Nginx 1.9.3 发布下载，线程池技术使个别场景性能超9倍

2015-07-26 13:18 feiying 0 阅读 63

Nginx 1.9.3 发布下载，更新内容如下：

*) Change: duplicate "http", "mail", and "stream" blocks are now disallowed.

*) Feature: connection limiting in the stream module.

*) Feature: data rate limiting in the stream module.

*) Bugfix: the "zone" directive inside the "upstream" block did not work on Windows.

*) Bugfix: compatibility with LibreSSL in the stream module. Thanks to Piotr Sikora.

*) Bugfix: in the "--builddir" configure parameter. Thanks to Piotr Sikora.

*) Bugfix: the "ssl_stapling_file" directive did not work; the bug had appeared in 1.9.2. Thanks to Faidon Liam botis and Brandon Black.

*) Bugfix: a segmentation fault might occur in a worker process if the "ssl_stapling" directive was used; the bu g had appeared in 1.9.2.Thanks to Matthew Baldwin.

详细信息请查看发行页面：

http://nginx.org/

Nginx（发音同 engine x）是一款轻量级的Web 服务器 / 反向代理服务器及电子邮件（IMAP/POP3）代理服务器，并在一个BSD-like 协议下发行。由俄罗斯的程序设计师Igor Sysoev所开发，最初供俄国大型的入口网站及搜寻引擎Rambler（俄文：Рамблер）使用。 其特点是占有内存少，并发能力强，事实上nginx的并发能力确实在同类型的网页伺服器中表现较好.目前中国大陆使用nginx网站用户有：新浪、网易、 腾讯,另外知名的微网志Plurk也使用nginx。

CentOS 6.2实战部署Nginx+MySQL+PHP http://www.linuxidc.com/Linux/2013-09/90020.htm

使用Nginx搭建WEB服务器 http://www.linuxidc.com/Linux/2013-09/89768.htm

搭建基于Linux6.3+Nginx1.2+PHP5+MySQL5.5的Web服务器全过程 http://www.linuxidc.com/Linux/2013-09/89692.htm

CentOS 6.3下Nginx性能调优 http://www.linuxidc.com/Linux/2013-09/89656.htm

CentOS 6.3下配置Nginx加载ngx_pagespeed模块 http://www.linuxidc.com/Linux/2013-09/89657.htm

CentOS 6.4安装配置Nginx+Pcre+php-fpm http://www.linuxidc.com/Linux/2013-08/88984.htm

Nginx安装配置使用详细笔记 http://www.linuxidc.com/Linux/2014-07/104499.htm

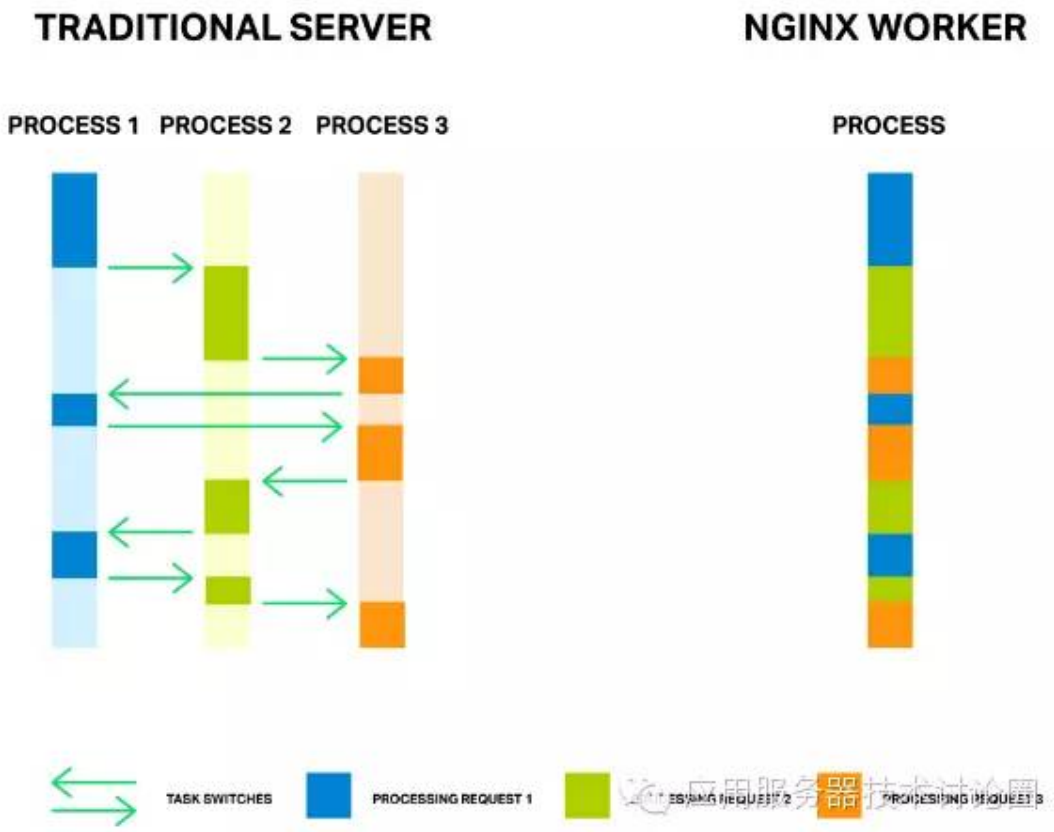
Nginx日志过滤 使用ngx_log_if不记录特定日志 http://www.linuxidc.com/Linux/2014-07/104686.htm

==========》

1. 引言

正如我们所知，NGINX采用了异步、事件驱动的方法来处理连接。这种处理方式无需（像使用传统架构的服务器一样）为每个请求创建额外的专用进程或者线程，而是在一个工作进程中处理多个连接和请求。为此，NGINX工作在非阻塞的socket模式下，并使用了epoll 和 kqueue这样有效的方法。

