学习Java IO系统，重点是学会IO模型，了解了各种IO模型之后就可以更好的理解Java IO

Java IO 是一套Java用来读写数据（输入和输出）的API。大部分程序都要处理一些输入，并由输入产生一些输出。Java为此提供了java.io包

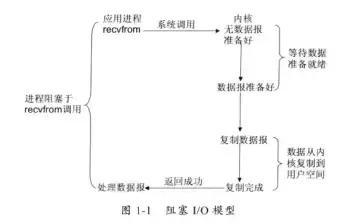
java中io系统可以分为Bio，Nio，Aio三种io模型

关于Bio，我们需要知道什么是同步阻塞IO模型，Bio操作的对象：流，以及如何使用Bio进行网络编程，使用Bio进行网络编程的问题关于Nio，我们需要知道什么是同步非阻塞IO模型，什么是多路复用Io模型，以及Nio中的Buffer,Channel,Selector的概念，以及如何使用Nio进行网络编程关于Aio，我们需要知道什么是异步非阻塞IO模型，Aio可以使用几种方式实现异步操作，以及如何使用Aio进行网络编程BIO

BIO是同步阻塞IO，JDK1.4之前只有这一个IO模型，BIO操作的对象是流，一个线程只能处理一个流的IO请求，如果想要同时处理多个流就需要使用多线程

流包括字符流和字节流，流从概念上来说是一个连续的数据流。当程序需要读数据的时候就需要使用输入流读取数据，当需要往外写数据的时候就需要输出流

阻塞IO模型



在Linux中，当应用进程调用**recvfrom**方法调用数据的时候，如果内核没有把数据准备好不会立刻返回，而是会经历等待数据准备就绪，数据从内核复制到用户空间之后再返回，这期间应用进程一直阻塞直到返回，所以被称为阻塞IO模型

流

BIO中操作的流主要有两大类，字节流和字符流，两类根据流的方向都可以分为输入流和输出流

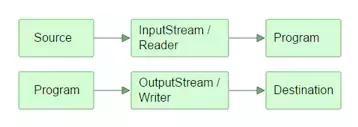
按照类型和输入输出方向可分为：

输入字节流：InputStream输出字节流：OutputStream输入字符流：Reader输出字符流：Writer字节流主要用来处理字节或二进制对象，字符流用来处理字符文本或字符串

使用InputStreamReader可以将输入字节流转化为输入字符流

Reader reader = new InputStreamReader(inputStream);使用OutputStreamWriter可以将输出字节流转化为输出字符流

Writer writer = new OutputStreamWriter(outputStream)我们可以在程序中通过InputStream和Reader从数据源中读取数据，然后也可以在程序中将数据通过OutputStream和Writer输出到目标媒介中



在使用字节流的时候，InputStream和OutputStream都是抽象类，我们实例化的都是他们的子类，每一个子类都有自己的作用范围



在使用字符流的时候也是，Reader和Writer都是抽象类，我们实例化的都是他们的子类，每一个子类都有自己的作用范围



**以读写文件为例**

从数据源中读取数据

输入字节流：**InputStream**

public static void main(String[] args) throws Exception{ File file = new File("D:/a.txt"); InputStream inputStream = new FileInputStream(file); byte[] bytes = new byte[(int) file.length()]; inputStream.read(bytes); System.out.println(new String(bytes)); inputStream.close();}输入字符流：**Reader**

public static void main(String[] args) throws Exception{ File file = new File("D:/a.txt"); Reader reader = new FileReader(file); char[] bytes = new char[(int) file.length()]; reader.read(bytes); System.out.println(new String(bytes)); reader.close();}输出到目标媒介

输出字节流：**OutputStream**

public static void main(String[] args) throws Exception{ String var = "hai this is a test"; File file = new File("D:/b.txt"); OutputStream outputStream = new FileOutputStream(file); outputStream.write(var.getBytes()); outputStream.close();}输出字符流：**Writer**

public static void main(String[] args) throws Exception{ String var = "hai this is a test"; File file = new File("D:/b.txt"); Writer writer = new FileWriter(file); writer.write(var); writer.close();}**BufferedInputStream**

