# 静态文件服务器

本文是我对V5Node项目的总结，该项目的特性包括：

1. 项目大多数的文件都是属于静态文件，只有数据部分存在动态请求。
2. 数据部分的请求都呈现为RESTful的特性。

所以项目主要包含两个部分就是静态服务器和RESTful服务器。本文讲的是静态文件服务器部分。

既是一个新的项目，那么创建v5node目录是应该的。既是一个Node应用，创建一个app.js文件也是应该的。

我们的app.js文件里的结构很明确：

var PORT = 8000;

var http = require('http');

var server = http.createServer(function (request, response) {

// TODO

});

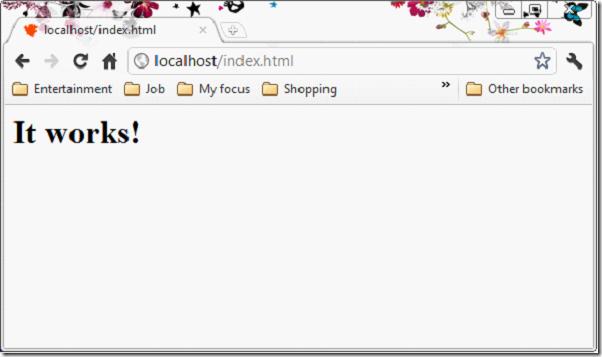
server.listen(PORT);

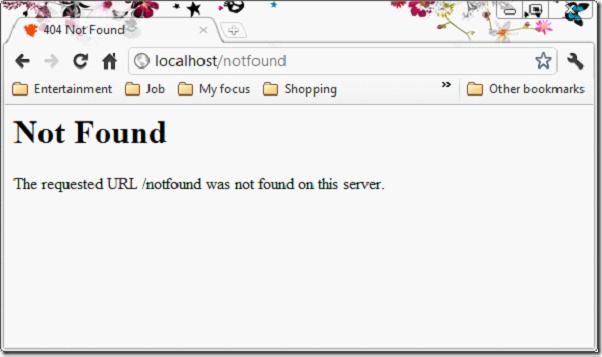
console.log("Server runing at port: " + PORT + ".");

因为当前要实现的功能是静态文件服务器，那么以Apache为例，让我们回忆一下静态文件服务器都有哪些功能。

浏览器发送URL，服务端解析URL，对应到硬盘上的文件。如果文件存在，返回200状态码，并发送文件到浏览器端；如果文件不存在，返回404状态码，发送一个404的文件到浏览器端。

以下两图是Apache经典的两种状态。





现在需求已经明了，那么我们开始实现吧。

## 实现路由

路由部分的实现在《The Node Beginner Book》已经被描述过，此处不例外。

添加url模块是必要的，然后解析pathname。

以下是实现代码：

var server = http.createServer(function (request, response) {

var pathname = url.parse(request.url).pathname;

response.write(pathname);

response.end();

});

现在的代码是向浏览器端输出请求的路径，类似一个echo服务器。接下来我们为其添加输出对应文件的功能。

## 读取静态文件

为了不让用户在浏览器端通过请求/app.js查看到我们的代码，我们设定用户只能请求assets目录下的文件。服务器会将路径信息映射到assets目录。

涉及到了文件读取的这部分，自然不能避开fs(file system)这个模块。同样，涉及到了路径处理，path模块也是需要的。

我们通过path模块的path.exists方法来判断静态文件是否存在磁盘上。不存在我们直接响应给客户端404错误。

如果文件存在则调用fs.readFile方法读取文件。如果发生错误，我们响应给客户端500错误，表明存在内部错误。正常状态下则发送读取到的文件给客户端，表明200状态。

var server = http.createServer(function (request, response) {

var pathname = url.parse(request.url).pathname;

var realPath = "assets" + pathname;

path.exists(realPath, function (exists) {

if (!exists) {

response.writeHead(404, {

'Content-Type': 'text/plain'

});

response.write("This request URL " + pathname + " was not found on this server.");

response.end();

} else {

fs.readFile(realPath, "binary", function (err, file) {

if (err) {

response.writeHead(500, {

'Content-Type': 'text/plain'

