制是否像Promise本身被设计的一样可靠。我们将会在附录B中重新讨论“观察者（Observable）”模式。

惰性

对于在你的代码中使用Promise而言一个实在的壁垒是，现存的所有代码都没有支持Promise。如果你有许多基于回调的代码，让代码保持相同的风格容易多了。

“一段基于动作（用回调）的代码将仍然基于动作（用回调），除非一个更聪明，具有Promise意识的开发者对它采取行动。”

Promise提供了一种不同的模式规范，如此，代码的表达方式可能会变得有一点儿不同，某些情况下，则根本不同。你不得不有意这么做，因为Promise不仅只是把那些为你服务至今的老式编码方法自然地抖落掉。

考虑一个像这样的基于回调的场景：

```
function foo(x,y,cb) {
    ajax(
        "http://some.url.1/?x=" + x + "&y=" + y,
        cb
    );
}

foo( 11, 31, function(err,text) {
    if (err) {
        console.error( err );
    }
    else {
        console.log( text );
    }
} );
```

将这个基于回调的代码转换为支持Promise的代码的第一步该怎么做，是立即明确的吗？这要看你的

经验。你练习的越多，它就感觉越自然。但当然，Promise没有明确告知到底怎么做—没有一个放之四海而皆准的答案—所以这要靠你的责任心。

就像我们以前讲过的，我们绝对需要一种支持Promise的Ajax工具来取代基于回调的工具，我们可以称它为 `request(..)` 。你可以制造自己的，正如我们已经做过的。但是不得不为每个基于回调的工具手动定义Promise相关的包装器的负担，使得你根本就不太可能选择将代码重构为Promise相关的。

Promise没有为这种限制提供直接的答案。但是大多数Promise库确实提供了帮助函数。想象一个这样的帮助函数：

```
// 填补的安全检查
if (!Promise.wrap) {
    Promise.wrap = function(fn) {
        return function() {
            var args = [].slice.call( arguments );

            return new Promise( function(resolve,reject){
                fn.apply(
                    null,
                    args.concat( function(err,v){
                        if (err) {
                            reject( err );
                        }
                        else {
                            resolve( v );
                        }
                    } )
                );
            } );
        };
    };
}
```

好吧，这可不是一个微不足道的工具。然而，虽然他可能看起来有点儿令人生畏，但也没有你想的那么糟。它接收一个函数，这个函数期望一个错误优先风格的回调作为第一个参数，然后返回一个可以自动创建Promise并返回的新函数，然后为你替换掉回调，与Promise的完成/拒绝连接在一起。

与其浪费太多时间谈论这个 `Promise.wrap(..)` 帮助函数 如何 工作，还不如让我们来看看如何使用它：

```
var request = Promise.wrap( ajax );

request( "http://some.url.1/" )
.then( .. )
..
```

哇哦，真简单！

`Promise.wrap(..)` 不会 生产Promise。它生产一个将会生产Promise的函数。某种意义上，一个Promise生产函数可以被看做一个“Promise工厂”。我提议将这样的东西命名为“promisory”（”Promise” + “factory”）。

这种将期望回调的函数包装为一个Promise相关的函数的行为，有时被称为“提升（lifting）”或“promise化（promisifying）”。但是除了“提升过的函数”以外，看起来没有一个标准的名词来称呼这个结果函数，所以我更喜欢“promisory”，因为我认为他更具描述性。

注意： Promisory不是一个瞎编的词。它是一个真实存在的词汇，而且它的定义是含有或载有一个promise。这正是这些函数所做的，所以这个术语匹配得简直完美！

那么，`Promise.wrap(ajax)` 生产了一个我们称为 `request(..)` 的 `ajax(..)` promisory，而这个promisory为Ajax应答生产Promise。

如果所有的函数已经都是promisory，我们就不需要自己制造它们，所以额外的步骤就有点儿多余。但是至少包装模式是（通常都是）可重复的，所以我们可以把它放进 `Promise.wrap(..)` 帮助函数中来支援我们的promise编码。

那么回到刚才的例子，我们需要为 `ajax(..)` 和 `foo(..)` 都做一个promisory。

```
// 为`ajax(..)`制造一个promisory
var request = Promise.wrap( ajax );

// 重构`foo(..)`，但是为了代码其他部分
// 的兼容性暂且保持它对外是基于回调的
// —仅在内部使用`request(..)`'的promise
function foo(x,y,cb) {
    request(
        "http://some.url.1/?x=" + x + "&y=" + y
    )
    .then(
        function fulfilled(text){
            cb( null, text );
        },
        cb
    );
}

// 现在，为了这段代码本来的目的，为`foo(..)`制造一个promisory
var betterFoo = Promise.wrap( foo );

// 并使用这个promisory
betterFoo( 11, 31 )
.then(
    function fulfilled(text){
```

```
        console.log( text );
    },
    function rejected(err){
        console.error( err );
    }
);
```

当然，虽然我们将 `foo(..)` 重构为使用我们的新 `request(..)` promisory，我们可以将 `foo(..)` 本身制成promisory，而不是保留基于会掉的实现并需要制造和使用后续的 `betterFoo(..)` promisory。这个决定只是要看 `foo(..)` 是否需要保持基于回调的形式以便于代码的其他部分兼容。

考虑这段代码：

```
// 现在，`foo(..)`也是一个promisory
// 因为它委托到`request(..)` promisory
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

foo( 11, 31 )
.then( .. )
..
```

虽然ES6的Promise没有为这样的promisory包装提供原生的帮助函数，但是大多数库提供它们，或者你可以制造自己的。不管哪种方法，这种Promise特定的限制是可以不费太多劲儿就可以解决的（当然是和回调地狱的痛苦相比！）。

Promise不可撤销

一旦你创建了一个Promise并给它注册了一个完成和/或拒绝处理器，就没有什么你可以从外部做的事情能停止这个进程，即使是某些其他的事情使这个任务变得毫无意义。

注意： 许多Promise抽象库都提供取消Promise的功能，但这是一个非常坏的主意！许多开发者都希望Promise被原生地设计为具有外部取消能力，但问题是这将允许Promise的一个消费者/监听器影响某些其他消费者监听同一个Promise的能力。这违反了未来值得可靠性原则（外部不可变），另外就是嵌入了“远距离行为（action at a distance）”的反模式（http://en.wikipedia.org/wiki/Action_at_a_distance_%28computer_programming%29）。不管它看起来多么有用，它实际上会直接将你引回与回调地狱相同的噩梦。

考虑我们早先的Promise超时场景：

```
var p = foo( 42 );

Promise.race( [
    p,
    timeoutPromise( 3000 )
] )
.then(
    doSomething,
    handleError
);

p.then( function(){
    // 即使是在超时的情况下也会发生 :(
} );
```

“超时”对于promise `p` 来说是外部的，所以 `p` 本身继续运行，这可能不是我们想要的。

一个选项是侵入性地定义你的解析回调：

```
var OK = true;

var p = foo( 42 );

Promise.race( [
    p,
    timeoutPromise( 3000 )
    .catch( function(err){
        OK = false;
        throw err;
    } )
] )
.then(
    doSomething,
    handleError
);

p.then( function(){
    if (OK) {
        // 仅在没有超时的情况下发生！ :)
    }
} );
```

很难看。这可以工作，但是远不理想。一般来说，你应当避免这样的场景。

但是如果你不能，这种解决方案的丑陋应当是一个线索，说明 取消 是一种属于在Promise之上的更高层抽象的功能。我推荐你找一个Promise抽象库来辅助你，而不是自己使用黑科技。

注意： 我的 *asynquence* Promise抽象库提供了这样的抽象，还为序列提供了一个 `abort()` 能力，这一切将在附录A中讨论。

一个单独的Promise不是真正的流程控制机制（至少没有多大实际意义），而流程控制机制正是 取消 要表达的；这就是为什么Promise取消显得尴尬。

相比之下，一个链条的Promise集合在一起—我称之为“序列”— 是 一个流程控制的表达，如此在这一层面的抽象上它就适于定义取消。

没有一个单独的Promise应该是可以取消的，但是一个 序列 可以取消是有道理的，因为你不会将一个序列作为一个不可变值传来传去，就像Promise那样。

Promise性能

这种限制既简单又复杂。

比较一下在基于回调的异步任务链和Promise链上有多少东西在动，很明显Promise有多得多的事情发生，这意味着它们自然地会更慢一点点。回想一下Promise提供的保证信任的简单列表，将它和你为了达到相同保护效果而在回调上面添加的特殊代码比较一下。

更多工作要做，更多的安全要保护，意味着Promise与赤裸裸的，不可靠的回调相比 确实 更慢。这些都很明显，可能很容易萦绕在你脑海中。

但是慢多少？好吧……这实际上是一个难到不可思议的问题，无法绝对，全面地回答。

坦白地说，这是一个比较苹果和橘子的问题，所以可能是问错了。你实际上应当比较的是，带有所有手动保护层的经过特殊处理的回调系统，是否比一个Promise实现要快。

如果说Promise有一种合理的性能限制，那就是它并不将可靠性保护的选项罗列出来让你选择—你总是一下得到全部。

如果我们承认Promise一般来说要比它的非Promise，不可靠的回调等价物 慢一点儿—假定在有些地方你觉得你可以自己调整可靠性的缺失—难道这意味着Promise应当被全面地避免，就好像你的整个应用程序仅仅由一些可能的“必须绝对最快”的代码驱动着？

扪心自问：如果你的代码有那么合理，那么 对于这样的任务，**JavaScript**是正确的选择吗？ 为了运行应用程序JavaScript可以被优化得十分高效（参见第五章和第六章）。但是在Promise提供的所有好处的光辉之下，过于沉迷它微小的性能权衡，真的 合适吗？

另一个微妙的问题是Promise使 所有事情 都成为异步的，这意味着有些应当立即完成的（同步的）步骤也要推迟到下一个Job步骤中（参见第一章）。也就是说一个Promise任务序列要比使用回调连接的相同序列要完成的稍微慢一些是可能的。

当然，这里的问题是：这些关于性能的微小零头的潜在疏忽，和我们在本章通篇阐述的Promise带来的益处相比，还值得考虑吗？

我的观点是，在几乎所有你可能认为Promise的性能慢到了需要被考虑的情况下，完全回避Promise并将它的可靠性和组合性优化掉，实际上是一种反模式。

相反地，你应当默认地在代码中广泛使用它们，然后再记录并分析你的应用程序的热（关键）路径。Promise 真的 是瓶颈？还是它们只是理论上慢了下来？只有在那 之后，拿着实际合法的基准分析观测数据（参见第六章），再将Promise从这些关键区域中重构移除才称得上是合理与谨慎。

Promise是有一点儿慢，但作为交换你得到了很多内建的可靠性，无Zalgo的可预测性，与组合性。也许真正的限制不是它们的性能，而是你对它们的益处缺乏认识？

复习

复习

Promise很牛。用它们。它们解决了肆虐在回调代码中的 控制倒转 问题。

它们没有摆脱回调，而是重新定向了这些回调的组织安排方式，是它成为一种坐落于我们和其他工具之间的可靠的中间机制。

Promise链还开始以顺序的风格定义了一种更好的（当然，还不完美）表达异步流程的方式，它帮我们的大脑更好的规划和维护异步JS代码。我们会在下一章中看到一个更好的解决 这个 问题的方法！

第四章: Generator

第四章： Generator

在第二章中，我们发现了在使用回调表达异步流程控制时的两个关键缺陷：

- 基于回调的异步与我们的大脑规划任务的各个步骤的过程不相符。
- 由于 控制倒转 回调是不可靠的，也是不可组合的。

在第三章中，我们详细地讨论了Promise如何反转回调的 控制倒转，重建了可靠性/可组合性。

现在让我们把注意力集中到用一种顺序的，看起来同步的风格来表达异步流程控制。使这一切成为可能的“魔法”是ES6的 **generator**。

链接

打破运行至完成

打破运行至完成

在第一章中，我们讲解了一个JS开发者们在他们的代码中几乎永恒依仗的一个认识：一旦函数开始执行，它将运行直至完成，没有其他的代码可以在运行期间干扰它。

这看起来可能很滑稽，ES6引入了一种新型的函数，它不按照“运行至完成”的行为进行动作。这种新型的函数称为“generator（生成器）”。

为了理解它的含义，让我们看看这个例子：

```
var x = 1;

function foo() {
    x++;
    bar();                  // <-- 这一行会发生什么？
    console.log( "x:", x );
}

function bar() {
    x++;
}

foo();                      // x: 3
```

在这个例子中，我们确信 `bar()` 会在 `x++` 和 `console.log(x)` 之间运行。但如果 `bar()` 不在这里呢？很明显结果将是 `2` 而不是 `3`。

现在让我们来燃烧你的大脑。要是 `bar()` 不存在，但以某种方式依然可以在 `x++` 和 `console.log(x)` 语句之间运行呢？这可能吗？

在 抢占式（**preemptive**） 多线程语言中， `bar()` 去“干扰”并正好在两个语句之间那一时刻运行，实质上时可能的。但JS不是抢占式的，也（还）不是多线程的。但是，如果 `foo()` 本身可以用某种办法在代码的这一部分指示一个“暂停”，那么这种“干扰”（并发）的 协作 形式就是可能的。

注意： 我使用“协作”这个词，不仅是因为它与经典的并发术语有关联（见第一章），也因为正如你将在下一个代码段中看到的，ES6在代码中指示暂停点的语法是 `yield` ——暗示一个让出控制权的礼貌的 协作。

这就是实现这种协作并发的ES6代码：

```
var x = 1;

function *foo() {
    x++;
    yield; // 暂停！
    console.log( "x:", x );
}

function bar() {
    x++;
}
```

注意： 你将很可能在大多数其他的JS文档/代码中看到，一个generator的声明被格式化为 `function* foo() { .. }` 而不是我在这里使用的 `function *foo() { .. }` ——唯一的区别是摆放 `*` 位置的风格。这两种形式在功能性/语法上是完全一样的，还有第三种 `function*foo() { .. }` （没空格）形式。这两种风格存在争议，但我基本上偏好 `function *foo..` ，因为当我在写作中用 `*foo()` 引用一个generator时，这种形式可以匹配我写的东西。如果我只说 `foo()` ，你就不会清楚地知道我是在说一个generator还是一个一般的函数。这纯粹是一个风格偏好的问题。

现在，我们该如何运行上面的代码，使 `bar()` 在 `yield` 那一点取代 `*foo()` 的执行？

```
// 构建一个迭代器`it`来控制generator
var it = foo();

// 在这里开始`foo()`！
it.next();
x;                          // 2
bar();
x;                          // 3
it.next();                  // x: 3
```

好了，这两段代码中有不少新的，可能使人困惑的东西，所以我们得跋涉好一段了。在我们用ES6的generator来讲解不同的机制/语法之前，让我们过一遍这个行为的流程：

1. `it = foo()` 操作 不会 执行 `*foo()` generator，它只不过构建了一个用来控制它执行的 迭代器（*iterator*）。我们一会更多地讨论 迭代器。
2. 第一个 `it.next()` 启动了 `*foo()` generator，并且运行 `*foo()` 第一行上的 `x++` 。
3. `*foo()` 在 `yield` 语句处暂停，就在这时第一个 `it.next()` 调用结束。在这个时刻，`*foo()` 依然运行而且是活动的，但是处于暂停状态。

4. 我们观察 `x` 的值，现在它是 `2`.
5. 我们调用 `bar()`，它再一次用 `x++` 递增 `x`。
6. 我们再一次观察 `x` 的值，现在它是 `3`。
7. 最后的 `it.next()` 调用使 `*foo()` generator从它暂停的地方继续运行，而后运行使用 `x` 的当前值 `3` 的 `console.log(..)` 语句。

清楚的是， `*foo()` 启动了，但 没有 运行到底—它停在 `yield` 。我们稍后继续 `*foo()` ，让它完成，但这甚至不是必须的。

所以，一个generator是一种函数，它可以开始和停止一次或多次，甚至没必要一定要完成。虽然为什么它很强大看起来不那么明显，但正如我们将要在本章剩下的部分将要讲到的，它是我们用于在我们的代码中构建“generator异步流程控制”模式的基础构建块儿之一。

输入和输出

一个generator函数是一种带有我们刚才提到的新型处理模型的函数。但它仍然是一个函数，这意味着依旧有一些不变的基本原则—即，它依然接收参数（也就是“输入”），而且它依然返回一个值（也就是“输出”）：

```
function *foo(x,y) {
    return x * y;
}

var it = foo( 6, 7 );

var res = it.next();

res.value;        // 42
```

我们将 `6` 和 `7` 分别作为参数 `x` 和 `y` 传递给 `*foo(..)` 。而 `*foo(..)` 将值 `42` 返回给调用端代码。

现在我们可以看到发生器的调用和一般函数的调用的一个不同之处了。 `foo(6,7)` 显然看起来很熟悉。但微妙的是， `*foo(..)` generator不会像一个函数那样实际运行起来。

相反，我们只是创建了 迭代器 对象，将它赋值给变量 `it` ，来控制 `*foo(..)` generator。当我们调用 `it.next()` 时，它指示 `*foo(..)` generator从现在的位置向前推进，直到下一个 `yield` 或者generator的最后。

`next(..)` 调用的结果是一个带有 `value` 属性的对象，它持有从 `*foo(..)` 返回的任何值（如果有的话）。换句话说， `yield` 导致在generator运行期间，一个值被从中发送出来，有点儿像一个中间的 `return` 。

但是，为什么我们需要这个完全间接的 迭代器 对象来控制generator还不清楚。我们回头会讨论它

的，我保证。

迭代通信

generator除了接收参数和拥有返回值，它们还内建有更强大，更吸引人的输入/输出消息能力，这是通过使用 `yield` 和 `next(..)` 实现的。

考虑下面的代码：

```
1. function *foo(x) {
2.     var y = x * (yield);
3.     return y;
4. }
5.
6. var it = foo( 6 );
7.
8. // 开始`foo(..)`
9. it.next();
10.
11. var res = it.next( 7 );
12.
13. res.value;        // 42
```

首先，我们将 `6` 作为参数 `x` 传入。之后我们调用 `it.next()` ，它启动了 `*foo(..)` .

在 `*foo(..)` 内部， `var y = x ..` 语句开始被处理，但它运行到了一个 `yield` 表达式。就在这时，它暂停了 `*foo(..)` （就在赋值语句的中间！），而且请求调用端代码为 `yield` 表达式提供一个结果值。接下来，我们调用 `it.next(7)` ，将 `7` 这个值传回去作为暂停的 `yield` 表达式的结果。

所以，在这个时候，赋值语句实质上是 `var y = 6 * 7` 。现在， `return y` 将值 `42` 作为结果返回给 `it.next( 7 )` 调用。

注意一个非常重要，而且即便是对于老练的JS开发者也非常容易犯糊涂的事情：根据你的角度，在 `yield` 和 `next(..)` 调用之间存在着错位。一般来说，你所拥有的 `next(..)` 调用的数量，会比你所拥有的 `yield` 语句的数量多一个——前面的代码段中有一个 `yield` 和两个 `next(..)` 调用。

为什么会有这样的错位？

因为第一个 `next(..)` 总是启动一个generator，然后运行至第一个 `yield` 。但是第二个 `next(..)` 调用满足了第一个暂停的 `yield` 表达式，而第三个 `next(..)` 将满足第二个 `yield` ，如此反复。

两个疑问的故事

实际上，你主要考虑的是哪部分代码会影响你是否感知到错位。

仅考虑generator代码：

```
var y = x * (yield);
return y;
```

这 第一个 `yield` 基本上是在 问一个问题：“我应该在这里插入什么值？”

谁来回答这个问题？好吧，第一个 `next()` 在这个时候已经为了启动generator而运行过了，所以很明显 它 不能回答这个问题。所以，第二个 `next(..)` 调用必须回答由 第一个 `yield` 提出的问题。

看到错位了吧—第二个对第一个？

但是让我们反转一下我们的角度。让我们不从generator的角度看问题，而从迭代器的角度看。

为了恰当地描述这种角度，我们还需要解释一下，消息可以双向发送— `yield ..` 作为表达式可以发送消息来应答 `next(..)` 调用，而 `next(..)` 可以发送值给暂停的 `yield` 表达式。考虑一下这段稍稍调整过的代码：

```
function *foo(x) {
    var y = x * (yield "Hello");    // <-- 让出一个值！
    return y;
}

var it = foo( 6 );

var res = it.next();    // 第一个`next()`，不传递任何东西
res.value;              // "Hello"

res = it.next( 7 );       // 传递`7`给等待中的`yield`
res.value;              // 42
```

`yield ..` 和 `next(..)` 一起成对地 在**generator**运行期间 构成了一个双向消息传递系统。

那么，如果只看 迭代器 代码：

```
var res = it.next();    // 第一个`next()`，不传递任何东西
res.value;              // "Hello"

res = it.next( 7 );       // 传递`7`给等待中的`yield`
res.value;              // 42
```

注意： 我们没有传递任何值给第一个 `next()` 调用，而且是故意的。只有一个暂停的 `yield` 才能接收这样一个被 `next(..)` 传递的值，但是当我们调用第一个 `next()` 时，在generator的最开始并 没有任何暂停的 **`yield`** 可以接收这样的值。语言规范和所有兼容此语言规范的浏览器只会无声

地 丢弃 任何传入第一个 `next()` 的东西。传递这样的值是一个坏主意，因为你只不过创建了一些令人困惑的无声“失败”的代码。所以，记得总是用一个无参数的 `next()` 来启动generator。

第一个 `next()` 调用（没有任何参数的）基本上是在 问一个问题：“ `*foo(..)` generator将要给我的 下一个 值是什么？”，谁来回答这个问题？第一个 `yield` 表达式。

看到了？这里没有错位。

根据你认为是 谁 在问问题，在 `yield` 和 `next(..)` 之间的错位既存在又不存在。

但等一下！跟 `yield` 语句的数量比起来，还有一个额外的 `next()` 。那么，这个最后的 `it.next(7)` 调用又一次在询问generator 下一个 产生的值是什么。但是没有 `yield` 语句剩下可以回答了，不是吗？那么谁来回答？

`return` 语句回答这个问题！

而且如果在你的generator中 没有 **`return`** —比起一般的函数，generator中的 `return` 当然不再是必须的—总会有一个假定/隐式的 `return;` （也就是 `return undefined;` ），它默认的目的就是回答由最后的 `it.next(7)` 调用 提出 的问题。

这些问题与回答—用 `yield` 和 `next(..)` 进行双向消息传递—十分强大，但还是看不出来这些机制与异步流程控制有什么联系。我们正在接近真相！

多迭代器

从语法使用上来看，当你用一个 迭代器 来控制generator时，你正在控制声明的generator函数本身。但这里有一个容易忽视的微妙细节：每当你构建一个 迭代器，你都隐含地构建了一个将由这个迭代器 控制的generator的实例。

你可以让同一个generator的多个实例同时运行，它们甚至可以互动：

```
1.  function *foo() {
2.      var x = yield 2;
3.      z++;
4.      var y = yield (x * z);
5.      console.log( x, y, z );
6.  }
7.
8.  var z = 1;
9.
10. var it1 = foo();
11. var it2 = foo();
12.
13. var val1 = it1.next().value;           // 2 <-- 让出2
14. var val2 = it2.next().value;           // 2 <-- 让出2
15.
```

```
val1 = it1.next( val2 * 10 ).value;        // 40  <-- x:20,  z:2
val2 = it2.next( val1 * 5 ).value;        // 600 <-- x:200, z:3

it1.next( val2 / 2 );                    // y:300
                                        // 20 300 3
it2.next( val1 / 4 );                    // y:10
                                        // 200 10 3
```

警告： 同一个generator的多个并发运行实例的最常见的用法，不是这样的互动，而是generator在没有输入的情况下，从一些连接着的独立资源中产生它自己的值。我们将在下一节中更多地讨论产生值。

让我们简单地走一遍这个处理过程：

1. 两个 `*foo()` 在同时启动，而且两个 `next()` 都分别从 `yield 2` 语句中得到了 `2` 的 `value` 。
2. `val2 * 10` 就是 `2 * 10` ，它被发送到第一个generator实例 `it1` ，所以 `x` 得到值 `20` 。 `z` 将 `1` 递增至 `2` ，然后 `20 * 2` 被 `yield` 出来，将 `val1` 设置为 `40` 。
3. `val1 * 5` 就是 `40 * 5` ，它被发送到第二个generator实例 `it2` 中，所以 `x` 得到值 `200` 。 `z` 又一次递增，从 `2` 到 `3` ，然后 `200 * 3` 被 `yield` 出来，将 `val2` 设置为 `600` 。
4. `val2 / 2` 就是 `600 / 2` ，它被发送到第一个generator实例 `it1` ，所以 `y` 得到值 `300` ，然后分别为它的 `x y z` 值打印出 `20 300 3` 。
5. `val1 / 4` 就是 `40 / 4` ，它被发送到第一个generator实例 `it2` ，所以 `y` 得到值 `10` ，然后分别为它的 `x y z` 值打印出 `200 10 3` 。

这是在你脑海中跑过的一个“有趣”的例子。你还能保持清醒？

穿插

回想第一章中“运行至完成”一节的这个场景：

```
var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}
```

使用普通的JS函数，当然要么是 `foo()` 可以首先运行完成，要么是 `bar()` 可以首先运行至完成，但是 `foo()` 不可能与 `bar()` 穿插它的独立语句。所以，前面这段代码只有两个可能的结果。

然而，使用generator，明确地穿插（甚至是在语句中间！）是可能的：

```
var a = 1;
var b = 2;

function *foo() {
    a++;
    yield;
    b = b * a;
    a = (yield b) + 3;
}

function *bar() {
    b--;
    yield;
    a = (yield 8) + b;
    b = a * (yield 2);
}
```

根据 迭代器 控制 `*foo()` 与 `*bar()` 分别以什么样的顺序被调用，前面这段代码可以产生几种不同的结果。换句话说，通过两个generator在同一个共享的变量上穿插，我们实际上可以展示（以一种模拟的方式）在第一章中讨论的，理论上的“线程的竞合状态”环境。

首先，让我们制造一个称为 `step(..)` 的帮助函数，让它控制 迭代器：

```
function step(gen) {
    var it = gen();
    var last;

    return function() {
        // 不论`yield`出什么，只管在下一次时直接把它塞回去！
        last = it.next( last ).value;
    };
}
```

`step(..)` 初始化一个generator来创建它的 `it` 迭代器，然后它返回一个函数，每次这个函数被调用时，都将 迭代器 向前推一步。另外，前一个被 `yield` 出来的值将被直接发给下一步。所以，`yield 8` 将变成 `8` 而 `yield b` 将成为 `b`（不管它在 `yield` 时是什么值）。

现在，为了好玩儿，让我们做一些实验，来看看将这些 `*foo()` 与 `*bar()` 的不同块儿穿插时的效果。我们从一个无聊的基本情况开始，保证 `*foo()` 在 `*bar()` 之前全部完成（就像我们在第一章中做的那样）：

```
// 确保重置了`a`和`b`
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

// 首先完全运行`*foo()`
s1();
s1();
s1();

// 现在运行`*bar()`
s2();
s2();
s2();
s2();

console.log( a, b );    // 11 22
```

最终结果是 `11` 和 `22`，就像第一章的版本那样。现在让我们把顺序混合穿插，来看看它如何改变 `a` 与 `b` 的值。

```
// 确保重置了`a`和`b`
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

s2();        // b--;
s2();        // 让出 8
s1();        // a++;
s2();        // a = 8 + b;
            // 让出 2
s1();        // b = b * a;
            // 让出 b
s1();        // a = b + 3;
s2();        // b = a * 2;
```

在我告诉你结果之前，你能指出在前面的程序运行之后 `a` 和 `b` 的值是什么吗？不要作弊！

```
console.log( a, b );    // 12 18
```

注意： 作为留给读者的练习，试试通过重新安排 `s1()` 和 `s2()` 调用的顺序，看看你能得到多少种

结果组合。别忘了你总是需要三个 `s1()` 调用和四个 `s2()` 调用。至于为什么，回想一下刚才关于使用 `yield` 匹配 `next()` 的讨论。

当然，你几乎不会想有意制造 这种 水平的，令人糊涂的穿插，因为他创建了非常难理解的代码。但是这个练习很有趣，而且对于理解多个generator如何并发地运行在相同的共享作用域来说很有教育意义，因为会有一些地方这种能力十分有用。

我们会在本章末尾更详细地讨论generator并发。

生成值

生成值

在前一节中，我们提到了一个generator的有趣用法，作为一种生产值的方式。这 不是 我们本章主要关注的，但如果我们不在这里讲一下基本我们会想念它的，特别是因为这种用法实质上是它的名称的由来：生成器。

