

完成端口通讯服务器设计 (IOCP Socket Server)

第一章：是谁神化了 IOCP

Windows 系统下的 socket 模型有多种，其中完成例程的效率也是相当高的，其它的也不差（相关模型知识这里不多做介绍，读者可以自己搜索或查阅有关资料）。但是不知道为什么，一提起 IOCP 就会有很多人质疑：IOCP 真的有这么神话吗？

尽管质疑，依然有很多人还是在茫茫网络中苦苦寻找一个完整的 IOCP 源码，希望能够对了解 IOCP 起到事半功倍的作用，不过得到的大多也只是残缺不全的。什么是 IOCP？IOCP 的机制是什么？IOCP 有怎样的性能？当一个人深入了解 IOCP 以后，才解开了它神话之谜：其实它没有什么神话。很多人之所以质疑 IOCP，说出上面那句话的时候，其实是他正在神化 IOCP，主要是因为对 IOCP 不了解，甚至不知道。所以，是谁神化了 IOCP 呢？是那些不了解 IOCP 但又想了解却没有进展的人。

IOCP 主要针对数据吞吐量和连接并发量而设计。有些人使用 IOCP，做的却是堵塞模式的事情：对每个连接自己建立一个发送队列，每次才投递一个发送请求给 IOCP，等该请求已决后才又出列一个再投递给 IOCP。任何一个服务器，能达到怎样的性能，对设计者的要求也是苛刻。根据服务器对性能要求，合理利用通讯模型，才是设计者的关键。如果在一个只有 100 个终端且每个终端每 10 秒才发送一个数据包的服务器系统里，用什么 Socket 模型都一样，甚至用 Win98 系统做都可以。

对于一个服务器而言，需要设计者对内存管理，对网络状况，对操作系统等等都要有深入了解，并具有深厚的技术功底。否则，还会产生更多神化 IOCP 的人。

服务器性能，系统支持是基础，设计者水平是关键。而这个水平条件，没有一个衡量的最终标准，它是永无止境的，会随着时间和经验的积累不断提高。

第二章：内存管理 (AWE)

有牛人曾经说过，服务器玩的就是内存。仔细想想，确实是如此。服务器对内存的需求是巨大的，对内存的要求也是苛刻的。如何在内存管理上下功夫使服务器性能达到一个质的飞跃，是服务器设计中的首要解决的问题。

说到内存，我想刚开始设计服务器的人会说，不就申请释放吗，有什么难呢。从操作步骤来说，确实就这么两个，没有再多了的工作了。当我们采用虚拟内存分配或堆分配从操作系统获取内存的时候，总以为我们获得了足够的内存就可以让服务器安心工作了。但事情并未就这么简单，操作系统在一定条件下，还可以征用已经分配给你的物理内存，它会将你的物理内存数据复制到页交换文件中，然后把本来给你的物理内存再分配给别的进程，当你的进程访问你所获得的虚拟地址集的数据时，它会再找个空（或许也是从别的进程征用）的物理内存，再从页交换文件里面调出你原来的数据放回到新的物理内存里面，并将这个物理内存映射到你申请的虚拟内存地址集内（有关这项内容请参考操作系统的内存管理）。这个过程是相当耗费 CPU 资源且十分缓慢的，尤其是对硬盘虚拟内存文件的读写。其它大道理本文不多说，关于操作系统内存管理的原理可以从《Windows 核心编程》、《Windows 操作系统》、《操作系统》等书籍上了解。

我们可以使用 lookaside lists 技术来重新使用已经分配的内存的，或者使用 SetWorkingSetSize 来设置标志告知操作系统不要交换我的内存，但不外乎多一次操作而已。这个操作到底消耗多少的 CPU 资源，本人也没有考究过，但从性能要求的角度来说，多一事不如少一事。本文讨论的内存管理，将采用 AWE（地址窗口化扩展）的技术，将申请到的物理内存保留为非分页内存，这部分的内存不会被页交换文件所交换，关于 AWE 请参阅以上提到的书籍。

（下面提到的“内存管理”，将仅针对应用程序自己的内存管理功能模块（下文称之为内存管理器）而言，已非上面提到的操作系统的内存管理。）

衡量内存管理器性能的有两个，一个是内存分配时的效率（分配效率），另一个内存交还时的效率（释放效率），亦即二者操作的时间性，这个时间越短那么可以认为它的效率越高。下面的讨论，假定内存管理器是以页为最小分配单位，至于页的大小是多少才合适，稍后再说。

先谈分配效率（下面提到方法仅为本人归纳后的方法，不是学术上的算法）：

1、单链表型

也就是将所有空闲内存块（即空闲内存碎片，下称空闲碎片）组成一个空闲碎片链表。当提出内存分配申请的时候，从这个链表头遍历查找合乎要求的内存或从大的碎片里面分割出来。这个方式简单，但如果空闲碎片多而且小于申请要求的时候，就需要做众多的循环操作。

单链表型排列方式可分为：

- a 按地址高低排列 n
- b 安碎片由小到大排列
- c 不排列（先进先出）

2、多链表型

事实上，多链表型就是将上述按 b 方式排列的单链表，根据一定的大小档次截断而组成的多个链表集（相关算法请参阅上文提到的书籍）。

多链表我将之分为 a、b 型。

a 型如下表：

链表	节点							
ListA: 0~<4K	1K	1K	2K	2K	2K	2K	3K	3K
ListB: 4K~<8K	4K	6K	7K	7K				
ListC: 8K~<16K								
ListD: 16K~<24K	16K	19K	22K	23K				
ListN: ...								

从上表我们看到，0~<4K 的 ListA 有 8 个碎片节点；4K~<8K 的 ListB 有 4 个碎片节点，分别是：4K、6K、7K、7K；8K~<16K 的 ListC 为空；16K~<24K 的 ListD 有 4 个碎片节点，分别是：16K、19K、22K、23K。

假如要分配一个 5K 的内存, 就可以直接从 ListB 查找(如果 ListB 为空, 则继续向更大空间的链表查找)直到底部的 ListN。这样就可以避免遍历 ListA, 假如 ListA 碎片极多的情况下那么就可以节省更多的时间。

b 型如下表

映射表	链表	节点			
a	ListA: 0~<4K				
b	ListB: 4K~<8K	4K	6K	7K	7K
c	ListC: 8K~<16K				
d	ListD: 16K~<24K				
...	ListN: ...	1M	20M	23M	31M

右边的情况不多说, 基本和 a 型的一致。那么左边的映射表是怎么回事呢? 映射表始终没有空的, 如果 ListA 不为空, 那么 a 就指向 ListA, 如果 ListA 为空, 那么它再指向 ListB, 以此类推直到底部的 ListN。如果要分配 9K 内存, 直接从 c 取链表头, 实际上它指向的 ListN, 这样当 N=1000+的时候, 节省的时间就和 a 型相比就非常可观了。

