这一章讨论了TCP/IP协议的基础和它在阻塞模式中Java Socket和ServerSocket对象的实现.这一章假设对TCP/IP的基本概念和Java套接字有一个认识,虽然提供了一个简要的回顾.

TCP Channel I/O和非阻塞模式在第5章讨论.

3.1 基本的TCP Sockets

在这节我们简要回顾基本的TCP/IP的Sockets和如何在Java中编程.

3.1.1 TCP总结

TCP/提供了在一个客户端-服务器端架构的端点之间可靠的双向的流的连接.一个TCP端点定义为{IP address, port}对,代表了TCP编程中的接口,作为一个TCP套接字,如2.2.5节中的定义.

通过流,意味着数据传输和接收被当做一个持续的字节流,没有消息边界.

有两种类型的TCP socket: "主动(active)"和"被动(passive)"(通常被称为"监听").一个TCP服务器创建一个TCP套接字;绑定到一个端口;使它进入"监听"状态;然后循环"接收"客户端的连接.客户端创建一个主动的TCP套接字,然后"连接"到服务器的端口.服务器"接收"这个连接请求,在这个过程中接收一个新的活动的socket代表它的连接端.现在服务器端和客户端连接上了,可以可靠地发送数据给另一方任意数量的数据,如果有必要两边同时.数据通过这个连接被完整的传递和以正确的顺序被发送,作为一个数据流而不是不同的消息.

TCP连接过程如图3.1.

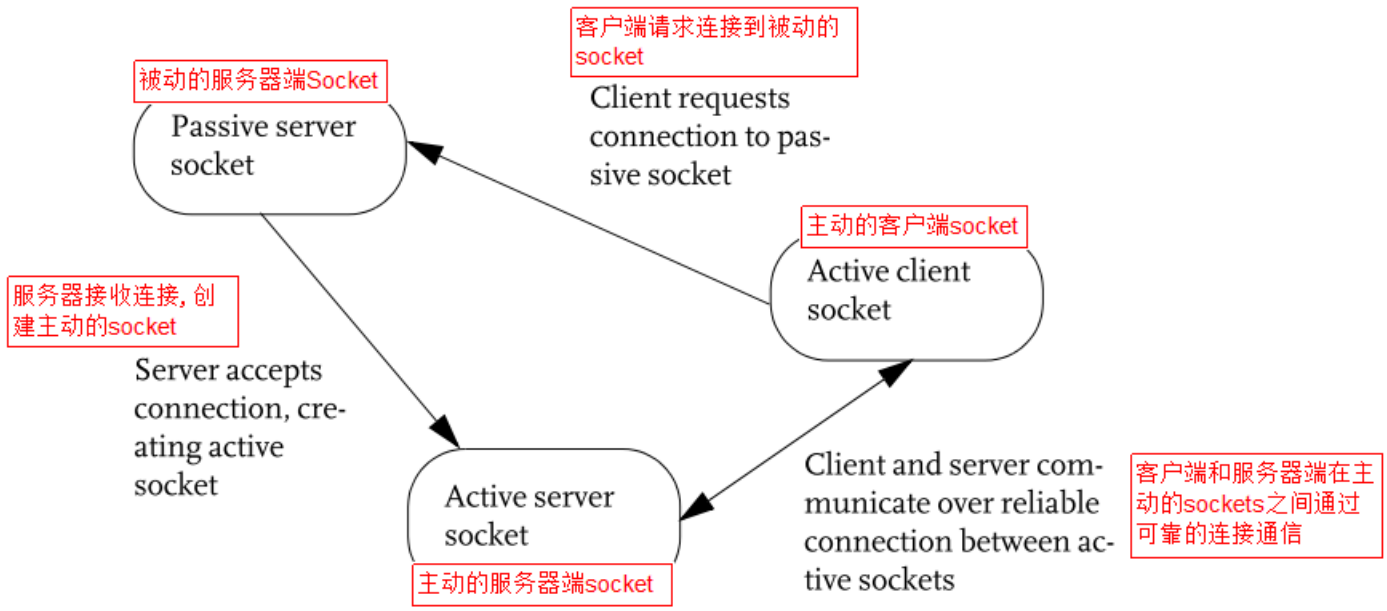

FIGURE 3.1. TCP client and server connection processing

产生的连接由{本地端口, 本地地址, 远程端口, 远程地址}对定义.每个TCP段包含了这个对,保证了它传递到正确的端点.

在这一节提供的材料在随后的章节阐明了TCP选项.TCP服务器和客户端的更多高级的架构在第12章讨论.

3.1.2 导入语句

下面的Java导入语句假设在这章的例子中至始至终存在.

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

```
import java.util.*;
```

3.1.3 在Java中的简单的TCP服务器

在Java中,一个服务器端的被动的套接字由java.net.ServerSocket表示.一个TCP服务器构造一个java.net.ServerSocket,然后循环调用ServerSocket.accept.循环的每次迭代返回一个java.net.Socket代表一个已经接收的连接.

一个最简单的可行TCP服务器处理每个连接在接受一个新的之前,如Example3.1描述:

```
public class TCPServer implements Runnable {
    private ServerSocket serverSocket;
    // constructor
    public TCPServer(int port) throws IOException {
        this.serverSocket = new ServerSocket(port);
    }
    public void run() {
        for (;;) {
            try {
                Socket socket = serverSocket.accept();
                // 注意：这里不是以线程的方式启动的
                new ConnectionHandler(socket).run();
            } catch (IOException e) { /* … */
            }
        } // end finally
    } // end run()
} // end class
```

Example 3.1 单线程的TCP服务器

这个类的连接处理类和随后的服务在Example3.2中展示.

```
public class ConnectionHandler implements Runnable {
    private Socket socket;
    public ConnectionHandler(Socket socket) {
        this.socket = socket;
    }
    public void run() {
        handleConversation(socket);
    }
    /**
     * @param socket
     *            Socket: must be closed on exit
     */
    public void handleConversation(Socket socket) {
        try {
            InputStream in = socket.getInputStream();
```

```
            // read request from the input:(读取请求)
            // conversation not shown … (交互没有展示)
            OutputStream out = socket.getOutputStream();
            // write reply to the output(回复)
            out.flush();
        } catch (IOException e) { /* … */
        } finally {
            try {
                socket.close();
            } catch (IOException e) {
            }
        } // end finally
    } // end run()
}
```

Example 3.2 TCP服务器连接处理器

Example 3.1的单线程设计不太适用,因为它顺序地处理客户端,而不是并发-一个新的客户端将阻塞当先前的客户端正在服务.为了解决客户端并发,服务器必为每个接收的连接使用一个不同的线程.这样的TCP服务器的最简单的形式,使用同样的连接-处理类,在Example 3.3描述.

```
public class TCPServer implements Runnable {
    private ServerSocket serverSocket;
    // constructor
    public TCPServer(int port) throws IOException {
        this.serverSocket = new ServerSocket(port);
    }
    public void run() {
        for (;;) {
            try {
                Socket socket = serverSocket.accept();
                // 以线程的方式启动
                new Thread(new ConnectionHandler(socket)).start();
            } catch (IOException e) { /* … */
            }
        } // end finally
    } // end run()
}
```

Example 3.3 简单的TCP服务器-多线程

一个连接-处理类简单地回复它的输入到它的输出,对于测试非常有用,在Example 3.4中展示.

```
public class EchoConnectionHandler extends ConnectionHandler {
    public EchoConnectionHandler(Socket socket) {
        super(socket);
    }
    /**
```

```
     * @param socket
     *            Socket: must be closed on exit
     */
    public void handleConversation(Socket socket) {
        try {
            InputStream in = socket.getInputStream();
            OutputStream out = socket.getOutputStream();
            // read requests from the input until EOF(读取请求直到EOF)
            byte[] buffer = new byte[8192];
            int count;
            while ((count = in.read(buffer)) >= 0) {
                // echo input to the output(回复)
                out.write(buffer, 0, count);
                out.flush();
            } // loop terminates at EOF
        } catch (IOException e) {
            /* … */
        } finally {
            try {
                socket.close();
            } catch (IOException e) {
            }
        } // end finally
    } // end run()
}
```

Example 3.4 TCP服务器连接处理器-echo

3.1.4 Java中的简单的TCP客户端

在Java中,连接的客户端的端点由java.net.Socket表示,通常构造已经连接到服务器的端口.一个典型的TCP客户端在Example 3.5中描述.

```
public class TCPClient implements Runnable {
    Socket socket;
    @Override
    public void run() {
        try {
            socket = new Socket(host, port);
            OutputStream out = socket.getOutputStream();
            // write request, not shown …(写入请求)
            out.flush();
            InputStream in = socket.getInputStream();
            // get reply …(取得回复)
        } catch (IOException e) { /* … */
        } finally
        // ensure socket is closed
        {
```

```
            try {
                if (socket != null)
                    socket.close();
            } catch (IOException e) {
            }
        } // end finally
    } // end run()O
}
```

Example 3.5 TCP客户端

3.2 TCP的功能和成本(cost)

正如我们上面看到的,TCP实现了一个双向的可靠的数据流,在任意一方任意的大量的数据可被传输,或者两边同时.

