

毕 业 设 计（论 文）

外 文 文 献 译 文 及 原 文

****

**学 生：**  **邵明娟**

**学 号： 201006080202**

**院 （系）： 电气与信息工程学院**

**专 业：**  **计算机科学与技术**

**指导教师：** **李思辉**

**2014** **年** **6**  **月** **12**  **日**

**Node.js:使用JavaScript构建高性能网络程序**

Node.js-又被叫做Node-是一种服务器端JavaScript运行环境（详见 <http://nodejs.org>）。它是基于谷歌开发的“V8”引擎运行实现的。V8和Node为了实现高性能和低内存消耗，主要使用C和C++实现。但是， V8支持的JavaScript主要是在浏览器环境下的（最值得注意的是，谷歌Chrome浏览器），而Node主要致力于支持长时间运行的服务器进程。

不同于大多数流行的运行环境，一个Node进程不需要依赖多线程去支持业务逻辑的一致性；它基于一个异步的I/O模型。可以把Node服务器端程序想象为嵌入在JavaScript引擎中的单线程Daemon，以此实现定制化。这与大多数通过库（函数库）来支持事件系统的其他编程语言不同：Node支持的事件模型是在语言层级的。

JavaScript非常适合这种方式，因为它支持事件回调。举例来说，当一个浏览器装载完毕一个文档，用户点击一个按钮，或者有一个Ajax请求，事件会触发一个回调。JavaScript函数原本就擅长创建匿名函数对象，这使得你极其容易创建一个匿名函数对象来注册事件处理。

**多线程VS事件**

应用程序开发人员需要处理多个I/O源（例如：网络服务器处理多个客户端连接）时，一直采用多线程编程技术。这项技术允许开发者把程序分到多个并行协作活动中，因此变得越来越流行。这不仅使程序逻辑更容易理解、实现和维护，而且也能够更快、更高效的运行。

对于需要执行大量 I/O 操作的应用程序（例如Web服务器）来说，多线程使应用程序能够更好地利用可用的处理器。在现代多核系统上运行多个并发线程是简单的，每个核心同时执行不同的线程实现真实的并行。在单核系统上，单个处理器执行一个线程，切换到另一个线程执行，以此类推。例如，当前的线程需要I/O操作（例如写TCP Socket）的时候处理器切换到另一个线程上。由于完成I/O操作会浪费很多处理器周期，所以要进行这样的切换。处理器与其浪费周期等待socket操作完成，不如任由I/O操作执行而去执行其他线程，以此保持繁忙状态做有用的工作。当I/O操作结束，处理器不再被I/O等待而阻塞后，再考虑让原来的线程变成等待执行状态。

尽管许多开发人员在生产应用中已成功地使用了多线程，但大多数人认为，多线程编程绝非易事。它在独立性和正确性上充满了问题，例如在线程之间死锁和保护共享资源失效。开发人员在多线程上也失去了某种程度的控制，因为哪个线程执行和执行多久通常是操作系统决定的。

事件驱动编程提供了一个更高效，可扩展的替代方案，使开发者能够更好地控制应用程序活动之间的切换。在这个模型中，应用程序依赖于事件通知机制，例如Unix系统中的select()和poll()系统调用，Linux系统中的epoll服务，以及BSD Unix分支（如OS X）中的kqueue和kevent调用。程序在确定的事件中注册interest（例如某个socket数据的读取）。当事件发生时，这个通知系统会通知应用程序处理事件。

异步I/O对于事件驱动编程非常重要，因为它可以防止应用程序在等待I/O操作时被阻塞。例如，如果应用程序要写入到一个socket 并占用它底层的缓冲区，通常在缓冲区可用之前socket会阻止应用程序的写入，这样就阻止了应用程序做其他任何有用的工作。但是如果socket不阻塞，那么它反而会通知应用程序现在不能写、等会儿再试。假设应用程序在该socket上已经注册了有事件通知系统的interest，由于socket有空的时候会通知它，所以它可以去做些别的事情。

有异步I/O的事件驱动编程和多线程编程一样，也是有问题的。其中一个问题是，并非所有的进程间通信方法都可以纳入我们前面提到的事件通知机制。例如，大多数操作系统上，两个应用程序通过共享内存通信，共享内存段不提供句柄或文件描述符，程序也就不能注册事件。在这样的情况下，开发人员必须使用一些备选方案，例如在写入共享内存的同时也写入管道或其他事件支持的机制。

