深入浅出Node.js

田永强　著

崔康　审校

序

序言

这本迷你书的发布可以说是众望所归，笔者作为“深入浅出Node.js”专栏的发起人倍感欣慰。如今已是2013年盛夏，距离专栏的第一篇文章发布已经快两年时间，可是在每月InfoQ中文站的文章访问量Top10排行榜上，总有该专栏的某一篇或者几篇文章位列其中，这足以说明，“深入浅出Node.js”这个专题得到了广大读者的肯定和欢迎。

一门技术的发展历程类似于人生，总要经历孕育、诞生、成长、成熟、衰落等几个过程，作为技术人员，应该敏锐地观察和把握住所在领域的技术发展趋势，才不至于被技术的浪潮所淹没。专栏诞生之初，应该是Node.js技术的萌芽之后并在起步成长的关键阶段。InfoQ中文站适时地推出了这个专栏，从Node.js的基本概念、到模块管理、再到异步IO，按照循序渐进的思路向读者呈现了Node.js的魅力。作者永强兄（朴灵）是一位非常地道的Node.js实践者和布道师，他对前后端的Javascript领域都有着多年的一线经验，对Node.js技术的优缺点也有着独到的见解。虽然工作很忙，但永强还是积极地为InfoQ撰稿，从而成就了这个专栏和这本迷你书。在Qcon北京2013技术大会上，永强更是担任了“Node.js”专题的出品人，俨然已是国内Node.js社区的领袖。

时至今日，Node.js技术已经进入了青年时期，不论你是否感觉到，它已经慢慢渗入到了很多公司、软件、技术者的心里。如果说在专栏诞生之初，开发者只是观望和评估，那么现在越来越多的领域已经把Node.js应用于生产环境当中并取得了预期的效果。如果你在互联网领域或者企业级Web领域，如果是前端JS或者后端Web服务器开发者，那么Node.js现在是一门必须要了解、最好能够掌握的好技术，一方面可以直接应用于自己的产品和解决方案中，另一方面Node.js的设计思想也能够开阔架构师和开发者的思维方式。这本迷你书正是你了解Node.js技术的最佳入口之一，除此之外，InfoQ中文站还发布了许多Node.js相关的新闻、文章和讲座，读者可以边看书边阅读其他网上内容，相信会收获不少。

在这个信息爆炸的年代，我们每天要面对成千上万的数据输入，大脑已经在超负荷运转了。有多少次我们曾经计划开始读一本好书、计划开始锻炼身体、计划开始写点日记，有多少次我们回首发现一切如故。“不积跬步无以至千里”，下载这本迷你书，设定一个阅读计划，给自己一个丰富有用知识、改善自身习惯的机会吧：）

崔康

关于作者

田永强，新浪微博@朴灵，前端工程师，曾就职于SAP，现就职于淘宝，花名朴灵，致力于NodeJS和Mobile Web App方面的研发工作。双修前后端JavaScript，寄望将NodeJS引荐给更多的工程师。兴趣：读万卷书，行万里路。个人Github地 址：http://github.com/JacksonTian。

（一）什么是Node.js

> 【编者按】：Node.js从2009年诞生至今，已经发展了两年有余，其成长的速度有目共睹。从在github的访问量超过Rails，到去年底Node.jsS创始人Ryan Dalh加盟Joyent获得企业资助，再到今年发布Windows移植版本，Node.js的前景获得了技术社区的肯定。InfoQ一直在关注Node.js的发展，在今年的两次Qcon大会（北京站和杭州站）都有专门的讲座。为了更好地促进Node.js在国内的技术推广，我们决定开设“深入浅出Node.js”专栏，邀请来自Node.js领域的布道师、开发人员、技术专家来讲述Node.js的各方面内容，让读者对Node.js有更深入的了解，并且能够积极投入到新技术的讨论和实践中。

专栏的第一篇文章《什么是Node.js》尝试从各个角度来阐述Node.js的基本概念、发展历史、优势等，对该领域不

熟悉的开发人员可以通过本文了解Node.js的一些基础知识。

从名字说起

有关Node.js的技术报道越来越多，Node.js的写法也是五花八门，有写成NodeJS的，有写成Nodejs的，到底哪一种写法最标准呢，我们不妨遵循官方的说法。在Node.js的官方网站上，一直将其项目称之为”Node“或者”Node.js“，没有发现其他的说法，”Node“用的最多，考虑到Node这个单词的意思和用途太广泛，容易让开发人员误解，我们采用了第二种称呼——”Node.js“，js的后缀点出了Node项目的本意，其他的名称五花八门，没有确切的出处，我们不推荐使用。

Node.js不是JS应用、而是JS运行平台

看到Node.js这个名字，初学者可能会误以为这是一个Javascript应用，事实上，Node.js采用C++语言编写而成，是一个Javascript的运行环境。为什么采用C++语言呢？据Node.js创始人Ryan Dahl回忆，他最初希望采用Ruby来写Node.js，但是后来发现Ruby虚拟机的性能不能满足他的要求，后来他尝试采用V8引擎，所以选择了C++语言。既然不是Javascript应用，为何叫.js呢？因为Node.js是一个Javascript的运行环境。提到Javascript，大家首先想到的是日常使用的浏览器，现代浏览器包含了各种组件，包括渲染引擎、Javascript引擎等，其中Javascript引擎负责解释执行网页中的Javascript代码。作为Web前端最重要的语言之一，Javascript一直是前端工程师的专利。不过，Node.js是一个后端的Javascript运行环境（支持的系统包括*nux、Windows），这意味着你可以编写系统级或者服务器端的Javascript代码，交给Node.js来解释执行，简单的命令类似于：

```
#node helloworld.js
```

Node.js采用了Google Chrome浏览器的V8引擎，性能很好，同时还提供了很多系统级的API，如文件操作、网络编程等。浏览器端的Javascript代码在运行时会受到各种安全性的限制，对客户系统的操作有限。相比之下，Node.js则是一个全面的后台运行时，为Javascript提供了其他语言能够实现的许多功能。

Node.js采用事件驱动、异步编程，为网络服务而设计

事件驱动这个词并不陌生，在某些传统语言的网络编程中，我们会用到回调函数，比如当socket资源达到某种状态时，注册的回调函数就会执行。Node.js的设计思想中以事件驱动为核心，它提供的绝大多数API都是基于事件的、异步的风格。以Net模块为例，其中的net.Socket对象就有以下事件：connect、data、end、timeout、drain、error、close等，使用Node.js的开发人员需要根据自己的业务逻辑注册相应的回调函数。这些回调函数都是异步执行的，这意味着虽然在代码结构中，这些函数看似是依次注册的，但是它们并不依赖于自身出现的顺序，而是等待相应的事件触发。事件驱动、异步编程的设计（感兴趣的读者可以查阅笔者的另一篇文章《Node.js的异步编程风格》），重要的优势在于，充分利用了系统资源，执行代码无须阻塞等待某种操作完成，有限的资源可以用于其他的任务。此类设计非常适合于后端的网络服务编程，Node.js的目标也在于此。在服务器开发中，并发的请求处理是个大问题，阻塞式的函数会导致资源浪费和时间延迟。通过事件注册、异步函数，开发人员可以提高资源的利用率，性能也会改善。

从Node.js提供的支持模块中，我们可以看到包括文件操作在内的许多函数都是异步执行的，这和传统语言存在区别，而且为了方便服务器开发，Node.js的网络模块特别多，包括HTTP、DNS、NET、UDP、HTTPS、TLS等，开发人员可以在此基础上快速构建Web服务器。以简单的helloworld.js为例：

```
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(80, "127.0.0.1");
```

上面的代码搭建了一个简单的http服务器（运行示例部署在http://helloworld.cnodejs.net/中，读者可以访问），在本地监听80端口，对于任意的http请求，服务器都返回一个头部状态码为200、Content-Type'值为text/plain'的”Hello World“文字响应。从这个小例子中，我们可以看出几点：

- Node.js的网络编程比较便利，提供的模块（在这里是http）开放了容易上手的API接口，短短几行代码就可以构建服务器。
- 体现了事件驱动、异步编程，在createServer函数的参数中指定了一个回调函数（采用Javascript的匿名函数实现），当有http请求发送过来时，Node.js就会调用该回调函数来处理请求并响应。当然，这个例子相对简单，没有太多的事件注册，在以后的文章中读者会看到更多的实际例子。

Node.js的特点

下面我们来说说Node.js的特点。事件驱动、异步编程的特点刚才已经详细说过了，这里不再重复。

Node.js的性能不错。按照创始人Ryan Dahl的说法，性能是Node.js考虑的重要因素，选择C++和V8而不是Ruby或者其他的虚拟机也是基于性能的目的。Node.js在设计上也是比较大胆，它以**单进程、单线程**模式运行（很吃惊，对吧？这和Javascript的运行方式一致），事件驱动机制是Node.js通过内部单线程高效率地维护事件循环队列来实现的，没有多线程的资源占用和上下文切换，这意味着面对大规模的http请求，Node.js凭借事件驱动搞定一切，习惯了传统语言的网络服务开发人员可能对多线程并发和协作非常熟悉，但是面对Node.js，我们需要接受和理解它的特点。由此我们是否可以推测出这样的设计会导致负载的压力集中在CPU（事件循环处理？）而不是内存（还记得Java虚拟机抛出OutOfMemory异常的日子吗？），眼见为实，不如来看看淘宝共享数据平台团队对Node.js的性能测试：

- 物理机配置：RHEL 5.2、CPU 2.2GHz、内存4G
- Node.js应用场景：MemCache代理，每次取100字节数据
- 连接池大小：50
- 并发用户数：100
- 测试结果（socket模式）：内存（30M）、QPS（16700）、CPU（95%）

从上面的结果，我们可以看到在这样的测试场景下，qps能够达到16700次，内存仅占用30M（其中V8堆占用22M），CPU则达到95%，可能成为瓶颈。此外，还有不少实践者对Node.js做了性能分析，总的来说，它的性能让人信服，也是受欢迎的重要原因。既然Node.js采用单进程、单线程模式，那么在如今多核硬件流行的环境中，单核性能出色的Node.js如何利用多核CPU呢？创始人Ryan Dahl建议，运行多个Node.js进程，利用某些通信机制来协调各项任务。目前，已经有不少第三方的Node.js多进程支持模块发布，专栏后面的文章会详细讲述Node.js在多核CPU下的编程。

Node.js的另一个特点是它支持的编程语言是Javascript。关于动态语言和静态语言的优缺点比较在这里不再展开讨论。只说三点：

```
var hostRequest = http.request(requestOptions,function(response) {
    var responseHTML ='';
    response.on('data', function (chunk) {
        responseHTML = responseHTML + chunk;
    });
    response.on('end',function(){
        console.log(responseHTML);
        // do something useful
   });
});
```

在上面的代码中，我们需要在end事件中处理responseHTML变量，由于Javascript的闭包特性，我们可以在两个回调函数之外定义responseHTML变量，然后在data事件对应的回调函数中不断修改其值，并最终在end事件中访问

处理。

1. Javascript作为前端工程师的主力语言，在技术社区中有相当的号召力。而且，随着Web技术的不断发展，特别是前端的重要性增加，不少前端工程师开始试水”后台应用“，在许多采用Node.js的企业中，工程师都表示因为习惯了Javascript，所以选择Node.js。
2. Javascript的匿名函数和闭包特性非常适合事件驱动、异步编程，从helloworld例子中我们可以看到回调函数采用了匿名函数的形式来实现，很方便。闭包的作用则更大，看下面的代码示例：
3. Javascript在动态语言中性能较好，有开发人员对Javacript、Python、Ruby等动态语言做了性能分析，发现Javascript的性能要好于其他语言，再加上V8引擎也是同类的佼佼者，所以Node.js的性能也受益其中。

Node.js发展简史

2009年2月，Ryan Dahl在博客上宣布准备基于V8创建一个轻量级的Web服务器并提供一套库。

2009年5月，Ryan Dahl在GitHub上发布了最初版本的部分Node.js包，随后几个月里，有人开始使用Node.js开发应用。

2009年11月和2010年4月，两届JSConf大会都安排了Node.js的讲座。

2010年年底，Node.js获得云计算服务商Joyent资助，创始人Ryan Dahl加入Joyent全职负责Node.js的发展。

2011年7月，Node.js在微软的支持下发布Windows版本。

Node.js应用案例

虽然Node.js诞生刚刚两年多，但是其发展势头逐渐赶超Ruby/Rails，我们在这里列举了部分企业应用Node.js的案例，听听来自客户的声音。

在社交网站LinkedIn最新发布的移动应用中，NodeJS是该移动应用的后台基础。LinkedIn移动开发主管Kiran Prasad对媒体表示，其整个移动软件平台都由NodeJS构建而成：

> LinkedIn内部使用了大量的技术，但是在移动服务器这一块，我们完全基于Node。
>
> （使用它的原因）第一，是因为其灵活性。第二，如果你了解Node，就会发现它最擅长的事情是与其他服务通信。移动应用必须与我们的平台API和数据库交互。我们没有做太多数据分析。相比之前采用的Ruby on Rails技术，开发团队发现Node在性能方面提高很多。他们在每台物理机上跑了15个虚拟服务器（15个实例），其中4个实例即可处理双倍流量。容量评估基于负载测试的结果。

