大纲

一、前言

二、Web服务器提供服务的方式

三、多进程、多线程、异步模式的对比

四、Web 服务请求过程

五、Linux I/O 模型

六、Linux I/O 模型具体说明

七、Linux I/O模型的具体实现

八、Apache 的工作模式

九、支持高并发的Web服务器

十、Nginx 详解

一、前言

注，在说Web服务器之前，先说说线程、进程、以及并发连接数。

1.进程与线程

       进程是具有一定独立功能的程序，关于某个数据集合上的一次运行活动，进程是系统进行资源分配和调度的一个独立单位。从逻辑角度来看，多线程的意义在于一个应用程序（进程）中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用来实现，而是作为进程来调度和管理以及资源分配。这就是进程和线程的重要区别，进程和线程的主要差别在于，进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。好了，我想我已经说明白进程与线程了，下面我们来说一说并发连接数。

2.并发连接数

(1).什么是最大并发连接数呢？

所谓，最大并发连接数是服务器同一时间能处理最大会话数量。

(2).何为会话？

我们打开一个网站就是一个客户端浏览器与服务端的一个会话，而我们浏览网页是基于http协议。

(3).HTTP协议如何工作？

HTTP支持两种建立连接的方式：非持久连接和持久连接(HTTP1.1默认的连接方式为持久连接)。

(4).浏览器与Web服务器之间将完成下列7个步骤

* 建立TCP连接
* Web浏览器向Web服务器发送请求命令
* Web浏览器发送请求头信息
* Web服务器应答
* Web服务器发送应答头信息
* Web服务器向浏览器发送数据
* Web服务器关闭TCP连接

       一般情况下，一旦Web服务器向浏览器发送了请求数据，它就要关闭TCP连接，但是浏览器一般其头信息加入了这行代码 Connection:keep-alive，TCP连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接目的，节省了为每 个请求建立新连接所需的时间，还节约了网络带宽。

3.并发连接数的计算方法

* 下载：用户下载服务器上的文件，则为一个连接，用户文件下载完毕后这个连接就消失了。有时候用户用迅雷的多线程方式下载的话，这一个用户开启了5个线程的话，就算是5个连接。
* 用户打开你的页面，就算停留在页面没有对服务器发出任何请求，那么在用户打开一面以后的15分钟内也都要算一个在线。
* 上面的情况用户继续打开同一个网站的其他页面，那么在线人数按照用户最后一次点击（发出请求）以后的15分钟计算，在这个15分钟内不管用户怎么点击（包括新窗口打开）都还是一人在线。
* 当用户打开页面然后正常关闭浏览器，用户的在线人数也会马上清除。

二、Web服务器提供服务的方式

       Web服务器由于要同时为多个客户提供服务，就必须使用某种方式来支持这种多任务的服务方式。一般情况下可以有以下三种方式来选择，多进程方式、多线程方式及异步方式。其中，多进程方式中服务器对一个客户要使用一个进程来提供服务，由于在操作系统中，生成一个进程需要进程内存复制等额外的开销，这样在客户较多时的性能就会降低。为了克服这种生成进程的额外开销，可以使用多线程方式或异步方式。在多线程方式中，使用进程中的多个线程提供服务， 由于线程的开销较小，性能就会提高。事实上，不需要任何额外开销的方式还是异步方式，它使用非阻塞的方式与每个客户通信，服务器使用一个进程进行轮询就行了。

       虽然异步方式最为高效，但它也有自己的缺点。因为异步方式下，多个任务之间的调度是由服务器程序自身来完成的，而且一旦一个地方出现问题则整个服务器就会出现问题。因此，向这种服务器增加功能，一方面要遵从该服务器自身特定的任务调度方式，另一方面要确保代码中没有错误存在，这就限制了服务器的功能，使得异步方式的Web服务器的效率最高，但功能简单，如Nginx服务器。

       由于多线程方式使用线程进行任务调度，这样服务器的开发由于遵从标准，从而变得简单并有利于多人协作。然而多个线程位于同一个进程内，可以访问同样的内存空间，因此存在线程之间的影响，并且申请的内存必须确保申请和释放。对于服务器系统来讲，由于它要数天、数月甚至数年连续不停的运转，一点点错误就会逐渐积累而最终导致影响服务器的正常运转，因此很难编写一个高稳定性的多线程服务器程序。但是，不是不能做到时。Apache的worker模块就能很好的支持多线程的方式。

       多进程方式的优势就在于稳定性，因为一个进程退出的时候，操作系统会回收其占用的资源，从而使它不会留下任何垃圾。即便程序中出现错误，由于进程是相互隔离的，那么这个错误不会积累起来，而是随着这个进程的退出而得到清除。Apache的prefork模块就是支持多进程的模块。

