

单一职责原则

文件

index.js 接收请求

package.json

文件夹

route 分配页面（route.js） 功能（file.js）

public 静态资源

views 模板

temp 暂存

**NodeJS服务端程序开发课程**

**（讲义）**

**目录**

[一、 NodeJS简介 3](#_Toc477125710)

[1.1 简介 3](#_Toc477125711)

[1.2 特性 3](#_Toc477125712)

[1.3 应用场景 5](#_Toc477125713)

[1.4 Node.js无法挑战老牌3P 6](#_Toc477125714)

[二、 NodeJS的安装 6](#_Toc477125715)

[三、 REPL环境 7](#_Toc477125716)

[3.1 REPL基本操作 7](#_Toc477125717)

[3.2 REPL操作变量 7](#_Toc477125718)

[3.3 REPL环境中的下划线字符 7](#_Toc477125719)

[3.4 退出REPL环境 8](#_Toc477125720)

[四、 NodeJS基础知识 8](#_Toc477125721)

[4.1 NodeJS中的控制台 8](#_Toc477125722)

[4.1.1 console.log() 8](#_Toc477125723)

[4.1.2 console.error() 8](#_Toc477125724)

[4.1.3 console.dir() 8](#_Toc477125725)

[4.1.4 console.time()与console.timeEnd()方法 8](#_Toc477125726)

[4.1.5 console.trace()方法 8](#_Toc477125727)

[4.1.6 console.assert ()方法 9](#_Toc477125728)

[4.2 全局作用域与全局函数 9](#_Toc477125729)

[4.2.1 全局作用域 9](#_Toc477125730)

[4.2.2 setTimeout函数与clearTimeout函数 9](#_Toc477125731)

[4.2.3 setInterval函数与clearInterval函数 9](#_Toc477125732)

[4.2.4 unref函数与ref函数 9](#_Toc477125733)

[4.2.5 与模块相关的全局函数及对象 9](#_Toc477125734)

[4.3 \_\_filename变量与\_\_dirname变量 10](#_Toc477125735)

[4.3.1 \_\_filename变量 10](#_Toc477125736)

[4.3.2 \_\_dirname变量 10](#_Toc477125737)

[4.4 事件处理机制及事件环机制 10](#_Toc477125738)

[4.4.1 EventEmiter类 10](#_Toc477125739)

[4.4.1 事件环机制 11](#_Toc477125740)

[五、 NodeJS模块机制 12](#_Toc477125741)

[5.1模块化规范： 12](#_Toc477125742)

[5.2 模块的引入： 12](#_Toc477125743)

[5.3 exports： 12](#_Toc477125744)

[5.4 module.exports 13](#_Toc477125745)

[六、 HTTP服务器 13](#_Toc477125746)

[6.1创建HTTP服务器 13](#_Toc477125747)

[七、 Buffer类处理二进制 15](#_Toc477125748)

[7.1 创建Buffer对象 15](#_Toc477125749)

[7.2 字符串长度与缓存区的长度 16](#_Toc477125750)

[7.3 Buffer对象与字符串对象之间的相互转换 17](#_Toc477125751)

[7.3.1 toString方法 17](#_Toc477125752)

[7.3.2 write方法 17](#_Toc477125753)

[7.3.3 StringDecoder方法 17](#_Toc477125754)

[7.4 Buffer对象与数值对象之间的相互转换 17](#_Toc477125755)

[7.5 Buffer对象与JSON对象之间的相互转换 18](#_Toc477125756)

[7.6 复制缓存数据 18](#_Toc477125757)

[7.7 Buffer类的类方法 19](#_Toc477125758)

[7.7.1 isBuffer方法 19](#_Toc477125759)

[7.7.2 byteLength方法 19](#_Toc477125760)

[7.7.3 concat方法 19](#_Toc477125761)

[7.7.4 isEncoding方法 20](#_Toc477125762)

[八、 FS文件系统 20](#_Toc477125763)

[8.1 读写操作 20](#_Toc477125764)

[8.1.1 文件的完整读写 20](#_Toc477125765)

[8.1.2 从指定位置处开始读写文件 22](#_Toc477125766)

[8.2 创建与读取目录 24](#_Toc477125767)

[8.2.1 创建目录 24](#_Toc477125768)

[8.2.2 读取目录 25](#_Toc477125769)

[8.3 查看与修改文件或目录的信息 25](#_Toc477125770)

[8.3.1 查看文件或目录信息 25](#_Toc477125771)

[8.3.2 检查文件或目录是否存在 26](#_Toc477125772)

[8.3.3 获取文件或目录的绝对路径 26](#_Toc477125773)

[8.3.4 修改文件访问时间及修改时间 26](#_Toc477125774)

[8.3.5 修改文件或目录的读写权限 27](#_Toc477125775)

[8.4 移动文件或目录 27](#_Toc477125776)

[8.5 创建与删除文件的硬链接 27](#_Toc477125777)

[8.6 截断文件 28](#_Toc477125778)

[8.7 删除空目录 28](#_Toc477125779)

[8.8 监视文件或目录 28](#_Toc477125780)

[8.9 使用文件流 28](#_Toc477125781)

[8.9.1 流的基本概念 28](#_Toc477125782)

[8.9.2 使用ReadStream对象读取文件 30](#_Toc477125783)

[8.9.3 使用WriteStream对象写入文件 31](#_Toc477125784)

[8.9.4 pipe管道 32](#_Toc477125785)

[九、 Path模块路径处理 32](#_Toc477125786)

[9.1 path.normalize(path) 32](#_Toc477125787)

[9.2 path.join([...paths]) 32](#_Toc477125788)

[9.3 path.resolve([...paths]) 32](#_Toc477125789)

[9.4 path.relative(from,to) 33](#_Toc477125790)

[9.5 path.dirname(path) 34](#_Toc477125791)

[9.6 path.basename(path[, ext]) 34](#_Toc477125792)

[9.7 path.extname(path) 34](#_Toc477125793)

[十、Express框架 34](#_Toc477125794)

[10.1 路由能力： 34](#_Toc477125795)

[10.2 静态文件伺服能力： 35](#_Toc477125796)

[10.3 模板引擎： 35](#_Toc477125797)

[十一、路由 35](#_Toc477125798)

[十二、中间件 37](#_Toc477125799)

[十三、GET请求和POST请求的参数 39](#_Toc477125800)

[十四、传统数据库技术回顾 39](#_Toc477125801)

[十五、NoSQL 41](#_Toc477125802)

[十六、MongoDB安装 42](#_Toc477125803)

[十七、数据库使用 43](#_Toc477125804)

[17.1 插入数据 44](#_Toc477125805)

[17.2 查找数据 45](#_Toc477125806)

[17.3 修改数据 45](#_Toc477125807)

[17.4 删除数据 45](#_Toc477125808)

[十八、Mongoose 46](#_Toc477125809)

[18.1 数据库连接 47](#_Toc477125810)

[18.2 定义模型 47](#_Toc477125811)

[十九、Cookie和Session 48](#_Toc477125812)

[19.1 Cookie 48](#_Toc477125813)

[19.2 Session 49](#_Toc477125814)

[19.3 session 50](#_Toc477125815)

1. NodeJS简介
   1. 简介

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. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。

Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。

Node.js 的包管理器 npm，是全球最大的开源库生态系统。

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。

官方网站：<https://nodejs.org/en/>

中文网站：<http://nodejs.cn/>

* 1. 特性

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。

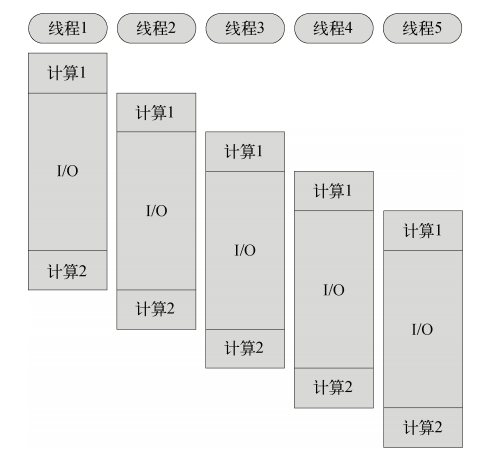
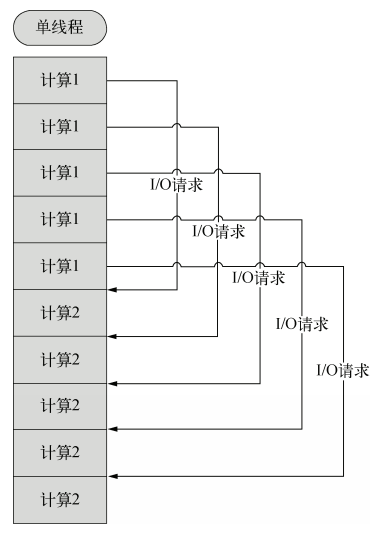
1.2.1 单线程

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”。

1.2.2 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。**

1.2.3 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。

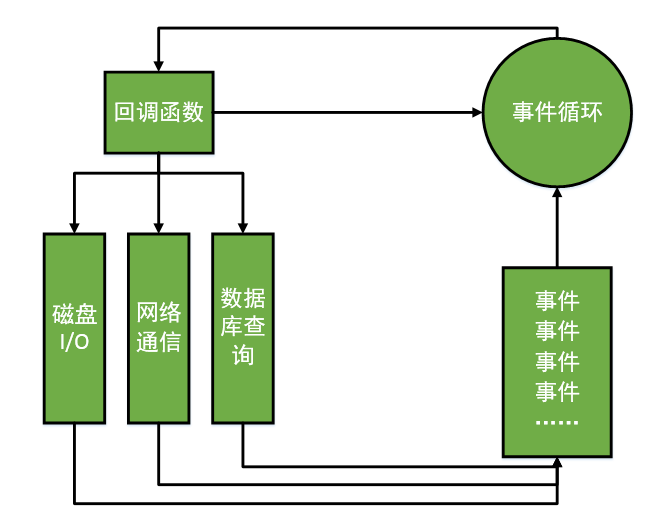
非阻塞就能解决问题了么？比如执行着小红的业务，执行过程中，小刚的I/O回调完成了，此时怎么办？？

**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。



* 1. 应用场景

Node.js适合用来开发什么样的应用程序呢？

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。

Node适合处理**高并发**场景，比如天猫双11，同时有很多用户都在发送请求。

**不适合计算。**

比如：

● 用户表单收集

● 考试系统

● 聊天室

● 图文直播

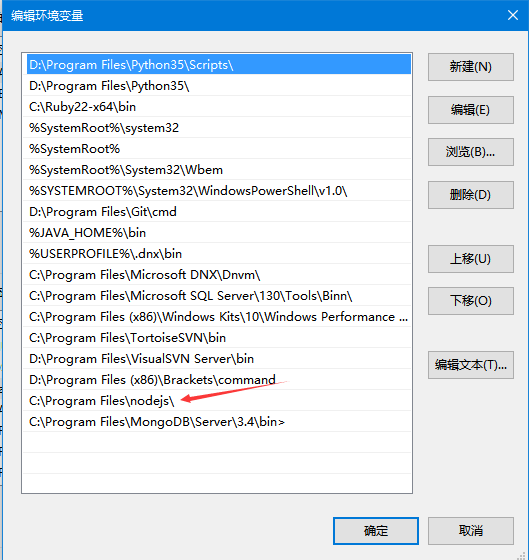
● 提供JSON的API（为前台Angular使用）

## 1.4 Node.js无法挑战老牌3P



1. NodeJS的安装

装完之后，我们在系统的环境变量里面，我们看一下：



环境变量：就是在系统的任何目录下，都能运行C:\Program Files\nodejs里边的node.exe

系统 -> 高级系统设置 -> 环境变量 -> path

1. REPL环境

NodeJS为我们提供了REPL（Read-Eval-Print-Loop）**可交互运行环境**。我们可以在REPL环境中输入JS表达式，回车，就可以看到计算结果

* Read 读，读取用户输入，解析输入的JS数据结构并存储在内存中
* Eval 执行， 执行输入的数据结构
* Print 打印，输出结果
* Loop 循环，循环操作以上步骤，直到ctrl+c以后

3.1 REPL基本操作

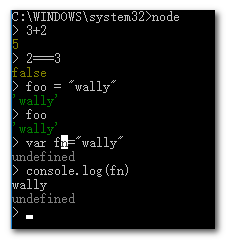
打开命令行，输入node，回车，显示“>”，就进入了NodeJS的REPL环境中。

在任何时候，当你输入表达式然后按下回车，REPL环境就会表达式的计算结果。就像: 3==2 输出false

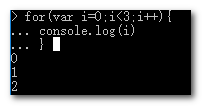
3.2 REPL操作变量

在REPL环境中，可以定义变量，如foo=”wally”，可以输入变量名foo，查看他的值。

如果用var声明变量，输出结果为undefined。这是因为REPL内部是使用eval解析。要想查看var声明变量的值，只能使用console.log查看。



在REPL环境中要输入多行命令，使用ctrl+enter换行。同样REPL可以执行函数、循环等命令



3.3 REPL环境中的下划线字符

在REPL环境中，可以通过“\_”来访问最近使用的表达式。如：

|  |
| --- |
| 1. > a=3 2. 3 3. > \_+=1 4. 4 5. > \_+=1 6. 5 7. > a 8. 3 |

从上面执行结果来看，输入下划线并不意味着可以修改变量的值。

3.4 退出REPL环境

**Ctrl+C** 两次退出REPL环境

1. NodeJS基础知识

4.1 NodeJS中的控制台

4.1.1 console.log()

console.log方法用于进行**标准输出流的输出，**就是在控制台显示一行字符串

4.1.2 console.error()

console.error方法用于进行**标准错误输出流的输出，**就是在控制台显示一行错误信息

4.1.3 console.dir()

console.dir方法用于**查看一个对象中的内容**，并将该对象的信息输出到控制台中。

**我们写的JS文件不能直接拖到浏览器中运行。但是NodeJS可以神奇的将js文件运行在服务器环境中。**

按住**shift + 右击**， 选择在此处打开 命令窗口，命令行直接就在当前路径下。

**Node执行一个JS文件的方式，命令行中找到要执行的JS文件所在的目录，运行命令node “文件名”,按下回车就可以执行这个JS文件中的代码。**如： > node test.js

|  |
| --- |
| 1. var obj = new Object() 2. obj.name = "wally" 3. obj.sayName = function(){ 4. console.log(this.name) 5. } 6. console.dir(obj) //输出结果是 ： { name: 'wally', sayName: [Function] } |

