

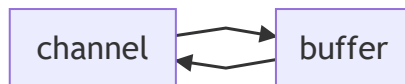
NIO 基础

non-blocking io 非阻塞 IO

三大组件

Channel & Buffer

channel 有一点类似于 stream，它就是读写数据的**双向通道**，可以从 channel 将数据读入 buffer，也可以将 buffer 的数据写入 channel，而之前的 stream 要么是输入，要么是输出，channel 比 stream 更为底层



常见的 Channel 有

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel

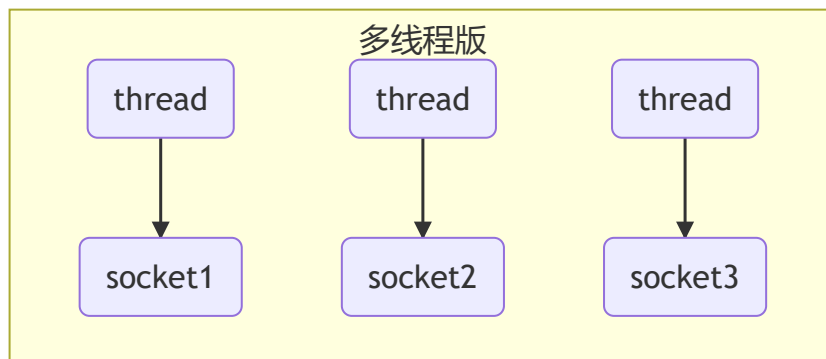
buffer 则用来缓冲读写数据，常见的 buffer 有

- ByteBuffer
 - MappedByteBuffer
 - DirectByteBuffer
 - HeapByteBuffer
- ShortBuffer
- IntBuffer
- LongBuffer
- FloatBuffer
- DoubleBuffer
- CharBuffer

1.2 Selector

selector 单从字面意思不好理解，需要结合服务器的设计演化来理解它的用途

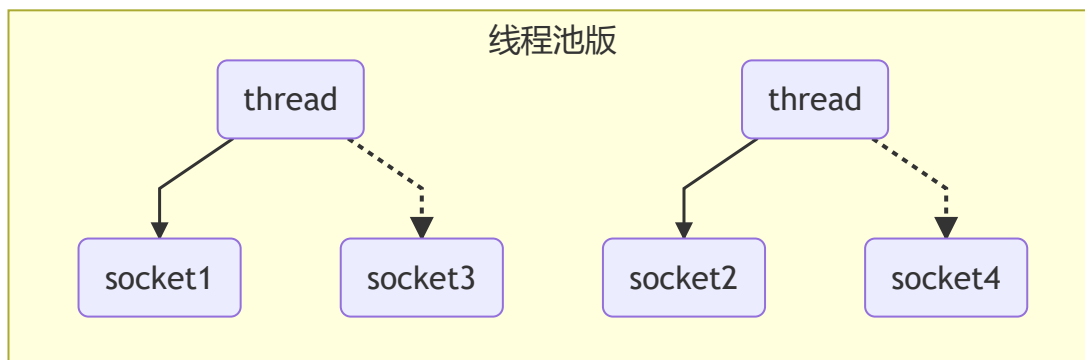
多线程版设计



⚠ 多线程版缺点

- 内存占用高（相当于一个餐厅有十名顾客，你用了十名服务员）
- 线程上下文切换成本高（一共有十个工位，你有二十名服务员，每一个进来收拾上一个人的东西，会耽误太多时间）
- 只适合连接数少的场景

线程池版设计

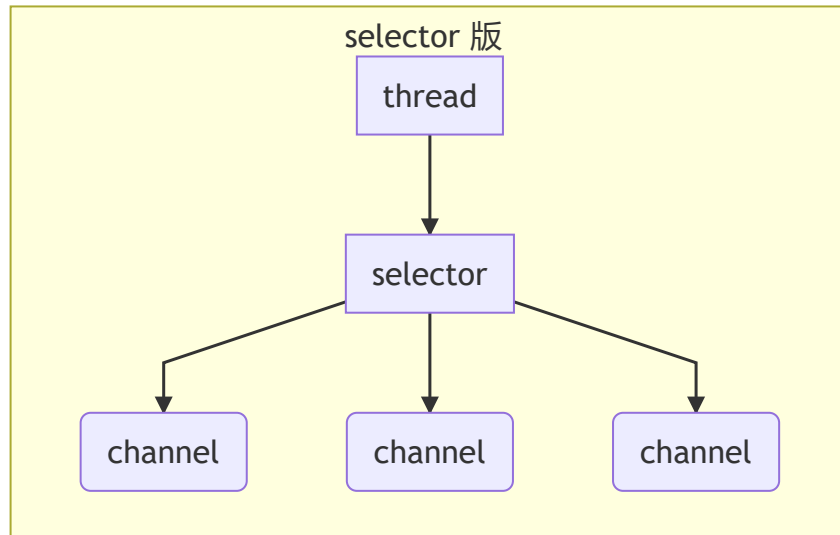


⚠ 线程池版缺点

- 阻塞模式下，线程仅能处理一个 socket 连接（当socket1正在处理的时候，这时候就发生阻塞了，socket3就无法进行。等到socket1执行完了释放连接，socket3才能执行）
- 仅适合短连接场景（做完一项业务赶紧释放连接，把线程阻塞给释放掉）

selector 版设计

selector 的作用就是配合一个线程来管理多个 channel，获取这些 channel 上发生的事件，这些 channel 工作在非阻塞模式下，不会让线程吊死在一个 channel 上。适合连接数特别多，但流量低的场景（low traffic）



调用 selector 的 select() 会阻塞直到 channel 发生了读写就绪事件，这些事件发生，select 方法就会返回这些事件交给 thread 来处理

2. ByteBuffer

有一普通文本文件 data.txt，内容为

```
woshixutianhao
```

使用 FileChannel 来读取文件内容

```
import lombok.extern.slf4j.Slf4j;

import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

@Slf4j
public class ByteBufferDemo1 {
    public static void main(String[] args) {
        try (FileInputStream file = new FileInputStream("data.txt")) {
            FileChannel channel = file.getChannel();
            ByteBuffer buffer = ByteBuffer.allocate(10);
            while (true) {
                // 向 buffer 写入
                int len = channel.read(buffer);
                log.debug("读到字节数: {}", len);
                if (len == -1) {
                    break;
                }
            }
        }
    }
}
```

```

    }
    // 切换 buffer 读模式
    buffer.flip();
    while(buffer.hasRemaining()) {
        log.debug("{} ", (char)buffer.get());
    }
    // 切换 buffer 写模式
    buffer.clear();
}
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

输出

```

D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\bin\java.exe "-javaagent:D:\IntelliJ IDEA
2021.3\lib\idea_rt.jar=53564:D:\IntelliJ IDEA 2021.3\bin" -Dfile.encoding=UTF-8 -
classpath
D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\charsets.jar;D:\JDK1.8.1\JDK1.8\Jav
a\jdk1.8.0_311\jre\lib\deploy.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\ex
t\access-bridge-
64.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\ext\clldrdata.jar;D:\JDK1.8.1
\JDK1.8\Java\jdk1.8.0_311\jre\lib\ext\dnsns.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0
_311\jre\lib\ext\jaccess.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\ext\jfx
rt.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\ext\localedata.jar;D:\JDK1.
8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\ext\nashorn.jar;D:\JDK1.8.1\JDK1.8\Java\jdk
1.8.0_311\jre\lib\ext\sunec.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\ext
\sunjc_provider.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\ext\sunmscapi.
jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\ext\sunpkcs11.jar;D:\JDK1.8.1\J
DK1.8\Java\jdk1.8.0_311\jre\lib\ext\zipfs.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_3
11\jre\lib\javaws.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\jce.jar;D:\JD
K1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\jfr.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0
_311\jre\lib\jfxswt.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\jsse.jar;D:
\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\management-
agent.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\jre\lib\plugin.jar;D:\JDK1.8.1\JD
K1.8\Java\jdk1.8.0_311\jre\lib\resources.jar;D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_31
1\jre\lib\rt.jar;D:\IDEACODE\Netty\target\test-
classes;D:\IDEACODE\Netty\target\classes;C:\Users\xu\.m2\repository\io\netty\net
ty-all\4.1.39.Final\netty-all-
4.1.39.Final.jar;C:\Users\xu\.m2\repository\org\projectlombok\lombok\1.16.18\lom
bok-1.16.18.jar;C:\Users\xu\.m2\repository\com\google\code\gson\gson\2.8.5\gson-
2.8.5.jar;C:\Users\xu\.m2\repository\com\google\guava\guava\19.0\guava-
19.0.jar;C:\Users\xu\.m2\repository\ch\qos\logback\logback-classic\1.2.3\logback-
classic-1.2.3.jar;C:\Users\xu\.m2\repository\ch\qos\logback\logback-
core\1.2.3\logback-core-1.2.3.jar;C:\Users\xu\.m2\repository\org\slf4j\slf4j-
api\1.7.25\slf4j-api-
1.7.25.jar;C:\Users\xu\.m2\repository\com\google\protobuf\protobuf-
java\3.11.3\protobuf-java-3.11.3.jar ByteBufferDemo1
22:53:08.568 [main] DEBUG ByteBufferDemo1 - 读到字节数: 10
22:53:08.572 [main] DEBUG ByteBufferDemo1 - w

```

```
22:53:08.572 [main] DEBUG ByteBufferDemo1 - o
22:53:08.572 [main] DEBUG ByteBufferDemo1 - s
22:53:08.572 [main] DEBUG ByteBufferDemo1 - h
22:53:08.572 [main] DEBUG ByteBufferDemo1 - i
22:53:08.572 [main] DEBUG ByteBufferDemo1 - x
22:53:08.573 [main] DEBUG ByteBufferDemo1 - u
22:53:08.573 [main] DEBUG ByteBufferDemo1 - t
22:53:08.573 [main] DEBUG ByteBufferDemo1 - i
22:53:08.573 [main] DEBUG ByteBufferDemo1 - a
22:53:08.573 [main] DEBUG ByteBufferDemo1 - 读到字节数: 4
22:53:08.573 [main] DEBUG ByteBufferDemo1 - n
22:53:08.573 [main] DEBUG ByteBufferDemo1 - h
22:53:08.573 [main] DEBUG ByteBufferDemo1 - a
22:53:08.573 [main] DEBUG ByteBufferDemo1 - o
22:53:08.573 [main] DEBUG ByteBufferDemo1 - 读到字节数: -1
```

Process finished with exit code 0

2.1 ByteBuffer 正确使用姿势

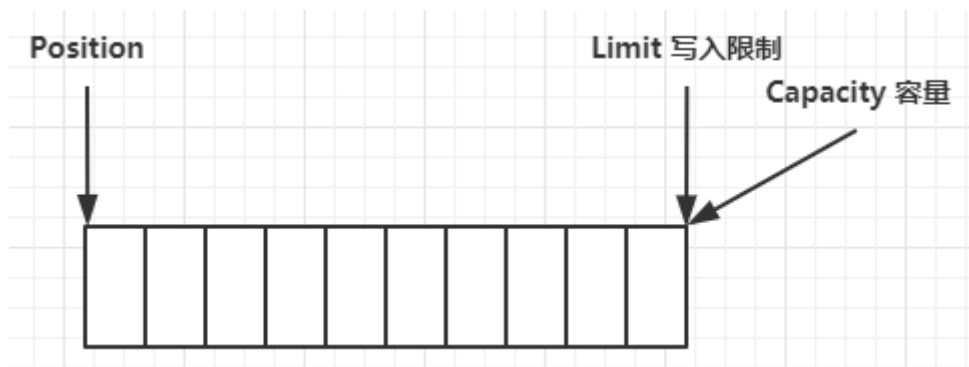
1. 向 buffer 写入数据，例如调用 `channel.read(buffer)`
2. 调用 `flip()` 切换至**读模式**
3. 从 buffer 读取数据，例如调用 `buffer.get()`
4. 调用 `clear()` 或 `compact()` 切换至**写模式**
5. 重复 1~4 步骤

2.2 ByteBuffer 结构

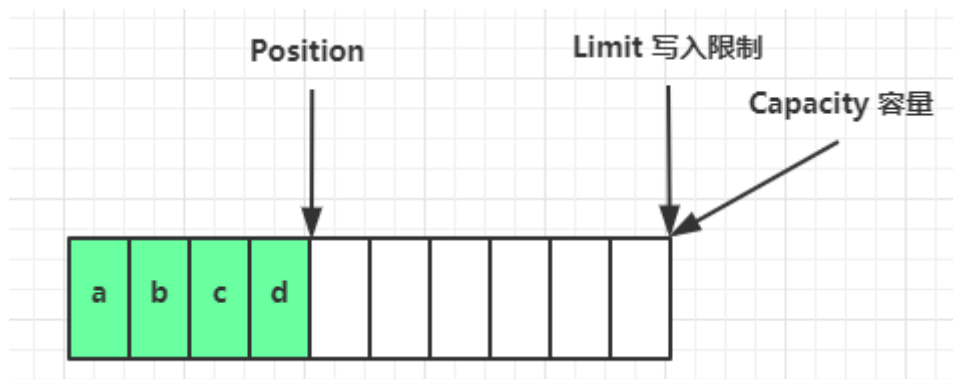
ByteBuffer 有以下重要属性

- capacity
- position
- limit

一开始



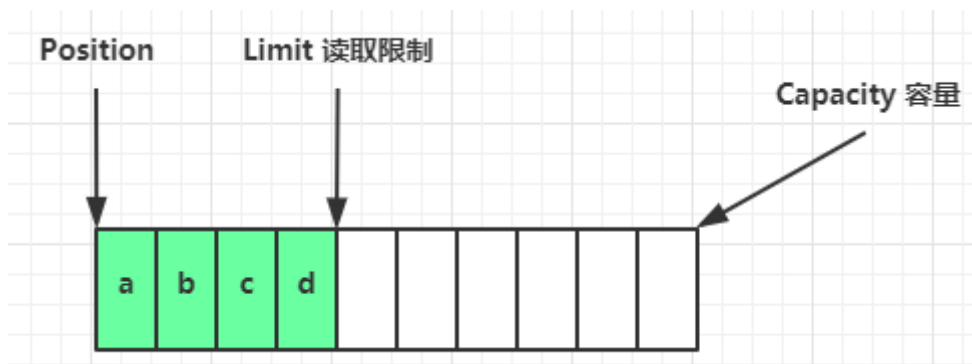
写模式下，position 是写入位置，limit 等于容量，下图表示写入了 4 个字节后的状态



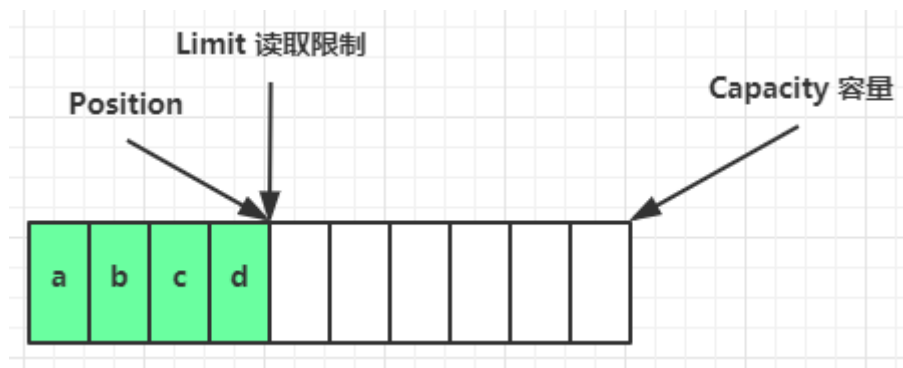
flip 动作发生后，position 切换为读取位置，limit 切换为读取限制

源码为

```
public final Buffer flip() {  
    limit = position;  
    position = 0;  
    mark = -1;  
    return this;  
}
```



读取 4 个字节后，状态



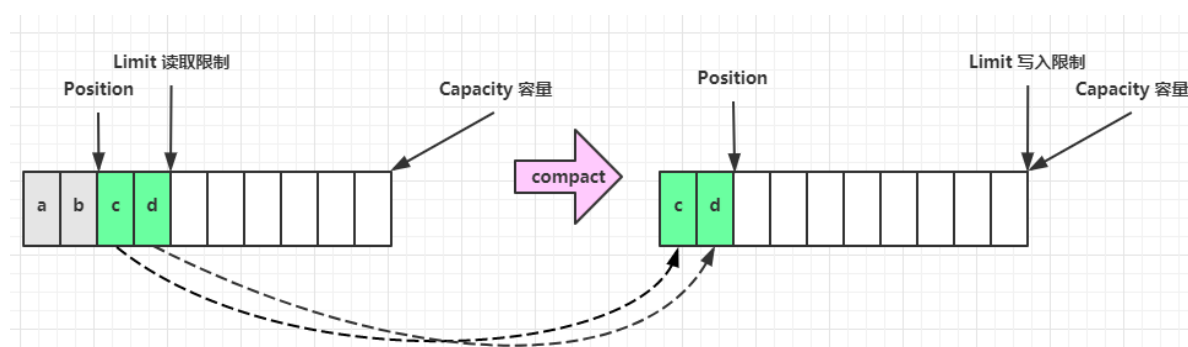
`clear` 动作发生后, 状态

`clear()`方法的源码

```
public final Buffer clear() {
    position = 0;
    limit = capacity;
    mark = -1;
    return this;
}
```



`compact` 方法, 是把未读完的部分向前压缩, 然后切换至写模式



💡 调试工具类

```
import io.netty.util.internal.StringUtil;

import java.nio.ByteBuffer;

import static io.netty.util.internal.MathUtil.isOutOfBounds;
import static io.netty.util.internal.StringUtil.NEWLINE;

public class ByteBufferUtil {
    private static final char[] BYTE2CHAR = new char[256];
    private static final char[] HEXDUMP_TABLE = new char[256 * 4];
    private static final String[] HEXPADDING = new String[16];
    private static final String[] HEXDUMP_ROWPREFIXES = new String[65536 >>> 4];
    private static final String[] BYTE2HEX = new String[256];
    private static final String[] BYTEPADDING = new String[16];

    static {
        final char[] DIGITS = "0123456789abcdef".toCharArray();
        for (int i = 0; i < 256; i++) {
            HEXDUMP_TABLE[i << 1] = DIGITS[i >>> 4 & 0x0F];
            HEXDUMP_TABLE[(i << 1) + 1] = DIGITS[i & 0x0F];
        }

        int i;

        // Generate the lookup table for hex dump paddings
        for (i = 0; i < HEXPADDING.length; i++) {
            int padding = HEXPADDING.length - i;
            StringBuilder buf = new StringBuilder(padding * 3);
            for (int j = 0; j < padding; j++) {
                buf.append(" ");
            }
            HEXPADDING[i] = buf.toString();
        }

        // Generate the lookup table for the start-offset header in each row (up
        // to 64KiB).
        for (i = 0; i < HEXDUMP_ROWPREFIXES.length; i++) {
            StringBuilder buf = new StringBuilder(12);
            buf.append(NEWLINE);
            buf.append(Long.toHexString(i << 4 & 0xFFFFFFFFL | 0x100000000L));
            buf.setCharAt(buf.length() - 9, '|');
            buf.append('|');
            HEXDUMP_ROWPREFIXES[i] = buf.toString();
        }

        // Generate the lookup table for byte-to-hex-dump conversion
        for (i = 0; i < BYTE2HEX.length; i++) {
            BYTE2HEX[i] = ' ' + StringUtil.byteToHexStringPadded(i);
        }
    }
}
```



```

// Generate the lookup table for byte dump paddings
for (i = 0; i < BYTEPADDING.length; i++) {
    int padding = BYTEPADDING.length - i;
    StringBuilder buf = new StringBuilder(padding);
    for (int j = 0; j < padding; j++) {
        buf.append(' ');
    }
    BYTEPADDING[i] = buf.toString();
}

// Generate the lookup table for byte-to-char conversion
for (i = 0; i < BYTE2CHAR.length; i++) {
    if (i <= 0x1f || i >= 0x7f) {
        BYTE2CHAR[i] = '.';
    } else {
        BYTE2CHAR[i] = (char) i;
    }
}
}

/**
 * 打印所有内容
 * @param buffer
 */
public static void debugAll(ByteBuffer buffer) {
    int oldlimit = buffer.limit();
    buffer.limit(buffer.capacity());
    StringBuilder origin = new StringBuilder(256);
    appendPrettyHexDump(origin, buffer, 0, buffer.capacity());
    System.out.println("+-----+----- all -----+");
    System.out.printf("position: [%d], limit: [%d]\n", buffer.position(),
oldlimit);
    System.out.println(origin);
    buffer.limit(oldlimit);
}

/**
 * 打印可读取内容
 * @param buffer
 */
public static void debugRead(ByteBuffer buffer) {
    StringBuilder builder = new StringBuilder(256);
    appendPrettyHexDump(builder, buffer, buffer.position(), buffer.limit() -
buffer.position());
    System.out.println("+-----+----- read -----+");
    System.out.printf("position: [%d], limit: [%d]\n", buffer.position(),
buffer.limit());
    System.out.println(builder);
}

private static void appendPrettyHexDump(StringBuilder dump, ByteBuffer buf,
int offset, int length) {
    if (isOutOfBounds(offset, length, buf.capacity())) {

```

```

        throw new IndexOutOfBoundsException(
            "expected: " + "0 <= offset(" + offset + ") <= offset +
length(" + length
            + ") <= " + "buf.capacity(" + buf.capacity() + ')');
    }
    if (length == 0) {
        return;
    }
    dump.append(
        "
        +-----+
        NEWLINE + "      | 0 1 2 3 4 5 6 7 8 9 a
b c d e f |" +
        NEWLINE + "+-----+
        -----+");

    final int startIndex = offset;
    final int fullRows = length >>> 4;
    final int remainder = length & 0xF;

    // Dump the rows which have 16 bytes.
    for (int row = 0; row < fullRows; row++) {
        int rowStartIndex = (row << 4) + startIndex;

        // Per-row prefix.
        appendHexDumpRowPrefix(dump, row, rowStartIndex);

        // Hex dump
        int rowEndIndex = rowStartIndex + 16;
        for (int j = rowStartIndex; j < rowEndIndex; j++) {
            dump.append(BYTE2HEX[getUnsignedByte(buf, j)]);
        }
        dump.append(" |");

        // ASCII dump
        for (int j = rowStartIndex; j < rowEndIndex; j++) {
            dump.append(BYTE2CHAR[getUnsignedByte(buf, j)]);
        }
        dump.append('|');
    }

    // Dump the last row which has less than 16 bytes.
    if (remainder != 0) {
        int rowStartIndex = (fullRows << 4) + startIndex;
        appendHexDumpRowPrefix(dump, fullRows, rowStartIndex);

        // Hex dump
        int rowEndIndex = rowStartIndex + remainder;
        for (int j = rowStartIndex; j < rowEndIndex; j++) {
            dump.append(BYTE2HEX[getUnsignedByte(buf, j)]);
        }
        dump.append(HEXPADDING[remainder]);
        dump.append(" |");

        // Ascii dump
        for (int j = rowStartIndex; j < rowEndIndex; j++) {

```


2.3 ByteBuffer 常见方法

分配空间

可以使用 allocate 方法为 ByteBuffer 分配空间，其它 buffer 类也有该方法

```
ByteBuffer buf = ByteBuffer.allocate(16);
```

注意：这个空间不能改变大小了，netty对这方面做了增强

还有一个方法也可以分配空间

```
import java.nio.ByteBuffer;

public class TestAllocate {
    public static void main(String[] args) {
        System.out.println(ByteBuffer.allocate(10).getClass());
        System.out.println(ByteBuffer.allocateDirect(10).getClass());
        /*
        class java.nio.HeapByteBuffer      -java堆内存，读写效率较低，会受到GC的影响
        class java.nio.DirectByteBuffer      -直接内存，读写效率高（少一次拷贝），不会受
        GC影响，但是它的分配的效率低
        */
    }
}
```

向 buffer 写入数据

有两种办法

- 调用 channel 的 read 方法
- 调用 buffer 自己的 put 方法

```
int readBytes = channel.read(buf);
```

和

```
buf.put((byte)127);
```

从 buffer 读取数据

同样有两种办法

- 调用 channel 的 write 方法
- 调用 buffer 自己的 get 方法

```
int writeBytes = channel.write(buf);
```

和

```
byte b = buf.get();
```

get 方法会让 position 读指针向后走，如果想重复读取数据

- 可以调用 rewind 方法将 position 重新置为 0
- 或者调用 get(int i) 方法获取索引 i 的内容，它不会移动读指针

mark 和 reset

mark 是在读取时，做一个标记，即使 position 改变，只要调用 reset 就能回到 mark 的位置,相当于 mark是对rewind方法的一个增强

```
public class TestRead {
    public static void main(String[] args) {
        ByteBuffer byteBuffer = ByteBuffer.allocate(10);
        byteBuffer.put(new byte[]{'a', 'b', 'c', 'd'});
        byteBuffer.flip();

        //取出第一个字节，也就是a
        System.out.println((char) byteBuffer.get());
        //在b的位置上加一个mark
        byteBuffer.mark();
        System.out.println((char) byteBuffer.get());
        //回到mark的位置
        byteBuffer.reset();
        System.out.println((char) byteBuffer.get());
    }
}
```

TestRead x 1

D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\bin\java.exe ...

a
b
b

注意

rewind 和 flip 都会清除 mark 位置

字符串与 ByteBuffer 互转

用string给buffer赋值

```
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;

public class TestString {
    public static void main(String[] args) {
        //1.getBytes
        ByteBuffer byteBuffer1 = ByteBuffer.allocate(16);
        byteBuffer1.put("hello".getBytes());
        ByteBufferUtil.debugAll(byteBuffer1);
        //2.Charset
        ByteBuffer byteBuffer2 = StandardCharsets.UTF_8.encode("hello");
        ByteBufferUtil.debugAll(byteBuffer2);
        //3.wrap
        ByteBuffer byteBuffer3 = ByteBuffer.wrap("hello".getBytes());
        ByteBufferUtil.debugAll(byteBuffer3);
    }
}
```

输出

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|00000000| 68 65 6c 6c 6f 00 00 00 00 00 00 00 00 00 00 |hello.....|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|00000000| 68 65 6c 6c 6f                                     |hello      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|00000000| 68 65 6c 6c 6f                                     |hello      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

Process finished with exit code 0
```

读取buffer

```
String string = StandardCharsets.UTF_8.decode(byteBuffer2).toString();
System.out.println(string);
```

⚠ Buffer 的线程安全

Buffer 是非线程安全的

2.4 Scattering Reads

分散读取，有一个文本文件 3parts.txt

```
onetwothree
```

使用如下方式读取，可以将数据填充至多个 buffer

```
try (RandomAccessFile file = new RandomAccessFile("3parts.txt", "rw")) {
    FileChannel channel = file.getChannel();
    ByteBuffer a = ByteBuffer.allocate(3);
    ByteBuffer b = ByteBuffer.allocate(3);
    ByteBuffer c = ByteBuffer.allocate(5);
    channel.read(new ByteBuffer[]{a, b, c});
    a.flip();
    b.flip();
    c.flip();
    debug(a);
```

```

        debug(b);
        debug(c);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

结果

```

+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 6f 6e 65                                     |one          |
+-----+
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 74 77 6f                                     |two          |
+-----+
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 74 68 72 65 65                             |three        |
+-----+

```

2.5 Gathering Writes

集中写

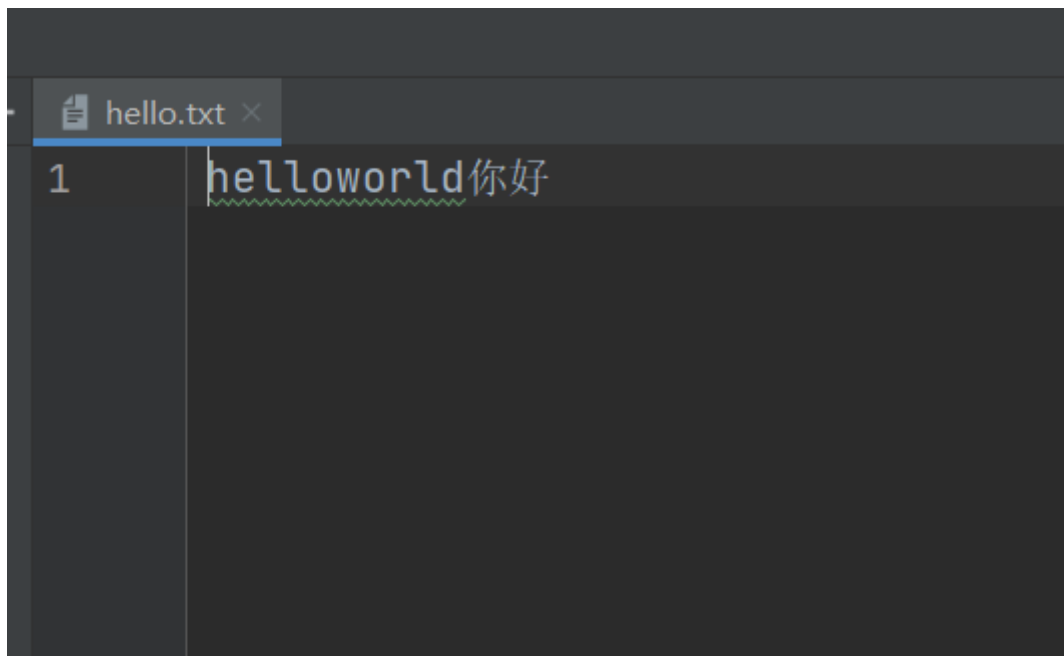
使用如下方式写入，可以将多个 buffer 的数据填充至 channel

```

public class TestGatheringWrites {
    public static void main(String[] args) {
        ByteBuffer b1 = StandardCharsets.UTF_8.encode(str: "hello");
        ByteBuffer b2 = StandardCharsets.UTF_8.encode(str: "world");
        ByteBuffer b3 = StandardCharsets.UTF_8.encode(str: "你好");
        try (FileChannel channel = new RandomAccessFile(name: "hello.txt", mode: "rw").getChannel()) {
            channel.write(new ByteBuffer[]{b1,b2,b3});
        } catch (IOException e) {}
    }
}

```

输出



2.6 练习

网络上有多条数据发送给服务端，数据之间使用 \n 进行分隔
但由于某种原因这些数据在接收时，被进行了重新组合，例如原始数据有3条为

- Hello,world\n
- I'm zhangsan\n
- How are you?\n

变成了下面的两个 byteBuffer (黏包，半包)

- Hello,world\nI'm zhangsan\nHo
- w are you?\n

现在要求你编写程序，将错乱的数据恢复成原始的按 \n 分隔的数据

```
import java.nio.ByteBuffer;

public class TestDemo {
    public static void main(String[] args) {
        ByteBuffer source = ByteBuffer.allocate(32);
```

```

//                                11                                24
source.put("Hello,world\nI'm zhangsan\nHo".getBytes());
split(source);

source.put("w are you?\nhaha!\n".getBytes());
split(source);
}

private static void split(ByteBuffer source) {
    //开启读取
    source.flip();
    int oldLimit = source.limit();
    for (int i = 0; i < oldLimit; i++) {
        if (source.get(i) == '\n') { //说明前面是一句话
            System.out.println(i);
            //给这个buffer分配空间，计算一下就可以
            ByteBuffer target = ByteBuffer.allocate(i + 1 -
source.position());
            // 设置需要写到的limit，这时候不用担心limit会改变，之前我们是存储过
oldLimit的

            source.limit(i + 1);
            target.put(source); // 从source 读，向 target 写
            ByteBufferUtil.debugAll(target);
            source.limit(oldLimit);
        }
    }
    source.compact();
}
}

```

3. 文件编程

3.1 FileChannel

⚠ FileChannel 工作模式

FileChannel 只能工作在阻塞模式下

获取

不能直接打开 FileChannel，必须通过 FileInputStream、FileOutputStream 或者 RandomAccessFile 来获取 FileChannel，它们都有 getChannel 方法

- 通过 FileInputStream 获取的 channel 只能读
- 通过 FileOutputStream 获取的 channel 只能写
- 通过 RandomAccessFile 是否能读写根据构造 RandomAccessFile 时的读写模式决定

读取

会从 channel 读取数据填充 ByteBuffer，返回值表示读到了多少字节，-1 表示到达了文件的末尾

```
int readBytes = channel.read(buffer);
```

写入

写入的正确姿势如下， SocketChannel

```
ByteBuffer buffer = ...;
buffer.put(...); // 存入数据
buffer.flip();   // 切换读模式

while(buffer.hasRemaining()) {
    channel.write(buffer);
}
```

在 while 中调用 channel.write 是因为 write 方法并不能保证一次将 buffer 中的内容全部写入 channel

关闭

channel 必须关闭，不过调用了 FileInputStream、FileOutputStream 或者 RandomAccessFile 的 close 方法会间接地调用 channel 的 close 方法

位置

获取当前位置

```
long pos = channel.position();
```

设置当前位置

```
long newPos = ...;
channel.position(newPos);
```

设置当前位置时，如果设置为文件的末尾

- 这时读取会返回 -1
- 这时写入，会追加内容，但要注意如果 position 超过了文件末尾，再写入时在新内容和原末尾之间会有空洞 (00)

大小

使用 size 方法获取文件的大小

强制写入

操作系统出于性能的考虑，会将数据缓存，不是立刻写入磁盘。可以调用 force(true) 方法将文件内容和元数据（文件的权限等信息）立刻写入磁盘

3.2 两个 Channel 传输数据

主要介绍了transferFrom和transferTo这两个方法

```
package com.xth.nio.file;

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

public class NIOFileTransferFrom {
    public static void main(String[] args) throws Exception{
        //创建流
        FileInputStream fileInputStream = new
FileInputStream("src/main/java/com/xth/nio/file/1.jpg");
        FileOutputStream fileOutputStream = new
FileOutputStream("src/main/java/com/xth/nio/file/2.jpg");
        //创建相应的channel
        FileChannel channel1 = fileInputStream.getChannel();
        FileChannel channel2 = fileOutputStream.getChannel();
        //开始copy，效率高，底层会利用操作系统的零拷贝进行优化
        //下面两种都是从channel1中读取数据拷贝到channel2
        channel2.transferFrom(channel1,0,channel1.size());
        channel1.transferTo(0,channel1.size(),channel2);
        //关闭相应的流
    }
}
```

```

        fileInputStream.close();
        fileOutputStream.close();
    }
}

```

超过 2g 大小的文件传输

原理是transferTo方法一次最多只能传输2G大小的数据，如果总数据大小超过2G，需要进行多次传输

```

public class TestFileChannelTransferTo {
    public static void main(String[] args) {
        try (
            FileChannel from = new FileInputStream("data.txt").getChannel();
            FileChannel to = new FileOutputStream("to.txt").getChannel();
        ) {
            // 效率高，底层会利用操作系统的零拷贝进行优化
            long size = from.size();
            // left 变量代表还剩余多少字节
            for (long left = size; left > 0; ) {
                System.out.println("当前位置为第:" + (size - left) + "个字节"
                    + "      剩余字节有:" + left);

                //transferTo的返回值是这次传输了多少字节
                left -= from.transferTo((size - left), left, to);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

3.3 Path

jdk7 引入了 Path 和 Paths 类

- Path 用来表示文件路径
- Paths 是工具类，用来获取 Path 实例

```
Path source = Paths.get("1.txt"); // 相对路径 使用 user.dir 环境变量来定位 1.txt

Path source = Paths.get("d:\\1.txt"); // 绝对路径 代表了 d:\\1.txt

Path source = Paths.get("d:/1.txt"); // 绝对路径 同样代表了 d:\\1.txt

Path projects = Paths.get("d:\\data", "projects"); // 代表了 d:\\data\\projects
```

- `.` 代表了当前路径
- `..` 代表了上一级路径

例如目录结构如下

```
d:
|- data
  |- projects
    |- a
    |- b
```

代码

```
Path path = Paths.get("d:\\data\\projects\\a\\..\\b");
System.out.println(path);
// 正常化路径,把路径格式化成我们平时见的路径
System.out.println(path.normalize());
```

会输出

```
// 下面这俩是等价的
d:\data\projects\a\..\b
d:\data\projects\b
```

3.4 Files

检查文件是否存在

```
Path path = Paths.get("helloworld/data.txt");
System.out.println(Files.exists(path));
```

创建一级目录

```
Path path = Paths.get("helloworld/d1");
Files.createDirectory(path);
```

- 如果目录已存在，会抛异常 `FileAlreadyExistsException`
- 不能一次创建多级目录，否则会抛异常 `NoSuchFileException`

创建多级目录用

```
Path path = Paths.get("helloworld/d1/d2");
Files.createDirectories(path);
```

拷贝文件

```
Path source = Paths.get("helloworld/data.txt");
Path target = Paths.get("helloworld/target.txt");

Files.copy(source, target);
```

- 如果文件已存在，会抛异常 `FileAlreadyExistsException`

如果希望用 source 覆盖掉 target，需要用 `StandardCopyOption` 来控制

```
Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
```

移动文件

```
Path source = Paths.get("helloworld/data.txt");
Path target = Paths.get("helloworld/data.txt");

Files.move(source, target, StandardCopyOption.ATOMIC_MOVE);
```

- StandardCopyOption.ATOMIC_MOVE 保证文件移动的原子性

删除文件

```
Path target = Paths.get("helloworld/target.txt");

Files.delete(target);
```

- 如果文件不存在，会抛异常 NoSuchFileException

删除目录

```
Path target = Paths.get("helloworld/d1");

Files.delete(target);
```

- 如果目录还有内容，会抛异常 DirectoryNotEmptyException

遍历目录文件

```
public static void main(String[] args) throws IOException {
    Path path = Paths.get("C:\\Program Files\\Java\\jdk1.8.0_91");
    //创建两个计数器，注意不能用int类型的变量
    AtomicInteger dirCount = new AtomicInteger();
    AtomicInteger fileCount = new AtomicInteger();
    Files.walkFileTree(path, new SimpleFileVisitor<Path>(){
        @Override
        public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes
attrs)

            throws IOException {
                System.out.println(dir);
                dirCount.incrementAndGet();
                //注意这里不要修改返回值
                return super.preVisitDirectory(dir, attrs);
            }
    });
}
```



```

    }

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
        throws IOException {
        System.out.println(file);
        fileCount.incrementAndGet();
        //注意这里不要修改返回值
        return super.visitFile(file, attrs);
    }
});
//打印统计数量
System.out.println(dirCount);
System.out.println(fileCount);
}

