**一、IO流的概念**

Java中对文件的操作是以流的方式进行的。流是Java内存中的一组有序数据序列。Java将数据从源（文件、内存、键盘、网络）读入到内存中，形成了流，然后将这些流还可以写到另外的目的地（文件、内存、控制台、网络），之所以称为流，是因为这个数据序列在不同时刻所操作的是源的不同部分。

**二、IO流的分类**

Java中的流，可以从不同的角度进行分类。

按照 **数据** **流的方向** 不同可以分为：输入流和输出流。

按照 **处理数据单位** 不同可以分为：字节流和字符流。

按照 **实现功能** 不同可以分为：节点流和处理流。

**字节流：** 一次读入或读出是 **8位** 二进制。

**字符流：** 一次读入或读出是 **16位** 二进制。

字节流和字符流的原理是相同的，只不过处理的单位不同而已。后缀是Stream是字节流，而后缀是Reader，Writer是字符流。

**节点流：** 直接与数据源相连，读入或读出。

直接使用节点流，读写不方便，为了更快的读写文件，才有了处理流。

**处理流：** 与节点流一块使用，在节点流的基础上，再套接一层，套接在节点流上的就是处理流。

Jdk提供的流继承了四大类：InputStream(字节输入流)，OutputStream（字节输出流），Reader（字符输入流），Writer（字符输出流）。

**2.1 关于字节流和字符流的区别**

实际上字节流在操作的时候本身是不会用到缓冲区的，是文件本身的直接操作的，但是字符流在操作的 时候下后是会用到缓冲区的，是通过缓冲区来操作文件的。

读者可以试着将上面的字节流和字符流的程序的最后一行关闭文件的代码注释掉，然后运行程序看看。你就会发现使用字节流的话，文件中已经存在内容，但是使用字符流的时候，文件中还是没有内容的，这个时候就要刷新缓冲区。

**使用字节流好还是字符流好呢？**

答案是字节流。首先因为硬盘上的所有文件都是以字节的形式进行传输或者保存的，包括图片等内容。但是字符只是在内存中才会形成的，所以在开发中，字节流使用广泛。

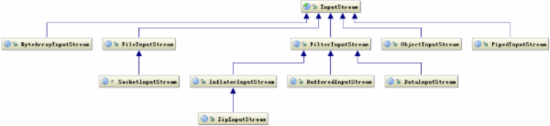
**三、IO框架结构**

我们以字节流与字符流的分类来分析io类的关系与结构。

**3.1 基于字节的 IO 操作接口**

基于字节的 IO 操作接口输入和输出分别是：InputStream 和 OutputStream，InputStream 输入流的类继承层次如下图所示：

图 1. InputStream 相关类层次结构



输入流根据数据类型和操作方式又被划分成若干个子类，每个子类分别处理不同操作类型，OutputStream 输出流的类层次结构也是类似，如下图所示：

图 2. OutputStream 相关类层次结构



这里就不详细解释每个子类如何使用了，如果不清楚的话可以参考一下 JDK 的 API 说明文档，这里只想说明两点，一个是操作数据的方式是可以组合使用的，如这样组合使用

OutputStream out = new BufferedOutputStream(**new** ObjectOutputStream(**new** FileOutputStream("fileName"))；

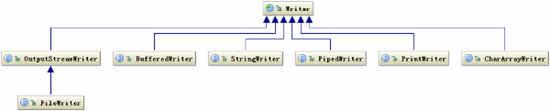
还有一点是流最终写到什么地方必须要指定，要么是写到磁盘要么是写到网络中，其实从上面的类图中我们发现，写网络实际上也是写文件，只不过写网络还有一步需要处理就是底层操作系统再将数据传送到其它地方而不是本地磁盘。关于网络 I/O 和磁盘 I/O 我们将在后面详细介绍。

**3.2 基于字符的 I/O 操作接口**

不管是磁盘还是网络传输，最小的存储单元都是字节，而不是字符，所以 I/O 操作的都是字节而不是字符，但是为啥有操作字符的 I/O 接口呢？这是因为我们的程序中通常操作的数据都是以字符形式，为了操作方便当然要提供一个直接写字符的 I/O 接口，如此而已。我们知道字符到字节必须要经过编码转换，而这个编码又非常耗时，而且还会经常出现乱码问题，所以 I/O 的编码问题经常是让人头疼的问题。关于 I/O 编码问题请参考另一篇文章  [***《深入分析***](http://www.ibm.com/developerworks/cn/java/j-lo-chinesecoding/) [***Java***](http://www.ibm.com/developerworks/cn/java/j-lo-chinesecoding/) [***中的中文编码问题》***](http://www.ibm.com/developerworks/cn/java/j-lo-chinesecoding/) 。