3.2.1 功能

在TCP中,数据接收是自动确认的,有序的,必要的话会重发.应用程序不能接收损坏(corrupt)的或者无序的数据,或者是数据"空洞".

传输是自动地一步一步增长网络间的容量(FIXME: Transmissions are automatically paced to the capacity of the intervening network).如果没有确认,有必要的话会重新传输.

所有可用的带宽被使用而不会使网络饱和(saturating)或者对其它网络用户不公平.

TCP迅速和可靠地调整去改变网络条件-根据不同的负载和路由.

TCP实现了"协商(negotiation)连接"来保证一个服务器启动和运行,服务器主机已经接收到了一个客户端请求,在客户端连接请求完成之前.

TCP实现了"协商关闭"来保证所有在传输中的数据已经被传输了,在连接最终丢弃之前接收到了.

3.2.2 成本

所有这些功能都有相应的成本.有计算开销,协议开销和时间开销:

(a) 连接协商由三方的数据包交换组成

客户端发送一个SYN;服务器回复一个SYN/ACK;然后客户端回复一个ACK;如果第一个SYN没有产生回复,它将以增长的时间间隔重试(重新发送).第一次重试间隔是实现依赖的,通常是3到6秒,每次失败的话至少双倍时间后重试.尝试去连接的总共花费的时间也是实现依赖的,通常限制在75秒或者3次重试.因此,总的来说,一个连接完全失败的连接尝试时间通常可能是: 6 + 12 + 24 = 42秒.

(b) 关闭协商由四方的数据包交换组成

每一端发送一个FIN,然后使用一个ACK回复给传入的FIN.

(c) 数据序列,确认和(FIXME: pacing)确实需要相当多的计算,包含了维护一个在两个端点之间当前数据包的移动的来回时间的平滑估量的统计(FIXME:which includes maintaining a statistically smoothed estimator of the current round-trip time for a packet travelling between the two endpoints).

(d) (**FIXME**: The provisions for congestion avoidance require an exponentially increasing retry timer on retransmissions (‘exponential backoff’) and a slow start to the transmission: this implies that the first few packets are generally

exchanged at a sub-optimal speed, although the speed increases exponentially to the maximum feasible)

3.2.3 TCP和请求-应答交易

TCP是为批量数据传输设计的.一个简单的请求-应答交易,理论上只需要在每个方向上只发送一个IP数据包,系统的总效率不是非常高,因为实际上至少有9个TCP片段交换,如Figure 3.2所展示的序列图.

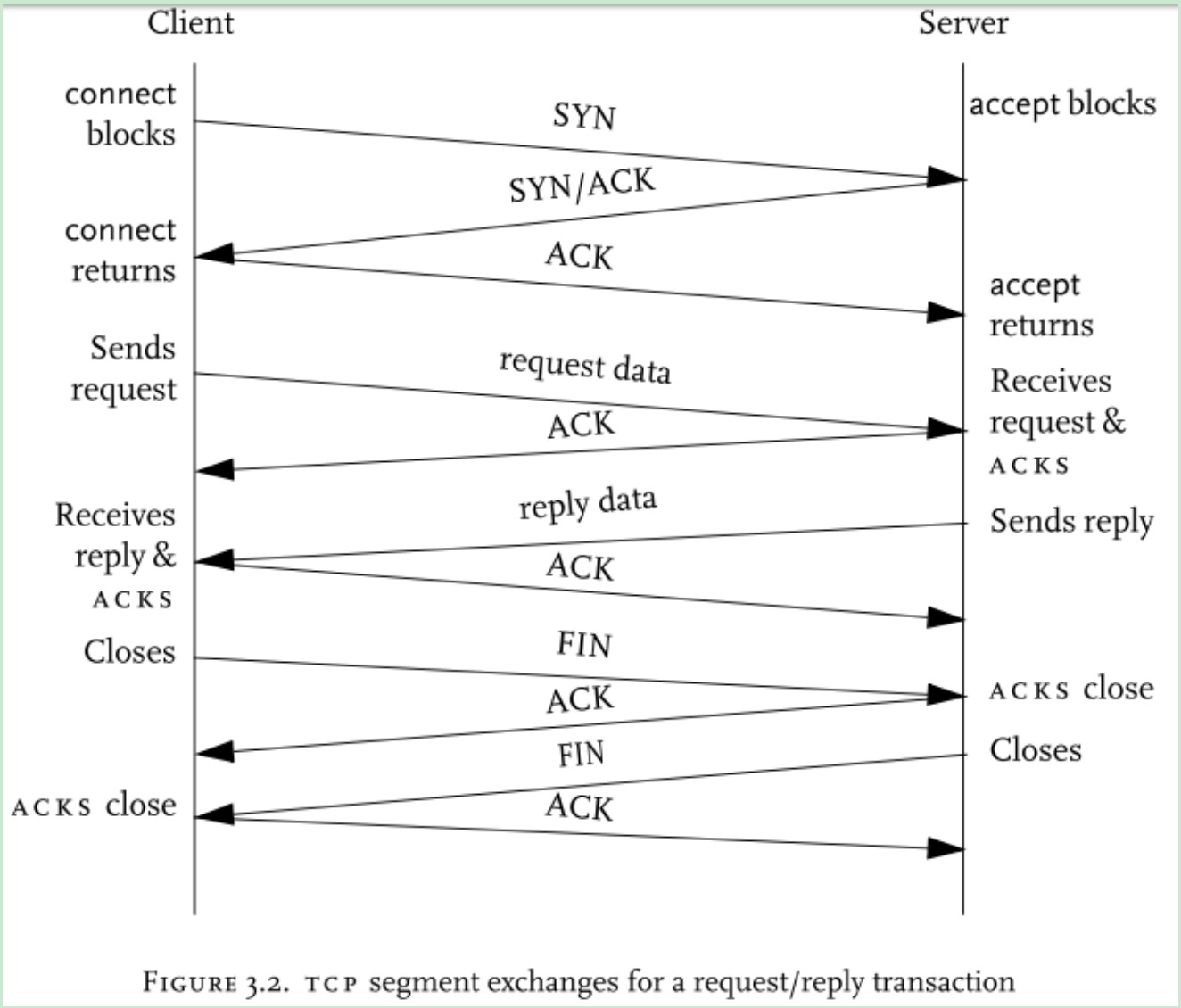

FIGURE 3.2. TCP segment exchanges for a request/reply transaction

在每个方向上的包是一步一步的,受限于(subject to)'缓慢启动'的需求.除了确认数据包.(FIXME: The packets in each direction are paced and are subject to the requirement for slow start, with the exception of acknowledgement packets.)

3.3 Socket初始化-服务器端

在这节,我们将查看所有可以采取的步骤和当初始化一个ServerSocket时可以设置的所有可能的参数.

