node.js

简介

简单来说， Node.js 就是运行在服务端（后端）的 JavaScript。

JavaScript语言的设计初衷是为了操作客户端（前端）页面的DOM元素，进行简单的页面交互，对于语言本身来说，并不存在前后之分，如果操作的不是DOM对象，而是数据库对象，是不是就可以变成后端语言了？

服务器端程序主要的工作就是------响应用户请求，提供相应的服务，包括：

操作文件（上传、下载）、数据的存取（操作数据库）、平台的稳定性、优化性能 等。

后端的主要开发程序有：PHP、JAVA 、ASP.net、python、ruby等等。这些程序的运行，都需要在服务器上部署不同的语言环境。

那么，问题来了！

JavaScript的运行环境是浏览器，后端又没有浏览器，怎么运行 JavaScript？？**2009年，大神来了！！！！**

2009年2月，Ryan Dahl在博客上宣布准备基于V8引擎（谷歌浏览器里面的js引擎）创建一个轻量级的Web服务器并提供一套库。 2009年5月，Ryan Dahl在GitHub上发布了最初版本的部分Node.js包，随后几个月里，有人开始使用Node.js开发应用。 2009年11月和2010年4月，两届JSConf大会都安排了Node.js的讲座。 2010年年底，Node.js获得云计算服务商Joyent资助，创始人Ryan Dahl加入Joyent全职负责Node.js的发展。 2011年7月，Node.js在微软的支持下发布Windows版本。

- 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。

nodeJS官网(英文) nodeJS中文网

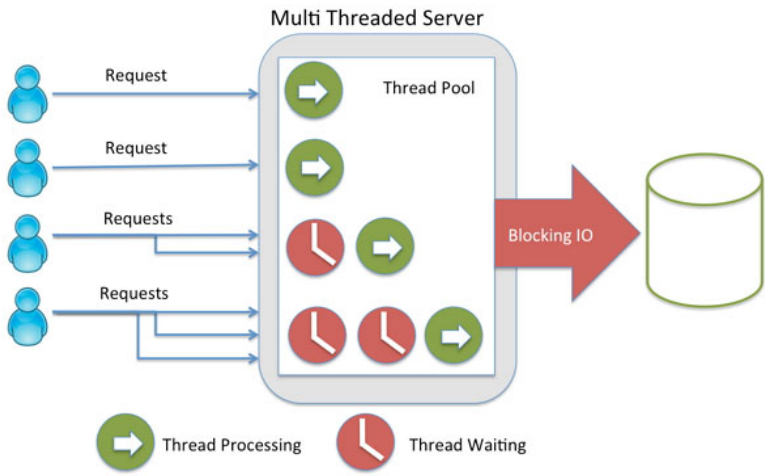
Node.js优、缺点

优点

1. 采用事件驱动、异步编程，为网络服务而设计。而且JavaScript也简单易学，很多前端设计人员可以很快上手做后端设计。
2. Node.js非阻塞模式的IO处理给Node.js带来在相对低系统资源耗用下的高性能与出众的负载能力，非常适合用作依赖其它IO资源的中间层服务。
3. Node.js轻量高效，可以认为是数据密集型分布式部署环境下的实时应用系统的完美解决方案。Node非常适合前端有大量的异步请求，需要服务后端有极高的响应速度，但所需的服务器端逻辑和处理不一定很多，单页面、多Ajax请求应用——如mail，聊天。不适合CPU使用率较重、IO使用率较轻的应用——如视频编码。

