

# Node.js 简介

---

## Node.js 是什么

官方主页上有一段解释

"Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices."

我们可以注意到: Node.js 是一个平台, 它构建在 Chrome 的 JavaScript 引擎(也就是著名的 V8 引擎)之上, 能快速构建具有扩展性的网络应用程序。这里官方用的 "网络应用程序", 整个描述没有提到 "web", "server" 等等概念。

举个例子, 类比来说, 概念上, Node.js 相当于 .net, JVM 或者 PythonVM, RubyVM。它是一个运行平台, 只不过它运行的是 javascript 语言而已。类似地, .net 一般运行 C#, VB 等编译过后的 IL, JVM 一般运行 Java 编译成的字节码, 而 PythonVM 解释运行 Python 代码, RubyVM 解释运行 Ruby 代码。严格来说 Node.js 更像 PythonVM, RubyVM, 因为他们都是解释运行的动态语言。

回到刚才的那段话中, 有几个是重点:

1. node.js 是一个构建在 Chrome JavaScript 运行环境的平台, 这是很重要的一点, node.js 并不是一门语言, 而是一个平台
2. node.js 致力于使构建速度快、稳定的网络程序更简单
3. node.js 具有事件驱动和非阻塞 I/O 的特色, 使之轻量级并且高效率
4. node.js 非常适合在分布式设备运行数据密集型实时应用程序

如果你能把握住这两点, 我觉得就算掌握了 Node.js 的真谛了。

Node.js 核心主要是由两部分组成的:

1. v8 引擎, 它负责把 javascript 代码解释成本地的二进制代码运行。
2. libuv, 类似 windows 上的窗口消息机制, 它主要负责订阅和处理系统的各种内核消息。而且它也实现了消息循环(是不是很耳熟? 没错, 这个几乎就和 windows 的窗口消息循环是一个概念)。它的前身是 linux 上的 libev, 专门封装 linux 上的内核消息机制。后来 Node.js 重写了它, 并在 windows 上使用 iocp 技术重新实现了一遍。所以 Nodejs 现在能跨平台运行在 windows 上了。

可以说 Node.js 其实就是 libuv 的一个应用而已。简单的说 Node.js 只是把 javascript 解释成 C++ 的回调, 并挂在 libuv 消息循环上, 等待处理。这样就实现了非阻塞的异步处理机制。

那么为什么是 javascript 而不是其他的语言。很简单, 因为 javascript 的闭包。这非常适合做回调函数。因为我们一般都希望当回调发生时, 闭包能记住它原来所在的上下文。这就是闭包最好的应用场景。

## Why Node.js?

Node.js 出现确实能为我们解决现实当中系统瓶颈提供了新的思路和方案, 下面我们看看它能解决什么问题。

### 并发连接

举个例子, 想象一个场景, 我们在银行排队办理业务, 我们看看下面两个模型。

#### (1) 系统线程模型:

这种模型的问题显而易见, 服务端只有一个线程, 并发请求(用户)到达只能处理一个, 其余的要先等待, 这就是阻塞, 正在享受服务的请求阻塞后面的请求了。

#### (2) 多线程、线程池模型:

这个模型已经比上一个有所进步, 它调节服务端线程的数量来提高对并发请求的接收和响应, 但并发量高的时候, 请求仍然需要等待, 它有个更严重的问题。到代码层面上来讲, 我们看看客户端请求与服务端通讯的过程: 服务端与客户端每建立一个连接, 都要为这个连接分配一套配套的资源, 主要体现为系统内存资源, 以 PHP 为例, 维护一个连接可能需要 20M 的内存。这就是为什么一般并发量大, 就需要多开服务器。那么 NodeJS 是怎么解决这个问题的呢? 我们来看另外一个模型, 想象一下我们在快餐店点餐吃饭的场景。

#### (3) 异步、事件驱动模型

我们同样是要发起请求，等待服务器端响应；但是与银行例子不同的是，这次我们点完餐后拿到了一个号码，拿到号码，我们往往会在位置上等待，而在我们后面的请求会继续得到处理，同样是拿了一个号码然后到一旁等待，接待员能一直进行处理。等到饭菜做号了，会喊号码，我们拿到了自己的饭菜，进行后续的处理（吃饭）。这个喊号码的动作在 **NodeJS** 中叫做回调（**Callback**），能在事件（烧菜，**I/O**）处理完成后继续执行后面的逻辑（吃饭），这体现了 **NodeJS** 的显著特点，异步机制、事件驱动整个过程没有阻塞新用户的连接（点餐），也不需要维护已经点餐的用户与厨师的连接。基于这样的机制，理论上陆续有用户请求连接，**NodeJS** 都可以进行响应，因此 **NodeJS** 能支持比 **Java**、**PHP** 程序更高的并发量虽然维护事件队列也需要成本，再由于 **NodeJS** 是单线程，事件队列越长，得到响应的时间就越长，并发量上去还是会力不从心。总结一下 **NodeJS** 是怎么解决并发连接这个问题的：更改连接到服务器的方式，每个连接发射（emit）一个在 **NodeJS** 引擎进程中运行的事件（**Event**），放进事件队列当中，而不是为每个连接生成一个新的 **OS** 线程（并为其分配一些配套内存）。

