# Netty 是什么?

在开始了解 Netty 是什么之前，我们先来回顾一下，如果我们需要实现一个客户端与服务端通信的程序，使用传统的 IO 编程，应该如何来实现?

## IO编程

我们简化下场景：客户端每隔两秒发送一个带有时间戳的 "hello world" 给服务端，服务端收到之后打印。

为了方便演示，下面例子中，服务端和客户端各一个类，把这两个类拷贝到你的 IDE 中，先后运行 `IOServer.java` 和 `IOClient.java`可看到效果。

下面是传统的 IO 编程中服务端实现

IOServer.java

```
/**
 * @author 闪电侠
 */
public class IOServer {
    public static void main(String[] args) throws
Exception {

        ServerSocket serverSocket = new
ServerSocket(8000);

        // (1) 接收新连接线程
```

```
        new Thread(() -> {
            while (true) {
                try {
                    // (1) 阻塞方法获取新的连接
                    Socket socket =
serverSocket.accept();

                    // (2) 每一个新的连接都创建一个线
程，负责读取数据
                    new Thread(() -> {
                        try {
                            int len;
                            byte[] data = new
byte[1024];
                            InputStream
inputStream = socket.getInputStream();
                            // (3) 按字节流方式读取
数据
                            while ((len =
inputStream.read(data)) != -1) {

System.out.println(new String(data, 0, len));
                            }
                        } catch (IOException e) {
                        }
                    }).start();

                } catch (IOException e) {
                }

            }
        }).start();
    }
```

```
}
```

Server 端首先创建了一个serverSocket来监听 8000 端口，然后创建一个线程，线程里面不断调用阻塞方法 serversocket.accept();获取新的连接，见(1)，当获取到新的连接之后，给每条连接创建一个新的线程，这个线程负责从该连接中读取数据，见(2)，然后读取数据是以字节流的方式，见(3)。

下面是传统的IO编程中客户端实现

IOClient.java

```
/**
 * @author 闪电侠
 */
public class IOClient {

    public static void main(String[] args) {
        new Thread(() -> {
            try {
                Socket socket = new
Socket("127.0.0.1", 8000);
                while (true) {
                    try {

socket.getOutputStream().write((new Date() + ":
hello world").getBytes());
                        Thread.sleep(2000);
                    } catch (Exception e) {
                    }
                }
            } catch (IOException e) {
            }
        }).start();
    }
}
```

客户端的代码相对简单，连接上服务端 8000 端口之后，每隔 2 秒，我们向服务端写一个带有时间戳的 "hello world"。

IO 编程模型在客户端较少的情况下运行良好，但是对于客户端比较多的业务来说，单机服务端可能需要支撑成千上万的连接，IO 模型可能就不太合适了，我们来分析一下原因。

上面的 demo，从服务端代码中我们可以看到，在传统的 IO 模型中，每个连接创建成功之后都需要一个线程来维护，每个线程包含一个 while 死循环，那么 1w 个连接对应 1w 个线程，继而 1w 个 while 死循环，这就带来如下几个问题：

1. 线程资源受限：线程是操作系统中非常宝贵的资源，同一时刻有大量的线程处于阻塞状态是非常严重的资源浪费，操作系统耗不起
2. 线程切换效率低下：单机 CPU 核数固定，线程爆炸之后操作系统频繁进行线程切换，应用性能急剧下降。
3. 除了以上两个问题，IO 编程中，我们看到数据读写是以字节流为单位。