再看看释放效率:

1 针对内存分配方案的第 1 种类型

释放步骤为:

- a 找到比释放内存块的地址低的并是相连接的空闲碎片然后与之合并再重新排列;
- b 不管 a 成立与否, 再找比释放内存块的地址高的并是相连接的空闲碎片然后与之合并再重新排列;
- c 在 a 和 b 不成立的情况, 则按排列规则插入空闲碎片链表。

上述步骤中, 我们发现空闲碎片是一个巨量级的时候效率及其低下。

2 针对内存分配方案第 2 种类型

- a 从 ListA 到 ListN 找到比释放内存块的地址低的并是相连接的空闲碎片然后与之合并再重新排列;
- b 不管 a 成立与否, 再从 ListA 到 ListN 找比释放内存块的地址高的并是相连接的空闲碎片然后与之合并再重新排列;

c 在 a 和 b 不成立的情况，根据释放内存的大小找到归档链表，按排列规则插入该空闲碎片链表。

在这个情况中，工作量比起 1 更加大的多。

3 内存块链表法

这个链表不是指上面所说的空闲碎片链表，而是所有的内存不管空闲的或是使用的，按地址由低到高排列的双向内存块链表。当然，我们不能在释放的时候再去排列所有的内存块，这样的话效率也是相当低的。

如何排列这个链表，可以从分配的时候下功夫：

pBlock 为空闲的碎片块

.....

```
if(pBlock->dwSize > dwSize)
```

```
{//如果空闲块的大小大于要分配的
```

```
//从大的里面切出一块来使用，该块容量减少
```

```
pBlock->dwSize -= dwSize;
```

```
//返回分配的地址
```

```
Result = (PGMEM_BLOCK)pBlock->pAddr;
```

```
//该空闲块向后指向新的空闲地址
```

```
pBlock->pAddr = (char*)Result + dwSize;
```

```
//获取一个新节点
```

```
PGMEM_BLOCK pTmp;
```

```
pTmp = pmbGMemNodePool;
```

```
pmbGMemNodePool = pmbGMemNodePool->pmbNext;
```

```
//将新分配出去的块插在该空闲块前面
```

```
pTmp->pAddr = Result; //分配出去的内存块地址
```

```
pTmp->dwSize = dwSize; //分配出去的内存块大小
```

```
pTmp->pmbNext = pBlock; //下个指针指向被分割的空闲块
```

```
pTmp->pmbPrior = pBlock->pmbPrior;
```

```
if (pTmp->pmbPrior)
```

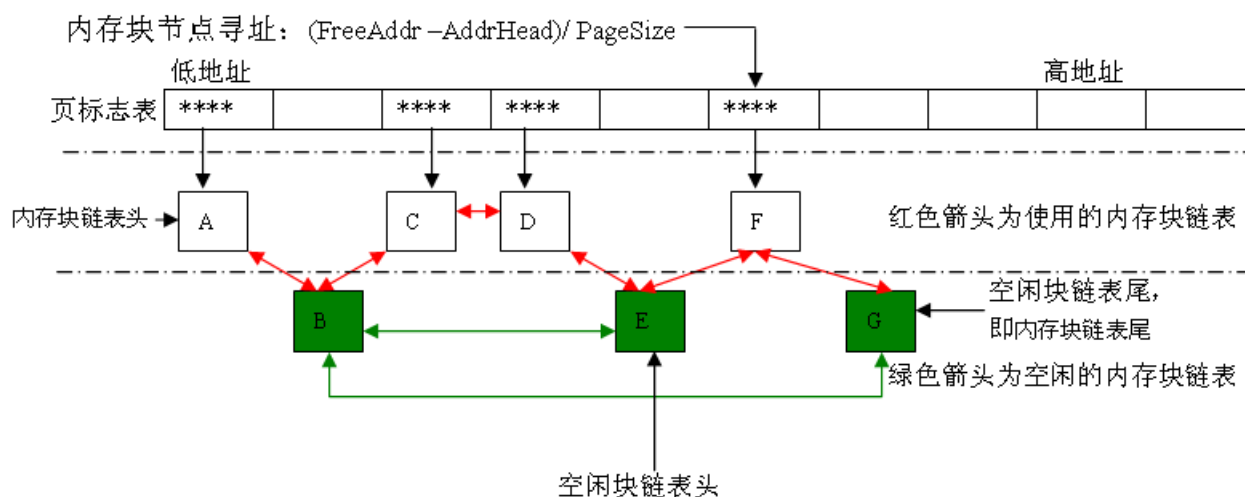
```
pTmp->pmbPrior->pmbNext = pTmp;
```

```
pBlock->pmbPrior = pTmp;
```

```
.....
```

```
}
```

根据上述代码，在分配的时候只需很少的代码量就可以完成了排列要求，这个小小的开支，就可以在释放的时候起到非常高的效率：



从上图看到，内存块链表顺序是：ABCDEFG，空闲块链表顺序是：EBG。通过要释放的块的地址经过计算 $(FreeAddr - AddrHead) / PageSize$ 得出该块在页标志的位置，就可以找到要释放的内存块在内存块链表中的位置是 F，通过上下指针就可以知道与 F 相邻的两个块是 EG，根据标志判断是否为空闲块，然后通过双向链表操作来合并这三个块，这样几乎不要任何遍历操作就可以轻松完成了释放合并的操作。当然如果相邻两个块都没有空闲的，则按排列规则插入空闲块队列。估计没有比这个更好的释放合并内存的算法了。

经过了上面的方法介绍，这个时候我们应该清楚我们该做什么了吧。如果采用内存分配第 2 种“多链表 b 型”的方案和内存释放第 3 种“内存块链表法”的方案，那么这个内存管理器一定是优越的吧。于是开始噼里啪啦的编码……发现一个头痛的问题，即使把内存释放的工作让独立线程来处理，空闲内存块的排列依然消耗很多时间。这个时候不禁问：我们到底在干什么？用这么强大的内存管理器来做操作系统吗？不，我们不是做操作系统，那还有什么方法让内存管理更为简单呢？

来看下表，一个应用服务器对内存动态需求量的统计表：

内存需求类型	内存需求量	占需求量的比率	使用频率
A	4M	低	低
B	1M	低	低
C	8K	低	低
D	4K	中	高
E	1K	中	中
F	512B	高	中
G	256B	高	中

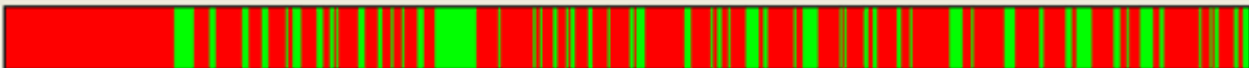
看了上面这个表终于明白，原来在内存需求中类型为 F 和 G 的最多，似乎豁然开朗，假定内存足够的情况下将内存管理器的页大小设置为 512B，那么即使空闲内存块是不排列的，分配内存的效率也是极高的。但对类型 D 这个使用频率最高的内存需求怎么解决呢？任何只想使用一个内存管理器就能做好服务器的想法是错误的。针对很多种内存需求，必须加以分类。针对这个 4K 内存需求，可以使用一个固定大小的内存池来解决，只要是这个内存类型需求的都要在这个内存池操作，比如 Socket I/O 操作的缓冲区等（将在以后介绍）。

