**Thread Pools in NGINX Boost Performance 9x!**

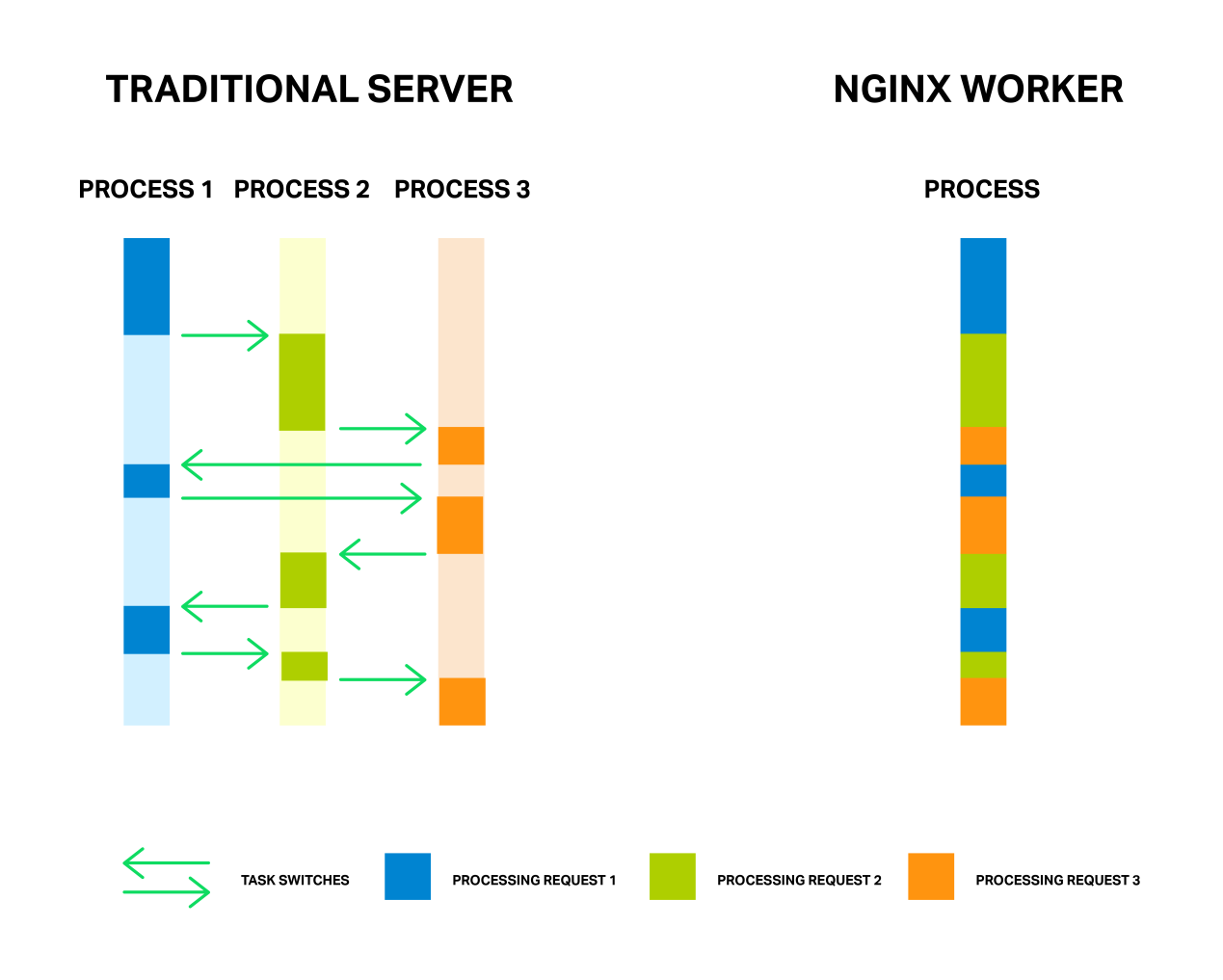
原文地址：<http://nginx.com/blog/thread-pools-boost-performance-9x/>

翻译：陈威凛 2015-06-21 00:12

导言：

众所周知，nginx使用异步和事件驱动的方式来处理连接。相较于使用单独进程或线程对应一个请求的传统服务端框架，nginx使用的是一个worker进程中处理多个连接和请求。为了达到这个目的，nginx处理socket时全都是使用非阻塞的方式，并且使用系统默认的高效的io复用接口epoll或者kqueue。

由于使用的总共的进程数不多（和cpu的核数一对一）且是固定的， 所以将会消耗更少的内存，而且CPU的将会在任务切换上花去更少的时间，众所周知nginx本身就是这一实现的极好的例子，nginx成功地同时处理上百万的请求，而且扩展性也特别好。



