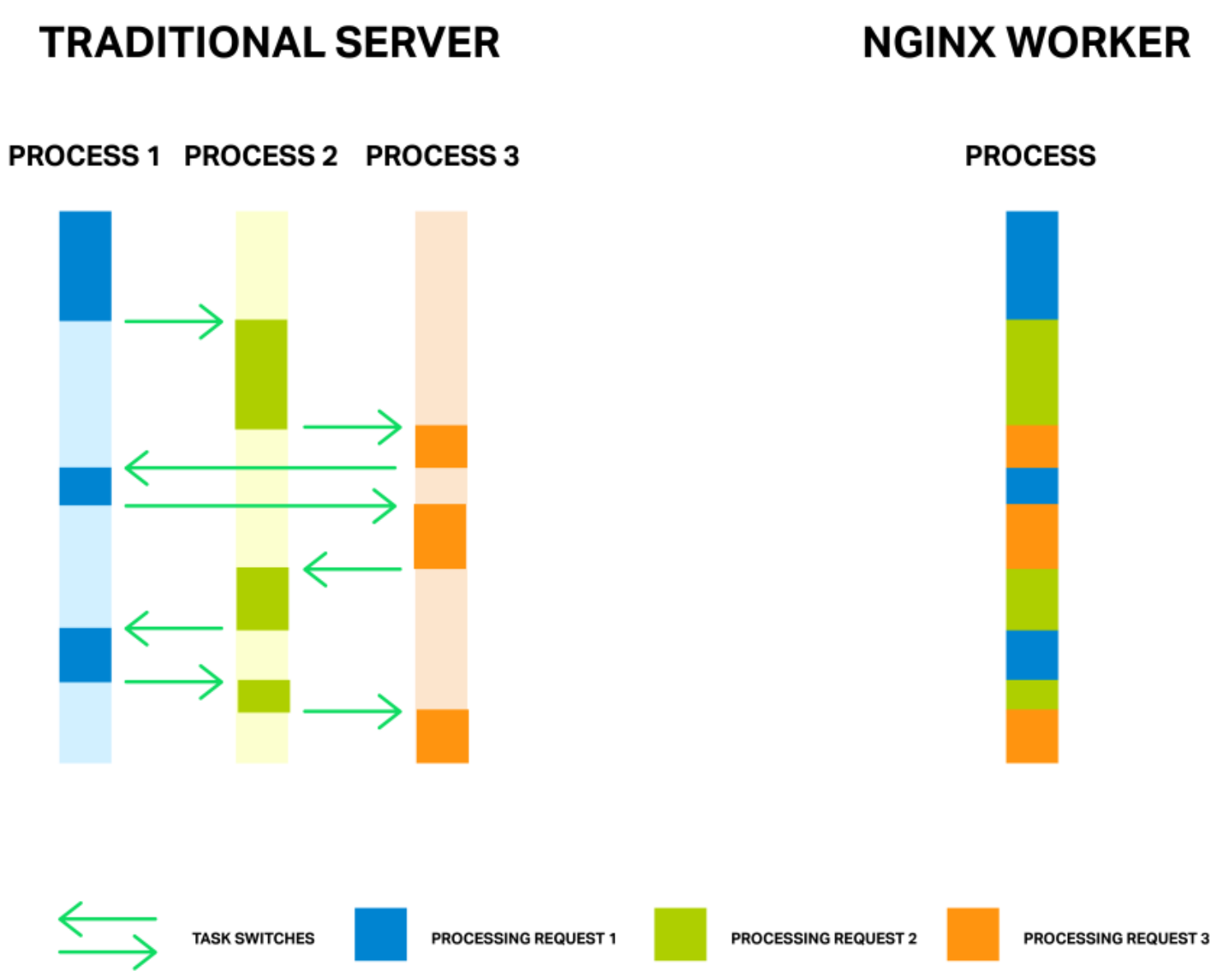
nginx 中的线程池使得性能提升 9 倍

众所周知，Nginx 使用 异步, 事件驱动来接收连接 。这就意味着对于每个请求不会新建一个专用的进程或者线程（就像传统服务端架构一样），它是在一个工作进程中接收多个连接和请求。为了达成这个目标，Nginx 用在一个非阻塞模式下的 sockets 来实现，并使用例如 epoll 和 kqueue 这样高效的方法。

因为满载的工作进程数量是很少的(通常只有一个 CPU 内核)而且固定的，更少的内存占用，CPU 轮训也不会浪费在任务切换上。这种连接方式的优秀之处已众所周知地被 Nginx 自身所证实。它非常成功地接受了百万量级的并发请求。



每个进程都消耗额外的内存，并且每个切换都消耗 CPU 切换和缓存清理

但是异步，事件驱动连接依旧有一个问题。或者，我喜欢称它为“敌人”。这

个敌人的名字叫：阻塞。不幸的是许多第三方模块使用阻塞调用，而且用户（有时候也有模块的开发者自己）并没有意识到有什么不妥之处。阻塞操作可以毁了 Nginx 的性能，必须要避免这样的代价。

甚至在当前的 Nginx 官方代码中在每种情况中避免阻塞调用也是不可能的，为了解决这个问题，新的线程池装置已经在 Nginx 的1.7.11 和 Nginx Plus Release 7 中实现。它是什么，如何使用，我们一会儿再介绍，现在我们来直面我们的敌人。