我们将要稍稍深入一下 迭代器 的话题，但我们会绕回到它们如何与generator关联，并使用generator来 生成 值。

发生器与迭代器

想象你正在生产一系列的值，它们中的每一个都与前一个值有可定义的关系。为此，你将需要一个有状态的发生器来记住上一个给出的值。

你可以用函数闭包（参加本系列的 作用域与闭包）来直接地实现这样的东西：

```
var gimmeSomething = (function(){
    var nextVal;

    return function(){
        if (nextVal === undefined) {
            nextVal = 1;
        }
        else {
            nextVal = (3 * nextVal) + 6;
        }

        return nextVal;
    };
})();

gimmeSomething();        // 1
gimmeSomething();        // 9
gimmeSomething();        // 33
gimmeSomething();        // 105
```

注意： 这里 `nextVal` 的计算逻辑已经被简化了，但从概念上讲，直到 下一次 `gimmeSomething()` 调用发生之前，我们不想计算 下一个值（也就是 `nextVal` ），因为一般对于持久性更强的，或者比简单的 `number` 更有限的资源的发生器来说，那可能是一种资源泄漏的设计。

生成随意的数字序列不是是一个很真实的例子。但是如果你从一个数据源中生成记录呢？你可以想象很多相同的代码。

事实上，这种任务是一种非常常见的设计模式，通常用迭代器解决。一个 迭代器 是一个明确定义的接口，用来逐个通过一系列从发生器得到的值。迭代器的JS接口，和大多数语言一样，是在你每次想从发生器中得到下一个值时调用的 `next()` 。

我们可以为我们的数字序列发生器实现标准的 迭代器；

```
var something = (function(){
    var nextVal;

    return {
        // `for..of`循环需要这个
        [Symbol.iterator]: function(){ return this; },

        // 标准的迭代器接口方法
        next: function(){
            if (nextVal === undefined) {
                nextVal = 1;
            }
            else {
                nextVal = (3 * nextVal) + 6;
            }

            return { done:false, value:nextVal };
        }
    };
})();

something.next().value;        // 1
something.next().value;        // 9
something.next().value;        // 33
something.next().value;        // 105
```

注意： 我们将在“Iterables”一节中讲解为什么我们在这个代码段中需要 `[Symbol.iterator]: ..` 这一部分。在语法上讲，两个ES6特性在发挥作用。首先， `[ .. ]` 语法称为一个 计算型属性名（参见本系列的 *this*与对象原型）。它是一种字面对象定义方法，用来指定一个表达式并使用这个表达式的结果作为属性名。另一个， `Symbol.iterator` 是ES6预定义的特殊 `Symbol` 值。

`next()` 调用返回一个对象，它带有两个属性： `done` 是一个 `boolean` 值表示 迭代器 的完成状

态； `value` 持有迭代的值。

ES6还增加了 `for..of` 循环，它意味着一个标准的 迭代器 可以使用原生的循环语法来自动地被消费：

```
for (var v of something) {
    console.log( v );

    // 不要让循环永无休止！
    if (v > 500) {
        break;
    }
}
// 1 9 33 105 321 969
```

注意： 因为我们的 `something` 迭代器总是返回 `done:false` ，这个 `for..of` 循环将会永远运行，这就是为什么我们条件性地放进一个 `break` 。对于迭代器来说永不终结是完全没有问题的，但是也有一些情况 迭代器 将运行在有限的值的集合上，而最终返回 `done:true` 。

`for..of` 循环为每一次迭代自动调用 `next()` —他不会给 `next()` 传入任何值—而且他将会在收到一个 `done:true` 时自动终结。这对于在一个集合的数据中进行循环十分方便。

当然，你可以手动循环一个迭代器，调用 `next()` 并检查 `done:true` 条件来知道什么时候停止：

```
for (
    var ret;
    (ret = something.next()) && !ret.done;
) {
    console.log( ret.value );

    // 不要让循环永无休止！
    if (ret.value > 500) {
        break;
    }
}
// 1 9 33 105 321 969
```

注意： 这种手动的 `for` 方式当然要比ES6的 `for..of` 循环语法难看，但它的好处是它提供给你一个机会，在有必要时传值给 `next(..)` 调用。

除了制造你自己的 迭代器 之外，许多JS中（就ES6来说）内建的数据结构，比如 `array` ，也有默认的 迭代器：

```
var a = [1,3,5,7,9];

```

```
for (var v of a) {
    console.log( v );
}
// 1 3 5 7 9
```

`for..of` 循环向 `a` 要来它的迭代器，并自动使用它迭代 `a` 的值。

注意： 看起来像是一个ES6的奇怪省略，普通的 `object` 有意地不带有像 `array` 那样的默认 迭代器。原因比我们要在这里讲的深刻得多。如果你想要的只是迭代一个对象的属性（不特别保证顺序），`Object.keys(..)` 返回一个 `array`，它可以像 `for (var k of Object.keys(obj)) { ..` 这样使用。像这样用 `for..of` 循环一个对象上的键，与用 `for..in` 循环内很相似，除了在 `for..in` 中会包含 `[[Prototype]]` 链的属性，而 `Object.keys(..)` 不会（参见本系列的 *this*与对象原型）。

Iterables

在我们运行的例子中的 `something` 对象被称为一个 迭代器，因为它的接口中有 `next()` 方法。但一个紧密关联的术语是 *iterable*，它指 包含有 一个可以迭代它所有值的迭代器的对象。

在ES6中，从一个 *iterable* 中取得一个 迭代器 的方法是，*iterable* 上必须有一个函数，它的名称是特殊的ES6符号值 `Symbol.iterator`。当这个函数被调用时，它就会返回一个 迭代器。虽然不是必须的，但一般来说每次调用应当返回一个全新的 迭代器。

前一个代码段的 `a` 就是一个 *iterable*。 `for..of` 循环自动地调用它的 `Symbol.iterator` 函数来构建一个 迭代器。我们当然可以手动地调用这个函数，然后使用它返回的 *iterator*：

```
var a = [1,3,5,7,9];

var it = a[Symbol.iterator]();

it.next().value;    // 1
it.next().value;    // 3
it.next().value;    // 5
..
```

在前面定义 `something` 的代码段中，你可能已经注意到了这一行：

```
[Symbol.iterator]: function(){ return this; }
```

这段有点让人困惑的代码制造了 `something` 值—— `something` 迭代器 的接口——也是一个 *iterable*；现在它既是一个 *iterable* 也是一个 迭代器。然后，我们把 `something` 传递给 `for..of` 循环：

```
for (var v of something) {
    ..
```

```
3. }
```

`for..of` 循环期待 `something` 是一个 *`iterable`*，所以它会寻找并调用它的 `Symbol.iterator` 函数。我们将这个函数定义为简单地 `return this` ，所以它将自己给出，而 `for..of` 不会知道这些。

Generator迭代器

带着 迭代器 的背景知识，让我们把注意力移回generator。一个generator可以被看做一个值的发生器，我们通过一个 迭代器 接口的 `next()` 调用每次从中抽取一个值。

所以，一个generator本身在技术上讲并不是一个 *`iterable`*，虽然很相似—当你执行generator时，你就得到一个 迭代器：

```
1. function *foo(){ .. }
2.
3. var it = foo();
```

我们可以用generator实现早前的 `something` 无限数字序列发生器，就像这样：

```
1. function *something() {
2.     var nextVal;
3.
4.     while (true) {
5.         if (nextVal === undefined) {
6.             nextVal = 1;
7.         }
8.         else {
9.             nextVal = (3 * nextVal) + 6;
10.        }
11.
12.        yield nextVal;
13.    }
14. }
```

注意： 在一个真实的JS程序中含有一个 `while..true` 循环通常是一件非常不好的事情，至少如果它没有一个 `break` 或 `return` 语句，那么它就很可能永远运行，并同步地，阻塞/锁定浏览器UI。然而，在generator中，如果这样的循环含有一个 `yield` ，那它就是完全没有问题的，因为generator将在每次迭代后暂停， `yield` 回主程序和/或事件轮询队列。说的明白点儿，“generator把 `while..true` 带回到JS编程中了！”

这变得相当干净和简单点儿了，对吧？因为generator会暂停在每个 `yield` ， `*something()` 函数的状态（作用域）被保持着，这意味着没有必要用闭包的模板代码来跨调用保留变量的状态了。

不仅是更简单的代码—我们不必自己制造 迭代器 接口了—它实际上是更合理的代码，因为它更清晰

地表达了意图。比如， `while..true` 循环告诉我们这个generator将要永远运行—只要我们一直向它请求，它就一直 产生 值。

现在我们可以在 `for..of` 循环中使用新得发亮的 `*something()` generator了，而且你会看到它工作起来基本一模一样：

```
for (var v of something()) {
    console.log( v );

    // 不要让循环永无休止！
    if (v > 500) {
        break;
    }
}
// 1 9 33 105 321 969
```

不要跳过 `for (var v of something()) ..` ！我们不仅仅像之前的例子那样将 `something` 作为一个值引用了，而是调用 `*something()` generator来得到它的 迭代器，并交给 `for..of` 使用。

如果你仔细观察，在这个generator和循环的互动中，你可能会有两个疑问：

- 为什么我们不能说 `for (var v of something) ..` ？因为这个 `something` 是一个generator，而不是一个 *iterable*。我们不得不调用 `something()` 来构建一个发生器给 `for..of` ，以便它可以迭代。
- `something()` 调用创建一个 迭代器，但是 `for..of` 想要一个 *iterable*，对吧？对，generator的 迭代器 上也有一个 `Symbol.iterator` 函数，这个函数基本上就是 `return this` ，就像我们刚才定义的 `something` *iterable*。换句话说generator的 迭代器 也是一个 *iterable* ！

停止Generator

在前一个例子中，看起来在循环的 `break` 被调用后， `*something()` generator的 迭代器 实例基本上被留在了一个永远挂起的状态。

但是这里有一个隐藏的行为为你处理这件事。 `for..of` 循环的“异常完成”（“提前终结”等等）—一般是由 `break` ， `return` ，或未捕捉的异常导致的—会向generator的 迭代器 发送一个信号，以使它终结。

注意： 技术上讲， `for..of` 循环也会在循环正常完成时向 迭代器 发送这个信号。对于generator来说，这实质上是一个无实际意义的操作，因为generator的 迭代器 要首先完成， `for..of` 循环才能完成。然而，自定义的 迭代器 可能会希望从 `for..of` 循环的消费者那里得到另外的信号。

虽然一个 `for..of` 循环将会自动发送这种信号，你可能会希望手动发送信号给一个 迭代器；你可以通过调用 `return(..)` 来这么做。

如果你在generator内部指定一个 `try..finally` 从句，它将总是被执行，即便是generator从外部被完成。这在你需要进行资源清理时很有用（数据库连接等）：

```
function *something() {
    try {
        var nextVal;

        while (true) {
            if (nextVal === undefined) {
                nextVal = 1;
            }
            else {
                nextVal = (3 * nextVal) + 6;
            }

            yield nextVal;
        }
    }
    // 清理用的从句
    finally {
        console.log( "cleaning up!" );
    }
}
```

前面那个在 `for..of` 中带有 `break` 的例子将会触发 `finally` 从句。但是你可以用 `return(..)` 从外部来手动终结generator的 迭代器 实例：

```
var it = something();
for (var v of it) {
    console.log( v );

    // 不要让循环永无休止！
    if (v > 500) {
        console.log(
            // 使generator得迭代器完成
            it.return( "Hello World" ).value
        );
        // 这里不需要`break`
    }
}
// 1 9 33 105 321 969
// cleaning up!
// Hello World
```

当我们调用 `it.return(..)` 时，它会立即终结generator，从而运行 `finally` 从句。而且，它会将返回的 `value` 设置为你传入 `return(..)` 的任何东西，这就是 `Hellow World` 如何立即返回来的。

我们现在也不必再包含一个 `break` ，因为generator的 迭代器 会被设置为 `done:true` ，所以 `for..of` 循环会在下一次迭代时终结。

generator的命名大部分源自于这种 消费生产的值 的用法。但要重申的是，这只是generator的用法之一，而且坦白的说，在这本书的背景下这甚至不是我们主要关注的。

但是现在我们更加全面地了解它们的机制是如何工作的，我们接下来可以将注意力转向generator如何实施于异步并发。

异步地迭代 Generator

异步地迭代Generator

generator要怎样处理异步编码模式，解决回调和类似的问题？让我们开始回答这个重要的问题。

我们应当重温一下第三章的一个场景。回想一下这个回调方式：

```
function foo(x,y,cb) {
    ajax(
        "http://some.url.1/?x=" + x + "&y=" + y,
        cb
    );
}

foo( 11, 31, function(err,text) {
    if (err) {
        console.error( err );
    }
    else {
        console.log( text );
    }
} );
```

如果我们想用generator表示相同的任务流控制，我们可以：

```
function foo(x,y) {
    ajax(
        "http://some.url.1/?x=" + x + "&y=" + y,
        function(err,data){
            if (err) {
                // 向`*main()`中扔进一个错误
                it.throw( err );
            }
            else {
                // 使用收到的`data`来继续`*main()`
                it.next( data );
            }
        }
    );
}
```

```
16.
17. function *main() {
18.     try {
19.         var text = yield foo( 11, 31 );
20.         console.log( text );
21.     }
22.     catch (err) {
23.         console.error( err );
24.     }
25. }
26.
27. var it = main();
28.
29. // 使一切开始运行！
30. it.next();
```

一眼看上去，这个代码段要比以前的回调代码更长，而且也许看起来更复杂。但不要让这种印象误导你。generator的代码段实际上要好 太多 了！但是这里有很多我们需要讲解的。

首先，让我们看看代码的这一部分，也是最重要的部分：

```
1. var text = yield foo( 11, 31 );
2. console.log( text );
```

花一点时间考虑一下这段代码如何工作。我们调用了一个普通的函数 `foo(..)` ，而且我们显然可以从Ajax调用那里得到 `text` ，即便它是异步的。

这怎么可能？如果你回忆一下第一章的最开始，我们有一个几乎完全一样的代码：

```
1. var data = ajax( "..url 1.." );
2. console.log( data );
```

但是这段代码不好用！你能发现不同吗？它就是在generator中使用的 `yield` 。

这就是魔法发生的地方！是它允许我们拥有一个看起来是阻塞的，同步的，但实际上不会阻塞整个程序的代码；它仅仅暂停/阻塞在generator本身的代码。

在 `yield foo(11,31)` 中，首先 `foo(11,31)` 调用被发起，它什么也不返回（也就是 `undefined` ），所以我们发起了数据请求，然后我们实际上做的是 `yield undefined` 。这没问题，因为这段代码现在没有依赖 `yield` 的值来做任何有趣的事。我们在本章稍后再重新讨论这个问题。

在这里，我们没有将 `yield` 作为消息传递的工具，只是作为进行暂停/阻塞的流程控制的工具。实际上，它会传递消息，但是只是单向的，在generator被继续运行之后。

那么，generator暂停在了 `yield` ，它实质上再问一个问题，“我该将什么值返回并赋给变

量 `text` ？”谁来回答这个问题？

看一下 `foo(..)` 。如果Ajax请求成功，我们调用：

```
it.next( data );
```

这将使generator使用应答数据继续运行，这意味着我们暂停的 `yield` 表达式直接收到这个值，然后因为它重新开始以运行generator代码，所以这个值被赋给本地变量 `text` 。

很酷吧？

退一步考虑一下它的意义。我们在generator内部的代码看起来完全是同步的（除了 `yield` 关键字本身），但隐藏在幕后的是，在 `foo(..)` 内部，操作可以完全是异步的。

这很伟大！ 这几乎完美地解决了我们前面遇到的问题：回调不能像我们的大脑可以关联的那样，以一种顺序，同步的风格表达异步处理。

实质上，我们将异步处理作为实现细节抽象出去，以至于我们可以同步地/顺序地推理我们的流程控制：“发起Ajax请求，然后在它完成之后打印应答。” 当然，我们仅仅在这个流程控制中表达了两个步骤，但同样的能力可以无边界地延伸，让我们需要表达多少步骤，就表达多少。

提示： 这是一个如此重要的认识，为了充分理解，现在回过头去再把最后三段读一遍！

同步错误处理

但是前面的generator代码会 让 出更多的好处给我们。让我们把注意力移到generator内部的 `try..catch` 上：

```
try {
    var text = yield foo( 11, 31 );
    console.log( text );
}
catch (err) {
    console.error( err );
}
```

这是怎么工作的？ `foo(..)` 调用是异步完成的， `try..catch` 不是无法捕捉异步错误吗？就像我们在第三章中看到的？

我们已经看到了 `yield` 如何让赋值语句暂停，来等待 `foo(..)` 去完成，以至于完成的响应可以被赋予 `text` 。牛X的是， `yield` 暂停 还 允许generator来 `catch` 一个错误。我们在前面的例子，我们用这一部分代码将这个错误抛出到generator中：

```
if (err) {
```

```
    // 向`*main()`中扔进一个错误
    it.throw( err );
}
```

generator的 `yield` 暂停特性不仅意味着我们可以从异步的函数调用那里得到看起来同步的 `return` 值，还意味着我们可以同步地捕获这些异步函数调用的错误！

那么我们看到了，我们可以将错误 抛入 generator，但是将错误 抛出 一个generator呢？和你期望的一样：

```
function *main() {
    var x = yield "Hello World";

    yield x.toLowerCase();    // 引发一个异常！
}

var it = main();

it.next().value;            // Hello World

try {
    it.next( 42 );
}
catch (err) {
    console.error( err );    // TypeError
}
```

当然，我们本可以用 `throw ..` 手动地抛出一个错误，而不是制造一个异常。

我们甚至可以 `catch` 我们 `throw(..)` 进generator的同一个错误，实质上给了generator一个机会来处理它，但如果generator没处理，那么 迭代器 代码必须处理它：

```
function *main() {
    var x = yield "Hello World";

    // 永远不会跑到这里
    console.log( x );
}

var it = main();

it.next();

try {
    // `*main()`会处理这个错误吗？我们走着瞧！
    it.throw( "Oops" );
```

```
}
catch (err) {
    // 不，它没处理！
    console.error( err );            // Oops
}
```

使用异步代码的，看似同步的错误处理（通过 `try..catch` ）在可读性和可推理性上大获全胜。

Generators + Promises

Generators + Promises

在我们前面的讨论中，我们展示了generator如何可以异步地迭代，这是一个用顺序的可推理性来取代混乱如面条的回调的一个巨大进步。但我们丢掉了两个非常重要的东西：Promise的可靠性和可组合性（见第三章）！

别担心—我们会把它们拿回来。在ES6的世界中最棒的就是将generator（看似同步的异步代码）与Promise（可靠性和可组合性）组合起来。

但怎么做呢？

回想一下第三章中我们基于Promise的方式运行Ajax的例子：

```
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

foo( 11, 31 )
.then(
    function(text){
        console.log( text );
    },
    function(err){
        console.error( err );
    }
);
```

在我们早先的运行Ajax的例子的generator代码中，`foo(..)` 什么也不返回（`undefined`），而且我们的 迭代器 控制代码也不关心 `yield` 的值。

但这里的Promise相关的 `foo(..)` 在发起Ajax调用后返回一个promise。这暗示着我们可以用 `foo(..)` 构建一个promise，然后从generator中 `yield` 出来，而后 迭代器 控制代码将可以收到这个promise。

那么 迭代器 应当对promise做什么？

它应当监听promise的解析（完成或拒绝），然后要么使用完成消息继续运行generator，要么使用拒绝理由向generator抛出错误。

让我重复一遍，因为它如此重要。发挥Promise和generator的最大功效的自然方法是 `yield` 一个**Promise**，并将这个Promise连接到generator的 迭代器 的控制端。

让我们试一下！首先，我们将Promise相关的 `foo(..)` 与generator `*main()` 放在一起：

```
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

function *main() {
    try {
        var text = yield foo( 11, 31 );
        console.log( text );
    }
    catch (err) {
        console.error( err );
    }
}
```

在这个重构中最强大的启示是， `*main()` 内部的代码 更本就没变！ 在generator内部，无论什么样的值被 `yield` 出去都是一个不可见的实现细节，所以我们甚至不会察觉它发生了，也不用担心它。

那么我们现在如何运行 `*main()` ？我们还有一些管道的实现工作要做，接收并连接 `yield` 的promise，使它能够根据解析来继续运行generator。我们从手动这么做开始：

```
var it = main();

var p = it.next().value;

// 等待`p` promise解析
p.then(
    function(text){
        it.next( text );
    },
    function(err){
        it.throw( err );
    }
);
```

其实，根本不费事，对吧？

这段代码应当看起来与我们早前做的很相似：手动地连接被错误优先的回调控制的generator。与 `if (err) { it.throw..` 不同的是，promise已经为我们分割为完成（成功）与拒绝（失败），否则 迭代器 控制是完全相同的。

现在，我们已经掩盖了一些重要的细节。

最重要的是，我们利用了这样一个事实：我们知道 `*main()` 里面只有一个Promise相关的步骤。如果我们想要能用Promise驱动一个generator而不管它有多少步骤呢？我们当然不想为每一个generator手动编写一个不同的Promise链！要是有这样一种方法该多好：可以重复（也就是“循环”）迭代的控制，而且每次一有Promise出来，就在继续之前等待它的解析。

另外，如果generator在 `it.next()` 调用期间抛出一个错误怎么办？我们是该退出，还是应该 `catch` 它并把它送回去？相似地，要是我们 `it.throw(..)` 一个Promise拒绝给generator，但是没有被处理，又直接回来了呢？

带有Promise的Generator运行器

你在这条路上探索得越远，你就越能感到，“哇，要是有一些工具能帮我做这些就好了。”而且你绝对是对的。这是一种如此重要的模式，而且你不想把它弄错（或者因为一遍又一遍地重复它而把自己累死），所以你最好的选择是把赌注压在一个工具上，而它以我们将要描述的方式使用这种特定设计的工具来 运行 `yield` Promise的generator。

有几种Promise抽象库提供了这样的工具，包括我的 *asynquence* 库和它的 `runner(..)`，我们将在本书的在附录A中讨论它。

但看在学习和讲解的份儿上，让我们定义我们自己的名为 `run(..)` 的独立工具：

```
// 感谢Benjamin Gruenbaum (@benjamingr在GitHub)在此做出的巨大改进！
function run(gen) {
    var args = [].slice.call( arguments, 1), it;

    // 在当前的上下文环境中初始化generator
    it = gen.apply( this, args );

    // 为generator的完成返回一个promise
    return Promise.resolve()
        .then( function handleNext(value){
            // 运行至下一个让出的值
            var next = it.next( value );

            return (function handleResult(next){
                // generator已经完成运行了？
                if (next.done) {
                    return next.value;
```

```
                }
                // 否则继续执行
                else {
                    return Promise.resolve( next.value )
                        .then(
                            // 在成功的情况下继续异步循环，将解析的值送回generator
                            handleNext,

                            // 如果`value`是一个拒绝的promise，就将错误传播回generator自己的错误处理g
                            function handleErr(err) {
                                return Promise.resolve(
                                    it.throw( err )
                                )
                                .then( handleResult );
                            }
                        );
                }
            })(next);
        } );
}
```

如你所见，它可能比你想要自己编写的东西复杂得多，特别是你将不会想为每个你使用的generator重复这段代码。所以，一个帮助工具/库绝对是可行的。虽然，我鼓励你花几分钟时间研究一下这点代码，以便对如何管理generator+Promise交涉得到更好的感觉。

你如何在我们 正在讨论 的Ajax例子中将 `run(..)` 和 `*main()` 一起使用呢？

```
function *main() {
    // ..
}

run( main );
```

就是这样！按照我们连接 `run(..)` 的方式，它将自动地，异步地推进你传入的generator，直到完成。

注意： 我们定义的 `run(..)` 返回一个promise，它被连接成一旦generator完成就立即解析，或者收到一个未捕获的异常，而generator没有处理它。我们没有在这里展示这种能力，但我们会在本章稍后回到这个话题。

ES7： `async` 和 `await`？

前面的模式—generator让出一个Promise，然后这个Promise控制generator的 迭代器 向前推进至它完成—是一个如此强大和有用的方法，如果我们能不通过乱七八糟的帮助工具库（也就是 `run(..)` ）来使用它就更好了。

在这方面可能有一些好消息。在写作这本书的时候，后ES6，ES7化的时间表上已经出现了草案，对这个问题提供早期但强大的附加语法支持。显然，现在还太早而不能保证其细节，但是有相当大的可能性它将蜕变为类似于下面的东西：

```
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

async function main() {
    try {
        var text = await foo( 11, 31 );
        console.log( text );
    }
    catch (err) {
        console.error( err );
    }
}

main();
```

如你所见，这里没有 `run(..)` 调用（意味着不需要工具库！）来驱动和调用 `main()` ——它仅仅像一个普通函数那样被调用。另外， `main()` 不再作为一个generator函数声明；它是一种新型的函数： `async function` 。而最后，与 `yield` 一个Promise相反，我们 `await` 它解析。

如果你 `await` 一个Promise， `async function` 会自动地知道做什么——它会暂停这个函数（就像使用generator那样）直到Promise解析。我们没有在这个代码段中展示，但是调用一个像 `main()` 这样的异步函数将自动地返回一个promise，它会在函数完全完成时被解析。

提示： `async` / `await` 的语法应该对拥有C#经验的读者看起来非常熟悉，因为它们基本上是一样的。

这个草案实质上是为我们已经衍生出的模式进行代码化的支持，成为一种语法机制：用看似同步的流程控制代码与Promise组合。将两个世界的最好部分组合，来有效解决我们用回调遇到的几乎所有主要问题。

这样的ES7化草案已经存在，并且有了早期的支持和热忱的拥护。这一事实为这种异步模式在未来的重要性上信心满满地投了有力的一票。

Generator中的Promise并发

至此，所有我们展示过的是一种使用Promise+generator的单步异步流程。但是现实世界的代码将总是有许多异步步骤。

如果你不小心，generator看似同步的风格也许会蒙蔽你，使你在如何构造你的异步并发上感到自满，导致性能次优的模式。那么我们想花一点时间来探索一下其他选项。

想象一个场景，你需要从两个不同的数据源取得数据，然后将这些应答组合来发起第三个请求，最后打印出最终的应答。我们在第三章中用Promise探索过类似的场景，但这次让我们在generator的环境下考虑它。

你的第一直觉可能是像这样的东西：

```
function *foo() {
    var r1 = yield request( "http://some.url.1" );
    var r2 = yield request( "http://some.url.2" );

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
    );

    console.log( r3 );
}

// 使用刚才定义的`run(..)`工具
run( foo );
```

这段代码可以工作，但在我们特定的这个场景中，它不是最优的。你能发现为什么吗？

因为 `r1` 和 `r2` 请求可以—而且为了性能的原因，应该—并发运行，但在这段代码中它们将顺序地运行；直到 `"http://some.url.1"` 请求完成之前， `"http://some.url.2"` URL不会被Ajax取得。这两个请求是独立的，所以性能更好的方式可能是让它们同时运行。

但是使用generator和 `yield` ，到底应该怎么做？我们知道 `yield` 在代码中只是一个单独的暂停点，所以你根本不能再同一时刻做两次暂停。

最自然和有效的答案是基于Promise的异步流程，特别是因为它们的时间无关的状态管理能力（参见第三章的“未来的值”）。

最简单的方式：

```
function *foo() {
    // 使两个请求“并行”
    var p1 = request( "http://some.url.1" );
    var p2 = request( "http://some.url.2" );

    // 等待两个promise都被解析
    var r1 = yield p1;
    var r2 = yield p2;

```

```
    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
    );

    console.log( r3 );
}

// 使用刚才定义的`run(..)`工具
run( foo );
```

为什么这与前一个代码段不同？看看 `yield` 在哪里和不在哪里。 `p1` 和 `p2` 是并发地（也就是“并行”）发起的Ajax请求promise。它们哪一个先完成都不要紧，因为promise会一直保持它们的解析状态。

然后我们使用两个连续的 `yield` 语句等待并从promise中取得解析值（分别取到 `r1` 和 `r2` 中）。如果 `p1` 首先解析， `yield p1` 会首先继续执行然后等待 `yield p2` 继续执行。如果 `p2` 首先解析，它将会耐心地保持解析值知道被请求，但是 `yield p1` 将会首先停住，直到 `p1` 解析。

不管是哪一种情况， `p1` 和 `p2` 都将并发地运行，并且在 `r3 = yield request..` Ajax请求发起之前，都必须完成，无论以哪种顺序。

