netty-02-IO

1.传统的socket讲解

什么是套接字:

```
Socket又称套接字，是支持TCP/IP的网络通信的基本操作单元，可以看做是不同主机之间的进程进行双向通信的端点，简单的说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程。
非常非常简单的举例说明下:Socket=Ip address+ TCP/UDP + port。
```

套接字之间的连接过程可以分为四个步骤:服务器监听，客户端请求服务器，服务器
确认，客户端确认，进行通信。
(1)服务器监听:是服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的
状态，实时监控网络状态。
(2) 客户端请求:是指由客户端的套接字提出连接请求，要连接的目标是服务器端的套接
字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地
址和端口号，然后就向服务器端套接字提出连接请求。
(3)服务器端连接确认:是指当服务器端套接字监听到或者说接收到客户端套接字的连接
请求，它就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端。
(4) 客户端连接确认: -旦客户端确认了此描述，连接就建立好了。双方开始进通信。而
服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

先看一个最基本的socket的代码示例:
网络编程的基本模型是Client/Server模型，，也就是两个进程直接进行相互通信，其中服务端提供配置信息(绑定的IP地址和监听端口)，客户端通过连接操作向服务端端监听的地址发起连接请求，通过3三次握手建立连接，如果连接成功，则双方即可以进行通信(网络套接字socket)。

- 客户端

```

package xxx.bio;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class Client {

    final static String ADDRESS = "127.0.0.1";
    final static int PORT = 8765;

    public static void main(String[] args) {

        Socket socket = null;
        BufferedReader in = null;
        PrintWriter out = null;

        try {
            socket = new Socket(ADDRESS, PORT);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);

            //向服务器端发送数据
            out.println("接收到客户端的请求数据...");
            out.println("接收到客户端的请求数据1111...");
            String response = in.readLine();
            System.out.println("Client: " + response);

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if(in != null){
                try {
                    in.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(out != null){
                try {
                    out.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            if(socket != null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            socket = null;
        }
```

```
    }
}
```

-服务端

```
package bhz.bio;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;


public class Server {

    final static int PROT = 8765;

    public static void main(String[] args) {

        ServerSocket server = null;
        try {
            server = new ServerSocket(PROT);
            System.out.println(" server start .. ");
            //进行阻塞
            Socket socket = server.accept();
            //新建一个线程执行客户端的任务
            new Thread(new ServerHandler(socket)).start();

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if(server != null){
                try {
                    server.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            server = null;
        }



    }
```

- ServerHandler

```
package bhz.bio;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class ServerHandler implements Runnable{

    private Socket socket ;

    public ServerHandler(Socket socket){
        this.socket = socket;
    }

    @Override
    public void run() {
        BufferedReader in = null;
        PrintWriter out = null;
        try {
            in = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));
            out = new PrintWriter(this.socket.getOutputStream(), true);
            String body = null;
            while(true){
                body = in.readLine();
                if(body == null) break;
                System.out.println("Server :" + body);
                out.println("服务器端回送响的应数据.");
            }

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if(in != null){
                try {
                    in.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(out != null){
                try {
                    out.close();
```

```
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            if(socket != null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            socket = null;
        }


    }

}
```

上面的示例我们可以看到，当我们客户端请求多的时候，那么我们的服务器端会生成大量的线程。那么这么解决？java1.5之前使用的是伪异步也就是使用线程池的方式，并没有提高性能，知识限制了开启线程的数量，防止服务器被线程蹦坏：

代码示例：
client（客户端）

```
package bhz.bio2;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

public class Client {

    final static String ADDRESS = "127.0.0.1";
    final static int PORT =8765;

    public static void main(String[] args) {
        Socket socket = null;
        BufferedReader in = null;
        PrintWriter out = null;
        try {
            socket = new Socket(ADDRESS, PORT);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);

            out.println("Client request");

            String response = in.readLine();
            System.out.println("Client:" + response);


        }  catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } finally {
            if(in != null){
                try {
                    in.close();
                } catch (Exception e1) {
                    e1.printStackTrace();
                }
            }
            if(out != null){
                try {
                    out.close();
                } catch (Exception e2) {
                    e2.printStackTrace();
                }
            }
            if(socket != null){
                try {
                    socket.close();
                } catch (Exception e3) {
                    e3.printStackTrace();
                }
            }
            socket = null;
        }



    }

}
```