通过以上讨论，采用内存分配方案的第 1 种类型 c 方式排列和内存释放方案第 3 种类型，来满足不固定大小的动态内存分配需求，即可一定程度上达到对性能要求的目的。当然，使用内存分配方案的第 2 种 b 类型的，也未必不可，多消耗些 CPU 资源而已。

内存管理没有绝对的方法，所以上述内容仅仅是讨论如何根据内存需求设计内存管理器。

例程截图：

GMem内存管理单元测试 (AWE——地址窗口扩展)



系统信息

可用物理内存: 277192704

物理页大小: 4096

占物理页: 12800

占物理内存: 52428800

物理内存空闲率: 34.8067

GMem信息

GMem页大小: 256

GMem页数量: 204800

GMem内存大小: 52428800

GMem使用量: 42621696

GMem使用率: 81.2944

摧毁

线程数: 4

资源掠夺

序	操作	耗时	平均

GMemMng操作(制造耗时: MS, 每次耗时: US)

定时器: 1000 操作次数: 305 随机分配 ☒ 随机释放 ☒ 定时制造 ☐ 定时释放 ☐
 制造耗时: 0 每次耗时: 0 手动分配 手动释放 制造碎片 全部释放

GMem内存管理单元测试 (AWE——地址窗口扩展)



系统信息

可用物理内存: 276770816

物理页大小: 4096

占物理页: 12800

占物理内存: 52428800

物理内存空闲率: 34.7537

GMem信息

GMem页大小: 256

GMem页数量: 204800

GMem内存大小: 52428800

GMem使用量: 45072640

GMem使用率: 85.9692

摧毁

线程数: 10

停止掠夺

序	操作	耗时	平均
0	2	13	650...
1	2	13	650...
2	2	13	650...
3	1	0	0.00...
4	1	0	0.00...
5	2	13	650...
6	2	13	650...

GMemMng操作(制造耗时: MS, 每次耗时: US)

定时器: 1000 操作次数: 305 随机分配 ☒ 随机释放 ☒ 定时制造 ☐ 定时释放 ☐
 制造耗时: 0 每次耗时: 0 手动分配 手动释放 制造碎片 全部释放

源码说明:

例程源码下载地址: <http://download.csdn.net/source/1607811>

GMem.cpp 和 GMem.h 是内存管理单元的源码文件, 欢迎指正。历程源码没有优化, 能达到测试目的即可, 提供的测试数据仅供参考, 并无实质意义。

第三章：不要迷信 API (单链表的另一种算法)

用这个标题可能会牵强了点。只是因为性能优化中遇到这样的事情，因此用来做标题而已，由此通过一个小事抛出本文介绍的内容：单链表的另一种算法。我本人也做了个 ARM 的 OS 内核，虽然及其简单，但是 OS 的机理大相径庭。操作系统要做到线程同步，需要进入中断（应用程序同步一般是软件中断）才能做到，在进入和退出这两个环节付出的代价相当昂贵的（事实上操作系统的分时处理机制（时钟中断）也在付出昂贵的代价，但那是不可避免的）。

有时候了解的越多，考虑的因素越复杂，做出的结果会越错误。正如上面所说的，线程同步需要使用临界会消耗巨大时间（比起非使用临界而言）。在做内存管理的释放函数的时候，有牛人告诉我，可以使用独立线程来处理释放工作，在释放函数里面给它发个消息就得了，比如使用 `PostThreadMessage` 一个函数就搞定了，又避免了使用临界。刚开始我一听，也感觉真的是很简单，避免了释放的时候释放函数做太多工作而造成堵塞。但随后一想，我向他提出了几个问题：“线程什么时候读取消息？隔 1 毫秒？隔 10 毫秒？会不会因为延时造成内存耗尽的错象？”他又给我推荐了 `SetEvent`，用它及时告知 `Thread` 去处理释放工作。感觉好像“非常有道理”。

不过我没有立即去这么做，而是做了几个代码段去测试（实际对于一个资深的程序员来说根本不需要测试他们是否使用同步机制，本人只想：既然都是使用临界如果那种方式效率高的话还是可取的）：

第一种情形：

```
dwTickCount = GetTickCount();

for(i = 0; i < 10000000; i++)

{

    EnterCriticalSection(&csSection);

    //在这里处理释放工作

    LeaveCriticalSection(&csSection);

}
```

第二种情形：

```
dwTickCount = GetTickCount();

for(i = 0; i < 10000000; i++)

{

    EnterCriticalSection(&csSection);

    //在这里把释放的地址放入处理队列
```



```

LeaveCriticalSection(&csSection);

//告诉处理线程，你该工作了

SetEvent(hEvent);

}

dwTickCount = GetTickCount() - dwTickCount;

第三种情形：

dwTickCount = GetTickCount();

for(i = 0; i < 10000000; i++)

{

    PostThreadMessage(dwThreadId, WM_USER, 0, 0);

    //告诉处理线程，你该工作了

    SetEvent(hEvent);

}

dwTickCount = GetTickCount() - dwTickCount;

```

```
MSG Msg;
```

```
while(PeekMessage(&Msg, 0, 0, 0, PM_REMOVE));
```

上面 3 种情形都有两个线程互相作用，测试结果是：

1、dwTickCount = 4281

2、dwTickCount = 17563

4、dwTickCount = 37297

尽管我的机器主板修了几次，巨慢无比，但在环境一样的前提下得出这么悬殊得结果，已经可以排除 2、3 种情形了（当然如果第 1 种情形中的释放处理工作消耗的时间远大于“2、3 种情形减轻 1 种情形”的话就另当别论了；另外第 3 种情形连续投递这么多消息或多或少内核处理也会耗时）。与其这么折腾，还不如在自己的释放代码上下

功能呢（源码可以看上一篇文章）。

通过这个小事，说明了一个问题，有时候在开发过程中，我们已经无意中过多的信任了 API，总以为看到的代码仅是一行 API 而已，比起（显示式）使用临界来说效率是优秀的（这个错误在以往本人也曾犯过，是在编码的时候一时兴起导致“一念之差”的大意行为）。在多线程环境下，部分 API 是需要进入“临界”这样的机制来同步的，诸如：SendMessage 和 PostMessage，GetMessge 和 PeekMessage。