如果这种流程控制处理模型听起来很熟悉，那是因为它基本上和我们在第三章中介绍的，因 `Promise.all([ .. ])` 工具成为可能的“门”模式是相同的。所以，我们也可以像这样表达这种流程控制：

```
function *foo() {
    // 使两个请求“并行”并等待两个promise都被解析
    var results = yield Promise.all( [
        request( "http://some.url.1" ),
        request( "http://some.url.2" )
    ] );

    var r1 = results[0];
    var r2 = results[1];

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
    );

    console.log( r3 );
}

// 使用前面定义的`run(..)`工具
run( foo );
```

注意： 就像我们在第三章中讨论的，我们甚至可以用ES6解构赋值来把 `var r1 = .. var r2 = ..` 赋值简写为 `var [r1,r2] = results` 。

换句话说，在generator+Promise的方式中，Promise所有的并发能力都是可用的。所以在任何地方，如果你需要比“这个然后那个”要复杂的顺序异步流程步骤时，Promise都可能是最佳选择。

Promises，隐藏起来

作为代码风格的警告要说一句，要小心你在 你的**generator**内部 包含了多少Promise逻辑。以我们描述过的方式在异步性上使用generator的全部意义，是要创建简单，顺序，看似同步的代码，并尽可能多地将异步性细节隐藏在这些代码之外。

比如，这可能是一种更干净的方式：

```
// 注意：这是一个普通函数，不是generator
function bar(url1,url2) {
    return Promise.all( [
        request( url1 ),
        request( url2 )
    ] );
}

function *foo() {
    // 将基于Promise的并发细节隐藏在`bar(..)`内部
    var results = yield bar(
        "http://some.url.1",
        "http://some.url.2"
    );

    var r1 = results[0];
    var r2 = results[1];

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
    );

    console.log( r3 );
}

// 使用刚才定义的`run(..)`工具
run( foo );
```

在 `*foo()` 内部，它更干净更清晰地表达了我们要做的事情：我们要求 `bar(..)` 给我们一些 `results` ，而我们将用 `yield` 等待它的发生。我们不必关心在底层一个 `Promise.all([ .. ])` 的Promise组合将被用来完成任务。

我们将异步性，特别是**Promise**，作为一种实现细节。

如果你要做一种精巧的序列流控制，那么将你的Promise逻辑隐藏在一个仅仅从你的generator中调用的函数里特别有用。举个例子：

```
function bar() {
    return Promise.all( [
                baz( .. )
                .then( .. ),
                Promise.race( [ .. ] )
            ] )
            .then( .. )
}
```

有时候这种逻辑是必须的，而如果你直接把它扔在你的generator内部，你就违背了大多数你使用generator的初衷。我们 应当 有意地将这样的细节从generator代码中抽象出去，以使它们不会搞乱更高层的任务表达。

在创建功能强与性能好的代码之上，你还应当努力使代码尽可能地容易推理和维护。

注意： 对于编程来说，抽象不总是一种健康的东西—许多时候它可能在得到简洁的同时增加复杂性。但是在这种情况下，我相信你的generator+Promise异步代码要比其他的选择健康得多。虽然有所有这些建议，你仍然要注意你的特殊情况，并为你和你的团队做出合适的决策。

Generator 委托

Generator 委托

在上一节中，我们展示了从generator内部调用普通函数，和它如何作为一种有用的技术来将实现细节（比如异步Promise流程）抽象出去。但是为这样的任务使用普通函数的缺陷是，它必须按照普通函数的规则行动，也就是说它不能像generator那样用 `yield` 来暂停自己。

在你身上可能发生这样的事情：你可能会试着使用我们的 `run(..)` 帮助函数，从一个generator中调用另个一generator。比如：

```
function *foo() {
    var r2 = yield request( "http://some.url.2" );
    var r3 = yield request( "http://some.url.3/?v=" + r2 );

    return r3;
}

function *bar() {
    var r1 = yield request( "http://some.url.1" );

    // 通过`run(..)`“委托”到`*foo()`
    var r3 = yield run( foo );

    console.log( r3 );
}

run( bar );
```

通过再一次使用我们的 `run(..)` 工具，我们在 `*bar()` 内部运行 `*foo()` 。我们利用了这样一个事实：我们早先定义的 `run(..)` 返回一个promise，这个promise在generator运行至完成时才解析（或发生错误），所以如果我们从一个 `run(..)` 调用中 `yield` 出一个promise给另一个 `run(..)` ，它就会自动暂停 `*bar()` 直到 `*foo()` 完成。

但这里有一个更好的办法将 `*foo()` 调用整合进 `*bar()` ，它称为 `yield` 委托。 `yield` 委托的特殊语法是： `yield * __` （注意额外的 `*` ）。让它在我们前面的例子中工作之前，让我们看一个更

简单的场景：

```
function *foo() {
    console.log( "`*foo()` starting" );
    yield 3;
    yield 4;
    console.log( "`*foo()` finished" );
}

function *bar() {
    yield 1;
    yield 2;
    yield *foo();    // `yield`-delegation!
    yield 5;
}

var it = bar();

it.next().value;    // 1
it.next().value;    // 2
it.next().value;    // `*foo()` starting
                    // 3
it.next().value;    // 4
it.next().value;    // `*foo()` finished
                    // 5
```

注意： 在本章早前的一个注意点中，我解释了为什么我偏好 `function *foo() ..` 而不是 `function* foo() ..` ，相似地，我也偏好——与关于这个话题的其他大多数文档不同——说 `yield *foo()` 而不是 `yield* foo()` 。 `*` 的摆放是纯粹的风格问题，而且要看你的最佳判断。但我发现保持统一风格很吸引人。

`yield *foo()` 委托是如何工作的？

首先，正如我们看到过的那样，调用 `foo()` 创建了一个 迭代器。然后， `yield *` 将（当前 `*bar()` generator的） 迭代器 的控制委托/传递给这另一个 `*foo()` 迭代器。

那么，前两个 `it.next()` 调用控制着 `*bar()` ，但当我们发起第三个 `it.next()` 调用时， `*foo()` 就启动了，而且这时我们控制的是 `*foo()` 而非 `*bar()` 。这就是为什么它称为委托——`*bar()` 将它的迭代控制委托给 `*foo()` 。

只要 `it` 迭代器 的控制耗尽了整个 `*foo()` 迭代器，它就会自动地将控制返回到 `*bar()` 。

那么现在回到前面的三个顺序Ajax请求的例子：

```
function *foo() {
    var r2 = yield request( "http://some.url.2" );
```

```
    var r3 = yield request( "http://some.url.3/?v=" + r2 );

    return r3;
}

function *bar() {
    var r1 = yield request( "http://some.url.1" );

    // 通过`run(..)`“委托”到`*foo()`
    var r3 = yield *foo();

    console.log( r3 );
}

run( bar );
```

这个代码段和前面使用的版本的唯一区别是，使用了 `yield *foo()` 而不是前面的 `yield run(foo)` 。

注意： `yield *` 让出了迭代控制，不是generator控制；当你调用 `*foo()` generator时，你就 `yield` 委托给它的 迭代器。但你实际上可以 `yield` 委托给任何 迭代器； `yield *[1,2,3]` 将会消费默认的 `[1,2,3]` 数组值 迭代器。

为什么委托？

`yield` 委托的目的很大程度上是为了代码组织，而且这种方式是与普通函数调用对称的。

想象两个分别提供了 `foo()` 和 `bar()` 方法的模块，其中 `bar()` 调用 `foo()` 。它们俩分开的原因一般是由于为了程序将它们作为分离的程序来调用而进行的恰当组织。例如，可能会有一些情况 `foo()` 需要被独立调用，而其他地方 `bar()` 来调用 `foo()` 。

由于这些完全相同的原因，将generator分开可以增强程序的可读性，可维护性，与可调试性。从这个角度讲， `yield *` 是一种快捷的语法，用来在 `*bar()` 内部手动地迭代 `*foo()` 的步骤。

如果 `*foo()` 中的步骤是异步的，这样的手动方式可能会特别复杂，这就是为什么你可能会需要那个 `run(..)` 工具来做它。正如我们已经展示的， `yield *foo()` 消灭了使用 `run(..)` 工具的子实例（比如 `run(foo)` ）的需要。

委托消息

你可能想知道，这种 `yield` 委托在除了与 迭代器 控制一起工作以外，是如何与双向消息传递一起工作的。仔细查看下面这些通过 `yield` 委托进进出出的消息流：

```
function *foo() {
    console.log( "inside `*foo()`:", yield "B" );

```

```
    console.log( "inside `*foo()`:", yield "C" );

    return "D";
}

function *bar() {
    console.log( "inside `*bar()`:", yield "A" );

    // `yield`-委托！
    console.log( "inside `*bar()`:", yield *foo() );

    console.log( "inside `*bar()`:", yield "E" );

    return "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// inside `*bar()`: 1
// outside: B

console.log( "outside:", it.next( 2 ).value );
// inside `*foo()`: 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// inside `*foo()`: 3
// inside `*bar()`: D
// outside: E

console.log( "outside:", it.next( 4 ).value );
// inside `*bar()`: 4
// outside: F
```

特别注意一下 `it.next(3)` 调用之后的处理步骤：

1. 值 `3` 被传入（通过 `*bar` 里的 `yield` 委托）在 `*foo()` 内部等待中的 `yield "C"` 表达式。
2. 然后 `*foo()` 调用 `return "D"`，但是这个值不会一路返回到外面的 `it.next(3)` 调用。
3. 相反地，值 `"D"` 作为结果被发送到在 `*bar()` 内部等待中的 `yield *foo()` 表示式—这个 `yield` 委托表达式实质上在 `*foo()` 被耗尽之前一直被暂停着。所以 `"D"` 被送到 `*bar()` 内部来让它打印。
4. `yield "E"` 在 `*bar()` 内部被调用，而且值 `"E"` 被让出到外部作为 `it.next(3)` 调用的结果。

从外部 迭代器（ `it` ）的角度来看，在初始的generator和被委托的generator之间的控制没有任何区别。

事实上， `yield` 委托甚至不必指向另一个generator；它可以仅被指向一个非generator的，一般的 *iterable*。比如：

```
function *bar() {
    console.log( "inside `*bar()`:", yield "A" );

    // `yield`-委托至一个非generator
    console.log( "inside `*bar()`:", yield *[ "B", "C", "D" ] );

    console.log( "inside `*bar()`:", yield "E" );

    return "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// inside `*bar()`: 1
// outside: B

console.log( "outside:", it.next( 2 ).value );
// outside: C

console.log( "outside:", it.next( 3 ).value );
// outside: D

console.log( "outside:", it.next( 4 ).value );
// inside `*bar()`: undefined
// outside: E

console.log( "outside:", it.next( 5 ).value );
// inside `*bar()`: 5
// outside: F
```

注意这个例子与前一个之间，被接收/报告的消息的不同之处。

最惊人的是，默认的 `array` 迭代器 不关心任何通过 `next(..)` 调用被发送的消息，所以值 `2` ， `3` ，与 `4` 实质上被忽略了。另外，因为这个 迭代器 没有明确的 `return` 值（不像前面使用的 `*foo()` ），所以 `yield *` 表达式在它完成时得到一个 `undefined` 。

异常也委托！

与 `yield` 委托在两个方向上透明地传递消息的方式相同，错误/异常也在双向传递：

```
function *foo() {
    try {
        yield "B";
    }
    catch (err) {
        console.log( "error caught inside `*foo()`:", err );
    }

    yield "C";

    throw "D";
}

function *bar() {
    yield "A";

    try {
        yield *foo();
    }
    catch (err) {
        console.log( "error caught inside `*bar()`:", err );
    }

    yield "E";

    yield *baz();

    // note: can't get here!
    yield "G";
}

function *baz() {
    throw "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// outside: B

console.log( "outside:", it.throw( 2 ).value );
```

```
// error caught inside `*foo()`: 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// error caught inside `*bar()`: D
// outside: E

try {
    console.log( "outside:", it.next( 4 ).value );
}
catch (err) {
    console.log( "error caught outside:", err );
}
// error caught outside: F
```

在这段代码中有一些事情要注意：

1. 但我们调用`it.throw(2)`时，它发送一个错误消息`2`到`*bar()`，而`*bar()`将它委托至`*foo()`，然后`*foo()`来`catch`它并平静地处理。之后，`yield "C"`把`"C"`作为返回的`value`发送回`it.throw(2)`调用。
2. 接下来值`"D"`被从`*foo()`内部`throw`出来并传播到`*bar()`，`*bar()`会`catch`它并平静地处理。然后`yield "E"`把`"E"`作为返回的`value`发送回`it.next(3)`调用。
3. 接下来，一个异常从`*baz()`中`throw`出来，而没有被`*bar()`捕获—我们没在外面`catch`它—所以`*baz()`和`*bar()`都被设置为完成状态。这段代码结束后，即便有后续的`next(..)`调用，你也不会得到值`"G"`—它们的`value`将返回`undefined`。

异步委托

最后让我们回到早先的多个顺序Ajax请求的例子，使用`yield`委托：

```
function *foo() {
    var r2 = yield request( "http://some.url.2" );
    var r3 = yield request( "http://some.url.3/?v=" + r2 );

    return r3;
}

function *bar() {
    var r1 = yield request( "http://some.url.1" );

    var r3 = yield *foo();

    console.log( r3 );
}

```

```
run( bar );
```

在 `*bar()` 内部，与调用 `yield run(foo)` 不同的是，我们调用 `yield *foo()` 就可以了。

在前一个版本的这个例子中，Promise机制（通过 `run(..)` 控制的）被用于将值从 `*foo()` 中的 `return r3` 传送到 `*bar()` 内部的本地变量 `r3` 。现在，这个值通过 `yield *` 机制直接返回。

除此以外，它们的行为是一样的。

“递归”委托

当然， `yield` 委托可以一直持续委托下去，你想连接多少步骤就连接多少。你甚至可以在具有异步能力的generator上“递归”使用 `yield` 委托——一个 `yield` 委托至自己的generator：

```
function *foo(val) {
    if (val > 1) {
        // 递归委托
        val = yield *foo( val - 1 );
    }

    return yield request( "http://some.url/?v=" + val );
}

function *bar() {
    var r1 = yield *foo( 3 );
    console.log( r1 );
}

run( bar );
```

注意： 我们的 `run(..)` 工具本可以用 `run( foo, 3 )` 来调用，因为它支持用额外传递的参数来进行generator的初始化。然而，为了在这里高调展示 `yield *` 的灵活性，我们使用了无参数的 `*bar()` 。

这段代码之后的处理步骤是什么？坚持住，它的细节要描述起来可是十分错综复杂：

1. `run(bar)` 启动了 `*bar()` generator。
2. `foo(3)` 为 `*foo(..)` 创建了 迭代器 并传递 `3` 作为它的 `val` 参数。
3. 因为 `3 > 1` ， `foo(2)` 创建了另一个 迭代器 并传递 `2` 作为它的 `val` 参数。
4. 因为 `2 > 1` ， `foo(1)` 又创建了另一个 迭代器 并传递 `1` 作为它的 `val` 参数。
5. `1 > 1` 是 `false` ，所以我们接下来用值 `1` 调用 `request(..)` ，并得到一个代表第一个Ajax调用的promise。
6. 这个promise被 `yield` 出来，回到 `*foo(2)` generator实例。
7. `yield *` 将这个promise传出并回到 `*foo(3)` 生成generator。另一个 `yield *` 把这个

promise传出到`*bar()` generator实例。而又有另一个`yield *`把这个promise传出到`run(..)`工具，而它将会等待这个promise（第一个Ajax请求）再处理。

8. 当这个promise解析时，它的完成消息会被发送以继续`*bar()`，`*bar()`通过`yield *`把消息传递进`*foo(3)`实例，`*foo(3)`实例通过`yield *`把消息传递进`*foo(2)` generator实例，`*foo(2)`实例通过`yield *`把消息传给那个在`*foo(3)` generator实例中等待的一般的`yield`。
9. 这第一个Ajax调用的应答现在立即从`*foo(3)` generator实例中被`return`，作为`*foo(2)`实例中`yield *`表达式的结果发送回来，并赋值给本地`val`变量。
0. `*foo(2)`内部，第二个Ajax请求用`request(..)`发起，它的promise被`yield`回到`*foo(1)`实例，然后一路`yield *`传播到`run(..)`（回到第7步）。当promise解析时，第二个Ajax应答一路传播回到`*foo(2)` generator实例，并赋值到他本地的`val`变量。
1. 最终，第三个Ajax请求用`request(..)`发起，它的promise走出到`run(..)`，然后它的解析值一路返回，最后被`return`到在`*bar()`中等待的`yield *`表达式。

天！许多疯狂的头脑杂技，对吧？你可能想要把它通读几遍，然后抓点儿零食放松一下大脑！

Generator 并发

Generator并发

正如我们在第一章和本章早先讨论过的，另个同时运行的“进程”可以协作地穿插它们的操作，而且许多时候这可以产生非常强大的异步表达式。

坦白地说，我们前面关于多个generator并发穿插的例子，展示了这真的容易让人糊涂。但我们也受到了启发，有些地方这种能力十分有用。

回想我们在第一章中看过的场景，两个不同但同时的Ajax应答处理需要互相协调，来确保数据通信不是竟合状态。我们这样把应答分别放在 `res` 数组的不同位置中：

```
function response(data) {
    if (data.url == "http://some.url.1") {
        res[0] = data;
    }
    else if (data.url == "http://some.url.2") {
        res[1] = data;
    }
}
```

但是我们如何在这种场景下使用多generator呢？

```
// `request(..)` 是一个基于Promise的Ajax工具

var res = [];

function *reqData(url) {
    res.push(
        yield request( url )
    );
}
```

注意： 我们将在这里使用两个 `*reqData(..)` generator的实例，但是这和分别使用两个不同generator的一个实例没有区别；这两种方式在道理上完全一样的。我们过一会儿就会看到两个generator的协调操作。

与不得不将 `res[0]` 和 `res[1]` 赋值手动排序不同，我们将使用协调过的顺序，让 `res.push(..)` 以可预见的顺序恰当地将值放在预期的位置。如此被表达的逻辑会让人感觉更干净。

但是我们将如何实际安排这种互动呢？首先，让我们手动实现它：

```
var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it1.next().value;
var p2 = it2.next().value;

p1
.then( function(data){
    it1.next( data );
    return p2;
} )
.then( function(data){
    it2.next( data );
} );
```

`*reqData(..)` 的两个实例都开始发起它们的Ajax请求，然后用 `yield` 暂停。之后我们再 `p1` 解析时继续运行第一个实例，而后来的 `p2` 的解析将会重启第二个实例。以这种方式，我们使用Promise的安排来确保 `res[0]` 将持有第一个应答，而 `res[1]` 持有第二个应答。

但坦白地说，这是可怕的手动，而且它没有真正让generator组织它们自己，而那才是真正的力量。让我们用不同的方法试一下：

```
// `request(..)` 是一个基于Promise的Ajax工具

var res = [];

function *reqData(url) {
    var data = yield request( url );

    // 传递控制权
    yield;

    res.push( data );
}

var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it1.next().value;
var p2 = it2.next().value;

p1.then( function(data){
    it1.next( data );
} );

p2.then( function(data){
```

```
    it2.next( data );
} );

Promise.all( [p1,p2] )
.then( function(){
    it1.next();
    it2.next();
} );
```

好的，这看起来好些了（虽然仍然是手动），因为现在两个 `*reqData(..)` 的实例真正地并发运行了，而且（至少是在第一部分）是独立的。

在前一个代码段中，第二个实例在第一个实例完全完成之前没有给出它的数据。但是这里，只要它们的应答一返回这两个实例就立即分别收到他们的数据，然后每个实例调用另一个 `yield` 来传送控制。最后我们在 `Promise.all([ .. ])` 的处理器中选择用什么样的顺序继续它们。

可能不太明显的是，这种方式因其对称性启发了一种可复用工具的简单形式。让我们想象使用一个称为 `runAll(..)` 的工具：

```
// `request(..)` 是一个基于Promise的Ajax工具

var res = [];

runAll(
    function*(){
        var p1 = request( "http://some.url.1" );

        // 传递控制权
        yield;

        res.push( yield p1 );
    },
    function*(){
        var p2 = request( "http://some.url.2" );

        // 传递控制权
        yield;

        res.push( yield p2 );
    }
);
```

注意： 我们没有包含 `runAll(..)` 的实现代码，不仅因为它长得无法行文，也因为它是一个我们已经在先前的 `run(..)` 中实现的逻辑的扩展。所以，作为留给读者的一个很好的补充性练习，请你自己动手改进 `run(..)` 的代码，来使它像想象中的 `runAll(..)` 那样工作。另外，我的 *asynquence* 库提供了一个前面提到过的 `runner(..)` 工具，它内建了这种能力，我们将在本书的附录A中讨论它。

这是 `runAll(..)` 内部的处理将如何操作：

1. 第一个generator得到一个代表从 `"http://some.url.1"` 来的Ajax应答，然后将控制权 `yield` 回到 `runAll(..)` 工具。
2. 第二个generator运行，并对 `"http://some.url.2"` 做相同的事，将控制权 `yield` 回到 `runAll(..)` 工具。
3. 第一个generator继续，然后 `yield` 出他的promise `p1` 。在这种情况下 `runAll(..)` 工具和我们前面的 `run(..)` 做同样的事，它等待promise解析，然后继续这同一个generator（没有控制传递！）。当 `p1` 解析时， `runAll(..)` 使用解析值再一次继续第一个generator，而后 `res[0]` 得到它的值。在第一个generator完成之后，有一个隐式的控制权传递。
4. 第二个generator继续， `yield` 出它的promise `p2` ，并等待它的解析。一旦 `p2` 解析， `runAll(..)` 使用这个解析值继续第二个generator，于是 `res[1]` 被设置。

在这个例子中，我们使用了一个称为 `res` 的外部变量来保存两个不同的Ajax应答的结果—这是我们的并发协调。

但是这样做可能十分有帮助：进一步扩展 `runAll(..)` 使它为多个generator实例提供 分享的 内部的变量作用域，比如一个我们将在下面称为 `data` 的空对象。另外，它可以接收被 `yield` 的非Promise值，并把它们交给下一个generator。

考虑这段代码：

```
// `request(..)` 是一个基于Promise的Ajax工具

runAll(
    function*(data){
        data.res = [];

        // 传递控制权（并传递消息）
        var url1 = yield "http://some.url.2";

        var p1 = request( url1 ); // "http://some.url.1"

        // 传递控制权
        yield;

        data.res.push( yield p1 );
    },
    function*(data){
        // 传递控制权（并传递消息）
        var url2 = yield "http://some.url.1";

        var p2 = request( url2 ); // "http://some.url.2"

        // 传递控制权
        yield;
```

```
25.
26.         data.res.push( yield p2 );
27.     }
28. );
```

在这个公式中，两个generator不仅协调控制传递，实际上还互相通信：通过 `data.res` ，和交换 `url1` 与 `url2` 的值的 `yield` 消息。这强大到不可思议！

这样的认识也是一种更为精巧的称为CSP（Communicating Sequential Processes—通信顺序处理）的异步技术的概念基础，我们将在本书的附录B中讨论它。

Thunks

Thunks

至此，我们都假定从一个generator中 `yield` 一个Promise—让这个Promise使用像 `run(..)` 这样的帮助工具来推进generator—是管理使用generator的异步处理的最佳方法。明白地说，它是的。

但是我们跳过了一个被轻度广泛使用的模式，为了完整性我们将简单地看一看它。

在一般的计算机科学中，有一种老旧的前JS时代的概念，称为“thunk”。我们不在这里赘述它的历史，一个狭隘的表达是，thunk是一个JS函数—没有任何参数—它连接并调用另一个函数。

换句话讲，你用一个函数定义包装函数调用—带着它需要的所有参数—来 推迟 这个调用的执行，而这个包装用的函数就是thunk。当你稍后执行thunk时，你最终会调用那个原始的函数。

举个例子：

```
function foo(x,y) {
    return x + y;
}

function fooThunk() {
    return foo( 3, 4 );
}

// 稍后

console.log( fooThunk() );    // 7
```

所以，一个同步的thunk是十分直白的。但是一个异步的thunk呢？我们实质上可以扩展这个狭隘的thunk定义，让它接收一个回调。

考虑这段代码：

```
function foo(x,y,cb) {
    setTimeout( function(){
        cb( x + y );
    }, 1000 );
}

function fooThunk(cb) {
    foo( 3, 4, cb );
```

```
}

// 稍后

fooThunk( function(sum){
    console.log( sum );        // 7
} );
```

如你所见， `fooThunk(..)` 仅需要一个 `cb(..)` 参数，因为它已经预先制定了值 `3` 和 `4` （分别为 `x` 和 `y` ）并准备传递给 `foo(..)` 。一个thunk只是在外面耐心地等待着它开始工作所需的最后一部分信息：回调。

但是你不会想要手动制造thunk。那么，让我们发明一个工具来为我们进行这种包装。

考虑这段代码：

```
function thunkify(fn) {
    var args = [].slice.call( arguments, 1 );
    return function(cb) {
        args.push( cb );
        return fn.apply( null, args );
    };
}

var fooThunk = thunkify( foo, 3, 4 );

// 稍后

fooThunk( function(sum) {
    console.log( sum );        // 7
} );
```

提示： 这里我们假定原始的（ `foo(..)` ）函数签名希望它的回调的位置在最后，而其它的参数在这之前。这是一个异步JS函数的相当普遍的“标准”。你可以称它为“回调后置风格”。如果因为某些原因你需要处理“回调优先风格”的签名，你只需要制造一个使用 `args.unshift(..)` 而非 `args.push(..)` 的工具。

前面的 `thunkify(..)` 公式接收 `foo(..)` 函数的引用，和任何它所需的参数，并返回thunk本身（ `fooThunk(..)` ）。然而，这并不是你将在JS中发现的thunk的典型表达方式。

与 `thunkify(..)` 制造thunk本身相反，典型的—可能有点儿让人困惑的— `thunkify(..)` 工具将产生一个制造thunk的函数。

额…是的。

考虑这段代码：

```
function thunkify(fn) {
    return function() {
        var args = [].slice.call( arguments );
        return function(cb) {
            args.push( cb );
            return fn.apply( null, args );
        };
    };
}
```

这里主要的不同之处是有一个额外的 `return function() { .. }` 。这是它在用法上的不同：

```
var whatIsThis = thunkify( foo );

var fooThunk = whatIsThis( 3, 4 );

// 稍后

fooThunk( function(sum) {
    console.log( sum );        // 7
} );
```

明显地，这段代码隐含的最大的问题是， `whatIsThis` 叫什么合适？它不是thunk，它是一个从 `foo(..)` 调用生产thunk的东西。它是一种“thunk”的“工厂”。而且看起来没有任何标准的意见来命名这种东西。

所以，我的提议是“thunkory”（”thunk” + “factory”）。于是， `thunkify(..)` 制造了一个thunkory，而一个thunkory制造thunks。这个道理与第三章中我的“promisory”提议是对称的：

```
var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// 稍后

fooThunk1( function(sum) {
    console.log( sum );        // 7
} );

fooThunk2( function(sum) {
    console.log( sum );        // 11
} );
```

注意： 这个例子中的 `foo(..)` 期望的回调不是“错误优先风格”。当然，“错误优先风格”更常见。如

果 `foo(..)` 有某种合理的错误发生机制，我们可以改变而使它期望并使用一个错误优先的回调。后续的 `thunkify(..)` 不会关心回调被预想成什么样。用法的唯一区别是 `fooThunk1(function(err,sum){..` 。

暴露出thunkory方法—而不是像早先的 `thunkify(..)` 那样将中间步骤隐藏起来—可能看起来像是没必要的混乱。但是一般来讲，在你的程序一开始就制造一些thunkory来包装既存API的方法是十分有用的，然后你就可以在你需要thunk的时候传递并调用这些thunkory。这两个区别开的步骤保证了功能上更干净的分离。

来展示一下的话：

```
// 更干净：
var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// 而这个不干净：
var fooThunk1 = thunkify( foo, 3, 4 );
var fooThunk2 = thunkify( foo, 5, 6 );
```

不管你是否愿意明确对付thunkory，thunk（ `fooThunk1(..)` 和 `fooThunk2(..)` ）的用法还是一样的。

s/promise/thunk/

那么所有这些thunk的东西与generator有什么关系？

一般性地比较一下thunk和promise：它们是不能直接互换的，因为它们在行为上不是等价的。比起单纯的thunk，Promise可用性更广泛，而且更可靠。

但从另一种意义上讲，它们都可以被看作是对一个值的请求，这个请求可能被异步地应答。

回忆第三章，我们定义了一个工具来promise化一个函数，我们称之为 `Promise.wrap(..)` —我们本来也可以叫它 `promisify(..)` 的！这个Promise化包装工具不会生产Promise；它生产那些继而生产Promise的promisories。这和我们当前讨论的thunkory和thunk是完全对称的。

