Java NIO - 零拷贝实现

Java NIO零拷贝

在 Java NIO 中的**通道（Channel）*就相当于操作系统的*内核空间**（kernel space）的缓冲区，而**缓冲区**（Buffer）对应的相当于操作系统的**用户空间**（user space）中的**用户缓冲区**（user buffer）。

- **通道**（Channel）是全双工的（双向传输），它既可能是读缓冲区（read buffer），也可能是网络缓冲区（socket buffer）。
- **缓冲区**（Buffer）分为堆内存（HeapBuffer）和堆外内存（DirectBuffer），这是通过 malloc() 分配出来的用户态内存。

堆外内存（DirectBuffer）在使用后需要应用程序手动回收，而堆内存（HeapBuffer）的数据在 GC 时可能会被自动回收。因此，在使用 HeapBuffer 读写数据时，为了避免缓冲区数据因为 GC 而丢失，NIO 会先把 HeapBuffer 内部的数据拷贝到一个临时的 DirectBuffer 中的本地内存（native memory），这个拷贝涉及到 `sun.misc.Unsafe.copyMemory()` 的调用，背后的实现原理与 `memcpy()` 类似。 最后，将临时生成的 DirectBuffer 内部的数据的内存地址传给 I/O 调用函数，这样就避免了再去访问 Java 对象处理 I/O 读写。

MappedByteBuffer

MappedByteBuffer 是 NIO 基于**内存映射（mmap）**这种零拷贝方式的提供的一种实现，它继承自 ByteBuffer。FileChannel 定义了一个 map() 方法，它可以把一个文件从 position 位置开始的 size 大小的区域映射为内存映像文件。抽象方法 map() 方法在 FileChannel 中的定义如下：

```
public abstract MappedByteBuffer map(MapMode mode, long position, long size)
        throws IOException;
```

- **mode**：限定内存映射区域（MappedByteBuffer）对内存映像文件的访问模式，包括只可读（READ_ONLY）、可读可写（READ_WRITE）和写时拷贝（PRIVATE）三种模式。
- **position**：文件映射的起始地址，对应内存映射区域（MappedByteBuffer）的首地址。
- **size**：文件映射的字节长度，从 position 往后的字节数，对应内存映射区域（MappedByteBuffer）的大小。

MappedByteBuffer 相比 ByteBuffer 新增了 fore()、load() 和 isLoad() 三个重要的方法：

- **fore()**：对于处于 READ_WRITE 模式下的缓冲区，把对缓冲区内容的修改强制刷新到本地文件。
- **load()**：将缓冲区的内容载入物理内存中，并返回这个缓冲区的引用。
- **isLoaded()**：如果缓冲区的内容在物理内存中，则返回 true，否则返回 false。

下面给出一个利用 MappedByteBuffer 对文件进行读写的使用示例：

```
private final static String CONTENT = "Zero copy implemented by MappedByteBuffer";
private final static String FILE_NAME = "/mmap.txt";
private final static String CHARSET = "UTF-8";
```

- **写文件数据**：打开文件通道 fileChannel 并提供读权限、写权限和数据清空权限，通过 fileChannel 映射到一个可写的内存缓冲区 mappedByteBuffer，将目标数据写入 mappedByteBuffer，通过 `force()` 方法把缓冲区更改的内容强制写入本地文件。

```
@Test
public void writeToFileByMappedByteBuffer() {
    Path path = Paths.get(getClass().getResource(FILE_NAME).getPath());
    byte[] bytes = CONTENT.getBytes(Charset.forName(CHARSET));
    try (FileChannel fileChannel = FileChannel.open(path, StandardOpenOption.READ,
            StandardOpenOption.WRITE, StandardOpenOption.TRUNCATE_EXISTING)) {
        MappedByteBuffer mappedByteBuffer = fileChannel.map(READ_WRITE, 0, bytes.length);
        if (mappedByteBuffer != null) {
            mappedByteBuffer.put(bytes);
            mappedByteBuffer.force();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

- **读文件数据**：打开文件通道 fileChannel 并提供只读权限，通过 fileChannel 映射到一个只可读的内存缓冲区 mappedByteBuffer，读取 mappedByteBuffer 中的字节数组即可得到文件数据。

