1

第1篇

IO 基础篇

第 1 章　Java IO 演进之路

1

第1章

Java IO 演进之路

课程目标

1、掌握 Java 中 BIO、NIO、AIO 之间的区别及应用场景。

2、透彻理解阻塞(Block)与非阻塞(Non-Block)区别。

3、透彻理解同步(Synchronization)和异步(Asynchronous)的区别。

内容定位

1.适合具有网络通信开发经验的人群。

2.适合具有 1-3 年 Java Web 开发经验的人群。

1.1 必须明白的几个概念

1.1.1 阻塞(Block)和非阻塞(Non-Block)

阻塞和非阻塞是进程在访问数据的时候，数据是否准备就绪的一种处理方式,当数据没有准备的时候。

阻塞：往往需要等待缓冲区中的数据准备好过后才处理其他的事情，否则一直等待在那里。

非阻塞:当我们的进程访问我们的数据缓冲区的时候，如果数据没有准备好则直接返回，不会等待。如果数据已经准备好，也直接返回。

1.1.2 同步(Synchronization)和异步(Asynchronous)

同步和异步都是基于应用程序和操作系统处理 IO 事件所采用的方式。比如同步：是应用程序要直接参与 IO 读写的操作。异步：所有的 IO 读写交给操作系统去处理，应用程序只需要等待通知。

同步方式在处理 IO 事件的时候，必须阻塞在某个方法上面等待我们的 IO 事件完成(阻塞 IO 事件或者通过轮询 IO 事件的方式),对于异步来说，所有的 IO 读写都交给了操作系统。这个时候，我们可以去做其他的事情，并不需要去完成真正的 IO 操作，当操作完成 IO 后，会给我们的应用程序一个通知。

同步 ： 阻塞到 IO 事件，阻塞到 read 或则 write。这个时候我们就完全不能做自己的事情。让读写方法加入到线程里面，然后阻塞线程来实现，对线程的性能开销比较大。

1.2 BIO 与 NIO 对比

下表总结了 Java BIO(Block IO)和 NIO(Non-Block IO)之间的主要差别异。

IO模型	BIO	NIO
通信	面向流(乡村公路)	面向缓冲(高速公路，多路复用技术)

处理	阻塞 IO(多线程)	非阻塞 IO(反应堆 Reactor)
触发	无	选择器(轮询机制)

1.2.1 面向流与面向缓冲

Java NIO 和 BIO 之间第一个最大的区别是，BIO 是面向流的，NIO 是面向缓冲区的。 Java BIO 面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO 的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

1.2.2 阻塞与非阻塞

Java BIO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 Java NIO 的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞 IO 的空闲时间用于在其它通道上执行 IO 操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

1.2.3 选择器的问世

Java NIO 的选择器(Selector)允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来"选择"通道 :这些通道里已经有可以处理的输入 ,或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

1.2.4 NIO 和 BIO 如何影响应用程序的设计

无论您选择 BIO 或 NIO 工具箱，可能会影响您应用程序设计的以下几个方面：

A.对 NIO 或 BIO 类的 API 调用。

B.数据处理逻辑。

C.用来处理数据的线程数。

1.API 调用

当然，使用 NIO 的 API 调用时看起来与使用 BIO 时有所不同，但这并不意外，因为并不是仅从一个 InputStream 逐字节读取，而是数据必须先读入缓冲区再处理。

2.数据处理

使用纯粹的 NIO 设计相较 BIO 设计，数据处理也受到影响。

在 BIO 设计中，我们从 InputStream 或 Reader 逐字节读取数据。假设你正在处理一基于行的文本数据流，例如：

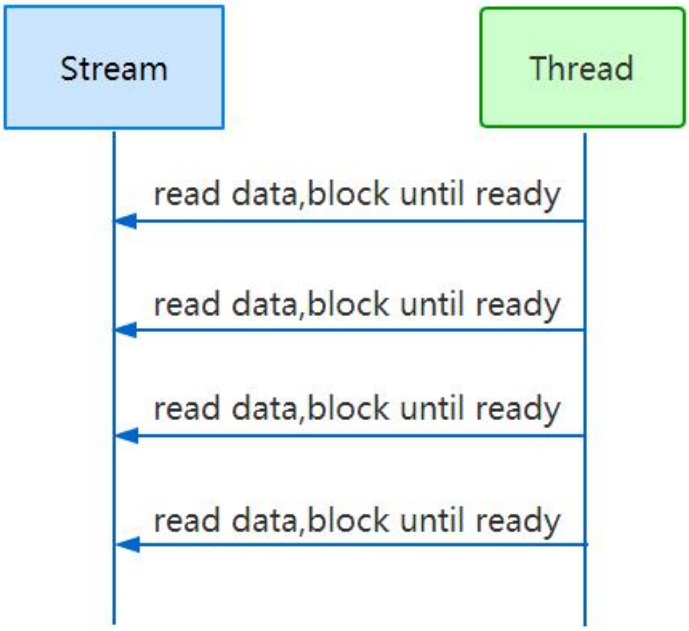
有如下一段文本：

```
Name:Tom
Age:18
Email: tom@qq.com
Phone:13888888888
```

