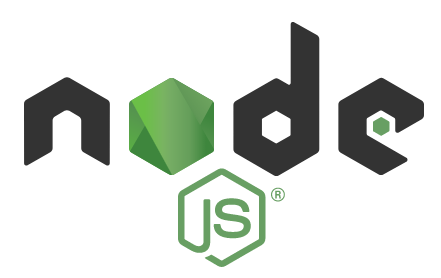
**Node.js基础学习笔记**



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。

2018年3月11日

目录

[一、Node简介 1](#_Toc510634226)

[1. Node的诞生 2](#_Toc510634227)

[2. Node的实现 2](#_Toc510634228)

[3. Node在Web中的用途 3](#_Toc510634229)

[二、Node环境配置 4](#_Toc510634230)

[1. NVM工具的使用 4](#_Toc510634231)

[三、基础概念概要 5](#_Toc510634232)

[1. REPL环境 5](#_Toc510634233)

[2. node基本语法 5](#_Toc510634234)

[3. 全局对象 6](#_Toc510634235)

[4. 全局函数 6](#_Toc510634236)

[5. Node 调试 6](#_Toc510634237)

[四、异步编程 7](#_Toc510634238)

[1. 异步操作 7](#_Toc510634239)

[2. 什么是异步 7](#_Toc510634240)

[3. 异步回调函数 7](#_Toc510634241)

[3.1. 回调函数的设计 7](#_Toc510634242)

[4. 异步回调的问题 8](#_Toc510634243)

[五、进程和线程 8](#_Toc510634244)

[1. 什么是进程 8](#_Toc510634245)

[2. 什么是线程 9](#_Toc510634246)

[3. 多线程的问题 9](#_Toc510634247)

[六、非阻塞I/O 10](#_Toc510634248)

[1. 什么是I/O 10](#_Toc510634249)

[2. 事件驱动和非阻塞机制 10](#_Toc510634250)

[3. 非阻塞的优势 12](#_Toc510634251)

[4. 平台的差异 12](#_Toc510634252)

[七、模块化结构 12](#_Toc510634253)

[1. 模块化代码结构 12](#_Toc510634254)

[2. CommonJS 规范概述 13](#_Toc510634255)

[3. 模块的分类 13](#_Toc510634256)

[4. 模块化开发的流程 14](#_Toc510634257)

[5. 定义模块 14](#_Toc510634258)

[5.1. 模块内全局环境（伪） 14](#_Toc510634259)

[5.2. module对象 15](#_Toc510634260)

[5.3. 模块定义 15](#_Toc510634261)

[6. 载入模块 15](#_Toc510634262)

[6.1. require 简介 15](#_Toc510634263)

[6.2. 模块的加载机制 16](#_Toc510634264)

[6.3. require扩展名 16](#_Toc510634265)

[6.4. require加载文件规则 16](#_Toc510634266)

[6.5. Require的实现机制 17](#_Toc510634267)

[7. 缓存模块 17](#_Toc510634268)

[8. 核心模块和对象 17](#_Toc510634269)

[8.1. 核心模块的意义 17](#_Toc510634270)

[9. 包 18](#_Toc510634271)

[9.1. 什么是包 18](#_Toc510634272)

[9.2. 包的加载机制 18](#_Toc510634273)

[10. NPM包管理工具 18](#_Toc510634274)

[10.1. NPM 概述 18](#_Toc510634275)

[10.2. 安装 NPM 19](#_Toc510634276)

[10.3. 常用NPM命令 19](#_Toc510634277)

[八、文件操作 19](#_Toc510634278)

[1. 相关模块 19](#_Toc510634279)

[2. 路径操作模块（path） 20](#_Toc510634280)

[3. 同步或异步调用 20](#_Toc510634281)

[4. 文件读取模块（fs） 20](#_Toc510634282)

[5. 文件写入模块（fs） 20](#_Toc510634283)

[6. 其他文件操作 21](#_Toc510634284)

[7. 缓冲区处理（Buffer二进制数据） 22](#_Toc510634285)

[7.1. 什么是缓冲区 22](#_Toc510634286)

[7.2. 为什么要有缓冲区 22](#_Toc510634287)

[7.3. 创建缓冲区 22](#_Toc510634288)

[7.4. Node默认支持的编码 22](#_Toc510634289)

[8. 文件流 23](#_Toc510634290)

[8.1. 文件操作进阶 23](#_Toc510634291)

[8.2. 什么是流 23](#_Toc510634292)

[8.3. 什么是文件流 23](#_Toc510634293)

[8.4. Node中文件流的操作 24](#_Toc510634294)

[九、网络操作 24](#_Toc510634295)

# 一、Node简介

Node.js是一个Javascript运行环境(runtime)，发布于2009年5月，由Ryan Dahl开发，实质是对Chrome V8引擎进行了封装。Node.js对一些特殊用例进行优化，提供替代的API。

Node.js 使用事件驱动， 非阻塞I/O 模型而得以轻量和高效。Node.js 的包管理器 npm，是全球最大的开源库生态系统。非常适合在分布式设备上运行数据密集型的实时应用。

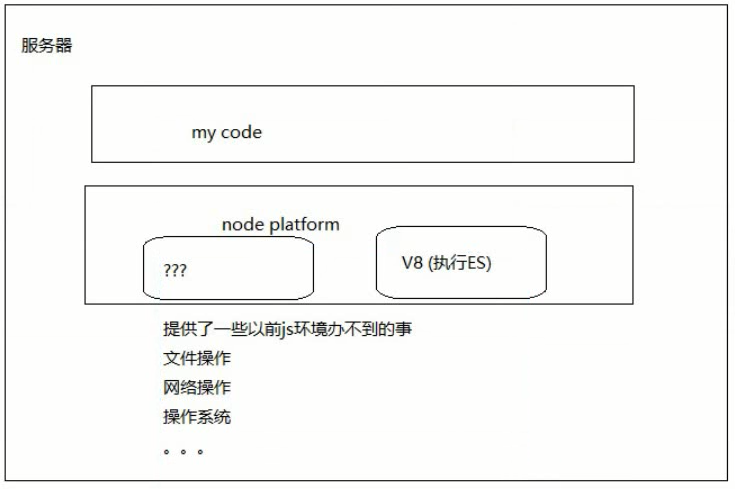
JavaScript是运行在浏览器的js引擎（engine）。

-语言的能力一定取决于平台

Node就是JavaScript语言在服务器运行的环境（平台）

运行环境（平台）有两层意思：

* 首先：JavaScript语言通过Node在服务器运行的，在这个意义上，node有点像JavaScript虚拟机；
* 其次：Node提供大量工具库，使得JavaScript语言与操作系统互动（比如读写文件，新建子进程），在这个意义上，Node又是JavaScript工具库