```
@Test
public void readFromFileByMappedByteBuffer() {
    Path path = Paths.get(getClass().getResource(FILE_NAME).getPath());
    int length = CONTENT.getBytes(Charset.forName(CHARSET)).length;
    try (FileChannel fileChannel = FileChannel.open(path, StandardOpenOption.READ)) {
        MappedByteBuffer mappedByteBuffer = fileChannel.map(READ_ONLY, 0, length);
        if (mappedByteBuffer != null) {
            byte[] bytes = new byte[length];
            mappedByteBuffer.get(bytes);
            String content = new String(bytes, StandardCharsets.UTF_8);
            assertEquals(content, "Zero copy implemented by MappedByteBuffer");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

下面介绍 `map()` 方法的**底层实现原理**。`map()` 方法是 `java.nio.channels.FileChannel` 的抽象方法，由子类 `sun.nio.ch.FileChannelImpl.java` 实现，下面是和内存映射相关的核心代码：

```
public MappedByteBuffer map(MapMode mode, long position, long size) throws IOException {
    int pagePosition = (int)(position % allocationGranularity);
    long mapPosition = position - pagePosition;
    long mapSize = size + pagePosition;
    try {
        addr = map0(imode, mapPosition, mapSize);
    } catch (OutOfMemoryError x) {
        System.gc();
        try {
            Thread.sleep(100);
        } catch (InterruptedException y) {
            Thread.currentThread().interrupt();
        }
        try {
            addr = map0(imode, mapPosition, mapSize);
        } catch (OutOfMemoryError y) {
```

```
            throw new IOException("Map failed", y);
        }
    }

    int isize = (int)size;
    Unmapper um = new Unmapper(addr, mapSize, isize, mfd);
    if ((!writable) || (imode == MAP_RO)) {
        return Util.newMappedByteBufferR(isize, addr + pagePosition, mfd, um);
    } else {
        return Util.newMappedByteBuffer(isize, addr + pagePosition, mfd, um);
    }
}
```

map() 方法通过本地方法 map0() 为文件分配一块虚拟内存，作为它的内存映射区域，然后返回这块内存映射区域的起始地址。

- 文件映射需要在 Java 堆中创建一个 MappedByteBuffer 的实例。如果第一次文件映射导致 OOM，则手动触发垃圾回收，休眠 100ms 后再尝试映射，如果失败则抛出异常。
- 通过 Util 的 newMappedByteBuffer （可读可写）方法或者 newMappedByteBufferR（仅读） 方法方法反射创建一个 DirectByteBuffer 实例，其中 DirectByteBuffer 是 MappedByteBuffer 的子类。

`map()` 方法返回的是内存映射区域的起始地址，通过（**起始地址** + **偏移量**）就可以获取指定内存的数据。这样一定程度上替代了 `read()` 或 `write()` 方法，底层直接采用 `sun.misc.Unsafe`类的 `getByte()` 和 `putByte()` 方法对数据进行读写。

```
private native long map0(int prot, long position, long mapSize) throws IOException;
```

上面是本地方法（native method）map0 的定义，它通过 JNI（Java Native Interface）调用底层 C 的实现，这个 native 函数（Java_sun_nio_ch_FileChannelImpl_map0）的实现位于 JDK 源码包下的`native/sun/nio/ch/FileChannelImpl.c`这个源文件里面。

```
JNIEXPORT jlong JNICALL
Java_sun_nio_ch_FileChannelImpl_map0(JNIEnv *env, jobject this,
                                     jint prot, jlong off, jlong len)
{
    void *mapAddress = 0;
    jobject fdo = (*env)->GetObjectField(env, this, chan_fd);
    jint fd = fdval(env, fdo);
    int protections = 0;
    int flags = 0;

    if (prot == sun_nio_ch_FileChannelImpl_MAP_RO) {
        protections = PROT_READ;
        flags = MAP_SHARED;
    } else if (prot == sun_nio_ch_FileChannelImpl_MAP_RW) {
        protections = PROT_WRITE | PROT_READ;
        flags = MAP_SHARED;
    } else if (prot == sun_nio_ch_FileChannelImpl_MAP_PV) {
        protections =  PROT_WRITE | PROT_READ;
        flags = MAP_PRIVATE;
    }

    mapAddress = mmap64(
        0,                    /* Let OS decide location */
        len,                  /* Number of bytes to map */
        protections,          /* File permissions */
        flags,                /* Changes are shared */
        fd,                   /* File descriptor of mapped file */
        off);                 /* Offset into file */
```

```
    if (mapAddress == MAP_FAILED) {
        if (errno == ENOMEM) {
            JNU_ThrowOutOfMemoryError(env, "Map failed");
            return IOS_THROWN;
        }
        return handle(env, -1, "Map failed");
    }

    return ((jlong) (unsigned long) mapAddress);
}
```

