Node.js入门和企业级应用开发实战

第1天课堂笔记（本课程共8天）

# 目录

[目录 2](#_Toc2200)

[一、Node.js简介 3](#_Toc854)

[1.1 简介 3](#_Toc26361)

[1.2 特点 3](#_Toc17771)

[1.3 适合开发什么？ 5](#_Toc16180)

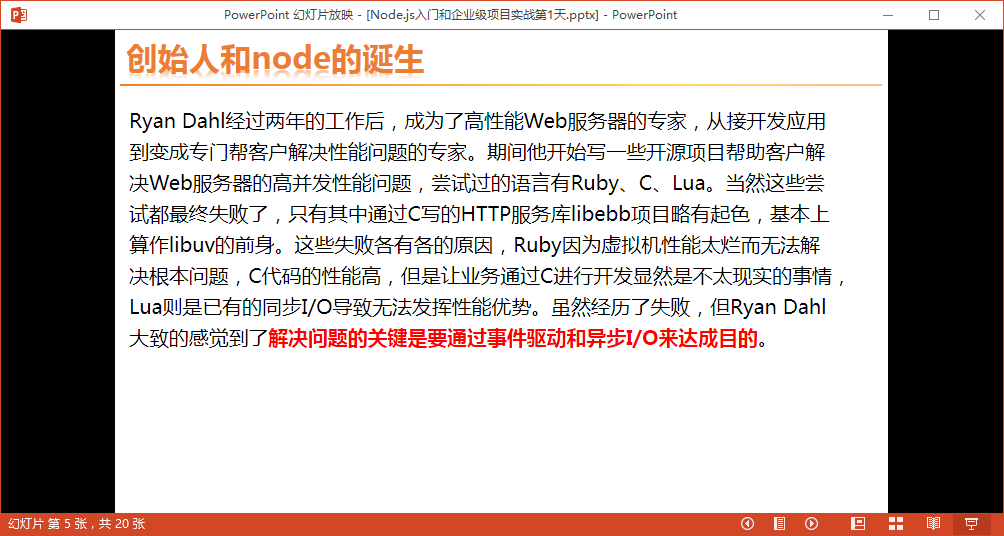
[1.4 Node.js无法挑战老牌3P 6](#_Toc16106)

[二、Node.js安装 7](#_Toc13131)

[三、HTTP模块 9](#_Toc2959)

# Node.js简介

[先了解一下什么是进程和线程](http://bbs.duchengjiu.top/read.php?tid=101&fid=7)



我们的node.js创始人，node的发明就是他在解决性能问题上的产物。此时Ryan Dahl发现解决性能问题是要靠**事件驱动、异步I/O**来达成目的。

我们现在用生活中的比喻，来说说什么是事件驱动，什么是异步I/O。



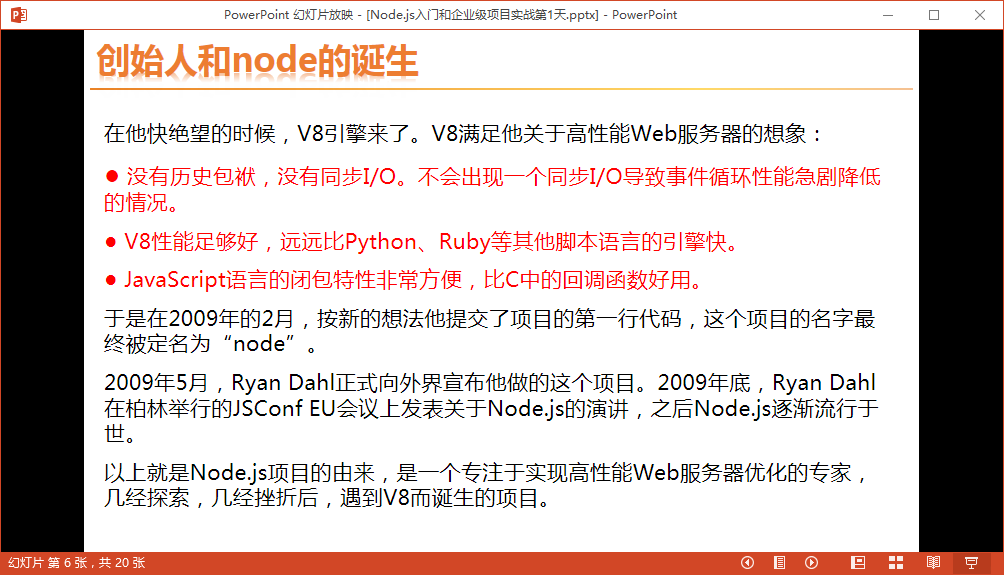
比如银行有6个窗口，ok，此时能并行6个客户，6个客户能够同时进行业务。其实，转念一想，完全可以并行10个人！20个人！30个人！甚至无数人！怎么弄呢？比如A客户在1号窗口要存钱，钱存进去电脑要写磁盘，滋啦滋啦的，挺耗费时间的，我们叫做磁盘I/O。i就是input写，o就是output读。此时如果干等着，我们称作这个1号服务窗口“被阻塞了”。

此时我们可以这样，当A用户正在磁盘I/O的时候，开始服务B客服！问问B客户您干吗啊！B客户说，我要来转账！此时赶紧让B客户填单子，省的一会因为B客户填单子，又一次阻塞了1号服务窗口。 此时我们又空出时间来服务C客户了，C客户说需要取钱，取2000个亿，银行没有这么多钱，需要去总行调取，此时“金库I/O阻塞”（实际在模拟数据库I/O阻塞）。此时又空出时间可以服务D客户了……问了D客户你要来干嘛，D客户还没有张嘴，我们的计算机给A客户存钱滋啦滋啦结束了，叮了一声（事件，回调函数），我们说D客户您稍等，我们给A存完钱，A客户满意的走了。此时恰好B客户也填好单子了，也叮了一声，我们服务B客户，当B客户转账计算机开始滋啦滋啦，此时服务D客户，D客户说我要来取钱，D客户开始填单子。此时B客户滋啦滋啦完毕，B也满意的走了。D填完单子了，钱也取好了，服务完毕了，D满意的走了。不一会儿，总行把2000个亿总来了，服务完毕C客服。

**微观上，A、B、C、D不是同时的，但是宏观上，是并行的！**

银行有6个窗口，就是6个进程，但是6个进程不是只能同时服务6个程序。更夸张的，如果只有1个进程，刚才的例子，就能证明也能服务号4个用户。

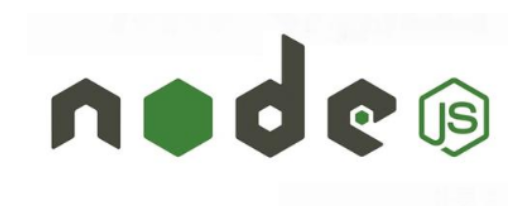
**你们肯定见过有些餐馆只有1个服务员。这1个服务员如果妥善安排好被阻塞的时间，此时效率甚至堪比10多个服务员！什么叫被阻塞的时间呢？比如一个处女座，纠结不知道点什么吃，您理他干嘛呀！先去服务别人啊！！他想好吃啥了，会喊你。此时宏观上你在同时服务很多客人。**



node表示节点，就是随便发明的名词，听着好听而已。node.js使用的是Chrome的JS解析引擎V8。就是把用于JS在前台渲染的引擎，移植到了服务器上，也就是说，语言是老语言就是JavaScript，引擎是浏览器的引擎，只不过就是把语句的执行位置，从浏览器挪到了服务器上。