因为满负载进程的数量很少（通常每核CPU只有一个）而且恒定，所以任务切换只消耗很少的内存，而且不会浪费CPU周期。通过NGINX本身的实例，这种方法的优点已经为众人所知。NGINX可以非常好地处理百万级规模的并发请求。



> 每个进程都消耗额外的内存，而且每次进程间的切换都会消耗CPU周期
> 并丢弃CPU高速缓存中的数据。

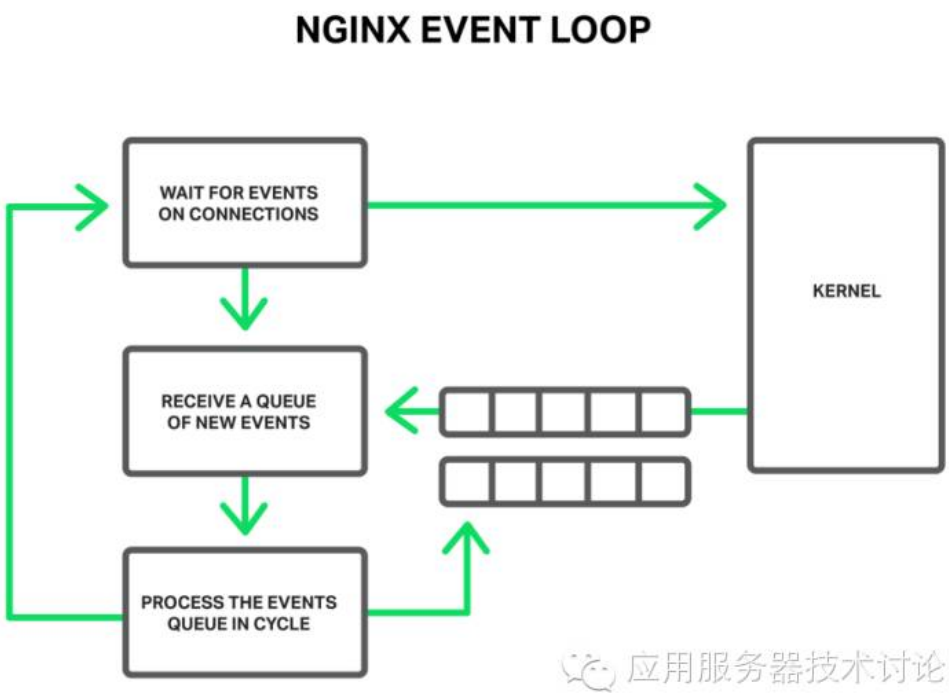
但是，异步、事件驱动方法仍然存在问题。或者，我喜欢将这一问题称为“敌兵”，这个敌兵的名字叫**阻塞（blocking）**。不幸的是，很多第三方模块使用了阻塞调用，然而用户（有时甚至是模块的开发者）并不知道阻塞的缺点。阻塞操作可以毁掉NGINX的性能，我们必须不惜一切代价避免使用阻塞。

即使在当前官方的NGINX代码中，依然无法在全部场景中避免使用阻塞，NGINX1.7.11中实现的线程池机制解决了这个问题。我们将在后面讲述这个线程池是什么以及该如何使用。现在，让我们先和我们的“敌兵”进行一次面对面的碰撞。

2. 问题

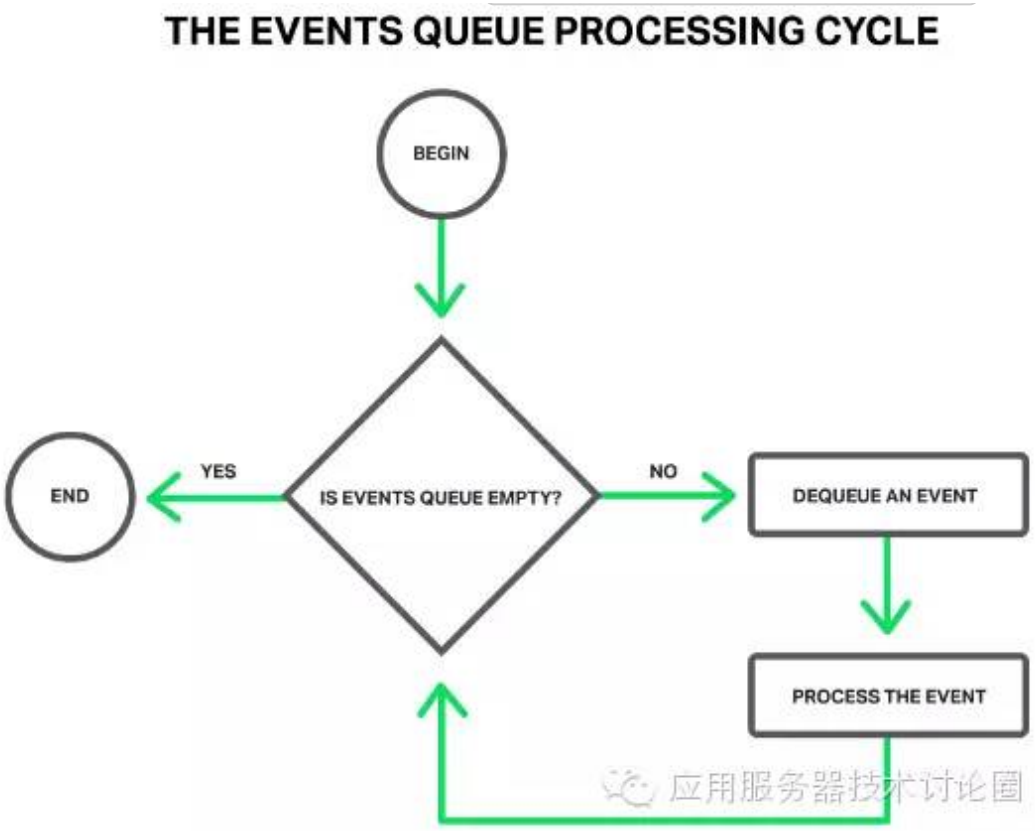
首先，为了更好地理解这一问题，我们用几句话说明下NGINX是如何工作的。

通常情况下，NGINX是一个事件处理器，即一个接收来自内核的所有连接事件的信息，然后向操作系统发出做什么指令的控制器。实际上，NGINX干了编排操作系统的全部脏活累活，而操作系统做的是读取和发送字节这样的日常工作。所以，对于NGINX来说，快速和及时的响应是非常重要的。



