JavaScript杂志

2018年08月01日 第一号

Nodejs专辑

理解NODEJS群集

NODEJS + NGINX 强强合作

Table of Contents

封面故事

如果你看的是离线版的PDF或者EPUB的话，会看到一个封面，上面是一只猫，名字叫做三猫。三猫是公猫，有一对蓝色的眼睛，和一身雪白的、绒绒的毛。它对自己的毛非常爱惜，常常花费整个小时的时间，把它们打整的干干净净。每当它对自己的毛感觉非常良好的时候，会在主人面前晃来晃去的刷存在。

三猫换过几届主人，这些主人目前依然承认这个关系，但是三猫不一定承认了。它住过几个城市，坐过几回飞机，还做过平面模特，一次出工，可以赚几百磅的猫粮呢。算是见过世面。

但是日常起居宅得厉害。如同很多猫一样，它遵守了猫的日常起居：大约16小时碎觉，2小时自己玩，2小时逗主人玩，1小时犯迷糊，1小时吃饭，1小时遛弯，1个小时打扫个猫卫生。我上班的时候，它抬起头看看，然后继续睡，我回来的时候，它过来迎接，然后玩闹一会儿，瞌睡一会儿，然后晚上10点起来吃夜宵的虾，继续睡。

对照它的起居和态度，我看到了一只猫的给我的生活意见。

稿件须知

欢迎投稿或者给线索，请发邮件到 `1000copy@gmail.com` 。

稿件要求和JS相关，因为是免费杂志，因此不会有稿费，但是回报依然是有的，可以在文中附带你的介绍。还可以免费给你登广告。 对了，格式要求是Markdown的。

可能你关心多久出一期？我不知道，有内容的话，有时间的话，就会做。快得话一个月一次，慢的话或许几个月去了，说不定看了这期，下期就没了。

对。这个杂志，就是这么佛系。

对了，总编叫做reco，找好文章编辑下，也自己写，也翻译，手下一个兵丁也是没有的。有兴趣的话，发一个简历，我看得上你的话，可以一起工作。钱是没有的，一分也没有。

如果你喜欢网站阅读的话，可以在这里看到全本，并且，你可以下载其他的离线电子书格式，比如epub，mobi等。

Node.js 介绍

Ryan Dahl于2007年创造了Nodejs。现象表明，Nodejs非常火热。事实证明，Nodejs的成功，因为Ryan做了两个明智的选择：异步IO范式和执行引擎V8。

在Nodejs之前，主流的Web编程模式是同步IO，就是说当应用代码访问IO等消耗时间的代码时，线程会一直等待直到完成，在此期间，此线程无法做其他的工作。PHP，JSP等都是一样的做法。虽然，在等待期间，应用代码理应可以做其他的事儿，但是很少有人这样做。

Ryan的工作是编写写高性能Web服务，这个领域内，异步IO、事件驱动是常常被采用的方法，他评估了很多种高级语言，发现很多语言虽然同时提供了同步IO和异步IO，但是开发人员一旦用了同步IO，他们就再也懒得写异步IO了，所以，最终，Ryan瞄向了JavaScript。因为JavaScript是单线程执行，根本不能进行同步IO操作，所以，JavaScript的这一“缺陷”导致了它只能使用异步IO。

Ryan直接采用了Gooogle V8，号称最快的运行时引擎。这改变了人们对JavaScript运行缓慢的看法，于是，当发现Node把JavaScript带入到后端服务器开发时，大量的存量JavaScript开发人员，终于找到了在后端变现自己的知识的最佳通道。所以Node一下子就火了起来。

在接受Mappingthejourney访谈是，Ryan也提到了一个相对比较小的，但是也比较实际的要素：“老实说，构建一个Web服务器并不是最简单的事情，我认为很多这些系统都留给他们的社区来做，但是，事实上是，没有人这样做”，而Ryan做了。

当然，语言本身不算太差，甚至还有些亮点的。JavaScript语言本身是完善的函数式语言，还有表达能力很强但是非常简洁的字面量对象和JSON。曾经JavaScript被认为就是个“玩具”，但是，在Gmail,Google Map这样的优秀的应用的激发下，人们发现，通过模块化的JavaScript代码，加上函数式编程，直接使用最新的ECMAScript标准，其实是可以完全满足工程上的需求的。

Node.js理解事件循环机制

by wayneli 编辑：reco

前沿

Node.js 是基于V8引擎的javascript运行环境. Node.js具有事件驱动, 非阻塞I/O等特点. 结合Node API, Node.js 具有网络编程, 文件系统等服务端的功能, Node.js用libuv库进行异步事件处理.

线程

Node.js的单线程含义, 实际上说的是执行同步代码的主线程. 一个Node程序的启动, 不止是分配了一个线程，而是我们只能在一个线程执行代码. 当出现I/O资源调用, TCP连接等外部资源申请的时候, 不会阻塞主线程, 而是委托给I/O线程进行处理,并且进入等待队列. 一旦主线程执行完成，将会消费事件队列(Event Queue). 因为只有一个主线程, 只占用CPU内核处理逻辑计算, 因此不适合在CPU密集型进行使用.

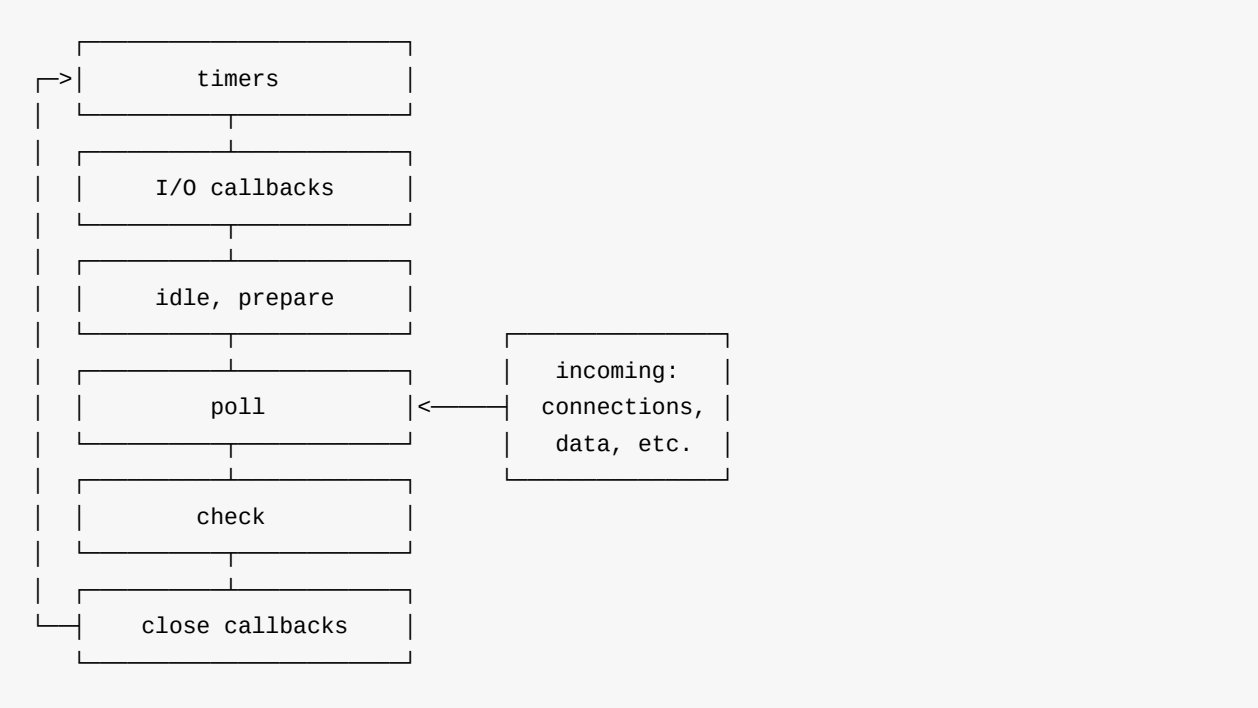
事件循环机制

什么是事件循环(EventLoop) ?EventLoop 是一种常用的机制，通过对内部或外部的事件提供者发出请求, 如文件读写, 网络连接 等异步操作, 完成后调用事件处理程序. 整个过程都是异步阶段。

!()[eventloop.png]

注意，上图的EVENT_QUEUE 给人看起来是只有一个队列,EventLoop有6个阶段, 同时每个阶段都有对应的一个先进先出的回调队列.当Node.js 启动, 就会初始化一个 event loop, 处理脚本时, 可能会发生异步API行为调用, 使用定时器任务或者nexTick, 处理完成后进入事件循环处理过程

事件循环阶段：



每一个阶段都有一个FIFO的callbacks队列, 每个阶段都有自己的事件处理方式. 当事件循环进入某个阶段时, 将会在该阶段内执行回调，直到队列耗尽或者回调的最大数量已执行, 那么将进入下一个处理阶段.

timers

这个阶段执行setTimeout(callback) and setInterval(callback)预定的callback; 一个timer指定一个下限时间而不是准确时间，在达到这个下限时间后执行回调。在指定时间过后，timers会尽可能早地执行回调，但系统调度或者其它回调的执行可能会延迟它们。

I/O callbacks

执行除了close事件的callbacks、被timers设定的callbacks、setImmediate()设定的callbacks之外的callbacks。这个阶段执行一些系统操作的回调。比如TCP错误，如一个TCP socket在想要连接时收到ECONNREFUSED, 类unix系统会等待以报告错误，这就会放到 I/O callbacks阶段的队列执行. 名字会让人误解为执行I/O回调处理程序, 实际上I/O回调会由poll阶段处理。

idle, prepare 阶段:

仅node内部使用;

poll 阶段

获取新的I/O事件, 适当的条件下node将阻塞在这里;poll 阶段有两个主要功能：执行下限时间已经达到的timers的回调，然后 处理 poll 队列里的事件。

当event loop进入 poll 阶段，并且 没有设定的timers（there are no timers scheduled），会发生下面两件事之一：

如果 poll 队列不空，event loop会遍历队列并同步执行回调，直到队列清空或执行的回调数到达系统上限；

如果 poll 队列为空，则发生以下两件事之一：

如果代码已经被setImmediate()设定了回调, event loop将结束 poll 阶段进入 check 阶段来执行 check 队列（里的回调）。 如果代码没有被setImmediate()设定回调，event loop将阻塞在该阶段等待回调被加入 poll 队列，并立即执行。 但是，当event loop进入 poll 阶段，并且 有设定的timers，一旦 poll 队列为空（poll 阶段空闲状态）：

event loop将检查timers,如果有1个或多个timers的下限时间已经到达，event loop将绕回 timers 阶段，并执行 timer 队列。 check阶段 这个阶段允许在 poll 阶段结束后立即执行回调。如果 poll 阶段空闲，并且有被setImmediate()设定的回调，event loop会转到 check 阶段而不是继续等待。

通常上来讲，随着代码执行，event loop终将进入 poll 阶段，在这个阶段等待 incoming connection, request 等等。但是，只要有被setImmediate()设定了回调，一旦 poll 阶段空闲，那么程序将结束 poll 阶段并进入 check 阶段，而不是继续等待 poll 事件们 （poll events）。

1. check 阶段: 执行setImmediate() 设定的callbacks;setImmediate()实际上是一个特殊的timer，跑在event loop中一个独立的阶段。它使用libuv的API来设定在 poll 阶段结束后立即执行回调。

close callbacks 阶段

比如socket.on(‘close’, callback)的callback会在这个阶段执行.如果一个 socket 或 handle 被突然关掉（比如 socket.destroy()），close事件将在这个阶段被触发，否则将通过 process.nextTick()触发

简单的 EventLoop

```
const fs = require('fs');
let counts = 0;
function wait (mstime) {
  let date = Date.now();
  while (Date.now() - date < mstime) {
    // do nothing
  }
}
function asyncOperation (callback) {
  fs.readFile(__dirname + '/' + __filename, callback);
}
const lastTime = Date.now();
setTimeout(() => {
  console.log('timers', Date.now() - lastTime + 'ms');
}, 0);
process.nextTick(() => {
  // 进入event loop
  // timers阶段之前执行
  wait(20);
  asyncOperation(() => {
    console.log('poll');
  });
});
```

为了让setTimeout优先于fs.readFile 回调, 执行了process.nextTick, 表示在进入 timers阶段前, 等待20ms后执行文件读取.

nextTick

process.nextTick 不属于事件循环的任何一个阶段，它属于该阶段与下阶段之间的过渡, 即本阶段执行结束, 进入下一个阶段前, 所要执行的回调。有给人一种插队的感觉。由于nextTick具有插队的机制，nextTick的递归会让事件循环机制无法进入下一个阶段. 导致I/O处理完成或者定时任务超时后仍然无法执行, 导致了其它事件处理程序处于饥饿状态. 为了防止递归产生的问题, Node.js 提供了一个 process.maxTickDepth (默认 1000)。

```
const fs = require('fs');
let counts = 0;
function wait (mstime) {
  let date = Date.now();
  while (Date.now() - date < mstime) {
    // do nothing
  }
}
function nextTick () {
  process.nextTick(() => {
    wait(20);
    nextTick();
  });
}
const lastTime = Date.now();
setTimeout(() => {
  console.log('timers', Date.now() - lastTime + 'ms');
}, 0);
nextTick();
```

此时永远无法跳到timer阶段, 因为在进入timers阶段前有不断的nextTick插入执行. 除非执行了1000次到了执行上限.

setImmediate

如果在一个I/O周期内进行调度，setImmediate（）将始终在任何定时器之前执行.setTimeout与 setImmediate setImmediate()被设计在 poll 阶段结束后立即执行回调；setTimeout()被设计在指定下限时间到达后执行回调;

无 I/O 处理情况下

```
setTimeout(function timeout () {
  console.log('timeout');
},0);
setImmediate(function immediate () {
  console.log('immediate');
});
```

输出结果是不确定的！setTimeout(fn, 0) 具有几毫秒的不确定性. 无法保证进入timers阶段, 定时器能够立即执行处理程序.

