Table of Contents

本文将介绍**非阻塞 IO** 和**异步 IO**，也就是大家耳熟能详的 NIO 和 AIO。很多初学者可能分不清楚异步和非阻塞的区别，只是在各种场合能听到**异步非阻塞**这个词。

本文会先介绍并演示阻塞模式，然后引入非阻塞模式来对阻塞模式进行优化，最后再介绍 JDK7 引入的异步 IO，由于网上关于异步 IO 的介绍相对较少，所以这部分内容我会介绍得具体一些。

希望看完本文，读者可以对非阻塞 IO 和异步 IO 的迷雾看得更清晰些，或者为初学者解开一丝丝疑惑也是好的。

阻塞模式 IO

我们已经介绍过使用 Java NIO 包组成一个简单的**客户端-服务端**网络通讯所需要的 ServerSocketChannel、SocketChannel 和 Buffer，我们这里整合一下它们，给出一个完整的可运行的例子：

```
public class Server {

    public static void main(String[] args) throws IOException {

        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

        // 监听 8080 端口进来的 TCP 链接
        serverSocketChannel.socket().bind(new InetSocketAddress(8080));

        while (true) {
```

```
            // 这里会阻塞，直到有一个请求的连接进来
            SocketChannel socketChannel = serverSocketChannel.accept();

            // 开启一个新的线程来处理这个请求，然后在 while 循环中继续监听 8080
端口
            SocketHandler handler = new SocketHandler(socketChannel);
            new Thread(handler).start();
        }
    }
}
```

这里看一下新的线程需要做什么，SocketHandler：

```
public class SocketHandler implements Runnable {

    private SocketChannel socketChannel;

    public SocketHandler(SocketChannel socketChannel) {
        this.socketChannel = socketChannel;
    }

    @Override
    public void run() {

        ByteBuffer buffer = ByteBuffer.allocate(1024);
        try {
            // 将请求数据读入 Buffer 中
            int num;
            while ((num = socketChannel.read(buffer)) > 0) {
                // 读取 Buffer 内容之前先 flip 一下
                buffer.flip();

                // 提取 Buffer 中的数据
                byte[] bytes = new byte[num];
                buffer.get(bytes);

                String re = new String(bytes, "UTF-8");
                System.out.println("收到请求：" + re);

                // 回应客户端
                ByteBuffer writeBuffer = ByteBuffer.wrap(("我已经收到你的请求，
你的请求内容是：" + re).getBytes());
                socketChannel.write(writeBuffer);

                buffer.clear();
            }
        } catch (IOException e) {
            IOUtils.closeQuietly(socketChannel);
        }
    }
}
```

最后，贴一下客户端 SocketChannel 的使用，客户端比较简单：

```
public class SocketChannelTest {
    public static void main(String[] args) throws IOException {
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.connect(new InetSocketAddress("localhost", 8080));

        // 发送请求
        ByteBuffer buffer = ByteBuffer.wrap("1234567890".getBytes());
        socketChannel.write(buffer);

        // 读取响应
        ByteBuffer readBuffer = ByteBuffer.allocate(1024);
        int num;
        if ((num = socketChannel.read(readBuffer)) > 0) {
            readBuffer.flip();

            byte[] re = new byte[num];
            readBuffer.get(re);

            String result = new String(re, "UTF-8");
            System.out.println("返回值: " + result);
        }
    }
}
```

上面介绍的阻塞模式的代码应该很好理解：来一个新的连接，我们就新开一个线程来处理这个连接，之后的操作全部由那个线程来完成。

那么，这个模式下的性能瓶颈在哪里呢？

1. 首先，每次来一个连接都开一个新的线程这肯定是不合适的。当活跃连接数在几十几百的时候当然是可以这样做的，但如果活跃连接数是几万几十万的时候，这么多线程明显就不行了。每个线程都需要一部分内存，内存会被迅速消耗，同时，线程切换的开销非常大。
2. 其次，阻塞操作在这里也是一个问题。首先，accept() 是一个阻塞操作，当 accept() 返回的时候，代表有一个连接可以使用了，我们这里是马上就新建线程来处理这个 SocketChannel 了，但是，但是这里不代表对方就将数据传输过来了。所以，SocketChannel#read 方法将阻塞，等待数据，明显这个等待是不值得的。同理，write 方法也需要等待通道可写才能执行写入操作，这边的阻塞等待也是不值得的。

