

互联网诞生在1950年代和60年代。一些年以后,大约在80年代,套接字的概念在BSD中介绍(加州大学伯克利分校软件-一个Unix变体)使用互联网协议(IP)用来在两个进程中通信。一些年后,在1996年,JDK 1.0为编程世界带来套接字的概念作为一个网络通信模型,它容易使用和跨平台。最后,编程者可以创建网络应用程序而不用多年的学习关于网络通信的知识。Java开发者可用写出一个简单的网络应用程序只需抓几门主题的表面(FIXME: *scratching the surface of a few subjects*),比如IP,IP地址,端口和Java网络。

IP将所有的通信分成数据包(数据块),从源地址到目的地址是单独处理的-没有传递的保证。在IP之上,我们有一些通用的协议,比如TCP(传输控制协议)和UDP(用户数据报协议)(这一章的应用程序使用这些协议),在这些之上,我们还有更多的协议,包括HTTP,TELNET,DNS,等等。套接字使用IP用来在机器之间通信,所以Java网络应用程序可以使用预定义的协议与存在的服务器"交流"。

在互联网上,每台机器可以通过一个数字标签标识,称为IP地址。每个Java开发者应该知道我们处理两种类型的IP地址:IPv4(用32位代表,比如124.32.45.23)和IPv6(用128位代表-比如2607:f0d0:1002:0051:0000:0000:0000:0004)。此外,知道IP地址由A,B,C,D和E类地址组成是重要的。因此,我们对D类IP地址由一个特殊的兴趣,让我们说,IPv4地址在224.0.0.1和239.255.255.255之间变化,意味着是组播组。另外,记住地址127.0.0.1是本地地址的保留的。

关注于端口,TCP/UDP端口范围在0和65535,它们在Java中用整型表示。某些服务类型通常在某些端口上建立:比如,如果你连接到一个主机的80端口,你可能是期望找到一个HTTP服务器。在端口21上,你可能是期望一个FTP服务器,在端口23上是一个Telnet服务器,在端口119上是一个NNTP服务器,等等。因此,当选择端口是时候要注意;要保证你不会干涉其它的进程并保持在范围内。

这些概念每个都有一本专门的书籍,但是本书对于创建Java客户端/服务器端应用程序来说有足够的信息。在一个客户端/服务器模型中,一个服务器运行在一台主机上,监听端口用来通过网络从客户端接收连接请求,甚至从同一台机器上。客户端使用IP地址(主机名)和端口去定位服务器,服务器根据它的请求服务每个客户端。在连接过程中,客户端通过一个本地端口数标识自身给服务器,可以明确地设置或者通过内核分配-一个套接字绑定到这个本地端口在这个连接中被使用(我们说客户端绑定到一个本地端口)。接收后,服务器获得一个绑定到新的本地端口的套接字,已设置远程端点的地址和端口的客户端-它需要一个新的端口这样它可以在原来的端口上继续监听连接请求。一旦通信固定,数据可以在套接字间来往,直到通信被故意关闭或者意外中断。

我们可以推断(conclude),对于Java,一个套接字是一个双向的软件端口在一个服务器程序和它的客户端程序之间,或者更通常的,在两个运行在网络上的程序,涉及双向通信。一个端口是一个IP地址和端口的组合。

Java在JDK 1.0中为套接字引入了支持,但是随着时间过去事情当然有所改变从版本到版本。跳到Java 7,NIO.2已经通过升级已存在的类为写基于TCP/UDP应用程序使用新的方法和加入新的接口和类改进了这个支持。首先,NIO.2引入了名为NetworkChannel的接口,提供方法为所有的网络通道类共有-任何实现这个接口的通道是一个网络套接字通道。专门用于异步套接字的通道的类,ServerSocketChannel,SocketChannel和DatagramChannel,实现了这个接口,带有用来绑定和返回本地地址的方法,通过新的SocketOption<T>接口和StandardSocketOptions类设置和获取套接字选项。这个接口的方法直接加入到了类中(为了检测连接状态,获取远程地址和关闭)将不再需要你调用socket()方法。

NIO.2也引入了MulticastChannel接口作为NetworkChannel的子接口。顾名思义,MulticastChannel接口对应一个网络通道支持IP组播。记住MulticastChannel只有被数据报通道实现(DatagramChannel类)。当加入一个组播组,你将得到一个成员资格(membership)key,代表了一个组播组的成员资格。通过这个成员资格key,你可以从不同的地址限制/不限制数据报,放弃成员资格,获得通道和/或组播组,这个成员资格key被创建。

---

注意:为了简要浏览下Java通道,请看下第7章的"通道的简要概览"小节,"ByteBuffer的简要概览"小节-可以考虑看下理解Java缓冲区如何工作的。

---

## NetworkChannel概览

在这节,我们将简要概览一下NetworkChannel方法。这个接口代表了一个到网络套接字的通道,

有一组套接字的5个通用的方法.我们在这里提出它们因为它们非常有用在下面的章节中.

我们从`bind()`方法开始,这个方法绑定通道的套接字到一个本地地址.更加精确(*precisely*)地说,这个方法将在套接字和一个本地地址之间建立关系,通常明确指定一个`InetSocketAddress`实例(这个类代表了一个套接字地址有IP(或者主机名)和端口),并且继承了抽象的`SocketAddress`类.本地地址也可以被自动分配如果我们传递`null`到`bind()`方法.这个方法用来在本地机器绑定到服务器套接字通道,套接字通道和数据报套接字通道.它将返回当前的通道:

```
NetworkChannel bind(SocketAddress local) throws IOException
```

`NetworkChannel`可以通道调用`getLocalAddress()`方法来取得绑定的地址.如果通道的套接字没有绑定,它将返回`null`.

```
SocketAddress getLocalAddress() throws IOException
```

## 套接字选项

剩下的`NetworkChannel`是三个方法处理当前通道支持的套接字选项.一个套接字选项与一个套接字关联通过`SocketOption<T>`接口代表.目前,`NIO.2`在`StandardSocketOptions`类中使用一组标准的选项实现了这个接口.这个就是它们:

- `IP_MULTICAST_IF`: 这个选项用来指定网络接口(`NetworkChannel`)被组播数据报使用,通道面向数据报的套接字发送;如果为`null`,操作系统将选择一个输出接口(如果有1个可用).默认情况下,它为`null`,但是选项的值可以在套接字绑定后设置.当我们讨论发送数据报,你将看到如何在你的机器上找到什么样的组播接口可用.
- `IP_MULTICAST_LOOP`: 这个选项的值是一个`boolean`,控制组播数据报的回路地址(这是操作系统依赖的).作为应用程序的作者,你需要去决定你是否想让你发送的数据回送到你的主机.
- `IP_MULTICAST_TTL`: 这个选项的值是一个在0到255之间的整数,它代表了通过面向数据报套接字组播数据报发送的生存时间.如果不是另外规定,组播数据报使用默认的值1发送,为了防止它们在本机网络之上被转发.使用这个选项我们可以控制组播数据报的范围.默认情况下,这个设置为1,但是这个选项的值可以在套接字绑定之后被设置.
- `IP_TOS`: 这个选项的值是一个整数代表了在IP数据包中通道套接字发送的服务类型(ToS)的字节的值-这个值的解释是特定于网络的.当前这个只有对IPv4可用,默认下它的值通常是0.这个选项的值可以在套接字绑定后的任意时间设置.
- `SO_BROADCAST`: 这个选项的值是一个`boolean`,表明广播数据报的传输是否被允许(特定于面向数据报的套接字发送到IPv4广播地址).默认情况下,它为`FALSE`,但是这个选项的值可以在任意时间被设置.
- `SO_KEEPAIVE`: 这个选项的值是一个`boolean`,表明是否连接应该保持存活.默认下,它设置为`FALSE`,但是这个选项的值可以在任意时间被设置.
- `SO_LINGER`: 这个选项的值是一个整数用秒代表了超时(停留的时间间隔).当通过`close()`方法尝试去关闭一个阻塞模式的套接字,它将在传输未发送的数据之前的等待的停留间隔的周期.默认下,它是一个负值,意味着这些选项不可用.这个选项的值可在任意时刻设置,并且最大值是操作系统依赖的.
- `SO_RCVBUF`: 这个选项是一个整数值,代表了在套接字接收缓冲区中的字节的数目-输入缓冲区由网络实现使用.默认下,这个值是操作系统依赖的,但是它可以在套接字绑定或者连接之前设置.依赖于操作系统,这个值可以在套接字绑定之后改变.负值是不被允许的.
- `SO_SNDBUF`: 这个选项是一个整数值,代表了在套接字发送缓冲区中的字节的数目-输出缓冲区由网络实现使用.默认下,这个值是操作系统依赖的,但是它可以在套接字绑定或者连接之前设置.依赖于操作系统,这个值可以在套接字绑定后改变.负值是不被允许的.
- `SO_REUSEADDR`: 这个选项的是一个`boolean`值表明地址是否可以被重用(书中写的`int`值,是错误的).这个在数据报组播中非常有用当我们想让多个程序绑定到相同的地址.在面向流的套接字中套接字可以被绑定到一个地址当一个之前的连接处于`TIME_WAIT`状态-`TIME_WAIT`意味着操作系统已经接收到一个关闭套接字的请求,但是可能需要从客户端等待后期的通信.默认下,这个选项的值是系统依赖的,但是它可以在套接字绑定或者连接前设置.

- TCP\_NODELAY: 这个选项的值是一个boolean值 (书中写的int值, 是错误的). 允许/禁止 Nagle's 算法 (更多关于Nagle's 算法, 见 [http://en.wikipedia.org/wiki/Nagle%27s\\_algorithm](http://en.wikipedia.org/wiki/Nagle%27s_algorithm)), 默认下, 它为FALSE, 但是它可以在任意时刻设置.

现在, 设置和获取一个选项可以通过 `NetworkChannel.getOption()` 和 `NetworkChannel.setOption()` 方法完成:

```
<T> T getOption(SocketOption<T> name) throws IOException
<T> NetworkChannel setOption(SocketOption<T> name, T value) throws IOException
```

检测一个特定通道 (一个网络套接字) 支持的选项可以在那个通道上调用 `NetworkChannel.supportedOptions()` 方法完成:

```
Set<SocketOption<?>> supportedOptions()
```

## 写一个TCP服务器/客户端应用程序

离我们写一个TCP教程的目标还远 (FIXME: It is far from our aim to write a TCP tutorial), 因此这是一个非常好的文档和大的主题, 包含了许多技术概念和理论, 但是我们将给出一个快速概览. TCP 就像一个电话连接-它通过一个套接字在两个端点之间建立一个连接, 套接字在通信期间至始至终保持打开. TCP 的主要功能就是提供点到点的通信机制. 在一台机器上的一个进程与另一台机器上的另一个进程或者在相同的机器上通信. 一个唯一的TCP连接由5个元素标识: 服务器的IP地址和端口, 客户端的IP地址和端口, 和协议 (TCP/IP, UDP, 等). 服务器监听一个单独的端口, 在同一时间与很多客户端交流. TCP 提供了许多优点 (比如, 与UDP比较 (FIXME: over UDP)) 包括数据包. TCP 负责许多重要任务, 包括将数据分成包, 缓冲的数据, 跟踪丢失的包 (重新发送丢失的或者无序的包), 根据应用程序处理能力控制传输数据的速率. 更多的是, TCP 支持字节数组或者使用流发送数据, 在Java中非常普遍.

