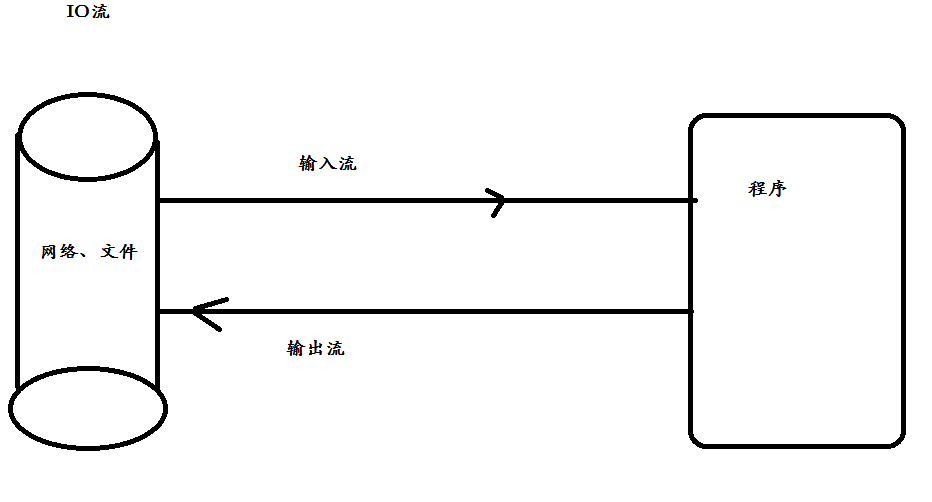
# javaIO编程.

随着java版本的不断升级与迭代，java的IO模型开始得到改变。从原始的BIO,到1.4以后发布的NIO,再到对NIO进行的改进AIO分别对IO模型做了优化，BIO是同步，阻塞的IO.

NIO是同步，非阻塞的IO,AIO是异步非阻塞的IO,那么性能就自然不用多说了，肯定是依次得到了提升的。

## 一 文件

基本的IO流与IO 模型,可以说它是通过序列化后的字节流来进行基本的通信的。



### BIO

#### 1.1.1无缓冲区IO

**public** **class** BIONOBuffer {

@SuppressWarnings("resource")

**public** **static** **void** main(String[] args) {

**long** timeA=System.*currentTimeMillis*();

FileInputStream bur=**null**;

FileOutputStream bor=**null**;

**try**{

bur = **new** FileInputStream(**new** File("C:/Users/pc/Desktop/[www.java1234.com]Activiti实战 PDF电子书-含书签目录.pdf"));

bor = **new** FileOutputStream(**new** File("C:/Users/pc/Desktop/拷贝/cs.pdf"));

**byte**[] buf=**new** **byte**[1024];

**while**((bur.read(buf))!=-1){

bor.write(buf);

bor.flush();

}

}**catch**(Exception exception){

exception.printStackTrace();

}**finally**{

**try** {

**if**(bur!=**null**){

bur.close();

}

**if**(bor!=**null**){

bor.close();

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**long** timeB=System.*currentTimeMillis*();

System.***out***.println((timeB-timeA)+" ms");

}

}

无缓冲区的用时(514+460+467)/3=480.333ms

#### 1.1.2有缓冲区IO

**public** **class** IOTransport {

**public** **static** **void** main(String[] args) {

**long** timeA=System.*currentTimeMillis*();

BufferedInputStream bur=**null**;

BufferedOutputStream bor=**null**;

**try**{

bur=**new** BufferedInputStream(**new** FileInputStream(**new** File("C:/Users/pc/Desktop/[www.java1234.com]Activiti实战 PDF电子书-含书签目录.pdf")));

bor=**new** BufferedOutputStream(**new** FileOutputStream(**new** File("C:/Users/pc/Desktop/拷贝/cs.pdf")));

**byte**[] buf=**new** **byte**[1024];

**while**(bur.read(buf)!=-1){

bor.write(buf);

bor.flush();

}

}**catch**(Exception exception){

exception.printStackTrace();

}**finally**{

**try** {

**if**(bur!=**null**){

bur.close();

}

**if**(bor!=**null**){

bor.close();

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**long** timeB=System.*currentTimeMillis*();

System.***out***.println((timeB-timeA)+" ms");