企业社会化服务网站Yammer则利用Node创建了针对其自身平台的跨域代理服务器，第三方的开发人员可以通过该服务器实现从自身域托管的Javascript代码与Yammer平台API的AJAX通信。Yammer平台技术主管Jim Patterson对Node的优点和缺点提出了自己的看法：

> （优点）因为Node是基于事件驱动和无阻塞的，所以非常适合处理并发请求，因此构建在Node上的代理服务器相比其他技术实现（如Ruby）的服务器表现要好得多。此外，与Node代理服务器交互的客户端代码是由javascript语言编写的，因此客户端和服务器端都用同一种语言编写，这是非常美妙的事情。
>
> （缺点）Node是一个相对新的开源项目，所以不太稳定，它总是一直在变，而且缺少足够多的第三方库支持。看起来，就像是Ruby/Rails当年的样子。

知名项目托管网站GitHub也尝试了Node应用。该Node应用称为NodeLoad，是一个存档下载服务器（每当你下载某个存储分支的tarball或者zip文件时就会用到它）。GitHub之前的存档下载服务器采用Ruby编写。在旧系统中，下载存档的请求会创建一个Resque任务。该任务实际上在存档服务器上运行一个git archive命令，从某个文件服务器中取出数据。然后，初始的请求分配给你一个小型Ruby Sinatra应用等待该任务。它其实只是在检查memcache flag是否存在，然后再重定向到最终的下载地址上。旧系统运行大约3个Sinatra实例和3个Resque worker。GitHub

的开发人员觉得这是Node应用的好机会。Node基于事件驱动，相比Ruby的阻塞模型，Node能够更好地处理git存档。在编写新下载服务器过程中，开发人员觉得Node非常适合该功能，此外，他们还里利用了Node库socket.io来监控下载状态。

不仅在国外，Node的优点也同样吸引了国内开发人员的注意，淘宝就实际应用了Node技术：

> MyFOX 是一个数据处理中间件，负责从一个MySQL集群中提取数据、计算并输出统计结果。用户提交一段SQL语句，MyFOX根据该SQL命令的语义，生成各个数据库分片所需要执行的查询语句，并发送至各个分片，再将结果进行汇总和计算。 MyFOX的特点是CPU密集，无文件IO，并只处理只读数据。起初MyFOX使用PHP编写，但遇到许多问题。例如PHP是单线程的，MySQL又需要阻塞查询，因此很难并发请求数据，后来的解决方案是使用nginx和dirzzle，并基于HTTP协议实现接口，并通过curl_multi_get命 令进行请求。不过MyFOX项目组最终还是决定使用Node.js来实现MyFOX。
>
> 选择Node.js有许多方面的原因，比如考虑了兴趣及社区发展，同时也希望可以提高并发能力，榨干CPU。例如，频繁地打开和关闭连接会让大量端口处于等待状态，当并发数量上去之后，时常会因为端口不够用（处于TIME_WAIT状态）而导致连接失败。之前往往是通过修改系统设置来减少等待时间以绕开这个错误，然而使用连接池便可以很好地解决这个问题。此外，以前MyFOX会在某些缓存失效的情况下出现十分密集的访问压力，使用Node.js便可以共享查询状态，让某些请求“等待片刻”，以便系统重新填充缓存内容。

小结

本文简要介绍了Node.js的基本知识，包括概念、特点、历史、案例等等。作为一个仅仅2岁的平台，Node.js的发展势头有目共睹，越来越多的企业开始关注并尝试Node.js，前后端开发人员应该了解相关的内容。

参考文献

[1] http://nodejs.org/

[2] http://beakkon.com/geek/node.js/why-node.js-single-thread-event-loop-javascript

[3] http://www.tbdata.org/archives/1285

[4] http://www.infoq.com/interviews/node-ryan-dahl

[5] http://www.infoq.com/cn/news/2011/08/enterprise-nodejs

[6] http://www.infoq.com/cn/news/2010/11/nodejs-joyent

[7] http://www.infoq.com/cn/news/2011/06/node-exe

[8] http://nodenode.com/post/1176414531/node-js-a-short-history

[9] http://www.infoq.com/cn/news/2011/05/nodeparty-hangzhou

【编者按】：本专栏欢迎有志于宣传和推广Node.js的布道师、开发人员和技术专家投稿，有意者请通过邮件与本专栏主持人崔康（cuikang[at]infoq.com）联系。

（二）Node.js&NPM的安装与配置

专栏的第二篇文章《Node&NPM的安装与配置》介绍Node的安装部署、环境配置以及NPM的安装。

Node.js安装与配置

Node.js已经诞生两年有余，由于一直处于快速开发中，过去的一些安装配置介绍多数针对0.4.x版本而言的，并非适合最新的0.6.x的版本情况了，对此，我们将在0.6.x的版本上介绍Node.js的安装和配置。（本文一律以0.6.1为例，0.6的其余版本，只需替换版本号即可。从http://nodejs.org/#download可以查看到最新的二进制版本和源代码）。

Windows平台下的Node.js安装

在过去，Node.js一直不支持在Windows平台下原生编译，需要借助Cygwin或MinGW来模拟POSIX系统，才能编译安装。幸运的是2011年6月微软开始与Joyent合作移植Node.js到Windows平台上（http://www.infoq.com/cn/news/2011/06/node-exe），这次合作的成果最终呈现在0.6.x的稳定版的发布上。这次的版本发布使得Node.js在Windows平台上的性能大幅度提高，使用方面也更容易和轻巧，完全摆脱掉Cygwin或MinGW等实验室式的环境，并且在某些细节方面，表现出比Linux下更高的性能，细节参见http://www.infoq.com/news/2011/11/Nodejs-Windows。

在Windows（Windows7）平台下，我将介绍二种安装Node.js的方法，即普通和文艺安装方法。

普通的安装方法

普通安装方法其实就是最简单的方法了，对于大多Windows用户而言，都是不太喜欢折腾的人，你可以从这里（http://nodejs.org/dist/v0.6.1/node-v0.6.1.msi）直接下载到Node.js编译好的msi文件。然后双击即可在程序的引导下完成安装。

在命令行中直接运行：

```
node -v
```

命令行将打印出:

```
v0.6.1
```

该引导步骤会将node.exe文件安装到C:\Program Files (x86)\nodejs\目录下，并将该目录添加进PATH环境变量。

文艺的安装方法

Windows平台下的文艺安装方法主要提供给那些热爱折腾，喜欢编译的同学们。在编译源码之前需要注意的是你的Windows系统是否包含编译源码的工具。Node.js的源码主要由C++代码和JavaScript代码构成，但是却用gyp工具（http://code.google.com/p/gyp/）来做源码的项目管理，该工具采用Python语言写成的。在Windows平台上，Node.js采用gyp来生成Visual Studio Solution文件，最终通过VC++的编译器将其编译为二进制文件。所以，你需要满足以下两个条件：

1. Python（Node.js建议使用2.6或更高版本，不推荐3.0），可以从这里（http://python.org/）获取。
2. VC++ 编译器，包含在Visual Studio 2010中（VC++ 2010 Express亦可），VS2010可以从这里（http://msdn.microsoft.com/en-us/vstudio/hh388567）找到。

下载Node.js的0.6.1版本的源码压缩包（http://nodejs.org/dist/v0.6.1/node-v0.6.1.tar.gz）并解压之。

通过命令行进入解压的源码目录，执行vcbuild.bat release命令，然后经历了漫长的等待后，编译完成后，在Release目录下可以找到编译好的node.exe文件。通过命令行执行node -v。

命令行返回结果为：

```
v0.6.1
```

事实上，如果你的编译环境中存在WiX工具集（http://wix.sourceforge.net/ ），执行vcbuild.bat msi release命令，你将会在Relase目录下找到node.msi。

是的，我们回到了一开始的普通安装方法。所谓文艺就是多走一些路，多看一些风景罢了。

Unix/Linux平台下的Node.js安装

由于Node.js尚处于v0.x.x的版本的快速发展中，Unix/Linux平台的发行版都不会预置Node的二进制文件，通过源码进行编译安装是目前最好的选择。而且用Unix/Linux系统的同学们多数都是文艺程序员，本节只介绍如何通过源码进行编译和安装。

安装条件

如同在Windows平台下一样，Node.js依然是采用gyp工具管理生成项目的，不同的是通过make工具进行最终的编译。所以Unix/Linux平台下你需要以下几个必备条件，才能确保编译完成：

1. Python。用于gyp，可以通过在shell下执行python命令，查看是否已安装python，并确认版本是否符合需求（2.6或更高版本，但不推荐3.0）。
2. 源代码编译器，通常 Unix/Linux平台都自带了C++的编译器（GCC/G++）。如果没有，请通过当前发行版的软件包安装工具安装make，g++这些编译工具。
    1. Debian/Ubuntu下的工具是apt-get
    2. RedHat/centOS下通过yum命令
    3. Mac OS X下你可能需要安装xcode来获得编译器
3. 其次，如果你计划在Node.js中启用网络加密，OpenSSL的加密库也是必须的。该加密库是libssl-dev，可以通过apt-get install libssl-dev等命令安装。

检查环境并安装

完成以上预备条件后，我们获取源码并进行环境检查吧：

```
wget http://nodejs.org/dist/v0.6.1/node-v0.6.1.tar.gz
tar zxvf node-v0.6.1.tar.gz
cd node-v0.6.1
./configure
```

上面几行命令是通过wget命令下载最新版本的代码，并解压之。./configure命令将会检查环境是否符合Nodejs的编译需要。

```
Checking for program g++ or c++ : /usr/bin/g++
Checking for program cpp : /usr/bin/cpp
Checking for program ar : /usr/bin/ar
Checking for program ranlib : /usr/bin/ranlib
Checking for g++ : ok
Checking for program gcc or cc : /usr/bin/gcc
Checking for program ar : /usr/bin/ar
Checking for program ranlib : /usr/bin/ranlib
Checking for gcc : ok
Checking for library dl : yes
Checking for openssl : yes
Checking for library util : yes
Checking for library rt : yes
Checking for fdatasync(2) with c++ : yes
'configure' finished successfully (7.350s)
```

如果检查没有通过，请确认上面提到的三个条件是否满足。如果configure命令执行成功，就可以进行编译了：

```
make
make install
```

Nodejs通过make工具进行编译和安装（如果make install不成功，请使用sudo以确保拥有权限）。完成以上两步后，检查一下是否安装成功：

```
node -v
```

检查是否返回：

```
v0.6.1
```

至此，Nodejs已经编译并安装完成。如需卸载，可以执行make uninstall进行卸载。

小结

以上介绍了*nix和Windows平台下Nodejs的安装，之后可以如同Nodejs官方网站上介绍的那样，编写example.js文件。

```
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/');
```

在命令行中执行它：

```
node example.js
```

你就可以通过浏览器访问http://127.0.0.1:1337得到Hello World的响应。

安装NPM

NPM的全称是Node Package Manager，如果你熟悉ruby的gem，Python的PyPL、setuptools，PHP的pear，那么你就知道NPM的作用是什么了。没错，它就是Nodejs的包管理器。Nodejs自身提供了基本的模块。但是在这些基本模块上开发实际应用需要较多的工作。所幸的是笔者执笔此文的时候NPM上已经有了5112个Nodejs库或框架，这些库从各个方面可以帮助Nodejs的开发者完成较为复杂的应用。这些库的数量和活跃也从侧面反映出Nodejs社区的发展是十分神速和活跃的。下面我将介绍安装NPM和通过NPM安装Nodejs的第三方库，以及在大陆的网络环境下，如何更好的利用NPM。

Unix/Linux下安装NPM

就像NPM的官网（http://npmjs.org/）上介绍的那样，安装NPM仅仅是一行命令的事情：

```
curl http://npmjs.org/install.sh | sh
```

这里详解一下这句命令的意思，curl http://npmjs.org/install.sh是通过curl命令获取这个安装shell脚本，按后通过管道符| 将获取的脚本交由sh命令来执行。这里如果没有权限会安装不成功，需要加上sudo来确保权限：

```
curl http://npmjs.org/install.sh | sudo sh
```

安装成功后执行npm命令，会得到一下的提示：

```
Usage: npm
where  is one of:
adduser, apihelp, author, bin, bugs, c, cache, completion,
config, deprecate, docs, edit, explore, faq, find, get,
help, help-search, home, i, info, init, install, la, link,
list, ll, ln, ls, outdated, owner, pack, prefix, prune,
publish, r, rb, rebuild, remove, restart, rm, root,
run-script, s, se, search, set, show, star, start, stop,
submodule, tag, test, un, uninstall, unlink, unpublish,
unstar, up, update, version, view, whoami
```