### 1. I/O 阻塞

**NodeJS** 解决的另外一个问题是 **I/O** 阻塞，看看这样的业务场景：需要从多个数据源拉取数据，然后进行处理。

1. 串行获取数据，这是我们一般的解决方案，以 **PHP** 为例 **I/O** 阻塞 -**PHP** 为例 假如获取 **profile** 和 **timeline** 操作各需要 **1S**，那么串行获取就需要 **2S**。

### 2. NodeJS 非阻塞 I/O，发射 / 监听事件来控制执行过程 非 I/O 阻塞 -**PHP** 为例

**NodeJS** 遇到 **I/O** 事件会创建一个线程去执行，然后主线程会继续往下执行的，因此，拿 **profile** 的动作触发一个 **I/O** 事件，马上就会执行拿 **timeline** 的动作，两个动作并行执行，假如各需要 **1S**，那么总的时间也就是 **1S**。它们的 **I/O** 操作执行完成后，发射一个事件，**profile** 和 **timeline**，事件代理接收后继续往下执行后面的逻辑，这就是 **NodeJS** 非阻塞 **I/O** 的特点。

总结一下：**Java**、**PHP** 也有办法实现并行请求（子线程），但 **NodeJS** 通过回调函数（**Callback**）和异步机制会做得很自然。

## NodeJS 的优缺点

优点：

1. 高并发（最重要的优点）
2. 适合 **I/O** 密集型应用

缺点：

1. 不适合 **CPU** 密集型应用；**CPU** 密集型应用给 **Node** 带来的挑战主要是：由于 **JavaScript** 单线程的原因，如果有长时间运行的计算（比如大循环），将会导致 **CPU** 时间片不能释放，使得后续 **I/O** 无法发起； 解决方案：分解大型运算任务为多个小任务，使得运算能够适时释放，不阻塞 **I/O** 调用的发起；
2. 只支持单核 **CPU**，不能充分利用 **CPU**
3. 可靠性低，一旦代码某个环节崩溃，整个系统都崩溃 其中 2 和 3 的原因：单进程，单线程 解决方案：
  - i. **Nnigx** 反向代理，负载均衡，开多个进程，绑定多个端口；
  - ii. 开多个进程监听同一个端口，使用 **cluster** 模块；
4. 开源组件库质量参差不齐，更新快，向下不兼容
5. **Debug** 不方便，错误没有 **stack trace**

## Node.js 安装

---

### Windows 环境

直接在官方 下载 **msi** 安装包一步步按照提示安装即可，建议安装 **LTS**(Long Time Support) 版本

### MacOS 环境

基本同 **Widows**，在官方 下载 **pkg** 安装包一步步按照提示安装即可，建议安装 **LTS**(Long Time Support) 版本

### Linux 环境

**Linux** 下安装 **Node.js** 环境要稍微复杂点，可以下载 **Source Code** 本地编译安装，也可以下载 **Binaries** 直接解压安装即可

以下以 **Linux Binaries** 为例安装 **Node.js**

LTS

Recommended For Most Users

Current

Latest Features

Windows Installer

node-v4.5.0-x64.msi


Macintosh Installer

node-v4.5.0.pkg

Source Code

node-v4.5.0.tar.gz

Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.exe)	32-bit	64-bit
Mac OS X Installer (.pkg)	64-bit	
Mac OS X Binaries (.tar.gz)	64-bit	
Linux Binaries (.tar.xz)	32-bit	64-bit
Source Code	node-v4.5.0.tar.gz	