}

}

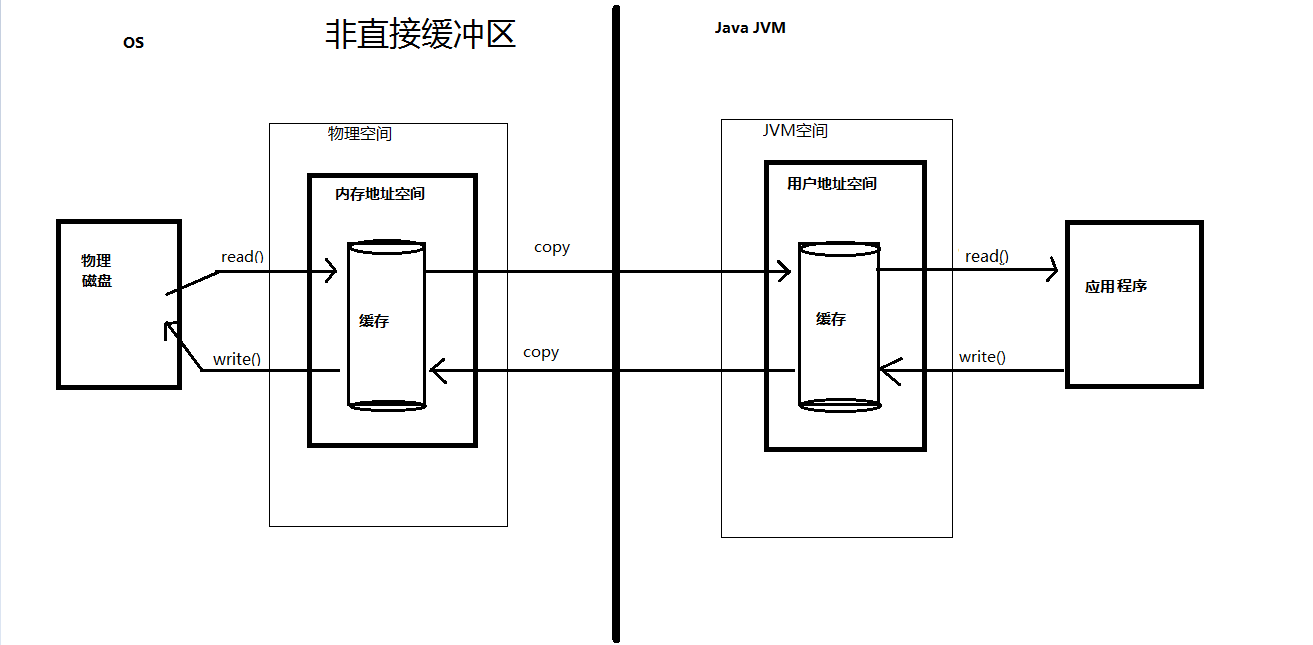
读取时间计算(401+368+372)/3=380.33ms

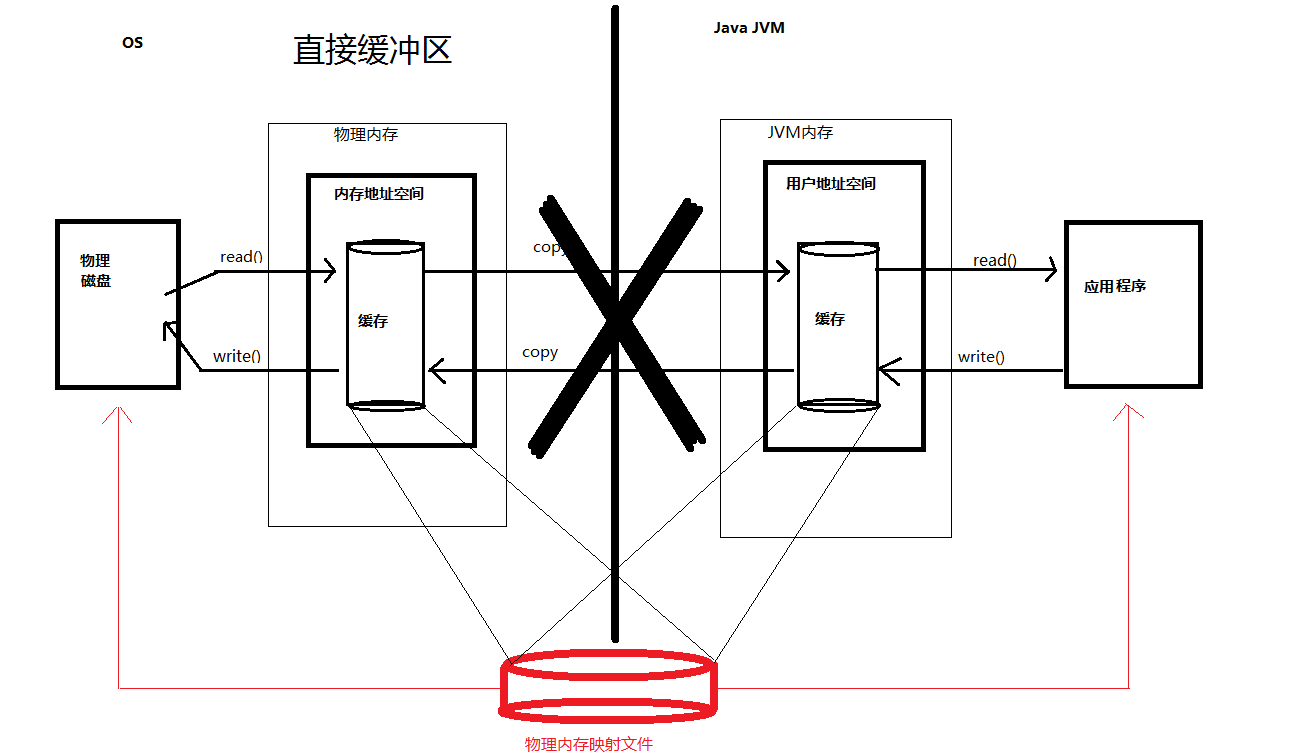
### NIO

#### jkd1.7之前NIO

关于直接缓冲区与非直接缓冲区的区别

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



字节缓冲区要么是直接的，要么是非直接的。如果为直接字节缓冲区，则 Java 虚拟机会尽最大努力直接在此缓冲区上执行本机 I/O 操作。也就是说，在每次调用基础操作系统的一个本机 I/O 操作之前（或之后），虚拟机都会尽量避免将缓冲区的内容复制到中间缓冲区中（或从中间缓冲区中复制内容）。

直接字节缓冲区可以通过调用此类的 allocateDirect() 工厂方法来创建。此方法返回的缓冲区进行分配和取消分配所需成本通常高于非直接缓冲区。直接缓冲区的内容可以驻留在常规的垃圾回收堆之外，因此，它们对应用程序的内存需求量造成的影响可能并不明显。所以，建议将直接缓冲区主要分配给那些易受基础系统的本机 I/O 操作影响的大型、持久的缓冲区。一般情况下，最好仅在直接缓冲区能在程序性能方面带来明显好处时分配它们。

直接字节缓冲区还可以通过 FileChannel 的 map() 方法 将文件区域直接映射到内存中来创建。该方法返回MappedByteBuffer 。 Java 平台的实现有助于通过 JNI 从本机代码创建直接字节缓冲区。如果以上这些缓冲区中的某个缓冲区实例指的是不可访问的内存区域，则试图访问该区域不会更改该缓冲区的内容，并且将会在访问期间或稍后的某个时间导致抛出不确定的异常。

字节缓冲区是直接缓冲区还是非直接缓冲区可通过调用其 isDirect() 方法来确定。提供此方法是为了能够在性能关键型代码中执行显式缓冲区管理。

##### 非直接缓冲区

**public** **class** BIOTransport {

**public** **static** **void** main(String[] args) {

**long** timeA=System.*currentTimeMillis*();

FileInputStream fileInputStream=**null**;

FileOutputStream fileOutputStream=**null**;

FileChannel inchan=**null**;

FileChannel outchan=**null**;

**try**{

fileInputStream = **new** FileInputStream(**new** File("C:/Users/pc/Desktop/[www.java1234.com]Activiti实战 PDF电子书-含书签目录.pdf"));

fileOutputStream = **new** FileOutputStream(**new** File("C:/Users/pc/Desktop/拷贝/cs.pdf"));

inchan = fileInputStream.getChannel();

outchan = fileOutputStream.getChannel();