server（服务端）

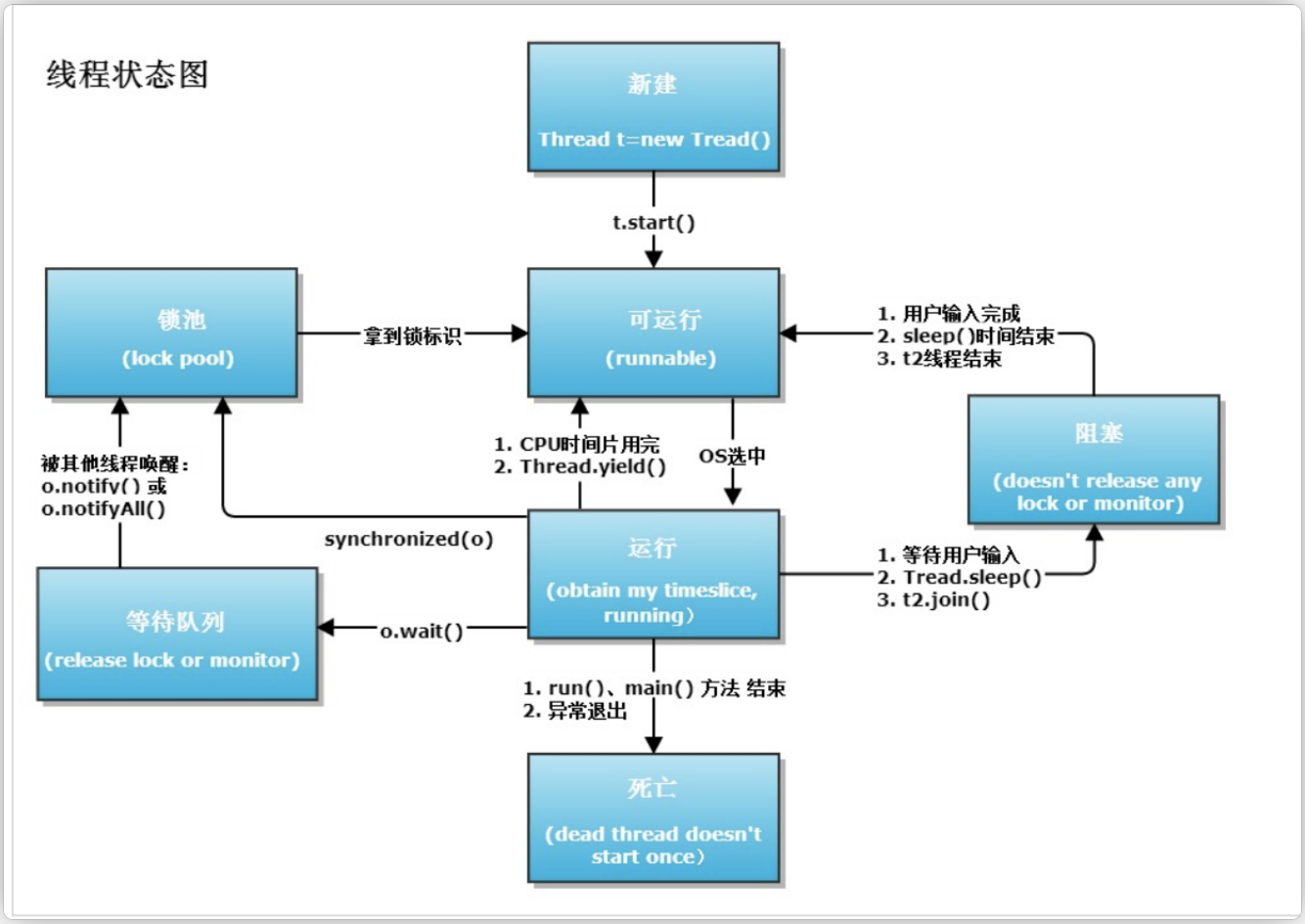
```java
package xxx.bio2;

import java.io.BufferedReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {

    final static int PORT = 8765;

    public static void main(String[] args) {
        ServerSocket server = null;
        BufferedReader in = null;
        PrintWriter out = null;
        try {
            server = new ServerSocket(PORT);
            System.out.println("server start");
            Socket socket = null;
            HandlerExecutorPool executorPool = new HandlerExecutorPool(50, 1000);
            while(true){
                socket = server.accept();
                executorPool.execute(new ServerHandler(socket));
            }

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if(in != null){
                try {
                    in.close();
                } catch (Exception e1) {
                    e1.printStackTrace();
                }
            }
            if(out != null){
                try {
                    out.close();
                } catch (Exception e2) {
                    e2.printStackTrace();
                }
            }
            if(server != null){
                try {
                    server.close();
                } catch (Exception e3) {
                    e3.printStackTrace();
                }
            }
            server = null;
        }



    }
}
```

HandlerExecutoirPool

```java
package xxx.bio2;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class HandlerExecutorPool {

    private ExecutorService executor;
    public HandlerExecutorPool(int maxPoolSize, int queueSize){
        this.executor = new ThreadPoolExecutor(
                Runtime.getRuntime().availableProcessors(),
                maxPoolSize,
                120L,
                TimeUnit.SECONDS,
                new ArrayBlockingQueue<Runnable>(queueSize));
    }

    public void execute(Runnable task){
        this.executor.execute(task);
    }



}
```

我们可以看到解决了线程问题，但是有阻塞：也就是我们一条消息传输过程中，另一条消息要传输只能等待上一条传输完毕才行。如果网速慢，阻塞会很大，这样效率很慢。

线程阻塞图



BIO和NIO和AIO

为了解决阻塞的问题；需要一个新的方法来处理上面的问题，这个方法就是NIO或者AIO
了解什么是AIO和NIO，我们先理解几个概念名称：BIO 同步和异步 阻塞和非阻塞

同步和异步：
同步和异步指的是一个执行流程中每个方法是否必须依赖前一个方法完成后才可以继续执行。假设我们的执行流程中：依次是方法一和方法二。

同步指的是调用一旦开始，调用者必须等到方法调用返回后，才能继续后续的行为。即方法二一定要等到方法一执行完成后才可以执行。
异步指的是调用立刻返回，调用者不必等待方法内的代码执行结束，就可以继续后续的行为。（具体方法内的代码交由另外的线程执行完成后，可能会进行回调）。即执行方法一的时候，直接交给其他线程执行，不由主线程执行，也就不会阻塞主线程，所以方法二不必等到方法一完成即可开始执行。

