1. 网络编程基础

//上节课

2. 深入BIO与NIO

2.1 BIO

BIO 有的称之为 basic(基本) IO，有的称之为 block(阻塞) IO，主要应用于文件 IO 和网络 IO，这里不再说文件 IO，前置基础资料中有详细说明，本次课程主要讲讲网络 IO。

在 JDK1.4 之前，我们建立网络连接的时候只能采用 BIO，需要先在服务端启动一个ServerSocket，然后在客户端启动 Socket 来对服务端进行通信，默认情况下服务端需要对每个请求建立一个线程等待请求，而客户端发送请求后，先咨询服务端是否有线程响应，如果没有则会一直等待或者遭到拒绝，如果有的话，客户端线程会等待请求结束后才继续执行，这就是阻塞式IO。

接下来通过一个例子复习回顾一下 BIO 的基本用法（基于 TCP）。

```
1  package com.hero.bio;
2
3  import java.io.InputStream;
4  import java.io.OutputStream;
5  import java.net.ServerSocket;
6  import java.net.Socket;
7
8  //BIO 服务器端程序
9  public class TCPServer {
10     public static void main(String[] args) throws Exception {
11         //1.创建ServerSocket对象
12         System.out.println("服务端 启动....");
13         System.out.println("初始化端口 9999 ");
14         ServerSocket ss=new ServerSocket(9999); //端口号
15
16         while (true) {
17             //2.监听客户端
18             Socket s = ss.accept(); //阻塞
19             //3.从连接中取出输入流来接收消息
20             InputStream is = s.getInputStream(); //阻塞
```

```
            byte[] b = new byte[10];
            is.read(b);
            String clientIP = s.getInetAddress().getHostAddress();
            System.out.println(clientIP + "说:" + new String(b).trim());
            //4.从连接中取出输出流并回话
            OutputStream os = s.getOutputStream();
            os.write("没钱".getBytes());
            //5.关闭
            s.close();
        }
    }
}
```

上述代码编写了一个服务器端程序，绑定端口号 9999，accept 方法用来监听客户端连接， 如果没有客户端连接，就一直等待，程序会阻塞到这里。

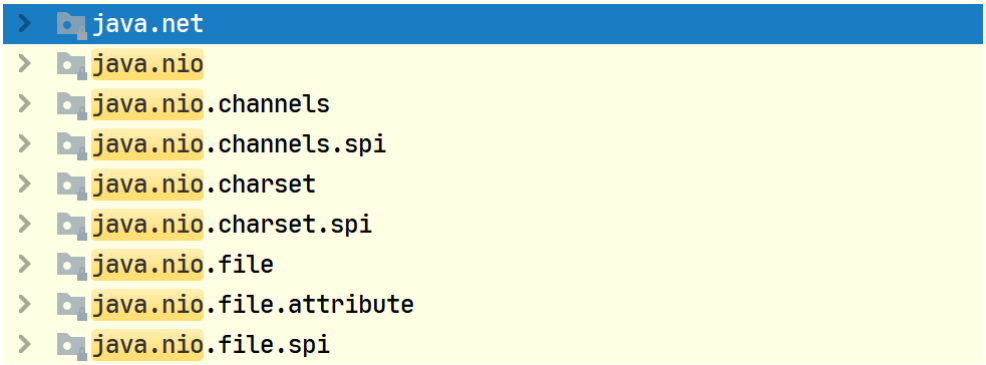
```
package com.hero.bio;

import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Scanner;

//BIO 客户端程序
public class TCPClient {
    public static void main(String[] args) throws Exception {
        while (true) {
            //1.创建Socket对象
            Socket s = new Socket("127.0.0.1", 9999);
            //2.从连接中取出输出流并发消息
            OutputStream os = s.getOutputStream();
            System.out.println("请输入:");
            Scanner sc = new Scanner(System.in);
            String msg = sc.nextLine();
            os.write(msg.getBytes());
            //3.从连接中取出输入流并接收回话
            InputStream is = s.getInputStream(); //阻塞
            byte[] b = new byte[20];
            is.read(b);
            System.out.println("老板说:" + new String(b).trim());
            //4.关闭
            s.close();
        }
    }
}
```

上述代码编写了一个客户端程序，通过 9999 端口连接服务器端，getInputStream 方法用来等待服务器端返回数据，如果没有返回，就一直等待，程序会阻塞到这里。

造成乱码的原因：读取的信息是按照特定编码读取的字节流信息，读取的时候受到读取数量限制，就有可能出现读取的不是一个完整的字节数组信息的情况。

2.2 NIO

2.2.1 概述



java.nio 全称 Java Non-Blocking IO，是指 JDK 提供的新 API。

从 JDK1.4 开始，Java 提供了一系列改进的输入/输出的新特性，被统称为 NIO(即 New IO)。新增了许多用于处理输入输出的类，这些类都被放在 java.nio 包及子包下，并且对原 java.io 包中的很多类进行改写，新增了满足 NIO 的功能。

NIO 和 BIO 有着相同的目的和作用，但是它们的实现方式完全不同；

- BIO 以流的方式处理数据，而 NIO 以块的方式处理数据，块 IO 的效率比流 IO 高很多。
- NIO 是非阻塞式的，这一点跟 BIO 也很不相同，使用它可以提供非阻塞式的高伸缩性网络。

NIO 主要有三大核心部分：

- Channel通道
- Buffer缓冲区
- Selector选择器

传统的 BIO 基于字节流和字符流进行操作，而 NIO 基于 Channel和 Buffer进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector用于监听多个通道的事件（比如：连接请求，数据到达等），因此使用单个线程就可以监听多个客户端通道。

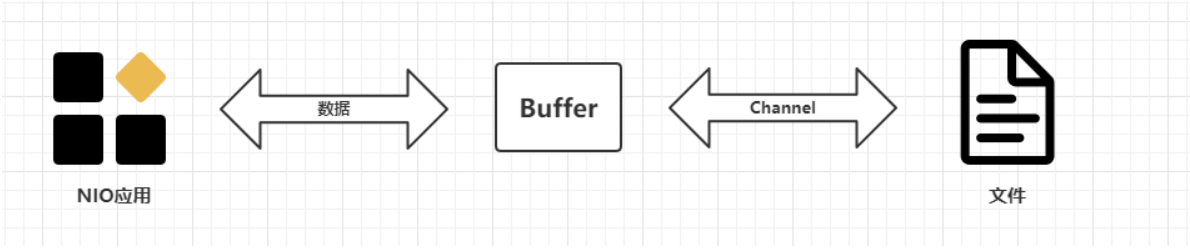
2.2.2 文件 IO

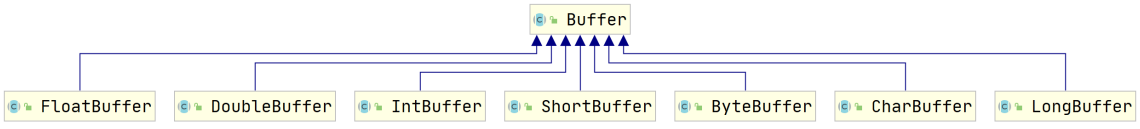
1) 概述和核心 API

缓冲区（Buffer）：实际上是一个容器，是一个特殊的数组，缓冲区对象内置了一些机制，能够跟踪和记录缓冲区的状态变化情况。

Channel 提供从文件、网络读取数据的渠道， 但是读取或写入的数据都必须经由 Buffer

如下图所示：





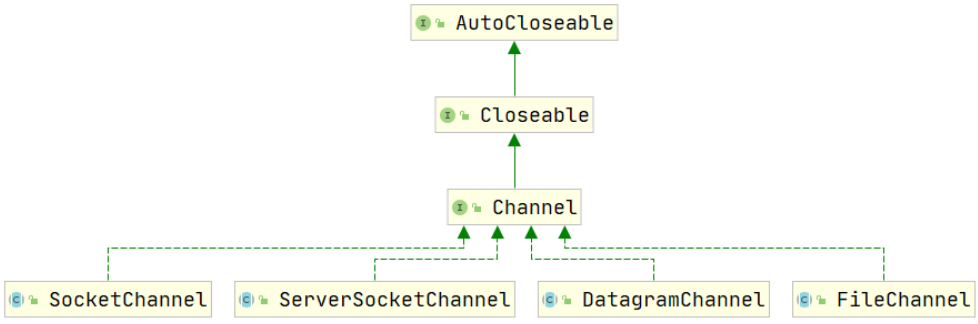
在 NIO 中，Buffer 是一个顶层父类，它是一个抽象类，常用的 Buffer 子类有：

- ByteBuffer，存储字节数据到缓冲区
- ShortBuffer，存储短整型数据到缓冲区
- CharBuffer，存储字符数据到缓冲区
- IntBuffer，存储整数数据到缓冲区
- LongBuffer，存储长整型数据到缓冲区
- DoubleBuffer，存储小数到缓冲区
- FloatBuffer，存储小数到缓冲区

对于 Java 中的基本数据类型， 都有一个 Buffer 类型与之相对应，最常用的自然是ByteBuffer 类（字节缓冲），该类的主要方法如下所示：

- public abstract ByteBuffer put(byte[] b); 存储字节数据到缓冲区
- public abstract byte[] get(); 从缓冲区获得字节数据
- public final byte[] array(); 把缓冲区数据转换成字节数组
- public static ByteBuffer allocate(int capacity); 设置缓冲区的初始容量
- public static ByteBuffer wrap(byte[] array); 把一个现成数组放到缓冲区中使用
- public final Buffer flip(); 翻转缓冲区，重置位置到初始位置（缓冲区有一个指针从头开始读取数据，读到缓冲区尾部时，可以使用这个方法，将指针重新定位到头）

Channel：类似于 BIO 中的 stream，例如 FileInputStream 对象，用来建立到目标（文件，网络套接字，硬件设备等）的一个连接，但是需要注意：BIO 中的 stream 是单向的，例如 FileInputStream 对象只能进行读取数据的操作，而 NIO 中的通道(Channel)是双向的， 既可以用来进行读操作，也可以用来进行写操作。



常用的 Channel 类有：FileChannel、DatagramChannel、ServerSocketChannel 和 SocketChannel。

- FileChannel 用于文件的数据读写
- DatagramChannel 用于 UDP 的数据读写
- ServerSocketChannel 和 SocketChannel 用于 TCP 的数据读写。

这里我们先讲解 FileChannel 类，该类主要用来对本地文件进行 IO 操作，主要方法如下所示：

- public int read(ByteBuffer dst)，从通道读取数据并放到缓冲区中
- public int write(ByteBuffer src)，把缓冲区的数据写到通道中
- public long transferFrom(ReadableByteChannel src, long position, long count)，从目标通道中复制数据到当前通道
- public long transferTo(long position, long count, WritableByteChannel target)，把数据从当前通道复制给目标通道