为了描绘这种对称性，让我们首先将 `foo(..)` 的例子改为假定一个“错误优先风格”回调的形式：

```
function foo(x,y,cb) {
    setTimeout( function(){
        // 假定 `cb(..)` 是“错误优先风格”
        cb( null, x + y );
    }, 1000 );
}
```

现在，我们将比较 `thunkify(..)` 和 `promisify(..)` （也就是第三章的 `Promise.wrap(..)` ）：

```
// 对称的：构建问题的回答者
var fooThunkory = thunkify( foo );
var fooPromisory = promisify( foo );

// 对称的：提出问题
var fooThunk = fooThunkory( 3, 4 );
var fooPromise = fooPromisory( 3, 4 );

// 取得 thunk 的回答
fooThunk( function(err,sum){
    if (err) {
        console.error( err );
    }
    else {
        console.log( sum );        // 7
    }
} );

// 取得 promise 的回答
fooPromise
.then(
    function(sum){
        console.log( sum );        // 7
    },
    function(err){
        console.error( err );
    }
);
```

thunkory和promisory实质上都是在问一个问题（一个值），thunk的 `fooThunk` 和promise的 `fooPromise` 分别代表这个问题的未来的答案。这样看来，对称性就清楚了。

带着这个视角，我们可以看到为了异步而 `yield` Promise的generator，也可以为异步而 `yield` thunk。我们需要的只是一个更聪明的 `run(..)` 工具（就像以前一样），它不仅可以寻找并连接一个被 `yield` 的Promise，而且可以给一个被 `yield` 的thunk提供回调。

考虑这段代码：

```
function *foo() {
    var val = yield request( "http://some.url.1" );
    console.log( val );
}

run( foo );
```

在这个例子中， `request(..)` 既可以是一个返回一个promise的promisory，也可以是一个返回一个thunk的thunkory。从generator的内部代码逻辑的角度看，我们不关心这个实现细节，这就它强大的地方！

所以， `request(..)` 可以使以下任何一种形式：

```
// promisory `request(..)` （见第三章）
var request = Promise.wrap( ajax );

// vs.

// thunkory `request(..)`
var request = thunkify( ajax );
```

最后，作为一个让我们早先的 `run(..)` 工具支持thunk的补丁，我们可能会需要这样的逻辑：

```
// ..
// 我们收到了一个回调吗？
else if (typeof next.value == "function") {
    return new Promise( function(resolve,reject){
        // 使用一个错误优先回调调用thunk
        next.value( function(err,msg) {
            if (err) {
                reject( err );
            }
            else {
                resolve( msg );
            }
        } );
    } )
    .then(
        handleNext,
        function handleErr(err) {
            return Promise.resolve(
                it.throw( err )
            )
            .then( handleResult );
        }
    );
}
```

现在，我们generator既可以调用promisory来 `yield` Promise，也可以调用thunkory来 `yield` thunk，而不论那种情况， `run(..)` 都将处理这个值并等待它的完成，以继续generator。

在对称性上，这两个方式是看起来相同的。然而，我们应当指出这仅仅从Promise或thunk表示延续generator的未来值的角度讲是成立的。

从更高的角度讲，与Promise被设计成的那样不同，thunk没有提供，它们本身也几乎没有任何可靠性和可组合性的保证。在这种特定的generator异步模式下使用一个thunk作为Promise的替代品是可以工作的，但与Promise提供的所有好处相比，这应当被看做是一种次理想的方法。

如果你有选择，那就偏向 `yield pr` 而非 `yield th` 。但是使 `run(..)` 工具可以处理两种类型的值本身没有什么问题。

注意： 在我们将要在附录A中讨论的，我的 *asynquence* 库中的 `runner(..)` 工具，可以处理 `yield` 的Promise，thunk和 *asynquence* 序列。

前ES6时代的 Generator

前ES6时代的Generator

我希望你已经被说服了，generator是一个异步编程工具箱里的非常重要的增强工具。但它是ES6中的新语法，这意味着你不能像填补Promise（它只是新的API）那样填补generator。那么如果我们不能奢望忽略前ES6时代的浏览器，我们该如何将generator带到浏览器中呢？

对所有ES6中的新语法的扩展，有一些工具——称呼他们最常见的名词是转译器（transpilers），也就是转换编译器（trans-compilers）——它们会拿起你的ES6语法，并转换为前ES6时代的等价代码（但是明显地变难看了！）。所以，generator可以被转译为具有相同行为但可以在ES5或以下版本进行工作的代码。

但是怎么做到的？ `yield` 的“魔法”听起来不像是那么容易转译的。在我们早先的基于闭包的 迭代器 例子中，实际上提示了一种解决方法。

手动变形

在我们讨论转译器之前，让我们延伸一下，在generator的情况下如何手动转译。这不仅是一个学院派的练习，因为这样做实际上可以帮助我们进一步理解它们如何工作。

考虑这段代码：

```
// `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" );
```

第一个要注意的事情是，我们仍然需要一个可以被调用的普通的 `foo()` 函数，而且它仍然需要返回一个 迭代器。那么让我们来画出非generator的变形草图：

```
function foo(url) {

    // ..

    // 制造并返回 iterator
    return {
        next: function(v) {
            // ..
        },
        throw: function(e) {
            // ..
        }
    };
}

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

下一个需要注意的地方是，generator通过挂起它的作用域/状态来施展它的“魔法”，但我们可以用函数闭包来模拟。为了理解如何写出这样的代码，我们将先用状态值注释generator不同的部分：

```
// `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;
    }
}
```

注意： 为了更准去地讲解，我们使用 `TMP1` 变量将 `val = yield request..` 语句分割为两部分。 `request(..)` 发生在状态 `*1*` ，而将完成值赋给 `val` 发生在状态 `*2*` 。在我们将代码转换为非generator的等价物后，我们就可以摆脱中间的 `TMP1` 。

换句话所，`*1*`是初始状态，`*2*`是`request(..)`成功的状态，`*3*`是`request(..)`失败的状态。你可能会想象额外的`yield`步骤将如何编码为额外的状态。

回到我们被转译的generator，让我们在这个闭包中定义一个变量`state`，用它来追踪状态：

```
function foo(url) {
    // 管理 generator 状态
    var state;

    // ..
}
```

现在，让我们在闭包内部定义一个称为`process(..)`的内部函数，它用`switch`语句来处理各种状态。

```
// `request(..)` 是一个支持Promise的Ajax工具

function foo(url) {
    // 管理 generator 状态
    var state;

    // generator-范围的变量声明
    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;
        }
    }

    // ..
}
```

在我们的generator中每种状态都在`switch`语句中有它自己的`case`。每当我们需要处理一个新状态时，`process(..)`就会被调用。我们一会就回来讨论它如何工作。

对任何generator范围的变量声明（ `val` ），我们将它们移动到 `process(..)` 外面的 `var` 声明中，这样它们就可以在 `process(..)` 的多次调用中存活下来。但是“块儿作用域”的 `err` 变量仅在 `*3*` 状态下需要，所以我们将它留在原处。

在状态 `*1*` ，与 `yield request(..)` 相反，我们 `return request(..)` 。在终结状态 `*2*` ，没有明确的 `return` ，所以我们仅仅 `return;` 也就是 `return undefined` 。在终结状态 `*3*` ，有一个 `return false` ，我们保留它。

现在我们需要定义 迭代器 函数的代码，以便人们恰当地调用 `process(..)` ：

```
function foo(url) {
    // 管理 generator 状态
    var state;

    // generator-范围的变量声明
    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;
        }
    }

    // 制造并返回 iterator
    return {
        next: function(v) {
            // 初始状态
            if (!state) {
                state = 1;
                return {
                    done: false,
                    value: process()
                };
            }
            // 成功地让出继续值
            else if (state == 1) {
                state = 2;
```

```
                return {
                    done: true,
                    value: process( v )
                };
            }
            // generator 已经完成了
            else {
                return {
                    done: true,
                    value: undefined
                };
            }
        },
        "throw": function(e) {
            // 在状态 *1* 中，有唯一明确的错误处理
            if (state == 1) {
                state = 3;
                return {
                    done: true,
                    value: process( e )
                };
            }
            // 否则，是一个不会被处理的错误，所以我们仅仅把它扔回去
            else {
                throw e;
            }
        }
    };
}
```

这段代码如何工作？

1. 第一个对 迭代器 的`next()`调用将把gtenerator从未初始化的状态移动到状态`1`，然后调用`process()`来处理这个状态。`request(..)`的返回值是一个代表Ajax应答的promise，它作为`value`属性从`next()`调用被返回。
2. 如果Ajax请求成功，第二个`next(..)`调用应当送进Ajax的应答值，它将我们的状态移动到`2`。`process(..)`再次被调用（这次它被传入Ajax应答的值），而从`next(..)`返回的`value`属性将是`undefined`。
3. 然而，如果Ajax请求失败，应当用错误调用`throw(..)`，它将状态从`1`移动到`3`（而不是`2`）。`process(..)`再一次被调用，这词被传入了错误的值。这个`case`返回`false`，所以`false`作为`throw(..)`调用返回的`value`属性。

从外面看——也就是仅仅与 迭代器 互动——这个普通的`foo(..)`函数与`*foo(..)` generator的工作方式是一样的。所以我们有效地将ES6 generator“转译”为前ES6可兼容的！

然后我们就可以手动初始化我们的generator并控制它的迭代器——调用`var it =`

`foo("..")` 和 `it.next(..)` 等等——或更好地，我们可以将它传递给我们先前定义的 `run(..)` 工具，比如 `run(foo,"..")` 。

自动转译

前面的练习——手动编写从ES6 generator到前ES6的等价物的变形过程——教会了我们generator在概念上是如何工作的。但是这种变形真的是错综复杂，而且不能很好地移植到我们代码中的其他generator上。手动做这些工作是不切实际的，而且将会把generator的好处完全抵消掉。

但走运的是，已经存在几种工具可以自动地将ES6 generator转换为我们在前一节延伸出的东西。它们不仅帮我们做力气活儿，还可以处理几种我们敷衍而过的情况。

一个这样的工具是regenerator（https://facebook.github.io/regenerator/），由Facebook的聪明伙计们开发的。

如果我们用regenerator来转译我们前面的generator，这就是产生的代码（在编写本文时）：

```
// `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]]);
});
```

这和我们的手动推导有明显的相似性，比如 `switch` / `case` 语句，而且我们甚至可以看到， `val` 被拉到了闭包外面，正如我们做的那样。

当然，一个代价是这个generator的转译需要一个帮助工具库 `regeneratorRuntime` ，它持有全部管理一个普通generator/迭代器 所需的可复用逻辑。它的许多模板代码看起来和我们的版本不同，但即便如此，概念还是可以看到的，比如使用 `context$1$0.next = 4` 追踪generator的下一个状态。

主要的结论是，generator不仅限于ES6+的环境中才有用。一旦你理解了它的概念，你可以在你的所有代码中利用他们，并使用工具将代码变形为旧环境兼容的。

这比使用 `Promise` API的填补来实现前ES6的Promise要做更多的工作，但是努力完全是值得的，因为对于以一种可推理的，合理的，看似同步的顺序风格来表达异步流程控制来说，generator实在是好太多了。

一旦你适应了generator，你将永远不会回到面条般的回调地狱了！

复习

复习

generator是一种ES6的新函数类型，它不像普通函数那样运行至完成。相反，generator可以暂停在一种中间完成状态（完整地保留它的状态），而且它可以从暂停的地方重新开始。

这种暂停/继续的互换是一种协作而非抢占，这意味着generator拥有的唯一能力是使用 `yield` 关键字暂停它自己，而且控制这个generator的 迭代器 拥有的唯一能力是继续这个generator（通过 `next(..)` ）。

`yield` / `next(..)` 的对偶不仅是一种控制机制，它实际上是一种双向消息传递机制。一个 `yield ..` 表达式实质上为了等待一个值而暂停，而下一个 `next(..)` 调用将把值（或隐含的 `undefined` ）传递回这个暂停的 `yield` 表达式。

与异步流程控制关联的generator的主要好处是，在一个generator内部的代码以一种自然的同步/顺序风格表达一个任务的各个步骤的序列。这其中的技巧是我们实质上将潜在的异步处理隐藏在 `yield` 关键字的后面—将异步处理移动到控制generator的 迭代器 代码中。

换句话说，generator为异步代码保留了顺序的，同步的，阻塞的代码模式，这允许我们的大脑更自然地推理代码，解决了基于回调的异步产生的两个关键问题中的一个。

第五章: 程序性能

第五章： 程序性能

这本书至此一直是关于如何更有效地利用异步模式。但是我们还没有直接解释为什么异步对于JS如此重要。最明显明确的理由就是 性能。

举个例子，如果你要发起两个Ajax请求，而且他们是相互独立的，但你在进行下一个任务之前需要等到他们全部完成，你就有两种选择来对这种互动建立模型：顺序和并发。

你可以发起第一个请求并等到它完成再发起第二个请求。或者，就像我们在promise和generator中看到的那样，你可以“并列地”发起两个请求，并在继续下一步之前让一个“门”等待它们全部完成。

显然，后者要比前者性能更好。而更好的性能一般都会带来更好的用户体验。

异步（并发穿插）甚至可能仅仅增强高性能的印象，即便整个程序依然要用相同的时间才成完成。用户对性能的印象意味着一切—如果不能再多的话！—和实际可测量的性能一样重要。

现在，我们想超越局部的异步模式，转而在程序级别的水平上讨论一些宏观的性能细节。

注意： 你可能会想知道关于微性能问题，比如 `a++` 与 `++a` 哪个更快。我们会在下一章“基准分析与调优”中讨论这类性能细节。

链接

- Web Workers
- SIMD
- asm.js
- 复习

Web Workers

Web Workers

如果你有一些处理密集型的任务，但你不想让它们在主线程上运行（那样会使浏览器/UI变慢），你可能会希望JavaScript可以以多线程的方式操作。

在第一章中，我们详细地谈到了关于JavaScript如何是单线程的。那仍然是成立的。但是单线程不是组织你程序运行的唯一方法。

想象将你的程序分割成两块儿，在UI主线程上运行其中的一块儿，而在一个完全分离的线程上运行另一块儿。

这样的结构会引发什么我们需要关心的问题？

其一，你会想知道运行在一个分离的线程上是否意味着它在并行运行（在多CPU/内核的系统上），如此在第二个线程上长时间运行的处理将 不会 阻塞主程序线程。否则，“虚拟线程”所带来的好处，不会比我们已经在异步并发的JS中得到的更多。

而且你会想知道这两块儿程序是否访问共享的作用域/资源。如果是，那么你就要对付多线程语言（Java，C++等等）的所有问题，比如协作式或抢占式锁定（互斥，等）。这是很多额外的工作，而且不应当轻易着手。

换一个角度，如果这两块儿程序不能共享作用域/资源，你会想知道它们将如何“通信”。

所有这些我们需要考虑的问题，指引我们探索一个在近HTML5时代被加入web平台的特性，称为“Web Worker”。这是一个浏览器（也就是宿主环境）特性，而且几乎和JS语言本身没有任何关系。也就是说，JavaScript 当前 并没有任何特性可以支持多线程运行。

但是一个像你的浏览器那样的环境可以很容易地提供多个JavaScript引擎实例，每个都在自己的线程上，并允许你在每个线程上运行不同的程序。你的程序中分离的线程块儿中的每一个都称为一个“（Web）Worker”。这种并行机制叫做“任务并行机制”，它强调将你的程序分割成块儿来并行运行。

在你的主JS程序（或另一个Worker）中，你可以这样初始化一个Worker：

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

这个URL应当指向JS文件的位置（不是一个HTML网页！），它将会被加载到一个Worker。然后浏览器会启动一个分离的线程，让这个文件在这个线程上作为独立的程序运行。

注意： 这种用这样的URL创建的Worker称为“专用（Dedicated）Wroker”。但与提供一个外部文件的URL不同的是，你也可以通过提供一个Blob URL（另一个HTML5特性）来创建一个“内联（Inline）Worker”；它实质上是一个存储在单一（二进制）值中的内联文件。但是，Blob超出了我们要在这里讨论的范围。

Worker不会相互，或者与主程序共享任何作用域或资源—那会将所有的多线程编程的噩梦带到我们面前—取而代之的是一种连接它们的基本事件消息机制。

`w1` Worker对象是一个事件监听器和触发器，它允许你监听Worker发出的事件也允许你向Worker发送事件。

这是如何监听事件（实际上，是固定的 `"message"` 事件）：

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

而且你可以发送 `"message"` 事件给Worker:

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

在Worker内部，消息是完全对称的:

```
// "mycoolworker.js"

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

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

要注意的是，一个专用Worker与它创建的程序是一对一的关系。也就是， `"message"` 事件不需要消除任何歧义，因为我们可以确定它只可能来自于这种一对一关系—不是从Wroker来的，就是从主页面来的。

通常主页面的程序会创建Worker，但是一个Worker可以根据需要初始化它自己的子Worker—称为subworker。有时将这样的细节委托给一个“主”Worker十分有用，它可以生成其他Worker来处理任务的一部分。不幸的是，在本书写作的时候，Chrome还没有支持subworker，然而Firefox支持。

要从创建一个Worker的程序中立即杀死它，可以在Worker对象（就像前一个代码段中的 `w1` ）上

调用 `terminate()` 。突然终结一个Worker线程不会给它任何机会结束它的工作，或清理任何资源。这和你关闭浏览器的标签页来杀死一个页面相似。

如果你在浏览器中有两个或多个页面（或者打开同一个页面的多个标签页！），试着从同一个文件URL中创建Worker，实际上最终结果是完全分离的Worker。待一会儿我们就会讨论“共享”Worker的方法。

注意： 看起来一个恶意的或者是呆头呆脑的JS程序可以很容易地通过在系统上生成数百个Worker来发起拒绝服务攻击（Dos攻击），看起来每个Worker都在自己的线程上。虽然一个Worker将会在存在于一个分离的线程上是有某种保证的，但这种保证不是没有限制的。系统可以自由决定有多少实际的线程/CPU/内核要去创建。没有办法预测或保证你能访问多少，虽然很多人假定它至少和可用的CPU/内核数一样多。我认为最安全的臆测是，除了主UI线程外至少有一个线程，仅此而已。

Worker 环境

在Worker内部，你不能访问主程序的任何资源。这意味着你不能访问它的任何全局变量，你也不能访问页面的DOM或其他资源。记住：它是一个完全分离的线程。

然而，你可以实施网络操作（Ajax，WebSocket）和设置定时器。另外，Worker可以访问它自己的几个重要全局变量/特性的拷贝，包括 `navigator` ， `location` ， `JSON` ，和 `applicationCache` 。

你还可以使用 `importScripts(..)` 加载额外的JS脚本到你的Worker中：

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

这些脚本会被同步地加载，这意味着在文件完成加载和运行之前， `importScripts(..)` 调用会阻塞Worker的执行。

注意： 还有一些关于暴露 `<canvas>` API给Worker的讨论，其中包括使canvas成为Transferable的（见“数据传送”一节），这将允许Worker来实施一些精细的脱线程图形处理，在高性能的游戏（WebGL）和其他类似应用中可能很有用。虽然这在任何浏览器中都还不存在，但是很有可能在近未来发生。

Web Worker的常见用途是什么？

- 处理密集型的数学计算
- 大数据集合的排序
- 数据操作（压缩，音频分析，图像像素操作等等）
- 高流量网络通信

数据传送

你可能注意到了这些用途中的大多数的一个共同性质，就是它们要求使用事件机制穿越线程间的壁垒

来传递大量的信息，也许是双向的。

在Worker的早期，将所有数据序列化为字符串是唯一的选择。除了在两个方向上进行序列化时速度上变慢了，另外一个主要缺点是，数据是被拷贝的，这意味着内存用量翻了一倍（以及在后续垃圾回收上的流失）。

谢天谢地，现在我们有了几个更好的选择。

如果你传递一个对象，在另一端一个所谓的“结构化克隆算法（Structured Cloning Algorithm）”（https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/The_structured_clone_algorithm）会用于拷贝/复制这个对象。这个算法相当精巧，甚至可以处理带有循环引用的对象复制。to-string/from-string的性能劣化没有了，但用这种方式我们依然面对着内存用量的翻倍。IE10以上版本，和其他主流浏览器都对此有支持。

一个更好的选择，特别是对大的数据集合而言，是“Transferable对象”（http://updates.html5rocks.com/2011/12/Transferable-Objects-Lightning-Fast）。它使对象的“所有权”被传送，而对象本身没动。一旦你传送一个对象给Worker，它在原来的位置就空了出来或者不可访问—这消除了共享作用域的多线程编程中的灾难。当然，所有权的传送可以双向进行。

选择使用Transferable对象不需要你做太多；任何实现了Transferable接口（https://developer.mozilla.org/en-US/docs/Web/API/Transferable）的数据结构都将自动地以这种方式传递（Firefox和Chrome支持此特性）。

举个例子，有类型的数组如 `Uint8Array`（见本系列的 *ES6与未来*）是一个“Transferables”。这是你如何用 `postMessage(..)` 来传送一个Transferable对象：

```
// `foo` 是一个 `Uint8Array`

postMessage( foo.buffer, [ foo.buffer ] );
```

第一个参数是未经加工的缓冲，而第二个参数是要传送的内容的列表。

不支持Transferable对象的浏览器简单地降级到结构化克隆，这意味着性能上的降低，而不是彻底的特性失灵。

共享的Workers

如果你的网站或应用允许多个标签页加载同一个网页（一个常见的特性），你也许非常想通过防止复制专用Worker来降低系统资源的使用量；这方面最常见的资源限制是网络套接字链接，因为浏览器限制同时连接到一个服务器的连接数量。当然，限制从客户端来的链接数也缓和了你的服务器资源需求。

在这种情况下，创建一个单独的中心化Worker，让你的网站或应用的所有网页实例可以 共享 它是十分有用的。

这称为 `SharedWorker` ，你会这样创建它（仅有Firefox与Chrome支持此特性）：

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

因为一个共享Worker可以连接或被连接到你的网站上的多个程序实例或网页，Worker需要一个方法来知道消息来自哪个程序。这种唯一的标识称为“端口（port）”—联想网络套接字端口。所以调用端程序必须使用Worker的 `port` 对象来通信：

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

// ..

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

另外，端口连接必须被初始化，就像这样：

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

在共享Worker内部，一个额外的事件必须被处理： `"connect"` 。这个事件为这个特定的连接提供端口 `object` 。保持多个分离的连接最简单的方法是在 `port` 上使用闭包，就像下面展示的那样，同时在 `"connect"` 事件的处理器内部定义这个连接的事件监听与传送：

```
// 在共享Worker的内部
addEventListener( "connect", function(evt){
    // 为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();
} );
```

除了这点不同，共享与专用Worker的功能和语义是一样的。

注意： 如果在一个端口的连接终结时还有其他端口的连接存活着的话，共享Worker也会存活下来，而专用Worker会在与初始化它的程序间接终结时终结。

填补 Web Workers

对于并行运行的JS程序在性能考量上，Web Worker十分吸引人。然而，你的代码可能运行在对此缺乏支持的老版本浏览器上。因为Worker是一个API而不是语法，所以在某种程度上它们可以被填补。

如果浏览器不支持Worker，那就根本没有办法从性能的角度来模拟多线程。Iframe通常被认为可以提供并行环境，但在所有的现代浏览器中它们实际上和主页运行在同一个线程上，所以用它们来模拟并行机制是不够的。

正如我们在第一章中详细讨论的，JS的异步能力（不是并行机制）来自于事件轮询队列，所以你可以用计时器（`setTimeout(..)`等等）来强制模拟的Worker是异步的。然后你只需要提供Worker API的填补就行了。这里有一份列表（https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills#web-workers），但坦白地说它们看起来都不怎么样。

我在这里（https://gist.github.com/getify/1b26accb1a09aa53ad25）写了一个填补`Worker`的轮廓。它很基础，但应该满足了简单的`Worker`支持，它的双向信息传递可以正确工作，还有`"onerror"`处理。你可能会扩展它来支持更多特性，比如`terminate()`或模拟共享Worker，只要你觉得合适。

注意： 你不能模拟同步阻塞，所以这个填补不允许使用`importScripts(..)`。另一个选择可能是转换并传递Worker的代码（一旦Ajax加载后），来重写一个`importScripts(..)`填补的一些异步形式，也许使用一个promise相关的接口。

SIMD

SIMD

一个指令，多个数据（SIMD）是一种“数据并行机制”形式，与Web Worker的“任务并行机制”相对应，因为他强调的不是程序逻辑的块儿被并行化，而是多个字节的数据被并行地处理。

使用SIMD，线程不提供并行机制。相反，现代CPU用数字的“向量”提供SIMD能力—想想：指定类型的数组—还有可以在所有这些数字上并行操作的指令；这些是利用底层操作的指令级别的并行机制。

使SIMD能力包含在JavaScript中的努力主要是由Intel带头的（https://01.org/node/1495），名义上是Mohammad Haghighat（在本书写作的时候），与Firefox和Chrome团队合作。SIMD处于早期标准化阶段，而且很有可能被加入未来版本的JavaScript中，很可能在ES7的时间框架内。

SIMD JavaScript提议向JS代码暴露短向量类型与API，它们在SIMD可用的系统中将操作直接映射为CPU指令的等价物，同时在非SIMD系统中退回到非并行化操作的“shim”。

对于数据密集型的应用程序（信号分析，对图形的矩阵操作等等）来说，这种并行数学处理在性能上的优势是十分明显的！

在本书写作时，SIMD API的早期提案形式看起来像这样：

```
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 ]
```

这里展示了两种不同的向量数据类型，32位浮点数和32位整数。你可以看到这些向量正好被设置为4个32位元素，这与大多数CPU中可用的SIMD向量的大小（128位）相匹配。在未来我们看到一个 `x8` （或更大！）版本的这些API也是可能的。

除了 `mul()` 和 `add()` ，许多其他操作也很可能被加入，比如 `sub()` ， `div()` ， `abs()` ， `neg()` ， `sqrt()` ， `reciprocal()` ， `reciprocalSqrt()` （算数运算）， `shuffle()` （重拍向量元素）， `and()` ， `or()` ， `xor()` ， `not()` （逻辑运算）， `equal()` ， `greaterThan()` ， `lessThan()` （比较运算）， `shiftLeft()` ， `shiftRightLogical()` ， `shiftRightArithmetic()` （轮换）， `fromFloat32x4()` ，和 `fromInt32x4()` （变换）。

注意： 这里有一个SIMD功能的官方“填补”（很有希望，预期的，着眼未来的填补）（https://github.com/johnmccutchan/ecmascript_simd ），它描述了许多比我们在这一节中没有讲到的许多计划中的SIMD功能。

asm.js

asm.js

“asm.js”（http://asmjs.org/ ）是可以被高度优化的JavaScript语言子集的标志。通过小心地回避那些特定的很难优化的（垃圾回收，强制转换，等等）机制和模式，asm.js风格的代码可以被JS引擎识别，而且用主动地底层优化进行特殊的处理。

与本章中讨论的其他性能优化机制不同的是，asm.js没必须要是必须被JS语言规范所采纳的东西。确实有一个asm.js规范（http://asmjs.org/spec/latest/ ），但它主要是追踪一组关于优化的候选对象的推论，而不是JS引擎的需求。

目前还没有新的语法被提案。取而代之的是，ams.js建议了一些方法，用来识别那些符合ams.js规则的既存标准JS语法，并且让引擎相应地实现它们自己的优化功能。

关于ams.js应当如何在程序中活动的问题，在浏览器生产商之间存在一些争议。早期版本的asm.js实验中，要求一个 `"use asm";` 编译附注（与strict模式的 `"use strict";` 类似）来帮助JS引擎来寻找asm.js优化的机会和提示。另一些人则断言asm.js应当只是一组启发式算法，让引擎自动地识别而不用作者做任何额外的事情，这意味着理论上既存的程序可以在不用做任何特殊的事情的情况下从asm.js优化中获益。

如何使用 asm.js 进行优化

关于asm.js需要理解的第一件事情是类型和强制转换。如果JS引擎不得不在变量的操作期间一直追踪一个变量内的值的类型，以便于在必要时它可以处理强制转换，那么就会有许多额外的工作使程序处于次优化状态。