三、多进程、多线程、异步模式的对比

Web服务器总的来说提供服务的方式有三种,

* 多进程方式
* 多线程的方式
* 异步方式

其中效率最高的是异步的方式，最稳定的是多进程方式，占用资源较少的是多线程的方式。

1.多进程

       此种架构方式中，web服务器生成多个进程并行处理多个用户请求，进程可以按需或事先生成。有的web服务器应用程序为每个用户请求生成一个单独的进程来进行响应，不过，一旦并发请求数量达到成千上万时，多个同时运行的进程将会消耗大量的系统资源。（即每个进程只能响应一个请求或多个进程对应多个请求）

优点：

* 最大的优势就在于稳定性，一个进程出错不会影响其它进程。如，服务器同时连接100个请求对就的是100个进程，其中一个进程出错，只会杀死一个进程，还有99个进程继续响应用户请求。
* 每个进程响应一个请求

缺点：

* 进程量大，进程切换次数过多，导致CPU资源使用效率低
* 每个进程的地址空间是独立的，很多空间中重复的数据，所以内存使用效率低
* 进程切换由于内核完成，占用CPU资源

2.多线程

       在多线程方式中，每个线程来响应一下请求，由于线程之间共享进程的数据，所以线程的开销较小，性能就会提高。

优点：

* 线程间共享进程数据
* 每个线程响应一个请求
* 线程切换不可避免（切换量级比较轻量）
* 同一进程的线程可以共享进程的诸多资源，比如打开的文件
* 对内存的需求较之进程有很大下降
* 读可以共享，写不可以共享

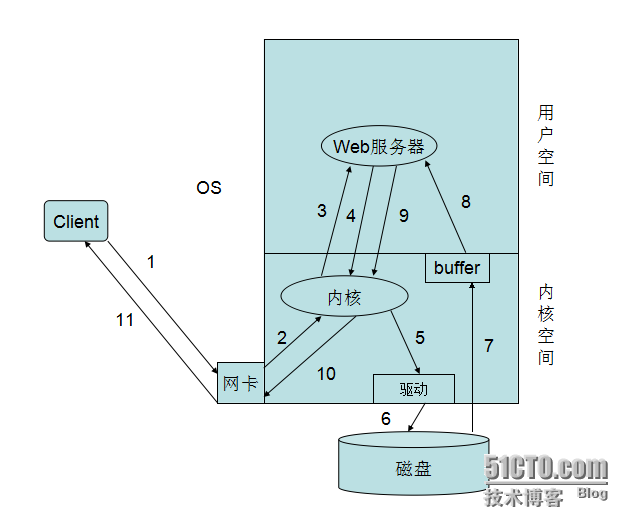
缺点：

* 线程快速切换时会带来线程抖动
* 多线程会导致服务器不稳定

3.异步方式

       一个进程或线程响应多个请求，不需要任何额外开销的，性能最高，占用资源最少。但也有问题一但进程或线程出错就会导致整个服务器的宕机。

四、Web 服务请求过程



在上面的讲解中我们说明，Web服务器的如何提供服务的，有多进程的方式、多线程的方式还有异步方式我们先简单这么理解，大家肯定还有很多疑问，我们先存疑，后面我们慢慢说，现在我们不管Web服务器是如何提供服务的，多进程也好、多线程好，异步也罢。下面我们来说一下，一个客户端的具体请求Web服务的具体过程，从上图中我们可以看到有11步，下面我们来具体说一下，

* 首先我们客户端发送一个请求到Web服务器，请求首先是到网卡。
* 网卡将请求交由内核空间的内核处理，其实就是拆包了，发现请求的是80端口。
* 内核便将请求发给了在用户空间的Web服务器，Web服务器接受到请求发现客户端请求的index.html页面、
* Web服务器便进行系统调用将请求发给内核
* 内核发现在请求的是一页面，便调用磁盘的驱动程序，连接磁盘
* 内核通过驱动调用磁盘取得的页面文件
* 内核将取得的页面文件保存在自己的缓存区域中便通知Web进程或线程来取相应的页面文件
* Web服务器通过系统调用将内核缓存中的页面文件复制到进程缓存区域中
* Web服务器取得页面文件来响应用户，再次通过系统调用将页面文件发给内核
* 内核进程页面文件的封装并通过网卡发送出去
* 当报文到达网卡时通过网络响应给客户端