2) 案例

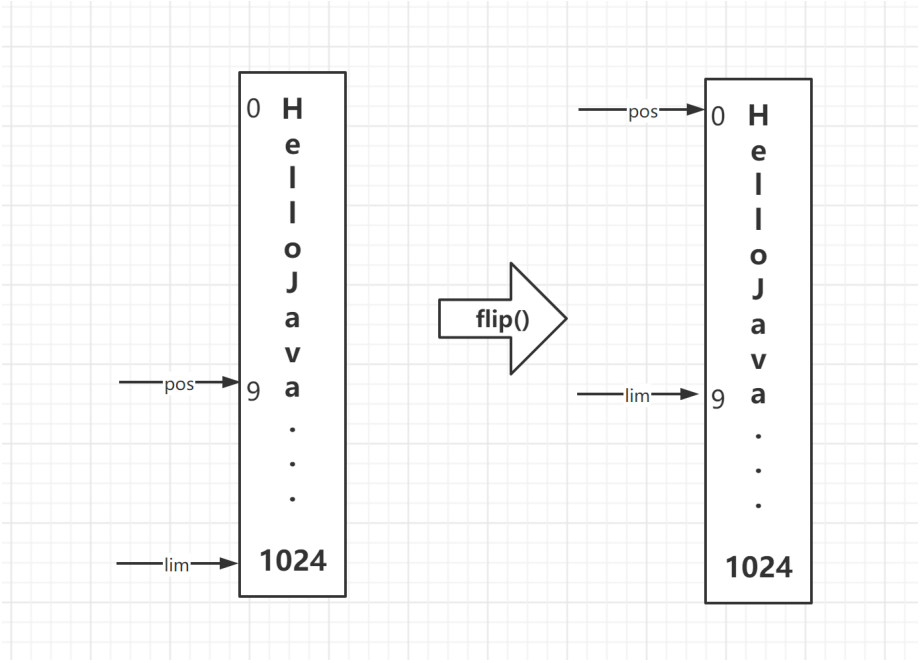
接下来我们通过 NIO 实现几个案例，分别演示一下本地文件的读、写和复制操作，并和 BIO 做个对比。

1.往本地文件中写数据

```
package com.hero.nio.file;

import org.junit.Test;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

//通过NIO实现文件IO
public class TestNIO {
    @Test  //往本地文件中写数据
    public void  test1() throws  Exception{
        //1. 创建输出流
        FileOutputStream fos=new FileOutputStream("basic.txt");
        //2. 从流中得到一个通道
        FileChannel fc=fos.getChannel();
        //3. 提供一个缓冲区
        ByteBuffer buffer=ByteBuffer.allocate(1024);
        //4. 往缓冲区中存入数据
        String str="HelloJava";
        buffer.put(str.getBytes());
        //5. 翻转缓冲区
        buffer.flip();
        //6. 把缓冲区写到通道中
        fc.write(buffer);
        //7. 关闭
        fos.close();
    }
}

```

NIO 中的通道是从输出流对象里通过 getChannel 方法获取到的，该通道是双向的，既可以读，又可以写。在往通道里写数据之前，必须通过 put 方法把数据存到 ByteBuffer 中，然后通过通道的 write 方法写数据。**在 write 之前，需要调用 flip 方法翻转缓冲区**，把内部重置到初始位置，这样在接下来写数据时才能把所有数据写到通道里。

flip()方法的作用：翻转缓冲区，在缓冲区里有一个指针从头（pos）写到尾（lim）。默认的pos是缓冲区内元素size，lim是缓冲区大小。当从缓冲区向通道去写时，是从pos位置去写，写到lim，这样就得不到数据。所以要将pos=lim，pos=0再写。



2.从本地文件中读数据

```
//从本地文件中读取数据
@Test
public void test2() throws  Exception{
    File file=new File("basic.txt");
    //1. 创建输入流
    FileInputStream fis=new FileInputStream(file);
    //2. 得到一个通道
    FileChannel fc=fis.getChannel();
    //3. 准备一个缓冲区
    ByteBuffer buffer=ByteBuffer.allocate((int)file.length());
    //4. 从通道里读取数据并存到缓冲区中
    fc.read(buffer);
    System.out.println(new String(buffer.array()));
    //5. 关闭
    fis.close();
}
```

上述代码从输入流中获得一个通道，然后提供 ByteBuffer 缓冲区，该缓冲区的初始容量和文件的大小一样，最后通过通道的 read 方法把数据读取出来并存储到了 ByteBuffer 中。

3.复制文件

通过 BIO 复制文件

```
@Test  //BIO复制文件
public void test3() throws Exception {
    FileInputStream fis = new FileInputStream("basic.txt");
    FileOutputStream fos = new FileOutputStream("basic2.txt");
    byte[] b = new byte[1024];
    while (true) {
        int res = fis.read(b);
        if (res == -1) {
            break;
```

```
        }
        fos.write(b, 0, res);
    }
    fis.close();
    fos.close();
}

```

上述代码分别通过输入流和输出流实现了文件的复制，是传统的BIO实现。

通过 NIO 复制相同的文件

```
@Test  //使用NIO实现文件复制
public void test4() throws Exception {
    //1. 创建两个流
    FileInputStream fis = new FileInputStream("basic2.txt");
    FileOutputStream fos = new FileOutputStream("basic3.txt");
    //2. 得到两个通道
    FileChannel sourceFC = fis.getChannel();
    FileChannel destFC = fos.getChannel();
    //3. 复制
    destFC.transferFrom(sourceFC, 0, sourceFC.size());
    //4. 关闭
    fis.close();
    fos.close();
}
```

上述代码分别从两个流中得到两个通道，sourceCh 负责读数据，destCh 负责写数据，然后直接调用 transferFrom 方法一步到位实现了文件复制。

2.2.3 网络 IO

1）概述

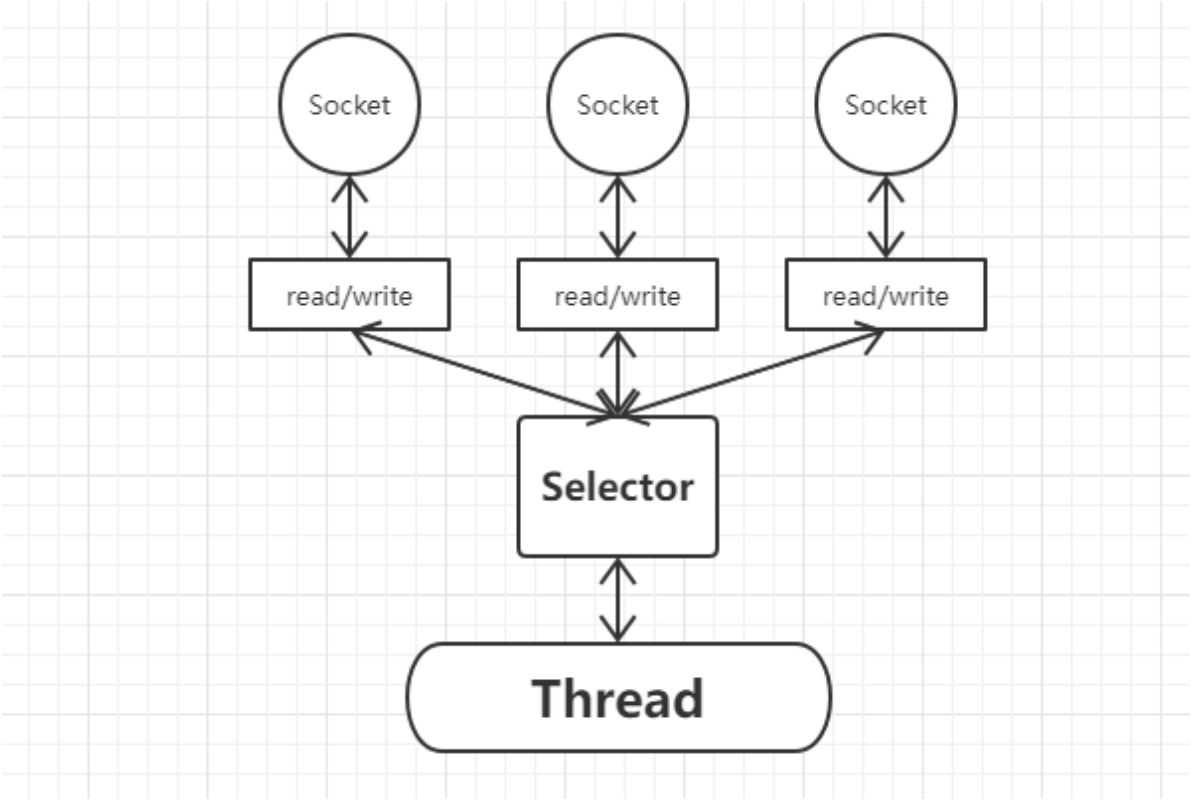
学习 NIO 主要就是进行网络 IO，Java NIO 中的网络通道是**非阻塞 IO 的实现**，**基于事件驱动**，非常适用于服务器需要维持大量连接，但是数据交换量不大的情况，例如：Web服务器、RPC、即时通信...

在 Java 中编写 Socket 服务器，通常有以下几种模式：

- 一个客户端连接用一个线程
  - 优点：程序编写简单
  - 缺点：如果连接非常多，分配的线程也会非常多，服务器可能会因为资源耗尽而崩溃。
- 把每个客户端连接交给一个拥有固定数量线程的连接池
  - 优点：程序编写相对简单， 可以处理大量的连接。
  - 缺点：线程的开销非常大，连接如果非常多，排队现象会比较严重。
- **使用 Java 的 NIO，用非阻塞的 IO 方式处理**
  - **优点：这种模式可以用一个线程，处理大量的客户端连接**
  - 缺点：代码复杂度较高，不易理解

2）Selector选择器

能够检测多个注册的通道上是否有事件发生（读、写、连接），如果有事件发生，便获取事件然后针对每个事件进行相应的处理。这样就可以只用一个单线程去管理多个通道，也就是管理多个连接。这样使得只有在连接真正有读写事件发生时，才会调用函数来进行读写，就大大地减少了系统开销，并且不必为每个连接都创建一个线程，不用去维护多个线程，并且避免了多线程之间的上下文切换导致的开销



该类的常用方法如下所示：

- public static Selector open()，得到一个选择器对象
- public int select(long timeout)，监控所有注册的通道，当其中有 IO 操作可以进行时，将对应的 SelectionKey 加入到内部集合中并返回，参数用来设置超时时间
- public Set selectedKeys()，从内部集合中得到所有的 SelectionKey

3）SelectionKey

代表了 Selector 和网络通道的注册关系

一共四种（就是连接事件）

- int OP_ACCEPT：有新的网络连接可以 accept，值为 16
- int OP_CONNECT：代表连接已经建立，值为 8
- int OP_READ 和 int OP_WRITE：代表了读、写操作，值为 1 和 4

