1、网络编程介绍

2、io模型

3、并发网络方案

4、nginx的方案

1、网络编程介绍

网络编程的本质是两个设备之间的数据交换，当然，在计算机网络中，设备主要指计算机。数据传递本身没有多大的难度，不就是把一个设备中的数据发送给两外一个设备，然后接受另外一个设备反馈的数据。

　　现在的网络编程基本上都是基于请求/响应方式的，也就是一个设备发送请求数据给另外一个，然后接收另一个设备的反馈。





socket bind accept connect close shutdown介绍

服务端tcp连接的程序：

listenfd = socekt(...);

listenfd(listenfd);

for(;;){

connfd = accept(listenfd,...);

dosomething();

close(connfd);

}

多进程

listenfd = socekt(...);

listenfd(listenfd);

for(;;){

connfd = accept(listenfd,...);

if( (pid = fork()) == 0){

dosomething();

exit(0);

}

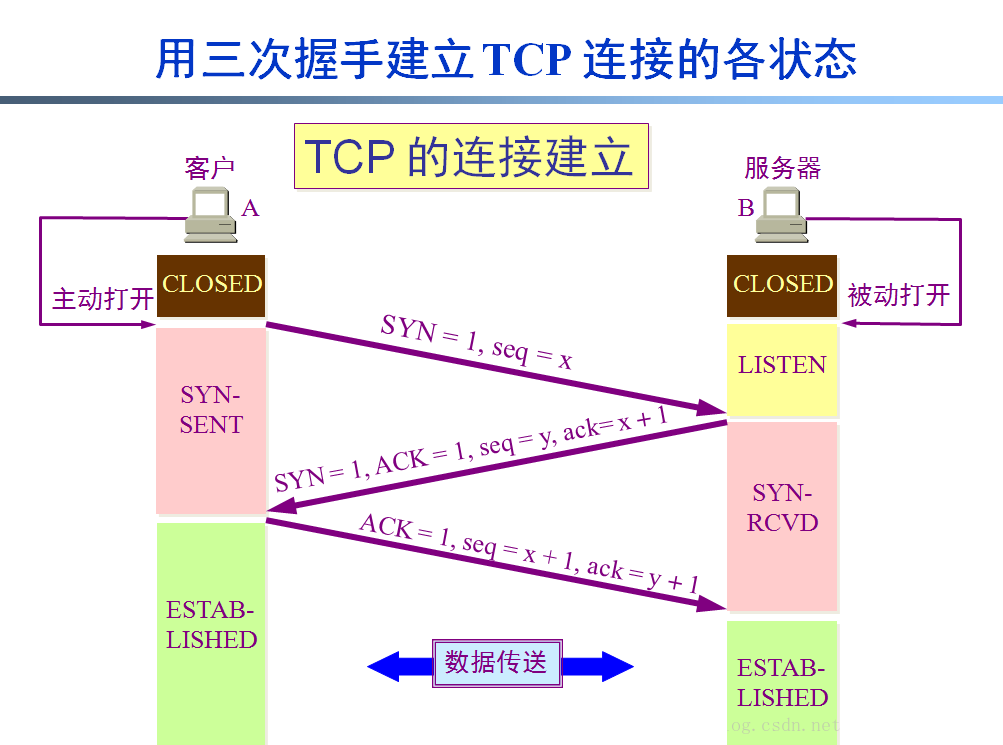
close(connfd);

}

网络编程的一些小问题：

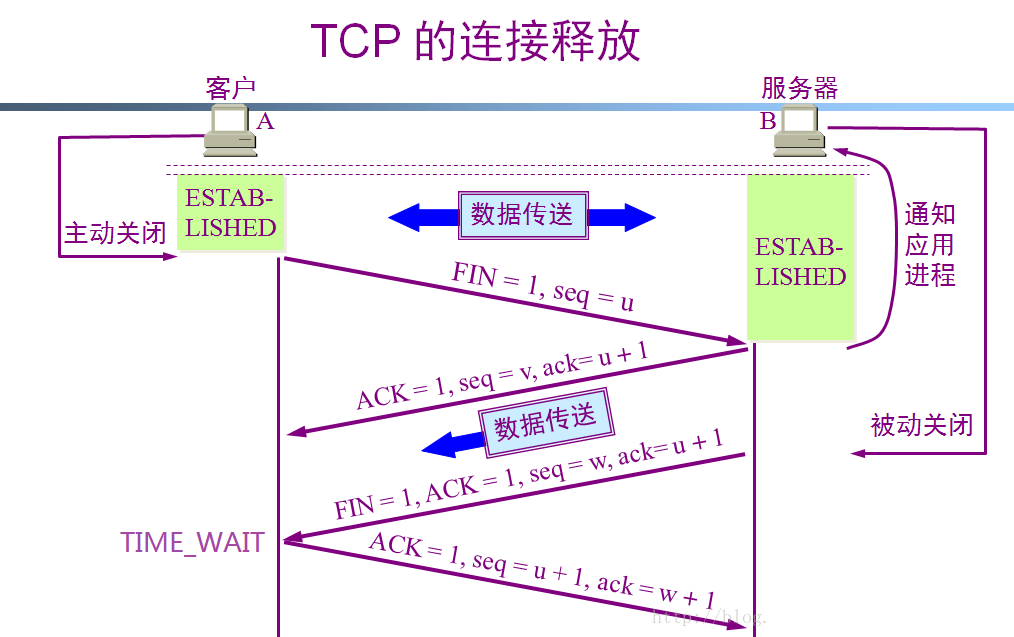
1、accept返回前连接中止

在三路握手完成从而连接建立之后，客户tcp发送一个RST（复位）分节，服务器处于繁忙状态，在收到RST后才调用accept。



捕获异常，再次调用accept

2、服务器进程中止



服务器进程中止时，socket描述符都被关闭，这导致向客户端发送一个FIN，客户端响应一个ACK，TCP处于终止工作的前半部分，如果客户端处于阻塞状态，应用层不能感知服务端关闭，则不能及时关闭连接。如果客户端向服务端发送数据，客户端将收到一个RST，客户端将被终止，

使用select/poll

3、服务器主机崩溃

主机崩溃时，不能发送FIN，客户TCP将不断重传数据分节，进程将被阻塞，一定时间后才会返回ETIMEOUT（超时）或目的地不可达错误。

可以对socket设置一个超时，或设置socket的SO\_KEEPALIVE选项，使得客户端不主动发数据也能检测服务器崩溃。

4、服务器主机崩溃后重启

当服务器主机崩溃后重启时，它的TCP丢失了崩溃前的所有连接信息，因此服务器TCP对于收到来自客户的数据分节响应一个RST。

5、服务器关机

服务器关机时，所有打开的描述符都被关闭，和服务器进程中止的情况一样。在客户中使用select/poll，使得服务器进程的终止一经发生，客户就能检测到。

listenfd = socekt(...);

listenfd(listenfd);

for(;;){

connfd = accept(listenfd,...);

if( (pid = fork()) == 0){

read();

write();

exit(0);

}

close(connfd);

}

在linux中，默认情况下所有的套接字都是阻塞的

Unix下的5种I/O模型

1、阻塞I/O

2、非阻塞I/O

3、I/O复用

4、信号驱动式I/O

5、异步I/O

IO发生时涉及的对象和步骤。对于一个network IO (这里我们以read举例)，它会涉及到两个系统对象，一个是调用这个IO的进程 (或线程)，另一个就是系统内核。当一个read操作发生时，它会经历两个阶段：  
    1）等待数据准备   
    2）将数据从内核拷贝到进程中  
   这些IO模型的区别就是在两个阶段上各有不同的情况。