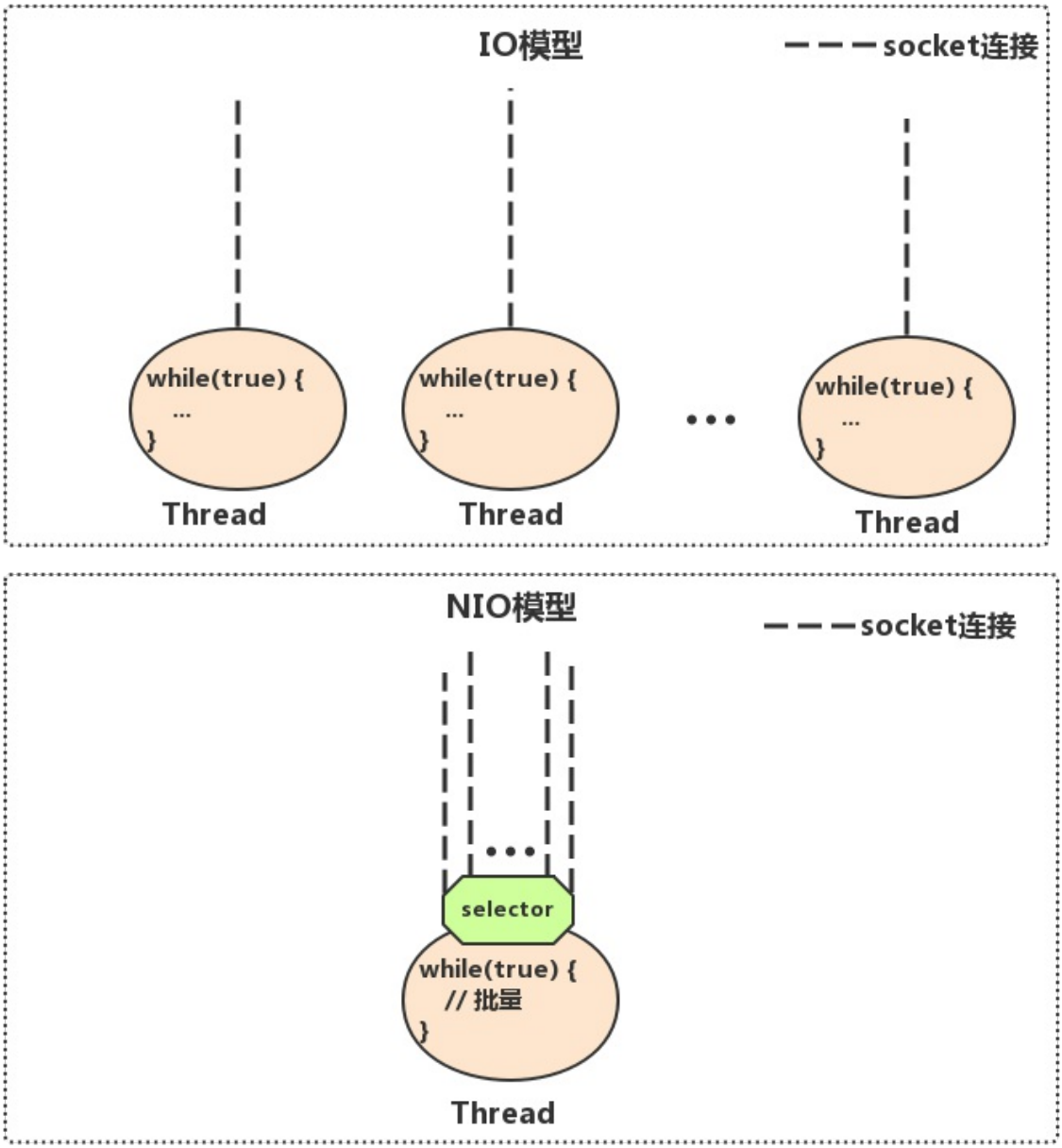
为了解决这三个问题，JDK 在 1.4 之后提出了 NIO。

# NIO 编程

关于 NIO 相关的文章网上也有很多，这里不打算详细深入分析，下面简单描述一下 NIO 是如何解决以上三个问题的。

## 线程资源受限

NIO 编程模型中，新来一个连接不再创建一个新的线程，而是可以把这条连接直接绑定到某个固定的线程，然后这条连接所有的读写都由这个线程来负责，那么他是怎么做到的？我们用一幅图来对比一下 IO 与 NIO




如上图所示，IO 模型中，一个连接来了，会创建一个线程，对应一个 while 死循环，死循环的目的就是不断监测这条连接上是否有数据可以读，大多数情况下，1w 个连接里面同一时刻只有少量的连接有数据可读，因此，很多个 while 死循环都白白浪费掉了，因为读不出啥数据。

而在 NIO 模型中，他把这么多 while 死循环变成一个死循环，这个死循环由一个线程控制，那么他又是如何做到一个线程，一个 while 死循环就能监测1w个连接是否有数据可读的呢?