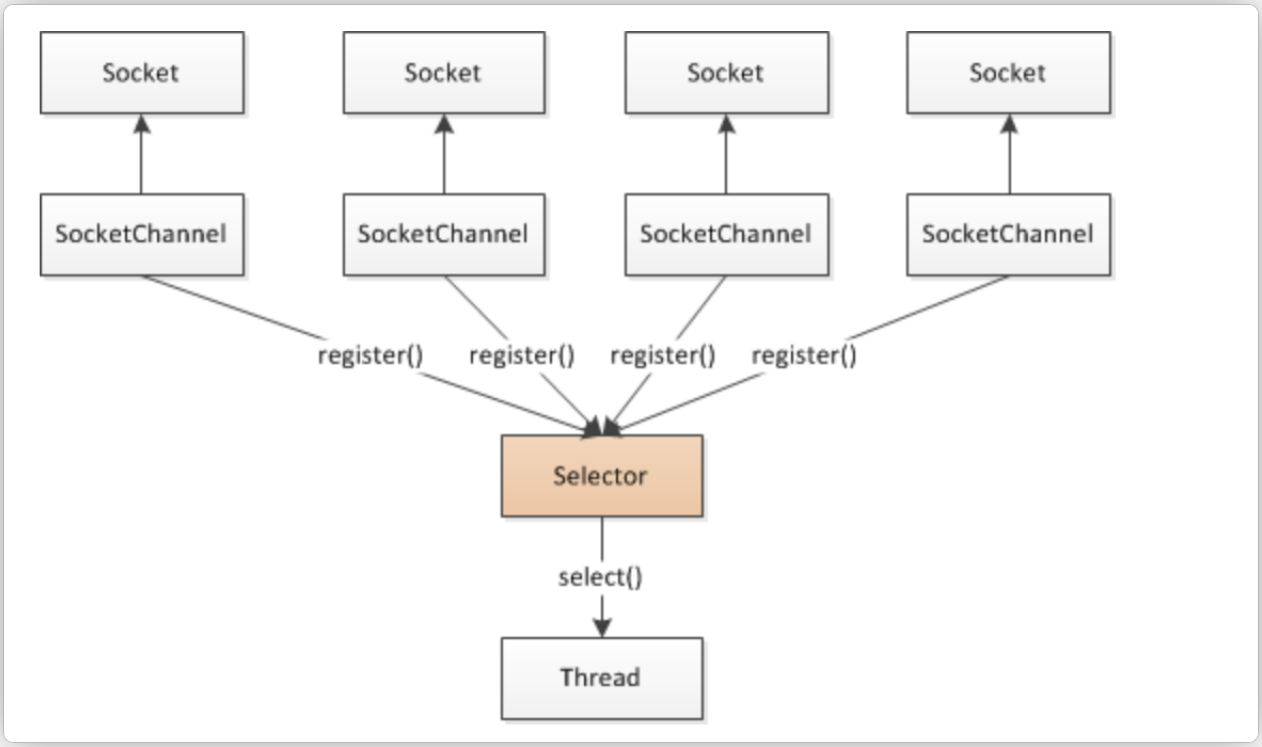
同步与异步关注的是方法的执行方是主线程还是其他线程，主线程的话需要等待方法执行完成，其他线程的话无需等待立刻返回方法调用，主线程可以直接执行接下来的代码。

阻塞与非阻塞
阻塞与非阻塞指的是单个线程内遇到同步等待时，是否在原地不做任何操作。
阻塞指的是遇到同步等待后，一直在原地等待同步方法处理完成。
非阻塞指的是遇到同步等待，不在原地等待，先去做其他的操作，隔断时间再来观察同步方法是否完成。

BIO：我们上面讲的列子其实就是一个bio的例子,BIO全称是Blocking IO，是JDK1.4之前的传统IO模型，本身是同步阻塞模式。 线程发起IO请求后，一直阻塞IO，直到缓冲区数据就绪后，再进入下一步操作。针对网络通信都是一请求一应答的方式，虽然简化了上层的应用开发，但在性能和可靠性方面存在着巨大瓶颈，试想一下如果每个请求都需要新建一个线程来专门处理，那么在高并发的场景下，机器资源很快就会被耗尽。

NIO：
NIO也叫Non-Blocking IO 是同步非阻塞的IO模型。线程发起io请求后，立即返回（非阻塞io）。同步指的是必须等待IO缓冲区内的数据就绪，而非阻塞指的是，用户线程不原地等待IO缓冲区，可以先做一些其他操作，但是要定时轮询检查IO缓冲区数据是否就绪。Java中的NIO 是new IO的意思。其实是NIO加上IO多路复用技术。普通的NIO是线程轮询查看一个IO缓冲区是否就绪，而Java中的new IO指的是线程轮询地去查看一堆IO缓冲区中哪些就绪，这是一种IO多路复用的思想。IO多路复用模型中，将检查IO数据是否就绪的任务，交给系统级别的select或epoll模型，由系统进行监控，减轻用户线程负担。

NIO主要有buffer、channel、selector三种技术的整合，通过零拷贝的buffer取得数据，每一个客户端通过channel在selector（多路复用器）上进行注册。服务端不断轮询channel来获取客户端的信息。channel上有connect,accept（阻塞）、read（可读）、write(可写)四种状态标识。根据标识来进行后续操作。所以一个服务端可接收无限多的channel。不需要新开一个线程。大大提升了性能。



AIO：
AIO是真正意义上的异步非阻塞IO模型。 上述NIO实现中，需要用户线程定时轮询，去检查IO缓冲区数据是否就绪，占用应用程序线程资源，其实轮询相当于还是阻塞的，并非真正解放当前线程，因为它还是需要去查询哪些IO就绪。而真正的理想的异步非阻塞IO应该让内核系统完成，用户线程只需要告诉内核，当缓冲区就绪后，通知我或者执行我交给你的回调函数。

