# NIO

## 1.概述

NIO的作用：传输数据

### 1.2.分类BIO NIO AIO

BIO - Blocking IO - 同步阻塞式IO

NIO - New IO - NonBlocking IO - 同步非阻塞式IO - JDK1.4

AIO - Asynchronous IO - 异步非阻塞式IO - JDK1.7 - AIO是基于NIO进行了改进，也把AIO称之为NIO.2；因为NIO出现时间比较早，所以市面上很多框架的底层都是基于NIO构建，对NIO进行了改进，导致AIO在市面上的占有率并不高

### 1.3.同步异步阻塞的基本概念

同步和异步：如果一个对象或者一段逻辑在一个时间段内允许被多个线程同时使用，此时称之为异步；如果一个对象或者一段逻辑在一个时间段内只允许被一个线程使用，此时称之为同步。

阻塞和非阻塞：一个线程在没有获取到结果之前会持续等待也不执行逻辑也不抛出异常，这种现象称之为阻塞；一个线程即使没有获取到结果也会继续往下执行或者抛出异常，此时称之为非阻塞。

NIO的三大组件：Buffer、Channel、Selector

## 2.BIO的缺点

### 2.1.阻塞：导致任务的执行效率变低

一对一连接：每当客户端产生一个连接，服务器端都需要产生一个线程去处理这个连接；如果产生了大量的客户端连接，服务器需要产生大量线程去处理这些连接；服务器端所能产生和承载的线程数量是有限的，如果线程过多会导致服务器的卡顿甚至崩溃。

如果客户端连接之后不做任何操作而是恶意保持连接，导致服务器端的线程无法释放；如果产生大量的恶意连接，导致服务器端的线程被大量占用

## 3.Buffer - 缓冲区

### 3.1.作用：存储数据

基于数组来实现的。Buffer针对基本类型来进行存储：ByteBuffer，ShortBuffer，IntBuffer，LongBuffer，FloatBuffer，DoubleBuffer，CharBuffer

### 3.2.重要的三个位置

capacity >= limit >= position

capacity：容量位。用于表示缓冲区的容量，指定之后大小不能变

limit：限制位。用于限制position所能达到的最大下标。当limit和position重合的时候，就表示所有元素已经遍历完毕。当缓冲区刚创建的时候，limit默认和capacity重合。

position：操作位。用于指向要读写的位置的。在缓冲区刚创建的时候，position默认为0。当对缓冲区进行读写操作的时候，position自动后挪。

### 3.3.ByteBuffer的数据操作测试

#### ByteBufferDemo01(三个位置关系)

package cn.tedu.buffer;

import org.junit.Test;  
  
import java.nio.ByteBuffer;  
  
public class ByteBufferDemo01 {  
  
 */\*\*  
 \* 测试ByteBuffer的数据存储方式  
 \* 以及各个位置的概念  
 \* capacity 容量位  
 \* limit 限制位，默认创建与容量位重合  
 \* position 操作位默认为0，存入或读取数据自动后挪  
 \*/* @Test  
 public void test01(){  
  
 ByteBuffer byteBuffer = ByteBuffer.*allocate*(10);  
  
 //获取当前缓冲区的容量值  
 int capacity = byteBuffer.capacity();  
 System.*out*.println(capacity);  
  
 //往缓冲区里存放数据  
 byteBuffer.put("abc".getBytes());  
 byteBuffer.put((byte) 0);  
 byteBuffer.put("def".getBytes());  
  
 /\*  
 手动遍历缓冲区  
 1.将limit挪动到position位置  
 2.将position位置归0  
 3.循环读取[position,limit)的位置  
  
 position limit  
 | |  
 [ a , b , c , d , e , f , 0 , 0 , 0 , 0 ]  
 |  
 capacity  
 \*/  
 byteBuffer.limit(byteBuffer.position());  
 byteBuffer.position(0);  
  
 while(byteBuffer.position()<byteBuffer.limit()){  
 byte b = byteBuffer.get();  
 System.*out*.println(b);  
 }  
  