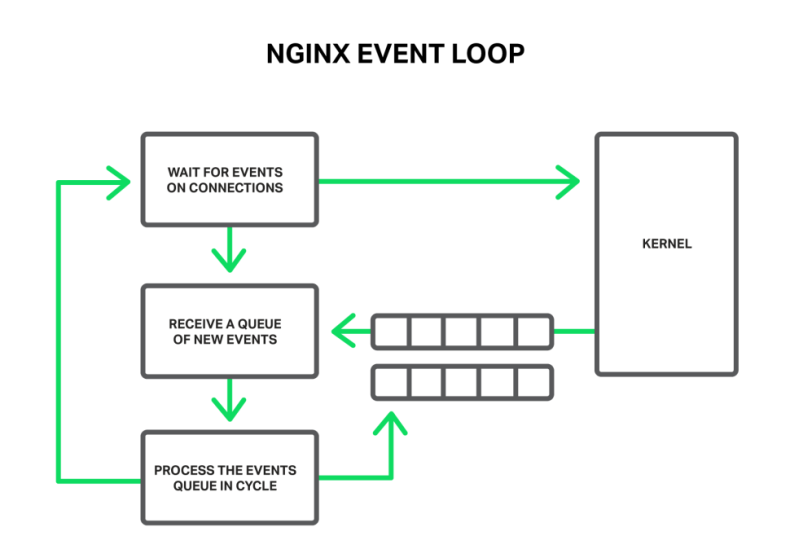
但是异步操作，事件驱动的方式任然有个问题，或者我更愿意称之为一个”enemy”(在此就不翻译成敌人了，太土)。这个”enemy”的名字就是：阻塞。不幸的是许多第三方模块使用了阻塞的调用，而且使用者（甚至有时候连模块的开发者）都没注意到这个缺陷。阻塞的调用将会严重影响nginx的性能，所以必须尽一切可能避免。

即使是在最新的官方nginx实现中，你也不可能完全避免阻塞的调用，为了解决这个问题，一个新的”thread pools”(不想翻译成线程池，保持专业术语不变)的机制被引入到了1.7.11版本的nginx中。这是什么机制，它如何实现，我们会在后面解释。现在我们先直面一下我们的”enemy”。

