# 【NIO/AIO/BIO】

## 主要内容

1. IO的模型发展
2. NIO/AIO/BIO概念简介
3. BIO下实现通信
4. NIO下实现通信
5. AIO下实现通信

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| IO的模型发展 | 了解 |
| NIO/AIO/BIO概念简介 | 掌握 |
| BIO下实现通信 | 掌握 |
| NIO下实现通信 | 理解 |
| Channel（通道）、Buffer（缓冲区）、Selector（选择器） | 掌握 |
| AIO下实现通信 | 了解 |

## 一、IO模型发展

在Java的软件设计开发中，通信架构是不可避免的，我们在进行不同系统或者不同进程之间的数据交互，或者在高并发下的通信场景下都需要用到网络通信相关的技术，对于一些经验丰富的程序员来说，Java早期的网络通信架构存在一些缺陷，其中最令人恼火的是基于性能低下的同步阻塞式的I/O通信（BIO），随着互联网开发下通信性能的高要求，Java在2002年开始支持了非阻塞式的I/O通信技术(NIO)。大多数读者在学习网络通信相关技术的时候，都只是接触到零碎的通信技术点，没有完整的技术体系架构，以至于对于Java的通信场景总是没有清晰的解决方案。本次课程将通过大量清晰直接的案例从最基础的BIO式通信开始介绍到NIO , AIO，读者可以清晰的了解到阻塞、同步、异步的现象、概念和特征以及优缺点。

### 1.1 通信技术整体解决的问题

* 局域网内的通信要求。
* 多系统间的底层消息传递机制。
* 高并发下，大数据量的通信场景需要。
* 游戏行业。无论是手游服务端，还是大型的网络游戏，Java语言都得到越来越广泛的应用。

### 1.2 I/O 模型基本说明

I/O 模型：就是用什么样的通道或者说是通信模式和架构进行数据的传输和接收，很大程度上决定了程序通信的性能，Java 共支持 3 种网络编程的/IO 模型：BIO、NIO、AIO

实际通信需求下，要根据不同的业务场景和性能需求决定选择不同的I/O模型

BIO是一个连接一个线程。

NIO是一个请求一个线程。

AIO是一个有效请求一个线程。

先来个例子理解一下概念，以银行取款为例：

**同步**就是普通水壶烧开水，要没事儿自己过来来看开没开；

**异步**就是响水壶烧开水，水开了水壶响了通知你。

**阻塞**是烧开水的过程中，你不能干其他事情（即你被阻塞住了）；

**非阻塞**是烧开水的过程里可以干其他事情。比如出去和老相好聊聊天，去客厅看看电视；最

同步与异步说的是你获得水开了的方式不同。

阻塞与非阻塞说的是你得到结果之前能不能干其他事情。两组概念描述的是不同的内容。

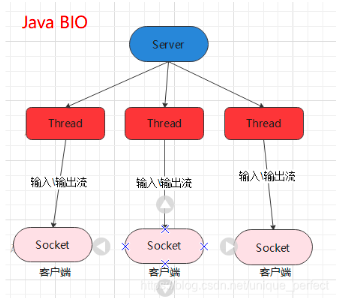
好的办法是 响水壶烧水（异步），烧开提示你之前可以去干别的事儿（非阻塞）

所以异步和非阻塞常常在一起

### 1.3 BIO、NIO、AIO模型

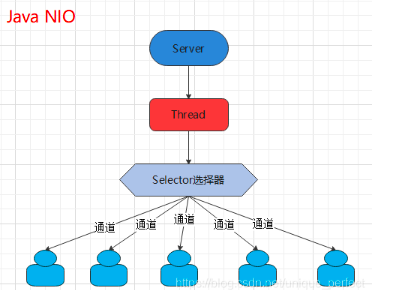
#### 1.3.1 Java BIO

同步并阻塞(传统阻塞型)，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销



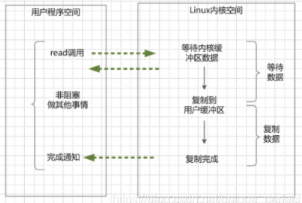
#### 1.3.2 NIO

Java NIO ： 同步非阻塞，服务器实现模式为一个线程处理多个请求(连接)，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有 I/O 请求就进行处理



#### 1.3.3**Java AIO**

Java AIO(NIO.2) ： 异步 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，一般适用于连接数较多且连接时间较长的应用



#### 1.3.4 BIO、NIO、AIO适用场景

1 BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序简单易理解。

2 NIO 方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，弹幕系统，服务器间通讯等。

编程比较复杂，JDK1.4 开始支持。

3 AIO 方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用 OS 参与并发操作，编程比较复杂，JDK7 开始支持。

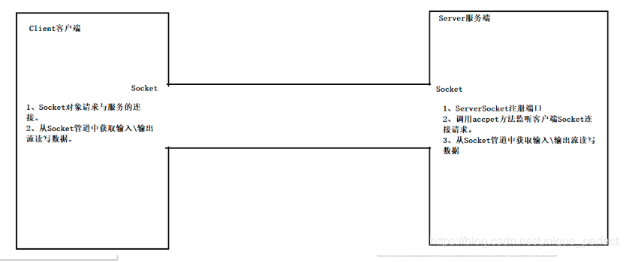
## 二、 BIO深入剖析

### 2.1 Java BIO 基本介绍

Java BIO 就是传统的 java io 编程，其相关的类和接口在 java.io

BIO(blocking I/O) ： 同步阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，可以通过线程池机制改善(实现多个客户连接服务器).