**Node平台选择了JavaScript：**

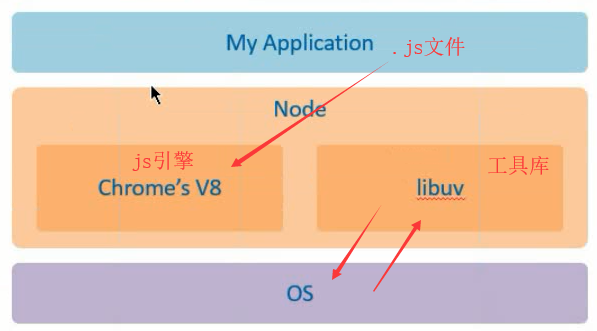


## Node的诞生



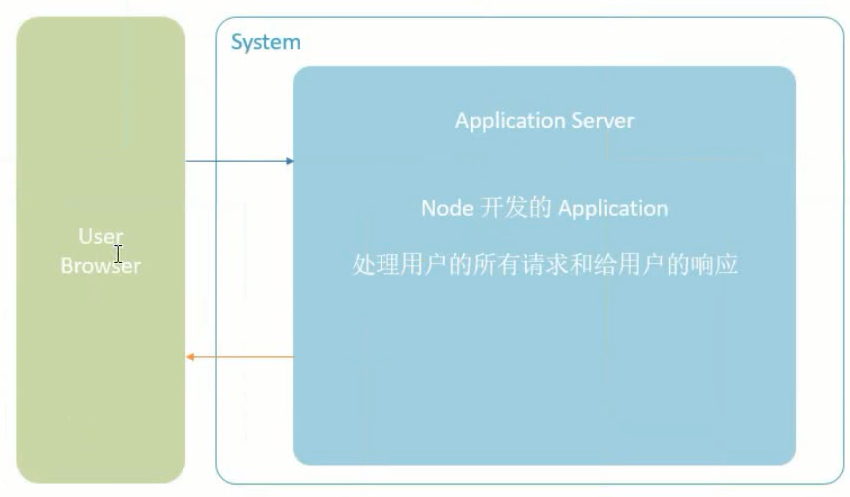
## Node的实现

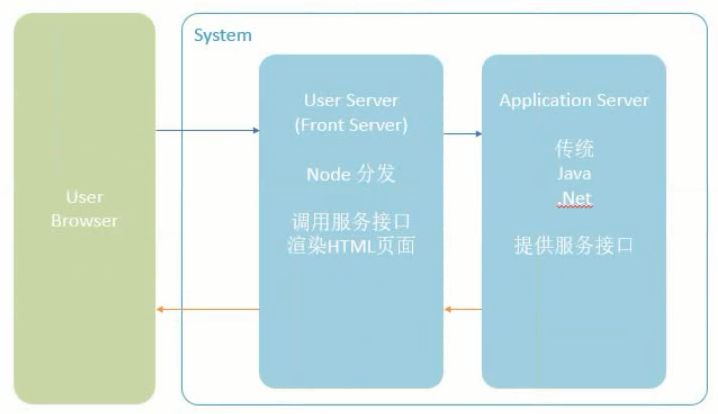
Node内部采用google浏览器的v8引擎，作为了JavaScript语言解释器。通过自行开发的libve库，调用操作系统资源。



## Node在Web中的用途

* 做网站 (全包)



* 分发数据请求，渲染HTML（只做一部分）  
  

# 二、Node环境配置

安装包方式（不推荐）：

太傻瓜式安装。Next、next、next。

更新的时候会出现覆盖性安装。

无法多版本之间切换

## NVM工具的使用

Node Version Manager（Node版本管理工具）

安装操作步骤

下载：nvm-windows

解压到一个全英文路径

编辑解压目录下的settings.txt文件（不存在则新建）

- root 配置为当前 nvm.exe 所在目录

- path 配置为 node 快捷方式所在的目录

- arch 配置为当前操作系统的位数（32/64）

- proxy 不用配置

配置环境变量 可以通过 window+r : sysdm.cpl

- NVM\_HOME = 当前 nvm.exe 所在目录

- NVM\_SYMLINK = node 快捷方式所在的目录

- PATH += %NVM\_HOME%;%NVM\_SYMLINK%;

- 打开CMD通过set [name]命令查看环境变量是否配置成功

- PowerShell中是通过dir env:[name]命令

NVM使用说明：

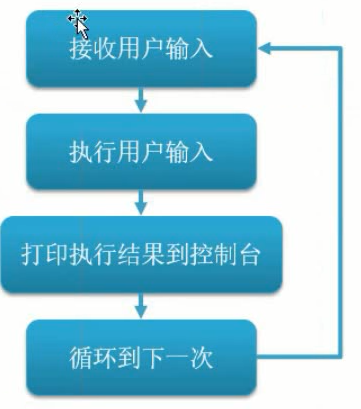
- https://github.com/coreybutler/nvm-windows/

NPM的目录之后使用再配置

# 三、基础概念概要

## REPL环境

REPL全称：Read Eval Print Loopl



• 进入 REPL：

– node

– node --use\_strict

• REPL 环境中：

– 类似 Chrome Developer Tools → Consoles

– 特殊变量下划线（\_）表示上一个命令的返回结果

– 通过 .exit 或执行 process.exit() 退出 REPL 交互

## node基本语法

进入 REPL 环境：

– $ node

• 执行脚本字符串：

– $ node -e 'console.log("Hello World")'

• 运行脚本文件：

– $ node index.js

– $ node path/index.js

– $ node path/index

• 查看帮助：

– $ node --help

## 全局对象

* **global**：

– 类似于客户端 JavaScript 运行环境中的 window

* **process**：

– 用于获取当前的 Node 进程信息，一般用于获取环境变量之类的信息

常用方法：

process.stdout.write( ); //控制台输出

process.stdin.read( )；//控制台输入

* **console**：

– Node 中内置的 console 模块，提供操作控制台的输入输出功能，常见使用方式与客户端类似

## 全局函数

定时器：

* setInterval(callback, millisecond)
* clearInterval(timer)

延时器：

* setTimeout(callback, millisecond)
* clearTimeout(timer)
* Buffer：Class
  + 用于操作二进制数据

## Node 调试

* 最方便也是最简单的：console.log()
* Node 原生的调试

– https://nodejs.org/api/debugger.html

* 第三方模块提供的调试工具

– $ npm install node-inspector –g

– $ npm install devtool -g

* 开发工具的调试

– 推荐 Visual Studio Code

– WebStorm

# 四、异步编程

## 异步操作

异步操作：

* Node 采用 Chrome V8 引擎处理 JavaScript 脚本，V8 最大特点就是单线程运行，一次只能运行一个任务。
* Node 大量采用异步操作（asynchronousoperation），即任务不是马上执行，而是插在任务队列的尾部，等到前面的任务运行完后再执行。提高代码的响应能力。
* 因为之后的操作大多数都是异步的方式，无法通过 trycatch 捕获异常，SO，错误优先的回调函数，第一个参数为上一步的错误信息。

