链滴

Java NIO 详解

作者：kyrie92

原文链接：https://ld246.com/article/1667273880763

来源网站：链滴

[name=18654169290][time=Mon, Oct 31, 2022 3:56 PM]

Java NIO详解

1.NIO 三大组件基本介绍

1.1 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，也以将Buffer的数据写入到Channel，而之前的Stream要么是输入，要么是输出，Channel比Stream为底层:

```
graph LR
channel --> buffer
buffer --> channel
```

常见的Channel有：

:::info

- FileChannel：用于处理文件读写的通道
- DatagramChannel：用于UDP数据读写的通道
- SocketChannel：用于TCP数据读写的通道，类似于Socket
- ServerSocketChannel：用于TCP数据读写的通道，类似于ServerSocket

:::

Buffer(即缓冲区)则用来缓冲读写数据，常见的Buffer有：

:::info

- ByteBuffer：字节Buffer，用的最多的，包括
  - MappedByteBuffer
  - DirectByteBuffer
  - HeapByteBuffer
- ShortBuffer
- IntBuffer
- LongBuffer
- DoubleBuffer
- FloatBuffer
- CharBuffer

:::

Selector

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，不会让线程阻塞在一个Channel上

```
graph TD
subgraph Selector架构设计
Thread --> Selector
Selector --> c1(Channel)
Selector --> c2(Channel)
Selector --> c3(Channel)
end
```

调用Selector的select()方法会阻塞直到Channel发生了读写就绪事件，这些事件发生，select方法就返回这些事件交给Thread来处理

2.ByteBuffer 的使用

2.1 Buffer基本概念

缓冲区(Buffer)：缓冲区本质上是一个**可以读写数据的内存块**，可以理解为是一个容器对象（含数组），该对象提供了一组方法，可以轻松地使用内存块，缓冲区对象内置了一些机制，能够跟踪和记录缓区的状态变化情况。Channel提供从文件、网络读取数据的渠道，但是读取和写入的数据都必须经由Buffer

2.2 Buffer类关键属性

```
// 标记
private int mark = -1;
// 位置，表示下一个要被读或写的元素的索引，每次读与写缓冲区都会改变该值，为下次疵读或写作备
private int position = 0;
// 表示缓冲区当前终点，不能对缓冲区超过极限的位置进行读写操作，且极限是可以修改的
private int limit;
// 容量，即可以容纳的最大数据量，在缓冲区创建时被设定并且不能改变
private int capacity;
```

2.3 使用示例

```
package com.mars.example.nio;

import java.nio.IntBuffer;

/**
 * @author: kyrie  @date: 2022/10/19 13:51
 * @desc: Java NIO 中 buffer的使用
 */
public class BasicBuffer {
    public static void main(String[] args) {
        // TP 1.创建一个buffer，大小为5，即可以存放5个int
        IntBuffer intBuffer = IntBuffer.allocate(5);
        // 向buffer 存放数据
        // TP 2. capacity()方法表示容量 put()方法表示存放
        for (int i =0; i < intBuffer.capacity(); i++) {
```

```
            intBuffer.put(i * 2);
        }

        // TP 调用clear()方法的结果是将各个标记恢复到初始状态，但是数据并没有被清除，此时重写入数据会覆盖之前的数据
        // TP 源码: position = 0; limit = capacity; mark = -1; return this;
        // intBuffer.clear();

        // TP error code: 写入时超过limit极限的操作会抛出异常：java.nio.BufferOverflowException
        // intBuffer.put(10);

        // 从buffer取数据
        // TP flip方法表示将buffer转换，读写切换（很重要，必须要切换，不切换则无法读取，具体参考源码内切换后变更的各种标识）
        // TP flip() 方法会重置 position 的值为0;源码：limit = position; position = 0; mark = -1;
        intBuffer.flip();

        // TP 手动设置position位置: 如下即表示从索引为1开始读取，结果会打印：2 4 6 8
        intBuffer.position(1);

        // TP 手动设置limit: 如下即表示极限操作为3，结果会打印：2 4
        intBuffer.limit(3);

        // TP error code: 设置的limit超过最大限制，会抛出异常：java.lang.IllegalArgumentExcepti
n
        // intBuffer.limit(6);

        // TP hasRemaining 方法表示是否还存在元素
        while (intBuffer.hasRemaining()) {
            // TP get 方法获取数据
            System.out.println(intBuffer.get());
        }
        // TP error code: 读取时超过limit极限的操作会抛出异常：java.nio.BufferUnderflowExceptio

        // System.out.println(intBuffer.get());
    }
}
```

