这章介绍UDP,用户数据报协议,和它在Java中的实现DatagramSocket和DatagramPacket.在这章我们只关注阻塞模式的点对点或者"单播"UDP.UDP通道I/O和非阻塞模式在第10章 组播(MulticaseSocket)中讨论,广播在第11章讨论.

9.1 概览

在这节我们简要浏览基本的单播UDP和如何在Java中编程.
UDP是指定在RFC 768中修订的.

9.1.1 UDP中的套接字

如我们在2.2.5节看到的,一个套接字是一个通信端点的抽象表示,与本机的一个IP地址和端口关联.在UDP套接字和TCP套接字之间有重大(significant)的区别:

(a) UDP套接字没有实现可靠的数据流.它们实现了不可靠的数据报,如在9.1.2节讨论的.
(b) UDP只有"主动的"套接字,与TCP对比,TCP有"主动的"和"被动的"套接字.
(c) UDP没有如TCP中的明确的"accept"阶段(*).
(d) UDP套接字没有明确的通过网络操作同时连接或者断开(FIXME: UDP sockets are not explicitly connected together or disconnected by network operations-together出现在这里有点让人不太明白).
(e) UDP套接字在Java中由DatagramSocket类型的对象表示.

UDP套接字和TCP套接字占据不同的"名称空间": 一个TCP套接字和一个UDP套接字总是有区别的,即使他们有相同的地址和端口.UDP套接字和TCP套接字不能交互连接.

---

* 如9.3.7节中讨论的,为了避免某些Java和内核的开销.一个发送者在做大量的发送之前可能会'连接'到远程地址和端口.这是一个本地操作,不是网络操作: 远程端不知道这个操作和任何的通信断开连接操作.

9.1.2 数据报

'数据报'是一个单一的传输,可能传递0次或多次.在同样的两个端点之间相对于(with respect to)其它的数据报它的顺序是不能保证的.(FIXME: Its sequencing with respect to other datagrams between the same two endpoints is not guaranteed)换句话说,它可能是无序传递的,或根本没有,或多次.
数据报在一个单一的IP数据包中发送.不像TCP流,数据报受限于(subject to)于大小的约束.

(a) IPv4协议限制它们为65507字节,大部分实现限制它们为8KB,尽管在一些实现中可以通过增加套接字的发送和接收缓冲区大小增加提高这个,如9.11节描述的.
(b) IPv4路由器有权(entitle)去分段任何的IP数据包,包括TCP分段和UDP数据报.不像TCP分段,然而,UDP数据报一旦被分段永远不会重新组装(reassemble),所以实际上被废弃了.IPv4 UDP的实际开发通常限制消息为521字节-单个IP数据包-为了避免分段问题(*).
(c) IPv6在IP层级上有"jumbograms"(FIXME: jumbograms查不出来是啥意思);这个允许UDP数据报达到2 - 1字节;见RFC 2675.然而,RFC接着说到(FIXME: Jumbograms are relevant only to IPv6 nodes that may be attached to links with a link MTU greater than 65,575 octets, and need not be implemented or understood by IPv6 nodes that do not support attachment to links with such large MTUs'. In other words, jumbograms can only be expected to be communicable among hosts which are all connected to such links.)

如果一个数据报完全传递,它完整的达到,也就是说,没有传输错误,丢弃和在字节中没有内部的顺序错误.然而,如果接收到的数据报大于可用空间,超过的数据安静的被忽略.

---

*

9.1.3 UDP的好处

(a) 由于协议是无连接的,连接或断开连接不需要网络开销(overhead).作为对比,我们已经在3.2.2节看到TCP需要三方数据包交换去建立连接,断开它需要四方数据包交换.
(b) UDP服务器的架构比TCP服务器的架构更加简单,没有连接的套接字去接收和关闭.
(c) 类似的,UDP客户端的架构也比TCP客户端简单,无需连接去创建或终止.

9.1.4 UDP的限制

(a) 不支持重新组装分段的数据报.
(b) 在实践中负载是非常有限的.数据报负载超过512字节会被路由器恰当的(apt)的分段,因此实际上会丢失(FIXME:therefore effectively lost).
(c) 不支持根据一般的网络条件(FIXME: pacing transmissions)。
(d) 不支持数据包排序.
(e) 不探测数据包丢失和重新传输.

其中有些限制比现实更常出现.数据报模型非常适合这样的应用程序:
(a) 请求-回复的交易(FIXME: Transactions are request-reply).
(b) 负载比较小.
(c) 服务器是无连接的.
(d) 事务是幂等的(FIXME: Transactions are idempotent).也就是说,可以重复而不会影响全部(overall)的计算结果(*).

