BIO、NIO、AIO、Netty

1. 什么是IO

- Java中I/O是以流为基础进行数据的输入输出的，所有数据被串行化(所谓串行化就是数据要按顺序进行输入输出)写入输出流。简单来说就是java通过io流方式和外部设备进行交互。
- 在Java类库中，IO部分的内容是很庞大的，因为它涉及的领域很广泛：标准输入输出，文件的操作，**网络上的数据传输流**，字符串流，对象流等等等。

- 比如程序从服务器上下载图片，就是通过流的方式从网络上以流的方式到程序中，在到硬盘中

2. 在了解不同的IO之前先了解：同步与异步，阻塞与非阻塞的区别

- 同步，一个任务的完成之前不能做其他操作，必须等待（等于在打电话）
- 异步，一个任务的完成之前，可以进行其他操作（等于在聊QQ）
- 阻塞，是相对于CPU来说的， 挂起当前线程，不能做其他操作只能等待
- 非阻塞,，无须挂起当前线程，可以去执行其他操作

3. 什么是BIO

- BIO：同步并阻塞，服务器实现一个连接一个线程，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，没处理完之前此线程不能做其他操作（如果是单线程的情况下，我传输的文件很大呢？），当然可以通过线程池机制改善。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。

4. 什么是NIO

- NIO:同步非阻塞，服务器实现一个连接一个线程，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。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之后开始支持。

5. 什么是AIO

- AIO：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请求都是由操作系统先完成了再通知服务器应用去启动线程进行处理，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，充分调用操作系统参与并发操作，编程比较复杂，JDK1.7之后开始支持。.
- AIO属于NIO包中的类实现，其实IO主要分为BIO和NIO，AIO只是附加品，解决IO不能异步的实现
- 在以前很少有Linux系统支持AIO，Windows的IOCP就是该AIO模型。但是现在的服务器一般都是支持AIO操作

6. 什么Netty

- Netty是由JBOSS提供的一个Java开源框架。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。
- Netty 是一个基于NIO的客户、服务器端编程框架，使用Netty 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户，服务端应用。Netty相当简化和流线化了网络应用的编程开发过程，例如，TCP和UDP的socket服务开发。

Netty是由NIO演进而来，使用过NIO编程的用户就知道NIO编程非常繁重，Netty是能够能跟好的使用NIO

7. BIO和NIO、AIO的区别

- BIO是阻塞的，NIO是非阻塞的.
- BIO是面向流的，只能单向读写，NIO是面向缓冲的, 可以双向读写
- 使用BIO做Socket连接时，由于单向读写，当没有数据时，会挂起当前线程，阻塞等待，为防止影响其它连接,，需要为每个连接新建线程处理.，然而系统资源是有限的,，不能过多的新建线程，线程过多带来线程上下文的切换，从来带来更大的性能损耗，因此需要使用NIO进行BIO多路复用，使用一个线程来监听所有Socket连接，使用本线程或者其他线程处理连接
- AIO是非阻塞 以异步方式发起 I/O 操作。当 I/O 操作进行时可以去做其他操作，由操作系统内核空间提醒IO操作已完成（不懂的可以往下看）

8. IO流的分类

按照读写的单位大小来分：

- `字符流`：以字符为单位，每次次读入或读出是16位数据。其只能读取字符类型数据。（Java代码接收数据为一般为`char数组，也可以是别的`）
- 字节流：以字节为单位，每次次读入或读出是8位数据。可以读任何类型数据，图片、文件、音乐视频等。（Java代码接收数据只能为`byte数组`）

按照实际IO操作来分：

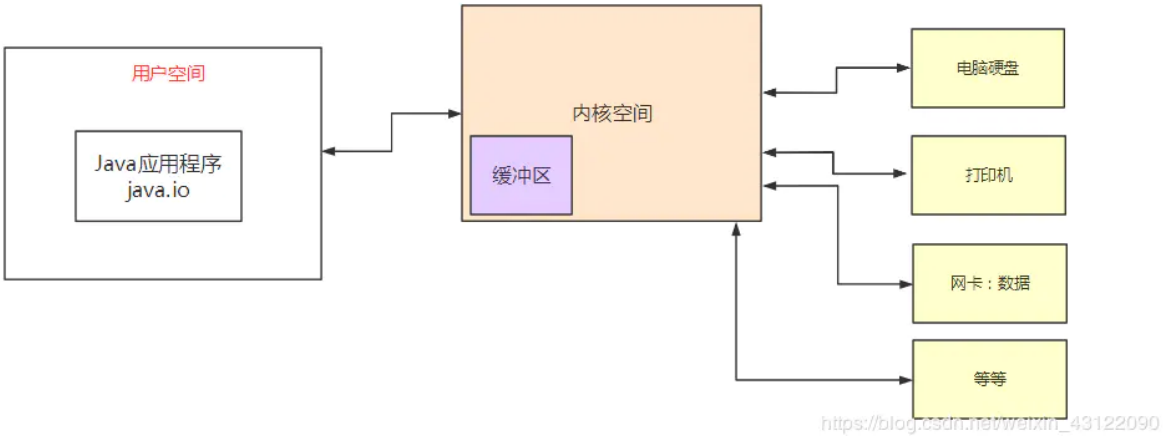
- 输出流：从内存读出到文件。只能进行写操作。
- 输入流：从文件读入到内存。只能进行读操作。
- **注意**：输出流可以帮助我们创建文件，而输入流不会。

按照读写时是否直接与硬盘，内存等节点连接分：

- 节点流：直接与数据源相连，读入或读出。
- 处理流：也叫包装流，是对一个对于已存在的流的连接进行封装，通过所封装的流的功能调用实现数据读写。如添加个Buffering缓冲区。（意思就是有个缓存区，等于软件和mysql中的redis）
- **注意**：为什么要有处理流？主要作用是在读入或写出时，对数据进行缓存，以减少I/O的次数，以便下次更好更快的读写文件，才有了处理流。

9. 什么是内核空间

- 我们的应用程序是不能直接访问硬盘的，我们程序没有权限直接访问，但是操作系统（Windows、Linux.....）会给我们一部分权限较高的内存空间，他叫内核空间，和我们的实际硬盘空间是有区别的



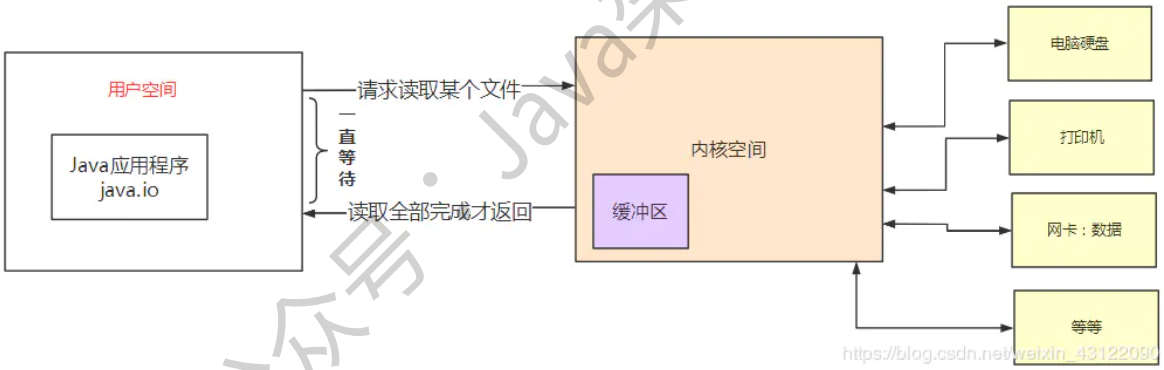
10. 五种IO模型

- **注意：我这里的用户空间就是应用程序空间**

1.阻塞BIO（blocking I/O）

- A拿着一支鱼竿在河边钓鱼，并且一直在鱼竿前等，在等的时候不做其他的事情，十分专心。只有鱼上钩的时，才结束掉等的动作，把鱼钓上来。
- 在内核将数据准备好之前，系统调用会一直等待所有的套接字，默认的是阻塞方式。

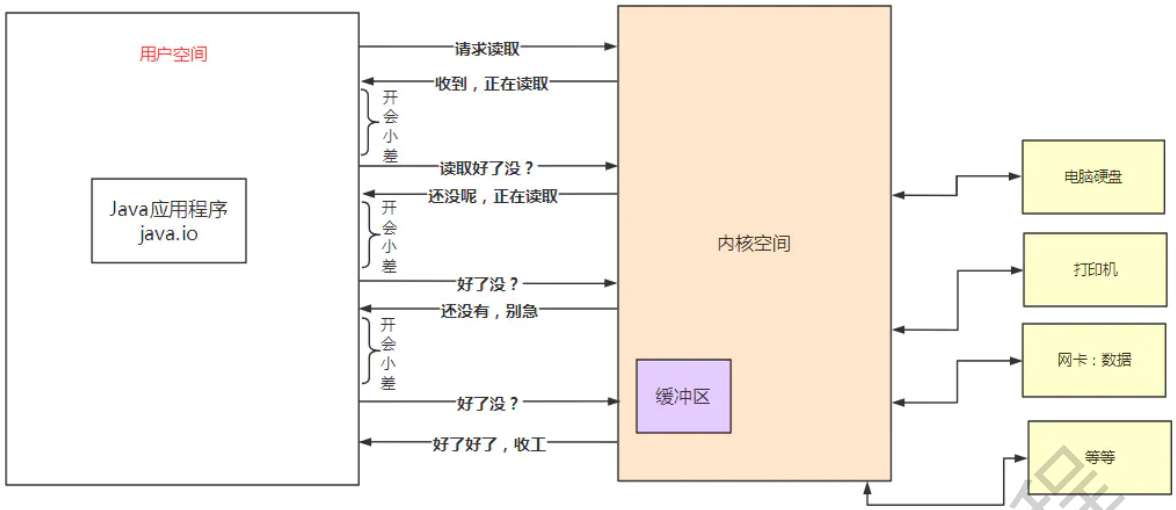
以线程为单位：期间正个过程是不能去操作其他任何事情



2.非阻塞NIO（noblocking I/O）

- B也在河边钓鱼，但是B不想将自己的所有时间都花费在钓鱼上，在等鱼上钩这个时间段中，B也在做其他的事情（一会看看书，一会读读报纸，一会又去看其他人的钓鱼等），但B在做这些事情的时候，每隔一个固定的时间检查鱼是否上钩。一旦检查到有鱼上钩，就停下手中的事情，把鱼钓上来。 **B在检查鱼竿是否有鱼，是一个轮询的过程。**