多线程服务器

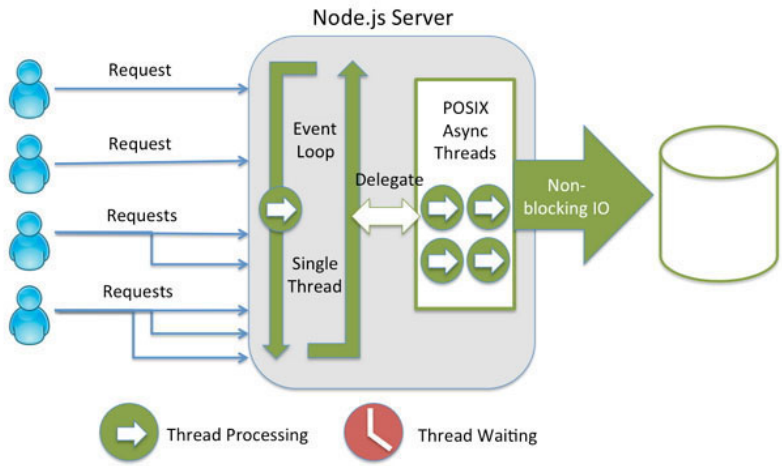
大部分后端语言，都采用的是多线程的方式（多线程服务器 Multi Threaded Server），同一时刻，可以响应多个请求，并开启多个线程，同时处理，但是，对于多线程的管理需要复杂的线程调度，容易造成线程堵塞，如图：



Request：请求
Thread Processing：线程进度
Thread Waiting：线程等待
Thread Pool：线程池
Blocking IO：堵塞IO
IO：input（输入） output（输出）

node服务器

Node Server使用的是js语言，js的运行机制是非堵塞单线程，同一时刻，js将响应的多个请求，添加至**事件队列（task queue）**中，并通过**事件循环（event loop）**机制来管理队列的执行，将事件按照先后顺序压入**执行栈（stack）**执行任务，如图：



Request：请求
Thread Processing：线程进度
Thread Waiting：线程等待
Delegate：委派
POSIX Async Threads：系统异步线程
Event Loop：事件循环
Non-Blocking IO：非堵塞IO
IO：input（输入） output（输出）

阻塞式 I/O

线程在执行中如果遇到磁盘读写或网络通信（统称为 I/O 操作），通常要耗费较长的时间，这时操作系统会剥夺这个线程的 CPU 控制权，使其暂停执行，同时将资源让给其他的工作线程，这种线程调度方式称为 阻塞。当 I/O 操作完毕时，操作系统 将这个线程的阻塞状态解除，恢复其对CPU的控制权，令其继续执行。这种 I/O 模式就是通常的阻塞式 I/O （Blocking I/O）或称为同步式 I/O（Synchronous I/O）

非阻塞式I/O

是针对 所有 I/O 操作不采用阻塞的策略。当线程遇到 I/O 操作时，不会以阻塞的方式等待 I/O 操作 的完成或数据的返回，而只是将 I/O 请求发送给操作系统，继续执行下一条语句。当操作 系统完成 I/O 操作时，以事件的形式通知执行 I/O 操作的线程，线程会在特定时候处理这个事件。为了处理异步 I/O，**线程必须有事件循环**，不断地检查有没有未处理的事件，依次予以处理。这种 I/O 模式就是非阻塞式I/O或称为异步式 I/O （ Asynchronous I/O ）

缺点

1、可靠性低 2、单进程，单线程，只支持单核CPU，不能充分的利用多核CPU服务器。一旦这个进程崩掉，那么整个web服务就崩掉了。

安装

Node.js安装包及源码下载地址为：https://nodejs.org/en/download/。

直接下载对应操作系统的安装文件，安装步骤（略）

配置全局PATH环境

REPL运行环境

为了使开发者方便测试javascript代码，提供了一个名为REPL(Read-Eval-Print-Loop，读取-解释-打印-循环)的可交互式运行环境。在**CMD窗口**中，输入“node”命令并按下回车键，即可进入REPL运行环境，基本命令如下：

命令	说明
.break	当你在书写一个多行函数中途想要放弃或重写此函数时，返回到命令提示符的起点处： > Ctrl+c相当于.break;　Ctrl+c两次会退出REPL环境
.clear	用于清除REPL运行环境的上下文对象中保存的所有变量和函数
.exit	退出REPL运行环境
.help	显示repl环境中所有基础命令
.save	把你输入的所有表达式保存到指定的文件中 例： foo="aab"; .save demo.js
.load	将把指定文件中所有的表达式一次加载到repl环境中

node执行js代码

node可以脱离浏览器执行js代码：

```
node xxx.js
```

如果我们编写一个服务器端的server.js文件，每次修改这个文件，都要重新执行一下node server.js，操作非常麻烦，可以借助一个工具，来自动完成这个工作。