```

统计 jar 的数目

```

Path path = Paths.get("C:\\Program Files\\Java\\jdk1.8.0_91");
AtomicInteger fileCount = new AtomicInteger();
Files.walkFileTree(path, new SimpleFileVisitor<Path>(){
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
        throws IOException {
        if (file.toFile().getName().endsWith(".jar")) { //如果以.jar结尾
            //就给数量+1
            fileCount.incrementAndGet();
        }
        //注意这里不要修改返回值

        return super.visitFile(file, attrs);
    }
});
System.out.println(fileCount);

```

删除多级目录

```

Path path = Paths.get("d:\\a");
Files.walkFileTree(path, new SimpleFileVisitor<Path>(){
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
        throws IOException {
        Files.delete(file);
        return super.visitFile(file, attrs);
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc)

```

```

        throws IOException {
            Files.delete(dir);
            return super.postVisitDirectory(dir, exc);
        }
    });
}

```

⚠ 删除很危险

删除是危险操作，确保要递归删除的文件夹没有重要内容

拷贝多级目录

```

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.function.Consumer;

public class TestFileCopy {
    public static void main(String[] args) throws IOException {
        String source = "D:\\Snipaste-1.16.2-x64";
        String target = "D:\\Snipaste-1.16.2-x64aaa";

        Files.walk(Paths.get(source)).forEach(new Consumer<Path>() {
            @Override
            public void accept(Path path) {
                try {
                    String targetName = path.toString().replace(source, target);
                    // 如果是目录
                    if (Files.isDirectory(path)) {
                        Files.createDirectory(Paths.get(targetName));
                    }
                    // 如果是普通文件
                    else if (Files.isRegularFile(path)) {
                        Files.copy(path, Paths.get(targetName));
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}

```

4. 网络编程

4.1 非阻塞 vs 阻塞

阻塞

- 阻塞模式下，相关方法都会导致线程暂停
 - `ServerSocketChannel.accept` 会在没有连接建立时让线程暂停
 - `SocketChannel.read` 会在没有数据可读时让线程暂停
 - 阻塞的表现其实就是线程暂停了，暂停期间不会占用 cpu，但线程相当于闲置
- 单线程下，阻塞方法之间相互影响，几乎不能正常工作，需要多线程支持
- 但多线程下，有新的问题，体现在以下方面
 - 32 位 jvm 一个线程 320k，64 位 jvm 一个线程 1024k，如果连接数过多，必然导致 OOM，并且线程太多，反而会因为频繁上下文切换导致性能降低
 - 可以采用线程池技术来减少线程数和线程上下文切换，但治标不治本，如果有很多连接建立，但长时间 inactive，会阻塞线程池中所有线程，因此不适合长连接，只适合短连接

服务器端

```
package c4;

import lombok.extern.slf4j.Slf4j;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.ArrayList;
import java.util.List;

@Slf4j
public class Server {
    public static void main(String[] args) throws IOException {
        // 使用 nio 来理解阻塞模式，单线程
        // 0. ByteBuffer
        ByteBuffer buffer = ByteBuffer.allocate(16);
        // 1. 创建了服务器
        ServerSocketChannel ssc = ServerSocketChannel.open();

        // 2. 绑定监听端口
        ssc.bind(new InetSocketAddress(8080));

        // 3. 连接集合
        List<SocketChannel> channels = new ArrayList<>();
        while (true) {
            // 4. accept 建立与客户端连接， SocketChannel 用来与客户端之间通信
            log.debug("connecting...");
            //阻塞方法，线程停止运行
            SocketChannel sc = ssc.accept();
```

```

log.debug("connected... {}", sc);
channels.add(sc);
for (SocketChannel channel : channels) {
    // 5. 接收客户端发送的数据
    log.debug("before read... {}", channel);
    // read也是一个阻塞方法，使线程停止运行
    channel.read(buffer);
    buffer.flip();
    ByteBufferUtil.debugRead(buffer);
    buffer.clear();
    log.debug("after read...{}", channel);
}
}
}
}

```

客户端

```

SocketChannel sc = SocketChannel.open();
sc.connect(new InetSocketAddress("localhost", 8080));
System.out.println("waiting...");

```

非阻塞

- 非阻塞模式下，相关方法都不会让线程暂停
 - 在 ServerSocketChannel.accept 在没有连接建立时，会返回 null，继续运行
 - SocketChannel.read 在没有数据可读时，会返回 0，但线程不必阻塞，可以去执行其它 SocketChannel 的 read 或是去执行 ServerSocketChannel.accept
 - 写数据时，线程只是等待数据写入 Channel 即可，无需等 Channel 通过网络把数据发送出去
- 但非阻塞模式下，即使没有连接建立，和可读数据，线程仍然在不断运行，白白浪费了 cpu
- 数据复制过程中，线程实际还是阻塞的（AIO 改进的地方）

服务器端，客户端代码不变

主要修改的是这两句代码

- ssc.configureBlocking(false); // 非阻塞模式
- sc.configureBlocking(false); // 非阻塞模式

第一句影响的是accept方法，让accept方法不再阻塞

第二句影响的是read方法，让read方法不再阻塞

```

// 使用 nio 来理解非阻塞模式，单线程
// 0. ByteBuffer
ByteBuffer buffer = ByteBuffer.allocate(16);

```

```

// 1. 创建了服务器
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.configureBlocking(false); // 非阻塞模式
// 2. 绑定监听端口
ssc.bind(new InetSocketAddress(8080));
// 3. 连接集合
List<SocketChannel> channels = new ArrayList<>();
while (true) {
    // 4. accept 建立与客户端连接， SocketChannel 用来与客户端之间通信
    SocketChannel sc = ssc.accept(); // 非阻塞，线程还会继续运行，如果没有连接建立，但sc
    是null
    if (sc != null) {
        log.debug("connected... {}", sc);
        sc.configureBlocking(false); // 非阻塞模式
        channels.add(sc);
    }
    for (SocketChannel channel : channels) {
        // 5. 接收客户端发送的数据
        int read = channel.read(buffer); // 非阻塞，线程仍然会继续运行，如果没有读到数
        据，read 返回 0
        if (read > 0) {
            buffer.flip();
            debugRead(buffer);
            buffer.clear();
            log.debug("after read...{}", channel);
        }
    }
}
}

```

多路复用

单线程可以配合 Selector 完成对多个 Channel 可读写事件的监控，这称之为多路复用

- 多路复用仅针对网络 IO、普通文件 IO 没法利用多路复用
- 如果不用 Selector 的非阻塞模式，线程大部分时间都在做无用功，而 Selector 能够保证
 - 有可连接事件时才去连接
 - 有可读事件才去读取
 - 有可写事件才去写入
- 限于网络传输能力，Channel 未必时时可写，一旦 Channel 可写，会触发 Selector 的可写事件

4.2 Selector

```
graph TD
    subgraph selector 版
        thread --> selector
        selector --> c1(channel)
        selector --> c2(channel)
        selector --> c3(channel)
    end
    end
```

好处

- 一个线程配合 selector 就可以监控多个 channel 的事件，事件发生线程才去处理。避免非阻塞模式下所做无用功
- 让这个线程能够被充分利用
- 节约了线程的数量
- 减少了线程上下文切换

创建

```
Selector selector = Selector.open();
```

绑定 Channel 事件

也称之为注册事件，绑定的事件 selector 才会关心

SelectionKey是：当事件发生时，通过它可以知道事件是什么类型和哪个channel的事件

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, 绑定事件);
```

- channel 必须工作在非阻塞模式
- FileChannel 没有非阻塞模式，因此不能配合 selector 一起使用
- 绑定的事件类型可以有
 - connect - 客户端连接成功时触发
 - accept - 服务器端成功接受连接时触发
 - read - 数据可读入时触发，有因为接收能力弱，数据暂不能读入的情况
 - write - 数据可写出时触发，有因为发送能力弱，数据暂不能写出的情况

监听 Channel 事件

可以通过下面三种方法来监听是否有事件发生，方法的返回值代表有多少 channel 发生了事件

方法1，阻塞直到绑定事件发生（也就是说，如果没有绑定事件的发生，就一直阻塞在这里）

```
int count = selector.select();
```

方法2，阻塞直到绑定事件发生，或是超时（时间单位为 ms）

```
int count = selector.select(long timeout);
```

方法3，不会阻塞，也就是不管有没有事件，立刻返回，自己根据返回值检查是否有事件

```
int count = selector.selectNow();
```

💡 select 何时不阻塞

- 事件发生时
 - 客户端发起连接请求，会触发 accept 事件
 - 客户端发送数据过来，客户端正常、异常关闭时，都会触发 read 事件，另外如果发送的数据大于 buffer 缓冲区，会触发多次读取事件
 - channel 可写，会触发 write 事件
 - 在 linux 下 nio bug 发生时
- 调用 selector.wakeup()
- 调用 selector.close()
- selector 所在线程 interrupt

4.3 处理 accept 事件

客户端代码为

```
public class Client {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 8080)) {
            System.out.println(socket);
            socket.getOutputStream().write("world".getBytes());
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

服务器端代码为

一定要把迭代器中的元素给清除

```
package c4;

import lombok.extern.slf4j.Slf4j;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

@Slf4j
public class ChannelDemo6 {
    public static void main(String[] args) {
        try (ServerSocketChannel channel = ServerSocketChannel.open()) {
            //给channel绑定端口
            channel.bind(new InetSocketAddress(8080));
            //设置accept方法为非阻塞方法
            channel.configureBlocking(false);

            //创建selector
            Selector selector = Selector.open();
            //给channel绑定selector，并监听accept事件
            channel.register(selector, SelectionKey.OP_ACCEPT);
            while (true) {
                //如果没有事件发生，会一直阻塞在这里
                int count = selector.select();
                log.debug("select count: {}", count);
            }
        }
    }
}
```



```

// 获取所有事件
Set<SelectionKey> keys = selector.selectedKeys();

// 遍历所有事件，逐一处理
Iterator<SelectionKey> iter = keys.iterator();
while (iter.hasNext()) {
    SelectionKey key = iter.next();
    // 判断事件类型
    if (key.isAcceptable()) {
        ServerSocketChannel c = (ServerSocketChannel)
key.channel();

        // 必须处理或者取消这个事件
        SocketChannel sc = c.accept();
        //或者取消，这两个必须要有一个
        //key.cancel();
        log.debug("{} ", sc);
    }
    // 处理完毕，必须将事件从selectedKeys中移除
    iter.remove();
}
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

💡 事件发生后能否不处理

事件发生后，要么处理，要么取消（cancel），不能什么都不做，否则下次该事件仍会触发，这是因为 nio 底层使用的是水平触发

4.4 处理 read 事件

```

@Slf4j
public class ChannelDemo6 {
    public static void main(String[] args) {
        try (ServerSocketChannel channel = ServerSocketChannel.open()) {
            channel.bind(new InetSocketAddress(8080));
            System.out.println(channel);
            Selector selector = Selector.open();
            channel.configureBlocking(false);
            channel.register(selector, SelectionKey.OP_ACCEPT);

            while (true) {
                int count = selector.select();
                // int count = selector.selectNow();
                log.debug("select count: {}", count);
            }
        }
    }
}

```

```

//          if(count <= 0) {
//              continue;
//          }

// 获取所有事件
Set<SelectionKey> keys = selector.selectedKeys();

// 遍历所有事件，逐一处理
Iterator<SelectionKey> iter = keys.iterator();
while (iter.hasNext()) {
    SelectionKey key = iter.next();
    // 判断事件类型
    if (key.isAcceptable()) {
        ServerSocketChannel c = (ServerSocketChannel)
key.channel();

        // 必须处理
        SocketChannel sc = c.accept();
        sc.configureBlocking(false);
        sc.register(selector, SelectionKey.OP_READ);
        log.debug("连接已建立: {}", sc);
    } else if (key.isReadable()) {
        SocketChannel sc = (SocketChannel) key.channel();
        ByteBuffer buffer = ByteBuffer.allocate(128);
        //如果客户端正常断开，返回-1
        int read = sc.read(buffer);
        if(read == -1) {
            key.cancel();
            sc.close();
        } else {
            buffer.flip();
            debug(buffer);
        }
    }
    // 处理完毕，必须将事件移除
    iter.remove();
}
}
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

开启两个客户端，修改一下发送文字，输出

```

sun.nio.ch.ServerSocketChannelImpl[/0:0:0:0:0:0:0:8080]
21:16:39 [DEBUG] [main] c.i.n.ChannelDemo6 - select count: 1
21:16:39 [DEBUG] [main] c.i.n.ChannelDemo6 - 连接已建立:
java.nio.channels.SocketChannel[connected local=/127.0.0.1:8080
remote=/127.0.0.1:60367]
21:16:39 [DEBUG] [main] c.i.n.ChannelDemo6 - select count: 1
+-----+

```

```

      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+-----+
|00000000| 68 65 6c 6c 6f                                     |hello      |
+-----+-----+-----+-----+-----+
21:16:59 [DEBUG] [main] c.i.n.ChannelDemo6 - select count: 1
21:16:59 [DEBUG] [main] c.i.n.ChannelDemo6 - 连接已建立:
java.nio.channels.SocketChannel[connected local=/127.0.0.1:8080
remote=/127.0.0.1:60378]
21:16:59 [DEBUG] [main] c.i.n.ChannelDemo6 - select count: 1
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+-----+
|00000000| 77 6f 72 6c 64                                     |world      |
+-----+-----+-----+-----+-----+

```

💡 为何要 `iter.remove()`

因为 `select` 在事件发生后，就会将相关的 `key` 放入 `selectedKeys` 集合，但不会在处理完后从 `selectedKeys` 集合中移除，需要我们自己编码删除。例如

- 第一次触发了 `ssckey` 上的 `accept` 事件，没有移除 `ssckey`
- 第二次触发了 `sckey` 上的 `read` 事件，但这时 `selectedKeys` 中还有上次的 `ssckey`，在处理时因为没有真正的 `serverSocket` 连上了，就会导致空指针异常

💡 `cancel` 的作用

`cancel` 会取消注册在 `selector` 上的 `channel`，并从 `keys` 集合中删除 `key` 后续不会再监听事件

⚠️ 不处理边界的问题

以前有同学写过这样的代码，思考注释中两个问题，以 `bio` 为例，其实 `nio` 道理是一样的

```

public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket ss=new ServerSocket(9000);
        while (true) {
            Socket s = ss.accept();
            InputStream in = s.getInputStream();
            // 这里这么写，有没有问题
            byte[] arr = new byte[4];
            while(true) {
                int read = in.read(arr);

```

```

        // 这里这么写，有没有问题
        if(read == -1) {
            break;
        }
        System.out.println(new String(arr, 0, read));
    }
}
}
}

```

客户端

```

public class Client {
    public static void main(String[] args) throws IOException {
        Socket max = new Socket("localhost", 9000);
        OutputStream out = max.getOutputStream();
        out.write("hello".getBytes());
        out.write("world".getBytes());
        out.write("你好".getBytes());
        max.close();
    }
}

```

输出

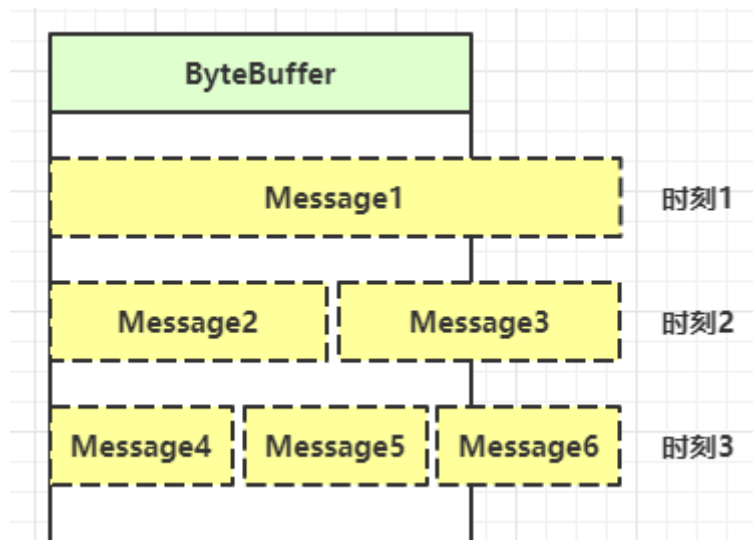
```

hell
owor
ld💎
💎好

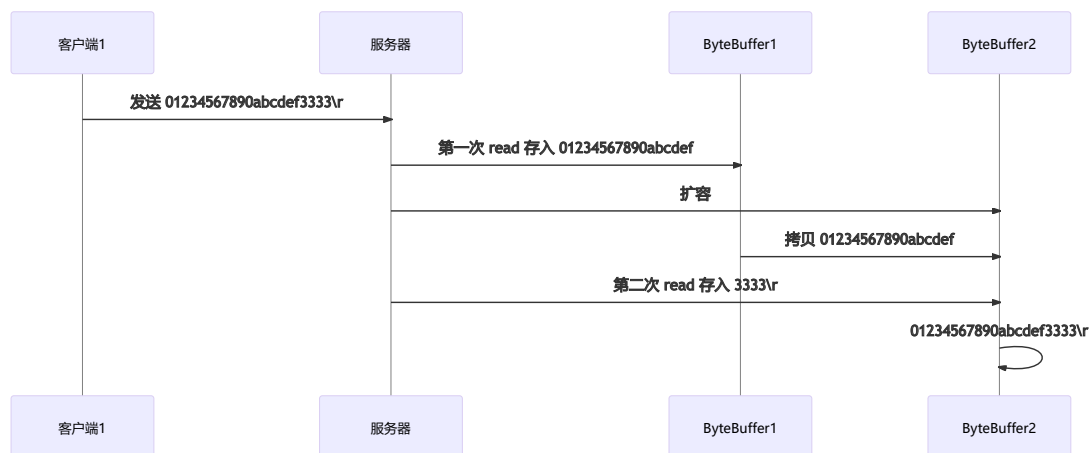
```

为什么？

处理消息的边界



1. 一种思路是固定消息长度，数据包大小一样，服务器按预定长度读取，缺点是浪费带宽
2. 另一种思路是按分隔符拆分，缺点是效率低
3. 第三种是TLV 格式，即 Type 类型、Length 长度、Value 数据，类型和长度已知的情况下，就可以方便获取消息大小，分配合适的 buffer，缺点是 buffer 需要提前分配，如果内容过大，则影响 server 吞吐量
4.
 1. Http 1.1 是 TLV 格式
 2. Http 2.0 是 LTV 格式



服务器端

```
private static void split(ByteBuffer source) {
    source.flip();
    for (int i = 0; i < source.limit(); i++) {
        // 找到一条完整消息
        if (source.get(i) == '\n') {
            int length = i + 1 - source.position();
            // 把这条完整消息存入新的 ByteBuffer
            ByteBuffer target = ByteBuffer.allocate(length);
```

```

        // 从 source 读, 向 target 写
        for (int j = 0; j < length; j++) {
            target.put(source.get());
        }
        debugAll(target);
    }
}
source.compact(); // 0123456789abcdef position 16 limit 16
}

public static void main(String[] args) throws IOException {
    // 1. 创建 selector, 管理多个 channel
    Selector selector = Selector.open();
    ServerSocketChannel ssc = ServerSocketChannel.open();
    ssc.configureBlocking(false);
    // 2. 建立 selector 和 channel 的联系 (注册)
    // SelectionKey 就是将来事件发生后, 通过它可以知道事件和哪个 channel 的事件
    SelectionKey sscKey = ssc.register(selector, 0, null);
    // key 只关注 accept 事件
    sscKey.interestOps(SelectionKey.OP_ACCEPT);
    log.debug("sscKey:{}", sscKey);
    ssc.bind(new InetSocketAddress(8080));
    while (true) {
        // 3. select 方法, 没有事件发生, 线程阻塞, 有事件, 线程才会恢复运行
        // select 在事件未处理时, 它不会阻塞, 事件发生后要么处理, 要么取消, 不能置之不理
        selector.select();
        // 4. 处理事件, selectedKeys 内部包含了所有发生的事件
        Iterator<SelectionKey> iter = selector.selectedKeys().iterator(); //
accept, read
        while (iter.hasNext()) {
            SelectionKey key = iter.next();
            // 处理key 时, 要从 selectedKeys 集合中删除, 否则下次处理就会有问题
            iter.remove();
            log.debug("key: {}", key);
            // 5. 区分事件类型
            if (key.isAcceptable()) { // 如果是 accept
                ServerSocketChannel channel = (ServerSocketChannel)
key.channel();
                SocketChannel sc = channel.accept();
                sc.configureBlocking(false);
                ByteBuffer buffer = ByteBuffer.allocate(16); // attachment
                // 将一个 ByteBuffer 作为附件关联到 selectionKey 上
                SelectionKey scKey = sc.register(selector, 0, buffer);
                scKey.interestOps(SelectionKey.OP_READ);
                log.debug("{} ", sc);
                log.debug("scKey:{}", scKey);
            } else if (key.isReadable()) { // 如果是 read
                try {
                    SocketChannel channel = (SocketChannel) key.channel(); // 拿
到触发事件的channel
                    // 获取 selectionKey 上关联的附件
                    ByteBuffer buffer = (ByteBuffer) key.attachment();
                    int read = channel.read(buffer); // 如果是正常断开, read 的方法的
返回值是 -1
                    if (read == -1) {

```

```

        key.cancel();
    } else {
        split(buffer);
        // 需要扩容
        if (buffer.position() == buffer.limit()) {
            ByteBuffer newBuffer =
ByteBuffer.allocate(buffer.capacity() * 2);
            buffer.flip();
            newBuffer.put(buffer); // 0123456789abcdef3333\n
            key.attach(newBuffer);
        }
    }

    } catch (IOException e) {
        e.printStackTrace();
        key.cancel(); // 因为客户端断开了,因此需要将 key 取消 (从
selector 的 keys 集合中真正删除 key)
    }
}
}
}
}
}

```

客户端

```

SocketChannel sc = SocketChannel.open();
sc.connect(new InetSocketAddress("localhost", 8080));
SocketAddress address = sc.getLocalAddress();
// sc.write(Charset.defaultCharset().encode("hello\nworld\n"));
sc.write(Charset.defaultCharset().encode("0123\n456789abcdef"));
sc.write(Charset.defaultCharset().encode("0123456789abcdef3333\n"));
System.in.read();

```

ByteBuffer 大小分配

- 每个 channel 都需要记录可能被切分的消息，因为 ByteBuffer 不能被多个 channel 共同使用，因此需要为每个 channel 维护一个独立的 ByteBuffer
- ByteBuffer 不能太大，比如一个 ByteBuffer 1Mb 的话，要支持百万连接就要 1Tb 内存，因此需要设计大小可变的 ByteBuffer
 - 一种思路是首先分配一个较小的 buffer，例如 4k，如果发现数据不够，再分配 8k 的 buffer，将 4k buffer 内容拷贝至 8k buffer，优点是消息连续容易处理，缺点是数据拷贝耗费性能，参考实现 <http://tutorials.jenkov.com/java-performance/resizable-array.html>
 - 另一种思路是用多个数组组成 buffer，一个数组不够，把多出来的内容写入新的数组，与前面的区别是消息存储不连续解析复杂，优点是避免了拷贝引起的性能损耗

4.5 处理 write 事件

一次无法写完例子

- 非阻塞模式下，无法保证把 buffer 中所有数据都写入 channel，因此需要追踪 write 方法的返回值（代表实际写入字节数）
- 用 selector 监听所有 channel 的可写事件，每个 channel 都需要一个 key 来跟踪 buffer，但这样又会导致占用内存过多，就有两阶段策略
 - 当消息处理器第一次写入消息时，才将 channel 注册到 selector 上
 - selector 检查 channel 上的可写事件，如果所有的数据写完了，就取消 channel 的注册
 - 如果不取消，会每次可写均会触发 write 事件

```
public class WriteServer {

    public static void main(String[] args) throws IOException {
        ServerSocketChannel ssc = ServerSocketChannel.open();
        ssc.configureBlocking(false);
        ssc.bind(new InetSocketAddress(8080));

        Selector selector = Selector.open();
        ssc.register(selector, SelectionKey.OP_ACCEPT);

        while(true) {
            selector.select();

            Iterator<SelectionKey> iter = selector.selectedKeys().iterator();
            while (iter.hasNext()) {
                SelectionKey key = iter.next();
                iter.remove();
                if (key.isAcceptable()) {
                    SocketChannel sc = ssc.accept();
                    sc.configureBlocking(false);
                    SelectionKey sckey = sc.register(selector,
SelectionKey.OP_READ);
                    // 1. 向客户端发送内容
                    StringBuilder sb = new StringBuilder();
                    for (int i = 0; i < 3000000; i++) {
                        sb.append("a");
                    }
                    ByteBuffer buffer =
Charset.defaultCharset().encode(sb.toString());
                    int write = sc.write(buffer);
                    // 3. write 表示实际写了多少字节
                    System.out.println("实际写入字节:" + write);
                    // 4. 如果有剩余未读字节，才需要关注写事件
                    if (buffer.hasRemaining()) {
                        // read 1 write 4
                        // 在原有关注事件的基础上，多关注 写事件
                    }
                }
            }
        }
    }
}
```



```

        sckey.interestOps(sckey.interestOps() +
SelectionKey.OP_WRITE);
        // 把 buffer 作为附件加入 sckey
        sckey.attach(buffer);
    }
} else if (key.isWritable()) {
    ByteBuffer buffer = (ByteBuffer) key.attachment();
    SocketChannel sc = (SocketChannel) key.channel();
    int write = sc.write(buffer);
    System.out.println("实际写入字节:" + write);
    if (!buffer.hasRemaining()) { // 写完了
        key.interestOps(key.interestOps() -
SelectionKey.OP_WRITE);
        key.attach(null);
    }
}
}
}
}
}
}
}
}

```

客户端

```

public class WriteClient {
    public static void main(String[] args) throws IOException {
        Selector selector = Selector.open();
        SocketChannel sc = SocketChannel.open();
        sc.configureBlocking(false);
        sc.register(selector, SelectionKey.OP_CONNECT | SelectionKey.OP_READ);
        sc.connect(new InetSocketAddress("localhost", 8080));
        int count = 0;
        while (true) {
            selector.select();
            Iterator<SelectionKey> iter = selector.selectedKeys().iterator();
            while (iter.hasNext()) {
                SelectionKey key = iter.next();
                iter.remove();
                if (key.isConnectable()) {
                    System.out.println(sc.finishConnect());
                } else if (key.isReadable()) {
                    ByteBuffer buffer = ByteBuffer.allocate(1024 * 1024);
                    count += sc.read(buffer);
                    buffer.clear();
                    System.out.println(count);
                }
            }
        }
    }
}

```

💡 write 为何要取消

只要向 channel 发送数据时，socket 缓冲可写，这个事件会频繁触发，因此应当只在 socket 缓冲区写不下时再关注可写事件，数据写完之后再次取消关注

4.6 更进一步

💡 利用多线程优化

现在都是多核 cpu，设计时要充分考虑别让 cpu 的力量被白白浪费

前面的代码只有一个选择器，没有充分利用多核 cpu，如何改进呢？

分两组选择器

- 单线程配一个选择器，专门处理 accept 事件
- 创建 cpu 核心数的线程，每个线程配一个选择器，轮流处理 read 事件

```
public class ChannelDemo7 {
    public static void main(String[] args) throws IOException {
        new BossEventLoop().register();
    }

    @Slf4j
    static class BossEventLoop implements Runnable {
        private Selector boss;
        private WorkerEventLoop[] workers;
        private volatile boolean start = false;
        AtomicInteger index = new AtomicInteger();

        public void register() throws IOException {
            if (!start) {
                ServerSocketChannel ssc = ServerSocketChannel.open();
                ssc.bind(new InetSocketAddress(8080));
                ssc.configureBlocking(false);
                boss = Selector.open();
                SelectionKey ssckey = ssc.register(boss, 0, null);
                ssckey.interestOps(SelectionKey.OP_ACCEPT);
                workers = initEventLoops();
                new Thread(this, "boss").start();
                log.debug("boss start...");
                start = true;
            }
        }
    }
}
```

```

    }
}

public WorkerEventLoop[] initEventLoops() {
//    EventLoop[] eventLoops = new
EventLoop[Runtime.getRuntime().availableProcessors()];
    WorkerEventLoop[] workerEventLoops = new WorkerEventLoop[2];
    for (int i = 0; i < workerEventLoops.length; i++) {
        workerEventLoops[i] = new WorkerEventLoop(i);
    }
    return workerEventLoops;
}

@Override
public void run() {
    while (true) {
        try {
            boss.select();
            Iterator<SelectionKey> iter =
boss.selectedKeys().iterator();
            while (iter.hasNext()) {
                SelectionKey key = iter.next();
                iter.remove();
                if (key.isAcceptable()) {
                    ServerSocketChannel c = (ServerSocketChannel)
key.channel();

                    SocketChannel sc = c.accept();
                    sc.configureBlocking(false);
                    log.debug("{} connected", sc.getRemoteAddress());
                    workers[index.getAndIncrement() %
workers.length].register(sc);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

@Slf4j
static class WorkerEventLoop implements Runnable {
    private Selector worker;
    private volatile boolean start = false;
    private int index;

    private final ConcurrentLinkedQueue<Runnable> tasks = new
ConcurrentLinkedQueue<>();

    public WorkerEventLoop(int index) {
        this.index = index;
    }

    public void register(SocketChannel sc) throws IOException {
        if (!start) {

```

```

        worker = Selector.open();
        new Thread(this, "worker-" + index).start();
        start = true;
    }
    tasks.add(() -> {
        try {
            SelectionKey skey = sc.register(worker, 0, null);
            skey.interestOps(SelectionKey.OP_READ);
            worker.selectNow();
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
    worker.wakeup();
}

@Override
public void run() {
    while (true) {
        try {
            worker.select();
            Runnable task = tasks.poll();
            if (task != null) {
                task.run();
            }
            Set<SelectionKey> keys = worker.selectedKeys();
            Iterator<SelectionKey> iter = keys.iterator();
            while (iter.hasNext()) {
                SelectionKey key = iter.next();
                if (key.isReadable()) {
                    SocketChannel sc = (SocketChannel) key.channel();
                    ByteBuffer buffer = ByteBuffer.allocate(128);
                    try {
                        int read = sc.read(buffer);
                        if (read == -1) {
                            key.cancel();
                            sc.close();
                        } else {
                            buffer.flip();
                            log.debug("{} message:",
sc.getRemoteAddress());

                            debugAll(buffer);
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                        key.cancel();
                        sc.close();
                    }
                }
                iter.remove();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```
}  
}  
}
```

💡 如何拿到 cpu 个数

- `Runtime.getRuntime().availableProcessors()` 如果工作在 docker 容器下，因为容器不是物理隔离的，会拿到物理 cpu 个数，而不是容器申请时的个数
- 这个问题直到 jdk 10 才修复，使用 jvm 参数 `UseContainerSupport` 配置，默认开启

4.7 UDP

- UDP 是无连接的，client 发送数据不会管 server 是否开启
- server 这边的 `receive` 方法会将接收到的数据存入 byte buffer，但如果数据报文超过 buffer 大小，多出来的数据会被默默抛弃

首先启动服务器端

```
public class UdpServer {  
    public static void main(String[] args) {  
        try (DatagramChannel channel = DatagramChannel.open()) {  
            channel.socket().bind(new InetSocketAddress(9999));  
            System.out.println("waiting...");  
            ByteBuffer buffer = ByteBuffer.allocate(32);  
            channel.receive(buffer);  
            buffer.flip();  
            debug(buffer);  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

输出

```
waiting...
```

运行客户端

```

public class UdpClient {
    public static void main(String[] args) {
        try (DatagramChannel channel = DatagramChannel.open()) {
            ByteBuffer buffer = StandardCharsets.UTF_8.encode("hello");
            InetAddress address = new InetAddress("localhost",
9999);

            channel.send(buffer, address);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

接下来服务器端输出

```

+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+
|00000000| 68 65 6c 6c 6f                               |hello|
+-----+-----+

```

5. NIO vs BIO

5.1 stream vs channel

- stream 不会自动缓冲数据，channel 会利用系统提供的发送缓冲区、接收缓冲区（更为底层）
- stream 仅支持阻塞 API，channel 同时支持阻塞、非阻塞 API，网络 channel 可配合 selector 实现多路复用
- 二者均为全双工（stream也是可以同时读写的！），即读写可以同时进行

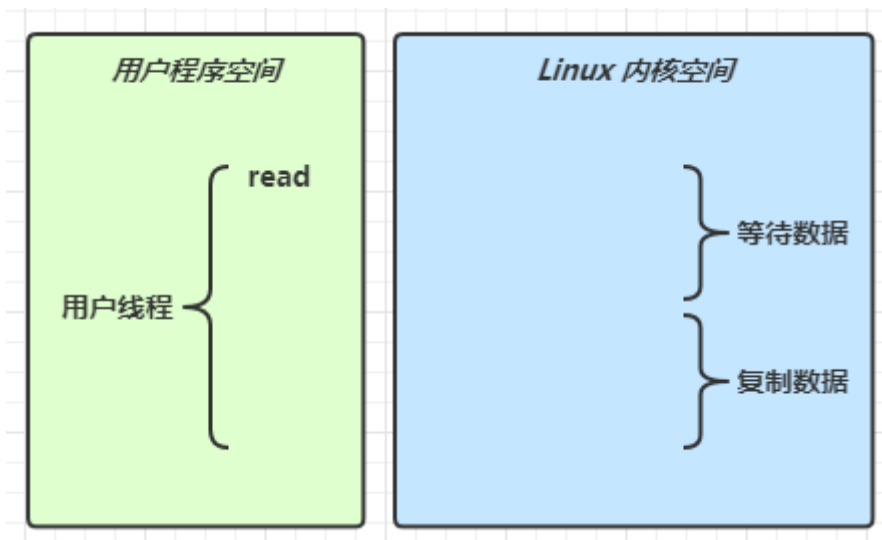
5.2 IO 模型

同步阻塞、同步非阻塞、同步多路复用、异步阻塞（没有此情况）、异步非阻塞

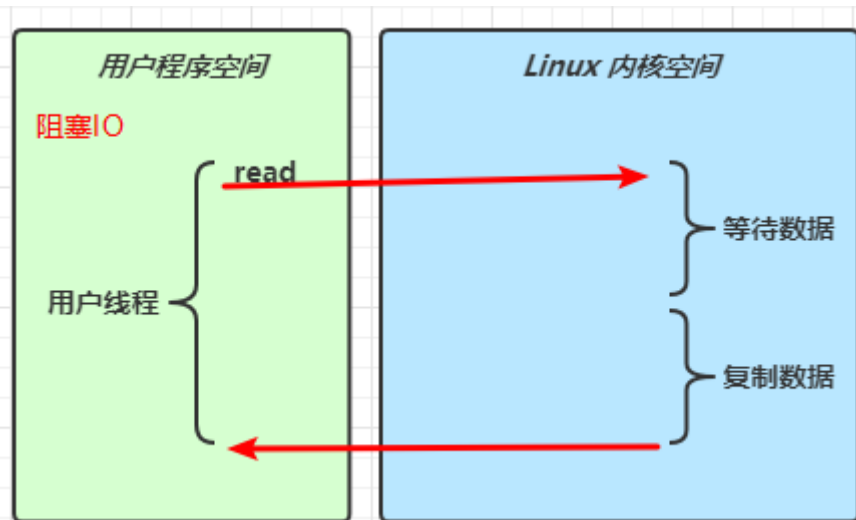
- 同步：线程自己去获取结果（一个线程）
- 异步：线程自己不去获取结果，而是由其它线程送结果（至少两个线程）

