零拷贝技术研究与实现

2007-09-12 13:52

|  |
| --- |
| 作者：梁健（firstdot） E-MAIL：firstdot@163.com  感谢王超、史晓龙的共同研究与大力帮助 一．基本概念 零 拷贝（zero-copy）基本思想是：数据报从网络设备到用户程序空间传递的过程中，减少数据拷贝次数，减少系统调用，实现CPU的零参与，彻底消除 CPU在这方面的负载。实现零拷贝用到的最主要技术是DMA数据传输技术和内存区域映射技术。如图1所示，传统的网络数据报处理，需要经过网络设备到操作 系统内存空间，系统内存空间到用户应用程序空间这两次拷贝，同时还需要经历用户向系统发出的系统调用。而零拷贝技术则首先利用DMA技术将网络数据报直接 传递到系统内核预先分配的地址空间中，避免CPU的参与；同时，将系统内核中存储数据报的内存区域映射到检测程序的应用程序空间（还有一种方式是在用户空 间建立一缓存，并将其映射到内核空间，类似于linux系统下的kiobuf技术），检测程序直接对这块内存进行访问，从而减少了系统内核向用户空间的内 存拷贝，同时减少了系统调用的开销，实现了真正的“零拷贝”。   图1 传统数据处理与零拷贝技术之比较 二．实现 在redhat7.3 上通过修改其内核源码中附带的8139too.c完成零拷贝的试验，主要想法是：在8139too网卡驱动模块启动时申请一内核缓存，并建立一数据结构对 其进行管理，然后试验性的向该缓存写入多个字符串数据，最后通过proc文件系统将该缓存的地址传给用户进程；用户进程通过读proc文件系统取得缓存地 址并对该缓存进行地址映射，从而可以从其中读取数据。哈哈，为了偷懒，本文只是对零拷贝思想中的地址映射部分进行试验，而没有实现DMA数据传输（太麻烦 了，还得了解硬件），本试验并不是一个IDS产品中抓包模块的一部分，要想真正在IDS中实现零拷贝，除了DMA外，还有一些问题需考虑，详见本文第三节 的分析。以下为实现零拷贝的主要步骤，详细代码见附录。  步骤一：修改网卡驱动程序 a．在网卡驱动程序中申请一块缓存：由于在linux2.4.X内核中支持的最大可分配连续缓存大小为2M，所以如果需要存储更大量的网络数据报文，则需要分配多块非连续的缓存，并使用链表、数组或hash表来对这些缓存进行管理。  #define PAGES\_ORDER 9 unsigned long su1\_2 su1\_2 = \_\_get\_free\_pages(GFP\_KERNEL,PAGES\_ORDER);  b. 向缓存中写入数据：真正IDS产品中的零拷贝实现应该是使用DMA数据传输把网卡硬件接收到的包直接写入该缓存。作为试验，我只是向该缓存中写入几个任意 的字符串，如果不考虑DMA而又想向缓存中写入真正的网络数据包，可以在8139too.c的rtl8139\_rx\_interrupt()中调用 netif\_rx()后插入以下代码：  //put\_pkt2mem\_n++; //包个数 //put\_mem(skb->data,pkt\_size); 其中put\_pkt2mem\_n变量和put\_mem函数见附录。  c. 把该缓存的物理地址传到用户空间：由于在内核中申请的缓存地址为虚拟地址，而在用户空间需要得到的是该缓存的物理地址，所以首先要进行虚拟地址到物理地址 的转换，在linux系统中可以使用内核虚拟地址减3G来获得对应的物理地址。把缓存的地址传到用户空间需要在内核与用户空间进行少量数据传输，这可以使 用字符驱动、proc文件系统等方式实现，在这里采用了proc文件系统方式。  int read\_procaddr(char \*buf,char \*\*start,off\_t offset,int count,int \*eof,void \*data) {      sprintf(buf,"%u\n",\_\_pa(su1\_2));      \*eof = 1;      return 9; } create\_proc\_read\_entry("nf\_addr",0,NULL,read\_procaddr,NULL);  步骤二：在用户程序中实现对共享缓存的访问 a.读取缓存地址：通过直接读取proc文件的方式便可获得。  char addr[9]; int fd\_procaddr; unsigned long ADDR; fd\_procaddr = open("/proc/nf\_addr",O\_RDONLY); read(fd\_procaddr,addr,9); ADDR = atol(addr);  b.把缓存映射到用户进程空间中：在用户进程中打开/dev/mem设备(相当于物理内存），使用mmap把网卡驱动程序申请的缓存映射到自己的进程空间，然后就可以从中读取所需要的网络数据包了。  char \*su1\_2; int fd; fd=open("/dev/mem",O\_RDWR);     su1\_2 = mmap(0,PAGES\*4\*1024, PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, ADDR);  三．分析      零拷贝中存在的最关键问题是同步问题，一边是处于内核空间的网卡驱动向缓存中写入网络数据包，一边是用户进程直接对缓存中的数据包进行分析（注意，不是拷 贝后再分析），由于两者处于不同的空间，这使得同步问题变得更加复杂。缓存被分成多个小块，每一块存储一个网络数据包并用一数据结构表示，本试验在包数据 结构中使用标志位来标识什么时候可以进行读或写，当网卡驱动向包数据结构中填入真实的包数据后便标识该包为可读，当用户进程对包数据结构中的数据分析完后 便标识该包为可写，这基本解决了同步问题。然而，由于IDS的分析进程需要直接对缓存中的数据进行入侵分析，而不是将数据拷贝到用户空间后再进行分析，这 使得读操作要慢于写操作，有可能造成网卡驱动无缓存空间可以写，从而造成一定的丢包现象，解决这一问题的关键在于申请多大的缓存，太小的缓存容易造成丢 包，太大的缓存则管理麻烦并且对系统性能会有比较大的影响。  四．