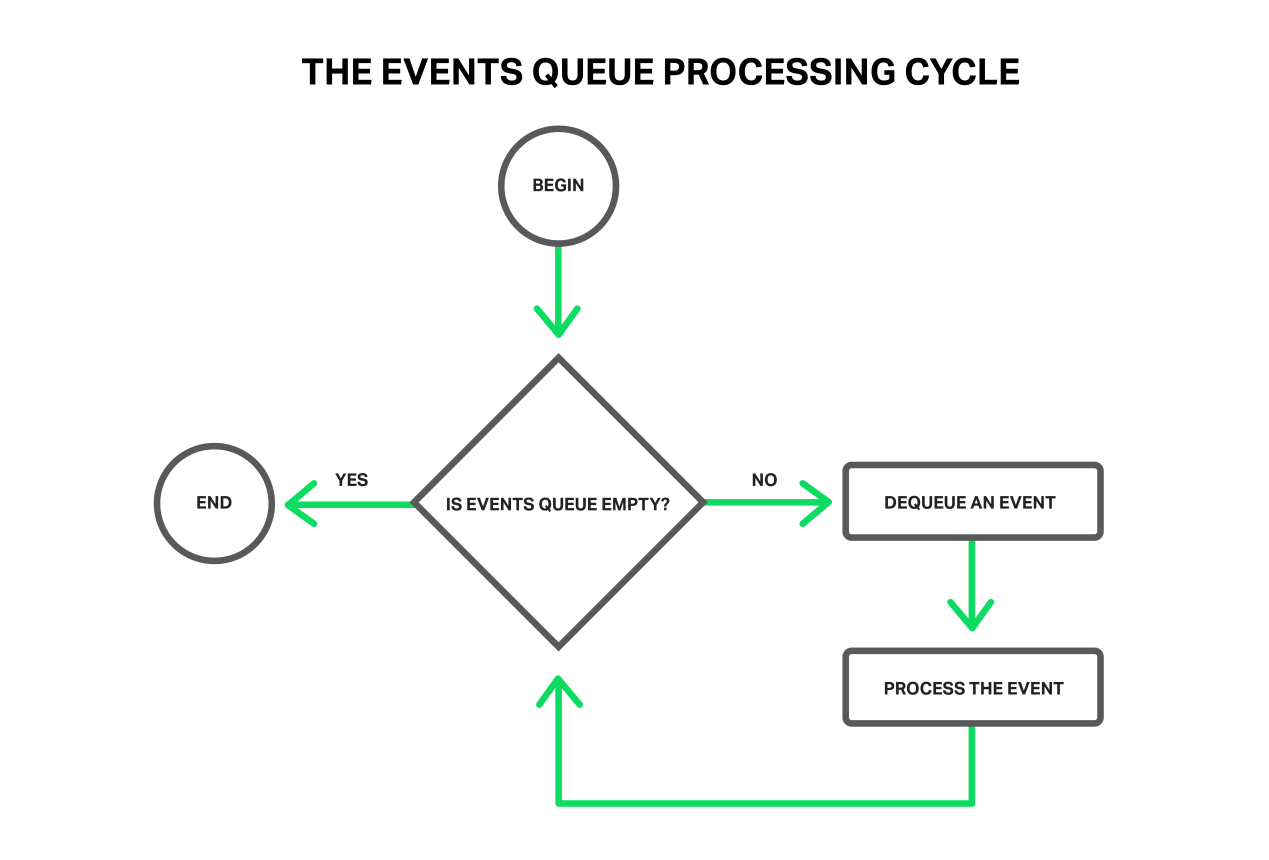
问题：

为了更好地理解这个问题(上面提及的enemy),我们先简述一下nginx是如何工作的。

通常来说nginx由事件处理角色，和一个控制器组成，该控制器主要是接收来自内核的事件，这些事件主要是socket的事件。而且该控制器也会向操作系统发送指令。事实上，操作系统只做一些收发的常规工作，而nginx负担了所有的其它工作。所以对于nginx来说及时响应事件是非常重要的。



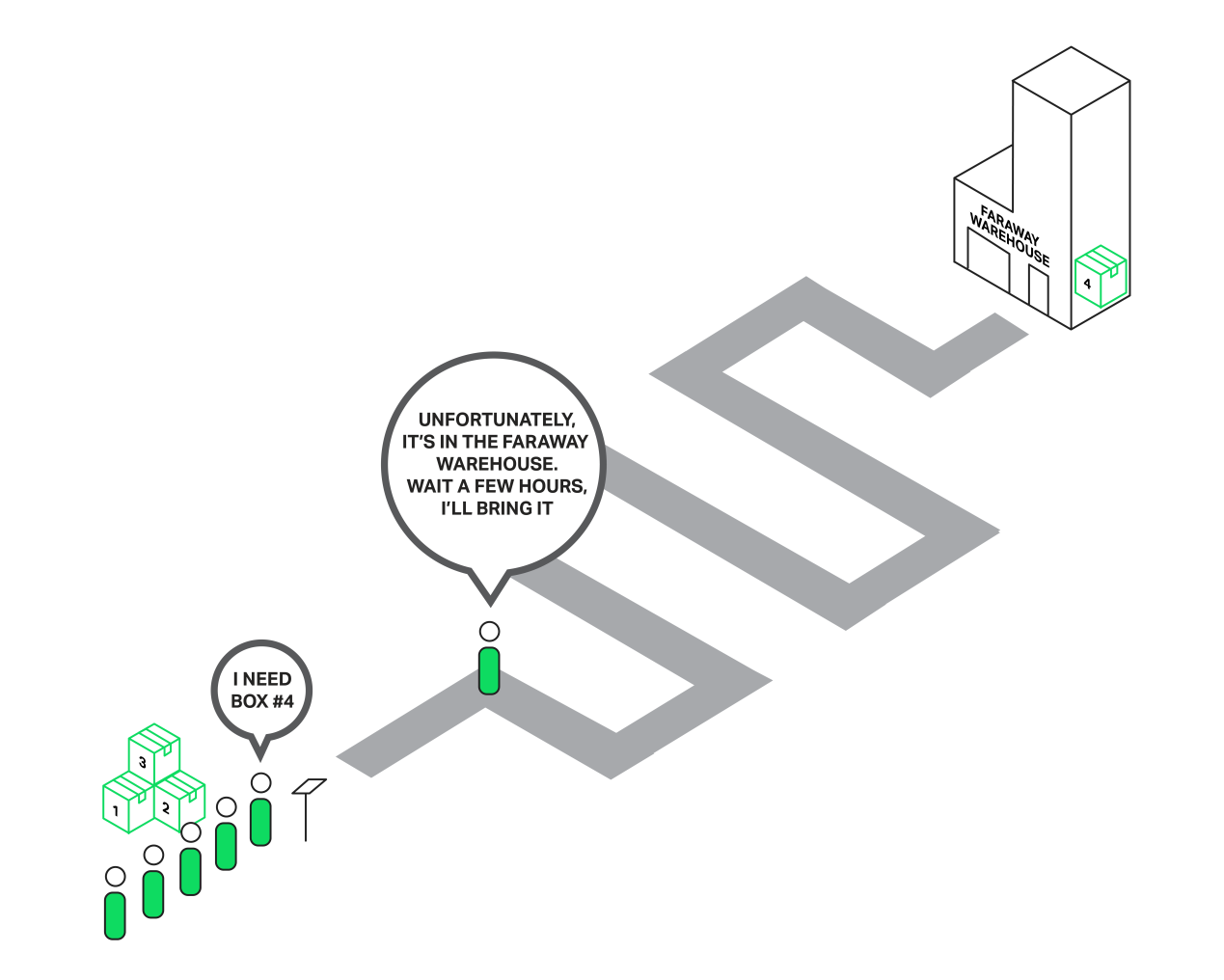
这些事件可以是超时事件，来自socket的读写就绪事件，或者一些错误触发的事件。Nginx接收这些事件然后一个个通过相应的方法处理(一般是对应的回调)，因此所有的这些处理都是在单独线程的简单循环的一个队列中被处理的，例如Nginx将事件从队列中取出，然后从socket上收发数据。在大多数情况下这是很快的（可能只需要cpu花费点拷贝内存的操作），而且所有的nginx的worker进程都是在各自的实例中处理这些事件的。