//ByteBuffer byteBuffer=ByteBuffer.allocateDirect(1024);

ByteBuffer byteBuffer=ByteBuffer.*allocate*(1024);

**while**(inchan.read(byteBuffer)!=-1){

//开启读模式

byteBuffer.flip();

outchan.write(byteBuffer);

byteBuffer.clear();

}

}**catch**(Exception exception){

exception.printStackTrace();

}**finally**{

**try** {

**if**(fileInputStream!=**null**){

fileInputStream.close();

}

**if**(fileOutputStream!=**null**){

fileOutputStream.close();

}

**if**(inchan!=**null**){

inchan.close();

}

**if**(outchan!=**null**){

outchan.close();

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**long** timeB=System.*currentTimeMillis*();

System.***out***.println((timeB-timeA)+" ms");

}

}

读取时间计算(523+562+518)/3=534.33ms

##### 直接缓冲区

**public** **class** BIOTransport {

**public** **static** **void** main(String[] args) {

**long** timeA=System.*currentTimeMillis*();

FileInputStream fileInputStream=**null**;

FileOutputStream fileOutputStream=**null**;

FileChannel inchan=**null**;

FileChannel outchan=**null**;

**try**{

fileInputStream = **new** FileInputStream(**new** File("C:/Users/pc/Desktop/[www.java1234.com]Activiti实战 PDF电子书-含书签目录.pdf"));

fileOutputStream = **new** FileOutputStream(**new** File("C:/Users/pc/Desktop/拷贝/cs.pdf"));

inchan = fileInputStream.getChannel();

outchan = fileOutputStream.getChannel();

ByteBuffer byteBuffer=ByteBuffer.*allocateDirect*(1024);

//ByteBuffer byteBuffer=ByteBuffer.allocate(1024);

**while**(inchan.read(byteBuffer)!=-1){

//开启读模式

byteBuffer.flip();

outchan.write(byteBuffer);

byteBuffer.clear();

}

}**catch**(Exception exception){

exception.printStackTrace();

}**finally**{

**try** {

**if**(fileInputStream!=**null**){

fileInputStream.close();

}

**if**(fileOutputStream!=**null**){

fileOutputStream.close();

}

**if**(inchan!=**null**){

inchan.close();

}

**if**(outchan!=**null**){

outchan.close();

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**long** timeB=System.*currentTimeMillis*();

System.***out***.println((timeB-timeA)+" ms");

}

}

时间计算(457+452+461)/3=456.333

#### 1.2.2 jkd1.7之后NIO

**public** **class** FileIOTransport {

**public** **static** **void** main(String[] args) {

FileChannel inChannel = **null**;

FileChannel outChannel = **null**;

**try**{

**long** start = System.*currentTimeMillis*();

inChannel = FileChannel.*open*(Paths.*get*("C:/Users/pc/Desktop/[www.java1234.com]Activiti实战 PDF电子书-含书签目录.pdf"), StandardOpenOption.***READ***);

outChannel = FileChannel.*open*(Paths.*get*("C:/Users/pc/Desktop/拷贝/cs.pdf"), StandardOpenOption.***WRITE***,

StandardOpenOption.***READ***, StandardOpenOption.***CREATE***);

// 内存映射文件

MappedByteBuffer inMappedByteBuf = inChannel.map(MapMode.***READ\_ONLY***, 0, inChannel.size());

MappedByteBuffer outMappedByteBuffer = outChannel.map(MapMode.***READ\_WRITE***, 0, inChannel.size());

// 直接对缓冲区进行数据的读写操作

**byte**[] dsf = **new** **byte**[inMappedByteBuf.limit()];

inMappedByteBuf.get(dsf);

outMappedByteBuffer.put(dsf);

inChannel.close();

outChannel.close();

**long** end = System.*currentTimeMillis*();

System.***out***.println((end - start)+ " ms");

}**catch**(Exception exception){

exception.printStackTrace();

}**finally**{

**try** {

**if**(inChannel!=**null**){

inChannel.close();

}

**if**(outChannel!=**null**){

outChannel.close();