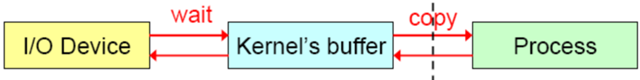
简单来说就是：用户请求-->送达到用户空间-->系统调用-->内核空间-->内核到磁盘上读取网页资源->返回到用户空间->响应给用户。上述简单的说明了一下，客户端向Web服务请求过程，在这个过程中，有两个I/O过程，一个就是客户端请求的网络I/O，另一个就是Web服务器请求页面的磁盘I/O。 下面我们就来说说Linux的I/O模型。

五、Linux I/O 模型

1.I/O模型分类

说明：我们都知道web服务器的进程响应用户请求，但无法直接操作I/O设备，其必须通过系统调用，请求kernel来协助完成I/O动作

内核会为每个I/O设备维护一个buffer，如下图：



对于数据输入而言，即等待(wait)数据输入至buffer需要时间，而从buffer复制(copy)数据至进程也需要时间

根据等待模式不同，I/O动作可分为五种模式。

* 阻塞I/O
* 非阻塞I/O
* I/O复用（select和poll）
* 信号（事件）驱动I/O（SIGIO）
* 异步I/O（Posix.1的aio\_系列函数）

2.I/O模型的相关术语

这里有必要先解释一下阻塞、非阻塞，同步、异步、I/O的概念。

(1).阻塞和非阻塞:

阻塞和非阻塞指的是执行一个操作是等操作结束再返回，还是马上返回。比如你去车站接朋友，这是一个操作。可以有两种执行方式。第一种，你这人特实诚，老早就到了车站一直等到车来了接到朋友为止。第二种，你到了车站，问值班的那趟车来了没有，“还没有”，你出去逛一圈，可能过会回来再问。第一种就是阻塞方式，第二种则是非阻塞的。我认为阻塞和非阻塞讲得是做事方法，是针对做事的人而言的。

(2).同步和异步:

       同步和异步又是另外一个概念，它是事件本身的一个属性。比如老板让你去搬一堆石头，而且只让你一个人干，你只好自己上阵，最后的结果是搬完了，还是你砸到脚了，只有搬完了你才知道。这就是同步的事件。如果老板还给你个小弟，你就可以让小弟去搬，搬完了告你一声。这就变成异步的了。其实异步还可以分为两种：带通知的和不带通知的。前面说的那种属于带通知的。有些小弟干活可能主动性不是很够，不会主动通知你，你就需要时不时的去关注一下状态。这种就是不带通知的异步。

       对于同步的事件，你只能以阻塞的方式去做。而对于异步的事件，阻塞和非阻塞都是可以的。非阻塞又有两种方式：主动查询和被动接收消息。被动不意味着一定不好，在这里它恰恰是效率更高的，因为在主动查询里绝大部分的查询是在做无用功。对于带通知的异步事件，两者皆可。而对于不带通知的，则只能用主动查询。

(3).I/O

       回到I/O，不管是I还是O，对外设(磁盘)的访问都可以分成请求和执行两个阶段。请求就是看外设的状态信息（比如是否准备好了），执行才是真正的I/O操作。在Linux 2.6之前，只有“请求”是异步事件，2.6之后才引入AIO把“执行”异步化。别看Linux/Unix是用来做服务器的，这点上比Windows落后了好多，IOCP（Windows上的AIO）在Win2000上就有了,呵呵。

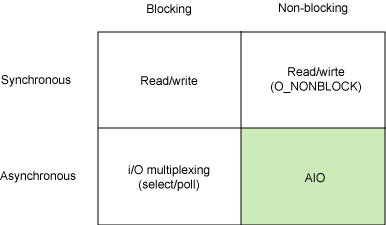
(4).总结

       Linux上的前四种I/O模型的“执行”阶段都是同步的，只有最后一种才做到了真正的全异步。第一种阻塞式是最原始的方法，也是最累的办法。当然累与不累要看针对谁。应用程序是和内核打交道的。对应用程序来说，这种方式是最累的，但对内核来说这种方式恰恰是最省事的。还拿接人这事为例，你就是应用程序，值班员就是内核，如果你去了一直等着，值班员就省事了。当然现在计算机的设计，包括操作系统，越来越为终端用户考虑了，为了让用户满意，内核慢慢的承担起越来越多的工作，IO模型的演化也是如此。

       非阻塞I/O ，I/O复用，信号驱动式I/O其实都是非阻塞的，当然是针对“请求”这个阶段。非阻塞式是主动查询外设状态。I/O复用里的select，poll也是主动查询，不同的是select和poll可以同时查询多个fd（文件句柄）的状态，另外select有fd个数的限制。epoll是基于回调函数的。信号驱动式I/O则是基于信号消息的。这两个应该可以归到“被动接收消息”那一类中。最后就是伟大的AIO的出现，内核把什么事都干了，对上层应用实现了全异步，性能最好，当然复杂度也最高。好了，下面我们就来详细说一说，这几种模式。

