Node.js 实战 (中国程序员 6)

作者：【葡】Pedro Teixeira 著

Node.js 实战

【葡】Pedro Teixeira 著

刘亚中 张京 张耀春 译

策划编辑：杜正彬

封面设计：梁杰

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目录

译者序一

Node.js是一个非常具有极客精神的技术社区（我更想强调它本身代表的是一个社区，相比之前的一些大语言分支 c/c++/java什么的，那些大语言在针对某一些应用都有很固定的范式去遵循，javascript/node.js还是一个比较年轻的语言或“技术”，所以对于大部分应用都没有任何范式，还有很大的空间去发挥。所以在1.0发布之前 我觉得node.js的社区属性多于技术，因为我个人认为技术必须是成熟稳定 才会去让人去学技术本身。这里我把nodejs称为社区的原因也是希望读者不要过多关注技术，就像我下面说的，太多东西变化太快，关注社区 才会永远知道这门“技术”的趋势 并且如果可能要在1.0之前都尽量参与其中。），无论从设计思路、API还是其背后的开发团队，当然也包括大量的模块开发者，你还有我，这其中当然包括本书的作者Pedro，他一直活跃在Node.js社区，开发了很多有趣有用的软件，包括体感游戏机Kinect的nodejs客户端、HTTP模拟框架nock等。

回到本书中来，书中并没有将内容写成作为API手册，而是一步步地，从如何安装模组，如何输出变量来进行简单测试，然后介绍了Buffer、EventEmitter、Timer等这些最基本的工具，接着分别介绍了几个I/O模组：文件、HTTP、流、TCP/UDP、TLS。然后最后几章中结合前面所讲，综合了各种基础模组的用法与设计思想，深入介绍了自动化、调试、回调这几个略微高级但极其重要的部分。

这本书不仅适合Node.js初学者用户，同样也适合有过一段开发经验但阅读相关书籍较少的同学，在阅读的过程中，你将会对自己之前认识不够清晰的一些技术细节有全新的认识，起到梳理知识点的作用（我自己就是其中的受益者）。

最后，由于Node.js本身确实更迭过快，甚至它还没有发布1.0版本，因此书中有很多代码以及部分内容已经与当前的Node.js源代码不符，或者说有了某种程度的更新，因此你在阅读本书的时候最终应当回归到Node.js自身的源代码，然而你从书中学到更多的应该是本书作者Pedro在分析问题以及代码风格上的优点，而不应停留在内容本身。

刘亚中

译者序二

在国内的Nodejs书中，我推荐这本书，本书内容比较全面，适合各个阶段的Node学习者，知识点覆盖度比较广，基本囊括了node的大部分组成，甚至包含调试和自动化测试相关。

读完本书后，你会了解到所谓的node世界其实都包含什么。

本书后面的习题也会帮助你更深地去了解部分章节的内容。

当然了，node的版本也在更新，本书的部分内容译者也做了一些更新，方便大家对新的参数和方法进行理解。

张耀春

译者序三

Node.js 是当前的一个热点，似乎不知道它，我都不好意思跟写代码的朋友打招呼。但我却认为它为我们开辟了一条新的道路，让我们使用HTML，CSS和JavaScript开发动态网站成为可能。更为吸引人的是它跨平台的特性，Linux, Mac, windows都对它提供了支持。原本，我是从学习Asp开始入门的，被微软“裹胁”着一路从Asp.net1.1一直到当前的Asp.net4.5。不能说微软的技术不好，只是觉得被人一直牵着鼻子。有了Node.js, 让我感到从未有过的畅快。

为了学习Node.js， 我看了五本中文书籍，仔细的阅读了Node.js的官方英文文档，并且在百度上查询了许多资料。但比起这本书来，它们都有这样那样的遗憾。我很怀疑这本书的作者是位老师。这是因为他写作的顺序十分有调理，学习曲线很平稳的上升，一步一步让人逐渐掌握Node.js。作者的选材也十分讲究，选择了Node.js最关键最常用的部分进行系统的讲解，并且编写了实用的练习题供读者复习。作者精心编写的示例既简单明了，而且也十分经典。更为难得的是，虽然Node.js更新很快，书中的示例依然在新版的Node.js中可以执行，并且运行良好。这说明作者对于Node.js的理解已经达到大师一级了。我通过翻译这本书，也是收益匪浅。我强烈推荐初学者将这本书作为Node.js的首选入门书籍。

