链滴

个人整理 - Java 后端面试题 - IO 篇

作者：valarchie

原文链接：https://ld246.com/article/1583388476421

来源网站：链滴

● 标★号的知识点为重要知识点

java中IO流的体系?

Java中的流分为两种，一种是字节流，另一种是字符流，分别由四个抽象类来表示（每种流包括输入输出两种所以一共四　　个）:InputStream，OutputStream，Reader，Writer。基于这四种IO流类根据不同需求派生出其他IO流。

★BIO,NIO,AIO?

　BIO是同步阻塞IO，NIO是同步非阻塞IO，AIO是异步非阻塞IO；三种IO方式相比较而言，BIO是个客户端对应一个线程，
　优化的话可以用线程池进行线程复用，但本质还是一个客户端-服务端通信对应一个线程；NIO只需一个线程负责多路复用
　器selector的轮询，就可以处理不同客户端channel中的读/写事件，所以多个客户端实际只对应一线程，另外服务器端
　和客户端均使用缓冲区的方式进行读写；AIO不需要轮询去查看读写事件是否就绪，而是由内核通回调函数通知并完成后续
　操作。

NIO和IO的区别

第一点，NIO少了1次从内核空间到用户空间的拷贝。

ByteBuffer.allocateDirect()分配的内存使用的是本机内存而不是Java堆上的内存，和网络或者磁盘互都在操作系统的内核空间中发生。

allocateDirect()的区别在于这块内存不由java堆管理, 但仍然在同一用户进程内。

第二点，NIO以块处理数据，IO以流处理数据

第三点，非阻塞，NIO1个线程可以管理多个输入输出通道

讲讲IO里面的常见类，字节流、字符流、接口、实现类

InputStream和OutStream属于字节流。底下有ByteArray、Object、File等实现类，以及Buffer增。

Reader和Writer属于字符流。底下有String、File、Buffer等实现。

讲讲NIO。

NIO通过观测多个缓冲区，哪个缓冲区就绪的话，就处理哪个缓冲区。原本的IO会对一个缓冲区等待效率比较慢，

而现在NIO是对一堆缓冲区进行等待，效率比较高。

★讲一下NIO和网络传输

NIO Reactor反应器模式，例如汽车是乘客访问的实体reactor，乘客上车后到售票员处Acceptor登，之后乘客便可休息睡觉了，到达乘客目的地后，售票员Aceptor将其唤醒即可。持久TCP长链接每个

lient和server之间有存在一个持久连接，当CCU（用户并发数量）上升，阻塞server无法为每个连接行1个线程，自己开发1个二进制协议，将message压缩至3-6倍，传输双向且消息频率高，假设serve链接了2000个client，每个client平均每分钟传输1-10个message，1个messaged的大小为几百字节几千字节，而server也要向client广播其他玩家的当前信息，需要高速处理消息的能力。Buffer，网络节存放传输的地方，从channel中读写，从buffer作为中间存储格式，channel是网络连接与buffer间据通道，像之前的socket的stream。

字节流和字符流的区别?

字节流不会用到内存缓冲区，文件本身直接操作。字符流操作使用内存缓存区，用缓存存操作文件。符流在输出前将所有内容暂时保存到内存中，即缓存区暂时存储，如果想不关闭也将字符流输出则可使用flush方法强制刷出。字节字符转化可能存在系统编码lang，要制定编码。getbyte字节流使用更广泛。

FileInputStream 在使用完以后，不关闭流，想二次使用可怎么操作?

可以使用apache的IOUtils包进行copy转成ByteArrayInputStream进行重复读取。

或者使用反射调用open方法进行重复读取。

★Java NIO使用

利用 Selector、Buffer、Channel三个部件。Selector可以同时监听一组通信信道（Channel）上的I/状态，前提是这个Selector已经注册到这些通信信道中。选择器Selector可以调用select()方法检查已注册的通信信道上I/O是否已经准备好，如果已经准备好的话，直接读取Buffer中的数据。

★IO与NIO的比较

面向流 VS 面向缓冲

Java I/O是面向流的， 每次从流（InputStream/OutputStream）中读一个或多个字节，直到读取完有字节，它们没有被缓存在任何地方。另外，它不能前后移动流中的数据，如需前后移动处理，需要将其缓存至一个缓冲区。

