



Nginx的并发能力在同类型网页服务器中的表现，相对而言是比较好的，因此受到了很多企业的青睐，我国使用Nginx网站的知名用户包括腾讯、淘宝、百度、京东、新浪、网易等等。

Nginx是网页服务器运维人员必备技能之一，下面为大家整理了一些比较常见的Nginx相关面试题

注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，请从下面的链接获取：[码云](#) 或者 [语雀](#)

聊聊：什么是Nginx？

Nginx是一个web服务器和反向代理服务器，用于HTTP、HTTPS、SMTP、POP3和IMAP协议。

Nginx—Ngine X，是一款免费的、自由的、开源的、高性能HTTP服务器和反向代理服务器；

也是一个IMAP、POP3、SMTP代理服务器；

Nginx以其高性能、稳定性、丰富的功能、简单的配置和低资源消耗而闻名。

也就是说Nginx本身就可以托管网站（类似于Tomcat一样），进行Http服务处理，也可以作为反向代理服务器 、负载均衡器和HTTP缓存。

Nginx 解决了服务器的C10K（就是在一秒之内连接客户端的数目为10k即1万）问题。

它的设计不像传统的服务器那样使用线程处理请求，而是一个更加高级的机制—事件驱动机制，是一种异步事件驱动结构。

聊聊： Nginx的一些特性。

Nginx服务器的特性包括：

反向代理/L7负载均衡器

嵌入式Perl解释器

动态二进制升级

可用于重新编写URL，具有非常好的PCRE支持

聊聊： Nginx的优缺点？

核心优点：

占内存小，可实现高并发连接，处理响应快

可实现http服务器、虚拟主机、方向代理、负载均衡

Nginx配置简单

可以不暴露正式的服务器IP地址

核心缺点：

动态处理差：

nginx处理静态文件好,耗费内存少，但是处理动态页面则很鸡肋，

现在一般前端用nginx作为反向代理抗住压力。

聊聊： Nginx应用场景？

http服务器。

Nginx是一个http服务可以独立提供http服务。可以做网页静态服务器。

虚拟主机。

可以实现在一台服务器虚拟出多个网站，例如个人网站使用的虚拟机。

反向代理，负载均衡。

当网站的访问量达到一定程度后，单台服务器不能满足用户的请求时，需要用多台服务器集群可以使用nginx做反向代理。

并且多台服务器可以平均分担负载，不会应为某台服务器负载高宕机而某台服务器闲置的情况。

nginz 中也可以配置安全管理、比如可以使用Nginx搭建API接口网关,对每个接口服务进行拦截。

聊聊： 使用“反向代理服务器”的优点是什么？

反向代理服务器可以隐藏源服务器的存在和特征。

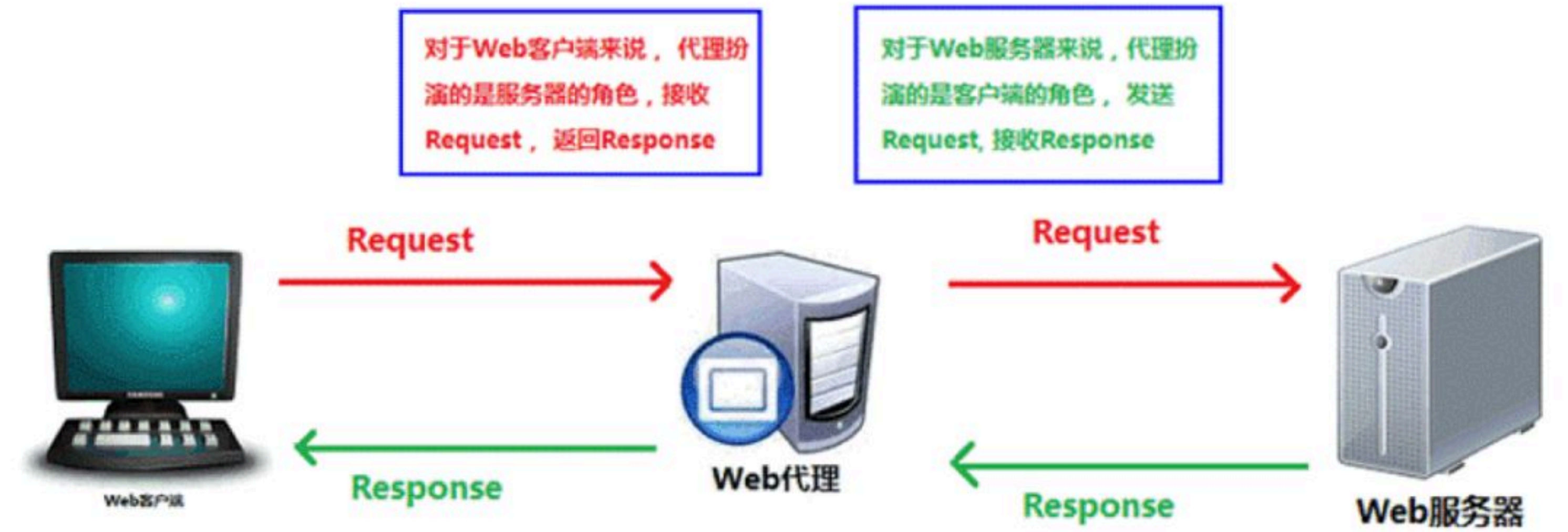
它充当互联网云和web服务器之间的中间层。

这对于安全方面来说是很好的，特别是当您使用web托管服务时。

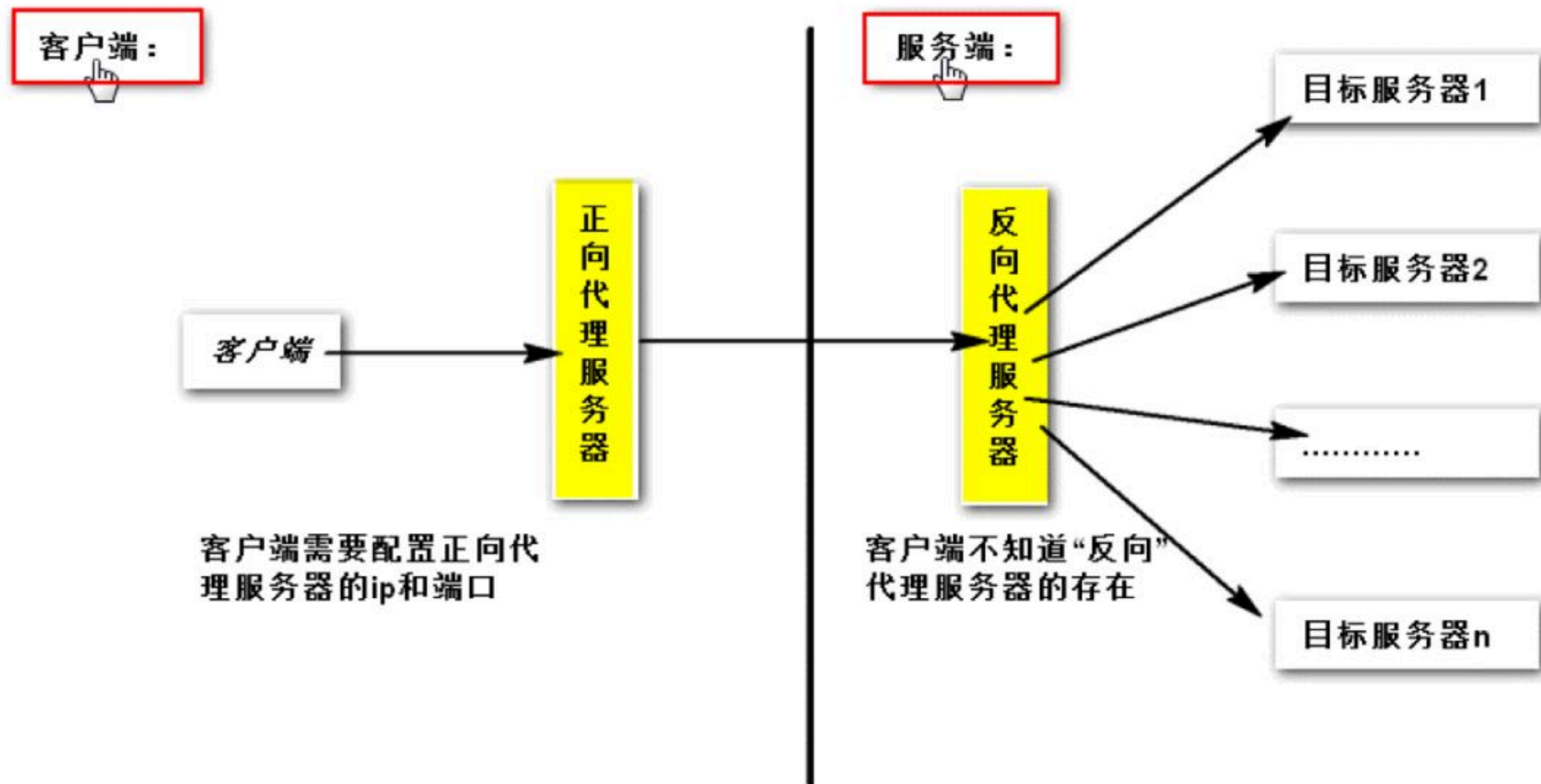
聊聊：什么是正向代理和反向代理？

首先，代理服务器一般指局域网内部的机器通过代理服务器发送请求到互联网上的服务器，代理服务器一般作用在客户端。例如：GoAgentFQ软件。

我们的客户端在进行FQ操作的时候，我们使用的正是正向代理，通过正向代理的方式，在我们的客户端运行一个软件，将我们的HTTP请求转发到其他不同的服务器端，实现请求的分发。

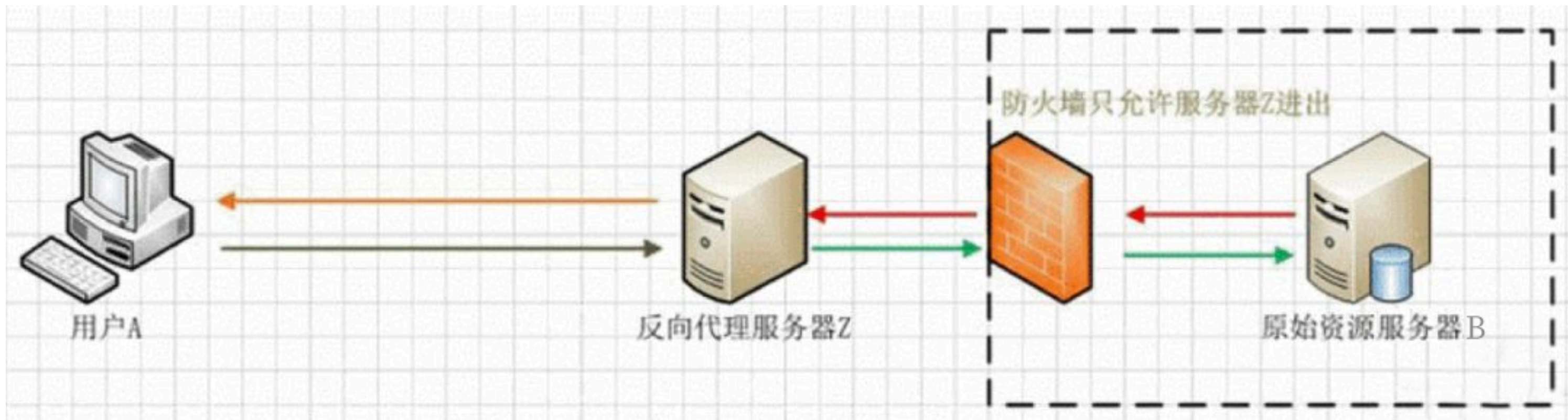


反向代理服务器作用在服务器端，它在服务器端接收客户端的请求，然后将请求分发给具体的服务器进行处理，然后再将服务器的相应结果反馈给客户端。Nginx就是一个反向代理服务器软件。



从上图可以看出：客户端必须设置正向代理服务器，当然前提是要知道正向代理服务器的IP地址，还有代理程序的端口。

反向代理正好与正向代理相反，对于客户端而言代理服务器就像是原始服务器，并且客户端不需要进行任何特别的设置。客户端向反向代理的命名空间（name-space）中的内容发送普通请求，接着反向代理将判断向何处（原始服务器）转交请求，并将获得的内容返回给客户端。



聊聊：反向代理好处

保护了真实的 web 服务器，web 服务器对外不可见，外网只能看到反向代理服务器，而反向代理服务器上并没有真实数据，因此，保证了 web 服务器的资源安全。

反向代理为基础产生了动静资源分离以及负载均衡的方式，减轻 web 服务器的负担，加速了对网站访问速度。

节约了有限的 IP 地址资源，企业内所有的网站共享一个在 internet 中注册的IP地址，这些服务器分配私有地址，采用虚拟主机的方式对外提供服务。

聊聊：什么是Nginx？它的优势和功能？

Nginx是一个web服务器和方向代理服务器，用于HTTP、HTTPS、SMTP、POP3和IMAP协议。因它的稳定性、丰富的功能集、示例配置文件和低系统资源的消耗而闻名。

优点：

(1) 更快

这表现在两个方面：一方面，在正常情况下，单次请求会得到更快的响应；另一方面，在高峰期（如有数以万计的并发请求），Nginx可以比其他Web服务器更快地响应请求。

(2) 高扩展性，跨平台

Nginx的设计极具扩展性，它完全是由多个不同功能、不同层次、不同类型且耦合度极低的模块组成。因此，当对某一个模块修复Bug或进行升级时，可以专注于模块自身，无须在意其他。而且在HTTP模块中，还设计了HTTP过滤器模块：一个正常的HTTP模块在处理完请求后，会有一串HTTP过滤器模块对请求的结果进行再处理。这样，当我们开发一个新的HTTP模块时，不但可以使用诸如HTTP核心模块、events模块、log模块等不同层次或者不同类型的模块，还可以原封不动地复用大量已有的HTTP过滤器模块。这种低耦合度的优秀设计，造就了Nginx庞大的第三方模块，当然，公开的第三方模块也如官方发布的模块一样容易使用。

Nginx的模块都是嵌入到二进制文件中执行的，无论官方发布的模块还是第三方模块都是如此。这使得第三方模块一样具备极其优秀的性能，充分利用Nginx的高并发特性，因此，许多高流量的网站都倾向于开发符合自己业务特性的定制模块。

（3）高可靠性：用于反向代理，宕机的概率微乎其微

高可靠性是我们选择Nginx的最基本条件，因为Nginx的可靠性是大家有目共睹的，很多家高流量网站都在核心服务器上大规模使用Nginx。Nginx的高可靠性来自于其核心框架代码的优秀设计、模块设计的简单性；另外，官方提供的常用模块都非常稳定，每个worker进程相对独立，master进程在1个worker进程出错时可以快速“拉起”新的worker子进程提供服务。

（4）低内存消耗

一般情况下，10 000个非活跃的HTTP Keep-Alive连接在Nginx中仅消耗2.5MB的内存，这是Nginx支持高并发连接的基础。

（5）单机支持10万以上的并发连接

这是一个非常重要的特性！

随着互联网的迅猛发展和互联网用户数量的成倍增长，各大公司、网站都需要应付海量并发请求，一个能够在峰值期顶住10万以上并发请求的Server，无疑会得到大家的青睐。

理论上，Nginx支持的并发连接上限取决于内存，10万远未封顶。当然，能够及时地处理更多的并发请求，是与业务特点紧密相关的。

（6）热部署

master管理进程与worker工作进程的分离设计，使得Nginx能够提供热部署功能，即可以在7×24小时不间断服务的前提下，升级Nginx的可执行文件。

当然，它也支持不停止服务就更新配置项、更换日志文件等功能。

（7）最自由的BSD许可协议

这是Nginx可以快速发展的强大动力。

BSD许可协议不只是允许用户免费使用Nginx，它还允许用户在自己的项目中直接使用或修改Nginx源码，然后发布。这吸引了无数开发者继续为Nginx贡献自己的智慧。

以上7个特点当然不是Nginx的全部，拥有无数个官方功能模块、第三方功能模块使得Nginx能够满足绝大部分应用场景，这些功能模块间可以叠加以实现更加强大、复杂的功能，有些模块还支持Nginx与Perl、Lua等脚本语言集成工作，大大提高了开发效率。这些特点促使用户在寻找一个Web服务器时更多考虑Nginx。

选择Nginx的核心理由还是它能在支持高并发请求的同时保持高效的服务

聊聊：为什么要用Nginx？

跨平台、配置简单、方向代理、高并发连接：处理2-3万并发连接数，官方监测能支持5万并发，内存消耗小：开启10个nginx才占150M内存，nginx处理静态文件好，耗费内存少，

而且Nginx内置的健康检查功能： 如果有一个服务器宕机，会做一个健康检查，再发送的请求就不会发送到宕机的服务器了。重新将请求提交到其他的节点上。

使用Nginx的话还能：

节省宽带：支持GZIP压缩，可以添加浏览器本地缓存

稳定性高：宕机的概率非常小

接收用户请求是异步的

聊聊： 请解释什么是C10K问题？

C10K问题是指无法同时处理大量客户端(10,000)的网络套接字。

聊聊： C10K问题的本质和解决方案

什么是C10K问题

所谓 c10k 问题，指的是服务器如何支持 10k 个并发连接，也就是 concurrent 10000 connection（这也是 c10k 这个名字的由来）。

由于硬件成本的大幅度降低和硬件技术的进步，如果一台服务器能够同时服务更多的客户端，那么也就意味着服务每一个客户端的成本大幅度降低。从这个角度来看，c10k 问题显得非常有意义。

C10K问题由来

互联网的基础是网络通信，早期的互联网可以说是一个小群体的集合。互联网还不够普及，用户也不多，一台服务器同时在线 100 个用户，在当时已经算是大型应用了，所以并不存在 C10K 的难题。互联网的爆发期是在 www 网站、浏览器出现后。最早的互联网称之为 Web1.0，大部分的使用场景是下载一个 HTML 页面，用户在浏览器中查看网页上的信息，这个时期也不存在 C10K 问题。

Web2.0 时代到来后，就不同了。一方面是，互联网普及率大大提高了，用户群体几何倍增长。另一方面是，互联网不再是单纯地浏览 www 网页，逐渐开始进行交互，而且应用程序的逻辑也变得更复杂。从简单的表单提交，到即时通信和在线实时互动，C10K 的问题才体现出来了。因为每一个用户都必须与服务器保持连接，才能进行实时数据交互。诸如 Facebook 这样的网站，同一时间的并发 TCP 连接很可能已经过亿。

早期的腾讯QQ也同样面临C10K问题，只不过他们是用了UDP这种原始的包交换协议来实现的，绕开了这个难题，当然过程肯定是痛苦的。如果当时有 epoll 技术，他们肯定会用 TCF 实际上，当时也有异步模式，如：select/poll 模型。这些技术都有一定的缺点：selelct 最大不能超过 1024；poll 没有限制，但每次收到数据时，需要遍历每一个连接，查看

这时候问题就来了，最初的服务器都是基于进程/线程模型的，新到来一个 TCP 连接，就需要分配 1 个进程（或者线程）。进程又是操作系统最昂贵的资源，一台机器无法创建很多进程。如果是 C10K，就要创建 1 万个进程，那么就单机而言，操作系统是无法承受的（往往出现效率低下、甚至完全瘫痪）。如果是采用分布式系统，维持 1 亿用户在线需要 10 万台服务器，成本巨大，也只有 Facebook、Google、Apple 等巨头，才有财力购买如此多的服务器。

基于上述考虑，如何突破单机性能局限，是高性能网络编程所必须要直面的问题。这些局限和问题，最早被 Dan Kegel 进行了归纳和总结，并首次系统地分析和提出了解决方案。后来，这种普遍的网络现象和技术局限，都被大家称为 C10K 问题。

C10K问题的本质

C10K 问题，本质上是操作系统的问题。对于 Web1.0/2.0 时代的操作系统而言，传统的同步阻塞 I/O 模型都是一样的，处理的方式都是 requests per second，并发 10K 和 100 的区别关键在于 CPU。