3.FileChannel 的使用

3.1 常用方法

public int read(ByteBuffer dst): 从Channel通道中读取数据并写入到缓冲区中
public int write(ByteBuffer src): 将缓冲区的数据写入到通道Channel中
public long transferFrom(ReadableByteChannel src, long position, long count): 从目标通道中制数据到当前通道中（应用于文件拷贝，速度很快）
public long transferTo(long position, long count,WritableByteChannel target): 从当前通道将数复制到目标通道中（应用于文件拷贝，速度很快）

3.2 FileChannel 案例应用

- [案例一：写数据到本地文件中]

```
package com.mars.example.nio.channel.fileChannel;

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

/**
 * @author: kyrie  @date: 2022/10/21 14:23
 * @desc: 使用FileChannel 写入数据到本地文件中
 */
public class NIOFileChannelSample01 {
    public static void main(String[] args) throws IOException {
        // 创建输出流
        FileOutputStream fileOutputStream = new FileOutputStream("/Users/kyrie/Documents/
oc/private/Netty/netty-study/src/main/resources/NIOFileChannelSample01.txt");
        // 通过输出流获取FileChannel
        FileChannel channel = fileOutputStream.getChannel();
        // 创建缓冲区（channel只与缓冲区交互）
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        //  数据源
        String source = "这是一段测试源数据";
        // 写入到buffer
        buffer.put(source.getBytes());
        // 读写切换
        buffer.flip();
        // 将缓冲区的数据写到至通道
        channel.write(buffer);
        // 关闭流
        fileOutputStream.close();
    }
}
```

● [案例二：读取本地文件的数据并输出到控制台]

```
package com.mars.example.nio.channel.fileChannel;

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

/**
 * @author: kyrie  @date: 2022/10/21 14:23
 * @desc: 使用FileChannel 读取NIOFileChannelSample01中的本地文件数据并打印到控制台
 */
public class NIOFileChannelSample02 {
    public static void main(String[] args) throws IOException {
        // 创建输入流
        File file = new File("/Users/kyrie/Documents/doc/private/Netty/netty-study/src/main/res
urces/NIOFileChannelSample01.txt");
```

```
        FileInputStream inputStream = new FileInputStream(file);
        // 通过输入流获取FileChannel
        FileChannel channel = inputStream.getChannel();

        // 创建缓冲区（channel只与缓冲区交互）
        ByteBuffer buffer = ByteBuffer.allocate((int) file.length());

        // 从FileChannel中将数据读取到缓冲区中
        channel.read(buffer);

        // 将byteBuffer的字节数据转成String
        String data = new String(buffer.array());

        // 控制台输出
        System.out.println(data);

        // 关闭流
        inputStream.close();
    }
}
```

- [案例三：使用一个Buffer完成文件读取。要求使用FileChannel的read方法和write方法]

```
package com.mars.example.nio.channel.fileChannel;

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

/**
 * @author: kyrie  @date: 2022/10/21 14:23
 * @desc: 使用一个Buffer完成文件读取
 *          1.使用FileChannel和方法 read、write完成文件的拷贝
 *          2.拷贝一个文本文件1.txt，放在项目下即可
 *          3.代码演示
 */
public class NIOFileChannelSample03 {
    public static void main(String[] args) throws IOException {
        // 创建输入流
        File file = new File("/Users/kyrie/Documents/doc/private/Netty/netty-study/src/main/res
urces/NIOFileChannelSample01.txt");
        FileInputStream inputStream = new FileInputStream(file);
        // 通过输入流获取FileChannel
        FileChannel inputChannel = inputStream.getChannel();

        // 创建输出流
        FileOutputStream outputStream = new FileOutputStream("/Users/kyrie/Documents/doc
private/Netty/netty-study/src/main/resources/NIOFileChannelSample03.txt");
        FileChannel outputChannel = outputStream.getChannel();

        // 写入到缓冲区
```

```
        ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
        while (true) {
            // 每次写入buffer时都重置下标识位，否则会重复读取（很重要：必须要复位）
            byteBuffer.clear();
            int read = inputChannel.read(byteBuffer);
            if (read == -1) {
                break;
            }
            // 读写切换
            byteBuffer.flip();
            int write = outputChannel.write(byteBuffer);
        }
        // 关闭流
        inputStream.close();
        outputChannel.close();
    }
}
```

