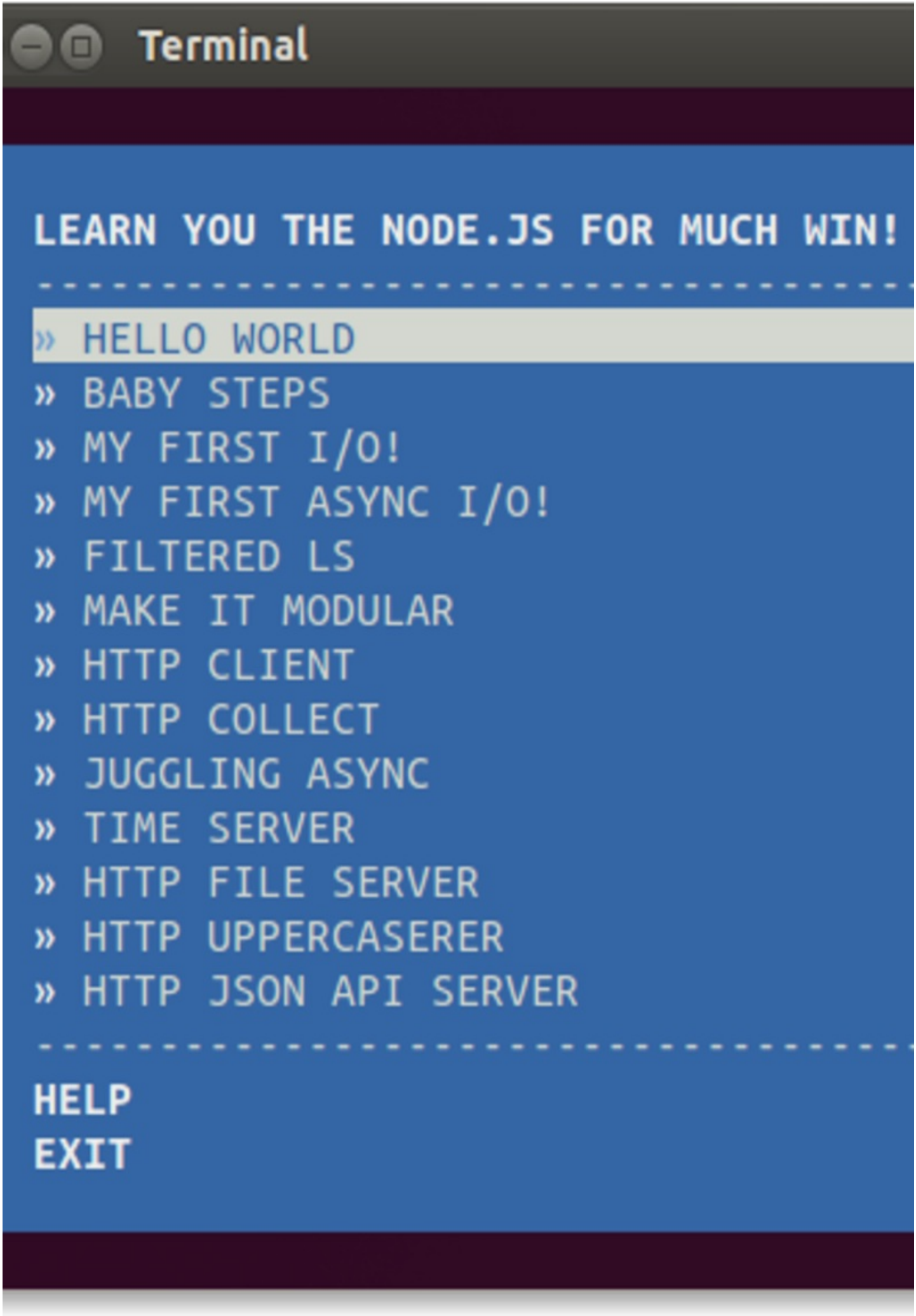
Terminal
LEARN YOU THE NODE.JS FOR MUCH WIN!
» HELLO WORLD
» BABY STEPS
» MY FIRST I/O!
» MY FIRST ASYNC I/O!
» FILTERED LS
» MAKE IT MODULAR
» HTTP CLIENT
» HTTP COLLECT
» JUGGLING ASYNC
» TIME SERVER
» HTTP FILE SERVER
» HTTP UPPERCASERER
» HTTP JSON API SERVER
HELP
EXIT