当然，最好的学习Node.js和学习任何一门程序语言一样，那就是亲自动手编写代码。在一次一次编写代码，测试，纠错中逐渐精通这门语言。希望这本书能够帮助读者在学习中少走弯路，快速上手，体会到编程的快乐。

感谢这本书另外2位合译作者（张耀春和刘亚中）和杜杜正彬编辑，我们一起合作完成了这本书的翻译，这是十分难得的一次愉快的合作。

张京

译者介绍

刘亚中

github爱好者 node.js贡献世界排名21 开源组织clibs团队成员，先后给node libuv npm mongoose等40余个开源项目贡献过代码 美国创业产品Pixbi团队成员 英国数据公司51Degrees nodejs顾问。

张耀春

豌豆荚FE，专职博客5年，拥有1200+博文发表在iteye上，常年混迹cnode和w3cplus，参加过w3ctech的node演讲，nodejs重度观察者，微博专业node号dailyNodejs发起者与分享者。

张京

美国西雅图城市大学MBA, 中国语言文化大学的文学学士，做过10多年的销售，之后有在全球领先的差旅管理企业主持开发企业级的差旅预定平台。自学编程8年，热爱nodejs, 在cnode中灌水。目前加入一家IDG风投的创业企业负责CRM开发、供应商管理、运营管理以及其他没人管的事。天生爱折腾!

校对

alsotang

阿里巴巴数据平台 Node.js 工程师。CNode 社区管理者之一。

简介

在欧洲JSConf2009大会上，一位名叫Ryan Dahl的年轻程序员介绍了他正在研究的项目，这是一个结合Google V8 JavaScript引擎和事件循环的服务器端项目，它使用JavaScript语言开发。这个项目与其它服务器端的JavaScript平台不同：所有的I/O都是原生的事件驱动，并且无法使用其它 IO 方式。

凭借着JavaScript语言的强大功能以及其易用性，使得编写异步应用程序的困难任务变得容易了。在他演讲结束时，听众响起鼓掌，他的项目也得到了空前的发展，并迅速流行起来，被大家广泛采用。

这个项目叫做Node.js, 在程序员中被简称为'Node'. Node可以用来以纯粹的事件驱动，非阻塞的架构，构建可以高效处理并发的软件。

Node可以助你轻松构建快速反应和可扩展的网络服务。

为什么突然以指数级的增长方式流行起来了？

服务器端的JavaScript存在已经有几年了，到底是什么东西让 Node 这个项目变得这么引人注目？

之前服务器端的JavaScript实际应用程序中，其存在理由以及实现方式主要是将其它平台，例如Ruby, PerL 和Python的经典最佳实践方式移植过来。 Node实现了跨越式的发展，它的声音是：“让我们使用在网络上行之有效的事件驱动模式可以轻松构建可扩展的服务器，并且是在这个平台上的唯一方式，它能让人们做任何事。”

对于Node的成功，JavaScript语言本身到底贡献有多大是有争论的，但这并不能解释为什么Node的流行程度远远超过其它服务端项目。当然，无所不在的JavaScript对其成功起到了举足轻重的作用，但Ryan Dahl指出，与其它服务器端的JavaScript项目不同，Node的主要目的不是统一服务器端和客户端的开发语言。

