**中软国际考题-笔试**

1. 单选题

1、以下说法错误的是（B）

A. IO是面向流的，NIO是面向缓冲区的

B. NIO的N可以理解为Non-blocking,一旦执行不会被阻止。

C. IO流是阻塞的，NIO流是不阻塞的

D. NIO有选择器，而IO没有

2、下列哪个不是Java 的IO类型（D）

A.字节流 B.字符流 C.输入流 D.异常流

3、下列关于NIO说法正确的是（A）

A.IO是面向流的，NIO是面向缓冲区的

B.NIO是阻塞的

C.NIO是JDK1.6之后的版本才出现的

D.NIO中的Selector只能在单线程处理1个 Channel

4、以下不属于NIO中主要的通道类型是（A）

A.DirectorChannel B.DatagramChannel C.SocketChannel D.ServerSocketChannel

5、以下关于NIO说法不正确的有（D）

1. 通过调用Selector.open()方法创建一个Selector
2. Channel表示一个数据流，从中可以读取数据
3. Buffer的关键实现包括ByteBuffer、CharBuffer、DoubleBuffer等
4. 要使用Selector不需要注册Channel，Selector会自动根据类型获取到相应的Channel

6、下列关于NIO的描述中，正确的是那个选项（B）

1. NIO是面向流的。
2. NIO的事件选择器允许一个单独的线程来监视多个输入通道。
3. NIO能前后移动流中的数据。
4. NIO使用多个线程来解决传统的阻塞问题。

7、下列关于Java NIO的描述，哪个是正确的（ B ）

1. Java NIO 是面向流的，并且是非阻塞的 IO
2. Java NIO 是面向缓冲的，并且是非阻塞的 IO
3. Java NIO 是面向缓冲的，并且是阻塞的 IO
4. Java NIO 是面向流的，并且是阻塞的 IO

8、以下哪个类对象是实现NIO非阻塞网络通信的核心（D）

A. SocketChannel B.Socket C. ServerSocketChannel D. Selector

9、下列哪个不是NIO包的新特点？（D）

A.内存映射技术 B.文件锁定

C.字符及编码 D.阻塞I/O

10、下列关于NIO的描述，其中错误的是（D）

A.NIO是异步时间驱动的网络应用框架，对于快速开发高性能的服务器提供保证。

B.NIO数据可以从Channel读到Buffer中，也可以从Buffer写到Channel中，一个Channel常对应一个Connection。

C.NIO是短连接的单线程。

D.NIO适合大文件传输。

11、为了提高读写性能，可以采用什么流？（D）

A.InputStream

B.DataInputStream

C.BufferedReader

D.BufferedInputStream

1. 多选题
2. 下面属于NIO中Buffer（缓冲区）对象的常用方法的是？（A、C、D）
3. clear B.submit C. flip
4. rewind; E. refreshCache

2、NIO和IO如何影响应用程序的设计？（A、B）

A.对NIO或IO类的API调用。 B.数据处理。 C.用来处理数据的线程数

1. 下面哪几个是NIO的核心部分？（A、C、D）

A.Channels B. Connection; C. Buffers

D. Selectors E. Stream

4、NIO主要有以下哪些缓冲区类型？（A、B、C、D）

A. ByteBuffer B. CharBuffer C. IntBuffer D. DoubleBuffer

5、以下对同步与阻塞说明正确的有（A、B）

A同步：方法调用在没得到结果之前，没有调用结果，不返回任何结果。  
B异步：方法调用在没得到结果之前，没有调用结果，返回状态信息。  
C阻塞：方法调用请求发送后自动关闭