# （0）起因

> 原文：http://yalishizhude.github.io/2015/07/25/nodejs官方权威教程learnyounode分享/

记得去年听朴灵来长沙介绍node.js和他的《深入浅出node.js》的时候（这本书据说销量不错，但是只能作为进阶，新手不太适合），当时只感觉一头雾水：什么是非阻塞事件驱动？node.js和jquery有什么区别？为什么他不用浏览器就可以执行node.js程序？...

怀着强烈的好奇心瞄了一眼，发现这玩意可不得了，可以用js写服务端，一种语言搞定整个前后端，那是多么牛逼的事情。

于是发现了两个比较好的学习网站：

- nodeschool
  最官方最权威的学习教程都在这里，本文分享的教程都是出于此
- CNode
  国内比较活跃的node.js社区，经常各种招聘以及技术分享

nodeschool上的教程比较多，也不全是关于node.js的，我翻译的13篇教程属于learnyounode系列，偏基础和入门。原教程只有题目，不提供代码。校验通过时会给出参考答案，有些代码是我写的，可能未必最优，但确实能解决所提问题。有兴趣的朋友可以自己安装然后验证代码。

```
npm install learnyounode
```

安装完成后执行 `learnyounode` 就可以查看题目和验证了。

# （1）helloworld

## 要求

编写一个程序，在终端（标准输出 stdout）打印出 “HELLO WORLD”。

## 提示

要编写一个 Node.js 程序，只需要创建一个后缀名为 .js 的文件，然后用 JavaScript去编写即可。完成后，可以通过命令行 node 命令来运行它，例如：

```
$ node program.js
```

和浏览器一样，可以在 Node.js 程序中书写 console：

```
console.log("HELLO WORLD")
```

此时在控制台上看到

```
HELLO WORLD
```

恭喜正式进入node.js世界~

node.js不仅仅为前端工程师转为全栈插上了翅膀，同时它还可以写桌面app，cs程序...总之它很强大，你值得学习~

# （2）命令行参数

## 要求

编写一个简单的程序，使其能接收一个或者多个命令行参数，并且在终端（标准输出 stdout）中打印出这些参数的总和。

## 提示

可以通过 process 这个全局对象来获取命令行中的参数。process 对象 拥有一个名为 argv 的属性，该属性是一个数组，数组中包含了整条命令的所有部分。首先，请先编写一个包含如下带简易代码的程序来熟悉一下：

```
console.log(process.argv)
```

通过执行命令 node program.js 并在后面多加几个参数，来运行我们的程序，比如：

```
$ node program.js 1 2 3
```

这样，就会得到这样一个数组：

```
[ 'node', '/path/to/your/program.js', '1', '2', '3' ]
```

需要考虑的问题是，如何去循环遍历这些代表数字的参数，从而得到他们的总和。process.argv数组的第一个元素永远都会是node，并且第二个参数总是指向程序的路径，所以，应该从第三个元素（index 是 2）开始，依次累加，直到数组末尾。

需要注意的是，所有 process.argv 中的数组的元素都是字符串类型的，因此，需要将它们强制转换成数字。可以通过加上 + 前缀，或者将其传给 Number() 来解决。例如： +process.argv[2] 或者 Number(process.argv[2])。

## 代码

```
var result = 0

for (var i = 2; i < process.argv.length; i++)
    result += Number(process.argv[i])

console.log(result)
```

# （3）同步IO读写

## 要求

编写一个程序，执行一个同步的文件系统操作，读取一个文件，并且在终端（标准输出stdout）打印出这个文件中的内容的行数。类似于执行 `cat file | wc -l` 这个命令。