 }  
  
 */\*\*  
 \* 使用byteBuffer提供的api遍历缓冲区  
 \*/* @Test  
 public void test02(){  
 ByteBuffer byteBuffer = ByteBuffer.*allocate*(10);  
  
 //获取当前缓冲区的容量值  
 int capacity = byteBuffer.capacity();  
 System.*out*.println(capacity);  
  
 //往缓冲区里存放数据  
 byteBuffer.put("abc".getBytes());  
 byteBuffer.put((byte) 0);  
 byteBuffer.put("def".getBytes());  
  
 //limit放到position上，position归零  
 byteBuffer.flip();  
  
 //判断position位置是否已在limit-1的位置  
 while(byteBuffer.hasRemaining()){  
 byte b = byteBuffer.get();  
 System.*out*.println(b);  
 }  
  
 }  
}

#### ByteBufferDemo02(数据定义场景)

package cn.tedu.buffer;

import org.junit.Test;  
  
import java.nio.ByteBuffer;  
  
public class ByteBufferDemo02 {  
  
  
 */\*\*  
 \* 测试将缓冲区转为字节数组  
 \* 然后再转为字符串进行输出  
 \* 在已知数据的场景  
 \*/* @Test  
 public void test01() {  
  
 //数据已知情况下直接使用字节数组创建缓冲区  
 ByteBuffer byteBuffer =  
 ByteBuffer.*wrap*("hello world".getBytes());  
 /\*  
 如果数据是以知的可以直接使用数据来创建缓冲区  
 当前缓冲区的limit跟capacity重合,由于是已知的  
 数据，所以读取时不会出现冗余字段直接整个数组  
 取出数据即可  
  
 \*/  
 byte[] data = byteBuffer.array();  
 String hello =  
 new String(data, 0, byteBuffer.limit());  
 System.*out*.println(hello);  
 }  
  
 */\*\*  
 \* 测试将缓冲区转为字节数组  
 \* 然后再转为字符串进行输出  
 \* 在未知数据的场景  
 \*/* @Test  
 public void test02(){  
  
 //创建缓冲区指定长度，并放入数据  
 ByteBuffer byteBuffer =  
 ByteBuffer.*allocate*(10);  
 byteBuffer.put("hello".getBytes());  
  
 //limit设置到position位置，然后position归0  
 byteBuffer.flip();  
  
 //将缓冲区转为字节数组，之后转为字符串输出  
 byte[] data = byteBuffer.array();  
 String hello =  
 new String(data, 0, byteBuffer.limit());  
 System.*out*.println(hello);  
 }  
}

### 3.4.常用API

|  |  |
| --- | --- |
| 方法 | 作用 |
| allocate(int capacity) | 创建缓冲区的时候指定缓冲区容量的大小，实际上是指定缓冲区底层的字节数组的大小 |
| wrap(byte[] array) | 利用传入的字节数组来构建缓冲区 |
| array() | 获取缓冲区底层的字节数组 |
| get() | 获取缓冲区中position位置上的字节 |
| get(byte[] dst) | 将缓冲区中的数据写到传入的字节数组中 |
| get(int index) | 获取指定下标上的字节 |
| put(byte b) | 向position位置上放入指定的字节 |
| put(byte[] src) | 向position位置上放入指定的字节数组 |
| put(byte[] src, int offset, int length) | 向position位置上放入指定的字节数组的部分元素 |
| put(ByteBuffer src) | 将字节缓冲区放入 |
| put(int index, byte b) | 向指定位置插入指定的字节 |
| capacity() | 获取容量位 |
| clear() | 清空缓冲区：position = 0; limit = capacity; mark = -1; |
| flip() | 反转缓冲区：limit = position; position = 0; mark = -1; |
| hasRemaing() | 判断position和limit之间是否还有空余 |
| limit() | 获取限制位 |
| limit(int newLimit) | 设置限制位 |
| mark() | 设置标记位 |
| position() | 获取操作位 |
| position(int newPosition) | 设置操作位 |
| remaining() | 获取position和limit之间剩余的元素个数 |
| reset() | 重置缓冲区：position = mark |
| rewind() | 重绕缓冲区：position = 0; mark = -1 |