六、Linux I/O 模型具体说明

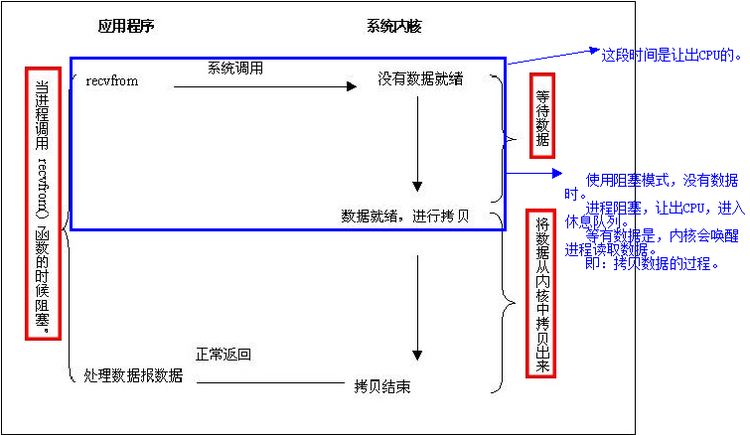
首先我们先来看一下，基本 Linux I/O 模型的简单矩阵，



从图中我们可以看到的模型有，同步阻塞I/O（阻塞I/O）、同步非阻塞I/O（非阻塞I/O ）、异步阻塞I/O（I/O复用），异步非阻塞I/O（有两种，信号驱动I/O和异步I/O）。好了现在就来具体说一说吧。

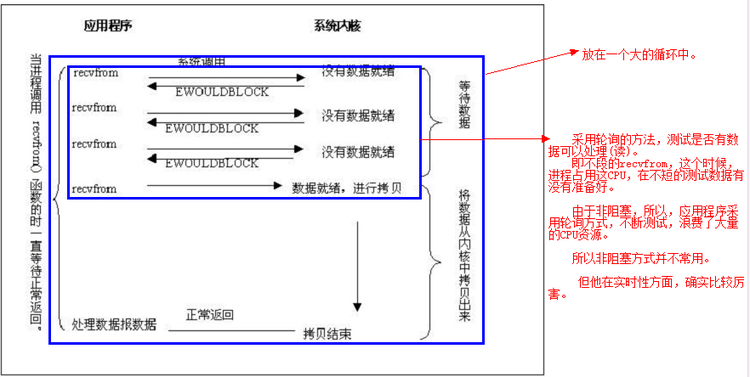
1.阻塞I/O

说明：应用程序调用一个IO函数，导致应用程序阻塞，等待数据准备好。 如果数据没有准备好，一直等待数据准备好了，从内核拷贝到用户空间,IO函数返回成功指示。这个不用多解释吧，阻塞套接字。下图是它调用过程的图示：（注，一般网络I/O都是阻塞I/O，客户端发出请求，Web服务器进程响应，在进程没有返回页面之前，这个请求会处于一直等待状态）



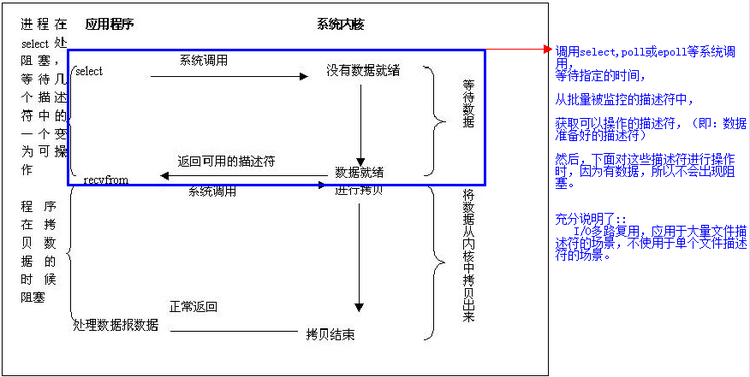
2.非阻塞I/O

我们把一个套接口设置为非阻塞就是告诉内核，当所请求的I/O操作无法完成时，不要将进程睡眠，而是返回一个错误。这样我们的I/O操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用CPU的时间，所有一般Web服务器都不使用这种I/O模型。具体过程如下图:



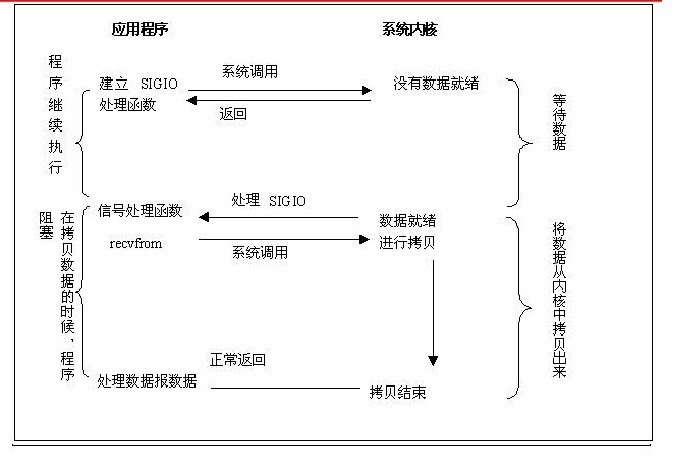
3.I/O复用（select和poll）

I/O复用模型会用到select或poll函数或epoll函数(Linux2.6以后的内核开始支持)，这两个函数也会使进程阻塞，但是和阻塞I/O所不同的的，这两个函数可以同时阻塞多个I/O操作。而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数。具体过程如下图:



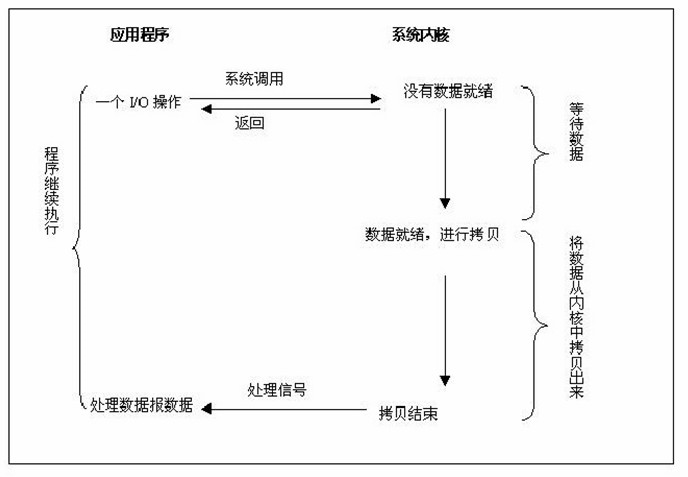
4.信号驱动I/O（SIGIO）

首先，我们允许套接口进行信号驱动I/O，并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据。具体过程如下图:

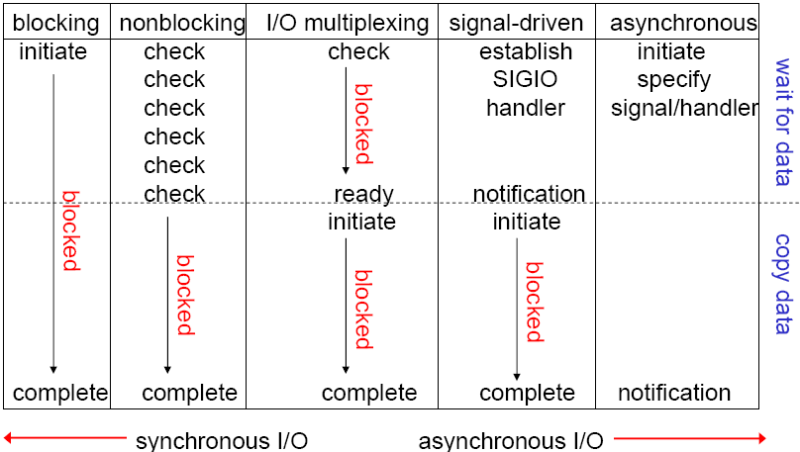


5.异步I/O（Posix.1的aio\_系列函数）

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作。具体过程如下图:



6.I/O 模型总结（如下图）



 0987654

从上图中我们可以看出，可以看出，越往后，阻塞越少，理论上效率也是最优。其五种I/O模型中，前三种属于同步I/O，后两者属于异步I/O。

同步I/O:

* 阻塞I/O
* 非阻塞I/O
* I/O复用（select和poll）

异步I/O:

* 信号驱动I/O（SIGIO） （半异步）
* 异步I/O（Posix.1的aio\_系列函数） （真正的异步）

异步 I/O 和 信号驱动I/O的区别:

* 信号驱动 I/O 模式下，内核可以复制的时候通知给我们的应用程序发送SIGIO 消息。
* 异步 I/O 模式下，内核在所有的操作都已经被内核操作结束之后才会通知我们的应用程序。