该类的常用方法如下所示：

- public abstract Selector selector()，得到与之关联的 Selector 对象
- public abstract SelectableChannel channel()，得到与之关联的通道
- public final Object attachment()，得到与之关联的共享数据
- public abstract SelectionKey interestOps(int ops)，设置或改变监听事件
- public final boolean isAcceptable()，是否可以 accept
- public final boolean isReadable()，是否可以读
- public final boolean isWritable()，是否可以写

4）ServerSocketChannel

用来在服务器端监听新的客户端 Socket 连接

常用方法如下所示：

- public static ServerSocketChannel open()，得到一个 ServerSocketChannel 通道
- public final ServerSocketChannel bind(SocketAddress local)，设置服务器端端口号
- public final SelectableChannel configureBlocking(boolean block)，设置阻塞或非阻塞模式， 取值 false 表示采用非阻塞模式
- public SocketChannel accept()，接受一个连接，返回代表这个连接的通道对象
- public final SelectionKey register(Selector sel, int ops)，注册一个选择器并设置监听事件

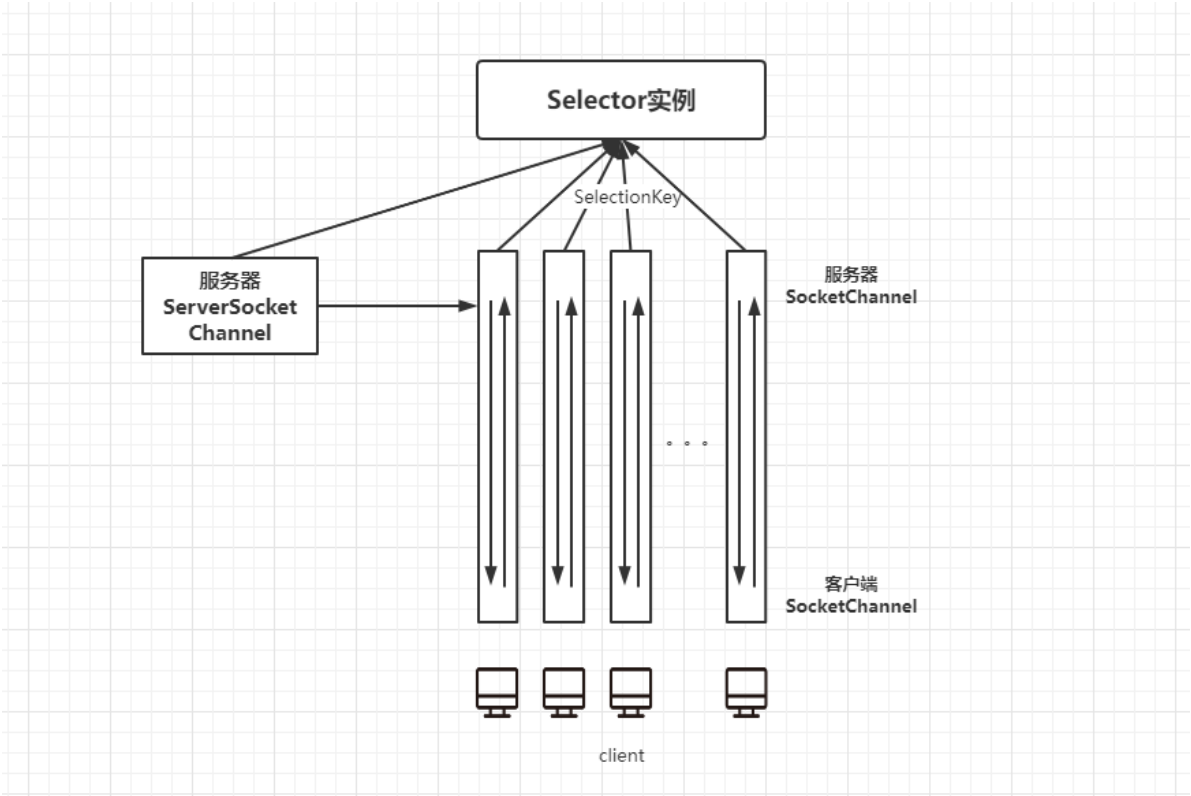
5）SocketChannel

网络 IO 通道，具体负责进行读写操作

NIO 总是把缓冲区的数据写入通道，或者把通道里的数据读到缓冲区。

常用方法如下所示：

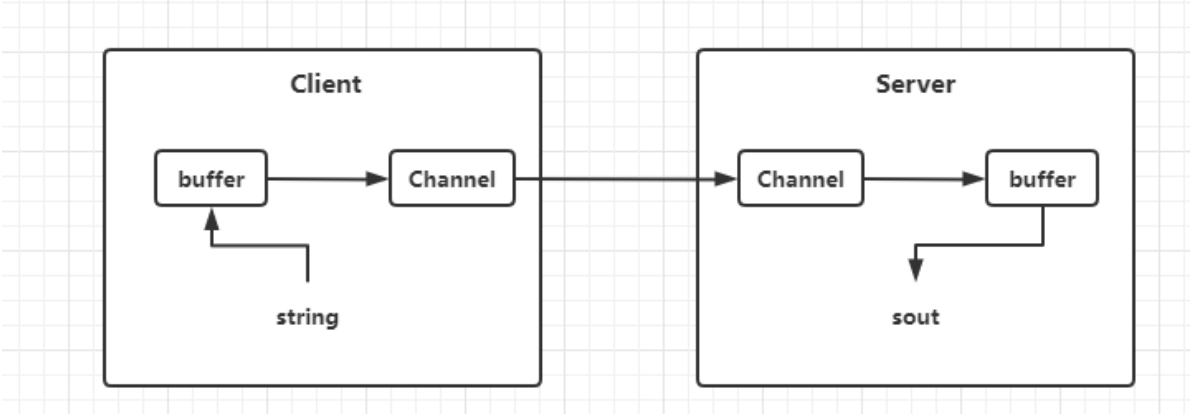
- public static SocketChannel open()，得到一个 SocketChannel 通道
- public final SelectableChannel configureBlocking(boolean block)，设置阻塞或非阻塞模式， 取值 false 表示采用非阻塞模式
- public boolean connect(SocketAddress remote)，连接服务器
- public boolean finishConnect()，如果上面的方法连接失败，接下来就要通过该方法完成连接操作
- public int write(ByteBuffer src)，往通道里写数据
- public int read(ByteBuffer dst)，从通道里读数据
- public final SelectionKey register(Selector sel, int ops, Object att)，注册一个选择器并设置监听事件，最后一个参数可以设置共享数据
- public final void close()，关闭通道



服务器端有一个选择器对象，服务器的ServerSocketChannel对象也要注册给selector，它的accept方法负责接收客户端的连接请求。有一个客户端连接过来，服务端就会建立一个通道。Selector会监控所有注册的通道，检查这些通道中是否有事件发生【连接、断开、读、写等事件】，如果某个通道有事件发生则做相应的处理。

2.2.4 NIO案例：客户端与服务器之间通信

API 学习完毕后，接下来我们使用 NIO 开发一个入门案例，实现服务器端和客户端之间的数据通信（非阻塞）。



上面代码用 NIO 实现了一个服务器端程序，能不断接受客户端连接并读取客户端发过来的数据

```
package com.hero.nio.socket;

import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

//网络客户端程序
public class NIOClient {
    public static void main(String[] args) throws  Exception{
        //1. 得到一个网络通道
        SocketChannel channel=SocketChannel.open();
        //2. 设置非阻塞方式
        channel.configureBlocking(false);
        //3. 提供服务器端的IP地址和端口号
        InetSocketAddress address=new InetSocketAddress("127.0.0.1",9999);
        //4. 连接服务器端，如果用connect()方法连接服务器不成功，则用finishConnect()方法进行连接
        if(!channel.connect(address)){
            //因为连接需要花时间，所以用while一直去尝试连接。在连接服务端时还可以做别的事，体现非阻塞。
            while(!channel.finishConnect()){
                //nio作为非阻塞式的优势，如果服务器没有响应（不启动服务端），客户端不会阻塞，最后会报错，客户端尝试链接服务器连不上。
                System.out.println("Client:连接金莲的同时，还可以干别的一些事情");
            }
        }
        //5. 得到一个缓冲区并存入数据
        String msg="你好，金莲，大郎在家吗？";
        ByteBuffer writeBuf = ByteBuffer.wrap(msg.getBytes());
        //6. 发送数据
        channel.write(writeBuf);
```

```
        //阻止客户端停止，否则服务端也会停止。
        System.in.read();
    }
}

```

```
package com.hero.nio.socket;

import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;

//网络服务器端程序
public class NIOServer {
    public static void main(String[] args) throws Exception {
        //1. 开启一个ServerSocketChannel通道（对象）
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        //2. 开启一个Selector选择器
        Selector selector = Selector.open();
        //3. 绑定端口号9999
        System.out.println("服务端 启动....");
        System.out.println("初始化端口 9999 ");
        serverSocketChannel.bind(new InetSocketAddress(9999));
        //4. 配置非阻塞方式
        serverSocketChannel.configureBlocking(false);
        //5. Selector选择器注册ServerSocketChannel通道，绑定连接操作
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
        //6. 循环执行：监听连接事件及读取数据操作
        while (true) {
            //6.1 监控客户端连接：
            // selecto.select()方法返回的是客户端的通道数，如果为0，则说明没有客户端连接。
            //nio非阻塞式的优势
            if (selector.select(2000) == 0) {
                System.out.println("Server：门庆没有找我，去找王妈妈搞点兼职做~");
                continue;
            }
            //6.2 得到SelectionKey,判断通道里的事件
            Iterator<SelectionKey> keyIterator = selector.selectedKeys().iterator();
            //遍历所有SelectionKey
            while (keyIterator.hasNext()) {
                SelectionKey key = keyIterator.next();
                //客户端先连接上，处理连接事件，然后客户端会向服务端发信息，再处理读取客户端数据事件。
                if (key.isAcceptable()) {//客户端连接请求事件
                    System.out.println("OP_ACCEPT");
                    SocketChannel socketChannel = serverSocketChannel.accept();
                    socketChannel.configureBlocking(false);
```

```
                    //注册通道 ,将通道交给selector选择器进行监控。
                    //参数01-选择器
                    //参数02-服务器要监控读事件，客户端发send数据，服务端读read数据
                    //参数03-客户端传过来的数据要放在缓冲区
                    socketChannel.register(selector, SelectionKey.OP_READ,
ByteBuffer.allocate(1024));
                }
                if (key.isReadable()) {//读取客户端数据事件
                    //数据在通道中，先得到通道
                    SocketChannel channel = (SocketChannel) key.channel();
                    //取到一个缓冲区，nio读写数据都是基于缓冲区。
                    ByteBuffer buffer = (ByteBuffer) key.attachment();
                    //从通道中将客户端发来的数据读到缓冲区
                    channel.read(buffer);
                    System.out.println("客户端发来数据：" + new
String(buffer.array()));
                }
                // 6.3 手动从集合中移除当前key,防止重复处理
                keyIterator.remove();
            }
        }
    }
}