以线程为单位：期间整个过程中是可以去做别的操作

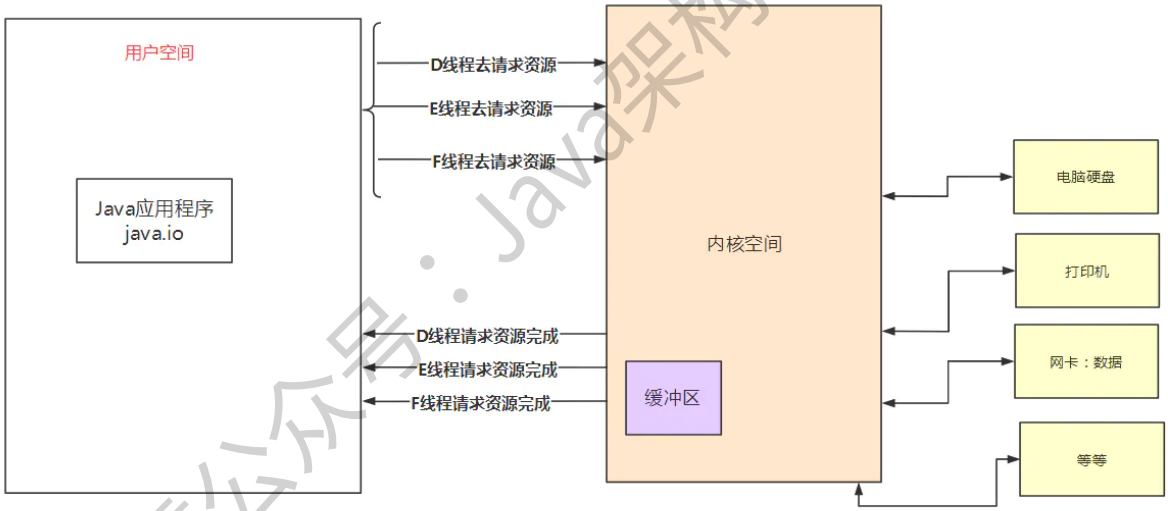


每次检查否处理好就是在轮训的过程，这是BIO的操作

3.异步AIO（asynchronous I/O）

- C也想钓鱼，但C有事情，于是他雇来了D、E、F，让他们帮他等待鱼上钩，一旦有鱼上钩，就打电话给C，C就会将鱼钓上去。

以线程为单位：期间整个过程中是可以去做别的操作



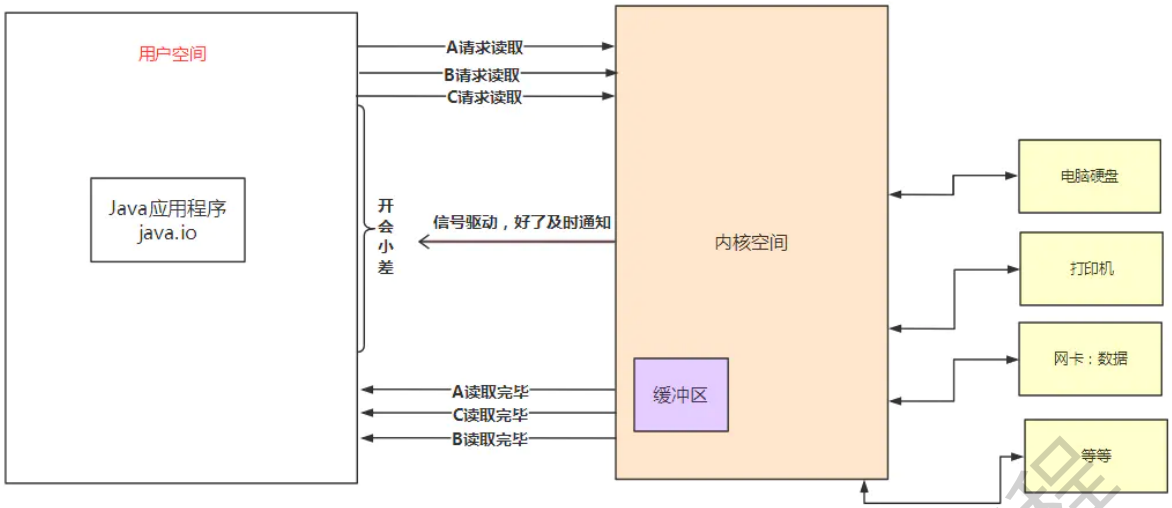
当内核中有数据报就绪时，由内核将数据报拷贝到应用程序中，

> 当应用程序请求数据时，内核一方面去取数据报内容返回，另一方面将程序控制权还给应用进程，应用进程继续处理其他事情，是一种非阻塞的状态。

4.信号驱动IO（signal blocking I/O）

- G也在河边钓鱼，但与A、B、C不同的是，G比较聪明，他给鱼竿上挂一个铃铛，当有鱼上钩的时候，这个铃铛就会被碰响，G就会将鱼钓上来。

以线程为单位：期间整个过程中是可以去做别的操作



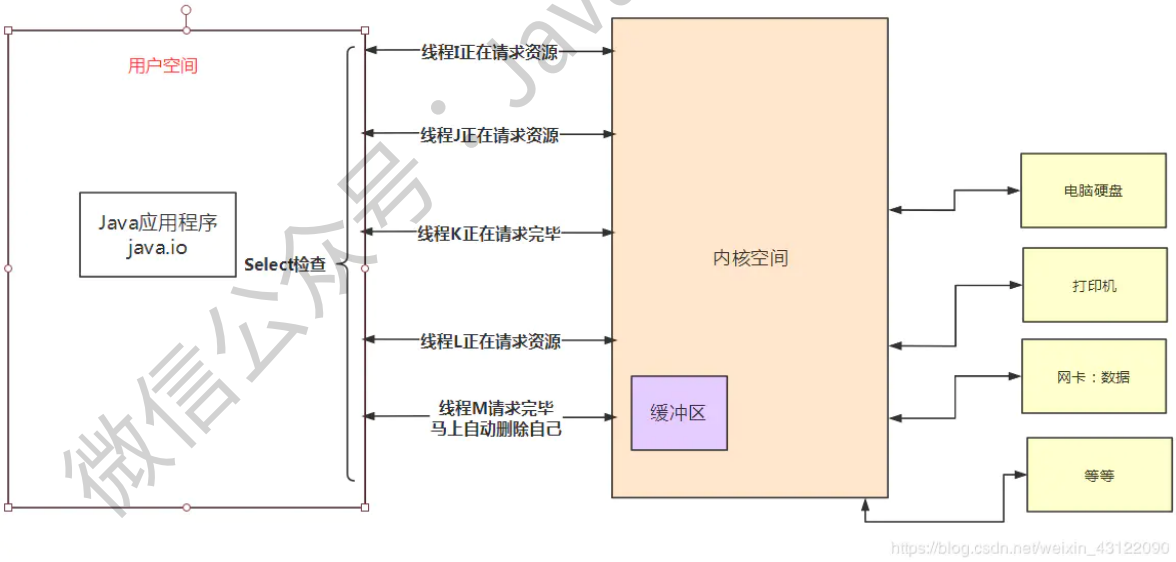
信号驱动IO模型，应用进程告诉内核：当数据报准备好的时候，给我发送一个信号，对SIGIO信号进行捕捉，并且调用我的信号处理函数来获取数据报。

> 信号驱动IO模型，应用进程告诉内核：当数据报准备好的时候，给我发送一个信号，对SIGIO信号进行捕捉，并且调用我的信号处理函数来获取数据报。

5.IO多路转接（I/O multiplexing）

- H同样也在河边钓鱼，但是H生活水平比较好，H拿了很多的鱼竿，一次性有很多鱼竿在等，H不断的查看每个鱼竿是否有鱼上钩。增加了效率，减少了等待的时间。

以线程为单位：期间整个过程中是可以去做别的操作



> IO多路转接是多了一个select函数，select函数有一个参数是文件描述符集合，对这些文件描述符进行循环监听，当某个文件描述符就绪时，就对这个文件描述符进行处理。