注意： 为了说明的目的，我们将在这里使用ams.js风格的代码，但要意识到的是你手写这些代码的情况不是很常见。asm.js的本意更多的是作为其他工具的编译目标，比如Emscripten（https://github.com/kripken/emscripten/wiki ）。当然你写自己的asm.js代码也是可能的，但是这通常不是一个好主意，因为那样的代码非常底层，而这意味着它会非常耗时而且易错。尽管如此，也会有情况使你想要为了ams.js优化的目的手动调整代码。

这里有一些“技巧”，你可以使用它们来提示支持asm.js的JS引擎变量/操作预期的类型是什么，以便于它可以跳过那些强制转换追踪的步骤。

举个例子：

```
var a = 42;

// ..

var b = a;
```

在这个程序中，赋值 `b = a` 在变量中留下了类型分歧的问题。然而，它可以写成这样：

```
var a = 42;

// ..

var b = a | 0;
```

这里，我们与值 `0` 一起使用了 `|` （“二进制或”），虽然它对值没有任何影响，但它确保这个值是一个32位整数。这段代码在普通的JS引擎中可以工作，但是当它运行在支持asm.js的JS引擎上时，它 可以 表示 `b` 应当总是被作为32位整数来对待，所以强制转换追踪可以被跳过。

类似地，两个变量之间的加法操作可以被限定为性能更好的整数加法（而不是浮点数）：

```
(a + b) | 0
```

再一次，支持asm.js的JS引擎可以看到这个提示，并推断 `+` 操作应当是一个32位整数加法，因为不论怎样整个表达式的最终结果都将自动是32位整数。

asm.js 模块

在JS中最托性能后腿的东西之一是关于内存分配，垃圾回收，与作用域访问。asm.js对于这些问题建一个的一个方法是，声明一个更加正式的asm.js“模块”——不要和ES6模块搞混；参见本系列的*ES6与未来*。

对于一个asm.js模块，你需要明确传入一个被严格遵循的名称空间——在规范中以 `stdlib` 引用，因为它应当代表需要的标准库——来引入需要的符号，而不是通过词法作用域来使用全局对象。在最基本的情况下， `window` 对象就是一个可接受的用于asm.js模块的 `stdlib` 对象，但是你可能应该构建一个更加被严格限制的对象。

你还必须定义一个“堆（heap）”——这只是一个别致的词汇，它表示在内存中被保留的位置，变量不必要求内存分配或释放已使用内存就可以使用——并将它传入，这样asm.js模块就不必做任何导致内存流失的的事情；它可以使用提前保留的空间。

一个“堆”就像一个有类型的 `ArrayBuffer` ，比如：

```
var heap = new ArrayBuffer( 0x10000 );    // 64k 的堆
```

使用这个提前保留的64k的二进制空间，一个asm.js模块可以在这个缓冲区中存储或读取值，而不受任何内存分配与垃圾回收的性能损耗。比如， `heap` 缓冲区可以在模块内部用于备份一个64位浮点数值的数组，像这样：

```
var arr = new Float64Array( heap );
```

好了，让我制作一个asm.js风格模块的快速，愚蠢的例子来描述这些东西是如何联系在一起的。我们将定义一个 `foo(..)` ，它为一个范围接收一个开始位置（ `x` ）和一个终止位置（ `y` ），并且计算这个范围内所有相邻的数字的积，然后最终计算这些值的平均值：

```
function fooASM(stdlib,foreign,heap) {
    "use asm";

    var arr = new stdlib.Int32Array( heap );

    function foo(x,y) {
        x = x | 0;
        y = y | 0;

        var i = 0;
        var p = 0;
        var sum = 0;
        var count = ((y|0) - (x|0)) | 0;

        // 计算范围内所有相邻的数字的积
        for (i = x | 0;
            (i | 0) < (y | 0);
            p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            // 存储结果
            arr[ p >> 3 ] = (i * (i + 1)) | 0;
        }

        // 计算所有中间值的平均值
        for (i = 0, p = 0;
            (i | 0) < (count | 0);
            p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            sum = (sum + arr[ p >> 3 ]) | 0;
        }

        return +(sum / count);
    }

    return {
```

```
        foo: foo
    };
}

var heap = new ArrayBuffer( 0x1000 );
var foo = fooASM( window, null, heap ).foo;

foo( 10, 20 );        // 233
```

注意： 这个asm.js例子是为了演示的目的手动编写的，所以它与那些支持asm.js的编译工具生产的代码的表现不同。但是它展示了asm.js代码的典型性质，特别是类型提示与为了临时变量存储而使用 `heap` 缓冲。

第一个 `fooASM(..)` 调用用它的 `heap` 分配区建立了我们的asm.js模块。结果是一个我们可以调用任意多次的 `foo(..)` 函数。这些调用应当会被支持asm.js的JS引擎特别优化。重要的是，前面的代码完全是标准JS，而且会在非asm.js引擎中工作的很好（但没有特别优化）。

很明显，使asm.js代码可优化的各种限制降低了广泛使用这种代码的可能性。对于任意给出的JS程序，asm.js没有必要为成为一个一般化的优化集合。相反，它的本意是提供针对一种处理特定任务—如密集数学操作（那些用于游戏中图形处理的）—的优化方法。

复习

复习

本书的前四章基于这样的前提：异步编码模式给了你编写更高效代码的能力，这通常是一个非常重要的改进。但是异步行为也就能帮你这么多，因为它在基础上仍然使用一个单独的事件轮询线程。

所以在这一章我们涵盖了几种程序级别的机制来进一步提升性能。

Web Worker让你在一个分离的线程上运行一个JS文件（也就是程序），使用异步事件在线程之间传递消息。对于将长时间运行或资源密集型任务挂载到一个不同线程，从而让主UI线程保持相应来说，它们非常棒。

SIMD提议将CPU级别的并行数学操作映射到JavaScript API上来提供高性能数据并行操作，比如在大数据集合上进行数字处理。

最后，asm.js描述了一个JavaScript的小的子集，它回避了JS中不易优化的部分（比如垃圾回收与强制转换）并让JS引擎通过主动优化识别并运行这样的代码。asm.js可以手动编写，但是极其麻烦且易错，就像手动编写汇编语言。相反，asm.js的主要意图是作为一个从其他高度优化的程序语言交叉编译来的目标—例如，Emscripten（https://github.com/kripken/emscripten/wiki）可以将C/C++转译为JavaScript。

虽然在本章没有明确地提及，在很早以前的有关JavaScript的讨论中存在着更激进的想法，包括近似地直接多线程功能（不仅仅是隐藏在数据结构API后面）。无论这是否会明确地发生，还是我们将看到更多并行机制偷偷潜入JS，但是在JS中发生更多程序级别优化的未来是可以确定的。

第六章: 基准分析与调优

第六章： 基准分析与调优

本书的前四章都是关于代码模式（异步与同步）的性能，而第五章是关于宏观的程序结构层面的性能，本章从微观层面继续性能的话题，关注的焦点在一个表达式/语句上。

好奇心最重的一个领域—确实，一些开发者十分痴迷于此—是分析和测试如何写一行或一块儿代码的各种选项，看哪一个更快。

我们将会看到这些问题中的一些，但重要的是要理解从最开始这一章就 不是 为了满足对微性能调优的痴迷，比如某种给定的JS引擎运行 `++a` 是否要比运行 `a++` 快。这一章更重要的目标是，搞清楚哪种JS性能要紧而哪种不要紧，和如何指出这种不同。

但在我们达到目的之前，我们需要探索一下如何最准确和最可靠地测试JS性能，因为有太多的误解和谜题充斥着我们集体主义崇拜的知识库。我们需要将这些垃圾筛出去以便找到清晰的答案。

链接

- 基准分析（Benchmarking）
- 上下文为王
- jsPerf.com
- 编写好的测试
- 微观性能
- 尾部调用优化 （TCO）
- 复习

基准分析（Benchmarking）

基准分析（Benchmarking）

好了，是时候开始消除一些误解了。我敢打赌，广大的JS开发者们，如果被问到如何测量一个特定操作的速度（执行时间），将会一头扎进这样的东西：

```
var start = (new Date()).getTime();    // 或者`Date.now()`

// 做一些操作

var end = (new Date()).getTime();

console.log( "Duration:", (end - start) );
```

如果这大致就是你想到的，请举手。是的，我就知道你会这么想。这个方式有许多错误，但是别难过；我们都这么干过。

这种测量到底告诉了你什么？对于当前的操作的执行时间来说，理解它告诉了你什么和没告诉你什么是学习如何正确测量JavaScript的性能的关键。

如果持续的时间报告为 `0`，你也许会试图认为它花的时间少于1毫秒。但是这不是非常准确。一些平台不能精确到毫秒，反而是在更大的时间单位上更新计时器。举个例子，老版本的windows（IE也是如此）只有15毫秒的精确度，这意味着要得到与 `0` 不同的报告，操作就必须至少要花这么长时间！

另外，不管被报告的持续时间是多少，你唯一真实知道的是，操作在当前这一次运行中大概花了这么长时间。你几乎没有信心说它将总是以这个速度运行。你不知道引擎或系统是否在就在那个确切的时刻进行了干扰，而在其他的时候这个操作可能会运行的快一些。

要是持续的时间报告为 `4` 呢？你确信它花了大概4毫秒？不，它可能没花那么长时间，而且在取得 `start` 或 `end` 时间戳时会有一些其他的延迟。

更麻烦的是，你也不知道这个操作测试所在的环境是不是过于优化了。这样的情况是有可能的：JS引擎找到了一个办法来优化你的测试用例，但是在更真实的程序中这样的优化将会被稀释或者根本不可能，如此这个操作将会比你测试时运行的慢。

那么…我们知道什么？不幸的是，在这种状态下，我们几乎什么都不知道。 可信度如此低的东西甚至不够你建立自己的判断。你的“基准分析”基本没用。更糟的是，它隐含的这种不成立的可信度很危险，不仅是对你，而且对其他人也一样：认为导致这些结果的条件不重要。

重复

“好的，”你说，“在它周围放一个循环，让整个测试需要的时间长一些。”如果你重复一个操作100次，而整个循环在报告上说总共花了137ms，那么你可以除以100并得到每次操作平均持续时间1.37ms，对吧？

其实，不确切。

对于你打算在你的整个应用程序范围内推广的操作的性能，仅靠一个直白的数据上的平均做出判断绝对是不够的。在一百次迭代中，即使是几个极端值（或高或低）就可以歪曲平均值，而后当你反复实施这个结论时，你就更进一步扩大了这种歪曲。

与仅仅运行固定次数的迭代不同，你可以选择将测试的循环运行一个特定长的时间。那可能更可靠，但是你如何决定运行多长时间？你可能会猜它应该是你的操作运行一次所需时间的倍数。错。

实际上，循环持续的时间应当基于你使用的计时器的精度，具体地将不精确的 ·可能性最小化。你的计时器精度越低，你就需要运行更长时间来确保你将错误的概率最小化了。一个15ms的计时器对于精确的基准分析来说太差劲儿了；为了把它的不确定性（也就是“错误率”）最小化到低于1%，你需要将测试的迭代循环运行750ms。一个1ms的计时器只需要一个循环运行50ms就可以得到相同的可信度。

但，这只是一个样本。为了确信你排除了歪曲结果的因素，你将会想要许多样本来求平均值。你还会想要明白最差的样本有多慢，最佳的样本有多快，最差与最佳的情况相差多少等等。你想知道的不仅是一个数字告诉你某个东西跑的多块，而且还需要一个关于这个数字有多可信的量化表达。

另外，你可能想要组合这些不同的技术（还有其他的），以便于你可以在所有这些可能的方式中找到最佳的平衡。

这一切只不过是开始所需的最低限度的认识。如果你曾经使用比我刚才几句话带过的东西更不严谨的方式进行基准分析，那么…“你不懂：正确的基准分析”。

Benchmark.js

任何有用而且可靠的基准分析应当基于统计学上的实践。我不是要在这里写一章统计学，所以我会带过一些名词：标准差，方差，误差边际。如果你不知道这些名词意味着什么—我在大学上过统计学课程，而我依然对他们有点儿晕—那么实际上你没有资格去写你自己的基准分析逻辑。

幸运的是，一些像John-David Dalton和Mathias Bynens这样的聪明家伙明白这些概念，并且写了一个统计学上的基准分析工具，称为Benchmark.js（http://benchmarkjs.com/）。所以我可以简单地说：“用这个工具就行了。”来终结这个悬念。

我不会重复他们的整个文档来讲解Benchmark.js如何工作；他们有很棒的API文档（http://benchmarkjs.com/docs）你可以阅读。另外这里还有一些了不起的文章（http://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks/）（http://monsur.hossa.in/2012/12/11/benchmarkjs.html）讲解细节与方法学。

但是为了快速演示一下，这是你如何用Benchmark.js来运行一个快速的性能测试：

```
function foo() {
    // 需要测试的操作
}

var bench = new Benchmark(
    "foo test",                 // 测试的名称
    foo,                       // 要测试的函数（仅仅是内容）
    {
        // ..                   // 额外的选项（参见文档）
    }
);

bench.hz;                      // 每秒钟执行的操作数
bench.stats.moe;              // 误差边际
bench.stats.variance;          // 所有样本上的方差
// ..
```

比起我在这里的窥豹一斑，关于使用Benchmark.js还有 许多 需要学习的东西。不过重点是，为了给一段给定的JavaScript代码建立一个公平，可靠，并且合法的性能基准分析，Benchmark.js包揽了所有的复杂性。如果你想要试着对你的代码进行测试和基准分析，这个库应当是你第一个想到的地方。

我们在这里展示的是测试一个单独操作X的用法，但是相当常见的情况是你想要用X和Y进行比较。这可以通过简单地在一个“Suite”（一个Benchmark.js的组织特性）中建立两个测试来很容易做到。然后，你对照地运行它们，然后比较统计结果来对为什么X或Y更快做出论断。

Benchmark.js理所当然地可以被用于在浏览器中测试JavaScript（参见本章稍后的“jsPerf.com”一节），但它也可以运行在非浏览器环境中（Node.js等等）。

一个很大程度上没有触及的Benchmark.js的潜在用例是，在你的Dev或QA环境中针对你的应用程序的JavaScript的关键路径运行自动化的性能回归测试。与在部署之前你可能运行单元测试的方式相似，你也可以将性能与前一次基准分析进行比较，来观测你是否改进或恶化了应用程序性能。

Setup/Teardown

在前一个代码段中，我们略过了“额外选项（extra options）” `{ .. }` 对象。但是这里有两个我们应当讨论的选项 `setup` 和 `teardown` 。

这两个选项让你定义在你的测试用例开始运行前和运行后被调用的函数。

一个需要理解的极其重要的事情是，你的 `setup` 和 `teardown` 代码 不会为每一次测试迭代而运行。考虑它的最佳方式是，存在一个外部循环（重复的轮回），和一个内部循环（重复的测试迭代）。 `setup` 和 `teardown` 会在每个 外部 循环（也就是轮回）迭代的开始和末尾运行，但不是在内部循环。

为什么这很重要？让我们想象你有一个看起来像这样的测试用例：

```
1. a = a + "w";
2. b = a.charAt( 1 );
```

然后，你这样建立你的测试 `setup` ：

```
1. var a = "x";
```

你的意图可能是相信对每一次测试迭代 `a` 都以值 `"x"` 开始。

但它不是！它使 `a` 在每一次测试轮回中以 `"x"` 开始，而后你的反复的 `+ "w"` 连接将使 `a` 的值越来越大，即便你永远唯一访问的是位于位置 `1` 的字符 `"w"` 。

当你想利用副作用来改变某些东西比如DOM，向它追加一个子元素时，这种意外经常会咬到你。你可能认为的父元素每次都被设置为空，但他实际上被追加了许多元素，而这可能会显著地歪曲你的测试结果。

上下文为王

上下文为王

不要忘了检查一个指定的性能基准分析的上下文环境，特别是在X与Y之间进行比较时。仅仅因为你的测试显示X比Y速度快，并不意味着“X比Y快”这个结论是实际上有意义的。

举个例子，让我们假定一个性能测试显示出X每秒可以运行1千万次操作，而Y每秒运行8百万次。你可以声称Y比X慢20%，而且在数学上你是对的，但是你的断言并不向像你认为的那么有用。

让我们更加苛刻地考虑这个测试结果：每秒1千万次操作就是每毫秒1万次操作，就是每微秒10次操作。换句话说，一次操作要花0.1毫秒，或者100纳秒。很难体会100纳秒到底有多小，可以这样比较一下，通常认为人类的眼睛一般不能分辨小于100毫秒的变化，而这要比X操作的100纳秒的速度慢100万倍。

即便最近的科学研究显示，大脑可能的最快处理速度是13毫秒（比先前的论断快大约8倍），这意味着X的运行速度依然要比人类大脑可以感知事情的发生要快12万5千倍。**X**运行的非常，非常快。

但更重要的是，让我们来谈谈X与Y之间的不同，每秒2百万次的差。如果X花100纳秒，而Y花80纳秒，差就是20纳秒，也就是人类大脑可以感知的间隔的65万分之一。

我要说什么？这种性能上的差别根本就一点儿都不重要！

但是等一下，如果这种操作将要一个接一个地发生许多次呢？那么差异就会累加起来，对吧？

好的，那么我们就要问，操作X有多大可能性将要一次又一次，一个接一个地运行，而且为了人类大脑能够感知的一线希望而不得不发生65万次。而且，它不得不在一个紧凑的循环中发生5百万到1千万次，才能接近于有意义。

虽然你们之中的计算机科学家会反对说这是可能的，但是你们之中的现实主义者们应当对这究竟有多大可能性进行可行性检查。即使在极其稀少的偶然中这有实际意义，但是在绝大多数情况下它没有。

你们大量的针对微小操作的基准分析结果—比如 `++x` 对 `x++` 的神话—完全是伪命题，只不过是用来支持在性能的基准上X应当取代Y的结论。

引擎优化

你根本无法可靠地这样推断：如果在你的独立测试中X要比Y快10微秒，这意味着X总是比Y快所以应当总是被使用。这不是性能的工作方式。它要复杂太多了。

举个例子，让我们想象（纯粹地假想）你在测试某些行为的微观性能，比如比较：

```
var twelve = "12";
var foo = "foo";

// 测试 1
var X1 = parseInt( twelve );
var X2 = parseInt( foo );

// 测试 2
var Y1 = Number( twelve );
var Y2 = Number( foo );
```

如果你明白与 `Number(..)` 比起来 `parseInt(..)` 做了什么，你可能会在直觉上认为 `parseInt(..)` 潜在地有“更多工作”要做，特别是在 `foo` 的测试用例下。或者你可能在直觉上认为在 `foo` 的测试用例下它们应当有同样多的工作要做，因为它们俩应当能够在第一个字符 `"f"` 处停下。

哪一种直觉正确？老实说我不知道。但是我会制造一个与你的直觉无关的测试用例。当你测试它的时候结果会是什么？我又一次在这里制造一个纯粹的假想，我们没实际上尝试过，我也不关心。

让我们假装 `X` 与 `Y` 的测试结果在统计上是相同的。那么你关于 `"f"` 字符上发生的事情的直觉得到确认了吗？没有。

在我们的假想中可能发生这样的事情：引擎可能会识别出变量 `twelve` 和 `foo` 在每个测试中仅被使用了一次，因此它可能会决定要内联这些值。然后它可能发现 `Number("12")` 可以替换为 `12`。而且也许在 `parseInt(..)` 上得到相同的结论，也许不会。

或者一个引擎的死代码移除启发式算法会搅和进来，而且它发现变量 `X` 和 `Y` 都没有被使用，所以声明它们是没有意义的，所以最终在任一个测试中都不做任何事情。

而且所有这些都只是关于一个单独测试运行的假设而言的。比我们在这里用直觉想象的，现代的引擎复杂得更加难以置信。它们会使用所有的招数，比如追踪并记录一段代码在一段很短的时间内的行为，或者使用一组特别限定的输入。

如果引擎由于固定的输入而用特定的方法进行了优化，但是在你的真实的程序中你给出了更多种类的输入，以至于优化机制决定使用不同的方式呢（或者根本不优化！）？或者如果因为引擎看到代码被基准分析工具运行了成千上万次而进行了优化，但在你的真实程序中它将仅会运行大约100次，而在这些条件下引擎认定优化不值得呢？

所有这些我们刚刚假想的优化措施可能会发生在我们的被限定的测试中，但在更复杂的程序中引擎可能不会那么做（由于种种原因）。或者正相反—引擎可能不会优化这样不起眼的代码，但是可能会更倾向于在系统已经被一个更精巧的程序消耗后更加积极地优化。

我想要说的是，你不能确切地知道这背后究竟发生了什么。你能搜罗的所有猜测和假想几乎不会提炼

成任何坚实的依据。

难道这意味着你不能真正地做有用的测试了吗？绝对不是！

这可以归结为测试 不真实 的代码会给你 不真实 的结果。在尽可能的情况下，你应当测试真实的，有意义的代码段，并且在最接近你实际能够期望的真实条件下进行。只有这样你得到的结果才有机会模拟现实。

像 `++x` 和 `x++` 这样的微观基准分析简直和伪命题一模一样，我们也许应该直接认为它就是。

jsPerf.com

jsPerf.com

虽然Bechmark.js对于在你使用的任何JS环境中测试代码性能很有用，但是如果你需要从许多不同的环境（桌面浏览器，移动设备等）汇总测试结果并期望得到可靠的测试结论，它就显得能力不足。

举例来说，Chrome在高端的桌面电脑上与Chrome移动版在智能手机上的表现就大相径庭。而一个充满电的智能手机与一个只剩2%电量，设备开始降低无线电和处理器的能源供应的智能手机的表现也完全不同。

如果在横跨多于一种环境的情况下，你想在任何合理的意义上宣称“X比Y快”，那么你就需要实际测试尽可能多的真实世界的环境。只因为Chrome执行某种X操作比Y快并不意味着所有的浏览器都是这样。而且你还可能想要根据你的用户的人口统计交叉参照多种浏览器测试运行的结果。

有一个为此目的而生的牛X网站，称为jsPerf（http://jsperf.com ）。它使用我们前面提到的Benchmark.js库来运行统计上正确且可靠的测试，并且可以让测试运行在一个你可交给其他人的公开URL上。

每当一个测试运行后，其结果都被收集并与这个测试一起保存，同时累积的测试结果将在网页上被绘制成图供所有人阅览。

当在这个网站上创建测试时，你一开始有两个测试用例可以填写，但你可以根据需要添加任意多个。你还可以建立在每次测试轮回开始时运行的 `setup` 代码，和在每次测试轮回结束前运行的 `teardown` 代码。

注意： 一个只做一个测试用例（如果你只对一个方案进行基准分析而不是相互对照）的技巧是，在第一次创建时使用输入框的占位提示文本填写第二个测试输入框，之后编辑这个测试并将第二个测试留为空白，这样它就会被删除。你可以稍后添加更多测试用例。

你可以顶一个页面的初始配置（引入库文件，定义工具函数，声明变量，等等）。如有需要这里也有选项可以定义setup和teardow行为—参照前面关于Benchmark.js的讨论中的“Setup/Teardown”一节。

可行性检查

jsPerf是一个奇妙的资源，但它上面有许多公开的糟糕测试，当你分析它们时会发现，由于在本章目前为止罗列的各种原因，它们有很大的漏洞或者是伪命题。

考虑：

```
// 用例 1
var x = [];
for (var i=0; i<10; i++) {
    x[i] = "x";
}

// 用例 2
var x = [];
for (var i=0; i<10; i++) {
    x[x.length] = "x";
}

// 用例 3
var x = [];
for (var i=0; i<10; i++) {
    x.push( "x" );
}
```

关于这个测试场景有一些现象值得我们深思：

- 开发者们在测试用例中加入自己的循环极其常见，而他们忘记了Benchmark.js已经做了你所需要的所有反复。这些测试用例中的`for`循环有很大的可能是完全不必要的噪音。
- 在每一个测试用例中都包含了 `x` 的声明与初始化，似乎是不必要的。回想早前如果 `x = []` 存在于 `setup` 代码中，它实际上不会在每一次测试迭代前执行，而是在每一个轮回的开始执行一次。这意味这 `x` 将会持续地增长到非常大，而不仅是 `for` 循环中暗示的大小 `10` 。

  那么这是有意确保测试仅被限制在很小的数组上（大小为 `10` ）来观察JS引擎如何动作？这 可能 是有意的，但如果是，你就不得不考虑它是否过于关注内微妙的部实现细节了。

  另一方面，这个测试的意图包含数组实际上会增长到非常大的情况吗？JS引擎对大数组的行为与真实世界中预期的用法相比有意义且正确吗？
- 它的意图是要找出 `x.length` 或 `x.push(..)` 在数组 `x` 的追加操作上拖慢了多少性能吗？好吧，这可能是一个合法的测试。但再一次， `push(..)` 是一个函数调用，所以它理所当然地要比 `[..]` 访问慢。可以说，用例1与用例2比用例3更合理。

这里有另一个展示苹果比橘子的常见漏洞的例子：

```
// 用例 1
var x = ["John","Albert","Sue","Frank","Bob"];
x.sort();

// 用例 2
```

```
var x = ["John","Albert","Sue","Frank","Bob"];
x.sort( function mySort(a,b){
    if (a < b) return -1;
    if (a > b) return 1;
    return 0;
} );
```

这里，明显的意图是要找出自定义的 `mySort(..)` 比较器比内建的默认比较器慢多少。但是通过将函数 `mySort(..)` 作为内联的函数表达式生命，你就创建了一个不合理的/伪命题的测试。这里，第二个测试用例不仅测试用户自定义的JS函数，而且它还测试为每一个迭代创建一个新的函数表达式。

不知这会不会吓到你，如果你运行一个相似的测试，但是将它更改为比较内联函数表达式与预先声明的函数，内联函数表达式的创建可能要慢2%到20%！

除非你的测试的意图 就是 要考虑内联函数表达式创建的“成本”，一个更好/更合理的测试是将 `mySort(..)` 的声明放在页面的setup中——不要放在测试的 `setup` 中，因为这会为每次轮回进行不必要的重复声明——然后简单地在测试用例中通过名称引用它： `x.sort(mySort)` 。

基于前一个例子，另一种造成苹果比橘子场景的陷阱是，不透明地对一个测试用例回避或添加“额外的工作”：

```
// 用例 1
var x = [12,-14,0,3,18,0,2.9];
x.sort();

// 用例 2
var x = [12,-14,0,3,18,0,2.9];
x.sort( function mySort(a,b){
    return a - b;
} );
```

将先前提到的内联函数表达式陷阱放在一边不谈，第二个用例的 `mySort(..)` 可以在这里工作是因为你给它提供了一组数字，而在字符串的情况下肯定会失败。第一个用例不会扔出错误，但是它的实际行为将会不同而且会有不同的结果！这应当很明显，但是：两个测试用例之间结果的不同，几乎可以否定了整个测试的合法性！

但是除了结果的不同，在这个用例中，内建的 `sort(..)` 比较器实际上要比 `mySort()` 做了更多“额外的工作”，内建的比较器将被比较的值转换为字符串，然后进行字典顺序的比较。这样第一个代码段的结果为 `[-14, 0, 0, 12, 18, 2.9, 3]` 而第二段代码的结果为 `[-14, 0, 0, 2.9, 3, 12, 18]` （就测试的意图来讲可能更准确）。

所以这个测试是不合理的，因为它的两个测试用例实际上没有做相同的任务。你得到的任何结果都将是伪命题。

这些同样的陷阱可以微妙的多：

```
// 用例 1
var x = false;
var y = x ? 1 : 2;

// 用例 2
var x;
var y = x ? 1 : 2;
```

这里的意图可能是要测试如果 `x` 表达式不是Boolean的情况下， `? :` 操作符将要进行的Boolean转换对性能的影响（参见本系列的 类型与文法）。那么，根据在第二个用例中将会有额外的工作进行转换的事实，你看起来没问题。

微妙的问题呢？你在第一个测试用例中设定了 `x` 的值，而没在另一个中设置，那么你实际上在第一个用例中做了在第二个用例中没做的工作。为了消灭任何潜在的扭曲（尽管很微小），可以这样：

```
// 用例 1
var x = false;
var y = x ? 1 : 2;

// 用例 2
var x = undefined;
var y = x ? 1 : 2;
```

现在两个用例都有一个赋值了，这样你想要测试的东西— `x` 的转换或者不转换—会更加正确的被隔离并测试。

编写好的测试

编写好的测试

来看看我能否清晰地表达我想在这里申明的更重要的事情。

好的测试作者需要细心地分析性地思考两个测试用例之间存在什么样的差别，和它们之间的差别是否是 有意的 或 无意的。

有意的差别当然是正常的，但是产生歪曲结果的无意的差异实在太容易了。你不得不非常非常小心地回避这种歪曲。另外，你可能预期一个差异，但是你的意图是什么对于你的测试的其他读者来讲不那么明显，所以他们可能会错误地怀疑（或者相信！）你的测试。你如何搞定这个呢？