另一个重要的问题是在某些编程语言编写的应用程序处理事件和异步I/O的复杂性源于不同的事件，需要在不同环境下执行不同的操作。程序通常采用回调函数来处理事件。在没有匿名函数和闭包的编程语言中，如C语言，开发人员必须为每个事件和每个上下文都编写独立的函数。要想确保函数在被调用去处理事件的时候都能访问它们所需的数据和上下文信息，是非常困难的。许多这样的应用程序最终被无头绪的代码和全局变量充满，无法理解、难以维护。

**不算古老的JavaScript**

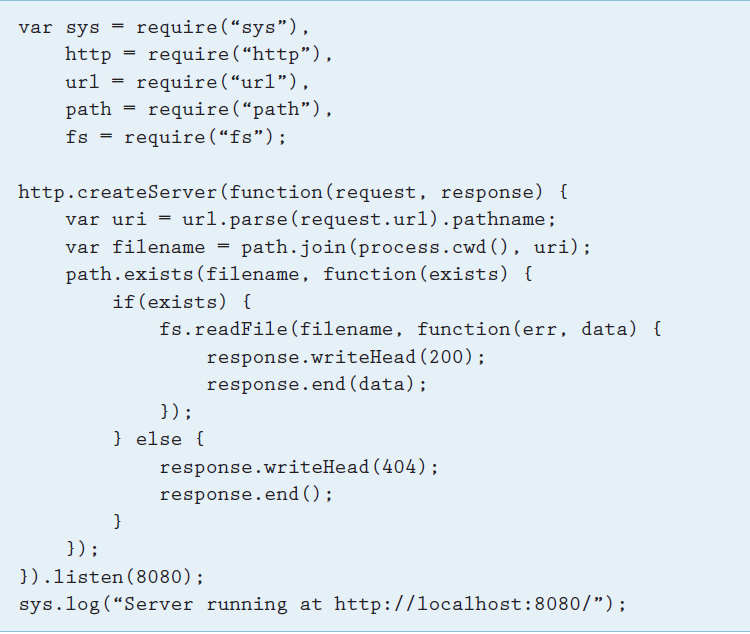
JavaScript作为一种编程语言，无论你怎么看待，毫无疑问，它已经成为任何现代的基于HTML的应用程序的核心要素。服务端JavaScript是一种合乎逻辑的进步，使这一个编程语言能使用在一个Web分布式应用程序的全部方面。这并不新奇，例如，Rhino JavaScript运行环境已经存在很长时间了。尽管如此，服务端的JavaScript不是一个主流的做法，直到最近才广泛流行起来。

我们相信事出有因。与HTML5相关的一组技术的出现降低了客户端平台的需求，迫切要求我们去了解和开发JavaScript，以获得创建富用户端界面的能力。NoSQL数据库（如CouchDB和Riak）使用JavaScript来定义数据视图和过滤条件。其他动态语言（如Ruby和Python等）已成为服务端开发可以接受的选择。最后，Mozilla和Google发布了高速、高扩展、高性能的JavaScript运行环境。

**Node的编程模型**

Node的I/O方式很严格：异步交互也不例外，这是规则。所有I/O操作都由高阶函数（以函数为参数的函数）处理，这些高阶函数指定了“做什么”和“什么时候做”。只在很少情况下Node开发者可以方便地使用同步工作函数，例如对文件的删除或重命名。但是一般情况下，在可能需要网络或文件的I/O调用，控制权会立即返回给调用者。当一些关键事件发生，例如网络socket可以读数据了、输出流可以写了、或发生错误了等，适当的回调函数将被调用。

图1是一个提供磁盘静态文件访问的HTTP Web服务器的简单实现例子。即使以前没做过Web开发，JavaScript的语法对以前接触过任何C类语言的人都是友好的。拿function(...)语法作为例子。它创建一个匿名函数：JavaScript是一种函数式语言，因此支持高阶函数。写或读Node程序的开发者会发现这到处都是。



*图1 一个简单的HTTP文件服务器。事件触发执行输入或输出操作的匿名函数。通过传入的请求触发服务器来解析目标URI，找到一个与该目标URI路径相匹配的本地文件，如果找到，读取文件内容，并写入对应的HTTP标头作为对客户端的响应。*