我们以underscore为例，来展示下通过npm安装第三方包的过程。

```
npm install underscore
```

返回：

```
underscore@1.2.2 ./node_modules/underscore
```

由于一些特殊的网络环境，直接通过npm install命令安装第三方库的时候，经常会出现卡死的状态。幸运的是国内CNode社区的@fire9同学利用空余时间搭建了一个镜像的NPM资源库，服务器架设在日本，可以绕过某些不必要的网络问题。你可以通过以下这条命令来安装第三方库：

```
npm --registry "http://npm.hacknodejs.com" install underscore
```

如果你想将它设为默认的资源库，运行下面这条命令即可：

```
npm config set registry "http://npm.hacknodejs.com/"
```

设置之后每次安装时就可以不用带上—registry参数。值得一提的是还有另一个镜像可用，该镜像地址是http://registry.npmjs.vitecho.com，如需使用，替换上面两行命令的地址即可。

Windows下安装NPM

由于Nodejs最初在Linux开发下的历史原因，导致NPM一开始也不支持Windows环境，但是随着Nodejs成功移植到到Windows平台，NPM在Windows下的需求亦是日渐增加。下面开始Windows下的NPM之旅吧。

安装GIT工具

由于github网站不支持直接下载打包了所有submodule的源码包，所以需要通过git工具来签出所有的源码。从http://code.google.com/p/msysgit/downloads/list，可以下载到msysgit这个Windows平台下的git客户端工具（最新版本文件为Git-1.7.7.1-preview20111027.exe）。在下载之后双击安装。

下载NPM源码

打开命令行工具（CMD），执行以下命令，可以通过msysgit签出NPM的所有源码和依赖代码并安装npm。

```
git clone --recursive git://github.com/isaacs/npm.git
cd npm
node cli.js install npm -gf
```

在执行这段代码之前，请确保node.exe是跟通过node.msi的方式安装的，或者在PATH环境变量中。这段命令也会将npm加入到PATH环境变量中去，之后可以随处执行npm命令。如果安装中遇到权限方面的错误，请确保cmd命令行工具是通过管理员身份运行的。安装成功后，执行以下命令：

```
npm install underscore
```

返回：

```
underscore@1.2.2 ./node_modules/underscore
```

如此，Windows平台下的NPM安装完毕。如果遭遇网络问题无法安装，请参照Linux下的NPM命令，添加镜像地址。

参考文献

- http://nodejs.org/
- https://github.com/joyent/node/wiki/Installation
- http://npmjs.org/doc/README.html#Installing-on-Windows-Experimental

（三）深入Node.js的模块机制

专栏的第三篇文章《深入Node.js的模块机制》。之前介绍了Node.js安装的基础知识，本文将深入Node.js的模块机制。

Node.js模块的实现

之前在网上查阅了许多介绍Node.js的文章，可惜对于Node.js的模块机制大都着墨不多。在后续介绍模块的使用之前，我认为有必要深入一下Node.js的模块机制。

CommonJS规范

早在Netscape诞生不久后，JavaScript就一直在探索本地编程的路，Rhino是其代表产物。无奈那时服务端JavaScript走的路均是参考众多服务器端语言来实现的，在这样的背景之下，一没有特色，二没有实用价值。但是随着JavaScript在前端的应用越来越广泛，以及服务端JavaScript的推动，JavaScript现有的规范十分薄弱，不利于JavaScript大规模的应用。那些以JavaScript为宿主语言的环境中，只有本身的基础原生对象和类型，更多的对象和API都取决于宿主的提供，所以，我们可以看到JavaScript缺少这些功能：

- JavaScript没有模块系统。没有原生的支持密闭作用域或依赖管理。
- JavaScript没有标准库。除了一些核心库外，没有文件系统的API，没有IO流API等。
- JavaScript没有标准接口。没有如Web Server或者数据库的统一接口。
- JavaScript没有包管理系统。不能自动加载和安装依赖。

于是便有了CommonJS（http://www.commonjs.org）规范的出现，其目标是为了构建JavaScript在包括Web服务器，桌面，命令行工具，及浏览器方面的生态系统。

CommonJS制定了解决这些问题的一些规范，而Node.js就是这些规范的一种实现。Node.js自身实现了require方法作为其引入模块的方法，同时NPM也基于CommonJS定义的包规范，实现了依赖管理和模块自动安装等功能。这里我们将深入一下Node.js的require机制和NPM基于包规范的应用。

简单模块定义和使用

在Node.js中，定义一个模块十分方便。我们以计算圆形的面积和周长两个方法为例，来表现Node.js中模块的定义方式。

```
var PI = Math.PI;
exports.area = function (r) {
    return PI * r * r;
};
exports.circumference = function (r) {
    return 2 * PI * r;
};
```

将这个文件存为circle.js，并新建一个app.js文件，并写入以下代码：

```
var circle = require('./circle.js');
console.log( 'The area of a circle of radius 4 is ' + circle.area(4));
```

可以看到模块调用也十分方便，只需要require需要调用的文件即可。

在require了这个文件之后，定义在exports对象上的方法便可以随意调用。Node.js将模块的定义和调用都封装得极其简单方便，从API对用户友好这一个角度来说，Node.js的模块机制是非常优秀的。

模块载入策略

Node.js的模块分为两类，一类为原生（核心）模块，一类为文件模块。原生模块在Node.js源代码编译的时候编译进了二进制执行文件，加载的速度最快。另一类文件模块是动态加载的，加载速度比原生模块慢。但是Node.js对原生模块和文件模块都进行了缓存，于是在第二次require时，是不会有重复开销的。其中原生模块都被定义在lib这个目录下面，文件模块则不定性。

```
node app.js
```

由于通过命令行加载启动的文件几乎都为文件模块。我们从Node.js如何加载文件模块开始谈起。加载文件模块的工作，主要由原生模块module来实现和完成，该原生模块在启动时已经被加载，进程直接调用到runMain静态方法。

```
// bootstrap main module.
Module.runMain = function () {
    // Load the main module--the command line argument.
    Module._load(process.argv[1], null, true);
};
```

_load静态方法在分析文件名之后执行

```
var module = new Module(id, parent);
```

并根据文件路径缓存当前模块对象，该模块实例对象则根据文件名加载。

```
module.load(filename);
```

实际上在文件模块中，又分为3类模块。这三类文件模块以后缀来区分，Node.js会根据后缀名来决定加载方法。

- .js。通过fs模块同步读取js文件并编译执行。
- .node。通过C/C++进行编写的Addon。通过dlopen方法进行加载。
- .json。读取文件，调用JSON.parse解析加载。

这里我们将详细描述js后缀的编译过程。Node.js在编译js文件的过程中实际完成的步骤有对js文件内容进行头尾包装。以app.js为例，包装之后的app.js将会变成以下形式：

```
(function (exports, require, module, __filename, __dirname) {
    var circle = require('./circle.js');
    console.log('The area of a circle of radius 4 is ' + circle.area(4));
});
```

这段代码会通过vm原生模块的runInThisContext方法执行（类似eval，只是具有明确上下文，不污染全局），返回为一个具体的function对象。最后传入module对象的exports，require方法，module，文件名，目录名作为实参并执行。

这就是为什么require并没有定义在app.js 文件中，但是这个方法却存在的原因。从Node.js的API文档中可以看到还有__filename、__dirname、module、exports几个没有定义但是却存在的变量。其中__filename和__dirname在查找文件路径的过程中分析得到后传入的。module变量是这个模块对象自身，exports是在module的构造函数中初始化的一个空对象（{}，而不是null）。

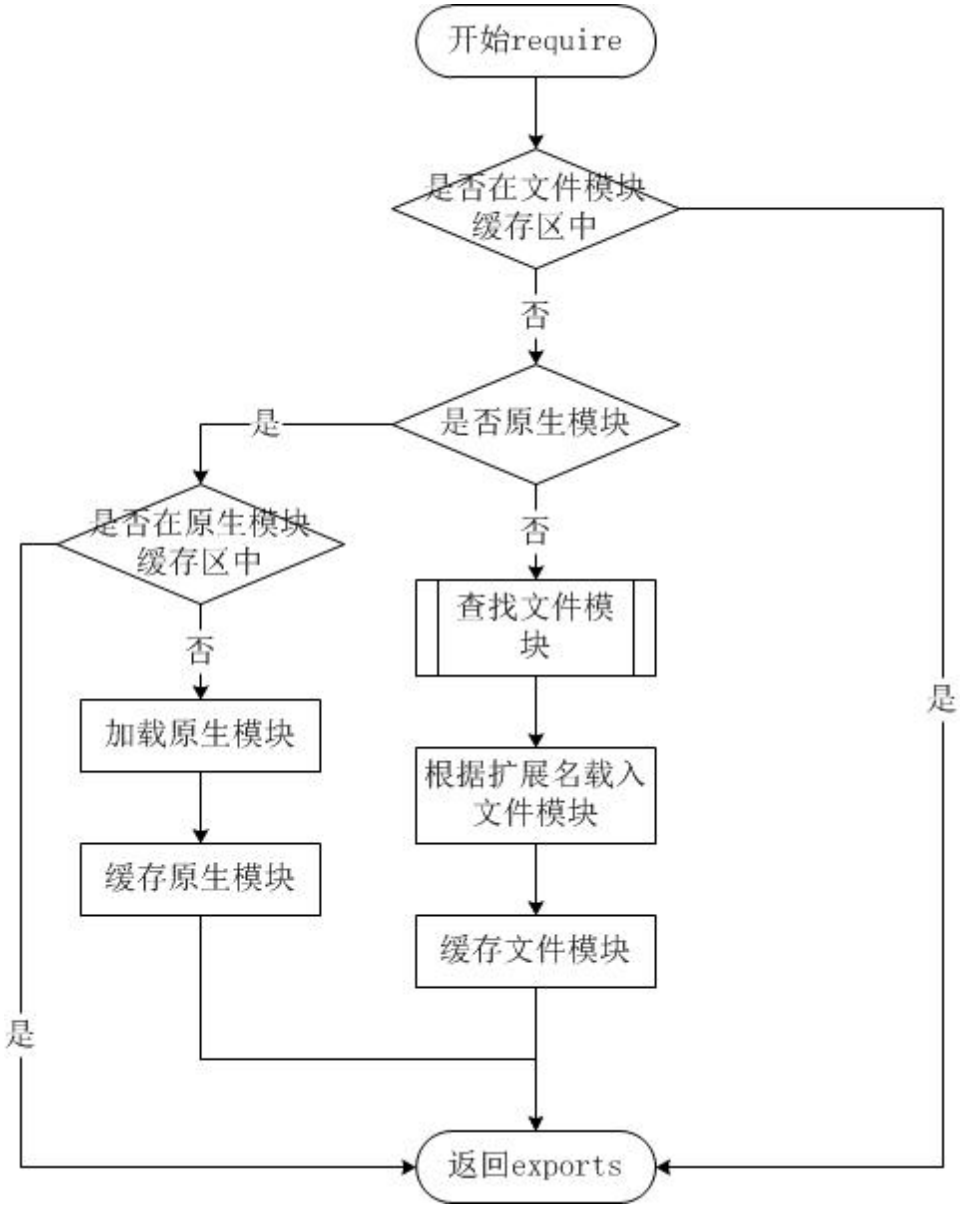
在这个主文件中，可以通过require方法去引入其余的模块。而其实这个require方法实际调用的就是load方法。

load方法在载入、编译、缓存了module后，返回module的exports对象。这就是circle.js文件中只有定义在exports对象上的方法才能被外部调用的原因。

以上所描述的模块载入机制均定义在lib/module.js中。

require方法中的文件查找策略

由于Node.js中存在4类模块（原生模块和3种文件模块），尽管require方法极其简单，但是内部的加载却是十分复杂的，其加载优先级也各自不同。



从文件模块缓存中加载

尽管原生模块与文件模块的优先级不同，但是都不会优先于从文件模块的缓存中加载已经存在的模块。

从原生模块加载

原生模块的优先级仅次于文件模块缓存的优先级。require方法在解析文件名之后，优先检查模块是否在原生模块列表中。以http模块为例，尽管在目录下存在一个http/http.js/http.node/http.json文件，require("http")都不会从这些文件中加载，而是从原生模块中加载。

原生模块也有一个缓存区，同样也是优先从缓存区加载。如果缓存区没有被加载过，则调用原生模块的加载方式进行加载和执行。

从文件加载

当文件模块缓存中不存在，而且不是原生模块的时候，Node.js会解析require方法传入的参数，并从文件系统中加载实际的文件，加载过程中的包装和编译细节在前一节中已经介绍过，这里我们将详细描述查找文件模块的过程，其中，也有一些细节值得知晓。

require方法接受以下几种参数的传递：

- http、fs、path等，原生模块。
- ./mod或../mod，相对路径的文件模块。
- /pathtomodule/mod，绝对路径的文件模块。
- mod，非原生模块的文件模块。

在进入路径查找之前有必要描述一下module path这个Node.js中的概念。对于每一个被加载的文件模块，创建这个模块对象的时候，这个模块便会有一个paths属性，其值根据当前文件的路径计算得到。我们创建modulepath.js这样一个文件，其内容为：

```
console.log(module.paths);
```