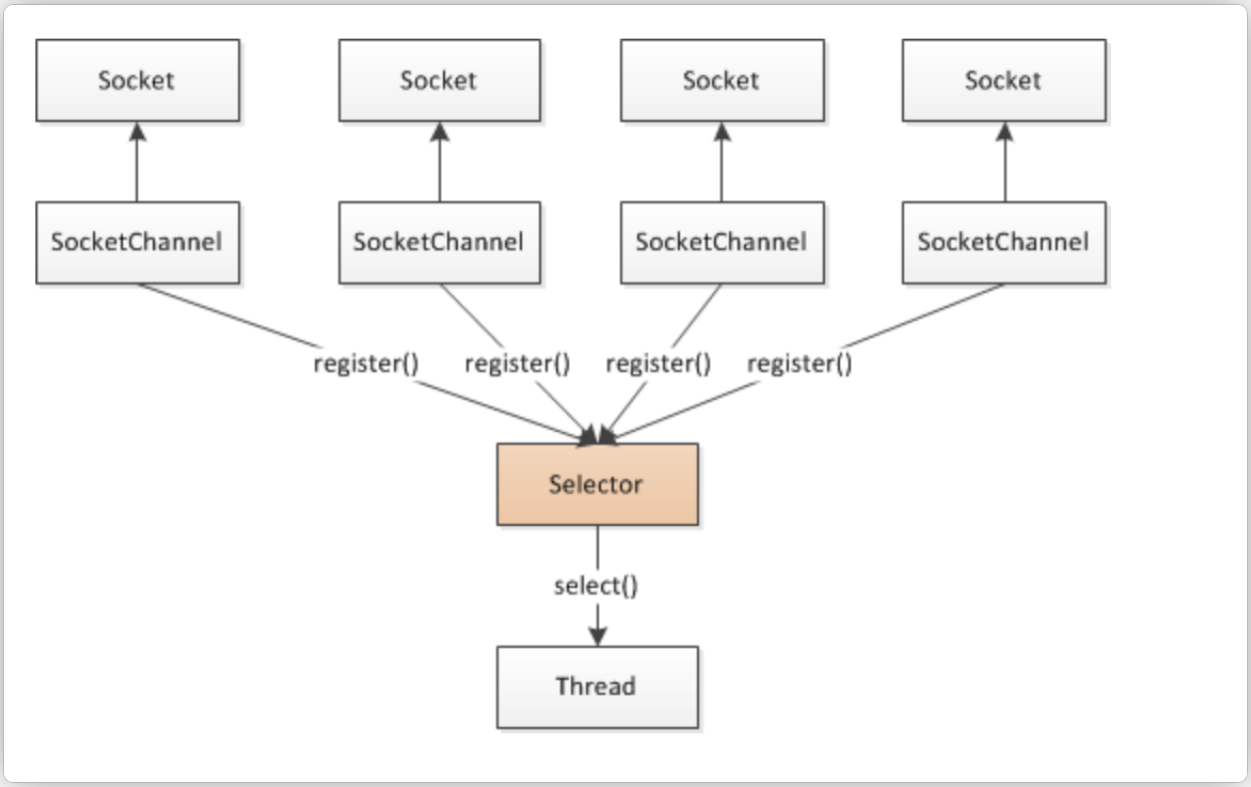
AIO可以做到真正的异步的操作，但实现起来比较复杂，支持纯异步IO的操作系统非常少，目前也就windows是IOCP技术实现了，而在Linux上，底层还是是使用的epoll实现的。

例子讲解
海底捞很好吃，但是经常要排队。我们就以生活中的这个例子进行讲解。
A顾客去吃海底捞，就这样干坐着等了一小时，然后才开始吃火锅。(BIO)
B顾客去吃海底捞，他一看要等挺久，于是去逛商场，每次逛一会就跑回来看有没有排到他。于是他最后既购了物，又吃上海底捞了。（NIO）
C顾客去吃海底捞，由于他是高级会员，所以店长说，你去商场随便玩吧，等下有位置，我立马打电话给你。于是C顾客不用干坐着等，也不用每过一会儿就跑回来看有没有等到，最后也吃上了海底捞（AIO）

2.NIO编程

在介绍NIO之前，先澄清一个概念，有的人叫NIO为new I0,有的人把NIO叫做Non-block IO,这里我们还是习惯说后者，即非阻塞I0.
学习NIO编程，我们首先要了解几个概念:

Buffer (缓冲区)、Channel (管道、通道) Selector (选择器、多路复用器)



Buffer是-一个对象，它包含一些要写入或者要读取的数据。在NIO类库中加入Buffer对象，体现了新库与原I0的一个重要的区别。在面向流的I0中，可以将数据直接写入或读取到Stream对象中。在NIO库中，**所有数据都是用缓冲区处理的(读写)。**缓冲区实质上是一个数组，通常它是一个字节数组(ByteBuffer)，也可以使用其他类型的数组。这个数组为缓冲区提供了数据的访问读写等操作属性，如位置、容量、上限等概念，参考api文档。
Buffer类型:我们最常用的就是ByteBuffer,实际上每一种java基本类型都
对于了一种缓存区(除了Boolean类型)
ByteBuffer、CharBuffer、ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer

使用的demo：
注意使用buf.flip();把位置复位为0，因为每次put我们的位置pos都会加1：
案例：

```
ackage com.netty.socketdemo;

import java.nio.IntBuffer;

public class client {
    public static void main(String[] args) {
       // 1 基本操作

        //创建指定长度的缓冲区
        IntBuffer buf = IntBuffer.allocate(10);
        buf.put(13);// position位置: 0 - > 1
        buf.put(21);// position位置: 1 - > 2
        buf.put(35);// position位置: 2 - > 3

        //把位置复位为0，也就是position位置: 3 - > 0
        buf.flip();
        System.out.println("使用flip复位: " + buf);
        System.out.println("容量为: " + buf.capacity()); //容量一旦初始化后不允许改变（warp方法包裹数组除外）
        System.out.println("限制为: " + buf.limit());        //由于只装载了三个元素,所以可读取或者操作的元素为3 则limit=3


        System.out.println("获取下标为1的元素: " + buf.get(1));
        System.out.println("get(index)方法，position位置不改变: " + buf);
        buf.put(1, 4);
        System.out.println("put(index, change)方法，position位置不变: " + buf);;

        for (int i = 0; i < buf.limit(); i++) {
            //调用get方法会使其缓冲区位置（position）向后递增一位
            System.out.print(buf.get() + "\t");
        }
        System.out.println("buf对象遍历之后为: " + buf);


        // 2 wrap方法使用

         //  wrap方法会包裹一个数组: 一般这种用法不会先初始化缓存对象的长度，因为没有意义，最后还会被wrap所包裹的数组覆盖掉。
         //  并且wrap方法修改缓冲区对象的时候，数组本身也会跟着发生变化。
```

```
         int[] arr = new int[]{1,2,5};
         IntBuffer buf1 = IntBuffer.wrap(arr);
         System.out.println(buf1);

         IntBuffer buf2 = IntBuffer.wrap(arr, 0 , 2);
         //这样使用表示容量为数组arr的长度，但是可操作的元素只有实际进入缓存区的元素长度
         System.out.println(buf2);




        // 3 其他方法

         IntBuffer buff= IntBuffer.allocate(10);
         int[] arr1 = new int[]{1,2,5};
        buff.put(arr1);
         System.out.println(buff);
         //一种复制方法
         IntBuffer buf3 = buff.duplicate();
         System.out.println(buf3);

         //设置buf2的位置属性
         //buf2.position(0);
        buff.flip();
         System.out.println(buff);

         System.out.println("可读数据为: " + buff.remaining());

         int[] arr3 = new int[buff.remaining()];
         //将缓冲区数据放入arr2数组中去
        buff.get(arr3);
         for(int i : arr3){
         System.out.print(Integer.toString(i) + ",");
         }

    }
}