- IO多路转接是属于阻塞IO，但可以对多个文件描述符进行阻塞监听，所以效率较阻塞IO的高。

11. 什么是比特(Bit),什么是字节(Byte),什么是字符(Char),它们长度是多少,各有什么区别

- Bit最小的二进制单位， 是计算机的操作部分取值0或者1

- Byte是计算机中存储数据的单元，是一个8位的二进制数，（计算机内部，一个字节可表示一个英文字母，两个字节可表示一个汉字。） `取值（-128-127）`
- Char是用户的可读写的最小单位，他只是抽象意义上的一个符号。如'5'，'中'，'￥' 等等等等。在java里面由16位bit组成Char 取值 `（0-65535）`
- Bit 是最小单位 计算机他只能认识0或者1
- Byte是8个字节 是给计算机看的
- 字符 是看到的东西 一个字符=二个字节

12. 什么叫对象序列化，什么是反序列化，实现对象序列化需要做哪些工作

- 对象序列化，将对象以二进制的形式保存在硬盘上
- 反序列化；将二进制的文件转化为对象读取
- 实现serializable接口，不想让字段放在硬盘上就加transient

13. 在实现序列化接口是时候一般要生成一个serialVersionUID字段,它叫做什么,一般有什么用

- 如果用户没有自己声明一个serialVersionUID,接口会默认生成一个serialVersionUID
- 但是强烈建议用户自定义一个serialVersionUID,因为默认的serialVersinUID对于class的细节非常敏感，反序列化时可能会导致InvalidClassException这个异常。
- （比如说先进行序列化，然后在反序列化之前修改了类，那么就会报错。因为修改了类，对应的SerialversionUID也变化了，而序列化和反序列化就是通过对比其SerialversionUID来进行的，一旦SerialversionUID不匹配，反序列化就无法成功。

14. 怎么生成SerialversionUID

- 可序列化类可以通过声明名为 "serialVersionUID" 的字段（该字段**必须是静态 (static)、最终 (final) 的 long 型字段**）显式声明其自己的 serialVersionUID
- 两种显示的生成方式（当你一个类实现了Serializable接口，如果没有显示的定义serialVersionUID，Eclipse会提供这个提示功能告诉你去定义 。在Eclipse中点击类中warning的图标一下，Eclipse就会自动给定两种生成的方式。

15. BufferedReader属于哪种流,它主要是用来做什么的,它里面有那些经典的方法

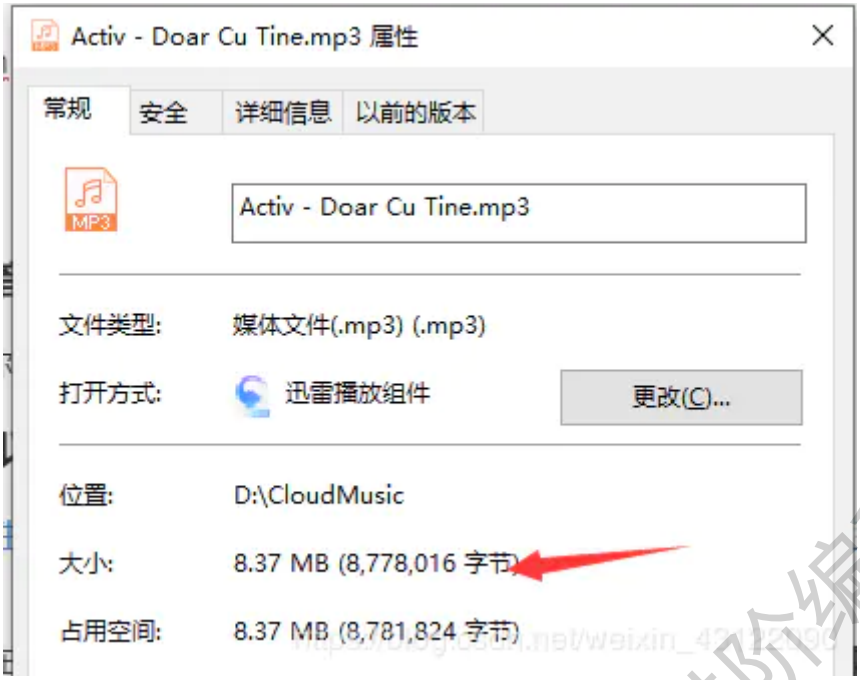
- 属于处理流中的缓冲流，可以将读取的内容存在内存里面，有readLine（）方法

16. Java中流类的超类主要有那些?

- 超类代表顶端的父类（都是抽象类）
- java.io.InputStream
- java.io.OutputStream
- java.io.Reader
- java.io.Writer

17. 为什么图片、视频、音乐、文件等 都是要字节流来读取

- 这个很基础，你看看你电脑文件的属性就好了，CPU规定了计算机存储文件都是按字节算的



18. IO的常用类和方法，以及如何使用

前面讲了那么多废话，现在我们开始进入主题，后面很长，从开始的文件操作到后面的**网络IO操作**都会有例子：

19. IO基本操作讲解

- 这里的基本操作就是普通的读取操作，如果想要跟深入的了解不同的IO开发场景必须先了解IO的基本操作

1 按字符流读取文件

1.1 按字符流的·节点流方式读取

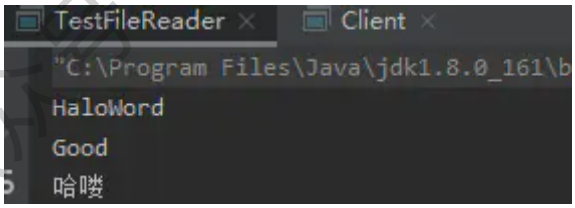
- 如果我们要取的数据基本单位是字符，那么用（**字符流**）这种方法读取文件就比较适合。比如：读取test.txt文件

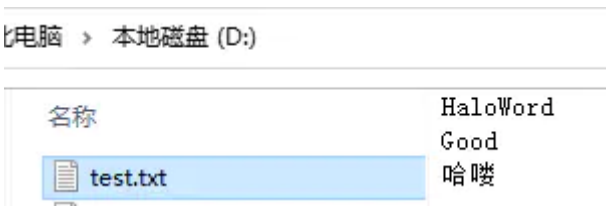
注释：

- `字符流`：以字符为单位，每次次读入或读出是16位数据。其只能读取字符类型数据。（Java代码接收数据为一般为`char数组，也可以是别的`）
- 字节流：以字节为单位，每次次读入或读出是8位数据。可以读任何类型数据，图片、文件、音乐视频等。（Java代码接收数据只能为`byte数组`）
- **FileReader 类：**（字符输入流） 注意：new FileReader("D:\test.txt");//文件必须存在

```
package com.test.io;

import java.io.FileReader;
import java.io.IOException;

public class TestFileReader {
    public static void main(String[] args) throws IOException {
        int num=0;
        //字符流接收使用的char数组
  char[] buf=new char[1024];
        //字符流、节点流打开文件类
  FileReader fr = new FileReader("D:\\test.txt");//文件必须存在
  //FileReader.read()：取出字符存到buf数组中,如果读取为-1代表为空即结束读取。
  //FileReader.read()：读取的是一个字符，但是java虚拟机会自动将char类型数据转换为int数据，
  //如果你读取的是字符A，java虚拟机会自动将其转换成97，如果你想看到字符可以在返回的字符数前加（char）强制转换如
  while((num=fr.read(buf))!=-1) { }
        //检测一下是否取到相应的数据
  for(int i=0;i<buf.length;i++) {
            System.out.print(buf[i]);
        }
    }
}
```

- **运行结果：**





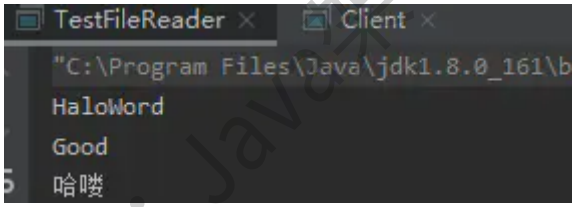
1.2 按字符流的·处理流方式读取

- 效果是一样，但是给了我们有不同的选择操作。进行了一个小封装，加缓冲功能，避免频繁读写硬盘。我这只是简单演示，处理流其实还有很多操作
- **BufferedReader 类：** 字符输入流使用的类，加缓冲功能，避免频繁读写硬盘

```
package com.test.io;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
```

```
public class TestBufferedReader {
    public static void main(String[] args) throws IOException {
        int num=0;
        //字符流接收使用的String数组
  String[] bufstring=new String[1024];
        //字符流、节点流打开文件类
  FileReader fr = new FileReader("D:\\test.txt");//文件必须存在
  //字符流、处理流读取文件类
  BufferedReader br = new BufferedReader(fr);
        //临时接收数据使用的变量
  String line=null;
        //BufferedReader.readLine()：单行读取，读取为空返回null
        while((line=br.readLine())!=null) {
            bufstring[num]=line;
            num++;
        }
        br.close();//关闭文件
  for(int i=0;i<num;i++) {
            System.out.println(bufstring[i]);
        }
    }
}
```

- 测试效果一样



2 按字符流写出文件

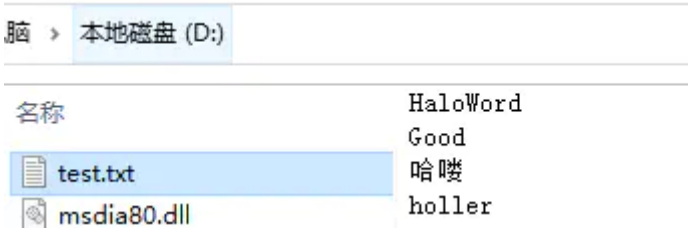
2.1 按字符流的·节点流方式写出

- 写出字符，使用（**字符流**）这种方法写出文件比较适合。比如：输出内容添加到test.txt文件
- **FileWriter类**：（字符输出流），如果写出文件不存在会自动创建一个相对应的文件。使用FileWriter写出文件默认是覆盖原文件，如果要想在源文件添加内容不覆盖的话，需要构造参数添加true参数：看示例了解

```
package com.test.io;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class TestFileWriter {
    public static void main(String[] args) throws IOException {
        //File是操作文件类
  File file = new File("D:\\test.txt");//文件必须存在
  //字符流、节点流写出文件类
  //new FileWriter(file,true)，这个true代表追加，不写就代表覆盖文件
  FileWriter out=new FileWriter(file,true);
        //写入的字节,\n代表换行
  String str="\nholler";
```

```
        //写入
  out.write(str);
        out.close();
    }
}
```

