Java NIO里面一个主要的类是Selector，这个类似一个观察者，只要在写程序过程把需要探知的socketchannel告诉Selector，我们就可以直接做其他的事情，当有事件发生的时候，他会通知我们，传回一个SelectionKey，读取这些Key就可以获取刚刚注册过的SocketChannel，然后就可以从这个Channel中读取所需要的数据，它保证我们在读写过程能够读取到这些数据，接着还可以针对这些数据进行处理。Selector内部原理其实很简单，就是针对一个注册过的Channel进行轮询访问，不断的轮询一旦轮询到一个Channel有所注册的事情发生，比如数据来了等就会发送报告，交出一个Key，让程序通过该Key直接读取这个 Channel的内容。

　　【\*：关于Java NIO我们在编程过程可以这样理解，实际上Java NIO的大部分内容都不是Java语言本身实现的，它之所以效率比较高是因为它本身使用的就是操作系统底层的IO部分的API而不是基于JVM级别的，而 它的实现 语言使用的是C，所以它在读写过程和操作系统紧密相关，而利用上边讲解的原理就可以完整实现Java的高效读取，这种方 式和我们前边遇到的Java IO的方式有本质上的区别，而如何使用Java书写接口而换用C语言书写 实现读者可以参考JNI的有关文档。】

　　接下来提供几个例子来说明Java NIO的API的相关用法：

——[$]使用映射文件来读取文本文件——

import java.io.FileInputStream;

import java.io.IOException;

import java.nio.MappedByteBuffer;

import java.nio.channels.FileChannel;

public class NativeMapFile {

public static void main(String args[]){

FileInputStream fileInputStream;

FileChannel fileChannel;

long fileSize;

MappedByteBuffer mBuffer;

try{

fileInputStream = new FileInputStream("D:/work/test.txt");

fileChannel = fileInputStream.getChannel();

fileSize = fileChannel.size();

mBuffer = fileChannel.map(FileChannel.MapMode.READ\_ONLY, 0, fileSize);

for( int i = 0; i < fileSize; i++ ){

System.out.print((char)mBuffer.get());

}

fileChannel.close();

fileInputStream.close();

}catch(IOException ex){

ex.printStackTrace();

}

}

}

可以知道一点就是直接使用这种方式读取文件，有可能会出现中文的乱码问题，不过这个问题稍候想办法 解决，这里先介绍几个NIO里面的几个重要知识点：FileChannel.MapMode是文件影射模型的安全的枚 举，主要有几个值：

FileChannel.MapMode.PRIVATE：专用（写入时拷贝）映射模式

FileChannel.MapMode.READ\_ONLY：只读映射模式

FileChannel.MapMode.READ\_WRITE：读取、写入映射模式

　　其实java.nio里面的一个核心类就是上边使用到的FileChannel，该类的介绍如 下：

　　定义：

public abstract class FileChannel extends AbstractInterruptibleChannel implements ByteChannel, GatheringByteChannel, ScatteringByteChannel

　　该类是用于读取、写入、映射和操作文件的通道，文件通道在文件内部有一个当前的position， 可以对其进行查询和修改，该文件本身包含一个可读写的长度的可变字节序列，并且可以通过查询该文件的当前大小，写入字节超出文件的当前大小时，则增加文件 的大小；截取该文件的 时候，则减小文件的大小，文件可能还有某个相关联的元数据，如访问权限、内容类型和最后的修改时间，该类却没有定义元数据的方法。除了字节 通道中常见的读取、写入和关闭操作，此类还定义了下列特定于文件的操作：