可以看出 map0() 函数最终是通过 `mmap64()` 这个函数对 Linux 底层内核发出内存映射的调用， `mmap64()` 函数的原型如下：

```
#include <sys/mman.h>

void *mmap64(void *addr, size_t len, int prot, int flags, int fd, off64_t offset);
```

下面详细介绍一下 `mmap64()` 函数各个参数的含义以及参数可选值：

- `addr`：文件在用户进程空间的内存映射区中的起始地址，是一个建议的参数，通常可设置为 0 或 NULL，此时由内核去决定真实的起始地址。当 + flags 为 MAP_FIXED 时，addr 就是一个必选的参数，即需要提供一个存在的地址。
- `len`：文件需要进行内存映射的字节长度
- `prot`：控制用户进程对内存映射区的访问权限
  - `PROT_READ`：读权限
  - `PROT_WRITE`：写权限
  - `PROT_EXEC`：执行权限
  - `PROT_NONE`：无权限
- `flags`：控制内存映射区的修改是否被多个进程共享
  - `MAP_PRIVATE`：对内存映射区数据的修改不会反映到真正的文件，数据修改发生时采用写时复制机制
  - `MAP_SHARED`：对内存映射区的修改会同步到真正的文件，修改对共享此内存映射区的进程是可见的
  - `MAP_FIXED`：不建议使用，这种模式下 addr 参数指定的必须的提供一个存在的 addr 参数
- `fd`：文件描述符。每次 map 操作会导致文件的引用计数加 1，每次 unmap 操作或者结束进程会导致引用计数减 1
- `offset`：文件偏移量。进行映射的文件位置，从文件起始地址向后的位移量

下面总结一下 MappedByteBuffer 的特点和不足之处：

- **MappedByteBuffer 使用是堆外的虚拟内存**，因此分配（map）的内存大小不受 JVM 的 -Xmx 参数限制，但是也是有大小限制的。 如果当文件超出 Integer.MAX_VALUE 字节限制时，可以通过 position 参数重新 map 文件后面的内容。
- **MappedByteBuffer 在处理大文件时性能的确很高，但也存内存占用、文件关闭不确定等问题**，被其打开的文件只有在垃圾回收的才会被关闭，而且这个时间点是不确定的。
- MappedByteBuffer 提供了文件映射内存的 mmap() 方法，也提供了释放映射内存的 unmap() 方法。然而 unmap() 是 FileChannelImpl 中的私有方法，无法直接显示调用。因此，**用户程序需要通过 Java 反射的调用 sun.misc.Cleaner 类的 clean() 方法手动释放映射占用的内存区域。**

```
public static void clean(final Object buffer) throws Exception {
    AccessController.doPrivileged((PrivilegedAction<Void>) () -> {
        try {
            Method getCleanerMethod = buffer.getClass().getMethod("cleaner", new Class[0]);
            getCleanerMethod.setAccessible(true);
            Cleaner cleaner = (Cleaner) getCleanerMethod.invoke(buffer, new Object[0]);
            cleaner.clean();
        } catch(Exception e) {
            e.printStackTrace();
        }
    });
}
```

DirectByteBuffer

DirectByteBuffer 的对象引用位于 Java 内存模型的堆里面，JVM 可以对 DirectByteBuffer 的对象进行内存分配和回收管理，一般使用 DirectByteBuffer 的静态方法 allocateDirect() 创建 DirectByteBuffer 实例并分配内存。

```
public static ByteBuffer allocateDirect(int capacity) {
    return new DirectByteBuffer(capacity);
}
```