编写更好，更清晰的测试。 另外，花些时间用文档确切地记录下你的测试意图是什么（使用jsPerf.com的“Description”字段，或/和代码注释），即使是微小的细节。明确地表示有意的差别，这将帮助其他人和未来的你自己更好地找出那些可能歪曲测试结果的无意的差别。

将与你的测试无关的东西隔离开来，通过在页面或测试的setup设置中预先声明它们，使它们位于测试计时部分的外面。

与将你的真实代码限制在很小的一块，并脱离上下文环境来进行基准分析相比，测试与基准分析在它们包含更大的上下文环境（但仍然有意义）时表现更好。这些测试将会趋向于运行得更慢，这意味着你发现的任何差别都在上下文环境中更有意义。

微观性能

微观性能

好了，直至现在我们一直围绕着微观性能的问题跳舞，并且一般上不赞成痴迷于它们。我想花一点儿时间直接解决它们。

当你考虑对你的代码进行性能基准分析时，第一件需要习惯的事情就是你写的代码不总是引擎实际运行的代码。我们在第一章中讨论编译器的语句重排时简单地看过这个话题，但是这里我们将要说明编译器能有时决定运行与你编写的不同的代码，不仅是不同的顺序，而是不同的替代品。

让我们考虑这段代码：

```
var foo = 41;

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

你也许会认为在最里面的函数的 `foo` 引用需要做一个三层作用域查询。我们在这个系列丛书的 作用域与闭包 一卷中涵盖了词法作用域如何工作，而事实上编译器通常缓存这样的查询，以至于从不同的作用域引用 `foo` 不会实质上“花费”任何额外的东西。

但是这里有些更深刻的东西需要思考。如果编译器认识到 `foo` 除了这一个位置外没有被任何其他地方引用，进而注意到它的值除了这里的 `41` 外没有任何变化会怎么样呢？

JS编译器能够决定干脆完全移除 `foo` 变量，并 内联 它的值是可能和可接受的，比如这样：

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

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

注意： 当然，编译器可能也会对这里的 `baz` 变量进行相似的分析和重写。

但你开始将你的JS代码作为一种告诉引擎去做什么的提示或建议来考虑，而不是一种字面上的需求，你就会理解许多对零碎的语法细节的痴迷几乎是毫无根据的。

另一个例子：

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

factorial( 5 );        // 120
```

啊，一个老式的“阶乘”算法！你可能会认为JS引擎将会原封不动地运行这段代码。老实说，它可能会—但我不是很确定。

但作为一段轶事，用C语言表达的同样的代码并使用先进的优化处理进行编译时，将会导致编译器认为 `factorial(5)` 调用可以被替换为常数值 `120` ，完全消除这个函数以及调用！

另外，一些引擎有一种称为“递归展开（unrolling recursion）”的行为，它会意识到你表达的递归实际上可以用循环“更容易”（也就是更优化地）地完成。前面的代码可能会被JS引擎 重写 为：

```
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，让我们想象在前一个片段中你曾经担心 `n * factorial(n-1)` 或 `n *= factorial(--n)` 哪一个运行的更快。也许你甚至做了性能基准分析来试着找出哪个更好。但是你忽略了一个事实，就是在更大的上下文环境中，引擎也许不会运行任何一行代码，因为它可能展开了递归！

说到 `--` ， `--n` 与 `n--` 的对比，经常被认为可以通过选择 `--n` 的版本进行优化，因为理论上在汇编语言层面的处理上，它要做的努力少一些。

在现代的JavaScript中这种痴迷基本上是没道理的。这种事情应当留给引擎来处理。你应该编写最合理的代码。比较这三个 `for` 循环：

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

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

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

就算你有一些理论支持第二或第三种选择要比第一种的性能好那么一点点，充其量只能算是可疑，第三个循环更加使人困惑，因为为了使提前递增的 `++i` 被使用，你不得不让 `i` 从 `-1` 开始来计算。而第一个与第二个选择之间的区别实际上无关紧要。

这样的事情是完全有可能的：JS引擎也许看到一个 `i++` 被使用的地方，并意识到它可以安全地替换为等价的 `++i` ，这意味着你决定挑选它们中的哪一个所花的时间完全被浪费了，而且这么做的产出毫无意义。

这是另外一个常见的愚蠢的痴迷于微观性能的例子：

```
var x = [ .. ];

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

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

这里的理论是，你应当在变量 `len` 中缓存数组 `x` 的长度，因为从表面上看它不会改变，来避免在循环的每一次迭代中都查询 `x.length` 所花的开销。

如果你围绕 `x.length` 的用法进行性能基准分析，与将它缓存在变量 `len` 中的用法进行比较，你会发现虽然理论听起来不错，但是在实践中任何测量出的差异都是在统计学上完全没有意义的。

事实上，在像v8这样的引擎中，可以看到(http://mrale.ph/blog/2014/12/24/array-length-caching.html)通过提前缓存长度而不是让引擎帮你处理它会使事情稍稍恶化。不要尝试在聪明上战胜你的JavaScript引擎，当它来到性能优化的地方时你可能会输给它。

不是所有的引擎都一样

在各种浏览器中的不同JS引擎可以称为“规范兼容的”，虽然各自有完全不同的方式处理代码。JS语言规范不要求与性能相关的任何事情—除了将在本章稍后将要讲解的ES6“尾部调用优化（Tail Call Optimization）”。

引擎可以自由决定哪一个操作将会受到它的关注而被优化，也许代价是在另一种操作上的性能降低一些。要为一种操作找到一种在所有的浏览器中总是运行的更快的方式是非常不现实的。

在JS开发者社区的一些人发起了一项运动，特别是那些使用Node.js工作的人，去分析v8 JavaScript引擎的具体内部实现细节，并决定如何编写定制的JS代码来最大限度的利用v8的工作方式。通过这样的努力你实际上可以在性能优化上达到惊人的高度，所以这种努力的收益可能十分高。

一些针对v8的经常被引用的例子是(https://github.com/petkaantonov/bluebird/wiki/Optimization-killers)：

- 不要将`arguments`变量从一个函数传递到任何其他函数中，因为这样的“泄露”放慢了函数实现。
- 将一个`try..catch`隔离到它自己的函数中。浏览器在优化任何含有`try..catch`的函数时都会苦苦挣扎，所以将这样的结构移动到它自己的函数中意味着你持有不可优化的危害的同时，让其周围的代码是可以优化的。

但与其聚焦在这些具体的窍门上，不如让我们在一般意义上对v8专用的优化方式进行一下合理性检验。

你真的在编写仅仅需要在一种JS引擎上运行的代码吗？即便你的代码 当前 是完全为了Node.js，那么假设v8将 总是 被使用的JS引擎可靠吗？从现在开始的几年以后的某一天，你有没有可能会选择除了Node.js之外的另一种服务器端JS平台来运行你的程序？如果你以前所做的优化现在在新的引擎上成为了执行这种操作的很慢的方式怎么办？

或者如果你的代码总是在v8上运行，但是v8在某个时点决定改变一组操作的工作方式，是的曾经快的现在变慢了，曾经慢的变快了呢？

这些场景也都不只是理论上的。曾经，将多个字符串值放在一个数组中然后在这个数组上调用`join("")`来连接这些值，要比仅使用`+`直接连接这些值要快。这件事的历史原因很微妙，但它与字符串值如何被存储和在内存中如何管理的内部实现细节有关。

结果，当时在业界广泛传播的“最佳实践”建议开发者们总是使用数组`join(..)`的方式。而且有许多人遵循了。

但是，某一天，JS引擎改变了内部管理字符串的方式，而且特别在`+`连接上做了优化。他们并没有放慢`join(..)`，但是他们在帮助`+`用法上做了更多的努力，因为它依然十分普遍。

注意： 某些特定方法的标准化和优化的实施，很大程度上决定于它被使用的广泛程度。这经常（隐喻地）称为“paving the cowpath”（不提前做好方案，而是等到事情发生了再去应对）。

一旦处理字符串和连接的新方式定型，所有在世界上运行的，使用数组 `join(..)` 来连接字符串的代码都不幸地变成了次优的方式。

另一个例子：曾经，Opera浏览器在如何处理基本包装对象的封箱/拆箱（参见本系列的 类型与文法）上与其他浏览器不同。因此他们给开发者的建议是，如果一个原生 `string` 值的属性（如 `length` ）或方法（如 `charAt(..)` ）需要被访问，就使用一个 `String` 对象取代它。这个建议也许对那时的Opera是正确的，但是对于同时代的其他浏览器来说简直就是完全相反的，因为它们都对原生 `string` 进行了专门的优化，而不是对它们的包装对象。

我认为即使是对今天的代码，这种种陷阱即便可能性不高，至少也是可能的。所以对于在我的JS代码中单纯地根据引擎的实现细节来进行大范围的优化这件事来说我会非常小心，特别是如果这些细节仅对一种引擎成立时。

反过来也有一些事情需要警惕：你不应当为了绕过某一种引擎难于处理的地方而改变一块代码。

历史上，IE是导致许多这种挫折的领头羊，在老版本的IE中曾经有许多场景，在当时的其他主流浏览器中看起来没有太多麻烦的性能方面苦苦挣扎。我们刚刚讨论的字符串连接在IE6和IE7的年代就是一个真实的问题，那时候使用 `join(..)` 就可能要比使用 `+` 能得到更好的性能。

不过为了一种浏览器的性能问题而使用一种很有可能在其他所有浏览器上是次优的编码方式，很难说是正当的。即便这种浏览器占有了你的网站用户的很大市场份额，编写恰当的代码并仰仗浏览器最终在更好的优化机制上更新自己可能更实际。

“没什么是比暂时的黑科技更永恒的。”你现在为了绕过一些性能的Bug而编写的代码可能要比这个Bug在浏览器中存在的时间长的多。

在那个浏览器每五年才更新一次的年代，这是个很难做的决定。但是如今，所有的浏览器都在快速地更新（虽然移动端的世界还有些滞后），而且它们都在竞争而使得web优化特性变得越来越好。

如果你真的碰到了一个浏览器有其他浏览器没有的性能瑕疵，那么就确保用你一切可用的手段来报告它。绝大多数浏览器都有为此而公开的Bug追迹系统。

提示： 我只建议，如果一个在某种浏览器中的性能问题真的是极端搅局的问题时才绕过它，而不是仅仅因为它使人厌烦或沮丧。而且我会非常小心地检查这种性能黑科技有没有在其他浏览器中产生负面影响。

大局

与担心所有这些微观性能的细节相反，我们应但关注大局类型的优化。

你怎么知道什么东西是不是大局的？你首先必须理解你的代码是否运行在关键路径上。如果它没在关键路径上，你的优化可能就没有太大价值。

“这是过早的优化！”你听过这种训诫吗？它源自Donald Knuth的一段著名的话：“过早的优化是万恶之源。”。许多开发者都引用这段话来说明大多数优化都是“过早”的而且是一种精力的浪费。事实是，像往常一样，更加微妙。

这是Knuth在语境中的原话：

> 程序员们浪费了大量的时间考虑，或者担心，他们的程序中的 不关键 部分的速度，而在考虑调试和维护时这些在效率上的企图实际上有很强大的负面影响。我们应当忘记微小的效率，可以说在大概*97%*的情况下：过早的优化是万恶之源。然而我们不应该忽略那 关键的 *3%*中的机会。*[强调]*

(http://web.archive.org/web/20130731202547/http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf, Computing Surveys, Vol 6, No 4, December 1974)

我相信这样转述Knuth的 意思 是合理的：“非关键路径的优化是万恶之源。”所以问题的关键是弄清楚你的代码是否在关键路径上—你应该优化它！—或者不。

我甚至可以激进地这么说：没有花在优化关键路径上的时间是浪费的，不管它的效果多么微小。没有花在优化非关键路径上的时间是合理的，不管它的效果多么大。

如果你的代码在关键路径上，比如将要一次又一次被运行的“热”代码块儿，或者在用户将要注意到的UX关键位置，比如循环动画或者CSS样式更新，那么你应当不遗余力地进行有意义的，可测量的重大优化。

举个例子，考虑一个动画循环的关键路径，它需要将一个字符串值转换为一个数字。这当然有多种方法做到，但是哪一个是最快的呢？

```
var x = "42";    // 需要数字 `42`

// 选择1：让隐式强制转换自动完成工作
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`+`二元操作符
var y = +x / 2;

// 选择5：使用`|`二元操作符
var y = (x | 0) / 2;
```

注意： 我将这个问题留作给读者们的练习，如果你对这些选择之间性能上的微小区别感兴趣的话，可以做一个测试。

当你考虑这些不同的选择时，就像人们说的，“有一个和其他的不一样。” `parseInt(..)` 可以工作，但它做的事情多的多—它会解析字符串而不是转换它。你可能会正确地猜想 `parseInt(..)` 是一个更慢的选择，而你可能应当避免使用它。

当然，如果 `x` 可能是一个 需要被解析 的值，比如 `"42px"` （比如CSS样式查询），那么 `parseInt(..)` 确实是唯一合适的选择！

`Number(..)` 也是一个函数调用。从行为的角度讲，它与 `+` 二元操作符是相同的，但它事实上可能慢一点儿，需要更多的机器指令运转来执行这个函数。当然，JS引擎也可能识别出了这种行为上的对称性，而仅仅为你处理 `Number(..)` 行为的内联形式（也就是 `+x` ）！

但是要记住，痴迷于 `+x` 和 `x | 0` 的比较在大多数情况下都是浪费精力。这是一个微观性能问题，而且你不应该让它使你的程序的可读性降低。

虽然你的程序的关键路径性能非常重要，但它不是唯一的因素。在几种性能上大体相似的选择中，可读性应当是另一个重要的考量。

尾部调用优化 (TCO)

尾部调用优化 （TCO）

正如我们早前简单提到的，ES6包含了一个冒险进入性能世界的具体需求。它是关于在函数调用时可能会发生的一种具体的优化形式：尾部调用优化（*TCO*）。

简单地说，一个“尾部调用”是一个出现在另一个函数“尾部”的函数调用，于是在这个调用完成后，就没有其他的事情要做了（除了也许要返回结果值）。

例如，这是一个带有尾部调用的非递归形式：

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

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

function baz() {
    return 1 + bar( 40 );    // 不是尾部调用
}

baz();                        // 42
```

`foo(y+1)` 是一个在 `bar(..)` 中的尾部调用，因为在 `foo(..)` 完成之后， `bar(..)` 也即而完成，除了在这里需要返回 `foo(..)` 调用的结果。然而， `bar(40)` 不是 一个尾部调用，因为在它完成后，在 `baz()` 能返回它的结果前，这个结果必须被加1。

不过于深入本质细节而简单地说，调用一个新函数需要保留额外的内存来管理调用栈，它称为一个“栈帧（stack frame）”。所以前面的代码段通常需要同时为 `baz()` ， `bar(..)` ，和 `foo(..)` 都准备一个栈帧。

然而，如果一个支持TCO的引擎可以认识到 `foo(y+1)` 调用位于 尾部位置 意味着 `bar(..)` 基本上完成了，那么当调用 `foo(..)` 时，它就并没有必要创建一个新的栈帧，而是可以重复利用既存的 `bar(..)` 的栈帧。这不仅更快，而且也更节省内存。

在一个简单的代码段中，这种优化机制没什么大不了的，但是当对付递归，特别是当递归会造成成百上千的栈帧时，它就变成了 相当有用的技术。引擎可以使用TCO在一个栈帧内完成所有调用！

在JS中递归是一个令人不安的话题，因为没有TCO，引擎就不得不实现一个随意的（而且各不相同

的）限制，规定它们允许递归栈能有多深，来防止内存耗尽。使用TCO，带有 尾部位置 调用的递归函数实质上可以没有边界地运行，因为从没有额外的内存使用！

考虑前面的递归 `factorial(..)`，但是将它重写为对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，而且它还是可以进行TCO优化的，因为两个内部的 `fact(..)` 调用都在 尾部位置。

注意： 一个需要注意的重点是，TCO尽在尾部调用实际存在时才会实施。如果你没用尾部调用编写递归函数，性能机制将仍然退回到普通的栈帧分配，而且引擎对于这样的递归的调用栈限制依然有效。许多递归函数可以像我们刚刚展示的 `factorial(..)` 那样重写，但是要小心处理细节。

ES6要求各个引擎实现TCO而不是留给它们自行考虑的原因之一是，由于对调用栈限制的恐惧，缺少 *TCO* 实际上趋向于减少特定的算法在JS中使用递归实现的机会。

如果无论什么情况下引擎缺少TCO只是安静地退化到性能差一些的方式上，那么它可能不会是ES6需要要求 的东西。但是因为缺乏TCO可能会实际上使特定的程序不现实，所以与其说它只是一种隐藏的实现细节，不如说它是一个重要的语言特性更合适。

ES6保证，从现在开始，JS开发者们能够在所有兼容ES6+的浏览器上信赖这种优化机制。这是JS性能的一个胜利！

复习

复习

有效地对一段代码进行性能基准分析，特别是将它与同样代码的另一种写法相比较来看哪一种方式更快，需要小心地关注细节。

与其运行你自己的统计学上合法的基准分析逻辑，不如使用Benchmark.js库，它会为你搞定。但要小心你如何编写测试，因为太容易构建一个看起来合法但实际上有漏洞的测试了—即使是一个微小的区别也会使结果歪曲到完全不可靠。

尽可能多地从不同的环境中得到尽可能多的测试结果来消除硬件/设备偏差很重要。jsPerf.com是一个用于大众外包性能基准分析测试的神奇网站。

许多常见的性能测试不幸地痴迷于无关紧要的微观性能细节，比如比较 `x++` 和 `++x` 。编写好的测试意味着理解如何聚焦大局上关注的问题，比如在关键路径上优化，和避免落入不同JS引擎的实现细节的陷阱。

尾部调用优化（TCO）是一个ES6要求的优化机制，它会使一些以前在JS中不可能的递归模式变得可能。TCO允许一个位于另一个函数的 尾部位置 的函数调用不需要额外的资源就可以执行，这意味着引擎不再需要对递归算法的调用栈深度设置一个随意的限制了。

附录A：库：asynquence

你不懂JS: 异步与性能

附录A: *asynquence* 库

第一章和第二章相当详细地探讨了常见的异步编程模式，以及如何通过回调解决它们。但我们也看到了为什么回调在处理能力上有着致命的缺陷，这将我们带到了第三章和第四章，Promise 与 Generator 为你的异步流程构建提供了一个更加坚实，可信，以及可推理的基础。

我在这本书中好几次提到我自己的异步库 *asynquence* (http://github.com/getify/asynquence) —— “async” + “sequence” = “asynquence”，现在我想简要讲解一下它的工作原理，以及它的独特设计为什么很重要和很有用。

在下一篇附录中，我们将要探索一些高级的异步模式，但为了它们的可用性能够使人接受你可能需要一个库。我们将使用 *asynquence* 来表达这些模式，所以你会想首先在这里花一点时间来了解这个库。

asynquence 绝对不是优秀异步编码的唯一选择；在这方面当然有许多了不起的库。但是 *asynquence* 提供了一种独特的视角 —— 通过将这些模式中最好的部分组合进一个单独的库，另外它基于一个基本的抽象：（异步）序列。

我的前提是，精巧的JS程序经常或多或少地需要将各种不同的异步模式交织在一起，而且这通常是完全依靠每个开发者自己去搞清楚的。与其引入关注于异步流程的不同方面的两个或更多的库，*asynquence* 将它们统一为各种序列步骤，成为单独一个需要学习和部署的核心库。

我相信 *asynquence* 有足够高的价值可以使 Promise 风格的异步流程控制编程变得超级容易完成，这就是我们为什么会在这里单单关注这个库。

开始之前，我将讲解 *asynquence* 背后的设计原则，然后我们将使用代码示例来展示它的API如何工作。

序列，抽象设计

对 *asynquence* 的理解开始于对一个基础抽象的理解：对于一个任务的任何一系列步骤来说，无论它们是同步的还是异步的，都可以被综合地考虑为一个“序列（sequence）”。换句话说，一个序列是一个容器，它代表一个任务，并由一个个完成这个任务的独立的（可能是异步的）步骤组成。

在这个序列中的每一个步骤都处于一个 Promise（见第三章） 的控制之下。也就是你向一个序列添加的每一个步骤都隐含地创建了一个 Promise，它被链接到这个序列的末尾。由于 Promise 的语义，在一个序列中的每一个步骤的推进都是异步的，即使你同步地完成这个步骤。

另外，一个序列将总是一步一步线性地进行，也就是步骤2总是发生在步骤1完成之后，如此类推。

当然，一个新的序列可以从既存的序列中分支出来，也就是分支仅在主序列在流程中到达那一点时发生。序列还可以用各种方式组合，包括使一个序列在流程中的一个特定的位置汇合另一个序列。

一个序列与 Promise 链有些相像。但是，在 Promise 链中，不存在一个可以引用整个链条的“把手”可以抓住。不管你持有哪一个 Promise 的引用，它都表示链条中当前的步骤外加挂载在它后面的其他步骤。实质上，你无法持有一个 Promise 链条的引用，除非你持有链条中第一个 Promise 的引用。

许多情况表明，持有一个综合地指向整个序列的引用是十分有用的。这些情况中最重要的一种就是序列的退出/取消。正如我们在第三章中展开谈过的那样，Promise 本身绝不应当是可以取消的，因为这违反了一个基本设计规则：外部不可变性。

但是序列没有这样的不可变性设计原则，这主要是由于序列不会作为需要不可变语义的未来值的容器被传递。所以序列是一个处理退出/取消行为的恰当的抽象层面。*asynquence* 序列可以在任何时候 `abort()` ，而且这个序列将会停止在那一点而不会因为任何原因继续下去。

为了流程控制，还有许多理由首选序列的抽象而非 Promise 链。

首先，Promise 链是一个更加手动的处理 —— 一旦你开始在你的程序中大面积地创建和链接 Promise ，这种处理可能会变得相当烦冗 —— 在那些使用 Promise 相当恰当的地方，这种烦冗会降低效率而使得开发者不愿使用Promise。

抽象意味着减少模板代码和烦冗，所以序列抽象是这个问题的一个好的解决方案。使用 Promise，你关注的是个别的步骤，而且不太会假定你将延续这个链条。而序列采用相反的方式，它假定序列将会无限地持续添加更多步骤。

当你开始考虑更高阶的 Promise 模式时（除了 `race([..])` 和 `all([..])` 以外），这种抽象复杂性的降低特别强大。

例如，在一个序列的中间，你可能想表达一个在概念上类似于 `try..catch` 的步骤，它的结果将总是成功，不管是意料之中的主线上的成功解析，还是为被捕获的错误提供一个正面的非错误信号。或者，你可能想表达一个类似于 retry/until 循环的步骤，它不停地尝试相同的步骤直到成功为止。

仅仅使用基本的 Promise，这类抽象不是很容易表达，而且在一个既存的 Promise 链的中间这样做不好看。但如果你将你的想法抽象为一个序列，并将一个步骤考虑为一个 Promise 的包装，这个包装可以隐藏这样的细节，它就可以使你以最合理的方式考虑流程控制，而不必关心细节。

第二，也许是更重要的，将异步流程控制考虑为一个序列中的步骤，允许你将这样的细节抽象出去 —— 每一个步骤中引入了哪一种异步性。在这种抽象之下，一个 Promise 将总是控制着步骤，但在抽象之上，这个步骤可以看起来像一个延续回调（简单的默认值），或者一个真正的 Promise，或者一个运行至完成的 Generator，或者… 希望你明白我的意思。

第三，序列可以通容易地被调整来适应于不同的思考模式，比如基于事件的，基于流的，或者基于相应式的编码。*asynquence* 提供了一种我称为“响应式序列”的模式（我们稍后讲解），它是 RxJS（“Reactive Extensions”） 中“响应式可监听”思想的变种，允许重复的事件每次触发一个新的序列实例。Promise 是一次性的，所以单独使用 Promise 来表达重复的异步性十分尴尬。

在一种我称为“可迭代序列”的模式中，另一种思考模式反转了解析/控制能力。与每一个步骤在内部控制它自己的完成（并因此推进这个序列）不同，序列被反转为通过一个外部迭代器来进行推进控制，而且在这个 可迭代序列 中的每一步仅仅应答 `next(..)` 迭代器 控制。

在本附录的剩余部分，我们将探索所有这些不同的种类，所以如果我们刚才的步伐太快也不要担心。

要点是，对于复杂的异步处理来说，序列是一个要比单纯的 Promise（Promise链）或单纯的 Generator 更加强大与合理的抽象，而 *asynquence* 被设计为使用恰当层面的语法糖来表达这种抽象，使得异步编程变得更加易于理解和更加令人愉快。

asynquence API

首先，你创建一个序列（一个 *asynquence* 实例）的方法是使用 `ASQ(..)` 函数。一个不带参数的 `ASQ()` 调用会创建一个空的初始序列，而向 `ASQ(..)` 传递一个或多个值或函数的话，它会使用每个参数值代表序列的初始步骤来创建序列。

注意： 为了这里所有的代码示例，我将使用 *asynquence* 在浏览器全局作用域中的顶层标识符： `ASQ` 。如果你通过一个模块系统（在浏览器或服务器中）引入并使用 *asynquence*，你当然可

以定义自己喜欢的符号，*asynquence* 不会关心这些！

许多在这里讨论的API方法都内建于 *asynquence* 的核心部分，而其他的API是通过引入可选的“contrib”插件包提供的。要知道一个方法是内建的还是通过插件定义的，可以参见 *asynquence* 的文档：http://github.com/getify/asynquence

步骤

如果一个函数代表序列中的一个普通步骤，那么这个函数会被这样调用：第一个参数是延续回调，而任何后续参数都是从前一个步骤中传递下来的消息。在延续回调被调用之前，这个步骤将不会完成。一旦延续回调被调用，你传递给它的任何参数值都会作为序列下一个步骤中的消息被发送。

要向一个序列添加额外的普通步骤，调用 `then(..)` （它实质上与 `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
} );
```

注意： 虽然 `then(..)` 这个名称与原生的 Promise API 完全一样，但是这个 `then(..)` 的含义是不同的。你可以传递任意多或者任意少的函数或值给 `then(..)` ，而它们中的每一个都被看作是一个分离的步骤。这里与完成/拒绝语义的双回调毫不相干。

在 Promise 中，可以把一个 Promise 与下一个你在 `then(..)` 的完成处理器中创建

并 `return` 的 Promise 链接。与此不同的是，在 *asynquence* 中，你所需要做的一切就是调用延续回调 —— 我总是称之为 `done()` ，但你可以起任何适合你的名字 —— 并将完成的消息作为参数值选择性地传递给它。

通过 `then(..)` 定义的每一个步骤都被认为是异步的。如果你有一个同步的步骤，你可以立即调用 `done(..)` ，或者使用更简单的 `val(..)` 步骤帮助函数：

```
// 步骤 1（同步）
ASQ( function(done){
    done( "Hello" );    // 手动同步
} )
// 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 );
} );
```

如你所见， `val(..)` 调用的步骤不会收到一个延续回调，因为这部分已经为你做好了 —— 而且参数列表作为一个结果显得不那么凌乱了！要向下一个步骤发送消息，你简单地使用 `return` 。

将 `val(..)` 考虑为表示一个同步的“仅含有值”的步骤，它对同步的值操作，比如 `logging` 之类，非常有用。

错误

与 Promise 相比 *asynquence* 的一个重要的不同之处是错误处理。

在 Promise 链条中，每个 Promise（步骤）都可以拥有自己独立的错误，而每个后续的步骤都有能力处理或不处理这个错误。这种语义（再一次）主要来自于对每个单独的 Promise 的关注，而非对整个链条（序列）的关注。

我相信，在大多数情况下，一个位于序列中某一部分的错误通常是不可恢复的，所以序列中后续的步骤毫无意义而应当被跳过。所以，默认情况下，在一个序列的任意一个步骤中的错误会将整个序列置于错误模式，而剩下的普通步骤将会被忽略。

如果你 确实 需要一个错误可以被恢复的步骤，有几个不同的API可以适应这种情况，比如 `try(..)`

—— 先前提到过的，有些像 `try..catch` 的步骤 —— 或者 `until(..)` —— 一个重试循环，它持续地尝试一个步骤直到它成功或你手动地 `break()` 这个循环。*asynquence* 甚至拥有 `pThen(..)` 和 `pCatch(..)` 方法，它们的工作方式与普通的 Promise 的 `then(..)` 和 `catch(..)` （见第三章）完全相同，所以如果你选择这么做，你就可以进行本地化的序列中错误处理。