D非阻塞：方法调用请求发送后，等待结果返回。

6、下面关于NIO的描述，哪些选项是不正确的？（A、B）

A. NIO是面向流的。

B. NIO使用多个线程来解决传统的阻塞问题。

C. NIO的一个事件选择器里可以注册多个数据传输通道。

D. NIO是以非阻塞的方式来读取数据

7、 下列关于Java NIO的描述，哪些是正确的？（A、B、C、E）

A. Java NIO 提供了文件锁。

B. Java NIO 提供了获得当前 JDK 支持的所有字符集的方法。

C. Java NIO 是面向缓冲的，并且是非阻塞的 IO。

D. Java NIO 的选择器必须使用多个线程来分别监视对应的输入通道。

E. Java NIO 的选择器允许使用一个单独的线程来监视多个输入通道。

8、NIO可用于以下哪些操作（ACD）

A.文件操作 B.数据库操作 C.网络通信 D.线程间通信

9、 NIO中主要有以下哪些通道类型：（A B C D）

A.FileChannel - 文件IO

B.DatagramChannel - UDP传输

C.SocketChannel - TCP传输

D.ServerSocketChannel -TCP传输

1. 填空题（每空1分，共10分）
2. 在NIO中负责从一个Channel读取的信息分散到N个缓冲区中(Buffer)的对象是\_\_\_\_\_\_\_\_\_\_对象。(Scatter)
3. 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\_\_\_\_\_，所以NIO的效率要比IO\_\_\_\_\_\_。 (块、高很多)
4. Buffer的使用方式：分配空间的方法名称是\_\_\_\_\_\_\_\_\_\_ ，读取Buffer的方法名称是\_\_\_\_\_\_\_\_\_，清除Buffer的方法名称是\_\_\_\_\_\_\_\_\_\_。(allocate、get、clear)
5. Java NIO (New IO)是JDK1.4后引入的新输入/输出API，主要由以下三个核心部分组成\_\_\_\_\_\_\_，\_\_\_\_\_\_\_和\_\_\_\_\_\_\_。(Channel，Buffer，Selector)

5、NIO的channel类型有 FileChannel 、 DatagramChannel、SocketChannel、ServerSocketChannel 等

6、在Java NIO中由于有了\_\_\_\_\_\_，因此NIO可以以\_\_\_\_\_\_的方式读取数据。(事件选择器、非阻塞)

7、Java 7中的 NIO 主要有如下三个核心部分，它们分别是**\_\_\_\_\_\_\_\_\_\_\_\_\_、\_\_\_\_\_\_\_\_\_\_\_\_\_、\_\_\_\_\_\_\_\_\_\_\_\_\_，**提供了两种文件锁，分别是**\_\_\_\_\_\_\_\_\_\_\_\_\_、\_\_\_\_\_\_\_\_\_\_\_\_\_。( Channel、Buffer、Selector(Java 7) 、共享锁、排它锁)**

8、将通道注册到选择器时，选择器对通道可以设置的监听事件有\_\_\_\_、\_\_\_\_、\_\_\_\_、\_\_\_\_\_四种。

答案：读、写、连接、接收

9、NIO 由哪几个核心部分组成：\_\_\_\_\_\_\_\_\_\_，\_\_\_\_\_\_\_\_\_\_，\_\_\_\_\_\_\_\_\_\_。（Channel 通道、Buffer 缓冲区、Selector 选择器）

1. 判断题
2. 在NIO中Selector对象用于检查一个或多个NIO Channel（通道）的状态是否处于可读、可写。（ T ）
3. Java中 NIO的Buffer对象用于和NIO Channel交互，通过Buffer把数据写入到Channels（T）
4. NIO可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的代价是解析数据可能会比从一个阻塞流中读取数据更复杂。（ T ）
5. IO中的Stream和NIO的channels都是单向的。（ F ）
6. Selector(选择器)可以监听多个通道的事件。 （ T ）
7. 调用Buffer.mark()方法，可以标记Buffer中的一个特定的position，表示读取的位置（ T ）
8. 当向Selector注册Channel时，Channel.register()方法会返回多个SelectionKey 对象（ F ）
9. Buffer中几个常用属性大小关系是mark<=position<=limit<=capacity。 （ T ）
10. 通过ServerSocketChannel类的静态方法getConnection()获得ServerSocketChannel对象。 （ F ）
11. 阻塞模式也可以使用Selector （ F ）
12. 选择器Selector支持单个线程处理多个Channel，将多个Channel注册到一个选择器中，选择器基于事件的方式处理。（ T ）
13. 写数据到Buffer只有从Channel写到Buffer一种方式。（ F ）
14. Java NIO系统与IO的主要区别是API不同，实现IO的方式基本相同（F）
15. NIO可以使用非阻塞的方式来读取数据。（ T ）
16. NIO的一个事件选择器里可以注册多个数据传输通道。（ T ）
17. NIO能前后移动流中的数据。（ F ）
18. NIO和IO一样都是面向流的。（ F ）
19. NIO使用事件循环的方式读取数据而不是传统的阻塞方式读取数据。（ T ）
20. Java NIO 是面向流的，并且是非阻塞的 IO。（ F ）
21. Java NIO 是面向缓冲的，并且是阻塞的 IO。（ F ）
22. Java NIO 是面向流的，并且是阻塞的 IO。（ F ）
23. Java NIO 的选择器必须使用多个线程来分别监视对应的输入通道。（ F ）
24. Java NIO 的选择器允许使用一个单独的线程来监视多个输入通道。（ T ）
25. Java NIO 没有提供文件锁的功能。（ F ）
26. Java NIO 只提供了排它锁而没有提供共享锁。（ F ）
27. 非阻塞式网络通信比阻塞式网络通信效率高。（T）
28. 在 NIO 库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的。在写入数据时，它是写入到缓冲区中的。任何时候访问 NIO 中的数据，您都是将它放到缓冲区中。（T）

