**NIO高级编程与Netty入门概述**

# NIO同步阻塞与同步非阻塞

# BIO与NIO

**IO(BIO)和NIO区别:其本质就是阻塞和非阻塞的区别**

**阻塞概念:应用程序在获取网络数据的时候,如果网络传输数据很慢，就会一直等待,直到传输完毕为止。**

**非阻塞概念:应用程序直接可以获取已经准备就绪好的数据,无需等待。**

IO为同步阻塞形式,NIO为同步非阻塞形式,NIO并没有实现异步,在JDK1.7后升级NIO库包，支持异步非阻塞

同学模型NIO2.0(AIO)

BIO：同步阻塞式IO，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。   
NIO：同步非阻塞式IO，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。   
AIO(NIO.2)：异步非阻塞式IO，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。

同步时，应用程序会直接参与IO读写操作,并且我们的应用程序会直接阻塞到某一个方法上,直到数据准备就绪:

或者采用轮训的策略实时检查数据的就绪状态,如果就绪则获取数据.

异步时,则所有的IO读写操作交给操作系统,与我们的应用程序没有直接关系，我们程序不需要关系IO读写，当操作

系统完成了IO读写操作时,会给我们应用程序发送通知,我们的应用程序直接拿走数据极即可。

## 伪异步

由于BIO一个客户端需要一个线程去处理，因此我们进行优化，后端使用线程池来处理多个客户端的请求接入，形成客户端个数M：线程池最大的线程数N的比例关系，其中M可以远远大于N，通过线程池可以灵活的调配线程资源，设置线程的最大值，防止由于海量并发接入导致线程耗尽。

原理：

当有新的客户端接入时，将客户端的Socket封装成一个Task（该Task任务实现了java的Runnable接口）投递到后端的线程池中进行处理，由于线程池可以设置消息队列的大小以及线程池的最大值，因此，它的资源占用是可控的，无论多少个客户端的并发访问，都不会导致资源的耗尽或宕机。

|  |
| --- |
|  |

## 使用多线程支持多个请求

服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善

|  |
| --- |
| //tcp服务器端...  **class** TcpServer {  **public** **static** **void** main(String[] args) **throws** IOException {  System.***out***.println("socket tcp服务器端启动....");  ServerSocket serverSocket = **new** ServerSocket(8080);  // 等待客户端请求  **try** {  **while** (**true**) {  Socket accept = serverSocket.accept();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **try** {  InputStream inputStream = accept.getInputStream();  // 转换成string类型  **byte**[] buf = **new** **byte**[1024];  **int** len = inputStream.read(buf);  String str = **new** String(buf, 0, len);  System.***out***.println("服务器接受客户端内容:" + str);  } **catch** (Exception e) {  // **TODO**: handle exception  }  }  }).start();  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  serverSocket.close();  }  }  }  **public** **class** TcpClient {  **public** **static** **void** main(String[] args) **throws** UnknownHostException, IOException {  System.***out***.println("socket tcp 客户端启动....");  Socket socket = **new** Socket("127.0.0.1", 8080);  OutputStream outputStream = socket.getOutputStream();  outputStream.write("我是蚂蚁课堂".getBytes());  socket.close();  }  } |

## 使用线程池管理线程

|  |
| --- |
| //tcp服务器端...  **class** TcpServer {    **public** **static** **void** main(String[] args) **throws** IOException {  ExecutorService newCachedThreadPool = Executors.*newCachedThreadPool*();  System.***out***.println("socket tcp服务器端启动....");  ServerSocket serverSocket = **new** ServerSocket(8080);  // 等待客户端请求  **try** {  **while** (**true**) {  Socket accept = serverSocket.accept();  //使用线程  newCachedThreadPool.execute(**new** Runnable() {  @Override  **public** **void** run() {  **try** {  InputStream inputStream = accept.getInputStream();  // 转换成string类型  **byte**[] buf = **new** **byte**[1024];  **int** len = inputStream.read(buf);  String str = **new** String(buf, 0, len);  System.***out***.println("服务器接受客户端内容:" + str);  } **catch** (Exception e) {  // **TODO**: handle exception  }  }  });    }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  serverSocket.close();  }  }  }  **public** **class** TcpClient {  **public** **static** **void** main(String[] args) **throws** UnknownHostException, IOException {  System.***out***.println("socket tcp 客户端启动....");  Socket socket = **new** Socket("127.0.0.1", 8080);  OutputStream outputStream = socket.getOutputStream();  outputStream.write("我是蚂蚁课堂".getBytes());  socket.close();  }  } |

## IO模型关系



## 什么是阻塞

阻塞概念:应用程序在获取网络数据的时候,如果网络传输很慢,那么程序就一直等着,直接到传输完毕。

## 什么是非阻塞

应用程序直接可以获取已经准备好的数据,无需等待.

