*Java Network Programming*

第4版

# Java网络编程

O'REILLY®

中国电力出版社

*Elliotte Rusty Harold* 著

李帅 荆涛 等译

# Java网络编程

《Java网络编程（第4版）》全面介绍了如何使用Java开发网络程序。你将学习如何使用Java的网络类库既快速又轻松地完成常见的网络编程任务，如编写多线程服务器、加密通信、广播到本地网络，以及向服务器端程序提交数据。

作者Elliotte Rusty Harold提供了真正可实用的程序来讲解他介绍的方法和类。第四版经过全面修订，已经涵盖REST、SPDY、异步I/O和很多其他高级技术。

- 研究Internet底层协议，如TCP/IP和UDP/IP。
- 了解Java的核心I/O API如何处理网络输入和输出。
- 探索InetAddress类如何帮助Java程序与DNS交互。
- 用Java的URI和URL类定位、识别和下载网络资源。
- 深入研究HTTP协议，包括REST、HTTP首部和cookie。
- 使用Java的底层Socket类编写服务器和网络客户端。
- 利用非阻塞I/O同时管理多个连接。

Elliotte Rusty Harold编写Java网络程序长达近20年。他不仅是一位资深作者，撰写过数十本有关Java、XML和HTML的图书，还经常在行业大会上发表演讲。他对多个开源项目做出过贡献，包括Jaxen XPath库和XOM。



O'REILLY®
oreilly.com.cn

ISBN 978-7-5123-6188-1
9 787512 361881 >
定价：78.00元

O'Reilly Media, Inc.授权中国电力出版社出版

第四版

# Java网络编程

*Elliotte Rusty Harold* 著
李帅 荆涛 等译



O'REILLY®

*Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo*

O'Reilly Media, Inc.授权中国电力出版社出版

中国电力出版社

**图书在版编目（CIP）数据**

Java网络编程：第4版/（美）哈诺德（Harold, E. R.）著；李帅等译. —北京：中国电力出版社，2014.9

书名原文：Java network programming, fourth edition

ISBN 978-7-5123-6188-1

I. ①J… II. ①哈… ②李… III. ①JAVA语言－程序设计 IV. ①TP312

中国版本图书馆CIP数据核字（2014）第151539号

北京市版权局著作权合同登记

图字：01-2014-4091号



封面设计/ Karen Montgomery，张健

出版发行/ 中国电力出版社（http://www.cepp.sgcc.com.cn）

地　　址/ 北京市东城区北京站西街19号（邮政编码100005）

经　　销/ 全国新华书店

印　　刷/ 北京丰源印刷厂

开　　本/ 787毫米×980毫米　16开本　29印张　545千字

版　　次/ 2014年9月第一版　2014年9月第一次印刷

印　　数/ 0001－3000册

定　　价/ 78.00元（册）

# O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

谨以此书献给我的小狗Thor。

## 作者介绍

**Elliotte**出生于路易斯安那州的新奥尔良，如今他还定期返回那里研究他的那盆秋葵。不过，他现在与他的妻子Beth、他的狗Thor一起，居住在布鲁克林附近的Prospect Height。他经常在行业会议上演讲，包括Software Development、Dr. Dobb's Architecture Design World、JavaZone、JAOO、SD Best Practices、Extreme Markup Languages和很多用户组。他的开源项目包括利用Java处理XML的XOM库，以及Amateur媒体播放器。

## 封面介绍

本书的封面动物是一只北美河獭（Lutra canadensis）。这些小的食肉动物会在美国和加拿大的水域出现，除苔地和美国西南部的闷热干燥区域不适合它们外，它们的栖息地几乎遍布各地。北美河獭大约重20磅，大概2.5英尺长，雌性一般比雄性长三分之一。它们的食物主要由鱼和青蛙等水生动物组成，但由于它们有三分之二的时间生活在陆地上，所以偶尔也会吃鸟类或啮齿动物。它们的毛分为两层——粗糙的外层毛和厚密的内层毛，可以有效地防御寒冷，事实上，它们看起来很喜欢在冰雪中玩耍。当潜水时，河獭的脉搏会从正常的每分钟170下降到20，这样就能保存氧气，可以在水下待得时间更长。这种动物喜爱群居，易于驯养。在欧洲，人们就曾训练一种河獭来捕鱼供人们食用。

# 目录

# 前言

Java 在过去20年中的发展着实惊人。既然Java的快速成长得益于Internet迅猛的发展，如果还认为利用Java进行网络编程对许多人来说仍然十分神秘，这就实在让人很难相信了。事情并不是这样。事实上，如本书所言，利用Java编写网络程序非常简单。过去在UNIX、Windows或Macintosh环境下有过网络编程经验的读者会惊喜地发现，利用Java编写同样的程序将会更为简单。Java核心API包括了完善的接口，可以提供大多数网络特性。实际上，对于用C或C++编写的应用层网络软件，用Java编写只会更简单，而不会更难。《Java网络编程（第四版）》将尽力向你展示如何利用Java的网络类库，快速而轻松地编写程序来完成一些常见的网络任务。这些任务包括：

- 通过HTTP浏览Web。
- 编写多线程服务器。
- 对通信进行加密，以保证机密性、真实性和消息完整性。
- 设计GUI客户端提供网络服务。
- 向服务器端程序提交数据。
- 使用DNS查找主机。
- 通过匿名FTP下载文件。
- 连接Socket完成底层网络通信。
- 组播到网络上的多个主机。

Java是第一个（不过不再是唯一的一个）提供了如此强大跨平台网络库的语言，可以处理所有这些任务。本书将向你展现这个函数库的强大能力，同时也会指出其复杂性。本

书的目的是使你能够将Java作为一个平台来完成重要的网络编程。为此，本书提供了网络基础的一般背景，并详细讨论了Java对于编写网络程序所提供的便利。你将学习到如何编写Java程序在Internet上共享数据，来实现游戏、协作、软件更新、文件传输等诸多方面。你还将对HTTP、SMTP、TCP/IP及其他支持Internet和Web的协议有深入的了解，了解其底层原理。当读完本书时，你将会获得必要的知识和工具，完全可以自行创建能够充分利用Internet的下一代软件。

# 关于第四版

1996年，在本书第一版的第1章中，我用了大量篇幅来介绍我认为可以用Java实现的那种动态的、分布式网络应用程序。在之后写这本书的后续版本时，最让人激动的一点是，几乎我预言的所有应用程序都得以实现。程序员们使用Java来查询数据库服务器，监视Web页面，控制望远镜，管理多用户游戏等，这些都是通过使用Java的内在功能来访问Internet。不论是一般意义上的Java，还是特别地用Java进行网络编程，都已经远远超出了广告宣传范畴，而进入了真正的实际应用程序。

本书也走过了一段相当长的路。第四版会用更多的笔墨来介绍HTTP和REST。HTTP原先只是众多网络协议之一，现在几乎可以把“之一”去掉，而作为网络协议的代名词。可以看到，其他协议通常都基于HTTP构建，在网络栈中建立自己的层。

Java 6、7和8中的java.net和支持包还有很多其他的小调整和更新，第四版中也将涵盖这些内容。这一版中介绍的新类包括：CookieManager、CookiePolicy、CookieStore、HttpCookie、SwingWorker、Executor、ExecutorService、AsynchronousSocketChannel、AsynchronousServerSocketChannel等。另外在Java的最后3个版本中，还为现有的一些类增加了很多其他方法，这些将在相应的章节中讨论。我还重写了这本书的很大一部分，以反映相应的变化，这包括一般意义上Java编程的变化，以及特殊意义上网络编程方面的改变。希望你能发现第四版与上一版相比，对Java网络编程的论述更为充分，更为准确，作为这一领域的权威教程和参考书，能让读者更乐于接受，生命力也将更长久。

# 本书结构

第1章“基本网络概念”，将详细阐释关于网络和Internet如何工作，程序员需要了解哪些知识。这一章还会介绍Internet的底层协议，如TCP/IP和UDP/IP。

接下来的两章将重点介绍Java编程的两个部分，这两个内容对几乎所有网络程序都至关重要，却经常被误解和误用，这就是I/O和线程。第2章“流”，将探讨Java的典型I/O模型，而不是新的I/O API，这些模型不会很快被废弃，它们仍然是大多数客户应用程序中

处理输入和输出的首选方式。理解Java在一般情况下如何处理I/O，这是理解Java如何处理网络I/O这种特殊情况的先决条件。第3章“线程”，将探讨多线程和同步，并特别强调线程如何用于异步I/O和网络服务器。

有经验的Java程序员可以略读或跳过这两章。不过，对所有人而言，第4章“Internet地址”都是必须要阅读的。它将展示Java程序如何通过`InetAddress`类与域名系统（Domain Name System，DNS）进行交互，这是所有网络程序都需要用到的一个类。一旦读完这一章，你就可以根据你的兴趣和需要直接跳到书中相关的部分继续阅读。

第5章“URL和URI”，将探讨Java的URL类，这是从多种网络服务器中下载信息和文件的一个抽象类，功能相当强大。基于URL类，你不必考虑服务器所用协议的细节，就能连接到网络服务器，并下载文件和文档。由此，你可以像访问HTTP服务器或读取本地硬盘上的文件一样，使用相同的代码来连接FTP服务器。你还将了解更新的URI类，相对于URL类，URI类与标准更一致，用于识别资源而非获取资源。

第6章“HTTP”将更深入地讨论HTTP，主要介绍REST、HTTP首部和cookie。第7章“URLConnection”将介绍如何使用URLConnection和HttpURLConnection类，从而不只是从Web服务器下载数据，还能上传文档和配置连接。

第8章到第10章将讨论Java用于网络访问的低层Socket类。第8章“客户端Socket”，介绍Java socket API，特别是Socket类。在此将展示如何编写与各种TCP服务器（包括whois、dict和HTTP服务器）交互的网络客户端。第9章“服务器Socket”，将展示如何使用ServerSocket类编写以上以及其他协议的服务器。最后，第10章“安全Socket”，将介绍如何使用安全Socket层（SSL）和Java安全Socket扩展（JSSE）来保护客户端-服务器通信。

第11章“非阻塞I/O”，涵盖了专门为网络服务器设计的新的I/O API。这些API能够让程序在试图读取或写入Socket前确定连接是否已经准备就绪。这就允许单个线程同时管理多个不同的连接，从而减轻虚拟机的工作负载。对于不需要同时打开多个连接的小服务器或小客户端来说，这些新的I/O API并没有太大帮助，但对于那些高吞吐量的服务器而言，却能提供显著的性能提升，使它们能够以网络所能提供的尽可能快的速度来传输网络能处理的尽可能多的数据。

第12章“UDP”，介绍用户数据报协议（User Datagram Protocol，UDP）及与之关联的`DatagramPacket`和`DatagramSocket`类，它们提供了快速、非可靠的通信。最后，第13章“IP组播”，展示了如何使用UDP同时与多个主机通信。

# 读者对象

本书假定读者熟悉Java语言和编程环境，还了解面向对象编程的一般概念。本书不是基础的语言教程。你应当完全熟悉Java的语法，而且编写过简单的应用程序。如果熟悉基本的Swing编程也很有好处，不过除了几个例子会用到Swing以外，一般情况下对此并没有要求。如果遇到某个网络编程问题需要更深入地理解（如线程和流），我也会在这本书中谈到，至少会做简单的介绍。

但是，本书不要求你有网络编程的经验。在这本书中，你会找到网络概念和网络应用程序开发的全面介绍。我不认为你能够随口说出很多网络编程的缩写词（TCP、UDP、SMTP等）。你将在这里学习需要了解的所有内容。

# Java版本

在Java 1.0之后，Java的网络类较之于核心API的其他部分，发展得非常缓慢。与AWT或I/O相比，几乎没有任何改变，仅仅有一些增加。当然，所有网络程序都广泛应用了I/O类，而且一些程序还大量使用了GUI。在编写本书时，我们假设你至少使用Java 5.0以上的版本。一般情况下，我会直接使用Java 5的特性，如泛型和改进的for循环等，而不会做更多解释。

对于网络编程，Java 5和Java 6的差别并不大。这两个版本的大多数例子看起来都完全相同。如果某个方法或类是Java 6、7或8新增的，我们会在它的声明后面加一个注释来特别指出，如下所示：

```
public void setFixedLengthStreamingMode(long contentLength) // Java 7
```

Java 7稍稍有些“好高骛远”，不过，有些情况下Java 7的特性可能特别有用或者很方便，例如，由于本书篇幅有限，要减少示例的篇幅以便放在这本书里，try-with-resources和multicatch就很有帮助，所以我并不避讳使用Java 7的特性，只是会特别指出使用了这样一些特性。

不过，总的来讲，Java的网络API自Java 1.0以来都相对稳定。Java 1.0之后的网络API很少被废弃，而且新增的特性也相对较少。即使在发布Java 8之后使用这本书应该也没有什么问题。但是，支持类中已经有越来越多新的API，特别是I/O，在Java 1.0之后它已经有3次重大修订。

# 关于示例

本书中介绍的大多数方法和类都会通过至少一个完整的运行程序来讲解，尽管这个程序

可能很简单。从我的经验来看，一个完整的运行程序对于展示方法的正确用法非常重要。如果没有一个实际的程序，就很容易深陷于众多术语当中，或者往往会掩盖作者自己也不清楚的知识点。Java API文档本身对于方法调用的描述通常过于简洁。在本书中，我会努力做到不过于啰唆也不过分简单。如果你对某个知识点已经很清楚，则完全可以跳过有关的说明。你不需要录入和运行这本书中的每一个示例，但是如果学习某个方法时遇到麻烦，则至少需要有一个运行示例才行。

每一章都包括至少一个（通常有多个）比较复杂的程序，会基于比较实际的设置来展示这一章中介绍的类和方法。它们通常会依赖这本书中没有讨论的Java特性。确实，在许多程序中，网络部件只是源代码的一小部分，通常也是不太难的部分。虽然如此，如果其他语言不像Java这样将网络摆在如此核心的地位上，要用那些语言来编写这些程序的话，绝不会像用Java编写那么简单。代码中的网络部分相当简单，这一点也反映出网络已经成为Java的一个核心特性，而并非程序本身要操心的琐事。本书中出现的所有示例程序都可以在线下载，通常这些示例会有所修改，还可能增加新内容。你可以从*http://www.cafeaulait.org/books/jnp4/*下载这些源代码。

我已经在Linux上测试过所有示例，在Windows和MacOS X上也测试了其中很多示例。这里给出的大多数示例都应当可以在支持Java 5或以后版本的其他平台、其他编译器和虚拟机上很好地工作。如果一个示例用Java 5或6无法编译，最常见的原因可能是其中使用了try-with-resources和multicatch。可以很容易地将这些例子重写为支持之前的Java版本，不过这样一来，代码会烦琐得多。

我确实很遗憾，由于篇幅限制，我不得不做出一些妥协。首先，我很少检查前置条件。大多数方法都假定会传入正确的数据，而且没有考虑null检查和保证代码质量的类似原则。另外，我把每个代码块的缩进减少为2个字符，每个延续行的缩进减少为4个字符，而没有采用Java的标准（分别缩进4个字符和8个字符）。希望这些缺陷不会影响到你。好的一方面是，这些让步确实让这一版比上一版简短得多（少了近百页）。

## 本书约定

正文为Minion Pro，常规字体（正如你现在所看到的）。

等宽印刷字体（`Constant width`）用于：

- 代码示例和片段。
- 在Java程序中出现的内容，包括关键字、操作符、数据类型、方法名、变量名、类名和接口名。

- 程序输出。
- 出现在HTML文档中的标记。

粗体等宽字体（**Constant width Bold**）用于：

- 要在屏幕上逐字输入的命令行和选项。

斜体字体（*italic*）用于：

- 新定义的术语。
- 路径名、文件名和程序名（但是，当程序名也是Java类名时，将像其他类名那样显示为等宽字体）。
- 主机和域名（*www.hpmor.com*）。
- URL（*http://www.cafeaulait.org/slides/*）。
- 其他书的标题（Java I/O）。

---

**提示**：表示一个提示、建议或一般说明。

---

**警告**：表示一个警告或警示。

---

重要的代码段和完整的示例一般都单独作为一段，例如：

```
Socket s = new Socket("java.oreilly.com", 80);
if (!s.getTcpNoDelay()) s.setTcpNoDelay(true);
```

当代码表示片段而非完整程序时，这暗指已经有适当的`import`语句。例如，在上面的代码段中，就可以假定已经导入了`java.net.Socket`。

有些示例混杂了用户输入和程序输出。在这种情况下，用户输入显示为粗体，如第9章中的下面这个例子：

```
% telnet rama.poly.edu 7
Trying 128.238.10.212...
Connected to rama.poly.edu.
Escape character is '^]'.
This is a test
This is a test
This is another test
This is another test
9876543210
9876543210
```

```
^]
telnet> close
Connection closed.
```

最后，尽管这里使用的许多示例只是玩具示例，并且不具有重用性，但我开发的一些类仍具有实用价值。你可以在你自己的代码中随意使用这些类或者其中的任何部分，而不需要特殊的许可。它们是公开的（不过，这绝对不包括那些说明文字）。

# 意见反馈

我很乐于听到来自读者的声音，无论是关于本书的一般意见，特定的修正，希望涵盖的其他主题，或者只是你自己在网络编程斗争中的故事，都可以向*elharo@ibiblio.org*发送电子邮件与我联系。但是要知道，我每天都会收到几百封邮件，无法亲自一一回复。为尽可能地得到亲笔回复，请注明你是本书的读者。如果某个程序不能像你预期的那样运行，并对此感到疑惑不解，请尽量简化情况的说明，并能再次生成这个bug，最好是一个类，将整个程序的文本粘贴在你的电子邮件的正文中。未经请求的附件会直接删除而不会打开。另外，一定要用你希望得到回复的那个地址来发送邮件，还要确保正确地设置了你的Reply-on（回复）地址！如果我花费了一个小时或更长时间来寻找一个有趣问题的答案，而且写了一个详细的回复，却发现邮件被退回，而退回的原因只是因为与我通信的人是在公用终端上发送邮件，没有设置浏览器首选项来包括其实际电子邮件地址，再没有比这更让人沮丧的事了。

我还坚信一句老话："如果你喜欢这本书，请告诉你的朋友。如果你不喜欢它，请告诉我。"我特别希望听到关于错误的信息。这是这本书的第四版。我已经竭尽所能让它成为一本完美的书，不过我还在努力。虽然我和O'Reilly公司的编辑们为本书奋力工作，但我确信肯定还有一些地方存在错误或排版问题。而且我相信至少还存在一个很棘手的大问题。如果你找到一个错误或排版问题，请告诉我，以便改正。我将把勘误发布在O'Reilly的Web网站上（*http://oreil.ly/java_np_errata*）。在向我报告错误之前，请检查这些页面，看看我是否已经知道了这个错误并且发布了勘误。所有已报告的错误都将在下一次印刷时改正。

# 使用代码示例

这本书将成为你工作的助手。一般说来，如果书中提供了代码示例，你可以在你的程序和文档中使用这些代码，除非复制使用了本书的大部分代码，否则不需要联系我们申请获得许可。例如，如果只是编写一个程序，其中用到了本书的几个代码段，这是不需要许可的。销售或发行O'Reilly图书的示例光盘则需要得到许可。如果引用本书的文字以

及利用书中的示例代码回答一个问题，这不需要专门获得许可。但是如果在你的产品文档大量使用本书中的示例代码，这是需要获得许可的。

我们希望大家使用代码时能注明引用出处，但并不强求。引用通常包括书名、作者、出版商和ISBN。例如：“*Java Network Programming*, Fourth Edition,by Elliotte Rusty Harold (O'Reilly). Copyright 2014 Elliotte Rusty Harold,978-1-449-35767-2”。

如果你认为对代码示例的使用超出了合理的使用范畴或上述许可范围，请随时联系我们：*permissions@oreilly.com*。

## Safari®图书在线

Safari图书在线（*www.safaribooksonline.com*）是一个按需而变的数字图书馆，通过图书和视频方式提供世界顶尖作者在技术和商业领域积累的专家经验。

技术专家、软件开发人员、Web设计人员和企业以及有创意的专业人员都使用Safari图书在线作为其主要资源来完成研究、解决问题、深入学习和资质培训。

Safari图书在线为机构、政府部门和个人提供了多种产品组合和定价程序。订阅者可以在一个可以快捷搜索的数据库中访问多家出版社提供的成千上万种图书、培训视频和正式出版前手稿，如O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology，以及其他数十家出版公司。关于Safari图书在线的更多信息，请访问我们的在线网站。

## 联系我们

请将关于本书的意见和问题通过以下地址提供给出版商：

美国：

O'Reilly Media，Inc.
1005 Gravenstein Highway North
Sebastopol，CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

针对这本书，我们还建有一个网页，列出了有关勘误、示例和其他信息。可以通过以下地址访问这个页面：

*http://oreil.ly/java-network-prgamming*

如果对这本书有什么意见，或者要询问技术上的问题，请将电子邮件发至：

*bookquestions@oreilly.com*

要想了解O'Reilly 图书、课程、会议和新闻的更多信息，请访问我们的网站：

*http://www.oreilly.com*
*http://www.oreilly.com.cn*

# 致谢

许多人都参与了本书的编写。我的编辑Mike Loukides，更使这本书增色不少，他提供了许多有用的意见，使本书大有改进。Peter“Peppar”Parnes博士在第13章中给予了我们极大的帮助。技术编辑们在搜寻错误和遗漏中提供了宝贵的援助和支持。Simon St. Laurent提供了一些宝贵的建议，指出哪些主题应当涵盖更多的内容。Scott Oaks为第3章提供了他在线程方面的经验，而且再一次地指出了他找到的许多小bug，从这里也可以看出，专家就是专家，多线程部分还是需要专家的审查。Ron Hitchens指出了新的I/O API中许多未被注意的方面。Marc Loy和Jim Elliott审查了书中一些最前沿的素材。Jim Farley和William Grosso在远程方法调用方面提供了许多有用的意见和援助。Timothy F. Rohaly很努力地确保我关闭了所有Socket，捕获了所有可能的异常，从而编写出最清楚、最安全和最典范的代码。John Zukowski找到了无数遗漏错误，要向他致以无尽的谢意。“鹰眼”Avner Gelb展示出惊人的能力，他居然能发现第一版中的错误，不知何故这些错误竟逃过了我自己、所有其他编辑和成千上万读者的眼睛。Alex Stangl和Ryan Cuprak在最新的这一版中继续帮助我们查找原来遗漏的错误和新出现的错误。

在此要感谢出版商，这并非出于惯例，而是因为出版商确实为公司、作者、编辑和产品工作人员等提供了此合作的条件。我想Tim O'Reilly应当得到这份特殊的荣誉，他使得O'Reilly Media绝对是作者为其踊跃写作的一家最好的出版公司。有这么一个人，没有他这本书就不会诞生，这个人就是他。如果你，读者，发现O'Reilly的图书一贯地比市场上大多数图书都要好，那么真的可以一直溯源归功于Tim。

感谢我的代理人David Rogelberg，是他让我相信，靠写书而不是在办公室工作来谋生也是可能的。在过去几年中，*ibiblio.org*的全体人员对于我更好地以各种方式与读者进行沟

通确实给予了很大的帮助。每一位褒奖或批评之前版本的读者都在帮助我编写更加优秀的新版本。所有这些人都应当得到感谢和应有的荣誉。最后，与往常一样，我要向我的妻子Beth致以最诚挚的谢意，没有她的爱和支持，这本书将永远无法诞生。

——Elliotte Rusty Harold
*elharo@ibiblio.org*

第1章

# 基本网络概念

网络编程不再是专家们的研究领域，它已经成为每个开发人员工具箱中的核心部分。如今，强调网络的程序比不涉及网络的更多。除了经典的应用程序，如电子邮件、Web浏览器和远程登录外，大多数主要的应用程序都有某种程度的内置网络功能。例如：

- 文本编辑器（如BBEdit）保存和打开直接来自FTP服务器的文件。
- IDE（如Eclipse和IntelliJ IDEA）与源代码存储库（如GitHub和Sourceforge）进行通信。
- 字处理软件（如Microsoft Word）从URL打开文件。
- 反病毒软件（如Norton AntiVirus）在每次计算机启动时，通过连接提供商的Web网站检查新的病毒定义。
- 音乐播放器（如Winamp和iTunes）向CDDB上传CD音轨长度，下载相应的音轨标题。
- 玩多人第一人称射击游戏（如Halo）的玩家实时互相厮杀。
- 运行IBM SurePOS ACE的超市现金登记程序对每次交易实时地与商店的服务器通信。服务器每晚向连锁的中心计算机上传其每天的收入情况。
- 日程计划应用程序（如Microsoft Outlook）自动与公司的员工实现日程同步。

Java是第一个从一开始就为网络应用而设计的编程语言。Java最初针对的是专用有线电视网，而不是Internet，不过首先考虑到了网络。最早的两个实用Java应用程序之一就是Web浏览器。随着Internet的不断发展，Java成为了唯一适合构建下一代网络应用程序的语言。

Java最大的秘密之一是，它简化了网络程序的编写。事实上，用Java编写网络程序几乎比任何其他语言都简单得多。这本书将为你展示数十个充分利用Internet的完整程序。有些是简单的教科书示例，而另外一些则是具有完备功能的应用程序。在查看这些有完整功能的应用程序时，你会注意到，其中用于网络的代码非常之少。即使在大量涉及网络的程序中，如Web服务器和客户端，其中几乎所有代码都是在处理数据或用户界面。程序中涉及网络的部分几乎总是最短、最简单的。简单地讲，Java应用程序通过Internet发送和接收数据非常容易。

在用Java（或者在这方面可以是任何语言）编写网络程序之前，需要理解一些网络背景概念，本章所涵盖的就是这些基本网络概念。从最一般的概念到最特定的概念，本章解释了你需要了解的一般网络，特别是基于IP和TCP/IP的网络以及Internet。这一章不是要教你如何连接网络或配置路由器，而是会让你学习到一些必要的内容来编写通过Internet实现通信的应用程序。本章涵盖的主题包括网络定义、TCP/IP分层模型、IP、TCP和UDP、防火墙和代理服务器、Internet以及Internet标准化过程。有经验的网络专家完全可以跳过这一章，直接阅读下一章，你将开始用Java开发所需的工具来编写你自己的网络程序。

# 网络

网络（network）是几乎可以实时相互发送和接收数据的计算机和其他设备的集合。网络通常用线缆连接，数据位转换为电磁波，通过线缆移动。不过，无线网络会通过无线电波传输数据，许多长距离的传输现在会用通过玻璃纤维发送可见光的光纤电缆来完成。传输数据的任何物理介质并没有什么神秘可言。从理论上讲，数据甚至可以通过用碳驱动的计算机发送烟信号来相互传输。这种网络的响应时间（和对环境的影响）可能很差。

网络中的每台机器称为一个节点（node）。大多数节点是计算机，但是打印机、路由器、网桥、网关、哑终端和可口可乐机也都是节点。你可以使用Java与可乐机进行交互，但多数情况下都是与其他计算机对话。具有完备功能的计算机节点也称为主机（host）。这里将用节点一词指代网络的所有设备，用主机一词表示通用的计算机节点。

每个网络节点都有地址（address），这是用于唯一标识节点的一个字节序列。你可以将这组字节看作是数字，但一般不能保证地址中的字节数或字节序（big endian或little endian）与Java中的简单数值类型一致。每个地址中的字节越多，可用的地址就越多，就可以有更多的设备同时连入网络。

不同的网络会以不同的方式分配地址。以太网（Ethernet）地址与物理以太网硬件关

联。以太网硬件的生产厂家使用预分配的厂商编码确保他们的硬件地址或与其他厂家的硬件地址不冲突。每家厂商都要负责保证不会生产出两块地址相同的以太网卡。Internet地址通常由负责分配地址的组织分配给计算机。不过，一个组织允许选择的地址由该组织的Internet服务供应商（ISP）分配。ISP从四个区域Internet注册机构之一获得IP地址（其中北美地区的注册机构是美国Internet号码注册中心（American Registry for Internet Numbers，ARIN）），而这四个机构的IP地址则由互联网名称与数字地址分配机构（Internet Corporation for Assigned Names and Numbers，ICANN）分配。

有些网络中，节点还有可以帮助人们进行标识的名字，如“www.elharo.com”或“Beth Harold's Computer”。某一时刻一个特定的名字通常就指示一个地址。但是，名字并不与地址锁定。名字可以改变，但地址保持不变，或者类似的，地址可以改变而名字保持不变。一个地址可以有多个名字，一个名字也可以指示多个不同的地址。

所有现代计算机网络都是包交换（分组交换）网络：流经网络的数据分割成小块，称为包（packet，也称分组），每个包都单独加以处理。每个包都包含了由谁发送和将发往何处的信息。将数据分成单独的带有地址的包，其最重要的优点是，多个即将交换的包可以在一条线缆上传输，这使得建立网络的成本更低：多个计算机可以互不干扰地共用同一条线缆（与之相反，如果使用传统的电话线，当你在一个交换区中打本地电话时，实际上你将独占从你的电话到通话人电话之间的一条线缆。当所有线缆都占用时，有时有紧急事件或者节假日里就会发生这种情况，那么并不是每个拿起电话的人都能听到拨号音。如果你一直不挂断，最后当线不忙时你才会听到拨号音。在一些电话服务不如美国的国家中，等待半个小时或更长时间才听到拨号音是很常见的事情）。分包还有另一个好处，这就是可以进行校验，用来检测包在传输中是否遭到破坏。

我们仍然遗漏了重要的一点：计算机来回传递数据时还需要提供些什么。协议（protocol）是定义计算机如何通信的一组明确的规则：包括地址格式、数据如何分包等。针对网络通信的不同方面，定义有很多不同的协议。例如，超文本传输协议（Hypertext Transfer Protocol，HTTP）定义了Web浏览器如何与服务器通信；在另一个方面，IEEE 802.3标准定义了另外一个协议，规定了数据位如何编码为某种特定类型线缆上的电信号。开放、公开的协议标准允许不同厂家的软件和设备相互通信：Web服务器不关心客户端是UNIX工作站、Android手机还是一个iPad，因为所有客户端都使用相同的HTTP，与平台无关。

## 网络的分层

通过网络发送数据是一项复杂的操作，必须仔细地协调网络的物理特性以及所发送数据的逻辑特征。通过网络发送数据的软件必须了解如何避免包的冲突，将数字数据转换为

模拟信号，检测和修正错误，将包从一台主机路由到另外的主机等。需要支持多个操作系统以及添加了异构网络电缆时，这个过程将变得更加复杂。

为了对应用程序开发人员和最终用户隐藏这种复杂性，网络通信的不同方面被分解为多个层。每一层表示为物理硬件（即线缆和电流）与所传输信息之间的不同抽象层次。在理论上，每一层只与紧挨其上和其下的层对话。将网络分层，这样就可以修改甚至替换某一层的软件，只要层与层之间的接口保持不变，就不会影响到其他层。

图1-1显示了你的网络中可能存在的协议栈。尽管如今在Internet上中间层协议相当稳定，但是上层和下层的协议变化很大。有些主机使用Ethernet；有些使用WiFi；有些使用PPP；还有一些主机使用其他的协议。类似地，这个栈顶层使用的协议完全取决于主机运行的程序。关键是，从栈的顶层来看，底层协议是什么并不重要，反之亦然，从底层来看，也不关心顶层协议是什么。这个分层模型实现了应用协议与网络硬件物理特性以及网络连接拓扑结构的解耦合。

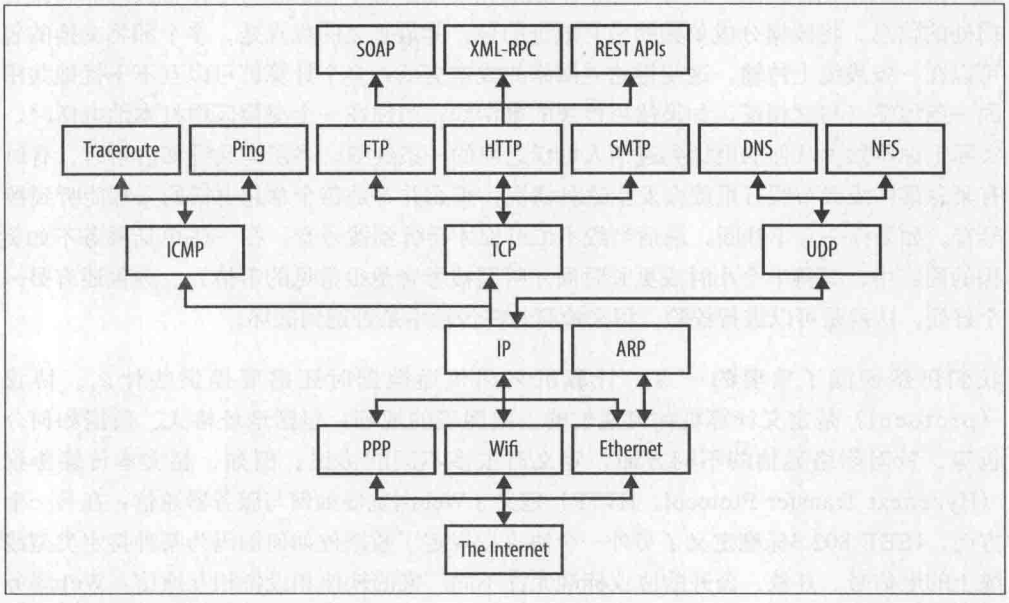


图1-1：网络不同层的协议

有几种不同的分层模型，分别适合特定类型网络的需要。本书采用适用于Internet的标准TCP/IP四层模型，如图1-2所示。在这个模型中，应用程序如Firefox和Warcraft运行在应用层，只与传输层对话。传输层只与应用层和网际层对话。网际层则只与主机网络层和传输层对话，绝不直接与应用层对话。主机网络层通过线缆、光纤或其他介质将数据移