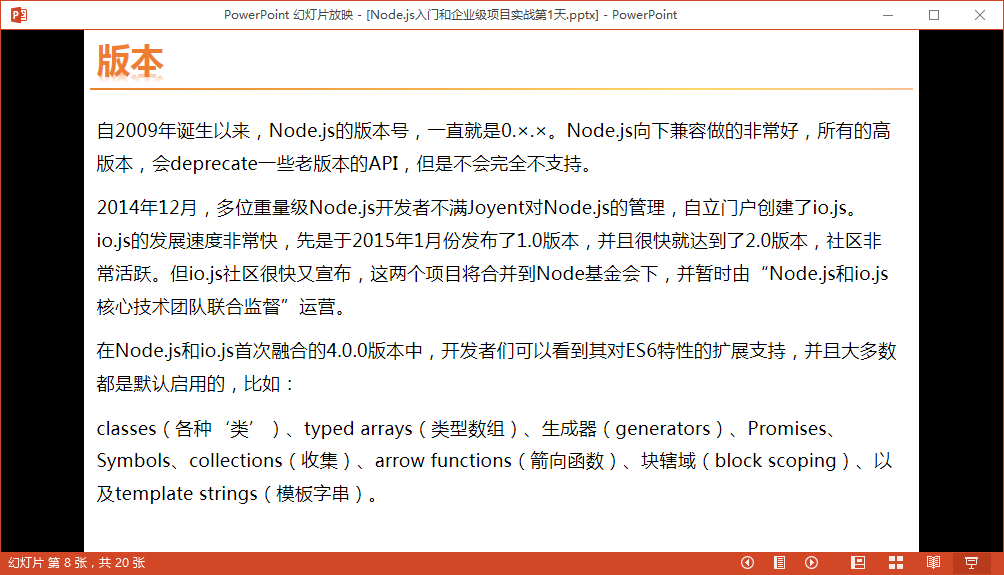


node.js的logo：



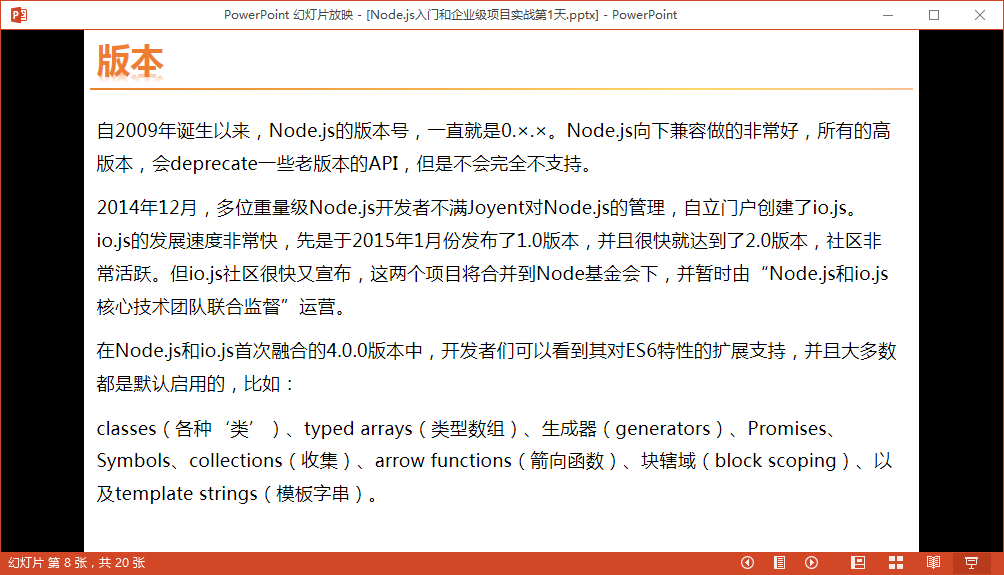
node.js的官网：https://nodejs.org/en/

开门注：Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.



我们学习Node.js 6.8.1。

Node.js和io.js的分分合合：



我们的理论知识要通过实践来学习，node.js的三大特点：

1. 单线程 每个进程可有多个线程
2. 非阻塞I/O
3. 事件驱动

我们要用一上午的时间，来仔仔细细体会这三个特点。

## 1.1 简介

V8引擎本身就是用于Chrome浏览器的JS解释部分，但是Ryan Dahl这哥们，鬼才般的，把这个V8搬到了服务器上，用于做服务器的软件。

Node.js是一个专注于实现高性能Web服务器优化的专家，几经探索，几经挫折后，遇到V8而诞生的项目。

Node.js是一个让JavaScript运行在服务器端的开发平台，它让JavaScript的触角伸到了服务器端，可以与PHP、JSP、Python、Ruby平起平坐。

但Node似乎有点不同：

● **Node.js不是一种独立的语言**，与PHP、JSP、Python、Perl、Ruby的“既是语言，也是平台”不同，Node.js的**使用JavaScript进行编程**，运行在JavaScript引擎上（V8）。

● 与PHP、JSP等相比（PHP、JSP、.net都需要运行在服务器程序上，Apache、Naginx、Tomcat、IIS。

），**Node.js跳过了Apache、Naginx、IIS等HTTP服务器，它自己不用建设在任何服务器软件之上**。Node.js的许多设计理念与经典架构（LAMP = Linux + Apache + MySQL + PHP）有着很大的不同，可以提供强大的伸缩能力。一会儿我们就将看到，Node.js没有web容器。