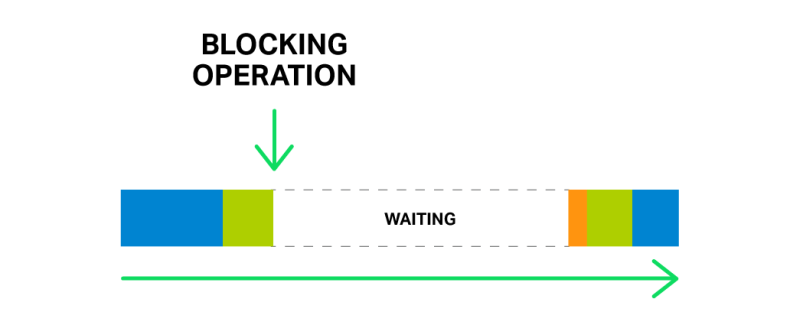
但是当处理一些很花费时间的操作时，将会发生啥事呢？如果这样的话，整个事件处理循环将会被卡住，等待当前操作结束。

所以我们所说的”阻塞操作”是指那些会在很长时间内卡住事件循环的操作。阻塞的原因有很多比如：nginx正在忙于大量的cpu密集型运算，或者需要等待一些资源(比如底层驱动的返回，一些互斥，或者一些来自数据库的同步响应)。关键的问题在于，当这些操作发生时，nginx的worker进程什么事也干不了，不能处理其它事件，即使是一些在队列中没有资源限制的事件。

举个例子，一个售货员前面有一大堆人排队买东西，在队列中的第一个人要买一个店里没用的东西，但是仓库里有。于是售货员就去仓库取东西。那么排队的人需要等很长事件，排队的人会很不爽。排队的每个人的等待时间变长了，但是他们要买的东西就在店里，触手可及，你能想象一下他们的表情吗？



几乎相同的情况会发生在nginx上，当nginx读取一个文件，而且未缓存到内存中，于是需要从硬盘读取，硬盘是很慢的。那么那些及时不需要读写硬盘的事件也会被强制在队列中等待。结果导致潜伏期过长系统资源不能被有效利用。



一些操作系统提供了异步读写文件的接口，而且nginx也可以使用这些接口（例如aio），比较好的例子就是FreeBSD了。不幸的是在linux上我们很难看到与之媲美的接口，虽然linux提供了一些读取文件的异步接口，但是这些接口任然有些显著的缺点，首先访问文件时需要内存的对齐，这一点nginx做得比较好，但是第二个缺点就比较糟糕了，在使用异步接口时需要将文件描述符设置成O\_DIRECT，这就意味着任何岁文件的访问将不适用缓存，而直击和硬盘打交道，增加了硬盘的负担。肯定的是在很多情况下，这不是最优方法。