好了，5种模型的比较比较清晰了，下面我们来说一下五种模型的具体实现。

七、Linux I/O模型的具体实现

1.主要实现方式有以下几种：

* select
* poll
* epoll
* kqueue
* /dev/poll
* iocp

注，其中iocp是Windows实现的，select、poll、epoll是Linux实现的，kqueue是FreeBSD实现的，/dev/poll是SUN的Solaris实现的。select、poll对应第3种（I/O复用）模型，iocp对应第5种（异步I/O）模型，那么epoll、kqueue、/dev/poll呢？其实也同select属于同一种模型，只是更高级一些，可以看作有了第4种（信号驱动I/O）模型的某些特性，如callback机制。

2.为什么epoll、kqueue、/dev/poll比select高级？

       答案是，他们无轮询。因为他们用callback取代了。想想看，当套接字比较多的时候，每次select()都要通过遍历FD\_SETSIZE个Socket来完成调度，不管哪个Socket是活跃的，都遍历一遍。这会浪费很多CPU时间。如果能给套接字注册某个回调函数，当他们活跃时，自动完成相关操作，那就避免了轮询，这正是epoll、kqueue、/dev/poll做的。这样子说可能不好理解，那么我说一个现实中的例子，假设你在大学读书，住的宿舍楼有很多间房间，你的朋友要来找你。select版宿管大妈就会带着你的朋友挨个房间去找，直到找到你为止。而epoll版宿管大妈会先记下每位同学的房间号，你的朋友来时，只需告诉你的朋友你住在哪个房间即可，不用亲自带着你的朋友满大楼找人。如果来了10000个人，都要找自己住这栋楼的同学时，select版和epoll版宿管大妈，谁的效率更高，不言自明。同理，在高并发服务器中，轮询I/O是最耗时间的操作之一，select、epoll、/dev/poll的性能谁的性能更高，同样十分明了。

3.Windows or \*nix （IOCP or kqueue、epoll、/dev/poll）？

       诚然，Windows的IOCP非常出色，目前很少有支持asynchronous I/O的系统，但是由于其系统本身的局限性，大型服务器还是在UNIX下。而且正如上面所述，kqueue、epoll、/dev/poll 与 IOCP相比，就是多了一层从内核copy数据到应用层的阻塞，从而不能算作asynchronous I/O类。但是，这层小小的阻塞无足轻重，kqueue、epoll、/dev/poll 已经做得很优秀了。

4.总结一些重点

* 只有IOCP(windows实现)是asynchronous I/O，其他机制或多或少都会有一点阻塞。
* select（Linux实现）低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善
* epoll(Linux实现)、kqueue（FreeBSD实现）、/dev/poll（Solaris实现）是Reacor模式，IOCP是Proactor模式。
* Apache 2.2.9之前只支持select模型，2.2.9之后支持epoll模型
* Nginx 支持epoll模型
* Java nio包是select模型

八、Apache 的工作模式

1.apache三种工作模式

我们都知道Apache有三种工作模块，分别为prefork、worker、event。

* prefork：多进程，每个请求用一个进程响应，这个过程会用到select机制来通知。
* worker：多线程，一个进程可以生成多个线程，每个线程响应一个请求，但通知机制还是select不过可以接受更多的请求。
* event：基于异步I/O模型，一个进程或线程，每个进程或线程响应多个用户请求，它是基于事件驱动（也就是epoll机制）实现的。

2.prefork的工作原理

       如果不用“--with-mpm”显式指定某种MPM,prefork就是Unix平台上缺省的MPM.它所采用的预派生子进程方式也是 Apache1.3中采用的模式.prefork本身并没有使用到线程,2.0版使用它是为了与1.3版保持兼容性;另一方面,prefork用单独的子 进程来处理不同的请求,进程之间是彼此独立的,这也使其成为最稳定的MPM之一。

3.worker的工作原理

       相对于prefork,worker是2.0版中全新的支持多线程和多进程混合模型的MPM.由于使用线程来处理,所以可以处理相对海量的请求,而 系统资源的开销要小于基于进程的服务器.但是,worker也使用了多进程,每个进程又生成多个线程,以获得基于进程服务器的稳定性.这种MPM的工作方 式将是Apache2.0的发展趋势。

4.event 基于事件机制的特性

       一个进程响应多个用户请求，利用callback机制，让套接字复用，请求过来后进程并不处理请求，而是直接交由其他机制来处理，通过epoll机制来通知请求是否完成；在这个过程中，进程本身一直处于空闲状态，可以一直接收用户请求。可以实现一个进程程响应多个用户请求。支持持海量并发连接数，消耗更少的资源。