3.3.1 构造

ServerSocket对象使用以下4个构造中的一个创建.

```
class ServerSocket {
    ServerSocket(int port) throws IOException;
```

```
    ServerSocket(int port, int backlog)throws IOException;
    ServerSocket(int port, int backlog, InetAddress localAddress) throws IOException;
    ServerSocket() throws IOException;
}
```

前三个构造函数创建一个已经"绑定"的服务器端套接字.一个绑定的套接字准备好使用-准备好ServerSocket.accept()被调用.默认的构造函数在JDK 1.4被引入,创建一个"未绑定"状态的服务器端套接字.一个未绑定的套接字必须使用ServerSocket.bind()去绑定在ServerSocket.accept()可以被调用之前.这个方法将在3.3.7描述.

首先我们看下构造已经-绑定的套接字的参数;然后我们看下绑定未绑定的套接字的方法.

3.3.2 端口

TCP服务器通常指定本地端口来监听连接,提供一个非零的端口数.如果端口数为0,系统分配一个端口来使用,这个值可以调用以下的方法来获取.

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

如果使用这种技术,需要一些外部的方法让实际的端口与客户端通信;另外,客户端不知道如何连接到服务器.通常这个功能更假设通过一个命名服务,比如LDAP(Lightweight Directory Access Protocol, 轻量级目录访问协议).在Java RMI中,这个功能假定使用RMI注册.在Sun RPC(Remote Procedure Call),假定使用端口映射服务.

本地端口通被一个接收的连接使用,也就是说,通过ServerSocket.accept()产生一个Socket,通过以下的方法返回.

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

这个总是和服务器套接字监听的端口相同.这个是客户端需要连接的端口,所以没有其它的可能性(*).

使用一个'众所周知'的端口,也就是说,端口范围在1-1023,在ServerSocket中可能需要特殊的权限,比如,像Unix -like中的超级用户权限.

---

* Sun的在线Java教程(Customer Networking/All About Sockets/What is a Socket?)在这点上已经错误很多年了.

3.3.3 Backlog

TCP本身在接收连接中可以在一个TCP服务器应用程序之前获得连接.它维护一个连接到一个监听的套接字的"backlog 队列",TCP自身已经完成,但是还没被应用程序接受(*).这个队列存在于底层的TCP实现和服务器创建的监听套接字进程之间.预完成连接的目的是可以加速连接阶段,但是队列是有长度限制的,以免预申请太多的连接到服务器,可能会不接受它们在它们以相同的速率而没有任何理由(FIXNE: so as not to pre-form too many connections to servers which are not accepting them at the same rate for any reason).当一个进来的连接请求已经接收,backlog队列还没满,TCP完成连接协议,然后把连接加入到backlog队列.同时,客户端应用程序已经完全连接,但是服务器应用还没有完全接收到ServerSocket.accept()产生的结果的连接.当它这么做的时候,实体从队列中移除.

backlog参数指定了backlog队列的最大长度.如果backlog省略,负数或者为0,系统选择的默认值将被使用.比如50.指定的backlog可以被底层的平台调整.如果backlog值对平台来说过多,平台将默默的调整到一个合法的值.在Java或者Berkely套接字API中没有方法去知晓有效的backlog值.

一个非常小的backlog值,比如1,可能是故意(deliberately)用来"削弱(cripple)"一个服务器应用,比如,为了产品演示(demonstration)目的,如果底层实现不调整向上,服务器仍然能正常工作,但是它处理并发客户端的能力被严重(severely)限制了.

---

* 这个定义一直随时间变化.它用于包括连接仍然形成,也就是说,这些SYN已经接收到,但是发送完成ACK还没有被接收到.

** P29

3.3.4 本地地址

一个服务器套接字的本地地址是监听传入的连接的IP地址.默认情况下,TCP服务器监听所有的本地的IP地址.它们可以去监听单个的本地IP地址,通过提供一个非null的loaclAddress到构造中.如果地址省略或者为null,套接字绑定到所有的本地地址.

指定一个本地的IP地址只是更有有意义,如果本机是多地址的,也就是说,有1个以上的IP地址,通常因为它有多于1个的物理网络接口.在这种情况下,一个服务器可能只想去通过一个IP地址而不是所有使它自身可用.看3.14多地址的讨论获取更多的细节.

本地IP地址由一个监听的服务器套接字的以下方法返回.

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

这些方法返回null,如果socket未绑定,如3.3.1和3.3.7节描述的(*).这种情况在JDK 1.4之前是不可能的,因为未绑定的ServerSocket不能构造,默认构造在JDK 1.4中才被加入.

---

* ServerSocket.getInetAddress在所有1.4.1之前的JDK版本中的文档都是不正确的, 返回'null如果套接字还没有连接'.但是ServerSocket对象从不连接.

3.3.5 重用本地地址

在3.3.7节描述的在绑定服务器socket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.

重用地址的方法在JDK 1.4中加入:

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

这个设置在开发中是有用的,当服务器频繁停止和启动.默认情况下,TCP阻止一个监听的端口重用当有一个

活动的,更通常来说,一个正在关闭端口的连接.关闭连接持续大约2分钟,因为一些协议完整性原因.在开发环境中,两分钟的等待可能是浪费的和令人烦恼(annoyance)的.设置这个选项,停止了浪费和减轻烦恼.

在服务器socket绑定之后改变这个设置,或者使用一个不是默认的构造函数构造,这个行为是无效的(必须在未绑定到一个端口前设置).

注意:这些方法设置和获取一个boolean状态,而不是某些"reuse-address"方式如它们名字所建议的.

Java没有定义这个设置的默认值,但在MacOS/X,根据http://lists.apple.com/archives/java-dev/2004/Dec/msgoo570.html ,它是true.所有的其它系统,我遇过的都是false.

3.3.6 设置接收的缓冲区大小

如3.3.7节描述的绑定服务器socket之前,你可能希望设置接收的缓冲区大小.你必须在绑定之前做这个,如果你想达到最大的吞吐量,使用一个大的接收缓冲区(大于64KB),因为一个大的缓冲区是有用的,如果发送端知道它,接收端只能通知缓冲区大于64KB,如果它允许window在连接顺序之间测量(FIXME: if it enables window scaling during the connection sequence).第一次可以出现在套接字绑定,也就是说,在ServerSocket.accept()返回之前.

因此,你必须在绑定一个服务器套接字之前设置接收缓存大小(有点歧义,应该是在设置大于64KB的缓冲区的时候才需要这样做).通过ServerSocket.accept()返回的Sockets继承这个设置(实际上所有的套接字选项设置都这样).

你可以设置一个大的接收缓冲区大小在服务器sokcet上在它绑定之后或者使用非有参的构造函数构造,但它在已经接收的连接上不会有期望的效果.你也可以在一个已经接收的socket上设置,但是这是无效的.

设置一个大的发送缓冲区大小在一个已经接收的Socket上不会有预期的效果.因为大的发送缓冲区大小不会通知给其它的终端.因此,ServerSocket.setSendBufferSize()方法是不需要的或者提供的.

接收的缓冲区大小设置和获取通过以下方法:

```
class ServerSocket {
    void setReceiveBufferSize(int size) throws SocketException;
    int  getReceiveBufferSize() throws SocketException;
}
```

查看3.13节获取socket缓冲区大小的更多讨论.

3.3.7 绑定操作

在JDK 1.4引入的默认构造函数产生一个ServerSocket必须在连接可以被接收之前绑定.这个使用以下JDK 1.4的方法:

```
class ServerSocket {
    void    bind(SocketAddress address) throws IOException;
    void    bind(SocketAddress address, int backlog) throws IOException;
    boolean  isBound();
}
```

address通常是一个使用一个端口构造的InetSocketAddress,如3.3.2节描述,一个localAddress,如3.3.4节描述,backlog如3.3.3节描述.

在一个ServerSocket关闭之后,它不能被重用,所有它不能再次被绑定.

在Berkeley Sockets API中,ServerSocket.bind()方法合并了bind()和listen()两个功能.