我们将其放到任意一个目录中执行node modulepath.js命令，将得到以下的输出结果。

```
[ '/home/jackson/research/node_modules',
'/home/jackson/node_modules',
'/home/node_modules',
'/node_modules' ]
```

Windows下：

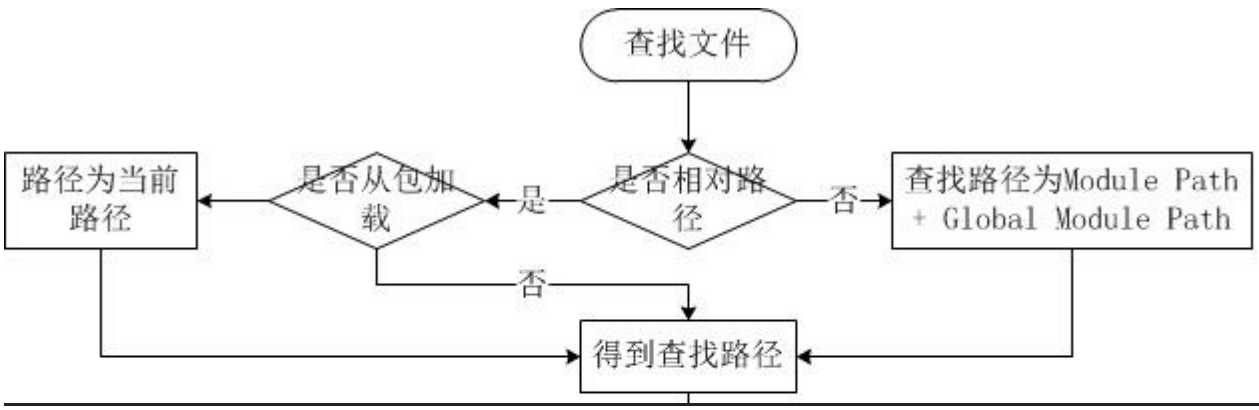
```
[ 'c:\\nodejs\\node_modules', 'c:\\node_modules' ]
```

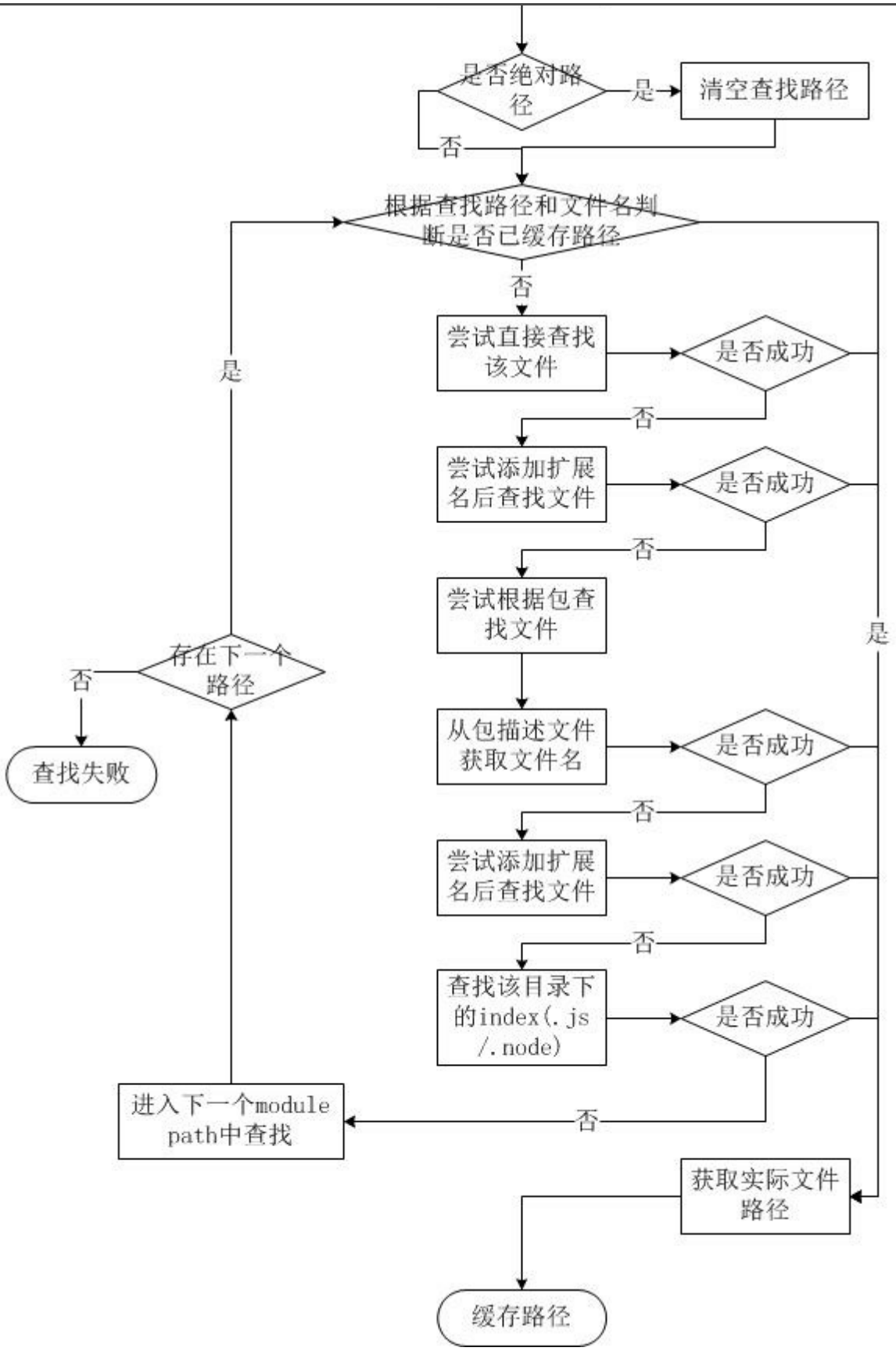
可以看出module path的生成规则为：从当前文件目录开始查找node_modules目录；然后依次进入父目录，查找父目录下的node_modules目录；依次迭代，直到根目录下的node_modules目录。

除此之外还有一个全局module path，是当前node执行文件的相对目录（`../../lib/node`）。如果在环境变量中设置了HOME目录和`NODE_PATH`目录的话，整个路径还包含NODE_PATH和HOME目录下的.node_libraries与.node_modules。其最终值大致如下：

```
[NODE_PATH，HOME/.node_modules，HOME/.node_libraries，execPath/../../lib/node]
```

下图是笔者从源代码中整理出来的整个文件查找流程：





简而言之，如果require绝对路径的文件，查找时不会去遍历每一个node_modules目录，其速度最快。其余流程如下：

1. 从module path数组中取出第一个目录作为查找基准。
2. 直接从目录中查找该文件，如果存在，则结束查找。如果不存在，则进行下一条查找。
3. 尝试添加.js、.json、.node后缀后查找，如果存在文件，则结束查找。如果不存在，则进行下一条。
4. 尝试将require的参数作为一个包来进行查找，读取目录下的package.json文件，取得main参数指定的文件。
5. 尝试查找该文件，如果存在，则结束查找。如果不存在，则进行第3条查找。

6. 如果继续失败，则取出module path数组中的下一个目录作为基准查找，循环第1至5个步骤。
7. 如果继续失败，循环第1至6个步骤，直到module path中的最后一个值。
8. 如果仍然失败，则抛出异常。

整个查找过程十分类似原型链的查找和作用域的查找。所幸Node.js对路径查找实现了缓存机制，否则由于每次判断路径都是同步阻塞式进行，会导致严重的性能消耗。

包结构

前面提到，JavaScript缺少包结构。CommonJS致力于改变这种现状，于是定义了包的结构规范（http://wiki.commonjs.org/wiki/Packages/1.0 ）。而NPM的出现则是为了在CommonJS规范的基础上，实现解决包的安装卸载，依赖管理，版本管理等问题。require的查找机制明了之后，我们来看一下包的细节。

一个符合CommonJS规范的包应该是如下这种结构：

- 一个package.json文件应该存在于包顶级目录下
- 二进制文件应该包含在bin目录下。
- JavaScript代码应该包含在lib目录下。
- 文档应该在doc目录下。
- 单元测试应该在test目录下。

由上文的require的查找过程可以知道，Node.js在没有找到目标文件时，会将当前目录当作一个包来尝试加载，所以在package.json文件中最重要的一个字段就是main。而实际上，这一处是Node.js的扩展，标准定义中并不包含此字段，对于require，只需要main属性即可。但是在除此之外包需要接受安装、卸载、依赖管理，版本管理等流程，所以CommonJS为package.json文件定义了如下一些必须的字段：

- name。包名，需要在NPM上是唯一的。不能带有空格。
- description。包简介。通常会显示在一些列表中。
- version。版本号。一个语义化的版本号（http://semver.org/ ），通常为x.y.z。该版本号十分重要，常常用于一些版本控制的场合。
- keywords。关键字数组。用于NPM中的分类搜索。
- maintainers。包维护者的数组。数组元素是一个包含name、email、web三个属性的JSON对象。
- contributors。包贡献者的数组。第一个就是包的作者本人。在开源社区，如果提交的patch被merge进master分支的话，就应当加上这个贡献patch的人。格式包含name和email。如：

```
"contributors": [{
    "name": "Jackson Tian",
    "email": "mail @gmail.com"
    }, {
    "name": "fengmk2",
    "email": "mail2@gmail.com"
}],
```

- bugs。一个可以提交bug的URL地址。可以是邮件地址（mailto:mailxx@domain），也可以是网页地址（http://url）。
- licenses。包所使用的许可证。例如：

```
"licenses": [{
    "type": "GPLv2",
    "url": "http://www.example.com/licenses/gpl.html",
}]
```

- repositories。托管源代码的地址数组。
- dependencies。当前包需要的依赖。这个属性十分重要，NPM会通过这个属性，帮你自动加载依赖的包。

以下是Express框架的package.json文件，值得参考。

```
{
    "name": "express",
    "description": "Sinatra inspired web development framework",
    "version": "3.0.0alpha1-pre",
    "author": "TJ Holowaychuk
```

除了前面提到的几个必选字段外，我们还发现了一些额外的字段，如bin、scripts、engines、devDependencies、author。这里可以重点提及一下scripts字段。包管理器（NPM）在对包进行安装或者卸载的时候需要进行一些编译或者清除的工作，scripts字段的对象指明了在进行操作时运行哪个文件，或者执行拿条命令。如下为一个较全面的scripts案例：

```
 "scripts": {
    "install": "install.js",
    "uninstall": "uninstall.js",
    "build": "build.js",
    "doc": "make-doc.js",
    "test": "test.js",
}
```

如果你完善了自己的JavaScript库，使之实现了CommonJS的包规范，那么你可以通过NPM来发布自己的包，为NPM上5000+的基础上再加一个模块。

```
npm publish
```

命令十分简单。但是在这之前你需要通过npm adduser命令在NPM上注册一个帐户，以便后续包的维护。NPM会分析该文件夹下的package.json文件，然后上传目录到NPM的站点上。用户在使用你的包时，也十分简明：

```
npm install
```

甚至对于NPM无法安装的包（因为某些奇怪的网络原因），可以通过github手动下载其稳定版本，解压之后通过以下命令进行安装：

```
npm install
```

只需将路径指向package.json存在的目录即可。然后在代码中require('package')即可使用。

Node.js中的require内部流程之复杂，而方法调用之简单，实在值得叹为观止。更多NPM使用技巧可以参见http://www.infoq.com/cn/articles/msh-using-npm-manage-node.js-dependence。

Node.js模块与前端模块的异同

通常有一些模块可以同时适用于前后端，但是在浏览器端通过script标签的载入JavaScript文件的方式与Node.js不同。Node.js在载入到最终的执行中，进行了包装，使得每个文件中的变量天然的形成在一个闭包之中，不会污染全局变量。而浏览器端则通常是裸露的JavaScript代码片段。所以为了解决前后端一致性的问题，类库开发者需要将类

库代码包装在一个闭包内。以下代码片段抽取自著名类库underscore的定义方式。

```
(function () {
    // Establish the root object, `window` in the browser, or `global` on the server.
    var root = this;
    var _ = function (obj) {
          return new wrapper(obj);
        };
    if (typeof exports !== 'undefined') {
        if (typeof module !== 'undefined' && module.exports) {
            exports = module.exports = _;
        }
        exports._ = _;
    } else if (typeof define === 'function' && define.amd) {
        // Register as a named module with AMD.
        define('underscore', function () {
            return _;
        });
    } else {
        root['_'] = _;
    }
}).call(this);
```

首先，它通过function定义构建了一个闭包，将this作为上下文对象直接call调用，以避免内部变量污染到全局作用域。续而通过判断exports是否存在来决定将局部变量*绑定给exports，并且根据define变量是否存在，作为处理在实现了AMD规范环境（http://wiki.commonjs.org/wiki/Modules/AsynchronousDefinition）下的使用案例。仅只当处于浏览器的环境中的时候，this指向的是全局对象（window对象），才将*变量赋在全局对象上，作为一个全局对象的方法导出，以供外部调用。