在使用InputStream的时候，都是一个字节一个字节的读或写，而BufferedInputStream为输入字节流提供了缓冲区，读数据的时候会一次读取一块数据放到缓冲区里，当缓冲区里的数据被读完之后，输入流会再次填充数据缓冲区，直到输入流被读完，有了缓冲区就能够提高很多io速度

使用方式将输入流包装到BufferedInputStream中

/\*\* \* inputStream 输入流 \* 1024 内部缓冲区大小为1024byte \*/BufferedInputStream bufferedInputStream = new BufferedInputStream(inputStream,1024);**BufferedOutputStream**

BufferedOutputStream可以为输出字节流提供缓冲区，作用与BufferedInputStream类似

使用方式将输出流包装到BufferedOutputStream中

/\*\* \* outputStream 输出流 \* 1024 内部缓冲区大小为1024byte \*/BufferedOutputStream bufferedOutputStream = new BufferedOutputStream(outputStream,1024);字节流提供了带缓冲区的，那字符流肯定也提供了BufferedReader和BufferedWriter

**BufferedReader**

为输入字符流提供缓冲区，使用方式如下

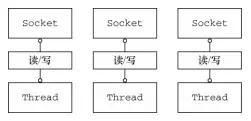
BufferedReader bufferedReader = new BufferedReader(reader,1024);**BufferedWriter**

为输出字符流提供缓冲区，使用方式如下

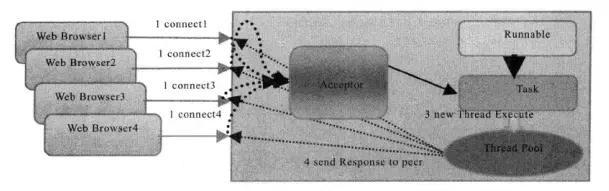
BufferedWriter bufferedWriter = new BufferedWriter(writer,1024);BIO模型 网络编程

当使用BIO模型进行Socket编程的时候，服务端通常使用while循环中调用accept方法，在没有客户端请求时，accept方法会一直阻塞，直到接收到请求并返回处理的相应，这个过程都是线性的，只有处理完当前的请求之后才会接受处理后面的请求，这样通常会导致通信线程被长时间阻塞

BIO模型处理多个连接：



在这种模式中我们通常用一个线程去接受请求，然后用一个线程池去处理请求，用这种方式并发管理多个Socket客户端连接，像这样：



使用BIO模型进行网络编程的问题在于缺乏弹性伸缩能力，客户端并发访问数量和服务器线程数量是1:1的关系，而且平时由于阻塞会有大量的线程处于等待状态，等待输入或者输出数据就绪，造成资源浪费，在面对大量并发的情况下，如果不使用线程池直接new线程的话，就会大致线程膨胀，系统性能下降，有可能导致堆栈的内存溢出，而且频繁的创建销毁线程，更浪费资源

使用线程池可能是更优一点的方案，但是无法解决阻塞IO的阻塞问题，而且还需要考虑如果线程池的数量设置较小就会拒绝大量的Socket客户端的连接，如果线程池数量设置较大的时候，会导致大量的上下文切换，而且程序要为每个线程的调用栈都分配内存，其默认值大小区间为 64 KB 到 1 MB，浪费虚拟机内存