这就是 NIO 模型中 selector 的作用，一条连接来了之后，现在不创建一个 while 死循环去监听是否有数据可读了，而是直接把这条连接注册到 selector 上，然后，通过检查这个 selector，就可以批量监测出有数据可读的连接，进而读取数据，下面我再举个非常简单的生活中的例子说明 IO 与 NIO 的区别。

在一家幼儿园里，小朋友有上厕所的需求，小朋友都太小以至于你要问他要不要上厕所，他才会告诉你。幼儿园一共有 100 个小朋友，有两种方案可以解决小朋友上厕所的问题：

1. 每个小朋友配一个老师。每个老师隔段时间询问小朋友是否要上厕所，如果要上，就领他去厕所，100 个小朋友就需要 100 个老师来询问，并且每个小朋友上厕所的时候都需要一个老师领着他去上，这就是IO模型，一个连接对应一个线程。
2. 所有的小朋友都配同一个老师。这个老师隔段时间询问所有的小朋友是否有人要上厕所，然后每一时刻把所有要上厕所的小朋友批量领到厕所，这就是 NIO 模型，所有小朋友都注册到同一个老师，对应的就是所有的连接都注册到一个线程，然后批量轮询。

这就是 NIO 模型解决线程资源受限的方案，实际开发过程中，我们会开多个线程，每个线程都管理着一批连接，相对于 IO 模型中一个线程管理一条连接，消耗的线程资源大幅减少

## 线程切换效率低下

由于 NIO 模型中线程数量大大降低，线程切换效率因此也大幅度提高

## IO读写面向流

IO 读写是面向流的，一次性只能从流中读取一个或者多个字节，并且读完之后流无法再读取，你需要自己缓存数据。
而 NIO 的读写是面向 Buffer 的，你可以随意读取里面任何一个字节

数据，不需要你自己缓存数据，这一切只需要移动读写指针即可。

简单讲完了 JDK NIO 的解决方案之后，我们接下来使用 NIO 的方案替换掉 IO 的方案，我们先来看看，如果用 JDK 原生的 NIO 来实现服务端，该怎么做

> 前方高能预警：以下代码可能会让你感觉极度不适，如有不适，请跳过

NIOServer.java

```
/**
 * @author 闪电侠
 */
public class NIOServer {
    public static void main(String[] args) throws
IOException {
        Selector serverSelector =
Selector.open();
        Selector clientSelector =
Selector.open();

        new Thread(() -> {
            try {
                // 对应IO编程中服务端启动
                ServerSocketChannel
listenerChannel = ServerSocketChannel.open();
                listenerChannel.socket().bind(new
InetSocketAddress(8000));

listenerChannel.configureBlocking(false);
```

```
listenerChannel.register(serverSelector,
SelectionKey.OP_ACCEPT);

                while (true) {
                    // 监测是否有新的连接，这里的1指的
是阻塞的时间为 1ms
                    if (serverSelector.select(1)
> 0) {
                        Set<SelectionKey> set =
serverSelector.selectedKeys();
                        Iterator<SelectionKey>
keyIterator = set.iterator();

                        while
(keyIterator.hasNext()) {
                            SelectionKey key =
keyIterator.next();

                            if
(key.isAcceptable()) {
                                try {
                                    // (1) 每来一个
新连接，不需要创建一个线程，而是直接注册到clientSelector
                                    SocketChannel
clientChannel = ((ServerSocketChannel)
key.channel()).accept();

clientChannel.configureBlocking(false);

clientChannel.register(clientSelector,
SelectionKey.OP_READ);
                                } finally {
```

```
keyIterator.remove();
                                }
                            }

                        }
                    }
                }
            } catch (IOException ignored) {
            }

        }).start();


        new Thread(() -> {
            try {
                while (true) {
                    // (2) 批量轮询是否有哪些连接有数据可读，这里的1指的是阻塞的时间为 1ms
                    if (clientSelector.select(1) > 0) {
                        Set<SelectionKey> set = clientSelector.selectedKeys();
                        Iterator<SelectionKey> keyIterator = set.iterator();

                        while (keyIterator.hasNext()) {
                            SelectionKey key = keyIterator.next();

                            if (key.isReadable()) {
```

```
                                try {
                                    SocketChannel
clientChannel = (SocketChannel) key.channel();
                                    ByteBuffer
byteBuffer = ByteBuffer.allocate(1024);
                                    // (3) 面向
Buffer

clientChannel.read(byteBuffer);

byteBuffer.flip();

System.out.println(Charset.defaultCharset().newDe
coder().decode(byteBuffer)

.toString());
                                } finally {

keyIterator.remove();

key.interestOps(SelectionKey.OP_READ);
                                }
                            }

                        }
                    }
                }
            } catch (IOException ignored) {
            }
        }).start();


    }
```

```
}
```