附录 a.     8139too.c中加入的代码  /\*add\_by\_liangjian for zero\_copy\*/ #include <linux/wrapper.h> #include <asm/page.h> #include <linux/slab.h> #include <linux/proc\_fs.h> #define PAGES\_ORDER 9 #define PAGES 512 #define MEM\_WIDTH     1500 /\*added\*/  /\*add\_by\_liangjian for zero\_copy\*/ struct MEM\_DATA {      //int key;      unsigned short width;/\*缓冲区宽度\*/      unsigned short length;/\*缓冲区长度\*/      //unsigned short wtimes;/\*写进程记数,预留，为以后可以多个进程写\*/      //unsigned short rtimes;/\*读进程记数,预留，为以后可以多个进程读\*/      unsigned short wi;/\*写指针\*/      unsigned short ri;/\*读指针\*/ } \* mem\_data; struct MEM\_PACKET {      unsigned int len;      unsigned char packetp[MEM\_WIDTH - 4];/\*sizeof(unsigned int) == 4\*/ }; unsigned long su1\_2;/\*缓存地址\*/ /\*added\*/  /\*add\_by\_liangjian for zero\_copy\*/ //删除缓存 void del\_mem() {      int pages = 0;      char \*addr;      addr = (char \*)su1\_2;      while (pages <=PAGES -1)      {          mem\_map\_unreserve(virt\_to\_page(addr));          addr = addr + PAGE\_SIZE;          pages++;      }      free\_pages(su1\_2,PAGES\_ORDER);     }     void init\_mem() /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*                   初始化缓存 \*       输入:   aMode:     缓冲区读写模式:   r,w         \* \*       返回:   00:     失败                         \* \*               >0:     缓冲区地址                   \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ {      int i;      int pages = 0;      char \*addr;      char \*buf;      struct MEM\_PACKET \* curr\_pack;           su1\_2 = \_\_get\_free\_pages(GFP\_KERNEL,PAGES\_ORDER);      printk("[%x]\n",su1\_2);      addr = (char \*)su1\_2;      while (pages <= PAGES -1)      {          mem\_map\_reserve(virt\_to\_page(addr));//需使缓存的页面常驻内存          addr = addr + PAGE\_SIZE;          pages++;      }      mem\_data = (struct MEM\_DATA \*)su1\_2;      mem\_data[0].ri = 1;            mem\_data[0].wi = 1;            mem\_data[0].length = PAGES\*4\*1024 / MEM\_WIDTH;            mem\_data[0].width = MEM\_WIDTH;      /\* initial su1\_2 \*/      for(i=1;i<=mem\_data[0].length;i++)      {          buf = (void \*)((char \*)su1\_2 + MEM\_WIDTH \* i);          curr\_pack = (struct MEM\_PACKET \*)buf;          curr\_pack->len = 0;      }     } int put\_mem(char \*aBuf,unsigned int pack\_size) /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*                 写缓冲区子程序                                 \* \*       输入参数     :   aMem:   缓冲区地址                       \* \*                       aBuf:   写数据地址                       \* \*       输出参数     :   <=00 :   错误                             \* \*                       XXXX :   数据项序号                       \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ {      register int s,i,width,length,mem\_i;      char \*buf;      struct MEM\_PACKET \* curr\_pack;       s = 0;      mem\_data = (struct MEM\_DATA \*)su1\_2;      width   = mem\_data[0].width;      length = mem\_data[0].length;      mem\_i   = mem\_data[0].wi;      buf = (void \*)((char \*)su1\_2 + width \* mem\_i);       for (i=1;i<length;i++){          curr\_pack = (struct MEM\_PACKET \*)buf;              if   (curr\_pack->len == 0){                      memcpy(curr\_pack->packetp,aBuf,pack\_size);                      curr\_pack->len = pack\_size;;                  s = mem\_i;              mem\_i++;                      if   (mem\_i >= length)                          mem\_i = 1;                  mem\_data[0].wi = mem\_i;                  break;              }              mem\_i++;              if   (mem\_i >= length){                      mem\_i = 1;                      buf = (void \*)((char \*)su1\_2 + width);              }              else buf = (char \*)su1\_2 + width\*mem\_i;          }       if(i >= length)              s = 0;      return s; } // proc文件读函数 int read\_procaddr(char \*buf,char \*\*start,off\_t offset,int count,int \*eof,void \*data) {      sprintf(buf,"%u\n",\_\_pa(su1\_2));      \*eof = 1;      return 9; } /\*added\*/  在8139too.c的rtl8139\_init\_module()函数中加入以下代码： /\*add\_by\_liangjian for zero\_copy\*/      put\_pkt2mem\_n = 0;      init\_mem();      put\_mem("data1dfadfaserty",16);      put\_mem("data2zcvbnm",11);      put\_mem("data39876543210poiuyt",21);      create\_proc\_read\_entry("nf\_addr",0,NULL,read\_procaddr,NULL); /\*added \*/      在8139too.c的rtl8139\_cleanup\_module()函数中加入以下代码： /\*add\_by\_liangjian for zero\_copy\*/      del\_mem();      remove\_proc\_entry("nf\_addr",NULL); /\*added\*/      b．用户空间读取缓存代码  #include <stdio.h> #include <unistd.h> #include <sys/stat.h> #include <sys/mman.h> #include <fcntl.h> #define PAGES 512 #define MEM\_WIDTH 1500 struct MEM\_DATA {      //int key;      unsigned short width;/\*缓冲区宽度\*/      unsigned short length;/\*缓冲区长度\*/      //unsigned short wtimes;/\*写进程记数,预留，为以后可以多个进程写\*/      //unsigned short rtimes;/\*读进程记数,预留，为以后可以多个进程读\*/      unsigned short wi;/\*写指针\*/      unsigned short ri;/\*读指针\*/ } \* mem\_data;  struct MEM\_PACKET {      unsigned int len;      unsigned char packetp[MEM\_WIDTH - 4];/\*sizeof(unsigned int) == 4\*/ };  int get\_mem(char \*aMem,char \*aBuf,unsigned int \*size) /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*                 读缓冲区子程序                                 \* \*       输入参数     :   aMem:   缓冲区地址                       \* \*                       aBuf:   返回数据地址, 其数据区长度应大于\* \*                               缓冲区宽度                       \* \*       输出参数     :   <=00 :   错误                             \* \*                       XXXX :   数据项序号                       \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ {      register int i,s,width,length,mem\_i;      char     \*buf;      struct MEM\_PACKET \* curr\_pack;       