五、简答题

1、请简述多路复用IO模型是什么？

答案：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。

2、什么是同步IO？什么是异步IO？

答案：

同步IO：当用户发出IO请求操作之后，如果数据没有就绪，需要通过用户线程或者内核不断地去轮询数据是否就绪，当数据就绪时，再将数据从内核拷贝到用户线程；

异步IO：只有IO请求操作的发出是由用户线程来进行的，IO操作的两个阶段都是由内核自动完成，然后发送通知告知用户线程IO操作已经完成。也就是说在异步IO中，不会对用户线程产生任何阻塞。

3、NIO和IO的主要区别？（5分）

答案：（1）面向流与面向缓冲

Java IO和NIO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。

（2）阻塞与非阻塞IO

Java 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。Java 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）。

（3）选择器（Selectors）

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。。

4、NIO中， 如果不显式的调用System.gc()那会出现什么问题？。(意思答对即可)

答案：如果DirectByteBuffer的空间够用，那么System.gc()是不会触发FullGC的。也就是说在空间不够用时，显示调用才能进行回收，如果不显式调用，那只能是抛出内存异常了。

5、NioSocket中服务端的处理过程分为哪几个步骤(意思答对即可)

答案：

1、创建ServerScoketChannel对象并设置相关参数（绑定监听端口号，是否使用阻塞模式）

2、创建Selector并注册到服务端套接字信道（ServerScoketChannel）上

3、使用Selector的select方法等待请求

4、接收到请求后使用selectedKeys方法获得selectionKey集合

5、根据选择键获得Channel、Selector和操作类型进行具体处理。

6 普通Socket和NioSocket的区别：

答案：

普通Socket是客户端发出一次请求、服务端接收到后响应、客户端接收到服务端的响应才能再次请求。

NioSocket是引入了三个概念：Channel、Selector、Buffer。Buffer是将很多请求打包，一次性发出去，有Selector扮演选择器的角色，将请求分转给对应的通道（Channel）进行处理响应。

7、什么是NIO？

答案：Java NIO (New IO)是JDK1.4后引入的新输入/输出API,提供基于缓冲区(buffer)的块写入/读取，而以前的I/O是基于流(Stream)的方式，NIO基于块的IO操作，将最耗时的缓存区读取和填充交由底层操作系统实现，因此速度上要快得多。

8、一个简单（标准）的NIO输入输出一般包含如下步骤?

答案：1. 从数据源获取通道

2. 分配缓冲区

3. 切换缓存区为写模式

4. 从通道读取数据写入缓冲区

5. 切换缓冲区为读模式

6. 缓冲区数据写入通道中

7. 关闭资源

9、试说明Java NIO 的通道 ( Channel ) 相对于流 ( Stream ) 的特点。

答案：

Java NIO的通道类似流，但又有些不同：

1）既可以从通道中读取数据，又可以写数据到通道。但流的读写通常是单向的。

2）通道可以异步地读写。

3）通道中的数据总是要先读到一个 Buffer，或者总是要从一个 Buffer中 写入。

10、非阻塞式网络通信和阻塞式网络通信的区别是什么？

传统的 IO 流都是阻塞式的。也就是说，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写入，该线程在此期间不能执行其他任务。因此，在完成网络通信进行 IO 操作时，由于线程会阻塞，所以服务器端必须为每个客户端都提供一个独立的线程进行处理，当服务器端需要处理大量客户端时，性能急剧下降。

Java NIO 是非阻塞模式的。当线程从某通道进行读写数据时，若没有数据可用时，该线程可以进行其他任务。线程通常将非阻塞 IO 的空闲时间用于在其他通道上执行 IO 操作，所以单独的线程可以管理多个输入和输出通道。因此， NIO 可以让服务器端使用一个或有限几个线程来同时处理连接到服务器端的所有客户端。