DirectByteBuffer 内部的字节缓冲区位在于堆外的（用户态）直接内存，它是通过 Unsafe 的本地方法 allocateMemory() 进行内存分配，底层调用的是操作系统的 malloc() 函数。

```
DirectByteBuffer(int cap) {
    super(-1, 0, cap, cap);
    boolean pa = VM.isDirectMemoryPageAligned();
    int ps = Bits.pageSize();
    long size = Math.max(1L, (long)cap + (pa ? ps : 0));
    Bits.reserveMemory(size, cap);

    long base = 0;
    try {
        base = unsafe.allocateMemory(size);
    } catch (OutOfMemoryError x) {
        Bits.unreserveMemory(size, cap);
        throw x;
    }
    unsafe.setMemory(base, size, (byte) 0);
    if (pa && (base % ps != 0)) {
        address = base + ps - (base & (ps - 1));
    } else {
        address = base;
    }
    cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
    att = null;
}
```

除此之外，初始化 DirectByteBuffer 时还会创建一个 Deallocator 线程，并通过 Cleaner 的 freeMemory() 方法来对直接内存进行回收操作，freeMemory() 底层调用的是操作系统的 free() 函数。

```
private static class Deallocator implements Runnable {
    private static Unsafe unsafe = Unsafe.getUnsafe();
```

```
    private long address;
    private long size;
    private int capacity;

    private Deallocator(long address, long size, int capacity) {
        assert (address != 0);
        this.address = address;
        this.size = size;
        this.capacity = capacity;
    }

    public void run() {
        if (address == 0) {
            return;
        }
        unsafe.freeMemory(address);
        address = 0;
        Bits.unreserveMemory(size, capacity);
    }
}
```

由于使用 DirectByteBuffer 分配的是系统本地的内存，不在 JVM 的管控范围之内，因此直接内存的回收和堆内存的回收不同，直接内存如果使用不当，很容易造成 OutOfMemoryError。

说了这么多，那么 DirectByteBuffer 和零拷贝有什么关系？前面有提到在 MappedByteBuffer 进行内存映射时，它的 map() 方法会通过 Util.newMappedByteBuffer() 来创建一个缓冲区实例，初始化的代码如下：

```
static MappedByteBuffer newMappedByteBuffer(int size, long addr, FileDescriptor fd,
                                            Runnable unmapper) {
    MappedByteBuffer dbb;
    if (directByteBufferConstructor == null)
        initDBBConstructor();
    try {
        dbb = (MappedByteBuffer)directByteBufferConstructor.newInstance(
            new Object[] { new Integer(size), new Long(addr), fd, unmapper });
    } catch (InstantiationException | IllegalAccessException | InvocationTargetException e) {
        throw new InternalError(e);
    }
    return dbb;
}

private static void initDBBRConstructor() {
    AccessController.doPrivileged(new PrivilegedAction<Void>() {
        public Void run() {
            try {
                Class<?> cl = Class.forName("java.nio.DirectByteBufferR");
                Constructor<?> ctor = cl.getDeclaredConstructor(
                    new Class<?>[] { int.class, long.class, FileDescriptor.class,
                                     Runnable.class });
                ctor.setAccessible(true);
                directByteBufferRConstructor = ctor;
            } catch (ClassNotFoundException | NoSuchMethodException |
                     IllegalArgumentException | ClassCastException x) {
                throw new InternalError(x);
            }
            return null;
        }});
}
```

DirectByteBuffer 是 MappedByteBuffer 的具体实现类。实际上，Util.newMappedByteBuffer() 方法通过反射机制获取 DirectByteBuffer 的构造器，然后创建一个 DirectByteBuffer 的实例，对应的是一个单独用于内存映射的构造方法：

```
protected DirectByteBuffer(int cap, long addr, FileDescriptor fd, Runnable unmapper) {
    super(-1, 0, cap, cap, fd);
    address = addr;
    cleaner = Cleaner.create(this, unmapper);
    att = null;
}
```

因此，除了允许分配操作系统的直接内存以外，DirectByteBuffer 本身也具有文件内存映射的功能，这里不做过多说明。我们需要关注的是，DirectByteBuffer 在 MappedByteBuffer 的基础上提供了内存映像文件的随机读取 get() 和写入 write() 的操作。

- 内存映像文件的随机读操作

```
public byte get() {
    return ((unsafe.getByte(ix(nextGetIndex()))));
}

public byte get(int i) {
    return ((unsafe.getByte(ix(checkIndex(i)))));
}
```

- 内存映像文件的随机写操作

```
public ByteBuffer put(byte x) {
    unsafe.putByte(ix(nextPutIndex()), ((x)));
    return this;
}

public ByteBuffer put(int i, byte x) {
    unsafe.putByte(ix(checkIndex(i)), ((x)));
    return this;
}
```