所以在设计前后端通用的JavaScript类库时，都有着以下类似的判断：

```
if (typeof exports !== "undefined") {
    exports.EventProxy = EventProxy;
} else {
    this.EventProxy = EventProxy;
}
```

即，如果exports对象存在，则将局部变量挂载在exports对象上，如果不存在，则挂载在全局对象上。

对于更多前端的模块实现可以参考国内淘宝玉伯的seajs（http://seajs.com/），或者思科杜欢的oye（http://www.w3cgroup.com/oye/）。

参考文献

- http://www.commonjs.org
- http://npmjs.org/doc/README.html
- http://www.infoq.com/cn/articles/msh-using-npm-manage-node.js-dependence
- http://nodejs.org/docs/latest/api/modules.html

（四）Node.js的事件机制

专栏的第四篇文章《Node.js的事件机制》。之前介绍了Node.js的模块机制，本文将深入Node.js的事件部分。

Node.js的事件机制

Node.js在其Github代码仓库（https://github.com/joyent/node）上有着一句短短的介绍：Evented I/O for V8 JavaScript。这句近似广告语的句子却道尽了Node.js自身的特色所在：基于V8引擎实现的事件驱动IO。在本文的这部分内容中，我来揭开这Evented这个关键词的一切奥秘吧。

Node.js能够在众多的后端JavaScript技术之中脱颖而出，正是因其基于事件的特点而受到欢迎。拿Rhino来做比较，可以看出Rhino引擎支持的后端JavaScript摆脱不掉其他语言同步执行的影响，导致JavaScript在后端编程与前端编程之间有着十分显著的差别，在编程模型上无法形成统一。在前端编程中，事件的应用十分广泛，DOM上的各种事件。在Ajax大规模应用之后，异步请求更得到广泛的认同，而Ajax亦是基于事件机制的。在Rhino中，文件读取等操作，均是同步操作进行的。在这类单线程的编程模型下，如果采用同步机制，无法与PHP之类的服务端脚本语言的成熟度媲美，性能也没有值得可圈可点的部分。直到Ryan Dahl在2009年推出Node.js后，后端JavaScript才走出其迷局。Node.js的推出，我觉得该变了两个状况：

1. 统一了前后端JavaScript的编程模型。
2. 利用事件机制充分利用用异步IO突破单线程编程模型的性能瓶颈，使得JavaScript在后端达到实用价值。

有了第二次浏览器大战中的佼佼者V8的适时助力，使得Node.js在短短的两年内达到可观的运行效率，并迅速被大家接受。这一点从Node.js项目在Github上的流行度和NPM上的库的数量可见一斑。

至于Node.js为何会选择Evented I/O for V8 JavaScript的结构和形式来实现，可以参见一下2011年初对作者Ryan Dahl的一次采访：http://bostinno.com/2011/01/31/node-js-interview-4-questions-with-creator-ryan-dahl/ 。

事件机制的实现

Node.js中大部分的模块，都继承自Event模块（http://nodejs.org/docs/latest/api/events.html ）。Event模块（events.EventEmitter）是一个简单的事件监听器模式的实现。具有addListener/on，once，removeListener，removeAllListeners，emit等基本的事件监听模式的方法实现。它与前端DOM树上的事件并不相同，因为它不存在冒泡，逐层捕获等属于DOM的事件行为，也没有preventDefault()、stopPropagation()、 stopImmediatePropagation() 等处理事件传递的方法。

从另一个角度来看，事件侦听器模式也是一种事件钩子（hook）的机制，利用事件钩子导出内部数据或状态给外部调用者。Node.js中的很多对象，大多具有黑盒的特点，功能点较少，如果不通过事件钩子的形式，对象运行期间的中间值或内部状态，是我们无法获取到的。这种通过事件钩子的方式，可以使编程者不用关注组件是如何启动和执行的，只需关注在需要的事件点上即可。

```
var options = {
    host: 'www.google.com',
    port: 80,
    path: '/upload',
    method: 'POST'
};
var req = http.request(options, function (res) {
    console.log('STATUS: ' + res.statusCode);
    console.log('HEADERS: ' + JSON.stringify(res.headers));
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
        console.log('BODY: ' + chunk);
    });
});
req.on('error', function (e) {
    console.log('problem with request: ' + e.message);
});
// write data to request body
req.write('data\n');
req.write('data\n');
req.end();
```

在这段HTTP request的代码中，程序员只需要将视线放在error，data这些业务事件点即可，至于内部的流程如何，无需过于关注。

值得一提的是如果对一个事件添加了超过10个侦听器，将会得到一条警告，这一处设计与Node.js自身单线程运行有关，设计者认为侦听器太多，可能导致内存泄漏，所以存在这样一个警告。调用：

```
emitter.setMaxListeners(0);
```

可以将这个限制去掉。

其次，为了提升Node.js的程序的健壮性，EventEmitter对象对error事件进行了特殊对待。如果运行期间的错误触发了error事件。EventEmitter会检查是否有对error事件添加过侦听器，如果添加了，这个错误将会交由该侦听器处理，否则，这个错误将会作为异常抛出。如果外部没有捕获这个异常，将会引起线程的退出。

事件机制的进阶应用

继承event.EventEmitter

实现一个继承了EventEmitter类是十分简单的，以下是Node.js中流对象继承EventEmitter的例子：

```
function Stream() {
    events.EventEmitter.call(this);
}
util.inherits(Stream, events.EventEmitter);
```

Node.js在工具模块中封装了继承的方法，所以此处可以很便利地调用。程序员可以通过这样的方式轻松继承EventEmitter对象，利用事件机制，可以帮助你解决一些问题。

多事件之间协作

在略微大一点的应用中，数据与Web服务器之间的分离是必然的，如新浪微博、Facebook、Twitter等。这样的优势在于数据源统一，并且可以为相同数据源制定各种丰富的客户端程序。以Web应用为例，在渲染一张页面的时候，通常需要从多个数据源拉取数据，并最终渲染至客户端。Node.js在这种场景中可以很自然很方便的同时并行发起对多个数据源的请求。

```
api.getUser("username", function (profile) {
    // Got the profile
});
api.getTimeline("username", function (timeline) {
    // Got the timeline
});
api.getSkin("username", function (skin) {
    // Got the skin
});
```

Node.js通过异步机制使请求之间无阻塞，达到并行请求的目的，有效的调用下层资源。但是，这个场景中的问题是对于多个事件响应结果的协调并非被Node.js原生优雅地支持。为了达到三个请求都得到结果后才进行下一个步骤，程序也许会被变成以下情况：

```
api.getUser("username", function (profile) {
    api.getTimeline("username", function (timeline) {
        api.getSkin("username", function (skin) {
            // TODO
        });
    });
});
```

这将导致请求变为串行进行，无法最大化利用底层的API服务器。

为解决这类问题，我曾写作一个模块（EventProxy，https://github.com/JacksonTian/eventproxy）来实现多事件协作，以下为上面代码的改进版：

```
var proxy = new EventProxy();
proxy.all("profile", "timeline", "skin", function (profile, timeline, skin) {
    // TODO
});
api.getUser("username", function (profile) {
    proxy.emit("profile", profile);
});
api.getTimeline("username", function (timeline) {
    proxy.emit("timeline", timeline);
});
api.getSkin("username", function (skin) {
    proxy.emit("skin", skin);
});
```

EventProxy也是一个简单的事件侦听者模式的实现，由于底层实现跟Node.js的EventEmitter不同，无法合并进Node.js中。但是却提供了比EventEmitter更强大的功能，且API保持与EventEmitter一致，与Node.js的思路保持契合，并可以适用在前端中。

这里的all方法是指侦听完profile、timeline、skin三个方法后，执行回调函数，并将侦听接收到的数据传入。

最后还介绍一种解决多事件协作的方案：Jscex（https://github.com/JeffreyZhao/jscex ）。Jscex通过运行时编译的思路（需要时也可在运行前编译），将同步思维的代码转换为最终异步的代码来执行，可以在编写代码的时候通过同步思维来写，可以享受到同步思维的便利写作，异步执行的高效性能。如果通过Jscex编写，将会是以下形式：

```
var data = $await(Task.whenAll({
    profile: api.getUser("username"),
    timeline: api.getTimeline("username"),
    skin: api.getSkin("username")
}));
// 使用data.profile, data.timeline, data.skin
// TODO
```

此节感谢Jscex作者@老赵（http://blog.zhaojie.me/ ）的指正和帮助。

利用事件队列解决雪崩问题

所谓雪崩问题，是在缓存失效的情景下，大并发高访问量同时涌入数据库中查询，数据库无法同时承受如此大的查询请求，进而往前影响到网站整体响应缓慢。那么在Node.js中如何应付这种情景呢。

```
var select = function (callback) {
        db.select("SQL", function (results) {
            callback(results);
        });
    };
```

以上是一句数据库查询的调用，如果站点刚好启动，这时候缓存中是不存在数据的，而如果访问量巨大，同一句SQL会被发送到数据库中反复查询，影响到服务的整体性能。一个改进是添加一个状态锁。

```
var status = "ready";
var select = function (callback) {
        if (status === "ready") {
            status = "pending";
            db.select("SQL", function (results) {
                callback(results);
                status = "ready";
            });
        }
    };
```

但是这种情景，连续的多次调用select发，只有第一次调用是生效的，后续的select是没有数据服务的。所以这个时候引入事件队列吧：

```
var proxy = new EventProxy();
var status = "ready";
var select = function (callback) {
        proxy.once("selected", callback);
        if (status === "ready") {
            status = "pending";
            db.select("SQL", function (results) {
                proxy.emit("selected", results);
                status = "ready";
            });
        }
    };
```

这里利用了EventProxy对象的once方法，将所有请求的回调都压入事件队列中，并利用其执行一次就会将监视器移除的特点，保证每一个回调只会被执行一次。对于相同的SQL语句，保证在同一个查询开始到结束的时间中永远只有一次，在这查询期间到来的调用，只需在队列中等待数据就绪即可，节省了重复的数据库调用开销。由于Node.js单线程执行的原因，此处无需担心状态问题。这种方式其实也可以应用到其他远程调用的场景中，即使外部没有缓存策略，也能有效节省重复开销。此处也可以用EventEmitter替代EventProxy，不过可能存在侦听器过多，引发警告，需要调用setMaxListeners(0)移除掉警告，或者设更大的警告阀值。

参考：

- http://nodejs.org/docs/latest/api/events.html
- https://github.com/JacksonTian/eventproxy/blob/master/README.md
- https://github.com/JeffreyZhao/jscex/blob/master/README-cn.md

（五）初探Node.js的异步I/O实现

专栏的第五篇文章《Node.js的异步实现》。之前介绍了Node.js的事件机制，也许读者对此尚会觉得意犹未尽，因为仅仅只是简单的事件机制，并不能道尽Node.js的神奇。如果Node.js是一盘别开生面的磁带，那么事件与异步分别是其A面和B面，它们共同组成了Node.js的别样之处。本文将翻转Node.js到B面，与你共同聆听。

异步I/O

在操作系统中，程序运行的空间分为内核空间和用户空间。我们常常提起的异步I/O，其实质是用户空间中的程序不用依赖内核空间中的I/O操作实际完成，即可进行后续任务。以下伪代码模仿了一个从磁盘上获取文件和一个从网络中获取文件的操作。异步I/O的效果就是getFileFromNet的调用不依赖于getFile调用的结束。

```
getFile("file_path");
getFileFromNet("url");
```

如果以上两个任务的时间分别为m和n。采用同步方式的程序要完成这两个任务的时间总花销会是m + n。但是如果是采用异步方式的程序，在两种I/O可以并行的状况下（比如网络I/O与文件I/O），时间开销将会减小为max(m, n)。

异步I/O的必要性

有的语言为了设计得使应用程序调用方便，将程序设计为同步I/O的模型。这意味着程序中的后续任务都需要等待

I/O的完成。在等待I/O完成的过程中，程序无法充分利用CPU。为了充分利用CPU，和使I/O可以并行，目前有两种方式可以达到目的：

- 多线程单进程 多线程的设计之处就是为了在共享的程序空间中，实现并行处理任务，从而达到充分利用CPU的效果。多线程的缺点在于执行时上下文交换的开销较大，和状态同步（锁）的问题。同样它也使得程序的编写和调用复杂化。
- 单线程多进程 为了避免多线程造成的使用不便问题，有的语言选择了单线程保持调用简单化，采用启动多进程的方式来达到充分利用CPU和提升总体的并行处理能力。 它的缺点在于业务逻辑复杂时（涉及多个I/O调用），因为业务逻辑不能分布到多个进程之间，事务处理时长要远远大于多线程模式。

前者在性能优化上还有回旋的余地，后者的做法纯粹是一种加三倍服务器的行为。 而且现在的大型Web应用中，单机的情形是十分稀少的，一个事务往往需要跨越网络几次才能完成最终处理。如果网络速度不够理想，m和n值都将会变大，这时同步I/O的语言模型将会露出其最脆弱的状态。 这种场景下的异步I/O将会体现其优势，max(m, n)的时间开销可以有效地缓解m和n值增长带来的性能问题。而当并行任务更多的时候，m + n + ...与max(m, n, ...)之间的孰优孰劣更是一目了然。从这个公式中，可以了解到异步I/O在分布式环境中是多么重要，而Node.js天然地支持这种异步I/O，这是众多云计算厂商对其青睐的根本原因。