Node.js自身哲学，是花最小的硬件成本，追求更高的并发，更高的处理性能。

官网：<https://nodejs.org/en/>

特点：Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

## 1.2 特点

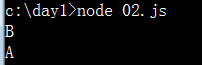
所谓的特点，就是Node.js是如何解决服务器高性能瓶颈问题的。

## 异步特性

Node.js中几乎所有的命令都是异步的！也就是说，Node.js中基本上不会遇见阻塞这个事情。

什么是异步：就是当面对一个需要耗时间的事情的时候，有另种态度：① 死等，死等这个耗时间的事情完成，同步，synchronouse。 ② 先做别的事情，当耗时的事情做完的时候，执行它的回调函数，异步asynchronouse。

|  |
| --- |
| //引用内置http模块，require表示引用  var http = require("http");  //创建一个服务器  var server = http.createServer(function(req,res){  res.setHeader("Content-type","text/html;charset=utf-8");  res.end("好高兴啊，我买了一个note" + (3.5 \* 2) + "，爆炸了！哈哈哈哈");  console.log("A");  });  console.log("B");  //监听3000端口号，和一个ip  server.listen(3000,"127.0.0.1"); |



http.createServer()这个函数是异步函数，所以程序不会死等“创建服务器”完毕，而是会先执行别的语句。

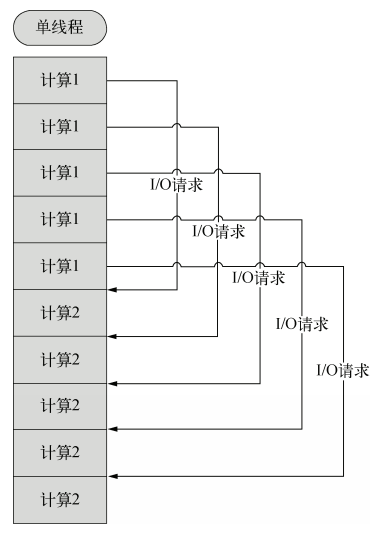
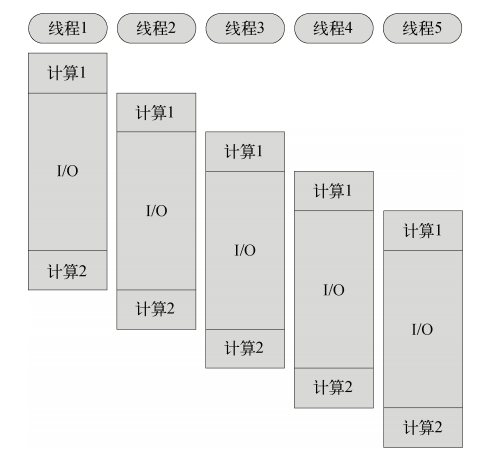
单线程

在Java、PHP或者.net等服务器端语言中，会为每一个客户端连接创建一个新的线程。而每个线程需要耗费大约2MB内存。也就是说，理论上，一个8GB内存的服务器可以同时连接的最大用户数为4000个左右。要让Web应用程序支持更多的用户，就需要增加服务器的数量，而Web应用程序的硬件成本当然就上升了。

Node.js不为每个客户连接创建一个新的线程，而仅仅使用一个线程。当有用户连接了，就触发一个内部事件，通过非阻塞I/O、事件驱动机制，让Node.js程序宏观上也是并行的。使用Node.js，一个8GB内存的服务器，可以同时处理超过4万用户的连接。

另外，带线程的带来的好处，还有操作系统完全不再有线程创建、销毁的时间开销。

坏处，就是一个用户造成了线程的崩溃，整个服务都崩溃了，其他人也崩溃了。



多线程、单线程的一个对比。

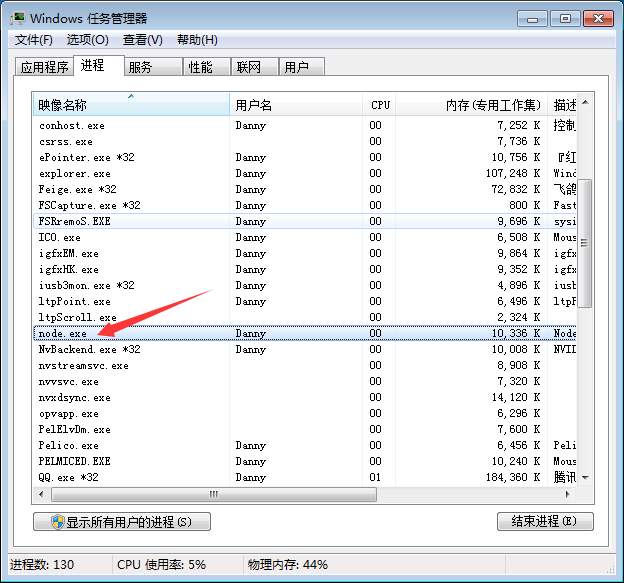
也就是说，单线程也能造成宏观上的“并发”。

我们写一段小程序，在createServer()外面创建了一个变量，此时我们发现这个变量是所有用户共享的。

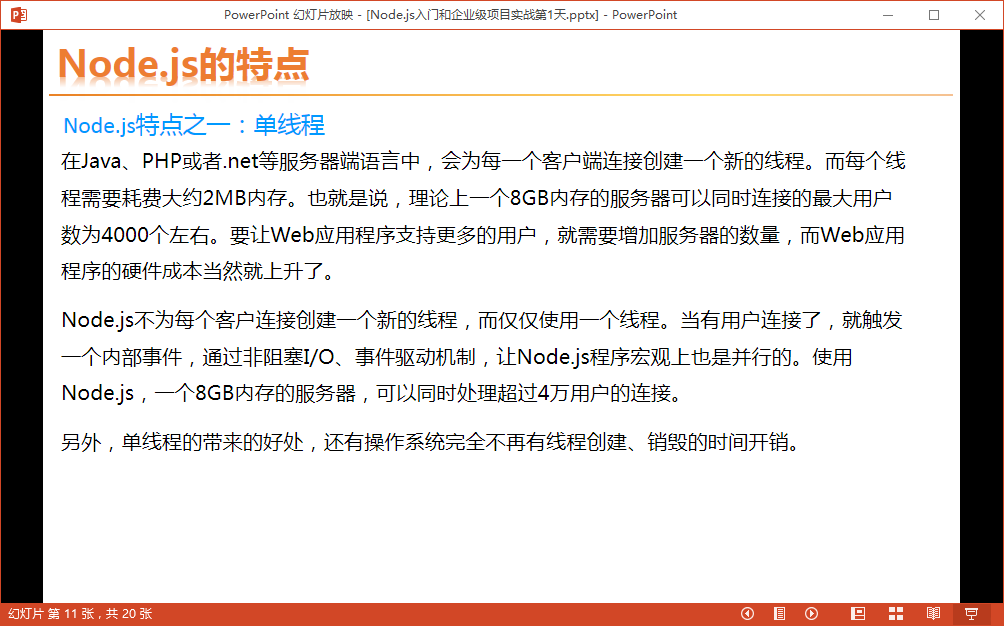
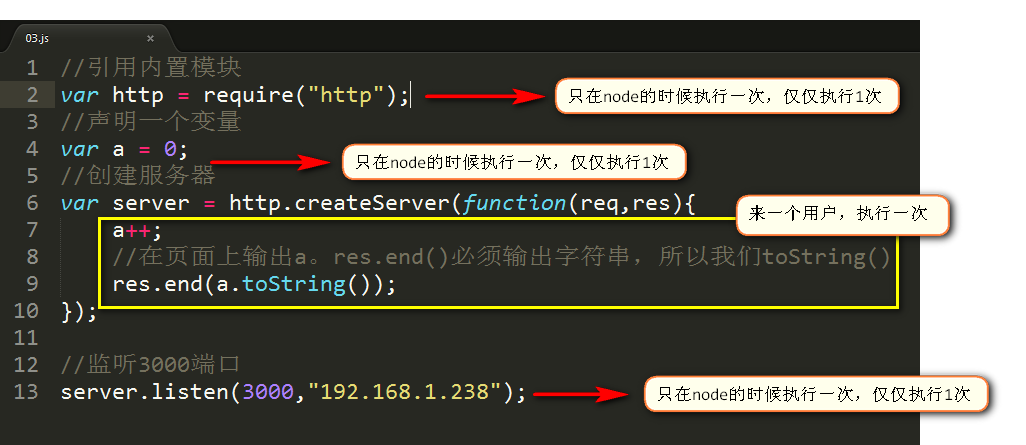
|  |
| --- |
| //引用内置模块  var http = require("http");  //声明一个变量  var a = 0;  //创建服务器  var server = http.createServer(function(req,res){  a++;  //在页面上输出a。res.end()必须输出字符串，所以我们toString()  res.end(a.toString());  });  //监听3000端口  server.listen(3000,"192.168.1.238"); |