六、程序实现题

1、NIO文件读取：请编写程序使用NIO的接口读取指定位置的文件内容（可以不考虑文件编码格式），文件为txt后缀。

|  |
| --- |
| package test;  import java.io.IOException;  import java.io.RandomAccessFile;  import java.nio.ByteBuffer;  import java.nio.channels.FileChannel;  public class NIOTest {  public static void main(String[] args) {  RandomAccessFile aFile = null;  try {  aFile = new RandomAccessFile("D://aa.txt", "rw");  FileChannel fileChannel = aFile.getChannel();  ByteBuffer buf = ByteBuffer.allocate(1024);  int bytesRead = fileChannel.read(buf);  System.out.println(bytesRead);  while (bytesRead != -1) {  buf.flip();  while (buf.hasRemaining()) {  System.out.print((char) buf.get());  }  buf.compact();  bytesRead = fileChannel.read(buf);  }  } catch (IOException e) {  e.printStackTrace();  } finally {  try {  if (aFile != null) {  aFile.close();  }  } catch (IOException e) {  e.printStackTrace();  }  }  }  } |

2、用IO和NIO两种方式实现文件拷贝。（10分）

|  |
| --- |
| import java.io.FileInputStream;  import java.io.FileOutputStream;  import java.io.IOException;  import java.io.InputStream;  import java.io.OutputStream;  import java.nio.ByteBuffer;  import java.nio.channels.FileChannel;    public class MyUtil {    private MyUtil() {  throw new AssertionError();  }    public static void fileCopy(String source, String target) throws IOException {  try (InputStream in = new FileInputStream(source)) {  try (OutputStream out = new FileOutputStream(target)) {  byte[] buffer = new byte[4096];  int bytesToRead;  while((bytesToRead = in.read(buffer)) != -1) {  out.write(buffer, 0, bytesToRead);  }  }  }  }    public static void fileCopyNIO(String source, String target) throws IOException {  try (FileInputStream in = new FileInputStream(source)) {  try (FileOutputStream out = new FileOutputStream(target)) {  FileChannel inChannel = in.getChannel();  FileChannel outChannel = out.getChannel();  ByteBuffer buffer = ByteBuffer.allocate(4096);  while(inChannel.read(buffer) != -1) {  buffer.flip();  outChannel.write(buffer);  buffer.clear();  }  }  }  }  } |

3、使用NIO实现文件的复制，将一个路径下的文件复制到指定的位置。

（10分）

|  |
| --- |
| package com.cloudcore.usercenter.consumer.test;    import java.io.File;  import java.io.FileInputStream;  import java.io.FileOutputStream;  import java.io.IOException;  import java.nio.ByteBuffer;  import java.nio.channels.FileChannel;    public class NioFileReadAndWrite {    public static void main(String[] args) {  File from = new File("d:\\source.pdf");  File to = new File("d:\\destion.pdf");  try {  nioCopyFile(from, to);  } catch (IOException e) {  e.printStackTrace();  }  }    /\*\*  \* nio实现文件复制  \*  \* @param from  \* @param to  \* @throws IOException  \*/  public static final void nioCopyFile(File from, File to) throws IOException {  FileInputStream fin = null;  FileOutputStream fout = null;  FileChannel fic = null;  FileChannel foc = null;  try {  fin = new FileInputStream(from);  fout = new FileOutputStream(to);  fic = fin.getChannel();  foc = fout.getChannel();  ByteBuffer bb = ByteBuffer.allocate(1024 << 4);  while (fic.read(bb) > 0) {  bb.flip();  foc.write(bb);  bb.clear();  }  } finally {  if (null != fic)  fic.close();  if (null != foc)  foc.close();  if (null != fin)  fin.close();  if (null != fout)  fout.close();  }  }  } |

4、实现简单（标准）的NIO输入输出

|  |
| --- |
| RandomAccessFile formFile = new RandomAccessFile("src\\a.txt", "rw");  RandomAccessFile toFile = new RandomAccessFile("src\\b.txt", "rw");  //获取channel  FileChannel fromChannel = formFile.getChannel();  FileChannel toChannel = toFile.getChannel();  // 定义缓冲大小  int bufSize = 1024\*4;  // 定义缓冲  ByteBuffer byteBuffer = ByteBuffer.allocate(bufSize);  int len = 0;  // 将数据从源channel写入到缓冲区  while( (len=fromChannel.read(byteBuffer)) !=-1 ){  //切换到读模式  byteBuffer.flip();  //读取缓冲区数据写到目标channel  toChannel.write(byteBuffer);  // 清空缓冲  byteBuffer.clear();  }  // 释放资源  toChannel.close();  fromChannel.close(); |

