新的Socket Channel可以在非阻塞模式下运行而且是可选择的，这两个功能使得大型应用程序的可 伸缩性和松耦合程度有所增加，比如Web服务器和中间件，这种做法就没有必要使得一个线程仅仅服务于某一个连接。使用NIO类，一个或者多个 线程可以同时支持多个激活的网络连接并且不会影响系统运行的性能。Java NIO处理这一部分内容主要有三个类：

　　DatagramChannel、SocketChannel、 ServerSocketChannel，它们都位于java.nio.channels.spi包下边，也就是说这些 通道类有可能在读取过程使用Selector类进行网络通道的预选择，而且这三个类中比较特殊的是ServerSocketChannel类， 因为这个类不像其他两个类一样具有read和write方法，它仅仅是为了监听连接以及创建一个新的SocketChannel对 象，它自己本身并不传输任何数据，也可以说它属于管道监听器，而不是一个标准的管道。【先区分一个概念：套接字端口 和套接字通道，一个通道是一个IO服务，提供了一系列的方法用来完成数据的读写以及传输，而区别于套接字端口的是，套接字端口是实现了协议的一些API用 来返回给Channel对象对应的响应。】非阻塞I/O（NIO）有效解决了 多线程服务器存在的线程开销问题，但在使用上略显得复杂一些。在NIO中使用多线程，主要目的已不是为了应对每个客户端请求而分配独立的服务线程，而是通 过多线程充分使用用多个CPU的处理能力和处理中的等待时间，达到提高服务能力的目的。

　　这里介绍一下三个主要的类：

public abstract class SocketChannel extends AbstractSelectableChannel implements ByteChannel,ScatteringByteChannel,GatheringByteChannel

　　针对面向流的连接套接字的可选择通道。套接字通道不是连接网络套接字的完整抽象。必须通过调用 socket 方法所获得的关联 Socket 对象来完成对套接字选项的绑定、关闭和操作。 不可能为任意的已有套接字创建通道，也不可能指定与套接字通道关联的套接字所使用的 SocketImpl 对象。通过调用此类的某个 open 方法创建套接字通道。新创建的套接字通道已打开，但尚未连接。试图在未连接的通道上调用 I/O 操作将导致抛出 NotYetConnectedException。可通过调用套接字 通道的 connect 方法连接该通道；一旦连接后，关闭套接字通道之前它会一直保持已连接状态。可通过调用套接字通道的 isConnected 方法来确定套接字通道是否已连接。套接字通道支持非阻塞连接：可创建一个套接字通道，并且通过 connect 方法可以发起到远程套接字的连接，之后通过 finishConnect 方法完成该连接。可通过调用 isConnectionPending 方法来确定是否正在进行连接操作。可单独地关闭 套接字通道的输入端和输出端，而无需实际关闭该通道。调用关联套接字对象的 shutdownInput 方法来关闭某个通道的输入端将导致该通道上的后续读取操作返回 -1（指示流的末尾）。 调用关联套接字对象的 shutdownOutput 方法来关闭通道的输出端将导致该通道上的后续写入操作抛出 ClosedChannelException。套接字通道支持异步关 闭，这与 Channel 类中所指定的异步 close 操作类似。如果一个线程关闭了某个套接字的输入端，而同时另一个线程被阻塞在该套接字通道上的读取操作中，那么处于阻塞线程中的读取操作将完成，而不读取 任何字节且返回 -1。如果一个线程关闭了某个套接字的输出端，而同时另一个线程被阻塞在该套接字通道上的写入操作中，那么阻塞线程将收到 AsynchronousCloseException。多个并发线程可安全地使用套 接字通道。尽管在任意给定时刻最多只能有一个线程进行读取和写入操作，但数据报通道支持并发的读写。connect 和 finishConnect 方法是相互同步的，如果正在调用其中某个方法的同时试图发起读取或写入操作，则在该调用完成之前该操作被阻塞。

public abstract class ServerSocketChannel extends AbstractSelectableChannel

　　针对面向流的侦听套接字的可选择通道。 服务器套接字通道不是侦听网络套接字的完整抽象。必须通过调用 socket 方法所获得的关联 ServerSocket 对象来完成对套接字选项的绑定和操作。不可能为任意的已有服务器套接字创建通道，也不可能指定与服务器套接字通道关联的服务器套接字所使用的 SocketImpl 对象。 通过调用此类的 open 方法创建服务器套接字通道。新创建的服务器套接字通道已打开，但尚未绑定。试图调用未绑定的服务器套接字通道的 accept 方法会导致抛出 NotYetBoundException。可通过调用相关服务器套接 字的某个 bind 方法来绑定服务器套接字通道。 多个并发线程可安全地使用服务器套接字通道。

public abstract class DatagramChannel extends AbstractSelectableChannel implements ByteChannel, ScatteringByteChannel, GatheringByteChannel