相信大部分没有接触过 NIO 的同学应该会直接跳过代码来到这一行：原来使用 JDK 原生 NIO 的 API 实现一个简单的服务端通信程序是如此复杂!

复杂得我都没耐心解释这一坨代码的执行逻辑(开个玩笑)，我们还是先对照 NIO 来解释一下几个核心思路

1. NIO 模型中通常会有两个线程，每个线程绑定一个轮询器 selector ，在我们这个例子中`serverSelector`负责轮询是否有新的连接，`clientSelector`负责轮询连接是否有数据可读
2. 服务端监测到新的连接之后，不再创建一个新的线程，而是直接将新连接绑定到`clientSelector`上，这样就不用 IO 模型中 1w 个 while 循环在死等，参见(1)
3. `clientSelector`被一个 while 死循环包裹着，如果在某一时刻有多条连接有数据可读，那么通过`clientSelector.select(1)`方法可以轮询出来，进而批量处理，参见(2)
4. 数据的读写面向 Buffer，参见(3)

其他的细节部分，我不愿意多讲，因为实在是太复杂，你也不用对代码的细节深究到底。总之，强烈不建议直接基于JDK原生NIO来进行网络开发，下面是我总结的原因

1. JDK 的 NIO 编程需要了解很多的概念，编程复杂，对 NIO 入门非常不友好，编程模型不友好，ByteBuffer 的 Api 简直反人类
2. 对 NIO 编程来说，一个比较合适的线程模型能充分发挥它的优势，而 JDK 没有给你实现，你需要自己实现，就连简单的自定义协议拆包都要你自己实现
3. JDK 的 NIO 底层由 epoll 实现，该实现饱受诟病的空轮询 bug 会导致 cpu 飙升 100%
4. 项目庞大之后，自行实现的 NIO 很容易出现各类 bug，维护

成本较高，上面这一坨代码我都不能保证没有 bug

正因为如此，我客户端代码都懒得写给你看了==!，你可以直接使用IOClient.java与NIOServer.java通信

JDK 的 NIO 犹如带刺的玫瑰，虽然美好，让人向往，但是使用不当会让你抓耳挠腮，痛不欲生，正因为如此，Netty 横空出世！

# Netty编程

那么 Netty 到底是何方神圣?
用一句简单的话来说就是：Netty 封装了 JDK 的 NIO，让你用得更爽，你不用再写一大堆复杂的代码了。
用官方正式的话来说就是：Netty 是一个异步事件驱动的网络应用框架，用于快速开发可维护的高性能服务器和客户端。