该文本行的流可以这样处理：

```
FileInputStream input = new FileInputStream("d://info.txt");
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String nameLine = reader.readLine();
String ageLine = reader.readLine();
String emailLine = reader.readLine();
String phoneLine = reader.readLine();
```

请注意处理状态由程序执行多久决定。换句话说，一旦 reader.readLine()方法返回，你就知道肯定文本行就已读完，readline()阻塞直到整行读完，这就是原因。你也知道此行包含名称；同样，第二个 readline()调用返回的时候，你知道这行包含年龄等。 正如你可以看到，该处理程序仅在有新数据读入时运行，并知道每步的数据是什么。一旦正在运行的线程已处理过读入的某些数据，该线程不会再回退数据（大多如此）。下图也说明了这条原则：



（Java BIO：从一个阻塞的流中读数据） 而一个 NIO 的实现会有所不同，下面是一个简单的例子：

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
```

注意第二行，从通道读取字节到 ByteBuffer。当这个方法调用返回时，你不知道你所需的所有数据是否在缓冲区内。你所知道的是，该缓冲区包含一些字节，这使得处理有点困难。

假设第一次 read(buffer)调用后，读入缓冲区的数据只有半行，例如，“Name:An”，你能处理数据吗？显然不能，需要等待，直到整行数据读入缓存，在此之前，对数据的任何处理毫无意义。

所以，你怎么知道是否该缓冲区包含足够的数据可以处理呢？好了，你不知道。发现的方法只能查看缓冲区中的数据。其结果是，在你知道所有数据都在缓冲区里之前，你必须检查几次缓冲区的数据。这不仅效率低下，而且可以使程序设计方案杂乱不堪。例如：

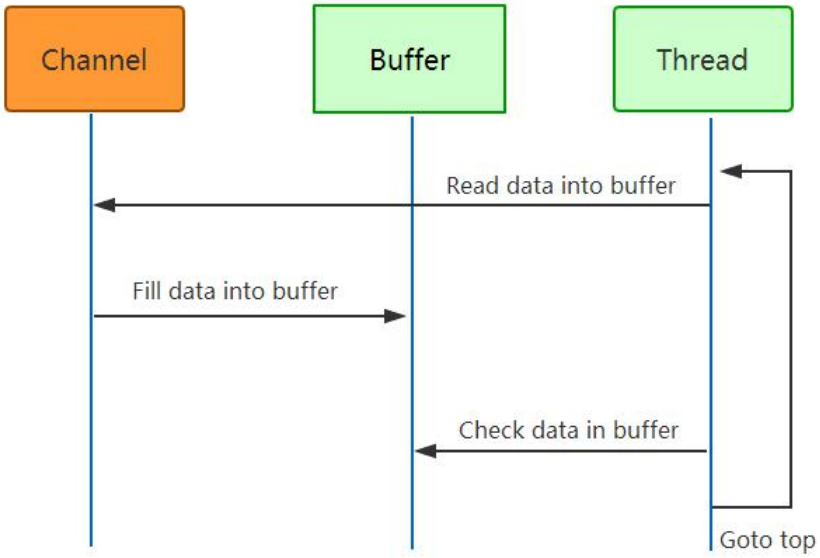
```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
while(!bufferFull(bytesRead)) {
    bytesRead = inChannel.read(buffer);
}
```

bufferFull()方法必须跟踪有多少数据读入缓冲区，并返回真或假，这取决于缓冲区是否已满。换句话说，如果缓冲