九、支持高并发的Web服务器

有几个基本条件：

1.基于线程，即一个进程生成多个线程，每个线程响应用户的每个请求。

2.基于事件的模型，一个进程处理多个请求，并且通过epoll机制来通知用户请求完成。

3.基于磁盘的AIO（异步I/O）

4.支持mmap内存映射，mmap传统的web服务器，进行页面输入时，都是将磁盘的页面先输入到内核缓存中，再由内核缓存中复制一份到web服务器上，mmap机制就是让内核缓存与磁盘进行映射，web服务器，直接复制页面内容即可。不需要先把磁盘的上的页面先输入到内核缓存去。

刚好，Nginx 支持以上所有特性。所以Nginx官网上说，Nginx支持50000并发，是有依据的。好了，基础知识就说到这边下面我们来谈谈我们今天讲解的重点Nginx。

十、Nginx 详解

1.简介

       传统上基于进程或线程模型架构的web服务通过每进程或每线程处理并发连接请求，这势必会在网络和I/O操作时产生阻塞，其另一个必然结果则是对内存或CPU的利用率低下。生成一个新的进程/线程需要事先备好其运行时环境，这包括为其分配堆内存和栈内存，以及为其创建新的执行上下文等。这些操作都需要占用CPU，而且过多的进程/线程还会带来线程抖动或频繁的上下文切换，系统性能也会由此进一步下降。另一种高性能web服务器/web服务器反向代理：Nginx（Engine X），nginx的主要着眼点就是其高性能以及对物理计算资源的高密度利用，因此其采用了不同的架构模型。受启发于多种操作系统设计中基于“事件”的高级处理机制，nginx采用了模块化、事件驱动、异步、单线程及非阻塞的架构，并大量采用了多路复用及事件通知机制。在nginx中，连接请求由为数不多的几个仅包含一个线程的进程worker以高效的回环(run-loop)机制进行处理，而每个worker可以并行处理数千个的并发连接及请求。

2.Nginx 工作原理

       Nginx会按需同时运行多个进程：一个主进程(master)和几个工作进程(worker)，配置了缓存时还会有缓存加载器进程(cache loader)和缓存管理器进程(cache manager)等。所有进程均是仅含有一个线程，并主要通过“共享内存”的机制实现进程间通信。主进程以root用户身份运行，而worker、cache loader和cache manager均应以非特权用户身份运行。

主进程主要完成如下工作：

* 读取并验正配置信息；
* 创建、绑定及关闭套接字；
* 启动、终止及维护worker进程的个数；
* 无须中止服务而重新配置工作特性；
* 控制非中断式程序升级，启用新的二进制程序并在需要时回滚至老版本；
* 重新打开日志文件；
* 编译嵌入式perl脚本；

worker进程主要完成的任务包括：

* 接收、传入并处理来自客户端的连接；
* 提供反向代理及过滤功能；
* nginx任何能完成的其它任务；

注，如果负载以CPU密集型应用为主，如SSL或压缩应用，则worker数应与CPU数相同；如果负载以IO密集型为主，如响应大量内容给客户端，则worker数应该为CPU个数的1.5或2倍。

3.Nginx 架构

       Nginx的代码是由一个核心和一系列的模块组成, 核心主要用于提供Web Server的基本功能，以及Web和Mail反向代理的功能；还用于启用网络协议，创建必要的运行时环境以及确保不同的模块之间平滑地进行交互。不过，大多跟协议相关的功能和某应用特有的功能都是由nginx的模块实现的。这些功能模块大致可以分为事件模块、阶段性处理器、输出过滤器、变量处理器、协议、upstream和负载均衡几个类别，这些共同组成了nginx的http功能。事件模块主要用于提供OS独立的(不同操作系统的事件机制有所不同)事件通知机制如kqueue或epoll等。协议模块则负责实现nginx通过http、tls/ssl、smtp、pop3以及imap与对应的客户端建立会话。在Nginx内部，进程间的通信是通过模块的pipeline或chain实现的；换句话说，每一个功能或操作都由一个模块来实现。例如，压缩、通过FastCGI或uwsgi协议与upstream服务器通信，以及与memcached建立会话等。

4.Nginx 基础功能

* 处理静态文件，索引文件以及自动索引；
* 反向代理加速(无缓存)，简单的负载均衡和容错；
* FastCGI，简单的负载均衡和容错；
* 模块化的结构。过滤器包括gzipping, byte ranges, chunked responses, 以及 SSI-filter 。在SSI过滤器中，到同一个 proxy 或者 FastCGI 的多个子请求并发处理；
* SSL 和 TLS SNI 支持；