## 4.Channel - 通道

### 4.1.作用：传输数据

Channel在传输的时候是针对缓冲区进行操作

### 4.2.常用的Channel

文件：FileChannel

UDP：DatagramChannel

TCP：SocketChannel，ServerSocketChannel

Channel默认是阻塞的，手动设置为非阻塞

Channel可以实现双向传输

### 4.3.使用TCP-Channel通道进行传输测试

#### TCPChannelClient

package cn.tedu.channel;

import org.junit.Test;  
  
import java.io.IOException;  
import java.net.InetSocketAddress;  
import java.nio.ByteBuffer;  
import java.nio.channels.SocketChannel;  
  
  
public class TCPChannelClient {  
  
 */\*\*  
 \* 创建TCP客户端的Channel  
 \* 发送hello server  
 \*/* @Test  
 public void test01() throws IOException {  
  
 //开启客户端通道  
 SocketChannel socketChannel =  
 SocketChannel.*open*();  
 /\*  
 发起连接因为是非阻塞的，  
 所以无论是否建立连接，都会继续往下执行  
 \*/  
 socketChannel.configureBlocking(false);  
  
 //连接到服务端127.0.0.1:8090  
 socketChannel.connect(  
 new InetSocketAddress("127.0.0.1",8090));  
 /\*  
 判断连接是否建立  
 如果连接多次没有成功，那么说明连接无法建立  
 抛出异常  
 \*/  
 while(!socketChannel.isConnected()){  
 /\*  
 如果没有建立连接，试图再次建立  
 finishConnect方法底层自动计数  
 如果计数多次依然没有建立连接，抛出异常  
 \*/  
 socketChannel.finishConnect();  
 }  
  
 //建立连接传输字节数据  
 socketChannel  
 .write(ByteBuffer.*wrap*("hello server".getBytes()));  
 //关流  
 socketChannel.close();  
 }  
  
}

#### TCPChannelServer

package cn.tedu.channel;

import org.junit.Test;  
  
import java.io.IOException;  
import java.net.InetSocketAddress;  
import java.nio.ByteBuffer;  
import java.nio.channels.ServerSocketChannel;  
import java.nio.channels.SocketChannel;  
  
public class TCPChannelServer {  
  
 */\*\*  
 \* 测试基于TCP写的Channel通道  
 \* 接受客户端的数据  
 \*/* @Test  
 public void test01() throws IOException {  
  
 //创建服务端通道，绑定8090端口  
 ServerSocketChannel serverSocketChannel =  
 ServerSocketChannel.*open*();  
  
 serverSocketChannel.bind(  
 new InetSocketAddress(8090));  
  
 //设置为非阻塞的  
 serverSocketChannel.configureBlocking(false);  
  
 SocketChannel socketChannel =null;  
  
 /\*  
 循坏接受客户端的数据，这里进行手动的  
 阻塞，一直等到服务端发送了数据再接受数据  
 \*/  
 while (socketChannel == null) {  
  
 socketChannel = serverSocketChannel.accept();  
 }  
  
 //指定缓冲流的容量  
 ByteBuffer byteBuffer = ByteBuffer.*allocate*(1024);  
 socketChannel.read(byteBuffer);  
  
 String result = new String(  
 byteBuffer.array(),  
 0, byteBuffer.position());  
  
 //将结果打印  
 System.*out*.println(result);  
  
 //关闭服务通道  
 serverSocketChannel.close();  
 }  
  
}

## 5.Selector - 多路复用选择器

### 5.1.作用：针对通道的指定事件来进行选择

Selector在使用的时候针对非阻塞通道进行操作

计算机生成了可选文字:
connect 
read 
while(true) ； 
Selector 
accept 
read 

### 5.2.基于Selector传输器的TCP-Channel

#### TCPSelectorChannelClient

package cn.tedu.selector;

import org.junit.Test;  
  
import java.io.IOException;  
import java.net.InetSocketAddress;  
import java.nio.ByteBuffer;  
import java.nio.channels.SocketChannel;  
  
public class TCPSelectorChannelClient {  
  
 */\*\*  
 \* 定义客户端进行数据的发送以及数据的读取  
 \*/* @Test  
 public void test01() throws IOException {  
  