BIO模型适用于链接数目固定而且比较少的架构，但是使用这种模型写的代码更直观简单易于理解

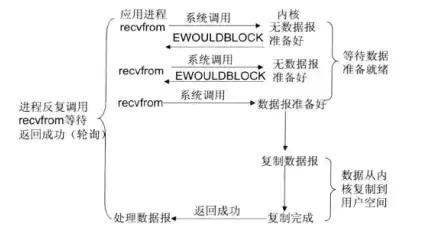
NIO

JDK 1.4版本以来，JDK发布了全新的I/O类库，简称NIO，是一种同步非阻塞IO模型

非阻塞IO模型

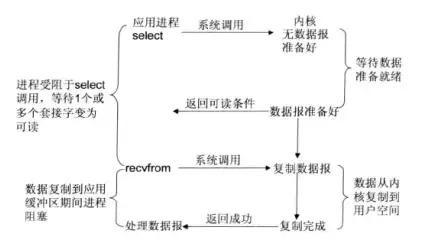
同步非阻塞IO模型实现：

**非阻塞IO模型**



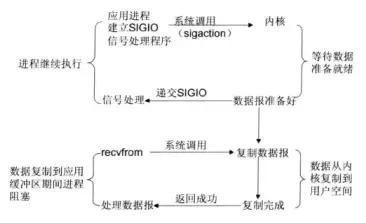
应用进程调用**recvfrom**系统调用，如果内核数据没有准备好，会直接返回一个EWOULDBLOCK错误，应用进程不会阻塞，但是需要应用进程不断的轮询调用**recvfrom**，直到内核数据准备就绪，之后等待数据从内核复制到用户空间（这段时间会阻塞，但是耗时极小），复制完成后返回

**IO复用模型**



IO复用模型，利用Linux系统提供的**select，poll**系统调用，将一个或者多个文件句柄（网络编程中的客户端链接）传递给select或者poll系统调用，应用进程阻塞在select上，这样就形成了一个进程对应多个Socket链接，然后select/poll会线性扫描这个Socket链接的集合，当只有少数socket有数据的时候，会导致效率下降，而且select/poll受限于所持有的文件句柄数量，默认值是1024个

**信号驱动 IO模型**



系统调用sigaction执行一个信号处理函数，这个系统调用不会阻塞应用进程，当数据准备就绪的时候，就为该进程生成一个SIGIO信号，通过信号回调通知应用程序调用recvfrom来读取数据

NIO的核心概念

**Buffer**（缓冲区）

Buffer是一个对象，它包含一些要写入或者读出的数据，在NIO中所有数据都是用缓存区处理的，在读数据的时候要从缓冲区中读，写数据的时候会先写到缓冲区中，缓冲区本质上是一块可以写入数据，然后可以从中读取数据的一个数组，提供了对数据的结构化访问以及在内部维护了读写位置等信息

实例化一个ByteBuffer

//创建一个容量为1024个byte的缓冲区ByteBuffer buffer=ByteBuffer.allocate(1024);如何使用Buffer：

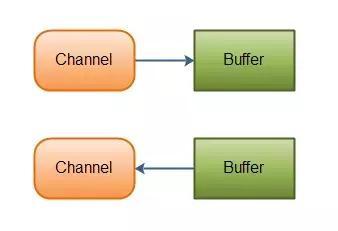
写入数据到Buffer调用flip()方法将Buffer从写模式切换到读模式从Buffer中读取数据调用clear()方法或者compact()方法清空缓冲区，让它可以再次被写入更多详细信息看这个：ifeve.com/buffers/

**Channel**（通道）

Channel（通道）数据总是从通道读取到缓冲区，或者从缓冲区写入到通道中，Channel只负责运输数据，而操作数据是Buffer

通道与流类似，不同地方:

在于条通道是双向的，可以同时进行读，写操作，而流是单向流动的，只能写入或者读取流的读写是阻塞的，通道可以异步读写



数据从Channel读到Buffer

inChannel.read(buffer);数据从Buffer写到Channel

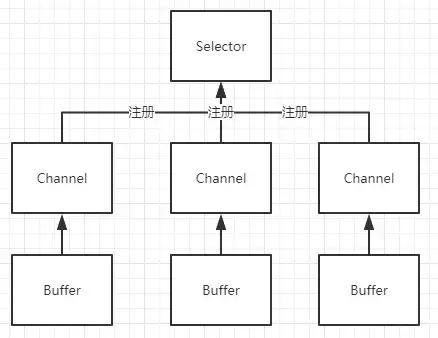
outChannel.write(buffer);更多详细信息看这个：<ifeve.com/channels/>

