第1章 简 介

如今，人们可以通过电脑来打电话，看电视，与朋友聊天，与其他人玩游戏，甚至可以通过电脑买到你能想到的任何东西，包括从歌曲到SUV[㊀]。计算机程序能够通过互联网相互通信使这一切成为了可能。很难统计现在有多少个人电脑接入互联网，但可以肯定，这个数量增长得非常迅速，相信不久就能达到10亿。除此之外，新的应用程序每天在互联网上层出不穷。随着日益增加的互联网访问带宽，我们可以预见，互联网将会对人们将来的生活产生长远的影响。

那么程序是如何通过网络进行相互通信的呢？本书的目的就是通过在Java编程语言环境下，带领你进入对这个问题的解答之路。Java语言从一开始就是为了让人们使用互联网而设计的，它为实现程序的相互通信提供了许多有用的抽象应用编程接口（*Application Programming Interface*，*API*），这类应用编程接口被称为套接字（*socket*）。

在我们开始探究套接字的细节之前，有必要向读者简单介绍计算机网络和通信协议的整体框架，以使读者能清楚我们的代码将应用的地方。本章的目的不是向读者介绍计算机网络和TCP/IP协议是如何工作的（已经有很多相关内容的教程[㊁㊂㊃㊄㊅]），而是介绍一些基本的概念和术语。

㊀ SUV，英文Sports Utility Vehicles的缩写，中文意思是运动型多用途汽车。—— 译者注

㊁ Comer, Douglas E., *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture* (fourth edition), Prentice-Hall, 2000.

㊂ Comer, Douglas E., and Stevens, David L., *Internetworking with TCP/IP, Volume III: Client-Server Programming and Applications* (Linux/POSIX Sockets Version), Prentice-Hall, 2001.

㊃ Peterson, Larry L., and Davie, Bruce S., *Computer Networks: A Systems Approach* (third edition), Morgan Kaufmann, 2003.

㊄ Stevens, W. Richard, *UNIX Network Programming: Networking APIs: Sockets and XTI* (second edition), Prentice-Hall, 1997.

㊅ Stevens, W. Richard, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, 1994.

1.1 计算机网络、分组报文和协议

计算机网络由一组通过通信信道相互连接的机器组成。我们把这些机器称为主机（*hosts*）和路由器（*routers*）。主机是指运行应用程序的计算机，这些应用程序包括网络浏览器（Web browser）、即时通信代理（IM agent）或者文件共享程序。运行在主机上的应用程序才是计算机网络的真正“用户”。路由器的作用是将信息从一个通信信道传递或转发（*forward*）到另一个通信信道。路由器上可能会运行一些程序，但大多数情况下它们是不运行应用程序的。基于本书的目的对通信信道（*communication channel*）进行解释：它是将字节序列从一个主机传输到另一个主机的一种手段，可能是有线电缆，如以太网（Ethernet），也可能是无线的，如WiFi[⊖]，或是其他方式的连接。

路由器非常重要，因为要想直接将所有不同主机连接起来是不可行的。相反，一些主机先得连接到路由器，这些路由器再连接到其他路由器，这样就形成了网络。这种布局使每个主机只需要用到数量相对较少的通信信道，大部分主机仅需要一条信道。在网络上相互传递信息的程序并不直接与路由器进行交互，它们基本上感觉不到路由器的存在。

这里的信息（*information*）是指由程序创建和解释的字节序列。在计算机网络环境中，这些字节序列称为分组报文（*packet*）。一组报文包括了网络用来完成工作的控制信息，有时还包括一些用户数据。用于定位分组报文目的地址的信息就是一个例子。路由器正是利用了这些控制信息来实现对每个报文的转发。

协议（*protocol*）相当于相互通信的程序间达成的一种约定，它规定了分组报文的交换方式和它们包含的意义。一组协议规定了分组报文的结构（例如报文中的哪一部分表明了其目的地址）以及怎样对报文中所包含的信息进行解析。设计一组协议，通常是为了在一定约束条件下解决某一特定的问题。比如，超文本传输协议（HyperText Transfer Protocol，HTTP）是为了解决在服务器间传递超文本对象的问题，这些超文本对象在服务器中创建和存储，并由Web浏览器进行可视化，以使其对用户有用。即时消息协议是为了使两个或更多用户间能够交换简短的文本信息。

要实现一个有用的网络，必须解决大量各种各样的问题。为了使这些问题可管理和模块化，人们设计了不同的协议来解决不同类型的问题。TCP/IP协议就是这样一组解决

⊖ WiFi，全称Wireless Fidelity，即无线保真，是一种短距离无线技术。—— 译者注

方案，有时也称为协议族（*protocol suite*）。它刚好是互联网所使用的协议，不过也能用在独立的专用网络中。本书以后所提到的网络（*network*）都是指任何使用了TCP/IP协议族的网络。TCP/IP协议族主要协议有IP协议（*Internet Protocol*[⊖]互联网协议），TCP协议（*Transmission Control Protocol*[⊜]，传输控制协议）和UDP协议（*User Datagram Protocol*[⊜]，用户数据报协议）。

事实证明将各种协议分层组织是一种非常有用的措施，TCP/IP协议族（实际上其他所有协议族）都是按这种方式组织的。图1-1展示了主机和路由器中的通信协议、应用程序和套接字API之间的关系，同时也展示了数据流从一个应用程序到另一个应用程序的过程（使用TCP协议）。标记为TCP、UDP和IP的方框分别代表了这些协议的实现，它们通常驻留在主机的操作系统中。应用程序通过套接字API对UDP协议和TCP协议所提供的服务进行访问。箭头描述了数据流从一个应用程序，经过TCP协议层和IP协议层，通过网络，再经过IP协议层和TCP协议层传输到另一端的应用程序。

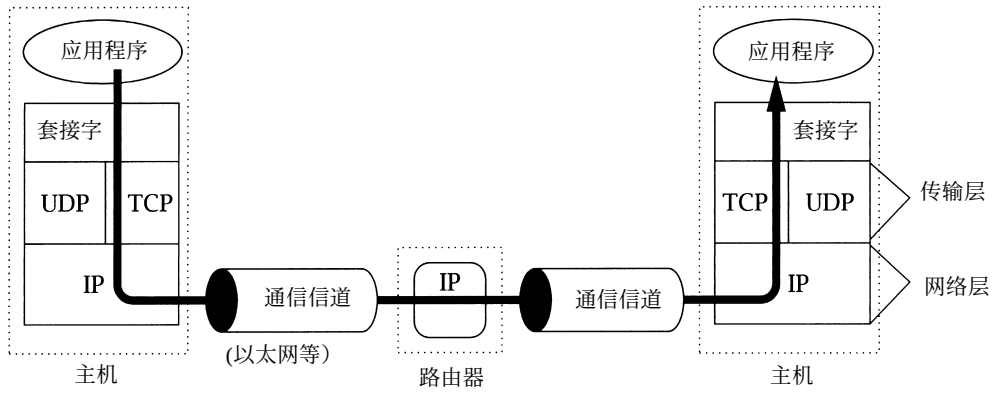


图1-1　一个TCP/IP网络

在TCP/IP协议族中，底层由基础的通信信道构成，如以太网或调制解调器拨号连接。这些信道由网络层（*network layer*）使用，而网络层则完成将分组报文传输到它们的目的地址的工作（也就是路由器的功能）。TCP/IP协议族中属于网络层的唯一协议是IP协议，它使两个主机间的一系列通信信道和路由器看起来像是单独一条主机到

⊖ Postel, John, "Internet Protocol," Internet Request for Comments 791, September 1981.
⊜ Postel, John, "Transmission Control Protocol," Internet Request for Comments 793, September 1981.
⊜ Postel, John, "User Datagram Protocol," Internet Request for Comments 768, August 1980.

主机的信道。

IP协议提供了一种数据报服务：每组分组报文都由网络独立处理和分发，就像信件或包裹通过邮政系统发送一样。为了实现这个功能，每个IP报文必须包含一个保存其目的地址（address）的字段，就像你所投递的每份包裹都写明了收件人地址。（我们随即会对地址进行更详细的说明。）尽管绝大部分递送公司会保证将包裹送达，但IP协议只是一个“尽力而为”（best-effort）的协议：它试图分发每一个分组报文，但在网络传输过程中，偶尔也会发生丢失报文，使报文顺序被打乱，或重复发送报文的情况。

IP协议层之上称为传输层（*transport layer*）。它提供了两种可选择的协议：TCP协议和UDP协议。这两种协议都建立在IP层所提供的服务基础上，但根据应用程序协议（*application protocol*）的不同需求，它们使用了不同的方法来实现不同方式的传输。TCP协议和UDP协议有一个共同的功能，即寻址。回顾一下，IP协议只是将分组报文分发到了不同的主机，很明显，还需要更细粒度的寻址将报文发送到主机中指定的应用程序，因为同一主机上可能有多个应用程序在使用网络。TCP协议和UDP协议使用的地址叫做端口号（*port number*），都是用来区分同一主机中的不同应用程序的。TCP协议和UDP协议也称为端到端传输协议（*end-to-end transport protocol*），因为它们将数据从一个应用程序传输到另一个应用程序，而IP协议只是将数据从一个主机传输到另一主机。

TCP协议能够检测和恢复IP层提供的主机到主机的信道中可能发生的报文丢失、重复及其他错误。TCP协议提供了一个可信赖的字节流（*reliable byte-stream*）信道，这样应用程序就不需要再处理上述的问题。TCP协议是一种面向连接（*connection-oriented*）的协议：在使用它进行通信之前，两个应用程序之间首先要建立一个TCP连接，这涉及相互通信的两台电脑的TCP部件间完成的握手消息（*handshake message*）的交换。使用TCP协议在很多方面都与文件的输入输出（*Input/Output*，*I/O*）相似。实际上，由一个程序写入的文件再由另一个程序读取就是一个TCP连接的适当模型。另一方面，UDP协议并不尝试对IP层产生的错误进行修复，它仅仅简单地扩展了IP协议“尽力而为”的数据报服务，使它能够在应用程序之间工作，而不是在主机之间工作。因此，使用了UDP协议的应用程序必须为处理报文丢失、顺序混乱等问题做好准备。

1.2　关于地址

寄信的时候，要在表格中填上邮政服务能够理解的收信人的地址。在给别人打电话时，必须拨电话号码。同样，一个程序要与另一个程序通信，就要给网络提供足够的信息，使其能够找到另一个程序。在TCP/IP协议中，有两部分信息用来定位一个指定的程序：互联网地址（*Internet address*）和端口号（*port number*）。其中互联网地址由IP协议使用，而附加的端口地址信息由传输协议（TCP或IP协议）对其进行解析。

互联网地址由二进制数字组成，有两种型式，分别对应了两个版本的标准互联网协议。现在最常用的版本是版本4，即IPv4[㊀]，另一个版本是刚开始开发的版本6，即IPv6[㊁]。IPv4的地址长32位，只能区分大约40亿个独立地址，对于如今的互联网来说，这是不够大的。（也许看起来很多，但由于地址的分配方式的原因，有很多都被浪费了）出于这个原因引入了IPv6，它的地址有128位长。

为了便于人们使用互联网地址（相对于程序内部的表示），两个版本的IP协议有不同的表示方法。IPv4地址被表示为一组4个十进制数，每两个数字之间由圆点隔开（如：10.1.2.3），这种表示方法叫做点分形式（*dotted-quad*）。点分形式字符串中的4个数字代表了互联网地址的4个字节，也就是说，每个数字的范围是0～255。

另一方面，IPv6地址的16个字节由几组16进制的数字表示，这些16进制数之间由分号隔开（如：2000:fdb8:0000:0000:0001:00ab:853c:39a1）。每组数字分别代表了地址中的两个字节，并且每组开头的0可以省略，因此前面的例子中，第5组和第6组数字可以缩写为:1:ab:。甚至，只包含0的连续组可以全部省略（但在一个地址中只能这样做一次）。因此，该例子的完整地址可以表示为2000:fdb8::1:00ab:853c:39a1。

从技术角度来讲，每个互联网地址代表了一台主机与底层的通信信道的连接，换句话说，也就是一个网络接口（*network interface*）。主机可以有多个接口，这并不少见，例如一台主机同时连接了有线以太网（Ethernet）和无线网（WiFi）。由于每个这样的连接都属于唯一的一台主机，所以只要它连接到网络，一个互联网地址就能定位这条主机。但是反过来，一台主机并不对应一个互联网地址。因为每台主机可以有多个接口，每个

㊀ Postel, John, "Internet Protocol," Internet Request for Comments 791, September 1981.