动到远程系统的主机网络层，然后再通过上述各层将数据逐级上移传输到远程系统的应用层。

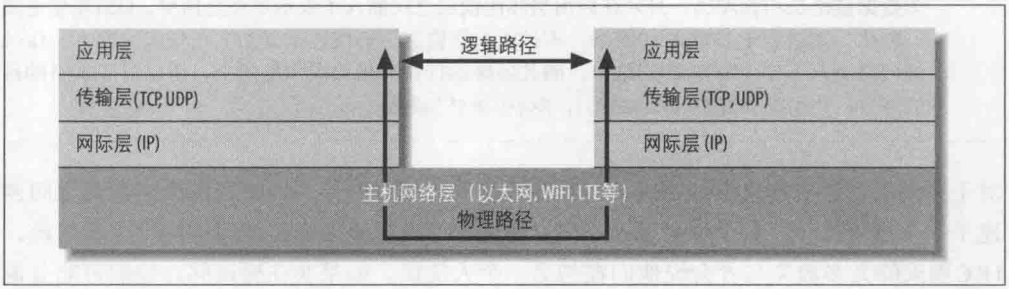


图1-2：网络分层

例如，当Web浏览器向Web服务器发送获取网页的请求时，浏览器实际上只与本地客户机的传输层对话。传输层将请求分解为TCP片，向数据添加序列号和校验和，然后将请求传递给本地网际层。网际层根据本地网络所需的大小将各TCP片分成IP数据报，并传递到主机网络层以便通过线缆传输数据。主机网络层将数字数据编码为适合特定物理介质的模拟信号，将请求发送到线缆，目标地址的远程系统的主机网络层可以由此读取请求。

远程系统的主机网络层将模拟信号解码为数字数据，将生成的IP数据报传递给服务器的网际层。网际层简单地检查IP数据报是否被破坏，如果已经分片则重组数据，然后传递给服务器的传输层。服务器的传输层检查是否所有的数据都已到达，对于丢失或破坏的部分则要求重传（这个请求实际上将向下通过服务器的网际层，再通过服务器的主机网络层，回到客户端系统，然后再在客户端系统向上返回客户端的传输层，传输层再通过本地网际层和主机网络层重传前面丢失的数据。所有这些对于应用层是完全透明的）。一旦服务器的传输层接收到足够多的连续顺序数据报，就将其重组写入一个流，由服务器应用层上运行的Web服务器读取。服务器响应这个请求，再通过服务器系统的各个分层发回响应，通过Internet进行传输并分发给Web客户端。

可以猜到，实际的过程更为错综复杂。主机网络层是最复杂的，特意地隐藏了很多细节。例如，通过Internet发送的数据在到达最终目的地之前，完全有可能经过几个路由器以及相应的分层。可能需要从大气中的无线电波转换为铜线缆中的电子信号，再转换到光纤电缆中的光脉冲，然后再反过来，从光脉冲转换到电子信号再到无线电波。不过，90%的情况下Java代码都将在应用层工作，只需要与传输层对话。其他10%的时间会在传输层处理，与应用层或网际层对话。主机网络层的复杂性对你是隐藏的，这是分层模型的关键。

**提示：** 如果阅读网络相关文献，你可能会看到另一个七层模型，称为开放系统互联参考模型（Open Systems Interconnection Reference Model，OSI）。对于Java网络程序，OSI模型过于复杂了。OSI模型与本书使用的TCP/IP模型之间最大的区别是，OSI模型将主机网络层分为数据链路层和物理层，另外在应用层和传输层之间插入了表示层和会话层。OSI模型更加一般化，更适合于非TCP/IP网络，不过大多数情况下仍然过于复杂。在任何情况下，Java的网络类只工作于TCP/IP网络中，而且始终运行在应用层或传输层上，所以对于本书的目的而言，使用更加复杂的OSI模型并不会带来任何好处。

对于应用层，它看起来像是在直接与其他系统的应用层对话，网络在两个应用层之间创建了一个逻辑路径。如果你考虑一下IRC聊天会话，就能很容易地理解这个逻辑路径。IRC聊天的大多数参与者会说他们在与另一个人交谈。如果实在要追问，他们可能会说是在与自己的计算机交谈（实际就是应用层），这个计算机再与另一个人的计算机交谈，那台计算机则与那个人交谈。深于一层的所有细节实际上都是不可见的，也确实应该如此。下面我们来详细地考虑各个分层。

## 主机网络层

作为Java程序员，你处在网络食物链中相当高的位置。在你的雷达探测之下发生着很多事情。在基于IP的Internet（Java唯一真正理解的网络）的标准参考模型中，网络中隐藏的部分属于主机网络层（host-to-work layer，也称为链路层、数据链路层或网络接口层）。主机网络层定义了一个特定的网络接口（如以太网卡或WiFi天线）如何通过物理连接向本地网络或世界其他地方发送IP数据报。

主机网络层中，由连接不同计算机的硬件（线缆、光纤电缆、无线电波或烟信号）组成的部分有时称为网络的物理层。作为Java程序员，不需要担心这一层，除非出现了问题，比如插头从计算机后面掉了下来，或者有人挖断了你与外部世界之间的T-1线。换句话说，Java从来都看不到物理层。

需要考虑主机网络层和物理层的主要原因是性能（如果你认为有必要考虑）。例如，如果你的客户端使用速度很快、很可靠的光纤连接，与在北大西洋的一个油井钻台上使用大延迟的卫星连接相比，协议和应用程序的设计就要有所不同。另外，如果你的客户端参与一个3G数据计划，带宽相对低时要按字节收费，你就要做出不同的选择。如果你要写一个一般的消费者应用，以上任何客户端都有可能使用这个应用，你就要尽量找到一个最佳的折中点，或者可能甚至要检测并动态适应不同的客户端功能。不过，不论你遇到哪一种物理链路，在这些网络上用来完成通信的API都是一样的。之所以能够做到这一点，就是因为有网际层。

## 网际层

网络的下一层，这也是需要你考虑的第一层，就是网际层（internet layer）。在OSI模型中，网际层使用了一个更一般的名字，称为网络层（network layer）。网络层协议定义了数据位和字节如何组织为更大的分组，称为包，还定义了寻址机制，不同计算机要按这个寻址机制查找对方。网际协议（IP）是世界上使用最广泛的网络层协议，也是Java唯一理解的网络层协议。

实际上，这是两个协议：IPv4和IPv6，IPv4使用32位地址，IPv6使用128位地址，另外还增加了一些技术特性来帮助完成路由。写这本书时，IPv4仍占Internet 业务流量90%以上的份额，不过IPv6正在迎头赶上，很可能在这本书下一版问世时超越IPv4。这是两个完全不同的网络协议，如果没有特殊的网关和/或隧道协议，即使在相同的网络上它们也无法做到互操作，不过Java几乎对你隐藏了所有这些区别。

在IPv4和IPv6中，数据按包在网际层上传输，这些包称为数据报（datagram）。每个IPv4数据报包含一个长度为20至60字节的首部，以及一个包含多达65 515字节数据的有效载荷（payload）。实际上，大多数IPv4数据报都小得多，从几十字节到稍大于8K字节不等。IPv6数据报包含一个更大的首部，数据可以多达4G字节。

图1-3展示各个部分在IPv4 数据报中是如何排列的。所有位和字节都采用big-endian方式，由左至右为最高位到最低位。

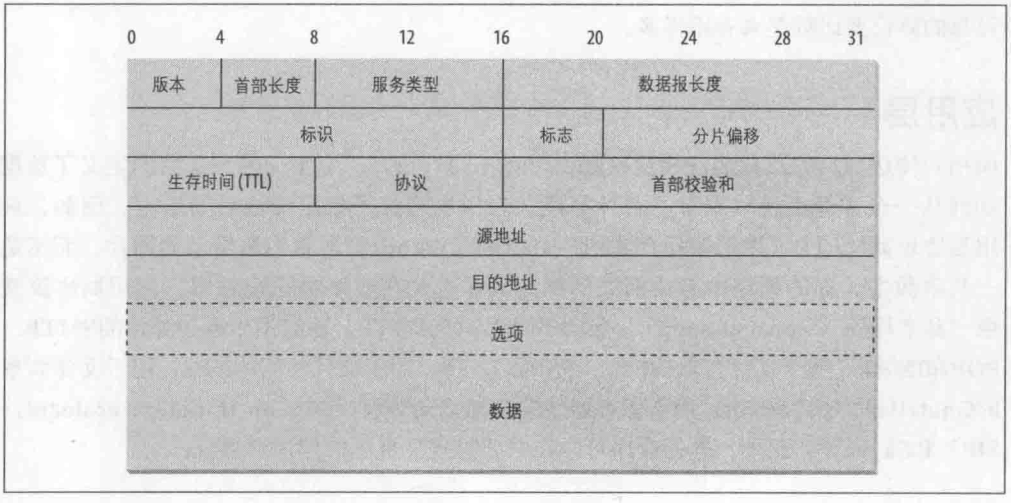


图1-3：IPv4数据报结构

除了路由和寻址，网际层的第二个作用是支持不同类型的主机网络层相互对话。Internet路由器会完成WiFi和Ethernet、Ethernet和DSL、DSL和光纤往返等协议之间的转换。如果没有网际层或类似的分层，则每个计算机只能与同一类网络上的其他计算机对话。网际层负责使用同构协议将异构网络相互连接。

## 传输层

原始数据报有一些缺点。最显著的缺点是不能保证可靠传送，即使能传送也可能在传输中遭到破坏。首部校验和只能检测首部中的破坏情况，而不能检测数据报中的数据部分。最后，即使数据报能到达目的地而未被破坏，也不一定会以发送时的顺序到达。各个数据报可能会经过不同路由从源到达目的地。如果数据报A在数据报B之前发送，这并不意味着数据报A会在数据报B之前到达。

传输层（transport layer）负责确保各包以发送的顺序接收，并保证没有数据丢失或破坏。如果丢包，传输层会请求发送方重传这个包。为实现这个目标，IP网络会给每个数据报添加一个附加首部，其中包含有更多信息。这一层上主要有两个协议。第一个是传输控制协议（Transmission Control Protocol，TCP），这是一个开销很高的协议，支持对丢失或破坏的数据进行重传，并按照发送时的顺序进行传送。第二个协议是用户数据报协议（User Datagram Protocol，UDP），它允许接收方检测被破坏的包，但不保证这些包以正确的顺序传送（或者包有可能根本未传送）。但是，UDP通常比TCP快。TCP称为可靠的（reliable）协议；UDP是不可靠的（unreliable）协议。后面我们将看到，不可靠的协议要比听起来有用得多。

## 应用层

向用户传送数据的层称为应用层（application layer）。它下面的三层共同定义了数据如何从一台计算机传输到另一台计算机。应用层确定了数据传输后的操作。例如，应用层协议如HTTP（用于国际互联网）可以确保Web浏览器将图像显示为图片，而不是一长串数字。你的程序中有关网络的部分大多都是在应用层花费时间。应用层协议就像一盆字母汤（alphabet soup）；除了用于Web的HTTP，还有用于电子邮件的SMTP、POP和IMAP；用于文件传输的FTP、FSP和TFTP；用于文件访问的NFS；用于文件共享的Gnutella和 BitTorrent；用于语音通信的会话启动协议（Session Initiation Protocol，SIP）和Skype等。此外，你的程序可以在必要时定义自己的应用层协议。

# IP、TCP和UDP

IP（网际协议）是冷战时期由军方资助开发的，所以最后包含了大量军方关心的特性。

首先，它必须健壮。如果前苏联以核武器攻击了克利夫兰的路由器，则整个网络不能停止运转，所有消息仍然必须到达预期的目的地（当然除了那些要到达克利夫兰的消息）。因此IP设计为允许任意两点之间有多个路由，可以绕过被破坏的路由器实现数据包的路由。

其次，军方有多种不同类型的计算机，所有这些计算机必须都能相互通话。因此IP必须是开放的，与平台无关；如果有一个协议用于IBM主机，另一个协议用于PDP-11，这样并不好。IBM主机有可能需要与PDP-11以及周围任何其他奇怪的计算机进行通话。

由于两点间存在多个路由，并且两点间的最短路径可能由于网络业务流量或其他因素（如克利夫兰的存在）而随时间改变，所以构成某个特定数据流的包可能不会采用相同的路由。另外，即使它们全部到达，也可能不会以发送的顺序到达。为了改进这种基本机制，将TCP置于IP之上，使连接的两端能够确认接收到IP包，以及请求重传丢失或被破坏的包。此外，TCP允许接收端的包按发送时的顺序重新组合在一起。

不过，TCP会有很大开销。因此，如果数据的顺序不是特别重要，而且单个包的丢失不会完全破坏数据流，那么有时也可以使用UDP发送数据包，而不需要TCP提供的保证。UDP是不可靠的协议，它不能保证包一定到达目的地，也不保证包会以发送时相同的顺序到达。虽然这对于文件传输等用途来说存在问题，但是在有些应用程序中，即使丢失部分数据最终用户也不会注意到，对于这种应用程序而言，UDP则完全可以接受。例如，丢失视频或音频中的一些数据位不会造成太大的质量下降。如果要等待类似TCP等协议请求重传丢失的数据，那才会是更严重的问题。此外，可以在应用层的UDP数据流中建立纠错码，来解决数据丢失问题。

可以在IP之上运行很多其他协议。最常使用的是ICMP，即网际控制消息协议（Internet Control Message Protocol），它使用原始IP数据报在主机之间传递错误消息。使用这个协议最著名的应用是ping程序。Java不支持ICMP，也不允许发送原始IP数据报（而只允许发送TCP片或UDP数据报）。Java支持的协议只有TCP和UDP，以及建立在TCP和UDP之上的应用层协议。所有其他传输层、网际层和更底层的协议，如ICMP、IGMP、ARP、RARP、RSVP和其他协议在Java程序中都只能通过链接到原生代码来实现。

## IP地址和域名

作为Java程序员，你不需要担心IP的内部工作原理，但你必须了解寻址。IPv4网络中的每台计算机都由一个4字节的数字标识。一般写为点分四段（dotted quad）格式，如199.1.32.90，这4个数中，每个数都是一个无符号字节，范围从0到255。IPv4网络中的每台计算机都有唯一的4字节地址。当数据通过网络传输时，包的首部会包括要发往的机

器地址（目的地址）和发送这个包的机器地址（源地址）。沿路的路由器通过检查目的地址来选择发送数据包的最佳路由。包括源地址是为了让接收方知道要向谁回复。

可能的IP地址有40亿多一点，无法做到地球上每人一个地址，更无法做到每台计算机一个地址。更糟的是，地址的分配并不是很高效。2011年4月，亚洲和澳大利亚的地址已经用光。无法再向这些地区分配更多的IPv4地址。在此之后，这些地区只能通过对现有的地址进行回收和再分配来维持。2012年9月，欧洲也用完了所有地址。北美洲、拉丁美洲和非洲只剩下为数不多的IP地址可以分配，不过也坚持不了多久。

现在正在向IPv6缓慢地过渡，它将使用16字节地址。这样就能有足够的IP地址来标识每个人、每台计算机甚至地球上的每一个设备。IPv6地址通常写为冒号分隔的8个区块，每个区块是4个十六进制数字，如*FEDC:BA98:7654:3210:FEDC:BA98:7654:3210*。前导的0不需要写。两个冒号表示多个0区块，但每个地址中至多出现一次。例如，*FEDC:0000:0000:0000:00DC:0000:7076:0010*可以写为更紧凑的形式，如*FEDC::DC:0:7076:10*。在IPv6和IPv4的混合网络中，IPv6地址的最后4字节有时写为IPv4的点分四段地址。例如，*FEDC:BA98:7654:3210:FEDC:BA98:7654:3210*可以写为*FEDC:BA98:7654:3210:FEDC:BA98:118.84.50.16*。IPv6只在Java 1.4及以后版本中支持。Java 1.3及以前版本只支持4字节地址。

虽然计算机可以轻松地处理数字，但人类对于记忆数字却不在行。因此，开发了域名系统（Domain Name System，DNS），用来将人类易于记忆的主机名（如www.oreilly.com）转换为数字Internet地址（如208.201.239.101）。当Java程序访问网络时，它们需要同时处理数字地址和相应的主机名。这些方法由`java.net.InetAddress`类提供，第4章将进行讨论。

有些计算机（尤其是服务器）有固定的地址。其他计算机（特别是局域网和无线连接上的客户端）可能每次启动时会收到不同的地址，这通常由DHCP服务器提供。你只需要记住IP地址可能会随着时间而改变，写代码时不要假定系统有相同的IP地址。例如，当保存应用程序状态时不要存储本地IP地址。相反，要在程序每次启动时进行新的查询。还有一种可能（尽管可能性不大），IP地址有可能在程序运行时改变（例如，DHCP租期到期），所以应当每次需要时检查当前的IP地址，而不是将其缓存。除此之外，动态分配和手工分配的地址之间的区别对Java程序没有影响。

有一些地址区块和模式很特殊。以10.、172.16.、172.31.和192.168.开头的所有IPv4地址都未分配。这些地址可以在内部网使用，但是使用这些地址的主机不允许加入全球Internet。这些不可路由的地址对于建立Internet上看不到的专用网会很有用。以127开头的IPv4地址（最常见的是127.0.0.1）总表示本地回送地址（local loopback address）。也

就是说，这些地址总指向本地计算机，而不论你在哪个计算机上运行。这个地址的主机名通常是*localhost*。在IPv6中，回送地址为0:0:0:0:0:0:0:1 (也就是 ::1)。地址0.0.0.0总指示起始主机，但是只能用作为源地址，而不能作为目的地址。类似的，所有以0.（8个二进制0）开头的IPv4地址都指示同一个本地网络上的一个主机。

4字节都使用相同数字的IPv4地址（如255.255.255.255）是一个广播地址。发送到这个地址的包将由本地网络上的所有节点接收，但不能超越这个本地网络。这通常用来完成发送。例如，一个临时使用的客户端（如笔记本电脑）启动时，它会向255.255.255.255发送一个特定的消息，查找本地DHCP服务器。这个网络上的所有节点都接收到这个包，不过只有DHCP服务器做出响应。具体地，它会向这个笔记本电脑发送本地网络配置的有关信息，包括这个笔记本电脑在余下的会话中要使用的IP地址以及用来解析主机名的DNS服务器的地址。

## 端口

如果每台计算机一次只做一件事情，那么地址可能就足够了。但是，现代计算机同时要做很多不同的事情。电子邮件需要与FTP请求分开，而FTP又要与Web业务流分开。这是通过端口（port）实现的。每台有IP地址的计算机都有几千个逻辑端口（确切地讲，每个传输层协议有65 535个端口）。这些只是计算机内存中的抽象，不表示任何物理实物，与USB端口不同。每个端口由1到65 535之间的一个数字标识。每个端口可以分配给一个特定的服务。

例如，Web的底层协议HTTP一般使用端口80。我们说Web服务器在端口80监听（listen）入站连接（incoming connection）。当数据发送到特定IP地址的某个机器上的Web服务器时，它还会发送到该机器的特定端口（通常是端口80）。接收方检查接口收到的各个包，将数据发送给监听这个端口的程序。各种通信业务流就是这样区分的。

1到1023的端口号保留给已知的服务，如finger、FTP、HTTP和IMAP。在UNIX系统上，包括Linux和Max OS X，只有以root用户运行的程序才可以接收这些端口的数据，但是所有程序都可以向这些端口发送数据。在Windows上，所有程序都可以使用这些端口，不需要专门的特权。表1-1给出了本书所讨论协议的已知端口。这种分配不是绝对的，特别是Web服务器通常就会在80以外的端口运行，可能是由于需要在同一台机器上运行多个服务器，或者因为安装服务器的人没有在端口80运行的root特权。在UNIX系统上，文件*/etc/services*存储了一个相当完整的端口分配列表。

表1-1：已知端口分配

| 协议 | 端口 | 协议 | 用途 |
|---|---|---|---|
| echo | 7 | TCP/UDP | echo是一个测试协议，通过回显另一台机器的输入来验证两台机器能否连接 |
| discard | 9 | TCP/UDP | discard是一种用处不太大的协议，它将忽略服务器收到的所有数据 |
| daytime | 13 | TCP/UDP | 提供服务器当前时间的ASCII表示 |
| FTP数据 | 20 | TCP | FTP使用两个已知端口。这个端口用来传输文件 |
| FTP | 21 | TCP | 这个端口用来发送FTP命令，如put和get |
| SSH | 22 | TCP | 用于加密的远程登录 |
| Telnet | 23 | TCP | 用于交互式远程命令行会话 |
| smtp | 25 | TCP | 简单邮件传输协议（Simple Mail Transfer Protocol）用来在机器间发送邮件 |
| time | 37 | TCP/UDP | 时间服务器返回服务器从1900年1月1日子夜后过去的秒数，这是一个4字节有符号big-endian整数 |
| whois | 43 | TCP | 用于Internet网络管理的简单目录服务 |
| finger | 79 | TCP | 返回本地系统中用户（或多个用户）有关信息的服务 |
| HTTP | 80 | TCP | 国际互联网的底层协议 |
| POP3 | 110 | TCP | 邮局协议版本3（Post Office Protocol Version 3）协议可将积累的电子邮件从主机传输到偶然连接的客户端 |
| NNTP | 119 | TCP | Usenet新闻传输。正式说法为“网络新闻传输协议”（Network News Transfer Protocol） |
| IMAP | 143 | TCP | Internet消息访问协议（Internet Message Access Protocol）是访问存储在服务器上的邮箱的协议 |
| dict | 2628 | TCP | 提供单词定义的UTF-8编码字典服务 |

# Internet

Internet是世界上最大的基于IP的网络。它是所有七个大洲（包括南极洲）多个不同国家的计算机使用IP相互对话的一个无组织的集合。Internet上每台计算机都至少有一个标识此计算机的IP地址。大多数还有至少一个主机名映射到这个IP地址。Internet不属于任何人（不过它的各个部分有相应的所有者）。它不受任何人控制，这不表示某些政府没有做过这种尝试。它只是约定以一种标准方式相互对话的一个非常大的计算机集合。

Internet不是唯一的基于IP的网络，但却是最大的一个。其他IP网络称为internet（首字母i小写）：例如没有连接Internet的高安全性内部网。Intranet大致描述了公司将大量数据置于内部Web服务器的实践做法，这些数据对本地网络以外的用户不可见。

除非在与更宽网络物理隔离的高安全性环境中工作，否则你使用的internet很可能就是Internet。为确保Internet上不同网络中的主机可以相互通信，就需要遵守一些对纯粹内部internet不适用的规则。最重要的规则是要处理不同组织、公司和个人地址的分配。如果每个人都随心所欲地随机挑选Internet地址，那么有相同地址的不同计算机出现在Internet上立即就会引起冲突。

## Internet地址分块

为避免这个问题，区域Internet注册机构会为Internet服务提供商（ISP）分配IPv4地址块。当公司或组织要建立一个基于IP的网络连接到Internet时，它们的ISP会给他们分配一个地址块。每个地址块有固定的前缀。例如，如果前缀是216.254.85，那么本地网络可以使用从216.254.85.0到216.254.85.255的地址。由于这个块固定了前24位，所以称为/24。 /23指定了前23位，而留出9位表示总共$2^9$或512个本地IP地址。/30子网（最小的子网）指定了子网中IP地址的前30位，留出2位表示总共$2^2$或4个本地IP地址。不过，所有块中最低地址用于标识网络本身，最高地址是这个网络的一个广播地址，所以比你原先预想的要少两个地址。

## 网络地址转换

出于IP地址越来越稀缺，而对原始IP地址的需求越来越大，如今大多数网络都使用了网络地址转换（Network Address Translation，NAT）。基于NAT的网络中，大多数节点只有不可路由的本地地址，这些地址可能从10.x.x.x、172.16.x.x到172.31.x.x，或192.168.x.x选择。将本地网络连接到ISP的路由器会把这些本地地址转换为更小的一组可路由的地址。

例如，我的公寓里有大约十来个IP节点，它们都共用一个外部可见的IP地址。我现在用的这个计算机的IP地址是192.168.1.5，不过，在你的网络上这个地址可能指示一个完全不同的主机（如果存在这样一个主机）。另外，你也无法向192.168.1.5发送数据到达我的计算机。实际上，必须把数据发送到216.254.85.72（即使如此，只有当我把NAT路由器配置为将入站连接传递到192.168.1.5时，数据才会真正送达我的计算机）。

路由器会监视出站和入站连接，调整IP包中的地址。对于出站的包，它将源地址改为路由器的外部地址（在我的网络上这是216.254.85.72）。对于入站的包，它将目的地址改为一个本地地址，如192.168.1.12。它如何记录哪些连接来自或发往哪台内部计算机，这

对于Java程序员并不是特别重要。只要正确地配置了你的机器，这个过程基本上就是透明的。你只需要记住外部地址和内部地址有可能不同就行了。

最后要说明的是，IPv6会使这里的大部分内容变得过时。NAT会毫无意义，不过防火墙还是有用的。仍然有可以路由的子网，不过那些子网会大得多。

## 防火墙

Internet上有些顽皮的人。为了把他们关在门外，在本地网络建立一个访问点，检查所有进出该访问点的业务流通常很有用。位于Internet和本地网络之间的一些硬件和软件会检查所有进出的数据，以保证其合法性，这就称为防火墙（firewall）。防火墙通常是将本地网络连接到更大的Internet的路由器的一部分，还可以完成其他任务，如网络地址转换。此外，防火墙也可以是单独的机器。现代操作系统如Mac OS X和红帽Linux通常有内置的个人防火墙，只监视发送到这个机器的业务流。无论采用何种方法，防火墙都要负责检查传入或传出其网络接口的各个包，根据一组规则接收或拒绝这些包。

过滤通常是基于网络地址和端口的。例如，所有来自C类网络193.28.25.x的通信会被拒绝，因为你过去遭遇过这个网络中黑客的攻击。出站SSH连接可能是允许的，但入站SSH连接不允许。端口80（Web）的入站连接是允许的，但只限于公司的Web服务器。更智能的防火墙会查看包的内容，确定是否接收或拒绝。防火墙具体的配置（哪些数据包允许通过而哪些不允许）取决于各个网站的安全需求。Java与防火墙没有太大关系，除非防火墙总是碍你的事。

## 代理服务器

代理服务器（proxy server）与防火墙有关。如果说防火墙会阻止一个网络上的主机与外界直接建立连接，那么代理服务器就起到了中间人的作用。这样一来，如果防火墙阻止一个机器连接外部网络，这个机器可以请求本地代理服务器的Web页面，而不是直接请求远程Web服务器的Web页面。然后代理服务器会请求Web服务器的页面，将响应转发给最初发出请求的机器。代理还可以用于FTP服务和其他连接。使用代理服务器的安全优势之一是外部主机只能看到代理服务器，而不会知道内部机器的主机名和IP地址，这就使得攻击内部网络更加困难。

防火墙一般工作于传输层或网际层，而代理服务器通常工作于应用层。代理服务器对一些应用层协议非常了解，如HTTP和FTP（一个值得注意的例外是SOCKS代理服务器，它工作于传输层，可以代理所有TCP和UDP连接，而不考虑应用层协议）。可以检查通过代理服务器的包，确保其中包含适当类型的数据。例如，看起来包含Telnet数据的FTP包可能会被拒绝。图1-4展示了代理服务器在分层模型中的位置。

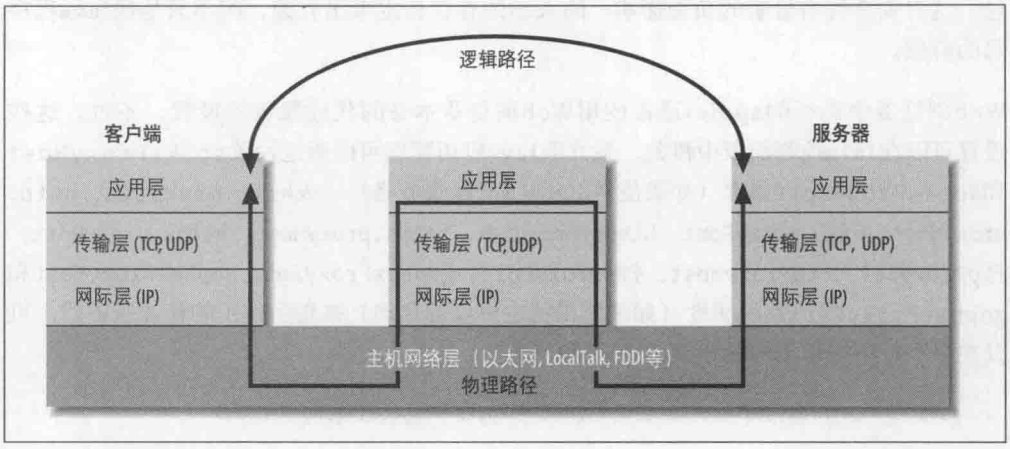


图1-4：通过代理服务器的分层连接

只要所有对Internet的访问都通过代理服务器转发，那么访问就可以受到严格的控制。例如，公司可能选择阻止访问*www.playboy.com*，但允许访问*www.microsoft.com*。一些公司允许入站FTP，但不允许出站FTP，这样机密数据就不会容易地被非法带出公司。其他公司已经开始使用代理服务器来跟踪员工的Web使用情况，这样可以看到谁在利用Internet获取技术支持，而谁在利用它找私人朋友。

代理服务器还可以用来实现本地缓存（local caching）。当请求Web服务器的文件时，代理服务器首先查看此文件是否已在缓存中。如果文件在缓存中，那么代理服务器将提供缓存中的文件，而不是Internet上的文件。如果这个文件不在缓存中，那么代理服务器将获取此文件，转发给请求方，并将它存储在缓存中，供下次请求使用。这种机制可以显著地降低Internet连接的负载，大大提高响应时间。美国在线（America Online）运转着世界上最大的代理服务器“场”之一，可以加快向用户传输数据的速度。如果你查看Web服务器的日志文件，可能会发现aol.com域客户的一些点击记录，但不像你想象的那么多，要知道AOL用户已超过300万。这是因为AOL代理服务器从其缓存中提供了许多页面，而不是为每位用户都重新请求页面。很多其他大的ISP也是这样做的。

代理服务器最大的问题在于它无法应对所有协议。通常已有的协议如HTTP、FTP和SMTP允许通过，而更新的协议如BitTorrent则不允许通过（有些网络管理员会认为这应算是一项功能）。在快速改变的Internet世界，这是一个很大的缺点，对于Java程序员而言更是一个缺点，因为它限制了定制协议的有效性。使用Java可以很容易地创建为你的应用而优化的新协议，这通常也很有用。但是，没有代理服务器能理解这些独一无二的协议。因此，有些开发人员开始通过HTTP来接入他们的协议，最著名的是SOAP。不

过，这对安全性有显著的负面影响。防火墙的存在肯定事出有因，而不只是找Java程序员的麻烦。

Web浏览器中运行的applet通常使用Web浏览器本身的代理服务器设置，不过，这些设置可以在Java控制面板中覆盖。独立的Java应用程序可以通过设置socksProxyHost和socksProxyPort属性（如果使用SOCKS代理服务器），或http.proxySet、http.proxyHost、http.proxyPort、https.proxySet、https.proxyHost、https.proxyPort、ftpProxySet、ftpProxyHost、ftpProxyPort、gopherProxySet、gopherProxyHost和gopherProxyPort系统属性（如果使用特定协议的代理）来指示使用的代理服务器。可以在命令行中使用-D参数设置系统属性，如下：

```
java -DsocksProxyHost=socks.cloud9.net -DsocksProxyPort=1080 MyClass
```

# 客户/服务器模型

大多数现代网络编程都基于客户/服务器模型。客户/服务器应用程序一般将大量数据存储在昂贵的高性能服务器或服务器云上，而大多数程序逻辑和用户界面由客户端软件处理，这些客户端软件运行在相对便宜的个人计算机上。在多数情况下，服务器主要发送数据，而客户端主要接收数据，但很少有一个程序只发送或只接收数据。更有可能的是客户端发起对话，而服务器等待客户端与它开始对话。图1-5展示了这两种可能性。在有些情况下，同一个程序会同时作为客户端和服务器。

你已经熟悉了许多客户/服务器系统的例子。在2013年，Internet上最流行的客户/服务器系统是Web。Web服务器（如Apache）响应Web客户端（如Firefox）的请求。数据存储在Web服务器上，会发送给请求数据的客户端。除了最初的页面请求，几乎所有数据都从服务器向客户端传输，而不是从客户端传输至服务器。FTP是符合客户/服务器模型的更古老的服务。FTP使用不同的应用协议和不同的软件，但依然分为发送文件的FTP服务器和接收文件的FTP客户端。人们通常使用FTP从客户端向服务器上传文件，所以很难讲数据传输主要是一个方向传输，但是要说FTP客户发起连接而FTP服务器进行响应，这仍然是正确的。

不是所有应用程序都简单地符合客户/服务器模型。例如，在网络游戏中，看起来两个玩家都能大致相同地来回发送数据（至少在公平游戏中是如此）。这种连接称为“对等”（peer-to-peer）连接。电话系统就是典型的对等网络例子。每部电话都可以呼叫另外的电话，或者被另外的电话呼叫。你不需要购买一部电话发送呼叫，另一部电话接收呼叫。

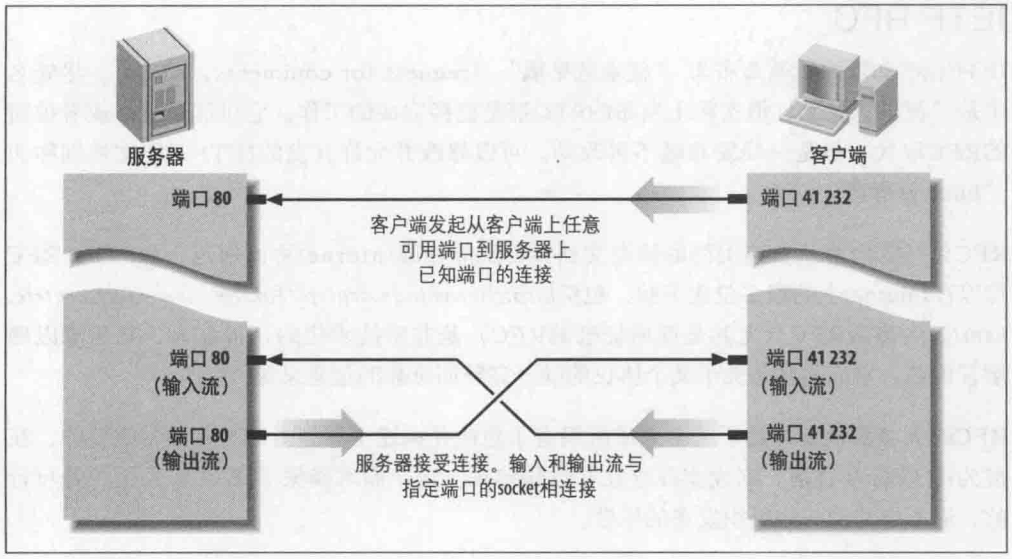


图1-5：客户/服务器连接

Java在其核心网络API中没有显式的对等通信。不过，应用程序可以很容易地通过几种方式提供对等通信，最常见的是同时作为服务器和客户端。另外，对等端可以通过中间服务器程序相互通信，这个程序将数据从一端转发到其他对等端。这样就很好地解决了两个对等端如何发现对方的问题。

