NIO 知识点总结

作者：xjlnjut730
原文链接：https://ld246.com/article/1576681653756
来源网站：链滴

1. BIO vs NIO

说起NIO，不可避免的需要拿来与BIO进行对比。在对比之前，我们先理解一下BIO的API。

BIO是Blocking IO的缩写。什么是Blocking IO呢？这里其实就是指传统的基于流的IO编程方式，下就是一个传统的BIO编程方式，代码展示的是如何从BioTest_in.txt读取数据，并将内容复制到BioTest out.txt：

```
public static void main(String[] args) throws Exception {
    FileInputStream fis = new FileInputStream("BioTest_in.txt");
    FileOutputStream fos = new FileOutputStream("BioTest_out.txt");

    byte[] bytes = new byte[512];

    int count;
    while ((count = fis.read(bytes)) > 0) {
        fos.write(bytes, 0, count);
    }

    fos.close();
    fis.close();
}
```

我们再来看看，用NIO如何完成一个类似的工作：

```
public static void main(String[] args) throws Exception {
    FileInputStream fis = new FileInputStream("input2.txt");
    FileOutputStream fos = new FileOutputStream("output2.txt");
    FileChannel inputChannel = fis.getChannel();
    FileChannel outputChannel = fos.getChannel();

    ByteBuffer buffer = ByteBuffer.allocate(20);

    while (true) {
        buffer.clear();

        int read = inputChannel.read(buffer);
        if(read == -1) {
            break;
        }
        buffer.flip();

        outputChannel.write(buffer);
    }

    inputChannel.close();
    outputChannel.close();
}
```

看起来NIO变得更复杂了，这还是没有引入Selector的情况下的代码，引入Selector复杂度会更高。

总体来说，BIO与NIO有以下区别：

1. BIO是基于Stream的方式，而NIO是基于Buffer的方式;

2. BIO是阻塞的，NIO是非阻塞的。

3. NIO增加了Selectors的机制。

1.1 基于Stream vs 基于Buffer

对于基于Stream的方式，可以同时从一个流中读取一个或多个字节。读取多少字节，完全取决于程，默认情况下也不会进行缓存。在这种情况下，你没有办法回溯流中之前读到的位置，除非把流中读的数据进行缓存。

而基于Buffer的方式，则不一样。数据在处理之前，会先读取到Buffer中。你可以在Buffer中读取任位置的数据，而不仅限于从前往后。不过，你还是需要确认缓存中是否已经包含了你所需要的所有数，也需要考虑当你还没处理完缓存中的数据之前，千万不要覆盖掉缓存中的数据。

看起来Stream也能通过使用BufferedInputStream等包装类，达到在缓存中进行多次多个位置的读的目的。但是在基于流的方式下，一个流要么是输入流，要么是输出流，不可能同时既是输入流又是出流。而基于Buffer，一个Buffer既可以读，也可以写，如下所示：

```
public static void main(String[] args) throws Exception {
    RandomAccessFile randomAccessFile = new RandomAccessFile("NioTest14.txt", "rw");

    FileChannel channel = randomAccessFile.getChannel();

    MappedByteBuffer mappedByteBuffer = channel.map(FileChannel.MapMode.READ_WRITE,
0, 5);
    // 读取
    System.out.println(mappedByteBuffer.get(4));
    mappedByteBuffer.clear();
    // 在同一个Buffer中写入
    mappedByteBuffer.put(0, (byte)'a');
    mappedByteBuffer.put(3, (byte)'b');

    channel.write(mappedByteBuffer);
    randomAccessFile.close();
}
```

1.2 阻塞 vs 非阻塞

在传统的基于流的IO都是阻塞的。这就意味着，当一个线程调用read()或write()方法时，线程会阻塞，直到数据读/写完成。在这个读/写过程中，该线程不能做其它事情。