内存映像文件的随机读写都是借助 ix() 方法实现定位的， ix() 方法通过内存映射空间的内存首地址（address）和给定偏移量 i 计算出指针地址，然后由 unsafe 类的 get() 和 put() 方法和对指针指向的数据进行读取或写入。

```
private long ix(int i) {
    return address + ((long)i << 0);
}
```

FileChannel

FileChannel 是一个用于文件读写、映射和操作的通道，同时它在并发环境下是线程安全的，基于 FileInputStream、FileOutputStream 或者 RandomAccessFile 的 getChannel() 方法可以创建并打开一个文件通道。FileChannel 定义了 transferFrom() 和 transferTo() 两个抽象方法，它通过在通道和通道之间建立连接实现数据传输的。

- `transferTo()`：通过 FileChannel 把文件里面的源数据写入一个 WritableByteChannel 的目的通道。

```
public abstract long transferTo(long position, long count, WritableByteChannel target)
        throws IOException;
```

- `transferFrom()`：把一个源通道 ReadableByteChannel 中的数据读取到当前 FileChannel 的文件里面。

```
public abstract long transferFrom(ReadableByteChannel src, long position, long count)
        throws IOException;
```

下面给出 FileChannel 利用 transferTo() 和 transferFrom() 方法进行数据传输的使用示例：

```
private static final String CONTENT = "Zero copy implemented by FileChannel";
private static final String SOURCE_FILE = "/source.txt";
private static final String TARGET_FILE = "/target.txt";
private static final String CHARSET = "UTF-8";
```

首先在类加载根路径下创建 source.txt 和 target.txt 两个文件，对源文件 source.txt 文件写入初始化数据。

```
@Before
public void setup() {
    Path source = Paths.get(getClassPath(SOURCE_FILE));
    byte[] bytes = CONTENT.getBytes(Charset.forName(CHARSET));
    try (FileChannel fromChannel = FileChannel.open(source, StandardOpenOption.READ,
            StandardOpenOption.WRITE, StandardOpenOption.TRUNCATE_EXISTING)) {
        fromChannel.write(ByteBuffer.wrap(bytes));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

对于 transferTo() 方法而言，目的通道 toChannel 可以是任意的单向字节写通道 WritableByteChannel；而对于 transferFrom() 方法而言，源通道 fromChannel 可以是任意的单向字节读通道 ReadableByteChannel。其中，FileChannel、SocketChannel 和 DatagramChannel 等通道实现了 WritableByteChannel 和 ReadableByteChannel 接口，都是同时支持读写的双向通道。为了方便测试，下面给出基于 FileChannel 完成 channel-to-channel 的数据传输示例。

通过 transferTo() 将 fromChannel 中的数据拷贝到 toChannel

```
@Test
public void transferTo() throws Exception {
    try (FileChannel fromChannel = new RandomAccessFile(
            getClassPath(SOURCE_FILE), "rw").getChannel();
         FileChannel toChannel = new RandomAccessFile(
            getClassPath(TARGET_FILE), "rw").getChannel()) {
        long position = 0L;
        long offset = fromChannel.size();
        fromChannel.transferTo(position, offset, toChannel);
    }
}
```

通过 transferFrom() 将 fromChannel 中的数据拷贝到 toChannel

```
@Test
public void transferFrom() throws Exception {
    try (FileChannel fromChannel = new RandomAccessFile(
            getClassPath(SOURCE_FILE), "rw").getChannel();
         FileChannel toChannel = new RandomAccessFile(
            getClassPath(TARGET_FILE), "rw").getChannel()) {
        long position = 0L;
        long offset = fromChannel.size();
        toChannel.transferFrom(fromChannel, position, offset);
    }
}
```

下面介绍 transferTo() 和 transferFrom() 方法的底层实现原理，这两个方法也是 java.nio.channels.FileChannel 的抽象方法，由子类 sun.nio.ch.FileChannelImpl.java 实现。transferTo() 和 transferFrom() 底层都是基于 sendfile 实现数据传输的，其中 FileChannelImpl.java 定义了 3 个常量，用于标示当前操作系统的内核是否支持 sendfile 以及 sendfile 的相关特性。