# Internet标准

本书会讨论几种应用层Internet协议，最著名的是HTTP。不过，这不是一本关于这些协议的书，如果你需要关于某个协议的详细信息，权威的资源还是该协议的标准文档。

虽然世界上有很多标准组织，但关于应用层网络编程和协议的大多数标准都由下面两个组织制定，它们分别是IETF（Internet Engineering Task Force，Internet工程任务组）和W3C（World Wide Web Consortium，国际互联网协会）。IETF是不太正式的民间团体，向所有感兴趣参与的团队开放。它的标准是根据“多数人的意见和正在运行的代码”做出决定，倾向于跟踪而不是引导实现。IETF标准包括TCP/IP、MIME和SMTP。与IETF不同，W3C是厂商组织，由缴纳会费的成员公司控制，明确拒绝个人参与。在极大程度上，W3C会尽力在实现之前定义标准。W3C标准包括HTTP、HTML和XML。

## IETF RFC

IETF标准和近似标准公布为“征求意见稿”（request for comments，RFC）。尽管名字是“征求意见”，但实际上发布的RFC都是已经完成的工作。它可能会过时或者被新的RFC取代，但是一经发布就不再改动。可以修改并允许开发的IETF工作文档则称为“Internet草案”。

RFC包括的内容从普遍关注的信息文档到详细的标准Internet协议规范，如FTP。RFC可以在Internet上的很多位置下载，包括*http://www.faqs.org/rfc/*和*http://www.ietf.org/rfc.html*。大多数RFC（尤其是面向标准的RFC）是非常技术化的，很庞大，几乎难以理解。但是，它们通常是关于某个协议的唯一完整而可靠的信息来源。

RFC的大多数提案开始于一个人或组织有了想法并构建了原型。原型是非常重要的。在成为IETF标准之前，必须实际存在并运转起来。这个需求确保了IETF标准至少是可行的，而不像其他一些组织发布的标准。

表1-2列出了为本书中讨论的协议提供正式文档的RFC。

表1-2：所选的Internet RFC

| RFC | 标题 | 描述 |
|---|---|---|
| RFC 5000 | Internet官方协议标准 | 描述标准化过程和不同Internet协议的当前状态。定期由新的RFC更新 |
| RFC1122<br>RFC1123 | 主机必备条件 | 描述所有Internet主机在不同层上（数据链路层、IP层、传输层和应用层）必须支持的协议 |
| RFC791,RFC919,RFC922<br>RFC950 | 网际层协议 | IP网际层协议 |
| RFC768 | 用户数据报协议 | 不可靠的无连接传输层协议 |
| RFC792 | Internet控制消息协议（ICMP） | 一种使用原始IP数据报的网际层协议，但Java不提供支持。最常见的用法是ping和traceroute |
| RFC793 | 传输控制协议 | 可靠的面向连接的基于流的传输层协议 |
| RFC2821 | 简单邮件传输协议 | 应用层协议，一台主机可通过它将电子邮件传输到另一台主机。这个标准没有考虑电子邮件的用户界面。它涵盖了从一台计算机向另一台发送电子邮件的机制 |
| RFC822 | 电子邮件消息格式 | ASCII文本邮件消息的基本语法。设计了MIME来扩展这个格式，以支持二进制数据，同时确保传输的消息仍符合这个标准 |

表1-2：所选的Internet RFC（续）

| RFC | 标题 | 描述 |
| --- | --- | --- |
| RFC854, RFC855 | Telnet协议 | 命令行环境下的一个应用层远程登录服务，基于一个抽象的网络虚拟终端（NVT）和TCP建立 |
| RFC862 | 回显协议 | 应用层协议，回显通过TCP和UDP接收的所有数据；可以用作调试工具 |
| RFC863 | 抛弃协议 | 应用层协议，接收通过TCP和UDP传输的数据包但不向客户端发送响应，可以用作调试工具 |
| RFC864 | 字符生成器协议 | 应用层协议，向通过TCP或UDP连接的任何客户端发送不确定的ASCII字符序列，也可以用作调试工具 |
| RFC865 | 日期引用 | 应用层协议，向任何通过TCP或UDP连接的用户返回一个引用，然后关闭连接 |
| RFC867 | 日期时间（Daytime）协议 | 应用层协议，向任何通过TCP或UDP连接的客户端返回一个表示服务器当前日期和时间的人可读的ASCII串。与之相比，各种NTP和时间服务器协议都不返回便于人们阅读的数据 |
| RFC868 | 时间协议 | 应用层协议，向任何通过TCP或UDP连接的客户端返回自1900年1月1日子夜之后的秒数。这个时间作为机器可读的32位无符号整数发送。这个标准还不完整，它没有指明整数如何编码为32位，不过在实际中会使用big-endian整数 |
| RFC959 | 文件传输协议 | 提供认证的（可选）双socket应用层协议，使用TCP进行文件传输 |
| RFC977 | 网络新闻传输协议 | 应用层协议，利用它可以在机器间通过TCP传输Usenet新闻，用于新闻客户端与新闻服务器的对话以及新闻服务器之间的对话 |
| RFC1034 | | |
| RFC1035 | 域名系统 | 分布式软件集合，通过它可以把便于人记忆的主机名如www.oreilly.com转换为计算机可以理解的数字如198.112.208.11。这个RFC定义了不同主机上的域名服务器如何使用UDP相互通信 |
| RFC1112 | IP组播的主机扩展 | 网际层方法，采用这个方法的系统可以将一个数据包直接发给多台主机。这称为组播，Java对组播的支持在第13章中讨论 |

表1-2：所选的Internet RFC（续）

| RFC | 标题 | 描述 |
| --- | --- | --- |
| RFC1288 | Finger协议 | 应用层协议，请求远程网站上一个用户的信息。可能有安全风险 |
| RFC1305 | 网络时间协议（版本3） | 同步系统间时钟的一个更精确的应用层协议，会考虑到网络的延迟 |
| RFC1939 | 邮局协议，版本3 | 应用层协议，用于不定时连接的电子邮件客户端（如Eudora）通过TCP从服务器获取邮件 |
| RFC1945 | 超文本传输协议（HTTP 1.0） | Web浏览器用来通过TCP与Web服务器对话的应用层协议的1.0版本，由W3C而不是IETF开发 |
| RFC2045<br>RFC2046<br>RFC2047 | 多用途Internet邮件扩展 | 对通过Internet电子邮件和其他面向ASCII的协议传输的二进制数据和非ASCII文本进行编码的一种方法 |
| RFC2141 | 统一资源名（URN）语法 | 与URL相似，但希望使用一种持久方式而不是临时位置指示具体资源 |
| RFC2616 | 超文本传输协议（HTTP 1.1） | Web浏览器用来通过TCP与Web服务器对话的应用层协议的1.1版本 |
| RFC2373 | IPv6寻址结构 | IPv6地址的格式和含义 |
| RFC3501 | Internet消息访问协议版本4修订1 | 用于远程访问存储在服务器上的邮箱的协议，包括下载消息、删除消息和将消息移动到不同的文件夹 |
| RFC3986 | 统一资源标识符（URI）：通用语法 | 与URL相似，但表示更广泛的路径。例如，即使这本书无法通过Internet获取，ISBN编号也可以是URI |
| RFC3987 | 国际化资源标识符（IRI） | 可以包含非ASCII字符的URI |

IETF一直在幕后工作，将现有实践做法编撰成文并使之标准化。虽然它的活动完全向大众公开，但一般都非常低调。没有多少人对网络的一些神秘领域（如Internet网关消息协议（IGMP））感到兴奋。这个过程的参与者大多数是工程师和计算机科学家，包括许多来自学院和公司的人。因此，尽管对于理想的实现存在着很多争论，但经过IETF认真的努力，确实已经制定了合理的标准。

遗憾的是，尽管IETF做出了努力，但这并不表示Web（而不是Internet）标准已经制定。具体地，IETF早期在标准化HTML方面的努力就完全是失败的。Netscape和其他主要厂商拒绝参与，甚至不承认这一过程，这是一个严重的问题。尽管HTML很简单，很清

晰，足以让形形色色的各个方面感兴趣，但对于这个问题的解决仍没有多大帮助。于是，1994年10月成立了国际互联网协会，它是由厂商控制的团体，或许可以避免阻碍IETF标准化HTML和HTTP道路上的困难。

## W3C推荐

虽然W3C的标准化过程与IETF的过程相似（在邮件列表中经过仔细讨论，这样一系列工作草案最终而成为规范），但W3C与IETF是完全不同的组织。与IETF向任何人开放不同，只有公司和其他组织可以成为W3C的成员。明确规定个人不允许参加，不过有些人可以成为某些工作组的受邀专家。不过，相对于更大群体中关注的专家数量而言，这些受邀专家的数量非常之少。W3C的成员每年的会费为50 000美元（非营利组织为每年5000美元），至少承担3年义务。IETF的成员除了自愿参与之外，不需要承担任何义务，而且每年也不用提交会费。另外，虽然有许多人参与开发W3C标准，但每项标准最终由一个人批准或否决，那就是W3C的总裁Tim Berners-Lee。IETF标准则根据参与此标准的多数人的意见来批准。很明显，IETF是比W3C更民主（有人会说这是无政府主义）、更开放的组织。

尽管W3C非常偏向于付费的公司成员，但迄今为止，它在排除Web标准化中棘手的政治困难方面确实比IETF做得好。它已制定出几个HTML标准，以及其他一些标准，如HTTP、PICS、XML、CSS、MathML等。不过，W3C在促使Mozilla和Microsoft等厂商完全一致地实现其标准方面却收效甚微。

W3C有5个基本标准等级。

*注解*

注解不外乎以下两种情况：或者是由W3C成员主动提供的提案（类似于IETF的Internet草案），或者是W3C人员或相关团队的随意想法，实际上并不描述一个完整的提案（类似于IETF信息性RFC）。注解不一定会促使形成工作组或W3C推荐。

*工作草案*

工作草案是工作组中某些成员（不一定是全部成员）当前的想法。它应当最终成为提议推荐，但到那时可能会有大量的改动。

*候选推荐*

候选推荐表示工作组在所有主要问题上已经得到多数人的同意，在等待第三方的意见和实现。如果实现没有遇到障碍，这个规范就可以提升为候选推荐。

*提议推荐*

提议推荐通常是完整的，除了细微的编辑修改，不太可能有太大的变化。提议推荐的主要目的是解决规范文档中的bug，而不是修正文档描述的底层技术的bug。

*推荐*

推荐是W3C标准的最高等级。不过，W3C非常谨慎，并不称之为“标准”，这是担心与反托拉斯法案相抵触。W3C将推荐描述为“代表W3C中大多数人的意见及得到主席批准盖章的工作。W3C认为由推荐说明的想法或技术适合于广泛部署和升级W3C的任务。”

## PR（公关）标准

寻求一些言论自由或者股票价格可能临时上涨的公司有时会滥用W3C和IETF标准过程。IETF会接受任何人的呈递书，W3C会接受任何W3C成员的呈递书。IETF称这些呈递书为“Internet草案”，在删除之前会公布6个月。W3C称这样的呈递书为“承认的呈递书”，会不确定地进行公布。但是，实际上这两个组织除了承认接收到这些文档之外并没有其他承诺。特别是他们不会许诺成立一个工作组或者开始标准化过程。虽然如此，新闻稿却总是将此类文档的呈递歪曲为比实际情况重要得多。PR（公关，public relation）代表通常能至少欺骗一些不了解标准化过程内部细节的愚蠢记者。不过，你应当能看出这些把戏的真实面目。

# 第2章

# 流

网络程序所做的很大一部分工作都是简单的输入和输出：将数据字节从一个系统移动到另一个系统。字节就是字节。在很大程度上讲，读取服务器发送给你的数据与读取文件并没什么不同。向客户端发送文本与写文件也没有什么不同。但是，Java中输入和输出（I/O）的组织与其他大多数语言（如Fortran、C和C++）都不一样。因此，这里要用几页来总结一下Java独特的I/O方法。

Java的I/O建立于流（stream）之上。输入流读取数据；输出流写入数据。不同的流类，如`java.io.FileInputStream`和`sun.net.TelnetOutputStream`会读/写某个特定的数据源。但是，所有输出流都有相同的基本方法来写入数据，所有输入流也使用相同的基本方法来读取数据。创建一个流之后，读/写时通常可以忽略读/写的具体细节。

过滤器（filter）流可以串链到输入流或输出流上。读/写数据时，过滤器可以修改数据（例如，通过加密或压缩），或者只是提供额外的方法，将读/写的数据转换为其他格式。例如，`java.io.DataOutputStream`类就提供了一个方法，可以将`int`转换为4字节，并把这些字节写入底层的输出流。

阅读器（reader）和书写器（writer）可以串链到输入流和输出流上，允许程序读/写文本（即字符）而不是字节。只要正确地使用，阅读器和书写器可以处理很多字符编码，包括多字节字符集，如SJIS和UTF-8。

流是同步的。也就是说，当程序（确切地讲是线程）请求一个流读/写一段数据时，在做任何其他操作前，它要等待所读/写的数据。Java还支持使用通道和缓冲区的非阻塞I/O。非阻塞I/O稍有些复杂，但在某些高吞吐量的应用程序中（如Web服务器），非阻塞I/O要快得多。通常情况下，基本流模型就是实现客户端所需要和应当使用的全部内

容。由于通道和缓冲区依赖于流，下面将首先介绍流和客户端，后面在第11章中还会讨论服务器使用的非阻塞I/O。

# 输出流

Java的基本输出流类是java.io.OutputStream：

```
public abstract class OutputStream
```

这个类提供了写入数据所需的基本方法。这些方法包括：

```
public abstract void write(int b) throws IOException
public void write(byte[] data) throws IOException
public void write(byte[] data, int offset, int length)
    throws IOException
public void flush() throws IOException
public void close() throws IOException
```

OutputStream的子类使用这些方法向某种特定介质写入数据。例如，FileOutputStream使用这些方法将数据写入文件。TelnetOutputStream使用这些方法将数据写入网络连接。ByteArrayOutputStream使用这些方法将数据写入可扩展的字节数组。但不管写入哪种介质，大多都会使用同样的这5个方法。有时甚至可能不知道所写入的流具体是何种类型。例如，在Java类库文档中找不到TelnetOutputStream。它被有意地隐藏在sun包中。java.net中很多类的很多方法都会返回TelnetOutputStream，如java.net.Socket的getOutputStream()方法。但是，这些方法声明为只返回OutputStream，而不是更特定的子类TelnetOutputStream。这正是多态的威力。如果你知道如何使用这些超类，也就知道如何使用所有这些子类。

OutputStream的基本方法是write(int b)。这个方法接受一个0到255之间的整数作为参数，将对应的字节写入到输出流中。这个方法声明为抽象方法，因为各个子类需要修改这个方法来处理特定的介质。例如，ByteArrayOutputStream可以用纯Java代码实现这个方法，将字节复制到数组中。与此不同，FileOutputStream则需要使用原生代码，这些代码了解如何将数据写入到主机平台的文件中。

注意，虽然这个方法接受一个int作为参数，但它实际上会写入一个无符号字节。Java没有无符号字节数据类型，所以这里要使用int来代替。无符号字节和有符号字节之间唯一的真正区别在于解释。它们都由8个二进制位组成，当使用write(int b)将int写入一个网络连接时，线缆上只会放8个二进制位。如果将一个超出0～255的int传入write(int b)，将写入这个数的最低字节，其他3字节将被忽略（这正是将int强制转换为byte的结果）。

**提示：** 不过，在极少数情况下，你可能会看到一些有问题的第三方类，在写超出0～255的值时，它们的做法有所不同，比如会抛出IllegalArgumentException异常或者总是写入255，所以尽可能要避免写超出0～255的int。

例如，字符生成器协议定义了一个发出ASCII文本的服务器。这个协议最流行的变体是发送72个字符的文本行，其中包含可显示的ASCII字符。（可显示的ASCII字符是33到126之间的字符，不包含各种空白符和控制字符）。第一行按顺序包含字符33到字符104。第二行包含字符34到字符105。第三行包含字符35到字符106。一直到第29行包含字符55到字符126。至此，字符将回绕，这样第30行包含字符56到字符126，加上字符33。各行用回车（ASCII 13）和换行（ASCII 10）结束。输出如下：

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefgh
"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghi
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmn
```

由于ASCII是一个7位字符集，所以每个字符都作为单字节发送。因此，这个协议可以直接使用基本write()方法实现，如以下代码段所示：

```
public static void generateCharacters(OutputStream out)
    throws IOException {

  int firstPrintableCharacter = 33;
  int numberOfPrintableCharacters = 94;
  int numberOfCharactersPerLine = 72;

  int start = firstPrintableCharacter;
  while (true) { /*无限循环*/
    for (int i = start; i < start + numberOfCharactersPerLine; i++) {
      out.write((
        (i - firstPrintableCharacter) % numberOfPrintableCharacters)
        + firstPrintableCharacter);
    }
    out.write('\r'); //回车
    out.write('\n'); //换行
    start = ((start + 1) - firstPrintableCharacter)
        % numberOfPrintableCharacters + firstPrintableCharacter;
  }
}
```

这里将一个OutputStream通过out参数传入generateCharacters()方法。一次向out写入1字节。这些字节作为33到126之间循环序列中的整数给出。这里的大部分运算都是让循环在这个范围内回绕。在写入每个72字符块之后，就向输出流写入一个回车和一个换行。然后计算下一个起始字符，重复这个循环。整个方法声明为抛出IOException异

常。这一点很重要，因为字符生成器服务器只在客户端关闭连接时才会终止。Java代码会把它看作是一个IOException异常。

一次写入1字节通常效率不高。例如，流出以太网卡的每个TCP分片包含至少40字节的开销用于路由和纠错。如果每字节都单独发送，那么与你预想的数据量相比，实际填入到网络中的数据可能会高出41倍以上！如果增加主机网络层协议的开销，情况可能更糟糕。因此，大多数TCP/IP实现都会在某种程度上缓存数据。也就是说，它们在内存中积累数据字节，只有积累到一定量的数据后，或者经过了一定的时间后，才将所积累的数据发送到最终目的地。不过，如果有多字节要发送，则一次全部发送不失为一个好主意。使用write(byte[] data)或write(byte[] data, int offset, int length)通常比一次写入data数组中的1字节要快得多。例如，下面是generateCharacters()方法的一个实现，它将整行打包在1字节数组中，一次发送一行：

```
public static void generateCharacters(OutputStream out)
    throws IOException {

  int firstPrintableCharacter = 33;
  int numberOfPrintableCharacters = 94;
  int numberOfCharactersPerLine = 72;
  int start = firstPrintableCharacter;
  byte[] line = new byte[numberOfCharactersPerLine + 2];
  // +2对应回车和换行

  while (true) { /*无限循环*/
    for (int i = start; i < start + numberOfCharactersPerLine; i++) {
      line[i - start] = (byte) ((i - firstPrintableCharacter)
        % numberOfPrintableCharacters + firstPrintableCharacter);
    }
    line[72] = (byte) '\r'; // 回车
    line[73] = (byte) '\n'; // 换行
    out.write(line);
    start = ((start + 1) - firstPrintableCharacter)
        % numberOfPrintableCharacters + firstPrintableCharacter;
  }
}
```

计算何时写哪些字节的算法与前面的实现中是一样的。重要的区别在于这些字节在写入网络之前先打包到1字节数组中。还要注意计算的int结果在存储到数组前要转换为字节。这在前面的实现中是不必要的，因为单字节write()方法就声明为接受一个int作为参数。

与在网络硬件中缓存一样，流还可以在软件中得到缓冲，即直接用Java代码缓存。一般说来，这可以通过把BufferedOutputStream或BufferedWriter串链到底层流上来实现，稍后将探讨这种技术。因此，在写入数据完成后，刷新（flush）输出流非常重要。例如，假设已经向使用HTTP Keep-Alive的HTTP 1.1服务器写入了300字节的请求，通常你

会等待响应，然后再发送更多的数据。不过，如果输出流有一个1024字节的缓冲区，那么这个流在发送缓冲区中的数据之前会等待更多的数据到达。在服务器响应到达之前不会向流写入更多数据，但是响应永远也不会到来，因为请求还没有发送！图2-1显示了这种两难境地。flush()方法可以强迫缓冲的流发送数据，即使缓冲区还没有满，以此来打破这种死锁状态。

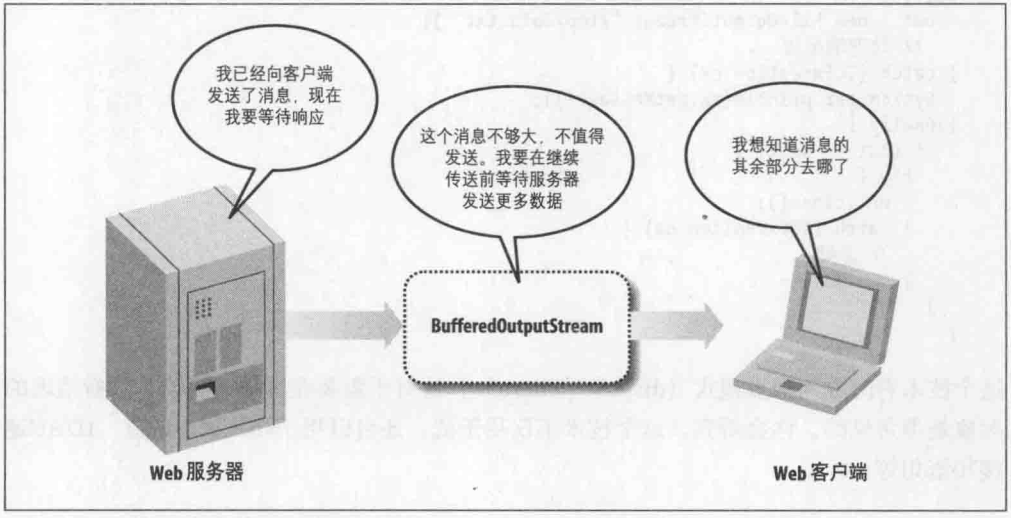


图2-1：如果不刷新输出流，数据可能丢失

不管你是否认为有必要，刷新输出流都很重要。取决于你如何控制流的引用，你可能知道流是否缓冲，也可能不知道（例如，不论你是否希望如此，System.out都会缓冲）。如果刷新输出对于某个特定的流来说没有必要，那么它也只是个低成本的操作。不过，如果有必要刷新输出，就必须完成这个操作。需要刷新输出时如果未能做到，那么会导致不可预知、不可重现的程序挂起，如果你未能首先清楚地知道问题出在哪里，那么诊断起来将会非常困难。相应地，应当在关闭流之前立即刷新输出所有流。否则，关闭流时留在缓冲区中的数据可能会丢失。

最后，当结束一个流的操作时，要通过调用它的close()方法将其关闭。这会释放与这个流关联的所有资源，如文件句柄或端口。如果流来自一个网络连接，那么关闭这个流也会终止这个连接。一旦输出流关闭，继续写入时就会抛出IOException异常。不过，有些流仍允许对这个对象做一些处理。例如，关闭的ByteArrayOutputStream仍然可以转换为实际的字节数组，关闭的DigestOutputStream仍然可以返回其摘要。

在一个长时间运行的程序中，如果未能关闭一个流，则可能会泄漏文件句柄、网络端口和其他资源。因此，在Java 6和更早版本中，明智的做法是在一个finally块中关闭流。

为了得到正确的变量作用域，必须在try块之外声明流变量，但必须在try块内完成初始化。另外，为了避免NullPointerException异常，在关闭流之前需要检查流变量是否为null。最后，通常都希望忽略关闭流时出现的异常，或者最多只是把这些异常记入日志。例如：

```
OutputStream out = null;
try {
  out = new FileOutputStream( "/tmp/data.txt" );
   // 处理输出流...
} catch (IOException ex) {
  System.err.println(ex.getMessage());
} finally {
  if (out != null) {
    try {
      out.close();
    } catch (IOException ex) {
      // 忽略
    }
  }
}
```

这个技术有时称为释放模式（dispose pattern），这对于需要在垃圾回收前先进行清理的对象是很常见的。你会看到，这个技术不仅用于流，还可以用于socket、通道、JDBC连接和语句等。

Java 7引入了“带资源的try”构造（try with resources），可以更简洁地完成这个清理。不需要在try块之外声明流变量，完全可以在try块的一个参数表中声明。例如，前面的代码段现在就变得简单多了：

```
try (OutputStream out = new FileOutputStream( "/tmp/data.txt" )) {
  // 处理输出流...
} catch (IOException ex) {
  System.err.println(ex.getMessage());
}
```

现在不再需要Finally子句。Java会对try块参数表中声明的所有AutoCloseable对象自动调用close()。

---

**提示：** 只要对象实现了Closeable接口，都可以使用“带资源的try”构造，这包括几乎所有需要释放的对象。到目前为止，JavaMail Transport对象是我见过的唯一的例外。这些对象还需要显式地释放。

---

# 输入流

Java的基本输入类是java.io.InputStream：

```
public abstract class InputStream
```

这个类提供了将数据读取为原始字节所需的基本方法。这些方法包括：

```
public abstract int read() throws IOException
public int read(byte[] input) throws IOException
public int read(byte[] input, int offset, int length) throws IOException
public long skip(long n) throws IOException
public int available() throws IOException
public void close() throws IOException
```

InputStream的具体子类使用这些方法从某种特定介质中读取数据。例如，FileInputStream从文件中读取数据。TelnetInputStream从网络连接中读取数据。ByteArrayInputStream从字节数组中读取数据。但无论读取哪种数据源，主要只使用以上这6个方法。有时你不知道正在读取的流具体是何种类型。例如，TelnetInputStream类隐藏在sun.net包中，没有提供相关文档。java.net包中的很多方法都会返回这个类的实例（例如java.net.URL的openStream()方法）。不过，这些方法声明为只返回InputStream，而不是更特定的子类TelnetInputStream。这又是多态在起作用。子类的实例可以透明地作为其超类的实例来使用。并不需要子类更特定的知识。

InputStream的基本方法是没有参数的read()方法。这个方法从输入流的源中读取1字节数据，作为一个0到255的int返回。流的结束通过返回-1来表示。read()方法会等待并阻塞其后任何代码的执行，直到有1字节的数据可供读取。输入和输出可能很慢，所以如果程序在做其他重要的工作，要尽量将I/O放在单独的线程中。

read()方法声明为抽象方法，因为各个子类需要修改这个方法来处理特定的介质。例如，ByteArrayInputStream会用纯Java代码实现这个方法，从其数组复制字节。不过，TelnetInputStream需要使用一个原生库，它知道如何从主机平台的网络接口读取数据。

下面的代码段从InputStream in中读取10字节，存储在byte数组input中。不过，如果检测到流结束，循环就会提前终止：

```
byte[] input = new byte[10];
for (int i = 0; i < input.length; i++) {
  int b = in.read();
  if (b == -1) break;
  input[i] = (byte) b;
}
```

虽然read()只读取1字节，但它会返回一个int。这样在把结果存储到字节数组之前就必须进行类型转换。当然，这会产生一个-128到127之间的有符号字节，而不是read()方法返回的0到255之间的一个无符号字节。不过，只要你清楚在做什么，这就不是大问题。你可以如下将一个有符号字节转换为无符号字节：

```
int i = b >= 0 ? b : 256 + b;
```

与一次写入1字节的数据一样，一次读取1字节的效率也不高。因此，有两个重载的read()方法，可以用从流中读取的多字节的数据填充一个指定的数组：read(byte[] input)和read(byte[] input, int offset, int length)。第一个方法尝试填充指定的数组input。第二个方法尝试填充指定的input中从offset开始连续length字节的子数组。

注意我说这些方法是在尝试填充数组，但不是一定会成功。尝试可能会以很多不同的方式失败。例如，你可能听说过，当你的程序正在通过DSL从远程Web服务器读取数据时，由于电话公司中心办公室的交换机存在bug，这会断开你与其他地方数百个邻居的连接。这会导致一个IOException异常。但更常见的是，读尝试可能不会完全失败，但也不会完全成功。可能读取到一些请求的字节，但未能全部读取到。例如，你可能尝试从一个网络连接中读取1024字节，现在实际上只有512字节到达，其他的仍在传输中。尽管它们最终会到达，但此时却不可用。考虑到这一点，读取多字节的方法会返回实际读取的字节数。例如，考虑下面的代码段：

```
byte[] input = new byte[1024];
int bytesRead = in.read(input);
```

它尝试从InputStream in向数组input中读入1024字节。不过，如果只有512字节可用，就只会读取这么多，bytesRead将会设置为512。为保证你希望的所有数据都真正读取到，要把读取方法放在循环中，这样会重复读取，直到数组填满为止。例如：

```
int bytesRead = 0;
int bytesToRead = 1024;
byte[] input = new byte[bytesToRead];
while (bytesRead < bytesToRead) {
  bytesRead += in.read(input, bytesRead, bytesToRead - bytesRead);
}
```

这项技术对于网络流尤为重要。一般来讲如果一个文件完全可用，那么文件的所有字节也都可用。不过，由于网络要比CPU慢得多，所以程序很容易在所有数据到达前清空网络缓冲区。事实上，如果这两个方法尝试读取暂时为空但打开的网络缓冲区，它通常会返回0，表示没有数据可用，但流还没有关闭。这往往比单字节的read()方法要好，在这种情况下单字节方法会阻塞正在运行的线程。

所有3个read()方法都用返回－1表示流的结束。如果流已结束，而又没有读取的数据，多字节read()方法会返回这些数据，直到缓冲区清空。其后任何一个read()方法调用会返回－1。－1永远不会放进数组中。数组中只包含实际的数据。前面的代码段中存在一个bug，因为它没有考虑所有1024字节可能永远不会到达的情况（这与前面所说的情况不同，那只是当时不可用，但以后所有字节总会到达）。要修复这个bug，需要先测试read()的返回值，然后再增加到bytesRead中。例如：

```
int bytesRead = 0;
int bytesToRead = 1024;
byte[] input = new byte[bytesToRead];
while (bytesRead < bytesToRead) {
  int result = in.read(input, bytesRead, bytesToRead - bytesRead);
  if (result == -1) break; // 流结束
  bytesRead += result;
}
```

如果不想等待所需的全部字节都立即可用，可以使用available()方法来确定不阻塞的情况下有多少字节可以读取。它会返回可以读取的最少字节数。事实上还能读取更多字节，但至少可以读取available()建议的字节数。例如：

```
int bytesAvailable = in.available();
byte[] input = new byte[bytesAvailable];
int bytesRead = in.read(input, 0, bytesAvailable);
//立即继续执行程序的其他部分...
```

在这种情况下，可以认为bytesRead与bytesAvailable相等。不过，不能期望bytesRead大于0，有可能没有可用的字节。在流的最后，available()会返回0。一般来说，read(byte[] input, int offset, int length)在流结束时返回-1；但如果length是0，那么它不会注意流的结束，而是返回0。

在少数情况下，你可能希望跳过数据不进行读取。skip()方法会完成这项任务。与读取文件相比，在网络连接中它的用处不大。网络连接是顺序的，一般情况下很慢，所以与跳过数据（不读取）相比，读取数据并不会多耗费太长时间。文件是随机访问的，所以要跳过数据，可以简单地实现为重新指定文件指针位置，而不需要处理要跳过的各字节。

与输出流一样，一旦结束对输入流的操作，应当调用它的close()方法将其关闭。这会释放与这个流关联的所有资源，如句柄或端口。一旦输入流已关闭，进一步读取这个流会抛出IOException异常。不过，有些流可能仍然允许处理这个对象。例如，你通常会在读取了数据并关闭流之后才会从java.security.DigestInputStream获取消息摘要。

## 标记和重置

InputStream类还有3个不太常用的方法，允许程序备份和重新读取已经读取的数据。这些方法是：

