什么是Node.js？

系统全面的Java语言教程，Java SE、Java EE、Android开发，2个月入门到精通

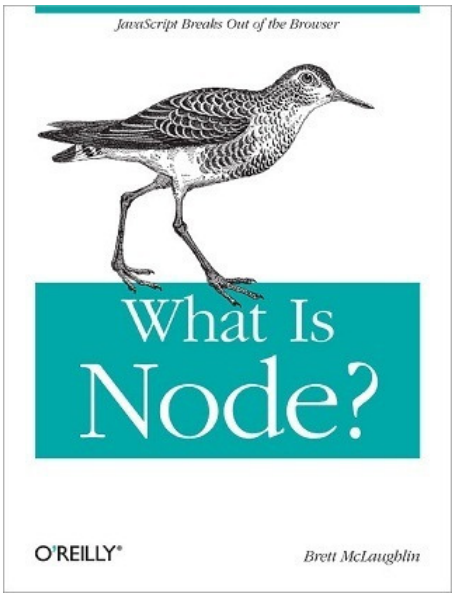
1

摘要：前不久Oreilly出了一本小册子“What is Node?”，扼要的讲解了Node的身世和所适用的场景，作者文笔轻松流畅、内容充实，是非常难得的学习资料。翻译出来，和大家分享~

译文全文：http://jayli.github.com/whatisnode/index.html

作者：Brett McLaughlin ，原文：What is Node?

1



Node不是万能药！但的确能解决一些关键问题。

学习Node不是一件轻松事儿，但你所收到的回报是对得起你的付出的。因为当下Web应用开发中的诸多难题唯有JavaScript才能解决。

目录

“你够酷吗？来用我吧！” Node.js 为最新潮的编程语言提供了一系列很酷的API和工具箱，它可以直接应用于传统的Rails、Ajax、Hadoop、甚至可以某种程度上用于iPhone开发和HTML5。如果你参加过一些大型技术会议，你总是会听到一些关于Node.js的主题演讲，尽管这些话题对普通的开发者来说依然有些难以企及。

你可能已经听说Node.js（有时我们将其简称为“Node”）是一个服务器端的解决方案，它可以运行JavaScript，并可以作为Web服务来处理HTTP请求。如果这些东东还不至于让你晕头转向的话，转眼间关于端口、sockets和线程的讨论就又成了当下最热门的话题，你会觉得这些东西让你眼花缭乱。这些内容真的属于JavaScript的范畴吗？为什么世界上那么多人宁愿将JavaScript脱离浏览器而运行，更不用说将JavaScript运行于服务器端了？

好消息是，你所听到的（所想到的）关于Node的一切都是正确的。Node的的确确是属于网络编程的范畴，用以处理服务器端的请求和响应。坏消息是和之前的Rails、Ajax和Hadoop一样，真正实用的技术资料实在太少。等到基于Node的“优秀的”框架成熟之后，技术资料一定会跟得上的，但何必要等到技术书籍和教程都出来之后再去尝试使用Node呢？现在就使用Node，说不定会给你的代码带来意想不到的改观，甚至让你的程序变得更易实现。

专家门的警告！

和大多数技术一样，Node也是新瓶装旧酒：它看起来不透明而且很怪异，但独受小开发团队的青睐。如果你没有接触过Node，则需要学习一些很容易上手的服务器端脚本。你需要花时间来搞清楚Node，因为即便是运行于服务器端的JavaScript，它和客户端JavaScript也极为不同。实际情况是，你不得不自己给自己洗脑，以便重新学习理解围绕JavaScript的事件处理机制、异步IO和一些网络基础知识。

不幸的是，这意味着如果你已经用Node作开发超过两年时间的话，你会觉得这篇文章内容很单调乏而且过于简单。你会开始寻找新的“刺激”，比如将Node运行于客户端，或者开始尝试事件I/O、反射器模式和npm。你会发现Node的世界是如此有趣，甚至很多Node高级技术具有某种史诗般的美感，而这些东西对于初学者来说依然是难于企及的。因此，或许你应该将你掌握的知识分享给你的同伴，尤其是对于那些不了解Node的同学，当他们开始对Node感兴趣时，给他们分享传授Node高级技术。