```

Buffer的flip()方法详解

原文链接：https://blog.csdn.net/u013096088/article/details/78638245
我们知道了，Buffer既可以用来读和写。如下：

```
public class NioTest {
    public static void main(String[] args) {
        // 分配内存大小为10的缓存区
        IntBuffer buffer = IntBuffer.allocate(10);
        // 往buffer里写入数据
        for (int i = 0; i < 5; ++i) {
            int randomNumber = new SecureRandom().nextInt(20);
            buffer.put(randomNumber);
        }
        // 将Buffer从写模式切换到读模式（必须调用这个方法）
        buffer.flip();
        // 读取buffer里的数据
        while (buffer.hasRemaining()) {
            System.out.println(buffer.get());
        }
    }
}
```

这里有个关键的方法flip(),buffer读写转换全靠它来实现。看看这个方法做了什么，进入jdk源码，如下：

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

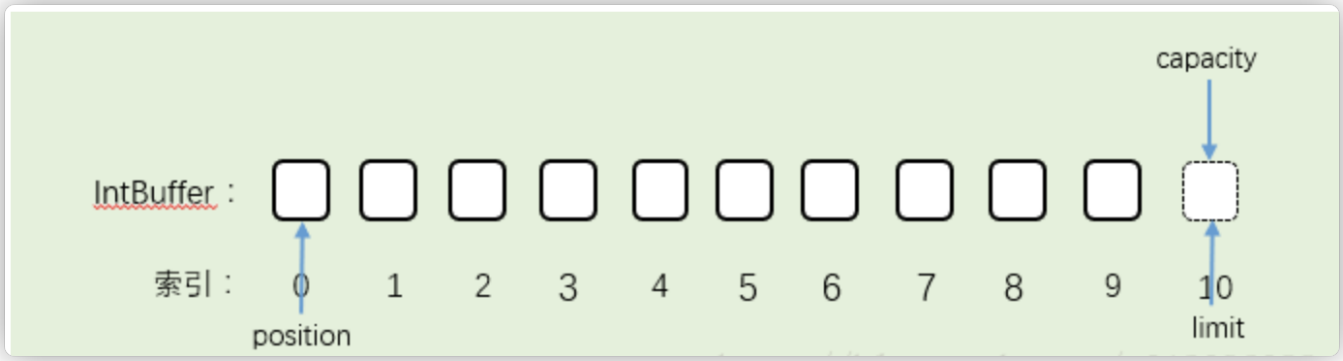
这里涉及的到Buffer的几个重要属性：position,limit,mark。所以，我们先得把这几个属性搞明白。

capacity,limit,position三个重要属性的含义

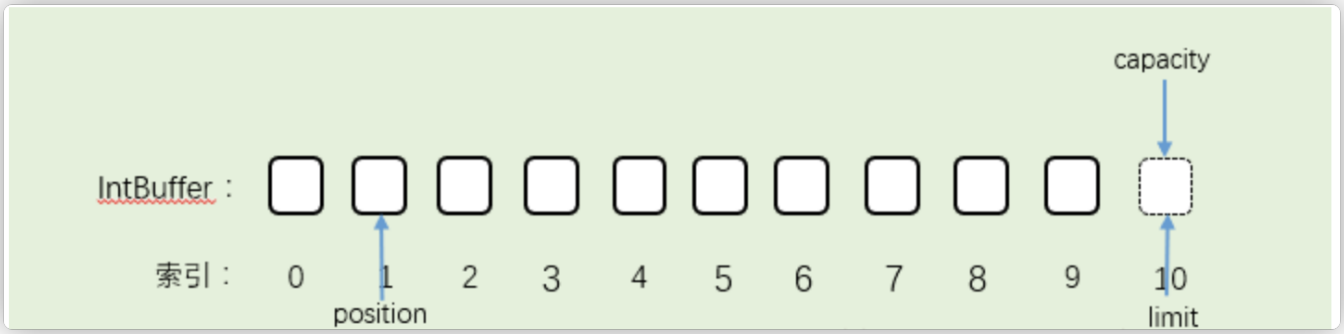
- Buffer是特定基本类型元素的线性有限序列。除内容外，Buffer区的基本属性还包括capacity(容量)、limit(限制)和position(位置)：
- capacity是它所包含的元素的数量。缓冲区的容量不能为负并且不能更改。
- limit是第一个不应该读取或写入的元素的索引。缓冲区的限制不能为负，并且不能大于其容量。
- position是下一个要读取或写入的元素的索引。缓冲区的位置不能为负，并且不能大于其限制。
-