这是因为node.js只在服务器上创建一个进程，node.js是单进程的。不管来多少用户，node.js永远只用这一个进程。

当node.exe跑起来之后，系统仅仅只有一个进程，不会随着用户的增多而增多。



我们要知道哪些语句是来一个用户执行一次，哪些语句是仅执行1次的：



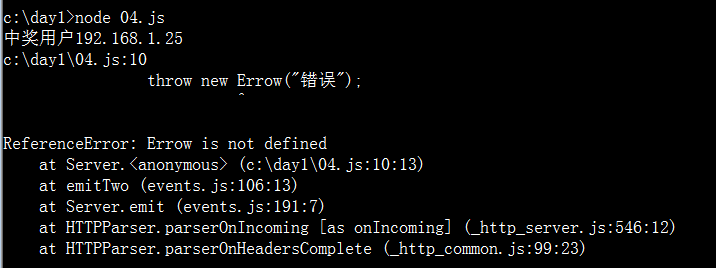
Node.js就相当于只有一个服务员的餐厅，利用自己的非阻塞I/O（异步特性）宏观上可以同时服务多个用户。

单线程的好处：减少内存的开销，也没有了进程创建、销毁时间开销。

单线程的坏处：一个用户的程序出了问题，可能这个问题来自用户，比如他输入了数字除以0，此时一个老鼠屎坏了一锅汤。

我们做了一个实验，随机生成9999以内的数字，如果是6666就抛出错误，一个同学刷出了6666全班马上个都不能访问服务器了。

|  |
| --- |
| //引用内置模块  var http = require("http");  //创建服务器  var server = http.createServer(function(req,res){  // 随机生成一个数字  var num = parseInt(Math.random() \* 9999);  // 如果是6666，那么抛出错误  if(num == 6666){  console.log("中奖用户" + req.connection.remoteAddress)  throw new Errow("错误");  }  //输出数字  res.end(num.toString() + "【ip】" + req.connection.remoteAddress);  });  //监听3000端口  server.listen(3000,"192.168.1.238"); |



非阻塞I/O non-blocking I/O

例如，当在访问数据库取得数据的时候，需要一段时间。在传统的单线程处理机制中，在执行了访问数据库代码之后，整个线程都将暂停下来，等待数据库返回结果，才能执行后面的代码。**也就是说，I/O阻塞了代码的执行，极大地降低了程序的执行效率。**

由于Node.js中采用了非阻塞型I/O机制，因此在执行了访问数据库的代码之后，将立即转而执行其后面的代码，把数据库返回结果的处理代码放在回调函数中，从而提高了程序的执行效率。

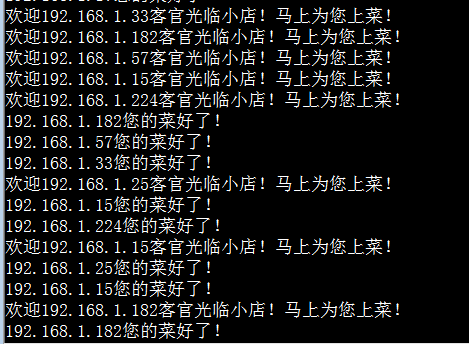
当某个I/O执行完毕时，将以事件的形式通知执行I/O操作的线程，线程执行这个事件的回调函数。为了处理异步I/O，线程必须有事件循环，不断的检查有没有未处理的事件，依次予以处理。

阻塞模式下，一个线程只能处理一项任务，要想提高吞吐量必须通过多线程。**而非阻塞模式下，一个线程永远在执行计算操作，这个线程的CPU核心利用率永远是100%。**所以，这是一种特别有哲理的解决方案：**与其人多，但是好多人闲着；还不如一个人玩命，往死里干活儿。**

## 非阻塞I/O

我们使用fs.readFile()函数，来看看什么是非阻塞I/O。

|  |
| --- |
| // 引用内置模块http模块  var http = require("http");  // file system文件系统模块  var fs = require("fs");  //创建服务器  var server = http.createServer(function(req,res){  console.log("欢迎" + req.connection.remoteAddress + "客官光临小店！马上为您上菜！")  //命令fs进行文件的读取，它是异步语句  fs.readFile("./test.txt",function(err,data){  console.log(req.connection.remoteAddress + "您的菜好了！");  if(err){  console.log("文件读取错误");  return;  }  //输出data，文件内容  res.end(data);  });  });  server.listen(3000,"192.168.1.238"); |



当A用户正在readFile()的时候，此时不会阻塞这个唯一的可怜的进程，如果阻塞了后果不堪设想，因为其他用户进不来了。而node.js是非阻塞I/O，所有语句都是异步的！

当fs把文件读取完毕之后，会执行里面的回调函数。而没有完成的时候，会执行其他人的命令。

|  |
| --- |
| fs.readFile("./test.txt",function(err,data){  console.log(req.connection.remoteAddress + "您的菜好了！");  if(err){  console.log("文件读取错误");  return;  }  //输出data，文件内容  res.end(data);  }); |

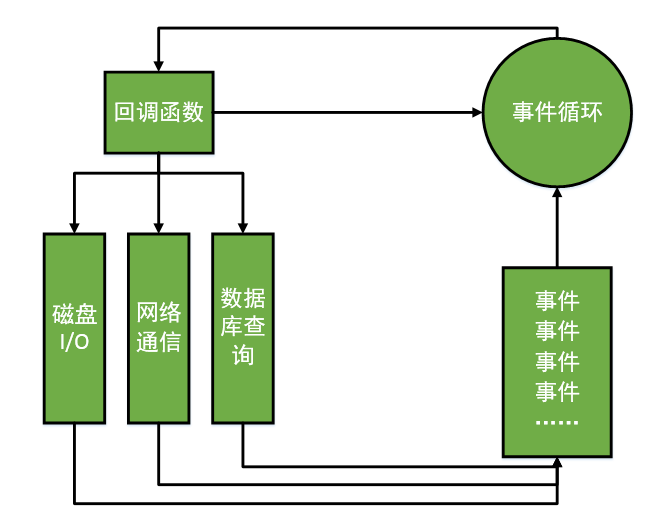
Node.js会非常善于寻找分段点，分段点越多，程序效率越高。



事件驱动event-driven

在Node中，客户端请求建立连接，提交数据等行为，会触发相应的事件。在Node中，在一个时刻，只能执行一个事件回调函数，但是在执行一个事件回调函数的中途，可以转而处理其他事件（比如，又有新用户连接了），然后返回继续执行原事件的回调函数，这种处理机制，称为“事件环”机制。