> 工作进程监听并处理来自内核的事件

事件可以是超时、socket读写就绪的通知，或者发生错误的通知。NGINX接收大量的事件，然后一个接一个地处理它们，并执行必要的操作。因此，所有的处理过程是通过一个线程中的队列，在一个简单循环中完成的。NGINX从队列中取出一个事件并对其做出响应，比如读写socket。在多数情况下，这种方式是非常快的（也许只需要几个CPU周期，将一些数据复制到内存中），NGINX可以在一瞬间处理掉队列中的所有事件。

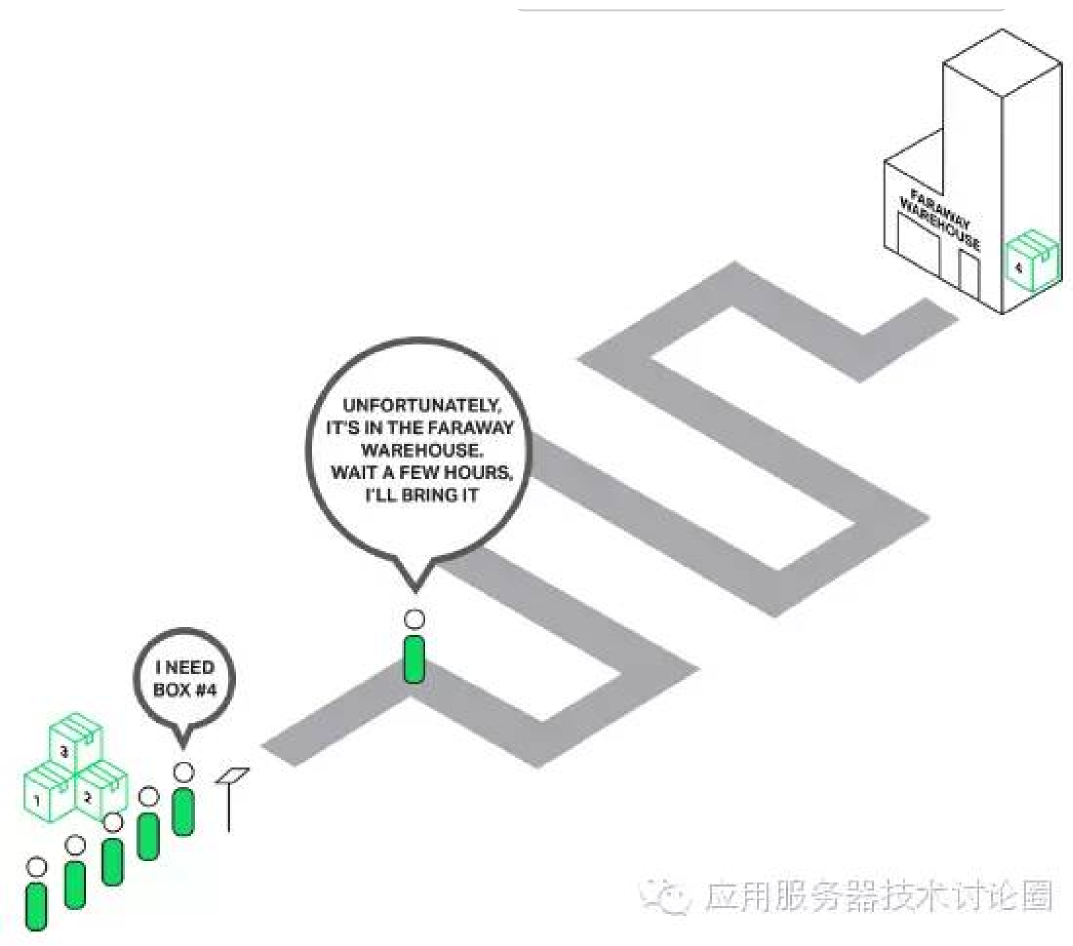


> 所有处理过程是在一个简单的循环中，由一个线程完成

但是，如果NGINX要处理的操作是一些又长又重的操作，又会发生什么呢？整个事件处理循环将会卡住，等待这个操作执行完毕。

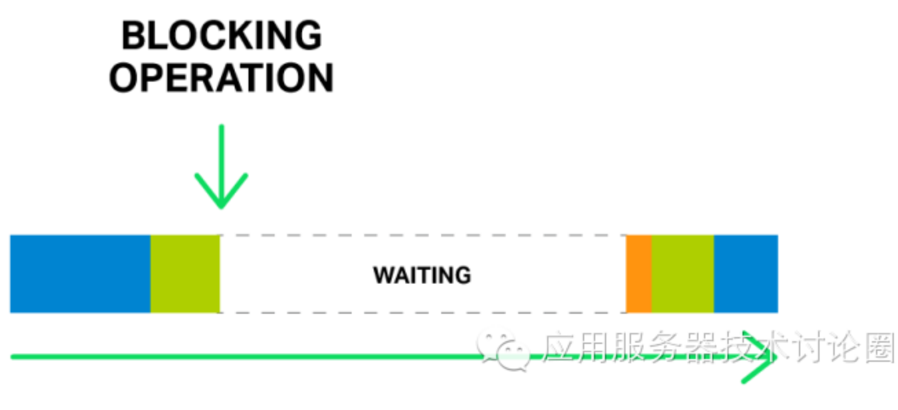
因此，所谓“阻塞操作”是指任何导致事件处理循环显著停止一段时间的操作。操作可以由于各种原因成为阻塞操作。例如，NGINX可能因长时间、CPU密集型处理，或者可能等待访问某个资源（比如硬盘，或者一个互斥体，亦或要从处于同步方式的数据库获得相应的库函数调用等）而繁忙。关键是在处理这样的操作期间，工作进程无法做其他事情或者处理其他事件，即使有更多的可用系统资源可以被队列中的一些事件所利用。