在I/O事件处理程序下：

```
var fs = require('fs')
fs.readFile(__filename, () => {
  setTimeout(() => {
    console.log('timeout')
  }, 0)
  setImmediate(() => {
    console.log('immediate')
  })
})
```

此时 setImmediate 优先于 setTimeout 执行，因为 poll阶段执行完成后 进入 check阶段. timers阶段处于下一个事件循环阶段了。

Nodejs的优点和缺点

作者：FengqiAsia 编辑：reco

要讲清楚这个问题，先讲讲整个Web应用程序架构中的瓶颈。一个常见的瓶颈在于服务器能够处理的并发连接的最大数量。

Node.js宣称，运行它的服务器能支持数万个并发连接。Node.js解决这个问题的方法是：更改连接到服务器的方式。每个连接发射一个在Node.js引擎的进程中运行的事件，而不是为每个连接生成一个新的OS线程，以及为此需要分配一些配套资源如内存。Node.js不会死锁，因为它根本不允许使用锁，它不会直接阻塞I/O调用。

优点：

1. 采用事件驱动、异步编程，为网络服务而设计。其实Javascript的匿名函数和闭包特性非常适合事件驱动、异步编程。
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非常适合如下情况：在响应客户端之前，您预计可能有很高的流量，但所需的服务器端逻辑和处理不一定很多。
4. 开发者比较容易找到。很多前端设计人员可以很快上手做后端设计。一般公司都不乏Web前端工程师，而此类工程师的学习门槛也非常低。这就意味着Node.js很容易招人，或者公司就隐藏了一些高手强大的社区支持——Node.js社区非常活跃，吸引很多优秀的工程师，这就意味着公司可以很容易从社区得到免费或者付费的支持系统性能考虑——JavaScript引擎Google V8，加之原生异步IO模型，使得Node.js在性能的表现非常出色，处理数以千计的并发请求非常轻松专业公司的支持。

缺点：

1. 可靠性低。默认情况下是单进程，单线程的，一旦这个进程崩掉，那么整个web服务就崩掉了。
2. 默认情况下只支持单核CPU，不能充分的利用多核CPU服务器。
3. 但Javascript是有垃圾回收机制的，这就意味着，系统的响应时间是不平滑的(GC垃圾回收会导致系统这一时刻停止工作)。如果想要构建硬实时应用系统，Erlang是个不错的选择

不过以上缺点可以可以通过代码的健壮性来弥补。目前Node.js的网络服务器有以下几种支持多进程的方式：

1. 开启多个进程，每个进程绑定不同的端口，用反向代理服务器如 Nginx 做负载均衡，好处是我们可以借助强大的 Nginx 做一些过滤检查之类的操作，同时能够实现比较好的均

衡策略，但坏处也是显而易见——我们引入了一个间接层。

2. 多进程绑定在同一个端口侦听。在Node.js中，提供了进程间发送“文件句柄”的功能，这个功能实在是太有用了（貌似是yahoo 的工程师提交的一个patch），不明真相的群众可以看这里： Unix socket magic
3. 一个进程负责监听、接收连接，然后把接收到的连接平均发送到子进程中去处理。

在Node.js v0.5.10+ 中，内置了cluster 库，官方宣称直接支持多进程运行方式。Node.js 官方为了让API 接口傻瓜化，用了一些比较tricky的方法，代码也比较绕。这种多进程的方式，不可避免的要牵涉到进程通信、进程管理之类的东西。此外，有两个Node.js的module：multi-node 和 cluster ，采用的策略和以上介绍的类似，但使用这些module往往有一些缺点：

1. 更新不及时
2. 复杂庞大，往往绑定了很多其他的功能，用户往往被绑架
3. 遇到问题难以解决

它不适合CPU使用率较重、IO使用率较轻的应用——如视频编码、人工智能等。Node.js的优势无法发挥简单Web应用——此类应用的特点是，流量低、物理架构简单，Node.js无法提供像Ruby的Rails或者Python的Django这样强大的框架

说服自己老板采用Node.js的方式构建一个简单的原型——花一周时间构建系统某一部分的原型是非常值得的，同时也很容易和老板在某一点达成一致，等到系统真的在某一部分应用了Node.js，就是打开局面的时候寻找开发者——首先JavaScript语言的普及度很高。

nodejs并行实践问答

作者：hasanyasin@stackoverflow Ankara, Turkey 翻译：reco

Node具有完全不同的编程范式，一旦正确理解，就更容易看到解决问题的不同方式。您永远不需要Node应用程序中的多个线程，因为您有不同的方式来执行相同的操作。您创建多个流程;但它与Apache Web Server的Prefork mpm的功能非常不同。

现在，让我们假设我们只有一个CPU核心，我们将开发一个应用程序（以Node的方式）来完成一些工作。我们的工作是逐个字节地处理一个大文件。我们软件的最佳方法是从文件的开头开始工作，逐个字节地跟踪到最后。

- 嘿，hasan，我想你要么是新手要么和我爷爷一样的老派！为什么不创建一些线程并使其更快？
- 哦，我们只有一个CPU核心。
- 所以呢？创建一些线程，让它更快！
- 它不像那样工作。如果我创建线程，我将使它变慢。因为我将在系统中添加大量开销以在线程之间进行切换，并在我的进程内尝试在这些线程之间进行通信。除了所有这些事实之外，我还必须考虑如何将单个作业分成多个可以并行完成的作业。
- 好的，我觉得你很穷！我们用我的电脑，它有32个核心！
- 哇，亲爱的朋友，你眞棒，非常感谢你。我很感激！

然后我们回头工作。现在我们拥有32个cpu核心，感谢我们的朋友。我们必须遵守的规则刚刚改变了。现在我们想要利用我们给予的所有这些财富。

要使用多个内核，我们需要找到一种方法将我们的工作分成可以并行处理的部分。如果它不是Node.js，我们会为此使用线程; 32个线程，每个cpu核心一个。但是，由于我们有Node.js，我们将创建32个Node进程。

线程可以是Node进程的一个很好的替代方案，甚至可能是更好的方法;但只有在已定义工作的特定工作中，我们才能完全控制如何处理工作。除此之外，对于其他任何类型的问题，如果工作来自外部，我们无法控制，我们希望尽快回答，Node的方式将会是无可争议地更加优越。

- 嘿，Hasan，你还在单线程吗？你怎么了，伙计？我刚刚为你提供了你想要的东西。你没有任何借口了。创建线程，让它运行得更快。
- 我已将作品分成几部分，每个进程都会同时处理其中一个部分。
- 你为什么不创建线程？

- 对不起，我觉得它不可用。要不你把你的电脑拿走？
- 不行，我很酷，我只是不明白为什么你不使用线程？
- 谢谢你的电脑。 :)我已经将工作分成几部分，我创建了并行处理这些部分的流程。所有CPU核心都将得到充分利用。我可以用线程代替进程来做到这一点;但Node有另外一种方式，而我的老板Parth Thakkar希望我使用Node。
- 好的，如果你需要另一台电脑，请告诉我。 ：p

如果我创建了33个进程而不是32个进程，那么操作系统的调度程序将暂停一个线程，启动另一个进程，在一些循环后暂停它，再次启动另一个进程.....这是不必要的开销。我不想要这个。实际上，在具有32个内核的系统上，我甚至不想创建正好32个进程，31个可以更好。因为不仅仅是我的应用程序可以在这个系统上工作。为其他东西留一点空间可能会很好，特别是如果我们有32个房间。

我相信我们现在在同一问题上了，现在的问题是关于充分利用处理器来执行CPU密集型任务。

- 嗯，hasan，我很抱歉嘲笑你一点。我相信我现在更了解你。但是我还需要解释一下：运行数百个线程的哼哼唧唧是什么？我到处都看到线程比创建进程更快创建？你创建进程而不是线程，你认为它是你这样使用Node是最优的。难道Node不适合这种工作吗？
- 不用担心，我也很酷。每个人都说这些东西，所以我想我已经习惯了听到它们。
- 那么？Nodejs不擅长此道？
- 即使线程也很好，Nodejs对此非常有用。至于线程和进程创建开销;在你重复的事情上，每一毫秒都很重要。但是，我只创建了32个进程，这将花费很少的时间。它只会发生一次。它没有任何区别。
- 我什么时候想创建数千个线程呢？
- 你永远不想创建成千上万的线程。但是，在正在进行外部工作的系统上，例如处理HTTP请求的Web服务器;如果你为每个请求使用一个线程，你就真的会创建数千个线程的。
- Node.js有所不同吗？对？
- 对，就是这样。这是Node真正闪耀的地方。就像线程比进程轻得多，函数调用比线程轻得多。Node.js调用函数，而不是创建线程。在Web服务器的示例中，每个传入的请求都会导致函数调用。
- 嗯，有趣;但是如果不使用多个线程，则只能同时运行一个函数。当很多请求同时到达Web服务器时，它如何工作？

- 关于函数如何运行，一次一个，从不并行运行，你是完全正确的。我的意思是在一个进程中，一次只运行一个代码范围。操作系统调度程序不会暂停此功能并切换到另一个功能，除非它暂停进程以给另一个进程留出时间，而不是我们进程中的另一个进程。
- 那么一个进程如何一次处理2个请求？
- 只要我们的系统有足够的资源（RAM，网络等），一个进程就可以同时处理数万个请求。这些功能如何运行是关键的差异。
- 嗯，我现在应该兴奋吗？
- 也许:)Node.js在队列上运行一个循环。在这个队列中是我们的工作，即我们开始处理传入请求的调用。这里最重要的一点是我们设计运行函数的方式。我们不是开始处理请求并让调用者等到我们完成工作，而是在完成可接受的工作量后快速结束我们的功能。当我们需要等待另一个组件做一些工作并返回一个值而不是等待它时，我们只需完成我们的功能，将剩下的工作添加到队列中。
- 听起来太复杂了？
- 不，不，我可能听起来很复杂;但系统本身非常简单，而且非常有意义。

现在我想停止引用这两个开发人员之间的对话，并在最后一个关于这些功能如何工作的快速示例之后完成我的回答。

通过这种方式，我们正在做OSScheduler通常会做的事情。我们暂停我们的工作，并让其他函数调用（如多线程环境中的其他线程）运行，直到我们再次轮到我们。这比将工作留给OSScheduler好得多，后者试图给系统上的每个线程提供时间。我们知道我们做得比OSScheduler做得好得多，我们应该在我们停止时停止。

下面是一个简单的例子，我们打开一个文件并读取它来对数据做一些工作。

同步方式：

```
打开文件
重复这个：
    阅读一些
    做的工作
```

异步方式：

```
打开文件并在准备就绪时执行：//我们的函数返回
    重复这个：
        阅读一些，当它准备就绪时：//再次返回
            做一些工作
```

如您所见，我们的函数要求系统打开文件，而不是等待它打开。文件准备好后，通过提供后续步骤完成自己。当我们返回时，Node在队列上运行其他函数调用。在遍历所有函数之后，事件循环移动到下一轮...

总之，Node具有与多线程开发完全不同的范式;但这并不意味着它缺乏东西。对于同步作业（我们可以决定处理的顺序和方式），它可以和多线程并行一样工作。对于来自外部的工作，如对服务器的请求，它就是更优越的。

ref@stackoverflow: Which would be better for concurrent tasks on node.js?

