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。





## 1.1 socket bind accept connect close shutdown

Close: 终止读和写两个方向。

Shutdown: TCP是全双工的，可关闭一半。

服务端tcp连接的简单程序：

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);

}

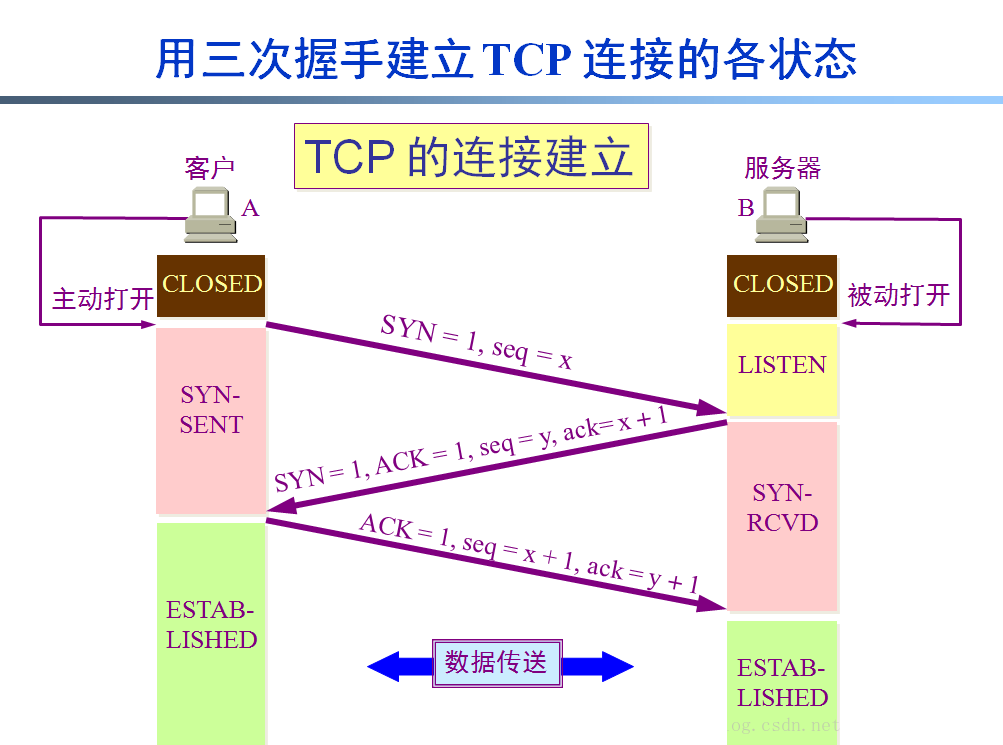
close(connfd);