```
public void mark(int readAheadLimit)
public void reset() throws IOException
public boolean markSupported()
```

为了重新读取数据，要用mark()方法标记流的当前位置。在以后某个时刻，可以用

reset()方法把流重置到之前标记的位置。接下来的读取操作会返回从标记位置开始的数据。不过，不能随心所欲地向前重置任意远的位置。从标记处读取和重置的字节数由mark()的readAheadLimit参数确定。如果试图重置得太远，就会抛出IOException异常。此外，一个流在任何时刻都只能有一个标记。标记第二个位置会清除第一个标记。

标记和重置通常通过将标记位置之后的所有字节存储在一个内部缓冲区中来实现。不过，不是所有输入流都支持这一点。在尝试使用标记和重置之前，要检查markSupported()方法是否返回true。如果返回true，那么这个流确实支持标记和重置。否则，mark()会什么都不做，而reset()将抛出一个IOException异常。

---

提示：在我看来，这是一个非常差的设计。实际上，不支持标记和重置的流比提供支持的更多。如果向抽象的超类附加一个功能，但这个功能对很多（甚至可能是大多数）子类都不可用，这就是一个很不好的想法。把这三个方法放在一个单独的接口中，由提供这个功能的类实现这个接口，这样做可能会更好。这个方法的缺点是不能在未知类型的任意输入流上调用这些方法，但实际上也不会这样做，因为并不是所有流都支持标记和重置。可以提供一个方法（如markSupported()）在运行时进行检查，这是针对该问题的一个更传统的非面向对象的解决方案。面向对象的方法是通过接口和类将其嵌入在类型系统中，这样就可以在编译时进行检查。

---

java.io中仅有的两个始终支持标记的输入流类是BufferedInputStream和ByteArrayInputStream。而其他输入流（如TelnetInputStream）如果先串链到缓冲的输入流时才支持标记。

## 过滤器流

InputStream和OutputStream是相当原始的类。它们可以单个或成组地读/写字节，但仅此而已。要确定这些字节的含义（比如，它们是整数还是IEEE 754浮点数或是Unicode文本），这完全由程序员和代码来完成。不过，有一些极为常见的数据格式，如果在类库中提供这些数据格式的固定实现，会很有好处。例如，许多作为网络协议一部分传递的整数是32位big-endian整数。许多通过Web发送的文本是7位ASCII、8位Latin-1或多字节UTF-8。许多由FTP传输的文件存储为zip格式。Java提供了很多过滤器类，可以附加到原始流中，在原始字节和各种格式之间来回转换。

过滤器有两个版本：过滤器流以及阅读器和书写器。过滤器流仍然主要将原始数据作为字节处理，例如通过压缩数据或解释为二进制数字。阅读器和书写器处理多种编码文本的特殊情况，如UTF-8和ISO 8859-1。

过滤器以链的形式进行组织，如图2-2所示。链中的每个环节都接收前一个过滤器或流的数据，并把数据传递给链中的下一个环节。在这个示例中，从本地网络接口接收到一个压缩的加密文本文件，在这里本地代码将这个文件表示为TelnetInputStream（TelnetInputStream没有相关文档提供说明）。通过一个BufferedInputStream缓冲这个数据来加速整个过程。由一个CipherInputStream将数据解密。再由一个GZIPInputStream解压解密后的数据。一个InputStreamReader将解压后的数据转换为Unicode文本。最后，文本由应用程序读取并处理。

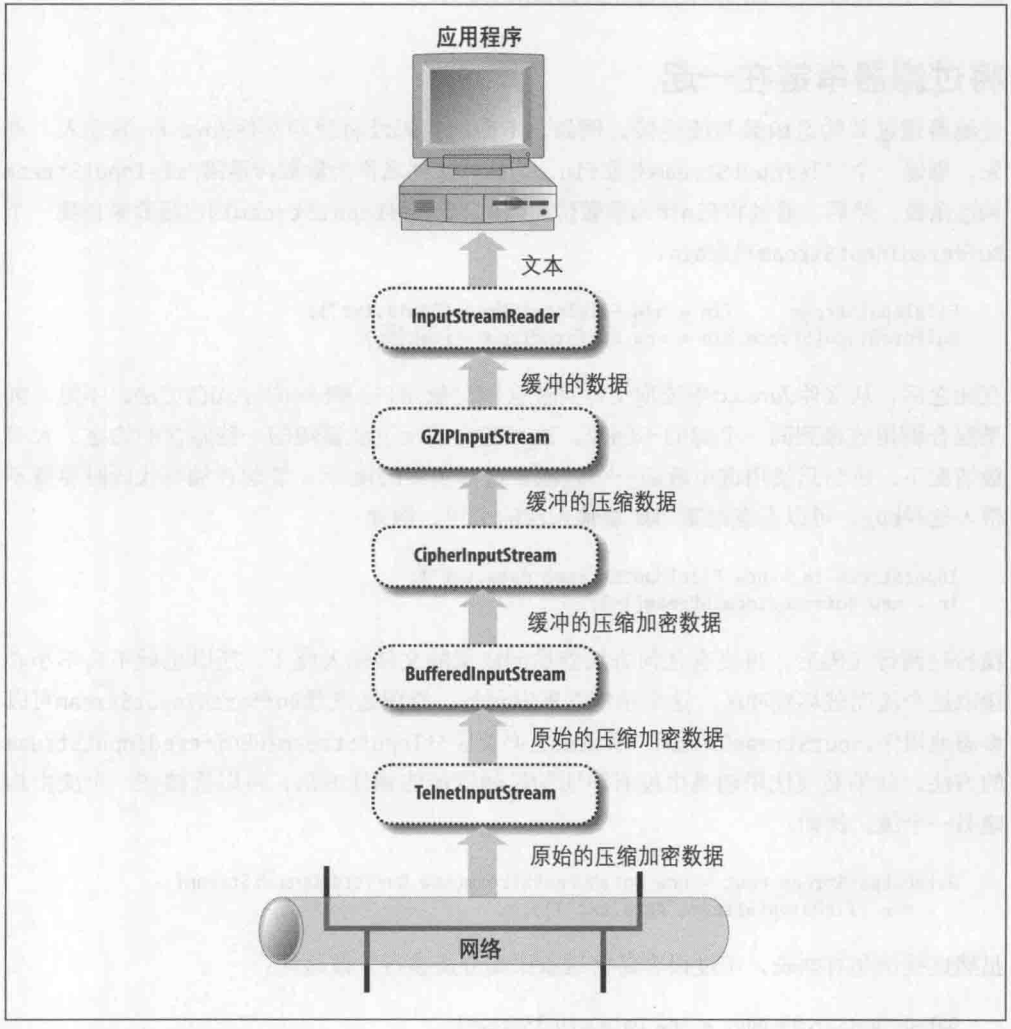


图2-2：过滤器链中的数据流

每个过滤器输出流都有与java.io.OutputStream相同的write()、close()和flush()方法。每个过滤器输入流都有与java.io.InputStream相同的read()、close()和available()方法。有些情况下，如BufferedInputStream和BufferedOutputStream，过滤器可能只有这些方法。过滤纯粹是内部操作，不提供任何新的公共接口。不过，在大多数情况下，过滤器流会增加一些公共方法提供额外的作用。有时除了平常的read()和write()方法之外，还需要使用这些方法，如PushbackInputStream的unread()方法。另外一些情况下，它们几乎完全代替了最初的接口。例如，PrintStream的write()方法就很少使用，而会使用它的print()和println()方法。

## 将过滤器串链在一起

过滤器通过其构造函数与流连接。例如，下面的代码段将缓冲文件*data.txt*的输入。首先，创建一个FileInputStream对象fin，为此将文件名作为参数传递给FileInputStream构造函数。然后，通过将fin作为参数传递给BufferedInputStream构造函数来创建一个BufferedInputStream对象bin：

```
FileInputStream     fin = new FileInputStream("data.txt");
BufferedInputStream bin = new BufferedInputStream(fin);
```

在此之后，从文件*data.txt*中读取文件可能会同时使用fin和bin的read()方法。不过，如果混合调用连接到同一个源的不同流，这可能会违反过滤器流的一些隐含的约定。大多数情况下，应当只使用链中最后一个过滤器进行实际的读/写。要想在编写代码时尽量不带入这种bug，可以有意地重写底层输入流的引用。例如：

```
InputStream in = new FileInputStream("data.txt");
in = new BufferedInputStream(in);
```

执行这两行代码后，再没有任何方法能访问底层的文件输入流了，所以也就不会不小心读取这个流而破坏缓冲区。这个示例之所以可行，原因是既然BufferedInputStream可以多态地用作InputStream的实例，所以没有必要区分InputStream和BufferedInputStream的方法。如果必须使用超类中没有声明的过滤器流的其他方法，可以直接在一个流中构建另一个流。例如：

```
DataOutputStream dout = new DataOutputStream(new BufferedOutputStream(
    new FileOutputStream("data.txt")));
```

虽然这些语句有些长，不过很容易将这条语句分成多行，像这样：

```
DataOutputStream dout = new DataOutputStream(
                        new BufferedOutputStream(
                        new FileOutputStream("data.txt")
                       )
```

```
);
```

这种连接是永久的。过滤器无法与流断开连接。

有时可能会需要使用链中多个过滤器的方法。例如，在读取Unicode文本文件时，可能希望读取前3字节中的字节顺序标记，来判断文件是用big-endian UCS-2、little-endian UCS-2，还是用UTF-8编码的，然后选择与编码匹配的Reader（阅读器）过滤器。或者当连接Web服务器时，可能希望读取服务器发送的首部，找到Content-encoding（内容编码），然后用这个内容编码方式选取正确的Reader（阅读器）过滤器来读取响应主体。或者可能希望通过网络连接使用DataOutputStream发送浮点数，然后从DataOutputStream所链接的DigestOutputStream中获取一个MessageDigest。在所有这些情况下，都需要保存和使用各个底层流的引用。不过，除了链中最后一个过滤器之外，无论如何你都不应该从其他的过滤器读取数据，或向其写入任何内容。

## 缓冲流

BufferedOutputStream类将写入的数据存储在缓冲区中（一个名为buf的保护字节数组字段），直到缓冲区满或刷新输出流。然后它将数据一次全部写入底层输出流。如果一次写入多字节，这与多次写入少量字节（但字节加起来是一样的）相比，前者往往要快得多。对于网络连接尤其是这样，因为每个TCP片或UDP包都有一定数量的开销，一般大约为40字节。这意味着，如果一次发送1字节，那么发送1K数据实际上需要通过线缆发送40K，而一次全部发送只需要发送1K多一点点数据。大多数网卡和TCP实现自身都提供了一定程度的缓冲，所以实际的数量不会那么夸张。尽管如此，缓冲网络输出通常会带来巨大的性能提升。

BufferedInputStream类也有一个作为缓冲区的保护字节数组，名为buf。当调用某个流的read()方法时，它首先尝试从缓冲区获得请求的数据。只有当缓冲区没有数据时，流才从底层的源中读取数据。这时，它会从源中读取尽可能多的数据存入缓冲区，而不管是否马上需要所有这些数据。不会立即用到的数据可以在以后调用read()时读取。当从本地磁盘中读取文件时，从底层流中读取几百字节的数据与读取1字节数据几乎一样快。因此，缓冲可以显著提升性能。对于网络连接，这种效果则不甚明显，在这里瓶颈往往是网络传送数据的速度，而不是网络接口向程序传送数据的速度或程序运行的速度。尽管如此，缓冲输入没有什么坏处，随着网络的速度加快会变得更为重要。

BufferedInputStream有两个构造函数，BufferedOutputStream也一样：

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int bufferSize)
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStream out, int bufferSize)
```

第一个参数是底层流，可以从中读取未缓冲的数据，或者向其写入缓冲的数据。如果给出第二个参数，它会指定缓冲区中的字节数。否则，输入流的缓冲区大小设置为2048字节，输出流的缓冲区大小设置为512字节。缓冲区的理想大小取决于所缓冲的流是何种类型。对于网络连接，你会希望比一般的包大小更大一些。不过，这很难预测，根据本地网络连接和协议的不同也有所区别。更快、更大带宽的网络倾向于使用更大的包，不过TCP片通常不会大于1K字节。

BufferedInputStream没有声明自己的任何新方法。它只覆盖了InputStream的方法。它支持标记和重置。两个多字节read()方法尝试根据需要多次从底层输入流中读取数据，从而完全填充指定的数组或子数组。只有当数组或子数组完全填满、到达流的末尾或底层流阻塞而无法进一步读取时，这两个read()方法才返回。大多数输入流都不这样做。它们在返回前只从底层流或数据源中读取一次。

BufferedOutputStream也没有声明自己的任何新方法。调用它的方法与调用任何输出流的方法是一样的。区别在于，每次写入会把数据放在缓冲区中，而不是直接放入底层的输出流。因此，需要发送数据时应当刷新输出流，这一点非常重要。

## PrintStream

PrintStream类是大多数程序员都会遇到的第一个过滤器输出流，因为System.out就是一个PrintStream。不过，还可以使用下面两个构造函数将其他输出流串链到打印流：

```
public PrintStream(OutputStream out)
public PrintStream(OutputStream out, boolean autoFlush)
```

默认情况下，打印流应当显式刷新输出。不过，如果autoFlush参数为true，那么每次写入1字节数组或换行，或者调用println()方法时，都会刷新输出流。

除了平常的write()、flush()和close()方法，PrintStream还有9个重载的print()方法和10个重载的println()方法：

```
public void print(boolean b)
public void print(char c)
public void print(int i)
public void print(long l)
public void print(float f)
public void print(double d)
public void print(char[] text)
public void print(String s)
public void print(Object o)
public void println()
public void println(boolean b)
public void println(char c)
public void println(int i)
```

```
public void println(long l)
public void println(float f)
public void println(double d)
public void println(char[] text)
public void println(String s)
public void println(Object o)
```

每个print()方法都将其参数以可预见的方式转换为一个字符串，再用默认的编码方式把字符串写入底层输出流。println()方法也完成相同的操作，但会在所写的行末尾追加一个与平台有关的行分隔符。在UNIX（包括Mac OS X）下是换行符（\n），在Mac OS 9下是回车符（\r），在Windows下是回车/换行对（\r\n）。

---

**警告：** PrintStream是有害的，网络程序员应当像躲避瘟疫一样避开它！

---

第一个问题是println()的输出是与平台有关的。取决于运行代码的机器，各行有时用换行符分隔，有时则用回车符或者回车/换行对来分隔。写入控制台时这不会产生问题，但对于编写必须遵循明确协议的网络客户端和服务器而言，这却是个灾难。大多数网络协议（如HTTP和Gnutela）明确指定行应当以回车/换行对结束。使用println()写出的程序很有可能可以在Windows上正常工作，但在UNIX和Mac上无法工作。虽然许多服务器和客户端能够“宽容”地接受而且能处理不正确的行结束符，但偶尔也有例外。

第二个问题是PrintStream假定使用所在平台的默认编码方式。不过，这种编码方式可能不是服务器或客户端所期望的。例如，一个接收XML文件的Web浏览器希望文件以UTF-8或UTF16方式编码，除非服务器另行要求。不过，一个使用PrintStream的Web服务器可能会从一个美国本地化环境的Windows系统发送CP1252编码的文件，或者从日本本地化环境的系统发送SJIS编码的文件，而不管客户端是否期望或理解这些编码方式。PrintStream不提供任何改变默认编码的机制。这个问题可以通过使用相关的PrintWriter类来修补。但是其他问题依旧。

第三个问题是PrintStream吞掉了所有异常。这使得PrintStream很适合作为教科书程序，如HelloWorld，因为要讲授简单的控制台输出，不用让学生先去学习异常处理和所有相关的知识。不过，网络连接不如控制台那么可靠。连接经常会由于网络拥塞、电话公司的错误、远程系统崩溃，以及很多其他原因而断开。网络程序必须准备处理数据流中意料之外的中断。要做到这一点，就需要使用异常处理。不过，PrintStream捕获了底层输出流抛出的所有异常。注意PrintStream中5个标准OutputStream方法的声明没有平常的throws IOException声明：

```
public abstract void write(int b)
public void write(byte[] data)
public void write(byte[] data, int offset, int length)
```

```
public void flush()
public void close()
```

实际上，PrintStream要依靠一个过时的不充分的错误标志。如果底层流抛出一个异常，就会设置这个内部错误标志。要由程序员使用checkError()方法来检查这个标志的值：

```
public boolean checkError()
```

要对PrintStream完成任何错误检查，代码必须显式地检查每一个调用。此外，一旦出现错误，就没有办法重置这个标志再进行进一步的错误检测。也没有关于这个错误的更多信息。简而言之，PrintStream提供的错误通知对于不可靠的网络连接来说还远远不够。

## 数据流

DataInputStream和DataOutputStream类提供了一些方法，可以用二进制格式读/写Java的基本数据类型和字符串。所用的二进制格式主要用于在两个不同的Java程序之间交换数据（可能通过网络连接、数据文件、管道或者其他中间介质）。输出流写入什么数据，输入流就能读取什么数据。不过，这碰巧与大多数交换二进制数的Internet协议所用的格式相同。例如，时间协议使用32位big-endian整数，类似于Java的int数据类型。负载受控的网络元素服务使用32位IEEE 754浮点数，类似于Java的float数据类型（这是有关联的，而不只是巧合。Java和大多数网络协议都是由UNIX程序员设计的，因此都会倾向于使用大多数UNIX系统中的常见格式）。然而，这并不适用于所有网络协议，所以请检查你使用的协议的具体细节。例如，网络时间协议（NTP）会把时间表示为64位无符号定点数，前32位是整数部分，后32位是小数部分。这与所有常见编程语言中的基本数据类型都不相同，不过处理起来相当简单，至少对于NTP必须使用这种格式。

DataOutputStream类提供了下面11种方法，可以写入特定的Java数据类型：

```
public final void writeBoolean(boolean b) throws IOException
public final void writeByte(int b) throws IOException
public final void writeShort(int s) throws IOException
public final void writeChar(int c) throws IOException
public final void writeInt(int i) throws IOException
public final void writeLong(long l) throws IOException
public final void writeFloat(float f) throws IOException
public final void writeDouble(double d) throws IOException
public final void writeChars(String s) throws IOException
public final void writeBytes(String s) throws IOException
public final void writeUTF(String s) throws IOException
```

所有数据都以big-endian格式写入。整数用尽可能少的字节写为2的补码。因此，byte会写为1字节， short写为2字节， int写为4字节，long写为8字节。浮点数和双精度数分

别写为4字节和8字节的IEEE 754格式。布尔数写为1字节，0表示false，1表示true。字符写为两个无符号字节。

最后三个方法有些棘手。writeChars()方法只是对String参数迭代（循环）处理，将各个字符按顺序写为一个2字节的big-endian Unicode字符（确切地讲是UTF-16码点）。writeBytes()方法迭代处理String参数，但只写入每个字符的低字节。因此，如果字符串中包含有Latin-1字符集以外的字符，其中的信息将会丢失。对于一些指定了ASCII编码的网络协议来说，这个方法或许有用，但多数情况下都应当避免使用。

writeChars和writeBytes都不会对输出流的字符串的长度编码。因此，你无法真正区分原始字符和作为字符串一部分的字符。writeUTF()方法则包括了字符串的长度。它将字符串本身用Unicode UTF-8编码的一个变体进行编码。由于这个变体编码方式与大多数非Java软件有点不兼容，所以应当只用于与其他使用DataInputStream读取字符串的Java程序进行数据交换。为了与所有其他软件交换UTF-8文本，应当使用有适当编码的InputStreamReader（如果Sun当初把这个方法及相应的读取方法命名为writeString()和readString()，而不是writeUTF()和readUTF()，那就不会产生任何混淆了）。

除了这些写入二进制数字和字符串的方法，DataOutputStream当然还有所有OutputStream类都有的平常的write()、flush()和close()方法。

DataInputStream与DataOutputStream是互补的。DataOutputStream写入的每一种格式，DataInputStream都可以读取。此外，DataInputStream还有通常的read()、available()、skip和close()方法，以及读取整个字节数组和文本行的方法。

有9个读取二进制数据的方法，这些方法对应于DataOutputStream的11个方法（writeBytes()或writeChars()没有相应的读取方法，这要通过一次读取1字节和字符来处理）：

```
public final boolean readBoolean() throws IOException
public final byte readByte() throws IOException
public final char readChar() throws IOException
public final short readShort() throws IOException
public final int readInt() throws IOException
public final long readLong() throws IOException
public final float readFloat() throws IOException
public final double readDouble() throws IOException
public final String readUTF() throws IOException
```

此外，DataInputStream提供了两个方法，可以读取无符号字节和无符号短整数，并返回等价的int。Java没有这些数据类型，但在读取C程序写入的二进制数据时会遇到：

```
public final int readUnsignedByte() throws IOException
public final int readUnsignedShort() throws IOException
```

DataInputStream有两个通常的多字节read()方法，可把数据读入一个数组或子数组，并返回读取的字节数。它还有两个readFully()方法，会重复地从底层输入流向一个数组读取数据，直到读取了所请求的字节数为止。如果不能读取到足够的数据，就会抛出IOException异常。如果你能提前知道要读取多少字节，这些方法尤其有用。例如，如果你已经从HTTP首部读取了Content-length（内容长度）字段，就能知道有多少字节的数据，这种情况下就可以很好地利用这些方法：

```
public final int read(byte[] input) throws IOException
public final int read(byte[] input, int offset, int length)
    throws IOException
public final void readFully(byte[] input) throws IOException
public final void readFully(byte[] input, int offset, int length)
throws IOException
```

最后，DataInputStream还提供了流行的readLine()方法，它读取用行结束符分隔的一行文本，并返回一个字符串：

```
public final String readLine() throws IOException
```

不过，任何情况下都不要使用这个方法，不仅是因为它已被废弃，而且它还有bug。之所以将这个方法废弃，是因为在大多数情况下它不能正确地将非ASCII字符转换为字节。这个任务现在由BufferedReader类的readLine()方法来处理。不过，这两个方法都存在同一个隐含的bug：它们并不总能把一个回车识别为行结束。实际上，readLine()只能识别换行或回车/换行对。在流中检测到回车时，readLine()在继续之前会等待，查看下一个字符是否为换行。如果是换行，就抛掉这个回车和换行，把这一行作为String返回。如果不是换行，就抛掉这个回车，把这一行作为String返回，刚读取的这个额外的字符会成为下一行的一部分。不过，如果回车是流的最后一个字符，那么readLine()会挂起，等待最后一个字符的出现，但这个字符永远也不会出现。

这个问题在读取文件时不太明显，因为几乎可以肯定会有下一个字符：如果没有别的字符，那么会由-1表示流结束。不过，在持久的网络连接中（如用于FTP和新型HTTP的连接），服务器或客户端可能只是在最后一个字符之后停止发送数据，并等待响应，而不会真正关闭连接。如果幸运，最终可能某一端的连接超时，你将得到一个IOException异常，不过这可能至少要花费几分钟，而且会使你丢失流的最后一行数据。如果不够幸运，程序将永远挂起。

## 阅读器和书写器

许多程序员在编码时有一个坏习惯，好像所有文本都是ASCII，或者至少是该平台的内置编码方式。虽然有些较老的、较简单的网络协议（如daytime、quote of the day和

chargen）确实指定文本采用ASCII编码方式，但对于HTTP和其他很多更新的协议却不是这样，它们允许多种本地化编码，如KOI8-R西里尔文、Big-5中文和土耳其语使用的ISO 8859-9。Java的内置字符集是Unicode的UTF-16编码。当编码不再是ASCII时，如果假定字节和字符实质上是一样的，这也会出问题。因此，对应于输入和输出流类层次体系，Java提供了一个基本上完整的镜像，用来处理字符而不是字节。

这个镜像体系中，两个抽象超类定义了读/写字符的基本API。java.io.Reader类指定读取字符的API。java.io.Writer指定写字符的API。对应输入和输出流使用字节的地方，阅读器和书写器会使用Unicode字符。Reader和Writer的具体子类允许读取特定的源和写入特定的目标。过滤器阅读器和书写器可以附加到其他阅读器或书写器上，以提供额外的服务或接口。

Reader和Writer最重要的具体子类是InputStreamReader和OutputStreamWriter类。InputStreamReader类包含一个底层输入流，可以从中读取原始字节。它根据指定的编码方式，将这些字节转换为Unicode字符。OutputStreamWriter从运行的程序中接收Unicode字符，然后使用指定的编码方式将这些字符转换为字节，再将这些字节写入底层输出流中。

除了这两个类，java.io包还提供了几个原始阅读器和书写器类，它们可以读取字符而不需要一个底层输入流，这些类包括：

- FileReader
- FileWriter
- StringReader
- StringWriter
- CharArrayReader
- CharArrayWriter

以上所列的前两个类可以处理文件，后四个由Java内部使用，所以在网络编程中不太常用。不过，除了构造函数不同，这些类与所有其他阅读器和书写器类一样，都有相同的公共接口。

## 书写器

Writer类是java.io.OutputStream类的映射。它是一个抽象类，有两个保护类型的构造函数。与OutputStream类似，Writer类从不直接使用；相反，会通过它的某个子类以多态方式使用。它有5个write()方法，另外还有flush()和close()方法：

```
protected Writer()
protected Writer(Object lock)
public abstract void write(char[] text, int offset, int length)
    throws IOException
public void write(int c) throws IOException
public void write(char[] text) throws IOException
public void write(String s) throws IOException
public void write(String s, int offset, int length) throws IOException
public abstract void flush() throws IOException
public abstract void close() throws IOException
```

write(char[] text, int offset, int length)方法是基础方法，其他四个write()都是根据它实现的。子类至少要覆盖这个方法以及flush()和close()，但是为了提供更高效的实现方法，大多数子类还覆盖了其他一些write()方法。例如，给定一个Writer对象w，可以这样写入字符串“Network”：

```
char[] network = {'N', 'e', 't', 'w', 'o', 'r', 'k'};
w.write(network, 0, network.length);
```

也可以用其他write()方法完成同样的任务：

```
w.write(network);
for (int i = 0; i < network.length; i++) w.write(network[i]);
w.write("Network");
w.write("Network", 0, 7);
```

所有这些例子表述都是同样的事情，只不过方式有所不同。在任何给定情况下，选择使用哪个方法主要考虑是否方便，以及你有什么偏好。不过，这些代码写入多少字节以及写入哪些字节，则取决于w使用的编码方式。如果使用big-endian UTF-16编码，那么它将依次写入下面14字节（以十六进制显示）：

```
00 4E 00 65 00 74 00 77 00 6F 00 72 00 6B
```

另一方面，如果w使用little-endian UTF-16，则将写入下面14字节的序列：

```
4E 00 65 00 74 00 77 00 6F 00 72 00 6B 00
```

如果w使用Latin-1、UTF-8或MacRoman，则写入下面7字节的序列：

```
4E 65 74 77 6F 72 6B
```

其他编码方式可能写入不同的字节序列。具体的输出取决于编码方式。

书写器可以缓冲，有可能直接串链到BufferedWriter，也有可能间接链入（因为其底层输出流是缓冲的）。为了强制将一个写入提交给输出介质，要调用flush()方法：

```
w.flush();
```

close()方法的行为与OutputStream的close()方法类似。close()刷新输出书写器，然后关闭底层输出流，并释放与之关联的所有资源：

```
public abstract void close() throws IOException
```

在书写器关闭后，进一步的写入会抛出IOException异常。

## OutputStreamWriter

OutputStreamWriter是Writer的最重要的具体子类。OutputStreamWriter会从Java程序接收字符。它根据指定的编码方式将这些字符转换为字节，并写入底层输出流。它的构造函数指定了要写入的输出流和使用的编码方式：

```
public OutputStreamWriter(OutputStream out, String encoding)
    throws UnsupportedEncodingException
```

JDK中包括一个Sun的native2ascii工具，其相关文档中列出了所有合法的编码方式。如果没有指定编码方式，就使用平台的默认编码方式。2013年，Mac上的默认编码方式是UTF-8，Linux上也大多如此。不过，如果本地操作系统配置为默认使用另外某个字符集，Linux上的默认编码方式可能有变化。在Windows上，默认编码方式会根据国家和配置而改变，但是在美国，Windows上默认的编码方式往往是Windows-1252，又叫做CP1252。默认字符集可能会在出乎意料的时候导致意外的问题。如果能明确地指定字符集，这往往比让Java为你选择一个字符集要好。例如，下面的代码段会用CP1253 Windows Greek编码方式写入荷马史诗奥德赛的前几个词：

```
OutputStreamWriter w = new OutputStreamWriter(
    new FileOutputStream("OdysseyB.txt"), "Cp1253");
 w.write("ἦμος δ' ἠριγένεια φάνη ῥοδοδάκτυλος Ἠώς");
```

除了构造函数，OutputStreamWriter只有通常的Writer方法（这些方法与所有Writer类中的用法相同），还有一个返回对象编码方式的方法：

```
public String getEncoding()
```

## 阅读器

Reader类是java.io.InputStream类的镜像。它是一个抽象类，有两个保护的构造函数。与InputStream和Writer类似，Reader类从不直接使用，只通过其子类来使用。它有三个read()方法，另外还有skip()、close()、ready()、mark()、reset()和markSupported()方法：

```
protected Reader()
protected Reader(Object lock)
```

```
public abstract int read(char[] text, int offset, int length)
    throws IOException
public int read() throws IOException
public int read(char[] text) throws IOException
public long skip(long n) throws IOException
public boolean ready()
public boolean markSupported()
public void mark(int readAheadLimit) throws IOException
public void reset() throws IOException
public abstract void close() throws IOException
```

read(char[] text, int offset, int length)方法是基础方法，其他两个read()方法都是根据它实现的。子类必须至少覆盖这个方法及close()，但是为了提供更高效的实现，大多数子类还会覆盖其他一些read()方法。

由于与对应的InputStream类似，大多数方法都很容易理解。read()方法将一个Unicode字符作为一个int返回，可以是0到65 535之间的一个值，或者在流结束时返回-1（理论上讲，它会返回一个UTF-16码点，不过这几乎等同于Unicode字符）。read(char[] text)方法尝试使用字符填充数组text，并返回实际读取的字符数，或者在流结束时返回-1。read(char[] text, int offset, int length)方法尝试将length个字符读入text的子数组中（从从offset开始持续length个字符）。它也会返回实际读取的字符数，或者在流结束时返回-1。skip(long n)方法跳过n个字符。mark()和reset()方法允许一些阅读器重置到字符序列中做标记的位置。markSupported()方法会告知阅读器是否支持标记和重置。close()方法会关闭阅读器和所有底层输入流，如果试图进一步读取则会抛出IOException异常。

尽管与InputStream非常相似，但也有所例外：Reader类有一个ready()方法，它与InputStream的available()的用途相同，但语义却不尽相同，尽管都涉及字节到字符转换。available()返回一个int，指定可以无阻塞地最少读取多少字节，但ready()只返回一个boolean，指示阅读器是否可以无阻塞地读取。问题在于，有些字符编码方式（如UTF-8）对于不同的字符会使用不同数量的字节。因此在实际从缓冲器区读取之前，很难说有多少个字符正在网络或文件系统的缓冲区中等待。

InputStreamReader是Reader的最重要的具体子类。InputStreamReader从其底层输入流（如FileInputStream或TelnetInputStream）中读取字节。它根据指定的编码方式将这些字节转换为字符，并返回这些字符。构造函数指定要读取的输入流和所用的编码方式：

```
public InputStreamReader(InputStream in)
public InputStreamReader(InputStream in, String encoding)
    throws UnsupportedEncodingException
```

如果没有指定编码方式，就使用平台的默认编码方式。如果指定了一个未知的编码方式，会抛出UnsupportedEncodingException异常。

例如，下面的方法将读取一个输入流，使用MacCyrillic编码方式将其全部转换为一个Unicode字符串：

```
public static String getMacCyrillicString(InputStream in)
    throws IOException {

  InputStreamReader r = new InputStreamReader(in, "MacCyrillic");
  StringBuilder sb = new StringBuilder();
  int c;
  while ((c = r.read()) != -1) sb.append((char) c);
  return sb.toString();
}
```

## 过滤器阅读器和书写器

InputStreamReader和OutputStreamWriter类就相当于输入和输出流之上的装饰器，把面向字节的接口改为面向字符的接口。完成之后，就可以将其他面向字符的过滤器放在使用java.io.FilterReader和java.io.FilterWriter类的阅读器或书写器上面。与过滤器流一样，有很多子类可以完成特定的过滤工作，包括：

- BufferedReader
- BufferedWriter
- LineNumberReader
- PushbackReader
- PrintWriter

BufferedReader和BufferedWriter类是基于字符的，对应于面向字节的BufferedInputStream和BufferedOutputStream类。BufferedInputStream和BufferedOutputStream中使用一个内部字节数组作为缓冲区，相应地，BufferedReader和BufferedWriter使用一个内部字符数组作为缓冲区。

当程序从BufferedReader读取时，文本会从缓冲区得到，而不是直接从底层输入流或其他文本源读取。当缓冲区清空时，将用尽可能多的文本再次填充，尽管这些文本不是全部都立即需要，这样可以使以后的读取速度更快。当程序写入一个BufferedWriter时，文本被放置在缓冲区中。只有当缓冲区填满或者当书写器显式刷新输出时，文本才会被移到底层输出流或其他目标，这使得写入也要快得多。

BufferedReader和BufferedWriter也有与阅读器和书写器关联的常用方法，如read()、ready()、write()和close()。这两个类都有两个构造函数，可以将BufferedReader或BufferedWriter串链到一个底层阅读器或书写器，并设置缓冲区的大小。如果没有设置大小，则使用默认的大小8192字符：

```
public BufferedReader(Reader in, int bufferSize)
public BufferedReader(Reader in)
public BufferedWriter(Writer out)
public BufferedWriter(Writer out, int bufferSize)
```

例如，前面的getMacCyrillicString()示例效率不太高，因为它每次只读取一个字符。由于MacCyrillic是一个单字节的字符集，所以也是每次读取1字节。不过，通过将一个BufferedReader串链到InputStreamReader，会使它运行得更快，如下所示：

```
public static String getMacCyrillicString(InputStream in)
    throws IOException {

  Reader r = new InputStreamReader(in, "MacCyrillic");
  r = new BufferedReader(r, 1024);
  StringBuilder sb = new StringBuilder();
  int c;
  while ((c = r.read()) != -1) sb.append((char) c);
  return sb.toString();
}
```

要让这个方法进行缓冲，只需要增加另外一行代码。算法的其他部分都不用改变，因为要用到的InputStreamReader方法只是Reader超类中声明的read()和close()方法，所有Reader子类都有这两个方法，BufferedReader也不例外。

BufferedReader类还有一个readLine()方法，它读取一行文本，并作为一个字符串返回：

```
public String readLine() throws IOException
```

这个方法可以替代DataInputStream中已经废弃的readLine()方法，它与该方法的行为基本相同。主要的区别在于，通过将BufferedReader串链到InputStreamReader，你可以采用正确的字符集读取行，而不是采用平台的默认编码方式。

这个BufferedWriter()类增加了一个其超类所没有的新方法，名为newLine()，也用于写入一行：

```
public void newLine() throws IOException
```

这个方法向输出插入一个与平台有关的行分隔符字符串。line.separator系统属性会确定这个字符串是什么：在UNIX和Mac OS X下可能是换行，在Mac OS 9下是回车，在Windows下是回车/换行对。由于网络协议一般会指定所需的行结束符，所以网络编程中不要使用这个方法，而应当显式地写入协议所需的行结束符。大多数情况下，所需的结束符都是回车/换行对。

## PrintWriter

PrintWriter类用于取代Java 1.0的PrintStream类，它能正确地处理多字节字符集和国际化文本。Sun最初计划废弃PrintStream而支持PrintWriter，但当它意识到这样做会使太多现有的代码失效（尤其是依赖于System.out的代码），就放弃了这种想法。尽管如此，新编写的代码还是应当使用PrintWriter而不是PrintStream。

除了构造函数，PrintWriter类也有与PrintStream几乎相同的方法集。包括：

```
public PrintWriter(Writer out)
public PrintWriter(Writer out, boolean autoFlush)
public PrintWriter(OutputStream out)
public PrintWriter(OutputStream out, boolean autoFlush)
public void flush()
public void close()
public boolean checkError()
public void write(int c)
public void write(char[] text, int offset, int length)
public void write(char[] text)
public void write(String s, int offset, int length)
public void write(String s)
public void print(boolean b)
public void print(char c)
public void print(int i)
public void print(long l)
public void print(float f)
public void print(double d)
public void print(char[] text)
public void print(String s)
public void print(Object o)
public void println()
public void println(boolean b)
public void println(char c)
public void println(int i)
public void println(long l)
public void println(float f)
public void println(double d)
public void println(char[] text)
public void println(String s)
public void println(Object o)
```