 //创建客户端通道，监听绑定127.0.0.1:8090  
 SocketChannel socketChannel =  
 SocketChannel.*open*();  
  
 socketChannel.connect(  
 new InetSocketAddress("localhost",8090));  
  
 //如第一次连接不正常则尝试重连  
 while(!socketChannel.isConnected()){  
 socketChannel.finishConnect();  
 }  
  
 //设置为非阻塞的  
 socketChannel.configureBlocking(false);  
  
 //创建数据缓冲区，将数据发送到服务端  
 ByteBuffer byteBuffer =  
 ByteBuffer.*wrap*("hello server".getBytes());  
 socketChannel.write(byteBuffer);  
  
 //读取服务器发送过来的数据  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
 socketChannel.read(buffer);  
  
 //打印读取的数据  
 System.*out*.println(  
 new String(buffer.array(),0,buffer.position()));  
  
 socketChannel.close();  
  
 }  
  
}

#### TCPSelectorChannelServer

package cn.tedu.selector;  
  
import org.junit.Test;  
  
import java.io.IOException;  
import java.net.InetSocketAddress;  
import java.nio.ByteBuffer;  
import java.nio.channels.\*;  
import java.util.Iterator;  
import java.util.Set;  
  
public class TCPSelectorChannelServer {  
  
 */\*\*  
 \* 创建具有选择器拦截的服务器  
 \* 将客户端的请求进行分类处理  
 \*/* @Test  
 public void test01() throws IOException {  
  
 //创建服务端通道绑定监听绑定8090端口  
 ServerSocketChannel serverSocketChannel =  
 ServerSocketChannel.*open*();  
  
 serverSocketChannel.bind(  
 new InetSocketAddress(8090));  
  
 //设置为非阻塞的  
 serverSocketChannel.configureBlocking(false);  
  
 //创建选择器  
 Selector selector = Selector.*open*();  
  
 //将选通道注册到选择器上，设置连接权限  
 serverSocketChannel.register(  
 selector, SelectionKey.*OP\_ACCEPT*);  
  
 //模拟服务器开启之后不关闭，一直接受客户单的连接  
 while (true) {  
 /\*  
 随着运行时间的延长，接收到的请求会越来越多  
 需要针对这些请求进行选择，将能触发事件的请求留下  
 将不能触发事件的请求过滤掉  
 \*/  
 selector.select();  
  
 /\*  
 选完之后，留下来的请求都是有用的请求  
 connect/read/write -> accept/write/read  
 因为这些请求中不一定存在所有的事件  
 所以需要获取请求的事件类型  
 \*/  
 Set<SelectionKey> selectionKeys =  
 selector.selectedKeys();  
  
 //讲选择器Key进行迭代  
 Iterator<SelectionKey> iterator =  
 selectionKeys.iterator();  
  
 while (iterator.hasNext()) {  
  
 //获取迭代器里每个选择器的key元素  
 SelectionKey selectionKey = iterator.next();  
  
 if (selectionKey.isAcceptable()) {  
  
 SocketChannel socketChannel =  
 serverSocketChannel.accept();  
 socketChannel.configureBlocking(false);  
  
 /\*  
 将根据需求确定，如果需要读操作，那么就给READ  
 如果需要写操作，那么就给WRITE  
 如果存在多个register，那么后边的会覆盖前边的  
 底层权限是二进制，0100为写 0001位读需要两个都具备  
 0100 + 0001 = 0101  
 \*/  
 socketChannel.register(  
 selector,  
 SelectionKey.*OP\_READ* + SelectionKey.*OP\_WRITE*);  
  
 }  
 if (selectionKey.isReadable()) {  
  
 /\*  
 客户端写，服务端读  
 判断为读的操作则使用缓冲区读取数据  
 读取完数据之后将选择器读权限剔除  
 \*/  
 SocketChannel socketChannelReadable =  
 (SocketChannel) selectionKey.channel();  
  