5、使用NIO实现一个 server与client通信的简单例子

（10分）

|  |
| --- |
| package cn.com.testnio.base;  import java.io.IOException;  import java.net.InetSocketAddress;  import java.nio.ByteBuffer;  import java.nio.channels.SelectionKey;  import java.nio.channels.Selector;  import java.nio.channels.ServerSocketChannel;  import java.nio.channels.SocketChannel;  import java.util.Iterator;  import java.util.Map;  import java.util.Set;  import java.util.concurrent.ConcurrentHashMap;  /\*\*  \* NIO是面向缓冲区的  \* User: lihaoquan  \*/  public class Server {  private Selector selector;  /\*\*  \* 值得注意的是 Buffer 及其子类都不是线程安全的。  \*/  private ByteBuffer readBuffer = ByteBuffer.allocate(1024);//设置缓冲区大小  /\*\*  \* ConcurrentHashMap是支持并发的,所以这里不使用HashMap  \*/  private Map<SocketChannel,byte[]> clientMessage  = new ConcurrentHashMap<SocketChannel,byte[]>();  /\*\*  \* 启动开关  \* @throws IOException  \*/  public void start() throws IOException {  //通过静态工厂生产服务端的channel  ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();  //设置非阻塞方式  serverSocketChannel.configureBlocking(false);  //绑定IP  serverSocketChannel.socket().bind(new InetSocketAddress(8080));  //创建选择器  selector = Selector.open();  //注册监听事件  serverSocketChannel.register(selector, SelectionKey.OP\_ACCEPT);  while (!Thread.currentThread().isInterrupted()) {  selector.select();  Set<SelectionKey> keys = selector.selectedKeys();  Iterator<SelectionKey> keyIterator = keys.iterator();  while (keyIterator.hasNext()) {  SelectionKey key = keyIterator.next();  if(!key.isValid()) {  continue;  }  if(key.isAcceptable()) {  accept(key);  }else if(key.isReadable()) {  read(key);  }  //去除本次keyIterator.next()的对象,但不会对下次遍历有影响  keyIterator.remove();  }  }  }  /\*\*  \* 读入  \* @param key  \* @throws IOException  \*/  public void read(SelectionKey key) throws IOException{  SocketChannel socketChannel = (SocketChannel) key.channel();  this.readBuffer.clear();  int numRead;  try {  /\*\*  \* 获取客户点的read操作读入数据块数量  \*/  numRead = socketChannel.read(readBuffer);  }catch (Exception e) {  e.printStackTrace();  key.cancel();  socketChannel.close();  clientMessage.remove(socketChannel);  return;  }  byte[] bytes = clientMessage.get(socketChannel);  if(bytes == null) {  bytes = new byte[0];  }  if(numRead > 0) {  byte[] newBytes =new byte[bytes.length+numRead];  System.arraycopy(bytes,0,newBytes,0,bytes.length);  System.arraycopy(readBuffer.array(), 0, newBytes, bytes.length, numRead);  clientMessage.put(socketChannel, newBytes);  System.out.println("message>>>"+new String(newBytes));  }else {  String message = new String(bytes);  System.out.println("data>>>"+message);  }  }  /\*\*  \* 接收  \* @param key  \* @throws IOException  \*/  public void accept(SelectionKey key) throws IOException {  ServerSocketChannel serverSocketChannel  = (ServerSocketChannel) key.channel();  SocketChannel clientChannel = serverSocketChannel.accept();  clientChannel.configureBlocking(false);  clientChannel.register(selector,SelectionKey.OP\_READ);  System.out.println("已成功连接新的客户端......");  }  public static void main(String[] args) throws Exception{  new Server().start();  }  }  //客户端  package cn.com.testnio.base;  import java.io.IOException;  import java.net.InetSocketAddress;  import java.nio.ByteBuffer;  import java.nio.channels.SelectionKey;  import java.nio.channels.Selector;  import java.nio.channels.SocketChannel;  import java.util.Iterator;  import java.util.Scanner;  import java.util.Set;  /\*\*  \* User: lihaoquan  \*/  public class Client {  /\*\*  \* 启动开关  \* @throws IOException  \*/  public void start() throws IOException {  //通过静态工厂生产客户端的channnel  SocketChannel socketChannel = SocketChannel.open();  //设置客户端请求为非阻塞方式  socketChannel.configureBlocking(false);  //绑定IP  socketChannel.connect(new InetSocketAddress(8080));  //创建选择器  Selector selector = Selector.open();  //注册监听事件  socketChannel.register(selector, SelectionKey.OP\_CONNECT);  //键盘输入  Scanner scanner = new Scanner(System.in);  while (true) {  selector.select();  Set<SelectionKey> keys = selector.selectedKeys();  Iterator<SelectionKey> keyIterator = keys.iterator();  while (keyIterator.hasNext()) {  SelectionKey key = keyIterator.next();  //去除本次keyIterator.next()的对象,但不会对下次遍历有影响  keyIterator.remove();  if(key.isConnectable()) {  socketChannel.finishConnect();  socketChannel.register(selector,SelectionKey.OP\_WRITE);  System.out.println("客户端已经连上服务器端!!");  break;  }else if(key.isWritable()) {  System.out.println("请输入信息!!");  String message = scanner.nextLine();  ByteBuffer writeBuffer = ByteBuffer.wrap(message.getBytes());  socketChannel.write(writeBuffer);  }  }  }  }  public static void main(String[] args) throws Exception{  new Client().start();  }  } |