### 阻塞 vs 非阻塞机制

当你决定去写一个Java TCP服务器/客户端应用程序, 你必须考虑你是否需要去写一个阻塞或者非阻塞应用程序. 这个决定是重要的因为实现是不同的, 复杂性可能也是关键.

一个阻塞机制的主要特性是假定一个给定的线程不能做任何事情直到I/O完全接收, 在一些情况中可能采取一个while-应用程序的流程是阻塞的, 因为方法不能立即返回. 非阻塞机制, 另一方面, 立即将I/O请求放入队列, 然后返回应用程序流程的控制 (方法立即返回). 请求随后将被内核处理.

从一个Java开发者的观点, 你也必须仔细考虑这些机制包含的复杂程序. 非阻塞机制比阻塞机制实现更加复杂, 但是它们允许你获得更好的性能和可扩展性.

---

注意: 非阻塞机制不同于异步机制 (尽管这个经常被争论, 取决于你问谁). 比如, 在一个非阻塞环境中, 如果一个回答不能立即返回, API立即使用一个错误返回, 不做其它的什么, 当在一个异步环境中, API总是立即返回, 开始了幕后的努力, 来服务你的请求 (FIXME: having started a behind-the-scenes effort to serve your request). 换句话说, 使用一个非阻塞机制, 一个函数不会等待当在栈上的时候, 使用一个异步机制, (FIXME: work may continue on behalf of the function call after that call has left the stack). 异步与并行 (作为线程) 更加类似, 而非阻塞通常引用一个轮询.

---

阻塞和非阻塞模式自从NIO都被实现了, 但是我们将尝试去使用新的NIO.2特性, (FIXME: spice up the code).

在后面的章节中, 我们将开发两种类型的应用程序. 让我们从一个使用阻塞机制的简单的应用程序开始.

## 写一个阻塞的TCP服务器

为了更好的理解如何完成这个任务最简单的方式就是遵循一个简单明了的一组步骤,伴随着代码块,在讨论的末尾将被聚合在一起.我们想去开发一个单线程的阻塞的TCP服务器,将回复给客户端它从客户端获得任意的东西.许多完成这个的步骤对于其它阻塞的TCP服务器同样适用.

### 创建一个新的服务器套接字通道

第一个步骤包含了为一个面向流的监听套接字创建一个可选择的通道,这个通常多亏于 `java.nio.channels.ServerSocketChannel` 类,这个类被多个并发线程使用是安全的.更确切地说,这个任务通过 `ServerSocketChannel.open()` 方法完成,如这里所示:

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
```

记住一个新的创建的服务器套接字通道是没有绑定或者连接的.绑定和连接在后面的步骤中将完成.

你可以通过调用 `ServerSocketChannel.isOpen()` 方法检查一个服务器套接字是否已经打开或者已经成功打开,返回相应的 `Boolean` 值:

```
if (serverSocketChannel.isOpen()) {  
    ...  
}
```

### 配置阻塞机制

如果服务器套接字通道已经成功打开,是时候去指定阻塞机制了.为了这样我们调用接收一个 `boolean` 值的 `ServerSocketChannel.configureBlocking()` 方法.如果我们传递 `true`,我们将使用阻塞机制;如果我们传递 `false`,我们将使用非阻塞机制:

```
serverSocketChannel.configureBlocking(true);
```

注意这个方法返回一个 `SelectableChannel` 对象,代表了一个可通过 `Selector` 被多路复用的通道.这个非常有用当我们在非阻塞模式中;因此我们这里将先忽略.

### 设置服务器套接字通道选项

这是一个可选的步骤.没有必须的选项(你可以使用默认值),但是我们将明确设置一些选项来向你展示这个如何做.更确切地说,一个服务器套接字通道支持两个选项: `SO_RCVBUF` 和 `SO_REUSEADDR`.我们将设置它们两个,如这里所示:

```
serverSocketChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4 * 1024);  
serverSocketChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true);
```

你可以通过调用继承的方法 `supportedOptions()` 方法来找到一个服务器套接字通道支持的选项:

```
Set<SocketOption<?>> options = serverSocketChannel.supportedOptions();  
for (SocketOption<?> option : options) System.out.println(option);
```

### 绑定服务器套接字通道

此时,我们可以绑定通道的套接字到一个本地地址和配置套接字来监听连接.为了这么做我们调用新的 `ServerSocketChannel.bind()` 方法(这个方法在之前的"NetworkChannel概览"小结介绍过).我们的服务器将在本机(127.0.0.1),端口5555(任意选择)等待传入的连接:

```
final int DEFAULT_PORT = 5555;
```



```
final String IP = "127.0.0.1";
serverSocketChannel.bind(new InetSocketAddress(IP, DEFAULT_PORT));
```

另一通用的方式有一个创建的一个InetSocketAddress对象组成,而不用指定IP地址,只需要端口(有一个构造是这样的)。在这种情况下,IP地址是通配符地址,端口数是指定的值。通配符地址是一个特殊的本地IP地址,只能被绑定操作使用,通常意味着"任意":

```
serverSocketChannel.bind(new InetSocketAddress(DEFAULT_PORT));
```

警告: 当你使用一个IP通配符地址,小心避免任何(FIMXE: undesirable complications),可能会发生如果你有多个网络接口使用独立的IP地址。在这种情况下,你不能保证如何去完成这个而不会有问題,推荐绑定套接字到一个指定的网络地址,而不是使用一个通配符。

除此之外,还有一个bind()方法,接受地址绑定到套接字和等待连接的最大的数目:

```
public abstract ServerSocketChannel bind(SocketAddress local, int pc) throws IOException
```

本地地址也会自动被分配如果我们传递null到bind()方法。你也可以通过调用ServerSocketChannel.getLocalAddress()方法得到绑定的本地地址,这个方法是从NetworkChannel接口继承的。这个方法返回null如果服务器套接字通道还没有绑定。

```
System.out.println(serverSocketChannel.getLocalAddress());
```

## 接收连接

在打开和绑定后,我们最终达到了接收的里程碑。因为我们在阻塞模式中,接收一个连接将阻塞应用程序直到一个新的连接可用或者一个I/O错误发生。我们通过调用ServerSocketChannel.accept()方法来标志我们的渴望去接收一个新的连接。当一个新的连接可用,这个方法为新的连接返回客户端套接字通道(或者更简单的,套接字通道)。这是一个SocketChannel类的实例,代表了一个面向流的连接套接字的可选择通道。

```
SocketChannel socketChannel = serverSocketChannel.accept();
```

注意: 尝试去调用一个未绑定的服务器套接字通道的accept()方法将抛出NotYetBoundException异常。

一旦我们接受一个新的连接,我们可以通过调用SocketChannel.getRemoteAddress()方法得到远程地址。这个一个新的方法在Java 7中(NIO.2),它返回这个通道的套接字连接到的远程地址:

```
System.out.println("Incoming connection from: " + socketChannel.getRemoteAddress());
```

## 通过一个连接传输数据

此时,服务器和客户端可以通过一个连接传输数据。它们可以发送和接收不同类型的数据包,映射为字节数组或者使用流连同标准的Java文件I/O机制。实现传输(发送/接收)是灵活的,特定实现处理,因此它包含许多方面。比如,我们的服务器我们选择使用ByteBuffer,我们要记住这是一个回显服务器-它从服务器读取什么,就写回什么。这里是传输的代码片段:

```
ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
...
```

```
while (socketChannel.read(buffer) != -1) {
    buffer.flip();
    socketChannel.write(buffer);
    if (buffer.hasRemaining()) {
        buffer.compact();
    } else {
        buffer.clear();
    }
}
```

SocketChannel类为ByteBuffer提供了一组read()/write()方法.由于他们相当直观,我们只是列出它们:

- 从这个通道读取顺序的字节到给定的缓冲区.这些方法返回读取的字节数目或者-1如果通道已经达到了流的末尾:

```
public abstract int read(ByteBuffer dst) throws IOException
public final long read(ByteBuffer[] dsts) throws IOException
public abstract long read(ByteBuffer[] dsts, int offset, int length) throws IOException
```

- 从给定的缓冲区往这个通道中写入顺序的字节.这些方法写入的字节数目;它可能为0:

```
public abstract int write(ByteBuffer src) throws IOException
public final long write(ByteBuffer[] srcs) throws IOException
public abstract long write(ByteBuffer[] srcs, int offset, int length) throws IOException
```

### 使用流代替缓冲区

如你所知,通道使用缓冲区是非常友好的,但是如果你决定去使用流代替(InputStream和OutputStream),你需要使用下面的代码;一旦你获得一个I/O流,你可以进一步探索标准的Java文件I/O机制.

```
InputStream in = socketChannel.socket().getInputStream();
OutputStream out = socketChannel.socket().getOutputStream();
```

### 为I/O关闭连接

你可以通过调用新NIO.2的SocketChannel.shutdownInput()或者SocketChannel.shutdownOutput()方法为I/O关闭连接而不关闭通道.为输入(或者读)关闭连接拒绝进一步的读,通过-1表明达到流的末尾.为输出(或者写)关闭连接将拒绝任何的写,通过抛出ClosedChannelException.