使用 **supervisor** 可以解决开发中的调试问题，安装supervisor：

```
npm install -g supervisor
```

接下来，使用 supervisor 命令启动 server.js：

```
supervisor app.js
-------------------------------------------
运行后，显示如下内容：
DEBUG: Running node-supervisor with
DEBUG:   program 'app.js'
DEBUG:   --watch '.'
DEBUG:   --extensions 'node|js'
DEBUG:   --exec 'node'
DEBUG: Starting child process with 'node app.js'
DEBUG: Watching directory '/home/byvoid/.' for changes.

当代码被改动时，运行的脚本会被终止，然后重新启动。在终端中显示的结果如下：
DEBUG: crashing child
DEBUG: Starting child process with 'node app.js'
```

全局对象

属性	说明
global	全局的命名空间对象(全局对象)
console	控制台
Buffer	用于处理二进制数据
__dirname	全局变量 返回被执行的 js 所在文件夹的绝对路径
__filename	全局变量 返回被执行的 js 的绝对路径
module	对当前模块的引用
process	用于访问进程信息
exports	这是一个对于 `module.exports` 的更简短的引用形式
require(module)	引入模块
setImmediate(cb[, ...args])	即时定时器（立即执行），它是在事件轮询之后执行,为了防止轮询阻塞,每次只会调用一个。
clearImmediate(immediateObject)	清除setImmediate定时器
setInterval(cb, ms)	间隔定时器
clearInterval(intervalObject)	清除setInterval定时器
setTimeout(cb, ms)	超时定时器
clearTimeout(timeoutObject)	清除setTimeout定时器

模块

模块是Node.js 应用程序的基本组成部分，文件和模块是一一对应的。换言之，一个xxx.js 文件就是一个模块，这个文件可能是JavaScript 代码、JSON 或者编译过的C/C++ 扩展。

Node.js 提供了 **exports** 和 **require** 两个对象，其中：

exports： 是公开模块的接口

require ：用于从外部获取一个模块的接口（即所获取模块的 exports 对象）。

创建模块

创建 hello.js 文件，代码如下：

```
var txt="hello world";
let run= function() {
    console.log(txt);
};
//--------------------------------
//输出对象
module.exports.hello=run; //module是公开的接口（对象），exports是向module上添加属性或方法
```

引入模块

创建一个 main.js，在main.js中引入hello.js模块

```
var hello = require('./hello');  //必须写路径！！！（可以省略.js后缀名）
//hello 就是

hello.world();
//通过 require('./hello') 加载这个模块，就可以直接访问hello.js中exports对象的成员函数了
```

NPM

module（模块）：每一个js文件，就是一个模块。

package（包）：由多个模块组成，将某个独立的功能封装起来，用于发布、更新、依赖管理和版本控制。

node根据common JS规范实现了包机制，开发了npm来解决包的发布和获取需求。

NPM（Node Package Manager）是NodeJS包管理和分发工具，是随同NodeJS一起安装的。

（详见 NPM笔记）

核心模块

Node.js本身提供了很多核心模块（又称为内置模块、原生模块），用于与操作系统互动（读写文件，访问网络等），这些核心模块被编译成二进制文件，具有最高的加载优先级。

FS模块

1. 文件操作

方法	功能
fs.readFile(filename,[options],callback)	**异步**读取文件（非堵塞，性能更好） callback：function（err，data）
fs.readFileSync('input.txt');	**同步**读取文件
fs.writeFile(filename,data, [options],callback)	文件写入内容，覆盖文件原有的内容（如果文件不存在，先创建） [options]：指定权限，读、写、执行。是否可续写
fs.appendFile(filename,data, [options],callback)	**异步**追加的方式写文件（如果文件不存在，先创建） data：文本或buffer流
fs.appendFileSync(file, data[, options])	**同步**追加
fs.unlink(path, callback)	**异步**删除文件
fs.unlinkSync(path)	**同步**删除文件
fs.stat(path,callback)	检测文件状态 callback：function（err，data） err：错误信息，stats：文件状态对象
fs.existsSync(path)	查看**文件** 或 **目录**是否存在
fs.rename(oldname,newname,callback)	重命名文件名或者目录