操作系统对异步I/O的支持

我们听到Node.js时，我们常常会听到异步，非阻塞，回调，事件这些词语混合在一起。其中，异步与非阻塞听起来似乎是同一回事。从实际效果的角度说，异步和非阻塞都达到了我们并行I/O的目的。但是从计算机内核I/O而言，异步/同步和阻塞/非阻塞实际上时两回事。

- I/O的阻塞与非阻塞 阻塞模式的I/O会造成应用程序等待，直到I/O完成。同时操作系统也支持将I/O操作设置为非阻塞模式，这时应用程序的调用将可能在没有拿到真正数据时就立即返回了，为此应用程序需要多次调用才能确认I/O操作完全完成。
- I/O的同步与异步 I/O的同步与异步出现在应用程序中。如果做阻塞I/O调用，应用程序等待调用的完成的过程就是一种同步状况。相反，I/O为非阻塞模式时，应用程序则是异步的。

异步I/O与轮询技术

当进行非阻塞I/O调用时，要读到完整的数据，应用程序需要进行多次轮询，才能确保读取数据完成，以进行下一步的操作。 轮询技术的缺点在于应用程序要主动调用，会造成占用较多CPU时间片，性能较为低下。现存的轮询技术有以下这些：

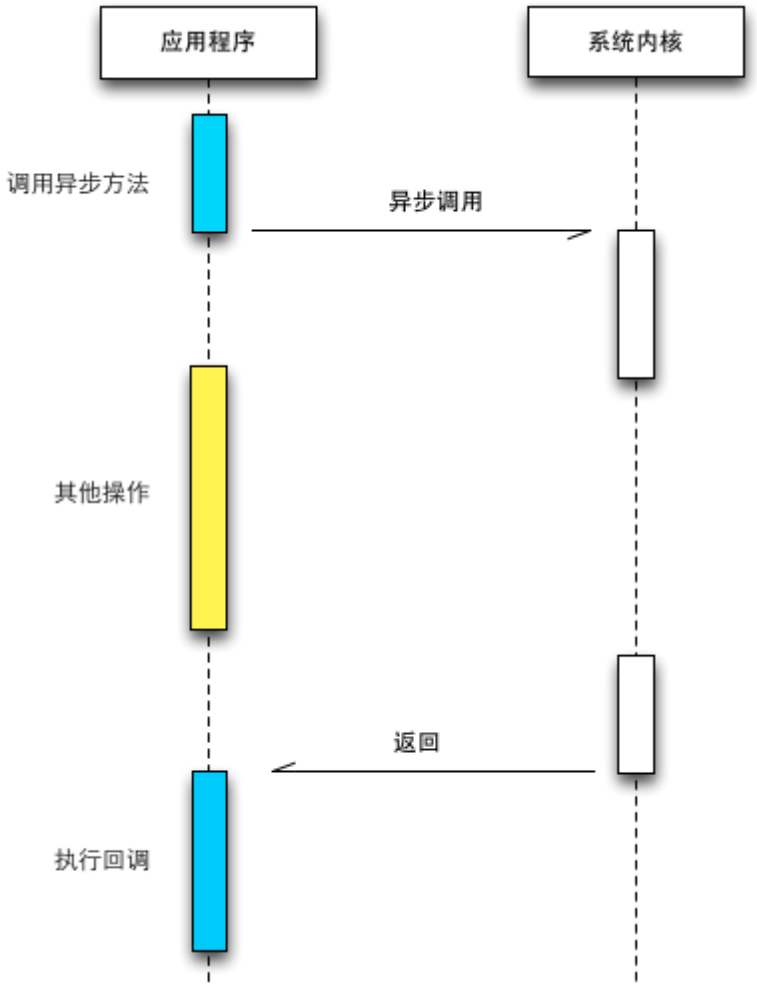
- read
- select
- poll
- epoll
- pselect
- kqueue

read是性能最低的一种，它通过重复调用来检查I/O的状态来完成完整数据读取。select是一种改进方案，通过对文件描述符上的事件状态来进行判断。操作系统还提供了poll、epoll等多路复用技术来提高性能。 轮询技术满足了异步I/O确保获取完整数据的保证。但是对于应用程序而言，它仍然只能算时一种同步，因为应用程序仍然需要主动去判断I/O的状态，依旧花费了很多CPU时间来等待。

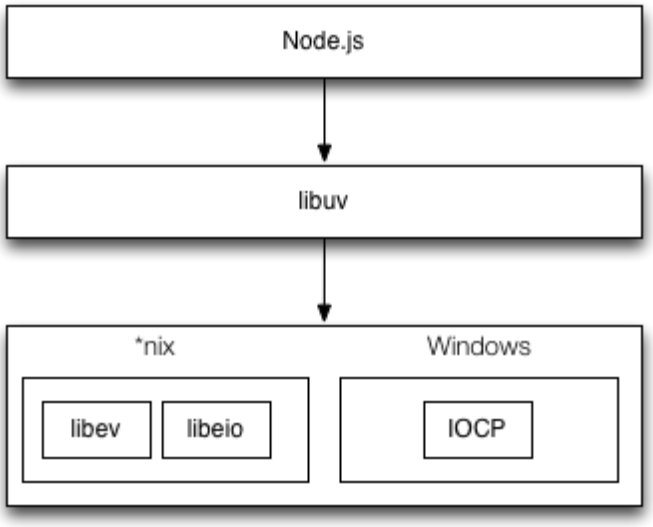
上一种方法重复调用read进行轮询直到最终成功，用户程序会占用较多CPU，性能较为低下。而实际上操作系统提供了select方法来代替这种重复read轮询进行状态判断。select内部通过检查文件描述符上的事件状态来进行判断数据是否完全读取。但是对于应用程序而言它仍然只能算是一种同步，因为应用程序仍然需要主动去判断I/O的状态，依旧花费了很多CPU时间等待，select也是一种轮询。

理想的异步I/O模型

理想的异步I/O应该是应用程序发起异步调用，而不需要进行轮询，进而处理下一个任务，只需在I/O完成后通过信号或是回调将数据传递给应用程序即可。



幸运的是，在Linux下存在一种这种方式，它原生提供了一种异步非阻塞I/O方式（AIO）即是通过信号或回调来传递数据的。 不幸的是，只有Linux下有这么一种支持，而且还有缺陷（AIO仅支持内核I/O中的O_DIRECT方式读取，导致无法利用系统缓存。参见：http://forum.nginx.org/read.php?2,113524,113587#msg-113587 以上都是基于非阻塞I/O进行的设定。另一种理想的异步I/O是采用阻塞I/O，但加入多线程，将I/O操作分到多个线程上，利用线程之间的通信来模拟异步。Glibc的AIO便是这样的典型http://www.ibm.com/developerworks/linux/library/l-async/。然而遗憾在于，它存在一些难以忍受的缺陷和bug。可以简单的概述为：Linux平台下没有完美的异步I/O支持。 所幸的是，libev的作者Marc Alexander Lehmann重新实现了一个异步I/O的库：libeio。libeio实质依然是采用线程池与阻塞I/O模拟出来的异步I/O。 那么在Windows平台下的状况如何呢？而实际上，Windows有一种独有的内核异步IO方案：IOCP。IOCP的思路是真正的异步I/O方案，调用异步方法，然后等待I/O完成通知。IOCP内部依旧是通过线程实现，不同在于这些线程由系统内核接手管理。IOCP的异步模型与Node.js的异步调用模型已经十分近似。 以上两种方案则正是Node.js选择的异步I/O方案。由于Windows平台和*nix平台的差异，Node.js提供了libuv来作为抽象封装层，使得所有平台兼容性的判断都由这一层次来完成，保证上层的Node.js与下层的libeio/libev及IOCP之间各自独立。Node.js在编译期间会判断平台条件，选择性编译unix目录或是win目录下的源文件到目标程序中。



下文我们将通过解释Windows下Node.js异步I/O（IOCP）的简单例子来探寻一下从JavaScript代码到系统内核之间都发生了什么。

Node.js的异步I/O模型

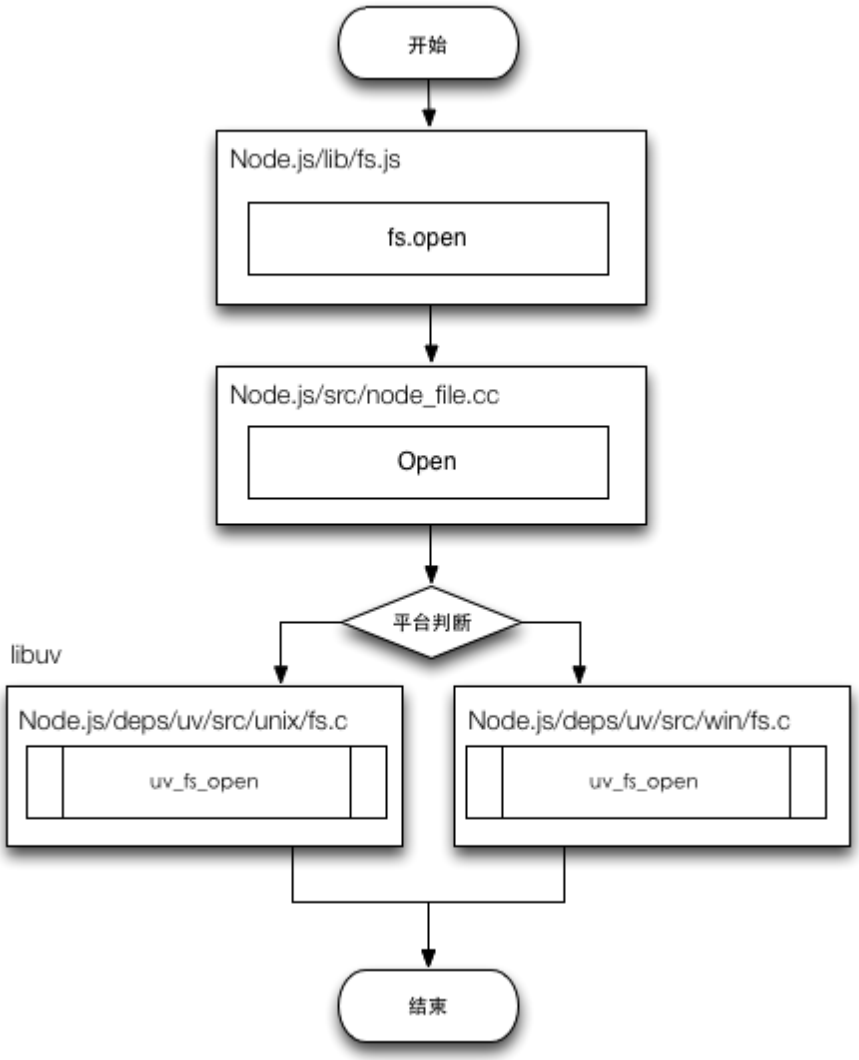
很多同学在遇见Node.js后必然产生过对回调函数究竟如何被调用产生过好奇。在文件I/O这一块与普通的业务逻辑的回调函数不同在于它不是由我们自己的代码所触发，而是系统调用结束后，由系统触发的。下面我们以最简单的fs.open方法来作为例子，探索Node.js与底层之间是如何执行异步I/O调用和回调函数究竟是如何被调用执行的。

```
fs.open = function(path, flags, mode, callback) {
  callback = arguments[arguments.length - 1];
  if (typeof(callback) !== 'function') {
    callback = noop;
  }

  mode = modeNum(mode, 438 /*=0666*/);

  binding.open(pathModule._makeLong(path),
               stringToFlags(flags),
               mode,
               callback);
};
```

fs.open的作用是根据指定路径和参数，去打开一个文件，从而得到一个文件描述符，是后续所有I/O操作的初始操作。



在JavaScript层面上调用的fs.open方法最终都透过node_file.cc调用到了libuv中的uv_fs_open方法，这里libuv作为封装层，分别写了两个平台下的代码实现，编译之后，只会存在一种实现被调用。

请求对象

在uv_fs_open的调用过程中，Node.js创建了一个FSReqWrap请求对象。从JavaScript传入的参数和当前方法都被封装在这个请求对象中，其中回调函数则被设置在这个对象的oncomplete_sym属性上。

```
req_wrap->object_->Set(oncomplete_sym, callback);
```

对象包装完毕后，调用QueueUserWorkItem方法将这个FSReqWrap对象推入线程池中等待执行。

```
QueueUserWorkItem(&uv_fs_thread_proc, req, WT_EXECUTELONGFUNCTION)
```

QueueUserWorkItem接受三个参数，第一个是要执行的方法，第二个是方法的上下文，第三个是执行的标志。当线程池中有可用线程的时候调用uv_fs_thread_proc方法执行。该方法会根据传入的类型调用相应的底层函数，以uv_fs_open为例，实际会调用到fs__open方法。调用完毕之后，会将获取的结果设置在req->result上。然后调用PostQueuedCompletionStatus通知我们的IOCP对象操作已经完成。

```
PostQueuedCompletionStatus((loop)->iocp, 0, 0, &((req)->overlapped))
```