```
//shut down connection for reading
socketChannel.shutdownInput();

//shut down connection for writing
socketChannel.shutdownOutput();
```

这些方法非常有用如果你想拒绝读/写尝试而不会关闭通道.检查连接当前是否为I/O关闭连接可以通过下面的代码完成:

```
boolean inputdown = socketChannel.socket().isInputShutdown();
boolean outputdown = socketChannel.socket().isOutputShutdown();
```

### 关闭通道

当一个连接变为无用的,它必须关闭.为了这个,你可以调用SocketChannel.close()方法(这

个将关闭服务器监听传入的连接,它只是为客户端关闭通道),和/或者  
ServerSocketChannel.close()方法(这个方法将关闭服务器监听传入的连接;后面的客户端不能再定位服务器)。

```
serverSocketChannel.close();  
socketChannel.close();
```

作为选择,我们可以通过替换代码为Java 7 try-with-资源特性关闭任意的资源-这是可行的,因为ServerSocketChannel和SocketChannel类实现了AutoCloseable接口.使用这个特性将保证资源被自动关闭.如果你不熟悉try-with-资源特性,看看  
<http://download.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>

## 把它们一起放到回显服务器

现在我们有了一切,我们需要创建我们的回显服务器.将之前的代码块放在一起,加上必要的导入语句和意大利面条式代码(spaghetti code)等.将给我提供以下的回显服务器:

```
public class Main {  
    public static void main(String[] args) {  
        final int DEFAULT_PORT = 5555;  
        final String IP = "127.0.0.1";  
  
        ByteBuffer buffer = ByteBuffer.allocateDirect(1024);  
  
        //create a new server-socket channel  
        try (ServerSocketChannel serverSocketChannel = ServerSocketChannel.open()) {  
  
            //continue if it was successfully created  
            if (serverSocketChannel.isOpen()) {  
  
                //set the blocking mode  
                serverSocketChannel.configureBlocking(true);  
                //set some options  
                serverSocketChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4 * 1024);  
                serverSocketChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true);  
                //bind the server-socket channel to local address  
                serverSocketChannel.bind(new InetSocketAddress(IP, DEFAULT_PORT));  
  
                //display a waiting message while ... waiting clients  
                System.out.println("Waiting for connections ...");  
  
                //wait for incoming connections  
                while (true) {  
                    try (SocketChannel socketChannel = serverSocketChannel.accept()) {  
                        System.out.println("Incoming connection from: " +  
socketChannel.getRemoteAddress());  
  
                        //transmitting data  
                        while (socketChannel.read(buffer) != -1) {  
  
                            buffer.flip();  
  
                            socketChannel.write(buffer);  
  
                            if (buffer.hasRemaining()) {  
                                buffer.compact();  
                            } else {  
                                buffer.clear();  
                            }  
                        }  
                    }  
                }  
            } catch (IOException ex) {
```

```
        }
    }
    } else {
        System.out.println("The server socket channel cannot be opened!");
    }
} catch (IOException ex) {
    System.err.println(ex);
}
}
```

## 写一个阻塞的TCP客户端

什么是一个好的服务器而没有客户端？我们不想找到这个问题的答案，所以让我们为回显服务器开发一个客户端。假设以下的场景：客户端连接到我们的服务器，发送一个"Hello!"消息，然后持续发送0到100之间的随机数直到数字50生成。当数字50生成，客户端停止发送然后关闭通道。服务器将回显（写回）一切它从客户端读取的。现在我们有了一个场景，让我们看看实现它的步骤。

### 创建一个新的套接字通道

第一个步骤是为一个面向流的连接的套接字创建一个可选择的通道。这个使用 `java.nio.channels.SocketChannel` 类完成，这个类被多个并发线程使用是线程安全的。更确切地说，这个任务通过 `SocketChannel.open()` 方法完成。如下：

```
SocketChannel socketChannel = SocketChannel.open();
```

记住一个新的创建的套接字通道是未连接的。创建和连接一个套接字通道是一次性地 (**FIMXE: in a single shot**) 包括调用 `SocketChannel.open(SocketAddress)` 方法。也可以分成两个步骤来这样做，如我们将要讨论的。

你可以通过调用 `SocketChannel.isOpen()` 方法检测一个套接字 (**这里应该是套接字，而不是服务器套接字**) 是否已经打开或者已经成功打开，这个方法返回相应的 `Boolean` 值。

```
if (socketChannel.isOpen()) {
    ...
}
```

### 配置阻塞机制

如果套接字通道已经成功打开，是时候去指定阻塞机制。我们将传递 `true` 值，因为我们想去激活阻塞机制：

```
socketChannel.configureBlocking(true);
```

### 设置套接字通道选项

一个套接字通道支持下面的选项：`SO_RCVBUF`，`SO_LINGER`，`IP_TOS`，`SO_OOBINLINE`，`SO_REUSEADDR`，`TCP_NODELAY`，`SO_KEEPALIVE`，和 `SO_SNDBUF`。它们中的一些展示在了下面：

```
socketChannel.setOption(StandardSocketOptions.SO_RCVBUF, 128 * 1024);
socketChannel.setOption(StandardSocketOptions.SO_SNDBUF, 128 * 1024);
socketChannel.setOption(StandardSocketOptions.SO_KEEPALIVE, true);
socketChannel.setOption(StandardSocketOptions.SO_LINGER, 5);
```

你可以通过调用继承的方法 `supportedOptions()` 方法来找到一个套接字 (**这里应该是套接字，而不是服务器套接字**) 支持的选项：



```
Set<SocketOption<?>> options = socketChannel.supportedOptions();
for(SocketOption<?> option : options) System.out.println(option);
```

## 连接通道的套接字

在打开一个套接字通道后和(随意绑定它),你应该连接到远程地址(服务器端地址)。因为我们在阻塞模式中,连接到一个远程地址将阻塞应用程序直到一个新的连接可用或者一个I/O错误发生。这个通过调用`SocketChannel.connect()`方法表示打算去连接,传递远程地址作为一个`InetSocketAddress`到这个方法,如下(记住我们的回显服务器运行在127.0.0.1,端口5555):

```
final int DEFAULT_PORT = 5555;
final String IP = "127.0.0.1";
socketChannel.connect(new InetSocketAddress(IP, DEFAULT_PORT));
```

这个方法返回一个boolean值代表了一个成功的连接尝试。你可以使用这个boolean值去检测连接的可用性,直到通过这个连接发送/接收数据包。另外,同样的检查可以通过调用`SocketChannel.isConnected()`方法完成,像这样:

```
if (socketChannel.isConnected()) {
    ...
}
```

注意:明显地,在真实世界情况中,在应用程序这种硬编码IP地址是一个不好的实践。在这种情况下,客户端将只能运行在和服务器同一台机器上, (FIXME: sort of defeats the purpose of remote communication)。在你的案例中,客户端很可能使用服务器的主机名而不是IP地址(很可能通过DNS配置)。IP地址通常变化,有时候甚至是通过DHCP动态分配的。

## 通过一个连接传输数据

连接已经建立,所以我们可以开始传输数据包。下面的代码发送"Hello!"消息,然后发送随机数直到数字50生成。我们使用`ByteBuffer`,`CharBuffer`,和`SocketChannel`类的`read()/write()`方法(我们之前列出了这些方法当我们开发服务器端代码的时候,所以你应该已经熟悉它们):

```
ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
ByteBuffer helloBuffer = ByteBuffer.wrap("Hello!".getBytes());
ByteBuffer randomBuffer;
CharBuffer charBuffer;
Charset charset = Charset.defaultCharset();
CharsetDecoder decoder = charset.newDecoder();
...
socketChannel.write(helloBuffer);

while (socketChannel.read(buffer) != -1) {

    buffer.flip();

    charBuffer = decoder.decode(buffer);
    System.out.println(charBuffer.toString());

    if (buffer.hasRemaining()) {
        buffer.compact();
    } else {
        buffer.clear();
    }
}
```

```
int r = new Random().nextInt(100);
if (r == 50) {
    System.out.println("50 was generated! Close the socket channel!");
    break;
} else {
    randomBuffer = ByteBuffer.wrap("Random number:"
                                   .concat(String.valueOf(r)).getBytes());
    socketChannel.write(randomBuffer);
}
}
```

## 关闭通道

当一个通道变得无用,它必须被关闭.为了这么做,你可以调用`SocketChannel.close()`方法,然后客户端将从服务器断开:

```
socketChannel.close();
```

再一次,Java 7 `try-with-resources`特性可用来自动关闭.

## 将它们一起放到客户端

现在我们有了一切,我们需要去创建客户端.将所有需要的元素放在一起,将提供给我们下面的客户端:

```
public class Main {
    public static void main(String[] args) {
        final int DEFAULT_PORT = 5555;
        final String IP = "127.0.0.1";
        ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
        ByteBuffer helloBuffer = ByteBuffer.wrap("Hello !".getBytes());
        ByteBuffer randomBuffer;
        CharBuffer charBuffer;
        Charset charset = Charset.defaultCharset();
        CharsetDecoder decoder = charset.newDecoder();

        //create a new socket channel
        try (SocketChannel socketChannel = SocketChannel.open()) {

            //continue if it was successfully created
            if (socketChannel.isOpen()) {

                //set the blocking mode
                socketChannel.configureBlocking(true);
                //set some options
                socketChannel.setOption(StandardSocketOptions.SO_RCVBUF, 128 * 1024);
                socketChannel.setOption(StandardSocketOptions.SO_SNDBUF, 128 * 1024);
                socketChannel.setOption(StandardSocketOptions.SO_KEEPALIVE, true);
                socketChannel.setOption(StandardSocketOptions.SO_LINGER, 5);
                //connect this channel's socket
                socketChannel.connect(new InetSocketAddress(IP, DEFAULT_PORT));

                //check if the connection was successfully accomplished
                if (socketChannel.isConnected()) {

                    //transmitting data
                    socketChannel.write(helloBuffer);

                    while (socketChannel.read(buffer) != -1) {

                        buffer.flip();
```

```
charBuffer = decoder.decode(buffer);
System.out.println(charBuffer.toString());

if (buffer.hasRemaining()) {
    buffer.compact();
} else {
    buffer.clear();
}

int r = new Random().nextInt(100);
if (r == 50) {
    System.out.println("50 was generated! Close the socket
channel!");
    break;
} else {
    randomBuffer = ByteBuffer.wrap("Random
number:".concat(String.valueOf(r)).getBytes());
    socketChannel.write(randomBuffer);
}
} else {
    System.out.println("The connection cannot be established!");
}
} else {
    System.out.println("The socket channel cannot be opened!");
}
} catch (IOException ex) {
    System.err.println(ex);
}
}
```

## 测试阻塞的回显应用程序

测试应用程序是一个简单的任务.首先,启动服务器,直到你看到"Waiting for connections ..."消息.接下来启动客户端然后看看输出.下面是一些可能的服务器输出:

---

```
Waiting for connections ...
Incoming connection from: /127.0.0.1:49911
```

---

这里是一些可能的客户端的输出:

---

