NODE.JS 入门手册

关于

本书致力于教会你如何用 Node.js 来开发应用，过程中会传授你所有所需的“高级”JavaScript 知识。本书绝不是一本“Hello World”的教程。

状态

你正在阅读的已经是本书的最终版。因此，只有当进行错误更正以及针对新版本 Node.js 的改动进行对应的修正时，才会进行更新。

本书中的代码案例都在 Node.js 0.4.9 版本中测试过，可以正确工作。

读者对象

本书最适合与我有相似技术背景的读者： 至少对一门诸如 Ruby、Python、PHP 或者 Java 这样面向对象的语言有一定的经验；对 JavaScript 处于初学阶段，并且完全是一个 Node.js 的新手。

这里指的适合对其他编程语言有一定经验的开发者，意思是说，本书不会对诸如数据类型、变量、控制结构等等之类非常基础的概念作介绍。要读懂本书，这些基础的概念我都默认你已经会了。

然而，本书还是会对 JavaScript 中的函数和对象作详细介绍，因为它们与其他同类编程语言中的函数和对象有很大的不同。

本书结构

读完本书之后，你将完成一个完整的 web 应用，该应用允许用户浏览页面以及上传文件。

当然了，应用本身并没有什么了不起的，相比为了实现该功能书写的代码本身，我们更关注的是如何创建一个框架来对我们应用的不同模块进行干净地剥离。 是不是很玄乎？稍后你就明白了。

本书先从介绍在 Node.js 环境中进行 JavaScript 开发和在浏览器环境中进行 JavaScript 开发的差异开始。

紧接着，会带领大家完成一个最传统的“Hello World”应用，这也是最基础的 Node.js 应用。

最后，会和大家讨论如何设计一个“真正”完整的应用，剖析要完成该应用需要实现的不同模块，并一步一步介绍如何来实现这些模块。

可以确保的是，在这过程中，大家会学到 JavaScript 中一些高级的概念、如何使用它们以及为什么使用这些概念就可以实现而其他编程语言中同类的概念就无法实现。

该应用所有的源代码都可以通过 本书 Github 代码仓库.

目录

JavaScript 与 Node.js

JavaScript 与你

抛开技术，我们先来聊聊你以及你和 JavaScript 的关系。本章的主要目的是想让你看看，对你而言是否有必要继续阅读后续章节的内容。

如果你和我一样，那么你很早就开始利用 HTML 进行“开发”，正因如此，你接触到了这个叫 JavaScript 有趣的东西，而对于 JavaScript，你只会基本的操作——为 web 页面添加交互。

而你真正想要的是“干货”，你想要知道如何构建复杂的 web 站点 —— 于是，你学习了一种诸如 PHP、Ruby、Java 这样的编程语言，并开始书写“后端”代码。

与此同时，你还始终关注着 JavaScript，随着通过一些对 jQuery，Prototype 之类技术的介绍，你慢慢了解到了很多 JavaScript 中的进阶技能，同时也感受到了 JavaScript 绝非仅仅是 *window.open()* 那么简单。 .

不过，这些毕竟都是前端技术，尽管当想要增强页面的时候，使用 jQuery 总让你觉得很爽，但到最后，你顶多是个 JavaScript *用户*，而非 JavaScript *开发者*。

然后，出现了 Node.js，服务端的 JavaScript，这有多酷啊？

于是，你觉得是时候该重新拾起既熟悉又陌生的 JavaScript 了。但是别急，写 Node.js 应用是一件事情；理解为什么它们要以它们书写的这种方式来书写则意味着——你要懂 JavaScript。这次是玩真的了。

问题来了： 由于 JavaScript 真正意义上以两种，甚至可以说是三种形态存在（从中世纪 90 年代的作为对 DHTML 进行增强的小玩具，到像 jQuery 那样严格意义上的前端技术，一直到现在的服务端技术），因此，很难找

到一个“正确”的方式来学习 JavaScript，使得让你书写 Node.js 应用的时候感觉自己是在真正开发它而不仅仅是使用它。

因为这就是关键： 你本身已经是个有经验的开发者，你不想通过到处寻找各种解决方案（其中可能还有不正确的）来学习新的技术，你要确保自己是通过正确的方式来学习这项技术。

当然了，外面不乏很优秀的学习 JavaScript 的文章。但是，有的时候光靠那些文章是远远不够的。你需要的是指导。

本书的目标就是给你提供指导。

简短申明

业界有非常优秀的 JavaScript 程序员。而我并非其中一员。

我就是上一节中描述的那个我。我熟悉如何开发后端 web 应用，但是对“真正”的 JavaScript 以及 Node.js，我都只是新手。我也只是最近学习了一些 JavaScript 的高级概念，并没有实践经验。

因此，本书并不是一本“从入门到精通”的书，更像是一本“从初级入门到高级入门”的书。

如果成功的话，那么本书就是我当初开始学习 Node.js 最希望拥有的教程。

服务端 JavaScript

JavaScript 最早是运行在浏览器中，然而浏览器只是提供了一个上下文，它定义了使用 JavaScript 可以做什么，但并没有“说”太多关于 JavaScript 语言本身可以做什么。事实上，JavaScript 是一门“完整”的语言： 它可以使用在不同的上下文中，其能力与其他同类语言相比有过之而无不及。

Node.js 事实上就是另外一种上下文，它允许在后端（脱离浏览器环境）运行 JavaScript 代码。

要实现在后台运行 JavaScript 代码，代码需要先被解释然后正确的执行。Node.js 的原理正是如此，它使用了 Google 的 V8 虚拟机（Google 的 Chrome 浏览器使用的 JavaScript 执行环境），来解释和执行 JavaScript 代码。

除此之外，伴随着 Node.js 的还有许多有用的模块，它们可以简化很多重复的劳作，比如向终端输出字符串。

因此，Node.js 事实上既是一个运行时环境，同时又是一个库。

要使用 Node.js,首先需要进行安装。关于如何安装 Node.js，这里就不赘述了，可以直接参考官方的安装指南。安装完成后，继续回来阅读本书下面的内容。

“Hello World”

好了,“废话”不多说了,马上开始我们第一个 Node.js 应用:“Hello World”。

打开你最喜欢的编辑器，创建一个 *helloworld.js* 文件。我们要做就是向 STDOUT 输出“Hello World”，如下是实现该功能的代码：

```
console.log("Hello World");
```

保存该文件，并通过 Node.js 来执行：

```
node helloworld.js
```

正常的话，就会在终端输出 *Hello World* 。

好吧，我承认这个应用是有点无趣，那么下面我们就来点“干货”。

一个完整的基于 Node.js 的 web 应用

用例

我们来把目标设定得简单点，不过也要够实际才行：

- 用户可以通过浏览器使用我们的应用。
- 当用户请求 *http://domain/start* 时，可以看到一个欢迎页面，页面上有一个文件上传的表单。
- 用户可以选择一个图片并提交表单，随后文件将被上传到 *http://domain/upload*，该页面完成上传后会把图片显示在页面上。

差不多了，你现在也可以去 Google 一下，找点东西乱搞一下来完成功能。但是我们现在先不做这个。

更进一步地说，在完成这一目标的过程中，我们不仅仅需要基础的代码而不管代码是否优雅。我们还要对此进行抽象，来寻找一种适合构建更为复杂的 Node.js 应用的方式。