Vue.js作者尤雨溪作序推荐： 这是一本小书，但麻雀虽小，五脏俱全。篇幅不长，涵盖的内容却面面俱到；虽然一些部分没有特别深入，但全书脉络清晰，行文通畅，浅显易懂，很适合新手入门。希望这本书能够帮助更多的开发者走进 Vue.js 的世界，让前端开发变成一件值得享受的事情。

购买

Nginx异步处理的优势

作者：Assassinの cnblogs 编辑：reco

Web服务器和客户端是一对多的关系，Web服务器必须有能力同时为多个客户端提供服务。一般来说，完成并发处理请求工作有三种方式可供选择、多进程、多线程、异步方式。

多进程方式

多进程方式是指，服务器每当接收到一个客户端时，就由服务器主进程生成一个子进程出来和该客户端建立连接进行交互，直到连接断开，改子进程就结束了。多进程方式的优点在于，设计和实现相对简单，各个子进程之间相互独立，处理客户端请求的过程彼此不受到干扰，并且当一个子进程产生问题时，不容易将影响蔓延到其他进程中，这保证了提供服务的稳定性。当子线程退出时，其占用资源会被操作系统回收，也不会留下任何垃圾。而其缺点也很明显。操作系统中生成一个子进程需要进行内存复制等操作，在资源和时间上回产生一定的额外开销，因此，如果Web服务器接收大量并发请求，就会对系统资源造成压力，导致系统性能下降。

初期的Apache服务器就是采用这种方式对外提供服务的。为了应对大量并发请求，Apache服务器采用“预生成进程”的机制对多进程方式进行改进，“预生成进程”的工作方式很好理解。它将生成子进程的时机提前，在客户端请求还没有到来之前就预先生成好，当请求到来时，主进程分配一个子进程和该客户端进行交互，交互完成之后，该进程也不结束，而被主进程管理起来等待下一个客户端请求的到来，改进的多进程方式在一定程度上缓解了大量并发请求情形下Web服务器对系统资源造成的压力。但是优化Apache服务器在最初的构架设计上采用了多进程方式，因此这不能从根本上解决问题。

多线程方式

多线程方式和多进程方式相似，他是指，服务器每接收到一个客户端时，会有服务器主进程派生一个线程出来和该客户端进行交互。

由于操作系统产生一个线程的开销远远小于产生一个进程的开销，所以多线程方式在很大程度上减轻了Web服务器对系统资源的要求。该方式使用线程进行任务调度，开发方面可以遵循一定的标准，这相对来说比较规范和有利于协作。但在线程管理方面，改方式有一定的不足。多个线程位于同一个进程内，可以访问同样的内存空间，彼此之间相互影响；同时，在开发过程中不过避免地要开发者自己对内存进行管理，其增加了出错的风险。服务器系统需要长时间连续不停地运转，错误的逐渐积累可能最终对整个服务器产生重大影响。

IIS服务，器使用了多线程方式对外提供服务，它的稳定性相对来说还是不错的，但对于经验丰富的Web服务器管理人员而言，他们通常还是会定期检查和重启服务器，以预防不可预料的故障发生。

异步方式

异步方式是和多进程方式及多线程方式完全不同的一种处理客户端请求的方式。在介绍改方式之前 ，我们先复习下同步、异步以及阻塞、非阻塞的概念。

网络通信中的同步机制和异步机制是描述通信模块的概念。同步机制，是指发送方发送请求后，需要等待接收到接收方发回的响应后，才接着发送下一个请求；异步机制，和同步机制正好相反，在异步机制中，发送方发出一个请求后，不等待接收方响应这个请求，就继续发送下个请求。在同步机制中，所有的请求在服务器端得到同步，发送方和接收方对请求的处理步调是一致的；在异步机制中，所有来着发送方的请求形成一个队列，接收方处理完成后通知发送方。

阻塞和非阻塞用来描述进程处理调用的方式，在网络通信中，主要指网络套接字Socket的阻塞和非阻塞方式，而Socket的实质也是IO操作，Socket的阻塞调用方式为，调用结果返回之前，当前线程从运行状态被挂起，一直等到调用结果返回之后，才进行就绪状态，获取CPU后继续执行；Socket的非阻塞调用方式和阻塞方式调用方式正好相反，在非阻塞方式中，如果调用结果不能马上返回，当前线程也不会被挂起，而是立即返回执行下一个调用。

在网络通信中，经常可以看到有人讲同步和阻塞等同、异步和非阻塞等同。事实上，这两对概念有一定的区别，不能混淆。两对概念的组合，就会产生四个新的概念，同步阻塞、异步阻塞、同步非阻塞、异步非阻塞。 同步阻塞方式，发送方向接收方发送请求后，一直等待响应；接收方处理请求是进行的IO操作如果不能马上得到结果，就一直等到返回结果后，才响应发送方，期间不能进行其他工作。比如、在超时排队付账时，客户（发送方）想收款员（接收方）付款（发送请求）后需要等待收款员找零，期间不能做其他的事情；而收款员要等待收款机返回结果（IO操作）后才能把零钱取出来交给客户（响应请求），期间也只能等待，不能做其他的事情。这种方式实现简单，但是效率不高。 同步非阻塞方式，发送方向接收方发送请求后，一直等待响应；接收方处理请求时进行的IO操作如果不能马上得到结果，就立即返回，去做其他事情，但由于没有得到请求处理结果，不响应发送方，发送方一直在等待，一直等IO操作完成后，接收方获得结果响应发送方后，接收方才进入下一次请求过程。在实际中不使用这种方式。 异步阻塞方法，发送方向接收方发送请求后，不用等待响应，可以继续其他工作；接收方处理请求是进行的IO操作如果不能马上得到结果，就一直等到返回结果后，才响应发送方，期间不能进行其他工作。这种方式在实际中也不使用。 异步非阻塞方式，发送方向接收方发送请求后，不用等待响应，可以继续其他工作；接收方处理请求时进行的IO操作富国不能马上得到结果，也不等待，而是马上返回去做其他事情。当IO操作完成以后，将完成状态和结果通知接收方，接收方再响应发送方。继续使用在超市付账排队的例子。客户（发送方）想收款员（接收方）付款（发送请求）后在等待收款员找零的

过程中，还可以做其他事情，比如打电话、聊天等；而收款员在等待收款机处理交易（IO操作）的过程中可以帮助客户将商品打包，当收款机产生结果后，收款员给客户结账（响应请求）。在四种方式中，这种方式是发送方和接收方通信效率最高的一种。

Nginx如何处理请求

从设计架构来说，Nginx服务器是与众不同的。不同之处一方面体现在它的模块化设计，另一方面，也是最重要的一方面，体现在它对客户端请求的处理机制上。Nginx服务器的一个显著优势是能够同时处理大量并发请求。它结合多进程机制和异步机制对外提供服务，异步机制使用的是异步非阻塞方式 。Nginx服务器启动后，可以产生一个主进程（master process）和多个工作进程（worker processes），其中可以在配置文件中指定产生的工作进程数量。Nginx服务器的所有工作进程都用于接收和处理客户端的请求。这类似于Apache使用的改进的多进程机制，预先生成多个工作进程，等待处理客户端请求。 每个工作进程使用了异步非阻塞方式，可以处理多个客户端请。当某个工作进程接收到客户端的请求以后，调用IO进行处理，如果不能立即得到结果，就去处理其他的请求；而客户端在此期间业务需等待响应，可以去处理其他的事情；当IO调用返回结果时，就会通知此工作进程；该进程的到通知，暂时挂起当前处理的事务，去响应客户端请求。 客户端请求数量增长、网络负载繁重时，Nginx服务器使用多进程机制能够保证不增长对系统资源的压力；同时使用异步非阻塞方式减少了工作进程在I/O调用上的阻塞延迟，保证了不降低对请求的处理能力。

Nginx事件驱动处理模型

在上面我们提到，Nginx服务器的工作进程调用IO后，就去进行其他工作了；当IO调用返回后，会通知工作进程。这里有一个问题，IO调用时如何把自己的状态通知给工作进程的呢？一般解决这个问题的方案有两种。一是，让工作进程在进行其他工作的过程中间隔一段时间就去检查一下IO 的运行状态，如果完成，就去响应客户端，如果未完成，就继续在进行的工作；二是，IO调用在完成后能主动通知工作进程。对于前者，虽然工作进程在IO调用过程中没有等待，但不断的检察仍然在时间和资源上导致了不小的开销，最理想的解决方案就是第二种。

具体来说，select/pool/epool/kqueue等这样的系统调用就是用来支持第二种解决方案的。这些系统调用，也常被称为事件驱动模型，他们提供了一种机制，让进程可以同时处理多个并发请求，不用关心IO调用的具体状态，IO调用完全由事件驱动模型来管理，事件准备好之后就通知工作进程事件已经就绪。

事件驱动处理库有被称为多路IO复用方法，最常见的包括以下三种：select模型、poll模型和epoll模型。

select库

select库，是各个版本的Linux和Windows平台都支持的基本事件驱动模型库，并且在接口的定义上也基本相同，只是部分参数的含义略有差异。使用select库的步骤一般是：

首先，创建所关注事件的描述符集合。对于一个描述符，可以关注其上面的读（Read）事件、写（Write）事件以及异常发生（Exception）事件，所以要创建三类事件描述集合，分别用来收集读事件的描述符、写事件的描述符和异常事件的描述符。

其次，调用底层提供的select（）函数，等待事件发生。这里需要注意的一点是，select的阻塞与是否设置非阻塞I/O是没有关系的。

然后，轮询所有事件描述符集合中的每个事件描述符，检查是否有相应的事件发生，如果有，就进行处理。

Nginx服务器在编译过程中如果没有为其他指定其他高性能事件驱动模型库，它将自动编译该库。我们可以使用--with-select_module和--without-select_module两个参数强制Nginx是否编译该库。

poll库

poll库，作为Linux平台上的基本事件驱动模型，是在Linux2.1.23中引入。Windows平台不支持poll库。

poll与select的基本工作方式是相同的，都是先创建一个关注事件的描述符集合，再去等待这些事件的发生，然后再轮询描述符集合，检查有没有事件发生，如果有，就进行处理。

poll库与select库的主要区别在于，select库需要为读事件、写事件和异常事件分别创建一个描述符集合，因此在最后轮询的时候，需要分别轮询这三个集合。而poll库只需要创建一个集合，在每个描述符对应的结构上分别设置读事件、写事件或者异常事件，最后轮询的时候，可以同时检查者三种事件是否发生。可以说，poll库是select库的优化实现。

Nginx服务器在编译过程中如果没有为其制定其他高性能事件驱动模型库，它将自动编译该库。我们可以使用--with-poll_module和--without-poll_module两个参数强制Nginx是否编译该库

epoll库

epoll库是Nginx服务器支持的是高性能事件驱动库之一，它是公认的非常优秀的事件驱动模块，和poll库及select库有很大的不同。epoll属于poll库的一个变种，是在Linux2.5.44中引入的，在Linux2.6及以上的版本都可以使用它。poll库和select库在实际工作中，最大的区别在于效率。

从前面的介绍我们知道，它们的处理方式都是创建一个待处理事件列表，然后把这个列表发给内核，返回的时候，再去轮询检查这个列表，以判断事件是否发生。这样在描述符比较多的应用中，效率就显得比较低下了。一种比较好的做法是，把描述符列表的管理交由内核负责，一旦有某种事件发生，内核把发生时间的描述符列表通知给进程，这样就避免了轮询整个描述符列表。epoll库就是这样一种模型。

首先，epoll库通过相关调用通知内核创建一个有N个描述符的事件列表；然后，给这些描述符设置锁关注的事件，并把它添加到内核的事件列表中去，在具体的编码过程中也可通过相关调用对时间列表中描述符进行修改和删除。

完成设置之后，epoll库就开始等待内核通知事件发生了。某一事件发生后，内核讲发生事件的描述符列表上报给epoll库。得到时间列表的epoll库，就可以进行事件处理了。

epoll库在Linux平台上是高效的。它支持一个进程打开大数目的时间描述符，上限是系统可以打开文件的最大数目；同时，epoll库的IO效率不随描述符数目增加而线性下降，因为它只会对内核上报的“活跃”的描述符进行操作