## 什么是异步

**实际生活中如：**

* 打电话、发短信就是异步。

**程序世界中：**

* setTImeout（）
* $.ajax()
* 文件操作
* Node所有的会发生阻塞的操作都是异步。

总结： 所有与网络有关和磁盘操作有关的都是异步。

## 异步回调函数

### 回调函数的设计

**Node回调函数：**

由于系统永远不知道用户什么时候会输入内容，所以代码不能永远停在一个地方；Node 中的操作方式就是以异步回调的方式解决无状态的问题；

对于一个函数如果需要定义回调函数：

– 回调函数一定作为参数的最后一个参数出现：

• function foo1(name, age, callback) { }

• function foo2(value, callback1, callback2) { }

– 回调函数的第一个参数默认接收错误信息，第二个参数才是真正的回调数据（便于外界获取调用的错误情况）：

Eg：foo1('赵小黑', 19, function(error, data) {

if(error) throw error;

console.log(data);

});

– Node 统一约定

因为之后的操作大多数都是异步的方式，无法通过 try

catch 捕获异常，SO

错误优先的回调函数

– 第一个参数为上一步的错误信息

如果不这样调用的话就体现不出异步了。

## 异步回调的问题

do1(function() {

do2(function() {

do3(function() {

do4(function() {

do5(function() {

do6()

});

});

});

});

});

相比较于传统的代码：

– 异步事件驱动的代码不容易阅读

– 不容易调试

– 不容易维护

# 五、进程和线程

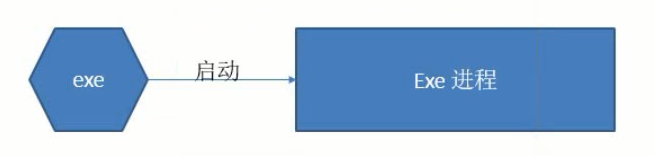
## 什么是进程

每一个 正在运行 的应用程序都称之为进程。

每一个应用程序都至少有一个进程

进程是用来给应用程序提供一个运行的环境

进程是操作系统为应用程序分配资源的一个单位



## 什么是线程

• 用来执行应用程序中的代码

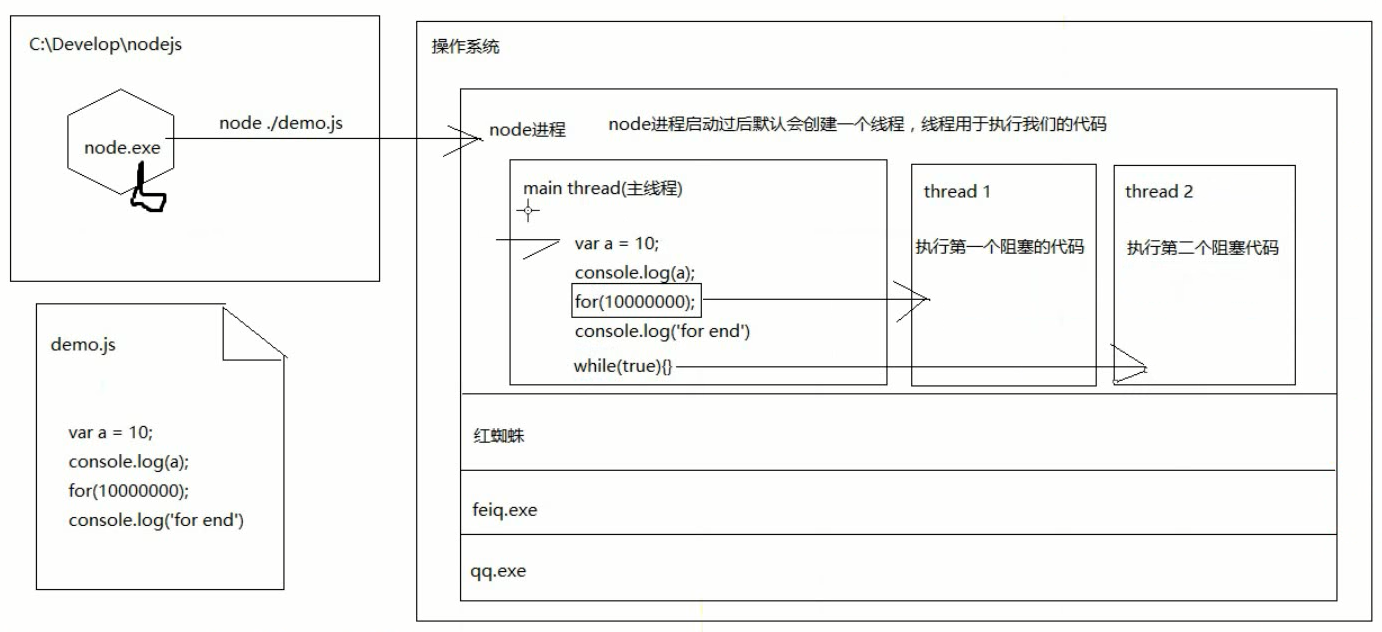
• 在一个进程内部，可以有很多的线程

• 在一个线程内部，同时只可以干一件事

• 而且传统的开发方式大部分都是 I/O 阻塞的

• 所以需要多线程来更好的利用硬件资源

• 给人带来一种错觉：线程越多越好



## 多线程的问题

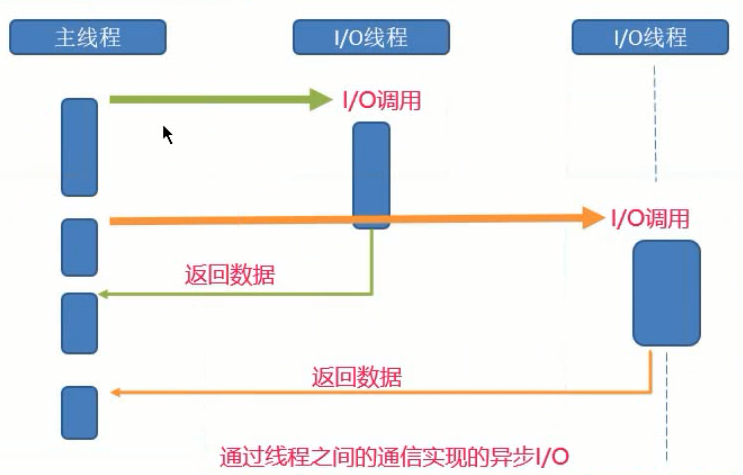
* 多线程都是假相（只是体现出多线程而已）
* 线程之间共享某些数据，同步某个状态都很麻烦
* 更致命的是：
  + -创建线程耗费时间
  + -线程数量有限
  + -CPU在不同线程自己转换，有上下文的转换，这个转换非常耗时。