在我看来，Node的成功有3个因素：

1. 简单 – 以事件驱动方式编程处理I/O，相对于其它现有的平台，它更易理解、更易实现，这是最佳的处理I/O的方法。

2. 简洁 – Node并不想解决所有问题。它使用干净并高效的API构建平台基础，只支持基本的互联网通讯协议。

3. 不妥协 – Node不去尝试兼容现有的程序，它不走平常路，不理他人的闲言碎语。

这本书聊些什么？

我们将分析Node是如何提供与众不同的服务端解决方案、你应该使用它的理由、以及如何启动它。我们将以一个整体介绍作为开头，但不会耽误很久，之后很快便会进入到程序的模块中。在本书结束时，你将能够构建并测试你自己的Node模块，面向服务的生产者线程/消费者线程（service producers/consumers）， 轻松的运用Node编码约定和API。

这本书不谈什么？

这本书不会介绍整个Node API.。正相反， 我们将探讨使用Node构建应用程序时，作者认为需要了解的相关内容。

这本书不会介绍任何Node框架：Node是构建框架的非常强大的工具，并且已经有很多现成的框架了，例如cluster模块管理，进程间通讯，web框架，网络流量采集工具，游戏引擎等等。在你深入了解以上这些框架之前，你应当熟悉Node的基础结构，并且了解Node能为构建这些框架提供什么。

阅读建议

本书没有假设你已经掌握Node的相关知识，但如果你对JavaScript有所了解，将对你阅读本书有帮助，这是因为所有示例都是使用JavaScript编写的。

练习题

在某些章节，本书提供了一些练习题。在本书结尾，你能找到这些练习题的答案，但我还是建议你自己尝试解决。如需进一步了解本书内容或更全面的API文档，请浏览Node官方网站 http://nodejs.org 。

源代码

在GitHub上，你可以找到本书的源代码和练习题： https://github.com/pgte/handson_nodejs_source_code

你也可以直接下载本书的源代码和练习题： https://github.com/pgte/handson_nodejs_source_code/zipball/master

本书将带你前往何处？

阅读本书后，你应当可以理解Node API，并且能够在之上进一步探索构建其它应用，例如adaptors、框架和模块。

为何Node？(why node)

事件循环(Event Loop) (why the event loop)

事件循环是一种利于系统进行非阻塞I/O（网络、文件以及进程内通信）的软件方法(Software Pattern)。老式的阻塞编程使用与一般的函数调用类似的方式来处理I/O：即处理进程会阻塞当前操作，直至完成。下面的伪代码可以让你简单理解阻塞I/O的基本过程：

```
var post = db.query('SELECT * FROM posts where id = 1');
// 从这一行开始，代码将不会继续往下运行，直至上一行的查询(query)操作完成
doSomethingWithPost(post);
doSomethingElse();
```

上面的代码究竟发生了什么？当数据库查询操作执行之后，程序进程/线程将处于空闲(Idle)的状态，等待着数据库查询返回结果，这就是调用阻塞。查询操作从额开始到响应可能会花掉成百上千的CPU周期，这将会造成这段时间内进程停顿。进程可能在之前就已经在为其他客户端工作，不过现在只有等到查询操作果响应后，才能继续之前的工作。

调用阻塞这种方式让并发I/O编程变得十分艰难，比如执行另一条数据库查询指令，与其他远程web服务进行通信等。程序的调用堆栈被耗时的I/O操作冻结，直到数据库服务器响应后才会恢复正常。

为了让进程在等待I/O的同时也可以继续运行其他代码，这里提供了两种可用的解决方案：创建更多的调用堆栈，或者使用事件回调函数。

方案1: 创建更多的调用堆栈 (solution1: create more call stack)

为了让进程可以同时处理更多的并发I/O，就需要更多的并发调用堆栈。为此你可以使用线程或者协作式多线程的方法：协程、纤程等。

多线程并发的配置、理解和调试过程都十分困难，主要原因是在访问/修改多线程中的共享状态量(Shared State)时同步过程的复杂性。程序员根本无法预知运行中的线程什么时候会突然退出，这样很容易导致一些奇怪并且难以重现的Bug。

为了解决多线程不可预知的问题，协作式多线程另辟蹊径，它被设定为不止一个调用堆栈，在某个"线程"在执行时需要明确地声明退出当前调度并让另一个并发"线程"接手工作。这样确实减缓了同步所带来的复杂度，不过写出来的程序会更加复杂，另外，由于程序员需要自己实现线程调度的过程，程序错误的几率也大大上升。