```

上面代码通过 NIO 实现了一个客户端程序，连接上服务器端后发送了一条数据

2.2.5 网络聊天室V1.0

接下来我们用 NIO 实现一个多人聊天案例，具体代码如下所示

```
package com.hero.nio.chat;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.text.SimpleDateFormat;
import java.util.*;

//聊天程序服务器端
public class ChatServer {
    private ServerSocketChannel listenerChannel; //监听通道
    private Selector selector;//选择器对象
    private static final int PORT = 9999; //服务器端口

    public ChatServer() {
        try {
            // 1. 开启Socket监听通道
            listenerChannel = ServerSocketChannel.open();
            // 2. 开启选择器
            selector = Selector.open();
            // 3. 绑定端口
            listenerChannel.bind(new InetSocketAddress(PORT));
```

```
            // 4. 设置为非阻塞模式
            listenerChannel.configureBlocking(false);
            // 5. 将选择器绑定到监听通道并监听accept事件
            listenerChannel.register(selector, SelectionKey.OP_ACCEPT);
            printInfo("真人网络聊天室 启动.......");
            printInfo("真人网络聊天室 初始化端口 9999.......");
            printInfo("真人网络聊天室 初始化网络ip地址 121.199.163.228.......");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void start() throws  Exception{
        try {
            while (true) { //不停监控
                if (selector.select(2000) == 0) {
                    System.out.println("Server:没有客户端连接，我去搞点兼职");
                    continue;
                }
                Iterator<SelectionKey> iterator =
selector.selectedKeys().iterator();
                while (iterator.hasNext()) {
                    SelectionKey key = iterator.next();
                    if (key.isAcceptable()) { //连接请求事件
                        SocketChannel sc=listenerChannel.accept();
                        sc.configureBlocking(false);
                        sc.register(selector,SelectionKey.OP_READ);

 System.out.println(sc.getRemoteAddress().toString().substring(1)+"上线
了...");
                    }
                    if (key.isReadable()) { //读取数据事件
                        readMsg(key);
                    }
                    //一定要把当前key删掉，防止重复处理
                    iterator.remove();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    //读取客户端发来的消息并广播出去
    public void readMsg(SelectionKey key) throws Exception{
        SocketChannel channel=(SocketChannel) key.channel();
        ByteBuffer buffer=ByteBuffer.allocate(1024);
        int count=channel.read(buffer);
        if(count>0){
            String msg=new String(buffer.array());
            //打印消息
            printInfo(msg);
            //全员广播消息
            broadCast(channel,msg);
        }
```

```
    }

    //给所有的客户端发广播
    public void broadCast(SocketChannel except,String msg) throws Exception{
        System.out.println("服务器广播了消息...");
        for(SelectionKey key:selector.keys()){
            Channel targetChannel=key.channel();
            if(targetChannel instanceof SocketChannel &&
targetChannel!=except){
                SocketChannel destChannel=(SocketChannel)targetChannel;
                ByteBuffer buffer=ByteBuffer.wrap(msg.getBytes());
                destChannel.write(buffer);
            }
        }
    }

    private void printInfo(String str) { //往控制台打印消息
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        System.out.println("[" + sdf.format(new Date()) + "] -> " + str);
    }

    public static void main(String[] args) throws Exception {
        new ChatServer().start();;
    }
}
```

上述代码使用 NIO 编写了一个聊天程序的服务器端，可以接受客户端发来的数据，并能把数据广播给所有客户端。

```
package com.hero.nio.chat;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

//聊天程序客户端
public class ChatClient {
    private final String HOST = "121.199.163.228"; //服务器地址
    private int PORT = 9999; //服务器端口
    private SocketChannel socketChannel; //网络通道
    private String userName; //聊天用户名

    public ChatClient() throws IOException {
        //1. 得到一个网络通道
        socketChannel=SocketChannel.open();
        //2. 设置非阻塞方式
        socketChannel.configureBlocking(false);
        //3. 提供服务器端的IP地址和端口号
        InetSocketAddress address=new InetSocketAddress(HOST,PORT);
        //4. 连接服务器端
        if(!socketChannel.connect(address)){
            while(!socketChannel.finishConnect()){  //nio作为非阻塞式的优势
                System.out.println("Client:连接服务器端的同时，咱也别闲着，去搞点兼
职~");
```

```
            }
        }
        //5. 得到客户端IP地址和端口信息，作为聊天用户名使用
        userName = socketChannel.getLocalAddress().toString().substring(1);
        System.out.println("---------------Client(" + userName + ") is
ready---------------");
    }


    //向服务器端发送数据
    public void sendMsg(String msg) throws Exception{
        if(msg.equalsIgnoreCase("bye")){
            socketChannel.close();
            return;
        }
        msg = userName + "说: "+ msg;
        ByteBuffer buffer=ByteBuffer.wrap(msg.getBytes());
        socketChannel.write(buffer);
    }


    //从服务器端接收数据
    public void receiveMsg() throws  Exception{
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        int size=socketChannel.read(buffer);
        if(size>0){
            String msg=new String(buffer.array());
            System.out.println(msg.trim());
        }
    }
}

```

上述代码通过 NIO 编写了一个聊天程序的客户端，可以向服务器端发送数据，并能接收服务器广播的数据。

```
package com.hero.nio.chat;

import java.util.Scanner;

//启动聊天程序客户端
public class TestChat {
    public static void main(String[] args) throws Exception {
        ChatClient chatClient=new ChatClient();

        new Thread(() -> {
            //监听服务器消息
            while(true){
                try {
                    chatClient.receiveMsg();
                    Thread.sleep(2000);
                }catch (Exception e){
                    e.printStackTrace();
                }
            }
```

```
        }).start();

        Scanner scanner=new Scanner(System.in);
        while (scanner.hasNextLine()){
            String msg=scanner.nextLine();
            chatClient.sendMsg(msg);
        }
    }
}
```

上述代码运行了聊天程序的客户端，该代码运行一次就是一个聊天客户端，可以同时运行多个聊天客户端。在一个聊天客户端中发送消息，会广播给所有其他聊天客户端。客户端互相发送消息，需要提前将服务端启动。

支持局域网聊天，也支持网络聊天。使用内网穿透将本机9999端口映射到公网121.199.163.228的9999端口，即可实现网络群聊。

2.3 AIO 编程

2.3.1 概念

JDK 7 引入了 Asynchronous IO，即 AIO，叫做异步不阻塞的 IO，也可以叫做NIO2。在进行 IO 编程中，常用到两种模式：Reactor模式 和 Proactor模式。

- NIO采用 Reactor 模式，**当有事件触发时，服务器端得到通知，进行相应的处理。**
- AIO采用 Proactor模式，引入**异步通道**的概念， 简化了程序编写，一个有效的请求才启动一个线程，它的**特点是先由操作系统完成后，才通知服务端程序启动线程去处理**，一般适用于连接数较多且连接时间较长的应用。

2.3.2 IO对比总结

IO 的方式通常分为几种：同步阻塞的 BIO、同步非阻塞的 NIO、异步非阻塞的 AIO。

- BIO 方式：适用于连接数目比较小且固定的架构
  - 这种方式对服务器资源要求比较高，并发局限于应用中
  - JDK1.4 以前的唯一选择，但程序直观简单易理解
  - 同步阻塞：食堂排队取餐：中午去食堂吃饭，排队等着，啥都干不了，到你了选餐，付款，然后找位子吃饭
- NIO 方式：适用于连接数目多且连接比较短（轻操作）的架构
  - 比如：聊天服务器，并发局限于应用中，编程比较复杂
  - JDK1.4 开始支持
  - 同步非阻塞：下馆子：点完餐，就去商场玩儿了。玩一会儿，就回饭馆问一声：好了没
- AIO 方式：使用于连接数目多且连接比较长（重操作）的架构
  - 比如：相册服务器，充分调用 OS 参与并发操作，编程比较复杂
  - JDK7 开始支持。
  - 异步非阻塞：海底捞外卖火锅，打电话订餐。海底捞会说，我们知道您的位置，一会给您送过来，请您安心工作。

对比	BIO	NIO	AIO
IO方式	同步阻塞	同步非阻塞（多路复用）	异步非阻塞
API使用难度	简单	复杂	复杂
可靠性	差	好	好
吞吐量	低	高	高

3. Netty核心技术

3.1 概述

Netty 是一个被广泛使用的，基于NIO的 Java 网络应用编程框架，Netty框架可以帮助开发者快速、简单的实现客户端和服务端的网络应用程序。“快速”和“简单”并不用产生维护性或性能上的问题。Netty 利用 Java 语言的NIO网络编程的能力，并隐藏其背后的复杂性，从而提供一个易用的 API，基于这些API，我们可以快速编写出一个客户端/服务器网络应用程序。

Netty 成长于用户社区，像大型公司 Facebook 和 Instagram 以及流行的开源项目（Apache Cassandra、Apache Storm、Elasticsearch、Dubbo等等），都利用其强大的对于网络抽象的核心代码实现网络通信。

特点：

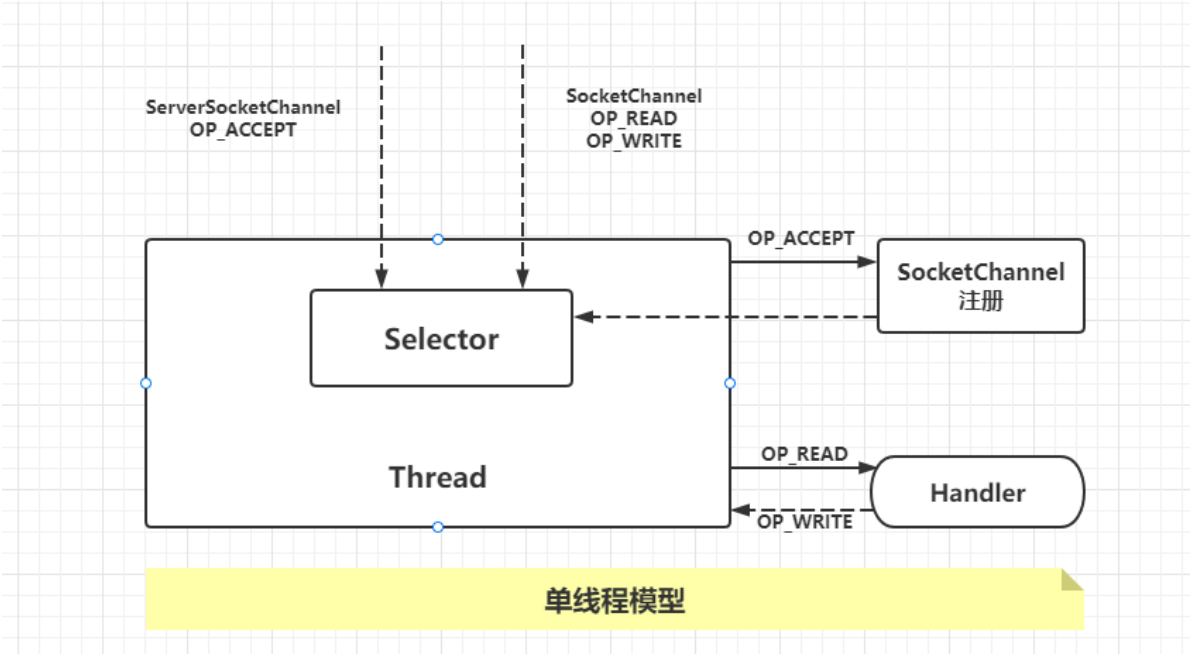
- API简单易用：支持阻塞和非阻塞式的socket
- 基于事件模型：可扩展性和灵活性更强
- 高度定制化的线程模型：支持单线程和多线程
- 高通吐、低延迟、资源占用率低
- 完整支持SSL和TLS
- 学习难度低