```
Hello !
Random number:71
Random number:60
Random number:22
Random number:4
Random number:60
Random number:13
...
50 was generated! Close the socket channel!
```

---

## 写一个非阻塞TCP客户端/服务器应用程序

在我们开始开发前,让我们简要浏览下非阻塞API,从NIO开始已经可用,所以对你来说它不应该完全是陌生的.记住,我们不会研究关于这个的太多细节,你很可能已经知道.

非阻塞套接字是关于允许在一个通道上进行I/O操作而不会阻塞进程.这个一开始正如一个阻塞的应用程序(FIXME:The story begins exactly as in a blocking application):服务器端打开,绑定到一个本地地址,从客户端接收请求,很明显,客户端,打开,连接到远程地址,发送请求到服务器端.

事情开始疯狂当所有的非阻塞基础的主要实体-`java.nio.channels.Selector`类达到现场.一个`Selector`通过一个无参的`open()`方法创建(`Selector`在Java 7中已经被修改).基本上,这个类有能力去识别当一个或者多个通道可用来数据传输和序列化请求用来帮助服务器去满足它的客户端(它监视每个记录的套接字通道).

此外,`Selector`处理多个套接字I/O 读/写操作在一个单独的线程中, - 解决了每个套接字连接专门的一个线程.在API中,`Selector`是一个`java.nio.channels.SelectableChannel`的多路复用器,可通过`SelectableChannel`的`register()`方法注册(在`ServerSocketChannel`和`SocketChannel`类中可用,是`SelectableChannel`的间接子类),通过`Selector`回收分配给通道的资源.

## 使用SelectionKey类

如果你持续追踪,让我们深入!每次一个通道向一个`Selector`注册,它通过`java.nio.channels.SelectionKey`类的实例代表,这些实例被称为选择的键-Java 7没有修改这个类.把键当做被选择器使用的帮助者去分类客户端请求-每个帮助者(键)代表了一个单独的客户端子请求,包含了标识客户端的信息和请求的类型(连接,读,写,等).当注册的时候,我们指明选择器,通常,产生的键的感兴趣的集合(感兴趣的集合标识了被选择器监视的键的通道的操作).一个键有四种可能的类型:

- `SelectionKey.OP_ACCEPT(acceptable)`: 关联的客户端请求一个连接(通常在服务器端创建用来表明一个客户端请求一个连接)
- `SelectionKey.OP_CONNECT(connectable)`: 服务器接收连接(通常在客户端创建)
- `SelectionKey.OP_READ(readable)`: 这个表明一个读操作
- `SelectionKey.OP_WRITE(writable)`: 这个表明一个写操作

一个选择器负责维护三种选择键的集合:

- `key-set`: 包含的键表示所有当前通道向`Selection`注册的集合
- `selected-key`: 包含了这样的键的集合,在之前的一个选择操作期间,每个键的通道被检测到在键的感兴趣集合中至少有一个操作准备好.
- `cancelled-key`: 包含了已经被取消的键的集合,但是它们的通道还没有被注销.

---

注意: 所有的三种集合在一个新创建的选择器中是空的.选择器自身是安全的被多个并发线程使用,但是它们的键然而不是这样.

---

当一些事情发生在战场上,选择器唤醒然后创建相应的键(`SelectionKey`类的实例).每个键持有关于应用程序形成的请求和请求的类型(尝试/接收连接和读/写操作)的信息.

选择器等待传入的连接进入一个无限循环(等待在选择器上记录的事件).通常,`Selector.select()`方法在循环的第一行,它阻塞应用程序直到至少有一个通道被选择,选择器的`Selector.wakeup()`方法被调用,或者当前的现在被中断-无论哪个先发生.(另外,"`select()`带有超时时间"是可用的,作为非阻塞的方法称为`selectionNow()`).

选择器等待一个客户端去尝试一个连接,当发生的时候,服务器应用程序通过选择器得到创建的集合.每个键,它检查类型(每个被处理的键通过明确地调用一个`Iterator`的`remove()`方法从集合中移除-这个可以防止相同的键再一次进入).可接受的键在这里被捕获,当`SelectionKey.isAcceptable()`方法返回`true`的时候,服务器通过调用`accept()`方法定位客户端套接字,设置它为非阻塞,使用`OP_READ`和/或`OP_WRITE`操作注册到选择器中.

此时,客户端套接字通道为读/写操作注册到了选择器中.按照(In keeping with)这个趋势,当客户端在套接字通道上写入数据,选择器将高速服务器这里有一些数据要读取-为了这样,SelectionKey.isReadable()方法返回true.如果客户端尝试去从服务器读取数据,处理是类似的,但是相反服务器写入数据,SelectionKey.isWritable()方法返回true.

图8-1展示了一个非阻塞流程的图解.

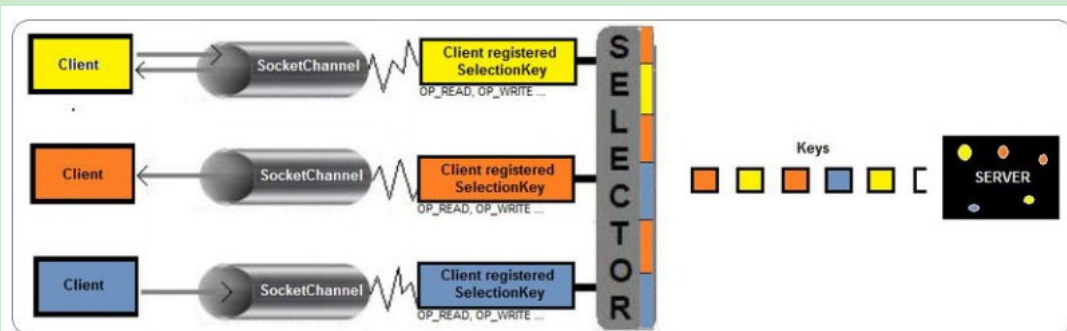


Figure 8-1. Selector base non-blocking flow.

这样,服务器准备好了(FIXME: rock).

注意,在非阻塞模式中,一个I/O操作可以传输少于请求的字节(部分读或写),或者可能根本没有字节.

### 使用Selector的方法

接下来,我们将在这节复习下要调用的方法,以及更多的方法,在在后面概览(下面的大部分描述是从官方Java 7 doc取出的).

- Selector.open(): 创建一个新的选择器.
- Selector.select(): 通过执行一个阻塞的选择操作选择键的集合.
- Selector.select(t): 和select()一样,但是只阻塞执行指定的毫秒数.如果时间到期,没有什么被选择,返回0.
- Selector.selectNow(): 和select()一样,但是是一个非阻塞的操作.它返回0如果没有什么被选择.
- Selector.selectedKeys(): 返回这个选择器的选择的键作为Set<SelectionKey>.
- Selector.keys(): 反正这个选择器的集合,作为Set<SelectionKey>.
- Selector.wakeup(): 引起第一个还没有返回的选择操作立即返回.
- SelectionKey.isValid(): 检查键是否有效.一个键的无效的如果它被取消,它的通道被关闭或者选择器被关闭.
- SelectionKey.isReadable(): 测试这个键的通道是否准备好读.
- SelectionKey.isWritable(): 测试这个键的通道是否准备好写.
- SelectionKey.isAcceptable(): 测试这个键的通道是否准备好接受一个新的套接字连接.
- SelectionKey.isConnectable(): 测试这个键的通道它的套接字连接操作是否完成或者失败.
- SelectionKey.cancel(): 请求这个键的通道的注册被它的选择器取消.
- SelectionKey.interestOps(): 获取这个键的感兴趣的集合.
- SelectionKey.interestOps(t): 设置键的感兴趣的集合为给定的值.
- SelectionKey.readyOps(): 获取这个键的准备好操作的集合.

此外,ServerSocketChannel和SocketChannel包含了register()方法,用来注册当前的通道到给定的选择器并返回一个选择键.它获得选择器,产生的键的感兴趣集合,和产生的键的附件



(可能为null)。

```
public final SelectionKey register(Selector s,int p,Object a) throws ClosedChannelException
```

### 写服务器

基于这些方法和之前的讨论,我们写了下面的非阻塞的回显服务器(每个步骤注释了来帮助你有一个好的理解):

```
public class Main {

    private Map<SocketChannel, List<byte[]>> keepDataTrack = new HashMap<>();
    private ByteBuffer buffer = ByteBuffer.allocate(2 * 1024);

    private void startEchoServer() {

        final int DEFAULT_PORT = 5555;

        //open Selector and ServerSocketChannel by calling the open() method
        try (Selector selector = Selector.open();
            ServerSocketChannel serverSocketChannel = ServerSocketChannel.open()) {

            //check that both of them were successfully opened
            if ((serverSocketChannel.isOpen()) && (selector.isOpen())) {

                //configure non-blocking mode
                serverSocketChannel.configureBlocking(false);

                //set some options
                serverSocketChannel.setOption(StandardSocketOptions.SO_RCVBUF, 256 * 1024);
                serverSocketChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true);

                //bind the server socket channel to port
                serverSocketChannel.bind(new InetSocketAddress(DEFAULT_PORT));

                //register the current channel with the given selector
                serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

                //display a waiting message while ... waiting!
                System.out.println("Waiting for connections ...");

                while (true) {
                    //wait for incoming events
                    selector.select();

                    //there is something to process on selected keys
                    Iterator keys = selector.selectedKeys().iterator();

                    while (keys.hasNext()) {
                        SelectionKey key = (SelectionKey) keys.next();

                        //prevent the same key from coming up again
                        keys.remove();

                        if (!key.isValid()) {
                            continue;
                        }

                        if (key.isAcceptable()) {
                            acceptOP(key, selector);
                        } else if (key.isReadable()) {
                            this.readOP(key);
                        }
                    }
                }
            }
        }
    }
}
```

```

        } else if (key.isWritable()) {
            this.writeOP(key);
        }
    }
}

} else {
    System.out.println("The server socket channel or selector cannot be
opened!");
}
} catch (IOException ex) {
    System.err.println(ex);
}
}

//isAcceptable returned true
private void acceptOP(SelectionKey key, Selector selector) throws IOException {

    ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
    SocketChannel socketChannel = serverChannel.accept();
    socketChannel.configureBlocking(false);

    System.out.println("Incoming connection from: " + socketChannel.getRemoteAddress());

    // write an welcome message
    socketChannel.write(ByteBuffer.wrap("Hello!\n".getBytes("UTF-8")));

    //register channel with selector for further I/O
    keepDataTrack.put(socketChannel, new ArrayList<byte[]>());
    socketChannel.register(selector, SelectionKey.OP_READ);
}

//isReadable returned true
private void readOP(SelectionKey key) {
    try {
        SocketChannel socketChannel = (SocketChannel) key.channel();

        buffer.clear();

        int numRead = -1;
        try {
            numRead = socketChannel.read(buffer);
        } catch (IOException e) {
            System.err.println("Cannot read error!");
        }

        if (numRead == -1) {
            this.keepDataTrack.remove(socketChannel);
            System.out.println("Connection closed by: " +
socketChannel.getRemoteAddress());
            socketChannel.close();
            key.cancel();
            return;
        }

        byte[] data = new byte[numRead];
        System.arraycopy(buffer.array(), 0, data, 0, numRead);
        System.out.println(new String(data, "UTF-8") + " from " +
socketChannel.getRemoteAddress());

        // write back to client
        doEchoJob(key, data);
    } catch (IOException ex) {
        System.err.println(ex);
    }
}

```

```

    }
}

//isWritable returned true
private void writeOP(SelectionKey key) throws IOException {