非阻塞 IO

说完了阻塞模式的使用及其缺点以后，我们这里就可以介绍非阻塞 IO 了。

非阻塞 IO 的核心在于使用一个 Selector 来管理多个通道，可以是 SocketChannel，也可以是 ServerSocketChannel，将各个通道注册到 Selector 上，指定监听的事件。

之后可以只用一个线程来轮询这个 Selector，看看上面是否有通道是准备好的，当通道准备好可读或可写，然后才去开始真正的读写，这样速度就很快了。我们就完全没有必要给每个通道都起一个线程。

NIO 中 Selector 是对底层操作系统实现的一个抽象，管理通道状态其实都是底层系统实现的，这里简单介绍下在不同系统下的实现。

select：上世纪 80 年代就实现了，它支持注册 FD_SETSIZE(1024) 个 socket，在那个年代肯定是够用的，不过现在嘛，肯定是不行了。

poll：1997 年，出现了 poll 作为 select 的替代者，最大的区别就是，poll 不再限制 socket 数量。

select 和 poll 都有一个共同的问题，那就是**它们都只会告诉你有几个通道准备好了，但是不会告诉你具体是哪几个通道。**所以，一旦知道有通道准备好以后，自己还是需要进行一次扫描，显然这个不太好，通道少的时候还行，一旦通道的数量是几十万个以上的时候，扫描一次的时间都很可观了，时间复杂度 O(n)。所以，后来才催生了以下实现。

epoll：2002 年随 Linux 内核 2.5.44 发布，epoll 能直接返回具体的准备好的通道，时间复杂度 O(1)。

除了 Linux 中的 epoll，2000 年 FreeBSD 出现了 **Kqueue**，还有就是，Solaris 中有 **/dev/poll**。

> 前面说了那么多实现，但是没有出现 Windows，Windows 平台的非阻塞 IO 使用 select，我们也不必觉得 Windows 很落后，在 Windows 中 IOCP 提供的异步 IO 是比较强大的。

我们回到 Selector，毕竟 JVM 就是这么一个屏蔽底层实现的平台，**我们面向 Selector 编程就可以了。**

之前在介绍 Selector 的时候已经了解过了它的基本用法，这边来一个可运行的实例代码，大家不妨看看：

```
public class SelectorServer {

    public static void main(String[] args) throws IOException {
        Selector selector = Selector.open();

        ServerSocketChannel server = ServerSocketChannel.open();
        server.socket().bind(new InetSocketAddress(8080));

        // 将其注册到 Selector 中，监听 OP_ACCEPT 事件
        server.configureBlocking(false);
        server.register(selector, SelectionKey.OP_ACCEPT);

        while (true) {
            int readyChannels = selector.select();
            if (readyChannels == 0) {
```

```
                continue;
            }
            Set<SelectionKey> readyKeys = selector.selectedKeys();
            // 遍历
            Iterator<SelectionKey> iterator = readyKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey key = iterator.next();
                iterator.remove();

                if (key.isAcceptable()) {
                    // 有已经接受的新的到服务端的连接
                    SocketChannel socketChannel = server.accept();

                    // 有新的连接并不代表这个通道就有数据，
                    // 这里将这个新的 SocketChannel 注册到 Selector，监听
OP_READ 事件，等待数据
                    socketChannel.configureBlocking(false);
                    socketChannel.register(selector, SelectionKey.OP_READ);
                } else if (key.isReadable()) {
                    // 有数据可读
                    // 上面一个 if 分支中注册了监听 OP_READ 事件的 SocketChannel
                    SocketChannel socketChannel = (SocketChannel)
key.channel();
                    ByteBuffer readBuffer = ByteBuffer.allocate(1024);
                    int num = socketChannel.read(readBuffer);
                    if (num > 0) {
                        // 处理进来的数据...
                        System.out.println("收到数据：" + new
String(readBuffer.array()).trim());
                        ByteBuffer buffer = ByteBuffer.wrap("返回给客户端的数
据...".getBytes());
                        socketChannel.write(buffer);
                    } else if (num == -1) {
                        // -1 代表连接已经关闭
                        socketChannel.close();
                    }
                }
            }
        }
    }
}
```