4.1.4 console.time()与console.timeEnd()方法

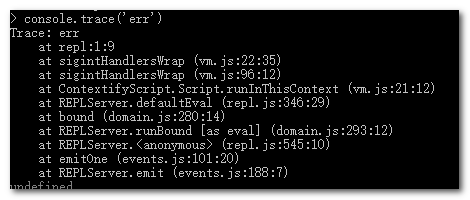
console.time()标记开始时间，console.timeEnd()标记结束时间，并且会将结束时间与开始时间之间经过的毫秒数输出到控制台。

|  |
| --- |
| 1. console.time() 2. for(var i=0;i<20000;i++){ 3. i\*i 4. } 5. console.timeEnd() |

4.1.5 console.trace()方法

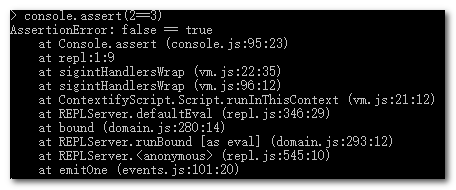
Console.trace()方法用于将当前位置处的栈信息作为标准错误信息进行输出。

该方法中使用参数，参数值可以是任何字符串，用于标识此处输出的标准错误信息。



4.1.6 console.assert ()方法

console.assert ()用于对一个表达式的执行结果进行评估，如果是结果是false，这输出一个消息字符串并抛出AssertionError异常。



4.2 全局作用域与全局函数

4.2.1 全局作用域

在Node中也存在一个全局作用域，即可以定义一个不需要模块加载就能使用的变量、函数或类。同时也预先定义了一些全局方法与全局类。

我们知道在浏览器中window全局对象。在Node中global对象代表了node的全局命名空间。也就是说任何函数、变量或对象都是该对象的一个属性值。

Console.log(global)

4.2.2 setTimeout函数与clearTimeout函数

与JS用法一致

4.2.3 setInterval函数与clearInterval函数

与JS用法一致

4.2.4 unref函数与ref函数

可以使用setTimeout或setInterval函数返回的定时器对象（也就是示例代码中的timer）的unref方法，取消setTimeout或setInterval中指定回调函数的调用。

当然也可以使用timer的ref()方法，恢复回调函数的调用。

|  |
| --- |
| 1. var timer = setInterval(function(){ 2. console.log(new Date()) 3. },1000) 4. timer.unref() 5. timer.ref() |

4.2.5 与模块相关的全局函数及对象

1）使用require加载模块

在NodeJS中每个文件都是一个独立的模块。

2）使用require.resolve函数查询完整模块名

使用该函数查询某个模块文件的带有完整绝对路径的文件名



3）require.cache对象

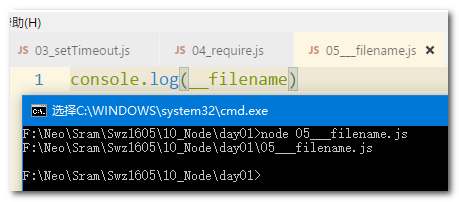
**在nodeJS中，模块只要require一次，就会被缓存起来，不存在重复require的情况。**

该方法代表了缓存了所有已被加载模块的缓存区。可以使用console.log(require.cache)查看缓存区的内容。

4.3 \_\_filename变量与\_\_dirname变量

4.3.1 \_\_filename变量

在任何模块内，可以使用\_\_filename变量获取当前模块文件的带有完整路径的文件名



4.3.2 \_\_dirname变量

在任何模块内，可以使用\_\_dirname变量获取当前模块文件所在目录的完整绝对路径。

4.4 事件处理机制及事件环机制

4.4.1 EventEmiter类

NodeJS中用于实现各种事件处理的**events模块**中，定义了一个EventEmiter类。所有可能触发事件的对象都是一个继承了EventEmiter类的之类的实例对象。EventEmiter的各种方法见下表（表中event代表事件名，listener代表事件处理函数）

表-EventEmiter的各种方法

|  |  |
| --- | --- |
| **方法名及参数** | **描述** |
| addListener(event,listener) | 对指定事件绑定事件处理函数 |
| on(event,listener) | 对指定事件绑定事件处理函数（addListener的别名） |
| once(event,listener) | 对指定事件指定只执行一次的事件处理函数 |
| removeListener(event,listener) | 对指定事件解绑事件处理函数 |
| setMaxListener(n) | 指定事件处理函数的最大数量。N为整数值，代表的是最大的可指定事件处理函数的数量 |
| listeners(event) | 获取指定事件的所有事件处理函数 |
| emit(event,[arg1],[arg2],…) | 手工触发指定事件 |

remvoeListener()：**是对指定事件解绑事件处理函数，解绑的是事件处理函数，而不是事件**

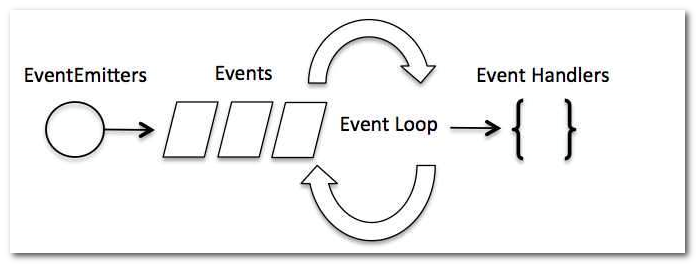
**注意，一旦一个事件被触发，所有绑定到它的监听器都会按顺序依次触发。 这意味着，在事件触发后、最后一个监听器完成执行前，任何 removeListener() 或 removeAllListeners() 调用都不会从 emit() 中移除它们。 随后的事件才会像预期的那样发生。**

|  |
| --- |
| 1. //1.引入events模块 2. const events = require('events') 3. //2.实例化 4. const emitter = new events.EventEmitter() 5. //解绑指定事件的 事件处理函数 6. function cbA(){ 7. console.log("A") 8. emitter.removeListener('eventName',cbB) 9. } 10. function cbB(){ 11. console.log("B") 12. } 13. //3. 绑定事件 14. emitter.on('eventName',cbA) 15. emitter.on('eventName',cbB) 16. //4. 发射，手动触发事件 17. console.log(emitter.listeners('eventName')) //[ [Function: cbA], [Function: cbB] ] 18. emitter.emit('eventName') //A B 19. console.log(emitter.listeners('eventName')) //[ [Function: cbA] ] 20. emitter.emit('eventName') //A |

4.4.1 事件环机制

NodeJS中，采用非阻塞I/O机制，就意味着所有要求应用程序进行的处理，比如说HTTP请求、数据库查询、文件的输入/输出等，都不会再处理结束之前阻碍其他任务的进行。也就是说，这些处理都是独立进行。但处理结束时，会触发回调函数。也就是说，在NodeJS中，我们所编写的是各种I/O事件的回调函数。

邮递员例子。



1. NodeJS模块机制

NodeJS有两种模块：原生模块（NodeJS自带的模块）和文件模块（动态模块，自己写的JS文件）。

原生模块不需要写路径，node.js自己会为我们找到模块并引入。

**文件模块必须要写路径**，否则会报错。JS文件的**后缀名**可以**省略**不写。

5.1模块化规范：

* CommonJS NodeJS采用的模块化规范
* CMD sea.js采用
* AMD require.js

5.2 模块的引入：

引入**原生模块**使用require(‘模块名’)，比如require(‘http’)

引入**文件模块**使用require(‘文件路径/文件名’)，如require(‘./src/a’)

5.3 exports：

文件模块（如a）中的变量或者函数，要想被别的模块(如b)使用。那么首先a中要被引用的变量或者函数要使用exports暴露出来，b中通过require应用a模块（如：var module = require(‘./src/a’)）,那么这里声明的变量module就代表a模块中的exports对象。

|  |
| --- |
| 1. //文件a.js 2. var str = “wally” 3. **exports**.str = str; //将str挂到exports对象下，暴露出去，以便引用他的模块使用 4. //文件b.js 5. var a = **require**(‘./a’) 6. console.log(a) //{str:”wally”} |
| 1. console.log(a.str) //wally |

例如

|  |
| --- |
| 1. //a.js 2. var str = "wally" 3. function show(){ 4. console.log("hello i'm show function") 5. } 6. exports.str = str; 7. exports.key = show; 8. //exports -> {str:"wally",key:function show(){}} 9. //b.js 10. var a = require('./src/a') //文件模块 11. console.log(a) //a -> { str: 'wally', key: [Function: show] } 12. a.key() |

5.4 module.exports

可以理解成module.exports能够替代exports，但是exports不能完全替代module.exports。可以理解成一个“包含关系”。

**module.exports可以返回其他类型的数据，exports返回只能是个对象。一般不操作module.exports**

|  |
| --- |
| 1. module.exports=["a"] 2. console.log(module.exports) //["a"] |

1. HTTP服务器

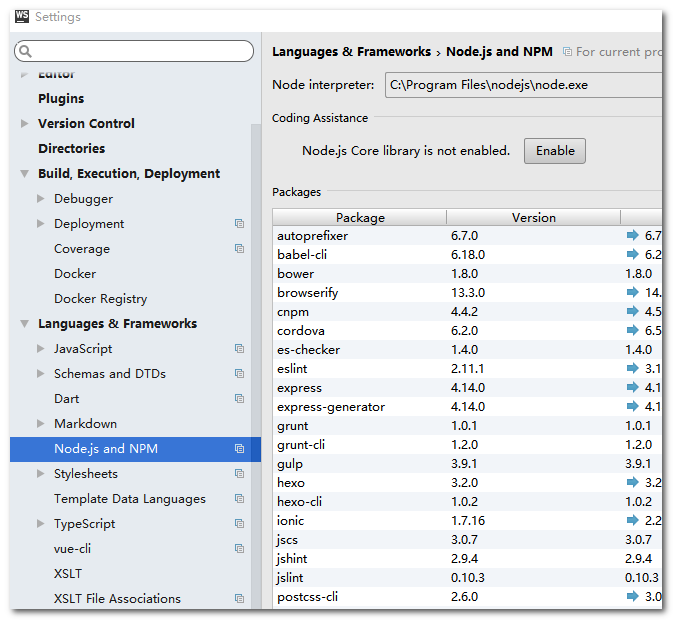
6.1创建HTTP服务器

**http 1.0 发送一次请求，接收一个文件**

**http 1.1 发送一次请求，接收多个文件**

**http 2.0 html5**

webstorm settings -> languages&frameworks -> node.js and npm -> enable



|  |
| --- |
| 1. const server = http.createServer(function (req,res) { 2. //req -> request 客户端请求 3. //res -> response 服务端响应 4. }) |

http.createServer()中接受一个可选参数，回调函数。回调函数中由两个参数：req, res

req -> request 客户端请求

res -> response 服务端响应

上述代码中的**server**为createServer所创建的**服务器对象**

|  |
| --- |
| 1. server.listen(3000,'127.0.0.1',511,function () { 3. }) |

**server.listen(port,[host],[backlog],[callback])**

* port 必选参数，监听的端口号
* host 可选 指定监听的地址
* backlog 可选 一个整数值 指定位于等待队列中的客户端连接的最大数量，如果超过这个数量，就会拒绝新的连接请求，默认值511
* callback 可选 对HTTP服务器指定了端口跟地址，服务器端就会立即开始监听来自于该地址的客户端连接，触发listening。可以使用listen的callback来指定listening事件触发时调用的回调函数。、

**res.writeHead(statusCode,[reasonPhrase],[headers])**

* statusCode，必选， 指定一个三位的HTTP状态码，如200,404
* reasonPhrase，可选， 指定状态码的描述信息
* headers，可选，一个对象，指定服务器端创建的响应头对象

content-type：指定内容类型

content-length：指定服务端响应内容的字节数

set-cookie：在客户端创建一个cookie

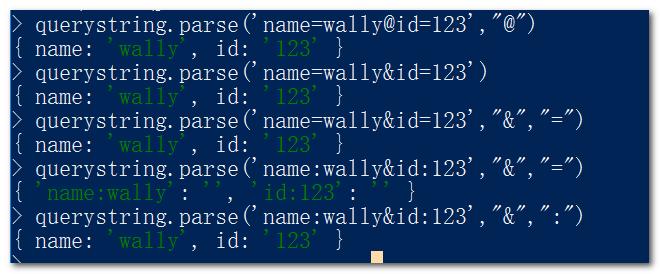
content-encoding：指定服务器端响应内容的编码方式

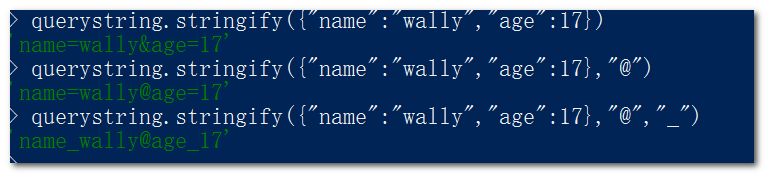
expires：指定缓存过期时间

**res.write(data)**： 发送响应内容，data为显示的内容

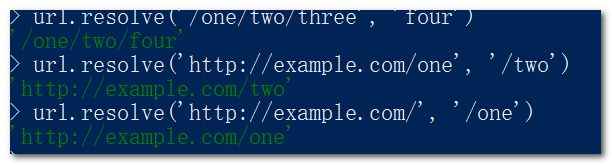
**res.end()** 结束响应，可以添加参数，参数为返回的内容，如果不设置服务器会被挂起

|  |
| --- |
|  |







1. **Buffer类处理二进制**

纯 Javascript 语言对 Unicode 友好，但是难以处理二进制数据。在处理 TCP 流和文件系统时经常需要操作字节流。Node 提供了一些机制，用于操作、创建、以及消耗字节流。