- **运行效果**:

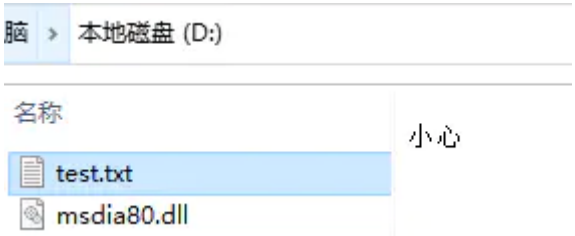


2.2 按字符流的·处理流方式写出

- **BufferedWriter**： 增加缓冲功能，避免频繁读写硬盘。 我这里： //new FileWriter(file)，这里我只给了他文件位置，我没加true代表覆盖源文件

```
package com.test.io;

import java.io.*;

public class TestBufferedWriter {
    public static void main(String[] args) throws IOException {
        //File是操作文件类
  File file = new File("D:\\test.txt");//文件必须存在
  //字符流、节点流写出文件类
  //new FileWriter(file)，这个我没加true代表覆盖文件
  Writer writer = new FileWriter(file);
        ////字符流、处理流写出文件类
  BufferedWriter bw = new BufferedWriter(writer);
        bw.write("\n小心");
        bw.close();
        writer.close();
    }
}
```

- **运行效果**:



3 按`字节`流写入写出文件

3.1 按字节流的·节点流写入写出文件

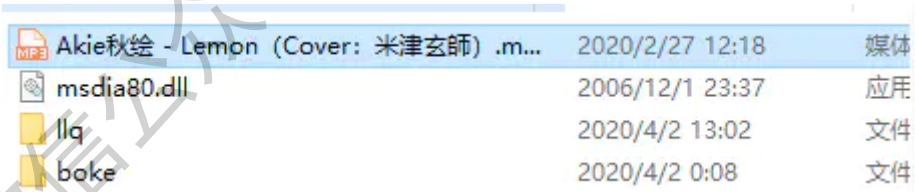
- 如果我们要取的数据 图片、文件、音乐视频等类型，就必须使用字节流进行读取写出

注释:

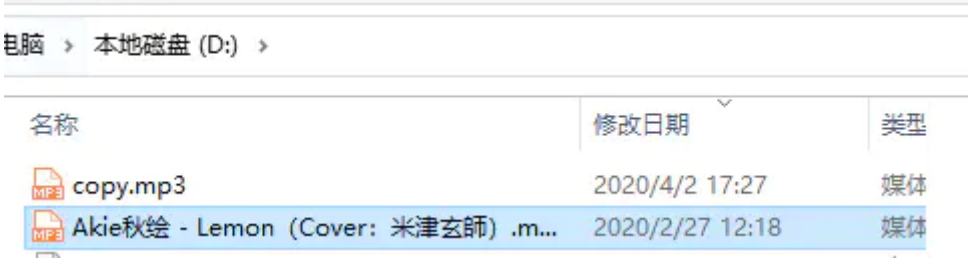
- `字符流`：以字符为单位，每次次读入或读出是16位数据。其只能读取字符类型数据。（Java代码接收数据为一般为`char数组，也可以是别的`）
- 字节流：以字节为单位，每次次读入或读出是8位数据。可以读任何类型数据，图片、文件、音乐视频等。（Java代码接收数据只能为`byte数组`）
- **FileInputStream**：（字节输入流）
- **FileOutputStream**：（字节输出流）

```
package com.test.io;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class TestFileOutputStream {
    public static void main(String[] args) throws IOException {
        //创建字节输入流、节点流方式读取文件
  FileInputStream fis = new FileInputStream("D:\\Akie秋绘 - Lemon（Cover：米津玄師）.mp3");
        //创建字节输入流、节点流方式输出文件
  FileOutputStream fos = new FileOutputStream("D:\\copy.mp3");

        //根据文件大小做一个字节数组
  byte[] arr = new byte[fis.available()];
        //将文件上的所有字节读取到数组中
  fis.read(arr);
        //将数组中的所有字节一次写到了文件上
  fos.write(arr);
        fis.close();
        fos.close();
    }
}
```

- **运行之前**：



运行之后：



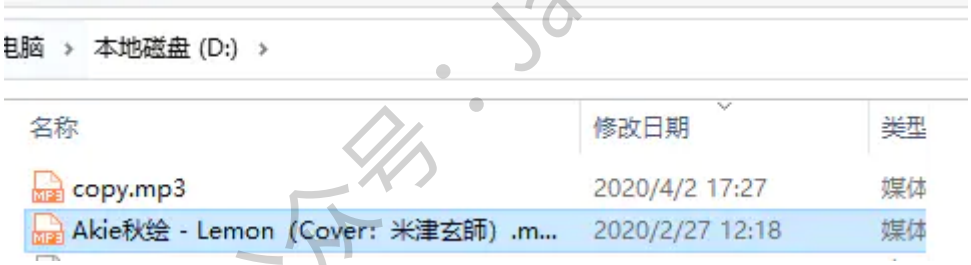
3.2 按字节流的·处理流写入写出文件

- **FileInputStream**：（字节输入流）
- **FileOutputStream**：（字节输出流）
- **BufferedInputStream**：（带缓冲区字节输入流）

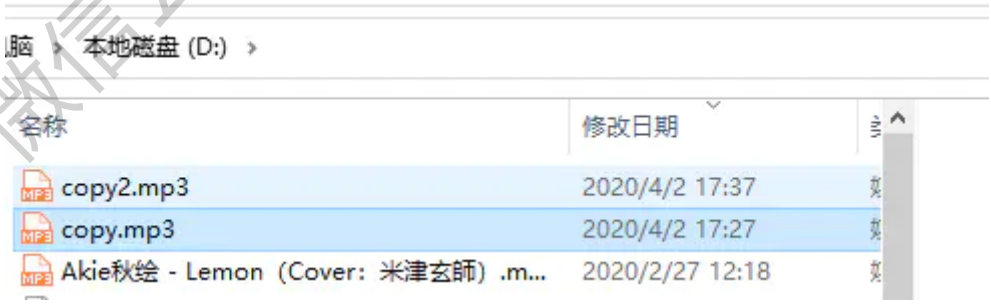
- **BufferedOutputStream**：（带缓冲区字节输入流） 带缓冲区的处理流，缓冲区的作用的主要目的是：避免每次和硬盘打交道，提高数据访问的效率。

```
package com.test.io;

import java.io.*;

public class TestBufferedOutputStream {
    //创建文件输入流对象,关联致青春.mp3
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("D:\\copy.mp3");
        //创建缓冲区对fis装饰
  BufferedInputStream bis = new BufferedInputStream(fis);
        //创建输出流对象,关联copy.mp3
        FileOutputStream fos = new FileOutputStream("D:\\copy2.mp3");
        //创建缓冲区对fos装饰
  BufferedOutputStream bos = new BufferedOutputStream(fos);
        //循环直接输出
  int i;
        while((i = bis.read()) != -1) {
            bos.write(i);
        }
        bis.close();
        bos.close();
    }
}
```

- **运行之前：**



运行之后：



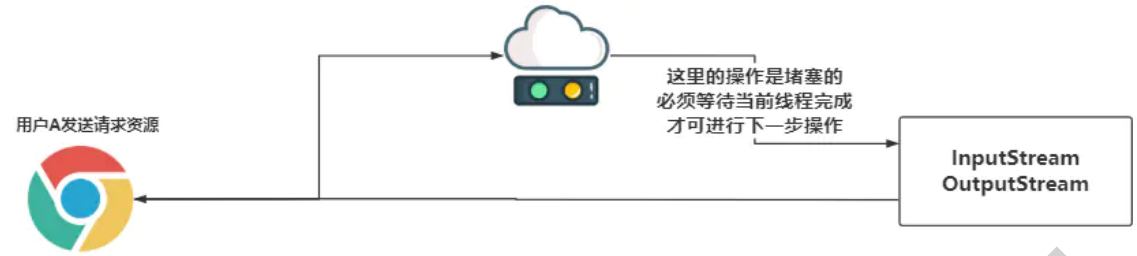
20. 网络操作IO讲解

- 我这使用Socket简单的来模拟网络编程IO会带来的问题
- 不懂Socket可以看我之前的文章，这个东西很容易懂的，就是基于TCP实现的网络通信，比http要快，很多实现网络通信的框架都是基于Socket来实现