所要读取的文件的完整路径会在命令行第一个参数提供。

## 提示

要执行一个对文件系统的操作，将会用到 fs 这个 Node 核心模块。要加载这类核心模块或者其他的”全局”模块，可以用下面的方式引入：

```
var fs = require('fs')
```

这样就可以通过 fs 这个变量来访问整个 fs 模块了。

在 fs 中，所有的同步（或者阻塞）的操作文件系统的方法名都会以 ‘Sync’ 结尾。要读取一个文件，将需要使用 `fs.readFileSync('/path/to/file')` 方法。这个方法会返回一个包含文件完整内容的 Buffer 对象。

(fs模块API文档)[https://nodejs.org/api/fs.html]

Buffer 对象是 Node 用来高效处理数据的方式，无论该数据是 ascii 还是二进制文件，或者其他的格式。Buffer 可以很容易地通过调用 `toString()` 方法转换为字符串。如：

```
var s tr = buf.toString()。
```

(Buffer模块API文档)[https://nodejs.org/api/buffer.html]

简单地计算行数方法，可以使用.split() 分割成子字符串数组，’ \n’ 可以作为分隔符。

## 代码

```
var fs = require('fs');
var res = fs.readFileSync(process.argv[2], 'utf-8');
console.log(res.split('\n').length-1);
```

# （4）异步IO读写

## 要求

编写一个程序，执行一个异步的对文件系统的操作：读取一个文件，并且在终端（标准输出stdout）打印出这个文件中的内容的行数。类似于执行 cat file | wc -l 这个命令。 所要读取的文件的完整路径会在命令行第一个参数提供。

## 提示

解决这个问题将需要用到Node.js最鲜明的风格的方式：异步。

‘’ ’ fs.readFile()’ ’ ’ 方法可以满足这个需求，这需要从传入的回调函数中去收集数据（这些数据会作为第二参数传递给回调函数），而不是使用方法的返回值。

记住，Node.js 回调函数都有像如下所示的特征：

```
function callback (err, data) { /* ... */ }
```

可以通过检查第一个参数的真假值来判断是否有错误发生。如果没有错误发生，第二个参数将获取到一个Buffer对象。和 readFileSync() 一样，可以传入 ‘utf8 ‘作为它的第二个参数，然后把回调函数作为第三个参数，这样，得到的将会是一个字符而不是 Buffer。

fs模块API文档

## 代码

```
var fs = require('fs');
fs.readFile(process.argv[2], function(err,data){
    if(err) throw err;
    console.log(data.toString().split('\n').length-1);
});
```

# （5）过滤器

## 要求

编写一个程序来打印出指定目录下的文件列表，并且以特定的文件名扩展名来过滤这个列表。命令行提供两个参数提，第一个是所给的文件目录路径（如：path/to/dir），第二个参数则是需要过滤出来的文件的扩展名。

举个例子：如果第二个参数是 txt，那么需要过滤出那些扩展名为 .txt的文件。

注意，第二个参数将不会带有开头的”.”。

需要在终端中打印出这个被过滤出来的列表，每一行一个文件。另外，必须使用异步的I/O 操作。

## 提示

fs.readdir() 方法接收两个参数：第一个是一个路径，第二个则是回调函数，这个回调函数会有如下特征：

```
function callback (err, list) { /* ... */ }
```

这里的 list 是一个数组，它所包含的元素是每个文件的文件名（字符串形式）。

fs模块API文档

node 自带的 path 模块也很有用，特别是它那个 extname 方法。

path模块API文档

## 代码

```
var fs = require('fs')
var path = require('path')

fs.readdir(process.argv[2], function (err, list) {
  list.forEach(function (file) {
    if (path.extname(file) === '.' + process.argv[3])
      console.log(file)
  })
})
```

# （6）自定义模块

## 要求

这个问题和前面一个一样，但是这次需要使用模块。将需要创建两个文件来解决这个问题。

编写一个程序来打印出所给文件目录的所含文件的列表，并且以特定的文件名后缀来过滤这个列表。这次将会提供两个参数，第一个参数是要列举的目录，第二个参数是要过滤的文件扩展名。在终端中打印出过滤出来的文件列表（一个文件一行）。此外，必须使用异步 I/O。

