Java sockets 101

Presented by developer Works, your source for great tutorials 30 Aug 2001

原文地址：https://www6.software.ibm.com/developerworks/education/j-sockets/index.html

作者：
Roy W. Miller (rmiller@rolemodelsoft.com), Software Developer, RoleModel Software, Inc
Adam Williams (awilliams@rolemodelsoft.com), Software Developer, RoleModel Software, Inc

翻译：
Tom Hu 胡继强
hujqiang@gmail.com

这篇指南将带你了解什么是套接字，并引导你学会如何使用 java 进行套接字编程。通过很多亲自实践的练习案例，从单客户端/服务器的通信到缓冲客户端来访问服务器，你将能通过学会处理典型的场景来解决现实世界中突然出现的问题。

Section 1. 指南说明

我需要这份指南吗？

套接字为两台计算机通信提供了一种机制，他在 James Gosling 开发出 java 语言之前就已经存在了。Java 可以让你高效的使用套接字，而不用关心操作系统的底层实现机制。很多书籍不是没有涉及到这部分内容就是说了很多凭空想象的废话。这份指南会告诉你如何在 Java 代码中高效的使用套接字。

本指南包括如下内容：

- 什么是套接字
- 它适合什么样的程序结构
- 最简单的可以执行的套接字实现程序 帮助你学习套接字基础知识
- 两个详细的“一步步”操作案例，包括多线程的套接字和缓冲池环境
- 一个关于现实环境中如何使用套接字的简短探讨

如果你能描述如何使用 java.net 程序包，这份指南对你可能用途不大，但对于新手而言，它是个不错的教程。如果你已有多年在 PC 和其他平台上的 socket 编程经验，你可能不会喜欢开始的几章。但如果你对套接字一无所知，并想在 java 程序中高效的使用它，这份指南将是个很好的开始。

Section 2. 套接字基础

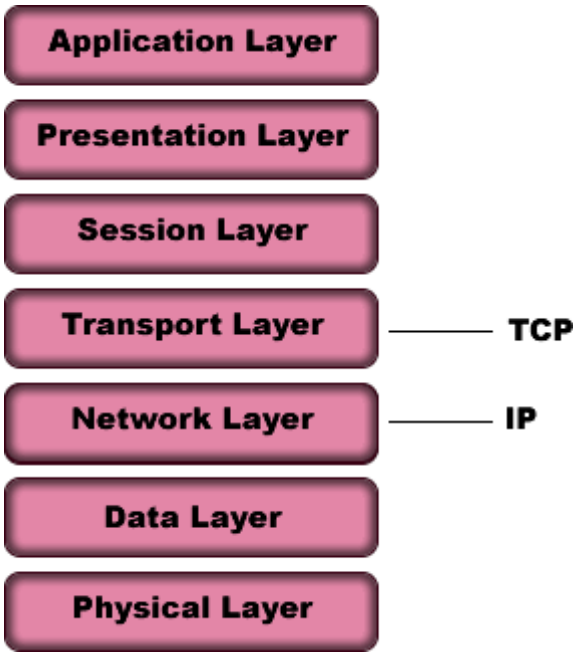
简介

很多程序员，不管是使用 java 还是其他语言，都不喜欢了解不同计算机间应用程序相互通信的底层机制。程序员希望处理比较容易理解的高层次的数据抽象。java 程序员希望使用他们熟悉的 java 架构来处理对象。

套接字存在于这样两个层面，我们极力避免的底层的通信细节，我们更愿意处理的高层抽象。这一节将探讨足够的底层通信细节，以便能更好的理解高层的应用抽象。

计算机网络

TCP/IP

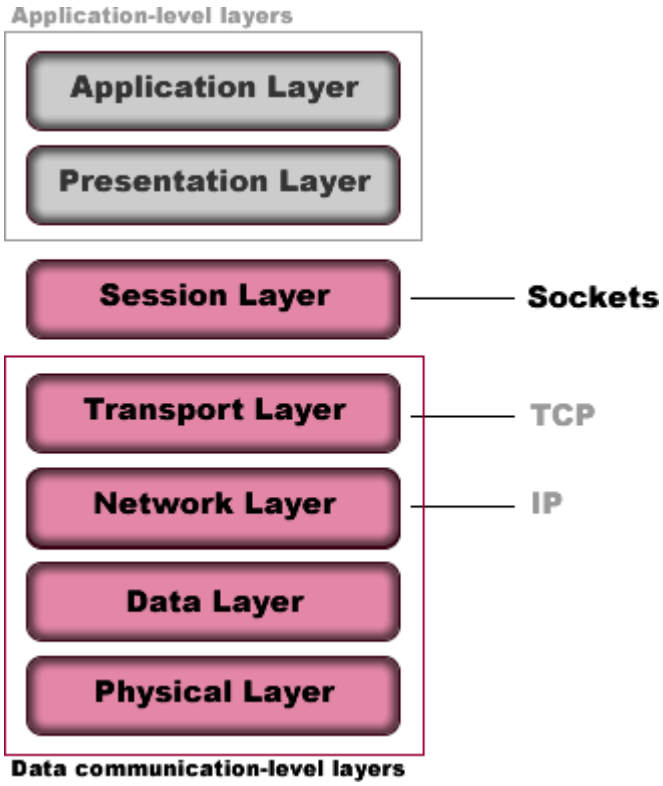


计算机之间通过一种很简单的方式进行运算和通信。计算机部件就是一个开关集合，通过开和关来存储和传输 1 和 0 格式的数据。当计算机之间需要通信时，它们所需要做的是约定好流的速度、顺序、时间等方式，然后来回进行字节流的传输即可。在两个应用进行通信时，你想怎么担心这些细节呢？

为了避免这种情况，我们需要一系列的报文协议来重复执行相同的工作。这可以让我们集中精力在应用层面的工作而不用考虑底层的网络通信细节。这些报文协议被称为“*stacks*”协议栈。最近最通用的协议栈是 TCP/IP.绝大部分的协议栈（包括 TCP/IP）都粗略的依附 ISO 标准。OSIRM 标准组织说可靠的计算机网络框架包括 7 个逻辑层次（参加上图）。所有公司的产品都是建立在这个模型的 7 个层次中的几个，从产生电子信号到在应用中展示数据。TCP/IP 存在于该模型中的两个层次，参加上图。我们不会详细阐述这个模型的各个层次，但我们会明确套接字所适用的层次。

套接字所在的层

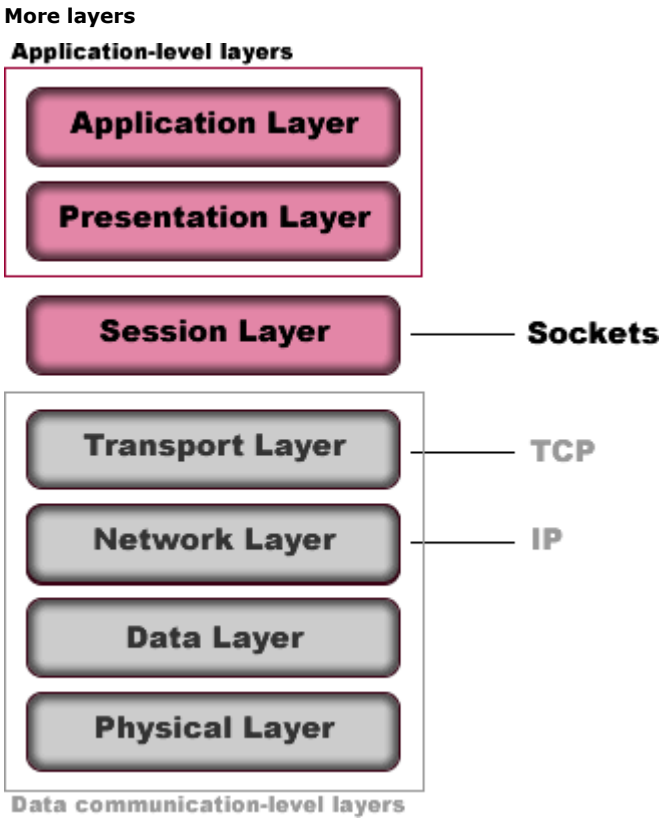
Where sockets fit



套接字适用于 OSI 模型的会话层，参加上图。会话层是加载应用导向的高层和实时进行数据通信的底层之间的层。会话层提供管理和控制计算机间通讯的数据流的服务。作为会话层的一部分，套接字提供了一种抽象，这个抽象隐藏了在线路传输过程中对于比特和字节的复杂处理过程。换句话说，套接字允许我们通过声明应用程序所需要传输的一些字节来传送数据。套接字完成了传输任务并隐藏了具体细节。

当你拎起电话听筒，你给声音传感器一些声波，传感器将你的声音转换成可传输的电子信号数据。电话机在电信通信网络中扮演着人类接口的角色。你不用关心你的声音如何被转换的，只需要选择你要联系的人就好了。与此相似，套接字扮演着一个高层次的接口，并隐藏掉了复杂的 01 传输过程。