3.4 Socket初始化-客户端

3.4.1 构造

客户端的Socket使用以下的构造中的一个创建.

```
class Socket {
    Socket(InetAddress host, int port) throws IOException;
    Socket(String host, int port) throws IOException;
    Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws
IOException;
    Socket(String host, int port, InetAddress localAddress, int localPort) throws
IOException;
    Socket() throws IOException;
    Socket(Proxy proxy) throws IOException;
}
```

前四个创建已经连接到指定目标的Socket.一个连接的Socket可以准备使用-I/O操作.

默认构造在JDK 1.4中引入创建一个未连接状态的socket.一个未连接的socket必须使用在3.4节描述的Socket.connect()连接到一个目标在它用来进行任何I/O操作之前.

列表上的最后一个构造,在JDK 1.5中引入,连接一个到本地代理服务器的套接字;在构造这样的一个socket后你必须调用Socket.connect()连接到通过代理服务器到真实的目标.

3.4.2 远程主机

host参数指定了要连接的远程主机.它可以指定为一个InetAddress或者一个String.

一个InetAddrss可以调用InetAddress.getByName或者InetAddress.getByAddress构造.一个String的host可能包含了一个主机名比如"java.sun.com",使用命名服务比如DNS解析,或者一个代表它的IP地址的文本.对于文本表示,只有地址格式的校验会被检查.IPv4是一个众所周知的"点分四组"格式,比如"192.168.1.24".对于IPv6,在RFC 2372中的文本IPv6地址格式的任意地址可以被接收,比如"1080::8:800:200C:417A"或者"::192.168.1.24".

远程主机可以使用以下方法获得:

```
class Socket {
    InetAddress getInetAddress();
}
```

如果socket没有连接返回null.远程地址也可以通过下面的JDK 1.4的代码顺序获得.

```
SocketAddress sa = socket.getRemoteSocketAddress();
if (sa != null)
    return ((InetSocketAddress)sa).getAddress();
```

```
return null;
```

3.4.3 远程端口

port参数指定了要连接的远程端口,也就是说,这就是服务器监听的端口,在3.3.2节中描述.

远程端口可以通过下面的方法获得:

```
class Socket {
    int getPort():
}
```

如果套接字没有连接,返回0.远程端口也可以通过下面的JDK 1.4代码顺序获得,如果套接字没有连接,也返回0.

```
SocketAddress sa = socket.getRemoteSocketAddress();
if (sa != null)
    return ((InetSocketAddress)sa).getPort();
return 0;
```

3.4.4 本地地址

localAddress参数指定了本地的IP地址.如果缺省或者为null,将由系统选择.在为一个TCP客户端指定本地IP地址有一点:它不完全这样做,只有在多宿主的主机中.(FIXME: It might be done to force the connection to go via a network interface known to be faster than the others),或者因为某些原因预决定IP路由器.

一个套接字绑定的本地IP地址可以通过下面的方法获得:

```
class Socket {
    InetAddress getLocalAddress();
}
```

如果套接字没有连接,返回null(XXX:从这里的上下文来看,这里说的返回null应该的对于上面getLocalAddress()而言的,而返回null这个描述是有误的).无论怎样,它对TCP客户端没有什么实用性.本地地址也可以通过下面的JDK 1.4代码顺序获得:

```
SocketAddress sa = socket.getLocalSocketAddress();
if (sa != null)
    return ((InetSocketAddress)sa).getAddress();
return null;
```

这些方法也工作于在一个服务器中的已经接收的套接字.结果可用于在多宿主主机中的TCP服务器.查看3.14节的多宿主的讨论.

3.4.5 本地端口

localPort参数指定了本地端口到绑定的套接字.如果缺省或者为null,将由系统分配.在为一个TCP客户端指定本地端口有一点,这个操作很少使用.

本地端口数到一个绑定的套接字可通过下面的方法获得:

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

如果套接字没有连接返回0(XXX：描述有误,和上面本地地址一样).端口数也可以通过下面的JDK 1.4的代码顺序获得,如果套接字没有连接也返回0:

```
SocketAddress sa = socket.getLocalSocketAddress();
if (sa != null)
    return ((InetSocketAddress)sa).getPort();
return 0;
```

这个信息对于TCP客户端没有什么实际的使用性.这个方法也工作于在一个服务器中接收的套接字,尽管结果总是和监听的服务器的端口一样,如3.3.2节讨论的.

3.4.6 代理对象

代理对象指定了代理的类型(Direct,Sock,HTTP)和它的SocketAddress.

3.4.7 设置接收缓冲区大小

如3.4.10节所描述的在接收套接字连接之前,你可能希望去设置接收的缓冲区大小.接收的缓冲区大小通过以下的方法设置和获取:

```
class Socket {
    void setReceiveBufferSize(int size) throws SocketException;
    int getReceiveBufferSize() throws SocketException;
}
```

你必须在连接之前设置接收缓冲区大小如果你想去使用一个大(≥ 64KB)的缓冲区和你想获得最大的吞吐量.你也仍然可以在套接字连接之后设置接收缓冲区大小,但是它不会有期望的效果,如3.3.6节所讨论的.在3.3.6节也讨论了,在一个连接的套接字上设置一个大的发送缓冲区大小会有预期的效果,因为大的发送缓冲区不需要去通知另一端.因此,你可以在套接字关闭之前的任意时刻设置发送缓冲区大小.

查看3.13节获得套接字缓冲区大小的进一步讨论.

3.4.8 绑定操作

一个从默认构造中产生的套接字在JDK 1.4中被引入,可以在连接之前'绑定'.这个等同于在构造中指定一个或者localAddress和localPort两者,在3.4.1节描述.这个可以通过JDK 1.4方法达成:

```
class Socket {
    void bind(SocketAddress address) throws IOException;
    boolean isBound();
}
```

address使用一个localAddress构造如3.4.4节描述的.端口数如3.4.5节描述的.

如在3.3.4和3.4.5节描述的,这个操作很少使用.

3.4.9 重用本地地址

如3.4.8节描述的在绑定套接字之前,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.

重用地址的方法在JDK 1.4中被加入:

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

像客户端套接字绑定操作自身一样,这个操作基本上完全没有意义(pointless)和几乎从不使用的.

3.4.10 连接操作

一个从默认构造函数产生的Socket在JDK 1.4中引入,或者从JDK 1.5中引入的Proxy构造产生.在它能为I/O使用之前必须连接.这个可以通过JDK 1.4方法中的一种达成:

```
class Socket {
    void connect(SocketAddress address) throws IOException;
    void connect(SocketAddress address, int timeout) throws IOException;
    boolean isConnected();
}
```

address通常是一个使用一个如3.4.2节描述的remoteHost和一个如3.4.3节描述的remotePort构造的InetSocketAddress,timeout指定了连接超时的毫秒数:如果为0或者缺省,则使用无限的超时时间:这个操作阻塞直到连接建立或者有错误发生.

connect方法可以在失败前等待timeout毫秒,但是它可以更快的失败(如果有主机,但端口没有监听,主机可以立即生成一个TCP RST).通常,超时周期将被耗尽如果服务器端的backlog队列(在3.3.3节描述)满的时候.

isConnect方法判断本地的套接字是否已经连接上.这个方法不会告诉你另一端是否关闭了连接.

一个套接字不能被关闭然后重连接.

3.5 接收客户端连接

一旦一个服务器端套接字被构造和绑定,客户端连接通过下面的方法接收:

```
class ServerSocket {
    Socket accept() throws IOException;
}
```

这个方法返回一个准备好I/O的连接的套接字.连接的套接字继承了服务器端套接字的许多设置.特别是包括了它的本地端口,发送和接收的缓冲区大小(XXX:描述有误),阻塞/非阻塞状态,但是不包括读的超时时间(XXX:描述不准确).

另一个没有继承的设置是本地地址.通过Socket.getLocalAddres或者Socket.getLocalSocketAddress返回的值是客户端用来连接到服务器的地址.这个在多地址主机中是重要的:见3.14节.

服务器需要去构造为了尽可能的频繁循环调用ServerSocket.accept,为了不拖延连接中的客户端.各种

体系架构都是可能的.接收的套接字通常传递到另一个线程去处理当接收的线程再次循环,如示例3.3中展示的最简单的可用的架构.更多高级的服务器架构在第12章讨论.

