```
音频文件等。在Java中,字节流的抽象基类是InputStream和OutputStream
                                                以字符为单位读写数据,主要用于处理文本数据。Java中使用Unicode字符
                                                集,因此字符流对国际化支持更好。字符流的抽象基类是Reader和Writer
                            按数据单位分类
                                                              直接与二进制数据交互,适合于处理所有类型的数据(如图片、视
                                                              频、声音文件等),通过字节方式处理,无需关心数据的具体内容
                                         区别与存在的意义
                                                               是为了更方便地处理字符数据(如文本文件),自动处理字符到字节的转换以
                                                              及考虑到字符编码等问题。直接使用字节流可能会导致编码错误和数据损坏
            三种分类方式的IO流
                                      输入流: 用于从源读取数据
                            按功能分类
                                       输出流: 用于向目的地写数据
                                         节点流 — 可以从或向一个特定的数据源(如文件、数组等)直接读写数据
                            按流的节点分类
                                                       用于封装一个已存在的输入流或输出流,为其提供一些额外的功能(如数据缓冲、格式转换等)
                                                     装饰器模式允许向对象动态添加新的功能,而不改变其结构
                                                     在Java IO中通过各种流(如BufferedInputStream, BufferedOutputStream, BufferedReader,
                                                     BufferedWriter等)实现,这些流包装了其他流,提供了例如缓冲的功能,增强了性能并简化了编程接口
                                                                    InputStream 是抽象组件
            在IO中使用的设计模式
                              Java的IO库广泛应用了装饰器模式
                                                                    FileInputStream 是 InputStream 的子类,属于具体组件,提供了字节流的输入操作
                                                                    FilterInputStream 属于抽象装饰者,装饰者用于装饰组件,为组件提供额外的
                                                                    功能。例如 BufferedInputStream 为 FileInputStream 提供缓存的功能
                                                     以 InputStream 为例
                                                                    实例化一个具有缓存功能的字节流对象时,只需要在 FileInputStream 对象上
                                                                                                                      ileInputStream fileInputStream = new FileInputStream(filePath);
                                                                    再套一层 BufferedInputStream 对象即可
                                                                                                                      ufferedInputStream bufferedInputStream = new BufferedInputStream(fileInputStream);
                                                                    DataInputStream 装饰者提供了对更多数据类型进行输入的操作,比如 int、double 等基本类型
                      磁盘操作: File —— File 类可以用于表示文件和目录的信息,但是它不表示文件的内容
                      字节操作: InputStream 和 OutputStream
                      字符操作: Reader 和 Writer
                                                                                                                                                                             将对象的状态持久保存到一个存储媒体(如文件、数
                                                                                                             序列化的类需要实现 Serializable 接口,它只是一个标准,没有任何
                                                                                                                                                                            据库)以便后续使用。
                                                                                                                                                        序列化主要用于以下几个场景
                                                                                     ObjectOutputStream.writeObject()
                                                                                                             方法需要实现,但是如果不去实现它的话而进行序列化,会抛出异常
                                                                                                                                                                             通过网络发送对象的状态信息,例如在分布式系统中
                                                                                                      serialVersionUID 是一个类中用于序列化控制的固定标识符。它确保反序列化时类版本的一致性。如果未指定,JVM
                                                                                                      会自动创建一个,但类的任何更改都可能导致生成的版本号不同,从而在反序列化时引发InvalidClassException。
                                                       serialVersionUID 提供了一种向前和向后兼容序列化对象的方式。如果类的字段发
                                                                                                    ─ 为什么需要 serialVersionUID
                                                                                                                          生了变化,你仍然可以反序列化旧版本的对象,只要serialVersionUID没有变化
                                                                                                                 transient 关键字可以使一些属性不会被序列化
                                                                                                                 ArrayList 中存储数据的数组 elementData 是用 transient 修饰的,因为这个数组
                                                                                     不想让变量序列化怎么办? —— transient
                                                                                                                 是动态扩展的,并不是所有的空间都被使用,因此就不需要所有的内容都被序列化
            IO常见类
                                                                                                                 通过重写序列化和反序列化方法,使得可以只序列化数组中有内容的那部分数据
                      对象操作: Serializable —— IO: 序列化和反序列化
                                                       反序列化是指把字节序列恢复为 Java 对象的过程 —— ObjectInputStream.readObject()
                                                                   一方面实现了数据的持久化,通过序列化可以把数据永久的保存到硬盘上
                                                                   二是利用序列化实现远程通信,即在网络上传送对象的字节序列
