深入浅出 Node.js

什么是 Node.js

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：

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：

```
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。

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。

Node.js&NPM 的安装与配置

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。
    a. Debian/Ubuntu 下的工具是 apt-get
    b. RedHat/centOS 下通过 yum 命令
    c. Mac OS X 下你可能需要安装 xcode 来获得编译器

相关厂商内容

Scala，让Java平台上的编程重现生机

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 <command>
where <command> 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。

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。

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。

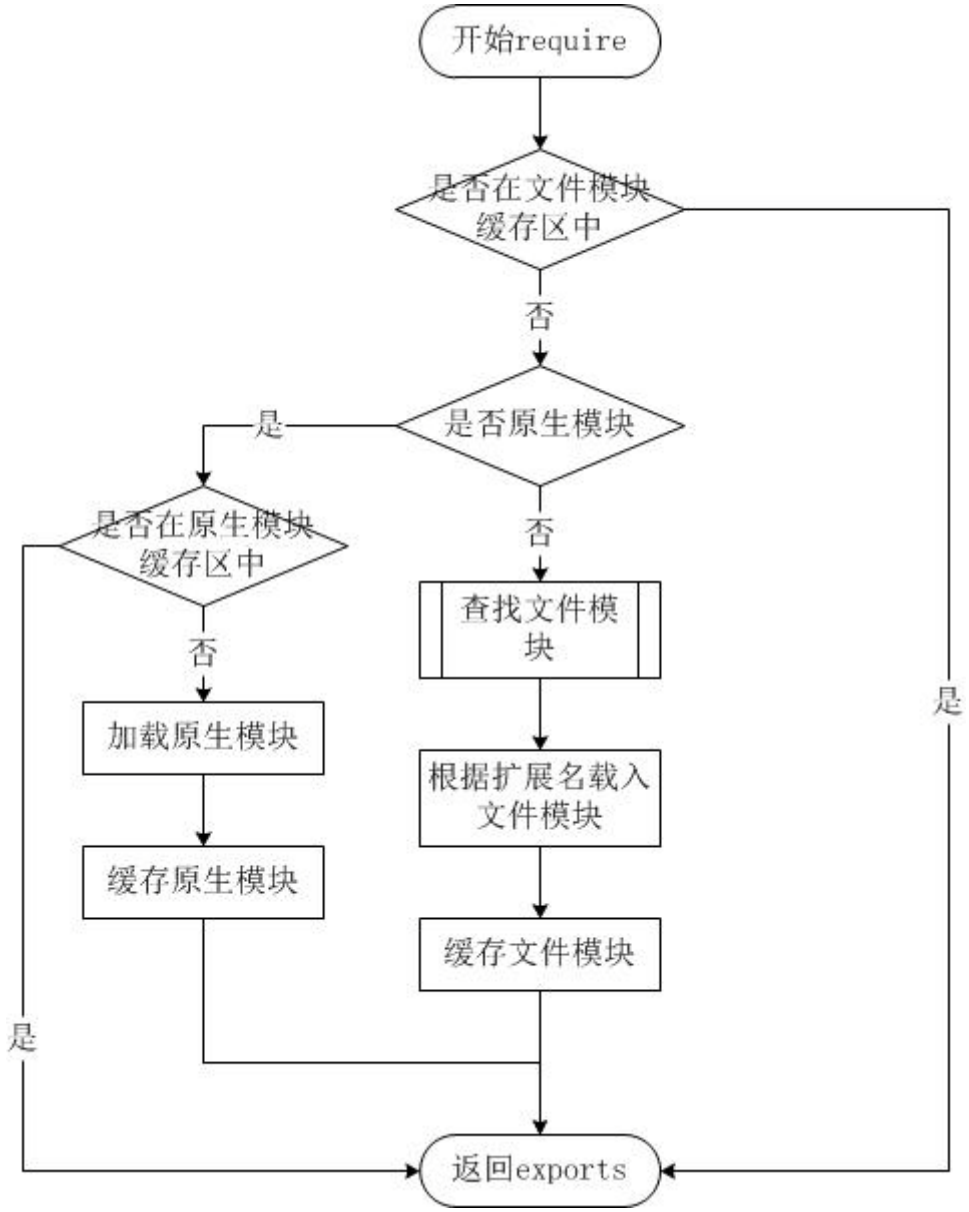
```
var status = "ready";
var select = function (callback) {
       if (status === "ready") {
           status = "pending";
           db.select("SQL", function (results) {
               callback(results);
               status = "ready";
           });
       }
   };
```

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。



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：