Node：几个小例子

首先，你应当意识到Node是用于运行独立的JavaScript程序的，而不是运行于浏览器中的某个HTML片段里。它是存放在文件系统中的真实存在的文件，由Node程序执行，以一种守护进程的模式运行，同时打开对某些端口的监听。

跳过 hello world

最经典的例子当然是“Hello World ”，在Node官网（http://nodejs.org/docs/latest ）上有源码。几乎每个人都是从Hello World开始接触Node的。现在让我们跳过这个最简单的例子，来看一些更有趣的例子：实现一个可以从服务器发送文件到客户端的程序（而不仅仅是发送一段文本到客户端）。

```
var sys = require("sys"),
http = require("http"),
url = require("url"),
path = require("path"),
fs = require("fs");
http.createServer(function(request, response) {
var uri = url.parse(request.url).pathname;
var filename = path.join(process.cwd(), uri);
path.exists(filename, function(exists) {
if(!exists) {
response.writeHead(404, {"Content-Type": "text/plain"});
response.end("404 Not Found\n");
return;
}
fs.readFile(filename, "binary", function(err, file) {
if(err) {
response.writeHead(500, {"Content-Type": "text/plain"});
response.end(err +"\n");
return;
}
response.writeHead(200);
response.end(file, "binary");
});
});
}).listen(8080);
console.log("Server running at http://localhost:8080/");
```

感谢Mike Amundsen，他给出了这段代码的相似的实现。这个例子是由Devon Govett在Nettuts+上提交的一段代码，尽管已经根据新版本的Node作了更新，但Devon的整个帖子是一个非常好的入门学习教材，对于初学者来说更是如此。

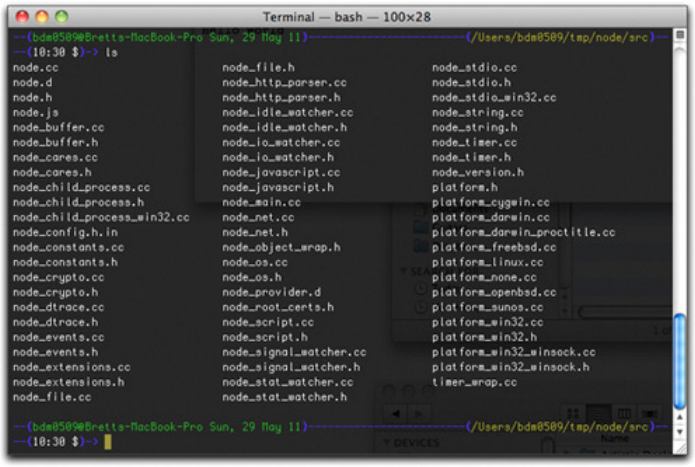
如果你是一个新手，你可以将上述代码保存到一个文本文件中，命名为NodeFileServer.js。在运行之前你需要一个Node运行环境，最新的Node版本可以从官网下载这个文件或者从github上将源码取下来。你需要编译源码，如果你没有用过Unix、对make和configure不甚熟悉，则需要查阅在线编译手册来寻求帮助。

Node不是JavaScript，Node可以运行JavaScript

刚刚你将NodeFileServer.js存成了某个文件，别担心，我们等下会回过头来运行它的。现在，让我们移步到现实当中来，在Unix中执行典型的配置和编译命令：

```
./configure
make
make install
```

这让我们确信一个事实：Node不是JavaScript，Node是一个可以运行JavaScript的程序，但Node绝对不是JavaScript。实际上，Node是基于C写的程序。可以通过ls来查看Node/src目录中的文件，可以看到Node的源码：



大多数人会以为，JavaScript是一门糟糕的语言，更不用说用它来实现服务器端的功能了，其实你只对了一半。不错，对于操作系统级别的Socket和网络编程来说，JavaScript可能并不能胜任。但Node并不是JavaScript实现的，它是基于C实现的。C语言是可以完美的胜任任意量级的网络编程的。而JavaScript则完全有能力将指令传递给C程序，然后由C程序来操控操作系统“地下城”。实际上，和C语言相比，JavaScript更容易被开发者们接触到，这是值得引起注意的地方，如果