Node.js底层是C++（V8也是C++写的）。**底层代码中，近半数都用于事件队列、回调函数队列的构建。**用事件驱动来完成服务器的任务调度，这是鬼才才能想到的。针尖上的舞蹈，用一个线程，担负起了处理非常多的任务的使命。



**单线程**，单线程的好处，减少了内存开销，操作系统的内存换页。

如果某一个事情，进入了，但是被I/O阻塞了，所以这个线程就阻塞了。

**非阻塞I/O**， 不会傻等I/O语句结束，而会执行后面的语句。

非阻塞就能解决问题了么？比如执行着小红的业务，执行过程中，小刚的I/O回调完成了，此时怎么办？？

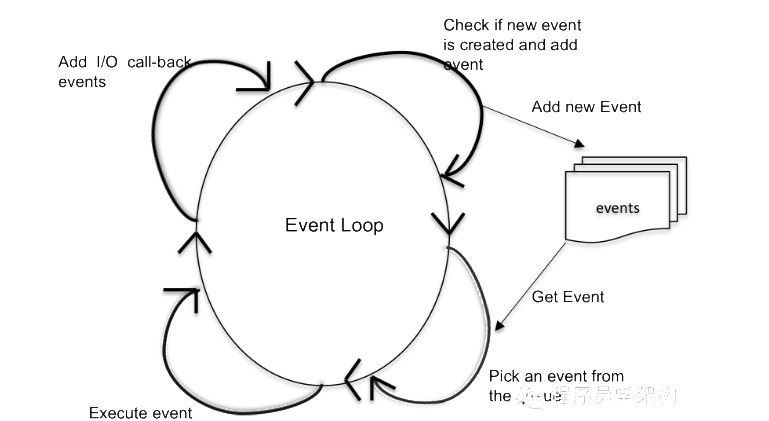
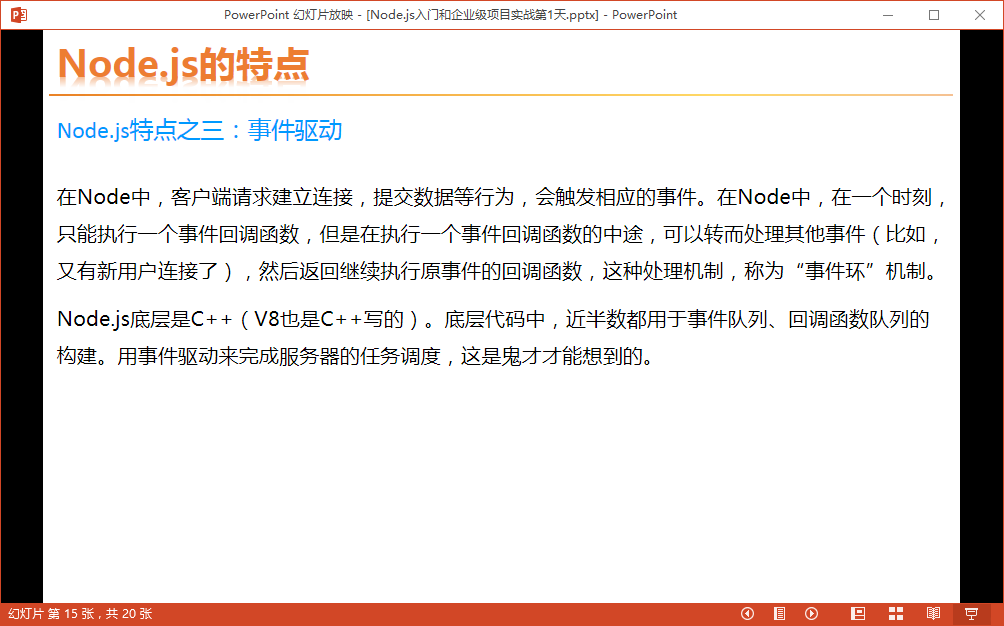
**事件机制，事件环**，不管是新用户的请求，还是老用户的I/O完成，都将以事件方式加入事件环，等待调度。

说是三个特点，实际上是一个特点，离开谁都不行，都玩儿不转了。

Node.js很像抠门的餐厅老板，只聘请1个服务员，服务很多人。结果，比很多服务员效率还高。

Node.js中所有的I/O都是异步的，回调函数，套回调函数。

Node.js是单线程的，所以必须非阻塞I/O，在A用户执行IO的时候，要处理B用户的事情，B用户IO的时候，处理C的。此时A可能IO结束，执行回调函数…… 任务会很乱，所以就必须有一些相应“排班儿”的策略，node.js就是用事件环（事件驱动）来解决这个事情。把所有的 I/O当做事件，当IO执行完毕，就相当于触发了一个事件，进入事件环，等待调用。



Node.js的特点：单线程（single thread） , 非阻塞I/O(non-blocking I/O) ,事件驱动( event driven)。

这些特点其实就是一个特点，彼此互为因果。因为单线程的，所以必须非阻塞I/O，就要排班，就要事件驱动。

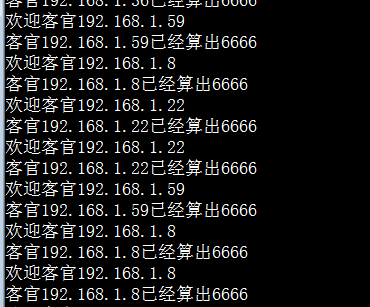
## 1.3 适合开发什么？

Node.js适合用来开发什么样的应用程序呢？

先说不适合的领域，就是计算。当你的业务计算量大的时候，此时CPU被这个计算占用了，其他用户进不来！比如下面的程序用红色区域模拟复杂计算：

|  |
| --- |
| // 引用内置模块http模块  var http = require("http");  //创建服务器  var server = http.createServer(function(req,res){  console.log("欢迎客官" + req.connection.remoteAddress);  while(true){  var num = parseInt(Math.random() \* 9999);  if(num == 6666) break;  }  console.log("客官" + req.connection.remoteAddress + "已经算出6666");  res.end(num.toString());  });  server.listen(3000,"192.168.1.238"); |