我们来打个比方，一个商店的营业员要接待他面前排起的一长队顾客。队伍中的第一位顾客想要的某件商品不在店里而在仓库中。这位营业员跑去仓库把东西拿来。现在整个队伍必须为这样的配货方式等待数个小时，队伍中的每个人都很不爽。你可以想见人们的反应吧？队伍中每个人的等待时间都要增加这些时间，除非他们要买的东西就在店里。



> 队伍中的每个人不得不等待第一个人的购买

在NGINX中会发生几乎同样的情况，比如当读取一个文件的时候，如果该文件没有缓存在内存中，就要从磁盘上读取。从磁盘（特别是旋转式的磁盘）读取是很慢的,而当队列中等待的其他请求可能不需要访问磁盘时，它们也得被迫等待。导致的结果是，延迟增加并且系统资源没有得到充分利用。



> 一个阻塞操作足以显著地延缓所有接下来的操作

一些操作系统为读写文件提供了异步接口，NGINX可以使用这样的接口（见AIO指令）。FreeBSD就是个很好的例子。不幸的是，我们不能在Linux上得到相同的福利。虽然Linux为读取文件提供了一种异步接口……缺点。……冲要对齐，但NGINX很

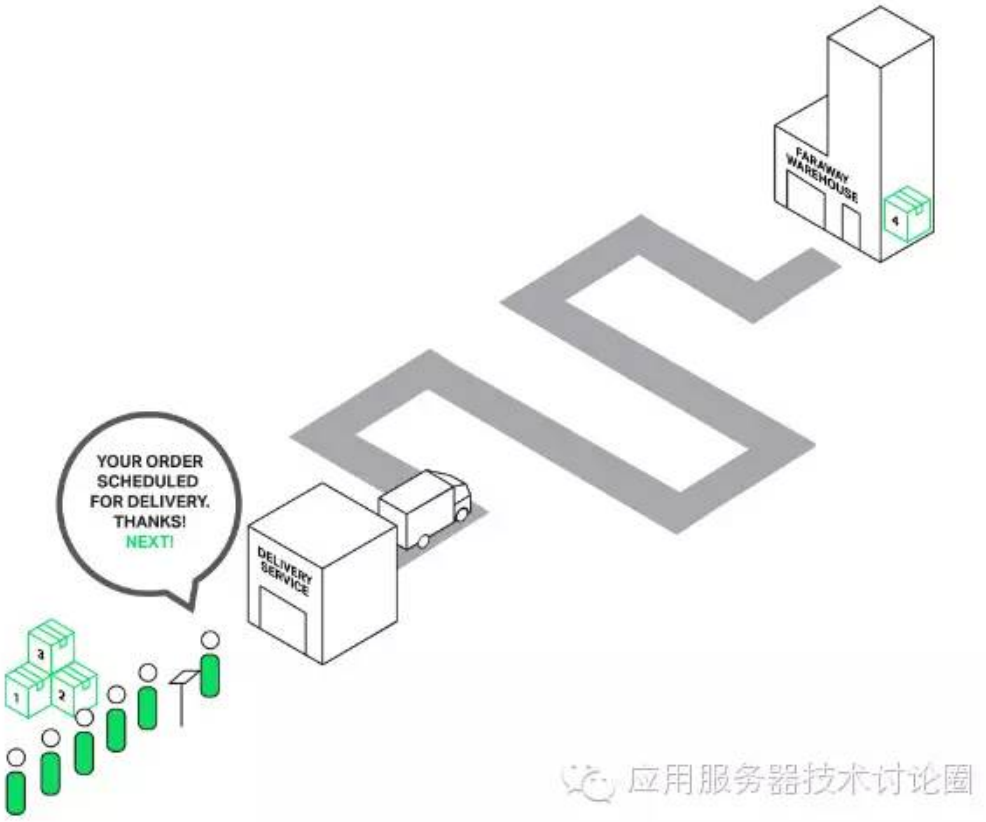
地处理了这个问题。但是，另一个缺点更糟糕。异步接口要求文件描述符中要设置O_DIRECT标记，就是说任何对文件的访问都将绕过内存中的缓存，这增加了磁盘的负载。在很多场景中，这都绝对不是最佳选择。

为了有针对性地解决这一问题，在NGINX 1.7.11中引入了线程池。默认情况下，NGINX+还没有包含线程池，但是如果你想试试的话，可以联系销售人员，NGINX+ R6是一个已经启用了线程池的构建版本。