数据报模型比TCP的流模型更加接近于实际的底层网络:包被交换,重新排序和丢失.数据报模型也比TCP流模型更加接近于许多应用程序的上层实现(overlying realities).本质上,数据报自然地提供了在"最后一个0"的传递保证.这个使得客户端
看到发送的请求相当简单:可以是传输已经接收到和确认,或者它被重新传输.
xxxxxxx

总结,UDP支持不可靠,无连接,大小限制的数据报通信在点对点架构中.每个端点创建一个数据报套接字.为了接收数据,UDP套接字必须绑定到一个端口.为了发送数据,一个数据报的数据包形成,通过一个数据报套接字发送到一个远程的IP地址和端口.

尽管UDP是一个点对点的协议,在许多实际的应用程序中,我们可以仍然区分作为客户端的发送请求的请求端点和等待回复,和作为服务器等待一个请求的响应端点和发送一个回复.

---

*

9.1.5 导入语句

下面的Java导入语句假设在这章的示例中至始至终存在.

```
import java.io.*;
import java.net.*;
import java.util.*;
```

9.2 简单的UDP服务器和客户端

在Java中,一个UDP套接字由java.net.DatagramSocket类型的对象表示;一个UDP数据包由java.net.DatagramPacket类型的对象表示.

9.2.1 简单的UDP服务器

最简单的可行的UDP服务器在Example 9.1中展示.

```
public class UDPServer implements Runnable {
    DatagramSocket socket;

    public UDPServer(int port) throws IOException {
        this.socket = new DatagramSocket(port);
    }

    @Override
    public void run() {
        for (;;) {
            try    {
                byte[] buffer = new byte[8192];
                DatagramPacket packet= new DatagramPacket(buffer, buffer.length);
                socket.receive(packet);
                new ConnectionHandler(socket, packet).run(); // 这里只是普通的方法调用,并没有启动线程.
            } catch (IOException e)    {
            // …
            }
        } // for (;;)
    } // run()
} // class

Example 9.1 Simple UDP server
```

这个的连接处理类和随后的服务在Example 9.2中展示.

```
class ConnectionHandler implements Runnable {
    DatagramSocket socket;
    DatagramPacket packet;

    ConnectionHandler(DatagramSocket socket, DatagramPacket packet)    {
        this.socket = socket;
```

```
        this.packet = packet;
    }

    @Override
    public void run() {
        handlePacket(socket, packet);
    }

    public void handlePacket(DatagramSocket socket,    DatagramPacket packet) {
        try    {
            byte[] buffer = packet.getData();
            int offset = packet.getOffset();
            int length = packet.getLength();
            // conversation not shown …
            // sets reply into buffer/offset/length
            packet.setData(buffer, offset, length);
            // write reply to the output
            socket.send(packet);
        } catch (IOException e) { /* ... */ }
    } // handlePacket()
} // class
```

Example 9.2 UDP server connection handler

Example 9.1是单线程设计的通常不怎么适用,由于它是顺序的处理客户端,不是并发的-一个新的客户端阻塞当之前的客户端在服务中.为了处理客户端并发,服务器必须使用为接收的连接使用一个不同的线程.这样的UDP服务器的最简单的的形式,使用了相同的连接处理类,在Example 9.3中描述.

```
public class ConcurrentUDPServer extends UDPServer {
    public ConcurrentUDPServer(int port) throws IOException    {
        super(port);
    }

    @Override
    public void run() {
        for (;;) {
            try    {
                byte[]  buffer= new byte[8192];
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                socket.receive(packet);
                new Thread(new ConnectionHandler(socket, packet)).start(); // 启动一个新的线程
            }
            catch (IOException e) {
            // …
            }
        } // for (;;)
    } // run()
} // class
```

Example 9.3 Simple UDP server—multithreaded

连接处理类简化了回复它的输入到它的输出-对于测试非常有用-在示例9.4展示.

```
class EchoConnectionHandler implements Runnable {
    DatagramSocket socket;
    DatagramPacket packet;
    ConnectionHandler(DatagramSocket socket, DatagramPacket packet)    {
        this.socket = socket;
        this.packet = packet;
    }

    @Override
    public void run() {
        handlePacket(socket, packet);
    }

    public void handlePacket(DatagramSocket socket,    DatagramPacket packet) {
        try {
            byte[] buffer = packet.getData();
            int offset = packet.getOffset();
            int length = packet.getLength();
            // The reply is the same as the request,
            // which is already in buffer/offset/length
            packet.setData(buffer, offset, length);
            // write reply to the output -- 写入回复到输出中
            socket.send(packet);
        }
        catch (IOException e) {
```

```
        // …
        }
    } // handlePacket()
} // class
```