### 2.2**Java BIO 工作机制**



对 BIO 编程流程的梳理

1) 服务器端启动一个 ServerSocket，注册端口，调用accpet方法监听客户端的Socket连接。

2) 客户端启动 Socket对服务器进行通信，默认情况下服务器端需要对每个客户 建立一个线程与之通讯

### 2.3基于BIO下通信代码实现

代码实现

|  |
| --- |
| **public class** LoginThread **extends** Thread {  **private** Socket **socket**;  **public** LoginThread(Socket socket) {  **this**.**socket** = socket;  }  @Override  **public void** run() {  **try**(ObjectInputStream ois = **new** ObjectInputStream(**socket**.getInputStream());  DataOutputStream dos = **new** DataOutputStream(**socket**.getOutputStream());  ) {  //1.获取客户端的请求数据并输出  User user = (User) ois.readObject();  System.***out***.println(**"info="** + user);  //2.给出客户端响应  **if** (**"bjsxt"**.equals(user.getUserId()) && user.getPassword().length() >= 6) {  dos.writeUTF(**"登录成功，欢迎您！！"**);  } **else** {  dos.writeUTF(**"登录失败，请再次输入"**);  }  } **catch** (IOException e) {  e.printStackTrace();  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  } **catch** (Exception e) {  e.printStackTrace();  }  } }  **public class** LoginServer {  **public static void** main(String[] args)  **throws** IOException, ClassNotFoundException {  *//1.创建一个ServerSocket，负责监听客户端请求* ServerSocket serverSocket = **new** ServerSocket(8800);  **int** i = 0;  **while**(**true**){  *//2.使用ServerSocket进行监听* Socket socket = serverSocket.accept();*//请求未到，在此阻塞  //3.创建一个新的线程来处理请求* Thread thread = **new** LoginThread(socket);  thread.start();  *//4.输出请求的次数，客户端的ip地址* String ip = socket.getInetAddress().getHostAddress();  System.***out***.println(**"这是第"**+(++i)+**"次访问,对方的IP地址是："**+ip);  }  } } |

### 2.4总结

1 每个Socket接收到，都会创建一个线程，线程的竞争、切换上下文影响性能；

2 每个线程都会占用栈空间和CPU资源；

3 并不是每个socket都进行IO操作，无意义的线程处理；

4 客户端的并发访问增加时。服务端将呈现1:1的线程开销，访问量越大，系统将发生线程栈溢出，线程创建失败，最终导致进程宕机或者僵死，从而不能对外提供服务。

## 三、**NIO深入剖析**

### 3.1**Java NIO 基本介绍**

Java NIO（New IO）也有人称之为 java non-blocking IO是从Java 1.4版本开始引入的一个新的IO API，可以替代标准的Java IO API。NIO与原来的IO有同样的作用和目的，但是使用的方式完全不同，NIO支持面向缓冲区的、基于通的IO操作。NIO将以更加高效的方式进行文件的读写操作。NIO可以理解为非阻塞IO,传统的IO的read和write只能阻塞执行，线程在读写IO期间不能干其他事情，比如调用socket.read()时，如果服务器一直没有数据传输过来，线程就一直阻塞，而NIO中可以配置socket为非阻塞模式。

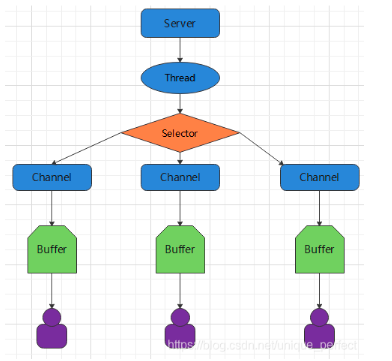
NIO 相关类都被放在 java.nio 包及子包下，并且对原 java.io 包中的很多类进行改写。

NIO 有三大核心部分：Channel( 通道) ，Buffer( 缓冲区), Selector( 选择器)

Java NIO 的非阻塞模式，使一个线程从某通道发送请求或者读取数据，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，而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 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。

通俗理解：NIO 是可以做到用一个线程来处理多个操作的。假设有 1000 个请求过来,根据实际情况，可以分配20 或者 80个线程来处理。不像之前的阻塞 IO 那样，非得分配 1000 个。