需要编写一个模块文件去做大部分的事情。这个模块必须导出（export）一个函数，这个函数将接收三个参数：目录名、文件扩展名、回调函数，并按此顺序传递。文件扩展名必须和传递给程序的扩展名字符串一模一样。也就是说，请不要把它转成正则表达式或者加上”.”前缀或者做其他的处理，而是直接传到模块中去，在模块中，可以做一些处理来使过滤器能正常工作。

这个回调函数必须以 Node 编程中惯用的约定形式（err,data）去调用。这个约定指明了，除非发生了错误，否则所传进去给回调函数的第一个参数将会是 null，第二个参数才会是数据。在本题中，这个数据将会是过滤出来的文件列表，并且是以数组的形式。如果接收到了一个错误，如：来自 `fs.readdir()` 的错误，则必须将这个错误作为第一个，也是唯一的参数传递给回调函数，并执行回调函数。

绝对不能直接在模块文件中把结果打印到终端中，只能在原始程序文件中编写打印结果的代码。

当程序接收到一些错误的时候，请简单的捕获它们，并且在终端中打印出相关的信息

这里有四则规定，模块必须遵守：

- 导出一个函数，这个函数能准确接收上述的参数。
- 当有错误发生，或者有数据的时候，准确调用回调函数。
- 不要改变其他的任何东西，比如全局变量或者 stdout。
- 处理所有可能发生的错误，并把它们传递给回调函数。

遵循一些约定的好处是，模块可以被任何其他也遵守这些约定的人所使用。

## 提示

通过创建一个仅包含目录读取和文件过滤相关的函数的文件来建立一个新的模块。要使模块导出（export）单一函数（single function），可以将函数赋值给 `module.exports` 对象：

```
module.exports = function (args) { /* ... */ }
```

或者也可以使用命名函数，然后把函数名赋值给 `module.exports` 。

要在原来的程序中使用新的模块，请用 require() 载入模块，这和载入 fs 模块时候用 `require('fs')` 一样，

唯一的区别在于本地模块需要加上 ‘./ ‘ 这个相对路径前缀。所以，如果模块文件名字是 mymodule.js，那么需要像这样写：

```
var mymodule = require('./mymodule.js')
```

‘.js’ 这个文件扩展名通常是可以省略的。

现在，mymodule 这个变量就指向了模块中 module.exports了，因为刚导出了一个单一的函数，所以现在所声明的变量 mymodule 就是那个模块所导出的函数了，就可以直接调用它了！

同样，请记住，尽早捕获错误，并且在回调中返回：

```
function bar (callback) {
  foo(function (err, data) {
    if (err)
      return callback(err) // 尽早返回错误

    // ... 没有错误，处理 `data`

    // 一切顺利，传递 null 作为 callback 的第一个参数

    callback(null, data)
  })
}
```

# 代码

## solution.js:

```
var filterFn = require('./solution_filter.js')
var dir = process.argv[2]
var filterStr = process.argv[3]

filterFn(dir, filterStr, function (err, list) {
  if (err)
    return console.error('There was an error:', err)

  list.forEach(function (file) {
    console.log(file)
  })
})
```

## solution_filter.js:

```
var fs = require('fs')
var path = require('path')

module.exports = function (dir, filterStr, callback) {

  fs.readdir(dir, function (err, list) {
    if (err)
      return callback(err)

    list = list.filter(function (file) {
      return path.extname(file) === '.' + filterStr
    })

    callback(null, list)
  })
}
```

# （7）http客户端

## 要求

编写一个程序来发起一个 HTTP GET 请求，所请求的 URL 为命令行参数的第一个。然后将每一个“data” 事件所得的数据，以字符串形式在终端（标准输出 stdout）的新的一行打印出来。

## 提示

完成这个练习，需要使用 Node.js 核心模块之一：http。

http模块API文档

`http.get()` 方法是用来发起简单的 GET 请求的快捷方式，使用这个方法可以一定程度简化程序。`http.get()` 的第一个参数是GET 的URL，第二个参数则是回调函数。