创建的进程、线程多了，数据拷贝频繁（缓存 I/O、内核将数据拷贝到用户进程空间、阻塞），进程/线程上下文切换消耗大，导致操作系统崩溃，这就是 C10K 问题的本质！

可见，解决 C10K 问题的关键就是：尽可能减少 CPU 等核心资源消耗，从而榨干单台服务器的性能，突破 C10K 问题所描述的瓶颈。

C10K问题的解决方案探讨

从网络编程技术的角度来说，主要思路为：

为每个连接分配一个独立的线程/进程。
同一个线程/进程同时处理多个连接（IO 多路复用）。

为每个连接分配一个独立的线程/进程

这一思路最为直接。但是，由于申请进程/线程会占用相当可观的系统资源，同时对于多进程/线程的管理会对系统造成压力，因此，这种方案不具备良好的可扩展性。

这一思路在服务器资源还没有富裕到足够程度的时候，是不可行的。即便资源足够富裕，效率也不够高。

总之，此思路技术实现会使得资源占用过多，可扩展性差，在实际应用中已被抛弃。

同一个线程/进程同时处理多个连接（IO多路复用）

IO 多路复用，从技术实现上，又分很多种。我们逐一来看看下述各种实现方式的优劣。

实现方式1：循环逐个处理各个连接，每个连接对应一个 socket

循环逐个处理各个连接，每个连接对应一个 socket。当所有 socket 都有数据的时候，这种方法是可行的。但是，当应用读取某个 socket 的文件数据不 ready 的时候，整个应用会阻塞在这里，等待该文件句柄 ready，即使别的文件句柄 ready，也无法往下处理。

实现小结：直接循环处理多个连接。

问题归纳：任一文件句柄的不成功会阻塞住整个应用。

实现方式2：使用 select 方法

使用 select 方法解决上面阻塞的问题，思路比较简单。在读取文件句柄之前，先查下它的状态，如果 ready 了，就进行处理；如果不 ready，就不进行处理；这不就解决了这个问题了嘛？于是，有了 select 方案。用一个 fd_set 结构体来告诉内核同时监控多个文件句柄，当其中有文件句柄的状态发生指定变化（例如某句柄由不可用变为可用）或超时，则调用返回。

之后，应用可以使用 FD_ISSET 来逐个查看，确定哪个文件句柄的状态发生了变化。这样做，小规模的连接问题不大，但当连接数很多（文件句柄个数很多）的时候，逐个检查状态就很慢了。因此，select 往往存在管理的句柄上限（FD_SETSIZE）。同时，在使用上，因为只有一个字段记录关注和发生事件，所以每次调用之前，要重新初始化 fd_set 结构体。

```
intselect(intnfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,structtimeval *timeout);
```

实现小结：有连接请求抵达了，再检查处理。

问题归纳：句柄上限+重复初始化+逐个排查所有文件句柄状态，效率不高。

实现方式3：使用 poll 方法

poll 主要解决 select 的前两个问题：

通过一个 pollfd 数组，向内核传递需要关注的事件，以消除文件句柄上限。
使用不同字段分别标注 “关注事件和发生事件”，来避免重复初始化。

实现小结：设计新的数据结构，提高使用效率。

问题归纳：逐个排查所有文件句柄状态，效率不高。

实现方式4：使用 epoll 方法

既然 “poll 逐个排查所有文件句柄状态” 效率不高，很自然的，在调用返回的时候，如果只给应用提供发生了状态变化（很可能是数据 ready）的文件句柄，进行排查的效率就高很多。epoll 采用了这种设计，适用于大规模的应用场景。实验表明：当文件句柄数目超过10之后，epoll 性能将优于 select 和 poll；当文件句柄数目达到 10K 的时候，epoll 已经超过 select 和 poll 两个数量级。

实现小结：只返回状态变化的文件句柄。

问题归纳：依赖特定平台（Linux）。

因为 Linux 是互联网企业中使用率最高的操作系统，所以 Epoll 就成为 “C10K killer、高并发、高性能、异步非阻塞” 这些技术的代名词了。FreeBSD 推出了 kqueue，Linux 推出了 epoll，Windows 推出了 IOCP，Solaris 推出了 /dev/poll。这些操作系统提供的功能，就是为了解决 C10K 问题。epoll 技术的编程模型就是异步非阻塞回调，也可以叫做 Reactor、事件驱动、事件轮循（EventLoop）。Nginx、libevent、node.js 这些就是 Epoll 时代的产物。

实现方式5：使用 libevent 库

由于 epoll、kqueue、IOCP 每个接口都有自己的特点，程序移植非常困难，所以需要对这些接口进行封装，以让它们易于使用和移植，其中 libevent 库就是其中之一。跨平台，封装底层平台的调用，提供统一的 API，但底层在不同平台上自动选择合适的调用。

按照 libevent 的官方网站，libevent 库提供了以下功能：当一个文件描述符的特定事件（如可读，可写或出错）发生了，或一个定时事件发生了，libevent 就会自动执行用户指定的回调函数，来处理事件。目前，libevent 已支持以下接口 /dev/poll、kqueue、event ports、select、poll 和 epoll。Libevent 的内部事件机制完全是基于所使用的接口的。因此，libevent 非常容易移植，也使它的扩展性非常容易。目前，libevent 已在以下操作系统中编译通过：Linux、BSD、Mac OS X、Solaris 和 Windows。使用 libevent 库进行开发非常简单，也很容易在各种 unix 平台上移植。

聊聊：Nginx如何实现超高并发？

面试官心理分析

主要是看应聘人员的对NGINX的基本原理是否熟悉，因为大多数人多多少少都懂点NGINX，但是真正其明白原理的可能少之又少。

明白其原理，才能做优化，否则只能照样搬样，出了问题也无从下手。

懂皮毛的人，一般会做个 Web Server，搭建一个 Web 站点；

初级运维可能搞个 HTTPS 、配置一个反向代理; 中级运维定义个 upstream、写个正则判断；

老鸟做个性能优化、写个ACL，还有可能改改源码。

面试题剖析

异步，非阻塞，使用了epoll 和大量的底层代码优化。

如果一个server采用一个进程负责一个request的方式，那么进程数就是并发数。正常情况下，会有很多进程一直在等待中。

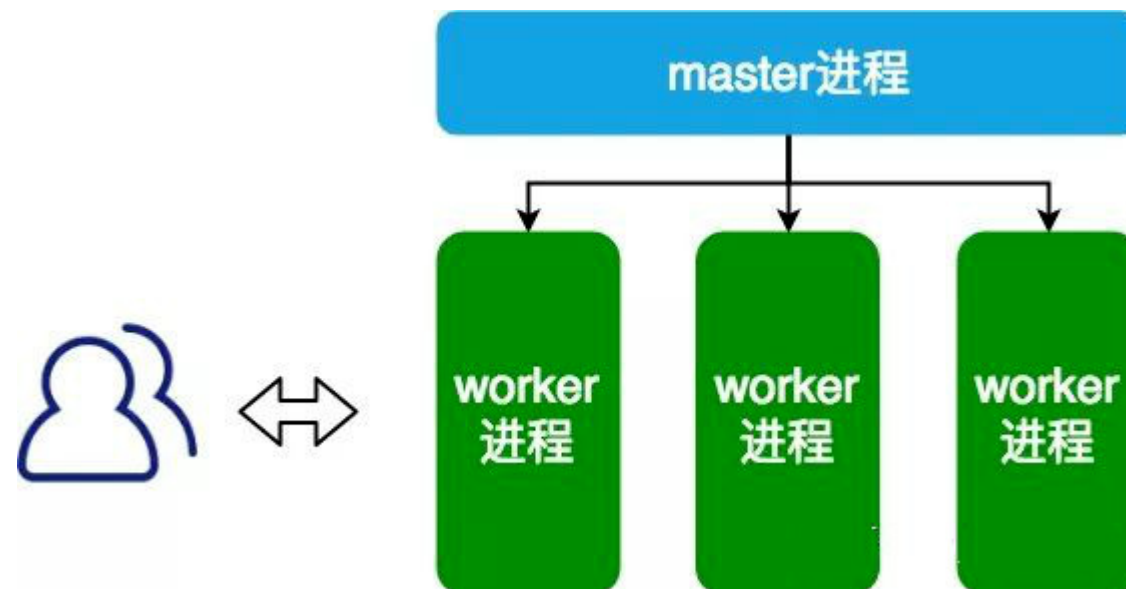
而nginx采用一个master进程，多个woker进程的模式。

master进程主要负责收集、分发请求。每当一个请求过来时，master就拉起一个worker进程负责处理这个请求。

同时master进程也负责监控woker的状态，保证高可靠性

woker进程一般设置为跟cpu核心数一致。nginx的woker进程在同一时间可以处理的请求数只受内存限制，可以处理多个请求。

Nginx 的异步非阻塞工作方式正把当中的等待时间利用起来了。在需要等待的时候，这些进程就空闲出来待命了，因此表现为少数几个进程就解决了大量的并发问题。



每进来一个request，会有一个worker进程去处理。但不是全程的处理，处理到什么程度呢？处理到可能发生阻塞的地方，比如向上游（后端）服务器转发request，并等待请求返回。那么，这个处理的worker很聪明，他会在发送完请求后，注册一个事件：“如果upstream返回了，告诉我一声，我再接着干”。于是他就休息去了。

此时，如果再有request 进来，他就可以很快再按这种方式处理。而一旦上游服务器返回了，就会触发这个事件，worker才会来接手，这个request才会接着往下走。

聊聊：Nginx如何实现超高并发？

Nginx 是一个高性能的 Web 服务器，能够同时处理大量的并发请求。

它结合多进程机制和异步机制，异步机制使用的是异步非阻塞方式，

接下来就给大家介绍一下 Nginx 的多线程机制和异步非阻塞机制。

1、多进程机制

服务器每当收到一个客户端时，就有 服务器主进程 （master process）生成一个 子进程（worker process）出来和客户端建立连接进行交互，直到连接断开，该子进程就结束了。

使用进程的好处是各个进程之间相互独立，不需要加锁，减少了使用锁对性能造成影响，同时降低编程的复杂度，降低开发成本。其次，采用独立的进程，可以让进程互相之间不会影 响，如果一个进程发生异常退出时，其它进程正常工作，master 进程则很快启动新的 worker 进程，确保服务不会中断，从而将风险降到最低。

缺点是操作系统生成一个子进程需要进行 内存复制等操作，在资源和时间上会产生一定的开销。当有大量请求时，会导致系统性能下降。

2、异步非阻塞机制

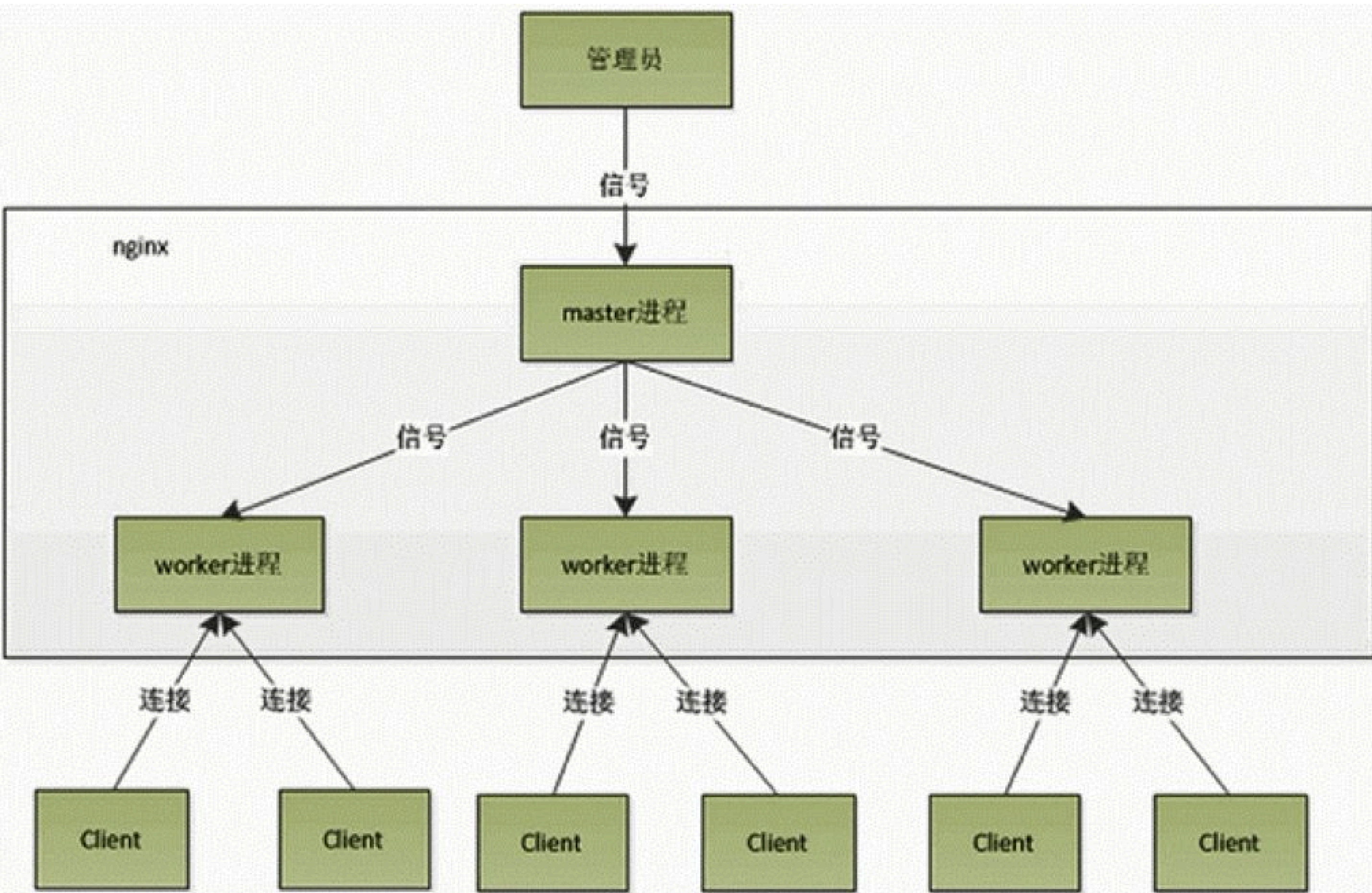
每个工作进程 使用 异步非阻塞方式，可以处理 多个客户端请求。

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 处理其他请求 （即为 非阻塞 ）；而 客户端 在此期间也 无需等待响应 ， 可以去处理其他事情（即为 异步 ）。

当 IO 返回时，就会通知此 工作进程 ； 该进程得到通知，暂时 挂起 当前处理的事务去 响应客户端请求 。

聊聊： 请解释Nginx服务器上的Master和Worker进程分别是什么？

主程序 Master process 启动后，通过一个 for 循环来 接收 和 处理外部信号 ；
主进程通过 fork() 函数产生 worker 子进程 ， 每个子进程执行一个 for循环来实现Nginx服务器对事件的接收和处理 。

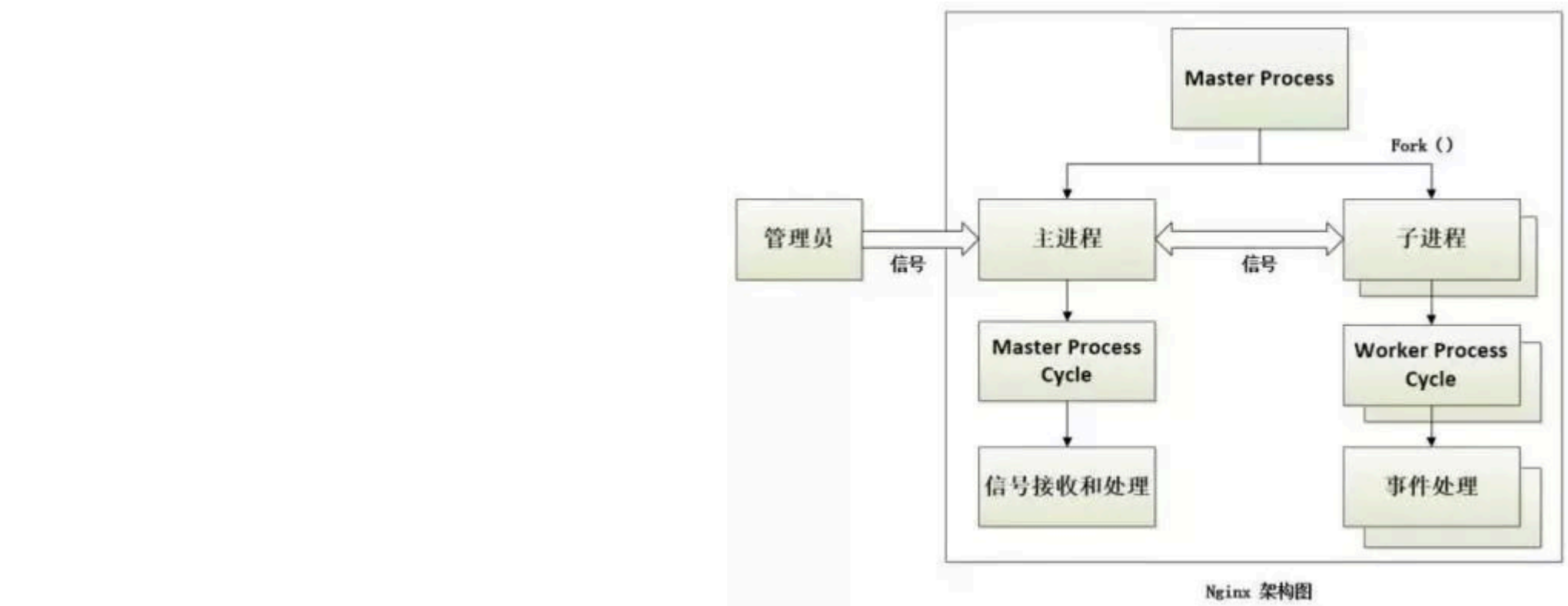


一般推荐 worker 进程数与CPU内核数一致，这样一来不存在大量的子进程生成和管理任务，避免了进程之间竞争CPU 资源和进程切换的开销。而且 Nginx 为了更好的利用 多核特性，提供了 CPU 亲缘性的绑定选项，我们可以将某一个进程绑定在某一个核上，这样就不会因为进程的切换带来 Cache 的失效。

对于每个请求，有且只有一个工作进程 对其处理。首先，每个 worker 进程都是从 master进程 fork 过来。在 master 进程里面，先建立好需要 listen 的 socket（listenfd） 之后，然后再 fork 出多个 worker 进程。

所有 worker 进程的 listenfd 会在新连接到来时变得可读 ，为保证只有一个进程处理该连接，所有 worker 进程在注册 listenfd 读事件前抢占 accept_mutex ，抢到互斥锁的那个进程注册 listenfd 读事件 ，在读事件里调用 accept 接受该连接。

当一个 worker 进程在 accept 这个连接之后，就开始读取请求、解析请求、处理请求，产生数据后，再返回给客户端 ，最后才断开连接。这样一个完整的请求就是这样的了。我们可以看到，一个请求，完全由 worker 进程来处理，而且只在一个 worker 进程中处理。



在 Nginx 服务器的运行过程中， 主进程和工作进程 需要进程交互。交互依赖于 Socket 实现的管道来实现。

聊聊：请列举Nginx服务器的最佳用途。

Nginx服务器的最佳用法是在网络上部署动态HTTP内容，使用SCGI、WSGI应用程序服务器、用于脚本的FastCGI处理程序。

它还可以作为负载均衡器。

聊聊：Nginx负载均衡实现过程

首先在http模块中配置使用upstream模块定义后台的webserver的池子，并且给池子命名， 比如命名为proxy-web，

在池子中我们可以添加多台后台webserver，其中状态检查、调度算法都是在池子中配置；

然后在server模块中定义虚拟 location 地址，但是这个虚拟location 地址 不指定自己的web目录站点，

它将使用location 匹配url然后转发到上面定义好的web池子中，

后台的webserver的池子，最后根据调度策略再转发到后台web server上。

聊聊：Nginx负载均衡配置

```
Upstream proxy_nginx {
    server 192.168.0.254 weight=1max_fails=2 fail_timeout=10s ;
    server 192.168.0.253 weight=2 max_fails=2fail_timeout=10s;
    server192.168.0.252 backup; server192.168.0.251 down;
}

server{
    listen 80;
    server_name xiaoka.com;

    location / {
        proxy_pass http:// proxy_nginx;
        proxy_set_header Host
        proxy_set_header X-Real-IP
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
```