});

response.end(err);

} else {

response.writeHead(200, {

'Content-Type': 'text/html'

});

response.write(file, "binary");

response.end();

}

});

}

});

});

以上这段简单的代码加上一个assets目录，就构成了我们最基本的静态文件服务器。

那么眼尖的你且看看，这个最基本的静态文件服务器存在哪些问题呢？答案是MIME类型支持。因为我们的服务器同时要存放html, css, js, png, gif, jpg等等文件。并非每一种文件的MIME类型都是text/html的。

## MIME类型支持

像其他服务器一样，支持MIME的话，就得一张映射表。

exports.types = {

"css": "text/css",

"gif": "image/gif",

"html": "text/html",

"ico": "image/x-icon",

"jpeg": "image/jpeg",

"jpg": "image/jpeg",

"js": "text/javascript",

"json": "application/json",

"pdf": "application/pdf",

"png": "image/png",

"svg": "image/svg+xml",

"swf": "application/x-shockwave-flash",

"tiff": "image/tiff",

"txt": "text/plain",

"wav": "audio/x-wav",

"wma": "audio/x-ms-wma",

"wmv": "video/x-ms-wmv",

"xml": "text/xml"

};

以上代码另存在mime.js文件中。该文件仅仅只列举了一些常用的MIME类型，以文件后缀作为key，MIME类型为value。那么引入mime.js文件吧。

var mime = require("./mime").types;

我们通过path.extname来获取文件的后缀名。由于extname返回值包含”.”，所以通过slice方法来剔除掉”.”，对于没有后缀名的文件，我们一律认为是unknown。

var ext = path.extname(realPath);

ext = ext ? ext.slice(1) : 'unknown';

接下来我们很容易得到真正的MIME类型了。

var contentType = mime[ext] || "text/plain";

response.writeHead(200, {'Content-Type': contentType});

response.write(file, "binary");

response.end();

对于未知的类型，我们一律返回text/plain类型。

## 缓存支持/控制

在MIME支持之后，静态文件服务器看起来已经很完美了。任何静态文件只要丢进assets目录之后就可以万事大吉不管了。看起来已经达到了Apache作为静态文件服务器的相同效果了。我们实现这样的服务器用的代码只有这么多行而已。是不是很简单呢？

但是，我们发现用户在每次请求的时候，服务器每次都要调用fs.readFile方法去读取硬盘上的文件的。当服务器的请求量一上涨，硬盘IO会吃不消。

在解决这个问题之前，我们有必要了解一番前端浏览器缓存的一些机制和提高性能的方案。

1. GZip压缩文件可以减少响应的大小，能够达到节省带宽的目的。
2. 浏览器缓存中存有文件副本的时候，不能确定有效的时候，会生成一个条件get请求。
   1. 在请求的头中会包含 If-Modified-Since。
   2. 如果服务器端文件在这个时间后发生过修改，则发送整个文件给前端。
   3. 如果没有修改，则返回304状态码。并不发送整个文件给前端。
   4. 另外一种判断机制是ETag。在此并不讨论。
3. 如果副本有效，这个get请求都会省掉。判断有效的最主要的方法是服务端响应的时候带上Expires的头。
   1. 浏览器会判断Expires头，直到制定的日期过期，才会发起新的请求。
   2. 另一个可以达到相同目的的方法是返回Cache-Control: max-age=xxxx。

欲了解更多缓存机制，请参见Steve Sounders著作的《高性能网站建设指南》。

为了简化问题，我们只做如下这几件事情：

1. 为指定几种后缀的文件，在响应时添加Expires头和Cache-Control: max-age头。超时日期设置为1年。
2. 由于这是静态文件服务器，为所有请求，响应时返回Last-Modified头。
3. 为带If-Modified-Since的请求头，做日期检查，如果没有修改，则返回304。若修改，则返回文件。

对于以上的静态文件服务器，Node给的响应头是十分简单的：

Connection: keep-alive

Content-Type: text/html

Transfer-Encoding: chunked

对于指定后缀文件和过期日期，为了保证可配置。那么建立一个config.js文件是应该的。

exports.Expires = {

fileMatch: /^(gif|png|jpg|js|css)$/ig,

maxAge: 60 \* 60 \* 24 \* 365

};