与其他的回调函数不同，这个回调函数有如下这些特征：

```
function callback (response) { /* ... */ }
```

response 对象是一个 Node 的 Stream 类型的对象，可以将 Node Stream当做一个会触发一些事件的对象，其中我们通常所需要关心的事件有三个： “data” ，” error” 以及 “en” 。可以像这样来监听一个事件：

```
response.on("data", function (data) { /* ... */ })
```

‘data’ 事件会在每个数据块到达并已经可以对其进行一些处理的时候被触发。数据块的大小将取决于数据源。

从 `http.get()` 所获得的 response 对象/Stream 还有一个 setEncoding() 的方法。如果调用这个方法，并为其指定参数为 utf8，那么 data 事件中会传递字符串，而不是标准的 Node Buffer 对象，这样，也不用再手动将 Buffer 对象转换成字符串了。

## 代码

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

http.get(process.argv[2], function (response) {
  response.setEncoding('utf8');
  response.on('data', console.log);
  response.on('error', console.error);
});
```

# （8）http收集器

## 要求

编写一个程序，发起一个 HTTP GET 请求，请求的 URL 为所提供的命令行参数的第一个。收集所有服务器所返回的数据（不仅仅包括 “data” 事件）然后在终端（标准输出 std out）用两行打印出来。

所打印的内容，第一行应该是一个整数，用来表示收到的字符串内容长度，第二行则是服务器返回的完整的字符串结果。

## 提示

有两种实现方法：

1) 可以把所有 “data” 事件所得的结果收集起来，暂存并追加在一起，而不是在收到后立刻打印出来。通过监听 “end” 事件，可以确定 stream 是否完成传输，如果传输结束了，就可以将收集到的结果打印出来了。

2) 使用一个第三方模块，来简化从 stream 中收集数据的繁琐步骤。这里有两个不同的模块都提供了一些有用的 API 来解决这个问题（似乎还有好多另外的模块可以选哦！）：bl (Buffer list) 或者 concat-stream，来选一个吧！

bl模块API文档
concat-stream模块API文档

要安装一个 Node 模块，需用到 Node 的包管理工具 npm，输入：

```
$ npm install bl
```

这样，相应的模块的最新版本便会被下载到当前目录下一个名为 node_modules 的子目录中 。任何在这个子目录中的模块都可以简单地使用 require 语法来将模块载入到程序中 ，并且不需要加 ./ 这样的路径前缀，如下所示：

```
var bl = require('bl')
```

可以把一个 stream pipe 到 bl 或 concat-stream 中去，它们会收集数据。一旦 stream 传输结束，一个回调函数会被执行，并且，这个回调函数会带上所收集的数据：

```
response.pipe(bl(function (err, data) { /* ... */ }))
// 或
response.pipe(concatStream(function (data) { /* ... */ }))
```

要注意的是可能需要使用 data.toString() 来把 Buffer 转换为字符串。

# 代码

- 方法一

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

http.get(process.argv[2], function(res){
var result = '';
res.setEncoding('utf8');
res.on('data', function(data){
result += data;
});
res.on('end', function(data){
console.log(result.length);
console.log(result);
});
});
```

- 方法二

```
var http = require('http')
var bl = require('bl')

http.get(process.argv[2], function (response) {
 response.pipe(bl(function (err, data) {
 if (err)
 return console.error(err)
 data = data.toString()
 console.log(data.length)
 console.log(data)
 }))
})
```

# （9）玩转异步

## 要求

这次的问题和之前的问题（HTTP 收集器）很像，也是需要使用到 http.get() 方法。然而，这一次，将有三个 URL 作为前三个命令行参数提供。

需要收集每一个 URL 所返回的完整内容，然后将它们在终端（标准输出stdout）打印出来。这次不需要打印出这些内容的长度，仅仅是内容本身即可（字符串形式）；每个 URL对应的内容为一行。重点是必须按照这些 URL 在参数列表中的顺序将相应的内容排列打印出来才算完成。

