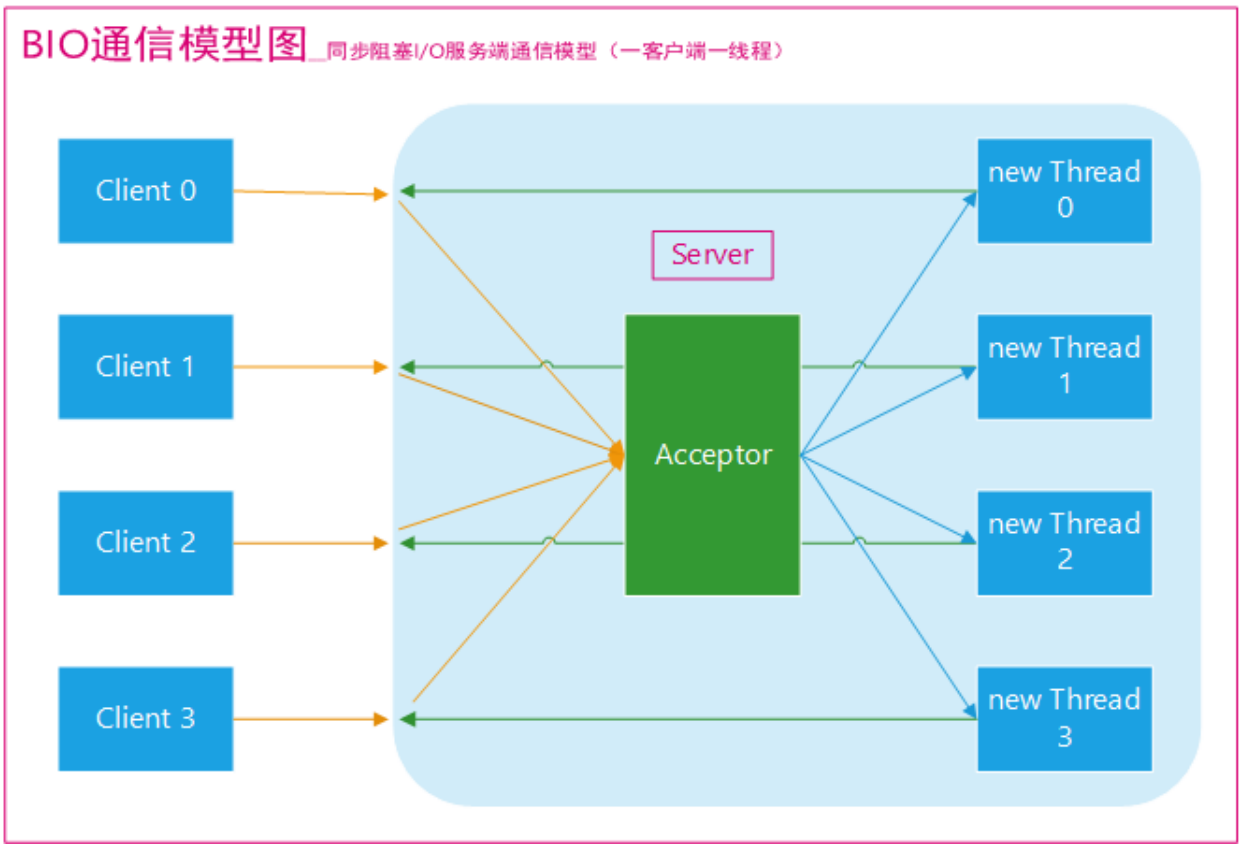
bio/nio/aio

一、IO的方式通常分为几种，同步阻塞的BIO、同步非阻塞的NIO、异步非阻塞的AIO。

1.1、传统的BIO编程



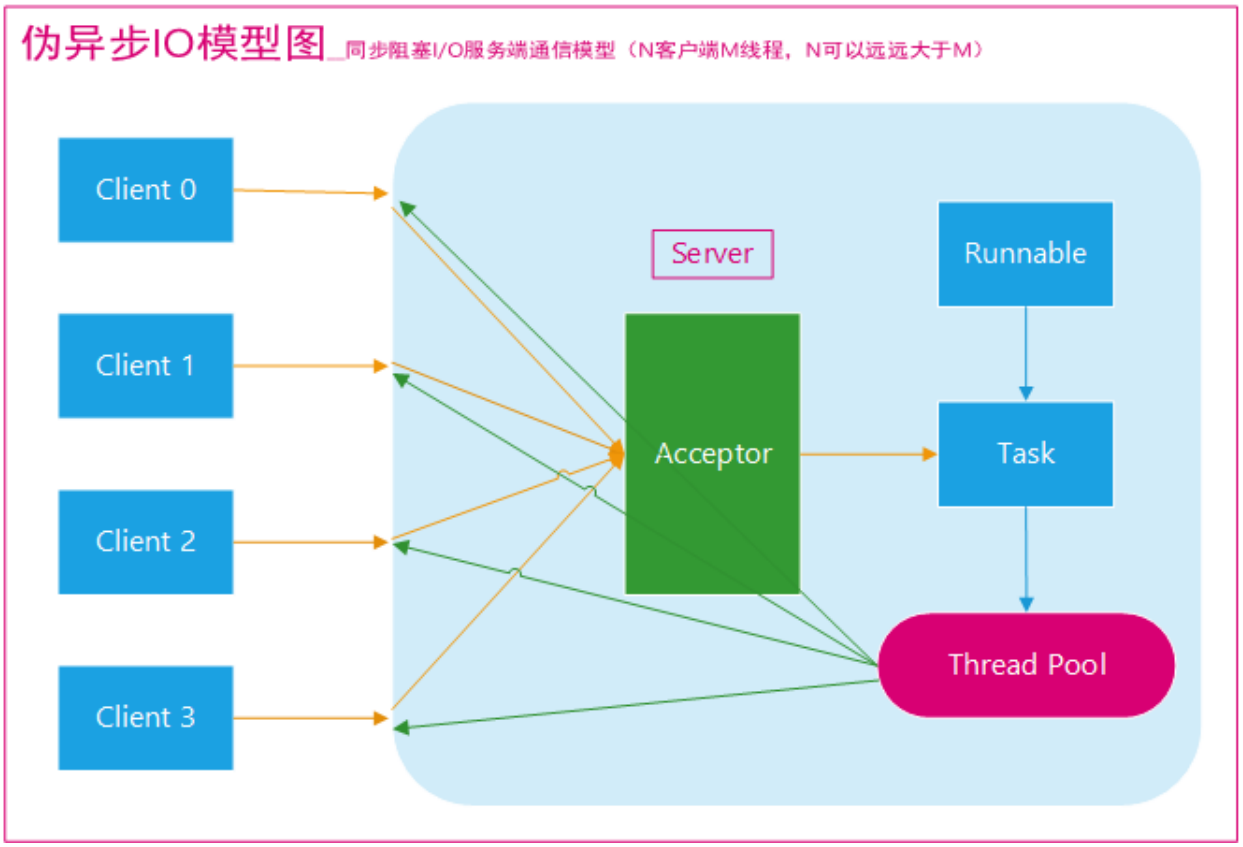
网络编程的基本模型是C/S模型，即两个进程间的通信。

传统的同步阻塞模型开发中，ServerSocket负责绑定IP地址，启动监听端口；Socket负责发起连接操作。连接成功后，双方通过输入和输出流进行同步阻塞式通信。

简单的描述一下BIO的服务端通信模型：采用BIO通信模型的服务端，通常由一个独立的Acceptor线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理没处理完成后，通过输出流返回应答给客户端，线程销毁。即典型的一请求一应答通宵模型。