编者：如果需要了解一下 Nginx Plus R7，请看我们博客中的 Announcing Nginx Plus R7

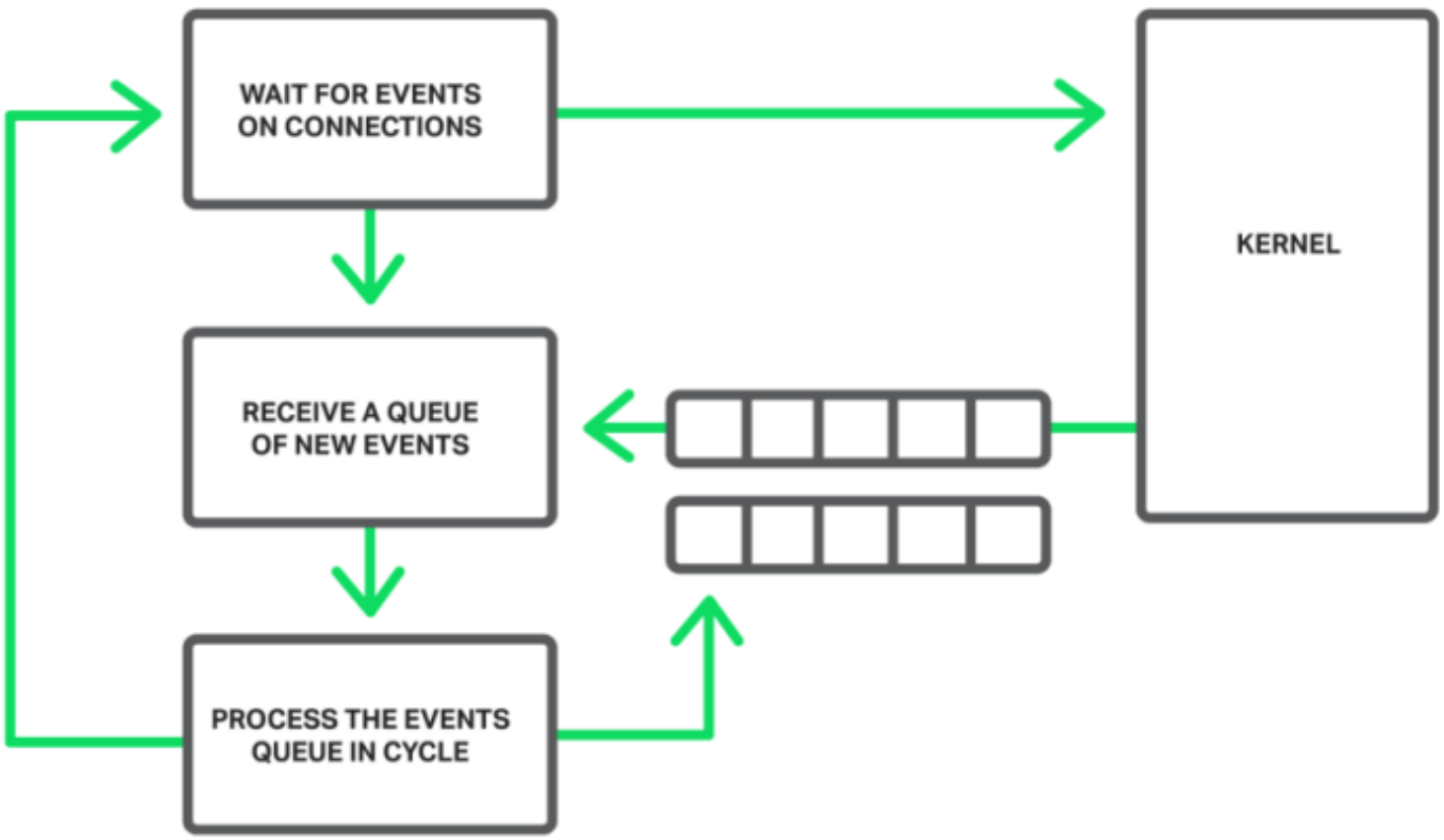
需要了解 Nginx Plus R7 里的新特性，请看这些相关文章

- HTTP/2 Now Fully Supported in NGINX Plus
- Socket Sharding in NGINX
- The New NGINX Plus Dashboard in Release 7
- TCP Load Balancing in NGINX Plus R7