重点是，你同时拥有两个选项，但是在我的经验中更常见的是默认情况。使用 Promise，要使一个步骤的链条在错误发生时一次性忽略所有步骤，你不得不小心不要在任何步骤中注册拒绝处理器；否则，这个错误会被视为处理过而被吞掉，而序列可能仍会继续下去（也许不是意料之中的）。要恰当且可靠地处理这种期待的行为有点儿尴尬。

要注册一个序列错误通知处理器，*asynquence* 提供了一个 `or(..)` 序列方法，它还有一个别名叫做 `onerror(..)` 。你可以在序列的任何位置调用这个方法，而且你可以注册任意多的处理器。这使得让多个不同的消费者监听一个序列是否失败变得很容易；从这个角度讲，它有点儿像一个错误事件处理器。

正如使用 Promise 那样，所有JS异常都会变为序列错误，或者你可以通过编程来发生一个序列错误：

```
var sq = ASQ( function(done){
    setTimeout( function(){
        // 为序列发出一个错误
        done.fail( "Oops" );
    }, 100 );
} )
.then( function(done){
    // 永远不会到达这里
} )
.or( function(err){
    console.log( err );           // Oops
} )
.then( function(done){
    // 也不会到达这里
} );

// 稍后

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

asynquence 与原生的 Promise 相比，在错误处理上另一个重要的不同就是“未处理异常”的默认行为。正如我们在第三章中以相当的篇幅讨论过的，一个没有被注册拒绝处理器的 Promise 如果被拒绝的话，将会无声地保持（也就是吞掉）那个错误；你不得不总是想着要用一个最后的 `catch(..)` 来终结一个链条。

在 *asynquence* 中，这种假设被颠倒过来了。

如果一个错误在序列上发生，而且 在那个时刻 它没有被注册错误处理器，那么这个错误会被报告至 `console` 。换言之，未处理的的拒绝将总是默认地被报告，因此不会被吞掉或丢掉。

为了防止重复的噪音，只要你向一个序列注册一个错误处理器，它就会使这个序列从这样的报告中退出。

事实上有许多情况你想要创建这样一个序列，它可能会在你有机会注册处理器之前就进入错误状态。这不常见，但可能时不时地发生。

在这样的情况下，你也可以通过在序列上调用 `defer()` 来使一个序列实例 从错误报告中退出。你应当仅在自己确信不会最终处理这样的错误时，才决定从报告中退出：

```
var sq1 = ASQ( function(done){
    doesnt.Exist();             // 将会向控制台抛出异常
} );

var sq2 = ASQ( function(done){
    doesnt.Exist();             // 仅仅会抛出一个序列错误
} )
// 错误报告中的退出
.defer();

setTimeout( function(){
    sq1.or( function(err){
        console.log( err );     // ReferenceError
    } );

    sq2.or( function(err){
        console.log( err );     // ReferenceError
    } );
}, 100 );

// ReferenceError （来自sq1）
```

这是一种比 Promise 本身拥有的更好的错误处理行为，因为它是一个成功的深渊，而不是一个失败的深渊（参见第三章）。

注意： 如果一个序列被导入（也就是被汇合入）另一个序列 —— 完整的描述参见“组合序列” —— 之后源序列从错误报告中退出，那么就必须考虑目标序列是否进行错误报告。

并行步骤

在你的序列中不是所有的步骤都将只拥有一个（异步）任务去执行；有些将会需要“并行”（并发地）执行多个步骤。在一个序列中，一个并发地处理多个子步骤的步骤称为一个 `gate(..)` —— 如果你喜

欢的话它还有一个别名 `all(..)` —— 而且它与原生的 `Promise.all([..])` 是对称的。

如果在 `gate(..)` 中的所有步骤都成功地完成了，那么所有成功的消息都将被传递到下一个序列步骤中。如果它们中的任何一个产生了一个错误，那么整个序列会立即进入错误状态。

考虑如下代码：

```
ASQ( function(done){
    setTimeout( done, 100 );
} )
.gate(
    function(done){
        setTimeout( function(){
            done( "Hello" );
        }, 100 );
    },
    function(done){
        setTimeout( function(){
            done( "World", "!" );
        }, 100 );
    }
)
.val( function(msg1,msg2){
    console.log( msg1 );    // Hello
    console.log( msg2 );    // [ "World", "!" ]
} );
```

为了展示差异，让我们把这个例子与原生 Promise 比较一下：

```
new Promise( function(resolve,reject){
    setTimeout( resolve, 100 );
} )
.then( function(){
    return Promise.all( [
        new Promise( function(resolve,reject){
            setTimeout( function(){
                resolve( "Hello" );
            }, 100 );
        } ),
        new Promise( function(resolve,reject){
            setTimeout( function(){
                // 注意：这里我们需要一个 [ ]
                resolve( [ "World", "!" ] );
            }, 100 );
        } )
    ] );
} )
```

```
.then( function(msgs){
    console.log( msgs[0] );    // Hello
    console.log( msgs[1] );    // [ "World", "!" ]
} );
```

讨厌。Promise 需要多得多的模板代码来表达相同的异步流程控制。这个例子很好地说明了为什么 *asynquence* API 和抽象使得对付 Promise 步骤容易多了。你的异步流程越复杂，它的改进程度就越高。

各种步骤

关于 *asynquence* 的 `gate(..)` 步骤类型，有好几种不同的 contrib 插件可能十分有用：

- `any(..)` 很像 `gate(..)`，除了为了继续主序列，只需要有一个环节最终必须成功。
- `first(..)` 很像 `any(..)`，除了只要有任何一个环节成功，主序列就会继续（忽略任何其余环节产生的后续结果）。
- `race(..)`（与 `Promise.race([..])` 对称）很像 `first(..)`，除了主序列会在任何环节完成时（不管成功还是失败）立即继续。
- `last(..)` 很像 `any(..)`，除了只有最后一个环节成功完成时才会把它的消息发送给主序列。
- `none(..)` 是 `gate(..)` 的反义：主序列仅在所有环节失败时才会继续（将所有环节的错误消息作为成功消息传送，或者反之）。

让我们首先定义一些帮助函数来使示例清晰一些：

```
function success1(done) {
    setTimeout( function(){
        done( 1 );
    }, 100 );
}

function success2(done) {
    setTimeout( function(){
        done( 2 );
    }, 100 );
}

function failure3(done) {
    setTimeout( function(){
        done.fail( 3 );
    }, 100 );
}

function output(msg) {
    console.log( msg );
}
```

现在，让我们展示一些这些 `gate(..)` 步骤的变种：

```
ASQ().race(
    failure3,
    success1
)
.or( output );        // 3


ASQ().any(
    success1,
    failure3,
    success2
)
.val( function(){
    var args = [].slice.call( arguments );
    console.log(
        args        // [ 1, undefined, 2 ]
    );
} );


ASQ().first(
    failure3,
    success1,
    success2
)
.val( output );        // 1


ASQ().last(
    failure3,
    success1,
    success2
)
.val( output );        // 2

ASQ().none(
    failure3
)
.val( output )        // 3
.none(
    failure3
    success1
)
.or( output );        // 1
```

另一个步骤种类是 `map(..)` ，它让你将一个数组的元素异步地映射为不同的值，而且在所有映射完成之前步骤不会前进。 `map(..)` 与 `gate(..)` 十分相似，除了它从一个数组，而非从一个指定的分离函数那里得到初始值，而且你定义一个函数回调来操作每一个值：

```
function double(x,done) {
    setTimeout( function(){
        done( x * 2 );
    }, 100 );
}

ASQ().map( [1,2,3], double )
.val( output );                    // [2,4,6]
```

另外， `map(..)` 可以从前一步骤传递来的消息中收到它的两个参数（数组或者回调）：

```
function plusOne(x,done) {
    setTimeout( function(){
        done( x + 1 );
    }, 100 );
}

ASQ( [1,2,3] )
.map( double )             // 收到消息`[1,2,3]`
.map( plusOne )             // 收到消息`[2,4,6]`
.val( output );             // [3,5,7]
```

另一个种类是 `waterfall(..)` ，它有些像混合了 `gate(..)` 的消息收集行为与 `then(..)` 的序列化处理。

步骤1首先被执行，然后来自步骤1的成功消息被传递给步骤2，然后两个成功消息走到步骤3，然后所有三个成功消息走到步骤4，如此继续，这样消息被某种程度上收集并从“瀑布”上倾泻而下。

考虑如下代码：

```
function double(done) {
    var args = [].slice.call( arguments, 1 );
    console.log( args );

    setTimeout( function(){
        done( args[args.length - 1] * 2 );
    }, 100 );
}

ASQ( 3 )
.waterfall(
    double,                    // [ 3 ]
```

```
    double,                    // [ 6 ]
    double,                    // [ 6, 12 ]
    double                    // [ 6, 12, 24 ]
)
.val( function(){
    var args = [].slice.call( arguments );
    console.log( args );    // [ 6, 12, 24, 48 ]
} );
```

如果在“瀑布”的任何一点发生错误，那么整个序列就会立即进入错误状态。

容错

有时你想在步骤一级管理错误，而不一定让它们使整个序列成为错误状态。*asynquence* 为此提供了两种步骤类型。

`try(..)` 尝试一个步骤，如果它成功，序列就会正常继续，但如果这个步骤失败了，失败的状态会转换成格式为 `{ catch: .. }` 的成功消息，它的值由错误消息填充：

```
ASQ()
.try( success1 )
.val( output )            // 1
.try( failure3 )
.val( output )            // { catch: 3 }
.or( function(err){
    // 永远不会到达这里
} );
```

你还可以使用 `until(..)` 构建一个重试循环，它尝试一个步骤，如果失败，就会在下一个事件轮询的 `tick` 中重试这个步骤，如此继续。

这种重试循环可以无限延续下去，但如果你想要从循环中跳出来，你可以在完成触发器上调用 `break()` 标志方法，它将主序列置为错误状态：

```
var count = 0;

ASQ( 3 )
.until( double )
.val( output )                    // 6
.until( function(done){
    count++;

    setTimeout( function(){
        if (count < 5) {
            done.fail();
        }
```

```
        else {
            // 跳出 `until(..)` 重试循环
            done.break( "Oops" );
        }
    }, 100 );
} )
.or( output );                  // Oops
```

Promise 式的步骤

如果你喜欢在你的序列中内联 Promise 风格的语义，比如 Promise 的 `then(..)` 和 `catch(..)`（见第三章），你可以使用 `pThen` 和 `pCatch` 插件：

```
ASQ( 21 )
.pThen( function(msg){
    return msg * 2;
} )
.pThen( output )                // 42
.pThen( function(){
    // 抛出一个异常
    doesnt.Exist();
} )
.pCatch( function(err){
    // 捕获这个异常（拒绝）
    console.log( err );           // ReferenceError
} )
.val( function(){
    // 主旋律回归到正常状态，
    // 因为前一个异常已经被
    // `pCatch(..)`捕获了
} );
```

`pThen(..)` 和 `pCatch(..)` 被设计为运行在序列中，但好像在普通的 Promise 链中动作。这样，你就可以在传递给 `pThen(..)` 的“完成”处理器中解析纯粹的 Promise 或者 *asynquence* 序列。

序列分支

一个有关 Promise 的可能十分有用的特性是，你可以在同一个 Promise 上添附多个 `then(..)` 处理器，这实质上在这个 Promise 的流程上创建了“分支”：

```
var p = Promise.resolve( 21 );

// （从`p`开始的）分支 1
p.then( function(msg){
    return msg * 2;
```

```
} )
.then( function(msg){
    console.log( msg );        // 42
} )

// (从`p`开始的)分支 2
p.then( function(msg){
    console.log( msg );        // 21
} );
```

使用 *asynquence* 的 `fork()` 可以很容易地进行同样的“分支”：

```
var sq = ASQ(..).then(..).then(..);

var sq2 = sq.fork();

// 分支 1
sq.then(..)..;

// 分支 2
sq2.then(..)..;
```

组合序列

与 `fork()` 相反的是，你可以通过将一个序列汇合进另一个来组合两个序列，使用 `seq(..)` 实例方法：

```
var sq = ASQ( function(done){
    setTimeout( function(){
        done( "Hello World" );
    }, 200 );
} );

ASQ( function(done){
    setTimeout( done, 100 );
} )
// 将序列 `sq` 汇合进这个系列
.seq( sq )
.val( function(msg){
    console.log( msg );        // Hello World
} )
```

`seq(..)` 可以像这里展示的那样接收一个序列本身，或者接收一个函数。如果是一个函数，那么它会期待这个函数被调用时返回一个序列，所以前面的代码可以这样写：

```
// ..
.seq( function(){
    return sq;
} )
// ..
```

另外，这个步骤还可以使用 `pipe(..)` 来完成：

```
// ..
.then( function(done){
    // 将 `sq` 导入延续回调 `done`
    sq.pipe( done );
} )
// ..
```

当一个序列被汇合时，它的成功消息流和错误消息流都会被导入。

注意： 正如早先的注意事项中提到过的，导入会使源序列从错误报告中退出，但不会影响目标序列的错误报告状态。

值与错误序列

如果一个序列的任意一个步骤只是一个普通值，那么这个值就会被映射到这个步骤的完成消息中：

```
var sq = ASQ( 42 );

sq.val( function(msg){
    console.log( msg );        // 42
} );
```

如果你想制造一个自动出错的序列：

```
var sq = ASQ.failed( "Oops" );

ASQ()
.seq( sq )
.val( function(msg){
    // 不会到达这里
} )
.or( function(err){
    console.log( err );        // Oops
} );
```

你还可能想要自动地创建一个延迟的值或者延迟的错误序列。使用 `after` 和 `failAfter` contrib 插件，这很容易：

```
var sq1 = ASQ.after( 100, "Hello", "World" );
var sq2 = ASQ.failAfter( 100, "Oops" );

sq1.val( function(msg1,msg2){
    console.log( msg1, msg2 );        // Hello World
} );

sq2.or( function(err){
    console.log( err );               // Oops
} );
```

你还可以使用 `after'(..)` 在一个序列的中间插入一个延迟：

```
ASQ( 42 )
// 在这个序列中插入一个延迟
.after( 100 )
.val( function(msg){
    console.log( msg );        // 42
} );
```

Promises 与回调

我认为 *asynquence* 序列在原生的 Promise 之上提供了许多价值，而且你会发现在很大程度上它在抽象层面上使用起来更舒适更强大。然而，将 *asynquence* 与其他非 *asynquence* 代码进行整合将是不可避免的现实。

使用 `promise(..)` 实例方法，你可以很容易地将一个 Promise（也就是 thenable —— 见第三章）汇合进一个序列：

```
var p = Promise.resolve( 42 );

ASQ()
.promise( p )            // 本可以写做：`function(){ return p; }`
.val( function(msg){
    console.log( msg );    // 42
} );
```

要向相反的方向走，从一个序列的特定步骤中分支/出让一个 Promise，使用 `toPromise` contrib 插件：

```
var sq = ASQ.after( 100, "Hello World" );

sq.toPromise()
// 现在这是一个标准的 promise 链了
.then( function(msg){
    return msg.toUpperCase();
} )
.then( function(msg){
    console.log( msg );        // HELLO WORLD
} );
```

有好几种帮助设施可以在使用回调的系统中适配 *asynquence*。要从你的序列中自动地生成一个“错误优先风格”回调，来接入一个面向回调的工具，使用 `errfcb`：

```
var sq = ASQ( function(done){
    // 注意：这里期待“错误优先风格”的回调
    someAsyncFuncWithCB( 1, 2, done.errfcb )
} )
.val( function(msg){
    // ..
} )
.or( function(err){
    // ..
} );

// 注意：这里期待“错误优先风格”的回调
anotherAsyncFuncWithCB( 1, 2, sq.errfcb() );
```

你还可能想要创建一个工具的序列包装版本 —— 与第三章的“promisory”和第四章的“thunkory”相比较 —— *asynquence* 为此提供了 `ASQ.wrap(..)`：

```
var coolUtility = ASQ.wrap( someAsyncFuncWithCB );

coolUtility( 1, 2 )
.val( function(msg){
    // ..
} )
.or( function(err){
    // ..
} );
```

注意： 为了清晰（和有趣！），让我们为来自 `ASQ.wrap(..)` 的产生序列的函数杜撰另一个名词，就像这里的 `coolUtility`。我提议“sequory”（“sequence” + “factory”）。

可迭代序列

一个序列普通的范例是，每一个步骤都负责完成它自己，进而推进这个序列。Promise 就是这样工作的。

不幸的是，有时你需要从外部控制一个 Promise/步骤，而这会导致尴尬的“能力抽取”。

考虑这个 Promise 的例子：

```
var domready = new Promise( function(resolve,reject){
    // 不想把这个放在这里，因为在逻辑上
    // 它属于代码的另一部分
    document.addEventListener( "DOMContentLoaded", resolve );
} );

// ..

domready.then( function(){
    // DOM 准备好了！
} );
```

关于 Promise 的“能力抽取”范模式看起来像这样：

```
var ready;

var domready = new Promise( function(resolve,reject){
    // 抽取 `resolve()` 能力
    ready = resolve;
} );

// ..

domready.then( function(){
    // DOM 准备好了！
} );

// ..

document.addEventListener( "DOMContentLoaded", ready );
```

注意： 在我看来，这种反模式是一种尴尬的代码风格，但有些开发者喜欢，我不能理解其中的原因。

asynquence 提供一种我称为“可迭代序列”的反转序列类型，它将控制能力外部化（它在 `domready` 这样的情况下十分有用）：

```
// 注意：这里`domready`是一个控制序列的 *迭代器*
var domready = ASQ.iterable();

// ..

domready.val( function(){
    // DOM 准备好了！
} );

// ..

document.addEventListener( "DOMContentLoaded", domready.next );
```

与我们在这个场景中看到的东西比起来，可迭代序列还有很多内容。我们将在附录B中回过头来讨论它们。

运行 Generator

在第四章中，我们衍生了一种称为 `run(..)` 的工具，它可以将 generator 运行至完成，监听被 `yield` 的 Promise 并使用它们来异步推进 generator。*asynquence* 正好有一个这样的内建工具，称为 `runner(..)` 。

为了展示，让我们首先建立一些帮助函数：

```
function doublePr(x) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            resolve( x * 2 );
        }, 100 );
    } );
}

function doubleSeq(x) {
    return ASQ( function(done){
        setTimeout( function(){
            done( x * 2)
        }, 100 );
    } );
}
```

现在，我们可以在一个序列的中间使用 `runner(..)` 作为一个步骤：

```
ASQ( 10, 11 )
.runner( function*(token){
```

```
    var x = token.messages[0] + token.messages[1];

    // yield 一个真正的 promise
    x = yield doublePr( x );

    // yield 一个序列
    x = yield doubleSeq( x );

    return x;
} )
.val( function(msg){
    console.log( msg );             // 84
} );
```

包装过的 Generator

你还可以创建自包装的 generator —— 也就是一个普通函数，运行你指定的 generator 并为它的完成返回一个序列 —— 通过 `ASQ.wrap(..)` 包装它：

```
var foo = ASQ.wrap( function*(token){
    var x = token.messages[0] + token.messages[1];

    // yield 一个真正的 promise
    x = yield doublePr( x );

    // yield 一个序列
    x = yield doubleSeq( x );

    return x;
}, { gen: true } );

// ..

foo( 8, 9 )
.val( function(msg){
    console.log( msg );             // 68
} );
```

`runner(..)` 还能做很多很牛的事情，我们会在附录B中回过头来讨论它。

复习

asynquence 是一个在 Promise 之上的简单抽象 —— 一个序列是一系列（异步）步骤，它的目标是使各种异步模式更加容易使用，而在功能上没有任何妥协。

在 *asynquence* 的核心API与它的 `contrib` 插件中，除了我们在这篇附录中看到的内容以外还有其他的好东西，我们把对这些剩余功能的探索作为练习留给读者。

现在你看到了 *asynquence* 的实质与精神。关键点是，一个序列由许多步骤组成，而这些步骤可以使许多不同种类的 `Promise`，或者它们可以是一个 `generator` 运行器，或者… 选择由你来决定，你有完全的自由为你的任务采用恰当的任何异步流程控制逻辑。

如果你能理解这些 *asynquence* 代码段，那么你现在就可以相当快地学会这个库；它实际上没有那么难学！

如果你依然对它如何（或为什么！）工作感到模糊，那么在进入下一篇附录之前，你将会想要多花一点时间去查看前面的例子，并亲自把玩一下 *asynquence*。附录B将会在几种更高级更强大的异步模式中使用 *asynquence*。

附录B：高级异步模式

你不懂JS： 异步与性能

附录B： 高级异步模式

为了了解主要基于 Promise 与 Generator 的面向序列异步流程控制，附录A介绍了 *asynquence* 库。

现在我们将要探索其他建立在既存理解与功能之上的高级异步模式，并看看 *asynquence* 是如何在不需要许多分离的库的情况下，使得这些精巧的异步技术与我们的程序进行混合与匹配的。

可迭代序列

我们在前一篇附录中介绍过 *asynquence* 的可迭代序列，我们将更加详细地重温它们。

为了复习，回忆一下：

```
var domready = ASQ.iterable();

// ..

domready.val( function(){
    // DOM 准备好了
} );

```

```
// ..

document.addEventListener( "DOMContentLoaded", domready.next );
```

现在，让我们定义将一个多步骤序列定义为一个可迭代序列：

```
var steps = ASQ.iterable();

steps
.then( function STEP1(x){
    return x * 2;
} )
.then( function STEP2(x){
    return x + 3;
} )
.then( function STEP3(x){
    return x * 4;
} );

steps.next( 8 ).value;    // 16
steps.next( 16 ).value;    // 19
steps.next( 19 ).value;    // 76
steps.next().done;        // true
```

如你所见，一个可迭代序列是一个标准兼容的 *iterator*（见第四章）。所以，就像一个 generator（或其他任何 可迭代对象）那样，它是可以使用ES6 `for..of` 循环进行迭代的，

```
var steps = ASQ.iterable();

steps
.then( function STEP1(){ return 2; } )
.then( function STEP2(){ return 4; } )
.then( function STEP3(){ return 6; } )
.then( function STEP4(){ return 8; } )
.then( function STEP5(){ return 10; } );

for (var v of steps) {
    console.log( v );
}
// 2 4 6 8 10
```

除了在前一篇附录中展示的事件触发的例子之外，可迭代序列的有趣之处还因为它们实质上可以被视为 generator 和 Promise 链的替代品，但具备更多灵活性。

考虑一个多Ajax请求的例子 —— 我们已经在第三章和第四章中看到过同样的场景，分别使用一个

Promise 链和一个 generator —— 表达为一个可迭代序列：

```
// 兼容序列的 ajax
var request = ASQ.wrap( ajax );

ASQ( "http://some.url.1" )
.runner(
    ASQ.iterable()

    .then( function STEP1(token){
        var url = token.messages[0];
        return request( url );
    } )

    .then( function STEP2(resp){
        return ASQ().gate(
            request( "http://some.url.2/?v=" + resp ),
            request( "http://some.url.3/?v=" + resp )
        );
    } )

    .then( function STEP3(r1,r2){ return r1 + r2; } )
)
.val( function(msg){
    console.log( msg );
} );
```

可迭代序列表达了一系列顺序的（同步的或异步的）步骤，它看起来与一个 Promise 链极其相似 —— 换言之，它要比单纯嵌套的回调看起来干净的多，但没有 generator 的基于 `yield` 的顺序化语法那么好。

但我们将可迭代序列传入 `ASQ#runner(..)`，它将可迭代序列像一个 generator 那样运行至完成。由于几个原因，一个可迭代序列的行为实质上与一个 generator 相同的事实是值得注意的：

首先，对于ES6 generator 的特定子集来说，可迭代对象是它的一种前ES6等价物，这意味着你既可以直接编写它们（为了在任何地方都能运行），也可以编写ES6 generator 并将它们转译/转换成可迭代序列（或者 Promise 链！）。

将一个异步运行至完成的 generator 考虑为一个 Promise 链的语法糖，是对它们之间的同构关系的一种重要认识。

在我们继续之前，我们应当注意到，前一个代码段本可以用 *asynquence* 表达为：

```
ASQ( "http://some.url.1" )
.seq( /*STEP 1*/ request )
.seq( function STEP2(resp){
```

```
    return ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );
} )
.val( function STEP3(r1,r2){ return r1 + r2; } )
.val( function(msg){
    console.log( msg );
} );
```

进一步，步骤2本可以被表达为：

```
.gate(
    function STEP2a(done,resp) {
        request( "http://some.url.2/?v=" + resp )
        .pipe( done );
    },
    function STEP2b(done,resp) {
        request( "http://some.url.3/?v=" + resp )
        .pipe( done );
    }
)
```

那么，为什么我们要在一个简单/扁平的 *asyquence* 链看起来可以很好地工作的情况下，很麻烦地将自己的控制流在一个 `ASQ#runner(..)` 步骤中表达为一个可迭代序列呢？

因为可迭代序列的形式有一种重要的技巧可以给我们更多的力量。继续读。

扩展可迭代序列

Generator，普通的 *asynquence* 序列，和 Promise 链，都是被 急切求值 的 —— 控制流程最初要表达的的内容 就是 紧跟在后面的固定流程。

然而，可迭代序列是 懒惰求值 的，这意味着在可迭代序列执行期间，如果有需要的话你可以用更多的步骤扩展这个序列。

注意： 你只能在一个可迭代序列的末尾连接，而不是在序列的中间插入。

为了熟悉这种能力，首先让我们看一个比较简单（同步）的例子：

```
function double(x) {
    x *= 2;

    // 我们应当继续扩展吗？
    if (x < 500) {
        isq.then( double );
```

```
    }

    return x;
}

// 建立单步可迭代序列
var isq = ASQ.iterable().then( double );

for (var v = 10, ret;
    (ret = isq.next( v )) && !ret.done;
) {
    v = ret.value;
    console.log( v );
}
```

这个可迭代序列开始时只有一个定义好的步骤（`isq.then(double)`），但是这个序列会在特定条件下（`x < 500`）持续扩展自己。*asynquence* 序列和 Promise 链在技术上都 可以 做相似的事情，但是我们将看到它们的这种能力不足的一些原因。

这个例子意义不大，而且本可以使用一个 generator 中的 `while` 循环来表达，所以我们将考虑更精巧的情况。

例如，你可以检查一个Ajax请求的应答，看它是否指示需要更多的数据，你可以条件性地向可迭代序列插入更多的步骤来发起更多的请求。或者你可以条件性地在Ajax处理器的末尾加入一个格式化步骤。

考虑如下代码：

```
var steps = ASQ.iterable()

.then( function STEP1(token){
    var url = token.messages[0].url;

    // 有额外的格式化步骤被提供吗？
    if (token.messages[0].format) {
        steps.then( token.messages[0].format );
    }

    return request( url );
} )

.then( function STEP2(resp){
    // 要为序列增加另一个Ajax请求吗？
    if (/x1/.test( resp )) {
        steps.then( function STEP5(text){
            return request(
                "http://some.url.4/?v=" + text
```

```
            );
        } );
    }

    return ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );
} )

.then( function STEP3(r1,r2){ return r1 + r2; } );
```

你可以在两个地方看到我们使用 `steps.then(..)` 条件性地扩展了 `step` 。为了运行这个 `steps` 可迭代序列，我们只要使用 `ASQ#runner(..)` 将它与一个 *asynquence* 序列（这里称为 `main` ）链接进我们的主程序流程中：

```
var main = ASQ( {
    url: "http://some.url.1",
    format: function STEP4(text){
        return text.toUpperCase();
    }
} )
.runner( steps )
.val( function(msg){
    console.log( msg );
} );
```

`steps` 可迭代序列的灵活性可以使用一个 `generator` 来表达吗？某种意义上可以，但我们不得不以一种有些尴尬的方式重新安排逻辑：

```
function *steps(token) {
    // **步骤 1**
    var resp = yield request( token.messages[0].url );

    // **步骤 2**
    var rvals = yield ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );

    // **步骤 3**
    var text = rvals[0] + rvals[1];

    // **步骤 4**
    // 有额外的格式化步骤被提供吗？
    if (token.messages[0].format) {
```

```
        text = yield token.messages[0].format( text );
    }

    // **步骤 5**
    // 要为序列增加另一个Ajax请求吗？
    if (/foobar/.test( resp )) {
        text = yield request(
            "http://some.url.4/?v=" + text
        );
    }

    return text;
}

// 注意：`*steps()`可以向先前的`step`一样被相同的`ASQ`序列运行
```

先把我们已经知道的序列的好处，以及看起来同步的 generator 语法（见第四章）放在一边，`steps` 逻辑不得不在 `*steps()` generator 形式中重排，来假冒可扩展的可迭代序列 `steps` 的动态机制。

那么，使用 Promise 或者序列如何表达这种功能呢？你 可以 这么做：

```
var steps = something( .. )
.then( .. )
.then( function(..){
    // ..