你想用Node进行一些严肃的编程的话，这个原因会被一再提及。

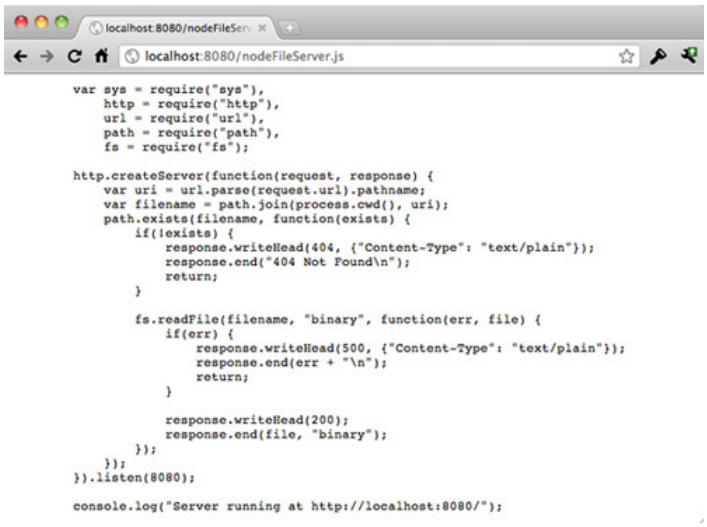
Node的基本用法进一步反映出了Node是如何和JavaScript一起工作的，Node不是JavaScript。你可以通过命令行来运行它：

```
— (bdm0509@Bretts-MacBook-Pro Sun,29 May 11)
— — — — — — — — — — (/Users/bdm0509/tmp/Node/src) —
— (09:09 $)-> export PATH=$HOME/local/Node/bin:$PATH
— (bdm0509@Bretts-MacBook-Pro Sun,29 May 11)
— — — — — — — — — — (/Users/bdm0509/tmp/Node/src) —
— (09:09 $)-> cd ~/examples
— (bdm0509@Bretts-MacBook-Pro Sun,29 May 11)
— — — — — — — — — — — — (/Users/bdm0509/examples) —
— (09:09 $)-> Node NodeFileServer.js
Server running at http://127.0.0.1:1337/
```

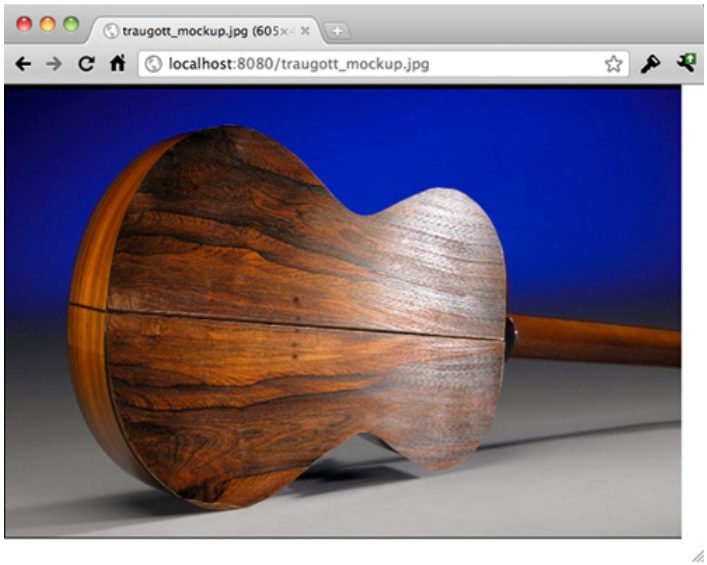
这里也不会对此做过多介绍，只要知道Node可以运行JavaScript，这就足够了。而且你只需学习JavaScript这一门编程语言即可，不用担心自己不懂C语言。记住这是最最重要的一点，不必了解C也可写出Node可运行的程序。

和Node服务器的交互

刚才我们在Node上运行了NodeFileServer.js。这时你可以访问你本机的1337端口，可以看到正常的输出。



没错，输出结果不足为奇。但应当意识到我们只用短短20行代码就实现了一个文件服务器。输出结果是你刚刚保存的脚本源文件的文本，并没有以二进制的形式输出。这个文件服务器可以输出它上面的任何文件。如果在同样目录下放入一张图片，在URL后缀中写上图片文件名，就像这样：http://localhost:8080/my_image.png。



