浙江科技学院学报,第 18 卷第 4 期,2006 年 12 月
Journal of Zhejiang University of Science and Technology
*Vol. 18 No. 4*, Dec. 2006

# 大并发高性能用户应用服务器程序设计

王　华,张震宇
(浙江科技学院 自动化与电气工程学院,杭州 310023)

**摘　要**: Windows 下完成端口(IOCP)模型只需少数几个工作线程以及异步读写操作就可处理 2 000 个以上的客户端的并发连接。这相对于多线程方式的客户端并发数而言有了质的提升,同时避免了 CPU 在线程(进程)调度和切换上的大量开销,从而构建了一个稳定、高性能的服务器程序。

**关键词**: 多线程;完成端口;服务器程序
**中图分类号**: TP311　　**文献标识码**: A　　**文章编号**: 1671-8798(2006)04-0268-06

# Design of User Application Server Program for Giant Concurrent Connects

WANG Hua, ZHANG Zhen-yu
(School of Automation and Electricity, Zhejiang University of Science and Technology, Hangzhou 310023, china )

**Abstract**: I/O complete port (IOCP) of Windows present an efficient solution to the one-thread-per-client bottleneck problem. Using only a few processing threads and asynchronous input/output or send/receive, IOCP can handle over 2 000 giant connects of clients. Compared with the number of concurrent connects of the multithread, the IOCP has obvious predominance, also the less consumption of CPU in switching and regulating the work thread, because of which it can build a scalable and high performance server program.

**Key words**: multithread; IOCP; server program

用户应用服务器(UAS)相当于一个代理服务器,在大型视频监控系统或者大型网络游戏平台中均占有重要地位。客户端通过与 UAS 交互实现登陆,对于大型(电信级)网络视频监控系统以及大型网络在线游戏平台,客户端数目可能非常大(可能会达到数千人甚至上万人);如果要保证能够同时平等地登陆服务器,那么 UAS 的效率就很重要了。经典的网络服务器逻辑大多采用多线程/多进程方式,当一个客户端发起到服务器的连接时,服务器将会创建一个线程,让这个新的线程来处理后续事务。这种以一个专门的线程/进程来代表一个客户端对象的编程方法非常直观,易于理解。对于网络服务器来说,如果客户端并发请求数目比较少的话,用简单的多线程模型就可以应付了。如果一个线程因为等待 I/O 操作完成而被挂起,那么操作系统将会调度,另外一个就绪的线程投入运行,从而形成并发执行。对于大型网络服务器程序来说,这种方式存在着局限性。首先,创建线程/进程和销毁线程/进程

**收稿日期**: 2006-10-24

**作者简介**: 王　华(1978—　),男,浙江淳安人,助教,硕士,主要从事数字图像测控系统、网络监控系统和多媒体技术的研究。

的代价非常高昂,尤其是在服务器采用 TCP“短连接”[1]方式或者 UDP 方式通讯的情况下。例如,HTTP 协议中,客户端发起一个连接后,发送一个请求,服务器回应了这个请求后,连接也就被关闭了。如果采用经典方式设计 HTTP 服务器,那么过于频繁地创建线程/销毁线程对性能造成的影响是很大的。其次,即使一个协议中采取 TCP“长连接”,客户端连上服务器后就一直保持此连接,经典的设计方式也是有弊病的。如果客户端并发请求量很高,在同一时刻有很多客户端等待服务器响应的情况下,将会有过多的线程并发执行,频繁的线程切换将用掉一部分计算能力。实际上,如果并发线程数目过多的话,往往会过早地耗尽物理内存,CPU 大部分时间花在了创建线程和切换线程上面,由于线程切换的同时也将引起内存调页,因此最终导致服务器性能急剧下降。另外,Windows 操作系统内核允许创建的线程数是 2 000 个左右。Windows 下的 IOCP 机制避免了这种大量线程的创建、管理等操作,IOCP 只需少数几个工作线程就可高效地处理海量的客户连接。