应用不同模块分析

我们来分解一下这个应用，为了实现上文的用例，我们需要实现哪些部分呢？

- 我们需要提供 Web 页面，因此需要一个 *HTTP 服务器*
- 对于不同的请求，根据请求的 URL，我们的服务器需要给予不同的响应，因此我们需要一个*路由*，用于把请求对应到请求处理程序（request handler）
- 当请求被服务器接收并通过路由传递之后，需要可以对其进行处理，因此我们需要最终的*请求处理程序*
- 路由还应该能处理 POST 数据，并且把数据封装成更友好的格式传递给请求处理入程序，因此需要*请求数据处理功能*
- 我们不仅仅要处理 URL 对应的请求，还要把内容显示出来，这意味着我们需要一些*视图逻辑*供请求处理程序使用，以便将内容发送给用户的浏览器

- 最后，用户需要上传图片，所以我们需要*上传处理功能*来处理这方面的细节

我们先来想想，使用 PHP 的话我们会怎么构建这个结构。一般来说我们会用一个 Apache HTTP 服务器并配上 mod_php5 模块。
从这个角度看，整个“接收 HTTP 请求并提供 Web 页面”的需求根本不需要 PHP 来处理。

不过对 Node.js 来说，概念完全不一样了。使用 Node.js 时，我们不仅仅在实现一个应用，同时还实现了整个 HTTP 服务器。事实上，我们的 Web 应用以及对应的 Web 服务器基本上是一样的。

听起来好像有一大堆活要做，但随后我们会逐渐意识到，对 Node.js 来说这并不是什么麻烦的事。

现在我们就来开始实现之路，先从第一个部分--HTTP 服务器着手。

构建应用的模块

一个基础的 HTTP 服务器

当我准备开始写我的第一个“真正的”Node.js 应用的时候，我不但不知道怎么写 Node.js 代码，也不知道怎么组织这些代码。

我应该把所有东西都放进一个文件里吗？网上有很多教程都会教你把所有的逻辑都放进一个用 Node.js 写的基础 HTTP 服务器里。但是如果我想加入更多的内容，同时还想保持代码的可读性呢？

实际上，只要把不同功能的代码放入不同的模块中，保持代码分离还是相当简单的。

这种方法允许你拥有一个干净的主文件（main file），你可以用 Node.js 执行它；同时你可以拥有干净的模块，它们可以被主文件和其他的模块调用。

那么，现在我们来创建一个用于启动我们的应用的主文件，和一个保存着我们的 HTTP 服务器代码的模块。

在我的印象里，把主文件叫做 *index.js* 或多或少是个标准格式。把服务器模块放进叫 *server.js* 的文件里则很好理解。

让我们先从服务器模块开始。在你的项目的根目录下创建一个叫 *server.js* 的文件，并写入以下代码：

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

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

搞定！你刚刚完成了一个可以工作的 HTTP 服务器。为了证明这一点，我们来运行并且测试这段代码。首先，用 Node.js 执行你的脚本：

```
node server.js
```

接下来，打开浏览器访问 http://localhost:8888/，你会看到一个写着“Hello World”的网页。

这很有趣，不是吗？让我们先来谈谈 HTTP 服务器的问题，把如何组织项目的事情先放一边吧，你觉得如何？我保证之后我们会解决那个问题的。

分析 HTTP 服务器

那么接下来，让我们分析一下这个 HTTP 服务器的构成。

第一行 *请求（require）* Node.js 自带的 *http* 模块，并且把它赋值给 *http* 变量。

接下来我们调用 http 模块提供的函数： *createServer* 。这个函数会返回一个对象，这个对象有一个叫做 *listen* 的方法，这个方法有一个数值参数，指定这个 HTTP 服务器监听的端口号。

咱们暂时先不管 *http.createServer* 的括号里的那个函数定义。

我们本来可以用这样的代码来启动服务器并侦听 8888 端口：

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

var server = http.createServer();
server.listen(8888);
```

这段代码只会启动一个侦听 8888 端口的服务器，它不做任何别的事情，甚至连请求都不会应答。

最有趣（而且，如果你之前习惯使用一个更加保守的语言，比如 PHP，它还很奇怪）的部分是 *createSever()* 的第一个参数，一个函数定义。

实际上，这个函数定义是 *createServer()* 的第一个也是唯一一个参数。因为在 JavaScript 中，函数和其他变量一样都是可以被传递的。

进行函数传递

举例来说，你可以这样做：

```
function say(word) {
  console.log(word);
}

function execute(someFunction, value) {
  someFunction(value);
}

execute(say, "Hello");
```

请仔细阅读这段代码！在这里，我们把 *say* 函数作为 *execute* 函数的第一个变量进行了传递。这里返回的不是 *say* 的返回值，而是 *say* 本身！

这样一来， *say* 就变成了 *execute* 中的本地变量 *someFunction* ，execute 可以通过调用 *someFunction()* （带括号的形式）来使用 *say* 函数。

当然，因为 *say* 有一个变量， *execute* 在调用 *someFunction* 时可以传递这样一个变量。

我们可以，就像刚才那样，用它的名字把一个函数作为变量传递。但是我们不一定要绕这个“先定义，再传递”的圈子，我们可以直接在另一个函数的括号中定义和传递这个函数：

```
function execute(someFunction, value) {
  someFunction(value);
}

execute(function(word){ console.log(word) }, "Hello");
```

我们在 *execute* 接受第一个参数的地方直接定义了我们准备传递给 *execute* 的函数。

用这种方式，我们甚至不用给这个函数起名字，这也是为什么它被叫做 *匿名函数* 。

这是我们和我所认为的“进阶”JavaScript 的第一次亲密接触，不过我们还是得循序渐进。现在，我们先接受这一点：在 JavaScript 中，一个函数可以作为另一个函数接收一个参数。我们可以先定义一个函数，然后传递，也可以在传递参数的地方直接定义函数。

函数传递是如何让 HTTP 服务器工作的

带着这些知识，我们再来看看我们简约而不简单的 HTTP 服务器：

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

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

现在它看上去应该清晰了很多：我们向 *createServer* 函数传递了一个匿名函数。

用这样的代码也可以达到同样的目的：

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

function onRequest(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}

http.createServer(onRequest).listen(8888);
```

也许现在我们该问这个问题了：我们为什么要用这种方式呢？

基于事件驱动的回调

这个问题可不好回答（至少对我来说），不过这是 Node.js 原生的工作方式。它是事件驱动的，这也是它为什么这么快的原因。

你也许会想花点时间读一下 Felix Geisendörfer 的大作 Understanding node.js，它介绍了一些背景知识。

这一切都归结于“Node.js 是事件驱动的”这一事实。好吧，其实我也不是特别确切的了解这句话的意思。不过我会试着解释，为什么它对我们用 Node.js 写网络应用（Web based application）是有意义的。

当我们使用 *http.createServer* 方法的时候，我们当然不只是想要一个侦听某个端口的服务器，我们还想要它在服务器收到一个 HTTP 请求的时候做点什么。

问题是，这是异步的：请求任何时候都可能到达，但是我们的服务器却跑在一个单进程中。

写 PHP 应用的时候，我们一点也不为此担心：任何时候当有请求进入的时候，网页服务器（通常是 Apache）就为这一请求新建一个进程，并且开始从头到尾执行相应的 PHP 脚本。

那么在我们的 Node.js 程序中，当一个新的请求到达 8888 端口的时候，我们怎么控制流程呢？

嗯，这就是 Node.js/JavaScript 的事件驱动设计能够真正帮上忙的地方了——虽然我们还得学一些新概念才能掌握它。让我们来看看这些概念是怎么应用在我们的服务器代码里的。