我们都希望系统能够提供效率更高的 API，来满足我们对服务器性能的需要，比如 Windows 提供了完成端口功能，是不是系统还有更有效的方式？！更有人想直接操作物理内存，甚至有人说：如果在驱动层上下功夫是不是效率更高？毫不隐讳的讲：我也有过这样大胆的想法。但在现在的条件下，我们指望 API 似乎已经没有多大希望。与其……不如优化我们的算法。下面我就介绍一个单向链表的另一种提高效率的算法。（由于本人信息闭塞，这个算法不知道是否已经有人发布过目前我尚未得知。）

在做 IOCP 服务器的时候，为了提高效率，我们采用内存池和连接池的方法以避免频繁向系统索要和归还内存造成系统内存更多的碎片（这种方式效率也是低的）。这个方法好是好，但内存池和连接池一般也都使用了临界来同步，同时对于一个数据区一般也习惯使用一个临界变量来达到同步目的，如果并发性高的话这种同步就会造成堵塞。能不能再提高一点效率以此来降低堵塞的可能性？

假如我们使用两个临界变量来同步一个单向链表会不会把堵塞几率降低一半呢？方法是这样的：单向链表采用后进后出的方式，一个临界负责链表头的同步，另一个临界负责链表尾的同步，但这样的前提是要保证这个链表不为空。

下面是根据上面设想以后的优化算法（后进后出）：

```
PGIO_BUF Glodt_AllocGBuf(void)
```

```
/*说明： 分配一个内存块 GBuf，提供给业务层
```

```
**输入： 无
```

```
**输出： 内存块 GloData 地址*/
```

```
{  
  
    EnterCriticalSection(&GloDataPoolHeadSection);  
  
    //确保链表至少有一个节点，pGloDataPoolHead 不为空，除非不初始化  
  
    //假如一个设计者考虑正常和异常情况能保证内存用之不尽的话，下面这个判断是多余，  
  
    //在以往的设计中本人就曾这么大胆过。  
  
    if (pGloDataPoolHead->pNext) //和常规算法 if (pGloDataPoolHead) 相比并没有多大开支
```

```

{

PGIO_BUF Result;


Result = (PGIO_BUF)pGloDataPoolHead;

pGloDataPoolHead = pGloDataPoolHead->pNext;

dwGloDataPoolUsedCount++;


LeaveCriticalSection(&GloDataPoolHeadSection);

//为什么会这样返回: (char *)Result + sizeof(GIO_DATA_INFO),

//这也是优化算法的一种, 将在以后介绍

return((char *)Result + sizeof(GIO_DATA_INFO));

}else

{

LeaveCriticalSection(&GloDataPoolHeadSection);

return(NULL);

}

}

```

```

void Glodt_FreeGBuf(PGIO_BUF pGloBuf)

```

/*说明: 业务层归还一个内存块 GBuf

**输入: 内存块 GloData 地址

**输出: 无*/

```

{

    EnterCriticalSection(&GloDataPoolTailSection);

    pGloBuf = (char *)pGloBuf - sizeof(GIO_DATA_INFO);

    ((PGIO_DATA)pGloBuf)->pNext = NULL;

    pGloDataPoolTail->pNext = (PGIO_DATA)pGloBuf;

    pGloDataPoolTail = (PGIO_DATA)pGloBuf;

    dwGloDataPoolUsedCount--;

    LeaveCriticalSection(&GloDataPoolTailSection);

}

```

以下是常规的算法（先进先出）：

PGIO_BUF Glodt_AllocGBuf(void)

/*说明： 分配一个内存块 GloBuf， 提供业务层

**输入： 无

**输出： 内存块 GloData 地址*/

```

{

    PGIO_BUF Result;

    EnterCriticalSection(&GloDataPoolSection);

    Result = (PGIO_BUF)pGloDataPoolHead;

    if (pGloDataPoolHead)

    {

        pGloDataPoolHead = pGloDataPoolHead->pNext;
    }
}

```

```

    dwGloDataPoolUsedCount++;

}

LeaveCriticalSection(&GloDataPoolSection);

return((char *)Result + sizeof(GIO_DATA_INFO));

}

```

```

void Glodt_FreeGBuf(PGIO_BUF pGloBuf)

```

/*说明：业务层归还一个内存块 GloBuf

**输入：内存块 GloData 地址

**输出：无*/

```

{

    EnterCriticalSection(&GloDataPoolSection);

    pGloBuf = (char *)pGloBuf - sizeof(GIO_DATA_INFO);

    ((PGIO_DATA)pGloBuf)->pNext = pGloDataPoolHead;

    pGloDataPoolHead = (PGIO_DATA)pGloBuf;

    dwGloDataPoolUsedCount--;

    LeaveCriticalSection(&GloDataPoolSection);

}

```

细心的读者应该会发现，为什么优化算法是这样：

```

if (pGloDataPoolHead->pNext)

```

```

{

```

```

PGIO_BUF Result;

...

LeaveCriticalSection(&GloDataPoolHeadSection);

return((char *)Result + sizeof(GIO_DATA_INFO));

}else

{

    LeaveCriticalSection(&GloDataPoolHeadSection);

    return(NULL);

}

```

和这样的算法有什么区别（这样代码量会更少而又简洁）：

```

PGIO_BUF Result;

if(pGloDataPoolHead->pNext)

{

...

}else

    Result = NULL;

LeaveCriticalSection(&GloDataPoolHeadSection);

return(NULL);

```

这个疑问，只有看了汇编后的代码才能解决了：前面的方法少执行了一两句汇编代码（现在仅谈算法效率，以后有时间再谈代码效率）。

上述单向链表的算法仅为个人搓见，希望能得到牛人指点迷津，使得算法更加有效率。

第四章：一个简单而又灵活的 IOCP 模块

（一）、四个 IOCP 的 API

1、创建完成端口：

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

2、关联完成端口

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

3、获取队列完成状态

```
BOOL WINAPI GetQueuedCompletionStatus(  
    __in HANDLE CompletionPort,  
    __out LPDWORD lpNumberOfBytes,  
    __out PULONG_PTR lpCompletionKey,  
    __out LPOVERLAPPED* lpOverlapped,  
    __in DWORD dwMilliseconds  
);
```

4、投递一个队列完成状态

```
BOOL WINAPI PostQueuedCompletionStatus(  
    __in HANDLE CompletionPort,  
    __in DWORD dwNumberOfBytesTransferred,  
    __in ULONG_PTR dwCompletionKey,  
    __in LPOVERLAPPED lpOverlapped  
);
```

创建和关联完成端口是同一个函数仅是参数传递不一样而已，有关其他参数这里不多重复，请参阅 MSDN。

（二）两个关键的参数

1、dwCompletionKey

在这里，本人把这个完成键扩展为如下定义：

函数指针定义：

```
typedef void(*PFN_ON_GIOCP_ERROR)(void* pCompletionKey, void* pOverlapped);
```



```
typedef void(*PFN_ON_GIOCP_OPER)(DWORD dwBytes, void* pCompletionKey, void* pOverlapped);
```