s = 0;      mem\_data = (void \*)aMem;      width   = mem\_data[0].width;      length = mem\_data[0].length;      mem\_i   = mem\_data[0].ri;      buf = (void \*)(aMem + width \* mem\_i);       curr\_pack = (struct MEM\_PACKET \*)buf;      if   (curr\_pack->len != 0){/\*第一个字节为0说明该部分为空\*/              memcpy(aBuf,curr\_pack->packetp,curr\_pack->len);              \*size = curr\_pack->len;              curr\_pack->len = 0;              s = mem\_data[0].ri;              mem\_data[0].ri++;              if(mem\_data[0].ri >= length)                      mem\_data[0].ri = 1;              goto ret;          }           for (i=1;i<length;i++){              mem\_i++;/\*继续向后找，最糟糕的情况是把整个缓冲区都找一遍\*/              if   (mem\_i >= length)                  mem\_i = 1;              buf = (void \*)(aMem + width\*mem\_i);              curr\_pack = (struct MEM\_PACKET \*)buf;              if   (curr\_pack->len == 0)                      continue;              memcpy(aBuf,curr\_pack->packetp,curr\_pack->len);              \*size = curr\_pack->len;              curr\_pack->len = 0;              s = mem\_data[0].ri = mem\_i;              mem\_data[0].ri++;              if(mem\_data[0].ri >= length)              mem\_data[0].ri = 1;              break;          }       ret:      return s; }  int main() {      char \*su1\_2;      char receive[1500];      int i,j;      int fd;      int fd\_procaddr;      unsigned int size;      char addr[9];      unsigned long ADDR;           j = 0;      /\*open device 'mem' as a media to access the RAM\*/      fd=open("/dev/mem",O\_RDWR);          fd\_procaddr = open("/proc/nf\_addr",O\_RDONLY);      read(fd\_procaddr,addr,9);      ADDR = atol(addr);      close(fd\_procaddr);      printf("%u[%8lx]\n",ADDR,ADDR);      /\*Map the address in kernel to user space, use mmap function\*/      su1\_2 = mmap(0,PAGES\*4\*1024, PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, ADDR);      perror("mmap");      while(1)      {          bzero(receive,1500);          i = get\_mem(su1\_2,receive,&size);          if (i != 0)          {              j++;              printf("%d:%s[size = %d]\n",j,receive,size);          }              else           {              printf("there have no data\n");              munmap(su1\_2,PAGES\*4\*1024);              close(fd);              break;          }      }      while(1); }  五．参考文献 1．CHRISTIAN KURMANN, FELIX RAUCH ,THOMAS M. STRICKER.  Speculative Defragmentation - Leading Gigabit Ethernet to True Zero-Copy Communication 2．ALESSANDRO RUBINI,JONATHAN CORBET.《LINUX DEVICE DRIVERS 2》,O’Reilly & Associates 2002. 3．胡希明,毛德操.《LINUX 内核源代码情景分析》,浙江大学出版社 2001   关 于作者：梁健，华北计算技术研究所在读硕士研究生，研究方向：信息安全。论文开题为《基于系统调用分析的主机异常入侵检测与防御》。对IDS有两年多的研 究经验，熟悉linux内核，熟悉linux c/c++编程、win32 API编程，对网络和操作系统安全感兴趣 |