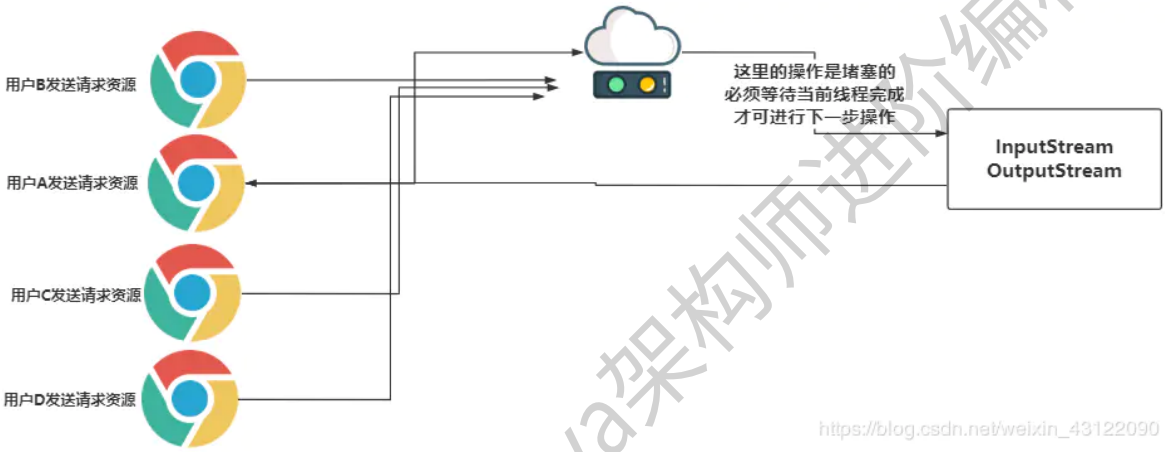
21. 网络操作IO编程演变历史

1 BIO编程会出现什么问题?

- BIO是阻塞的
- **例子：** 阻塞IO（blocking I/O） A拿着一支鱼竿在河边钓鱼，并且一直在鱼竿前等，在等的时候不做其他的事情，十分专心。只有鱼上钩的时，才结束掉等的动作，把鱼钓上来。



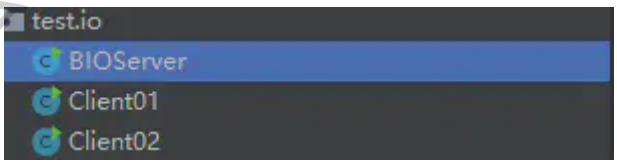
- 看起来没问题，但是我很多请求一起发送请求资源怎么办：



> **那不是要等待第一个人资源完成后后面的人才可以继续？** 因为BIO是阻塞的所以读取写出操作都是非常浪费资源的

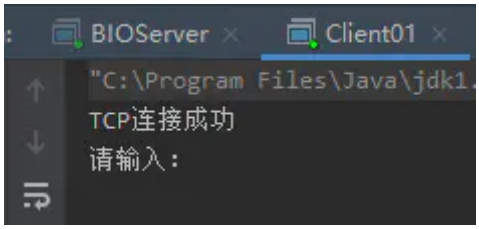
BIO代码示例： （后面有代码，往后移动一点点，认真看，代码学习量很足）

- 我这有三个类，我模拟启动服务端，然后启动客户端，模拟客户端操作未完成的时候启动第二个客户端



1. 启动服务端（后面有代码，我这是教运行顺序）

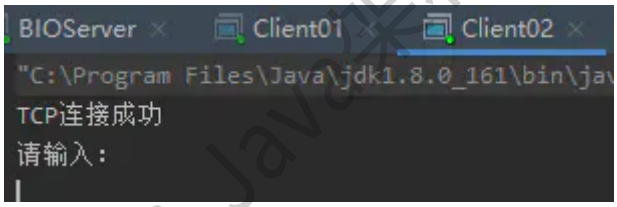
2. 启动第一个客户端，发现服务器显示连接成功 先不要在控制台 输入 ，模拟堵塞。（我的代码输入了就代表请求完成了）



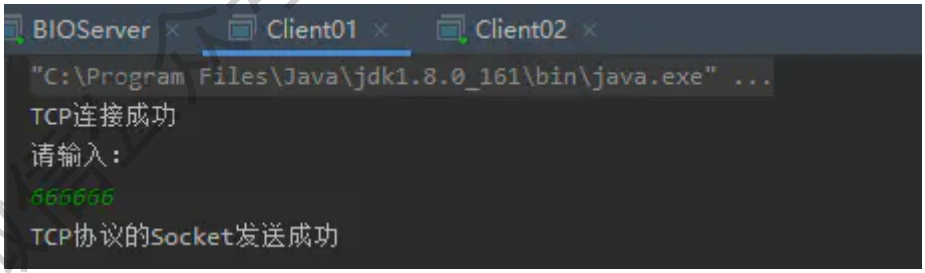
.

3. 启动第二个客户端，`发现服务端没效果`，而客户端连接成功（在堵塞当中）`我这启动了俩个Client，注意看，(这俩个代码是一样的)`

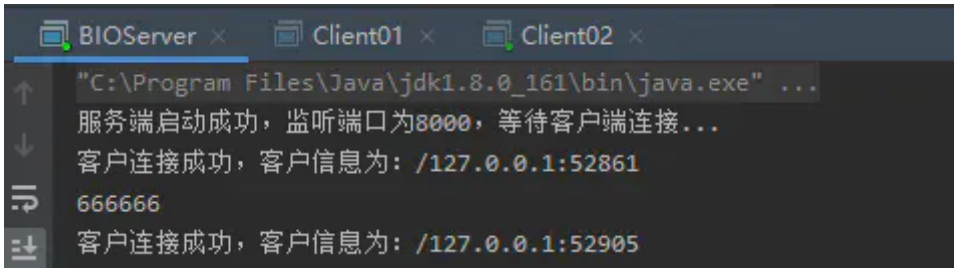
.



4. 第一个客户控制台输入，输入完后就会关闭第一个客户端， 在看服务端发现第二个客户端连接上来了



.



BIO通信代码：

- TCP协议Socket使用BIO进行通信：服务端（先执行）

```
package com.test.io;

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

//TCP协议Socket使用BIO进行通信：服务端
public class BIOServer {
    // 在main线程中执行下面这些代码
  public static void main(String[] args) {
        //使用Socket进行网络通信
  ServerSocket server = null;
        Socket socket = null;
        //基于字节流
  InputStream in = null;
        OutputStream out = null;
        try {
            server = new ServerSocket(8000);
            System.out.println("服务端启动成功，监听端口为8000，等待客户端连接...");
            while (true){
                socket = server.accept(); //等待客户端连接
  System.out.println("客户连接成功，客户信息为：" +
socket.getRemoteSocketAddress());
                in = socket.getInputStream();
                byte[] buffer = new byte[1024];
                int len = 0;
                //读取客户端的数据
  while ((len = in.read(buffer)) > 0) {
                    System.out.println(new String(buffer, 0, len));
                }
                //向客户端写数据
  out = socket.getOutputStream();
                out.write("hello!".getBytes());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- TCP协议Socket使用BIO进行通信：客户端（第二执行）

```
package com.test.io;

import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Scanner;

//TCP协议Socket使用BIO进行通信：客户端
public class Client01 {
    public static void main(String[] args) throws IOException {
        //创建套接字对象socket并封装ip与port
        Socket socket = new Socket("127.0.0.1", 8000);
```

```
        //根据创建的socket对象获得一个输出流
  //基于字节流
  OutputStream outputStream = socket.getOutputStream();
        //控制台输入以IO的形式发送到服务器
  System.out.println("TCP连接成功 \n请输入：");
        String str = new Scanner(System.in).nextLine();
        byte[] car = str.getBytes();
        outputStream.write(car);
        System.out.println("TCP协议的Socket发送成功");
        //刷新缓冲区
  outputStream.flush();
        //关闭连接
  socket.close();
    }
}
```

- TCP协议Socket使用BIO进行通信：客户端（第三执行）

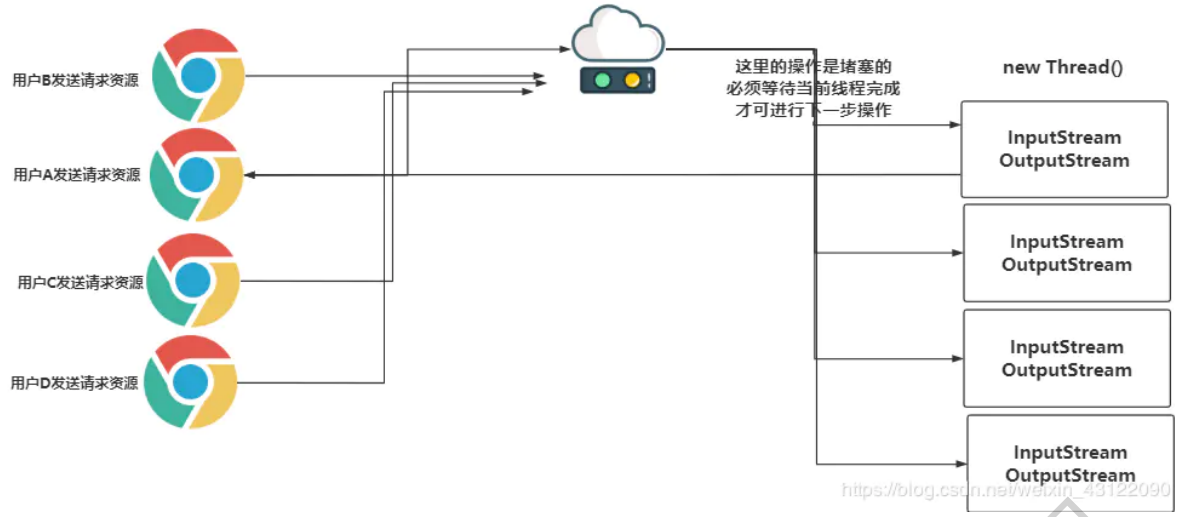
```
package com.test.io;

import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Scanner;