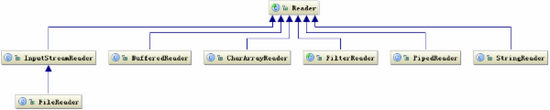
下图是写字符的 I/O 操作接口涉及到的类，Writer 类提供了一个抽象方法 write(char cbuf[], int off, int len) 由子类去实现。

图 3. Writer 相关类层次结构



读字符的操作接口也有类似的类结构，如下图所示：

图 4.Reader 类层次结构

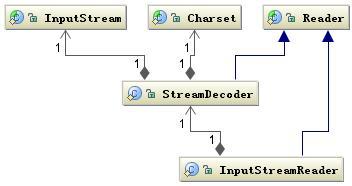


读字符的操作接口中也是 int read(char cbuf[], int off, int len)，返回读到的 n 个字节数，不管是 Writer 还是 Reader 类它们都只定义了读取或写入的数据字符的方式，也就是怎么写或读，但是并没有规定数据要写到哪去，写到哪去就是我们后面要讨论的基于磁盘和网络的工作机制。

**3.3 字节与字符的转化接口**

另外数据持久化或网络传输都是以字节进行的，所以必须要有字符到字节或字节到字符的转化。字符到字节需要转化，其中读的转化过程如下图所示：

图 5. 字符解码相关类结构



InputStreamReader 类是字节到字符的转化桥梁，InputStream 到 Reader 的过程要指定编码字符集，否则将采用操作系统默认字符集，很可能会出现乱码问题。StreamDecoder 正是完成字节到字符的解码的实现类。也就是当你用如下方式读取一个文件时：

清单 1.读取文件

**try** {

StringBuffer str = **new** StringBuffer();

**char**[] buf = **new** **char**[1024];

FileReader f = **new** FileReader("file");

**while**(f.**read**(buf)>0){

str.**append**(buf);

}

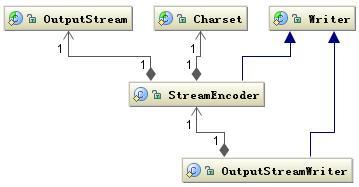
str.toString();

} **catch** (IOException e) {}

FileReader 类就是按照上面的工作方式读取文件的，FileReader 是继承了 InputStreamReader 类，实际上是读取文件流，然后通过 StreamDecoder 解码成 char，只不过这里的解码字符集是默认字符集。

写入也是类似的过程如下图所示：

图 6. 字符编码相关类结构



通过 OutputStreamWriter 类完成，字符到字节的编码过程，由 StreamEncoder 完成编码过程。

**四、具体的IO实现类**

对 **文件** 进行操作：FileInputStream（字节输入流），FileOutputStream（字节输出流），FileReader（字符输入流），FileWriter（字符输出流）

对 **管道** 进行操作：PipedInputStream（字节输入流）,PipedOutStream（字节输出流），PipedReader（字符输入流），PipedWriter（字符输出流）

**PipedInputStream** 的一个实例要和PipedOutputStream的一个实例共同使用，共同完成管道的读取写入操作。主要用于线程操作。

字节/字符数组：ByteArrayInputStream，ByteArrayOutputStream，CharArrayReader，CharArrayWriter是在内存中开辟了一个字节或字符数组。

**Buffered缓冲流** ：BufferedInputStream，BufferedOutputStream，BufferedReader,BufferedWriter,是带缓冲区的处理流，缓冲区的作用的主要目的是：避免每次和硬盘打交道，提高数据访问的效率。

**转化流** ：InputStreamReader/OutputStreamWriter，把字节转化成字符。

**数据流** ：DataInputStream，DataOutputStream。

因为平时若是我们输出一个8个字节的long类型或4个字节的float类型，那怎么办呢？可以一个字节一个字节输出，也可以把转换成字符串输出，但是这样转换费时间，若是直接输出该多好啊，因此这个数据流就解决了我们输出数据类型的困难。数据流可以直接输出float类型或long类型，提高了数据读写的效率。

**打印流** ：printStream，printWriter，一般是打印到控制台，可以进行控制打印的地方。

**对象流** ：ObjectInputStream，ObjectOutputStream，把封装的对象直接输出，而不是一个个在转换成字符串再输出。

**序列化流** ：SequenceInputStream。

**对象序列化** ：把对象直接转换成二进制，写入介质中。

使用对象流需要实现Serializable接口，否则会报错。而若用transient关键字修饰成员变量，不写入该成员变量，若是引用类型的成员变量为null，值类型的成员变量为0.