通过零拷贝实现有效数据传输

**零拷贝，零开销**

[Sathish K. Palaniappan](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#author1), 系统软件工程师, IBM

[Pramod B. Nagaraja](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#author2), 助理系统软件工程师, IBM

**简介：** 本文解释了如何通过一种称为*零拷贝* 的方法来提高运行于 Linux® 和 UNIX® 平台上的 I/O 密集型 Java™ 应用程序的性能。零拷贝不仅消除了中间缓冲区之间的冗余数据拷贝，还减少了用户空间和内核空间之间的上下文切换次数。

**本文的标签:**  [零拷贝](javascript:launchTagWindow('/developerworks/mydeveloperworks/bookmarks/html?tag=%25E9%259B%25B6%25E6%258B%25B7%25E8%25B4%259D'))

[标记本文！](http://www.ibm.com/developerworks/cn/java/j-zerocopy/)

**发布日期：** 2008 年 9 月 23 日   
**级别：** 中级   
**其他语言版本：** [英文](http://www.ibm.com/developerworks/java/library/j-zerocopy/)   
**访问情况** 1163 次浏览   
**建议:** 1 ([查看或添加评论](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#icomments))

[1 star2 stars3 stars4 stars5 stars](javascript:void(0);) 平均分 （共 3 个评分 ）

很多 Web 应用程序都会提供大量的静态内容，其数量多到相当于读完整个磁盘的数据再将同样的数据写回响应套接字（socket）。此动作看似只需较少的 CPU 活动，但它的效率非常低：首先内核读出全盘数据，然后将数据跨越内核用户推到应用程序，然后应用程序再次跨越内核用户将数据推回，写出到套接字。应用程序实际上在这里担当了一个不怎么高效的中介角色，将磁盘文件的数据转入套接字。

数据每遍历用户内核一次，就要被拷贝一次，这会消耗 CPU 周期和内存带宽。幸运的是，您可以通过一个叫 *零拷贝* — 很贴切 — 的技巧来消除这些拷贝。使用零拷贝的应用程序要求内核直接将数据从磁盘文件拷贝到套接字，而无需通过应用程序。零拷贝不仅大大地提高了应用程序的性能，而且还减少了内核与用户模式间的上下文切换。

Java 类库通过 java.nio.channels.FileChannel 中的 transferTo() 方法来在 Linux 和 UNIX 系统上支持零拷贝。可以使用transferTo() 方法直接将字节从它被调用的通道上传输到另外一个可写字节通道上，数据无需流经应用程序。本文首先展示了通过传统拷贝语义进行的简单文件传输引发的开销，然后展示了使用 transferTo() 零拷贝技巧如何提高性能。

**数据传输：传统方法**

考虑一下从一个文件中读出数据并将数据传输到网络上另一程序的场景（这个场景表述出了很多服务器应用程序的行为，包括提供静态内容的 Web 应用程序、FTP 服务器、邮件服务器等）。操作的核心在清单 1 的两个调用中（参见 [下载](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#download)，查找完整示例代码的链接）：

**清单 1. 把字节从文件拷贝到套接字**

|  |
| --- |
| File.read(fileDesc, buf, len);  Socket.send(socket, buf, len); |

清单 1 的概念很简单，但实际上，拷贝的操作需要四次用户模式和内核模式间的上下文切换，而且在操作完成前数据被复制了四次。图 1 展示了数据是如何在内部从文件移动到套接字的：

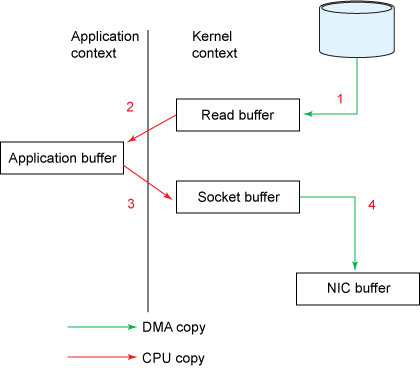
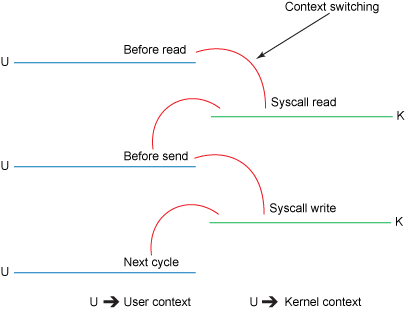
**图 1. 传统的数据拷贝方法**  