## 提示

不要期待这三台服务器能好好的一起玩耍！他们可能不会把完整的响应的结果按照希望的顺序返回，所以不能天真地只是在收到响应后直接打印出来，因为这样做的话，他们的顺序可能会乱掉。

需要去跟踪到底有多少 URL 完整地返回了他们的内容，然后用一个队列存储起来。一旦拥有了所有的结果，才可以把它们打印到终端。

对回调进行计数是处理 Node 中的异步的基础。比起自己去做，去依赖一个第三方的模块或者库会更方便，比如 async 或者 after。不过，在本次练习中，应该首先尝试自己去解决，而不是依赖外部的模块。

## 代码

- 方法一

```
var http = require('http');
var result = ['','',''];
var isEnd = [false, false, false];

  http.get(process.argv[2], function(res){
    res.setEncoding('utf8');
    res.on('data', function(data){
      result[0] += data;
    });
    res.on('end', function(data){
      isEnd[0] = true;
      if(isEnd[0]&&isEnd[1]&&isEnd[2]){
        console.log(result[0]);
        console.log(result[1]);
        console.log(result[2]);
      }
    });
  });
  http.get(process.argv[3], function(res){
    res.setEncoding('utf8');
    res.on('data', function(data){
      result[1] += data;
    });
    res.on('end', function(data){
      isEnd[1] = true;
      if(isEnd[0]&&isEnd[1]&&isEnd[2]){
        console.log(result[0]);
        console.log(result[1]);
        console.log(result[2]);
      }
    });
  });
  http.get(process.argv[4], function(res){
    res.setEncoding('utf8');
    res.on('data', function(data){
      result[2] += data;
    });
    res.on('end', function(data){
      isEnd[2] = true;
      if(isEnd[0]&&isEnd[1]&&isEnd[2]){
        console.log(result[0]);
        console.log(result[1]);
        console.log(result[2]);
      }
    });
  });
```

- 方法二

```
var http = require('http')
  var bl = require('bl')
  var results = []
  var count = 0

  function printResults () {
    for (var i = 0; i < 3; i++)
      console.log(results[i])
  }

  function httpGet (index) {
    http.get(process.argv[2 + index], function (response) {
      response.pipe(bl(function (err, data) {
        if (err)
    return console.error(err)

        results[index] = data.toString()
        count++

        if (count == 3)
    printResults()
      }))
    })
  }

  for (var i = 0; i < 3; i++)
httpGet(i)
```

# （10）授时服务器

## 要求

编写一个 TCP 时间服务器

服务器监听一个端口，以获取一些TCP连接，这个端口会经由第一个命令行参数传递给程序。针对每一个 TCP 连接，都必须写入当前的日期和24小时制的时间，如下格式：

```
"YYYY-MM-DD hh:mm"
```

然后紧接着是一个换行符。

月份、日、小时和分钟必须用零填充成为固定的两位数：

```
"2013-07-06 17:42"
```

## 提示

这次练习中，将会创建一个 TCP 服务器。这里将不会涉及到任何 HTTP 的事情，因此只需使用 net 这个 Node 核心模块就可以了。它包含了所有的基础网络功能。
net 模块拥有一个名叫 net.createServer() 的方法，它会接收一个回调函数。和 Node 中其他的回调函数不同，createServer() 所用的回调函数将会被调用多次。服务器每收到一个 TCP 连接，都会调用一次这个回调函数。这个回调函数有如下特征：

```
function callback (socket) { /* ... */ }
```

net.createServer() 也会返回一个 TCP 服务器的实例，必须调用 server.listen(portNumber) 来让服务器开始监听一个特定的端口。

一个典型的 Node TCP 服务器将会如下所示：

```
var net = require('net')
var server = net.createServer(function (socket) {
  // socket 处理逻辑
})
server.listen(8000)
```

net模块API文档

记住，请一定监听由第一个命令行参数指定的端口。

socket 对象包含了很多关于各个连接的信息（meta-data），但是它也同时是一个 Node 双工流（duplex Stream），所以，它即可以读，也可以写。对这个练习来说，只需要对socket 写数据和关闭它就可以了。