三个属性的关系与图解
拿第一个例子来分析，这三个属性的作用。
1.分配内存大小为10的缓存区。索引10的空间是我虚设出来，实际不存在，为了能明显表示capacity。IntBuffer的容量为10，所以capacity为10，在这里指向索引为10的空间。Buffer初始化

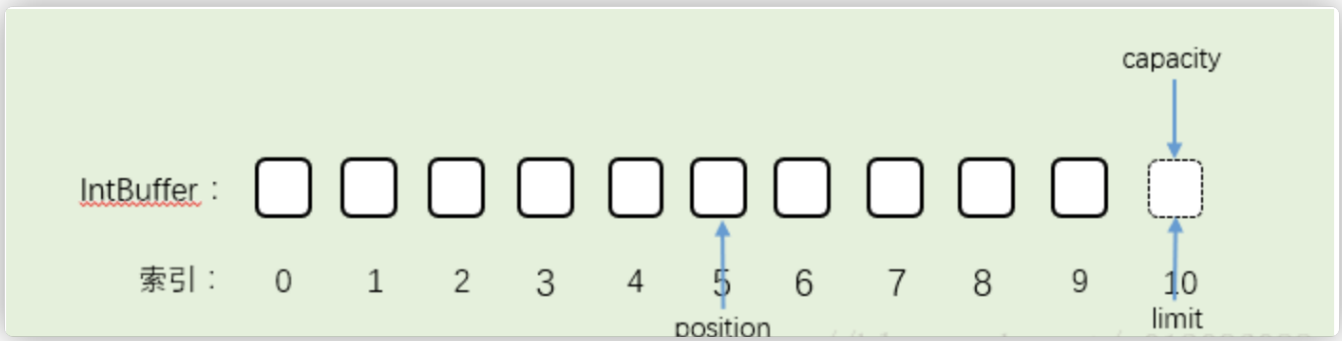
的时候，limit和capacity指向同一索引。position指向0。



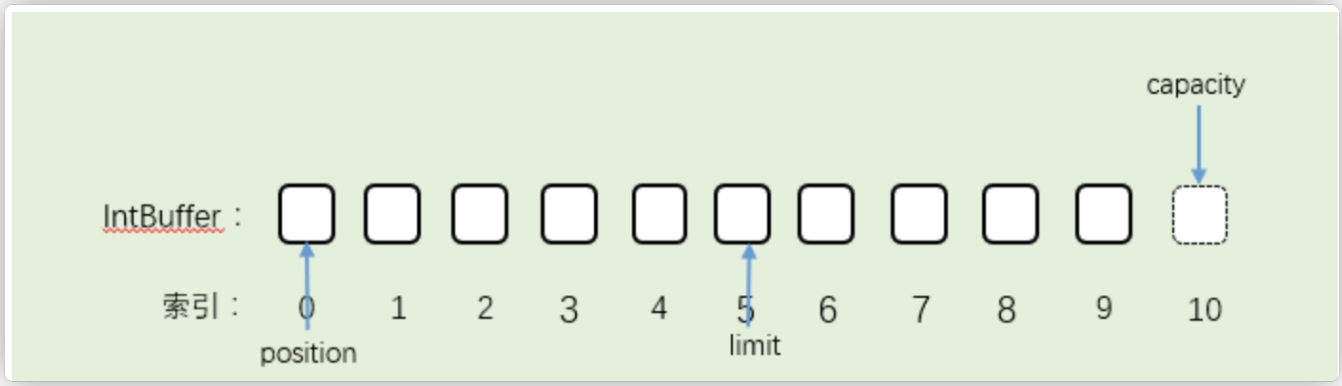
2.往Buffer里加一个数据。position位置移动，capacity不变，limit不变。



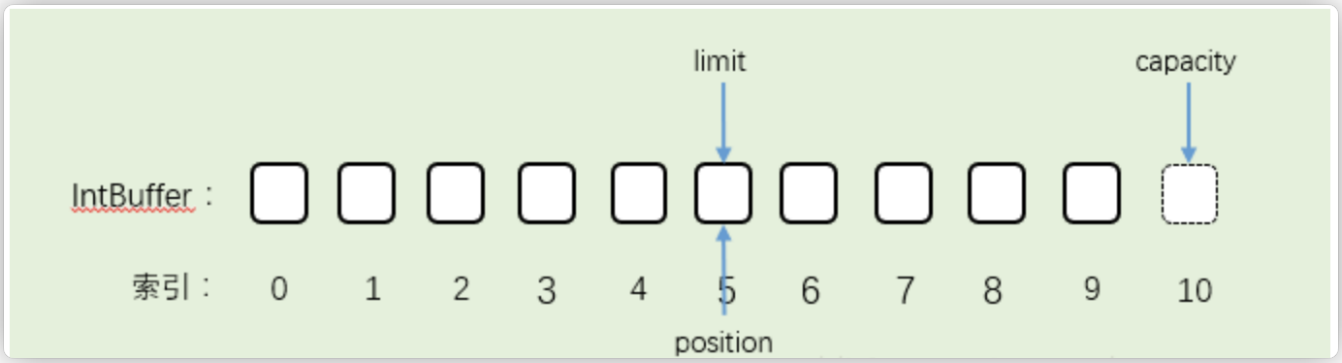
3.Buffer读完之后，往bufer里写了5个数据，position指向索引为5的第6个数据，capacity不变，limit不变。



4.执行flip()。这时候对照着，之前flip源码去看。把position的值赋给limit,所以limit=5，然后position=0。capacity不变。结果就是：



5.Buffer开始往外写数据。每写一个,position就下移一个位置，一直移到limit的位置，结束。



上图的顺序就是代码中的IntBuffer从初始化，到读数据，再写数据三个状态下，capacity,position,limit三个属性的变化和关系。
大家可以发现：

