**而对于需要独立运行的JS，NodeJS就是一个解析器。**

#### require

**模块名可使用相对路径（以**./**开头），或者是绝对路径（以**/**或**C:**之类的盘符开头）**。另外，**模块名中的**.js**扩展名可以省略**。

var foo1 = require('./foo');

var foo2 = require('./foo.js');

var foo3 = require('/home/user/foo');

var foo4 = require('/home/user/foo.js');

另外，可以使用以下方式加载和使用一个JSON文件。

var data = require('./data.json');

#### exports

exports**对象是当前模块的导出对象**，用于**导出模块公有方法和属性**。**别的模块通过**require**函数使用当前模块时得到的就是当前模块的**exports**对象**。

exports.hello = function () {

console.log('Hello World!');

};

#### 模块初始化

一个模块中的JS代码仅在模块第一次被使用时执行一次，并在执行过程中初始化模块的导出对象。之后，**缓存起来的导出对象被重复利用**。

var counter1 = require('./util/counter');

var counter2 = require('./util/counter');

console.log(counter1.count());

console.log(counter2.count());

console.log(counter2.count());

运行该程序的结果如下：

$ node main.js

1

2

3

可以看到，counter.js并没有因为被require了两次而初始化两次。

1. 内置模块

**如果传递给**require**函数的是NodeJS内置模块名称，不做路径解析，直接返回内部模块的导出对象，例如**require('fs')**。**

2. node\_modules目录

也能像内置模块那么写，因为它会自动添加这个路径

3. NODE\_PATH环境变量

与PATH环境变量类似，NodeJS允许通过NODE\_PATH环境变量来指定额外的模块搜索路径。NODE\_PATH环境变量中包含一到多个目录路径，路径之间在Linux下使用:分隔，**在Windows下使用**;**分隔**。例如定义了以下**NODE\_PATH环境变量：**

NODE\_PATH=/home/user/lib:/home/lib

当使用require('foo/bar')的方式加载模块时，则NodeJS依次尝试以下路径。

/home/user/lib/foo/bar

/home/lib/foo/bar

#### index.js

**当模块的文件名是**index.js**，加载模块时可以使用模块所在目录的路径代替模块文件路径**，因此接着上例，**以下两条语句等价**。

#### package.json

如果想**自定义入口模块的文件名和存放位置，就需要在包目录下包含一个**package.json**文件**，并在其中指定入口模块的路径。

* **编写代码前先规划好目录结构，才能做到有条不紊。**
* **稍大些的程序可以将代码拆分为多个模块管理，更大些的程序可以使用包来组织模块。**
* **合理使用**node\_modules**和**NODE\_PATH**来解耦包的使用方式和物理路径。**
* **使用NPM加入NodeJS生态圈互通有无。**

#### 小文件拷贝

我们使用NodeJS内置的fs模块简单实现这个程序如下。

var fs = require('fs');

function copy(src, dst) {

fs.writeFileSync(dst, fs.readFileSync(src));

}

function main(argv) {

copy(argv[0], argv[1]);

}

main(process.argv.slice(2));

**以上程序使用**fs.readFileSync**从源路径读取文件内容，并使用**fs.writeFileSync**将文件内容写入目标路径。**

**豆知识：** process**是一个全局变量，可通过**process.argv**获得命令行参数。由于**argv[0]**固定等于NodeJS执行程序的绝对路径，**argv[1]**固定等于主模块的绝对路径，因此第一个命令行参数从**argv[2]**这个位置开始。**

#### 大文件拷贝

上边的程序拷贝一些小文件没啥问题，但这种一次性把所有文件内容都读取到内存中后再一次性写入磁盘的方式不适合拷贝大文件，内存会爆仓。对于大文件，我们只能读一点写一点，直到完成拷贝。因此上边的程序需要改造如下。

var fs = require('fs');

function copy(src, dst) {

fs.createReadStream(src).pipe(fs.createWriteStream(dst));

}

function main(argv) {

copy(argv[0], argv[1]);

}

main(process.argv.slice(2));

以上程序使用fs.createReadStream创建了一个源文件的只读数据流，并使用fs.createWriteStream创建了一个目标文件的只写数据流，并且用pipe方法把两个数据流连接了起来。连接起来后发生的事情，说得抽象点的话，水顺着水管从一个桶流到了另一个桶。