2. 目录操作

方法	说明
fs.readdir(path,callback)	读取目录
fs.mkdir(path,[mode],callback)	创建目录 path：被创建目录的完整路径以及目录名（绝对路径） [mode]：目录权限，默认0777（可读可写可执行）
fs.rmdir(path,callback)	删除 **空** 目录

path模块

操作文件，就必定会遇到解析各种复杂路径的问题，path用于处理**服务器上（硬盘）**的文件路径

方法	说明
path.normalize()	规范化给定的 `path`，修正格式错误，并解析 `'..'` 和 `'.'` 片段
path.join()	路径合并
path.resolve(字符串)	获取绝对路径
path.relative(from，to)	获取from 到 to 相对路径
path.parse(路径)	将路径解析成对象

./ 在 require() 中使用跟 __dirname 的效果相同, 不因启动脚本目录不同而改变

在其他情况下跟 process.cwd()（返回运行 node 命令时所在的文件夹的绝对路径）效果相同，是相对于启动脚本所在目录的路径

http模块

Node.js 标准库提供了 http 模块，其中封装了一个高效的 HTTP 服务器和一个简易的 HTTP 客户端

方法	说明
http.Server()	构造函数，服务器是这个构造函数的实例 例：const sev=new http.Server()
http.createServer(callback)	工厂模式创建服务器的（快捷方法）
http.get(path,callback)	发送get请求
http.request(options,callback)	是一个 HTTP 客户端工具，用于向 HTTP 服务器发起请求 callback作为回调函数，需要传递一个参数 options是一个对象，表示请求的参数，常用的参数有host、port（默认为80）、method（默认为GET）、path（请求的相对于根的路径，默认是“/”。

1.创建web服务器

```
const http=require("http");
//Server是一个构造函数
//console.log(http.Server);

//创建一个server的实例
var server = new http.Server();

//监听事件
server.on('request',(req,res)=>{
    console.log(req.url);
    //设置应答头信息
    res.writeHead(200,{"content-type":"text/html;charset=utf-8"});
```

```
    res.write('hello world 你好<br>');
    res.end('服务器已经停止 server already end\n');
});

//定义端口
server.listen(8080,"127.0.0.1");
console.log("server is runing at 8080:127.0.0.1");
//------------------------------------------------------------------------

//http.createServer为以上的捷径方法
http.createServer((req,res)=>{

})
```

2. request

http请求分为二部分：

请求头：如果请求的内容少的话就直接在请求头，协议完成之后立即读取 get请求

请求体：大文件需要放在请求体传输post请求，这个传输过程需要一定的传输时间

因此提供了三个事件用于控制请求体传输：

1. data：当请求体数据到来时，该事件被触发，该事件一共一个参数chunk，表示接受到的数据。
2. end：当请求体数据传输完成时，该事件被触发，此后将不会再有数据到来。
3. close：用户当前请求结束时，该事件被触发，不同于end，如果用户强制终止了传输，也会触发close

request的属性

名称	含义
ccomplete	客户端请求是否已经发送完成
httpVersion	HTTP协议版本，通常是1.0或1.1
method	HTTP请求方法，如：GET,POST
url	原始的请求路径
headers	HTTP请求头
trailers	HTTP请求尾(不常见)
connection	当前HTTP连接套接字，为net.Socket的实例
socket	connection属性的别名
client	client属性的别名

```
const http = require('http');
const fs = require('fs');
const url = require('url');
//querystring用于处理URL中的查询字符串
```

```
const querystring=require("querystring");


const server=http.createServer((req,res)=>{
   var post="";
   req.on("data",function(chunk){
       post+=chunk;
   });

    //接受请求结束后
    req.on("end",function(){
        //post接收的数据 将字符串变为json的格式
        var postData=querystring.parse(post);

        //get接收的数据
        var getData=url.parse(req.url);
    })
});

server.listen(8080,"127.0.0.1");
console.log("server ruuning at http://127.0.0.1:8080");
```