区准备好被处理，那么表示缓冲区满了。

bufferFull()方法扫描缓冲区，但必须保持在 bufferFull()方法被调用之前状态相同。如果没有，下一个读入缓冲区的数据可能无法读到正确的位置。这是不可能的，但却是需要注意的又一问题。

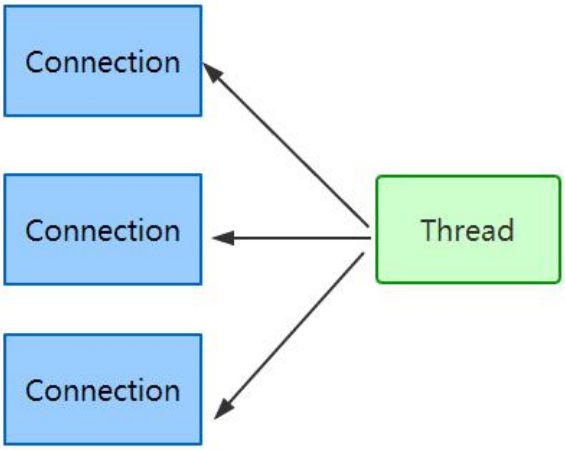
如果缓冲区已满，它可以被处理。如果它不满，并且在你的实际案例中有意义，你或许能处理其中的部分数据。但是许多情况下并非如此。下图展示了"缓冲区数据循环就绪"：



3. 设置处理线程数

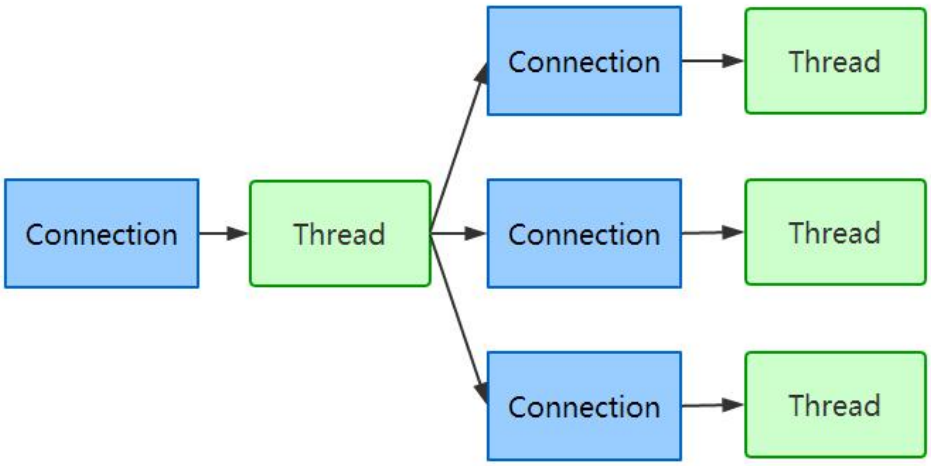
NIO 可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的代价是解析数据可能会比从一个阻塞流中读取数据更复杂。

如果需要管理同时打开的成千上万个连接，这些连接每次只是发送少量的数据，例如聊天服务器，实现 NIO 的服务器可能是一个优势。同样，如果你需要维持许多打开的连接到其他计算机上，如 P2P 网络中，使用一个单独的线程来管理你所有出站连接，可能是一个优势。一个线程多个连接的设计方案如：



Java NIO：单线程管理多个连接

如果你有少量的连接使用非常高的带宽，一次发送大量的数据，也许典型的 IO 服务器实现可能非常契合。下图说明了一个典型的 IO 服务器设计：



Java BIO：一个典型的 IO 服务器设计- 一个连接通过一个线程处理。

1.4 Java AIO 详解

jdk1.7 (NIO2)才是实现真正的异步 AIO、把 IO 读写操作完全交给操作系统，学习了 linux epoll 模式，下面我们来做一

些演示。

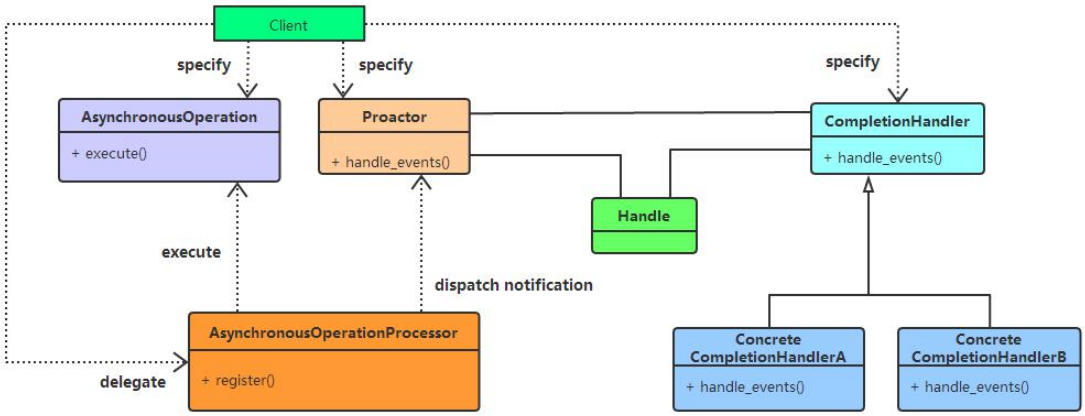
1.4.1 AIO（Asynchronous IO）基本原理

服务端:AsynchronousServerSocketChannel

客服端:AsynchronousSocketChannel

用户处理器:CompletionHandler 接口,这个接口实现应用程序向操作系统发起 IO 请求,当完成后处理具体逻辑，否则做自己该做的事情，

“真正”的异步IO需要操作系统更强的支持。在IO多路复用模型中，事件循环将文件句柄的状态事件通知给用户线程，由用户线程自行读取数据、处理数据。而在异步IO模型中，当用户线程收到通知时，数据已经被内核读取完毕，并放在了用户线程指定的缓冲区内，内核在IO完成后通知用户线程直接使用即可。异步IO模型使用了Proactor设计模式实现了这一机制，如下图所示：



1.4.2 AIO 初体验

服务端代码：