1、阻塞I/O

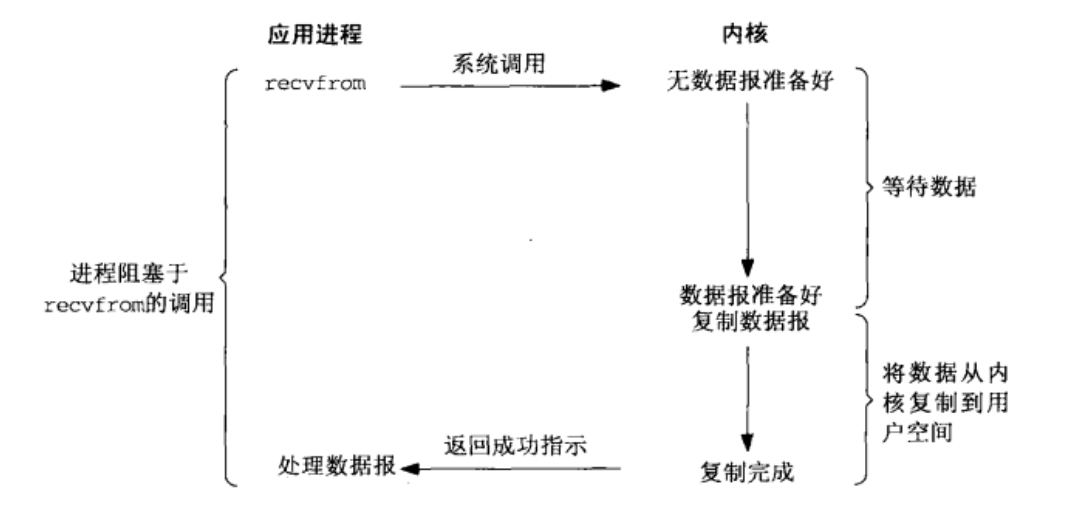


图1 阻塞IO

    当用户进程调用了recvfrom这个系统调用，内核就开始了IO的第一个阶段：准备数据。对于network io来说，很多时候数据在一开始还没有到达（比如，还没有收到一个完整的TCP分节），这个时候内核就要等待足够的数据到来。而在用户进程这边，整个进程会被阻塞。当内核一直等到数据准备好了，它就会将数据从内核中拷贝到用户内存，然后内核返回结果，用户进程才解除阻塞的状态，重新运行起来。  
   **所以，blocking IO的特点就是在IO执行的两个阶段（等待数据和拷贝数据两个阶段）都被阻塞了。**

   网络编程中的listen()、send()、recv() 等接口都是阻塞型的。使用这些接口可以很方便的构建服务器/客户机的模型。下面是一个简单地“一问一答”的服务器。

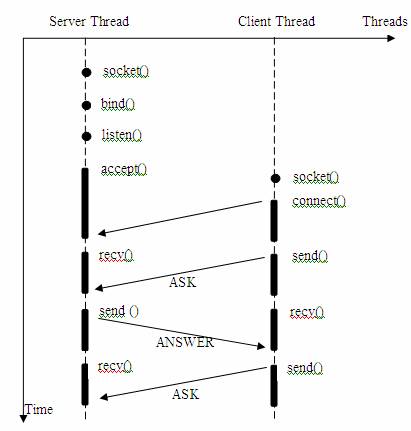


图2 简单的一问一答的服务器/客户机模型

    我们注意到，大部分的socket接口都是阻塞型的。所谓阻塞型接口是指系统调用（一般是IO接口）不返回调用结果并让当前线程一直阻塞，只有当该系统调用获得结果或者超时出错时才返回。  
    实际上，除非特别指定，几乎所有的IO接口 ( 包括socket接口 ) 都是阻塞型的。这给网络编程带来了一个很大的问题，如在调用send()的同时，线程将被阻塞，在此期间，线程将无法执行任何运算或响应任何的网络请求。

    一个简单的改进方案是在服务器端使用多线程（或多进程）。多线程（或多进程）的目的是让每个连接都拥有独立的线程（或进程），这样任何一个连接的阻塞都不会影响其他的连接。具体使用多进程还是多线程，并没有一个特定的模式。**传统意义上，进程的开销要远远大于线程，所以如果需要同时为较多的客户机提供服务，则不推荐使用多进程；如果单个服务执行体需要消耗较多的CPU资源，譬如需要进行大规模或长时间的数据运算或文件访问，则进程较为安全。**通常，使用pthread\_create ()创建新线程，fork()创建新进程。  
    我们假设对上述的服务器 / 客户机模型，提出更高的要求，即让服务器同时为多个客户机提供一问一答的服务。于是有了如下的模型。

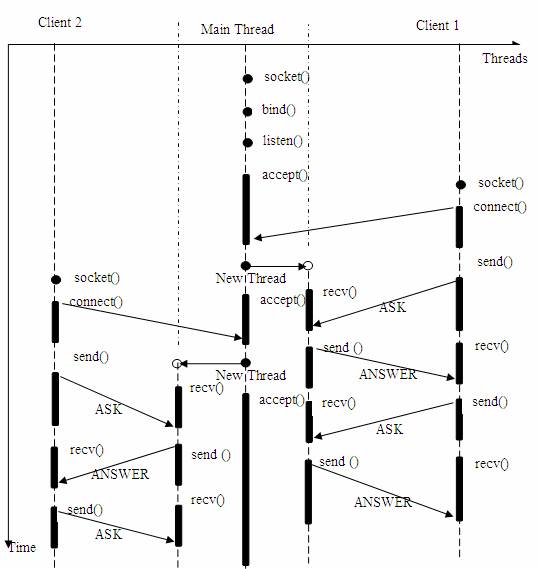


图3 多线程的服务器模型

在上述的线程 / 时间图例中，主线程持续等待客户端的连接请求，如果有连接，则创建新线程，并在新线程中提供为前例同样的问答服务。  
    很多初学者可能不明白为何一个socket可以accept多次。实际上socket的设计者可能特意为多客户机的情况留下了伏笔，让accept()能够返回一个新的socket。下面是 accept 接口的原型：  
     int accept(int s, struct sockaddr \*addr, socklen\_t \*addrlen);   
    输入参数s是从socket()，bind()和listen()中沿用下来的socket句柄值。执行完bind()和listen()后，[操作系统](http://lib.csdn.net/base/operatingsystem)已经开始在指定的端口处监听所有的连接请求，如果有请求，则将该连接请求加入请求队列。调用accept()接口正是从 socket s 的请求队列抽取第一个连接信息，创建一个与s同类的新的socket返回句柄。新的socket句柄即是后续read()和recv()的输入参数。如果请求队列当前没有请求，则accept() 将进入阻塞状态直到有请求进入队列。  
    上述多线程的服务器模型似乎完美的解决了为多个客户机提供问答服务的要求，但其实并不尽然。如果要同时响应成百上千路的连接请求，则无论多线程还是多进程都会严重占据系统资源，降低系统对外界响应效率，而线程与进程本身也更容易进入假死状态。  
    很多程序员可能会考虑使用**“线程池”或“连接池”**。“线程池”旨在减少创建和销毁线程的频率，其维持一定合理数量的线程，并让空闲的线程重新承担新的执行任务。“连接池”维持连接的缓存池，尽量重用已有的连接、减少创建和关闭连接的频率。这两种技术都可以很好的降低系统开销，都被广泛应用很多大型系统，如websphere、tomcat和各种[数据库](http://lib.csdn.net/base/mysql)等。但是，“线程池”和“连接池”技术也只是在一定程度上缓解了频繁调用IO接口带来的资源占用。而且，**所谓“池”始终有其上限，当请求大大超过上限时，“池”构成的系统对外界的响应并不比没有池的时候效果好多少。所以使用“池”必须考虑其面临的响应规模，并根据响应规模调整“池”的大小。**  
    对应上例中的所面临的可能同时出现的上千甚至上万次的客户端请求，“线程池”或“连接池”或许可以缓解部分压力，但是不能解决所有问题。总之，多线程模型可以方便高效的解决小规模的服务请求，但面对大规模的服务请求，多线程模型也会遇到瓶颈，可以用非阻塞接口来尝试解决这个问题。