使用 socket.write(data) 可以写数据到 socket 中，用 socket.end() 可以关闭一个 socket。另外， .end() 方法也可以接收一个数据对象作为参数，因此，可简单地使用 socket.end(data) 来完成写数据和关闭两个操作。

# 代码

方法一：

```
var net = require('net');

net.createServer(function(socket){
 var date= new Date();
 socket.end(date.toLocaleDateString());
}).listen(process.argv[2]);
```

方法二：

```
var net = require('net')

function zeroFill(i) {
 return (i 10 ? '0' : '') + i
}

function now () {
 var d = new Date()
 return d.getFullYear() + '-'
 + zeroFill(d.getMonth() + 1) + '-'
 + zeroFill(d.getDate()) + ' '
 + zeroFill(d.getHours()) + ':'
 + zeroFill(d.getMinutes())
}

var server = net.createServer(function (socket) {
 socket.end(now() + '\n')
})

server.listen(Number(process.argv[2]))
```

# （11）文件服务器

## 要求

编写一个 HTTP 文件 服务器，它用于将每次所请求的文件返回给客户端。

服务器需要监听所提供的第一个命令行参数所制定的端口。

同时，第二个会提供给程序的参数则是所需要响应的文本文件的位置。在这一题中必须使用 `fs.createReadStream()` 方法以 stream 的形式作出请求相应。

## 提示

由于我们需要创建的是一个 HTTP 服务而不是普通的 TCP 服务，因此，应该使用 http 这个 Node 核心模块。它和 net 模块类似，http 模块拥有一个叫做 `http.createServer()` 的方法，所不同的是它所创建的服务器是用 HTTP 协议进行通信的。

`http.createServer()` 接收一个回调函数作为参数，回调函数会在你的服务器每一次进行连接的时候执行，这个回调函数有以下的特征：

```
function callback (request, response) { /* ... */ }
```

在这里，这两个参数是代表一个 HTTP 请求以及相应的响应的两个对象。request 用来从请求中获取一些的属性，例如请求头和查询字符（query-string），而 response 会发送数据给客户端，包括响应头部和响应主体。

request 和 response 也都是 Node stream！这意味着，如果需要的话，可以使用流式处理（streaming）所抽象的那些方法来实现发送和接收数据。

`http.createServer()` 会返回一个 HTTP 服务器的实例。这里需要调用 server.listen(portNumber) 方法去监听一个特定的端口。

一个典型的 Node HTTP 服务器将会是这个样子：

```
var http = require('http')
var server = http.createServer(function (req, res) {
  // 处理请求的逻辑...
})
server.listen(8000)
```

http模块API文档

fs模块API文档

fs 这个核心模块也含有一些用来处理文件的流式（ stream ） API。可以使用 fs.createReadStream() 方法来为命令行参数指定的文件创建一个 stream。这个方法会返回一个 stream 对象，该对象可以使用类似 src.pipe(dst) 的语法把数据从 src流传输(pipe) 到 dst 流中。通过这种形式，可以轻松地把一个文件系统的 stream 和一个 HTTP 响应的 stream 连接起来。

# 代码

```
var http = require('http')
var fs = require('fs')

var server = http.createServer(function (req, res) {
 res.writeHead(200, { 'content-type': 'text/plain' })

 fs.createReadStream(process.argv[3]).pipe(res)
})

server.listen(Number(process.argv[2]))
```

# （12）大写转换器

## 要求

编写一个 HTTP 服务器，它只接受 POST 形式的请求，并且将 POST 请求主体（body）所带的字符转换成大写形式，然后返回给客户端。

服务器需要监听由第一个命令行参数所指定的端口。

## 提示

这里将不限制你使用 stream 处理 request 和 response 对象，并且这将更为简单。

在 npm 中，有很多不同的模块可以用来在 stream 传输过程中 “转换” stream 中的数据。对于本次练习来说，through2-map 这个模块有一个比较简单的 API 可以使用。

through2-map 允许你创建一个 transform stream，它仅需要一个函数就能完成「接收一个数据块，处理完后返回这个数据块」的功能 ，它的工作模式类似于 Array#map()，但是是针对 stream 的：

```
var map = require('through2-map')
inStream.pipe(map(function (chunk) {
  return chunk.toString().split('').reverse().join('')
})).pipe(outStream)
```

