使用 libevent 和 libev 提高网络应用性能——I/O模型演进变化史 - yutingliuyl

yutingliuyl 关注 - 0 粉丝 - 8 +加关注

构建现代的server应用程序须要以某种方法同一时候接收数百、数千甚至数万个事件，不管它们是内部请求还是网络连接，都要有效地处理它们的操作。

有很多解决方式，但事件驱动也被广泛应用到网络编程中。并大规模部署在高连接数高吞吐量的server程序中，如 http server程序、ftp server程序等。

相比于传统的网络编程方式，事件驱动可以极大的减少资源占用，增大服务接待能力，并提高网络传输效率。

这些事件驱动模型中， libevent 库和 libev库可以大大提高性能和事件处理能力。

在本文中。我们要讨论在 UNIX/Linux 应用程序中使用和部署这些解决方式所用的基本结构和方法。

libev 和 libevent 都能够在高性能应用程序中使用。

在讨论libev 和 libevent之前，我们看看I/O模型演进变化历史

1、堵塞网络接口：处理单个client

我们 第一次接触到的网络编程一般都是从 listen()、send()、recv()等接口開始的。使用这些接口能够非常方便的构建server /客户机的模型。

堵塞I/O模型图：在调用recv()函数时，发生在内核中等待数据和复制数据的过程。

当调用recv()函数时。系统首先查是否有准备好的数据。假设数据没有准备好，那么系统就处于等待状态。当数据准备好后，将数据从系统缓冲区拷贝到用户空间，然后该函数返回。在套接应用程序中，当调用recv()函数时，未必用户空间就已经存在数据，那么此时recv()函数就会处于等待状态。

我们注意到。大部分的 socket 接口都是堵塞型的。所谓堵塞型接口是指系统调用（通常是 IO 接口）不返回调用结果并让当前线程一直堵塞，仅仅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，差点儿全部的 IO 接口 (包含 socket 接口) 都是堵塞型的。这给网络编程带来了一个非常大的问题。如在调用 send() 的同一时候，线程将被堵塞，在此期间。线程将无法运行不论什么运算或响应不论什么的网络请求。这给多客户机、多业务逻辑的网络编程带来了挑战。这时。非常多程序猿可能会选择多线程的方式来解决问题。

使用堵塞模式的套接字，开发网络程序比較简单。easy实现。

当希望可以马上发送和接收数据。且处理的套接字数量比較少的情况下。即一个一个处理client，server没什么压力。使用堵塞模式来开发网络程序比較合适。

堵塞模式给网络编程带来了一个非常大的问题，如在调用 send()的同一时候。线程将被堵塞，在此期间，线程将无法运行不论什么运算或响应不论什么的网络请求。

假设非常多client同一时候訪问server，server就不能同一时候处理这些请求。这时，我们可能会选择

多线程的方式来解决问题。

2、多线程/进程处理多个client

应对多客户机的网络应用，最简单的解决方案是在server端使用多线程（或多进程）。多线程（或多进程）的目的是让每一个连接都拥有独立的线程（或进程），这样不论什么一个连接的堵塞都不会影响其它的连接。

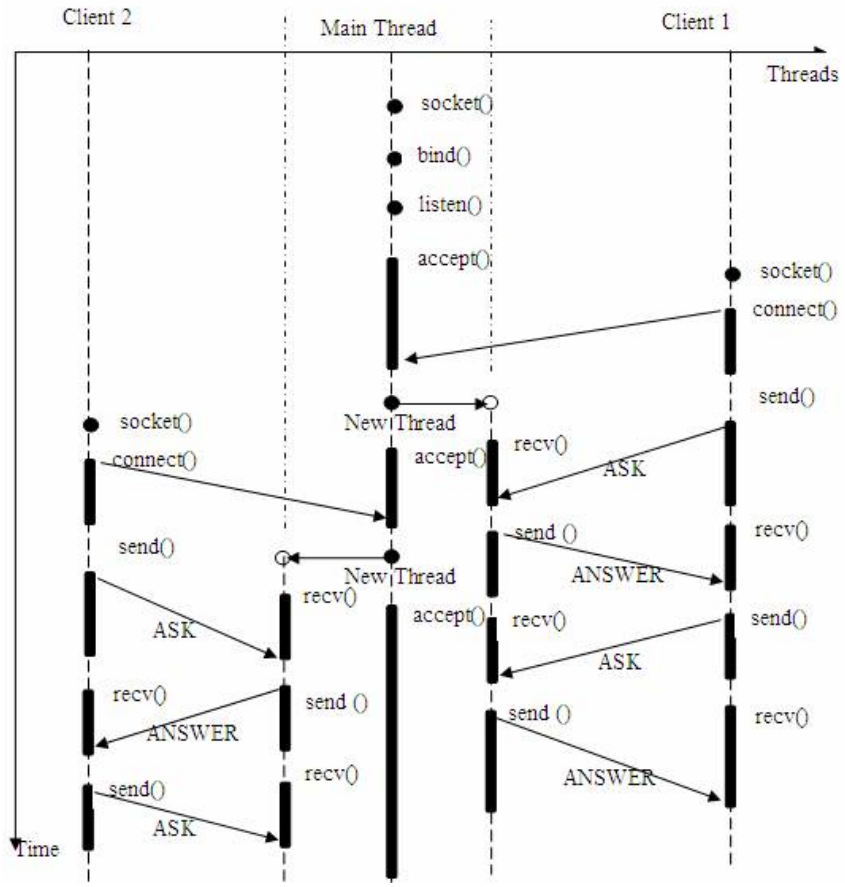
详细使用多进程还是多线程，并没有一个特定的模式。

传统意义上，进程的开销要远远大于线程，所以，假设须要同一时候为较多的客户机提供服务。则不推荐使用多进程；假设单个服务运行体须要消耗较多的 CPU 资源，譬如须要进行大规模或长时间的数据运算或文件訪问，则进程较为安全。通常，使用 pthread_create () 创建新线程。fork() 创建新进程。即：

(1) a new Connection 进来。用 fork() 产生一个 Process 处理。

(2) a new Connection 进来。用 pthread_create() 产生一个 Thread 处理。

多线程/进程server同一时候为多个客户机提供应答服务。模型例如以下：



主线程持续等待client的连接请求，假设有连接，则创建新线程，并在新线程中提供为前例相同的问答服务。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>
void do_service(int conn);
void err_log(string err, int sockfd) {
        perror("binding");      close(sockfd);          exit(-1);
}
int main(int argc, char *argv[])
{
        unsigned short port = 8000;
        int sockfd;
        sockfd = socket(AF_INET, SOCK_STREAM, 0);// 创建通信端点：套接字
        if(sockfd < 0) {
                perror("socket");
                exit(-1);
        }

        struct sockaddr_in my_addr;
        bzero(&my_addr, sizeof(my_addr));
        my_addr.sin_family = AF_INET;
        my_addr.sin_port   = htons(port);
        my_addr.sin_addr.s_addr = htonl(INADDR_ANY);

        int err_log = bind(sockfd, (struct sockaddr*)&my_addr, sizeof(my_addr));
        if( err_log != 0)   err_log("binding");
        err_log = listen(sockfd, 10);
        if(err_log != 0) err_log("listen");

        struct sockaddr_in peeraddr; //传出参数
    socklen_t peerlen = sizeof(peeraddr); //传入传出参数。必须有初始值
    int conn; // 已连接套接字(变为主动套接字，即能够主动connect)
        pid_t pid;
    while (1) {
        if ((conn = accept(sockfd, (struct sockaddr *)&peeraddr, &peerlen)) < 0) //3次握手完毕的序列
            err_log("accept error");
        printf("recv connect ip=%s port=%d/n", inet_ntoa(peeraddr.sin_addr),ntohs(peeraddr.sin_port));

        pid = fork();
        if (pid == -1)
            err_log("fork error");
        if (pid == 0) {// 子进程
            close(listenfd);
            do_service(conn);
            exit(EXIT_SUCCESS);
        }
        else
            close(conn); //父进程
    }
        return 0;
}