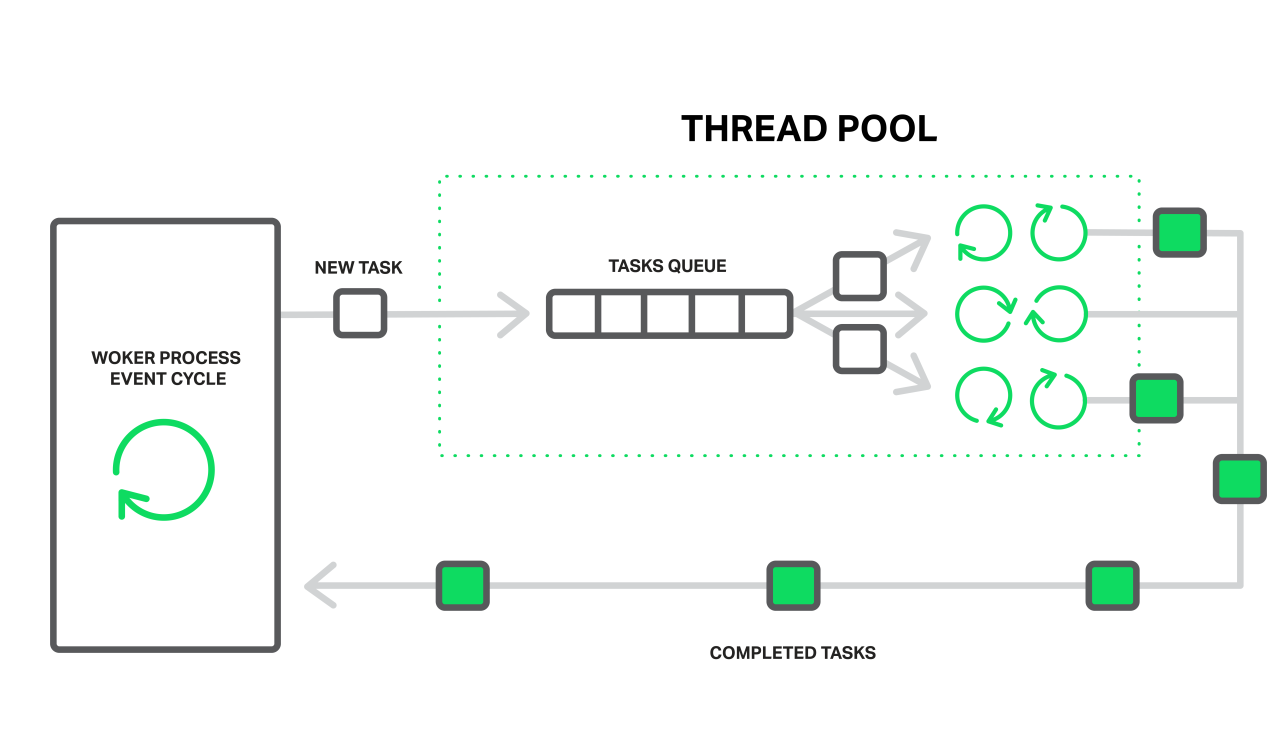
针对这一难点，nginx的1.7.11版本引入了thread pools.当然这一机制并不是默认加入到NginxPlus(针对企业的版本)中的。如果你想编译带有thread pools的NGINX Plus R6版本，你可以联系客服。

下面认识一下什么事thread pools，以及它是如何工作的。

Thread Pools：

让我们先回到那个需要从很远的仓库取回商品的售货员那儿吧。但是他变聪明了(或许他是在被愤怒的客户揍了一顿后变聪明的，谁知道呢！)，他订了一个快递服务。现在如果有人向他买只有很远仓库才有的东西时，他就不需要亲自跑去取货物了，只要告诉快递去处理就可以了，与此同时他可以继续服务其他客户了。因此只有那些所要的东西不在店内的客户需要等待，其他客户可以被及时服务到。

在NGINX中thread pools扮演了快递服务的角色，它由一个任务队列和一些处理队列的线程组成。当一个nginx的worker进程需要执行一个可能的长时间操作时，它就将该任务放入到thread pools的队列中，而一些空闲的线程会主动执行这些任务，而worker就不需要自己处理这些任务了。



看起来好像我们有了另一个队列，对了，但是这个队列受制于特殊的资源。我们不可能从驱动中读取数据的比驱动本身产生数据还快吧！但是现在至少底层驱动不会阻塞住其它事件的处理了，只有那些需要访问文件的事件才会等待。

从硬盘读取数据的操作经常被当作阻塞操作的典型例子。但是nginx对thread pools的引入，却可以用于执行那些不希望在主事件循环中执行的操作。

当前nginx中应用到thread pools的只有大多数系统中的read()调用和linux中的sendfile()。我们会继续测试并且产生对比数据，如果好处易见的话，在将来的热lease版本中我们也许会将一些其它的操作放入thread pools中

Benchmarking:

是时候将理论付诸实践了。为了验证thread pools对性能的影响，我们将会模拟阻塞和非阻塞的混合操作来做测试。

测试需要生产一个在内存中没有被缓存的数据集合。在一个48GB内存的主机上，我们已经生成了256GB的随机数据，这些数据都是来源于一个4MB的文件。然后我们配置了一下nginx去测试。

Nginx的配置文件很简单：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

worker\_processes 16;

events {

accept\_mutex off;

}

http {

include mime.types;

default\_type application/octet-stream;

access\_log off;

sendfile on;

sendfile\_max\_chunk 512k;

server {

listen 8000;

location / {

root /storage;