● [案例四：实现一个图片拷贝，使用transferFrom方法实现]

```
package com.mars.example.nio.channel.fileChannel;

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

/**
 * @author: kyrie  @date: 2022/10/21 14:23
 * @desc: 实现一个图片拷贝，使用transferFrom方法实现
 */
public class NIOFileChannelSample04 {
    public static void main(String[] args) throws IOException {
        // 创建输入流
        FileInputStream inputStream = new FileInputStream("/Users/kyrie/Documents/doc/priva
e/Netty/netty-study/src/main/resources/1.png");
        // 通过输入流获取FileChannel
        FileChannel inputChannel = inputStream.getChannel();

        // 创建输出流
        FileOutputStream outputStream = new FileOutputStream("/Users/kyrie/Documents/doc
private/Netty/netty-study/src/main/resources/1_copy.png");
        FileChannel outputChannel = outputStream.getChannel();
        // 使用transferFrom将源通道的数据拷贝到当前this通道
        outputChannel.transferFrom(inputChannel, 0, inputChannel.size());
        // 关闭流
        inputChannel.close();
        outputChannel.close();
        inputStream.close();
        outputChannel.close();
    }
}
```

4.Buffer和Channel使用注意事项

● ByteBuffer支持类型化的put 和 get，put放入的是什么类型，get 获取时就需要使用对应的类型来，否则可能会抛出异常：java.nio.BufferUnderflowException

```
package com.mars.example.nio.buffer;

import java.nio.ByteBuffer;

/**
 * @author: kyrie  @date: 2022/10/21 17:09
 * @desc: 验证ByteBuffer：put放入的是什么类型，get 获取时就需要使用对应的类型来取，否则可会抛出异常
 */
public class NIOByteBufferPutGetSample {
    public static void main(String[] args) {
        ByteBuffer buffer = ByteBuffer.allocate(64);
        // 插入
        buffer.putInt(10);
        buffer.putLong(9);
        buffer.putChar('k');
        buffer.putDouble(0.09);

        buffer.flip();

        // 按照对应类型取出
        /*System.out.println(buffer.getInt());
        System.out.println(buffer.getLong());
        System.out.println(buffer.getChar());
        System.out.println(buffer.getDouble());*/

        // 未按照对应类型取出，结果抛出: java.nio.BufferUnderflowException
        System.out.println(buffer.getInt());
        System.out.println(buffer.getLong());
        System.out.println(buffer.getLong());
        System.out.println(buffer.getDouble());
    }
}
```

● 可以将一个普通的Buffer转成只读Buffer，对只读的Buffer进行写入操作会抛出异常: java.nio.ReadnlyBufferException

```
package com.mars.example.nio.buffer;

import java.nio.ByteBuffer;

/**
 * @author: kyrie  @date: 2022/10/21 17:17
 * @desc: 验证：将普通Buffer转成只读Buffer
 */
public class NIOByteBufferTransformReadonly {
    public static void main(String[] args) {
        ByteBuffer buffer = ByteBuffer.allocate(64);
```

```
        buffer.putInt(1);
        buffer.flip();

        // 设置Buffer为只读Buffer
        ByteBuffer readOnlyBuffer = buffer.asReadOnlyBuffer();
        System.out.println(readOnlyBuffer.getClass());
        // 只读的Buffer 写入数据会抛异常: java.nio.ReadOnlyBufferException
        readOnlyBuffer.putInt(2);

        while (buffer.hasRemaining()) {
            System.out.println(buffer.getInt());
        }
    }
}
```