欢迎语句和计算出来的语句是挨着的



node适合什么，**I/O越多，越适合！！计算越多，越不适合！！**

善于I/O，不善于计算。因为Node.js最擅长的就是任务调度，如果你的业务有很多的CPU计算，实际上也相当于这个计算阻塞了这个单线程，就不适合Node开发。

**当应用程序需要处理大量并发的I/O，而在向客户端发出响应之前，应用程序内部并不需要进行非常复杂的处理的时候，Node.js非常适合。Node.js也非常适合与web socket配合，开发长连接的实时交互应用程序。**

比如：

● 用户表单收集

● 考试系统

● 聊天室

● 图文直播

● 提供JSON的API（为前台Angular使用）

## 1.4 Node.js无法挑战老牌3P



# 二、Node.js安装

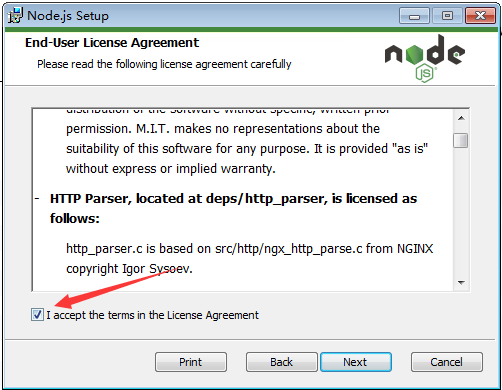
Node.js和Java非常像，跨平台的。不管是Linux还是windows编程是完全一致的（有一些不一样，比如路径的表述）。Linux版本的Node.js环境和windows环境是不一样的，但是编程语言一样。很像Java虚拟机。

我们学习的是最新版，2015年9月19日最新版本4.1.0。

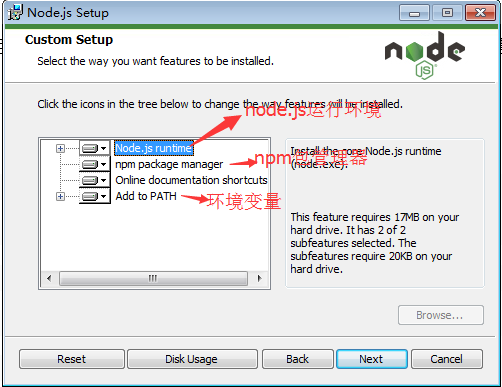
安装包的大小只有10M不到，真的就是一个小玩具。

和PHP一样，Node.js要配置安装环境，极其简单！任何系统下windows、linux下都简单，linux讲1天。

window下，下载一个msi文件的安装包。



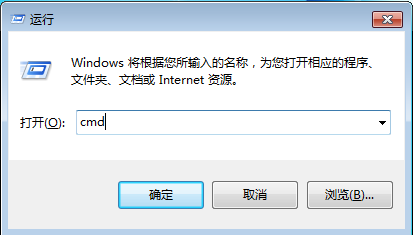
安装路径一定不能是中文的。默认：C:\Program Files\nodejs\ 。 最好别改动。



如果电脑是32位的，用另外一个安装包。

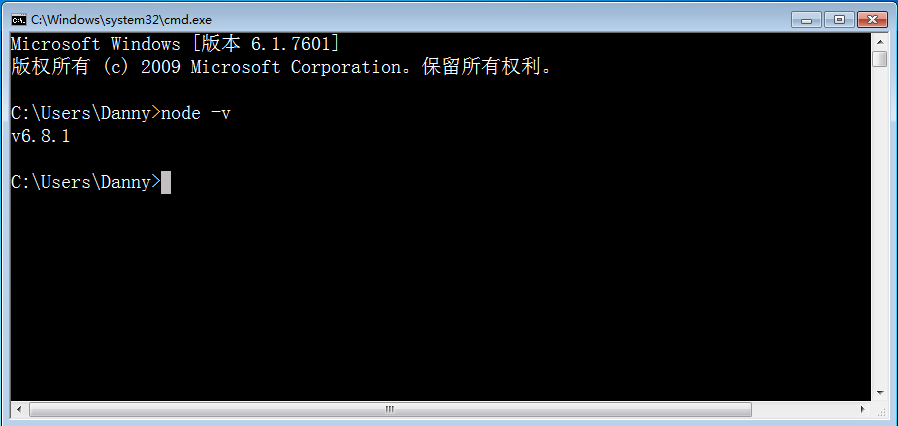
此时就要验证node.js是否已经安装成功，步骤是：

1. 按windows键+R键，运行窗口被打开。
2. 输入cmd，表示打开命令行，终端。



1. 在打开的命令行窗口中，输入

|  |
| --- |
| node -v |

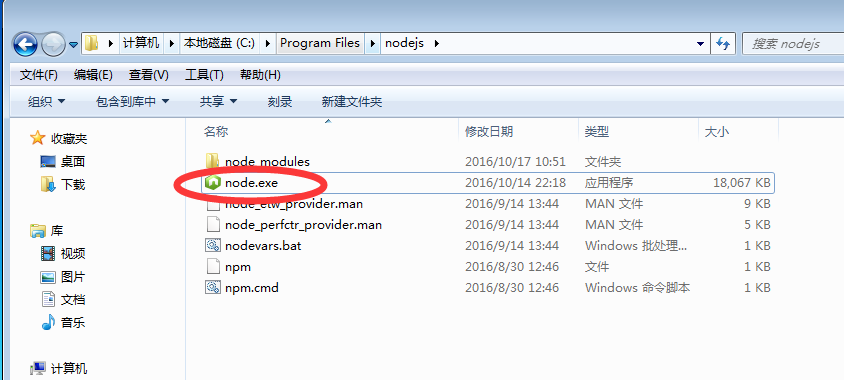


此时会显示版本号，显示出来了就说明装好了。

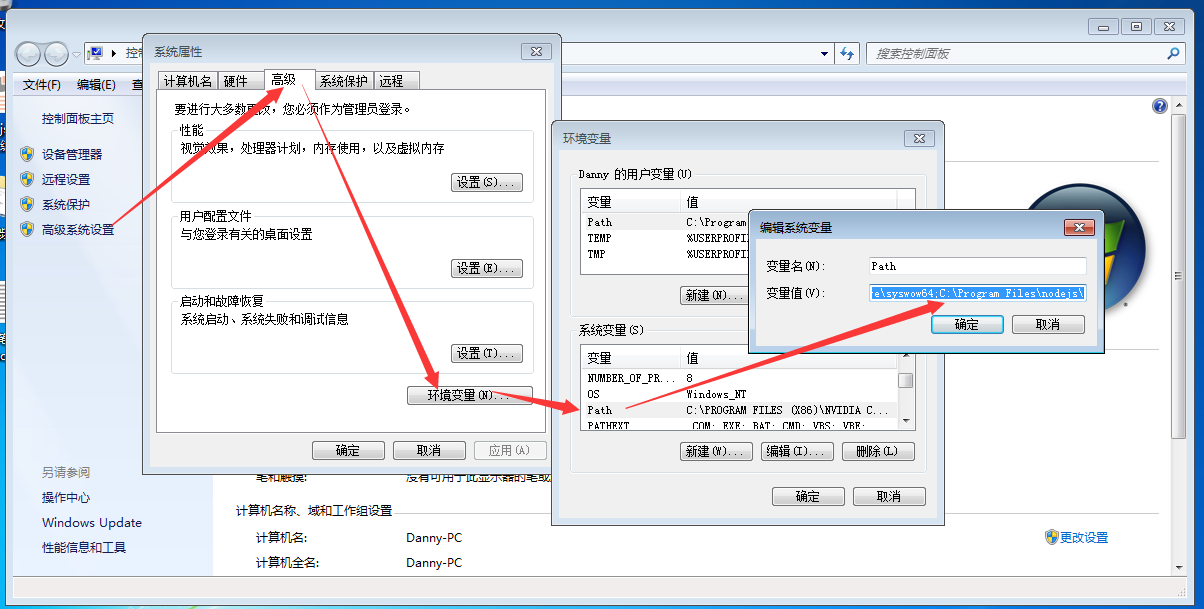
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*下面的知识不需要操作，看看了解一下\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