在应用中使用套接字



当你使用套接字进行编码时，程序代码运行在表示层。表示层提供了应用层所需使用的通用表示信息。假设你计划将你的应用连接到使用 EBCDIC 编码的传统银行系统。你的应用领域的对象使用 ASCII 的格式进行数据存储。在这种情况下，你要负责写程序在表示层将 EBCDIC 编码的数据转换成 ASCII 码，并为你的应用层提供一个领域对象。这时，你的应用层可以任意处理它领域内的对象。

你写的套接字程序仅在表示层生效，而应用层并不知道套接字是如何工作的。

什么是套接字

现在我们知道了套接字扮演的角色，问题是，什么是套接字。Bruce Eckel 在他的书《java 编程思想》中这样描述的：

套接字是一种软件抽象，用来表示两台机器间连接的终端。对于指定的连接，每台机器上都有一个套接字，你可以认为两台机器间存在一个“电缆”，电缆的两端分别插在两台机器的“插座”中。当然，这根“电缆”和“插座”是根本不存在的。这种抽象的重点是我们不必关心我们不需要的东西。

简单的说，一台机器上的套接字与另一台机器上的套接字建立了一个通信通道。程序员可以通过这个通道在两台机器间进行数据传送。当你发送数据时，TCP/IP 协议栈的每一层都会在你的数据报文头部添加适合的信息来组合报文。协议栈通过这些报文头来将其数据发送到目的地。好消息是，java 语言以流的形式封装了这些所有的格式用于程序代码中，这也就是为什么有时候我们称之为流套接字。

将套接字理解为电话的电话听筒，你和我利用听筒在专用的通道中进行沟通，通话并不会终止直到将听筒挂掉。（除非我们使用手机，\(^o^)/~）在我们挂掉之前，我们各自的电话线路都显示繁忙。

如果你需要在两台计算机间通讯，并且没有高层的通讯设备（如 ORB、CORBA、RMI、IIOP 等），套接字是你的最佳选择。低层的套接字实现细节也同样棘手。幸运的是 java 平台为你提供了简单但强大的高层次的抽象，可以让你非常简单的使用套接字。

套接字的类型

通常来说，在 java 语言中有两种类型的套接字：

- TCP 套接字 （通过 Socket 类进行实现，后面我们将讨论到）
- UDP 套接字 （通过 DatagramSocket 类进行实现）

TCP 和 UDP 实现了相同的功能，但实现方式不同。它们都是接收传输协议数据包并将其传输到表示层。TCP 将消息分组，形成数据报并在接收完成后进行重组。它同样支持丢包重发。若使用 TCP，高层不用担心数据的丢失等问题。UDP 不支持这些特性，它按顺序完整进行报文传输。高层必须自己包装数据是完整的，且顺序正确。

总体上，UDP 占用很少的资源，这只在你的应用并没有一次需要交换很多的数据并且不需重新组合很多数据包来构成一个消息的情况下。否则，TCP 是最简单并且总体最高效的选择。

因为大部分读者更喜欢 TCP 而不是 UDP，我们的探讨仅限于将 java 中面向 TCP 的类。

Section 3. 一个秘密的套接字

介绍

java 平台在 java.net 包中提供了套接字的实现。在本指南中，我们将使用 java.net 包中的以下三个类：

- URLConnection
- Socket
- ServerSocket

在 java.net 包中有很多类，但这些是你最常用的。让我们先从 URLConnection 开始。这个类提供了一种使用套接字的方法，而不用关系套接字的具体细节。

使用简单套接字

URLConnection 类是所有在应用和 URL 之间实现了通信连接的类的抽象父类。URLConnectionS 在获取网络服务器的文档时是最常用的，但也可以用于连接使用 URL 标识的所有类型的资源。这个类的实例可以实现对资源的读和写操作。例如，你可以连接一个 servlet，并且向服务器发送一个格式规范的 XML 字符串。URLConnection 的子类实现了（如 HttpURLConnection）额外的特性。对于本文档中的实例，没有任何额外的特性，我们使用 URLConnection 提供的默认行为。

实现对 URL 的连接需要如下几个步骤：

- 创建 URLConnection
- 使用不同的 setter 方法配置 URLConnection 的实例
- 连接到 URL
- 使用不同的 getter 方法实现与 URL 的交互

接下来，我们来看一下如何通过程序代码来实现 URLConnection 与服务器的连接的。

URLClient 类

我们从 URLClient 类的结构开始

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


public class URLClient {
   protected URLConnection connection;
```

```
    public static void main(String[] args) {
    }
    public String getDocumentAt(String urlString) {
    }
}
```

程序开始导入需要的程序包，java.net 和 java.io.

建立一个 URLConnection 的实例，connection

一个 main()方法，实现获取文档的逻辑流程的方法。

还有一个 getDocumentAt()方法，连接到服务器并请求文档。接下来我们将详细探讨这几个方法。

获取一个文档

main()方法实现了获取文档的逻辑流程：

```
public static void main(String[] args) {
    URLClient client = new URLClient();
    String yahoo = client.getDocumentAt("http://www.yahoo.com");
    System.out.println(yahoo);
}
```

我们的 main()方法只不过创建了一个 URLclient 实例，并调用了 getDocumengAt()方法，参数 Wie 一个合格的 URL。当这个方法返回一个文档我们将其存放在一个字符串变量中并打印到控制台上。真正的工作是在 getDocumentAt()方法里面做的。

从服务器获取文档

getDocumentAt()方法实现了从网络获取文档的真正方法。

```
public String getDocumentAt(String urlString) {
    StringBuffer document = new StringBuffer();
    try {
        URL url = new URL(urlString);
        URLConnection conn = url.openConnection();
        BufferedReader reader = new BufferedReader(new InputStreamReader(conn.getInputStream()));

        String line = null;
        while ((line = reader.readLine()) != null)
                document.append(line + "\n");
        reader.close();
    } catch (MalformedURLException e) {
        System.out.println("Unable to connect to URL: " + urlString);
    } catch (IOException e) {
        System.out.println("IOException when connecting to URL: " + urlString);
    }
    return document.toString();
}
```

getDocumentAt()方法包含了一个字符参数，这个参数是我们需要访问的 URL 地址。首先，我们创建一个 StringBuffer 对象来保存获取到文档的每一行。接下来我们根据传入的 urlString 参数创建一个 URL 对象，

根据 URL 对象创建一个 URLConnection 对象。

```
URLConnection conn = url.openConnection();
```

URLConnection 对象创建完成后，我们通过它的 getInputStream 来获取 InputStream 对象，并根据这个对象建立 InputStreamReader 对象。根据 InputStreamReader 我们可以将其中的内如读入到 BufferReader 对象中，从而得到我们需要从服务器获得的文档信息。在 java 代码中我们将经常使用这种封装技术，但我们不会经常提到它。在我们进行下一步之前你应该熟悉它。

```
BufferedReader reader =
    new BufferedReader(new InputStreamReader(conn.getInputStream()));
```

一旦创建了 BufferReader 对象，我们获取文档内容就很方便了。在 while 循环中调用 readLine()方法即可。

```
String line = null;
while ((line = reader.readLine()) != null)
   document.append(line + "\n");
```

readLine()方法会一直运行知道遇到行结束符（如换行符）。如果它没有遇到换行符她会一直等待。当连接关闭时，它将返回 null 值。在这种情况下，我们将获取到的行添加到名字为 document 的 StringBuffer 对象中。这保持了从服务器获取到的文档的原始格式。
当我们读取结束后，我们关闭 BufferReader 对象。

```
reader.close();
```

如果传给 URL 构造方法的 urlString 对象不合法，将会抛出 MalformedURLException 异常。如果其他程序出错，比如在获取 InputStream 时，IOException 会被抛出。

总结

实际上，URLConnection 使用套接字来获取我们指定的 URL（将 URL 解析成 IP 地址），但是我们并不知道也不需要关心它。我们接下来将使用套接字。
在继续之前，我们回顾一下使用 URLConnection 的步骤：

1. 使用一个你需要连接的、合法的 URL 字符串来初始化一个 URL 对象。如果 url 字符串不合法将会抛出 MalformedURLException。
2. 通过 URL 对象打开一个连接。
3. 将连接获取的 InputStream 封装成 BufferReader 对象，这样你可以获取字符了。
4. 使用 BufferReader 读取文档。
5. 关闭 BufferReader。

你可以在 Code listing forURLClient 获取 URLClient 完整的代码。

一个简单的例子

背景

这一节我们要介绍的例子说明了如何在 java 代码中使用套接字和服务器套接字。客户端使用套接字来连接服务器。服务器使用套接字服务器监听 3000 端口。客户端获取服务器 C 盘下的一个文件内容。

为了讲解更清晰，我们将例子分为客户端和服务器端，最后我们将两者结合起来，这样你就能从全局把握他们。

我们使用 IBM 的 Java3.5 可视化编程环境，使用的 JDK 版本是 1.2，JDK1.1.7 以后的版本都可以。（注，此处作者使用 JDK 版本较低，我们可以使用 JDK1.6 或任何 java 编辑器，比如 Eclipse 等）。客户端和服务器端

会运行在同一台机器上，不用担心没有网络连接。

创建 Java 类 RemoteFileClient

下面试 Java 类 RemoteFileClient 的结构：

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

public class RemoteFileClient {
   protected String hostIp;
   protected int hostPort;
   protected BufferedReader socketReader;
   protected PrintWriter socketWriter;

   public RemoteFileClient(String aHostIp, int aHostPort) {
      hostIp = aHostIp;
      hostPort = aHostPort;
   }
   public static void main(String[] args) {
   }
   public void setUpConnection() {
   }
   public String getFile(String fileNameToGet) {
   }
   public void tearDownConnection() {
   }
}
```