Java I/O是面向缓冲， 数据会被读取到一个缓冲区，需要时可以在缓冲区中前后移动处理，这增加了理过程的灵活性。但与此同时在处理缓冲区前需要检查该缓冲区中是否包含有所需要处理的数据，并要确保更多数据读入缓冲区时，不会覆盖缓冲区内尚未处理的数据。

阻塞 VS 非阻塞

Java IO的各种流是阻塞的。当某个线程调用read()或write()方法时，该线程被阻塞，直到有数据被读到或者数据完全写入。阻塞期间该线程无法处理任何其它事情。

Java NIO非阻塞模式。读写请求并不会阻塞当前线程，在数据可读/写前当前线程可以继续做其它事，所以一个单独的线程可以管理多个输入和输出通道。

选择器

Java NIO的选择器允许一个单独的线程同时监视多个通道，可以注册多个通道到同一个选择器上，然使用一个单独的线程来“选择”已经就绪的通道。这种“选择”机制为一个单独线程管理多个通道提了可能。

零拷贝

Java NIO中提供的FileChannel拥有transferTo和transferFrom两个方法，可直接把FileChannel中的据拷贝到另外一个Channel，或者直接把另外一个Channel中的数据拷贝到FileChannel。该接口常被于高效的网络/文件的数据传输和大文件拷贝。在操作系统支持的情况下，通过该方法传输数据并不要将源数据从内核态拷贝到用户态，再从用户态拷贝到目标通道的内核态，同时也避免了两次用户态内核态间的上下文切换，也即使用了“零拷贝”，所以其性能一般高于Java IO中提供的方法。

通道

NIO的一大创新就是通道，channel可以是双向的，而IO流是单向的。

谈谈reactor模型。

同步的等待多个事件源到达（采用select()实现）

将事件多路分解以及分配相应的事件服务进行处理，这个分派采用server集中处理（dispatch）

分解的事件以及对应的事件服务应用从分派服务中分离出去（handler）

1. Reactor 将I/O事件分派给对应的Handler
2. Acceptor 处理客户端新连接，并分派请求到处理器链中
3. Handlers 执行非阻塞读/写 任务

reactor有三种模型

1.Reactor单线程模型

Reactor单线程模型，指的是所有的I/O操作都在同一个NIO线程上面完成，NIO线程的职责如下：
作为NIO服务端，接收客户端的TCP连接；
作为NIO客户端，向服务端发起TCP连接；
读取通信对端的请求或者应答消息；
向通信对端发送消息请求或者应答消息；
Reactor线程是个多面手，负责多路分离套接字，Accept新连接，并分派请求到处理器链中。该模型用于处理器链中业务处理组件能快速完成的场景。不过，这种单线程模型不能充分利用多核资源，所实际使用的不多。
对于一些小容量应用场景，可以使用单线程模型，但是对于高负载、大并发的应用却不合适，主要原如下：
一个NIO线程同时处理成百上千的链路，性能上无法支撑。即便NIO线程的CPU负荷达到100%，也法满足海量消息的编码、解码、读取和发送；
当NIO线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往进行重发这更加重了NIO线程的负载，最终导致大量消息积压和处理超时，NIO线程会成为系统的性能瓶颈；
可靠性问题。一旦NIO线程意外跑飞，或者进入死循环，会导致整个系统通讯模块不可用，不能接收处理外部信息，造成节点故障。

2.单Reactor多线程模型

Reactor多线程模型与单线程模型最大区别就是有一组NIO线程处理I/O操作，它的特点如下：
有一个专门的NIO线程--acceptor新城用于监听服务端，接收客户端的TCP连接请求；
网络I/O操作--读、写等由一个NIO线程池负责，线程池可以采用标准的JDK线程池实现，它包含一个务队列和N个可用的线程，
由这些NIO线程负责消息的读取、解码、编码和发送；1个NIO线程可以同时处理N条链路，但是1个路只对应1个NIO线程，防止发生并发操作问题。
在绝大多数场景下，Reactor多线程模型都可以满足性能需求；但是，在极特殊应用场景中，一个NI线程负责监听和处理所有的客户端

连接可能会存在性能问题。例如百万客户端并发连接，或者服务端需要对客户端的握手信息进行安全证，认证本身非常损耗性能。
这类场景下，单独一个Acceptor线程可能会存在性能不足问题，为了解决性能问题，产生了第三种Rector线程模型--主从Reactor多线程模型。