NIO的非阻塞模型，使一个线程在请求从channel中读取数据时，可以在channel中的数据准备好才行读取操作，而不是阻塞住直到数据可读。这样在channel中的数据准备好之前，线程可以做其它事。写操作也是类似的，在channel可写之前，线程可以去做其它事情，直到channel准备好。

非阻塞的方式带来了一个显著的优势：将线程与单个IO读写的绑定关系解放出来。这意味着，如果单channel中的IO读写操作还没准备好，线程完全可以去处理其它channel的IO读写。换句说，单个线具备了同时管理多个IO操作的能力。

1.3 Selectors

NIO中的Selectors允许单个线程去监控多个channel。你可以将多个channel注册到一个selector上然后使用一个线程去"select"准备好读取的channel，或者去"select"准备好写入的channel。这个sele

tor机制使单个线程管理多个channel变得容易了许多。示例代码如下：

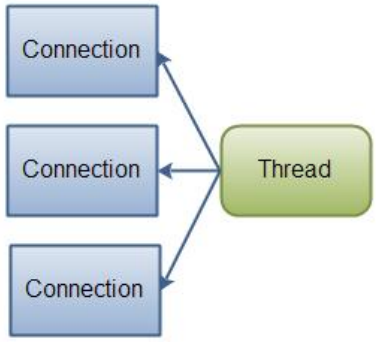
```
// 注意：为了排版，省略了大量try catch与具体处理逻辑
public static void main(String[] args) throws Exception {
    ServerSocketChannel serverSocketChannel = ...;
    Selector selector = Selector.open();
    // 注册socket连接至selector上，注册的channel为serverSocketChannel
    serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

    while (true) {
        // 选择感兴越的事件（目前是OP_ACCEPT），如果存在，就往下执行，不存在，这里就阻塞住
也有不阻塞的API：selectNow()）。
        selector.select();
        // 获得目前可以获取到的selectionKey列表
        Set<SelectionKey> selectionKeys = selector.selectedKeys();

        selectionKeys.forEach(selectionKey -> {
            // 当前selectionKey是否可连接
            if(selectionKey.isAcceptable()) {
                ServerSocketChannel server = (ServerSocketChannel)selectionKey.channel();
                SocketChannel client = server.accept();
                client.configureBlocking(false);
                // ...
                // 注册OP_READ到slector上，注册的channel为client
                client.register(selector, SelectionKey.OP_READ);

            // 当前selectionKey是否可读
            } else if(selectionKey.isReadable()) {
                SocketChannel client = (SocketChannel)selectionKey.channel();
                ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
                int count = client.read(byteBuffer);
                // ...
            }
        });
        // 清空selectionKeys，表示已经处理。不清理会重复执行事件
        selectionKeys.clear();
    }
}
```

1.4 如何选择

NIO允许使用单个或者少量的线程去管理多个channel（网络请求/文件），代价是相对基于流的方式解析数据变得更加复杂，正如上面的代码所示。

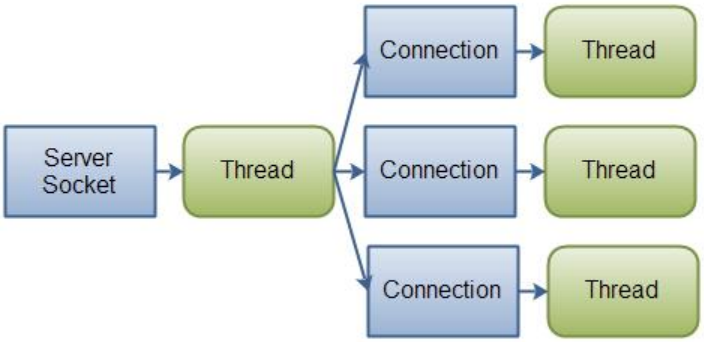
对于单个socket连接来说，不考虑zero-copy的因素，引入NIO实际会增加请求延时，降低吞吐量。以如果是写一个客户端程序，完全可以用BIO来降低编程的复杂度。