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

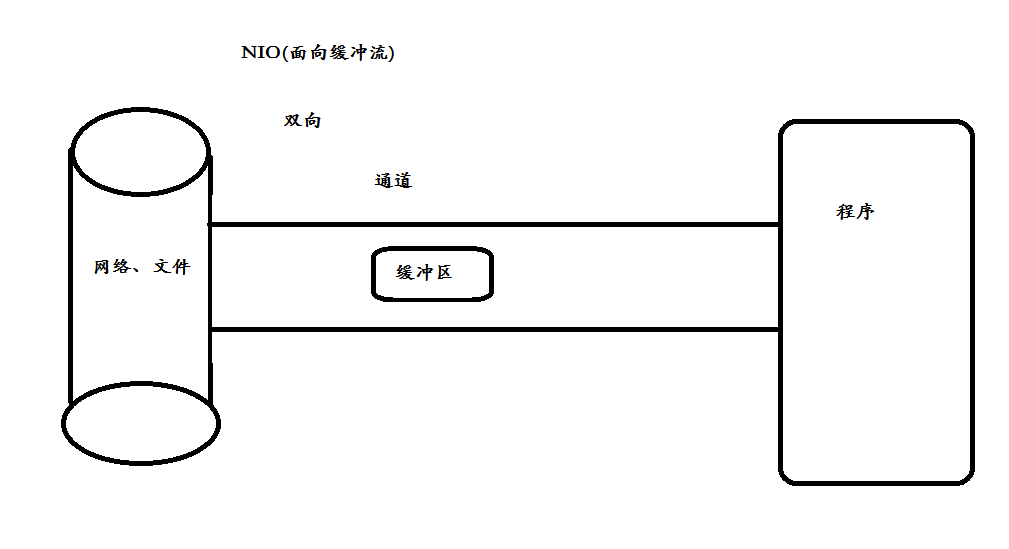
读取时间计算(144+143+145)/3=144ms

由上面的IO计算方式可以看出，选择IO流的时候应该选什么了吧？很明显JDK1.7以后文件的IO读取方式做了很大的改变。NIO的读取效率明显远远的超过了BIO,而且超出了至少3倍之多。以上是对文件的IO的总结，当然还有网络的IO的区别，以上还没有体现出NIO 非阻塞的优势。

### AIO

## 二 网络

面向缓冲区来进行数据传输的。



### 2.1 BIO

这个时候我们假设用BIO进行通信

#### 2.1.1 server

**public** **class** ServerSockets {

**static** ExecutorService *executorService*=Executors.*newCachedThreadPool*();

**private** **static** ServerSocket *serverSocket*;

**public** **static** **void** main(String[] args) **throws** IOException {

*serverSocket* = **new** ServerSocket(8080);

**while**(**true**){

Socket accept = *serverSocket*.accept();

*executorService*.execute(**new** MutiThreadServer(accept));

}

}

}

**public** **class** MutiThreadServer **implements** Runnable{

**private** Socket socket=**null**;

**public** MutiThreadServer(Socket socket){

**this**.socket=socket;

}

@Override

**public** **void** run() {

BufferedReader reader=**null**;

PrintWriter writer=**null**;

**try** {

reader=**new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));

writer=**new** PrintWriter(socket.getOutputStream(), **true**);

String inline=**null**;

**long** a=System.*currentTimeMillis*();

**while**((inline=reader.readLine())!=**null**){

writer.println(inline);

writer.flush();

}

**long** b=System.*currentTimeMillis*();

System.***out***.println(" "+(b-a)+"ms");

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

**try** {

**if**(**null**!=reader){

reader.close();

}

**if**(**null**!=writer){

writer.close();

}

} **catch** (Exception e2) {

e2.printStackTrace();

}

}

}

}

#### 2.1.2 client

**public** **class** IOSocketClient {

**private** **static** **long** *TIME*=1000\*1000\*1000;

**public** **static** **void** main(String[] args) {

ExecutorService executorService=Executors.*newCachedThreadPool*();

**for**(**int** i=0;i<10;i++){

executorService.execute(**new** Runnable() {

@Override

**public** **void** run() {

Socket client=**null**;

BufferedReader reader=**null**;

PrintWriter writer=**null**;

**try** {

client=**new** Socket();

client.connect(**new** InetSocketAddress("localhost", 8080));

