书《深入浅出Node.js》阅读笔记

第1章：Node简介

1.2.1 为什么是JavaScript

Ryan Dahl是一名资深的C/C++程序员，在创造出Node之前，他的主要工作都是围绕高性能Web服务器进行的。经历过一些尝试和失败之后，他找到了设计高性能，Web服务器的几个要点：事件驱动、非阻塞I/O。

所以Ryan Dahl最初的目标是写一个基于事件驱动、非阻塞I/O的Web服务器，以达到更高的性能，提供Apache等服务器之外的选择。他提到，大多数人不设计一种更简单和更有效率的程序的主要原因是他们用到了阻塞I/O的库。写作Node的时候，Ryan Dahl曾经评估过C、Lua、Haskell、Ruby等语言作为备选实现，结论为：C的开发门槛高，可以预见不会有太多的开发者能将它用于日常的业务开发，所以舍弃它；Ryan Dahl觉得自己还不足够玩转Haskell，所以舍弃它；Lua自身已经含有很多阻塞I/O库，为其构建非阻塞I/O库也不能改变人们继续使用阻塞I/O库的习惯，所以也舍弃它；而Ruby的虚拟机由于性能不好而落选。

相比之下，JavaScript比C的开发门槛要低，比Lua的历史包袱要少。尽管服务器端JavaScript存在已经很多年了，但是后端部分一直没有市场，可以说历史包袱为零，为其导入非阻塞I/O库没有额外阻力。另外，JavaScript在浏览器中有广泛的事件驱动方面的应用，暗合Ryan Dahl喜好基于事件驱动的需求。当时，第二次浏览器大战也渐渐分出高下，Chrome浏览器的JavaScript引擎V8摘得性能第一的桂冠，而且其基于新BSD许可证发布，自然受到Ryan Dahl的欢迎。考虑到高性能、符合事件驱动、没有历史包袱这 3 个主要原因，JavaScript成为了Node的实现语言。

1.2.2 为什么叫Node

起初，Ryan Dahl称他的项目为web.js，就是一个Web服务器，但是项目的发展超过了他最初单纯开发一个Web服务器的想法，变成了构建网络应用的一个基础框架，这样可以在它的基础上构建更多的东西，诸如服务器、客户端、命令行工具等。Node发展为一个强制不共享任何资源的单线程、单进程系统，包含十分适宜网络的库，为构建大型分布式应用程序提供基础设施，其目标也是成为一个构建快速、可伸缩的网络应用平台。它自身非常简单，通过通信协议来组织许多Node，非常容易通过扩展来达成构建大型网络应用的目的。每一个Node进程都构成这个网络应用中的一个节点，这是它名字所含意义的真谛。

Chrome浏览器和Node的组件构成如图1-1所示。我们知道浏览器中除了V8作为JavaScript引擎外，还有一个WebKit布局引擎。 HTML5在发展过程中定义了更多更丰富的API。在实现上，浏览器提供了越来越多的功能暴露给JavaScript和HTML标签。这个愿景美好，但对于前端浏览器的发展现状而言，HTML5标准统一的过程是相对缓慢的。JavaScript作为一门图灵完备的语言，长久以来却限制在浏览器的沙箱中运行，它的能力取决于浏览器中间层提供的支持有多少。

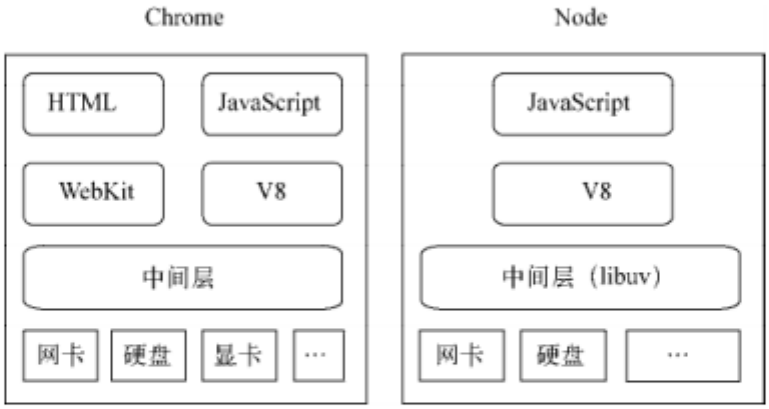


图1-1　Chrome浏览器和Node的组件构成

除了HTML、WebKit和显卡这些UI相关技术没有支持外，Node的结构与Chrome十分相似。它们都是基于事件驱动的异步架构，浏览器通过事件驱动来服务界面上的交互，Node通过事件驱动来服务I/O，这个细节将在第3章中详述。在Node中，JavaScript可以随心所欲地访问本地文件，可以搭建WebSocket服务器端，可以连接数据库，可以如Web Workers一样玩转多进程。如今，JavaScript可以运行在不同的地方，不再继续限制在浏览器中与CSS样式表、DOM树打交道。如果HTTP协议栈是水平面，Node就是浏览器在协议栈另一边的倒影。Node不处理UI，但用与浏览器相同的机制和原理运行。Node打破了过去JavaScript只能在浏览器中运行的局面。前后端编程环境统一，可以大大降低前后端转换所需要的上下文交换代价。