### **3.2 Java NIO 工作机制**



每个 channel 都会对应一个 Buffer一个线程对应Selector ， 一个Selector对应多个 channel(连接)程序切换到哪个 channel 是由事件决定的Selector 会根据不同的事件，在各个通道上切换Buffer 就是一个内存块 ， 底层是一个数组数据的读取写入是通过 Buffer完成的 , BIO 中要么是输入流，或者是输出流, 不能双向，但是 NIO 的 Buffer 是可以读也可以写。

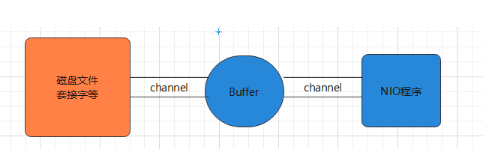
Java NIO系统的核心在于：通道(Channel)和缓冲区 (Buffer)。通道表示打开到 IO 设备(例如：文件、 套接字)的连接。若需要使用 NIO 系统，需要获取 用于连接 IO 设备的通道以及用于容纳数据的缓冲 区。然后操作缓冲区，对数据进行处理。简而言之，Channel 负责传输， Buffer 负责存取数据

### 3.3NIO 三大核心

#### 3.3.1NIO核心一：缓冲区(Buffer)

缓冲区（Buffer）

一个用于特定基本数据类型的容器。由 java.nio 包定义的，所有缓冲区 都是 Buffer 抽象类的子类.。Java NIO 中的 Buffer 主要用于与 NIO 通道进行 交互，数据是从通道读入缓冲区，从缓冲区写入通道中的



**Buffer 类及其子类**

Buffer就像一个数组，可以保存多个相同类型的数据。根据数据类型不同 ，有以下 Buffer 常用子类：

ByteBuffer 、CharBuffer 、ShortBuffer 、IntBuffer 、LongBuffer 、FloatBuffer

DoubleBuffer

上述 Buffer 类他们都采用相似的方法进行管理数据，只是各自 管理的数据类型不同而已。都是通过如下方法获取一个 Buffer 对象：

**缓冲区的基本属性**

Buffer 中的重要概念：

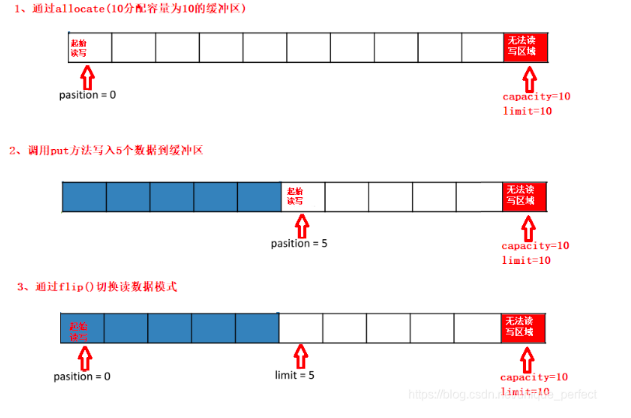
容量 (capacity) ：作为一个内存块，Buffer具有一定的固定大小，也称为"容量"，缓冲区容量不能为负，并且创建后不能更改。

限制 (limit)：表示缓冲区中可以操作数据的大小（limit 后数据不能进行读写）。缓冲区的限制不能为负，并且不能大于其容量。 写入模式，限制等于buffer的容量。读取模式下，limit等于写入的数据量。

位置 (position)：下一个要读取或写入的数据的索引。缓冲区的位置不能为 负，并且不能大于其限制

标记 (mark)与重置 (reset)：标记是一个索引，通过 Buffer 中的 mark() 方法 指定 Buffer 中一个特定的 position，之后可以通过调用 reset() 方法恢复到这 个 position.

标记、位置、限制、容量遵守以下不变式： 0 <= mark <= position <= limit <= capacity



**Buffer常见方法**

Buffer clear() 清空缓冲区并返回对缓冲区的引用

Buffer flip() 为 将缓冲区的界限设置为当前位置，并将当前位置充值为 0

int capacity() 返回 Buffer 的 capacity 大小

boolean hasRemaining() 判断缓冲区中是否还有元素

int limit() 返回 Buffer 的界限(limit) 的位置

Buffer limit(int n) 将设置缓冲区界限为 n, 并返回一个具有新 limit 的缓冲区对象

Buffer mark() 对缓冲区设置标记

int position() 返回缓冲区的当前位置 position

Buffer position(int n) 将设置缓冲区的当前位置为 n , 并返回修改后的 Buffer 对象

int remaining() 返回 position 和 limit 之间的元素个数

Buffer reset() 将位置 position 转到以前设置的 mark 所在的位置

Buffer rewind() 将位置设为为 0， 取消设置的 mark

**缓冲区的数据操作**

Buffer 所有子类提供了两个用于数据操作的方法：get()put() 方法

取获取 Buffer中的数据

get() ：读取单个字节

get(byte[] dst)：批量读取多个字节到 dst 中

get(int index)：读取指定索引位置的字节(不会移动 position)