　　针对面向数据报套接字的可选择通道。 数据报通道不是网 络数据报套接字的完整抽象。必须通过调用 socket 所获得的关联 DatagramSocket 对象来完成套接字选项的绑定和操作。不可能为任意的已有数据报套接字创建通道，也不可能指定与数据报通道关联的数据报套接字所使用的 DatagramSocketImpl 对象。通过调用此类的 open 方法创建数据报通道。新创建的数据报通道已打开，但尚未连接。为了使用 send 和 receive 方法，无需连接数据报通道。但是如果为了避免作为每次发送和接收操作的一部分而执行的安全检查开销，也可以通过调用数据报 通道的 connect 方法来建立数据报通道连接。为了使用 read 和 write 方法，必须建立数据报通道连接，因为这些方法不接受或返回套接字地址。 一旦建立了连接，在断开数据报通道的连接或将其关闭之前，该数据报通道 保持连接状态。可通过调用数据报通道的 isConnected 方法来确定它是否已连接。多个并发线程可安全地使用数据报通 道。尽管在任意给定时刻最多只能有一个线程进行读取和写入操作，但数据报通道支持并发的读写。

　　ServerSocketChannel类：

　　该类是基于通道监听器，它和java.net.ServerSocket做 了同样的事情，但是添加了通道的语义，包括在非阻塞模式下的一些操作。ServerSocketChannel类的对象的创建使用的是静 态工厂方法open()，有可能直接和一个未绑定的ServerSocket对象直接联系，使用 ServerSocket对象的socket()方法可以返回一个ServerSocketChannel 对象。ServerSocketChannel对象并不提供bind()方法，但是提供了accept()方法，这个方法将会直接返 回SocketChannel类的对象，它适用于非阻塞模式；如果在非阻塞模式下调用 accept()方法，如果没有任何连接那么accept()方法会立即返回null。

　　——[$]提供一个非阻塞模式的accept()方法例子——

package org.susan.java.io;

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.channels.ServerSocketChannel;

import java.nio.channels.SocketChannel;

public class ChannelAccept {

public static final String GREETING = "Hello I must be going.\n";

public static void main(String[] argv) throws Exception {

int port = 1234;

if (argv.length > 0) {

port = Integer.parseInt(argv[0]);

}

ByteBuffer buffer = ByteBuffer.wrap(GREETING.getBytes());

ServerSocketChannel ssc = ServerSocketChannel.open();

ssc.socket().bind(new InetSocketAddress(port));

ssc.configureBlocking(false);

while (true) {

System.out.println("Waiting for connections");

SocketChannel sc = ssc.accept();

if (sc == null) {

Thread.sleep(2000);

} else {

System.out.println("Incoming connection from: " + sc.socket().getRemoteSocketAddress());

buffer.rewind();

sc.write(buffer);

sc.close();

}

}

}

}

　　这段代码的输出为：

Waiting for connections

Incoming connection from: /127.0.0.1:59377

Waiting for connections

　　注意中间一行，中间一行代码是用浏览器直接输入：http://localhost:1234的 时候控制台打印出来的，而网页显示内容就是GREETING字符串的内容。

　　SocketChannel类：

　　该类扮演了Socket通道中的客户端，可以初始化一个连接到请求监听服务器。而每一个SocketChannel 对象都是使用java.net.Socket对象创建的， 静态方法open()创建了一个新的SocketChannel对象，而SocketChannel对象调用socket()方法可以返回一个 Socket对象，反之调用Socket的getChannel()方法可以返回原来的SocketChannel对象，当Socket对象直接创建并且和 SocketChannel对象没有任何联系的时候，它的getChannel()方法会直接返回null。当 该连接通道建立好了过后，可以使用connect()方法完成真正的连接过程，一旦某个套接字通道连接创建成功过后，它就会一直保持连接状态直到关闭为 止。

　　使用SocketChannel的关闭连接的时候使用的是finishConnect()方 法，在非阻塞模式调用了这个方法过后：

\* connect() 方法如果没有调用的时候，直接抛出异常：NoConnectionException

\* 当 连接打开了但是还没有完成的时候，什么都不会发生，finishConnect()方法会直接返回false

\* SocketChannel 类的对象调用了connect()方法过后可以直接从阻塞模式切换到非阻塞模式，如果有必要的调用线程会产生 阻塞直到连接完成，这时finishConnect()方法返回true