这些方法的行为大多与PrintStream中相同。只有4个write()方法有所例外，它们写入字符而不是字节。此外，如果底层的书写器能正确地处理字符集转换，那么PrintWriter的所有方法也能处理这种转换。这是对非国际化的PrintStream类的改进，但对于网络编程来说，仍然不太适合。很遗憾，PrintWriter也存在困扰PrintStream类的平台依赖性和错误报告信息量小等问题。

---

**提示：** 本章快速地浏览了java.io包，涵盖了编写网络编程所需的最起码的知识。更详细、更全面的介绍及更多的示例请查阅这个系列中我的另一本书《Java I/O》（O'Reilly出版）。

---

# 第3章

# 线程

回想网络从前的好日子，大约20世纪90年代初期，我们没有Web，没有HTTP，也没有图形界面浏览器。不过，我们有Usenet新闻、FTP和命令行接口，这一切是那么让人怀念！但是尽管旧时光很美好，却也存在一些问题。例如，当使用Kermit通过2400b/s调制解调器从热门的FTP网站下载几K字节的自由软件时，可能经常会遇到这样的错误消息：

```
% ftp eunl.java.sun.com
Connected to eunl.javasoft.com.
220 softwarenl FTP server (wu-2.4.2-academ[BETA- 16]+opie-2.32(1) 981105)
    ready.
Name (eunl.java.sun.com:elharo): anonymous
530-
530-    Server is busy. Please try again later or try one of our other
530-    ftp servers at ftp.java.sun.com. Thank you.
530-
530 User anonymous access denied.
Login failed.
```

事实上，在Internet只有几百万用户而不是数十亿用户的时候，我们远比现在更容易碰上超负荷的拥塞网站。问题在于，大多数FTP服务器会为每个连接创建（fork）一个新的进程（也就是说，100个并发用户意味着要处理100个额外的进程）。由于进程是相当重量级的，太多进程会很快让服务器吃不消。问题不在于机器不够强大，或者网络速度不够快，而是因为FTP服务器实现得太差。如果不是每个连接都需要一个新的进程，完全可以为更多的并发用户提供服务。

早期Web服务器也有这个问题，不过这个问题由于HTTP连接的短暂特性而有所掩盖。由于Web页面和嵌入的图片一般很小（至少与通常通过FTP获取的软件包相比要小得

多），也由于Web浏览器在获取各个文件后会“挂起”连接，而不是一次保持数分钟或几小时的连接，所以Web用户不会像FTP用户那样对服务器施加太多负担。不过，随着使用量的增长，Web服务器的性能仍会下降。根本问题在于，很容易编写代码将每个入站连接和每个新任务都当作单独的进程来处理（至少在UNIX下是如此），这种解决方案将无法扩展。等到服务器要处理成千上万个同时的连接时，性能就会变得像爬行一样慢了。

这个问题至少有两种解决方案。第一种是重用进程，而不是创建新的进程。服务器启动时，就创建固定数量的进程（比如300个）来处理请求。入站请求将放入一个队列。每个进程从队列中删除一个请求，为这个请求提供服务，然后返回到队列来得到下一个请求。尽管仍有300个单独的进程在运行，但是由于避免了建立和销毁进程的所有开销，现在这300个进程可以完成1000个进程的任务。这个数目是粗略估计的。你的实际情况可能有所出入，尤其是当你的服务器还没有达到一定的处理规模，即还没有遭遇可扩展性问题的时候，到底能有多少获益并不确定。不过，不论不生成新进程会有什么好处，起码重用旧的进程总会有更好的表现。

这个问题的第二种解决方案是，使用轻量级的线程来处理连接，而不是重量级的进程。虽然每个单独的进程都有自己的一块内存，但线程在资源使用上更宽松，因为它们会共享内存。使用线程来代替进程，可以再让你的服务器性能提升三倍。再结合使用可重用线程池（而不是可重用进程池），在同样的硬件和网络连接条件下，服务器的运行可以快9倍多！在服务器硬件上运行多个不同的线程，其影响是相对最小的，因为这些线程都在一个进程中运行。如果并发线程数达到4000至20 000时，大多数Java虚拟机可能会由于内存耗尽而无法承受。不过，通过使用线程池而不是为每个连接生成新线程，服务器每分钟就可以用不到100个线程来处理数千个短连接。

## 线程的替代方法

如果一个应用同时需要数千个持续很长时间的连接（这种应用相当少见），就要考虑异步I/O而不是线程。我们将在第11章介绍这个内容。选择器支持一个线程查询一组socket，找出哪些socket已经准备就绪可以读/写数据，然后顺序地处理这些准备好的socket。在这种情况下，必须基于通道和缓冲区来设计I/O而不是流。

由于现代虚拟机和操作系统中线程可以提供很高的性能，而且构建一个基于线程的服务器相对简单，所以开始时总会考虑采用基于线程的设计，直到遇到难以逾越的困难。如果确实遇到麻烦，应该考虑将应用分解到多个冗余的服务器上，而不要完全依仗一个服务器上的3倍性能提升。

> 当然，分解又会带来相应的设计问题，特别是在一致性方面，这是单个系统中没有的问题。不过，与利用单个系统相比，不论实现的效率如何，这样确实可以提供更大的可扩展性和冗余性。

遗憾的是，这种性能的提升并不是没有代价的，它会增加程序的复杂性。特别是多线程服务器（和其他多线程程序）要求程序员解决一些问题，而对于单线程程序而言原本不存在这些问题，尤其是安全性和活动性问题。因为不同的线程共享相同的内存，一个线程完全有可能会破坏另一个线程使用的变量和数据结构。这就类似于如果一个程序在没有内存保护机制的操作系统（如Windows 95）中运行，则有可能会破坏整个系统。因此，不同线程必须非常注意当时使用的资源。一般来讲，每个线程只有在确保资源不会改变或者它有独占访问权的时候才可以使用某个资源。不过，也可能两个线程太过小心，每个线程都在等待对资源的独占访问权，却永远都得不到。这会导致死锁，在这种情况下两个线程都在等待另一个线程所占有的资源。在没有得到另一个线程所保留的资源时，两个线程都不会继续处理，但同时都不愿意放弃已经占有的资源。

# 运行线程

线程如果以小写字母t打头（thread），这表示虚拟机中的一个单独、独立的执行路径。如果以大写字母T打头（Thread），则是java.lang.Thread类的一个实例。在虚拟机中执行的线程与虚拟机构造的Thread对象之间存在一种一一对应的关系。如果确实需要区分这两者，大多数情况下从上下文就能明显地看出到底是线程还是Thread对象。

为了启动一个新线程在虚拟机中运行，要构造一个Thread类的一个实例，调用它的start()方法，如下所示：

```
Thread t = new Thread();
t.start();
```

当然，这个线程没什么意思，因此它什么都没有做。要让线程完成一些操作，可以对Thread类派生子类，覆盖其run()方法。或者实现Runnable接口，将Runnable对象传递给Thread构造函数。我一般会选择第二种方法，因为这样可以更清楚地将线程完成的任务与线程本身区分开，但是在本书以及其他地方，你会看到这两种技术都在使用。不论采用哪一种方法，关键都在于run()方法，它的签名（signature）是：

```
public void run()
```

你应当把线程要做的所有工作都放在这个方法中。这个方法可以调用其他方法；可以构造其他对象；甚至可以生成其他线程。不过，线程要在这里启动，并在这里结束。

当run()方法完成时，线程也就消失了。事实上，run()对于线程就像main()方法对于非线程化传统程序的作用一样。单线程程序会在main()方法返回时退出。多线程程序会在main()方法以及所有非守护线程（nondaemon thread）都返回时才退出（守护线程完成后台任务，如垃圾回收，它们并不阻止虚拟机退出）。

## 派生Thread

考虑编写一个程序来计算多个文件的安全散列算法（SHA）摘要。在很大程度上，这是一个受限于I/O的程序（也就是说，它的速度会受到从磁盘读取文件所花费时间的限制）。如果将它编写为一个标准的程序，串行地处理这些文件，程序就会花费大量时间等待磁盘驱动器返回数据。这个限制对于网络程序尤其明显：程序运行的速度要比网络提供输入的速度快。因此程序的大量时间都耗费在阻塞中。这些时间本可以被其他线程所用，可以处理其他输入源或者完成不依赖于慢速输入的操作（不是所有多线程程序都有这个特点。有时即使并没有线程有大量空闲时间可以分配给其他线程，起码采用多线程设计程序会更容易，可以将程序分解为多个线程，分别执行独立的操作）。

示例3-1是Thread的子类，它的run()方法将为指定文件计算一个256位的SHA-2消息摘要。为此要用一个DigestInputStream读取这个文件。这个过滤器流在读取文件时会计算一个加密散列函数。读取结束时，可以从digest()方法得到这个散列。

示例3-1：DigestThread

```
import java.io.*;
import java.security.*;
import javax.xml.bind.*; // 对应DatatypeConverter， Java 6或JAXB 1.0需要有这个声明

public class DigestThread extends Thread {

  private String filename;

  public DigestThread(String filename) {
    this.filename = filename;
  }

  @Override
  public void run() {
    try {
      FileInputStream in = new FileInputStream(filename);
      MessageDigest sha = MessageDigest.getInstance( "SHA-256" );
      DigestInputStream din = new DigestInputStream(in, sha);
      while (din.read() != -1) ;
      din.close();
      byte[] digest = sha.digest();

      StringBuilder result = new StringBuilder(filename);
      result.append(": ");
      result.append(DatatypeConverter.printHexBinary(digest));
```

```
      System.out.println(result);
    } catch (IOException ex) {
      System.err.println(ex);
    } catch (NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }
  }

  public static void main(String[] args) {
    for (String filename : args) {
      Thread t = new DigestThread(filename);
      t.start();
    }
  }
}
```

main()方法从命令行中读取文件名，针对每个文件名启动一个新的DigestThread。这个线程的工作实际上在run()方法中完成。这里，由一个DigestInputStream读取文件，然后结果摘要以十六进制编码方式显示到System.out。注意，首先在一个本地StringBuffer变量result中建立线程的整个输出，再用一次方法调用将它显示在控制台上。更明显的一种方法是使用System.out.print()一次显示一部分，但这里不采取这种方法。其原因稍后讨论。

由于run()方法的签名是固定的，所以无法向其传递参数或从中返回值。因此，需要其他方法向线程传递信息和从中获得信息。传递信息最简单的方法是向构造函数传递参数，这会设置Thread子类中的字段，如前面所示。

由于线程的异步特性，要获得线程的信息并传回最初的调用线程，这会更加困难。示例3-1回避了这个问题，它没有将任何信息传回调用线程，只是把结果显示在System.out上。不过，大多数情况下，你会希望把信息传递给程序的其他部分。可以把计算结果存储于一个字段，并提供一个获取方法返回这个字段的值。但是如何知道这个值的计算什么时候结束呢？这非常棘手，本章后面将更详细地进行讨论。

---

**警告：** 如果对Thread派生子类，就应当只覆盖run()，而不要覆盖其他方法！Thread类的其他各个方法（如start()、interrupt()、join()、sleep()等）都有非常特定的语义，它们与虚拟机的交互很难在你自己的代码中重新实现。应当覆盖run()，并根据需要提供额外的构造函数和其他方法，但不要替换Thread的任何其他标准方法。

---

## 实现Runnable接口

要想避免覆盖标准Thread方法，一种办法就是不要派生Thread类，而是将希望线程完成的任务编写为Runnable接口的一个实例。这个接口声明了run()方法，这与Thread类完全一样：

```
public void run()
```

任何实现这个接口的类都必须提供这个方法，除了这个方法外，你可以自由地创建任何其他方法（可以使用你选择的任何方法名），而绝不会无意地妨碍线程的行为。它还允许你将线程的任务放在其他类的子类中，如Applet或HTTPServlet。要启动执行Runnable任务的一个线程，可以把这个Runnable对象传入Thread构造函数。例如：

```
Thread t = new Thread(myRunnableObject);
t.start();
```

对于大多数通过派生Thread子类来解决的问题，可以很容易地改为使用Runnable接口。示例3-2展示了这一点，它将示例3-1改写为使用Runnable接口，而不是派生Thread的子类。除了名字的改变外，需要做的唯一修改就是将extends Thread改为implements Runnable，并在main()方法中把DigestRunnable对象传递给Thread构造函数。程序的基本逻辑没有变。

示例3-2：DigestRunnable

```
import java.io.*;
import java.security.*;
import javax.xml.bind.*; // 对应DatatypeConverter，  Java 6或JAXB 1.0需要有这个声明

public class DigestRunnable implements Runnable {

  private String filename;

  public DigestRunnable(String filename) {
    this.filename = filename;
  }

  @Override
  public void run() {
    try {
      FileInputStream in = new FileInputStream(filename);
      MessageDigest sha = MessageDigest.getInstance("SHA-256");
      DigestInputStream din = new DigestInputStream(in, sha);
      while (din.read() != -1) ;
      din.close();
      byte[] digest = sha.digest();

      StringBuilder result = new StringBuilder(filename);
      result.append(": ");
      result.append(DatatypeConverter.printHexBinary(digest));
      System.out.println(result);
    } catch (IOException ex) {
      System.err.println(ex);
    } catch (NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }
  }
```

```
  public static void main(String[] args) {
  for (String filename : args) {
      DigestRunnable dr = new DigestRunnable(filename);
      Thread t = new Thread(dr);
      t.start();
    }
  }
}
```

并不认为实现Runnable接口一定优于扩展Thread类，没有强有力的理由支持这一点，反之亦然。在一些特殊的情况下，例如本章后面的示例3-14，在每个Thread对象的构造函数中调用Thread类的一些实例方法可能很有用。这就需要使用子类。在另外一些特定的情况下，可能需要将run()方法放在某个类中，而这个类要扩展另一个类（如HTTPServlet），这种情况下就必须使用Runnable接口。最后，有些崇尚面向对象的人认为，线程完成的任务实际上不是一种Thread，因此应当放在一个单独的类或接口（如Runnable）中，而不应放在Thread的子类中。我部分同意这种观点，但我不认为这个观点像其声称的那样理由充分。所以，本书中主要使用Runnable接口，但你也可以使用对你来说最方便的任何方法。

# 从线程返回信息

习惯了传统单线程过程式模型的程序员在转向多线程环境时，最难掌握的一点就是如何从线程返回信息。从结束的线程获得信息，这是多线程编程中最常被误解的方面之一。run()方法和start()方法不返回任何值。例如，假设不只是像示例3-1和示例3-2那样简单地显示SHA-256摘要，摘要线程需要把摘要返回给执行主线程。大多数人的第一个反应就是把结果存储在一个字段中，再提供一个获取方法，如示例3-3和示例3-4所示。示例3-3是一个计算指定文件摘要的Thread子类。示例3-4是一个简单的命令行用户界面，会接收文件名，并创建线程为这些文件计算摘要。

*示例3-3：使用存取方法返回结果的线程*

```
import java.io.*;
import java.security.*;

public class ReturnDigest extends Thread {

  private String filename;
  private byte[] digest;

  public ReturnDigest(String filename) {
    this.filename = filename;
  }

  @Override
  public void run() {
```

```
    try {
      FileInputStream in = new FileInputStream(filename);
      MessageDigest sha = MessageDigest.getInstance("SHA-256");
      DigestInputStream din = new DigestInputStream(in, sha);
      while (din.read() != -1) ; // 读取整个文件
      din.close();
      digest = sha.digest();
    } catch (IOException ex) {
      System.err.println(ex);
    } catch (NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }
  }

  public byte[] getDigest() {
    return digest;
  }
}
```

示例3-4：使用存取方法获得线程输出的主程序

```
import javax.xml.bind.*; // 对应DatatypeConverter

public class ReturnDigestUserInterface {

  public static void main(String[] args) {
    for (String filename : args) {
      // 计算摘要
      ReturnDigest dr = new ReturnDigest(filename);
      dr.start();

      // 现在显示结果
      StringBuilder result = new StringBuilder(filename);
      result.append(": ");
      byte[] digest = dr.getDigest();
      result.append(DatatypeConverter.printHexBinary(digest));
      System.out.println(result);
    }
  }
}
```

ReturnDigest类把计算结果存储在私有字段digest中，可以通过getDigest()来访问。ReturnDigestUserInterface中的main()方法循环处理由命令行得到的一个文件列表。它为每个文件启动一个新的ReturnDigest线程，然后试图使用getDigest()获取结果。不过，当你运行这个程序时，结果却不像你期望的那样：

```
D:\JAVA\JNP4\examples\03>java ReturnDigestUserInterface *.java
Exception in thread "main" java.lang.NullPointerException
  at javax.xml.bind.DatatypeConverterImpl.printHexBinary
  (DatatypeConverterImpl.java:358)
  at javax.xml.bind.DatatypeConverter.printHexBinary(DatatypeConverter.java:560)
  at ReturnDigestUserInterface.main(ReturnDigestUserInterface.java:15)
```

问题在于，主程序会在线程有机会初始化摘要之前就获取并使用摘要。在单线程程序里这个控制流可以正常工作，但在这里不行，因为在单线程程序里，dr.start()只是在同一个线程中调用run()方法，而这里情况有所不同。dr.start()启动的计算可能在main()方法调用dr.getDigest()之前结束，也可能还没有结束。如果没有结束，dr.getDigest()则会返回null，第一次尝试访问digest时会抛出一个NullPointerException异常。

## 竞态条件

一种可能的方法是把dr.getDigest()调用移到main()方法的后面部分，如下所示：

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

  ReturnDigest[] digests = new ReturnDigest[args.length];

  for (int i = 0; i < args.length; i++) {
    // 计算摘要
    digests[i] = new ReturnDigest(args[i]);
    digests[i].start();
  }

  for (int i = 0; i < args.length; i++) {
    // 现在显示结果
    StringBuffer result = new StringBuffer(args[i]);
    result.append(": ");
    byte[] digest = digests[i].getDigest();
    result.append(DatatypeConverter.printHexBinary(digest));

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

如果你够幸运，这会正常工作，你将得到期望的输出，如下：

```
D:\JAVA\JNP4\examples\03>java ReturnDigest2 *.java
AccumulatingError.java: 7B261F7D88467A1D30D66DD29EEEDE495EA16FCD3ADDB8B613BC2C5DC
BenchmarkScalb.java: AECE2AD497F11F672184E45F2885063C99B2FDD41A3FC7C7B5D4ECBFD2B0
CanonicalPathComparator.java: FE0AACF55D331BBF555528A876C919EAD826BC79B659C489D62
Catenary.java: B511A9A507B43C9CDAF626D5B3A8CCCD80149982196E66ED1BFFD5E55B11E226
...
```

但我要强调一下这里所说的“幸运”。你可能得不到这个输出。事实上，你可能仍会得到一个NullPointerException异常。这段代码是否能正常工作，完全取决于每个ReturnDigest线程是否在其getDigest()方法被调用之前结束。如果第一个for循环太快，在第一个for循环生成的线程结束之前就进入了第二个for循环，那么我们又会回到原点，遭遇同样的问题。更糟糕的是，程序看起来好像被挂起了，没有任何输出，甚至连栈轨迹都没有。

到底会得到正确的结果还是异常，或者是一个挂起的程序，这取决于很多因素，包括程序生成了多少线程，系统的CPU和磁盘的速度，系统使用多少个CPU，以及Java虚拟机为不同线程分配时间所用的算法。这些称为竞态条件（race condition）。能否得到正确结果依赖于不同线程的相对速度，而你无法控制这一点！我们需要一种更好的方法，以保证在摘要就绪前不会调用getDigest()。

## 轮询

大多数新手采用的解决方案是，让获取方法返回一个标志值（或者可能抛出一个异常），直到设置了结果字段为止。然后主线程定期询问获取方法，查看是否返回了标志之外的值。这个例子中，这表示要重复地测试digest是否为空，只有不为空才使用。例如：

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

  ReturnDigest[] digests = new ReturnDigest[args.length];

  for (int i = 0; i < args.length; i++) {
    // 计算摘要
    digests[i] = new ReturnDigest(args[i]);
    digests[i].start();
  }

  for (int i = 0; i < args.length; i++) {
    while (true) {
      // 现在显示结果
      byte[] digest = digests[i].getDigest();
      if (digest != null) {
        StringBuilder result = new StringBuilder(args[i]);
        result.append(": ");
        result.append(DatatypeConverter.printHexBinary(digest));
        System.out.println(result);
        break;
      }
    }
  }
}
```

这个解决方案是可行的。它会以正确的顺序给出正确的答案，而不考虑各个线程的相对运行速度。不过，它做了大量不需要做的工作。

更糟糕的是，这个解决方案不能保证一定能工作。在有些虚拟机上，主线程会占用所有可用的时间，而没有给具体的工作线程留出任何时间。主线程太忙于检查工作的完成情况，以至于没有时间来具体完成任务！显然这不是一个好方法。

## 回调

事实上，还有一种更简单有效的方法来解决这个问题。有了这个方法，我们完全可以淘汰前面的做法，不必通过无限循环来重复地轮询每个ReturnDigest对象来查看是否结束。这个方法的技巧在于，不是在主程序中重复地询问每个ReturnDigest线程是否结束（就像一个五岁小孩在长途汽车旅行中反复地问“我们到了吗？”这真的很烦人），而是让线程告诉主程序它何时结束。这是通过调用主类（即启动这个线程的类）中的一个方法来做到的。这被称为回调（callback），因为线程在完成时反过来调用其创建者。这样一来，主程序就可以在等待线程结束期间休息，而不会占用运行线程的时间。

当线程的`run()`方法接近结束时，要做的最后一件事情就是基于结果调用主程序中的一个已知方法。不是由主程序询问每个线程来寻求答案，而是由每个线程告知主程序答案。例如，示例3-5展示了一个与前面示例很相似的`CallbackDigest`类。不过，在`run()`方法的末尾，对于最初启动这个线程的类，它要将`digest`传递给这个类的`CallbackDigestUserInterface.receiveDigest()`静态方法。

示例3-5：CallbackDigest

```
import java.io.*;
import java.security.*;

public class CallbackDigest implements Runnable {

  private String filename;

  public CallbackDigest(String filename) {
    this.filename = filename;
  }

  @Override
  public void run() {
    try {
      FileInputStream in = new FileInputStream(filename);
      MessageDigest sha = MessageDigest.getInstance("SHA-256");
      DigestInputStream din = new DigestInputStream(in, sha);
      while (din.read() != -1) ; // 读取整个文件
      din.close();
      byte[] digest = sha.digest();
      CallbackDigestUserInterface.receiveDigest(digest, filename);
    } catch (IOException ex) {
      System.err.println(ex);
    } catch (NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }
  }
}
```

示例3-6中所示的`CallbackDigestUserInterface`类提供了`main()`方法。不过，与这

个程序的其他变体中的main()方法不同，这个方法只是为命令行中指定的文件启动线程。它不会具体读取、显示或对计算结果完成其他的操作。这些功能由一个单独的方法receiveDigest()来处理。receiveDigest()不在main()方法中调用，沿着main()方法的控制流所能到达的任何方法也不会调用这个receiveDigest()方法。实际上，它由每个线程单独调用。也就是说，receiveDigest()在摘要线程中运行，而不是在执行主线程中运行。

示例3-6：CallbackDigestUserInterface

```
import javax.xml.bind.*; // 对应DatatypeConverter， Java 6或JAXB 1.0需要有这个声明

public class CallbackDigestUserInterface {

  public static void receiveDigest(byte[] digest, String name) {
    StringBuilder result = new StringBuilder(name);
    result.append(": ");
    result.append(DatatypeConverter.printHexBinary(digest));
    System.out.println(result);
  }

  public static void main(String[] args) {
    for (String filename : args) {
      // 计算摘要
      CallbackDigest cb = new CallbackDigest(filename);
      Thread t = new Thread(cb);
      t.start();
    }
  }
}
```

示例3-5和示例3-6使用静态方法完成回调，这样CallbackDigest只需要知道CallbackDigestUserInterface中要调用的方法名。不过，回调实例方法也不会太难（而且回调实例方法更为常见）。这种情况下，进行回调的类必须有其回调对象的一个引用。通常情况下，这个引用通过线程构造函数的参数来提供。当run()方法接近结束时，要做的最后一件事情就是调用回调对象的实例方法来传递结果。例如，示例3-7展示了一个与前面很类似的CallbackDigest类。不过，这一次它有一个额外的字段，这是一个名为callback的InstanceCallbackDigestUserInterface对象。在run()方法的末尾，digest被传递给callback的receiveDigest()方法。InstanceCallbackDigestUserInterface对象本身在构造函数中设置。

示例3-7：InstanceCallbackDigest

```
import java.io.*;
import java.security.*;

public class InstanceCallbackDigest implements Runnable {

  private String filename;
```

```
  private InstanceCallbackDigestUserInterface callback;

  public InstanceCallbackDigest(String filename,
   InstanceCallbackDigestUserInterface callback) {
    this.filename = filename;
    this.callback = callback;
  }

  @Override
  public void run() {
    try {
      FileInputStream in = new FileInputStream(filename);
      MessageDigest sha = MessageDigest.getInstance("SHA-256");
      DigestInputStream din = new DigestInputStream(in, sha);
      while (din.read() != -1) ; // 读取整个文件
      din.close();
      byte[] digest = sha.digest();
      callback.receiveDigest(digest);
    } catch (IOException | NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }
  }
}
```

示例3-8中的InstanceCallbackDigestUserInterface类有main()方法和receiveDigest()方法，用于处理传入的摘要。示例3-8只是显示摘要，但功能更强的类还可以完成其他操作，如将摘要存储在一个字段中，用它启动另一个线程，或者对它完成进一步的计算。

示例3-8：InstanceCallbackDigestUserInterface

```
import javax.xml.bind.*; // 对应DatatypeConverter，Java 6或JAXB 1.0需要有这个声明

public class InstanceCallbackDigestUserInterface {

  private String filename;
  private byte[] digest;

  public InstanceCallbackDigestUserInterface(String filename) {
    this.filename = filename;
  }

  public void calculateDigest() {
    InstanceCallbackDigest cb = new InstanceCallbackDigest(filename, this);
    Thread t = new Thread(cb);
    t.start();
  }

  void receiveDigest(byte[] digest) {
    this.digest = digest;
    System.out.println(this);
  }

  @Override
  public String toString() {
```

```
    String result = filename + ": ";
    if (digest != null) {
      result += DatatypeConverter.printHexBinary(digest);
    } else {
      result += "digest not available";
    }
    return result;
  }

  public static void main(String[] args) {
    for (String filename : args) {
      // 计算摘要
      InstanceCallbackDigestUserInterface d
          = new InstanceCallbackDigestUserInterface(filename);
      d.calculateDigest();
    }
  }
}
```

使用实例方法代替静态方法进行回调要复杂一些，但有很多优点。首先，主类（这个例子中主类是InstanceCallbackDigestUserInterface）的各个实例只映射至一个文件，可以自然地跟踪记录这个文件的信息，而不需要额外的数据结构。此外，这个实例在必要时可以很容易地重新计算某个特定文件的摘要。实际上，经证明这种机制有更大的灵活性。但是也有一个警告。注意这里新增了启动线程的calculateDigest()方法。从逻辑上考虑，你可能认为这属于构造函数。不过，在构造函数中启动线程很危险，特别是线程将回调原来的对象时。这里有一个竞态条件，可能会在构造函数结束而且对象完全初始化之前允许新线程进行回调。在这里不太可能，因为启动新线程是构造函数做的最后一件事。不过，至少理论上是有可能的。因此，万无一失的做法是避免在构造函数中启动线程。

相比于轮询机制，回调机制的第一个优点是不会浪费那么多CPU周期。但更重要的优点是回调更灵活，可以处理涉及更多线程、对象和类的更复杂的情况。例如，如果有多个对象对线程的计算结果感兴趣，那么线程可以保存一个要回调的对象列表。特定的对象可以通过调用Thread或Runnable类的一个方法把自己添加到这个列表中来完成注册，表示自己对计算结果很感兴趣。如果有多个类的实例对结果感兴趣，可以定义一个新的interface（接口），所有这些类都要实现这个新接口。这个interface（接口）将声明回调方法。

如果你对此有种似曾相识的感觉，可能是因为你以前见过这种机制。这正是在Swing、AWT和JavaBean中处理事件的方法。AWT在程序之外的一个单独的线程中运行。组件和bean通过回调在特定接口（如ActionListener和PropertyChangeListener）中声明的方法来通知事件的发生。监听者对象使用Component类中的方法（如addActionListener()和addPropertyChangeListener()）来完成注册，表示对特定组件触发的事件感兴趣。

在组件内部，已注册的监听者存储在由java.awt.AWTEventMulticaster构成的一个链表中。这种机制有一个更一般的名字：观察者（Observer）设计模式。

## Future、Callable和Executor

Java 5 引入了多线程编程的一个新方法，通过隐藏细节可以更容易地处理回调。不再是直接创建一个线程，你要创建一个ExecutorService，它会根据需要为你创建线程。可以向ExecutorService提交Callable任务，对于每个Callable任务，会分别得到一个Future。之后可以向Future请求得到任务的结果。如果结果已经准备就绪，就会立即得到这个结果。如果还没有准备好，轮询线程会阻塞，直到结果准备就绪。这种做法的好处是，你可以创建很多不同的线程，然后按你需要的顺序得到你需要的答案。

例如，假设你要找出一个很大的数字数组中的最大值。如果采用最原始的方法实现，需要的时间为O(n)，其中n是数组中的元素个数。不过，如果可以将这个工作分解到多个线程，每个线程分别在一个单独的内核上运行，这样就会快得多。为了便于说明，下面假设需要两个线程。

Callable接口定义了一个call()方法，它可以返回任意的类型。示例3-9是一个Callable，它会采用最明显的方式查找数组的一个分段中的最大值。

示例3-9：FindMaxTask

```
import java.util.concurrent.Callable;

class FindMaxTask implements Callable<Integer> {

  private int[] data;
  private int start;
  private int end;

  FindMaxTask(int[] data, int start, int end) {
    this.data = data;
    this.start = start;
    this.end = end;
  }

  public Integer call() {
    int max = Integer.MIN_VALUE;
    for (int i = start; i < end; i++) {
      if (data[i] > max) max = data[i];
    }
    return max;
  }
}
```

尽管可以直接调用call()方法，但这并不是它的本来目的。实际上，你要把Callable对象提交给一个Executor，它会为每个Callable对象创建一个线程（Executor还可以使用

其他策略，例如，它可以使用一个线程按顺序调用这些callable，不过对于这个问题来说，每个callable分别对应一个线程是一个很好的策略）。如示例3-10所示。

示例3-10：MultithreadedMaxFinder

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

public class MultithreadedMaxFinder {

  public static int max(int[] data) throws InterruptedException, ExecutionException {

    if (data.length == 1) {
      return data[0];
    } else if (data.length == 0) {
      throw new IllegalArgumentException();
    }

    // 将任务分解为两部分
    FindMaxTask task1 = new FindMaxTask(data, 0, data.length/2);
    FindMaxTask task2 = new FindMaxTask(data, data.length/2, data.length);

    // 创建2个线程
    ExecutorService service = Executors.newFixedThreadPool(2);

    Future<Integer> future1 = service.submit(task1);
    Future<Integer> future2 = service.submit(task2);

    return Math.max(future1.get(), future2.get());
  }
}
```

这里会同时搜索这两个子数组，所以对于合适的硬件和规模很大的输入，这个程序运行的速度几乎可以达到原来的两倍。不仅如此，与先找出数组前一半的最大值再找出数组后一半的最大值的做法相比，这个代码几乎同样简单和直接，而不用担心线程或异步性。不过，这里有一个重要的区别。在示例3-10的最后一个语句中，调用future1.get()时，这个方法会阻塞，等待第一个FindMaxTask完成。只有当第一个FindMaxTask完成时，才会调用future2.get()。也有可能第二个线程已经结束，在这种情况下，结果值会直接返回，但是如果第二个线程还没有结束，同样的，也会等待这个线程完成。一旦两个线程都结束，将比较它们的结果，并返回最大值。

Future是一种非常方便的做法，可以启动多个线程来处理一个问题的不同部分，然后等待它们全部都结束之后再继续。executor和executor服务允许你用不同的策略将任务分配给不同的线程。这个例子只使用了两个线程，不过完全可以使用更多的线程，并重用这些线程来完成多个任务。只要你能把任务分解到适当独立的部分，Executor就可以隐藏异步性的很多细节。

# 同步

我的书架上堆满了书，包括很多一样的书，过时的书，以及10年里我从没看过、可能将来也不会再看的书。这些年来，为了购置这些书，花了我上万美元，甚至更多。不过，离我的公寓两个街区就有一个中心布鲁克林公共图书馆，那里的书架上也堆满了书。100多年来，这个图书馆为其收藏花费了上百万美元。但区别在于它的书由布鲁克林的居民所共享，因此这些书有很高的利用率。所收藏的大多数书一年会使用很多次。虽然公共图书馆在买书和存书方面比我花的钱多很多，但图书馆阅读每页书的成本却比我的个人书架要低很多。这正是共享资源的优点。

当然，共享资源也有缺点。如果我需要图书馆的一本书，我必须走到图书馆，还要在书架上寻找我需要的书。我必须排队办手续拿到这本书，或者我只能在图书馆看这本书而不能带回家。有时候，别人已经借走了这本书，我就必须填写预约单，申请这本书归还时能为我保留。另外我不能在书边上写笔记，不能对段落做标记，也不能撕下几页粘在我的公告牌上（唔，也许我可以这样做，但是如果这样做了，就会大大地降低这本书对于将来其他借阅者的用处；如果图书馆抓到了我，我可能会失去借阅资格）。从图书馆借书而不是自己买一本，在时间和方便性上会有很大的损失，但能够节约钱和存储空间。

线程就像是图书馆的借阅者，它从一个中心资源池中借阅。线程通过共享内存、文件句柄、socket和其他资源使得程序更高效。只要两个线程不同时使用相同的资源，多线程的程序就比多进程程序高效得多，在多进程程序中，每个进程都要为各个资源维护自己的副本。多线程程序的缺点是，如果两个线程同时访问同一个资源，其中一个就必须等待另一个结束。如果其中一个没有等待，资源就可能被破坏。让我们看一个特定的例子。考虑示例3-1和示例3-2中的`run()`方法。如前所述，这个方法把结果建立为一个`String`，然后使用一个`System.out.println()`调用将这个`String`在控制台打印出来。输出如下：

```
Triangle.java: B4C7AF1BAE952655A96517476BF9DAC97C4AF02411E40DD386FECB58D94CC769
InterfaceLister.java: 267D0EFE73896CD550DC202935D20E87CA71536CB176AF78F915935A6
Squares.java: DA2E27EA139785535122A2420D3DB472A807841D05F6C268A43695B9FDFE1B11
UlpPrinter.java: C8009AB1578BF7E730BD2C3EADA54B772576E265011DF22D171D60A1881AFF51
```