引入config.js文件。

var config = require("./config");

我们在相应之前判断后缀名是否符合我们要添加过期时间头的条件。

var ext = path.extname(realPath);

ext = ext ? ext.slice(1) : 'unknown';

if (ext.match(config.Expires.fileMatch)) {

var expires = new Date();

expires.setTime(expires.getTime() + config.Expires.maxAge \* 1000);

response.setHeader("Expires", expires.toUTCString());

response.setHeader("Cache-Control", "max-age=" + config.Expires.maxAge);

}

这次的响应头中多了两个header。

Cache-Control: max-age=31536000

Connection: keep-alive

Content-Type: image/png

Expires: Fri, 09 Nov 2012 12:55:41 GMT

Transfer-Encoding: chunked

浏览器在发送请求之前由于检测到Cache-Control和Expires（Cache-Control的优先级高于Expires，但有的浏览器不支持Cache-Control，这时采用Expires），如果没有过期，则不会发送请求，而直接从缓存中读取文件。

接下来我们为所有请求的响应都添加Last-Modified头。

读取文件的最后修改时间是通过fs模块的fs.stat()方法来实现的。关于stat的详细介绍请参见[此处](http://www.cnitblog.com/guopingleee/archive/2008/11/13/51411.aspx)。

fs.stat(realPath, function (err, stat) {

var lastModified = stat.mtime.toUTCString();

response.setHeader("Last-Modified", lastModified);

});

我们同时也要检测浏览器是否发送了If-Modified-Since请求头。如果发送而且跟文件的修改时间相同的话，我们返回304状态。

if (request.headers[ifModifiedSince] && lastModified == request.headers[ifModifiedSince]) {

response.writeHead(304, "Not Modified");

response.end();

}

如果没有发送或者跟磁盘上的文件修改时间不相符合，则发送回磁盘上的最新文件。

通过Expires和Last-Modified两个方案以及与浏览器之间的通力合作，会节省相当大的一部分网络流量，同时也会降低部分硬盘IO的请求。如果在这之前还存在CDN的话，整个方案就比较完美了。

由于Expires和Max-Age都是由浏览器来进行判断的，如果判断成功，http请求都不会发送到服务端的，这里只能通过fiddler和浏览器配合进行测试。但是Last-Modified却是可以通过curl来进行测试的。

#:~$ curl --header "If-Modified-Since: Fri, 11 Nov 2011 19:14:51 GMT" -i http://localhost:8000

HTTP/1.1 304 Not Modified

Content-Type: text/html

Last-Modified: Fri, 11 Nov 2011 19:14:51 GMT

Connection: keep-alive

注意，我们看到这个304请求的响应是不带body信息的。所以，达到我们节省带宽的需求。只需几行代码，就可以省下许多的带宽费用。

但是，貌似我们有提到gzip这样的东西。对于CSS、JS等文件如果不采用GZip的话，还是会浪费掉部分网络带宽。那么接下来把GZip代码添加进来。

### GZip启用

如果你是前端达人，你应该是知道YUI Compressor或Google Closure Complier这样的压缩工具的。在这基础上，再进行gzip压缩，则会减少很多的网络流量。那么，我们看看Node中，怎么把gzip搞起来。

要用到gzip，就需要zlib模块，该模块在Node的0.5.8版本开始原生支持。

var zlib = require("zlib");

对于图片一类的文件，不需要进行gzip压缩，所以我们在config.js中配置一个启用压缩的列表。

exports.Compress = {

match: /css|js|html/ig

};

这里为了防止大文件，也为了满足zlib模块的调用模式，将读取文件改为流的形式进行读取。

var raw = fs.createReadStream(realPath);

var acceptEncoding = request.headers['accept-encoding'] || "";

var matched = ext.match(config.Compress.match);

if (matched && acceptEncoding.match(/\bgzip\b/)) {

response.writeHead(200, "Ok", {

'Content-Encoding': 'gzip'