Stream（数据流）

**当内存中无法一次装下需要处理的数据时，或者一边读取一边处理更加高效时，我们就需要用到数据流。NodeJS中通过各种**Stream**来提供对数据流的操作。**

数据从只读数据流到只写数据流的搬运，并包括了防爆仓控制。因为这种使用场景很多，例如上边的大文件拷贝程序，**NodeJS直接提供了**.pipe**方法来做这件事情**

chunk可以理解为流数据中的一块数据，对可读流它是Stream读取的数据块，对可写流它是Stream写入的数据块。  
chunk数据块类型有两种模式，一个是常规模式regularMode，一个是objectMode，在创建Stream时可以指定objectMode。这里说明一下两种模式的区别：在regularMode时，chunk只能是String, Buffer, Null, Undefined类型；而在objectMode时，chunk可以是任意类型。值得注意的是，regularMode消耗的是**字节**，而objectMode消耗的是**object**。

#### File System（文件系统）

NodeJS通过fs内置模块提供对文件的操作。fs模块提供的API基本上可以分为以下三类：

* **文件属性读写。**

**其中常用的有**fs.stat**、**fs.chmod**、**fs.chown**等等。**

* **文件内容读写。**

**其中常用的有**fs.readFile**、**fs.readdir**、**fs.writeFile**、**fs.mkdir**等等。**

* **底层文件操作。**

**其中常用的有**fs.open**、**fs.read**、**fs.write**、**fs.close**等等。**

NodeJS最精华的异步IO模型在fs模块里有着充分的体现，例如上边提到的这些API都通过回调函数传递结果。以fs.readFile为例：

fs.readFile(pathname, function (err, data) {

if (err) {

// Deal with error.

} else {

// Deal with data.

}

});

如上边代码所示，**基本上所有**fs**模块API的回调参数都有两个。第一个参数在有错误发生时等于异常对象，第二个参数始终用于返回API方法执行结果。**

Path（路径）

* path.normalize

将传入的路径转换为标准路径，具体讲的话，除了解析路径中的.与..外，还能去掉多余的斜杠。如果有程序需要使用路径作为某些数据的索引，但又允许用户随意输入路径时，就需要使用该方法保证路径的唯一性。以下是一个例子：

var cache = {};

function store(key, value) {

cache[path.normalize(key)] = value;

}

store('foo/bar', 1);

store('foo//baz//../bar', 2);

console.log(cache); // => { "foo/bar": 2 }

**坑出没注意：** 标准化之后的路径里的斜杠在Windows系统下是\，而在Linux系统下是/。如果想保证任何系统下都使用/作为路径分隔符的话，需要用.replace(/\\/g, '/')再替换一下标准路径。

* path.join

将传入的多个路径拼接为标准路径。该方法可避免手工拼接路径字符串的繁琐，并且能在不同系统下正确使用相应的路径分隔符。以下是一个例子：

path.join('foo/', 'baz/', '../bar'); // => "foo/bar"

* path.extname

当我们需要根据不同文件扩展名做不同操作时，该方法就显得很好用。以下是一个例子：

path.extname('foo/bar.js'); // => ".js"

#### 同步遍历

了解了必要的算法后，我们可以简单地实现以下目录遍历函数。

function travel(dir, callback) {

fs.readdirSync(dir).forEach(function (file) {

var pathname = path.join(dir, file);

if (fs.statSync(pathname).isDirectory()) {

travel(pathname, callback);

} else {

callback(pathname);

}

});

}

可以看到，该函数以某个目录作为遍历的起点。遇到一个子目录时，就先接着遍历子目录。遇到一个文件时，就把文件的绝对路径传给回调函数。回调函数拿到文件路径后，就可以做各种判断和处理。因此假设有以下目录：

- /home/user/

- foo/

x.js

- bar/

y.js

z.css

使用以下代码遍历该目录时，得到的输入如下。

travel('/home/user', function (pathname) {

console.log(pathname);

});