7、请基于NIO技术分别编写客户端NIOClient与服务器端NIOServer，并实现两者之间的数据通信：（10分）

|  |
| --- |
| NIOClient.java 示例代码：  import java.io.IOException;  import java.net.InetSocketAddress;  import java.nio.ByteBuffer;  import java.nio.channels.SelectionKey;  import java.nio.channels.Selector;  import java.nio.channels.ServerSocketChannel;  import java.nio.channels.SocketChannel;  import java.util.Iterator;  public class NIOClient {  // 管道管理器  private Selector selector;  public NIOClient init(String serverIp, int port) throws IOException {  // 获取socket通道  SocketChannel channel = SocketChannel.open();  channel.configureBlocking(false);  // 获得通道管理器  selector = Selector.open();  // 客户端连接服务器，需要调用channel.finishConnect()方法连接。  channel.connect(new InetSocketAddress(serverIp, port));  // 为该通道注册SelectionKey.OP\_CONNECT事件  channel.register(selector, SelectionKey.OP\_CONNECT);  return this;  }  public void listen() throws IOException {  System.out.println("客户端启动");  // 轮询访问selector  while (true) {  // 选择注册过的io操作的事件(第一次为SelectionKey.OP\_CONNECT)  selector.select();  Iterator<SelectionKey> ite = selector.selectedKeys().iterator();  while (ite.hasNext()) {  SelectionKey key = ite.next();  // 删除已选的key，防止重复处理  ite.remove();  if (key.isConnectable()) {  SocketChannel channel = (SocketChannel) key.channel();  // 如果正在连接，则完成连接  if (channel.isConnectionPending()) {  channel.finishConnect();  }  channel.configureBlocking(false);  // 向服务器发送消息  channel.write(ByteBuffer.wrap(  new String("我是客户端数据.").getBytes()));  // 连接成功后，注册接收服务器消息的事件  channel.register(selector, SelectionKey.OP\_READ);  System.out.println("客户端连接成功");  } else if (key.isReadable()) { // 有可读数据事件。  SocketChannel channel = (SocketChannel) key.channel();  ByteBuffer buffer = ByteBuffer.allocate(10);  channel.read(buffer);  byte[] data = buffer.array();  String message = new String(data);  System.out.println(  "从服务器端接收数据的大小是:" + buffer.position() +  " 从服务器端接收数据的内容是: " + message);  }  }  }  }  public static void main(String[] args) throws IOException {  new NIOClient().init("127.0.0.1", 9981).listen();  }  }  NIOServer.java 示例代码：  import java.io.IOException;  import java.net.InetSocketAddress;  import java.nio.ByteBuffer;  import java.nio.channels.SelectionKey;  import java.nio.channels.Selector;  import java.nio.channels.ServerSocketChannel;  import java.nio.channels.SocketChannel;  import java.util.Iterator;  public class NIOServer {  // 通道管理器  private Selector selector;  // 获取一个ServerSocket通道，并初始化通道  public NIOServer init(int port) throws IOException {  // 获取一个ServerSocket通道  ServerSocketChannel serverChannel = ServerSocketChannel.open();  serverChannel.configureBlocking(false);  serverChannel.socket().bind(new InetSocketAddress(port));  // 获取通道管理器  selector = Selector.open();  // 将通道管理器与通道绑定，通道注册SelectionKey.OP\_ACCEPT事件，  // 只有当该事件到达时，Selector.select()会返回，否则一直阻塞。  serverChannel.register(selector, SelectionKey.OP\_ACCEPT);  return this;  }  public void listen() throws IOException {  System.out.println("服务器端启动成功");  // 使用轮询访问selector  while (true) {  // 当有注册的事件到达时，方法返回，否则阻塞。  selector.select();  // 获取selector中的迭代器，选中项为注册的事件  Iterator<SelectionKey> ite = selector.selectedKeys().iterator();  while (ite.hasNext()) {  SelectionKey key = ite.next();  // 删除已选key，防止重复处理  ite.remove();  // 客户端请求连接事件  if (key.isAcceptable()) {  ServerSocketChannel server =  (ServerSocketChannel) key.channel();  // 获得客户端连接通道  SocketChannel channel = server.accept();  channel.configureBlocking(false);  // 向客户端发消息  channel.write(ByteBuffer.wrap(  new String("向客户端发送信息").getBytes()));  // 为客户端通道注册SelectionKey.OP\_READ事件。  channel.register(selector, SelectionKey.OP\_READ);  System.out.println("客户端请求连接事件");  } else if (key.isReadable()) {// 有可读数据事件  // 获取客户端传输数据可读取消息通道。  SocketChannel channel = (SocketChannel) key.channel();  // 创建读取数据缓冲器  ByteBuffer buffer = ByteBuffer.allocate(10);  int read = channel.read(buffer);  byte[] data = buffer.array();  String message = new String(data);  System.out.println(  "从客户端接收的信息大小是:" + buffer.position() +  "从客户端接收的信息是: " + message);  }  }  }  }  public static void main(String[] args) throws IOException {  new NIOServer().init(9981).listen();  }  }  编译成功后，在D:/abc/下生成Transfer.class文件，运行java Transfer |