3. response

response 是返回给客户端的信息，决定了用户最终能看到的结果。作为第二个参数传递，一般简称为 response 或 res。

response的方法

方法	说明
response.writeHead(statusCode, [headers])	向请求的客户端发送响应头。 statusCode 是 HTTP 状态码，如 200 （请求成功）、404 （未找到）等。 headers 是一个类似关联数组的对象，表示响应头的每个属性。该函数在一个请求内最多只 能调用一次，如果不调用，则会自动生成一个响应头。
response.write(data, [encoding])	向请求的客户端发送响应内容。 data 是 一个 Buffer 或字符串，表示要发送的内容。如果 data 是字符串，那么需要指定 encoding 来说明它的编码方式，默认是 utf-8。在 response.end 调用之前， response.write 可以被多次调用。
response.end([data], [encoding])	结束响应，告知客户端所有发送已经完 成。 当所有要返回的内容发送完毕的时候，该函数 必须 被调用一次。它接受两个可选参数（与 response.write 参数相同）。如果不调用该函数，客户端将永远处于 等待状态。

4. 获取GET请求内容

由于GET请求直接被嵌入在路径中,URL完整的请求路径，包括了？后面的部分，因此你可以手动解析后面的内容作为GET的参数，Nodejs的url模块中的parse函数提供了这个功能

```
const http = require('http');
const net  = require('net');
const url  = require('url');
const util = require('util');

http.createServer((req,res)=>{
    res.write(util.inspect(url.parse(req.url,true)));
    //利用url模块去解析客户端发送过来的URL
    res.end(util.inspect(url.parse(req.url,false)));
}).listen(8080);
```

5. 获取POST请求内容

POST请求的内容全部都在请求体中，http.ServerRequest并没有一个属性内容为请求体，原因是等待请求体传输可能是一件耗时的工作。譬如上传文件。恶意的POST请求会大大消耗服务器的资源。所以Nodejs是不会解析请求体，当你需要的时候，需要手动来做。

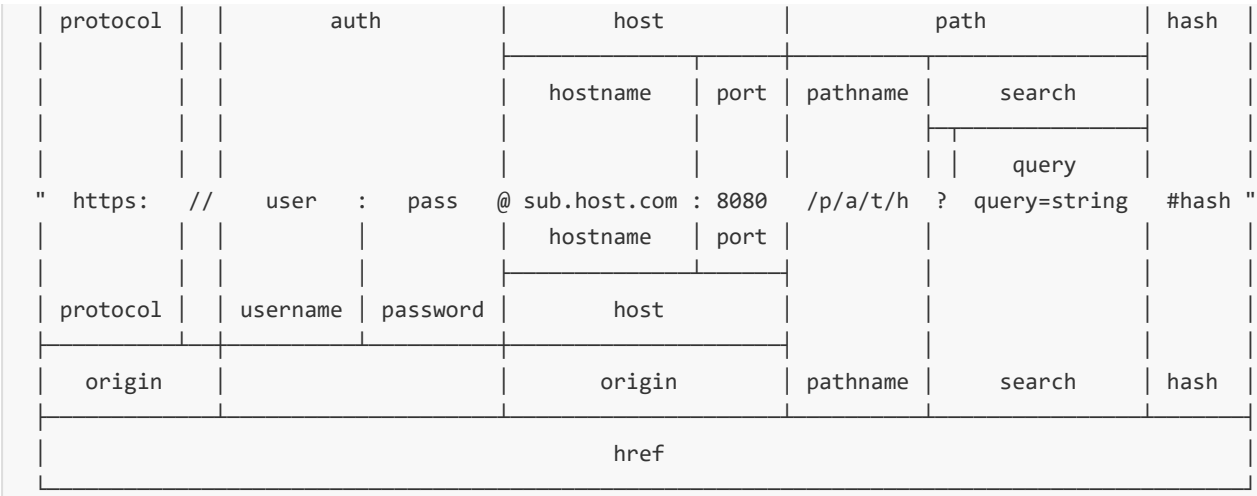
```
const http = require('http');
const net  = require('net');
const url  = require('url');
const util = require('util');
//querystring用于处理URL中的查询字符串
const querystring = require('querystring');

http.createServer((req,res)=>{
    var post = '';
    req.on('data',(chunk)=>{
        post+=chunk;
    });
    res,on('end',()=>{
        //将字符串变为json的格式
        post  =  querystring.parse(post);
        //向前端返回字符串
        res.end(util.inspect(post));
    });
})
```