原文：从Web请求处理机制中剖析多进程、多线程、异步IO by Assassinの

node+Nginx联手

作者：reco

node.js 做服务器？

久经考验的nginx 前置顶住压力，后面多个node服务器完成业务支撑，这样的做法是放心的，是走正道的。

这里要做一个实验：

```
1. 一个nginx作为前台的服务器
2. 全部请求，经过负载均衡，尽可能均衡的分步到后面的2台node服务器
```

准备node

首先启动两台node，分别监听3000，3001端口。为了区分，helloworld会带一个端口返回，通知客户端，以便区别是谁在提供服务。

node.js server

```
$cat 1.js
require('http').createServer(function (request, response) {
  response.end('hello world\n'+process.argv[2]);
}).listen(process.argv[2]);
$node 1.js 3000
$node 1.js 3001
```

验证node 服务器启动

```
λ curl localhost:3000
hello world
3000
λ curl localhost:3001
hello world
3001
```

准备nginx 服务器的配置。

要点是通过upstream 指令把两个node服务器打成一个服务器池。然后通过location指令，要求全部根目录请求转发到这个池内。池会对它之内的服务器做负载均衡。

nginx conf

```
worker_processes  1;
events {
    worker_connections  1024;
}
http {
     upstream node_server_pool {
       server localhost:3001 max_fails=1;
       server localhost:3000 max_fails=1;
    }
    server{
      listen       80;
      server_name localhost;
      location /
       {
        proxy_pass http://node_server_pool;
       }
    }
}
```

模拟客户端访问

我用curl多次访问nginx服务器，可以通过返回的字符串知道服务器。你看，真的可以负载均衡:一会儿helloworld 3000，一会儿helloworld 3001。

```
λ curl localhost
hello world
3000

λ curl localhost
hello world
3001

λ curl localhost
hello world
3001

λ curl localhost
hello world
3000
```

你看，ngnix是公平的。哪怕就一个客户的多次访问都会换着服务器来。

参考

Node.js + Nginx - What now? - Stack Overflow - http://stackoverflow.com/questions/5009324/node-js-nginx-what-now 为高负载网络优化 Nginx 和 Node.js - 技术翻译 - 开源中国社区 - http://www.oschina.net/translate/optimising-nginx-node-js-and-networking-for-heavy-workloads 让node.js充分利用多核服务器的性能，运用nginx做反向代理和负载均衡 - snoopyxdy的日志 - 网易博客 - http://snoopyxdy.blog.163.com/blog/static/601174402011729546489 52/

PM2概述

翻译：reco

为什么要使用PM2？在本概述的最后，您将更好地了解使用PM2作为流程管理器的好处。

永远激活

一旦启动，您的应用程序将永远存在，并在崩溃和计算机重新启动时自动重新启动。

```
var http = require("http");
http.createServer(function (request, response) {
   response.writeHead(200, {'Content-Type': 'text/plain'});
   response.end('Hello World\n');
}).listen(3000);
console.log('Server running at http://127.0.0.1:3000/');
```

这就像运行一样简单：

```
pm2 start app.js
```

进程管理

您的所有应用程序都在后台运行，并且可以轻松管理。

PM2创建一个进程列表，您可以使用以下方法访问：

```
pm2 ls
```

使用 `pm2 start` 和 `pm2 delete` 添加和删除进程列表中的进程。

使用 `pm2 start` ， `pm2 stop` ， `pm2 restart` 管理你的进程。

日志管理

应用程序日志保存在服务器的硬盘中，放在 `~/.pm2/logs` 中。

使用以下方法访问实时日志：

```
pm2 log <app_name>
```

负载均衡

PM2可以通过创建共享同一服务器端口的多个子进程来扩展您的应用程序。这样做还允许您以零秒停机时间重新启动应用程序。

以群集模式启动：

```
pm2 start -i max
```

终端监控

您可以在终端中监控您的应用程序并检查应用程序运行状况（CPU使用率，使用的内存，请求/分钟等）。

```
pm2 monit
```

使用SSH轻松部署

自动部署，避免逐个在所有服务器中进行ssh。

```
pm2 deploy
```

ref: PM2 offical document

浅谈Node.js的工作原理及优缺点

原文：https://blog.csdn.net/kaosini/article/details/8090510 编辑：reco

如果您听说过Node，或者阅读过一些文章，宣称Node是多么多么的棒，那么您可能会想："Node究竟是什么东西？"即便是在参阅Node的主页之后，您甚至可能还是不明白 Node为何物？Node肯定不适合每个程序员，但它可能是某些程序员一直苦苦追寻的东西。

Node旨在解决什么问题？

Node公开宣称的目标是"旨在提供一种简单的构建可伸缩网络程序的方法"。当前的服务器程序有什么问题？我们来做个数学题。在Java™和PHP这类语言中，每个连接都会生成一个新线程，每个新线程可能需要2MB的配套内存。在一个拥有8GBRAM的系统上，理论上最大的并发连接数量是4,000个用户。随着您的客户群的增长，如果希望您的Web应用程序支持更多用户，那么，您必须添加更多服务器。当然，这会增加服务器成本、流量成本和人工成本等成本。除这些成本上升外，还有一个潜在技术问题，即用户可能针对每个请求使用不同的服务器，因此，任何共享资源都必须在所有服务器之间共享。鉴于上述所有原因，整个Web应用程序架构（包括流量、处理器速度和内存速度）中的瓶颈是：服务器能够处理的并发连接的最大数量。

Node解决这个问题的方法是：更改连接到服务器的方式。每个连接发射一个在Node引擎的进程中运行的事件，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）。Node声称它绝不会死锁，因为它根本不允许使用锁，它不会直接阻塞 I/O 调用。Node还宣称，运行它的服务器能支持数万个并发连接。

现在您有了一个能处理数万个并发连接的程序，那么您能通过Node实际构建什么呢？如果您有一个Web应用程序需要处理这么多连接，那将是一件很"恐怖"的事！那是一种"如果您有这个问题，那么它根本不是问题"的问题。在回答上面的问题之前，我们先看看Node的工作原理以及它的设计运行方式。

Node肯定不是什么？

没错，Node是一个服务器程序。但是，基础Node产品肯定不像Apache或Tomcat。本质上，那些服务器"安装就绪型"服务器产品，支持立即部署应用程序。通过这些产品，您可以在一分钟内启动并运行一个服务器。Node肯定不是这种产品。Apache能通过添加一个PHP模块来允许开发人员创建动态Web页，添加一个SSL模块来实现安全连接，与此类似，Node也有模块

概念，允许向Node内核添加模块。实际上，可供选择的用于 Node的模块有数百个之多，社区在创建、发布和更新模块方面非常活跃，一天甚至可以处理数十个模块。本文后面将讨论Node的整个模块部分。

Node如何工作？

Node本身运行V8 JavaScript。等等，服务器上的JavaScript？没错，您没有看错。对于只在客户机上使用JavaScript的程序员而言，服务器端JavaScript可能是一个新概念，但这个概念本身并非遥不可及，因此为何不能在服务器上使用客户机上使用的编程语言？

什么是V8？V8 JavaScript引擎是Google用于其Chrome浏览器的底层JavaScript引擎。很少有人考虑JavaScript在客户机上实际做了些什么？实际上，JavaScript引擎负责解释并执行代码。Google使用V8创建了一个用C++编写的超快解释器，该解释器拥有另一个独特特征；您可以下载该引擎并将其嵌入任何应用程序。V8 JavaScript引擎并不仅限于在一个浏览器中运行。因此，Node实际上会使用Google编写的V8 JavaScript引擎，并将其重建为可在服务器上使用。太完美了！既然已经有一个不错的解决方案可用，为何还要创建一种新语言呢？

事件驱动编程

许多程序员接受的教育使他们认为，面向对象编程是完美的编程设计，这使得他们对其他编程方法不屑一顾。Node使用了一个所谓的事件驱动编程模型。

例一：

```
$("#myButton").click(function(){
    if ($("#myTextField").val() != $(this).val())
        alert("Field must match button text");
});
```

实际上，服务器端和客户端没有任何区别。没错，这没有按钮点击操作，也没有向文本字段键入的操作，但在一个更高的层面上，事件正在发生。一个连接被建立，这是一个事件；数据通过连接进行接收，这也是一个事件；数据通过连接停止，这还是一个事件！

为什么这种设置类型对Node很理想？JavaScript是一种很棒的事件驱动编程语言，因为它允许使用匿名函数和闭包，更重要的是，任何写过代码的人都熟悉它的语法。事件发生时调用的回调函数可以在捕获事件处进行编写。这样可以使代码容易编写和维护，没有复杂的面向对象框架，没有接口，没有过度设计的可能性。只需监听事件，编写一个回调函数，其他事情都可以交给系统处理！

示例Node应用程序

最后，我们来看一些代码！让我们将讨论过的所有内容汇总起来，从而创建我们的第一个Node应用程序。我们已经知道，Node对于处理高流量应用程序很理想，所以我们将创建一个非常简单的Web应用程序，一个为实现最快速度而构建的应用程序。下面是“老板”交代的关于我们的样例应用程序的具体要求：创建一个随机数字生成器RESTful API。这个应用程序应该接受一个输入：一个名为“number”的参数。然后，应用程序返回一个介于0和该参数之间的随机数字，并将生成的数字返回给调用者。由于“老板”希望该应用程序成为一个广泛流行的应用程序，因此它应该能处理50000个并发用户。我们来看看以下代码：

```
http.createServer(function(request, response) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    var params = url.parse(request.url, true).query;
    var input = params.number;
    var numInput = new Number(input);
    var numOutput = new Number(Math.random() * numInput).toFixed(0);
    response.write(numOutput);
    response.end();
}).listen(80);
console.log("Random Number Generator Running...");
```

启动应用程序

上面的代码放入一个名为“random.js”的文件中。现在，要启动这个应用程序并运行它（以便创建HTTP服务器并监听端口80上的连接），只需在您的命令提示中输入以下命令：% node random.js。下面是服务器已经启动并运行时看起来的样子：

```
# node random.js
Random Number Generator Running...
```

访问应用程序

应用程序已经启动并运行。Node正在监听所有连接，我们来测试一下。由于我们创建了一个简单的RESTful API，所以可以使用Web浏览器来访问这个应用程序。键入以下地址（确保您已完成了上面的步骤）：http://localhost/?number=27。

您的浏览器窗口将更改到一个介于0到27之间的随机数字。单击浏览器上的“重新载入”按钮，您会得到另一个随机数字。就是这样，这就是您的第一个Node应用程序！

Node对什么有好处？

到此为止，您可能能够回答“Node是什么”这个问题了，但您可能还有一个问题：“Node有什么用途？”这是一个需要提出的重要问题，因为肯定有些东西能受益于Node。

它对什么有好处？

正如您此前所看到的，Node非常适合以下情况：在响应客户端之前，您预计可能有很高的流量，但所需的服务器端逻辑和处理不一定很多。Node表现出众的典型示例包括：

1、RESTful API

提供RESTful API的Web服务接收几个参数，解析它们，组合一个响应，并返回一个响应（通常是较少的文本）给用户。这是适合Node的理想情况，因为您可以构建它来处理数万条连接。它仍然不需要大量逻辑；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。

2、Twitter队列

想像一下像Twitter这样的公司，它必须接收tweets并将其写入数据库。实际上，每秒几乎有数千条tweet达到，数据库不可能及时处理高峰时段所需的写入数量。Node成为这个问题的解决方案的重要一环。如您所见，Node能处理数万条入站tweet。它能快速而又轻松地将它们写入一个内存排队机制（例如memcached），另一个单独进程可以从那里将它们写入数据库。Node在这里的角色是迅速收集tweet，并将这个信息传递给另一个负责写入的进程。想象一下另一种设计（常规PHP服务器会自己尝试处理对数据库本身的写入）：每个tweet都会在写入数据库时导致一个短暂的延迟，因为数据库调用正在阻塞通道。由于数据库延迟，一台这样设计的机器每秒可能只能处理2000条入站tweet。每秒处理100万条tweet则需要500个服务器。相反，Node能处理每个连接而不会阻塞通道，从而能够捕获尽可能多的tweets。一个能处理50000条tweet的Node机器仅需20台服务器即可。

3、电子游戏统计数据

如果您在线玩过《使命召唤》这款游戏，当您查看游戏统计数据时，就会立即意识到一个问题：要生成那种级别的统计数据，必须跟踪海量信息。这样，如果有数百万玩家同时在线玩游戏，而且他们处于游戏中的不同位置，那么很快就会生成海量信息。Node是这种场景的一种很好的解决方案，因为它能采集游戏生成的数据，对数据进行最少的合并，然后对数据进行排队，以便将它们写入数据库。使用整个服务器来跟踪玩家在游戏中发射了多少子弹看起来很愚蠢，如果您使用Apache这样的服务器，可能会有一些有用的限制；但相反，如果您专门使用一个服务器来跟踪一个游戏的所有统计数据，就像使用运行Node的服务器所做的那样，那看起来似乎是一种明智之举。