8、1、试编写程序解决下列问题（使用Java NIO）：

1. 列出 Java 支持的全部字符集
2. 将1）的结果写入到一个CSV文件中

CharsetsDemo.java

|  |
| --- |
| **import** java.io.IOException;  **import** java.nio.charset.Charset;  **import** java.nio.file.Files;  **import** java.nio.file.Paths;  **import** java.util.ArrayList;  **import** java.util.List;  **import** java.util.SortedMap;  **public** **class** CharsetsDemo {  **public** **static** **void** main(String[] args) **throws** IOException {  SortedMap<String, Charset> mapCharset = Charset.*availableCharsets*();  List<String> contents = **new** ArrayList<>();  **for** (String alias : mapCharset.keySet()) {  contents.add(alias + "," + mapCharset.get(alias));  }  Files.*write*(Paths.*get*("./Charsets.csv"), contents, Charset.*defaultCharset*());  System.***out***.println("文件写入结束！请尽快确认！");  }  } |

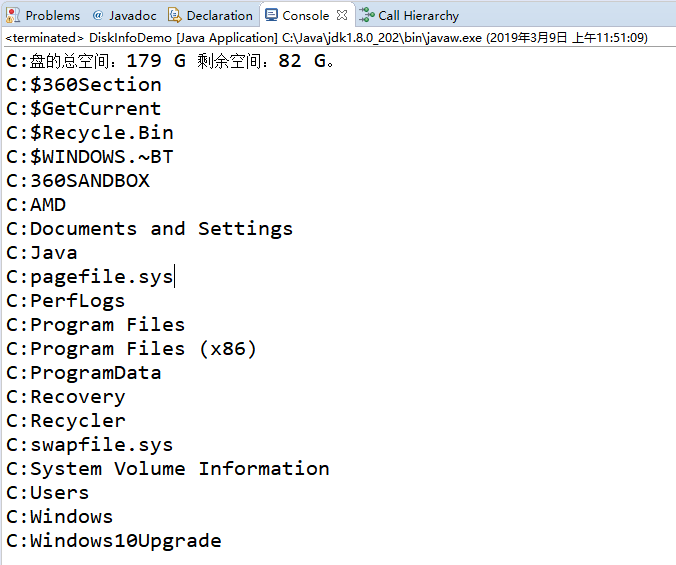
9、请编写一个控制台程序，要求能够提供如下功能：

1. 能够输出C:盘的实际空间可可用空间的大小。
2. 能够列出C:盘根目录下的所有文件和子目录。

DiskInfoDemo.java

|  |
| --- |
| **import** java.io.IOException;  **import** java.nio.file.FileStore;  **import** java.nio.file.Files;  **import** java.nio.file.Paths;  **import** java.text.MessageFormat;  **public** **class** DiskInfoDemo {  **private** **static** **final** String ***DISK\_INFO\_FMT*** = "{0}盘的总空间：{1} G\t剩余空间：{2} G。";  **private** **static** **final** **int** ***G\_BASE*** = 1024 \* 1024 \* 1024;  **public** **static** **void** main(String[] args) {  String driveName = "C:";  **try** {  FileStore fileStore = Files.*getFileStore*(Paths.*get*(driveName));  System.***out***.println(  MessageFormat.*format*(  ***DISK\_INFO\_FMT***,  driveName,  fileStore.getTotalSpace() / ***G\_BASE***,  fileStore.getUsableSpace() / ***G\_BASE***)  );  Files.*list*(Paths.*get*(driveName)).forEach(  path -> System.***out***.println(path));  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

参考运行结果：



10、使用NIO非阻塞方式实现网络通信（TCP）的读写操作。

答案：

//客户端

public void client() throws IOException{

//1. 获取通道

SocketChannel sChannel = SocketChannel.open(new

InetSocketAddress("127.0.0.1", 9898));