1. 0 <= position <= limit <= capacity
2. capacity始终不变

图中很好的阐述了，Buffer读写切换的过程。即flip()的反转原理。接下来我们从代码中检测上面的分析过程。想一下下面代码打印的内容，然后执行一编代码看看对不对。

```
public class NioTest {
    public static void main(String[] args) {
        // 分配内存大小为10的缓存区
        IntBuffer buffer = IntBuffer.allocate(10);

        System.out.println("capacity:" + buffer.capacity());

        for (int i = 0; i < 5; ++i) {
            int randomNumber = new SecureRandom().nextInt(20);
            buffer.put(randomNumber);
        }

        System.out.println("before flip limit:" + buffer.limit());

        buffer.flip();

        System.out.println("after flip limit:" + buffer.limit());

        System.out.println("enter while loop");

        while (buffer.hasRemaining()) {
            System.out.println("position:" + buffer.position());
            System.out.println("limit:" + buffer.limit());
            System.out.println("capacity:" + buffer.capacity());
            System.out.println("元素:" + buffer.get());
        }
    }
}

```

nio和io有什么区别

NIO vs IO之间的理念上面的区别（NIO将阻塞交给了后台线程执行）
IO是面向流的，NIO是面向缓冲区的
Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方；
NIO则能前后移动流中的数据，因为是面向缓冲区的
IO流是阻塞的，NIO流是不阻塞的
Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了
Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。NIO可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的代价是解析数据可能会比从一个阻塞流中读取数据更复杂。
非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。
选择器
Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来"选择"通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。
来源： https://blog.csdn.net/evan_man/article/details/50910542

nio案例

知道了buffer的使用，我们来看一个基本的NIO案例：
server.java

```
package xxx.nio;

import java.io.IOException;
import java.net.InetSocketAddress;
```

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

public class Server implements Runnable{
    //1 多路复用器（管理所有的通道）
    private Selector seletor;
    //2 建立缓冲区
    private ByteBuffer readBuf = ByteBuffer.allocate(1024);
    //3
    private ByteBuffer writeBuf = ByteBuffer.allocate(1024);
    public Server(int port){
        try {
            //1 打开路复用器
            this.seletor = Selector.open();
            //2 打开服务器通道
            ServerSocketChannel ssc = ServerSocketChannel.open();
            //3 设置服务器通道为非阻塞模式
            ssc.configureBlocking(false);
            //4 绑定地址
            ssc.bind(new InetSocketAddress(port));
            //5 把服务器通道注册到多路复用器上，并且监听阻塞事件
            ssc.register(this.seletor, SelectionKey.OP_ACCEPT);

            System.out.println("Server start, port :" + port);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        while(true){
            try {
                //1 必须要让多路复用器开始监听
                this.seletor.select();
                //2 返回多路复用器已经选择的结果集
                Iterator<SelectionKey> keys = this.seletor.selectedKeys().iterator();
                //3 进行遍历
                while(keys.hasNext()){
                    //4 获取一个选择的元素
                    SelectionKey key = keys.next();
                    //5 直接从容器中移除就可以了
                    keys.remove();
                    //6 如果是有效的
                    if(key.isValid()){
                        //7 如果为阻塞状态
                        if(key.isAcceptable()){
                            this.accept(key);
                        }
                        //8 如果为可读状态
                        if(key.isReadable()){
                            this.read(key);
                        }
                        //9 写数据
                        if(key.isWritable()){
                            //this.write(key); //ssc
                        }
                    }

                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    private void write(SelectionKey key){
        //ServerSocketChannel ssc =  (ServerSocketChannel) key.channel();
        //ssc.register(this.seletor, SelectionKey.OP_WRITE);
    }

    private void read(SelectionKey key) {
        try {
            //1 清空缓冲区旧的数据
            this.readBuf.clear();
            //2 获取之前注册的socket通道对象
            SocketChannel sc = (SocketChannel) key.channel();
            //3 读取数据
            int count = sc.read(this.readBuf);
            //4 如果没有数据
            if(count == -1){
                key.channel().close();
                key.cancel();
                return;
            }
            //5 有数据则进行读取 读取之前需要进行复位方法(把position 和limit进行复位)
            this.readBuf.flip();
```

```
            //6 根据缓冲区的数据长度创建相应大小的byte数组，接收缓冲区的数据
            byte[] bytes = new byte[this.readBuf.remaining()];
            //7 接收缓冲区数据
            this.readBuf.get(bytes);
            //8 打印结果
            String body = new String(bytes).trim();
            System.out.println("Server : " + body);

            // 9..可以写回给客户端数据

        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    private void accept(SelectionKey key) {
        try {
            //1 获取服务通道
            ServerSocketChannel ssc =  (ServerSocketChannel) key.channel();
            //2 执行阻塞方法
            SocketChannel sc = ssc.accept();
            //3 设置阻塞模式
            sc.configureBlocking(false);
            //4 注册到多路复用器上，并设置读取标识
            sc.register(this.seletor, SelectionKey.OP_READ);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {

        new Thread(new Server(8765)).start();;
    }


}
```

Client端

```
ackage xxx.nio;

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

public class Client {

    //需要一个Selector
    public static void main(String[] args) {

        //创建连接的地址
        InetSocketAddress address = new InetSocketAddress("127.0.0.1", 8765);

        //声明连接通道
        SocketChannel sc = null;

        //建立缓冲区
        ByteBuffer buf = ByteBuffer.allocate(1024);

        try {
            //打开通道
            sc = SocketChannel.open();
            //进行连接
            sc.connect(address);

            while(true){
                //定义一个字节数组，然后使用系统录入功能：
                byte[] bytes = new byte[1024];
                System.in.read(bytes);

                //把数据放到缓冲区中
                buf.put(bytes);
                //对缓冲区进行复位
                buf.flip();
                //写出数据
                sc.write(buf);
                //清空缓冲区数据
                buf.clear();
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(sc != null){
                try {
                    sc.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
```