Node也可以展示出二进制的图片文件。当你回头再看这段短小的程序时，一定觉得这太不可思议了。用JavaScript轻易就能写出一个你想要的服务程序难道不让人感到惬意吗？不仅如此，假设你想写一个可以处理多个请求的服务（这是一个提示，同时打开四个五个甚至十个浏览器访问服务器），这也是很容易做到的。Node让人着迷的地方在于，你完全可以用很简单而且很不起眼的JavaScript程序来完成你想要的这些结果。

快速入门手册

围绕Node的话题总是会比纯粹运行在服务器端的代码更值得花点时间来讨论。不管怎样，我们还是从一段代码开始我们的话题，概览一下NodeFileServer.js文件，观察代码：

```
var http = require('http');
```

```
http.createServer(function (req, res) {
res.writeHead(200, {'Content-Type': 'text/plain'});
res.end('Hello World\n');
}).listen(1337, "127.0.0.1");
console.log('Server running at http://127.0.0.1:1337/');
```

首先调用了函数require()，require()是程序员最常用的函数之一。实际上，在CommonJS规范中也有提到这个函数，在讨论到关于JavaScript模块概念的时候有提及，此外，Davd Flanagan在2009年的一个很酷的实现中也有提到。换句话说，require()对于你来说可能是个新鲜事物，但它不是Node随意添加的一个函数，他是使用JavaScript进行模块化编程的核心概念，Node将这一特性发挥的淋漓尽致。

接下来，http变量用以创建一个服务器。这个服务使用一个回调函数来处理当产生一个连接时的动作。这里的回调函数并未对请求作过多修饰，仅仅以text/plain格式输出一个字符串“Hello World”作为请求响应。这个逻辑非常简单。

实际上，这里展示了使用Node的标准模式：

定义交互类型，并获得一个用以处理这个交互的变量（通过require()）。
创建一个新的服务（通过createServer()）。
给服务绑定一个回调，用以处理请求。包括处理请求的函数应当包括一个请求...，以及一个响应
通知服务器启动服务，这里需要指定IP和端口（通过listen）。

解释器之惑

尽管通过这种方法可以使用JavaScript轻易的实现一个服务（不管运行代码的虚机实际上跑的是C程序还是其他什么程序），这种做法回避了一个问题：你需要使用JavaScript写出一个服务器吗？为了找到这个问题的答案，我们来考虑一个非常典型的场景。

JSON的处理

这是一种非常典型的web应用，前台使用HTML和CSS，JavaScript用来作数据验证，并和后台进行数据交互。由于你处于web交互的最顶端，你使用Ajax提交数据到后台并从后台获取数据，而不是单单依靠表单提交来实现。如果你是这样做的话，那么你同样会非常喜欢使用JSON的。JSON是如今最流行的传输数据的格式。

因此，这个Ajax也可以比作“把在线拍卖网站里的某些吉他的信息发给我”。这个请求通过网络到达一个运行PHP程序的服务器。PHP服务器不得不给JavaScript返回很多信息，而且这些信息必须以某种形式的数据包发给客户端，而且这个数据包是可以被JavaScript解析的。因此数据可以打包成数组，然后转换为JSON，就像这样：

```
$itemGuitar = array(
'id'=>'itemGuitar',
'description'=>'Pete Townshend once played this guitar while his own axe ' .
was in the shop having bits of drumkit removed from it.',
'price' => 5695.99,
'urls' => array('http://www.thewho.com',
'http://en.wikipedia.com/wiki/Pete_Townshend')
);

$output = json_encode($itemGuitar);
print($output);
```

回到客户端，JavaScript得到这个返回的数据包，由于经过转换，数据编程了JSON格式。就像这样：

```
{
"id": "itemGuitar",
"description": "Pete Townshend once played this guitar...",
"price": 5695.99,
"urls": ["http://www.thewho.com", "http://en.wikipedia.com/wiki/Pete_Townshend"]
}
```

这种转换是标准的，转换前后也是相互等价的。接下来，就可以将这个字符串转换为JavaScript对象，可以调用eval()，就像这样：

```
var itemDetails = eval('('+ jsonDataString +')');
```

计算结果是一个普通的JavaScript对象，这个对象的属性和JSON数组的数据结构保持一致。当然，由于jsonDataString通常是由服务器返回的，通常需要这样来解析返回结果：

```
var itemDetails = eval('('+ request.responseText +')');
```

