Java Standard IO vs New IO

> 1，Java标准io和NIO的区别
> 2，什么场景下适合使用NIO ，什么场景下适合使用标准IO
> 3，**为什么要使用 NIO?**

Java New input/output (NIO) was introduced with JDK 1.4. Picking up where Standard IO leaves, NIO provides high-speed, block-oriented IO to Java library.
JDK 1.4引入了新的输入/输出(NIO)。继标准IO之后，NIO为Java库提供了高速的、面向块的IO。
By defining classes to hold data, and by processing that data in blocks, NIO takes advantage of low-level optimizations in a way that the `java.io` package could not, without using native code.
通过定义类来保存数据，并以块的形式处理数据，NIO以一种与java不同的方式利用了低级优化。IO包不能，没有使用本地代码。
In this article, we will focus on identifying the most noticeable differences between Standard IO vs New IO which we must know before deciding which one to use in our next project.
在本文中，我们将重点识别标准IO和新IO之间最显著的区别，在决定在下一个项目中使用哪个IO之前，我们必须知道这些区别。

Recalling Standard IO 召回标准IO

Java IO refers to the interface between a computer and the rest of the world, or between a single program and the rest of the computer.
Java IO指的是计算机和其他世界之间的接口，或者是一个程序和计算机的其他部分之间的接口。
In Java programming, IO classes have until recently been carried out using a stream metaphor. All IO is viewed as the movement of single bytes, one at a time, through an object called a `Stream`.
在Java编程中，IO类**直到最近**才使用流隐喻来实现。所有的IO都被看作是单个字节的移动，一次一个，通过一个名为Stream的对象。
Stream IO is used for contacting the outside world. It is also used internally, for turning objects into bytes and then back into objects. It is known as serialization and deserialization.
流IO用于与外部世界联系。它也可以在内部使用，用于将对象转换为字节，然后再转换为对象。它被称为序列化和反序列化。

为什么要使用 NIO?

NIO 的创建目的是为了让 Java 程序员可以实现高速 I/O 而无需编写自定义的本机代码。NIO 将最耗时的 I/O 操作(即填充和提取缓冲区)转移回操作系统，因而可以极大地提高速度。

Introducing Java New IO Java New IO简介

Java NIO was created to allow Java programmers to implement high-speed input-output operations without having to write custom **native code.**
创建Java NIO是为了允许Java程序员实现高速的输入-输出操作，而无需编写定制的本机代码。
NIO moves the most time-consuming I/O activities (namely, filling and draining buffers) back into the operating system, thus allowing for a great increase in speed.
NIO将最耗时的I/O活动(即填充和清空缓冲区)移回操作系统，从而大大提高了速度。 **（NIO为什么能够提高速度）**
If above introductions have left you thirsty then don’ t worry if you will feel better as we go forward. Let’ s start by finding the differences.
如果上面的介绍让你感到口渴，那么不要担心，如果我们继续下去，你会感觉更好。让我们从求差开始。

Differences between IO and NIO

IO Streams vs NIO Blocks

The most important distinction between the standard IO library (`java.io.*`) and New IO (`java.nio.*`) is how data is packaged and transmitted from the source to the target. As previously mentioned, standard I/O deals with the data in streams, whereas NIO deals with the data in blocks.

标准IO库(java.io.*)和新IO (java.nio.*)之间最重要的区别是数据如何打包和从源到目标的传输。如前所述，标准I/O处理流中的数据，而NIO处理块中的数据(

面向流与面向缓冲:Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。 Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。)

A stream-oriented I/O system deals with data one or more bytes at a time. An input stream produces one byte of data, and an output stream consumes one byte of data. It is very easy to create filters for streamed data. It is also relatively simply to chain several filters together so that each one does its part in what amounts to a single, sophisticated processing mechanism.

面向流 的 I/O 系统一次一个字节地处理数据。一个输入流产生一个字节的数据，一个输出流消费一个字节的数据。为流式数据创建过滤器非常容易。链接几个过滤器，以便每个过滤器只负责单个复杂处理机制的一部分，这样也是相对简单的。不利的一面是，面向流的 I/O 通常相当慢。

Important thing is that bytes are not cached anywhere. Furthermore, we cannot move forth and back in the data in a stream. If you need to move forth and back in the data read from a stream, we must cache it in a buffer first.

重要的是字节不会被缓存到任何地方。此外，我们不能在流中的数据中来回移动。如果需要在从流中读取的数据中来回移动，必须首先将其缓存到缓冲区中。

A block-oriented I/O system deals with data in blocks. Each operation produces or consumes a block of data in one step. Processing data by the block can be much faster than processing it by the (streamed) byte. You can move forth and back in the buffer as you need to.

面向块的I/O系统以块的形式处理数据。每个操作在一个步骤中产生或消耗一个数据块。按块处理数据比按(流)字节处理数据要快得多。您可以根据需要在缓冲区中前后移动。

Data blocks give us a bit more flexibility during processing. However, we also need to check if the buffer contains all the data we need in order to fully process it. And, we need to make sure that when reading more data into the buffer, we do not overwrite data in the buffer we have not yet processed.

On downside, block-oriented I/O lacks some of the elegance and simplicity of stream-oriented I/O.