**请同学们看具体实践代码与例子：** [***http://blog.csdn.net/a511596982/article/details/8284358***](http://blog.csdn.net/a511596982/article/details/8284358)

**五、IO 调优**

下面就磁盘 IO 和网络 IO 的一些常用的优化技巧进行总结如下：

**5.1 磁盘 I/O 优化**

**5.1.1 性能检测**

我们的应用程序通常都需要访问磁盘读取数据，而磁盘 I/O 通常都很耗时，我们要判断 I/O 是否是一个瓶颈，我们有一些参数指标可以参考：

如我们可以压力测试应用程序看系统的 I/O wait 指标是否正常，例如测试机器有 4 个 CPU，那么理想的 I/O wait 参数不应该超过 25%，如果超过 25% 的话，I/O 很可能成为应用程序的性能瓶颈。Linux 操作系统下可以通过 iostat 命令查看。

通常我们在判断 I/O 性能时还会看另外一个参数就是 IOPS，我们应用程序需要最低的 IOPS 是多少，而我们的磁盘的 IOPS 能不能达到我们的要求。每个磁盘的 IOPS 通常是在一个范围内，这和存储在磁盘的数据块的大小和访问方式也有关。但是主要是由磁盘的转速决定的，磁盘的转速越高磁盘的 IOPS 也越高。

现在为了提高磁盘 I/O 的性能，通常采用一种叫 RAID 的技术，就是将不同的磁盘组合起来来提高 I/O 性能，目前有多种 RAID 技术，每种 RAID 技术对 I/O 性能提升会有不同，可以用一个 RAID 因子来代表，磁盘的读写吞吐量可以通过 iostat 命令来获取，于是我们可以计算出一个理论的 IOPS 值，计算公式如下所以：

( 磁盘数 \* 每块磁盘的 IOPS)/( 磁盘读的吞吐量 +RAID 因子 \* 磁盘写的吞吐量 )=IOPS

这个公式的详细信息请查阅参考资料  [***Understanding Disk I/O***](http://blog.scoutapp.com/articles/2011/02/10/understanding-disk-i-o-when-should-you-be-worried) 。

**5.1.2 提升 I/O 性能**

提升磁盘 I/O 性能通常的方法有：

1. 增加缓存，减少磁盘访问次数
2. 优化磁盘的管理系统，设计最优的磁盘访问策略，以及磁盘的寻址策略，这里是在底层操作系统层面考虑的。
3. 设计合理的磁盘存储数据块，以及访问这些数据块的策略，这里是在应用层面考虑的。如我们可以给存放的数据设计索引，通过寻址索引来加快和减少磁盘的访问，还有可以采用异步和非阻塞的方式加快磁盘的访问效率。
4. 应用合理的 RAID 策略提升磁盘 IO，每种 RAID 的区别我们可以用下表所示：

表 1.RAID 策略 

|  |  |
| --- | --- |
| **磁盘阵列** | **说明** |
| RAID 0 | 数据被平均写到多个磁盘阵列中，写数据和读数据都是并行的，所以磁盘的 IOPS 可以提高一倍。 |
| RAID 1 | RAID 1 的主要作用是能够提高数据的安全性，它将一份数据分别复制到多个磁盘阵列中。并不能提升 IOPS 但是相同的数据有多个备份。通常用于对数据安全性较高的场合中。 |
| RAID 5 | 这中设计方式是前两种的折中方式，它将数据平均写到所有磁盘阵列总数减一的磁盘中，往另外一个磁盘中写入这份数据的奇偶校验信息。如果其中一个磁盘损坏，可以通过其它磁盘的数据和这个数据的奇偶校验信息来恢复这份数据。 |
| RAID 0+1 | 如名字一样，就是根据数据的备份情况进行分组，一份数据同时写到多个备份磁盘分组中，同时多个分组也会并行读写。 |