//2. 切换非阻塞模式

sChannel.configureBlocking(false);

//3. 分配指定大小的缓冲区

ByteBuffer buf = ByteBuffer.allocate(1024);

//4. 发送数据给服务端

Scanner scan = new Scanner(System.in)；

while(scan.hasNext()){

String str = scan.next();

buf.put((new Date().toString() + "\n" + str).getBytes());

buf.flip();

sChannel.write(buf);

buf.clear();

}

//5. 关闭通道

sChannel.close();

}

//服务端

public void server() throws IOException{

//1. 获取通道

ServerSocketChannel ssChannel = ServerSocketChannel.open();

//2. 切换非阻塞模式

ssChannel.configureBlocking(false);

//3. 绑定连接

ssChannel.bind(new InetSocketAddress(9898));

//4. 获取选择器

Selector selector = Selector.open();

//5. 将通道注册到选择器上, 并且指定“监听接收事件”

ssChannel.register(selector, SelectionKey.OP\_ACCEPT);

//6. 轮询式的获取选择器上已经“准备就绪”的事件

while(selector.select() > 0){

//7. 获取当前选择器中所有注册的“选择键(已就绪的监听事件)”

Iterator<SelectionKey> it = selector.selectedKeys().iterator();

while(it.hasNext()){

//8. 获取准备“就绪”的是事件

SelectionKey sk = it.next();

//9. 判断具体是什么事件准备就绪

if(sk.isAcceptable()){

//10. 若“接收就绪”，获取客户端连接

SocketChannel sChannel = ssChannel.accept();

//11. 切换非阻塞模式

sChannel.configureBlocking(false);

//12. 将该通道注册到选择器上

sChannel.register(selector, SelectionKey.OP\_READ);

}else if(sk.isReadable()){

//13. 获取当前选择器上“读就绪”状态的通道

SocketChannel sChannel = (SocketChannel) sk.channel();

//14. 读取数据

ByteBuffer buf = ByteBuffer.allocate(1024);

int len = 0;

while((len = sChannel.read(buf)) > 0 ){

buf.flip();

System.out.println(new String(buf.array(), 0, len));

buf.clear();

}

}

//15. 取消选择键 SelectionKey

it.remove();

}

}

}

}

11、使用FileChannel读取数据到Buffer中（10分）

|  |
| --- |
| RandomAccessFile aFile =newRandomAccessFile("data/nio-data.txt","rw");  FileChannel inChannel = aFile.getChannel();  ByteBuffer buf = ByteBuffer.allocate(48);  intbytesRead = inChannel.read(buf);  while(bytesRead != -1) {  System.out.println("Read "+ bytesRead);  buf.flip();  while(buf.hasRemaining()){  System.out.print((char) buf.get());  }  buf.clear();  bytesRead = inChannel.read(buf);  }  aFile.close(); |