现在，让我们走进线程池，看看它是什么以及如何工作的。

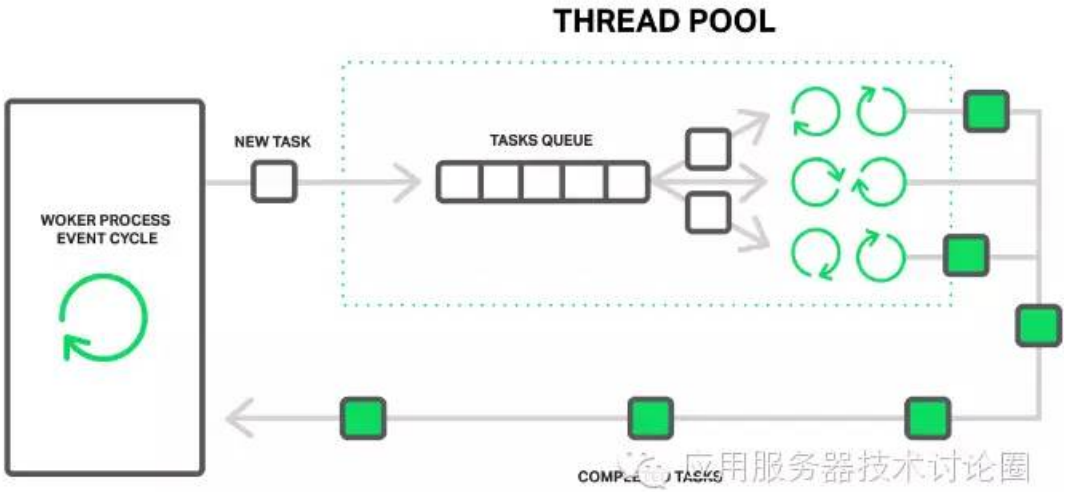
3. 线程池

让我们回到那个可怜的，要从大老远的仓库去配货的售货员那儿。这回，他已经变聪明了（或者也许是在一群愤怒的顾客教训了一番之后，他才变得聪明的？），雇用了一个配货服务团队。现在，当任何人要买的东西在大老远的仓库时，他不再亲自去仓库了，只需要将订单丢给配货服务，他们将处理订单，同时，我们的售货员依然可以继续为其他顾客服务。因此，只有那些要买仓库里东西的顾客需要等待配货，其他顾客可以得到即时服务。



> 传递订单给配货服务不会阻塞队伍

对NGINX而言，线程池执行的就是配货服务的功能。它由一个任务队列和一组处理这个队列的线程组成。
当工作进程需要执行一个潜在的长操作时，工作进程不再自己执行这个操作，而是将任务放到线程池队列中，任何空闲的线程都可以从队列中获取并执行这个任务。



工作进程将阻塞操作卸给线程池

那么，这就像我们有了另外一个队列。是这样的，但是在这个场景中，队列受限于特殊的资源。磁盘的读取速度不能比磁盘产生数据的速度快。不管怎么说，至少现在磁盘不再延误其他事件，只有访问文件的请求需要等待。

“从磁盘读取”这个操作通常是阻塞操作最常见的示例，但是实际上，NGINX中实现的线程池可用于处理任何不适合在主循环中执行的任务。

目前，卸载到线程池中执行的两个基本操作是大多数操作系统中的read()系统调用和Linux中的sendfile()。接下来，我们将对线程池进行测试（test）和基准测试（benchmark），在未来的版本中，如果有明显的优势，我们可能会卸载其他操作到线程池中。

4. 基准测试

现在让我们从理论过度到实践。我们将进行一次模拟基准测试（synthetic benchmark），模拟在阻塞操作和非阻塞操作的最差混合条件下，使用线程池的效果。

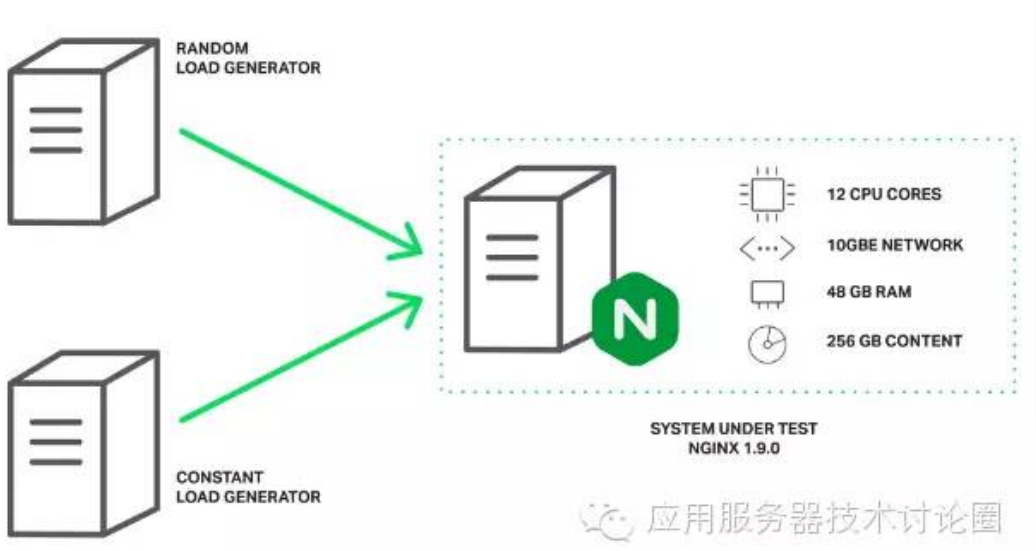
另外，我们需要一个内存肯定放不下的数据集。在一台48GB内存的机器上，我们已经产生了每文件大小为4MB的随机数据，总共256GB，然后配置NGINX，版本为1.9.0。

配置很简单：

```
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;
        }
    }
}
```

如上所示，为了达到更好的性能，我们调整了几个参数：禁用了logging和accept_mutex，同时，启用了sendfile并设置了sendfile_max_chunk的大小。最后一个指令可以减少阻塞调用sendfile()所花费的最长时间，因为NGINX不会尝试一次将整个文件发送出去，而是每次发送大小为512 KB的块数据。