3.主从Reactor多线程模型

特点是：服务端用于接收客户端连接的不再是1个单独的NIO线程，而是一个独立的NIO线程池。Acceptor接收到客户端TCP连接请求处理完成后
（可能包含接入认证等），将新创建的SocketChannel注册到I/O线程池（sub reactor线程池）的某个/O线程上，由它负责SocketChannel
的读写和编解码工作。
Acceptor线程池只用于客户端的登录、握手和安全认证，一旦链路建立成功，就将链路注册到后端suReactor线程池的I/O线程上，
有I/O线程负责后续的I/O操作。第三种模型比起第二种模型，是将Reactor分成两部分，mainReacto负责监听server socket，accept新连接，
并将建立的socket分派给subReactor。subReactor负责多路分离已连接的socket，读写网 络数据，业务处理功能，其扔给worker线程池完成。
通常，subReactor个数上可与CPU个数等同。

★select,poll,epoll

(1)select==>时间复杂度O(n)

它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以select具有O(n的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。

(2)poll==>时间复杂度O(n)

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状， 但是它没有最大连接数的限制，原因是它是基于链表来存储的.

(3)epoll==>时间复杂度O(1)

epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通我们。所以我们说epoll实际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有义的。（复杂度降低到了O(1)）

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，旦某个描述符就绪 （一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，pol，epoll本质上都是同步I/O，因为他们都 需要在读写事件就绪后自己负责进行读写，也就是说这个读过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现 会负责把数据从内核拷贝到用户间。

netty的线程模型，netty如何基于reactor模型上实现的。

https://blog.csdn.net/quxing10086/article/details/80296245

为什么选择netty。

1. API使用简单，开发门槛低；

2. 功能强大，预置了多种编解码功能，支持多种主流协议；

3. 定制能力强，可以通过ChannelHandler对通信框架进行灵活地扩展；

4. 性能高，通过与其他业界主流的NIO框架对比，Netty的综合性能最优；

5. 成熟、稳定，Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为NIO的BUG而恼；

6. 社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会加入；

7. 经历了大规模的商业应用考验，质量得到验证。在互联网、大数据、网络游戏、企业应用、电信软等众多行业得到成功商用，

证明了它已经完全能够满足不同行业的商业应用了。

什么是TCP粘包，拆包。解决方式是什么。

客户端在发送数据包的时候，每个包都固定长度，比如1024个字节大小，如果客户端发送的数据长度足1024个字节，则通过补充空格的方式补全到指定长度；

客户端在每个包的末尾使用固定的分隔符，例如\r\n，如果一个包被拆分了，则等待下一个包发送过之后找到其中的\r\n，然后对其拆分后的头部部分与前一个包的剩余部分进行合并，这样就得到了一完整的包；

将消息分为头部和消息体，在头部中保存有当前整个消息的长度，只有在读取到足够长度的消息之后算是读到了一个完整的消息；通过自定义协议进行粘包和拆包的处理。

netty是如何解决粘包的?

● FixedLengthFrameDecoder

对于使用固定长度的粘包和拆包场景，可以使用FixedLengthFrameDecoder，该解码一器会每次读固定长度的消息，如果当前读取到的消息不足指定长度，那么就会等待下一个消息到达后进行补足。

● LineBasedFrameDecoder与DelimiterBasedFrameDecoder

对于通过分隔符进行粘包和拆包问题的处理，Netty提供了两个编解码的类，LineBasedFrameDecodr和

DelimiterBasedFrameDecoder。这里LineBasedFrameDecoder的作用主要是通过换行符，即\n或\r\n对数据进行处理； 而DelimiterBasedFrameDecoder的作用则是通过用户指定的分隔符对数据行粘包和拆包处理。

● LengthFieldBasedFrameDecoder与LengthFieldPrepender

这里LengthFieldBasedFrameDecoder与LengthFieldPrepender需要配合起来使用，其实本质上来，这两者一个是解码，一个是编码的关系。它们处理粘拆包的主要思想是在生成的数据包中添加一个度字段，用于记录当前数据包的长度。

● 自定义粘包与拆包器

对于粘包与拆包问题，其实前面三种基本上已经能够满足大多数情形了，但是对于一些更加复杂的协，可能有一些定制化的需求。

对于这些场景，其实本质上，我们也不需要手动从头开始写一份粘包与拆包处理器，而是通过继承LenthFieldBasedFrameDecoder

和LengthFieldPrepender来实现粘包和拆包的处理。

转自我的github

技术讨论群QQ:1398880