本质是什么呢？为什么能运行node命令了呢？

安装目录：



我们的nodejs安装包非常贴心，帮我们自动设置了环境变量。



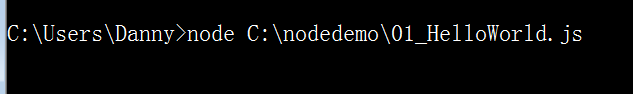
所谓的系统环境变量，就是一个个路径，中间用;隔开。这些路径里面的exe程序，可以在cmd控制台中的任何路径下调用！比如，你现在是c:\users\danny文件夹，也能用c:\program files\nodejs里面的程序。

如果没有环境变量，我们只能手动寻找node.exe：

使用cd命令可以切换文件夹，cd就是change directory的意思。



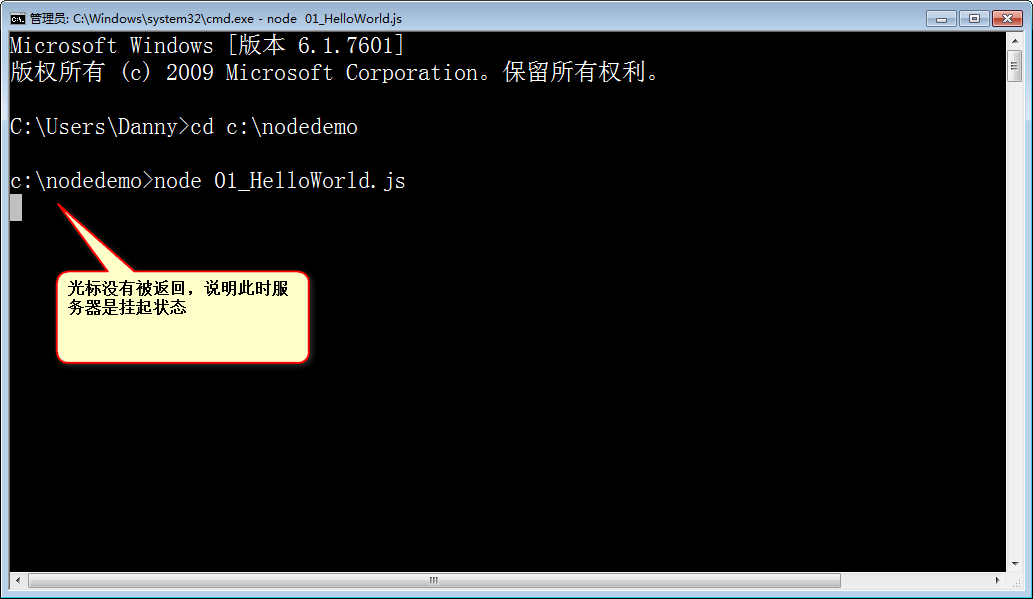
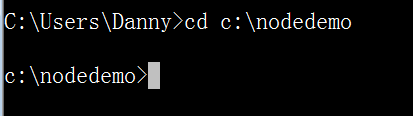
运行文件，就要用node命令来运行：



|  |
| --- |
| 1. node 路径名 |

推荐，不要使用完整的路径名，而是通过先进入案例文件夹，然后node相对地址

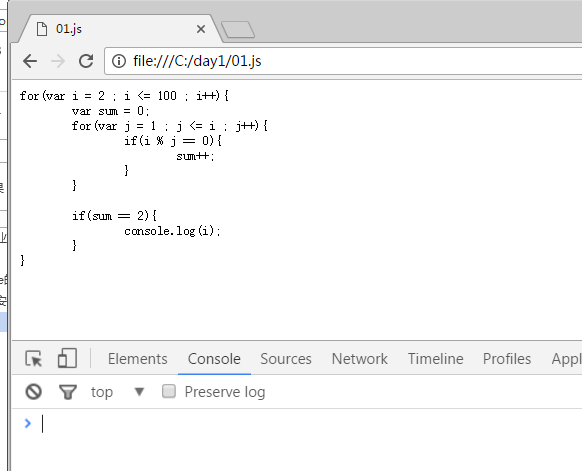
cd命令，就是change directory的缩写，表示更改当前目录。



# 三、HelloWorld

JavaScript是需要运行的宿主环境的！不能独立运行。JS是一个脚本语言，跟C语言、Java不一样。C语言是自己要经过编译，就能运行；Java运行在JVM里面。而JS和他们都不一样，它是一个脚本语言，运行在哪里是不限制的。

我们现在学习过的宿主环境只有一个，就是浏览器环境。我们写一个找质数的js程序，然后发现js文件不能直接用浏览器运行哦，会源码显示，而不是在控制台输出：



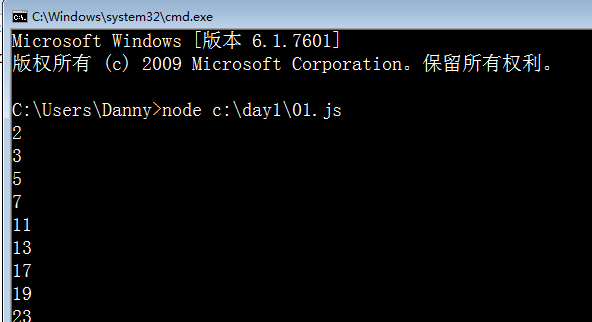
我们必须写一个html页面，然后用script标签引用js才能运行js。这是浏览器环境运行js的方法。那么，除了浏览器之外，还有没有别的环境能够运行js呢，有，就是我们今天学习的node.js！

node.js是一个新的js宿主环境！可以独立于浏览器运行JavaScript。

运行js的方法是，打开cmd，就是控制台，也叫作终端。直接输入node文件名就行，**运行谁就node谁**。

|  |
| --- |
| node c:\day1\01.js |

就可以在cmd中运行这个找质数的js程序。这就是node环境。图示如下：

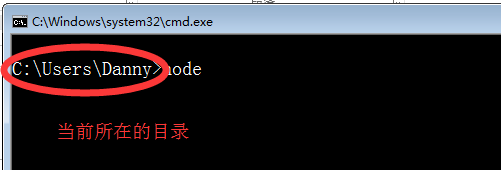


如果控制台非常乱，不爽，就可以使用清屏命令：

|  |
| --- |
| cls |

一下子

使用node命令的时候，一定要注意此时的光标位置：



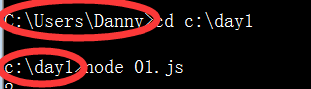
你的光标在c:\users\danny ，而案例在c:\day1里面，所以此时两种办法：