- 下载 Linux Binaries （根据自己的系统下载 32 位或是 64 位版本，如果不知道自己系统版本请使用 32 位版本）
- 解压下载文件 `xz -d xxxx` 其中 `xxxx` 为你下载文件名，以 `.tar.xz` 结尾
- 解包 `tar -xvf xxxx.tar` 其中 `xxxx` 为你上一步解压后的文件名，以 `.tar` 结尾
- 移动文件夹 `sudo mkdir /usr/local/lib/node && mv xxxx /usr/local/lib/node` 其中 `xxxx` 为上一步解压后的 **Node** 文件夹名
- 添加环境变量（这里以 **Ubuntu** 为例，其他系统请自行百度）`sudo vim /etc/environment` 添加一行  
`NODE_HOME=/usr/local/lib/node/xxxx` 并在 `PATH=...` 后添加 `:/usr/local/lib/node/xxxx/bin`
- 使环境变量修改生效 `source /etc/environment`
- 验证安装 `node -v` `npm -v` 应该显示版本号 

## Node.js Hello World

### 命令行交互下的 Hello World

命令行下运行

```
node
```

进入 Node.js 交互

输入

```
console.log('hello World');
```

回车即可运行



### 写在源文件里的 Hello world

新建文件 `hello-world.js`

以文本方式打开，在其中输入

```
console.log('hello World');
```

保存，使用命令 `node hello-world.js` 运行即可



## 第一个 Node.js Web Server

Hello World 程序很简单，下面上点“干货”，使用 node 创建一个简单的 http 服务器。

通过查阅 Node.js 官方 [api 文档](#) 我们可以很快的发现 http 模块，这个模块可以发起 http 请求，也可以创建一个 http server。

## 引入 required 模块

我们使用 require 指令来载入 http 模块，并将实例化的 HTTP 赋值给变量 http，实例如下：

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

## 创建服务器

接下来我们使用 http.createServer() 方法创建服务器，并使用 listen() 方法绑定 8888 端口。函数通过 request, response 参数来接收和响应数据。

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

http.createServer(function (request, response) {

    // 发送 HTTP 头部
    // HTTP 状态值: 200 : OK
    // 内容类型: text/plain
    response.writeHead(200, {'Content-Type': 'text/plain'});

    // 发送响应数据 "Hello World"
    response.end('Hello World\n');

}).listen(8888);

// 终端打印如下信息
console.log('Server running at http://127.0.0.1:8888/');
```

分析：

- 第一行请求（require）Node.js 自带的 http 模块，并且把它赋值给 http 变量。
- 接下来我们调用 http 模块提供的函数： createServer() 。这个函数会返回一个对象，这个对象有一个叫做 listen() 的方法，这个方法有一个数值参数， 指定这个 HTTP 服务器监听的端口号。

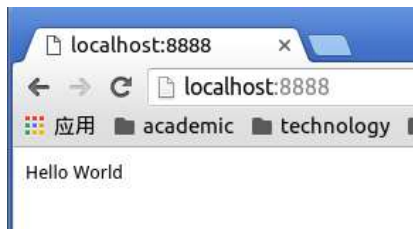
## 测试结果

以上代码我们完成了一个可以工作的 HTTP 服务器。使用 node 命令执行以上的代码：

```
node server.js
```



接下来，打开浏览器访问 <http://127.0.0.1:8888/>，你会看到一个写着 "Hello World" 的网页。



## Node.js 回调函数

Node.js 异步编程的直接体现就是回调。

异步编程依托于回调来实现，但不能说使用了回调后程序就异步化了。

回调函数在完成任务后就会被调用，Node 使用了大量的回调函数，Node 所有 API 都支持回调函数。

例如，我们可以一边读取文件，一边执行其他命令，在文件读取完成后，我们将文件内容作为回调函数的参数返回。这样在执行代码时就没有阻塞或等待文件 I/O 操作。这就大大提高了 Node.js 的性能，可以处理大量的并发请求。

## 阻塞代码实例

创建一个文件 `input.txt`，内容如下：

```
这是一个文本文件
```

创建 `main.js` 文件, 代码如下：

```
var fs = require("fs");

var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("程序执行结束!");
````
```

以上代码执行结果如下：

```
$ node main.js 这是一个文本文件
```

```
程序执行结束! ````
```

## 非阻塞代码实例

修改 `main.js` 文件, 代码如下：

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
  if (err) return console.error(err);
  console.log(data.toString());
});

console.log("程序执行结束!");
```

以上代码执行结果如下：

```
$ node main.js
程序执行结束!
这是一个文本文件
```

以上两个实例我们了解了阻塞与非阻塞调用的不同。第一个实例在文件读取完后才执行完程序。第二个实例我们呢不需要等待文件读取完，这样就可以在读取文件时同时执行接下来的代码，大大提高了程序的性能。

因此，阻塞是按顺序执行的，而非阻塞是不需要按顺序的，所以如果需要处理回调函数的参数，我们就需要写在回调函数内。