完成键结构定义:

```
typedef struct _COMPLETION_KEY
{
    PFN_ON_GIOCP_OPER pfnOnIoCpOper;
    PFN_ON_GIOCP_ERROR pfnOnIoCpError;
}GIOCP_COMPLETION_KEY, *PGIOCP_COMPLETION_KEY;
```

在其他 Io 操作的时候, 满足这个既定格式的, 可以在这个数据结构基础之上进行扩展。

2、lpOverlapped

本模块尚未用到, 但在以后的异步 Socket 里面是 Io 操作的关键。

(三) 工作线程源码

```
typedef struct _GWORKER
{
    _GWORKER* pNext;
    DWORD dwThreadId;
    DWORD dwRunCount; //记录工作线程循环次数, 为监视窗口提供数据
    HANDLE hFinished; //表示工作线程已经结束
    void *pData; //每个工作线程独立拥有的数据项
}GWORKER, *PGWORKER;
```

```
DWORD WINAPI GlocpWorkerThread(PGWORKER pWorker)
```

/*说明: 工作线程函数

**输入: 被创建的工作者的结构指针

**输出: */

```
{
    BOOL bResult;
    DWORD dwBytes;
    PGIOCP_COMPLETION_KEY pCompletionKey;
    LPOVERLAPPED pOverlapped;
```

//调用工作线程开始工作的的回调函数, 以便创建每个线程独立拥有的会话, 比如数据库连接会话,

//并使用 Glocp_SetWorkerData 设置该工作线程独立关联的数据项, 比如数据库连接类的指针

```
pfnOnGlocpWorkerThreadBegin((DWORD)pWorker);
```

//死循环标签, 也可以使用 for (;;))

Loop:

```

//等待完成端口事件
bResult = GetQueuedCompletionStatus(hGlopcCompletionPort, &dwBytes, (PULONG_PTR)&pCompletionKey,
&pOverlapped, INFINITE);
//工作线程计数器累加，表示工作线程活动计数
pWorker->dwRunCount++;
//如果完成键是空，表示要结束工作线程
if(!pCompletionKey)
goto End;
//等待完成事件失败
if(bResult)
//调用读写处理函数
pCompletionKey->pfnOnIoCpOper(dwBytes, pCompletionKey, pOverlapped);
else
//调用错误处理函数
pCompletionKey->pfnOnIoCpError(pCompletionKey, pOverlapped);
//继续等待完成事件
goto Loop;
End:
//结束了，调用回调处理函数，比如摧毁数据库库会话对象，
pfnOnGlopcWorkerThreadEnd((DWORD)pWorker);
//设置结束标志
SetEvent(pWorker->hFinished);

return(0);
}

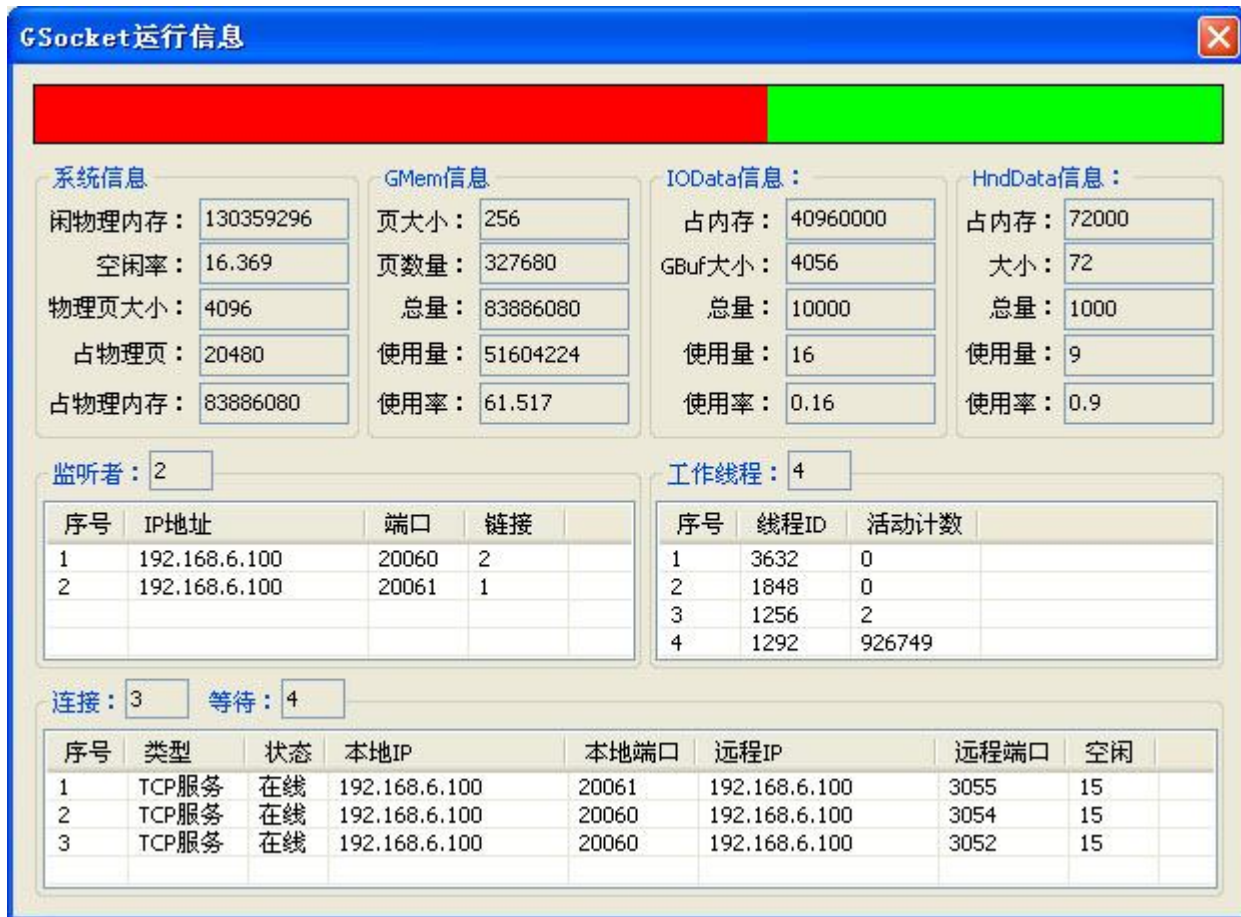
```

从这个工作线程代码来看，它做到了与 Io 操作结果的无关性，它不知道你是读操作还是写操作的返回，或者是 AcceptEx 的返回等等。它看上去似乎非常简单，也正因为这样它的扩展性是相当强大的。如果 TcpServer 功能模块的操作和这个完成端口句柄关联，就可以实现一个面向连接的服务器；如果 TcpClient 功能模块的操作和这个完成端口句柄关联，就可以实现一个[完成端口的客户端](#)，如何与 TcpServer 功能模块一起使用，即可实现[集群服务器](#)之间的通讯；如果 UDP 功能模块的操作和这个完成端口句柄关联，即可实现[完成端口的 P2P](#) 通讯，如果和 TcpServer 功能模块一起使用，即可实现[双协议服务器](#)。