但如果你需要管理的是成千上万的连接，每个连接只有少量的数据。最典型的就是类似于聊天室这种用，NIO就具有非常明显的优势。



Java NIO:单个线程管理大量的连接

如果你只有少量的连接，而且每个连接包含大量的网络IO，也许一个典型的BIO就是最佳的选择。



Java IO: 一个连接由一个线程管理

2 Channel

一个channel代表与一个实体的连接，实体可以是硬件设备、文件、网络套接字或编程组件，这些实都支持一个或多个的IO操作，比如读和写。

一个channel要么是开启的要么是关闭的。一个channel一旦创建就是开启状态，一旦关闭就会保持闭。如果一个channel是关闭状态，试图对该channel进行IO操作都会抛出ClosedChannelException判断一个channel是否开启可以通过isOpen方法。

channel是否为多线程安全取决于具体的实现。

3 Selector

Selector是什么？它是一个SelectableChannel对象的多路调节器(multiplexor)。

一个selector可以通过Selector类本身的open创建得到，这种创建是使用了系统默认的select provide来创建的。一个selector也可以通过调用一个自定义select provider的openSelector方法得到。一个s lector创建后会一直开启，直到调用close方法。

selector与channel的注册关系表现为SelectionKey对象。一个selector维持三个selection key的集：

1. key set, 包含当前channel注册在selector上的keys。这个集合可以通过keys方法得到。
2. selected-key set, 这个集合里包含了channel感兴趣的的key。这个集合通过selectedKeys方法得

。selected-key set是key set的一个子集。

3. cancelled-key set 是保存那些已经被取消，但还没有从key set中移除的key.这个集合不会被直接问到，cancelled-key set是key set的一个子集。

刚创建selector时，这三个集合都是空的。

channel调用register方法时，一个对应的key就会加入到selector的key set中。cancelled-key set会下一次selection操作时被移除。key set 本身是不能被直接修改的。

当key的channel本身关闭，或者调用key的cancel方法，都会导致这个key被加到cancelled-key中。cncelled-key set会在下一次selection操作时被移除，此时，这些key会从selector的所有key set中被除。

当执行selection操作时，key将会被加到selected-key中。通过调用set的remove方法，或者通过set iterator的remove方法，一个key可以直接从selected-key中移除。除此之外，selected-key没有其的移除方式。它并不会随着selection操作移除。

3.1 Selection

在每一个selection操作中，key可以被添加到selector的selected-key set中，也可以从中被移除。selction操作具体是指select(), select(long)和selectNow()方法，涉及以下三个步骤：

1. 先把canceled-key中的key从所有集合中移除，并且移除对应channel的注册信息。这个步骤执行，canceled-key就会变空。

2. 在selection操作开始时，JVM会去查询底层操作系统，以确认每个channel是否准备好由其key对的兴趣集合标记的任何IO操作。对于一个channel而言，如果准备好对应的操作，接下来两个步骤会执行：

    1. 如果channel的key并没有在selected-key中，则会把key加入selected-key中，同时也会加入ready-operation set，这个集合用来标记这个channel可以进行的具体操作。所有之前记录的ready set被清除。

    2. 如果channel的key已经在selected-key中，则会把key加入ready-operation set，所有之前记的ready set都会被保留。换句话说，从底层操作系统返回的readdy set是按位或(bitwise-disjoined)。

所有在key set中的key，如果开始时没有interest set。那么要么selected-key被更新，要么key对应的eady-operation set会被更新。

3. 如果在执行第2步时，有key加入了cancelled-key　set，那么它们会继续在第1步中处理。

selection操作在等待一个或多个channel时是否会保持阻塞，取决于执行的具体方法，是select()，selct(long)还是selectNow()。

4 Buffer

Buffer本身就是一块内存，底层实现上，它实际上是个数组。数据的读、写都是通过Buffer来实现的除了数组之外，Buffer还提供了对于数据的结构化访问方式，并且可以追踪到系统的读写过程。Java的7种原生数据类型都有各自对应的Buffer类型，如IntBuffer，LongBuffer，CharBuffer等,没有BoolanBuffer。