程序的主要流程就是显式的调用函数。这些函数不被I/O相关的事情阻塞，而是注册相应的回调处理。如果你在其他编程语言的事件库中看到类似的概念，你可能考虑防止循环阻塞的调用事件藏在哪里。事件循环概念藏在Node的实现中，这是Node的核心，主程序的目的就是简单的去建立合适的处理程序。http.createServer函数就是对一个低级的HTTP协议高效实现的封装，它只接收一个函数作为参数。当一个新的请求的数据可读时就会调用http.createServer。在其他环境中，同步地读文件然后返回，这种天真的实现毁了高效率。Node 不允许同步读取文件，只能通过 ReadFile 注册另一个函数，当数据可读时调用它。

**并行编程**

一个Node服务器的进程，通常调用命令行使用类似“node <文件名>”的单线程运行方式调用，但可以同时处理多个客户端。这似乎有些矛盾，但回想一下隐藏在代码中的主循环，循环中发生的只是一些注册调用循环内没有实际的I/O，更不用说业务逻辑处理。I/O相关的事件（例如建立一个连接，或从一个套接字、文件或外部系统接收或发送字节）触发实际的处理。



*图2.一个简单的流式HTTP文件服务器。从磁盘读文件，然后用HTTP的“chunked”传送编码发送给客户端*

图2是一个简化HTTP服务器的一个变体，它复杂了点，但也提供了更多。在此基础上，它接收一个HTTP请求的URL，并将URL的路径映射到服务器上的文件名。但是这次，该文件不再一次全部读出，而是一段一段的读取。在某些情况下，提供给脚本的函数作为回调函数被调用。文件系统准备好处理一定数目的字节时、文件被读取完时或者当发生某种错误时，这些都是上述场景的例子。如果数据可用，它就被写入HTTP输出流。Node强大的HTTP库支持HTTP 1.1的块传输编码。此外，读文件和写HTTP流都是异步的。

图2中的示例显示了如何轻松地开发可以建立一个高性能、异步、事件驱动、资源需求适度的网络服务器。这要归功于JavaScript的函数性，使得JavaScript支持事件回调。实际上，这种模式对于客户端JavaScript开发者是众所周知的事情。默认使用异步I/O迫使开发人员从一开始就接受异步模型。这是Node与其他异步I/O编程环境之间的主要区别之一。对其他编程环境而言，异步I/O只是众多选择之一，通常被认为太超前。

**多进程运行**

在多物理CPU或者多核环境下，并发执行不再是幻想而是现实。尽管操作系统可以有效分配Node的异步I/O交互，使它们和其他进程在系统上并发执行，但Node的进程仍然运行在单线程模式下，其核心业务逻辑从不并行执行。对于Node一般采用的解决方案是运行多个进程实例。

为了支持多进程运行，multi-node库（参见http://github.com/kris zyp/multi-node）补充了操作系统的进程间共享套接字的能力（并且它是用少于200行的Node的JavaScript代码实现的，现在Node官方已经有Cluster模块支持该功能）。例如，你可以通过调用运行HTTP服务器，如在图1和图2并行调用multi-node的listen()函数。这将启动多个进程，所有进程侦听相同的端口，有效地利用操作系统本身作为一个高效的负载均衡器。

**服务器端JavaScript生态系统**

Node 是支持服务器端JavaScript开发的框架和环境中非常著名的一个。Node社区已建立了一个为Node提供包或兼容Node的完整生态系统。其中诸如node-mysql或node-couchdb等工具已经发挥出很重要的作用，它们为关系数据库或NoSQL数据存储提供了异步交互的支持。许多框架提供了全功能的Web栈，例如Connect和Express，这和Ruby世界的规范性产物Rack和Rails是相似的，他们是如此地流行。Node的包管理工具npm能安装各种库和他们的依赖。最后，许多提供给客户端JavaScript的库被写入CommonJS模块，也能在Node中工作。一个令人印象深刻的模块列表是http://github.com/ ry/node/wiki/modules。

因此，在大多数Web开发项目中，JavaScript的知识是一种先进的UI交互的先决条件，选择使用一种编程语言完成一切，变得相当诱人。Node.js的框架可以在不牺牲性能和编程主流风格基础上，很容易地使用一个极具表现力的功能性语言进行服务器编程。

**Node.js: Using JavaScript to Build High-Performance Network Programs**

Node.js — also called Node — is a server-side JavaScript environment (see http:// nodejs.org). It’s based on Google’s runtime implementation — the aptly named “V8” engine. V8 and Node are mostly implemented in C and C++, focusing on performance and low memory consumption. But, whereas V8 supports mainly JavaScript in the browser (most notably, Google Chrome), Node aims to support long-running server processes.

