# BIO和伪异步

BIO的主要问题是每当有一个新的client请求接入时，服务器端必须创建一个新的线程处理接入的客户端链路，一个线程只能处理一个客户端连接。

为了改进一线程一连接模型，后来演进出了通过**线程池**或者消息队列实现1或者多线程处理N个client的模型，但是它的底层通行机制依然使用的是同步阻塞I/O，所以被称为“伪异步”。

*//bio，服务器端为每个连接创建一个线程处理*

while (true){

socket = server.**accept**();

new **Thread**(new **TimeServerHandler**(socket)).**start**();

}

*//伪异步，服务器端从线程池中选一个线程处理连接*

while (true){

socket = server.**accept**();

executePool.**execute**(new **TimeServerHandler**(socket));

}

伪异步利用线程池，避免了为每个请求都创建一个独立线程造成线程资源耗尽问题，但是由于底层还是**同步阻塞模型**，无法从根本上解决问题。

在java同步I/O中，比如下面InputStream中定义的read方法，对Socket的输入流进行读取操作，除非：

1. 有数据可读；
2. 可用数据已经读取完毕；
3. 发生异常。

否则这个方法会一直阻塞下去。

*//This method blocks until input data is available, end of file is detected, or an exception is thrown.*

public int **read**(byte b[]) throws IOException {

return **read**(b, 0, b.length);

}

当一方发送请求或者应答消息比较缓慢，或者网络延迟比较高时，读取输入流的一方的通信线程将被长时间阻塞。比如服务器端调用readLine方法接收客户端的消息，将一直阻塞到消息接收完成或者发生异常，在这期间其他消息只能在消息队列中排队。

将消息或者任务放到**线程池**后就返回，不再直接访问I/O线程或者进行I/O读写，伪异步通过这种方式不会使得业务线程不会被阻塞。想这种通过线程池做缓冲区的做法一般称为伪异步I/O。但是这种其实还是阻塞的，这种方式解决的是每个请求建立一个独立线程造成资源耗尽的问题，从线程池拿到一个线程之后的处理还是同步阻塞的。

# NIO

NIO的编程要比BIO复杂的多，因为使用要通过轮询时间来做相应的处理，但是客户端发起连接操作是异步的，通过在多路复用器注册后等待结果就可以了，不用像在BIO中因为同步被阻塞。

NIO主要是Non-Blocking，通过在多路复用器注册以及轮询selectedKey，在没有可读写的数据时不需要同步等待，使得I/O线程可以处理其他的链路。

## Buffer

在NIO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；在写入数据，写入到缓存区中。任何时候访问NIO中的数据，都是通过缓冲区进行操作。

## Channel

channel是一个双向的通道，网络数据通过channel读取和写入。通道和流的不同之处在于通道是双向的，流只能在一个方向移动（一个流只能是InputSteam或者OutputStream的子类）。

## Selector

多路复用器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发生读、写事件，这个Channel就处于就绪状态，会被Selector轮询出来，进行后续的I/O操作。

多路复用的核心就是Selector轮询注册在其上的channel，当发现某个或者多个channel处于就绪态之后，从阻塞态返回就绪的channel的选择键集合进行I/O操作。

# AIO

AIO编程模型比NIO简单，不需要使用多路复用器Selector对注册的通道进行轮询操作就可以实现异步读写，是真正的异步非阻塞I/O。

# 几种I/O模型对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 同步阻塞I/O(BIO) | 伪异步I/O | 非阻塞I/O(NIO) | 异步I/O(AIO) |
| 客户端数：I/O线程数 | 1:1 | M:N | M:1(1个I/O线程可以处理多个客户端连接) | M:0(可以不启动额外的I/O线程) |
| 是否阻塞 | 阻塞 | 阻塞 | 非阻塞 | 非阻塞 |
| 是否同步 | 同步 | 同步 | 同步 | 异步 |