web一般请求流程

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服务器解包发现客户端请求的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

select，poll是**主动查询**，它们可以同时查询多个fd（文件句柄）的状态，另外select有fd个数的限制，poll没有限制。select和poll不同的是，他们创建的事件描述符不同，select创建读、写、异常三个集合，而poll在一个集合内设定三种描述，由于select和poll每个循环都会检查事件的发生，而poll的事件比较少，性能上比select要好一些；

epoll是基于**回调函数**的，无轮询。如果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(Linux)、kqueue(FreeBSD)、/dev/poll(soloris)做的。举个经典例子，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。

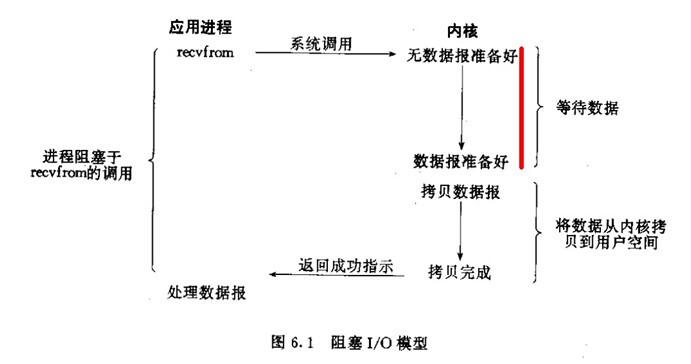
Nginx vs apache:

apache:对于每一个连接，apache都是在一个进程内处理完毕。请求过来，建立新的线程处理，i/o阻塞。线程在内核等待，切换到其他线程。

nginx： Nginx采用epoll模型，异步非阻塞。对于Nginx来说，把一个完整的连接请求处理都划分成了事件，一个一个的事件。比如accept（）， recv（），磁盘I/O，send（）等，每部分都有相应的模块去处理，一个完整的请求可能是由几百个模块去处理。真正核心的就是事件收集和分发模块，这就是管理所有模块的核心。只有核心模块的调度才能让对应的模块占用CPU资源，从而处理请求。拿一个HTTP请求来说，首先在事件收集分发模块注册感兴趣的监听事件，注册好之后不阻塞直接返回，接下来就不需要再管了，等待有连接来了内核会通知你(epoll的轮询会告诉进程)，cpu就可以处理其他事情去了。一旦有请求来，那么对整个请求分配相应的上下文（其实已经预先分配好），这时候再注册新的感兴趣的事件(read函数)，同样客户端数据来了内核会自动通知进程可以去读数据了，读了数据之后就是解析，解析完后去磁盘找资源（I/O），一旦I/O完成会通知进程，进程开始给客户端发回数据send()，这时候也不是阻塞的，调用后就等内核发回通知发送的结果就行。整个下来把一个请求分成了很多个阶段，每个阶段都到很多模块去注册，然后处理，都是异步非阻塞。异步这里指的就是做一个事情，不需要等返回结果，做好了会自动通知你。

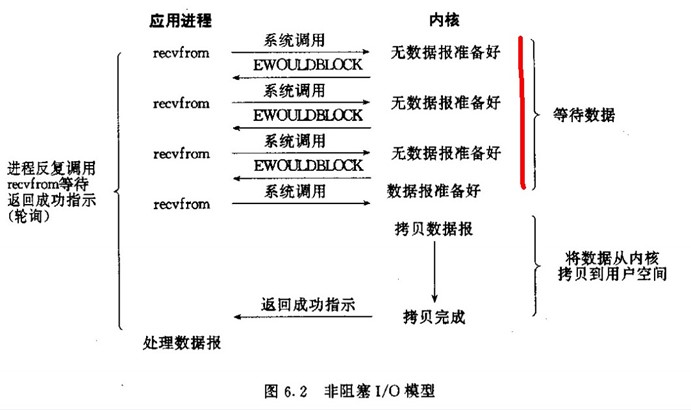
阻塞：默认情况下，所有套接字都是阻塞的。进程挂起，内核等待外部IO响应，IO完成传送数据到kernel buffer，数据再从buffer复制到用户的进程空间。