㊁ Deering, S., and Hinden, R, "Internet Protocol, Version 6 (IPv6) Specification," Internet Request for Comments 2460, December 1998.

接口又可以有多个地址。（实际上一个接口可以同时拥有IPv4地址和IPv6地址）。

TCP或UDP协议中的端口号总与一个互联网地址相关联。回到前面我们作类比的例子，一个端口号就相当于指定街道上一栋大楼的某个房间号。邮政服务通过街道地址把信分发到一个邮箱，再由清空邮箱的人把这封信递送到这栋楼的正确房间中。或者考虑一个公司的内部电话系统：要与这个公司中的某个人通话，首先要拨打该公司的总机电话号码连接到其内部电话系统，然后再拨打你要找的那个人的分机号码。在上面的例子中，互联网地址就相对于街道地址或公司的总机电话号码，端口号就相当于房间号或分机号码。端口号是一组16位的无符号二进制数，每个端口号的范围是1～65 535（0被保留）。

每个版本的IP协议都定义了一些特殊用途的地址。其中值得注意的一个是回环地址（*loopback address*），该地址总是被分配个一个特殊的回环接口（*loopback interface*）。回环接口是一种虚拟设备，它的功能只是简单地将发送给它的报文直接回发给发送者。回环接口在测试中非常有用，因为发送给这个地址的报文能够立即返回到目标地址。而且每台主机上都有回环接口，即使当这台计算机没有其他接口（也就是说没有连接到网络），回环接口也能使用。IPv4的回环地址是127.0.0.1[⊖]，IPv6的回环地址是0:0:0:0:0:0:0:1。

IPv4地址中的另一种特殊用途的保留地址包括那些“私有用途”的地址。它们包括IPv4中所有以10或192.168开头的地址，以及第一个数是172，第二个数在16～31的地址。（在IPv6中没有相应的这类地址）这类地址最初是为了在私有网络中使用而设计的，不属于公共互联网的一部分。现在这类地址通常被用在家庭或小型办公室中，这些地方通过*NAT*（*Network Address Translation*，网络地址转换）设备连接到互联网。NAT设备的功能就像一个路由器，转发分组报文时将转换（重写）报文中的地址和端口。更准确地说，它将一个接口中报文的私有地址端口对（private address, port pairs）映射成另一个接口中的公有地址端口对（public address, port pairs）。这就使一小组主机（如家庭网络）能够有效地共享同一个IP地址。重要的是这些内部地址不能从公共互联网访问。如果你在拥有私有类型地址的计算机上试验本书的例子，并试图与另一台没有这类地址的主机进行通信，通常只有这台拥有私有类型地址的主机发起的通信才能成功。

相关的类型的地址包括本地链接（*link-local*），或称为“自动配置”地址。IPv4中，

⊖ 从技术上讲，任何由127开头的IPv4地址都应该回环。

这类地址由169.254开头，在IPv6中，前16位由FE8开头的地址是本地链接地址。这类地址只能用来在连接到同一网络的主机之间进行通信，路由器不会转发这类地址的信息。

最后，另一类地址由多播（*multicast*）地址组成。普通的IP地址（有时也称为“单播”地址）只与唯一一个目的地址相关联，而多播地址可能与任意数量的目的地址关联。我们将在第4章中简要地对多播技术作进一步介绍。IPv4中的多播地址在点分格式中，第一个数字在224～239之间。IPv6中，多播地址由FF开始。

1.3　关于名字

也许你更习惯于通过名字来指代一个主机，例如：host.example.com。然而，互联网协议只能处理二进制的网络地址，而不是主机名。首先应该明确的是，使用主机名而不使用地址是出于方便性的考虑，这与TCP/IP提供的基本服务是相互独立的。你也可以不使用名字来编写和使用TCP/IP应用程序。当使用名字来定位一个通信终端时，系统将做一些额外的工作把名字解析成地址。有两个原因证明这额外的步骤是值得的：第一，相对于点分形式（或IPv6中的十六进制数字串），人们更容易记住名字；第二，名字提供了一个间接层，使IP地址的变化对用户不可见。在本书英文版第一版的写作期间，网络服务器www.mkp.com的地址就改变过。由于我们通常都使用网络服务器的名字，而且地址的改变很快就被反应到映射主机名和网络地址的服务上（我们马上会对其进行更多的介绍），如www.mkp.com从之前的地址208.164.121.48对应到了现在的地址，这种变化对通过名字访问该网络服务器的程序是透明的。

名字解析服务可以从各种各样的信息源获取信息。两个主要的信息源是域名系统（*Domain Name System*，*DNS*）和本地配置数据库。DNS[⊖]是一种分布式数据库，它将像*www.mkp.com*这样的域名映射到真实的互联网地址和其他信息上。DNS协议[⊜]允许连接到互联网的主机通过TCP或UDP协议从DNS数据库中获取信息。本地配置数据库通常是一种与具体操作系统相关的机制，用来实现本地名称与互联网地址的映射。

⊖ Mockapetris, Paul, “Domain Names-Concepts and Facilities,” Internet Request for Comments 1034, November 1987.

⊜ Mockapetris, Paul, “Domain Names-Implementation and Specification,” Internet Request for Comments 1035, November 1987.

1.4 客户端和服务器

在前面的邮政和电话系统例子中，每次通信都是由发信方或打电话者发起，而另一方则通过发回反馈信或接听电话来对通信的发起者作出响应。互联网通信也与这个过程类似。客户端（*client*）和服务器（*server*）这两个术语代表了两种角色：客户端是通信的发起者，而服务器程序则被动等待客户端发起通信，并对其作出响应。客户端与服务器组成了应用程序（*application*）。客户端和服务器这两个术语对典型的情况作出了描述，服务器具有一定的特殊能力，如提供数据库服务，并使任何客户端能够与之通信。

一个程序是作为客户端还是服务器，决定了它在与其对等端（*peer*）建立通信时使用的套接字API的形式（客户端的对等端是服务器，反之亦然）。更进一步来说，客户端与服务器端的区别非常重要，因为客户端首先需要知道服务器的地址和端口号，反之则不需要。如果有必要，服务器可以使用套接字API，从收到的第一个客户端通信消息中获取其地址信息。这与打电话非常相似：被呼叫者不需要知道拨电话者的电话号码。就像打电话一样，只要通信连接建立成功，服务器和客户端之间就没有区别了。

客户端如何才能找到服务器的地址和端口号呢？通常情况，客户端知道服务器的名字，例如使用*URL*（*Universal Resource Locator*，统一资源定位符）如*http://www.mkp.com*，再通过名字解析服务获取其相应的互联网地址。

获取服务器的端口号则是另一种情况。从原理上来讲，服务器可以使用任何端口号，但客户端必须能够获知这些端口号。在互联网上，一些常用的端口号被约定赋给了某些应用程序。例如，端口号21被*FTP*（*File Transfer Protocol*，文件传输协议）使用。当你运行FTP客户端应用程序时，它将默认通过这个端口号连接服务器。互联网的端口号授权机构维护了一个包含所有已约定使用的端口号列表（见*http://www.iana.org/assignments/port-numbers*）。

1.5 什么是套接字

socket（套接字）是一种抽象层，应用程序通过它来发送和接收数据，就像应用程序打开一个文件句柄，将数据读写到稳定的存储器上一样。使用socket可以将应用程序添加到网络中，并与处于同一个网络中的其他应用程序进行通信。一台计算机上的应用程序向socket写入的信息能够被另一台计算机上的另一个应用程序读取，反之亦然。

不同类型的socket与不同类型的底层协议族以及同一协议族中的不同协议栈相关联，

本书只涵盖了TCP/IP协议族的内容。现在TCP/IP协议族中的主要socket类型为流套接字（*stream socket*）和数据报套接字（*datagram socket*）。流套接字将TCP作为其端对端协议（底层使用IP协议），提供了一个可信赖的字节流服务。一个TCP/IP流套接字代表了TCP连接的一端。数据报套接字使用UDP协议（底层同样使用IP协议），提供了一个“尽力而为”（best-effort）的数据报服务，应用程序可以通过它发送最长65 500字节的个人信息。当然，其他协议族也支持流套接字和数据报套接字，但本书只对TCP流套接字和UDP数据报套接字进行讨论。一个TCP/IP套接字由一个互联网地址，一个端对端协议（TCP或UDP协议）以及一个端口号唯一确定。随着进一步学习，你将了解到把一个套接字绑定到一个互联网地址上的多种方法。

图1-2描述了一个主机中，应用程序、套接字抽象层、协议、端口号之间的逻辑关系。值得注意的是一个套接字抽象层可以被多个应用程序引用。每个使用了特定套接字的程序都可以通过那个套接字进行通信。前面已提到，每个端口都标识了一台主机上的一个应用程序。实际上，一个端口确定了一台主机上的一个套接字。从图1-2中我们可以看到，主机中的多个程序可以同时访问同一个套接字。在实际应用中，访问相同套接字的不同程序通常都属于同一个应用（例如，Web服务程序的多个副本），但从理论上讲，它们是可以属于不同应用的。

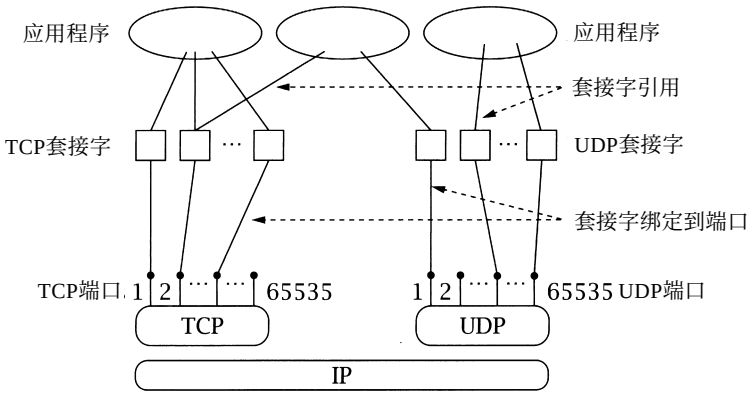


图1-2　套接字、协议、端口

1.6　练习

1. 你能举出一个现实生活中不符合客户/服务器模型的例子吗？

2. 在你家中有多少种不同的网络？其中有多少支持双向数据传输？
3. IP协议是一个“尽力而为”的协议，需要将信息切分成多个数据报。这些数据报有可能在传输的过程中丢失、重复或被打乱顺序。TCP协议将对这些问题的处理都隐藏了起来，从而提供了一个能够传输完整字节流的可信赖服务。如果由你来设计，你会怎样实现在IP协议上提供TCP服务？为什么在已经有了TCP协议的情况下还有人用UDP协议？

