1、Java 中 IO 流？

Java 中 IO 流分为几种?

1. 按照流的流向分，可以分为输入流和输出流；
2. 按照操作单元划分，可以划分为字节流和字符流；
3. 按照流的角色划分为节点流和处理流。

Java Io 流共涉及 40 多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系， Java I0 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

1. InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。
2. OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

2、 Java IO与 NIO的区别

NIO即New IO，这个库是在JDK1.4中才引入的。NIO和IO有相同的作用和目的，但实现方式不同，NIO主要用到的是块，所以NIO的效率要比IO高很多。在Java API中提供了两套NIO，一套是针对标准输入输出NIO，另一套就是网络编程NIO。

3、常用io类有那些

```
File
FileInputSteam，FileOutputStream
BufferInputStream，BufferedOutputSream
PrintWrite
FileReader，FileWriter
BufferReader，BufferedWriter
ObjectInputStream，ObjectOutputSream
```

4、字节流与字符流的区别

以字节为单位输入输出数据，字节流按照8位传输 以字符为单位输入输出数据，字符流按照16位传输

5、阻塞 IO 模型

最传统的一种 IO 模型，即在读写数据过程中会发生阻塞现象。当用户线程发出 IO 请求之后，内核会去查看数据是否就绪，如果没有就绪就会等待数据就绪，而用户线程就会处于阻塞状态，用户线程交出 CPU。当数据就绪之后，内核会将数据拷贝到用户线程，并返回结果给用户线程，用 户线程才解除 block 状态。典型的阻塞 IO 模型的例子为： data = socket.read();如果数据没有就绪，就会一直阻塞在 read 方法

6、非阻塞 IO 模型

当用户线程发起一个 read 操作后，并不需要等待，而是马上就得到了一个结果。 如果结果是一个error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作。一旦内核中的数据准备好了，并且又再次收到了用户线程的请求，那么它马上就将数据拷贝到了用户线程，然后返回。所以事实上，在非阻塞 IO 模型中，用户线程需要不断地询问内核数据是否就绪，也就说非阻塞 IO不会交出 CPU，而会一直占用 CPU。 典型的非阻塞 IO 模型一般如下：

```
while(true){
    data = socket.read();
    if(data!= error){
        //处理数据
        break;
    }
}
```

但是对于非阻塞 IO 就有一个非常严重的问题， 在 while 循环中需要不断地去询问内核数据是否就绪，这样会导致 CPU 占用率非常高，因此一般情况下很少使用 while 循环这种方式来读取数据。

7、多路复用 IO 模型

多路复用 IO 模型是目前使用得比较多的模型。 Java NIO 实际上就是多路复用 IO。在多路复用 IO模型中，会有一个线程不断去轮询多个 socket 的状态，只有当 socket 真正有读写事件时，才真正调用实际的 IO 读写操作。因为在多路复用 IO 模型中，只需要使用一个线程就可以管理多个socket，系统不需要建立新的进程或者线程，也不必维护这些线程和进程，并且只有在真正有socket 读写事件进行时，才会使用 IO 资源，所以它大大减少了资源占用。在 Java NIO 中，是通过 selector.select()去查询每个通道是否有到达事件，如果没有事件，则一直阻塞在那里，因此这种方式会导致用户线程的阻塞。多路复用 IO 模式，通过一个线程就可以管理多个 socket，只有当 socket 真正有读写事件发生才会占用资源来进行实际的读写操作。因此，多路复用 IO 比较适合连接数比较多的情况。

另外多路复用 IO 为何比非阻塞 IO 模型的效率高是因为在非阻塞 IO 中，不断地询问 socket 状态时通过用户线程去进行的，而在多路复用 IO 中，轮询每个 socket 状态是内核在进行的，这个效率要比用户线程要高的多。

不过要注意的是，多路复用 IO 模型是通过轮询的方式来检测是否有事件到达，并且对到达的事件 逐一进行响应。因此对于多路复用 IO 模型来说， 一旦事件响应体很大，那么就会导致后续的事件 迟迟得不到处理，并且会影响新的事件轮询。

8、信号驱动 IO 模型

在信号驱动 IO 模型中，当用户线程发起一个 IO 请求操作，会给对应的 socket 注册一个信号函数，然后用户线程会继续执行，当内核数据就绪时会发送一个信号给用户线程，用户线程接收到信号之后，便在信号函数中调用 IO 读写操作来进行实际的 IO 请求操作。