\* 当连接启用从connect()直到finishConnect()，SocketChannel 对象的内部状态已经更新到已连接，这种时候finishConnect()方法也会返回true，而且SocketChannel对象接着继续进行数据传 送

\* 当连接启用的时候，什么事情都不发生，该方法直接返回true

　　——[$]使用SocketChannel在非阻塞模式下的异步调用——

package org.susan.java.io;

import java.net.InetSocketAddress;

import java.nio.channels.SocketChannel;

public class ConnectAsync {

public static void main(String[] argv) throws Exception {

String host = "localhost";

int port = 80;

if (argv.length == 2) {

host = argv[0];

port = Integer.parseInt(argv[1]);

}

InetSocketAddress addr = new InetSocketAddress(host, port);

SocketChannel sc = SocketChannel.open();

sc.configureBlocking(false);

System.out.println("initiating connection");

sc.connect(addr);

while (!sc.finishConnect()) {

doSomethingUseful();

}

System.out.println("connection established");

sc.close();

}

private static void doSomethingUseful() {

System.out.println("doing something useless");

}

}

　　该段代码输出为：

initiating connection

connection established

　　DatagramChannel类：

　　和SocketChannel不一样的是：

SocketChannel操作是基于TCP/IP协议， 而DatagramChannel是基于UDP/IP协议，所以这里仅仅讲解DatagramChannel区别于 SocketChannel的方法，其他方法内容都大同小异，该类这里就不提供例子了。

　　主要区分的方法：

public abstract int send(ByteBuffer src,SocketAddress target)：

　　通过此通道发送数据报。如果此通道处于非阻塞模式并且 底层输出缓冲区中没有足够的空间，或者如果此通道处于阻塞模式并且缓冲区中有足够的空间，则将给定缓冲区中的剩余字节以单个数据报的形式传送到给定的目标 地址。从字节缓冲区传输数据报如同通过正规的 write 操作一样。此方法执行的安全检查与 DatagramSocket 类的 send 方法执行的安全检查完全相同。也就是说，如果该套接字未连接到指定的远程地址，并且已安装了安全管理器， 则对于每个发送的数据报，此方法都会验证安全管理器的 checkConnect 方法是否允许使用该数据报的目标地址和端口号。 避免此项安全检查开销的方法是首先通过 connect 方法连接该套接字。可在任意时间调用此方法。但是如果另一个线程已经在此通道上发起了一个写入操作，则 在该操作完成前此方法的调用被阻塞。

　　使用原因：

\* 当应用程序需能够承受数据丢失的时候

\* 应用程序想通知某些客户端而且不需要知道对方是否收到数据包的时候

\* 吞吐率比稳定性重要的时候

\* 需要进行多播或者广播方式发送数据出去

\* 基 于包比基于流好的时候【抱歉，这点我不是很理解，所以留下英文原文：The packet metaphor fits the task at hand better than the stream metaphor.】

　　这里提供两个例子，一个是客户端、一个是服务端：

　　——[$]服务端代码——

package org.susan.java.io;

import java.net.InetSocketAddress;

import java.net.SocketAddress;

import java.net.SocketException;

import java.nio.ByteBuffer;

import java.nio.ByteOrder;

import java.nio.channels.DatagramChannel;

public class TimeServer {

private static final int DEFAULT\_TIME\_PORT = 37;

private static final long DIFF\_1900 = 2208988800L;

protected DatagramChannel channel;

public TimeServer(int port) throws Exception {

this.channel = DatagramChannel.open();

this.channel.socket().bind(new InetSocketAddress(port));

System.out.println("Listening on port " + port + " for time requests");

}

public void listen() throws Exception {

ByteBuffer longBuffer = ByteBuffer.allocate(8);

longBuffer.order(ByteOrder.BIG\_ENDIAN);

longBuffer.putLong(0, 0);

longBuffer.position(4);

ByteBuffer buffer = longBuffer.slice();

while (true) {

buffer.clear();

SocketAddress sa = this.channel.receive(buffer);

if (sa == null) {

continue;

}

System.out.println("Time request from " + sa);

buffer.clear();

longBuffer.putLong(0, (System.currentTimeMillis() / 1000) + DIFF\_1900);

this.channel.send(buffer, sa);

}

}

public static void main(String[] argv) throws Exception {

int port = DEFAULT\_TIME\_PORT;

if (argv.length > 0) {

port = Integer.parseInt(argv[0]);