当调用一次 `channel.read` 或 `stream.read` 后，会切换至操作系统内核态来完成真正数据读取，而读取又分为两个阶段，分别为：

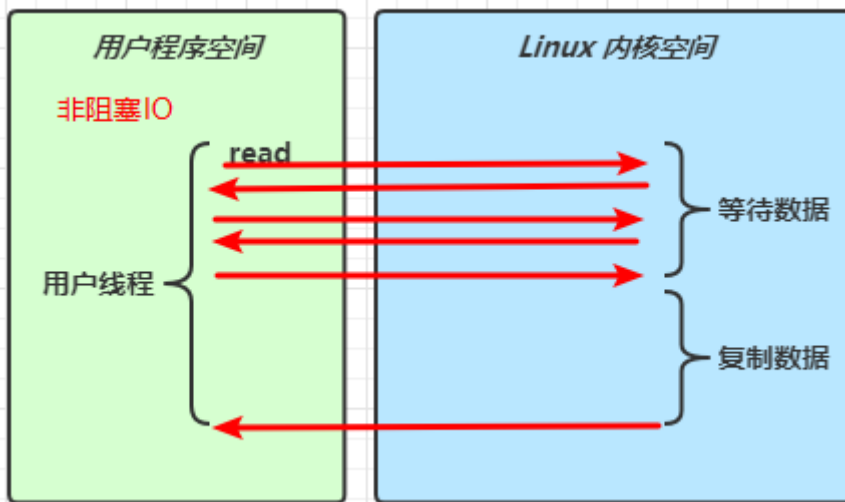
- 等待数据阶段
- 复制数据阶段



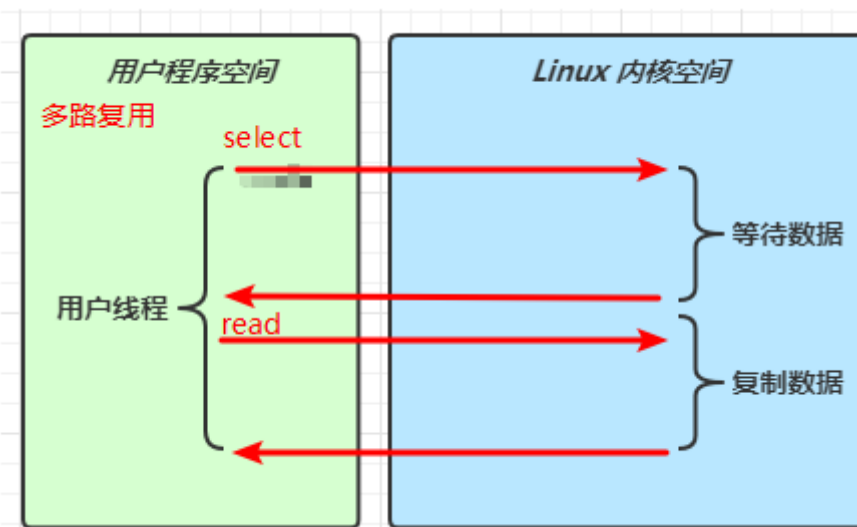
- 阻塞IO



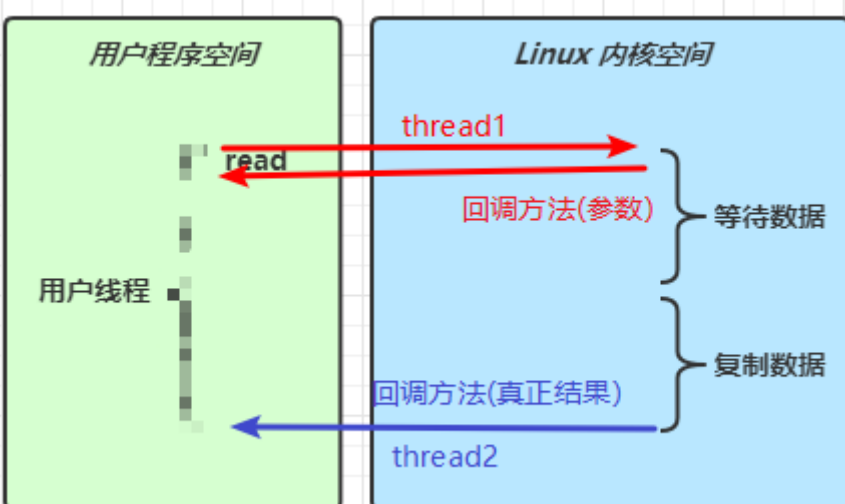
- 非阻塞 IO



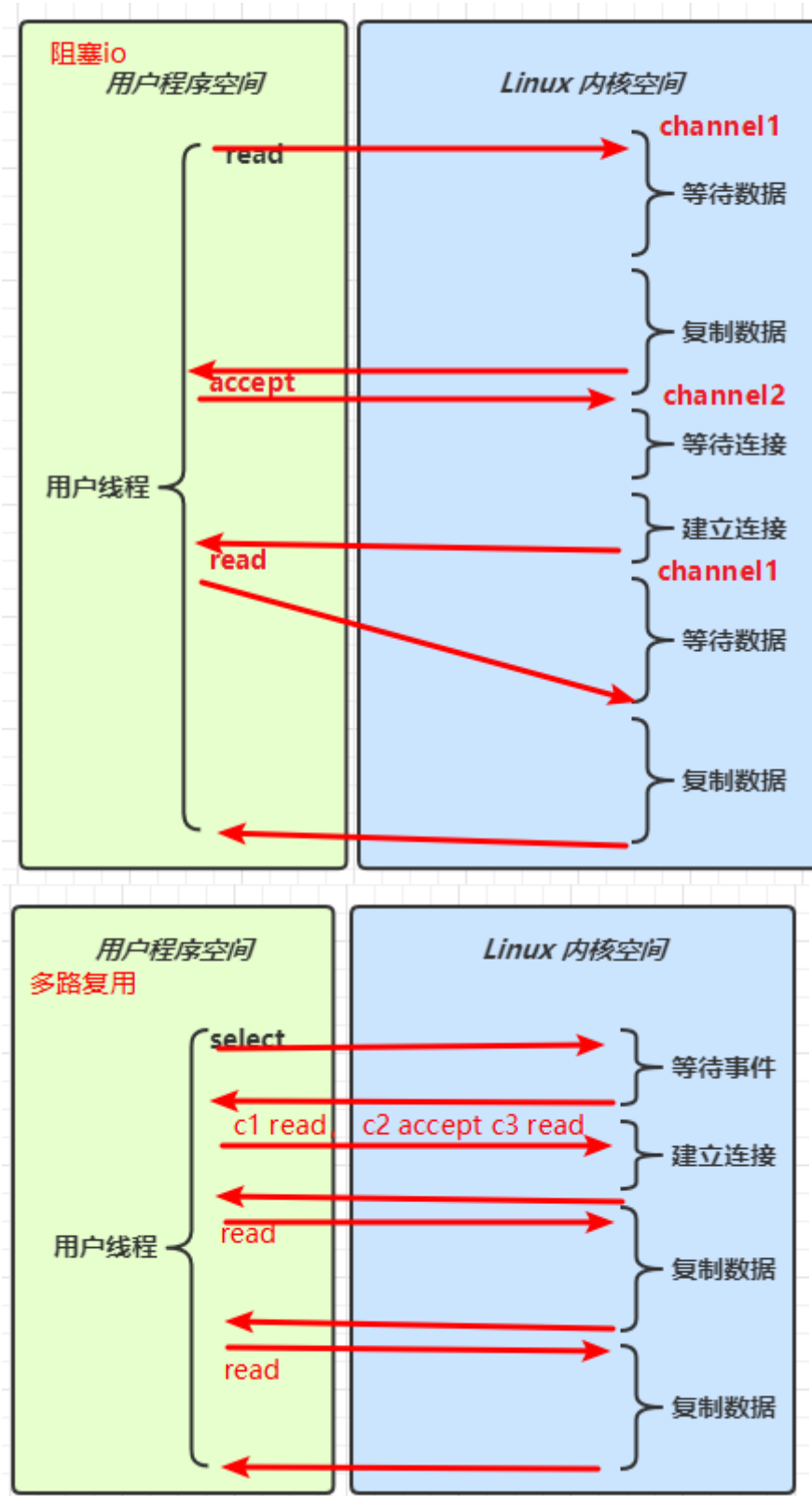
- 多路复用



- 信号驱动 (不经常用)
- 异步 IO



- 阻塞 IO vs 多路复用



🔗 参考

5.3 零拷贝

传统 IO 问题

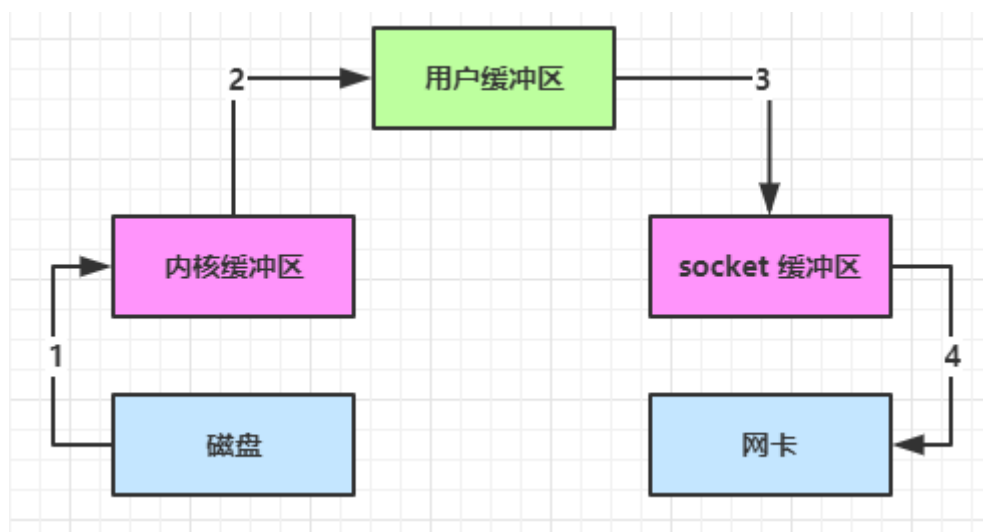
传统的 IO 将一个文件通过 socket 写出

```
File f = new File("helloworld/data.txt");
RandomAccessFile file = new RandomAccessFile(file, "r");

byte[] buf = new byte[(int)f.length()];
file.read(buf);

Socket socket = ...;
socket.getOutputStream().write(buf);
```

内部工作流程是这样的：



1. java 本身并不具备 IO 读写能力，因此 read 方法调用后，要从 java 程序的**用户态**切换至**内核态**，去调用操作系统（Kernel）的读能力，将数据读入**内核缓冲区**。这期间用户线程阻塞，操作系统使用 DMA（Direct Memory Access）来实现文件读，其间也不会使用 cpu

DMA 也可以理解为硬件单元，用来解放 cpu 完成文件 IO

1. 从**内核态**切换回**用户态**，将数据从**内核缓冲区**读入**用户缓冲区**（即 byte[] buf），这期间 cpu 会参与拷贝，无法利用 DMA
2. 调用 write 方法，这时将数据从**用户缓冲区**（byte[] buf）写入 **socket 缓冲区**，cpu 会参与拷贝
3. 接下来要向网卡写数据，这项能力 java 又不具备，因此又得从**用户态**切换至**内核态**，调用操作系统的写能力，使用 DMA 将 **socket 缓冲区**的数据写入网卡，不会使用 cpu

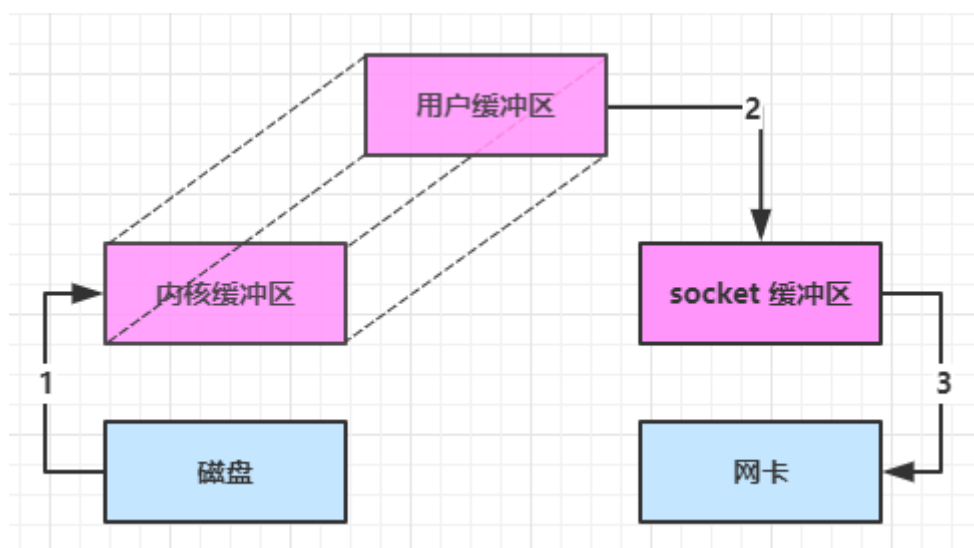
可以看到中间环节较多，java 的 IO 实际不是物理设备级别的读写，而是缓存的复制，底层的真正读写是操作系统来完成的

- 用户态与内核态的切换发生了 3 次，这个操作比较重量级
- 数据拷贝了共 4 次

NIO 优化

通过 DirectByteBuffer

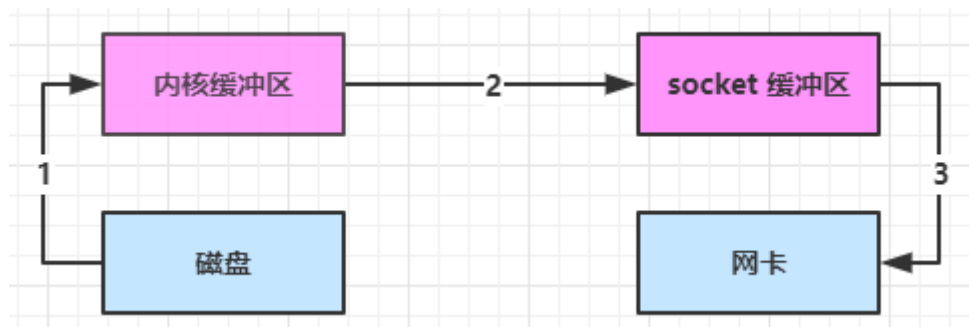
- ByteBuffer.allocate(10) HeapByteBuffer 使用的还是 java 内存
- ByteBuffer.allocateDirect(10) DirectByteBuffer 使用的是操作系统内存



大部分步骤与优化前相同，不再赘述。唯有一点：java 可以使用 DirectByteBuffer 将堆外内存映射到 jvm 内存中来直接访问使用

- 这块内存不受 jvm 垃圾回收的影响，因此内存地址固定，有助于 IO 读写
- java 中的 DirectByteBuffer 对象仅维护了此内存的虚引用，内存回收分成两步
 - DirectByteBuffer 对象被垃圾回收，将虚引用加入引用队列
 - 通过专门线程访问引用队列，根据虚引用释放堆外内存
- 减少了一次数据拷贝，用户态与内核态的切换次数没有减少

进一步优化（底层采用了 linux 2.1 后提供的 sendFile 方法），java 中对应着两个 channel 调用 transferTo/transferFrom 方法拷贝数据

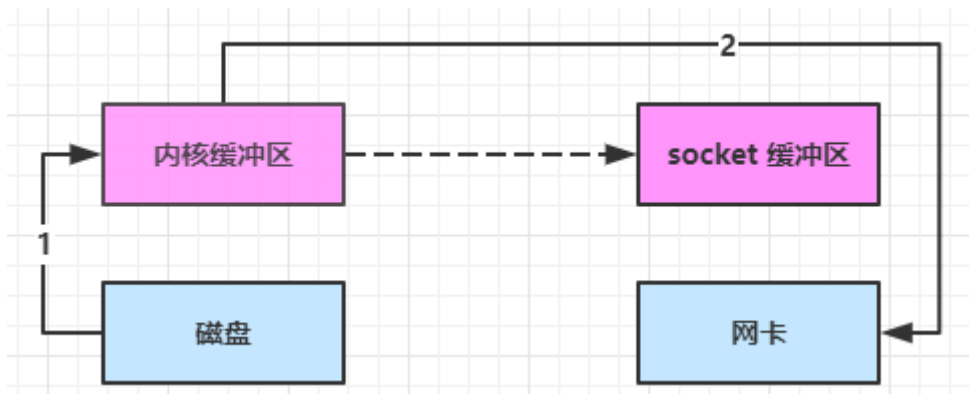


1. java 调用 transferTo 方法后，要从 java 程序的**用户态**切换至**内核态**，使用 DMA将数据读入**内核缓冲区**，不会使用 cpu
2. 数据从**内核缓冲区**传输到 **socket 缓冲区**，cpu 会参与拷贝
3. 最后使用 DMA 将 **socket 缓冲区**的数据写入网卡，不会使用 cpu

可以看到

- 只发生了一次用户态与内核态的切换
- 数据拷贝了 3 次

进一步优化 (linux 2.4)



1. java 调用 transferTo 方法后，要从 java 程序的**用户态**切换至**内核态**，使用 DMA将数据读入**内核缓冲区**，不会使用 cpu
2. 只会将一些 offset 和 length 信息拷入 **socket 缓冲区**，几乎无消耗
3. 使用 DMA 将 **内核缓冲区**的数据写入网卡，不会使用 cpu

整个过程仅只发生了一次用户态与内核态的切换，数据拷贝了 2 次。所谓的【零拷贝】，并不是真正无拷贝，而是在不会拷贝重复数据到 jvm 内存中，零拷贝的优点有

- 更少的用户态与内核态的切换
- 不利用 cpu 计算，减少 cpu 缓存伪共享
- 零拷贝适合小文件传输

5.3 AIO

AIO 用来解决数据复制阶段的阻塞问题

- 同步意味着，在进行读写操作时，线程需要等待结果，还是相当于闲置
- 异步意味着，在进行读写操作时，线程不必等待结果，而是将来由操作系统来通过回调方式由另外的线程来获得结果

异步模型需要底层操作系统（Kernel）提供支持

- Windows 系统通过 IOCP 实现了真正的异步 IO
- Linux 系统异步 IO 在 2.6 版本引入，但其底层实现还是用多路复用模拟了异步 IO，性能没有优势

文件 AIO

先来看看 AsynchronousFileChannel

```
@Slf4j
public class AioDemo1 {
    public static void main(String[] args) throws IOException {
        try{
            AsynchronousFileChannel s =
                AsynchronousFileChannel.open(
                    Paths.get("1.txt"), StandardOpenOption.READ);
            ByteBuffer buffer = ByteBuffer.allocate(2);
            log.debug("begin...");
            s.read(buffer, 0, null, new CompletionHandler<Integer, ByteBuffer>()
            {
                @Override
                public void completed(Integer result, ByteBuffer attachment) {
                    log.debug("read completed...{}", result);
                    buffer.flip();
                    debug(buffer);
                }

                @Override
                public void failed(Throwable exc, ByteBuffer attachment) {
                    log.debug("read failed...");
                }
            });

        } catch (IOException e) {
```

```

        e.printStackTrace();
    }
    log.debug("do other things...");
    System.in.read();
}
}

```

输出

```

13:44:56 [DEBUG] [main] c.i.aio.AioDemo1 - begin...
13:44:56 [DEBUG] [main] c.i.aio.AioDemo1 - do other things...
13:44:56 [DEBUG] [Thread-5] c.i.aio.AioDemo1 - read completed...2
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+-----+-----+-----+
|00000000| 61 0d                                     |a.          |
+-----+-----+-----+-----+-----+-----+-----+

```

可以看到

- 响应文件读取成功的是另一个线程 Thread-5
- 主线程并没有 IO 操作阻塞

💡 守护线程

默认文件 AIO 使用的线程都是守护线程，所以最后要执行 `System.in.read()` 以避免守护线程意外结束

网络 AIO

```

public class AioServer {
    public static void main(String[] args) throws IOException {
        AsynchronousServerSocketChannel ssc =
        AsynchronousServerSocketChannel.open();
        ssc.bind(new InetSocketAddress(8080));
        ssc.accept(null, new AcceptorHandler(ssc));
        System.in.read();
    }

    private static void closeChannel(AsynchronousSocketChannel sc) {
        try {

```

```

        System.out.printf("[%s] %s close\n",
Thread.currentThread().getName(), sc.getRemoteAddress());
        sc.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private static class ReadHandler implements CompletionHandler<Integer,
ByteBuffer> {
    private final AsynchronousSocketChannel sc;

    public ReadHandler(AsynchronousSocketChannel sc) {
        this.sc = sc;
    }

    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        try {
            if (result == -1) {
                closeChannel(sc);
                return;
            }
            System.out.printf("[%s] %s read\n",
Thread.currentThread().getName(), sc.getRemoteAddress());
            attachment.flip();
            System.out.println(Charset.defaultCharset().decode(attachment));
            attachment.clear();
            // 处理完第一个 read 时, 需要再次调用 read 方法来处理下一个 read 事件
            sc.read(attachment, attachment, this);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {
        closeChannel(sc);
        exc.printStackTrace();
    }
}

private static class WriteHandler implements CompletionHandler<Integer,
ByteBuffer> {
    private final AsynchronousSocketChannel sc;

    private WriteHandler(AsynchronousSocketChannel sc) {
        this.sc = sc;
    }

    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        // 如果作为附件的 buffer 还有内容, 需要再次 write 写出剩余内容
        if (attachment.hasRemaining()) {
            sc.write(attachment);
        }
    }
}

```

```

    }
}

@Override
public void failed(Throwable exc, ByteBuffer attachment) {
    exc.printStackTrace();
    closeChannel(sc);
}
}

private static class AcceptHandler implements
CompletionHandler<AsynchronousSocketChannel, Object> {
    private final AsynchronousServerSocketChannel ssc;

    public AcceptHandler(AsynchronousServerSocketChannel ssc) {
        this.ssc = ssc;
    }

    @Override
    public void completed(AsynchronousSocketChannel sc, Object attachment) {
        try {
            System.out.printf("[%s] %s connected\n",
Thread.currentThread().getName(), sc.getRemoteAddress());
        } catch (IOException e) {
            e.printStackTrace();
        }
        ByteBuffer buffer = ByteBuffer.allocate(16);
        // 读事件由 ReadHandler 处理
        sc.read(buffer, buffer, new ReadHandler(sc));
        // 写事件由 WriteHandler 处理
        sc.write(Charset.defaultCharset().encode("server hello!"),
ByteBuffer.allocate(16), new WriteHandler(sc));
        // 处理完第一个 accept 时, 需要再次调用 accept 方法来处理下一个 accept 事件
        ssc.accept(null, this);
    }

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

```

二. Netty 入门

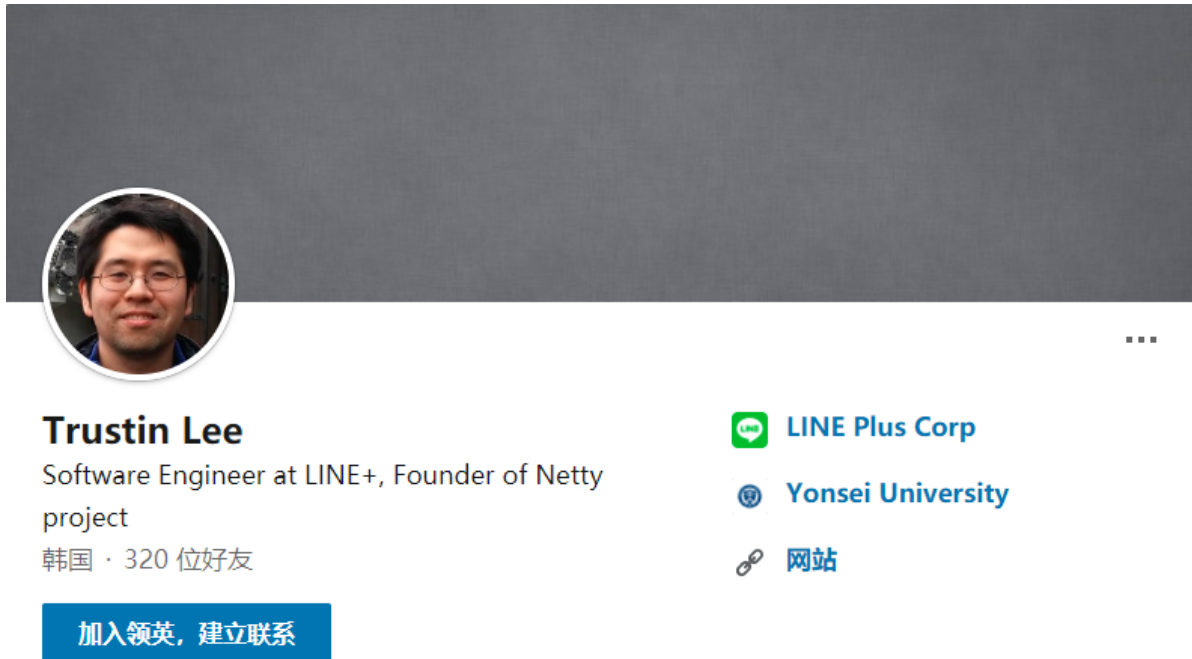
1. 概述

1.1 Netty 是什么?

Netty is an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients.

翻译：Netty 是一个异步的（利用多线程）、基于事件驱动的网络应用框架，用于快速开发可维护、高性能的网络服务器和客户端

1.2 Netty 的作者



他还是另一个著名网络应用框架 Mina 的重要贡献者

1.3 Netty 的地位

Netty 在 Java 网络应用框架中的地位就好比：Spring 框架在 JavaEE 开发中的地位

以下的框架都使用了 Netty，因为它们有网络通信需求！

- Cassandra - nosql 数据库
- Spark - 大数据分布式计算框架
- Hadoop - 大数据分布式存储框架
- RocketMQ - ali 开源的消息队列
- ElasticSearch - 搜索引擎
- gRPC - rpc 框架
- Dubbo - rpc 框架
- Spring 5.x - flux api 完全抛弃了 tomcat，使用 netty 作为服务器端
- Zookeeper - 分布式协调框架

1.4 Netty 的优势

- Netty vs NIO, 工作量大, bug 多
- - 需要自己构建协议
 - 解决 TCP 传输问题, 如粘包、半包
 - epoll 空轮询导致 CPU 100%
 - 对 API 进行增强, 使之更易用, 如 FastThreadLocal => ThreadLocal, ByteBuffer => ByteBuf
- Netty vs 其它网络应用框架
- - Mina 由 apache 维护, 将来 3.x 版本可能会有较大重构, 破坏 API 向下兼容性, Netty 的开发迭代更迅速, API 更简洁、文档更优秀
 - 久经考验, 16年, Netty 版本
 - 2.x 2004
 - 3.x 2008
 - 4.x 2013
 - 5.x 已废弃 (没有明显的性能提升, 维护成本高)

2. Hello World

2.1 目标

开发一个简单的服务器端和客户端

- 客户端向服务器端发送 hello, world
- 服务器仅接收, 不返回

加入依赖

```
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-all</artifactId>
  <version>4.1.39.Final</version>
</dependency>
```

2.2 服务器端

```
package netty.c1;

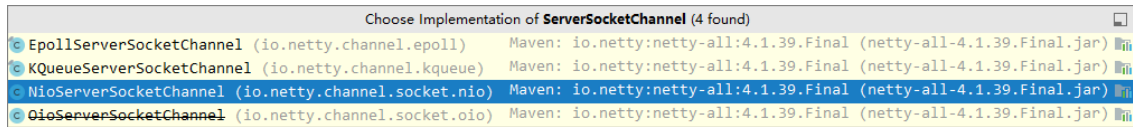
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringDecoder;

public class HelloServer {
    public static void main(String[] args) {
        new ServerBootstrap()
            // 1 处, 创建 NioEventLoopGroup, 可以简单理解为 线程池 + selector 后面
            // 会详细展开
            .group(new NioEventLoopGroup())
            // 2 处, 选择服务 Socket 实现类, 其中 NioServerSocketChannel 表示基于
            // NIO 的服务器端实现, 其它实现还有
            .channel(NioServerSocketChannel.class)
            // 3 处, 为啥方法叫 childHandler, 是接下来添加的处理器都是给
            // SocketChannel 用的, 而不是给 ServersocketChannel。
            // ChannelInitializer 处理器 (仅执行一次), 它的作用是待客户端
            // SocketChannel 建立连接后, 执行 initChannel 以便添加更多的处理器
            .childHandler(new ChannelInitializer<NioSocketChannel>() {
                protected void initChannel(NioSocketChannel ch) {
                    // 5 处, SocketChannel 的处理器, 解码 ByteBuf => String
                    ch.pipeline().addLast(new StringDecoder());
                    // 6 处, SocketChannel 的业务处理器, 使用上一个处理器的处理结果
                    ch.pipeline().addLast(new
SimpleChannelInboundHandler<String>() {
                        @Override
                        protected void channelRead0(ChannelHandlerContext
ctx, String msg) {
                            System.out.println(msg);
                        }
                    });
                }
            })
            //4 处, ServerSocketChannel 绑定的监听端口
            .bind(8080);
    }
}
```

代码解读

- 1 处, 创建 NioEventLoopGroup, 可以简单理解为 线程池 + selector 后面会详细展开

- 2 处，选择服务 Scoket 实现类，其中 NioServerSocketChannel 表示基于 NIO 的服务器端实现，其它实现还有



- 3 处，为啥方法叫 childHandler，是接下来添加的处理器都是给 SocketChannel 用的，而不是给 ServerSocketChannel。ChannelInitializer 处理器（仅执行一次），它的作用是待客户端 SocketChannel 建立连接后，执行 initChannel 以便添加更多的处理器
- 4 处，ServerSocketChannel 绑定的监听端口
- 5 处，SocketChannel 的处理器，解码 ByteBuf => String
- 6 处，SocketChannel 的业务处理器，使用上一个处理器的处理结果

2.3 客户端

```
package netty.c1;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.Channel;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringEncoder;

import java.util.Date;

public class HelloClient {
    public static void main(String[] args) throws InterruptedException {
        new Bootstrap()
            //1 处，创建 NioEventLoopGroup，同 Server
            .group(new NioEventLoopGroup()) // 1
            // 2 处，选择客户 Socket 实现类，NioSocketChannel 表示基于 NIO 的客户端
            // 实现，其它实现还有
            .channel(NioSocketChannel.class) // 2
            //3 处，添加 SocketChannel 的处理器，ChannelInitializer 处理器（仅执行
            // 一次），
            // 它的作用是待客户端 SocketChannel 建立连接后，执行 initChannel 以便添
            // 加更多的处理器
            .handler(new ChannelInitializer<Channel>() { // 3
                @Override
                protected void initChannel(Channel ch) {
                    // 8 处，消息会经过通道 handler 处理，这里是将 String =>
                    ByteBuf 发出
                    ch.pipeline().addLast(new StringEncoder()); // 8
                }
            })
            //4 处，指定要连接的服务器和端口
            .connect("127.0.0.1", 8080) // 4
            //5 处，Netty 中很多方法都是异步的，如 connect，这时需要使用 sync 方法等
            // 待 connect 建立连接完毕
            //这个是一个阻塞方法，直到连接建立
    }
}
```

```

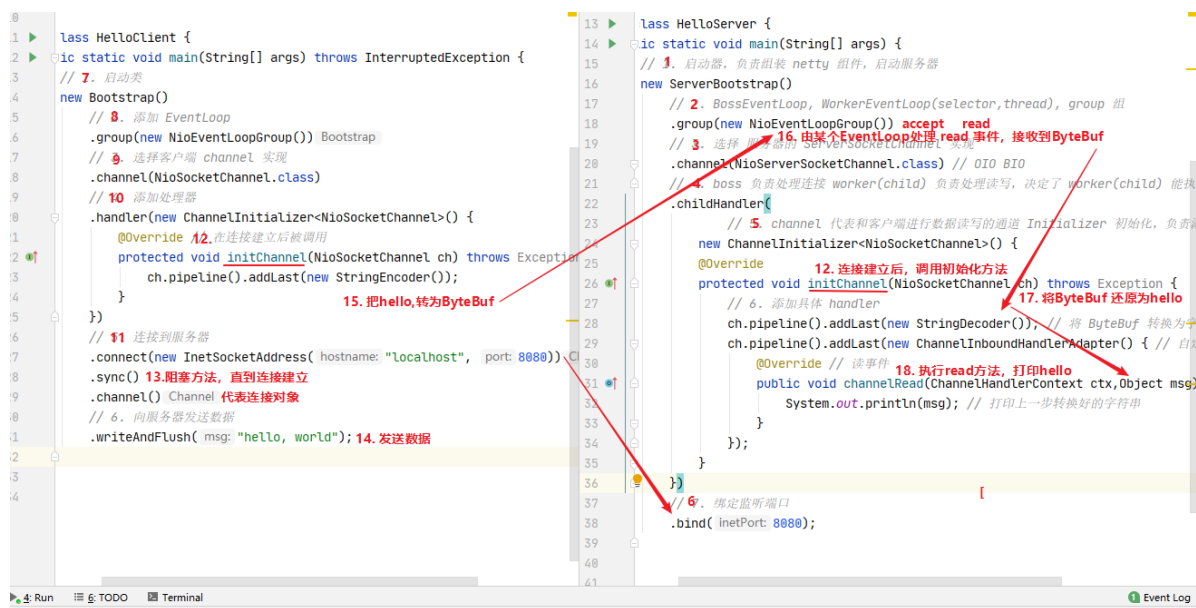
        .sync() // 5
        //6 处, 获取 channel 对象, 它即为通道抽象, 可以进行数据读写操作
        .channel() // 6
        //7 处, 写入消息并清空缓冲区
        .writeAndFlush(new Date() + ": hello world!"); // 7
    }
}

```

代码解读

- 1 处, 创建 NioEventLoopGroup, 同 Server
- 2 处, 选择客户 Socket 实现类, NioSocketChannel 表示基于 NIO 的客户端实现, 其它实现还有
- 3 处, 添加 SocketChannel 的处理器, ChannelInitializer 处理器 (仅执行一次), 它的作用是待客户端 SocketChannel 建立连接后, 执行 initChannel 以便添加更多的处理器
- 4 处, 指定要连接的服务器和端口
- 5 处, Netty 中很多方法都是异步的, 如 connect, 这时需要使用 sync 方法等待 connect 建立连接完毕
- 6 处, 获取 channel 对象, 它即为通道抽象, 可以进行数据读写操作
- 7 处, 写入消息并清空缓冲区
- 8 处, 消息会经过通道 handler 处理, 这里是将 String => ByteBuf 发出
- 数据经过网络传输, 到达服务器端, 服务器端 5 和 6 处的 handler 先后被触发, 走完一个流程

2.4 流程梳理



💡 提示

一开始需要树立正确的观念

- 把 channel 理解为数据的通道
- 把 msg 理解为流动的数据，最开始输入是 ByteBuf，但经过 pipeline 的加工，会变成其它类型对象，最后输出又变成 ByteBuf
- 把 handler 理解为数据的处理工序
 - 工序有多道，合在一起就是 pipeline，pipeline 负责发布事件（读、读取完成...）传播给每个 handler，handler 对自己感兴趣的事件进行处理（重写了相应事件处理方法）
 - handler 分 Inbound 和 Outbound 两类
- 把 eventLoop 理解为处理数据的工人
 - 工人可以管理多个 channel 的 io 操作，并且一旦工人负责了某个 channel，就要负责到底（绑定）
 - 工人既可以执行 io 操作，也可以进行任务处理，每位工人有任务队列，队列里可以堆放多个 channel 的待处理任务，任务分为普通任务、定时任务
 - 工人按照 pipeline 顺序，依次按照 handler 的规划（代码）处理数据，可以为每道工序指定不同的工人

3. 组件

3.1 EventLoop

事件循环对象

EventLoop 本质是一个单线程执行器（同时维护了一个 Selector），里面有 run 方法处理 Channel 上源源不断的 io 事件。

它的继承关系比较复杂

- 一条线是继承自 `j.u.c.ScheduledExecutorService` 因此包含了线程池中所有的方法
- 另一条线是继承自 netty 自己的 `OrderedEventExecutor`，
 - 提供了 `boolean inEventLoop(Thread thread)` 方法判断一个线程是否属于此 EventLoop
 - 提供了 `parent` 方法来看看自己属于哪个 EventLoopGroup

事件循环组

EventLoopGroup 是一组 EventLoop，Channel 一般会调用 EventLoopGroup 的 register 方法来绑定其中一个 EventLoop，后续这个 Channel 上的 io 事件都由此 EventLoop 来处理（保证了 io 事件处理时的线程安全）

- 继承自 netty 自己的 EventExecutorGroup
- - 实现了 Iterable 接口提供遍历 EventLoop 的能力
 - 另有 next 方法获取集合中下一个 EventLoop

以一个简单的实现为例：

```
// 内部创建了两个 EventLoop，每个 EventLoop 维护一个线程
DefaultEventLoopGroup group = new DefaultEventLoopGroup(2);
System.out.println(group.next());
System.out.println(group.next());
System.out.println(group.next());
```

输出

```
io.netty.channel.DefaultEventLoop@60f82f98
io.netty.channel.DefaultEventLoop@35f983a6
io.netty.channel.DefaultEventLoop@60f82f98
```

也可以使用 for 循环

```
DefaultEventLoopGroup group = new DefaultEventLoopGroup(2);
for (EventExecutor eventLoop : group) {
    System.out.println(eventLoop);
}
```

输出

```
io.netty.channel.DefaultEventLoop@60f82f98
io.netty.channel.DefaultEventLoop@35f983a6
```

💡 优雅关闭

优雅关闭 `shutdownGracefully` 方法。该方法会首先切换 `EventLoopGroup` 到关闭状态从而拒绝新的任务的加入，然后在任务队列的任务都处理完成后，停止线程的运行。从而确保整体应用是在正常有序的状态下退出的

演示 NioEventLoop 处理 io 事件

服务器端两个 nio worker 工人

```
new ServerBootstrap()
    .group(new NioEventLoopGroup(1), new NioEventLoopGroup(2))
    .channel(NioServerSocketChannel.class)
    .childHandler(new ChannelInitializer<NioSocketChannel>() {
        @Override
        protected void initChannel(NioSocketChannel ch) {
            ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
                @Override
                public void channelRead(ChannelHandlerContext ctx, Object msg) {
                    ByteBuf byteBuf = msg instanceof ByteBuf ? ((ByteBuf) msg) :
null;

                    if (byteBuf != null) {
                        byte[] buf = new byte[16];
                        ByteBuf len = byteBuf.readBytes(buf, 0,
byteBuf.readableBytes());
                        log.debug(new String(buf));
                    }
                }
            });
        }
    }).bind(8080).sync();
```

客户端，启动三次，分别修改发送字符串为 zhangsan（第一次），lisi（第二次），wangwu（第三次）

```
public static void main(String[] args) throws InterruptedException {
    Channel channel = new Bootstrap()
        .group(new NioEventLoopGroup(1))
        .handler(new ChannelInitializer<NioSocketChannel>() {
            @Override
            protected void initChannel(NioSocketChannel ch) throws Exception
{
                system.out.println("init...");
            }
        }).connect("localhost", 8080).sync().channel();
}
```



```
        ch.pipeline().addLast(new LoggingHandler(LogLevel.DEBUG));
    }
})
.channel(NioSocketChannel.class).connect("localhost", 8080)
.sync()
.channel();

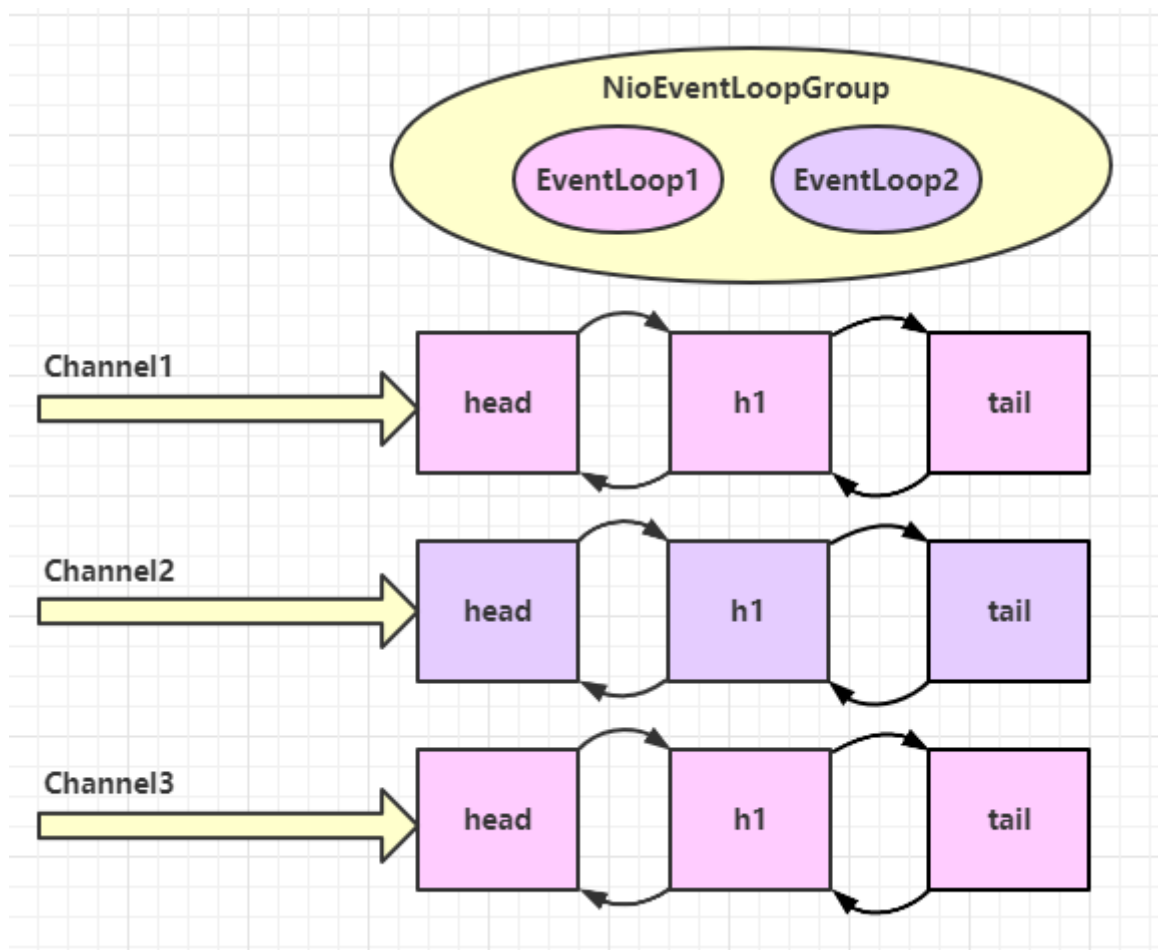
channel.writeAndFlush(ByteBufAllocator.DEFAULT.buffer().writeBytes("wangwu".getBytes()));
Thread.sleep(2000);

channel.writeAndFlush(ByteBufAllocator.DEFAULT.buffer().writeBytes("wangwu".getBytes()));
```

