[搭建node.js环境 2](#_Toc375160709)

[运行node.js 3](#_Toc375160710)

[安装模块 6](#_Toc375160711)

[基本介绍 9](#_Toc375160712)

[特点 9](#_Toc375160713)

[能解决的问题 10](#_Toc375160714)

[优缺点 15](#_Toc375160715)

[优点 15](#_Toc375160716)

[缺点 15](#_Toc375160717)

[性能分析 16](#_Toc375160718)

[适用场景 23](#_Toc375160719)

[适用的具体场景举例 23](#_Toc375160720)

[不适用的具体场景举例 27](#_Toc375160721)

[与传统服务器进行比较 28](#_Toc375160722)

[实际应用中的差异 28](#_Toc375160723)

[工作原理对比 28](#_Toc375160724)

[性能对比 30](#_Toc375160725)

[总结 32](#_Toc375160726)

[在游戏开发中的应用 32](#_Toc375160727)

[参考资料 32](#_Toc375160728)

[基于Node.js的游戏服务器端框架Pomelo 32](#_Toc375160729)

[参考资料 32](#_Toc375160730)

[框架 33](#_Toc375160731)

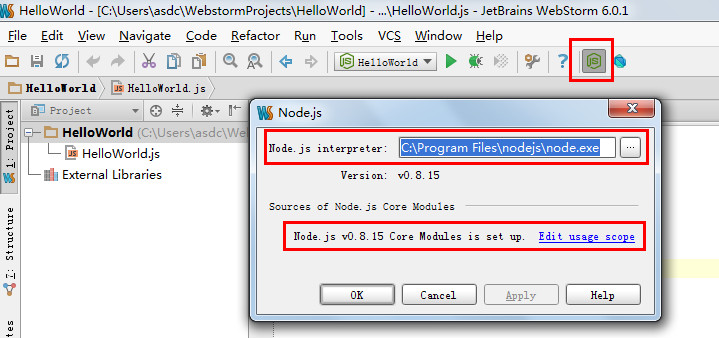
[Express.js 33](#_Toc375160732)

# 搭建node.js环境

    1.根据个人情况[下载](http://nodejs.org/download/)不同版本的安装包，并安装好Node.js环境。安装步骤很简单，不需要配置和修改什么，一切都走默认路线即可。

      2.在安装完Node.js并没有什么异常提示之后，在cmd下，进入nodeJs安装目录，敲入 node 命令，测试一下，看看node.js是否安装成功。

      3.安装WebStorm软件，安装成功之后，需要在其中配置一下Node.js：如下图

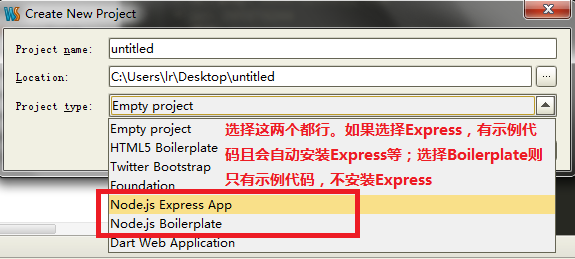


Sources of Node.js Core Modules: 指的是配置源文件路径，把Node.js源码解压后的路径配置到此即可。

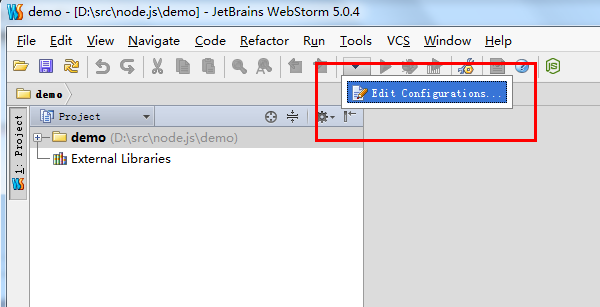
我配置的是：D:\ToolInstall\nodeJs\node\_modules\npm

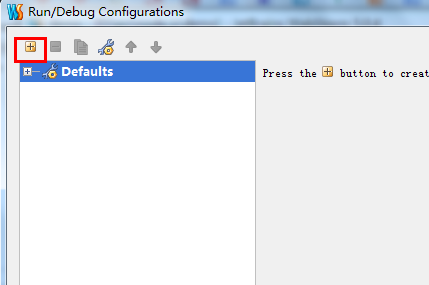
## 运行node.js

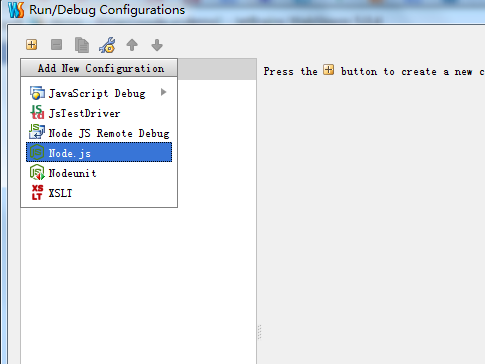
1. 创建一个Node.js项目（如果不是Node.js项目，也可以运行，但没有智能提示）。