应用场景：

- 互联网行业：分布式系统远程过程调用，高性能的RPC框架
- 游戏行业：大型网络游戏高性能通信
- 大数据：Hadoop的高性能通信和序列化组件 Avro 的 RPC 框架

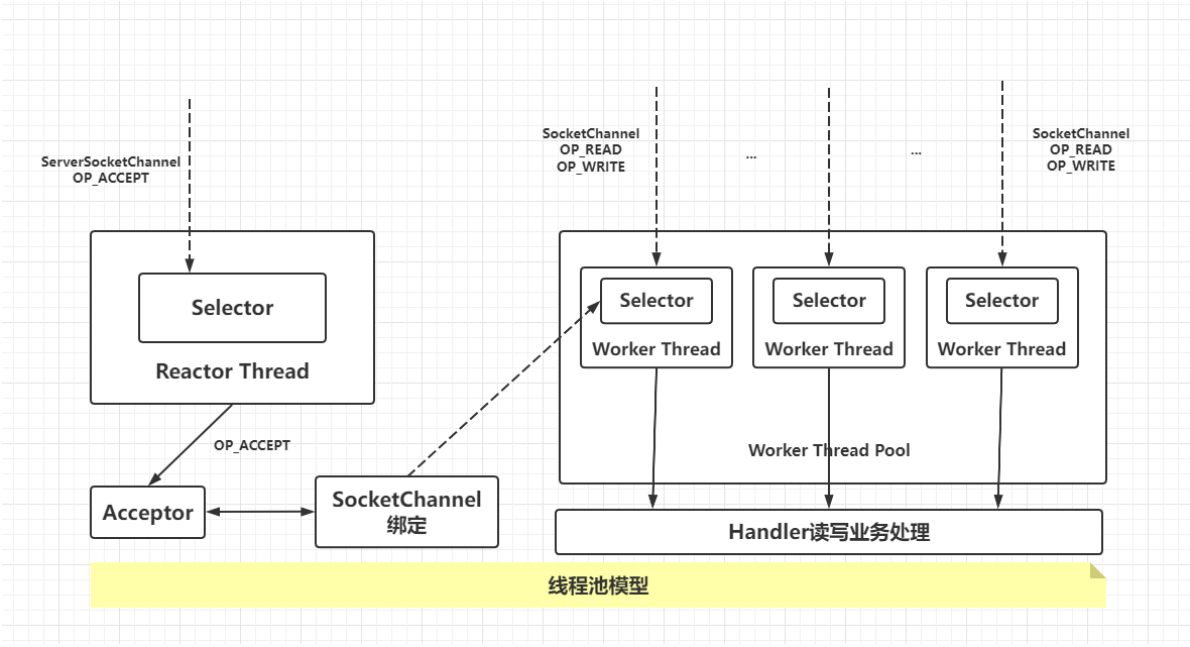
3.2 线程模型

3.2.1 单线程模型



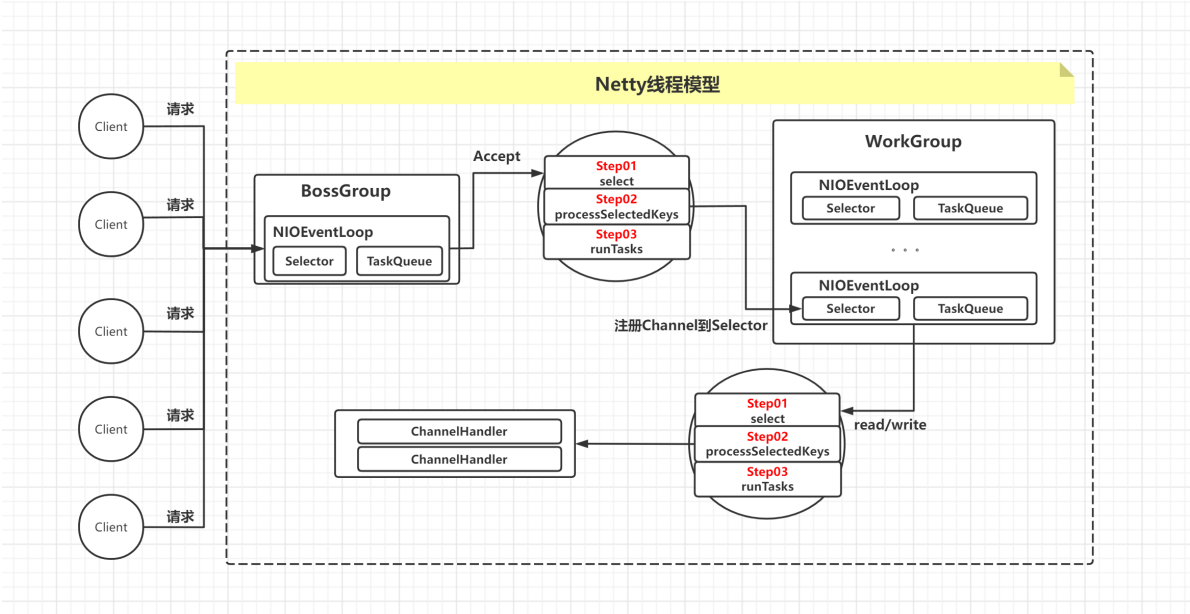
服务端用一个线程通过多路复用搞定所有的 IO 操作（包括连接，读、写等），编码简单，清晰明了，但是如果客户端连接数量较多，将无法支撑，咱们前面的 NIO 案例就属于这种模型。

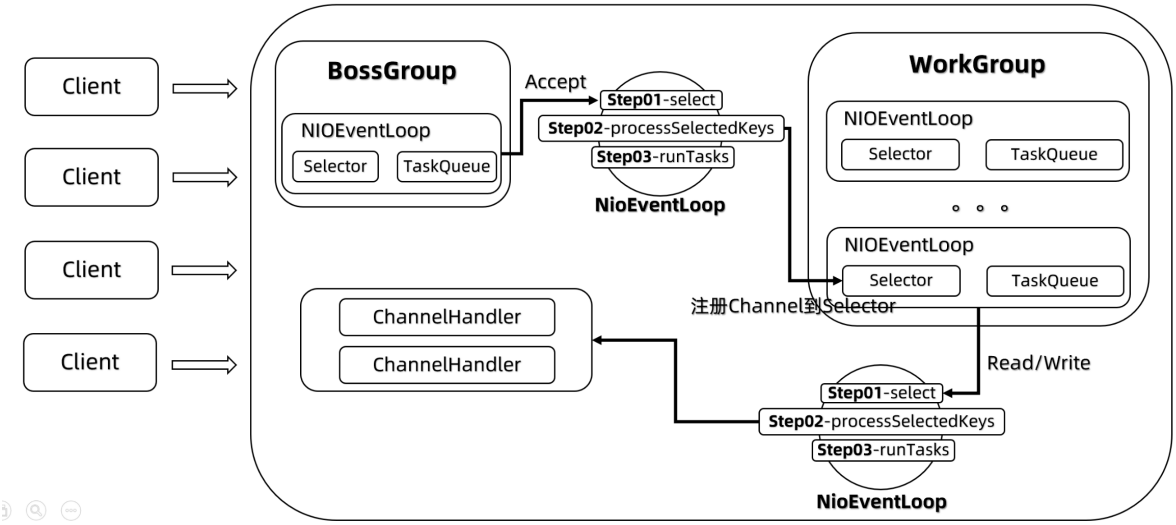
3.2.2 线程池模型



服务端用一个线程专门处理客户端连接请求，用一个线程池负责 IO 操作。在绝大多数场景下，该模型都能满足网络编程需求。

3.2.3 Netty 线程模型





各组件间的关系

- Netty 抽象出两组线程池：**BossGroup、WorkerGroup**
  - BossGroup 专门负责接收客户端连接
  - WorkerGroup 专门负责网络读写操作
  - BossGroup 和 WorkerGroup 类型都是 **NioEventLoopGroup**，相当于一个事件循环组
- **NioEventLoopGroup** 可以有多个线程，即含有多个NioEventLoop
- **NioEventLoop** 表示一个不断循环的执行处理任务的**线程**
  - 每个 NioEventLoop 中包含有一个 Selector，一个 taskQueue
    - **Selector 上可以注册监听多个 NioChannel，也就是监听Socket网络通信**
    - **每个 NioChannel 只会绑定在唯一的 NioEventLoop 上**
    - **每个 NioChannel 都绑定有一个自己的 ChannelPipeline**
  - NioEventLoop 内部采用串行化（Pipeline）设计：责任链模式
    - 消息读取 ==> 解码 ==> 处理（handlers） ==> 编码 ==> 发送，始终由IO线程 NioEventLoop 负责

一个Client连接的执行流程

1. Boss的NioEventLoop 循环执行步骤：
    1. 轮询 accept 事件
    2. 处理 accept 事件：
        - 与client建立连接，生成NioSocketChannel ， 并将其注册到某个worker的NIOEventLoop的 selector，
    3. 处理任务队列的任务 ， 即 runTasks
2. Worker的NIOEventLoop 循环执行步骤：
    1. 轮询read、write 事件
    2. 在对应NioSocketChannel中，处理业务相关操作（ChannelHandler）
    3. 处理任务队列的任务，即 runTasks
3. 每个Worker的NioEventLoop 处理业务时会使用管道Pipeline。Pipeline中包含了 Channel，通过管道可以获取到对应Channel，Channel 中维护了很多的Handler处理器。

3.3 核心API

1. ServerBootstrap 和 Bootstrap

- **ServerBootstrap** 是 Netty 中的**服务端启动助手**，通过它可以完成服务端的各种配置；
- **Bootstrap** 是 Netty 中的**客户端启动助手**，通过它可以完成客户端的各种配置。