图 2 展示了上下文切换：

**图 2. 传统上下文切换**  


这里涉及的步骤有：

1. read() 调用（参见 [图 2](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#fig2)）引发了一次从用户模式到内核模式的上下文切换。在内部，发出 sys\_read()（或等效内容）以从文件中读取数据。直接内存存取（direct memory access，DMA）引擎执行了第一次拷贝（参见 [图 1](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#fig1)），它从磁盘中读取文件内容，然后将它们存储到一个内核地址空间缓存区中。
2. 所需的数据被从读取缓冲区拷贝到用户缓冲区，read() 调用返回。该调用的返回引发了内核模式到用户模式的上下文切换（又一次上下文切换）。现在数据被储存在用户地址空间缓冲区。
3. send() 套接字调用引发了从用户模式到内核模式的上下文切换。数据被第三次拷贝，并被再次放置在内核地址空间缓冲区。但是这一次放置的缓冲区不同，该缓冲区与目标套接字相关联。
4. send() 系统调用返回，结果导致了第四次的上下文切换。DMA 引擎将数据从内核缓冲区传到协议引擎，第四次拷贝独立地、异步地发生 。

使用中间内核缓冲区（而不是直接将数据传输到用户缓冲区）看起来可能有点效率低下。但是之所以引入中间内核缓冲区的目的是想提高性能。在读取方面使用中间内核缓冲区，可以允许内核缓冲区在应用程序不需要内核缓冲区内的全部数据时，充当 “预读高速缓存（readahead cache）” 的角色。这在所需数据量小于内核缓冲区大小时极大地提高了性能。在写入方面的中间缓冲区则可以让写入过程异步完成。

不幸的是，如果所需数据量远大于内核缓冲区大小的话，这个方法本身可能成为一个性能瓶颈。数据在被最终传入到应用程序前，在磁盘、内核缓冲区和用户缓冲区中被拷贝了多次。

零拷贝通过消除这些冗余的数据拷贝而提高了性能。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#ibm-pcon)

**数据传输：零拷贝方法**

再次检查 [传统场景](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#trad)，您就会注意到第二次和第三次拷贝根本就是多余的。应用程序只是起到缓存数据并将其传回到套接字的作用而以，别无他用。数据可以直接从读取缓冲区传输到套接字缓冲区。transferTo() 方法就能够让您实现这个操作。清单 2 展示了transferTo() 的方法签名：

**清单 2. transferTo() 方法**

|  |
| --- |
| public void transferTo(long position, long count, WritableByteChannel target); |

transferTo() 方法将数据从文件通道传输到了给定的可写字节通道。在内部，它依赖底层操作系统对零拷贝的支持；在 UNIX 和各种 Linux 系统中，此调用被传递到 sendfile() 系统调用中，如清单 3 所示，清单 3 将数据从一个文件描述符传输到了另一个文件描述符：

**清单 3. sendfile() 系统调用**

|  |
| --- |
| #include <sys/socket.h>  ssize\_t sendfile(int out\_fd, int in\_fd, off\_t \*offset, size\_t count); |

[清单 1](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#listing1) 中的 file.read() 和 socket.send() 调用动作可以替换为一个单一的 transferTo() 调用，如清单 4 所示：

**清单 4. 使用 transferTo() 将数据从磁盘文件拷贝到套接字**

|  |
| --- |
| transferTo(position, count, writableChannel); |

图 3 展示了使用 transferTo() 方法时的数据路径：

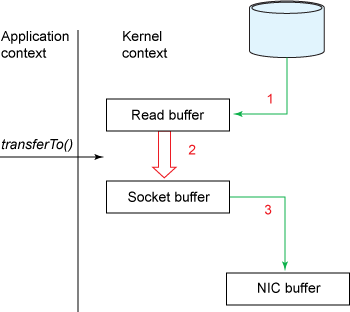
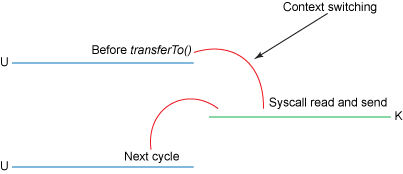
**图 3. 使用 transferTo() 方法的数据拷贝**  