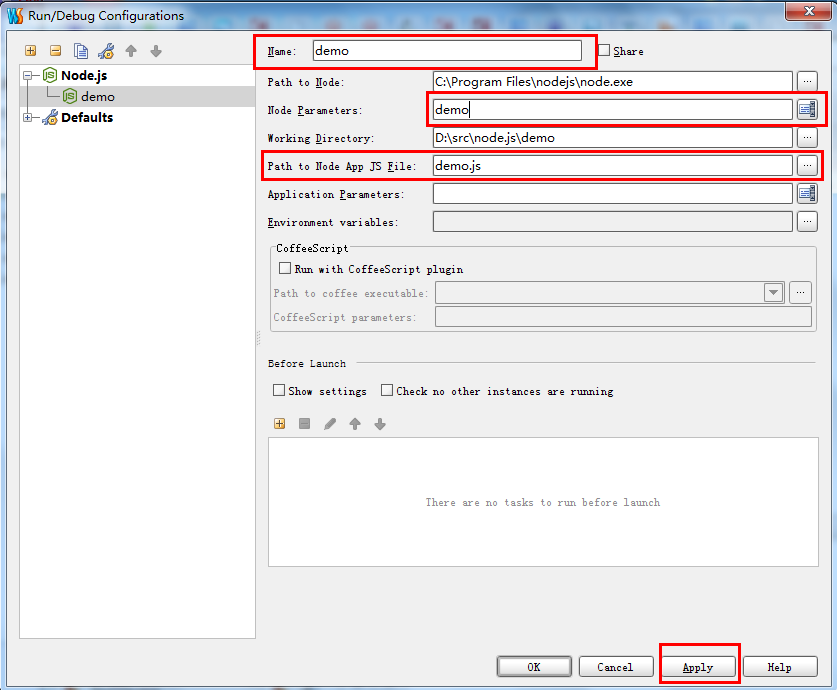


1. 配置编译环境





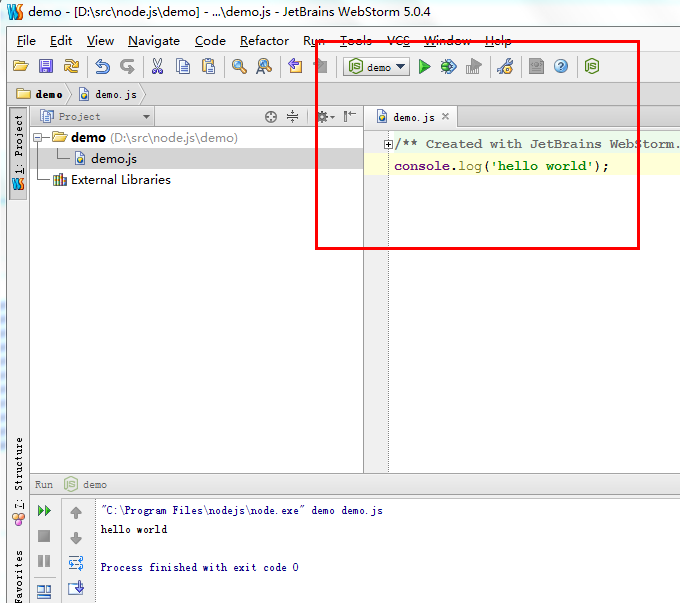




Working Directory为node.js文件所在目录，需要选择。

Path to Node App Js File：准备运行的node.js文件，需要选择

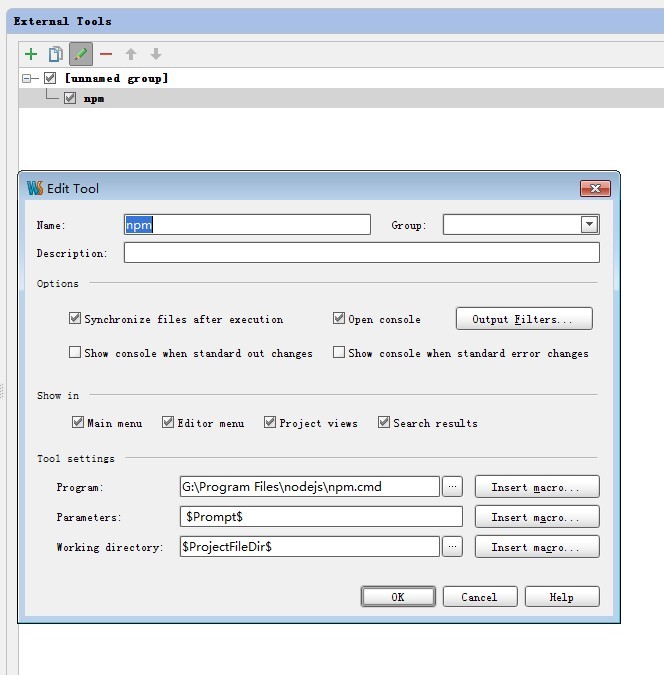
1. 运行node.js工程

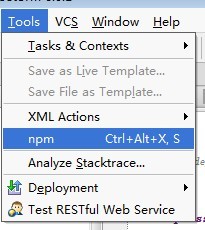
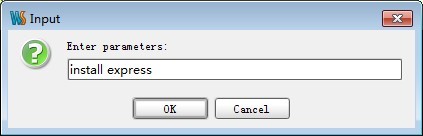


点击箭头执行。

## 安装模块

1. 添加自定义的工具npm

ctrl+alt+s打开**setting**菜单，找到**external tools**  
点+号，在tool setting里填空：  
program -> 外部命令所在位置，填入npm的完整路径  
parameters -> 参数，这里设为 $Prompt$运行命令时表示弹个窗口让你输入  
Working Directory -> 在哪个目录下运行这条命令，输入$ProjectFileDir$表示在当前项目的根目录下运行。  
点右边的insert macro可以看到更多的选项，除了macro的说明，webstorm还会告诉你对应macro在当前环境下是个什么值的，不怕看不懂e文了。[](http://ww2.sinaimg.cn/large/7327f6d5gw1e4sips6984j20ig0irtag.jpg)  
最后给这个命令起个名字，叫npm保存完事。

1. 在顶部菜单找到tools就可以看到刚保存的npm。点击它就会弹出个窗口，输入install express试试吧。  
   [](http://ww2.sinaimg.cn/large/7327f6d5gw1e4siqh4em2j205p06ewep.jpg)  
   [](http://ww4.sinaimg.cn/large/7327f6d5gw1e4siqz9rkgj20br03s0st.jpg)

# 基本介绍

Node.js不是JS应用、而是JS运行平台

Node.js采用了Google Chrome浏览器的V8引擎，性能很好，同时还提供了很多系统级的API，如文件操作、网络编程等。浏览器端的Javascript代码在运行时会受到各种安全性的限制，对客户系统的操作有限。相比之下，Node.js则是一个全面的后台运行时，为Javascript提供了其他语言能够实现的许多功能。

# 特点

1. 它是一个Javascript运行环境
2. 依赖于Chrome V8引擎进行代码解释
3. 事件驱动
4. 非阻塞I/O
5. 轻量、可伸缩，适于实时数据交互应用
6. 单进程，单线程
7. 内存占用率低，CPU占用率高

**Node.js采用事件驱动、异步编程，为网络服务而设计**

事件驱动这个词并不陌生，在某些传统语言的网络编程中，我们会用到回调函数，比如当socket资源达到某种状态时，注册的回调函数就会执行。Node.js的设计思想中以事件驱动为核心，它提供的绝大多数API都是基于事件的、异步的风格。以Net模块为例，其中的net.Socket对象就有以下事件：connect、data、end、timeout、drain、error、close等，使用Node.js的开发人员需要根据自己的业务逻辑注册相应的回调函数。这些回调函数都是异步执行的，这意味着虽然在代码结构中，这些函数看似是依次注册的，但是它们并不依赖于自身出现的顺序，而是等待相应的事件触发。事件驱动、异步编程的设计（感兴趣的读者可以查阅笔者的另一篇文章《[Node.js的异步编程风格](http://www.infoq.com/cn/news/2011/09/nodejs-async-code)》），重要的优势在于，充分利用了系统资源，执行代码无须阻塞等待某种操作完成，有限的资源可以用于其他的任务。此类设计非常适合于后端的网络服务编程，Node.js的目标也在于此。在服务器开发中，并发的请求处理是个大问题，阻塞式的函数会导致资源浪费和时间延迟。通过事件注册、异步函数，开发人员可以提高资源的利用率，性能也会改善。