                                                       常见的序列化反序列化协议: XML, JSON 等
                                                                        · Java标准序列化 —— 使用java.io.Serializable接口。任何实现此接口的类都可以通过ObjectOutputStream进行序列化,反序列化通过ObjectInputStream
                                                                        - Externalizable接口 —— 与Serializable类似,但给予了类更多的控制权来保存和恢复其状态
                                                       Java支持几种序列化方式
                                                                         JSON/XML等格式序列化 —— 可以使用各种库(如Jackson、Gson)将对象转换为JSON或XML格式,便于Web传输。
                                                  InetAddress: 用于表示网络上的硬件资源,即 IP 地址;
                                                 · URL: 统一资源定位符;
                                                                           - ServerSocket: 服务器端类
                      网络操作: Socket —— Java 中的网络支持
                                                  Sockets: 使用 TCP 协议实现网络通信
                                                                            Socket: 客户端类
                                                                            服务器和客户端通过 InputStream 和 OutputStream 进行输入输出
                                                                            DatagramSocket: 通信类
                                                  Datagram: 使用 UDP 协议实现网络通信
                                                                            DatagramPacket: 数据包类
                                         应用进程被阻塞,直到数据复制到应用进程缓冲区中才返回
                                        一在阻塞的过程中,其它程序还可以执行,因此阻塞不意味着整个操作系统都被阻塞
                               阻塞式 I/O -
                                         因为其他程序还可以执行,因此不消耗 CPU 时间,这种模型的执行效率会比较高
                                              应用进程执行系统调用之后,内核返回一个错误码
                               非阻塞式 I/O(NIO)
                                             应用进程可以继续执行,但是需要不断的执行系统调用来获知 I/O 是否完成,这种方式称为轮询(polling)
                                              由于 CPU 要处理更多的系统调用,因此这种模型是比较低效的
                                                                                          之后再使用 recvfrom 把
                                                使用 select 或者 poll 等待数据,并且可以等待多个套接字中的任
                                                何一个变为可读,这一过程会被阻塞,当某一个套接字可读时返回
                                                                                          数据从内核复制到进程中
                                                它可以让单个进程具有处理多个 I/O 事件的能力又被称为 Event Driven I/O,即事件驱动 I/O
                               I/O 复用(select 和 poll)
                                                如果一个 Web 服务器没有 I/O 复用,那么每一个 Socket 连接都需要创建
JAVA IO
                                                一个线程去处理。如果同时有几万个连接,那么就需要创建相同数量的线程
                                                相比于多进程和多线程技术,I/O 复用不需要进程线程创建和切换的开销,系统开销更小
                                                应用进程使用 sigaction 系统调用,内核立即返回,应用进程可以继续执行,也就是说等待数据阶段应用进程是非阻塞的
                               信号驱动式 I/O(SIGIO)
                                                内核在数据到达时向应用进程发送 SIGIO 信号,应用进程收到之后在信号处理程序中调用 recvfrom 将数据从内核复制到应用进程中
            Unix IO 模型
                                                相比于非阻塞式 I/O 的轮询方式,信号驱动 I/O 的 CPU 利用率更高
                                           进行 aio read 系统调用会立即返回,应用进程继续执行,不会被阻塞,内核会在所有操作完成之后向应用进程发送信号
                               异步 I/O(AIO) -
                                           异步 I/O 与信号驱动 I/O 的区别在于,异步 I/O 的信号是通知应用进程 I/O 完成,而信号驱动 I/O 的信号是通知应用进程可以开始 I/O
                                                   同步 I/O: 应用进程在调用 recvfrom 操作时会阻塞。
                                    同步 I/O 与异步 I/O -
                                                   异步 I/O: 不会阻塞。
                         I/O 模型比较
                                                  前四种 I/O 模型的主要区别在于第一个阶段,而第二个阶段是一
                                                                                                                                                                    I/O multiplexing | signal-driven I/O | asynchronous I/O
                                    五大 I/O 模型比较
                                                                                                                                                  blocking
                                                                                                                                                           nonblocking
                                                  样的: 将数据从内核复制到应用进程过程中, 应用进程会被阻塞
                                                                                                                                                            check
                                                                                                                                                                       check
                                                                                                                                                  initiate
                                                                                                                                                                                           initiate
                                                                                                                                                            check
                             IO多路复用(I/O Multiplexing)是一种有效的技术,用于提高程序在处理多个输入/输出操作时的效率。它允许单个线程
                                                                                                                                                            check
                             或进程监视多个文件描述符,一旦一个或多个文件描述符准备好进行读写操作,程序就能被通知,从而执行相应的IO操作
                                                                                                                                                            check