 ByteBuffer byteBuffer = ByteBuffer.*allocate*(1024);  
 socketChannelReadable.read(byteBuffer);  
 System.*out*.println(  
 new String(byteBuffer.array(), 0, byteBuffer.position()));  
  
 //将选择器内的读权限剔除  
 socketChannelReadable.register(  
 selector,  
 selectionKey.interestOps() - SelectionKey.*OP\_READ*);  
  
 }  
 if (selectionKey.isWritable()) {  
  
 /\*  
 判断为数据的写操作执行数据写出  
 \*/  
 SocketChannel socketChannel =  
 (SocketChannel) selectionKey.channel();  
  
 socketChannel.write(  
 ByteBuffer.*wrap*("收到数据..".getBytes()));  
  
 //剔除选择器权限  
 socketChannel.register(  
 selector,  
 selectionKey.interestOps() - SelectionKey.*OP\_WRITE*);  
  
 }  
  
 //处理完成之后，需要将一大类事件移除  
 iterator.remove();  
 }  
  
 }  
  
 }  
  
}

## 6.常见通信框架

### 6.1.通信框架

Mina出身于开源界的大牛Apache组织

Netty出身于商业开源大亨Jboss

Grizzly则出身于Sun公司

### 6.2.设计理念

#### 6.2.1.Mina

主导作者：Trustin Lee（韩国）

Mina(Multipurpose Infrastructure for Network Applications) 是 Apache 组织一个较新的项目，它为开发高性能和高可用性的网络应用程序提供了非常便利的框架

当前发行的 Mina 版本2.04支持基于 Java NIO 技术的 TCP/UDP 应用程序开发、串口通讯程序。

目前，正在使用Mina的应用包括：Apache Directory Project、AsyncWeb、AMQP（Advanced Message Queuing Protocol）、RED5 Server（Macromedia Flash Media RTMP）、ObjectRADIUS、 Openfire等等。

#### 6.2.1.Netty

主导作者：Trustin Lee（韩国）

Netty是一款异步的事件驱动的网络应用框架和工具，用于快速开发可维护的高性能、高扩展性协议服务器和客户端。

Netty是一个NIO客户端/服务器框架，支持快速、简单地开发网络应用，如协议服务器和客户端。

它极大简化了网络编程，如TCP和UDP套接字服务器。

#### 6.2.1.Grizzly

Sun公司提供

Grizzly是一种应用程序框架，专门解决编写成千上万用户访问服务器时候产生的各种问题。

Grizzly使用JAVA NIO作为基础，并隐藏其编程的复杂性，提供了容易使用的高性能的API。

Grizzly利用了非阻塞socket到协议处理层，使用了高性能的缓冲和缓冲管理使用高性能的线程池。

### 6.4.Netty简介

Netty是目前最流行的由JBOSS提供的一个Java开源框架NIO框架，Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。

相比JDK原生NIO，Netty提供了相对十分简单易用的API，非常适合网络编程，Netty是完全基于NIO实现的，作为一个异步NIO框架，Netty的所有IO操作都是异步非阻塞的，通过Future-Listener机制，用户可以方便的主动获取或者通过通知机制获得IO操作结果。

#### 6.4.1.Netty的优点

API使用简单，开发门槛低，功能强大，预置了多种编解码功能，支持多种主流协议定制能力强，可以通过ChannelHandler对通信框架进行灵活地扩展，性能高，通过与其他业界主流的NIO框架对比，Netty的综合性能最优，成熟、稳定，Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为NIO的BUG而烦恼。

社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会加入。经历了大规模的商业应用考验。

#### 6.4.2.与Mina相的比较：

都是Trustin Lee的作品，Netty更晚。Mina将内核和一些特性的联系过于紧密，使得用户在不需要这些特性的时候无法脱离，相比下性能会有所下降，Netty解决了这个设计问题。

Netty的文档更清晰，很多Mina的特性在Netty里都有，Netty更新周期更短，新版本的发布比较快，它们的架构差别不大，Mina靠apache生存，而Netty靠jboss，和jboss的结合度非常高，Netty有对google protocal buf的支持，有更完整的ioc容器支持(spring,guice,jbossmc和osgi)。