方案2: 使用事件驱动I/O (solution2: use event-driven I/O)

事件驱动I/O是一种计算机程序的运行方法，通常你需要向系统先注册回调函数，当指定的事件发生时，之前注册的函数就会执行。

事件回调是一个函数，只会在满足特定条件后被调用，例如之前数据库查询返回结果后。

为了在之前的例子中使用事件回调，你需要更改你的代码：

```
var callback = function(post) {
    doSomethingWithPost(post); // 仅在db.query()返回结果之后被调用
});
db.query('SELECT * FROM posts where id = 1', callback);
doSomethingElse();  // 这行代码将在db.query返回调用状态后单独地运行
```

上面的例子中，先定义了一个函数，并假定在数据库操作完成后被调用，接着你将这个函数作为回调参数告诉db.query，之后db.query就会履行约定，在查询结果返回后调用它。

下面我们使用一个内联的匿名函数来让整段程序看起来更紧凑一些：

```
db.query('SELECT * FROM posts where id = 1',
   function(post) {
       doSomethingWithPost(post); // 仅在db.query()返回结果之后被调用
   }
  );
  doSomethingElse(); // // 这行代码将在db.query返回调用状态后单独地运行
```

在处理db.query()的过程中，程序可以接着运行doSomethngElse()，或者甚至接受另外的请求。

长久以来，在C语言系统编程的"黑客"社区内，事件驱动早已被承认是作为拓展服务器来处理高并发连接的最佳实践。与多线程或其他方法相比，事件驱动被公认为更高效地使用了内存（更少的上下文需要存储）和时间（更少的上下文切换）。

这样的看法也逐渐渗透到其他的编程平台和社区当中，其中比较知名的事件循环实现有：Ruby的Event Machine、Perl的AnyEvent以及Python的Twisted等等。

贴士：访问 http://en.wikipedia.org/wiki/Reactor_pattern，可以获得更多关于服务器实现事件驱动的资源。

要实现一个基于以上这些事件驱动框架的应用程序需要相关框架的知识和代码库。比如你想使用Event Machine，就必须放弃使用Ruby的同步代码库。那么为了从非阻塞编程中获益，你将受限于Event Machine中有限的代码库。而在Event Machine中使用同步代码库，由于事件循环时常被阻塞，I/O事件的处理时间延长直接导致程序运行起来很慢，这违背了事件驱动的设计初衷，服务器拓展的效果也就大打折扣。

Node从开始设计的第一天起，就被定位于非阻塞I/O的服务器平台，这意味着在这个平台上，每一件事都是非阻塞的。由于JavaScript本身内含很小，也没有提供任何有关I/O的代码库（不像C/Ruby/Python，JavaScript并没有一个标准的I/O代码库），因此Node就很清楚自己的任务，从零开始，搭建自己的非阻塞软件体系。

为什么是JavaScript? (Why JavaScript)

Ryan Dahl在构建Node平台之初，他选择是的C语言，但很快他就意识到要在很多回调函数中维护上下文是十分复杂的，也会破坏整体的代码结构。因此他把目光转向了Lua。

Lua已经拥有若干阻塞的I/O代码库，如果硬要把阻塞和非阻塞混在一起，只会迷惑大多数开发者，如此就没办法构建可拓展的软件了，因此Lua也不理想。接着Dahl开始考虑JavaScript。

JavaScript支持函数的闭包(Closure)，并且将函数作为整个语言的一等"公民"而存在，这两个重要的特性使得这门语言和事件驱动I/O编程非常契合。

闭包系函数从闭包环境可以继承变量。当回调函数执行时，JavaScript运行时会记录下函数在声明时的上下文及其内部所有变量，同时也可以访问到父作用域的变量，依此类推。笔者认为这一特性才是为什么Node在编程社区能获得如此成功的最重要因素。

在Web浏览器中，如果想要监听一个事件（例如按钮单击事件）：

```
var clickCount = 0;
```

```
document.getElementById('mybutton').onclick = function() {
   clickCount ++;
   alert('Clicked ' + clickCount + ' times.');
  };
```