这个问题

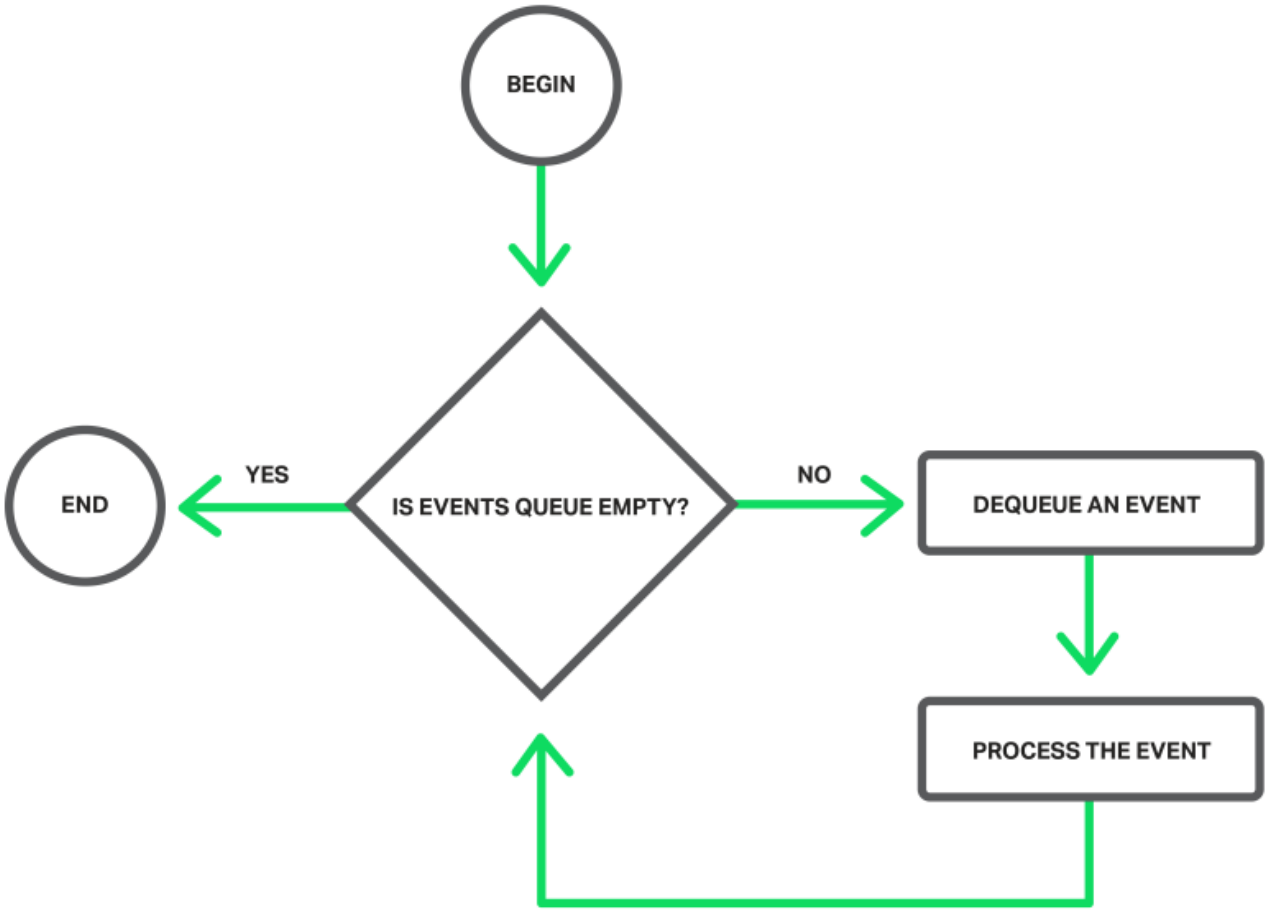
首先，为了更好地理解问题所在，我们需要用简单的话解释一下 Nginx 如何工作的。

通常来说，Nginx 是一个事件处理器，一个从内核中接受所有连接时发生的事件信息，然后给出对应的操作到操作系统中，告诉它应该做什么。事实上，Nginx 在操作系统进行常规的读写字节的时候，通过调度操作系统把所有难做的工作都做了。因此，Nginx 及时，快速的返回响应是非常重要的。



工作进程从内核中监听并执行事件

这些事件可以是超时，提示说可以从 sockets 里面读数据或写数据，或者是一个错误被触发了。Nginx 接收一堆事件然后一个个执行，做出必要的操作。这样所有的过程都在一个线程中通过一个简单的循环队列完成。Nginx 从一个队列中推出一个事件 然后响应它，例如读写 socket 数据。在大多数情况下，这一过程非常的快(或许只是需要很少的 CPU 轮询从内存中拷贝一些数据)而且 Nginx 继续执行队列中的所有事件非常的快。