这个循环应该被编码,这样它不会拖延任何地方,只在ServerSocket.accept.在accept和派发到另一个线程之间这个通常排除做任何的I/O操作,然而后者是被管理的.这个影响了应用协议的设计:它应该不需要从客户端读取任何东西在派发连接到它自己的线程之前.

3.7 套接字I/O

3.6.1 输出

在Java中,输出到一个套接字是通过从套接字中Socket.getOutputStream获得一个OutputStream来完成,如下面所示,或通过在第5章讨论的高性能的套接字管道.这节讨论输出流.

```
Socket  socket; // initialization not shown
OutputStream  out = socket.getOutputStream();
byte [] buffer   = new byte [ 8192 ];
int offset = 0 ;
int count = buffer.length;
out.write(buffer,offset,count);
```

对于本地发送缓冲区而言,在一个TCP套接字上的所有输出操作都是同步的,对于网络和远程应用程序而言是异步的.(FIXME: All that a output operation does is buffer the data locally to be sent according to the timing and pacing rules of TCP).如果本地套接字发送缓冲区是满的,套接字的一次写操作通常延迟直到发送缓冲区空间被释放,作为之前传输的确认接收的结果.只要有足够的本地缓冲区空间可用,控制回到应用程序.如果缓冲空间可用于数据的一部分,这部分数据被缓冲,应用程序延迟直到进一步的空间出现;这个持续直到所有的数据被写入到缓冲中.很明显这个意味着如果数据的数量超过了发送缓冲区大小,开始超过的(initial excess)将被写入到网络中,只有最后未超过的部分数据被本地缓冲,当write方法返回.

这意味着,在上面的输出示例中,当out.write返回,所有的count字节被写入到本地发送缓冲区.

这是Java和其它的套接字实现比如Berkeley Sockets或winsock之间的一个差异点.在Java流I/O中,write方法阻塞直到所有的数据被处理.其它的阻塞模式套接字-write实现返回一个count,至少为1,但是可能小于发送的数目;唯一的保证是某些数据已经被缓冲.

在写入到一个套接字之后,不能保证所有的数据已经被应用程序(或TCP)的另一端接收.一个应用程序可以确定数据传输已经达到远程应用程序的唯一方式是:通过远程应用发送(send)明确地接收到一个确认.通常,这样的一个确定内置在了应用程序间的协议,并通过TCP传递.换句话说,大部分的TCP会话遵循一个请求-回复(request-reply)模型.

没有更多的保证数据被写入到了一个套接字,被发送到了网络;同样也没有任何保证之前的write操作已经接收或者发送.你可以通过写入的字节的总量减去发送缓冲区大小明确地计算有多少数据被发送到了网络,但是这个仍然不能告诉你数据是否已经被接收到,所以这个没有什么意义.

最好附着(包装)一个BufferedOutputStream到从套接字中获取的输出流.理论上,BufferedOutputStream的缓冲应该是传输的最大请求或者回复,如果这个预先知道并且不会没有原因的

变大;否则它应该至少和套接字的发送缓冲区一样大;这个最小化了内核的上下文切换,它使得TCP一次写入更多的数据,允许它形成更大的分段和更有效地使用网络.同样它最小化了JVM和JNI之间的来回切换.你必须在合适的时候刷新(flush)缓冲区,也就是说,在完成了一次请求信息的写操作,和在读取回复之前,保证在BufferedOutputStream缓冲中的所有的数据是从套接字获取的.

为了发送Java数据类型,直接在套接字输出流中附着一个DataOutputStream,或者最后,如上所以附着在BufferedOutputStream.

```
DataOutput dos = new DataOutputStream(out);
// examples …
dos.writeBoolean(…);
dos.writeByte(…);
dos.writeChar(…);
dos.writeDouble(…);
dos.writeFloat(…);
dos.writeLong(…);
dos.writeShort(…);
dos.writeUTF(…);// write a String
```

为了发送序列化Java对象,在你的输出流中包装一个ObjectOutputStream.

```
ObjectOutput oos = new ObjectOutputStream(out);
// example …
Object object;// initialization not shown
oos.writeObject(object);
```

由于ObjectOutputStream继承了DataOutputStream,你可以使用上面的数据类型方法.然后要注意ObjectOutputStream加入了自己的协议到数据流中,所以其它终端你只能使用ObjectInputStream.你可以使用ObjectOutputStream写入数据类型,然后用DataInputStream读取它们.

如上面对DataOutputStream的建议,你应该使用BufferedOutputStream协同一个ObjectOutputStream.

3.6.2 对象流死锁(Object stream deadlock)

注意使用对象输出和输入流的一个死锁问题.下面的代码片段总是死锁如果同时出现在客户端和服务器端:

```
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
```

3.6.3 输入

类似地,一个套接字的输入流通过Socket.getInputStream获取一个输入流,如下所示,或者通过高性能套接字通道,在第5章讨论.这节讨论输入流.

```
Socket socket;// initialization not shown
InputStream in = socket.getInputStream();
byte[] buffer = new byte[8192];
```

```
int offset = 0;
int size = buffer.length;
int count = in.read(buffer,offset,size);
```

在一个TCP套接字上的输入操作阻塞直到至少有一些数据被接收到(*).然而,接收到的数据长度可能小于请求的数据长度.如果一些数据已经被接收到了套接字接收缓冲区中,输入操作可能只返回那些数据.如果接收缓冲区空了,输入阻塞直到一些数据被接收到,可能是一个单独的TCP片段,可能只返回那些数据.换句话说,count可能小于输入中的size,如上面的示例.

这个行为体现在套接字输入流自身从InputStream继承的read方法,和任何插入式(interposed)的I/O流的read方法,比如,BufferedInputStream, DataInputStream, ObjectInputStream, 或者PushbackInputStream.

然而,DataInput.readFully方法内部循环直到请求的数据完全读取或者直到EOP或者一个异常出现,无论哪一个先出现.readFully方法在一些从Reader继承的read方法内部调用,DataInputStream(readBoolean, readChar, readDouble, readFloat, readInt, readLong, readShort, 和 readUTF)方法的数据类型,和通过ObjectInputStream.readObject,所以这些方法也读取请求的全部数据或者抛出一个异常.

当且仅当另一端已经关闭了套接字或者关闭了输出-见3.7,接收的count为-1.

一个套接字的输入流的InputStream.available方法返回当前套接字缓冲区的数据的数量.这个可能为0.这是它做的一切.不要去预测未来:不要在一些网络协议操作中去询问另一端当前有多少数据正在发送中(也就是说,它已经发送了多少数据),或者在最后的write方法中发送了多少数据,或者下一条信息多达,或者它将要一共发送多少(比如,一共文件中有多少数据正在被发送).在TCP中没有任何这样的协议,所以不能.

最好附着一个BufferedInputStream到从套接字中获取的输入流.这个最小化了内核的上下文切换,并且能更快的汲取(drain)套接字接收缓冲区,反过来会减少发送端的拖延.它也最小化了JVM和JNI之间的来回切换.理论上BufferedInputStream缓冲区应该至少和套接字的缓冲区一样大,这样接收缓冲区可能尽可能快的汲取.

```
Socket socket;// initialization not shown
InputStream in = socket.getInputStream();
in = new BufferedInputStream
(in, socket.getReceiveBufferSize());
```

为了接收Java数据类型,使用DataInputStream直接附着在套接字输入流或者最好,附着一个BufferedInputStream(如上所示):

```
DataInputdis = new DataInputStream(in);
// examples …
booleanbl = dis.readBoolean();
byte b = dis.readByte();
char c = dis.readChar();
double d = dis.readInt();
float f = dis.readInt();
long l = dis.readInt();
short s = dis.readInt();
String str = dis.readUTF();
```

为了接收序列化Java对象,在你的输入流中包装一个ObjectInputStream:

```
ObjectInput ois = new ObjectInputStream(in);
// example …
Object object = ois.readObject();
```

因为ObjectInputStream继承了DataInputStream,你也可以使用上面所示的数据类型方法.然后要注意ObjectInputStream假设了在数据流中使用的是ObjectOoutputStream协议,这样你只能用来输入,如果你在另一端的输入使用ObjectOutputStream.你不可以使用DataOutputStream写入数据类型然后使用ObjectInputStream读取它们.

