IO

1. 什么是流，什么是比特，什么是字节，什么是字符

- 流：流是一串连续不断的数据集合，可以一段一段的写入长的数据流，读取数据流的时候是不知道流的分段情况的，只能从前往后读取数据流
- 比特：是二进制的最小单位，0或者1
- 字节：是计算机操作数据的最小单位，8位bit，（-128~127）
- 字符：用户可读写的最小单位，16位bit，（0~65535）

2. 什么是IO，包含哪几种流

> 它是指计算机与外部世界或者一个程序与计算机的其余部分的之间的接口。它对于任何计算机系统都非常关键，因而所有 I/O 的主体实际上是内置在操作系统中的。单独的程序一般是让系统为它们完成大部分的工作。 在 Java 编程中，直到最近一直使用 流 的方式完成 I/O。所有 I/O 都被视为单个的字节的移动，通过一个称为 Stream 的对象一次移动一个字节。流 I/O 用于与外部世界接触。它也在内部使用，用于将对象转换为字节，然后再转换回对象

字节流，字符流，输入流，输出流

3. 什么是字节流和字符流，区别，之间如何转换

- 字节流：操作的是byte类型的数据，直接对文件本身进行操作，主要操作类是OutputStream、InputStream的子类
- 字符流：操作的是字符类型的数据，使用缓冲区缓冲字符，不关闭流就不会输出任何内容，主要操作类是Reader、Writer的子类

4. 什么是输入流和输出流，区别

- 输入流：把数据写入存储介质的操作
- 输出流：从存储介质中读取数据的操作

5. 什么是NIO

- NIO与IO的作用和目的是相同的，都是计算机与外部世界或者一个程序与计算机的其余部分的之间的接口，只不过，IO是以流的方式处理数据，而NIO是以块的方式处理数据
- IO：流式数据创建过滤器很容易，面向流的IO慢
- NIO：按照块处理比按照流处理的快，但是缺少优雅性和简单性

6. 什么是AIO

Java AIO即Async非阻塞，是异步非阻塞的IO。

7. 什么是BIO

Java BIO即Block I/O， 同步并阻塞的IO。

Java BIO即Block I/O， 同步并阻塞的IO。

8. NIO,BIO,AIO之间的区别与联系

- BIO：同步阻塞IO模式，必须等待这件事情做完了才去做下一件事情
- AIO：异步非阻塞IO模式，不用等待这件事情做完了才去做下一件事，这件事情做完了就会自动告诉我他做完了
- NIO：同时支持阻塞与非阻塞模式

9. Java中BIO、NIO、AIO分别适用哪些场景

- BIO：适用于连接数较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解
- NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。
- AIO方式适用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

10. 使用BIO实现文件的读取和写入

```
//Initializes The Object
User1 user = new User1();
user.setName("hollis");
user.setAge(23);
System.out.println(user);

//Write Obj to File
ObjectOutputStream oos = null;
try {
    oos = new ObjectOutputStream(new FileOutputStream("tempFile"));
    oos.writeObject(user);
} catch (IOException e) {
    e.printStackTrace();
} finally {
    IOUtils.closeQuietly(oos);
}

//Read Obj from File
File file = new File("tempFile");
ObjectInputStream ois = null;
try {
    ois = new ObjectInputStream(new FileInputStream(file));
    User1 newUser = (User1) ois.readObject();
    System.out.println(newUser);
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} finally {
    IOUtils.closeQuietly(ois);
    try {
        FileUtils.forceDelete(file);
    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

11. 使用AIO实现文件的读取和写入

```
public class ReadFromFile {
  public static void main(String[] args) throws Exception {
    Path file = Paths.get("/usr/a.txt");
    AsynchronousFileChannel channel = AsynchronousFileChannel.open(file);

    ByteBuffer buffer = ByteBuffer.allocate(100_000);
    Future<Integer> result = channel.read(buffer, 0);

    while (!result.isDone()) {
      ProfitCalculator.calculateTax();
    }
    Integer bytesRead = result.get();
    System.out.println("Bytes read [" + bytesRead + "]");
  }
}
class ProfitCalculator {
  public ProfitCalculator() {
  }
  public static void calculateTax() {
  }
}

public class WriteToFile {

  public static void main(String[] args) throws Exception {
    AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(
        Paths.get("/asynchronous.txt"), StandardOpenOption.READ,
        StandardOpenOption.WRITE, StandardOpenOption.CREATE);
    CompletionHandler<Integer, Object> handler = new CompletionHandler<Integer, Object>() {

      @Override
      public void completed(Integer result, Object attachment) {
        System.out.println("Attachment: " + attachment + " " + result
            + " bytes written");
        System.out.println("CompletionHandler Thread ID: "
            + Thread.currentThread().getId());
      }