常用方法：

- **服务端ServerBootstrap**
    - ServerBootstrap **group**(parentGroup , childGroup)， 该方法用于设置两个EventLoopGroup，连接线程组和工作线程组
    - public B **channel**(Class<? extends C> channelClass)，该方法用来**设置服务端或客户端通道的实现类型**
    - public B **option**(ChannelOption option, T value)，用来给 **ServerChannel** 添加配置
    - public ServerBootstrap **childOption**(ChannelOption childOption, T value)，用来给**接收通道**添加配置
    - public ServerBootstrap **childHandler**(ChannelHandler childHandler)，该方法用来设置业务处理类（自定义handler）
    - public ChannelFuture **bind**(int inetPort) ，该方法用于**设置占用端口号**
- **客户端Bootstrap**
    - public B **group**(EventLoopGroup group) ，该方法用来设置客户端的 EventLoopGroup
    - public B **channel**(Class<? extends C> channelClass)，该方法用来**设置服务端或客户端通道的实现类型**
    - public ChannelFuture **connect**(String inetHost, int inetPort) ，该方法用来**配置连接服务端地址信息**，host:port

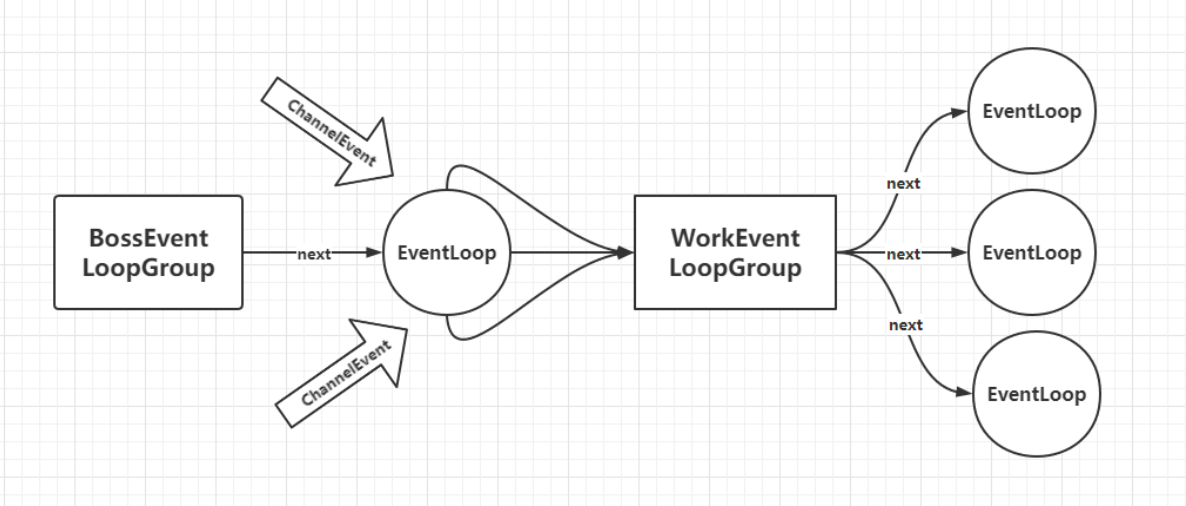
2. EventLoopGroup（Boss\WorkerGroup）

在 Netty 服务端编程中，一般需要提供两个 **EventLoopGroup**： **①BossEventLoopGroup专门负责接收客户端连接、②WorkerEventLoopGroup专门负责网络读写操作。**

- Netty 为了更好的利用多核 CPU 资源，一般会有多个 EventLoop 同时工作，每个 EventLoop 维护着一个 Selector 实例。
- EventLoopGroup 提供 next 接口，可以从组里面按照一定规则获取其中一个 EventLoop 来处理任务。
- **EventLoopGroup 本质是一组 EventLoop，池化管理的思想**

通常一个服务端口即一个ServerSocketChannel 对应一个Selector 和一个EventLoop 线程，BossEventLoop 负责接收客户端的连接并将 SocketChannel 交给 WorkerEventLoopGroup 来进行 IO 处理。

如下图所示：



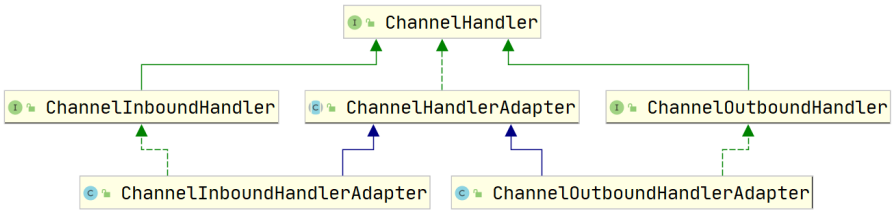
- BossEventLoopGroup 通常是单线程的 EventLoop，EventLoop 维护着一个注册了 ServerSocketChannel 的 Selector 实例
- Boss的EventLoop 不断轮询 Selector 将连接事件分离出来，通常是 OP_ACCEPT 事件， 然后将接收到的 SocketChannel 交给 WorkerEventLoopGroup
- WorkerEventLoopGroup 会由 next 选择其中一个 EventLoop 来将这个 SocketChannel 注册到其维护的 Selector 并对其后续的事件进行处理。

常用方法：

- public NioEventLoopGroup()，构造方法
- public Future<?> shutdownGracefully()，断开连接，关闭线程

3. ChannelHandler 及其实现类

ChannelHandler 接口定义了许多事件处理的方法，我们通过重写这些方法实现业务功能。API 关系如下图所示：

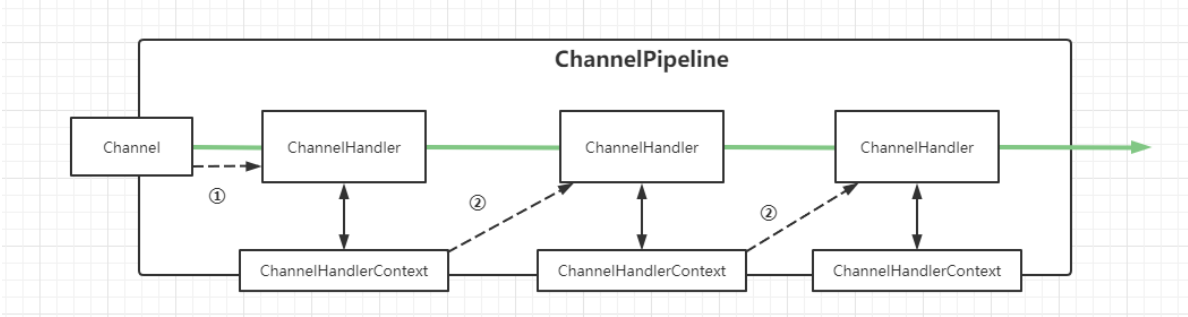


我们经常需要自定义一个 Handler 类去继承 ChannelInboundHandlerAdapter，然后通过重写相应方法实现业务逻辑，我们接下来看看一般都需要重写哪些方法：

- **channelActive**(ChannelHandlerContext ctx)，**通道就绪事件**
- **channelRead**(ChannelHandlerContext ctx, Object msg)，**通道读取数据事件**
- **channelReadComplete**(ChannelHandlerContext ctx)，**数据读取完毕事件**
- **exceptionCaught**(ChannelHandlerContext ctx, Throwable cause)，**通道发生异常事件**

4. ChannelPipeline

ChannelPipeline是一个 **Handler 的集合**，它负责处理和拦截 inbound 或者 outbound 的事件和操作，相当于一个贯穿 Netty 的链（责任链模式）。




```
01-事件传递到ChannelPipeline中的第一个ChannelHandler
02-ChannelHandler使用分配的ChannelHandlerContext将事件传递给ChannelPipeline中的下一个ChannelHander
```

上图绿线的线串起来的就是Pipeline，它包含3个处理不同业务的ChannelHandler，依次通过这三个ChannelHandler。因为这3个ChannelHandler不知道彼此。所以要用ChannelHandlerContext上下文来说明，ChannelHandlerContext包含ChannelHandler、Channel、pipeline的信息。

- ChannelPipeline **addFirst**(ChannelHandler... handlers)，把业务处理类（handler）添加到Pipeline链中的**第一个位置**
- ChannelPipeline **addLast**(ChannelHandler... handlers)，把业务处理类（handler）添加到Pipeline链中的**最后一个位置**

5. ChannelHandlerContext

ChannelHandlerContext是事件处理器上下文对象， Pipeline链中的实际处理节点。 每个处理节点 ChannelHandlerContext 中包含一个具体的事件处理器 ChannelHandler ， 同时 ChannelHandlerContext 中也绑定了对应的 Pipeline 和 Channel 的信息，方便对 ChannelHandler进行调用。

常用方法：

- ChannelFuture **close**()，关闭通道
- ChannelOutboundInvoker **flush**()，刷新
- ChannelFuture **writeAndFlush**(Object msg)，将数据写到ChannelPipeline中当前 ChannelHandler 的下一个 ChannelHandler 开始处理（**出栈**交给下一个handler将继续处理）。

6. ChannelOption

Netty 在创建 Channel 实例后，一般都需要设置 ChannelOption 参数。ChannelOption 是Socket 的标准化参数而非 Netty 的独创。

常配参数：

1. ChannelOption.**SO_BACKLOG**：用来**初始化服务器可连接队列大小**，对应 TCP/IP 协议 listen 函数中的 backlog 参数。
    - 服务端处理客户端连接请求是顺序处理的，所以同一时间只能处理一个客户端连接。
    - 如果请求连接过多，服务端将不能及时处理，多余连接放在队列中等待，backlog 参数指定了等待队列大小。
2. ChannelOption.**SO_KEEPALIVE**，**连接是否一直保持**（是否长连接）。

7. ChannelFuture

ChannelFuture表示 Channel 中异步 IO 操作的未来结果，在 Netty 中异步IO操作都是直接返回，调用者并不能立刻获得结果，但是可以通过 ChannelFuture 来获取 IO 操作的处理状态。Netty异步非阻塞处理事件，如果事件很费时，会通过Future异步处理，不会阻塞。

常用方法：

- Channel **channel**()，返回当前正在进行IO操作的通道
- ChannelFuture **sync**()，等待异步操作执行完毕

8. Unpooled 类

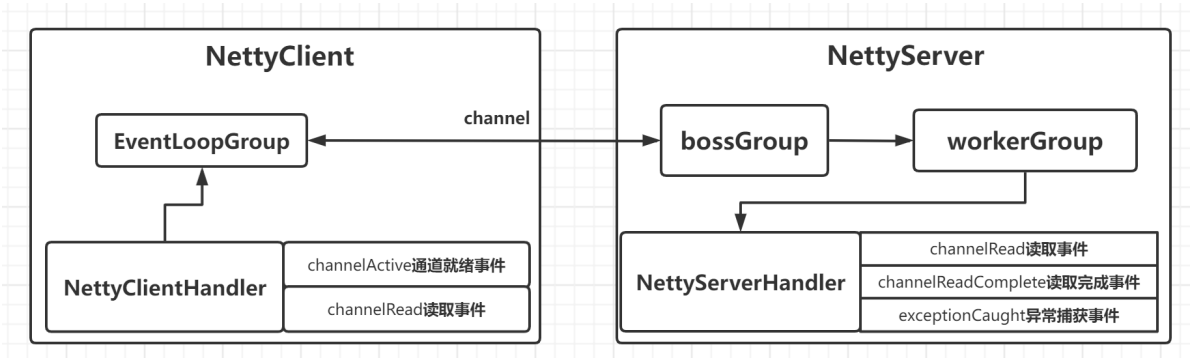
Unpooled 是 Netty 提供的一个专门用来操作缓冲区的工具类

常用方法：

- ByteBuf copiedBuffer(CharSequence string, Charset charset)，通过给定的数据和字符编码返回一个 ByteBuf 对象（类似于 NIO 中的 ByteBuffer 对象）