Node模块

尽管不是本文最初计划讨论的主题，但应广大读者要求，本文已经扩展为包含一个 Node Modules和Node Package Manager简介。正如已经习惯使用Apache的开发人员那样，您也可以通过安装模块来扩展Node的功能。但是，可用于Node的模块极大地增强了这个产品，那些模块非常有用，将使用Node的开发人员通常会安装几个模块。因此，模块也就变得越来越重要，甚至成为整个产品的一个关键部分。

在“参考资料”部分，我提供了一个指向模块页面的链接，该页面列示了所有可用模块。为了展示模块能够提供的可能性，我在数十个可用模块中包含了以下几个模块：一个用于编写动态创建的页面（比如PHP），一个用于简化MySQL使用，一个用于帮助使用WebSockets，还有一个用来协助文本和参数解析的模块。我不会详细介绍这些模块，这是因为这篇概述文章旨在帮助您了解Node并确定是否需要深入学习（再次重申），如果需要，那么您肯定有机会用到这些可用模块。

另外，Node的一个特性是Node Package Module，这是一个内置功能，用于安装和管理Node模块。它自动处理依赖项，因此您可以确定：您想要安装的任何模块都将正确安装并包含必要的依赖项。它还支持将您自己的模块发布到Node社区，假如您选择加入社区并编写自己的模块的话。您可以将NPM视为一种允许轻松扩展Node功能的方法，不必担心这会破坏您的Node安装。同样，如果您选择深入学习Node，那么NPM将是您的Node解决方案的一个重要组成部分。

结束语

阅读本文之后，您在本文开头遇到的问题“Node.js究竟是什么东西?” 应该已经得到了解答，您应该能通过几个清晰简洁的句子回答这个问题。如果这样，那么您已经走到了许多程序员的前面。我和许多人都谈论过Node，但他们对Node究竟用于做什么一直很迷惑。可以理解，他们具有的是Apache的思维方式，认为服务器就是一个应用程序，将HTML文件放入其中，一切就会正常运转。由于大多数程序员都熟悉Apache及其用途，因此，描述Node的最简单方法就是将它与Apache进行比较。Node是一个程序，能够完成Apache能够完成的所有任务（借助一些模块），而且，作为一个可以将其作为基础进行构建的可扩展JavaScript平台，Node还能完成更多的任务。

从本文可以看出，Node完成了它提供高度可伸缩服务器的目标。它使用了Google的一个非常快速的JavaScript引擎，即V8引擎。它使用一个事件驱动设计来保持代码最小且易于阅读。所有这些因素促成了Node的理想目标，即编写一个高度可伸缩的解决方案变得比较容易。

与理解Node是 什么同样重要的是，理解它不是什么。Node并不只是Apache的一个替代品，它旨在使PHP Web应用程序更容易伸缩。事实远非如此。尽管Node还处于初始阶段，但它发展得非常迅速，社区参与度非常高，社区成员创建了大量优秀模块，一年之内，这个不断发展的产品就有可能出现在您的企业中。

Threads in Node 10.5.0: a practical intro

by Fernando Doglio 翻译：reco

几天前，Node.js的10.5.0版本已经发布，其中包含的一个主要功能是增加了初始（和实验性）线程支持。这很有趣，特别是来自一种总是它是非常棒的异步I / O引以为傲的语言，因此需要线程。那么为什么我们需要Node中的线程？

快速而简单的答案是：让Node在过去难受的区域中表现出色：处理繁重的*CPU*密集型计算。就是这个原因，导致Node.js在AI，机器学习，数据科学等领域表现并不出色。nodejs正在努力寻求解决这个问题，但仍然没有像部署微服务那样高效。

因此，我将尝试简化官方文档提供的技术文档，把它变成更实用，更简单的示例集。希望这足以让你开始。

那么我们如何使用新的**Threads**模块呢？

首先，您将需要名为“worker_threads”的模块。

请注意，这仅在执行脚本时使用--experimental-worker标志时才有效，否则将无法找到该模块。

注意标志如何引用工作者而不是线程，这是它们将在整个文档中引用的方式：工作线程或简单的工作者。

如果你过去使用过多处理，你会发现这种方法有很多相似之处，但如果你没有，请不要担心，我会尽可能多地解释。

你能用它们做什么？

工作线程就像我之前提到的那样，用于CPU密集型任务，将它们用于I / O将浪费资源，因为根据官方文档，Node提供的处理异步I / O的内部机制更多比使用工作线程更有效，所以...不要打扰。

让我们从一个简单的例子开始，你将如何创建一个工人并使用它。

Example 1:

```
    const { Worker, isMainThread,  workerData } = require('worker_threads');
    let currentVal = 0;
    let intervals = [100,1000, 500]
    function counter(id, i){
        console.log("[", id, "]", i)
        return i;
    }
    if(isMainThread) {
        console.log("this is the main thread")
        for(let i = 0; i < 2; i++) {
            let w = new Worker(__filename, {workerData: i});
        }
        setInterval((a) => currentVal = counter(a,currentVal + 1), intervals[2], "Main
Thread");
    } else {
        console.log("this isn't")
        setInterval((a) => currentVal = counter(a,currentVal + 1), intervals[workerDat
a], workerData);
    }
```

上面的例子将简单地输出一组显示增量计数器的行，这将使用不同的速度增加它们的值。

IF语句中的代码创建了2个工作线程，由于传递了 `__filename` 参数，因此它们的代码取自同一个文件。 工作人员现在需要文件的完整路径，他们无法处理相对路径，因此这就是使用此值的原因。

将2个worker作为全局参数发送，其形式为您在第二个参数中看到的workerData属性。 然后可以通过具有相同名称的常量访问该值（请参阅如何在文件的第一行中创建常量，并在稍后的最后一行中使用该常量）。

这个例子是你可以用这个模块做的最基本的事情之一，但它不是那么有趣，是吗？ 让我们看另一个例子。

Example 2: Actually doing something

```
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads');
const request = require("request");
if(isMainThread) {
    console.log("This is the main thread")
    let w = new Worker(__filename, {workerData: null});
    w.on('message', (msg) => { //A message from the worker!
        console.log("First value is: ", msg.val);
        console.log("Took: ", (msg.timeDiff / 1000), " seconds");
    })
    w.on('error', console.error);
    w.on('exit', (code) => {
        if(code != 0)
              console.error(new Error(`Worker stopped with exit code ${code}`))
   });
    request.get('http://www.google.com', (err, resp) => {
        if(err) {
            return console.error(err);
        }
        console.log("Total bytes received: ", resp.body.length);
    })
} else { //the worker's code
    function random(min, max) {
        return Math.random() * (max - min) + min
    }
    const sorter = require("./test2-worker");
    const start = Date.now()
    let bigList = Array(1000000).fill().map( (_) => random(1,10000))
    sorter.sort(bigList);
    parentPort.postMessage({ val: sorter.firstValue, timeDiff: Date.now() - start});
}
```

and test2-worker.js:

```
module.exports = {
    firstValue: null,
    sort: function(list) {
        let sorted = list.sort();
        this.firstValue = sorted[0]
    }
}
```

让我们现在尝试做一些“重”计算，同时在主线程中做一些异步的东西。

这一次，我们请求Google.com的主页，同时对随机生成的100万个数字进行排序。这将花费几秒钟，因此我们很好地展示了它的表现。我们还将测量工作线程执行排序所需的时间，然后我们将该值（连同第一个排序值）发送到主线程，我们将在其中显示结果。

这个例子的主要内容是演示线程之间的通信。

工作人员可以通过on方法在主线程中接收消息。我们可以听到的事件是代码上显示的事件。每当我们使用parentPort.postMessage方法从实际线程发送消息时，就会触发消息事件。您还可以在工作器实例上使用相同的方法向线程的代码发送消息，并使用parentPortobject捕获它们。

如果你想知道，我使用的辅助模块的代码就在这里，虽然没有什么值得注意的。

现在让我们看一个非常相似的例子，但是使用更清晰的代码，让您最终了解如何构建工作线程的代码。

Example 3: bringing it all together

作为最后一个例子，我将坚持使用相同的功能，但向您展示如何清理它并具有更易维护的版本。

```
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads');
const request = require("request");
function startWorker(path, cb) {
    let w = new Worker(path, {workerData: null});
    w.on('message', (msg) => {
        cb(null, msg)
    })
    w.on('error', cb);
    w.on('exit', (code) => {
        if(code != 0)
              console.error(new Error(`Worker stopped with exit code ${code}`))
   });
    return w;
}
console.log("this is the main thread")
let myWorker = startWorker(__dirname + '/workerCode.js', (err, result) => {
    if(err) return console.error(err);
    console.log("[[Heavy computation function finished]]")
    console.log("First value is: ", result.val);
    console.log("Took: ", (result.timeDiff / 1000), " seconds");
})
const start = Date.now();
request.get('http://www.google.com', (err, resp) => {
    if(err) {
        return console.error(err);
    }
    console.log("Total bytes received: ", resp.body.length);
    //myWorker.postMessage({finished: true, timeDiff: Date.now() - start}) //you could
 send messages to your workers like this
})
```

并且您的线程代码可以在另一个文件中，例如：

```
const {  parentPort } = require('worker_threads');
function random(min, max) {
    return Math.random() * (max - min) + min
}
const sorter = require("./test2-worker");
const start = Date.now()
let bigList = Array(1000000).fill().map( (_) => random(1,10000))
/**
//you can receive messages from the main thread this way:
parentPort.on('message', (msg) => {
    console.log("Main thread finished on: ", (msg.timeDiff / 1000), " seconds...");
})
*/
sorter.sort(bigList);
parentPort.postMessage({ val: sorter.firstValue, timeDiff: Date.now() - start});
```

分析一下可以得知：

主线程和工作线程现在将其代码放在不同的文件中。这更容易维护和扩展。 startWorker函数返回新实例，如果您愿意，可以稍后向其发送消息。 如果主线程的代码实际上是主线程（我们删除了主IF语句），您不再需要担心。 您可以在worker的代码中看到如何从主线程接收消息，从而允许双向异步通信。

请记住：这仍然是高度实验性的，这里解释的内容可能会在未来的版本中发生 去阅读PR评论和文档，有关于此的更多信息，我只关注它的基本步骤。 玩的开心！四处游玩，报告错误并提出改进建议，这刚刚开始！

ref@medium: Threads in Node 10.5.0: a practical intro by Fernando Doglio

Understanding the NodeJS cluster module

by Antonio Santiago http://www.acuriousanimal.com/ 翻译：reco

NodeJS进程在单个进程上运行，这意味着默认情况下它不会从多核系统中获益。如果你有一个8核CPU并通过$ node app.js运行NodeJS程序，它将在一个进程中运行，其余的CPU是浪费的。

幸运的是，NodeJS提供的集群模块，包含一组功能和属性可帮助我们创建充分使用所有CPU的程序。毫无疑问，集群模块用于最大化CPU使用率的机制是通过创建多个进程，类似于旧的fork（）系统调用Unix系统。

介绍集群模块

集群模块是一个NodeJS模块，它包含一组功能和属性，可帮助我们分配流程以利用多核系统。特别是如果您在HTTP服务器应用程序中工作

通过集群模块，主进程可以fork任意数量的工作进程，并与它们通过IPC通信发送消息进行通信。请记住，进程之间没有共享内存。

接下来的行是NodeJS文档中的句子汇编。我复制它们过来，以便可以帮助你理解整个事情的运作方式。

1. Node.js的单个实例运行起来时是在单线程的。为了利用多核系统，用户有时会想要启动一个Node.js进程集群来处理负载。
2. 群集模块允许轻松创建所有共享服务器端口的子进程。
3. 使用child_proces.fork()方法生成worker（子）进程，以便它们可以通过IPC与父进行通信并来回传递服务器句柄。 child_process.fork()专门用于生成新的Node.js进程。返回的ChildProcess将内置一个额外的通信通道，允许通过send()方法在父节点和子节点之间来回传递消息。有关详细信息，请参阅subprocess.send()。
4. 请务必记住，生成的Node.js子进程独立于父进程，但两者之间建立的IPC通信通道除外。每个进程都有自己的内存，并有自己的V8实例。由于需要额外的资源分配，因此不建议生成大量子Node.js进程。

因此，大多数魔法都是由child_process模块完成的，该模块可以生成新进程并帮助它们之间进行通信，例如，创建管道。你可以在Node.js Child Processes找到一篇很棒的文章：你需要知道的一切。

A basic example

啰嗦完毕，让我们看到一个眞正的例子。接下来我们展示一个非常基本的代码，它可以创建一个主进程，用于检索CPU的数量并为每个CPU分配一个工作进程，并且每个子进程在控制台中打印一条消息并退出。