});

raw.pipe(zlib.createGzip()).pipe(response);

} else if (matched && acceptEncoding.match(/\bdeflate\b/)) {

response.writeHead(200, "Ok", {

'Content-Encoding': 'deflate'

});

raw.pipe(zlib.createDeflate()).pipe(response);

} else {

response.writeHead(200, "Ok");

raw.pipe(response);

}

对于支持压缩的文件格式以及浏览器端接受gzip或deflate压缩，我们调用压缩。若不，则管道方式转发给response。

启用压缩其实就这么简单。如果你有fiddler的话，可以监听一下请求，会看到被压缩的请求。

## 安全问题

我们搞了一大堆的事情，但是安全方面也不能少。想想哪一个地方是最容易出问题的？

我们发现上面的这段代码写得还是有点纠结的，通常这样纠结的代码我是不愿意拿出去让人看见的。但是，假如一个同学用浏览器访问http://localhost:8000/../app.js 怎么办捏？

不用太害怕，浏览器会自动干掉那两个作为父路径的点的。浏览器会把这个路径组装成http://localhost:8000/app.js的，这个文件在assets目录下不存在，返回404 Not Found。

但是聪明一点的同学会通过curl -i http://localhost:8000/../app.js 来访问。于是，问题出现了。

# curl -i http://localhost:8000/../app.js

HTTP/1.1 200 Ok

Content-Type: text/javascript

Last-Modified: Thu, 10 Nov 2011 17:16:51 GMT

Expires: Sat, 10 Nov 2012 04:59:27 GMT

Cache-Control: max-age=31536000

Connection: keep-alive

Transfer-Encoding: chunked

var PORT = 8000;

var http = require("http");

var url = require("url");

var fs = require("fs");

var path = require("path");

var mime = require("./mime").types;

那么怎么办呢？暴力点的解决方案就是禁止父路径。

首先替换掉所有的..，然后调用path.normalize方法来处理掉不正常的/。

var realPath = path.join("assets", path.normalize(pathname.replace(/\.\./g, "")));

于是这个时候通过curl -i http://localhost:8000/../app.js 访问，/../app.js会被替换掉为//app.js。normalize方法会将//app.js返回为/app.js。再加上真实的assets，就被实际映射为assets/app.js。这个文件不存在，于是返回404。搞定父路径问题。与浏览器的行为保持一致。

## Welcome页的锦上添花

再来回忆一下Apache的常见行为。当进入一个目录路径的时候，会去寻找index.html页面，如果index.html文件不存在，则返回目录索引。目录索引这里我们暂不考虑，如果用户请求的路径是/结尾的，我们就自动为其添加上index.html文件。如果这个文件不存在，继续返回404错误。

如果用户请求了一个目录路径，而且没有带上/。那么我们为其添加上/index.html，再重新做解析。

那么不喜欢硬编码的你，肯定是要把这个文件配置进config.js。这样你就可以选择各种后缀作为welcome页面。

exports.Welcome = {

file: "index.html"

};

那么第一步，为/结尾的请求，自动添加上”index.html”。

if (pathname.slice(-1) === "/") {

pathname = pathname + config.Welcome.file;

}

第二步，如果请求了一个目录路径，并且没有以/结尾。那么我们需要做判断。如果当前读取的路径是目录，就需要添加上/和index.html

if (stats.isDirectory()) {

realPath = path.join(realPath, "/", config.Welcome.file);

}

由于我们目前的结构发生了一点点变化。所以需要重构一下函数。而且，fs.stat方法具有比fs.exsits方法更多的功能。我们直接替代掉它。

就这样。一个各方面都比较完整的静态文件服务器就这样打造完毕。

**Range支持，搞定媒体断点支持**

关于http1.1中的Range定义，可以参见这两篇文章：

* <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>
* <http://labs.apache.org/webarch/http/draft-fielding-http/p5-range.html>

接下来，我将简单地介绍一下range的作用和其定义。

当用户在听一首歌的时候，如果听到一半（网络下载了一半），网络断掉了，用户需要继续听的时候，文件服务器不支持断点的话，则用户需要重新下载这个文件。而Range支持的话，客户端应该记录了之前已经读取的文件范围，网络恢复之后，则向服务器发送读取剩余Range的请求，服务端只需要发送客户端请求的那部分内容，而不用整个文件发送回客户端，以此节省网络带宽。