我们创建了服务器，并且向创建它的方法传递了一个函数。无论何时我们的服务器收到一个请求，这个函数就会被调用。

我们不知道这件事情什么时候会发生，但是我们现在有了一个处理请求的地方：它就是我们传递过去的那个函数。至于它是被预先定义的函数还是匿名函数，就无关紧要了。

这个就是传说中的 *回调* 。我们给某个方法传递了一个函数，这个方法在有相应事件发生时调用这个函数来进行 *回调* 。

至少对我来说，需要一些功夫才能弄懂它。你如果还是不太确定的话就再去读读 Felix 的博客文章。

让我们再来琢磨琢磨这个新概念。我们怎么证明，在创建完服务器之后，即使没有 HTTP 请求进来、我们的回调函数也没有被调用的情况下，我们的代码还继续有效呢？我们试试这个：

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

function onRequest(request, response) {
  console.log("Request received.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}

http.createServer(onRequest).listen(8888);

console.log("Server has started.");
```

注意：在 *onRequest* （我们的回调函数）触发的地方，我用 *console.log* 输出了一段文本。在 HTTP 服务器开始工作*之后*，也输出一段文本。

当我们与往常一样，运行它 *node server.js* 时，它会马上在命令行上输出 “Server has started.”。当我们向服务器发出请求（在浏览器访问

http://localhost:8888/ ），“Request received.”这条消息就会在命令行中出现。

这就是事件驱动的异步服务器端 JavaScript 和它的回调啦！

（请注意，当我们在服务器访问网页时，我们的服务器可能会输出两次“Request received.”。那是因为大部分服务器都会在你访问 http://localhost:8888 /时尝试读取 http://localhost:8888/favicon.ico ）

服务器是如何处理请求的

好的，接下来我们简单分析一下我们服务器代码中剩下的部分，也就是我们的回调函数 *onRequest()* 的主体部分。

当回调启动，我们的 *onRequest()* 函数被触发的时候，有两个参数被传入：*request* 和 *response* 。

它们是对象，你可以使用它们的方法来处理 HTTP 请求的细节，并且响应请求（比如向发出请求的浏览器发回一些东西）。

所以我们的代码就是：当收到请求时，使用 *response.writeHead()* 函数发送一个 HTTP 状态 200 和 HTTP 头的内容类型（content-type），使用 *response.write()* 函数在 HTTP 相应主体中发送文本“Hello World"。

最后，我们调用 *response.end()* 完成响应。

目前来说，我们对请求的细节并不在意，所以我们没有使用 *request* 对象。

服务端的模块放在哪里

OK，就像我保证过的那样，我们现在可以回到我们如何组织应用这个问题上了。我们现在在 *server.js* 文件中有一个非常基础的 HTTP 服务器代码，而且我提到通常我们会有一个叫 *index.js* 的文件去调用应用的其他模块（比如 *server.js* 中的 HTTP 服务器模块）来引导和启动应用。

我们现在就来谈谈怎么把 server.js 变成一个真正的 Node.js 模块，使它可以被我们（还没动工）的 *index.js* 主文件使用。

也许你已经注意到，我们已经在代码中使用了模块了。像这样：

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

...

http.createServer(...);
```

Node.js 中自带了一个叫做“http”的模块，我们在我们的代码中请求它并把返回值赋给一个本地变量。

这把我们的本地变量变成了一个拥有所有 *http* 模块所提供的公共方法的对象。

给这种本地变量起一个和模块名称一样的名字是一种惯例，但是你也可以按照自己的喜好来：

```
var foo = require("http");

...

foo.createServer(...);
```

很好，怎么使用 Node.js 内部模块已经很清楚了。我们怎么创建自己的模块，又怎么使用它呢？

等我们把 *server.js* 变成一个真正的模块，你就能搞明白了。

事实上，我们不用做太多的修改。把某段代码变成模块意味着我们需要把我们希望提供其功能的部分 *导出* 到请求这个模块的脚本。

目前，我们的 HTTP 服务器需要导出的功能非常简单，因为请求服务器模块的脚本仅仅是需要启动服务器而已。

我们把我们的服务器脚本放到一个叫做 *start* 的函数里，然后我们会导出这个函数。

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

function start() {
  function onRequest(request, response) {
    console.log("Request received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

这样，我们现在就可以创建我们的主文件 *index.js* 并在其中启动我们的 HTTP 了，虽然服务器的代码还在 *server.js* 中。

创建 *index.js* 文件并写入以下内容：

```
var server = require("./server");

server.start();
```

正如你所看到的，我们可以像使用任何其他的内置模块一样使用 server 模块：请求这个文件并把它指向一个变量，其中已导出的函数就可以被我们使用了。

好了。我们现在就可以从我们的主要脚本启动我们的的应用了，而它还是老样子：

```
node index.js
```

非常好，我们现在可以把我们的应用的不同部分放入不同的文件里，并且通过生成模块的方式把它们连接到一起了。

我们仍然只拥有整个应用的最初部分：我们可以接收 HTTP 请求。但是我们得做点什么——对于不同的 URL 请求，服务器应该有不同的反应。

对于一个非常简单的应用来说，你可以直接在回调函数 *onRequest()* 中做这件事情。不过就像我说过的，我们应该加入一些抽象的元素，让我们的例子变得更有趣一点儿。

处理不同的 HTTP 请求在我们的代码中是一个不同的部分，叫做“路由选择”——那么，我们接下来就创造一个叫做 *路由* 的模块吧。

如何来进行请求的“路由”

我们要为路由提供请求的 URL 和其他需要的 GET 及 POST 参数，随后路由需要根据这些数据来执行相应的代码（这里“代码”对应整个应用的第三部分：一系列在接收到请求时真正工作的处理程序）。

因此，我们需要查看 HTTP 请求，从中提取出请求的 URL 以及 GET/POST 参数。这一功能应当属于路由还是服务器（甚至作为一个模块自身的功能）确实值得探讨，但这里暂定其为我们的 HTTP 服务器的功能。

我们需要的所有数据都会包含在 request 对象中，该对象作为 *onRequest()* 回调函数的第一个参数传递。但是为了解析这些数据，我们需要额外的 Node.JS 模块，它们分别是 *url* 和 *querystring* 模块。

```
                            url.parse(string).query

                                           |

           url.parse(string).pathname      |

                       |                   |

                       |                   |

                     ------ ------------------


http://localhost:8888/start?foo=bar&hello=world

                                ---       -----

                              |           |
```

```
                                  |          |


              querystring(string)["foo"]    |


                                            |


                        querystring(string)["hello"]
```

当然我们也可以用 *querystring* 模块来解析 POST 请求体中的参数，稍后会有演示。

现在我们来给 *onRequest()*函数加上一些逻辑，用来找出浏览器请求的 URL 路径：

```
var http = require("http");
var url = require("url");

function start() {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }
```

```
  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

好了，我们的应用现在可以通过请求的 URL 路径来区别不同请求了--这使我们得以使用路由（还未完成）来将请求以 URL 路径为基准映射到处理程序上。

在我们所要构建的应用中，这意味着来自*/start* 和*/upload* 的请求可以使用不同的代码来处理。稍后我们将看到这些内容是如何整合到一起的。

现在我们可以来编写路由了，建立一个名为 *router.js* 的文件，添加以下内容：