**以复制文件为例**

FileInputStream fileInputStream=new FileInputStream(new File(src));FileOutputStream fileOutputStream=new FileOutputStream(new File(dst));//获取输入输出channel通道FileChannel inChannel=fileInputStream.getChannel();FileChannel outChannel=fileOutputStream.getChannel();//创建容量为1024个byte的bufferByteBuffer buffer=ByteBuffer.allocate(1024);while(true){ //从inChannel里读数据，如果读不到字节了就返回-1，文件就读完了 int eof =inChannel.read(buffer); if(eof==-1){ break; } //将Buffer从写模式切换到读模式 buffer.flip(); //开始往outChannel写数据 outChannel.write(buffer); //清空buffer buffer.clear();}inChannel.close();outChannel.close();fileInputStream.close();fileOutputStream.close();**Selector**（多路复用选择器）

Selector是NIO编程的基础，主要作用就是将多个Channel注册到Selector上，如果Channel上发生读或写事件，Channel就处于就绪状态，就会被Selector轮询出来，然后通过SelectionKey就可以获取到已经就绪的Channel集合，进行IO操作了

Selector与Channel，Buffer之间的关系



更多详细信息看这个：<ifeve.com/selectors/

NIO模型 网络编程

JDK中NIO使用多路复用的IO模型，通过把多个IO阻塞复用到一个select的阻塞上，实现系统在单线程中可以同时处理多个客户端请求，节省系统开销，在JDK1.4和1.5 update10版本之前，JDK的Selector基于select/poll模型实现，在JDK 1.5 update10以上的版本，底层使用epoll代替了select/poll

epoll较select/poll的优点在于：

epoll支持打开的文件描述符数量不在受限制，select/poll可以打开的文件描述符数量有限select/poll使用轮询方式遍历整个文件描述符的集合，epoll基于每个文件描述符的callback函数回调**select，poll，epoll**都是IO多路复用的机制。I/O多路复用就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写

NIO提供了两套不同的套接字通道实现网络编程，服务端：ServerSocketChannel和客户端SocketChannel，两种通道都支持阻塞和非阻塞模式

**服务端代码**

服务端接受客户端发送的消息输出，并给客户端发送一个消息

//创建多路复用选择器Selector Selector selector=Selector.open(); //创建一个通道对象Channel，监听9001端口 ServerSocketChannel channel = ServerSocketChannel.open().bind(new InetSocketAddress(9001)); //设置channel为非阻塞 channel.configureBlocking(false); // /\*\* \* 1.SelectionKey.OP\_CONNECT：连接事件 \* 2.SelectionKey.OP\_ACCEPT：接收事件 \* 3.SelectionKey.OP\_READ：读事件 \* 4.SelectionKey.OP\_WRITE：写事件 \* \* 将channel绑定到selector上并注册OP\_ACCEPT事件 \*/ channel.register(selector,SelectionKey.OP\_ACCEPT); while (true){ //只有当OP\_ACCEPT事件到达时，selector.select()会返回（一个key），如果该事件没到达会一直阻塞 selector.select(); //当有事件到达了，select()不在阻塞，然后selector.selectedKeys()会取到已经到达事件的SelectionKey集合 Set keys = selector.selectedKeys(); Iterator iterator = keys.iterator(); while (iterator.hasNext()){ SelectionKey key = (SelectionKey) iterator.next(); //删除这个SelectionKey，防止下次select方法返回已处理过的通道 iterator.remove(); //根据SelectionKey状态判断 if (key.isConnectable()){ //连接成功 } else if (key.isAcceptable()){ /\*\* \* 接受客户端请求 \* \* 因为我们只注册了OP\_ACCEPT事件，所以有客户端链接上，只会走到这 \* 我们要做的就是去读取客户端的数据，所以我们需要根据SelectionKey获取到serverChannel \* 根据serverChannel获取到客户端Channel，然后为其再注册一个OP\_READ事件 \*/ // 1，获取到ServerSocketChannel ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel(); // 2，因为已经确定有事件到达，所以accept()方法不会阻塞 SocketChannel clientChannel = serverChannel.accept(); // 3，设置channel为非阻塞 clientChannel.configureBlocking(false); // 4，注册OP\_READ事件 clientChannel.register(key.selector(),SelectionKey.OP\_READ); } else if (key.isReadable()){ // 通道可以读数据 /\*\* \* 因为客户端连上服务器之后，注册了一个OP\_READ事件发送了一些数据 \* 所以首先还是需要先获取到clientChannel \* 然后通过Buffer读取clientChannel的数据 \*/ SocketChannel clientChannel = (SocketChannel) key.channel(); ByteBuffer byteBuffer = ByteBuffer.allocate(BUF\_SIZE); long bytesRead = clientChannel.read(byteBuffer); while (bytesRead>0){ byteBuffer.flip(); System.out.println("client data ："+new String(byteBuffer.array())); byteBuffer.clear(); bytesRead = clientChannel.read(byteBuffer); } /\*\* \* 我们服务端收到信息之后，我们再给客户端发送一个数据 \*/ byteBuffer.clear(); byteBuffer.put("客户端你好，我是服务端，你看这NIO多难".getBytes("UTF-8")); byteBuffer.flip(); clientChannel.write(byteBuffer); } else if (key.isWritable() && key.isValid()){ //通道可以写数据 } } }**客户端代码**