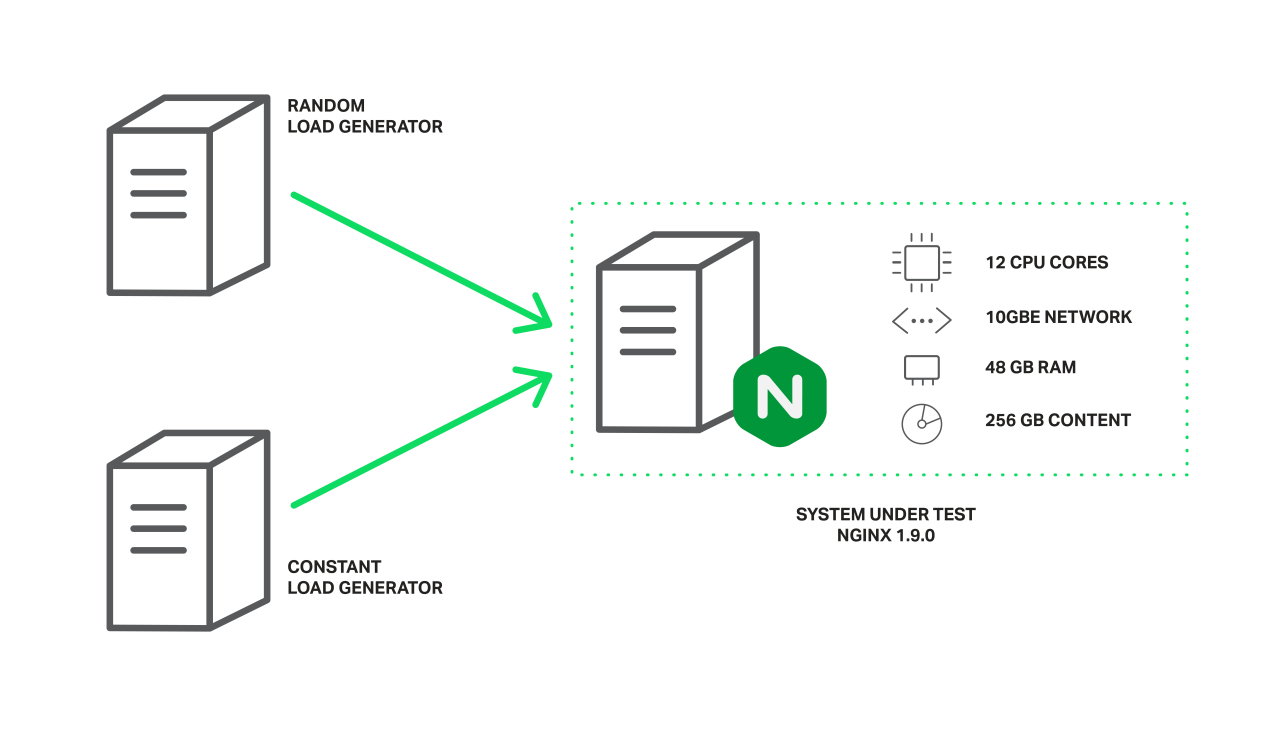
}

}

}

正如你所见，为了获得好的性能，我们适当开关了一些配置项：logging和accept\_mutex被关闭，而sendfile打开了，sendfile\_max\_chunk被设置成512K。sendfile\_max\_chunk可以减少在花费在sendfile()上的阻塞时间，因为，nginx不会试图一次性读取完整的文件，而是每次读取512KB，多次读取。

bla bla bla bla这里介绍了测试机器的配置，就不做翻译了。

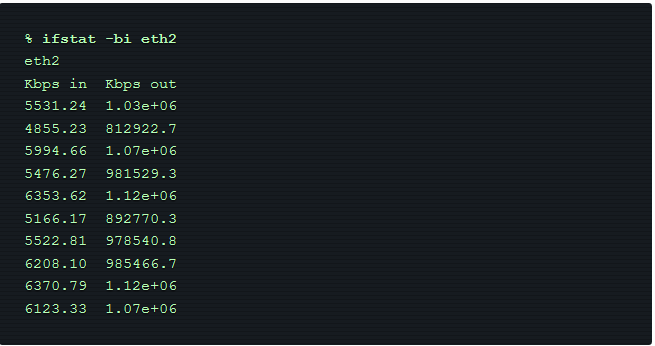


测试环境的客户端使用配置条件完全一样的两台机器。其中一台通过lua脚本产生负载，脚本想服务端通过200个并行的链接向服务端随机请求文件，几乎每个请求所需要的文件都没有明早缓存，而是去阻塞地从硬盘上读取。我们叫这种负载叫做随机负载。

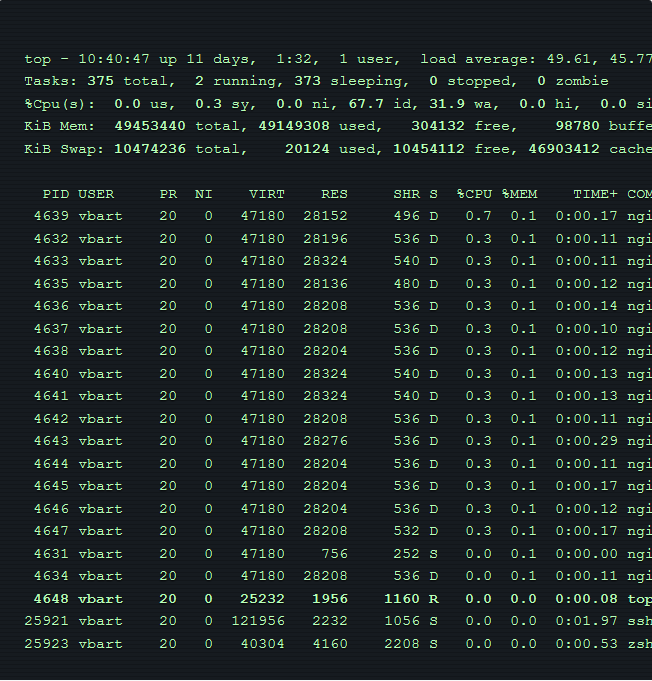
第二台客户机将会在50个并发的连接中多次请求相同的文件，由于文件被频繁地访问，所以在服务端这些文件就会始终在缓存中，通常情况下nginx会很快地响应这种请求，但是当worker进程被其它请求阻塞住时，性能就会大大降低，我们叫这种负载叫常数负载。