3.4 Netty案例：客户端与服务器之间通信

目标：API 学习完毕后，接下来我们使用 Netty 开发一个网络应用程序，实现服务端和客户端之间的数据通信。



实现步骤：

1. 导入依赖坐标
2. 编写Netty服务端程序：配置线程组，配置自定义业务处理类，绑定端口号，然后启动Server，等待Client连接
3. 编写服务端-业务处理类Handler：继承 ChannelInboundHandlerAdapter，并分别重写了三个方法
    - 读取事件
    - 读取完成事件
    - 异常捕获事件
4. 编写客户端程序：配置了线程组，配置了自定义的业务处理类，然后启动Client，连接Server。
5. 编写客户端-业务处理类：继承 ChannelInboundHandlerAdapter ，并分别重写了2个方法
    - 通道就绪事件
    - 读取事件

1. 导入依赖

```
<dependencies>
    <dependency>
        <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.8.Final</version>
    </dependency>
</dependencies>
```

上述代码在 pom 文件中引入了 netty 的坐标

2. 服务端程序Server

```
package com.hero.netty.demo01;

public class NettyServer {
    public static void main(String[] args) throws Exception{
        //1. 创建一个线程组：接收客户端连接
        EventLoopGroup bossGroup =new NioEventLoopGroup();
        //2. 创建一个线程组：处理网络操作
        EventLoopGroup workerGroup =new NioEventLoopGroup();
        //3. 创建服务端启动助手来配置参数
        ServerBootstrap b=new ServerBootstrap();
        b.group(bossGroup,workerGroup) //4.设置两个线程组
```

```
            .channel(NioServerSocketChannel.class) //5.使用
NioServerSocketChannel作为服务端通道的实现
            .option(ChannelOption.SO_BACKLOG,128) //6.设置线程队列中等待连接的个数
            .childOption(ChannelOption.SO_KEEPALIVE,true) //7.保持活动连接状态
            .childHandler(new ChannelInitializer<SocketChannel>() {  //8. 创建一
个通道初始化对象
                public void initChannel(SocketChannel sc){   //9. 往Pipeline链中
添加自定义的handler类
                    sc.pipeline().addLast(new NettyServerHandler());
                }
            });
            System.out.println("......服务端 启动中 init port:9999 ......");
            ChannelFuture cf=b.bind(9999).sync();  //10. 绑定端口 bind方法是异步的
sync方法是同步阻塞的
            System.out.println("......服务端 启动成功 ......");

            //11. 关闭通道，关闭线程组
            cf.channel().closeFuture().sync();
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
```

上述代码编写了一个服务端程序，配置了线程组，配置了自定义业务处理类，绑定端口号，然后启动Server，等待Client连接。

3. 服务端-业务处理类ServerHandler

```
package com.hero.netty.demo01;

//服务端的业务处理类
public class NettyServerHandler extends ChannelInboundHandlerAdapter {

    //读取数据事件，msg就客戶端发过来的数据。
    public void channelRead(ChannelHandlerContext ctx,Object msg){
        //System.out.println("Server:"+ctx);
        //用缓冲区接受数据
        ByteBuf buf=(ByteBuf) msg;
        //转成字符串
        System.out.println("client msg: "+buf.toString(CharsetUtil.UTF_8));
    }

    //数据读取完毕事件，读取完客户端数据后回复客户端
    public void channelReadComplete(ChannelHandlerContext ctx){
        //Unpooled.copiedBuffer获取到缓冲区
        //第一个参数是向客户端传的字符串
        ctx.writeAndFlush(Unpooled.copiedBuffer("宝塔镇河
妖",CharsetUtil.UTF_8));
    }

    //异常发生事件
    public void exceptionCaught(ChannelHandlerContext ctx,Throwable t){
        //异常时关闭ctx，ctx是相关信息的汇总，关闭它其它的也就关闭了。
```

```
        ctx.close();
    }
}
```

上述代码定义了一个服务端业务处理类，继承 ChannelInboundHandlerAdapter，并分别重写了三个方法。

4. 客户端程序Client

```
package com.hero.netty.demo01;

public class NettyClient {
    public static void main(String[] args) throws  Exception{

        //1. 创建一个线程组
        EventLoopGroup group=new NioEventLoopGroup();
        //2. 创建客户端的启动助手，完成相关配置
        Bootstrap b=new Bootstrap();
        b.group(group)  //3. 设置线程组
         .channel(NioSocketChannel.class)  //4. 设置客户端通道的实现类
         .handler(new ChannelInitializer<SocketChannel>() {  //5. 创建一个通道初始化对象
             @Override
             protected void initChannel(SocketChannel socketChannel) throws Exception {
                 socketChannel.pipeline().addLast(new NettyClientHandler()); //6.往Pipeline链中添加自定义的handler
             }
         });
        System.out.println("......客户端 准备就绪 msg发射......");

        //7.启动客户端去连接服务端   connect方法是异步的    sync方法是同步阻塞的
        ChannelFuture cf=b.connect("127.0.0.1",9999).sync();

        //8.关闭连接(异步非阻塞)
        cf.channel().closeFuture().sync();

    }
}

```

上述代码编写了一个Client程序，配置了线程组，配置了自定义的业务处理类，然后启动Client，连接Server。

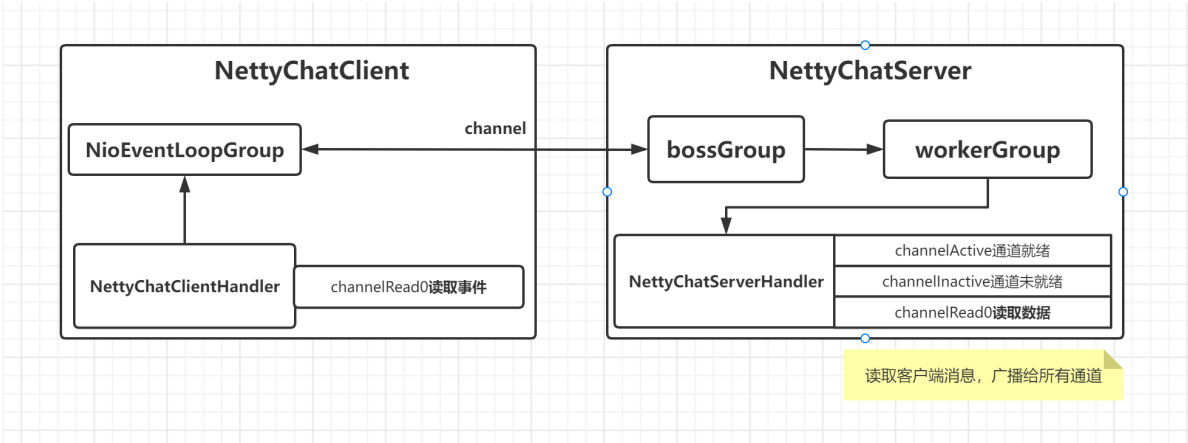
5. 客户端-业务处理类ClientHandler

```
package com.hero.netty.demo01;

//客户端业务处理类
public class NettyClientHandler extends ChannelInboundHandlerAdapter {

    //通道就绪事件
    public void channelActive(ChannelHandlerContext ctx){
        //System.out.println("Client:"+ctx);
        ctx.writeAndFlush(Unpooled.copiedBuffer("天王盖地
虎",CharsetUtil.UTF_8));
    }

    //读取数据事件
    public void channelRead(ChannelHandlerContext ctx,Object msg){
        ByteBuf buf=(ByteBuf) msg;
        System.out.println("server msg: "+buf.toString(CharsetUtil.UTF_8));
    }

}

```

上述代码自定义了一个Client业务处理类，继承 ChannelInboundHandlerAdapter ， 并分别重写了2个方法

3.5 网络聊天室V2.0

> 先看效果：demo02chat



刚才通过 Netty 实现了一个基础案例，基本了解了 Netty 的 API 和运行流程。接下来，我们在上个案例的基础上，再实现一个Netty版本的多人聊天案例。

实现步骤：

1. **编写聊天程序服务端**：配置线程组，**配置编解码器**，配置自定义业务处理类，绑定端口号，然后启动Server，等待Client连接
2. 编写服务端-业务处理类Handler：
    - 当通道就绪时，输出上线
    - 当通道未就绪时，输出离线
    - 当通道发来数据时，读取数据，进行广播
3. 编写聊天程序客户端：配置了线程组，**配置编解码器**，配置了自定义的业务处理类，然后启动Client，连接Server。
    - 连接服务端成功后，获取客户端与服务端建立的Channel
    - 获取系统键盘输入，将用户输入信息通过Channel发送给服务端
4. 编写客户端-业务处理类：
    - 读取事件：监听服务端广播消息

1. 聊天服务端程序ChatServer

```
package com.hero.netty.demo02chat;

//聊天程序服务端
public class NettyChatServer {

    private int port; //服务端端口号

    public NettyChatServer(int port) {
        this.port = port;
    }

    public void run() throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .option(ChannelOption.SO_BACKLOG, 128)
                    .childOption(ChannelOption.SO_KEEPALIVE, true)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        public void initChannel(SocketChannel ch) {
                            ChannelPipeline pipeline = ch.pipeline();
                            //往pipeline链中添加一个解码器
                            pipeline.addLast("decoder", new
StringDecoder());
                            //往pipeline链中添加一个编码器
                            pipeline.addLast("encoder", new
StringEncoder());
                            //往pipeline链中添加自定义的handler(业务处理类)
                            pipeline.addLast(new NettyChatServerHandler());
                        }
                    });
            System.out.println("网络真人聊天室 Server 启动.....");
            ChannelFuture f = b.bind(port).sync();
```

```
            f.channel().closeFuture().sync();
        } finally {
            workerGroup.shutdownGracefully();
            bossGroup.shutdownGracefully();
            System.out.println("网络真人聊天室 Server 关闭......");
        }
    }

    public static void main(String[] args) throws Exception {
        new NettyChatServer(9999).run();
    }
}
```