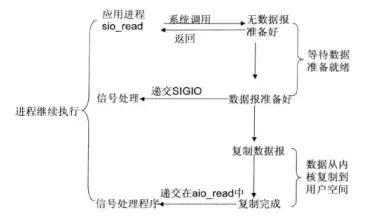
客户端连接上服务端后，先给服务端发送一个消息，并接受服务端发送的消息

Selector selector = Selector.open();SocketChannel clientChannel = SocketChannel.open();//将channel设置为非阻塞clientChannel.configureBlocking(false);//连接服务器clientChannel.connect(new InetSocketAddress(9001));//注册OP\_CONNECT事件clientChannel.register(selector, SelectionKey.OP\_CONNECT);while (true){ //如果事件没到达就一直阻塞着 selector.select(); Iterator<SelectionKey> iterator = selector.selectedKeys().iterator(); while (iterator.hasNext()){ SelectionKey key = iterator.next(); iterator.remove(); if (key.isConnectable()){ /\*\* \* 连接服务器端成功 \* \* 首先获取到clientChannel，然后通过Buffer写入数据，然后为clientChannel注册OP\_READ时间 \*/ clientChannel = (SocketChannel) key.channel(); if (clientChannel.isConnectionPending()){ clientChannel.finishConnect(); } clientChannel.configureBlocking(false); ByteBuffer byteBuffer = ByteBuffer.allocate(1024); byteBuffer.clear(); byteBuffer.put("服务端你好，我是客户端，你看这NIO难吗".getBytes("UTF-8")); byteBuffer.flip(); clientChannel.write(byteBuffer); clientChannel.register(key.selector(),SelectionKey.OP\_READ); } else if (key.isReadable()){ //通道可以读数据 clientChannel = (SocketChannel) key.channel(); ByteBuffer byteBuffer = ByteBuffer.allocate(BUF\_SIZE); long bytesRead = clientChannel.read(byteBuffer); while (bytesRead>0){ byteBuffer.flip(); System.out.println("server data ："+new String(byteBuffer.array())); byteBuffer.clear(); bytesRead = clientChannel.read(byteBuffer); } } else if (key.isWritable() && key.isValid()){ //通道可以写数据 } }}使用原生NIO类库十分复杂，NIO的类库和Api繁杂，使用麻烦，需要对网络编程十分熟悉，才能编写出高质量的NIO程序，所以并不建议直接使用原生NIO进行网络编程，而是使用一些成熟的框架，比如Netty