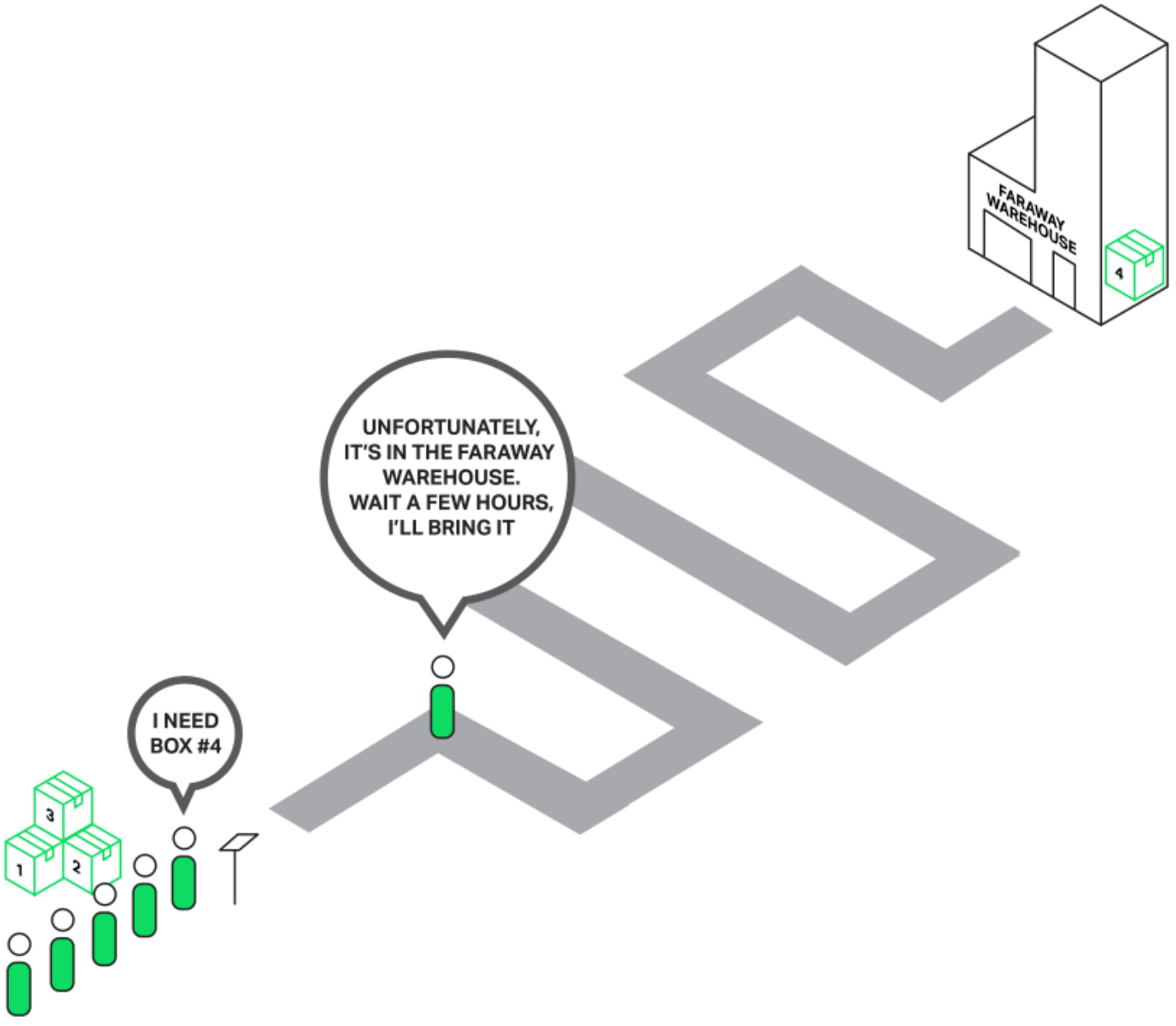


所有的过程都在一个线程中的一个简单循环中完成

但是如果某个耗时而且重量级的操作被触发了会发生什么呢？整个事件循环系统会有一个很扯淡的等待时间，直到这个操作完成。

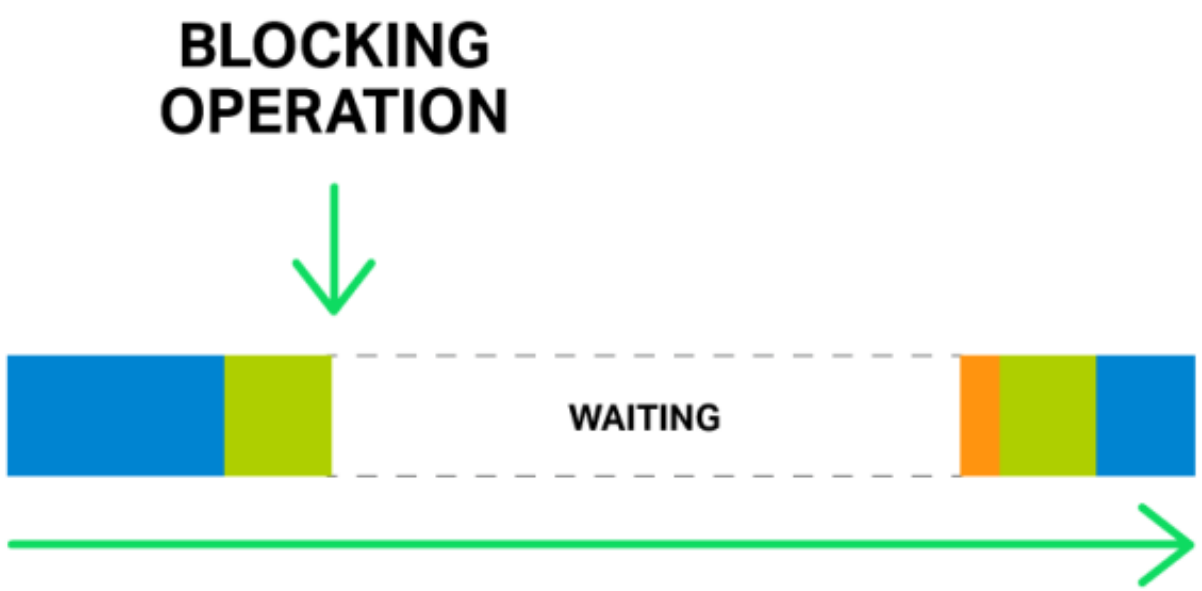
所以，我们说的“一个阻塞操作”的意思是任何一个会占用大量时间，使接收事件的循环暂停的操作。操作可以因为很多原因被阻塞。例如，Nginx 或许会因为长时间的 CPU 密集型操作而忙碌，或者是不得不等待一个资源访问(如硬盘访问，或者一个互斥的或函数库从数据库里用同步操作的方式获取返回这种)。关键是当做这些操作的时候，子进程无法做其他任何事情，也不能接收其他的事件响应，即使是有很多的系统资源是空闲的，而且一些队列里的事件可以利用这些空闲资源的时候。

想象一下，一个店里的销售人员面对着一个长长的队列，队列里的第一个人跟你要不在店里，而是在仓库里的东西��。这个销售人员得跑去仓库提货。现在整个队列一定是因为这次提货等了好几个小时，而且队列里的每个人都很不开心。你能想象一下队列里的人会做出什么反应么？在这等待的这几个小时里面，队列中等待的人在增加，每个人都等着很可能就在店里面的想要的东西（而不是仓库里的）



队列中的每个人都在等在第一个人的订单完成

Nginx 里面所发生的事情跟这个情况是很相似的。当读取一个并不在内存中缓存，而是在硬盘中的文件的时候。硬盘是很慢的（尤其是正在转的那个），而其他的在队列中的请求可能并不需要访问硬盘，结果他们不得不等待。结果是延迟在增加，但是系统资源并没有满负荷。



Just one blocking operation can delay all following operations for a significant time