1.2、伪异步I/O编程

为了改进这种一连接一线程的模型，我们可以使用线程池来管理这些线程（需要了解更多请参考前面提供的文章），实现1个或多个线程处理N个客户端的模型（但是底层还是使用的同步阻塞I/O），通常被称为"伪异步I/O模型"。



2.1、NIO

NIO提供了与传统BIO模型中的Socket和ServerSocket相对应的SocketChannel和ServerSocketChannel两种不同的套接字通道实现。

新增的着两种通道都支持阻塞和非阻塞两种模式。

阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。

对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用NIO的非阻塞模式来开发。

2.2、缓冲区 Buffer

Buffer是一个对象，包含一些要写入或者读出的数据。

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，也是写入到缓冲区中。任何时候访问NIO中的数据，都是通过缓冲区进行操作。

缓冲区实际上是一个数组，并提供了对数据结构化访问以及维护读写位置等信息。

具体的缓存区有这些：ByteBuffe、CharBuffer、 ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer。他们实现了相同的接口：Buffer。

2.3、通道 Channel

我们对数据的读取和写入要通过Channel，它就像水管一样，是一个通道。通道不同于流的地方就是通道是双向的，可以用于读、写和同时读写操作。

底层的操作系统的通道一般都是全双工的，所以全双工的Channel比流能更好的映射底层操作系统的API。

Channel主要分两大类：

SelectableChannel：用户网络读写

FileChannel：用于文件操作

2.4、多路复用器 Selector

Selector是Java NIO 编程的基础。

Selector提供选择已经就绪的任务的能力：Selector会不断轮询注册在其上的Channel，如果某个Channel上面发生读或者写事件，这个Channel就处于就绪状态，会被Selector轮询出来，然后通过SelectionKey可以获取就绪Channel的集合，进行后续的I/O操作。

一个Selector可以同时轮询多个Channel，因为JDK使用了epoll()代替传统的select实现，所以没有最大连接句柄1024/2048的限制。所以，只需要一个线程负责Selector的轮询，就可以接入成千上万的客户端。

3、AIO编程

NIO 2.0引入了新的异步通道的概念，并提供了异步文件通道和异步套接字通道的实现。

异步的套接字通道时真正的异步非阻塞I/O，对应于UNIX网络编程中的事件驱动I/O（AIO）。他不需要过多的Selector对注册的通道进行轮询即可实现异步读写，从而简化了NIO的编程模型。

属性 ↓ I/O 模型 →	同步阻塞 I/O（BIO）	伪异步 I/O	非阻塞 I/O（NIO）	异步 I/O（AIO）
Client 数:I/O 线程	1:1	M:N(M>=N)	M:1	M:0
I/O（阻塞）类型	阻塞 I/O	阻塞 I/O	非阻塞 I/O	非阻塞 I/O
I/O（同步）类型	同步 I/O	同步 I/O	同步 I/O（多路复用）	异步 I/O
API 使用难度	简单	简单	复杂	一般
调试难度	简单	简单	复杂	复杂
可靠性	非常差	差	高	高
吞吐量	低	中	高	高

解答

一：事件分离器

在IO读写时，把 IO请求 与 读写操作 分离调配进行，需要用到事件分离器。根据处理机制的不同，事件分离器又分为：同步的Reactor和异步的Proactor。

Reactor模型：

- 应用程序在事件分离器注册 读就绪事件 和 读就绪事件处理器
- 事件分离器等待读就绪事件发生
- 读就绪事件发生，激活事件分离器，分离器调用 读就绪事件处理器（即：可以进行读操作了，开始读）
- 读事件处理器开始进行读操作，把读到的数据提供给程序使用

Proactor模型：

- 应用程序在事件分离器注册 读完成事件 和 读完成事件处理器，并向操作系统发出异步读请求
- 事件分离器等待操作系统完成读取
- 在分离器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分离器读操作完成
- 事件分离器监听到 读完成事件 后，激活 读完成事件的处理器
- 读完成事件处理器 处理用户自定义缓冲区中的数据给应用程序使用