最后输出

```
22:03:34 [DEBUG] [nioEventLoopGroup-3-1] c.i.o.EventLoopTest - zhangsan
22:03:36 [DEBUG] [nioEventLoopGroup-3-1] c.i.o.EventLoopTest - zhangsan
22:05:36 [DEBUG] [nioEventLoopGroup-3-2] c.i.o.EventLoopTest - lisi
22:05:38 [DEBUG] [nioEventLoopGroup-3-2] c.i.o.EventLoopTest - lisi
22:06:09 [DEBUG] [nioEventLoopGroup-3-1] c.i.o.EventLoopTest - wangwu
22:06:11 [DEBUG] [nioEventLoopGroup-3-1] c.i.o.EventLoopTest - wangwu
```

可以看到两个工人轮流处理 channel，但工人与 channel 之间进行了绑定



再增加两个非 nio 工人

```
package netty.c1;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.buffer.ByteBuf;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import lombok.extern.slf4j.Slf4j;

import java.nio.ByteBuffer;
import java.nio.charset.Charset;

@Slf4j
public class HelloServer {
    public static void main(String[] args) {
        DefaultEventLoopGroup group = new DefaultEventLoopGroup();
        new ServerBootstrap()
            .group(new NioEventLoopGroup())
            .channel(NioServerSocketChannel.class)
            .childHandler(new ChannelInitializer<NioSocketChannel>() {
                protected void initChannel(NioSocketChannel ch) {
```

```

        ch.pipeline().addLast("handler1", new
ChannelInboundHandlerAdapter() {
            @Override
            public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
                ByteBuf byteBuf= (ByteBuf) msg;

                log.debug(byteBuf.toString(Charset.defaultCharset()));
                ctx.fireChannelRead(msg); //递交给下一个拦截器
            }
        }).addLast(group, "handler2", new
ChannelInboundHandlerAdapter() {
            @Override
            public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
                ByteBuf byteBuf= (ByteBuf) msg;

                log.debug(byteBuf.toString(Charset.defaultCharset()));
            }
        });
    }

    })
    .bind(8080);
}
}

```

客户端代码不变，启动三次，分别修改发送字符串为 zhangsan（第一次），lisi（第二次），wangwu（第三次）

输出

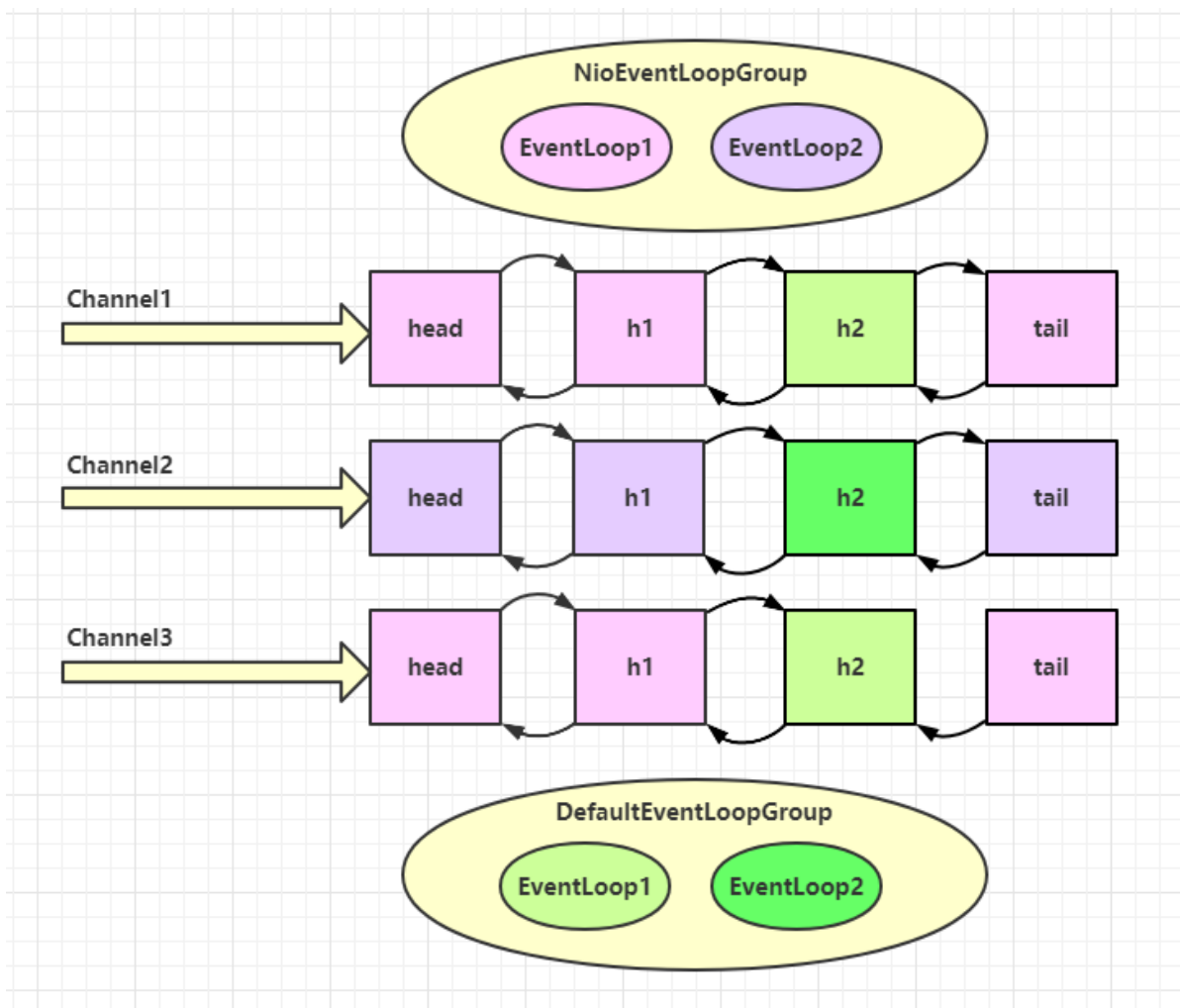
The screenshot shows an IDE with three main components:

- Code Editor:** Displays the server-side code. The line `ch.pipeline().addLast(new StringEncoder()); // 8` is highlighted. Comments explain the connection process and the role of the `channel` object.
- Log Window:** Shows two log entries from `netty.c1.HelloServer`:


```

21:39:14.179 [nioEventLoopGroup-3-3] DEBUG netty.c1.HelloServer - hahah
21:39:14.180 [defaultEventLoopGroup-2-2] DEBUG netty.c1.HelloServer - hahah
      
```
- Evaluate Window:** Shows the result of evaluating the expression `channel.writeAndFlush("hahah")`. The result is a `DefaultChannelPromise` object with details:
 - `channel`: `NioSocketChannel@1415`
 - `checkpoint`: `0`
 - `result`: `{Object@1657}`
 - `executor`: `{NioEventLoop@1636}`
 - `listeners`: `null`
 - `waiters`: `0`
 - `notifyingListeners`: `false`

可以看到，nio 工人和 非 nio 工人也分别绑定了 channel（LoggingHandler 由 nio 工人执行，而我们自己的 handler 由非 nio 工人执行）



💡 handler 执行中如何换人?

下面是源码

关键代码 `io.netty.channel.AbstractChannelHandlerContext#invokeChannelRead()`

```
static void invokeChannelRead(final AbstractChannelHandlerContext next, Object msg) {
    final Object m = next.pipeline.touch(ObjectUtil.checkNotNull(msg, "msg"), next);
    //这个返回的是下一个handler的eventLoop, executor就是eventLoop
    EventExecutor executor = next.executor();

    // 是, 直接调用
    if (executor.inEventLoop()) { //判断当前handler中的线程, 是否跟eventLoop是同一个线程
        next.invokeChannelRead(m);
    }
    // 不是, 将要执行的代码作为任务提交给下一个事件循环处理 (换人)
    else {
        executor.execute(new Runnable() {
            @Override
            public void run() {
                next.invokeChannelRead(m);
            }
        });
    }
}
```

```
    });  
  }  
}
```

- 如果两个 handler 绑定的是同一个线程，那么就直接调用
- 否则，把要调用的代码封装为一个任务对象，由下一个 handler 的线程来调用

演示 NioEventLoop 处理普通任务

NioEventLoop 除了可以处理 io 事件，同样可以向它提交普通任务

```
NioEventLoopGroup nioworkers = new NioEventLoopGroup(2);  
  
log.debug("server start...");  
Thread.sleep(2000);  
nioworkers.execute(()->{  
    log.debug("normal task...");  
});
```

输出

```
22:30:36 [DEBUG] [main] c.i.o.EventLoopTest2 - server start...  
22:30:38 [DEBUG] [nioEventLoopGroup-2-1] c.i.o.EventLoopTest2 - normal task...
```

可以用来执行耗时较长的任务

演示 NioEventLoop 处理定时任务

```
NioEventLoopGroup nioworkers = new NioEventLoopGroup(2);  
  
log.debug("server start...");  
Thread.sleep(2000);  
nioworkers.scheduleAtFixedRate(() -> {  
    log.debug("running...");  
}, 0, 1, TimeUnit.SECONDS);
```

输出

```
22:35:15 [DEBUG] [main] c.i.o.EventLoopTest2 - server start...
22:35:17 [DEBUG] [nioEventLoopGroup-2-1] c.i.o.EventLoopTest2 - running...
22:35:18 [DEBUG] [nioEventLoopGroup-2-1] c.i.o.EventLoopTest2 - running...
22:35:19 [DEBUG] [nioEventLoopGroup-2-1] c.i.o.EventLoopTest2 - running...
22:35:20 [DEBUG] [nioEventLoopGroup-2-1] c.i.o.EventLoopTest2 - running...
...
```

可以用来执行定时任务

3.2 Channel

channel 的主要作用

- close() 可以用来关闭 channel
- closeFuture() 用来处理 channel 的关闭
 - sync 方法作用是同步等待 channel 关闭
 - 而 addListener 方法是异步等待 channel 关闭
- pipeline() 方法添加处理器
- write() 方法将数据写入（但是不会立即发出，因为netty的channel是有一个缓冲区的，满足条件才会发）
- writeAndFlush() 方法将数据写入并刷出

ChannelFuture

```
package netty.c3;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.Channel;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringEncoder;

import java.util.Date;

public class HelloClient {
    public static void main(String[] args) throws InterruptedException {

        ChannelFuture channelFuture = new Bootstrap()
            .group(new NioEventLoopGroup())
```

```

        .channel(NioSocketChannel.class)
        .handler(new ChannelInitializer<Channel>() {
            @Override
            protected void initChannel(Channel ch) { //在连接后被调用
                ch.pipeline().addLast(new StringEncoder());
            }
        })
        // 这个connect方法是异步非阻塞的，main线程只发起了调用，真正执行的并不是这个线程

        // 连接是很慢的，如果不加阻塞就一直往下跑，信息可能就丢啦
        }).connect("127.0.0.1", 8080);
    channelFuture.sync(); //不加sync是发不过去的
    //可能是有这个对象的，但是不一定要有这个连接
    Channel channel = channelFuture.channel();
    channel.writeAndFlush("hello");
    System.out.println();
}
}

```

- ChannelFuture 对象，它的作用是利用 channel() 方法来获取 Channel 对象

注意 connect 方法是异步的，意味着不等连接建立，方法执行就返回了。因此 channelFuture 对象中不能【立刻】获得到正确的 Channel 对象

实验如下：

```

ChannelFuture channelFuture = new Bootstrap()
    .group(new NioEventLoopGroup())
    .channel(NioSocketChannel.class)
    .handler(new ChannelInitializer<Channel>() {
        @Override
        protected void initChannel(Channel ch) {
            ch.pipeline().addLast(new StringEncoder());
        }
    })
    .connect("127.0.0.1", 8080);

System.out.println(channelFuture.channel()); // 1
channelFuture.sync(); // 2
System.out.println(channelFuture.channel()); // 3

```

- 执行到 1 时，连接未建立，打印 [id: 0x2e1884dd]
- 执行到 2 时，sync 方法是同步等待连接建立完成
- 执行到 3 时，连接肯定建立了，打印 [id: 0x2e1884dd, L:/127.0.0.1:57191 - R:/127.0.0.1:8080]

除了用 sync 方法可以让异步操作同步以外，还可以使用回调的方式：

```
package netty.c3;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.Channel;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringEncoder;

import java.util.Date;

public class HelloClient {
    public static void main(String[] args) throws InterruptedException {

        ChannelFuture channelFuture = new Bootstrap()
            .group(new NioEventLoopGroup())
            .channel(NioSocketChannel.class)
            .handler(new ChannelInitializer<Channel>() {
                @Override
                protected void initChannel(Channel ch) { //在连接后被调用
                    ch.pipeline().addLast(new StringEncoder());
                }
            })
            // 这个connect方法是异步非阻塞的，main线程只发起了调用，真正执行的并不是这个线程
            // 连接是很慢的，如果不加阻塞就一直往下跑，信息可能就丢啦
            .connect("127.0.0.1", 8080);

        channelFuture.addListener(new ChannelFutureListener() {
            //在连接建立好的时候，会调用这个方法
            //但是要注意，这里面的方法也不是用的主线程来执行的，是用的连接的那个线程来执行的
            @Override
            public void operationComplete(ChannelFuture channelFuture) throws
Exception {
                Channel channel = channelFuture.channel();
                channel.writeAndFlush("hello");
            }
        });
    }
}
```

- 执行到 1 时，连接未建立，打印 [id: 0x749124ba]
- ChannelFutureListener 会在连接建立时被调用（其中 operationComplete 方法），因此执行到 2 时，连接肯定建立了，打印 [id: 0x749124ba, L:/127.0.0.1:57351 - R:/127.0.0.1:8080]

CloseFuture

这个案例是：满足客户端一直输入字符串，然后服务端进行打印，如果客户端输入了q，就退出，并且客户端需要做一些善后工作

同步

```
package netty.c3;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.Channel;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringEncoder;
import io.netty.handler.logging.LogLevel;
import io.netty.handler.logging.LoggingHandler;
import lombok.extern.slf4j.Slf4j;

import java.net.InetSocketAddress;
import java.util.Scanner;

@Slf4j
public class CloseFutureClient {
    public static void main(String[] args) throws InterruptedException {
        NioEventLoopGroup group = new NioEventLoopGroup();
        ChannelFuture channelFuture = new Bootstrap()
            .group(group)
            .channel(NioSocketChannel.class)
            .handler(new ChannelInitializer<NioSocketChannel>() {
                @Override
                protected void initChannel(NioSocketChannel ch) throws
Exception { // 在连接建立后被调用
                    ch.pipeline().addLast(new StringEncoder());
                }
            })
            .connect(new InetSocketAddress("localhost", 8080));
        //得到channel,但是连接建立完, sync就不会再阻塞了
        Channel channel = channelFuture.sync().channel();
        new Thread(new Runnable() {
            @Override
            public void run() {
                Scanner scanner = new Scanner(System.in);
                while (true) {
                    String line = scanner.nextLine();
                    if ("q".equals(line)) {
                        channel.close(); // close也是一个异步操作
                        //所以善后的工作不能写在这里
                        break;
                    }
                    channel.writeAndFlush(line);
                }
            }
        })
    }
}
```

```

    }, "input").start();

    // 获取 closeFuture 对象, 1) 同步处理关闭, 2) 异步处理关闭
    ChannelFuture closeFuture = channel.closeFuture();
    //1.同步
    closeFuture.sync();
    log.debug("关闭之后的善后工作");
}
}

```

异步:

```

package netty.c3;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.Channel;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringEncoder;
import io.netty.handler.logging.LogLevel;
import io.netty.handler.logging.LoggingHandler;
import lombok.extern.slf4j.Slf4j;

import java.net.InetSocketAddress;
import java.util.Scanner;

@Slf4j
public class CloseFutureClient {
    public static void main(String[] args) throws InterruptedException {
        NioEventLoopGroup group = new NioEventLoopGroup();
        ChannelFuture channelFuture = new Bootstrap()
            .group(group)
            .channel(NioSocketChannel.class)
            .handler(new ChannelInitializer<NioSocketChannel>() {
                @Override
                protected void initChannel(NioSocketChannel ch) throws
Exception { // 在连接建立后被调用
                    ch.pipeline().addLast(new StringEncoder());
                }
            })
            .connect(new InetSocketAddress("localhost", 8080));
        //得到channel,但是连接建立完, sync就不会再阻塞了
        Channel channel = channelFuture.sync().channel();
        new Thread(new Runnable() {
            @Override
            public void run() {
                Scanner scanner = new Scanner(System.in);
                while (true) {
                    String line = scanner.nextLine();
                    if ("q".equals(line)) {
                        channel.close(); // close也是一个异步操作
                        //所以善后的工作不能写在这里
                    }
                }
            }
        }).start();
    }
}

```

```

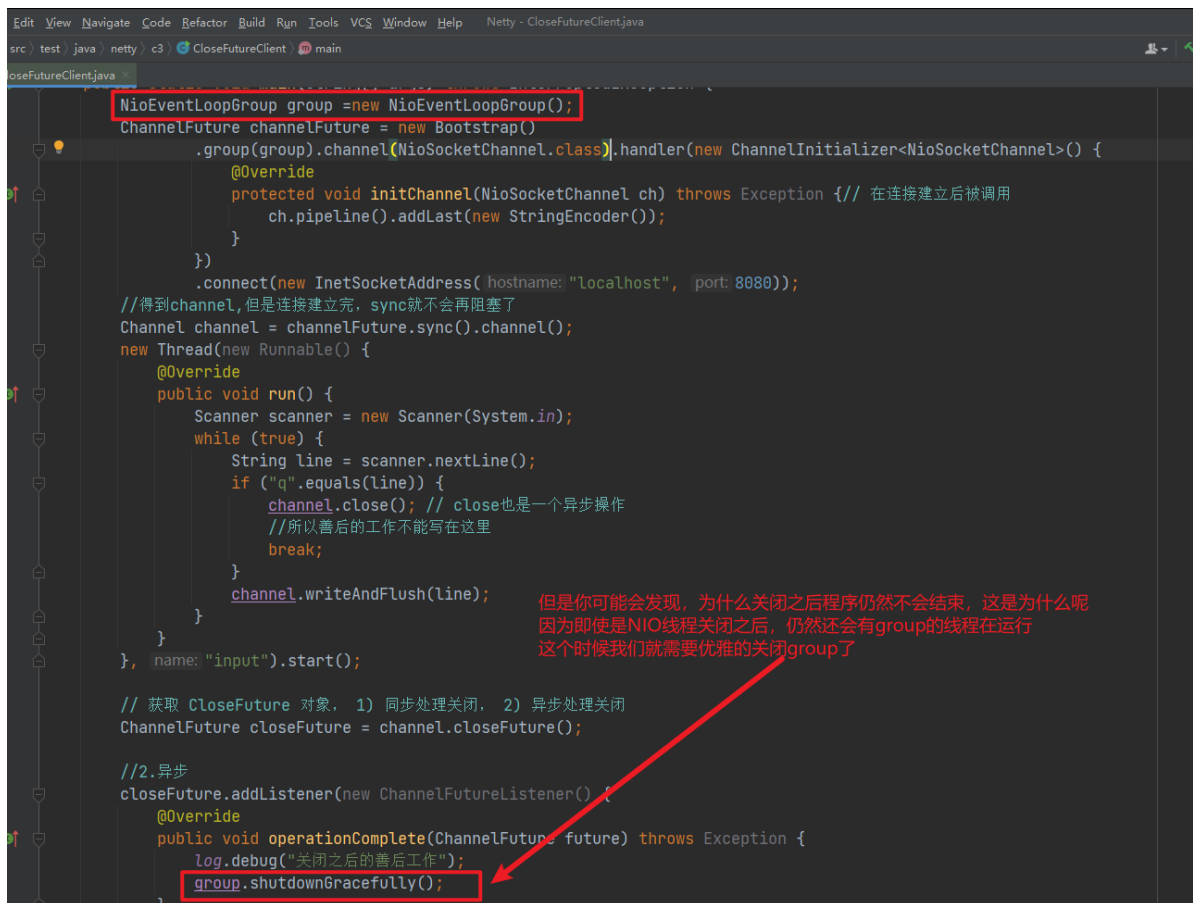
        break;
    }
    channel.writeAndFlush(line);
}
}
}, "input").start();

// 获取 closeFuture 对象， 1) 同步处理关闭， 2) 异步处理关闭
ChannelFuture closeFuture = channel.closeFuture();

//2.异步
closeFuture.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture future) throws
Exception {
        log.debug("关闭之后的善后工作");
    }
});
}
}

```

优雅地关闭线程



```
src test java netty c3 CloseFutureClient main
CloseFutureClient.java

NioEventLoopGroup group = new NioEventLoopGroup();
ChannelFuture channelFuture = new Bootstrap()
    .group(group).channel(NioSocketChannel.class).handler(new ChannelInitializer<NioSocketChannel>() {
        @Override
        protected void initChannel(NioSocketChannel ch) throws Exception {
            ch.pipeline().addLast(new StringEncoder());
        }
    })
    .connect(new InetSocketAddress("localhost", 8080));
//得到channel,但是连接建立完, sync就不会再阻塞了
Channel channel = channelFuture.sync().channel();
new Thread(new Runnable() {
    @Override
    public void run() {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            String line = scanner.nextLine();
            if ("q".equals(line)) {
                channel.close(); // close也是一个异步操作
                //所以善后的工作不能写在这里
                break;
            }
            channel.writeAndFlush(line);
        }
    }
}, name: "input").start();

// 获取 CloseFuture 对象, 1) 同步处理关闭, 2) 异步处理关闭
ChannelFuture closeFuture = channel.closeFuture();

//2. 异步
closeFuture.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture future) throws Exception {
        log.debug("关闭之后的善后工作");
        group.shutdownGracefully();
    }
});
```

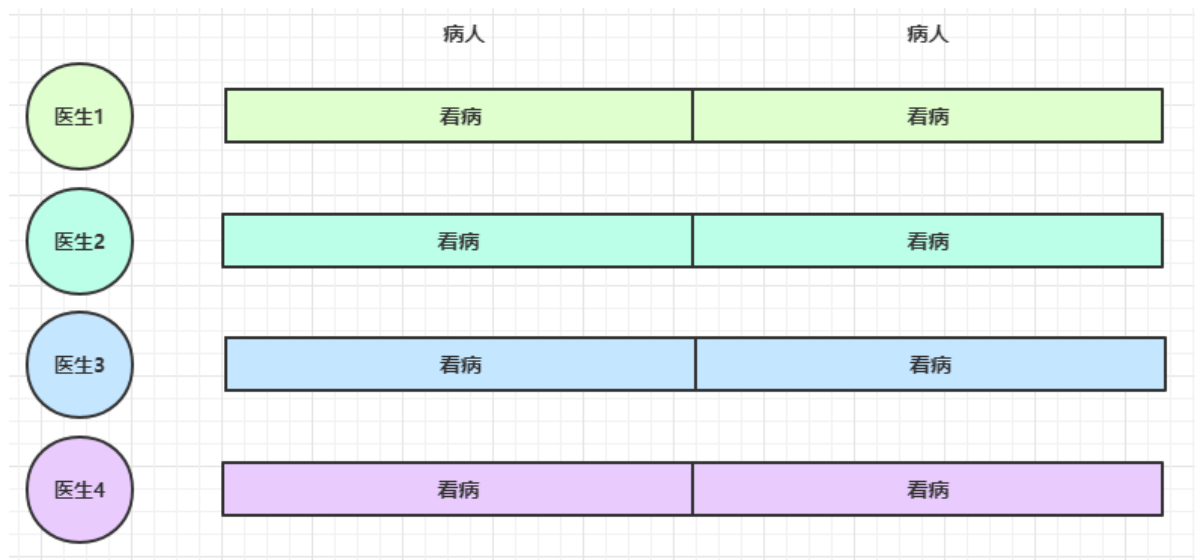
但是你会发现, 为什么关闭之后程序仍然不会结束, 这是为什么呢
因为即使是NIO线程关闭之后, 仍然还会有group的线程在运行
这个时候我们就需要优雅的关闭group了

####

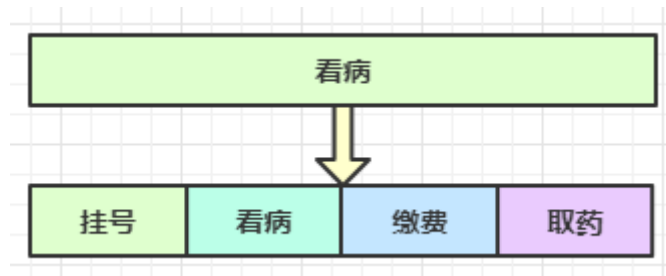
💡 异步提升的是什么

- 有些同学看到这里会有疑问: 为什么不在一个线程中去执行建立连接、去执行关闭 channel, 那样不是也可以吗? 非要用这么复杂的异步方式: 比如一个线程发起建立连接, 另一个线程去真正建立连接
- 还有同学会笼统地回答, 因为 netty 异步方式用了多线程、多线程就效率高。其实这些认识都比较片面, 多线程和异步所提升的效率并不是所认为的

思考下面的场景, 4 个医生给人看病, 每个病人花费 20 分钟, 而且医生看病的过程中是以病人为单位的, 一个病人看完了, 才能看下一个病人。假设病人源源不断地来, 可以计算一下 4 个医生一天工作 8 小时, 处理的病人总数是: $4 * 8 * 3 = 96$



经研究发现，看病可以细分为四个步骤，经拆分后每个步骤需要 5 分钟，如下



因此可以做如下优化，只有一开始，医生 2、3、4 分别要等待 5、10、15 分钟才能执行工作，但只要后续病人源源不断地来，他们就能够满负荷工作，并且处理病人的能力提高到了 $4 * 8 * 12$ 效率几乎是原来的四倍



要点

- 单线程没法异步提高效率，必须配合多线程、多核 cpu 才能发挥异步的优势
- 异步并没有缩短响应时间，反而有所增加
- 合理进行任务拆分，也是利用异步的关键

3.3 Future & Promise

在异步处理时，经常用到这两个接口

首先要说明 netty 中的 Future 与 jdk 中的 Future 同名，但是是两个接口，netty 的 Future 继承自 jdk 的 Future，而 Promise 又对 netty Future 进行了扩展

- jdk Future 只能同步等待任务结束（或成功、或失败）才能得到结果
- netty Future 可以同步等待任务结束得到结果，也可以异步方式得到结果，但都是要等任务结束
- netty Promise 不仅有 netty Future 的功能，而且脱离了任务独立存在，只作为两个线程间传递结果的容器

功能/名称	jdk Future	netty Future	Promise
cancel	取消任务	-	-
isCanceled	任务是否取消	-	-
isDone	任务是否完成，不能区分成功失败	-	-
get	获取任务结果，阻塞等待	-	-
getNow	-	获取任务结果，非阻塞，还未产生结果时返回 null	-
await	-	等待任务结束，如果任务失败，不会抛异常，而是通过 isSuccess 判断	-
sync	-	等待任务结束，如果任务失败，抛出异常	-
isSuccess	-	判断任务是否成功	-
cause	-	获取失败信息，非阻塞，如果没有失败，返回null	-
addListener	-	添加回调，异步接收结果	-
setSuccess	-	-	设置成功结果
setFailure	-	-	设置失败结果

JDK里面的future的get方法

future可以理解为一个容器，可以往里面取出东西，但是不能主动往里面放东西

```
package netty.c3;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.*;

@Slf4j
public class TestJDKFuture {
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        ExecutorService service = Executors.newFixedThreadPool(2);
        //submit方法是非阻塞的
        Future<Integer> future = service.submit(new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                log.debug("模拟执行一段时间");
            }
        });
    }
}
```

```

        Thread.sleep(1000);
        return 50;
    }
});
//通过future对象得到执行线程的返回值,get方法是阻塞的
log.debug("结果为{}", future.get());
}
}

```

结果:

```

D:\JDK1.8.1\JDK1.8\Java\jdk1.8.0_311\bin\java.exe ...
Connected to the target VM, address: '127.0.0.1:59632', transport: 'socket'
12:53:19.466 [pool-1-thread-1] DEBUG netty.c3.TestJDKFuture - 模拟执行一段时间
12:53:20.482 [main] DEBUG netty.c3.TestJDKFuture - 结果为50

```

Netty中的Future方法

```

package netty.c3;

import io.netty.channel.EventLoop;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.util.concurrent.Future;
import io.netty.util.concurrent.GenericFutureListener;
import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;

@Slf4j
public class TestNettyFuture {
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        NioEventLoopGroup group = new NioEventLoopGroup();
        EventLoop eventLoop = group.next();
        //得到netty的future对象
        Future<Integer> future = eventLoop.submit(new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                log.debug("模拟执行一段时间");
                Thread.sleep(1000);
                return 70;
            }
        });
        //这个是在主线程进行同步
        log.debug("得到结果{}", future.get());
        //新开一个线程, 进行异步
        future.addListener(new GenericFutureListener<Future<? super
Integer>>() {
            @Override

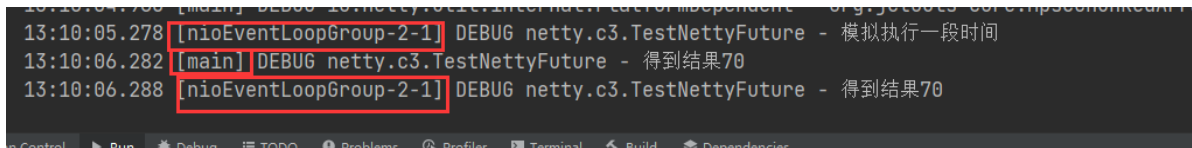
```



```

        public void operationComplete(Future<? super Integer> future)
        throws Exception {
            log.debug("得到结果{}", future.get());
        }
    });
}
}

```



```

13:10:05.278 [nioEventLoopGroup-2-1] DEBUG netty.c3.TestNettyFuture - 模拟执行一段时间
13:10:06.282 [main] DEBUG netty.c3.TestNettyFuture - 得到结果70
13:10:06.288 [nioEventLoopGroup-2-1] DEBUG netty.c3.TestNettyFuture - 得到结果70

```

netty的promise

```

package netty.c3;

import io.netty.channel.EventLoop;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.util.concurrent.DefaultPromise;
import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.ExecutionException;

@Slf4j
public class TestPromise {

    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        EventLoop eventLoop = new NioEventLoopGroup().next();
        DefaultPromise<Object> promise = new DefaultPromise<>(eventLoop);
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    log.debug("开始计算");
                    int i=1/0;
                    Thread.sleep(1000);
                    promise.setSuccess(60);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    promise.setFailure(e);
                }
            }
        }).start();
        log.debug("结果是: {}", promise.get());
    }
}

```

例1

同步处理任务成功

```
DefaultEventLoop eventExecutors = new DefaultEventLoop();  
DefaultPromise promise = new DefaultPromise<>(eventExecutors);
```

```
eventExecutors.execute()->{  
    try {  
        Thread.sleep(1000);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    log.debug("set success, {}",10);  
    promise.setSuccess(10);  
});  
  
log.debug("start...");  
log.debug!("{}",promise.getNow()); // 还没有结果  
log.debug!("{}",promise.get());
```

输出

```
11:51:53 [DEBUG] [main] c.i.o.DefaultPromiseTest2 - start...  
11:51:53 [DEBUG] [main] c.i.o.DefaultPromiseTest2 - null  
11:51:54 [DEBUG] [defaultEventLoop-1-1] c.i.o.DefaultPromiseTest2 - set success, 10  
11:51:54 [DEBUG] [main] c.i.o.DefaultPromiseTest2 - 10
```

例2

异步处理任务成功

```
DefaultEventLoop eventExecutors = new DefaultEventLoop();  
DefaultPromise promise = new DefaultPromise<>(eventExecutors);
```

// 设置回调, 异步接收结果

```
promise.addListener(future -> {  
    // 这里的 future 就是上面的 promise  
    log.debug("{}","future.getNow());  
});
```

// 等待 1000 后设置成功结果

```
eventExecutors.execute()->{  
    try {  
        Thread.sleep(1000);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    log.debug("set success, {}",10);  
    promise.setSuccess(10);  
});
```

```
log.debug("start...");
```

输出

```
11:49:30 [DEBUG] [main] c.i.o.DefaultPromiseTest2 - start...  
11:49:31 [DEBUG] [defaultEventLoop-1-1] c.i.o.DefaultPromiseTest2 - set success, 10  
11:49:31 [DEBUG] [defaultEventLoop-1-1] c.i.o.DefaultPromiseTest2 - 10
```

例3

同步处理任务失败 - sync & get

```
DefaultEventLoop eventExecutors = new DefaultEventLoop();
```

```
DefaultPromise promise = new DefaultPromise<>(eventExecutors);
```

```
eventExecutors.execute(() -> {  
    try {  
        Thread.sleep(1000);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    RuntimeException e = new RuntimeException("error...");  
    log.debug("set failure, {}", e.toString());  
    promise.setFailure(e);  
});  
  
log.debug("start...");  
log.debug("{} ", promise.getNow());  
promise.get(); // sync() 也会出现异常, 只是 get 会再用 ExecutionException 包一层异常
```

输出

```
12:11:07 [DEBUG] [main] c.i.o.DefaultPromiseTest2 - start...
```

```
12:11:07 [DEBUG] [main] c.i.o.DefaultPromiseTest2 - null
```

```
12:11:08 [DEBUG] [defaultEventLoop-1-1] c.i.o.DefaultPromiseTest2 - set failure,  
java.lang.RuntimeException: error...
```

```
Exception in thread "main" java.util.concurrent.ExecutionException: java.lang.RuntimeException:  
error...
```

```
    at io.netty.util.concurrent.AbstractFuture.get(AbstractFuture.java:41)
```

```
    at com.itcast.oio.DefaultPromiseTest2.main(DefaultPromiseTest2.java:34)
```

```
Caused by: java.lang.RuntimeException: error...
```

```
    at com.itcast.oio.DefaultPromiseTest2.lambda$main$0(DefaultPromiseTest2.java:27)
```

```
    at io.netty.channel.DefaultEventLoop.run(DefaultEventLoop.java:54)
```

```
    at
```

```
io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:918)
```

```
    at io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
```

```
    at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
```

at java.lang.Thread.run(Thread.java:745)

例4

同步处理任务失败 - await

```
DefaultEventLoop eventExecutors = new DefaultEventLoop();
DefaultPromise promise = new DefaultPromise<>(eventExecutors);

eventExecutors.execute() -> {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    RuntimeException e = new RuntimeException("error...");
    log.debug("set failure, {}", e.toString());
    promise.setFailure(e);
});

log.debug("start...");
log.debug!("{}", promise.getNow());
promise.await(); // 与 sync 和 get 区别在于，不会抛异常
log.debug("result {}", (promise.isSuccess() ? promise.getNow() : promise.cause()).toString());
```

输出

```
12:18:53 [DEBUG] [main] c.i.o.DefaultPromiseTest2 - start...
12:18:53 [DEBUG] [main] c.i.o.DefaultPromiseTest2 - null
12:18:54 [DEBUG] [defaultEventLoop-1-1] c.i.o.DefaultPromiseTest2 - set failure,
java.lang.RuntimeException: error...
12:18:54 [DEBUG] [main] c.i.o.DefaultPromiseTest2 - result java.lang.RuntimeException: error...
```

例5

异步处理任务失败

```
DefaultEventLoop eventExecutors = new DefaultEventLoop();
DefaultPromise promise = new DefaultPromise<>(eventExecutors);

promise.addListener(future -> {
    log.debug("result {}", (promise.isSuccess() ? promise.getNow() : promise.cause()).toString());
});

eventExecutors.execute() -> {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    RuntimeException e = new RuntimeException("error...");
    log.debug("set failure, {}", e.toString());
    promise.setFailure(e);
});

log.debug("start...");
```

输出

```
12:04:57 [DEBUG] [main] c.i.o.DefaultPromiseTest2 - start...
12:04:58 [DEBUG] [defaultEventLoop-1-1] c.i.o.DefaultPromiseTest2 - set failure,
java.lang.RuntimeException: error...
12:04:58 [DEBUG] [defaultEventLoop-1-1] c.i.o.DefaultPromiseTest2 - result
java.lang.RuntimeException: error...
```

例6

await 死锁检查

```

DefaultEventLoop eventExecutors = new DefaultEventLoop();

DefaultPromise promise = new DefaultPromise<>(eventExecutors);

eventExecutors.submit(()->{

    System.out.println("1");

    try {

        promise.await();

        // 注意不能仅捕获 InterruptedException 异常

        // 否则 死锁检查抛出的 BlockingOperationException 会继续向上传播

        // 而提交的任务会被包装为 PromiseTask, 它的 run 方法中会 catch 所有异常然后设置为 Promise
        // 的失败结果而不会抛出

        } catch (Exception e) {

            e.printStackTrace();

        }

        System.out.println("2");

    });

eventExecutors.submit(()->{

    System.out.println("3");

    try {

        promise.await();

    } catch (Exception e) {

        e.printStackTrace();

    }

    System.out.println("4");

    });