```
package com.gupaoedu.vip.netty.io.aio;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousChannelGroup;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * AIO服务端
 */
public class AIOServer {

    private final int port;

    public static void main(String args[]) {
        int port = 8000;
        new AIOServer(port);
    }

    public AIOServer(int port) {
        this.port = port;
        listen();
    }

    private void listen() {
        try {
            ExecutorService executorService = Executors.newCachedThreadPool();
            AsynchronousChannelGroup threadGroup = AsynchronousChannelGroup.withCachedThreadPool(executorService, 1);
            final AsynchronousServerSocketChannel server = AsynchronousServerSocketChannel.open(threadGroup);
            server.bind(new InetSocketAddress(port));
            System.out.println("服务已启动，监听端口" + port);

            server.accept(null, new CompletionHandler<AsynchronousSocketChannel, Object>(){
                final ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
                public void completed(AsynchronousSocketChannel result, Object attachment){
                    System.out.println("IO操作成功，开始获取数据");
                    try {
                        buffer.clear();
                        result.read(buffer).get();
                        buffer.flip();
                        result.write(buffer);
                        buffer.flip();
                    } catch (Exception e) {
                        System.out.println(e.toString());
                    } finally {
                        try {
                            result.close();
                            server.accept(null, this);
                        } catch (Exception e) {
                            System.out.println(e.toString());
                        }
                    }

                    System.out.println("操作完成");
                }

                @Override
                public void failed(Throwable exc, Object attachment) {
                    System.out.println("IO操作是失败: " + exc);
```

```
            }
        });

        try {
            Thread.sleep(Integer.MAX_VALUE);
        } catch (InterruptedException ex) {
            System.out.println(ex);
        }
    } catch (IOException e) {
        System.out.println(e);
    }
  }
}
```

客户端代码：

```
package com.gupaoedu.vip.netty.io.aio;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;

/**
 * AIO 客户端
 */
public class AIOClient {
    private final AsynchronousSocketChannel client;

    public AIOClient() throws Exception{
        client = AsynchronousSocketChannel.open();
    }

    public void connect(String host,int port)throws Exception{
        client.connect(new InetSocketAddress(host,port),null,new CompletionHandler<Void,Void>() {
            @Override
            public void completed(Void result, Void attachment) {
                try {
                    client.write(ByteBuffer.wrap("这是一条测试数据".getBytes())).get();
                    System.out.println("已发送至服务器");
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }

            @Override
            public void failed(Throwable exc, Void attachment) {
                exc.printStackTrace();
            }
        });
        final ByteBuffer bb = ByteBuffer.allocate(1024);
        client.read(bb, null, new CompletionHandler<Integer,Object>(){

                @Override
                public void completed(Integer result, Object attachment) {
                    System.out.println("IO 操作完成" + result);
                    System.out.println("获取反馈结果" + new String(bb.array()));
                }

                @Override
                public void failed(Throwable exc, Object attachment) {
                    exc.printStackTrace();
                }
            }
```

```
        );

        try {
            Thread.sleep(Integer.MAX_VALUE);
        } catch (InterruptedException ex) {
            System.out.println(ex);
        }

    }

    public static void main(String args[])throws Exception{
        new AIOClient().connect("localhost",8000);
    }
}
```

执行结果：

服务端

客户端

1.5 各 IO 模型对比与总结

最后再来一张表总结

属性	同步阻塞 IO(BIO)	伪异步 IO	非阻塞 IO（NIO）	异步 IO(AIO)
客户端数:IO 线程数	1:1	M:N(M>=N)	M:1	M:0
阻塞类型	阻塞	阻塞	非阻塞	非阻塞
同步	同步	同步	同步(多路复用)	异步
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