任意时刻，一次只能从队列中处理一个事件。执行事件的时候，可能直接或间接地引发一个或多个后续事件。

并发是指两个或多个事件链随时间发展交替执行，以至于从更高的层次来看，就像是同时在运行（尽管在任意时刻只处理一个事件）。

通常需要对这些并发执行的“进程”（有别于操作系统中的进程概念）进行某种形式的交互协调，比如需要确保执行顺序或者需要防止竞态出现。这些“进程”也可以通过把自身分割为更小的块，以便其他“进程”插入进来。

第 2 章

回调

在第 1 章里，我们探讨了与 JavaScript 异步编程相关的概念和术语。我们的关注点是理解处理所有事件（异步函数调用）的单线程（一次一个）事件循环队列。我们还介绍了多个并发模式以不同的方式解释同时运行的事件链或“进程”（任务、函数调用，等等）之间的关系（如果有的话！）。

第 1 章的所有例子都是把函数当作独立不可分割的运作单元来使用的。在函数内部，语句以可预测的顺序执行（在编译器以上的层级！），但是在函数顺序这一层级，事件（也就是异步函数调用）的运行顺序可以有多种可能。

在所有这些示例中，函数都是作为回调（callback）使用的，因为它是事件循环“回头调用”到程序中的目标，队列处理到这个项目的时候会运行它。

你肯定已经注意到了，到目前为止，回调是编写和处理 JavaScript 程序异步逻辑的最常用方式。确实，回调是这门语言中最基础的异步模式。

无数 JavaScript 程序，甚至包括一些最为高深和复杂的，所依赖的异步基础也仅限于回调（当然，它们使用了第 1 章介绍的各种并发交互模式）。回调函数是 JavaScript 的异步主力军，并且它们不辱使命地完成了自己的任务。

但是……回调函数也不是没有缺点。很多开发者因为更好的异步模式 promise（promise 也是“承诺、希望”的意思，此处一语双关）而激动不已。但是，只有理解了某种抽象的目标和原理，才能有效地应用这种抽象机制。

本章将深入探讨这两点，以便弄懂为什么更高级的异步模式（后续章节和附录 B 中将会讨

论）是必需和备受期待的。

2.1 continuation

让我们回到第 1 章中给出的异步回调的例子，为了突出重点，以下稍作了修改：

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

`// A` 和 `// B` 表示程序的前半部分（也就是*现在*的部分），而 `// C` 标识了程序的后半部分（也就是*将来*的部分）。前半部分立刻执行，然后是一段时间不确定的停顿。在未来的某个时刻，如果 Ajax 调用完成，程序就会从停下的位置继续执行后半部分。

换句话说，回调函数包裹或者说封装了程序的延续（continuation）。

让我们进一步简化这段代码：

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

请在这里稍作停留，思考一下你自己会如何（向对 JavaScript 运作机制不甚了解的某位人士）描述这段程序的运行方式。然后试着把你的描述大声说出来。这有助于你理解我接下来要展示的要点。

大多数人刚才可能想到或说出的内容会类似于“执行 A，然后设定一个延时等待 1000 毫秒，到时后马上执行 C”。你的描述准确度如何呢？

也可能进一步修改为“执行 A，设定延时 1000 毫秒，然后执行 B，然后定时到时后执行 C”。这比第一个版本要更精确一些。你能指出其中的区别吗？

尽管第二个版本更精确一些，但是在匹配大脑对这段代码的理解和代码对于 JavaScript 引擎的意义方面，两个版本对这段代码的解释都有不足。这种不匹配既微妙又显著，也正是理解回调作为异步表达和管理方式的缺陷的关键所在。

一旦我们以回调函数的形式引入了单个 continuation（或者几十个，就像很多程序所做的那样！），我们就容许了大脑工作方式和代码执行方式的分歧。一旦这两者出现分歧（这远不是这种分歧出现的唯一情况，我想你明白这一点！），我们就得面对这样一个无法逆转的事实：代码变得更加难以理解、追踪、调试和维护。

2.2 顺序的大脑

我非常确定大多数人都听到过别人自称“能一心多用”。人们试图让自己成为多任务执行者的努力有各种方式，包括从搞笑（比如小孩玩的拍脑袋然后揉肚子这样声东击西的游戏招数）到日常生活（边走路边嚼口香糖），再到十分危险的行为（边开车边发短信）。