同样见3.6.2讨论的对象流死锁.

---

* 除非你使用第5章讨论的非阻塞I/O.

3.6.4 通道I/O

通道I/O在JDK 1.4引入,通过文件和套接字提供了高性能和可扩展的I/O.将在第5章详细讨论.

3.7 终止

终止一个连接的最简单的方式就是关闭套接字,终止两个方向的连接,并释放平台的套接字资源.

在关闭一个套接字之前,TCP 'shutdown'功能为套接字的输入和输入独立地提供了一种终止的方式,如3.7.1和3.7.2节讨论的.

套接字必须由会话的双方关闭,当会话完成,如3.7.4讨论的.

当服务器提供的服务被终止,监听中的套接字必须被关闭,如3.7.4讨论的.这个可以在和接收的套接字会话处理中的时候完成,而不是干扰这些会话.

3.7.1 输出关闭(半关闭)

输出关闭也称为'半关闭'.它使用以下方法完成:

```
class Socket {
    void shutdownOutput() throws IOException;
    boolean isOutputShutdown();
}
```

输出关闭有以下的影响:

(a) 本地的,本地套接字和它的输入流行为通常是为了读取,但是写,套接字和它的输出行为好像已经被这个终端关闭了:随后套接字的写将抛出IOException.

(b) TCP的正常的连接-关闭顺序(一个FIN通过一个ACK确认)是排队去发送在所有挂起的数据已经被发送并确认.

(c) 远程端,远程套接字行为通常是为了写目的,而不是读取目的,套接字的行为好像已经被关闭了:从套接字中的进一步的读将返回一个EOF.也就是,-1的count或者EOFException,决定于被调用的方法.

(d) 当本地套接字最终被关闭,连接-终止顺序已经被发送,不会重复;如果另一端已经同样做了一个半关闭操作,套接字上的所有的协议交互现在是完成的.

3.7.3 关闭一个连接的套接字

一旦会话完成,套接字必须被关闭.在Java中,这个通常通过下面的方法完成:

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

实际上完成这个有以下几种方式:
(a) 使用socket.close()关闭套接字自身;
(b) 通过调用socket.getOutputStream().close()关闭从套接字获得的输出流;
(c) 通过调用socket.getInputStream().close()关闭从套接字获得的输入流.

为了关闭套接字和释放所有的资源,其中任何一个就足够了,并且其中的某一个是必要的.你不能使用在任意给定的套接字上的除此之外的其它技术.作为一个通用的规则,你应该关闭输出流而不是输入流或者是套接字,因为输出流可能需要刷新.

关闭一个套接字是一个输出操作,并且,就像上面讨论的输出操作,它通常是异步发生的(但是见 3.13):不能保证另一端已经接收到了关闭,或者再一次,它已经接收到了之前输出操作的数据.服务器端和客户端都必须关闭套接字.

如果Socket.close()抛出一个IOException,这意味着可能你已经关闭了套接字,比如,上面列出的方式的另一种.也可能意味着TCP已经探测到它不能发送之前缓冲的数据.如上面讨论的,你的应该程序协议是唯一可同步探测这个问题的方式.

Socket.close()出现IOException并不意味着另一端已经关闭了它的连接.另一端可能已经关闭了它的连接,但是这是一个正常的状态,TCP协议设计明确地(FIXME: caters for it).两边都必须关闭,总要有先关闭的.关闭一个套接字如果另一端已经关闭了,不会抛出IOException.

---

isClose方法判断本地套接字是否已经关闭.它不会判断关于另一端连接的任何信息.

3.7.4 关闭一个TCP服务器

服务器通常应该有一些机制去关闭.通常这个通过在一个接收的连接上发送一个协议命令完成;也可以通过一个命令行或者图形用户界面完成.

关闭一个服务器需要关闭监听中的套接字.在Java中,这个意味着调用以下的方法:

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

套接字上的任意的并发或者随后的ServerSocket.accept操作将抛出SocketException.然而任何存在的已经接收的套接字不会会关闭中的ServerSocket影响.

ServerSocket.accept抛出的异常的消息文本在JDK1.4.1中是'Socket Closed',但这个可能在Java不同的版本和实现中变化.

---

isClose方法判断本地套接字是否已经关闭.它不会判断关于另一端连接的任何信息.

3.8 套接字工厂

在面向对象设计中,工厂是创建对象的对象(或者类).一个套接字工厂是一个创建套接字或者服务器套接字,或者两者的工厂.和所有的工厂对象一样,套接字工厂关注于对象创建处理;隐藏了系统其它部分的实现细节;为能提供的不同的实现提供了一致的对象创建接口.

Java在三个层次上支持套接字工厂: java.net.socket工厂,RMI 套接字工厂,和SSL套接字工厂.这些就爱那个在下面分别描述.

java.net.socket工厂由Java使用去提供套接字自身的实现.

java.net.Socket和java.net.ServerSockt类是实际的门面.这些门面类定义了Java套接字API,但是代理了做实际工作的套接字实现对象的所有动作,

套接字实现继承了抽象的java.net.SocketImpl类:

```
class SocketImpl {
    // …
}
```

工厂提供了java.net.SocketImplFactory接口的实现:

```
interface SocketImplFactory {
    SocketImpl createSocketImpl();
}
```

一个默认的套接字工厂总是被安装,产生一个SocketImpl对象,类型为package-protected的类java.net.PlainSocketImpl.这个类有本地C语言套接字API接口的native方法,比如,Berkeley Sockets API或者winsock.

套接字工厂可以被设置;

```
class Socket {
    static void setSocketFactory(SocketImplFactory factory);
}
```

setSocketFactory方法在JVM的生命周期只能被调用一次.它需要被分配一个RuntimePermission 'setFactory',否则抛出SecurityException.

应用程序很少或者不使用这个功能.

3.8.2 RMI套接字工厂(未完成)

3.8.3 SSL套接字工厂

javax.net工厂类SocketFactory和ServerSocketFactory在第7章讨论.

3.9 TCP中的权限

如果安装了Java security管理器,每个套接字操作需要java.net.SocketPermission.

权限在一个安全策略文件中被管理-一个名为'java.policy'的策略文件,通过JDK和JRE提供的policytool程序管理.Java 2权限框架在JDK 'Guide to Features/Security'文档中描述,在这里就不讨论了.

在策略文件中的一个SocketPermission条目有两个字段: 'action',也就是,要尝试的网络操作.和'target',也就是动作引用的本地或者远程TCP端点,如下格式:

```
host[:port]
```

host是一个明确或者通配符的主机名或者一个IP地址,port是一个端口数或范围.action字段和每个TCP网络操作的相应的target字段在表格3.1显示.

表格3-1 TCP中的权限

动作	描述
accept	ServerSocket.accept方法需要.目标host指明了正在被接收的远程TCP端点.
connect	Socket的非默认构造函数和它的connect方法,和获取InetAddress对象的时候需要.目标host指明了要正在连接的远程TCP端点.
listen	ServerSocket的非默认构造函数和它的bind方法需要.目标host指明了本地TCP套接字,也就是,ServerSocket要绑定的本地端点.host的唯一合理的值为'localhost'.默认的策略文件给'localhost:1024-'分配了'listen'权限.
resolve	'accept','connect',或者'listen'权限隐含的,所以没有必要明确地指定它.

3.10 TCP中的异常

在阻塞模式的TCP套接字操作中有意义的Java异常,和它们的来源和起因,在表格3-2展示.

在这个表格中,'C'或'U'表明了异常为检查的(C)是未检查的(U).