首先我们引入 java.net 和 java.io 两个包。java.net 包含有所需要的套接字工具。java.io 包含有对流进行读写的工具，这是和 TCP 套接字进行通讯的唯一方法。

我们在此 java 类实例中使用成员变量来支持套接字流的读取和写入，同样也使用成员变量来存储远程服务器的详细信息。

在构造方法中，传入远程服务器的 IP 地址和端口号，并将它们传入类实例变量中。

在这个 java 类中有一个 main()方法和其他三个方法，随后我们将详细讨论它们。现在仅仅知道 setUpConnection()实现连接了远程服务器，getFile()方法实现获取远程服务器文件，tearDownConnection()方法断开远程服务器连接。

mian 方法的实现

现在我们重新写 main 方法，首先创建一个 RemoteFileClient 实例，使用它获取远程服务器文件，接下将文件内容打印出来。

```
public static void main(String[] args) {
   RemoteFileClient remoteFileClient = new RemoteFileClient("127.0.0.1", 3000);
   remoteFileClient.setUpConnection();
   String fileContents =
      remoteFileClient.getFile("C:\\WINNT\\Temp\\RemoteFile.txt");
   remoteFileClient.tearDownConnection();
```

```
    System.out.println(fileContents);
}
```

main 方法使用 IP 地址和端口号两个参数实例化了一个 RemoteFileClient 对象。接下来建立了一个和远程服务器的连接。再接下来客户端从远程服务器获取指定的文件。最后客户端关掉和服务器的连接。我们将把获取的文件内容打印到控制台以验证是否准确。

建立连接

我们重写 setUpConnection 方法，实现与套接字的连接并允许我们访问他的数据流。

```
public void setUpConnection() {
    try {
        Socket client = new Socket(hostIp, hostPort);

        socketReader = new BufferedReader(
                        new InputStreamReader(client.getInputStream()));
        socketWriter = new PrintWriter(client.getOutputStream());

    } catch (UnknownHostException e) {
        System.out.println("Error setting up socket connection: unknown host at " + hostIp + ":" + hostPort);
    } catch (IOException e) {
        System.out.println("Error setting up socket connection: " + e);
    }
}
```

setUpConnection 方法根据服务器的 IP 地址和端口号建立 Socket 连接。

```
Socket client = new Socket(hostIp, hostPort);
```

我们将套接字的 InputStream 流封装到 BufferReader 对象中，这样我们就可以从流中读取数据行了。接下来，我们将 OutputStream 对象封装到 PrintWriter 中，这样我们就可以将我们的请求发送到服务器了。

```
socketReader = new BufferedReader(new InputStreamReader(client.getInputStream()));
socketWriter = new PrintWriter(client.getOutputStream());
```

记住一点，我们的客户端和服务器之间完全是来回进行字节的传送。客户端和服务器都必须知道对方将要发送什么，这样它们才能正确的进行响应。在这种情况下，服务器知道我们将发送一个合法的文件路径。将创建 Socket 实例时，UnknownHostException 可能会被触发。这里我们不做任何操作，但会打印到控制台一些信息来帮助我们了解什么地方出问题了。同样，在获取套接字的 InputStream 或 OutputStream 时如果会有 IOException 被抛出，我们会打印响应的信息到控制台。这是这份指南的通常方法。在产品代码中，我们需要更多的代码去处理它们。

和服务器通信

此处我们实现 getFile()方法。它将告诉服务器我们需要哪个文件，并且在服务器将文件发送回去的时候接收文件内容。

```
public String getFile(String fileNameToGet) {
    StringBuffer fileLines = new StringBuffer();
```

```
   try {
      socketWriter.println(fileNameToGet);
      socketWriter.flush();

      String line = null;
      while ((line = socketReader.readLine()) != null)
         fileLines.append(line + "\n");
   } catch (IOException e) {
      System.out.println("Error reading from file: " + fileNameToGet);
   }

   return fileLines.toString();
}
```

调用 getFile 方法是需要一个合法的文件路径字符串。方法使用一个名为 fileLines 的 StringBuffer 对象来存储我们读取的服务器上文件的每一行内容。

```
StringBuffer fileLines = new StringBuffer();
```

在 try catch 块中，我们使用建立连接时创建的 PrintWriter 对象来发送我们的请求。

```
socketWriter.println(fileNameToGet);
socketWriter.flush();
```

注意我们调用 PrintWriter 的 flush()方法而不是关闭它。这会强制将所有数据发送到服务器而不用关闭套接字。

一旦我们将数据发送到套接字，我们期待有所响应。我们需要等待套接字的 InputStream，并在一个 while 循环中调用 BufferReader 的对象的 readLine 方法。我们将获取到的每一行添加到名为 fileLInes 的 StringBuffer 对象中，在每一行后面添加一个换行符来标识一行。

```
String line = null;
while ((line = socketReader.readLine()) != null)
   fileLines.append(line + "\n");
```

关闭连接

此处我们覆写 tearDownConnection 方法，它负责使用连接之后的清理工作。

```
public void tearDownConnection() {
   try {
      socketWriter.close();
      socketReader.close();
   } catch (IOException e) {
      System.out.println("Error tearing down socket connection: " + e);
   }
}
```

tearDownConnection 方法分别关闭我们在套接字的 InputStream 和 OutputStream 对象里创建的 BufferReader 和 PrintWriter 对象。在做关闭操作时，会关掉潜在的从服务器获取的流，所以我们需要捕捉可能发生的 IOException。

包装客户端

到这里，我们的 java 类已经完成。在进行下一步的服务器端介绍之前，让我们回顾一下创建和使用套接字的步骤。

1. 通过需要连接的服务器的 IP 地址和端口号实例化一个 Socket 对象（可能会抛出异常）。
2. 从这个 Socket 对象获取流来进行读写。
3. 通过 BufferReader 和 PrintWriter 将流进行封装
4. 对 Socket 对象进行读和写
5. 关闭流。

You can find the complete code listing for `RemoteFileClient` at Code listing for `RemoteFileClient`.

你可以在 Code listing for `RemoteFileClient`.找到完整的代码。

创建 RemoteFileServer 类

这是 RemoteFileServer 类的结构信息

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

public class RemoteFileServer {
   protected int listenPort = 3000;
   public static void main(String[] args) {
   }
   public void acceptConnections() {
   }
   public void handleConnection(Socket incomingConnection) {
   }
}
```

类似于客户端类，首先我们需要引入 java.net 和 java.io 两个包。接下来，我们定义一个成员变量来存放端口号，来实现连接的监听。默认情况下，我们设置为 3000

我们的类有一个 main 方法和其他两个方法。随后我们会详细讨论它们。现在你只需要明白 acceptConnection 方法的功能是允许客户端连接到这台服务器，handleConnection 方法和客户端套接字进行交互，将客户端请求的文件内容发送给它。

实现 main 方法

这里我们将实现 main 方法，它会创建一个 RemoteFileServer 对象，并调用它的 acceptconnection 方法。

```
public static void main(String[] args) {
   RemoteFileServer server = new RemoteFileServer();
   server.acceptConnections();
}
```

服务器端的 main 方法比客户端的 main 方法简单多了。首先创建一个 RemoteFileServer 对象，来监听默认端口上的连接，然后调用 acceptConnections 方法告诉服务器来监听。

接收连接

这里我们实现 acceptConnection 方法，它将创建一个服务器端套接字，并等到连接请求。