聊聊：nginx的常用的负载均衡算法？

1、round-robin

round-robin的意思是循环轮询。

Nginx最简单的负载均衡配置如下

```
http {
    upstream app1 {
        server 10.10.10.1;
        server 10.10.10.2;
    }
    server {
        listen 80;
        location / {
            proxy_pass http://app1;
        }
    }
}
```

upstream app1用来指定一个服务器组，该组的名字是app1，包含两台服务器。在指定服务器组里面包含的服务器时以形式“server ip/domain: port”的形式指定，其中80端口可以忽略。然后在接收到请求时通过“proxy_pass <http://app1>”把对应的请求转发到组app1上。Nginx默认的负载均衡算法就是循环轮询，如上配置我们采用的就是循环轮询，其会把接收到的请求循环的分发给其包含的（当前可用的）服务器。使用如上配置时，Nginx会把第1个请求给10.10.10.1，把第2个请求给10.10.10.2，第3个请求给10.10.10.1，以此类推。

2、least-connected

least-connected算法的中文翻译是最少连接，即每次都找连接数最少的服务器来转发请求。例如Nginx负载中有两台服务器，A和B，当Nginx接收到一个请求时，A正在处理的请求数是10，B正在处理的请求数是20，则Nginx会把当前请求交给A来处理。要启用最少连接负载算法只需要在定义服务器组时加上“least_conn”，如：

```
upstream app1 {
    least_conn;
    server 10.10.10.1;
    server 10.10.10.2;
}
```

3、ip-hash

ip-hash算法会根据请求的客户端IP地址来决定当前请求应该交给谁。使用ip-hash算法时Nginx会确保来自同一客户端的请求都分发到同一服务器。要使用ip-hash算法时只需要在定义服务器组时加上“ip-hash ”指令，如：

```
upstream app1 {
    ip_hash;
```

```
server 10.10.10.1;
server 10.10.10.2;
}
```

4、weighted

weighted算法也就是权重算法，会根据每个服务的权重来分发请求，权重大的请求相对会多分发一点，权重小的会少分发一点。这通常应用于多个服务器的性能不一致时。需要使用权重算法时只需要在定义服务器组时在服务器后面指定参数weight，如：

```
upstream app1 {
    server 10.10.10.1 weight=3;
    server 10.10.10.2;
}
```

聊聊：Nginx 有哪些负载均衡策略

Nginx 默认提供的负载均衡策略：

1、轮询（默认）round_robin

每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器 down 掉，能自动剔除。

2、IP 哈希 ip_hash

每个请求按访问 ip 的 hash 结果分配，这样每个访客固定访问一个后端服务器，可以解决 session 共享的问题。

当然，实际场景下，一般不考虑使用 ip_hash 解决 session 共享。

3、最少连接 least_conn

下一个请求将被分派到活动连接数量最少的服务器

4、权重 weight

weight的值越大分配到的访问概率越高，主要用于后端每台服务器性能不均衡的情况下，达到合理的资源利用率。 还可以通过插件支持其他策略。

聊聊：ngx_http_upstream_module的作用是什么？

ngx_http_upstream_module用于定义可通过fastcgi传递、proxy传递、uwsgi传递、memcached传递和scgi传递指令来引用的服务器组。

聊聊： Nginx配置高可用性怎么配置？

当上游服务器(真实访问服务器)，一旦出现故障或者是没有及时相应的话，应该直接轮训到下一台服务器，保证服务器的高可用

Nginx配置代码：

```
server {
    listen      80;
    server_name www.lijie.com;
    location / {
        ### 指定上游服务器负载均衡服务器
        proxy_pass http://backServer;
        ###nginx与上游服务器(真实访问的服务器)超时时间 后端服务器连接的超时时间_发起握手等候响应超时时间
        proxy_connect_timeout 1s;
        ###nginx发送给上游服务器(真实访问的服务器)超时时间
        proxy_send_timeout 1s;
        ### nginx接受上游服务器(真实访问的服务器)超时时间
        proxy_read_timeout 1s;
        index   index.html index.htm;
    }
}
```

聊聊： Nginx为什么不使用多线程？

apache:

创建多个进程或线程，而每个进程或线程都会为其分配cpu和内存（线程要比进程小的多，所以worker支持比perfork高的并发），并发过大会榨干服务器资源。

Nginx:

采用单线程来异步非阻塞处理请求（管理员可以配置Nginx主进程的工作进程的数量）(epoll)，不会为每个请求分配cpu和内存资源，节省了大量资源，同时也减少了大量的CPU的上下文切换。

所以才使得Nginx支持更高的并发。

聊聊： Nginx为什么不使用多线程？

答：

Nginx:

采用单线程来异步非阻塞处理请求（管理员可以配置Nginx主进程的工作进程的数量），

不会为每个请求分配cpu和内存资源，节省了大量资源，同时也减少了大量的CPU的上下文切换，所以才使得Nginx支持更高的并发。

聊聊： Nginx主要特性

支持SSL 和TLSSNI.

Nginx它支持内核Poll模型，能经受高负载的考验,有报告表明能支持高达50,000个并发连接数。

Nginx具有很高的稳定性。

例如当前 apache一旦上到200个以上进程，web响应速度就明显非常缓慢了。而Nginx采取了分阶段资源分配技术，使得它的CPU与内存占用率非常低。nginx官方表示保持10,000个没有活动的连接，它只占2.5M内存，所以类似DOS这样的攻击对nginx来说基本上是毫无用处的。

Nginx支持热部署。

它的启动特别容易，并且几乎可以做到7*24不间断运行，即使运行数个月也不需要重新启动。对软件版本进行进行热升级。

Nginx采用master-slave模型,能够充分利用SMP的优势，且能够减少工作进程在磁盘IO的阻塞延迟。

当采用select()/poll()调用时，还可以限制每个进程的连接数。

Nginx采用master-slave模型,能够充分利用SMP的优势，且能够减少工作进程在磁盘IO的阻塞延迟。

当采用select()/poll()调用时，还可以限制每个进程的连接数。

Nginx采用了一些os提供的最新特性如对sendfile (Linux2.2+)， accept-filter

(FreeBSD4.1+)，TCP_DEFER_ACCEPT (Linux 2.4+)的支持,从而大大提高了性能。

免费开源，可以做高并发负载均衡。

聊聊： Nginx常用命令

1、启动

nginx

2、停止

nginx -s stop

或

nginx -s quit

3、重载配置

```
./sbin/nginx -s reload(平滑重启 )
```

或

```
service nginx reload
```

4、重载指定配置文件

```
nginx -c /usr/local/nginx/conf/nginx.conf
```

5、查看版本

```
nginx -v
```

6、检查配置文件是否正确

```
nginx -t
```

7、显示帮助命令

```
nginx -h
```

聊聊：Nginx IO事件模型以及连接数上限

```
events {  
    use epoll; #epoll 是多路复用 IO(I/O Multiplexing)中的一种方式,但是仅用于 linux2.6  
    #以上内核,可以大大提高 nginx 的性能  
    worker_connections 1024;#单个后台 worker process 进程的最大并发链接数  
    # multi_accept on;  
}
```

聊聊：动态资源、静态资源分离的原因

动态资源、静态资源分离是让动态网站里的动态网页根据一定规则把不变的资源和经常变的资源区分开来,

动静资源做好了拆分以后,我们就可以根据静态资源的特点将其做缓存操作,这就是网站静态化处理的核心思路

动态资源、静态资源分离简单的概括是: 动态文件与静态文件的分离

二者分离的原因

在我们的软件开发中,

有些请求是需要后台处理的(如: .jsp,.do等等),

有些请求是不需要经过后台处理的（如：css、html、jpg、js等等文件）

这些不需要经过后台处理的文件称为静态文件， 否则动态文件。

因此我们后台处理忽略静态文件。这会有人又说那我后台忽略静态文件不就完了吗

当然这是可以的， 但是这样后台的请求次数就明显增多了。

在我们对资源的响应速度有要求的时候， 我们应该使用这种动静分离的策略去解决

动、静分离将网站静态资源（HTML， JavaScript， CSS， img等文件）与后台应用分开部署， 提高用户访问静态代码的速度， 降低对后台应用访问

这里我们将静态资源放到nginx中， 动态资源转发到tomcat服务器中

聊聊： Nginx怎么做的动静分离？

只需要指定路径对应的目录。

location/可以使用正则表达式匹配。并指定对应的硬盘中的目录。如下：

```
location /image/ {
    root /usr/local/static/;
    autoindex on;
}
```

创建目录

```
mkdir /usr/local/static/image
```

进入目录

```
cd /usr/local/static/image
```

放一张照片上去

```
1.jpg
```

重启 nginx

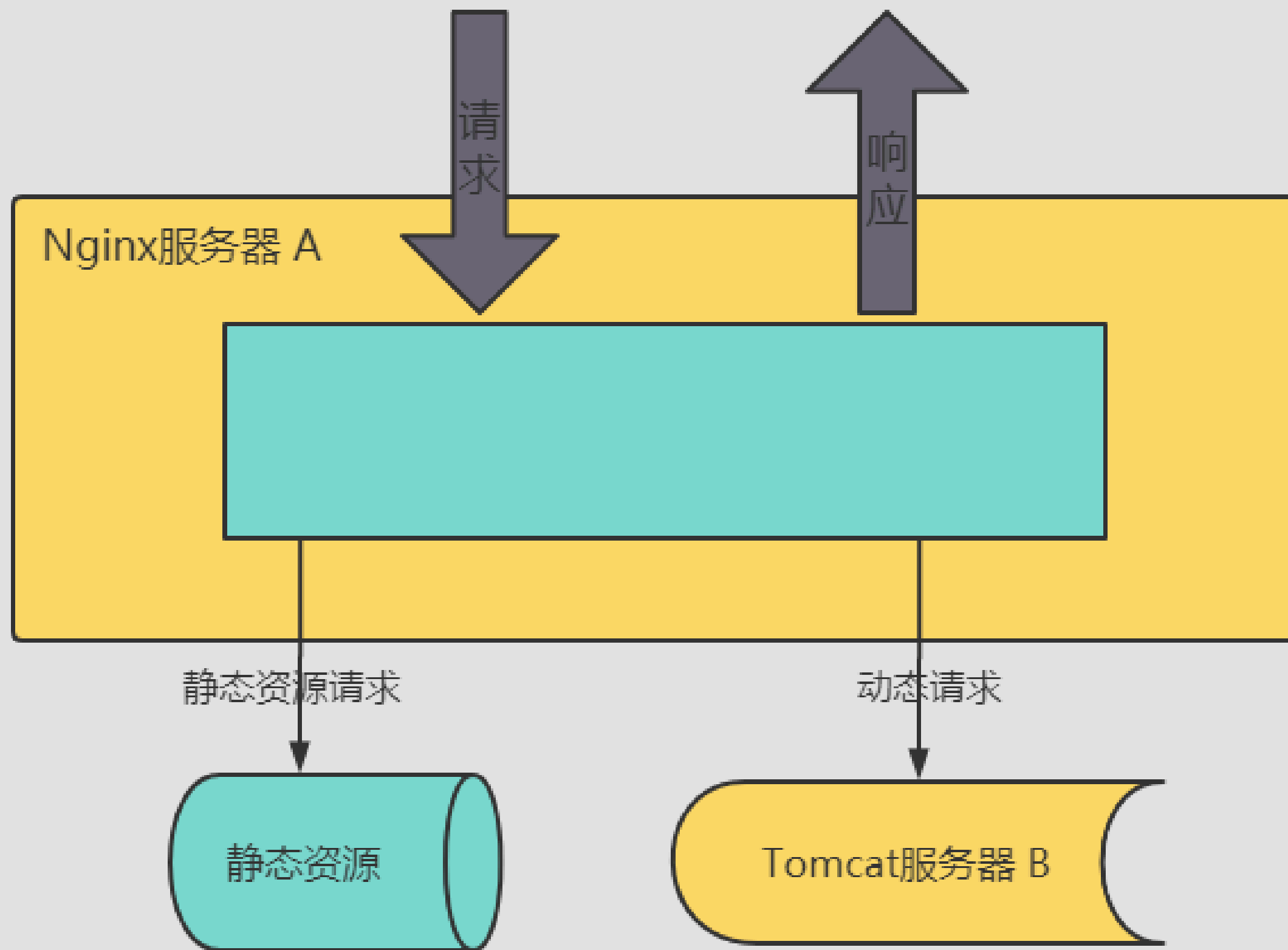
sudo nginx -s reload

打开浏览器 输入 server_name/image/1.jpg 就可以访问该静态图片了

聊聊：什么是动静分离？

动静分离是让动态网站里的动态网页根据一定规则把不变的资源和经常变的资源区分开来，动静资源做好了拆分以后，我们就可以根据静态资源的特点将其做缓存操作，这就是网站静态化处理的核心思路，实际上，何谓动？何谓静呢？拿我们 Java 来说 jsp、servlet 等就是动，因为他们离开我们的 web 服务器的支持就会无法正常工作。而 js、css 等文件就是静了。因为离开 web 服务器他一样能正常的工作。

动静分离简单的概括是：动态文件与静态文件的分离。



聊聊：Nginx动静态资源分离做过吗，为什么要这样做？

动态资源、静态资源分离，是让动态网站里的动态网页根据一定规则把不变的资源 and 经常变的资源区分开来。

比如说 js、css、html从A服务器返回。图片从B服务器返回，其他请求从Tomcat服务器C返回。

后台应用分开部署，提高用户访问静态代码的速度。而且现在还有CDN服务，不需要限制于服务器的带宽。

聊聊：Nginx如何做动静分离

Nginx 根据客户端请求的 url 来判断请求的是否是静态资源，如果请求的 url 包含 jpg、png，则由 Nginx 处理。如果请求的 url 是 .php 或者 .jsp 等等，这个时候这个请求是动态的，将转发给 tomcat 处理。

总结来说，Nginx 是通过 url 来区分请求的类型，并转发给不同的服务端。

聊聊：Nginx动静分离的好处

api 接口服务化：动静分离之后，后端应用更为服务化，只需要通过提供 api 接口即可，可以为多个功能模块甚至是多个平台的功能使用，可以有效的节省后端人力，更便于功能维护。

前后端开发并行：前后端只需要关心接口协议即可，各自的开发相互不干扰，并行开发，并行自测，可以有效的提高开发时间，也可以有些的减少联调时间

减轻后端服务器压力，提高静态资源访问速度：后端不用再将模板渲染为 html 返回给用户端，且静态服务器可以采用更为专业的技术提高静态资源的访问速度。

聊聊：Nginx 和 Apache、Tomcat 之间的不同点

Tomcat 和Nginx/Apache区别：

- 1、Nginx/Apache 是Web Server,而Apache Tomact是一个servlet container
- 2、tomcat可以对jsp进行解析，nginx和apache只是web服务器，可以简单理解为只能提供html静态文件服务。

Nginx和Apache区别：

- 1) Nginx轻量级，同样起web 服务，比apache占用更少的内存及资源 。
- 2) Nginx 抗并发，nginx 处理请求是异步非阻塞的，而apache 则是阻塞型的，在高并发下nginx 能保持低资源低消耗高性能 。
- 3) Nginx提供负载均衡，可以做做反向代理，前端服务器
- 4) Nginx多进程单线程，异步非阻塞；Apache多进程同步，阻塞。

聊聊：Nginx和Apache 之间的不同点

nginx	apache
1.nginx 是一个基于web服务器	1.Apache 是一个基于流程的服务器
2.所有请求都由一个线程来处理	2.单线程处理单个请求

nginx	apache
3.nginx避免子进程的概念	3.apache是基于子进程的
4.nginx类似于速度	4.apache类似于功率
5.nginx在内存消耗和连接方面比较好	5.apache在内存消耗和连接方面并没有提高
6.nginx在负载均衡方面表现较好	6.apache当流量达到进程的极限时，apache将拒绝新的连接
7.对于PHP来说，nginx更可取，因为他支持PHP	7.apache支持的php python Perl和其他语言，使用插件，当应用程序基于python和ruby时，它非常有用
8.nginx 不支持像ibmi 和 openvms 一样的os	8.apache支持更多的os
9.nginx 只具有核心功能	9.apache 提供了比Nginx更多的功能
10.nginx 性能和可伸缩性不依赖于硬件	10.apache 依赖于CPU和内存等硬件组件

聊聊：Nginx与Apache对比

优点

轻量级，采用 C 语言 进行编写，同样的 web 服务，会占用更少的内存及资源。

抗并发，nginx 以 epoll and kqueue 作为开发模型，处理请求是异步非阻塞的，多个连接对应一个进程，负载能力比 apache 高很多，而 apache 则是同步多进程模型，只能一个连接对应一个进程，当压力过大时，它是会被阻塞型的。

在高并发下 nginx 能保持低资源低消耗高性能 ，而 apache 在 PHP 处理慢或者前端压力很大的情况下，很容易出现进程数飙升，从而拒绝服务的现象。

设计高度模块化，编写模块相对简单。

配置简洁，正则配置让很多事情变得简单，而且改完配置能使用 -t 测试配置有没有问题，apache 配置复杂，重启的时候发现配置出错了，会很崩溃。

一般用于处理静态文件，静态处理性能比 apache 高三倍以上。

作为负载均衡服务器，支持 7 层负载均衡。

本身就是一个反向代理服务器，而且可以作为非常优秀的邮件代理服务器。

nginx 启动特别容易, 并且几乎可以做到 7*24 不间断运行，即使运行数个月也不需要重新启动，支持热部署，比如：实现不间断服务的情况下进行软件版本的升级与版本的回退。

社区活跃，各种高性能模块出品迅速。

缺点

apache 的 rewrite 比 nginx 强大，在 rewrite 频繁的情况下，用 apache。

apache 发展到现在，模块超多，基本想到的都可以找到。

apache 更为成熟，少 bug ，nginx 的 bug 相对较多。

apache 超稳定，Nginx 一个进程死掉时，会影响到多个用户的使用，稳定性差。

apache 对 PHP 支持比较简单，nginx 需要配合其他后端用。

apache 在处理动态请求有优势，nginx 在这方面是鸡肋，一般动态请求要 apache 去做，nginx 适合静态和反向。

apache 仍然是目前的主流，拥有丰富的特性，成熟的技术和开发社区。

聊聊：Nginx与Apache选择

Apache

- apache 的 rewrite 比 **nginx** 强大，在 rewrite 频繁的情况下，用 apache
- apache 发展到现在，模块超多，基本想到的都可以找到
- apache 更为成熟，少 bug ， nginx 的 bug 相对较多
- apache 超稳定
- apache 对 **PHP** 支持比较简单，nginx 需要配合其他后端用
- apache 在处理动态请求有优势，nginx 在这方面是鸡肋，一般动态请求要 apache 去做，nginx 适合静态和反向。
- apache 仍然是目前的主流，拥有丰富的特性，成熟的技术和开发社区