```
            }
        }

    }

}
```

3.AIO

AIO编程，在NIO基础之上引入了异步通道的概念，并提供了异步文件和异步套接字通道的实现，从而在真正意义上实现了异步非阻塞，之前我们学习的NIO只是非阻塞而并非异步。而AIO它不需要通过多路复用器对注册的通道进行轮询操作即可实现异步读写，从而简化了NIO编程模型。也可以称之为NIO2.0,这种模式才真正的属于我们异步非阻塞的模型。
AsynchronousServerScoketChannel
AsynchronousScoketChanel

案例：

- server

```
package com.netty.aiodemo;

import java.net.InetSocketAddress;
import java.nio.channels.AsynchronousChannelGroup;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class server {
    //线程池
    private ExecutorService executorService;
    //线程组
    private AsynchronousChannelGroup threadGroup;
    //服务器通道
    public AsynchronousServerSocketChannel assc;

    public server(int port){
        try {
            //创建一个缓存池
            executorService = Executors.newCachedThreadPool();
            //创建线程组
            threadGroup = AsynchronousChannelGroup.withCachedThreadPool(executorService, 1);
            //创建服务器通道
            assc = AsynchronousServerSocketChannel.open(threadGroup);
            //进行绑定
            assc.bind(new InetSocketAddress(port));

            System.out.println("server start , port : " + port);
            //进行阻塞
            assc.accept(this, new serverHandler());
            //一直阻塞 不让服务器停止
            Thread.sleep(Integer.MAX_VALUE);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        server serve = new server(8765);
    }
}

```

- serverHandle

```java
package com.netty.aiodemo;

import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.util.concurrent.ExecutionException;


public class serverHandler implements CompletionHandler<AsynchronousSocketChannel, server> {
    @Override
    public void completed(AsynchronousSocketChannel asc, server attachment) {
        //当有下一个客户端接入的时候 直接调用Server的accept方法，这样反复执行下去，保证多个客户端都可以阻塞
        attachment.assc.accept(attachment, this);
        read(asc);
    }

    private void read(final AsynchronousSocketChannel asc) {
        //读取数据
        ByteBuffer buf = ByteBuffer.allocate(1024);
        asc.read(buf, buf, new CompletionHandler<Integer, ByteBuffer>() {
            @Override
```

```
            public void completed(Integer resultSize, ByteBuffer attachment) {
                //进行读取之后,重置标识位
                attachment.flip();
                //获得读取的字节数
                System.out.println("Server -> " + "收到客户端的数据长度为:" + resultSize);
                //获取读取的数据
                String resultData = new String(attachment.array()).trim();
                System.out.println("Server -> " + "收到客户端的数据信息为:" + resultData);
                String response = "服务器响应, 收到了客户端发来的数据: " + resultData;
                write(asc, response);
            }
            @Override
            public void failed(Throwable exc, ByteBuffer attachment) {
                exc.printStackTrace();
            }
        });
    }

    private void write(AsynchronousSocketChannel asc, String response) {
        try {
            ByteBuffer buf = ByteBuffer.allocate(1024);
            buf.put(response.getBytes());
            buf.flip();
            asc.write(buf).get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }

    public void failed(Throwable exc, server attachment) {
        exc.printStackTrace();
    }
}

```

- client

```java
package com.netty.socketdemo;

import java.nio.IntBuffer;

public class client {
    public static void main(String[] args) {
       // 1 基本操作

        //创建指定长度的缓冲区
        IntBuffer buf = IntBuffer.allocate(10);
        buf.put(13);// position位置: 0 - > 1
        buf.put(21);// position位置: 1 - > 2
        buf.put(35);// position位置: 2 - > 3

        //把位置复位为0, 也就是position位置: 3 - > 0
        buf.flip();
        System.out.println("使用flip复位: " + buf);
        System.out.println("容量为: " + buf.capacity()); //容量一旦初始化后不允许改变 (warp方法包裹数组除外)
        System.out.println("限制为: " + buf.limit());        //由于只装载了三个元素,所以可读取或者操作的元素为3 则limit=3


        System.out.println("获取下标为1的元素: " + buf.get(1));
        System.out.println("get(index)方法, position位置不改变: " + buf);
        buf.put(1, 4);
        System.out.println("put(index, change)方法, position位置不变: " + buf);;

        for (int i = 0; i < buf.limit(); i++) {
            //调用get方法会使其缓冲区位置 (position) 向后递增一位
            System.out.print(buf.get() + "\t");
        }
        System.out.println("buf对象遍历之后为: " + buf);


        // 2 wrap方法使用

         //  wrap方法会包裹一个数组: 一般这种用法不会先初始化缓存对象的长度, 因为没有意义, 最后还会被wrap所包裹的数组覆盖掉。
         //  并且wrap方法修改缓冲区对象的时候, 数组本身也会跟着发生变化。
         int[] arr = new int[]{1,2,5};
         IntBuffer buf1 = IntBuffer.wrap(arr);
         System.out.println(buf1);

         IntBuffer buf2 = IntBuffer.wrap(arr, 0 , 2);
         //这样使用表示容量为数组arr的长度, 但是可操作的元素只有实际进入缓存区的元素长度
         System.out.println(buf2);




        // 3 其他方法

         IntBuffer buff= IntBuffer.allocate(10);
         int[] arr1 = new int[]{1,2,5};
```

```
        buff.put(arr1);
         System.out.println(buff);
         //一种复制方法
         IntBuffer buf3 = buff.duplicate();
         System.out.println(buf3);

         //设置buf2的位置属性
         //buf2.position(0);
        buff.flip();
         System.out.println(buff);

         System.out.println("可读数据为: " + buff.remaining());

         int[] arr3 = new int[buff.remaining()];
         //将缓冲区数据放入arr2数组中去
        buff.get(arr3);
         for(int i : arr3){
         System.out.print(Integer.toString(i) + ",");
         }

    }
}

```