                                                                                                                                                                                                    wait for
                                                                                                                                                            check
                                                     select 是最早的IO多路复用接口之一。它允许程序监视一组文件描述符,以便知道它们何时可以进行非阻塞读取或写入。
                                                                                                                                                            check
                                                     缺点是所能处理的文件描述符数量有限,通常受系统限制;同时,每次调用select时,都需要重置文件描述符的集合,这在有大量描述符时会导致效率低下
                                                                                                                                                            check
                                                                                                                                                            check
                                                   poll 功能与select类似,但没有文件描述符数量的限制。它使用一个结构体数组来管理文件描述符集合,从而解决了select的一些限制。
                                                                                                                                                            check
                                                                                                                                                                               notification
                                                                                                                                                                       ready
                       常见的IO多路复用技术/应用场景
                                                   与select一样,poll也需要在每次调用时检查整个数组,因此在描述符非常多的情况下效率可能会降低
            IO多路复用
                                                    epoll 是Linux特有的IO多路复用解决方案,相对于select和poll有显著的性能提升,尤其是在处理大量描述符时
                                                                                                                                                                                                    copy data
                                                                                                                                                                                                    from kernel
                                                    epoll可以是边缘触发 (ET) 或水平触发 (LT) , 这为不同的使用场景提供了灵活性。它使用
                                                                                                                                                                                                    to user
                                                    一个事件表来跟踪每个文件描述符,从而避免了每次调用时重新扫描整个描述符集的开销
                                                                                                                                                                                          notification
                                                                                                                                                 complete
                                                                                                                                                                                complete
                                                                                                                                                            complete
                                                                                                                                                                      complete
                                                        当 epoll wait() 检测到描述符事件到达时,将此事件通知进程,进程可以不立即处理该事件,下次调
                                               水平触发
                                                        用 epoll_wait() 会再次通知进程。是默认的一种模式,并且同时支持 Blocking 和 No-Blocking
                                                                                                                                                                                          handles both
                                                                                                                                                            1st phase handled differently,
                       epoll 的描述符事件有两种触发模式
                                                       和 LT 模式不同的是,通知之后进程必须立即处理事件,下次再调用 epoll_wait() 时不会再得到事件到达的通知
                                                                                                                                                                                           phases
                                                                                                                                                            2nd phase handled the same
                                               边缘触发
                                                                                                                                                           (blocked in call to recvfrom)
                                                        很大程度上减少了 epoll 事件被重复触发的次数,因此效率要比 LT 模式高。只支持 No-Blocking,
                                                        以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死
                                                       同步且阻塞:BIO 是一种传统的I/O模型,使用时服务器实现一个客户端一个线程模型,即每当一个新的客户端连接时,服务器就需要创建一个新的线程来处理该连接
                                                       资源消耗高:因为每个请求都需要创建独立的线程进行处理,当并发数较高时,线程数量爆增,会导致严重的资源消耗和上下文切换问题
                                        BIO (Blocking I/O)
                                                       适用场景:适合连接数目较少且固定的架构,这种方式可以简化开发过程,因为通信是阻塞的,程序编码简单