OutputStream outputStream = client.getOutputStream();

writer=**new** PrintWriter(**new** OutputStreamWriter(outputStream),**true**);

LockSupport.*parkNanos*(*TIME*);

writer.print("h");

LockSupport.*parkNanos*(*TIME*);

writer.print("e");

LockSupport.*parkNanos*(*TIME*);

writer.print("l");

LockSupport.*parkNanos*(*TIME*);

writer.print("l");

LockSupport.*parkNanos*(*TIME*);

writer.print("o");

LockSupport.*parkNanos*(*TIME*);

writer.println("!");

writer.flush();

reader=**new** BufferedReader(**new** InputStreamReader(client.getInputStream()));

System.***out***.println(reader.readLine());

} **catch** (Exception e) {

} **finally**{

**try** {

client.close();

reader.close();

writer.close();

} **catch** (Exception e2) {

e2.printStackTrace();

}

}

}

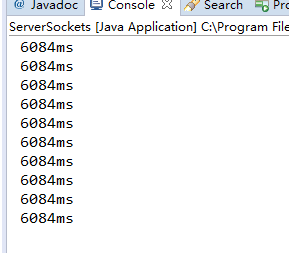
});

}

}

}

运行结果



如上图所示，服务器端十个参与通信的线程的时间，对于服务气短来说由于客户端漫长准备数据的等待造成了IO阻塞，以至于客户端要为单个客户端线程等待了近6秒的时间，这个就是BIO容易造成服务端与客户端进行线程之间的阻塞。那么有什么办法可以解决这个阻塞问题呢。这个时候NIO就是我们最好的良药了。

### 2.2 NIO

**(1)缓冲区Buffer**  
 Buffer是一个对象，它包含一些要写入或者要读出的数据，在NIO库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的；在写入数据时，写入到缓冲区中，任何时候访问NIO中的数据，都是通过缓冲区进行操作。   
 缓冲区实质上是一个数组。通常它是一个字节数组(ByteBuffer)，也可以使用其他种类的数组，但是一个缓冲区不仅仅是一个数组，缓冲区提供了对数据的结构化访问以及维护读写位置(limit)等信息。常用的有ByteBuffer，其它还有CharBuffer、ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer

**(2)通道Channel**  
 Channel是一个通道，可以通过它读取和写入数据，它就像自来水管一样，网络数据通过Channel读取和写入。通道与流的不同之处在于通道是双向的，流只是一个方向上移动(一个流必须是InputStream或者OutputStream的子类)，而且通道可以用于读、写或者用于读写。同时Channel是全双工的，因此它可以比流更好的映射底层操作系统的API。特别是在Unix网络编程中，底层操作系统的通道都是全双工的，同时支持读写操作。我们常用到的ServerSocketChannnel和SocketChannel都是SelectableChannel的子类。

**(3)多路复用器Selector**  
 多路复用器Selector是Java NIO编程的基础，多路复用器提供选择已经就绪的任务的能力，简单的说，Selector会不断的轮询注册在其上的Channel，如果某个Channel上面有新的TCP连接接入、读和写事件，这个Channel就处于就绪状态，会被Selector轮询出来，然后通过SelectionKey可以获取就绪Channel的集合，进行后续的I/O操作。  
 一个多用复用器Selector可以同时轮询多个Channel，由于JDK使用了epoll()代替传统的select实现，所以它并没有最大连接句柄1024/2048的限制，这也意味着只需要一个线程负责Selector的轮询，就可以接入成千上万的客户端。

#### 2.2.1 server

**public** **class** NIOServersx {

**public** **static** **void** main(String[] args) **throws** IOException {

System.***out***.println("服务器端已经启动....");

// 1.创建通道

ServerSocketChannel sChannel = ServerSocketChannel.*open*();

// 2.切换读取模式

sChannel.configureBlocking(**false**);

// 3.绑定连接

sChannel.bind(**new** InetSocketAddress(8080));

// 4.获取选择器

Selector selector = Selector.*open*();

// 5.将通道注册到选择器 "并且指定监听接受事件"

sChannel.register(selector, SelectionKey.***OP\_ACCEPT***);

// 6. 轮训式 获取选择 "已经准备就绪"的事件

**while** (selector.select() > 0) {

// 7.获取当前选择器所有注册的"选择键(已经就绪的监听事件)"

java.util.Iterator<SelectionKey> it = selector.selectedKeys().iterator();