所有数据的读写都是通过Buffer来进行的，永远不会出现直接向Channel写入数据的情况，或者直接Channel读取数据的情况。

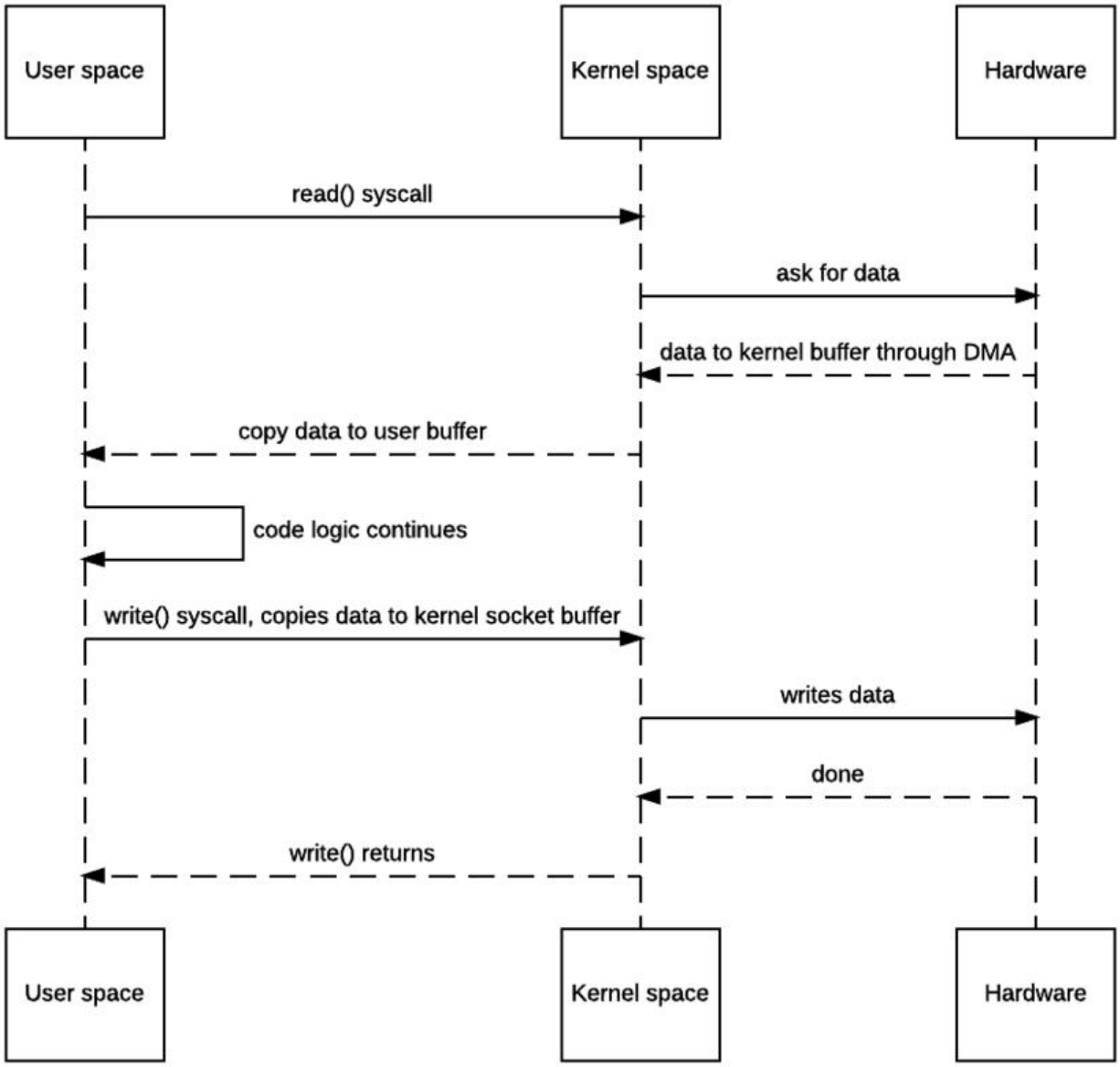
4.1 IO底层实现

假如应用中低延时是非常重要的指标，那么我们就有必要从操作系统层面了解下IO的底层实现，我们看一下本文中的第一个BIO的例子：

```
public static void main(String[] args) throws Exception {
    FileInputStream fis = new FileInputStream("BioTest_in.txt");
    FileOutputStream fos = new FileOutputStream("BioTest_out.txt");

    byte[] bytes = new byte[512];

    int count;