**2、非阻塞IO**  
    可以通过设置socket使其变为非阻塞。当对一个非阻塞 socket执行读操作时，流程是这个样子：

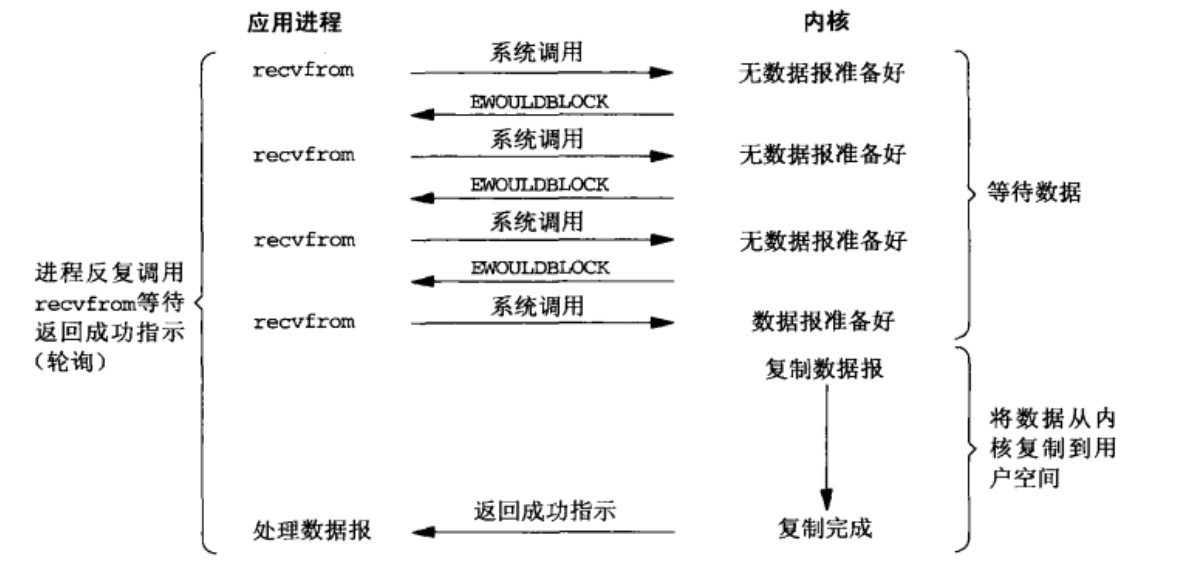


图4 非阻塞IO

    从图中可以看出，当用户进程发出read操作时，如果内核中的数据还没有准备好，那么它并不会阻塞用户进程，而是立刻返回一个error。从用户进程角度讲 ，它发起一个read操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作。一旦内核中的数据准备好了，并且又再次收到了用户进程的system call，那么它马上就将数据拷贝到了用户内存，然后返回。  
    **所以，在非阻塞式IO中，用户进程其实是需要不断的主动询问内核数据准备好了没有。**

    非阻塞的接口相比于阻塞型接口的显著差异在于，在被调用之后立即返回。使用如下的函数可以将某句柄fd设为非阻塞状态。  
    fcntl( fd, F\_SETFL, O\_NONBLOCK );   
    下面将给出只用一个线程，但能够同时从多个连接中检测数据是否送达，并且接受数据的模型。

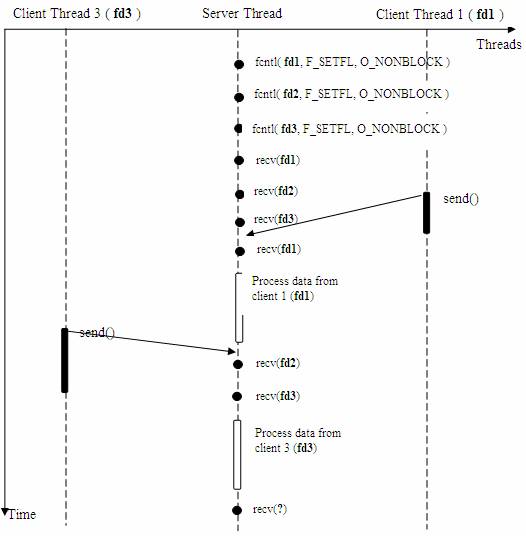


图5 使用非阻塞的接收数据模型

在非阻塞状态下，recv() 接口在被调用后立即返回，返回值代表了不同的含义。如在本例中，  
    \* recv() 返回值大于 0，表示接受数据完毕，返回值即是接受到的字节数；  
    \* recv() 返回 0，表示连接已经正常断开；  
    \* recv() 返回 -1，且 errno 等于 EAGAIN，表示 recv 操作还没执行完成；  
    \* recv() 返回 -1，且 errno 不等于 EAGAIN，表示 recv 操作遇到系统错误 errno。  
    可以看到服务器线程可以通过循环调用recv()接口，可以在单个线程内实现对所有连接的数据接收工作。但是上述模型绝不被推荐。因为，**循环调用recv()将大幅度推高CPU 占用率；此外，在这个方案中recv()更多的是起到检测“操作是否完成”的作用，实际操作系统提供了更为高效的检测“操作是否完成“作用的接口，例如select()多路复用模式，可以一次检测多个连接是否活跃。**  
    **3、IO复用**

**Io复用其实复用的还是io连接，而是复用进程。**  
    IO multiplexing这个词可能有点陌生，但是如果我说select/epoll，大概就都能明白了。有些地方也称这种IO方式为**事件驱动IO**(event driven IO)。我们都知道，select/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。它的基本原理就是select/epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。它的流程如图：

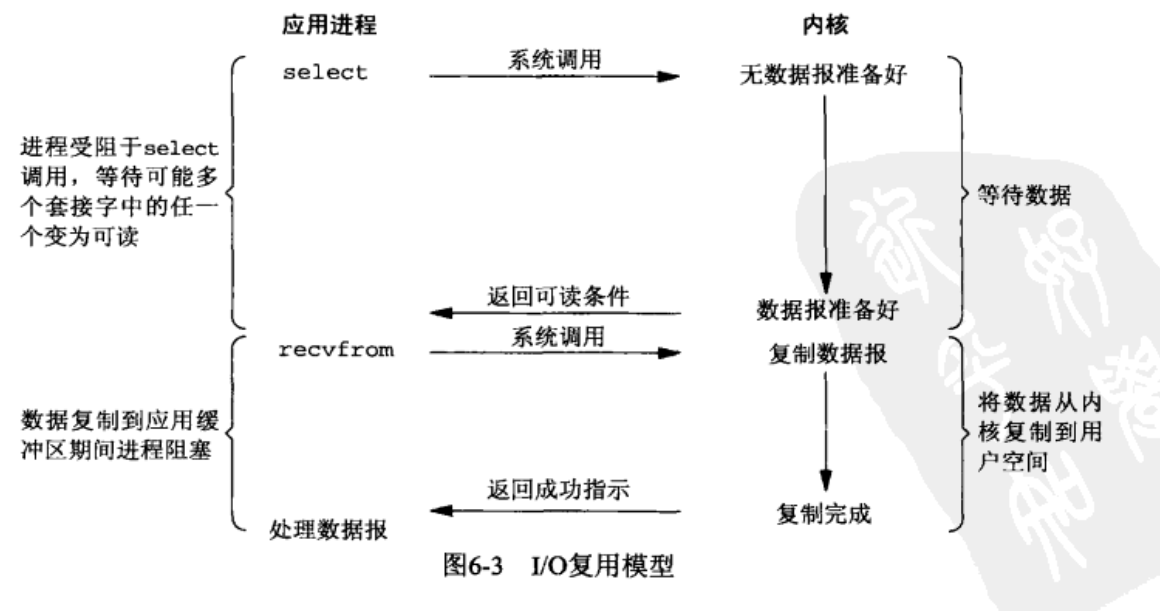


图6 多路复用IO

    当用户进程调用了select，那么整个进程会被阻塞，而同时，内核会“监视”所有select负责的socket，当任何一个socket中的数据准备好了，select就会返回。这个时候用户进程再调用read操作，将数据从内核拷贝到用户进程。  
    这个图和阻塞 IO的图其实并没有太大的不同，事实上还更差一些。因为这里需要使用两个系统调用(select和recvfrom)，而阻塞 IO只调用了一个系统调用(recvfrom)。但是，用select的优势在于它可以同时处理多个connection。（多说一句：所以，如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用multi-threading + blocking IO的web server性能更好，可能延迟还更大。select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）  
    **在多路复用模型中，对于每一个socket，一般都设置成为非阻塞，但是，如上图所示，整个用户的process其实是一直被阻塞的。**只不过process是被select这个函数阻塞，而不是被socket IO给阻塞。因此select()与非阻塞IO类似。