Unlike in most other modern environments, a Node process doesn’t rely on multithreading to support concurrent execution of business logic; it’s based on an asynchronous I/O eventing model. Think of the Node server process as a single-threaded daemon that embeds the JavaScript engine to support customization. This is different from most eventing systems for other programming languages, which come in the form of libraries: Node supports the eventing model at the language level.

JavaScript is an excellent fit for this approach because it supports event callbacks. For exam­ple, when a browser completely loads a docu­ment, a user clicks a button, or an Ajax request is fulfilled, an event triggers a callback. JavaScript’s functional nature makes it extremely easy to create anonymous function objects that you can register as event handlers.

**Multithreading versus Events**

Application developers who deal with multiple I/O sources, such as networked servers handling multiple client connections, have long employed multithreaded programming techniques. Such techniques became popular because they let developers divide their applications into con­current cooperating activities. This promised to not only make program logic easier to under­stand, implement, and maintain but also enable faster, more efficient execution.

For applications such as Web servers performing significant amounts of I/O, multiple threads enable applications to better use available processors. Running multiple concurrent threads on a modern multicore system is straightforward, with each core simultaneously executing a different thread with true parallelism. On single-core systems, the single processor executes one thread, switches to another and executes it, and so on. For example, the processor switches its execution context to another thread when the current thread performs an I/O operation, such as writing to a TCP socket. The switch occurs because completing that operation can take many processor cycles. Rather than wasting cycles waiting for the socket operation to finish, the processor sets the I/O operation in motion and executes another thread, thus keeping itself busy doing useful work. When the I/O operation ends, the processor again considers the original thread to be ready to execute because it’s no longer blocked while waiting for I/O.

Even though many developers have suc­cessfully used multithreading in production applications, most agree that multithreaded programming is anything but easy. It’s fraught with problems that can be difficult to isolate and correct, such as deadlock and failure to protect resources shared among threads. Developers also lose some degree of control when draw­ing on multithreading because the OS typically decides which thread executes and for how long.

Event-driven programming offers a more efficient, scalable alternative that provides devel­opers much more control over switching between application activities. In this model, the applica­tion relies on event notification facilities such as the **select()** and **poll()** Unix sys­tem calls, the Linux **epoll** service, and the **kqueue** and **kevent** calls available in BSD Unix variants such as OS X. Applications register interest in certain events, such as data being ready to read on a particular socket. When the event occurs, the notifica­tion system notifies the application so that it can handle the event.

Asynchronous I/O is important for event-driven programming because it prevents the application from get­ting blocked while waiting in an I/O operation. For example, if the appli­cation writes to a socket and fills the socket’s underlying buffer, ordinar­ily, the socket blocks the application’s writes until buffer space becomes available, thus preventing the appli­cation from doing any other useful work. But, if the socket is nonblock­ing, it instead returns an indication to the application that further writ­ing isn’t currently possible, thereby informing the application that it should try again later. Assuming the application has registered interest with the event notification system in that socket, it can go do something else, knowing that it will receive an event when the socket’s write buffer has available space.

Like multithreaded program­ming, event-driven programming with asynchronous I/O can be problematic. One problem is that not all interprocess-communication approaches can be tied into the event notification facilities we mentioned earlier. For example, on most OSs, for two applications to communicate through shared memory, shared-memory segments provide no han­dles or file descriptors enabling the application to register for events. For such cases, developers must resort to alternatives such as writing to a pipe or some other event-capable mechanism together with writing to shared memory.

Another significant problem is the sheer complexity of writing applications in certain programming languages to deal with events and asynchronous I/O. This is because different events require different actions in different contexts. Pro­grams typically employ callback functions to deal with events. In languages that lack anonymous functions and closures, such as C, developers must write individual functions specifically for each event and event context. Ensuring that these functions all have access to the data and context information they require when they’re called to handle an event can be incredibly perplexing. Many such applications end up being little more than impen­etrable, unmaintainable tangles of spaghetti code and global variables.