这就是最最典型的JSON处理，但存在一个非常严重的问题。

对实体代码微妙的破坏性

（译注：这个小标题着实让人费解，作者这里拐弯抹角的解释了Node的一个好处，就是前端和后端都采用同样的语言JavaScript，在作JSON解析时是无障碍的，而当前端使用JavaScript作JSON编码，后台用PHP作JSON解码时，多少会因为多种语言的JSON解析的实现不同而带来一些兼容性问题）

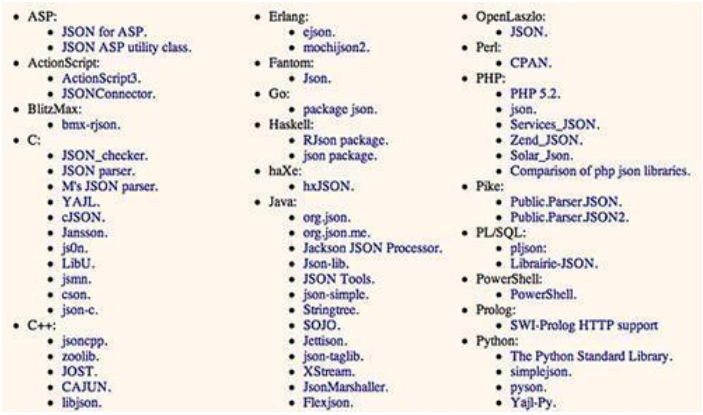
首先，这类代码的一个主要问题是，它对解释器的依赖比较严重。在上个例子中，解释器就是指内置的JSON解析器或者实现解析JSON的代码，这实际上依赖了两样东西：和eval()解析响应文本的操作一样的基于Java的JSON解析器，以及基于PHP的JSON解析器。在PHP5.2.0中已经包含了JSON解析器，但却是以外部依赖的形式给出的，并不是内置于PHP的内核中。

但这并不是大肆宣扬解释器的种种。毕竟解释器本身还存在很多问题，比如将“I”解析成了“i”，数组中的元素1解释成了2。当然，在JSON工具正式发布之前会有大量的测试，以保证在各种复杂场景中都不会出现错误，包括在客户端的解析结果和在服务器端的解析结果完全一致。无论如何，这都需要大量的测试才行。

不管怎样，JSON依然存在很多实际的问题。

基于某种语言（基于JavaScript或者PHP）的JSON解析器选择是一个很大的问题。换句话说，问题不是在于“翻译”（translation）而在于“翻译器”（translator）（译注：作者的意思是说JSON本身的规则没有问题，反倒是各种语言的JSON实现的质量参差不齐，甚至有很多bug）。当一个语言的版本比较稳定时，基于这门语言的JSON解析器的运用和推广会比较快。结果是，JSON解析器变的越来越强大，以至于可以解析任意复杂的数据结构，即便这么复杂的数据结构根本不会实际用到。反之，每次迭代中（每次计算迭代的路径和数据类型的组合），也很有可能出现JSON解释器无法解析的数据结构（或者很深的JSON路径）的情况。

下图就是可选的JSON解释器



- ASP:
  - JSON for ASP.
  - JSON ASP utility class.
- ActionScript:
  - ActionScript3.
  - JSONConnector.
- BlitzMax:
  - bmx-rjson.
- C:
  - JSON_checker.
  - JSON parser.
  - M's JSON parser.
  - YAJL.
  - cJSON.
  - Jansson.
  - js0n.
  - LibU.
  - jsmn.
  - cson.
  - json-c.
- C++:
  - jsoncpp.
  - zoolib.
  - JOST.
  - CAJUN.
  - libjson.
- Erlang:
  - ejson.
  - mochijson2.
- Fantom:
  - Json.
- Go:
  - package json.
- Haskell:
  - RJson package.
  - json package.
- haXe:
  - hxJSON.
- Java:
  - org.json.
  - org.json.me.
  - Jackson JSON Processor.
  - Json-lib.
  - JSON Tools.
  - json-simple.
  - Stringtree.
  - SOJO.
  - Jettison.
  - json-taglib.
  - XStream.
  - JsonMarshaller.
  - Flexjson.
- OpenLaszlo:
  - JSON.
- Perl:
  - CPAN.
- PHP:
  - PHP 5.2.
  - json.
  - Services_JSON.
  - Zend_JSON.
  - Solar_Json.
  - Comparison of php json libraries.
- Pike:
  - Public.Parser.JSON.
  - Public.Parser.JSON2.
- PL/SQL:
  - pljson:
  - Librairie-JSON.
- PowerShell:
  - PowerShell.
- Prolog:
  - SWI-Prolog HTTP support
- Python:
  - The Python Standard Library.
  - simplejson.
  - pyson.
  - Yajl-Py.