```
public void acceptConnections() {
```

```
   try {
      ServerSocket server = new ServerSocket(listenPort);
      Socket incomingConnection = null;
      while (true) {
         incomingConnection = server.accept();
         handleConnection(incomingConnection);
      }
   } catch (BindException e) {
      System.out.println("Unable to bind to port " + listenPort);
   } catch (IOException e) {
      System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);
   }
}
```

acceptConnection 方法利用需要监听的端口号建立一个 ServerSocket 对象。通过调用 ServerSocket 的 acdept 方法来开始监听。accept 方法在没有连接请求时一直处于空闲状态。在此基础上，accept 方法返回一个新的 Socket 对象，其端口是服务器随机指定的，并将此对象传给 handleConnection 方法。注意，这里的获取连接是使用的一个无限循环，此处没有退出机制。

只要你创建 ServerSocket，java 代码可能会因为无法绑定指定的端口号而回抛出错误，有可能是因为该端口号已经被占用。所以我们需要捕捉可能的 BindException。同时，应该向客户端一样在尝试获取服务器套接字时，捕捉 IOException。注意，你可以在调用 accept 方法时通过 setSoTime 方法设定一个超时时间，参数时毫秒，来避免长时间的等待。调用 setSoTime 方法后，若超过设定的时间后会导致 accept 抛出 IOException。

处理连接

这里我们实现 handleConnection 方法。它将使用连接的流来接收输入和发送输出。

```
public void handleConnection(Socket incomingConnection) {
   try {
      OutputStream outputToSocket = incomingConnection.getOutputStream();
      InputStream inputFromSocket = incomingConnection.getInputStream();

      BufferedReader streamReader =
         new BufferedReader(new InputStreamReader(inputFromSocket));

      FileReader fileReader = new FileReader(new File(streamReader.readLine()));

      BufferedReader bufferedFileReader = new BufferedReader(fileReader);
      PrintWriter streamWriter =
         new PrintWriter(incomingConnection.getOutputStream());
      String line = null;
      while ((line = bufferedFileReader.readLine()) != null) {
         streamWriter.println(line);
      }

      fileReader.close();
      streamWriter.close();
```

```
        streamReader.close();
    } catch (Exception e) {
        System.out.println("Error handling a client: " + e);
    }
}
```

和客户端一样，我们使用 getOutStream 方法和 getInputStream 方法来获取 socket 的流，将 InputStream 包装到 BufferReader，OutputStream 包装到 PrintWriter。在服务器端，我们需要增加一些代码来读取目标文件，并将文件内容一行行的发送到客户端。下面是核心代码：

```
        FileReader fileReader = new FileReader(new File(streamReader.readLine()));
        BufferedReader bufferedFileReader = new BufferedReader(fileReader);
        String line = null;
        while ((line = bufferedFileReader.readLine()) != null) {
            streamWriter.println(line);
        }
```

下面代码需要详细解释。让我们一点点的看：

```
FileReader fileReader = new FileReader(new File(streamReader.readLine()));
```

首先我们在套接字的 InputStream 上使用 BufferReader。我们需要获取一个合法的文件路径，这样我们便可以使用路径名称来创建一个 File 对象。我们创建一个新的 FileReader 对象来处理读取文件。

```
BufferedReader bufferedFileReader = new BufferedReader(fileReader);
```

这里我们用 BufferReader 包装了 FIleReader，这样我们可以一行行的读取文件。

接下来，我们调用 BufferReader 的 readLine 方法。这个调用会一直等待直到有字节出现。当我们获取到字节，我们将它们存放到本地变量 line 中，并将它们发送到客户端。当读写完毕，我们关闭这些流。

注意，在我们从套接字读文件结束后关闭 streamWriter 和 streamReader 对象。你可能会问，我们为什么不在获取文件名称后立刻关闭 streamReader 对象。原因是你如果关闭了它，你将获取不到任何数据。如果你在关闭 streamWriter 之前关闭 streamReader，你可以向套接字发送数据，但没有数据通过这个通道，因为它已经关闭了。

服务器总结

在进行更多的练习例子之前，我们先回顾一下创建服务器套接字的步骤：

1. 通过需要监听的端口号来实例化一个 ServerSocket 对象，如果有问题可能会抛出一个异常
2. 调用 accept 方法来等待获取连接
3. 获取套接字的流来进行读写
4. 为简化工作，对流进行包装
5. 对套接字进行读写
6. 关闭打开的流，记住不要在关闭 Writer 对象钱关闭 Reader。

你可以在 code listing for`RemoteFileServer`.找到完整的 RemoteFileServer 代码。

多线程的例子

简介

上一个例子讲解了套接字的基础知识，它并没有深入下去。如果你仅仅掌握这些，你只能实现单客户端

连接。原因是 handleConnection 是一个阻塞方法。只有当前连接的事物处理完成后，才允许服务器连接其他客户端。多数情况下你需要支持多线程的服务器。

要使 RemoteFileServer 同时支持多个客户端，不需要进行太多的改动。事实上，我们之前讨论过待处理事项，只需要改动一个方法，虽然改动需要我们创建新的对象来接受传入的连接。我们也会为你展示服务器端套接字如何处理很多等待的连接来使用我们的服务器。这个示例讲述了一个效率低的多线程使用，所以耐心一些。

接收连接

我们修改一下 acceptConnection 方法。它将创建一个 ServerSocket 对象来处理请求，并实现连接。

```
public void acceptConnections() {
       try {
       ServerSocket server = new ServerSocket(listenPort, 5);
       Socket incomingConnection = null;
       while (true) {
          incomingConnection = server.accept();
          handleConnection(incomingConnection);
       }
   } catch (BindException e) {
   System.out.println("Unable to bind to port " + listenPort);
   } catch (IOException e) {
   System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);
   }
}
```

我们新的服务器连接仍旧需要 acceptConnection 方法，所以这段代码实质上是相同的。高亮显示的代码指示出明显的不同。对于多线程版本，在实例化一个 ServerSocket 对象时，我们给它指定客户端连接的最大数。如果我们不指定的话，默认客户端连接数是 50

下面是它运行原理。假定我们指定了 5 个连接数，即同时有 5 个客户端可以连接到我们的服务器。我们的服务器将开始处理第一个连接，但它需要很长时间。因为我们的我们有 5 个候选连接，我们可以一次提交 5 个请求。我们在处理一个，因此有另外 5 个在等待。也就是有 6 个要么在等待，要么在处理。如果第 7 请求连接，但我们的服务器仍然在忙于处理已经连接的资源，（记住 2-6 还在队列中呢），第 7 个连接将被拒绝。我们将在服务器示例中阐述如何在客户端限制同时进行连接的数量。

处理连接，第一部分

这里我们讨论 handleConnection 方法的结构，它将产生一个新的线程来处理连接。我们将通过两个部分来讨论。这一节我们重点关注方法本身，接下来会讨论使用到的 ConnectionHandler 帮助类。

```
public void handleConnection(Socket connectionToHandle) {
    new Thread(new ConnectionHandler(connectionToHandle)).start();
}
```

这个方法和 RemoteFileServer 比有了很大的变化。在服务器接受一个连接后我们仍然叫 handleConnection，但现在我们将 Socket 对象传给 ConnectionHandler 的一个实例，这个实例是多线程的。通过 ConnectionHandler，我们创建了一个新的 Thread 对象并启动起来。ConnectionHandler 的 run 方法含有 Socket 对象的读写和文件的读代码，这些代码在 RemoteFileServer 的 handleConnection 方法中用过。

处理连接：第二部分

这是 ConnectionHandler 类的结构：

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

public class ConnectionHandler implements Runnable{
  Socket socketToHandle;

  public ConnectionHandler(Socket aSocketToHandle) {
    socketToHandle = aSocketToHandle;
  }

  public void run() {
  }
}
```

这个助手类相当的简单。像我们其他类一样，首先导入 java.net 和 java.io 包。这个类只有一个实例变量，socketToHandle，它存放了这个实例要操作的 Socket。

这个类的构造方法将一个 Socket 类型的对象参数传给 socketToHandle。

注意，这个类实现了 Runable 接口，实现这个接口的类必须实现 run()方法。稍后我们会讨论 run 方法的细节。现在你会知道，处理连接的代码和先前在 RemoteFileServer 类中使用的一样。

实现 run()方法

这里我们实现 run 方法。通过这个方法我们获取连接的流，通过流来实现对连接的读写，并在读写完毕后关闭它。

```
  public void run() {
    try {
      PrintWriter streamWriter =
        new PrintWriter(socketToHandle.getOutputStream());
      BufferedReader streamReader =
        new BufferedReader(new
          InputStreamReader(socketToHandle.getInputStream()));

      String fileToRead = streamReader.readLine();
      BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead));

      String line = null;
      while ((line = fileReader.readLine()) != null)
        streamWriter.println(line);

      fileReader.close();
      streamWriter.close();
      streamReader.close();
    } catch (Exception e) {
      System.out.println("Error handling a client: " + e);
    }
```

```
}
```

ConnectionHandler 的 run()方法实现了 RemoteFileServer 的 handleConnection 方法的功能。首先，我们通过 BufferReader 和 PrintWriter 封装了 InputStream 和 OutputStream（各自通过 getOutputStream()方法和 getInputStream()方法）。接下来我们一行行的读取文件。

```
FileReader fileReader = new FileReader(new File(streamReader.readLine()));
      BufferedReader bufferedFileReader = new BufferedReader(fileReader);
      String line = null;
      while ((line = bufferedFileReader.readLine()) != null) {
          streamWriter.println(line);
      }
```