}

try {

TimeServer server = new TimeServer(port);

server.listen();

} catch (SocketException e) {

System.out.println("Can't bind to port " + port + ", try a different one");

}

}

}

　　——[$]客户端代码——

package org.susan.java.io;

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.ByteOrder;

import java.nio.channels.DatagramChannel;

import java.util.Date;

import java.util.Iterator;

import java.util.LinkedList;

import java.util.List;

public class TimeClient {

private static final int DEFAULT\_TIME\_PORT = 37;

private static final long DIFF\_1900 = 2208988800L;

protected int port = DEFAULT\_TIME\_PORT;

protected List remoteHosts;

protected DatagramChannel channel;

public TimeClient(String[] argv) throws Exception {

if (argv.length == 0) {

throw new Exception("Usage: [ -p port ] host ...");

}

parseArgs(argv);

this.channel = DatagramChannel.open();

}

protected InetSocketAddress receivePacket(DatagramChannel channel,

ByteBuffer buffer) throws Exception {

buffer.clear();

return ((InetSocketAddress) channel.receive(buffer));

}

protected void sendRequests() throws Exception {

ByteBuffer buffer = ByteBuffer.allocate(1);

Iterator it = remoteHosts.iterator();

while (it.hasNext()) {

InetSocketAddress sa = (InetSocketAddress) it.next();

System.out.println("Requesting time from " + sa.getHostName() + ":" + sa.getPort());

buffer.clear().flip();

channel.send(buffer, sa);

}

}

public void getReplies() throws Exception {

ByteBuffer longBuffer = ByteBuffer.allocate(8);

longBuffer.order(ByteOrder.BIG\_ENDIAN);

longBuffer.putLong(0, 0);

longBuffer.position(4);

ByteBuffer buffer = longBuffer.slice();

int expect = remoteHosts.size();

int replies = 0;

System.out.println("");

System.out.println("Waiting for replies...");

while (true) {

InetSocketAddress sa;

sa = receivePacket(channel, buffer);

buffer.flip();

replies++;

printTime(longBuffer.getLong(0), sa);

if (replies == expect) {

System.out.println("All packets answered");

break;

}

System.out.println("Received " + replies + " of " + expect + " replies");

}

}

protected void printTime(long remote1900, InetSocketAddress sa) {

long local = System.currentTimeMillis() / 1000;

long remote = remote1900 - DIFF\_1900;

Date remoteDate = new Date(remote \* 1000);

Date localDate = new Date(local \* 1000);

long skew = remote - local;

System.out.println("Reply from " + sa.getHostName() + ":" + sa.getPort());

System.out.println(" there: " + remoteDate);

System.out.println(" here: " + localDate);

System.out.print(" skew: ");

if (skew == 0) {

System.out.println("none");

} else if (skew > 0) {

System.out.println(skew + " seconds ahead");

} else {

System.out.println((-skew) + " seconds behind");

}

}

protected void parseArgs(String[] argv) {

remoteHosts = new LinkedList();

for (int i = 0; i < argv.length; i++) {

String arg = argv[i];

if (arg.equals("-p")) {

i++;

this.port = Integer.parseInt(argv[i]);

continue;

}

InetSocketAddress sa = new InetSocketAddress(arg, port);

if (sa.getAddress() == null) {

System.out.println("Cannot resolve address: " + arg);

continue;

}

remoteHosts.add(sa);

}

}

public static void main(String[] argv) throws Exception {

TimeClient client = new TimeClient(argv);

client.sendRequests();

client.getReplies();

}

}

　　上边代码就不详细讲解了，运行的时候需要先运行服务端，然后再运行客户端，客户端记得在参数列 表中添加-p port来运行。还有一类通道 读写这里不做介绍，就是基于Pipe的，因为这方面的类我很少使用，所以怕总结出来的和翻译出来的内容会误导读者，如果有兴趣的读者了解这部分内容可以来 Email补充该章节内容。

　　【\*：和Socket Channel有关的在Java NIO里面还有一个关键的概念是Selector，这个概念在最后讲解，通过上边部分的讲解，已经基本将Java NIO里面的内容详细解说了一边，可能有些地方有遗漏而且总结得比较粗略，所以本章节所有标题上都加上了草案，希望读者理解。】

　　到此针对通道做一个简单的小节：

\* 通道读取支持分散集中的机制进行高效读写数据

\* 文件通道是常用的方式，主要理解文件锁、和操作系统本地代码的相互关系、以及文件的访问控制

\* 内存映射文件的理解需深入

\* 基于网络套接字的三个Channel类的理解