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;
if (cluster.isMaster) {
  masterProcess();
} else {
  childProcess();
}
function masterProcess() {
  console.log(`Master ${process.pid} is running`);
  for (let i = 0; i < numCPUs; i++) {
    console.log(`Forking process number ${i}...`);
    cluster.fork();
  }
  process.exit();
}
function childProcess() {
  console.log(`Worker ${process.pid} started and finished`);
  process.exit();
}
```

将代码保存在app.js文件中并运行执行：$ node app.js. 输出应类似于：

```
$ node app.js
```

输出的样子：

```
Master 8463 is running
Forking process number 0...
Forking process number 1...
Forking process number 2...
Forking process number 3...
Worker 8464 started and finished
Worker 8465 started and finished
Worker 8467 started and finished
Worker 8466 started and finished
```

Code explanation

当我们运行app.js程序时，会创建一个开始运行我们代码的操作系统进程。在开始时，导入 `const cluster = require('cluster')` 并在if语句中检查isMaster属性。

因为进程是第一个进程，所以isMaster属性为true，然后我们运行masterProcess函数的代码。此函数没有太大的秘密，它根据您的机器的CPU数量循环，并使用cluster.fork()方法分叉当前进程。

fork()眞正做的是创建一个新的Nodejs进程，就像你通过命令行使用$ node app.js运行它一样，就是你有很多进程运行你的app.js程序。

创建并执行子进程时，它与主进程相同，即导入集群模块并执行if语句。一旦与子进程存在差异，cluster.isMaster的值为false，因此它们结束运行childProcess函数。

注意，我们使用process.exit()显式终止master和worker进程，默认情况下返回零值。

Comunicating master and worker processes

创建工作进程时，在工作者和主服务器之间创建IPC通道，允许我们使用send（）方法在它们之间进行通信，该方法接受JavaScript对象作为参数。 请记住，它们是不同的进程，因此我们不能使用共享内存作为通信方式。

在主进程中，我们可以使用进程引用（即someChild.send（{...}））向工作进程发送消息，并且在工作进程中，我们可以使用当前进程引用向主进程发送消息，即process.send（）。

我们已经更新了以前的代码，以允许主进程向工作进程发送和接收消息，工作人员也可以从主进程接收和发送消息：

```
function childProcess() {
  console.log(`Worker ${process.pid} started`);
  process.on('message', function(message) {
    console.log(`Worker ${process.pid} recevies message '${JSON.stringify(message)}'`)
;
  });
  console.log(`Worker ${process.pid} sends message to master...`);
  process.send({ msg: `Message from worker ${process.pid}` });
  console.log(`Worker ${process.pid} finished`);
}
```

工作进程只是为了理解。 首先，我们使用process.on（'message'，handler）方法监听注册侦听器的消息事件。 稍后我们使用process.send（{...}）发送消息。 请注意，该消息是一个纯JavaScript对象。

```
let workers = [];
function masterProcess() {
  console.log(`Master ${process.pid} is running`);
  // Fork workers
  for (let i = 0; i < numCPUs; i++) {
    console.log(`Forking process number ${i}...`);
    const worker = cluster.fork();
    workers.push(worker);
    // Listen for messages from worker
    worker.on('message', function(message) {
      console.log(`Master ${process.pid} recevies message '${JSON.stringify(message)}'
 from worker ${worker.process.pid}`);
    });
  }
  // Send message to the workers
  workers.forEach(function(worker) {
    console.log(`Master ${process.pid} sends message to worker ${worker.process.pid}..
.`);
    worker.send({ msg: `Message from master ${process.pid}` });
  }, this);
}
```

masterProcess函数分为两部分。 在第一个循环中，我们分配的工作量与我们拥有的CPU一样多。 cluster.fork（）返回表示工作进程的工作对象，我们将引用存储在数组中并注册一个侦听器以接收来自该工作器进程实例的消息。

稍后，我们遍历工作进程数组并从主进程向该具体工作程序发送消息。

如果您运行代码：

```
$ node app.js
```

输出将类似于：

```
Master 4045 is running
Forking process number 0...
Forking process number 1...
Master 4045 sends message to worker 4046...
Master 4045 sends message to worker 4047...
Worker 4047 started
Worker 4047 sends message to master...
Worker 4047 finished
Master 4045 recevies message '{"msg":"Message from worker 4047"}' from worker 4047
Worker 4047 recevies message '{"msg":"Message from master 4045"}'
Worker 4046 started
Worker 4046 sends message to master...
Worker 4046 finished
Master 4045 recevies message '{"msg":"Message from worker 4046"}' from worker 4046
Worker 4046 recevies message '{"msg":"Message from master 4045"}'
```

这里我们不是用process.exit（）来终止进程，所以要关闭你需要使用ctrl + c的应用程序。

Conclusion

集群模块为NodeJS提供了使用CPU全部能力而所需的功能。虽然在这篇文章中没有看到，但是群集模块补充了子进程模块，该模块提供了大量工具来处理进程：启动，停止和管道输入/输出等。集群模块允许我们轻松创建工作进程。 此外，它神奇地创建了一个IPC通道，用于传递传递JavaScript对象的主进程和工作进程。

Clusted HTTP

集群模块允许我们提高多核CPU系统中应用程序的性能。无论是使用API还是基于ExpressJS的Web服务器，这都非常重要，我们希望利用我们的NodeJS应用程序运行的每台机器上的所有CPU。集群模块允许我们在一组工作进程之间对传入请求进行负载均衡，并因此提高了应用程序的吞吐量。

在这篇文章中，我们将看到如何在创建HTTP服务器时使用集群模块，使用普通的HTTP模块和ExpressJS.Lets，看看我们如何创建一个真正基本的HTTP服务器来获取集群模块的利润。

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;
if (cluster.isMaster) {
  masterProcess();
} else {
  childProcess();
}
function masterProcess() {
  console.log(`Master ${process.pid} is running`);
  for (let i = 0; i < numCPUs; i++) {
    console.log(`Forking process number ${i}...`);
    cluster.fork();
  }
}
function childProcess() {
  console.log(`Worker ${process.pid} started...`);
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end(`Hello World from ${process.pid}`);
  }).listen(3000);
}
```

我们将代码分为两部分，一部分对应于主进程，另一部分是初始化工作进程的部分。 这样，masterProcess函数就每个CPU代码分配一个工作进程。 另一方面，childProcess只是在端口3000上创建一个HTTP服务器侦听器，并返回一个带有200状态代码的漂亮的Hello World文本字符串。

如果运行代码，输出必须显示如下内容：

```
$ node app.js
```

输出：

```
Master 1859 is running
Forking process number 0...
Forking process number 1...
Forking process number 2...
Forking process number 3...
Worker 1860 started...
Worker 1862 started...
Worker 1863 started...
Worker 1861 started...
```

从客户端访问的效果：

```
$ curl http://localhost:3000/
Hello World from 7036
$ curl http://localhost:3000/
Hello World from 7037
$ curl http://localhost:3000/
Hello World from 7038
$ curl http://localhost:3000/
Hello World from 7039
```

基本上，我们的主进程正在为每个CPU运行一个新的工作进程，该进程运行处理请求的HTTP服务器。正如您所看到的，这可以提高您的服务器性能，因为有一个处理一百万个请求而不是四个进程参与这些请求是不一样的。

How cluster module works with network connections ?

前面的例子很简单，但隐藏了一些棘手的东西，一些神奇的特性简化了我们作为开发人员的生活。

在任何操作系统中，进程都可以使用端口与其他系统进行通信，这意味着，给定端口只能由该进程使用。那么，问题是，工作进程如何使用相同的端口？

答案是主进程是在给定端口中侦听并在所有子进程/工作进程之间对请求进行负载均衡的进程。从官方文档：

> 使用child_process.fork（）方法生成工作进程，以便它们可以通过IPC与父进程通信并来回传递服务器句柄。

群集模块支持两种分发传入连接的方法。

第一个（除了Windows之外的所有平台上都是默认的）是轮询方法，其中主进程侦听端口，接受新连接并以轮询方式在工作者之间分配它们，其中一些是内置的智慧以避免工作进程过载。

第二种方法是主进程创建侦听套接字并将其发送给感兴趣的工作者。然后工作进程直接接受传入的连接。

只要有一些工作进程还活跃着，服务器就会继续接受连接。如果没有工作进程，则将删除现有连接，并拒绝新连接。

群集模块负载平衡的其他替代方法

集群模块允许主进程接收请求并在所有工作进程之间对其进行负载平衡。这是一种提高性能的方法，但它不是唯一的方法。

在Node.js后的进程中，负载均衡性能：比较集群模块，iptables和Nginx，您可以找到以下性能比较：节点集群模块，iptables和nginx反向代理。

结论

如今，任何Web应用程序都必须具有性能，我们需要支持高吞吐量并快速提供数据。

集群模块是一种可能的解决方案，它允许我们拥有一个主进程并为每个核心创建一个工作进程，以便它们运行HTTP服务器。群集模块提供两个强大功能：

通过创建IPC通道并允许使用process.send（）发送消息，简化主服务器和工作服务器之间的通信， 允许工作进程共享同一个端口。这样做使得主进程成为接收请求并在工作者之间多路复用的进程。

Using PM2 to manage NodeJS cluster

集群模块允许我们创建工作进程以提高NodeJS应用程序的性能。这在Web应用程序中尤其重要，其中主进程接收所有请求并在工作进程之间对它们进行负载平衡。

但所有这些功能都伴随着管理与流程管理相关的所有复杂性的应用程序所需的成本：如果工作进程意外存在，工作进程如何正常退出，如果需要重新启动所有工作人员等等，会发生什么？ 。

在这篇文章中，我们介绍了PM2工具。虽然它是一个通用的流程管理器，但这意味着它可以管理任何类型的流程，如python，ruby 等，而不仅仅是NodeJS流程，该工具专门用于管理想要使用群集模块的NodeJS应用程序。

介绍PM2

如前所述，PM2是一个通用流程管理器，即控制其他流程执行的程序（如检查您是否有新电子邮件的python程序），并执行以下操作：检查您的流程是否正在运行，重新执行你的过程如果由于某种原因意外退出，记录其输出等。

对我们来说最重要的是PM2简化了NodeJS应用程序的执行，以便作为集群运行。是的，您编写应用程序而不必担心群集模块，并且PM2创建了一定数量的工作进程来运行您的应用程序。

PM2方式

在继续之前，您需要在系统上安装PM2。通常它安装为全局模块，使用

$ npm install pm2 -g

当使用PM2时，我们可以忘记与主进程相关的代码部分，这将是PM2的责任，所以我们非常基本的HTTP服务器可以重写为：

```
const http = require('http');
console.log(`Worker ${process.pid} started...`);
http.createServer((req, res) => {
  res.writeHead(200);
  res.end('Hello World');
  process.exit(1);
}).listen(3000);
```

选择这样运行起来：

```
$ pm2 start app.js -i 3
```

请注意选项-i，用于指示要创建的实例数。 想法是该数字与您的CPU核心数相同。 如果您不了解它们，可以设置-i 0让PM2自动检测到它。

结论

虽然NodeJS集群模块是一种提高性能的强大机制，但它需要管理应用程序可以找到的所有情况所需的复杂性：如果工作者存在会发生什么，我们如何在没有停机的情况下重新加载应用程序集群等等。PM2是一个专门设计用于NodeJS集群的流程管理器。 它允许集群应用程序，重新启动或重新加载，除了提供工具以查看日志输出，监视等外，还没有所需的代码复杂性。

Graceful shutdown NodeJS HTTP server when using PM2

所以你创建了一个收到大量请求的NodeJS服务器，你真的很开心，但是，就像每一个软件一样，你发现了一个bug或者为它添加了一个新功能。 很明显，您需要关闭NodeJS进程并重新启动，以便进行新代码。 问题是：如何以优雅的方式实现这一点，以便继续提供传入的请求？

启动HTTP服务器

在了解我们必须如何关闭HTTP服务器之后，让我们看看通常如何创建一个。 下一个代码显示了一个带有ExpressJS服务的非常基本的代码，它将返回Hello World！ 访问/ hello路径时。 你也可以传递一个路径参数，即/ hello / John的名字，这样它就会返回Hello John !!!

```
const express = require('express')
const expressApp = express()
// Responds with Hello World or optionally the name you pass as path param
expressApp.get('/hello/:name?', function (req, res) {
  const name = req.params.name
  if (name) {
    return res.send(`Hello ${name}!!!`)
  }
  return res.send('Hello World !!!')
})
// Start server
expressApp.listen(3000, function () {
  console.log('App listening on port 3000!')
})
```

app.listen（）函数的作用是使用核心http模块启动一个新的HTTP服务器，并返回对HTTP服务器对象的引用。 具体来说，listen（）的源代码如下：

```
app.listen = function listen() {
  var server = http.createServer(this);
  return server.listen.apply(server, arguments);
};
```

注意：创建快速服务器的另一种方法是将我们的expressApp引用直接传递给http。 createServer（），类似于：const server = http.createServer（app）.listen（3000）。

How to shutdown properly an HTTP server ?