Netty比Mina使用起来更简单，Netty中可以自定义的处理upstream events或者downstream events，可以使用decoder和encoder来解码和编码发送内容。

Netty和Mina在处理UDP时有一些不同，Netty将UDP无连接的特性暴露出来；而Mina对UDP进行了高级层次的抽象，可以把UDP当成"面向连接"的协议，而要Netty做到这一点比较困难。

从任务调度粒度上看，Mina会将有IO任务的session写入队列中，当循环执行任务时，则会轮询所有的session，并依次把session中的所有任务取出来运行。这样粗粒度的调度是不公平调度，会导致某些请求的延迟很高。

# JUC

## 1.概述

JUC指的是JDK1.5中提供的一套并发包及其子包：java.util.concurrent，java.util.concurrent.lock，java.util.concurrent.atomic

JUC的主要内容：阻塞式队列、并发映射、锁、执行器服务、原子性操作。

## 2.BlockingQueue - 阻塞式队列

### 2.1.概述及重要方法

满足队列的特点：FIFO(First In First Out)

BlockingQueue在使用的时候需要容量，且容量是固定的，不可扩容。

阻塞：如果队列为空，则试图获取元素的线程会被阻塞；如果队列已满，则试图放入元素的线程会被阻塞。

BlockingQueue中不允许元素为null

适应于生产消费模型

### 2.2.重要方法

所有实现类的方法基本一样：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 抛出异常 | 返回特殊值 | 永久阻塞 | 定时阻塞 |
| 添加 | add –  IllegalStateException | offer - false | put | offer |
| 获取 | remove –  NoSuchElementException | poll - null | take | poll |

## 3.常用的实现类

### 3.1.ArrayBlockingQueue - 阻塞式顺序队列：

底层必然基于数组来存储数据

使用的时候需要指定容量

#### 3.1.1. ArrayBlockingQueue代码测试

##### ArrayBlockingQueueDemo01(存入)

package cn.tedu.blockingqueue;

import org.junit.Before;  
import org.junit.Test;  
  
import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.TimeUnit;  
  
public class ArrayBlockingQueueDemo01 {  
  
 private ArrayBlockingQueue<String> arrayBlockingQueue;  
  
 */\*\*  
 \* 测试阻塞队列的  
 \* ArrayBlockingQueue  
 \* 添加元素add  
 \*/* @Before  
 public void test01() {  
  
 //创建队列需要制定容量，制定后不能改变  
 arrayBlockingQueue = new ArrayBlockingQueue<>(5);  
  
 //添加元素  
 arrayBlockingQueue.add("a");  
 arrayBlockingQueue.add("a");  
 arrayBlockingQueue.add("a");  
 arrayBlockingQueue.add("a");  
 arrayBlockingQueue.add("a");  
 }  
  
 */\*\*  
 \* 单队列满了以后再次使用add()方法放入元素  
 \* 会抛出异常- IllegalStateException: Queue full  
 \*/* @Test  
 public void test02() {  
 arrayBlockingQueue.add("a");  
 }  
  
 */\*\*  
 \* 使用offer()方法放入元素  
 \* 会返回false  
 \*/* @Test  
 public void test03(){  
 boolean result = arrayBlockingQueue.offer("c");  
 System.*out*.println(result);  
 }  
  
 */\*\*  
 \* 使用put()方法放入元素  
 \* 会产生阻塞效果，还需要抛异常  
 \*/* @Test  
 public void test04() throws InterruptedException {  
 arrayBlockingQueue.put("e");  
 }  
  
 */\*\*  
 \* 使用offer()指定定时阻塞  
 \* 时间到了元素未加入则返回false  
 \*/* @Test  
 public void test05() throws InterruptedException {  
 boolean offer =  
 arrayBlockingQueue.offer("e", 5, TimeUnit.*SECONDS*);  
 System.*out*.println(offer);  
 }  
}