    // 扩展这个链条，对吧？
    steps = steps.then( .. );

    // ..
})
.then( .. );
```

这里要抓住的问题很微妙但很重要。那么，考虑试着将我们的 `stpes` Promise 链连接到我们的主程序流程中 —— 这次使用 Promise 代替 *asynquence* 来表达：

```
var main = Promise.resolve( {
    url: "http://some.url.1",
    format: function STEP4(text){
        return text.toUpperCase();
    }
} )
.then( function(..){
    return steps;            // 提示！
} )
```

```
.val( function(msg){
    console.log( msg );
} );
```

现在你能发现问题吗？仔细观察！

对于序列步骤的顺序来说，这里有一个竞合状态。当你 `return steps` 时， `steps` 在那个时刻 可能是原本定义好的 promise 链了，或者它现在可能通过 `steps = steps.then(..)` 调用正指向扩张的 promise 链，这要看事情以什么顺序发生。

这里有两种可能的结果：

- 如果 `steps` 仍然是原来的 Promise 链，一旦它稍后通过 `steps = steps.then(..)` “扩展”，这个位于链条末尾的扩展过的 promise 是 不会 被 `main` 流程考虑的，因为它已经通过这个 `steps` 链了。这就是不幸的 急切求值 限制。
- 如果 `steps` 已经是扩展过的 promise 链了，那么由于这个扩展过的 promise 正是 `main` 要通过的东西，所以它会如我们期望的那样工作。

第一种情况除了展示竞合状态不可容忍的明显事实，它还展示了 promise 链的 急切求值。相比之下，我们可以很容易地扩展可迭代序列而没有这样的问题，因为可迭代序列是 懒惰求值 的。

你越需要自己的流程控制动态，可迭代序列就越显得强大。

提示： 在 *asynquence* 的网站 (https://github.com/getify/asynquence/blob/master/README.md#iterable-sequences)上可以看到更多关于可迭代序列的信息与示例。

事件响应式

（至少！）从第三章看来这应当很明显：Promise 是你异步工具箱中的一种非常强大的工具。但它们明显缺乏处理事件流的能力，因为一个 Promise 只能被解析一次。而且坦白地讲，对于 *asynquence* 序列来说这也正是它的一个弱点。

考虑这样一个场景：你想要在一个特定事件每次被触发时触发一系列步骤。一个单独的 Promise 或序列不能表示这个事件全部的发生状况。所以，你不得不为每一个事件的发生创建一个全新的 Promise 链（或序列），比如：

```
listener.on( "foobar", function(data){

    // 创建一个新的事件处理 Promise 链
    new Promise( function(resolve,reject){
        // ..
    } )
    .then( .. )
```

```
    .then( .. );

} );
```

在这种方式拥有我们需要的基本功能，但是对于表达我们意图中的逻辑来说远不能使人满意。两种分离的能力混杂在这个范例中：事件监听，与事件应答；而关注点分离原则恳求我们将这些能力分开。

细心的读者会发现，这个问题与我们在第二章中详细讲解过的问题是有些对称的；它是一种控制反转问题。

想象一下非反转这个范例，就像这样：

```
var observable = listener.on( "foobar" );

// 稍后
observable
.then( .. )
.then( .. );

// 在其他的地方
observable
.then( .. )
.then( .. );
```

值 `observable` 不是一个真正的 Promise，但你可以像监听一个 Promise 那样 监听 它，所以它们是有密切关联的。事实上，它可以被监听很多次，而且它会在每次事件（ `"foobar"` ）发生时都发送通知。

提示： 我刚刚展示过的这个模式，是响应式编程（reactive programming，也称为 RP）背后的概念和动机的 大幅度简化，响应式编程已经由好几种了不起的项目和语言实现/详细论述过了。RP的一个变种是函数响应式编程（functional reactive programming，FRP），它指的是在数据流之上实施函数式编程技术（不可变性，参照完整性，等等）。“响应式”指的是随着事件的推移散布这种功能，以对事件进行应答。对此感兴趣的读者应当考虑学习“响应式可监听对象”，它源于由微软开发的神奇的“响应式扩展”库(对于 JavaScript 来说是 “RxJS”，http://rxjs.codeplex.com/)；它可要比我刚刚展示过的东西精巧和强大太多了。另外，Andre Staltz 写过一篇出色的文章(https://gist.github.com/staltz/868e7e9bc2a7b8c1f754)，用具体的例子高效地讲解了RP。

ES7 可监听对象

在本书写作时，有一个早期ES7提案，一种称为“Observable（可监听对象）”的新数据类型(https://github.com/jhusain/asyncgenerator#introducing-observable)，它在精神上与我们在这里讲解过的相似，但是绝对更精巧。

这种可监听对象的概念是，你在一个流上“监听”事件的方法是传入一个 generator —— 其实 迭代器 才是有趣的部分 —— 它的 `next(..)` 方法会为每一个事件而调用。

你可以想象它是这样一种东西：

```
// `someEventStream` 是一个事件流，来自于鼠标点击之类

var observer = new Observer( someEventStream, function*(){
    while (var evt = yield) {
        console.log( evt );
    }
} );
```

你传入的 generator 将会 `yield` 而暂停 `while` 循环，来等待下一个事件。添附在 generator 实例上的 迭代器 的 `next(..)` 将会在每次 `someEventStream` 发布一个新事件时被调用，因此这个事件将会使用 `evt` 数据推进你的 generator/迭代器。

在这里的监听事件功能中，重要的是 迭代器 的部分，而不是 generator。所以从概念上讲，你实质上可以传入任何可迭代对象，包括 `ASQ.iterable()` 可迭代序列。

有趣的是，还存在一些被提案的适配方案，使得从特定类型的流中构建可监听对象变得容易，例如为DOM事件提案的 `fromEvent(..)` 。如果你去看看 `fromEvent(..)` 在早期ES7提案中推荐的实现方式，你会发现它与我们将要在下一节中看到的 `ASQ.react(..)` 极其相似。

当然，这些都是早期提案，所以最终脱颖而出的东西可能会在外观/行为上与这里展示的有很大的不同。但是看到在不同的库与语言提案在概念上的早期统一还是很激动人心的！

响应式序列

将这种可监听对象（和F/RP）的超级简要的概览作为我们的启发与动机，我们现在将展示一种“响应式可监听对象”的很小的子集的适配方案，我称之为“响应式序列”。

首先，让我们从如何创建一个可监听对象开始，使用一个称为 `react(..)` 的 *asynquence* 插件工具：

```
var observable = ASQ.react( function setup(next){
    listener.on( "foobar", next );
} );
```

现在，让我们看看如何为这个 `observable` 定义一个“响应的”序列 —— 在F/RP中，这通常称为“监听”：

```
observable
.seq( .. )
```

```
.then( .. )
.val( .. );
```

所以，你只需要通过在这个可监听对象后面进行链接就可以了。很容易，是吧？

在F/RP中，事件流经常会通过一组函数式的变形，比如 `scan(..)` ， `map(..)` ， `reduce(..)` ，等等。使用响应式序列，每个事件会通过一个序列的新的实例。让我们看一个更具体的例子：

```
ASQ.react( function setup(next){
    document.getElementById( "mybtn" )
    .addEventListener( "click", next, false );
} )
.seq( function(evt){
    var btnID = evt.target.id;
    return request(
        "http://some.url.1/?id=" + btnID
    );
} )
.val( function(text){
    console.log( text );
} );
```

响应式序列的“响应式”部分来源于分配一个或多个事件处理器来调用事件触发器（调用 `next(..)` ）。

响应式序列的“序列”部分正是我们已经探索过的：每一个步骤都可以是任何合理的异步技术 —— 延续回调，Promise 或者 generator。

一旦拟建立了一个响应式序列，只要事件被持续地触发，它就会一直初始化序列的实例。如果你想停止一个响应式序列，你可以调用 `stop()` 。

如果一个响应式序列被 `stop()` 了，你可能还想注销事件处理器；为此你可以注册一个拆卸处理器：

```
var sq = ASQ.react( function setup(next,registerTeardown){
    var btn = document.getElementById( "mybtn" );

    btn.addEventListener( "click", next, false );

    // 只要`sq.stop()`被调用，它就会被调用
    registerTeardown( function(){
        btn.removeEventListener( "click", next, false );
    } );
} )
.seq( .. )
.then( .. )
.val( .. );
```

```
14.
15. // 稍后
16. sq.stop();
```

注意： 在 `setup(..)` 处理器内部的 `this` 绑定引用是 `sq` 响应式序列，所以你可以在响应式序列的定义中使用 `this` 引用，比如调用 `stop()` 之类的方法，等等。

这是一个来自 Node.js 世界的例子，使用响应式序列处理到来的HTTP请求：

```
1. var server = http.createServer();
2. server.listen(8000);
3.
4. // 响应式监听
5. var request = ASQ.react( function setup(next,registerTeardown){
6.     server.addListener( "request", next );
7.     server.addListener( "close", this.stop );
8.
9.     registerTeardown( function(){
10.         server.removeListener( "request", next );
11.         server.removeListener( "close", request.stop );
12.     } );
13. });
14.
15. // 应答请求
16. request
17. .seq( pullFromDatabase )
18. .val( function(data,res){
19.     res.end( data );
20. } );
21.
22. // 关闭 node
23. process.on( "SIGINT", request.stop );
```

`next(..)` 触发器还可以很容易地适配 node 流，使用 `onStream(..)` 和 `unStream(..)`：

```
1. ASQ.react( function setup(next){
2.     var fstream = fs.createReadStream( "/some/file" );
3.
4.     // 将流的 "data" 事件导向 `next(..)`
5.     next.onStream( fstream );
6.
7.     // 监听流的结束
8.     fstream.on( "end", function(){
9.         next.unStream( fstream );
10.     } );
11. } )
```

```
.seq( .. )
.then( .. )
.val( .. );
```

你还可以使用序列组合来构成多个响应式序列流：

```
var sq1 = ASQ.react( .. ).seq( .. ).then( .. );
var sq2 = ASQ.react( .. ).seq( .. ).then( .. );

var sq3 = ASQ.react(..)
.gate(
    sq1,
    sq2
)
.then( .. );
```

这里的要点是，`ASQ.react(..)`是一个F/RP概念的轻量级适配，使得将一个事件流与一个序列的连接成为可能，因此得名“响应式序列”。对于基本的响应式用法，响应式序列的能力通常是足够的。

注意： 这里有一个使用`ASQ.react(..)`来管理UI状态的例子(http://jsbin.com/rozipaki/6/edit?js,output)，和另一个使用`ASQ.react(..)`来处理HTTP请求/应答流的例子(https://gist.github.com/getify/bba5ec0de9d6047b720e)。

Generator 协程

希望第四章帮助你很好地熟悉了ES6 generator。特别地，我们将重温并更加深入“Generator 并发性”的讨论。

我们想象了一个`runAll(..)`工具，它可以接收两个或更多的 generator 并且并发地运行它们，让它们协作地将控制权从一个`yield`到下一个，并带有可选的消息传递。

除了能够将一个 generator 运行至完成之外，我们在附录A中谈论过的`AQS#runner(..)`是一个`runAll(..)`概念的近似实现，它可以将多个 generator 并发地运行至完成。

那么让我们看看如何实现第四章的并发Ajax场景：

```
ASQ(
    "http://some.url.2"
)
.runner(
    function*(token){
        // 转移控制权
        yield token;

```

```
        var url1 = token.messages[0]; // "http://some.url.1"

        // 清空消息重新开始
        token.messages = [];

        var p1 = request( url1 );

        // 转移控制权
        yield token;

        token.messages.push( yield p1 );
    },
    function*(token){
        var url2 = token.messages[0]; // "http://some.url.2"

        // 传递消息并转移控制权
        token.messages[0] = "http://some.url.1";
        yield token;

        var p2 = request( url2 );

        // 移控制权
        yield token;

        token.messages.push( yield p2 );

        // 讲结果传递给下一个序列步骤
        return token.messages;
    }
)
.val( function(res){
    // `res[0]` comes from "http://some.url.1"
    // `res[1]` comes from "http://some.url.2"
} );
```

以下是 `ASQ#runner(..)` 和 `runAll(..)` 之间的主要不同：

- 每个 generator（协程）都被提供了一个称为`token`的参数值，它是一个当你想要明确地将控制权传递给下一个协程时`yield`用的特殊值。
- `token.messages`是一个数组，持有从前一个序列步骤中传入的任何消息。它也是一种数据结构，你可以用来在协程之间分享消息。
- `yield`一个 Promise（或序列）值不会传递控制权，但会暂停这个协程处理直到这个值准备好。
- 这个协程处理运行到最后`return`或`yield`的值将会传递给序列中的下一个步骤。

为了适应不同的用法，在 `ASQ#runner(..)` 功能的基础上包装一层帮助函数也很容易。

状态机

许多程序员可能很熟悉的一个例子是状态机。在一个简单包装工具的帮助下，你可以创一个易于表达的状态机处理器。

让我们想象一个这样的工具。我们称之为 `state(..)`，我们将传递给它两个参数值：一个状态值和一个处理这个状态的 generator。`state(..)` 将担负起创建并返回一个适配器 generator 的脏活，并把它传递给 `ASQ#runner(..)`。

考虑如下代码：

```
function state(val,handler) {
    // 为这个状态制造一个协程处理器
    return function*(token) {
        // 状态转换处理器
        function transition(to) {
            token.messages[0] = to;
        }

        // 设置初始状态（如果还没有设置的话）
        if (token.messages.length < 1) {
            token.messages[0] = val;
        }

        // 持续运行直到最终状态（false）
        while (token.messages[0] !== false) {
            // 当前的状态匹配这个处理器吗？
            if (token.messages[0] === val) {
                // 委托到状态处理器
                yield *handler( transition );
            }

            // 要把控制权转移给另一个状态处理器吗？
            if (token.messages[0] !== false) {
                yield token;
            }
        }
    };
}
```

如果你仔细观察，你会发现 `state(..)` 返回了一个接收 `token` 的 generator，然后它建立一个 `while` 循环，这个循环会运行到状态机直到到达它的最终状态（我们随意地将它选定为 `false` 值）为止；这正是我们想要传递给 `ASQ#runner(..)` 的那种 generator！

我们还随意地保留了 `token.messages[0]` 值槽，放置我们的状态机将要追踪的当前状态，这意味着我们甚至可以指定初始状态，作为序列中前一个步骤传递来的值。

我们如何将 `state(..)` 帮助函数与 `ASQ#runner(..)` 一起使用呢？

```
var prevState;

ASQ(
    /* 可选的：初始状态值 */
    2
)
// 运行我们的状态机
// 转换是：2 -> 3 -> 1 -> 3 -> false
.runner(
    // 状态 `1` 处理器
    state( 1, function *stateOne(transition){
        console.log( "in state 1" );

        prevState = 1;
        yield transition( 3 );    // 前往状态 `3`
    } ),

    // 状态 `2` 处理器
    state( 2, function *stateTwo(transition){
        console.log( "in state 2" );

        prevState = 2;
        yield transition( 3 );    // 前往状态 `3`
    } ),

    // 状态 `3` 处理器
    state( 3, function *stateThree(transition){
        console.log( "in state 3" );

        if (prevState === 2) {
            prevState = 3;
            yield transition( 1 ); // 前往状态 `1`
        }
        // 完成了！
        else {
            yield "That's all folks!";

            prevState = 3;
            yield transition( false ); // 终止状态
        }
    } )
)
// 状态机运行完成，所以继续
.val( function(msg){
    console.log( msg );    // That's all folks!
} );
```

重要的是，`*stateOne(..)`，`*stateTwo(..)`，和`*stateThree(..)` generator 本身会在每次进入那种状态时被调用，它们会在你`transition(..)`到另一个值时完成。虽然没有在这里展示，但是这些状态 generator 处理器理所当然地可以通过`yield` Promise/序列/thunk 来异步地暂停。

隐藏在底层的 generator 是由`state(..)`帮助函数产生的，实际上被传递给`ASQ#runner(..)`的 generator 是持续并发运行至状态机长度的那一个，它们的每一个都协作地将控制权`yield`给下一个，如此类推。

注意： 看看这个“乒乓”的例子(http://jsbin.com/qutabu/1/edit?js,output)，它展示了由`ASQ#runner(..)`驱动的 generator 的协作并发的用法。

通信序列化处理（CSP）

“通信序列化处理（Communicating Sequential Processes —— CSP）”是由 C. A. R. Hoare 在1978年的一篇学术论文(http://dl.acm.org/citation.cfm?doid=359576.359585)中首先被提出的，后来在1985年的一本同名书籍中被描述过。CSP描述了一种并发“进程”在处理期间进行互动（也就是“通信”）的形式方法。

你可能会回忆起我们在第一章检视过的并发“进程”，所以我们对CSP的探索将会建立在那种理解之上。

就像大多数计算机科学中的伟大概念一样，CSP深深地沉浸在学术形式主意中，被表达为一种代数处理。然而，我怀疑满是符号的代数定理不会给读者带来太多实际意义，所以我们将找其他的方法将CSP带进我们的大脑。

我会将很多CSP的形式描述和证明留给 Hoare 的文章，与其他许多美妙的相关作品。取而代之的是，我们将尽可能以一种非学院派的、但愿是可以直接理解的方法，来试着简要地讲解CSP的思想。

消息传递

CSP的核心原则是，在独立进程之间的通信/互动都必须通过正式的消息传递。也许与你的期望背道而驰，CSP的消息传递是作为同步行为进行描述的，发送进程与接收进程都不得不为消息的传递做好准备。

这样的同步消息怎么会与 JavaScript 中的异步编程有联系？

这种联系具体来自于 ES6 generator 的性质 —— generator 被用于生产看似同步的行为，而这些行为的内部既可以是同步的也可以（更可能）是异步的。

换言之，两个或更多并发运行的 generator 可能看起来像是在互相同步地传递消息，而同时保留了系统的异步性基础，因为每个 generator 的代码都会被暂停（也就是“阻塞”）来等待一个异步动作的运行。

这是如何工作的？

想象一个称为“A”的 generator，它想要给 generator “B” 发送一个消息。首先，“A” `yield` 出要发送给“B”的消息（因此暂停了“A”）。当“B”准备好并拿走这个消息时，“A”才会继续（解除阻塞）。

与此对称的，想象一个 generator “A”想要 从 “B”接收一个消息。“A” `yield` 出一个从“B”取得消息的请求（因此暂停了“A”），一旦“B”发送了一个消息，“A”就拿来这个消息并继续。

对于这种CSP消息传递理论来说，一个更广为人知的表达形式是 ClojureScript 的 core.async 库，以及 *go* 语言。它们将CSP中描述的通信语义实现为一种在进程之间打开的管道，称为“频道（channel）”。

注意： 频道 这个术语描述了问题的一部分，因为存在一种模式，会有多于一个的值被一次性发送到这个频道的“缓冲”中；这与你对流的认识相似。我们不会在这里深入这个问题，但是对于数据流的管理来说它可能是一个非常强大的技术。

在CSP最简单的概念中，一个我们在“A”和“B”之间建立的频道会有一个称为 `take(..)` 的阻塞方法来接收一个值，以及一个称为 `put(..)` 的阻塞方法来发送一个值。

它看起来可能像这样：

```
var ch = channel();

function *foo() {
    var msg = yield take( ch );

    console.log( msg );
}

function *bar() {
    yield put( ch, "Hello World" );

    console.log( "message sent" );
}

run( foo );
run( bar );
// Hello World
// "message sent"
```

将这种结构化的、（看似）同步的消息传递互动，与 `ASQ#runner(..)` 通过 `token.messages` 数组与协作的 `yield` 提供的、非形式化与非结构化的消息共享相比较。实质上， `yield put(..)` 是一种可以同时发送值并为了传递控制权而暂停执行的单一操作，而前一个例子中我们将这两个步骤分开实施。

另外CSP强调，你不会真正明确地“传递控制权”，而是这样设计你的并发过程：要么为了从频道中接

收值而阻塞，要么为了试着向这个频道中发送值而阻塞。这种围绕着消息的发送或接收的阻塞，就是你如何在协程之间协调行为序列的方法。

注意： 预先奉告：这种模式非常强大，但要习惯它有些烧脑。你可能会需要实践它一下，来习惯这种协调并发性的新的思考方式。

有好几个了不起的库已经用 JavaScript 实现了这种风格的CSP，最引人注目的是“js-csp”(https://github.com/ubolonton/js-csp)，由 James Long (http://twitter.com/jlongster)开出的分支(https://github.com/jlongster/js-csp)，以及他特意撰写的作品(http://jlongster.com/Taming-the-Asynchronous-Beast-with-CSP-in-JavaScript)。另外，关于将 ClojureScript 中 go 风格的 core.async CSP 适配到 JS generator 的话题，无论怎么夸赞 David Nolen (http://twitter.com/swannodette) 的许多作品很精彩都不为过 (http://swannodette.github.io/2013/08/24/es6-generators-and-csp)。

asynquence 的 CSP 模拟

因为我们是在我的 *asynquence* 库的上下文环境中讨论异步模式的，你可能会对这个话题很感兴趣：我们可以很容易地在 `ASQ#runner(..)` generator 处理上增加一个模拟层，来近乎完美地移植CSP的API和行为。这个模拟层放在与 *asynquence* 一起发放的 “asynquence-contrib”包的可选部分。

与早先的 `state(..)` 帮助函数非常类似， `ASQ.csp.go(..)` 接收一个 generator —— 用 go/core.async 的术语来讲，它称为一个 goroutine —— 并将它适配为一个可以与 `ASQ#runner(..)` 一起使用的新 generator。

与被传入一个 `token` 不同，你的 goroutine 接收一个创建好的频道（下面的 `ch` ），这个频道会被本次运行的所有 goroutine 共享。你可以使用 `ASQ.csp.chan(..)` 创建更多频道（这通常十分有用）。

在CSP中，我们使用频道消息传递上的阻塞作为所有异步性的模型，而不是为了等待 Promise/序列/thunk 的完成而发生的阻塞。

所以，与 `yield` 从 `request(..)` 中返回的 Promise 不同的是， `request(..)` 应当返回一个频道，你从它那里 `take(..)` 一个值。换句话说，一个单值频道在这种上下文环境/用法上大致上与一个 Promise/序列是等价的。

让我们先制造一个兼容频道版本的 `request(..)` ：

```
function request(url) {
    var ch = ASQ.csp.channel();
    ajax( url ).then( function(content){
        // `putAsync(..)` 是 `put(..)` 的另一个版本，
        // 它可以在一个 generator 的外部使用。它为操作
```

```
        // 的完成返回一个 promise。我们不在这里使用这个
        // promise，但如果有需要的话我们可以在值被
        // `taken(..)` 之后收到通知。
        ASQ.csp.putAsync( ch, content );
    } );
    return ch;
}
```

在第三章中，“promisory”是一个生产 Promise 的工具，第四章中“thunkory”是一个生产thunk的工具，最后，在附录A中我们发明了“sequory”表示一个生产序列的工具。

很自然地，我们需要为一个生产频道的工具杜撰一个对称的术语。所以就让我们不出意料地称它为“chanory”（“channel” + “factory”）吧。作为一个留给读者的练习，请试着亲手定义一个 `channelify(..)` 的工具，就像 `Promise.wrap(..)` / `promisify(..)`（第三章），`thunkify(..)`（第四章），和 `ASQ.wrap(..)`（附录A）一样。

先考虑这个使用 *asyquence* 风格CSP的并发Ajax的例子：

```
ASQ()
.runner(
    ASQ.csp.go( function*(ch){
        yield ASQ.csp.put( ch, "http://some.url.2" );

        var url1 = yield ASQ.csp.take( ch );
        // "http://some.url.1"

        var res1 = yield ASQ.csp.take( request( url1 ) );

        yield ASQ.csp.put( ch, res1 );
    } ),
    ASQ.csp.go( function*(ch){
        var url2 = yield ASQ.csp.take( ch );
        // "http://some.url.2"

        yield ASQ.csp.put( ch, "http://some.url.1" );

        var res2 = yield ASQ.csp.take( request( url2 ) );
        var res1 = yield ASQ.csp.take( ch );

        // 讲结果传递给序列的下一个步骤
        ch.buffer_size = 2;
        ASQ.csp.put( ch, res1 );
        ASQ.csp.put( ch, res2 );
    } )
)
.val( function(res1,res2){
    // `res1` comes from "http://some.url.1"
```

```
    // `res2` comes from "http://some.url.2"
} );
```

消息传递在两个 goroutines 之间进行的 URL 字符串交换是非常直接的。第一个 goroutine 向第一个URL发起一个Ajax请求，它的应答被放进 `ch` 频道。第二个 goroutine 想第二个URL发起一个Ajax请求，然后从 `ch` 频道取下第一个应答 `res1` 。在这个时刻，应答 `res1` 和 `res2` 都被完成且准备好了。

如果在 goroutine 运行的末尾 `ch` 频道还有什么剩余价值的话，它们将被传递进序列的下一个步骤中。所以，为了从最后的 goroutine 中传出消息，把它们 `put(..)` 进 `ch` 。就像展示的那样，为了避免最后的那些 `put(..)` 阻塞，我们通过把 `ch` 的 `buffer_size` 设置为 `2` （默认是 `0` ）来将它切换到缓冲模式。

注意： 更多使用 *asynquence* 风格CSP的例子可以参见这里 (https://gist.github.com/getify/e0d04f1f5aa24b1947ae)。

复习

Promise 和 generator 为我们能够创建更加精巧和强大的异步性提供了基础构建块。

asynquence 拥有许多工具，用于实现 的迭代序列，响应式序列（也就是“可监听对象”），并发协程，甚至 *CSP goroutines*。

将这些模式，与延续回调和 Promise 能力相组合，使得 *asynquence* 拥有了混合不同异步处理的强大功能，一切都整合进一个干净的异步流程控制抽象：序列。

附录C：鸣谢

你不懂JS：异步与性能

附录C：鸣谢

为了这本书和整个系列的诞生，我有很多人要感谢。

首先，我必须感谢我的妻子Christen Simpson，和我的两个孩子Ethan和Emily，忍受着老爹总是在电脑上敲打。即使在没有写书时，我对JavaScript的痴迷也将我的眼睛粘在屏幕上太久了。我从家庭那里借来的时间是这些书可以如此深入和完整地向你，读者，解释JavaScript的原因。我欠我的家庭一切。

我要感谢我在O'Reilly的编辑，他们是Simon St.Laurent和Brian MacDonald，还有其他的编辑和市场员工。和他们一起工作很棒，而且在这种“开源”写作，编辑，和生产的实验期间提供了特别的通融。

感谢许多通过提供编辑意见和订正来参与使这部丛书变得更好的朋友们，他们是Shelley Powers，Tim Ferro，Evan Borden，Forrest L. Norvell，Jennifer Davis，Jesse Harlin，Kris Kowal，Rick Waldron，Jordan Harband，Benjamin Gruenbaum，Vyacheslav Egorov，David Nolen，和许多其他人。一个巨大感谢送给Jake Archibald为本书作序。

感谢社区中无数的朋友们，包括TC39协会的成员，他们和我们分享了那么多的知识，特别是以耐心和细节容忍我无休止的问题和探究。John-David Dalton，Juriy “kangax” Zaytsev，Mathias Bynens，Axel Rauschmayer，Nicholas Zakas，Angus Croll，Reginald Braithwaite，Dave Herman，Brendan Eich，Allen Wirfs-Brock，Bradley Meck，Domenic Denicola，David Walsh，Tim Disney，Peter van der Zee，Andrea Giammarchi，Kit Cambridge，Eric Elliott，和其他许多我甚至不能接触到的人。

你不懂*JS* 系列丛书诞生于Kickstarter，所以我也要感谢我的所有（将近）500位慷慨的支持者，没有他们这部丛书不可能诞生：

> *Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee*

> *Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hodsden, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel בר-לבב Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuț Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave ♥♬★ furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziółkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Carlos Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George,*

> *Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry, Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard*

这部丛书是以开源的风格书写的，包括编辑和生产。我们感激GitHub使这样的事情在社区中成为可能！

再次感谢所有无数的朋友，尽管我不能叫上名字但是我依然亏欠感谢。但愿这部丛书被我们所有人“拥有”，并为增进对JavaScript语言的意识和理解做出贡献，成为当下和未来所有社区贡献者的助益。