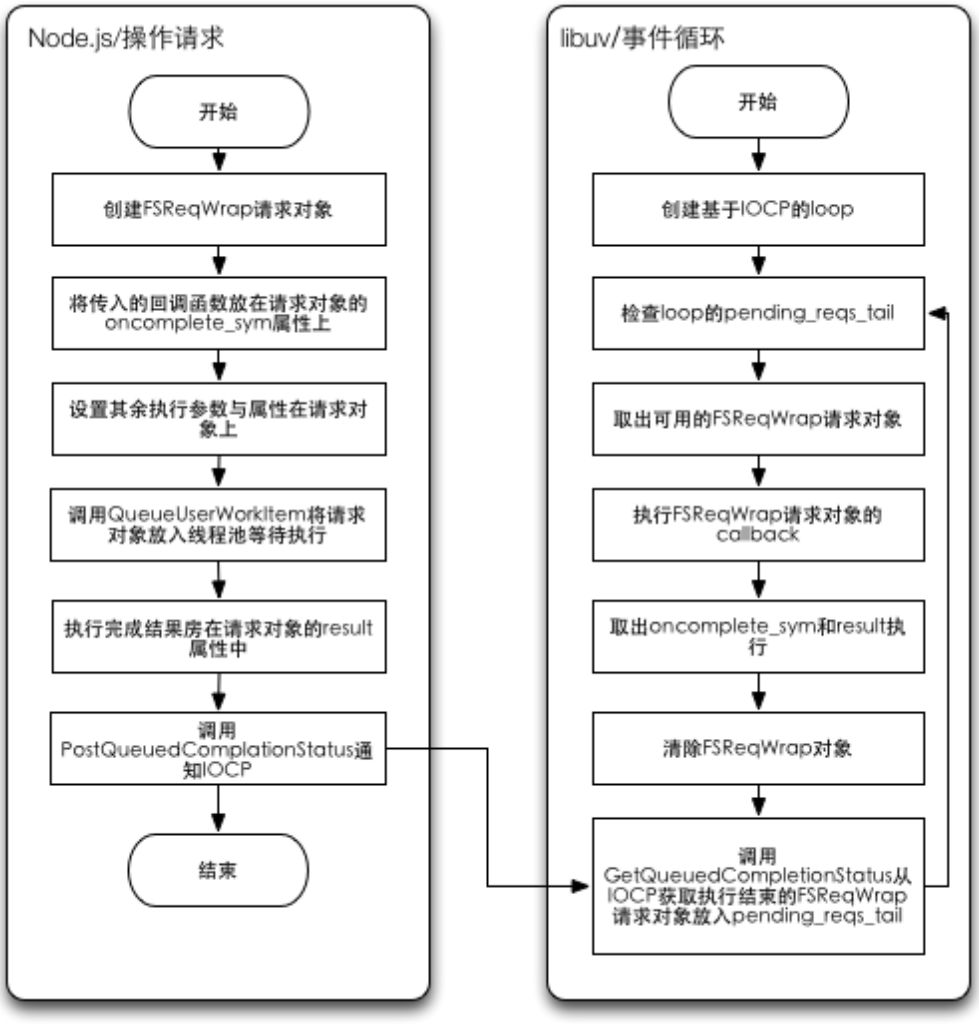
PostQueuedCompletionStatus方法的作用是向创建的IOCP上相关的线程通信，线程根据执行状况和传入的参数判定退出。 至此，由JavaScript层面发起的异步调用第一阶段就此结束。

事件循环

在调用uv_fs_open方法的过程中实际上应用到了事件循环。以在Windows平台下的实现中，启动Node.js时，便创建了一个基于IOCP的事件循环loop，并一直处于执行状态。

```
uv_run(uv_default_loop());
```

每次循环中，它会调用IOCP相关的GetQueuedCompletionStatus方法检查是否线程池中有执行完的请求，如果存在，poll操作会将请求对象加入到loop的pending_reqs_tail属性上。 另一边这个循环也会不断检查loop对象上的pending_reqs_tail引用，如果有可用的请求对象，就取出请求对象的result属性作为结果传递给oncomplete_sym执行，以此达到调用JavaScript中传入的回调函数的目的。 至此，整个异步I/O的流程完成结束。其流程如下：



事件循环和请求对象构成了Node.js的异步I/O模型的两个基本元素，这也是典型的消费者生产者场景。在Windows下通过IOCP的GetQueuedCompletionStatus、PostQueuedCompletionStatus、QueueUserWorkItem方法与事件循环实。对于*nix平台下，这个流程的不同之处在与实现这些功能的方法是由libeio和libev提供。

参考：

- 《nodejs异步IO的实现》http://cnodejs.org/blog/?p=244
- 《linux AIO （异步IO）那点事儿》http://cnodejs.org/blog/?p=2426
- 《libev 设计分析》http://cnodejs.org/blog/?p=2489

- 《Node Roadmap》http://nodejs.org/nodeconf.pdf
- 《多路复用select(2)与事件通知poll(2)、epoll(7)内核源码初探》http://blog.dccmx.com/2011/04/select-poll-epoll-in-kernel/
- 《使用异步 I/O 大大提高应用程序的性能》http://www.ibm.com/developerworks/cn/linux/l-async/

（六）Buffer那些事儿

作为前端的JSer，是一件非常幸福的事情，因为在字符串上从来没有出现过任何纠结的问题。我们来看看PHP对字符串长度的判断结果：

```
<?php
echo strlen("0123456789");
echo strlen("零一二三四五六七八九");
echo mb_strlen("零一二三四五六七八九", "utf-8");
echo "\n";
```

以上三行判断分别返回10、30、10。对于中国人而言，strlen这个方法对于Unicode的判断结果是非常让人疑惑。而看看JavaScript中对字符串长度的判断，就知道这个length属性对调用者而言是多么友好。

```
console.log("0123456789".length); // 10
console.log("零一二三四五六七八九".length); /10
console.log("\u00bd".length); // 1
```

尽管在计算机内部，一个中文字和一个英文字占用的字节位数是不同的，但对于用户而言，它们拥有相同的长度。我认为这是JavaScript中 String处理得精彩的一个点。正是由于这个原因，所有的数据从后端传输到前端被调用时，都是这般友好的字符串。所以对于前端工程师而言，他们是没有字 符串Buffer的概念的。如果你是一名前端工程师，那么从此在与Node.js打交道的过程中，一定要小心Buffer啦，因为它比传统的String 要调皮一点。

你该小心Buffer啦

像许多计算机的技术一样，都是从国外传播过来的。那些以英文作为母语的传道者们应该没有考虑过英文以外的使用者，所以你有可能看到如下这样一段代码在向你描述如何在data事件中连接字符串。

```
var fs = require('fs');
var rs = fs.createReadStream('testdata.md');
var data = '';
rs.on("data", function (trunk){
    data += trunk;
});
rs.on("end", function () {
    console.log(data);
});
```

如果这个文件读取流读取的是一个纯英文的文件，这段代码是能够正常输出的。但是如果我们再改变一下条件，将每次读取的buffer大小变成一个奇数，以模拟一个字符被分配在两个trunk中的场景。

```
var rs = fs.createReadStream('testdata.md', {bufferSize: 11});
```

我们将会得到以下这样的乱码输出：

```
事件循���和请求���象构成了Node.js���异步I/O模型的���个基本���素，这也是典���的消费���生产者场景。
```

造成这个问题的根源在于data += trunk语句里隐藏的错误，在默认的情况下，trunk是一个Buffer对象。这句话的实质是隐藏了toString的变换的：

```
data = data.toString() + trunk.toString();
```

由于汉字不是用一个字节来存储的，导致有被截破的汉字的存在，于是出现乱码。解决这个问题有一个简单的方案，是设置编码集：

```
var rs = fs.createReadStream('testdata.md', {encoding: 'utf-8', bufferSize: 11});
```

这将得到一个正常的字符串响应：

```
事件循环和请求对象构成了Node.js的异步I/O模型的两个基本元素，这也是典型的消费者生产者场景。
```

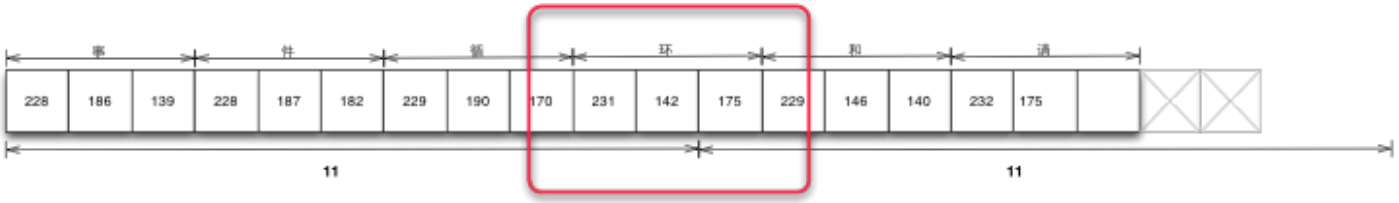
遗憾的是目前Node.js仅支持hex、utf8、ascii、binary、base64、ucs2几种编码的转换。对于那些因为历史遗留问题依旧还生存着的GBK，GB2312等编码，该方法是无能为力的。

有趣的string_decoder

在这个例子中，如果仔细观察，会发现一件有趣的事情发生在设置编码集之后。我们提到data += trunk等价于data = data.toString() + trunk.toString()。通过以下的代码可以测试到一个汉字占用三个字节，而我们按11个字节来截取trunk的话，依旧会存在一个汉字被分割在两个trunk中的情景。

```
console.log("事件循环和请求对象".length);
console.log(new Buffer("事件循环和请求对象").length);
```

按照猜想的toString()方式，应该返回的是事件循xxx和请求xxx象才对，其中“环”字应该变成乱码才对，但是在设置了encoding（默认的utf8）之后，结果却正常显示了，这个结果十分有趣。



在好奇心的驱使下可以探查到data事件调用了string_decoder来进行编码补足的行为。通过string_decoder对象输出第一个截取Buffer(事件循xx)时，只返回事件循这个字符串，保留xx。第二次通过string_decoder对象输出时检测到上次保留的xx，将上次剩余内容和本次的Buffer进行重新拼接输出。于是达到正常输出的目的。

string_decoder，目前在文件流读取和网络流读取中都有应用到，一定程度上避免了粗鲁拼接trunk导致的乱码错

误。但是，遗憾在于string_decoder目前只支持utf8编码。它的思路其实还可以扩展到其他编码上，只是最终是否会支持目前尚不可得知。

连接Buffer对象的正确方法

那么万能的适应各种编码而且正确的拼接Buffer对象的方法是什么呢？我们从Node.js在github上的源码中找出这样一段正确读取文件，并连接buffer对象的方法：

```
var buffers = [];
var nread = 0;
readStream.on('data', function (chunk) {
    buffers.push(chunk);
    nread += chunk.length;
});
readStream.on('end', function () {
    var buffer = null;
    switch(buffers.length) {
        case 0: buffer = new Buffer(0);
            break;
        case 1: buffer = buffers[0];
            break;
        default:
            buffer = new Buffer(nread);
            for (var i = 0, pos = 0, l = buffers.length; i < l; i++) {
                var chunk = buffers[i];
                chunk.copy(buffer, pos);
                pos += chunk.length;
            }
        break;
    }
});
```

在end事件中通过细腻的连接方式，最后拿到理想的Buffer对象。这时候无论是在支持的编码之间转换，还是在不支持的编码之间转换（利用iconv模块转换），都不会导致乱码。

简化连接Buffer对象的过程

上述一大段代码仅只完成了一件事情，就是连接多个Buffer对象，而这种场景需求将会在多个地方发生，所以，采用一种更优雅的方式来完成该过程是必要的。笔者基于以上的代码封装出一个bufferhelper模块，用于更简洁地处理Buffer对象。可以通过NPM进行安装：

```
npm install bufferhelper
```

下面的例子演示了如何调用这个模块。与传统data += trunk之间只是bufferHelper.concat(chunk)的差别，既避免了错误的出现，又使得代码可以得到简化而有效地编写。

```
var http = require('http');
var BufferHelper = require('bufferhelper');
http.createServer(function (request, response) {
    var bufferHelper = new BufferHelper();
    request.on("data", function (chunk) {
    bufferHelper.concat(chunk);
    });
    request.on('end', function () {
    var html = bufferHelper.toBuffer().toString();
    response.writeHead(200);
    response.end(html);
    });

}).listen(8001);
```

所以关于Buffer对象的操作的最佳实践是：

- 保持编码不变，以利于后续编码转换
- 使用封装方法达到简洁代码的目的

参考

- https://github.com/joyent/node/blob/master/lib/fs.js#L107
- https://github.com/JacksonTian/bufferhelper