四个线程并行地运行，生成了这个输出。每个线程向控制台写入一行。写入这些行的顺序是不可预见的，因为线程的调度不可预知，但每一行会作为一个整体写入。不过，假如使用这个`run()`方法的另一个版本，不是将中间结果存储在`String`变量`result`中，而是当这些中间结果可用时就直接显示在控制台上：

```
@Override
public void run() {
```

```
    try {
      FileInputStream in = new FileInputStream(filename);
      MessageDigest sha = MessageDigest.getInstance("SHA-256");
      DigestInputStream din = new DigestInputStream(in, sha);
      while (din.read() != -1) ; // 读取整个文件
      din.close();
      byte[] digest = sha.digest();
      System.out.print(input + ": ");
      System.out.print(DatatypeConverter.printHexBinary(digest));
      System.out.println();
    } catch (IOException ex) {
      System.err.println(ex);
    } catch (NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }
  }
```

运行这个程序并提供同样的输入时，输出可能会是这样：

```
Triangle.java: B4C7AF1BAE952655A96517476BF9DAC97C4AF02411E40DD386FECB58D94CC769
InterfaceLister.java: Squares.java: UlpPrinter.java:
C8009AB1578BF7E730BD2C3EADA54B772576E265011DF22D171D60A1881AFF51
267D0EFE73896CD550DC202935D20E87CA71536CB176AF78F915935A6E81B034
DA2E27EA139785535122A2420D3DB472A807841D05F6C268A43695B9FDFE1B11
```

不同文件的摘要都混在了一起！无法区分哪个数字属于哪个摘要。很明显，这是个很大的问题。

出现这种混乱的原因在于，System.out由4个不同的线程共享。如果一个线程通过多个System.out.print()语句向控制台写输出，有可能它还没有完成所有写入，就有另一个线程插进来，开始写它的输出。哪个线程会抢先于其他线程，这个具体顺序无法确定。每次运行这个程序时，你都可能会看到稍有不同的输出。

需要有一种办法能够指定一个共享资源只能由一个线程独占访问来执行一个特定的语句序列。在这个例子中，共享资源是System.out，需要独占访问的语句是：

```
System.out.print(input + ": ");
System.out.print(DatatypeConverter.printHexBinary(digest));
System.out.println();
```

## 同步块

为了指示这5行代码应当一起执行，要把它们包围在一个synchronized块中，它会对System.out对象同步，如下所示：

```
synchronized (System.out) {
  System.out.print(input + ": ");
  System.out.print(DatatypeConverter.printHexBinary(digest));
```

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

一旦线程开始打印这些值，所有其他线程在打印它们的值之前就必须停止，需要等待这个线程结束。同步要求在同一个对象上同步的所有代码要连续地运行，而不能并行运行。例如，如果另一个不同的类和不同进程中的代码恰好也对System.out同步，那么它也不能与这个代码块并行运行。不过，对不同对象同步的代码或者根本不同步的代码仍可以与这个代码并行运行。即使它使用了System.out也可以这么做。Java没有提供任何方法来阻止其他线程使用共享资源。它只能防止对同一个对象同步的其他线程使用这个共享资源。

---

**提示：** 事实上，PrintStream类在内部要求大多数方法都对PrintStream对象同步（如这个例子中的System.out）。换句话说，调用System.out.println()的其他各个线程都会对System.out同步，而且必须等待这个代码结束。从这个方面来看，PrintStream很独特。大多数其他OutputStream子类都不会自行同步。

---

只要有多个线程共享资源，都必须考虑同步。这些线程可能是相同Thread子类的实例，或者使用了相同的Runnable类，也可能是完全不同的类的实例。关键在于这些线程所共享的资源，而不是这些线程是哪个类。只有当两个线程都拥有相同对象的引用时，同步才成为问题。在前面的例子中，问题就在于多个线程都访问同一个PrintStream对象System.out。在这种情况下，导致冲突的是一个静态类变量。不过，实例对象也会有问题。

例如，假设你的Web服务器保存了一个日志文件。这个日志文件可能用示例3-11中的类来表示。这个类本身不使用多线程。不过，如果Web服务器使用多线程处理入站连接，那么每个线程都需要访问相同的日志文件，因而会访问相同的LogFile对象。

示例3-11：LogFile

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

public class LogFile {

  private Writer out;

  public LogFile(File f) throws IOException {
    FileWriter fw = new FileWriter(f);
    this.out = new BufferedWriter(fw);
  }

  public void writeEntry(String message) throws IOException {
    Date d = new Date();
    out.write(d.toString());
```

```
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }

  public void close() throws IOException {
    out.flush();
    out.close();
  }
}
```

在这个类中，writeEntry()方法获得当前日期和时间，然后使用四个单独的out.write()调用写入底层文件。如果两个或多个线程都有同一个LogFile对象的引用，而且一个线程在写数据的过程中被另一个线程打断，这就会出现问题。一个线程可能写入了日期和一个tab（制表符），然后下一个线程可能写入三个完整的记录。然后，第一个线程再写入消息、回车和换行。解决这个问题的方法还是同步。不过，在这个例子中，要对哪个对象同步呢？对此有两个很好的选择。第一种选择是对Writer对象out同步。例如：

```
public void writeEntry(String message) throws IOException {
  synchronized (out) {
    Date d = new Date();
    out.write(d.toString());
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }
}
```

这样就能正常工作了，这是因为所有使用这个LogFile对象的线程也会使用属于这个LogFile的同一个out对象。out是私有的，这并没有关系。虽然它会由其他线程和对象使用，但只会在LogFile类中引用。此外，虽然在这里是对out对象同步，但需要保护不被中断的是writeEntry()方法。Writer类都有自己的内部同步，这会防止一个线程中断另一个线程中的write()方法（对于输入和输出流则并非如此，只有PrintStream有所例外。写一个输出流时有可能被另一个线程中断）。每个Writer都有一个lock字段，指示书写器要对所写的哪个对象同步。

第二种可能性是对LogFile对象本身同步。这很简单，只要用到this关键词。例如：

```
public void writeEntry(String message) throws IOException {
  synchronized (this) {
    Date d = new Date();
    out.write(d.toString());
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }
}
```

## 同步方法

由于对对象本身同步整个方法体是很常见的，所以Java为此提供了一个快捷方式。可以通过向方法声明添加synchronized修饰符，对当前对象（this引用）同步整个方法。例如：

```
public synchronized void writeEntry(String message) throws IOException {
  Date d = new Date();
  out.write(d.toString());
  out.write('\t');
  out.write(message);
  out.write("\r\n");
}
```

对于同步问题，仅仅向所有方法添加synchronized修饰符并不是一劳永逸的解决方案。首先，它使得很多VM的性能严重下降（不过在这方面更多最新的VM已经大为改进），可能会使代码速度降低三分之一或者更多。其次，它大大增加了死锁的可能性。第三，也是最重要的，并不总是对象本身需要防止同时修改或访问，如果只是对该方法所属类的实例进行同步，可能并不能保护真正需要保护的对象。例如，在这个例子中，我们真正要避免的是两个线程同时写入out。如果其他类有与LogFile完全无关的out的引用，这些写入就会失败。不过在这个例子中，同步LogFile对象就足够了，因为out是一个私有实例变量。由于不会提供这个对象的引用，除非通过LogFile类，否则其他对象没有办法调用这个对象。因此，在这里同步LogFile对象与同步out有同样的效果。

## 同步的替代方式

对于由线程调度引起的不一致的行为，同步并不总是这个问题最好的解决方案。还有很多技术可以完全避免使用同步。第一个方法是，在可能的情况下，要使用局部变量而不是字段。局部变量不存在同步问题。每次进入一个方法时，虚拟机将为这个方法创建一组全新的局部变量。这些变量在方法外部是不可见的，而且方法退出时将被撤销。因此，一个局部变量不可能由两个不同的线程共享。每个线程都有自己单独的一组局部变量。

基本类型的方法参数也可以在单独的线程中安全地修改，因为Java按值而不是按引用来传递参数。以此推断，像Math.sqrt()等纯函数只取0个或多个基本类型参数，完成一些计算，并返回一个值，而不与任何类的字段交互，这些函数自然就是线程安全的。这些方法通常声明为静态方法或者应当声明为静态方法。

对象类型的方法参数有些麻烦，因为按值传递的实际参数是对象的引用。例如，假设将一个数组的引用传入sort()方法。当这个方法对数组排序时，没有办法阻止同样由这个数组引用的其他线程修改数组中的值。

String参数是安全的，因为它们是不可变（immutable）的（也就是说，一旦创建了一个String对象，它就不能被任何线程修改）。不可变对象永远也不会改变状态。其字段的值只在构造函数运行时设置一次，其后就不会再改变。StringBuffer参数是不安全的，因为它们并不是不可变的，在创建后还可以修改。

构造函数一般不需要担心线程安全问题。在构造函数返回前，没有线程有这个对象的引用，所以不可能有两个线程都有这个对象的引用（最有可能的问题是构造函数依赖于另一个线程中的另一个对象，这个对象可能在构造函数运行时改变，但这种情况并不常见。另外，如果构造函数以某种方式把它正在创建的对象的引用传递给一个不同的线程，也有可能出现问题，但这种情况也不常见）。

你可以在自己的类中利用不可变性。要使一个类做到线程安全，这往往是最简单的方法，通常会比确定哪些方法或代码段要进行同步容易得多。要使一个对象不可变，只要将其所有字段声明为private（私有）和final（最终），而且不要编写任何可能改变它们的方法。核心Java库中的很多类都是不可变的（如java.lang.String、java.lang.Integer、java.lang.Double等）。这使得这些类在某些方面用处不大，但的确能让它们有更强的线程安全性。

第三种技术是将非线程安全的类用作为线程安全的类的一个私有字段。只要包含类只以线程安全的方式访问这个非安全类，而且只要永远不让这个私有字段的引用泄漏到另一个对象中，那么这个类就是安全的。这个技术的一个例子就是Web服务器可能使用非同步的LogFile类，但是为每个单独的线程提供它自己单独的日志，这样各个线程之间就不会有共享的资源。

有些情况下，你可以使用java.util.concurrent.atomic包中特意设计为保证线程安全但可变的类。具体来讲，可以不使用int而使用AtomicInteger。不使用long而使用AtomicLong。不使用boolean而使用AtomicBoolean。不使用int[]而使用AtomicIntegerArray。不使用引用变量，而是把对象存储在一个AtomicReference中，不过需要说明，这不会让对象本身也是线程安全的，只是该引用变量的获取和设置是线程安全的。如果可以利用现代CPU上快速的机器级线程安全指令，这些类可能比其相应基本类型的同步访问快得多。

对于映射和列表等集合，可以使用java.util.Collections的方法把它们包装在一个线程安全的版本中。例如，如果有一个集合foo，可以用Collections.synchronizedSet(foo)得到这个集合的一个线程安全视图。如果有一个列表foo，可以使用Collections.synchronizedList(foo)来得到它的线程安全视图。对于映射，需要调用Collections.synchronizedMap(foo)等。为了能正常工作，在此之后必须只使用Collections.synchronizedSet/List/Map返回的视图。如果偶尔访问了原来的底层数据结构，那么不

论是原来的数据结构还是同步视图都无法做到线程安全。

不管怎样，要认识到这只是单个的原子方法调用。如果需要作为一个原子连续地完成两个操作，中间不能有中断，就需要同步。因此，举例来说，即使一个列表通过Collections.synchronizedList()同步，如果希望迭代处理这个列表，仍然需要对它同步，因为这可能涉及很多连续的原子操作。尽管每个方法调用确实是原子的，可以保证安全，但是如果没有明确的同步，这个操作序列并不一定安全。

## 死锁

同步会导致另一个可能的问题：死锁（deadlock）。如果两个线程需要独占访问同样的一个资源集，而每个线程分别有这些资源的不同子集的锁，就会发生死锁。如果两个线程都不愿意放弃已经拥有的资源，就会进入无限停止状态。在一般意义上讲，这不完全是挂起，因为从操作系统角度看，程序仍然是活动的，行为也是正常的，但对于用户而言。这与挂起没有什么区别。

再来看图书馆的例子，在下面这种情况下会发生死锁：Jack和Jill都要撰写关于托马斯·杰斐逊的学期论文，他们都需要两本书《Thomas Jefferson and Sally Hemings: An American Controversy》和《Sally Hemings and Thomas Jefferson: History, Memory and Civic Culture》。如果Jill已经借到了第一本，而Jack借到了第二本，倘若他们都不愿意放弃已经借到的书，那么都将无法完成论文。最终截止日期到了，他们都得到一个F。这就是死锁问题。

更糟糕的是，死锁可能是偶发性的bug，很难检测。死锁通常取决于不可预知的时间问题。大多数情况下，Jack或Jill会首先到达图书馆，把两本书都借到。这种情况下，先拿到书的人撰写好论文，再归还这两本书，然后另一个人拿到书并撰写论文。只有在极少情况下他们会同时到达，每人拿到其中一本书。100次里有99次，或者1000次里有999次，程序都能完全正常地运行。只有在极少情况下会毫无征兆地挂起。当然，如果一个多线程服务器每分钟处理成百上千个请求，即使每百万个请求才发生一次问题，也会迅速地让服务器挂起。

要防止死锁，最重要的技术是避免不必要的同步。如果有其他方法可以确保线程安全，比如让对象不可变或保存对象的一个局部副本，就最好使用那种方法（而不是同步）。同步应当是确保线程安全的最后一道防线。如果确实需要同步，要保持同步块尽可能小，而且尽量不要一次同步多个对象。但这可能很困难，因为你的代码会调用Java类库的很多方法，这些方法可能会在你不知情的情况下同步一些对象。因此，实际同步的对象可能比你预想的要多得多。

一般情况下，最好仔细考虑是否会出现死锁问题，并围绕这一点来设计你的代码。如果多个对象需要操作相同的共享资源集，要确保以相同的顺序请求这些资源。例如，如果类A和类B需要独占访问对象X和对象Y，要确保两个类都先请求X后请求Y。除非已经拥有X，否则这两个类都不能请求Y，如果能做到这一点，死锁就不是问题。

# 线程调度

当多个线程同时运行时（更正确的说法是，当多个线程可以同时运行时），必须考虑线程调度问题。你需要确保所有重要的线程至少要得到一些时间来运行，更重要的线程要得到更多的时间。此外，你希望保证线程以合理的顺序执行。如果Web服务器有10个排队的请求，每个请求都需要5s进行处理，你不会希望串行地处理它们。如果这样做，第一个请求将在5秒内结束，而第二个将需要10秒，第三个需要15秒，依此类推，直到最后一个请求，它必须等待将近1分钟才能得到服务。到那时，用户很可能已经去浏览其他网页了。通过并行地运行线程，就能够在总共仅仅10秒内处理完所有10个请求。这种策略之所以可行，这是因为在为一个典型的Web请求提供服务时，会有大量的空闲时间，在这段时间内线程只是在等待网络跟上CPU的速度，虚拟机的线程调度器完全可以将这段时间用于其他线程。不过，CPU受限的线程（而不是在网络程序中更常见的I/O受限线程）可能永远不会达到这种程度，如果CPU受限，线程往往更多地忙于处理，而不是等待更多的输入。这样的线程可能占用所有可用的CPU资源，使得所有其他线程处于“饥饿”状态。仔细考虑一下就可以避免这个问题。事实上，与同步不当或死锁相比，避免“饥饿”问题要容易得多。

## 优先级

不是所有线程创建时都是均等的。每个线程都有一个优先级，指定为一个从0到10的整数。当多个线程可以运行时，虚拟机通常只运行最高优先级的线程，但这并不是一个严格的规则。在Java中，10是最高优先级，0是最低优先级。默认优先级为5，除非特意指定其他设置，否则你的线程都将具有这个优先级。

---

**警告**：这与UNIX区分进程优先级的一般方式刚好相反，UINX中，进程的优先级数越大，进程获得的CPU时间就越少。

---

以下三个优先级（1、5和10）通常指定为三个命名常量Thread.MIN_PRIORITY、Thread.NORM_PRIORITY和Thread.MAX_PRIORITY：

```
public static final int MIN_PRIORITY = 1;
public static final int NORM_PRIORITY = 5;
```

```
public static final int MAX_PRIORITY = 10;
```

**警告**：并不是所有操作系统都支持这11个不同的优先级。例如，Windows只有7个优先级。在Windows上，优先级1和2、3和4、6和7以及8和9会做同样的处理（也就是说，优先级为9的线程并不会抢占优先级为8的线程）。

有时你希望给一个线程更多的时间。与用户交互的线程应当获得非常高的优先级，这样就能感觉到响应非常快。另一方面，在后台完成计算的线程应当获得低优先级。很快结束的任务应当有高优先级。将花费很长时间的任务应当有低优先级，这样就不会妨碍其他任务。线程的优先级可以使用setPriority()方法来改变：

```
public final void setPriority(int newPriority)
```

如果试图超出最大优先级，或者设置为一个非正的优先级，这些将抛出一个IllegalArgumentException异常。

例如，在示例3-11中，你可能希望完成计算的线程比生成这个线程的主程序有更高的优先级。这很容易做到，只需要修改calculateDigest()方法，将生成的各个线程的优先级设置为8：

```
public void calculateDigest() {
  ListCallbackDigest cb = new ListCallbackDigest(filename);
  cb.addDigestListener(this);
  Thread t = new Thread(cb);
  t.setPriority(8);
  t.start();
}
```

不过，一般情况下要尽量避免对线程使用太高的优先级，因为这要冒一定的风险，可能使其他低优先级线程遭受“饥饿”之苦。

## 抢占

每个虚拟机都有一个线程调度器，确定在给定时刻运行哪个线程。主要有两种线程调度：抢占式（preemptive）和协作式（cooperative）。抢占式线程调度器确定一个线程正常地轮到其CPU时间时，会暂停这个线程，将CPU控制权交给另外的线程。协作式线程调度器在将CPU控制权交给其他线程前，会等待正在运行的线程自己暂停。与使用抢占式线程调度的虚拟机相比，使用协作式线程调度器的虚拟机更容易使线程陷入“饥饿”，因为一个高优先级的非协作线程会独占整个CPU。

所有Java虚拟机都确保在不同优先级之间使用抢占式线程调度。也就是说，当一个低优

先级线程正在运行，而一个高优先级线程准备运行时，虚拟机会或早或晚（可能很早）暂停这个低优先级线程，让高优先级线程运行。高优先级线程就抢占（preempt）了低优先级线程。

如果多个相同优先级的线程准备运行，这种情况比较棘手。抢占式线程调度器偶尔会暂停其中一个线程，让下一个线程得到一些CPU时间。不过，协作式线程调度器不会这样。它将等待正在运行的线程明确放弃控制或者运行到最后停止（到达停止点）。倘若正在运行的线程永远也不放弃控制权，而且永远不会到达停止点，如果没有更高优先级线程抢占这个正在运行的线程，那么其他所有线程都会陷入“饥饿”状态。这很不好。重要的一点是，要确保所有线程自身定期地暂停，这样其他线程才可以有运行的机会。

---

**警告：** 如果在一个使用抢占式线程调度的虚拟机上开发，饥饿问题可能很难发现。即使你的机器上没有出现问题，这并不表示这个问题不会在客户的机器上出现（如果他们的虚拟机使用协作式线程调度）。目前大多数虚拟机都使用抢占式线程调度，但有些较早的虚拟机仍采用协作式调度，另外在一些特定用途的Java虚拟机（如面向嵌入式环境的虚拟机）中也可能会遇到协作式调度。

---

为了能让其他线程有机会运行，一个线程有10种方式可以暂停或者指示它准备暂停。这包括：

- 可以对I/O阻塞。
- 可以对同步对象阻塞。
- 可以放弃。
- 可以休眠。
- 可以连接另一个线程。
- 可以等待一个对象。
- 可以结束。
- 可以被更高优先级线程抢占。
- 可以被挂起。
- 可以停止。

要检查你编写的每一个`run()`方法，确保这些条件之一会以合理的频率出现。最后两种可能性已经废弃不用，因为它们可能会让对象处于不一致的状态，所以我们来看能够让线程成为虚拟机中协作的一员的其他8种方法。

## 阻塞

任何时候线程必须停下来等待它没有的资源时，就会发生阻塞。要让网络程序中的线程自动放弃CPU控制权，最常见的方式是对I/O阻塞。由于CPU比网络和磁盘快得多，网络程序经常会在等待数据从网络到达或向网络发送数据时阻塞。即使只阻塞几毫秒，这一点时间也足够其他线程用来完成重要的任务。

线程在进入一个同步方法或代码块时也会阻塞。如果这个线程没有所同步对象的锁，而其他线程拥有这个锁，这个线程就会暂停，直到锁被释放为止。如果这个锁永远也不释放，那么这个线程会永久停止。

不论是对I/O阻塞还是对锁阻塞，都不会释放线程已经拥有的锁。对于I/O阻塞，这不是个大问题，因为无非有两种情况：最后I/O终将不再阻塞而线程将继续执行；或者将抛出一个`IOException`异常，然后线程退出这个同步块或方法，并释放它的锁。不过，如果一个线程由于没有得到一个锁而阻塞，将永远不会放弃它自己有的锁。如果一个线程等待第二个线程拥有的锁，而第二个线程等待第一个线程拥有的锁，就会导致死锁。

## 放弃

要让线程放弃控制权，第二种方式是显式地放弃。为此线程可以通过调用`Thread.yield()`静态方法来做到。这将通知虚拟机，如果有另一个线程准备运行，可以运行该线程。有些虚拟机（特别是在实时操作系统上）会忽略这个提示。

在放弃之前，线程应当确保它或与它关联的`Runnable`对象处于一致状态，可以由其他对象使用。放弃并不会释放这个线程拥有的锁。因此，在理想情况下，在线程放弃时不应当做任何同步。一个线程放弃时，如果等待运行的其他线程都是因为需要这个线程所拥有的同步资源而阻塞，那么这些线程将不能运行。实际上，控制权将回到唯一可以运行的线程，即刚刚放弃的这个线程，这很大程度上失去了放弃的意义。

在实际中让线程放弃非常简单。如果线程的`run()`方法只包含一个无限循环，那么只要在循环的末尾加一个`Thread.yield()`调用。例如：

```
public void run() {
  while (true) {
    // 完成线程的工作...
    Thread.yield();
  }
}
```

这会使其他有相同优先级的线程有机会运行。

如果每次循环迭代都要花费很多时间，你可能希望在代码的其余部分散布更多的

Thread.yield()调用。在没有必要放弃的情况下，这种防范措施效果不甚明显。

### 休眠

休眠是更有力的放弃方式。放弃只是表示线程愿意暂停，让其他有相同优先级的线程有机会运行，而进入休眠的线程有所不同，不管有没有其他线程准备运行，休眠线程都会暂停。这样一来，不只是其他有相同优先级的线程得到机会，还会给较低优先级的线程一个运行的机会。不过，进入休眠的线程仍然拥有它已经获得的所有锁。因此，其他需要相同锁的线程会阻塞，即使CPU可用。所以要避免在同步方法或块内让线程休眠。

通过调用以下两个重载的Thread.sleep()静态方法之一，线程可以进入休眠。第一个方法接受要休眠的毫秒数作为参数。第二个接受毫秒数和毫微秒数。

```
public static void sleep(long milliseconds) throws InterruptedException
public static void sleep(long milliseconds, int nanoseconds)
    throws InterruptedException
```

虽然多数现代计算机时钟至少有接近毫秒级的精确度，但精确度达到毫微秒级的极少。不能保证在任何虚拟机上都能将实际的休眠时间控制在毫微秒甚至毫秒级。如果本地硬件不支持这个精度，休眠时间将舍入为可测量的最接近的值。例如：下面的run()方法尝试每5分钟加载一个页面，如果失败，就向Web管理员发email提醒这个问题：

```
public void run() {
  while (true) {
    if (!getPage("http://www.ibiblio.org/")) {
        mailError("webmaster@ibiblio.org");
    }
    try {
        Thread.sleep(300000); // 300,000毫秒 == 5分钟
    } catch (InterruptedException ex) {
        break;
    }
  }
}
```

线程不能绝对保证一定会休眠所期望的那么长时间。有时，在请求唤醒呼叫之后过一段时间线程才会真正唤醒，因为VM正在忙于做其他事情。也可能时间还没有到，但有其他线程完成了一些操作而唤醒了休眠的线程。一般情况下，这是通过调用休眠线程的interrupt()方法来实现的。

```
public void interrupt()
```

有些情况下线程与Thread对象之间的区别很重要，这里就是如此。一个线程在休眠，并不意味着其他醒着的线程不能处理这个线程相应的Thread对象（通过它的方法和字段）。具体地，另一个线程可以调用休眠Thread对象的interrupt()方法，这

会让休眠中的线程得到一个InterruptedException异常。在此之后，这个线程会被唤醒并正常执行，至少在再次进入休眠之前会正常执行。在前面的例子中，使用了一个InterruptedException来结束一个线程，否则这个线程会永远运行下去。抛出InterruptedException时，就会打破无限循环，run()方法结束，相应地线程将会停止。用户选择菜单中的Exit或指示希望程序退出时，用户界面线程会调用这个线程的interrupt()方法。

---

**警告：** 如果一个线程对一个I/O操作（如读或写）阻塞，中断这个线程的效果很大程度上依赖于具体的平台。通常这将是一个什么都不做的操作。也就是说，线程继续阻塞。在Solaris上，read()或write()方法可能抛出一个InterruptedIOException，这是IOException的一个子类。不过，其他平台上通常不会这样，而且在Solaris上这也不适用于所有流类。如果你的程序体系结构需要可中断的I/O，就应当认真考虑使用第11章讨论的非阻塞I/O，而不是流。与流不同，缓冲区和通道都明确设计为支持读/写阻塞时中断。

---

## 连接线程

一个线程可能需要另一个线程的结果，这是很常见的。例如，Web浏览器在一个线程中加载HTML页面，它可能要生成一个单独的线程来获取页面中嵌入的各个图片。如果IMG元素没有指定HEIGHT和WIDTH属性，主线程在结束页面的显示之前，可能必须等待所有图片加载完毕。Java提供了三个join()方法，允许一个线程在继续执行前等待另一个线程结束。这些方法是：

```
public final void join() throws InterruptedException
public final void join(long milliseconds) throws InterruptedException
public final void join(long milliseconds, int nanoseconds)
    throws InterruptedException
```

第一种方法无限等待被连接（joined）的线程结束。后面两个方法会等待指定的一段时间，然后会继续执行，即使被连接的线程还没有结束。与sleep()方法一样，不能保证毫微秒级的精度。

连接线程（即调用join()方法的线程）等待被连接的线程（也就是说，调用的是这个线程的join()方法）结束。例如，考虑下面这段代码。我们希望找到一个随机double数组中的最小数、最大数和中间数。用有序数组能更快地完成。我们生成一个新线程对数组排序，然后连接到这个线程等待它的结果。只有当它结束时，才会读取所需的值。

```
double[] array = new double[10000];                    // 1
for (int i = 0; i < array.length; i++) {               // 2
  array[i] = Math.random();                            // 3
```

```
}                                                          // 4
SortThread t = new SortThread(array);                      // 5
t.start();                                                 // 6
try {                                                      // 7
  t.join();                                                // 8
  System.out.println("Minimum: " + array[0]);               // 9
  System.out.println("Median: " + array[array.length/2]);   // 10
  System.out.println("Maximum: " + array[array.length-1]);  // 11
} catch (InterruptedException ex) {                        // 12
}                                                          // 13
```

前面第1行到第4行先执行，用随机数填充数组。然后第5行创建一个新的SortThread。第6行启动这个线程对数组排序。在找到数组的最小数、中间数和最大数之前，需要等待排序线程结束。因此，第8行将当前线程连接到排序线程。至此，执行这些代码的线程会停止执行。它会等待排序线程结束执行。直到排序线程结束运行并撤销之后，第9行到第11行才会获取最小、中间和最大值。注意在这里没有引用暂停的线程。并不是调用这个Thread对象的join()方法；它没有作为参数传递给该方法，而只是隐式地作为当前线程存在。如果是在程序main()方法的正常控制流中，可能任何地方都没有指向这个线程的Thread变量。

连接到另一个线程的线程可以被中断，如果有其他线程调用其interrupt()方法，它就会像休眠线程一样被中断。线程将这个调用作为一个InterruptedException异常。此后，它会从捕获这个异常的catch块开始正常执行。在前面的例子中，如果线程被中断，它将跳过最小值、中间值和最大值的计算，因为如果排序线程在结束前被中断，这些值是不可用的。

可以使用join()修复示例3-4。示例3-4的问题是，main()方法的速度会超过生成结果的线程（即main()方法要使用的结果）。通过在使用各个线程的结果前连接各个线程，就能容易地解决这个问题。如示例3-12所示。

示例3-12：通过连接生成所需结果的线程，避免竞态条件

```
import javax.xml.bind.DatatypeConverter;

public class JoinDigestUserInterface {

  public static void main(String[] args) {

    ReturnDigest[] digestThreads = new ReturnDigest[args.length];

    for (int i = 0; i < args.length; i++) {
      // 计算摘要
      digestThreads[i] = new ReturnDigest(args[i]);
      digestThreads[i].start();
    }

    for (int i = 0; i < args.length; i++) {
```

```
        try {
          digestThreads[i].join();
          // 现在显示结果
          StringBuffer result = new StringBuffer(args[i]);
          result.append(": ");
          byte[] digest = digestThreads[i].getDigest();
          result.append(DatatypeConverter.printHexBinary(digest));
          System.out.println(result);
        } catch (InterruptedException ex) {
          System.err.println("Thread Interrupted before completion");
        }
      }
    }
}
```

由于示例3-12以启动线程同样的顺序连接这些线程，这样修复也有一个副作用，它会按构造线程所用的参数的顺序显示输出，而不是按线程结束的顺序。这种修改不会让程序变慢，但有时如果你希望线程一旦结束就获得结果，而不是等待其他无关线程先结束，这可能会是个问题。

---

**提示：** 如今，连接线程可能没有Java 5之前那么重要。具体来讲，很多原来需要join()的设计现在用Executor和Future可以更容易地实现。

---

## 等待一个对象

线程可以等待（wait）一个它锁定的对象。在等待时，它会释放这个对象的锁并暂停，直到它得到其他线程的通知。另一个线程以某种方式修改这个对象，通知等待对象的线程，然后继续执行。这与连接线程不同，并不要求等待线程和通知线程在另一个线程继续前必须结束。等待会暂停执行，直到一个对象或资源达到某种状态。连接也会暂停执行，不过是直到一个线程结束。

在暂停线程的方法中，等待一个对象的做法并不太出名。这是因为它不涉及Thread类的任何方法。实际上，要等待某个特定的对象，希望暂停的线程首先必须使用synchronized获得这个对象的锁，然后调用这个对象的三个重载wait()方法之一：

```
public final void wait() throws InterruptedException
public final void wait(long milliseconds) throws InterruptedException
public final void wait(long milliseconds, int nanoseconds)
throws InterruptedException
```

这些方法不在Thread类中，而是在java.lang.Object类中。因此，可以在任何类的任何对象上调用这些方法。调用其中一个方法时，调用它的线程会释放所等待的对象的锁（但不会释放它拥有的其他对象的锁），并进入休眠。线程会保持休眠，直到发生以下3种情况之一：

- 时间到期。
- 线程被中断。
- 对象得到通知。

超时时间（timeout）与sleep()和join()方法中的超时时间相同，即线程经过指定的一段时间后（在本地硬件时钟的精度范围内）会唤醒。当时间到期时，线程会从紧挨着wait()调用之后的语句继续执行。不过，如果线程不能立即重新获得所等待的对象的锁，它可能仍要阻塞一段时间。

中断（Interruption）与sleep()和join()的工作方式相同：其他线程调用这个线程的interrupt()方法。这将导致一个InterruptedException异常，并在捕获这个异常的catch块内继续执行。不过，在抛出异常前线程要重新获得所等待对象的锁，所以调用interrupt()方法后，该线程可能仍要阻塞一段时间。

第三种可能的方法是通知（notification），这是一个新方法。在其他线程在这个线程所等待的对象上调用notify()或notifyAll()方法时，就会发生通知。这两个方法都在java.lang.Object类中：

```
public final void notify()
public final void notifyAll()
```

这两个方法都必须在线程所等待的对象上调用，而不是在Thread本身调用。在通知一个对象之前，线程必须首先使用同步方法或块获得这个对象的锁。notify()基本上随机地从等待这个对象的线程列表中选择一个线程，并将它唤醒。notifyAll()方法会唤醒等待指定对象的每一个线程。

一旦等待线程得到通知，它就试图重新获得所等待对象的锁。如果成功，就会从紧接着wait()调用之后的语句继续执行。如果失败，它就会对这个对象阻塞，直到可以得到锁，然后继续执行紧接着wait()调用之后的语句。

例如，假设一个线程正在从网络连接中读取一个JAR归档文件。这个归档文件中第一项是清单文件。另一个线程可能对这个清单文件的内容感兴趣，即使归档文件的其余部分尚不可用。对清单文件感兴趣的线程会创建一个定制的ManifestFile对象，将这个对象的引用传递给将要读取JAR归档文件的线程，并等待这个对象。读取归档文件的线程首先用流中的项填写ManifestFile，然后通知ManifestFile，再继续读取JAR归档文件的其余部分。当阅读器线程通知ManifestFile时，原来的线程会被唤醒，按其计划处理现在已经完全准备就绪的ManifestFile对象。第一个线程的工作方式如下：

```
ManifestFile m = new ManifestFile();
JarThread t = new JarThread(m, in);
```

```
synchronized (m) {
  t.start();
  try {
    m.wait();
    // 处理清单文件...
  } catch (InterruptedException ex) {
    // 处理异常...
  }
}
```

JarThread类工作如下：

```
ManifestFile theManifest;
InputStream in;

public JarThread(Manifest m, InputStream in) {
  theManifest = m;
  this.in= in;
}