1.4.1 异步I/O

关于异步I/O，向前端工程师解释起来或许会容易一些，因为发起Ajax调用对于前端工程师而言是再熟悉不过的场景了。下面的代码用于发起一个Ajax请求：

```
$.post('/url', {title: '深入浅出Node.js'}, function (data) {
  console.log('收到响应');
});
console.log('发送Ajax结束');
```

熟悉异步的用户必然知道，“收到响应”是在“发送Ajax结束”之后输出的。在调用$.post()后，后续代码是被立即执行的，而“收到响应”的执行时间是不被预期的。我们只知道它将在这个异步请求结束后执行，但并不知道具体的时间点。异步调用中对于结果值的捕获是符合“Don’t call me, I will call you”的原则的，这也是注重结果，不关心过程的一种表现。图1-2是一个经典的Ajax调用。

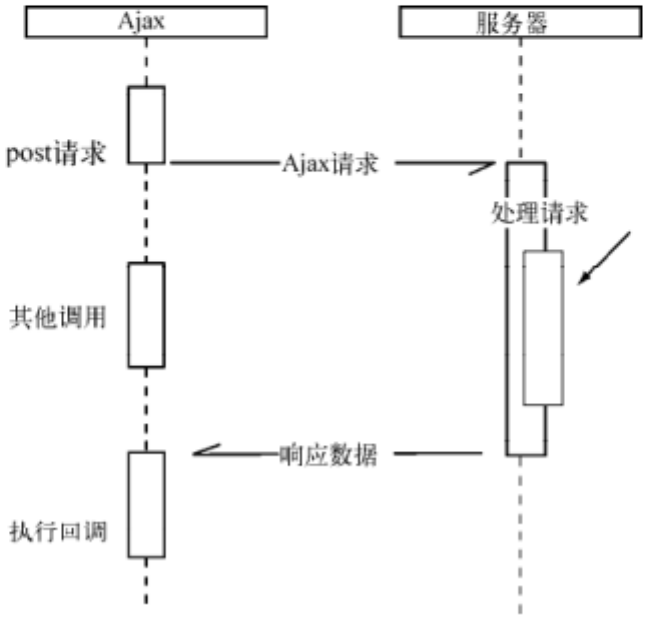
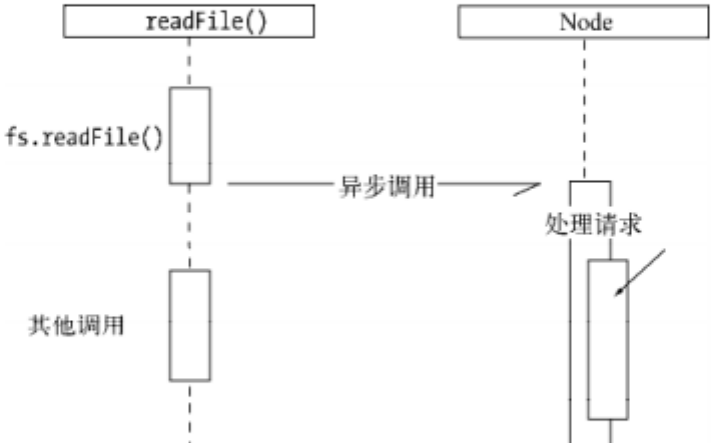