（四）工作线程的不确定性

当我们创建一定数量的工作线程的时候，这个 IOCP 的多线程机制和我们常规的多线程机制有着很大的区别。比如说使用 PostQueuedCompletionStatus 投递一个完成事件，我们无法预知是由哪个线程来处理，这个和常规的线程 WaitForSingleObject 有着本质区别（SetEvent 可以指定某个线程来响应）。并且多个线程的工作率是不平衡，甚至会出现有的工作线程忙得要死有的却闲得要死的现象，即使你设置了并发线程数量（NumberOfConcurrentThreads）足够大（当然在多核或多 CPU 情况下），还是会出现工作量不平衡的现象。

如下图(单核)：



（五）结束工作线程

先看下面代码:

```
PGWORKER pWorker;

//给每个工作者线程抛出结束事件
pWorker = pGlocpWorkerHead;
while(pWorker)
{
    PostQueuedCompletionStatus(hGlocpCompletionPort, 0, 0, NULL);
    pWorker = pWorker->pNext;
}

//等待每个工作线程都结束
pWorker = pGlocpWorkerHead;
while(pWorker)
{
    WaitForSingleObject(pWorker->hFinished, INFINITE);
}
```

```

CloseHandle(pWorker->hFinished);
pWorker = pWorker->pNext;
}

```

由 `PostQueuedCompletionStatus` 抛出一个假完成事件，完成键和重叠结构都是空的，工作线程会这样判断结束：

```

if(!pCompletionKey)
    goto End;

```

并且注意：有多少个工作线程，就调用多少个 `PostQueuedCompletionStatus`，确保工作线程都结束了（`WaitForSingleObject`），才能释放 Worker 占用的内存。另外，在调用 `PostQueuedCompletionStatus` 结束工作线程之前一定确保没有“未决的 Io 请求”。

第五章：妙用 `PerHandleData` 和 `PerIoData`

在部分系统需求下，使用 `PerHandleData` 和 `PerIoData` 来设计 IOCP 服务器是非常有效的。但随之而来的问题也是麻烦的：如何才能高效的申请和回收 `PerHandleData` 和 `PerIoData`？

曾有不少人说动态申请内存和初始化是高效的，优于使用临界锁（或其他锁）的内存池。持这个看法的人，估计和我前面说的“迷信 API”的情形似乎有些类似，不要觉得“显式使用临界”就低效于直接使用 API 获得资源。有个问题我们可以深思一下：在多线程下操作系统是如何同步管理内存的？难道它不需要同步？事实上，它不仅需要同步，而且所做的工作远比你自己的内存池出栈这么简单，因为你的申请是带有“size”的。

我个人认为自己的内存池和动态申请系统内存已经没有必要再去争议。下面我就介绍自己如何设计 `PerHandleData` 和 `PerIoData` 的，希望对初学者有帮助，也希望得到高人指点。

一、`PerIoData`

关于 `PerIoData` 的定义：

```

typedef struct _GIO_DATA_INFO
{
    OVERLAPPED Overlapped;
    WSABUF WSABuf;
    GIO_OPER_TYPE OperType;
    void *pOwner;
}GIO_DATA_INFO, *PGIO_DATA_INFO;

```

```

typedef struct _GIO_DATA
{
    OVERLAPPED Overlapped;
    WSABUF WSABuf;
    GIO_OPER_TYPE OperType;
    void *pOwner;
    char cData[1];
}GIO_DATA, *PGIO_DATA;

```

GIO_DATA_INFO 作为 PerIoData 的固定定义，是 IO 操作必要的信息。GIO_DATA 从 cData 开始才是真正的 IO 数据区，cData[] 不固定多大，默认是 1，可以在系统运行前根据参数设置来设置 PerIoData 的大小，扣除 GIO_DATA_INFO 部分就是 IO 有效数据部分。

另外，我们可以在通讯层严格封装，在业务层向通讯层申请内存的时候，返回的是 cData 地址：

```
PGIO_BUF GloDat_AllocGBuf(void)
{
    EnterCriticalSection(&GloDataPoolHeadCS);
    if(pGloDataPoolHead->pNext)
    {
        PGIO_BUF Result;

        Result = (PGIO_BUF)pGloDataPoolHead;
        pGloDataPoolHead = pGloDataPoolHead->pNext;
        LeaveCriticalSection(&GloDataPoolHeadCS);
        return((char *)Result + dwGBufOffset);
    }else
    {
        LeaveCriticalSection(&GloDataPoolHeadCS);
        return(NULL);
    }
}
```

说明：

“return((char*)Result+dwGBufOffset);”是将返回地址偏移后指向 cData；dwGBufOffset 的值 = sizeof(GIO_DATA_INFO) + sizeof(PackHead)，无通讯协议情况下：sizeof(PackHead) = 0；

业务层发送的时候调用的发送函数：

```
BOOL GCommProt_PostSendGBuf(DWORD dwClientContext, PGIO_BUF pGBuf, DWORD dwBytes)
{
    pGBuf = (PGIO_BUF)((char *)pGBuf - dwGBufOffset);
    ((PGIO_DATA)pGBuf)->WSABuf.len = dwBytes;
    ...
    ...WSASend(...)...
}
```

第一句“pGBuf = (PGIO_BUF)((char *)pGBuf - dwGBufOffset);”已经使 pGBuf 和 PGIO_DATA 对齐了，与申请时“return((char*)Result+dwGBufOffset);”是反操作。这样避免了发送数据的复制过程（当然同一缓冲多发的時候另当别论），即可直接调用 WSASend 投递。

二、PerHandleData

关于 PerHandleData 的定义：

```
typedef struct _GHND_DATA
{
```

```

void* pfnOnlocpOper;
void* pfnOnlocpError;
SOCKET Socket;
_GHND_DATA* pNext;
_GHND_DATA* pPrior;
GHND_TYPE htType;
GHND_STATE hsState;
DWORD dwAddr;
DWORD dwPort;
DWORD dwTickCountActive;
void* pOwner;
void* pData;
} GHND_DATA, *PGHND_DATA;

```

成员 pfnOnlocpOper 和 pfnOnlocpError 的说明请看前一篇。使用双向链表是为了建立客户连接链表时使用，避免删除节点时做遍历工作。pData 是为了建立客户登录模式的时候与 UserInfo 关联：pData =pUserInfo，pUserInfo->dwClientContext=(DWORD)pHndData，可用于服务器主动发送模式。