这并不是说JSON本身很糟糕，实际上，我们认为JSON的流行正是得益于其在新领域中的应用（译注：作者的言外之意是，在新领域中的初次JSON实现往往伴随很多问题）。对于新的领域，我们不禁要问：“这个新东东支持JSON吗？”因此，JSON需要不断进化，需要不断的测试，不断的兼容新的平台。而作为程序员的，可能需要重新组织你的数据结构，或者等待新的版本出现以满足你的需求，或者干脆直接hack JSON。而这些正是我们所说的编程资源的浪费。

假设你可以自己动手丰衣足食实现一个解释器，即便这样，你也没有通过“抄近道”拣到便宜，而是用JavaScript重复造轮子而已。

而Node则规避了此类问题，刚刚你读到的文字——关于内嵌JSON的PHP5.2.0、关于将对象转换为数组、关于采用新的结构组织数据的方式、关于JSON中新特性的实现——这一切扰人的问题在Node中都将不复存在，因为前端通过JavaScript作JSON编码，后台使用JavaScript作JSON解码，永远不会出问题。

JavaScript中eval()的潜在隐患

正如我们不用将Node当作一门新的语言来对待一样，在Node中通过eval()来执行一段代码也和JavaScript中的eval()一样（不被推荐）。众所周知eval()是非常危险的。eval()用以执行一段文本表示的代码逻辑，可以理解为在文本框中“直接敲入SQL代码来执行查询”，这是不安全的，这实际上是恶意SQL注入。当每次eval()执行一段字符串的时候，（美国）中西部的一只小狗都会瑟瑟发抖，东部海滩上的某位母亲的脚趾会被刺伤并受到诅咒。eval()非常危险。网上有很多关于此的资料，这里不再赘述。可以用google查询“eval JavaScript evil”或者“eval JavaScript injection”获取更多信息。

当然，如果没有任何其他上下文的约束，在Node中也是允许使用eval()的，因此eval()的隐患在Node依然存在。毕竟Node的目的并不是完全解决eval()的问题。Node被称之为基于事件的JavaScript或基于事件的I/O，这里所说的“基于事件”是Node中非常重要的概念。但要彻底理解什么是基于事件，以及为什么基于事件能让你规避eval()的危险，则需要理解JSON在应用之中是如何工作的，此外还要搞清楚适应于web应用典型架构的特有数据结构。

基于事件的Web应用

传统的Web表单提交就是典型的基于事件的模式。换句话说，在Web表单里输入了很多数据（用户输入文本框，点选复选框，从列表中选中某些项等等），之后这些数据提交给服务器。这个场景中实际是一个单一的程序事件：使用POST方式将表单数据提交。这也是基于Ajax的Web应用的工作原理。

一次性发送大量数据

对于Ajax来说，是可以和基于事件编程扯上一点关系。客户端和服务器端之间有些交互可以认为是基于事件的。典型的场景是输入一个省市代码，发送请求到服务器获得城市和省的名称。这里通过XmlHttpRequest的Ajax并不需要将很多数据一次性扔给服务器。但这并不能改变大部分web应用都是基于页面刷新这种模式的现状。Ajax已经更广泛的用于很多有意思的视觉相关的交互，快速的作表单验证，无刷新提交数据，这样就可以避免重新载入页面。因此，尽管并未通过提交表单来发起一个真正的POST请求，通过Ajax可以模拟POST表单提交。

