通过零拷贝实现有效数据传输

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。

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，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

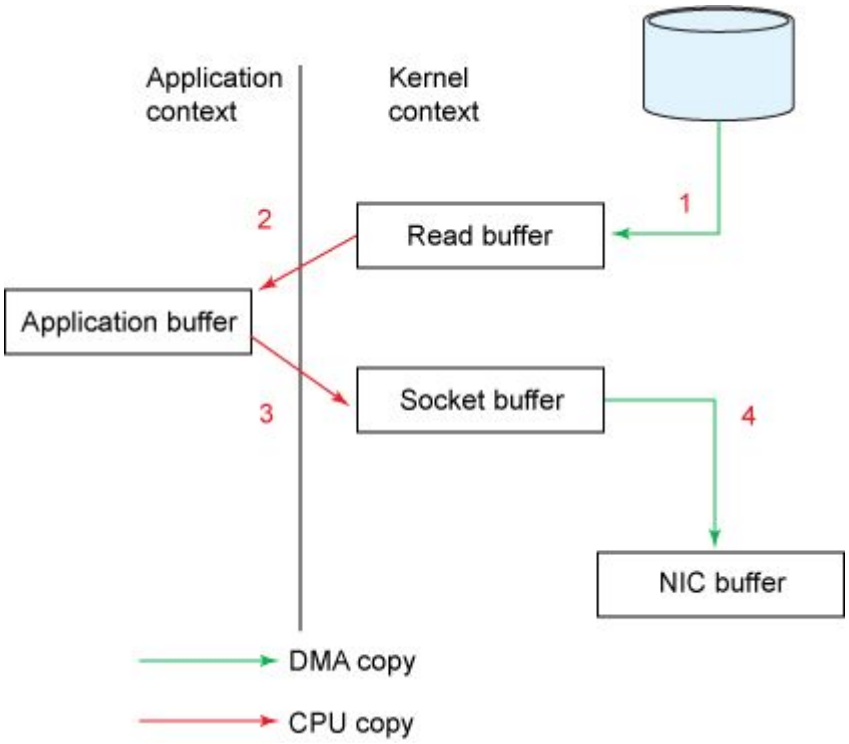
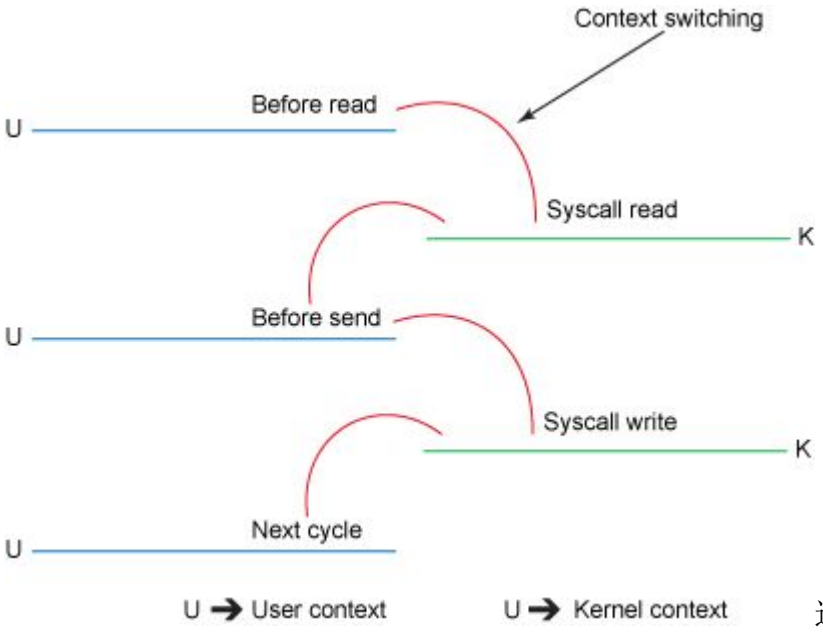


图 2 展示了上下文切换：

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）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），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

数据传输：零拷贝方法

再次检查 传统场景，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 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