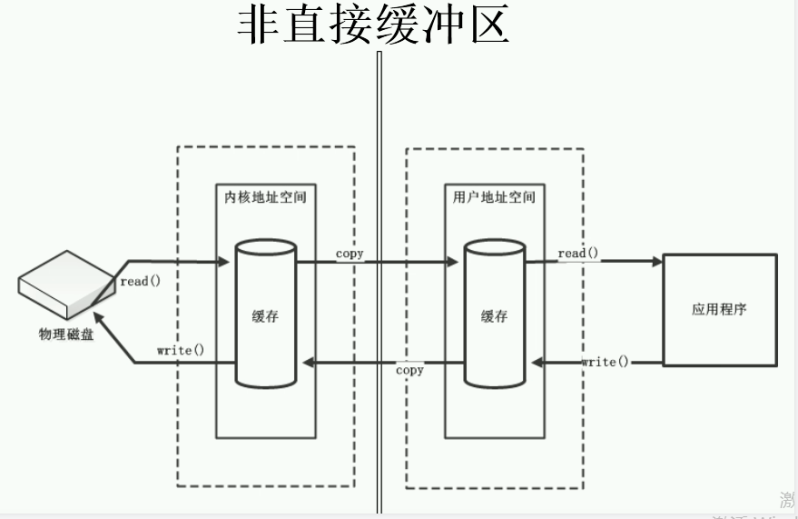
放到 入数据到 Buffer 中 中

put(byte b)：将给定单个字节写入缓冲区的当前位置

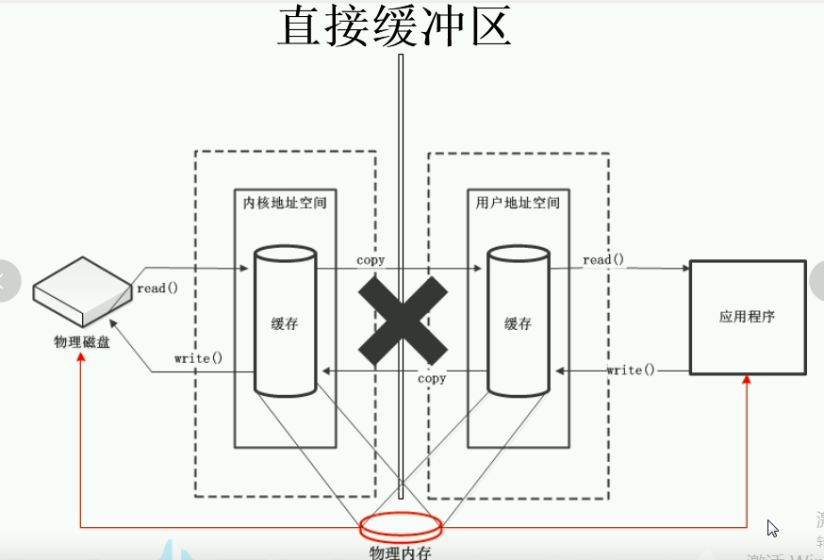
put(byte[] src)：将 src 中的字节写入缓冲区的当前位置

put(int index, byte b)：将指定字节写入缓冲区的索引位置(不会移动 position)

**直接缓冲区和非直接缓冲区**



解释：竖左边的是操作系统（OS），右边是Java虚拟机(JVM),应用程序无论是读操作还是写操作都必须在OS和JVM之间进行复制。



解释：在NIO中，直接开辟物理内存映射文件，应用程序直接操作物理内存映射文件，这样就少了中间的copy过程，可以极大得提高读写效率。但这种方式也存在一个问题，消耗的内存会增大，内存的释放只能通过Java的垃圾回收机制来释放，并不是一不使用就回收。

创建的缓冲区，在JVM内存外开辟内存，在每次调用基础操作系统的一个本机IO之前或者之后，虚拟机都会避免将缓冲区的内容复制到中间缓冲区（或者从中间缓冲区复制内容），缓冲区的内容驻留在物理内存内，会少一次复制过程，如果需要循环使用缓冲区，用直接缓冲区可以很大地提高性能。虽然直接缓冲区使JVM可以进行高效的I/O操作，但它使用的内存是操作系统分配的，绕过了JVM堆栈，建立和销毁比堆栈上的缓冲区要更大的开销。