坦率的讲，这种传统的Ajax交互方式也阻碍了Ajax程序员的创新。每次发送一个请求时（不管请求的数据多么小），都会在网络里走一个来回。服务器必须针对这个请求作出响应，通常是开辟一个新的进程。因此，如果你真正置身于一个事件模型的环境中做开发，你可能需要通过发起10到15个单独的小请求来保持你的页面和服务器之间的联系，服务器也会为之创建10到15个线程（可能更少，这取决于服务器处理新请求时分配线程池的策略），当这个数量乘以1000或者10000或者100000时（译注：每个页面需要10个请求，那么越多用户访问这个页面，所发起的请求个数就会越来越多），就会出现内存溢出、逻辑交错带来的冲突、网络瘫痪、系统崩溃这些问题。

结果是，在大多数场景中，Web应用需要保持对事件的最小依赖。有一个折衷方案，就是服务器端程序的响应返回的不是一个微小的数据片段，而是带有更多冗余数据结构的数据包，通常是JSON数据，这时就又遇到了eval()的问题。问题当然出在eval()身上，但这也和Web本身和服务器线程控制、包括页面和服务器之间的HTTP请求和响应策略（至少在这个场景下）有密不可分的关系。

或许有些人对上文提到的问题不以为然，因为你知道有很多方法来规避直接eval()带来的问题，你会使用诸如JSON.parse()来代替eval()。同样有很多令人信服的论据鼓励我们小心的使用eval()。这些东东都是值得进一步讨论的。但不管怎样，看一看eval()带来了太多类似栈溢出（Stack Overflow）这类的问题吧，你会发现大部分程序员并未正确或者安全的使用eval()。这着实是一个问题。因为太多菜鸟程序员似乎根本没有意识到eval()的问题有多严重。

不断的发送少量的数据

Node带来了架构应用的新思路，我们可以基于Node采用事件模型来架构Web应用，或者说“小型的”事件模型。换句话说，你应当基于大量的事件发送大量的请求，每个请求的数据包都很小，或者根据需要从后台抓取少量数据，而不是发送很少的请求，每次请求都带有大量的数据。在很多场景中，大多数情况下你需要唤醒GUI程序（Java Swing程序员的GUI知识储备可以派上用场了）。因此，当用户输入姓氏和名字后，移步到下一个输入框，这时就已经发起了一个请求来验证输入的用户名是否已经存在。省市代码、地址和电话号码的验证也是同理。页面上每发生一个事件，都会产生一个请求和响应。

这有什么不同吗？为什么Node可以做到，并规避了已有的线程问题？其实Node并没有这么神秘，Node官网充分解释了其哲学：

Node的目标是提供一种构建可伸缩的网络应用的方案，在hello world例子中，服务器可以同时处理很多客户端连接。Node和操作系统有一种约定，如果创建了新的链接，操作系统就将通知Node，然后进入休眠。如果有人创建了新的链接，那么它（Node）执行一个回调，每一个链接只占用了非常小的（内存）堆栈开销。

Node是无阻塞的，不会出现同源竞争线程的情况（Node非常乐于处理即时的请求，发生了什么事情，那就让他发生吧），新请求到达服务器时，不需要为这个请求单独作什么事情。Node仅仅是悠闲的坐在那里等待（请求的发生），有请求就处理请求。用非常简单的代码就可以实现，而不用花费程序员宝贵的精力去实现一整套服务器端逻辑。

没错，混乱不可避免

值得一提的是，非阻塞系统带来的问题也会出现在这种编程模式中：一个进程（非线程）等待一个数据存储操作，这时产生了另外一个抓取与之无关的数据的操作，这个意外的操作会对现有的等待造成影响（译注：作者的意思是说多个操作同时发生或者没有按照预定顺序发生时，会产生混乱，也就是说，操作本身并不是原子性的）。但要注意，大多数基于事件的web编程模式都是“只读的”！你大概也没有遇到过通过“微请求”来修改数据的情况，或者说非常罕见。相反，通过这种请求来验证数据合法性、查询数据的情形则非常常见。这种情况下，最好直接根据请求作响应。数据库本身会作加锁操作，一般来讲，一个优秀的数据库完全可以高效的做到数据操作的加锁解锁，而不用服务器端的程序代码去多做什么。而Node又比操作系统处理线程的保持和释放更加高效，使得服务器不必单独为“web响应”开辟一个进程。