PerHandleData 的难点是何时回收。同一个 PerHandleData 可以投递多个 PerIoData，为了确保所有 IO 请求都返回的时候，才能回收 PerHandleData，一般有两个做法：1、使用重用计数，即每次 IO 投递都要进行计数操作；2、记录所有 IO 操作的 PerIoData，即在 PerHandleData 定义一个链表头把每个 IO 请求的 PerIoData 放入这个链表，每个 IO 请求返回后再出列。当计数器为 0 或该 IO 链表头为空的时候，即可回收 PerHandleData，以上两个方法都需要同步，且属于“**后回收**”方式。我们都知道，Socket 最频繁的就是收发操作，对于高频率的地方使用同步，无疑降低了不少性能（尽管我们使用高效的内核锁）。本人使用的是“**前回收**”方式，即不必等“所有 IO 请求都返回”在连接断开（错误发生）的时候立即回收 PerHandleData，但要保证在“所有 IO 请求没有返回”前不能再利用刚回收的 PerHandleData。要做到这个目的，就需要延时。当然在这里不能设置延时操作，这样的话会适得其反。我们都知道，后进先出的链栈是不可能保证这个要求的。假如后进后出呢？有这个可能性，但要有一定数量的节点在刚回收的节点前面，就是“等前面的使用完了才会轮到我”即可达到延时的目的。对于建立连接模式的 socket，都需要有 MaxConnection 来限制连接数量，以确保不超出服务器极限资源。假如把 PerHandleData 的数量设置超过 MaxConnection 值，那么即可达到上述目的：

当 MaxConnection=1000 时，设置 PerHandleData 的数量为 1100（多出的 100 这个数可以称之为“**延时量**”）。那么在 PerHandleData 的资源链表里面，至少永远有 100 个节点在空闲着，这个后进后出的链表即可实现 PerHandleData 再利用的延时功能。问题也来了：“**延时量**”设置多少为合适？个人认为极限不低于 MaxConnection 的 10%，50%为宜，保守是 100%以上。即 MaxConnection=5000 的时候，延时量=5000，PerHandleData 数量=10000（牺牲这点内存是值得的）。

要实现这个后进后出的方式，看来使用内核链栈是不可能了。我还是使用“**双锁单链表**”的算法方式，表头和表尾使用不同的临界锁（算法请看前面第三篇）。这个时候有人可能有人又批评了，临界效率太低了，聪明反被聪明误。我倒不觉得，客户连接和断开的频率远比数据收发的低，还是能达到目的的（特定模式的需求除外，比如：连接上来发个数据包就断开）。

第六章：功能强大的 IOCP Socket Servre 模块源码

一、声明

版权声明：

- 1、通讯模块代码版权归作者所有；
- 2、未经许可不得全部或部分用于任何项目开发；
- 3、未经许可不得部分修改后再利用源码。

免责声明：

- 1、由于设计缺陷或其它 Bug 造成的后果，作者不承担责任；
- 2、未经许可的使用作者不提供任何技术支持服务。

权利和义务：

- 1、任何获得源码并发现 Bug 的个人或单位均有义务向作者反映；
- 2、作者保留追究侵权者法律责任的权利。

二、开发背景

部分代码由前项目分离而来，尚未有应用考验，**但对于初学者的学习和进阶会有很大帮助**，通过这个例程可以深入了解 IOCP Socket。性能上尚未有定论，但应该不会令你失望。

三、功能说明

- 1、可以关闭 Socket 的 Buffer；
- 2、可以关闭 MTU（不等待 MTU 满才发送）；
- 3、可以多 IP 或多端口监听；
- 4、可以重用 socket（主动关闭除外）；
- 5、可以 0 缓冲接收（Socket 的 Buffe = 0 时，避免过多的锁定内存页）；
- 6、可以 0 缓冲连接（客户端仅连接，不一定立即发数据）；
- 7、可以条件编译：
 - a、是否使用内核 Singly-linked lists；
 - b、是否使用处理线程（工作线程和处理线程分开）；
 - c、是否使用内核锁来同步链表。
- 8、可以实现集群服务器模式的通讯（有客户端 socket）；
- 9、可以单独设置每个连接的 Data 项来实现连接和 Userinfo 的关联；
- 10、每个线程有 OnBegin 和 OnEnd，用于设置线程独立的对象（数据库会话对象）；
- 11、可以提供详细的运行情况，便于了解 IOCP 下的机制，以及进行调试分析；
- 12、可以发起巨量连接和数据（需要硬件配置来支持）。

四、缺陷

- 1、不支持 UDP；
- 2、不兼容 IPv6；
- 3、不带通讯协议，无法处理粘包；
- 4、工作线程和处理线程隔离还不是很明确；
- 5、设计尚需再完善和优化。

五、通讯速率测试部分截图

服务器端运行信息:

GSocket运行信息

系统信息

闲物理内存：212062208
空闲率：26.6284
物理页大小：4096
占物理页：20480
占物理内存：83886080

GMem信息

页大小：256
页数量：327680
总量：83886080
使用量：51589376
使用率：61.4993

IOData信息：

占内存：40960000
GBuf大小：4060
总量：10000
使用量：805
使用率：8.05

HndData信息：

占内存：57200
大小：52
总量：1100
使用量：109
使用率：9.90909

监听者：2

线程总数：6

工作线程：4

序号	IP地址	端口	链接
1	192.168.6.100	20020	100
2	192.168.6.100	20021	0

序号	线程名	状态	活动计数
3	工作者:3856	1	345390
4	工作者:3212	1	354923
5	工作者:2124	1	344737
6	工作者:396	1	346644

连接：100

等待：7

列表：100

序号	类型	状态	本地IP	本地端口	远程IP	远程端口	空闲
1	TCP服务	在线	192.168.6.100	20020	192.168.6.102	3221	31
2	TCP服务	在线	192.168.6.100	20020	192.168.6.102	3219	78
3	TCP服务	在线	192.168.6.100	20020	192.168.6.102	3218	15
4	TCP服务	在线	192.168.6.100	20020	192.168.6.102	3217	15