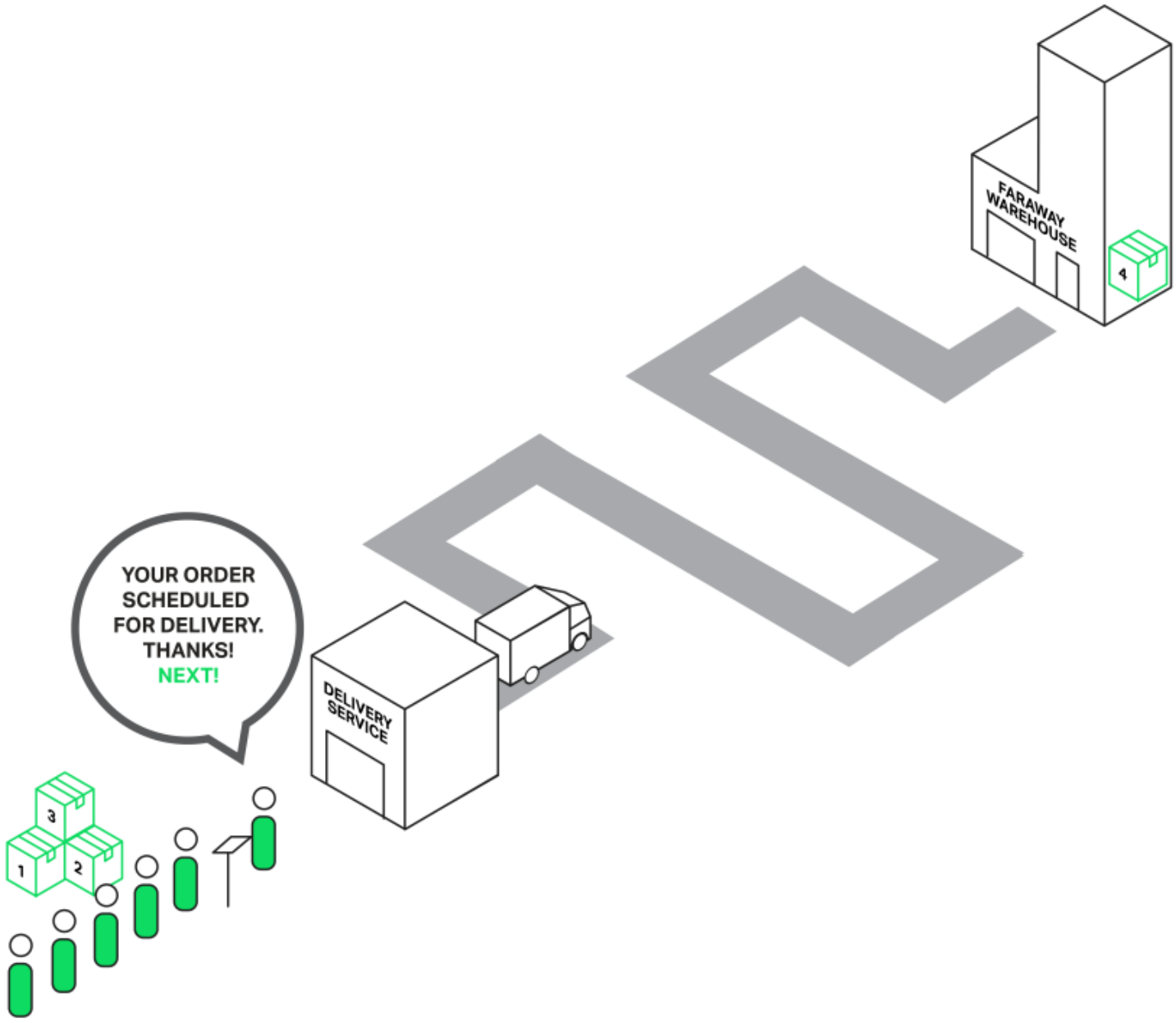
一些操作系统提供了一个异步接口去读取和发送文件，Nginx 使用了这个接口（详情查看 aio 指令。这里有个很好的例子就是 FreeBSD，不幸的是，我们不能保证所有的 Linux 都是一样的。尽管 Linux 提供了一种读取文件的异步接口，然而仍然有一些重要的缺点。其中一个就是文件和缓冲区访问的对齐问题，而 Nginx 就能很好地处理。第二个问题就很严重了。异步接口需要 *O_DIRECT* 标志被设置在文件描述中。这就意味着任意访问这个文件都会通过内存缓存，并增加硬盘的负载。在大多数情况下这并不能提升性能。