## 1　完成端口模型的概念

对于一个需要应付同时有大量客户端并发请求的网络服务器来说,线程池是一个非常好的解决方案。线程池不光能够避免频繁地创建线程和销毁线程,而且能够用数目很少的线程处理大量客户端并发请求。系统产生固定数目的线程为客户提供服务,线程数量与客户的数量没什么联系。一方面,一个线程可以分时地为多个客户服务,另一方面,在一个客户的会话期间,多个线程接力为它提供服务[2]。线程池方法避免了线程的频繁创建和销毁带来的开销。操作系统为基于线程池的解决方案提供了一个极好的支持,这就是 Windows 下的完成端口(IOCP)原型。采用这种技术,可以利用为数不多的线程为成千上万的客户同时提供网络服务。实践表明,这种实现方法可以获得极好的性能和强大的扩展能力,程序的设计也不复杂。如果事先开好 $N$ 个线程,让它们在那里堵塞,可以将所有用户的请求都投递到一个消息队列中去。然后,那 $N$ 个线程逐一从消息队列中去取出消息并加以处理,就可以避免针对每一个用户请求都开线程。这样,不仅减少了线程的资源,也提高了线程的利用率。

完成端口并不是代表某个 TCP 或者 UDP 的端口,而是一个 Windows 的内核对象,它包括两部分:一个是线程池,另一个就是消息通知队列,I/O 完成端口是允许应用程序使用线程池来处理异步 I/O 请求的机制,线程池中的线程都负责处理 I/O 请求。通过 I/O 完成端口应用程序可以更快更有效地处理异步 I/O 请求。由操作系统把已经完成的重叠 I/O 请求的通知放入其中,一旦某项 I/O 操作完成,某个可以对该操作结果进行处理的工作者线程就会收到一则通知,而套接字在被创建后,可以在任何时候与某个完成端口进行关联。

通常情况下,会在应用程序中创建一定数量的工作者线程来处理这些通知。线程数量取决于应用程序的特定需要。理想的情况是,线程数量等于处理器的数量,不过这也要求任何线程都不应该执行诸如同步读写、等待事件通知等阻塞型的操作,以免线程阻塞。每个线程都将分到一定的 CPU 时间,在此期间该线程可以运行,然后另一个线程将分到一个时间片并开始执行。如果某个线程执行了阻塞型的操作,操作系统将剥夺其未使用的剩余时间片并让其他线程开始执行。也就是说,前一个线程没有充分使用其时间片,应用程序应该准备其他线程来充分利用这些时间片。

## 2　完成端口模型的使用

服务端完成端口的使用步骤如图 1 所示。

其中创建完成端口,如以下代码所示:

```
HANDLE CreateIoCompletionPort (
HANDLE FileHandle,
HANDLE ExistingCompletionPort,
ULONG_PTR CompletionKey,
DWORD NumberOfConcurrentThreads
);
```

参数 FileHandle 代表关联上的文件句柄,参数 ExistingCompletionPort 代表完成端口的句柄,参数 CompletionKey 为完成键,代表连接到服务器的客户端的 Socket 信息,参数 NumberOfConcurrentThreads 为处理 I/O 完成包的线程数。对于初始化完成端口时,此时完成端口并未与某个文件句柄关联,FileHandle 的值设为 INVALID_HANDLE_VALUE,参数 ExistingCompletionPort 值为 NULL,CompletionKey 值为 0,参数 NumberOfConcurrentThreads 值为 0,返回值就是创建的完成端口的句柄。完成端口创建后,就要创建若干个工作线程,用