```

输出

```

1
2
3
4

```

io.netty.util.concurrent.BlockingOperationException: DefaultPromise@47499c2a(incomplete)

```
at io.netty.util.concurrent.DefaultPromise.checkDeadLock(DefaultPromise.java:384)
at io.netty.util.concurrent.DefaultPromise.await(DefaultPromise.java:212)
at com.itcast.oio.DefaultPromiseTest.lambda$main0(DefaultPromiseTest.java:27)
at io.netty.util.concurrent.PromiseTask$RunnableAdapter.call(PromiseTask.java:38)
at io.netty.util.concurrent.PromiseTask.run(PromiseTask.java:73)
at io.netty.channel.DefaultEventLoop.run(DefaultEventLoop.java:54)
at
io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:918)
at io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
at java.lang.Thread.run(Thread.java:745)
io.netty.util.concurrent.BlockingOperationException: DefaultPromise@47499c2a(incomplete)
at io.netty.util.concurrent.DefaultPromise.checkDeadLock(DefaultPromise.java:384)
at io.netty.util.concurrent.DefaultPromise.await(DefaultPromise.java:212)
at com.itcast.oio.DefaultPromiseTest.lambda$main1(DefaultPromiseTest.java:36)
at io.netty.util.concurrent.PromiseTask$RunnableAdapter.call(PromiseTask.java:38)
at io.netty.util.concurrent.PromiseTask.run(PromiseTask.java:73)
at io.netty.channel.DefaultEventLoop.run(DefaultEventLoop.java:54)
at
io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:918)
at io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
at java.lang.Thread.run(Thread.java:745)
```

###

3.4 Handler & Pipeline

ChannelHandler 用来处理 Channel 上的各种事件，分为入站、出站两种。所有 ChannelHandler 被连成一串，就是 Pipeline

- 入站处理器通常是 ChannelInboundHandlerAdapter 的子类，主要用来读取客户端数据，写回结果
- 出站处理器通常是 ChannelOutboundHandlerAdapter 的子类，主要对写回结果进行加工


```

        System.out.println(6);
        ctx.write(msg, promise); // 6
    }
    });
}
})
.bind(8080);

```

客户端

```

new Bootstrap()
    .group(new NioEventLoopGroup())
    .channel(NioSocketChannel.class)
    .handler(new ChannelInitializer<Channel>() {
        @Override
        protected void initChannel(Channel ch) {
            ch.pipeline().addLast(new StringEncoder());
        }
    })
    .connect("127.0.0.1", 8080)
    .addListener((ChannelFutureListener) future -> {
        future.channel().writeAndFlush("hello,world");
    });

```

服务器端打印:

```

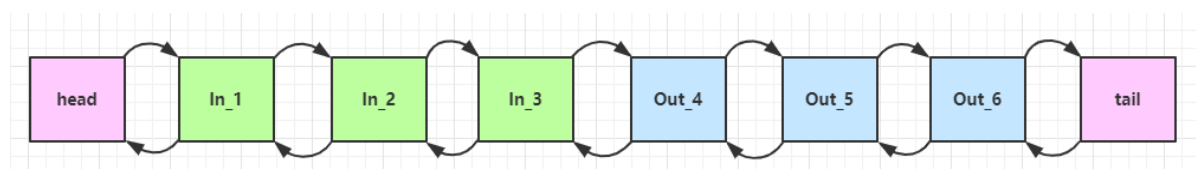
1
2
3
6
5
4

```

可以看到,

ChannelInboundHandlerAdapter 是按照 addLast 的顺序执行的,

而 ChannelOutboundHandlerAdapter 是按照 addLast 的逆序执行的。ChannelPipeline 的实现是一个 ChannelHandlerContext (包装了 ChannelHandler) 组成的双向链表



- 入站处理器中, `ctx.fireChannelRead(msg)` 是 **调用下一个入站处理器**
- - 如果注释掉 1 处代码, 则仅会打印 1
 - 如果注释掉 2 处代码, 则仅会打印 1 2
- 3 处的 `ctx.channel().write(msg)` 会 **从尾部开始触发** 后续出站处理器的执行
- - 如果注释掉 3 处代码, 则仅会打印 1 2 3
- 类似的, 出站处理器中, `ctx.write(msg, promise)` 的调用也会 **触发上一个出站处理器**
- - 如果注释掉 6 处代码, 则仅会打印 1 2 3 6

```

    super.write(ctx, msg, promise);
  });
  pipeline.addLast( name: "h3", (ChannelInboundHandlerAdapter)
    log.debug("3");
    ctx.writeAndFlush(ctx.alloc().buffer().writeBytes("
    ch.writeAndFlush(ctx.alloc().buffer().writeBytes(

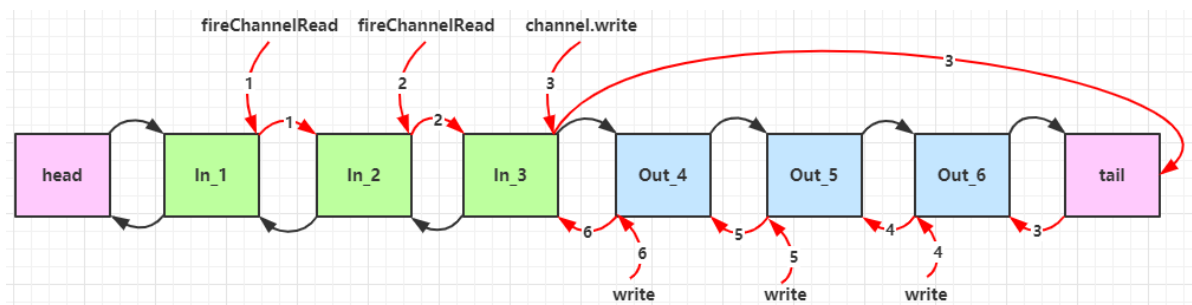
```

这个是从当前位置往前找

这个是从尾巴开始往前找

- `ctx.channel().write(msg)` vs `ctx.write(msg)`
- - 都是触发出站处理器的执行
 - `ctx.channel().write(msg)` 从尾部开始查找出站处理器
 - `ctx.write(msg)` 是从当前节点找上一个出站处理器
 - 3 处的 `ctx.channel().write(msg)` 如果改为 `ctx.write(msg)` 仅会打印 1 2 3, 因为节点3 之前没有其它出站处理器了
 - 6 处的 `ctx.write(msg, promise)` 如果改为 `ctx.channel().write(msg)` 会打印 1 2 3 6 6 6... 因为 `ctx.channel().write()` 是从尾部开始查找, 结果又是节点6 自己

图1 - 服务端 pipeline 触发的原始流程, 图中数字代表了处理步骤的先后次序



利用EmbeddedChannel 进行单元测试

```
package netty.c3;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.ByteBufAllocator;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.channel.ChannelOutboundHandlerAdapter;
import io.netty.channel.ChannelPromise;
import io.netty.channel.embedded.EmbeddedChannel;
import io.netty.handler.logging.LogLevel;
import io.netty.handler.logging.LoggingHandler;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;

import java.nio.charset.Charset;

@Slf4j
public class TestEmbeddedChannel {
    public static void main(String[] args) {
        ChannelInboundHandlerAdapter h1 = new ChannelInboundHandlerAdapter() {
            @Override
            public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
                log.debug("1");
                super.channelRead(ctx, msg);
            }
        };
        ChannelInboundHandlerAdapter h2 = new ChannelInboundHandlerAdapter() {
            @Override
            public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
                log.debug("2");
                super.channelRead(ctx, msg);
            }
        };
        ChannelOutboundHandlerAdapter h3 = new ChannelOutboundHandlerAdapter() {
            @Override
            public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise) throws Exception {
                log.debug("3");
                super.write(ctx, msg, promise);
            }
        };
        ChannelOutboundHandlerAdapter h4 = new ChannelOutboundHandlerAdapter() {
            @Override
            public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise) throws Exception {
                log.debug("4");
                super.write(ctx, msg, promise);
            }
        };
        EmbeddedChannel channel = new EmbeddedChannel(h1, h2, h3, h4);
    }
}
```

```

        // 模拟进站操作

channel.writeInbound(ByteBufAllocator.DEFAULT.buffer().writeBytes("hello".getBytes()));

        // 模拟出站操作

channel.writeOutbound(ByteBufAllocator.DEFAULT.buffer().writeBytes("world".getBytes()));

    }
}

```

3.5 ByteBuf

是对字节数据的封装

也可以直接无参，无参创建的大小是256，ByteBuf可以自动扩容

1) 创建

```

ByteBuf buffer = ByteBufAllocator.DEFAULT.buffer(10);
log(buffer);

```

上面代码创建了一个默认的 ByteBuf（池化基于直接内存的 ByteBuf），初始容量是 10

输出

```

read index:0 write index:0 capacity:10

```

其中 log 方法参考如下

```

private static void log(ByteBuf buffer) {
    int length = buffer.readableBytes();
    int rows = length / 16 + (length % 15 == 0 ? 0 : 1) + 4;
    StringBuilder buf = new StringBuilder(rows * 80 * 2)
        .append("read index:").append(buffer.readerIndex())
        .append(" write index:").append(buffer.writerIndex())
        .append(" capacity:").append(buffer.capacity())
        .append(NEWLINE);
    appendPrettyHexDump(buf, buffer);
    System.out.println(buf.toString());
}

```

2) 直接内存 vs 堆内存

可以使用下面的代码来创建池化基于堆的 ByteBuf

```
ByteBuf buffer = ByteBufAllocator.DEFAULT.heapBuffer(10);
```

也可以使用下面的代码来创建池化基于直接内存的 ByteBuf

```
ByteBuf buffer = ByteBufAllocator.DEFAULT.directBuffer(10);
```

- 直接内存创建和销毁的代价昂贵，但读写性能高（少一次内存复制），适合配合池化功能一起用
- 直接内存对 GC 压力小，因为这部分内存不受 JVM 垃圾回收的管理，但也要注意及时主动释放

3) 池化 vs 非池化

池化的最大意义在于可以重用 ByteBuf，优点有

- 没有池化，则每次都得创建新的 ByteBuf 实例，这个操作对直接内存代价昂贵，就算是堆内存，也会增加 GC 压力
- 有了池化，则可以重用池中 ByteBuf 实例，并且采用了与 jemalloc 类似的内存分配算法提升分配效率
- 高并发时，池化功能更节约内存，减少内存溢出的可能

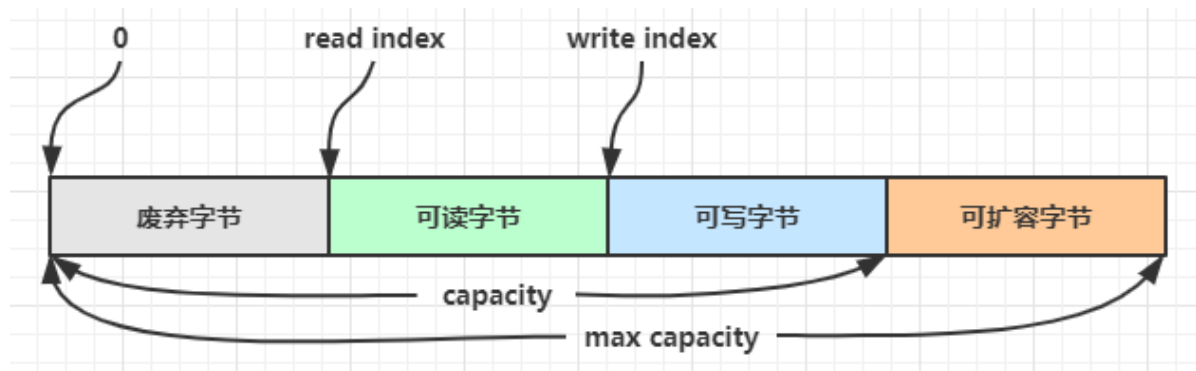
池化功能是否开启，可以通过下面的系统环境变量来设置

```
-Dio.netty allocator.type={unpooled|pooled}
```

- 4.1 以后，非 Android 平台默认启用池化实现，Android 平台启用非池化实现
- 4.1 之前，池化功能还不成熟，默认是非池化实现

4) 组成

ByteBuf 由四部分组成



最开始读写指针都在 0 位置

5) 写入

方法列表，省略一些不重要的方法

方法签名	含义	备注
writeBoolean(boolean value)	写入 boolean 值	用一字节 01 00 代表 true false
writeByte(int value)	写入 byte 值	
writeShort(int value)	写入 short 值	
writeInt(int value)	写入 int 值	Big Endian, 即 0x250, 写入后 00 00 02 50
writeIntLE(int value)	写入 int 值	Little Endian, 即 0x250, 写入后 50 02 00 00
writeLong(long value)	写入 long 值	
writeChar(int value)	写入 char 值	
writeFloat(float value)	写入 float 值	
writeDouble(double value)	写入 double 值	
writeBytes(ByteBuf src)	写入 netty 的 ByteBuf	
writeBytes(byte[] src)	写入 byte[]	

方法签名	含义	备注
<code>writeBytes(ByteBuffer src)</code>	写入 nio 的 ByteBuffer	
<code>int writeCharSequence(CharSequence sequence, Charset charset)</code>	写入字符串	

注意

- 这些方法的未指明返回值的，其返回值都是 `ByteBuffer`，意味着可以链式调用
- 网络传输，默认习惯是 Big Endian

先写入 4 个字节

```
buffer.writeBytes(new byte[]{1, 2, 3, 4});
log(buffer);
```

结果是

```
read index:0 write index:4 capacity:10
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 01 02 03 04                      |...|
+-----+-----+-----+-----+
```

再写入一个 int 整数，也是 4 个字节

```
buffer.writeInt(5);
log(buffer);
```

结果是


```
read index:0 write index:8 capacity:10
```

```
      +-----+
      | 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
+-----+-----+
|00000000| 01 02 03 04 00 00 00 05                |.....|
+-----+-----+
```

还有一类方法是 set 开头的一系列方法，也可以写入数据，但不会改变写指针位置

6) 扩容

再写入一个 int 整数时，容量不够了（初始容量是 10），这时会引发扩容

```
buffer.writeInt(6);
log(buffer);
```

扩容规则是

- 如何写入后数据大小未超过 512，则选择下一个 16 的整数倍，例如写入后大小为 12，则扩容后 capacity 是 16
- 如果写入后数据大小超过 512，则选择下一个 2^n ，例如写入后大小为 513，则扩容后 capacity 是 $2^{10}=1024$ (29=512 已经不够了)
- 扩容不能超过 max capacity 会报错

结果是

```
read index:0 write index:12 capacity:16
```

```
      +-----+
      | 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
+-----+-----+
|00000000| 01 02 03 04 00 00 00 05 00 00 00 06      |.....|
+-----+-----+
```

7) 读取

例如读了 4 次，每次一个字节

```
System.out.println(buffer.readByte());
System.out.println(buffer.readByte());
System.out.println(buffer.readByte());
System.out.println(buffer.readByte());
log(buffer);
```

读过的内容，就属于废弃部分了，再读只能读那些尚未读取的部分

```
1
2
3
4
read index:4 write index:12 capacity:16
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+
|00000000| 00 00 00 05 00 00 00 06      | ..... |
+-----+-----+
```

如果需要重复读取 int 整数 5，怎么办？

可以在 read 前先做个标记 mark

```
buffer.markReaderIndex();
System.out.println(buffer.readInt());
log(buffer);
```

结果

```

5
read index:8 write index:12 capacity:16
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+
|00000000| 00 00 00 06                                     |....|
+-----+-----+

```

这时要重复读取的话，重置到标记位置 reset

```

buffer.resetReaderIndex();
log(buffer);

```

这时

```

read index:4 write index:12 capacity:16
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+
|00000000| 00 00 00 05 00 00 00 06                             |.....|
+-----+-----+

```

还有种办法是采用 get 开头的一系列方法，这些方法不会改变 read index

8) retain & release

由于 Netty 中有堆外内存的 ByteBuf 实现，堆外内存最好是手动来释放，而不是等 GC 垃圾回收。

- UnpooledHeapByteBuf 使用的是 JVM 内存，只需等 GC 回收内存即可
- UnpooledDirectByteBuf 使用的就是直接内存了，需要特殊的方法来回收内存
- PooledByteBuf 和它的子类使用了池化机制，需要更复杂的规则来回收内存

回收内存的源码实现，请关注下面方法的不同实现

```

protected abstract void deallocate()

```

Netty 这里采用了引用计数法来控制回收内存，每个 ByteBuf 都实现了 ReferenceCounted 接口

- 每个 ByteBuf 对象的初始计数为 1
- 调用 release 方法计数减 1，如果计数为 0，ByteBuf 内存被回收
- 调用 retain 方法计数加 1，表示调用者没用完之前，其它 handler 即使调用了 release 也不会造成回收
- 当计数为 0 时，底层内存会被回收，这时即使 ByteBuf 对象还在，其各个方法均无法正常使用

谁来负责 release 呢？

不是我们想象的（一般情况下）

```
ByteBuf buf = ...  
try {  
    ...  
} finally {  
    buf.release();  
}
```

请思考，因为 pipeline 的存在，一般需要将 ByteBuf 传递给下一个 ChannelHandler，如果在 finally 中 release 了，就失去了传递性（当然，如果在这个 ChannelHandler 内这个 ByteBuf 已完成了它的使命，那么便无须再传递）

基本规则是，**谁最后使用者，谁负责 release**，详细分析如下

- 起点，对于 NIO 实现来讲，在 `io.netty.channel.nio.AbstractNioByteChannel.NioByteUnsafe#read` 方法中首次创建 ByteBuf 放入 pipeline (line 163 `pipeline.fireChannelRead(byteBuf)`)
- 进站 ByteBuf 处理原则
 - 对原始 ByteBuf 不做处理，调用 `ctx.fireChannelRead(msg)` 向后传递，这时无须 release
 - 将原始 ByteBuf 转换为其它类型的 Java 对象，这时 ByteBuf 就没用了，必须 release
 - 如果不调用 `ctx.fireChannelRead(msg)` 向后传递，那么也必须 release
 - 注意各种异常，如果 ByteBuf 没有成功传递到下一个 ChannelHandler，必须 release
 - 假设消息一直向后传，那么 TailContext 会负责释放未处理消息（原始的 ByteBuf）
- 出站 ByteBuf 处理原则
 - 出站消息最终都会转为 ByteBuf 输出，一直向前传，由 HeadContext flush 后 release
- 异常处理原则
 - 有时候不清楚 ByteBuf 被引用了多少次，但又必须彻底释放，可以循环调用 release 直到返回 true

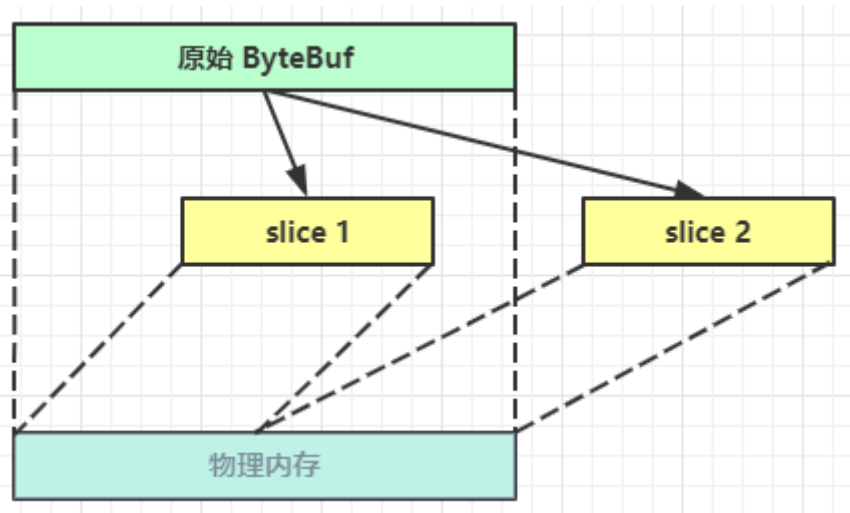
```
//
io.netty.channel.DefaultChannelPipeline#onUnhandledInboundMessage(java.lang.Object)
protected void onUnhandledInboundMessage(Object msg) {
    try {
        logger.debug(
            "Discarded inbound message {} that reached at the tail of the
pipeline. " +
            "Please check your pipeline configuration.", msg);
    } finally {
        ReferenceCountUtil.release(msg);
    }
}
```

具体代码

```
// io.netty.util.ReferenceCountUtil#release(java.lang.Object)
public static boolean release(Object msg) {
    if (msg instanceof ReferenceCounted) {
        return ((ReferenceCounted) msg).release();
    }
    return false;
}
```

9) slice

【零拷贝】的体现之一，对原始 ByteBuf 进行切片成多个 ByteBuf，切片后的 ByteBuf 并没有发生内存复制，还是使用原始 ByteBuf 的内存，切片后的 ByteBuf 维护独立的 read，write 指针



例，原始 ByteBuf 进行一些

初始操作

```
package netty.c4;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.ByteBufAllocator;

import static io.netty.buffer.ByteBufUtil.appendPrettyHexDump;
import static io.netty.util.internal.StringUtil.NEWLINE;

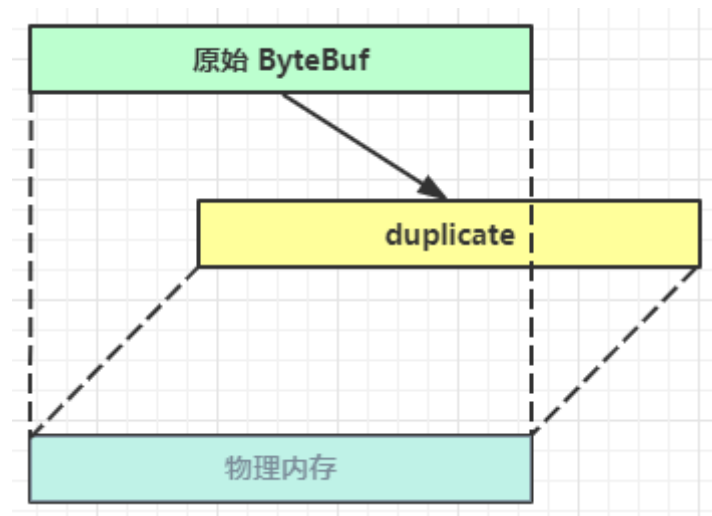
public class TestSlice {
    public static void main(String[] args) {
        ByteBuf buffer = ByteBufAllocator.DEFAULT.buffer(10);
        buffer.writeBytes(new byte[]{ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' });
        log(buffer);
        ByteBuf f1 = buffer.slice(0, 5);
        f1.retain(); //习惯
        ByteBuf f2 = buffer.slice(5, 5);
        f2.retain(); //习惯

        log(f1);
        log(f2);
        f1.setByte(0, 'b');
        log(buffer);
    }

    //打印日志的方法
    private static void log(ByteBuf buffer) {
        int length = buffer.readableBytes();
        int rows = length / 16 + (length % 15 == 0 ? 0 : 1) + 4;
        StringBuilder buf = new StringBuilder(rows * 80 * 2)
            .append("read index:").append(buffer.readerIndex())
            .append(" write index:").append(buffer.writerIndex())
            .append(" capacity:").append(buffer.capacity())
            .append(NEWLINE);
        appendPrettyHexDump(buf, buffer);
        System.out.println(buf.toString());
    }
}
```

10) duplicate

【零拷贝】的体现之一，就好比截取了原始 ByteBuf 所有内容，并且没有 max capacity 的限制，也是与原始 ByteBuf 使用同一块底层内存，只是读写指针是独立的



11) copy

会将底层内存数据进行深拷贝，因此无论读写，都与原始 ByteBuf 无关

12) CompositeByteBuf

【零拷贝】的体现之一，可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf，避免拷贝

```
package netty.c4;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.ByteBufAllocator;
import io.netty.buffer.ByteBufUtil;
import io.netty.buffer.CompositeByteBuf;

import static io.netty.buffer.ByteBufUtil.appendPrettyHexDump;
import static io.netty.util.internal.StringUtil.NEWLINE;

public class TestCompositeByteBuf {
    public static void main(String[] args) {
        ByteBuf buf1 = ByteBufAllocator.DEFAULT.buffer(5);
        buf1.writeBytes(new byte[]{1, 2, 3, 4, 5});
        ByteBuf buf2 = ByteBufAllocator.DEFAULT.buffer(5);
        buf2.writeBytes(new byte[]{6, 7, 8, 9, 10});
    }
}
```

```

CompositeByteBuf buf = ByteBufAllocator.DEFAULT.compositeBuffer();
//一定要加这个true
buf.addComponents(true,buf1,buf2);
log(buf);
}
private static void log(ByteBuf buffer) {
    int length = buffer.readableBytes();
    int rows = length / 16 + (length % 15 == 0 ? 0 : 1) + 4;
    StringBuilder buf = new StringBuilder(rows * 80 * 2)
        .append("read index:").append(buffer.readerIndex())
        .append(" write index:").append(buffer.writerIndex())
        .append(" capacity:").append(buffer.capacity())
        .append(NEWLINE);
    appendPrettyHexDump(buf, buffer);
    System.out.println(buf.toString());
}
}

```

结果

```

20:33:37.443 [main] DEBUG io.netty.buffer.AbstractByteBuf - -Dio.netty.buffer.c
20:33:37.443 [main] DEBUG io.netty.buffer.AbstractByteBuf - -Dio.netty.buffer.c
20:33:37.445 [main] DEBUG io.netty.util.ResourceLeakDetectorFactory - Loaded de
read index:0 write index:10 capacity:10
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 01 02 03 04 05 06 07 08 09 0a |.....|
+-----+

Process finished with exit code 0

```

CompositeByteBuf 是一个组合的 ByteBuf，它内部维护了一个 Component 数组，每个 Component 管理一个 ByteBuf，记录了这个 ByteBuf 相对于整体偏移量等信息，代表着整体中某一段的数据。

- 优点，对外是一个虚拟视图，组合这些 ByteBuf 不会产生内存复制
- 缺点，复杂了很多，多次操作会带来性能的损耗

13) Unpooled

Unpooled 是一个工具类，类如其名，提供了非池化的 ByteBuf 创建、组合、复制等操作

这里仅介绍其跟【零拷贝】相关的 wrappedBuffer 方法，可以用来包装 ByteBuf


```

ByteBuf buf1 = ByteBufAllocator.DEFAULT.buffer(5);
buf1.writeBytes(new byte[]{1, 2, 3, 4, 5});
ByteBuf buf2 = ByteBufAllocator.DEFAULT.buffer(5);
buf2.writeBytes(new byte[]{6, 7, 8, 9, 10});

// 当包装 ByteBuf 个数超过一个时，底层使用了 CompositeByteBuf
ByteBuf buf3 = Unpooled.wrappedBuffer(buf1, buf2);
System.out.println(ByteBufUtil.prettyHexDump(buf3));

```

输出

```

+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 01 02 03 04 05 06 07 08 09 0a |.....|
+-----+

```

也可以用来包装普通字节数组，底层也不会有拷贝操作

```

ByteBuf buf4 = Unpooled.wrappedBuffer(new byte[]{1, 2, 3}, new byte[]{4, 5, 6});
System.out.println(buf4.getClass());
System.out.println(ByteBufUtil.prettyHexDump(buf4));

```

输出

```

class io.netty.buffer.CompositeByteBuf
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 01 02 03 04 05 06 |.....|
+-----+

```

💡 ByteBuf 优势

- 池化 - 可以重用池中 ByteBuf 实例，更节约内存，减少内存溢出的可能
- 读写指针分离，不需要像 ByteBuffer 一样切换读写模式
- 可以自动扩容

- 支持链式调用，使用更流畅
- 很多地方体现零拷贝，例如 slice、duplicate、CompositeByteBuf

4. 双向通信

4.1 练习

实现一个 echo server

编写 server

```
new ServerBootstrap()
    .group(new NioEventLoopGroup())
    .channel(NioServerSocketChannel.class)
    .childHandler(new ChannelInitializer<NioSocketChannel>() {
        @Override
        protected void initChannel(NioSocketChannel ch) {
            ch.pipeline().addLast(new ChannelInboundHandlerAdapter(){
                @Override
                public void channelRead(ChannelHandlerContext ctx, Object msg) {
                    ByteBuf buffer = (ByteBuf) msg;

                    System.out.println(buffer.toString(Charset.defaultCharset()));

                    // 建议使用 ctx.alloc() 创建 ByteBuf
                    ByteBuf response = ctx.alloc().buffer();
                    response.writeBytes(buffer);
                    ctx.writeAndFlush(response);

                    // 思考：需要释放 buffer 吗
                    // 思考：需要释放 response 吗

                }
            });
        }
    }).bind(8080);
```

编写 client

```
NioEventLoopGroup group = new NioEventLoopGroup();
Channel channel = new Bootstrap()
    .group(group)
    .channel(NioSocketChannel.class)
    .handler(new ChannelInitializer<NioSocketChannel>() {
```

```

@Override
protected void initChannel(NioSocketChannel ch) throws Exception {
    ch.pipeline().addLast(new StringEncoder());
    ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
        @Override
        public void channelRead(ChannelHandlerContext ctx, Object msg) {
            ByteBuf buffer = (ByteBuf) msg;

            System.out.println(buffer.toString(Charset.defaultCharset()));

            // 思考: 需要释放 buffer 吗
        }
    });
}

}).connect("127.0.0.1", 8080).sync().channel();

channel.closeFuture().addListener(future -> {
    group.shutdownGracefully();
});

new Thread(() -> {
    Scanner scanner = new Scanner(System.in);
    while (true) {
        String line = scanner.nextLine();
        if ("q".equals(line)) {
            channel.close();
            break;
        }
        channel.writeAndFlush(line);
    }
}).start();

```

💡 读和写的误解

我最初在认识上有这样的误区，认为只有在 netty, nio 这样的多路复用 IO 模型时，读写才不会相互阻塞，才可以实现高效的双向通信，但实际上，Java Socket 是全双工的：在任意时刻，线路上存在 A 到 B 和 B 到 A 的双向信号传输。即使是阻塞 IO，读和写是可以同时进行的，只要分别采用读线程和写线程即可，读不会阻塞写、写也不会阻塞读

例如

```

public class TestServer {
    public static void main(String[] args) throws IOException {
        ServerSocket ss = new ServerSocket(8888);
        Socket s = ss.accept();

        new Thread(() -> {
            try {

```

```

        BufferedReader reader = new BufferedReader(new
InputStreamReader(s.getInputStream()));
        while (true) {
            System.out.println(reader.readLine());
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}).start();

new Thread(() -> {
    try {
        BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(s.getOutputStream()));
        // 例如在这个位置加入 thread 级别断点，可以发现即使不写入数据，也不妨碍前面
        线程读取客户端数据
        for (int i = 0; i < 100; i++) {
            writer.write(String.valueOf(i));
            writer.newLine();
            writer.flush();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}).start();
}
}

```

客户端

```

public class TestClient {
    public static void main(String[] args) throws IOException {
        Socket s = new Socket("localhost", 8888);

        new Thread(() -> {
            try {
                BufferedReader reader = new BufferedReader(new
InputStreamReader(s.getInputStream()));
                while (true) {
                    System.out.println(reader.readLine());
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }).start();

        new Thread(() -> {
            try {
                BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(s.getOutputStream()));
                for (int i = 0; i < 100; i++) {
                    writer.write(String.valueOf(i));
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }).start();
    }
}

```

```

        writer.newLine();
        writer.flush();
    }
} catch (IOException e) {
    e.printStackTrace();
}
}).start();
}
}

```

三. Netty 进阶

1. 粘包与半包

1.1 粘包现象

服务端代码

```

public class HelloWorldServer {
    static final Logger log = LoggerFactory.getLogger(HelloWorldServer.class);
    void start() {
        NioEventLoopGroup boss = new NioEventLoopGroup(1);
        NioEventLoopGroup worker = new NioEventLoopGroup();
        try {
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.group(boss, worker);
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new LoggingHandler(LogLevel.DEBUG));
                    ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
                        @Override
                        public void channelActive(ChannelHandlerContext ctx)
                            throws Exception {
                            log.debug("connected {}", ctx.channel());
                            super.channelActive(ctx);
                        }

                        @Override
                        public void channelInactive(ChannelHandlerContext ctx)
                            throws Exception {
                            log.debug("disconnect {}", ctx.channel());
                            super.channelInactive(ctx);
                        }
                    });
                }
            });
            ChannelFuture channelFuture = serverBootstrap.bind(8080);
            log.debug("{} binding...", channelFuture.channel());
            channelFuture.sync();
            log.debug("{} bound...", channelFuture.channel());
            channelFuture.channel().closeFuture().sync();
        } catch (Exception e) {
            log.error("server bootstrap failed", e);
        }
    }
}

```

```

    } catch (InterruptedException e) {
        log.error("server error", e);
    } finally {
        boss.shutdownGracefully();
        worker.shutdownGracefully();
        log.debug("stoped");
    }
}

public static void main(String[] args) {
    new HelloWorldServer().start();
}
}

```

客户端代码希望发送 10 个消息，每个消息是 16 字节

```

public class HelloWorldClient {
    static final Logger log = LoggerFactory.getLogger(HelloWorldClient.class);
    public static void main(String[] args) {
        NioEventLoopGroup worker = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.group(worker);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    log.debug("connnetted...");
                    ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
                        @Override
                        public void channelActive(ChannelHandlerContext ctx)
throws Exception {
                            log.debug("sending...");
                            Random r = new Random();
                            char c = 'a';
                            for (int i = 0; i < 10; i++) {
                                ByteBuf buffer = ctx.alloc().buffer();
                                buffer.writeBytes(new byte[]{0, 1, 2, 3, 4, 5,
6, 7, 8, 9, 10, 11, 12, 13, 14, 15});
                                ctx.writeAndFlush(buffer);
                            }
                        }
                    });
                }
            });
            ChannelFuture channelFuture = bootstrap.connect("127.0.0.1",
8080).sync();
            channelFuture.channel().closeFuture().sync();

        } catch (InterruptedException e) {
            log.error("client error", e);
        } finally {

```

```

        worker.shutdownGracefully();
    }
}
}

```

服务器端的某次输出，可以看到一次就接收了 160 个字节，而非分 10 次接收

```

08:24:46 [DEBUG] [main] c.i.n.HelloWorldServer - [id: 0x81e0fda5] binding...
08:24:46 [DEBUG] [main] c.i.n.HelloWorldServer - [id: 0x81e0fda5,
L:/0:0:0:0:0:0:0:0:8080] bound...
08:24:55 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x94132411, L:/127.0.0.1:8080 - R:/127.0.0.1:58177] REGISTERED
08:24:55 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x94132411, L:/127.0.0.1:8080 - R:/127.0.0.1:58177] ACTIVE
08:24:55 [DEBUG] [nioEventLoopGroup-3-1] c.i.n.HelloWorldServer - connected [id:
0x94132411, L:/127.0.0.1:8080 - R:/127.0.0.1:58177]
08:24:55 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x94132411, L:/127.0.0.1:8080 - R:/127.0.0.1:58177] READ: 160B
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+-----+-----+-----+-----+
|00000000| 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
|00000010| 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
|00000020| 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
|00000030| 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
|00000040| 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
|00000050| 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
|00000060| 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
|00000070| 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
|00000080| 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
|00000090| 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
+-----+-----+-----+-----+-----+-----+-----+-----+
08:24:55 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x94132411, L:/127.0.0.1:8080 - R:/127.0.0.1:58177] READ COMPLETE

```

1.2 半包现象

客户端代码希望发送 1 个消息，这个消息是 160 字节，代码改为

```

ByteBuffer buffer = ctx.alloc().buffer();
for (int i = 0; i < 10; i++) {
    buffer.writeBytes(new byte[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15});
}
ctx.writeAndFlush(buffer);

```

为现象明显，服务端修改一下接收缓冲区，其它代码不变

```
serverBootstrap.option(ChannelOption.SO_RCVBUF, 10);
```

服务器端的某次输出，可以看到接收的消息被分为两节，第一次 20 字节，第二次 140 字节

```
08:43:49 [DEBUG] [main] c.i.n.HelloWorldServer - [id: 0x4d6c6a84] binding...
08:43:49 [DEBUG] [main] c.i.n.HelloWorldServer - [id: 0x4d6c6a84,
L:/0:0:0:0:0:0:0:0:8080] bound...
08:44:23 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x1719abf7, L:/127.0.0.1:8080 - R:/127.0.0.1:59221] REGISTERED
08:44:23 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x1719abf7, L:/127.0.0.1:8080 - R:/127.0.0.1:59221] ACTIVE
08:44:23 [DEBUG] [nioEventLoopGroup-3-1] c.i.n.HelloWorldServer - connected [id:
0x1719abf7, L:/127.0.0.1:8080 - R:/127.0.0.1:59221]
08:44:24 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x1719abf7, L:/127.0.0.1:8080 - R:/127.0.0.1:59221] READ: 20B
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+
|00000000| 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
|00000010| 00 01 02 03                                     |....|
+-----+-----+
08:44:24 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x1719abf7, L:/127.0.0.1:8080 - R:/127.0.0.1:59221] READ COMPLETE
08:44:24 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x1719abf7, L:/127.0.0.1:8080 - R:/127.0.0.1:59221] READ: 140B
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+
|00000000| 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00 01 02 03 |.....|
|00000010| 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00 01 02 03 |.....|
|00000020| 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00 01 02 03 |.....|
|00000030| 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00 01 02 03 |.....|
|00000040| 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00 01 02 03 |.....|
|00000050| 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00 01 02 03 |.....|
|00000060| 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00 01 02 03 |.....|
|00000070| 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00 01 02 03 |.....|
|00000080| 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f           |.....|
+-----+-----+
08:44:24 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x1719abf7, L:/127.0.0.1:8080 - R:/127.0.0.1:59221] READ COMPLETE
```

注意

serverBootstrap.option(ChannelOption.SO_RCVBUF, 10) 影响的底层接收缓冲区（即滑动窗口）大小，仅决定了 netty 读取的最小单位，netty 实际每次读取的一般是它的整数倍

1.3 现象分析

粘包

- 现象，发送 abc def，接收 abcdef
- 原因
 - 应用层：接收方 ByteBuf 设置太大（Netty 默认 1024）
 - 滑动窗口：假设发送方 256 bytes 表示一个完整报文，但由于接收方处理不及时且窗口大小足够大，这 256 bytes 字节就会缓冲在接收方的滑动窗口中，当滑动窗口中缓冲了多个报文就会粘包
 - Nagle 算法：会造成粘包