在 Buffer 类实例化中存储了原始数据。 Buffer 类似于一个整数数组，在 V8 堆（the V8 heap）原始存储空间给它分配了内存。一旦创建了 Buffer 实例，则无法改变大小。

Buffer 类是全局对象，所以访问它不必使用 require('buffer') 。

**Node存储二进制的中介。**

7.1 创建Buffer对象

* **new Buffer(size)**

被创建的Buffer对象拥有一个length属性，属性值是缓存区大小。

|  |
| --- |
| 1. var bf=new Buffer(128) 2. console.log(bf) |

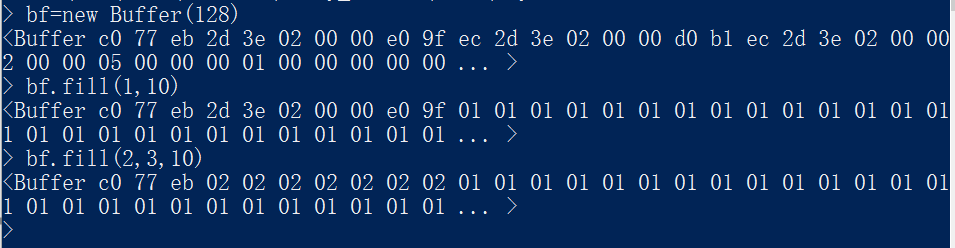
**bf.fill(value,[offset],[end])**

Buffer对象的fill方法**初始化缓存区中的所有内容**，接受三个参数：

一参：必选，需要被写入的数值

二参：可选，指定从第几个字节处开始写入被指定的数值，默认值是0，即从缓存区的起始位置写入

三参：可选，指定将数值一直写入到第几个字节处，默认值是Buffer对象的大小



* **new Buffer(array)**

使用一个数组来初始化缓存区



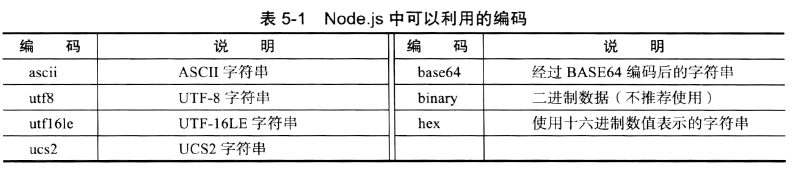
* **new Buffer(str,[encoding])**

使用一个字符串初始化缓存区。接受两个参数：

一参：必选，用于初始化缓存区的字符串

二参：指定文字编码格式的字符串，默认“utf8”。

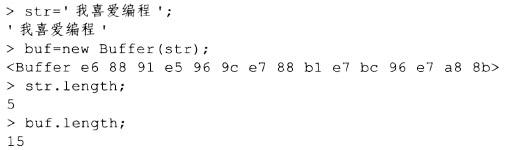
在Node中可以使用的编码：





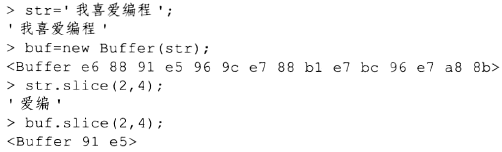
7.2 字符串长度与缓存区的长度

在Node中，一个**字符串的长度**与根据该字符串说创建的**缓存区的长度并不相同**。因为在计算**字符串的长度**时，**以文字作为一个单位**，而在计算**缓存区的长度**时，**以字节作为一个单位**。



**bf.slice(start,end)**

**slice方法并不是复制缓存区中的数据，而是与该数据共享内存区域，因此，如果修改使用slice方法取出的数据，则缓存区中保存的数据也将被修改。**



7.3 Buffer对象与字符串对象之间的相互转换

**7.3.1 toString方法**

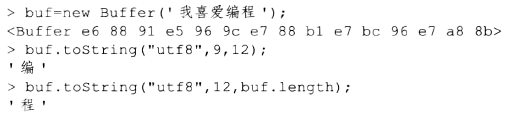
可以使用toString方法将Buffer对象中保存的数据转换成字符串。

**bf.toString([encoding],[start],[end])**

**一参：可选，文字编码格式，默认utf8**

**二参：可选，指定被转换数据的起始位置**

**三参：可选，指定被转换数据的终止位置**



**7.3.2 write方法**

有时我们需要向已经创建的Buffer对象中写入字符串，这时使用write方法

**bf.write(string, [offset],[length],[encoding])**

**一参：必选，指定需要写入的字符串**

**二、三参：可选，指定字符串转换为字节数据后的写入位置。字节的书写位置为从第1+offset个字节开始到offset+length个字节为止。**

**四参：可选，写入字符串时的编码格式**

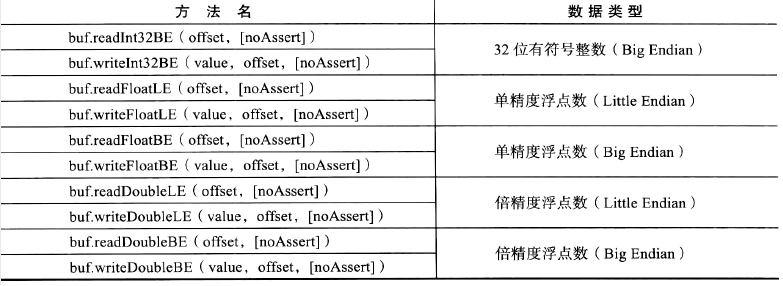
|  |
| --- |
| 1. var bf = new Buffer("思软科技") 2. bf.write("有",6,3) 3. console.log(bf.toString()) //思软有技 |

**7.3.3 StringDecoder方法**

7.4 Buffer对象与数值对象之间的相互转换

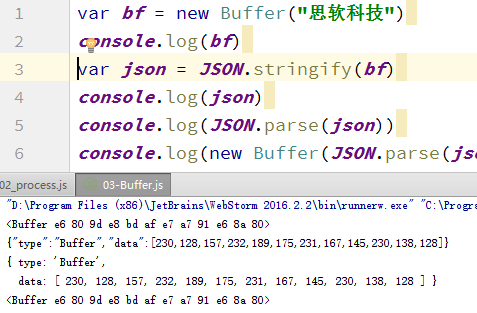
在NodeJS中，为Buffer对象准备了将Buffer对象中的二进制数据读取为JavaScript中的Number类型的数据，或者将Number类型的数据转换为精确类型后向Buffer总写入的方法。见下表：





7.5 Buffer对象与JSON对象之间的相互转换

在NodeJS中,可以使用JSON.stringify方法将Buffer对象中保存的数据转换为一个字符串,也可以使用JSON.parse方法将一个经过转换后的字符串还原为一个数组。



7.6 复制缓存数据

当需要把**Buffer对象中保存的二进制数据复制到另一个Buffer对象中**时，可以使用Buffer对象的copy方法，copy方法使用如下：

**bf.copy(targetBuffer,[targetStart],[sourceStart],[sourceEnd])**

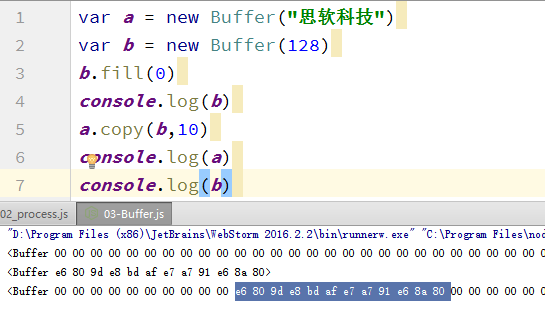
**copy方法接受四个参数。**

**一参：必选，指定复制的目标Buffer对象**

**二参：可选，指定目标Buffer对象中从第几个字节开始写入数据，参数为正整数，默认值为0**

**三参：可选，指定从复制源Buffer对象中获取数据时的起始位置，默认值0**

**四参：可选，指定从复制源Buffer对象中获取数据时的结束为止，默认是Buffer对象的长度**



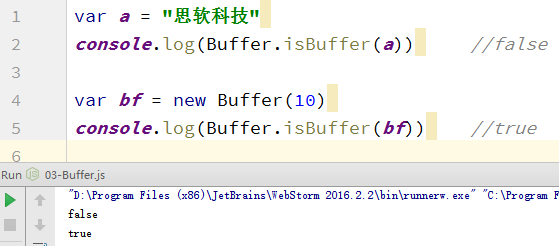
7.7 Buffer类的类方法

**7.7.1 isBuffer方法**

用来判断一个对象是否为一个Buffer对象，使用方法如下：

**Buffer.isBuffer(obj)**

**参数obj，用于指定需要被判断的对象，如果对象是Buffer对象，返回true。**



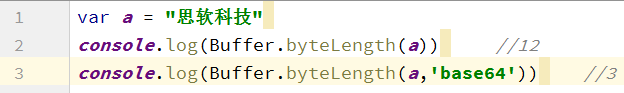
**7.7.2 byteLength方法**

byteLength方法**计算一个指定字符串的字节数**。

**Buffer.byteLength(string,[encoding])**

**一参：必选，指定需要计算字节数的字符串**

**二参：可选，用于指定按什么编码方式来计算字节数，默认utf8**



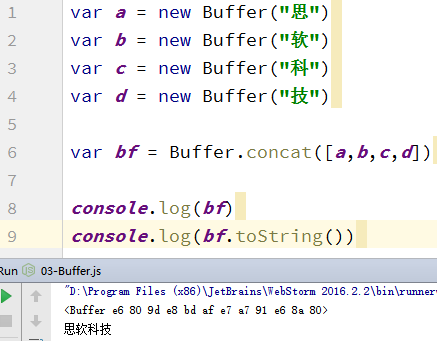
**7.7.3 concat方法**

concat方法用于将几个Buffer对象结合创建为一个新的Buffer对象，方法如下：

**Buffer.concat(list,[totalLength])**

**一参：必选，一个存放了多个Buffer对象的数组**

**二参：可选，知道那个被创建的Buffer对象的总长度**



**7.7.4 isEncoding方法**

isEncoding方法用于**检测一个字符串是否为一个有效的编码格式字符串**,使用方法如下:

**Buffer.isEncoding(encoding)**

**参数，用于指定需要被检测的字符串。如果是有效的编码格式字符串，返回true**

1. **FS文件系统**

文件读取依赖于fs模块，所以必须首先引入fs模块，**var fs = require(‘fs’)**

8.1 读写操作

8.1.1 文件的完整读写

**fs.readFile(file[, options], callback)**

**一参：必选，指定读取文件的完整文件路径及文件名**

**二参：可选，一个对象，指定读取文件时需要使用的选项，可以使用flag属性字段对该文件采取什么操作，默认值是“r”，可指定值如下：**

* **r ：读取文件**
* **r+ ：读取并写入文件**
* **rs ：以同步方式读取文件并通知操作系统忽略本地文件系统缓存。**
* **w ：写入文件，如果文件不存在则创建该文件，如果该文件存在则清空文件内容**
* **wx ：与w类似，但是以排他方式写入文件**
* **w+ ：读取并写入文件。如果文件不存在则创建该文件，如果该文件存在则清空文件内容**
* **wx+：与w+类似，但是以排他方式打开文件**
* **a ：追加写入文件，如果文件不存在则创建该文件**
* **ax ： 最有与a类似，但是以排他方式写入文件**
* **a+ ： 读取并追加写入文件，如果文件不存在则创建该文件**
* **ax+ ： 作业与a+类似，但是以排他方式打开文件**

**在options参数中，可使用encoding指定以何种编码格式读取文件，可指定属性值有utf8,asci,base64.**

**三参：必选，文件读取完毕时执行的回调函数。function(err,data){}。第一个参数err，为读取摁键操作失败时出发的错误对象。第二个参数值data，为读取到的文件内容。**

|  |
| --- |
| 1. var fs = require('fs') 2. fs.readFile('src/test.txt',function (err,data) { 3. if(err){ 4. console.log(err) 5. } 6. console.log(data.toString()) 7. }) |

**fs.readFileSync(file[, options])**

|  |
| --- |
| 1. try{ 2. var data = fs.readFileSync('src/test1.txt') 3. console.log(data.toString()) 4. }catch (ex){ 5. console.log("读取文件时发生错误") 6. } |

**fs.writeFile(file, data[, options], callback)**

**一参：必选，指定需要被写入文件的完整文件路径及文件名**

**二参：必选，指定需要写入的内容，可以是一个字符串或一个Buffer对象**

**三参：可选，一个对象，指定写入文件时需要使用的选项，可以使用的选项有：**

**flag属性：指定对该文件采取何种操作，默认是’w’，其余参照readFile配置**

**mode属性：指定当文件被打开时对该文件的读写权限，默认值为0666（可读写）。该属性值及fs模块中的各方法中的mode参数值的指定方法均如下：使用4个数字组成mode属性或mode参数值，第一个**数字**必须是0**，**第二个**数字用于**规定文件或目录所有者的权限**，**第三个**数字用于**规定文件或目录所有者所属用户组的权限**，**第四个**数字规定**其他人的权限**。可设定的数字如下：