图1-2 经典的Ajax调用

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

fs.readFile('/path', function (err, file) {
  console.log('读取文件完成')
});
console.log('发起读取文件');
```

这里的“发起读取文件”是在“读取文件完成”之前输出的。同样，“读取文件完成”的执行也取决于读取文件的异步调用何时结束。图1-3是一个经典的异步调用。



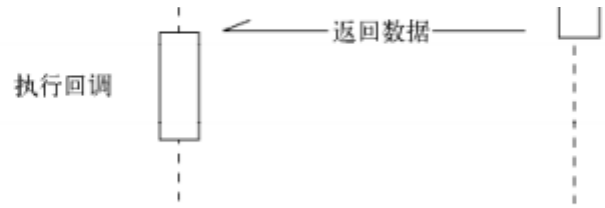


图1-3　经典的异步调用

在Node中，绝大多数的操作都以异步的方式进行调用。Ryan Dahl排除万难，在底层构建了

1.4.3　单线程

Node保持了JavaScript在浏览器中单线程的特点。而且在Node中，JavaScript与其余线程是无法共享任何状态的。单线程的最大好处是不用像多线程编程那样处处在意状态的同步问题，这里没有死锁的存在，也没有线程上下文交换所带来的性能上的开销。

同样，单线程也有它自身的弱点，这些弱点是学习Node的过程中必须要面对的。积极面对这些弱点，可以享受到Node带来的好处，也能避免潜在的问题，使其得以高效利用。单线程的弱点具体有以下3方面。

- 无法利用多核CPU。
- 错误会引起整个应用退出，应用的健壮性值得考验。
- 大量计算占用CPU导致无法继续调用异步I/O。

像浏览器中JavaScript与UI共用一个线程一样，JavaScript长时间执行会导致UI的渲染和响应被中断。在Node中，长时间的CPU占用也会导致后续的异步I/O发不出调用，已完成的异步I/O的回调函数也会得不到及时执行。

最早解决这种大计算量问题的方案是Google公司开发的Gears。它启用一个完全独立的进程，将需要计算的程序发送给这个进程，在结果得出后，通过事件将结果传递回来。这个模型将计算量分发到其他进程上，以此来降低运算造成阻塞的几率。后来，HTML5定制了Web Workers的标准，Google放弃了Gears，全力支持Web Workers。Web Workers能够创建工作线程来进行计算，以解决JavaScript大计算阻塞UI渲染的问题。工作线程为了不阻塞主线程，通过消息传递的方式来传递运行结果，这也使得工作线程不能访问到主线程中的UI。

Node采用了与Web Workers相同的思路来解决单线程中大计算量的问题：`child_process`。

子进程的出现，意味着Node可以从容地应对单线程在健壮性和无法利用多核CPU方面的问题。通过将计算分发到各个子进程，可以将大量计算分解掉，然后再通过进程之间的事件消息来传递结果，这可以很好地保持应用模型的简单和低依赖。通过Master-Worker的管理方式，也可以很好地管理各个工作进程，以达到更高的健壮性。

1.5.1　I/O密集型

在Node的推广过程中，无数次有人问起Node的应用场景是什么。如果将所有的脚本语言拿到一处来评判，那么从单线程的角度来说，Node处理I/O的能力是值得竖起拇指称赞的。通常，说Node擅长I/O密集型的应用场景基本上是没人反对的。Node面向网络且擅长并行I/O，能够有效地组织起更多的硬件资源，从而提供更多好的服务。

I/O密集的优势主要在于Node利用事件循环的处理能力，而不是启动每一个线程为每一个请求服务，资源占用极少。

我们将相同的斐波那契数列计算（F_0=0，F_1=1，F_n=$F_{(n-1)}$+$F_{(n-2)}$(n≥2)）分别用各种脚本语言写了算法实现，并进行了n=40的计算，以比较性能。这个测试主要偏重CPU栈操作，表1-1是其中一次运算耗时的排行。在这些脚本语言中（其中C和Go语言是静态语言，用于参考），Node是足够高效的，它优秀的运算能力主要来自V8的深度性能优化。

表1-1　计算斐波那契数列的耗时排行

语　言	用户态时间	排　名	版　本
C with -O2	0m0.202s	#0	i686-apple-darwin11-llvm-gcc-4.2 (GCC) 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.11.00)
Node（C++模块）	0m1.001s	#1	v0.8.8, gcc -O2
Java	0m1.305s	#2	Java(TM) SE Runtime Environment (build 1.6.0_35-b10-428-11M3811) Java HotSpot(TM) 64-Bit Server VM (build 20.10-b01-428, mixed mode)
Go	0m1.667s	#3	Go version go1.0.2
Scala	0m1.808s	#4	Scala code runner version 2.9.2 -- Copyright 2002-2011, LAMP/EPFL
LuaJIT	0m2.579s	#5	LuaJIT 2.0.0-beta10 -- Copyright (C) 2005-2012 Mike Pall.
Node	0m2.872s	#6	v0.8.8
Ruby 2.0.0-p0	0m27.777s	#7	ruby 2.0.0p0 (2013-02-24 revision 39474) [x86_64-darwin12.2.0]