      @Override
      public void failed(Throwable e, Object attachment) {
        System.err.println("Attachment: " + attachment + " failed with:");
        e.printStackTrace();
      }
    };

    System.out.println("Main Thread ID: " + Thread.currentThread().getId());
    fileChannel.write(ByteBuffer.wrap("Sample".getBytes()), 0, "First Write",
        handler);

    fileChannel.write(ByteBuffer.wrap("Box".getBytes()), 0, "Second Write",
```

```
            handler);

    }
}
```

12. 使用NIO实现文件的读取和写入

```
static void readNIO() {
    String pathname = "C:\\Users\\adew\\Desktop\\jd-gui.cfg";
    FileInputStream fin = null;
    try {
        fin = new FileInputStream(new File(pathname));
        FileChannel channel = fin.getChannel();

        int capacity = 100;// 字节
        ByteBuffer bf = ByteBuffer.allocate(capacity);
        System.out.println("限制是: " + bf.limit() + "容量是: " + bf.capacity()
                           + "位置是: " + bf.position());
        int length = -1;

        while ((length = channel.read(bf)) != -1) {

            /*
                 * 注意，读取后，将位置置为0，将limit置为容量，以备下次读入到字节缓冲中，从0开始存储
                 */
            bf.clear();
            byte[] bytes = bf.array();
            System.out.write(bytes, 0, length);
            System.out.println();

            System.out.println("限制是: " + bf.limit() + "容量是: " + bf.capacity()
                               + "位置是: " + bf.position());

        }

        channel.close();

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (fin != null) {
            try {
                fin.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
static void writeNIO() {
    String filename = "out.txt";
    FileOutputStream fos = null;
    try {

        fos = new FileOutputStream(new File(filename));
        FileChannel channel = fos.getChannel();
        ByteBuffer src = Charset.forName("utf8").encode("你好你好你好你好你好");
        // 字节缓冲的容量和limit会随着数据长度变化，不是固定不变的
        System.out.println("初始化容量和limit: " + src.capacity() + ","
                           + src.limit());
        int length = 0;

        while ((length = channel.write(src)) != 0) {
            /*
                 * 注意，这里不需要clear，将缓冲中的数据写入到通道中后 第二次接着上一次的顺序往下读
                 */
            System.out.println("写入长度:" + length);
        }

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (fos != null) {
            try {
                fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

13. IO流需不需要关闭,如果关闭的话应该如何关闭。需要注意什么

- IO流一定要关闭，不然会内存泄漏，进而导致内存溢出
- 一般在finally块中关闭，并且关闭流的操作可能会抛出异常，要对其进行捕获

```
public class CloseIOStream {
    public static void main(String[] args) {
        InputStream is = null;
        OutputStream os = null;
        try {
            // ...
        }
        finally {
            if (is != null)
                try{
                    is.close();

                }
```

```
                catch (IOException e) {}

            if (os != null)
                try{
                    os.close();
                }
                catch (IOException e){}
        }
    }
}
```

14. Java 7 中关闭IO的更优雅的方式是什么

在Java 7 中新定义了一个AutoCloseable接口，他位于java.lang包下。主要在try-with-resources语句中会被自动调用，用于自动释放资源。

15. 什么是同步？什么是异步?

A调用B

- 同步：A调用B后，B马上去执行，执行完成会告诉A，A本次调用会得到结果
- 异步：A调用B后，B并不会马上执行，但是一定会执行，执行完成会告诉A，A本次调用不会得到结果

16. 什么是阻塞？什么是非阻塞?

A调用B

- 阻塞：A调用B之后一直等待B的结果，等不到就不做别的事情
- 非阻塞：A调用B之后就去做自己事情了，不需要等待B完成

17. 同步，异步 和 阻塞，非阻塞之间的区别?

同步，异步，是描述被调用方的。 阻塞，非阻塞，是描述调用方的。 同步不一定阻塞，异步也不一定非阻塞。没有必然关系。

18. IO模型有哪5种?

- 阻塞式IO模型
- 非阻塞IO模型
- IO复用模型
- 信号驱动IO模型
- 异步IO模型

19. 什么是阻塞IO模型

> 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。

20. 什么是非阻塞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。

21. 什么是多路复用IO模型

> 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。
>
> 多路复用IO比较适合连接数比较多的情况。
>
> 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。

22. 什么是信号驱动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。

23. 什么是异步IO模型

> 当进程发起一个 IO 操作，进程返回（不阻塞），但也不能返回果结；内核 把整个 IO 处理完后，会通知进程结果。如果 IO 操作成功则进程直接获取到数据。