表格3-2 TCP中的异常

Exception	Thrown by & cause	C/U
java.net.BindException	由ServerSocket和Socket的构造,和它们的bind方法抛出,如果请求的本地地址或者端口不能被分配(补充,还有一种情况: 地址已经被使用).	C
java.net.ConnectException	由Socket构造和它的connect方法抛出,当错误地连接到一个远程地址和端口,通常因为连接被拒绝(在指定的(address, port)上没有监听)(补充,还有一种情况: 连接超时)	C
java.rmi.ConnectException		C
java.lang.IllegalArgumentException	由InetSocketAddress,Socket和ServerSocket的一些方法抛出,如果一个参数为null或者超出异常.	U
	由Socket.connect,Socket	

java.nio.channels.IllegalBlockingModeException	的输入和输出流操作,和ServerSocket.accept抛出,如果套接字有一个关联的非阻塞模式的通道;从JDK 1.4开始.	C
java.io.InterruptedIOException	由Socket输入流操作抛出,如果发生超时.在JDK 1.4之前.	C
java.io.IOException	基本的I/O异常类.与TCP关联的衍生异常类包括BindExceptin, ConnectException, EOFException, InterruptedIOException, NoRouteToHostException, ProtocolException, SocketException和UnknownHostException	C
java.net.NoRouteToHostException	由Socket的非默认构造函数和它的connect方法抛出,表明一个错误发生当连接到一个远程地址和端口,大部分通常是如果远程的主机不能达到,因为一个间接的防火墙,或者中间的路由器down了.	C
java.net.ProtocolException	由Socket和ServerSocket的构造方法,和Socket的输入和输出流操作抛出,表明一个错误发生在底层的协议,比如TCP错误.	C
java.lang.SecurityException	由Socket和ServerSocket的某些方法抛出,如果一个需要的权限没有分配,如表格3.1所示.	U
java.net.SocketException	由Socket的许多方法和Socket输入流操作抛出,表明一个底层的TCP错误发生,或者套接字被另一个线程通过InterruptibleChannel.close关闭了. 如果消息包含文本'Connection reset',连接的另一端已经发出了一个reset(RST):从此时起套接字是无用的,它应该被关闭和废弃. 许多异常从它衍生出来:包括BindException和ConnectException.	C
java.net.SocketTimeoutException	由Socket输入流操作抛出,表明发生了超时;从JDK 1.4开始;为了向后兼容JDK 1.4之前的程序继承了InterruptedIOException	C
java.net.UnknownHostException	由InetAddress的工厂方法抛出,当使用String主机名,由这些方法隐式解析.表明了命名的主机	C

的IP地址不能从命名服务中确认.

3.11 套接字选项

套接字选项控制了TCP的高级特性.在Java中,套接字选项通过java.net.Socket和java.net.ServerSocket的方法控制.

下面出现的套接字选项或多或少是按照他们的相对重要性的顺序.

3.12 套接字超时(需要重译)

不能假设一个应用程序永远等待一个远程的服务,或是服务总是及时响应,或是服务或中间的网络基础设施在检测方面只能失败.事实上,一个TCP连接可以在不会被服务器或者客户端检测到的情况下失败.任何无限超时读取的网络程序迟早会无限延迟.

在3.17节描述的'keep-alive'特性提供了这个问题的部分解决方案,如果平台支持的话.Java程序可以运行于任何的平台,无权承担这个.即使平台知道,不支持keep-alive,默认的延迟是两个小时,在死亡的(dead)的连接被检测到,这只能由管理员系统范围的改变(this can only be altered system-wide by an administrator),如果真的发生的话.(Usually this two-hour detection period is only palatable as a final fall-back).

因为这些原因,谨慎(prudent)的网络程序总是使用一个有限的读超时.这个通过以下方法管理:

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

timeout指定为毫秒,并且必须为正数,表明一个有限的超时,或者为0,表明不超时.默认情况下,read超时是无限的.

如果在套接字的阻塞读操作之前设置为一个正数(有限)的超时值,read将阻塞直到timeout周期,如果数据不可用,然后将抛出一个InterruptedIOException(补充, 注意: JDK 1.4之后抛出的是SocketTimeoutException,在JDK 1.4之前抛出的是此异常).如果超时是无限的,read将永远阻塞,或者直到一个错误发生.

客户端只是传输一个请求和等待一个回复,timeout的持续时间应该考虑两边的预期的传输时间加上请求的延迟时间-在另一端的延迟当回复正在被取回或者计算的时候.总的预期时间你应该等待多久是一个策略问题:一开始,time-out应该设置为预期时间总和的两倍.通常,timeouts应该稍微设置的长一些而不是短一些.

服务器等待一个客户端的请求,超时值完全是一个策略问题:在获取连接之前服务器准备等待一个请求多久?选择的周期应该足够长去支持沉重的网络负载和合理数量的客户端,但是不应该太长而占用宝贵的服务器资源(给一个服务器连接分配的资源由连接的套接字自身组成,和一个线程和客户端上下文的其它方面).

超时也可以在ServerSocket上设置:

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

timeout和之前一样设置为毫秒.这个设置决定了ServerSocket.accpet()在获得一个InterruptedIOException之前阻塞多久.这个设置不会被接收的连接的连接继承,也就是说,不会被

ServerSocket.accept()返回的套接字继承.一个ServerSocket超时可以在一个单独的线程中用来去轮询若干的ServerSockets,虽然在5.3.1节描述的Selector提供了一种更好的方式这样做.

设置一个套接字超时不会对一个已经在处理中的阻塞的套接字操作有影响.

3.13 套接字缓冲区

TCP为每个套接字分配一个发送缓冲区和接收缓冲区.这些缓冲区存在于内核的地址空间或者TCP协议栈(如有不同的话),而不是在"jvm或者进程地址空间".这些缓冲区的默认大小是由底层平台的TCP实现决定的,而不是Java.在最初的TCP实现中,发送和接收缓冲区默认都是2KB.在一些实现中,它们现在通常的默认大小差不多为28KB, 32KB, 或者64KB,但是你必须检查你的目标系统的特征.

3.13.1 方法

一个套接字的发送和接收缓冲区的大小通过以下方法管理:

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

class ServerSocket {
    void setReceiveBufferSize(int size) throws SocketException;
    int  getReceiveBufferSize()       throws SocketException;
}
```

size指定为字节.给这些方法提供的值只是作为底层平台的一个建议,在两端可能被调整来适应允许的范围,或者向上或者向下调整为合适的边界.

你可以在关闭套接字之前的任意时刻设置套接字的发送缓冲区大小.对于接收缓冲区,见3.3.6(服务器)和3.4.7(客户端)的讨论.

通过'get'方法返回的值可能和你设置的值不匹配.它们也可能与正在被底层平台使用的实际值也不匹配.

3.13.2 一个套接字缓冲区应该多大?

8KB,或者32KB,或者64KB,在如今的网络速度中足够吗?

较大的缓冲区,TCP可能操作更高效.较大的缓冲区大小利用网络的能力从而更有效:它们减少了向网络写入的物理次数;通过一个较大的包大小分担了40字节的TCP开销和IP数据包头;允许更多的字节被传输,'填充管道';允许更多的数据被传输在停止前.

下面的原则应该被遵守.

(a) 在以太网上,4KB肯定是不够的:将缓冲区从4KB提升到16KB将带来40%的吞吐量提升.
(b) 套接字缓冲区大小应该总是至少是连接的最大的分段大小(maxumiun segment size - MSS)的三倍,通常由网络接口的最大传输单元(maximum transmission unit - MTU)决定,(FIXME: less 40 to account for the size of the TCP and IP headers).以太网,MSS小于1500,使用8KB或者以上的缓冲区大小这是不存在问题的,但是其它物理层行为有所不同.
(c) 发送缓冲区大小应该至少和另一端的接收缓冲区一样大.
(d) 对于每次发送大量数据的应用程序,将缓冲区大小从48KB增加到64KB可能只会让你的应用程序单次的更高效的性能提升.对于这样的应用程序的最大性能,发送缓冲区应该和中间的网络产生的宽带-延迟一样大.

(e) 对于每次接收大量数据的应用程序的最大性能(比如,应用程序的另一端发送大量的数据),接收缓冲区需要尽可能和上面的约束一样大,因为TCP根据在接收者的缓冲区空间限制了发送者-一个发送者可以不发送数据除非它知道在接收者中有空间.如果接收的应用程序在从缓冲区中读取数据方面比较慢,它的接收缓冲区大小需要更大,这样不会拖延发送者.
(f) 对于发送和接收大量数据的应用程序,同时增加缓冲区大小.
(g) 在如今的大多数实现中缓冲区大小至少是8KB(WINSOCK),28KB(OS/2),52KB(Solaris).早期的TCP实现允许大于52,000字节的最大缓冲区大小.一些当前的实现支持最大256,000,000字节的大小或者更多.
(h) 为了在一个服务器中接收超过64KB的缓冲区大小,你必须在监听套接字之前设置缓冲区,接收的套接字将继承这个属性,如3.3.6节描述的.
(i) 无论缓冲区大小是什么,你应该通过以那个大小的块的方式写入来帮助TCP,也就是说,通过使用至少那个大小的BufferedOutputStream或者ByteBuffer.

3.14 多宿主

如我们在2.2.5节看到的,一个多宿主是一个有多于1个IP地址的主机.多宿主对于TCP服务器有重要的影响,对客户端没有什么影响.

3.14.1 多宿主-服务器

一个TCP服务器通常在所有的本地IP地址上监听,这样的一种服务器通常不必关心它可能运行于一个多宿主的主机.下面是一个TCP服务器可能需要注意多宿主的情况.

如果一个服务器服务于一个子网,它应该将它自身绑定到合适的本地IP地址.这个反过来可能需要在3.3.5节讨论的Socket.setReuseAddress方法的使用.

如果服务器提供它自己的IP地址给客户端,它必须返回一个客户端可以访问的IP地址.通常客户端不比访问服务器的所有IP地址,但是可能只需要访问其中一个.如果客户端已经连接到了服务器,保证一个返回的IP地址可用的最简单的方式就是强制它的地址为客户端用来连接的地址,由被接收的套接字的Socket.getLocalAddress方法给出,如3.4.4节描述的.

在目录服务中(directory services)当注册服务描述符的时候,服务器必须通过一个所有客户端都能访问的IP地址通知它自己.保证被通知服务地址是最可用的最好的方式是在每种情况中通知'most public(大多数公共的)'IP地址或者主机名.

3.14.2 多宿主-客户端

一个TCP客户端通常不关注它的本地地址,如我们之前在3.4.8看到的.如果有某些原因需要关注它用于连接的网络接口,它应该指定IP地址当这当这么做的时候,如3.4.4节讨论的.

3.15 Nagle's 算法(未完成)

3.16 Linger on close

Socket.setSoLinger方法控制当一个套接字关闭时TCP的行为:

```
class Socket {
    void setSoLinger(boolean linger, int timeout) throws SocketException;
    int getSoLinger() throws SocketException;
}
```

timeout指定为秒.getLinger方法的返回值为-1表明了默认设置(linger = false).
TCP为"linger on close"定义了三种不同的行为,如表3.3所示.

表3.3 TCP 'linger'设置

linger	timeout	描述
false	ignored	默认情况.当套接字被关闭(a),关闭中的线程不会阻塞,但是套接字不会被立即销毁:它进入CLOSING状态,当任意剩余的数据被传输,FIN-ACK关闭协议与另一端交互;套接字然后进入TIME-WAIT状态,持续一个TCP分段的最大生命周期的两次,为了保证后面传输到套接字的数据被一个TCP RST拒绝,被选的时间间隔,TCP可以转播关闭协议的最后一部分如果有必要的话,这样本地和远程端口对在TIME-WAIT期间不会被重用,这样已经关闭的连接的延迟的数据分段不会传递到新的连接.当TIME-WAIT到期,套接字被销毁(b).
true	≠ 0	'Linger'.当套接字被关闭(a),关闭的线程阻塞('lingers'),同时任意挂起的数据被发送,关闭协议交互,或者超时过期,看哪一个先发生;线程然后继续运行;如果超时过期,(i)连接'hard-closed'如下面描述的(c),或者(ii)剩余的数据仍然排队等待传递,在连接通过FIN-ACK关闭之后,如上面描述的(d).这些语义是平台依赖的(Java不能克服它们). 在Java中,timeout是一个int值,指定为秒,限于 秒;一些平台进一步限制为 = 32.767秒,通过使用一个内部的16-bit有符号的数量代表了百分之一秒(e).
true	0	'Hard-Close'.当套接字被关闭(a),任何挂起的数据被废弃,关闭协议交互(FIN-ACK)不会发生,取而代之的是,发出一个RST,引起另一端抛出一个SocketException 'connection reset by peer'.

(a) 也就是通过Socket.close,Socket.getXXXStream.close,或者Socket.shutdownOutput.
(b) 许多实现不必阻止本地端口的重用在TIME-WAIT期间,即使当连接到一个不同的远程端点.
(c)这个行为是必须,通过winsock 2规范3.4.
(d) 这个行为是必须的,通过a Posix.1g draft (quoted in the comp.unix.bsd newsgroup by W.R. Stevens, 22 May 1996,但是IEEE Std 1003.1-201并没有这样,它没有定义.
(e) 如果timeout过去,Berkeley Sockets和winsock API都设置EWOULDBLOCK,并返回-1,尽管这个在IEEE Std 1003.1-201中并没有指定.在JDK 1.4.2中,Java忽略了这些,所有你不能在Java中判断在关闭端timeout是否过期了.作者已经为Socket.close请了一个改进,在这种情况下抛出InterruptedIOException.

3.17 Keep-Alive

3.21 将它们综合在一起

一个修正的TCP服务器实现了上面建议的改进,在示例3.6中展示.

```
public class ConcurrentTCPServer implements Runnable {

    public static final int BUFFER_SIZE = 128 * 1024;// 128k
    public static final int TIMEOUT = 30 * 1000; // 30s
    ServerSocket serverSocket;

    ConcurrentTCPServer(int port) throws IOException {
        // (Don't specify localAddress for ServerSocket)
        serverSocket = new ServerSocket(port);
        // Set receive buffer size for accepted sockets
        // before accepting any, i.e. before they are connected
        serverSocket.setReceiveBufferSize(BUFFER_SIZE);
        // Don't set server socket timeout
    }

    public void run() {
        for (;;) {
            try {
                Socket socket = serverSocket.accept();
                // set send buffer size
                socket.setSendBufferSize(BUFFER_SIZE);
                // Don't wait forever for client requests
                socket.setSoTimeout(TIMEOUT);
                // despatch connection handler on new thread
                new Thread(new ConnectionHandler(socket)).start();
            } catch (IOException e) {
                /* Exception handling, not shown … */
            }
        } // end for (;;)
    } // end run()
}
```

示例3.6 改进的TCP服务器

进一步的服务器架构在第5章和第12章讨论.

一个修正的TCP客户端实现了上面的建议的改进,在示例3.7展示.

```
public class TCPClient implements Runnable {

    public static final int BUFFER_SIZE = 128 * 1024;// 128k
    public static final int TIMEOUT = 30 * 1000; // 30s
    Socket socket;

    public TCPClient(String host, int port) throws IOException {
        this.socket = new Socket();
        // Set receive buffer size before connecting
        socket.setReceiveBufferSize(BUFFER_SIZE);
        // connect to target
        socket.connect(new Inet4Address(host, port));
        // Set send buffer size and read timeout
```

```
        socket.setSendBufferSize(BUFFER_SIZE);
        socket.setSoTimeout(TIMEOUT);
    }

    public void run() {
        try {
            // prepare to send request
            OutputStream out = new BufferedOutputStream(socket.getOutputStream(),
BUFFER_SIZE);
            // send request data, not shown …
            // flush request
            out.flush();
            // prepare to read reply
            InputStream in = new BufferedInputStream(socket.getInputStream(),
BUFFER_SIZE);
            // receive reply & process it, not shown …
        } catch (IOException e) {
            /* … */
        } finally {
            try {
                if (socket != null)
                    socket.close();
            } catch (IOException e) {
                // ignored
            }
        } // end finally
    } // end run()
}
```

示例3.7 改进的TCP客户端