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

/home/user/foo/x.js

/home/user/bar/y.js

/home/user/z.css

#### 异步遍历

如果读取目录或读取文件状态时使用的是异步API，目录遍历函数实现起来会有些复杂，但原理完全相同。travel函数的异步版本如下。

function travel(dir, callback, finish) {

fs.readdir(dir, function (err, files) {

(function next(i) {

if (i < files.length) {

var pathname = path.join(dir, files[i]);

fs.stat(pathname, function (err, stats) {

if (stats.isDirectory()) {

travel(pathname, callback, function () {

next(i + 1);

});

} else {

callback(pathname, function () {

next(i + 1);

});

}

});

} else {

finish && finish();

}

}(0));

});

}

这里不详细介绍异步遍历函数的编写技巧，在后续章节中会详细介绍这个。总之我们可以看到异步编程还是蛮复杂的。

## 网络操作

使用NodeJS内置的http模块简单实现一个HTTP服务器。

var http = require('http');

http.createServer(function (request, response) {

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

response.end('Hello World\n');

}).listen(8124);

以上程序**创建了一个HTTP服务器并监听**8124**端口，打开浏览器访问该端口**http://127.0.0.1:8124/**就能够看到效果。**

#### *HTTP*

'http'模块提供两种使用方式：

* 作为服务端使用时，创建一个HTTP服务器，监听HTTP客户端请求并返回响应。
* 作为客户端使用时，发起一个HTTP客户端请求，获取服务端响应。

如开门红中的例子所示，首先需要使用.createServer方法创建一个服务器，然后调用.listen方法监听端口。之后，每当来了一个客户端请求，创建服务器时传入的回调函数就被调用一次。可以看出，这是一种事件机制。

还能把request对象当作一个只读数据流来访问请求体数据。以下是一个例子。

http.createServer(function (request, response) {

var body = [];

console.log(request.method);

console.log(request.headers);

request.on('data', function (chunk) {

body.push(chunk);

});

request.on('end', function () {

body = Buffer.concat(body);

console.log(body.toString());

});

}).listen(80);

HTTP响应本质上也是一个数据流，同样由响应头（headers）和响应体（body）组成。

在回调函数中，除了可以使用response对象来写入响应头数据外，还能把response对象当作一个只写数据流来写入响应体数据。例如在以下例子中，服务端原样将客户端请求的请求体数据返回给客户端。

http.createServer(function (request, response) {

response.writeHead(200, { 'Content-Type': 'text/plain' });

request.on('data', function (chunk) {

response.write(chunk);

});

request.on('end', function () {

response.end();

});

}).listen(80);

接下来我们看看客户端模式下如何工作。为了发起一个客户端HTTP请求，我们需要指定目标服务器的位置并发送请求头和请求体，以下示例演示了具体做法。

var options = {

hostname: 'www.example.com',

port: 80,

path: '/upload',

method: 'POST',

headers: {

'Content-Type': 'application/x-www-form-urlencoded'

}

};

var request = http.request(options, function (response) {});

request.write('Hello World');

request.end();

可以看到，.request方法创建了一个客户端，并指定请求目标和请求头数据。之后，就可以把request对象当作一个只写数据流来写入请求体数据和结束请求。另外，由于HTTP请求中GET请求是最常见的一种，并且不需要请求体，因此http模块也提供了以下便捷API。

http.get('http://www.example.com/', function (response) {});

当客户端发送请求并接收到完整的服务端响应头时，就会调用回调函数。在回调函数中，除了可以使用response对象访问响应头数据外，还能把response对象当作一个只读数据流来访问响应体数据。以下是一个例子。

http.get('http://www.example.com/', function (response) {

var body = [];

console.log(response.statusCode);

console.log(response.headers);

response.on('data', function (chunk) {

body.push(chunk);

});

response.on('end', function () {

body = Buffer.concat(body);

console.log(body.toString());

});

});

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

200

{ 'content-type': 'text/html',

server: 'Apache',

'content-length': '801',

date: 'Tue, 05 Nov 2013 06:08:41 GMT',

connection: 'keep-alive' }

<!DOCTYPE html>

#### *HTTPS*

**官方文档：**<http://nodejs.org/api/https.html>

https模块与http模块极为类似，区别在于https模块需要额外处理SSL证书。

在服务端模式下，创建一个HTTPS服务器的示例如下。

var options = {

key: fs.readFileSync('./ssl/default.key'),

cert: fs.readFileSync('./ssl/default.cer')

};

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

// ...