至于客户端，大家可以继续使用上一节介绍阻塞模式时的客户端进行测试。

NIO.2 异步 IO

More New IO，或称 NIO.2，随 JDK 1.7 发布，包括了引入异步 IO 接口和 Paths 等文件访问接口。

异步这个词，我想对于绝大多数开发者来说都很熟悉，很多场景下我们都会使用异步。

通常，我们会有一个线程池用于执行异步任务，提交任务的线程将任务提交到线程池就可以立马返回，不必等到任务真正完成。如果想要知道任务的执行结果，通常是通过传递一个回调函数的方式，任务结束后去调用这个函数。

同样的原理，Java 中的异步 IO 也是一样的，都是由一个线程池来负责执行任务，然后使用回调或自己去查询结果。

大部分开发者都知道为什么要这么设计了，这里再啰嗦一下。异步 IO 主要是为了控制线程数量，减少过多的线程带来的内存消耗和 CPU 在线程调度上的开销。

在 Unix/Linux 等系统中，JDK 使用了并发包中的线程池来管理任务，具体可以查看 AsynchronousChannelGroup 的源码。

在 Windows 操作系统中，提供了一个叫做 I/O Completion Ports 的方案，通常简称为 **IOCP**，操作系统负责管理线程池，其性能非常优异，所以**在 Windows 中 JDK 直接采用了 IOCP 的支持**，使用系统支持，把更多的操作信息暴露给操作系统，也使得操作系统能够对我们的 IO 进行一定程度的优化。

> 在 Linux 中其实也是有异步 IO 系统实现的，但是限制比较多，性能也一般，所以 JDK 采用了自建线程池的方式。

本文还是以实用为主，想要了解更多信息请自行查找其他资料，下面对 Java 异步 IO 进行实践性的介绍。

总共有三个类需要我们关注，分别是 **AsynchronousSocketChannel**，**AsynchronousServerSocketChannel** 和 **AsynchronousFileChannel**，只不过是在之前介绍的 FileChannel、SocketChannel 和 ServerSocketChannel 的类名上加了个前缀 **Asynchronous**。

Java 异步 IO 提供了两种使用方式，分别是返回 Future 实例和使用回调函数。

1、返回 Future 实例

返回 java.util.concurrent.Future 实例的方式我们应该很熟悉，JDK 线程池就是这么使用的。Future 接口的几个方法语义在这里也是通用的，这里先做简单介绍。

- future.isDone();

  判断操作是否已经完成，包括了**正常完成、异常抛出、取消**

- future.cancel(true);

  取消操作，方式是中断。参数 true 说的是，即使这个任务正在执行，也会进行中断。

- future.isCancelled();

  是否被取消，只有在任务正常结束之前被取消，这个方法才会返回 true

- future.get();

  这是我们的老朋友，获取执行结果，阻塞。

- future.get(10, TimeUnit.SECONDS);

  如果上面的 get() 方法的阻塞你不满意，那就设置个超时时间。

2、提供 CompletionHandler 回调函数

java.nio.channels.CompletionHandler 接口定义：

```
public interface CompletionHandler<V,A> {

    void completed(V result, A attachment);

    void failed(Throwable exc, A attachment);
}
```

> 注意，参数上有个 attachment，虽然不常用，我们可以在各个支持的方法中传递这个参数值

```
AsynchronousServerSocketChannel listener =
AsynchronousServerSocketChannel.open().bind(null);

// accept 方法的第一个参数可以传递 attachment
listener.accept(attachment, new CompletionHandler<AsynchronousSocketChannel,
Object>() {
    public void completed(
      AsynchronousSocketChannel client, Object attachment) {
          //
      }
    public void failed(Throwable exc, Object attachment) {
          //
      }
});
```