**while** (it.hasNext()) {

// 8.获取准备就绪的事件

SelectionKey sk = it.next();

// 9.判断具体是什么事件准备就绪

**if** (sk.isAcceptable()) {

// 10.若"接受就绪",获取客户端连接

SocketChannel socketChannel = sChannel.accept();

// 11.设置阻塞模式

socketChannel.configureBlocking(**false**);

// 12.将该通道注册到服务器上

socketChannel.register(selector, SelectionKey.***OP\_READ***);

} **else** {

// 13.获取当前选择器"就绪" 状态的通道

SocketChannel socketChannel = (SocketChannel) sk.channel();

// 14.读取数据

ByteBuffer buf = ByteBuffer.*allocate*(1024);

**int** len = 0;

**long** timeA = System.*currentTimeMillis*();

**while** ((len = socketChannel.read(buf)) > 0) {

buf.flip();

System.***out***.print(**new** String(buf.array(), 0, len));

buf.clear();

}

**long** timeB = System.*currentTimeMillis*();

System.***out***.println((timeB-timeA));

}

it.remove();

}

}

}

}

此时我们启动NIO的服务器端。

#### 2.2.2 client

**public** **class** IOSocketClient {

**private** **static** **long** *TIME*=1000\*1000\*1000;

**public** **static** **void** main(String[] args) {

ExecutorService executorService=Executors.*newCachedThreadPool*();

**for**(**int** i=0;i<10;i++){

executorService.execute(**new** Runnable() {

@Override

**public** **void** run() {

Socket client=**null**;

BufferedReader reader=**null**;

PrintWriter writer=**null**;

**try** {

client=**new** Socket();

client.connect(**new** InetSocketAddress("localhost", 8080));

OutputStream outputStream = client.getOutputStream();

writer=**new** PrintWriter(**new** OutputStreamWriter(outputStream),**true**);

LockSupport.*parkNanos*(*TIME*);

writer.print("h");

LockSupport.*parkNanos*(*TIME*);

writer.print("e");

LockSupport.*parkNanos*(*TIME*);

writer.print("l");

LockSupport.*parkNanos*(*TIME*);

writer.print("l");

LockSupport.*parkNanos*(*TIME*);

writer.print("o");

LockSupport.*parkNanos*(*TIME*);

writer.println("!");

writer.flush();

reader=**new** BufferedReader(**new** InputStreamReader(client.getInputStream()));

System.***out***.println(reader.readLine());

} **catch** (Exception e) {

} **finally**{

**try** {

client.close();

reader.close();

writer.close();

} **catch** (Exception e2) {

e2.printStackTrace();

}

}

}

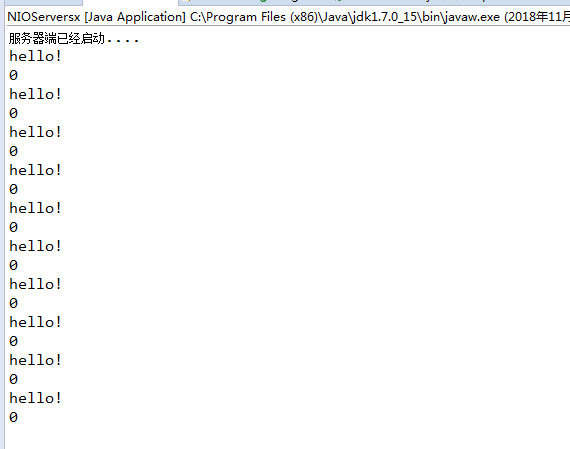
});

}

}

}

运行结果



如上图所示是nio的服务器端阻塞时间，如上图所示我们的服务器端IO并没有阻塞，也不会占用服务器端的资源，总是在客户端单个线程准备好数据以后才进行完整的一次传输，如上就是阻塞与非阻塞的区别。当然以后服务器端代码比较复杂，而且也可以进行复用。我们就此产生了nerry框架对nio进行了网络通信的封装，以方便我们的编码人员。但是此时还有一个不好的地方就是，所有的数据等待开启线程还是一个同步的过程，所以怎么实现异步非阻塞就是我们的AIO处理的了。

### 2.3 AIO