第 3 章 发送和接收数据

通常情况下，在程序中使用套接字是因为需要向其他程序提供信息，或使用其他程序提供的信息。这并不是什么魔法：任何要交换信息的程序之间在信息的编码方式上必须达成共识（如将信息表示为位序列），以及哪个程序发送信息，什么时候和怎样接收信息都将影响程序的行为。程序间达成的这种包含了信息交换的形式和意义的共识称为协议，用来实现特定应用程序的协议叫做应用程序协议。前面章节中的回馈程序示例中的应用程序协议都过于简单：客户端和服务器的行为都不受它们之间所交换的信息内容的影响。而在绝大部分实际应用中，客户端和服务器的行为都要依赖于它们所交换的信息，因此应用程序协议通常更加复杂。

TCP/IP协议以字节的方式传输用户数据，并没有对其进行检查和修改。这个特点使应用程序可以非常灵活地对其传输的信息进行编码。大部分的应用程序协议是根据由字段序列组成的离散信息定义的，其中每个字段中都包含了一段以位序列编码的特定的信息。应用程序协议中明确定义了信息的发送者应该怎样排列和解释这些位序列，同时还要定义接收者应该怎样解析，这样才使信息的接收者能够抽取出每个字段的意义。TCP/IP协议的唯一约束是，信息必须在块（chunks）中发送和接收，而块的长度必须是8位的倍数，因此，我们可以认为在TCP/IP协议中传输的信息是字节序列。鉴于此，我们可以进一步把传输的信息看作数字序列或数组，每个数字的取值范围是0到255。这与8位编码的二进制数值范围是一致的：00000000代表0，00000001代表1，00000010代表2，等等，最多到11111111，即255。

如果你建立了一个程序使用套接字与其他程序交换信息，通常符合下面两种情况之一：要么是你设计和编写了套接字的客户端和服务器端，这种情况下你能够随心所欲地定义自己的应用程序协议；要么是你实现了一个已经存在的协议，或许是一个协议标准。任何一种情况，在“线路上”将不同类型的信息进行字节编码和解码的基本原理还是一样的。顺便说一下，如果“线路”是由一个程序写，由另一程序读的文件，本章的所有内容对其也是适用的。

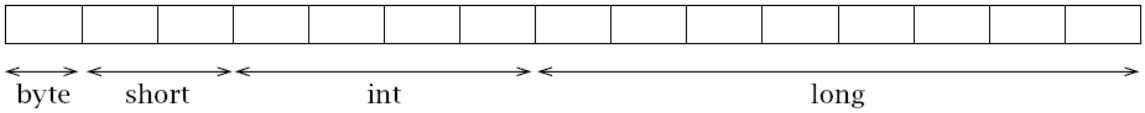
3.1 信息编码

首先，我们来考虑一下简单数据类型，如int，long，char，String等，是如何通过套接字发送和接收的。从前面章节我们已经知道，传输信息时可以通过套接字将字节信息写入一个OutputStream实例中（该实例已经与一个Socket相关联），或将其封装进一个DatagramPacket实例中（该实例将由DatagramSocket发送）。然而，这些操作所能处理的唯一数据类型是字节和字节数组。作为一种强类型语言，Java需要把其他数据类型（int，String等）显式转换成字节数组。所幸的是Java的内置工具能够帮助我们完成这些转换。在第2.2.1节的TCPEchoClient.java示例程序中，我们看到过String类的getBytes()方法，该方法就是将一个Sring实例中的字符转换成字节的标准方式。在考虑数据类型转换的细节之前，我们先来看看大部分基本数据类型的表示方法。

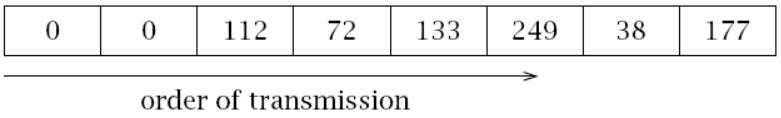
3.1.1 基本整型

如我们所见，TCP和UDP套接字使我们能发送和接收字节序列（数组），即范围在0-255之间的整数。使用这个功能，我们可以对值更大的基本整型数据进行编码，不过发送者和接收者必须先在一些方面达成共识。一是要传输的每个整数的字节大小（*size*）。例如，Java程序中，int数据类型由32位表示，因此，我们可以使用4个字节来传输任意的int型变量或常量；short数据类型由16位表示，传输short类型的数据只需要两个字节；同理，传输64位的long类型数据则需要8个字节。

下面我们考虑如何对一个包含了4个整数的序列进行编码：一个byte型，一个short型，一个int型，以及一个long型，按照这个顺序从发送者传输到接收者。我们总共需要15个字节：第一个字节存放byte型数据，接下来两个字节存放short型数据，再后面4个字节存放int型数据，最后8个字节存放long型数据，如下所示：

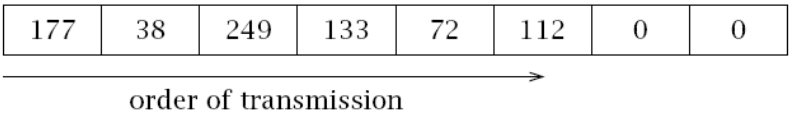


我们已经做好深入研究的准备了吗？未必。对于需要超过一个字节来表示的数据类型，我们必须知道这些字节的发送顺序。显然有两种选择：从整数的右边开始，由低位到高位地发送，即*little-endian*顺序；或从左边开始，由高位到低位发送，即*big-endian*顺序。（注意，幸运的是字节中位的顺序在实现时是以标准的方式处理的）考虑长整型数123456787654321L，其64位（以十六进制形式）表示为0x0000704885F926B1。如果我们以big-endian顺序来传输这个整数，其字节的十进制数值序列就如下所示：



order of transmission：传输顺序

如果我们以little-endian顺序传输，则字节的十进制数组序列为：



order of transmission：传输顺序

关键的一点是，对于任何多字节的整数，发送者和接收者必须在使用big-endian顺序还是使用little-endian顺序上达成共识[1]。如果发送者使用了little-endian顺序来发送上述整数，而接收者以big-endian顺序对其进行接收，那么接收者将取到错误的值，它会将这个8字节序列的整数解析成127651645446695157 76L。

发送者和接收者需要达成共识的最后一个细节是：所传输的数值是有符号的（*signed*）还是无符号的（*unsigned*）。Java中的四种基本整型都是有符号的，它们的值以二进制补码（*two's-complement*）的方式存储，这是有符号数值的常用表示方式。在处理有k位的有符号数时，用二进制补码的形式表示负整数$-n$（$1 \le n \le 2^{k-1}$），则补码的二进制值就为$2^k - n$。而对于非负整数p（$0 \le p \le 2^{k-1} - 1$），只是简单地用k位二进制数来表示p的值。因此，对于给定的k位，我们可以通过二进制补码来表示-2^{k-1}到$2^{k-1}-1$范围的值。注意，最高位（msb）标识了该数是正数（msb = 0）还是负数（msb = 1）。另外，如果使用无符号（*unsigned*）编码，k位可以直接表示0到$2^k - 1$之间的数值。例如，32位数值0xffffffff（所有位全为1），将其解析为有符号数时，二进制补码整数表示-1；将其解析为无符号数时，它表示4294967295。由于Java并不支持无符号整型，如果要在Java中编码和解码无符号数，则需要做一点额外的工作。在此假设我们处理的都是有符号整数数据。

[1] Java中包含了一个ByteOrder类来表示这两种可能的顺序。该类有两个静态字段存放了，存放了其仅有的两个实例：ByteOrder.BIG_ENDIAN和ByteOrder.LITTLE_ENDIAN。第 5 章中将对其作详细介绍。

那么我们怎样才能将消息的正确值存入字节数组呢？为了清楚地展示需要做的步骤，我们将对如何使用“位操作（bit-diddling）”（移位和屏蔽）来显式编码进行介绍。示例程序BruteForceCoding.java中有一个特殊的方法encodeIntBigEndian()能够对任何值的基本类型数据进行编码。它的参数包括用来存放数值的字节数组，要进行编码的数值（表示为long型，它是最长的整型，能够保存其他整型的值），数值在字节数组中开始位置的偏移量，以及该数值写到数组中的字节数。如果我们在发送端进行了编码，那么必须能够在接收端进行解码。BruteForceCoding类同时还提供了decodeIntBigEndian()方法，用来将字节数组的子集解码到一个Java的long型整数中。

BruteForceCoding.java

```
0 public class BruteForceCoding {
1 private static byte byteVal = 101; // one hundred and one
2 private static short shortVal = 10001; // ten thousand and one
3 private static int intVal = 100000001; // one hundred million and one
4 private static long longVal = 1000000000001L;// one trillion and one
5
6 private final static int BSIZE = Byte.SIZE / Byte.SIZE;
7 private final static int SSIZE = Short.SIZE / Byte.SIZE;
8 private final static int ISIZE = Integer.SIZE / Byte.SIZE;
9 private final static int LSIZE = Long.SIZE / Byte.SIZE;
10
11 private final static int BYTEMASK = 0xFF; // 8 bits
12
13 public static String byteArrayToDecimalString(byte[] bArray) {
14 StringBuilder rtn = new StringBuilder();
15 for (byte b : bArray) {
16 rtn.append(b & BYTEMASK).append(" ");
17 }
18 return rtn.toString();
19 }
20
21 // Warning: Untested preconditions (e.g., 0 <= size <= 8)
22 public static int encodeIntBigEndian(byte[] dst, long val, int offset, int size)
{
23 for (int i = 0; i < size; i++) {
24 dst[offset++] = (byte) (val >> ((size - i - 1) * Byte.SIZE));
25 }
26 return offset;
27 }
28
29 // Warning: Untested preconditions (e.g., 0 <= size <= 8)
30 public static long decodeIntBigEndian(byte[] val, int offset, int size) {
31 long rtn = 0;
32 for (int i = 0; i < size; i++) {
33 rtn = (rtn << Byte.SIZE) | ((long) val[offset + i] & BYTEMASK);
34 }
35 return rtn;
36 }
37
38 public static void main(String[] args) {
39 byte[] message = new byte[BSIZE + SSIZE + ISIZE + LSIZE];
40 // Encode the fields in the target byte array
41 int offset = encodeIntBigEndian(message, byteVal, 0, BSIZE);
42 offset = encodeIntBigEndian(message, shortVal, offset, SSIZE);
43 offset = encodeIntBigEndian(message, intVal, offset, ISIZE);
44 encodeIntBigEndian(message, longVal, offset, LSIZE);
45 System.out.println("Encoded message: " + byteArrayToDecimalString(message));
46
47 // Decode several fields
```

```
48 long value = decodeIntBigEndian(message, BSIZE, SSIZE);
49 System.out.println("Decoded short = " + value);
50 value = decodeIntBigEndian(message, BSIZE + SSIZE + ISIZE, LSIZE);
51 System.out.println("Decoded long = " + value);
52
53 // Demonstrate dangers of conversion
54 offset = 4;
55 value = decodeIntBigEndian(message, offset, BSIZE);
56 System.out.println("Decoded value (offset " + offset + ", size " + BSIZE + ")
= "
57 + value);
58 byte bVal = (byte) decodeIntBigEndian(message, offset, BSIZE);
59 System.out.println("Same value as byte = " + bVal);
60 }
61
62 }
```