9、异步 IO 模型

异步 IO 模型才是最理想的 IO 模型，在异步 IO 模型中，当用户线程发起 read 操作之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度，当它受到一个 asynchronous read 之后，它会立刻返回，说明 read 请求已经成功发起了，因此不会对用户线程产生任何 block。然后，内核会等待数据准备完成，然后将数据拷贝到用户线程，当这一切都完成之后，内核会给用户线程发送一个信号，告诉它 read 操作完成了。也就说用户线程完全不需要实际的整个 IO 操作是如何进行的， 只需要先发起一个请求，当接收内核返回的成功信号时表示 IO 操作已经完成，可以直接去使用数据了。

也就说在异步 IO 模型中， IO 操作的两个阶段都不会阻塞用户线程，这两个阶段都是由内核自动完成，然后发送一个信号告知用户线程操作已完成。用户线程中不需要再次调用 IO 函数进行具体的读写。这点是和信号驱动模型有所不同的，在信号驱动模型中，当用户线程接收到信号表示数据已经就绪，然后需要用户线程调用 IO 函数进行实际的读写操作；而在异步 IO 模型中，收到信号表示 IO 操作已经完成，不需要再在用户线程中调用 IO 函数进行实际的读写操作。

注意，异步 IO 是需要操作系统的底层支持，在 Java 7 中，提供了 Asynchronous IO。 更多参考： http://www.importnew.com/19816.html

10、JAVA NIO

NIO 主要有三大核心部分： Channel(通道)， Buffer(缓冲区), Selector。传统 IO 基于字节流和字符流进行操作， 而 NIO 基于 Channel 和 Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。 Selector(选择区)用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。 NIO 和传统 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。

11、NIO 的缓冲区

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。 NIO 的缓冲导向方法不同。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。

12、NIO 的非阻塞

IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 NIO 的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去

做别的事情。 线程通常将非阻塞 IO 的空闲时间用于在其它通道上执行 IO 操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

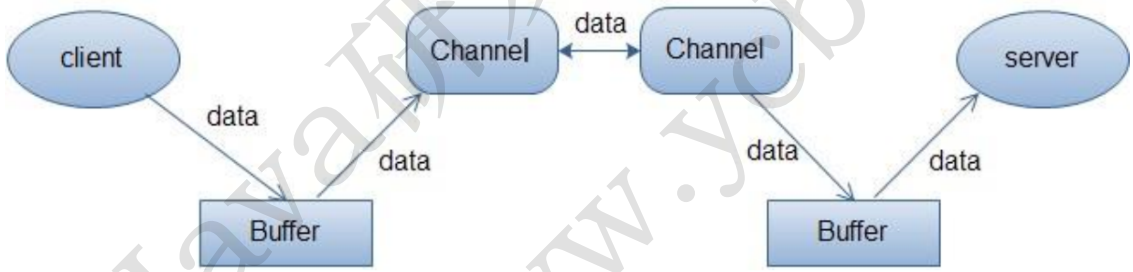
13、Channel

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：

1. FileChannel
2. DatagramChannel
3. SocketChannel
4. ServerSocketChannel 这里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）。 下面演示的案例基本上就是围绕这 4 个类型的 Channel 进行陈述的。

14、Buffer

Buffer，故名思意， 缓冲区，实际上是一个容器，是一个连续数组。 Channel 提供从文件、网络读取数据的渠道，但是读取或写入的数据都必须经由 Buffer。



上面的图描述了从一个客户端向服务端发送数据，然后服务端接收数据的过程。客户端发送数据时，必须先将数据存入 Buffer 中，然后将 Buffer 中的内容写入通道。服务端这边接收数据必须通过 Channel 将数据读入到 Buffer 中，然后再从 Buffer 中取出数据来处理。

在 NIO 中， Buffer 是一个顶层父类，它是一个抽象类，常用的 Buffer 的子类有：ByteBuffer、IntBuffer、 CharBuffer、 LongBuffer、 DoubleBuffer、 FloatBuffer、ShortBuffer

15、Selector

Selector 类是 NIO 的核心类， Selector 能够检测多个注册的通道上是否有事件发生，如果有事件发生，便获取事件然后针对每个事件进行相应的响应处理。这样一来，只是用一个单线程就可以管理多个通道，也就是管理多个连接。这样使得只有在连接真正有读写事件发生时，才会调用函数来进行读写，就大大地减少了系统开销，并且不必为每个连接都创建一个线程，不用去维护多个线程，并且避免了多线程之间的上下文切换导致的开销。