数据块在处理过程中给了我们更多的灵活性。但是，我们还需要检查缓冲区是否包含我们需要的所有数据，以便完全处理它。而且，我们需要确保在读取更多数据到缓冲区时，不会覆盖缓冲区中尚未处理的数据。

缺点是，面向块的I/O缺乏一些面向流的I/O的优雅和简单性。

NIO性能的优势就来源于缓冲的机制，不管是读或者写都需要以块的形式写入到缓冲区中。NIO实际上让我们对IO的操作更接近于操作系统的实际过程。

> Read more: 3 ways to read files using Java NIO

Synchronous Standard vs Asynchronous New IO

Java IO' s various streams are blocking or synchronous. That means, that when a thread invokes a `read()` or `write()` operation, that thread is blocked until there is some data to read, or the data is fully written.

The thread will be in blocked state for this period. This has been cited as a good solid reason for bringing multi-threading in modern languages.
Java IO的各种流是阻塞的或同步的。这意味着，当线程调用read()或write()操作时，该线程将被阻塞，直到有一些数据要读或数据完全写入为止。线程在此期间将处于阻塞状态。这被认为是在现代语言中引入多线程的一个很好的理由。

In asynchronous IO, a thread can request that some data be written to a channel, but not wait for it to be fully written. The thread can then go on and do something else in the mean time. Usually these threads spend their idle time on when not blocked in IO calls, is usually performing IO on other channels in the meantime. That is, a single thread can now manage multiple channels of input and output.
在异步IO中，线程可以请求将一些数据写入通道，但不需要等待数据完全写入。然后线程可以继续在同一时间做其他事情。通常这些线程将空闲时间花费在IO调用中没有阻塞时，通常同时在其他通道上执行IO。也就是说，一个线程现在可以管理多个输入和输出通道。

Synchronous programs often have to resort to polling, or to the creation of many, many threads, to deal with lots of connections. With asynchronous I/O, you can listen for I/O events on an arbitrary number of channels, without polling and without extra threads.
同步程序经常不得不求助于轮询，或者创建很多很多的线程来处理大量的连接。使用异步I/O，您可以在任意数量的通道上侦听I/O事件，而无需轮询和额外的线程。
The central object in asynchronous I/O is called the Selector. A Selector is where you register your interest for various IO events, and it is the object that tells you when those events occur. So, the first thing we need to do is create a Selector.
异步I/O中的中心对象称为Selector。Selector是你为各种IO事件注册兴趣的地方，它是告诉你这些事件何时发生的对象。我们需要做的第一件事是创建一个Selector。

```
Selector selector = Selector.open();
```

Later on, we will call the `register()` method on various `Channel` objects, in order to register our interest in IO events happening inside those objects. The first argument to `register()` is always the `Selector`.
稍后，我们将在各种Channel对象上调用register()方法，以便注册我们对这些对象内部发生的IO事件感兴趣的内容。register()的第一个参数总是Selector。

> Read more: How to define Path in java NIO

Java IO vs NIO APIs
No prize for guessing that the API calls when using NIO look different than when using IO. Here in NIO, rather than just reading the data byte for byte from e.g. an `InputStream`, the data must first be read into a `Buffer`, and then be processed from thereafter.
无需猜测，使用NIO时的API调用与使用IO时的调用看起来不同。在NIO中，不是从例如InputStream中逐个字节地读取数据，而是必须首先将数据读入Buffer，然后再对其进行处理。
Java Example to read a file using Standard IO.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class WithoutNIOExample
{
    public static void main(String[] args)
    {
```

```
        BufferedReader br = null;
         String sCurrentLine = null;
         try
         {
             br = new BufferedReader(
             new FileReader("test.txt"));
             while ((sCurrentLine = br.readLine()) != null)
             {
                 System.out.println(sCurrentLine);
             }
         }
         catch (IOException e)
         {
             e.printStackTrace();
         }
         finally
         {
             try
             {
                 if (br != null)
                 br.close();
             } catch (IOException ex)
             {
                 ex.printStackTrace();
             }
         }
     }
}
```

Java Example to read a file using New IO.

```
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class ReadFileWithFixedSizeBuffer
{
    public static void main(String[] args) throws IOException
    {
         RandomAccessFile aFile = new RandomAccessFile
```

```
                ("test.txt", "r");
        FileChannel inChannel = aFile.getChannel();
        ByteBuffer buffer = ByteBuffer.allocate(1024);
  //将数据读入到Buffer中
        while(inChannel.read(buffer) > 0)
        {
            buffer.flip();
            for (int i = 0; i < buffer.limit(); i++)
            {
                System.out.print((char) buffer.get());
            }
            buffer.clear(); // do something with the data and clear/comp
act it.
        }
        inChannel.close();
        aFile.close();
    }
}
```

Conclusion

NIO allows you to manage multiple channels using only a single (or fewer) threads, but the cost is that parsing the data might be somewhat more complicated than when reading data from a blocking stream using standard IO.

NIO允许仅使用一个(或更少)线程管理多个通道，但代价是解析数据可能比使用标准IO从阻塞流中读取数据要复杂一些。

If you need to manage thousands of open connections simultaneously, which each only send a little data, for instance a chat server, implementing the server in NIO is probably an advantage. Similarly, if you need to keep a lot of open connections to other computers, e.g. in a P2P network, using a single thread to manage all of your outbound connections might be an advantage.

如果您需要同时管理数千个打开的连接，每个连接只发送少量数据(例如聊天服务器)，那么在NIO中实现服务器可能是一个优势。类似地，如果你需要保持与其他计算机的大量开放连接，例如在P2P网络中，使用单个线程来管理所有出站连接可能是一个优势。nio的适用于io高并发场景

If you have fewer connections with very high bandwidth, sending a lot of data at a time, standard IO server implementation should be your choice.

如果你有很少的连接和非常高的带宽，一次发送大量的数据，标准IO服务器实现应该是你的选择。