● NIO 还提供了 MappedByteBuffer，可以让文件直接在内存(堆外内存)中进行修改，而如何同步到件由NIO来完成

```
package com.mars.example.nio.buffer;

import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;

/**
 * @author: kyrie  @date: 2022/10/21 17:29
 * @desc: 直接内存: MappedBuffer使用,可以让文件直接在内存中修改，即操作系统不需要拷贝一次 - 零拷贝
 */
public class MappedByteBufferSample {
    public static void main(String[] args) throws IOException {
        RandomAccessFile randomAccessFile = new RandomAccessFile("/Users/kyrie/Document/doc/private/Netty/netty-study/src/main/resources/NIOFileChannelSample01.txt", "rw");
        // 获取对应的通道
        FileChannel channel = randomAccessFile.getChannel();

        // TP map函数参数:
        // FileChannel.MapMode.READ_WRITE:模式(读写)
        // 0: 直接修改的起始位置
        // 5: 映射到内存的大小，即将上述文件的多少个字节映射到内存，可以修改的范围为[0~5)
        MappedByteBuffer mappedByteBuffer = channel.map(FileChannel.MapMode.READ_WRIE, 0, 5);
        // 按照索引进行修改
        mappedByteBuffer.put(0, (byte) 'H');
        mappedByteBuffer.put(3, (byte) 'K');
        // 下句会抛异常：java.lang.IndexOutOfBoundsException 5-表示大小，不代表索引
        // mappedByteBuffer.put(5, (byte) 'Y');
        randomAccessFile.close();
        System.out.println("修改成功...");
    }
}
```

● Buffer的分散与聚合：NIO还支持通过多个Buffer（即Buffer数组）完成读写操作，即：Scattering

和 Gathering

```
package com.mars.example.nio.buffer;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.Buffer;
import java.nio.ByteBuffer;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Arrays;

/**
 * @author: kyrie  @date: 2022/10/21 17:44
 * @desc: Buffer的分散与聚合
 * Scattering: 将数据写入到buffer时，可以采用buffer数组，依次写入【Buffer分散】
 * Gathering:  从buffer读取数据时，可以采用buffer数组，依次读【Buffer聚合】
 */
public class ScatteringAndGatheringSample {
    public static void main(String[] args) throws IOException {
        // 使用ServerSocketChannel 和 SocketChannel
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        InetSocketAddress inetSocketAddress = new InetSocketAddress(8888);
        // 绑定端口到socket并启动
        serverSocketChannel.socket().bind(inetSocketAddress);
        // 创建Buffer数组
        ByteBuffer[] byteBuffers = new ByteBuffer[2];
        byteBuffers[0] = ByteBuffer.allocate(5);
        byteBuffers[1] = ByteBuffer.allocate(3);

        // 等待客户端连接
        SocketChannel socketChannel = serverSocketChannel.accept();
        int msgLimit = 8;
        while (true) {
            int byteRead = 0;
            while (byteRead < msgLimit) {
                long read = socketChannel.read(byteBuffers);
                byteRead += read;
                System.out.println("当前累计读取到的字节数为: byteRead=" + byteRead);
                // 使用流打印buffer信息
                Arrays.stream(byteBuffers).map(buffer -> "position=" + buffer.position() +
                        ",limit=" + buffer.limit()).forEach(System.out::println);

                Arrays.asList(byteBuffers).forEach(Buffer::flip);

                long byteWrite = 0;
                while (byteWrite < msgLimit) {
                    // 回写到客户端通道上
                    long write = socketChannel.write(byteBuffers);
                    byteWrite += 1;
                }
                // 复位操作
                Arrays.asList(byteBuffers).forEach(Buffer::clear);
                System.out.println("byteRead=" + byteRead + ",byteWrite=" + byteWrite);
```

```
        }
      }
    }
}
```

5.Selector 的使用

5.1 创建Selector

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

5.2 绑定Channel事件(注册事件)

```
// 必须设置为非阻塞模型，否则抛异常
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, 绑定事件);
```

:::danger

- Channel必须工作在非阻塞模式下
- FileChannel没有非阻塞模式，因此不能配合Selector一起使用
- 绑定的事件类型有以下几种：
  - OP_CONNECT：客户端连接成功时触发
  - OP_ACCEPT：服务器端成功接受连接时触发
  - OP_READ：数据可读入时触发，存在因为接收能力弱，数据暂不能读入的情况
  - OP_WRITE：数据可写入时触发，存在因为发送能力弱，数据暂不能写出的情况