**代码实现**

|  |
| --- |
| */\*\*  \* 一、缓冲区（Buffer）：在 Java NIO 中负责数据的存取。缓冲区就是数组。用于存储不同数据类型的数据  \*   \* 根据数据类型不同（boolean 除外），提供了相应类型的缓冲区：  \* ByteBuffer !!  \* CharBuffer  \* ShortBuffer  \* IntBuffer  \* LongBuffer  \* FloatBuffer  \* DoubleBuffer  \*   \* 上述缓冲区的管理方式几乎一致，通过 allocate() 获取缓冲区  \*   \* 二、缓冲区存取数据的两个核心方法：  \* put() : 存入数据到缓冲区中  \* get() : 获取缓冲区中的数据  \*   \* 三、缓冲区中的四个核心属性：  \* capacity : 容量，表示缓冲区中最大存储数据的容量。一旦声明不能改变。  \* limit : 界限，表示缓冲区中可以操作数据的大小。（limit 后数据不能进行读写）  \* position : 位置，表示缓冲区中正在操作数据的位置  \* mark : 标记，表示记录当前 position 的位置。可以通过 reset() 恢复到 mark 的位置  \*   \* 0 <= mark <= position <= limit <= capacity  \*   \* 四、直接缓冲区与非直接缓冲区：  \* 非直接缓冲区：通过 allocate() 方法分配缓冲区，将缓冲区建立在 JVM 的内存中  \* 直接缓冲区：通过 allocateDirect() 方法分配直接缓冲区，将缓冲区建立在物理内存中。可以提高效率  \*/* **public class** TestBuffer {    @Test  **public void** test3(){  *//分配直接缓冲区* ByteBuffer buf = ByteBuffer.*allocateDirect*(1024);    System.***out***.println(buf.isDirect());  }    @Test  **public void** test2(){  String str = **"abcde"**;    ByteBuffer buf = ByteBuffer.*allocate*(1024);    buf.put(str.getBytes());    buf.flip();    **byte**[] dst = **new byte**[buf.limit()];  buf.get(dst, 0, 2);  System.***out***.println(**new** String(dst, 0, 2));  System.***out***.println(buf.position());    *//mark() : 标记* buf.mark();    buf.get(dst, 2, 2);  System.***out***.println(**new** String(dst, 2, 2));  System.***out***.println(buf.position());    *//reset() : 恢复到 mark 的位置* buf.reset();  System.***out***.println(buf.position());    *//判断缓冲区中是否还有剩余数据* **if**(buf.hasRemaining()){    *//获取缓冲区中可以操作的数量* System.***out***.println(buf.remaining());  }  }    @Test  **public void** test1(){  String str = **"abcde"**;    *//1. 分配一个指定大小的缓冲区* ByteBuffer buf = ByteBuffer.*allocate*(1024);    System.***out***.println(**"-----------------allocate()----------------"**);  System.***out***.println(buf.position());  System.***out***.println(buf.limit());  System.***out***.println(buf.capacity());    *//2. 利用 put() 存入数据到缓冲区中* buf.put(str.getBytes());    System.***out***.println(**"-----------------put()----------------"**);  System.***out***.println(buf.position());  System.***out***.println(buf.limit());  System.***out***.println(buf.capacity());    *//3. 切换读取数据模式* buf.flip();    System.***out***.println(**"-----------------flip()----------------"**);  System.***out***.println(buf.position());  System.***out***.println(buf.limit());  System.***out***.println(buf.capacity());    *//4. 利用 get() 读取缓冲区中的数据* **byte**[] dst = **new byte**[buf.limit()];  buf.get(dst);  System.***out***.println(**new** String(dst, 0, dst.**length**));    System.***out***.println(**"-----------------get()----------------"**);  System.***out***.println(buf.position());  System.***out***.println(buf.limit());  System.***out***.println(buf.capacity());    *//5. rewind() : 可重复读* buf.rewind();    System.***out***.println(**"-----------------rewind()----------------"**);  System.***out***.println(buf.position());  System.***out***.println(buf.limit());  System.***out***.println(buf.capacity());    *//6. clear() : 清空缓冲区. 但是缓冲区中的数据依然存在，但是处于“被遗忘”状态* buf.clear();    System.***out***.println(**"-----------------clear()----------------"**);  System.***out***.println(buf.position());  System.***out***.println(buf.limit());  System.***out***.println(buf.capacity());    System.***out***.println((**char**)buf.get());    }  } |

#### 3.3.2 NIO核心二：通道(Channel)

通道Channe概述

通道（Channel）：由 java.nio.channels 包定义 的。Channel 表示 IO 源与目标打开的连接。 Channel 类似于传统的“流”。只不过 Channel 本身不能直接访问数据，Channel 只能与 Buffer 进行交互。

[1] NIO 的通道类似于流，但有些区别如下：

通道可以同时进行读写，而流只能读或者只能写

通道可以实现异步读写数据

通道可以从缓冲读数据，也可以写数据到缓冲:

[2] BIO 中的 stream 是单向的，例如 FileInputStream 对象只能进行读取数据的操作，而 NIO 中的通道(Channel)是双向的，可以读操作，也可以写操作。

[3] Channel 在 NIO 中是一个接口

**常用的Channel实现类**

FileChannel：用于读取、写入、映射和操作文件的通道。

DatagramChannel：通过 UDP 读写网络中的数据通道。

SocketChannel：通过 TCP 读写网络中的数据。

ServerSocketChannel：可以监听新进来的 TCP 连接，对每一个新进来的连接都会创建一个 SocketChannel。 【ServerSocketChanne 类似 ServerSocket , SocketChannel 类似 Socket】