下面是我总结的使用 Netty 不使用 JDK 原生 NIO 的原因

1. 使用 JDK 自带的NIO需要了解太多的概念，编程复杂，一不小心 bug 横飞
2. Netty 底层 IO 模型随意切换，而这一切只需要做微小的改动，改改参数，Netty可以直接从 NIO 模型变身为 IO 模型
3. Netty 自带的拆包解包，异常检测等机制让你从NIO的繁重细节中脱离出来，让你只需要关心业务逻辑
4. Netty 解决了 JDK 的很多包括空轮询在内的 Bug
5. Netty 底层对线程，selector 做了很多细小的优化，精心设计的 reactor 线程模型做到非常高效的并发处理
6. 自带各种协议栈让你处理任何一种通用协议都几乎不用亲自动手
7. Netty 社区活跃，遇到问题随时邮件列表或者 issue
8. Netty 已经历各大 RPC 框架，消息中间件，分布式通信中间件线上的广泛验证，健壮性无比强大

看不懂没有关系，这些我们在后续的课程中我们都可以学到，接下来我们用 Netty 的版本来重新实现一下本文开篇的功能吧

首先，引入 Maven 依赖，本文后续 Netty 都是基于 4.1.6.Final 版本

```
    <dependency>
        <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.6.Final</version>
    </dependency>
```

然后，下面是服务端实现部分

NettyServer.java

```
/**
 * @author 闪电侠
 */
public class NettyServer {
    public static void main(String[] args) {
        ServerBootstrap serverBootstrap = new
ServerBootstrap();

        NioEventLoopGroup boss = new
NioEventLoopGroup();
        NioEventLoopGroup worker = new
NioEventLoopGroup();
        serverBootstrap
                .group(boss, worker)

.channel(NioServerSocketChannel.class)
```

```
                .childHandler(new
ChannelInitializer<NioSocketChannel>() {
                    protected void
initChannel(NioSocketChannel ch) {
                        ch.pipeline().addLast(new
StringDecoder());
                        ch.pipeline().addLast(new
SimpleChannelInboundHandler<String>() {
                            @Override
                            protected void
channelRead0(ChannelHandlerContext ctx, String
msg) {

System.out.println(msg);
                            }
                        });
                    }
                })
                .bind(8000);
    }
}
```

这么一小段代码就实现了我们前面 NIO 编程中的所有的功能，包括服务端启动，接受新连接，打印客户端传来的数据，怎么样，是不是比 JDK 原生的 NIO 编程优雅许多?

初学 Netty 的时候，由于大部分人对 NIO 编程缺乏经验，因此，将 Netty 里面的概念与 IO 模型结合起来可能更好理解

1. boss 对应 IOServer.java 中的接受新连接线程，主要负责创建新连接
2. worker 对应 IOServer.java 中的负责读取数据的线程，主

要用于读取数据以及业务逻辑处理

然后剩下的逻辑我在后面的系列文章中会详细分析，你可以先把这段代码拷贝到你的 IDE 里面，然后运行 main 函数

然后下面是客户端 NIO 的实现部分

NettyClient.java

```
/**
 * @author 闪电侠
 */
public class NettyClient {
    public static void main(String[] args) throws
InterruptedException {
        Bootstrap bootstrap = new Bootstrap();
        NioEventLoopGroup group = new
NioEventLoopGroup();

        bootstrap.group(group)
                .channel(NioSocketChannel.class)
                .handler(new
ChannelInitializer<Channel>() {
                    @Override
                    protected void
initChannel(Channel ch) {
                        ch.pipeline().addLast(new
StringEncoder());
                    }
                });

        Channel channel =
bootstrap.connect("127.0.0.1", 8000).channel();

        while (true) {
            channel.writeAndFlush(new Date() + ":
hello world!");
            Thread.sleep(2000);
        }
    }
}
```

在客户端程序中，group对应了我们IOClient.java中 main 函数起的线程，剩下的逻辑我在后面的文章中会详细分析，现在你要做的事情就是把这段代码拷贝到你的 IDE 里面，然后运行 main 函数，最后回到 NettyServer.java 的控制台，你会看到效果。

使用 Netty 之后是不是觉得整个世界都美好了，一方面 Netty 对 NIO 封装得如此完美，写出来的代码非常优雅，另外一方面，使用 Netty 之后，网络通信这块的性能问题几乎不用操心，尽情地让 Netty 榨干你的 CPU 吧。