AIO

JDK1.7升级了Nio类库，成为Nio2.0，最主要的是提供了异步文件的IO操作，以及事件驱动IO，AIO的异步套接字通道是真正的异步非阻塞IO

异步IO模型



在Linux系统中，应用进程发起read操作，立刻可以去做其他的事，内核会将数据准备好并且复制到用空间后告诉应用进程，数据已经复制完成read操作

aio模型 网络编程

**异步操作**

aio不需要通过多路复用器对注册的通道进行轮询操作就可以实现异步读写，从而简化了NIO的编程模型

aio通过异步通道实现异步操作，异步通道提供了两种方式获取操作结果：

通过Future类来获取异步操作的结果，不过要注意的是future.get()是阻塞方法，会阻塞线程通过回调的方式进行异步，通过传入一个CompletionHandler的实现类进行回调，CompletionHandler定义了两个方法，completed和failed两方法分别对应成功和失败Aio中的Channel都支持以上两种方式

AIO提供了对应的异步套接字通道实现网络编程，服务端：AsynchronousServerSocketChannel和客户端AsynchronousSocketChannel

**服务端**

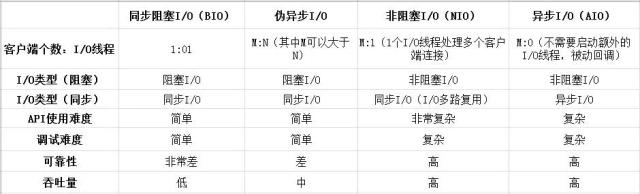
服务端向客户端发送消息，并接受客户端发送的消息

AsynchronousServerSocketChannel server = AsynchronousServerSocketChannel.open().bind(new InetSocketAddress("127.0.0.1", 9001));//异步接受请求server.accept(null, new CompletionHandler<AsynchronousSocketChannel, Void>() { //成功时 @Override public void completed(AsynchronousSocketChannel result, Void attachment) { try { ByteBuffer buffer = ByteBuffer.allocate(1024); buffer.put("我是服务端，客户端你好".getBytes()); buffer.flip(); result.write(buffer, null, new CompletionHandler<Integer, Void>(){ @Override public void completed(Integer result, Void attachment) { System.out.println("服务端发送消息成功"); } @Override public void failed(Throwable exc, Void attachment) { System.out.println("发送失败"); } }); ByteBuffer readBuffer = ByteBuffer.allocate(1024); result.read(readBuffer, null, new CompletionHandler<Integer, Void>() { //成功时调用 @Override public void completed(Integer result, Void attachment) { System.out.println(new String(readBuffer.array())); } //失败时调用 @Override public void failed(Throwable exc, Void attachment) { System.out.println("读取失败"); } }); } catch (Exception e) { e.printStackTrace(); } } //失败时 @Override public void failed(Throwable exc, Void attachment) { exc.printStackTrace(); }});//防止线程执行完TimeUnit.SECONDS.sleep(1000L);**客户端**

客户端向服务端发送消息，并接受服务端发送的消息

AsynchronousSocketChannel client = AsynchronousSocketChannel.open();Future<Void> future = client.connect(new InetSocketAddress("127.0.0.1", 9001));//阻塞，获取连接future.get();ByteBuffer buffer = ByteBuffer.allocate(1024);//读数据client.read(buffer, null, new CompletionHandler<Integer, Void>() { //成功时调用 @Override public void completed(Integer result, Void attachment) { System.out.println(new String(buffer.array())); } //失败时调用 @Override public void failed(Throwable exc, Void attachment) { System.out.println("客户端接收消息失败"); }});ByteBuffer writeBuffer = ByteBuffer.allocate(1024);writeBuffer.put("我是客户端，服务端你好".getBytes());writeBuffer.flip();//阻塞方法Future<Integer> write = client.write(writeBuffer);Integer r = write.get();if(r>0){ System.out.println("客户端消息发送成功");}//休眠线程TimeUnit.SECONDS.sleep(1000L);总结

各IO模型对比：



伪异步IO是指使用线程池处理请求的Bio模型

从Unsafe的方法入手，发现Unsafe主要有以下几个方面的功能：

* 操纵对象属性
* 操纵数组元素
* 线程挂起与恢复、CAS

分配内存并操作