IO为同步阻塞形式,NIO为同步非阻塞形式。NIO没有实现异步,在JDK1.7之后，升级了NIO库包

,支持异步费阻塞通讯模型NIO2.0(AIO)

## NIO非阻塞代码

|  |
| --- |
| **//nio 异步非阻塞**  **class Client {**  **public static void main(String[] args) throws IOException {**  **System.*out*.println("客户端已经启动....");**  **// 1.创建通道**  **SocketChannel sChannel = SocketChannel.*open*(new InetSocketAddress("127.0.0.1", 8080));**  **// 2.切换异步非阻塞**  **sChannel.configureBlocking(false);**  **// 3.指定缓冲区大小**  **ByteBuffer byteBuffer = ByteBuffer.*allocate*(1024);**  **Scanner scanner= new Scanner(System.*in*);**  **while (scanner.hasNext()) {**  **String str=scanner.next();**  **byteBuffer.put((new Date().toString()+"\n"+str).getBytes());**  **// 4.切换读取模式**  **byteBuffer.flip();**  **sChannel.write(byteBuffer);**  **byteBuffer.clear();**  **}**  **sChannel.close();**  **}**  **}**  **// nio**  **class Server {**  **public static void main(String[] args) throws IOException {**  **System.*out*.println("服务器端已经启动....");**  **// 1.创建通道**  **ServerSocketChannel sChannel = ServerSocketChannel.*open*();**  **// 2.切换读取模式**  **sChannel.configureBlocking(false);**  **// 3.绑定连接**  **sChannel.bind(new InetSocketAddress(8080));**  **// 4.获取选择器**  **Selector selector = Selector.*open*();**  **// 5.将通道注册到选择器 "并且指定监听接受事件"**  **sChannel.register(selector, SelectionKey.*OP\_ACCEPT*);**  **// 6. 轮训式 获取选择 "已经准备就绪"的事件**  **while (selector.select() > 0) {**  **// 7.获取当前选择器所有注册的"选择键(已经就绪的监听事件)"**  **Iterator<SelectionKey> it = selector.selectedKeys().iterator();**  **while (it.hasNext()) {**  **// 8.获取准备就绪的事件**  **SelectionKey sk = it.next();**  **// 9.判断具体是什么事件准备就绪**  **if (sk.isAcceptable()) {**  **// 10.若"接受就绪",获取客户端连接**  **SocketChannel socketChannel = sChannel.accept();**  **// 11.设置阻塞模式**  **socketChannel.configureBlocking(false);**  **// 12.将该通道注册到服务器上**  **socketChannel.register(selector, SelectionKey.*OP\_READ*);**  **} else if (sk.isReadable()) {**  **// 13.获取当前选择器"就绪" 状态的通道**  **SocketChannel socketChannel = (SocketChannel) sk.channel();**  **// 14.读取数据**  **ByteBuffer buf = ByteBuffer.*allocate*(1024);**  **int len = 0;**  **while ((len = socketChannel.read(buf)) > 0) {**  **buf.flip();**  **System.*out*.println(new String(buf.array(), 0, len));**  **buf.clear();**  **}**  **}**  **it.remove();**  **}**  **}**  **}**  **}** |

## 选择KEY

1、SelectionKey.OP\_CONNECT

2、SelectionKey.OP\_ACCEPT

3、SelectionKey.OP\_READ

4、SelectionKey.OP\_WRITE

如果你对不止一种事件感兴趣，那么可以用“位或”操作符将常量连接起来，如下：

int interestSet = SelectionKey.OP\_READ | SelectionKey.OP\_WRITE;

在SelectionKey类的源码中我们可以看到如下的4中属性，四个变量用来表示四种不同类型的事件：可读、可写、可连接、可接受连接

# Netty快速入门

## 什么是Netty

 Netty 是一个基于 JAVA NIO 类库的异步通信框架，它的架构特点是：异步非阻塞、基于事件驱动、高性能、高可靠性和高可定制性。

## Netty应用场景

1.分布式开源框架中dubbo、Zookeeper，RocketMQ底层rpc通讯使用就是netty。

2.游戏开发中，底层使用netty通讯。

## 为什么选择netty

在本小节，我们总结下为什么不建议开发者直接使用JDK的NIO类库进行开发的原因：

1)      NIO的类库和API繁杂，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等；

2)      需要具备其它的额外技能做铺垫，例如熟悉Java多线程编程，因为NIO编程涉及到Reactor模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的NIO程序；

3)      可靠性能力补齐，工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等等，NIO编程的特点是功能开发相对容易，但是可靠性能力补齐工作量和难度都非常大；

4)      JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询，最终导致CPU 100%。官方声称在JDK1.6版本的update18修复了该问题，但是直到JDK1.7版本该问题仍旧存在，只不过该bug发生概率降低了一些而已，它并没有被根本解决。该BUG以及与该BUG相关的问题单如下：