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

    List<byte[]> channelData = keepDataTrack.get(socketChannel);
    Iterator<byte[]> its = channelData.iterator();

    while (its.hasNext()) {
        byte[] it = its.next();
        its.remove();
        socketChannel.write(ByteBuffer.wrap(it));
    }

    key.interestOps(SelectionKey.OP_READ);
}

private void doEchoJob(SelectionKey key, byte[] data) {
    SocketChannel socketChannel = (SocketChannel) key.channel();
    List<byte[]> channelData = keepDataTrack.get(socketChannel);
    channelData.add(data);

    key.interestOps(SelectionKey.OP_WRITE);
}

public static void main(String[] args) {
    Main main = new Main();
    main.startEchoServer();
}
}

```

## 写客户端

关于客户端,结构基本是一样的,稍有些差异:

- 首先,客户端套接字通道使用SelectionKey.OP\_CONNECT选择注册,因此客户端想被选择器通知当服务器接收到连接。
- 第二,客户端不用尝试一个无限的连接,因此服务器可以不被激活;另外,Selector.select()带有超时的方法适合它(超时时间500到1,000毫秒将做这项工作)。
- 第三,客户端必须坚持键是否可连接(也就是说,如果SelectionKey.isConnectedable()方法返回true)。如果键是连接的,它混合套接字通道的isConnectionPending()和finishConnect()方法在一个有条件的语句中用来关闭等待的连接。当你需要去判断一个连接操作是否在这个通道上处理,调用SocketChannel.isConnectionPending()方法,这个方法返回一个Boolean值。同样,完成连接一个套接字通道的处理可以通过SocketChannel.finishConnect()方法完成。

最后,客户端准备好I/O操作。我们重现如在阻塞的客户端/服务器应用程序中的方案:客户端连接到我们的服务器然后发送一个"Hello!"消息,然后持续发送0到100之间的随机数直到数字50生成。当50生成,客户端停止发送并关闭通道。服务器将回显(回复)它从客户端读取的一切。

```

public class Main {
    public static void main(String[] args) {
        final int DEFAULT_PORT = 5555;
        final String IP = "127.0.0.1";

        ByteBuffer buffer = ByteBuffer.allocateDirect(2 * 1024);
        ByteBuffer randomBuffer;
    }
}

```

```
CharBuffer charBuffer;

Charset charset = Charset.defaultCharset();
CharsetDecoder decoder = charset.newDecoder();

//open Selector and ServerSocketChannel by calling the open() method
try (Selector selector = Selector.open();
     SocketChannel socketChannel = SocketChannel.open()) {

    //check that both of them were successfully opened
    if ((socketChannel.isOpen()) && (selector.isOpen())) {

        //configure non-blocking mode
        socketChannel.configureBlocking(false);

        //set some options
        socketChannel.setOption(StandardSocketOptions.SO_RCVBUF, 128 * 1024);
        socketChannel.setOption(StandardSocketOptions.SO_SNDBUF, 128 * 1024);
        socketChannel.setOption(StandardSocketOptions.SO_KEEPALIVE, true);

        //register the current channel with the given selector
        socketChannel.register(selector, SelectionKey.OP_CONNECT);

        //connect to remote host
        socketChannel.connect(new java.net.InetSocketAddress(IP, DEFAULT_PORT));

        System.out.println("Localhost: " + socketChannel.getLocalAddress());

        //waiting for the connection
        while (selector.select(1000) > 0) {

            //get keys
            Set keys = selector.selectedKeys();
            Iterator its = keys.iterator();

            //process each key
            while (its.hasNext()) {
                SelectionKey key = (SelectionKey) its.next();

                //remove the current key
                its.remove();

                //get the socket channel for this key
                try (SocketChannel keySocketChannel = (SocketChannel) key.channel()) {

                    //attempt a connection
                    if (key.isConnectable()) {

                        //signal connection success
                        System.out.println("I am connected!");

                        //close pendent connections
                        if (keySocketChannel.isConnectionPending()) {
                            keySocketChannel.finishConnect();
                        }

                        //read/write from/to server
                        while (keySocketChannel.read(buffer) != -1) {

                            buffer.flip();

                            charBuffer = decoder.decode(buffer);
                            System.out.println(charBuffer.toString());
                        }
                    }
                }
            }
        }
    }
}
```

```

        if (buffer.hasRemaining()) {
            buffer.compact();
        } else {
            buffer.clear();
        }

        int r = new Random().nextInt(100);
        if (r == 50) {
            System.out.println("50 was generated! Close the
socket channel!");

            break;
        } else {
            randomBuffer = ByteBuffer.wrap("Random
number:".concat(String.valueOf(r)).getBytes("UTF-8"));
            keySocketChannel.write(randomBuffer);
        }
    }
}
} catch (IOException ex) {
    System.err.println(ex);
}
}
} else {
    System.out.println("The socket channel or selector cannot be opened!");
}
} catch (IOException ex) {
    System.err.println(ex);
}
}
}

```

## 测试非阻塞回显应用程序

测试这个应用程序是个简单的任务。首先,我们启动服务器等待直到你看到"Waiting for connections ...."消息。接下来通过启动一组客户端和看看输出,图8-2展示了运行服务器和3个客户端实例的示例。

```

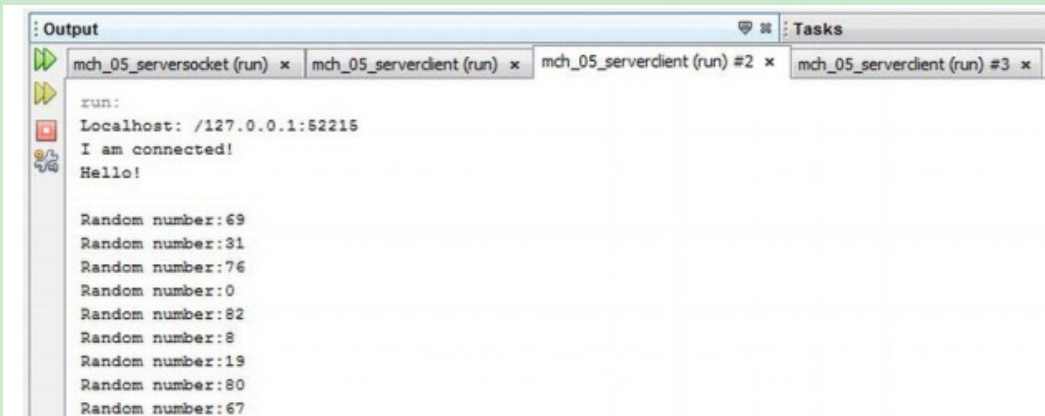
run:
Waiting for connections ...
Incoming connection from: /127.0.0.1:52212
Random number:47 from /127.0.0.1:52212
Incoming connection from: /127.0.0.1:52215
Random number:69 from /127.0.0.1:52215
Incoming connection from: /127.0.0.1:52218
Random number:35 from /127.0.0.1:52218
Random number:82 from /127.0.0.1:52212
Random number:31 from /127.0.0.1:52215
Random number:52 from /127.0.0.1:52218
Random number:9 from /127.0.0.1:52212
Random number:76 from /127.0.0.1:52215
Random number:41 from /127.0.0.1:52218
Random number:37 from /127.0.0.1:52212
Random number:0 from /127.0.0.1:52215
Random number:26 from /127.0.0.1:52218
Random number:18 from /127.0.0.1:52212
Random number:82 from /127.0.0.1:52215

```

Figure 8-2. Non-blocking server echo application output.

图8-3展示了客户单2的输出。





```
run:
Localhost: /127.0.0.1:52215
I am connected!
Hello!

Random number: 69
Random number: 31
Random number: 76
Random number: 0
Random number: 82
Random number: 8
Random number: 19
Random number: 80
Random number: 67
```

Figure 8-3. Non-blocking client echo application output

记住即使它看上去像一个多线程应用程序,但它只是一个基于多路复用技术的单线程应用程序。

## 写UDP服务器/客户端应用程序

由于TCP已经有了它的辉煌时刻(FIXME: Since TCP has had its moment of glory),是时候关注UDP。UDP建立于IP之上,有一些重要特性组成。举例来说,数据包大小在单个IP数据包中包含的数量被限制-最多65507字节;这是65535字节IP数据包大小减去最小的IP头20字节,和减去8字节的UDP头。另外,每个数据包是独立的,被分别处理(没有包知道其它包)。此外,数据包可以任意顺序达到,它们中的一些可以丢失而不会发送被通知,或者它们可以比它们被处理的更快或者更慢达到-不能保证传递/接收数据在一个特定的序列,同样不能保证传递的数据将被接收到。

因为发送者不能追踪数据包的路由器,每个数据包封装了远程的IP地址和端口。如果TCP像一个电话,UDP就像一封信。发送者在信封上(UDP数据包)写上接受者的地址(远程IP和端口)和发送者的地址(本地IP和端口),将信(要发送的数据)放到信封中,然后发信。它不知道新是否将会达到接受者。此外,近期的信可以比旧的更快达到,一封信可能永远不会达到-信不知道彼此。记住TCP适合高-可靠性数据传输而UDP适合低-负载传递。通常,在应用程序中使用UDP,可靠性不是关键,而速度是。UDP更适合从一个系统到另一系统发送消息-当顺序不是重要的,并且你不需要所有的消息被其它的机器接收到。

在接下来的章节,我们将写一个基于UDP的单线程阻塞客户端/服务器应用程序。我们将从服务器端开始。

### 写一个UDP服务器

为了帮助你理解,我们将开发的过程分为不连续的步骤,并采用前面的NIO.2的特性打算去增加性能和简化开发。再一次,我们将写一个回显服务器和一个发送一些文本的客户端并且接收回来。

#### 创建一个面向数据报的服务器套接字通道

写一个客户端/服务器应用程序整个流程包含了java.nio.channels.DatagramChannel类,代表了一个面向数据报套接字的线程安全的可选择通道。因此,我们将通过创建一个新的DatagramChannel开始我们的服务器,可通过调用NIO.2 DatagramChannel.open()方法完成。这个方法得到一个参数称为一个协议族参数,实际上是一个java.net.ProtocolFamily对象。这是一个在NIO.2中的新的接口,代表了通信协议族-目前它有一个实现,java.net.StandardProtocolFamily,定义了两个枚举常量:

- StandardProtocolFamily.INET: IP版本4 (IPv4)
- StandardProtocolFamily.INET6: IP版本6 (IPv6)

所以我们可以像这样创建IPv4的面向数据报的服务器套接字:

```
DatagramChannel datagramChannel = DatagramChannel.open(StandardProtocolFamily.INET);
```

旧的无参的`DatagramChannel.open()`方法仍然可用并且可被使用,因为它没有被废弃.但是在这种情况下,通道的套接字的协议族是平台(配置)依赖的,因此是未指定的.

你可以通过调用`DatagramChannel.isOpen()`方法检查一个面向数据报的套接字通道是否已经打开或者已经成功打开,这个方法返回响应的`Boolean`值:

```
if (datagramChannel.isOpen()) {  
    ...  
}
```

一个客户端面向数据报的套接字通道可以相同的方式创建和检查.

### 设置面向数据报套接字通道选项

面向数据报套接字通道支持以下的选项(尽管你可以在大多数情况下使用默认值):

`SO_REUSEADDR`, `SO_BROADCAST`, `IP_MULTICAST_LOOP`, `SO_SNDBUF`, `IP_MULTICAST_TTL`, `IP_TOS`, `IP_MULTICAST_IF`, 和 `SO_RCVBUF`.作为一个示例,我们可以设置输入和输入缓冲区被网络实现使用,如下:

```
datagramChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4 * 1024);  
datagramChannel.setOption(StandardSocketOptions.SO_SNDBUF, 4 * 1024);
```

注意你可以通过调用继承的方法`supportedOptions()`来获取一个面向数据报套接字通道支持的选项:

```
Set<SocketOption<?>> options = datagramChannel.supportedOptions();  
for(SocketOption<?> option : options) System.out.println(option);
```

### 绑定面向数据报套接字通道

此时我们绑定通道的套接字到一个本地地址和配置套接字去监听连接.为了这样,我们调用新的`DatagramChannel.bind()`方法(这个方法在之前的"NetworkChannel"小结介绍过).我们的服务器将在主机(127.0.0.1)和端口5555(可随意选择)在等待传入的连接:

```
final int LOCAL_PORT = 5555;  
final String LOCAL_IP = "127.0.0.1";  
datagramChannel.bind(new InetSocketAddress(LOCAL_IP, LOCAL_PORT));
```

通配符地址也可以被使用:

```
datagramChannel.bind(new InetSocketAddress(LOCAL_PORT));
```

本地地址可以被自动分配如果我们传递`null`到`bind()`方法.你可以通过调用`DatagramChannel.getLocalAddress()`(书中写的是`ServerSocketChannel.getLocalAddress()`,是错误的)方法获得绑定的本地地址,这个方法从`NetworkChannel`接口继承而来.这个方法返回`null`如果面向数据报的套接字还没有绑定.

```
System.out.println(datagramChannel.getLocalAddress());
```

### 传输数据包

此时我们的服务器准备好去接收和发送数据包。因为UDP是无连接的网络协议，你 cannot 通过读和写一个 `DatagramChannel` 像你操作其它的通道一样—随后，你将看到如何在UDP上建立一个连接。取而代之，你使用 `DatagramChannel.send()` 和 `DatagramChannel.receive()` 方法来发送和接收数据。

当你发送一个数据包，你传递给 `send()` 方法一个 `ByteBuffer`，包含了宝贵的数据和远程地址（服务器或者客户端，决定于是谁发送）。这里是这个是怎样工作的，根据官方文档（见

<http://download.oracle.com/javase/7/docs/api/>）：

如果这个通道处于非阻塞模式，没有足够的空间在底层的输出缓冲区中，或者这个通道处于阻塞模式，足够的空间变得可用，在给定的缓冲区中的剩余字节作为一个单独的数据报传输给给定的远程地址。这个方法可以在任意时刻被调用。如果另一个线程已经在这个通道上初始化了一个写的操作，这个方法的调用将阻塞直到第一个操作完成。如果这个通道的套接字没有绑定，这个方法将首先让套接字绑定到一个自动分配的地址。如同如果通过调用 `bind()` 方法，而使用一个 `null` 参数。

这个方法返回发送的字节的数目。

当你接受一个数据包，你传递缓冲区 (`ByteBuffer`) 到这个方法，数据报将被传输到这个缓冲区。再一次，这里是它如何工作的，根据文档（见

<http://download.oracle.com/javase/7/docs/api/>）：

如果一个数据报立即可用，或者这个通道处于阻塞模式，最终变得可用，然后数据报被拷贝给定的字节缓冲区，它的源地址返回。如果这个通道处于非阻塞模式，一个数据报不能立即可用，这个方法立即返回 `null`。这个方法可以在任意时刻被调用。如果另一个线程已经在通道上已经初始化一个读操作，这个方法的调用将阻塞直到第一个操作完成。如果这个通道的套接字没有绑定，这个方法首先将套接字绑定到一个自动分配的地址，如同如果通过调用 `bind()` 方法，而使用一个 `null` 参数。

这个方法返回数据报的源地址，或者 `null` 如果这个通道处于非阻塞模式并且没有数据报立即可用。远程地址可以用来找到发送一个回复数据包到哪里。

另外，你可以通过调用 `DatagramChannel.getRemoteAddress()` 方法得到远程地址。这个是 Java 7 (NIO.2) 中的新方法，它返回这个通道的套接字连接的远程地址—记住，对一个UDP无连接情况，这个方法返回 `null` (**FIXME: keep in mind that for a UDP connectionless case, this method returns null. 没怎么理解**)。

```
System.out.println("Connected to: " + datagramChannel.getRemoteAddress());
```

我们的数据报回显服务器将在一个无限循环中监听传入的数据包，在阻塞模式中（默认情况下），当一个数据包达到，它将从远程地址和数据得到。数据基于远程地址发回。

```
final int MAX_PACKET_SIZE = 65507;
ByteBuffer echoText = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);
...
while (true) {
    SocketAddress clientAddress = datagramChannel.receive(echoText);
    echoText.flip();
    System.out.println("I have received " + echoText.limit() + " bytes from " +
        clientAddress.toString() + "! Sending them back
    ...");
    datagramChannel.send(echoText, clientAddress);
    echoText.clear();
}
```

## 关闭数据报通道

当一个数据报通道变得无用，它必须被关闭。为了这样做，你可以调用 `DatagramChannel.close()` 方法：

```
datagramChannel.close();
```

在声明一次,Java 7 try-with-resource特性可以用来自动关闭。

### 将它们一起放到服务器中

现在我们准备好了一切,我们需要去创建我们的服务器.将之前的所有信息放在一起,将提供我们如下服务器:

```
public class Main {
    public static void main(String[] args) {
        final int LOCAL_PORT = 5555;
        final String LOCAL_IP = "127.0.0.1"; //modify this to your local IP
        final int MAX_PACKET_SIZE = 65507;

        ByteBuffer echoText = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);

        //create a new datagram channel
        try (DatagramChannel datagramChannel =
DatagramChannel.open(StandardProtocolFamily.INET)) {

            //check if it the channel was successfully opened
            if (datagramChannel.isOpen()) {

                System.out.println("Echo server was successfully opened!");
                //set some options
                datagramChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4 * 1024);
                datagramChannel.setOption(StandardSocketOptions.SO_SNDBUF, 4 * 1024);
                //bind the channel to local address
                datagramChannel.bind(new InetSocketAddress(LOCAL_IP, LOCAL_PORT));
                System.out.println("Echo server was binded on: " +
datagramChannel.getLocalAddress());
                System.out.println("Echo server is ready to echo ...");

                //transmitting data packets
                while (true) {

                    SocketAddress clientAddress = datagramChannel.receive(echoText);

                    echoText.flip();
                    System.out.println("I have received " + echoText.limit() + " bytes from
" + clientAddress.toString() + "! Sending them back ...");
                    datagramChannel.send(echoText, clientAddress);
                    echoText.clear();

                }
            } else {
                System.out.println("The channel cannot be opened!");
            }
        } catch (Exception ex) {
            if (ex instanceof ClosedChannelException) {
                System.err.println("The channel was unexpected closed ...");
            }
            if (ex instanceof SecurityException) {
                System.err.println("A security exception occurred ...");
            }
            if (ex instanceof IOException) {
                System.err.println("An I/O error occurred ...");
            }

            System.err.println("\n" + ex);
        }
    }
}
```

```

    }
}
}

```

## 写一个无连接的UDP客户端

写一个无连接UDP客户端类似于写一个UDP服务器。如之前展示的以同样的方式创建一个新的DatagramChannel然后设置你需要的选项，你可以开始发送和接收数据包。一个面向数据报的客户端套接字通道不必绑定到一个本地地址，因为服务器将从每个接收到的数据报获取IP地址和端口——换句话说，它知道客户端在哪里(FIXME: where the client lives)。此外，如果这个通道的套接字没有绑定，send()和receice()方法经首先让套接字(客户端或服务器)绑定到一个自动分配的地址，如果如果调用bind()方法，使用一个null参数。但是记住如果服务器端自动绑定(不明确)，客户端应该注意选择地址(更确切地说，选择的IP地址和端口)。相反也是正确的，如果服务器发送第一个数据包。

我们的客户端直到服务器在地址127.0.0.1，端口5555；因此，它发送第一个数据包然后从它接收回复。这里是代码：

```

public class Main {
    public static void main(String[] args) throws IOException {
        final int REMOTE_PORT = 5555;
        final String REMOTE_IP = "127.0.0.1"; //modify this accordingly if you want to test
remote
        final int MAX_PACKET_SIZE = 65507;

        CharBuffer charBuffer = null;
        Charset charset = Charset.defaultCharset();
        CharsetDecoder decoder = charset.newDecoder();
        ByteBuffer textToEcho = ByteBuffer.wrap("Echo this: I'm a big and ugly
server!".getBytes());
        ByteBuffer echoedText = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);

        //create a new datagram channel
        try (DatagramChannel datagramChannel =
DatagramChannel.open(StandardProtocolFamily.INET)) {

            //check if it the channel was successfully opened
            if (datagramChannel.isOpen()) {

                //set some options
                datagramChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4 * 1024);
                datagramChannel.setOption(StandardSocketOptions.SO_SNDBUF, 4 * 1024);

                //transmitting data packets
                int sent = datagramChannel.send(textToEcho, new InetSocketAddress(REMOTE_IP,
REMOTE_PORT));
                System.out.println("I have successfully sent " + sent + " bytes to the Echo
Server!");

                datagramChannel.receive(echoedText);
                Thread.sleep(5000);
                echoedText.flip();
                charBuffer = decoder.decode(echoedText);
                System.out.println(charBuffer.toString());
                echoedText.clear();

            } else {
                System.out.println("The channel cannot be opened!");
            }
        } catch (Exception ex) {
            if (ex instanceof ClosedChannelException) {
                System.err.println("The channel was unexpected closed ...");
            }
        }
    }
}

```



```
    }  
    if (ex instanceof SecurityException) {  
        System.err.println("A security exception occurred ...");  
    }  
    if (ex instanceof IOException) {  
        System.err.println("An I/O error occurred ...");  
    }  
  
    System.err.println("\n" + ex);  
}  
}
```

### 测试UDP无连接回显应用程序

测试这个应用程序的一个简单的任务,启动服务器然后等待直到你看到下面的消息:

```
Echo server was successfully opened!  
Echo server was binded on: /127.0.0.1:5555  
Echo server is ready to echo .
```

然后启动客户端看下输出.这里是从UDP服务器可能的输出.