:::

5.3 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

- 方法一：使用 select() ，阻塞直到绑定事件发生

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

- 方法二：使用 select(long timeout)，阻塞直到绑定事件发生或是超时(单位:ms)

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

- 方法三：使用 selectNow()，不会阻塞，不管有没有事件，立刻返回，需程序根据返回值检查是否事件

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

:::warning

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() 方法
    - 调用selector.close() 方法
    - selector 所在线程 interrupt

:::

5.4 处理 accept 事件

```
package com.mars.example.nio.selector;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import java.util.Iterator;

/**
 * @author: kyrie  @date: 2022/11/1 09:53
 * @desc: 处理 accept 事件
 */
public class AcceptEventHandler {

    private static final Logger logger = LoggerFactory.getLogger(AcceptEventHandler.class);

    public static void main(String[] args) {
        try (ServerSocketChannel channel = ServerSocketChannel.open()) {
            channel.bind(new InetSocketAddress(9999));
            Selector selector = Selector.open();
            channel.configureBlocking(false);
            channel.register(selector, SelectionKey.OP_ACCEPT);

            while (true) {
                int count = selector.select();
                logger.info("当前接收到的事件个数为: {}", count);
                // 获取所有的事件
                Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
                while (iterator.hasNext()) {
                    SelectionKey key = iterator.next();
                    if (key.isAcceptable()) {
                        SocketChannel socketChannel = channel.accept();
                        logger.info("当前SocketChannel={}", socketChannel);
                    }
```

```
                iterator.remove();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

- 事件发生后，能否不处理?

:::info

事件发生后，要么处理，要么取消(cancel)，不能什么都不做，斗则下次该事件仍会触发，这是因为NO底层使用的水平触发

:::

5.5 处理 read 事件

```
package com.mars.example.nio.selector;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

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

import static com.mars.example.nio.buffer.ByteBufferUtil.debugAll;

/**
 * @Author: kyrie  @date: 2022/11/1 11:01
 * @Description: 处理 read 事件
 * @Package: com.mars.example.nio.selector
 * @Email: wuxiang@roowoo.cn
 */
public class ReadEventHandler {

    private static final Logger logger = LoggerFactory.getLogger(ReadEventHandler.class);

    public static void main(String[] args) {
        try (ServerSocketChannel serverSocketChannel = ServerSocketChannel.open()) {
            serverSocketChannel.bind(new InetSocketAddress(9999));
            serverSocketChannel.configureBlocking(false);
            Selector selector = Selector.open();
            serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
            while (true) {
                int count = selector.select();
                if (count <= 0) {
                    continue;
                }
                Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
```

```
                while (iterator.hasNext()) {
                    SelectionKey key = iterator.next();
                    if (key.isAcceptable()) {
                        SocketChannel channel = serverSocketChannel.accept();
                        channel.configureBlocking(false);
                        channel.register(selector, SelectionKey.OP_READ);
                        logger.info("客户端: {}，已建立连接", channel);
                    } else if (key.isReadable()) {
                        SocketChannel channel = (SocketChannel) key.channel();
                        ByteBuffer buffer = ByteBuffer.allocate(20);
                        int read = channel.read(buffer);
                        if (-1 == read) {
                            key.cancel();
                            channel.close();
                        } else {
                            buffer.flip();
                            debugAll(buffer);
                        }
                    }
                    iterator.remove();
                }
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

● 为什么要执行 iterator.remove() ?

:::warning

因为select在事件发生后，就会将相关的key放入selectedKeys集合，但不会再处理完后从selectedKeys集合中移除，需要我们自己编码移除:

● 第一次出发了key上的 accept 事件，没有移除key

● 第二次出发了key上的 read　事件，但这时selectedKeys中还有上次的key，再处理时因为没有真的serverSocket连上，就会导致NPE异常

:::

● cancel()方法的作用

:::info

cancel() 会取消注册在selector上的channel，并从keys集合中删除key，后续不在监听该事件

:::