初探 Node.js 的异步 I/O 实现

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：

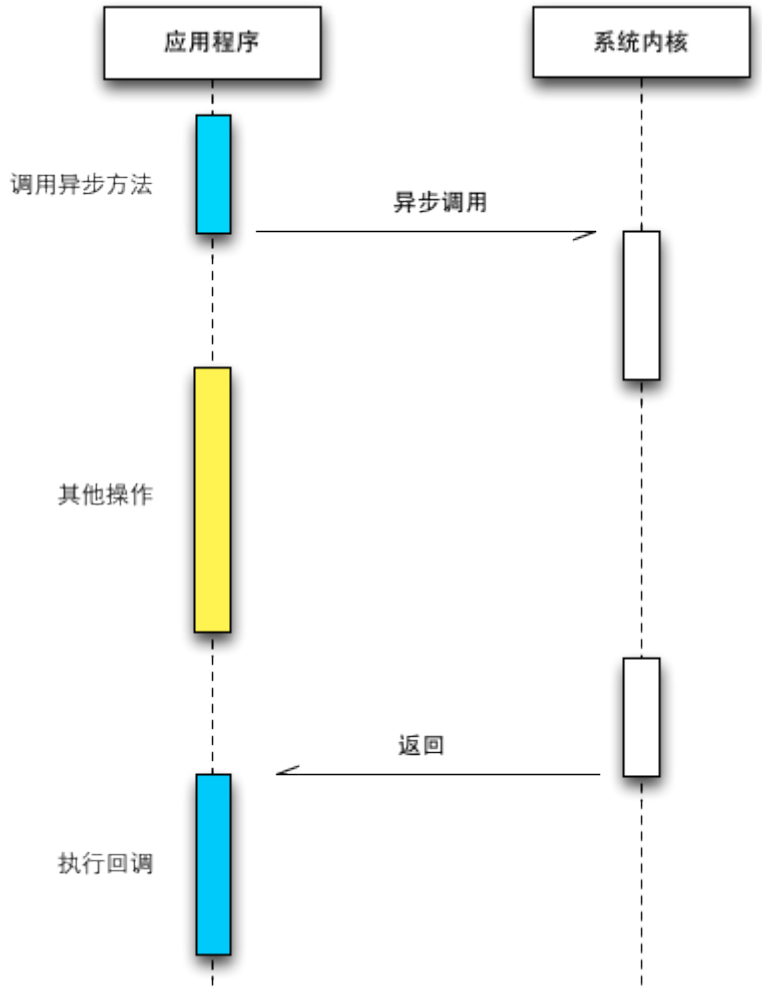
- read
- select
- poll
- epoll
- pselect
- kqueue

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。

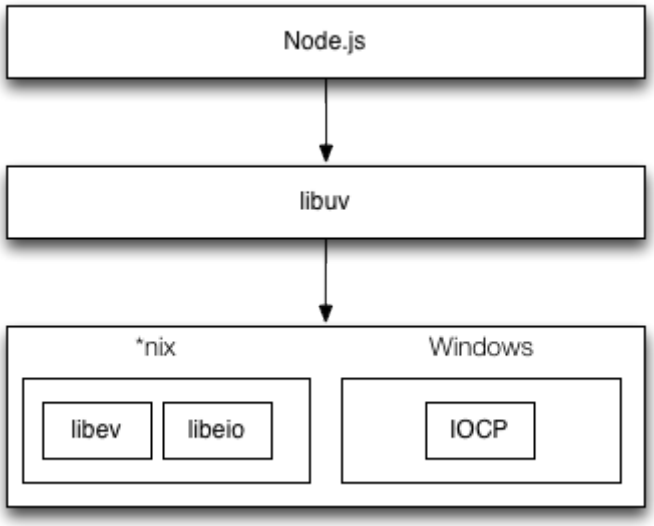
不幸的是，只有 Linux 下有这么一种支持，而且还有缺陷（AIO 仅支持内核 I/O 中的 O_DIRECT 方式读取，导致无法利用系统缓存。参见：http://forum.nginx.org/read.php?2,113524,113587#msg-113587

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

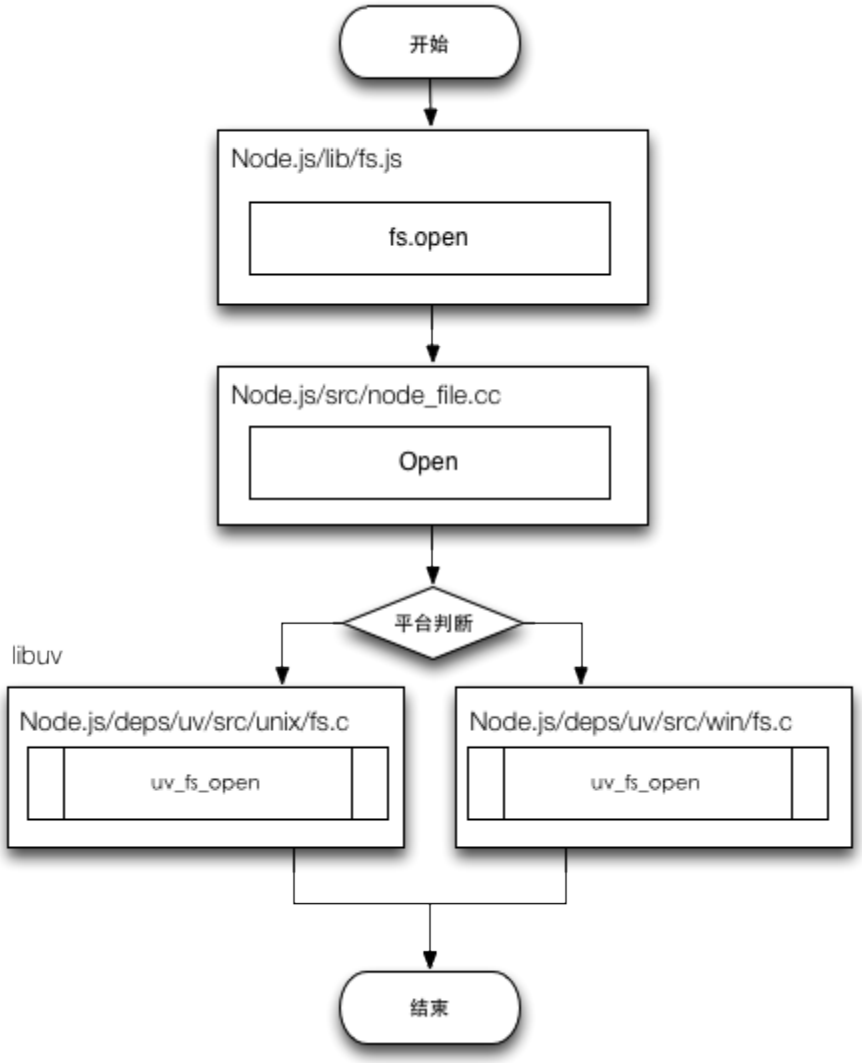