```
function route(pathname) {
  console.log("About to route a request for " + pathname);
}

exports.route = route;
```

如你所见，这段代码什么也没干，不过对于现在来说这是应该的。在添加更多的逻辑以前，我们先来看看如何把路由和服务器整合起来。

我们的服务器应当知道路由的存在并加以有效利用。我们当然可以通过硬编码的方式将这一依赖项绑定到服务器上，但是其它语言的编程经验告诉我们这会是一件非常痛苦的事，因此我们将使用依赖注入的方式较松散地添加路由模块（你可以读读 Martin Fowlers 关于依赖注入的大作来作为背景知识）。

首先，我们来扩展一下服务器的 *start()*函数，以便将路由函数作为参数传递过去：

```
var http = require("http");
var url = require("url");

function start(route) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    route(pathname);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

同时，我们会相应扩展 *index.js*，使得路由函数可以被注入到服务器中：

```
var server = require("./server");
var router = require("./router");

server.start(router.route);
```

在这里，我们传递的函数依旧什么也没做。

如果现在启动应用（*node index.js，始终记得这个命令行*），随后请求一个 URL，你将会看到应用输出相应的信息，这表明我们的 HTTP 服务器已经在使用路由模块了，并会将请求的路径传递给路由：

```
bash$ node index.js


Request for /foo received.


About to route a request for /foo
```

（以上输出已经去掉了比较烦人的/favicon.ico 请求相关的部分）。

行为驱动执行

请允许我再次脱离主题，在这里谈一谈函数式编程。

将函数作为参数传递并不仅仅出于技术上的考量。对软件设计来说，这其实是个哲学问题。想想这样的场景：在 index 文件中，我们可以将 *router* 对象传递进去，服务器随后可以调用这个对象的 *route* 函数。

就像这样，我们传递一个东西，然后服务器利用这个东西来完成一些事。嗨那个叫路由的东西，能帮我把这个路由一下吗？

但是服务器其实不需要这样的东西。它只需要把事情做完就行，其实为了把事情做完，你根本不需要东西，你需要的是动作。也就是说，你不需要*名词*，你需要*动词*。

理解了这个概念里最核心、最基本的思想转换后，我自然而然地理解了函数编程。

我是在读了 Steve Yegge 的大作名词王国中的死刑之后理解函数编程。你也去读一读这本书吧，真的。这是曾给予我阅读的快乐的关于软件的书籍之一。

路由给真正的请求处理程序

回到正题，现在我们的 HTTP 服务器和请求路由模块已经如我们的期望，可以相互交流了，就像一对亲密无间的兄弟。

当然这还远远不够，路由，顾名思义，是指我们要针对不同的 URL 有不同的处理方式。例如处理*/start* 的“业务逻辑”就应该和处理*/upload* 的不同。

在现在的实现下，路由过程会在路由模块中“结束”，并且路由模块并不是真正针对请求“采取行动”的模块，否则当我们的应用程序变得更为复杂时，将无法很好地扩展。

我们暂时把作为路由目标的函数称为请求处理程序。现在我们不要急着来开发路由模块，因为如果请求处理程序没有就绪的话，再怎么完善路由模块也没有多大意义。

应用程序需要新的部件，因此加入新的模块 -- 已经无需为此感到新奇了。我们来创建一个叫做 requestHandlers 的模块，并对于每一个请求处理程序，添加一个占位用函数，随后将这些函数作为模块的方法导出：

```
function start() {
  console.log("Request handler 'start' was called.");
}

function upload() {
  console.log("Request handler 'upload' was called.");
}

exports.start = start;
exports.upload = upload;
```

这样我们就可以把请求处理程序和路由模块连接起来，让路由“有路可寻”。

在这里我们得做个决定：是将 requestHandlers 模块硬编码到路由里来使用，还是再添加一点依赖注入？虽然和其他模式一样，依赖注入不应该仅仅为使用而使用，但在现在这个情况下，使用依赖注入可以让路由和请求处理程序之间的耦合更加松散，也因此能让路由的重用性更高。

这意味着我们得将请求处理程序从服务器传递到路由中，但感觉上这么做更离谱了，我们得一路把这堆请求处理程序从我们的主文件传递到服务器中，再将之从服务器传递到路由。

那么我们要怎么传递这些请求处理程序呢？别看现在我们只有 2 个处理程序，在一个真实的应用中，请求处理程序的数量会不断增加，我们当然不想每次有一个新的 URL 或请求处理程序时，都要为了在路由里完成请求

到处理程序的映射而反复折腾。除此之外，在路由里有一大堆 *if request == x then call handler y* 也使得系统丑陋不堪。

仔细想想，有一大堆东西，每个都要映射到一个字符串（就是请求的 URL）上？似乎关联数组（associative array）能完美胜任。

不过结果有点令人失望，JavaScript 没提供关联数组 -- 也可以说它提供了？事实上，在 JavaScript 中，真正能提供此类功能的是它的对象。

在这方面，http://msdn.microsoft.com/en-us/magazine/cc163419.aspx 有一个不错的介绍，我在此摘录一段：

在 C++或 C# 中，当我们谈到对象，指的是类或者结构体的实例。对象根据他们实例化的模板（就是所谓的类），会拥有不同的属性和方法。但在 JavaScript 里对象不是这个概念。在 JavaScript 中，对象就是一个键/值对的集合 -- 你可以把 JavaScript 的对象想象成一个键为字符串类型的字典。

但如果 JavaScript 的对象仅仅是键/值对的集合，它又怎么会拥有方法呢？好吧，这里的值可以是字符串、数字或者……函数！

好了，最后再回到代码上来。现在我们已经确定将一系列请求处理程序通过一个对象来传递，并且需要使用松耦合的方式将这个对象注入到 *route()* 函数中。

我们先将这个对象引入到主文件 *index.js* 中：

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");

var handle = {}
handle["/"] = requestHandlers.start;
handle["/start"] = requestHandlers.start;
handle["/upload"] = requestHandlers.upload;

server.start(router.route, handle);
```

虽然 *handle* 并不仅仅是一个“东西”（一些请求处理程序的集合），我还是建议以一个动词作为其命名，这样做可以让我们在路由中使用更流畅的表达式，稍后会有说明。

正如所见，将不同的 URL 映射到相同的请求处理程序上是很容易的：只要在对象中添加一个键为"/"的属性，对应 *requestHandlers.start* 即可，这样我们就可以干净简洁地配置*/start* 和/的请求都交由 *start* 这一处理程序处理。

在完成了对象的定义后，我们把它作为额外的参数传递给服务器，为此将 *server.js* 修改如下：

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    route(handle, pathname);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

这样我们就在 *start()*函数里添加了 *handle* 参数，并且把 handle 对象作为第一个参数传递给了 *route()*回调函数。

然后我们相应地在 *route.js* 文件中修改 *route()*函数：

```
function route(handle, pathname) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname]();
  } else {
    console.log("No request handler found for " + pathname);
  }
}

exports.route = route;
```

通过以上代码，我们首先检查给定的路径对应的请求处理程序是否存在，如果存在的话直接调用相应的函数。我们可以用从关联数组中获取元素一样的方式从传递的对象中获取请求处理函数，因此就有了简洁流畅的形如*handle[pathname]();*的表达式，这个感觉就像在前方中提到的那样：“嗨，请帮我处理了这个路径”。

有了这些，我们就把服务器、路由和请求处理程序在一起了。现在我们启动应用程序并在浏览器中访问 *http://localhost:8888/start*，以下日志可以说明系统调用了正确的请求处理程序：

```
Server has started.