AsynchronousFileChannel

网上关于 Non-Blocking IO 的介绍文章很多，但是 Asynchronous IO 的文章相对就少得多了，所以我这边会多介绍一些相关内容。

首先，我们就来关注异步的文件 IO，前面我们说了，文件 IO 在所有的操作系统中都不支持非阻塞模式，但是我们可以对文件 IO 采用异步的方式来提高性能。

下面，我会介绍 AsynchronousFileChannel 里面的一些重要的接口，都很简单，读者要是觉得无趣，直接滑到下一个标题就可以了。

实例化：

```
AsynchronousFileChannel channel =
AsynchronousFileChannel.open(Paths.get("/Users/hongjie/test.txt"));
```

一旦实例化完成，我们就可以着手准备将数据读入到 Buffer 中：

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
Future<Integer> result = channel.read(buffer, 0);
```

> 异步文件通道的读操作和写操作都需要提供一个文件的开始位置，文件开始位置为 0

除了使用返回 Future 实例的方式，也可以采用回调函数进行操作，接口如下：

```
public abstract <A> void read(ByteBuffer dst,
                              long position,
                              A attachment,
                              CompletionHandler<Integer,? super A> handler);
```

顺便也贴一下写操作的两个版本的接口：

```
public abstract Future<Integer> write(ByteBuffer src, long position);

public abstract <A> void write(ByteBuffer src,
                               long position,
                               A attachment,
                               CompletionHandler<Integer,? super A> handler);
```

我们可以看到，AIO 的读写主要也还是与 Buffer 打交道，这个与 NIO 是一脉相承的。

另外，还提供了用于将内存中的数据刷入到磁盘的方法：

```
public abstract void force(boolean metaData) throws IOException;
```

> 因为我们对文件的写操作，操作系统并不会直接针对文件操作，系统会缓存，然后周期性地刷入到磁盘。如果希望将数据及时写入到磁盘中，以免断电引发部分数据丢失，可以调用此方法。参数如果设置为 true，意味着同时也将文件属性信息更新到磁盘。

还有，还提供了对文件的锁定功能，我们可以锁定文件的部分数据，这样可以进行排他性的操作。

```
public abstract Future<FileLock> lock(long position, long size, boolean
shared);
```

> position 是要锁定内容的开始位置，size 指示了要锁定的区域大小，shared 指示需要的是共享锁还是排他锁

当然，也可以使用回调函数的版本：

```
public abstract <A> void lock(long position,
                              long size,
                              boolean shared,
                              A attachment,
                              CompletionHandler<FileLock,? super A> handler);
```

文件锁定功能上还提供了 tryLock 方法，此方法会快速返回结果：

```
public abstract FileLock tryLock(long position, long size, boolean shared)
    throws IOException;
```

> 这个方法很简单，就是尝试去获取锁，如果该区域已被其他线程或其他应用锁住，那么立刻返回 null，否则返回 FileLock 对象。

AsynchronousFileChannel 操作大体上也就以上介绍的这些接口，还是比较简单的，这里就少一些废话早点结束好了。

AsynchronousServerSocketChannel

这个类对应的是非阻塞 IO 的 ServerSocketChannel，大家可以类比下使用方式。

我们就废话少说，用代码说事吧：

```
package com.javadoop.aio;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;

public class Server {

    public static void main(String[] args) throws IOException {

          // 实例化，并监听端口
        AsynchronousServerSocketChannel server =
```

```
                AsynchronousServerSocketChannel.open().bind(new
InetSocketAddress(8080));

        // 自己定义一个 Attachment 类，用于传递一些信息
        Attachment att = new Attachment();
        att.setServer(server);

        server.accept(att, new CompletionHandler<AsynchronousSocketChannel,
Attachment>() {
            @Override
            public void completed(AsynchronousSocketChannel client, Attachment
att) {
                try {
                    SocketAddress clientAddr = client.getRemoteAddress();
                    System.out.println("收到新的连接：" + clientAddr);

                    // 收到新的连接后，server 应该重新调用 accept 方法等待新的连
接进来
                    att.getServer().accept(att, this);

                    Attachment newAtt = new Attachment();
                    newAtt.setServer(server);
                    newAtt.setClient(client);
                    newAtt.setReadMode(true);
                    newAtt.setBuffer(ByteBuffer.allocate(2048));