Nginx

- 轻量级，采用 C 语言 进行编写，同样的 web 服务，会占用更少的内存及资源
- 抗并发，nginx 以 epoll and kqueue 作为开发模型，处理请求是异步非阻塞的，负载能力比 apache 高很多，而 apache 则是阻塞型的。在高并发下 nginx 能保持低资源低消耗高性能，而 apache 在 PHP 处理慢或者前端压力很大的情况下，很容易出现进程数飙升，从而拒绝服务的现象。
- nginx 处理静态文件好，静态处理性能比 apache 高三倍以上
- nginx 的设计高度模块化，编写模块相对简单
- nginx 配置简洁，正则配置让很多事情变得简单，而且改完配置能使用 -t 测试配置有没有问题，apache 配置复杂，重启的时候发现配置出错了，会很崩溃
- nginx 作为负载均衡服务器，支持 7 层负载均衡
- nginx 本身就是一个反向代理服务器，而且可以作为非常优秀的邮件代理服务器
- 启动特别容易, 并且几乎可以做到 7*24 不间断运行，即使运行数个月也不需要重新启动，还能够不间断服务的情况下进行软件版本的升级
- 社区活跃，各种高性能模块出品迅速

聊聊：Nginx如何处理HTTP请求。

Nginx使用反应器模式。

主事件循环等待操作系统发出准备事件的信号，这样数据就可以从套接字读取，在该实例中读取到缓冲区并进行处理。

单个线程可以提供数万个并发连接。

聊聊：Nginx是如何处理一个请求的呢？

首先，nginx在启动时，会解析配置文件，得到需要监听的端口与ip地址，

然后在nginx的master进程里面，先初始化好这个监控的socket，再进行listen

然后再fork出多个子进程出来, 子进程会竞争accept新的连接。

此时，客户端就可以向nginx发起连接了。

当客户端与nginx进行三次握手，与nginx建立好一个连接后，

此时，某一个子进程会accept成功，然后创建nginx对连接的封装，即ngx_connection_t结构体，

接着，根据事件调用相应的事件处理模块，如http模块与客户端进行数据的交换，

最后，nginx或客户端来主动关掉连接，到此，一个连接就寿终正寝了

聊聊：Nginx处理HTTP请求过程的 11 个阶段？

Nginx 处理 HTTP 请求的过程大概可以分为 11 个阶段，如下：

Read Request Headers：解析请求头。

Identify Configuration Block：识别由哪一个 location 进行处理，匹配 URL。

Apply Rate Limits：判断是否限速。例如可能这个请求并发的连接数太多超过了限制，或者 QPS 太高。

Perform Authentication：连接控制，验证请求。例如可能根据 Referrer 头部做一些防盗链的设置，或者验证用户的权限。

Generate Content：生成返回给用户的响应。为了生成这个响应，做反向代理的时候可能会和上游服务（Upstream Services）进行通信，然后这个过程中还可能会有些子请求或者重定向，那么还会走一下这个过程（Internal redirects and subrequests）。

Response Filters：过滤返回给用户的响应。比如压缩响应，或者对图片进行处理。

Log：记录日志。

以上这七个步骤从整体上介绍了一下处理流程，下面还会再说一下实际的处理过程。

聊聊： Nginx处理HTTP请求的11个阶段

下面介绍一下详细的 11 个阶段，每个阶段都可能对应着一个甚至多个 HTTP 模块，通过这样一个模块对比，我们也能够很好的理解这些模块具体是怎样发挥作用的。

POST_READ	realip
SERVER_REWRITE	rewrite
FIND_CONFIG	
REWRITE	rewrite
POST_REWRITE	
PREACCESS	limt_conn, limit_req
ACCESS	auth_basic, access, auth_request
POST_ACCESS	
PRECONTENT	try_files
CONTENT	index, autoindex, concat

LOG

access_log

POST_READ：在 read 完请求的头部之后，在没有对头部做任何处理之前，想要获取到一些原始的值，就应该在这个阶段进行处理。这里面会涉及到一个 realip 模块。

SERVER_REWRITE：和下面的 REWRITE 阶段一样，都只有一个模块叫 rewrite 模块，一般没有第三方模块会处理这个阶段。

FIND_CONFIG：做 location 的匹配，暂时没有模块会用到。

REWRITE：对 URL 做一些处理。

POST_WRITE：处于 REWRITE 之后，也是暂时没有模块会在这个阶段出现。

接下来是确认用户访问权限的三个模块：

PREACCESS：是在 ACCESS 之前要做一些工作，例如并发连接和 QPS 需要进行限制，涉及到两个模块：limt_conn 和 limit_req

ACCESS：核心要解决的是用户能不能访问的问题，例如 auth_basic 是用户名和密码，access 是用户访问 IP，auth_request 根据第三方服务返回是否可以访问。

POST_ACCESS：是在 ACCESS 之后会做一些事情，同样暂时没有模块会用到。

最后的三个阶段处理响应和日志：

PRECONTENT：在处理 CONTENT 之前会做一些事情，例如会把子请求发送给第三方的服务去处理，try_files 模块也是在这个阶段中。

CONTENT：这个阶段涉及到的模块就非常多了，例如 index, autoindex, concat 等都是在这个阶段生效的。

LOG：记录日志 access_log 模块。

以上的这些阶段都是严格按照顺序进行处理的，当然，每个阶段中各个 HTTP 模块的处理顺序也很重要，如果某个模块不把请求向下传递，后面的模块是接收不到请求的。

而且每个阶段中的模块也不一定所有都要执行一遍，下面就接着讲一下各个阶段模块之间的请求顺序。

聊聊：Nginx处理HTTP请求的11个阶段的顺序处理

如下图所示，每一个模块处理之间是有序的，那么这个顺序怎么才能得到呢？其实非常简单，在源码 ngx_module.c 中，有一个数组 ngx_module_name，其中包含了在编译 Nginx 的时候的 with 指令所包含的所有模块，它们之间的顺序非常关键，在数组中顺序是相反的。

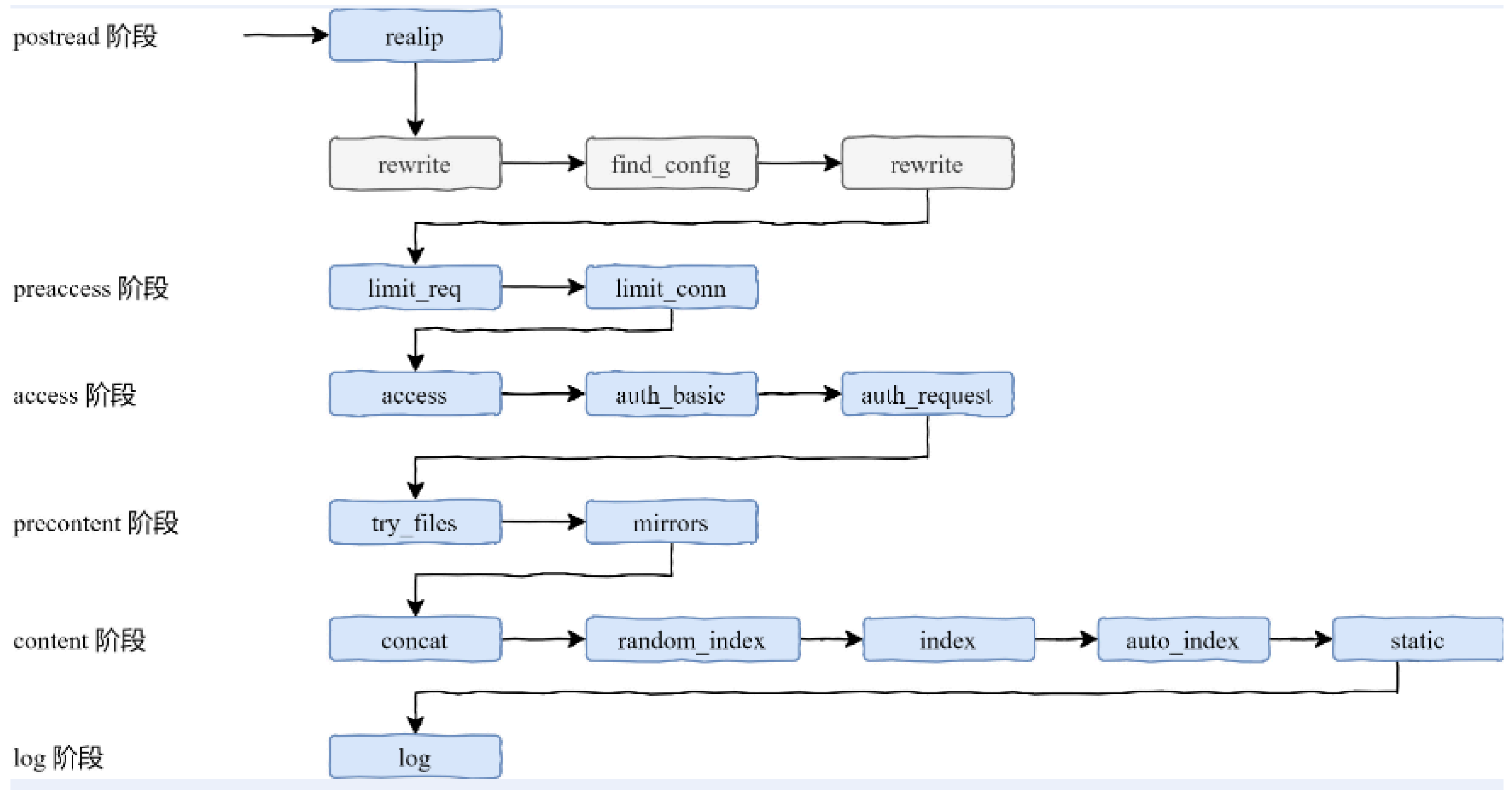
```
char *ngx_module_names[] = {
    ...
    "ngx_http_static_module",
    "ngx_http_autoindex_module",
    "ngx_http_index_module",
```

```
    "ngx_http_random_index_module",
    "ngx_http_mirror_module",
    "ngx_http_try_files_module",
    "ngx_http_auth_request_module",
    "ngx_http_auth_basic_module",
    "ngx_http_access_module",
    "ngx_http_limit_conn_module",
    "ngx_http_limit_req_module",
    "ngx_http_realip_module",
    "ngx_http_referer_module",
    "ngx_http_rewrite_module",
    "ngx_http_concat_module",
    ... ..
}
```

灰色部分的模块是 Nginx 的框架部分去执行处理的，第三方模块没有机会在这里得到处理。

在依次向下执行的过程中，也可能不按照这样的顺序。例如，在 access 阶段中，有一个指令叫 satisfy，它可以指示当有一个满足的时候就直接跳到下一个阶段进行处理，例如当 access 满足了，就直接跳到 try_files 模块进行处理，而不会再执行 auth_basic、auth_request 模块。

在 content 阶段中，当 index 模块执行了，就不会再执行 auto_index 模块，而是直接跳到 log 模块。整个 11 个阶段所涉及到的模块和先后顺序如下图所示：



聊聊：Nginx配置文件nginx.conf有哪些属性模块？

```
worker_processes 1;
events {
    worker_connections 1024;
}
```

```
# worker进程的数量
# 事件区块开始
# 每个worker进程支持的最大连接数
# 事件区块结束
```

```
http {  
    include      mime.types;  
    default_type application/octet-stream;  
    sendfile      on;  
    keepalive_timeout  65;  
    server {  
        listen      80;  
        server_name  localhost;  
        location / {  
            root    html;  
            index   index.html index.htm;  
        }  
        error_page   500 502 503 504    /50x.html;  
        location = /50x.html {  
            root    html;  
        }  
    }  
    .....  
}
```

```
# HTTP区块开始  
# Nginx支持的媒体类型库文件  
# 默认的媒体类型  
# 开启高效传输模式  
# 连接超时  
# 第一个Server区块开始，表示一个独立的虚拟主机站点  
# 提供服务的端口，默认80  
# 提供服务的域名主机名  
# 第一个location区块开始  
# 站点的根目录，相当于Nginx的安装目录  
# 默认的首页文件，多个用空格分开  
# 第一个location区块结果  
# 出现对应的http状态码时，使用50x.html回应客户  
# location区块开始，访问50x.html  
# 指定对应的站点目录为html
```

聊聊：在Nginx中，如何使用未定义的服务器名称来阻止处理请求？

只需将请求删除的服务器, 可以定义为：

```
Server {  
    listen 80;  
    server_name " " ;  
    return 444;  
}
```

这里，服务器名被保留为一个空字符串，它将在没有“主机”头字段的情况下匹配请求，而一个特殊的Nginx的非标准代码444被返回，从而终止连接。

聊聊：Nginx的进程模型

master-worker模式

在master-worker模式下，有一个master进程和至少一个的worker进程。

单进程模式。
单进程模式只有一个进程。

聊聊： Nginx服务器上的Master和Worker进程分别是什么？

Master进程： 读取及评估配置和维持

Worker进程： 处理请求

聊聊： Nginx惊群

惊群效应（thundering herd）是指多进程（多线程）在同时阻塞等待同一个事件的时候（休眠状态），

如果等待的这个事件发生，那么他就会唤醒等待的所有进程（或者线程），但是最终却只能有一个进程（线程）获得这个时间的“控制权”，对该事件进行处理，

而其他进程（线程）获取“控制权”失败，只能重新进入休眠状态，这种现象和性能浪费就叫做惊群效应。

聊聊： 惊群效应消耗了什么

Linux 内核对用户进程（线程）频繁地做无效的调度、上下文切换等使系统性能大打折扣。上下文切换（context switch）过高会导致 CPU 像个搬运工，频繁地在寄存器和运行队列之间奔波，更多的时间花在了进程（线程）切换，而不是在真正工作的进程（线程）上面。

直接的消耗包括 CPU 寄存器要保存和加载（例如程序计数器）、系统调度器的代码需要执行。间接的消耗在于多核 cache 之间的共享数据。为了确保只有一个进程（线程）得到资源，需要对资源操作进行加锁保护，加大了系统的开销。目前一些常见的服务器软件有的是通过锁机制解决的，比如 Nginx（它的锁机制是默认开启的，可以关闭）；还有些认为惊群对系统性能影响不大，没有去处理，比如 Lighttpd。

聊聊： Nginx惊群效应? 以及解决方案

对于 Nginx 的惊群问题，我们首先需要理解的是，在 Nginx 启动过程中，master 进程会监听配置文件中指定的各个端口，然后 master 进程就会调用 fork() 方法创建各个子进程，根据进程的工作原理，子进程是会继承父进程的全部内存数据以及监听的端口的，也就是说 worker 进程在启动之后也是会监听各个端口的。

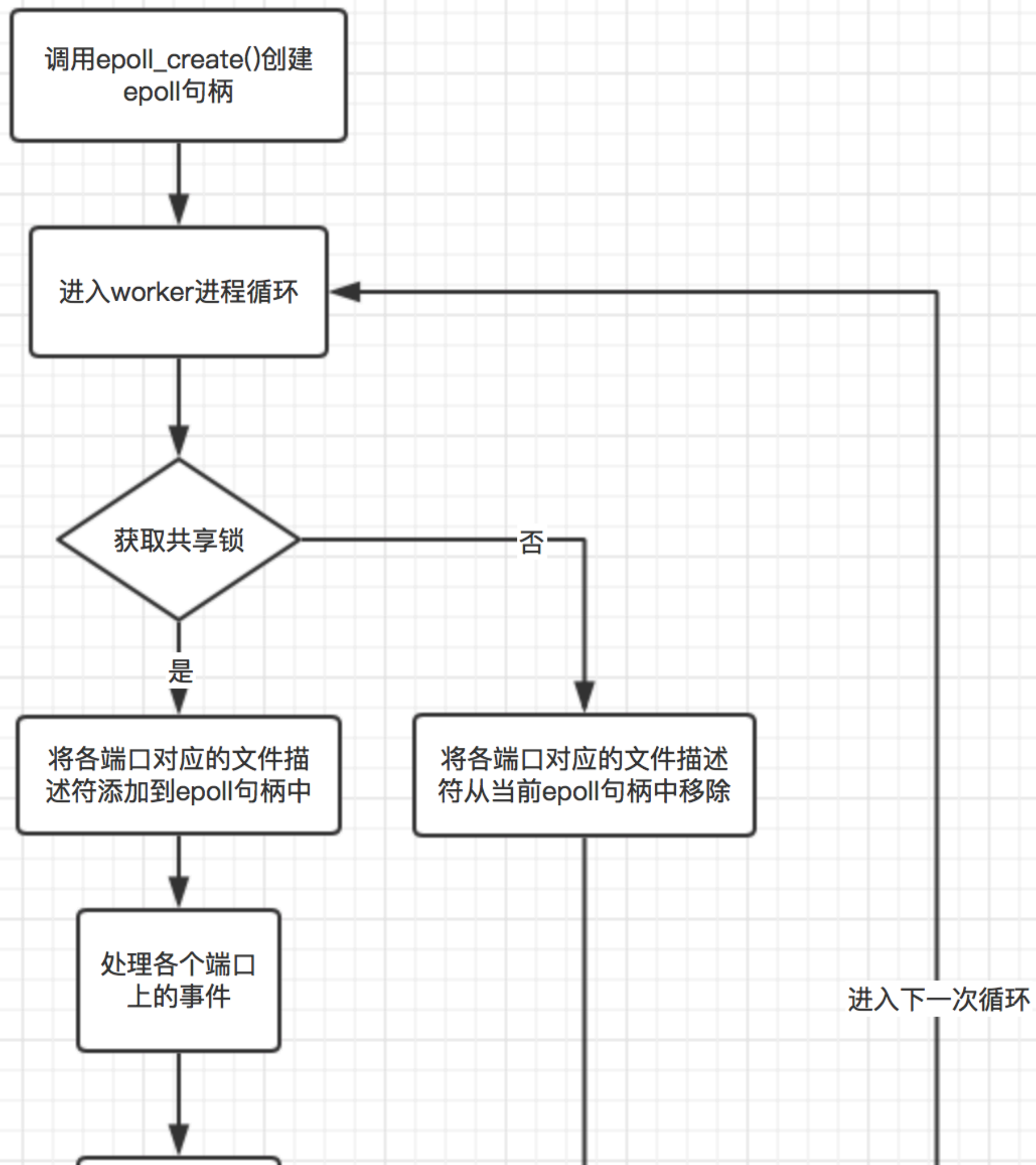
关于惊群，指的就是当客户端有新建连接的请求到来时，就会触发各个 worker 进程的连接建立事件，但是只有一个 worker 进程能够正常处理该事件，而其他的 worker 进程会发现事件已经失效，从而重新循环进入等待状态。这种由于一个事件而“惊”起了所有 worker 进程的现象就是惊群问题。很明显，如果所有的 worker 进程都被触发了，那么这将消耗大量的资源。

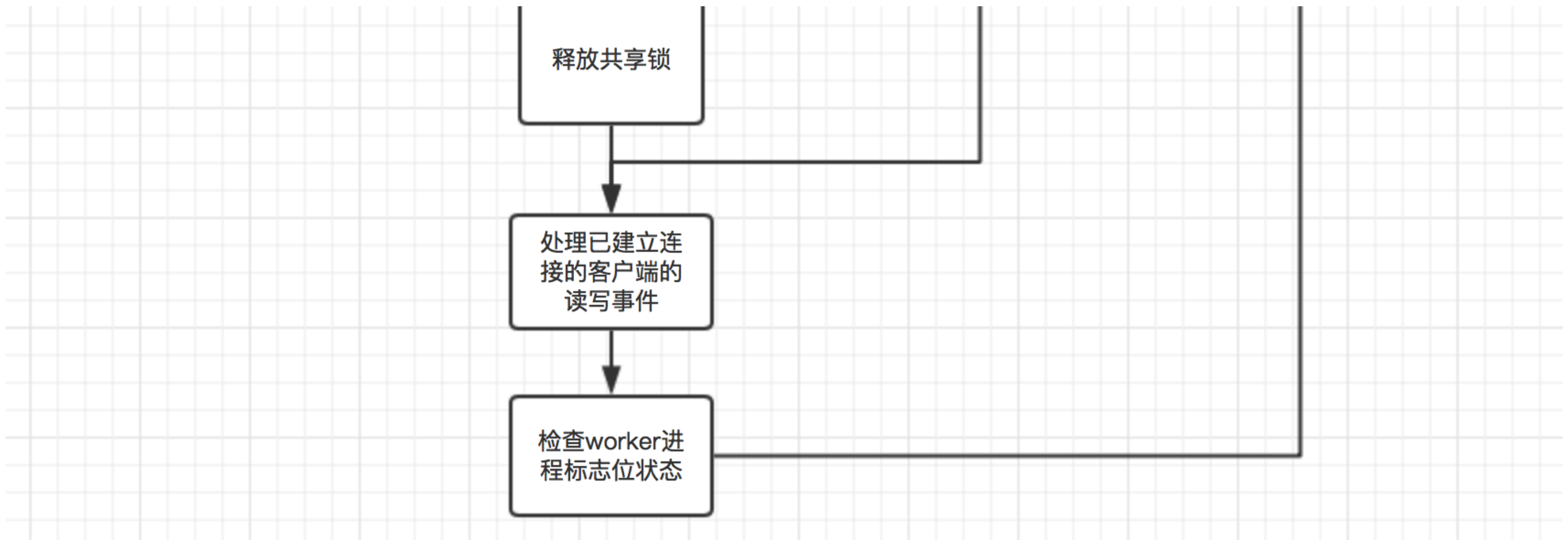
解决方案

在 Nginx 中，每个 worker 进程被创建的时候，都会调用 ngx_worker_process_init() 方法初始化当前 worker 进程，这个过程中有一个非常重要的步骤，即每个 worker 进程都会调用 epoll_create() 方法为自己创建一个独有的 epoll 句柄。