Request for /start received.
```

```
About to route a request for /start


Request handler 'start' was called.
```

并且在浏览器中打开 *http://localhost:8888/*可以看到这个请求同样被 *start* 请求处理程序处理了：

```
Request for / received.


About to route a request for /


Request handler 'start' was called.
```

让请求处理程序作出响应

很好。不过现在要是请求处理程序能够向浏览器返回一些有意义的信息而并非全是“Hello World”，那就更好了。

这里要记住的是，浏览器发出请求后获得并显示的“Hello World”信息仍是来自于我们 *server.js* 文件中的 *onRequest* 函数。

其实“处理请求”说白了就是“对请求作出响应”，因此，我们需要让请求处理程序能够像 *onRequest* 函数那样可以和浏览器进行“对话”。

不好的实现方式

对于我们这样拥有 PHP 或者 Ruby 技术背景的开发者来说，最直截了当的实现方式事实上并不是非常靠谱： 看似有效，实则未必如此。

这里我指的“直截了当的实现方式”意思是：让请求处理程序通过 *onRequest* 函数直接返回（*return()*）他们要展示给用户的信息。

我们先就这样去实现，然后再来看为什么这不是一种很好的实现方式。

让我们从让请求处理程序返回需要在浏览器中显示的信息开始。我们需要将 *requestHandler.js* 修改为如下形式：

```
function start() {
  console.log("Request handler 'start' was called.");
  return "Hello Start";
}

function upload() {
  console.log("Request handler 'upload' was called.");
  return "Hello Upload";
}
```

```
exports.start = start;
exports.upload = upload;
```

好的。同样的，请求路由需要将请求处理程序返回给它的信息返回给服务器。因此，我们需要将 *router.js* 修改为如下形式：

```
function route(handle, pathname) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    return handle[pathname]();
  } else {
    console.log("No request handler found for " + pathname);
    return "404 Not found";
  }
}

exports.route = route;
```

正如上述代码所示，当请求无法路由的时候，我们也返回了一些相关的错误信息。

最后，我们需要对我们的 *server.js* 进行重构以使得它能够将请求处理程序通过请求路由返回的内容响应给浏览器，如下所示：

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    response.writeHead(200, {"Content-Type": "text/plain"});
    var content = route(handle, pathname)
    response.write(content);
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

如果我们运行重构后的应用，一切都会工作的很好：请求 http://localhost:8888/start,浏览器会输出“Hello Start”，请求 http://localhost:8888/upload 会输出“Hello Upload”,而请求 http://localhost:8888/foo 会输出“404 Not found”。

好，那么问题在哪里呢？简单的说就是： 当未来有请求处理程序需要进行非阻塞的操作的时候，我们的应用就“挂”了。

没理解？没关系，下面就来详细解释下。

阻塞与非阻塞

正如此前所提到的，当在请求处理程序中包括非阻塞操作时就会出问题。但是，在说这之前，我们先来看看什么是阻塞操作。

我不想去解释“阻塞”和“非阻塞”的具体含义，我们直接来看，当在请求处理程序中加入阻塞操作时会发生什么。

这里，我们来修改下 *start* 请求处理程序，我们让它等待 10 秒以后再返回“Hello Start”。因为，JavaScript 中没有类似 *sleep()*这样的操作，所以这里只能够来点小 Hack 来模拟实现。

让我们将 *requestHandlers.js* 修改成如下形式：

```
function start() {
  console.log("Request handler 'start' was called.");

  function sleep(milliSeconds) {
    var startTime = new Date().getTime();
    while (new Date().getTime() < startTime + milliSeconds);
  }

  sleep(10000);
  return "Hello Start";
}

function upload() {
  console.log("Request handler 'upload' was called.");
```

```
  return "Hello Upload";
}

exports.start = start;
exports.upload = upload;
```

上述代码中，当函数 *start()*被调用的时候，Node.js 会先等待 10 秒，之后才会返回“Hello Start”。当调用 *upload()*的时候，会和此前一样立即返回。

（当然了，这里只是模拟休眠 10 秒，实际场景中，这样的阻塞操作有很多，比方说一些长时间的计算操作等。）

接下来就让我们来看看，我们的改动带来了哪些变化。

如往常一样，我们先要重启下服务器。为了看到效果，我们要进行一些相对复杂的操作（跟着我一起做）： 首先，打开两个浏览器窗口或者标签页。在第一个浏览器窗口的地址栏中输入 http://localhost:8888/start，但是先不要打开它！

在第二个浏览器窗口的地址栏中输入 http://localhost:8888/upload， 同样的，先不要打开它！

接下来，做如下操作：在第一个窗口中（“/start”）按下回车，然后快速切换到第二个窗口中（“/upload”）按下回车。

注意，发生了什么： /start URL 加载花了 10 秒，这和我们预期的一样。但是，/upload URL 居然*也*花了 10 秒，而它在对应的请求处理程序中并没有类似于 *sleep()*这样的操作！

这到底是为什么呢？原因就是 *start()*包含了阻塞操作。形象的说就是“它阻塞了所有其他的处理工作”。

这显然是个问题，因为 Node 一向是这样来标榜自己的：“*在 node 中除了代码，所有一切都是并行执行的*”。

这句话的意思是说，Node.js 可以在不新增额外线程的情况下，依然可以对任务进行并行处理 —— Node.js 是单线程的。它通过事件轮询（event loop）来实现并行操作，对此，我们应该要充分利用这一点 —— 尽可能的避免阻塞操作，取而代之，多使用非阻塞操作。

然而，要用非阻塞操作，我们需要使用回调，通过将函数作为参数传递给其他需要花时间做处理的函数（比方说，休眠 10 秒，或者查询数据库，又或者是进行大量的计算）。

对于 Node.js 来说，它是这样处理的：“*嘿，probablyExpensiveFunction()（译者注：这里指的就是需要花时间处理的函数），你继续处理你的事情，我（Node.js 线程）先不等你了，我继续去处理你后面的代码，请你提供一个 callbackFunction()，等你处理完之后我会去调用该回调函数的，谢谢！*”

（如果想要了解更多关于事件轮询细节，可以阅读 Mixu 的博文——理解 node.js 的事件轮询。）

接下来，我们会介绍一种错误的使用非阻塞操作的方式。

和上次一样，我们通过修改我们的应用来暴露问题。

这次我们还是拿 *start* 请求处理程序来“开刀”。将其修改成如下形式：