但是，我们真的能一心多用吗？我们真的能同时执行两个有意识的、故意的动作，并对二者进行思考或推理吗？我们最高级的大脑功能是以并行多线程的形式运行的吗？

答案可能出乎你的意料：很可能并不是这样。

看起来我们的大脑并不是以这样的方式构建起来的。很多人（特别是 A 型人）可能不愿意承认，但我们更多是单任务执行者。实际上，在任何特定的时刻，我们只能思考一件事情。

我这里所说的并不是所有我们不自觉、无意识地自动完成的脑功能，比如心跳、呼吸和眨眼等。对维持生命来说，这些是至关重要的，但我们并不需要有意识地分配脑力来执行这些任务。谢天谢地，当我们忙于在 3 分钟内第 15 次查看社交网络更新时，我们的大脑在后台（多线程！）执行了所有这些重要任务。

我们在讨论的是此时处于意识前端的那些任务。对我来说，此时此刻的任务就是编写本书。就在此刻，我还在执行任何其他更高级的脑功能吗？不，并没有。我很容易分心，并且频繁地分心——写前面几段的时候就分心了几十次！

我们在假装并行执行多个任务时，实际上极有可能是在进行快速的上下文切换，比如与朋友或家人电话聊天的同时还试图打字。换句话说，我们是在两个或更多任务之间快速连续地来回切换，同时处理每个任务的微小片段。我们切换得如此之快，以至于对外界来说，我们就像是在并行地执行所有任务。

这听起来是不是和异步事件并发机制（比如 JavaScript 中的形式）很相似呢？！如果你还没意识到的话，就回头把第 1 章再读一遍吧！

实际上，把广博复杂的神经学简化（即误用）为一种这里我足以讨论的形式就是，我们大脑的工作方式有点类似于事件循环队列。

如果把我打出来的每个字母（或单词）看作一个异步事件，那么在这一句中我的大脑就有几十次机会被其他某个事件打断，比如因为我的感官甚至随机思绪。

我不会在每次可能被打断的时候都转而投入到其他“进程”中（这值得庆幸，否则我根本没法写完本书！）。但是，中断的发生经常频繁到让我觉得我的大脑几乎是不停地切换到不同的上下文（即“进程”）中。很可能 JavaScript 引擎也是这种感觉。

2.2.1 执行与计划

好吧，所以我们的大脑可以看作类似于单线程运行的事件循环队列，就像 JavaScript 引擎那样。这个比喻看起来很贴切。

但是，我们的分析还需要比这更加深入细致一些。显而易见的是，在我们如何计划各种任务和我们的大脑如何实际执行这些计划之间，还存在着很大的差别。

再一次用此书的写作进行类比。此刻，我心里大致的计划是写啊写啊一直写，依次完成我脑海中已经按顺序排好的一系列要点。我并没有将任何中断或非线性的行为纳入到我的写作计划中。然而，尽管如此，实际上我的大脑还是在不停地切换状态。

虽然在执行的层级上，我们的大脑是以异步事件方式运作的，但我们的任务计划似乎还是以顺序、同步的方式进行："我要先去商店，然后买点牛奶，然后去一下干洗店。"

你会注意到，这个较高层级的思考（计划）过程看起来并不怎么符合异步事件方式。实际上，我们认真思考的时候很少是以事件的形式进行的。取而代之的是，我们按照顺序（A，然后 B，然后 C）仔细计划着，并且会假定有某种形式的临时阻塞来保证 B 会等待 A 完成，C 会等待 B 完成。

开发者编写代码的时候是在计划一系列动作的发生。优秀的开发者会认真计划。"我需要把 z 设为 x 的值，然后把 x 设为 y 的值"，等等。

编写同步代码的时候，语句是一条接一条执行的，其工作方式非常类似于待办任务清单。

```
// 交换x和y(通过临时变量z)
z = x;
x = y;
y = z;
```

这三条语句是同步执行的，所以 `x = y` 会等待 `z = x` 执行完毕，然后 `y = z` 等待 `x = y` 执行完毕。换个说法就是，这三条语句临时绑定按照特定顺序一个接一个地执行。谢天谢地，这里我们不需要处理异步事件的细节。如果需要的话，代码马上就会变得复杂得多！