**5.2、网络 I/O 优化**

网络 I/O 优化通常有一些基本处理原则：

1. 一个是减少网络交互的次数：要减少网络交互的次数通常我们在需要网络交互的两端会设置缓存，比如 Oracle 的 JDBC 驱动程序，就提供了对查询的 SQL 结果的缓存，在客户端和数据库端都有，可以有效的减少对数据库的访问。关于 Oracle JDBC 的内存管理可以参考《 Oracle JDBC 内存管理》。除了设置缓存还有一个办法是，合并访问请求：如在查询数据库时，我们要查 10 个 id，我可以每次查一个 id，也可以一次查 10 个 id。再比如在访问一个页面时通过会有多个 js 或 css 的文件，我们可以将多个 js 文件合并在一个 HTTP 链接中，每个文件用逗号隔开，然后发送到后端 Web 服务器根据这个 URL 链接，再拆分出各个文件，然后打包再一并发回给前端浏览器。这些都是常用的减少网络 I/O 的办法。
2. 减少网络传输数据量的大小：减少网络数据量的办法通常是将数据压缩后再传输，如 HTTP 请求中，通常 Web 服务器将请求的 Web 页面 gzip 压缩后在传输给浏览器。还有就是通过设计简单的协议，尽量通过读取协议头来获取有用的价值信息。比如在代理程序设计时，有 4 层代理和 7 层代理都是来尽量避免要读取整个通信数据来取得需要的信息。
3. 尽量减少编码：通常在网络 I/O 中数据传输都是以字节形式的，也就是通常要序列化。但是我们发送要传输的数据都是字符形式的，从字符到字节必须编码。但是这个编码过程是比较耗时的，所以在要经过网络 I/O 传输时，尽量直接以字节形式发送。也就是尽量提前将字符转化为字节，或者减少字符到字节的转化过程。
4. 根据应用场景设计合适的交互方式：所谓的交互场景主要包括同步与异步阻塞与非阻塞方式，下面将详细介绍。

**5.2.1 同步与异步**

所谓同步就是一个任务的完成需要依赖另外一个任务时，只有等待被依赖的任务完成后，依赖的任务才能算完成，这是一种可靠的任务序列。要么成功都成功，失败都失败，两个任务的状态可以保持一致。而异步是不需要等待被依赖的任务完成，只是通知被依赖的任务要完成什么工作，依赖的任务也立即执行，只要自己完成了整个任务就算完成了。至于被依赖的任务最终是否真正完成，依赖它的任务无法确定，所以它是不可靠的任务序列。我们可以用打电话和发短信来很好的比喻同步与异步操作。

在设计到 IO 处理时通常都会遇到一个是同步还是异步的处理方式的选择问题。因为同步与异步的 I/O 处理方式对调用者的影响很大，在数据库产品中都会遇到这个问题。因为 I/O 操作通常是一个非常耗时的操作，在一个任务序列中 I/O 通常都是性能瓶颈。但是同步与异步的处理方式对程序的可靠性影响非常大，同步能够保证程序的可靠性，而异步可以提升程序的性能，必须在可靠性和性能之间做个平衡，没有完美的解决办法。

**5.2.2 阻塞与非阻塞**

阻塞与非阻塞主要是从 CPU 的消耗上来说的，阻塞就是 CPU 停下来等待一个慢的操作完成 CPU 才接着完成其它的事。非阻塞就是在这个慢的操作在执行时 CPU 去干其它别的事，等这个慢的操作完成时，CPU 再接着完成后续的操作。虽然表面上看非阻塞的方式可以明显的提高 CPU 的利用率，但是也带了另外一种后果就是系统的线程切换增加。增加的 CPU 使用时间能不能补偿系统的切换成本需要好好评估。