从Node.js提供的支持模块中，我们可以看到包括文件操作在内的许多函数都是异步执行的，这和传统语言存在区别，而且为了方便服务器开发，Node.js的网络模块特别多，包括HTTP、DNS、NET、UDP、HTTPS、TLS等，开发人员可以在此基础上快速构建Web服务器。以简单的helloworld.js为例：

var http = require('http');

http.createServer(function (req, res) {

res.writeHead(200, {'Content-Type': 'text/plain'});

res.end('Hello World\n');

}).listen(80, "127.0.0.1");

上面的代码搭建了一个简单的http服务器（运行示例部署在<http://helloworld.cnodejs.net/>中，读者可以访问），在本地监听80端口，对于任意的http请求，服务器都返回一个头部状态码为200、Content-Type'值为text/plain'的”Hello World“文字响应。从这个小例子中，我们可以看出几点：

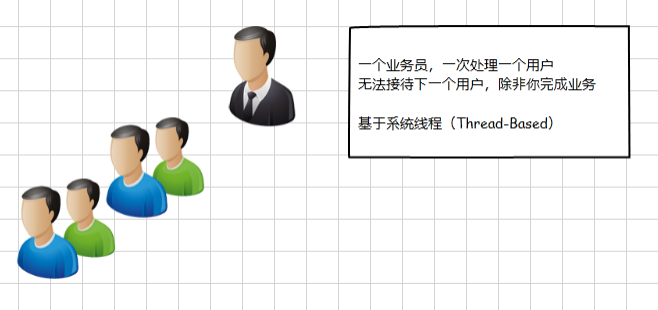
* Node.js的网络编程比较便利，提供的模块（在这里是http）开放了容易上手的API接口，短短几行代码就可以构建服务器。
* 体现了事件驱动、异步编程，在createServer函数的参数中指定了一个回调函数（采用Javascript的匿名函数实现），当有http请求发送过来时，Node.js就会调用该回调函数来处理请求并响应。当然，这个例子相对简单，没有太多的事件注册，在以后的文章中读者会看到更多的实际例子。

# 能解决的问题

1. 并发连接

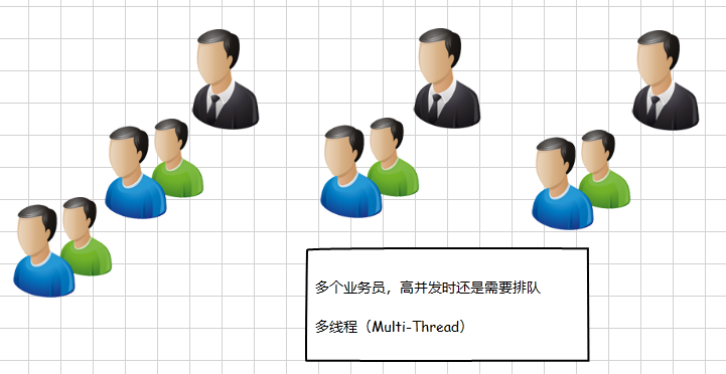
　　举个例子，想象一个场景，我们在银行排队办理业务，我们看看下面两个模型

　　（1）系统线程模型：



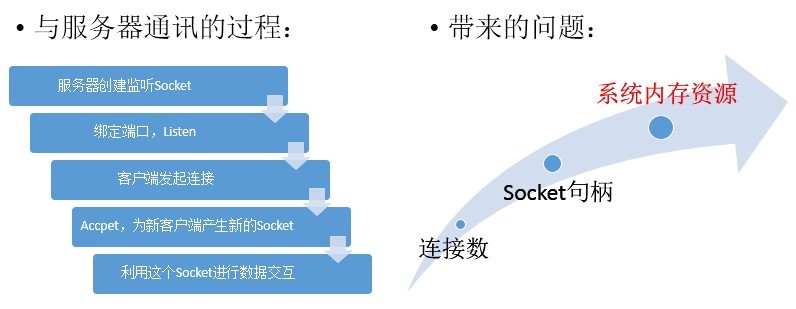
　　这种模型的问题显而易见，服务端只有一个线程，并发请求（用户）到达只能处理一个，其余的要先等待，这就是阻塞，正在享受服务的请求阻塞后面的请求了

　　（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：

　　回到代码层面上来讲，我们看看客户端请求与服务端通讯的过程：



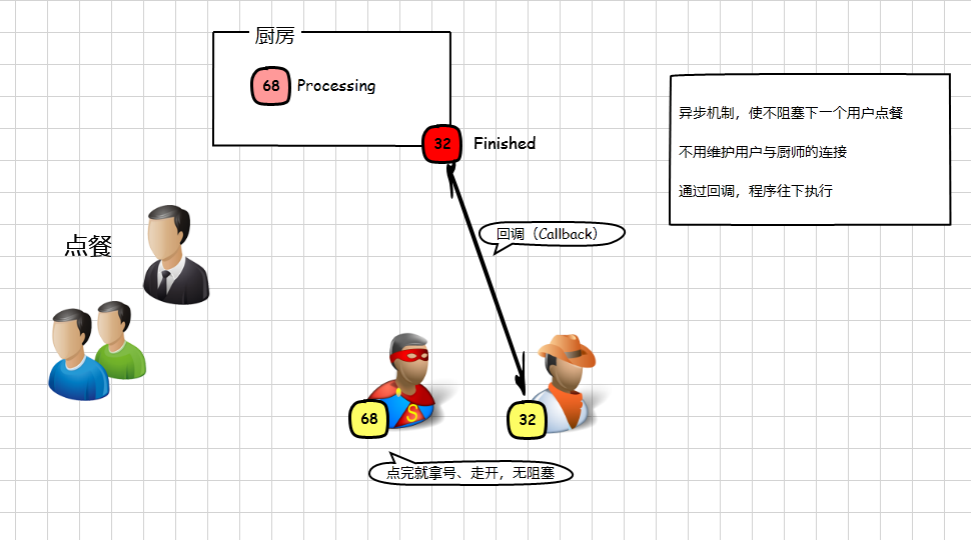
　　服务端与客户端每建立一个连接，都要为这个连接分配一套配套的资源，主要体现为系统内存资源，以PHP为例，维护一个连接可能需要20M的内存