进程在readbuffer时被阻塞。进程被挂起，CPU切换进程。



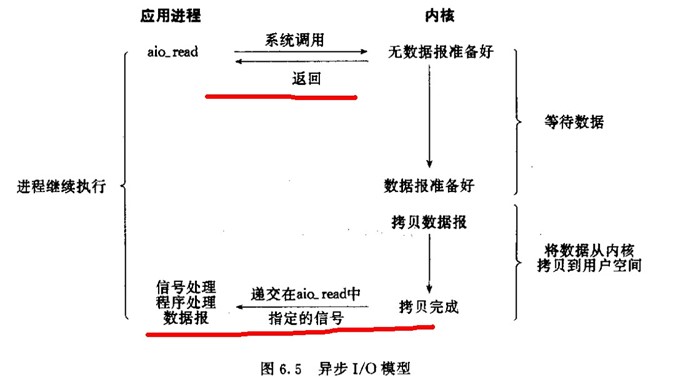
非阻塞：事件未准备好，处理其他部分。

非阻塞轮询：进程没有挂起一直询问事件是否ready。



异步非阻塞：在整个操作（包括将数据从内核拷贝到用户空间）完成后才通知用户进程

可见nginx 实现



select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。

**select/epoll的特点**

select的特点：select 选择句柄的时候，是遍历所有句柄，也就是说句柄有事件响应时，select需要遍历所有句柄才能获取到哪些句柄有事件通知，因此效率是非常低。但是如果连接很少的情况下， select和epoll的LT触发模式相比， 性能上差别不大。  
这里要多说一句，select支持的句柄数是有限制的， 同时只支持1024个，这个是句柄集合限制的，如果超过这个限制，很可能导致溢出，而且非常不容易发现问题， 当然可以通过修改linux的socket内核调整这个参数。  
epoll的特点：epoll对于句柄事件的选择不是遍历的，是事件响应的，就是句柄上事件来就马上选择出来，不需要遍历整个句柄链表，因此效率非常高，内核将句柄用红黑树保存的。  
对于epoll而言还有ET和LT的区别，LT表示水平触发，ET表示边缘触发，两者在性能以及代码实现上差别也是非常大的。

 Epoll模型主要负责对大量并发用户的请求进行及时处理，完成服务器与客户端的数据交互。其具体的实现步骤如下：  
(a) 使用epoll\_create()函数创建文件描述，设定将可管理的最大socket描述符数目。  
(b) 创建与epoll关联的接收线程，应用程序可以创建多个接收线程来处理epoll上的读通知事件，线程的数量依赖于程序的具体需要。  
(c) 创建一个侦听socket描述符ListenSock；将该描述符设定为非阻塞模式，调用Listen（）函数在套接字上侦听有无新的连接请求，在 epoll\_event结构中设置要处理的事件类型EPOLLIN，工作方式为 epoll\_ET，以提高工作效率，同时使用epoll\_ctl()注册事件，最后启动网络监视线程。  
(d) 网络监视线程启动循环，epoll\_wait()等待epoll事件发生。  
(e) 如果epoll事件表明有新的连接请求，则调用accept（）函数，将用户socket描述符添加到epoll\_data联合体，同时设定该描述符为非 阻塞，并在epoll\_event结构中设置要处理的事件类型为读和写，工作方式为epoll\_ET.  
(f) 如果epoll事件表明socket描述符上有数据可读，则将该socket描述符加入可读队列，通知接收线程读入数据，并将接收到的数据放入到接收数据 的链表中，经逻辑处理后，将反馈的数据包放入到发送数据链表中，等待由发送线程发送。

epoll的操作就这么简单，总共不过4个 API：epoll\_create, epoll\_ctl, epoll\_wait和close。