void do_service(int conn) {
    char recvbuf[1024];
    while (1)  {
        memset(recvbuf, 0, sizeof(recvbuf));
        int ret = read(conn, recvbuf, sizeof(recvbuf));
        if (ret == 0)    { //客户端关闭了
            printf("client close/n");
            break;
        }
        else if (ret == -1)
            ERR_EXIT("read error");
        fputs(recvbuf, stdout);
        write(conn, recvbuf, ret);
    }
}
```

非常多刚開始学习的人可能不明确为何一个 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() 可以返回一个新的 socket。以下是 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。运行完 bind() 和 listen() 后，操作系统已经開始在指定的port处监听全部的连接请求。假设有请求。则将该连接请求增加请求队列。

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 句柄即是兴许 read() 和 recv() 的输入参数。假设请求队列当前没有请求。则 accept() 将进入堵塞状态直到有请求进入队列。

上述多线程的server模型似乎完美的攻克了为多个客户机提供问答服务的要求，但事实上并不尽然。假设要同一时候响应成百上千路的连接请求，则不管多线程还是多进程都会严重占领系统资源，减少系统对外界响应效率。而线程与进程本身也更easy进入假死状态。

因此其缺点：

1）用 fork() 的问题在于每个 Connection 进来时的成本太高,假设同一时候接入的并发连接数太多easy进程数量非常多,进程之间的切换开销会非常大,同一时候对于老的内核(Linux)会产生雪崩效应。

2）用 Multi-thread 的问题在于 Thread-safe 与 Deadlock 问题难以解决。另外有 Memory-leak 的问题要处理,这个问题对于非常多程序猿来说无异于恶梦,尤其是对于连续server的server程序更是不能够接受。 假设才用 Event-based 的方式在于实做上不好写，尤其是要注意到事件产生时必须 Nonblocking。于是会须要实做 Buffering 的问题。而 Multi-thread 所会遇到的 Memory-leak 问题在这边会更严重。

而在多 CPU 的系统上没有办法使用到全部的 CPU resource。

由此可能会考虑使用“线程池”或“连接池”。

“线程池”旨在降低创建和销毁线程的频率，其维持一定合理数量的线程。并让空暇的线程又一次承担新的运行任务。“连接池”维持连接的缓存池。尽量重用已有的连接、降低创建和关闭连接的频率。这两种技术都能够非常好的降低系统开销，都被广泛应用非常多大型系统。如apache，mysql数据库等。

可是，“线程池”和“连接池”技术也仅仅是在一定程度上缓解了频繁调用 IO 接口带来的资源占用。并且。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，并依据响应规模调整“池”的大小。

相应上例中的所面临的可能同一时候出现的上千甚至上万次的client请求，“线程池”或“连接池”也许能够缓解部分压力，可是不能解决全部问题。

由于多线程/进程导致过多的占用内存或 CPU等系统资源。

3、非堵塞的server模型

以上面临的非常多问题，一定程度是 IO 接口的堵塞特性导致的。

多线程是一个解决方式，还一个方案就是使用非堵塞的接口。

非堵塞的接口相比于堵塞型接口的显著差异在于。在被调用之后马上返回。

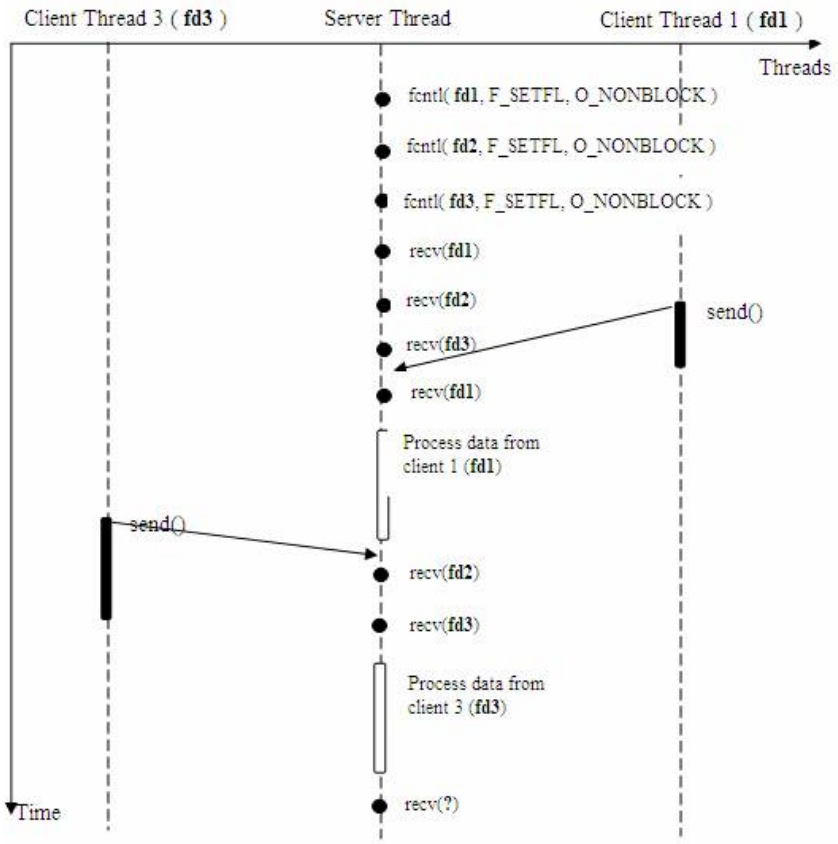
使用例如以下的函数能够将某句柄 fd 设为非堵塞状态。

我们能够使用 fcntl(fd, F_SETFL, flag | O_NONBLOCK); 将套接字标志变成非堵塞：

```
fcntl( fd, F_SETFL, O_NONBLOCK );
```

以下将给出仅仅用一个线程。但可以同一时候从多个连接中检测数据是否送达，而且接受数据。

使用非堵塞的接收数据模型:



在非堵塞状态下，recv() 接口在被调用后马上返回，返回值代表了不同的含义。

调用recv。假设设备临时没有数据可读就返回-1，同一时候置errno为EWOULDBLOCK（或者EAGAIN，这两个宏定义的值同样）。表示本来应该堵塞在这里（would block，虚拟语气），其实并没有堵塞而是直接返回错误，调用者应该试着再读一次（again）。

这样的行为方式称为轮询（Poll）。调用者仅仅是查询一下。而不是堵塞在这里死等

如在本例中，

- recv() 返回值大于 0，表示接受数据完成，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 操作还没运行完毕；
- recv() 返回 -1。且 errno 不等于 EAGAIN，表示 recv 操作遇到系统错误 errno。

这样能够同一时候监视多个设备：

```
while(1){
 非堵塞read(设备1);
 if(设备1有数据到达)
    处理数据;
 非堵塞read(设备2);
 if(设备2有数据到达)
```

处理数据;

.....

}

假设read(设备1)是堵塞的，那么仅仅要设备1没有数据到达就会一直堵塞在设备1的read调用上，即使设备2有数据到达也不能处理，使用非堵塞I/O就能够避免设备2得不到及时处理。

类似一个快递的样例：这里使用忙轮询的方法：每隔1微妙（while(1)差点儿不间断）到A楼一层(内核缓冲区）去看快递来了没有。假设没来。马上返回。

而快递来了，就放在A楼一层，等你去取。

非堵塞I/O有一个缺点，假设全部设备都一直没有数据到达，调用者须要重复查询做无用功，假设堵塞在那里。操作系统能够调度别的进程运行，就不会做无用功了。在实际应用中非堵塞I/O模型比较少用。

能够看到server线程能够通过循环调用 recv() 接口。能够在单个线程内实现对全部连接的数据接收工作。

可是上述模型绝不被推荐。由于。循环调用 recv() 将大幅度推高 CPU 占用率。此外。在这个方法中，recv() 很多其它的是起到检测“操作是否完毕”的作用，实际操作系统提供了更为高效的检测“操作是否完毕“作用的接口。比如 select()。

4、IO复用事件驱动server模型

简单介绍：主要是select和epoll；对一个IOport，两次调用，两次返回。比堵塞IO并没有什么优越性。关键是能实现同一时候对多个IOport进行监听；

I/O复用模型会用到select、poll、epoll函数。这几个函数也会使进程堵塞，可是和堵塞I/O所不同的的，这两个函数能够同一时候堵塞多个I/O操作。并且能够同一时候对多个读操作，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。

我们先具体解释select：

SELECT函数进行IO复用server模型的原理是：当一个client连接上server时。server就将其连接的fd增加fd_set集合，等到这个连接准备好读或写的时候，就通知程序进行IO操作，与client进行数据通信。

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。

4.1 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 maxfdp, //Winsock中此參数无意义
     fd_set* readfds, //进行可读检测的Socket
     fd_set* writefds, //进行可写检测的Socket
     fd_set* exceptfds, //进行异常检测的Socket
     const struct timeval* timeout //非堵塞模式中设置最大等待时间
)
```

参数列表:

int maxfdp :是一个整数值，意思是“最大fd加1（max fd plus 1）. 在三个描写叙述符集(readfds, writefds, exceptfds)中找出最高描写叙述符

编号值，然后加 1也可将maxfdp设置为 FD_SETSIZE。这是一个< sys/types.h >中的常数，它说明了最大的描写叙述符数（常常是 256或1024）。

可是对大多数应用程序而言，此值太大了。

确实，大多数应用程序仅仅应用 3 ~ 1 0个描写叙述符。假设将第三个参数设置为最高描写叙述符编号值加 1，内核就仅仅需在此范围内寻找打开的位。而不必在数百位的大范围内搜索。

fd_set *readfds: 是指向fd_set结构的指针，这个集合中应该包含文件描写叙述符，我们是要监视这些文件描写叙述符的读变化的。即我们关

心能否够从这些文件里读取数据了，假设这个集合中有一个文件可读，select就会返回一个大于0的值。表示有文件可读，假设没有可读的文件。则依据timeout参数再推断是否超时，若超出timeout的时间，select返回0，若错误发生返回负值。能够传入NULL值。表示不关心不论什么文件的读变化。

fd_set *writefds: 是指向fd_set结构的指针，这个集合中应该包含文件描写叙述符，我们是要监视这些文件描写叙述符的写变化的，即我们关

心能否够向这些文件里写入数据了，假设这个集合中有一个文件可写，select就会返回一个大于0的值，表示有文件可写，假设没有可写的文件，则依据timeout参数再推断是否超时，若超出timeout的时间，select返回0。若错误发生返回负值。能够传入NULL值，表示不关心不论什么文件的写变化。

fd_set *errorfds: 同上面两个参数的意图，用来监视文件错误异常。

readfds , writefds,*errorfds每一个描写叙述符集存放在一个fd_set 数据类型中.如图：

struct timeval* timeout :是select的超时时间，这个參数至关重要。它能够使select处于三种状态：
第一，若将NULL以形参传入，即不传入时间结构，就是将select置于堵塞状态，一定等到监视文件描写叙述符集合中某个文件描写叙述符发生变化为止。
第二，若将时间值设为0秒0毫秒，就变成一个纯粹的非堵塞函数，无论文件描写叙述符是否有变化，都立马返回继续运行。文件无变化返回0，有变化返回一个正值；
第三，timeout的值大于0，这就是等待的超时时间，即 select在timeout时间内堵塞。超时时间之内有事件到来就返回了。否则在超时后无论如何一定返回，返回值同上述。

4.2 使用select库的步骤是

（1）创建所关注的事件的描写叙述符集合（fd_set)，对于一个描写叙述符。能够关注其上面的读(read)、写(write)、异常(exception)事件，所以通常，要创建三个fd_set， 一个用来收集关注读事件的描写叙述符，一个用来收集关注写事件的描写叙述符。另外一个用来收集关注异常事件的描写叙述符集合。

（2）调用select(),等待事件发生。

这里须要注意的一点是，select的堵塞与是否设置非堵塞I/O是没有关系的。

（3）轮询全部fd_set中的每个fd 。检查是否有对应的事件发生。假设有，就进行处理。

```
/* 可读、可写、异常三种文件描写叙述符集的申明和初始化。

*/
fd_set readfds, writefds, exceptionfds;
FD_ZERO(&readfds);
FD_ZERO(&writefds);
FD_ZERO(&exceptionfds);

int max_fd;
```

```
/* socket配置和监听。

*/
sock = socket(...);
bind(sock, ...);
listen(sock, ...);

/* 对socket描写叙述符上发生关心的事件进行注冊。

*/
FD_SET(&readfds, sock);
max_fd = sock;

while(1) {
        int i;
        fd_set r,w,e;

        /* 为了反复使用readfds 、writefds、exceptionfds，将它们复制到暂时变量内。*/
        memcpy(&r, &readfds, sizeof(fd_set));
        memcpy(&w, &writefds, sizeof(fd_set));
        memcpy(&e, &exceptionfds, sizeof(fd_set));

        /* 利用暂时变量调用select()堵塞等待。timeout=null表示等待时间为永远等待直到发生事件。*/
        select(max_fd + 1, &r, &w, &e, NULL);

        /* 测试是否有client发起连接请求，假设有则接受并把新建的描写叙述符增加监控。*/
        if(FD_ISSET(&r, sock)){
                new_sock = accept(sock, ...);
                FD_SET(&readfds, new_sock);
                FD_SET(&writefds, new_sock);
                max_fd = MAX(max_fd, new_sock);
        }
        /* 对其他描写叙述符发生的事件进行适当处理。描写叙述符依次递增，最大值各系统有所不同（比方在作者系统上最大为1024）。
        在linux能够用命令ulimit -a查看（用ulimit命令也对该值进行改动）。


        在freebsd下，用sysctl -a | grep kern.maxfilesperproc来查询和改动。

*/

        for(i= sock+1; i <max_fd+1; ++i) {
                if(FD_ISSET(&r, i))
                        doReadAction(i);
                if(FD_ISSET(&w, i))
                        doWriteAction(i);
        }
}
```

4.3 和select模型紧密结合的四个宏

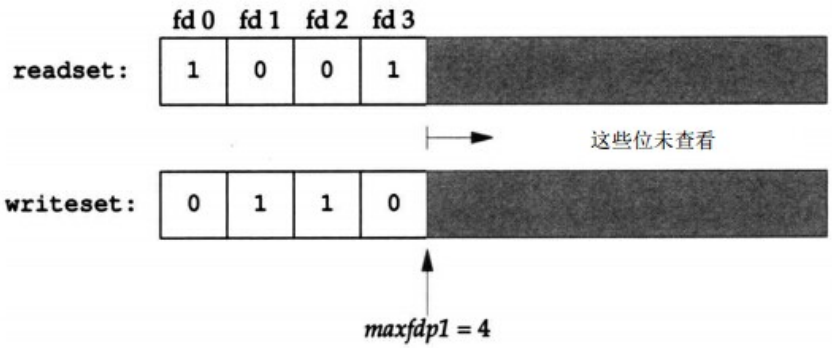
FD_ZERO(int fd, fd_set* fds) //清除其全部位
FD_SET(int fd, fd_set* fds) //在某 fd_set 中标记一个fd的相应位为1
FD_ISSET(int fd, fd_set* fds) // 测试该集中的一个给定位是否仍旧设置
FD_CLR(int fd, fd_set* fds) //删除相应位

这里，fd_set 类型能够简单的理解为按 bit 位标记句柄的队列，比如要在某 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。详细的置位、验证可使用 FD_SET、FD_ISSET 等宏实现。

比如，编写下列代码：

```
fd_setreadset,writeset;
FD_ZERO(&readset);
FD_ZERO(&writeset);
FD_SET(0,&readset);
FD_SET(3,&readset);
FD_SET(1,&writeset);
FD_SET(2,&writeset);
select(4,&readset,&writeset,NULL,NULL);
```

然后，下图显示了这两个描写叙述符集的情况：



由于描写叙述符编号从0開始，所以要在最大描写叙述符编号值上加1。

第一个参数实际上是要检查的描写叙述符数（从描写叙述符0開始）。

4.4 select有三个可能的返回值

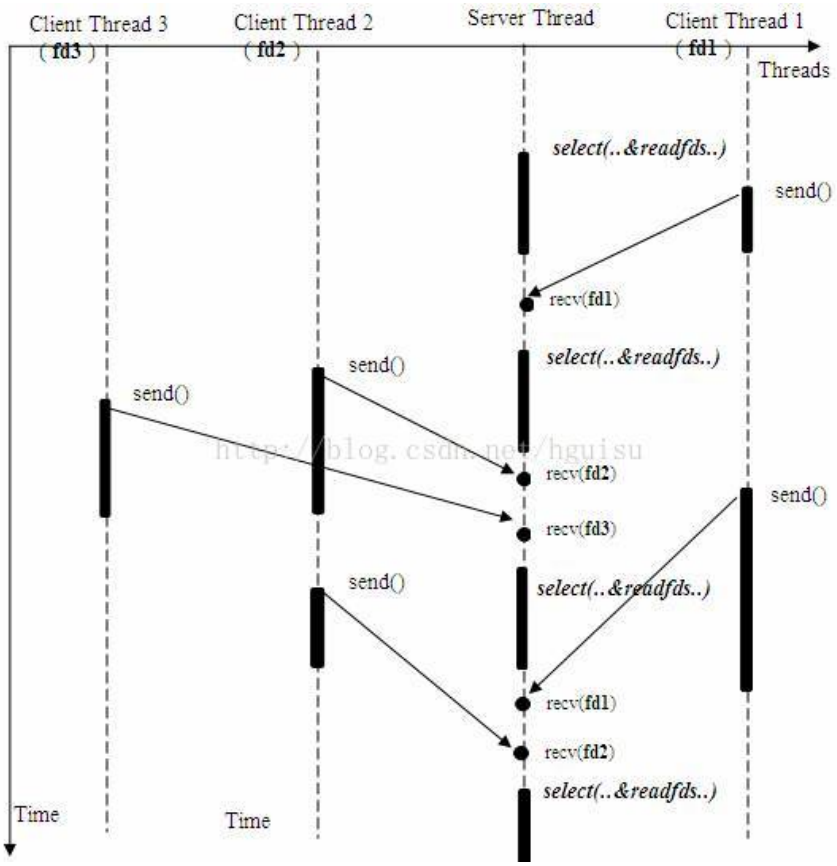
(1)返回值 - 1表示出错。这是可能发生的，比如在所指定的描写叙述符都没有准备好时捕捉到一个信号。
(2)返回值0表示没有描写叙述符准备好。

若指定的描写叙述符都没有准备好，并且指定的时间已经超过。则发生这样的情况。
(3)返回一个正值说明了已经准备好的描写叙述符数，在这样的情况下，三个描写叙述符集中仍旧打开的位是相应于已准备好的描写叙述符位。

4.5 使用**select()**的接收数据模型图：

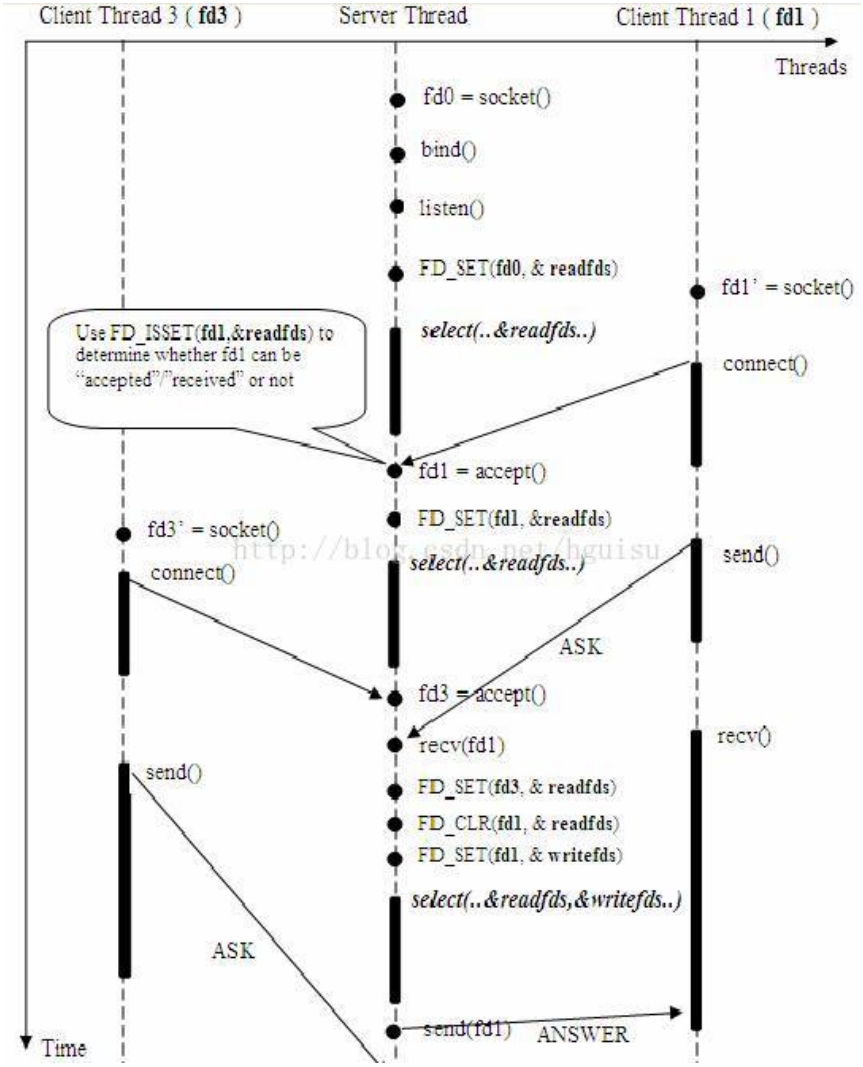
以下将又一次模拟上例中从多个client接收数据的模型。

使用select()的接收数据模型



上述模型仅仅是描写叙述了使用 select() 接口同一时候从多个client接收数据的过程；因为 select() 接口能够同一时候对多个句柄进行读状态、写状态和错误状态的探测。所以能够非常easy构建为多个client提供独立问答服务的server系统。

使用select()接口的基于事件驱动的server模型



这里须要指出的是。client的一个 connect() 操作，将在server端激发一个“可读事件”，所以 select() 也能探测来自client的 connect() 行为。

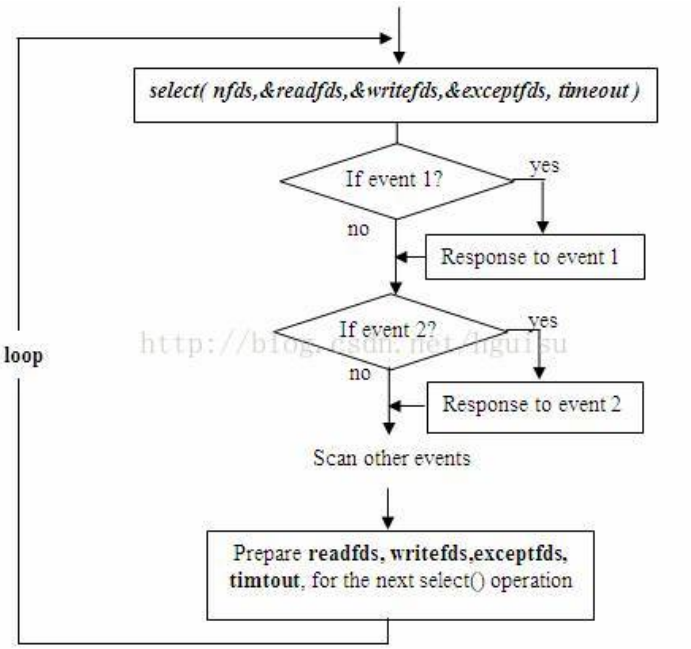
上述模型中，最关键的地方是怎样动态维护 select() 的三个参数 readfds、writefds 和 exceptfds。作为输入参数，readfds 应该标记全部的须要探测的“可读事件”的句柄，当中永远包含那个探测 connect() 的那个“母”句柄；同一时候，writefds 和 exceptfds 应该标记全部须要探测的“可写事件”和“错误事件”的句柄 (使用 FD_SET() 标记)。

作为输出参数，readfds、writefds 和 exceptfds 中的保存了 select() 捕捉到的全部事件的句柄值。程序猿须要检查的全部的标记位 (使用 FD_ISSET() 检查)，以确定究竟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，假设 select() 发现某句柄捕捉到了“可读事件”，server程序应及时做 recv() 操作。并依据接收到的数据准备好待发送数据。并将相应的句柄值增加 writefds，准备下一次的“可写事件”的 select() 探测。相同，假设 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。

下图描写叙述的是上述模型中的一个运行周期。

一个运行周期



这样的模型的特征在于每个运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。我们能够将这样的模型归类为“事件驱动模型”。

4.6 select的优缺点

相比其它模型，使用 select() 的事件驱动模型仅仅用单线程（进程）运行，占用资源少，不消耗太多 CPU，同一时候可以为多client提供服务。

假设试图建立一个简单的事件驱动的server程序，这个模型有一定的参考价值。但这个模型依然有着非常多问题。

select的缺点:

（1）单个进程可以监视的文件描写叙述符的数量存在最大限制

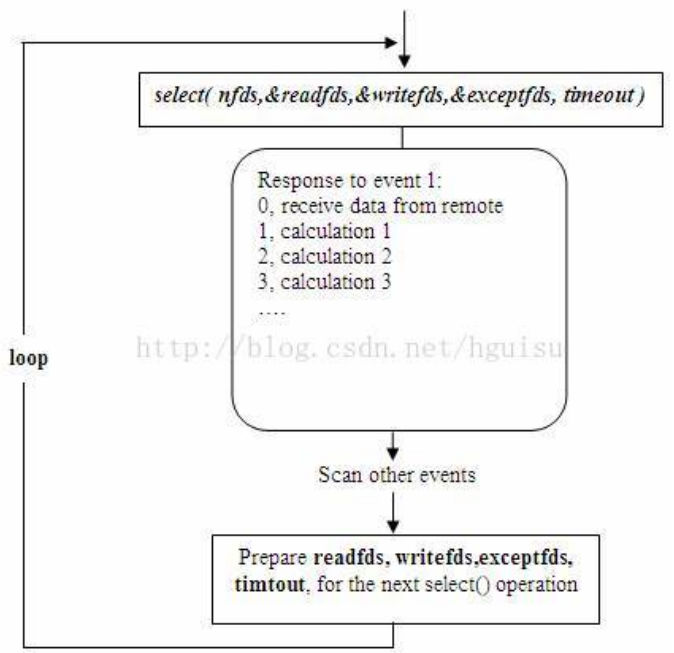
（2）select须要复制大量的句柄数据结构。产生巨大的开销

（3）select返回的是含有整个句柄的列表，应用程序须要消耗大量时间去轮询各个句柄才干发现哪些句柄发生了事件

（4）select的触发方式是水平触发，应用程序假设没有完毕对一个已经就绪的文件描写叙述符进行IO操作，那么之后每次select调用还是会将这些文件描写叙述符通知进程。相相应方式的是边缘触发。

(6) 该模型将事件探测和事件响应夹杂在一起。一旦事件响应的运行体庞大，则对整个模型是灾难性的。例如以下例。庞大的运行体 1 的将直接导致响应事件 2 的运行体迟迟得不到运行，并在非常大程度上减少了事件探测的及时性。

庞大的运行体对使用select()的事件驱动模型的影响



非常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 …。

假设须要实现更高效的server程序，类似 epoll 这种接口更被推荐。

4.7 poll事件模型

poll库是在linux2.1.23中引入的。windows平台不支持poll. poll与select的基本方式同样。都是先创建一个关注事件的描写叙述符的集合，然后再去等待这些事件发生。然后再轮询描写叙述符集合，检查有没有事件发生。假设有，就进行处理。

因此。poll有着与select相似的处理流程：

（1）创建描写叙述符集合，设置关注的事件

（2）调用poll()。等待事件发生。以下是poll的原型：

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

类似select，poll也能够设置等待时间，效果与select一样。

（3）轮询描写叙述符集合，检查事件。处理事件。

在这里要说明的是。poll与select的主要差别在与，select须要为读、写、异常事件分别创建一个描写叙述符集合，最后轮询的时候，须要分别轮询这三个集合。而poll仅仅须要一个集合，在每一个描写叙述符相应的结构上分别设置读、写、异常事件，最后轮询的时候。能够同一时候检查三种事件。

4.7 epoll事件模型

epoll是和上面的poll和select不同的一个事件驱动库，它是在linux 2.5.44中引入的。它属于poll的一个变种。

poll和select库。它们的最大的问题就在于效率。它们的处理方式都是创建一个事件列表，然后把这个列表发给内核，返回的时候，再去轮询检查这个列表，这样在描写叙述符比较多的应用中。效率就显得比较低下了。

epoll是一种比较好的做法，它把描写叙述符列表交给内核。一旦有事件发生。内核把发生事件的描写叙述符列表通知给进程，这样就避免了轮询整个描写叙述符列表。

以下对epoll的使用进行说明：

（1）.创建一个epoll描写叙述符，调用epoll_create()来完毕，epoll_create()有一个整型的參数size，用来告诉内核。要创建一个有size个描写叙述符的事件列表（集合）

int epoll_create(int size)

（2）.给描写叙述符设置所关注的事件，并把它加入到内核的事件列表中去，这里须要调用epoll_ctl()来完毕。

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)

这里op參数有三种。分别代表三种操作：

a. EPOLL_CTL_ADD, 把要关注的描写叙述符和对其关注的事件的结构。加入到内核的事件列表中去

b. EPOLL_CTL_DEL，把先前加入的描写叙述符和对其关注的事件的结构，从内核的事件列表中去除

c. EPOLL_CTL_MOD，改动先前加入到内核的事件列表中的描写叙述符的关注的事件

（3）. 等待内核通知事件发生，得到发生事件的描写叙述符的结构列表。该过程由epoll_wait()完毕。得到事件列表后，就能够进行事件处理了。

int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout)

在使用epoll的时候，有一个须要特别注意的地方。那就是epoll触发事件的文件有两种方式：

（1）Edge Triggered(ET)，在这样的情况下，事件是由数据到达边界触发的。所以要在处理读、写的时候，要不断的调用read/write。直到它们返回EAGAIN。然后再去epoll_wait(),等待下次事件的发生。这样的方式适用要遵从以下的原则：

a. 使用非堵塞的I/O。b.直到read/write返回EAGAIN时，才去等待下一次事件的发生。

（2）Level Triggered(LT), 在这样的情况下，epoll和poll类似，但处理速度上可能比poll快。

在这样的情况下，仅仅要有数据没有读、写完，调用epoll_wait()的时候，就会有事件被触发。

```
/* 新建并初始化文件描写叙述符集。

*/
struct epoll_event ev;
struct epoll_event events[MAX_EVENTS];

/* 创建epoll句柄。*/
int epfd = epoll_create(MAX_EVENTS);

/* socket配置和监听。*/
sock = socket(...);
bind(sock, ...);
listen(sock, ...);

/* 对socket描写叙述符上发生关心的事件进行注冊。*/
ev.events = EPOLLIN;
ev.data.fd = sock;
epoll_ctl(epfd, EPOLL_CTL_ADD, sock, &ev);

while(1) {
        int i;
        /*调用epoll_wait()堵塞等待。等待时间为永远等待直到发生事件。*/
        int n = epoll_wait(epfd, events, MAX_EVENTS, -1);
```

```
        for(i=0; i <n; ++i) {
                /* 测试是否有client发起连接请求，假设有则接受并把新建的描写叙述符增加监控。*/
                if(events.data.fd == sock) {
                        if(events.events & POLLIN){
                                new_sock = accept(sock, ...);
                                ev.events = EPOLLIN | POLLOUT;
                                ev.data.fd = new_sock;
                                epoll_ctl(epfd, EPOLL_CTL_ADD, new_sock, &ev);
                        }
                }else{
                /* 对其他描写叙述符发生的事件进行适当处理。*/
                        if(events.events & POLLIN)
                                doReadAction(i);
                        if(events.events & POLLOUT)
                                doWriteAction(i);
                }
        }
}
```

epoll支持水平触发和边缘触发，理论上来说边缘触发性能更高。可是使用更加复杂，由于不论什么意外的丢失事件都会造成请求处理错误。Nginx就使用了epoll的边缘触发模型。

这里提一下水平触发和边缘触发就绪通知的差别：

这两个词来源于计算机硬件设计。

它们的差别是仅仅要句柄满足某种状态，水平触发就会发出通知。而仅仅有当句柄状态改变时。边缘触发才会发出通知。比如一个socket经过长时间等待后接收到一段100k的数据，两种触发方式都会向程序发出就绪通知。如果程序从这个socket中读取了50k数据。并再次调用监听函数，水平触发依旧会发出就绪通知，而边缘触发会由于socket“有数据可读”这个状态没有发生变化而不发出通知且陷入长时间的等待。
因此在使用边缘触发的 api 时，要注意每次都要读到 socket返回 EWOULDBLOCK为止

遗憾的是不同的操作系统特供的 epoll 接口有非常大差异，所以使用类似于 epoll 的接口实现具有较好跨平台能力的server会比較困难。

幸运的是，有非常多高效的事件驱动库能够屏蔽上述的困难，常见的事件驱动库有 libevent 库。还有作为 libevent 替代者的 libev 库。

这些库会依据操作系统的特点选择最合适的事件探测接口。而且增加了信号 (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。

下章将介绍怎样使用 libev 库替换 select 或 epoll 接口。实现高效稳定的server模型。

5、libevent方法

libevent是一个事件触发的网络库，适用于windows、linux、bsd等多种平台，内部使用select、epoll、kqueue等系统调用管理事件机制。著名分布式缓存软件memcached也是libevent based，并且libevent在使用上能够做到跨平台。并且依据libevent官方站点上发布的数据统计，似乎也有着非凡的性能。

libevent 库实际上没有更换 `select()`、`poll()` 或其它机制的基础。而是使用对于每一个平台最高效的高性能解决方式在实现外加上一个包装器。

为了实际处理每一个请求，libevent 库提供一种事件机制。它作为底层网络后端的包装器。

事件系统让为连接加入处理函数变得很简便，同一时候减少了底层 I/O 复杂性。这是 libevent 系统的核心。

libevent 库的其它组件提供其它功能。包含缓冲的事件系统（用于缓冲发送到client/从client接收的数据）以及 HTTP、DNS 和 RPC 系统的核心实现。

1、libevent有以下一些特点和优势：

1）事件驱动，高性能；
2）轻量级，专注于网络。
3) 跨平台，支持 Windows、Linux、Mac Os等；
4) 支持多种 I/O多路复用技术， epoll、poll、dev/poll、select 和kqueue 等。
5) 支持 I/O。定时器和信号等事件

2、libevent部分组成：

1）event 及 event_base事件管理包含各种**IO（socket）**、定时器、信号等事件，也是libevent应用最广的模块。

2) evbuffer event 及 event_base 缓存管理是指evbuffer功能；提供了高效的读写方法

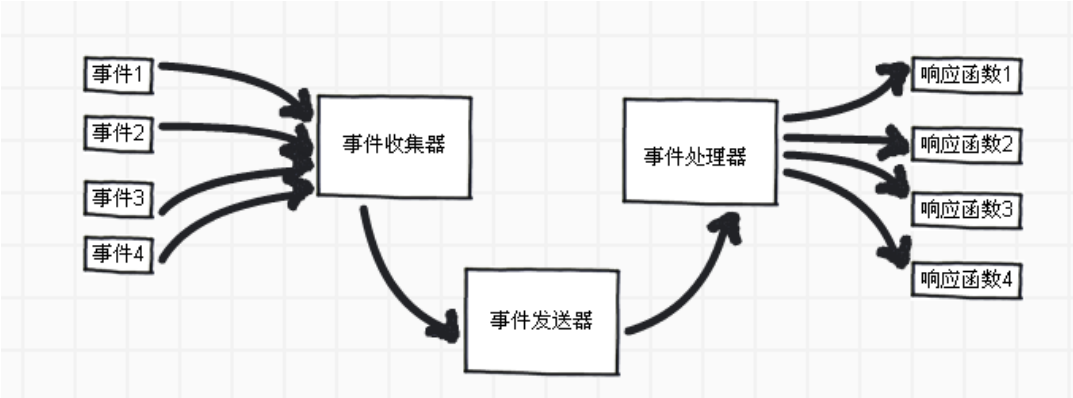
3) evdns DNS是libevent提供的一个异步DNS查询功能；

4) evhttp HTTP是libevent的一个轻量级http实现，包含server和client

libevent也支持ssl，这对于有安全需求的网络程序非常的重要。可是其支持不是非常完好，比方http server的实现就不支持ssl。

3、事件处理框架

libevent是事件驱动的库，所谓事件驱动，简单地说就是你点什么button(即产生什么事件),电脑运行什么操作(即调用什么函数)。

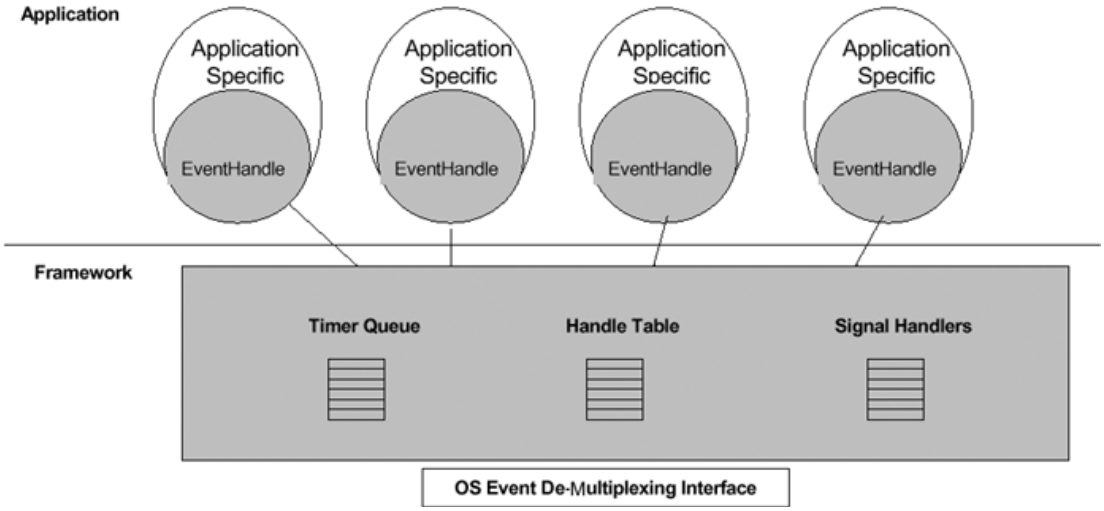


Libevent框架本质上是一个典型的Reactor模式。所以仅仅须要弄懂Reactor模型。libevent就八九不离十了。

Reactor模式。是一种事件驱动机制。

应用程序须要提供对应的接口并注冊到Reactor上，假设对应的事件发生，Reactor将主动调用应用程序注冊的接口，这些接口又称为“回调函数”。

在Libevent中也是一样。向Libevent框架注冊对应的事件和回调函数；当这些事件发生时，Libevent会调用这些回调函数处理对应的事件（I/O读写、定时和信号）。



使用Reactor模型，必备的几个组件：事件源、Reactor框架、多路复用机制和事件处理程序，先来看看Reactor模型的总体框架，接下来再对每一个组件做逐一说明。

1) 事件源

Linux上是文件描写叙述符。Windows上就是Socket或者Handle了，这里统一称为“句柄集”。程序在指定的句柄上注册关心的事件，比方I/O事件。

1）2）event demultiplexer——事件多路分发机制

由操作系统提供的I/O多路复用机制，比方select和epoll。程序首先将其关心的句柄（事件源）及其事件注册到event demultiplexer上。当有事件到达时，event demultiplexer会发出通知“在已经注册的句柄集中，一个或多个句柄的事件已经就绪”；程序收到通知后，就能够在非堵塞的情况下对事件进行处理了。

相应到**libevent**中，依旧是**select**、**poll**、**epoll**等，可是libevent使用结构体eventop进行了封装，以统一的接口来支持这些I/O多路复用机制，达到了对外隐藏底层系统机制的目的。

3

）Reactor——反应器

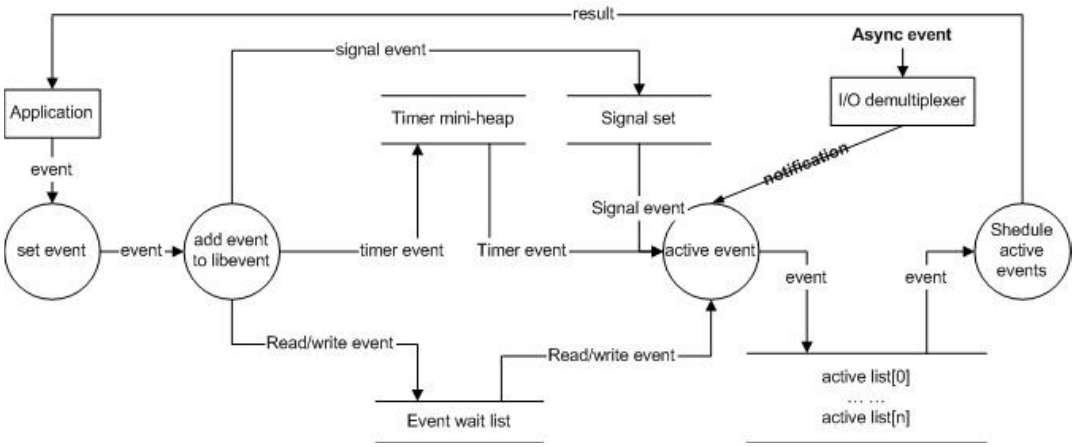
Reactor，是事件管理的接口，内部使用event demultiplexer注册、注销事件；并执行事件循环。当有事件进入“就绪”状态时，调用注册事件的回调函数处理事件。
相应到**libevent**中。就是**event_base**结构体。

4）Event Handler——事件处理程序

事件处理程序提供了一组接口，每一个接口对应了一种类型的事件。供Reactor在对应的事件发生时调用，运行对应的事件处理。通常它会绑定一个有效的句柄。

相应到**libevent**中，就是**event**结构体。

结合Reactor框架，我们来理一下libevent的事件处理流程，请看下图：



event_init() 初始化:

首先要隆重介绍event_base对象:

```
struct event_base {
    const struct eventop *evsel;
    void *evbase;
    int event_count;        /* counts number of total events */
    int event_count_active; /* counts number of active events */

    int event_gotterm;      /* Set to terminate loop */

    /* active event management */
    struct event_list **activequeues;
    int nactivequeues;
    struct event_list eventqueue;
    struct timeval event_tv;
    RB_HEAD(event_tree, event) timetree;
};
```

event_base对象整合了事件处理的一些全局变量, 角色是event对象的"总管家", 他包含了：

事件引擎函数对象(evsel, evbase),

当前入列事件列表(event_count, event_count_active, eventqueue),

全局终止信号(event_gotterm),

活跃事件列表(avtivequeues),

事件队列树(timetree)...

初始化时创建event_base对象, 选择 当前OS支持的事件引擎(epoll, poll, select...)并初始化， 创建全局信号队列(signalqueue), 活跃队列的内存分配(依据设置的priority个数,默觉得1).

event_set

event_set来设置event对象,包含全部者event_base对象, fd, 事件(EV_READ| EV_WRITE|EV_PERSIST), 回掉函数和参数,事件优先级是当前event_base的中间级别(current_base->nactivequeues/2)

设置监视事件后，事件处理函数能够仅仅被调用一次或总被调用。

仅仅调用一次：事件处理函数被调用后，即从事件队列中删除。须要在事件处理函数中再次增加事件，才干在下次事件发生时被调用；

总被调用：设置为EV_PERSIST，仅仅增加一次，处理函数总被调用，除非採用event_remove显式地删除。

event_add() 事件加入:

int event_add(struct event *ev, struct timeval *tv)

这个接口有两个参数, 第一个是要加入的事件, 第二个参数作为事件的超时值(timer). 假设该值非NULL, 在加入本事件的同一时候加入超时事件(EV_TIMEOUT)到时间队列树(timetree), 依据事件类型处理例如以下:

EV_READ => EVLIST_INSERTED => eventqueue
EV_WRITE => EVLIST_INSERTED => eventqueue
EV_TIMEOUT => EVLIST_TIMEOUT => timetree
EV_SIGNAL => EVLIST_SIGNAL => signalqueue

event_base_loop() 事件处理主循环

这里是事件的主循环,仅仅要flags不是设置为EVLOOP_NONBLOCK, 该函数就会一直循环监听事件/处理事件.

每次循环过程中, 都会处理当前触发(活跃)事件:

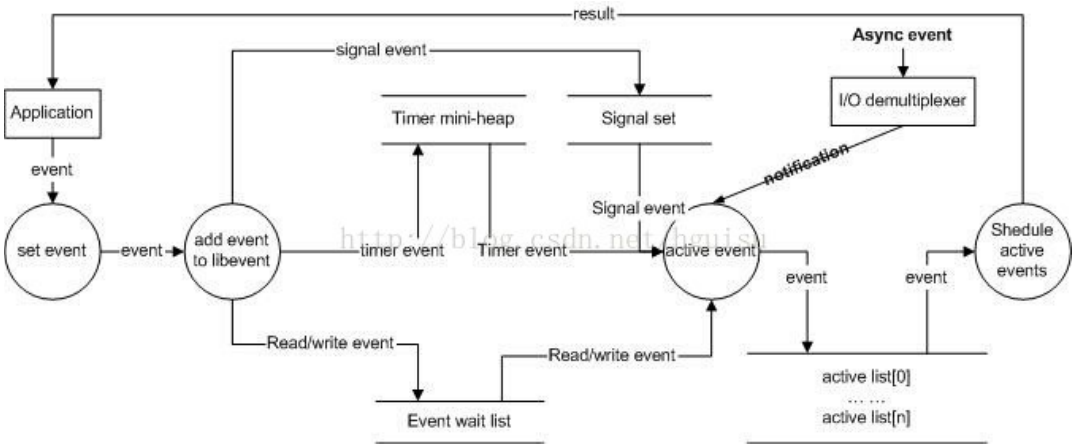
(a). 检测当前是否有信号处理(gotterm, gotsig), 这些都是全局参数,不适合多线程
(b). 时间更新,找到离当前近期的时间事件, 得到相对超时事件tv
(c). 调用事件引擎的dispatch wait事件触发, 超时值为tv, 触发事件加入到activequeues
(d). 处理活跃事件, 调用caller的callbacks (event_process_acitve)

典型的**libevent**的应用大致总体流程：

创建 libevent server的基本方法是， 注册当发生某一操作（比方接受来自client的连接）时应该运行的函数，然后调用主事件循环`event_dispatch()`。运行过程的控制如今由 libevent 系统处理。

注册事件和将调用的函数之后，事件系统開始自治。在应用程序执行时，能够在事件队列中加入（注冊）或删除（取消注冊）事件。事件注冊很方便，能够通过它加入新事件以处理新打开的连接，从而构建灵活的网络处理系统

（环境设置）-> (创建event_base) -> (注冊event，将此event增加到event_base中) -> (设置event各种属性。事件等) ->(将event增加事件列表 addevent) ->(開始事件监视循环、分发dispatch)。



样例：

比如，能够打开一个监听套接字，然后注册一个回调函数，每当须要调用 accept() 函数以打开新连接时调用这个回调函数，这样就创建了一个网络server。例1例如以下所看到的的代码片段说明基本过程：

例1：打开监听套接字，注册一个回调函数（每当须要调用 accept() 函数以打开新连接时调用它），由此创建网络server：

```
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <event.h>
using namespace std;


// 事件base
struct event_base* base;

// 读事件回调函数
void onRead(int iCliFd, short iEvent, void *arg)
{
    int iLen;
    char buf[1500];
    iLen = recv(iCliFd, buf, 1500, 0);
   if (iLen <= 0) {
        cout << "Client Close" << endl;
       // 连接结束(=0)或连接错误(<0)。将事件删除并释放内存空间
        struct event *pEvRead = (struct event*)arg;
        event_del(pEvRead);
        delete pEvRead;
        close(iCliFd);
        return;
    }
    buf[iLen] = 0;
    cout << "Client Info:" << buf << endl;
}

// 连接请求事件回调函数
void onAccept(int iSvrFd, short iEvent, void *arg)
{
   int iCliFd;
   struct sockaddr_in sCliAddr;
   socklen_t iSinSize = sizeof(sCliAddr);
   iCliFd = accept(iSvrFd, (struct sockaddr*)&sCliAddr, &iSinSize);
    // 连接注册为新事件 (EV_PERSIST为事件触发后不默认删除)
   struct event *pEvRead = new event;
   event_set(pEvRead, iCliFd, EV_READ|EV_PERSIST, onRead, pEvRead);
   event_base_set(base, pEvRead);
   event_add(pEvRead, NULL);
}



int main()
{
    int iSvrFd;
    struct sockaddr_in sSvrAddr;
    memset(&sSvrAddr, 0, sizeof(sSvrAddr));
    sSvrAddr.sin_family = AF_INET;
    sSvrAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    sSvrAddr.sin_port = htons(8888);

    // 创建tcpSocket（iSvrFd），监听本机8888端口
    iSvrFd = socket(AF_INET, SOCK_STREAM, 0);
    bind(iSvrFd, (struct sockaddr*)&sSvrAddr, sizeof(sSvrAddr));
    listen(iSvrFd, 10);

    // 初始化base
    base = event_base_new();
```

```
     struct event evListen;
    // 设置事件
    event_set(&evListen, iSvrFd, EV_READ|EV_PERSIST, onAccept, NULL);
     // 设置为base事件
     event_base_set(base, &evListen);
     // 加入事件
     event_add(&evListen, NULL);

     // 事件循环
     event_base_dispatch(base);
     return 0;

 }
```

`event_set()` 函数创建新的事件结构，

`event_add()` 在事件队列机制中加入事件。

然后，`event_dispatch()` 启动事件队列系统，開始监听（并接受）请求。

使用其它语言的实现

虽然 C 语言非常适合很多系统应用程序。可是在现代环境中不常常使用 C 语言，脚本语言更灵活、更有用。

幸运的是，Perl 和 PHP 等大多数脚本语言是用 C 编写的，所以能够通过扩展模块使用 libevent 等 C 库。

4、libev库

官方文档：http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod

与 libevent 一样，libev 系统也是基于事件循环的系统，它在 `poll()`、`select()` 等机制的本机实现的基础上提供基于事件的循环。

libev是libevent之后的一个事件驱动的编程框架。其接口和libevent基本类似。据官方介绍。其性能比libevent还要高，bug比libevent还少。

libev API 比較原始，没有 HTTP 包装器。可是 libev 支持在实现中内置很多其它事件类型。比如，一种 evstat 实现能够监视多个文件的属性变动，能够在 例4 所看到的的 HTTP 文件解决方式中使用它。

可是，libevent 和 libev 的基本过程是同样的。创建所需的网络监听套接字。注冊在运行期间要调用的事件。然后启动主事件循环，让 libev 处理过程的其余部分。

Libev是一个event loop：向libev注冊感兴趣的events，比方Socket可读事件，libev会对所注冊的事件的源进行管理，并在事件发生时触发对应的程序。

事件驱动框架：

定义一个监控器、书写触发动作逻辑、初始化监控器、设置监控器触发条件、将监控器增加大事件驱动器的循环中就可以。

libev的事件驱动过程能够想象成例如以下的伪代码：

```
do_some_init()
is_run = True
while is_run:
    t = caculate_loop_time()
```

```
    deal_loop(t)
    deal_with_pending_event()
do_some_clear()
```

首先做一些初始化操作，然后进入到循环中，该循环通过一个状态位来控制是否运行。

在循环中。计算出下一次轮询的时间，这里轮询的实现就采用了系统提供的epoll、kqueue等机制。

再轮询结束后检查有哪些监控器的被触发了，依次运行触发动作。

Libev 除了提供了主要的三大类事件（IO事件、定时器事件、信号事件）外还提供了周期事件、子进程事件、文件状态改变事件等多个事件。

libev所实现的功能就是一个强大的reactor,可能notify事件主要包含以下这些：

- ev_io // IO可读可写
- ev_stat // 文件属性变化
- ev_async // 激活线程
- ev_signal // 信号处理
- ev_timer // 定时器
- ev_periodic // 周期任务
- ev_child // 子进程状态变化
- ev_fork // 开辟进程
- ev_cleanup // event loop退出触发事件
- ev_idle // 每次event loop空暇触发事件
- ev_embed // TODO(zhangyan04):I have no idea.
- ev_prepare // 每次event loop之前事件
-
- ev_check // 每次event loop之后事件

libev 相同须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，当中包含 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)
```

初始化内容包含回调函数 callback，被探测的句柄 fd 和须要探测的事件。EV_READ 表“可读事件”。EV_WRITE 表“可写事件”。

如今，用户须要做的不过在合适的时候，将某些 ev_io 从 ev_loop 增加或剔除。一旦增加，下个循环即会检查 ev_io 所指定的事件有否发生；假设该事件被探测到，则 ev_loop 会自己主动运行 ev_io 的回调函数 callback()；假设 ev_io 被注销。则不再检测相应事件。

不管某 ev_loop 启动与否，都能够对其加入或删除一个或多个 ev_io，加入删除的接口是 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，我们能够easy得出例如以下的“一问一答”的server模型。因为没有考虑server端主动终止连接机制，所以各个连接能够维持随意时间，client能够自由选择退出时机。

IO事件、定时器事件、信号事件：

```
#include<ev.h>
```

```
#include <stdio.h>
#include <signal.h>
#include <sys/unistd.h>

ev_io io_w;
ev_timer timer_w;
ev_signal signal_w;

void io_action(struct ev_loop *main_loop,ev_io *io_w,int e)
{
        int rst;
        char buf[1024] = {''};
        puts("in io cb\n");
        read(STDIN_FILENO,buf,sizeof(buf));
        buf[1023] = '';
        printf("Read in a string %s \n",buf);
        ev_io_stop(main_loop,io_w);

}

void timer_action(struct ev_loop *main_loop,ev_timer *timer_w,int e)
{
        puts("in tiemr cb \n");
        ev_timer_stop(main_loop,io_w);

}

void signal_action(struct ev_loop *main_loop,ev_signal signal_w,int e)
{
        puts("in signal cb \n");
        ev_signal_stop(main_loop,io_w);
        ev_break(main_loop,EVBREAK_ALL);
}

int main(int argc ,char *argv[])
{
        struct ev_loop *main_loop = ev_default_loop(0);
        ev_init(&io_w,io_action);
        ev_io_set(&io_w,STDIN_FILENO,EV_READ);
        ev_init(&timer_w,timer_action);
        ev_timer_set(&timer_w,2,0);
        ev_init(&signal_w,signal_action);
        ev_signal_set(&signal_w,SIGINT);

        ev_io_start(main_loop,&io_w);
        ev_timer_start(main_loop,&timer_w);
        ev_signal_start(main_loop,&signal_w);

        ev_run(main_loop,0);

return 0;
}
```

这里使用了3种事件监控器，分别监控IO事件、定时器事件以及信号事件。因此定义了3个监控器(watcher)，以及触发监控器时要运行动作的回调函数。Libev定义了多种监控器，命名方式为 `ev_xxx` 这里xxx代表监控器类型，事实上现是一个结构体。

```
typedef struct ev_io
{
  ....
} ev_io;
```

通过宏定义能够简写为 `ev_xxx`。

回调函数的类型为 `void cb_name(struct ev_loop *main_loop,ev_xxx *io_w,int event)` 。

在main中，首先定义了一个事件驱动器的结构 `struct ev_loop *main_loop` 这里调用 `ev_default_loop(0)` 生成一个预制的全局驱动器。这里能够参考Manual中的选择。

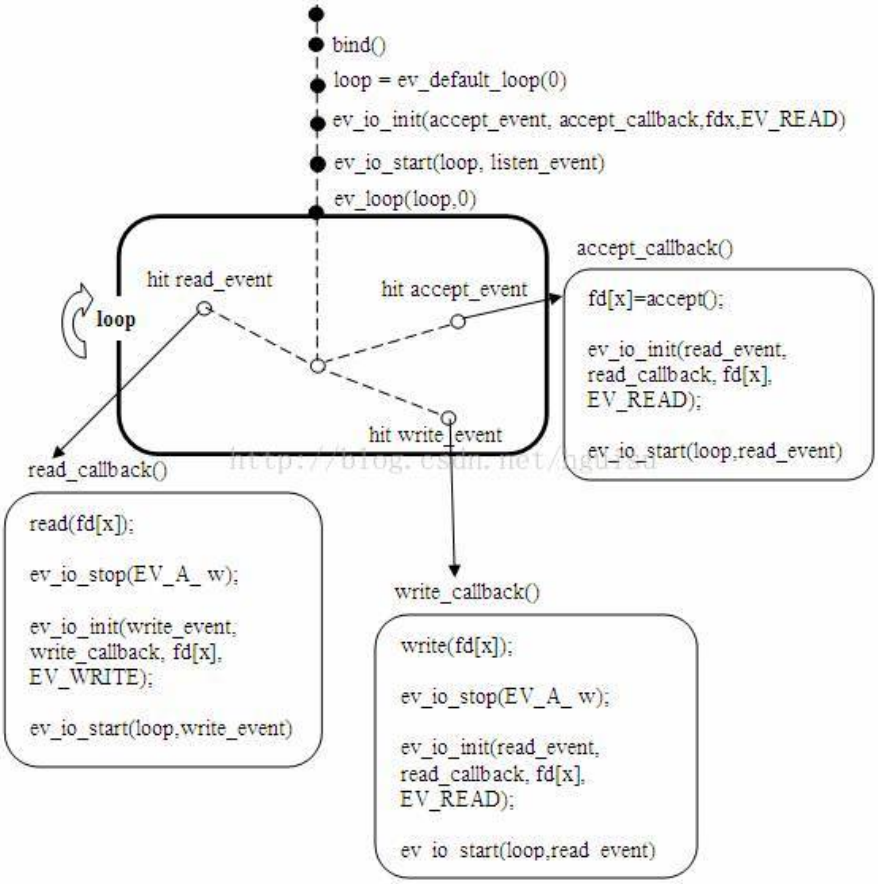
然后依次初始化各个监控器以及设置监控器的触发条件。

初始化监控器的过程是将对应的回调函数即触发时的动作注册到监控器上。

设置触发条件则是该条件产生时才去运行注册到监控器上的动作。对于IO事件，通常是设置特定fd上的的可读或可写事件，定时器则是多久后触发。这里定时器的触发条件中还有第三参数。表示第一次触发后，是否循环。若为0则吧循环，否则按该值循环。信号触发器则是设置触发的信号。

在初始化并设置好触发条件后，先调用ev_xxx_start 将监控器注册到事件驱动器上。接着调用 ev_run 開始事件驱动器。

使用libev库的server模型



上述模型能够接受随意多个连接，且为各个连接提供全然独立的问答服务。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它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。

因为传统的 web server。ftp server及其它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server程序具有参考价值；另外，对于须要实现远程监视或远程遥控的应用程序，上述模型相同提供了一个可行的实现方案。

php-了libev扩展socket：

```
<?php
/* 使用异步io訪问socket Use some async I/O to access a socket */

// `sockets' extension still logs warnings
// for EINPROGRESS, EAGAIN/EWOULDBLOCK etc.
error_reporting(E_ERROR);

$e_nonblocking = array (/*EAGAIN or EWOULDBLOCK*/11, /*EINPROGRESS*/115);

// Get the port for the WWW service
$service_port = getservbyname('www', 'tcp');
```

```
// Get the IP address for the target host
$address = gethostbyname('google.co.uk');

// Create a TCP/IP socket
$socket = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);
if ($socket === FALSE) {
    echo \"socket_create() failed: reason: \"
        .socket_strerror(socket_last_error()) . \"n\";
}

// Set O_NONBLOCK flag
socket_set_nonblock($socket);

// Abort on timeout
$timeout_watcher = new EvTimer(10.0, 0., function () use ($socket) {
    socket_close($socket);
    Ev::stop(Ev::BREAK_ALL);
});

// Make HEAD request when the socket is writable
$write_watcher = new EvIo($socket, Ev::WRITE, function ($w)
    use ($socket, $timeout_watcher, $e_nonblocking) {
    // Stop timeout watcher
    $timeout_watcher->stop();
    // Stop write watcher
    $w->stop();

    $in = \"HEAD / HTTP/1.1rn\";
    $in .= \"Host: google.co.ukrn\";
    $in .= \"Connection: Closernrn\";

    if (!socket_write($socket, $in, strlen($in))) {
        trigger_error(\"Failed writing $in to socket\", E_USER_ERROR);
    }

    $read_watcher = new EvIo($socket, Ev::READ, function ($w, $re)
        use ($socket, $e_nonblocking) {
        // Socket is readable. recv() 20 bytes using non-blocking mode
        $ret = socket_recv($socket, $out, 20, MSG_DONTWAIT);

        if ($ret) {
            echo $out;
        } elseif ($ret === 0) {
            // All read
            $w->stop();
            socket_close($socket);
            return;
        }

        // Caught EINPROGRESS, EAGAIN, or EWOULDBLOCK
        if (in_array(socket_last_error(), $e_nonblocking)) {
            return;
        }

        $w->stop();
        socket_close($socket);
    });

    Ev::run();
});

$result = socket_connect($socket, $address, $service_port);

Ev::run();
?>
```

结束语

libevent 和 libev 都提供灵活且强大的环境。支持为处理server端或client请求实现高性能网络（和其它 I/O）接口。

目标是以高效（CPU/RAM 使用量低）的方式支持数千甚至数万个连接。在本文中，您看到了一些演示样例，包含 libevent 中内置的 HTTP 服务，能够使用这些技术支持基于 IBM Cloud、EC2 或 AJAX 的 web 应用程序。

参考：

http://www.ibm.com/developerworks/cn/linux/l-cn-edntwk/

http://www.ibm.com/developerworks/cn/aix/library/au-libev/

《 UNIX 环境高级编程》