BruteForceCoding.java

1. **数据项编码：**第 1-4 行

2. **Java 中的基本整数所占字节数：**第 6-9 行

3. byteArrayToDecimalString()：第 13-19 行

该方法把给定数组中的每个字节作为一个无符号十进制数打印出来。BYTEMASK 的作用是防止在字节数值转换成 int 类型时，发生符号扩展（*sign-extended*），即转换成无符号整型。

4.encodeIntBigEndian()：第 22-27 行

赋值语句的右边，首先将数值向右移动，以使我们需要的字节处于该数值的低 8 位中。然后，将移位后的数转换成 byte 型，并存入字节数组的适当位置。在转换过程中，除了低 8 位以外，其他位都将丢弃。这个过程将根据给定数值所占字节数迭代进行。该方法还将返回存入数值后字节数组中新的偏移位置，因此我们不必做额外的工作来跟踪偏移量。

5. decodeIntBigEndian()：第 30-36 行

根据给定数组的字节大小进行迭代，通过每次迭代的左移操作，将所取得字节的值累积到一个 long 型整数中。

6. **示例方法：**第 38-60 行

- **准备接收整数序列的数组：**第 39 行
- **对每项进行编码：**第 40-44 行

对 byte，short，int 以及 long 型整数进行编码，并按照前面描述的顺序存入字节数组。

- **打印编码后数组的内容：**第 45 行
- **对编码字节数组中的某些字段进行解码：**第 47-51 行

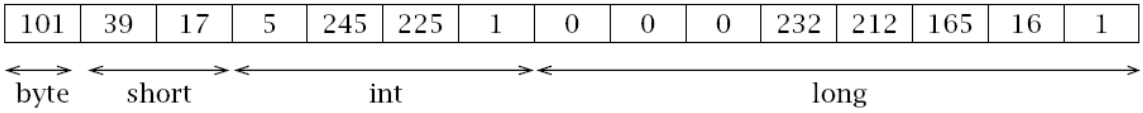
解码后输出的值应该与编码前的原始值相等。

- **转换问题：**第 53-59 行

在字节数组偏移量为 4 的位置，该字节的十进制值是 245，然而，当将其作为一个有符号字节读取时，其值则为-11（回忆有符号整数的二进制补码表示方法）。如果我们将返回值直接存入一个 long 型整数，它只是简单地变成这个 long 型整数的最后一个字节，值为 245。如果将返回值放入一个字节型整数，其值则为-11。到底哪个值正确取决于你的应用程序。如果你从 *N* 个字节解码后希望得到一个有符号的数值，就必须将解码结果（长的结果）存入一个刚好占用 *N* 个字节的基本整型中。如果你希望得到一个无符号的数组，就必须将解码结果存入更长的基本整型中，该整型至少要占用 *N*+1 个字节。

注意，在 encodeIntBigEndian() 和 decodeIntBigEndian()方法的开始部分，我们可能需要做一些前提条件检测，如 $0 \le size \le 8$ 和 $dst \ne null$ 等。你能举出需要做的其他前期检测吗？

运行以上程序，其输出显示了一下字节的值（以十进制形式）：



如你所见，上面的强制（brute-force）编码方法需要程序员做很多工作：要计算和命名每个数值的偏移量和大小，并要为编码过程提供合适的参数。如果没有将 encodeIntBigEndian()方法提出来作为一个独立的方法，情况会更糟。基于以上原因，强制编码方法是不推荐使用的，而且 Java 也提供了一些更加易用的内置机制。不过，值得注意的是强制编码方法也有它的优势，除了能够对标准的 Java 整型进行编码外，encodeIntegerBigEndian() 方法对 1 到 8 字节的任何整数都适用——例如，如果愿意的话，你可以对一个 7 字节的整数进行编码。

构建本例中的消息的一个相对简单的方法是使用 DataOutputStream 类和 ByteArrayOutputStream 类。DataOutputStream 类允许你将基本数据类型，如上述整型，写入一个流中：它提供了 writeByte()，writeShort()，writeInt()，以及 writeLong()方法，这些方法按照 big-endian 顺序，将整数以适当大小的二进制补码的形式写到流中。ByteArrayOutputStream 类获取写到流中的字节序列，并将其转换成一个字节数组。用这两个类来构建我们的消息的代码如下：

```
ByteArrayOutputStream buf = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(buf);
out.writeByte(byteVal);
out.writeShort(shortVal);
out.writeInt(intVal);
out.writeLong(longVal);
out.flush();
byte[] msg = buf.toByteArray();
```

也许你想运行这段代码，来证实它与 BruteForceEncoding.java 的输出结果一样。

讲了这么多发送方相关的内容，那么接收方将如何恢复传输的数据呢？正如你想的那样，Java 中也提供了与输出工具类相似的输入工具类，分别是 DataInputStream 类和 ByteArrayInputStream 类。在后面讨论如何解析传入的消息时，我们将对这两个类的使用举例。并且，在第 5 章中，我们还会看到另一种方法，使用 ByteBuffer 类将基本数据类型转换成字节序列。

最后，本节的所有内容基本上也适用于 BigInteger 类，该类支持任意大的整数。对于基本整型，发送者和接收者必须在使用多大空间（字节数）来表示一个数值上达成共识。但是这又与使用 BigInteger 相矛盾，因为 BigInteger 可以是任意大小。一种解决方法是使用基于长度的帧，

我们将在第 3.3 节看到这种方法。

3.1.2 字符串和文本

历史悠久的文本（可打印，即可显示的字符串）可能是用来表示信息最常用的方式。文本使用起来非常方便，因为人们习惯于处理各种各样以字符串形式表示的信息，如书本中，报纸中，以及电脑显示器上的信息。因此，只要我们指定如何对要传输的文本进行编码，我们就几乎能发送其他任何类型的数据：先将其表示成文本形式，再对文本进行编码。显然，我们可以将数字和 boolean 类型的数据表示成 String 类型，如“123478962”，“6.02e23”，“true”，“false”等。我们也已经看到，通过调用 getBytes()方法，可以将一个字符串转换成字节数组（见 TCPEchoClient.java）。当然，还有其他方法实现这个功能。

为了更好地理解这个过程，我们首先得将文本视为由符号和字符（*characters*）组成。实际上每个 String 实例都对应了一个字符序列（数组，char[]类型）。一个字符在 Java 内部表示为一个整数。例如，字符“a”，即字母“a”的符号，与整数 97 对应；字符“X”对应了 88，而符号“!”（感叹号）则对应了 33。

在一组符号与一组整数之间的映射称为编码字符集（*coded character set.*）。或许你听说过 *ASCII* 编码字符集（*ASCII*，*American Standard Code for Information Interchange*，美国标准信息交换码）。ASCII 码将英语字母、数字、标点符号以及一些特殊符号（不可打印字符）映射成 0 到 127 的整数。自 20 世纪 60 年代以来，ASCII 码就被用来进行数据传输，甚至在今天，它也广泛应用在应用程序协议中，如 HTTP（万维网所用的协议）。然而，由于它忽略了许多英语以外的其他语言所使用的符号，在如今全球化经济环境下，使用 ASCII 码来开发应用程序和设计协议就显得不够理想。

因此，Java使用了一种称为*Unicode*的国际标准编码字符集来表示char型和String型值。Unicode字符集将“世界上大部分的语言和符号”[1]映射到整数 0 至 65535 之间，能更好地适用于国际化程序。例如，日文平假名中代表音节“o”的符号映射成了整数 12362。Unicode包含了ASCII码：每个ASCII码中定义的符号在Unicode中所映射整数与其在ASCII码中映射的整数相同。这就为ASCII与Unicode之间提供了一定程度的向后兼容性。

发送者与接收者必须在符号与整数的映射方式上达成共识，才能使用文本信息进行通信。这就是他们所要达成一致的所有内容吗？还得根据情况而定。对于每个整数值都比 255 小的一小组字符，则不需要其他信息，因为其每个字符都能够作为一个单独的字节进行编码。对于可能使用超过一个字节的大整数的编码方式，就有多种方式在线路上对其进行编码。因此，发送者和接收者还需要对这些整数如何表示成字节序列统一意见，即编码方案（*encoding scheme*）。编码字符集和字符的编码方案结合起来称为字符集（*charset*，见 RFC 2278）。你也可以定义自己的字符集，但没有理由这样做，世界上已经有大量不同的标准（*standardized*）字符集在使用。Java 提供了对任意字符集的支持，而且每种实现都必须支持以下至少一种字符集：US-ASCII（ASCII 的另一个名字），ISO-8859-1，UTF-8，UTF-16BE，UTF-16LE，UTF-16。

调用 String 实例的 getBytes()方法，将返回一个字节数组，该数组根据平台默认字符集（*default charset*）对 String 实例进行了编码。很多平台的默认字符集都是 UTF-8，然而在一些经常使用 ASCII 字符集以外的字符的地区，情况有所不同。要保证一个字符串按照特定（*particular*）字符集编码，只需要将该字符集的名字作为参数（String 类型）传递给 getBytes()方法，其返回的字节数组就包含了由指定字符集表示的字符串。（注意，在第 2.2.1 节的 TCP 回显客户端/服务器示例程序与编码是无关的，因为它们根本没有对接收到的数据进行解释。）

[1] The Unicode Consortium, *The Unicode Standard, Version 5.0*, Addison Wesley, 2006.

下面举例来对 getBytes()方法进行说明。如果在著作本书的平台上调用"Test!".getBytes()，你将获得按照 UTF-8 字符集编码的字节数组；然而如果你调用"Test!".getBytes("UTF-16BE")，

84	101	115	116	33

你将得到如下数组：在这种情况下每个值被编码成了两个字节的序列，高位在前；

0	84	0	101	0	115	0	116	0	33

如果调用 "Test!".getBytes("IBM037")，返回结果将是：

227	133	162	163	90

上面的例子说明，*发送者和接收者必须在文本字符串的表示方式上达成共识*。最简单的方法就是定义一个标准字符集。

我们知道，可以通过先将字符串转换成独立的字节，再将其写到流中的方式，把 String 写入到 OutputStream 中去。这个方法在每次调用 getBytes()方法时，都得指定编码方式。在本章后续内容中，我们将看到只需要简单指定一次编码就能构建文本消息的方法。

3.1.3 位操作：布尔值编码

位图（*Bitmaps*）是对布尔信息进行编码的一种非常紧凑的方式，通常用在协议中。位图的主要思想是整型数据中的每一位都能够对一个布尔值编码——通常是 0 表示 false，1 表示 true。要操纵位图，你需要了解如何使用 Java 中的“位操作”方法来设置和清除单独的一位。掩码（*mask*）是一个的整数值，其中有一位或多位被设为 1，其他各位被清空（即，设为 0）。在这里我们处理的是 int 大小的位图和掩码（32 位），但这些方法对其他类型的整数也同样适用。

我们将int中的各位从 0 到 31 进行编号，其中 0 代表最低位。一般来说，如果一个int值在第 *i*位值为 1，其他位都为 0 的话，该int型整数的值就是 2^i。因此编号为 5 的位表示 32，编号为 12 的位表示 4096，等等。这里有一些掩码声明的例子：

```
final int BIT5 = (1<<5);
final int BIT7 = 0x80;
final int BITS2AND3 = 12; // 8+4
int bitmap = 1234567;
```