这台测试服务器有2个Intel Xeon E5645处理器（共计：12核、24超线程）和10-Gbps的网络接口。磁盘子系统是由4块西部数据WD1003FBYX 磁盘组成的RAID10阵列。所有这些硬件由Ubuntu服务器14.04.1 LTS供电。



为基准测试配置负载生成器和NGINX

客户端有2台服务器，它们的规格相同。在其中一台上，在**wrk**中使用Lua脚本创建了负载程序。脚本使用200个并行连接向服务器请求文件，每个请求都可能未命中缓存而从磁盘阻塞读取。我们将这种负载称作随机负载。

在另一台客户端机器上，我们将运行**wrk**的另一个副本，使用50个并行连接多次请求同一个文件。因为这个文件将被频繁地访问，所以它会一直驻留在内存中。在正常情况下，NGINX能够非常快速地服务这些请求，但是如果工作进程被其他请求阻塞的话，性能将会下降。我们将这种负载称作恒定负载。

性能将由服务器上**ifstat**监测的吞吐率（throughput）和从第二台客户端获取的**wrk**结果来度量。

现在，没有使用线程池的第一次运行将不会带给我们非常振奋的结果：

```
% ifstat -bi eth2
eth2
Kbps in  Kbps out5531.24  1.03e+064855.23  812922.75994.66  1.07e+065476.27  981529.36353.62  1.
12e+065166.17  892770.35522.81  978540.86208.10  985466.76370.79  1.12e+066123.33  1.07e+06
```

如上所示，使用这种配置，服务器产生的总流量约为1Gbps。从下面所示的**top**输出，我们可以看到，工作进程的大部分时间花在阻塞I/O上（它们处于top的D状态）：

```
top - 10:40:47 up 11 days,  1:32,  1 user,  load average: 49.61, 45.77 62.89Tasks: 375 total,  2
running, 373 sleeping,  0 stopped,  0 zombie%Cpu(s):  0.0 us,  0.3 sy,  0.0 ni, 67.7 id, 31.9 wa
,  0.0 hi,  0.0 si,  0.0 stKiB Mem:  49453440 total, 49149308 used,   304132 free,    98780 buff
ersKiB Swap: 10474236 total,    20124 used, 10454112 free, 46903412 cached Mem

  PID USER      PR  NI    VIRT    RES     SHR S  %CPU %MEM    TIME+ COMMAND
 4639 vbart    20   0   47180  28152     496 D   0.7  0.1  0:00.17 nginx
 4632 vbart    20   0   47180  28196     536 D   0.3  0.1  0:00.11 nginx
 4633 vbart    20   0   47180  28324     540 D   0.3  0.1  0:00.11 nginx
 4635 vbart    20   0   47180  28136     480 D   0.3  0.1  0:00.12 nginx
 4636 vbart    20   0   47180  28208     536 D   0.3  0.1  0:00.14 nginx
 4637 vbart    20   0   47180  28208     536 D   0.3  0.1  0:00.10 nginx
 4638 vbart    20   0   47180  28204     536 D   0.3  0.1  0:00.12 nginx
 4640 vbart    20   0   47180  28324     540 D   0.3  0.1  0:00.13 nginx
 4641 vbart    20   0   47180  28324     540 D   0.3  0.1  0:00.13 nginx
 4642 vbart    20   0   47180  28208     536 D   0.3  0.1  0:00.11 nginx
 4643 vbart    20   0   47180  28276     536 D   0.3  0.1  0:00.29 nginx
 4644 vbart    20   0   47180  28204     536 D   0.3  0.1  0:00.11 nginx
 4645 vbart    20   0   47180  28204     536 D   0.3  0.1  0:00.17 nginx
 4646 vbart    20   0   47180  28204     536 D   0.3  0.1  0:00.12 nginx
 4647 vbart    20   0   47180  28208     532 D   0.3  0.1  0:00.17 nginx
 4631 vbart    20   0   47180    756     252 S   0.0  0.1  0:00.00 nginx
 4634 vbart    20   0   47180  28208     536 D   0.0  0.1  0:00.11 nginx
 4648 vbart    20   0   25232   1956    1160 R   0.0  0.0  0:00.08 top25921 vbart    20   0  121
956   2232    1056 S   0.0  0.0  0:01.97 sshd25923 vbart    20   0   40304   4160    2208 S   0.
0  0.0  0:00.53 zsh
```

在这种情况下，吞吐率受限于磁盘子系统，而CPU在大部分时间里是空闲的。从**wrk**获得的结果也非常低：

```
Running 1m test @ http://192.0.2.1:8000/1/1/1
  12 threads and 50 connections
  Thread Stats   Avg    Stdev     Max  +/- Stdev
    Latency     7.42s  5.31s   24.41s   74.73%
    Req/Sec     0.15    0.36     1.00    84.62%  488 requests in 1.01m, 2.01GB readRequests/sec:
8.08Transfer/sec:     34.07MB
```

请记住，文件是从内存送达的！第一个客户端的200个连接创建的随机负载，使服务器端的全部的工作进程忙于从磁盘读取文件，因此产生了过大的延迟，并且无法在合理的时间内处理我们的请求。

现在，我们的线程池要登场了。为此，我们只需在location块中添加aio threads指令：

```
location / { root /storage; aio threads; }
```

接着，执行NGINX reload重新加载配置。

然后，我们重复上述的测试：

```
% ifstat -bi eth2
eth2
Kbps in  Kbps out60915.19  9.51e+0659978.89  9.51e+0660122.38  9.51e+0661179.06  9.51e+0661798.4
0  9.51e+0657072.97  9.50e+0656072.61  9.51e+0661279.63  9.51e+0661243.54  9.51e+0659632.50  9.5
0e+06
```