1 ：执行权限

2 ：写权限

3 ：读权限

如果需要读写等符合权限，可以对三个数字加运行，例如2+4=6来设置读写权限。

|  |
| --- |
| 1. fs.writeFile('src/test.txt','这是写入的文字哦\r\n',function (err) { 2. if (err) console.log(err) 3. console.log("写文件成功") 4. }) |

写入缓存区中的数据：

|  |
| --- |
| 1. var bf = new Buffer('思软科技') 2. fs.writeFile('src/test.txt',bf,function (err) { 3. if (err) console.log(err) 4. console.log("写文件成功") 5. }) |

追加数据：

|  |
| --- |
| 1. var options = {flag:'a'} 2. fs.writeFile('src/test.txt','\r\n这是追加的数据哦',options,function (err) { 3. if (err) console.log(err) 4. console.log("写文件成功") 5. }) |

设置encoding,实现复制文件:

|  |
| --- |
| 1. fs.readFile('src/00.jpg','base64',function (err,data) { 2. fs.writeFile('src/02.jpg',data.toString(),'base64',function (err) { 3. if (err) console.log(err) 4. console.log("写文件成功") 5. }) 6. }) |

**fs.writeFileSync(file, data[, options]) 同步写文件**

**fs.appendFile(file, data[, options], callback) 追加数据到文件底部**

|  |
| --- |
| 1. fs.appendFile('src/test.txt',"这里是追加的数据",'utf8',function (err) { 2. if (err) console.log(err) 3. console.log("写文件成功") 4. }) |

**fs.appendFileSync(file, data[, options])**

8.1.2 从指定位置处开始读写文件

**fs.open(path, flags[, mode], callback)**

|  |
| --- |
| 1. //打开文件 2. fs.open('src/test.txt','r',function (err,fd) { 3. //fd 打开文件时返回的文件描述符 一个整数值 4. console.log(fd) 5. }) |

**fs.openSync(path, flags[, mode])**

在打开文件之后，可以在回调函数中使用fs模块中**read**方法从文件的**指定位置处读取文件**，也可以使用fs模块中的**write**方法从**文件的指定处开始写入数据**。

**fs.read(fd, buffer, offset, length, position, callback)**

**read**方法从文件的**指定位置处读取文件**，一直**读到文件底部**，然后将读取到的内容**输出到一个缓存区**中。

一参：fd，必须是open方法使用的回调函数中返回的文件描述符

二参：buffer，一个Buffer对象，指定将文件数据读取到哪个缓存区中

三参：offset，指定向缓存区中写入数据时的开始写入位置（以字节为单位）

四参：length，指定从文件中读取的字节数

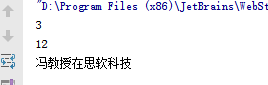
五参：position，指定读取文件时的开始位置（以字节为单位）

六参：callback，function(err,bytesRead,buffer){}

bytesRead，一个整数值，代表实际读取的字节数

buffer，被读取的缓存区对象

|  |
| --- |
| 1. //打开文件 2. fs.open('src/test.txt','r',function (err,fd) { 3. //fd 打开文件时返回的文件描述符 一个整数值 4. console.log(fd) 5. var bf = new Buffer("冯教授在哪儿呢呀") 6. fs.read(fd,bf,12,12,0,function (err,bytesRead,buffer) { 7. console.log(bytesRead) 8. console.log(buffer.toString()) 9. }) 10. }) |

输出结果是：

**fs.readSync(fd, buffer, offset, length, position)**

在**open**打开文件**之后**，可以使用fs模块中**write方法**从一个**缓存区中读取数据**并从**文件的指定处开始写入数据**

**fs.write(fd, buffer, offset, length[, position], callback)**

一参：fd，必须是open方法使用的回调函数中返回的文件描述符

二参：buffer，一个Buffer对象，指定从哪个缓存区中读取数据

三参：offset，指定从缓存区中读取数据时的开始读取位置（以字节为单位）

四参：length，指定从缓存区中读取的字节数

五参：position，指定写入文件时的开始位置（以字节为单位）

六参：callback，function(err,written,buffer){}

written，一个整数值，代表被写入的字节数

buffer，被读取的缓存区对象

|  |
| --- |
| 1. //打开文件 2. fs.open('src/test.txt','w',function (err,fd) { 3. //fd 打开文件时返回的文件描述符 一个整数值 4. console.log(fd) 5. var bf = new Buffer("冯教授在哪儿呢呀") 6. fs.write(fd,bf,0,12,0,function (err,written,buffer) { 7. console.log(written) 8. console.log(buffer.toString()) 9. }) 10. }) |

**fs.writeSync(fd, buffer, offset, length[, position])**

在fs模块中，可以使用**close**方法关闭文件

**fs.close(fd, callback)**

一参：fd，必须是open方法使用的回调函数中返回的文件描述符

二参：callback，回调函数，接受一个参数，参数值为关闭文件操作失败时触发的错误对象

|  |
| --- |
| 1. //打开文件 2. fs.open('src/a.txt','wx',function (err,fd) { 3. //fd 打开文件时返回的文件描述符 一个整数值 4. console.log(fd) 5. var bf = new Buffer("冯教授在哪儿呢呀") 6. fs.write(fd,bf,0,12,0,function (err,written,buffer) { 7. console.log(written) 8. console.log(buffer.toString()) 9. fs.close(fd,function (err) { 10. if(err) console.log(err) 11. console.log("close") 12. }) 13. }) 14. }) |

**fs.closeSync(fd)**

在使用write方法或writeSync方法在文件中写入数据时，操作系统的做法是首先将部分数据读到内存中再把数据写到文件中，**当数据读完时并不代表数据已经写完**，因为有一部分数据可能还留在内存缓存区中。这时如果你调用close方法关闭文件，那么这部分数据会丢失，这时，可以调用**fsync**方法对文件进行同步操作，即**将内存缓存区中的剩余数据全部写入文件**，方法如下：**fs.fsync(fd,[callback])**

一参：fd，必须是open方法使用的回调函数中返回的文件描述符

二参：callback，回调函数，接受一个参数，参数值为操作失败时触发的错误对象

|  |
| --- |
| 1. //打开文件 2. fs.open('src/b.txt','wx',function (err,fd) { 3. //fd 打开文件时返回的文件描述符 一个整数值 4. console.log(fd) 5. var bf = new Buffer("冯教授在哪儿呢呀") 6. fs.write(fd,bf,0,12,0,function (err,written,buffer) { 7. console.log(written) 8. console.log(buffer.toString()) 9. fs.fsync(fd) 10. fs.close(fd,function (err) { 11. if(err) console.log(err) 12. console.log("close") 13. }) 14. }) 15. }) |

**fs.fsyncSync(fd)**

8.2 创建与读取目录

**8.2.1 创建目录**

mkdir方法创建目录，方法如下：**fs.mkdir(path[, mode], callback)**

**一参：必选，指定需要被创建的目录的完整路径及目录名**

**二参：可选，指定该目录的权限，默认值为0777（任何人可读写目录）**

**三参：必选，callback，回调函数中使用一个参数，err**

|  |
| --- |
| 1. fs.mkdir('./src/dir',function (err) { 2. if (err) console.log(err) 3. console.log("创建dir成功") 4. }) |

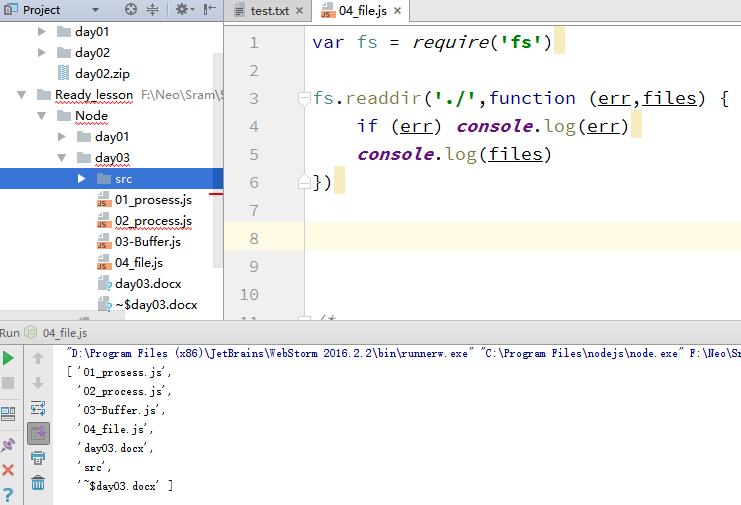
**fs.mkdirSync(path[, mode])**

**8.2.2 读取目录**

**fs.readdir(path[, options], callback) 读取目录**

**一参：必选，指定需要被读取的目录的完整路径及目录名**

**二参，回调函数，function(err,files){}，files为一个数组，保存了读取到的文件中的所有文件名**



**fs.readdirSync(path[, options])**

8.3 查看与修改文件或目录的信息

8.3.1 查看文件或目录信息

在fs模块中，可以使用**stat**方法或**lstat**方法**查看文件或目录的信息**。这两个方法的唯一区别是当查看符号链接文件的信息时，必须使用lstat。使用方法如下：

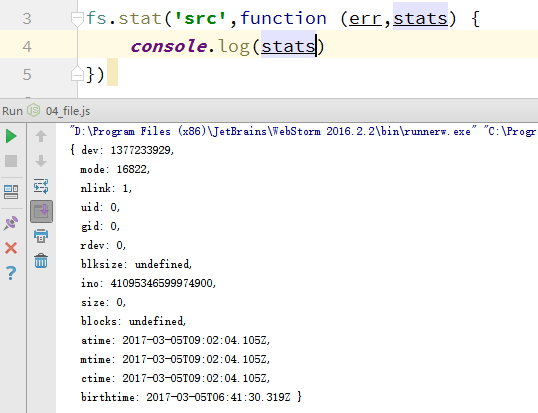
**fs.stat(path, callback)**

**fs.lstat(path, callback)**

**一参：必选，指定需要被查看的文件或目录的完整路径及文件名或目录名**

**二参，回调函数，function(err,stats){}，stats参数值为一个fs.Stats对象，该对象拥有如下的方法：**

* isFile：判断被查看的对象是否为一个**文件**，如果是返回true
* isDirectory：判断被查看的对象是否是一个**目录**，如果是返回true
* ~~isBlockDevice：用于判断被查看的文件是否为一个块设备文件，仅在UNIX系统下有效~~
* dev~~：该属性为文件或目录所在设备ID ，仅在UNIX系统下有效~~
* mode：使用数字形式代表的文件或目录的权限标志
* nlink：文件或目录的硬连接数量
* size：文件尺寸（文件中的字节数）
* atime：文件的访问跟时间
* mtime：文件的修改时间
* ctime：文件的创建时间



**fs.statSync(path)**

**fs.lstatSync(path)**

在使用open方法中打开文件并获取到文件描述符后，可以使用fs模块中的**fstat**方法**查询被打开的文件信息，**方法如下：**fs.fstat(fd,callback)**

8.3.2 检查文件或目录是否存在

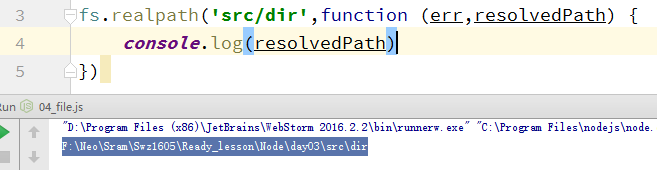
**fs.exists(path,callback)** 检查一个文件或目录是否存在

|  |
| --- |
| 1. fs.exists('src1',function (exists) { 2. if (exists){ 3. console.log("该文件存在") 4. }else{ 5. console.log("不存在") 6. } 7. }) |

**fs.existsSync(path)**

8.3.3 获取文件或目录的绝对路径

**fs.realpath(path[, options], callback)** 获取一个文件或目录的绝对路径



**fs.realpathSync(path[, options])**

8.3.4 修改文件访问时间及修改时间

**fs.utimes(path, atime, mtime, callback)**

**atime：指定修改后的访问时间**

**mtime指定修改后的修改时间**

|  |
| --- |
| 1. fs.utimes('src/test.txt',new Date(),new Date(),function (err) { 2. console.log("修改成功") 3. }) |

**fs.utimesSync(path, atime, mtime)**

在使用open方法打开文件并获取到文件描述符后，可以使用**futimes**修改文件的访问时间或修改时间，方法如下：**fs.futimes(fd, atime, mtime, callback)**

**fs.futimesSync(fd, atime, mtime)**

8.3.5 修改文件或目录的读写权限

**fs.chmod(path, mode, callback)**

|  |
| --- |
| 1. //0600 代表所有者可读写，其他人没有任何权限 2. fs.chmod('./src/test1.txt',0600,function (err) { 3. if (err) {console.log("1-修改权限失败"); return} 4. console.log("1-修改权限成功") 5. }) 6. //0644 代表所有者可读写，其他人可读 7. fs.chmod('src/test2.txt',0644,function (err) { 8. if (err) {console.log("2-修改权限失败"); return} 9. console.log("2-修改权限成功") 10. }) 11. //0755 代表所有者有所有权限，其他人可读和执行 12. fs.chmod('src/test3.txt',0755,function (err) { 13. if (err) {console.log("3-修改权限失败"); return} 14. console.log("3-修改权限成功") 15. }) 16. //0740 代表所有者有所有权限，所有者所在的组只读 17. fs.chmod('src/test4.txt',0740,function (err) { 18. if (err) {console.log("4-修改权限失败"); return} 19. console.log("4-修改权限成功") 20. }) |