# 六、非阻塞I/O

Node充分利用单线程

## 什么是I/O

i/o（input/output），可以理解为输入到输出之间的转化过程



例如：

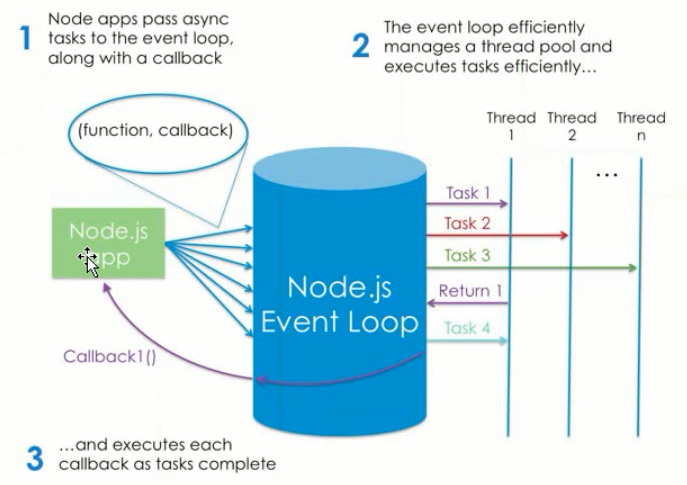
– 敲击键盘（输入）看到编辑器中多出字符（输出）

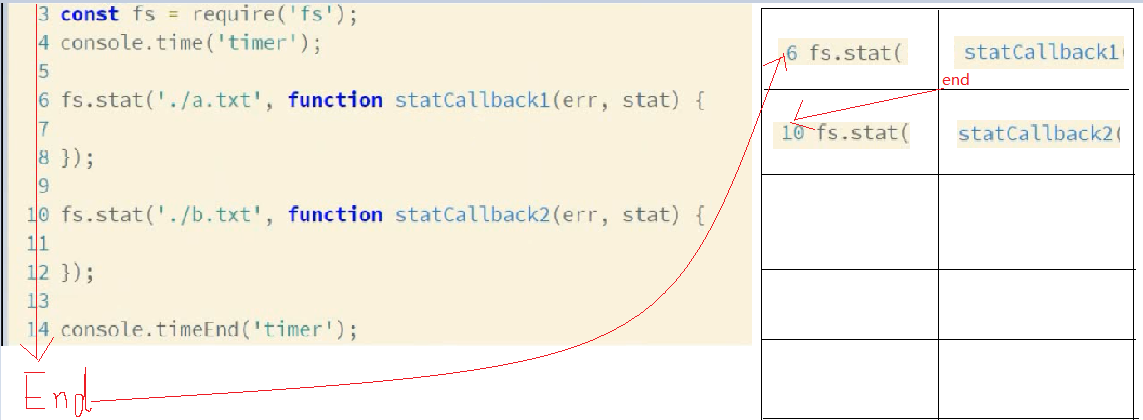
– 移动鼠标（输入）看到光标移动（输出）

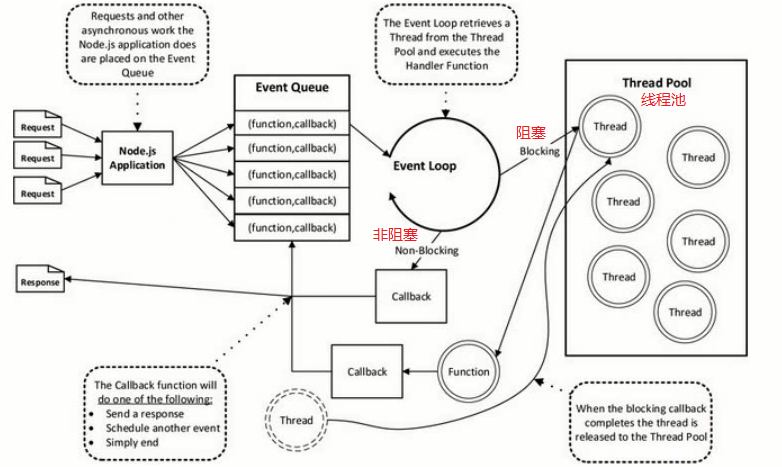
– 小霸王

## 事件驱动和非阻塞机制

1. Node平台将一个任务连同该任务的回调函数放到一个事件循环系统中。
2. 事件循环高效的管理系统线程池同时高效执行每一个任务。
3. 当事件执行完成才会调用回调函数





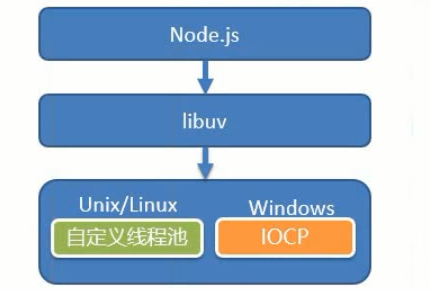


## 非阻塞的优势

* 提高代码的响应效率
* 充分利用单核 CPU 的优势
* 改善 I/O 的不可预测带来的问题
* 如何提高一个人的工作效率
* 但是目前市面上大多数都是多核CPU  
  node采用了多核变单核来运行node程序

## 平台的差异

由于 Windows 和 \*nix 平台的差异，Node 提供了libuv 作为抽象封装层，保证上层的 Node 与下层的自定义线程池及 IOCP 之间各自独立。

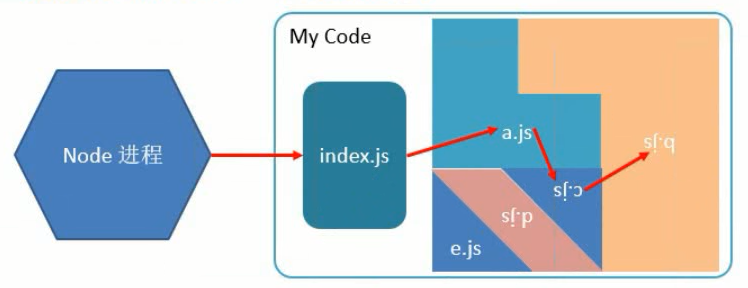


# 七、模块化结构

Node 实现 CommonJS 规范，所以可以使用模块化的方式组织代码结构

## 模块化代码结构

* Node 采用的模块化结构是按照 CommonJS 规范
* 模块与文件是一一对应关系，即加载一个模块，实际上就是加载对应的一个模块文件。



## CommonJS 规范概述

CommonJS 就是一套约定标准，不是技术；用于约定我们的代码应该是怎样的一种结构；

http://wiki.commonjs.org/wiki/CommonJS

* 所有代码都运行在模块作用域，不会污染全局作用域。
* 模块可以多次加载，但是只会在第一次加载时运行一次，然后运行结果就被缓存了，以后再加载，就直接读取缓存结果。要想让模块再次运行，必须清除缓存。
* 模块加载的顺序，按照其在代码中出现的顺序