　　这就是为什么一般并发量一大，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：

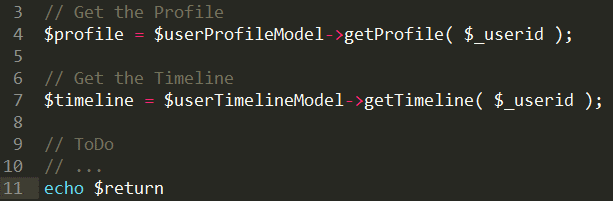
　　更改连接到服务器的方式，每个连接发射（emit）一个在NodeJS引擎进程中运行的事件（Event），放进事件队列当中，

　　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）

　　2. I/O阻塞

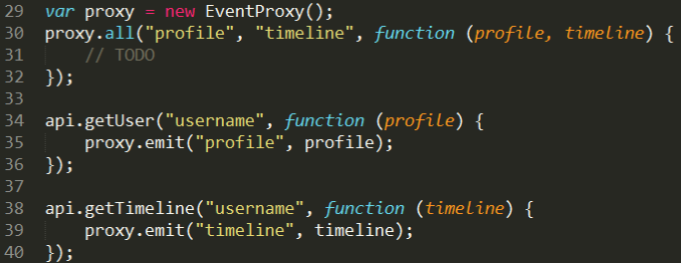
　　NodeJS解决的另外一个问题是I/O阻塞，看看这样的业务场景：需要从多个数据源拉取数据，然后进行处理

　　（1）串行获取数据，这是我们一般的解决方案，以PHP为例



　　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



　　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

# 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
3. 轻量

## 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；

1. 只支持单核CPU，不能充分利用CPU
2. 可靠性低，一旦代码某个环节崩溃，整个系统都崩溃

　　　　原因：单进程，单线程

　　　　解决方案：（1）Nnigx反向代理，负载均衡，开多个进程，绑定多个端口；

　　　　　　　　　（2）开多个进程监听同一个端口，使用cluster模块；

1. 开源组件库质量参差不齐，更新快，向下不兼容
2. Debug不方便，错误没有stack trace

# 性能分析

关于单线程的优化和无法充分利用多核CPU的讨论

**JavaScript单线程的误解**

在我接触JavaScript（无论浏览器还是NodeJS）的时间里，总是遇到有朋友有多线程的需求。而在NodeJS方面，有朋友甚至直接说到，NodeJS是单线程的，无法很好的利用多核CPU。

诚然，在前端的浏览器中，由于前端的JavaScript与UI占据同一线程，执行JavaScript确实为UI响应造成了一定程度上的麻烦。但是，除非用到超大的循环语句执行JavaScript，或是用阻塞式的Ajax，或是太过频繁的定时器执行外，JavaScript并没有给前端应用带来明显的问题，所以也很少有朋友抱怨JavaScript是单线程而不能很好利用多核CPU的问题，因为没有因此出现性能瓶颈。

但是，我们可以用Ajax和Web Worker回应这个误解。当Ajax请求发送之后，除非是同步请求，否则其余的JavaScript代码会很快被执行到。在Ajax发送完成，直到接收到响应的这段时间里，这个网络请求并不会阻塞JavaScript的执行，而网络请求已经发生，这是必然的事。那么，答案就很明显了，JavaScript确实是执行在单线程上的，但是，整个Web应用执行的宿主（浏览器）并非以单线程的方式在执行。而Web Worker的产生，就是直接为了解决JavaScript与UI占用同一线程造成的UI响应问题的，它能新开一条线程去执行JavaScript。

同理，NodeJS中的JavaScript也确实是在单线程上执行，但是作为宿主的NodeJS，它本身并非是单线程的，NodeJS在I/O方面有动用到一小部分额外的线程协助实现异步。程序员没有机会直接创建线程，这也是有的同学想当然的认为NodeJS的单线程无法很好的利用多核CPU的原因，他们甚至会说，难以想象由多人一起协作开发一个单线程的程序。

NodeJS 封装了内部的异步实现后，导致程序员无法直接操作线程，也就造成所有的业务逻辑运算都会丢到JavaScript的执行线程上，这也就意味着，在高并发请求的时候，I/O的问题是很好的解决了，但是所有的业务逻辑运算积少成多地都运行在JavaScript线程上，形成了一条拥挤的JavaScript运算线程。NodeJS的弱点在这个时候会暴露出来，单线程执行运算形成的瓶颈，拖慢了I/O的效率。这大概可以算得上是密集运算情况下无法很好利用多核 CPU的缺点。这条拥挤的JavaScript线程，给I/O形成了性能上限。

但是，事情又并非绝对的。回到前端浏览器中，为了解决线程拥挤的情况，Web Worker应运而生。而同样，Node也提供了child\_process.fork来创建Node的子进程。在一个Node进程就能很好的解决密集 I/O的情况下，fork出来的其余Node子进程可以当作常驻服务来解决运算阻塞的问题（将运算分发到多个Node子进程中上去，与Apache创建多个子进程类似）。当然child\_process/Web Worker的机制永远只能解决单台机器的问题，大的Web应用是不可能一台服务器就能完成所有的请求服务的。拜NodeJS在I/O上的优势，跨OS的多Node之间通信的是不算什么问题的。解决NodeJS的运算密集问题的答案其实也是非常简单的，就是将运算分发到多个CPU上。请参考文章后的multi-node的性能测试，可以看到在多Node进程的情景下，响应请求的速度被大幅度提高（感谢CNode社区的snoopy友情测试）。

在文章的写作过程中，Node最新发布的0.6.0版本，新增了cluster模块。该模块的作用是可以通过fork的方式创建出多个子进程实例，这些实例会自动共享相同的侦听端口。你可以根据当前计算机上的CPU数量来创建相应的实例数，以此达到分发请求，充分利用CPU的目的。详情请参阅[官方文档](http://http/nodejs.org/docs/v0.5.10/api/cluster.html)。在之前的解决运算密集问题中，工程师需要multi-node这样的库或者其他方案去手动分发请求，在cluster模块的支持下，可以释放掉工程师在解决此问题上的大部分精力。

**事件式编程**

延续上一节的讨论。我们知道NodeJS/JavaScript具有异步的特性，从NodeJS的API设计中可以看出来，任何涉及I/O的操作，几乎都被设计成事件回调的形式，且大多数的类都继承自EventEmitter。这么做的好处有两个，一个是充分利用无阻塞I/O的特性，提高性能；另一个好处则是封装了底层的线程细节，通过事件消息留出业务的关注点给编程者，从而不用关注多线程编程里牵扯到的诸多技术细节。