记住，我们应该从客户端获取一个有效的文件路径，然后使用这个路径来新建一个 File 对象。使用 File 对象来新建一个 FIleReader 对象来读取文件，然后将 FileReader 对象来创建一个 BufferReader 对象从而允许我们能在一个 while 循环中进行行的读取。注意，在没有字节流进入时，readLine 方法会阻塞。一旦我们获取到数据，我们将其放入变量 line 中，然后将它们通过客户端写出去。当我们完成读写时，将打开的流关闭掉。

多线程服务器总结

我们的多线程服务器到此结束，在我们进行缓存示例前，先回顾一下创建和使用多线程版本服务器的步骤。

1. 修改 acceptConnection 方法，使用 50 个默认的连接数目来实例化一个 ServerSocket 对象。连接数目可以任意指定，但需大于 1。
2. 修改 handleConnection 方法，通过 ConnectionHandler 对象来创建新的 Thread 线程。
3. 实现 ConnectionHandler 类，借用 RemoteFileServer 的 handleConnection 方法的代码。

你可以在 Code listing for`MultithreadedRemoteFileServer` 获取 MultithreadedRemoteFileServer 的代码列表。ConnectionHandler 的完整代码在 Code listing for `ConnectionHandler` 。

缓冲池示例

介绍

接下来我们介绍的的 MultithreadedServer 会在客户端请求连接时在一个新的线程中创建 ConnectionHandler。这就意味着有一堆线程会处于等待状态。创建一个线程对于系统来说并不是微不足道的。如果性能是系统考虑的主要因素，则构建一个高效的服务器是很可取的。我们怎样管理服务器才能更有效呢？我们可以管理一个有限制数目的池来存放即将进来的连接，这样的设计会提供如下优点：

- 它限制了同时可以进行连接的数目。
- 我们只需要启动 ConnectionHandler 线程一次。

幸运的是，在多线程示例的基础上添加连接池的代码并不需要太大的变动。实际上客户端的代码不需要任何变动。在服务端，在服务器启动时我们创建一定数量的 ConnectionHandler 对象，将传入的连接放到连接池中，并使其他线程监听剩余的连接。这里有很多我们没有涉及到的细节。例如，我们应该拒绝超过连接池建立时设定的数目的连接。

注意：我们不会涉及 acceptConnection 方法。

Creating the `PooledRemoteFileServer` class 创建 PooledRemoteFileServer 类

这是 PooledRemoteFileServer 类的结构

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

public class PooledRemoteFileServer {
   protected int maxConnections;
   protected int listenPort;
   protected ServerSocket serverSocket;

   public PooledRemoteFileServer(int aListenPort, int maxConnections) {
      listenPort = aListenPort;
      this.maxConnections = maxConnections;
   }
   public static void main(String[] args) {
   }
   public void setUpHandlers() {
   }
   public void acceptConnections() {
   }
   protected void handleConnection(Socket incomingConnection) {
   }
}
```

到目前 import 语句应该非常熟悉了。我们创建了一下实例变量：

- 我们服务器可以同时接受的最大客户端数量。
- 接入连接的端口。
- ServerSocket，将接受客户的连接请求。

构造方法使用端口和最大连接数量。

我们的方法有一个 main 方法和其他三个方法。随后我们会详细介绍。现在我们仅仅知道 setUpHandlers 方法创建了和 maxConnections 同样数目的 PooledConnectionHandler 实例，其他两个方法和先前我们介绍的一样，acceptConnection 方法监听连接的客户端，handleConnection 方法处理每个创建完成的连接。

实现 main 方法

这里我们实现修改后的 main 方法，它将创建一个 PooledRemoteServer 来处理指定数目的客户端连接，并调用它接受连接的方法。

```
public static void main(String[] args) {
   PooledRemoteFileServer server = new PooledRemoteFileServer(3000, 3);
   server.setUpHandlers();
   server.acceptConnections();
}
```

main 方法很简单。实例化一个 PooledRemoteServer 对象，该对象通过调用 setUpHandlers 方法创建三个 PooledConnectionHandler 对象。一旦对象创建完毕，调用 acceptConnection 方法告诉它开始接受连接。

设置连接处理程序

```
public void setUpHandlers() {
   for (int i = 0; i < maxConnections; i++) {
      PooledConnectionHandler currentHandler = new PooledConnectionHandler();
      new Thread(currentHandler, "Handler " + i).start();
   }
}
```

setUpHandlers 方法创建了 maxConnections 个 PooledConnectionHandler，并在新的线程中启动它。使用一个继承了 Runnable 接口的对象作为参数来创建 Thread 对象允许我们调用 Thread 的 start 方法，并期望 Runnable 的 run 方法被调用。换句话说，我们的 PooledConnectionHandler 将等待来处理请求的连接。这种处理是在它自己的线程中进行。在我们的示例中仅创建了 3 个线程，并且一旦服务器开始允许这个数量不可以被更改。

处理连接

这里我们实现修改后的 handleConnection 方法，它将代理处理一个连接到 PooledConnectionHandler.

```
protected void handleConnection(Socket connectionToHandle) {
PooledConnectionHandler.processRequest(connectionToHandle);
}
```

现在我们要求 PooledConnectionHandler 来处理所有接入的连接，processRequest 方法是个静态方法。
这是 PooledConnectionHandler 类的结构：

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

public class PooledConnectionHandler implements Runnable {
   protected Socket connection;
   protected static List pool = new LinkedList();

   public PooledConnectionHandler() {
   }
   public void handleConnection() {
   }
   public static void processRequest(Socket requestToHandle) {
   }
   public void run() {
   }
}
```

这个助手类和 ConnectionHandler 非常像，但是有很多处理连接池的方法。这个类有两个独立的实例变量：

- connection，目前被调用到的 Socket
- 一个静态的 LinkedList 对，称为 pool，存放需要被连接的 connection。

填充连接池

这里我们实现 PooledConnectionHandler 类的 processRequest 方法。它将接入的请求加入到池中，并通知

其他对象等待此连接池，因为它现在有内容。

```
public static void processRequest(Socket requestToHandle) {
   synchronized (pool) {
      pool.add(pool.size(), requestToHandle);
      pool.notifyAll();
   }
}
```

这个方法需要一些 java 关键字 synchronized 如何工作的一些背景知识。我们将尝试一个线程方面的简短课程。

首先，一些定义：

- **原子方法**（或代码块），在执行时不可以被打断的。
- **互斥锁。**一个单独的“锁”，当客户端希望执行原子方法时必须获取。

所以，若 A 对象想使用对象 B 中带有 synchronized 关键字的方法 doSomething 时，必须从对象 B 获取互斥锁。也就是说，当 A 获取到了互斥锁后，其他任何对象都不可以调用对象 B 的带有 synchronized 关键字的方法。

带有 synchronized 关键字的阻塞是一个不同的脆弱的动物。你可以同步任何对象，不止是含有阻塞方法的对象。在我们的示例中，processRequest 方法含有一个 synchronized 阻塞的 pool 对象（注意它是一个存放需要处理的连接的 LinkedList 对象）。我们做这些的理由是同一时间不可以有多个对象修改连接池。

现在我们确保只有一个对象可以操作连接池，可以将进入的连接添加到 LinkedList 对象的末尾。一旦将新的连接添加完毕，使用如下代码通知其他进程来访问连接池，告诉它们连接池已经就绪了。

```
pool.notifyAll();
```

所有继承 Object 的子类都继承了 notifyAll 方法。这个方法和我们接下来要讨论的 wait 方法结合来通知其他线程一些条件已经满足。也就是说其他线程在等待这个条件满足。

从连接池中获取连接

这里我们实现 PooledConnectionHandler 类修改后的 run 方法，它将等待连接池，一旦连接池有了连接它立即处理。

```
public void run() {
       while (true) {
           synchronized (pool) {
               while (pool.isEmpty()) {
                   try {
                       pool.wait();
                   } catch (InterruptedException e) {
                       return;
                   }
                }
                connection = (Socket) pool.remove(0);
           }
           handleConnection();
       }
```

```
}
```

前面的部分讲到，一个线程等待连接池满足一定条件的通知。在我们的示例中，在连接池中有三个线程等待连接池中的连接。每个 PooledConnectionHandler 在自己的线程中运行，并通过调用 pool 的 wait 方法被阻塞。当 processRequest 方法调用 notifyAll 方法时，所有的 PooledConnectionHandler 对象接到通知，连接池已经就绪了。每一个线程继续调用 wait 方法，并且判断 while 循环的条件。因为连接池中只有一个连接，所以连接池会变空，其他线程继续等待。第一个获取非空连接池的线程将跳出 while 循环，并将从连接池中获取第一个连接。

```
connection = (Socket) pool.remove(0);
```

一旦处理线程获取了可用的连接，它调用 handleConnection 方法来进行处理。

在我们示例中，连接池可能不会获取大于一条的连接，因为处理速度太快了。若在连接池中有多余一条的连接，其他的现场则不需要等待新的连接添加到池中。当判断 whil 循环条件是，pool 已经非空，线程会获取一个连接来进行处理。