```
private static volatile boolean transferSupported = true;
private static volatile boolean pipeSupported = true;
private static volatile boolean fileSupported = true;
```

- `transferSupported`：用于标记当前的系统内核是否支持 sendfile() 调用，默认为 true。
- `pipeSupported`：用于标记当前的系统内核是否支持文件描述符（fd）基于管道（pipe）的 sendfile() 调用，默认为 true。
- `fileSupported`：用于标记当前的系统内核是否支持文件描述符（fd）基于文件（file）的 sendfile() 调用，默认为 true。

下面以 transferTo() 的源码实现为例。FileChannelImpl 首先执行 transferToDirectly() 方法，以 sendfile 的零拷贝方式尝试数据拷贝。如果系统内核不支持 sendfile，进一步执行 transferToTrustedChannel() 方法，以 mmap 的零拷贝方式进行内存映射，这种情况下目的通道必须是 FileChannelImpl 或者 SelChImpl 类型。如果以上两步都失败了，则执行 transferToArbitraryChannel() 方法，基于传统的 I/O 方式完成读写，具体步骤是初始化一个临时的 DirectBuffer，将源通道 FileChannel 的数据读取到 DirectBuffer，再写入目的通道 WritableByteChannel 里面

```
public long transferTo(long position, long count, WritableByteChannel target)
        throws IOException {
    // 计算文件的大小
    long sz = size();
    // 校验起始位置
    if (position > sz)
        return 0;
    int icount = (int)Math.min(count, Integer.MAX_VALUE);
    // 校验偏移量
    if ((sz - position) < icount)
        icount = (int)(sz - position);

    long n;

    if ((n = transferToDirectly(position, icount, target)) >= 0)
        return n;

    if ((n = transferToTrustedChannel(position, icount, target)) >= 0)
        return n;

    return transferToArbitraryChannel(position, icount, target);
}
```

接下来重点分析一下 transferToDirectly() 方法的实现，也就是 transferTo() 通过 sendfile 实现零拷贝的精髓所在。可以看到，transferToDirectlyInternal() 方法先获取到目的通道 WritableByteChannel 的文件描述符 targetFD，获取同步锁然后执行 transferToDirectlyInternal() 方法。

```
private long transferToDirectly(long position, int icount, WritableByteChannel target)
        throws IOException {
    // 省略从target获取targetFD的过程
    if (nd.transferToDirectlyNeedsPositionLock()) {
        synchronized (positionLock) {
            long pos = position();
            try {
                return transferToDirectlyInternal(position, icount,
                        target, targetFD);
            } finally {
                position(pos);
            }
        }
    } else {
        return transferToDirectlyInternal(position, icount, target, targetFD);
    }
}
```

最终由 transferToDirectlyInternal() 调用本地方法 transferTo0()，尝试以 sendfile 的方式进行数据传输。如果系统内核完全不支持 sendfile，比如 Windows 操作系统，则返回 UNSUPPORTED 并把 transferSupported 标识为 false。如果系统内核不支持 sendfile 的一些特性，比如说低版本的 Linux 内核不支持 DMA gather copy 操作，则返回 UNSUPPORTED_CASE 并把 pipeSupported 或者 fileSupported 标识为 false。

```
private long transferToDirectlyInternal(long position, int icount,
                                        WritableByteChannel target,
                                        FileDescriptor targetFD) throws IOException {
    assert !nd.transferToDirectlyNeedsPositionLock() ||
            Thread.holdsLock(positionLock);

    long n = -1;
    int ti = -1;
    try {
        begin();
        ti = threads.add();
        if (!isOpen())
            return -1;
        do {
            n = transferTo0(fd, position, icount, targetFD);
        } while ((n == IOStatus.INTERRUPTED) && isOpen());
        if (n == IOStatus.UNSUPPORTED_CASE) {
            if (target instanceof SinkChannelImpl)
                pipeSupported = false;
            if (target instanceof FileChannelImpl)
                fileSupported = false;
            return IOStatus.UNSUPPORTED_CASE;
        }
        if (n == IOStatus.UNSUPPORTED) {
            transferSupported = false;
            return IOStatus.UNSUPPORTED;
        }
        return IOStatus.normalize(n);
    } finally {
        threads.remove(ti);
        end (n > -1);
    }
```

```
}
```

本地方法（native method）transferTo0() 通过 JNI（Java Native Interface）调用底层 C 的函数，这个 native 函数（Java_sun_nio_ch_FileChannelImpl_transferTo0）同样位于 JDK 源码包下的 native/sun/nio/ch/FileChannelImpl.c 源文件里面。JNI 函数 Java_sun_nio_ch_FileChannelImpl_transferTo0() 基于条件编译对不同的系统进行预编译，下面是 JDK 基于 Linux 系统内核对 transferTo() 提供的调用封装。