要*设置* int 变量中的特定一位，需要将该 int 值与特定位对应的掩码进行按位或（bitwise-OR）操作（|）：

```
bitmap |= BIT5;
// bit 5 is now one
```

要*清空*特定一位，则将该整数与特定所对应的掩码的按位补码（特定位为 0，其他位为 1）进行按位与（bitwise-AND）操作。Java 中的按位与操作符是&，而按位补码操作符是~：

```
bitmap &= ~BIT7;
// bit 7 is now zero
```

也可以通过将相应的所有掩码进行按位或操作，一次设置和清空多位：

```
// clear bits 2, 3 and 5
bitmap &= ~(BITS2AND3|BIT5);
```

要测试一个整数的特定位是否已经被设置，可以将该整数与特定位对应的掩码进行按位与，并将操作结果与 0 比较：

```
boolean bit6Set = (bitmap & (1<<6)) != 0;
```

3.2 组合输入输出流

Java 中与流相关的类可以组合起来从而提供强大的功能。例如，我们可以将一个 Socket 实例的 OutputStream 包装在一个 BufferedOutputStream 实例中，这样可以先将字节暂时缓存在一起，然后再一次全部发送到底层的通信信道中，以提高程序的性能。我们还能再将这个 BufferedOutputStream 实例包裹在一个 DataOutputStream 实例中，以实现发送基本数据类型的功能。以下是实现这种组合的代码：

```
Socket socket = new Socket(server, port);
DataOutputStream out = new DataOutputStream(
      new BufferedOutputStream(socket.getOutputStream()));
```

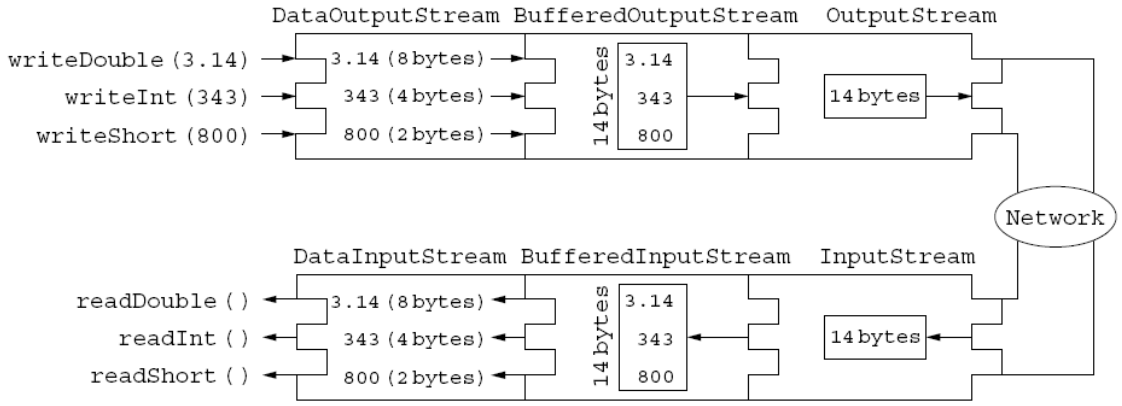


图 3.1：流组合

bytes：字节；Network：网络

输入输出	类型
Buffered[Input/Output]Stream	性能
Checked[Input/Output]Stream	维护
Cipher[Input/Output]Stream	加密/解密
Data[Input/Output]Stream	数据处理
Digest[Input/Output]Stream	维护
GZIP[Input/Output]Stream	压缩/解压缩
Object[Input/Output]Stream	数据处理

表 3.1：Java 输入输出类

图 3.1 展示了这中组合。在这个例子中，我们先将基本数据的值，一个一个写入 DataOutputStream 中，DataOutputStream 再将这些数据以二进制的形式写入 BufferedOutputStream 将三次写入的数据缓存起来，然后再由 BufferedOutputStream 一次性地将这些数据写入套接字的 OutputStream，最后由 OutputStream 将数据发送到网络。在另一个终端，我们创建了相应的组合 InputStream，以有效地接收基本数据类型。

对 Java 输入输出 API 的完整介绍不在本书的讨论范围中，不过，表 3.1 列出了一些相关的 Java 输入输出类，为介绍它们的强大功能起一个抛砖引玉的作用。

3.3 成帧与解析

当然，将数据转换成在线路上传输的格式只完成了一半工作，在接收端还必须将接收到的字节序列还原成原始信息。应用程序协议通常处理的是由一组字段组成的离散的信息。成帧（*Framing*）技术则解决了接收端如何定位消息的首尾位置的问题。无论信息是编码成了文本、多字节二进制数、或是两者的结合，应用程序协议必须指定消息的接收者如何确定何时消息已完整接收。

如果一条完整的消息负载在一个 DatagramPacket 中发送，这个问题就变得很简单了：DatagramPacket 负载的数据有一个确定的长度，接收者能够准确地知道消息的结束位置。然而，如果通过 TCP 套接字来发送消息，情况将变得更复杂，因为 TCP 协议中没有消息边界的概念。如果一个消息中的所有字段都有固定的长度，同时每个消息又是由固定数量的字段组成的话，消息的长度就能够确定，接收者就可以简单地将消息长度对应的字节数读到一个 byte[]缓存区中。在 TCPEchoClient.java 示例程序中我们就是用的这个方法，在该例中我们能从服务器获得消息的字节数。但是如果消息的长度是可变的（例如消息中包含了一些变长的文本字符串），我们事先就无法知道需要读取多少字节。

如果接收者试图从套接字中读取比消息本身更多的字节，将可能发生以下两种情况之一：如果信道中没有其他消息，接收者将阻塞等待，同时无法处理接收到的消息；如果发送者也在等待接收端的响应信息，则会形成*死锁*（*deadlock*）；另一方面，如果信道中还有其他消息，则接收者会将后面消息的一部分甚至全部读到第一条消息中去，这将产生一些协议错误。因此，在使用 TCP 套接字时，成帧就是一个非常重要的考虑因素。

一些相同的考虑也适用于查找消息中每个字段的边界：接收者需要知道每个字段的结束位置和下一个字段的开始位置。因此，我们在此介绍的消息成帧技术几乎都可以应用到字段上。然而，最简单并使代码最简洁的方法是将这两个问题分开处理：首先定位消息的结束位置，然后将消息作为一个整体进行解析。在此我们专注于将整个消息作为一帧进行处理。

主要有两个技术使接收者能够准确地找到消息的结束位置：

- 基于定界符（*Delimiter-based*）：消息的结束由一个唯一的标记（*unique marker*,）指出，即发送者在传输完数据后显式添加的一个特殊字节序列。这个特殊标记不能在传输的数据中出现。
- 显式长度（*Explicit length*）：在变长字段或消息前附加一个固定大小的字段，用来指示该字段或消息中包含了多少字节。

基于定界符的方法的一个特殊情况是，可以用在 TCP 连接上传输的最后一个消息上：在发送完这个消息后，发送者就简单地关闭（使用 shutdownOutput()或 close()方法）发送端的 TCP

连接。接收者读取完这条消息的最后一个字节后，将接收到一个流结束标记（即 read()方法返回 -1），该标记指示出已经读取到达了消息的末尾。

基于定界符的方法通常用在以文本方式编码的消息中：定义一个特殊的字符或字符串来标识消息的结束。接收者只需要简单地扫描输入信息（以字节的方式）来查找定界序列，并将定界符前面的字符串返回。这种方法的缺点是*消息本身不能包含有定界字符*，否则接收者将提前认为消息已经结束。在基于定界符的成帧方法中，发送者要保证满足这个先决条件。幸运的是，填充（*stuffing*）技术能够对消息中出现才定界符进行修改，从而是接收者不将其识别为定界符。在接收者扫描定界符时，还能识别出修改过的数据，并在输出消息中对其进行还原，从而使其与原始消息一致。这个技术的缺点是发送者和接收者双方都必须扫描消息。

基于长度的方法更简单一些，不过要使用这种方法必须知道消息长度的上限。发送者先要确定消息的长度，将长度信息存入一个整数，作为消息的前缀。消息的长度上限定义了用来编码消息长度所需要的字节数：如果消息的长度小于 256 字节，则需要 1 个字节；如果消息的长度小于 65536 字节，则需要 2 个字节，等等。

为了展示以上技术，我们将介绍下面定义的 Framer 接口。它有两个方法：frameMsg()方法用来添加成帧信息并将指定消息输出到指定流，nextMsg()方法则扫描指定的流，从中抽取出下一条消息。

Framer.java

```
0 import java.io.IOException;
1 import java.io.OutputStream;
2
3 public interface Framer {
4 void frameMsg(byte[] message, OutputStream out) throws IOException;
5 byte[] nextMsg() throws IOException;
6 }
```

Framer.java

DelimFramer.java 类实现了基于定界符的成帧方法，其定界符为“换行”符（"\n", 字节值为 10）。 frameMethod()方法并没有实现*填充*，当成帧的字节序列中包含有定界符时，它只是简单地抛出异常。（扩展该方法以实现填充功能将作为练习留给读者）nextMsg()方法扫描流，直到读取到了定界符，并返回定界符前面的所有字符，如果流为空则返回 null。如果累积了一个消息的不少字符，但直到流结束也没有找到定界符，程序将抛出一个异常来指示成帧错误。

DelimFramer.java

```
0 import java.io.ByteArrayOutputStream;
1 import java.io.EOFException;
2 import java.io.IOException;
3 import java.io.InputStream;
4 import java.io.OutputStream;
5
6 public class DelimFramer implements Framer {
7
8 private InputStream in; // data source
9 private static final byte DELIMITER = "\n"; // message delimiter
10
11 public DelimFramer(InputStream in) {
12 this.in = in;
13 }
14
```

```
15 public void frameMsg(byte[] message, OutputStream out) throws IOException {
16 // ensure that the message does not contain the delimiter
17 for (byte b : message) {
18 if (b == DELIMITER) {
19 throw new IOException("Message contains delimiter");
20 }
21 }
22 out.write(message);
23 out.write(DELIMITER);
24 out.flush();
25 }
26
27 public byte[] nextMsg() throws IOException {
28 ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
29 int nextByte;
30
31 // fetch bytes until find delimiter
32 while ((nextByte = in.read()) != DELIMITER) {
33 if (nextByte == -1) { // end of stream?
34 if (messageBuffer.size() == 0) { // if no byte read
35 return null;
36 } else { // if bytes followed by end of stream: framing error
37 throw new EOFException("Non-empty message without delimiter");
38 }
39 }
40 messageBuffer.write(nextByte); // write byte to buffer
41 }
42
43 return messageBuffer.toByteArray();
44 }
45 }
```

DelimFramer.java

1.**构造函数：**第 11-13 行

获取消息的输入流作为参数传递给该函数。

2.**frameMsg() 方法用于添加帧信息：**第 15-25 行

- **校验消息形式的有效性：**第 17-21 行

检查消息中是否包含了定界符，如果包含，则抛出一个异常。

- **写消息：**第 22 行

将成帧的消息输出到流中。

- **写定界符：**第 23 行

3. nextMsg()**方法从输入中提取消息：**第 27-44 行

- **读取流中的每个字节，直到遇到定界符为止：**第 32 行
- **处理流的终点：**第 33-39 行

如果在遇到定界符之前就已经到了流的终点，则分两种情况：一是从帧的构造开始或从遇到前一个定界符以来，缓存区已经接收了一些字节，这时程序将抛出一个异常；否则 nextMsg()方法将返回 null 以表示全部消息已接收完。