对于每一个需要监听的端口，都有一个文件描述符与之对应，而 worker 进程只有将该文件描述符通过 epoll_ctl() 方法添加到当前进程的 epoll 句柄中，并且监听 accept 事件，此时才会被客户端的连接建立事件触发，从而处理该事件。从这里也可以看出，worker 进程如果没有将所需要监听的端口对应的文件描述符添加到该进程的 epoll 句柄中，那么其是无法被触发对应的事件的。

基于这个原理，nginx 就使用了一个共享锁来控制当前进程是否有权限将需要监听的端口添加到当前进程的 epoll 句柄中，也就是说，只有获取锁的进程才会监听目标端口。通过这种方式，就保证了每次事件发生时，只有一个 worker 进程会被触发。如下图所示为 worker 进程工作循环的一个示意图：





这里关于图中的流程，需要说明的一点是，每个 worker 进程在进入循环之后就会尝试获取共享锁，如果没有获取到，就会将所监听的端口的文件描述符从当前进程的 epoll 句柄中移除（即使并不存在也会移除），这么做的主要目的是防止丢失客户端连接事件，即使这可能造成少量的惊群问题，但是并不严重。

试想一下，如果按照理论，在当前进程释放锁的时候就将监听的端口的文件描述符从 epoll 句柄中移除，那么在下一个 worker 进程获取锁之前，这段时间各个端口对应的文件描述符是没有任何 epoll 句柄进行监听的，此时就会造成事件的丢失。如果反过来，按照图中的在获取锁失败的时候才移除监听的文件描述符，由于获取锁失败，则说明当前一定有一个进程已经监听了这些文件描述符，因而此时移除是安全的。

但是这样会造成的一个问题是，按照上图，当前进程在一个循环执行完毕的时候，会释放锁，然后处理其他的事件，注意这个过程中其是没有释放所监听的文件描述符的。此时，如果另一个进程获取到了锁，并且监听了文件描述符，那么这个时候就有两个进程监听了文件描述符，因而此时如果客户端发生连接建立事件，那么就会触发两个 worker 进程。这个问题是可以容忍的，主要原因有两点：

这个时候发生的惊群现象只触发了更少的 worker 进程，比起每次都惊起所有的 worker 进程要好很多；
会发生这种惊群问题的主要原因是，当前进程释放了锁，但是没有释放所监听的文件描述符，但是 worker 进程在释放锁之后主要是处理客户端连接的读写事件和检查标志位，这个过程是非常短的，在处理完之后，其就会尝试获取锁，这个时候就会释放所监听的文件描述符了，而相较而言，获取锁的 worker 进程在等待处理客户端的连接建立事件的事件就更长了，因而会发生惊群问题的概率还是比较小的。

源码讲解

worker 进程初始事件的方法主要是在 ngx_process_events_and_timers() 方法中进行的，下面我们就来看看该方法是如何处理整个流程的，如下是该方法的源码：

```
void ngx_process_events_and_timers(ngx_cycle_t *cycle) {
    ngx_uint_t flags;
    ngx_msec_t timer, delta;
    if (ngx_trylock_accept_mutex(cycle) == NGX_ERROR) {
        return;
    }
    // 这里开始处理事件，对于kqueue模型，其指向的是ngx_kqueue_process_events()方法，
    // 而对于epoll模型，其指向的是ngx_epoll_process_events()方法
    // 这个方法的主要作用是，在对应的事件模型中获取事件列表，然后将事件添加到ngx_posted_accept_events
    // 队列或者ngx_posted_events队列中
    (void) ngx_process_events(cycle, timer, flags);
    // 这里开始处理accept事件，将其交由ngx_event_accept.c的ngx_event_accept()方法处理；
    ngx_event_process_posted(cycle, &ngx_posted_accept_events);
    // 开始释放锁
    if (ngx_accept_mutex_held) {
        ngx_shmtx_unlock(&ngx_accept_mutex);
    }
    // 如果不需要在事件队列中进行处理，则直接处理该事件
    // 对于事件的处理，如果是accept事件，则将其交由ngx_event_accept.c的ngx_event_accept()方法处理；
    // 如果是读事件，则将其交由ngx_http_request.c的ngx_http_wait_request_handler()方法处理；
    // 对于处理完成的事件，最后会交由ngx_http_request.c的ngx_http_keepalive_handler()方法处理。
    // 这里开始处理除accept事件外的其他事件
    ngx_event_process_posted(cycle, &ngx_posted_events);
}
```

上面的代码中，我们省略了大部分的检查工作，只留下了骨架代码。首先，worker 进程会调用 ngx_trylock_accept_mutex() 方法获取锁，这其中如果获取到了锁就会监听各个端口对应的文件描述符。然后会调用 ngx_process_events() 方法处理 epoll 句柄中监听到的事件。接着会释放共享锁，最后就是处理已建立连接的客户端的读写事件。下面我们来看一下 ngx_trylock_accept_mutex() 方法是如何获取共享锁的：

```
ngx_int_t ngx_trylock_accept_mutex(ngx_cycle_t *cycle) {
    // 尝试使用CAS算法获取共享锁
    if (ngx_shmtx_trylock(&ngx_accept_mutex)) {
```

```
// ngx_accept_mutex_held为1表示当前进程已经获取到了锁
if (ngx_accept_mutex_held && ngx_accept_events == 0) {
    return NGX_OK;
}
// 这里主要是将当前连接的文件描述符注册到对应事件的队列中，比如kqueue模型的change_list数组
// nginx在启用各个worker进程的时候，默认情况下，worker进程是会继承master进程所监听的socket句柄的，
// 这就导致一个问题，就是当某个端口有客户端事件时，就会把监听该端口的进程都给唤醒，
// 但是只有一个worker进程能够成功处理该事件，而其他的进程被唤醒之后发现事件已经过期，
// 因而会继续进入等待状态，这种现象称为"惊群"现象。
// nginx解决惊群现象的方式一方面是通过这里的共享锁的方式，即只有获取到锁的worker进程才能处理
// 客户端事件，但实际上，worker进程是通过在获取锁的过程中，为当前worker进程重新添加各个端口的监听事件，
// 而其他worker进程则不会监听。也就是说同一时间只有一个worker进程会监听各个端口，
// 这样就避免了"惊群"问题。
// 这里的ngx_enable_accept_events()方法就是为当前进程重新添加各个端口的监听事件的。
if (ngx_enable_accept_events(cycle) == NGX_ERROR) {
    ngx_shmtx_unlock(&ngx_accept_mutex);
    return NGX_ERROR;
}
// 标志当前已经成功获取到了锁
ngx_accept_events = 0;
ngx_accept_mutex_held = 1;
return NGX_OK;
}
// 前面获取锁失败了，因而这里需要重置ngx_accept_mutex_held的状态，并且将当前连接的事件给清除掉
if (ngx_accept_mutex_held) {
    // 如果当前进程的ngx_accept_mutex_held为1，则将其重置为0，并且将当前进程在各个端口上的监听
    // 事件给删除掉
    if (ngx_disable_accept_events(cycle, 0) == NGX_ERROR) {
        return NGX_ERROR;
    }
    ngx_accept_mutex_held = 0;
}
```

```
    return NGX_OK;
}
```

上面的代码中，本质上主要做了三件事：

通过 ngx_shmtx_trylock() 方法尝试使用CAS方法获取共享锁；

获取锁之后则调用 ngx_enable_accept_events() 方法监听目标端口对应的文件描述符；

如果没有获取到锁，则调用 ngx_disable_accept_events() 方法释放所监听的文件描述符；

总结

惊群现象指所有的工作进程都在等待一个 socket，当 socket 客户端连接时，所有工作线程都被唤醒，但最终有且仅有一个工作线程去处理该连接，其他进程又要进入睡眠状态。

Nginx 通过控制争抢处理 socket 的进程数量和抢占 ngx_accept_mutex 锁解决惊群现象。只有一个 ngx_accept_mutex 锁，谁拿到锁，谁处理该 socket 的请求。

聊聊：Nginx IO模型

Nginx 支持多种并发模型，并发模型的具体实现根据系统平台而有所不同。

在支持多种并发模型的平台上，Nginx 自动选择最高效的模型。

但我们也可以使用 use 指令在配置文件中显式地定义某个并发模型。

聊聊：Nginx IO模型

Nginx 支持多种并发模型，并发模型的具体实现根据系统平台而有所不同。在支持多种并发模型的平台上，Nginx 自动选择最高效的模型。但我们也可以使用 use 指令在配置文件中显式地定义某个并发模型。

Nginx中支持的并发模型

select

IO 多路复用、标准并发模型。在编译 Nginx 时，如果所使用的系统平台没有更高效并发模型，select 模块将被自动编译。configure 脚本的选项：-with-select_module 和 --without-select_module 可被用来强制性地开启或禁止 select 模块的编译。

poll

IO 多路复用、标准并发模型。与 select 类似，在编译 Nginx 时，如果所使用的系统平台没有更高效的并发模型，poll 模块将被自动编译。configure 脚本的选项：-with-poll_module 和 --without-poll_module 可用于强制性地开启或禁止 poll 模块的编译。

epoll

IO 多路复用、高效并发模型，可在 Linux 2.6+ 及以上内核可以使用。

kqueue

IO 多路复用、高效并发模型，可在 FreeBSD 4.1+, OpenBSD 2.9+, NetBSD 2.0, and Mac OS X 平台中使用。

/dev/poll

高效并发模型，可在 Solaris 7 11/99+, HP/UX 11.22+ (eventport), IRIX 6.5.15+, and Tru64 UNIX 5.1A+ 平台使用。

eventport

高效并发模型，可用于 Solaris 10 平台，PS：由于一些已知的问题，建议使用/dev/poll替代。

**聊聊：为什么 epoll 快，比较一下 Apache 常用的 select 和 Nginx 常用的 epoll：
select**

最大并发数限制，因为一个进程所打开的 FD（文件描述符）是有限制的，由 FD_SETSIZE 设置，默认值是 1024/2048，因此 Select 模型的最大并发数就被相应限制了。自己改改这个 FD_SETSIZE？想法虽好，可是先看看下面吧。

效率问题，select 每次调用都会线性扫描全部的 FD 集合，这样效率就会呈现线性下降，把 FD_SETSIZE 改大的后果就是，大家都慢慢来，什么？都超时了。

内核/用户空间，内存拷贝问题，如何让内核把 FD 消息通知给用户空间呢？在这个问题上 select 采取了内存拷贝方法，在 FD 非常多的时候，非常的耗费时间。

总结为：

连接数受限

查找配对速度慢

数据由内核拷贝到用户态消耗时间。

epoll

Epoll 没有最大并发连接的限制，上限是最大可以打开文件的数目，这个数字一般远大于 2048, 一般来说这个数目和系统内存关系很大，具体数目可以 cat /proc/sys/fs/file-max 查看。

效率提升，Epoll 最大的优点就在于它只管你 “活跃” 的连接 ， 而跟连接总数无关，因此在实际的网络环境中，Epoll 的效率就会远远高于 select 和 poll。

内存共享，Epoll 在这点上使用了 “共享内存”，这个内存拷贝也省略了。

聊聊： 你如何通过不同于80的端口开启Nginx？

为了通过一个不同的端口开启Nginx，你必须进入/etc/Nginx/sites-enabled/，如果这是默认文件，那么你必须打开名为“default”的文件。

编辑文件，并放置在你想要的端口：

```
server {  
    listen 81;  
}
```

聊聊： location的作用是什么？

location指令的作用是根据用户请求的URI来执行不同的应用， 也就是根据用户请求的网站URL进行匹配， 匹配成功即进行相关的操作。

聊聊： location的语法能说出来吗？

location主要用于URI的匹配。

什么是 uri：

URI：Uniform Resource Identifier，统一资源标识符；如下图的数据.html；

URN：Uniform Resource Name，统一资源名称，如下图的ste.org/img.png，比URI多个域名；

URL：Uniform Resource Locator，统一资源定位符，如下面，URL包含了http协议、端口、域名、文件名。

URI的匹配， 示例如下：

#优先级1,精确匹配，根路径

```
location =/ {  
    return 400;  
}
```

#优先级2,以某个字符串开头,以av开头的，优先匹配这里，区分大小写

```
location ^~ /av {  
    root /data/av/;  
}
```


#优先级3，区分大小写的正则匹配，匹配/media*****路径

```
location ~ /media {  
    alias /data/static/;  
}
```

#优先级4 ，不区分大小写的正则匹配，所有的****.jpg|gif|png 都走这里

```
location ~* .*\. (jpg|gif|png|js|css)$ {  
    root /data/av/;  
}
```

#优先7，通用匹配

```
location / {  
    return 403;  
}
```

聊聊：Nginx如何定义错误提示页面

```
#error_page 500 502 503 504 /50x.html;  
location = /50x.html {  
    root /root;  
}
```

聊聊：是否有可能将Nginx的错误替换为502错误、503？

502 =错误网关

503 =服务器超载

有可能，但是您可以确保fastcgi_intercept_errors被设置为ON，并使用错误页面指令。

```
Location / {  
    fastcgi_pass 127.0.01:9001;  
    fastcgi_intercept_errors on;  
    error_page 502 =503/error_page.html;
```

聊聊：nginx中500、502、503、504 有什么区别？

500：

Internal Server Error 内部服务错误，比如脚本错误，编程语言语法错误。

502：

Bad Gateway错误，网关错误。比如服务器当前连接太多，响应太慢，页面素材太多、带宽慢。

503：

Service Temporarily Unavailable，服务不可用，web服务器不能处理HTTP请求，可能是临时超载或者是服务器进行停机维护。

504：

Gateway timeout 网关超时，程序执行时间过长导致响应超时，例如程序需要执行20秒，而nginx最大响应等待时间为10秒，这样就会出现超时。

聊聊： Nginx如何精准匹配路径

```
location = /get {  
    #规则 A  
}
```

聊聊： Nginx路径匹配优先级

多个location 配置的情况下匹配顺序为

- 首先匹配 =
- 其次匹配 ^~
- 再其次是按文件中顺序的正则匹配
- 最后是交给 / 通用匹配。

当有匹配成功时候，停止匹配，按当前匹配规则处理请求。

聊聊： 如何将请求转发给后端应用服务器

```
location = / {  
    proxy_pass http://tomcat:8080/index  
}
```

聊聊： 如何根据文件类型设置过期时间

```
location ~* \.(js|css|jpg|jpeg|gif|png|swf)$ {
    if (-f $request_filename) {
        expires 1h;
        break;
    }
}
```

聊聊： 如何禁止访问某个目录

```
location ^~/path/ {
    deny all;
}
```

聊聊： 在Nginx中， 解释如何在URL中保留双斜线？

要在URL中保留双斜线， 就必须使用merge_slashes_off;

语法:merge_slashes [on/off]

默认值: merge_slashes on

环境: http, server

聊聊： Nginx rewrite全局变量

Nginx rewrite 常用的全局变量如下：

变量	说明
\$args	存放了请求 url 中的请求指令。比如 http://www.myweb.name/server/source?arg1=value1&arg2=value2 中的arg1=value1&arg2=value2
\$content_length	存放请求头中的 Content-length 字段
\$content_type	存放了请求头中的 Content-type 字段
\$document_root	存放了针对当前请求的根路径
\$document_uri	请求中的 uri， 不包含请求指令 ， 比如比如 http://www.myweb.name/server/source?arg1=value1&arg2=value2 中的 /server/source
\$host	存放了请求 url 中的主机字段， 比如 http://www.myweb.name/server/source?arg1=value1&arg2=value2 中的 www.myweb.name。如果请求中的主机部分字段不可用或者为空， 则存放 nginx 配置中该 server 块中 server_name 指令的配置值

变量	说明
\$http_user_agent	存放客户端的代理
\$http_cookie	cookie
\$limit_rate	nginx 配置中 limit_rate 指令的配置值
\$remote_addr	客户端的地址
\$remote_port	客户端与服务器端建立连接的端口号
\$remote_user	变量中存放了客户端的用户名
\$request_body_file	存放了发给后端服务器的本地文件资源的名称
\$request_method	存放了客户端的请求方式，如 get,post 等
\$request_filename	存放当前请求的资源文件的路径名
\$requet_uri	当前请求的 uri,并且带有指令
\$query_string	\$args含义相同
\$scheme	客户端请求使用的协议，如 http, https, ftp 等
\$server_protocol	客户端请求协议的版本，如 "HTTP/1.0", "HTTP/1.1"
\$server_addr	服务器的地址
\$server_name	客户端请求到达的服务器的名称
\$server_port	客户端请求到达的服务器的端口号
\$uri	同 \$document_uri

请解释ngx_http_upstream_module的作用是什么？

ngx_http_upstream_module用于定义可通过fastcgi传递、proxy传递、uwsgi传递、memcached传递和scgi传递指令来引用的服务器组。

聊聊： stub_status和sub_filter指令的作用是什么？

Stub_status指令： 该指令用于了解Nginx当前状态的当前状态， 如当前的活动连接， 接受和处理当前读/写/等待连接的总数

Sub_filter指令： 它用于搜索和替换响应中的内容， 并快速修复陈旧的数据

聊聊：Nginx 压缩了解吗，如何开启压缩？

开启nginx gzip压缩后，图片、css、js等静态资源的大小会减小，可节省带宽，提高传输效率，但是会消耗CPU资源。

```
gzip on;
#开启gzip压缩功能
gzip_min_length 1k;
#设置允许压缩的页面最小字节数，页面字节数从header头的content-length中获取。默认值是0,不管页面多大都进行压缩。建议设置成大于1k。如果小于1k可能会越压越大。
gzip_buffers 4 16k;
#压缩缓冲区大小。表示申请4个单位为16k的内容作为压缩结果流缓存，默认值是申请与原始数据大小相同的内存空间来存储gzip压缩结果。
gzip_http_version 1.0;
#压缩版本（默认1.1，前端为squid2.5时使用1.0）用于设置识别http协议版本，默认是1.1,目前大部分浏览器已经支持gzip解压，使用默认即可。
gzip_comp_level 2;
#压缩比率。用来指定gzip压缩比，1压缩比量小，处理速度快；9压缩比量大，传输速度快，但处理最慢，也必将消耗cpu资源。
gzip_types text/plain application/x-javascript text/css application/xml;
#用来指定压缩的类型，“text/html”类型总是会被压缩。
gzip_vary on;
#vary header支持。该选项可以让前端的缓存服务器缓存经过gzip压缩的页面，例如用squid缓存经过nginx压缩的数据。
```

要注意：需要和不需要压缩的对象

- (1) 大于1k的纯文本文件html,js,css,xml,html.
- (2) 图片，视频等不要压缩，因为不但不会减小，在压缩时消耗cpu和内存资源。

注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，请从下面的链接获取：[码云](#) 或者 [语雀](#)