还有一个事情需要注意，当 run 方法拥有一个互斥锁，processRequest 方法是如何将连接放入 pool 的呢？原因是调用 wait 方法会释放锁，并在它返回时重新获取锁。这就允许其他代码同步连接池对象并获取锁。

处理连接：多次

我们实现修改后的 handleConnection 方法，它将抓取连接上的流，使用它们并在结束后清除掉。

```
public void handleConnection() {
   try {
      PrintWriter streamWriter = new PrintWriter(connection.getOutputStream());
      BufferedReader streamReader =
         new BufferedReader(new InputStreamReader(connection.getInputStream()));

      String fileToRead = streamReader.readLine();
      BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead));

      String line = null;
      while ((line = fileReader.readLine()) != null)
         streamWriter.println(line);

      fileReader.close();
      streamWriter.close();
      streamReader.close();
   } catch (FileNotFoundException e) {
      System.out.println("Could not find requested file on the server.");
   } catch (IOException e) {
      System.out.println("Error handling a client: " + e);
   }
}
```

不像多线程版本的服务器，PooledConnectionHandler 有一个 handlerConnection 方法。这里面的代码和非连接池版本的 run 方法的代码一样。首先，我们 OutputStream 和 InputStream 封装到 PrintWriter 和 BufferReader。然后按行读取目标文件，和多线程版本一样。当获取一些字节是，将它们放入本地变量 line

中，并将它们写到客户端。当读写结束后，我们关闭 FileReader 和打开的流。

连接池版本服务器总结。

连接池版本的服务器到此结束。让我们回顾一下创建和使用它的步骤：

1、 创建一个新的类型的连接线程来处理池中的连接

2、 修改这个服务器来创建和使用 PooledConnectionHandler 的集。

你可以在这里得到完整的代码列表。Code listing for PooledRemoteFileServer

现实中的 Socket

介绍

到目前为止我们讨论的示例都是在 java 语言层面的，但是如何在现实中应用呢？套接字的简单使用甚至多线程和连接池版本可能在很多应用中都不适合。你应该在其他类中更聪明的使用套接字来处理你遇到的问题。

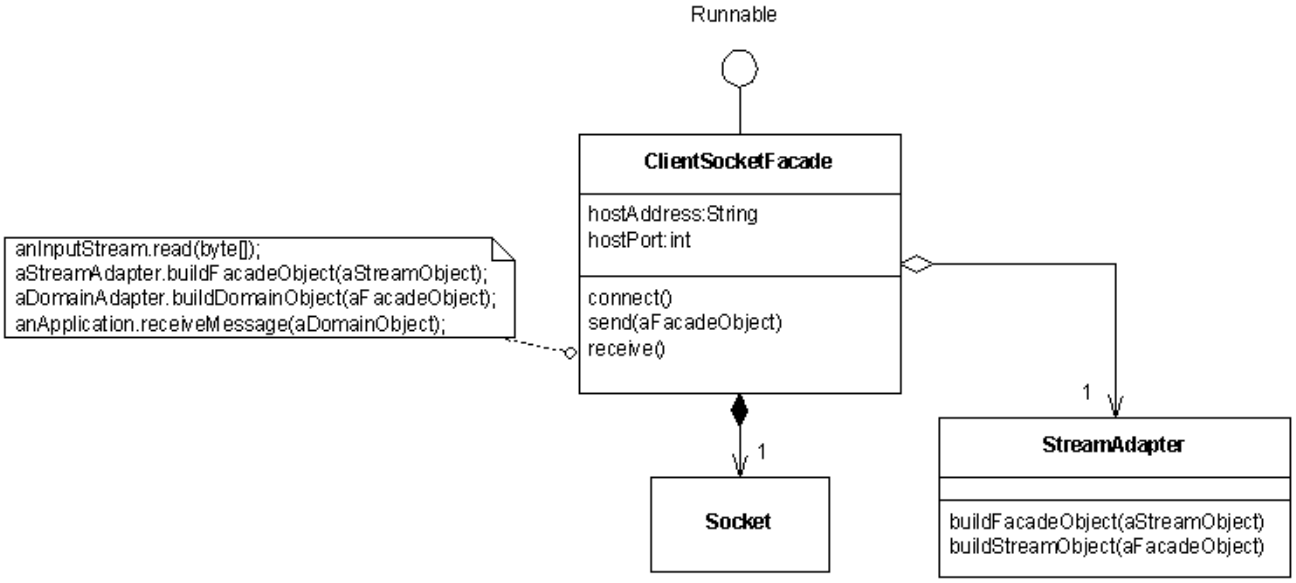
我们最近在做一项工作，就是将应用从大型机环境到 TCP/IP 环境的迁移。这个应用的目的是满足零售出口和财务统计之间的通讯。我们的应用程序是中介。同样的，需要在零售的一个出口和财务的出口之间进行通讯。我们需要通过 Socket 处理客户端和服务端的通讯。同样我们需要将应用的对象转换成套接字可以处理的数据类型进行传输。

我们不能在这份指南中覆盖所有细节，但让我们开始了解一些高层面的东西。你可以将这些东西迁移到你自己遇到的问题领域。

客户端

在客户端，系统的关键角色是 Socket，ClientSocketFacade 和 StreamAdapter。下面图形展示了其 UML 图：

客户端 UML



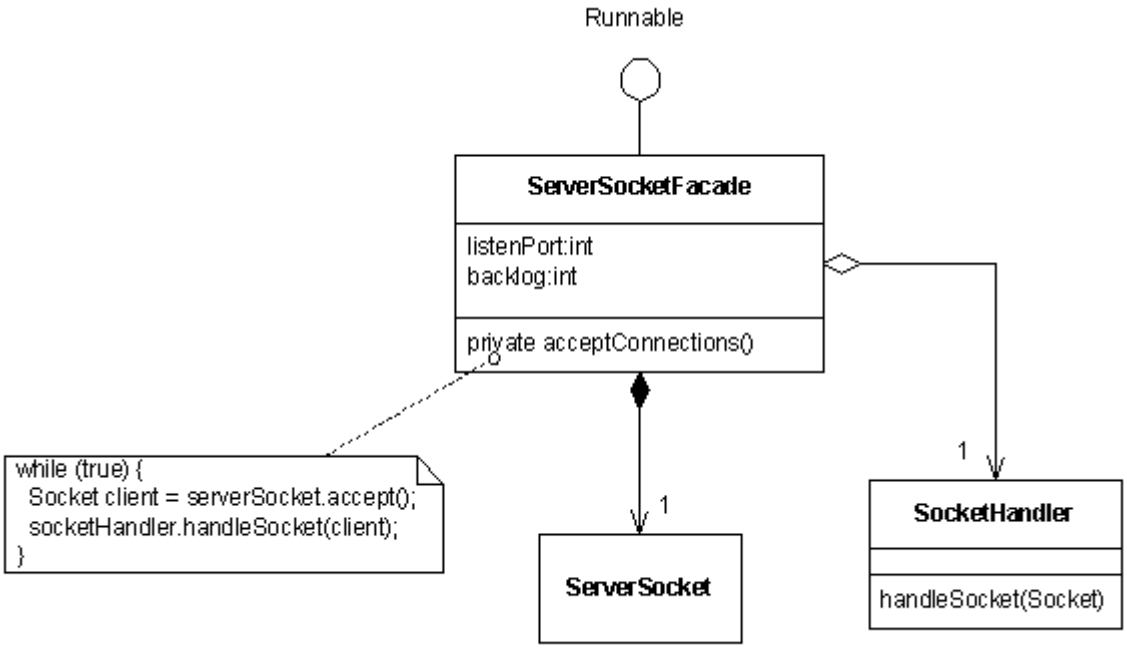
我们创建了一个实现了 Runnable 和 Socket 的 ClientSocketFacade 类。应用程序可以通过主机 IP，端口等实例化一个 ClientSocketFacade 类并让它在一个新的线程中运行。ClientSocketFacade 中的 run 方法对 Socket 实现了懒加载。因为 Socket 的实例已经完成，它调用自己的 receive 方法，这个方法会一直阻塞直到有服务器通过 Socket 来发送消息。

服务器不论何时发来数据，我们的 ClientSocketFacade 类都会唤醒并处理传入的数据。传输数据是直连方式的。我们的应用程序可以通过调用 ClientSocketFacade 的 send 方法来发送一个 StreamObject。

上面我们唯一没有讨论到的是 StreamAdapter。当一个应用调用 ClientSocketFacade 来发送数据时，代理模式会通知 StreamAdapter 的一个实例。ClientSocketFacade 代理发送数据到相同的 StreamAdapter 实例。一个 StreamAdapter 实例将最终的消息放到 Socket 的 OutputStream 中，并转化从 Socket 中接收到的 InputStream。例如，服务器需要获取已发送的数据的字节数。StreamAdapter 可以在发送消息前进行计算和预估消息。当服务器接收后，同样的 StreamAdapter 可以处理真实的长度并且读取正确的字节数来组建 StreamReadyObject。