所以，如果说同步的大脑计划能够很好地映射到同步代码语句，那么我们的大脑在规划异步代码方面又是怎样的呢？

答案是代码（通过回调）表达异步的方式并不能很好地映射到同步的大脑计划行为。

实际上你能想象按照以下思路来计划待办任务吗？

> "我要去商店，但是路上肯定会接到电话。'嗨，妈妈。'然后她开始说话的时候，我要在 GPS 上查找商店的地址，但是 GPS 加载需要几秒钟时间，于是我把收音机的音量关小，以便听清妈妈讲话。接着我意识到忘了穿外套，外面有点冷，不

过没关系，继续开车，继续和妈妈打电话。这时候安全带警告响起，提醒我系好安全带。‘是的，妈妈，我系着安全带呢。我一直都有系啊！’啊，GPS 终于找到方向了，于是……”

如果我们这样计划一天中要做什么以及按什么顺序来做的话，事实就会像听上去那样荒谬。但是，在实际执行方面，我们的大脑就是这么运作的。记住，不是多任务，而是快速的上下文切换。

对我们程序员来说，编写异步事件代码，特别是当回调是唯一的实现手段时，困难之处就在于这种思考 / 计划的意识流对我们中的绝大多数来说是不自然的。

我们的思考方式是一步一步的，但是从同步转换到异步之后，可用的工具（回调）却不是按照一步一步的方式来表达的。

这就是为什么精确编写和追踪使用回调的异步 JavaScript 代码如此之难：因为这并不是我们大脑进行计划的运作方式。

唯一比不知道代码为什么崩溃更可怕的事情是，不知道为什么一开始它是工作的！这就是经典的“纸牌屋”心理：“它可以工作，可我不知道为什么，所以谁也别碰它！”你可能听说过“他人即地狱”（萨特）这种说法，对程序员来说则是“他人的代码即地狱”。而我深信不疑的是：“不理解自己的代码才是地狱。”回调就是主要元凶之一。

2.2.2 嵌套回调与链式回调

考虑：

```
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) ;
} );
```

你很可能非常熟悉这样的代码。这里我们得到了三个函数嵌套在一起构成的链，其中每个函数代表异步序列（任务，“进程”）中的一个步骤。

这种代码常常被称为回调地狱（callback hell），有时也被称为毁灭金字塔（pyramid of

doom，得名于嵌套缩进产生的横向三角形状）。

但实际上回调地狱与嵌套和缩进几乎没有什么关系。它引起的问题要比这些严重得多。本章后面的内容会就此类问题的现象和原因展开讨论。

一开始我们在等待 `click` 事件，然后等待定时器启动，然后等待 Ajax 响应返回，之后可能再重头开始。

一眼看去，这段代码似乎很自然地将其异步性映射到了顺序大脑计划。

首先（*现在*）我们有：

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

然后是*将来*，我们有：

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

接着还是*将来*，我们有：

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

最后（最晚的*将来*），我们有：

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

但以这种方式线性地追踪这段代码还有几个问题。

首先，例子中的步骤是按照 1、2、3、4……的顺序，这只是一个偶然。实际的异步 JavaScript 程序中总是有很多噪声，使得代码更加杂乱。在大脑的演习中，我们需要熟练地绕过这些噪声，从一个函数跳到下一个函数。对于这样满是回调的代码，理解其中的异步流不是不可能，但肯定不自然，也不容易，即使经过大量的练习也是如此。

另外，其中还有一个隐藏更深的错误，但在代码例子中，这个错误并不明显。我们另外设计一个场景（伪代码）来展示这一点：

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

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

        doE();
    } );

    doF();
```

尽管有经验的你能够正确确定实际的运行顺序，但我敢打赌，这比第一眼看上去要复杂一些，需要费一番脑筋才能想清楚。实际运行顺序是这样的：

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

你第一眼看到前面这段代码就分析出正确的顺序了吗？

好吧，有些人可能会认为我的函数命名有意误导了大家，所以不怎么公平。我发誓，我只是按照从上到下的出场顺序命名的。不过还是让我再试一次吧：