    大部分Unix/Linux都支持select函数，该函数用于探测多个文件句柄的状态变化。下面给出select接口的原型：  
    FD\_ZERO(int fd, fd\_set\* fds)   
    FD\_SET(int fd, fd\_set\* fds)   
    FD\_ISSET(int fd, fd\_set\* fds)   
    FD\_CLR(int fd, fd\_set\* fds)   
    int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds,   
    struct timeval \*timeout)   
    这里，fd\_set 类型可以简单的理解为按 bit 位标记句柄的队列，例如要在某 fd\_set 中标记一个值为16的句柄，则该fd\_set的第16个bit位被标记为1。具体的置位、验证可使用 FD\_SET、FD\_ISSET等宏实现。在select()函数中，readfds、writefds和exceptfds同时作为输入参数和输出参数。如果输入的readfds标记了16号句柄，则select()将检测16号句柄是否可读。在select()返回后，可以通过检查readfds有否标记16号句柄，来判断该“可读”事件是否发生。另外，用户可以设置timeout时间。  
    下面将重新模拟上例中从多个客户端接收数据的模型。

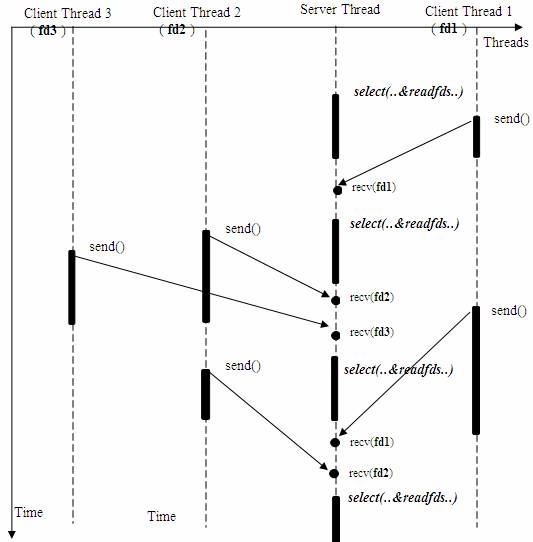


图7 使用select()的接收数据模型

    述模型只是描述了使用select()接口同时从多个客户端接收数据的过程；由于select()接口可以同时对多个句柄进行读状态、写状态和错误状态的探测，所以可以很容易构建为多个客户端提供独立问答服务的服务器系统。如下图。

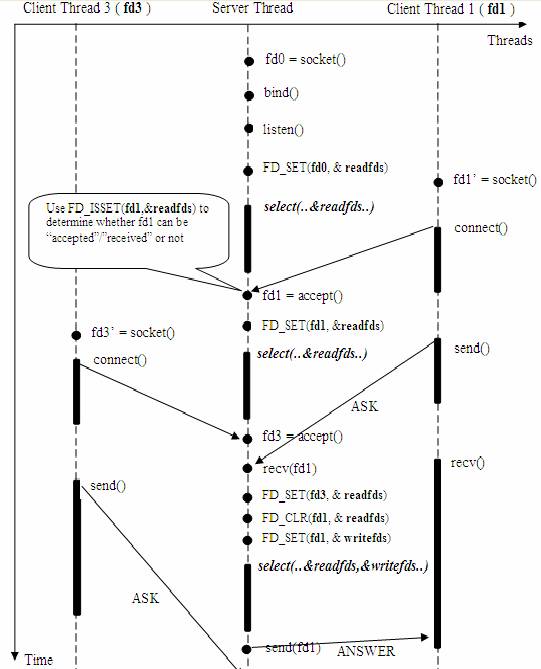


图8 使用select()接口的基于事件驱动的服务器模型

    这里需要指出的是，客户端的一个 connect() 操作，将在服务器端激发一个“可读事件”，所以 select() 也能探测来自客户端的 connect() 行为。  
    上述模型中，最关键的地方是如何动态维护select()的三个参数readfds、writefds和exceptfds。作为输入参数，readfds应该标记所有的需要探测的“可读事件”的句柄，其中永远包括那个探测 connect() 的那个“母”句柄；同时，writefds 和 exceptfds 应该标记所有需要探测的“可写事件”和“错误事件”的句柄 ( 使用 FD\_SET() 标记 )。  
    作为输出参数，readfds、writefds和exceptfds中的保存了 select() 捕捉到的所有事件的句柄值。程序员需要检查的所有的标记位 ( 使用FD\_ISSET()检查 )，以确定到底哪些句柄发生了事件。  
    上述模型主要模拟的是“一问一答”的服务流程，所以如果select()发现某句柄捕捉到了“可读事件”，服务器程序应及时做recv()操作，并根据接收到的数据准备好待发送数据，并将对应的句柄值加入writefds，准备下一次的“可写事件”的select()探测。同样，如果select()发现某句柄捕捉到“可写事件”，则程序应及时做send()操作，并准备好下一次的“可读事件”探测准备。下图描述的是上述模型中的一个执行周期。

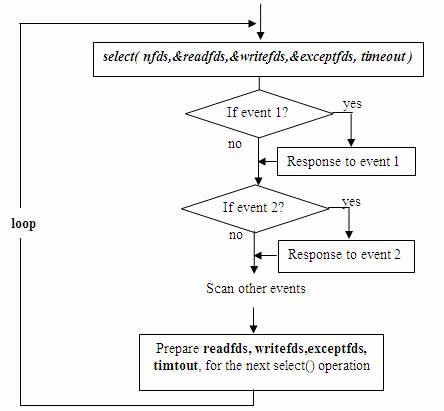


图9 多路复用模型的一个执行周期

    这种模型的特征在于每一个执行周期都会探测一次或一组事件，一个特定的事件会触发某个特定的响应。我们可以将这种模型归类为“**事件驱动模型**”。  
    相比其他模型，使用select() 的事件驱动模型只用单线程（进程）执行，占用资源少，不消耗太多 CPU，同时能够为多客户端提供服务。如果试图建立一个简单的事件驱动的服务器程序，这个模型有一定的参考价值。  
    但这个模型依旧有着很多问题。**首先select()接口并不是实现“事件驱动”的最好选择。因为当需要探测的句柄值较大时，select()接口本身需要消耗大量时间去轮询各个句柄。**很多操作系统提供了更为高效的接口，如linux提供了epoll，BSD提供了kqueue，Solaris提供了/dev/poll，…。如果需要实现更高效的服务器程序，类似epoll这样的接口更被推荐。遗憾的是不同的操作系统特供的epoll接口有很大差异，所以使用类似于epoll的接口实现具有较好跨平台能力的服务器会比较困难。  
    **其次，该模型将事件探测和事件响应夹杂在一起，一旦事件响应的执行体庞大，则对整个模型是灾难性的。**如下例，庞大的执行体1的将直接导致响应事件2的执行体迟迟得不到执行，并在很大程度上降低了事件探测的及时性。