那么HTTP1.1规范的Range是怎样一个约定呢。

1. 如果Server支持Range，首先就要告诉客户端，咱支持Range，之后客户端才可能发起带Range的请求。

response.setHeader('Accept-Ranges', 'bytes');

1. Server通过请求头中的Range: bytes=0-xxx来判断是否是做Range请求，如果这个值存在而且有效，则只发回请求的那部分文件内容，响应的状态码变成206，表示Partial Content，并设置Content-Range。如果无效，则返回416状态码，表明Request Range Not Satisfiable（<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.17>）。如果不包含Range的请求头，则继续通过常规的方式响应。
2. 有必要对Range请求做一下解释。

ranges-specifier = byte-ranges-specifier

byte-ranges-specifier = bytes-unit "=" byte-range-set

byte-range-set = 1#( byte-range-spec | suffix-byte-range-spec )

byte-range-spec = first-byte-pos "-" [last-byte-pos]

first-byte-pos = 1\*DIGIT

last-byte-pos = 1\*DIGIT

上面这段定义来自w3定义的协议<http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.35>。大致可以表述为Range: bytes=[start]-[end][,[start]-[end]]。简言之有以下几种情况：

bytes=0-99，从0到99之间的数据字节。

bytes=-100，文件的最后100个字节。

bytes=100-，第100个字节开始之后的所有字节。

bytes=0-99,200-299，从0到99之间的数据字节和200到299之间的数据字节。

那么，我们就开始实现吧。首先判断Range请求和检测其是否有效。为了保持代码干净，我们封装一个parseRange方法，这个方法属于util性质的，那么我们放进utils.js文件。

var utils = require("./utils");

我们暂且不支持多区间。于是遇见逗号，就报416错误。

exports.parseRange = function (str, size) {

if (str.indexOf(",") != -1) {

return;

}

var range = str.split("-"),

start = parseInt(range[0], 10),

end = parseInt(range[1], 10);

// Case: -100

if (isNaN(start)) {

start = size - end;

end = size - 1;

// Case: 100-

} else if (isNaN(end)) {

end = size - 1;

}

// Invalid

if (isNaN(start) || isNaN(end) || start > end || end > size) {

return;

}

return {

start: start,

end: end

};

};

如果满足Range的条件，则为响应添加上Content-Range和修改掉Content-Lenth。

response.setHeader("Content-Range", "bytes " + range.start + "-" + range.end + "/" + stats.size);

response.setHeader("Content-Length", (range.end - range.start + 1));

非常开心的一件事情是，Node的读文件流，原生支持range读取。

var raw = fs.createReadStream(realPath, {"start": range.start, "end": range.end});

设置状态码为206。

由于选取Range之后，依然还是需要经过GZip的。于是代码已经有点面条的味道了。重构一下吧。于是代码大致如此：

var compressHandle = function (raw, statusCode, reasonPhrase) {

var stream = raw;

var acceptEncoding = request.headers['accept-encoding'] || "";

var matched = ext.match(config.Compress.match);

if (matched && acceptEncoding.match(/\bgzip\b/)) {

response.setHeader("Content-Encoding", "gzip");

stream = raw.pipe(zlib.createGzip());

} else if (matched && acceptEncoding.match(/\bdeflate\b/)) {

response.setHeader("Content-Encoding", "deflate");

stream = raw.pipe(zlib.createDeflate());

}

response.writeHead(statusCode, reasonPhrase);

stream.pipe(response);

};

if (request.headers["range"]) {

var range = utils.parseRange(request.headers["range"], stats.size);

if (range) {

response.setHeader("Content-Range", "bytes " + range.start + "-" + range.end + "/" + stats.size);

response.setHeader("Content-Length", (range.end - range.start + 1));

var raw = fs.createReadStream(realPath, {

"start": range.start,

"end": range.end

});

compressHandle(raw, 206, "Partial Content");