**FileChannel的常用方法**

int read(ByteBuffer dst) 从 从 Channel 到 中读取数据到 ByteBuffer

long read(ByteBuffer[] dsts) 将 Channel 到 中的数据“分散”到 ByteBuffer[]

int write(ByteBuffer src) 将 ByteBuffer 到 中的数据写入到 Channel

long write(ByteBuffer[] srcs) 将 ByteBuffer[] 到 中的数据“聚集”到 Channel

long position() 返回此通道的文件位置

FileChannel position(long p) 设置此通道的文件位置

long size() 返回此通道的文件的当前大小

FileChannel truncate(long s) 将此通道的文件截取为给定大小

void force(boolean metaData) 强制将所有对此通道的文件更新写入到存储设备中

代码实现

|  |
| --- |
| *\*\*  \* 一、通道（Channel）：用于源节点与目标节点的连接。在 Java NIO 中负责缓冲区中数据的传输。Channel 本身不存储数据，因此需要配合缓冲区进行传输。  \*   \* 二、通道的主要实现类  \* java.nio.channels.Channel 接口：  \* |--FileChannel  \* |--SocketChannel  \* |--ServerSocketChannel  \* |--DatagramChannel  \*   \* 三、获取通道  \* 1. Java 针对支持通道的类提供了 getChannel() 方法  \* 本地 IO：  \* FileInputStream/FileOutputStream  \*   \*   \* 网络IO：  \* Socket  \* ServerSocket  \* DatagramSocket  \*   \* 2. 在 JDK 1.7 中的 NIO.2 针对各个通道提供了静态方法 open()  \* 3. 在 JDK 1.7 中的 NIO.2 的 Files 工具类的 newByteChannel()  \*   \* 四、通道之间的数据传输  \* transferFrom()  \* transferTo()  \*   \* 五、分散(Scatter)与聚集(Gather)  \* 分散读取（Scattering Reads）：将通道中的数据分散到多个缓冲区中  \* 聚集写入（Gathering Writes）：将多个缓冲区中的数据聚集到通道中  \*   \* 六、字符集：Charset  \* 编码：字符串 -> 字节数组  \* 解码：字节数组 -> 字符串  \*   \*/* **public class** TestChannel {    *//字符集* @Test  **public void** test6() **throws** IOException{  Charset cs1 = Charset.*forName*(**"GBK"**);  *//获取编码器* CharsetEncoder ce = cs1.newEncoder();*//编码器  //获取解码器* CharsetDecoder cd = cs1.newDecoder();*//解码器* CharBuffer cBuf = CharBuffer.*allocate*(1024);  cBuf.put(**"今天天气"**);   cBuf.flip();  *//编码* ByteBuffer bBuf = ce.encode(cBuf);    **for** (**int** i = 0; i < 10; i++) {  System.***out***.println(bBuf.get());  }    *//解码* bBuf.flip();*//否则乱码* CharBuffer cBuf2 = cd.decode(bBuf);  System.***out***.println(cBuf2.toString());    System.***out***.println(**"------------------------------------"**);      }   *//查看有多少字符集* @Test  **public void** test5(){  Map<String, Charset> map = Charset.*availableCharsets*();    Set<Entry<String, Charset>> set = map.entrySet();    **for** (Entry<String, Charset> entry : set) {  System.***out***.println(entry.getKey() + **"="** + entry.getValue());  }  }    *//分散和聚集* @Test  **public void** test4() **throws** Exception{  RandomAccessFile raf1 = **new** RandomAccessFile(**"1.txt"**, **"rw"**);    *//1. 获取通道* FileChannel channel1 = raf1.getChannel();    *//2. 分配指定大小的缓冲区* ByteBuffer buf1 = ByteBuffer.*allocate*(100);  ByteBuffer buf2 = ByteBuffer.*allocate*(1024);    *//3. 分散读取* ByteBuffer[] bufs = {buf1, buf2};  channel1.read(bufs);    **for** (ByteBuffer byteBuffer : bufs) {  byteBuffer.flip();  }    System.***out***.println(**new** String(bufs[0].array(), 0, bufs[0].limit()));  System.***out***.println(**"-----------------"**);  System.***out***.println(**new** String(bufs[1].array(), 0, bufs[1].limit()));    *//4. 聚集写入* RandomAccessFile raf2 = **new** RandomAccessFile(**"2.txt"**, **"rw"**);  FileChannel channel2 = raf2.getChannel();    channel2.write(bufs);  }    *//通道之间的数据传输(直接缓冲区) ----简单易行 推荐！* @Test  **public void** test3() **throws** Exception{  FileChannel inChannel = FileChannel.*open*(Paths.*get*(**"1.jpg"**), StandardOpenOption.***READ***);  FileChannel outChannel = FileChannel.*open*(Paths.*get*(**"4.jpg"**), StandardOpenOption.***WRITE***, StandardOpenOption.***READ***, StandardOpenOption.***CREATE***);   *// inChannel.transferTo(0, inChannel.size(), outChannel);* outChannel.transferFrom(inChannel, 0, inChannel.size());    inChannel.close();  outChannel.close();  }      *//利用通道完成文件的复制（非直接缓冲区）* @Test  **public void** test1(){*//10874-10953 // long start = System.currentTimeMillis();* FileInputStream fis = **null**;  FileOutputStream fos = **null**;  *//①获取通道* FileChannel inChannel = **null**;  FileChannel outChannel = **null**;  **try** {  fis = **new** FileInputStream(**"1.jpg"**);  fos = **new** FileOutputStream(**"2.jpg"**);    inChannel = fis.getChannel();  outChannel = fos.getChannel();    *//②分配指定大小的缓冲区* ByteBuffer buf = ByteBuffer.*allocate*(1024);    *//③将通道中的数据存入缓冲区中* **while**(inChannel.read(buf) != -1){  buf.flip(); *//切换读取数据的模式  //④将缓冲区中的数据写入通道中* outChannel.write(buf);  buf.clear(); *//清空缓冲区* }  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  **if**(outChannel != **null**){  **try** {  outChannel.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }    **if**(inChannel != **null**){  **try** {  inChannel.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }    **if**(fos != **null**){  **try** {  fos.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }    **if**(fis != **null**){  **try** {  fis.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }   *// long end = System.currentTimeMillis(); // System.out.println("耗费时间为：" + (end - start));* }  } |