//TCP协议Socket：客户端
public class Client02 {
    public static void main(String[] args) throws IOException {
        //创建套接字对象socket并封装ip与port
        Socket socket = new Socket("127.0.0.1", 8000);
        //根据创建的socket对象获得一个输出流
  //基于字节流
  OutputStream outputStream = socket.getOutputStream();
        //控制台输入以IO的形式发送到服务器
  System.out.println("TCP连接成功 \n请输入：");
        String str = new Scanner(System.in).nextLine();
        byte[] car = str.getBytes();
        outputStream.write(car);
        System.out.println("TCP协议的Socket发送成功");
        //刷新缓冲区
  outputStream.flush();
        //关闭连接
  socket.close();
    }
}
```

为了解决堵塞问题，可以使用多线程，请看下面

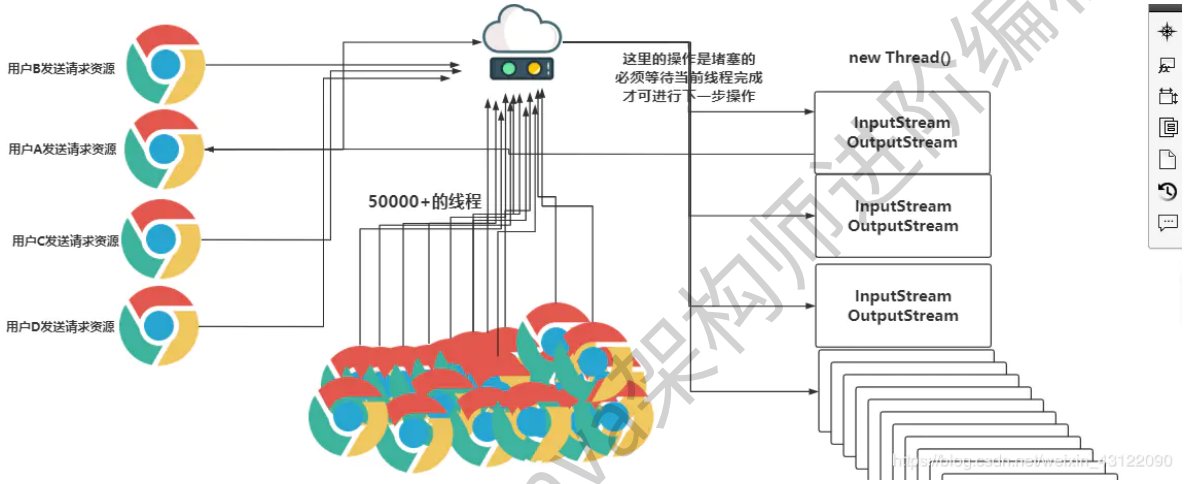
2 多线程解决BIO编程会出现的问题

这时有人就会说，我多线程不就解决了吗?

- 使用多线程是可以解决堵塞等待时间很长的问题，因为他可以充分发挥CPU
- 然而系统资源是有限的，不能过多的新建线程，线程过多带来线程上下文的切换，从来带来更大的性能损耗

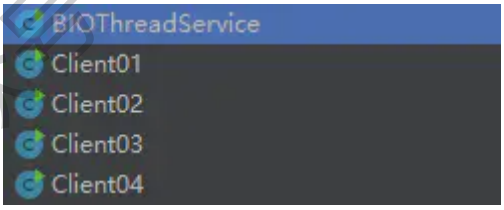


万一请求越来越多，线程越来越多那我CPU不就炸了？

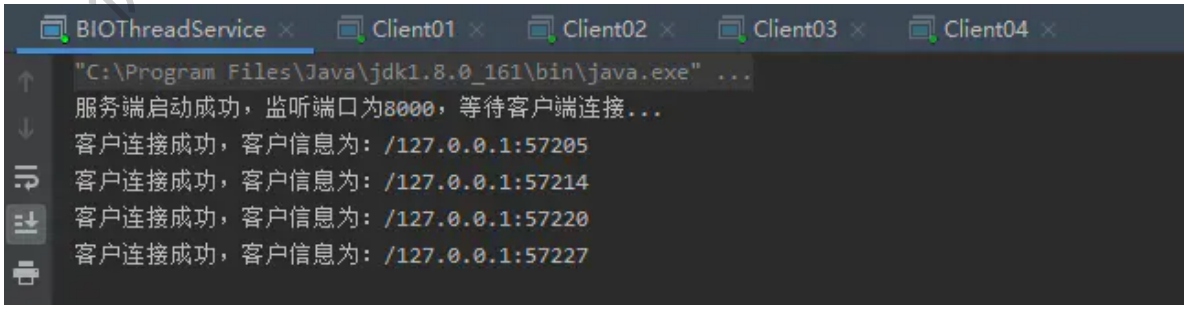


多线程BIO代码示例：

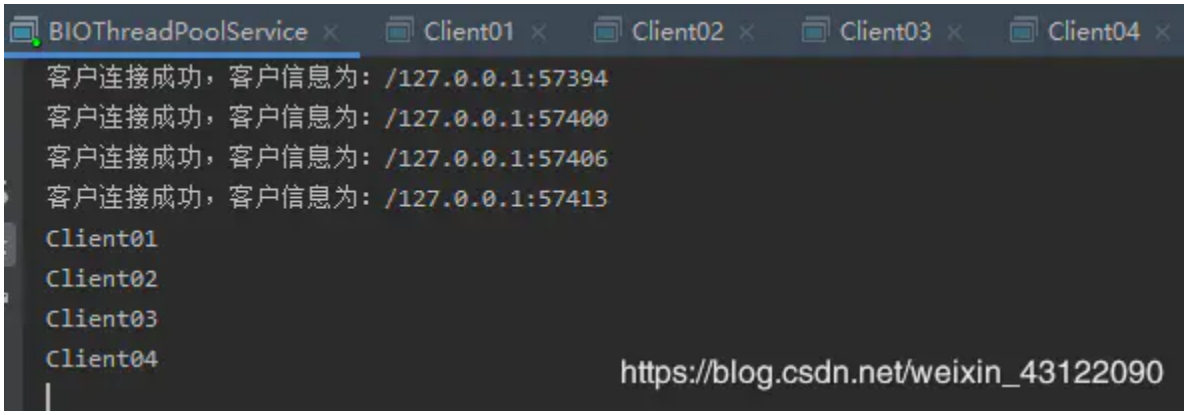
- 四个客户端，这次我多复制了俩个一样客户端类



`先启动服务端，在启动所有客户端，测试`，发现连接成功（`后面有代码`）



在所有客户端输入消息（`Client01、Client02这些是我在客户端输入的消息`）：发现没有问题



多线程BIO通信代码：

- 服务端的代码，客户端的代码还是上面之前的代码

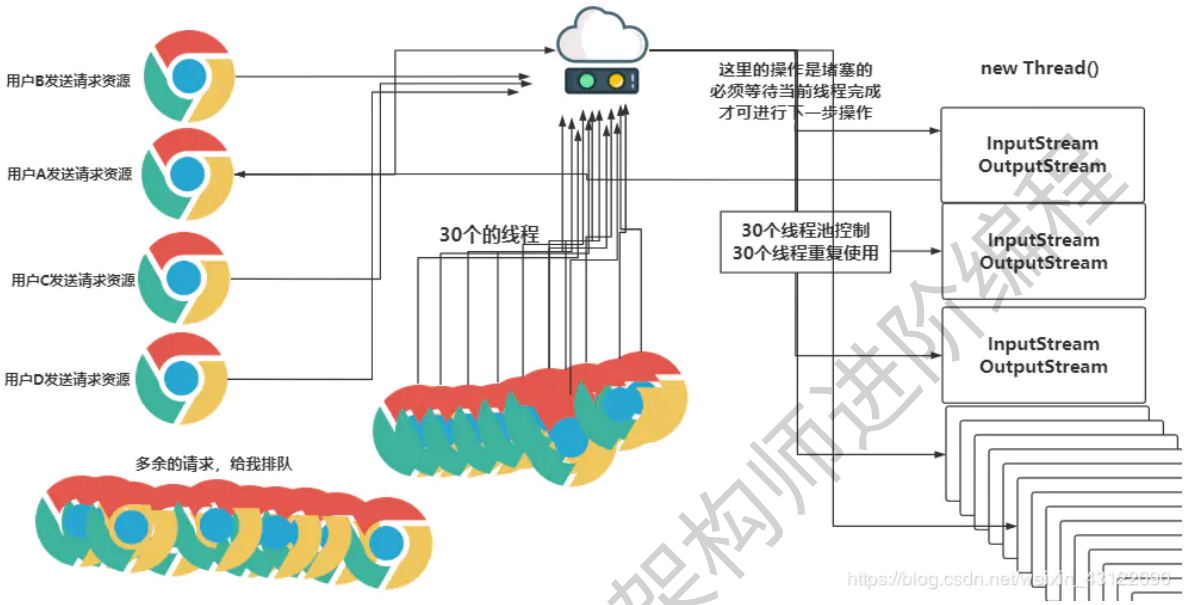
```
package com.test.io;

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

//TCP协议Socket使用多线程BIO进行通行：服务端
public class BIOThreadService {
    public static void main(String[] args) {
        try {
            ServerSocket server = new ServerSocket(8000);
            System.out.println("服务端启动成功，监听端口为8000，等待客户端连接... ");
            while (true) {
                Socket socket = server.accept();//等待客户连接
  System.out.println("客户连接成功，客户信息为：" +
socket.getRemoteSocketAddress());
                //针对每个连接创建一个线程， 去处理IO操作
  //创建多线程创建开始
  Thread thread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            InputStream in = socket.getInputStream();
                            byte[] buffer = new byte[1024];
                            int len = 0;
                            //读取客户端的数据
  while ((len = in.read(buffer)) > 0) {
                                System.out.println(new String(buffer, 0, len));
                            }
                            //向客户端写数据
  OutputStream out = socket.getOutputStream();
                            out.write("hello".getBytes());
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                });
                thread.start();
            }
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

为了解决线程太多，这时又来了，线程池

3 线程池解决多线程BIO编程会出现的问题

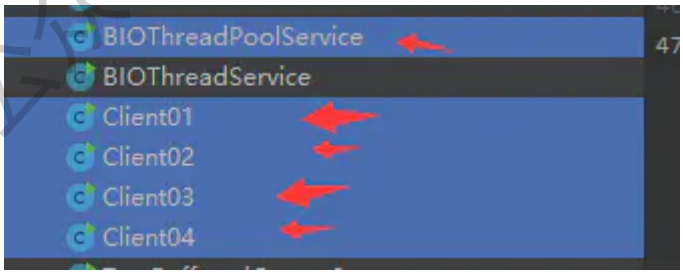
这时有人就会说，我TM用线程池?