图 4 展示了使用 transferTo() 方法时的上下文切换：

**图 4. 使用 transferTo() 方法的上下文切换**  


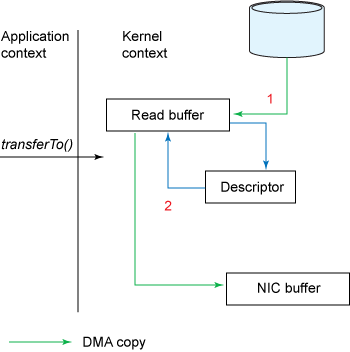
使用 [清单 4](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#listing4) 所示的 transferTo() 方法时的步骤有：

1. transferTo() 方法引发 DMA 引擎将文件内容拷贝到一个读取缓冲区。然后由内核将数据拷贝到与输出套接字相关联的内核缓冲区。
2. 数据的第三次复制发生在 DMA 引擎将数据从内核套接字缓冲区传到协议引擎时。

改进的地方：我们将上下文切换的次数从四次减少到了两次，将数据复制的次数从四次减少到了三次（其中只有一次涉及到了 CPU）。但是这个代码尚未达到我们的零拷贝要求。如果底层网络接口卡支持*收集操作* 的话，那么我们就可以进一步减少内核的数据复制。在 Linux 内核 2.4 及后期版本中，套接字缓冲区描述符就做了相应调整，以满足该需求。这种方法不仅可以减少多个上下文切换，还可以消除需要涉及 CPU 的重复的数据拷贝。对于用户方面，用法还是一样的，但是内部操作已经发生了改变：

1. transferTo() 方法引发 DMA 引擎将文件内容拷贝到内核缓冲区。
2. 数据未被拷贝到套接字缓冲区。取而代之的是，只有包含关于数据的位置和长度的信息的描述符被追加到了套接字缓冲区。DMA 引擎直接把数据从内核缓冲区传输到协议引擎，从而消除了剩下的最后一次 CPU 拷贝。

图 5 展示了结合使用 transferTo() 方法和收集操作的数据拷贝：

**图 5. 结合使用 transferTo() 和收集操作时的数据拷贝**  


[**回页首**](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#ibm-pcon)

**构建一个文件服务器**

接下来就让我们实际应用一下零拷贝，在客户机和服务器间传输文件（参见 [下载](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#download)，查找示例代码）。TraditionalClient.java 和TraditionalServer.java 是基于传统的复制语义的，它们使用了 File.read() 和 Socket.send()。TraditionalServer.java 是一个服务器程序，它在一个特定的端口上监听要连接的客户机，然后以每次 4K 字节的速度从套接字读取数据。TraditionalClient.java 连接到服务器，从文件读取 4K 字节的数据（使用 File.read()），并将内容通过套接字发送到服务器（使用 socket.send()）。

TransferToServer.java 和 TransferToClient.java 执行的功能与此相同，但使用 transferTo() 方法（sendfile() 系统调用）来将文件从服务器传输到客户机。

**性能比较**

我们在一个运行 2.6 内核的 Linux 系统上执行了示例程序，并以毫秒为单位分别度量了使用传统方法和 transferTo() 方法传输不同大小的文件的运行时间。表 1 展示了度量的结果：

**表 1. 性能对比：传统方法与零拷贝**

|  |  |  |
| --- | --- | --- |
| **文件大小** | **正常文件传输（ms）** | **transferTo（ms）** |
| **7MB** | 156 | 45 |
| **21MB** | 337 | 128 |
| **63MB** | 843 | 387 |
| **98MB** | 1320 | 617 |
| **200MB** | 2124 | 1150 |
| **350MB** | 3631 | 1762 |
| **700MB** | 13498 | 4422 |
| **1GB** | 18399 | 8537 |

如您所见，与传统方法相比，transferTo() API 大约减少了 65% 的时间。这就极有可能提高了需要在 I/O 通道间大量拷贝数据的应用程序的性能，如 Web 服务器。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-zerocopy/#ibm-pcon)

**结束语**

我们已经展示了使用 transferTo() 方法较使用传统方法 — 从一个通道读出数据并将其写入到另外一个通道 — 的性能优势。中间缓冲区拷贝 — 甚至于那些隐藏在内核内的拷贝 — 都会产生一定的开销。在需要在通道间大量拷贝数据的应用程序中，零拷贝技巧能够显著地提高性能。