##### ArrayBlockingQueueDemo02(取出)

package cn.tedu.blockingqueue;

import org.junit.Before;  
import org.junit.Test;  
  
import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.TimeUnit;  
  
public class ArrayBlockingQueueDemo02 {  
  
 private ArrayBlockingQueue<String> arrayBlockingQueue;  
  
 */\*\*  
 \* 测试阻塞队列的  
 \* ArrayBlockingQueue  
 \* 取出元素  
 \*/* @Before  
 public void test01() {  
  
 //创建队列需要制定容量，制定后不能改变  
 arrayBlockingQueue = new ArrayBlockingQueue<>(5);  
 }  
  
 */\*\*  
 \* 单队列满了以后再次使用add()方法放入元素  
 \* 会抛出异常- NoSuchElementException  
 \*/* @Test  
 public void test02() {  
 arrayBlockingQueue.remove();  
 }  
  
 */\*\*  
 \* 使用poll()方法放入元素  
 \* 会返回null  
 \*/* @Test  
 public void test03(){  
 String result = arrayBlockingQueue.poll();  
 System.*out*.println(result);  
 }  
  
 */\*\*  
 \* 使用take()方法取出元素  
 \* 会产生阻塞效果，还需要抛异常  
 \*/* @Test  
 public void test04() throws InterruptedException {  
 arrayBlockingQueue.take();  
 }  
  
 */\*\*  
 \* 使用poll()指定定时阻塞  
 \* 时间到了未取到元素则返回null  
 \*/* @Test  
 public void test05() throws InterruptedException {  
 String poll =  
 arrayBlockingQueue.poll( 5, TimeUnit.*SECONDS*);  
 System.*out*.println(poll);  
 }  
}

### 3.2.LinkedBlockingQueue - 阻塞式链式队列

底层必然基于节点来存储数据，在使用的时候可以指定容量也可以不指定。如果指定容量，则容量不可变；如果不指定容量，则容量默认为Integer.MAX\_VALUE = 231-1不可变。因为实际开发中，一般不会在队列中存储21亿个元素，所以一般认为此时的容量是无限的。

### 3.3.PriorityBlockingQueue - 具有优先级的阻塞式队列

底层是基于数组来存储元素，使用的时候可以指定容量也可以不指定。如果不指定则默认初始容量是11。

PriorityBlockingQueue会对放入的元素来进行排序，要求元素对应的类实现Comparable接口，覆盖compareTo方法指定比较规则

PriorityBlockingQueue在迭代遍历的时候不保证排序。

#### 3.3.1.PriorityBlockingQueue测试案例

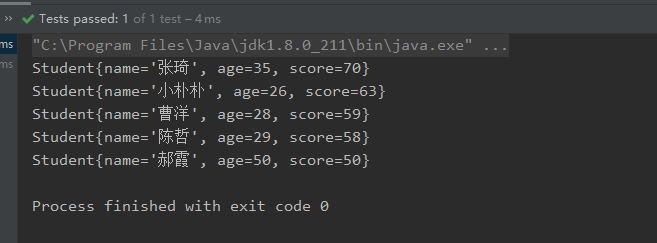
package cn.tedu.blockingqueue;

import org.junit.Test;  
  
import java.util.Comparator;  
import java.util.Iterator;  
import java.util.concurrent.PriorityBlockingQueue;  
  
public class PriorityBlockingQueueDemo01 {  
  
 */\*\*  
 \* 测试LinkedBlockingQueue  
 \* 查看特性，放入的元素是自动按照字典排序  
 \*/* @Test  
 public void test01() {  
  
 PriorityBlockingQueue<String> priorityBlockingQueue =

new PriorityBlockingQueue();  
 priorityBlockingQueue.put("t");  
 priorityBlockingQueue.put("a");  
 priorityBlockingQueue.put("y");  
 priorityBlockingQueue.put("e");  
 priorityBlockingQueue.put("k");  
 priorityBlockingQueue.put("r");  
 priorityBlockingQueue.put("p");  
  
 //[a, e, p, p, k, y, r, t]  
 System.*out*.println(priorityBlockingQueue);  
  