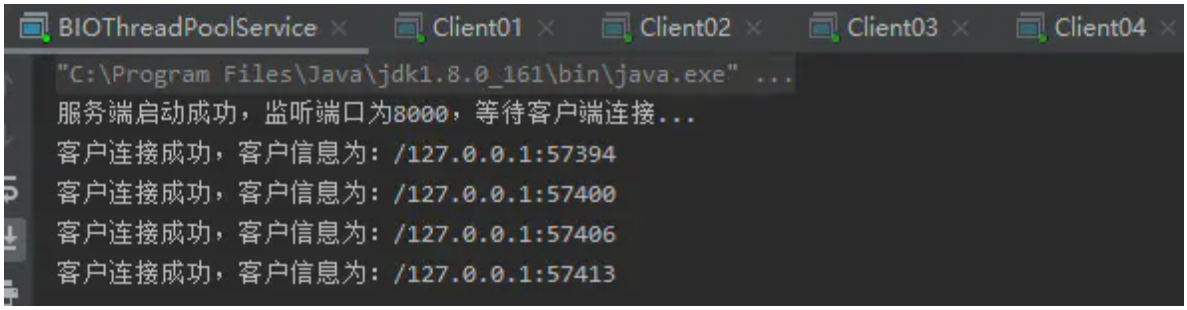
- 线程池固然可以解决这个问题，万一需求量还不够还要扩大线程池。当是这是我们自己靠着自己的思想完成的IO操作，Socket 上来了就去创建线程去抢夺CPU资源，MD，线程都TM做IO去了，CPU也不舒服呀
- 这时呢：Jdk官方坐不住了，兄弟BIO的问题交给我，我来给你解决：`NIO的诞生`

线程池BIO代码示例：

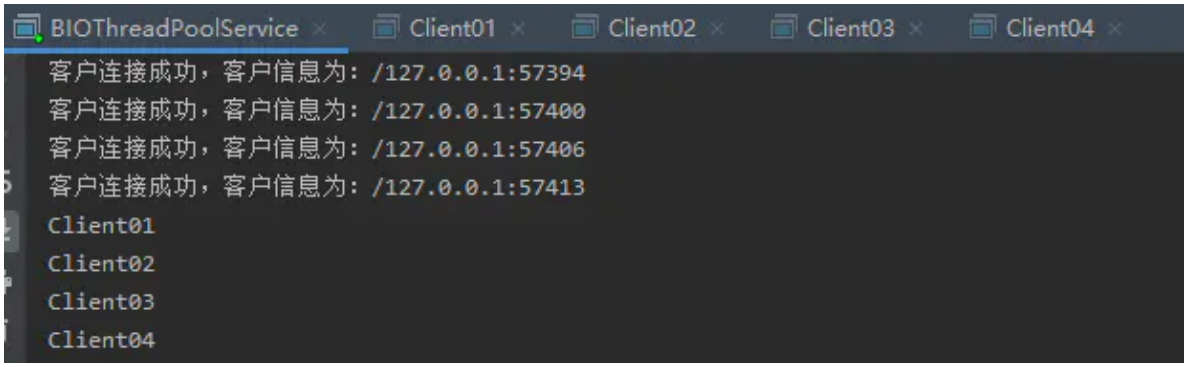
- 四个客户端



`先启动服务端，在启动所有客户端，测试`，（`后面有代码`）



在所有客户端输入消息（`Client01、Client02这些是我在客户端输入的消息`）：发现没有问题



线程池BIO通信代码：

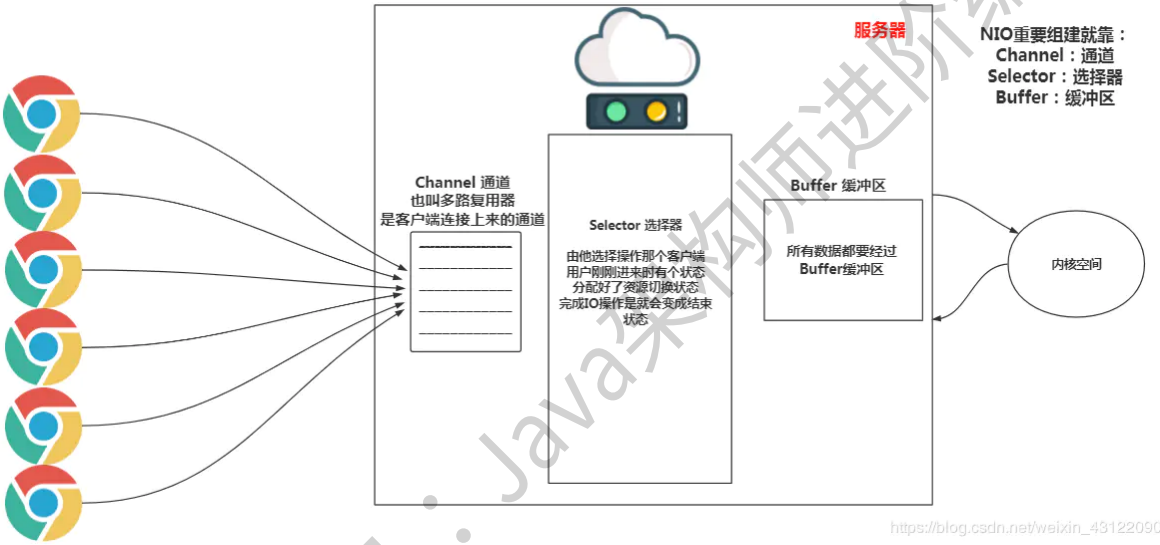
- 服务端的代码，客户端的代码还是上面的代码

```
package com.test.io;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

//TCP协议Socket使用线程池BIO进行通行：服务端
public class BIOThreadPoolService {
    public static void main(String[] args) {
        //创建线程池
  ExecutorService executorService = Executors.newFixedThreadPool(30);
        try {
            ServerSocket server = new ServerSocket(8000);
            System.out.println("服务端启动成功，监听端口为8000，等待客户端连接...");
            while (true) {
                Socket socket = server.accept();//等待客户连接
  System.out.println("客户连接成功，客户信息为：" +
socket.getRemoteSocketAddress());
                //使用线程池中的线程去执行每个对应的任务
  executorService.execute(new Thread(new Runnable() {
                    public void run() {
                        try {
                            InputStream in = socket.getInputStream();
                            byte[] buffer = new byte[1024];
                            int len = 0;
                            //读取客户端的数据
  while ((len = in.read(buffer)) > 0) {
                                System.out.println(new String(buffer, 0, len));
                            }
                            //向客户端写数据
  OutputStream out = socket.getOutputStream();
                            out.write("hello".getBytes());
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
```

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

4 使用NIO实现网络通信

- NIO是JDK1.4提供的操作，他的流还是流，没有改变，服务器实现的还是一个连接一个线程，当是：`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。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之后开始支持。



`看不懂介绍可以认真看看代码实例，其实不难`

什么是通道（Channel）

- Channel是一个对象，可以通过它读取和写入数据。 通常我们都是将数据写入包含一个或者多个字节的缓冲区，然后再将缓存区的数据写入到通道中，将数据从通道读入缓冲区，再从缓冲区获取数据。
- Channel 类似于原I/O中的流（Stream），但有所区别：
  - 流是单向的，通道是双向的，可读可写。
  - 流读写是阻塞的，通道可以异步读写。

什么是选择器（Selector）

- Selector可以称他为通道的集合，每次客户端来了之后我们会把Channel注册到Selector中并且我们给他一个状态，在用死循环来环判断(`判断是否做完某个操作，完成某个操作后改变不一样的状态`)状态是否发生变化，知道IO操作完成后在退出死循环

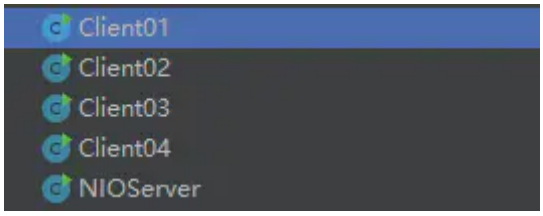
什么是Buffer（缓冲区）

- Buffer 是一个缓冲数据的对象， 它包含一些要写入或者刚读出的数据。
- 在普通的面向流的 I/O 中，一般将数据直接写入或直接读到 Stream 对象中。当是有了Buffer（缓冲区）后，数据第一步到达的是Buffer（缓冲区）中