- **将无定界符的字节写入消息缓存区：**第 40 行
- **将消息缓存区中的内容以字节数组的形式返回：**第 43 行

我们的定界符帧有一个限制，即它不支持多字节定界符。如何对其进行修改以支持多字节定界符将作为练习留给我们的读者。

LengthFramer.java类实现了基于长度的成帧方法，适用于长度小于 65535 ($2^{16} - 1$)字节的消息。发送者首先给出指定消息的长度，并将长度信息以big-endian顺序存入两个字节的整数中，再将这两个字节放在完整的消息内容前，连同消息一起写入输出流。在接收端，我们使用DataInputStream以读取整型的长度信息；readFully() 方法将阻塞等待，直到给定的数组完全填满，这正是我们需要的。值得注意的是，使用这种成帧方法，发送者不需要检查要成帧的消息内容，而只需要检查消息的长度是否超出了限制。

LengthFramer.java

```
0 import java.io.DataInputStream;
1 import java.io.EOFException;
2 import java.io.IOException;
3 import java.io.InputStream;
4 import java.io.OutputStream;
5
6 public class LengthFramer implements Framer {
7 public static final int MAXMESSAGELENGTH = 65535;
8 public static final int BYTEMASK = 0xff;
9 public static final int SHORTMASK = 0xffff;
10 public static final int BYTESHIFT = 8;
11
12 private DataInputStream in; // wrapper for data I/O
13
14 public LengthFramer(InputStream in) throws IOException {
15 this.in = new DataInputStream(in);
16 }
17
18 public void frameMsg(byte[] message, OutputStream out) throws IOException {
19 if (message.length > MAXMESSAGELENGTH) {
20 throw new IOException("message too long");
21 }
22 // write length prefix
23 out.write((message.length >> BYTESHIFT) & BYTEMASK);
24 out.write(message.length & BYTEMASK);
25 // write message
26 out.write(message);
27 out.flush();
28 }
29
30 public byte[] nextMsg() throws IOException {
31 int length;
32 try {
33 length = in.readUnsignedShort(); // read 2 bytes
34 } catch (EOFException e) { // no (or 1 byte) message
35 return null;
36 }
37 // 0 <= length <= 65535
38 byte[] msg = new byte[length];
39 in.readFully(msg); // if exception, it's a framing error.
40 return msg;
```

```
41 }
42 }
```

LengthFramer.java

1. **构造函数：**第 14-16 行

获取帧消息源的输入流，并将其包裹在一个 DataInputStream 中。

2. **frameMsg()方法用来添加成帧信息：**第 18-28 行

- **校验消息长度：**第 19-21 行

由于我们用的是长为两个字节的字段，因此消息的长度不能超过 65535。（注意该值太大而不能存入一个 short 型整数中，因此我们每次只向输出流写一个字节）。

- **输出长度字段：**第 23-24 行

添加长度信息（无符号 short 型整数）前缀，输出消息的字节数。

- **输出消息：**第 26 行

3.nextMsg()**方法用于从输入流中提取下一帧：**第 30-41 行

- **读取长度前缀：**第 32-36 行

readUnsignedShort()方法读取两个字节，将它们作为 big-endian 整数进行解释，并以 int 型整数返回它们的值。

- **读取指定数量的字节：**第 38-39 行

readfully() 将阻塞等待，直到接收到足够的字节来填满指定的数组。

- **以字节的形式返回消息：**第 40 行

3.4 Java 特定编码

当你使用套接字时，通常要么是你需要同时创建通信信道两端的程序（这种情况下你也拥有了协议的完全控制权），要么实现一个*给定的*协议进行通信。如果你知道（i）通信双方都使用 Java 实现，而且（ii）你拥有对协议的完全控制权，那么就可以使用 Java 的内置工具如 Serializable 接口或*远程方法调用*（*Remote Method Invocation*，*RMI*）工具。RMI 能够调用不同 Java 虚拟机上的方法，并隐藏了所有繁琐的参数编码解码细节。序列化（Serialization）处理了将实际的 Java 对象转换成字节序列的工作，因此你可以在不同虚拟机之间传递 Java 对象实例。

这些能力可能就好像沟通的涅槃，但是在实际中，由于种种原因，它们并不总是最好的解决方案。首先，由于它们都是很笼统的工具，因而在通信开销上不能做到最高效。例如，一个对象的序列化形式，其包含的信息在 Java 虚拟机（JVM）环境以外是毫无意义的。其次，Serializable 和 Externalizable 接口不能用于已经定义了不同传输格式的情况——如一个标准的协议。最后，用户自定义的类必须自己实现它们的序列化接口，*而这项工作很容易出错*。再强调一次，在某些情况下，这些 Java 的内置工具的确很有用，但是有些时候，“实现你自己的方法”可能更简单、容易或更有效。

3.5 构建和解析协议消息

本章结束时我们再看一个简单的例子，对在实现别人定义的协议时可能用到的技术进行了介绍。这个例子程序是一个简单的“投票”协议，如图 3.2 所示。这里，一个客户端向服务器发送了一个请求消息，消息中包含了一个候选人 ID，范围是 0 至 1000。

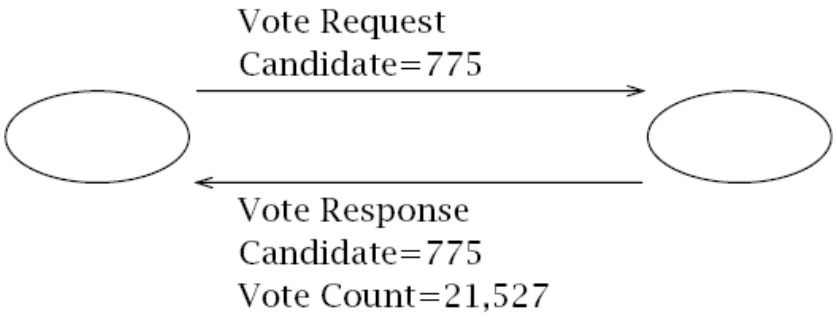


图 3.2：投票协议

Vote Reques：投票请求， Candidate：候选人，Vote Count：选票总数

程序支持两种请求。一种是*查询*（*inquiry*），即向服务器询问给定候选人当前获得的投票总数。服务器发回一个响应消息，包含了原来的候选人 ID 和该候选人当前（查询请求收到时）获得的选票总数。另一种是*投票*（*voting*）请求，即向指定候选人投一票。服务器对这种请求也发回响应消息，包含了候选人 ID 和其获得的选票数（包括了刚投的一票）。

在实现一个协议时，定义一个专门的类来存放消息中所包含的信息是大有裨益的。该类提供了操作消息中的字段的方法——同时用来维护不同字段之间的不变量。在我们的例子中，客户端和服务器端发送的消息都非常简单，它们唯一的区别是服务器端发送的消息包含了选票总数和一个表示响应消息（不是请求消息）的标志。因此，我们可以用一个类来表示客户端和服务器端的两种消息。VoteMsg.java 类展示了每条消息中的基本信息：

- 布尔值 *isInquiry*，其值为 true 时表示该消息是查询请求（为 false 时表示该消息是投票信息）；
- 布尔值 *isResponse*，指示该消息是响应（由服务器发送）还是请求；
- 整型变量 *candidateID* 指示了候选人的 ID；
- 长整型变量 *voteCount* 指示出所查询的候选人获得的总选票数。

这个类还维护了以下字段间的不变量：

- *candidateID* 的范围是 0 到 1000。
- *voteCount* 在响应消息中只能是一个非零值（*isResponse* 为 true）。
- *voteCount* 不能为负数。

VoteMsg.java

```
0 public class VoteMsg {
1 private boolean isInquiry; // true if inquiry; false if vote
2 private boolean isResponse;// true if response from server
3 private int candidateID; // in [0,1000]
4 private long voteCount; // nonzero only in response
5
6 public static final int MAX_CANDIDATE_ID = 1000;
7
8 public VoteMsg(boolean isResponse, boolean isInquiry, int candidateID, long
voteCount)
9 throws IllegalArgumentException {
10 // check invariants
11 if (voteCount != 0 && !isResponse) {
12 throw new IllegalArgumentException("Request vote count must be zero");
13 }
14 if (candidateID < 0 || candidateID > MAX_CANDIDATE_ID) {
15 throw new IllegalArgumentException("Bad Candidate ID: " + candidateID);
16 }
17 if (voteCount < 0) {
18 throw new IllegalArgumentException("Total must be >= zero");
19 }
20 this.candidateID = candidateID;
21 this.isResponse = isResponse;
22 this.isInquiry = isInquiry;
23 this.voteCount = voteCount;
24 }
25
26 public void setInquiry(boolean isInquiry) {
27 this.isInquiry = isInquiry;
28 }
29
30 public void setResponse(boolean isResponse) {
31 this.isResponse = isResponse;
32 }
33
34 public boolean isInquiry() {
35 return isInquiry;
36 }
37
38 public boolean isResponse() {
39 return isResponse;
40 }
41
42 public void setCandidateID(int candidateID) throws IllegalArgumentException {
43 if (candidateID < 0 || candidateID > MAX_CANDIDATE_ID) {
44 throw new IllegalArgumentException("Bad Candidate ID: " + candidateID);
45 }
46 this.candidateID = candidateID;
47 }
48
49 public int getCandidateID() {
50 return candidateID;
51 }
52
53 public void setVoteCount(long count) {
54 if ((count != 0 && !isResponse) || count < 0) {
55 throw new IllegalArgumentException("Bad vote count");
56 }
57 voteCount = count;
58 }
59
60 public long getVoteCount() {
61 return voteCount;
```

```
62 }
63
64 public String toString() {
65 String res = (isInquiry ? "inquiry" : "vote") + " for candidate " + candidateID;
66 if (isResponse) {
67 res = "response to " + res + " who now has " + voteCount + " vote(s)";
68 }
69 return res;
70 }
71 }
```

VoteMsg.java

现在我们有了一个用 Java 表示的投票消息，还需要根据一定的协议来对其进行编码和解码。VoteMsgCoder 接口提供了对投票消息进行序列化和反序列化的方法。

VoteMsgCoder.java

```
0 import java.io.IOException;
1
2 public interface VoteMsgCoder {
3 byte[] toWire(VoteMsg msg) throws IOException;
4 VoteMsg fromWire(byte[] input) throws IOException;
5 }
```

VoteMsgCoder.java

toWire()方法用于根据一个特定的协议，将投票消息转换成一个字节序列，fromWire()方法则根据相同的协议，对给定的字节序列进行解析，并根据信息的内容构造出消息类的一个实例。

为了介绍不同的信息编码方法，我们展示了两个实现 VoteMsgCoder 接口的类。一个使用的是基于文本的编码方式，另一个使用的是二进制的编码方式。如果你能确定一直不变地使用一种编码方式，那么 toWire()方法和 fromWire()方法就可以定义为 VoteMsg 类的一部分。这里我们这样做是为了强调抽象表示与编码的细节是相互独立的。

