服务器在合理的时间之内处理大量客户机请求的能力**取决于服务器使用 I/O 流的效率**。同时为成百上千个客户机提供服务的服务器必须能够并发地使用 I/O 服务。Java 平台直到 JDK 1.4（也就是 Merlin）才支持非阻塞 I/O 调用。用 Java 语言写的服务器，由于其线程与客户机之比几乎是一比一，因而易于受到大量线程开销的影响，其结果是既导致了性能问题又缺乏可伸缩性。

为了解决这个问题，Java 平台的最新发行版引入了一组新的类。Merlin 的 java.nio 包充满了解决线程开销问题的技巧，包中最重要的是新的 **SelectableChannel 类和 Selector 类**。 通道（channel）是客户机和服务器之间的一种通信方式。 选择器（selector）与 Windows 消息循环类似，它从不同客户机捕获各种事件并将它们分派到相应的事件处理程序。在本文，我们将向您展示这两个类如何协同工作，从而为 Java 平台创建非阻塞 I/O 机制。

Merlin 之前的 I/O 编程

我们将从考察基础的、Merlin 之前的服务器-套接字（server-socket）程序开始。在 ServerSocket 类的生存期中，其重要功能如下：

\* 接受传入连接

\* 从客户机读取请求

\* 为请求提供服务

我们来考察一下以上每一个步骤，我们用代码片段来说明。 首先，我们创建一个新的 ServerSocket ：

ServerSocket s = new ServerSocket();

接着，我们要接受传入调用。这里，调用 accept() 应该可以完成任务，但其中有个小陷阱您得当心：

Socket conn = s.accept( );

**对 accept() 的调用将一直阻塞，直到服务器套接字接受了一个请求连接的客户机请求。**一旦建立了连接，服务器就使用 LineNumberReader 读取客户机请求。因为 LineNumberReader **要到缓冲区满时才成批地读取数据**，所以这个调用在读时阻塞。 下面的片段显示了工作中的 LineNumberReader （阻塞等等）。

InputStream in = conn.getInputStream();

InputStreamReader rdr = new InputStreamReader(in);

LineNumberReader lnr = new LineNumberReader(rdr);

Request req = new Request();

while (!req.isComplete() )

{

String s = lnr.readLine();

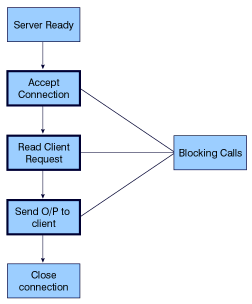
req.addLine(s);

}

InputStream.read() 是另一种读取数据的方式。不幸的是， **read 方法也要一直阻塞到数据可用为止， write 方法也一样。**

图 1 描绘了服务器的典型工作过程。黑体线表示处于阻塞的操作。

图 1. 典型的工作中的服务器



在 JDK 1.4 之前，自由地使用线程是处理阻塞问题最典型的办法。但这个解决办法会产生它自己的问题 ― 即**线程开销**，线程开销同时影响性能和可伸缩性。不过，随着 Merlin 和 java.nio 包的到来，一切都变了。

在下面的几个部分中，我们将考察 java.nio 的基本思想，然后把我们所学到的一些知识应用于修改前面描述的服务器-套接字示例。

**反应器模式（Reactor pattern）**

**NIO 设计背后的基石是反应器设计模式。** 分布式系统中的服务器应用程序必须处理多个向它们发送服务请求的客户机。然而，**在调用特定的服务之前，服务器应用程序必须将每个传入请求多路分用并分派到各自相应的服务提供者。**反应器模式正好适用于这一功能。它允许事件驱动应用程序将服务请求**多路分用并进行分派**，然后，这些服务请求被并发地从一个或多个客户机传送到应用程序。

反应器模式的核心功能

\* 将事件多路分用

\* 将事件分派到各自相应的事件处理程序

反应器模式与观察者模式（Observer pattern）在这个方面极为相似：当一个主体发生改变时，所有依属体都得到通知。不过，观察者模式与单个事件源关联，而反应器模式则与多个事件源关联。