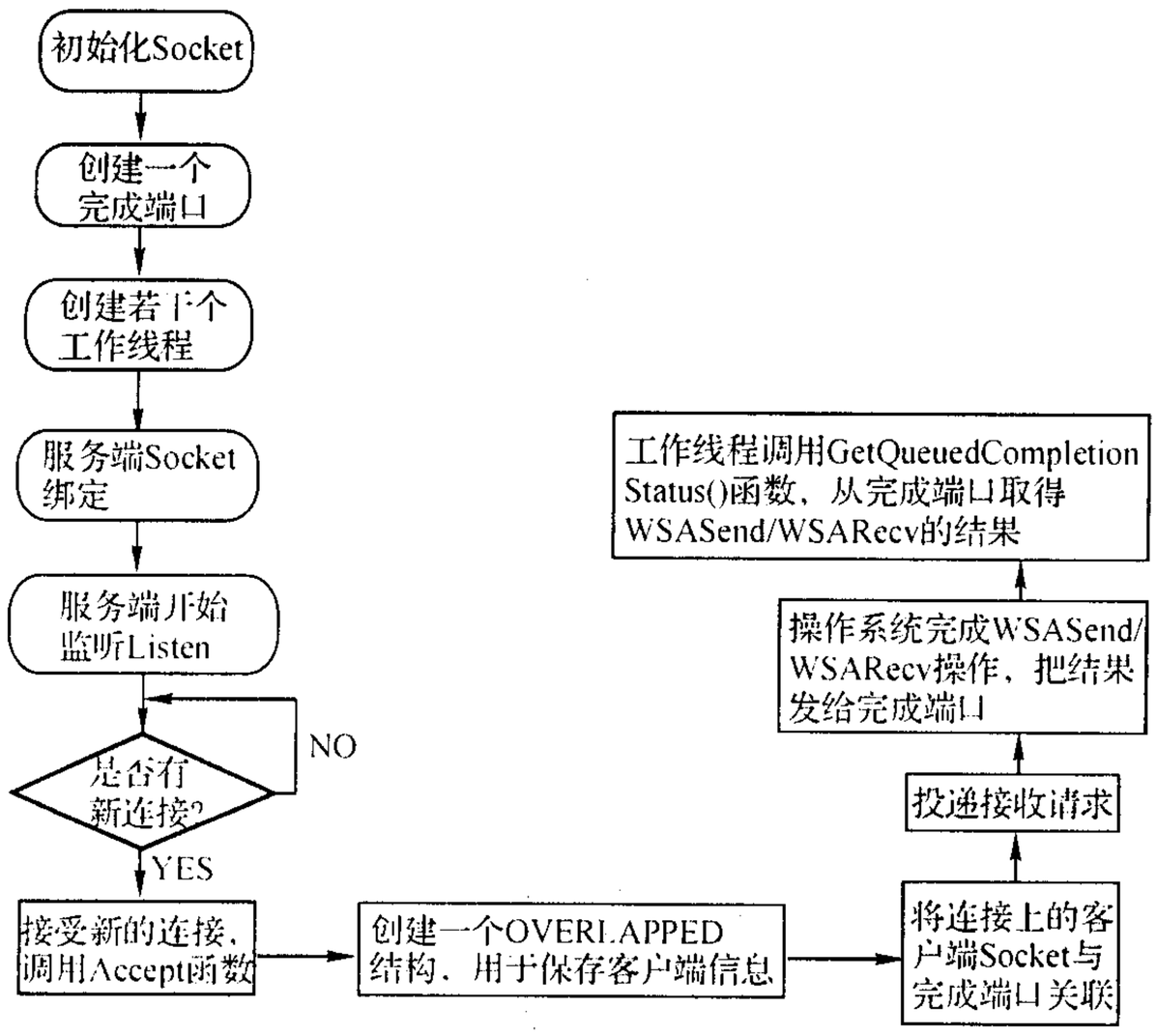


**图 1 服务端完成端口使用步骤**

于数据的收发，根据微软提供的文档，工作线程个数为 CPU 个数的 2 倍为宜[2]，代码如下：

```
//获得系统信息，以便获得 CPU 个数
GetSystemInfo(&SystemInfo);
//开启 CPU 个数的 2 倍个的线程
for(i = 0; i<SystemInfo. dwNumberOfProcessors * 2; i++)
{
  HANDLE ThreadHandle;
  //创建一个服务器工作线程，并且将完成端口传递给该线程
  if ((ThreadHandle = CreateThread(NULL, 0, ServerWorkerThread, CompletionPort,0, &ThreadID)) == NULL)
  {
    printf("CreateThread() failed with error %d\n", GetLastError());
    return;
  }
  CloseHandle(ThreadHandle);
}
```