## 模块的分类

* 文件模块

– 就是我们自己写的功能模块文件

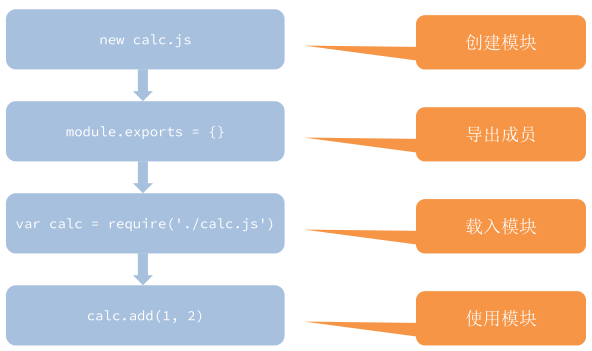
* 核心模块

– Node 平台自带的一套基本的功能模块，也有人称之为 Node平台的 API

* 第三方模块

– 社区或第三方个人开发好的功能模块，可以直接拿回来用

## 模块化开发的流程



## 定义模块

### 模块内全局环境（伪）

我们在之后的文件操作中必须使用绝对路径

* \_\_dirname

– 用于获取当前文件所在目录的完整路径；

– 在 REPL 环境无效；

* \_\_filename

– 用来获取当前文件的完整路径；

– 在 REPL 环境同样无效；

* module

– 模块对象

* exports

– 映射到 module.exports 的别名

* require()

– require.cache

– require.extensions

– require.main

– require.resolve()

### module对象

* Node 内部提供一个 Module 构建函数。所有模块都是

Module 的实例，属性如下：

– module.id 模块的识别符，通常是带有绝对路径的模块文件名。

– module.filename 模块定义的文件的绝对路径。

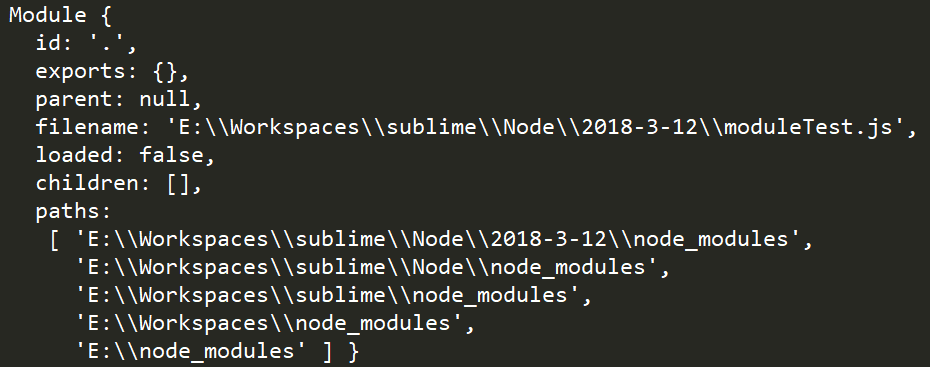
– module.loaded 返回一个布尔值，表示模块是否已经完成加载。

– module.parent 返回一个对象，表示调用该模块的模块。

– module.children 返回一个数组，表示该模块要用到的其他模块。

– module.exports 表示模块对外输出的值。

* 载入一个模块就是构建一个 Module 实例



### 模块定义

一个新的 JS 文件就是一个模块；

一个合格的模块应该是有导出成员的，否则模块就失去了定义的价值；

模块内部是一个独立（封闭）的作用域（模块与模块之间不会冲突）；

模块之间必须通过导出或导入的方式协同；

导出方式：

– exports.name = value;

– module.exports = { name: value };

## 载入模块

require 函数

### require 简介

Node 使用 CommonJS 模块规范，内置的 require 函数用于加载模块文件。

require 的基本功能是，读入并执行一个 JavaScript 文件，然后返回该模块的 exports 对象。

如果没有发现指定模块，会报错。

### 模块的加载机制

id: 路径的情况就是直接以相对路径的方式找文件



### require扩展名

* require 加载文件时可以省略扩展名：

//如果module是目录就加载目录下的Default.js文件，没有则加载index.js

– require('./module');

// 此时文件按 JS 文件执行

– require('./module.js');

// 此时文件按 JSON 文件解析

– require('./module.json');

// 此时文件预编译好的 C++ 模块执行

– require('./module.node');

### require加载文件规则

* 通过 ./ 或 ../ 开头：则按照相对路径从当前文件所在文件夹开始寻找模块；

– require('../file.js'); => 上级目录下找 file.js 文件

* 通过 / 开头：则以系统根目录开始寻找模块；

– require('/Users/iceStone/Documents/file.js');

=> 以绝对路径的方式找

* 如果参数字符串不以“./“ 或 ”/“ 开头，则表示加载的是一个默认提供的核心模块（位于 Node 的系统安装目录中）：

– require('fs'); => 加载核心模块中的文件系统模块

* 或者从当前目录向上搜索 node\_modules 目录中的文件：

– require('my\_module'); => 各级 node\_modules 文件夹中搜索 my\_module.js 文件；

* 如果 require 传入的是一个目录的路径，会自动查看该目录的 package.json 文件，然后加载 main 字段指定的入口文件
* 如果package.json文件没有main字段，或者根本就没有package.json文件，则默认找目录下的 index.js 文件作为模块：

– require('./calcuator'); => 当前目录下找 calculator目录中的 index.js 文件

### Require的实现机制

* + 将传入的模块 ID 通过加载规则找到对应的模块文件
  + 读取这个文件里面的代码
  + 通过拼接的方式为该段代码构建私有空间
  + 执行该代码
  + 拿到 module.exports 返回

## 缓存模块

* 第一次加载某个模块时，Node 会缓存该模块。以后再加载该模块，就直接从缓存取出该模块的module.exports 属性（不会再次执行该模块）
* 如果需要多次执行模块中的代码，一般可以让模块暴露行为（函数）
* 模块的缓存可以通过 require.cache 拿到，同样也可以删除

## 核心模块和对象

https://nodejs.org/api/

学会查 API，远远比会几个 API 更重要

### 核心模块的意义

如果只是在服务器运行 JavaScript 代码，意义并不大，因为无法实现任何功能（读写文件，访问网络）。

Node 的用处在于它本身还提供的一系列功能模块，用于与操作系统互动。

这些核心的功能模块在 Node 中内置。

内置如下模块：

* **path**：处理文件路径。
* **fs**：操作（CRUD）文件系统。
* **child\_process**：新建子进程。
* **util**：提供一系列实用小工具。
* **http**：提供 HTTP 服务器功能。
* **url**：用于解析 URL。
* **querystring**：解析 URL 中的查询字符串。
* **crypto**：提供加密和解密功能。