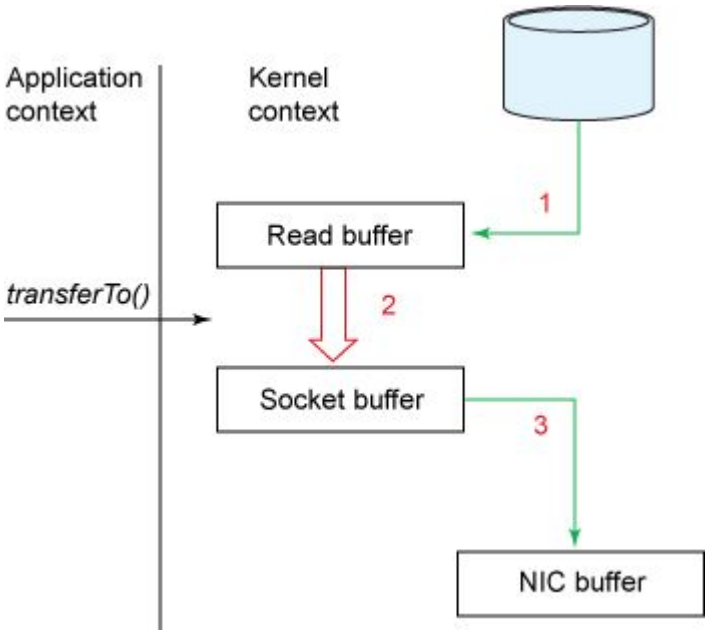
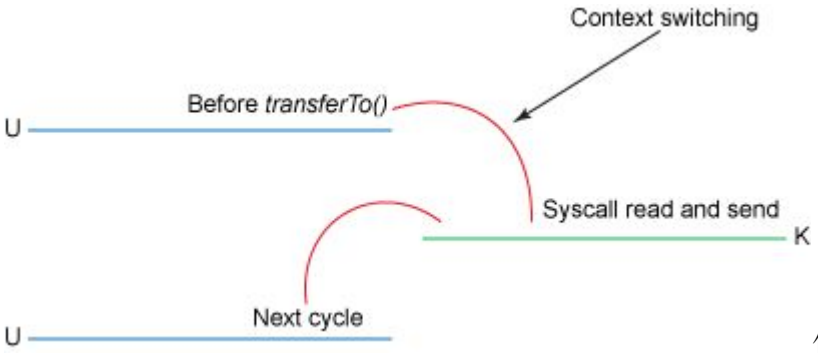


图 4 展示了使用 transferTo() 方法时的上下文切换：

图 4. 使用 transferTo() 方法的上下文切换



使用 清单 4 所示的 transferTo() 方法时的步骤有：

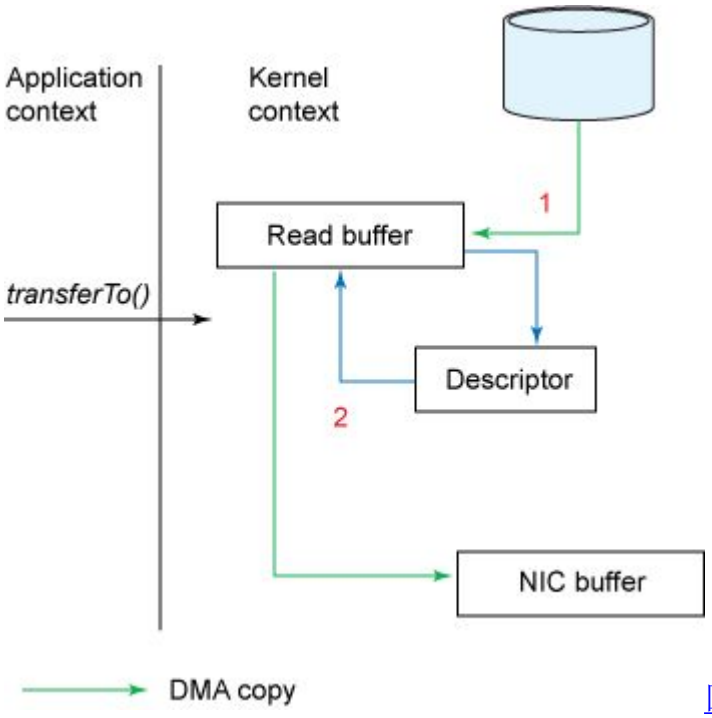
5　transferTo()方法引发 DMA 引擎将文件内容拷贝到一个读取缓冲区。然后由内核将数据拷贝到与输出套接字相关联的内核缓冲区。
6　数据的第三次复制发生在 DMA 引擎将数据从内核套接字缓冲区传到协议引擎时。

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：

7　transferTo()方法引发 DMA 引擎将文件内容拷贝到内核缓冲区。
8　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

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，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

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。