这些为数不多的工作线程不断循环调用 GetQueuedCompletionStatus()函数并返回完成通知。当应用程序调用一个重叠操作函数时，要把指向一个 overlapped 结构的指针包括在其参数中。当操作完成后，可以通过 GetQueuedCompletionStatus()函数拿回这个指针。不过，单是根据这个指针所指向的 OVERLAPPED 结构，应用程序并不能分辨究竟完成的是哪个操作。要实现对操作的跟踪，可以定义一个 OVERLAPPED 结构，在其中加入所需的跟踪信息。无论何时调用重叠操作函数时，总是会通过其 lpOverlapped 参数传递一个 OVERLAPPEDPLUS 结构（例如 WSASend、WSARecv 等函数）。这就允许程序为每一个重叠调用操作设置某些操作状态信息，当操作结束后，可以通过 GetQueuedCompletionStatus()函数获得自定义结构的指针。注意 OVERLAPPED 字段不要求一定是这个扩展后的结构的第一个字段。当得到了指向 OVERLAPPED 结构的指针以后，可以用 CONTAINING_RECORD 宏取出其中指向扩展结构的指针，OVERLAPPEDPLUS 结构的定义如下：

```
typedef struct _OVERLAPPEDPLUS
{
OVERLAPPED OverLapped;
SOCKET s, sclient;
int OpCode;
WSABUF wbuf;
DWORD dwBytes, dwFlags;
} OVERLAPPEDPLUS;
```

其中，每句柄键(PerHandleKey)变量的内容，

是在把完成端口与套接字进行关联时所设置的完成键参数;Overlaped 参数返回的是一个指向发出重叠操作时所使用的那个 OVERLAPPEDPLUS 结构的指针。如果重叠操作调用失败时,也就是说,返回值是 SOCKET_ERROR,并且错误原因不是 WSA_IO_PENDING,那么完成端口将不会收到任何完成通知[3]。如果重叠操作调用成功,或者发生原因是 WSA_IO_PENDING 的错误时,完成端口将总是能够收到完成通知。把将使用该完成端口的套接字与之关联起来,方法是再次调用 CreateIoCompletionPort()函数,以下代码为接收到一个新的客户端套接字,并把它和前面创建的完成端口关联起来:

```
SOCKET ClientSocket,ListenSocket;
//初始化一个服务器监听 socket,用于监听客户端连接
if ((ListenSocket = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
{
  printf(初始化 Socket 失败,错误代码: %d\n", WSAGetLastError());
  return;
}
if (bind (ListenSocket, (PSOCKADDR)&InternetAddr,sizeof(InternetAddr)) ==SOCKET_ERROR)
{
  printf("绑定失败,错误代码:%d\n", WSAGetLastError());
  return;
}
if (listen(ListenSocket, 15) ==SOCKET_ERROR)//开始监听
{
  printf("listen() failed with error %d\n", WSAGetLastError());
  return;
}
while(TRUE) //开始接收从客户端来的连接
{
if((ClientSocket = WSAAccept(Listen, NULL, NULL, NULL, 0)) == SOCKET_ERROR)
  printf("WSAAccept() failed with error %d\n", WSAGetLastError());
  return;
}
//与前面创建的完成端口关联起来,将关键项也与指定的一个完成端口关联
if(CreateIoCompletionPort((HANDLE)ClientSocket,CompletionPort,(DWORD),0,0) == NULL)
{
  printf("创建完成端口失败,错误原因: %d\n", GetLastError());
  return;
}
```

这时就完成了客户端套接字与完成端口的关联操作。在这个套接字上进行的任何重叠操作都将通过完成端口发出完成通知。CreateIoCompletionPort()函数中的第二个参数就是创建完成端口后返回的完称端口的句柄,第三个参数用来设置一个与该套接字相关的"完成键(completion key)"(完成键可以是任何数据类型)。每当完成通知到来时,应用程序可以读取相应的完成键,因此,完成键可用来给套接字传递一些背景信息。在创建了完成端口、将一个或多个套接字与之相关联之后,就可以利用多个工作线程来处理完成通知。