或使用jQuery来简单代码：

```
var clickCount = 0;
$('button#mybutton').click(function() {
    clickCount ++;
    alert('Clicked ' + clickCount + ' times.');
});
```

从上面的两个例子中，我们将事件函数赋值给另一个变量或者当作参数传给另一个函数。相关事件（本例中是点击按钮）一旦触发，事件函数便会执行。事件函数可以访问在声明函数时的作用域内的每一个变量，具体地说就是我们可以在事件函数内访问到父作用域内声明的变量clickCount。

这里我们使用了全局变量clickCount来累加用户点击按钮的次数。如果我们不希望系统的其他代码可以访问到clickCount，就不能使用全局变量，因此我们用一个闭包函数将上面的代码包裹，让clickCount只能在刚创建的闭包函数的作用域内访问：

```
(function() {
    var clickCount = 0;
    $('button#mybutton').click(function() {
        clickCount ++;
        alert('Clicked ' + clickCount + ' times.');
    });
})();
```

在例子的第七行，我们定义了闭包函数后就立即执行它。如果你对此用法陌生，请不用担心，之后我们会对它进行说明。

从上面的几例中，你不需要再担心同步复杂所带来的问题：在回调函数返回之前，我们一定可以确保它不会中断程序的其他部分。

停止害怕，我爱JS (How I learn to stop fearing and love JavaScript)

JavaScript有好有坏。1995年，Netscape的Brendan Eich创造了它，匆匆开发出第一版后，就搭载在当时最新的Netscape Web浏览器中发布了。因为开发过程匆忙，一些好的，甚至是无与伦比的语言特性被加入到JavaScript，但，也有一些欠考虑的部分被加入其中。

本书没有说明JavaScript好坏之分的章节，如果你想了解这个话题，推荐阅读Douglas Crockford的著作"JavaScript, The Good Parts"。

尽管JavaScript存在着一些缺陷，它依然快速地，并有点出人意料地成为了Web浏览器默认的脚本语言。当时的JavaScript主要用于操作HTML文档，人们也因此创造了第一个动态的Web应用程序。

在1998年晚些时候，万维网联盟(W3C)发表了文档对象模型(DOM)标准，提供了统一的API来操作HTML文档。JavaScript本身存在的一些怪癖，以及深受开发者嗤之以鼻的DOM API，对它造成了非常不好的负面影响。当然还有一部分原因是浏览器厂商之间兼容性的问题（有时甚至同一厂商内的不同版本也会存在兼容性问题）。

尽管一些开发者社区对JavaScript的不满日益积累，但它却被广泛地使用。无论好坏，今天JavaScript已经成为这个星球上部署最广泛的编程语言，并且至今还在不断增长。

你要是学习了这门语言中为人称道的部分，像原型继承、函数闭包等等，并且学会怎么规避错误，JavaScript将会是你工作中非常愉快的伙伴。

函数声明 (Function declaration styles)

在JavaScript中声明一个函数的方式有很多，最简单的方式就是匿名函数：

```
function() {
    console.log('hello');
}
```

我们声明了一个函数，由于没有调用，它几乎没什么用处。不过在上面的例子，我们也没法去调用我们声明的匿名函数。

我们可以立即执行一个匿名函数：

```
(function() {
   console.log('hello');
})();
```

在定义了匿名函数后，我们立刻就执行了函数内的代码。注意这里我们将整个函数的声明包括在了第一对圆括号内。

下面看一看命名函数的声明方式：

```
function myFunction() {
    console.log('hello');
}
```

这里我们声明了一个命名函数：myFunction，这个函数只在函数声明的作用域内可用。

```
myFunction();
```

让我们加入一些内部作用域：

```
function myFunction() {
   console.log('hello');
}
```

```
(function() {
    myFunction();
})();
```

JavaScript将函数作为一等"公民"对象，即意味着我们可以把函数赋值给任何一个变量：

```
var myFunc = function() {
 console.log('hello');
```