3.5.1 基于文本的表示方法

首先，我们介绍一个用文本方式对消息进行编码的版本。该协议指定使用 US-ASCII 字符集对文本进行编码。消息的开头是一个所谓的“魔术字符串”，即一个字符序列，用于接收者快速将投票协议的消息和网络中随机到来的垃圾消息区分开。投票/查询布尔值被编码成字符形式，‘v’表示投票消息，‘i’表示查询消息。消息的状态，即是否为服务器的响应，由字符‘R’指示。状态标记后面是候选人 ID，其后跟的是选票总数，它们都编码成十进制字符串。VoteMsgTextCoder 类提供了一种基于文本的 VoteMsg 编码方法。

VoteMsgTextCoder.java

```
0 import java.io.ByteArrayInputStream;
1 import java.io.IOException;
2 import java.io.InputStreamReader;
3 import java.util.Scanner;
4
5 public class VoteMsgTextCoder implements VoteMsgCoder {
6 /*
7 * Wire Format "VOTEPROTO" <"v"|"i"> [<RESPFLAG>] <CANDIDATE> [<VOTECNT>]
8 * Charset is fixed by the wire format.
9 */
```

```
10
11 // Manifest constants for encoding
12 public static final String MAGIC = "Voting";
13 public static final String VOTESTR = "v";
14 public static final String INQSTR = "i";
15 public static final String RESPONSESTR = "R";
16
17 public static final String CHARSETNAME = "US-ASCII";
18 public static final String DELIMSTR = " ";
19 public static final int MAX_WIRE_LENGTH = 2000;
20
21 public byte[] toWire(VoteMsg msg) throws IOException {
22 String msgString = MAGIC + DELIMSTR + (msg.isInquiry() ? INQSTR : VOTESTR)
23 + DELIMSTR + (msg.isResponse() ? RESPONSESTR + DELIMSTR : "")
24 + Integer.toString(msg.getCandidateID()) + DELIMSTR
25 + Long.toString(msg.getVoteCount());
26 byte data[] = msgString.getBytes(CHARSETNAME);
27 return data;
28 }
29
30 public VoteMsg fromWire(byte[] message) throws IOException {
31 ByteArrayInputStream msgStream = new ByteArrayInputStream(message);
32 Scanner s = new Scanner(new InputStreamReader(msgStream, CHARSETNAME));
33 boolean isInquiry;
34 boolean isResponse;
35 int candidateID;
36 long voteCount;
37 String token;
38
39 try {
40 token = s.next();
41 if (!token.equals(MAGIC)) {
42 throw new IOException("Bad magic string: " + token);
43 }
44 token = s.next();
45 if (token.equals(VOTESTR)) {
46 isInquiry = false;
47 } else if (!token.equals(INQSTR)) {
48 throw new IOException("Bad vote/inq indicator: " + token);
49 } else {
50 isInquiry = true;
51 }
52
53 token = s.next();
54 if (token.equals(RESPONSESTR)) {
55 isResponse = true;
56 token = s.next();
57 } else {
58 isResponse = false;
59 }
60 // Current token is candidateID
61 // Note: isResponse now valid
62 candidateID = Integer.parseInt(token);
63 if (isResponse) {
64 token = s.next();
65 voteCount = Long.parseLong(token);
66 } else {
67 voteCount = 0;
68 }
69 } catch (IOException ioe) {
70 throw new IOException("Parse error...");
71 }
72 return new VoteMsg(isResponse, isInquiry, candidateID, voteCount);
```

```
73 }
74 }
```

VoteMsgTextCoder.java

toWire()方法简单地创建一个字符串，该字符串中包含了消息的所有字段，并由空白符隔开。fromWire()方法首先检查“魔术”字符串，如果在消息最前面没有魔术字符串，则抛出一个异常。这里说明了在实现协议时非常重要的一点：*永远不要对从网络来的任何输入进行任何假设*。你的程序必须时刻为任何可能的输入做好准备，并能够很好地对其进行处理。在这个例子中，如果接收到的不是期望的消息，fromWire()方法将抛出一个异常，否则，就使用 Scanner 实例，根据空白符一个一个地获取字段。注意，消息的字段数与其是请求消息（由客户端发送）还是响应消息（由服务器发送）有关。如果输入流提前结束或格式错误，fromWire()方法将抛出一个异常。

3.5.2 二进制表示方法

下面我们将展示另一种对投票协议消息进行编码的方法。与基于文本的格式相反，二进制格式使用固定大小的消息。每条消息由一个特殊字节开始，该字节的最高六位为一个“魔术”值010101。这一点少量的冗余信息为接收者收到适当的投票消息提供了一定程度的保证。该字节的最低两位对两个布尔值进行了编码。消息的第二个字节总是 0，第三、第四个字节包含了candidateID 值。只有响应消息的最后 8 个字节才包含了选票总数信息。

VoteMsgBinCoder.jav

```
0 import java.io.ByteArrayInputStream;
1 import java.io.ByteArrayOutputStream;
2 import java.io.DataInputStream;
3 import java.io.DataOutputStream;
4 import java.io.IOException;
5
6 /* Wire Format
7 * 1 1 1 1 1 1
8 * 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
9 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
10 * | Magic |Flags| ZERO |
11 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
12 * | Candidate ID |
13 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
14 * | |
15 * | Vote Count (only in response) |
16 * | |
17 * | |
18 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
19 */
20 public class VoteMsgBinCoder implements VoteMsgCoder {
21
22 // manifest constants for encoding
23 public static final int MIN_WIRE_LENGTH = 4;
24 public static final int MAX_WIRE_LENGTH = 16;
25 public static final int MAGIC = 0x5400;
26 public static final int MAGIC_MASK = 0xfc00;
27 public static final int MAGIC_SHIFT = 8;
28 public static final int RESPONSE_FLAG = 0x0200;
29 public static final int INQUIRE_FLAG = 0x0100;
30
31 public byte[] toWire(VoteMsg msg) throws IOException {
32 ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
```

```
33 DataOutputStream out = new DataOutputStream(byteStream); // converts ints
34
35 short magicAndFlags = MAGIC;
36 if (msg.isInquiry()) {
37 magicAndFlags |= INQUIRE_FLAG;
38 }
39 if (msg.isResponse()) {
40 magicAndFlags |= RESPONSE_FLAG;
41 }
42 out.writeShort(magicAndFlags);
43 // We know the candidate ID will fit in a short: it's > 0 && < 1000
44 out.writeShort((short) msg.getCandidateID());
45 if (msg.isResponse()) {
46 out.writeLong(msg.getVoteCount());
47 }
48 out.flush();
49 byte[] data = byteStream.toByteArray();
50 return data;
51 }
52
53 public VoteMsg fromWire(byte[] input) throws IOException {
54 // sanity checks
55 if (input.length < MIN_WIRE_LENGTH) {
56 throw new IOException("Runt message");
57 }
58 ByteArrayInputStream bs = new ByteArrayInputStream(input);
59 DataInputStream in = new DataInputStream(bs);
60 int magic = in.readShort();
61 if ((magic & MAGIC_MASK) != MAGIC) {
62 throw new IOException("Bad Magic #: " +
63 ((magic & MAGIC_MASK) >> MAGIC_SHIFT));
64 }
65 boolean resp = ((magic & RESPONSE_FLAG) != 0);
66 boolean inq = ((magic & INQUIRE_FLAG) != 0);
67 int candidateID = in.readShort();
68 if (candidateID < 0 || candidateID > 1000) {
69 throw new IOException("Bad candidate ID: " + candidateID);
70 }
71 long count = 0;
72 if (resp) {
73 count = in.readLong();
74 if (count < 0) {
75 throw new IOException("Bad vote count: " + count);
76 }
77 }
78 // Ignore any extra bytes
79 return new VoteMsg(resp, inq, candidateID, count);
80 }
81 }
```

VoteMsgBinCoder.jav

就像在第 3.1.1 节中一样，我们创建了一个 ByteArrayOutputStream 并将其包裹在一个 DataOutputStream 中来接收结果。这个编码方法利用了在合法 candidateID 中，其最高两个字节始终为 0 的特点。还要注意的是，该方法通过使用按位或操作，使用 1 位对每个布尔值进行编码。

3.5.3 发送和接收

通过流发送消息非常简单，只需要创建消息，调用 toWire()方法，添加适当的成帧信息，

再写入流。当然，接收消息就要按照相反的顺序执行。这个过程适用于 TCP 协议，而对于 UDP 协议，不需要显式地成帧，因为 UDP 协议中保留了消息的边界信息。为了对发送与接收过程进行展示，我们考虑投票服务的如下几点：1）维护一个候选人 ID 与其获得选票数的映射，2）记录提交的投票，3）根据其获得的选票数，对查询指定的候选人和为其投票的消息做出响应。首先，我们实现一个投票服务器所用到的服务。当接收到投票消息时，投票服务器将调用 VoteService 类的 handleRequest() 方法对请求进行处理。

VoteService.java

```
0 import java.util.HashMap;
1 import java.util.Map;
2
3 public class VoteService {
4
5 // Map of candidates to number of votes
6 private Map<Integer, Long> results = new HashMap<Integer, Long>();
7
8 public VoteMsg handleRequest(VoteMsg msg) {
9 if (msg.isResponse()) { // If response, just send it back
10 return msg;
11 }
12 msg.setResponse(true); // Make message a response
13 // Get candidate ID and vote count
14 int candidate = msg.getCandidateID();
15 Long count = results.get(candidate);
16 if (count == null) {
17 count = 0L; // Candidate does not exist
18 }
19 if (!msg.isInquiry()) {
20 results.put(candidate, ++count); // If vote, increment count
21 }
22 msg.setVoteCount(count);
23 return msg;
24 }
25 }
```

VoteService.java

1.**创建候选人 ID 与选票数量的映射：**第 6 行

对于查询请求，给定的候选人 ID 用来在映射中查询其获得的选票数量。对于投票请求，增加后的选票数又存回映射。

2.handleRequest()：第 8-24 行

- **返回响应：**第 9-12 行

如果投票消息已经是一个响应信息，则直接发回而不对其进行处理和修改。否则，对其响应消息标志进行设置。

- **查找当前获得的选票总数：**第 13-18 行

根据候选人 ID 从映射中获取其获得的选票总数。如果该候选人 ID 在映射中不存在，则将其获得的选票数设为 0.