从现实的角度而言，事件式编程也更贴合现实。举一个业务场景为例：家庭主妇在家中准备中餐，她需要完成两道菜，一道拌黄瓜，一道西红柿蛋汤。以PHP为例，家庭主妇会先做完拌黄瓜，接着完成西红柿蛋汤，是以顺序/串行执行的。但是现在突然出了一点意外，凉拌黄瓜需要的酱油用光了，需要她儿子出门帮她买酱油回来。那么PHP家庭主妇在叫她儿子出门打酱油的这段时间都是属于等待状态的，直到酱油买回来，才会继续下一道菜的制作。那么，在NodeJS的家庭主妇又会是怎样一个场景呢，很明显，在等待儿子打酱油回来的时间里，她可以暂停凉拌黄瓜的制作，而直接进行西红柿蛋汤的过程，儿子打完酱油回来，继续完成她的凉拌黄瓜。没有浪费掉等待的时间。实例伪代码如下：

var mother = new People();

var child = new People();

child.buySoy(function (soy) {

mother.cook("cucumber", soy);

});

mother.cook("tomato");

接下来，将上面这段代码转换为基于事件/任务异步模式的代码：

var proxy = new EventProxy();

var mother = new People();

proxy.bind("cook\_cucumber", function (soy) {

mother.cook("cucumber", soy);

});

proxy.bind("cook\_tomato", function () {

mother.cook("tomato");

});

var child = new People();

child.buySoy(function (soy) {

proxy.trigger("cucumber", soy);

});

proxy.trigger("tomato");

代码量多了很多，但是业务逻辑点都是很清楚的：通过bind方法预定义了cook\_cucumber和cook\_tomato两个任务。这里的bind方法可以认为是await的消息式实现，需要第一个参数来标识该任务的名字，流程在执行的过程中产生的消息会触发这些任务执行。可以看出，事件式编程中，用户只需要关注它所需要的几个业务事件点就可以，中间的等待都由底层为你调配好了。这里的代码只是举例事件/任务异步模式而用，在简单的场景中，第一段代码即可。做NodeJS的编程，会更感觉是在做现实的业务场景设计和任务调度，没有顺序保证，程序结构更像是一个状态机。

个人觉得在事件式编程中，程序员需要转换一下思维，才能接受和发挥好这种异步/无阻塞的优势。同样，这种事件式编程带来的一个问题就在于业务逻辑是松散和碎片式的。这对习惯了顺序式，Promise式编程的同学而言，接受它是比较痛苦的事情，而且这种散布的业务逻辑对于非一开始就清楚设计的人而言，阅读存在相当大的问题。

我提到事件式编程更贴近于现实生活，是更自然的，所以这种编程风格也导致你的代码跟你的生活一样，是一件复杂的事情。幸运的是，自己的生活要自己去面对，对于一个项目而言，并不需要每个人都去设计整个大业务逻辑，对于架构师而言，业务逻辑是明了的，借助事件式编程带来的业务逻辑松耦合的好处，在设定大框架后，将业务逻辑划分为适当的粒度，对每一个实现业务点的程序员而言，并没有这个痛苦存在。二八原则在这个地方非常有效。

**深度嵌套回调问题**

JavaScript/NodeJS 对单个异步事件的处理十分容易，但容易出现问题出现的地方是“多个异步事件之间的结果协作”。以NodeJS服务端渲染页面为例，渲染需要数据，模板，本地化资源文件，这三个部分都是要通过异步来获取的，原生代码的写法会导致嵌套，因为只有这样才能保证渲染的时候数据，模板，本地化资源都已经获取到了。但问题是，这三个步骤之间实际是无耦合的，却因为原生代码没有promise的机制，将可以并行执行（充分利用无阻塞I/O）的步骤，变成串行执行的过程，直接降低了性能。代码如下：

var render = function (template, data) {

\_.template(template, data);

};

$.get("template", function (template) { // something

$.get("data", function (data) { // something

$.get("l10n", function (l10n) { // something

render(template, data);

});

});

});

面对这样的代码，许多工程师都表示不爽。这个弱点也形成了对NodeJS推广的一个不大不小的障碍。对于追求性能和维护性的同学，肯定不满足于以上的做法。本人对于JavaScript的事件和回调都略有偏爱，并且认为事件，回调，并行，松耦合是可以达成一致的。以下一段代码是用[EventProxy](http://https/github.com/JacksonTian/eventproxy)实现的：

var proxy = new EventProxy();

var render = function (template, data, l10n) {

\_.template(template, data);

};

proxy.assign("template", "data", "l10n", render);

$.get("template", function (template) { // something

proxy.trigger("template", template);

});

$.get("data", function (data) { // something

proxy.trigger("data", data);

});

$.get("l10n", function (l10n) { // something

proxy.trigger("l10n", l10n);

});

代码量看起来比原生实现略多，但是从逻辑而言十分清晰。模板、数据、本地化资源并行获取，性能上的提高不言而喻，assign方法充分利用了事件机制来保证最终结果的正确性。在事件，回调，并行，松耦合几个点上都达到期望的要求。

关于更多EventProxy的细节可参考其[官方页面](http://https/github.com/JacksonTian/eventproxy)。

**深度回调问题的延伸**

EventProxy解决深度回调的方式完全基于事件机制，这需要建立在事件式编程的认同上，那么必然也存在对事件式编程不认同的同学，而且习惯顺序式，promise式，向其推广bind/trigger模式实在难以被他们接受。[Jscex](http://https/github.com/JeffreyZhao/jscex)和[Streamline.js](http://https/github.com/Sage/streamlinejs%E2%80%8B)是目前比较成熟的同步式编程的解决方案。可以通过同步式的思维来进行编程，最终执行的代码是通过编译后的目标代码，以此通过工具来协助用户转变思维。

**结语**

对于优秀的东西，我们不能因为其表面的瑕疵而弃之不用，总会有折衷的方案来满足需求。NodeJS在实时性方面的功效有目共睹，即便会有一些明显的缺点，但是随着一些解决方案的出现，相信没有什么可以挡住其前进的脚步。

**附录（多核环境下的并发测试）**

服务器环境：

* 网络环境：内网
* 压力测试服务器：
* 服务器系统：Linux 2.6.18
* 服务器配置：Intel(R) Xeon(TM) CPU 3.40GHz 4 CPUS
* 内存：6GB
* NodeJS版本: v0.4.12

客户端测试环境：

* 发包工具：apache 2.2.19自带的ab测试工具
* 服务器系统：Linux 2.6.18
* 服务器配置：Pentium(R) Dual-Core CPU E5800 @ 3.20GHz 2CPUS
* 内存：1GB

单线程Node代码：

var http = require('http');

var server = http.createServer(function (request, response) {

var j = 0;

for (var i = 0; i & lt; 100000; i++) {

j += 2 / 3;

}

response.end(j + '');

});

server.listen(8881);

console.log('Server running at http://10.1.10.150:8881/');