}

## 1.2网络编程的一些小问题：

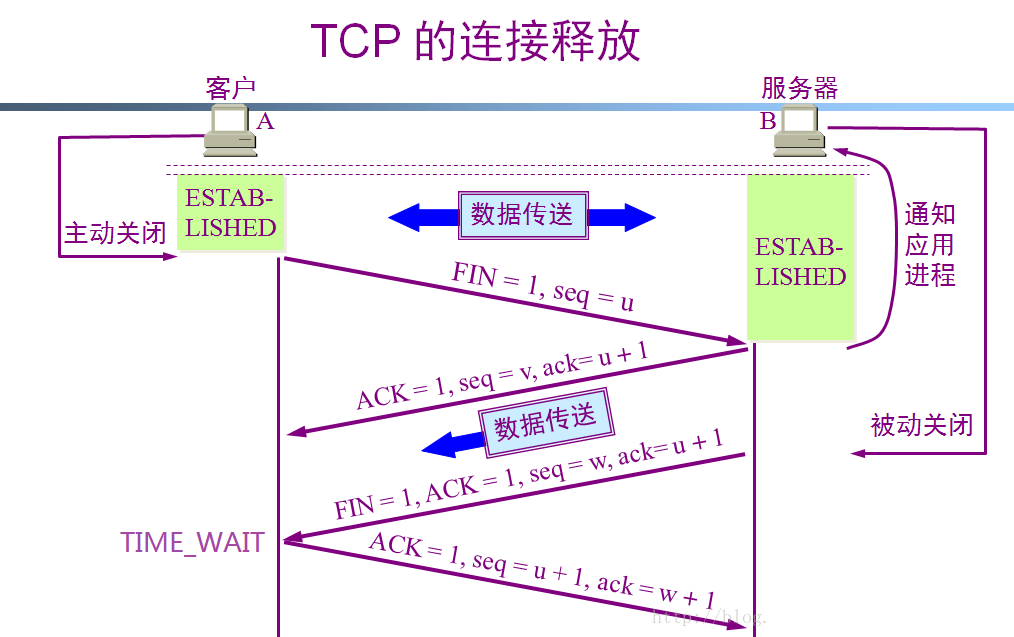
### 1、accept返回前连接中止

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。

### 6、比新建连接难的是关闭连接。谁来主动关闭？time\_wait?

服务端主动？关闭连接后会进入TIME\_WAIT状态，占用服务器资源。

客户端主动？赖着不走怎么办？

谁关闭？

服务端与客户端协商，客户端主动断开，恶意客户超时踢掉。

Time\_wait?

1. 修改配置，**内核time\_wait等待的值、time\_wait最大数量、tcp\_tw\_recycle:回收TIME\_WAIT连接、tcp\_tw\_reuse:复用TIME\_WAIT连接**

2）将TCP短连接改造为长连接

3）**RST**强行关闭**：**

使用RST代替FIN，发送一个RST标志给对端，该TCP连接将立刻夭折，无论是否有排队数据未发送或未被确认。

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

# 2、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

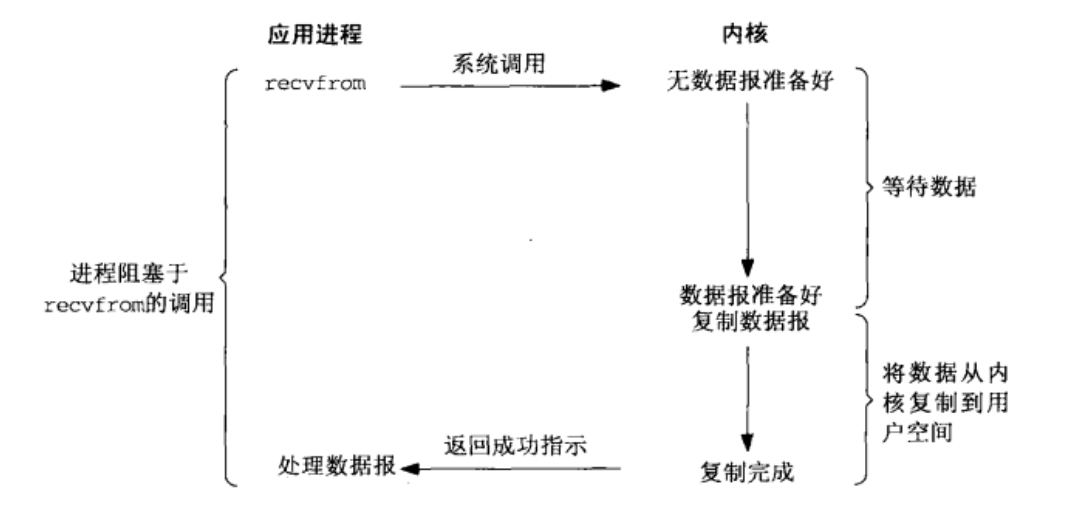


图1 阻塞IO

    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。

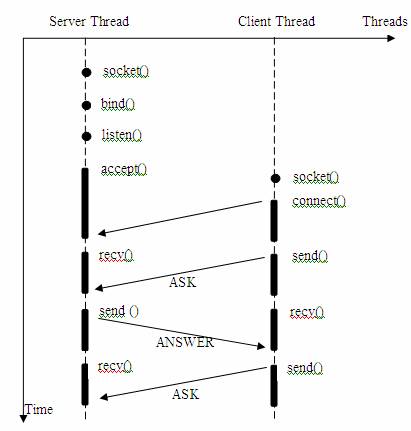


图2 简单的一问一答的服务器/客户机模型

    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。

**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：

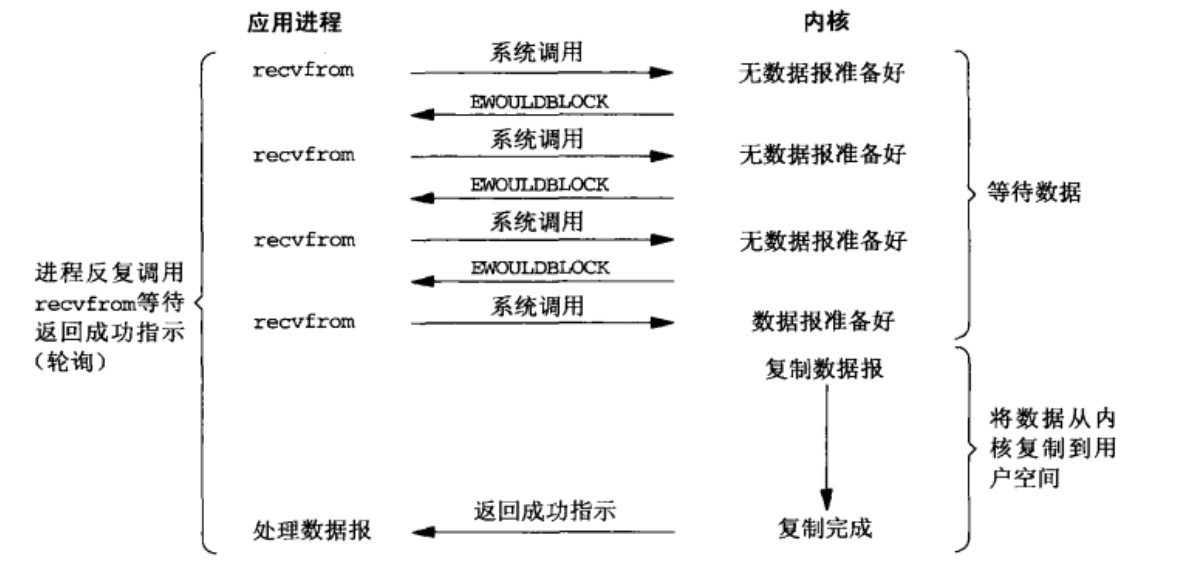


图4 非阻塞IO

    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。

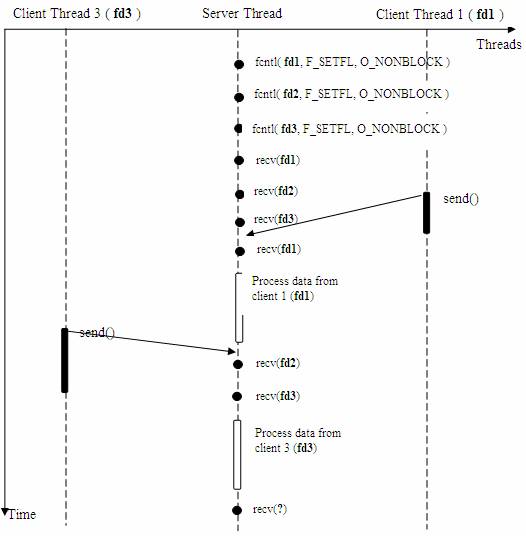


图5 使用非阻塞的接收数据模型

    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。**  
   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：

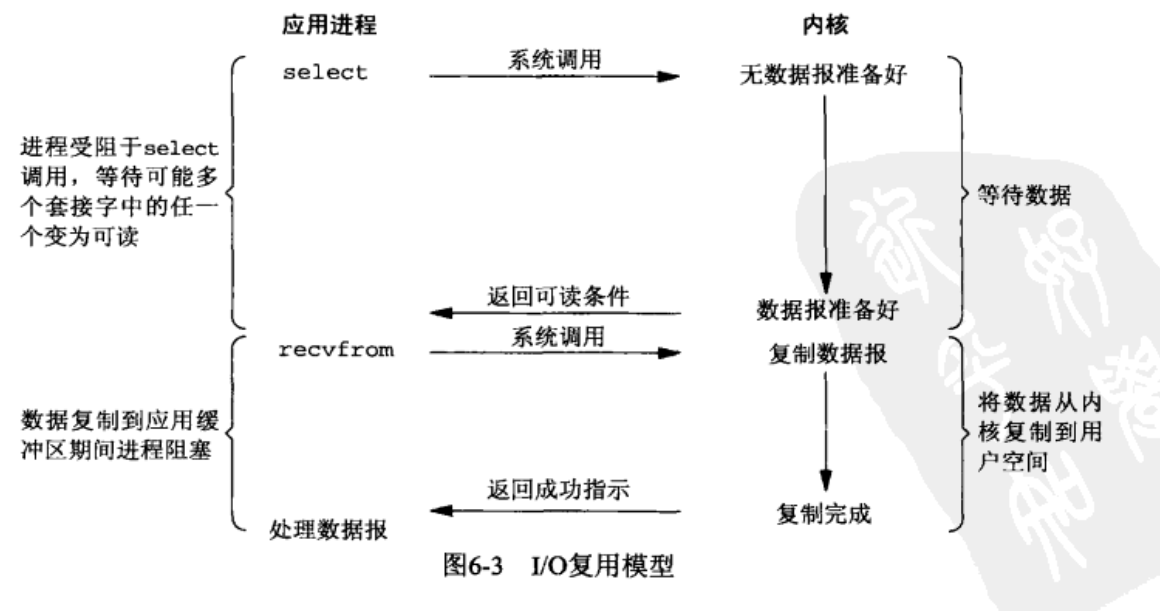


图6 多路复用IO

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。

**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。

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：  
       int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds,   
    struct timeval \*timeout) 

    下面将重新模拟上例中从多个客户端接收数据的模型。

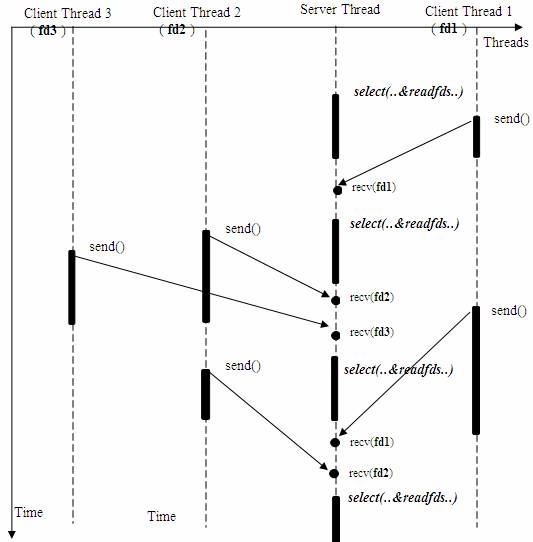


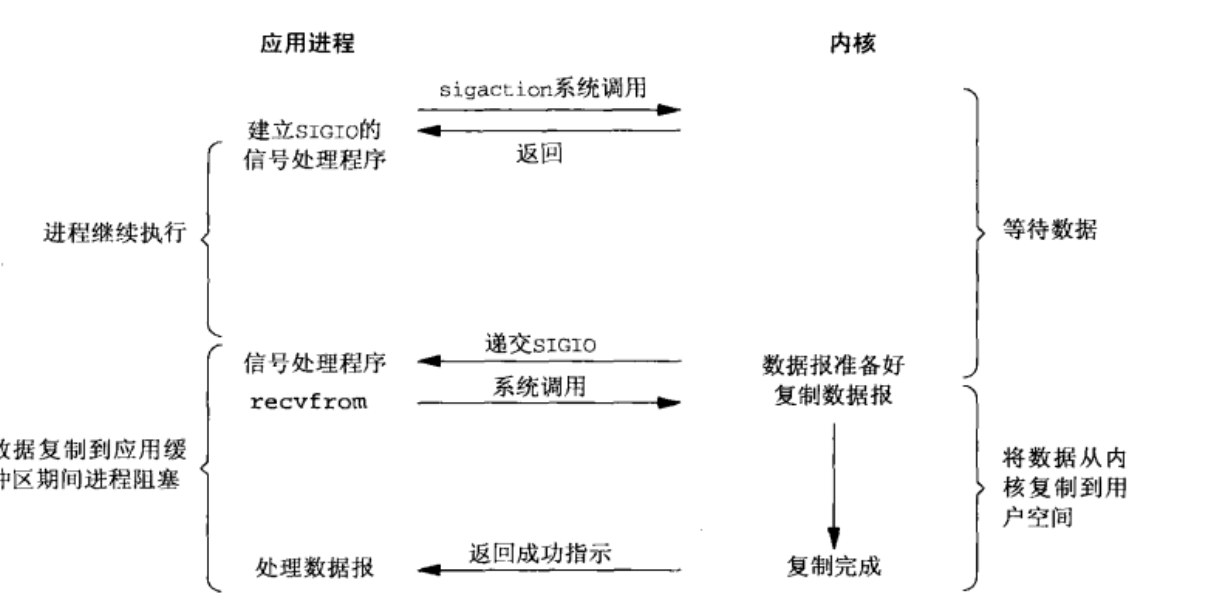
图7 使用select()的接收数据模型

    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。

    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，…。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。跨平台能力的比较困难。

4、信号驱动式I/0

调用sigaction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。



**5、异步IO（Asynchronous I/O）**

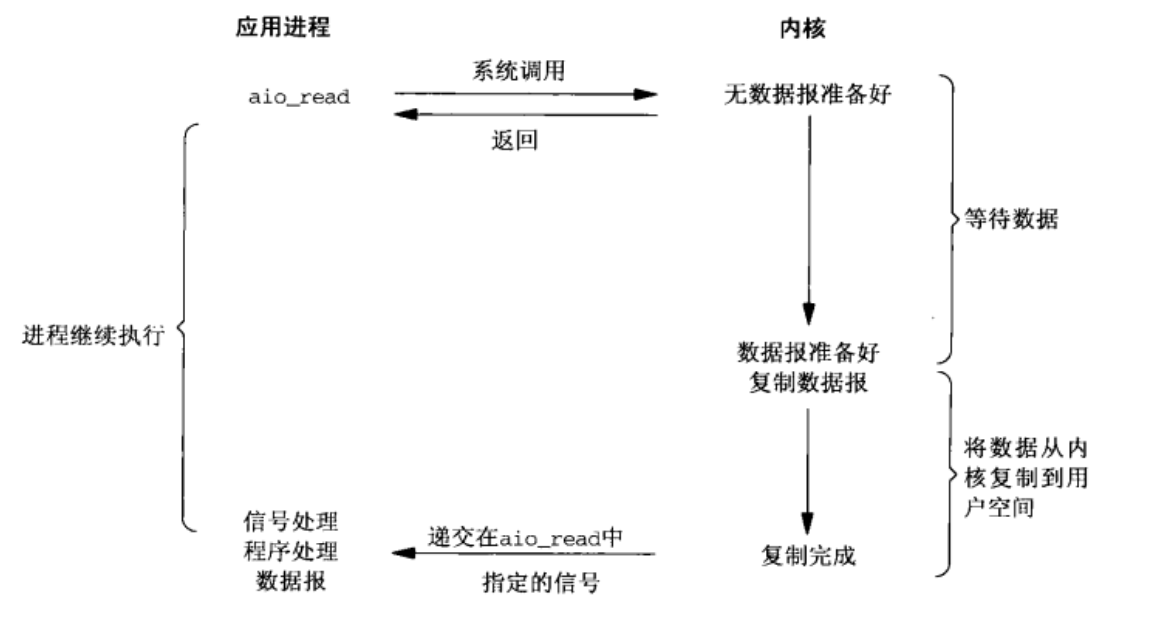


图11 异步IO

    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。

   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。

**阻塞和非阻塞的区别在哪？**

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。

**同步 IO和异步 IO的区别在哪？**

       两者的区别就在于同步IO做IO 操作的时候会将process阻塞。

按照这个定义，之前所述的阻塞 IO，非阻塞 IO，IO 复用都属于同步 IO。有人可能会说，非阻塞 IO并没有被阻塞啊。这里有个非常注意的地方，定义中所指的”IO 操作”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。**而异步 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

。

## select：

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。

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。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()都要通过遍历所有描述符来完成调度,不管哪个Socket是活跃的,都遍历一遍。这会浪费很多CPU时间。

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

epoll的优点

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

select 是一个进程所打开的FD是有一定限制的，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。

2、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， 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”的描述符进行操作

## select+阻塞、非阻塞？

阻塞IO: 如果一个wirte事件到达，select返回，内核拷贝数据到tcp缓冲区，缓冲区到达高水位，阻塞。

非阻塞IO: 需要应用层缓冲区。

# 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。

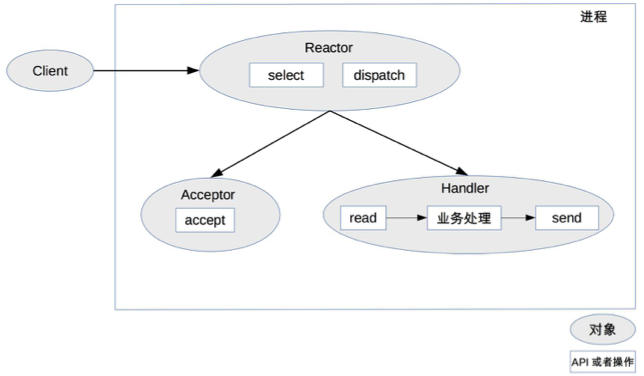
缺点：使用多线程的缺点比较多，多线程解决并发受限于系统资源，创建一个线程默认要创建1M大小的栈空间，此外线程创建，销毁，切换都需要消耗系统资源，因此线程数量到一定程度后哪怕增加系统资源也无法增加线程数量，对于大的并发量无法支持。如果程序间需要涉及到数据的同步互斥，那么整个逻辑会比较麻烦，线程间的资源同步互斥比较难控制，除了问题也比较难调试。一个线程崩溃了可能引起整个进程的崩溃，进而影响服务器性能。

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

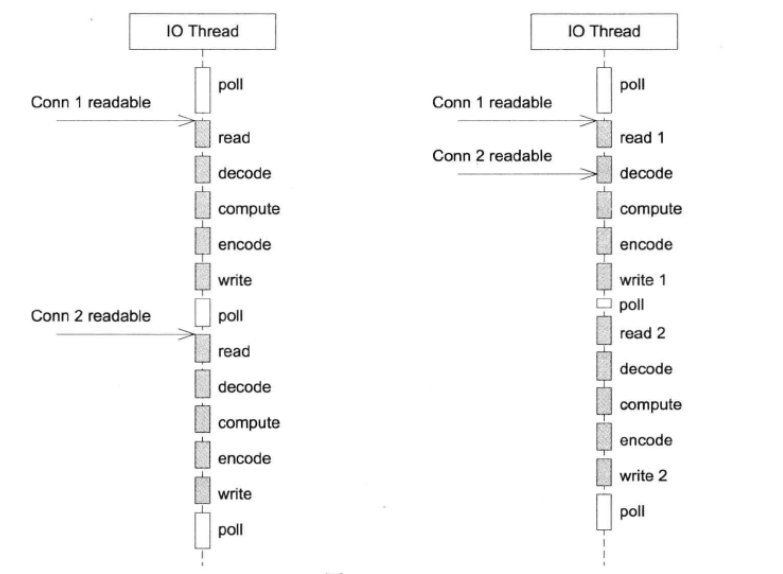
“线程池”和“连接池”技术也只是在一定程度上缓解了频繁调用IO接口带来的资源占用。而且，**所谓“池”始终有其上限，当请求大大超过上限时，“池”构成的系统对外界的响应并不比没有池的时候效果好多少。所以使用“池”必须考虑其面临的响应规模，并根据响应规模调整“池”的大小。**

## 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**。

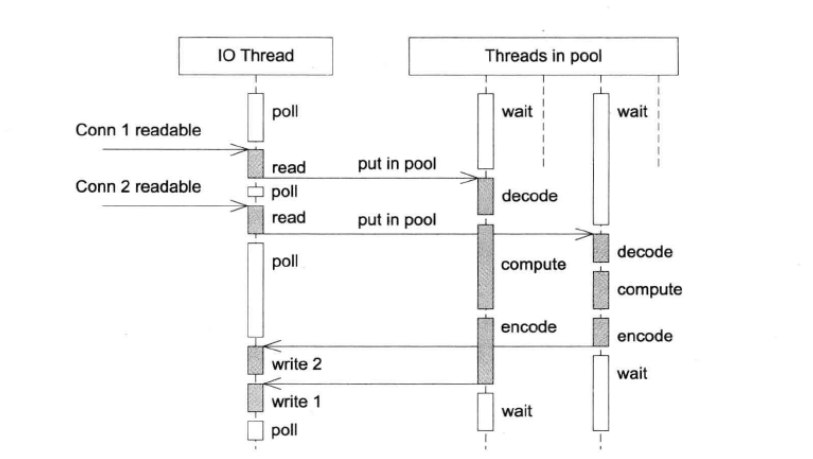


上图就是基于socket的reactor模式，socket作为文件fd，被多路事件分离器监听，这个多路事件分离器一般就是指Select/poll这种IO多路复用的机制，select/poll将事件分离出来后，委托给反应器去寻找创建对应的事件处理器，当创建完成后，以后有socket事件进来，直接由select分离出来，交由对应的Handler去处理。

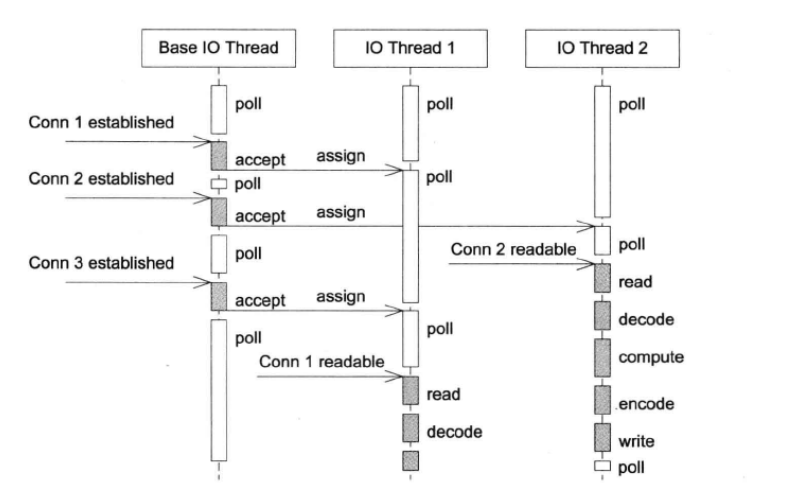


单线程Reactor的执行顺序如图，在没有事件时，线程阻塞在select/poll/epoll上，事件到达后线程进行读写等操作，由于是单线程，从poll收到事件后到下一次调用poll前线程不能抢占，TCP连接的优先级得不到保证。

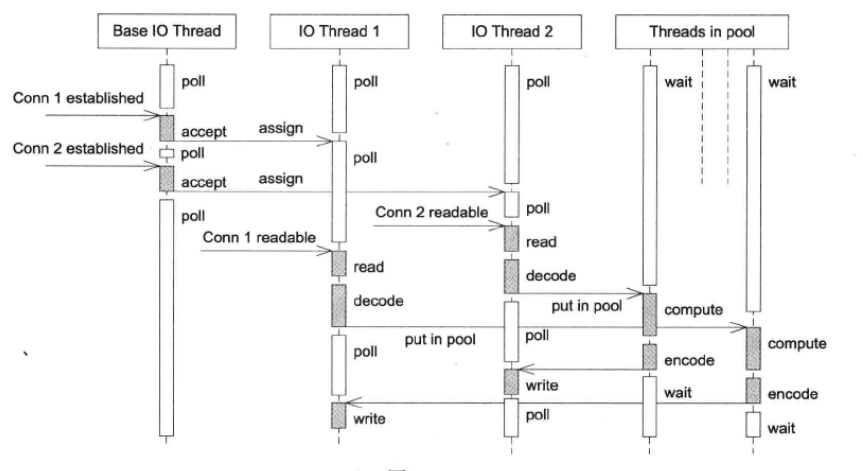
## 方案5、reactor + thread(pool)



## 方案6、reactors in threads/processes



## 方案7、reactors + thread pool



# 4 、nginx

## Nginx的进程模型

Nginx在启动后，会有一个master进程和多个worker进程。

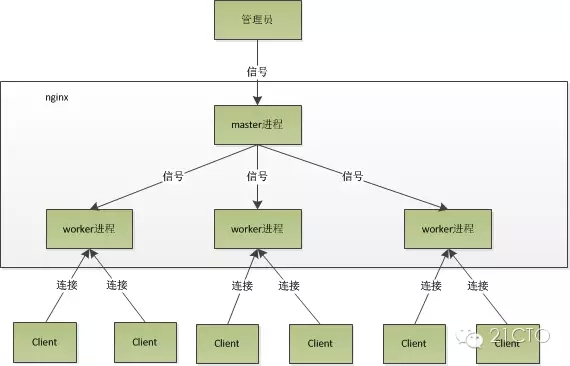
**master进程**

主要用来管理worker进程，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。

**worker进程：**

而基本的网络事件，则是放在worker进程中来处理了。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。

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。

### 2、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？，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。