聊聊：Nginx开启gzip压缩

Nginx 开启 Gzip 压缩功能， 可以使网站的 css、js 、xml、html 文件在传输时进行压缩，提高访问速度, 进而优化 Nginx 性能。

网站加载的速度取决于浏览器必须下载的所有文件的大小。减少要传输的文件的大小可以使网站不仅加载更快，而且对于那些宽带是按量计费的人来说也更友好。

gzip 是一种流行的数据压缩程序。

您可以使用 gzip 压缩 Nginx 实时文件。这些文件在检索时由支持它的浏览器解压缩，好处是 web 服务器和浏览器之间传输的数据量更小，速度更快。

gzip 不一定适用于所有文件的压缩。例如，文本文件压缩得非常好，通常会缩小两倍以上。

另一方面，诸如 JPEG或 PNG 文件之类的图像已经按其性质进行压缩，使用 gzip 压缩很难有好的压缩效果或者甚至没有效果。压缩文件会占用服务器资源，因此最好只压缩那些压缩效果好的文件。

聊聊：开启压缩功能的好坏

好处:

压缩是可以节省带宽，提高传输效率。

坏处:

但是由于是在服务器上进行压缩，会消耗服务器起源

聊聊：Nginx的gzip压缩的原理和优点

Nginx 开启 Gzip 压缩功能， 可以使网站的 css、js 、xml、html 文件在传输时进行压缩，提高访问速度, 进而优化 Nginx 性能。

网站加载的速度取决于浏览器必须下载的所有文件的大小。减少要传输的文件的大小可以使网站不仅加载更快，而且对于那些宽带是按量计费的人来说也更友好。

gzip 是一种流行的数据压缩程序。您可以使用 gzip 压缩 Nginx 实时文件。这些文件在检索时由支持它的浏览器解压缩，好处是 web 服务器和浏览器之间传输的数据量更小，速度更快。

gzip 不一定适用于所有文件的压缩。例如，文本文件压缩得非常好，通常会缩小两倍以上。另一方面，诸如 JPEG或 PNG 文件之类的图像已经按其性质进行压缩，使用 gzip 压缩很难有好的压缩效果或者甚至没有效果。压缩文件会占用服务器资源，因此最好只压缩那些压缩效果好的文件。

Nginx gzip配置作用

Nginx 开启 Gzip 压缩功能， 可以使网站的 css、js 、xml、html 文件在传输时进行压缩，提高访问速度, 进而优化 Nginx 性能! Web 网站上的图片，视频等其它多媒体文件以及大文件，因为压缩效果不好，所以对于图片没有必要支压缩，如果想要优化，可以图片的生命周期设置长一点，让客户端来缓存。

开启 Gzip 功能后，Nginx 服务器会根据配置的策略对发送的内容, 如 css、js、xml、html 等静态资源进行压缩, 使得这些内容大小减少，在用户接收到返回内容之前对其进行处理，以压缩后的数据展现给客户。这样不仅可以节约大量的出口带宽，提高传输效率，还能提升用户快的感知体验, 一举两得; 尽管会消耗一定的 cpu 资源，但是为了给用户更好的体验还是值得的。

经过 Gzip 压缩后页面大小可以变为原来的 30% 甚至更小，这样，用户浏览页面的时候速度会快得多。Gzip 的压缩页面需要浏览器和服务器双方都支持，实际上就是服务器端压缩，传到浏览器后浏览器解压并解析。浏览器那里不需要我们担心，因为目前的巨大多数浏览器 都支持解析 Gzip 过的页面。

Nginx gzip配置参数

参数

参数	描述
gzip on	决定是否开启 gzip 模块，on 表示开启， off 表示关闭
gzip_min_length 1k	设置允许压缩的页面最小字节(从 header 头的 Content-Length 中获取) ， 当返回内容大于此值时才会使用 gzip 进行压缩,以 K 为单位,当值为 0 时， 所有页面都进行压缩。建议大于 1k
gzip_buffers 4 16k	设置 gzip 申请内存的大小， 其作用是按块大小的倍数申请内存空间,param2:int(k) 后面单位是 k。这里设置以 16k 为单位,按照原始数据大小以 16k 为单位的 4 倍申请内存
gzip_http_version 1.1	识别 http 协议的版本,早起浏览器可能不支持 gzip 自解压,用户会看到乱码
gzip_comp_level 2	设置 gzip 压缩等级， 等级越底压缩速度越快文件压缩比越小， 反之速度越慢文件压缩比越大； 等级1-9， 最小的压缩最快 但是消耗 cpu
gzip_types text/plain	设置需要压缩的 MIME 类型,非设置值不进行压缩， 即匹配压缩类型
gzip_vary on	启用应答头"Vary: Accept-Encoding"
gzip_proxied off	nginx 做为反向代理时启用
gzip_disable msie6	IE5.5 和 IE6 SP1 使用 msie6 参数来禁止 gzip 压缩)指定哪些不需要 gzip 压缩的浏览器(将和User-Agents进行匹配),依赖于 PCRE 库

Nginx gzip参数介绍

gzip on

打开或关闭gzip

默认 off 关闭

代码块 http, server, location, if in location

gzip_buffers

设置用于处理请求压缩的缓冲区数量和大小。比如 32 4K 表示按照内存页（one memory page）大小以 4K 为单位（即一个系统中内存页为 4K）， 申请 32 倍的内存空间。建议此项不设置，使用默认值。

```
Syntax: gzip_buffers number size;
Default:
gzip_buffers 32 4k|16 8k;
Context:      http, server, location
```

gzip_comp_level

设置 gzip 压缩级别，级别越底压缩速度越快文件压缩比越小，反之速度越慢文件压缩比越大

```
Syntax: gzip_comp_level level;
Default:
gzip_comp_level 1;
Context:      http, server, location
```

不是压缩级别越高越好，其实 gzip_comp_level 1 的压缩能力已经够用了，后面级别越高，压缩的比例其实增长不大，反而很吃处理性能。

另一方面，压缩一定要和静态资源缓存相结合，缓存压缩后的版本，否则每次都压缩高负载下服务器肯定吃不住。

gzip_disable

通过表达式，表明哪些 UA 头不使用 gzip 压缩

```
Syntax: gzip_disable regex ...;
Default:      -
Context:      http, server, location
This directive appeared in version 0.6.23.
```

gzip_min_length

当返回内容大于此值时才会使用gzip进行压缩,以 K 为单位,当值为 0 时，所有页面都进行压缩。

```
Syntax: gzip_min_length length;
Default:
gzip_min_length 20;
Context:      http, server, location
```

gzip_http_version

用于识别 http 协议的版本，早期的浏览器不支持 gzip 压缩，用户会看到乱码，所以为了支持前期版本加了此选项。默认在 http/1.0 的协议下不开启 gzip 压缩。

```
Syntax: gzip_http_version 1.0 | 1.1;
Default:
gzip_http_version 1.1;
Context:      http, server, location
```

在应用服务器前，如果还有一层 Nginx 的集群作为负载均衡，在这一层上，若果没有开启 gzip。

如果我们使用了proxy_pass 进行反向代理，那么 nginx 和后端的 upstream server 之间默认是用 HTTP/1.0 协议通信的。

如果我们的 Cache Server 也是 nginx，而前端的 nginx 没有开启 gzip。同时，我们后端的 nginx 上没有设置gzip_http_version 为 1.0，那么 Cache 的 url 将不会进行 gzip 压缩。

gzip_proxied

Nginx 做为反向代理的时候启用：

- off – 关闭所有的代理结果数据压缩
- expired – 如果header中包含"Expires"头信息， 启用压缩
- no-cache – 如果header中包含"Cache-Control:no-cache"头信息， 启用压缩
- no-store – 如果header中包含"Cache-Control:no-store"头信息， 启用压缩
- private – 如果header中包含"Cache-Control:private"头信息， 启用压缩
- no_last_modified – 启用压缩， 如果header中包含"Last_Modified"头信息， 启用压缩
- no_etag – 启用压缩， 如果header中包含"ETag"头信息， 启用压缩
- auth – 启用压缩， 如果header中包含"Authorization"头信息， 启用压缩
- any – 无条件压缩所有结果数据

```
Syntax: gzip_proxied off | expired | no-cache | no-store | private | no_last_modified | no_etag | auth | any ...;
Default:
gzip_proxied off;
Context:      http, server, location
```

gzip_types

设置需要压缩的 MIME 类型,如果不在设置类型范围内的请求不进行压缩

```
Syntax: gzip_types mime-type ...;
Default:
```

```
gzip_types text/html;
Context:    http, server, location
```

gzip_vary

增加响应头“Vary: Accept-Encoding”，告诉接收方发送的数据经过了压缩处理，开启后的效果是在响应头部添加了Accept-Encoding:gzip，这对于本身不支持 gzip 压缩的客户端浏览器有用。

```
Syntax: gzip_vary on | off;
Default:
gzip_vary off;
Context:    http, server, location
```

Nginx gzip 案例

我们首先，使用 vim 打开 nginx 的默认配置路径，具体命令如下：

```
vim /etc/nginx/conf.d/default.conf
```


我们执行如上命令，打开配置文件，此时配置文件如下：

```
server {
    listen      80;
    server_name localhost;

    #charset koi8-r;
    #access_log  /var/log/nginx/host.access.log  main;

    location / {
        root    /usr/share/nginx/html;
        index   index.html index.htm;
    }
```

现在，我们在 nginx 的根路径下新建一个 index.html，并写入内容，并使用浏览器访问，此时，浏览器输出如下：

Name	× Headers Preview Response Timing
 192.168.80.128	<div>Remote Address: 192.168.80.128:80</div> <div>Referrer Policy: no-referrer-when-downgrade</div> <hr/> <div>▼ Response Headers view source</div> <div>Accept-Ranges: bytes</div> <div>Connection: keep-alive</div> <div>Content-Length: 859825</div> <div>Content-Type: text/html</div>

现在，我们打开 nginx.conf 配置文件，如下图所示：

```
haicoder(www.haicoder.net)# vim /etc/nginx/nginx.conf
```

在 nginx 的 http 模块的配置里面，开启 gzip 配置，具体配置如下：

```
gzip on;
gzip_min_length 1k;
gzip_buffers 4 16k;
gzip_http_version 1.1;
gzip_comp_level 9;
gzip_types text/plain application/x-javascript text/css application/xml text/javascript application/x-httpd-php application/javascript;
gzip_disable "MSIE [1-6]\.";
gzip_vary on;
```



配置完成后，如下图所示：

```
http {  
    include      /etc/nginx/mime.types;  
    default_type application/octet-stream;  
  
    log_format  main  '$http_user_agent $remote_addr - $remote_user [$time_local] "$request" '  
                      '$status $body_bytes_sent "$http_referer" '  
                      '"$http_user_agent" "$http_x_forwarded_for"';  
  
    access_log  /var/log/nginx/access.log  main;  
  
    sendfile    on;  
    #tcp_nopush  on;  
  
    keepalive_timeout  65;  
  
    gzip  on;  
    gzip_min_length  1k;  
    gzip_buffers      4 16k;  
    gzip_http_version 1.1;  
    gzip_comp_level  9;  
    gzip_types        text/plain application/x-javascript text/css application/xml text/javascript application/x-ht  
    gzip_disable "MSIE [1-6]\.";  
    gzip_vary on;  
}
```

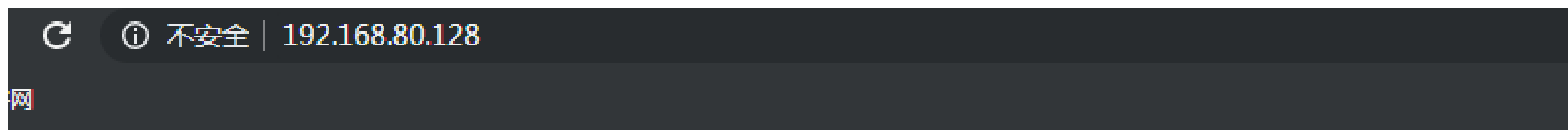
现在，我们使用 reload 重新加载配置，具体命令如下：

```
nginx -s reload
```

执行完毕后，我们再次使用浏览器访问，此时输出如下：

Name	×	Headers	Preview	Response	Timing
 192.168.80.128	<div>▼ General</div> <div>Request URL: http://192.168.80.128/</div> <div>Request Method: GET</div> <div>Status Code:  200 OK</div> <div>Remote Address: 192.168.80.128:80</div> <div>Referrer Policy: no-referrer-when-downgrade</div> <div>▼ Response Headers view source</div> <div>Connection: keep-alive</div> <div><div>Content-Encoding: gzip</div></div> <div>Content-Type: text/html</div>				

我们看到，此时输出了 gzip，并且浏览器的内容如下：



00

**Welcome to
haicoder(www.haicoder.net)**

我们看到，浏览器的内容也正常输出了。

Nginx gzip配置总结

Nginx 开启 Gzip 压缩功能， 可以使网站的 css、js 、xml、html 文件在传输时进行压缩， 提高访问速度, 进而优化 Nginx 性能。

聊聊： Nginx是否支持将请求压缩到上游？

您可以使用Nginx模块gunzip将请求压缩到上游。

gunzip模块是一个过滤器， 它可以对不支持“gzip”编码方法的客户机或服务器使用“内容编码:gzip”来解压缩响应。

聊聊： 如何在Nginx中获得当前的时间？

要获得Nginx的当前时间， 必须使用SSI模块、\$date_gmt和 \$date_local的变量。

```
Proxy_set_header THE-TIME $date_gmt;
```

聊聊： 用Nginx服务器解释-s的目的是什么？

用于运行Nginx -s参数的可执行文件。

聊聊： 如何在Nginx服务器上添加模块？

在编译过程中， 必须选择Nginx模块， 因为Nginx不支持模块的运行时间选择。

Nginx动态添加模块

很多时候， 我们根据当时的项目情况和业务需求安装完 Nginx 后， 后续随着业务的发展， 往往会给安装好的 Nginx 添加其他的功能模块。在为 Nginx 添加功能模块时， 要求 Nginx 不停机。

这就涉及到如何为已安装的 Nginx 动态添加模块的问题。本文， 就和小伙伴们一起探讨如何为已安装的 Nginx 动态添加模块的问题。

聊聊： 如何为Nginx动态添加模块

这里以安装第三方 ngx_http_google_filter_module 模块为例。

Nginx 的模块是需要重新编译 Nginx， 而不是像 Apache 一样配置文件引用 .so。

下载扩展

下载第三方扩展模块 ngx_http_google_filter_module

```
# cd /data/software/  
# git clone https://github.com/cuber/nginx_http_google_filter_module
```

查看安装模块

查看 Nginx 编译安装时安装了哪些模块，将命令行切换到 Nginx 执行程序所在的目录并输入 ./nginx -V，具体如下：

```
[root@binghe sbin]# ./nginx -V  
nginx version: nginx/1.19.1  
built by gcc 4.4.7 20120313 (Red Hat 4.4.7-17) (GCC)  
built with OpenSSL 1.0.2 22 Jan 2015  
TLS SNI support enabled  
configure arguments: --prefix=/usr/local/nginx-1.19.1 --with-openssl=/usr/local/src/openssl-1.0.2 --with-pcre=/usr/local/src/pcre-8.37 --with-zlib=/usr/local/src/zlib-1.2.11  
[root@binghe sbin]#
```

可以看出编译安装 Nginx 使用的参数如下：

```
--prefix=/usr/local/nginx-1.19.1 --with-openssl=/usr/local/src/openssl-1.0.2 --with-pcre=/usr/local/src/pcre-8.37 --with-zlib=/usr/local/src/zlib-1.2.11
```

重新编译

加入需要安装的模块，重新编译，这里添加 --add-module=/data/software/nginx_http_google_filter_module，具体如下：

```
./configure --prefix=/usr/local/nginx-1.19.1 --with-openssl=/usr/local/src/openssl-1.0.2 --with-pcre=/usr/local/src/pcre-8.37 --with-zlib=/usr/local/src/zlib-1.2.11 --add-module=/data/software/nginx_http_google_filter_module
```

如上，将之前安装Nginx的参数全部加上，最后添加 --add-module=/data/software/nginx_http_google_filter_module，之后，我们要进行编译操作，如下：

```
# make //千万不要make install，不然就真的覆盖
```

这里，需要注意的是：不要执行 make install 命令。

替换nginx二进制文件

```
# 备份原来的nginx执行程序  
# mv /usr/local/nginx-1.19.1/sbin/nginx /usr/local/nginx-1.19.1/sbin/nginx.bak
```

```
# 将新编译的nginx执行程序复制到/usr/local/nginx-1.19.1/sbin/目录下
# cp /opt/nginx/sbin/nginx /usr/local/nginx-1.19.1/sbin/
```

聊聊：如何设置超时时间

```
http {
    keepalive_timeout 60; ###设置客户端连接保持会话的超时时间，超过这个时间，服务器会关闭该连接。

    tcp_nodelay on; ####打开 tcp_nodelay，在包含了 keepalive 参数才有效

    client_header_timeout 15; ####设置客户端请求头读取超时时间，如果超过这个时间，客户端还没有发送任何数据， Nginx 将返回“Request time out(408)”

    client_body_timeout 15;####设置客户端请求主体读取超时时间，如果超过这个时间，客户端还没有发送任何数据， Nginx 将返回“Request time out(408)”

    send_timeout 15; ####指定响应客户端的超时时间。这个超过仅限于两个连接活动之间的时间，如果超过这个时间，客户端没有任何活动，Nginx 将会关闭连接。

}
```

聊聊：Nginx如何限制浏览器和爬虫

Nginx限制爬虫

修改 nginx.conf，禁止网络爬虫的 ua，返回 403，具体配置如下：

```
server{
    listen 80;
    server_name 127.0.0.1;
    # 添加如下内容即可防止爬虫
    if ($http_user_agent ~* "qihoobot|Baiduspider|Googlebot|Googlebot-Mobile|Googlebot-Image|Mediapartners-Google|Adsbot-Google|f
    {
        return 403;
    }
}
```

Nginx限制浏览器访问

限制浏览器访问：


```
if ($http_user_agent ~* "Firefox|MSIE")
{
    return 403;
}
```

Nginx限制IP访问

有时候我们需要针对屏蔽某些恶意的 IP 访问我们的网站，或者限制仅仅某些白名单 IP 才能访问我们的网站。

这时候我们就可以在Nginx中通过简单的配置来达到目的。

```
#添加IP至allow（例如我们将10.208.96.192和10.208.96.193加入）
location = /index.html
{
    allow 10.208.96.192;
    allow 10.208.96.193;
    deny all;
    root /work/weichuangli;
}
```

屏蔽单个ip访问

```
# 格式： deny ip;
deny 123.68.23.5;
```

允许单个ip访问

```
# 格式： allow ip;
allow 123.68.25.6;
```

屏蔽所有ip访问

```
deny all;
```

允许所有ip访问

```
allow all;
```

屏蔽ip段访问

```
# deny ip/mask
# 屏蔽172.12.62.0到172.45.62.255访问的命令
deny 172.12.62.0/24;
```

允许ip段访问

```
# allow ip/mask
# 屏蔽172.102.0.0到172.102.255.255访问的命令
allow 172.102.0.0/16;
```

聊聊：502报错可能原因有哪些？



- 1) FastCGI进程是否已经启动
- 2) FastCGI worker进程数是否不够
- 3) FastCGI执行时间过长
- 4) FastCGI Buffer不够

nginx和apache一样，有前端缓冲限制，可以调整缓冲参数

```
fastcgi_buffer_size 32k;
fastcgi_buffers 8 32k;
```

- 5) Proxy Buffer不够

如果你用了Proxying，调整

```
proxy_buffer_size 16k;
proxy_buffers 4 16k;
```

6) php脚本执行时间过长

将php-fpm.conf的<value name="request_terminate_timeout">0s</value>的0s改成一个时间

聊聊：Nginx 如何解决跨域问题

跨域是前端开发中经常会遇到的问题，前端调用后台服务时，通常会遇到 No ‘Access-Control-Allow-Origin’ header is present on the requested resource 的错误，这是因为浏览器的同源策略拒绝了我们的请求。