关闭HTTP服务器的正确方法是调用server.close（）函数，这将阻止服务器接受新连接，同时保留现有连接直到响应它们。

下一代码提供了一个新的/关闭端点，一旦被调用将停止HTTP服务器并退出应用程序（停止nodejs进程）：

```
app.get('/close', (req, res) => {
  console.log('Closing the server...')
  server.close(() => {
    console.log('--> Server call callback run !!')
    process.exit()
  })
})
```

很明显，通过端点关闭服务器不是正确的方法。

Graceful shutdown/restart with and without PM2

优雅关闭的目标是关闭到服务器的传入连接，但不会杀死我们正在处理的当前连接。

当使用像PM2这样的流程管理器时，我们管理一个进程集群，每个进程都充当HTTP服务器。PM2实现平稳重启的方式是：

1. 向每个工作进程发送SIGNINT信号，
2. 工作进程负责捕获信号，清理或释放任何使用过的资源并完成其过程，
3. 最后PM2产生了一个新的进程

因为这是按照我们的集群流程顺序完成的，所以客户不能受到重启的影响，因为始终会有一些进程在工作并参与请求。

当我们部署新代码并希望重新启动服务器以便新的更改生效而不会有传入请求的风险时，这非常有用。 我们可以在app中实现这个下一个代码：

```
// Graceful shutdown
process.on('SIGINT', () => {
  const cleanUp = () => {
    // Clean up other resources like DB connections
  }
  console.log('Closing server...')
  server.close(() => {
    console.log('Server closed !!! ')
    cleanUp()
    process.exit()
  })
  // Force close server after 5secs
  setTimeout((e) => {
    console.log('Forcing server close !!!', e)
    cleanUp()
    process.exit(1)
  }, 5000)
})
```

当它捕获的SINGINT信号时，我们调用server.close（）以避免接受更多请求，一旦它关闭，我们清理我们的应用程序使用的任何资源，如关闭数据库连接，关闭打开的文件等，调用cleanUp（）函数 最后，我们使用process.exit（）退出进程。 此外，如果由于某种原因我们的代码花费太多时间来关闭服务器，我们强制它在setTimeout（）中运行非常相似的代码。

Conclusions

在创建HTTP服务器时，无论是服务于页面还是API的Web服务器，我们都需要考虑到它需要及时更新以及需要错误修复的事实，因此我们需要思考如何可以做到对客户的以最小化的影响。

在集群模式下运行nodejs进程是提高应用程序性能的常用方法，我们需要考虑如何正常关闭所有这些进程以不影响传入请求。

使用process.exit（）终止节点进程在使用HTTP服务器时是不够的，因为它会突然终止所有通信，我们需要先停止接受新连接，释放我们的应用程序使用的任何资源，最后停止整个进程。

ref: Understanding the NodeJS cluster module

by Antonio Santiago http://www.acuriousanimal.com/

对Node.js创造者的访问

作者：mappingthejourney.com 翻译：Reco

Ryan Dahl是Google Brain的一名软件工程师。 他是Node.js的创建者。Node.js是基于Chrome的V8 JavaScript引擎构建的JavaScript运行时。 目前，他正致力于深入学习深度学习研究项目。 他的重点主要是图像到图像的变换，如着色和超分辨率。 他为几个开源项目做出了贡献，包括HTTP Parser，libuv。

普拉莫德：大家好。 欢迎来mappingthejourney旅程。 当我们听说Node.js时，我们也听说了Ryan Dahl。 他告诉我们，我们正在使用的I / O方法是完全错误的，并教会我们如何使用异步编程模型构建软件。 今天的客人是Ryan Dahl本人，一个黑客，杰出的程序员和Node的创造者。 让你参加这个节目让我感到非常兴奋和荣幸。 欢迎Ryan

Ryan：你好！ 很高兴在那里.....在这里。

普拉莫德：Ryan我们知道你是Node的创造者，告诉我们你在搞技术之前之前的生活吗？

Ryan：当然。 我在圣地亚哥长大; 我六岁的时候，我的妈妈有一台Apple 2C，所以我想我早就可以使用电脑了。 顺便说一下，我才36岁。 所以，就像互联网出现一样，我成年了。 我去了圣地亚哥的社区学院，然后去了加州大学圣地亚哥分校，在那里我学习数学。 然后，我去了罗切斯特大学的数学研究生院。

是啊。 在那里，我研究了代数拓扑，这是一个非常抽象的主题，我发现它几年非常漂亮，但后来我厌倦了它，因为它似乎不适用于现实生活。 毕业后，好吧.....那就是博士学位。 程序。 一旦我意识到我不想成为我余生的数学家，我就退出了那个项目。 并购买了一张前往南美洲的单程机票并在那里待了一年，在那里我有点挨饿的学生模式，并找到了一个与这个人做一些网站的工作，Eric。 这就是我的编程生涯开始的方式。 这个公司当时正在为一个滑雪板公司开发Ruby on Rails网站。

普拉莫德：太好了！ 这必须是退出博士学位的理由。 计划，前往南美洲并作为Web Dev工作。

Ryan：是的。 我的意思是，所以...来自研究生院，你习惯于处理非常抽象的问题，而在网站上工作则是一个非常具体的过程。 但我试图将它变成一个美丽的数学理论，就像我在研究生院接触过的那样。 而且我认为让我思考...我想我更喜欢Ruby开发，我想，你可以在Ruby中更清楚地表达你的想法。 这很有意思。 我认为Rails在这方面令人印象深刻。 它给了这个新的搭建软件的结构，当然实际上它并不是全新的，但我认为Rails推广了模型视图控制器的结构。而且你知道，这两件事结合在一起，对我来说真的很有趣。

普拉莫德 ： 是的，构建Web应用程序非常有趣。 Ruby是一个完美的工具。 接下来，您继续在德国担任自由网络开发人员。 您参与的其中一个项目是Node。 我想你会在接下来的六到八个月内继续研究它吗？

Ryan：对。 因此，在离开南美之后，我和我的女朋友搬到了德国，因为她是德国人，不得不重返大学。 我开始在那里参加Ruby会议，人们在谈论这种新的模型视图控制器范例。 如果我正确地宣布这一点，其中一个人就是Chris Neukirchen。 他开发了名为Rack的项目，它是Web服务器的简化抽象。 因此，它将Web服务器转换为单一功能接口，您可以在其中获得请求，然后返回响应。

结合我自由职业时期在Nginx模块上为Engineyard做的一些，让我想到了.....让我退后一步。 在Nginx中，一切都是异步的。 因此，当您为它构建模块时，必须非常小心地进行非阻塞。 是的，我认为Chris Neukirchen的Rack，加上Nginx的网络服务器与非阻塞IO，两者结合起来，可能是非常很有趣的一件事。

普拉莫德 ： 现在你有了Rack和Nginx的想法。“你是如何说服自己的，好吧我将在接下来的6个月内构建可以在服务器端运行Javascript的框架，这可能会大大提高性能？

Ryan：那么，那两个简化的Web服务器接口，就是Rack，以及异步部分，就是Nginx，我一直在考虑。 然后Chrome于2008年12月发布。随之而来的是V8 Javascript解释器。 我不应该说，解释器。 这是一个JIT运行时。 所以，当V8出来的时候，我开始用它来探索它，它看起来很迷人，干净，而且很快，突然间，我灵光一闪：哦！ Javascript是单线程的，每个人都在做非阻塞。

我用手指做了一个比划(Air quotes)，但就像在网络浏览器中一样，人们在制作AJAX请求时会发出非阻塞请求。 我想：哦，哇！ 我认为JavaScript加异步IO加上一些HTTP服务器的东西会很酷。 我对这个想法感到非常兴奋，因为我在接下来的四年里不停地努力。

普拉莫德 ： 是的Javascript加async i / o工作得非常好。 我相信开发人员正在等待看到这样做的框架。 而且好奇，在这段时间里，有没有任何导师，或者你曾与任何人协商过“？ 还是只是你？

Ryan：只是我一个人。 我有一些编程的朋友给了建议，当然.....我的意思是，第一点就是我在我的房间里。 但后来，我最终搬到旧金山，在Joyent工作，并结识了许多非常优秀的编程专业人士。 是的，很多人在那之后指导并提出了对Node做出贡献的想法。

普拉莫德：很好。 带我们完成您开发Node的过程。 我知道自从你在2009年创建Node以来，Ryan，已经很久了。

Ryan：我认为至少对我自己而言，生命中没有什么比我沉浸于我眞正相信的想法更快乐的事情。并且有时间坐下来继续努力做到。 而且我认为Node是一个等待变成现实的想法，并且我还没有做过; 我不做的话，别人会去做的。 但事实恰恰相反，我刚好处于失业状态，并有一些空闲时间，可以连续工作几个月，这正是一个初创产品所需要的。 所以，是的，这很棒，很有趣。

普拉莫德 ：很好。 这太妙了。 你做得很好。 Node建立在“纯异步”编程模型的基础之上。 这个想法是如何为Nodejs制定出来的？

Ryan：是的，我认为这是一个非常有趣的问题。 现在，已经有好几年了，自从2012年或2013年以来我一直没有在Node上工作。当然，Node在这一点上是一个很大的项目。 所以，是的，我想.....当它第一次出现时，我四处奔走并进行了一系列谈话，试图说服人们应该这样做。 也许我们正在做的I / O方式是错误的，也许如果我们以非阻塞方式完成所有事情，我们就可以解决编程方面的许多困难。 也许我们可以完全忘记线程，只使用流程抽象和序列化通信。 但是在一个进程中，我们可以完全异步处理许多请求。 我当时坚信这个想法，但在过去的几年里，我认为这可能不是编程的最终目的。 特别是当Go出来的时候。

嗯，我想Go很久以前就出来了，但是当我第一次听到关于Go的时候，大约是2012年。他们有一个非常好的运行时，有适当的绿色线程和易于使用的抽象。 我认为阻塞I / O - 再次阻止引号中的I / O，因为它在Go和操作系统之间的接口处都是绿色线程，我认为它都是非阻塞I / O.

但是它们呈现给用户的接口是阻塞的，我认为这是一个更好的编程模型。 如果它阻塞，你可以更容易地思考你在很多情况下所做的事情。 你知道，如果你有一堆以下的行动，那么能够说：做事A，等待回应，可能是出错，这很好。 做事B，等待回复，错误出来。 在Nodejs中，这更难，因为你必须跳转到另一个函数调用。

普拉莫德 ：是的，我喜欢Go的编程模型。 使用goroutines非常简单有趣。 实际上，我们正在使用工作来构建分布式应用程序。

Ryan：是的，我认为这是...对于特定类别的应用程序，就像，如果你正在构建一个服务器，我无法想象使用Go以外的任何东西。 也就是说，我认为Node的非阻塞范式对于没有线程的JavaScript来说效果很好。 而且我认为很多回调地狱的问题，你必须跳进许多匿名函数来完成你正在做的事情，在如今这个问题已经减轻，使用async关键字，async功能现在已经是Javascript的一部分。所以，Javascript的新版本使这异步编码变得更容易。 也就是说，我认为Node不是构建大规模服务器网络的最佳系统。 我会用Go来做到这一点。 老实说，这就是我离开Node的原因。 GO让我意识到：哦，实际上，这不是有史以来最好的服务器端系统。

是啊。 我认为Node在客户端的红红火火令我感到很奇怪。 就像是在构建网站时做一些脚本。 例如，Browserify。 一种打包客户端Javascript的工具。 因此，您可以使用客户端Javascript完成这些服务器端处理。 然后，你知道，可能是小型服务器.....也许是小型开发服务器，而且在这里和那里，也许有一些眞正的服务器提供实时流量。 Node.js可能很有用，或者它可能是正确的选择。 但是，如果您正在构建一个大规模分布式DNS服务器，我不会选择Node。

普拉莫德 ：对于全世界的所有开发者来说，这应该是一个很好的选择。 选择合适的应用工具非常重要。 你对Node没有任何偏见。 您在JsConf 2009 Berlin中向世界介绍了Node.js. 它突然收获的成功和表现出来的张力会让你感到惊讶吗？