    while ((count = fis.read(bytes)) > 0) {
        fos.write(bytes, 0, count);
    }

    fos.close();
    fis.close();
}
```

4.1.1 read() & write()

在操作系统层面，发生了如下事情：



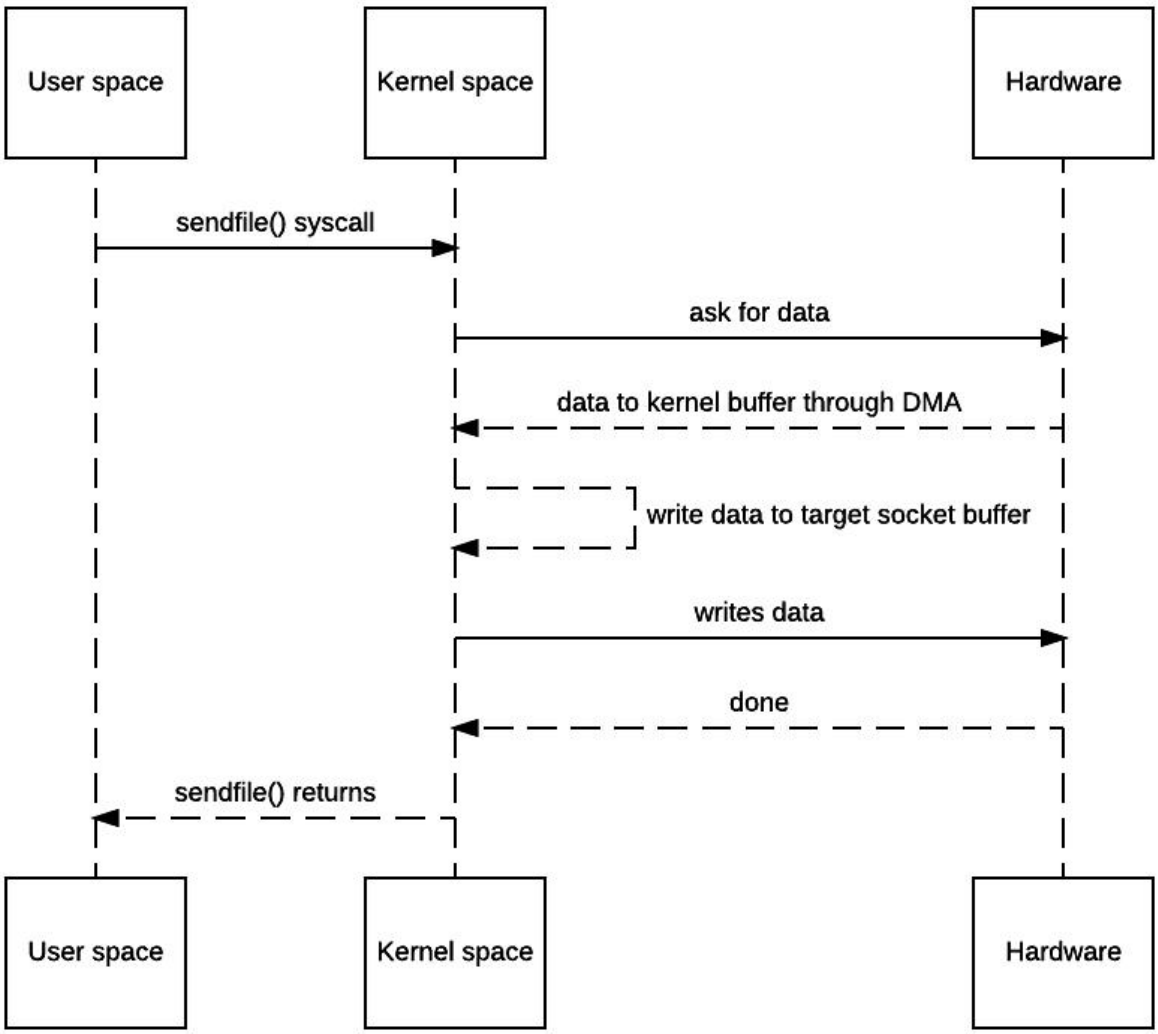
1. JVM发送read()的系统调用。
2. 操作系统将上下文切换到内核态，读取硬盘上的数据到input socket buffer。
3. 操作系统将数据复制至用户态，并将上下文切换回用户态，此时read()方法返回。
4. JVM执行处理逻辑，并发送write()的系统调用。
5. 操作系统将上下文切换至内核态，并将数据从用户态复制到output socket buffer。
6. 操作系统返回至用户态，JVM继续执行后续的逻辑。