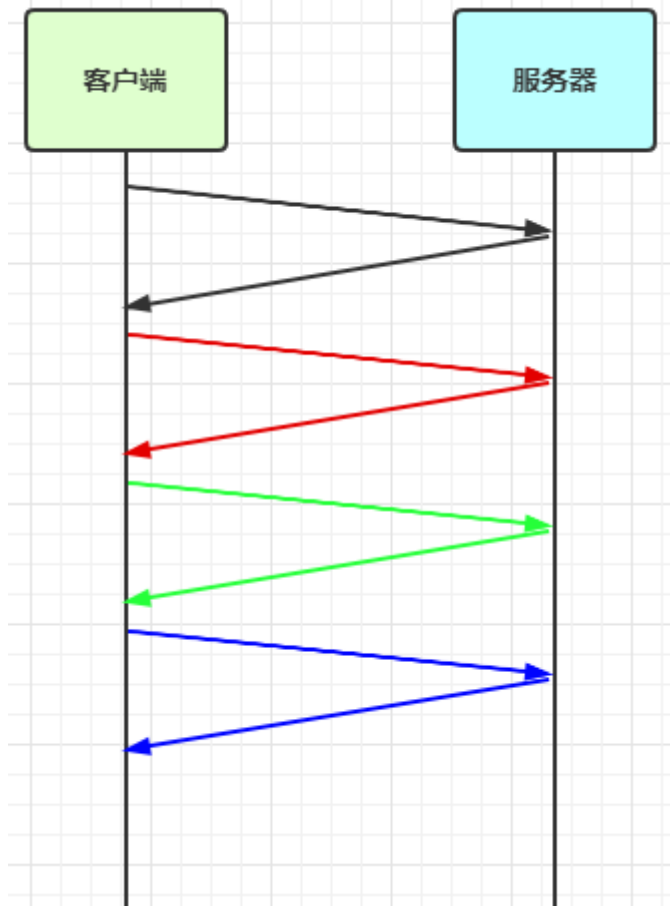
半包

- 现象，发送 abcdef，接收 abc def
- 原因
 - 应用层：接收方 ByteBuf 小于实际发送数据量
 - 滑动窗口：假设接收方的窗口只剩了 128 bytes，发送方的报文大小是 256 bytes，这时放不下了，只能先发送前 128 bytes，等待 ack 后才能发送剩余部分，这就造成了半包
 - MSS 限制：当发送的数据超过 MSS 限制后，会将数据切分发送，就会造成半包

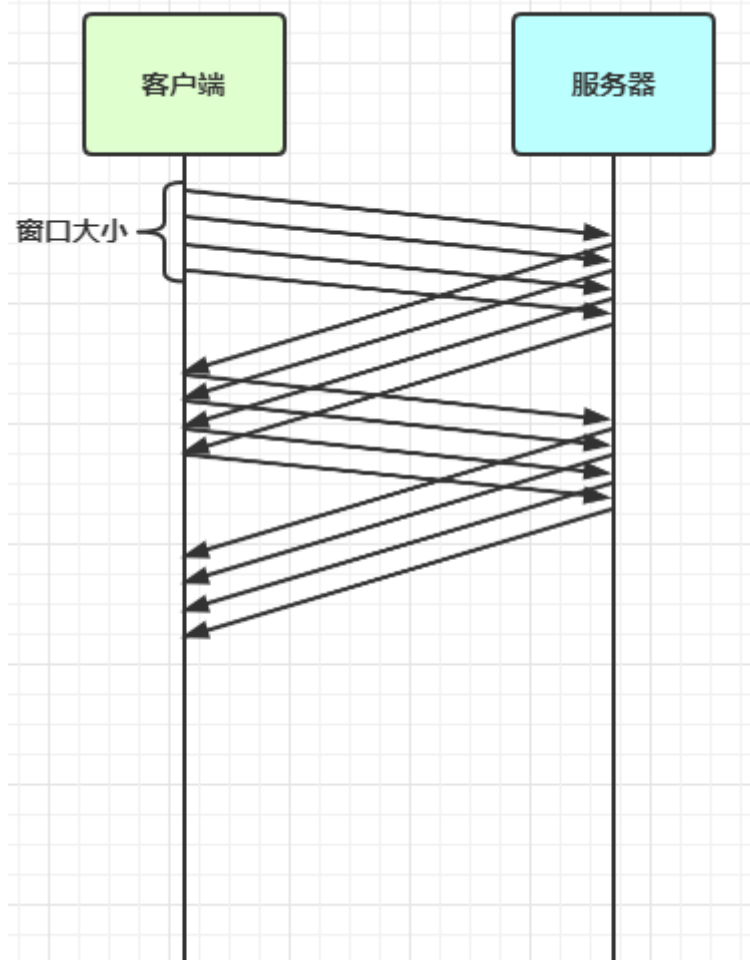
本质是因为 TCP 是流式协议，消息无边界

滑动窗口

- TCP 以一个段 (segment) 为单位, 每发送一个段就需要进行一次确认应答 (ack) 处理, 但如果这么做, 缺点是包的往返时间越长性能就越差



- 为了解决此问题, 引入了窗口概念, 窗口大小即决定了无需等待应答而可以继续发送的数据最大值

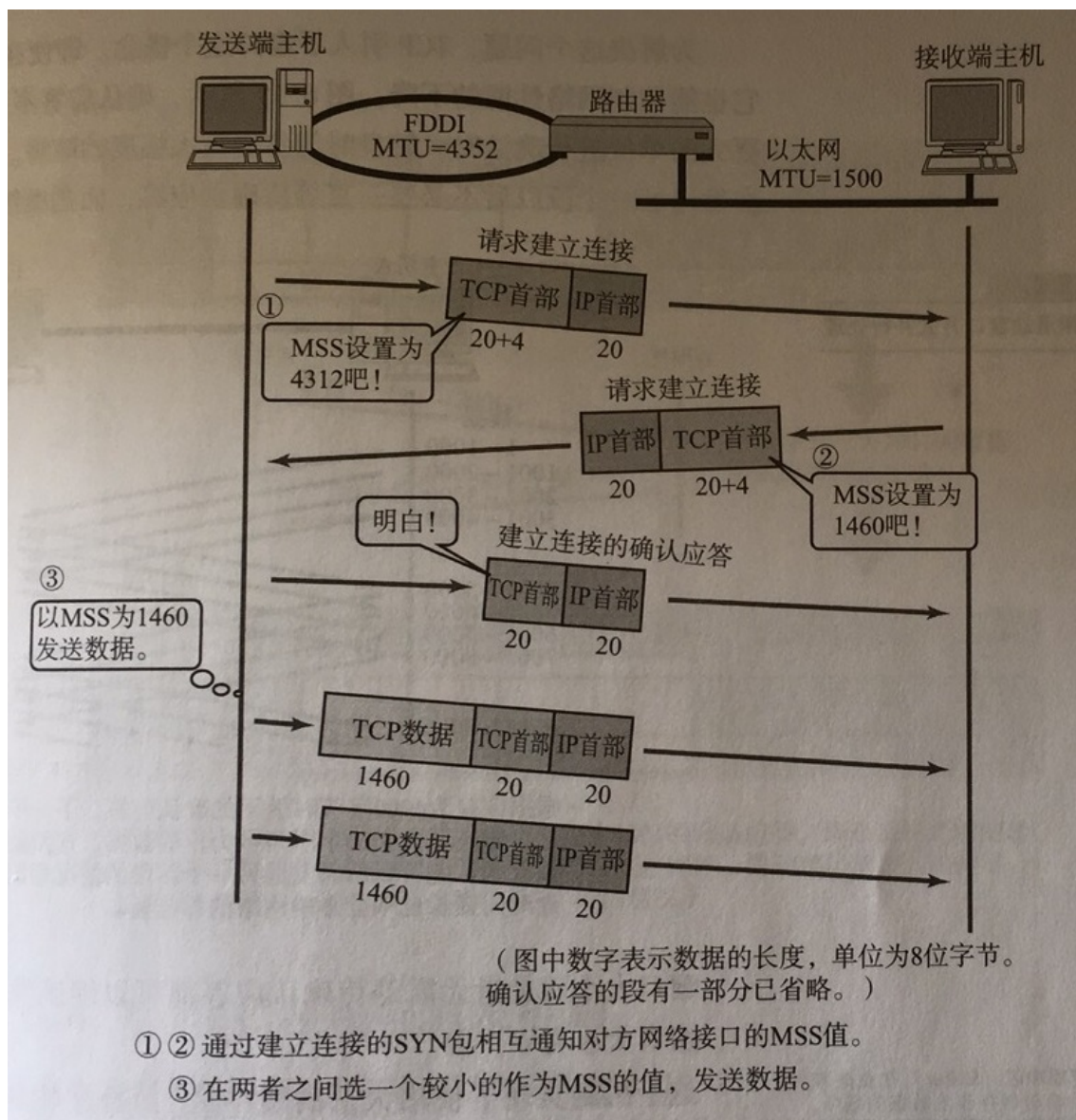


- 窗口实际就起到一个缓冲区的作用, 同时也能起到流量控制的作用

- - 图中深色的部分即要发送的数据，高亮的部分即窗口
 - 窗口内的数据才允许被发送，当应答未到达前，窗口必须停止滑动
 - 如果 1001~2000 这个段的数据 ack 回来了，窗口就可以向前滑动
 - 接收方也会维护一个窗口，只有落在窗口内的数据才能允许接收

MSS 限制

- 链路层对一次能够发送的最大数据有限制，这个限制称之为 MTU (maximum transmission unit)，不同的链路设备的 MTU 值也有所不同，例如
- 以太网的 MTU 是 1500
- FDDI (光纤分布式数据接口) 的 MTU 是 4352
- 本地回环地址的 MTU 是 65535 - 本地测试不走网卡
- MSS 是最大段长度 (maximum segment size)，它是 MTU 刨去 tcp 头和 ip 头后剩余能够作为数据传输的字节数
- ipv4 tcp 头占用 20 bytes，ip 头占用 20 bytes，因此以太网 MSS 的值为 $1500 - 40 = 1460$
- TCP 在传递大量数据时，会按照 MSS 大小将数据进行分割发送
- MSS 的值在三次握手时通知对方自己 MSS 的值，然后在两者之间选择一个小值作为 MSS



Nagle 算法

- 即使发送一个字节，也需要加入 tcp 头和 ip 头，也就是总字节数会使用 41 bytes，非常不经济。因此为了提高网络利用率，tcp 希望尽可能发送足够大的数据，这就是 Nagle 算法产生的缘由
- 该算法是指发送端即使还有应该发送的数据，但如果这部分数据很少的话，则进行延迟发送
- - 如果 SO_SNDBUF 的数据达到 MSS，则需要发送
 - 如果 SO_SNDBUF 中含有 FIN（表示需要连接关闭）这时将剩余数据发送，再关闭
 - 如果 TCP_NODELAY = true，则需要发送
 - 已发送的数据都收到 ack 时，则需要发送
 - 上述条件不满足，但发生超时（一般为 200ms）则需要发送
 - 除上述情况，延迟发送

1.4 解决方案

1. 短链接，发一个包建立一次连接，这样连接建立到连接断开之间就是消息的边界，缺点效率太低
2. 每一条消息采用固定长度，缺点浪费空间
3. 每一条消息采用分隔符，例如 \n，缺点需要转义
4. 每一条消息分为 head 和 body，head 中包含 body 的长度

方法1，短链接

以解决粘包为例

```
public class HelloWorldClient {
    static final Logger log = LoggerFactory.getLogger(HelloWorldClient.class);

    public static void main(String[] args) {
        // 分 10 次发送
        for (int i = 0; i < 10; i++) {
            send();
        }
    }

    private static void send() {
        NioEventLoopGroup worker = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.group(worker);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    log.debug("connected...");
                    ch.pipeline().addLast(new LoggingHandler(LogLevel.DEBUG));
                    ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
                        @Override
                        public void channelActive(ChannelHandlerContext ctx)
                            throws Exception {
                            log.debug("sending...");
                            ByteBuf buffer = ctx.alloc().buffer();
                            buffer.writeBytes(new byte[]{0, 1, 2, 3, 4, 5, 6, 7,
                                8, 9, 10, 11, 12, 13, 14, 15});
                            ctx.writeAndFlush(buffer);
                            // 发完即关
                            ctx.close();
                        }
                    });
                }
            });
        } catch {}
    }
}
```

```

        ChannelFuture channelFuture = bootstrap.connect("localhost",
8080).sync();
        channelFuture.channel().closeFuture().sync();

    } catch (InterruptedException e) {
        log.error("client error", e);
    } finally {
        worker.shutdownGracefully();
    }
}
}

```

输出，略

半包用这种办法还是不好解决，因为接收方的缓冲区大小是有限的

方法2，固定长度

让所有数据包长度固定（假设长度为 8 字节），服务器端加入

（要注意，这个解码器要放在所有handle之前）

```

//这是一个解码器，是8是因为我们可以和客户端约定好每条信息都是长度为8的固定大小
ch.pipeline().addLast(new FixedLengthFrameDecoder(8));

```

客户端测试代码，注意，采用这种方法后，客户端什么时候 flush 都可以

```

public class HelloWorldClient {
    static final Logger log = LoggerFactory.getLogger(HelloWorldClient.class);

    public static void main(String[] args) {
        NioEventLoopGroup worker = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.group(worker);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    log.debug("connnetted...");
                    ch.pipeline().addLast(new LoggingHandler(LogLevel.DEBUG));
                    ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
                        @Override
                        public void channelActive(ChannelHandlerContext ctx)
throws Exception {

```

```

        log.debug("sending...");
        // 发送内容随机的数据包
        Random r = new Random();
        char c = 'a';
        ByteBuffer buffer = ctx.alloc().buffer();
        for (int i = 0; i < 10; i++) {
            byte[] bytes = new byte[8];
            for (int j = 0; j < r.nextInt(8); j++) {
                bytes[j] = (byte) c;
            }
            c++;
            buffer.writeBytes(bytes);
        }
        ctx.writeAndFlush(buffer);
    }
}

});
ChannelFuture channelFuture = bootstrap.connect("192.168.0.103",
9090).sync();
channelFuture.channel().closeFuture().sync();

} catch (InterruptedException e) {
    log.error("client error", e);
} finally {
    worker.shutdownGracefully();
}
}
}

```

客户端输出

```

12:07:00 [DEBUG] [nioEventLoopGroup-2-1] c.i.n.HelloWorldClient - connetted...
12:07:00 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0x3c2ef3c2] REGISTERED
12:07:00 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0x3c2ef3c2] CONNECT: /192.168.0.103:9090
12:07:00 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0x3c2ef3c2, L:/192.168.0.103:53155 - R:/192.168.0.103:9090] ACTIVE
12:07:00 [DEBUG] [nioEventLoopGroup-2-1] c.i.n.HelloWorldClient - sending...
12:07:00 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0x3c2ef3c2, L:/192.168.0.103:53155 - R:/192.168.0.103:9090] WRITE: 80B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 61 61 61 61 00 00 00 00 62 00 00 00 00 00 00 |aaaa....b.....|
|00000010| 63 63 00 00 00 00 00 00 64 00 00 00 00 00 00 |cc.....d.....|
|00000020| 00 00 00 00 00 00 00 00 66 66 66 66 00 00 00 |.....ffff....|
|00000030| 67 67 67 00 00 00 00 00 68 00 00 00 00 00 00 |ggg....h.....|
|00000040| 69 69 69 69 69 00 00 00 6a 6a 6a 6a 00 00 00 |iiii...jjj....|
+-----+

```

```
12:07:00 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0x3c2ef3c2, L:/192.168.0.103:53155 - R:/192.168.0.103:9090] FLUSH
```

服务端输出

```
12:06:51 [DEBUG] [main] c.i.n.HelloWorldServer - [id: 0xe3d9713f] binding...
12:06:51 [DEBUG] [main] c.i.n.HelloWorldServer - [id: 0xe3d9713f,
L:/192.168.0.103:9090] bound...
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] REGISTERED
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] ACTIVE
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] c.i.n.HelloWorldServer - connected [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155]
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] READ: 8B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 61 61 61 61 00 00 00 00 |aaaa...|
+-----+
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] READ: 8B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 62 00 00 00 00 00 00 00 |b.....|
+-----+
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] READ: 8B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 63 63 00 00 00 00 00 00 |cc.....|
+-----+
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] READ: 8B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 64 00 00 00 00 00 00 00 |d.....|
+-----+
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] READ: 8B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 00 00 00 00 00 00 00 00 |.....|
+-----+
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] READ: 8B
+-----+
```



```

      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 66 66 66 66 00 00 00 00          |ffff....|
+-----+-----+-----+-----+
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] READ: 8B
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 67 67 67 00 00 00 00 00          |ggg....|
+-----+-----+-----+-----+
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] READ: 8B
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 68 00 00 00 00 00 00 00          |h.....|
+-----+-----+-----+-----+
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] READ: 8B
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 69 69 69 69 69 00 00 00          |iiii...|
+-----+-----+-----+-----+
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] READ: 8B
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 6a 6a 6a 6a 00 00 00 00          |jjjj....|
+-----+-----+-----+-----+
12:07:00 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0xd739f137, L:/192.168.0.103:9090 - R:/192.168.0.103:53155] READ COMPLETE

```

缺点是，数据包的大小不好把握

- 长度定的太大，浪费
- 长度定的太小，对某些数据包又显得不够

方法3，固定分隔符

服务端加入，默认以 `\n` 或 `\r\n` 作为分隔符，如果超出指定长度仍未出现分隔符，则抛出异常

```
//这个1024的意思是如果超过这个长度还没有找到需要的字符就会失败报错
ch.pipeline().addLast(new LineBasedFrameDecoder(1024));
```

客户端在每条消息之后，加入 \n 分隔符

```
public class HelloWorldClient {
    static final Logger log = LoggerFactory.getLogger(HelloWorldClient.class);

    public static void main(String[] args) {
        NioEventLoopGroup worker = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.group(worker);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    log.debug("connected...");
                    ch.pipeline().addLast(new LoggingHandler(LogLevel.DEBUG));
                    ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
                        @Override
                        public void channelActive(ChannelHandlerContext ctx)
                            throws Exception {
                            log.debug("sending...");
                            Random r = new Random();
                            char c = 'a';
                            ByteBuffer buffer = ctx.alloc().buffer();
                            for (int i = 0; i < 10; i++) {
                                for (int j = 1; j <= r.nextInt(16)+1; j++) {
                                    buffer.writeByte((byte) c);
                                }
                                buffer.writeByte(10);
                                c++;
                            }
                            ctx.writeAndFlush(buffer);
                        }
                    });
                }
            });
            ChannelFuture channelFuture = bootstrap.connect("192.168.0.103",
2090).sync();
            channelFuture.channel().closeFuture().sync();

            } catch (InterruptedException e) {
                log.error("client error", e);
            } finally {
                worker.shutdownGracefully();
            }
        }
    }
}
```

客户端输出

```

14:08:18 [DEBUG] [nioEventLoopGroup-2-1] c.i.n.HelloWorldClient - connetted...
14:08:18 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0x1282d755] REGISTERED
14:08:18 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0x1282d755] CONNECT: /192.168.0.103:9090
14:08:18 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0x1282d755, L:/192.168.0.103:63641 - R:/192.168.0.103:9090] ACTIVE
14:08:18 [DEBUG] [nioEventLoopGroup-2-1] c.i.n.HelloWorldClient - sending...
14:08:18 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0x1282d755, L:/192.168.0.103:63641 - R:/192.168.0.103:9090] WRITE: 60B
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 61 0a 62 62 62 0a 63 63 63 0a 64 64 0a 65 65 65 |a.bbb.ccc.dd.eee|
|00000010| 65 65 65 65 65 65 65 0a 66 66 0a 67 67 67 67 67 |eeeeeee.ff.ggggg|
|00000020| 67 67 0a 68 68 68 68 0a 69 69 69 69 69 69 0a |gg.hhhh.iiiiii.|
|00000030| 6a 6a 6a 6a 6a 6a 6a 6a 6a 6a 6a 0a          |jjjjjjjjjjj.  |
+-----+
14:08:18 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0x1282d755, L:/192.168.0.103:63641 - R:/192.168.0.103:9090] FLUSH

```

服务端输出

```

14:08:18 [DEBUG] [nioEventLoopGroup-3-5] c.i.n.HelloWorldServer - connected [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641]
14:08:18 [DEBUG] [nioEventLoopGroup-3-5] i.n.h.l.LoggingHandler - [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641] READ: 1B
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 61                                     |a          |
+-----+
14:08:18 [DEBUG] [nioEventLoopGroup-3-5] i.n.h.l.LoggingHandler - [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641] READ: 3B
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 62 62 62                                     |bbb          |
+-----+
14:08:18 [DEBUG] [nioEventLoopGroup-3-5] i.n.h.l.LoggingHandler - [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641] READ: 3B
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 63 63 63                                     |ccc          |
+-----+
14:08:18 [DEBUG] [nioEventLoopGroup-3-5] i.n.h.l.LoggingHandler - [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641] READ: 2B
      +-----+

```

```

      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 64 64                                     |dd          |
+-----+-----+-----+-----+
14:08:18 [DEBUG] [nioEventLoopGroup-3-5] i.n.h.l.LoggingHandler - [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641] READ: 10B
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 65 65 65 65 65 65 65 65 65 65          |eeeeeeeeee |
+-----+-----+-----+-----+
14:08:18 [DEBUG] [nioEventLoopGroup-3-5] i.n.h.l.LoggingHandler - [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641] READ: 2B
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 66 66                                     |ff          |
+-----+-----+-----+-----+
14:08:18 [DEBUG] [nioEventLoopGroup-3-5] i.n.h.l.LoggingHandler - [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641] READ: 7B
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 67 67 67 67 67 67 67                    |gggggggg   |
+-----+-----+-----+-----+
14:08:18 [DEBUG] [nioEventLoopGroup-3-5] i.n.h.l.LoggingHandler - [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641] READ: 4B
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 68 68 68 68                               |hhhh       |
+-----+-----+-----+-----+
14:08:18 [DEBUG] [nioEventLoopGroup-3-5] i.n.h.l.LoggingHandler - [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641] READ: 7B
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 69 69 69 69 69 69 69                    |iiiiiii   |
+-----+-----+-----+-----+
14:08:18 [DEBUG] [nioEventLoopGroup-3-5] i.n.h.l.LoggingHandler - [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641] READ: 11B
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 6a 6a 6a 6a 6a 6a 6a 6a 6a 6a 6a        |jjjjjjjjjj |
+-----+-----+-----+-----+
14:08:18 [DEBUG] [nioEventLoopGroup-3-5] i.n.h.l.LoggingHandler - [id:
0xa4b3be43, L:/192.168.0.103:9090 - R:/192.168.0.103:63641] READ COMPLETE

```

缺点，处理字符数据比较合适，但如果内容本身包含了分隔符（字节数据常常会有此情况），那么就会解析错误

方法4，预设长度

在发送消息前，先约定用定长字节表示接下来数据的长度

在服务端加上

```
// 最大长度，长度偏移，长度占用字节，长度调整，剥离字节数
ch.pipeline().addLast(new LengthFieldBasedFrameDecoder(1024, 0, 1, 0, 1));
```

客户端代码

```
public class HelloWorldClient {
    static final Logger log = LoggerFactory.getLogger(HelloWorldClient.class);

    public static void main(String[] args) {
        NioEventLoopGroup worker = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.group(worker);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    log.debug("connected...");
                    ch.pipeline().addLast(new LoggingHandler(LogLevel.DEBUG));
                    ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
                        @Override
                        public void channelActive(ChannelHandlerContext ctx)
                            throws Exception {
                            log.debug("sending...");
                            Random r = new Random();
                            char c = 'a';
                            ByteBuffer buffer = ctx.alloc().buffer();
                            for (int i = 0; i < 10; i++) {
                                byte length = (byte) (r.nextInt(16) + 1);
                                // 先写入长度
                                buffer.writeByte(length);
                                // 再
                                for (int j = 1; j <= length; j++) {
                                    buffer.writeByte((byte) c);
                                }
                                c++;
                            }
                            ctx.writeAndFlush(buffer);
                        }
                    });
                }
            });
        } catch {
            // ...
        }
        ChannelFuture channelFuture = bootstrap.connect("192.168.0.103",
            9090).sync();
    }
}
```

```

        channelFuture.channel().closeFuture().sync();

    } catch (InterruptedException e) {
        log.error("client error", e);
    } finally {
        worker.shutdownGracefully();
    }
}
}

```

客户端输出

```

14:37:10 [DEBUG] [nioEventLoopGroup-2-1] c.i.n.HelloWorldClient - connetted...
14:37:10 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0xf0f347b8] REGISTERED
14:37:10 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0xf0f347b8] CONNECT: /192.168.0.103:9090
14:37:10 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0xf0f347b8, L:/192.168.0.103:49979 - R:/192.168.0.103:9090] ACTIVE
14:37:10 [DEBUG] [nioEventLoopGroup-2-1] c.i.n.HelloWorldClient - sending...
14:37:10 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0xf0f347b8, L:/192.168.0.103:49979 - R:/192.168.0.103:9090] WRITE: 97B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
+-----+-----+-----+-----+-----+
|00000000| 09 61 61 61 61 61 61 61 61 61 09 62 62 62 62 62 |.aaaaaaaa.bbbbb|
|00000010| 62 62 62 62 06 63 63 63 63 63 63 08 64 64 64 64 |bbbb.cccccc.dddd|
|00000020| 64 64 64 64 0f 65 65 65 65 65 65 65 65 65 65 65 |dddd.eeeeeeeeeee|
|00000030| 65 65 65 65 0d 66 66 66 66 66 66 66 66 66 66 66 |eeee.ffffffffffff|
|00000040| 66 66 02 67 67 02 68 68 0e 69 69 69 69 69 69 69 |ff.gg.hh.iiiiiii|
|00000050| 69 69 69 69 69 69 69 09 6a 6a 6a 6a 6a 6a 6a 6a |iiiiiii.jjjjjjjj|
|00000060| 6a                                     |j                                     |
+-----+-----+-----+-----+-----+
14:37:10 [DEBUG] [nioEventLoopGroup-2-1] i.n.h.l.LoggingHandler - [id:
0xf0f347b8, L:/192.168.0.103:49979 - R:/192.168.0.103:9090] FLUSH

```

服务端输出

```

14:36:50 [DEBUG] [main] c.i.n.HelloWorldServer - [id: 0xdff439d3] binding...
14:36:51 [DEBUG] [main] c.i.n.HelloWorldServer - [id: 0xdff439d3,
L:/192.168.0.103:9090] bound...
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] REGISTERED
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] ACTIVE
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] c.i.n.HelloWorldServer - connected [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979]

```

```

14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] READ: 9B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 61 61 61 61 61 61 61 61 61 |aaaaaaaaa|
+-----+
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] READ: 9B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 62 62 62 62 62 62 62 62 62 |bbbbbbbbb|
+-----+
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] READ: 6B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 63 63 63 63 63 63 |ccccccc|
+-----+
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] READ: 8B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 64 64 64 64 64 64 64 64 |ddddddd|
+-----+
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] READ: 15B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 65 65 65 65 65 65 65 65 65 65 65 65 65 65 |eeeeeeeeeeeeeee|
+-----+
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] READ: 13B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 66 66 66 66 66 66 66 66 66 66 66 66 66 |fffffffffffffff|
+-----+
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] READ: 2B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 67 67 |gg|
+-----+
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] READ: 2B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 68 68 |hh|

```

```

+-----+-----+-----+-----+
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] READ: 14B
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 69 69 69 69 69 69 69 69 69 69 69 69 69 69 |iiiiiiiiiiiiiii|
+-----+-----+-----+-----+
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] READ: 9B
      +-----+
      | 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+-----+-----+-----+
|00000000| 6a 6a 6a 6a 6a 6a 6a 6a 6a |jjjjjjjjj|
+-----+-----+-----+-----+
14:37:10 [DEBUG] [nioEventLoopGroup-3-1] i.n.h.l.LoggingHandler - [id:
0x744f2b47, L:/192.168.0.103:9090 - R:/192.168.0.103:49979] READ COMPLETE

```

2. 协议设计与解析

2.1 为什么需要协议？

TCP/IP 中消息传输基于流的方式，没有边界。

协议的目的就是划定消息的边界，制定通信双方要共同遵守的通信规则

例如：在网络上传输

下雨天留客天留我不留

是中文一句著名的无标点符号句子，在没有标点符号情况下，这句话有数种拆解方式，而意思却是完全不同，所以常被用作讲述标点符号的重要性

一种解读

下雨天留客，天留，我不留

另一种解读

下雨天，留客天，留我不？留

如何设计协议呢？其实就是给网络传输的信息加上“标点符号”。但通过分隔符来断句不是很好，因为分隔符本身如果用于传输，那么必须加以区分。因此，下面一种协议较为常用

定长字节表示内容长度 + 实际内容

例如，假设一个中文字符长度为 3，按照上述协议的规则，发送信息方式如下，就不会被接收方弄错意思了

0f下雨天留客06天留09我不留

小故事

很久很久以前，一位私塾先生到一家任教。双方签订了一纸协议：“无鸡鸭亦可无鱼肉亦可白菜豆腐不可少不得束修金”。此后，私塾先生虽然认真教课，但主人家则总是给私塾先生以白菜豆腐为菜，丝毫未见鸡鸭鱼肉的款待。私塾先生先是很不解，可是后来也就想通了：主人把鸡鸭鱼肉的款都会换为束修金的，也罢。至此双方相安无事。

年关将至，一个学年段亦告结束。私塾先生临行时，也不见主人家为他交付束修金，遂与主家理论。然主家亦振振有词：“有协议为证——无鸡鸭亦可，无鱼肉亦可，白菜豆腐不可少，不得束修金。这白纸黑字明摆着的，你有什么要说的呢？”

私塾先生据理力争：“协议是这样的——无鸡，鸭亦可；无鱼，肉亦可；白菜豆腐不可，少不得束修金。”

双方唇枪舌战，你来我往，真个是不亦乐乎！

这里的束修金，也作“束脩”，应当是泛指教师应当得到的报酬

2.2 redis 协议举例

```
NioEventLoopGroup worker = new NioEventLoopGroup();
byte[] LINE = {13, 10};
try {
    Bootstrap bootstrap = new Bootstrap();
    bootstrap.channel(NioSocketChannel.class);
    bootstrap.group(worker);
    bootstrap.handler(new ChannelInitializer<SocketChannel>() {
        @Override
        protected void initChannel(SocketChannel ch) {
            ch.pipeline().addLast(new LoggingHandler());
            ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
                // 会在连接 channel 建立成功后, 会触发 active 事件
                @Override
                public void channelActive(ChannelHandlerContext ctx) {
                    set(ctx);
                    get(ctx);
                }
                private void get(ChannelHandlerContext ctx) {
                    ByteBuf buf = ctx.alloc().buffer();
                    buf.writeBytes("*2".getBytes());
                    buf.writeBytes(LINE);
                    buf.writeBytes("$3".getBytes());
                    buf.writeBytes(LINE);
                    buf.writeBytes("get".getBytes());
                    buf.writeBytes(LINE);
                    buf.writeBytes("$3".getBytes());
                    buf.writeBytes(LINE);
                    buf.writeBytes("aaa".getBytes());
                    buf.writeBytes(LINE);
                    ctx.writeAndFlush(buf);
                }
                private void set(ChannelHandlerContext ctx) {
                    ByteBuf buf = ctx.alloc().buffer();
                    buf.writeBytes("*3".getBytes());
                    buf.writeBytes(LINE);
                    buf.writeBytes("$3".getBytes());
                    buf.writeBytes(LINE);
                    buf.writeBytes("set".getBytes());
                    buf.writeBytes(LINE);
                    buf.writeBytes("$3".getBytes());
                    buf.writeBytes(LINE);
                    buf.writeBytes("aaa".getBytes());
                    buf.writeBytes(LINE);
                    buf.writeBytes("$3".getBytes());
                    buf.writeBytes(LINE);
                    buf.writeBytes("bbb".getBytes());
                    buf.writeBytes(LINE);
                    ctx.writeAndFlush(buf);
                }
            });
        }
    });
}
```

@Override

```

        public void channelRead(ChannelHandlerContext ctx, Object msg)
        throws Exception {
            ByteBuf buf = (ByteBuf) msg;
            System.out.println(buf.toString(Charset.defaultCharset()));
        }
    });
}

});
ChannelFuture channelFuture = bootstrap.connect("localhost", 6379).sync();
channelFuture.channel().closeFuture().sync();
} catch (InterruptedException e) {
    log.error("client error", e);
} finally {
    worker.shutdownGracefully();
}
}

```

2.3 http 协议举例

HttpServerCodec既是入站处理器，又是出站

```

package netty.c5;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.http.DefaultFullHttpResponse;
import io.netty.handler.codec.http.HttpRequest;
import io.netty.handler.codec.http.HttpResponseStatus;
import io.netty.handler.codec.http.HttpServerCodec;
import io.netty.handler.logging.LogLevel;
import io.netty.handler.logging.LoggingHandler;
import lombok.extern.slf4j.Slf4j;

import java.nio.charset.StandardCharsets;

import static io.netty.handler.codec.http.HttpHeaderNames.CONTENT_LENGTH;

@Slf4j
public class TestHttp {
    public static void main(String[] args) {
        NioEventLoopGroup boss = new NioEventLoopGroup();
        NioEventLoopGroup worker = new NioEventLoopGroup();
        try {
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.group(boss, worker);
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>()
{

```

```

@Override
protected void initChannel(SocketChannel ch) throws Exception {
    ch.pipeline().addLast(new LoggingHandler(LogLevel.DEBUG));
    ch.pipeline().addLast(new HttpServerCodec());
    //用这个Handler可以固定的处理泛型中类型的数据，就跟下面写的那个Handler
    一样的 if (msg instanceof HttpRequest)
        ch.pipeline().addLast(new
SimpleChannelInboundHandler<HttpRequest>() {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx,
HttpRequest msg) throws Exception {
        // 获取请求
        log.debug(msg.uri());

        // 返回响应
        DefaultFullHttpResponse response =
            new
DefaultFullHttpResponse(msg.protocolVersion(), HttpResponseStatus.OK);
        byte[] bytes = "<h1>Hello, world!</h1>".getBytes();
        //设置响应的长度，要不然浏览器会一直等待服务器继续发送响应

        response.headers().setInt(CONTENT_LENGTH, bytes.length);
        //向响应里面写入内容
        response.content().writeBytes(bytes);
        // 写回响应(写出仍然会经过HttpServerCodec处理器)
        ctx.writeAndFlush(response);
    }
});
/*ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {
        log.debug("{} ", msg.getClass());
        if (msg instanceof HttpRequest) { // 请求行，请求头
        } else if (msg instanceof HttpContent) { //请求体

        }
    }
});*/
}
});
ChannelFuture channelFuture = serverBootstrap.bind(8080).sync();
channelFuture.channel().closeFuture().sync();
} catch (InterruptedException e) {
    log.error("server error", e);
} finally {
    boss.shutdownGracefully();
    worker.shutdownGracefully();
}
}
}

```

2.4 自定义协议要素

- 魔数，用来在第一时间判定是否是无效数据包
- 版本号，可以支持协议的升级
- 序列化算法，消息正文到底采用哪种序列化反序列化方式，可以由此扩展，例如：json、protobuf、hessian、jdk
- 指令类型，是登录、注册、单聊、群聊... 跟业务相关
- 请求序号，为了双工通信，提供异步能力
- 正文长度
- 消息正文

编解码器

根据上面的要素，设计一个登录请求消息和登录响应消息，并使用 Netty 完成收发

```
@Slf4j
public class MessageCodec extends ByteToMessageCodec<Message> {

    @Override
    protected void encode(ChannelHandlerContext ctx, Message msg, ByteBuf out)
        throws Exception {
        // 1. 4 字节的魔数
        out.writeBytes(new byte[]{1, 2, 3, 4});
        // 2. 1 字节的版本，
        out.writeByte(1);
        // 3. 1 字节的序列化方式 jdk 0 , json 1
        out.writeByte(0);
        // 4. 1 字节的指令类型
        out.writeByte(msg.getMessageType());
        // 5. 4 个字节
        out.writeInt(msg.getSequenceId());
        // 无意义，对齐填充
        out.writeByte(0xff);
        // 6. 获取内容的字节数组
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(msg);
        byte[] bytes = bos.toByteArray();
        // 7. 长度
        out.writeInt(bytes.length);
        // 8. 写入内容
        out.writeBytes(bytes);
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
        //对照着编码进行解码就可以了
    }
}
```

```

        int magicNum = in.readInt();
        byte version = in.readByte();
        byte serializerType = in.readByte();
        byte messageType = in.readByte();
        int sequenceId = in.readInt();
        in.readByte();
        int length = in.readInt();
        byte[] bytes = new byte[length];
        in.readBytes(bytes, 0, length);
        ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(bytes));
        Message message = (Message) ois.readObject();
        log.debug("{} {} {} {} {} {}", magicNum, version, serializerType,
messageType, sequenceId, length);
        log.debug("{} ", message);
        out.add(message);
    }
}

```

测试

```

EmbeddedChannel channel = new EmbeddedChannel(
    new LoggingHandler(),
    //记得要加这个帧拦截器，要不然可能会出现粘包半包的现象
    new LengthFieldBasedFrameDecoder(
        1024, 12, 4, 0, 0),
    new MessageCodec()
);
// encode
LoginRequestMessage message = new LoginRequestMessage("zhangsan", "123", "张三");
//      channel.writeOutbound(message);
// decode
ByteBuf buf = ByteBufAllocator.DEFAULT.buffer();
new MessageCodec().encode(null, message, buf);

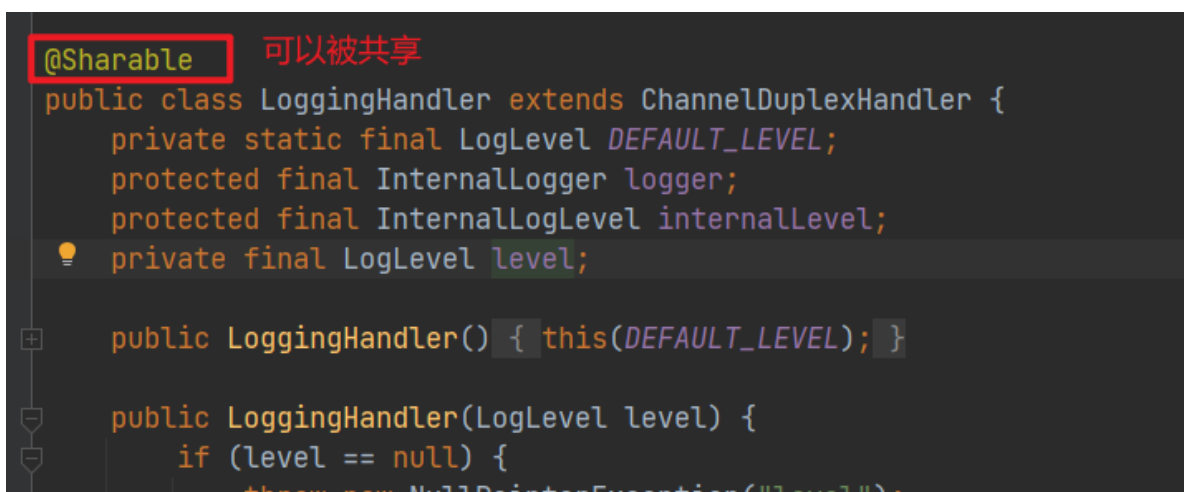
ByteBuf s1 = buf.slice(0, 100);
ByteBuf s2 = buf.slice(100, buf.readableBytes() - 100);
s1.retain(); // 引用计数 2
channel.writeInbound(s1); // release 1
channel.writeInbound(s2);