现在，我们的服务器产生的流量是**9.5Gbps**，相比之下，没有使用线程池时只有约1Gbps！

理论上还可以产生更多的流量，但是这已经达到了机器的最大网络吞吐能力，所以在这次NGINX的测试中，NGINX受限于网络接口。工作进程的大部分时间只是休眠和等待新的事件（它们处于top的S状态）：

```
top - 10:43:17 up 11 days,  1:35,  1 user,  load average: 172.71, 93.84, 77.90Tasks: 376 total,
1 running, 375 sleeping,  0 stopped,  0 zombie%Cpu(s):  0.2 us,  1.2 sy,  0.0 ni, 34.8 id, 61.5
wa,  0.0 hi,  2.3 si,  0.0 stKiB Mem:  49453440 total, 49096836 used,   356604 free,    97236 bu
ffersKiB Swap: 10474236 total,    22860 used, 10451376 free, 46836580 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 4654 vbart    20   0  309708  28844    596 S   9.0  0.1   0:08.65 nginx
 4660 vbart    20   0  309748  28920    596 S   6.6  0.1   0:14.82 nginx
 4658 vbart    20   0  309452  28424    520 S   4.3  0.1   0:01.40 nginx
 4663 vbart    20   0  309452  28476    572 S   4.3  0.1   0:01.32 nginx
 4667 vbart    20   0  309584  28712    588 S   3.7  0.1   0:05.19 nginx
 4656 vbart    20   0  309452  28476    572 S   3.3  0.1   0:01.84 nginx
 4664 vbart    20   0  309452  28428    524 S   3.3  0.1   0:01.29 nginx
 4652 vbart    20   0  309452  28476    572 S   3.0  0.1   0:01.46 nginx
 4662 vbart    20   0  309552  28700    596 S   2.7  0.1   0:05.92 nginx
 4661 vbart    20   0  309464  28636    596 S   2.3  0.1   0:01.59 nginx
 4653 vbart    20   0  309452  28476    572 S   1.7  0.1   0:01.70 nginx
 4666 vbart    20   0  309452  28428    524 S   1.3  0.1   0:01.63 nginx
 4657 vbart    20   0  309584  28696    592 S   1.0  0.1   0:00.64 nginx
 4655 vbart    20   0  30958   28476    572 S   0.7  0.1   0:02.81 nginx
 4659 vbart    20   0  309452  28468    564 S   0.3  0.1   0:01.20 nginx
 4665 vbart    20   0  309452  28476    572 S   0.3  0.1   0:00.71 nginx
 5180 vbart    20   0   25232   1952   1156 R   0.0  0.0   0:00.45 top
 4651 vbart    20   0   20032    752    252 S   0.0  0.0   0:00.00 nginx25921 vbart    20   0  1
21956   2176    1000 S   0.0  0.0  0:01.98 sshd25923 vbart    20   0   40304   3840    2208 S
 0.0  0.0  0:00.54 zsh
```

如上所示，基准测试中还有大量的CPU资源剩余。

wrk的结果如下：

```
Running 1m test @ http://192.0.2.1:8000/1/1/1
  12 threads and 50 connections
  Thread Stats   Avg      Stdev     Max  +/- Stdev
    Latency   226.32ms  392.76ms   1.72s   93.48%
    Req/Sec    20.02     10.84    59.00    65.91%  15045 requests in 1.00m, 58.86GB readRequest
s/sec:    250.57Transfer/sec:      0.98GB
```

服务器处理4MB文件的平均时间从7.42秒降到226.32毫秒（减少了33倍），每秒请求处理数提升了31倍（250 vs 8）！

对此，我们的解释是请求不再因为工作进程被阻塞在读文件，而滞留在事件队列中，等待处理，它们可以被空闲的进程处理掉。只要磁盘子系统能做到最好，就能服务好第一个客户端上的随机负载，NGINX可以使用剩余的CPU资源和网络容量，从内存中读取，以服务于上述的第二个客户端的请求。

5. 依然没有银弹

在抛出我们对阻塞操作的担忧并给出一些令人振奋的结果后，可能大部分人已经打算在你的服务器上配置线程池了。先别着急。

实际上，最幸运的情况是，读取和发送文件操作不去处理缓慢的硬盘驱动器。如果我们有足够多的内存来存储数据集，那么操作系统将会足够聪明地在被称作“页面缓存”的地方，缓存频繁使用的文件。

“页面缓存”的效果很好，可以让NGINX在几乎所有常见的用例中展示优异的性能。从页面缓存中读取比较快，没有人会说这种操作是“阻塞”。而另一方面，卸载任务到一个线程池是有一定开销的。

因此，如果内存有合理的大小并且待处理的数据集不是很大的话，那么无需使用线程池，NGINX已经工作在最优化的方式下。

卸载读操作到线程池是一种适用于非常特殊任务的技术。只有当经常请求的内容的大小，不适合操作系统的虚拟机缓存时，这种技术才是最有用的。至于可能适用的场景，比如，基于NGINX的高负载流媒体服务器。这正是我们已经模拟的基准测试的场景。

我们如果可以改进卸载读操作到线程池，将会非常有意义。我们只需要知道所需的文件数据是否在内存中，只有不在内存中时，读操作才应该卸载到一个单独的线程中。

再回到售货员那个比喻的场景中，这回，售货员不知道要买的商品是否在店里，他必须要么总是将所有的订单提交给配货服务，要么总是亲自处理它们。

人艰不拆，操作系统缺少这样的功能。第一次尝试是在2010年，人们试图将这一功能添加到Linux作为fincore()系统调用，但是没有成功。后来还有一些尝试，是使用RWF_NONBLOCK标记作为preadv2()系统调用来实现这一功能（详情见LWN.net上的非阻塞缓冲文件读取操作和异步缓冲读操作）。但所有这些补丁的命运目前还不明朗。悲催的是，这些补丁尚没有被内核接受的主要原因，貌似是因为旷日持久的撕逼大战（bikeshedding）。

