C++高并发服务器

目录

[1 阻塞型网络编程接口 2](#_Toc508611021)

[2 多线程服务器程序 3](#_Toc508611022)

[3 非阻塞服务器程序 5](#_Toc508611023)

[4 Select模型 6](#_Toc508611024)

[5 poll模型 13](#_Toc508611025)

[6 epoll模型 14](#_Toc508611026)

[7 libev 的服务器 18](#_Toc508611027)

# 1 阻塞型网络编程接口

几乎所有的程序员第一次接触到的网络编程都是从 listen()、send()、recv() 等接口开始的。使用这些接口可以很方便的构建服务器 / 客户机的模型。

我们假设希望建立一个简单的服务器程序，实现向单个客户机提供类似于“一问一答”的内容服务。

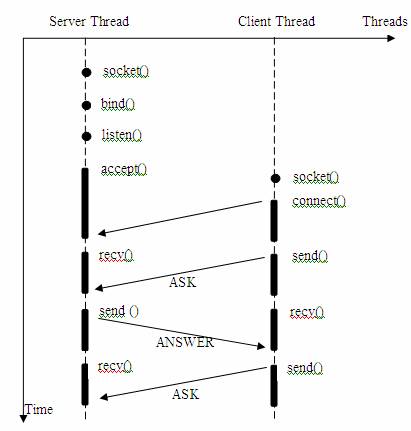


图 1. 简单的一问一答的服务器 / 客户机模型

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。这给多客户机、多业务逻辑的网络编程带来了挑战。这时，很多程序员可能会选择多线程的方式来解决这个问题。

文件夹normal为普通的客户端服务器socket代码示例。

# 2 多线程服务器程序

应对多客户机的网络应用，最简单的解决方式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。

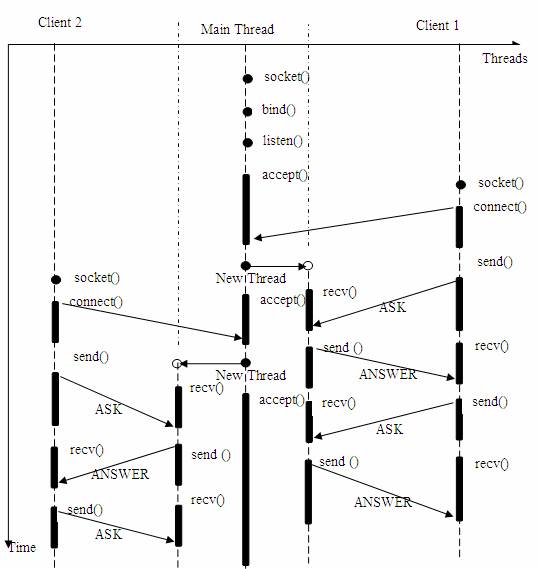
我们假设对上述的服务器 / 客户机模型，提出更高的要求，即让服务器同时为多个客户机提供一问一答的服务。于是有了如下的模型。

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 和各种数据库等。(附带线程池代码threadpool)

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，多线程模型并不是最佳方案。

文件夹fork中附带创建进程的server代码。

# 3 非阻塞服务器程序

以上面临的很多问题，一定程度是 IO 接口的阻塞特性导致的。多线程是一个解决方案，还一个方案就是使用非阻塞的接口。

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。

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。

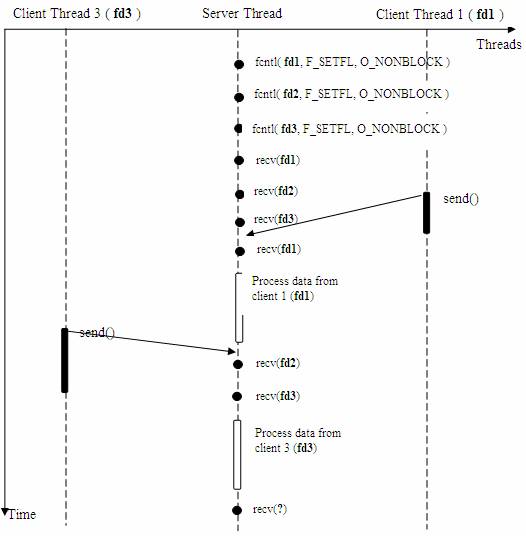


图 3. 使用非阻塞的接收数据模型

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()。

# 4 Select模型

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)

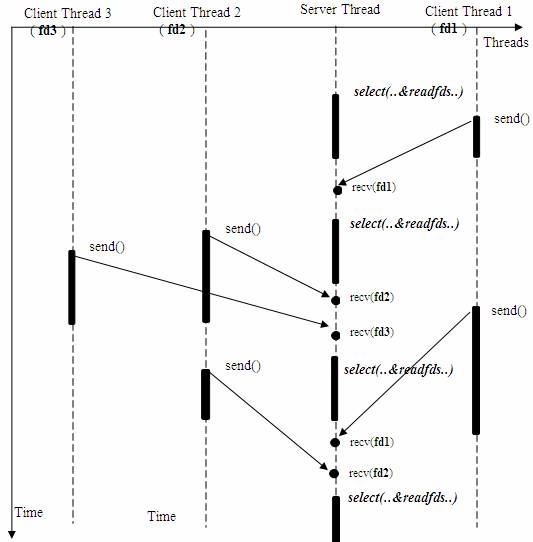
FD\_ISSET(int fd, fd\_set\* fds)

FD\_CLR(int fd, fd\_set\* fds)

I nt 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。



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。

上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。

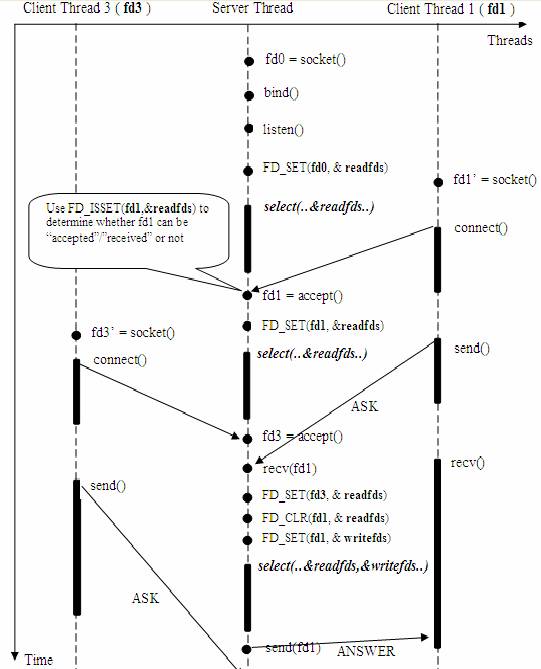


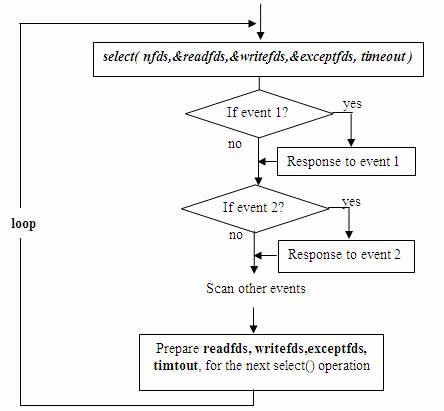
图 5. 使用 select() 接口的基于事件驱动的服务器模型

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。



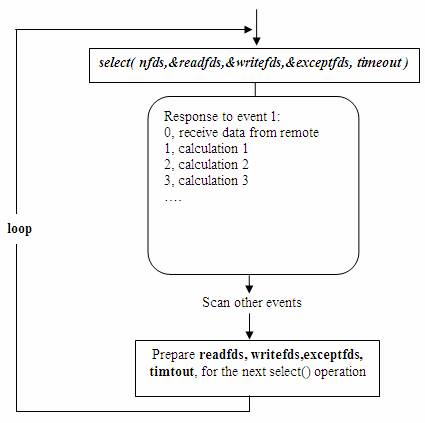
这种模型的特征在于每一个执行周期都会探测一次或一组事件，一个特定的事件会触发某个特定的响应。我们可以将这种模型归类为“事件驱动模型”。

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。



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。

nt select(int nfds, fd\_set \*readfds, fd\_set \*writefds,

fd\_set \*exceptfds, struct timeval \*timeout);

nfds: 监控的文件描述符集里最大文件描述符加1，因为此参数会告诉内核检测前多少个文件描述符的状态

readfds：监控有读数据到达文件描述符集合，传入传出参数

writefds：监控写数据到达文件描述符集合，传入传出参数

exceptfds：监控异常发生达文件描述符集合,如带外数据到达异常，传入传出参数

timeout：定时阻塞监控时间，3种情况：

1.NULL，永远等下去

2.设置timeval，等待固定时间

3.设置timeval里时间均为0，检查描述字后立即返回，轮询

struct timeval {

long tv\_sec; /\* seconds \*/

long tv\_usec; /\* microseconds \*/

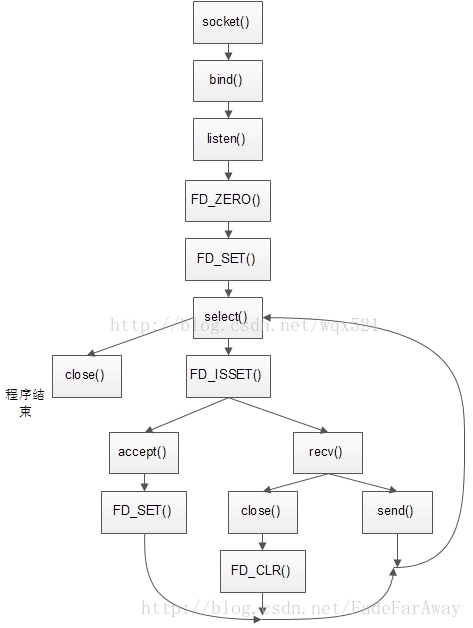
};

void FD\_CLR(int fd, fd\_set \*set); 把文件描述符集合里fd清0

int FD\_ISSET(int fd, fd\_set \*set); 测试文件描述符集合里fd是否置1

void FD\_SET(int fd, fd\_set \*set); 把文件描述符集合里fd位置1

void FD\_ZERO(fd\_set \*set); 把文件描述符集合里所有位清0



示例代码如 select 文件夹内所述

# 5 poll模型

poll提供的功能与select类似，它和select在本质上没有多大差别，但是poll没有最大文件描述符数量的限制，而且在处理流设备时，它能够提供额外的信息。

poll和select同样存在一个缺点就是，包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间，而不论这些文件描述符是否就绪，它的开销随着文件描述符数量的增加而线性增大。

另外，select()和poll()将就绪的文件描述符告诉进程后，如果进程没有对其进行IO操作，那么下次调用select()和poll()的时候将再次报告这些文件描述符，所以它们一般不会丢失就绪的消息，这种方式称为水平触发（Level Triggered）。

函数原型如下：

#include <poll.h>

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

struct pollfd {

int fd; /\* 文件描述符 \*/

short events; /\* 监控的事件 \*/

short revents; /\* 监控事件中满足条件返回的事件 \*/

};

#define POLLIN 0x0001 //不阻塞地可读高优先级外的数据

#define POLLRDNORM 0x0040 //不阻塞地可读普通数据

#define POLLRDBAND 0x0080 //不阻塞地可读非0优先级波段数据

#define POLLPRI 0x0002 //不阻塞地可读高优先级数据

#define POLLOUT 0x0004 //不阻塞地可写普通数据

#define POLLWRNORM 0x0100 //不阻塞地可写普通数据

#define POLLWRBAND 0x0200 //不阻塞地可写非0优先级波段数据

#define POLLERR 0x0008 //已出错

#define POLLHUP 0x0010 //已挂断

#define POLLNVAL 0x0020 //描述符不引用-打开文件

nfds 监控数组中有多少文件描述符需要被监控

timeout

-1 永远等待

0 不等待

>0 等待时间

poll文件夹中示例代码

# 6 epoll模型

epoll为处理大批量句柄而作了改进的poll。当然，这不是2.6内核才有的，它是在2.5.44内核中被引进的(epoll(4) is a new API introduced in Linuxkernel 2.5.44)，它几乎具备了之前所说的一切优点，被公认为Linux2.6下性能最好的多路I/O就绪通知方法。

epoll是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率，因为它会复用文件描述符集合来传递结果而不用迫使开发者每次等待事件之前都必须重新准备要被侦听的文件描述符集合，另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核IO事件异步唤醒而加入Ready队列的描述符集合就行了。

目前epell是linux大规模并发网络程序中的热门首选模型。

epoll除了提供select/poll那种IO事件的电平触发（Level Triggered）外，还提供了边沿触发（Edge Triggered只告诉进程哪些文件描述符刚刚变为就绪状态，它只说一遍，如果我们没有采取行动，那么它将不会再次告知，这种方式称为边缘触发），这就使得用户空间程序有可能缓存IO状态，减少epoll\_wait/epoll\_pwait的调用，提高应用程序效率。

epoll同样只告知那些就绪的文件描述符，而且当我们调用epoll\_wait()获得就绪文件描述符时，返回的不是实际的描述符，而是一个代表就绪描述符数量的值，你只需要去epoll指定的一个数组中依次取得相应数量的文件描述符即可，这里也使用了内存映射（mmap）技术，这样便彻底省掉了这些文件描述符在系统调用时复制的开销。

另一个本质的改进在于epoll采用基于事件的就绪通知方式。在select/poll中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描，而epoll事先通过epoll\_ctl()来注册一个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll\_wait()时便得到通知。

epoll有2种工作方式：LT和ET：

LT（level triggered）是缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表。

ET（edge-triggered）是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了（比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK 错误）。但是请注意，如果一直不对这个fd作IO操作（从而导致它再次变成未就绪），内核不会发送更多的通知（only once），不过在TCP协议中，ET模式的加速效用仍需要更多的benchmark确认。

ET和LT的区别就在这里体现，LT事件不会丢弃，而是只要读buffer里面有数据可以让用户读，则不断的通知你。而ET则只在事件发生之时通知。可以简单理解为LT是水平触发，而ET则为边缘触发。LT模式只要有事件未处理就会触发，而ET则只在高低电平变换时（即状态从1到0或者0到1）触发。

相关API：

1、创建一个epoll句柄，参数size用来告诉内核监听的文件描述符个数

#include <sys/epoll.h>

int epoll\_create(int size) size：告诉内核监听的数目

2、控制某个epoll监控的文件描述符上的事件：注册、修改、删除。

#include <sys/epoll.h>

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)

epfd：为epoll\_creat的句柄

op： 表示动作，用3个宏来表示：

EPOLL\_CTL\_ADD (注册新的fd到epfd)，

EPOLL\_CTL\_MOD (修改已经注册的fd的监听事件)，

EPOLL\_CTL\_DEL (从epfd删除一个fd)；

event：告诉内核需要监听的事件

struct epoll\_event {

\_\_uint32\_t events; /\* Epoll events \*/

epoll\_data\_t data; /\* User data variable \*/

};

其中上面的结构体中的events可选参数是：

EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；

EPOLLOUT：表示对应的文件描述符可以写；

EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；

EPOLLERR：表示对应的文件描述符发生错误；

EPOLLHUP：表示对应的文件描述符被挂断；

EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(LevelTriggered)来说的。

EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

其中上面结构体中的epoll\_data\_t结构体为：

typedef union epoll\_data{

void \*ptr;

int fd;

uint32\_t u32;

uint64\_t u64;

} epoll\_data\_t;

一般会把注册的文件描述符放进去，这样当事件被激活时，返回epoll\_event结构体,这样在epoll\_event的data中就可以知道是哪一个被激活了。

3、等待所监控文件描述符上有事件的产生，类似于select()调用。

#include <sys/epoll.h>

int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout)

events：用来从内核得到事件的集合，

maxevents：告之内核这个events有多大，这个maxevents的值不能大于创建epoll\_create()时的size，

timeout：是超时时间

-1：阻塞

0：立即返回，非阻塞

>0：指定微秒

返回值：成功返回有多少文件描述符就绪，时间到时返回0，出错返回-1

查看一个进程打开大数目的socket描述符：

cat /proc/sys/fs/file-max

设置最大打开文件描述符限制

sudo vi /etc/security/limits.conf

写入以下配置,soft软限制，hard硬限制

\* 　　　　soft 　　　nofile 　　　65536

\*　　　　 hard　　　nofile 　　　100000

epoll实例程序如epoll中server.cpp

# 7 libev 服务器

libev 同样需要循环探测事件是否产生。Libev 的循环体用 ev\_loop 结构来表达，并用 ev\_loop( ) 来启动。

void ev\_loop( ev\_loop\* loop, int flags )

Libev 支持八种事件类型，其中包括 IO 事件。一个 IO 事件用 ev\_io 来表征，并用 ev\_io\_init() 函数来初始化：

void ev\_io\_init(ev\_io \*io, callback, int fd, int events)

初始化内容包括回调函数 callback，被探测的句柄 fd 和需要探测的事件，EV\_READ 表“可读事件”，EV\_WRITE 表“可写事件”。

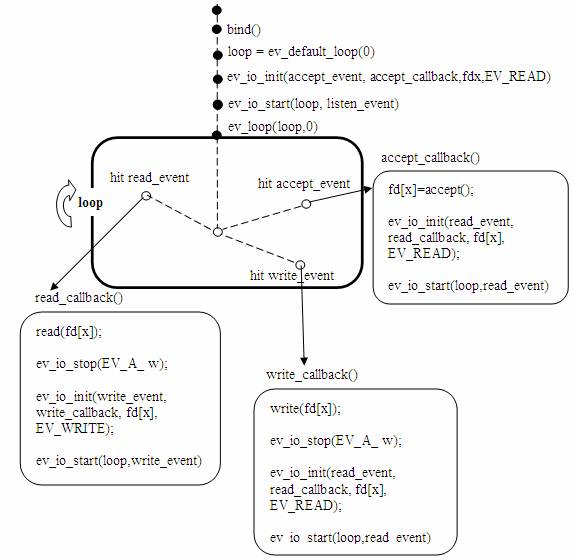
现在，用户需要做的仅仅是在合适的时候，将某些 ev\_io 从 ev\_loop 加入或剔除。一旦加入，下个循环即会检查 ev\_io 所指定的事件有否发生；如果该事件被探测到，则 ev\_loop 会自动执行 ev\_io 的回调函数 callback()；如果 ev\_io 被注销，则不再检测对应事件。

无论某 ev\_loop 启动与否，都可以对其添加或删除一个或多个 ev\_io，添加删除的接口是 ev\_io\_start() 和 ev\_io\_stop()。

void ev\_io\_start( ev\_loop \*loop, ev\_io\* io )

void ev\_io\_stop( EV\_A\_\* )

由此，我们可以容易得出如下的“一问一答”的服务器模型。由于没有考虑服务器端主动终止连接机制，所以各个连接可以维持任意时间，客户端可以自由选择退出时机。



上述模型可以接受任意多个连接，且为各个连接提供完全独立的问答服务。借助 libev 提供的事件循环 / 事件驱动接口，上述模型有机会具备其他模型不能提供的高效率、低资源占用、稳定性好和编写简单等特点。

由于传统的 web 服务器，ftp 服务器及其他网络应用程序都具有“一问一答”的通讯逻辑，所以上述使用 libev 库的“一问一答”模型对构建类似的服务器程序具有参考价值；另外，对于需要实现远程监视或远程遥控的应用程序，上述模型同样提供了一个可行的实现方案。

6 select/pool