所谓同源是指，域名，协议，端口相同，浏览器执行一个脚本时同源的脚本才会被执行。

如果非同源，那么在请求数据时，浏览器会在控制台中报一个异常，提示拒绝访问。

这个问题我们通常会使用 CORS(跨源资源共享)或者 JSONP 去解决，这两种方法也是使用较多的方法。

细聊：Nginx 如何 解决跨域问题

什么是跨域

跨域是前端开发中经常会遇到的问题，前端调用后台服务时，通常会遇到 No ‘Access-Control-Allow-Origin’ header is present on the requested resource 的错误，这是因为浏览器的同源策略拒绝了我们的请求。

所谓同源是指，域名，协议，端口相同，浏览器执行一个脚本时同源的脚本才会被执行。如果非同源，那么在请求数据时，浏览器会在控制台中报一个异常，提示拒绝访问。这个问题我们通常会使用 CORS(跨源资源共享)或者 JSONP 去解决，这两种方法也是使用较多的方法。

Nginx解决跨域方案一

解决跨域

这个使用 Nginx 的代理功能即可，在 a 服务器的 Nginx 添加如下示例配置：

```
location ~ /xxx/ {
    proxy_pass http://b.com;
}
```

这样就把路径中带有 /xxx/ 的请求都转到了 b.com。如果不需要保存 cookie，保持 session 这样的功能，这样就可以了。

然而，本项目就是要用到 cookie，所以就有了下边的内容。

设置domain

因为 cookie 当中是有 domain 的，两个服务器的一般不同，比如 a 服务器返回的 Response Headers 中是

```
Set-Cookie:JSESSIONID=_3y4u02v4cbpBw10DoCrMSnjg7m34xuum1XRWBF1Uno; path=/; domain=a.com
```

而 b 服务器返回的是

```
Set-Cookie:JSESSIONID=_3y4u02v4cbpBw10DoCrMSnjg7m34xuum1XRWBF1Uno; path=/; domain=b.com
```

这时候如果 a 项目的页面调用了 b 的接口，浏览器发现接口返回的 domain 不是 a.com，就不会把 cookie 保存起来，session 也就失效了。Nginx 引入了 proxy_cookie_domain 来解决这个问题。示例：

```
location ~ /xxx/ {
    proxy_cookie_domain b.com a.com;
    proxy_pass http://b.com;
}
```

这样就可以在 Nginx 转接请求的时候自动把 domain 中的 b.com 转换成 a.com，这样 cookie 就可以设置成功了。

但是，对于有些情况这样转换不灵光。比如，b 项目的 domain 是 .b.com，前边多了一个小点，那对应的改为 proxy_cookie_domain .b.com a.com; 可以不？通过实践，不行！！！！

通过查看 Nginx 文档，找到了解决办法。其实，除了上边那种配置方式外，Nginx 还支持正则配置：

```
location ~ /xxx/ {
    proxy_cookie_domain ~\.?b.com a.com;
    proxy_pass http://b.com;
}
```

这样就可以把 domain 中的 .b.com 转换成 a.com 啦。

设置path

正常情况下完成以上两步就可以了，因为 cookie 中的 path 一般默认的是 path=/，也就是所有请求都可以访问本 cookie。但有些服务器会指定，只允许某个层级下的请求可以访问 cookie，比如：

```
Set-Cookie:JSESSIONID=_3y4u02v4cbpBw10DoCrMSnjg7m34xuum1XRWBF1Uno; path=/sub/; domain=b.com
```

这样就只允许相对根路径，以 /sub/ 开头的请求路径才能访问 cookie。这时候就又可能出现 cookie 无效的问题了，为了解决这个问题，可以使用 proxy_cookie_path。示例：

```
location ~ /xxx/ {
    proxy_cookie_domain ~\.?b.com a.com;
    proxy_cookie_path /sub/ /;
    proxy_pass http://b.com;
}
```

这样就把只允许 /sub/ 层级下的请求访问 cookie，改为允许所有请求访问 cookie 了。

Nginx解决跨域方案二

或者，我们也可以直接简单粗暴的设置全局配置了，如下：

```
http {
    include      mime.types;
    default_type  application/octet-stream;
    sendfile      on;
    #连接超时时间，服务器会在这个时间过后关闭连接。
    keepalive_timeout  10;
    # gzip压缩
    gzip  on;
    # 直接请求nginx也是会报跨域错误的这里设置允许跨域
    # 如果代理地址已经允许跨域则不需要这些， 否则报错(虽然这样nginx跨域就没意义了)
    add_header Access-Control-Allow-Origin *;
    add_header Access-Control-Allow-Headers X-Requested-With;
    add_header Access-Control-Allow-Methods GET,POST,OPTIONS;
    # srever模块配置是http模块中的一个子模块，用来定义一个虚拟访问主机
    server {
        listen      80;
        server_name  localhost;
        # 根路径指到index.html
        location / {
```

```
        root    html;
        index   index.html index.htm;
    }
# localhost/api 的请求会被转发到192.168.0.103:8080
location /api {
    rewrite ^/b/(.*)$ /$1 break; # 去除本地接口/api前缀，否则会出现404
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_pass http://192.168.0.103:8080; # 转发地址
}
# 重定向错误页面到/50x.html
error_page    500 502 503 504    /50x.html;
location = /50x.html {
    root    html;
}
}
}
```

聊聊：漏桶流算法和令牌桶算法知道？

漏桶算法

漏桶算法是网络世界中流量整形或速率限制时经常使用的一种算法，它的主要目的是控制数据注入到网络的速率，平滑网络上的突发流量。漏桶算法提供了一种机制，通过它，突发流量可以被整形以便为网络提供一个稳定的流量。也就是我们刚才所讲的情况。漏桶算法提供的机制实际上就是刚才的案例：突发流量会进入到一个漏桶，漏桶会按照我们定义的速率依次处理请求，如果水流过大也就是突发流量过大就会直接溢出，则多余的请求会被拒绝。所以漏桶算法能控制数据的传输速率。

令牌桶算法

令牌桶算法是网络流量整形和速率限制中最常使用的一种算法。典型情况下，令牌桶算法用来控制发送到网络上的数据的数目，并允许突发数据的发送。Google开源项目Guava中的RateLimiter使用的就是令牌桶控制算法。令牌桶算法的机制如下：存在一个大小固定的令牌桶，会以恒定的速率源源不断产生令牌。如果令牌消耗速率小于生产令牌的速度，令牌就会一直产生直至装满整个令牌桶。

参考：

限流：计数器、漏桶、令牌桶 三大算法的原理与实战（史上最全）

聊聊： Nginx限流怎么做的？

Nginx限流就是限制用户请求速度，防止服务器受不了

限流有3种

正常限制访问频率（正常流量）

突发限制访问频率（突发流量）

限制并发连接数

Nginx的限流都是基于漏桶流算法， 底下会说道什么是漏桶流

正常限制访问频率（正常流量）

限制一个用户发送的请求， 我Nginx多久接收一个请求。

Nginx中使用ngx_http_limit_req_module模块来限制的访问频率， 限制的原理实质是基于漏桶算法原理来实现的。在nginx.conf配置文件中可以使用limit_req_zone命令及limit_req命令限制单个IP的请求处理频率。

#定义限流维度， 一个用户一分钟一个请求进来， 多余的全部漏掉

```
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/m;
```

#绑定限流维度

```
server{

    location/seckill.html{

        limit_req zone=zone;

        proxy_pass http://lj_seckill;

    }

}
```

突发限制访问频率（突发流量）

限制一个用户发送的请求， 我Nginx多久接收一个。

上面的配置一定程度可以限制访问频率， 但是也存在着一个问题： 如果突发流量超出请求被拒绝处理， 无法处理活动时候的突发流量， 这时候应该如何进一步处理呢？ Nginx提供burst参数结合nodelay参数可以解决流量突发的问题， 可以设置能处理的超过设置的请求数外能额外处理的请求数。我们可以将之前的例子添加burst参数以及nodelay参数：

#定义限流维度，一个用户一分钟一个请求进来，多余的全部漏掉

```
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/m;
```

#绑定限流维度

```
server{

    location/seckill.html{
        limit_req zone=zone burst=5 nodelay;
        proxy_pass http://lj_seckill;
    }

}
```

为什么就多了一个 burst=5 nodelay; 呢，多了这个可以代表Nginx对于一个用户的请求会立即处理前五个，多余的就慢慢来落，没有其他用户的请求我就处理你的，有其他的请求的话我Nginx就漏掉不接受你的请求

限制并发连接数

Nginx中的ngx_http_limit_conn_module模块提供了限制并发连接数的功能，可以使用limit_conn_zone指令以及limit_conn执行进行配置。接下来我们可以通过一个简单的例子来看下：

```
http {
    limit_conn_zone $binary_remote_addr zone=myip:10m;
    limit_conn_zone $server_name zone=myServerName:10m;
}
```

```
server {
    location / {
        limit_conn myip 10;
        limit_conn myServerName 100;
        rewrite / http://www.lijie.net permanent;
    }
}
```

上面配置了单个IP同时并发连接数最多只能10个连接，并且设置了整个虚拟服务器同时最大并发数最多只能100个链接。当然，只有当请求的header被服务器处理后，虚拟服务器的连接数才会计数。刚才有提到过Nginx是基于漏桶算法原理实现的，实际上限流一般都是基于漏桶算法和令牌桶算法实现的。接下来我们来看看两个算法的介绍：

聊聊：限流了解吗，怎么限流的？

Nginx 提供两种限流方式，一是控制速率，二是控制并发连接数。

1、控制速率 限流

ngx_http_limit_req_module 模块提供了漏桶算法(leaky bucket)，可以限制单个IP的请求处理频率。

Nginx限流使用的是leaky bucket算法，

限流：计数器、漏桶、令牌桶 三大算法的原理与实战（史上最全）

控制速率基本原理

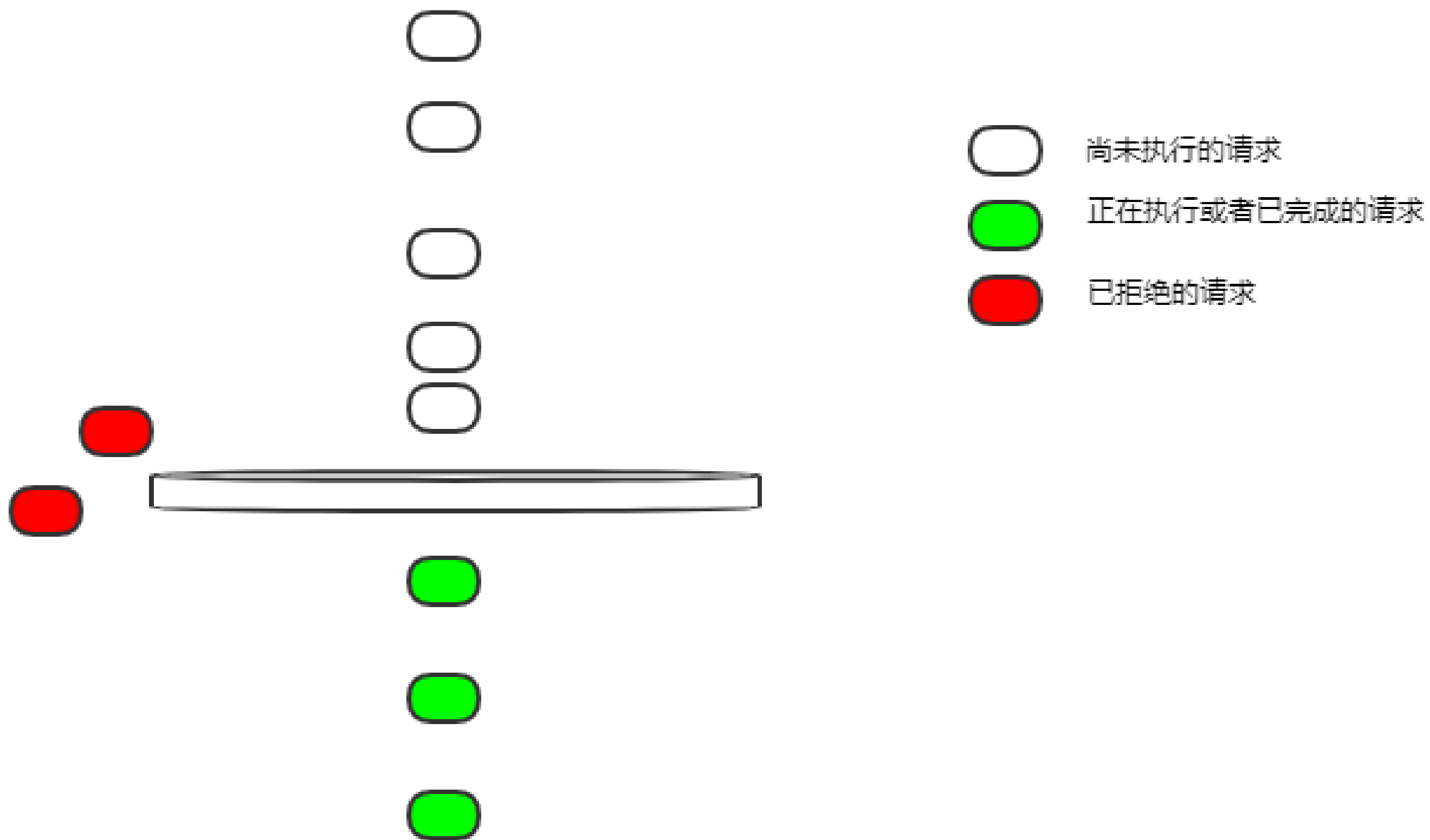
我们从最简单的限流配置开始：

```
limit_req_zone $binary_remote_addr zone=ip_limit:10m rate=10r/s;

server {
    location /login/ {
        limit_req zone=ip_limit;
        proxy_pass http://login_upstream;
    }
}

$binary_remote_addr 针对客户端ip限流；
zone=ip_limit:10m 限流规则名称为ip_limit，允许使用10MB的内存空间来记录ip对应的限流状态；
rate=10r/s 限流速度为每秒10次请求
location /login/ 对登录进行限流
```

限流速度为每秒10次请求，如果有10次请求同时到达一个空闲的nginx，他们都能得到执行吗？



漏桶漏出请求是匀速的。

10r/s是怎样匀速的呢？每100ms漏出一个请求。

在这样的配置下，桶是空的，所有不能实时漏出的请求，都会被拒绝掉。

所以如果10次请求同时到达，那么只有一个请求能够得到执行，其它的，都会被拒绝。

这不太友好，大部分业务场景下我们希望这10个请求都能得到执行。

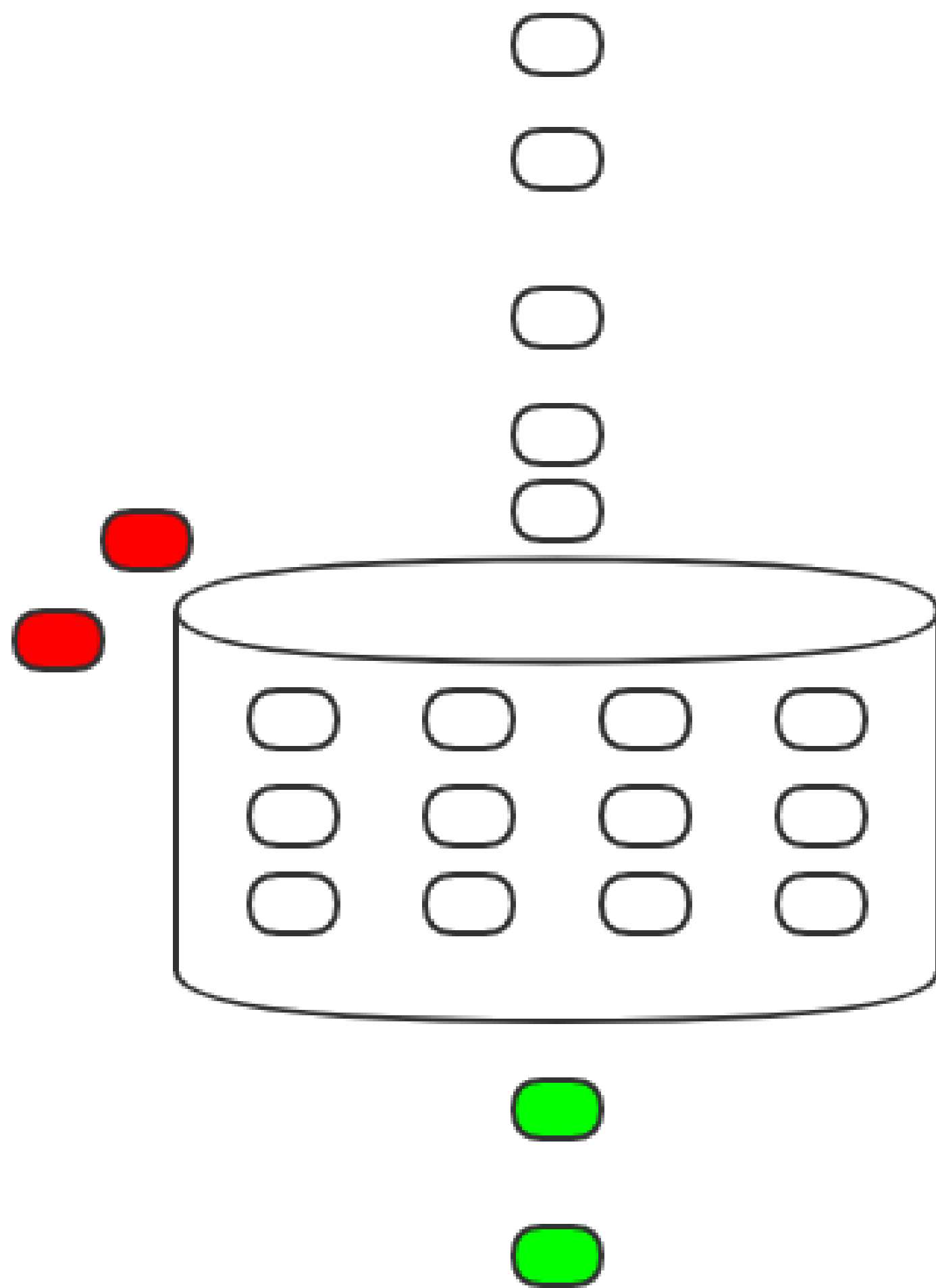
Burst

我们把配置改一下，解决上一节的问题

```
limit_req_zone $binary_remote_addr zone=ip_limit:10m rate=10r/s;
```

```
server {  
    location /login/ {  
        limit_req zone=ip_limit burst=12;  
        proxy_pass http://login_upstream;  
    }  
}
```