（七）Connect模块解析（之一）

Connect模块背景

Node.js的愿望是成为一个能构建高速，可伸缩的网络应用的平台，它本身具有基于事件，异步，非阻塞，回调等特性，这在前几篇专栏中有过描述。正是基于这样的一些特性，Node.js平台上的Web框架也具有不同于其他平台的一些特性，其中Connect是众多Web框架中的佼佼者。 Connect在它的官方介绍中，它是Node的一个中间件框架。超过18个捆绑的中间件和一些精选第三方中间件。尽管Connect可能不是性能最好的Node.jsWeb框架，但它却几乎是最为流行的Web框架。为何Connect能在众多框架中胜出，其原因不外乎有如下几个：

- 模型简单
- 中间件易于组合和插拔
- 中间件易于定制和优化
- 丰富的中间件

Connect自身十分简单，其作用是基于Web服务器做中间件管理。至于如何如何处理网络请求，这些任务通过路由分派给管理的中间件们进行处理。它的处理模型仅仅只是一个中间队列，进行流式处理而已，流式处理可能性能不是最优，但是却是最易于被理解和接受。基于中间件可以自由组合和插拔的情况，优化它十分容易。 Connect模块目前在NPM仓库的MDO（被依赖最多的模块）排行第八位。但这并没有真实反映出它的价值，因为排行第五位的Express框架实际上是依赖Connect创建而成的。关于Express的介绍，将会在后续的专栏中一一为你讲解。

中间件

让我们回顾一下Node.js最简单的Web服务器是如何编写的：

```
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');
```

我们从最朴素的Web服务器处理流程开始，可以看到HTTP模块基于事件处理网络访问无外乎两个主要的因素，请求和响应。同理的是Connect的中间件也是扮演这样一个角色，处理请求，然后响应客户端或是让下一个中间件继续处理。如下是一个中间件最朴素的原型：

```
function (req, res, next) {
  // 中间件
}
```

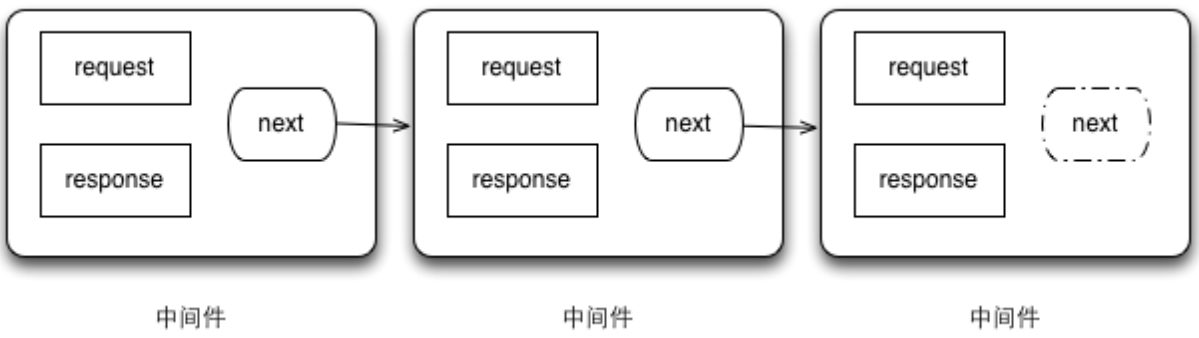
在中间件的上下文中，有着三个变量。分别代表请求对象、响应对象、下一个中间件。如果当前中间件调用了res.end()结束了响应，执行下一个中间件就显得没有必要。

流式处理

为了演示中间件的流式处理，我们可以看看中间件的使用形式：

```
var app = connect();
// Middleware
app.use(connect.staticCache());
app.use(connect.static(__dirname + '/public'));
app.use(connect.cookieParser());
app.use(connect.session());
app.use(connect.query());
app.use(connect.bodyParser());
app.use(connect.csrf());
app.use(function (req, res, next) {
  // 中间件
});
app.listen(3001);
```

Conncet提供use方法用于注册中间件到一个Connect对象的队列中，我们称该队列叫做中间件队列。



Conncet的部分核心代码如下，它通过use方法来维护一个中间件队列。然后在请求来临的时候，依次调用队列中的中间件，直到某个中间件不再调用下一个中间件为止。

```
app.stack = [];
app.use = function(route, fn){
  // ...

  // add the middleware
  debug('use %s %s', route || '/', fn.name || 'anonymous');
  this.stack.push({ route: route, handle: fn });

  return this;
};
```

值得注意的是，必须要有一个中间件调用res.end()方法来告知客户端请求已被处理完成，否则客户端将一直处于等待状态。 流式处理也是Node.js中用于流程控制的经典模式，Connect模块是典型的应用了它。流式处理的好处在于，每一个中间层的职责都是单一的，开发者通过这个模式可以将复杂的业务逻辑进行分解。

路由

从前文可以看到其实app.use()方法接受两个参数，route和fn，既路由信息和中间件函数，一个完整的中间件，其实包含路由信息和中间件函数。路由信息的作用是过滤不匹配的URL。请求在遇见路由信息不匹配时，直接传递给下一个中间件处理。 通常在调用app.use()注册中间件时，只需要传递一个中间件函数即可。实际上这个过程中，Connect会将/作为该中间件的默认路由，它表示所有的请求都会被该中间件处理。 中间件的优势类似于Java中的过滤器，能够全局性地处理一些事务，使得业务逻辑保持简单。 任何事物均有两面性，当你调用app.use()添加中间件的时候，需要考虑的是中间件队列是否太长，因为每一层中间件的调用都是会降低性能的。为了提高性能，在添加中间件的时候，如非全局需求的，尽量附带上精确的路由信息。 以multipart中间件为例，它用于处理表单提交的文件信息，相对而言较为耗费资源。它存在潜在的问题，那就是有可能被人在客户端恶意提交文件，造成服务器资源的浪费。如果不采用路由信息加以限制，那么任何URL都可以被攻击。

```
app.use("/upload", connect.multipart({ uploadDir: path }));
```

加上精确的路由信息后，可以将问题减小。

MVC目录

借助Connect可以自由定制中间件的优势，可以自行提升性能或是设计出适合自己需要的项目。Connect自身提供了路由功能，在此基础上，可以轻松搭建MVC模式的框架，以达到开发效率和执行效率的平衡。以下是笔者项目中采用的目录结构，清晰地划分目录结构可以帮助划分代码的职责，此处仅供参考。

```
├── Makefile // 构建文件，通常用于启动单元测试运行等操作
├── app.js // 应用文件
├── automation  // 自动化测试目录
├── bin  // 存放启动应用相关脚本的目录
├── conf  // 配置文件目录
├── controllers  // 控制层目录
├── helpers  // 帮助类库
├── middlewares  // 自定义中间件目录
├── models  // 数据层目录
├── node_modules  // 第三方模块目录
├── package.json  // 项目包描述文件
├── public  // 静态文件目录
│   ├── images  // 图片目录
│   ├── libs  // 第三方前端JavaScript库目录
│   ├── scripts  // 前端JavaScript脚本目录
│   └── styles  // 样式表目录
├── test  // 单元测试目录
└── views  // 视图层目录
```

参考：

- Connect主页 http://www.senchalabs.org/connect/
- NPM仓库 http://search.npmjs.org/

（八）Connect模块解析（之二）静态文件中间件

上一篇专栏简单介绍了Connect模块的基本架构，它的执行模型十分简单，中间件机制也使得它十分易于扩展，具备良好的可伸缩性。在Connect的良好机制下，我们本章开始将逐步解开Connect生态圈中中间件部分，这部分给予Connect良好的功能扩展。

静态文件中间件

也许你还记得我曾经写过的Node.js静态文件服务器实战，那篇文章中我叙述了如何利用Node.js实现一个静态文件服务器的许多技术细节，包括路由实现，MIME，缓存控制，传输压缩，安全、欢迎页、断点续传等。但是这里我们不需要去亲自处理细节，`Connect`的`static`中间件为我们提供上述所有功能。代码只需寥寥3行即可：

```
var connect = require('connect');
var app = connect();
app.use(connect.static(__dirname + '/public'));
```

在项目中需要临时搭建静态服务器，也无需安装apache之类的服务器，通过NPM安装Connect之后，三行代码即可解决需求。 这里需要提及的是在使用该模块的一点性能相关的细节。

动静分离

前一章提及，`app.use()`方法在没有指定路由信息时，相当于`app.use("/", middleware)`。这意味着静态文件中间件将会在处理所有路径的请求。在动静态请求混杂的场景下，静态中间件会在动态请求时也调用`fs.stat`来检测文件

系统是否存在静态文件。这造成了不必要的系统调用，使得性能降低。 解决影响性能的方法既是动静分离。利用路由检测，避免不必要的系统调用，可以有效降低对动态请求的性能影响。

```
app.use('/public', connect.static(__dirname + '/public'));
```

在大型的应用中，动静分离通常无需到一个Node.js实例中进行，CDN的方式直接在域名上将请求分离。小型应用中，适当的进行动静分离即可避免不必要的性能损耗。

缓存策略

缓存策略包含客户端和服务端两个部分。 客户端的缓存，主要是利用浏览器对HTTP协议响应头中`cache-control`和`expires`字段的支持。浏览器在得到明确的相应头后，会将文件缓存在本地，依据`cache-control`和`expires`的值进行相应的过期策略。这使得重复访问的过程中，浏览器可以从本地缓存中读取文件，而无需从网络读取文件，提升加载速度，也可以降低对服务器的压力。 默认情况下静态中间件的最大缓存时设置为0，意味着它在浏览器关闭后就被清除。这显然不是我们所期望的结果。除非是在开发环境可以无视`maxAge`的设置外，生产环境请务必设置缓存，因为它能有效节省网络带宽。

```
app.use('/public', connect.static(__dirname + '/public', {maxAge: 86400000}));
```

`maxAge`选项的单位为毫秒。YUI3的CDN服务器设置过期时间为10年，是一个值得参考的值。 静态文件如果在客户端被缓存，在需要清除缓存的时候，又该如何清除呢？这里的实现方法较多，一种较为推荐的做法是为文件进行md5处理。

```
http://some.url/some.js?md5
```

当文件内容产生改变时，md5值也将发生改变，浏览器根据URL的不同会重新获取静态文件。md5的方式可以避免不必要的缓存清除，也能精确清除缓存。 由于浏览器本身缓存容量的限制，尽管我们可能设置了10年的过期时间，但是也许两天之后就被新的静态文件挤出了本地缓存。这将持续引起静态服务器的响应，也即意味着，客户端缓存并不能完全解决降低服务器压力的问题。 为了解决静态服务器重复读取磁盘造成的压力，这里需要引出第二个相关的中间件：`staticCache`。

```
app.use(connect.staticCache());
app.use(“/public”, connect.static(__dirname + '/public', {maxAge: 86400000}));
```

这是一个提供上层缓存功能的中间件，能够将磁盘中的文件加载到内存中，以提高响应速度和提高性能。 它的官方测试数据如下：

```
static(): 2700 rps
node-static: 5300 rps
static() + staticCache(): 7500 rps
```

另一个专门用于静态文件托管的模块叫`node-static`，其性能是Connect静态文件中间件的效率的两倍。但是在缓存中间件的协助下，可以弥补性能损失。 事实上，这个中间件在生产环境下并不推荐被使用，而且它将在Connect 3.0版本中被移除。但是它的实现中有值得玩味的地方，这有助于我们认识Node.js模型的优缺点。 `staticCache`中间件有两个主要的选项：`maxObjects`和`maxLength`。代表的是能存储多少个文件和单个文件的最大尺寸，其默认值为128和256kb。为何会有这两个选项的设定，原因在于V8有内存限制的原因，作为缓存，如果没有良好的过期策略，缓存将会无限增加，直到内存溢出。设置存储数量和单个文件大小后，可以有效抑制缓存区的大小。 事实上，该缓存还存在的缺陷是单机情况下，通常为了有效利用CPU，Node.js实例并不只有一个，多个实例进程之间将

会存在冗余的缓存占用，这对于内存使用而言是浪费的。 除此之外，V8的垃圾回收机制是暂停JavaScript线程执行，通过扫描的方式决定是否回收对象。如果缓存对象过大，键太多，则扫描的时间会增加，会引起JavaScript响应业务逻辑的速度变慢。 但是这个模块并非没有存在的意义，上述提及的缺陷大多都是V8内存限制和Node.js单线程的原因。解决该问题的方式则变得明了。 **风险转移**是Node.js中常用于解决资源不足问题的方式，尤其是内存方面的问题。将缓存点，从Node.js实例进程中转移到第三方成熟的缓存中去即可。这可以保证：

1. 缓存内容不冗余。
2. 集中式缓存，减少不一致性的发生。
3. 缓存的算法更优秀以保持较高的命中率。
4. 让Node.js保持轻量，以解决它更擅长的问题。

Connect推荐服务器端缓存采用`varnish`这样的成熟缓存代理。而笔者目前的项目则是通过`Redis`来完成后端缓存的任务。

参考内容

- https://www.varnish-cache.org/releases
- http://www.senchalabs.org/connect/static.html
- http://www.senchalabs.org/connect/staticCache.html