## 3　服务端程序的进一步优化

### 3.1　内存管理

完成端口的最大优点在于其管理海量连接时的处理效率,但是它在传输大量数据方面没有什么优势。对于大型的视频监控系统,用户应用服务程序设计的初衷就是为了解决同时处理海量客户端连接问题,也就是一个连接效率问题。由于每个连接的数据包很小,一般都是一些信令数据,所以只需对于完成端口的连接效率的管理做一些适当的优化,这样就更能体现完成端口的威力了。当有一个新的连接到来的时候,服务程序都要为该程序创建一个结构体用于存储该客户端 Socket 的信息,而且在每一次的 WSASend 和 WSARecv 中,工作线程也要向系统投递一个 OVERLAPPED 类型的结构体用于数据的收发[4]。如果每来一个连接就开辟一块空间,服务端程序就要频繁地创建和

释放内存，造成系统性能的下降。为了避免频繁地创建释放这些结构体，可以预先创建一定数量的这些类型的结构体，保存于数组容器中，使用时只需在容器中取出预先创建的结构体。至于要创建多少个结构体，可以根据硬件的性能，以及客户端连接的大致数量来决定。

### 3.2 连接管理

服务端程序在接受客户端连接的时候通常调用Accept函数，该函数是一个阻塞函数，一直要到有客户端连接上来后Accept才返回，所以，在该Accept函数没有返回的时候又有另外一个客户端连接请求已经发送过来，则此时该客户端就要处于等待，这里可以采用微软的Winsosk扩展函数AcceptEx函数，AcceptEx是异步的，直接就返回了，所以服务端程序可以发出多个AcceptEx调用，等待客户端连接。AcceptEx的原型如下：

```
BOOL AcceptEx(
    SOCKET        sListenSocket,
    SOCKET        sAcceptSocket,
    PVOID         lpOutputBuffer,
    DWORD         dwReceiveDataLength,
    DWORD         dwLocalAddressLength,
    DWORD         dwRemoteAddressLength,
    LPDWORD       lpdwBytesReceived,
    LPOVERLAPPED  lpOverlapped
);
```

第一个参数sListenSocket为服务端的监听套接字，第二个SOCKET类型的参数值sAcceptSocket是事先用socket()函数创建的一个SOCKET类型的Socket，用于代表客户端Socket，这样，可以预先创建一批套接字，每次调用AcceptEx函数的时候就分配一个套接字做为sAcceptSocket参数[5]。与Accept不同的是，Accept创建的Socket会自动继承监听Socket的属性，而AcceptEx却不会，因此，如果有必要，在AcceptEx成功接受了一个连接之后，就必须调用：

```
int setsockopt(SOCKET s, int level, int optname, const char FAR * optval, int optlen);
```

来做到这一点[4]。此外，服务端程序不能一下子发出太多AcceptEx调用等待客户连接，这样对程序的性能有影响。同时，当服务程序发出的AcceptEx调用耗尽的时候需要新增加AcceptEx调用，可以把FD_ACCEPT事件和一个EVENT关联起来，然后用WaitForSingleObject等待这个Event。当已经发出AccpetEx调用数目耗尽而又有新的客户端需要连接上来，FD_ACCEPT事件将被触发，EVENT变为已传信状态，WaitForSingleObject返回，重新发出足够的AcceptEx调用。

## 4 完成端口模型与多线程模型性能比较

在一台CPU为P4 3.0 GHz，内存为512 M的计算机上模拟1 000个客户端连接。为了尽量做到同步发起请求，笔者编写了一个多线程客户端，而不是简单的循环发起请求，对于测试服务器程序的性能比较有效。服务器硬件参数为两组：第一组，单个超线程CPU P4 3.0 GHz，内存512 M，WinXP操作系统；第二组，双CPU P4 3.0 GHz，内存1 G，WinXP操作系统。表1为两组参数下的性能比较。