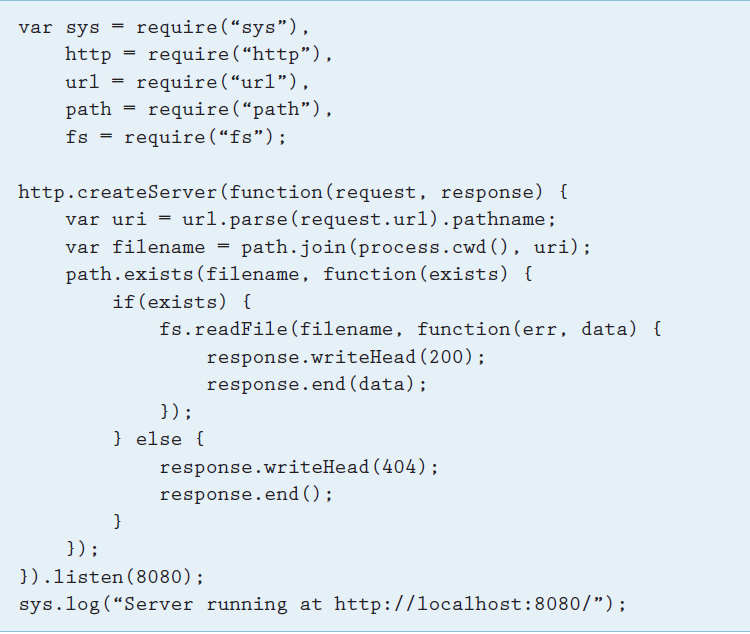
**Not Your Father’s JavaScrip**

Whatever you might think about JavaScript as a programming lan­guage, there’s little to no doubt it has become a central element of any modern HTML-based applica­tion. Server-side JavaScript is a logi­cal next step, enabling the use of a single programming language for all aspects of a Web-based distributed application. This idea isn’t new — for example, the Rhino JavaScript exe­cution environment has been avail­able for a long time. Still, server-side JavaScript isn’t yet a mainstream approach and has only recently gained massive popularity.

We believe that a number of factors have led to this effect. The advent of the set of technologies col­lectively labeled “HTML 5” reduces the appeal of alternative client-side platforms, enforcing the need to get to know and exploit JavaScript to create rich user interfaces. NoSQL-type databases such as CouchDB and Riak use JavaScript to define data views and filter criteria. Other dynamic languages, such as Ruby and Python, have become acceptable choices for server-side development. Finally, both Mozilla and Google have released high-performance JavaScript runtime implementations that are extremely fast and scalable.

**The Node Programming Model**

Node’s I/O approach is strict: asyn­chronous interactions aren’t the exception; they’re the rule. Every I/O operation is handled by means of higher-order functions — that is, functions taking functions as a parameter — that specify what to do when there’s something to do. In only rare circumstances have Node’s developers added a convenience func­tion that works synchronously — for example, for removing or renaming files. But, generally, when opera­tions that might require network or file I/O are invoked, control is imme­diately returned to the caller. When something interesting happens — for example, if data becomes available for reading from a network socket, an output stream is ready for writing, or an error occurs — the appropriate callback function is called.



*Figure 1. A simple HTTP file server. Events trigger anonymous functions that execute input or output operations. Incoming requests trigger the server to parse the target URI, look for a local file matching the URI path, and, if found, read the file contents and write them along with appropriate HTTP headers as a response to the client.*

Figure 1 is a simple example of implementing an HTTP Web server that serves static files from disk. Even to non-Web developers, JavaScript’s syntax should be fairly obvi­ous for those with prior exposure to any C-like language. One of the more specific topics is the **function(...)** syntax. This creates an unnamed function: JavaScript is a functional language and, as such, supports higher-order functions. A developer writing or looking at a Node program will see these everywhere.

The program’s main flow is deter­mined by the functions that are explicitly called. These functions never block on anything I/O-related, but rather register appropriate han­dler callbacks. If you’ve seen a simi­lar concept in eventing libraries for other programming languages, you might wonder where the explicit blocking call to invoke the event loop hides. The event loop concept is so core to Node’s behavior that it’s hid­den in the implementation; the main program’s purpose is simply to set up appropriate handlers. The **http. createServer** function, which is a wrapper around a low-level efficient HTTP protocol implementation, is passed a function as the only argu­ment. This function is invoked when­ever data for a new request is ready to be read. In another environment, a naïve implementation might ruin the effect of eventing by synchro­nously reading a file and sending it back. Node offers no opportunity to read a file synchronously — the only option is to register another func­tion via **readFile** that gets invoked whenever data can be read.