Ryan：是的。 我的意思是，我基本上处于连续四年的惊喜状态。 因为它变得非常快，人们非常喜欢它。 嗯是的。

普拉莫德 ：此后你加入了Joyant，并在Node上全职工作，你搬到了SF吧？ 经历如何？ 开发人员喜欢它，你就是这一切的中心。

Ryan：当然，这是一次一生的经历，我觉得这是一切的中心，或者在大会的时候还是不在大会的时候都是如此。 有一次，我去了日本，人们要求和我一起拍照，我意识到.....我不知道; 我觉得有点奇怪。 同样的，当我在线的时候，我想在那个时候，我觉得每当我评论一些东西时，我会得到100多个响应。

所以，我发现我必须非常谨慎地选择我的话，以及我是如何展示自己的，因为看起来人们在听，这很奇怪。 而且我不喜欢这样。我的意思是，我是一名程序员，我想编写代码，有时会分享我的意见，而不必过于仔细考虑。 所以，我想我不是一个.....是的。 我没那么喜欢它的那个方面。

普拉莫德 ：当你创造和介绍Node时，你是29,30岁？ 而Node.js产生了如此巨大的影响。

Ryan： 是的。 我的意思是，那时我肯定是一个新手开发者。

普拉莫德：好的。 Ryan，同时有很多服务器端的JavaScript项目。 Node.js不是唯一的选择。 您认为Node的成功是什么？

Ryan：对。 有几个人试图让服务器端的JavaScript事情发生。 我甚至不能再列举它们了，我忘了它们是什么。 好吧，无论如何。

问题是他们都在做阻塞式的I / O，但是由于你没有线程而没有Javascript的结构。 因此，如果您正在阻止I / O，那么您实际上无法提供请求。 就像，你一次做一个，这是永远不会工作的。 那，加上我喜欢的几个事实，我就可以坐下来让HTTP服务器运行良好。 所以，我有一个演示，你可以.....我有一个HTTP服务器，然后是一个原始的TCP服务器。 而且我让这些东西运作良好，以便人们可以坐下来建立一个网站，而不会经历太多的麻烦。

老实说，构建一个Web服务器并不是最简单的事情，我认为很多这些系统都留给他们的社区来做，因此，没有人这样做。 因为没有什么可以使用系统。 我认为重要的是，当您发布软件框架或任何软件时，您有一个坐下来，很快的完成一个可以立即使用的演示。 这是Node的做的最佳选择之一;，是人们可以下载它并直接使用Web服务器。

普拉莫德 ：是的。 好的演示以及如果人们可以轻松下载，安装和使用它会产生很大的不同。 此外，人们知道javascript，他们可以立即开始编码。 当我开始在Node.js上工作时，它就更容易了，因为我很熟悉javascript。

Ryan：是的。 我认为我们理所当然地认为在语言之间切换是多么容易。 我的意思是，即使你知道另一种语言，要创建一个良好的使用的上下文依然是非常困难的。 而且有很多人非常熟悉Javascript。 给他们这些工具，以便能够在其他环境中使用它激发人们的兴趣。 有了它，你突然可以比以前做得更多。

普拉莫德 ： 是的。 2012年，Node.js已经拥有庞大的开发人员基础。 你为什么要离开，将控制权交给Joyent的Isaac Schlueter？

Ryan：是的。 对。 所以，我的意思是，我认为这是几件事的结合。 所以，我认为最重要的是，那时，我已经在Node上工作了四年。 我已经达到了我想要的程度。 我从不希望Node成为一个庞大的API。 我希望它是一种小型，紧凑的核心，人们可以在其上构建模块。

还有一些主要的东西.....我想要支持的关键功能。 因此，早期增加了扩展模块; 我们得到了所有网络库，HTTP，UDP，TCP，我们可以访问所有文件系统。

然后，有一大块，可能是一年的工作，差不多五个人，正在把它移植到Windows上。 我们希望将Windows对异步IO的抽象，也就是 `IO完成端口` 也利用起来。而这需要重写核心库，其结果是就是我们看到的libuv库。

是的，但在某些时候，所有这一切都已完成，我们已经在Windows上发布了。 而且你知道，它就像是这样：好吧。 我的意思是，这是我打算创造的，我很高兴我有机会完成它。 当然，你知道，我的余生将会需要花费大量时间去解决无数的错误，但是.....你知道，有足够多的人参与其中。 我不需要这样做，我想做其他事情。 加上Go出来的事实，我没有看到Node是服务器的终极解决方案。 然后，是的，每当我发表博客文章时，也只是不想成为关注的焦点。

普拉莫德： 很好。 是的，有些人不喜欢被人们所关注。 当你开始使用Node时，你肯定有一些目标。 今天Node.js如何与它相匹配呢？

Ryan ： 我的意思是.....Node.js被成千上万的人使用，如果不是数百万人，我认为它肯定超出了我认为会发生的任何期望。 是的，很酷。

普拉莫德 ： Ryan在你与Node的精彩旅程之后，你决定做什么工作？

Ryan：所以，在Node之后，我搬了.....在我离开Joyent并退出Node项目之后，我搬到了纽约，并花了一些时间来完成我的项目。 所以，我有一堆项目。 你知道，当时Instagram已经问世了，它很新，而且看起来很简单，而且每个人都在说：哇，这太简单了，我可以做一个。 我忍不住想出同样的事情。 所以，我有一个社交网络项目; 我有一个用于C ++的构建系统项目，我有另一个用于HTML的构建系统项目，有点像Browserify，它可以打包你的Javascript和HTML，但是更聪明。

是的，我有一堆项目，其中没有一个项目在我的脑海中真正实现了。 虽然我认为其中一些目前仍然处于困境，就像我的社交网络项目一样，我将在某个时候回到这个项目。 是的，我这样做了一段时间。 然后我开始阅读...好吧，我开始听到卷积网络和图像分类如何解决，这让我对机器学习感兴趣。

普拉莫德 ：您也是Google Brain‘sResidency计划的一部分。 那段经历怎么样？

Ryan：是的。 所以，我在山景城度过了一年。 所以，回溯下过去。 所以，TensorFlow两年前发布了。

随之而来的是，他们宣布了这项Google Brain‘sResidency计划，他们邀请了20人来到Google Brain，这是Google的机器学习研究实验室之一。 而且人们.....我认为用它来创作的想法并不一定是那些曾经学过机器学习但是有一些数学和编程背景的人，并且对机器学习感兴趣，喜

欢和喜欢这些新想法。 因为机器学习正在快速变化，并且已经完成了大量的工作。

但是现在社区已经将神经网络缩小为最有用的机器学习算法，这可能只是引入了一大堆人并且只是来上个手，而这个新的ML框架，称为TensorFlow，会导致一些有趣的想法。 所以，是的，我在那里度过了一年，基本上编程模型和撰写关于这些模型的论文。 我主要处理图像到图像转换问题。 所以，你知道，如果你有一些输入图像，你想要预测一些输出图像。 我发现这个问题很有意思，因为，例如.....让我举一些例子。

嗯嗯，着色问题。 您可以将黑白照片作为输入，您可以尝试将图片的颜色预测为输出。 这个问题很酷的是有无限的训练数据。 您可以拍摄任何彩色照片并对其进行去饱和，然后这就是您的输入图像，对吧？

因此，机器学习的一个问题是您需要大量数据，而对于这类任务，这个问题不是问题。 而且，最近在生成模型中已经做了很多工作，即输出图像的模型。 特别是，有生成对抗性网络和像素CNN，它已经证明了学习自然图像的多样性的能力，就像真正理解真实图像是什么，什么不是实际图像，看起来像是什么真实的形象。

所以，是的，我的想法是将最近的工作放在生成模型中并采用这种无限的训练数据，看看我们是否可以做一些图像转换问题。 所以，我做了一些关于超分辨率的工作，它采用低分辨率图像并提高分辨率。 这也是图像到图像转换问题。 我现在已经完成了两个关于着色的项目。

普拉莫德：很好的解释Ryan。 是的，我已经读过，tensor-flow是许多机器学习问题的一个很好的平台。 图像分类，转换，我没有得到太多，但我相信它是迷人的。 你在继续你的ML工作吗？

Ryan：对。 所以现在，作为一名软件工程师，我仍然在Google工作，处理同样的问题。 研究生成模型并试图帮助研究人员构建下一代系统，下一代模型。

普拉莫德：很好的生成模型，与之前的javascript，Node.js或Web开发工作有很大的不同。

Ryan： 是的，我想是的。 但我也从数学开始，所以我有一个相当不错的数学知识基础，我想。 是的，我猜...我认为人们喜欢将其他人折叠到某些区域，我不想那样做。 我不想成为一个Javascript人，我不想成为一个机器学习者。 你知道，我认为人们...只是探索可能的事情是有趣的。 令人兴奋的是建立一些以前没有做过的新事物，这可能会以某种方式使人类受益。

普拉莫德： 很好。 是的，很高兴知道机器学习需要良好的数学背景。 在你最近的一篇关于乐观虚无主义的博客中，你说我们有一天能够模仿大脑并建立一个像人类一样理解和思考的机器。 我们在多大程度上实现了这一目标？

Ryan：是的。 我必须对预言有点小心.....我的意思是，这是我的看法。 我们远不及人类智慧。 我的意思是，我们使用的机器学习系统非常非常简单，根本不起作用。 事实上，我有一篇关于我的居住权的博客文章，其中列举了开发这些模型时遇到的所有困难。 我认为那些不在该领域工作的人有这样的想法，你可以采取这种模式，并通过它推送一些数据，它只是工作。 但事实并非如此。 这些东西非常挑剔且不太清楚，需要经过许多个月的仔细调整和实验才能获得最适度的结果。

所以，我们离它不远了，但是说，我认为这个基础.....最近有一些有希望的技术得到了改进，即卷积网似乎有效，而且传播似乎有效。 事实上，这些东西是基于一个模型，这个神经网络模型，不是大脑般的，但大脑以某种方式激发它，是非常诱人的。 我们还有GPU，我们展示了如何在这些方面进行培训并在一定程度上在GPU之间分配培训。 所以，我认为.....建立更大，更智能的系统的基础正在发生。 而就我个人而言，我是一个无神论者，而且我相信除了化学物质和神经元之外，我脑子里除此之外别无他物。 而且我认为我的意识，我们所有的意识都以某种方式编码在这些神经元之间的相互作用中。 所以，我不明白为什么有一天我们无法在这个领域进行足够的研究和工作，以便模仿这种行为。 当然预测这将会花费多长时间还太早了。

普拉莫德： 很好。 您已经看到了所有Ryan以及您希望在未来20年内在哪里看到技术？

Ryan：我的意思是，我对机器学习及其带来的可能性感到非常兴奋。 我认为，在我们获得真正的人工智能之前，就像之前一样，这种技术有很多应用。 我的意思是，任何你可以.....戴上智能玻璃的系统都可以帮助你，从这项技术中获益匪浅。 所以，你知道，有无数的工业流程可以利用起来这种事情。 我认为回收中心，排序.....分类回收与计算机视觉。 我的意思是，许多系统可以从简单的机器学习系统中受益。 而且我认为我们将越来越多地将这些系统应用于不同的流程。 所以，我认为这将对科技行业产生重大影响，并且对整个人类产生重大影响。

普拉莫德 ： 是机器学习非常令人兴奋。 当我在山景城看到自动驾驶汽车时，我非常兴奋。 有一天，我想坐下来完全控制它。 谢谢你，Ryan，为我们提供了这个漂亮的框架Node.js，并感谢你在节目中。 祝你未来的项目顺利。 很高兴和你谈话。

Ryan： 是的，太棒了。 感谢你的邀请，谈论它是很有趣的话题。

普拉莫德： 谢谢。 就是这样，听众。 我非常喜欢和Ryan说话，这是一个谦逊而且很棒的家伙。 他早年在科技方面取得了如此多的成就。 这样一个鼓舞人心的故事。 再见，我将在两周后与另一个有趣的旅程见面。 Shukriya。

ref：https://mappingthejourney.com/single-post/2017/08/31/episode-8-interview-with-ryan-dahl-creator-of-nodejs/

后记

Node很火，这是好事儿，可是如果没有多层次的人员和理论的接入，这样的热闹就未必是可持续的。

相关的书也太匮乏了,node原理相关的更少，大量的教你弄个聊天室教你弄个博客,反观人家java的什么，`java并发编程`，`深入java虚拟机`，`java编程思想`,这个阵势就不一样了。

我Reco不才，虽然写不出经典巨著，但是愿意为自己的所爱奉献一点社区贡献，这就是本杂志的来由了。

Reco 2018年08月02日