服务器端

服务器端的图片很相似：



我们在 ServerSocketFacade 中对 ServerSocket 进行了封装，ServerSocketFacade 实现了 Runnable 接口并且拥有一个 ServerSocket 的实例。我们的应用程序可以通过制定特定的服务器端口和最大连接数（默认是 50）来初始化一个 ServerSocketFacade。应用程序可以在一个新的进程中运行来隐藏 ServerSocket 的交互细节。

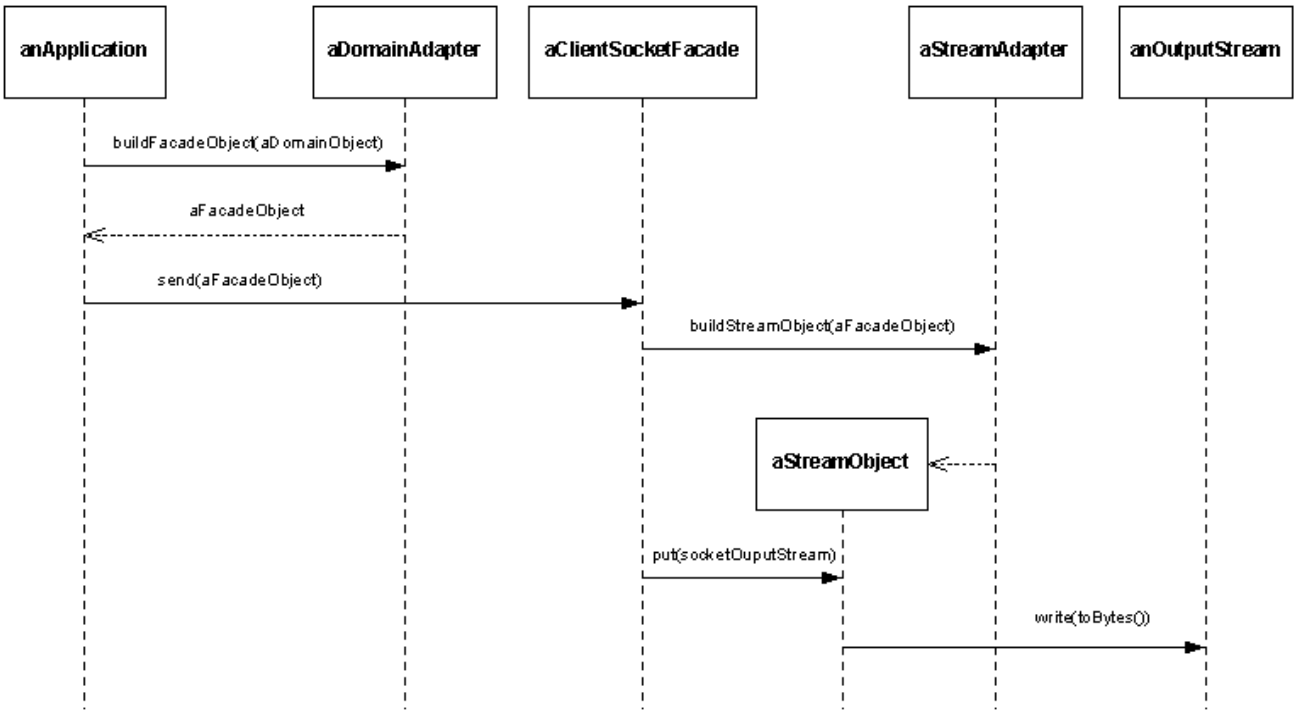
ServerSocketFacade 中的 run 方法调用 acceptConnections 方法，acceptConnections 方法首先创建一个新的 ServerSocket 实例并调用 accept 方法。Accept 方法会一直阻塞直到有客户端申请一个新的连接。每次发生时，ServerSocketFacade 会被唤醒，并且通过调用 SocketHandler 实例的 handleSocket 方法处理 accept 方法返回的 Socket 对象。为了处理从客户端到服务器的新通道，SocketHandler 对象做了它应该做的。

业务逻辑

一旦我们将 Socket 部署好，来实现应用程序的业务逻辑就变得非常简单。应用程序通过使用 ClientSocketFacade 的实例来经由 Socket 发送数据到服务器并且获取反馈。应用程序负责将不同格式的数据转换为 ClientSocketFacade 的格式，并从响应中构建应用程序识别的数据。

向服务器发送消息

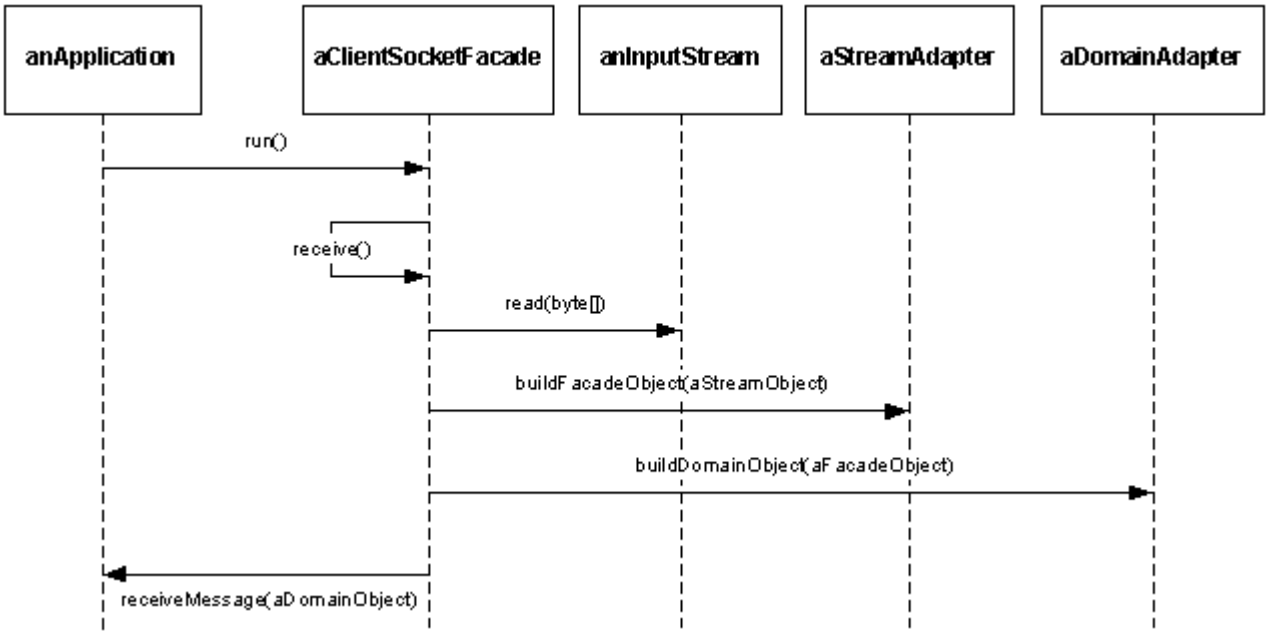
下面是应用程序发送消息的 UML 图



简单起见，我们没有展示 ClientSocketFacade 实例如何请求 Socket 实例获取 OutputStream(使用 getOutputStream 方法)的。一旦我们了解了 OutputStream 会非常简单的了解这个 UML 图。注意 ClientSocketFacade 隐藏了 Socket 和我们的应用程序的底层实现细节。我们的应用和 ClientSocketFacade 对象的实例交互，而不是任何低层次的类。

接收消息

下面是我们的应用接收消息的 UML 图



注意我们的程序在一个线程中运行着 ClientSocketFacade 的一个实例。当 ClientSocketFacade 启动后，receive 方法会监听 Socket 实例的 InputStream。Receive 方法调用 InputStream 的 read(byte[])方法。Read(byte[])方法会一直阻塞直到它收到数据，并将从 InputStream 接收到的字节流组成字节数组。当数据进入时，aClientSocketFacade 实例使用 aStreamAdapter 和 aDomainAdapter 来构建一个本地域对象供应用程序使用。

这时，域对象又回来了。再重复一遍，ClientSocketFacade 隐藏了程序的的底层实现细节。

总结

总结

Java 语言简化了应用程序中使用 Socket 的步骤。最基本的是 java.net 包中的 Socket 和 ServerSocket 两个类。一旦你了解了背景知识，这些类非常容易使用。在现实生活中使用 Socket 就是使用面向对象的设计原则在应用程序中封装不同层次的数据的良好诠释。我们展示了一些对你有帮助的类。这些类的结构都隐藏掉了 Socket 底层的细节实现—它可以使用插件似的 ClientSocketFacades 和 ServerSocketFacades 类。你仍然需要在某些地方处理一些凌乱的 byte 细节，但你可以只做一次。更好的是，你可以在未来的项目中复用底层的帮助类。