} else {

response.removeHeader("Content-Length");

response.writeHead(416, "Request Range Not Satisfiable");

response.end();

}

} else {

var raw = fs.createReadStream(realPath);

compressHandle(raw, 200, "Ok");

}

通过curl --header "Range:0-20" -i http://localhost:8000/index.html请求测试一番试试。

HTTP/1.1 206 Partial Content

Server: Node/V5

Accept-Ranges: bytes

Content-Type: text/html

Content-Length: 21

Last-Modified: Fri, 11 Nov 2011 19:14:51 GMT

Content-Range: bytes 0-20/54

Connection: keep-alive

<html>

<body>

<h1>I

index.html文件并没有被整个发送给客户端。这里之所以没有完全的21个字节，是因为\t和\r都各算一个字节。

再用curl --header "Range:0-100" -i http://localhost:8000/index.html反向测试一下吧。

HTTP/1.1 416 Request Range Not Satisfiable

Server: Node/V5

Accept-Ranges: bytes

Content-Type: text/html

Last-Modified: Fri, 11 Nov 2011 19:14:51 GMT

Connection: keep-alive

Transfer-Encoding: chunked

嗯，要的就是这个效果。至此，Range支持完成，这个静态文件服务器支持一些流媒体文件。

嗯。就这么简单。

# 模块

## fs 模块

### fs.readFile

fs.readFile(filename,[encoding],[callback(err,data)])是最简单的读取文件的函数。它接受一个必选参数 filename，表示要读取的文件名。第二个参数 encoding是可选的，表示文件的字符编码。callback 是回调函数，用于接收文件的内容。如果不指定 encoding，则 callback 就是第二个参数。回调函数提供两个参数 err 和 data，err 表示有没有错误发生，data 是文件内容。如果指定了 encoding，data 是一个解析后的字符串，否则 data 将会是以 Buffer 形式表示的二进制数据。

不指定编码方式就默认使用二进制的方式进行读取数据。

var fs = require('fs');

fs.readFile('content.txt', function(err, data) {

if (err) {

console.error(err);

} else {

console.log(data);

}

});

### fs.readFileSync

fs.readFileSync(filename, [encoding])是 fs.readFile 同步的版本。它接受的参数和 fs.readFile 相同，而读取到的文件内容会以函数返回值的形式返回。如果有错误发生，fs 将会抛出异常，你需要使用 try 和 catch 捕捉并处理异常。

### fs.open

fs.open(path, flags, [mode], [callback(err, fd)])是 POSIX open 函数的封装，与 C 语言标准库中的 fopen 函数类似。它接受两个必选参数，path 为文件的路径，flags 可以是以下值。  
θ r ：以读取模式打开文件。  
θ r+ ：以读写模式打开文件。  
θ w ：以写入模式打开文件，如果文件不存在则创建。  
θ w+ ：以读写模式打开文件，如果文件不存在则创建。  
θ a ：以追加模式打开文件，如果文件不存在则创建。  
θ a+ ：以读取追加模式打开文件，如果文件不存在则创建。  
mode 参数用于创建文件时给文件指定权限，默认是 0666①。回调函数将会传递一个文件描述符 fd②。

### fs.read

fs.read(fd, buffer, offset, length, position, [callback(err, bytesRead,buffer)])是 POSIX read 函数的封装，相比fs.readFile 提供了更底层的接口。fs.read的功能是从指定的文件描述符 fd 中读取数据并写入 buffer 指向的缓冲区对象。offset 是buffer 的写入偏移量。length 是要从文件中读取的字节数。position 是文件读取的起始位置，如果 position 的值为 null，则会从当前文件指针的位置读取。回调函数传递bytesRead 和 buffer，分别表示读取的字节数和缓冲区对象。

实例:

var fs = require('fs');

fs.open('content.txt', 'r', function(err, fd) {

if (err) {

console.error(err);

return;

}

var buf = new Buffer(8);

fs.read(fd, buf, 0, 8, null, function(err, bytesRead, buffer) {

if (err) {

console.error(err);

return;

}

console.log('bytesRead: ' + bytesRead);

console.log(buffer);

})

});