\* 以不影响通道当前位置的方式，对文件中绝对位置的字节进行读取和写入

\* 将文件中的某个区域直接映射到内容中，对于较大的文件，这通常比调用普通的read或write方 式更加高效

\* 强制对底层存储设备进行文件更新，以确保在系统崩溃的时候不丢失数据

\* 以一种可以被很多操作系统优化为直接向文件系统缓存发送或从中读取的高速传输方法，将字节从文件传 输到某个其他通道中，反之亦然

\* 可以锁定某个文件区域，以阻止其他程序进行访问

　　多个并发线程可安全地使用文件通道，可随时调用关闭方法， 正如Channel接口中所指定，对于涉及通道位置或者可以更改文件大小的操作，在任意给定时间只能进行一个这样的操作， 如果尝试在第一操作仍在进行时发起第二个操作，则会导致在第一个操作完成之前阻塞第二个操作。可以并发处理其他操作， 特别是采用显示位置的操作，但是否并发处理则取决于系统，因此是未指定的。确保此类的实例所提供的文件视图与同一程序中其他实例所提供的相同文件视图是 一致的。但是，此类的实例所提供的视图不一定与其他并发运行的程序所看到的视图一致，这取决于底层操作系统所执行的缓冲策略和各种网络文件系统协议所引入的延迟。不 管其他程序是以何种语言编写的，而且也不管是运行在相同机器还是不同机器上都是如 此。此种不一致的确切性质取决于系统，因此是未指定的。此类没有定义打开现有文件或创建新文件的方法，以后的版本中可能添加这些方法。在此版本中，可从现有的 FileInputStream、 FileOutputStream 或 RandomAccessFile 对象获得文件通道，方法是调用该对象的 getChannel 方法，这会返回一个连接到相同底层文件的文件通道。文件通道的状态与其 getChannel 返回该通道的对象密切相关。显式或者 通过读取或写入字节来更改通道的位置将更改发起对象的文件位置，反之亦然。通过文件通道更改此文件的长度将更改通过发起对象看到的长度，反之亦然。通过写 入字节更改此文件的内容将更改发起对象所看到的内容，反之亦然。此类在各种情况下指定要求“允许读取操作”、“允许写入操作”或“允许读取和写入操作”的 某个实例。通过 FileInputStream 实例的 getChannel 方法所获得的通道将允许进行读取操作。通过 FileOutputStream 实例的 getChannel 方法所获得的通道将允许进行写入操作。最后，如果使用模式 "r" 创建 RandomAccessFile 实例，则通过该实例的 getChannel 方法所获得的通道将允许进行读取操作，如果使用模式 "rw" 创建实例，则获得的通道将允许进行读取和写入操作。如果从文件输出流中获得了允许进行写入操作 的文件通道，并且该输出流是通过调用 FileOutputStream(File,boolean) 构造方法且为第二个参数传入 true 来创建的，则该文件通道可能处于添加模式。 在此模式中，每次调用相关的写入操作都会首先将位置移到文件的末尾，然后再写入请求的数据。在单个原子操作中是否移动位置和写入数据是与系统相关的，因此 是未指定的。

　　【思考\*：使用FileChannel的时候主要是针对字节序列进行操作，那么和前面提及 到的Java普通IO交互的部分大部分都是字节流读写器，参考字节流读写器中间可以发现一部分Stream类型的读写器可以通过getChannel方法 获取文件通道。】

　　下边再引入一个FileChannel的例子：

　　——[$]写入文件，使用新的IO——

import java.io.FileOutputStream;

import java.io.IOException;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;

public class NativeFileWrite {

public static void main(String args[]){

FileOutputStream fileOutputStream;

FileChannel fileChannel;

ByteBuffer byteBuffer;

try{

fileOutputStream = new FileOutputStream("D:/work/test.txt");

fileChannel = fileOutputStream.getChannel();

byteBuffer = ByteBuffer.allocateDirect(26);

for( int i = 0; i < 26; i++ )

byteBuffer.put((byte)('A' + i));

byteBuffer.rewind();

fileChannel.write(byteBuffer);

fileChannel.close();

fileOutputStream.close();

}catch(IOException ex){

ex.printStackTrace();

}

}

}

该例子演示了向一个文件内写入英文字母A到Z的全过程，下边是写入过后test.txt里面的文 件内容：

ABCDEFGHIJKLMNOPQRSTUVWXYZ

　　这里再介绍一个类：ByteBuffer类，该类针对字节进行了六种操作：

\* 读写单个字节的绝对和相对get和put方法

\* 将此缓冲区的连续字节列传输到数组中的相对批量get方法

\* 将byte数组或其他字节缓冲区中的连续字节序列传输到此缓冲区的相对批量put方法

\* 读写其他基本类型值，并按照特定的字节序列在字节序列之间转换这些值的get和put方法

\* 创建视图缓冲区的方法，这些方法允许将字节缓冲区视为包含其他基本类型值的缓冲区

\* 对字节缓冲区进行compacting、duplicating和slicing的方法