```
doA( function(){
    doC();

    doD( function(){
        doF();
    } )

    doE();
} );

doB();
```

现在，我是按照实际执行顺序来命名的。但我还是敢打赌，即使对这种情况有了经验，也不能自然而然地就追踪到代码的执行顺序 A → B → C → D → E → F。显然，你需要在代码中不停地上下移动视线，对不对？

但即使你能够很轻松地得出结论，还是有一个可能导致严重问题的风险。你能够指出这一点吗？

如果 doA(..) 或 doD(..) 实际并不像我们假定的那样是异步的，情况会如何呢？啊，那顺序就更麻烦了。如果它们是同步的（或者根据程序当时的状态，只在某些情况下是同步的），那么现在运行顺序就是 A → C → D → F → E → B。

现在你听到的背景中模糊的声音就是无数 JavaScript 开发者的掩面叹息。

问题是出在嵌套上吗？是它导致跟踪异步流如此之难吗？确实，部分原因是这样。

但是，让我们不用嵌套再把前面的嵌套事件 / 超时 /Ajax 的例子重写一遍吧：

```
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();
    }
}
```

这种组织形式的代码不像前面以嵌套 / 缩进的形式组织的代码那么容易识别了，但是它和回调地狱一样脆弱，易受影响。为什么？

在线性（顺序）地追踪这段代码的过程中，我们不得不从一个函数跳到下一个，再跳到下一个，在整个代码中跳来跳去以“查看”流程。而且别忘了，这还是简化的形式，只考虑了最优情况。我们都知道，真实的异步 JavaScript 程序代码要混乱得多，这使得这种追踪的难度会成倍增加。

还有一点需要注意：要把步骤 2、步骤 3 和步骤 4 连接在一起让它们顺序执行，只用回调的话，代价可以接受的唯一方式是把步骤 2 硬编码到步骤 1 中，步骤 3 硬编码到步骤 2 中，步骤 4 硬编码到步骤 3 中，以此类推。如果实际上步骤 2 总会引出步骤 3 是一个固定条件的话，硬编码本身倒不一定是坏事。

但是，硬编码肯定会使代码更脆弱一些，因为它并没有考虑可能导致步骤执行顺序偏离的异常情况。比如，如果步骤 2 失败，就永远不会到达步骤 3，不管是重试步骤 2，还是跳转到其他错误处理流程，等等。

这些问题都可以通过在每个步骤中手工硬编码来解决，但这样的代码通常是重复的，并且在程序中的其他异步流中或其他步骤中无法复用。

尽管我们的大脑能够以顺序的方式（这个，然后这个，然后这个）计划一系列任务，但大脑运作的事件化的本质使得控制流的恢复 / 重试 / 复制几乎不费什么力气。如果你出外办

事的时候发现把购物清单落在了家里，那么这一天并不会因为你没有预知到这一点就成为世界末日了。你的大脑很容易就能针对这个小意外做出计划：回家拿清单，然后立刻返回商店就是了。

但是，手工硬编码（即使包含了硬编码的出错处理）回调的脆弱本性可就远没有这么优雅了。一旦你指定（也就是预先计划）了所有的可能事件和路径，代码就会变得非常复杂，以至于无法维护和更新。

这才是回调地狱的真正问题所在！嵌套和缩进基本上只是转移注意力的枝节而已。

如果这还不够的话，我们还没有提及两个或更多回调 continuation 同时发生的情况，或者如果步骤 3 进入了带有 gate 或 latch 的并行回调的分支，还有……不行，我脑子转不动了，你怎么样？！

现在你抓住重点了吗？我们的顺序阻塞式的大脑计划行为无法很好地映射到面向回调的异步代码。这就是回调方式最主要的缺陷：对于它们在代码中表达异步的方式，我们的大脑需要努力才能同步得上。

2.3 信任问题