url模块

用于 **url** 的处理与解析。

关于URI的格式是由RFC 3986规定的，一个url是一个结构化的字符串，它包含多个有意义的组成部分。 当被解析时，会返回一个 URL 对象，它包含每个组成部分作为属性。

```
┌─────────────────────────────────────────────────────────────────────────────────────────────┐
│                                            href                                             │
├──────────┬──┬─────────────────────┬────────────────────┬─────────────────────────────┬───────┤
```



名称	解释
origin	起点 起源
protocol	协议 常见协议：http、https、ftp、file等
auth	身份 某些协议需要验证身份才能登陆， 例如：ftp://username:password@xxx.xxx.xxx.xxx:port
host	主机 包括 主机名（hostname）和 端口号（port）
hostname	主机名
port	端口
path	路径 包含：pathname（路径名）和 search（ ）
search	搜索（查询） 从“？”开始到“#”为止之间的部分为查询部分 包含：query（ 查询字符串）
query	查询字符串 格式：key1=value1 **&** key2=value2
hash	哈希 # 代表网页中的一个锚点， 浏览器会自动把该锚点位置滚动到页面可视区域内 #号是用来指导浏览器动作的，对服务器端完全无用。所以，HTTP请求中不包含#。 每一次改变#后的部分，都会在浏览器的访问历史中增加一个记录，使用"后退"按钮，就可以回到上一个位置。 这对于ajax应用程序特别有用，可以用不同的#值，表示不同的访问状态，然后向用户给出可以访问某个状态的链接(单页面路由)。 window.location.hash这个属性可读可写。读取时，可以用来判断网页状态是否改变；写入时，则会在不重载网页的前提下，创造一条访问历史记录。

该模块提供了一些实用函数， 可以通过以下方式使用：

方法	说明
url.parse(urlString，参数)	解析url地址，返回一个对象 参数：ture 可以将对象中query输出为对象，false：输出字符串
url.toString()	在URL对象上调用 toString() 方法将返回序列化的URL。 返回值与url.href和url.toJSON()的相同
url.toJSON()	在URL对象上调用 toJSON() 方法将返回序列化的URL。 返回值与url.href和url.toString()的相同

属性	说明
url.host	获取及设置URL的主机(host)部分
url.hostname	获取及设置URL的主机名(hostname)部分
url.href	获取及设置序列化的URL
url.origin	获取只读序列化的URL origin部分
url.protocol	获取及设置URL的协议(protocol)部分
url.username	获取及设置URL的用户名(username)部分
url.password	获取及设置URL的密码(password)部分
url.pathname	获取及设置URL的路径(path)部分
url.port	获取及设置URL的端口(port)部分
url.search	获取及设置URL的序列化查询(query)部分
url.searchParams	获取表示URL查询参数的URLSearchParams对象。该属性是只读的
url.hash	获取及设置URL的分段(hash)部分

events模块

1. 事件驱动编程

事件驱动编程（Evnet-driven programming）是一种编程风格，由事件来决定程序的执行流程，事件由事件处理器（event handler）或事件回调（event callback）来处理，事件回调是当某个特定事件发生时被调用的函数，比如数据库返回了查询结果或服务器响应到了一个客户端的连接请求。

events 是 Node.js 最重要的模块，没有“之一”，原因是 Node.js 本身架构就是事件式的，而它提供了唯一的接口，所以堪称 Node.js 事件编程的基石。大多数时候我们不会直接使用 EventEmitter，而是在对象中继承它。包括 fs、net、 http 在内的，只要是**支持事件响应的核心模块都是 EventEmitter 的子类。**