我们通过使用ifstat监视服务的吞吐量，和第二个客户端获取的结果来反应整个性能测试的结果。

现在第一次运行中未使用thread pools，结果并没有什么惊喜：

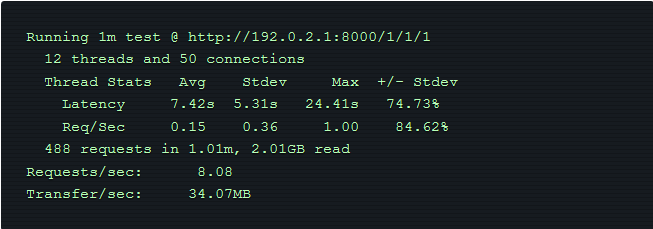


正如你所见，在这种配置下，服务端大概可以产生1 Gbps的传输速度。在top命令的输出中，我们可以看见所有的worker进程都被IO阻塞挂起了(它们的状态是D)



在这个例子中吞吐量受制于磁盘，因为cpu大部分处于空闲状态。

而第二个客户机测试的结果也不尽如人意：



别忘了，第二个客户机请求的文件可都是在内存中有缓存的！第二个客户机的请求的响应时间这么长，是由于所有的worker进程都在忙着从硬盘中读取数据(这是第一个客户机的请求，也就是所谓的随机负载)，导致第二个客户端的请求没法在适当的时间响应。

是时候玩玩threa pools了，我们只需要在location配置块下添加aio threads：

location / {

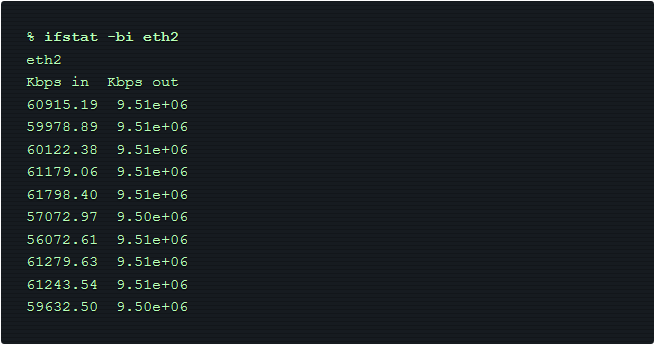
root /storage;

aio threads;

}

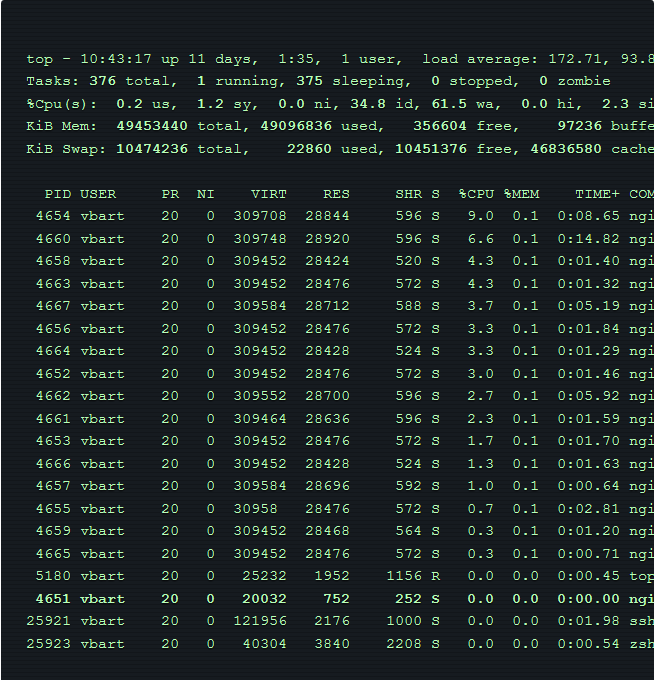
发送命令让nginx重新加载配置文件生效。

然后我们重复上述测试， 结果如下：



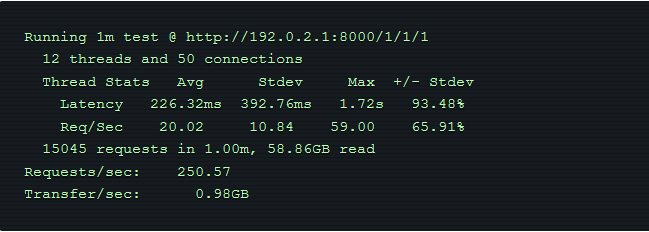
现在我们的服务产生了9.5Gbps传输速度，对比一下没用thread pools时的1Gbps的速度吧！