Example 9.4 UDP server connection handler—echo service

9.2.2 简单的UDP客户端

客户端的DatagramSocket通常绑定到系统选择的端口.一个为之前的UDP服务器的简单的UDP客户端在Example 9.5展示.

```
public class UDPClient implements Runnable {
    DatagramSocket socket; // socket for communications
    InetAddress address; // remote host
    int port; // remote port

    public UDPClient(InetAddress address, int port) throws IOException {
        this.socket = new DatagramSocket();// ephemeral port - 临时端口
        this.address = address;
        this.port = port;
    }

    /**
     * Send the data in {buffer, offset, length}
     * and overwrite it with the reply.
     * @return the actual reply length.
     * @exception IOException on any error
     */
    public int sendReceive(byte[] buffer, int offset, int length) throws IOException {
        try    {
            // Create packet
            DatagramPacket packet = new DatagramPacket
            (buffer, offset, length, address, port);
            socket.send(packet);
            socket.receive(packet);
            return packet.getLength();
        } catch (IOException e)    {
        // …
        }
    } // sendReceive()
} // class
```

Example 9.5 Simple UDP client

9.2.3 DatagramPacket类

如我们在9.1.2节所看到的,一个UDP数据报是一个单独的传输单元,在Java中它很方便的作为一个单独的对象表示.这个对象是DatagramPacket.UDP不是流协议,所以TCP的基于流的编程技术用在UDP中不适用.不像TCP套接字,支持所熟悉的InputStream和OutputStream,UDP套接字在Java中通过发送和接收全部的DatagramPacket的方法支持输入和输出.

9.2.4 DatagramPacket构造函数

一个DatagramSocket通过下面的方法中的一个构造:

```
class DatagramPacket {
    DatagramPacket(byte[] data, int length);
    DatagramPacket(byte[] data, int length, InetAddress address, int port);
    DatagramPacket(byte[] data, int offset, int length);
    DatagramPacket(byte[] data, int offset, int length, InetAddress address, int port);
    DatagramPacket(byte[] data, int length, SocketAddress socketAddress);
    DatagramPacket(byte[] data, int offset, int length, SocketAddress socketAddress);
}
```

length和offset是非负数,下面的等式是适用的(*):

$$0 \le length + offset \le data.length \quad (EQ\ 9\text{-}1)$$

注意,不像TCP协议,在UDP中可能发送和接收都是0长度的数据报.这在一个应用程序协议中可能是有用的,比如,'I'm alive'消息或者pings.

data,offset,length,address和端口属性都可以单独设置在构造之后通过相应的set方法,它们也都可以使用相应的get方法获取.

这些属性在下面将单独描述.

---

* 这个等式在Java中总是适用的当处理字节数组的时候,offset和length,通过任意的构造函数或者方法

执行的时候都会执行.如果违背这个等式将抛出IllegalArgumentException.

9.2.5 DatagramPacket data

data在一个DatagramPacket被指定为一个字节数组,offset和length.这些可以在构造中一起设置,也可以随后通过下面的方法设置和获取.

```
class DatagramPacket {
    byte[] getData();
    void   setData(byte[] data);
    void   setData(byte[] data, int length);
    void   setData(byte[] data, int offset, int length);
    int    getLength();
    void   setLength(int length);
    int    getOffset();
    void   setOffset(int offset);
}
```

data,offset和length的默认值分别为null,0和0.当设置任意的这些属性的时候.等式9.1将强制执行.

当发送一个数据报,由应用程序决定格式化数据为字节数组.这样做的一种方式就是DataOutputStream,或者ObjectOutputStream,结合到ByteArrayOutputStream.

```
DatagramPacket  packet = new DatagramPacket();
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream  dos = new DataOutputStream(baos);
dos.writeInt(…);   // or as per application protocol
dos.flush();
packet.setData(baos.toByteArray(), baos.size());
```

当接收一个数据报,Java将接收到数据放置到DatagramPacket的字节数组中从当前的offset属性值开始,到DatagramPacket的当前的length属性的前一个结束.应用程序必须预分配字节数组,保证它能对于预期的数据是充分的.同时设置适当的offset和length:字节数字,offset,和长度必须满足等式9.1.通常应用程序将设置offset为0,length为字节数组的长度在接收到一个DatagramPacket之前.

Java调整DatagramPacket的length属性为接收到的数据的长度如果它小于原始的长度;然而,如我们在9.1.2节所看到的,超过的数据将被安静的截断,长度不会受到干扰:换句换说,没有迹象表明(如过度长度)接收数据被截断.