## 包

### 什么是包

由于 Node 是一套轻内核的平台，虽然提供了一系列的内置模块，但是不足以满足开发者的需求，于是乎出现了包（package）的概念：

与核心模块类似，就是将一些预先设计好的功能或者说 API 封装到一个文件夹，提供给开发者使用；

### 包的加载机制

* id: 包名的情况：require('http')

-先在系统核心（优先级最高）的模块中找；

-以后不要创建一些和现有的包重名的包；

-然后再到当前项目中 node\_modules 目录中找；

## NPM包管理工具

Node Package Manager

### NPM 概述

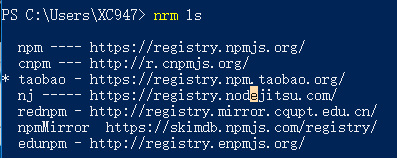
随着时间的发展，NPM 出现了两层概念：

* 第一层含义是 Node 的开放式模块登记和管理系统，亦可以说是一个生态圈，一个社区
* 另一层含义是 Node 默认的模块管理器，是一个命令行下的软件，用来安装和管理 Node 模块。

官方链接： https://www.npmjs.com/

国内加速镜像： <https://npm.taobao.org/>

可以通过 NRM： Node Registry Manager 的工具来选择源：



### 安装 NPM

NPM 不需要单独安装。默认在安装 Node 的时候，会连带一起安装 NPM。但是，Node 附带的 NPM 可能不是最新版本，最好用下面的命令，更新到最新版本。

$ npm install npm -g

默认安装到当前系统 Node 所在目录下。

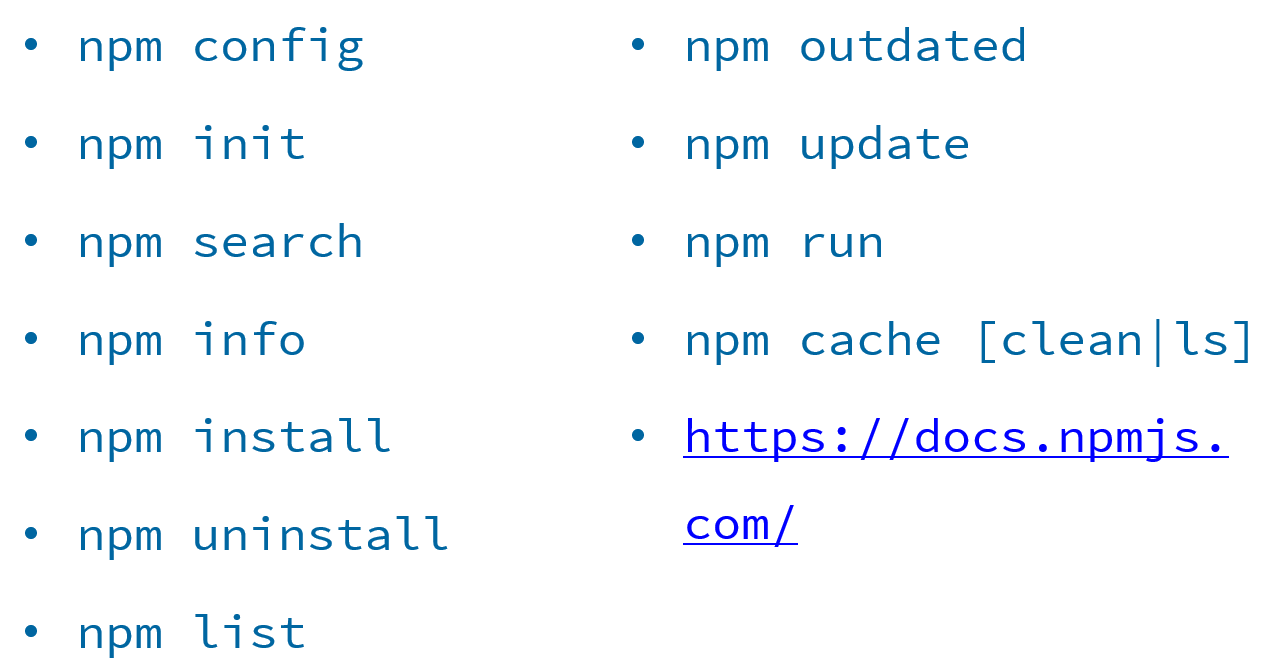
由于之前使用 NVM 的方式安装的 Node 所以需要重新配置 NPM 的全局目录

配置 NPM 的全局目录：

$ npm config set prefix [pathtonpm]

将 NPM 目录配置到其他目录时，必须将该目录放到环境变量中，否则无法再全局使用

### 常用NPM命令



https://docs.npmjs.com/

# 八、文件操作

一个应用程序免不了操作文件

## 相关模块

* **fs**：

基础的文件操作 API

* **path**：

提供和路径相关的操作 API

* **\*readline**：

用于读取大文本文件，一行一行读

* **fs-extra**（第三方）：

<https://www.npmjs.com/package/fs-extra>

## 路径操作模块（path）

在文件操作的过程中，都”必须”使用物理路径（绝对路径）path 模块提供了一系列与路径相关的 API

**path.join([p1][,p2][,p3]…)** => 连接多个路径

**path.basename(p, ext)** => 获取文件名

**path.dirname(p)** => 获取文件夹路径

**path.extname(p)** => 获取文件扩展名

**path.format(obj)** 和 **path.parse(p)**

**path.relative(from, to)** => 获取从 from 到 to 的相对路径

更多的使用请看说明文档：<http://nodejs.cn/api/path.html>

源码地址：<https://github.com/nodejs/node/blob/master/lib/path.js>

## 同步或异步调用

fs模块对文件的几乎所有操作都有同步和异步两种形式，在异步方法后面加 “**Sync**”’就变成同步的了

例如：readFile() 和 readFileSync()

区别：

* 同步调用会阻塞代码的执行，异步则不会
* 异步调用会将读取任务下达到任务队列，直到任务执行完成才会回调
* 异常处理方面，同步必须使用 try catch 方式，异步可以通过回调函数的第一个参数