                    // 这里也可以继续使用匿名实现类，不过代码不好看，所以这里专门
定义一个类
                    client.read(newAtt.getBuffer(), newAtt, new
ChannelHandler());
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            }

            @Override
            public void failed(Throwable t, Attachment att) {
                System.out.println("accept failed");
            }
        });
        // 为了防止 main 线程退出
        try {
            Thread.currentThread().join();
        } catch (InterruptedException e) {
        }
    }
}
```

看一下 ChannelHandler 类：

```
package com.javadoop.aio;

import java.io.IOException;
```

```
import java.nio.ByteBuffer;
import java.nio.channels.CompletionHandler;
import java.nio.charset.Charset;

public class ChannelHandler implements CompletionHandler<Integer, Attachment>
{

    @Override
    public void completed(Integer result, Attachment att) {
        if (att.isReadMode()) {
            // 读取来自客户端的数据
            ByteBuffer buffer = att.getBuffer();
            buffer.flip();
            byte bytes[] = new byte[buffer.limit()];
            buffer.get(bytes);
            String msg = new String(buffer.array()).toString().trim();
            System.out.println("收到来自客户端的数据: " + msg);

            // 响应客户端请求，返回数据
            buffer.clear();
            buffer.put("Response from server!".getBytes(Charset.forName("UTF-
8")));
            att.setReadMode(false);
            buffer.flip();
            // 写数据到客户端也是异步
            att.getClient().write(buffer, att, this);
        } else {
            // 到这里，说明往客户端写数据也结束了，有以下两种选择:
            // 1\. 继续等待客户端发送新的数据过来
//            att.setReadMode(true);
//            att.getBuffer().clear();
//            att.getClient().read(att.getBuffer(), att, this);
            // 2\. 既然服务端已经返回数据给客户端，断开这次的连接
            try {
                att.getClient().close();
            } catch (IOException e) {
            }
        }
    }

    @Override
    public void failed(Throwable t, Attachment att) {
        System.out.println("连接断开");
    }
}
```

顺便再贴一下自定义的 Attachment 类:

```
public class Attachment {
    private AsynchronousServerSocketChannel server;
    private AsynchronousSocketChannel client;
    private boolean isReadMode;
    private ByteBuffer buffer;
```

```
    // getter & setter
}
```

这样，一个简单的服务端就写好了，接下来可以接收客户端请求了。上面我们用的都是回调函数的方式，读者要是感兴趣，可以试试写个使用 Future 的。

AsynchronousSocketChannel

其实，说完上面的 AsynchronousServerSocketChannel，基本上读者也就知道怎么使用 AsynchronousSocketChannel 了，和非阻塞 IO 基本类似。

这边做个简单演示，这样读者就可以配合之前介绍的 Server 进行测试使用了。

```
package com.javadoop.aio;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.charset.Charset;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class Client {

    public static void main(String[] args) throws Exception {
        AsynchronousSocketChannel client = AsynchronousSocketChannel.open();
          // 来个 Future 形式的
        Future<?> future = client.connect(new InetSocketAddress(8080));
        // 阻塞一下，等待连接成功
        future.get();

        Attachment att = new Attachment();
        att.setClient(client);
        att.setReadMode(false);
        att.setBuffer(ByteBuffer.allocate(2048));
        byte[] data = "I am obot!".getBytes();
        att.getBuffer().put(data);
        att.getBuffer().flip();

        // 异步发送数据到服务端
        client.write(att.getBuffer(), att, new ClientChannelHandler());

        // 这里休息一下再退出，给出足够的时间处理数据
        Thread.sleep(2000);
    }
}
```

往里面看下 ClientChannelHandler 类：

```
package com.javadoop.aio;

import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.CompletionHandler;
import java.nio.charset.Charset;

public class ClientChannelHandler implements CompletionHandler<Integer,
Attachment> {

    @Override
    public void completed(Integer result, Attachment att) {
        ByteBuffer buffer = att.getBuffer();
        if (att.isReadMode()) {
            // 读取来自服务端的数据
            buffer.flip();
            byte[] bytes = new byte[buffer.limit()];
            buffer.get(bytes);
            String msg = new String(bytes, Charset.forName("UTF-8"));
            System.out.println("收到来自服务端的响应数据: " + msg);