2. EventEmitter

events 模块只提供了一个对象： **events.EventEmitter**，EventEmitter 的核心就 是事件发射与事件监听器功能的封装。EventEmitter 的每个事件由一个事件名和若干个参 数组成，事件名是一个字符串，通常表达一定的语义。对于每个事件，EventEmitter 支持 若干个事件监听器。当事件发射时，注册到这个事件的事件监听器被依次调用，事件参数作 为回调函数参数传递。

```
var emitter = new events.EventEmitter();
emitter.on('someEvent', function(arg1, arg2) {
    console.log( arg1, arg2);
});
```

后端事件与前端DOM树上的事件有所不相同，因为它不存在冒泡，逐层捕获等属于DOM的事件行为，也没有preventDefault()、stopPropagation()等处理事件传递的方法。

事件侦听器模式是一种事件钩子（hook）的机制，通过事件钩子的方式，可以使编程者不用关注组件是如何启动和执行的，只需关注在需要的事件点上即可。作为后端服务器来说，Node.js重点事件更常见的是网络事件，包括：

1. 来自web服务器的响应
2. 从文件读取数据
3. 从数据库返回数据

EventEmitter类的各种方法

方法名与参数	描述
addListener(event,listener)	为指定事件绑定事件处理函数
on(event, listener)	addListener方法的别名
once(event, listener)	为指定事件注册单次监听器，触发后立刻解除该监听器
removeListener(event, listener)	移除指定事件的某个监听器
setMaxListeners(n)	指定事件处理函数的最大数量，n为正数值，代表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

```
//来自web服务器的响应的简单案例
const http=require("http");

//创建一个server的实例
var server = new http.Server();//Server是一个类（构造函数）

//在server上监听request事件
server.on('request',(req,res)=>{

    console.log("请求的内容是："+req.url);
```

```
    //设置应答头信息
    res.writeHead(200,{"content-type":"text/html;charset=utf-8"});
    res.write('hello world 你好<br>');
    res.end('服务器已经停止 server already end');
});
//定义端口
server.listen(8080);
console.log("server runing at 8080");
```

until模块

弥补js功能不足，新增的部分API。

方法	说明
util.format()	返回一个格式化后的字符串
util.inspect()	是一个将任意对象转换为字符串的函数
util.isArray()	判断是否是数组
util.isRegExp()	判断是否是正则对象
util.isDate()	判断是否是日期对象
util.isError()	判断是否是error对象
util.promisify(original)	转换为基于promise的方法

第三方模块

node自带的NPM程序，提供了丰富的第三方模块，可以登录NPM官网搜索查询。

实例演示

clean-css

1. CSS减肥优化
2. 检查CSS是否正确（检查有无拼写错误、是否忘记结尾的}等）。

安装

```
npm install clean-css
```

应用

```
//官方代码
//var CleanCSS = require('clean-css');
//var input = 'a{font-weight:bold;}';
//var options = { /* options */ };
```

```
//var output = new CleanCSS(options).minify(input);

//引入模块
var fs = require('fs');

//引入clean-css模块
var CleanCSS = require('clean-css');

var style=`
body{
    margin: 0;
    padding:0;
}
.btn{
    width: 100px;
    height: 50px;
    background: #ccc;
}`;


//实例化对象
var cssMinify = new CleanCSS({
  format: 'keep-breaks' // formats output the default way but adds line breaks for improved
readability  参数设置，请查看clean-css手册
});

var min = cssMinify.minify(style);
console.log(min);

//写入新文件
fs.writeFile('./new_style.css', min.styles,function(){
    console.log('success');
});

//-----------------------------------------------------------------
//读取style.css文件
fs.readFile('./style.css', 'utf8', function (err, data) {
    if (err) {
        throw err;
    }
    //压缩文件
    var min = cssMinify.minify(data);
    //console.log(min);
    //另存文件
    fs.writeFile('./new_style.css', min.styles,function(){
        console.log('success');
    });
});
```