```

解读

魔数	版本		序列化算法	消息类型	消息序号	消息正文长度	消息正文										
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000	12	34	56	78	01	00	00	00	00	00	02	ff	00	00	00	db	.4Vx.....
00000010	ac	ed	00	05	73	72	00	2c	63	6f	6d	2e	69	74	63	61sr.,com.itca
00000020	73	74	2e	6e	65	74	74	79	2e	70	6f	74	6f	63	6f	6c	st.netty.potocol
00000030	2e	4c	6f	67	69	6e	52	65	71	75	65	73	74	4d	65	73	.LoginRequestMes
00000040	73	61	67	65	aa	d6	eb	c4	4b	8c	ce	be	02	00	03	4c	sage....K.....L
00000050	00	08	6e	69	63	6b	6e	61	6d	65	74	00	12	4c	6a	61	..nicknamet..Lja
00000060	76	61	2f	6c	61	6e	67	2f	53	74	72	69	6e	67	3b	4c	va/lang/String;L
00000070	00	08	70	61	73	73	77	6f	72	64	71	00	7e	00	01	4c	..passwordq~..L
00000080	00	08	75	73	65	72	6e	61	6d	65	71	00	7e	00	01	78	..usernameq~..x
00000090	72	00	20	63	6f	6d	2e	69	74	63	61	73	74	2e	6e	65	r. com.itcast.ne
000000a0	74	74	79	2e	70	6f	74	6f	63	6f	6c	2e	4d	65	73	73	tty.potocol.Mess
000000b0	61	67	65	e3	8d	05	6d	af	c7	fd	1d	02	00	01	49	00	age...m.....I.
000000c0	0a	73	65	71	75	65	6e	63	65	49	64	78	70	00	00	00	.sequenceIdxp...
000000d0	02	74	00	06	e5	b0	8f	e5	bc	a0	74	00	03	31	32	33	.t.....t..123
000000e0	74	00	08	7a	68	61	6e	67	73	61	6e						t..zhangsan

💡 什么时候可以加 `@Sharable`



- 当 handler 不保存状态时，就可以安全地在多线程下被共享
- 但要注意对于编解码器类（都不能共享，如果共享了可能会有安全隐患），不能继承 ByteToMessageCodec 或 CombinedChannelDuplexHandler 父类，他们的构造方法对 `@Sharable` 有限制
- 如果能确保编解码器不会保存状态，可以继承 MessageToMessageCodec 父类

```

@Slf4j
@ChannelHandler.Sharable
/**
 * 必须和 LengthFieldBasedFrameDecoder 一起使用，确保接到的 ByteBuf 消息是完整的
 */
public class MessageCodecSharable extends MessageToMessageCodec<ByteBuf,
Message> {

```

```

@Override
protected void encode(ChannelHandlerContext ctx, Message msg, List<Object>
outList) throws Exception {
    ByteBuf out = ctx.alloc().buffer();
    // 1. 4 字节的魔数
    out.writeBytes(new byte[]{1, 2, 3, 4});
    // 2. 1 字节的版本,
    out.writeByte(1);
    // 3. 1 字节的序列化方式 jdk 0 , json 1
    out.writeByte(0);
    // 4. 1 字节的指令类型
    out.writeByte(msg.getMessageType());
    // 5. 4 个字节
    out.writeInt(msg.getSequenceId());
    // 无意义, 对齐填充
    out.writeByte(0xff);
    // 6. 获取内容的字节数组
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(bos);
    oos.writeObject(msg);
    byte[] bytes = bos.toByteArray();
    // 7. 长度
    out.writeInt(bytes.length);
    // 8. 写入内容
    out.writeBytes(bytes);
    outList.add(out);
}

@Override
protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
    int magicNum = in.readInt();
    byte version = in.readByte();
    byte serializerType = in.readByte();
    byte messageType = in.readByte();
    int sequenceId = in.readInt();
    in.readByte();
    int length = in.readInt();
    byte[] bytes = new byte[length];
    in.readBytes(bytes, 0, length);
    ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(bytes));
    Message message = (Message) ois.readObject();
    log.debug("{} , {} , {} , {} , {} , {}", magicNum, version, serializerType,
messageType, sequenceId, length);
    log.debug("{} ", message);
    out.add(message);
}
}

```

3. 聊天室案例

3.1 聊天室业务介绍

```
/**
 * 用户管理接口
 */
public interface UserService {

    /**
     * 登录
     * @param username 用户名
     * @param password 密码
     * @return 登录成功返回 true, 否则返回 false
     */
    boolean login(String username, String password);
}
```

```
/**
 * 会话管理接口
 */
public interface Session {

    /**
     * 绑定会话
     * @param channel 哪个 channel 要绑定会话
     * @param username 会话绑定用户
     */
    void bind(Channel channel, String username);

    /**
     * 解绑会话
     * @param channel 哪个 channel 要解绑会话
     */
    void unbind(Channel channel);

    /**
     * 获取属性
     * @param channel 哪个 channel
     * @param name 属性名
     * @return 属性值
     */
    Object getAttribute(Channel channel, String name);

    /**
     * 设置属性
     * @param channel 哪个 channel
     * @param name 属性名
     * @param value 属性值
     */
    void setAttribute(Channel channel, String name, Object value);
}
```

```
/**
 * 根据用户名获取 channel
 * @param username 用户名
 * @return channel
 */
Channel getChannel(String username);
}
```

```
/**
 * 聊天组会话管理接口
 */
public interface GroupSession {

    /**
     * 创建一个聊天组，如果不存在才能创建成功，否则返回 null
     * @param name 组名
     * @param members 成员
     * @return 成功时返回组对象，失败返回 null
     */
    Group createGroup(String name, Set<String> members);

    /**
     * 加入聊天组
     * @param name 组名
     * @param member 成员名
     * @return 如果组不存在返回 null，否则返回组对象
     */
    Group joinMember(String name, String member);

    /**
     * 移除组成员
     * @param name 组名
     * @param member 成员名
     * @return 如果组不存在返回 null，否则返回组对象
     */
    Group removeMember(String name, String member);

    /**
     * 移除聊天组
     * @param name 组名
     * @return 如果组不存在返回 null，否则返回组对象
     */
    Group removeGroup(String name);

    /**
     * 获取组成员
     * @param name 组名
     * @return 成员集合，没有成员会返回 empty set
     */
    Set<String> getMembers(String name);

    /**
     * 获取组成员的 channel 集合，只有在线的 channel 才会返回
     */
}
```

```

    * @param name 组名
    * @return 成员 channel 集合
    */
    List<Channel> getMembersChannel(String name);
}

```

3.2 聊天室业务-登录

```

@Slf4j
public class ChatServer {
    public static void main(String[] args) {
        NioEventLoopGroup boss = new NioEventLoopGroup();
        NioEventLoopGroup worker = new NioEventLoopGroup();
        LoggingHandler LOGGING_HANDLER = new LoggingHandler(LogLevel.DEBUG);
        MessageCodecSharable MESSAGE_CODEC = new MessageCodecSharable();
        try {
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.group(boss, worker);
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>()
{
            @Override
            protected void initChannel(SocketChannel ch) throws Exception {
                ch.pipeline().addLast(new ProtocolFrameDecoder());
                ch.pipeline().addLast(LOGGING_HANDLER);
                ch.pipeline().addLast(MESSAGE_CODEC);
                ch.pipeline().addLast(new
SimpleChannelInboundHandler<LoginRequestMessage>() {
                    @Override
                    protected void channelRead0(ChannelHandlerContext ctx,
LoginRequestMessage msg) throws Exception {
                        String username = msg.getUsername();
                        String password = msg.getPassword();
                        boolean login =
UserServiceFactory.getUserService().login(username, password);
                        LoginResponseMessage message;
                        if(login) {
                            message = new LoginResponseMessage(true, "登录成
功");
                        } else {
                            message = new LoginResponseMessage(false, "用户名
或密码不正确");
                        }
                        ctx.writeAndFlush(message);
                    }
                });
            }
        });
        Channel channel = serverBootstrap.bind(8080).sync().channel();
        channel.closeFuture().sync();
    } catch (InterruptedException e) {
}

```

```
log.error("server error", e);
} finally {
    boss.shutdownGracefully();
    worker.shutdownGracefully();
}
}
}
```

```
@Slf4j
public class ChatClient {
    public static void main(String[] args) {
        NioEventLoopGroup group = new NioEventLoopGroup();
        LoggingHandler LOGGING_HANDLER = new LoggingHandler(LogLevel.DEBUG);
        MessageCodecSharable MESSAGE_CODEC = new MessageCodecSharable();
        CountDownLatch WAIT_FOR_LOGIN = new CountDownLatch(1);
        AtomicBoolean LOGIN = new AtomicBoolean(false);
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.group(group);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new ProtocolFrameDecoder());
                    // ch.pipeline().addLast(LOGGING_HANDLER);
                    ch.pipeline().addLast(MESSAGE_CODEC);
                    ch.pipeline().addLast("client handler", new
ChannelInboundHandlerAdapter() {
                        // 接收响应消息
                        @Override
                        public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
                            log.debug("msg: {}", msg);
                            if ((msg instanceof LoginResponseMessage)) {
                                LoginResponseMessage response =
(LoginResponseMessage) msg;
                                if (response.isSuccess()) {
                                    // 如果登录成功
                                    LOGIN.set(true);
                                }
                                // 唤醒 system in 线程
                                WAIT_FOR_LOGIN.countDown();
                            }
                        }
                    })
                }

                // 在连接建立后触发 active 事件
                @Override
                public void channelActive(ChannelHandlerContext ctx)
throws Exception {
                    // 负责接收用户在控制台的输入，负责向服务器发送各种消息
                    new Thread(() -> {
                        Scanner scanner = new Scanner(System.in);
                        System.out.println("请输入用户名:");
                    })
                }
            })
        }
    }
}
```

```

        String username = scanner.nextLine();
        System.out.println("请输入密码:");
        String password = scanner.nextLine();
        // 构造消息对象
        LoginRequestMessage message = new
LoginRequestMessage(username, password);
        // 发送消息
        ctx.writeAndFlush(message);
        System.out.println("等待后续操作...");
        try {
            WAIT_FOR_LOGIN.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 如果登录失败
        if (!LOGIN.get()) {
            ctx.channel().close();
            return;
        }
        while (true) {

            System.out.println("=====");
            System.out.println("send [username]
[content]");
            System.out.println("gsend [group name]
[content]");
            System.out.println("gcreate [group name]
[m1,m2,m3...]");

            System.out.println("gmembers [group name]");
            System.out.println("gjoin [group name]");
            System.out.println("gquit [group name]");
            System.out.println("quit");

            System.out.println("=====");
            String command = scanner.nextLine();
            String[] s = command.split(" ");
            switch (s[0]){
                case "send":
                    ctx.writeAndFlush(new
ChatRequestMessage(username, s[1], s[2]));
                    break;
                case "gsend":
                    ctx.writeAndFlush(new
GroupChatRequestMessage(username, s[1], s[2]));
                    break;
                case "gcreate":
                    Set<String> set = new HashSet<>
(Arrays.asList(s[2].split(",")));
                    set.add(username); // 加入自己
                    ctx.writeAndFlush(new
GroupCreateRequestMessage(s[1], set));
                    break;
                case "gmembers":
                    ctx.writeAndFlush(new
GroupMembersRequestMessage(s[1]));

```

```

                break;
            case "gjoin":
                ctx.writeAndFlush(new
GroupJoinRequestMessage(username, s[1]));
                break;
            case "gquit":
                ctx.writeAndFlush(new
GroupQuitRequestMessage(username, s[1]));
                break;
            case "quit":
                ctx.channel().close();
                return;
        }
    }
    }, "system in").start();
}
});
}
});
channel channel = bootstrap.connect("localhost",
8080).sync().channel();
channel.closeFuture().sync();
} catch (Exception e) {
    log.error("client error", e);
} finally {
    group.shutdownGracefully();
}
}
}

```

3.3 聊天室业务-单聊

服务器端将 handler 独立出来

登录 handler

```

@ChannelHandler.Sharable
public class LoginRequestMessageHandler extends
SimpleChannelInboundHandler<LoginRequestMessage> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, LoginRequestMessage
msg) throws Exception {
        String username = msg.getUsername();
        String password = msg.getPassword();
        boolean login = UserServiceFactory.getUserService().login(username,
password);
        LoginResponseMessage message;
        if(login) {

```

```

        SessionFactory.getSession().bind(ctx.channel(), username);
        message = new LoginResponseMessage(true, "登录成功");
    } else {
        message = new LoginResponseMessage(false, "用户名或密码不正确");
    }
    ctx.writeAndFlush(message);
}
}

```

单聊 handler

```

@ChannelHandler.Sharable
public class ChatRequestMessageHandler extends
SimpleChannelInboundHandler<ChatRequestMessage> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ChatRequestMessage
msg) throws Exception {
        String to = msg.getTo();
        Channel channel = SessionFactory.getSession().getChannel(to);
        // 在线
        if(channel != null) {
            channel.writeAndFlush(new ChatResponseMessage(msg.getFrom(),
msg.getContent()));
        }
        // 不在线
        else {
            ctx.writeAndFlush(new ChatResponseMessage(false, "对方用户不存在或者不在
线"));
        }
    }
}

```

3.4 聊天室业务-群聊

创建群聊

```

@ChannelHandler.Sharable
public class GroupCreateRequestMessageHandler extends
SimpleChannelInboundHandler<GroupCreateRequestMessage> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx,
GroupCreateRequestMessage msg) throws Exception {
        String groupName = msg.getGroupName();
        Set<String> members = msg.getMembers();
        // 群管理器
        GroupSession groupSession = GroupSessionFactory.getGroupSession();
    }
}

```

```

        Group group = groupSession.createGroup(groupName, members);
        if (group == null) {
            // 发生成功消息
            ctx.writeAndFlush(new GroupCreateResponseMessage(true, groupName +
"创建成功"));
            // 发送拉群消息
            List<Channel> channels = groupSession.getMembersChannel(groupName);
            for (Channel channel : channels) {
                channel.writeAndFlush(new GroupCreateResponseMessage(true, "您已被
拉入" + groupName));
            }
        } else {
            ctx.writeAndFlush(new GroupCreateResponseMessage(false, groupName +
"已经存在"));
        }
    }
}

```

群聊

```

@ChannelHandler.Sharable
public class GroupChatRequestMessageHandler extends
SimpleChannelInboundHandler<GroupChatRequestMessage> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx,
GroupChatRequestMessage msg) throws Exception {
        List<Channel> channels = GroupSessionFactory.getGroupSession()
            .getMembersChannel(msg.getGroupName());

        for (Channel channel : channels) {
            channel.writeAndFlush(new GroupChatResponseMessage(msg.getFrom(),
msg.getContent()));
        }
    }
}

```

加入群聊


```

@ChannelHandler.Sharable
public class GroupJoinRequestMessageHandler extends
SimpleChannelInboundHandler<GroupJoinRequestMessage> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx,
GroupJoinRequestMessage msg) throws Exception {
        Group group =
GroupSessionFactory.getGroupSession().joinMember(msg.getGroupName(),
msg.getUsername());
        if (group != null) {
            ctx.writeAndFlush(new GroupJoinResponseMessage(true,
msg.getGroupName() + "群加入成功"));
        } else {
            ctx.writeAndFlush(new GroupJoinResponseMessage(true,
msg.getGroupName() + "群不存在"));
        }
    }
}

```

退出群聊

```

@ChannelHandler.Sharable
public class GroupQuitRequestMessageHandler extends
SimpleChannelInboundHandler<GroupQuitRequestMessage> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx,
GroupQuitRequestMessage msg) throws Exception {
        Group group =
GroupSessionFactory.getGroupSession().removeMember(msg.getGroupName(),
msg.getUsername());
        if (group != null) {
            ctx.writeAndFlush(new GroupJoinResponseMessage(true, "已退出群" +
msg.getGroupName()));
        } else {
            ctx.writeAndFlush(new GroupJoinResponseMessage(true,
msg.getGroupName() + "群不存在"));
        }
    }
}

```

查看成员

```

@ChannelHandler.Sharable
public class GroupMembersRequestMessageHandler extends
SimpleChannelInboundHandler<GroupMembersRequestMessage> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx,
GroupMembersRequestMessage msg) throws Exception {
        Set<String> members = GroupSessionFactory.getGroupSession()
            .getMembers(msg.getGroupName());
        ctx.writeAndFlush(new GroupMembersResponseMessage(members));
    }
}

```

3.5 聊天室业务-退出

```

@Slf4j
@ChannelHandler.Sharable
public class QuitHandler extends ChannelInboundHandlerAdapter {

    // 当连接断开时触发 inactive 事件
    @Override
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
        SessionFactory.getSession().unbind(ctx.channel());
        log.debug("{} 已经断开", ctx.channel());
    }

    // 当出现异常时触发
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        SessionFactory.getSession().unbind(ctx.channel());
        log.debug("{} 已经异常断开 异常是{}", ctx.channel(), cause.getMessage());
    }
}

```

3.6 聊天室业务-空闲检测

连接假死

原因

- 网络设备出现故障，例如网卡，机房等，底层的 TCP 连接已经断开了，但应用程序没有感知到，仍然占用着资源。

- 公网网络不稳定，出现丢包。如果连续出现丢包，这时现象就是客户端数据发不出去，服务端也一直收不到数据，就这么一直耗着
- 应用程序线程阻塞，无法进行数据读写

问题

- 假死的连接占用的资源不能自动释放
- 向假死的连接发送数据，得到的反馈是发送超时

服务器端解决

- 怎么判断客户端连接是否假死呢？如果能收到客户端数据，说明没有假死。因此策略就可以定为，每隔一段时间就检查这段时间内是否接收到客户端数据，没有就可以判定为连接假死

```
// 用来判断是不是 读空闲时间过长，或 写空闲时间过长
// 5s 内如果没有收到 channel 的数据，会触发一个 IdleState#READER_IDLE 事件
ch.pipeline().addLast(new IdleStateHandler(5, 0, 0));
// ChannelDuplexHandler 可以同时作为入站和出站处理器
ch.pipeline().addLast(new ChannelDuplexHandler() {
    // 用来触发特殊事件
    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws
Exception{
        IdleStateEvent event = (IdleStateEvent) evt;
        // 触发了读空闲事件
        if (event.state() == IdleState.READER_IDLE) {
            log.debug("已经 5s 没有读到数据了");
            ctx.channel().close();
        }
    }
});
```

客户端定时心跳

- 客户端可以定时向服务器端发送数据，只要这个时间间隔小于服务器定义的空闲检测的时间间隔，那么就能防止前面提到的误判，客户端可以定义如下心跳处理器

```
// 用来判断是不是 读空闲时间过长，或 写空闲时间过长
// 3s 内如果没有向服务器写数据，会触发一个 IdleState#WRITER_IDLE 事件
ch.pipeline().addLast(new IdleStateHandler(0, 3, 0));
// ChannelDuplexHandler 可以同时作为入站和出站处理器
ch.pipeline().addLast(new ChannelDuplexHandler() {
```

```

        // 用来触发特殊事件
        @Override
        public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws
Exception{
            IdleStateEvent event = (IdleStateEvent) evt;
            // 触发了写空闲事件
            if (event.state() == IdleState.WRITER_IDLE) {
                //
                log.debug("3s 没有写数据了，发送一个心跳包");
                ctx.writeAndFlush(new PingMessage());
            }
        }
    }
});

```

四. 优化与源码

1. 优化

1.1 扩展序列化算法

序列化，反序列化主要用在消息正文的转换上

- 序列化时，需要将 Java 对象变为要传输的数据（可以是 byte[]，或 json 等，最终都需要变成 byte[]）
- 反序列化时，需要将传入的正文数据还原成 Java 对象，便于处理

目前的代码仅支持 Java 自带的序列化，反序列化机制，核心代码如下

```

// 反序列化
byte[] body = new byte[bodyLength];
byteByf.readBytes(body);
ObjectInputStream in = new ObjectInputStream(new ByteArrayInputStream(body));
Message message = (Message) in.readObject();
message.setSequenceId(sequenceId);

// 序列化
ByteArrayOutputStream out = new ByteArrayOutputStream();
new ObjectOutputStream(out).writeObject(message);
byte[] bytes = out.toByteArray();

```

为了支持更多序列化算法，抽象一个 Serializer 接口

```

public interface Serializer {

    // 反序列化方法
    <T> T deserialize(Class<T> clazz, byte[] bytes);

    // 序列化方法
    <T> byte[] serialize(T object);

}

```

提供两个实现，我这里直接将实现加入了枚举类 `Serializer.Algorithm` 中

```

enum SerializerAlgorithm implements Serializer {
    // Java 实现
    Java {
        @Override
        public <T> T deserialize(Class<T> clazz, byte[] bytes) {
            try {
                ObjectInputStream in =
                    new ObjectInputStream(new ByteArrayInputStream(bytes));
                Object object = in.readObject();
                return (T) object;
            } catch (IOException | ClassNotFoundException e) {
                throw new RuntimeException("SerializerAlgorithm.Java 反序列化错误",
e);
            }
        }

        @Override
        public <T> byte[] serialize(T object) {
            try {
                ByteArrayOutputStream out = new ByteArrayOutputStream();
                new ObjectOutputStream(out).writeObject(object);
                return out.toByteArray();
            } catch (IOException e) {
                throw new RuntimeException("SerializerAlgorithm.Java 序列化错误",
e);
            }
        }
    },
    // Json 实现(引入了 Gson 依赖)
    Json {
        @Override
        public <T> T deserialize(Class<T> clazz, byte[] bytes) {
            return new Gson().fromJson(new String(bytes,
StandardCharsets.UTF_8), clazz);
        }

        @Override
        public <T> byte[] serialize(T object) {
            return new Gson().toJson(object).getBytes(StandardCharsets.UTF_8);
        }
    }
}

```

```

    }
};

// 需要从协议的字节中得到是哪种序列化算法
public static SerializerAlgorithm getByInt(int type) {
    SerializerAlgorithm[] array = SerializerAlgorithm.values();
    if (type < 0 || type > array.length - 1) {
        throw new IllegalArgumentException("超过 SerializerAlgorithm 范围");
    }
    return array[type];
}
}

```

增加配置类和配置文件

```

public abstract class Config {
    static Properties properties;
    static {
        try (InputStream in =
Config.class.getResourceAsStream("/application.properties")) {
            properties = new Properties();
            properties.load(in);
        } catch (IOException e) {
            throw new ExceptionInInitializerError(e);
        }
    }
    public static int getServerPort() {
        String value = properties.getProperty("server.port");
        if(value == null) {
            return 8080;
        } else {
            return Integer.parseInt(value);
        }
    }
    public static Serializer.Algorithm getSerializerAlgorithm() {
        String value = properties.getProperty("serializer.algorithm");
        if(value == null) {
            return Serializer.Algorithm.Java;
        } else {
            return Serializer.Algorithm.valueOf(value);
        }
    }
}
}

```

配置文件

```
serializer.algorithm=Json
```

修改编解码器

```
/**
 * 必须和 LengthFieldBasedFrameDecoder 一起使用，确保接到的 ByteBuf 消息是完整的
 */
public class MessageCodecSharable extends MessageToMessageCodec<ByteBuf,
Message> {
    @Override
    public void encode(ChannelHandlerContext ctx, Message msg, List<Object>
outList) throws Exception {
        ByteBuf out = ctx.alloc().buffer();
        // 1. 4 字节的魔数
        out.writeBytes(new byte[]{1, 2, 3, 4});
        // 2. 1 字节的版本，
        out.writeByte(1);
        // 3. 1 字节的序列化方式 jdk 0 , json 1
        out.writeByte(Config.getSerializerAlgorithm().ordinal());
        // 4. 1 字节的指令类型
        out.writeByte(msg.getMessageType());
        // 5. 4 个字节
        out.writeInt(msg.getSequenceId());
        // 无意义，对齐填充
        out.writeByte(0xff);
        // 6. 获取内容的字节数组
        byte[] bytes = Config.getSerializerAlgorithm().serialize(msg);
        // 7. 长度
        out.writeInt(bytes.length);
        // 8. 写入内容
        out.writeBytes(bytes);
        outList.add(out);
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
        int magicNum = in.readInt();
        byte version = in.readByte();
        byte serializerAlgorithm = in.readByte(); // 0 或 1
        byte messageType = in.readByte(); // 0,1,2...
        int sequenceId = in.readInt();
        in.readByte();
        int length = in.readInt();
        byte[] bytes = new byte[length];
        in.readBytes(bytes, 0, length);

        // 找到反序列化算法
        Serializer.Algorithm algorithm = Serializer.Algorithm.values()
[serializerAlgorithm];
        // 确定具体消息类型
        Class<? extends Message> messageClass =
Message.getMessageClass(messageType);
        Message message = algorithm.deserialize(messageClass, bytes);
    }
}
```

```
//      log.debug("{} , {} , {} , {} , {} , {}", magicNum, version, serializerType,
messageType, sequenceId, length);
//      log.debug("{} ", message);
      out.add(message);
  }
}
```

其中确定具体消息类型，可以根据 消息类型字节 获取到对应的 消息 class

```
@Data
public abstract class Message implements Serializable {

    /**
     * 根据消息类型字节，获得对应的消息 class
     * @param messageType 消息类型字节
     * @return 消息 class
     */
    public static Class<? extends Message> getMessageClass(int messageType) {
        return messageClasses.get(messageType);
    }

    private int sequenceId;

    private int messageType;

    public abstract int getMessageType();

    public static final int LoginRequestMessage = 0;
    public static final int LoginResponseMessage = 1;
    public static final int ChatRequestMessage = 2;
    public static final int ChatResponseMessage = 3;
    public static final int GroupCreateRequestMessage = 4;
    public static final int GroupCreateResponseMessage = 5;
    public static final int GroupJoinRequestMessage = 6;
    public static final int GroupJoinResponseMessage = 7;
    public static final int GroupQuitRequestMessage = 8;
    public static final int GroupQuitResponseMessage = 9;
    public static final int GroupChatRequestMessage = 10;
    public static final int GroupChatResponseMessage = 11;
    public static final int GroupMembersRequestMessage = 12;
    public static final int GroupMembersResponseMessage = 13;
    public static final int PingMessage = 14;
    public static final int PongMessage = 15;
    private static final Map<Integer, Class<? extends Message>> messageClasses =
new HashMap<>();

    static {
        messageClasses.put(LoginRequestMessage, LoginRequestMessage.class);
        messageClasses.put(LoginResponseMessage, LoginResponseMessage.class);
        messageClasses.put(ChatRequestMessage, ChatRequestMessage.class);
        messageClasses.put(ChatResponseMessage, ChatResponseMessage.class);
    }
}
```



```

        messageClasses.put(GroupCreateRequestMessage,
GroupCreateRequestMessage.class);
        messageClasses.put(GroupCreateResponseMessage,
GroupCreateResponseMessage.class);
        messageClasses.put(GroupJoinRequestMessage,
GroupJoinRequestMessage.class);
        messageClasses.put(GroupJoinResponseMessage,
GroupJoinResponseMessage.class);
        messageClasses.put(GroupQuitRequestMessage,
GroupQuitRequestMessage.class);
        messageClasses.put(GroupQuitResponseMessage,
GroupQuitResponseMessage.class);
        messageClasses.put(GroupChatRequestMessage,
GroupChatRequestMessage.class);
        messageClasses.put(GroupChatResponseMessage,
GroupChatResponseMessage.class);
        messageClasses.put(GroupMembersRequestMessage,
GroupMembersRequestMessage.class);
        messageClasses.put(GroupMembersResponseMessage,
GroupMembersResponseMessage.class);
    }
}

```

1.2 参数调优

1) CONNECT_TIMEOUT_MILLIS

- 属于 SocketChannel 参数
- 用在客户端建立连接时，如果在指定毫秒内无法连接，会抛出 timeout 异常
- SO_TIMEOUT 主要用在阻塞 IO，阻塞 IO 中 accept, read 等都是无限等待的，如果不希望永远阻塞，使用它调整超时时间

```

@Slf4j
public class TestConnectionTimeout {
    public static void main(String[] args) {
        NioEventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap()
                .group(group)
                .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 300)
                .channel(NioSocketChannel.class)
                .handler(new LoggingHandler());
            ChannelFuture future = bootstrap.connect("127.0.0.1", 8080);
            future.sync().channel().closeFuture().sync(); // 断点1
        } catch (Exception e) {
            e.printStackTrace();
            log.debug("timeout");
        } finally {

```

```

        group.shutdownGracefully();
    }
}

```

另外源码部分 `io.netty.channel.nio.AbstractNioChannel.AbstractNioUnsafe#connect`

```

@Override
public final void connect(
    final SocketAddress remoteAddress, final SocketAddress localAddress,
    final ChannelPromise promise) {
    // ...
    // Schedule connect timeout.
    int connectTimeoutMillis = config().getConnectTimeoutMillis();
    if (connectTimeoutMillis > 0) {
        connectTimeoutFuture = eventLoop().schedule(new Runnable() {
            @Override
            public void run() {
                ChannelPromise connectPromise =
AbstractNioChannel.this.connectPromise;
                ConnectTimeoutException cause =
                    new ConnectTimeoutException("connection timed out: " +
remoteAddress); // 断点2
                if (connectPromise != null && connectPromise.tryFailure(cause))
{
                    close(voidPromise());
                }
            }
        }, connectTimeoutMillis, TimeUnit.MILLISECONDS);
    }
    // ...
}

```

2) SO_BACKLOG

- 属于 ServerSocketChannel 参数

```

sequenceDiagram
    participant c as client
    participant s as server
    participant sq as syns queue
    participant aq as accept queue

    s ->> s : bind()
    s ->> s : listen()

```

```

c ->> c : connect()
c ->> s : 1. SYN
Note left of c : SYN_SEND
s ->> sq : put
Note right of s : SYN_RECV
s ->> c : 2. SYN + ACK
Note left of c : ESTABLISHED
c ->> s : 3. ACK
sq ->> aq : put
Note right of s : ESTABLISHED
aq -->> s :
s ->> s : accept()

```

1. 第一次握手，client 发送 SYN 到 server，状态修改为 SYN_SEND，server 收到，状态改变为 SYN_RECV，并将该请求放入 sync queue 队列
2. 第二次握手，server 回复 SYN + ACK 给 client，client 收到，状态改变为 ESTABLISHED，并发送 ACK 给 server
3. 第三次握手，server 收到 ACK，状态改变为 ESTABLISHED，将该请求从 sync queue 放入 accept queue

其中

- 在 linux 2.2 之前，backlog 大小包括了两个队列的大小，在 2.2 之后，分别用下面两个参数来控制
- sync queue - 半连接队列
 - 大小通过 /proc/sys/net/ipv4/tcp_max_syn_backlog 指定，在 `syncookies` 启用的情况下，逻辑上没有最大值限制，这个设置便被忽略
- accept queue - 全连接队列
 - 其大小通过 /proc/sys/net/core/somaxconn 指定，在使用 listen 函数时，内核会根据传入的 backlog 参数与系统参数，取二者的较小值
 - 如果 accept queue 队列满了，server 将发送一个拒绝连接的错误信息到 client

netty 中

可以通过 `option(ChannelOption.SO_BACKLOG, 值)` 来设置大小

可以通过下面源码查看默认大小

```
public class DefaultServerSocketChannelConfig extends DefaultChannelConfig
                                                implements
ServerSocketChannelConfig {

    private volatile int backlog = NetUtil.SOMAXCONN;
    // ...
}
```

课堂调试关键断点为: `io.netty.channel.nio.NioEventLoop#processSelectedKey`

oio 中更容易说明, 不用 debug 模式

```
public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket ss = new ServerSocket(8888, 2);
        Socket accept = ss.accept();
        System.out.println(accept);
        System.in.read();
    }
}
```

客户端启动 4 个

```
public class Client {
    public static void main(String[] args) throws IOException {
        try {
            Socket s = new Socket();
            System.out.println(new Date()+" connecting...");
            s.connect(new InetSocketAddress("localhost", 8888),1000);
            System.out.println(new Date()+" connected...");
            s.getOutputStream().write(1);
            System.in.read();
        } catch (IOException e) {
            System.out.println(new Date()+" connecting timeout...");
            e.printStackTrace();
        }
    }
}
```

第 1, 2, 3 个客户端都打印, 但除了第一个处于 accept 外, 其它两个都处于 accept queue 中

```
Tue Apr 21 20:30:28 CST 2020 connecting...
Tue Apr 21 20:30:28 CST 2020 connected...
```

第 4 个客户端连接时

```
Tue Apr 21 20:53:58 CST 2020 connecting...
Tue Apr 21 20:53:59 CST 2020 connecting timeout...
java.net.SocketTimeoutException: connect timed out
```

3) ulimit -n (数字)

linux客户端会限制一个进程可以同时打开的文件数量

- 属于操作系统参数

4) TCP_NODELAY

默认是false 使用了nagle算法来提高TCP的传输效率，建议设置成true

nagle算法大概意思就是积攒TCP请求到一定程度后会一起发出去，提高效率

- 属于 SocketChannel 参数

5) SO_SNDBUF & SO_RCVBUF

发送缓冲区和接收缓冲区，在滑动窗口那里解释过，建议不要调整

- SO_SNDBUF 属于 SocketChannel 参数
- SO_RCVBUF 既可用于 SocketChannel 参数，也可以用于 ServerSocketChannel 参数（建议设置到 ServerSocketChannel 上）

6) ALLOCATOR

ByteBuf的分配器

- 属于 SocketChannel 参数
- 用来分配 ByteBuf， ctx.alloc()

7) RCVBUF_ALLOCATOR

- 属于 SocketChannel 参数
- 控制 netty 接收缓冲区大小

- 负责入站数据的分配，决定入站缓冲区的大小（并可动态调整），统一采用 direct 直接内存，具体池化还是非池化由 allocator 决定

1.3 RPC 框架

1) 准备工作

这些代码可以认为是现成的，无需从头编写练习

为了简化起见，在原来聊天项目的基础上新增 Rpc 请求和响应消息

```
@Data
public abstract class Message implements Serializable {

    // 省略旧的代码

    public static final int RPC_MESSAGE_TYPE_REQUEST = 101;
    public static final int RPC_MESSAGE_TYPE_RESPONSE = 102;

    static {
        // ...
        messageClasses.put(RPC_MESSAGE_TYPE_REQUEST, RpcRequestMessage.class);
        messageClasses.put(RPC_MESSAGE_TYPE_RESPONSE, RpcResponseMessage.class);
    }
}
```

请求消息

```
@Getter
@Override
public class RpcRequestMessage extends Message {

    /**
     * 调用的接口全限定名，服务端根据它找到实现
     */
    private String interfaceName;

    /**
     * 调用接口中的方法名
     */
    private String methodName;

    /**
     * 方法返回类型
     */
}
```

```

        */
        private Class<?> returnType;
        /**
         * 方法参数类型数组
         */
        private Class[] parameterTypes;
        /**
         * 方法参数值数组
         */
        private Object[] parameterValue;

        public RpcRequestMessage(int sequenceId, String interfaceName, String
methodName, Class<?> returnType, Class[] parameterTypes, Object[]
parameterValue) {
            super.setSequenceId(sequenceId);
            this.interfaceName = interfaceName;
            this.methodName = methodName;
            this.returnType = returnType;
            this.parameterTypes = parameterTypes;
            this.parameterValue = parameterValue;
        }

        @Override
        public int getMessageType() {
            return RPC_MESSAGE_TYPE_REQUEST;
        }
    }
}

```

响应消息

```

@Data
@ToString(callSuper = true)
public class RpcResponseMessage extends Message {
    /**
     * 返回值
     */
    private Object returnValue;
    /**
     * 异常值
     */
    private Exception exceptionValue;

    @Override
    public int getMessageType() {
        return RPC_MESSAGE_TYPE_RESPONSE;
    }
}

```

服务器架子

```

@Slf4j
public class RpcServer {
    public static void main(String[] args) {
        NioEventLoopGroup boss = new NioEventLoopGroup();
        NioEventLoopGroup worker = new NioEventLoopGroup();
        LoggingHandler LOGGING_HANDLER = new LoggingHandler(LogLevel.DEBUG);
        MessageCodecSharable MESSAGE_CODEC = new MessageCodecSharable();

        // rpc 请求消息处理器，待实现
        RpcRequestMessageHandler RPC_HANDLER = new RpcRequestMessageHandler();
        try {
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.group(boss, worker);
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>()
{
            @Override
            protected void initChannel(SocketChannel ch) throws Exception {
                ch.pipeline().addLast(new ProtocolFrameDecoder());
                ch.pipeline().addLast(LOGGING_HANDLER);
                ch.pipeline().addLast(MESSAGE_CODEC);
                ch.pipeline().addLast(RPC_HANDLER);
            }
        });
            Channel channel = serverBootstrap.bind(8080).sync().channel();
            channel.closeFuture().sync();
        } catch (InterruptedException e) {
            log.error("server error", e);
        } finally {
            boss.shutdownGracefully();
            worker.shutdownGracefully();
        }
    }
}

```

客户端架子

```

public class RpcClient {
    public static void main(String[] args) {
        NioEventLoopGroup group = new NioEventLoopGroup();
        LoggingHandler LOGGING_HANDLER = new LoggingHandler(LogLevel.DEBUG);
        MessageCodecSharable MESSAGE_CODEC = new MessageCodecSharable();

        // rpc 响应消息处理器，待实现
        RpcResponseMessageHandler RPC_HANDLER = new RpcResponseMessageHandler();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.group(group);

```



```

bootstrap.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(new ProtocolFrameDecoder());
        ch.pipeline().addLast(LOGGING_HANDLER);
        ch.pipeline().addLast(MESSAGE_CODEC);
        ch.pipeline().addLast(RPC_HANDLER);
    }
});
Channel channel = bootstrap.connect("localhost",
8080).sync().channel();
channel.closeFuture().sync();
} catch (Exception e) {
    log.error("client error", e);
} finally {
    group.shutdownGracefully();
}
}
}