```
#if defined(__linux__) || defined(__solaris__)
#include <sys/sendfile.h>
#elif defined(_AIX)
#include <sys/socket.h>
#elif defined(_ALLBSD_SOURCE)
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>

#define lseek64 lseek
#define mmap64 mmap
#endif

JNIEXPORT jlong JNICALL
Java_sun_nio_ch_FileChannelImpl_transferTo0(JNIEnv *env, jobject this,
                                            jobject srcFDO,
                                            jlong position, jlong count,
                                            jobject dstFDO)
{
    jint srcFD = fdval(env, srcFDO);
    jint dstFD = fdval(env, dstFDO);

#if defined(__linux__)
    off64_t offset = (off64_t)position;
    jlong n = sendfile64(dstFD, srcFD, &offset, (size_t)count);
    return n;
#elif defined(__solaris__)
    result = sendfilev64(dstFD, &sfv, 1, &numBytes);
    return result;
#elif defined(__APPLE__)
    result = sendfile(srcFD, dstFD, position, &numBytes, NULL, 0);
    return result;
#endif
}
```

对 Linux、Solaris 以及 Apple 系统而言，transferTo0() 函数底层会执行 sendfile64 这个系统调用完成零拷贝操作，sendfile64() 函数的原型如下：

```
#include <sys/sendfile.h>

ssize_t sendfile64(int out_fd, int in_fd, off_t *offset, size_t count);
```

下面简单介绍一下 sendfile64() 函数各个参数的含义：

- `out_fd`：待写入的文件描述符
- `in_fd`：待读取的文件描述符
- `offset`：指定 in_fd 对应文件流的读取位置，如果为空，则默认从起始位置开始
- `count`：指定在文件描述符 in_fd 和 out_fd 之间传输的字节数

在 Linux 2.6.3 之前，out_fd 必须是一个 socket，而从 Linux 2.6.3 以后，out_fd 可以是任何文件。也就是说，sendfile64() 函数不仅可以进行网络文件传输，还可以对本地文件实现零拷贝操作。

其它的零拷贝实现

*Netty*零拷贝

Netty 中的零拷贝和上面提到的操作系统层面上的零拷贝不太一样, 我们所说的 Netty 零拷贝完全是基于（Java 层面）用户态的，它的更多的是偏向于数据操作优化这样的概念，具体表现在以下几个方面：

Netty 通过 DefaultFileRegion 类对 java.nio.channels.FileChannel 的 tranferTo() 方法进行包装，在文件传输时可以将文件缓冲区的数据直接发送到目的通道（Channel）

ByteBuf 可以通过 wrap 操作把字节数组、ByteBuf、ByteBuffer 包装成一个 ByteBuf 对象, 进而避免了拷贝操作 ByteBuf 支持 slice 操作, 因此可以将 ByteBuf 分解为多个共享同一个存储区域的 ByteBuf，避免了内存的拷贝 Netty 提供了 CompositeByteBuf 类，它可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf，避免了各个 ByteBuf 之间的拷贝 其中第 1 条属于操作系统层面的零拷贝操作，后面 3 条只能算用户层面的数据操作优化。

*RocketMQ*和*Kafka*对比

RocketMQ 选择了 mmap + write 这种零拷贝方式，适用于业务级消息这种小块文件的数据持久化和传输；而 Kafka 采用的是 sendfile 这种零拷贝方式，适用于系统日志消息这种高吞吐量的大块文件的数据持久化和传输。但是值得注意的一点是，Kafka 的索引文件使用的是 mmap + write 方式，数据文件使用的是 sendfile 方式。

消息队列	零拷贝方式	优点	缺点
RocketMQ	mmap + write	适用于小块文件传输，频繁调用时，效率很高	不能很好的利用 DMA 方式，会比 sendfile 多消耗 CPU，内存安全性控制复杂，需要避免 JVM Crash 问题
Kafka	sendfile	可以利用 DMA 方式，消耗 CPU 较少，大块文件传输效率高，无内存安全性问题	小块文件效率低于 mmap 方式，只能是 BIO 方式传输，不能使用 NIO 方式