**表1 两种模型性能比较**

| 服务器模型名称 | 单个超线程CPU P4 3.0 GHz内存为512 M | | 双P4 3.0 GHz CPU内存为1 G | |
|---|---|---|---|---|
| | 多线程模型 | 完成端口模型 | 多线程模型 | 完成端口模型 |
| 服务器程序占用CPU情况/% | 45 | 22 | 26 | 13 |
| 服务器程序物理内存使用情况/M | 60 | 35 | 58 | 33 |
| 客户端一次连接成功率/% | 90.5 | 96.8 | 93 | 98.5 |

从表1数据可以看出，无论是单个超线程CPU还是双CPU，多线程方式的服务器程序与采用完成端口模型的服务器程序相比，性能差距明显。在单个超线程CPU下，由于是CPU内核指令模拟双CPU效果，因而不是严格的多线程。此时的多线程模型服务器程序CPU占用达45%，而完成端口模型只用了22%的CPU资源。至于物理内存使用情况，由于在多线程模型下系统要为每个线程分配交换空间以及缓存区，因而占用内存较大，而完成端口模型无需太多交换页面，占用的缓存区也少，所以使用的物理内存较少。在双CPU下，多线程模型的服务器程序CPU占用率为26%，而完成端口模型为

13%。此时由于是真正的多 CPU,多线程模型服务器程序性能也提升不少,但是完成端口模型的 4 个(CPU 个数的 2 倍)工作线程加完成消息通知队列的效率更高,占用 CPU 资源更少,内存使用也较少。从客户端的一次连接成功率来看,无论是第一组配置还是第二组配置的服务器硬件,完成端口的成功率都要大大高于多线程模型。这是由于在多线程模式下,CPU 忙于为每一个新客户端创建服务线程,同时还要维护已经创建的类似的大量线程。所以,对于后续的客户端连接请求可能无法马上响应,造成客户端连接超时;然而完成端口模型的情况就要好很多,由于采用了完成消息通知队列的方式,客户端连接请求的成功率大大提高。

## 5　结　语

完成端口模型中有类似于 WEB 应用服务器如 Tomcat 的线程池,但同时还多了一个完成消息队列,线程池中的线程都负责处理 I/O 请求,相对于收到 I/O 请求时创建线程,通过 I/O 完成端口应用程序可以更快更有效的处理异步 I/O 请求[6]。线程调用 GetQueuedCompletionStatus 等待一个完成包到达完成端口,而不是直接等待异步 I/O 操作完成。线程池中所有线程都阻塞在这个函数,当一个完成包到达完成端口时,线程按后入先出(LIFO)的顺序释放。这意味着当一个完成包到达完成端口时,系统释放最后一个在该函数阻塞的线程来处理完成请求。完成端口模型是 Windows 平台下独具特色的高效的用于处理海量连接的理想模型,基于完成端口的服务端程序开发并不复杂,程序结构流程清晰,目前除了大型视频监控系统用户应用服务器采用此模型以外,一些 MMORPG(大型多人在线角色扮演游戏)游戏服务器的架构也是基于完成端口。总之,在 Windows 平台下,处理海量连接的服务器架构,完成端口模型已经是不二之选。

**参考文献:**


[1] 张静华. 应用套接字模型实现网络通信[J]. 山西电子技术, 2004(4):20-21.

[2] ANTHONY Jones, JIM Ohlund. Network Programming for Microsoft Windows (2nd Edition)[M]. Seattle: Mircosoft Press, 2002.

[3] 王暹昊. 用 I/O 完成端口设计多线程的服务应用程序[J]. 计算机与现代化, 2004(3):96-98.

[4] EVERANDFOREVER. 编写完成端口网络服务器的一些说明[EB/OL]. (2004-01-24)[2006-10-15]. http://dev.csdn.net/Develop/article/23/23504.shtm.

[5] HZGMAXWELL. I/O 完成端口[EB/OL]. (2006-04-09)[2006-10-15]. http://hzgmaxwell.blogchina.com/4564081.html.

[6] 郑琪,方思行. 通用多线程服务器的设计与实现[J]. 计算机工程与应用, 2003, 16:147-148.