在延时与吞吐量还没有成为我们服务的瓶颈时，上面的代码可以很好的工作。但是如果仔细思考的话对于单个用户来说，性能仍然不够好。因为上面的例子中有4次上下文的切换，和2次没有意义的拷贝。

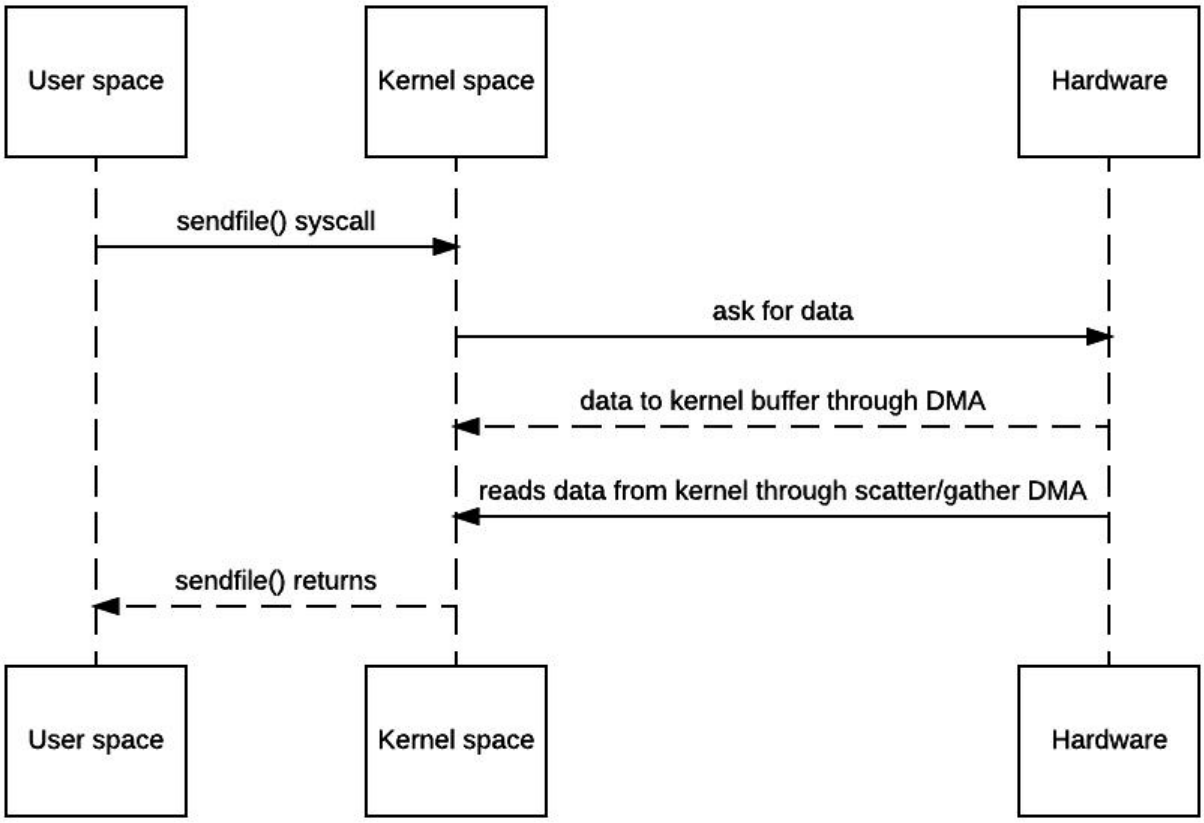
注意到上面这个例子中，从内核态复制内容到用户态，以及从用户态复制回内核态是完全没有必要的因为我们除了复制文件，没有对内容做任何修改。在这种情况下，zero copy完全可以被使用。zero copy的具体实现没有一个标准，取决于操作系统是如何实现的。在Linux系统中提供了sendfile()。