同步和异步的区别就在于 读 操作由谁完成：同步的Reactor是指程序发出读请求后，由分离器监听到可以进行读操作时（需要获得读操作条件）通知事件处理器进行读操作，异步的Proactor是指程序发出读请求后，操作系统立刻异步地进行读操作了，读完之后在通知分离器，分离器激活处理器直接取用已读到的数据。

二：同步阻塞IO（BIO）

我们熟知的Socket编程就是BIO，一个socket连接一个处理线程（这个线程负责这个Socket连接的一系列数据传输操作）。阻塞的原因在于：操作系统允许的线程数量是有限的，多个socket申请与服务端建立连接时，服务端不能提供相应数量的处理线程，没有分配到处理线程的连接就会阻塞等待或被拒绝。

三：同步非阻塞IO（NIO）

New IO是对BIO的改进，基于Reactor模型。我们知道，一个socket连接只有在特点时候才会发生数据传输IO操作，大部分时间这个"数据通道"是空闲的，但还是占用着线程。NIO作出的改进就是"一个请求一个线程"，在连接到服务端的众多socket中，只有需要进行IO操作的才能获取服务端的处理线程进行IO。这样就不会因为线程不够用而限制了socket的接入。客户端的socket连接到服务端时，就会在事件分离器注册一个 IO请求事件 和 IO 事件处理器。在该连接发生IO请求时，IO事件处理器就会启动一个线程来处理这个

IO请求，不断尝试获取系统的IO的使用权限，一旦成功（即：可以进行IO），则通知这个socket进行IO数据传输。

NIO还提供了两个新概念：Buffer和Channel

Buffer:

- 是一块连续的内存块。
- 是 NIO 数据读或写的中转地。

Channel:

- 数据的源头或者数据的目的地
- 用于向 buffer 提供数据或者读取 buffer 数据 ,buffer 对象的唯一接口。
- 异步 I/O 支持

Buffer作为IO流中数据的缓冲区，而Channel则作为socket的IO流与Buffer的传输通道。客户端socket与服务端socket之间的IO传输不直接把数据交给CPU使用，
而是先经过Channel通道把数据保存到Buffer，然后CPU直接从Buffer区读写数据，一次可以读写更多的内容。

使用Buffer提高IO效率的原因（这里与IO流里面的BufferedXXStream、BufferedReader、BufferedWriter提高性能的原理一样）：IO的耗时主要花在数据传输的路上，普通的IO是一个字节一个字节地传输，
而采用了Buffer的话，通过Buffer封装的方法（比如一次读一行，则以行为单位传输而不是一个字节一次进行传输）就可以实现"一大块字节"的传输。比如：IO就是送快递，普通IO是一个快递跑一趟，采用了Buffer的IO就是一车跑一趟。很明显，buffer效率更高，花在传输路上
的时间大大缩短。

四：异步阻塞IO（AIO）

NIO是同步的IO，是因为程序需要IO操作时，必须获得了IO权限后亲自进行IO操作才能进行下一步操作。AIO是对NIO的改进（所以AIO又叫NIO.2），它是基于Proactor模型的。每个socket连接在事件分离器注册 IO完成事件 和 IO完成事件处理器。程序需要进行IO时，向分离器发出IO请求并把所用的Buffer区域告知分离器，分离器通知操作系统进行IO操作，操作系统自己不断尝试获取IO权限并进行IO操作（数据保存在Buffer区），操作完成后通知分离器；分离器检测到 IO完成事件，则激活 IO完成事件处理器，处理器会通知程序说"IO已完成"，程序知道后就直接从Buffer区进行数据的读写。

也就是说：**AIO是发出IO请求后，由操作系统自己去获取IO权限并进行IO操作；NIO则是发出IO请求后，由线程不断尝试获取IO权限，获取到后通知应用程序自己进行IO操作。**