```
var exec = require("child_process").exec;

function start() {
  console.log("Request handler 'start' was called.");
  var content = "empty";

  exec("ls -lah", function (error, stdout, stderr) {
    content = stdout;
  });

  return content;
```

```
}

function upload() {
  console.log("Request handler 'upload' was called.");
  return "Hello Upload";
}

exports.start = start;
exports.upload = upload;
```

上述代码中，我们引入了一个新的 Node.js 模块，*child_process*。之所以用它，是为了实现一个既简单又实用的非阻塞操作：*exec()*。

*exec()*做了什么呢？它从 Node.js 来执行一个 shell 命令。在上述例子中，我们用它来获取当前目录下所有的文件（“ls -lah”）,然后，当/*start*URL 请求的时候将文件信息输出到浏览器中。

上述代码是非常直观的： 创建了一个新的变量 *content*（初始值为“empty”），执行“ls -lah”命令，将结果赋值给 content，最后将 content 返回。

和往常一样，我们启动服务器，然后访问“http://localhost:8888/start” 。

之后会载入一个漂亮的 web 页面，其内容为“empty”。怎么回事？

这个时候，你可能大致已经猜到了，*exec()*在非阻塞这块发挥了神奇的功效。它其实是个很好的东西，有了它，我们可以执行非常耗时的 shell 操作而无需迫使我们的应用停下来等待该操作。

（如果想要证明这一点，可以将“ls -lah”换成比如“find /”这样更耗时的操作来效果）。

然而，针对浏览器显示的结果来看，我们并不满意我们的非阻塞操作，对吧？

好，接下来，我们来修正这个问题。在这过程中，让我们先来看看为什么当前的这种方式不起作用。

问题就在于，为了进行非阻塞工作，*exec()*使用了回调函数。

在我们的例子中，该回调函数就是作为第二个参数传递给 *exec()*的匿名函数：

```
function (error, stdout, stderr) {
  content = stdout;
}
```

现在就到了问题根源所在了：我们的代码是同步执行的，这就意味着在调用 *exec()*之后，Node.js 会立即执行 *return content* ；在这个时候，*content* 仍然是“empty”，因为传递给 *exec()*的回调函数还未执行到——因为 *exec()* 的操作是异步的。

我们这里“ls -lah”的操作其实是非常快的（除非当前目录下有上百万个文件）。这也是为什么回调函数也会很快的执行到 —— 不过，不管怎么说它还是异步的。

为了让效果更加明显，我们想象一个更耗时的命令： “find /”，它在我机器上需要执行 1 分钟左右的时间，然而，尽管在请求处理程序中，我把“ls -lah”换成“find /”，当打开/start URL 的时候，依然能够立即获得 HTTP 响应 —— 很明显，当 *exec()*在后台执行的时候，Node.js 自身会继续执行后面的代码。并且我们这里假设传递给 *exec()*的回调函数，只会在“find /”命令执行完成之后才会被调用。

那究竟我们要如何才能实现将当前目录下的文件列表显示给用户呢？

好，了解了这种不好的实现方式之后，我们接下来来介绍如何以正确的方式让请求处理程序对浏览器请求作出响应。

以非阻塞操作进行请求响应

我刚刚提到了这样一个短语 —— “正确的方式”。而事实上通常“正确的方式”一般都不简单。

不过，用 Node.js 就有这样一种实现方案： 函数传递。下面就让我们来具体看看如何实现。

到目前为止，我们的应用已经可以通过应用各层之间传递值的方式（请求处理程序 -> 请求路由 -> 服务器）将请求处理程序返回的内容（请求处理程序最终要显示给用户的内容）传递给 HTTP 服务器。

现在我们采用如下这种新的实现方式：相对采用将内容传递给服务器的方式，我们这次采用将服务器“传递”给内容的方式。 从实践角度来说，就是将 *response* 对象（从服务器的回调函数 *onRequest()*获取）通过请求路由传递给请求处理程序。 随后，处理程序就可以采用该对象上的函数来对请求作出响应。

原理就是如此，接下来让我们来一步步实现这种方案。

先从 *server.js* 开始：

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    route(handle, pathname, response);
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

相对此前从 *route()*函数获取返回值的做法，这次我们将 response 对象作为第三个参数传递给 *route()*函数，并且，我们将 *onRequest()*处理程序中所有有关 *response* 的函数调都移除，因为我们希望这部分工作让 *route()* 函数来完成。

下面就来看看我们的 *router.js*:

```
function route(handle, pathname, response) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response);
  } else {
    console.log("No request handler found for " + pathname);
    response.writeHead(404, {"Content-Type": "text/plain"});
```

```
    response.write("404 Not found");
    response.end();
  }
}

exports.route = route;
```

同样的模式：相对此前从请求处理程序中获取返回值，这次取而代之的是直接传递 *response* 对象。

如果没有对应的请求处理器处理，我们就直接返回“404”错误。

最后，我们将 *requestHandler.js* 修改为如下形式：

```
var exec = require("child_process").exec;

function start(response) {
  console.log("Request handler 'start' was called.");

  exec("ls -lah", function (error, stdout, stderr) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write(stdout);
    response.end();
  });
}

function upload(response) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello Upload");
  response.end();
}
```

```
exports.start = start;
exports.upload = upload;
```

我们的处理程序函数需要接收 response 参数，为了对请求作出直接的响应。

start 处理程序在 *exec()*的匿名回调函数中做请求响应的操作，而 *upload* 处理程序仍然是简单的回复“Hello World”，只是这次是使用 *response* 对象而已。

这时再次我们启动应用（*node index.js*），一切都会工作的很好。

如果想要证明*/start* 处理程序中耗时的操作不会阻塞对*/upload* 请求作出立即响应的话，可以将 *requestHandlers.js* 修改为如下形式：

```
var exec = require("child_process").exec;

function start(response) {
  console.log("Request handler 'start' was called.");

  exec("find /",
    { timeout: 10000, maxBuffer: 20000*1024 },
    function (error, stdout, stderr) {
      response.writeHead(200, {"Content-Type": "text/plain"});
      response.write(stdout);
      response.end();
    });
}
```

```
function upload(response) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

这样一来，当请求 http://localhost:8888/start 的时候，会花 10 秒钟的时间才载入，而当请求 http://localhost:8888/upload 的时候，会立即响应，纵然这个时候/start 响应还在处理中。

更有用的场景

到目前为止，我们做的已经很好了，但是，我们的应用没有实际用途。

服务器，请求路由以及请求处理程序都已经完成了，下面让我们按照此前的用例给网站添加交互：用户选择一个文件，上传该文件，然后在浏览器中看到上传的文件。 为了保持简单，我们假设用户只会上传图片，然后我们应用将该图片显示到浏览器中。

好，下面就一步步来实现，鉴于此前已经对 JavaScript 原理性技术性的内容做过大量介绍了，这次我们加快点速度。

要实现该功能，分为如下两步： 首先，让我们来看看如何处理 POST 请求（非文件上传），之后，我们使用 Node.js 的一个用于文件上传的外部模块。之所以采用这种实现方式有两个理由。

第一，尽管在 Node.js 中处理基础的 POST 请求相对比较简单，但在这过程中还是能学到很多。

第二，用 Node.js 来处理文件上传（multipart POST 请求）是比较复杂的，它*不*在本书的范畴，但，如何使用外部模块却是在本书涉猎内容之内。

处理 POST 请求

考虑这样一个简单的例子：我们显示一个文本区（textarea）供用户输入内容，然后通过 POST 请求提交给服务器。最后，服务器接受到请求，通过处理程序将输入的内容展示到浏览器中。