4.1.2 sendfile()

使用sendfile()，流程图就变成这样：



这里操作系统仍然存在一个内核态的拷贝。从操作系统的角度，这里已经zero-copy了，因为没有数从内核态拷贝至用户态。为什么还在内核态中再拷贝一次呢？这是因为，通常硬件DMA访问需要连的内存空间(和缓冲区）。如果硬件支持scatter-n-gather，内核态的拷贝就是可以避免的，此时流程变成这样：

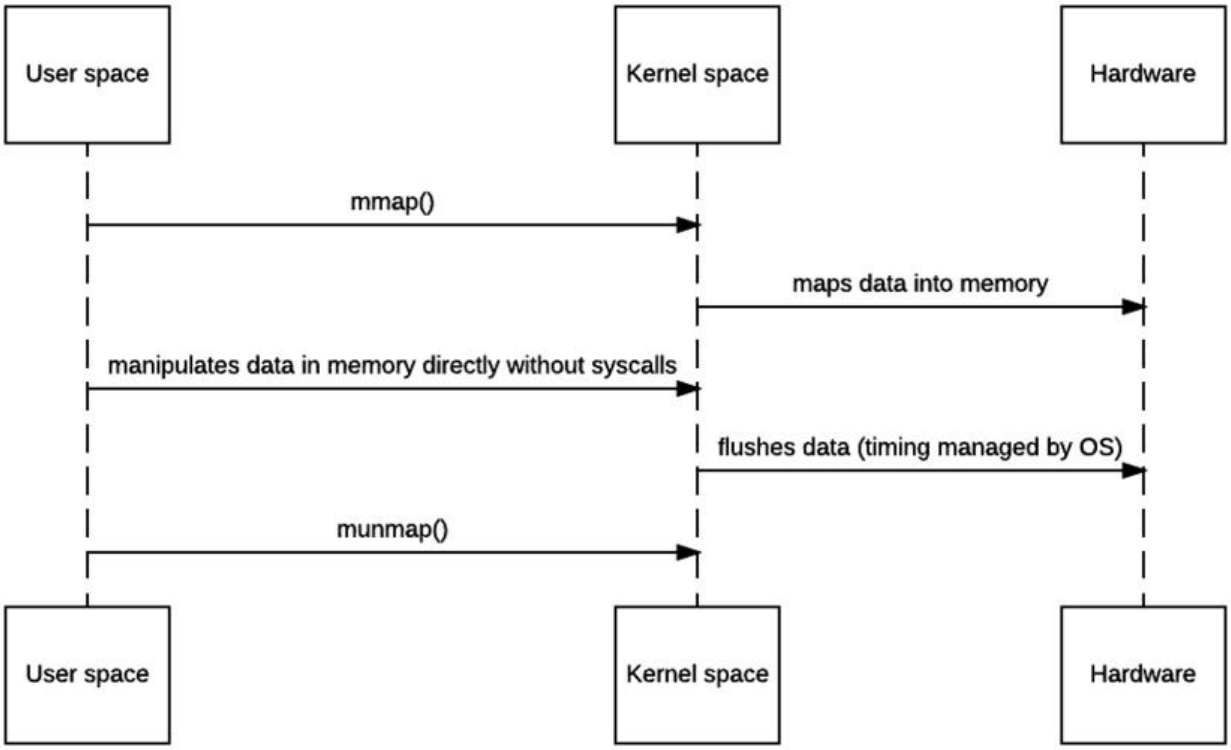


很多web服务器支持zero-copy，比如Tomcat和Apache，需要通过参数启用。

在Java的NIO中，提供了transferTo方法对zero-copy提供了支持。

4.1.3 mmap()

上面zero-copy的解决方案存在一个限制，就是程序中无法操作内容，只能将其直接转发。这里有一开销更大，但是更有用的方式，就是mmap，short for memory-map。



mmap允许程序将文件映射至内核态，而且可以通过用户态中直接访问，避免了不必要的复制。有得有失，这里仍然包含了4次上下文切换。但是一旦操作系统将文件映射至内存，就可以获得操作系统拟内存管理的所有好处：热点内容可以更有效率的缓存，所有数据都是页对齐，因此对于写操作，不要从缓冲区中进行复制。

然而，天下没有免费的午餐。mmap虽然避免了额外的复制，但是它并不保证代码会更有效率，这取于操作系统的实现，也许会增加启动和销毁时的开销(因为需要找到容纳文件的内存空间并且在TLB上护，也需要在unmapping后刷新到磁盘)。page fault会开销更大，因为内核需要从硬盘读取来更新存空间和TLB。因此在引入mmap()时，必须进行性能测试，以避免糟糕的性能。

在Java中，相对应的类是nio中的MappedByteBuffer。

4.2 HeapByteBuffer