**fs.chownSync(path, uid, gid)**

**fs.fchmod(fd, mode, callback)**

**fs.fchmodSync(fd, mode)**

8.4 移动文件或目录

**fs.rename(oldPath, newPath, callback)：**

rename方法可以**移动文件或目录**，也可进行**重命名**操作

|  |
| --- |
| 1. fs.rename('./src/a.txt','./src/dir/a.text',function (err) { 2. if (err){ 3. console.log(err) 4. return 5. } 6. console.log("成功") 7. }) |

**fs.renameSync(oldPath, newPath)**

8.5 创建与删除文件的硬链接

在操作系统中, 一个文件被创建之后就拥有了一个文件名,因此该文件的硬链接数量为1。但是我们可以通过某种特殊的操作为该文件再指定一个文件名,而这种特殊的操作称为对该文件创建硬链接。在对该文件创建一个硬链接之后,虽然表面上看起来拥有了两个不同的文件, 但是在硬盘中这两个文件只不过是同一个文件的多个硬链接, 如果修改一个文件中的内容再打开另一个文件, 就会看见另一个文件中的内容也被修改。

在 fs模块中, 可以使用nnk 方法创建文件的硬链接, 该方法的使用方式如下所示：

**fs.link(srcpath,dstpath,callback)**

srcpath参数用于指定需要被创建硬链接的文件的完整路径及文件名;

dstpath参数用于指定被创建的硬链接的完整路径及文件名,该硬链接与原文件必须位于同一巻中;

可以使用 unlink 方法删除文件的硬链接：

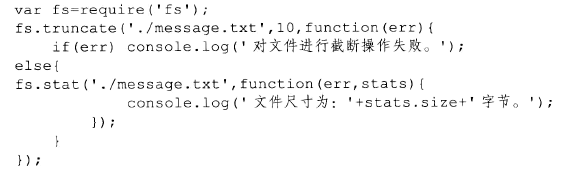
**fs.unlink(path, callback)**

|  |
| --- |
| 1. fs.link('./src/b.txt','./src/dir/a.txt',function (err) { 2. if (err) { 3. console.log(err) 4. return 5. } 6. console.log("硬链接创建成功") 7. }) |

8.6 截断文件

**fs.truncate(path, len, callback)**

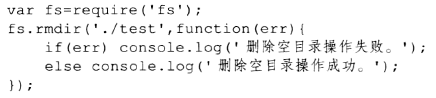
**fs.truncateSync(path, len)**



8.7 删除空目录

**fs.rmdir(path, callback)**

**fs.rmdirSync(path)**



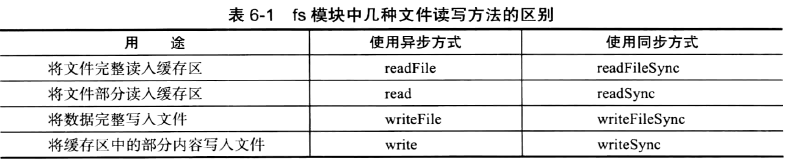
8.8 监视文件或目录

**fs.watch(filename[, options][, listener])**

**fs.watchFile(filename[, options], listener)**

8.9 使用文件流

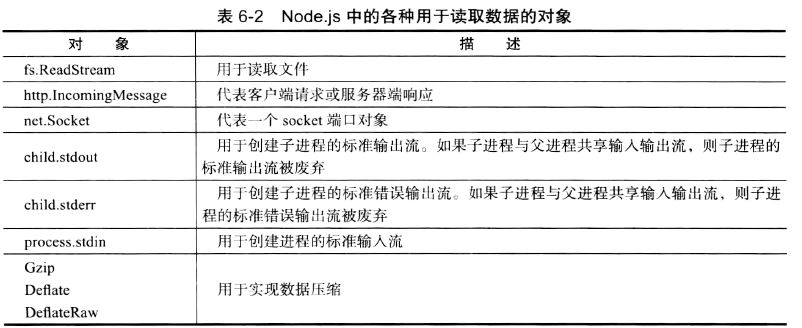
**8.9.1 流的基本概念**

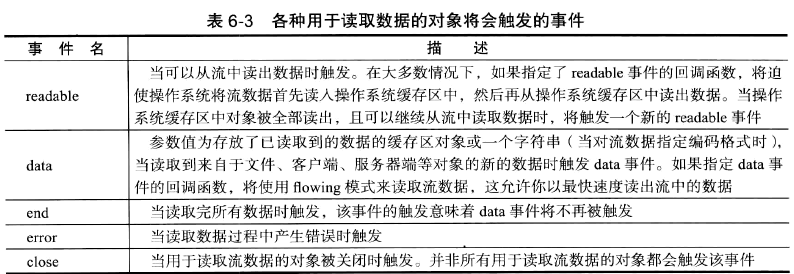


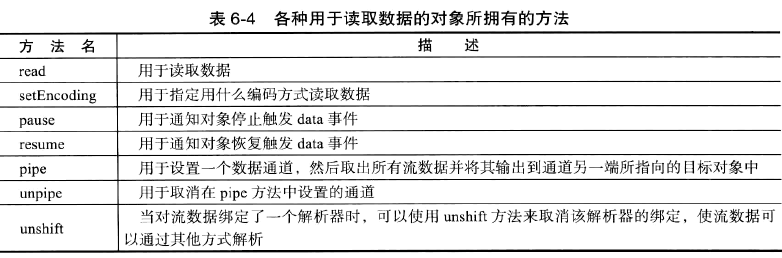
在使用 readFile方法或 readFileSync方法读取文件内容时, Node.js首先将文件内容完整地读入缓存区, 再从该缓存区中读取文件内容。 在使用 writeFile方法或writeFilesync方法写入文件内容时, Node.js首先将该文件内容完整地读入缓存区, 然后一次性将缓存区中内容写入到文件中。也就是说,在使用 readFile方法或 readFileSync方法读取文件内容或者使用writeFile方法或writeFileSync方法写入文件内容时, Node.js将该文内容视为一个整体, 为其分配缓存区并且一次性将文件内容读取到缓存区中, **在这个期间, Node js将不能执行任何其他处理** 。

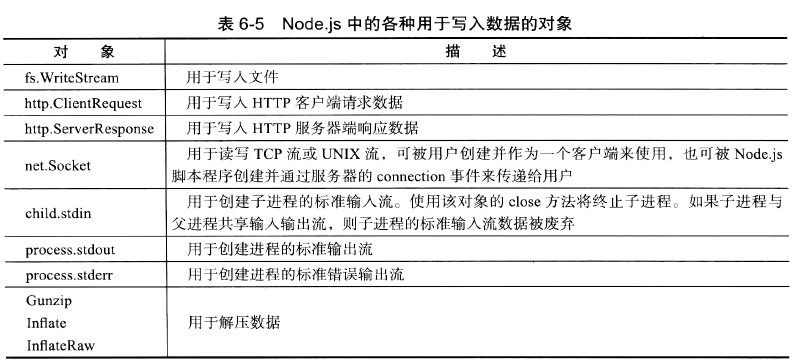
如果使用 read方法或 readSync方法读取文件内容, Node.js将不断地将文件中一小块内容读人缓存区, 最后从该缓存区中读取文件内容, 如果使用 write方法或 writesync方法写入文件内容, Node.js将执行以下过程: ①将需要书写的数据书写到一个内存缓冲区; ②待缓冲区写满后再将该缓冲区中内容写入到文件中 ; ③重复执行过程①与过程②, 直到数据全部写入文件为止。也就是说, 如果使用 read方法或 readsync方法读取文件内容或者使用 write方法或writeSync方法写入文件, 在读写文件过程中允许 NodeJs执行其他处理。

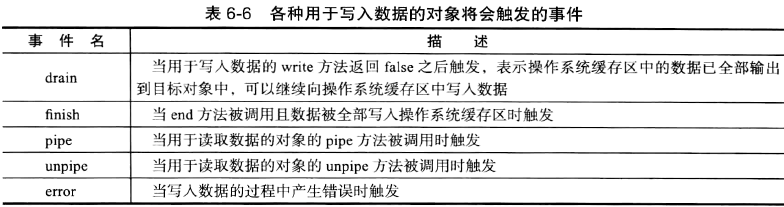
在一个应用程序中, **流**是一组有序的、 有起点和终点的**字节数据的传输手段**。 在应用程序中各种对象之间交換与传输数据的时候. 总是先将该对象中所包含的数据转换为各种形式的流数据(即字节数据),再通过流的传输,到达目的对象后再将流数据转换为该对象中可以使用的数据

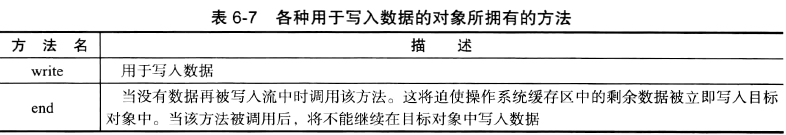












**8.9.2 使用ReadStream对象读取文件**

在 fs模块中, 可以使用 createReadStream方法创建一个将文件内容读取为流数据的ReadStream对象,使用方式如下所示：

**fs.createReadStream(path, [options])**

path：指定需要被读取的文件的完整路径及文件名 。

options：一个对象, 其中可使用的属性及其作用如下所示

flags:用于指定对该文件采取什么操作,默认值为'r',可指定属性值与readFile方法中所使用的 options参数对象中所使用的flags属性的可指定属性值相同

encoding: 用于指定使用什么编码格式来读取该文件, 可指定属性值为 “utf8”、“ascii”与“base64”。默认值为nu11。

autoC1ose:用于指定是否关闭在读取文件时操作系统内部使用的文件描述符。如果属性值设定为 false,则文件不会自动关闭, 即使在读取文件过程中产生了错误,开发者也必须通过使用 close方法或 c1oseSync方法来手动关闭文件。如果属性值设定为true, 当文件读取完毕或读取文件过程中产生错误时文件将自动关闭。 autoclose属性的默认属性值为 true。

start:使用整数值来指定文件的开始读取位置(单位为字节数)。

end: 使用整数值来指定文件的结束读取位置(单位为字节数)。

|  |
| --- |
| 1. var fs = require('fs') 2. var file = fs.createReadStream('src/b.txt') 3. file.on('open',function (fd) { 4. console.log("开始读文件") 5. }) 6. file.on('data',function (data) { 7. console.log("读取到的数据是: ", data.toString()) 8. }) 9. file.on('end',function () { 10. console.log("文件读取完毕") 11. }) 12. file.on('close',function () { 13. console.log("文件已关闭") 14. }) 15. file.on('error',function (err) { 16. console.log(err) 17. }) |

可以使用 ReadStream对象的 pause方法暂停 data事件的触发， resume方法恢复 data 事件的触发

|  |
| --- |
| 1. var fs = require('fs') 2. var file = fs.createReadStream('src/b.txt') 3. file.pause() 4. file.on('data',function (chunk) { 5. console.log(chunk.toString()) 6. }) 7. setTimeout(function () { 8. file.resume() 9. },2000) |

**8.9.3 使用WriteStream对象写入文件**

**fs.createWriteStream(path, [options] )**

**writable.write(chunk, [encoding], [ca11back] )**

**writable.end( [chunk] , [encoding] , [ca11back] )**

WriteStream对象具有一个 bytesWritten属性, 属性值为当前已在文件中写入数据的字节数。

|  |
| --- |
| 1. var fs = require('fs') 2. var ReadStream = fs.createReadStream('src/test.txt') 3. var WriteStream = fs.createWriteStream('src/c.txt') 4. ReadStream.on('data',function (chunk) { 5. WriteStream.write(chunk,function () { 6. console.log(chunk.toString()) 7. }) 8. }) 9. WriteStream.on('open',function () { 10. console.log("open") 11. }) 12. ReadStream.on('end',function () { 13. console.log("读入完毕") 14. WriteStream.end("再见","utf8",function () { 15. console.log(WriteStream.bytesWritten) 16. }) 17. }) |

**8.9.4 pipe管道**

|  |
| --- |
| 1. var fs = require('fs') 2. var ReadStream = fs.createReadStream('src/test.txt') 3. var WriteStream = fs.createWriteStream('src/d.txt') 4. ReadStream.pipe(WriteStream,{end:false}) 5. WriteStream.on('close',function () { 6. console.log("close") 7. }) 8. ReadStream.on('end',function () { 9. console.log("读入完毕") 10. WriteStream.end("再见","utf8",function () { 11. console.log(WriteStream.bytesWritten) 12. }) 13. }) |

1. **Path模块路径处理**

**9.1 path.normalize(path)**

该方法将非标准路径字符串转換为标准路径字符申, 在转换过程中执行以下处理:

1. 解析路径字符串中的“..”字符串与“.”字符串,返回解析后的标准路径。
2. 将多个斜杠字符串转换为一个斜杠字符串,例如将“\\”转换为“\”。
3. 将 Windows操作系统中的反斜杠字符串转换为正斜杠字符串。
4. 如果路径字符串以斜杠字符串结尾, 则在转换后的完整路径字符串末尾保留该斜杠字符串

在 normauze方法中, 使用一个参数, 参数值为需要被转换的路径字符串。该方法返回被转換后的路径字符串.

|  |
| --- |
| 1. var path = require('path') 2. var \_path = path.normalize('..//src//dir/..') 3. console.log(\_path); //../src |

**9.2 path.join([...paths])**

path.join() 方法使用平台特定的分隔符把全部给定的 path 片段连接到一起，并规范化生成的路径.