burst=12 漏桶的大小设置为12



尚未执行的请求



正在执行或者已完成的请求



已拒绝的请求



逻辑上叫漏桶，实现起来是FIFO队列，把得不到执行的请求暂时缓存起来。

这样漏出的速度仍然是100ms一个请求，但并发而来，暂时得不到执行的请求，可以先缓存起来。只有当队列满了的时候，才会拒绝接受新请求。

这样漏桶在限流的同时，也起到了削峰填谷的作用。

在这样的配置下，如果有10次请求同时到达，它们会依次执行，每100ms执行1个。

虽然得到执行了，但因为排队执行，延迟大大增加，在很多场景下仍然是不能接受的。

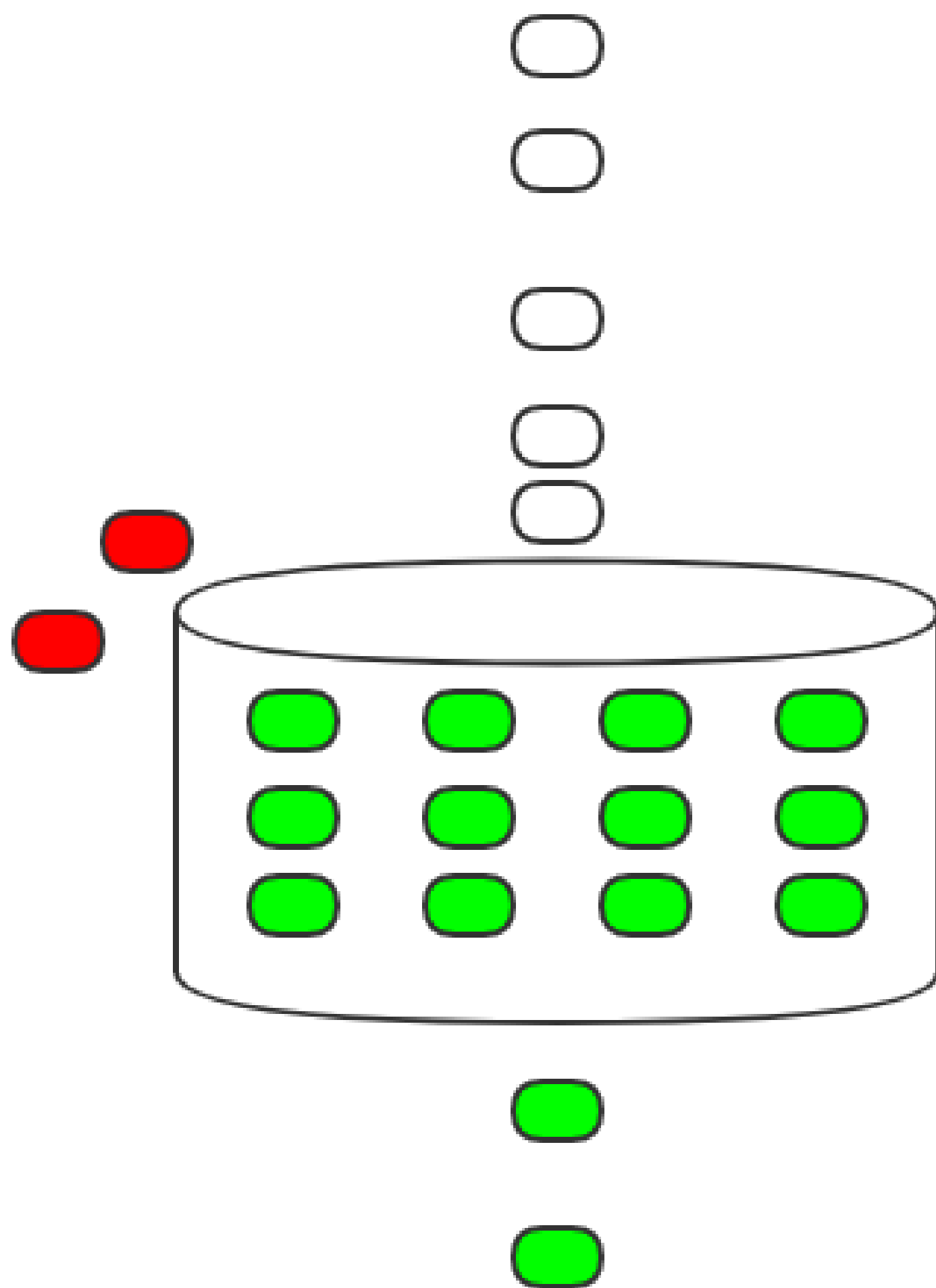
NoDelay

继续修改配置，解决Delay太久导致延迟增加的问题

```
limit_req_zone $binary_remote_addr zone=ip_limit:10m rate=10r/s;
```

```
server {  
    location /login/ {  
        limit_req zone=ip_limit burst=12 nodelay;  
        proxy_pass http://login_upstream;  
    }  
}
```

nodelay 把开始执行请求的时间提前，以前是delay到从桶里漏出来才执行，现在不delay了，只要入桶就开始执行



尚未执行的请求



正在执行或者已完成的请求



已拒绝的请求



要么立刻执行，要么被拒绝，请求不会因为限流而增加延迟了。

因为请求从桶里漏出来还是匀速的，桶的空间又是固定的，最终平均下来，还是每秒执行了5次请求，限流的目的还是达到了。

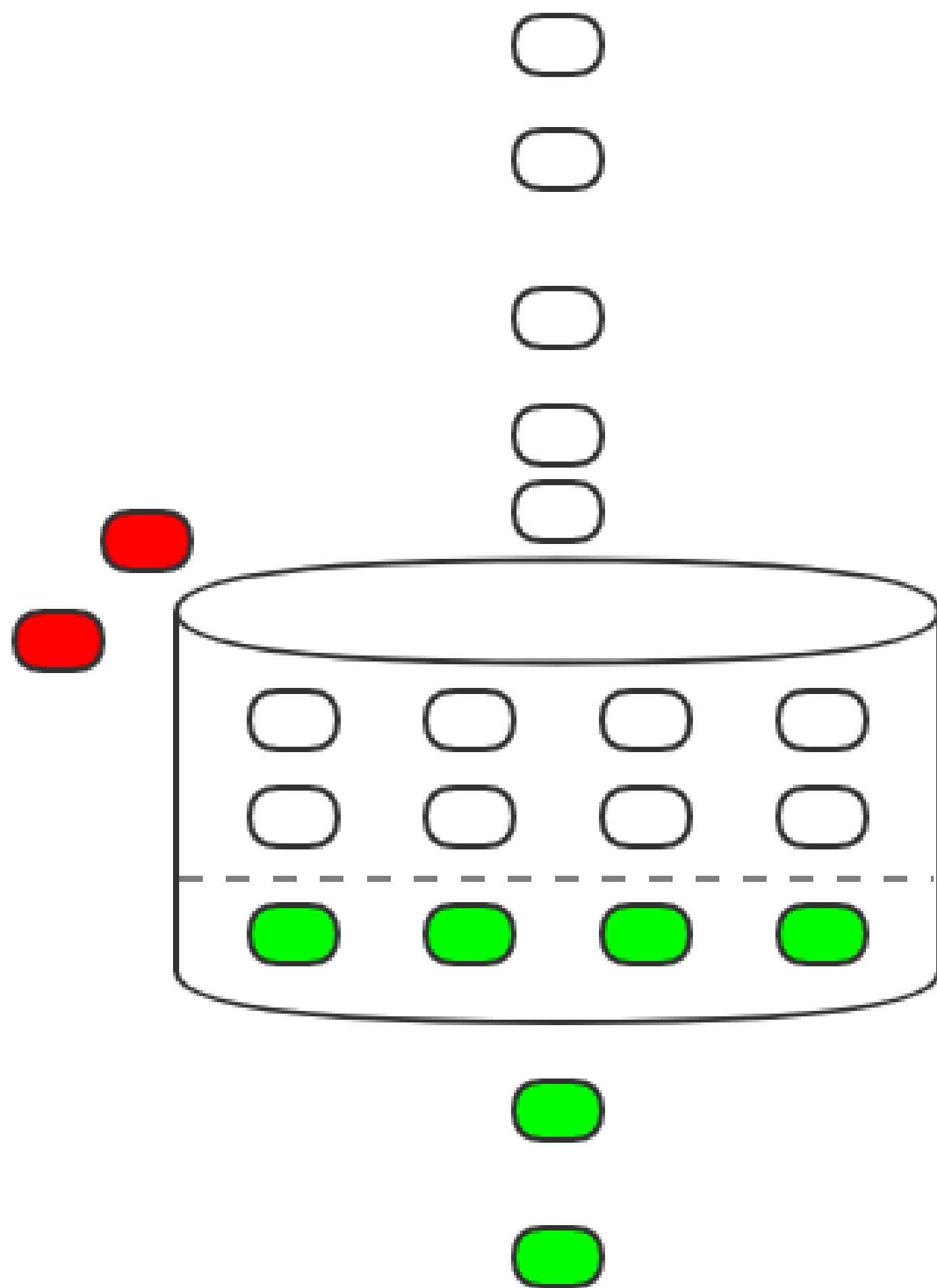
但这样也有缺点，限流是限了，但是限得不那么匀速。以上面的配置举例，如果有12个请求同时到达，那么这12个请求都能够立刻执行，然后后面的请求只能匀速进桶，100ms执行1个。如果有一段时间没有请求，桶空了，那么又可能出现并发的12个请求一起执行。

大部分情况下，这种限流不均匀，不算是大问题。不过nginx也提供了一个参数来控制并发执行也就是nodelay的请求的数量。

```
limit_req_zone $binary_remote_addr zone=ip_limit:10m rate=10r/s;
```

```
server {  
    location /login/ {  
        limit_req zone=ip_limit burst=12 delay=4;  
        proxy_pass http://login_upstream;  
    }  
}
```

delay=4 从桶内第5个请求开始delay



尚未执行的请求



正在执行或者已完成的请求



已拒绝的请求



这样通过控制delay参数的值，可以调整允许并发执行的请求的数量，使得请求变的均匀起来，在有些耗资源的服务上控制这个数量，还是有必要的。

控制速率 基本原理讲完了，

接下来，开始 控制速率 限流 的基础配置

1.1 控制速率 限流 的基础配置：

基于客户端192.168.1.1进行限流，

定义了一个大小为10M，名称为myLimit的内存区，用于存储IP地址访问信息。

rate设置IP访问频率，rate=5r/s表示每秒只能处理每个IP地址的5个请求。

```
http {  
    limit_req_zone 192.168.1.1 zone=myLimit:10m rate=5r/s;  
}
```

```
server {  
    location / {  
        limit_req zone=myLimit;  
        rewrite / http://www.hac.cn permanent;  
    }  
}
```

参数解释：

limit_req_zone: 定义需要限流的对象。

zone: 定义共享内存区来存储访问信息。

rate: 用于设置最大访问速率。

Nginx限流是按照毫秒级为单位的，也就是说1秒处理5个请求会变成每200ms只处理一个请求。如果200ms内已经处理完1个请求，但是还是有新的请求到达，这时候Nginx就会拒绝处理该请求。

1.2 突发流量限制 访问频率

上面rate设置了 5r/s,

如果有时候流量突然变大，超出的请求就被拒绝返回503了，突发的流量影响业务就不好了。

这时候可以加上burst 参数，一般再结合 nodelay 一起使用。

```
server {
    location / {
        limit_req zone=myLimit burst=20 nodelay;
        rewrite / http://www.hac.cn permanent;
    }
}
```

burst=20 nodelay 表示这20个请求立马处理，不能延迟，相当于特事特办。不过，即使这20个突发请求立马处理结束，后续来了请求也不会立马处理。

burst=20 相当于缓存队列中占了20个坑，即使请求被处理了，这20个位置也只能按 100ms一个来释放。

2、控制并发连接数

ngx_http_limit_conn_module 提供了限制连接数功能。

主要是利用limit_conn_zone和limit_conn两个指令。

利用连接数限制 某一个用户的ip连接的数量来控制流量。

```
limit_conn_zone $binary_remote_addr zone=perip:10m;
limit_conn_zone $server_name zone=perserver:10m;

server {
    listen      80;
    server_name localhost;
    charset utf-8;
```

```

    location / {
        limit_conn perip 10;      # 单个客户端ip与服务器的连接数.
        limit_conn perserver 100; # 限制与服务器的总连接数
        root    html;
        index   index.html index.htm;
    }
}

```

limit_conn perip 10 作用的key 是 \$binary_remote_addr, 表示限制单个IP同时最多能持有10个连接。

limit_conn perserver 100 作用的key是 \$server_name, 表示虚拟主机(server) 同时能处理并发连接的总数。

聊聊：nginx如何配置https

```

#server端基本配置 server {
listen 80;
listen 443 ssl spdy;
server_name io.123.com;
include    ssl/io.com;      #注意看下一个文件
location / {
proxy_pass http://lb_io;
if``($scheme = http ) {
return``301 https://$host$request_uri;      #此项配置为转换为https的基本配置
}
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header Host $host;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;
proxy_http_version 1.1;
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection``"upgrade"``;
}
access_log /data/logs/nginx/access/niuaero.log main;
}
ssl_certificate    ssl/ca/io.com.pem;      #这个为购买的https证书, 供应商会生成
ssl_certificate_key ssl/ca/io.com.key;

```

```
ssl_session_timeout 5m;
ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
#启用TLS1.1、TLS1.2要求OpenSSL1.0.1及以上版本，若您的OpenSSL版本低于要求，请使用 ssl_protocols TLSv1;
ssl_ciphers HIGH:!RC4:!MD5:!aNULL:!eNULL:!NULL:!DH:!EDH:!EXP:+MEDIUM;
ssl_prefer_server_ciphers ``on``;
```

聊聊： Nginx常用优化配置

调整worker_processes指定Nginx需要创建的worker进程数量，刚才有提到worker进程数一般设置为和CPU核心数一致。

调整worker_connections设置Nginx最多可以同时服务的客户端数。结合worker_processes配置可以获得每秒可以服务的最大客户端数。

启动gzip压缩，可以对文件大小进行压缩，减少了客户端http的传输带宽，可以大幅度提高页面的加载速度。

启用缓存，如果请求静态资源，启用缓存是可以大幅度提升性能的。

聊聊： Nginx常见的优化配置有哪些？

1、调整worker_processes

指Nginx要生成的worker数量,最佳实践是每个CPU运行1个工作进程。

了解系统中的CPU核心数，输入

```
$ grep processor / proc / cpuinfo | wc -l
```

2、最大化worker_connections

Nginx Web服务器可以同时提供服务的客户端数。与worker_processes结合使用时，获得每秒可以服务的最大客户端数

最大客户端数/秒=工作进程*工作者连接数

为了最大化Nginx的全部潜力，应将工作者连接设置为核心一次可以运行的允许的最大进程数1024。

3、启用Gzip压缩

压缩文件大小，减少了客户端http的传输带宽，因此提高了页面加载速度

建议的gzip配置示例如下:(在http部分内)

```
gzip on;  
gzip_vary on;  
gzip_proxied any;  
gzip_comp_level 1 ;  
gzip_buffers 16 8k;  
gzip_http_version 1.1 ;  
gzip_types text / plain text / css application / json application / javascript text / xml  
application / xml application / xml + rss text / javascript;
```

4、为静态文件启用缓存

为静态文件启用缓存，以减少带宽并提高性能，可以添加下面的命令，限定计算机缓存网页的静态文件：

```
location ~* \.(jpg|jpeg|png|gif|ico|css|js)$ {  
    expires 365d;  
}
```

5、Timeouts

keepalive连接减少了打开和关闭连接所需的CPU和网络开销，获得最佳性能需要调整的变量可参考：

```
client_body_timeout 12;  
client_header_timeout 12;  
keepalive_timeout 15;  
send_timeout 10;
```


6、禁用access_logs

访问日志记录，它记录每个nginx请求，因此消耗了大量CPU资源，从而降低了nginx性能。

完全禁用访问日志记录

```
access_log off;
```

如果必须具有访问日志记录，则启用访问日志缓冲

配置如下：

```
access_log /var/log/nginx/access.log main buffer=32k flush=1m;
```

当系统处于负载状态时，启用日志缓冲区以降低 nginx worker 进程阻塞。

大量的磁盘读写和 cpu 资源使用对于服务器资源也是一种巨大消耗。

将日志数据缓冲到内存中可能是很小的一个优化手段， buffer 参数意义是缓冲区的大小，功能是当缓冲区已经写满时，日志会被写入文件中；

flush 参数意义是缓冲区内日志在缓冲区内存中保存的最长时间，功能即当缓存中的日志超过最大缓存时间，也会被写入到文件中， 不足的地方即写入到日志文件的日志有些许延迟，即时调试中应当关闭日志缓冲。。

7、指令定义错误日志目录及记录错误日志的等级

为了精确定位 nginx 的错误日志，使用自带的 error_log 指令定义错误日志目录及记录错误日志的等级，配置如下：

Bash

```
error_log /var/log/nginx/error.log warn;
```

error_log 指令配置时需要一个必选的日志目录和一个可选的错误等级选项。

除 if 指令外， error_log 指令能在所有的上下文中使用。错误日志等级包括：

debug、info、notice、warn、error、crit、alert 和 emerg。给出的日志

等级顺序就是记录最小到最严谨的日志等级顺序。需要注意的是 debug 日志

需要在编译 nginx 服务器时，带上 --with-debug 标识才能使用。

当服务器配置出错时，首先需要查看错误日志以定位问题。错误日志

也是定位应用服务器(如 FastCGI 服务)的利器。

通过错误日志，我们可以调试 worker 进程连接错误、内存分配、客户端 IP 和 应用服务器等问题。

错误日志格式不支持自定义日志格式；但他同样记录当前时间、日志等级和具体信息等数据。

注意：错误日志的默认设置适用于全局。

要覆盖它，请将 error_log 指令放在 main （顶级）配置上下文中。error_log 在开源 NGINX 1.5.2 版中添加了在同一配置级别指定多个指令的功能。

细致聊聊：如何进行Nginx的调优

1、worker_processes的进程数，数量要与CPU数量一致，通过lscpu查看

```
worker_processes 1;
```

2.1 worker process打开文件数的优化

```
worker_rlimit_nofile 65535;
```

2.2 优化了nginx的worker进程最多打开数量的参数之后，还需要优化系统内核参数（允许打开最多文件的参数）

临时配置：

```
ulimit -Hn 100000
```

```
ulimit -Sn 100000
```

永久配置：

```
vim /etc/security/limits.conf
```

```
* soft nofile 100000
```

```
* hard nofile 100000
```

3、单个进程最大连接数的优化

```
events {
```

```
    worker_connections 2048;
```

```
    multi_accept on;
```

```
    use epoll;  
}
```

4、隐藏版本信息的优化

```
server_tokens off;
```

5、高效文件传输模式的

```
sendfile on;  
tcp_nopush on;  
tcp_nodelay on;
```

6、访问日志关闭的优化

```
access_log off;
```

7、超时时间的优化

```
keepalive_timeout 10; //设置客户端保持活动状态的超时时间  
client_header_timeout 10; //客户端请求头读取超时时间  
client_body_timeout 10; //客户端请求体读取超时时间  
reset_timedout_connection on; //在客户端停止响应之后,允许服务器关闭连接,释放socket关联的内存  
send_timeout 10; //指定客户端的超时时间,如果在10s内客户端没有任何响应,nginx会自动断开连接
```

8、gzip的优化

```
gzip on; //开启压缩  
gzip_min_length 1000; //小文件不压缩  
gzip_comp_level 6; //压缩比例  
gzip_types text/plain text/css application/json application/x-javascript text/xml  
application/xml application/xml+rss text/javascript; //对指定文件类型进行压缩
```

9、缓存静态页面的优化（文件句柄是打开文件的唯一标示）

```
open_file_cache max=100000 inactive=20s; //设置服务器最大缓存10万个文件句柄, 关闭20s内无请求的句柄  
open_file_cache_valid 30s; //文件句柄的有效期为30s
```

```
open_file_cache_min_uses 2;//最少打开2次才会被缓存
open_file_cache_errors on;
```

注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，请从下面的链接获取：[码云](#) 或者 [语雀](#)

参考文献

限流：计数器、漏桶、令牌桶 三大算法的原理与实战（史上最全）

推荐阅读：

《场景题：假设10W人突访，你的系统如何做到不 雪崩？》

《尼恩Java面试宝典》

《Springcloud gateway 底层原理、核心实战 (史上最全)》

《Flux、Mono、Reactor 实战（史上最全）》

《sentinel （史上最全）》

《Nacos (史上最全)》

《分库分表 Sharding-JDBC 底层原理、核心实战（史上最全）》

《TCP协议详解 (史上最全)》

《clickhouse 超底层原理 + 高可用实操 （史上最全）》

《nacos高可用（图解+秒懂+史上最全）》

《队列之王： Disruptor 原理、架构、源码 一文穿透》

《环形队列、 条带环形队列 Striped-RingBuffer （史上最全）》

《一文搞定： SpringBoot、SLF4j、Log4j、Logback、Netty之间混乱关系（史上最全）

《单例模式（史上最全）