附录

Code listing for `URLClient`

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

public class URLClient {
   protected HttpURLConnection connection;
   public String getDocumentAt(String urlString) {
      StringBuffer document = new StringBuffer();
      try {
         URL url = new URL(urlString);
         URLConnection conn = url.openConnection();
         BufferedReader reader =
           new BufferedReader(new InputStreamReader(conn.getInputStream()));

         String line = null;
         while ((line = reader.readLine()) != null)
            document.append(line + "\n");

         reader.close();
      } catch (MalformedURLException e) {
         System.out.println("Unable to connect to URL: " + urlString);
      } catch (IOException e) {
         System.out.println("IOException when connecting to URL: " + urlString);
      }

      return document.toString();
   }
   public static void main(String[] args) {
      URLClient client = new URLClient();
      String yahoo = client.getDocumentAt("http://www.yahoo.com");

      System.out.println(yahoo);
   }
}
```

Code listing for `RemoteFileClient`

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

public class RemoteFileClient {
   protected BufferedReader socketReader;
   protected PrintWriter socketWriter;
```

```
protected String hostIp;
protected int hostPort;

public RemoteFileClient(String aHostIp, int aHostPort) {
   hostIp = aHostIp;
   hostPort = aHostPort;
}
public String getFile(String fileNameToGet) {
   StringBuffer fileLines = new StringBuffer();

   try {
      socketWriter.println(fileNameToGet);
      socketWriter.flush();

      String line = null;
      while ((line = socketReader.readLine()) != null)
         fileLines.append(line + "\n");
   } catch (IOException e) {
      System.out.println("Error reading from file: " + fileNameToGet);
   }

   return fileLines.toString();
}
public static void main(String[] args) {
   RemoteFileClient remoteFileClient = new RemoteFileClient("127.0.0.1", 3000);
   remoteFileClient.setUpConnection();
   String fileContents =
     remoteFileClient.getFile("C:\\WINNT\\Temp\\RemoteFile.txt");
   remoteFileClient.tearDownConnection();

   System.out.println(fileContents);
}
public void setUpConnection() {
   try {
      Socket client = new Socket(hostIp, hostPort);

      socketReader =
        new BufferedReader(new InputStreamReader(client.getInputStream()));
      socketWriter = new PrintWriter(client.getOutputStream());

   } catch (UnknownHostException e) {
      System.out.println("Error setting up socket connection:
        unknown host at " + hostIp + ":" + hostPort);
   } catch (IOException e) {
```

```
            System.out.println("Error setting up socket connection: " + e);
        }
    }
    public void tearDownConnection() {
        try {
            socketWriter.close();
            socketReader.close();
        } catch (IOException e) {
            System.out.println("Error tearing down socket connection: " + e);
        }
    }
}
```

Code listing for `RemoteFileServer`

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

public class RemoteFileServer {
    int listenPort;
    public RemoteFileServer(int aListenPort) {
        listenPort = aListenPort;
    }
    public void acceptConnections() {
        try {
            ServerSocket server = new ServerSocket(listenPort);
            Socket incomingConnection = null;
            while (true) {
                incomingConnection = server.accept();
                handleConnection(incomingConnection);
            }
        } catch (BindException e) {
            System.out.println("Unable to bind to port " + listenPort);
        } catch (IOException e) {
            System.out.println("Unable to instantiate a
              ServerSocket on port: " + listenPort);
        }
    }
    public void handleConnection(Socket incomingConnection) {
        try {
            OutputStream outputToSocket = incomingConnection.getOutputStream();
            InputStream inputFromSocket = incomingConnection.getInputStream();

            BufferedReader streamReader =
              new BufferedReader(new InputStreamReader(inputFromSocket));
```

```
            FileReader fileReader = new FileReader(new File(streamReader.readLine()));

            BufferedReader bufferedFileReader = new BufferedReader(fileReader);
            PrintWriter streamWriter =
              new PrintWriter(incomingConnection.getOutputStream());
            String line = null;
            while ((line = bufferedFileReader.readLine()) != null) {
               streamWriter.println(line);
            }

            fileReader.close();
            streamWriter.close();
            streamReader.close();
         } catch (Exception e) {
            System.out.println("Error handling a client: " + e);
         }
      }
      public static void main(String[] args) {
         RemoteFileServer server = new RemoteFileServer(3000);
         server.acceptConnections();
      }
}
```

Code listing for `MultithreadedRemoteFileServer`

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

public class MultithreadedRemoteFileServer {
   protected int listenPort;
   public MultithreadedRemoteFileServer(int aListenPort) {
      listenPort = aListenPort;
   }
   public void acceptConnections() {
      try {
         ServerSocket server = new ServerSocket(listenPort, 5);
         Socket incomingConnection = null;
         while (true) {
            incomingConnection = server.accept();
            handleConnection(incomingConnection);
         }
      } catch (BindException e) {
         System.out.println("Unable to bind to port " + listenPort);
      } catch (IOException e) {
```

```
            System.out.println("Unable to instantiate a
              ServerSocket on port: " + listenPort);
        }
    }
    public void handleConnection(Socket connectionToHandle) {
        new Thread(new ConnectionHandler(connectionToHandle)).start();
    }
    public static void main(String[] args) {
        MultithreadedRemoteFileServer server =
          new MultithreadedRemoteFileServer(3000);
        server.acceptConnections();
    }
}
```

Code listing for `ConnectionHandler`

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

public class ConnectionHandler implements Runnable {
    protected Socket socketToHandle;
    public ConnectionHandler(Socket aSocketToHandle) {
        socketToHandle = aSocketToHandle;
    }
    public void run() {
        try {
            PrintWriter streamWriter =
              new PrintWriter(socketToHandle.getOutputStream());
            BufferedReader streamReader =
              new BufferedReader(new InputStreamReader(socketToHandle.getInputStream()));

            String fileToRead = streamReader.readLine();
            BufferedReader fileReader =
              new BufferedReader(new FileReader(fileToRead));

            String line = null;
            while ((line = fileReader.readLine()) != null)
                streamWriter.println(line);

            fileReader.close();
            streamWriter.close();
            streamReader.close();
        } catch (Exception e) {
            System.out.println("Error handling a client: " + e);
        }
```

```
    }
}
```

Code listing for `PooledRemoteFileServer`

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

public class PooledRemoteFileServer {
    protected int maxConnections;
    protected int listenPort;
    protected ServerSocket serverSocket;
    public PooledRemoteFileServer(int aListenPort, int maxConnections) {
        listenPort = aListenPort;
        this.maxConnections = maxConnections;
    }
    public void acceptConnections() {
        try {
            ServerSocket server = new ServerSocket(listenPort, 5);
            Socket incomingConnection = null;
            while (true) {
                incomingConnection = server.accept();
                handleConnection(incomingConnection);
            }
        } catch (BindException e) {
            System.out.println("Unable to bind to port " + listenPort);
        } catch (IOException e) {
            System.out.println("Unable to instantiate a
              ServerSocket on port: " + listenPort);
        }
    }
    protected void handleConnection(Socket connectionToHandle) {
        PooledConnectionHandler.processRequest(connectionToHandle);
    }
    public static void main(String[] args) {
        PooledRemoteFileServer server = new PooledRemoteFileServer(3000, 3);
        server.setUpHandlers();
        server.acceptConnections();
    }
    public void setUpHandlers() {
        for (int i = 0; i < maxConnections; i++) {
            PooledConnectionHandler currentHandler =
              new PooledConnectionHandler();
            new Thread(currentHandler, "Handler " + i).start();
```

```
      }
   }
}
```

Code listing for `PooledConnectionHandler`

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

public class PooledConnectionHandler implements Runnable {
   protected Socket connection;
   protected static List pool = new LinkedList();
   public PooledConnectionHandler() {
   }
   public void handleConnection() {
      try {
         PrintWriter streamWriter = new PrintWriter(connection.getOutputStream());
         BufferedReader streamReader =
           new BufferedReader(new InputStreamReader(connection.getInputStream()));

         String fileToRead = streamReader.readLine();
         BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead));

         String line = null;
         while ((line = fileReader.readLine()) != null)
            streamWriter.println(line);

         fileReader.close();
         streamWriter.close();
         streamReader.close();
      } catch (FileNotFoundException e) {
         System.out.println("Could not find requested file on the server.");
      } catch (IOException e) {
         System.out.println("Error handling a client: " + e);
      }
   }
   public static void processRequest(Socket requestToHandle) {
      synchronized (pool) {
         pool.add(pool.size(), requestToHandle);
         pool.notifyAll();
      }
   }
   public void run() {
      while (true) {
```

```
            synchronized (pool) {
                while (pool.isEmpty()) {
                    try {
                        pool.wait();
                    } catch (InterruptedException e) {
                        return;
                    }
                }
                connection = (Socket) pool.remove(0);
            }
            handleConnection();
        }
    }
}
```