通过调用ByteBuffer.allocate()方法得到。这里的Heap表示该ByteBuffer是存放在JVM的堆空间中，此，它也支持GC和缓存优化。然而，它不是页对齐的，这就意味着，如果你需要通过JNI来调用时，JM需要复制一份至对齐缓部区。

4.3 DirectByteBuffer

通过调用ByteBuffer.allocateDirect()方法得到。JVM会使用malloc()直接在堆外分配内存。因为它不受JVM的管理，分配的内存是页对齐的，也不受GC的管理。这就意味着，如果需要在native code中作（比如在写OpenGL时），DirectByteBuffer就是一个完美的方案。然而，你需要去自己管理内存分配与释放以避免内存泄漏。

4.4 MappedByteBuffer

通过调用FileChannel.map()方法得到。类似于DirectByteBuffer，它也是分配在JVM堆外的。Mapp

dByteBuffer本质上就是OS mmap()的一个包装，以便代码直接操作映射的物理内存。

4.5 小结

sendfile() 与 mmap()在操作系统底层，提供了高效、低延时的socket数据操作解决方案。这里强调，在实际开发中，应用场景各不相同，所以不存在绝对的银弹。如果性能、延伸没有达到瓶颈，不需花精力去把代码切换到这两个解决方案上。在软件实施过程中，在大多数情况下，获得最佳的ROI（资回报率），是使软件能够工作正常，而不是使软件工作更快。由于脱离JVM的保护，对于复杂逻辑很容易使软件变得不可靠，更容易崩溃。

参考资料

1. http://tutorials.jenkov.com/java-nio/nio-vs-io.html

2. https://www.quora.com/Why-is-NIO-much-more-efficient-than-BIO-even-though-NIO-doent-save-any-CPU-circles

3. https://xunnanxu.github.io/2016/09/10/It-s-all-about-buffers-zero-copy-mmap-and-Java-NO/