顺序的人脑计划和回调驱动的异步 JavaScript 代码之间的不匹配只是回调问题的一部分。还有一些更深入的问题需要考虑。

让我们再次思考一下程序中把回调 continuation（也就是后半部分）的概念：

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

`// A` 和 `// B` 发生于*现在*，在 JavaScript 主程序的直接控制之下。而 `// C` 会延迟到*将来*发生，并且是在第三方的控制下——在本例中就是函数 `ajax(..)`。从根本上来说，这种控制的转移通常不会给程序带来很多问题。

但是，请不要被这个小概率迷惑而认为这种控制切换不是什么大问题。实际上，这是回调驱动设计最严重（也是最微妙）的问题。它以这样一个思路为中心：有时候 `ajax(..)`（也就是你交付回调 continuation 的第三方）不是你编写的代码，也不在你的直接控制下。多数情况下，它是某个第三方提供的工具。

我们把这称为*控制反转*（inversion of control），也就是把自己程序一部分的执行控制交给某个第三方。在你的代码和第三方工具（一组你希望有人维护的东西）之间有一份并没有明确表达的契约。

2.3.1 五个回调的故事

可能现在还不能很明显地看出为什么这是一个大问题。让我构造一个有点夸张的场景来说明这种信任风险吧。

假设你是一名开发人员，为某个销售昂贵电视的网站建立商务结账系统。你已经做好了结账系统的各个界面。在最后一页，当用户点击“确定”就可以购买电视时，你需要调用（假设由某个分析追踪公司提供的）第三方函数以便跟踪这个交易。

你注意到，可能是为了提高性能，他们提供了一个看似用于异步追踪的工具，这意味着你需要传入一个回调函数。在传入的这个 continuation 中，你需要提供向客户收费和展示感谢页面的最终代码。

代码可能是这样：

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

很简单，是不是？你写好代码，通过测试，一切正常，然后就进行产品部署。皆大欢喜！

六个月过去了，没有任何问题。你几乎已经忘了自己写过这么一段代码。某个上班之前的早晨，你像往常一样在咖啡馆里享用一杯拿铁。突然，你的老板惊慌失措地打电话过来，让你放下咖啡赶紧到办公室。

到了办公室，你得知你们的一位高级客户购买了一台电视，信用卡却被刷了五次，他很生气，这可以理解。客服已经道歉并启动了退款流程。但是，你的老板需要知道这样的事情为何会出现。“这种情况你没有测试过吗？！”

你甚至都不记得自己写过这段代码。但是，你得深入研究这些代码，并开始寻找问题产生的原因。

通过分析日志，你得出一个结论：唯一的解释就是那个分析工具出于某种原因把你的回调调用了五次而不是一次。他们的文档中完全没有提到这种情况。

沮丧的你联系他们的客服，而客服显然和你一样吃惊。他们保证，一定会向开发者提交此事，之后再给你回复。第二天，你收到一封很长的信，是解释他们的发现的，于是你立刻将其转发给你的老板。

显然，分析公司的开发者开发了一些实验性的代码，在某种情况下，会在五秒钟内每秒重试一次传入的回调函数，然后才会因超时而失败。他们从来没打算把这段代码提交到产品中，但不知道为什么却这样做了，他们很是尴尬，充满了歉意。他们以漫长的篇幅解释了他们是如何确定出错点的，并保证绝不会再发生同样的事故，等等。

然后呢？

你和老板讨论此事，他对这种状况却不怎么满意。他坚持认为，你不能再信任他们了（你们受到了伤害）。对此你也只能无奈接受，并且你需要找到某种方法来保护结账代码，保证不再出问题。

经过修补之后，你实现了像下面这样的简单临时代码，大家似乎也很满意：