```

服务器端的 service 获取

```

public class ServicesFactory {

    static Properties properties;
    static Map<Class<?>, Object> map = new ConcurrentHashMap<>();

    static {
        try (InputStream in =
Config.class.getResourceAsStream("/application.properties")) {
            properties = new Properties();
            properties.load(in);
            Set<String> names = properties.stringPropertyNames();
            for (String name : names) {
                if (name.endsWith("Service")) {
                    Class<?> interfaceClass = Class.forName(name);
                    Class<?> instanceClass =
Class.forName(properties.getProperty(name));
                    map.put(interfaceClass, instanceClass.newInstance());
                }
            }
        } catch (IOException | ClassNotFoundException | InstantiationException |
IllegalAccessException e) {
            throw new ExceptionInInitializerError(e);
        }
    }

    public static <T> T getService(Class<T> interfaceClass) {
        return (T) map.get(interfaceClass);
    }
}

```

相关配置 application.properties

```
serializer.algorithm=json
cn.itcast.server.service.HelloService=cn.itcast.server.service.HelloServiceImpl
```

2) 服务器 handler

```
@Slf4j
@ChannelHandler.Sharable
public class RpcRequestMessageHandler extends
SimpleChannelInboundHandler<RpcRequestMessage> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, RpcRequestMessage
message) {
        RpcResponseMessage response = new RpcResponseMessage();
        response.setSequenceId(message.getSequenceId());
        try {
            // 获取真正的实现对象
            HelloService service = (HelloService)

            ServicesFactory.getService(Class.forName(message.getInterfaceName()));

            // 获取要调用的方法
            Method method =
service.getClass().getMethod(message.getMethodName(),
message.getParameterTypes());

            // 调用方法
            Object invoke = method.invoke(service, message.getParameterValue());
            // 调用成功
            response.setReturnValue(invoke);
        } catch (Exception e) {
            e.printStackTrace();
            // 调用异常
            response.setExceptionValue(e);
        }
        // 返回结果
        ctx.writeAndFlush(response);
    }
}
```

3) 客户端代码第一版

只发消息

```
@Slf4j
public class RpcClient {
    public static void main(String[] args) {
        NioEventLoopGroup group = new NioEventLoopGroup();
        LoggingHandler LOGGING_HANDLER = new LoggingHandler(LogLevel.DEBUG);
        MessageCodecSharable MESSAGE_CODEC = new MessageCodecSharable();
        RpcResponseMessageHandler RPC_HANDLER = new RpcResponseMessageHandler();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.group(group);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new ProtocolFrameDecoder());
                    ch.pipeline().addLast(LOGGING_HANDLER);
                    ch.pipeline().addLast(MESSAGE_CODEC);
                    ch.pipeline().addLast(RPC_HANDLER);
                }
            });
            Channel channel = bootstrap.connect("localhost",
            8080).sync().channel();

            ChannelFuture future = channel.writeAndFlush(new RpcRequestMessage(
                1,
                "cn.itcast.server.service.HelloService",
                "sayHello",
                String.class,
                new Class[]{String.class},
                new Object[]{"张三"}
            )).addListener(promise -> {
                if (!promise.isSuccess()) {
                    Throwable cause = promise.cause();
                    log.error("error", cause);
                }
            });

            channel.closeFuture().sync();
        } catch (Exception e) {
            log.error("client error", e);
        } finally {
            group.shutdownGracefully();
        }
    }
}
```

4) 客户端 handler 第一版

```
@Slf4j
@ChannelHandler.Sharable
public class RpcResponseMessageHandler extends
SimpleChannelInboundHandler<RpcResponseMessage> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, RpcResponseMessage
msg) throws Exception {
        log.debug("{} ", msg);
    }
}
```

5) 客户端代码 第二版

包括 channel 管理，代理，接收结果

```
@Slf4j
public class RpcClientManager {

    public static void main(String[] args) {
        HelloService service = getProxyService(HelloService.class);
        System.out.println(service.sayHello("zhangsan"));
        //      System.out.println(service.sayHello("lisi"));
        //      System.out.println(service.sayHello("wangwu"));
    }

    // 创建代理类
    public static <T> T getProxyService(Class<T> serviceClass) {
        ClassLoader loader = serviceClass.getClassLoader();
        Class<?>[] interfaces = new Class[]{serviceClass};
        //
        "张三"
        Object o = Proxy.newProxyInstance(loader, interfaces, (proxy, method,
args) -> {
            // 1. 将方法调用转换为 消息对象
            int sequenceId = SequenceIdGenerator.nextId();
            RpcRequestMessage msg = new RpcRequestMessage(
                sequenceId,
                serviceClass.getName(),
                method.getName(),
                method.getReturnType(),
                method.getParameterTypes(),
                args
            );
            // 2. 将消息对象发送出去
            getChannel().writeAndFlush(msg);
            sayHello
        });
    }
}
```

```

// 3. 准备一个空 Promise 对象，来接收结果                                指定 promise 对象异步
接收结果线程
    DefaultPromise<Object> promise = new DefaultPromise<>
(getChannel().eventLoop());
    RpcResponseMessageHandler.PROMISES.put(sequenceId, promise);

//          promise.addListener(future -> {
//              // 线程
//          });

// 4. 等待 promise 结果
promise.await();
if(promise.isSuccess()) {
    // 调用正常
    return promise.getNow();
} else {
    // 调用失败
    throw new RuntimeException(promise.cause());
}
});
return (T) o;
}

private static Channel channel = null;
private static final Object LOCK = new Object();

// 获取唯一的 channel 对象
public static Channel getChannel() {
    if (channel != null) {
        return channel;
    }
    synchronized (LOCK) { // t2
        if (channel != null) { // t1
            return channel;
        }
        initChannel();
        return channel;
    }
}

// 初始化 channel 方法
private static void initChannel() {
    NioEventLoopGroup group = new NioEventLoopGroup();
    LoggingHandler LOGGING_HANDLER = new LoggingHandler(LogLevel.DEBUG);
    MessageCodecSharable MESSAGE_CODEC = new MessageCodecSharable();
    RpcResponseMessageHandler RPC_HANDLER = new RpcResponseMessageHandler();
    Bootstrap bootstrap = new Bootstrap();
    bootstrap.channel(NioSocketChannel.class);
    bootstrap.group(group);
    bootstrap.handler(new ChannelInitializer<SocketChannel>() {
        @Override
        protected void initChannel(SocketChannel ch) throws Exception {
            ch.pipeline().addLast(new ProtocolFrameDecoder());
            ch.pipeline().addLast(LOGGING_HANDLER);
        }
    });
}

```

```

        ch.pipeline().addLast(MESSAGE_CODEC);
        ch.pipeline().addLast(RPC_HANDLER);
    }
});
try {
    channel = bootstrap.connect("localhost", 8080).sync().channel();
    channel.closeFuture().addListener(future -> {
        group.shutdownGracefully();
    });
} catch (Exception e) {
    log.error("client error", e);
}
}
}

```

6) 客户端 handler 第二版

```

@Slf4j
@ChannelHandler.Sharable
public class RpcResponseMessageHandler extends
SimpleChannelInboundHandler<RpcResponseMessage> {

    // 序号 用来接收结果的 promise 对象
    public static final Map<Integer, Promise<Object>> PROMISES = new
ConcurrentHashMap<>();

    @Override

    protected void channelRead0(ChannelHandlerContext ctx, RpcResponseMessage
msg) throws Exception {
        log.debug("{} ", msg);
        // 拿到空的 promise
        Promise<Object> promise = PROMISES.remove(msg.getSequenceId());
        if (promise != null) {
            Object returnValue = msg.getReturnValue();
            Exception exceptionValue = msg.getExceptionValue();
            if(exceptionValue != null) {
                promise.setFailure(exceptionValue);
            } else {
                promise.setSuccess(returnValue);
            }
        }
    }
}

```

2. 源码分析

2.1 启动剖析

我们就来看看 netty 中对下面的代码是怎样进行处理的

```
//1 netty 中使用 NioEventLoopGroup （简称 nio boss 线程）来封装线程和 selector
Selector selector = Selector.open();

//2 创建 NioServerSocketChannel，同时会初始化它关联的 handler，以及为原生 ssc 存储 config
NioServerSocketChannel attachment = new NioServerSocketChannel();

//3 创建 NioServerSocketChannel 时，创建了 java 原生的 ServerSocketChannel
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.configureBlocking(false);

//4 启动 nio boss 线程执行接下来的操作

//5 注册（仅关联 selector 和 NioServerSocketChannel），未关注事件
SelectionKey selectionKey = serverSocketChannel.register(selector, 0,
attachment);

//6 head -> 初始化器 -> ServerBootstrapAcceptor -> tail，初始化器是一次性的，只为添加 acceptor

//7 绑定端口
serverSocketChannel.bind(new InetSocketAddress(8080));

//8 触发 channel active 事件，在 head 中关注 op_accept 事件
selectionKey.interestOps(SelectionKey.OP_ACCEPT);
```

入口 `io.netty.bootstrap.ServerBootstrap#bind`

关键代码 `io.netty.bootstrap.AbstractBootstrap#doBind`

```
private ChannelFuture doBind(final SocketAddress localAddress) {
    // 1. 执行初始化和注册 regFuture 会由 initAndRegister 设置其是否完成，从而回调 3.2
    处代码
    final ChannelFuture regFuture = initAndRegister();
    final Channel channel = regFuture.channel();
    if (regFuture.cause() != null) {
        return regFuture;
    }

    // 2. 因为是 initAndRegister 异步执行，需要分两种情况来看，调试时也需要通过 suspend 断
    点类型加以区分
    // 2.1 如果已经完成
```

```

        if (regFuture.isDone()) {
            ChannelPromise promise = channel.newPromise();
            // 3.1 立刻调用 doBind0
            doBind0(regFuture, channel, localAddress, promise);
            return promise;
        }
        // 2.2 还没有完成
        else {
            final PendingRegistrationPromise promise = new
PendingRegistrationPromise(channel);
            // 3.2 回调 doBind0
            regFuture.addListener(new ChannelFutureListener() {
                @Override
                public void operationComplete(ChannelFuture future) throws Exception
            {
                Throwable cause = future.cause();
                if (cause != null) {
                    // 处理异常...
                    promise.setFailure(cause);
                } else {
                    promise.registered();
                    // 3. 由注册线程去执行 doBind0
                    doBind0(regFuture, channel, localAddress, promise);
                }
            }
        });
        return promise;
    }
}

```

关键代码 `io.netty.bootstrap.AbstractBootstrap#initAndRegister`

```

final ChannelFuture initAndRegister() {
    Channel channel = null;
    try {
        channel = channelFactory.newChannel();
        // 1.1 初始化 - 做的事就是添加一个初始化器 ChannelInitializer
        init(channel);
    } catch (Throwable t) {
        // 处理异常...
        return new DefaultChannelPromise(new FailedChannel(),
GlobalEventExecutor.INSTANCE).setFailure(t);
    }

    // 1.2 注册 - 做的事就是将原生 channel 注册到 selector 上
    ChannelFuture regFuture = config().group().register(channel);
    if (regFuture.cause() != null) {
        // 处理异常...
    }
    return regFuture;
}

```


关键代码 `io.netty.bootstrap.ServerBootstrap#init`

```
// 这里 channel 实际上是 NioServerSocketChannel
void init(Channel channel) throws Exception {
    final Map<ChannelOption<?>, Object> options = options0();
    synchronized (options) {
        setChannelOptions(channel, options, logger);
    }

    final Map<AttributeKey<?>, Object> attrs = attrs0();
    synchronized (attrs) {
        for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {
            @SuppressWarnings("unchecked")
            AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();
            channel.attr(key).set(e.getValue());
        }
    }

    ChannelPipeline p = channel.pipeline();

    final EventLoopGroup currentChildGroup = childGroup;
    final ChannelHandler currentChildHandler = childHandler;
    final Entry<ChannelOption<?>, Object>[] currentChildOptions;
    final Entry<AttributeKey<?>, Object>[] currentChildAttrs;
    synchronized (childOptions) {
        currentChildOptions =
            childOptions.entrySet().toArray(new OptionArray(0));
    }
    synchronized (childAttrs) {
        currentChildAttrs = childAttrs.entrySet().toArray(new AttrArray(0));
    }

    // 为 NioServerSocketChannel 添加初始化器
    p.addLast(new ChannelInitializer<Channel>() {
        @Override
        public void initChannel(final Channel ch) throws Exception {
            final ChannelPipeline pipeline = ch.pipeline();
            ChannelHandler handler = config.handler();
            if (handler != null) {
                pipeline.addLast(handler);
            }

            // 初始化器的职责是将 ServerBootstrapAcceptor 加入至
            NioServerSocketChannel
            ch.eventLoop().execute(new Runnable() {
                @Override
                public void run() {
                    pipeline.addLast(new ServerBootstrapAcceptor(
                        ch, currentChildGroup, currentChildHandler,
                        currentChildOptions, currentChildAttrs));
                }
            });
        }
    });
}
```

```

    }
    });
}

```

关键代码 `io.netty.channel.AbstractChannel.AbstractUnsafe#register`

```

public final void register(EventLoop eventLoop, final ChannelPromise promise) {
    // 一些检查, 略...

    AbstractChannel.this.eventLoop = eventLoop;

    if (eventLoop.inEventLoop()) {
        register0(promise);
    } else {
        try {
            // 首次执行 execute 方法时, 会启动 nio 线程, 之后注册等操作在 nio 线程上执行
            // 因为只有一个 NioServerSocketChannel 因此, 也只会有一个 boss nio 线程
            // 这行代码完成的事实是 main -> nio boss 线程的切换
            eventLoop.execute(new Runnable() {
                @Override
                public void run() {
                    register0(promise);
                }
            });
        } catch (Throwable t) {
            // 日志记录...
            closeForcibly();
            closeFuture.setClosed();
            safeSetFailure(promise, t);
        }
    }
}

```

`io.netty.channel.AbstractChannel.AbstractUnsafe#register0`

```

private void register0(ChannelPromise promise) {
    try {
        if (!promise.setUncancellable() || !ensureOpen(promise)) {
            return;
        }
        boolean firstRegistration = neverRegistered;
        // 1.2.1 原生的 nio channel 绑定到 selector 上, 注意此时没有注册 selector 关注
        // 事件, 附件为 NioServerSocketChannel
        doRegister();
        neverRegistered = false;
        registered = true;
    }
}

```

```

// 1.2.2 执行 NioServerSocketChannel 初始化器的 initChannel
pipeline.invokeHandlerAddedIfNeeded();

// 回调 3.2 io.netty.bootstrap.AbstractBootstrap#doBind0
safeSetSuccess(promise);
pipeline.fireChannelRegistered();

// 对应 server socket channel 还未绑定, isActive 为 false
if (isActive()) {
    if (firstRegistration) {
        pipeline.fireChannelActive();
    } else if (config().isAutoRead()) {
        beginRead();
    }
}
} catch (Throwable t) {
    // Close the channel directly to avoid FD leak.
    closeForcibly();
    closeFuture.setClosed();
    safeSetFailure(promise, t);
}
}

```

关键代码 `io.netty.channel.ChannelInitializer#initChannel`

```

private boolean initChannel(ChannelHandlerContext ctx) throws Exception {
    if (initMap.add(ctx)) { // Guard against re-entrance.
        try {
            // 1.2.2.1 执行初始化
            initChannel((C) ctx.channel());
        } catch (Throwable cause) {
            exceptionCaught(ctx, cause);
        } finally {
            // 1.2.2.2 移除初始化器
            ChannelPipeline pipeline = ctx.pipeline();
            if (pipeline.context(this) != null) {
                pipeline.remove(this);
            }
        }
        return true;
    }
    return false;
}

```

关键代码 `io.netty.bootstrap.AbstractBootstrap#doBind0`

```

// 3.1 或 3.2 执行 doBind0

```

```

private static void doBind0(
    final ChannelFuture regFuture, final Channel channel,
    final SocketAddress localAddress, final ChannelPromise promise) {

    channel.eventLoop().execute(new Runnable() {
        @Override
        public void run() {
            if (regFuture.isSuccess()) {
                channel.bind(localAddress,
promise).addListener(ChannelFutureListener.CLOSE_ON_FAILURE);
            } else {
                promise.setFailure(regFuture.cause());
            }
        }
    });
}

```

关键代码 `io.netty.channel.AbstractChannel.AbstractUnsafe#bind`

```

public final void bind(final SocketAddress localAddress, final ChannelPromise
promise) {
    assertEventLoop();

    if (!promise.setUncancellable() || !ensureOpen(promise)) {
        return;
    }

    if (Boolean.TRUE.equals(config().getOption(ChannelOption.SO_BROADCAST)) &&
        localAddress instanceof InetSocketAddress &&
        !((InetSocketAddress) localAddress).getAddress().isAnyLocalAddress() &&
        !PlatformDependent.isWindows() && !PlatformDependent.maybeSuperUser()) {
        // 记录日志...
    }

    boolean wasActive = isActive();
    try {
        // 3.3 执行端口绑定
        doBind(localAddress);
    } catch (Throwable t) {
        safeSetFailure(promise, t);
        closeIfClosed();
        return;
    }

    if (!wasActive && isActive()) {
        invokeLater(new Runnable() {
            @Override
            public void run() {
                // 3.4 触发 active 事件
                pipeline.fireChannelActive();
            }
        });
    }
}

```

```

    }

    safeSetSuccess(promise);
}

```

3.3 关键代码 `io.netty.channel.socket.nio.NioServerSocketChannel#doBind`

```

protected void doBind(SocketAddress localAddress) throws Exception {
    if (PlatformDependent.javaVersion() >= 7) {
        javaChannel().bind(localAddress, config.getBacklog());
    } else {
        javaChannel().socket().bind(localAddress, config.getBacklog());
    }
}

```

3.4 关键代码 `io.netty.channel.DefaultChannelPipeline.HeadContext#channelActive`

```

public void channelActive(ChannelHandlerContext ctx) {
    ctx.fireChannelActive();
    // 触发 read (NioServerSocketChannel 上的 read 不是读取数据，只是为了触发 channel
    的事件注册)
    readIfIsAutoRead();
}

```

关键代码 `io.netty.channel.nio.AbstractNioChannel#doBeginRead`

```

protected void doBeginRead() throws Exception {
    // Channel.read() or ChannelHandlerContext.read() was called
    final SelectionKey selectionKey = this.selectionKey;
    if (!selectionKey.isValid()) {
        return;
    }

    readPending = true;

    final int interestOps = selectionKey.interestOps();
    // readInterestOp 取值是 16，在 NioServerSocketChannel 创建时初始化好，代表关注
    accept 事件
    if ((interestOps & readInterestOp) == 0) {
        selectionKey.interestOps(interestOps | readInterestOp);
    }
}

```

2.2 NioEventLoop 剖析

NioEventLoop 线程不仅要处理 IO 事件，还要处理 Task（包括普通任务和定时任务），

提交任务代码 `io.netty.util.concurrent.SingleThreadEventExecutor#execute`

```
public void execute(Runnable task) {
    if (task == null) {
        throw new NullPointerException("task");
    }

    boolean inEventLoop = inEventLoop();
    // 添加任务，其中队列使用了 jctools 提供的 mpsc 无锁队列
    addTask(task);
    if (!inEventLoop) {
        // inEventLoop 如果为 false 表示由其它线程来调用 execute，即首次调用，这时需要向
        // eventLoop 提交首个任务，启动死循环，会执行到下面的 doStartThread
        startThread();
        if (isShutdown()) {
            // 如果已经 shutdown，做拒绝逻辑，代码略...
        }
    }

    if (!addTaskWakesUp && wakesUpForTask(task)) {
        // 如果线程由于 IO select 阻塞了，添加的任务的线程需要负责唤醒 NioEventLoop 线程
        wakeup(inEventLoop);
    }
}
```

唤醒 select 阻塞线程 `io.netty.channel.nio.NioEventLoop#wakeup`

```
@Override
protected void wakeup(boolean inEventLoop) {
    if (!inEventLoop && wakenUp.compareAndSet(false, true)) {
        selector.wakeup();
    }
}
```

启动 EventLoop 主循环

`io.netty.util.concurrent.SingleThreadEventExecutor#doStartThread`

```

private void doStartThread() {
    assert thread == null;
    executor.execute(new Runnable() {
        @Override
        public void run() {
            // 将线程池的当前线程保存在成员变量中，以便后续使用
            thread = Thread.currentThread();
            if (interrupted) {
                thread.interrupt();
            }

            boolean success = false;
            updateLastExecutionTime();
            try {
                // 调用外部类 SingleThreadEventExecutor 的 run 方法，进入死循环，run
                SingleThreadEventExecutor.this.run();
                success = true;
            } catch (Throwable t) {
                logger.warn("Unexpected exception from an event executor: ", t);
            } finally {
                // 清理工作，代码略...
            }
        }
    });
}

```

方法见下

`io.netty.channel.nio.NioEventLoop#run` 主要任务是执行死循环，不断看有没有新任务，有没有 IO 事件

```

protected void run() {
    for (;;) {
        try {
            try {
                // calculateStrategy 的逻辑如下：
                // 有任务，会执行一次 selectNow，清除上一次的 wakeup 结果，无论有没有 IO
                // 事件，都会跳过 switch
                // 没有任务，会匹配 SelectStrategy.SELECT，看是否应当阻塞
                switch (selectStrategy.calculateStrategy(selectNowSupplier,
                    hasTasks())) {
                    case SelectStrategy.CONTINUE:
                        continue;

                    case SelectStrategy.BUSY_WAIT:

                    case SelectStrategy.SELECT:
                        // 因为 IO 线程和提交任务线程都有可能执行 wakeup，而 wakeup 属于
                        // 比较昂贵的操作，因此使用了一个原子布尔对象 wakenUp，它取值为 true 时，表示该由当前线程唤醒
                        // 进行 select 阻塞，并设置唤醒状态为 false
                        boolean oldwakeup = wakenUp.getAndSet(false);

```

并 wakeup

会及时执行呢?

到超时

```
// 如果在这个位置，非 EventLoop 线程抢先将 wakenUp 置为 true，

// 下面的 select 方法不会阻塞
// 等 runAllTasks 处理完成后，到再循环进来这个阶段新增的任务会不

// 因为 oldwakenUp 为 true，因此下面的 select 方法就会阻塞，直

// 才能执行，让 select 方法无谓阻塞
select(oldwakenUp);

    if (wakenUp.get()) {
        selector.wakeup();
    }
    default:
}
} catch (IOException e) {
    rebuildSelector0();
    handleLoopException(e);
    continue;
}

cancelledKeys = 0;
needsToSelectAgain = false;
// ioRatio 默认是 50
final int ioRatio = this.ioRatio;
if (ioRatio == 100) {
    try {
        processSelectedKeys();
    } finally {
        // ioRatio 为 100 时，总是运行完所有非 IO 任务
        runAllTasks();
    }
} else {
    final long ioStartTime = System.nanoTime();
    try {
        processSelectedKeys();
    } finally {
        // 记录 io 事件处理耗时
        final long ioTime = System.nanoTime() - ioStartTime;
        // 运行非 IO 任务，一旦超时会退出 runAllTasks
        runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
    }
}
} catch (Throwable t) {
    handleLoopException(t);
}
try {
    if (isShuttingDown()) {
        closeAll();
        if (confirmShutdown()) {
            return;
        }
    }
} catch (Throwable t) {
    handleLoopException(t);
}
```



```

    }
}
}

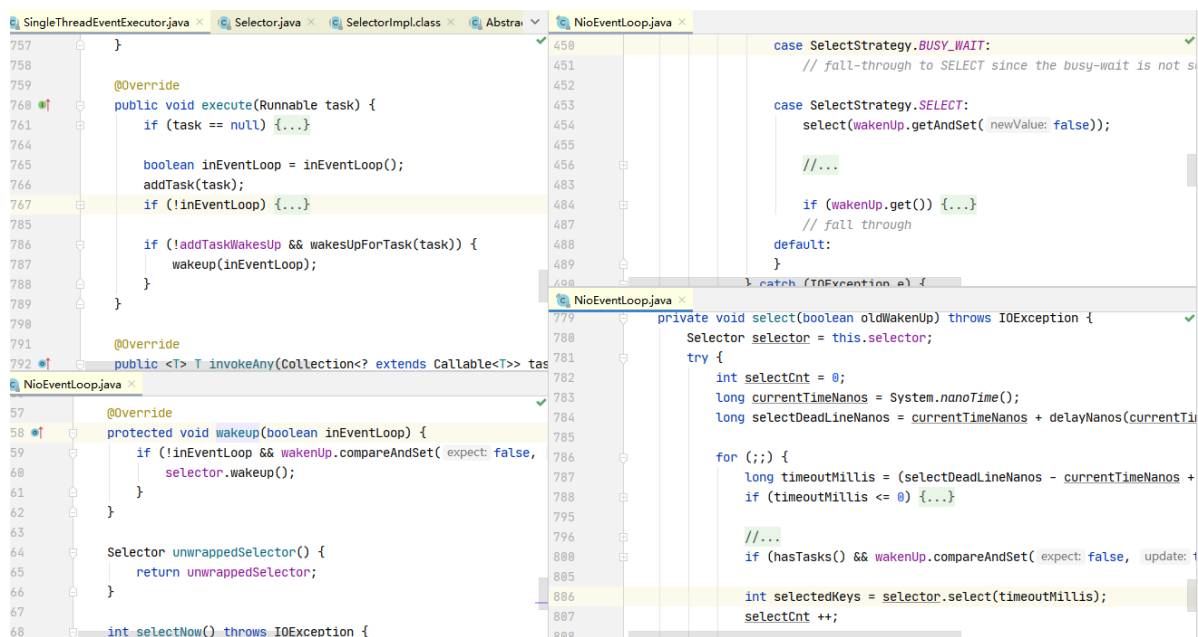
```

⚠ 注意

这里有个费解的地方就是 wakeup，它既可以由提交任务的线程来调用（比较好理解），也可以由 EventLoop 线程来调用（比较费解），这里要知道 wakeup 方法的效果：

- 由非 EventLoop 线程调用，会唤醒当前在执行 select 阻塞的 EventLoop 线程
- 由 EventLoop 自己调用，会本次的 wakeup 会取消下一次的 select 操作

参考下图



io.netty.channel.nio.NioEventLoop#select

```

private void select(boolean oldwakeup) throws IOException {
    Selector selector = this.selector;
    try {
        int selectCnt = 0;
        long currentTimeNanos = System.nanoTime();
        // 计算等待时间
        // * 没有 scheduledTask, 超时时间为 1s
        // * 有 scheduledTask, 超时时间为 `下一个定时任务执行时间 - 当前时间`
    }
}

```

```

        long selectDeadlineNanos = currentTimeNanos +
delayNanos(currentTimeNanos);

        for (;;) {
            long timeoutMillis = (selectDeadlineNanos - currentTimeNanos +
500000L) / 1000000L;
            // 如果超时，退出循环
            if (timeoutMillis <= 0) {
                if (selectCnt == 0) {
                    selector.selectNow();
                    selectCnt = 1;
                }
                break;
            }

            // 如果期间又有 task 退出循环，如果没这个判断，那么任务就会等到下次 select 超时
            // 时才能被执行
            // wakeup.compareAndSet(false, true) 是让非 NioEventLoop 不必再执行
            wakeup

            if (hasTasks() && wakeup.compareAndSet(false, true)) {
                selector.selectNow();
                selectCnt = 1;
                break;
            }

            // select 有限时阻塞
            // 注意 nio 有 bug，当 bug 出现时，select 方法即使没有时间发生，也不会阻塞住，
            // 导致不断空轮询，cpu 占用 100%
            int selectedKeys = selector.select(timeoutMillis);
            // 计数加 1
            selectCnt ++;

            // 醒来后，如果有 IO 事件、或是由非 EventLoop 线程唤醒，或者有任务，退出循环
            if (selectedKeys != 0 || oldwakeup || wakeup.get() || hasTasks()
|| hasScheduledTasks()) {
                break;
            }
            if (Thread.interrupted()) {
                // 线程被打断，退出循环
                // 记录日志
                selectCnt = 1;
                break;
            }

            long time = System.nanoTime();
            if (time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >=
currentTimeNanos) {
                // 如果超时，计数重置为 1，下次循环就会 break
                selectCnt = 1;
            }
            // 计数超过阈值，由 io.netty.selectorAutoRebuildThreshold 指定，默认 512
            // 这是为了解决 nio 空轮询 bug
            else if (SELECTOR_AUTO_REBUILD_THRESHOLD > 0 &&
selectCnt >= SELECTOR_AUTO_REBUILD_THRESHOLD) {
                // 重建 selector

```

```

        selector = selectRebuildSelector(selectCnt);
        selectCnt = 1;
        break;
    }

    currentTimeNanos = time;
}

if (selectCnt > MIN_PREMATURE_SELECTOR_RETURNS) {
    // 记录日志
}
} catch (CancelledKeyException e) {
    // 记录日志
}
}
}

```

处理 keys `io.netty.channel.nio.NioEventLoop#processSelectedKeys`

```

private void processSelectedKeys() {
    if (selectedKeys != null) {
        // 通过反射将 Selector 实现类中的就绪事件集合替换为 SelectedSelectionKeySet
        // SelectedSelectionKeySet 底层为数组实现，可以提高遍历性能（原本为 HashSet）
        processSelectedKeysOptimized();
    } else {
        processSelectedKeysPlain(selector.selectedKeys());
    }
}
}

```

`io.netty.channel.nio.NioEventLoop#processSelectedKey`

```

private void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {
    final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();
    // 当 key 取消或关闭时会导致这个 key 无效
    if (!k.isValid()) {
        // 无效时处理...
        return;
    }

    try {
        int readyOps = k.readyOps();
        // 连接事件
        if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
            int ops = k.interestOps();
            ops &= ~SelectionKey.OP_CONNECT;
            k.interestOps(ops);
        }
    }
}

```

```

        unsafe.finishConnect();
    }

    // 可写事件
    if ((readyOps & SelectionKey.OP_WRITE) != 0) {
        ch.unsafe().forceFlush();
    }

    // 可读或可接入事件
    if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 ||
readyOps == 0) {
        // 如果是可接入
        io.netty.channel.nio.AbstractNioMessageChannel.NioMessageUnsafe#read
        // 如果是可读
        io.netty.channel.nio.AbstractNioByteChannel.NioByteUnsafe#read
        unsafe.read();
    }
} catch (CancelledKeyException ignored) {
    unsafe.close(unsafe.voidPromise());
}
}

```

2.3 accept 剖析

nio 中如下代码，在 netty 中的流程

```

//1 阻塞直到事件发生
selector.select();

Iterator<SelectionKey> iter = selector.selectedKeys().iterator();
while (iter.hasNext()) {
    //2 拿到一个事件
    SelectionKey key = iter.next();

    //3 如果是 accept 事件
    if (key.isAcceptable()) {

        //4 执行 accept
        SocketChannel channel = serverSocketChannel.accept();
        channel.configureBlocking(false);

        //5 关注 read 事件
        channel.register(selector, SelectionKey.OP_READ);
    }
    // ...
}
}

```

先来看可接入事件处理 (accept)

```

public void read() {
    assert eventLoop().inEventLoop();
    final ChannelConfig config = config();
    final ChannelPipeline pipeline = pipeline();
    final RecvByteBufAllocator.Handle allocHandle =
unsafe().recvBufAllocHandle();
    allocHandle.reset(config);

    boolean closed = false;
    Throwable exception = null;
    try {
        try {
            do {
                // doReadMessages 中执行了 accept 并创建 NioSocketChannel 作为消息放
入 readBuf

                // readBuf 是一个 ArrayList 用来缓存消息
                int localRead = doReadMessages(readBuf);
                if (localRead == 0) {
                    break;
                }
                if (localRead < 0) {
                    closed = true;
                    break;
                }
                // localRead 为 1, 就一条消息, 即接收一个客户端连接
                allocHandle.incMessagesRead(localRead);
            } while (allocHandle.continueReading());
        } catch (Throwable t) {
            exception = t;
        }

        int size = readBuf.size();
        for (int i = 0; i < size; i++) {
            readPending = false;
            // 触发 read 事件, 让 pipeline 上的 handler 处理, 这时是处理
            //
io.netty.bootstrap.ServerBootstrap.ServerBootstrapAcceptor#channelRead
            pipeline.fireChannelRead(readBuf.get(i));
        }
        readBuf.clear();
        allocHandle.readComplete();
        pipeline.fireChannelReadComplete();

        if (exception != null) {
            closed = closeOnReadError(exception);

            pipeline.fireExceptionCaught(exception);
        }
    }
}

```

```

        if (closed) {
            inputShutdown = true;
            if (isOpen()) {
                close(voidPromise());
            }
        }
    } finally {
        if (!readPending && !config.isAutoRead()) {
            removeReadOp();
        }
    }
}
}

```

关键代码 `io.netty.bootstrap.ServerBootstrap.ServerBootstrapAcceptor#channelRead`

```

public void channelRead(ChannelHandlerContext ctx, Object msg) {
    // 这时的 msg 是 NioSocketChannel
    final Channel child = (Channel) msg;

    // NioSocketChannel 添加 childHandler 即初始化器
    child.pipeline().addLast(childHandler);

    // 设置选项
    setChannelOptions(child, childOptions, logger);

    for (Entry<AttributeKey<?>, Object> e: childAttrs) {
        child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());
    }

    try {
        // 注册 NioSocketChannel 到 nio worker 线程, 接下来的处理也移交至 nio worker
        // 线程
        childGroup.register(child).addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws Exception
            {
                if (!future.isSuccess()) {
                    forceClose(child, future.cause());
                }
            }
        });
    } catch (Throwable t) {
        forceClose(child, t);
    }
}

```

又回到了熟悉的 `io.netty.channel.AbstractChannel.AbstractUnsafe#register` 方法

```

public final void register(EventLoop eventLoop, final ChannelPromise promise) {
    // 一些检查, 略...

    AbstractChannel.this.eventLoop = eventLoop;

    if (eventLoop.inEventLoop()) {
        register0(promise);
    } else {
        try {
            // 这行代码完成的事实是 nio boss -> nio worker 线程的切换
            eventLoop.execute(new Runnable() {
                @Override
                public void run() {
                    register0(promise);
                }
            });
        } catch (Throwable t) {
            // 日志记录...
            closeForcibly();
            closeFuture.setClosed();
            safeSetFailure(promise, t);
        }
    }
}

```

io.netty.channel.AbstractChannel.AbstractUnsafe#register0

```

private void register0(ChannelPromise promise) {
    try {
        if (!promise.setUncancellable() || !ensureOpen(promise)) {
            return;
        }
        boolean firstRegistration = neverRegistered;
        doRegister();
        neverRegistered = false;
        registered = true;

        // 执行初始化器, 执行前 pipeline 中只有 head -> 初始化器 -> tail
        pipeline.invokeHandlerAddedIfNeeded();
        // 执行后就是 head -> logging handler -> my handler -> tail

        safeSetSuccess(promise);
        pipeline.fireChannelRegistered();

        if (isActive()) {
            if (firstRegistration) {
                // 触发 pipeline 上 active 事件
                pipeline.fireChannelActive();
            } else if (config().isAutoRead()) {
                beginRead();
            }
        }
    }
}

```

```

        }
    }
} catch (Throwable t) {
    closeForcibly();
    closeFuture.setClosed();
    safeSetFailure(promise, t);
}
}

```

回到了熟悉的代码 `io.netty.channel.DefaultChannelPipeline.HeadContext#channelActive`

```

public void channelActive(ChannelHandlerContext ctx) {
    ctx.fireChannelActive();
    // 触发 read (NioSocketChannel 这里 read, 只是为了触发 channel 的事件注册, 还未涉及
    // 数据读取)
    readIfIsAutoRead();
}

```

`io.netty.channel.nio.AbstractNioChannel#doBeginRead`

```

protected void doBeginRead() throws Exception {
    // Channel.read() or ChannelHandlerContext.read() was called
    final SelectionKey selectionKey = this.selectionKey;
    if (!selectionKey.isValid()) {
        return;
    }

    readPending = true;
    // 这时候 interestOps 是 0
    final int interestOps = selectionKey.interestOps();
    if ((interestOps & readInterestOp) == 0) {
        // 关注 read 事件
        selectionKey.interestOps(interestOps | readInterestOp);
    }
}

```

2.4 read 剖析

再来看可读事件 `io.netty.channel.nio.AbstractNioByteChannel.NioByteUnsafe#read`, 注意发送的数据未必能够一次读完, 因此会触发多次 nio read 事件, 一次事件内会触发多次 pipeline read, 一次事件会触发一次 pipeline read complete


```

public final void read() {
    final ChannelConfig config = config();
    if (shouldBreakReadReady(config)) {
        clearReadPending();
        return;
    }
    final ChannelPipeline pipeline = pipeline();
    // io.netty allocator.type 决定 allocator 的实现
    final ByteBufAllocator allocator = config.getAllocator();
    // 用来分配 byteBuf, 确定单次读取大小
    final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();
    allocHandle.reset(config);

    ByteBuf byteBuf = null;
    boolean close = false;
    try {
        do {
            byteBuf = allocHandle.allocate(allocator);
            // 读取
            allocHandle.lastBytesRead(doReadBytes(byteBuf));
            if (allocHandle.lastBytesRead() <= 0) {
                byteBuf.release();
                byteBuf = null;
                close = allocHandle.lastBytesRead() < 0;
                if (close) {
                    readPending = false;
                }
                break;
            }

            allocHandle.incMessagesRead(1);
            readPending = false;
            // 触发 read 事件, 让 pipeline 上的 handler 处理, 这时是处理
            NioSocketChannel 上的 handler
            pipeline.fireChannelRead(byteBuf);
            byteBuf = null;
        }
        // 是否要继续循环
        while (allocHandle.continueReading());

        allocHandle.readComplete();
        // 触发 read complete 事件
        pipeline.fireChannelReadComplete();

        if (close) {
            closeOnRead(pipeline);
        }
    } catch (Throwable t) {
        handleReadException(pipeline, byteBuf, t, close, allocHandle);
    } finally {
        if (!readPending && !config.isAutoRead()) {
            removeReadOp();
        }
    }
}

```

```
}  
}
```

```
io.netty.channel.DefaultMaxMessagesRecvByteBufAllocator.MaxMessageHandle#continueReading(io.netty.util.UncheckedBooleanSupplier)
```

```
public boolean continueReading(UncheckedBooleanSupplier maybeMoreDataSupplier) {  
    return  
        // 一般为 true  
        config.isAutoRead() &&  
        // respectMaybeMoreData 默认为 true  
        // maybeMoreDataSupplier 的逻辑是如果预期读取字节与实际读取字节相等，返回 true  
        (!respectMaybeMoreData || maybeMoreDataSupplier.get()) &&  
        // 小于最大次数，maxMessagePerRead 默认 16  
        totalMessages < maxMessagePerRead &&  
        // 实际读到了数据  
        totalBytesRead > 0;  
}
```