                                                          同步非阻塞:NIO 支持使用一个线程来处理多个连接,通过所谓的"选择器 (Selector)"轮询连接是否有数据可读写,这样可以使单个线程高效地管理多个并发连接。
                                                          缓冲区 (Buffer): NIO 基于缓冲区操作,数据需要先读到缓冲区再进行处理,支持缓冲区的复用。
                                        NIO (Non-blocking I/O)
            在Java网络编程中,主要存在三种I/O模型
                                                          通道 (Channel): 不同于BIO中的流, NIO使用通道进行数据的读写操作,通道可以同时读写,更加灵活。
                                                          适用场景:适用于连接数较多且连接时间较长的应用,如网络服务器。
                                                           异步非阻塞:AIO 是真正的异步非阻塞I/O,对应于UNIX网络编程中的POSIX的asynchronous I/O,Java从1.7开始引入了AIO。
                                                           异步通道:提供了异步通道的概念,允许请求在完成时通知应用程序,即应用程序可以继续执行其他任务,当数据准备就绪时系统会自动将数据读取或写入。
                                        AIO (Asynchronous I/O)
                                                           回调机制:支持通过回调机制处理I/O操作结果,提高程序的响应能力。
                                                           适用场景:适用于高性能、大量数据交换的网络应用,特别是服务器需要处理成千上万的并发连接,这种场景下AIO可以显著减少通信延迟。
                                                                             NIO 是非阻塞的
                                                   NIO 与普通 I/O 的区别主要有以下两点
                                                                              NIO 面向块, I/O 面向流 —— 普通I/O实际上就是BIO
                                                   流与块 —— I/O 与 NIO 最重要的区别是数据打包和传输的方式,I/O 以流的方式处理数据,而 NIO 以块的方式处理数据
                                                                     通道 Channel 是对原 I/O 包中的流的模拟,可以通过它读取和写入数据
                                                                      通道与流的不同之处在于,流只能在一个方向上移动(一个流必须是 InputStream 或者 OutputStream 的子类),而通道是双向的,可以用于读、写或者同时用于读写
                                                   通道与缓冲区
                                                                      发送给一个通道的所有数据都必须首先放到缓冲区中,同样地,从通道中读取的任何数据都要先读到缓冲区中
                                                                      一不会直接对通道进行读写数据,而是要先经过缓冲区
                                                               缓冲区
                                                                      缓冲区实质上是一个数组
                                                           ·NIO 常常被叫做非阻塞 IO,主要是因为 NIO 在网络通信中的非阻塞特性被广泛使用
                                                           · NIO 实现了 IO 多路复用中的 Reactor 模型,一个线程 Thread 使用一个选择器 Selector 通过轮询的方式去监听多个通道 Channel 上的事件,从而让一个线程就可以处理多个事件,
                                                   选择器
                                                           通过配置监听的通道 Channel 为非阻塞,那么当 Channel 上的 IO 事件还未到达时,就不会进
                                                           入阻塞状态一直等待,而是继续轮询其它 Channel, 找到 IO 事件已经到达的 Channel 执行
                                                           因为创建和切换线程的开销很大,因此使用一个线程来处理多个事件会有更好的性能
                                                               内存映射文件 I/O 是一种读和写文件数据的方法,它可以比常规的基于流或者基于通道的 I/O 快得多
                                                   内存映射文件
                                                               向内存映射文件写入可能是危险的,只是改变数组的单个元素这样的简单操作,就可能会直接修改磁盘上的文件
                                                                Netty是一个高性能、异步事件驱动的NIO框架,提供了对TCP、UDP和文件传输的支持。作为当前最流行的NIO框架,Netty在互联网领域、大
                                                                数据分布式计算领域、游戏行业、通信行业等获得了广泛的应用,一些业界著名的开源组件也基于Netty构建,比如RPC框架、zookeeper等
                                                                      - api简单,开发门槛低
            Java NIO (N是new的意思, jdk1.4引入的新库)
                                             Java N(A)IO - 框架: Netty
                                                                      功能强大,内置了多种编码、解码功能
                                                                      - 与其它业界主流的NIO框架对比,netty的综合性能最优
                                                                      社区活跃,使用广泛,经历过很多商业应用项目的考验
                                                                      定制能力强,可以对框架进行灵活的扩展
                                                                                                                    通道 (Channel) 是全双工的(双向传输),它既可能是读缓冲
                                                                                                                    区 (read buffer) ,也可能是网络缓冲区 (socket buffer)
                                                       在 Java NIO 中的通道 (Channel) 就相当于操作系统的内核空间 (kernel space) 的缓冲区,而缓
                                                       冲区 (Buffer) 对应的相当于操作系统的用户空间 (user space) 中的用户缓冲区 (user buffer)
                                                                                                                    缓冲区 (Buffer) 分为堆内存 (HeapBuffer) 和堆外内存
                                                                                                                     (DirectBuffer) , 这是通过 malloc() 分配出来的用户态内存
                                                                  FileChannel 是一个用于文件读写、映射和操作的通道,同时它在并发环境下是线程安全的
                                                       FileChannel
                                                                  基于 FileInputStream、FileOutputStream 或者 RandomAccessFile
                                             零拷贝实现
                                                                  的 getChannel() 方法可以创建并打开一个文件通道
                                                                      MappedByteBuffer 是 NIO 基于**内存映射(mmap)**这种零拷贝方式的提供的一种实现,它继承自 ByteBuffer
                                                       MappedByteBuffer
                                                                      FileChannel 定义了一个 map() 方法,它可以把一个文件从 position 位置开始的 size 大小的区域映射为内存映像文件
                                                                     DirectByteBuffer 的对象引用位于 Java 内存模型的堆里面,JVM 可以对 DirectByteBuffer 的对象进行内存分配和回收管理
```

一般使用 DirectByteBuffer 的静态方法 allocateDirect() 创建 DirectByteBuffer 实例并分配内存

以字节 (8位二进制) 为单位读写数据, 主要用于处理原始二进制数据如图像、