5.Nginx IMAP/POP3 代理服务功能

* 使用外部 HTTP 认证服务器重定向用户到 IMAP/POP3 后端；
* 使用外部 HTTP 认证服务器认证用户后连接重定向到内部的 SMTP 后端；
* 认证方法：
* POP3: POP3 USER/PASS, APOP, AUTH LOGIN PLAIN CRAM-MD5;
* IMAP: IMAP LOGIN;
* SMTP: AUTH LOGIN PLAIN CRAM-MD5;
* SSL 支持；
* 在 IMAP 和 POP3 模式下的 STARTTLS 和 STLS 支持；

6.Nginx 支持的操作系统

* FreeBSD 3.x, 4.x, 5.x, 6.x i386; FreeBSD 5.x, 6.x amd64;
* Linux 2.2, 2.4, 2.6 i386; Linux 2.6 amd64;
* Solaris 8 i386; Solaris 9 i386 and sun4u; Solaris 10 i386;
* MacOS X (10.4) PPC;
* Windows 编译版本支持 windows 系列操作系统;

7.Nginx 结构与扩展

* 一个主进程和多个工作进程，工作进程运行于非特权用户；
* kqueue (FreeBSD 4.1+), epoll (Linux 2.6+), rt signals (Linux 2.2.19+), /dev/poll (Solaris 7 11/99+), select, 以及 poll 支持；
* kqueue支持的不同功能包括 EV\_CLEAR, EV\_DISABLE （临时禁止事件）， NOTE\_LOWAT, EV\_EOF, 有效数据的数目，错误代码；
* sendfile (FreeBSD 3.1+), sendfile (Linux 2.2+), sendfile64 (Linux 2.4.21+), 和 sendfilev (Solaris 8 7/01+) 支持；
* 输入过滤 (FreeBSD 4.1+) 以及 TCP\_DEFER\_ACCEPT (Linux 2.4+) 支持；
* 10,000 非活动的 HTTP keep-alive 连接仅需要 2.5M 内存。
* 最小化的数据拷贝操作；

8.Nginx 其他HTTP功能

* 基于IP 和名称的虚拟主机服务；
* Memcached 的 GET 接口；
* 支持 keep-alive 和管道连接；
* 灵活简单的配置；
* 重新配置和在线升级而无须中断客户的工作进程；
* 可定制的访问日志，日志写入缓存，以及快捷的日志回卷；
* 4xx-5xx 错误代码重定向；
* 基于 PCRE 的 rewrite 重写模块；
* 基于客户端 IP 地址和 HTTP 基本认证的访问控制；
* PUT, DELETE, 和 MKCOL 方法；
* 支持 FLV （Flash 视频）；
* 带宽限制；

9.为什么选择Nginx

* 在高连接并发的情况下，Nginx是Apache服务器不错的替代品: Nginx在美国是做虚拟主机生意的老板们经常选择的软件平台之一. 能够支持高达 50,000 个并发连接数的响应, 感谢Nginx为我们选择了 [epoll and kqueue](http://www.kegel.com/c10k.html) 作为开发模型。
* Nginx作为负载均衡服务器: Nginx 既可以在内部直接支持 [Rails](http://wiki.codemongers.com/NginxRubyonRailsMongrel) 和 [PHP](http://wiki.codemongers.com/NginxFcgiExample) 程序对外进行服务, 也可以支持作为 [HTTP代理](http://wiki.codemongers.com/NginxHttpProxyModule) 服务器对外进行服务. Nginx采用C进行编写, 不论是系统资源开销还是CPU使用效率都比 [Perlbal](http://www.danga.com/perlbal/) 要好很多。
* 作为邮件代理服务器: Nginx 同时也是一个非常优秀的邮件代理服务器（最早开发这个产品的目的之一也是作为邮件代理服务器）, [Last.fm](http://blog.fastmail.fm/2007/01/04/webimappop-frontend-proxies-changed-to-nginx/) 描述了成功并且美妙的使用经验.
* Nginx 是一个 [#installation 安装] 非常的简单 , [配置文件](http://wiki.nginx.org/NginxConfiguration) 非常简洁（还能够支持perl语法）,[Bugs](http://nginx.net/CHANGES-0.5) 非常少的服务器: 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. 你还能够 [不间断服务的情况下进行软件版本的升级](http://wiki.nginx.org/NginxChsCommandLine#utnbotf) 。
* Nginx 的诞生主要解决[C10K](http://blog.csdn.net/typename/article/details/5598145)问题

好了，到这里Nginx的理论部分就说到这了，下一篇博文中我们将详细说明，Nginx的安装与应用。希望大家有所收获……