此外，Node也计划实现“进程分支”（process forking），HTML5 Web Workers API为更复杂的进程控制提供了引擎（规范）支持。同样，如果你采用基于事件的模型来架构web应用，你的程序可能至少有100多个场景需要线程的支持。最终你会发现，你的编程思路和思考问题的方式发生了改变，你的注意力将放在服务器端处理请求的逻辑上，而不必在乎Node如何工作。

Node的用武之地

这里我们讨论另外一种web开发模式，不管是不是采用了Node、或者是不是采用了基于事件的编程模式，这都无关紧要，因为这种模式实在太重要了。简言之：对症下药！概括讲就是，针对不同的问题采取不同的解决方案，而不管这种解决方案是否解决其他问题。

思维定势

不止在web设计领域，在所有编程之中都存在某种思维定势。可以这么描述这种思维定势：你学到的、掌握的越多，你能解决的问题就越多，你所掌握的技能的应用场景也就越多。这看起来理所当然，除非你在技术上钻研的更深。没错，学习新的语言和新的工具并广泛使用它总不是坏事。但往往会进入一个误区，就是，因为你了解它，所以你使用它，而不是因为你所掌握的技能和工具是“最适合”你的业务的。

我们来看一下Ajax，关于Ajax已经有太多太多的讨论了。我们知道，Ajax为无刷新的快速查询请求提供了可靠的解决方案。而如今因为Ajax的滥用以至于过分替代了传统的表单提交。我们遇到一个新技术、学习它、掌握它、应用它，然后“滥用它”。毕竟很多业务场景仅仅需要传统的表单提交，而不需要Ajax。说起来简单，实际上还有成千上万的滥用Ajax的案例场景，仅仅因为某个应用的开发工程师对Ajax的盲目尊崇。

同样的，Node也面临这样一个问题。当你初识Node发现它的种种好处，就想到处使用它。就会一股脑的将PHP或Perl程序换成Node。结果呢？糟透了。其实你已经害上了强迫症，总是想将Node用于有违其设计初衷的场景中：使用JavaScript提交大量数据给Node，或者通过Node返回给JavaScript大量的JSON数据，交给前端去作eval()，或者干脆使用Node作一个文件服务器用以返回HTML页面或做HTTP重定向。

但这些场景均不是Node所擅长的。Node更擅长处理体积小的请求以及基于事件的I/O，使用Node解决客户端和服务器之间的快速沟通，使用表单提交将大量的数据发送给服务器，使用PHP和Perl来处理重型数据库操作以及动态HTML页面的生成。使用Node运行于服务器端来处理体积不大的请求。不管是采用Rails还是Spring以及各式各样的服务端容器，只要按需索取即可。一定要明白你需要解决的问题是什么，基于此采取最佳解决方案，而不是基于你当下所掌握的技能来解决遇到的问题。

Node的简单的初衷

还有最后一点需要注意，当你越来越深入你的编程时，你会发现你不必每个工具、API和所使用的框架都达到精通。将刀用在刀刃上，不要将锤子当成钻头来使用。了解每个工具所适用的场景和能解决的问题，然后找到这个工具的最适合的应用场景。如果你想变成超人式的通才（程序员往往什么都想知道），你离“专家”也就越来越远，所谓专家，就是指在一两个方面达到非常精通。当然，每个老板都希望能找到超人式的通才，但这种人往往可遇不可求。

学习Node可能会有些吃力，但是非常值得的。为什么？因为你正在寻求基于JavaScript的web应用的解决方案。这意味着你已有的JavaScript编程技能不会丢掉，当你需要使用PHP或者Perl时，你必须重新学习一门新的语言，而Node不必如此大动干戈。学习新语言带来的问题比学习他们带来的好处要大的多。

学习Node所面临的挑战是，你需要更加活跃思维，将程序拆成低耦合的小片段，然后像组装数组一样的组装他们。但Node和基于事件的I/O并不能解决所有问题，但确定的是，很多关键问题，只能依靠Node来解决。

参考文献

Node:Up and Running
The secrets of Node’s success
Why a JavaScript hater thinks everyone needs to learn JavaScript in the next year
JavaScript spread to the edges and became permanent in the process
What is Node.js and what does it do?

JavaScript

提问时间 **2011-10-13 10:31**

ajax
15 1 3
答案被采用率:
0%