四进程Node代码：

var http = require('http');

var server = http.createServer(function (request, response) {

var j = 0;

for (var i = 0; i & lt; 100000; i++) {

j += 2 / 3;

}

response.end(j + '');

});

var nodes = require("./lib/multi-node").listen({

port: 8883,

nodes: 4

}, server);

console.log('Server running at http://10.1.10.150:8883/');

这里简单介绍一下multi-node这个插件，这个插件就是利用require("child\_process").spawn()方法来创建多个子线程。由于浮点计算和字符串拼接都是比较耗CPU的运算，所以这里我们循环10W次，每次对j加上0.66666。最后比较一下，开多子进程node到底比单进程node在CPU密集运算上快多少。

以下是测试结果：

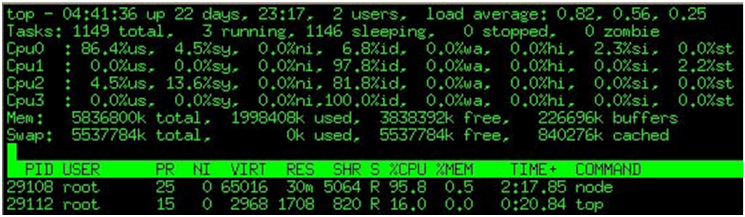
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Comm. | 500/30 | 500/30 | 1000/30 | 1000/30 | 3000/30 | 3000/30 |
| Type | 单进程 | 多子进程 | 单进程 | 多子进程 | 单进程 | 多子进程 |
| **RPS** | **2595** | **5597** | **2540** | **5509** | **2571** | **5560** |
| TPQ | 0.38 | 0.18 | 0.39 | 0.19 | 0.39 | 0.18 |
| 80% REQ | 72 | 65 | 102 | 85 | 157 | 142 |
| Fail | 0 | 0 | 0 | 0 | 0 | 0 |

说明：

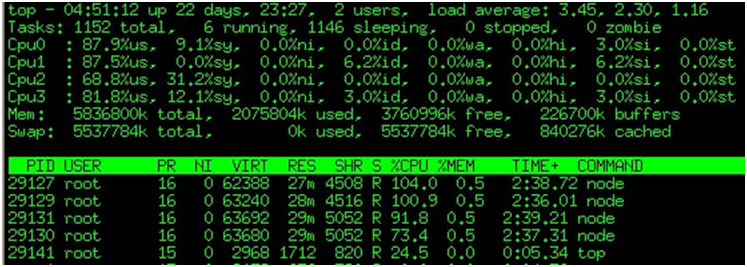
* 单进程：只开一个node.js进程。
* 多子进程：开一个node.js进程，并且开3个它的子进程。
* 3000/30：代表命令 ./ab -c 3000 -t 30 http://10.1.10.150:8888/。3000个客户端，最多发30秒，最多发5W个请求。
* RPS：代表每秒处理请求数，并发的主要指标。
* TPQ：每个请求处理的时间，单位毫秒。
* Fail：代表平均处理失败请求个数。
* 80% Req：代表80%的请求在多少毫秒内返回。

从结果及图1～3上看：开多个子进程可以显著缓解node.js的CPU利用率不足的情况，提高node.js的CPU密集计算能力。

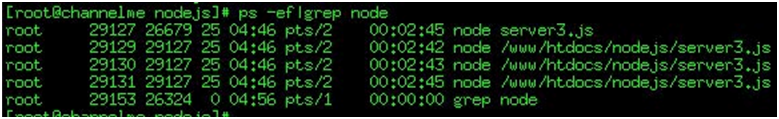
**图1：单个进程的node.js在压力测试下的情况，无法充分利用4核CPU的服务器性能。**



**图2：多进程node，可以充分利用4核CPU榨干服务器的性能。**



**图3：多子进程截图，可以看到一共跑了4个进程。**



# 适用场景

总而言之，NodeJS适合运用在高并发、I/O密集、少量业务逻辑的场景，如实时通信、web推送、页游等场景。

## 适用的具体场景举例

1. RESTful API

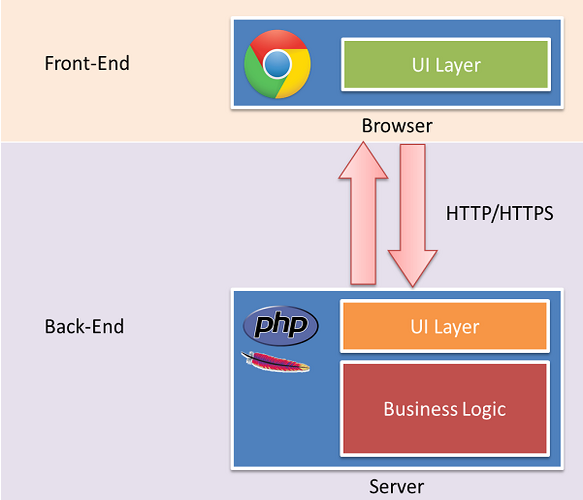
　　这是NodeJS最理想的应用场景，可以处理数万条连接，本身没有太多的逻辑，只需要请求API，组织数据进行返回即可。

　　它本质上只是从某个数据库中查找一些值并将它们组成一个响应。

　　由于响应是少量文本，入站请求也是少量的文本，因此流量不高，一台机器甚至也可以处理最繁忙的公司的API需求。

1. 统一Web应用的UI层（如果使用Node.js构建UI，岂不是又要多两次（Node.js与Php之间）Http请求？！）

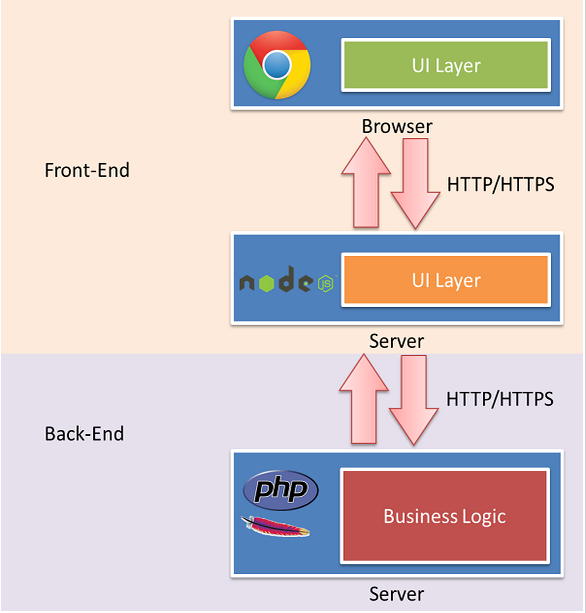
　　目前MVC的架构，在某种意义上来说，Web开发有两个UI层，一个是在浏览器里面我们最终看到的，另一个在server端，负责生成和拼接页面。