```
Echo server was successfully opened!  
Echo server was binded on: /127.0.0.1:5555  
Echo server is ready to echo ...  
I have received 37 bytes from /127.0.0.1:49155! Sending them back ...
```

这里是UDP客户端可能的一些输出:

```
I have successfully sent 37 bytes to the Echo Server!  
Echo this: I'm a big and ugly server!
```

**警告** 不要望手动去停止UDP服务器在完成测试后!

### 写一个连接的UDP客户端

如果你想去使用`DatagramChannel.read()`和`DatagramChannel.write()`方法(基于`ByteBuffer`),而不是`send()`和`receive()`,你需要去写一个连接的UDP客户端.在一个连接的客户端场景中,通道的套接字被配置,这样它只从给定的远程端地址接受数据报,只向给定的远程端地址发送数据报.在连接建立后,数据报不能接受/发送 从/到任意其它的地址.一个面向数据报的套接字保持连接直到它明确的断开或者直到它被关闭.

这种类型的客户端必须明确地调用`DatagramChannel.connect()`方法,传递服务器端远程地址给它,如下:

```
final int REMOTE_PORT = 5555;  
final String REMOTE_IP = "127.0.0.1";  
datagramChannel.connect(new InetSocketAddress(REMOTE_IP, REMOTE_PORT));
```

注意,不像`SocketChannel.connect()`方法,这个方法实际上不跨越网络发送/接收任意数据包,因为UDP是一个无连接的协议-这个方法返回相当快,在一个具体的场景中不会阻塞应用程序.没

有必要用`finishConnect()`或者`isConnectionPending()`方法.这个方法可以在任意时刻被调用,因为它不会影响此时被调用的已经在处理中的读/写操作.如果这个通道的套接字没有绑定,这个方法将首先让套接字绑定到一个自动分配的地址,如同如果调用`bind()`方法,使用一个`null`参数.

你可以通过调用`DatagramChannel.isConnected()`方法检查一个连接的状态.相应的`Boolean`值将返回(`true`如果这个通道的套接字打开和连接):

```
if (datagramChannel.isConnected()) {  
    ...  
}
```

下面的应用程序是一个我们的UDP回显服务器的UDP连接的客户端.它连接到远程地址然后使用`read()/write()`方法传输数据:

```
public class Main {  
    public static void main(String[] args) throws IOException {  
        final int REMOTE_PORT = 5555;  
        final String REMOTE_IP = "127.0.0.1"; //modify this accordingly if you want to test  
remote  
        final int MAX_PACKET_SIZE = 65507;  
  
        CharBuffer charBuffer = null;  
        Charset charset = Charset.defaultCharset();  
        CharsetDecoder decoder = charset.newDecoder();  
        ByteBuffer textToEcho = ByteBuffer.wrap("Echo this: I'm a big and ugly  
server!".getBytes());  
        ByteBuffer echoedText = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);  
  
        //create a new datagram channel  
        try (DatagramChannel datagramChannel =  
DatagramChannel.open(StandardProtocolFamily.INET)) {  
  
            //set some options  
            datagramChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4 * 1024);  
            datagramChannel.setOption(StandardSocketOptions.SO_SNDBUF, 4 * 1024);  
  
            //check if it the channel was successfully opened  
            if (datagramChannel.isOpen()) {  
  
                //connect to remote address  
                datagramChannel.connect(new InetSocketAddress(REMOTE_IP, REMOTE_PORT));  
  
                //check if it the channel was successfully connected  
                if (datagramChannel.isConnected()) {  
  
                    //transmitting data packets  
                    int sent = datagramChannel.write(textToEcho);  
                    System.out.println("I have successfully sent " + sent + " bytes to the  
Echo Server!");  
  
                    datagramChannel.read(echoedText);  
  
                    echoedText.flip();  
                    charBuffer = decoder.decode(echoedText);  
                    System.out.println(charBuffer.toString());  
                    echoedText.clear();  
  
                } else {  
                    System.out.println("The channel cannot be connected!");  
                }  
            } else {  
                System.out.println("The channel cannot be opened!");  
            }  
        }  
    }  
}
```

```

    }
    } catch (Exception ex) {
        if (ex instanceof ClosedChannelException) {
            System.err.println("The channel was unexpected closed ...");
        }
        if (ex instanceof SecurityException) {
            System.err.println("A security exception occurred ...");
        }
        if (ex instanceof IOException) {
            System.err.println("An I/O error occurred ...");
        }

        System.err.println("\n" + ex);
    }
}
}

```

总所周知的read() / write() 方法在DatagramChannel中是可用的:

- 从这个通道中读取字节序列到给定的缓冲区. 这些方法返回读取的字节数目或者-1如果这个通道已经达到了流的末尾:

```

public abstract int read(ByteBuffer dst) throws IOException
public final long read(ByteBuffer[] dsts) throws IOException
public abstract long read(ByteBuffer[] dsts, int offset, int length) throws IOException

```

- 从给定的缓冲区往这个通道中写入字节序列. 这个方法会写入的字节数目; 可以为0:

```

public abstract int write(ByteBuffer src) throws IOException
public final long write(ByteBuffer[] srcs) throws IOException
public abstract long write(ByteBuffer[] srcs, int offset, int length) throws IOException

```

## 测试UDP连接的回显应用程序

测试这个应用程序是一个简单的任务. 首先, 启动服务直到你看到这些消息:

---

```

Echo server was successfully opened!
Echo server was binded on: /127.0.0.1:5555
Echo server is ready to echo ...

```

---

然后启动客户端看下输出. UDP服务器输出在这里展示:

---

```

Echo server was successfully opened!
Echo server was binded on: /127.0.0.1:5555
Echo server is ready to echo ...
I have received 37 bytes from /127.0.0.1:57374! Sending them back ...

```

---

这里是UDP客户端输出:

---

```

I have successfully sent 37 bytes to the Echo Server!
Echo this: I'm a big and ugly server!

```

---

## 组播

你可能已经熟悉术语组播。但是,如果你不熟悉,让我们这个概念有一个简要的概览。不用学术的描述和定义,组播被认为是广播的互联网版本。比如,一个电视基站从一个源广播它的信号,但是这个信号可以达到信号区域内的居住的任何人-只有没有必要的器材或者拒绝去捕获这个信号接收传输将失败。

在计算机世界中,TV基站可以被当成一个主节点,或者机器,传播数据报到目的地主机组。这个要感谢组播传输服务,从一个源发送数据报在单次调用中到多个接受者-这与单播传输服务是相反的,特定于基于点对点连接的高层的网络协议,需要重复的单播发送相同的数据到多个点(实际上,它发送数据的一份拷贝到每个点)。

组播引入了组的概念代表了数据报的接受者,一个组通过一个D类IP地址标识(一个组播组IPv4地址在224.0.0.1和239.255.255.255之间)。当一个新的接收者(客户端)想加入组播组,它需要通过相应的IP地址加入到组然后监听传入的数据报。

许多现实的案例可基于组播程序化,比如在线会议,新闻发布,广告,邮箱组和数据共享管理。

下面,我们讨论NIO.2对组播的贡献。

## MulticastChannel概览

NIO.2带有一些新的接口映射一个网络通道支持IP组播。这就是java.nio.channels.MulticastChannel接口。在API层面上,这个NetworkChannel的子接口,在之前的章节中提到过,它由一个单独的类实现:DatagramChannel类。

基本上,它定义而来两个join()方法和一个close()方法。关注于join()方法,这里是一个简要的概览:

- 第一个join()方法被一个想加入一个组播组的客户端调用用来接收传入的数据报。我们需要传递组的IP地址和网络接口(要加入的组的网络接口)(你将立即看到如何去检查你的机器有一个组播能力的网络接口)。如果指明的组成功加入,这个方法返回一个MembershipKey实例。这在NIO.2是新的,它是一个令牌代表了一个IP组播组的成员资格(下一节看到)。

```
MembershipKey join(InetAddress g, NetworkInterface i) throws IOException
```

- 第二个join()方法也用来加入一个组播组。在这种情况下,无论如何,我们指明一个源地址,组成员可开始接收数据报,成员资格是累积的,意味着这个方法可以使用相同的组和接口被再次调用,接收被其它源地址发送到组的数据报。

```
MembershipKey join(InetAddress g, NetworkInterface i, InetAddress s) throws IOException
```

注意 一个组播通道可以加入几个组播组,包括在多个接口上相同的组。

close()方法用来去丢弃成员资格(如果任意的组被加入)和关闭通道。

## MembershipKey概览

当你加入一个组播组的时候,你得到一个成员资格key,可用来执行组内部的不同类型的操作。最通用的在这里呈现:

- 阻塞/开启(Block/unblock): 你可以通过调用block()方法从一个特定的源阻塞发送数据报并传递源地址。此外,你可以使用相同的地址通过调用unblock()方法疏通阻塞的源。
- 获取组: 你可以得到组播组的源地址,通过调用无参的group()方法创建的成员资格key。这个方法返回一个InetAddress对象。
- 获取通道: 你可以获取通道,通过调用无参的channel()方法创建的成员资格key。这个方法返回一个MulticastChannel对象。
- 获取源地址: 如果成员资格key是源地址特定的(只从一个特定的源地址接收数据报),你可以

通过调用无参的`sourceAddress()`方法得到源地址.这个方法返回一个`InetAddress`对象.

- 获取网络接口: 你可以获取网络接口,通过调用参数的`networkInterface()`方法创建的成员资格`key`.这个方法返回一个`NetworkInterface`对象.
- 检查有效性: 你可以通过调用`isValid()`方法来检查一个成员资格是否有效.这个方法返回一个`boolean`值.
- 丢弃: 你可以通过无参的`drop()`方法丢弃成员资格(通道不在接受任何发送到组的数据报).

一个成员资格`key`是有效的当你创建它的时候并且保持有效直到通过使用`drop()`方法或者通道被关闭成员资格被丢弃.

### NetworkInterface概览

`NetworkInterface`类代表了一个网络接口,由一个名称和一组分配到这个接口的IP地址组成.它用来标识一个组播组加入的本地接口.比如,下面的代码将返回关于在你的机器上能找到的所有网络接口的信息:

```
public class Main {
    public static void main(String argv[]) throws Exception {
        Enumeration enumInterfaces = NetworkInterface.getNetworkInterfaces();
        while (enumInterfaces.hasMoreElements()) {
            NetworkInterface net = (NetworkInterface) enumInterfaces.nextElement();
            System.out.println("Network Interface Display Name: " + net.getDisplayName());
            System.out.println(net.getDisplayName() + " is up and running ?" + net.isUp());
            System.out.println(net.getDisplayName() + " Supports Multicast: " +
net.supportsMulticast());
            System.out.println(net.getDisplayName() + " Name: " + net.getName());
            System.out.println(net.getDisplayName() + " Is Virtual: " + net.isVirtual());
            System.out.println("IP addresses:");
            Enumeration enumIP = net.getInetAddresses();
            while (enumIP.hasMoreElements()) {
                InetAddress ip = (InetAddress) enumIP.nextElement();
                System.out.println("IP address:" + ip);
            }
        }
    }
}
```

这个应用程序将返回关于在你的机器上能找到的所有网络接口的信息,每一个将给出它的显示名称(一个人类可读的字符串描述网络设备)和名称(实际的名称用来标识一个网络接口).此外,每个网络接口被检查是否支持组播,如果是一个虚拟的(一个子接口),并且如果它是启动和运行的.

图8-4展示了在我的机器上的输出的片段.框中的接口用来测试组播应用程序-它的名称为`eth3`,将在后面的客户端/服务器组播应用程序中用来指明这个接口.