**5.2.3 两种的方式的组合**

组合的方式可以由四种，分别是：同步阻塞、同步非阻塞、异步阻塞、异步非阻塞，这四种方式都对 I/O 性能有影响。下面给出分析，并有一些常用的设计用例参考。

表 2. 四种组合方式 

|  |  |
| --- | --- |
| **组合方式** | **性能分析** |
| 同步阻塞 | 最常用的一种用法，使用也是最简单的，但是 I/O 性能一般很差，CPU 大部分在空闲状态。 |
| 同步非阻塞 | 提升 I/O 性能的常用手段，就是将 I/O 的阻塞改成非阻塞方式，尤其在网络 I/O 是长连接，同时传输数据也不是很多的情况下，提升性能非常有效。  这种方式通常能提升 I/O 性能，但是会增加 CPU 消耗，要考虑增加的 I/O 性能能不能补偿 CPU 的消耗，也就是系统的瓶颈是在 I/O 还是在 CPU 上。 |
| 异步阻塞 | 这种方式在分布式数据库中经常用到，例如在网一个分布式数据库中写一条记录，通常会有一份是同步阻塞的记录，而还有两至三份是备份记录会写到其它机器上，这些备份记录通常都是采用异步阻塞的方式写 I/O。  异步阻塞对网络 I/O 能够提升效率，尤其像上面这种同时写多份相同数据的情况。 |
| 异步非阻塞 | 这种组合方式用起来比较复杂，只有在一些非常复杂的分布式情况下使用，像集群之间的消息同步机制一般用这种 I/O 组合方式。如 Cassandra 的 Gossip 通信机制就是采用异步非阻塞的方式。  它适合同时要传多份相同的数据到集群中不同的机器，同时数据的传输量虽然不大，但是却非常频繁。这种网络 I/O 用这个方式性能能达到最高。 |

虽然异步和非阻塞能够提升 I/O 的性能，但是也会带来一些额外的性能成本，例如会增加线程数量从而增加 CPU 的消耗，同时也会导致程序设计的复杂度上升。如果设计的不合理的话反而会导致性能下降。在实际设计时要根据应用场景综合评估一下。

下面举一些 **异步和阻塞** 的操作实例：

在 Cassandra 中要查询数据通常会往多个数据节点发送查询命令，但是要检查每个节点返回数据的完整性，所以需要一个异步查询同步结果的应用场景，部分代码如下：

清单 3.异步查询同步结果 

**class** **AsyncResult** **implements** **IAsyncResult**{

**private** **byte**[] result\_;

**private** AtomicBoolean done\_ = **new** AtomicBoolean(false);

**private** Lock lock\_ = **new** ReentrantLock();

**private** Condition condition\_;

**private** **long** startTime\_;

**public** **AsyncResult**(){

condition\_ = lock\_.newCondition();// 创建一个锁

startTime\_ = System.currentTimeMillis();

}

/\*\*\* 检查需要的数据是否已经返回，如果没有返回阻塞 \*/

**public** **byte**[] **get**(){

lock\_.**lock**();

**try**{

**if** (!done\_.**get**()){condition\_.**await**();}

}**catch** (InterruptedException ex){

**throw** **new** AssertionError(ex);

}**finally**{lock\_.unlock();}

**return** result\_;

}

/\*\*\* 检查需要的数据是否已经返回 \*/

**public** boolean **isDone**(){**return** done\_.**get**();}

/\*\*\* 检查在指定的时间内需要的数据是否已经返回，如果没有返回抛出超时异常 \*/

**public** **byte**[] **get**(**long** timeout, TimeUnit tu) throws TimeoutException{

lock\_.**lock**();

**try**{ boolean bVal = true;

**try**{

**if** ( !done\_.**get**() ){

**long** overall\_timeout = timeout - (System.currentTimeMillis() - startTime\_);

**if**(overall\_timeout > 0)// 设置等待超时的时间

bVal = condition\_.**await**(overall\_timeout, TimeUnit.MILLISECONDS);

**else** bVal = false;

}

}**catch** (InterruptedException ex){

**throw** **new** AssertionError(ex);

}

**if** ( !bVal && !done\_.**get**() ){// 抛出超时异常

**throw** **new** TimeoutException("Operation timed out.");

}

}**finally**{lock\_.unlock(); }

**return** result\_;

}

/\*\*\* 该函数拱另外一个线程设置要返回的数据，并唤醒在阻塞的线程 \*/

**public** **void** **result**(Message response){

**try**{

lock\_.**lock**();

**if** ( !done\_.**get**() ){

result\_ = response.getMessageBody();// 设置返回的数据

done\_.**set**(true);

condition\_.signal();// 唤醒阻塞的线程

}

}**finally**{lock\_.unlock();}

}

}

参考：   
Java IO流分析整理   
http://blog.csdn.net/llhhyy1989/article/details/7388059   
JDK API , java io 札记整理-从模式看JDK IO   
http://www.myexception.cn/program/842855.html   
深入分析 Java I/O 的工作机制   
https://www.ibm.com/developerworks/cn/java/j-lo-javaio/   
深入分析 Java 中的中文编码问题   
http://www.ibm.com/developerworks/cn/java/j-lo-chinesecoding/   
java中的IO整理   
http://blog.csdn.net/a511596982/article/details/8284358   
Java IO学习笔记：概念原理   
http://lavasoft.blog.51cto.com/62575/95384