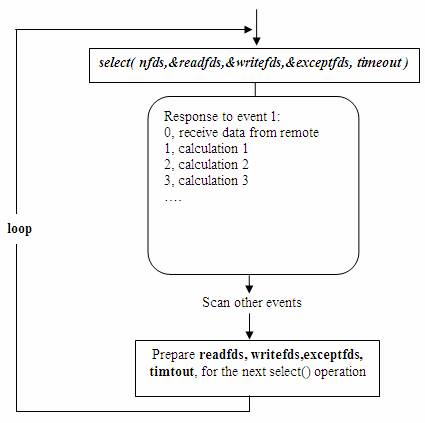


图10 庞大的执行体对使用select()的事件驱动模型的影响

    幸运的是，有很多高效的事件驱动库可以屏蔽上述的困难，常见的事件驱动库有**libevent库**，还有作为libevent替代者的**libev库**。这些库会根据操作系统的特点选择最合适的事件探测接口，并且加入了信号(signal) 等技术以支持异步响应，这使得这些库成为构建事件驱动模型的不二选择。下章将介绍如何使用libev库替换select或epoll接口，实现高效稳定的服务器模型。

实际上，Linux内核从2.6开始，也引入了支持异步响应的IO操作，如aio\_read, aio\_write，这就是异步IO。

1. 信号驱动式I/O

应用程序提交read请求，调用system call，然后内核开始处理相应的IO操作，而同时，应用程序并不等内核返回响应，就会开始执行其他的处理操作（应用程序没有被IO阻塞），当内核执行完毕，返回read响应，就会产生一个信号或执行一个基于线程的回调函数来完成这次IO处理过程。在这里IO的读写操作是在IO事件发生之后由应用程序来完成。异步IO读写操作总是立即返回，而不论IO是否阻塞，因为真正的读写操作已经有内核掌管。也就是说同步IO模型要求用户代码自行执行IO操作（将数据从内核缓冲区移动用户缓冲区或者相反），而异步操作机制则是由内核来执行IO操作（将数据从内核缓冲区移动用户缓冲区或者相反）。可以这样认为，同步IO向应用程序通知的是IO就绪事件，而异步IO向应用程序通知的是IO完成事件。

**5、异步IO（Asynchronous I/O）**  
    Linux下的asynchronous IO其实用得不多，从内核2.6版本才开始引入。先看一下它的流程：

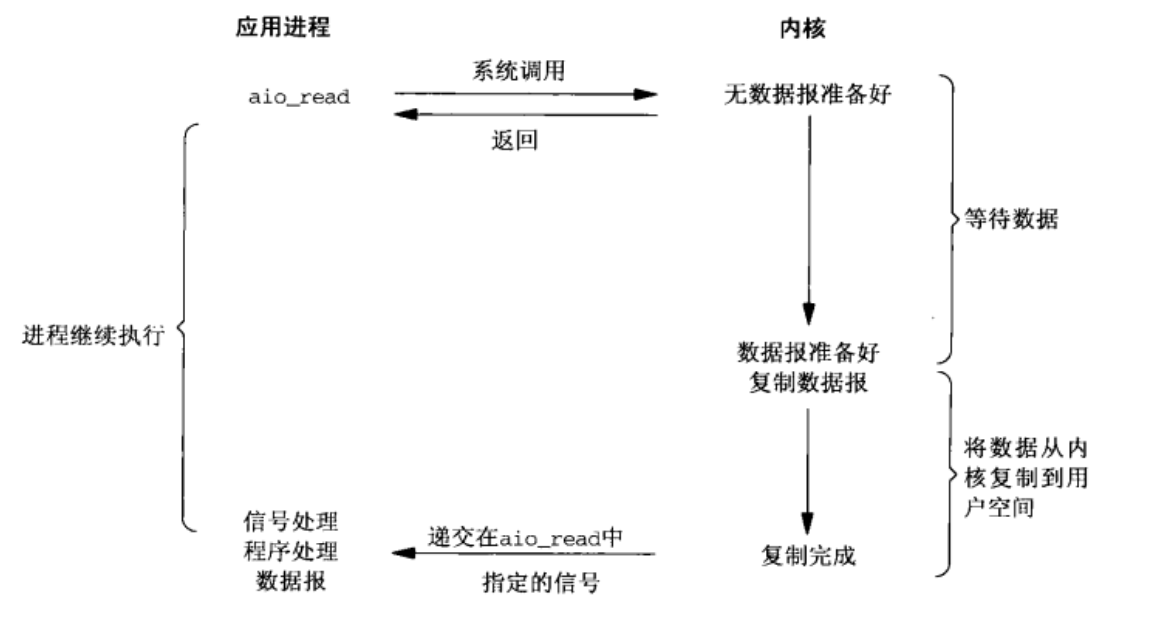


图11 异步IO

    用户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何阻塞。然后，内核会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，内核会给用户进程发送一个signal，告诉它read操作完成了。

    用异步IO实现的服务器这里就不举例了，以后有时间另开文章来讲述。异步IO是真正非阻塞的，它不会对请求进程产生任何的阻塞，因此对高并发的网络服务器实现至关重要。  
    到目前为止，已经将五个IO模型都介绍完了。现在回过头来回答最初的那几个问题：阻塞和非阻塞的区别在哪，synchronous IO和asynchronous IO的区别在哪。  
先回答最简单的这个：阻塞与非阻塞。前面的介绍中其实已经很明确的说明了这两者的区别。调用阻塞 IO会一直阻塞住对应的进程直到操作完成，而非阻塞 IO在内核还在准备数据的情况下会立刻返回。  
    在说明synchronous IO和asynchronous IO的区别之前，需要先给出两者的定义。Stevens给出的定义（其实是POSIX的定义）是这样子的：  
    \* A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes;  
    \* An asynchronous I/O operation does not cause the requesting process to be blocked;  
    两者的区别就在于synchronous IO做”IO operation”的时候会将process阻塞。按照这个定义，之前所述的阻塞 IO，非阻塞 IO，IO multiplexing都属于synchronous IO。有人可能会说，非阻塞 IO并没有被阻塞啊。这里有个非常“狡猾”的地方，定义中所指的”IO operation”是指真实的IO操作，就是例子中的recvfrom这个系统调用。**非阻塞 IO在执行recvfrom这个系统调用的时候，如果内核的数据没有准备好，这时候不会阻塞进程。但是当内核中数据准备好的时候，recvfrom会将数据从内核拷贝到用户内存中，这个时候进程是被阻塞了，在这段时间内进程是被阻塞的。**而asynchronous IO则不一样，当进程发起IO操作之后，就直接返回再也不理睬了，直到内核发送一个信号，告诉进程说IO完成。在这整个过程中，进程完全没有被阻塞。

   总的来说，UNP中总结的**IO模型有5种之多：阻塞IO，非阻塞IO，IO复用，信号驱动IO，异步IO**。前四种都属于同步IO。阻塞IO不必说了。非阻塞IO ，IO请求时加上O\_NONBLOCK一类的标志位，立刻返回，IO没有就绪会返回错误，需要请求进程主动轮询不断发IO请求直到返回正确。IO复用同非阻塞IO本质一样，不过利用了新的select系统调用，由内核来负责本来是请求进程该做的轮询操作。看似比非阻塞IO还多了一个系统调用开销，不过因为可以支持多路IO，才算提高了效率。信号驱动IO，调用sigaltion系统调用，当内核中IO数据就绪时以SIGIO信号通知请求进程，请求进程再把数据从内核读入到用户空间，这一步是阻塞的。  
异步IO，如定义所说，不会因为IO操作阻塞，IO操作全部完成才通知请求进程。  
    各个IO Model的比较如图所示：

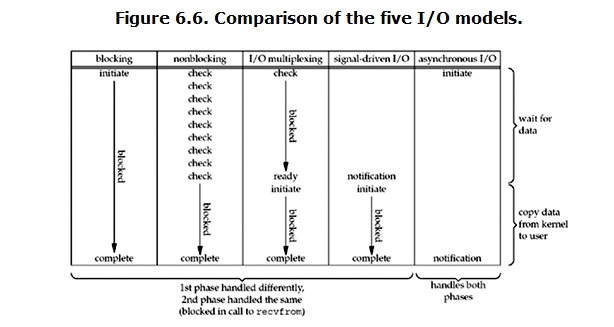


图12 各种IO模型的比较

    经过上面的介绍，会发现非阻塞 IO和asynchronous IO的区别还是很明显的。在非阻塞 IO中，虽然进程大部分时间都不会被阻塞，但是它仍然要求进程去主动的check，并且当数据准备完成以后，也需要进程主动的再次调用recvfrom来将数据拷贝到用户内存。而asynchronous IO则完全不同。它就像是用户进程将整个IO操作交给了他人（内核）完成，然后他人做完后发信号通知。在此期间，用户进程不需要去检查IO操作的状态，也不需要主动的去拷贝数据。

epoll跟select都能提供多路I/O复用的解决方案。在现在的Linux内核里有都能够支持，其中epoll是Linux所特有，而select则应该是POSIX所规定，一般操作系统均有实现

**select：**

select本质上是通过设置或者检查存放fd标志位的数据结构来进行下一步处理。这样所带来的缺点是：

1、 单个进程可监视的fd数量被限制，即能监听端口的大小有限。

      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察看。32位机默认是1024个。64位机默认是2048.

