以Java的视角来聊聊BIO、NIO与AIO的区别?

> 题目：说一下BIO/AIO/NIO 有什么区别？及异步模式的用途和意义？

1F

说一说I/O

首先来说一下什么是I/O?

在计算机系统中I/O就是输入（Input）和输出(Output)的意思，针对不同的操作对象，可以划分为磁盘I/O模型，网络I/O模型，内存映射I/O, Direct I/O、数据库I/O等，只要具有输入输出类型的交互系统都可以认为是I/O系统，也可以说I/O是整个操作系统数据交换与人机交互的通道，这个概念与选用的开发语言没有关系，是一个通用的概念。

在如今的系统中I/O却拥有很重要的位置，现在系统都有可能处理大量文件，大量数据库操作，而这些操作都依赖于系统的I/O性能，也就造成了现在系统的瓶颈往往都是由于I/O性能造成的。因此，为了解决磁盘I/O性能慢的问题，系统架构中添加了缓存来提高响应速度；或者有些高端服务器从硬件级入手，使用了固态硬盘（SSD）来替换传统机械硬盘；在大数据方面，Spark越来越多的承担了实时性计算任务，而传统的Hadoop体系则大多应用在了离线计算与大量数据存储的场景，这也是由于磁盘I/O性能远不如内存I/O性能而造成的格局（Spark更多的使用了内存，而MapReduece更多的使用了磁盘）。因此，一个系统的优化空间，往往都在低效率的I/O环节上，很少看到一个系统CPU、内存的性能是其整个系统的瓶颈。也正因为如此，Java在I/O上也一直在做持续的优化，从JDK 1.4开始便引入了NIO模型，大大的提高了以往BIO模型下的操作效率。

这里先给出BIO、NIO、AIO的基本定义与类比描述：

- BIO （Blocking I/O）：同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。这里使用那个经典的烧开水例子，这里假设一个烧开水的场景，有一排水壶在烧开水，BIO的工作模式就是， 叫一个线程停留在一个水壶那，直到这个水壶烧开，才去处理下一个水壶。但是实际上线程在等待水壶烧开的时间段什么都没有做。

- NIO （New I/O）：同时支持阻塞与非阻塞模式，但这里我们以其同步非阻塞I/O模式来说明，那么什么叫做同步非阻塞？如果还拿烧开水来说，NIO的做法是叫一个线程不断的轮询每个水壶的状态，看看是否有水壶的状态发生了改变，从而进行下一步的操作。

线程来处理。对应到烧开水中就是，为每个水壶上面装了一个开关，水烧开之后，水壶会自动通知我水烧开了。

进程中的IO调用步骤大致可以分为以下四步：

1. 进程向操作系统请求数据；
2. 操作系统把外部数据加载到内核的缓冲区中;
3. 操作系统把内核的缓冲区拷贝到进程的缓冲区；
4. 进程获得数据完成自己的功能；

当操作系统在把外部数据放到进程缓冲区的这段时间（即上述的第二，三步），如果应用进程是挂起等待的，那么就是同步IO，反之，就是异步IO，也就是AIO 。

2F

BIO（Blocking I/O）同步阻塞I/O

这是最基本与简单的I/O操作方式，其根本特性是做完一件事再去做另一件事，一件事一定要等前一件事做完，这很符合程序员传统的顺序来开发思想，因此BIO模型程序开发起来较为简单，易于把握。

但是BIO如果需要同时做很多事情（例如同时读很多文件，处理很多tcp请求等），就需要系统创建很多线程来完成对应的工作，因为BIO模型下一个线程同时只能做一个工作，如果线程在执行过程中依赖于需要等待的资源，那么该线程会长期处于阻塞状态，我们知道在整个操作系统中，线程是系统执行的基本单位，在BIO模型下的线程 阻塞就会导致系统线程的切换，从而对整个系统性能造成一定的影响。当然如果我们只需要创建少量可控的线程，那么采用BIO模型也是很好的选择，但如果在需要考虑高并发的web或者tcp服务器中采用BIO模型就无法应对了，如果系统开辟成千上万的线程，那么CPU的执行时机都会浪费在线程的切换中，使得线程的执行效率大大降低。此外，关于线程这里说一句题外话，在系统开发中线程的生命周期一定要准确控制，在需要一定规模并发的情形下，尽量使用线程池来确保线程创建数目在一个合理的范围之内，切莫编写线程数量创建上限的代码。