上述代码通过 Netty 编写了一个服务端程序。

> 注意：我往 Pipeline 链中添加了处理字符串的编码器和解码器，它们加入到 Pipeline 链中后会自动工作，使得服务端读写字符串数据时更加方便，不用人工处理 编解码操作。

2. 服务端业务处理类ChatServerHandler

```
package com.hero.netty.demo02chat;

//自定义一个服务端业务处理类
public class NettyChatServerHandler extends
SimpleChannelInboundHandler<String> {

    public static List<Channel> channels = new ArrayList<>();

    @Override  //通道就绪
    public void channelActive(ChannelHandlerContext ctx)  {
        Channel inChannel=ctx.channel();
        channels.add(inChannel);
        System.out.println("
[Server]:"+inChannel.remoteAddress().toString().substring(1)+"上线");
    }
    @Override  //通道未就绪
    public void channelInactive(ChannelHandlerContext ctx)  {
        Channel inChannel=ctx.channel();
        channels.remove(inChannel);
        System.out.println("
[Server]:"+inChannel.remoteAddress().toString().substring(1)+"离线");
    }
    @Override  //读取数据
    protected void channelRead0(ChannelHandlerContext ctx, String s)  {
        Channel inChannel=ctx.channel();
        System.out.println("s = " + s);
        for(Channel channel:channels){
            if(channel!=inChannel){
                channel.writeAndFlush("
["+inChannel.remoteAddress().toString().substring(1)+"]"+"说："+s+"\n");
            }
        }
    }
```

```
30
31  }
32
```

上述代码通过继承 SimpleChannelInboundHandler 类自定义了一个服务端业务处理类，并在该类中重写了四个方法。

- 当通道就绪时，输出上线
- 当通道未就绪时，输出离线
- 当通道发来数据时，读取数据，进行广播

3. 聊天程序客户端ChatClient

```
1  package com.hero.netty.demo02chat;
2
3  //聊天程序客户端
4  public class NettyChatClient {
5      private final String host; //服务端IP地址
6      private final int port;  //服务端端口号
7
8      public NettyChatClient(String host, int port) {
9          this.host = host;
10         this.port = port;
11     }
12
13     public void run(){
14         EventLoopGroup group = new NioEventLoopGroup();
15         try {
16             Bootstrap bootstrap = new Bootstrap()
17                     .group(group)
18                     .channel(NioSocketChannel.class)
19                     .handler(new ChannelInitializer<SocketChannel>() {
20                         @Override
21                         public void initChannel(SocketChannel ch){
22                             ChannelPipeline pipeline=ch.pipeline();
23                             //往pipeline链中添加一个解码器
24                             pipeline.addLast("decoder",new StringDecoder());
25                             //往pipeline链中添加一个编码器
26                             pipeline.addLast("encoder",new StringEncoder());
27                             //往pipeline链中添加自定义的handler(业务处理类)
28                             pipeline.addLast(new NettyChatClientHandler());
29                         }
30                     });
31
32             ChannelFuture cf=bootstrap.connect(host,port).sync();
33             Channel channel=cf.channel();
34             System.out.println("------
"+channel.localAddress().toString().substring(1)+"------");
35             Scanner scanner=new Scanner(System.in);
36             while (scanner.hasNextLine()){
37                 String msg=scanner.nextLine();
38                 channel.writeAndFlush(msg+"\r\n");
39             }
```

```
            cf.channel().closeFuture().sync();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            group.shutdownGracefully();
        }
    }

    public static void main(String[] args) throws Exception {
        new NettyChatClient("121.199.163.228",9999).run();
    }
}
```

上述代码通过 Netty 编写了一个客户端程序。客户端同样需要配置编解码器

4. 客户端业务处理类ChatClientHandler

```
package com.hero.netty.demo02chat;

//自定义一个客户端业务处理类
public class NettyChatClientHandler extends
SimpleChannelInboundHandler<String> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String s) throws
Exception {
        System.out.println(s.trim());
    }
}

```

上述代码通过继承 SimpleChannelInboundHandler 自定义了一个客户端业务处理类，重写了一个方法用来读取服务端发过来的数据。

3.6 编码和解码

- 为什么要编解码呢？因为计算机数据传输的是二进制的字节数据
- 解码：字节数据 --> 字符串（字符数据）
- 编码 ： 字符串（字符数据）--> 字节数据

1. 概述

我们在编写网络应用程序的时候需要注意 codec (编解码器)，因为数据在网络中传输的都是二进制字节码数据，而我们拿到的目标数据往往不是字节码数据。因此在发送数据时就 需要编码，接收数据时就需要解码。

codec 的组成部分有两个：decoder(解码器)和 encoder(编码器)。

- **encoder** 负责把业务数据转换成字节码数据

- **decoder** 负责把字节码数据转换成业务数据

其实 **Java 的序列化技术**就可以作为 codec 去使用，但是它的硬伤太多：

1. **无法跨语言**，这应该是 Java 序列化最致命的问题了
2. **序列化后的体积太大**，是二进制编码的 5 倍多
3. **序列化性能太低**

Netty 自身提供了一些 编解码器，如下：

- **StringEncoder**对字符串数据进行编码
- **ObjectEncoder**对 Java 对象进行编码

Netty 本身自带的 ObjectDecoder 和 ObjectEncoder 可以用来实现 POJO 对象或各种业务对象的编码和解码，但其内部使用的仍是 **Java 序列化技术**，所以在某些场景下不适用。对于 POJO 对象或各种业务对象要实现编码和解码，我们需要更高效更强的技术。

2. Google 的 Protobuf

Protobuf 是 Google 发布的开源项目，全称 Google Protocol Buffers，特点如下：

- **支持跨平台、多语言**（支持目前大多数语言，例如 C++、C#、Java、python 等）
- **高性能，高可靠性**
- 使用 protobuf 编译器能自动生成代码，Protobuf 是将类的定义使用.proto 文件进行描述，然后通过 protoc.exe 编译器根据.proto 自动生成.java 文件

在使用 Netty 开发时，经常会结合 Protobuf 作为 codec (编解码器)去使用，具体用法如下所示。

使用步骤：

1. 第一步：将传递数据的实体类生成【基于构建者模式设计】
2. 第二步：配置编解码器
3. 第三步：传递数据使用生成后的实体类

3. 导入 protobuf 依赖

在 pom 导入 protobuf 的坐标

```
<dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>3.6.1</version>
</dependency>
```

4. proto 文件

假设我们要处理的数据是图书信息，那就需要为此编写 proto 文件

```
syntax = "proto3";  设置版本号
   option java_outer_classname = "BookMessage";设置生成的Java类名
   message Book{  内部类的类名，POJO
       int32 id = 1;  设置类的属性
       string name = 2;  等号后的数字是序号，不是属性的值
   }
```

```
syntax = "proto3";
   option java_outer_classname = "BookMessage";
   message Book{
       int32 id = 1;
       string name = 2;
   }
```

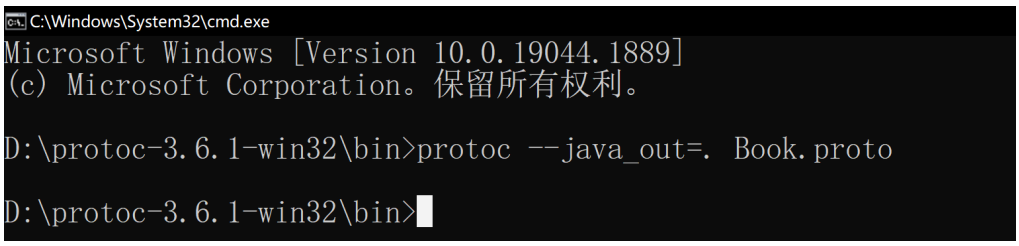
5. 生成 Java 类

通过 protoc.exe 根据描述文件生成 Java 类

```
cd C:\protoc-3.6.1-win32\bin
```

执行以下命令：

```
protoc --java_out=. Book.proto
```



会生成BookMessage.java

把生成的 BookMessage.java 拷贝到项目中

6. 客户端

```
package com.hero.netty.demo03EncoderDecoder;


public class NettyEncoderDecoderClient {

    public static void main(String[] args) throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();
        Bootstrap b = new Bootstrap();
```

```
        b.group(group)
                .channel(NioSocketChannel.class)
                .handler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel sc) {
                        sc.pipeline().addLast("encoder",new
ProtobufEncoder());
                        sc.pipeline().addLast(new NettyClientHandler());
                    }
                });

        // 启动客户端
        ChannelFuture cf = b.connect("127.0.0.1", 9999).sync(); // (5)

        // 等待连接关闭
        cf.channel().closeFuture().sync();
    }
}

```

上述代码在编写客户端程序时，要向 Pipeline 链中添加 ProtobufEncoder 编码器对象。

7. 客户端业务类

```
package com.hero.netty.demo03EncoderDecoder;

public class NettyEncoderDecoderClientHandler extends
ChannelInboundHandlerAdapter {

    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        BookMessage.Book book=
BookMessage.Book.newBuilder().setId(1).setName("天王盖地虎").build();
        ctx.writeAndFlush(book);
    }
}
```

上述代码在往服务端发送图书（POJO）时就可以使用生成的 BookMessage 类搞定，非常方便

8. 服务端

```
package com.hero.netty.codec;

public class NettyEncoderDecoderServer {

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

        EventLoopGroup pGroup = new NioEventLoopGroup(); //线程组：用来处理网络
事件处理（接受客户端连接）
        EventLoopGroup cGroup = new NioEventLoopGroup(); //线程组：用来进行网络
通讯读写

        ServerBootstrap b = new ServerBootstrap();
        b.group(pGroup, cGroup)
```

```
                .channel(NioServerSocketChannel.class) //注册服务端channel
                .option(ChannelOption.SO_BACKLOG, 128)
                . childOption(ChannelOption.SO_KEEPALIVE, true)
                .childHandler(new ChannelInitializer<SocketChannel>() {
                    public void initChannel(SocketChannel sc) throws
Exception {
                        sc.pipeline().addLast("decoder",new
ProtobufDecoder(BookMessage.Book.getDefaultInstance()));
                        sc.pipeline().addLast(new NettyServerHandler());
                    }
                });
        ChannelFuture cf = b.bind(9999).sync();
        System.out.println("......Server is Starting......");

        //释放
        cf.channel().closeFuture().sync();
        pGroup.shutdownGracefully();
        cGroup.shutdownGracefully();
    }
}

```

上述代码在编写服务端程序时，要向 Pipeline 链中添加 ProtobufDecoder 解码器对象。

9. 服务端业务类

```
package com.hero.netty.demo03EncoderDecoder;

public class NettyEncoderDecoderServerHandler extends
ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        BookMessage.Book book=(BookMessage.Book)msg;
        System.out.println("客户端 msg: "+book.getName());
    }
}
```

上述代码在服务端接收数据时，直接就可以把数据转换成 POJO 使用，很方便。