@Override
public void run() {
  synchronized (theManifest) {
    // 从流in读入清单文件...
    theManifest.notify();
  }
  // 读取流的其余部分...
}
```

当多个线程希望等待同一个对象时，等待和通知会更为常用。例如，一个线程可能在读取一个Web服务器日志文件，文件中每一行包含要处理的一项。读取时每一行放在一个java.util.List中。多个线程在添加项时会等待这个List来处理这些项。每次添加一项时，会使用notifyAll()方法通知等待线程。如果有多个线程在等待这个对象，首选notifyAll()，因为没有办法选择要通知哪个线程。当等待一个对象的所有线程得到通知时，这些线程都会被唤醒，并试图获得这个对象的锁。不过，只有一个线程能立即成功。得到锁的这个线程将继续执行。其余线程会阻塞，直到第一个线程释放这个锁。如果多个线程等待同一个对象，那么轮到最后一个线程获得这个对象的锁并继续执行时，可能已经过去了相当长的时间。在这段时间内，这个线程等待的对象完全有可能再次置于不可接受的状态。因此，一般要将wait()调用放在检查当前对象状态的循环中。不要假定因为线程得到了通知，对象现在就处于正确的状态。要保证对象进入正确的状态之后，再也不会进入不正确的状态，如果无法保证这一点，就要显式地进行检查。例如，下面显示了等待日志文件项的客户端线程：

```
private List<String> entries;

public void processEntry() {
```

```
    synchronized (entries) { //必须对等待的对象同步
      while (entries.isEmpty()) {
        try {
          entries.wait();
          //停止等待，因为entries.size()变为非0，
          //但是我们不知道它仍然是非0，
          //所以再次通过循环检查它现在的状态
        } catch (InterruptedException ex) {
          //如果被中断，则最后一项已经处理过，所以返回
          return;
        }
      }
      String entry = entries.remove(entries.size()-1);
      // 处理这一项...
    }
  }
```

下面给出的代码会读取日志文件，并将项添加到列表中：

```
  public void readLogFile() {
    while (true) {
      String entry = log.getNextEntry();
      if (entry == null) {
        // 没有更多项要添加到列表，
        //所以中断所有仍在等待的线程。
        // 否则它们将永远等待下去
        for (Thread thread : threads) thread.interrupt();
        break;
      }
      synchronized (entries) {
        entries.add(0, entry);
        entries.notifyAll();
      }
    }
  }
```

## 结束

线程要以合理的方式放弃CPU控制权，最后一种方法是结束（finishing）。当run()方法返回时，线程将撤销，其他线程可以接管CPU。在网络应用程序中，包装一个阻塞操作的线程往往会这样做，例如从服务器下载一个文件，这样应用程序的其他部分就不会被阻塞。

另一方面，如果run()方法太简单，总是很快就结束，而不会阻塞，那就存在一个很实际的问题：到底有没有必要生成一个线程。虚拟机在建立和撤销线程时会有很大的开销。如果线程会在极短的时间内结束，那么使用一次简单的方法调用而不是单独的线程可能会结束得更快。

# 线程池和Executor

向程序添加多个线程会极大地提升性能，尤其是I/O受限的程序，如大多数网络程序。不过，线程自身也存在开销。启动一个线程时，以及线程撤销后进行清理时，都需要虚拟机做大量工作，尤其是生成数百个线程的程序，即使对中低吞吐量的网络服务器而言，这种情况也很常见。即使线程能很快结束，这也会加重垃圾回收器或VM其他部分的负担而影响性能，就好像每分钟分配几千个任何其他类型的对象。更重要的是，在运行线程之间切换也会带来开销。如果线程自然阻塞（例如等待网络数据），那么没有什么真正的影响，但如果线程是CPU受限的，倘若能避免线程间的大量切换，整个任务可能会更快地结束。最后，也是最重要的，虽然线程有助于更高效地利用计算机有限的CPU资源，但所能提供的资源毕竟是有限的。一旦已经生成足够多的线程来使用计算机所有可用的空闲时间，那么再生成更多线程只会将MIPS和内存浪费在线程管理上。

利用java.util.concurrent中的Executors类，可以非常容易地建立线程池。只需要将各个任务作为一个Runnable对象提交给这个线程池，你就会得到一个Future对象，可以用来检查任务的进度。下面来看一个例子。假设你希望使用一个java.util.zip.GZIPOutputStream对当前目录中的每一个文件完成gzip压缩。这是一个过滤器流，会压缩它写的所有数据。

一方面，这是一个有大量I/O的操作，因为所有文件都必须进行读/写。另一方面，数据压缩是一个“CPU密集”度很高的操作，所以你不希望同时运行太多线程。这是使用线程池的大好机会。每个客户端线程将压缩文件，同时主程序将确定要压缩哪个文件。在这个例子中，主程序的速度很可能会远远超过压缩线程，因为它要做的所有工作就是列出目录中的文件。因此，毫无疑问，首先要填充线程池，然后启动池中压缩文件的线程。不过，为了让这个示例尽可能具有地一般性，我们将让主程序与压缩线程并行运行。

示例3-13显示了GZipRunnable类。它包含一个字段来标识要压缩的文件。run()方法会压缩这个文件并返回。

示例3-13：GZipRunnable类

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

public class GZipRunnable implements Runnable {

  private final File input;

  public GZipRunnable(File input) {
    this.input = input;
  }
```

```
  @Override
  public void run() {
    // 不压缩已经压缩的文件
    if (!input.getName().endsWith(".gz")) {
      File output = new File(input.getParent(), input.getName() + ".gz");
      if (!output.exists()) { // 不覆盖已经存在的文件
        try ( // with resources,  要求使用Java 7
          InputStream in = new BufferedInputStream(new FileInputStream(input));
          OutputStream out = new BufferedOutputStream(
          new GZIPOutputStream(
            new FileOutputStream(output)));
        ) {
            int b;
            while ((b = in.read()) != -1) out.write(b);
            out.flush();
        } catch (IOException ex) {
          System.err.println(ex);
        }
      }
    }
  }
}
```

注意GZipRunnable中可以使用Java 7的try-with-resources语句。输入和输出流都在try块的最前面声明，并在try块的末尾自动关闭。

还要注意输入和输出的缓冲。这对于I/O有限的应用的性能非常重要，在网络程序中尤其重要。最坏情况下，缓冲对性能没有影响，但在最好情况下，会让执行速度提高一个数量级。

示例3-14是主程序。它构造了线程池，线程数固定为4，并迭代处理命令行中列出的所有文件和目录。这些文件以及这些目录中的文件将用来构建一个GZipRunnable。这个runnable提交到线程池，最终由这4个线程之一处理。

示例3-14：GZipThread用户接口类

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

public class GZipAllFiles {

public final static int THREAD_COUNT = 4;

public static void main(String[] args) {

  ExecutorService pool = Executors.newFixedThreadPool(THREAD_COUNT);

  for (String filename : args) {
    File f = new File(filename);
    if (f.exists()) {
      if (f.isDirectory()) {
        File[] files = f.listFiles();
```

```
        for (int i = 0; i < files.length; i++) {
          if (!files[i].isDirectory()) { // 不递归处理目录
            Runnable task = new GZipRunnable(files[i]);
            pool.submit(task);
            }
          }
        } else {
          Runnable task = new GZipRunnable(f);
          pool.submit(task);
        }
      }
    }

    pool.shutdown();
  }
}
```

一旦将所有文件增加到这个池，就可以调用pool.shutdown()。这有可能在还有工作要完成的情况下发生。这个方法不会中止等待的工作。它只是通知线程池再没有更多任务需要增加到它的内部队列，而且一旦完成所有等待的工作，就应当关闭。

你要写的网络程序可能有大量线程，对于这些网络程序来说，很少像这样关闭，因为它有一个确定的终点：即所有文件都得到处理时。大多数网络服务器会无限继续，直到通过一个管理界面将其关闭。在这些情况下，你可能希望调用shutdownNow()，中止当前处理中的任务，并忽略所有等待的任务。

# 第4章

# Internet地址

连接到Internet的设备称为节点（node）。计算机节点称为主机（host）。每个节点或主机都由至少一个唯一的数来标识，这称为Internet地址或IP地址。目前大多数IP地址是四字节长，这被称作IPv4地址。不过，一小部分IP地址是16字节长（而且这种地址的数量正在增加），这被称作IPv6地址（4和6指Internet协议的版本，不是地址中的字节数）。IPv4和IPv6地址都是字节的有序序列，和数组一样。它们不是数，它们的顺序也不具有任何可预测或有用的意义。

IPv4地址一般写为四个无符号字节，每字节范围从0到255，最高字节在前。为方便人们查看，各字节用点号分隔。例如，*login.ibiblio.org*的地址是152.19.134.132。这称为点分四段（dotted quad）格式。

IPv6地址通常写为冒号分隔的8个区块，每个区块是4个十六进制数字。例如，写这本书时，*www.hamiltonweather.tk*的地址是*2400:cb00:2048:0001:0000:0000:6ca2:c665*。前导的0不需要写出。因此，*www.hamiltonweather.tk*的地址可以写为*2400:cb00:2048:1:0:0:6ca2:c665*。两个冒号表示多个0区块，但每个地址中双冒号至多出现一次。例如，*2001:4860:4860:0000:0000:0000:0000:8888*可以写为紧缩的*2001:4860:4860::8888*。在IPv6和IPv4的混合网络中，IPv6地址的最后4字节有时写为IPv4的点分四段地址。例如，*FEDC:BA98:7654:3210:FEDC:BA98:7654:3210*可以写为*FEDC:BA98:7654:3210:FEDC:BA98:118.84.50.16*。

IP地址对于计算机来说很不错，但对人来说是个问题，人们很难记忆长的数字。在20世纪50年代，G. A. Miller发现大多数人可以记住每个数中大约7个数字，有些人可以记住多达9个，而另外一些人只能记住5个。有关的更多信息参见《Psychological Review》卷63第81～97页上的《The Magic Number Seven, Plus or Minus Two: Some Limits on Our

Capacity for Processing Information》。这就是为什么电话号码被分成多个部分，每个部分包括3或4个数字，另外还有一个3个数字的区号。很明显，IP地址多达12个十进制数字已经超出了大多数人的记忆能力。我能记住大概两个IP地址，前提是我每天都用它们，而且这两个地址处于同一个子网内。

为避免到处携带记满IP地址的通讯录，Internet的设计者发明了域名系统（Domain Name System，DNS）。DNS将人们可以记忆的主机名（如*login.ibiblio.org*）与计算机可以记忆的IP地址（如*152.19.134.132*）关联在一起。服务器通常至少有一个主机名。客户端往往有一个主机名，但也可能没有，特别是有些客户端的IP地址会在启动时动态指定。

---

**提示：**通俗地讲，人们通常使用“Internet地址”表示一个主机名（或者甚至是电子邮件地址，或者完全URL）。在关于网络编程的书中，准确区分地址和主机名非常重要。在本书中，地址总是数字IP地址，而不是人类可读的主机名。

---

有些机器有多个名。例如，*www.beand.com*和*xom.nu*实际上是同一台Linux主机。名*www.beand.com*实际上指示一个Web网站而不是一个特定的机器。在过去，当这个Web网站从一台机器移到另一台机器时，这个名字将重新指派给这台新机器，使它永远指向网站的当前服务器。通过这种方法，就不会因为网站迁移到一个新主机而要更新Web上的URL。一些常见的名字如*www*和*news*通常就是提供这些服务的机器的别名。例如，*news.speakeasy.net*是我的ISP的新闻服务器的别名。由于服务器可能随着时间而改变，别名可能跟着服务转移。

有时，一个名会映射到多个IP地址，这时就要由DNS服务器负责随机选择一台机器来响应各个请求。这个特性在业务流量非常大的Web网站经常使用，它将负载分摊到多个系统上。例如，*www.oreilly.com*实际上是两台机器，一台位于208.201.239.100，一台位于208.201.239.101。

每台连接到Internet的计算机都应当能访问一个称为域名服务器（domain name server）的机器，它通常是一个运行特殊DNS软件的UNIX主机，这种软件了解不同主机名和IP地址之间的映射。大多数域名服务器只知道其本地网络上主机的地址，以及其他网站中一些域名服务器的地址。如果客户端请求本地域之外一个机器的地址，本地域名服务器就会询问远程位置的域名服务器，再将答案转发给请求者。

大多数情况下，可以使用主机名，而让DNS处理向IP地址的转换。只要能连接到一个域名服务器，就不需要担心在你的机器、本地域名服务器和Internet其余部分之间传递主机名和地址的有关细节。不过，你至少要能访问一个域名服务器才能使用本章和本书其余大部分的例子。这些程序在独立计算机上无法运行。你的机器必须连接到Internet。

# InetAddress类

java.net.InetAddress类是Java对IP地址（包括IPv4和IPv6）的高层表示。大多数其他网络类都要用到这个类，包括Socket、ServerSocket、URL、DatagramSocket、DatagramPacket等。一般地讲，它包括一个主机名和一个IP地址。

## 创建新的InetAddress对象

InetAddress类没有公共构造函数。实际上，InetAddress有一些静态工厂方法，可以连接到DNS服务器来解析主机名。最常用的是InetAddress.getByName()。例如，可以如下查找*www.oreilly.com*：

```
InetAddress address = InetAddress.getByName("www.oreilly.com");
```

这个方法并不只是设置InetAddress类中的一个私有String字段。实际上它会建立与本地DNS服务器的一个连接，来查找名字和数字地址（如果你之前查找过这个主机，这个信息可能会在本地缓存，如果是这样，就不需要再建立网络连接）。如果DNS服务器找不到这个地址，这个方法会抛出一个UnknownHostException异常，这是IOException的一个子类。

示例4-1展示了一个完整的程序，它为*www.oreilly.com*创建一个InetAddress对象，这里包括所有必要的导入和异常处理。

示例4-1：显示www.oreilly.com 地址的程序

```
import java.net.*;

public class OReillyByName {

  public static void main (String[] args) {
    try {
      InetAddress address = InetAddress.getByName("www.oreilly.com");
      System.out.println(address);
    } catch (UnknownHostException ex) {
      System.out.println("Could not find www.oreilly.com");
    }
  }
}
```

结果如下：

```
% java OReillyByName
www.oreilly.com/208.201.239.36
```

还可以按IP地址反向查找。例如，如果希望得到地址208.201.239.100的主机名，可以向InetAddress.getByName()传入一个点分四段地址：

```
InetAddress address = InetAddress.getByName("208.201.239.100");
System.out.println(address.getHostName());
```

如果你查找的地址没有相应的主机名，getHostName()就会返回你提供的点分四段地址。

之前我提到过*www.oreilly.com*实际上有两个地址。getHostName()返回哪一个地址是不确定的。如果出于某种原因你需要得到一个主机的所有地址，可以调用getAllByName()，它会返回一个数组：

```
try {
  InetAddress[] addresses = InetAddress.getAllByName("www.oreilly.com");
  for (InetAddress address : addresses) {
    System.out.println(address);
  }
} catch (UnknownHostException ex) {
  System.out.println("Could not find www.oreilly.com");
}
```

最后，getLocalHost()方法会为运行这个代码的主机返回一个InetAddress对象：

```
InetAddress me = InetAddress.getLocalHost();
```

这个方法尝试连接DNS来得到一个真正的主机名和IP地址，如“elharo.laptop.corp.com”和“192.1.254.68”；不过如果失败，它就会返回回送地址，即主机名“localhost”和点分四段地址“127.0.0.1”。

示例4-2显示了运行这个代码的机器的地址。

示例4-2：查找本地机器的地址

```
import java.net.*;

public class MyAddress {

  public static void main (String[] args) {
    try {
      InetAddress address = InetAddress.getLocalHost();
      System.out.println(address);
    } catch (UnknownHostException ex) {
      System.out.println("Could not find this computer's address.");
    }
  }
}
```

输出如下（我是在*titan.oit.unc.edu*上运行这个程序）：

```
% java MyAddress
titan.oit.unc.edu/152.2.22.14
```

你可能会看到完全限定名如*titan.oit.unc.edu*，也可能会看到部分名如titan，这取决于本地DSN服务器为本地域中的主机返回什么结果。如果没有连接到Internet，系统也没有固定IP地址或域名，可能会看到域名为*localhost*，IP地址为127.0.0.1。

如果你知道一个数字地址，可以由这个地址创建一个`InetAddress`对象，而不必使用`InetAddress.getByAddress()`与DNS交互。这个方法可以为不存在或者无法解析的主机创建地址：

```
public static InetAddress getByAddress(byte[] addr) throws UnknownHostException
public static InetAddress getByAddress(String hostname, byte[] addr)
    throws UnknownHostException
```

第一个`InetAddress.getByAddress()`工厂方法用一个IP地址（而没有主机名）创建一个`InetAddress`对象。第二个`InetAddress.getByAddress()`方法使用一个IP地址和一个主机名创建`InetAddress`对象。例如，下面这个代码段会为107.23.216.196创建一个`InetAddress`：

```
byte[] address = {107, 23, (byte) 216, (byte) 196};
InetAddress lessWrong = InetAddress.getByAddress(address);
InetAddress lessWrongWithname = InetAddress.getByAddress(
    "lesswrong.com", address);
```

需要说明，它必须把两个大数字转换为字节。

与其他工厂方法不同，这两个方法不能保证这个主机一定存在，或者主机名能正确地映射到IP地址。只有当作为`address`参数传入的字节数组大小不合法时（不是4字节，也不是16字节），这两个方法才会抛出一个`UnknownHostException`异常。如果域名服务器不可用，或者可能有不正确的信息，这会很有用。例如，我的主干网中所有计算机、打印机或路由器都没有注册任何DNS服务器。因为我记不住为哪些系统分配了哪些地址，所以我写了一个简单的程序，尝试依次连接所有254个可能的本地地址，来看哪些地址是活动的（这样花费的时间是把所有地址写在一张纸上所需时间的10倍）。

## 缓存

由于DNS查找的开销可能相当大（如果请求需要经过多个中间服务器，或者尝试解析一个不可达的主机，这大约需要几秒的时间），所以`InetAddress`类会缓存查找的结果。一旦得到一个给定主机的地址，就不会再次查找，即使你为同一个主机创建一个新的`InetAddress`对象，也不会再次查找地址。只要在程序运行期间IP地址没有改变，这就没有问题。

负面结果（即主机未找到错误）稍有些问题。有可能刚开始尝试解析一个主机时失败，但随后再次尝试时解析会成功，这种情况并不少见。由于从远程DNS服务器发来的信

息还在传输中，第一次尝试超时。然后这个地址到达本地服务器，所以下一次请求时可用。出于这个原因，Java对于不成功的DNS查询只缓存10秒。

这些时间可以用系统属性networkaddress.cache.ttl和networkaddress.cache.negative.ttl来控制。其中第一个属性networkaddress.cache.ttl指定了成功的DNS查找结果在Java缓存中保留的时间（秒数），networkaddress.cache.negative.ttl指定了不成功的查找结果缓存的时间（秒数）。在这些时限内，再次尝试查找相同的主机会返回相同的值。-1解释为“永不过期”。

除了在InetAddress类中的本地缓存，本地主机、本地域名服务器和Internet上其他地方的DNS服务器也会缓存各种查询的结果。对此，Java没有提供有关的控制方法。因此，在Internet上传播IP地址改变的信息可能要花费几个小时。在此期间，你的程序可能会遇到各种异常，包括UnknownHostException、NoRouteToHostException和ConnectException异常，这取决于对DNS所做的改变。

### 按IP地址查找

调用getByName()并提供一个IP地址串作为参数时，会为所请求的IP地址创建一个InetAddress对象，而不检查DNS。这说明，可能会为实际上不存在也无法连接的主机创建InetAddress对象。由包含IP地址的字符串来创建InetAddress对象时，这个对象的主机名初始设置为这个字符串。只有当请求主机名时（显式地通过getHostName()请求），才会真正完成主机名的DNS查找。从点分四段地址208.201.239.37确定*www.oreilly.com*时就采用了这种方式。如果请求主机名并最终完成了一个DNS查找，但是指定IP地址的主机无法找到，那么主机名会保持为最初的点分四段字符串。不过，不会抛出UnknownHostException异常。

主机名要比IP地址稳定得多。有些服务多年以来一直使用同一个主机名，但IP地址更换了很多次。如果要在使用主机名（如*www.oreilly.com*）或使用IP地址（如208.201.239.37）之间做出选择，一定要选择主机名。只有当主机名不可用时才使用IP地址。

### 安全性问题

从主机名创建一个新的InetAddress对象被认为是一个潜在的不安全操作，因为这需要一个DNS查找。在默认安全管理器控制下的不可信applet只允许获得它的初始主机（其代码基）的IP地址，这可能是本地主机。不允许不可信代码由任何其他主机名创建InetAddress对象。不论代码使用InetAddress.getByName()方法、InetAddress.getAllByName()方法、InetAddress.getLocalHost()方法，还是其他方法，都是如此。不可信代码可以由字符串形式的IP地址构造InetAddress对象，但不会为这样的地址完成DNS查找。

由于禁止与代码基之外的主机建立网络连接，不可信的代码不允许对第三方主机完成任意的DNS查找。任意的DNS查找会打开一个隐藏的通道，通过它，程序可以与第三方主机对话。例如，假设一个从*www.bigisp.com*下载的applet希望将消息“macfaq.dialup.cloud9.net is vulnerable”发送给*crackersinc.com*。它只需要请求*macfaq.dialup.cloud9.net.is.vulnerable.crackersinc.com*的DNS信息。为了解析这个主机名，这个applet会联系本地DNS服务器。本地DNS服务器会联系位于*crackersinc.com*的DNS服务器。尽管这些主机不存在，但黑客可以查看*crackersinc.com*的DNS错误日志来获取这个消息。如果再结合压缩、纠错、加密，以及将电子邮件消息发送给一个第四方网站的定制DNS服务器，这个机制还可以复杂得多，但这个版本已经足以证明上述观点。由于任意DNS查找会泄漏信息，所以要禁止任意的DNS查找。

不可信代码允许调用`InetAddress.getLocalHost()`。不过，在这种环境下，`getLocalHost()`总是返回主机名*localhost/127.0.0.1*。禁止applet找出真正的主机名和地址的原因在于，运行applet的计算机可能故意隐藏在防火墙的后面。在这种情况下，applet不应作为通道来获得Web服务器还没有的信息。

与所有安全性检查一样，禁止DNS解析可以对可信代码放宽要求。要测试一个主机能否解析，所用的特定SecurityManager方法是checkConnect()：

```
public void checkConnect(String hostname, int port)
```

当port参数为−1时，这个方法检查能否调用DNS解析指定的hostname。（如果port参数大于−1，这个方法检查是否允许在指定端口对指定主机建立连接）。hostname参数可以是主机名（如*www.oreilly.com*），也可以是点分四段IP地址（如208.201.239.37），或者还可以是十六进制IPv6地址如FEDC::DC:0:7076:10。

## 获取方法

`InetAddress`包含4个获取方法，可以将主机名作为字符串返回，将IP地址返回为字符串和字节数组：

```
public String getHostName()
public String getCanonicalHostName()
public byte[] getAddress()
public String getHostAddress()
```

没有对应的`setHostName()`和`setAddress()`方法，这说明`java.net`之外的包无法在后台改变`InetAddress`对象的字段。这使得`InetAdddress`不可变，因此是线程安全的。

`getHostName()`方法返回一个`String`，其中包含主机的名字，以及这个`InetAddress`对象表示的IP地址。如果这台机器没有主机名或者安全管理器阻止确定主机名，就会返回点

分四段格式的数字IP地址。例如：

```
InetAddress machine = InetAddress.getLocalHost();
String localhost = machine.getHostName();
```

getCanonicalHostName()方法也类似，不过在与DNS联系方面更积极一些。getHostName()只是在不知道主机名时才会联系DNS，而getCanonicalHostName()知道主机名时也会联系DNS，可能会替换原来缓存的主机名。例如：

```
InetAddress machine = InetAddress.getLocalHost();
String localhost = machine.getCanonicalHostName();
```

如果开始只有一个占分四段IP地址而没有主机名，getCanonicalHostName()方法尤其有用。示例4-3首先使用InetAddress.getByName()，然后对得到的对象应用getCanonicalHostName()，可以把点分四段地址*208.201.239.37*转换为一个主机名。

示例4-3：给定地址，找出主机名

```
import java.net.*;

public class ReverseTest {

public static void main (String[] args) throws UnknownHostException {
  InetAddress ia = InetAddress.getByName("208.201.239.100");
    System.out.println(ia.getCanonicalHostName());
  }
}
```

结果如下：

```
% java ReverseTest
oreilly.com
```

getHostAddress()方法返回一个字符串，其中包含点分四段格式的IP地址。示例4-4使用这个方法按通常的格式显示本地机器的IP地址。

示例4-4：找到本地机器的IP地址

```
import java.net.*;

public class MyAddress {

  public static void main(String[] args) {
    try {
      InetAddress me = InetAddress.getLocalHost();
      String dottedQuad = me.getHostAddress();
      System.out.println("My address is " + dottedQuad);
    } catch (UnknownHostException ex) {
      System.out.println("I'm sorry. I don't know my own address.");
    }
}
```

}

结果如下：

```
% java MyAddress
My address is 152.2.22.14.
```

当然，具体的输出依赖于运行程序的机器。

如果希望知道一台机器的IP地址（很少这样做），可以使用getAddress()方法，它会以网络字节顺序将IP地址作为一个字节数组返回。最高字节（即地址的点分四段形式中的第一字节）是数组的第一字节，即数组的元素0。如果要考虑到IPv6地址，不要对这个数组的长度做任何假定。如果需要知道这个数组的长度，可以使用数组的length字段：

```
InetAddress me = InetAddress.getLocalHost();
byte[] address = me.getAddress();
```

返回的字节是无符号的，这会带来一个问题。与C不同，Java没有无符号字节这种基本数据类型。值大于127的字节会当作负数。因此，如果要对getAddress()返回的字节做任何处理，需要将字节提升为int，并做适当的调整。下面给出一种做法：

```
int unsignedByte = signedByte < 0 ? signedByte + 256 : signedByte;
```

这里，signedByte可能为正也可能为负。条件操作符?测试signedByte是否为负。如果为负，则为signedByte加上256使其成为正数。否则保持不变。signedByte会在完成加法操作之前自动提升为int，所以不存在这种回绕（wraparound）问题。

之所以要查看IP地址的原始字节，一个原因是想要确定地址的类型。测试getAddress()所返回数组的字节数可以确定处理的是IPv4还是IPv6地址，如示例4-5所示。

示例4-5：确定IP地址是IPv4还是IPv6

```
import java.net.*;

public class AddressTests {

  public static int getVersion(InetAddress ia) {
    byte[] address = ia.getAddress();
    if (address.length == 4) return 4;
    else if (address.length == 16) return 6;
    else return -1;
  }
}
```

## 地址类型

有些IP地址和地址模式有特殊的含义。例如，前面提到127.0.0.1是本地回送地址。

224.0.0.0到239.255.255.255范围内的IPv4地址是组播地址，可以同时发送到多个订购的主机。Java提供了10个方法来测试InetAddress对象是否符合其中某个标准：

```
public boolean isAnyLocalAddress()
public boolean isLoopbackAddress()
public boolean isLinkLocalAddress()
public boolean isSiteLocalAddress()
public boolean isMulticastAddress()
public boolean isMCGlobal()
public boolean isMCNodeLocal()
public boolean isMCLinkLocal()
public boolean isMCSiteLocal()
public boolean isMCOrgLocal()
```

如果地址是通配地址（wildcard address），isAnyLocalAddress()方法返回true，否则返回false。通配地址可以匹配本地系统中的任何地址。如果系统有多个网络接口（如一个系统有多个以太网卡，或者系统有一个以太网卡和一个802.11 WiFi接口），这会很重要。在IPv4中，通配地址是0.0.0.0。在IPv6中通配地址是0:0:0:0:0:0:0:0（又写作::）。

如果地址是回送地址（loopback address），isLoopbackAddress()方法返回true，否则返回false。回送地址直接在IP层连接同一台计算机，而不使用任何物理硬件。因此，通过连接回送地址，可以绕过可能有bug或不存在的以太网、PPP和其他驱动程序进行测试，这有助于隔离问题。连接回送地址与从系统中连接同一个系统的正常IP地址有所不同。在IPv4中，这个地址是127.0.0.1。在IPv6中，这个回送地址是0:0:0:0:0:0:0:1（又写作::1）。

如果地址是一个IPv6本地链接地址，isLinkLocalAddress()方法返回true，否则返回false。IPv6本地链接地址可以用于帮助IPv6网络实现自配置，与IPv4网络上的DHCP非常相似，但没有必要使用服务器。路由器不会把发送给本地链接地址的包转发到本地子网以外。所有本地链接地址都以8字节FE80:0000.0000:0000开头。后8字节用本地地址填充，这个地址通常从以太网卡生产商分配的以太网MAC地址复制。

如果地址是一个IPv6本地网站地址，isSiteLocalAddress()方法返回true，否则返回false。本地网站地址与本地链接地址相似，不过本地网站地址可以由路由器在网站或校园内转发，但不应转发到网站以外。本地网站地址以8字节FEC0:0000.0000:0000开头。后8字节用本地地址填充，这个地址通常从以太网卡生产商分配的以太网MAC地址复制。

如果地址是一个组播地址，isMulticastAddress()方法返回true，否则返回false。组播会将内容广播给所有预订的计算机，而不是某一台计算机。在IPv4中，组播地址都在224.0.0.0到239.255.255.255范围内。在IPv6中，组播地址都以字节FF开头。第13章将讨论组播。

如果地址是全球组播地址，isMCGlobal()方法返回true，否则返回false。全球组播地址可能在世界范围内都有订购者。所有组播地址都以FF开头。在IPv6中，全球组播地址以FF0E或FF1E开头，这取决于这个组播地址是已知的永久分配地址还是一个临时地址。在IPv4中，所有组播地址都是全球范围的，至少对这个方法而言是这样。在第13章你将看到，IPv4使用生存时间（TTL）值而不是地址来控制范围。

如果地址是一个组织范围组播地址，isMCOrgLocal()方法返回true，否则返回false。组织范围组播地址可能在公司或组织的所有网站中都有订购者，但不包括组织以外。组织组播地址以FF08或FF18开头，这取决于这个组播地址是已知的永久分配地址还是一个临时地址。

如果地址是一个网站范围组播地址，isMCSiteLocal()方法返回true，否则返回false。发送到网站范围地址的包只会在本地网站内传输。网站组播地址以FF05或FF15开头，这取决于这个组播地址是已知的永久分配地址还是一个临时地址。

如果地址是一个子网范围组播地址，isMCLinkLocal()方法返回true，否则返回false。发送到子网组播地址的包只会在自己的子网内传输。子网组播地址以FF02或FF12开头，这取决于这个组播地址是已知的永久分配地址还是一个临时地址。

如果地址是一个本地接口组播地址，isMCNodeLocal()方法返回true，否则返回false。发送到本地接口地址的包不能发送到最初的网络接口以外，即使是相同节点上的不同网络接口也不行。这主要用于网络调试和测试。本地接口组播地址以2字节FF01或FF11开头，这取决于这个组播地址是已知的永久分配地址还是一个临时地址。

---

**提示：** 这个方法的名与当前的术语不太一致。IPv6协议的早期草案称这种地址为“本地节点”地址，因而方法名为“isMCNodeLocal”。实际上，在将这个方法增加到JDK之前，IPNG工作组就已经修改了这个方法名，不过很遗憾，Sun没有及时拿到备忘录。

---

示例4-6是一个简单的程序，这里使用这10个方法来测试从命令行输入的一个地址的性质。

示例4-6：测试IP地址的性质

```
import java.net.*;

public class IPCharacteristics {

  public static void main(String[] args) {

    try {
      InetAddress address = InetAddress.getByName(args[0]);
```

```
      if (address.isAnyLocalAddress()) {
        System.out.println(address + " is a wildcard address.");
      }
      if (address.isLoopbackAddress()) {
        System.out.println(address + " is loopback address.");
      }
      if (address.isLinkLocalAddress()) {
        System.out.println(address + " is a link-local address.");
      } else if (address.isSiteLocalAddress()) {
        System.out.println(address + " is a site-local address.");
      } else {
        System.out.println(address + " is a global address.");
      }

      if (address.isMulticastAddress()) {
        if (address.isMCGlobal()) {
          System.out.println(address + " is a global multicast address.");
        } else if (address.isMCOrgLocal()) {
          System.out.println(address
           + " is an organization wide multicast address.");
        } else if (address.isMCSiteLocal()) {
          System.out.println(address + " is a site wide multicast
                             address.");
        } else if (address.isMCLinkLocal()) {
          System.out.println(address + " is a subnet wide multicast
                             address.");
        } else if (address.isMCNodeLocal()) {
          System.out.println(address
           + " is an interface-local multicast address.");
        } else {
          System.out.println(address + " is an unknown multicast
                             address type.");
        }
      } else {
        System.out.println(address + " is a unicast address.");
      }
    } catch (UnknownHostException ex) {
      System.err.println("Could not resolve " + args[0]);
    }
  }
}
```

下面是一个IPv4和IPv6地址的输出：

```
$ java IPCharacteristics 127.0.0.1
/127.0.0.1 is loopback address.
/127.0.0.1 is a global address.
/127.0.0.1 is a unicast address.
$ java IPCharacteristics 192.168.254.32
/192.168.254.32 is a site-local address.
/192.168.254.32 is a unicast address.
$ java IPCharacteristics www.oreilly.com
www.oreilly.com/208.201.239.37 is a global address.
www.oreilly.com/208.201.239.37 is a unicast address.
$ java IPCharacteristics 224.0.2.1
```

```
/224.0.2.1 is a global address.
/224.0.2.1 is a global multicast address.
$ java IPCharacteristics FF01:0:0:0:0:0:0:1
/ff01:0:0:0:0:0:0:1 is a global address.
/ff01:0:0:0:0:0:0:1 is an interface-local multicast address.
$ java IPCharacteristics FF05:0:0:0:0:0:0:101
/ff05:0:0:0:0:0:0:101 is a global address.
/ff05:0:0:0:0:0:0:101 is a site wide multicast address.
$ java IPCharacteristics 0::1
/0:0:0:0:0:0:0:1 is loopback address.
/0:0:0:0:0:0:0:1 is a global address.
/0:0:0:0:0:0:0:1 is a unicast address.
```

## 测试可达性

InetAddress类有两个isReachable()方法，可以测试一个特定节点对当前主机是否可达（也就是说，能否建立一个网络连接）。连接可能由于很多原因而阻塞，包括防火墙、代理服务器、行为失常的路由器和断开的线缆等，或者只是因为试图连接时远程主机没有开机。