```
var tracked = false;

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

经过第 1 章之后，这段代码对你来说应该很熟悉，因为这里我们其实就是创建了一个 latch 来处理对回调的多个并发调用。

但是，后来有一个 QA 工程师问道：“如果他们根本不调用这个回调怎么办？”哎呦！之前你们双方都没有想到这一点。

然后，你开始沿着这个兔子洞深挖下去，考虑着他们调用你的回调时所有可能的出错情况。这里粗略列出了你能想到的分析工具可能出错的情况：

- 调用回调过早（在追踪之前）；
- 调用回调过晚（或没有调用）；
- 调用回调的次数太少或太多（就像你遇到过的问题！）；
- 没有把所需的环境 / 参数成功传给你的回调函数；
- 吞掉可能出现的错误或异常；
- ……

这感觉就像是一个麻烦列表，实际上它就是。你可能已经开始慢慢意识到，对于被传给你无法信任的工具的每个回调，你都将不得不创建大量的混乱逻辑。

现在你应该更加明白回调地狱是多像地狱了吧。

2.3.2 不只是别人的代码

有些人可能会质疑这件事情是否真像我声称的那么严重。可能你没有真正和第三方工具打

过很多交道，如果并不是完全没有的话。可能你使用的是带版本的 API 或者自托管的库，所以其行为不会在你不知道的情况下被改变。

请思考这一点：你能够真正信任理论上（在自己的代码库中）你可以控制的工具吗？

不妨这样考虑：多数人都同意，至少在某种程度上我们应该在内部函数中构建一些防御性的输入参数检查，以便减少或阻止无法预料的问题。

过分信任输入：

```
function addNumbers(x,y) {
    // +是可以重载的,通过类型转换,也可以是字符串连接
    // 所以根据传入参数的不同,这个运算并不是严格安全的
    return x + y;
}

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

针对不信任输入的防御性代码：

```
function addNumbers(x,y) {
    // 确保输入为数字
    if (typeof x != "number" || typeof y != "number") {
        throw Error( "Bad parameters" );
    }

    // 如果到达这里,可以通过+安全的进行数字相加
    return x + y;
}

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

依旧安全但更友好一些的：

```
function addNumbers(x,y) {
    // 确保输入为数字
    x = Number( x );
    y = Number( y );

    // +安全进行数字相加
    return x + y;
}

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

不管你怎么做，这种类型的检查 / 规范化的过程对于函数输入是很常见的，即使是对于理论上完全可以信任的代码。大体上说，这等价于那条地缘政治原则：“信任，但要核实。”

所以，据此是不是可以推断出，对于异步函数回调的组成，我们应该要做同样的事情，而不只是针对外部代码，甚至是我们知道在我们自己控制下的代码？当然应该。

但是，回调并没有为我们提供任何东西来支持这一点。我们不得不自己构建全部的机制，而且通常为每个异步回调重复这样的工作最后都成了负担。

回调最大的问题是控制反转，它会导致信任链的完全断裂。

如果你的代码中使用了回调，尤其是但也不限于使用第三方工具，而且你还没有应用某种逻辑来解决所有这些控制反转导致的信任问题，那你的代码现在已经有了 bug，即使它们还没有给你造成损害。隐藏的 bug 也是 bug。

确实是地狱。

2.4 省点回调

回调设计存在几个变体，意在解决前面讨论的一些信任问题（不是全部！）。这种试图从回调模式内部挽救它的意图是勇敢的，但却注定要失败。

举例来说，为了更优雅地处理错误，有些 API 设计提供了*分离回调*（一个用于成功通知，一个用于出错通知）：

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

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

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

在这种设计下，API 的出错处理函数 `failure()` 常常是可选的，如果没有提供的话，就是假定这个错误可以吞掉。

ES6 Promise API 使用的就是这种分离回调设计。第 3 章会介绍 ES6 Promise 的更多细节。

还有一种常见的回调模式叫作“error-first 风格”（有时候也称为“Node 风格”，因为几乎所有 Node.js API 都采用这种风格），其中回调的第一个参数保留用作错误对象（如果有的话）。如果成功的话，这个参数就会被清空 / 置假（后续的参数就是成功数据）。不过，如

果产生了错误结果，那么第一个参数就会被置起 / 置真（通常就不会再传递其他结果）：

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

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

在这两种情况下，都应该注意到以下几点。

首先，这并没有像表面看上去那样真正解决主要的信任问题。这并没有涉及阻止或过滤不想要的重复调用回调的问题。现在事情更糟了，因为现在你可能同时得到成功或者失败的结果，或者都没有，并且你还是不得不编码处理所有这些情况。

另外，不要忽略这个事实：尽管这是一种你可以采用的标准模式，但是它肯定更加冗长和模式化，可复用性不高，所以你还得不厌其烦地给应用中的每个回调添加这样的代码。

那么完全不调用这个信任问题又会怎样呢？如果这是个问题的话（可能应该是个问题！），你可能需要设置一个超时来取消事件。可以构造一个工具（这里展示的只是一个“验证概念”版本）来帮助实现这一点：

```
function timeoutify(fn,delay) {
    var intv = setTimeout( function(){
            intv = null;
            fn( new Error( "Timeout!" ) );
        }, delay )
    ;

    return function() {
        // 还没有超时?
        if (intv) {
            clearTimeout( intv );
            fn.apply( this, arguments );
        }
    };
}
```

以下是使用方式：

```
// 使用"error-first 风格" 回调设计
function foo(err,data) {
    if (err) {
        console.error( err );
```