pypy	0m30.010s	#8	Python 2.7.2 (341e1e3821ff, Jun 07 2012, 15:42:54) [PyPy 1.9.0 with GCC 4.2.1]
Ruby 1.9.x	0m37.404s	#9	ruby 1.9.3p194 (2012-04-20 revision 35410) [x86_64-darwin12.1.0]
Lua	0m40.709s	#10	Lua 5.1.4 Copyright (C) 1994-2008 Lua.org, PUC-Rio
Jython	0m53.699s	#11	Jython 2.5.2
PHP	1m17.728s	#12	PHP 5.4.6 (cli) (built: Sep 8 2012 23:49:53)
Python	1m17.979s	#13	Python 2.7.2
Perl	2m41.259s	#14	This is perl 5, version 12, subversion 4 (v5.12.4) built for darwin-thread-multi-2level
Ruby 1.8.x	3m35.135s	#15	ruby 1.8.7 (2012-02-08 patchlevel 358) [universal-darwin12.0]

这样的测试结果尽管不能完全反映出各个语言的性能优劣，但已经可以表明Node在性能上不俗的表现。从另一个角度来说，这可以表明CPU密集型应用其实并不可怕。CPU密集型应用给Node带来的挑战主要是：由于JavaScript单线程的原因，如果有长时间运行的计算（比如大循环），将会导致CPU时间片不能释放，使得后续I/O无法发起。但是适当调整和分解大型运算任务为多个小任务，使得运算能够适时释放，不阻塞I/O调用的发起，这样既可同时享受到并行异步I/O的好处，又能充分利用CPU。

关于CPU密集型应用，Node的异步I/O已经解决了在单线程上CPU与I/O之间阻塞无法重叠利用的问题，I/O阻塞造成的性能浪费远比CPU的影响小。对于长时间运行的计算，如果它的耗时超过普通阻塞I/O的耗时，那么应用场景就需要重新评估，因为这类计算比阻塞I/O还影响效率，甚至说就是一个纯计算的场景，根本没有I/O。此类应用场景或许应当采用多线程的方式进行计算。Node虽然没有提供多线程用于计算支持，但是还是有以下两个方式来充分利用CPU。

- Node可以通过编写C/C++扩展的方式更高效地利用CPU，将一些V8不能做到性能极致的地方通过C/C++来实现。由上面的测试结果可以看到，通过C/C++扩展的方式实现斐波那契数列计算，速度比Java还快。
- 如果单线程的Node不能满足需求，甚至用了C/C++扩展后还觉得不够，那么通过子进程的方式，将一部分Node进程当做常驻服务进程用于计算，然后利用进程间的消息来传递结果，将计算与I/O分离，这样还能充分利用多CPU。

CPU密集不可怕，如何合理调度是诀窍。

1.5.4 分布式应用

阿里巴巴的数据平台对Node的分布式应用算是一个典型的例子。分布式应用意味着对可伸缩性的要求非常高。数据平台通常要在一个数据库集群中去寻找需要的数据。阿里巴巴开发了中间层应用NodeFox、ITier，将数据库集群做了划分和映射，查询调用依旧是针对单张表进行SQL查询，中间层分解查询SQL，并行地去多台数据库中获取数据并合并。NodeFox能实现对多台MySQL数据库的查询，如同查询一台MySQL一样，而ITier更强大，查询多个数据库如同查询单个数据库一样，这里的多个数据库是指不同的数据库，如MySQL或其他的数据库。

这个案例其实也是高效利用并行I/O的例子。Node高效利用并行I/O的过程，也是高效使用数据库的过程。对于Node，这个行为只是一次普通的I/O。对于数据库而言，却是一次复杂的计算，所以也是进而充分压榨硬件资源的过程。

- **前后端编程语言环境统一**。这类倚重点的代表是雅虎。雅虎开放了Cocktail框架，利用自己深厚的前端沉淀，将YUI3这个前端框架的能力借助Node延伸到服务器端，使得使用者摆脱了日常工作中一边写JavaScript一边写PHP所带来的上下文交换负担。
- **Node带来的高性能I/O用于实时应用**。Voxer将Node应用在实时语音上。国内腾讯的朋友网将Node应用在长连接中，以提供实时功能，花瓣网、蘑菇街等公司通过socket.io实现实时通知的功能。
- **并行I/O使得使用者可以更高效地利用分布式环境**。阿里巴巴和eBay是这方面的典型。阿里巴巴的NodeFox和eBay的ql.io都是借用Node并行I/O的能力，更高效地使用已有的数据。
- **并行I/O，有效利用稳定接口提升Web渲染能力**。雪球财经和LinkedIn的移动版网站均是这种案例，撇弃同步等待式的顺序请求，大胆采用并行I/O，加速数据的获取进而提升Web的渲染速度。
- **云计算平台提供Node支持**。微软将Node引入Azure的开发中，阿里云、百度均纷纷在云服务器上提供Node应用托管服务，Joyent更是云计算中提供Node支持的代表。这类平台看重JavaScript带来的开发上的优势，以及低资源占用、高性能的特点。
- **游戏开发领域**。游戏领域对实时和并发有很高的要求，网易开源了pomelo实时框架，可以应用在游戏和高实时应用中。
- **工具类应用**。过去依赖Java或其他语言构建的前端工具类应用，纷纷被一些前端工程师用Node重写，用前端熟悉的语言为前端构建熟悉的工具。