1.close()函数

```
1. #include<unistd.h>
2. int close(int sockfd);      //返回成功为0，出错为-1.</span>
```

close 一个套接字的默认行为是把套接字标记为已关闭，然后立即返回到调用进程，该套接字描述符不能再由调用进程使用，也就是说它不能再作为read或write的第一个参数，然而TCP将尝试发送已排队等待发送到对端的任何数据，发送完毕后发生的是正常的TCP连接终止序列。

在多进程并发服务器中，父子进程共享着套接字，套接字描述符引用计数记录着共享着的进程个数，当父进程或某一子进程close掉套接字时，描述符引用计数会相应的减一，当引用计数仍大于零时，这个close调用就不会引发TCP的四路握手断连过程。

2.shutdown()函数

```
1. #include<sys/socket.h>
2. int shutdown(int sockfd,int howto);  //返回成功为0，出错为-1.</span>
```

该函数的行为依赖于howto的值

1.SHUT_RD：值为0，关闭连接的读这一半。

2.SHUT_WR：值为1，关闭连接的写这一半。

3.SHUT_RDWR：值为2，连接的读和写都关闭。

终止网络连接的通用方法是调用close函数。但使用shutdown能更好的控制断连过程（使用第二个参数）。

3.两函数的区别

close与shutdown的区别主要表现在：

close函数会关闭套接字ID，如果有其他的进程共享着这个套接字，那么它仍然是打开的，这个连接仍然可以用来读和写，并且有时候这是非常重要的 ，特别是对于多进程并发服务器来说。

而shutdown会切断进程共享的套接字的所有连接，不管这个套接字的引用计数是否为零，那些试图读得进程将会接收到EOF标识，那些试图写的进程将会检测到SIGPIPE信号，同时可利用shutdown的第二个参数选择断连的方式。

下面将展示一个客户端例子片段来说明使用close和shutdown所带来的不同结果：

客户端有两个进程，父进程和子进程，子进程是在父进程和服务器建连之后fork出来的，子进程发送标准输入终端键盘输入数据到服务器端，知道接收到EOF标识，父进程则接受来自服务器端的响应数据。

```
1. /* First  Sample client fragment,
2.  * 多余的代码及变量的声明已略       */
3.    s=connect(...);
```

```
4.    if( fork() )
{   /*      The child, it copies its stdin to the socket              */
5.        while( gets(buffer) >0)
6.            write(s,buf,strlen(buffer));
7.            close(s);
8.            exit(0);
9.    }
10.    else {          /* The parent, it receives answers  */
11.         while( (n=read(s,buffer,sizeof(buffer)){
12.             do_something(n,buffer);
13.             /* Connection break from the server is assumed  */
14.             /* ATTENTION: deadlock here                     */
15.          wait(0); /* Wait for the child to exit          */
16.          exit(0);
17.     }
```

对于这段代码，我们所期望的是子进程获取完标准终端的数据，写入套接字后close套接字，并退出，服务器端接收完数据检测到EOF（表示数据已发送完），也关闭连接，并退出。接着父进程读取完服务器端响应的数据，并退出。然而，事实会是这样子的嘛，其实不然！子进程close套接字后，套接字对于父进程来说仍然是可读和可写的，尽管父进程永远都不会写入数据。因此，此socket的断连过程没有发生，因此，服务器端就不会检测到EOF标识，会一直等待从客户端来的数据。而此时父进程也不会检测到服务器端发来的EOF标识。这样服务器端和客户端陷入了死锁（deadlock）。如果用shutdown代替close，则会避免死锁的发生。

```
1. if( fork() ) {  /* The child                    */
2.       while( gets(buffer)
3.          write(s,buffer,strlen(buffer));
4.       shutdown(s,1); /* Break the connection
5.
                        *for writing, The server will detect EOF now. Note
: reading from
6.
                  *the socket is still allowed. The server may send some m
ore data
7.                    *after receiving EOF, why not? */
8.       exit(0);
9.  }
```