这意味着一个datagram的length属性在多次接收操作中重用,但必须在第二次和随后的接收之前重置;否则,它不断的(continually)缩减(shrinks)到目前为止最小的datagram的大小.它也意味着,应用程序不能区分一个数据报的精确的最大长度和数据报太大被截断了.通常处理这个问题的技术就是分配最大的预期的数据报加1个字节的空间.如果一个数据报的额外的长度接收到了,这是出乎意料的,也就是说,至少一个字节太长(FIXME: It also means that the application can't distinguish between a datagram which was exactly the maximum length and a datagram which was too big and got truncated. The usual technique for handling this problem is to allocate space for the largest expected datagram plus one byte. If a datagram of this extra length is received, it was unexpectedly large, i.e. at least one byte too long).

由应用程序决定格式化数据为字节数组.这样做的一种方式就是DataInputStream,或者ObjectInputStream,结合到ByteArrayInputStream.

```
DatagramPacket       packet;
// initialization & receive not shown …
ByteArrayInputStream bais = new ByteArrayInputStream(packet.getData(), packet.getOffset(),
packet.getLength());
DataInputStream      dis = new DataInputStream(bais);
int i = dis.readInt();// or as per application protocol
```

应该要清楚这个技术是这节之前给出的ByteArrayOutputputStream技术的倒转.

9.2.6 DatagramPacket 地址和端口

一个DatagramPacket关联一个IP地址和端口:这些代表了来自于哪里的一个绑定的datagram的远程的UDP套接字.或者一个未绑定的datagram正在被发送.它们可以通过构造函数或下面的方法设置:

```
class DatagramPacket {
    SocketAddress getSocketAddress();
    void          setSocketAddress(SocketAddress address);
    InetAddress    getAddress();
    void          setAddress(InetAddress address);
    int           getPort();
    void          setPort(int port);
}
```

要记住JDK 1.4 SocketAddress代表了一个{InetAddress, port}对.

当一个数据报被接收,Java设置它的{addressm port}到它的数据报的远程源(FIXME: remote source),这样你可以知道它从哪里来.

当发送一个数据报,它的{address, port}必须已经设置到了数据报将要被发送的目标.如果数据报启动会话(initiates the conversation),应用程序必须设置自身的{address, port}.然而,回复一个刚接收到的数据报,它只是简单的重用接收到的数据报:使用DatagramPacket.setData方法将回复的数据放到数据报中,保持{address, port}原样,而不是构造一个新的DatagramPacket和设置它的{address, port}.这种技术可以节省创建对象,在请求和回复之间避免易于出错的拷贝{address, port}.

9.3 数据报套接字初始化

9.3.1 构造

DatagramSocket使用下面的构造函数之一创建.

```
class DatagramSocket {
    DatagramSocket()         throws IOException;
    DatagramSocket(int port) throws IOException;
    DatagramSocket(int port, InetAddress localAddress) throws IOException;
    DatagramSocket(SocketAddress localSocketAddress) throws IOException;
}
```

在大多数这样的情况下,套接字构造已经'绑定',意味着它已经关联了一个本地的IP地址和端口.一个绑定的套接字可以立即被使用,用来发送和接收.然而,如果构造函数使用一个null的SocketAddress,套接字是'未绑定'构造的,也就是说,还没有关联一个本地的IP地址或者端口.一个未绑定的套接字可以立即用来发送,但是接收之前它必须首先使用DatagramSocket.bind来'绑定',在9.3.5节描述.

首先我们看下构造绑定的套接字的构造的函数;然后我们看下未绑定套接字的方法.

9.3.2 端口

UDP接收端通常指定它们要接收的本地端口,通过提供一个非0的端口到构造函数中或者bind方法.如果端口省略或者为0,一个临时-系统分配-端口将被使用.可以通过调用下面的方法获得:

```
class DatagramSocket {
    int getLocalPort();
}
```

临时端口通常被UDP客户端使用,除非他们需要去指定一个特殊的端口去满足本地的网络约束,比如防火墙策略.临时端口通常不会被UDP服务器使用, 由于一些外部因素通信需要实际的端口到客户端(FIXME: as otherwise some external means is required of communicating the actual port number to clients);否则我们不知道如何发送到接收器:通常这个功能假设通过命名服务比如一个LDAP目录服务.在Sun RPC,假设通过RPC 端口映射服务.

9.3.3 本地地址

一个数据报套接字的本地地址是它接收的IP地址.默认情况下,UDP套接字在所有的本地IP地址上接收.它们可以用来在一个单独的本地IP地址上接收,通过提供一个非null的InetAddress或者带有非null的InetAddress的InetSocketAddress,到一个DatagramSocket的构造函数上.