为了着重解决这些问题，在 Nginx 1.7.11 和 Nginx Plus Release 7 中加入了线程池。

现在让我们深入介绍一些线程池是什么，它是如何工作的。

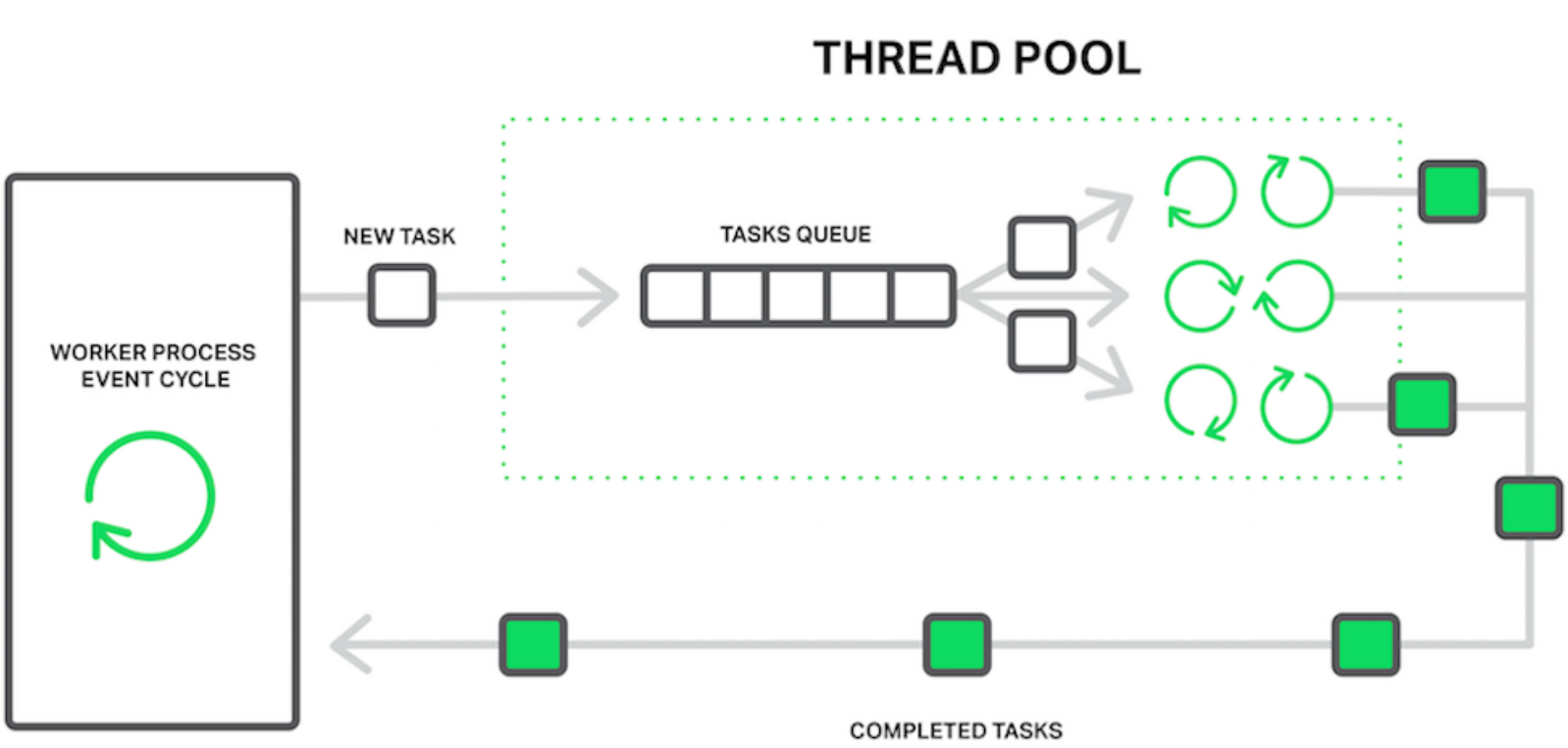
线程池

现在我们重新扮演那个可怜的要从很远的仓库提货的售货员。但是现在他变聪明了(或者是因为被一群愤怒的客户揍了一顿之后变聪明了？)并且招聘了一个快递服务。现在有人要远在仓库的产品的时候，他不需要自己跑去仓库提货了，他只需要扔个订单给快递服务，快递会处理这个订单，而售货员则继续服务其他客户。只有那些需要需要远在仓库的产品的客户需要等待递送，其他人则可以立即得到服务。



Passing an order to the delivery service unblocks the queue

在这个方面，Nginx 的线程池就是做物流服务的，它由一个任务队列和很多处理队列的线程组成。当一个 worker 要做一个耗时很长的操作的时候，它不需要自己去处理这个操作，而是把这个任务发到线程队列中，当有空闲的线程的时候它会被拿出来去处理。



The worker process offloads blocking operations to the thread pool

看起来我们有了另一条队列。是的。但是这个队列只被具体的资源所限制。我们读数据不能比生产数据还要快。现在，至少是这趟车不会阻塞其他事件进程了，而且只有需要访问文件的请求会等待而已。

从硬盘读取文件的操作是一个在阻塞例子中常用的栗子，然而事实上在Nginx 中实现的线程池可以做任何不适合在主工作轮询中执行的任务。

那个时候，扔到线程池里只有三个基本操作：大多数操作系统的 read() 系统调用, Linux 的 sendfile 和 Linux 中调用 aio_write() 去写入一些例如缓存的临时文件。我们将继续测试并对实现做性能基准测试，而且我们未来将会把更多的可以获得明显收益的其他操作也放到线程池中。

编辑注：对 aio_write() 的支持在 Nginx 1.9.13 和 Nginx Plus R9 中添加。

基准测试

是时候从理论转为实际了。为了证明使用线程池的效果，我们做了一个模拟阻塞和非阻塞操作混合中最坏情况的合成基准测试。

这需要一组不适合放在内存中的数据集。一台拥有 48G RAM的机器上，我们用 4MB 的文件生成了 256GB 的随机数据，随后配置 Nginx 1.9.0 并打开服务。

配置项非常简单：

```
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;
        }
    }
}
```

就像你看到的那样，为了得到更好的性能，我们对一些配置做了调整：logging 和 accept_mutex 被关掉了，sendfile 打开了， sendfile_max_chunk 也设置了。最后一个指令可以减少阻塞 sendfile 调用的最大花费时间，因为 Nginx 不会一次性发送整个文件，而是分成 512KB 的小块。

这个机器有两个 Intel Xeon E5645(共计 12 核，24 线程)，有一个 10-Gps 网卡，硬盘子系统是由4块西部数码的 WD 1003FBYX 硬盘用 RAID10 阵列组成。所有这些硬件由 Ubuntu Server 14.04.1 LTS 提供支持。

客户端由两台相同规格的机器组成，其中一台机器上， wrk 创建Lua脚本，这个脚本从服务器上以乱序发起并发请求，获取文件，并发量是 200 个连接。而且每个请求结果很可能并不能命中缓存，需要阻塞地从硬盘中读取文件，我们称这个负载为 **随机负载**

我们的第二个客户端机器会执行另一个 wrk。它会多次请求同一个文件，以 50 个并发请求。因为这个文件被经常访问，所以它会被一直放在内存中。通常情况下，Nginx 对这些请求会响应得非常快。所以我们叫这个负载是 **固定负载**