2、 对socket进行扫描时是线性扫描，即采用轮询的方法，效率较低：

       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与kqueue做的。

3、需要维护一个用来存放大量fd的数据结构，这样会使得用户空间和内核空间在传递该结构时复制开销大

**poll：**

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。

它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：

1、大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。                                                                                                                                      2、poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。

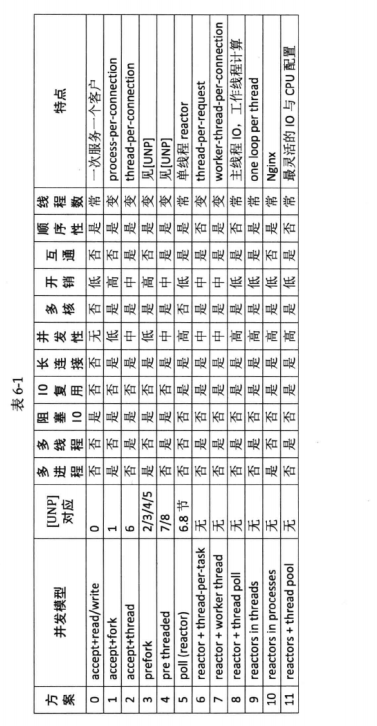
**epoll:**

epoll支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些fd刚刚变为就需态，并且只会通知一次。还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll\_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll\_wait便可以收到通知

select+阻塞、非阻塞？

http://www.52im.net/thread-568-1-1.html

3、并发网络方案



方案1、accept+read/write。

最简单的方式，并不是并发服务器，只能一次服务一个客户。

方案2、accept+fork（pool）。

传统的unix并发编程方案，对每一个tcp连接启动一个进程，适合并发数不大的情况，适合事务处理远大于fork()的开销，如数据库服务器。且适合长连接，不适合短连接。

多进程的优点：每个进程相互独立，一个进程的崩溃不会引起另外进程崩溃。通过增加CPU就可以提高性能，且没有同步互斥的复杂控制逻辑。

多进程缺点：进程的创建销毁消耗系统资源，此外如果有跨进程的数据通信就比较复杂，使用只有小数据量的进程间通信的场景，进程的多少受限于系统资源。

方案3、accept+thread(pool)。

开启线程的开销比进程的开销小，这是是对方案2的优化，

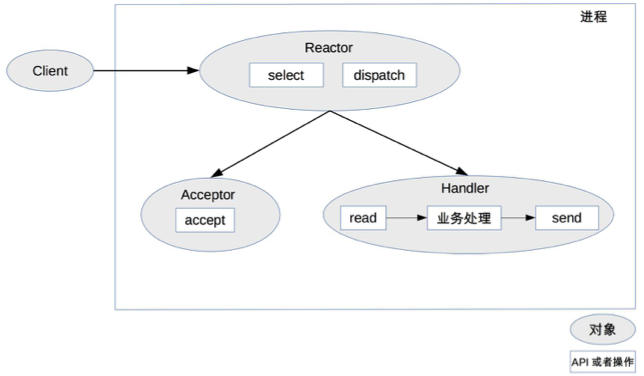
优点：使用多线程解决并发的优点是编码比较简单，代码量少，同时响应比较快。

缺点：使用多线程的缺点比较多，多线程解决并发受限于系统资源，创建一个线程默认要创建1M大小的栈空间，此外线程创建，销毁，切换都需要消耗系统资源，因此线程数量到一定程度后哪怕增加系统资源也无法增加线程数量，对于大的并发量无法支持。如果程序间需要涉及到数据的同步互斥，那么整个逻辑会比较麻烦，线程间的资源同步互斥比较难控制，除了问题也比较难调试。一个线程崩溃了可能引起整个进程的崩溃，进而影响服务器性能。

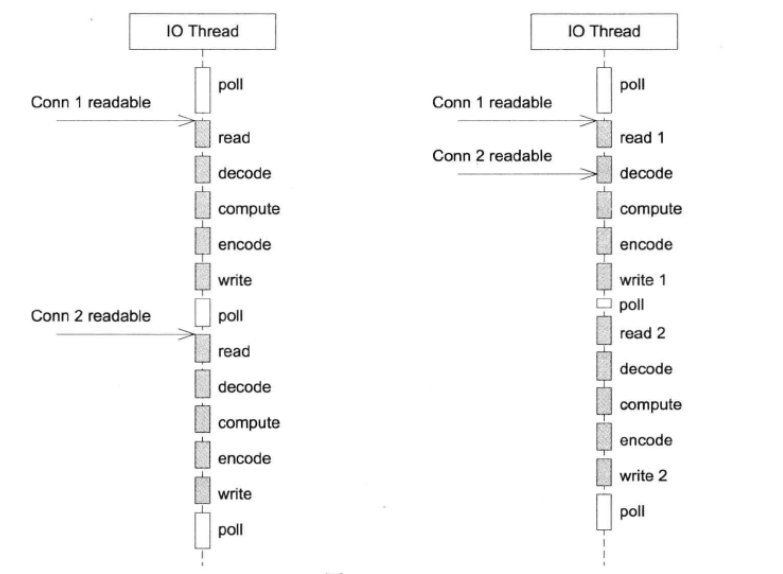
这几种方案都是阻塞式网络编程，程序通常阻塞在read()上，但TCP是全双工协议，当一个进程阻塞在read()上，同时进程又要发数据怎么办，另一种方法就是io复用

方案4、单线程reactor

Reactor模式是一个架构模式，它主要解决的问题是**高并发场景下的服务器的性能问题**。

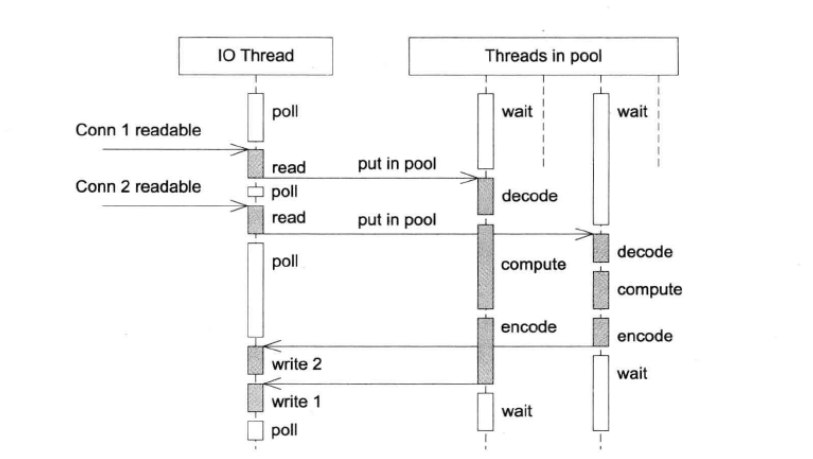


上图就是基于socket的reactor模式，socket作为文件fd，被多路事件分离器监听，这个多路事件分离器一般就是指Select/poll这种IO多路复用的机制，select/poll将事件分离出来后，委托给反应器去寻找创建对应的事件处理器，当创建完成后，以后有socket事件进来，直接由select分离出来，交由对应的Handler去处理。