**Concurrent Programming**

A node server process, usu­ally invoked from the command line using something like “**node <scriptname>**,” runs single-threaded, yet can serve many clients concurrently. This seems a contra­diction, but recall that there’s an implicit main loop around the code, and what’s actually happening in that loop is just a number of regis­tration calls. No actual I/O, let alone business-logic processing, happens in the loop body. I/O-related events trigger the actual processing, such as a connection being made or bytes being sent or received from a socket, file, or external system.

Figure 2 is a slightly more com­plex variant of the simplistic HTTP server, but it does a lot more. Again, it parses the URI from an HTTP request and maps the URI’s path component to a filename on the server. But this time, the file is read in smaller chunks rather than all at once. In certain situations, the func­tion provided for the scenario as a callback is invoked. Example situ­ations include when the file system layer is ready to hand a number of bytes to the application, when the file has been read completely, or when some kind of error occurs. If data is available, it’s written to the HTTP output stream. Node’s sophis­ticated HTTP library supports HTTP 1.1’s chunked transfer encoding. Again, both reading from the file and writing to the HTTP stream hap­pen asynchronously.

Figure 2 is a slightly more com­plex variant of the simplistic HTTP server, but it does a lot more. Again, it parses the URI from an HTTP request and maps the URI’s path component to a filename on the server. But this time, the file is read in smaller chunks rather than all at once. In certain situations, the func­tion provided for the scenario as a callback is invoked. Example situ­ations include when the file system layer is ready to hand a number of bytes to the application, when the file has been read completely, or when some kind of error occurs. If data is available, it’s written to the HTTP output stream. Node’s sophis­ticated HTTP library supports HTTP 1.1’s chunked transfer encoding. Again, both reading from the file and writing to the HTTP stream hap­pen asynchronously.

The example in Figure 2 shows how easily developers can build a high-performance, asynchronous, event-driven network server with modest resource requirements. The main reason is that JavaScript, owing to its functional nature, supports event callbacks. In fact, this pattern is well known to any client-side JavaScript developer. In addition, making asynchronous I/O the default forces developers to adopt the asynchro­nous model from the start. This is one of the main differences between Node and using asynchronous I/O in other programming environments, in which it’s only one of many options and is often considered too advanced.



*Figure 2. A simple streaming HTTP file server. Chunks of the file are read from disk and sent to the client using HTTP’s “chunked” transfer encoding.*

**Running Multiple Processes**

In hardware environments in which more than one physical CPU or core is available, parallel execution isn’t an illusion but a reality. Although the OS can efficiently schedule a Node process with its asynchronous I/O interactions in parallel with other processes running on the sys­tem, Node still runs in a single pro­cess and thus never executes its core business logic in parallel. The com­mon solution to this problem in the Node world is to run multiple pro­cess instances.

To support this, the multi-node library (see http://github.com/kris zyp/multi-node) leverages the OS’s capability of sharing sockets between processes (and is imple­mented in fewer than 200 lines of Node JavaScript). For example, you can run HTTP servers such as those in Figures 1 and 2 in paral­lel by invoking multi-node’s **lis­ten()** function. This starts multiple processes that all listen on the same port, effectively using the OS as an efficient load balancer.

**A Server-Side JavaScript Ecosystem**

Node is one of the better-known frameworks and environments that support server-side JavaScript devel­opment. The community has created a whole ecosystem of libraries for, or compatible with, Node. Among these, tools such as node-mysql or node-couchdb play an important role by supporting asynchronous interaction with relational and NoSQL data stores, respectively. Many frameworks pro­vide a full-featured Web stack, such as Connect and Express, which are comparable to Rack and Rails in the Ruby world in scope, if not (yet?) in popularity. The Node package man­ager, npm, enables installation of libraries and their dependencies. Finally, many libraries available for client-side JavaScript that were writ­ten to comply with the CommonJS module system also work with Node. An impressive list of modules avail­able for Node is at http://github.com/ ry/node/wiki/modules.

Given that, in most Web develop­ment projects, JavaScript knowl­edge is a prerequisite for advanced UI interactions, the option of using one programming language for everything becomes quite tempt­ing. Node.js ’ s architecture makes it easy to use a highly expressive, functional language for server pro­gramming, without sacrificing per­formance and stepping out of the programming mainstream.