- 缓冲区实质上是一个数组(`底层完全是数组实现的，感兴趣可以去看一下`)。通常它是一个字节数组，内部维护几个状态变量，可以实现在同一块缓冲区上反复读写（不用清空数据再写）。

代码实例：

- 目录结构



- 运行示例，先运行服务端，在运行所有客户端控制台输入消息就好了。：`我这客户端和服务端代码有些修该变，后面有代码`

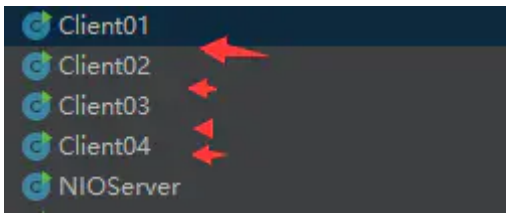
- `服务端示例，先运行，想要搞定NIO请认真看代码示例，真的很清楚`

```
package com.test.io;

import com.lijie.iob.RequestHandler;

import java.io.IOException;
import java.net.InetSocketAddress;
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;
import java.util.Set;

public class NIOServer {
    public static void main(String[] args) throws IOException {
        //111111111
        //Service端的Channel，监听端口的
  ServerSocketChannel serverChannel = ServerSocketChannel.open();
        //设置为非阻塞
  serverChannel.configureBlocking(false);
        //nio的api规定这样赋值端口
  serverChannel.bind(new InetSocketAddress(8000));
        //显示Channel是否已经启动成功，包括绑定在哪个地址上
  System.out.println("服务端启动成功，监听端口为8000，等待客户端连接..."+
serverChannel.getLocalAddress());
```

```
        //22222222
        //声明selector选择器
  Selector selector = Selector.open();
        //这句话的含义，是把selector注册到Channel上面，
  //每个客户端来了之后，就把客户端注册到Selector选择器上,默认状态是Accepted
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);

        //33333333
        //创建buffer缓冲区，声明大小是1024，底层使用数组来实现的
  ByteBuffer buffer = ByteBuffer.allocate(1024);
        RequestHandler requestHandler = new RequestHandler();

        //444444444
        //轮询，服务端不断轮询，等待客户端的连接
  //如果有客户端轮询上来就取出对应的Channel，没有就一直轮询
  while (true) {
            int select = selector.select();
            if (select == 0) {
                continue;
            }
            //有可能有很多，使用Set保存Channel
            Set<SelectionKey> selectionKeys = selector.selectedKeys();
            Iterator<SelectionKey> iterator = selectionKeys.iterator();
            while (iterator.hasNext()) {
                //使用SelectionKey来获取连接了客户端和服务端的Channel
                SelectionKey key = iterator.next();
                //判断SelectionKey中的Channel状态如何，如果是OP_ACCEPT就进入
  if (key.isAcceptable()) {
                    //从判断SelectionKey中取出Channel
                    ServerSocketChannel channel = (ServerSocketChannel)
key.channel();
                    //拿到对应客户端的Channel
                    SocketChannel clientChannel = channel.accept();
                    //把客户端的Channel打印出来
  System.out.println("客户端通道信息打印：" + clientChannel.getRemoteAddress());
                    //设置客户端的Channel设置为非阻塞
  clientChannel.configureBlocking(false);
                    //操作完了改变SelectionKey中的Channel的状态OP_READ
                    clientChannel.register(selector, SelectionKey.OP_READ);
                }
                //到此轮训到的时候，发现状态是read，开始进行数据交互
  if (key.isReadable()) {
                    //以buffer作为数据桥梁
  SocketChannel channel = (SocketChannel) key.channel();
                    //数据要想读要先写，必须先读取到buffer里面进行操作
  channel.read(buffer);
                    //进行读取
  String request = new String(buffer.array()).trim();
                    buffer.clear();
                    //进行打印buffer中的数据
  System.out.println(String.format("客户端发来的消息： %s : %s",
channel.getRemoteAddress(), request));
                    //要返回数据的话也要先返回buffer里面进行返回
  String response = requestHandler.handle(request);
                    //然后返回出去
  channel.write(ByteBuffer.wrap(response.getBytes()));
                }
```

```
                iterator.remove();
            }
        }
    }
}
```

- 客户端示例：（我这用的不是之前的了，有修改）运行起来客户端控制台输入消息就好了。 要模拟测试，请复制粘贴改一下，修改客户端的类名就行了，四个客户端代码一样的，



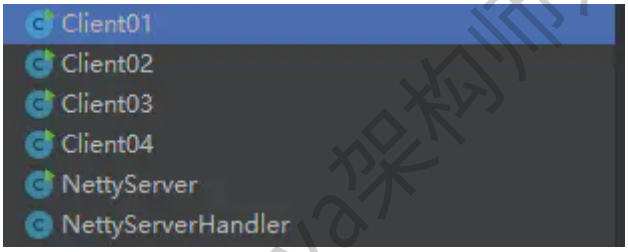

```
package com.test.io;

import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Scanner;

//TCP协议Socket：客户端
public class Client01 {
    public static void main(String[] args) throws IOException {
        //创建套接字对象socket并封装ip与port
        Socket socket = new Socket("127.0.0.1", 8000);
        //根据创建的socket对象获得一个输出流
  OutputStream outputStream = socket.getOutputStream();
        //控制台输入以IO的形式发送到服务器
  System.out.println("TCP连接成功 \n请输入：");
        while(true){
            byte[] car = new Scanner(System.in).nextLine().getBytes();
            outputStream.write(car);
            System.out.println("TCP协议的Socket发送成功");
            //刷新缓冲区
  outputStream.flush();
        }
    }
}
```


5 使用Netty实现网络通信

- Netty是由JBOSS提供的一个Java开源框架。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。
- Netty 是一个基于NIO的客户、服务器端编程框架，使用Netty 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户，服务端应用。Netty相当简化和流线化了网络应用的编程开发过程，例如，TCP和UDP的Socket服务开发。

Netty是由NIO演进而来，使用过NIO编程的用户就知道NIO编程非常繁重，Netty是能够能跟好的使用NIO

- Netty的原里就是NIO，他是基于NIO的一个完美的封装，并且优化了NIO，使用他非常方便，简单快捷
- 我直接上代码：



- 1、先添加依赖：

```
      <dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
            <version>4.1.16.Final</version>
        </dependency>
```

- 2、NettyServer 模板，看起来代码那么多，其实只需要添加一行消息就好了
- 请认真看中间的代码

```
package com.lijie.iob;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.serialization.ClassResolvers;
import io.netty.handler.codec.serialization.ObjectEncoder;
import io.netty.handler.codec.string.StringDecoder;

public class NettyServer {
    public static void main(String[] args) throws InterruptedException {
        EventLoopGroup bossGroup = new NioEventLoopGroup();
```

```
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel socketChannel)
throws Exception {
                            ChannelPipeline pipeline = socketChannel.pipeline();
                            pipeline.addLast(new StringDecoder());
                            pipeline.addLast("encoder", new ObjectEncoder());
                            pipeline.addLast(" decoder", new
io.netty.handler.codec.serialization.ObjectDecoder(Integer.MAX_VALUE,
ClassResolvers.cacheDisabled(null)));

                            //重点，其他的都是复用的
  //这是真正的IO的业务代码，把他封装成一个个的个Hand1e类就行了
  //把他当成 SpringMVC的Controller
                            pipeline.addLast(new NettyServerHandler());

                        }
                    })
                    .option(ChannelOption.SO_BACKLOG, 128)
                    .childOption(ChannelOption.SO_KEEPALIVE, true);
            ChannelFuture f = b.bind(8000).sync();
            System.out.println("服务端启动成功，端口号为:" + 8000);
            f.channel().closeFuture().sync();
        } finally {
            workerGroup.shutdownGracefully();
            bossGroup.shutdownGracefully();
        }
    }
}
```

- 3、需要做的IO操作，重点是继承ChannelInboundHandlerAdapter类就好了

```
package com.lijie.iob;

import io.netty.channel.Channel;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

public class NettyServerHandler extends ChannelInboundHandlerAdapter {
    RequestHandler requestHandler = new RequestHandler();

    @Override
    public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
        Channel channel = ctx.channel();
        System.out.println(String.format("客户端信息： %s",
channel.remoteAddress()));
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
```

```
        Channel channel = ctx.channel();
        String request = (String) msg;
        System.out.println(String.format("客户端发送的消息 %s : %s",
channel.remoteAddress(), request));
        String response = requestHandler.handle(request);
        ctx.write(response);
        ctx.flush();
    }
}
```

- 4 客户的代码还是之前NIO的代码，我在复制下来一下吧

```
package com.test.io;

import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Scanner;

//TCP协议Socket：客户端
public class Client01 {
    public static void main(String[] args) throws IOException {
        //创建套接字对象socket并封装ip与port
        Socket socket = new Socket("127.0.0.1", 8000);
        //根据创建的socket对象获得一个输出流
  OutputStream outputStream = socket.getOutputStream();
        //控制台输入以IO的形式发送到服务器
  System.out.println("TCP连接成功 \n请输入：");
        while(true){
            byte[] car = new Scanner(System.in).nextLine().getBytes();
            outputStream.write(car);
            System.out.println("TCP协议的Socket发送成功");
            //刷新缓冲区
  outputStream.flush();
        }
    }
}
```

- 运行测试，还是之前那样，启动服务端，在启动所有客户端控制台输入就好了：