也许可产生更高的传输速度，但是9.5Gbps已经达到带宽的上限，所以在这种测试下nginx受制于带宽的大小。而worker进程大部分时间只是sleep和wait新的事件触发（它们的状态是S如下top命令的输出）：



还有很多cpu空闲时间。

第二个客户端的结果：



请求4MB文件的时间从7.42秒减少到226.32毫秒（减少了33倍），而每秒处理的请求数增加了31倍（从8增长为250）

产生这样的结果是由于第二个客户端的请求不再被第一个客户机的磁盘读写请求阻塞住，而磁盘的读写是被thread pools处理的。只要磁盘尽可能地处理来自第一个客户端的随机负载，nginx就可以利用cpu空闲的资源和网络的负载来将内存中缓存的文件发送给第二个客户机。

任然不是一劳永逸的方法：

出于对阻塞调用的恐惧，再加上出人意料的测试结果，可能你早已摩拳擦掌准备在你的nginx上配置thread pools了吧。别急！！

幸运的是事实上大部分的文件读写操作都不是在很慢的硬盘上处理的，如果你有足够的RAM空间存储，操作系统会很聪明地将当前使用到的文件缓存到所谓的页高速缓冲存储器

中去。页高速缓冲存储器可以让nginx在大部分测试用例中有很高性能的表现。从页高速缓冲存储器上读取是相当快的，没人会认为这是阻塞操作。相反使用thread pools却有一些额外的性能开销。

所以你的系统有合理大小的RAM，而且你的操作的数据集合不是特别的大，在这种情况下，nginx其实已经是在一个最优方式下运行了，及时没有使用thread pools

对于一些特定的任务，将读操作交给thread pools执行，在技术上是合适的。当当前请求的数据和虚拟缓存中的数据不匹配时，例如给予nginx的流媒体服务。这种情况我们已经在测试中模拟了。

将文件读操作交给thread pools是比较好的，前提是我们需要一个有效的方法去识别当前请求的文件数据是否在缓存中，当不在缓存中就交给thread pools处理

回到售货员的例子中，现在他不知道客户要买的东西到底在店里还是在仓库里，他需要判断是交给快递去处理，还是自己搞定。

没法做出判断的罪魁祸首是操作系统没法告知你数据是否在缓存中。2010年linux总尝试添加的系统调用fincore()就是想解决这个缺陷，但好像这个调用不起作用。后来也有一些其它的尝试，例如在设置了RWF\_NOBLOCK属性后调用preadv2()系统调用。这些不定现在没音儿了。

相反的FreeBSD的用户不会担心这些。FreeBSD已经有一个足够好的一部读文件的接口了，而你不需要使用thread pools。

配置Thread Pools：

如果你已经确定你的程序可以从thread pools中获得益处，那么现在让我们深入研究一下配置文件

这个配置文件是很简单和灵活的。首先你得是nginx1.7.11或更新的。编译前添加--with-threads配置参数。配置文件看起来很平淡，你只要将aio threads添加到http， server或者location块中。

\*\*\*\*\*\*\*\*\*\*\*\*

aio thread；

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这是thread pools的最简化的配置，其实它是下面配置的简化配置：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

thread\_pool default threads=32 max\_queue=65536;

aio threads=default;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这个配置定义了thread pools默认开启了32个工作线程，任务队列的最长长度是最多容下65536个请求。当任务队列满了，nginx会打印log并拒绝服务，日志如下：

thread pool "NAME" queue overflow: N tasks waiting

这个错误意味着线程已经没法很快地处理任务了，任务开始慢慢积压了。你可以试着增加队列长度，不过那其实没用，那意味着你的系统没法处理这么多请求。

正如你注意到的，在thread pools配置中，你可以配置线程的个数，任务队列的最长长度，指定thread pools的名字。配置指定thread pools的名字，意味着你可以配置多个独立的thread pools，在不同的块中配置不同的thread pools：

http {

thread\_pool one threads=128 max\_queue=0;

thread\_pool two threads=32;

server {

location /one {

aio threads=one;

}

location /two {

aio threads=two;

}

}

…

}

max\_queue参数没有制定，那么就会使用默认值65536。正如你在上面配置中看到的max\_queue可以配置成0，在这种情况下有多少threads就处理多少任务。

下面一些配置就不翻译了。困了。。。。。。。。。。