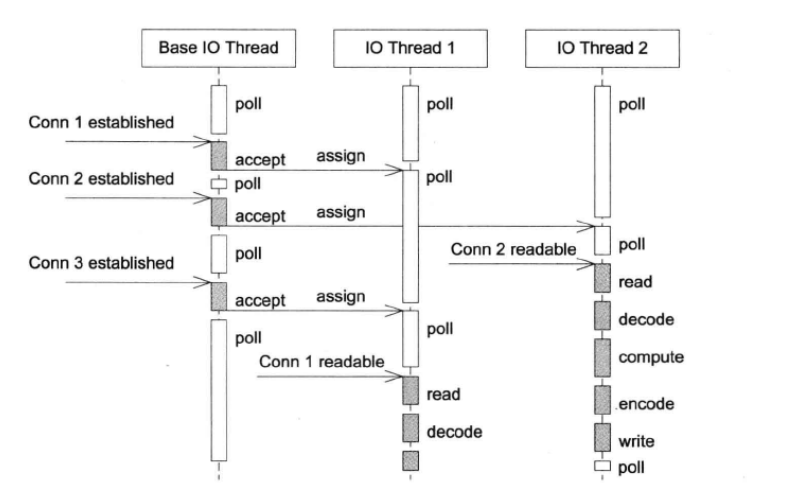


单线程Reactor的执行顺序如图，在没有事件时，线程阻塞在select/poll/epoll上，事件到达后线程进行读写等操作，由于是单线程，从poll收到事件后到下一次调用poll前线程不能抢占，TCP连接的优先级得不到保证。

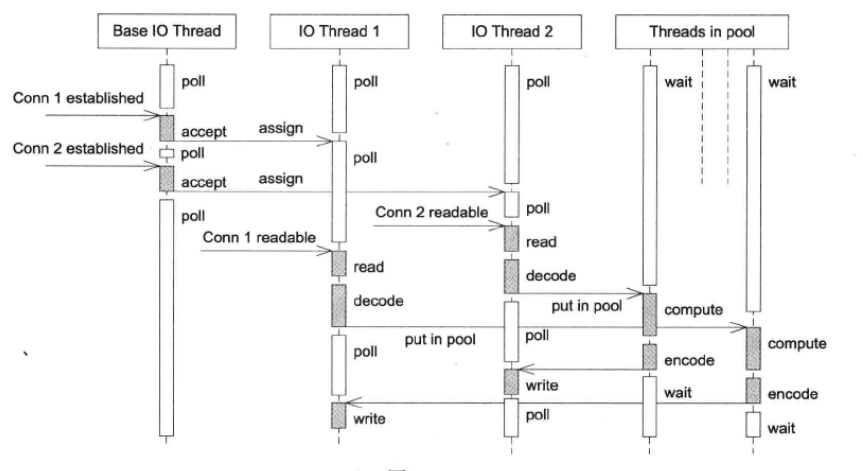
方案5、reactor + thread(pool)



方案6、reactors in threads/processes



方案7、reactors + thread pool



4 、nginx

## Nginx的进程模型

在工作方式上，Nginx分为单工作进程和多工作进程两种模式。在单工作进程模式下，除主进程外，还有一个工作进程，工作进程是单线程的；在多工作进程模式下，每个工作进程包含多个线程。Nginx默认为单工作进程模式。

Nginx在启动后，会有一个master进程和多个worker进程。

**master进程**

主要用来管理worker进程，包含：接收来自外界的信号，向各worker进程发送信号，监控worker进程的运行状态，当worker进程退出后(异常情况下)，会自动重新启动新的worker进程。

master进程充当整个进程组与用户的交互接口，同时对进程进行监护。它不需要处理网络事件，不负责业务的执行，只会通过管理worker进程来实现重启服务、平滑升级、更换日志文件、配置文件实时生效等功能。

我们要控制nginx，只需要通过kill向master进程发送信号就行了。比如kill -HUP pid，则是告诉nginx，从容地重启nginx，我们一般用这个信号来重启nginx，或重新加载配置，因为是从容地重启，因此服务是不中断的。master进程在接收到HUP信号后是怎么做的呢？

首先master进程在接到信号后，会先重新加载配置文件，然后再启动新的worker进程，并向所有老的worker进程发送信号，告诉他们可以光荣退休了。新的worker在启动后，就开始接收新的请求，而老的worker在收到来自master的信号后，就不再接收新的请求，并且在当前进程中的所有未处理完的请求处理完成后，再退出。

当然，直接给master进程发送信号，这是比较老的操作方式，nginx在0.8版本之后，引入了一系列命令行参数，来方便我们管理。比如，./nginx -s reload，就是来重启nginx，./nginx -s stop，就是来停止nginx的运行。

如何做到的呢？我们还是拿reload来说，我们看到，执行命令时，我们是启动一个新的nginx进程，而新的nginx进程在解析到reload参数后，就知道我们的目的是控制nginx来重新加载配置文件了，它会向master进程发送信号，然后接下来的动作，就和我们直接向master进程发送信号一样了。

**worker进程：**

而基本的网络事件，则是放在worker进程中来处理了。多个worker进程之间是对等的，他们同等竞争来自客户端的请求，各进程互相之间是独立的。一个请求，只可能在一个worker进程中处理，一个worker进程，不可能处理其它进程的请求。worker进程的个数是可以设置的，一般我们会设置与机器cpu核数一致，这里面的原因与nginx的进程模型以及事件处理模型是分不开的。

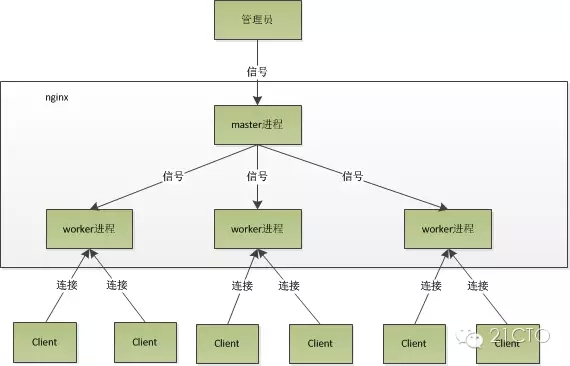
worker进程之间是平等的，每个进程，处理请求的机会也是一样的。当我们提供80端口的http服务时，一个连接请求过来，每个进程都有可能处理这个连接，怎么做到的呢？首先，每个worker进程都是从master进程fork过来，在master进程里面，先建立好需要listen的socket（listenfd）之后，然后再fork出多个worker进程。

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。worker进程之间是平等的，每个进程，处理请求的机会也是一样的。当我们提供80端口的http服务时，一个连接请求过来，每个进程都有可能处理这个连接，怎么做到的呢？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。

nginx的进程模型，可以由下图来表示：



**Nginx为啥性能高－多进程IO模型**

**1、nginx采用多进程模型好处**

首先，对于每个worker进程来说，独立的进程，不需要加锁，所以省掉了锁带来的开销，同时在编程以及问题查找时，也会方便很多。

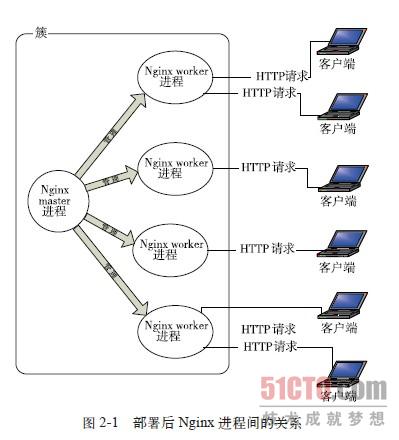
其次，采用独立的进程，可以让互相之间不会影响，一个进程退出后，其它进程还在工作，服务不会中断，master进程则很快启动新的worker进程。当然，worker进程的异常退出，肯定是程序有bug了，异常退出，会导致当前worker上的所有请求失败，不过不会影响到所有请求，所以降低了风险。