　　不讨论这种架构是好是坏，但是有另外一种实践，面向服务的架构，更好的做前后端的依赖分离。

　　如果所有的关键业务逻辑都封装成REST调用，就意味着在上层只需要考虑如何用这些REST接口构建具体的应用。

　　那些后端程序员们根本不操心具体数据是如何从一个页面传递到另一个页面的，他们也不用管用户数据更新是通过Ajax异步获取的还是通过刷新页面



1. 大量Ajax请求的应用

　　例如个性化应用，每个用户看到的页面都不一样，缓存失效，需要在页面加载的时候发起Ajax请求，

　　NodeJS能响应大量的并发请求

1. [JSON APIs](http://nodeguide.com/convincing_the_boss.html#json-apis)

Building light-weight REST / JSON api is something where node.js really shines. Its non-blocking I/O model combined with JavaScript make it a great choice for wrapping other data sources such as databases or web services and exposing them via a JSON interface.

1. [Single page apps](http://nodeguide.com/convincing_the_boss.html#single-page-apps) 单页有很多请求（如ajax请求）的应用

If you are planning to write an AJAX heavy single page app (think gmail), node.js is a great fit as well. The ability to process many requests / seconds with low response times, as well as sharing things like validation code between the client and server make it a great choice for modern web applications that do lots of processing on the client.

1. [Shelling out to unix tools](http://nodeguide.com/convincing_the_boss.html#shelling-out-to-unix-tools) 应用unix的工具

With node.js still being young, it's tempting to re-invent all kinds of software for it. However, an even better approach is tapping into the vast universe of existing command line tools. Node's ability to spawn thousands of child processes and treating their outputs as a stream makes it an ideal choice for those seeking to leverage existing software.

1. [Streaming data](http://nodeguide.com/convincing_the_boss.html#streaming-data) 流数据

Traditional web stacks often treat http requests and responses as atomic events. However, the truth is that they are streams, and many cool node.js applications can be built to take advantage of this fact. One great example is parsing [file uploads in real time](http://transloadit.com/blog/2010/12/realtime-encoding-over-150x-faster), as well as building proxies between different data layers.

1. [Soft Realtime Applications](http://nodeguide.com/convincing_the_boss.html#soft-realtime-applications) 实时应用

Another great aspect of node.js is the ease at which you can develop soft real time systems. By that I mean stuff like twitter, chat software, sport bets or interfaces to instant messaging networks.

But please be careful here, since JavaScript is a dynamic / garbage collected language, your response times may sometimes vary depending on how often and long the garbage collection kicks in (at which point your program is stopped). So don't try to build hard realtime systems in node, that require consistent response times. Erlang is probably a better choice for these kinds of applications.

## 不适用的具体场景举例

1. [CPU heavy apps](http://nodeguide.com/convincing_the_boss.html#cpu-heavy-apps) Cpu密集型和少量的IO操作的应用

Even though I love node.js, there are several use cases where it simply doesn't make sense. The most obvious such case is apps that are very heavy on CPU usage, and very light on actual I/O. So if you're planning to write video encoding software, artificial intelligence or similar CPU hungry software, please do not use node.js. While you can twist and bend things quite a bit, you'll probably get better results with C or C++.

That being said, node.js allows you to easily write C++ addons, so you could certainly use it as a scripting engine on top of your super-secret algorithms.

1. [Simple CRUD / HTML apps](http://nodeguide.com/convincing_the_boss.html#simple-crud-html-apps) 如网站开发等，这些还是用Asp.net/Jsp/Php技术为好

While node.js will eventually be a fun tool for writing all kinds of web applications, you shouldn't expect it to provide you with more benefits than PHP, Ruby or Python at this point. Yes, your app might end up slightly more scalable, but no - your app will not magically get more traffic just because you write it in node.js.

The truth is that while we are starting to see good frameworks for node.js, there is nothing as powerful as Rails, CakePHP or Django on the scene yet. If most of your app is simply rendering HTML based on some database, using node.js will not provide many tangible business benefits yet.

1. [NoSQL + Node.js + Buzzword Bullshit](http://nodeguide.com/convincing_the_boss.html#nosql-node.js-buzzword-bullshit)

If the architecture for your next apps reads like the cookbook of NoSQL ingredients, please pause for a second and read this.

Yes, Redis, CouchDB, MongoDB, Riak, Casandra, etc. all look really tempting, but so did that red apple Eve couldn't resist. If you're already taking a technological risk with using node.js, you shouldn't multiply it with more technology you probably don't fully understand yet.

Sure, there are legitimate use cases for choosing a document oriented database. But if you are trying to build a business on top of your software, sticking to conservative database technology (like postgres or mysql) might just outweigh the benefits of satisfying your inner nerd and impressing your friends.

# 与传统服务器进行比较

1、和传统的服务端脚本语言, 原生的非阻塞io,能够很大程度上提高并发负载能力  
2、对于java,c等直接开发的非阻塞服务,在开发效率和成本上要更胜出.  
3、从语言特性上讲,js语言本身对事件机制由很好的支持,

## 实际应用中的差异

 需求是从数据库表中查出某个字段，并打印出来，采用多线程阻塞I/O服务器的方式通常的做法是：

 name = query("select name from db");

output(name);

数据库层把这个查询发送到数据库，查询结果返回数据库之间，进程就会阻塞对于高并发，I/O密集行的网络应用中，一方面进程很长时间处于等待状态，一方面为了应付新的请求不断的增加新的进程[7]，每个进程执行栈都要占用内存，计算机的资源很快将会耗尽。若采用Nodejs基于事件驱动的非阻塞服务器的方式，以代码如下：

query("select name from db " function(name){

output(name);});

此时查询的结果不是调用函数的结果，而是提供给一个稍后将调用的回调函数，控制会立刻返回事件循环，而进程能够继续处理其他请求。

## 工作原理对比

实际应用的差异的本质就是其工作原理的不同，Nodejs构建的异步非阻塞服务器有两部分组成。在上层是谷歌GoogleV8引擎(单线程)，事件循环和C/C++的库运行在其的基础上，同时负责监听HTTP/TCP请求，在服务器的底层有libuv(包含libio)和其他的C++库。他们的主要作用是提供异步的I/O。Nodejs事件驱动非阻塞I/O服务器模型如图2所示。