/start 请求处理程序用于生成带文本区的表单，因此，我们将 *requestHandlers.js* 修改为如下形式：

```
function start(response) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
```

```
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

好了，现在我们的应用已经很完善了，都可以获得威比奖（Webby Awards）了，哈哈。（译者注：威比奖是由国际数字艺术与科学学院主办的评选全球最佳网站的奖项，具体参见详细说明）通过在浏览器中访问 http://localhost:8888/start 就可以看到简单的表单了，要记得重启服务器哦！

你可能会说：这种直接将视觉元素放在请求处理程序中的方式太丑陋了。说的没错，但是，我并不想在本书中介绍诸如 MVC 之类的模式，因为这对于你了解 JavaScript 或者 Node.js 环境来说没多大关系。

余下的篇幅，我们来探讨一个更有趣的问题： 当用户提交表单时，触发 */upload* 请求处理程序处理 POST 请求的问题。

现在，我们已经是新手中的专家了，很自然会想到采用异步回调来实现非阻塞地处理 POST 请求的数据。

这里采用非阻塞方式处理是明智的，因为 POST 请求一般都比较“重”——用户可能会输入大量的内容。用阻塞的方式处理大数据量的请求必然会导致用户操作的阻塞。

为了使整个过程非阻塞，Node.js 会将 POST 数据拆分成很多小的数据块，然后通过触发特定的事件，将这些小数据块传递给回调函数。这里的特定的事件有 *data* 事件（表示新的小数据块到达了）以及 *end* 事件（表示所有的数据都已经接收完毕）。

我们需要告诉 Node.js 当这些事件触发的时候，回调哪些函数。怎么告诉呢？ 我们通过在 *request* 对象上注册*监听器*（listener） 来实现。这里的

request 对象是每次接收到 HTTP 请求时候，都会把该对象传递给 *onRequest* 回调函数。

如下所示：

```
request.addListener("data", function(chunk) {
  // called when a new chunk of data was received
});

request.addListener("end", function() {
  // called when all chunks of data have been received
});
```

问题来了，这部分逻辑写在哪里呢？ 我们现在只是在服务器中获取到了 *request* 对象 —— 我们并没有像之前 *response* 对象那样，把 request 对象传递给请求路由和请求处理程序。

在我看来，获取所有来自请求的数据，然后将这些数据给应用层处理，应该是 HTTP 服务器要做的事情。因此，我建议，我们直接在服务器中处理 POST 数据，然后将最终的数据传递给请求路由和请求处理器，让他们来进行进一步的处理。

因此，实现思路就是：将 *data* 和 *end* 事件的回调函数直接放在服务器中，在 *data* 事件回调中收集所有的 POST 数据，当接收到所有数据，触发 *end*

事件后，其回调函数调用请求路由，并将数据传递给它，然后，请求路由再将该数据传递给请求处理程序。

还等什么，马上来实现。先从 *server.js* 开始：

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var postData = "";
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    request.setEncoding("utf8");

    request.addListener("data", function(postDataChunk) {
      postData += postDataChunk;
      console.log("Received POST data chunk '"+
      postDataChunk + "'.");
    });

    request.addListener("end", function() {
      route(handle, pathname, response, postData);
    });

  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

上述代码做了三件事情： 首先，我们设置了接收数据的编码格式为 UTF-8，然后注册了“data”事件的监听器，用于收集每次接收到的新数据块，并将其赋值给 *postData* 变量，最后，我们将请求路由的调用移到 *end* 事件处理程序中，以确保它只会当所有数据接收完毕后才触发，并且只触发一次。我们同时还把 POST 数据传递给请求路由，因为这些数据，请求处理程序会用到。

上述代码在每个数据块到达的时候输出了日志，这对于最终生产环境来说，是很不好的（数据量可能会很大，还记得吧？），但是，在开发阶段是很有用的，有助于让我们看到发生了什么。

我建议可以尝试下，尝试着去输入一小段文本，以及大段内容，当大段内容的时候，就会发现 *data* 事件会触发多次。

再来点酷的。我们接下来在/upload 页面，展示用户输入的内容。要实现该功能，我们需要将 *postData* 传递给请求处理程序，修改 *router.js* 为如下形式：

```
function route(handle, pathname, response, postData) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response, postData);
  } else {
```

```
    console.log("No request handler found for " + pathname);
    response.writeHead(404, {"Content-Type": "text/plain"});
    response.write("404 Not found");
    response.end();
  }
}

exports.route = route;
```

然后，在 *requestHandlers.js* 中，我们将数据包含在对 *upload* 请求的响应中：

```
function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
```

```
  response.write("You've sent: " + postData);
  response.end();
}

exports.start = start;
exports.upload = upload;
```

好了，我们现在可以接收 POST 数据并在请求处理程序中处理该数据了。

我们最后要做的是： 当前我们是把请求的整个消息体传递给了请求路由和请求处理程序。我们应该只把 POST 数据中，我们感兴趣的部分传递给请求路由和请求处理程序。在我们这个例子中，我们感兴趣的其实只是 *text* 字段。

我们可以使用此前介绍过的 *querystring* 模块来实现：

```
var querystring = require("querystring");

function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
```

```
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent the text: "+
  querystring.parse(postData).text);
  response.end();
}

exports.start = start;
exports.upload = upload;
```

好了，以上就是关于处理 POST 数据的全部内容。

处理文件上传

最后，我们来实现我们最终的用例：允许用户上传图片，并将该图片在浏览器中显示出来。

回到 90 年代，这个用例完全可以满足用于 IPO 的商业模型了，如今，我们通过它能学到这样两件事情： 如何安装外部 Node.js 模块，以及如何将它们应用到我们的应用中。

这里我们要用到的外部模块是 Felix Geisendörfer 开发的 *node-formidable* 模块。它对解析上传的文件数据做了很好的抽象。 其实说白了，处理文件上传*“就是”*处理 POST 数据 —— 但是，麻烦的是在具体的处理细节，所以，这里采用现成的方案更合适点。

使用该模块，首先需要安装该模块。Node.js 有它自己的包管理器，叫 *NPM*。它可以让安装 Node.js 的外部模块变得非常方便。通过如下一条命令就可以完成该模块的安装：

```
npm install formidable
```

如果终端输出如下内容：

```
npm info build Success: formidable@1.0.2
npm ok
```

就说明模块已经安装成功了。

现在我们就可以用 *formidable* 模块了——使用外部模块与内部模块类似，用 require 语句将其引入即可：

```
var formidable = require("formidable");
```

这里该模块做的就是将通过 HTTP POST 请求提交的表单，在 Node.js 中可以被解析。我们要做的就是创建一个新的 *IncomingForm*，它是对提交表单的抽象表示，之后，就可以用它解析 request 对象，获取表单中需要的数据字段。

node-formidable 官方的例子展示了这两部分是如何融合在一起工作的：

```
var formidable = require('formidable'),
    http = require('http'),
    sys = require('sys');

http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    // parse a file upload
    var form = new formidable.IncomingForm();
    form.parse(req, function(err, fields, files) {
      res.writeHead(200, {'content-type': 'text/plain'});
      res.write('received upload:\n\n');
      res.end(sys.inspect({fields: fields, files: files}));
    });
    return;
  }

  // show a file upload form
  res.writeHead(200, {'content-type': 'text/html'});
  res.end(
    '<form action="/upload" enctype="multipart/form-data" '+
    'method="post">'+
    '<input type="text" name="title"><br>'+
```

```
    '<input type="file" name="upload" multiple="multiple"><br>'+
    '<input type="submit" value="Upload">'+
    '</form>'
  );
}).listen(8888);
```

如果我们将上述代码，保存到一个文件中，并通过 *node* 来执行，就可以进行简单的表单提交了，包括文件上传。然后，可以看到通过调用 *form.parse* 传递给回调函数的 *files* 对象的内容，如下所示：

```
received upload:




{ fields: { title: 'Hello World' },

  files:

   { upload:

      { size: 1558,

        path: '/tmp/1c747974a27a6292743669e91f29350b',
```

```
        name: 'us-flag.png',

        type: 'image/png',

        lastModifiedDate: Tue, 21 Jun 2011 07:02:41 GMT,

        _writeStream: [Object],

        length: [Getter],

        filename: [Getter],

        mime: [Getter] } } }
```

为了实现我们的功能，我们需要将上述代码应用到我们的应用中，另外，我们还要考虑如何将上传文件的内容（保存在*/tmp* 目录中）显示到浏览器中。

我们先来解决后面那个问题： 对于保存在本地硬盘中的文件，如何才能在浏览器中看到呢？

显然，我们需要将该文件读取到我们的服务器中，使用一个叫 *fs* 的模块。

我们来添加*/show*URL 的请求处理程序，该处理程序直接硬编码将文件 */tmp/test.png* 内容展示到浏览器中。当然了，首先需要将该图片保存到这个位置才行。

将 *requestHandlers.js* 修改为如下形式：

```
var querystring = require("querystring"),
    fs = require("fs");

function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
```

```
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent the text: "+
  querystring.parse(postData).text);
  response.end();
}

function show(response, postData) {
  console.log("Request handler 'show' was called.");
  fs.readFile("/tmp/test.png", "binary", function(error, file) {
    if(error) {
      response.writeHead(500, {"Content-Type": "text/plain"});
      response.write(error + "\n");
      response.end();
    } else {
      response.writeHead(200, {"Content-Type": "image/png"});
      response.write(file, "binary");
      response.end();
    }
  });
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

我们还需要将这新的请求处理程序，添加到 *index.js* 中的路由映射表中：

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");

var handle = {}
handle["/"] = requestHandlers.start;
handle["/start"] = requestHandlers.start;
handle["/upload"] = requestHandlers.upload;
handle["/show"] = requestHandlers.show;
```

```
server.start(router.route, handle);
```

重启服务器之后，通过访问 http://localhost:8888/show，就可以看到保存在*/tmp/test.png* 的图片了。

好，最后我们要的就是：

- 在*/start* 表单中添加一个文件上传元素
- 将 node-formidable 整合到我们的 *upload* 请求处理程序中，用于将上传的图片保存到*/tmp/test.png*
- 将上传的图片内嵌到*/upload*URL 输出的 HTML 中

第一项很简单。只需要在 HTML 表单中，添加一个 *multipart/form-data* 的编码类型，移除此前的文本区，添加一个文件上传组件，并将提交按钮的文案改为“Upload file”即可。 如下 *requestHandler.js* 所示：

```
var querystring = require("querystring"),
    fs = require("fs");

function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
```

```
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" enctype="multipart/form-data" '+
    'method="post">'+
    '<input type="file" name="upload">'+
    '<input type="submit" value="Upload file" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent the text: "+
  querystring.parse(postData).text);
  response.end();
}

function show(response, postData) {
  console.log("Request handler 'show' was called.");
  fs.readFile("/tmp/test.png", "binary", function(error, file) {
    if(error) {
      response.writeHead(500, {"Content-Type": "text/plain"});
      response.write(error + "\n");
      response.end();
    } else {
      response.writeHead(200, {"Content-Type": "image/png"});
      response.write(file, "binary");
      response.end();
    }
  });
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

很好。下一步相对比较复杂。这里有这样一个问题： 我们需要在 *upload* 处理程序中对上传的文件进行处理，这样的话，我们就需要将 *request* 对象传递给 node-formidable 的 *form.parse* 函数。

但是，我们有的只是 *response* 对象和 *postData* 数组。看样子，我们只能不得不将 *request* 对象从服务器开始一路通过请求路由，再传递给请求处理程序。 或许还有更好的方案，但是，不管怎么说，目前这样做可以满足我们的需求。

到这里，我们可以将 *postData* 从服务器以及请求处理程序中移除了 —— 一方面，对于我们处理文件上传来说已经不需要了，另外一方面，它甚至可能会引发这样一个问题： 我们已经“消耗”了 *request* 对象中的数据，这意味着，对于 *form.parse* 来说，当它想要获取数据的时候就什么也获取不到了。（因为 Node.js 不会对数据做缓存）

我们从 *server.js* 开始 —— 移除对 postData 的处理以及 *request.setEncoding* （这部分 node-formidable 自身会处理），转而采用将 *request* 对象传递给请求路由的方式：

```
var http = require("http");
var url = require("url");
```

```
function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");
    route(handle, pathname, response, request);
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

接下来是 router.js —— 我们不再需要传递 *postData* 了，这次要传递 *request* 对象：

```
function route(handle, pathname, response, request) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response, request);
  } else {
    console.log("No request handler found for " + pathname);
    response.writeHead(404, {"Content-Type": "text/html"});
    response.write("404 Not found");
    response.end();
  }
}

exports.route = route;
```

现在，*request* 对象就可以在我们的 *upload* 请求处理程序中使用了。

node-formidable 会处理将上传的文件保存到本地/*tmp* 目录中，而我们需

要做的是确保该文件保存成*/tmp/test.png*。 没错，我们保持简单，并假设只允许上传 PNG 图片。

这里采用*fs.renameSync(path1,path2)*来实现。要注意的是，正如其名，该方法是同步执行的， 也就是说，如果该重命名的操作很耗时的话会阻塞。 这块我们先不考虑。

接下来，我们把处理文件上传以及重命名的操作放到一起，如下*requestHandlers.js* 所示：

```
var querystring = require("querystring"),
    fs = require("fs"),
    formidable = require("formidable");

function start(response) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" enctype="multipart/form-data" '+
    'method="post">'+
    '<input type="file" name="upload" multiple="multiple">'+
    '<input type="submit" value="Upload file" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
```

```
    response.write(body);
    response.end();
}

function upload(response, request) {
  console.log("Request handler 'upload' was called.");

  var form = new formidable.IncomingForm();
  console.log("about to parse");
  form.parse(request, function(error, fields, files) {
    console.log("parsing done");
    fs.renameSync(files.upload.path, "/tmp/test.png");
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("received image:<br/>");
    response.write("<img src='/show' />");
    response.end();
  });
}

function show(response) {
  console.log("Request handler 'show' was called.");
  fs.readFile("/tmp/test.png", "binary", function(error, file) {
    if(error) {
      response.writeHead(500, {"Content-Type": "text/plain"});
      response.write(error + "\n");
      response.end();
    } else {
      response.writeHead(200, {"Content-Type": "image/png"});
      response.write(file, "binary");
      response.end();
    }
  });
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

好了，重启服务器，我们应用所有的功能就可以用了。选择一张本地图片，将其上传到服务器，然后浏览器就会显示该图片。

总结与展望

恭喜，我们的任务已经完成了！我们开发完了一个 Node.js 的 web 应用，应用虽小，但却“五脏俱全”。 期间，我们介绍了很多技术点：服务端 JavaScript、函数式编程、阻塞与非阻塞、回调、事件、内部和外部模块等等。

当然了，还有许多本书没有介绍到的： 如何操作数据库、如何进行单元测试、如何开发 Node.js 的外部模块以及一些简单的诸如如何获取 GET 请求之类的方法。

但本书毕竟只是一本给初学者的教程 —— 不可能覆盖到所有的内容。

幸运的是，Node.js 社区非常活跃（作个不恰当的比喻就是犹如一群有多动症小孩子在一起，能不活跃吗？）， 这意味着，有许多关于 Node.js 的资源，有什么问题都可以向社区寻求解答。 其中 Node.js 社区的 wiki 以及 NodeCloud 就是最好的资源。