#### **3.3.3 NIO核心三：选择器(Selector)**

**选择器(Selector)概述**

选择器（Selector） 是 SelectableChannle 对象的多路复用器，Selector 可以同时监控多个 SelectableChannel 的 IO 状况，也就是说，利用 Selector可使一个单独的线程管理多个 Channel。Selector 是非阻塞 IO 的核心

Java 的 NIO，用非阻塞的 IO 方式。可以用一个线程，处理多个的客户端连接，就会使用到 Selector(选择器)

Selector 能够检测多个注册的通道上是否有事件发生(注意:多个 Channel 以事件的方式可以注册到同一个

Selector)，如果有事件发生，便获取事件然后针对每个事件进行相应的处理。这样就可以只用一个单线程去管理多个通道，也就是管理多个连接和请求。

只有在 连接/通道 真正有读写事件发生时，才会进行读写，就大大地减少了系统开销，并且不必为每个连接都创建一个线程，不用去维护多个线程

避免了多线程之间的上下文切换导致的开销

**选择器（Selector）的应用**

//1. 获取通道

ServerSocketChannel ssChannel = ServerSocketChannel.open();

//2. 切换非阻塞模式

ssChannel.configureBlocking(false);

//3. 绑定连接

ssChannel.bind(new InetSocketAddress(9898));

//4. 获取选择器

Selector selector = Selector.open();

//5. 将通道注册到选择器上, 并且指定“监听接收事件”

ssChannel.register(selector, SelectionKey.OP\_ACCEPT);

当调用 register(Selector sel, int ops) 将通道注册选择器时，选择器对通道的监听事件，需要通过第二个参数 ops 指定。可以监听的事件类型（用 可使用 SelectionKey 的四个常量 表示）：