 }  
  
 */\*\*  
 \* 自定义排序的两种方式  
 \* 第一种方式：在domain封装数据的类里实现比较器  
 \* 之后按照自己的逻辑进行排序。  
 \* 第二种方式：创建一个Comparator比较器，重写  
 \* 内部类方法，指定范围为封装类，在重写的方法里  
 \* 定义排序逻辑  
 \*/* @Test  
 public void test02() throws InterruptedException {  
  
 //重写比较逻辑，将分数降序处理  
 Comparator<Student> comparator = new Comparator<Student>() {  
 @Override  
 public int compare(Student o1, Student o2) {  
 return o2.getScore() - o1.getScore();  
 }  
  
 };  
  
 // 在定义队列的时候，传入的比较器的优先级要高于Comparable  
 PriorityBlockingQueue<Student> queue =  
 new PriorityBlockingQueue<>(5,comparator);  
 queue.put(new Student("郝霞", 50, 50));  
 queue.put(new Student("曹洋", 28, 59));  
 queue.put(new Student("张琦", 35, 70));  
 queue.put(new Student("小朴朴", 26, 63));  
 queue.put(new Student("陈哲", 29, 58));  
  
 int size = queue.size();  
 for (int i = 0; i < size; i++) {  
  
 Student student = queue.poll();  
 System.*out*.println(student);  
 }  
  
 }  
  
 */\*\*  
 \* 使用迭代器遍历元素  
 \*/* @Test  
 public void test03() throws InterruptedException {  
  
 //重写比较逻辑，将分数降序处理  
 Comparator<Student> comparator = new Comparator<Student>() {  
 @Override  
 public int compare(Student o1, Student o2) {  
 return o2.getScore() - o1.getScore();  
 }  
 };  
  
 // 在定义队列的时候，传入的比较器的优先级要高于Comparable  
 PriorityBlockingQueue<Student> queue =  
 new PriorityBlockingQueue<>(5,comparator);  
 queue.put(new Student("郝霞", 50, 50));  
 queue.put(new Student("曹洋", 28, 59));  
 queue.put(new Student("张琦", 35, 70));  
 queue.put(new Student("小朴朴", 26, 63));  
 queue.put(new Student("陈哲", 29, 58));  
  
 Iterator<Student> studentIterator = queue.iterator();  
 while(studentIterator.hasNext()){  
  
 Student student = studentIterator.next();  
 System.*out*.println(student);  
 }  
 }  
  
}  
  
class Student implements Comparable<Student> {  
  
 private String name;  
 private int age;  
 private int score;  
  
 public Student(String name, int age, int score) {  
 this.name = name;  
 this.age = age;  
 this.score = score;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
  
 public int getScore() {  
 return score;  
 }  
  
 public void setScore(int score) {  
 this.score = score;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 ", score=" + score +  
 '}';  
 }  
  
 // 指定排序规则  
 // 按照年龄升序  
 // 升序：this - o  
 // 降序：o - this  
 @Override  
 public int compareTo(Student o) {  
 return this.age - o.age;  
 }  
  
}

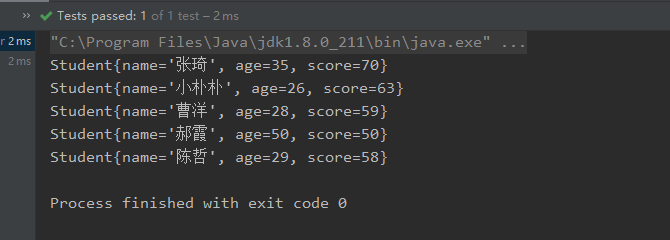
#### 3.3.2.普通循环遍历结果

代码是按照分数降序排序的，使用普通for循环遍历则正常排序。



#### 3.3.3.迭代器遍历结果

并没有按照比较器里写的按照分数降序进行排序，证明了使用迭代器则不能正常的进行元素的排序。



### 3.4.SynchronousQueue - 同步队列

在使用的时候不需要指定容量，容量默认为1且只能为1

### 3.5.BlockingDeque - 阻塞式双端队列

特点：允许从两端放入元素也允许从两端获取元素

遵循阻塞特点，在使用的时候需要指定容量