            // 接下来，有以下两种选择:
            // 1\. 向服务端发送新的数据
//            att.setReadMode(false);
//            buffer.clear();
//            String newMsg = "new message from client";
//            byte[] data = newMsg.getBytes(Charset.forName("UTF-8"));
//            buffer.put(data);
//            buffer.flip();
//            att.getClient().write(buffer, att, this);
            // 2\. 关闭连接
            try {
                att.getClient().close();
            } catch (IOException e) {
            }
        } else {
            // 写操作完成后，会进到这里
            att.setReadMode(true);
            buffer.clear();
            att.getClient().read(buffer, att, this);
        }
    }

    @Override
    public void failed(Throwable t, Attachment att) {
        System.out.println("服务器无响应");
    }
}
```

以上代码都是可以运行调试的，如果读者碰到问题，请在评论区留言。

Asynchronous Channel Groups

为了知识的完整性，有必要对 group 进行介绍，其实也就是介绍 AsynchronousChannelGroup 这个类。之前我们说过，异步 IO 一定存在一个线程池，这个线程池负责接收任务、处理 IO 事件、回调等。这个线程池就在 group 内部，group 一旦关闭，那么相应的线程池就会关闭。

AsynchronousServerSocketChannels 和 AsynchronousSocketChannels 是属于 group 的，当我们调用 AsynchronousServerSocketChannel 或 AsynchronousSocketChannel 的 open() 方法的时候，相应的 channel 就属于默认的 group，这个 group 由 JVM 自动构造并管理。

如果我们想要配置这个默认的 group，可以在 JVM 启动参数中指定以下系统变量：

- java.nio.channels.DefaultThreadPool.threadFactory

  此系统变量用于设置 ThreadFactory，它应该是 java.util.concurrent.ThreadFactory 实现类的全限定类名。一旦我们指定了这个 ThreadFactory 以后，group 中的线程就会使用该类产生。

- java.nio.channels.DefaultThreadPool.initialSize

  此系统变量也很好理解，用于设置线程池的初始大小。

可能你会想要使用自己定义的 group，这样可以对其中的线程进行更多的控制，使用以下几个方法即可：

- AsynchronousChannelGroup.withCachedThreadPool(ExecutorService executor, int initialSize)
- AsynchronousChannelGroup.withFixedThreadPool(int nThreads, ThreadFactory threadFactory)
- AsynchronousChannelGroup.withThreadPool(ExecutorService executor)

熟悉线程池的读者对这些方法应该很好理解，它们都是 AsynchronousChannelGroup 中的静态方法。

至于 group 的使用就很简单了，代码一看就懂：

```
AsynchronousChannelGroup group = AsynchronousChannelGroup
        .withFixedThreadPool(10, Executors.defaultThreadFactory());
AsynchronousServerSocketChannel server =
AsynchronousServerSocketChannel.open(group);
AsynchronousSocketChannel client = AsynchronousSocketChannel.open(group);
```

AsynchronousFileChannels 不属于 group。但是它们也是关联到一个线程池的，如果不指定，会使用系统默认的线程池，如果想要使用指定的线程池，可以在实例化的时候使用以下方法：

```
public static AsynchronousFileChannel open(Path file,
                                           Set<? extends OpenOption> options,
                                           ExecutorService executor,
                                           FileAttribute<?>... attrs) {
    ...
}
```

到这里，异步 IO 就算介绍完成了。

小结

我想，本文应该是说清楚了非阻塞 IO 和异步 IO 了，对于异步 IO，由于网上的资料比较少，所以不免篇幅多了些。

我们也要知道，看懂了这些，确实可以学到一些东西，多了解一些知识，但是我们还是很少在工作中将这些知识变成工程代码。一般而言，我们需要在网络应用中使用 NIO 或 AIO 来提升性能，但是，在工程上，绝不是了解了一些概念，知道了一些接口就可以的，需要处理的细节还非常多。

这也是为什么 Netty/Mina 如此盛行的原因，因为它们帮助封装好了很多细节，提供给我们用户友好的接口，后面有时间我也会对 Netty 进行介绍。

（全文完）