性能测评会被在服务器上的用 ifstat 监控的吞吐量以及两个客户端的 wrk 结果作为标准。

现在，首先运行没有线程池的那个，结果并不是很满意。

```
% ifstat -bi eth2
eth2
Kbps in  Kbps out
5531.24  1.03e+06
4855.23  812922.7
5994.66  1.07e+06
5476.27  981529.3
6353.62  1.12e+06
5166.17  892770.3
5522.81  978540.8
6208.10  985466.7
6370.79  1.12e+06
6123.33  1.07e+06
```

正如你所看到的那样，以这个配置，服务器大概总共有 1Gbps 的吞吐量。top 的输出表明大部分 worker 进程都耗时在了阻塞 输入/输出 操作上(D 状态下):

```
top - 10:40:47 up 11 days,  1:32,  1 user,  load average: 49.61, 45.77 62.8
Tasks: 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.
KiB Mem:  49453440 total, 49149308 used,   304132 free,    98780 buffers
KiB 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 top
25921 vbart    20   0  121956   2232   1056 S   0.0  0.0   0:01.97 sshd
25923 vbart    20   0   40304   4160   2208 S   0.0  0.0   0:00.53 zsh
```

这种情况下，吞吐量受限于硬盘系统，CPU 则没事做，wrk 返回的结果表示非常的慢：

```
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 read
Requests/sec:      8.08
Transfer/sec:     34.07MB
```

记住，这些文件应该被放在内存中！过大的延迟是因为所有的工作进程在从硬盘中读文件的时候非常的繁忙，它们在应对第一个客户端 *随机负载* 发出的200个并发请求。而不能在合适的时间内响应我们的请求。

是时候把线程池放进来啦。为了加入我们只需要添加 aio thread 指令到 location 中。

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

然后让 Nginx 去读取这个配置项。

然后我们重复测试：

```
% ifstat -bi eth2
eth2
Kbps in  Kbps out
60915.19  9.51e+06
59978.89  9.51e+06
60122.38  9.51e+06
```

```
61179.06  9.51e+06
61798.40  9.51e+06
57072.97  9.50e+06
56072.61  9.51e+06
61279.63  9.51e+06
61243.54  9.51e+06
59632.50  9.50e+06
```

现在我们的服务器产生了 9.5Gbps , 对比一下没有线程池的 1Gbps 左右的结果。

它可能会更高，但是这个数值已经到达了最大物理网卡容量了。所以在这个测试中， Nginx 受限于网卡接口。工作进程大部分时候都沉睡并等待心得事件(*top* 命令 S 模式下):

```
top - 10:43:17 up 11 days,  1:35,  1 user,  load average: 172.71, 93.84, 77
Tasks: 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.
KiB Mem:  49453440 total, 49096836 used,   356604 free,    97236 buffers
KiB 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 nginx
25921 vbart     20   0  121956   2176   1000 S   0.0  0.0   0:01.98 sshd
25923 vbart     20   0   40304   3840   2208 S   0.0  0.0   0:00.54 zsh
```

仍然有大量的 CPU 资源。

wrk的结果：

```
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 read
Requests/sec:    250.57
Transfer/sec:      0.98GB
```

响应 4MB 文件的平均时间从 7.42 秒降低到了 226.32 毫秒(少了33倍)。每秒请求量增加了31倍(250 vs 8)！

解释就是我们的请求不再等着事件队列去由 worker 进程执行读取的阻塞操作。而是被空闲的线程处理。硬盘系统尽其所能去干活的时候它也能从第一台机器上发出的随机负载请求。 Nginx 使用剩余的 CPU 资源和网络容量去响应第二个客户端的请求，从内存里拿到数据。

仍旧不是银弹

出于对阻塞操作的恐惧并取得了一些令人欣喜的结果之后，你们大多数可能已经准备去在你们的服务器上配置线程池了，别急。

真相是这样的。大多数读文件和发送文件都很幸运地不会去处理慢速硬盘。如果你有足够的RAM去存储数据集，之后操作系统都会很聪明地存储那些经常访问的文件，这叫做“page cache”。

页缓存活很棒，使得 Nginx 在大多数情况下都能表现出极佳的性能。从页缓存中读数据非常的快，以至于没人把它当作是”阻塞”操作。并一方面，扔到线程池会有一些开销的。

所以如果你有足够的 RAM，而且你的执行数据不会非常大的时候，Nginx 在没有线程池的情况下已经做了最好的优化。

把读操作扔在线程池里是针对特定任务的技术手段，当经常访问的内容的空间和操作系统的 VM 缓存不匹配的时候很有用。情况可能是这样的，例如负载很大的，以 Nginx 为基础的流媒体服务器，这就是我们做基准测试时候的情况。

如果我们可以提升读操作放到线程池的性能就很好了。我们所需要的是一个有效的方式去知道需要的数据是不是在内存中，而且只有第二种情况我们应该把读操作分到另一个线程中去。

继续回到销售员的比较上，当前销售员并不知道顾客要的商品是不是在店里，所以它要么把所有的订单都给快递服务，要么自己接管所有订单。

罪魁祸首就是操作系统丢掉了这些新功能。第一个把它以 fincore 系统调用加到 Linux 上的尝试发生在 2010 年，但是并没有结果。随后有数次尝试，作为一个新的 preadv2() 带着 RWF_NONBLOCK 标志的系统调用实现(详情查看 LWN.net 上的信息 Non-blocking buffered file read operations 和 Asynchronous buffered read operations)。所有这些补丁的命运仍旧是不清楚。令人伤心的是这里有一个主要原因，为什么这些补丁至今仍然没有被内核所接受， 继续被放逐

另一方面，FreeBSD 的用户一点儿都不用担心，FreeBSD 早就有了一套足够好的异步读取文件的接口，你应该用它来替换线程池。

配置线程池

那么如果你确定你的情况在配置线程池后会获得一些收益，那么时时候去深入了解这些配置了。

它的配置非常的简单灵活。第一件事就是你得有 Nginx 1.7.11 及其以上的版本，带着 *–with-threads* 参数编译到 *configure* 命令上。Nginx Plus 用户需要版本 7 及其以上。最简单的情况下，配置看来来很普通，你所做的就是把 aio threads 指令配置到合适的上下文中。

```
# in the 'http', 'server', or 'location' context
aio threads;
```