客户端设置：

GSocket测试例程（完成端口通讯模块——IOCP Socket）

GSocket设置

运行信息

GMem页大小：256
GMem总容量：83886080
GBuf大小：4060
IOData总量：10000
HndData总量：1100
接收投递：6

关闭MTU：☒
0缓冲接受：☐
0缓冲接收：☐
Socket缓冲：8192
工作线程：6
并发线程：6

Server设置

接受超时：5
空闲超时：60
最大连接：1000
接受投递：6
转发延时循环：0

自收自发：☒
本地IP：192.168.6.102
本地端口：20020
TCP监听(尚未监听)
UDP监听

Client设置：

发送间隔：1
本地IP：192.168.6.102

包大小：3996
本地端口：20030

心跳：50
远程IP：192.168.6.100

连接数：100
远程端口：20020
TCP连接
UDP连接

停止服务

客户：100

随机断开 ☐ 复位

发送：160962876

接收：158961199

收发：319924075

总时间：26708

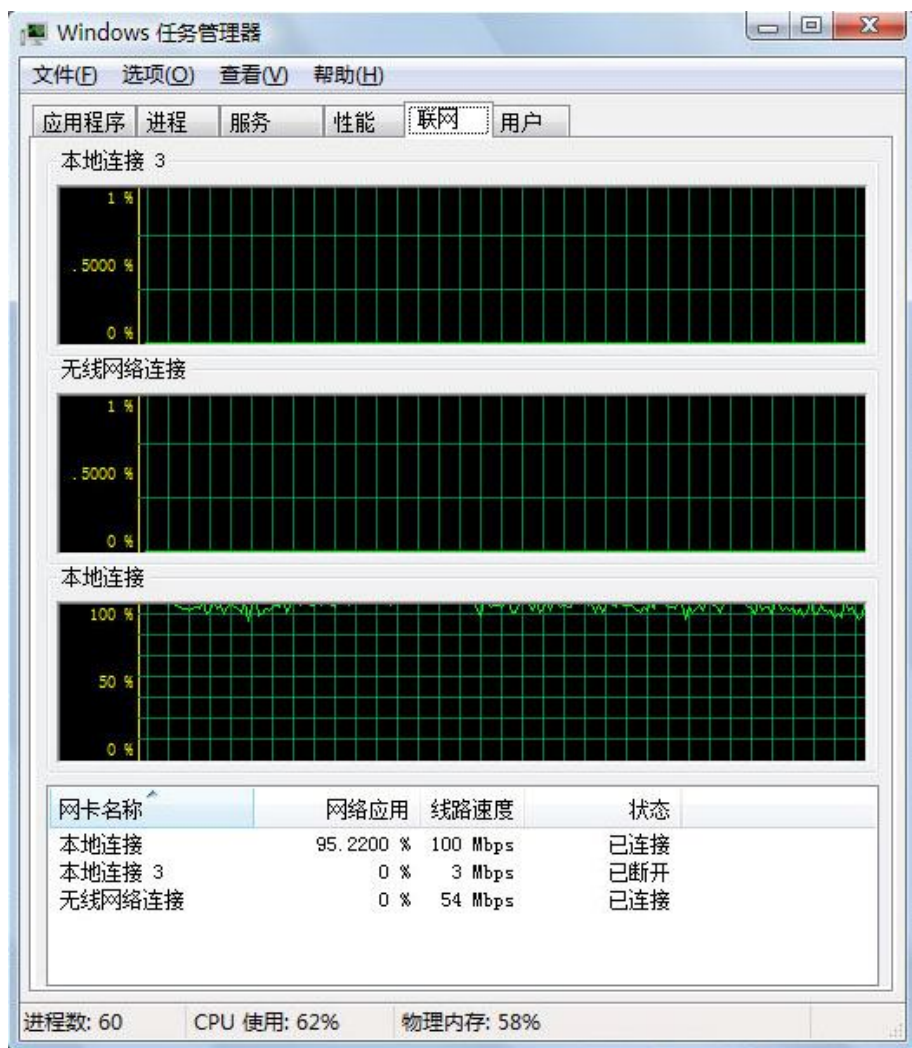
速率：11978.5859

序	类型	状态	远程	发送	接收	收发	双程耗时	双程速率	错误
1	TCP客户	在线	192....	10765224	10715972	21481196	577	13.8509...	2624
2	TCP客户	在线	192....	10869120	10743936	21613056	577	13.8509...	2667
3	TCP客户	在线	192....	10497492	10753807	21251299	577	13.8509...	2666
4	TCP客户	在线	192....	10849140	10759716	21608856	577	13.8509...	2661
5	TCP客户	在线	192....	10789200	10750620	21539820	546	14.6373...	2651

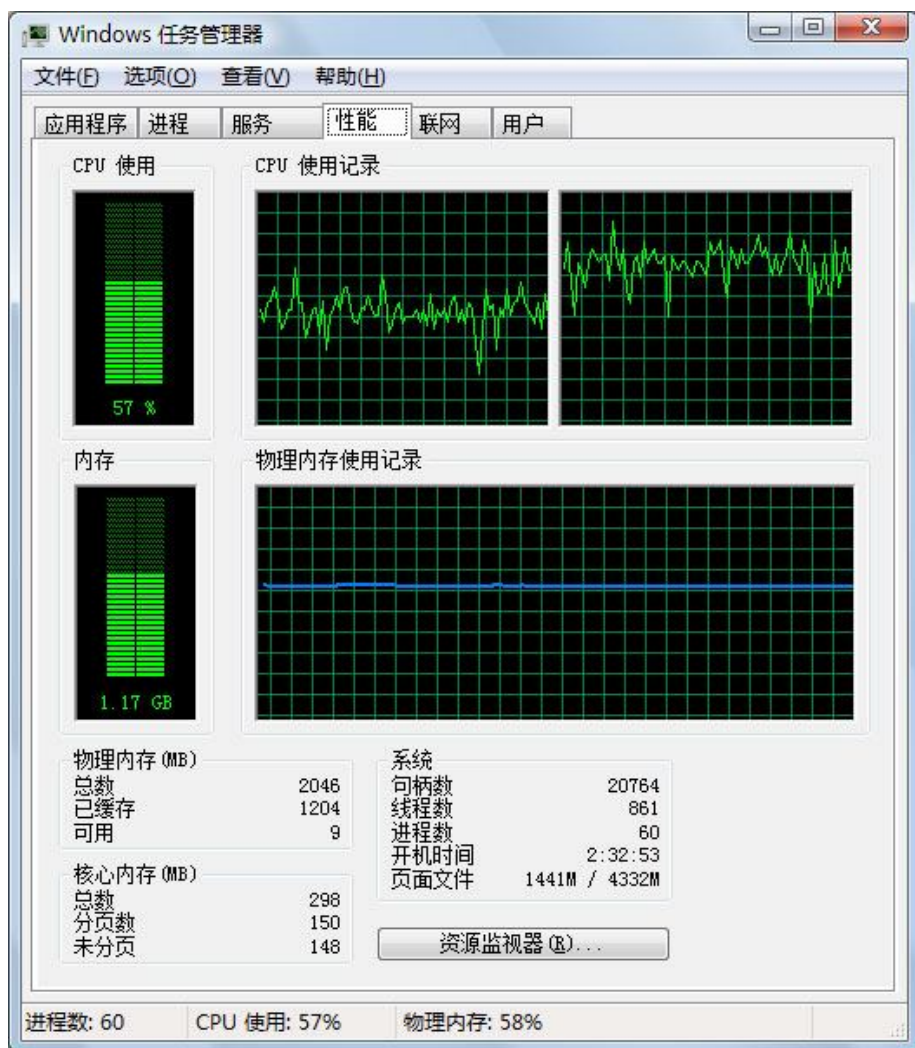
客户端运行信息：



客户端网络利用率：



客户端 CPU 利用率:



下载连接:

<http://d.download.csdn.net/down/1679785/guestcode>