在上面的例子中，从 inStream 传进来的数据会被转换成字符串（如果它不是字符串的话），并且字符会反转处理，然后传入 outStream。所以，我们这里是做了一个字符串反转器！记住！尽管，数据块（chunk）的大小是由上游（up-stream）所决定的，但是还是可以在这之上对传进来的数据做一点小小的处理的。

要安装 through2-map，输入:

```
$ npm install through2-map
```

through2-map模块API文档

## 代码

方法一：

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

http.createServer(function(req,res){
var postData = '';
req.addListener( “data” , function (postDataChunk) {
if(req.method==='POST'){
postData += postDataChunk;
}
});
req.addListener( “end” , function(){
if(req.method==='POST'){
res.end(postData.toUpperCase(),'utf8');
}
});
}).listen(process.argv[2]);
```

方法二：

```
var http = require('http')
var map = require('through2-map')

var server = http.createServer(function (req, res) {
if (req.method != 'POST')
return res.end('send me a POST\n')

req.pipe(map(function (chunk) {
return chunk.toString().toUpperCase()
})).pipe(res)
})

server.listen(Number(process.argv[2]))
```

# （13）JSON API服务器

## 要求

编写一个 HTTP 服务器，每当接收到一个路径为 '/api/parsetime' 的 GET 请求的时候，响应一些 JSON 数据。我们期望请求会包含一个查询参数（query string），key 是 “iso ，值是 ISO 格式的时间。

如:

```
/api/parsetime?iso=2013-08-10T12:10:15.474Z
```

所响应的 JSON 应该只包含三个属性：'hour'，'minute' 和 'second'。例如：

```
{
  "hour": 14,
  "minute": 23,
  "second": 15
}
```

然后增再加一个接口，路径为 '/api/unixtime'，它可以接收相同的查询参数（query strng），但是它的返回会包含一个属性：'unixtime'，相应值是一个 UNIX 时间戳。例如:

```
{ "unixtime": 1376136615474 }
```

服务器需要监听第一个命令行参数所指定的端口。

## 提示

HTTP 服务器的 request 对象含有一个 url 属性，你可以通过它来决定具体需要走哪一条”路由”。

可以使用 Node 的核心模块 'url' 来处理 URL 和 查询参数（query string）。
`url.parse(request.url, true)` 方法会处理 request.url，它返回的对象中包含了一些很有帮助的属性，方便方便你处理 querystring。

举个例子，可以在命令行窗口输入以下命令试试：

```
$ node -pe "require('url').parse('/test?q=1', true)"
```

服务器的响应应该是一个 JSON 字符串的形式。请查看 JSON.stringify() 来获取更多信息。

为了争做 Web 世界的好公民，正确地为响应设置 Content-Type 属性：

```
res.writeHead(200, { 'Content-Type': 'application/json' })
```

JavaScript 的 Date 可以将日期以 ISO 的格式展现出来，如： `new Date().toISOString()` 。并且，如果把一个字符串传给 Date的构造函数，它也可以帮你将字符串处理成日期类型。另外，Date#getTime() 放个应该也会很有用。

url模块API文档

# 代码

```
var http = require('http');
var url = require('url');
var querystring = require('querystring');

http.createServer(function(req,res){
var obj = url.parse(req.url);
var param = querystring.parse(obj.query);
res.writeHead(200, { “Content-Type”: “application/json” });
if( “/api/parsetime” ===obj.pathname){
var date = new Date(param.iso);
var retObj = {
hour: date.getHours(),
minute: date.getMinutes(),
second: date.getSeconds()
};
res.end(JSON.stringify(retObj));
}
if('/api/unixtime'===obj.pathname){
var date = new Date(param.iso);
var retObj = {unixtime: date.getTime()};
res.end(JSON.stringify(retObj));
}
}).listen(process.argv[2]);
```