通道和选择器

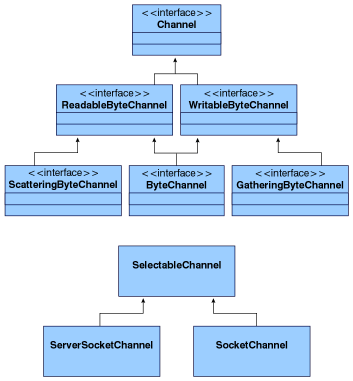
NIO 的非阻塞 I/O 机制是围绕 **选择器** 和 **通道**构建的。 Channel 类表示服务器和客户机之间的一种通信机制。与反应器模式一致， Selector 类是 Channel 的多路复用器。 Selector 类将传入客户机请求多路分用并将它们分派到各自的请求处理程序。

我们将仔细考察 Channel 类和 Selector 类的各个功能，以及这两个类如何协同工作，创建非阻塞 I/O 实现。

通道做什么

通道表示连到一个实体（例如：硬件设备、文件、网络套接字或者能执行一个或多个不同 I/O 操作（例如：读或写）的程序组件）的开放连接。可以异步地关闭和中断 NIO 通道。所以，如果一个线程在某条通道的 I/O 操作上阻塞时，那么另一个线程可以将这条通道关闭。类似地，如果一个线程在某条通道的 I/O 操作上阻塞时，那么另一个线程可以中断这个阻塞线程。

图 2. java.nio.channels 的类层次结构



如图 2 所示，在 java.nio.channels 包中有不少通道接口。我们主要关心 java.nio.channels.SocketChannel 接口和 java.nio.channels.ServerSocketChannel 接口。 这两个接口可用来分别代替 java.net.Socket 和 java.net.ServerSocket 。尽管我们当然将把注意力放在以非阻塞方式使用通道上，但通道可以以阻塞方式或非阻塞方式使用。

创建一条非阻塞通道

为了实现基础的非阻塞套接字读和写操作，我们要处理两个新类。它们是来自 java.net 包的 **InetSocketAddress 类，它指定连接到哪里，以及来自 java.nio.channels 包的 SocketChannel 类，它执行实际的读和写操作。**

这部分中的代码片段显示了一种经过修改的、非阻塞的办法来创建基础的服务器-套接字程序。请注意这些代码样本与第一个示例中所用的代码之间的变化，从添加两个新类开始：

String host = ......;

InetSocketAddress socketAddress = new InetSocketAddress(host, 80);

SocketChannel channel = SocketChannel.open();

channel.connect(socketAddress);

缓冲区的角色

Buffer 是包含特定基本数据类型数据的抽象类。从本质上说，它是一个包装器，它将带有 getter/setter 方法的固定大小的数组包装起来，这些 getter/setter 方法使得缓冲区的内容可以被访问。 Buffer 类有许多子类，如下：

\* ByteBuffer

\* CharBuffer

\* DoubleBuffer

\* FloatBuffer

\* IntBuffer

\* LongBuffer

\* ShortBuffer

**ByteBuffer 是唯一支持对其它类型进行读写的类**，因为其它类都是特定于类型的。一旦连接上，就可以使用 ByteBuffer 对象从通道读数据或将数据写到通道。

为了使通道成为非阻塞的，我们在通道上调用 configureBlockingMethod(false) ，如下所示：

channel.configureBlockingMethod(false);

在阻塞模式中，线程将在读或写时阻塞，一直到读或写操作彻底完成。如果在读的时候，数据尚未完全到达套接字，则线程将在读操作上阻塞，一直到数据可用。

**在非阻塞模式中，线程将读取已经可用的数据（不论多少），然后返回执行其它任务。**如果将真（true）传递给 configureBlockingMethod() ，则通道的行为将与在 Socket 上进行阻塞读或写时的行为完全相同。唯一的主要差别，如上所述，是这些阻塞读和写可以被其它线程中断。

**单靠 Channel 创建非阻塞 I/O 实现是不够的。要实现非阻塞 I/O， Channel 类必须与 Selector 类配合进行工作。**

选择器做什么

在反应器模式情形中， Selector 类充当 Reactor 角色。 Selector 对多个 SelectableChannels 的事件进行多路复用。每个 Channel 向 Selector 注册事件。当事件从客户机处到来时， Selector 将它们多路分用并将这些事件分派到相应的 Channel 。