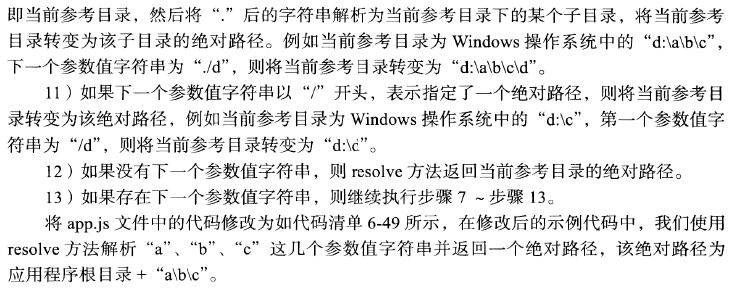
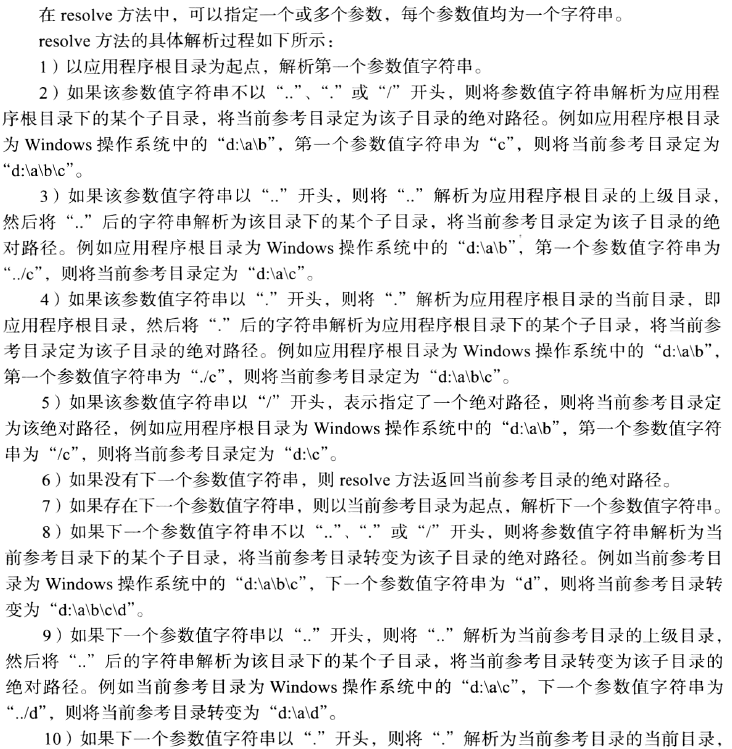
如果任一路径片段不是一个字符串，则抛出 TypeError。

|  |
| --- |
| 1. path.join('/foo', 'bar', 'baz/asdf', 'quux', '..') 2. // 返回: '/foo/bar/baz/asdf' 3. path.join('foo', {}, 'bar') 4. // 抛出 TypeError: path.join 的参数必须为字符串 |

9.3 path.resolve([...paths])

path.resolve() 方法会把一个路径或路径片段的序列解析为一个绝对路径

|  |
| --- |
| 1. console.log(path.resolve('/dist','/src')) 2. console.log(path.resolve('/day03','../dir')) 3. console.log(path.resolve('/foo/bar', '/tmp/file/')) |



9.4 path.relative(from,to)

path.relative() 方法返回从 from到 to 的相对路径。 如果 from 和 to 各自解析到同一路径（调用 path.resolve()），则返回一个长度为零的字符串。

9.5 path.dirname(path)

path.dirname() 方法返回一个 path 的目录名

9.6 path.basename(path[, ext])

path.basename() 方法返回一个 path 的最后一部分

9.7 path.extname(path)

path.extname() 方法返回 path 的扩展名，即从 path 的最后一部分中的最后一个 .（句号）字符到字符串结束。 如果 path 的最后一部分没有 . 或 path 的文件名（见 path.basename()）的第一个字符是 .，则返回一个空字符串。

十、Express框架

Express框架是后台的Node框架，所以和jQuery、zepto、yui、bootstrap都不一个东西。

Express在后台的受欢迎的程度，和jQuery一样，就是企业的事实上的标准。

● 原生Node开发，会发现有很多问题。比如：

■ 呈递静态页面很不方便，需要处理每个HTTP请求，还要考虑304问题

■ 路由处理代码不直观清晰，需要写很多正则表达式和字符串函数

■ 不能集中精力写业务，要考虑很多其他的东西

● EXPRESS的哲学是在你的想法和服务器之间充当薄薄的一层。这并不意味着他不够健壮，或者没有足够的有用特性，而是尽量少干预你，让你充分表达自己的思想，同时提供一些有用的东西。

英语官网：http://expressjs.com/

中文官网：<http://www.expressjs.com.cn/>

整体感知，Express框架。

安装Express框架，就是使用npm的命令。

|  |
| --- |
| 1. npm install **--save** express |

--save参数，表示自动修改package.json文件，自动添加依赖项。

10.1 路由能力：

|  |
| --- |
| 1. var express = require("express"); 2. var app = express(); 3. app.get("/",function(req,res){ 4. res.send("你好"); 5. }); 6. app.get("/haha",function(req,res){ 7. res.send("这是haha页面，哈哈哈哈哈哈"); 8. }); 9. app.get(/^\/student\/([\d]{10})$/,function(req,res){ 10. res.send("学生信息，学号" + req.params[0]); 11. }); 12. app.get("/teacher/:gonghao",function(req,res){ 13. res.send("老师信息，工号" + req.params.gonghao); 14. }); 15. app.listen(3000); |

10.2 静态文件伺服能力：

|  |
| --- |
| 1. app.use(express.static("./public")); |

10.3 模板引擎：

|  |
| --- |
| 1. var express = require("express"); 2. var app = express(); 3. app.set("view engine","ejs"); //设置模板的使用的语言 4. app.get("/",function(req,res){ 5. res.render("haha",{ //render：渲染模板 6. "news" : ["我是小新闻啊","我也是啊","哈哈哈哈"] 7. }); 8. }); 9. app.listen(3000); |

Ejs：去看官网

<%= 变量 %> = ： 解析变量

<% for(){ %> 没有等号： 执行语法

<%- include(“”) %> -： 不执行语法，只是填充页面

文件夹里没有 npm init

已经有package.json npm install 要安装的插件，和文件夹的名字不能一样

自动重启服务器：npm i supervisor -g 插件 使用命令行 supervisor --harmony index

npm i ejs --save 不需要require()

public：放静态文件

views：放模板

models：放数据库

routes：路由文件

index.js：程序主文件

package.json：配置文件

我们学习的是Express4.X，和Express3.X差别非常大。

十一、路由

当用get请求访问一个网址的时候，做什么事情：

|  |
| --- |
| 1. app.get("网址",function(req,res){ 3. }); |

当用post访问一个网址的时候，做什么事情：

|  |
| --- |
| 1. app.post("网址",function(req,res){ 3. }); |

如果想处理这个网址的任何method的请求，那么写all

|  |
| --- |
| 1. app.all("/",function(){ 3. }); |

这里的网址，不分大小写，也就是说，你路由是

|  |
| --- |
| 1. app.get("/AAb",function(req,res){ 2. res.send("你好"); 3. }); |

实际上小写的访问也行。

所有的GET参数，? 后面的都已经被忽略。 锚点#也被忽略

你路由到/a ， 实际/a?id=2&sex=nan 也能被处理。

正则表达式可以被使用。正则表达式中，未知部分用圆括号分组，然后可以用req.params[0]、[1]得到。

req.params类数组对象。

|  |
| --- |
| 1. app.get(**/^\/student\/([\d]{10})$/**,function(req,res){ 2. res.send("学生信息，学号" + **req.params[0]**); 3. }); |

冒号是更推荐的写法。

|  |
| --- |
| 1. app.get("/student/:id",function(req,res){ 2. var id = req.params["id"]; 3. var reg= /^[\d]{6}$/; //正则验证 4. if(reg.test(id)){ 5. res.send(id); 6. }else{ 7. res.send("请检查格式"); 8. } 9. }); |

表单可以自己提交到自己上。

|  |
| --- |
| 1. app.get("/",function(req,res){ 2. res.render("form"); 3. }); 4. app.post("/",function(req,res){ 5. //将数据添加进入数据库 6. res.send("成功"); 7. }); |

适合进行 RESTful路由设计。简单说，就是一个路径，但是http method不同，对这个页面的使用也不同。

/student/345345

get 读取学生信息

add 添加学生信息

delete 删除学生新

十二、中间件

如果我的的get、post回调函数中，没有next参数，那么就匹配上第一个路由，就不会往下匹配了。

如果想往下匹配的话，那么需要写**next()**

|  |
| --- |
| 1. app.get("/",function(req,res,**next**){ 2. console.log("1"); 3. **next();** 4. }); 5. app.get("/",function(req,res){ 6. console.log("2"); 7. }); |

下面两个路由，感觉没有关系：

|  |
| --- |
| 1. app.get("**/:username/:id**",function(req,res){ 2. console.log("1"); 3. res.send("用户信息" + req.params.username); 4. }); 5. app.get("**/admin/login**",function(req,res){ 6. console.log("2"); 7. res.send("管理员登录"); 8. }); |

但是实际上冲突了，因为admin可以当做username用户名, login可以当做id。

解决方法1：交换位置。 也就是说，express中所有的路由（中间件）的顺序至关重要。

匹配上第一个，就不会往下匹配了。 具体的往上写，抽象的往下写。

|  |
| --- |
| 1. app.get("/admin/login",function(req,res){ 2. console.log("2"); 3. res.send("管理员登录"); 4. }); 5. app.get("/:username/:id",function(req,res){ 6. console.log("1"); 7. res.send("用户信息" + req.params.username); 8. }); |

解决方法2：

|  |
| --- |
| 1. app.get("/:username/:id",function(req,res,**next**){ 2. var username = req.params.username; 3. //检索数据库，如果username不存在，那么next() 4. if(检索数据库){ 5. console.log("1"); 6. res.send("用户信息"); 7. }else{ 8. **next();** 9. } 10. }); 11. app.get("/admin/login",function(req,res){ 12. console.log("2"); 13. res.send("管理员登录"); 14. }); |

路由get、post这些东西，就是中间件，中间件讲究顺序，匹配上第一个之后，就不会往后匹配了。next函数才能够继续往后匹配。

app.use()也是一个中间件。与get、post不同的是，他的网址不是精确匹配的。而是能够有小文件夹拓展的。

比如网址： http://127.0.0.1:3000/admin/aa/bb/cc/dd

|  |
| --- |
| 1. app.use("/admin",function(req,res){ 2. res.write(req.originalUrl + "\n"); // /admin/aa/bb/cc/dd 3. res.write(req.baseUrl + "\n"); // /admin 4. res.write(req.path + "\n"); // /aa/bb/cc/dd 5. res.end("你好"); 6. }); |

如果写一个/

|  |
| --- |
| 1. //当你不写路径的时候，实际上就相当于"/"，就是所有网址 2. app.use(function(req,res,next){ 3. console.log(new Date()); 4. next(); 5. }); |

app.use()就给了我们增加一些特定功能的便利场所。

实际上app.use()的东西，基本上都从第三方能得到。

● 大多数情况下，渲染内容用res.render()，将会根据views中的模板文件进行渲染。如果不想使用views文件夹，想自己设置文件夹名字，那么app.set("views","aaaa");

● 如果想写一个快速测试页，当然可以使用res.send()。这个函数将根据内容，自动帮我们设置了Content-Type头部和200状态码。send()只能用一次，和end一样。和end不一样在哪里？能够自动设置MIME类型。

● 如果想使用不同的状态码，可以：

res.status(404).send('Sorry, we cannot find that!');

● 如果想使用不同的Content-Type，可以：

res.set('Content-Type', 'text/html');

十三、GET请求和POST请求的参数

● GET请求的参数在URL中，在原生Node中，需要使用url模块来识别参数字符串。在Express中，不需要使用url模块了。可以直接使用**req.query**对象。gg

● POST请求在express中不能直接获得，必须使用body-parser模块。使用后，将可以用req.body得到参数。但是如果表单中含有文件上传，那么还是需要使用formidable模块。

Node中全是回调函数，所以我们自己封装的函数，里面如果有异步的方法，比如I/O，那么就要用回调函数的方法封装。

错误：

|  |
| --- |
| 1. res.reder("index",{ 2. "name" : student.getDetailById(234234).name 3. }); |

正确:

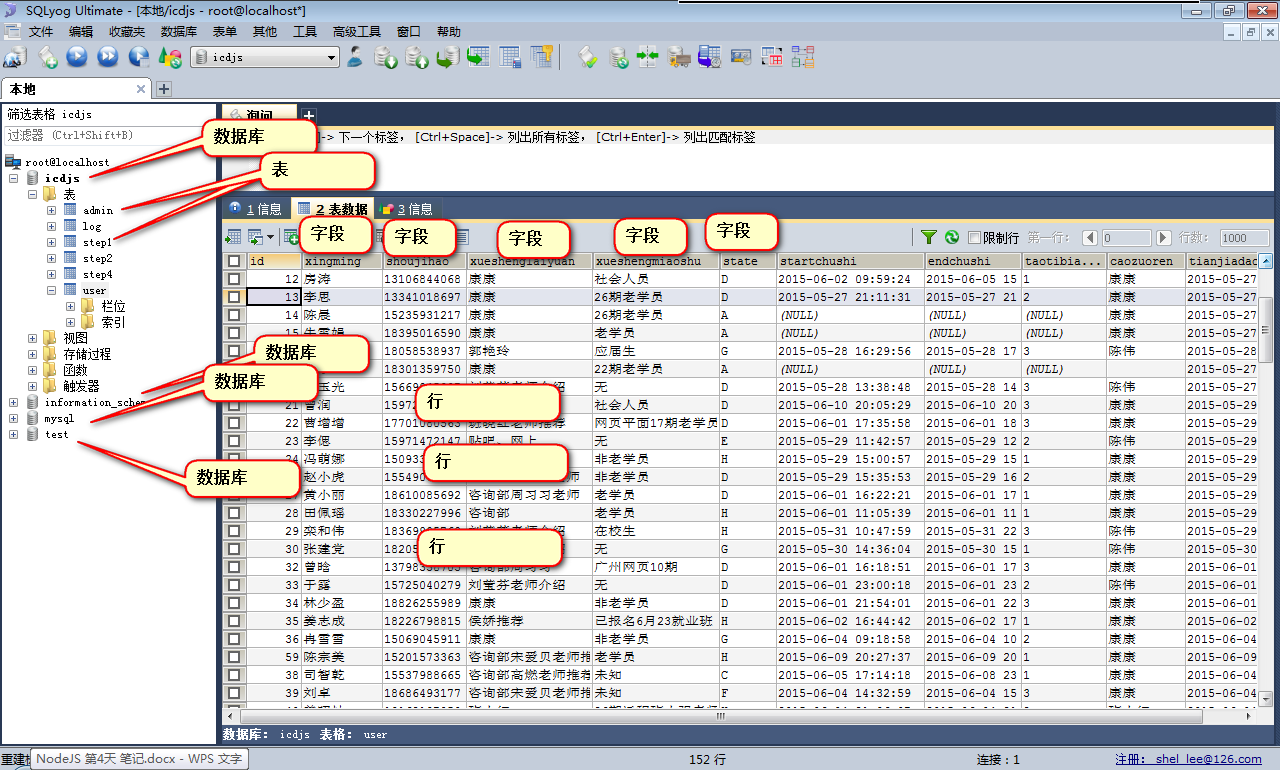
|  |
| --- |
| 1. student.getDetailByXueHao(234234,function(**detail**){ 2. res.render("index",{ 3. "name" : **detail**.name 4. }) 5. }); |

# 十四、传统数据库技术回顾

数据库就是存储数据的，那么存储数据就用txt就行了啊，为什么要有数据库？

理由之1： 数据库有行、列的概念，数据有关系，数据不是散的。

老牌数据库，比如MySQL、SQL Server、Oracle、Access。这些数据库，我们管他们叫做结构型数据库。为什么？因为每个表中，都有明确的字段，每行记录，都有这些字段。不能有的行有，有的行没有。



理由二：数据库能够提供非常方便的接口，让增删改查操作变得简单

我们的老牌数据库，都无一例外的使用SQL语言，管理数据库。

SQL就是structure query language。

比如，查询所有女生： SELECT \* FROM step1 WHERE xingbie = '女';

再比如，查询所有女生，并且年龄20~24之间，且在北京：

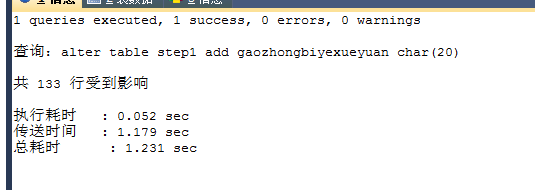
|  |
| --- |
| 1. SELECT \* FROM step1 WHERE xingbie = '女' AND nianling < 24 AND nianling >= 20 AND xianzaisuozaidi = '北京'; |

理由之三：数据库不能自己玩儿，要给向PHP、.net、jsp等语言提供接口。

用php这些语言，能够向数据库之中增删改查。

老牌数据库，都是结构型数据库，现在出了什么问题？

比如，我们现在想往一个已经有1000条数据的数据库中增加一个字段“高中信息”。



之前已经存在的数据，实际上不需要增加这个字段。因为这些用户已经填写完毕表单了，不需要再手机高中信息了。我们的意图就是在今后注册的用户，需要填写高中信息。但是，我们刚才说了，所谓的字段，是表的一个结构。所有的行都必须拥有，不能有的行有这个字段，有的行没有这个字段。

可想而知，大数据时代，数据库中有100万条数据都算少的。我们如果要动字段，时间太长。

**所以，字段这个东西，太不灵活。**

**数据不灵活。一个字段，需要是同样类型的数据。**不能一行记录是文本，一行记录是数字。

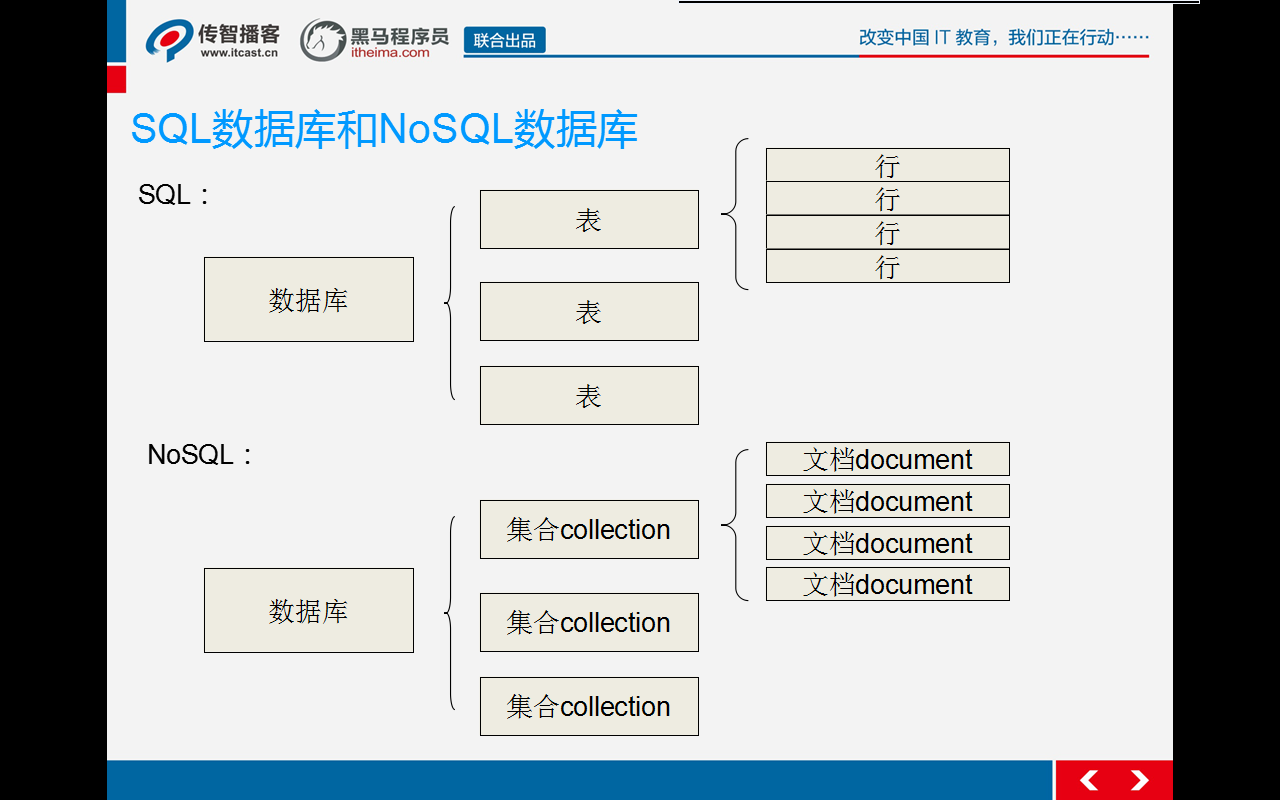
非结构型数据库NoSQL应运而生。

NoSQL是个怪胎，无法挑战老牌数据库，但是在大数据时代有自己的意义。

# 十五、NoSQL

非结构型数据库。**没有行、列的概念。用JSON来存储数据。**

**集合就相当于“表”，文档就相当于“行”。**



文档就是JSON，上下文语境中，也是JavaScript范畴，所以我们的数据库也是JS范畴的东西，JS全栈。



因此，我们总结NoSQL数据库在以下的这几种情况下比较适用：1、数据模型比较简单；2、需要灵活性更强的IT系统；3、对数据库性能要求较高；4、不需要高度的数据一致性；5、对于给定key，比较容易映射复杂值的环境。

我们看，有些系统，特别需要筛选。比如，筛选出所有女生大于20岁的。那么SQL型数据库，非常擅长！因为它有行、列的概念。

但是，有些系统，真的不需要进行那么多的筛选，比如站内信。站内信只需要存储就好了。不需要筛选。那么NoSQL的。

NoSQL不是银弹，没有资格挑战老牌数据库，还是特定情况下，是适合的。

# 十六、MongoDB安装

官网：<https://www.mongodb.com/>

手册：<https://docs.mongodb.org/manual/>

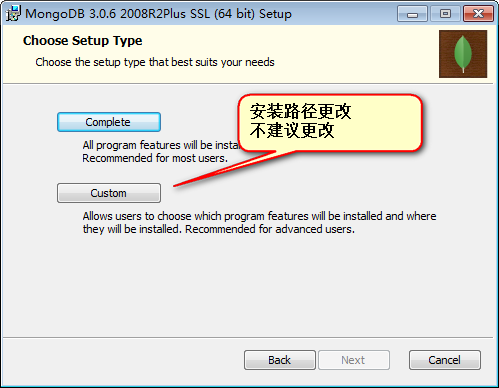
1. 命令行操作mongodb 增删改查
2. 用express来操作mongodb（mongoose插件） 增删改查
3. Mongoose：

在expressjs中链接数据库

mongoose.connect 连接到1个数据库

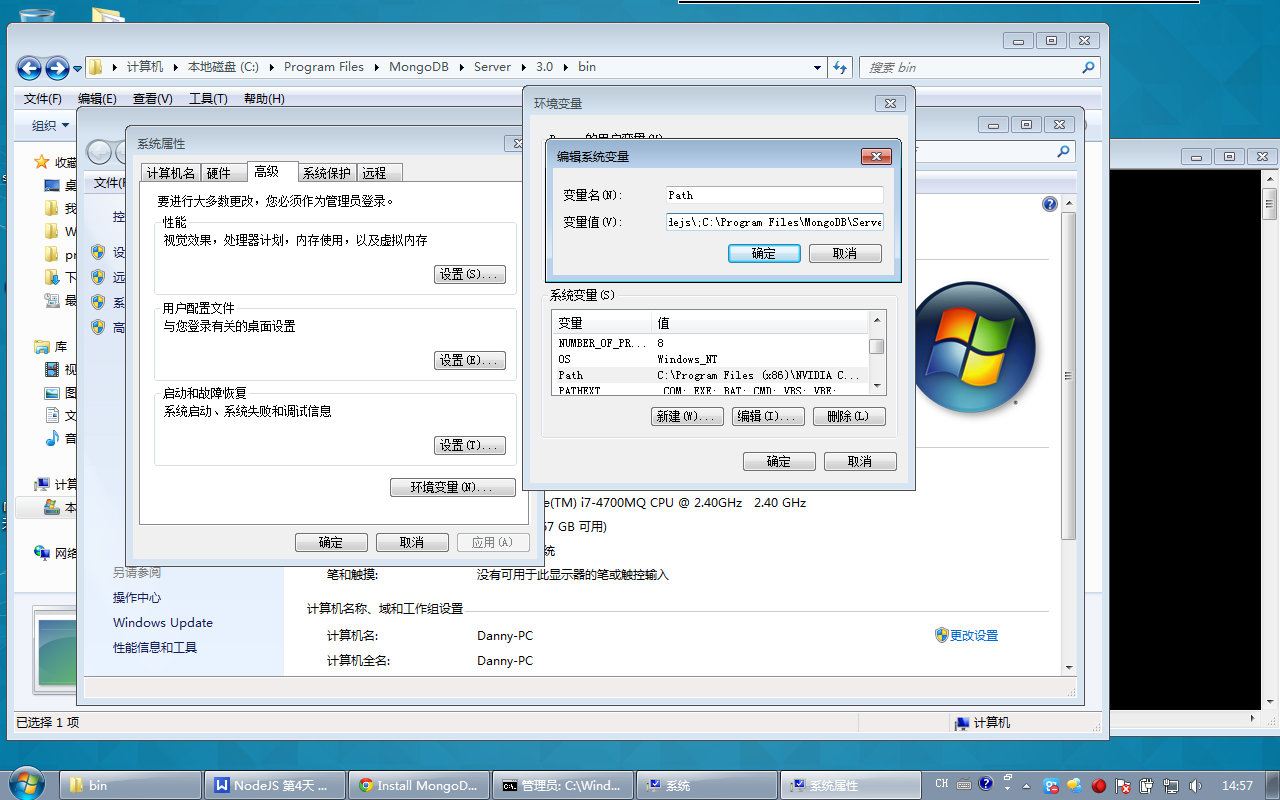
mongoose.**createConnection 创建多个数据库的事例**

win7系统需要安装补丁，KB2731284。



此时，我们看一下装好的文件夹：

C:\Program Files\MongoDB\Server\3.0\bin 加入到系统的path环境变量中



那么我们就能在系统的任何盘符，使用mongo命令了：

mongo 使用数据库

mongod 开机

mongoimport 导入数据

开机命令：



--dbpath就是选择数据库文档所在的文件夹。

也就是说，mon goDB中，真的有物理文件，对应一个个数据库。U盘可以拷走。

一定要保持，开机这个CMD不能动了，不能关，不能ctrl+c。 一旦这个cmd有问题了，数据库就自动关闭了。

所以，**应该再开一个cmd**。输入



那么，运行环境就是mongo语法了。

列出所有数据库：

|  |
| --- |
| 1. show dbs |

使用某个数据库

|  |
| --- |
| 1. use 数据库名字 |

如果想新建数据库，也是use。use一个不存在的，就是新建。

查看当前所在数据库

|  |
| --- |
| 1. db |

插入数据：



student就是所谓的集合。集合中存储着很多json。

student是第一次使用，集合将自动创建。

# 十七、数据库使用

要管理数据库，必须先开机，开机使用**mongod --dbpath c:\mongo**

管理数据库：mongo (一定要在新的cmd中输入)