读 : SelectionKey.OP\_READ （1）

写 : SelectionKey.OP\_WRITE （4）

连接 : SelectionKey.OP\_CONNECT （8）

接收 : SelectionKey.OP\_ACCEPT （16）

若注册时不止监听一个事件，则可以使用“位或”操作符连接。

### 3.4基于NIO下通信代码实现

阻塞式代码实现

|  |
| --- |
| */\*\*  \* 一、使用 NIO 完成网络通信的三个核心：  \*   \* 1. 通道（Channel）：负责连接  \*   \* java.nio.channels.Channel 接口：  \* |--SelectableChannel  \* |--SocketChannel  \* |--ServerSocketChannel  \* |--DatagramChannel  \*   \* |--Pipe.SinkChannel  \* |--Pipe.SourceChannel  \*   \* 2. 缓冲区（Buffer）：负责数据的存取  \*   \* 3. 选择器（Selector）：是 SelectableChannel 的多路复用器。用于监控 SelectableChannel 的 IO 状况  \*   \*/* **public class** TestBlockingNIO {   *//客户端* @Test  **public void** client() **throws** IOException{  *//1. 获取通道* SocketChannel sChannel = SocketChannel.*open*(**new** InetSocketAddress(**"127.0.0.1"**, 9898));    FileChannel inChannel = FileChannel.*open*(Paths.*get*(**"1.jpg"**), StandardOpenOption.***READ***);    *//2. 分配指定大小的缓冲区* ByteBuffer buf = ByteBuffer.*allocate*(1024);    *//3. 读取本地文件，并发送到服务端* **while**(inChannel.read(buf) != -1){  buf.flip();  sChannel.write(buf);  buf.clear();  }    *//4. 关闭通道* inChannel.close();  sChannel.close();  }    *//服务端* @Test  **public void** server() **throws** IOException{  *//1. 获取通道* ServerSocketChannel ssChannel = ServerSocketChannel.*open*();    FileChannel outChannel = FileChannel.*open*(Paths.*get*(**"5.jpg"**), StandardOpenOption.***WRITE***, StandardOpenOption.***CREATE***);    *//2. 绑定连接* ssChannel.bind(**new** InetSocketAddress(9898));    *//3. 获取客户端连接的通道* SocketChannel sChannel = ssChannel.accept();    *//4. 分配指定大小的缓冲区* ByteBuffer buf = ByteBuffer.*allocate*(1024);    *//5. 接收客户端的数据，并保存到本地* **while**(sChannel.read(buf) != -1){  buf.flip();  outChannel.write(buf);  buf.clear();  }    *//6. 关闭通道* sChannel.close();  outChannel.close();  ssChannel.close();    }   } |

非阻塞实现

|  |
| --- |
| **public class** TestNonBlockingNIO {    *//客户端* @Test  **public void** client() **throws** IOException{  *//1. 获取通道* SocketChannel sChannel = SocketChannel.*open*(**new** InetSocketAddress(**"127.0.0.1"**, 9898));    *//2. 切换非阻塞模式* sChannel.configureBlocking(**false**);    *//3. 分配指定大小的缓冲区* ByteBuffer buf = ByteBuffer.*allocate*(1024);    *//4. 发送数据给服务端* Scanner scan = **new** Scanner(System.***in***);    buf.put((**new** Date().toString() + **"\n"** ).getBytes());  buf.flip();  sChannel.write(buf);  buf.clear();    *//5. 关闭通道* sChannel.close();  }   *//服务端* @Test  **public void** server() **throws** IOException{  *//1. 获取通道* ServerSocketChannel ssChannel = ServerSocketChannel.*open*();    *//2. 切换非阻塞模式* ssChannel.configureBlocking(**false**);    *//3. 绑定连接* ssChannel.bind(**new** InetSocketAddress(9898));    *//4. 获取选择器* Selector selector = Selector.*open*();    *//5. 将通道注册到选择器上, 并且指定“监听接收事件”* ssChannel.register(selector, SelectionKey.***OP\_ACCEPT***);    *//6. 轮询式的获取选择器上已经“准备就绪”的事件* **if** (selector.select() > 0){*//说明至少有一个准备就绪了   //7. 获取当前选择器中所有注册的“选择键(已就绪的监听事件)”* Iterator<SelectionKey> it = selector.selectedKeys().iterator();   **while**(it.hasNext()){  *//8. 获取准备“就绪”的是事件* SelectionKey sk = it.next();   System.***out***.println(sk.isAcceptable()+**"--"**+sk.isReadable());    *//9. 判断具体是什么事件准备就绪* **if**(sk.isAcceptable()){  *//10. 若“接收就绪”，获取客户端连接* SocketChannel sChannel = ssChannel.accept();    *//11. 切换非阻塞模式* sChannel.configureBlocking(**false**);    *//12. 将该通道注册到选择器上  //sChannel.register(selector, SelectionKey.OP\_READ);* }**else if**(sk.isReadable()){  *//13. 获取当前选择器上“读就绪”状态的通道* SocketChannel sChannel = (SocketChannel) sk.channel();    */\*\*14. 读取数据\*/* ByteBuffer buf = ByteBuffer.*allocate*(20);    **int** len = 0;  **while**((len = sChannel.read(buf)) > 0 ){  buf.flip();  System.***out***.println(**new** String(buf.array(), 0, len));  buf.clear();  }  }   *//15. 取消选择键 SelectionKey* it.remove();     }  } } |

## 四、**AIO深入剖析**

### 4.1**Java AIO 基本介绍**

Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。

### 4.2**Java AIO 工作机制**

AIO异步非阻塞，基于NIO的，可以称之为NIO2.0

|  |  |  |
| --- | --- | --- |
| BIO | NIO | AIO |
| Socket | SocketChannel | AsynchronousSocketChannel |
| ServerSocket | ServerSocketChannel | AsynchronousServerSocketChannel |

与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可, 这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区,对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。在JDK1.7中，这部分内容被称作NIO.2，主要在Java.nio.channels包下增加了下面四个异步通道：

AsynchronousSocketChannel

AsynchronousServerSocketChannel

AsynchronousFileChannel

AsynchronousDatagramChannel

### 4.3总结

BIO、NIO、AIO：

Java BIO ： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

Java NIO ： 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。

BIO、NIO、AIO适用场景分析:

BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。

NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。

AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。Netty!