- **如果有新的投票，则更新选票总数：**第 19-21 行

如果之前候选人不存在，则创建新的映射，否则，只是简单地修改已有的映射。

- **设置选票总数并返回消息：**第 22-23 行

下面我们将展示如何实现一个 TCP 投票客户端，该客户端通过 TCP 套接字连接到投票服务器，在一次投票后发送一个查询请求，并接收查询和投票结果。

VoteClientTCP.java

```
0 import java.io.OutputStream;
1 import java.net.Socket;
2
3 public class VoteClientTCP {
4
5 public static final int CANDIDATEID = 888;
6
7 public static void main(String args[]) throws Exception {
8
9 if (args.length != 2) { // Test for correct # of args
10 throw new IllegalArgumentException("Parameter(s): <Server> <Port>");
11 }
12
13 String destAddr = args[0]; // Destination address
14 int destPort = Integer.parseInt(args[1]); // Destination port
15
16 Socket sock = new Socket(destAddr, destPort);
17 OutputStream out = sock.getOutputStream();
18
19 // Change Bin to Text for a different framing strategy
20 VoteMsgCoder coder = new VoteMsgBinCoder();
21 // Change Length to Delim for a different encoding strategy
22 Framer framer = new LengthFramer(sock.getInputStream());
23
24 // Create an inquiry request (2nd arg = true)
25 VoteMsg msg = new VoteMsg(false, true, CANDIDATEID, 0);
26 byte[] encodedMsg = coder.toWire(msg);
27
28 // Send request
29 System.out.println("Sending Inquiry (" + encodedMsg.length + " bytes): ");
30 System.out.println(msg);
31 framer.frameMsg(encodedMsg, out);
32
33 // Now send a vote
34 msg.setInquiry(false);
35 encodedMsg = coder.toWire(msg);
36 System.out.println("Sending Vote (" + encodedMsg.length + " bytes): ");
37 framer.frameMsg(encodedMsg, out);
38
39 // Receive inquiry response
40 encodedMsg = framer.nextMsg();
41 msg = coder.fromWire(encodedMsg);
42 System.out.println("Received Response (" + encodedMsg.length
43 + " bytes): ");
44 System.out.println(msg);
45
46 // Receive vote response
47 msg = coder.fromWire(framer.nextMsg());
48 System.out.println("Received Response (" + encodedMsg.length
49 + " bytes): ");
50 System.out.println(msg);
51
52 sock.close();
```

```
53 }
54 }
```

VoteClientTCP.java

1.**参数处理：**第 9-14 行

2.**创建套接字，获取输出流：**第 16-17 行

3.**创建二进制编码器和基于长度的成帧器：**第 20-22 行

我们将使用一个编码器对投票消息进行编码和解码，这里为我们的协议选择的是二进制编码器。其次，由于 TCP 协议是一个基于流的服务，我们需要提供字节的帧。在此，我们使用 LengthFramer 类，它为每条消息添加一个长度前缀。注意，我们只需要改变具体的类，就能方便地转换成基于定界符的成帧方法和基于文本的编码方式，这里将 VoteMsgCoder 和 Framer 换成 VoteMsgTextCoder 和 DelimFramer 即可。

4.**创建和发送消息：**第 24-37 行

创建，编码，成帧和发送查询请求，后面是为相同候选人的投票消息。

5.**获取和解析响应：**第 39-50 行

我们使用 nextMsg()方法用于返回下一条编码后的消息，并通过 fromWire()方法对其进行解析/解码。

6.**关闭套接字：**第 52 行

下面我们示范 TCP 版本的投票服务器。该服务器反复地接收新的客户端连接，并使用 VoteService 类为客户端的投票消息作出响应。

VoteServerTCP.java

```
0 import java.io.IOException;
1 import java.net.ServerSocket;
2 import java.net.Socket;
3
4 public class VoteServerTCP {
5
6 public static void main(String args[]) throws Exception {
7
8 if (args.length != 1) { // Test for correct # of args
9 throw new IllegalArgumentException("Parameter(s): <Port>");
10 }
11
12 int port = Integer.parseInt(args[0]); // Receiving Port
13
14 ServerSocket servSock = new ServerSocket(port);
15 // Change Bin to Text on both client and server for different encoding
16 VoteMsgCoder coder = new VoteMsgBinCoder();
17 VoteService service = new VoteService();
18
19 while (true) {
20 Socket clntSock = servSock.accept();
21 System.out.println("Handling client at " + clntSock.getRemoteSocketAddress());
22 // Change Length to Delim for a different framing strategy
23 Framer framer = new LengthFramer(clntSock.getInputStream());
24 try {
```

```
25 byte[] req;
26 while ((req = framer.nextMsg()) != null) {
27 System.out.println("Received message (" + req.length + " bytes)");
28 VoteMsg responseMsg = service.handleRequest(coder.fromWire(req));
29 framer.frameMsg(coder.toWire(responseMsg), clntSock.getOutputStream());
30 }
31 } catch (IOException ioe) {
32 System.err.println("Error handling client: " + ioe.getMessage());
33 } finally {
34 System.out.println("Closing connection");
35 clntSock.close();
36 }
37 }
38 }
39 }
```

VoteServerTCP.java

1.**为服务器端建立编码器和投票服务：**第 15-17 行

2.**反复地接收和处理客户端连接：**第 19-37 行

- **接收新的客户端，打印客户端地址：**第 20-21 行
- **为客户端创建成帧器：**第 23 行
- **从客户端获取消息并对其解码：**第 26-28 行

  反复地向成帧器发送获取下一条消息的请求，直到其返回 null，即指示了消息的结束。

- **处理消息，发送响应信息：**第 28-29 行

  将解码后的消息传递给投票服务，以进行下一步处理。编码，成帧和回发响应消息。

UDP 版本的投票客户端与 TCP 版本非常相似。需要注意的是，在 UDP 客户端中我们不需要使用成帧器，因为 UDP 协议为我们维护了消息的边界信息。对于 UDP 协议，我们使用基于文本的编码方式对消息进行编码，不过只要客户端与服务器能达成一致，也能够很方便地改成其他编码方式。

VoteClientUDP.java

```
0 import java.io.IOException;
1 import java.net.DatagramPacket;
2 import java.net.DatagramSocket;
3 import java.net.InetAddress;
4 import java.util.Arrays;
5
6 public class VoteClientUDP {
7
8 public static void main(String args[]) throws IOException {
9
10 if (args.length != 3) { // Test for correct # of args
11 throw new IllegalArgumentException("Parameter(s): <Destination>" +
12 " <Port> <Candidate#>");
13 }
14
15 InetAddress destAddr = InetAddress.getByName(args[0]); // Destination addr
16 int destPort = Integer.parseInt(args[1]); // Destination port
17 int candidate = Integer.parseInt(args[2]); // 0 <= candidate <= 1000 req'd
```

```
18
19 DatagramSocket sock = new DatagramSocket(); // UDP socket for sending
20 sock.connect(destAddr, destPort);
21
22 // Create a voting message (2nd param false = vote)
23 VoteMsg vote = new VoteMsg(false, false, candidate, 0);
24
25 // Change Text to Bin here for a different coding strategy
26 VoteMsgCoder coder = new VoteMsgTextCoder();
27
28 // Send request
29 byte[] encodedVote = coder.toWire(vote);
30 System.out.println("Sending Text-Encoded Request (" + encodedVote.length
31 + " bytes): ");
32 System.out.println(vote);
33 DatagramPacket message = new DatagramPacket(encodedVote, encodedVote.length);
34 sock.send(message);
35
36 // Receive response
37 message = new DatagramPacket(new byte[VoteMsgTextCoder.MAX_WIRE_LENGTH],
38 VoteMsgTextCoder.MAX_WIRE_LENGTH);
39 sock.receive(message);
40 encodedVote = Arrays.copyOfRange(message.getData(), 0, message.getLength());
41
42 System.out.println("Received Text-Encoded Response (" + encodedVote.length
43 + " bytes): ");
44 vote = coder.fromWire(encodedVote);
45 System.out.println(vote);
46 }
47 }
```

VoteClientUDP.java

1.**设置** DatagramSocket **和连接：**第 10-20 行

通过调用 connect()方法，我们不必 1）为发送的每个数据报文指定远程地址和端口，也不必 2）测试接收到的每个数据报文的源地址。

2.**创建选票和编码器：**第 22-26 行

这次使用的是文本编码器，但我们也可以很容易地换成二进制编码器。注意这里我们不需要成帧器，因为只要每次发送都只有一个投票消息，UDP 协议就已经为我们保留了边界信息。

3.**向服务器发送请求消息：**第 28-34 行

4.**接收，解码和打印服务器响应信息：**第 36-45 行

在创建 DatagramPacket 时，我们需要知道消息的最大长度，以避免数据被截断。当然，在对数据报文进行解码时，我们只使用数据报文中包含的实际字节，因此调用了 Arrays.copyOfRange()方法来复制返回的数据报文中数组的子序列。

最后是 UDP 投票服务器，同样，也与 TCP 版本非常相似。

VoteServerUDP.java

```
0 import java.io.IOException;
1 import java.net.DatagramPacket;
2 import java.net.DatagramSocket;
```

```
3 import java.util.Arrays;
4
5 public class VoteServerUDP {
6
7 public static void main(String[] args) throws IOException {
8
9 if (args.length != 1) { // Test for correct # of args
10 throw new IllegalArgumentException("Parameter(s): <Port>");
11 }
12
13 int port = Integer.parseInt(args[0]); // Receiving Port
14
15 DatagramSocket sock = new DatagramSocket(port); // Receive socket
16
17 byte[] inBuffer = new byte[VoteMsgTextCoder.MAX_WIRE_LENGTH];
18 // Change Bin to Text for a different coding approach
19 VoteMsgCoder coder = new VoteMsgTextCoder();
20 VoteService service = new VoteService();
21
22 while (true) {
23 DatagramPacket packet = new DatagramPacket(inBuffer, inBuffer.length);
24 sock.receive(packet);
25 byte[] encodedMsg = Arrays.copyOfRange(packet.getData(), 0, packet.getLength());
26 System.out.println("Handling request from " + packet.getSocketAddress() + " ("
27 + encodedMsg.length + " bytes)");
28
29 try {
30 VoteMsg msg = coder.fromWire(encodedMsg);
31 msg = service.handleRequest(msg);
32 packet.setData(coder.toWire(msg));
33 System.out.println("Sending response (" + packet.getLength() + " bytes):");
34 System.out.println(msg);
35 sock.send(packet);
36 } catch (IOException ioe) {
37 System.err.println("Parse error in message: " + ioe.getMessage());
38 }
39 }
40 }
41 }
```

VoteServerUDP.java

1.**设置：**第 17-20 行

为服务器创建接收缓存区，编码器，以及投票服务。

2.**反复地接收和处理客户端的投票消息：**第 22-39 行

- **为接收数据报文创建** DatagramPacket：第 23 行

在每次迭代中将数据区重置为输入缓存区。

- **接收数据报文，抽取数据：**第 24-25 行

UDP 替我们完成了成帧的工作！

- **解码和处理请求：**第 30-31 行

服务将响应返回给消息。

- **编码并发送响应消息：**第 32-35 行

3.6 结束

我们已经看到如何将基本数据类型表示成字节序列“在信道”上传输。我们还考虑了一些微妙的文本字符串编码方法，以及一些成帧和消息解析的基本方法。我们还见到了基于文本编码和二进制编码的协议的例子。

这里可能值得重申一下我们在前言中所说的：本章并不会使你成为专家！那需要大量的经验。但是本章中的代码能够作为你进一步探索的起点。

3.7 练习

1.在Java中，大于 $2^{31} - 1$ (小于 $2^{32} - 1$) 的数不能表示为int型，但是它们可以表示为 32 位的二进制数。试写一个方法将这类整数写入流中。这需要使用一个long型变量和一个OuputStream类实例作为参数。

2.扩展 DelimFramer 类，使其能够处理任意的多字节定界符。请确保你的实现是高效的。

3.扩展 DelimFramer 类实现“字节填充”，以使包含有定界符的消息也能够传输。（请在较好的计算机网络教程中参见该算法）

4. 假定所有字节值都有同样的可能，请问在一个包含了随机位值的消息中，通过 VoteMsgBin 的“魔术测试”的概率是多少？假设一个使用 ASCII 编码的文本消息发送给了一个处理二进制编码的投票消息 voteMsg 的程序，哪些字符出现在消息中的第一个字节时，可能使其通过“魔术测试”？

5.BruteForceEncoding 的 encodeIntBigEndian()只适用于满足了一定前提条件的情况下，如 $0 \leq size \leq 8$ 等。修改该方法，使其能够对这些前提条件进行检测，当条件不满足时抛出异常。