```

: Output - mch_01 (run)
Network Interface Display Name: RAS Async Adapter
RAS Async Adapter is up and running ?false
RAS Async Adapter Supports Multicast: true
RAS Async Adapter Name: ppp1
RAS Async Adapter Is Virtual: false
IP addresses:
Network Interface Display Name: Realtek RTL8139/810x Family Fast Ethernet NIC
Realtek RTL8139/810x Family Fast Ethernet NIC is up and running ?true
Realtek RTL8139/810x Family Fast Ethernet NIC Supports Multicast: true
Realtek RTL8139/810x Family Fast Ethernet NIC Name: eth3
Realtek RTL8139/810x Family Fast Ethernet NIC Is Virtual: false
IP addresses:
IP address:/46.214.194.219
IP address:/2002:2ed6:c2db:b:45e7:e8cb:6564:5c5a
IP address:/fec0:0:0:b:45e7:e8cb:6564:5c5a%2
IP address:/2002:2ed6:c2db:b:11b5:d11:db0e:24f4
IP address:/fe80:0:0:0:45e7:e8cb:6564:5c5a%11
IP address:/2002:2ed6:c2db:b:0:0:0:0
IP address:/fec0:0:0:0:b:0:0:0:0%2
Network Interface Display Name: Atheros AR5005G Wireless Network Adapter
...

```

Figure 8-4. Find out local interfaces.

### 写一个UDP组播服务器

在这节,我们将写一个UDP组播服务器发送包含服务器的当前的日期和时间的组数据报.这个将每10秒重复一次.现在,我们已经有了写UDP客户端/服务器应用程序的经验,没有必要一步一步重复整个过程.我将只指出传输一个普通的UDP客户端/服务器应用程序到一个UDP组播客户端/服务器应用程序的主要差异.

我们通过调用`open()`方法创建一个新的`DatagramChannel`对象开始开发流程.接下来,我们看到两个重要选项: `IP_MULTICAST_IF` 和 `SO_REUSEADDR`.第一个将指明在这种情况下网络接口用来IP组播数据报,第二个应该在绑定套接字之前允许-这个需要允许组的多个成员去绑定相同的地址:

```

NetworkInterface networkInterface = NetworkInterface.getByNames("eth3");
...
datagramChannel.setOption(StandardSocketOptions.IP_MULTICAST_IF, networkInterface);

datagramChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true);

```

下面,我们通过调用`bind()`方法绑定通道的套接字到本地地址:

```

final int DEFAULT_PORT = 5555;
datagramChannel.bind(new InetSocketAddress(DEFAULT_PORT));

```

最后,我们准备传输数据报的代码.因为我们需要每10秒发送服务器日期和时间给组,我们需要一个无限循环,包含了10秒休眠周期和调用`send()`方法.组播组IP地址我们随意选择如225.4.5.6,它通过一个`InetAddress`对象映射:

```

final int DEFAULT_PORT = 5555;
final String GROUP = "225.4.5.6";
ByteBuffer datetime;
...
while (true) {
    //sleep for 10 seconds
    try {
        Thread.sleep(10000);
    } catch (InterruptedException ex) {}
    System.out.println("Sending data ...");
}

```

```
datetime = ByteBuffer.wrap(new Date().toString().getBytes());
datagramChannel.send(datetime, new
                        InetSocketAddress(InetAddress.getByName(GROUP), DEFAULT_PORT));
datetime.flip();
}
```

将所有一切放在一起将产生下面的应用程序：

```
public class Main {
    public static void main(String[] args) {
        final int DEFAULT_PORT = 5555;
        final String GROUP = "225.4.5.6";
        ByteBuffer datetime;

        //create a new channel
        try (DatagramChannel datagramChannel =
DatagramChannel.open(StandardProtocolFamily.INET)) {

            //check if the channel was successfully created
            if (datagramChannel.isOpen()) {

                //get the network interface used for multicast
                NetworkInterface networkInterface = NetworkInterface.getByName("eth3");

                //set some options
                datagramChannel.setOption(StandardSocketOptions.IP_MULTICAST_IF,
networkInterface);
                datagramChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true);

                //bind the channel to the local address
                datagramChannel.bind(new InetSocketAddress(DEFAULT_PORT));
                System.out.println("Date-time server is ready ... shortly I'll start sending
...");

                //transmitting datagrams
                while (true) {

                    //sleep for 10 seconds
                    try {
                        Thread.sleep(10000);
                    } catch (InterruptedException ex) {
                    }
                    System.out.println("Sending data ...");

                    datetime = ByteBuffer.wrap(new Date().toString().getBytes());
                    datagramChannel.send(datetime, new
InetSocketAddress(InetAddress.getByName(GROUP), DEFAULT_PORT));
                    datetime.flip();

                }

            } else {
                System.out.println("The channel cannot be opened!");
            }
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

### 写一个UDP组播客户端

UDP组播客户端的代码基本和服务器相同，稍有区别。首先，你可能想去检查远程地址实际上是否

为一个组播地址-这个通过调用`InetAddress.isMulticastAddress()`方法是可行的, 返回一个`boolean`. 第二, 因为这是一个客户端, 它必须通过调用`join()`方法中的一个加入到组. 数据报传输代码只适用于从UDP组播服务器接收数据报. 下面的应该程序的一个可行的客户端实现:

```
public class Main {
    public static void main(String[] args) {
        final int DEFAULT_PORT = 5555;
        final int MAX_PACKET_SIZE = 65507;
        final String GROUP = "225.4.5.6";

        CharBuffer charBuffer = null;
        Charset charset = Charset.defaultCharset();
        CharsetDecoder decoder = charset.newDecoder();
        ByteBuffer datetime = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);

        //create a new channel
        try (DatagramChannel datagramChannel =
DatagramChannel.open(StandardProtocolFamily.INET)) {

            InetAddress group = InetAddress.getByName(GROUP);
            //check if the group address is multicast
            if (group.isMulticastAddress()) {
                //check if the channel was successfully created
                if (datagramChannel.isOpen()) {

                    //get the network interface used for multicast
                    NetworkInterface networkInterface = NetworkInterface.getByName("eth3");

                    //set some options
                    datagramChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true);
                    //bind the channel to the local address
                    datagramChannel.bind(new InetSocketAddress(DEFAULT_PORT));

                    //join the multicast group and get ready to receive datagrams
                    MembershipKey key = datagramChannel.join(group, networkInterface);

                    //wait for datagrams
                    while (true) {

                        if (key.isValid()) {

                            datagramChannel.receive(datetime);
                            datetime.flip();
                            charBuffer = decoder.decode(datetime);
                            System.out.println(charBuffer.toString());
                            datetime.clear();
                        } else {
                            break;
                        }
                    }

                } else {
                    System.out.println("The channel cannot be opened!");
                }
            } else {
                System.out.println("This is not multicast address!");
            }
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

### 阻塞和非阻塞数据报

有时候加入组播组可能带给你不是预期的数据报(原因和这里不相关).你可以通过调用 `MembershipKey.block()` 方法阻塞从一个发送者接受数据报,并将发送者的 `InetAddress` 传递给这个方法.另外,你可以开启相同的发送者,然后再一次启动从它接收数据报,通过调用 `MembershipKey.unblock()` 方法,并传递相同的 `InetAddress`.通常,你将处于以下两种场景:

- 你有一组你想加入的发送者的地址.假设地址存储在一个 `List` 中,你可以循环它然后分别加入每个地址,如这里所示:

```
List<InetAddress> like = ...;
DatagramChannel datagramChannel = ...;
if(!like.isEmpty()){
    for(InetAddress source: like){
        datagramChannel.join(group, network_interface, source);
    }
}
```

- 你有一组你不想加入的发送者的地址.假设地址存储在一个 `List` 中,你可以循环它然后分别阻塞每个地址,如这里所示:

```
List<InetAddress> dislike = ...;
DatagramChannel datagramChannel = ...;
MembershipKey key = datagramChannel.join(group, network_interface);
if(!dislike.isEmpty()){
    for(InetAddress source: dislike){
        key.block(source);
    }
}
```

### 测试UDP组播应用程序

测试这个应用程序是一个简单的任务.首先,启动组播服务器直到你看到这些消息:

---

```
Date-time server is ready ... shortly I'll start sending ..
```

---

然后启动客户端看下输出.这里是UDP组播服务器的一些简单示例输出:

---

```
Date-time server is ready ... shortly I'll start sending ...
Sending data ...
Sending data ...
Sending data ...
Sending data ...
Sending data ...
```

---

这里是UDP客户端的输出(客户端在一会儿之后启动):

---

```
Sat Oct 08 09:40:09 GMT+02:00 2011
Sat Oct 08 09:40:19 GMT+02:00 2011
```

---

在这个示例上执行一些测试将暴露一些问题.当服务器启动,它发送数据报而不会注意是否由客户

端正在监听这些数据报。同样,它不会注意当客户端加入或者离开组。相反,客户端启动接收数据报当它加入组,但是它不会注意服务器是否由于某些原因而停止发送。如果服务器掉线(go offline),客户端仍然在等待,它将再次接收当服务器再次上线和开始发送。它可以作为一个有趣的练习去尝试解决这些问题如果你的案例中需要更多的控制。同样,你可能想用线程去尝试,阻塞/非阻塞模式,无连接/连接的特性加入更多的灵活性和性能到你的组播应用程序中。

## 总结

这章覆盖了创建TCP/UDP客户端/服务器应用程序的NIO.2特性。如讨论的,NIO.2通过更新已存在的类使用新的方法和为写这样的应用程序加入了新的接口/类改进了这个支持。

这章以NetworkChannel接口开始,对所有的网络通道类通过了通用的方法。它也覆盖了专门用来同步套接字通道的主要的类: ServerSocketChannel, SocketChannel和DatagramChannel。也讨论了MulticastChannel接口-NetworkChannel的子接口映射一个网络通道支持IP组播。最后,你看到如何写一个单线程阻塞/非阻塞TCP客户端/服务器应用程序,单线程阻塞UDP客户端/服务器应用程序,单线程组播UDP客户端/服务器应用程序。