清屏：

|  |
| --- |
| 1. cls |

查看所有数据库列表

|  |
| --- |
| 1. show dbs |

使用数据库、创建数据库

|  |
| --- |
| 1. use sram |

如果真的想把这个数据库创建成功，那么必须插入一个数据。

数据库中不能直接插入数据，只能往集合(collections)中插入数据。不需要创建集合，只需要写点语法：

|  |
| --- |
| 1. db.student.insert({“name”:”xiaoming”}); |

db.student 系统发现student是一个陌生的集合名字，所以就自动创建了集合。

删除数据库，删除当前所在的数据库

|  |
| --- |
| 1. db.dropDatabase(); |

## 17.1 插入数据

插入数据，随着数据的插入，数据库创建成功了，集合也创建成功了。

|  |
| --- |
| 1. **db.student.insert({"name":"xiaoming"});** |

我们不可能一条一条的insert。所以，我们希望用sublime在外部写好数据库的形式，然后导入数据库：

|  |
| --- |
| 1. mongoimport **--db test** **--collection restaurants** **--drop** **--file primer-dataset.json** |

**-db test 想往哪个数据库里面导入**

**--collection restaurants 想往哪个集合中导入**

**--drop 把集合清空**

**--file primer-dataset.json 哪个文件**

**这样，我们就能用sublime创建一个json文件，然后用mongoimport命令导入，这样学习数据库非常方便。**

## 17.2 查找数据

查找数据，用find。find中没有参数，那么将列出这个集合的所有文档：

|  |
| --- |
| 1. db.restaurants.find() |

精确匹配：

|  |
| --- |
| 1. db.student.find({"score**.**shuxue":70}); |

多个条件：

|  |
| --- |
| 1. db.student.find({"score.shuxue":70 , "age":12}) |

大于条件：

|  |
| --- |
| 1. db.student.find({"score.yuwen":**{$gt:50}**}); |

或者。寻找所有年龄是9岁，或者11岁的学生

|  |
| --- |
| 1. db.student.find({**$or**:[{"age":9},{"age":11}]}); |

查找完毕之后，打点调用sort，表示升降排序。

|  |
| --- |
| 1. db.restaurants.find().**sort(** { "borough": 1, "address.zipcode": 1 } **)** |

## 17.3 修改数据

修改里面还有查询条件。你要修改谁，要告诉mongo。

查找名字叫做小明的，把年龄更改为16岁：

|  |
| --- |
| 1. db.student.update({"name":"小明"},{$set:{"age":16}}); |

查找数学成绩是70，把年龄更改为33岁：

|  |
| --- |
| 1. db.student.update({"score.shuxue":70},{$set:{"age":33}}); |

更改所有匹配项目："

By default, the update() method updates a single document. To update multiple documents, use the multi option in the update() method.

|  |
| --- |
| 1. db.student.update({"sex":"男"},{$set:{"age":33}},{multi: true}); |

完整替换，不出现$set关键字了：

|  |
| --- |
| 1. db.student.update({"name":"小明"},{"name":"大明","age":16}); |

## 17.4 删除数据

|  |
| --- |
| 1. db.restaurants.remove( { "borough": "Manhattan" } ) |

By default, the remove() method removes all documents that match the remove condition. Use the justOne option to limit the remove operation to only one of the matching documents.

|  |
| --- |
| 1. db.restaurants.remove( { "borough": "Queens" }, **{ justOne: true }** ) |

# 十八、Mongoose

需要引包：

|  |
| --- |
| 1. npm install mongodb |

mongoose是一个将JavaScript对象与数据库产生关系的一个框架，object related model。操作对象，就是操作数据库了；对象产生了，同时也持久化了。

这个思路是Java三大框架SSH中Hibernate框架的思路。彻底改变了人们使用数据库的方式。

<http://mongoosejs.com/>

mongoose就是:



官网上的hello world：

|  |
| --- |
| 1. //引包，**并不需要引用mongodb这个包** 2. var mongoose = require('mongoose'); 3. //链接数据库,**haha是数据库名字** 4. mongoose.connect('mongodb://localhost**/haha**'); 5. //创建了一个模型。猫的模型。所有的猫，都有名字，是字符串。“类”。 6. var Cat = mongoose.model('Cat', { name: String }); 7. //实例化一只猫 8. var kitty = new Cat({ name: 'Zildjian' }); 9. //调用这只猫的save方法，保存这只猫 10. kitty.save(function (err) { 11. console.log('喵喵喵'); 12. }); 13. var tom = new Cat({"name":"汤姆"}); 14. tom.save(function(){ 15. console.log('喵喵喵'); 16. }); |

上面的代码中，没有一个语句是明显的操作数据库，感觉都在创建类、实例化类、调用类的方法。都在操作对象，但是数据库同步被持久了。

mongoose的哲学，就是让你用操作对象的方式操作数据库。

创建一个模型

|  |
| --- |
| 1. mongoose.model("Cat",{"name" : String , "age" : Integer}); |

就可以被实例化

|  |
| --- |
| 1. var kitty = new Cat({ name: 'Zildjian' }); |

然后这个实例就可以被save。

中文博客： <https://cnodejs.org/topic/51ff720b44e76d216afe34d9>

mongoose首先要想明白一件事儿，所有的操作都不是对数据库进行的。所有的操作都是对类、实例进行的。但是数据库的持久化自动完成了。

## 18.1 数据库连接

公式：

|  |
| --- |
| 1. var mongoose = require('mongoose'); 2. //创建数据库连接 3. var db = mongoose.**createConnection**('mongodb://127.0.0.1:27017/haha'); 4. //监听open事件 5. db.once('open', function (callback) { 6. console.log("数据库成功连接"); 7. }); |

## 18.2 定义模型

**创造schema → 定义一些schema的静态方法 → 创造模型**

创造schema用什么语句？ new mongoose.schema({});

创造模型用什么语句？ db.model(“Student”,schema名字);

|  |
| --- |
| 1. //创建了一个schema结构。 2. var studentSchema = new mongoose.Schema({ 3. name : {type : String}, 4. age : {type : Number}, 5. sex : {type : String} 6. }); 7. //创建静态方法 8. studentSchema.statics.zhaoren = function(name, callback) { 9. this.model('Student').find({name: name}, callback); //this.model('Student')指的是当前这个类 10. }; 11. //创建修改的静态方法 12. studentSchema.statics.xiugai = function(conditions,update,options,callback){ 13. this.model("Student").update(conditions, update, options, callback); 14. } 15. //创建了一个模型，就是学生模型，就是学生类。 16. //类是基于schema创建的。 17. var studentModel = db.model('Student', studentSchema); |

解释一下，什么是静态方法，什么是类方法：

|  |
| --- |
| 1. //类 2. function Student(){ 4. } 5. //实例化一个学生 6. var xiaoming = new Student(); 7. //实例方法，因为这个sleep方法的执行者是类的实例 8. xiaoming.sleep(); 9. //静态方法（类方法），这个方法的执行者是这个类，不是这个类的实例。 10. Student.findAllBuJiGe(); |

前台界面：不操作数据库，只操作类！

# 十九、Cookie和Session

## 19.1 Cookie

● HTTP是无状态协议。简单地说，当你浏览了一个页面，然后转到同一个网站的另一个页面，服务器无法认识到，这是同一个浏览器在访问同一个网站。每一次的访问，都是没有任何关系的。

那么世界就乱套了，比如我上一次访问，登陆了，下一次访问，又让我登陆，不存在登陆这事儿了。

● Cookie是一个简单到爆的想法：当访问一个页面的时候，服务器在下行HTTP报文中，命令浏览器存储一个字符串；浏览器再访问同一个域的时候，将把这个字符串携带到上行HTTP请求中。

第一次访问一个服务器，不可能携带cookie。 必须是服务器得到这次请求，在下行响应报头中，携带cookie信息，此后每一次浏览器往这个服务器发出的请求，都会携带这个cookie。

特点

● cookie是不加密的，用户可以自由看到；

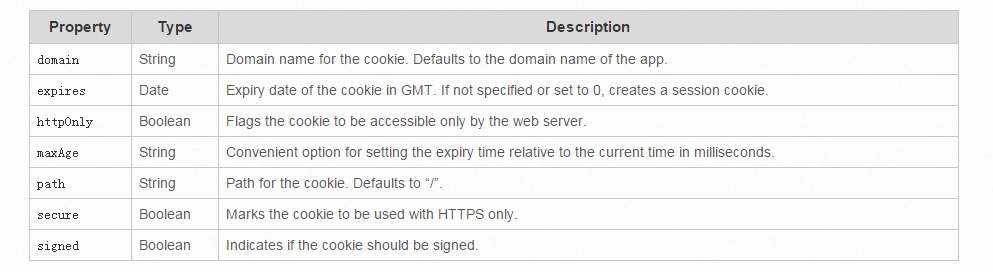
● 用户可以删除cookie，或者禁用它

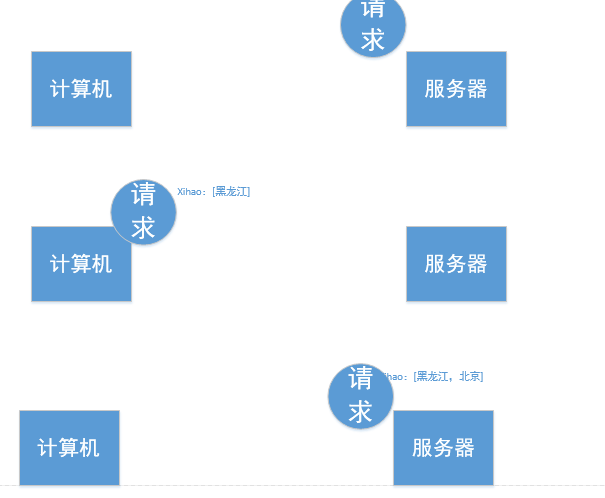
● cookie可以被篡改

● cookie可以用于攻击

● cookie存储量很小。未来实际上要被localStorage替代，但是后者IE9兼容。

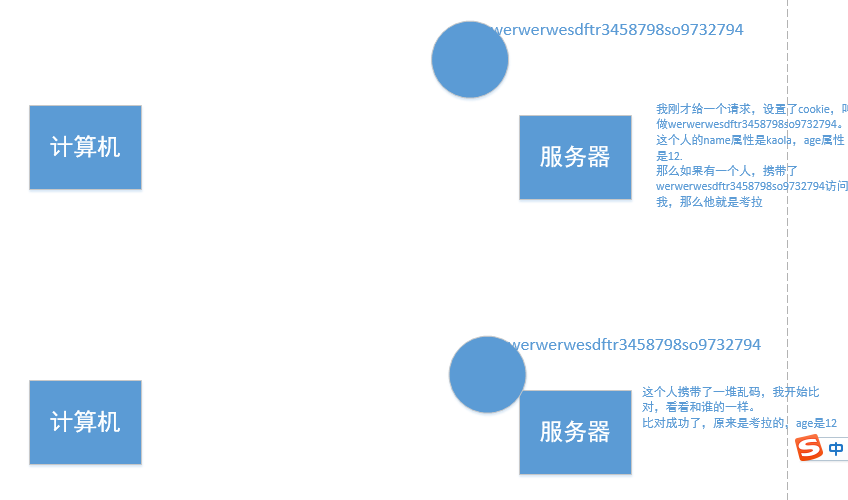
express中的cookie，你肯定能想到。 res负责设置cookie， req负责识别cookie。





## 19.2 Session

会话。 Session不是一个天生就有的技术，而是依赖cookie。



session依赖cookie，当一个浏览器禁用cookie的时候，登陆效果消失； 或者用户清除了cookie，登陆也消失。

session比cookie不一样在哪里呢？ session下发的是乱码，并且服务器自己缓存一些东西，下次浏览器的请求带着乱码上来，此时与缓存进行比较，看看是谁。

所以，一个乱码，可以对应无限大的数据。

任何语言中，session的使用，是“机理透明”的。他是帮你设置cookie的，但是足够方便，让你感觉不到这事儿和cookie有关。

## 19.3 session

|  |
| --- |
| 1. var session = require("express-session"); 2. app.use(session({ 3. secret: 'keyboard cat', 4. resave: false, 5. saveUninitialized: true 6. })) 7. app.get("/",function(req,res){ 8. if(**req.session.login == "1"**){ 9. res.send("欢迎" + **req.session.username**); 10. }else{ 11. res.send("没有成功登陆"); 12. } 13. }); 14. app.get("/login",function(req,res){ 15. **req.session.login = "1"**; //设置这个session 16. req.session.username = "考拉"; 17. res.send("你已经成功登陆"); 18. }); |

加密使用的是MD5加密。

1 C4CA4238A0B923820DCC509A6F75849B

2 C81E728D9D4C2F636F067F89CC14862C

我爱北京天安没和陈伟：

97AFBA2EE71DE6A78B61149A391A378F

我爱北京天安没与陈伟：

CCB721C1E26067BBF4BB49DFDBFA0C9E

我爱北京天安没和陈猥：

C3D110D2441EF9323D3F1E6082E708F8

不管你加密多大的东西，哪怕10M文字，都会加密为32位的字符串，就是密码。并且神奇的，数学上能够保证，哪怕你更改1个文字，都能大变。所以MD5也能用于比对版本。

MD5是数学上，不能破解的。 不能反向破解。

也就是说，C4CA4238A0B923820DCC509A6F75849B 没有一个函数，能够翻译成为1的。

但是，有的人做数据库，就是把1~999999所有数字都用MD5加密了，然后进行了列表，所以有破解的可能。

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |