moDemos

# 1.socket网络编程

## 1.1.同步/异步、阻塞/非阻塞

同步和异步的概念，主要针对客户端来说；阻塞/非阻塞的概念，主要针对服务器端来说。

* 同步：所谓同步，指的是client发出一个调用后，在没有得到结果之前，这个调用就不返回。也就是必须一件一件事情的顺序执行，一件事做完了才能做下一件事；
* 异步：异步则不同；指的是client发出一个调用后，不能立即得到需要的结果，但是调用会立即返回；一旦这个调用在server端执行完成后，server端将通过 状态、通知或者回调 来通知调用者。
* 阻塞：阻塞指的是server端处理完这个请求之前，处理线程将被挂起；
* 非阻塞：指的是server端收到调用请求后，立即返回，而后通过其他方式通知client这个调用的执行结果；

## 1.2.Linux下的五种I/O模型

### 1.2.1.阻塞I/O模型

特点是：进程会一直阻塞，直到数据拷贝完成；

应用程序调用一个IO函数，导致该应用程序阻塞，直到数据准备好了才能够继续执行，如果数据一直没有准备好，将一直等待；

调用recv/recvfrom函数时，发生在内核中的等待数据和赋值数据的过程如下图：



这种模型下，一般来说，一旦接收到一个请求，会开辟一个线程或进程处理这个请求，主进程/主线程继续等待下一个请求；

#### 源码路径

moDemos/socket/block下的client和server目录；

#### 交互序列

客户端和服务器端使用TCP的方式进行阻塞式网络通信时:

**客户端**的依次函数调用顺序是：

1. Socket：创建一个socket；
2. Bind：非必需，如果不执行bind，系统默认会找一个可用的port来给客户端使用；
3. Connect：在服务器端已经就绪的情况下，以此函数向其发起握手请求；
4. Send：发送数据给服务器端；
5. Recv：收取服务器端发送过来的数据；
6. Close：关闭这个已经创建的socket；

**服务器端**的依次函数调用顺序是：

1. Socket：创建一个socket；
2. Bind：必须，对于服务器端来说，一般来说一定要手动指定一个port；否则客户端在connect的时候无法获知需要连接到哪个端口；
3. Listen：指定监听能力；
4. Accept：监听connect请求；
5. Recv：收取客户端发送来的request；
6. Send：发送数据给客户端；
7. Close：关闭这个socket；

使用UDP的方式进行阻塞式网络通信时：

**客户端**的依次函数调用顺序是：

1. Socket：创建一个socket；
2. Bind：非必需，如果不执行bind，系统默认会找一个可用的port来给客户端使用；
3. ~~Connect：在服务器端已经就绪的情况下，以此函数向其发起握手请求；~~
4. Sendto：发送数据给服务器端；
5. Recvfrom：收取服务器端发送过来的数据；
6. Close：关闭这个已经创建的socket；

**服务器端**的依次函数调用顺序是：

1. Socket：创建一个socket；
2. Bind：必须，对于服务器端来说，一般来说一定要手动指定一个port；否则客户端在connect的时候无法获知需要连接到哪个端口；
3. ~~Listen：指定监听能力；~~
4. ~~Accept：监听connect请求；~~
5. Recvfrom：收取客户端发送来的request；
6. Sendto：发送数据给客户端；
7. Close：关闭这个socket；

#### FAQ

**1.SIGPIPE引起的程序自动退出**

Client连接server时，如果server没有启动，一定概率下，程序执行到send/sendto函数的时候，会突然自行退出，并提示是收到了“SIGPIPE”的信号量所致。

正常情况下，如果一个进程收到“SIGPIPE”信号，默认的动作就是退出进程。

所以针对这个情况有两个处理方式:1.自行实现一个函数，注册给SIGPIPE这个信号量；2.将SIGPIPE信号量忽略。

代码里用到了后者： signal(SIGPIPE, SIG\_IGN);

这样处理后，程序不会退出了。我们还可以根据send/sendto函数执行出错后的errno进行判断，如果errno等于SIGPIPE，就知道是收到了这个信号量。

**2.程序退出后，短时间内不能再次启动**

在client/server进程执行结束退出后，如果立刻再次执行该进程，进程bind到port的函数将执行失败。因为对于端口的使用，默认是要过一段时间才会释放，一般系统设置为2—4分钟，所以需要等待这么长时间之后才能再次执行bind。

但是如果希望不等待直接能够执行，可以在创建了socket之后，对该socket置状态：int on=1; int ret = setsockopt(mSockId,SOL\_SOCKET, SO\_REUSEADDR,&on,sizeof(on));

设置该状态后，可以确保退出后即释放端口，马上可以再次启动。

### 1.2.2.非阻塞I/O模型

特点是：进程反复调用IO函数，直到能够收到数据为止；

把一个socket设置成非阻塞，就是要告诉内核，当所请求的I/O操作无法完成的时候，不要将进程睡眠，而是返回一个错误值。一旦进程发现返回值是这个特定的错误值，就知道是由于数据还没有就绪导致的返回出错，就可以等待后再次申请数据，直到有数据为止。

这个返回值就是errno=EWOULDBLOCK；

图示如下：



#### 代码路径

moDemos/socket/noblock下的client和server目录；

#### 设置方式

有两种方式可以用来设置一个socket为非阻塞的：

* 方式1：int b\_on = 1; ioctl(mSockId, FIONBIO, &b\_on);需要包含的头文件是：#include <sys/ioctl.h>;
* 方式2：int flags = fcntl(mSockId, F\_GETFL, 0); ret = fcntl(mSockId, F\_SETFL, flags | O\_NONBLOCK);需要包含的头文件是：#include <unistd.h> #include <fcntl.h>;函数fcntl返回负值说明执行出错了。

### 1.2.3.IO复用模型

特点是：同样是非阻塞IO，但是可以一次对多个IO端口进行监测，这是比单纯的非阻塞IO优越的地方。

主要有select、poll、epoll三种方式，图示如下：



其中，select是POSIX规定支持的，一般操作系统都支持；而epoll是linux特有的；

Select和poll，本质上都是通过设置或检查存放fd标志位的数据结构，来决定后续的处理方式；epoll则不同，是通过注册事件的方式，一旦某个fd就绪会通过回掉函数激活处理；

**支持一个进程所能打开的最大连接数**

|  |  |
| --- | --- |
| select | 单个进程所能打开的最大连接数有FD\_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是32\*32，同理64位机器上FD\_SETSIZE为32\*64），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。 |
| poll | poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的 |
| epoll | 虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接 |

**FD剧增后带来的IO效率问题**

|  |  |
| --- | --- |
| select | 因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。 |
| poll | 同上 |
| epoll | 因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。 |

**消息传递方式**

|  |  |
| --- | --- |
| select | 内核需要将消息传递到用户空间，都需要内核拷贝动作 |
| poll | 同上 |
| epoll | epoll通过内核和用户空间共享一块内存来实现的。 |

**表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。**

#### 1.2.3.1.Select

原理是：当一个客户端连接上服务器时，服务器将其fd加入到fd\_set集合中，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，与客户端进行通信。

**函数原型**

int select(

int maxfdp, //winsock中该参数无意义

fd\_set \* readfds,

fd\_set \* writefds,

fd\_set \* exceptfds,

const struct timeval\* timeout //非阻塞模式下的最大超时时间

);

各参数的含义如下：

* Maxfdp：意为“最大的fd加1(max fd plus 1)”.在readfds、writefds和exceptfds中找到最大的fd值后加1；或者直接设置为FD\_SETSIZE，这是<sys/types.h>中的常数，是系统能支持的最大的fd个数，但对于大多数程序而言这个值太大了，不适合使用。
* Readfds：要监视读变化的所有文件描述符集合。如果这个集合中的文件描述符有1个可读，select就会返回一个大于0的值；如果没有可读的文件，根据timeout参数判断是否超时，如果超时了返回0，否则返回负值。可以传入NULL，表示不关心所有文件的读变化。
* Writefds：要监视写变化的所有文件描述符集合。同上。
* Exceptfds：要监视的异常错误的文件描述符集合。同上。
* Timeout：如果传入NULL，select处于阻塞状态，一定要等到文件描述符中某个文件发生变化为止；如果设置为0，select为纯粹的非阻塞函数，不管有没有文件描述符变化，都会立即返回，文件无变化返回0，有变化返回一个正数；如果设置为大于0的数值，在timeout内阻塞，返回值如上所述；

**使用步骤**

1. 创建所关注的事件的文件描述符集合:readfds，writefds，exceptfds；
2. 调用select，等待事件发生，需要注意，**select函数的阻塞与socket是否是阻塞的并无直接关联**；
3. 轮询所有fd\_set中的每一个fd，检查是否有事件，如果有，进行处理；

**相关宏**

FD\_ZERO(fd\_set \* fds); //清除所有标记位

FD\_SET(int fd, fd\_set \* fds); //设置fd\_set中对应fd的位为1

FD\_ISSET(int fd, fd\_set \* fds); //判断fd\_set中的某个fd是否被设置

FD\_CLR(int fd, fd\_set \* fds); //从fd\_set中清除fd对应的位，不再监视该文件

**返回值**

Select返回处于就绪状态并且已经包含在fd\_set结构中的描述字总数；如果超时了返回0；如果有监视错误返回负值；

* 0-：出错，对文件描述符的监视失败了，将清空所有文件描述符；
* 0：超时；
* 0+：有文件可操作，通过FD\_ISSET遍历可用的文件描述符；

Select返回后，在所有的文件描述符集合中，依旧是1的位就是准备好的描述符。这也是每次select成功后，都要通过FD\_ISSET判断文件描述符的原因。

**优缺点**

* 优点：单线程执行，CPU资源消耗小，同时能够应对多客户端；
* 缺点1：单个进程所能监视的文件描述符的最大值有限制(/proc/sys/fs/file-max)；
* 缺点2：需复制大量的句柄信息，空间消耗大；
* 缺点3：返回值是句柄列表，需要轮询处理，时间消耗大；
* 缺点4：触发方式是**水平触发**，应用程序一旦没有对一个已经就绪的描述符做IO操作，之后每次调用select还是会将这个描述符通知给进程；与之相对的是**边缘触发**；

**其他**

客户端的一个connect动作，将在服务器端激发一个“可读事件”，所以select能够探测到来自客户端的connect动作；

如果select发现某个句柄有了“可读事件”，服务器应尽快做recv动作；如果发现了“可写事件”，应及时准备send动作；

#### 1.2.3.2.Poll

Poll与select的相似度极高，且优缺点相同，不做研究。

#### 1.2.3.3.Epoll

由于select和poll在处理大量网络请求时的缺陷，epoll技术被提出。在linux内核版本2.6中正式引入了这个实现，其优点概述为：

* Epoll没有最大并发连接数的限制，上限是系统最大可以打开的文件数量，这个数字远远大于select限制的2048(甚至1024)；一般来说这个数字与系统内存关系很大，可以通过：cat /proc/sys/fs/file-max查看；
* 效率提升：epoll只关心“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，epoll的效率远高于select和poll；
* 内存拷贝：epoll使用了mmap的内核与用户空间的共享内存的方式，省去了内存拷贝的问题；

**两种工作方式**

Epoll支持两种工作方式：水平触发和边缘触发。

水平触发(Level triggered)是缺省的工作方式，支持block和non-block两种。这种模式下，内核告诉使用者一个文件描述符是否就绪了，然后就可以对这个fd进行操作。如果不做任何操作，内核还会继续通知使用者。

边缘触发(edge triggered)是高速的工作方式，只支持non-block这一种。这种模式下，当描述符变为就绪时，内核发出通知。之后内核不会再对这个描述符发送更多的就绪通知，直到使用者做了某些操作(数据读取完成等)使这个文件描述符不再是就绪了。

简单说，一旦就绪了，水平触发方式会不断的发就绪通知直到使用者处理完成；而边缘触发方式发送一次后将不再发送。

**API**

int epoll\_create(int size);

创建一个epoll的句柄。

Size指定了能够监听的数目一共有多少。

需要注意的是，一旦创建完成，epoll句柄会占用一个fd值，在linux下的/proc/processId/fd下，是可以查找到这个fd的，所以在使用完epoll之后，一定要close()关闭，否则会造成句柄泄漏；

成功时返回一个非负的文件描述符；失败时返回-1并置errno；

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

epoll的事件注册函数；

参数说明：epfd是epoll\_create时创建的epoll的句柄；op代表了三种可能的动作：EPOLL\_CTL\_ADD—注册新的fd到epfd中，EPOLL\_CTL\_MOD--修改已经注册的fd的监听事件，EPOLL\_CTL\_DEL—从epfd中删除一个fd；参数fd是需要监听的文件描述符；event参数告诉内核需要监听的事件是什么，其结构体定义是：

Typedef union epoll\_data

{

Void \*ptr;

Int fd;

\_\_uint32\_t u32;

\_\_uint64\_t u64;

}epoll\_data\_t;

Struct epoll\_event

{

\_\_uint32\_t events;

Epoll\_data\_t data;

};

这个结构体中events属性的取值，可以是以下几个宏的组合：

* EPOLLIN:对应的文件描述符可读(包括对端的SOCKET正常关闭)；
* EPOLLOUT:对应的文件描述符可写；
* EPOLLPRI:对应的文件描述符有紧急的数据可读(也有说是表示有带外数据到来)；
* EPOLLERR:对应的文件描述符发生错误；
* EPOLLHUP:对应的文件描述符被挂断；
* EPOLLET:将epoll设置为边缘触发模式(edge triggered)；
* EPOLLONESHOT:只监听一次事件，监听结束后如果需要再次监听这个文件，需要再次将文件描述符加入到epoll队列中；

函数执行成功返回0；执行失败返回-1并置errno；

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

收集在epoll监控的事件中已经就绪的事件。参数events不得为NULL!epoll会把就绪的事件赋值到events数组中。Maxevents告知内核这个events有多大，不能大于创建epoll时指定的size；timeout是超时时间(**单位是毫秒**)，0会立即返回，-1将永久阻塞或行为不确定。

如果函数调用成功，返回对应IO上已经准备好的文件描述符数目，返回0标识超时，返回-1说明有错误发生，errno被置。

**通用框架**

几乎所有的epoll程序都使用这样的框架：

    for( ; ; )

    {

        nfds = epoll\_wait(epfd,events,20,500);

        for(i=0;i<nfds;++i)

        {

            if(events[i].data.fd==listenfd) //有新的连接

            {

                connfd = accept(listenfd,(sockaddr \*)&clientaddr, &clilen);

                ev.data.fd=connfd;

                ev.events=EPOLLIN|EPOLLET;

                epoll\_ctl(epfd,EPOLL\_CTL\_ADD,connfd,&ev); //将新的fd添加到epoll的监听队列中

            }

            else if( events[i].events&EPOLLIN ) //接收到数据，读socket

            {

                n = read(sockfd, line, MAXLINE)) < 0    //读

                ev.data.ptr = md;     //md为自定义类型，添加数据

                ev.events=EPOLLOUT|EPOLLET;

                epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);//修改标识符，等待下一个循环时发送数据，异步处理的精髓

            }

            else if(events[i].events&EPOLLOUT) //有数据待发送，写socket

            {

                struct myepoll\_data\* md = (myepoll\_data\*)events[i].data.ptr;    //取数据

                sockfd = md->fd;

                send( sockfd, md->ptr, strlen((char\*)md->ptr), 0 );        //发送数据

                ev.data.fd=sockfd;

                ev.events=EPOLLIN|EPOLLET;

                epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev); //修改标识符，等待下一个循环时接收数据

            }

            Else

            {

                //其他的处理

            }

        }

    }

**MyFAQ**

执行accept的时候出错，errno=22，desc=[Invalid argument]

有两个原因可能导致这个问题：

1. Accept的参数没送对：accept(serverSockId, &clientSockAddr, &clientAddrLen);后两个参数可以不使用，但是要送入；
2. 没有listen：执行accept之前要首先让server开始listen；

#### 1.2.3.4.Libevent

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。其可以做到跨平台，性能也很卓越。

Libevent库还可以提供其他一些功能，例如HTTP、DNS等，主要的优点是:

* 事件驱动，性能高；
* 轻量级，专注于网络；
* 跨平台，支持windows、linux、mac os等；
* 支持多种I/O复用技术:epoll，kqueue，select，dev/poll等；
* 支持IO，定时器和信号等事件；

**组成概述**

通用的libevent，通常包含如下几个部分：

1. Event及event\_base事件管理，包括各种IO(socket等)、定时器、信号等，也是libevent被使用最广泛的模块；
2. Evbuffer，一个高效的缓存管理；
3. Evdns，一个异步的DNS查询功能；
4. Evhttp，一个轻量级的http服务，包括客户端和服务器端；

Libevent也支持ssl，这对于有安全要求的网络程序非常重要，但是并不是特别完善，比如http server的实现就不支持ssl。

#### 1.2.3.5.libev

# 2.线程池

## 2.1.使用原因

网络服务器等应用场合有一个共同点，就是很有可能在单位时间内需要处理数目极为巨大的连接请求，但是任务的处理时间却很短。

传统的多线程方案，是在收到请求后创建一个新的线程，由该线程执行任务。任务执行完成后线程退出，这就是“即时创建、即时销毁”的处理方式。但是如果处理任务的时间非常短，而执行频率又很高，那么服务器很可能处于不断的创建线程、销毁线程的状态。

线程执行时间分为三部分：T1—线程创建的时间；T2—线程执行时间；T3—线程销毁时间；如果T2很短，那么（T1+T3）时间占比将很高。

因此出现了线程池的概念，在初始化的时候将线程创建出来，使用的时候只占用T2的时间，可以大大降低任务的整体执行时间和系统消耗。

## 2.2.基本组成部分

线程池一般最少由以下四个部分组成：

* 线程管理器：创建并管理这个线程池；
* 工作线程：线程池中的所有线程；
* 任务接口：每个具体的任务提供的接口，供如上的工作线程调用以实现具体功能；
* 任务队列：存放还没有被处理到的任务；