另一方面，FreeBSD的用户完全不必担心。FreeBSD已经具备足够好的异步读取文件接口，我们应该用这个接口而不是线程池。

6. 配置线程池

所以，如果你确信在你的场景中使用线程池可以带来好处，那么现在是时候深入了解线程池的配置了。

线程池的配置非常简单、灵活。首先，获取NGINX 1.7.11或更高版本的源代码，使用`--with-threads`配置参数编译。在最简单的场景中，配置看起来很朴实。我们只需要在`http`、 `server`，或者`location`上下文中包含aio threads指令即可：

```
aio threads;
```

这是线程池的最简配置。实际上的精简版本示例如下：

```
thread_pool default threads=32 max_queue=65536;aio threads=default;
```

这里定义了一个名为“default”，包含32个线程，任务队列最多支持65536个请求的线程池。如果任务队列过载，NGINX将输出如下错误日志并拒绝请求：

```
thread pool "NAME" queue overflow: N tasks waiting
```

错误输出意味着线程处理作业的速度有可能低于任务入队的速度了。你可以尝试增加队列的最大值，但是如果这无济于事，那么这说明你的系统没有能力处理如此多的请求了。

正如你已经注意到的，你可以使用thread_pool指令，配置线程的数量、队列的最大值，以及线程池的名称。最后要说明的是，可以配置多个独立的线程池，将它们置于不同的配置文件中，用做不同的目的：

```
http { thread_pool one threads=128 max_queue=0; thread_pool two threads=32; server { location /o
ne { aio threads=one; }

        location /two { aio threads=two; }
    }
…
}
```

如果没有指定`max_queue`参数的值，默认使用的值是65536。如上所示，可以设置`max_queue`为0。在这种情况下，线程池将使用配置中全部数量的线程，尽可能地同时处理多个任务；队列中不会有等待的任务。

现在，假设我们有一台服务器，挂了3块硬盘，我们希望把该服务器用作“缓存代理”，缓存后端服务器的全部响应信息。预期的缓存数据量远大于可用的内存。它实际上是我们个人CDN的一个缓存节点。毫无疑问，在这种情况下，最重要的事情是发挥硬盘的最大性能。

我们的选择之一是配置一个RAID阵列。这种方法毁誉参半，现在，有了NGINX，我们可以有其他的选择：

```
# 我们假设每块硬盘挂载在相应的目录中：/mnt/disk1、/mnt/disk2、/mnt/disk3proxy_cache_path /mnt/disk1
levels=1:2 keys_zone=cache_1:256m max_size=1024G
                 use_temp_path=off;
proxy_cache_path /mnt/disk2 levels=1:2 keys_zone=cache_2:256m max_size=1024G
                 use_temp_path=off;
proxy_cache_path /mnt/disk3 levels=1:2 keys_zone=cache_3:256m max_size=1024G
                 use_temp_path=off;

thread_pool pool_1 threads=16;
thread_pool pool_2 threads=16;
thread_pool pool_3 threads=16;

split_clients $request_uri $disk {    33.3%     1;    33.3%     2;
    *         3;
}

location / {
    proxy_pass http://backend;
    proxy_cache_key $request_uri;
    proxy_cache cache_$disk;
    aio threads=pool_$disk;
    sendfile on;
}
```

在这份配置中，使用了3个独立的缓存，每个缓存专用一块硬盘，另外，3个独立的线程池也各自专用一块硬盘。

缓存之间（其结果就是磁盘之间）的负载均衡使用split_clients模块，`split_clients`非常适用于这个任务。

在 proxy_cache_path指令中设置`use_temp_path=off`，表示NGINX会将临时文件保存在缓存数据的同一目录中。这是为了避免在更新缓存时，磁盘之间互相复制响应数据。

这些调优将带给我们磁盘子系统的最大性能，因为NGINX通过单独的线程池并行且独立地与每块磁盘交互。每块磁盘由16个独立线程和读取和发送文件专用任务队列提供服务。

我敢打赌，你的客户喜欢这种量身定制的方法。请确保你的磁盘也持有同样的观点。

这个示例很好地证明了NGINX可以为硬件专门调优的灵活性。这就像你给NGINX下了一道命令，让机器和数据用最佳姿势来搞基。而且，通过NGINX在用户空间中细粒度的调优，我们可以确保软件、操作系统和硬件工作在最优模式下，尽可能有效地利用系统资源。

7. 总结

综上所述，线程池是一个伟大的功能，将NGINX推向了新的性能水平，除掉了一个众所周知的长期危害——阻塞——尤其是当我们真正面对大量内容的时候。

甚至，还有更多的惊喜。正如前面提到的，这个全新的接口，有可能没有任何性能损失地卸载任何长期阻塞操作。NGINX在拥有大量的新模块和新功能方面，开辟了一方新天地。许多流行的库仍然没有提供异步非阻塞接口，此前，这使得它们无法与NGINX兼容。我们可以花大量的时间和资源，去开发我们自己的无阻塞原型库，但这么做始终都是值得的吗？现在，有了线程池，我们可以相对容易地使用这些库，而不会影响这些模块的性能。

分享：

阅读 63 0

应用服务器技术讨论圈 更多文章

- 东方通加码大数据业务 拟募资8亿收购微智信业 (/html/308/201504/206211355/1.html)
- 玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)
- 金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)
- GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)
- Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

猜您喜欢

- 有勇气，来狮城互联网看看？ (/html/193/201608/2650712579/1.html)
- Joyent CTO谈容器在2016年亟需改变的问题 (/html/175/201601/403508930/1.html)
- 成为高级程序员的10个步骤 (/html/339/201501/202738248/1.html)
- Android Studio和Genymotion安装常见错误 (/html/203/201603/401921220/1.html)
- Linux下查看进程IO工具iopp (/html/357/201606/2247483917/1.html)