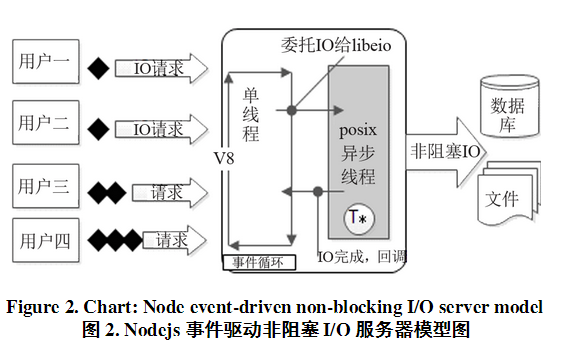
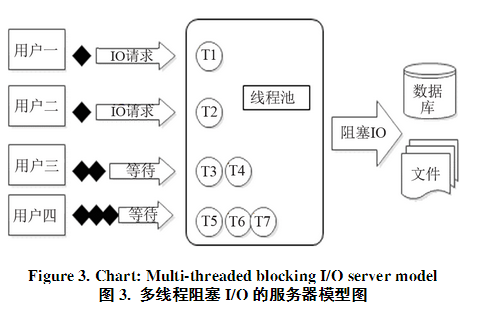


图2描述了Nodejs服务器的工作原理，当一个请求从浏览器发出，运行在谷歌GoogleV8引擎中的主线程会检查它是否含有I/O请求，如果他是I/O请求会立刻委托给底层所在的服务器一个POSIX线程池来执行异步的I/O[8]， 主线是空闲的，所以可以接收其他请求，当与数据库或者文件交换数据完成后，事件触发会向主线程发出一个响应，告诉主线程已经读取完毕，当谷歌GoogleV8引擎执行完正在做的事情空闲时，它将获取这个结果，然后返回到浏览器。

线程阻塞式服务器与异步非阻塞服务器在工作原理上有着较大差异，图3是多线程阻塞式服务器的模型图，其中T表示一个线程。其中的核心模块是线程池，线程池的容量直接决定了服务器的并发数量。

图3模拟了有四个用户登录到多线程服务器，其中两个用户不停的点击刷新按钮导致服务器开启了许多线程(每一次请求开启一个线程)，当一个请求到来时，线程池中一个线程执行它的阻塞I/O操作，去数据库或者文件中进行数据的交换，线程池控制着线程的切换和线程上下文的执行，当I/O操作结束后，操作系统上下文切换回到先前的线程返回结果。请求直接面对这种阻塞的I/O要求更多的内存和CPU资源。

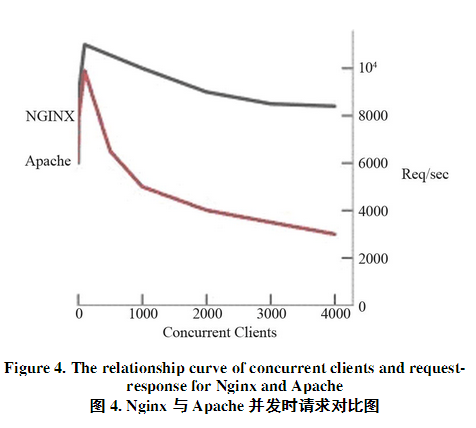


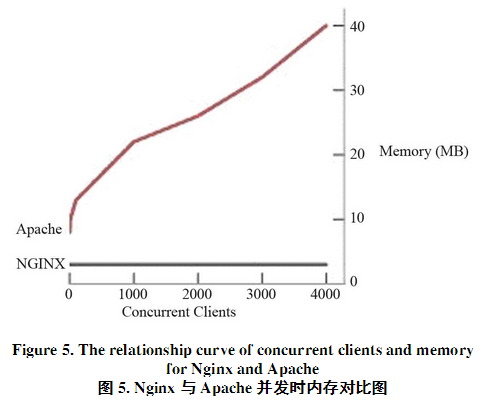
## 性能对比

因为同为这种异步非阻塞体系结构，Nodejs和Nginx性能上表现的同样出色，因为Nginx与Nodejs都是采用事件循环，非阻塞I/O体系，我们将用Nginx与传统HTTP服务的代表Apache(多线程体系结构)进行数据上的对比来侧面证明Nodejs的性能[9,10]。

从图4可以看出，随着并发用户的增加，Apache能及时响应的客户骤减，而Nginx减少的较为缓慢，Apache每一次连接都会开辟一个新线程，当线程达到连接池的上限时便会出现以上症状，而Nginx采用的是事件循环机制则不会出现以上情况。

Nginx与Apache并发时内存使用情况如图5所示，因为Apache这种模式开辟了更多的线程，开辟线程与线程之间的切换会消耗大量内存和CPU资源，显然面对大量请求同时来临时系统的资源会迅速耗尽，这也是引言中提到那些问题的根源，Nginx和No- dejs采用事件机制，Nodejs有效的利用了Javascript闭包和作用域，单线程的模式将内存的消耗降到最低。





## 总结

多线程服务器支持同步、阻塞I/O模型来提供了更简单的方式执行I/O[11]，但是在面对高并发，大负荷时，多线程服务器会使用更多的线程，因为为了保证数据的安全性，每次请求将开辟一个线程，为了支持更多的线程和更多的线程上下文切换会消耗大量内存和CPU利用率，事件驱动非阻塞I/O服务器的体系结构顶层采用事件循环，底层再用内核级异步I/O，连接没有和线程直接关联，这种模式需要主线程不停循环和少量的内核级线程执行I/O，因为有更少的线程，因此更少的上下文切换，它使用更少的内存和CPU也少。从而很高效率利用硬件资源，在面对高并发，大数据量时可以表现更加出色，以上从理论与实际数据的引证证明了这一点。

# 在游戏开发中的应用

## 参考资料

“基于HTML5与Nodejs开发实时性网页对战游戏坦克大战”（pdf）

[web版五子棋(nodejs + socket.io)](http://cnodejs.org/topic/4f3a86cdb43c3c846a00b8ad)

[使用HTML5,WebSockets,nodejs和socket.io构建实时游戏](http://www.oschina.net/question/89964_56072)

## 基于Node.js的游戏服务器端框架Pomelo

### 参考资料

[Pomelo：网易开源基于 Node.js 的游戏服务端框架](http://www.cnblogs.com/lhb25/archive/2012/12/02/pomelo-game-server-framewok-for-node-js.html)

[Pomelo](http://pomelo.netease.com/index.html)

[网易NodeJS开源游戏框架pomelo访谈](http://www.infoq.com/cn/news/2012/11/netease-nodejs-framework)

# 框架

## Express.js

Express.js 是对 [Node.js](http://www.oschina.net/p/nodejs) 的一个高性能的封装，是一个Web应用框架。

示例代码：

创建一个express应用程序

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var express = require('express');var app = express();    app.get('/', function(req, res){      res.send('hello world');  });    app.listen(3000); |