创建 Selector 最简单的办法是使用 open() 方法，如下所示：

Selector selector = Selector.open();

通道遇上选择器

每个要为客户机请求提供服务的 Channel 都必须首先创建一个连接。下面的代码创建称为 Server 的 ServerSocketChannel 并将它绑定到本地端口：

ServerSocketChannel serverChannel = ServerSocketChannel.open();

serverChannel.configureBlocking(false);

InetAddress ia = InetAddress.getLocalHost();

InetSocketAddress isa = new InetSocketAddress(ia, port );

serverChannel.socket().bind(isa);

每个要为客户机请求提供服务的 Channel 都必须接着将自己向 Selector 注册。 Channel 应根据它将处理的事件进行注册。例如，接受传入连接的 Channel 应这样注册，如下：

SelectionKey acceptKey = channel.register( selector,SelectionKey.OP\_ACCEPT);

**Channel 向 Selector 的注册用 SelectionKey 对象表示**。满足以下三个条件之一， Key 就失效：

**\* Channel 被关闭。**

**\* Selector 被关闭。**

**\* 通过调用 Key 的 cancel() 方法将 Key 本身取消。**

Selector 在 select() 调用时阻塞。接着，它开始等待，直到建立了一个新的连接，或者另一个线程将它唤醒，或者另一个线程将原来的阻塞线程中断。

注册服务器

Server 是那个将自己向 Selector 注册以接受所有传入连接的 ServerSocketChannel ，如下所示：

SelectionKey acceptKey = serverChannel.register(sel, SelectionKey.OP\_ACCEPT);

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

......

Server 被注册后，我们根据每个关键字（key）的类型以迭代方式对一组关键字进行处理。一个关键字被处理完成后，就都被从就绪关键字（ready keys）列表中除去，如下所示：

Set readyKeys = sel.selectedKeys();

Iterator it = readyKeys.iterator();

while (it.hasNext())

{

SelectionKey key = (SelectionKey)it.next();

it.remove();

....

....

....

}

如果关键字是可接受（acceptable）的，则接受连接，注册通道，以接受更多的事件（例如：读或写操作）。 如果关键字是可读的（readable）或可写的（writable），则服务器会指示它已经就绪于读写本端数据：

SocketChannel socket;

if (key.isAcceptable()) {

System.out.println("Acceptable Key");

ServerSocketChannel ssc = (ServerSocketChannel) key.channel();

socket = (SocketChannel) ssc.accept();

socket.configureBlocking(false);

SelectionKey another =

socket.register(sel,SelectionKey.OP\_READ|SelectionKey.OP\_WRITE);

}

if (key.isReadable()) {

System.out.println("Readable Key");

String ret = readMessage(key);

if (ret.length() > 0) {

writeMessage(socket,ret);

}

}

if (key.isWritable()) {

System.out.println("Writable Key");

String ret = readMessage(key);

socket = (SocketChannel)key.channel();

if (result.length() > 0 ) {

writeMessage(socket,ret);

}

}

对 JDK 1.4 中的非阻塞 I/O 的介绍的最后一部分留给您：运行这个示例。

在这个简单的非阻塞服务器-套接字示例中，服务器读取发送自客户机的文件名，显示该文件的内容，然后将内容写回到客户机。

这里是您运行这个示例需要做的事情：

1. 安装 JDK 1.4（请参阅 参考资料）。

2. 将两个 源代码文件复制到您的目录。

3. 编译和运行服务器， java NonBlockingServer 。

4. 编译和运行客户机， java Client 。

5. 输入类文件所在目录的一个文本文件或 java 文件的名称。

6. 服务器将读取该文件并将其内容发送到客户机。

7. 客户机将把从服务器接收到的数据打印出来。（由于所用的 ByteBuffer 的限制，所以将只读取 1024 字节。）

8. 输入 quit 或 shutdown 命令关闭客户机。

Merlin 的新 I/O 包覆盖范围很广。Merlin 的新的**非阻塞 I/O 实现的主要优点有两方面：线程不再在读或写时阻塞，以及 Selector 能够处理多个连接，从而大幅降低了服务器应用程序开销。**

我们已经着重论述了新的 java.nio 包的这两大优点。我们希望，您将把在这里所学到的知识应用到自己的实际应用程序开发工作中。