这是关于线程池的最少配置了。实际上它是下面这个配置的缩减版。

```
# in the 'main' context
thread_pool default threads=32 max_queue=65536;

# in the 'http', 'server', or 'location' context
aio threads=default;
```

它定义了一个叫做 default 的线程池，这个线程池有32个工作线程，并有最大的任务队列 —— 65536 个任务。如果任务队列超出了，Nginx 会拒绝请求，并记录这个错误：

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

这个错误意味着可能是因为线程并不能处理这些工作处理得足够快，快过添加到队列中。你可以试着增加队列最大值，但如果这么做没什么用的话，它就意味着你的系统不能提供如此之多的连接容量。

你可能早就注意到了，带着 thread_pool 指令，你可以配置线程的数量，队列的最大长度，还有特定的线程池的名称。最后一个提示就是，你可以配置多个独立的线程池，并在你配置项的不同地方去使用，针对不同的目的：

```
# in the 'main' context
thread_pool one threads=128 max_queue=0;
thread_pool two threads=32;

http {
    server {
        location /one {
            aio threads=one;
        }

        location /two {
            aio threads=two;
        }

    }
    # ...
}
```

如果 **max_queue** 没有指定，默认值是 65536。如图所示，你也可以配置 **max_queue** 到 0。这种情况下线程池只能处理所配置的线程一样多的任务; 队列中不会有等待的任务。

现在想象一下你有一个带着三块硬盘的服务器，而且你想让这台服务器作为一台 *缓存服务器* ，从后端拿到的所有响应存储起来的那种。那么缓存量是

远远超过内存容量的。它实际上就是你的一个个人 CDN 缓存节点。当然，在这种情况下，从硬盘获得巨大的性能提升就显得尤为重要。

你的其中一个选项是调整 RAID 阵列，这种方式有它自己的优缺点。现在你既然有 Nginx，那么你可以用另一种方式了：

```
# 我们假定每个硬盘在这些目录中挂载
# /mnt/disk1, /mnt/disk2, or /mnt/disk3

# in the 'main' context
thread_pool pool_1 threads=16;
thread_pool pool_2 threads=16;
thread_pool pool_3 threads=16;

http {
    proxy_cache_path /mnt/disk1 levels=1:2 keys_zone=cache_1:256m max_size=
                     use_temp_path=off;
    proxy_cache_path /mnt/disk2 levels=1:2 keys_zone=cache_2:256m max_size=
                     use_temp_path=off;
    proxy_cache_path /mnt/disk3 levels=1:2 keys_zone=cache_3:256m max_size=
                     use_temp_path=off;

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

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

在这份配置文件中， **thread_pool** 指令为每个硬盘定义了一个专有独立的线程池。添加了 proxy_cache_path 指令为每个硬盘定义了专有，独立的缓存。

split_clients 模块用来平衡多个缓存的负载(同时也是硬盘的负载)，它非常适

合这个任务。

proxy_cache_path指令中的 **use_temp_path=off** 参数告诉 Nginx 把临时文件保存到和响应缓存相同的目录中。要避免在在更新缓存的时候在硬盘间相互拷贝响应数据。

所有这些配置一起使我们可以在当然的硬盘子系统中获得最大的性能收益。因为 Nginx 通过对不同硬盘开辟的并行，独立的线程池。每个硬盘有16个独立线程和一个读写文件的专用任务队列。

我敢打赌，你的客户一定喜欢这个量身定制的方法。不过要确保你的硬盘和例子里的一样。

这个例子非常好地展示了 Nginx 可以多么灵活地专门为你的硬件做出调整。就像你给了 Nginx 一个硬件和你的数据集如何交互的最佳实践手册。而且 Nginx 也在用户空间上做了很棒的协调，你可以确保你的软件，操作系统以及硬件结合在一起以最佳的模式尽可能高效地运用你所有的系统资源。

结论

总的来说，线程池是一个非常伟大的特性，它使得 Nginx 在性能上达到了一个新的高度，它解决了一个著名而且持久的敌人 —— 阻塞，尤其是在谈论非常大体积的内容时。

还会有更多的东西到来的。正如前面所提到的，这个新的接口是很有潜力的，它允许我们把任何的耗时阻塞操作扔到线程池里而不损失性能。 Nginx 为大量的新模块和心功能带来了新的希望。仍然有大量受欢迎的库不支持异步非阻塞接口，此前它们和 Nginx 并不兼容。我们会花大量的时间和资源为一些库开发我们自己的，新的非阻塞接口。但它值得我们付出这些努力么？现在，线程池已经就绪了，使用这些库相对来说可能变得更简单了，使用这些模块也不会很影响性能。

敬请关注。

来自： https://annatarhe.github.io/2017/08/11/Thread-Pools-in-NGINX-Boost-Performance-9x.html

扩展阅读

Nginx线程池性能提升9倍（Thread Pools in NGINX Boost Performance 9x!）
Reddit月浏览量从百万扩容到十亿的陷阱和教训
亿级短视频社交美拍架构实战
优酷、YouTube、Twitter及JustinTV视频网站架构设计
Reddit: 从扩展到每月10亿页面浏览量的过程中的各种失误中学到的经验教训

为您推荐

使用Nginx+Lua(OpenResty)开发高性能Web应用
使用Node.js+Socket.IO搭建WebSocket实时应用
web安全实战
nginx常用代理配置
CentOS 7.2基于Kubernetes部署简单应用示例

更多

Nginx
线程池
Nginx