**1、nginx多进程事件模型：异步非阻塞**

虽然nginx采用多worker的方式来处理请求，每个worker里面只有一个主线程，那能够处理的并发数很有限啊，多少个worker就能处理多少个并发，何来高并发呢？非也，这就是nginx的高明之处，**nginx采用了异步非阻塞的方式来处理请求**，也就是说，nginx是可以同时处理成千上万个请求的。

一个worker进程可以同时处理的请求数只受限于内存大小，而且在架构设计上，不同的worker进程之间处理并发请求时几乎没有同步锁的限制，worker进程通常不会进入睡眠状态，因此，当Nginx上的进程数与CPU核心数相等时（最好每一个worker进程都绑定特定的CPU核心），进程间切换的代价是最小的。

而apache的常用工作方式（apache也有异步非阻塞版本，但因其与自带某些模块冲突，所以不常用），每个进程在一个时刻只处理一个请求，因此，当并发数上到几千时，就同时有几千的进程在处理请求了。这对操作系统来说，是个不小的挑战，进程带来的内存占用非常大，进程的上下文切换带来的cpu开销很大，自然性能就上不去了，而这些开销完全是没有意义的。



为什么nginx可以采用异步非阻塞的方式来处理呢，或者异步非阻塞到底是怎么回事呢？

我们先回到原点，看看一个请求的完整过程:首先，请求过来，要建立连接，然后再接收数据，接收数据后，再发送数据。

具体到系统底层，就是读写事件，而当读写事件没有准备好时，必然不可操作，如果不用非阻塞的方式来调用，那就得阻塞调用了，事件没有准备好，那就只能等了，等事件准备好了，你再继续吧。阻塞调用会进入内核等待，cpu就会让出去给别人用了，对单线程的worker来说，显然不合适，当网络事件越多时，大家都在等待呢，cpu空闲下来没人用，cpu利用率自然上不去了，更别谈高并发了。

好吧，你说加进程数，这跟apache的线程模型有什么区别，注意，别增加无谓的上下文切换。所以，在nginx里面，最忌讳阻塞的系统调用了。不要阻塞，那就非阻塞喽。非阻塞就是，事件没有准备好，马上返回EAGAIN，告诉你，事件还没准备好呢，你慌什么，过会再来吧。

好吧，你过一会，再来检查一下事件，直到事件准备好了为止，在这期间，你就可以先去做其它事情，然后再来看看事件好了没。虽然不阻塞了，但你得不时地过来检查一下事件的状态，你可以做更多的事情了，但带来的开销也是不小的。

[关于IO模型：http://blog.csdn.net/hguisu/article/details/7453390](http://blog.csdn.net/hguisu/article/details/7453390)

**nginx支持的事件模型如下（nginx的wiki）:**

Nginx支持如下处理连接的方法（I/O复用方法），这些方法可以通过use指令指定。

* **select**– 标准方法。 如果当前平台没有更有效的方法，它是编译时默认的方法。你可以使用配置参数 –with-select\_module 和 –without-select\_module 来启用或禁用这个模块。
* **poll**– 标准方法。 如果当前平台没有更有效的方法，它是编译时默认的方法。你可以使用配置参数 –with-poll\_module 和 –without-poll\_module 来启用或禁用这个模块。
* **kqueue**– 高效的方法，使用于 FreeBSD 4.1+, OpenBSD 2.9+, NetBSD 2.0 和 MacOS X. 使用双处理器的MacOS X系统使用kqueue可能会造成内核崩溃。
* **epoll** – 高效的方法，使用于Linux内核2.6版本及以后的系统。在某些发行版本中，如SuSE 8.2, 有让2.4版本的内核支持epoll的补丁。
* **rtsig** – 可执行的实时信号，使用于Linux内核版本2.2.19以后的系统。默认情况下整个系统中不能出现大于1024个POSIX实时(排队)信号。这种情况 对于高负载的服务器来说是低效的；所以有必要通过调节内核参数 /proc/sys/kernel/rtsig-max 来增加队列的大小。可是从Linux内核版本2.6.6-mm2开始， 这个参数就不再使用了，并且对于每个进程有一个独立的信号队列，这个队列的大小可以用 RLIMIT\_SIGPENDING 参数调节。当这个队列过于拥塞，nginx就放弃它并且开始使用 poll 方法来处理连接直到恢复正常。
* **/dev/poll** – 高效的方法，使用于 Solaris 7 11/99+, HP/UX 11.22+ (eventport), IRIX 6.5.15+ 和 Tru64 UNIX 5.1A+.
* **eventport** – 高效的方法，使用于 Solaris 10. 为了防止出现内核崩溃的问题， 有必要安装[这个](http://sunsolve.sun.com/search/document.do?assetkey=1-26-102485-1) 安全补丁。

在linux下面，只有epoll是高效的方法

下面再来看看epoll到底是如何高效的  
Epoll是[Linux内核](http://www.hudong.com/wiki/Linux%C3%A5%C2%86%C2%85%C3%A6%20%C2%B8)为处理大批量句柄而作了改进的[poll](http://www.hudong.com/wiki/poll)。 要使用epoll只需要这三个系统调用：epoll\_create(2)， epoll\_ctl(2)， epoll\_wait(2)。它是在2.5.44内核中被引进的(epoll(4) is a new API introduced in Linux kernel 2.5.44)，在2.6内核中得到广泛应用。

epoll的优点

* 支持一个进程打开大数目的[socket](http://www.hudong.com/wiki/socket)描述符(FD)

select 最不能忍受的是一个进程所打开的FD是有一定限制的，由FD\_SETSIZE设置，默认值是2048。对于那些需要支持的上万连接数目的IM服务器来说显 然太少了。这时候你一是可以选择修改这个宏然后重新编译内核，不过资料也同时指出这样会带来网络效率的下降，二是可以选择多进程的解决方案(传统的 Apache方案)，不过虽然linux上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步的高效，所以也不是一种完 美的方案。不过 epoll则没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左 右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

* IO效率不随FD数目增加而线性下降

传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，不过由于网络延时，任一时间只有部分的socket是”活跃”的，但 是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对”活跃”的socket进行操 作—这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有”活跃”的socket才会主动的去调用 callback函数，其他idle状态socket则不会，在这点上，epoll实现了一个”伪”AIO，因为这时候推动力在os内核。

在一些 benchmark中，如果所有的socket基本上都是活跃的—比如一个高速LAN环境，epoll并不比select/poll有什么效率，相 反，如果过多使用epoll\_ctl,效率相比还有稍微的下降。但是一旦使用idle connections模拟[WAN](http://www.hudong.com/wiki/WAN)环境,epoll的效率就远在select/poll之上了。

* 使用[mmap](http://www.hudong.com/wiki/mmap)加速内核与用户空间的消息传递。

这 点实际上涉及到epoll的具体实现了。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很 重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。而如果你想我一样从2.5内核就关注epoll的话，一定不会忘记手工 mmap这一步的。

* 内核微调

这一点其实不算epoll的优点了，而是整个linux平台的优点。也许你可以怀疑linux平台，但是你无法回避linux平台赋予你微调内核的能力。比如，内核[TCP/IP](http://www.hudong.com/wiki/TCP/IP)协 议栈使用内存池管理sk\_buff结构，那么可以在运行时期动态调整这个内存pool(skb\_head\_pool)的大小— 通过echo XXXX>/proc/sys/net/core/hot\_list\_length完成。再比如listen函数的第2个参数(TCP完成3次握手 的数据包队列长度)，也可以根据你平台内存大小动态调整。更甚至在一个数据包面数目巨大但同时每个数据包本身大小却很小的特殊系统上尝试最新的NAPI网卡驱动架构。

**(epoll内容，参考epoll\_互动百科)**

推荐设置worker的个数为cpu的核数，在这里就很容易理解了，更多的worker数，只会导致进程来竞争cpu资源了，从而带来不必要的上下文切换。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。

像这种小的优化在nginx中非常常见，同时也说明了nginx作者的苦心孤诣。比如，nginx在做4个字节的字符串比较时，会将4个字符转换成一个int型，再作比较，以减少cpu的指令数等等。