3F

NIO (New I/O) 同步非阻塞I/O

关于NIO，国内有很多技术博客将英文翻译成No-Blocking I/O，非阻塞I/O模型 ，当然这样就与BIO形成了鲜明的特性对比。NIO本身是基于事件驱动的思想来实现的，其目的就是解决BIO的大并发问

机制，以socket使用来说，多路复用器通过不断轮询各个连接的状态，只有在socket有流可读或者可写时，应用程序才需要去处理它，在线程的使用上，就不需要一个连接就必须使用一个处理线程了，而是只是有效请求时（确实需要进行I/O处理时），才会使用一个线程去处理，这样就避免了BIO模型下大量线程处于阻塞等待状态的情景。

相对于BIO的流，NIO抽象出了新的通道（Channel）作为输入输出的通道，并且提供了缓存（Buffer）的支持，在进行读操作时，需要使用Buffer分配空间，然后将数据从Channel中读入Buffer中，对于Channel的写操作，也需要现将数据写入Buffer，然后将Buffer写入Channel中。

如下是NIO方式进行文件拷贝操作的示例，见下图：

```
public static void copyFile( String srcFileName, String dstFileName ) throws IOException {
    FileInputStream fis = new FileInputStream(srcFileName);
    FileOutputStream fos = new FileOutputStream(dstFileName);

    FileChannel readChannel = fis.getChannel();
    FileChannel writeChannel = fos.getChannel();

    ByteBuffer buffer = ByteBuffer.allocate(1024);

    while ( true ) {
        buffer.clear(); //清空缓存区
        if ( readChannel.read(buffer) == -1 ){  //从输入Channel中读取数据到buffer中
            break;
        }
        buffer.flip();  //将缓存区游标置于0
        writeChannel.write(buffer); //将缓存中数据写入输出Channel中
    }

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


通过比较New IO的使用方式我们可以发现，新的IO操作不再面向 Stream来进行操作了，改为了通道Channel，并且使用了更加灵活的缓存区类Buffer，Buffer只是缓存区定义接口， 根据需要，我们可以选择对应类型的缓存区实现类。在java NIO编程中，我们需要理解以下3个对象Channel、Buffer和Selector。

- **Channel**

首先说一下Channel，国内大多翻译成“通道”。Channel和IO中的Stream(流)是差不多一个等级的。只不过Stream是单向的，譬如：InputStream, OutputStream。而Channel是双向的，既可以用来进行读操作，又可以用来进行写操作，NIO中的Channel的主要实现有：FileChannel、DatagramChannel、SocketChannel、ServerSocketChannel；通过看名字就可以猜出个所以然来：分别可以对应文件IO、UDP和TCP（Server和Client）。

- **Buffer**

NIO中的关键Buffer实现有：ByteBuffer、CharBuffer、DoubleBuffer、 FloatBuffer、IntBuffer、LongBuffer,、ShortBuffer，分别对应基本数据类型: byte、char、double、 float、int、 long、short。当然NIO中还有MappedByteBuffer, HeapByteBuffer, DirectByteBuffer等这里先不具体陈述其用法细节。

说一下 DirectByteBuffer 与 HeapByteBuffer 的区别?

它们 ByteBuffer 分配内存的两种方式。HeapByteBuffer 顾名思义其内存空间在 JVM 的 heap（堆）上分配，可以看做是 jdk 对于 byte[] 数组的封装；而 DirectByteBuffer 则直接利用了系统接口进行内存申请，其内存分配在c heap 中，这样就减少了内存之间的拷贝操作，如此一来，在使用 DirectByteBuffer 时，系统就可以直接从内存将数据写入到 Channel 中，而无需进行 Java 堆的内存申请，复制等操作，提高了性能。既然如此，为什么不直接使用 DirectByteBuffer，还要来个 HeapByteBuffer? 原因在于， DirectByteBuffer 是通过full gc来回收内存的，DirectByteBuffer会自己检测情况而调用 system.gc()，但是如果参数中使用了 DisableExplicitGC 那么就无法回收该快内存了，-XX:+DisableExplicitGC标志自动将 System.gc() 调用转换成一个空操作，就是应用中调用 System.gc() 会变成一个空操作，那么如果设置了就需要我们手动来回收内存了，所以DirectByteBuffer使用起来相对于完全托管于 java 内存管理的Heap ByteBuffer 来说更复杂一些，如果用不好可能会引起OOM。Direct ByteBuffer 的内存大小受 -XX:MaxDirectMemorySize JVM 参数控制（默认大小64M），在 DirectByteBuffer 申请内存空间达到该设置大小后，会触发 Full GC。

- **Selector**

Selector 是NIO相对于BIO实现多路复用的基础，Selector 运行单线程处理多个 Channel，如果你的应用打开了多个通道，但每个连接的流量都很低，使用 Selector 就会很方便。例如在一个聊天服务器中。要使用 Selector , 得向 Selector 注册 Channel，然后调用它的 select() 方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新的连接进来、数据接收等。

这里我们再来看一个NIO模型下的TCP服务器的实现，我们可以看到Selector 正是NIO模型下 TCP Server 实现IO复用的关键，请仔细理解下段代码while循环中的逻辑，见下图:

```java
private static int TIMEOUT = 3000;
public static void nioTcpserver( int listenPort ) throws IOException {
    //创建一个选择器
    Selector selector = Selector.open();

    //实例化一个信道
    ServerSocketChannel listnChannel = ServerSocketChannel.open();
    //将该信道绑定到指定端口
    listnChannel.socket().bind(new InetSocketAddress(listenPort));
    //配置信道为非阻塞模式
    listnChannel.configureBlocking(false);
    //将选择器注册到各个信道
    listnChannel.register(selector, SelectionKey.OP_ACCEPT);

    //不断轮询select方法，获取准备好的信道所关联的Key集
    while (true){
        //一直等待,直至有信道准备好了I/O操作
        if (selector.select(TIMEOUT) == 0){
            //在等待信道准备的同时，也可以异步地执行其他任务，
            continue;
        }
        //获取准备好的信道所关联的Key集合的iterator实例
        Iterator<SelectionKey> keyIter = selector.selectedKeys().iterator();
        //循环取得集合中的每个键值
        while (keyIter.hasNext()){
            SelectionKey key = keyIter.next();
            //如果服务端信道感兴趣的I/O操作为accept
            if (key.isAcceptable()){
                handleAccept(key);
            }
            //如果客户端信道感兴趣的I/O操作为read
            if (key.isReadable()){
                handleAccept(key);
            }
            //如果该键值有效，并且其对应的客户端信道感兴趣的I/O操作为write
            if (key.isValid() && key.isWritable()) {
                handleWrite(key);
            }
            //这里需要手动从键集中移除当前的key
            keyIter.remove();
        }
    }
}
```


4F

AIO (Asynchronous I/O) 异步非阻塞I/O

Java AIO就是Java作为对异步IO提供支持的NIO.2 ， Java NIO2 (JSR 203)定义了更多的 New I/O APIs， 提案2003提出，直到2011年才发布， 最终在JDK 7中才实现。JSR 203除了提供更多的文件系统操作API(包括可插拔的自定义的文件系统)， 还提供了对socket和文件的异步 I/O操作。 同时实现了JSR-51提案中的socket channel全部功能,包括对绑定， option配置的支持以及多播multicast的

从编程模式上来看AIO相对于NIO的区别在于，NIO需要使用者线程不停的轮询IO对象，来确定是否有数据准备好可以读了，而AIO则是在数据准备好之后，才会通知数据使用者，这样使用者就不需要不停地轮询了。当然AIO的异步特性并不是Java实现的伪异步，而是使用了系统底层API的支持，在Unix系统下，采用了epoll IO模型，而windows便是使用了IOCP模型。关于Java AIO，本篇只做一个抛砖引玉的介绍，如果你在实际工作中用到了，那么可以参考Netty在高并发下使用AIO的相关技术。

> **总 结**
>
> IO实质上与线程没有太多的关系，但是不同的IO模型改变了应用程序使用线程的方式，NIO与AIO的出现解决了很多BIO无法解决的并发问题，当然任何技术抛开适用场景都是耍流氓，复杂的技术往往是为了解决简单技术无法解决的问题而设计的，在系统开发中能用常规技术解决的问题，绝不用复杂技术，否则大大增加系统代码的维护难度，学习IT技术不是为了炫技，而是要实实在在解决问题。