很多同学在遇见 Node.js 后必然产生过对回调函数究竟如何被调用产生过好奇。在文件 I/O 这一块与普通的业务逻辑的回调函数不同在于它不是由我们自己的代码所触发，而是系统调用结束后，由系统触发的。下面我们以最简单的 fs.open 方法来作为例子，探索 Node.js 与底层之间是如何执行异步 I/O 调用和回调函数究竟是如何被调用执行的。

```
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))
```

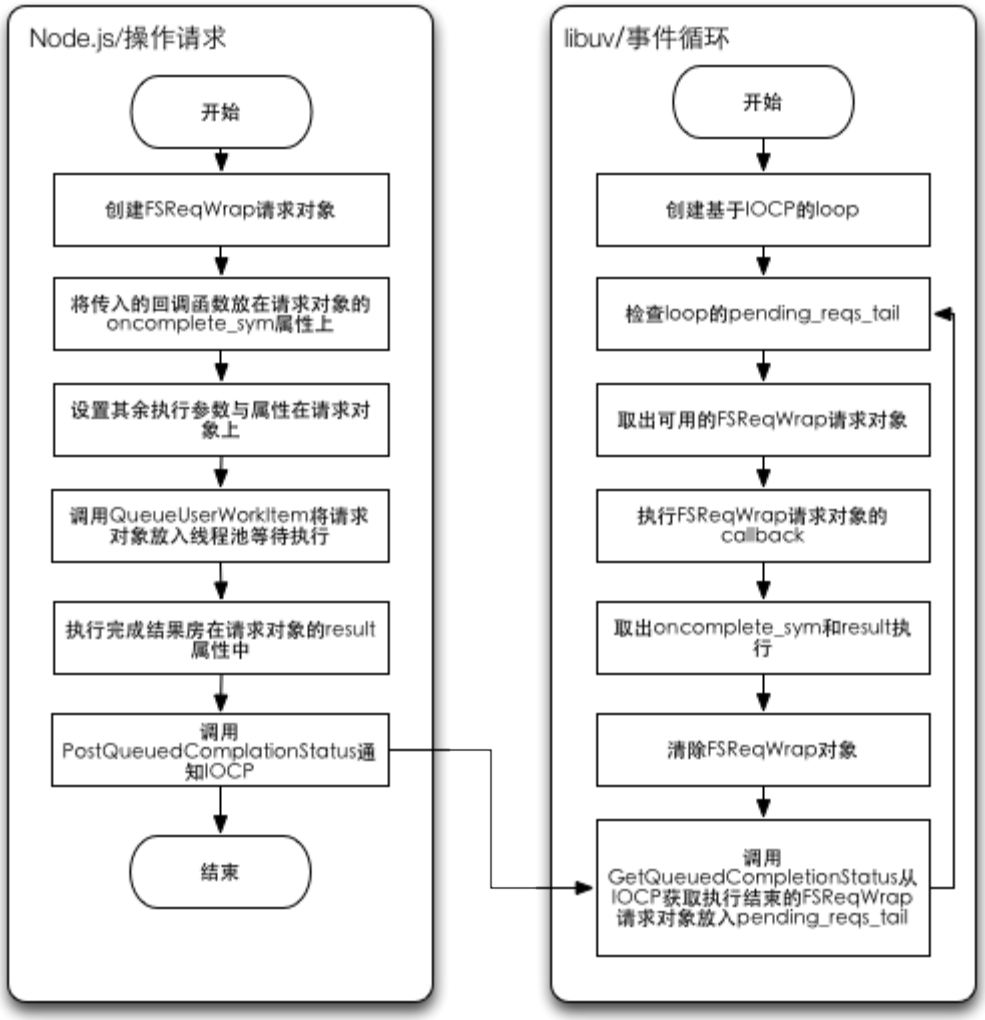
PostQueuedCompletionStatus 方法的作用是向创建的 IOCP 上相关的线程通信，线程根据执行状况和传入的参数判定退出。
至此，由 JavaScript 层面发起的异步调用第一阶段就此结束。

事件循环

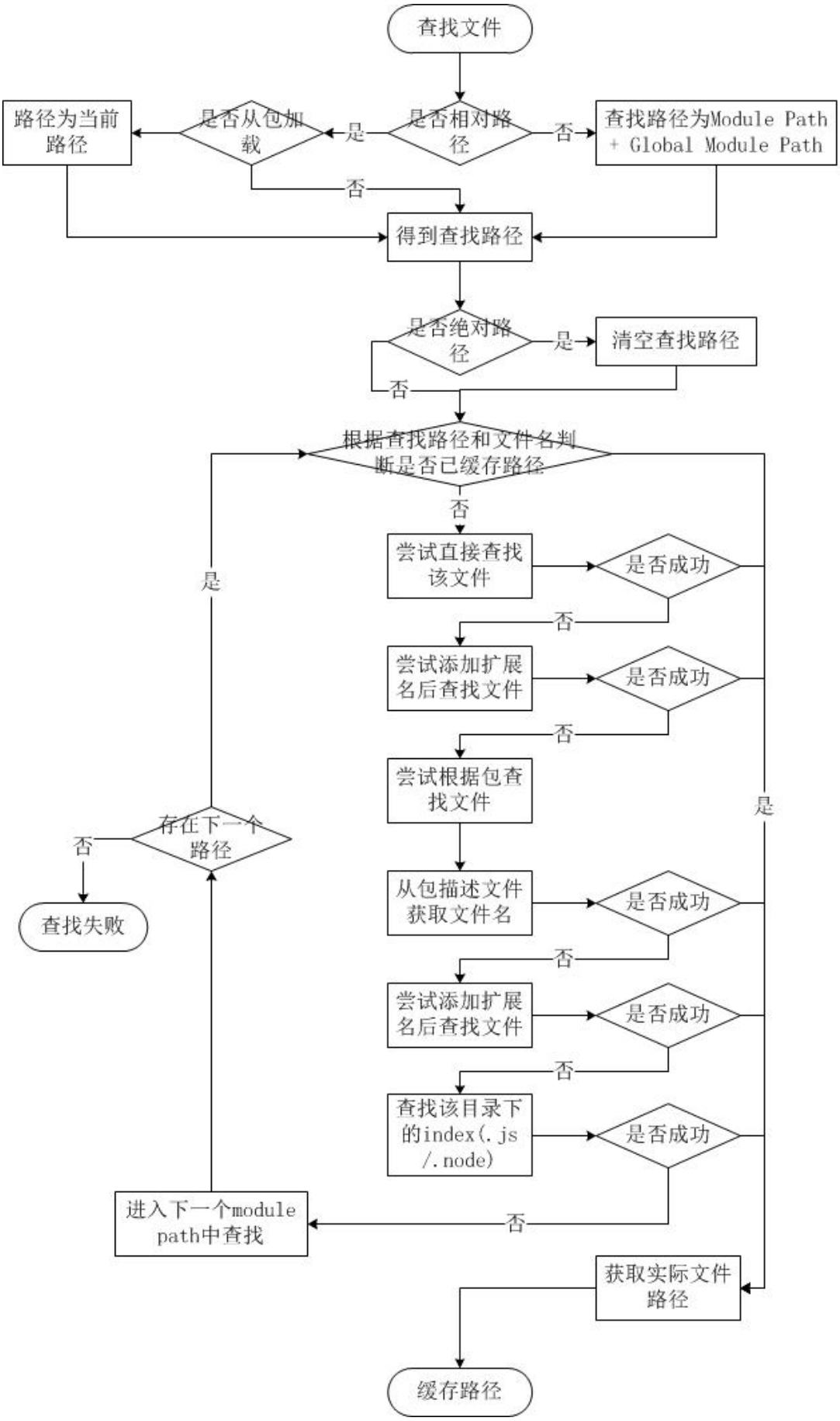
在调用 uv_fs_open 方法的过程中实际上应用到了事件循环。以在 Windows 平台下的实现中，启动 Node.js 时，便创建了一个基于 IOCP 的事件循环 loop，并一直处于执行状态。

```
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。


查找文件
是否相对路径
是
否
是否从包加载
路径为当前路径
查找路径为Module Path + Global Module Path
否
得到查找路径
是否绝对路径
是
清空查找路径
否
根据查找路径和文件名判断是否已缓存路径
否
尝试直接查找该文件
是否成功
否
尝试添加扩展名后查找文件
是否成功
否
尝试根据包查找文件
从包描述文件获取文件名
是否成功
否
尝试添加扩展名后查找文件
是否成功
否
查找该目录下的index(.js/.node)
是否成功
否
进入下一个module path中查找
存在下一个路径
否
查找失败
是
是
获取实际文件路径
缓存路径

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 <folder>
```

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 <package>
```

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 <package.json folder>
```

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。