如果InetAddress省略或为null('通配符'地址),套接字绑定到所有的本地地址,意味着它从任意的地址接收.

指定一个本地的IP地址只是更有意义,如果本地主机是多地址的.也就是说,有多个IP地址,通常因为它有多于1个的物理网络接口.在这种的情况下,一个UDP接收器可能只想去通过这些IP地址中的1个使得自身可用而不是它们中的所有.查看在9.12节的多地址讨论获得更多细节.

一个数据报套接字绑定的UP地址通过以下地方方法返回:

```
class DatagramSocket {
    InetAddress getInetAddress();
    SocketAddress getLocalSocketAddress();
}
```

这些方法返回null,如果套接字还没有绑定.

9.3.4 重用本地地址

如早9.3.4节所描述的在绑定数据报套接字之前,你可能希望去设置'重用本地地址'选项.这个实际上意味着重用本地端口.

重用-地址方法在JDK 1.4中加入:

```
class DatagramSocket {
    void setReuseAddress(boolean reuse) throws SocketException;
    boolean getReuseAddress() throws SocketException;
}
```

你必须使用这个设置如果你想在相同的端口在多IP地址上接收通过绑定多个DatagramSocket到相同端口和不同的本地IP地址,要么在相同的JVM中或者在相同主机的多个JVM中.

在一个数据报套接字绑定,或者构造除了使用一个null SocketAddress后改变这个设置,不会有效果.注意这个方法设置和获取一个boolean状态,而不是一些某种形式的地址如它们的名称可能建议的(FIXME: not some sort of address as their names may suggest.).

9.3.5 Bind操作

一个DatagramSocket构造使用一个null SocketAddress在构造函数中在JDK 1.4中引入必须在数据报可以接受之前绑定(尽管它不需要为数据报被发送而绑定).数据报的绑定由JDK 1.4的方法支持.

```
class DatagramSocket {
    void bind(SocketAddress localSocketAddress) throws IOException;
    boolean isBound();
}
```

localSocketAddress通常是一个使用一个端口的InetSocketAddress,如9.3.2节所描述的.InetAddress如9.3.3节描述.套接字要么绑定到一个临时端口,要么绑定到一个指定的本地端口,在一个指定的本地IP地址或者它们中的所有,根据localSocketAddress参数的值,如表格9.1所示.

表格9.1 UDP bind 参数

SocketAdddress	InetAddres	port	Bound To
null	-	-	临时端口,所有的本地IP地址
InetSocketAddress	null	zero	临时端口,所有的本地IP地址
InetSocketAddress	null	non-zero	指定端口,所有的本地IP地址
InetSocketAddress	non-null	zero	临时端口,指定的本地IP地址
InetSocketAddress	non-null	non-zero	指定的端口和本地IP地址

如在9.3.2节所讨论的,一个服务器通常应该指定一个特殊的端口:客户端可以这样也可以不这样,主要依赖于它是否需要因为防火墙-穿透(firewall-traversal)目的使用一个固定的端口.bind方法必须在执行通道数据报输入之前被调用,将在10.2节讨论.多地址的主机在9.12节讨论.

一个DatagramSocket不能被重新绑定.isBound方法返回true如果套接字已经绑定.

9.3.6 设置缓冲区大小

在使用一个数据报套接字之前,它的发送和接收缓冲区大小应该被调整,如9.11节描述的.

9.3.7 Connect 操作

一个DatagramSocket可以连接到一个远程的UDP套接字.不像TCP的连接操作,一个UDP '连接'只是一个本地的操作:它不影响网络,因此它不会影响远程端;它仅仅(conditions)作为本地端的条件,这样数据可以接收或者发送到指定的地址.如果一个传入的DatagramPacket的{address,port}属性与设置指定到connect方法中的不一致,数据报将被安静地废弃;如果一个输出的DatagramPakcet的这些与指定到connect方法中的设置不一致,DatagramSocket.send将抛出IllegalArgumentException.

数据报套接字的连接通过下面的方法支持:

```
class DatagramSocket {
    void connect(InetAddress address, int port);
    void connect(SocketAddress socketAddress) throws SocketException;
    boolean isConnected();
}
```

(a) address和port指定了一个远程UDP套接字的{address, port},如在9.2.6节描述的,或者
(b) sokcetAddress通常是一个InetSocketAddress使用一个远程UDP套接字{address, port}构造,如9.2.6节所描述的.

由于没有涉及网络协议,不能保证远程UDP端口存在,即使connect方法可能已经成功完成.因此随后的在一个连接的DatagramSocket上的发送或者接收操作可能抛出PortUnreachableException.更坏的是,发送和接收可能简单的安静失败而不会抛出异常.实际的行为的发生依赖于平台和JDK版本.