## 文件读取模块（fs）

* 异步文件读取

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

* 同步文件读取

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

* 文件流的方式读取（后面单独介绍）

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

## 文件写入模块（fs）

确保操作没有额外的问题，一定使用绝对路径的方式

* 异步文件写入

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

* 同步文件写入

**fs.writeFileSync(file,data,[,option])**

* 流式文件写

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

默认写入操作是覆盖源文件

* 异步追加

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

* 同步追加

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

## 其他文件操作

* 验证路径是否存在（过时的API）

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

**fs.existsSync(path)** // => 返回布尔类型 exists

* 获取文件信息

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

**fs.statSync(path)** // => 返回一个fs.Stats实例

* 移动文件

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

* 重命名文件或目录

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

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

* 删除文件

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

**fs.unlinkSync(path)**

* 创建一个目录

**fs.mkdir(path[,model],callback)**

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

* 删除一个空目录

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

**fs.rmdirSync(path)**

* 读取一个目录

**fs.readdir(path,callback(err,files))**

**fs.readdirSync(path) // => 返回files**

* 监视文件变化：

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

**options:{persistent,interval}**

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

## 缓冲区处理（Buffer二进制数据）

什么是缓冲区？为什么要有缓冲区？缓冲区操作？

### 什么是缓冲区

缓冲区就是内存中操作数据的容器，只是数据容器而已，通过缓冲区可以很方便的操作二进制数据，而且在大文件操作时必须有缓冲区。

### 为什么要有缓冲区

JS 是比较擅长处理字符串，但是早期的应用场景主要用于处理 HTML 文档，不会有太大篇幅的数据处理，也不会接触到二进制的数据。而在 Node 中操作数据、网络通信是没办法完全以字符串的方式操作的。所以在 Node 中引入了一个二进制的缓冲区的实现：Buffer

### 创建缓冲区

例如：

* 创建长度为4个字节的缓冲区

**var buffer = new Buffer(4);**

* 通过指定数组内容的方式创建

**var buffer = new Buffer([00,01]);**

* 通过指定编码的方式创建

**var buffer = new Buffer('hello', 'utf8');**

### Node默认支持的编码

Buffers 和 JavaScript 字符串对象之间转换时需要一个明确的编码方法。下面是字符串的不同编码。

* **'ascii'** - 7位的 ASCII 数据。这种编码方式非常快，它会移除最高位内容。
* **'utf8'** - 多字节编码 Unicode 字符。大部分网页和文档使用这类编码方式。
* **'utf16le'** - 2个或4个字节, Little Endian (LE) 编码 Unicode 字符。编码范围 (U+10000 到 U+10FFFF) 。
* **'ucs2'** - 'utf16le'的子集。
* **'base64'** - Base64 字符编码。
* **'binary'** - 仅使用每个字符的头8位将原始的二进制信息进行编码。在需使用 Buffer 的情况下，应该尽量避免使用这个已经过时的编码方式。这个编码方式将会在未来某个版本中弃用。
* **'hex'** - 每个字节都采用 2 进制编码。

## 文件流

### 文件操作进阶

抽象具体化理解：

问题: 两个大水桶，需要从一个桶倒入另外一个桶？

* 直接把有水的桶中的水直接倒入另一个桶————读取文件
* 从有水的桶中一点一点的舀到另一个水桶中————缓冲区一点点读取
* 管道 ————文件流

### 什么是流

现实过程的流：

人流、水流

在程序开发中的概念：

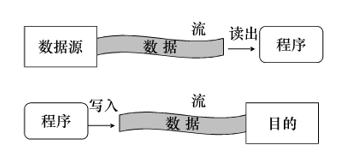
* 流是程序输入或输出的一个连续的字节序列
* 文件流、网络流
* 设备的输入输出都是用流来处理的

### 什么是文件流

在node的文档中经常见到Stream（流），任何数据的最根本，表现形式都二进制



文件流就是以面向对象的概念对文件数据进行的抽象，文件流定义了一些对文件数据的操作方式。



### Node中文件流的操作

在Node核心模块fs中定义了Stream的API

http://nodejs.cn/api/stream.html

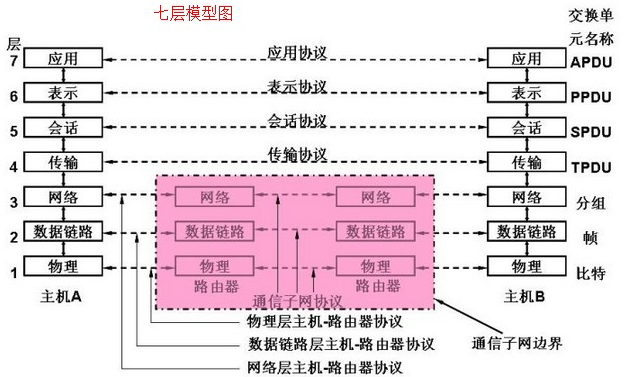
# 九、网络操作

学习 Node 内部与网络操作相关的模块

nodejs最重要的方面之一是具有非常迅速的实现HTTP和HTTPS服务器和服务的能力。http服务是相当低层次的，你可能要用到不同的模块，如express来实现完整的Web服务器，http模块不提供处理路由、cookie、缓存等的调用。我们主要用http模块的地方是实现供应用程序使用的后端Web服务。

## HTTP基础知识

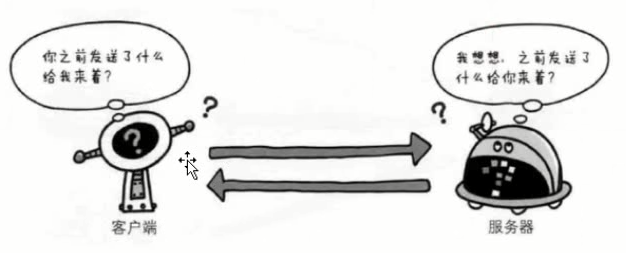
OSI参考模型：



### 什么是HTTP

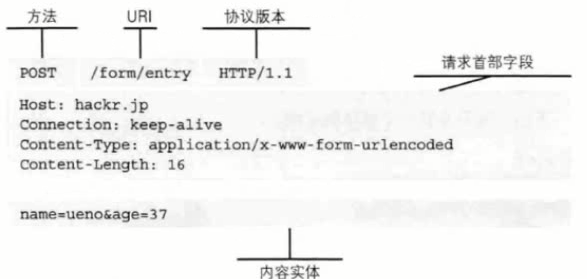
超文本传输协议（HTTP，HyperText Transfer Protocol)是互联网上应用最为广泛的一种计算机与计算机通信的网络协议。

使用HTTP协议，每当有新的请求是，就会有对应的新响应产生洗衣本身并不保留之前的一切请求或响应报文的信息，这是为了更快处理大量事务，确保协议的可伸缩性，而特意把HTTP协议设计成如此简单