});

可以看到，与创建HTTP服务器相比，多了一个options对象，通过key和cert字段指定了HTTPS服务器使用的私钥和公钥。

另外，NodeJS支持SNI技术，可以根据HTTPS客户端请求使用的域名动态使用不同的证书，因此同一个HTTPS服务器可以使用多个域名提供服务。接着上例，可以使用以下方法为HTTPS服务器添加多组证书。

server.addContext('foo.com', {

key: fs.readFileSync('./ssl/foo.com.key'),

cert: fs.readFileSync('./ssl/foo.com.cer')

});

server.addContext('bar.com', {

key: fs.readFileSync('./ssl/bar.com.key'),

cert: fs.readFileSync('./ssl/bar.com.cer')

});

在客户端模式下，发起一个HTTPS客户端请求与http模块几乎相同，示例如下。

var options = {

hostname: 'www.example.com',

port: 443,

path: '/',

method: 'GET'

};

var request = https.request(options, function (response) {});

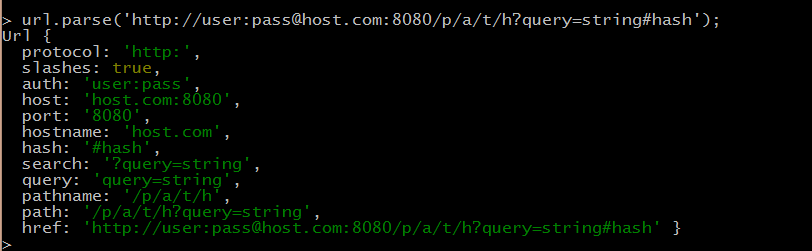
request.end();

但如果目标服务器使用的SSL证书是自制的，不是从颁发机构购买的，默认情况下https模块会拒绝连接，提示说有证书安全问题。在options里加入rejectUnauthorized: false字段可以禁用对证书有效性的检查，从而允许https模块请求开发环境下使用自制证书的HTTPS服务器。

#### *URL*

处理HTTP请求时url模块使用率超高，因为该模块允许解析URL、生成URL，以及拼接URL。

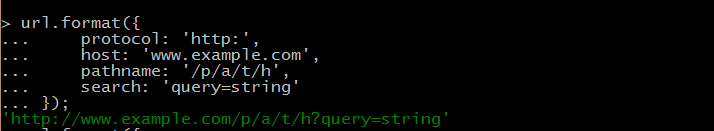
我们可以使用.parse方法来将一个URL字符串转换为URL对象



传给.parse方法的不一定要是一个完整的URL，例如在HTTP服务器回调函数中，request.url不包含协议头和域名，但同样可以用.parse方法解析。

.parse方法还支持第二个和第三个布尔类型可选参数。第二个参数等于true时，该方法返回的URL对象中，query字段不再是一个字符串，而是一个经过querystring模块转换后的参数对象。第三个参数等于true时，该方法可以正确解析不带协议头的URL，例如//www.example.com/foo/bar。

反过来，format方法允许将一个URL对象转换为URL字符串，示例如下。



另外，.resolve方法可以用于拼接URL，示例如下。

#### *Query String*

querystring模块用于实现URL参数字符串与参数对象的互相转换，示例如下。

querystring.parse('foo=bar&baz=qux&baz=quux&corge');

/\* =>

{ foo: 'bar', baz: ['qux', 'quux'], corge: '' }

\*/

querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' });

/\* =>

'foo=bar&baz=qux&baz=quux&corge='

\*/

#### *Zlib*

zlib模块提供了数据压缩和解压的功能。当我们处理HTTP请求和响应时，可能需要用到这个模块。

首先我们看一个使用zlib模块压缩HTTP响应体数据的例子。这个例子中，判断了客户端是否支持gzip，并在支持的情况下使用zlib模块返回gzip之后的响应体数据。

http.createServer(function (request, response) {

var i = 1024,

data = '';

while (i--) {

data += '.';

}

if ((request.headers['accept-encoding'] || '').indexOf('gzip') !== -1) {

zlib.gzip(data, function (err, data) {

response.writeHead(200, {

'Content-Type': 'text/plain',

'Content-Encoding': 'gzip'

});

response.end(data);

});