## Netty服务器端

|  |
| --- |
| class ServerHandler extends SimpleChannelHandler {    /\*\*  \* 通道关闭的时候触发  \*/  @Override  public void channelClosed(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {  System.*out*.println("channelClosed");  }  /\*\*  \* 必须是连接已经建立,关闭通道的时候才会触发.  \*/  @Override  public void channelDisconnected(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {  super.channelDisconnected(ctx, e);  System.*out*.println("channelDisconnected");  }  /\*\*  \* 捕获异常  \*/  @Override  public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) throws Exception {  super.exceptionCaught(ctx, e);  System.*out*.println("exceptionCaught");  }  /\*\*  \* 接受消息  \*/  public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) throws Exception {  super.messageReceived(ctx, e);  // System.out.println("messageReceived");  System.*out*.println("服务器端收到客户端消息:"+e.getMessage());  //回复内容  ctx.getChannel().write("好的");  }  }  // netty 服务器端  public class NettyServer {  public static void main(String[] args) {  // 创建服务类对象  ServerBootstrap serverBootstrap = new ServerBootstrap();  // 创建两个线程池 分别为监听监听端口 ，nio监听  ExecutorService boos = Executors.*newCachedThreadPool*();  ExecutorService worker = Executors.*newCachedThreadPool*();  // 设置工程 并把两个线程池加入中  serverBootstrap.setFactory(new NioServerSocketChannelFactory(boos, worker));  // 设置管道工厂  serverBootstrap.setPipelineFactory(new ChannelPipelineFactory() {  public ChannelPipeline getPipeline() throws Exception {  ChannelPipeline pipeline = Channels.*pipeline*();  //将数据转换为string类型.  pipeline.addLast("decoder", new StringDecoder());  pipeline.addLast("encoder", new StringEncoder());  pipeline.addLast("serverHandler", new ServerHandler());  return pipeline;  }  });  // 绑定端口号  serverBootstrap.bind(new InetSocketAddress(9090));  System.*out*.println("netty server启动....");  }  } |

## Netty客户端

|  |
| --- |
| package com.itmayiedu;  import java.net.InetSocketAddress;  import java.util.Scanner;  import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  import org.jboss.netty.bootstrap.ClientBootstrap;  import org.jboss.netty.channel.Channel;  import org.jboss.netty.channel.ChannelFuture;  import org.jboss.netty.channel.ChannelHandlerContext;  import org.jboss.netty.channel.ChannelPipeline;  import org.jboss.netty.channel.ChannelPipelineFactory;  import org.jboss.netty.channel.ChannelStateEvent;  import org.jboss.netty.channel.Channels;  import org.jboss.netty.channel.ExceptionEvent;  import org.jboss.netty.channel.MessageEvent;  import org.jboss.netty.channel.SimpleChannelHandler;  import org.jboss.netty.channel.socket.nio.NioClientSocketChannelFactory;  import org.jboss.netty.handler.codec.string.StringDecoder;  import org.jboss.netty.handler.codec.string.StringEncoder;  class ClientHandler extends SimpleChannelHandler {    /\*\*  \* 通道关闭的时候触发  \*/  @Override  public void channelClosed(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {  System.*out*.println("channelClosed");  }  /\*\*  \* 必须是连接已经建立,关闭通道的时候才会触发.  \*/  @Override  public void channelDisconnected(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {  super.channelDisconnected(ctx, e);  System.*out*.println("channelDisconnected");  }  /\*\*  \* 捕获异常  \*/  @Override  public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) throws Exception {  super.exceptionCaught(ctx, e);  System.*out*.println("exceptionCaught");  }  /\*\*  \* 接受消息  \*/  public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) throws Exception {  super.messageReceived(ctx, e);  // System.out.println("messageReceived");  System.*out*.println("服务器端向客户端回复内容:"+e.getMessage());  //回复内容  // ctx.getChannel().write("好的");  }  }  public class NettyClient {  public static void main(String[] args) {  System.*out*.println("netty client启动...");  // 创建客户端类  ClientBootstrap clientBootstrap = new ClientBootstrap();  // 线程池  ExecutorService boos = Executors.*newCachedThreadPool*();  ExecutorService worker = Executors.*newCachedThreadPool*();  clientBootstrap.setFactory(new NioClientSocketChannelFactory(boos, worker));  clientBootstrap.setPipelineFactory(new ChannelPipelineFactory() {  public ChannelPipeline getPipeline() throws Exception {  ChannelPipeline pipeline = Channels.*pipeline*();  // 将数据转换为string类型.  pipeline.addLast("decoder", new StringDecoder());  pipeline.addLast("encoder", new StringEncoder());  pipeline.addLast("clientHandler", new ClientHandler());  return pipeline;  }  });  //连接服务端  ChannelFuture connect = clientBootstrap.connect(new InetSocketAddress("127.0.0.1", 9090));  Channel channel = connect.getChannel();  System.*out*.println("client start");  Scanner scanner= new Scanner(System.*in*);  while (true) {  System.*out*.println("请输输入内容...");  channel.write(scanner.next());  }  }      } |

## Maven坐标

|  |
| --- |
| **<dependency>**  **<groupId>io.netty</groupId>**  **<artifactId>netty</artifactId>**  **<version>3.3.0.Final</version>**  **</dependency>** |