1. 绝对路径法：



如果懒得输入绝对路径，此时可以把文件“㩐”到cmd里面，会自动补全它的路径。

1. 使用cd改变光标目录，



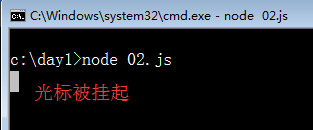
|  |
| --- |
| cd c:\day1 |

一旦光标到了c:\day1里面，此时可以使用相对路径找到这个文件。

|  |
| --- |
| node 01.js |

上面我们写了一个找质数的程序，现在我们来真格的，写一个服务器。

|  |
| --- |
| //引用内置http模块，require表示引用  var http = require("http");  //创建一个服务器  var server = http.createServer(function(req,res){  res.end("你好，我要好好学习node.js");  });  //监听3000端口  server.listen(3000,"127.0.0.1"); |



Node.js是服务器的程序，写的js语句，都将运行在服务器上。返回给客户的，都是已经处理好的纯html。

|  |
| --- |
| 1. //require表示引包，引包就是引用自己的一个特殊功能 2. var http = require("http"); 3. //创建服务器，参数是一个回调函数，表示如果有请求进来，要做什么 4. var server = http.createServer(function(req,res){ 5. //req表示请求，request; res表示响应，response 6. //设置HTTP头部，状态码是200，文件类型是html，字符集是utf8 7. res.writeHead(200,{"Content-type":"text/html;charset=UTF-8"}); 8. res.end("哈哈哈哈，我买了一个iPhone" + (1+2+3) + "s"); 9. }); 10. //运行服务器，监听3000端口（端口号可以任改） 11. server.listen(3000,"127.0.0.1"); |

如果想修改程序，必须中断当前运行的服务器，重新node一次，刷新，才行。

**ctrl+c，就可以打断挂起的服务器程序。**此时按上箭头，能够快速调用最近的node命令。

你会发现，我们本地写一个js，打死都不能直接拖入浏览器运行，但是有了node，我们任何一个js文件，都可以通过node来运行。**也就是说，node就是一个js的执行环境。**

我们现在，要跑起来一个服务器，这个服务器的脚本，要以.js存储。是一个js文件。用node命令运行这个js文件罢了。

Node.js没有根目录的概念，因为它根本没有任何的web容器！

让node.js提供一个静态服务，都非常难！

也就是说，node.js中，如果看见一个网址是

|  |
| --- |
| 1. 127.0.0.1:3000/fang |

别再去想，一定有一个文件夹，叫做fang了。可能/fang的物理文件，是同目录的test.html

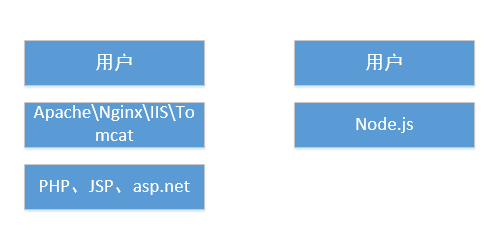
URL和真实物理文件，是没有关系的。URL是通过了Node的顶层路由设计，呈递某一个静态文件的。

# 三、HTTP模块

Node.js中，将很多的功能，划分为了一个个mudule，大陆的书翻译为模块；台湾的书，翻译为模组。

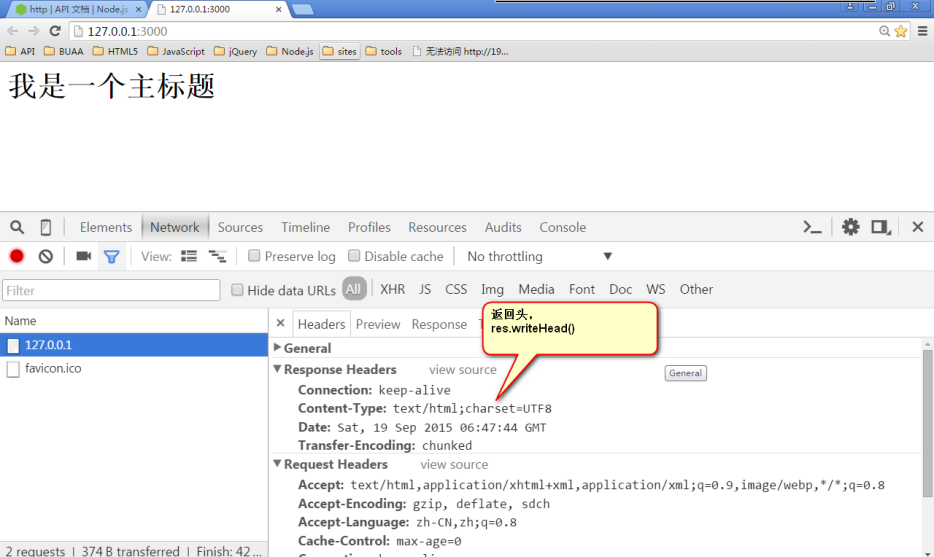
这是因为，有一些程序需要使用fs功能（文件读取功能），有一些不用的，所以为了效率，你用啥，你就require啥。

|  |
| --- |
| 1. //这个案例简单讲解http模块 2. //引用模块 3. var http = require("http"); 4. //创建一个服务器，回调函数表示接收到请求之后做的事情 5. var **server** = http.**createServer**(function(req,res){ 6. //req参数表示请求，res表示响应 7. console.log("服务器接收到了请求" + req.url); 8. res.end(); 9. }); 10. //监听端口 11. **server.listen**(3000,"127.0.0.1"); |



设置一个响应头：

|  |
| --- |
| 1. res.writeHead(200,{"Content-Type":"text/plain;charset=UTF8"}); |



我们现在来看一下req里面能够使用的东西。

最关键的就是req.url属性，表示用户的请求URL地址。所有的路由设计，都是通过req.url来实现的。

我们比较关心的不是拿到URL，而是识别这个URL。

识别URL，用到两个新模块，第一个就是url模块，第二个就是querystring模块

字符串查询，用querystring处理

|  |
| --- |
| 1. querystring.parse('foo=bar&baz=qux&baz=quux&corge') 2. // returns 3. { foo: 'bar', baz: ['qux', 'quux'], corge: '' } 4. // Suppose gbkDecodeURIComponent function already exists, 5. // it can decode `gbk` encoding string 6. querystring.parse('w=%D6%D0%CE%C4&foo=bar', null, null, 7. { decodeURIComponent: gbkDecodeURIComponent }) 8. // returns 9. { w: '中文', foo: 'bar' } |