} else {

response.writeHead(200, {

'Content-Type': 'text/plain'

});

response.end(data);

}

}).listen(80);

接着我们看一个使用zlib模块解压HTTP响应体数据的例子。这个例子中，判断了服务端响应是否使用gzip压缩，并在压缩的情况下使用zlib模块解压响应体数据。

var options = {

hostname: 'www.example.com',

port: 80,

path: '/',

method: 'GET',

headers: {

'Accept-Encoding': 'gzip, deflate'

}

};

http.request(options, function (response) {

var body = [];

response.on('data', function (chunk) {

body.push(chunk);

});

response.on('end', function () {

body = Buffer.concat(body);

if (response.headers['content-encoding'] === 'gzip') {

zlib.gunzip(body, function (err, data) {

console.log(data.toString());

});

} else {

console.log(data.toString());

}

});

}).end();

#### *Net*

**官方文档：**<http://nodejs.org/api/net.html>

net模块可用于创建Socket服务器或Socket客户端。由于Socket在前端领域的使用范围还不是很广，这里先不涉及到WebSocket的介绍，仅仅简单演示一下如何从Socket层面来实现HTTP请求和响应。

net.createServer(function (conn) {

conn.on('data', function (data) {

conn.write([

'HTTP/1.1 200 OK',

'Content-Type: text/plain',

'Content-Length: 11',

'',

'Hello World'

].join('\n'));

});

}).listen(80);

接着我们来看一个使用Socket发起HTTP客户端请求的例子。这个例子中，Socket客户端在建立连接后发送了一个HTTP GET请求，并通过data事件监听函数来获取服务器响应。

var options = {

port: 80,

host: 'www.example.com'

};

var client = net.connect(options, function () {

client.write([

'GET / HTTP/1.1',

'User-Agent: curl/7.26.0',

'Host: www.baidu.com',

'Accept: \*/\*',

'',

''

].join('\n'));

});

client.on('data', function (data) {

console.log(data.toString());

client.end();

});

* 问： 为什么http模块创建的HTTP服务器返回的响应是chunked传输方式的？

答： 因为默认情况下，使用.writeHead方法写入响应头后，允许使用.write方法写入任意长度的响应体数据，并使用.end方法结束一个响应。由于响应体数据长度不确定，因此NodeJS自动在响应头里添加了Transfer-Encoding: chunked字段，并采用chunked传输方式。但是当响应体数据长度确定时，可使用.writeHead方法在响应头里加上Content-Length字段，这样做之后NodeJS就不会自动添加Transfer-Encoding字段和使用chunked传输方式。

* http和https模块支持服务端模式和客户端模式两种使用方式。
* request和response对象除了用于读写头数据外，都可以当作数据流来操作。
* url.parse方法加上request.url属性是处理HTTP请求时的固定搭配。
* 使用zlib模块可以减少使用HTTP协议时的数据传输量。
* 通过net模块的Socket服务器与客户端可对HTTP协议做底层操作。
* 小心踩坑。

## 进程管理

NodeJS可以**感知和控制自身进程的运行环境和状态**，也可以**创建子进程并与其协同工作**，这使得NodeJS可以把多个程序组合在一起共同完成某项工作，并在其中充当**胶水和调度器**的作用。

NodeJS自带的fs模块比较基础，把一个目录里的所有文件和子目录都拷贝到另一个目录里需要写不少代码。另外我们也知道，终端下的cp命令比较好用，一条cp -r source/\* target命令就能搞定目录拷贝。那我们首先看看如何使用NodeJS调用终端命令来简化目录拷贝，示例代码如下：

var child\_process = require('child\_process');

var util = require('util');

function copy(source, target, callback) {

child\_process.exec(

util.format('cp -r %s/\* %s', source, target), callback);

}

copy('a', 'b', function (err) {

// ...

});

从以上代码中可以看到，子进程是异步运行的，通过回调函数返回执行结果。

#### *如何获取命令行参数*

在NodeJS中可以通过process.argv获取命令行参数。但是比较意外的是，node执行程序路径和主模块文件路径固定占据了argv[0]和argv[1]两个位置，而第一个命令行参数从argv[2]开始。为了让argv使用起来更加自然，可以按照以下方式处理。

function main(argv) {

// ...

}

main(process.argv.slice(2));