连接一个数据报套接字节省Java开销:它允许连接允许到目标上只在连接的时候被检查,而不是在每次传输或者从目标接收.如果请求的'connct'行为的SocketPermission没有授予(granted),将抛出SecurityException:这是一个运行时错误,编译的时候不会被检查.

在Berkeley套接字API中,连接一个数据报套接字也减少了某些内核的开销.在JDK 1.4之前,Java不是调用底层的connect()API,它只是在Java层面模仿它的行为,这样内核开销不会减少,PortUnreachableException不是由在一个连接的DatagramSocket上的I/O操作抛出.

isConnect方法判断本地套接字是否已经连接：它不会告诉你关于另一个端点的信息,包含它是否存在.
一个DatagramSocket不能被关闭然后连接或者重新连接,但是它可以被断开连接如9.5节描述,然后重新连接.
如果你没有JDK 1.4,和它的DatagramSocket.isConnect方法,你可以使用下面的方法：

```
class DatagramSocket {
    InetAddress getInetAddress();
    int getPort();
}
```

为了相同的目的.这些方法返回连接的套接字的远程地址或断开,或者null或者-1(视情况而定),如果套接字当前没有连接.

9.4 数据报I/O

数据报I/O通过下面的方法执行：

```
class DatagramSocket {
    void send(DatagramPacket packet) throws IOException:
    void receive(DatagramPacket packet) throws IOException:
}
```

9.4.1 数据报输出

一旦你已经格式化输出数据到一个字节数组中,然后使用一个来自它的DatagramPacket构造,如9.2.5节所示,在数据报中设置地址,如9.2.6节所示,发送谁到目的地只是简单的调用DatagramSocket.send方法,如下面所示：

```
// initializations not shown
InetAddress address;
DatagramPacket packet;
int port;
byte[] buffer;// see section 9.2.5
DatagramSocket socket = new DatagramSocket();
packet.setData(buffer, length);

// pre-JDK 1.4 …
packet.setAddress(address);
packet.setPort(port);
socket.send(packet);
```

从JDK 1.4开始,setAddress和setPort的调用可以单独调用setSocketAddress来替代：

```
// From JDK 1.4 …
packet.setSocketAddress(new InetSocketAddress(address, port));
```

DatagramSocket.send方法不是同步的：这就是为什么在Example 9.2.1(书中写的是9.3,应该是错误的)中所展示的并发的UDP服务器不需要第二个DatagramSocket去发送回复给传入的数据包.任何内部的顺序需要由底层UDP实现处理.(这段说的其实有点让人不是那么理解,不过可以看下send()方法的源实现,虽然此方法不是同步的,但是此方法的内部,其实是做了同步操作的)
数据报套接字的异常处理在9.8节讨论.

9.4.2 数据报输入

接收数据报甚至比发送还简单,如下面所示：

```
int port; // initialization not shown
DatagramSocket socket = new DatagramSocket(port);
// declare array at required length+1
byte[] buffer = new byte[8192+1];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
socket.receive(packet);
```

如果数据报在等待,receive方法阻塞到一个数据报被接收到.此时packet.getData方法可以获得接收到的数据,packet.getOffset为它的偏移位,packet.getLength为它的总长度,所以我们可以如下处理：

```
ByteArrayInputStream bais = new ByteArrayInputStream(packet.getData(),
packet.getOffset(),packet.getLength());
// etc as in section 9.2.5
```

在这种情况下,我们不用指定offset当创建packet的时候,所有我们可以使用简单的形式：

```
new ByteArrayInputStream(packet.getData(), packet.getLength());
// …
```

尽管实际的编程中不应该像这样假设offset为0.
如果我们想去回复这个数据报,我们在9.2.6看到过,一个接受的数据报已经包含了发送者的地址和端口,所以我们所要做的就是设置回复的数据然后调用如9.4.1节描述的DatagrmSocket.send.
DatagramSocket.receive方法不是同步的.

默认的数据报套接字实现类(在9.6节讨论)的receive方法是同步的.从多线程中在同一个数据报套接字上并发接收是由Java进行顺序化的.

9.5 终止

在UDP中没有明确的断开连接协议,所以当一个会话由一个特定的端点结束,不需要特别的动作.连接的套接字数据报可以被断开,如果需要的话可以在未连接或者连接的模式中重用(FIXME:re-used in either unconnected or connected mode if required).数据报套接字比粗在使用后关闭.

9.5.1 断开连接

如果一个数据报套接字已经连接了,它可以使用下面的方法断开:

```
class DatagramSocket {
    void disconnect();
}
```

这个方法什么都不做如果套接字没有连接.
如果一个数据报套接字用在了连接模式中,它通常应该在一个会话的一开始就连接,在会话结束的时候断开.