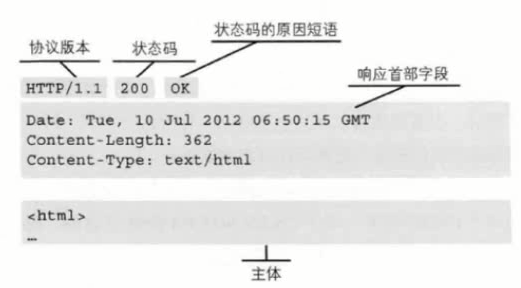


**HTTP中常见的名称：**

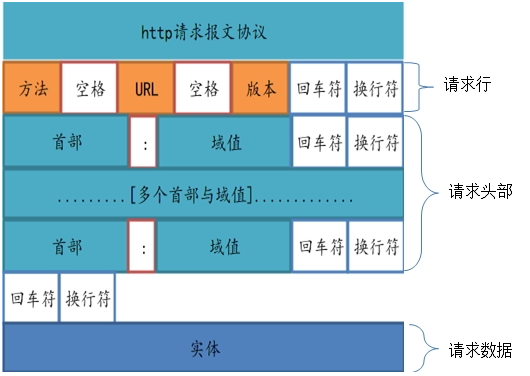
* **Request Message**（请求报文） -header -body



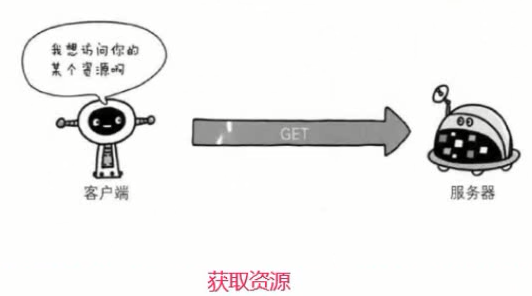
* **Response Message**（响应报文） -header -body



**HTTP请求报文：**



* 常见的方法字段：

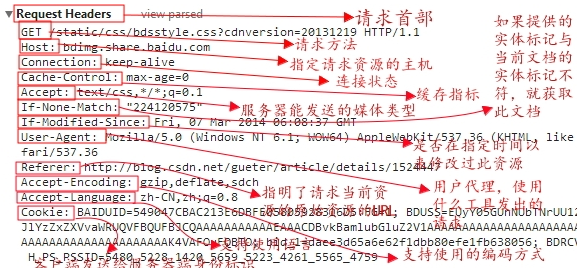
①**GET**:请求获取Request-URI(URI:通用资源标识符,URL是其子集，URI注重的是标识，而URL强调的是位置，可以将URL看成原始的URI),所标识的资源  


②**POST**：在Request-URI所标识的资源后附加新的数据；支持HTML表单提交，表单中有用户添入的数据，这些数据会发送到服务器端，由服务器存储至某位置（例如发送处理程序）

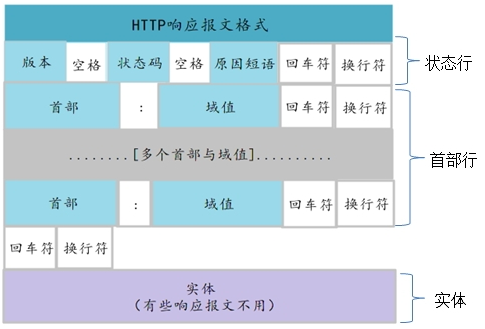


* URL：

统一资源定位符（URL），一个完整的包括类型、主机名和可选路径名的统一资源引用名



**HTTP请求报文：**



HTTP响应报文同样也分为三部分，有状态行、首部行、实体

状态行:HTTP响应报文的第一行

状态行包括三个字段：协议版本、状态码与原因短语。

* 状态码：

**1xx**

这一类型的状态码，代表请求已被接受，需要继续处理。这类响应是临时响应，只包含状态行和某些可选的响应头信息，并以空行结束。

**2xx**

这一类型的状态码，代表请求已成功被服务器接收、理解、并接受。

**3xx**

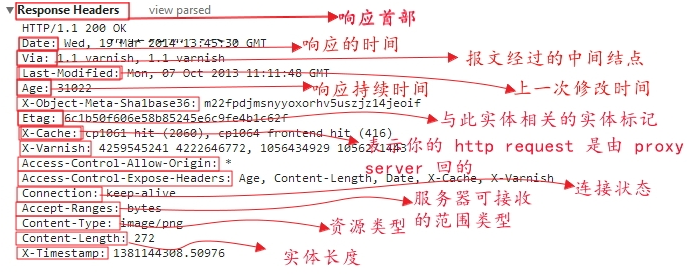
这类状态码代表需要客户端采取进一步的操作才能完成请求。通常，这些状态码用来重定向，后续的请求地址（重定向目标）在本次响应的Location域中指明。

**4xx**

这类的状态码代表客户端类的错误

**5xx**

服务器类的错误



### 浏览器的本质作用

•将用户在地址栏中输入的URL地址封装为一个请求消 息（包含请求头和请求体字符串）

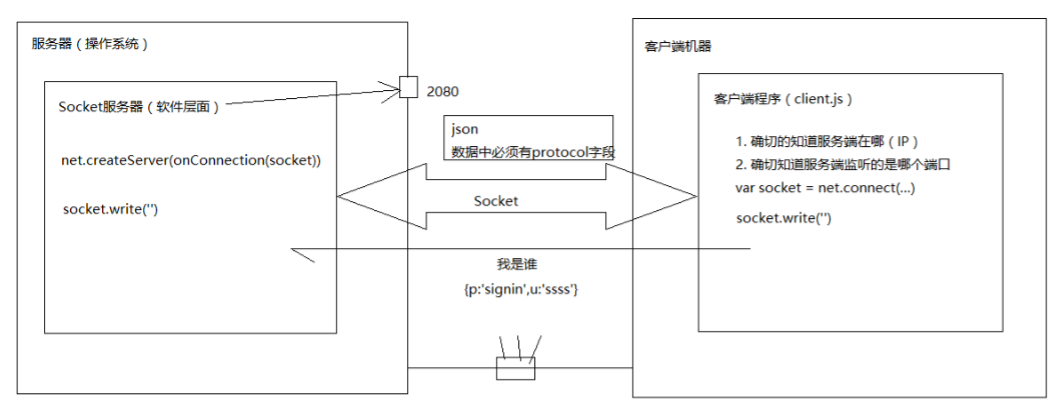
• 将得到的请求消息通过Socket的方式发送到所请求的网站服务器

•接收服务器返回的响应消息（包含响应头和响应体的字 符串）

•解析响应消息中的数据，渲染到界面上

### 什么是socket

网络上的两个程序通过一个双向的通信连接实现数据的交换，这个连接的一端称为一个socket。



### Node.js的net模块

Node.js中提供了net模块，该模块提供了对TCP、Socket的封装与支持，它包含了创建TCP服务器/客户端的方法。net模块继承自events和stream模块，所以该模块创建的服务器/客户端也是一个事件发射器，而其创建的客户端socket套接字对象又是一个可读写的Stream。net模块网络操作的基础模块，Node.js中其它网络操作相关模块，如：Http模块等，都是基于net模块的进一步封装。

### HTTP/1.0和HTTP/1.1支持的方法



常用的就是**增**、**删**、**改**、**查**方法

### 客户端与服务端