9.5.2 关闭

当你用完了一个DatagramSocket,你必须关闭它:

```
class DatagramSocket {
    void close() throws IOException;
    boolean isClosed();
}
```

isClosed犯法判断本地套接字是否已经关闭.它不会告诉你任何关于连接的信息,(FIXME: for the very good reason that in UDP there is no connection in the network sense).
在关闭后,套接字不能再用来做任何事(FIXME:the socket is no longer usable for any purpose).

9.6 套接字工厂

如我们在3.8.1节所看到的,java.net套接字工厂由Java使用用来去提供它自身的套接字实现.
java.net.DatagramSocket类实际上是表面的.它定义了Java套接字API,但是代理了它(做所有的实际的工作)的所有的动作到套接字实现对象,数据报套接字实现继承了抽象java.net.DatagramSocketImpl类:

```
class DatagramSocketImpl {
    // …
}
```

工厂提供了DatagramSocketImpl类型的对象,实现了java.net.DatagramSocketImpFactory接口:

```
interface DatagramSocketImplFactory {
    DatagramSocketImpl createDatagramSocketImpl();
}
```

一个默认的套接字工厂总是被设置的,产生包保护的java.net.PlainDatagramSocketImpl类的对象.这个类有本地的方法,与本地的C-语言套接字API交互.比如,Berkeley套接字API或者Win sock.
套接字工厂可以这样设置:

```
class DatagramSocket {
    static void setDatagramSocketImplFactory(DatagramSocketImplFactory factory);
}
```

setDatagramSocketImplFactory(书中写的是setSocketFactory,是简写还是写错了?)方法在JVM的生命周期中只能被调用一次.它需要RuntimePermission 'setFactory'被授予,否则将抛出SecurityException.

应用程序很少或者无需使用这个工具.

9.8 许可(Permissions)

如果Java2安全管理被安装,数据报套接字操作需要各种java.net.SocketPermissions.这些在表9.2展示.

表9.2 UDP中的许可

动作	说明
accept	在DatagramSocket的receive和connect方法中需要.目标主机和端口指定了远程UDP套接字.
connect	在DatagramSocket的send和connect方法中需要,当获取InetAddress对象的时候.目标主机和端口指定了远程UDP套接字(书中多了and port字样).
listen	当构造DatagramSocket的时候需要,和在它的bind方法中.目标主机和端口指定了本地UDP套接字.大多数指定为如'localhost:1024'(FIXME: Most loosely specified as "localhost:1024-").
resolve	这个许可是'connect'操作必须(imply)的,所以有必要明确地指定它.目标主机和端口指定了远程UDP套接字.

```
public synchronized void receive(DatagramPacket p) throws IOExce
    synchronized (p) {
        if (!isBound())
            bind(new InetSocketAddress(0));
        if (connectState == ST_NOT_CONNECTED) {
            // check the address is ok with the security manager
            SecurityManager security = System.getSecurityManager
            if (security != null) {
                while(true) {
                    String peekAd = null;
                    int peekPort = 0;
                    // peek at the packet to see who it is from.
                    if (!oldImpl) {
                        // We can use the new peekData() API
                        DatagramPacket peekPacket = new Datagram
                        peekPort = getImpl().peekData(peekPacket
                        peekAd = peekPacket.getAddress().getHost
                    } else {
                        InetAddress adr = new InetAddress();
                        peekPort = getImpl().peek(adr);
                        peekAd = adr.getHostAddress();
                    }
                    try {
                        security.checkAccept(peekAd, peekPort);
```

```
public void checkAccept(String host, int port) {
    if (host == null) {
        throw new NullPointerException("host can't be null");
    }
    if (!host.startsWith("[") && host.indexOf(':') != -1) {
        host = "[" + host + "]";
    }
    checkPermission(new SocketPermission(host+":"+port,
        SecurityConstants.SOCKET_ACCEPT_ACTION));
}
```

9.8 异常

有意义的的Java异常可能出现哎数据报套接字操作中在表9.3展示.

表9.3 UDP中的异常

异常	由..抛出	含义
java.net.BindException	DatagramSocket构造函数和bind方法	请求的本地地址或者端口不能被分配,比如,它已经在使用中,并且'重用地址'选项没有设置
java.lang.IllegalArgumentException	DatagramSocket的几个构造和方法;DatagramSocket和InetSocketAddress的几个方法	违反了等式9.1,或者参数为null或者超出了范围.这个一个RuntimeException,因此没有在方法的签名中出现,或者被编译器检查.
java.nio.IllegalBlockingModeException	DatagramSocket的send和receive方法	套接字有一个管理的通道,处于阻塞模式中.从JDK1.4开始.
java.io.InterruptedIOException	DatagramSocket的receive方法	发生超时;在JDK 1.4之前.
java.io.IOException	DatagramSocket的send方法	出现了一般的I/O错误. 从这个异常衍生出的异常包括IllegalBlockingModeExcepion,InterruptedIOException,SocketException和UnkownHostException
java.net.PortUnreachableException	DatagramSocket的send和receice方法	套接字连接到一个当前不可达的目标.从JDK 1.4开始.
java.lang.SecurityException	DatagramSocket的几个方法	需要的SocketPermission没有被授予,如表格9.2所示.这是一个RuntimeExeption,没有在方法签名中展示或者由编译器检查.
java.net.SocketException	DatagramSocket的许多方法,尤其是receive	一个底层的UDP错误发生,或者套接字被另一线程通过DatagramChannel的close方法关闭. 许多异常从这个异常衍生,包括BindException.
java.net.SocketTimeoutException	DatagramSocket的	超时发生.从JDK 1.4开始.继承了InterruptedIOException为了

java.net.SocketTimeoutException	receive方法	了向后兼容JDK1.4之前的程序.
java.net.UnknownHostException	InetAddress的所有方法,或者不明确使用这些,当使用String主机名称的时候.	命名的IP地址主机不能决定(FIXME: The IP address of the named host cannot be determined).

9.9 套接字选项

有几个套接字选项可用,控制UDP套接字的高级特性.在Java中,数据报套接字选项可用通过java.net.DatagramSocket的方法设置和获取.

数据报套接字在下面展示,按照它们相对重要的顺序.在这章最后出现的一个选错,是你最不需要关注的.

9.10 超时

如我们已经在3.12节看到的(将在13.2.5和13.2.6节再次看到),不能假设一个应用程序可以永远等待一个远程的服务,或者服务将永远以一种即时的方式回答(rendered),或者服务或者中间的(intervening)网络设置将以可探测的方式失败(FIXME: the intervening network infrastructure will only fail in detectable ways).任何读取无限超时的网络程序尽快或者随后将经历一个无限的延迟.

因为这些原因,谨慎(prudent)的网络程序总是在客户端几乎使用一个有限的接收超时.接收超时通过下面的方法设置和获取:

```
class DatagramSocket {
    void setSoTimeout(int timeout) throws SocketException;
    int  getSoTimeout() throws SocketException;
}
```

timeout用毫秒指定,必须为一个正数,表明一个有限的超时,或者为0,表明一个无限的超时.默认情况下,读超时是无限的.

如果超时在套接字上一个阻塞的接收操作之前设置为一个有限的值,接收将阻塞达到超时周期如果数据不可用,然后将抛出InterruptedIOException,或者如表9.3所示,JDK 1.4 SocketTimeoutException从它衍生而来.如果超时是无限的,接收将永远阻塞,如果数据不可用并没有错误发生.

在3.12节关于TCP超时期间的特性同样适用于UDP.

UDP服务器等待一个客户端请求不需要使用超时,不像TCP服务器,它们没有连接去终止.一个UDP服务器超时在单个线程中可用来去轮询大量的DatagramSocket,尽管在5.3.1节描述的Selector类提供了一种更好的方式去这么做.

9.11 缓冲区

UDP为每个套接字分配发送缓冲区和接收缓冲区.这些缓冲区存在于内核等待地址空间或者UDP协议栈(如果不同的话),而不是在JVM或者进程地址空间.这些缓冲区的大小由底层平台的UDP实现决定,而不是Java.在当前的UDP实现中,发送和接收缓冲区大小至少为8KB,通常为32KB,或者更多,可以设置如256M这么大或者在一些实现中更多.在决定和如何去改变套接字缓冲区大小以前目标系统的特性必须被调查研究.

发送和接收缓冲区大小通过下面的方法设置和获取:

```
class DatagramSocket {
    void setReceiveBufferSize(int size) throws SocketException;
    int getReceiveBufferSize() throws SocketException;
    int getSendBufferSize() throws SocketException;
    void setSendBufferSize(int size) throws SocketException;
}
```

size指定为字节,提供给这些方法的值只是给底层平台一个示意,在任何一个方向都可以被调整来适应合法的范围,或者向上或者向下调整为适当的边界.这些方法的返回值可能和你发送的值不匹配,也可能与底层平台使用的实际值不匹配.

你可以在套接字关闭前的任意时刻执行这些操作.

9.11.1 一个数据报套接字的缓冲区应该多大?