```
public boolean isReachable(int timeout) throws IOException
public boolean isReachable(NetworkInterface interface, int ttl, int timeout)
    throws IOException
```

这些方法尝试使用traceroute（更确切地讲，就是ICMP echo请求）查看指定地址是否可达。如果主机在timeout毫秒内响应，则方法返回true；否则返回false。如果出现网络错误则抛出IOException异常。第二个方法还允许指定从哪个本地网络接口建立连接，以及“生存时间”（连接被丢弃前尝试的最大网络跳数）。

## Object方法

与其他各个类一样，java.net.InetAddres继承自java.lang.Object。因此，它可以访问Object类的所有方法。它覆盖了3个方法来提供更特殊的行为：

```
public boolean equals(Object o)
public int hashCode()
public String toString()
```

如果一个对象本身是InetAddress类的实例，而且与一个InetAddress对象有相同的IP地址，只有此时才会与该InetAddress对象相等，并不要求这两个对象有相同的主机名。因此，*www.ibiblio.org*的InetAddress对象等于*www.cafeaulait.org*的InetAddress对象，因为这两个主机名指向相同的IP地址。示例4-7为*www.ibiblio.org*和*helios.ibiblio.org*创建InetAddress对象，然后指出它们是否为同一台机器。

示例4-7：www.ibiblio.org和helios.ibiblio.org相同吗？

```
import java.net.*;

public class IBiblioAliases {

  public static void main (String args[]) {
    try {
      InetAddress ibiblio = InetAddress.getByName("www.ibiblio.org");
      InetAddress helios = InetAddress.getByName("helios.ibiblio.org");
      if (ibiblio.equals(helios)) {
        System.out.println
            ("www.ibiblio.org is the same as helios.ibiblio.org");
      } else {
        System.out.println
            ("www.ibiblio.org is not the same as helios.ibiblio.org");
      }
    } catch (UnknownHostException ex) {
      System.out.println("Host lookup failed.");
    }
  }
}
```

运行这个程序时会发现：

```
% java IBiblioAliases
www.ibiblio.org is the same as helios.ibiblio.org
```

hashCode()方法与equals()方法一致。hashCode()方法返回的int只根据IP地址来计算。它不考虑主机名。如果两个InetAddress对象有相同的地址，就会有相同的散列码，即使它们的主机名有所不同。

与所有好的类一样，java.net.InetAddress有一个toString()方法，可以返回对象的简单文本表示。示例4-1到示例4-2在将InetAddress对象传递给System.out.println()时都隐式调用了这个方法。如你所见，由toString()生成的字符串有如下形式：

```
主机名/点分四段地址
```

不是所有InetAddress都有主机名。如果没有，在Java 1.3及以前版本中就替换为点分四段式地址。在Java 1.4及以后版本中，主机名设置为空字符串。

## Inet4Address和Inet6Address

Java使用了两个类Inet4Address和Inet6Address，来区分IPv4地址和IPv6地址：

```
public final class Inet4Address extends InetAddress
public final class Inet6Address extends InetAddress
```

大多数情况下，你确实不必考虑一个地址是IPv4还是IPv6地址。在Java程序所在的应用

层，完全不需要了解（即使确实需要知道，只需检查getAddress()返回的字节数组的大小，这比使用instanceof来测试地址是哪一个子类要快得多）。Inet4Address覆盖了InetAddress的几个方法，但没有以任何公共方式改变其行为。Inet6Address也类似，但加入了超类中未出现的一个新方法，isIPv4CompatibleAddress()：

```
public boolean isIPv4CompatibleAddress()
```

当且仅当地址实际上是填充在IPv6“容器”中的一个IPv4地址时，这个方法返回true，这意味着只有最后4字节不是0。也就是说，地址的形式为*0:0:0:0:0:0:xxxx*。如果是这样，可以从getBytes()返回的数组中提取最后4字节，用这个数据创建一个Inet4Address。不过，很少需要这样做。

# NetworkInterface类

NetworkInterface类表示一个本地IP地址。这可以是一个物理接口，如额外的以太网卡（常见于防火墙和路由器），也可以是一个虚拟接口，与机器的其他IP地址绑定到同一个物理硬件。NetworkInterface类提供了一些方法可以枚举所有本地地址（而不考虑接口），并由它们创建InetAddress对象，然后这些InetAddress对象可用于创建socket、服务器socket等。

## 工厂方法

由于NetworkInterface对象表示物理硬件和虚拟地址，所以不能任意构造。与InetAddress类一样，有一些静态工厂方法可以返回与某个网络接口关联的NetworkInterface对象。可以通过IP地址、名字或枚举来请求一个NetworkInterface。

### public static NetworkInterface getByName(String name) throws SocketException

getByName()方法返回一个NetworkInterface对象，表示有指定名字的网络接口。如果没有这样一个接口，就返回null。如果在查找相关网络接口时底层网络栈遇到问题，会抛出一个SocketException异常，不过这种情况不太可能发生。

名字的格式与平台有关。在典型的UNIX系统上，以太网接口名的形式为eth0、eth1等。本地回送地址的名字可能类似于“lo”。在Windows上，名字是类似“CE31”和“ELX100”的字符串，取自这个特定网络接口的厂商名和硬件模型名。例如，下面的代码段尝试找到UNIX系统上的主以太网接口：

```
try {
  NetworkInterface ni = NetworkInterface.getByName("eth0");
```

```
    if (ni == null) {
      System.err.println("No such interface: eth0");
    }
  } catch (SocketException ex) {
    System.err.println("Could not list sockets.");
  }
```

### public static NetworkInterface getByInetAddress(InetAddress address) throws SocketException

getByInetAddress()方法返回一个NetworkInterface对象，表示与指定IP地址绑定的网络接口。如果本地主机上没有网络接口与这个IP地址绑定，就返回null。如果发生错误，就抛出一个SocketException异常。例如，下面的代码段会找到本地回送地址的网络接口：

```
try {
  InetAddress local = InetAddress.getByName("127.0.0.1");
  NetworkInterface ni = NetworkInterface.getByInetAddress(local);
  if (ni == null) {
    System.err.println("That's weird. No local loopback address.");
  }
} catch (SocketException ex) {
  System.err.println("Could not list network interfaces." );
} catch (UnknownHostException ex) {
  System.err.println("That's weird. Could not lookup 127.0.0.1.");
}
```

### public static Enumeration getNetworkInterfaces() throws SocketException

getNetworkInterfaces()方法返回一个java.util.Enumeration，这会列出本地主机上的所有网络接口。示例4-8是一个简单的程序，会列出本地主机上的所有网络接口：

示例4-8：列出所有网络接口的程序

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

public class InterfaceLister {

  public static void main(String[] args) throws SocketException {
    Enumeration<NetworkInterface> interfaces = NetworkInterface.getNetwork
    Interfaces();
    while (interfaces.hasMoreElements()) {
      NetworkInterface ni = interfaces.nextElement();
      System.out.println(ni);
    }
  }
}
```

下面是在IBiblio登录服务器上运行这个程序的结果：

```
% java InterfaceLister
```

```
name:eth1 (eth1) index: 3 addresses:
/192.168.210.122;

name:eth0 (eth0) index: 2 addresses:
/152.2.210.122;

name:lo (lo) index: 1 addresses:
/127.0.0.1;
```

你会看到这台主机有两个单独的以太网卡及本地回送地址。索引为2的以太网卡的IP地址是152.2.210.122，索引为3的以太网卡的IP地址是192.168.210.122。与往常一样，回送地址为127.0.0.1。

## 获取方法

有了NetworkInterface对象，就可以查询其IP地址和名字。这几乎是这些对象所能完成的唯一操作。

### public Enumeration getInetAddresses()

一个网络接口可以绑定多个IP地址。现在这种情况不太常见，但确实是存在的。getInetAddresses()方法返回一个java.util.Enumeration，其中对于与这个接口绑定的每一个IP地址都包含一个InetAddress。例如，下面的代码段列出etho接口的所有IP地址：

```
NetworkInterface eth0 = NetworkInterrface.getByName("eth0");
Enumeration addresses = eth0.getInetAddresses();
while (addresses.hasMoreElements()) {
  System.out.println(addresses.nextElement());
}
```

### public String getName()

getName()方法返回某个特定NetworkInterface对象的名，如eth0或lo。

### public String getDisplayName()

getDisplayName()方法声称可以返回特定NetworkInterface的一个更友好的名字，类似于“Ethernet Card 0”。不过，根据我在UNIX上的测试，它总是返回与getName()同样的结果。在Windows上，可以看到稍微友好一些的名字，如“Local Area Connection”或“Local Area Connection 2”。

# 一些有用的程序

现在你已经了解了java.net.InetAddress类的有关知识。只使用这个类中的工具就能编

写一些相当有用的程序。这里将看到两个例子：一个可以检查地址是否是一个已知的垃圾邮件发送者，另一个可以通过离线处理日志文件来提升Web服务器的性能。

## SpamCheck

很多服务会监视垃圾邮件发送者（spammer），并通知客户端试图与之连接的主机是否是一个已知的垃圾邮件发送者。这些实时黑洞列表（real-time blackhole lists，RBL）需要非常快地对查询做出响应，而且要处理相当大的工作负载。可能会有数千个甚至上百万个主机反复查询，查看试图建立连接的一个IP地址是否是一个已知的垃圾邮件发送者。

这个问题的性质要求响应必须很快，理想情况下，还应该可以缓存。另外，负载应当分布到多个服务器上，最好是位于世界各地的不同服务器。看起来可以使用Web服务器、SOAP、UDP、定制协议或者其他某种机制来实现，不过实际上只使用DNS就能巧妙地实现这个服务。

要查看一个IP地址是否是一个已知的垃圾邮件发送者，可以逆置这个地址的字节，增加黑洞服务的域，然后查找这个地址。如果找到这个地址，说明它是一个垃圾邮件发送者。如果没有找到，就说明它不是。例如，如果你想向*sbl.spamhaus.org*询问*207.87.34.17*是否是一个垃圾邮件发送者，就要查找主机名*17.34.87.207.sbl.spamhaus.org*（需要说明，尽管这里包含数字部分，不过这是一个主机名ASCII字符串，而不是一个点分四段IP地址）。

如果DNS查询成功（更确切地讲，如果它返回地址127.0.0.2），那么这个主机就是一个已知的垃圾邮件发送者。如果查找失败，也就是说，它抛出一个`UnknownHostException`，说明这个地址不是一个垃圾邮件发送者。示例4-9 实现了这个检查。

示例4-9：SpamCheck

```
import java.net.*;

public class SpamCheck {

  public static final String BLACKHOLE = "sbl.spamhaus.org";

  public static void main(String[] args) throws UnknownHostException {
    for (String arg: args) {
      if (isSpammer(arg)) {
        System.out.println(arg + " is a known spammer.");
      } else {
        System.out.println(arg + " appears legitimate.");
      }
    }
```

```
  }

  private static boolean isSpammer(String arg) {
    try {
      InetAddress address = InetAddress.getByName(arg);
      byte[] quad = address.getAddress();
      String query = BLACKHOLE;
      for (byte octet : quad) {
        int unsignedByte = octet < 0 ? octet + 256 : octet;
        query = unsignedByte + "." + query;
      }
      InetAddress.getByName(query);
      return true;
    } catch (UnknownHostException e) {
      return false;
    }
  }
}
```

以下给出一些示例输出：

```
$ java SpamCheck 207.34.56.23 125.12.32.4 130.130.130.130
207.34.56.23 appears legitimate.
125.12.32.4 appears legitimate.
130.130.130.130 appears legitimate.
```

如果使用这个技术，要注意掌握黑洞列表策略和地址的变化。出于很明显的原因，黑洞服务器经常成为DDOS和其他攻击的目标，所以如果黑洞服务器改变了地址，或者停止响应任何查询，你不能因此阻塞所有通信。

另外还要注意，不同的黑洞列表采用的协议可能稍有差别。例如，有些黑洞列表返回的垃圾邮件IP是*127.0.0.1*而不是*127.0.0.2*。

## 处理Web服务器日志文件

Web服务器日志会跟踪记录访问Web网站的主机。默认情况下，日志会报告连接服务器的网站的IP地址。不过，通常可以从网站的名字而不是其IP地址获得更多信息。大多数Web服务器有一个选项，可以存储主机名而不是IP地址，不过这可能会影响性能，因为每次访问时服务器都需要做一个DNS请求。如果先记录IP地址，稍后在服务器不太忙时再转换为主机名，或者甚至干脆在另外一个机器上完成转换，这样效率会更高。示例4-10给出了一个名为Weblog的程序，它读取Web服务器日志文件，显示各行时将IP地址转换为主机名。

大多数Web服务器都对常见的日志文件格式进行了标准化。常见日志文件格式中的一行一般如下：

```
205.160.186.76 unknown - [17/Jun/2013:22:53:58 -0500]
```

```
"GET /bgs/greenbg.gif HTTP 1.0" 200 50
```

这一行指示位于IP地址205.160.186.76的Web浏览器在2013年6月17日下午11:53（58秒）访问这个Web服务器上的文件*/bgs/greenbg.gif*。文件已找到（响应码200），向浏览器成功地传输了50字节数据。

第一个域是IP地址，或者如果启用了DNS解析，则是要建立连接的主机名。接下来是一个空格。因此，对我们来说，解析日志文件很简单，第一个空格之前的都是IP地址，其后的内容不需要改变。

点分四段格式IP地址使用`java.net.InetAddress`的常用方法转换为主机名。示例4-10展示了这些代码。

示例4-10：处理Web服务器日志文件

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

public class Weblog {

  public static void main(String[] args) {
    try (FileInputStream fin = new FileInputStream(args[0]);
      Reader in = new InputStreamReader(fin);
      BufferedReader bin = new BufferedReader(in);) {

      for (String entry = bin.readLine();
        entry != null;
        entry = bin.readLine()) {
        // 分解IP地址
        int index = entry.indexOf(' ');
        String ip = entry.substring(0, index);
        String theRest = entry.substring(index);

        // 向DNS请求主机名并显示
        try {
          InetAddress address = InetAddress.getByName(ip);
          System.out.println(address.getHostName() + theRest);
        } catch (UnknownHostException ex) {
          System.err.println(entry);
        }
      }
    } catch (IOException ex) {
      System.out.println("Exception: " + ex);
    }
  }
}
```

要处理的文件名作为命令行上的第一个参数传递给`Weblog`。从这个文件打开一个`FileInputStream fin`，并将一个`InputStreamReader`串链至`fin`。这个`InputStreamReader`通过串链到`BufferedReader`类的一个实例进行缓冲。文件在一个`for`循环中逐行处理。

每次循环时都会把一行放入String变量entry。然后entry被分解成两个子串：ip以及theRest，ip包含第一个空格之前的所有内容，theRest是第一个空格之后到字符串末尾的全部内容。第一个空格的位置由entry.indexOf(" ")确定。子串ip使用getByName()转换为一个InetAddress对象。然后getHostName()查询主机名。最后，在System.out上显示主机名以及这一行的所有其他内容（theRest）。输出可以通过标准输出重定向的方式，发送到新文件中。

Weblog比你预想的更高效。大多数Web浏览器会对提供的每个网页生成多个日志文件项，因为不只是页面本身有一个日志项，页面中的每个图片也分别对应有一个日志项。很多访问者访问网站时会请求多个页面。DNS查找成本很高，如果每个网站每次出现在日志文件中时都要查找，这样做并不合适。InetAddress类会缓存请求过的地址。如果再次请求相同的地址，它可以从缓存中获取，这比从DNS获取要快得多。

尽管如此，这个程序肯定还可以更快。在我最初的测试中，每个日志项花费的时间大于1秒（具体的数字取决于网络连接的速度、本地和远程DNS服务器的速度，以及程序运行时的网络拥塞状况）。这个程序耗费了大量时间等待DNS返回请求，在此期间什么也不做。显然，这正是多线程设计所要解决的问题。可以由一个主线程读取日志文件，将各个日志项传递给其他线程进行处理。

这里绝对需要一个线程池。经过几天之后，即使是低吞吐量的Web服务器也会生成包含数百万行的日志文件。如果试图为每一项生成一个新的线程来处理这样一个日志文件，即使是最强大的虚拟机也会很快吃不消，特别是主线程读取文件项的速度要比各个线程解析域名并结束的速度快得多。因此，很有必要重用线程。线程数目存储在一个可调整的参数numberOfThreads中，所以可以调整这个参数来满足VM和网络栈的需要（同时发起太多DNS请求也会出现问题）。

现在程序分为两个类。第一个类LookupTask如示例4-11所示，这是一个Callable类，它会解析一个日志文件项，查找一个地址，并把这个地址替换为相应的主机名。看上去好像没有太多工作，并不需要占用太多CPU，确实如此。不过，由于这里涉及网络连接，而且多个不同的DNS服务器之间可能还有一系列网络连接构成一个层次结构，所以有大量闲置时间，可以由其他线程更好地加以利用。

示例4-11：LookupTask

```
import java.net.*;
import java.util.concurrent.Callable;

public class LookupTask implements Callable<String> {

  private String line;
```

```
  public LookupTask(String line) {
    this.line = line;
  }

  @Override
  public String call() {
    try {
      // 分解IP地址
      int index = line.indexOf(' ');
      String address = line.substring(0, index);
      String theRest = line.substring(index);
      String hostname = InetAddress.getByName(address).getHostName();
      return hostname + " " + theRest;
    } catch (Exception ex) {
      return line;
    }
  }
}
```

第二个类PooledWeblog如示例4-12所示，其中包含main()方法，它会读取文件，并为每一行创建一个LookupTask。各个任务提交给一个executor，它可以并行和串行运行多个（不过不是全部）任务。

submit()方法返回的Future连同原来的行存储在一个队列中（以防异步线程中出错）。由一个循环从这个队列中读取值，并显示这些值。这样可以保持日志文件原来的顺序。

示例4-12：PooledWebLog

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

// 由于使用了try-with-resources和multi-catch，需要Java 7
public class PooledWeblog {

  private final static int NUM_THREADS = 4;

  public static void main(String[] args) throws IOException {
    ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);
    Queue<LogEntry> results = new LinkedList<LogEntry>();

    try (BufferedReader in = new BufferedReader(
      new InputStreamReader(new FileInputStream(args[0]), "UTF-8"));) {
      for (String entry = in.readLine(); entry != null; entry = in.readLine()) {
        LookupTask task = new LookupTask(entry);
        Future<String> future = executor.submit(task);
        LogEntry result = new LogEntry(entry, future);
        results.add(result);
      }
    }

    // 开始打印结果。每次结果未准备就绪时就会阻塞
    for (LogEntry result : results) {
```

```
      try {
        System.out.println(result.future.get());
      } catch (InterruptedException | ExecutionException ex) {
        System.out.println(result.original);
      }
    }

    executor.shutdown();
  }

  private static class LogEntry {
    String original;
    Future<String> future;

    LogEntry(String original, Future<String> future) {
      this.original = original;
      this.future = future;
    }
  }
}
```

使用这样的线程允许并行地处理同一个日志文件，从而节省大量时间。在我不太科学的测试中，多线程版本比串行版本快10到50倍。但是我们的技术编辑在另一个系统上运行同样的测试，只看到4倍的速度提升。不过无论如何，这仍是一个不小的进步。

这个设计还有一个缺点。与为每一个日志文件项生成一个线程相比，尽管Callable任务队列更为高效，但是日志文件可能很庞大，所以这个程序仍会占用大量内存。为避免这一点，可以把输出放在一个单独的线程中，它与输入线程共享同一个队列。由于解析输入的同时可以处理和显示之前的日志文件项，所以队列不会膨胀得过大。但是这又会带来另一个问题。你需要一个单独的信号指示输出已经完成，因为空队列已经不足以证明任务已经完成。最容易的方法是统计输入行数，确保它与输出行数一致。

# 第5章

# URL和URI

第4章中，你学习了如何通过主机名和IP地址确定主机在Internet的地址。这一章我们将提高力度，进一步学习如何确定资源的地址，任何给定主机上可能会有任意多个资源。

HTML是一个超文本（hypertext）标记语言，因为它提供了一种方法，可以指定URL标识的其他文档的链接。URL可以唯一地标识一个资源在Internet上的位置。URL是最常见的URI，即统一资源标识符（Uniform Resource Identifier）。URI可以由资源的网络位置来标识资源（如URL），也可以由资源的名字、编号或其他特性来标识。

URL类是Java程序在网络上定位和获取数据的最简单的方法。你不需要考虑所使用协议的细节，也不用担心如何与服务器通信。只要把URL告诉Java，它就会为你获得数据。

## URI

统一资源标识符（Uniform Resource Identifier，URI）是采用一种特定语法标识一个资源的字符串。所标识的资源可能是服务器上的一个文件。不过，也可能是一个邮件地址、新闻消息、图书、人名、Internet主机、Oracle的最新股价或者任何其他内容。

资源是由URI标识的内容。URI则是标识一个资源的字符串。没错，这里构成了一个环。不要花太多时间去考虑资源是什么或者不是什么，因为你根本不会看到资源。从服务器接收到的只是资源的一种字节表示。不过一个资源可能有多种不同的表示。例如，*https://www.un.org/en/documents/udhr/*标识了人权宣言（Universal Declaration of Human Rights），不过这个宣言还有纯文本、XML、PDF和其他格式的表示。另外，这个资源还有英语、法语、阿拉伯语和很多其他语言的表示。

其中有些表示本身就是资源。例如，*https://www.un.org/en/documents/udhr/*特别标识了英语版本的人权宣言。

---

**提示：** 好的Web体系结构的重要原则之一就是要充分使用URI。如果有人想要得到某个资源的地址，或者想要指示某个资源，可以提供一个URI（实际上是URL）。如果一个资源是另一个资源的一部分，或是其他资源的一个集合，或者是另一个资源在某个特定时刻的状态，它也完全可以有自己的URI。例如，在一个邮件服务中，每个用户、接收到的每个消息、发送的每个消息、收件箱的每个过滤视图、每个联系人、每个过滤规则，以及用户可能查看的每一个页面都要有一个唯一的URI。

尽管按层次构建的URI是一些很晦涩的字符串，不过实际中可以用人可读的子结构来设计。例如，*http://mail.example.com/*可能是一个特定的邮件服务器，*http://mail.example.com/johndoe*可能是这个服务器上John Doe的邮箱，*http://mail.example.com/johndoe?messageID=162977.1361.JavaMail.nobody%40meetup.com*则是这个邮箱中的一个特定的消息。

---

URI的语法由一个模式和一个模式特定部分组成，模式和模式特定部分用一个冒号分隔，如下所示：

```
模式:模式特定部分
```

模式特定部分的语法取决于所用的模式。当前的模式包括：

*data*
: 链接中直接包含的Base64编码数据，参见RFC 2397。

*file*
: 本地磁盘上的文件。

*ftp*
: FTP服务器。

*http*
: 使用超文本传输协议的国际互联网服务器。

*mailto*
: 电子邮件地址。

*magnet*
: 可以通过对等网络（如BitTorrent）下载的资源。

*telnet*
: 与基于Telnet的服务的连接。

*urn*

统一资源名（Uniform Resource Name，URN）。

此外，Java还大量使用了一些非标准的定制模式，如*rmi*、*jar*、*jndi*和*doc*，来实现各种不同用途。

URI中的模式特定部分并没有特定的语法。不过，很多都采用一种层次结构形式，如：

```
//authority/path?query
```

这个URI的*authority*部分指定了负责解析该URI其余部分的授权机构（authority）。例如，URI *http://www.ietf.org/rfc/rfc3986.txt*的模式为http，授权机构为*www.ietf.org*，另外路径为*/rfc/rfc3986.txt*（包括前面的斜线）。这表示位于www.ietf.org的服务器负责将路径*/rfc/rfc3986.txt*映射到一个资源。这个URI没有查询部分。URI *http://www.powells.com/cgibin/biblio?inkey=62-1565928709-0*的模式为http、授权机构为www.powells.com、路径为*/cgi-bin/biblio*，另外查询为`inkey=62-1565928709-0`。URI *urn:isbn:156592870*模式为*urn*，但模式特定部分没有采用层次结构的*//authority/path?query*形式。

尽管当前大多数的URI例子都使用Internet主机作为授权机构，不过未来的模式可能不是这样。但是，如果授权机构是Internet主机，那么还可以提供可选的用户名和端口，使授权机构更为特定。例如，URL *ftp://mp3:mp3@ci43198-a.ashvil1.nc.home.com:33/VanHalen-Jump.mp3*的授权机构是*mp3:mp3@ci43198-a.ashvil1.nc.home.com:33*。这个授权机构有用户名*mp3*、口令*mp3*、主机*ci43198-a.ashvil1.nc.home.com*和端口*33*。它的模式是ftp，路径是*/VanHalen-Jump.mp3*（在大多数情况下，在URI中包含口令是一个很大的安全漏洞，除非像这里一样，你确实想让全世界所有人都知道口令）。

路径是授权机构用来确定所标识资源的字符串。不同的授权机构可能会把相同的路径解释为指向不同的资源。例如，授权机构是*www.landoverbaptist.org*时，路径*/index.html*可能表示某个资源，而授权机构是*www.churchofsatan.com*时，路径*/index.html*则可能表示完全不同的一个资源。路径可以是分层的，在这种情况下，各个部分之间用斜线分隔，“.”和“..”操作符用于在这个层次结构中导航。这是从UNIX操作系统的路径名语法继承而来的（Web和URL都是在UNIX下发明的）。它们可以很方便地映射到存储在一个UNIX Web服务器上的文件系统。不过，不能保证任何特定路径的所有部分都能实际对应到特定文件系统的文件或目录。例如，在URI *http://www.amazon.com/exec/obidos/ISBN%3D1565924851/cafeaulaitA/002-3777605-3043449*中，这个层次结构的所有部分只是用来从数据库提取信息，并不存储在文件系统中。*ISBN%3D1565924851*根据ISBN号从数据库选择某本书，*cafeaulaitA*指定如果由此链接完成一次交易谁将获得推荐费，而*002-3777605-3043449*是一个会话密钥，用来跟踪访问者在网站中所走过的路径。

有些URI并不分层，至少在文件系统意义上是如此。例如，*snews://secnews.netscape.com/netscape.devs-java*的路径为*/netscape.devs-java*。虽然由*netscape*和*devs-java*之间的“.”指示新闻组名存在层次性，但这并未编码为URI的一部分。

模式部分由小写字母、数字和加号、点及连号符组成。典型URI的其他三部分（授权机构、路径和查询）分别由ASCII字母数字符号组成（即字母A-Z、a-z和数字0-9）。此外，还可以使用标点符号-、_、.、!和~。定界符（如/、?、&和=）可以有其预定义的用途。所有其他字符，包括非ASCII字母数字（如á和ζ），应当用百分号（%）转义，其后是该字符按UTF-8编码的十六进制码，另外一些定界符实际上没有用作为定界符，那么也需要这样转义。例如在UTF-8中，á是2字节0xC3 0xA1，所以要编码为%c3%a1。汉字“木”的Unicode 码点为0x6728。在UTF-8中，这会编码为3字节E6, 9C和A8。因此，它在URI中编码为%E6%9C%A8。

如果你没有像这样将非ASCII字符编码为十六进制码，而是将它们直接包含在URI中，那么你得到的不是一个URI，而是IRI（国际化资源标识符，Internationalized Resource Identifier）。IRI更容易录入，也更容易读，但是很多软件和协议只接受和支持URI。

除非用于特定URL中的模式特定部分，否则诸如“/”和“@”等标点符号也必须编码，要用百分号转义。例如，URI *http://www.cafeaulait.org/books/javaio2/*中的斜线不需要编码为%2F，因为它们是按http URI模式所指定的方式分隔这个层次结构。不过，如果一个文件名包括“/”字符，例如，如果为了与这本书的书名更为一致，将最后一个目录命名为*Java I/O*而不是*javaio2*，那么URI就必须写为*http://www.cafeaulait.org/books/Java%20I%2FO/*。对于UNIX或Windows用户而言，实际上并没有看上去那么牵强。Mac文件名经常包括一个斜线。许多平台上的文件名通常都包含需要编码的字符，包括@、$、+、=等。当然，一般情况下URL并非由文件名得来。

## URLs

URL是一个URI，除了标识一个资源，还会为资源提供一个特定的网络位置，客户端可以用它来获取这个资源的一个表示。与之不同，通用的URI可以告诉你一个资源是什么，但是无法告诉你它在哪里，以及如何得到这个资源。在实际世界中，这就像书名《哈里波特与死亡圣器》与这本书在图书馆的具体位置“312室第28行第7个书架”之间的区别。在Java中，这就类似于`java.net.URI`类（只标识资源）与`java.net.URL`类（既能标识资源，又能获取资源）之间的差别。

URL中的网络位置通常包括用来访问服务器的协议（如FTP、HTTP）、服务器的主机名或IP地址，以及文件在该服务器上的路径。典型的URL类似于*http://www.ibiblio.org/*

*javafaq/javatutorial.html*。它指示服务器*www.ibiblio.org*的*javafaq*目录下有一个名为*javatutorial.html*的文件，这个文件可以通过HTTP访问。

URL的语法为：

```
protocol://userInfo@host:port/path?query#fragment
```

这里的协议（protocol）是对URI中模式（scheme）的另一种叫法（URI RFC中使用“模式”。Java文档中使用“协议”）。在URL中，协议部分可以是file、ftp、http、https、magnet、telnet或其他各种字符串（但不包括urn）。

URL的主机（host）部分是提供所需资源的服务器的名字。这可以是一个主机名，如*www.oreilly.com*或*utopia.poly.edu*，也可以是服务器的IP地址，如204.148.40.9或128.238.3.21。

用户信息（userInfo）部分是服务器的登录信息（可选）。如果有这一部分，其中包含一个用户名，极少见的情况下还会包含一个口令。

端口（port）号也是可选的。如果服务在其默认端口运行（HTTP服务器的默认端口是80），就不需要这个部分。

用户信息、主机和端口合在一起构成权威机构（authority）。

路径（path）指向指定服务器上的一个特定目录。路径通常看上去类似一个文件系统路径，如*/forum/index.php*。它可能确实映射到服务器上的一个文件系统，不过也有可能并不映射到一个文件系统。如果确实映射到一个文件系统，路径则相对于服务器的文档根目录，而不一定相对于服务器上文件系统的根目录。一般来讲，向公众开放的服务器不会将其整个文件系统展示给客户端，而是只展示指定目录中的内容。这个目录称为文档根目录，所有路径和文件名都相对于这个目录。因此，在UNIX服务器上，公众可用的所有文件可能位于*/var/public/html*，但是对于某个从远程机器连接的人来说，这个目录就好像是文件系统的根目录一样。

查询（query）字符串向服务器提供附加参数。一般只在http URL中使用，其中包含表单数据，作为输入提供给在这个服务器上运行的程序。

最后，片段（fragment）指向远程资源的某个特定部分。如果远程资源是HTML，那么这个片段标识符将指定该HTML文档中的一个锚（anchor）。如果远程资源是XML，那么这个片段标识符是一个XPointer。有些文献将URL的片段部分称为“段”（section），Java文档则莫名其妙地把片段标识符称为“Ref”。片段标识符目标在HTML文档中用*id*属性创建，如：

```
<h3 id="xtocid1902914">Comments</h3>
```

这个标记标识文档中的某个点。为了引用这个点，URL不仅要包括文档的文件名，还要包括片段标识符，与URL的其余部分用#隔开：

```
http://www.cafeaulait.org/javafaq.html#xtocid1902914
```

---

**提示：** 从技术上讲，包含片段标识符的字符串是URL引用，而不是URL。但是Java不区分URL和URL引用。

---

## 相对URL

URL可以告诉Web浏览器一个文档的大量信息：用于获取此文档的协议、文档所在的主机，以及文档在该主机上的路径。大多数信息可能与该文档中引用的其他URL相同。因此，并不要求完整地指定每一个URL，URL可以继承其父文档（即这个URL所在的文档）的协议、主机名和路径。如果继承了父文档的部分信息，这些不完整的URL称为相对URL（relative URL）。相反，完整指定的URL称为绝对URL（absolute URL）。在相对URL中，可以认为缺少的各个部分都与所在文档的URL中对应的部分相同。例如，假设在浏览*http://www.ibiblio.org/javafaq/javatutorial.html*时单击这个超链接：

```
<a href="javafaq.html">
```

浏览器从*http://www.ibiblio.org/javafaq/javatutorial.html*末尾截去*javatutorial.html*，得到*http://www.ibiblio.org/javafaq/*。然后将*javafaq.html*附加到*http://www.ibiblio.org/javafaq/*末尾，得到*http://www.ibiblio.org/javafaq/javafaq.html*。最后加载这个文档。

如果相对链接以“/”开头，那么它相对于文档根目录，而不是相对于当前文件。因此，如果浏览*http://www.ibiblio.org/javafaq/javatutorial.html*时单击下面的链接：

```
<a href="/projects/ipv6/">
```

浏览器会去掉*/javafaq/javatutorial.html*，将*/projects/ipv6/*附加到*http://www.ibiblio.org*末尾，得到*http://www.ibiblio.org/projects/ipv6/*。

相对URL有很多优点。首先可以减少录入，不过这一点并不太重要。更重要的是，相对URL允许用多种协议来提供一个文档树：例如，HTTP和FTP。HTTP可能用于直接浏览，FTP可以用于镜像网站。最重要的一点是，相对URL允许将整个文档树从一个网站移动或复制到另一个网站，而不会破坏所有的内部链接。

# URL类

java.net.URL类是对统一资源定位符（如*http://www.lolcats.com/*或*ftp://ftp.redhat.com/pub/*）的抽象。它扩展了java.lang.Object，是一个final类，不能对其派生子类。它不依赖于继承来配置不同类型URL的实例，而使用了策略（strategy）设计模式。协议处理器就是策略，URL类构成上下文，通过它来选择不同的策略。

虽然把URL存储为字符串会很简单，但将URL作为对象考虑会很有帮助，这个对象的字段包括模式（也就是协议）、主机名、端口、路径、查询字符串和片段标识符（也称为ref），每个字段可以单独设置。实际上，java.net.URL类正是这样组织的，虽然不同版本Java之间在细节上稍有差别。

URL是不可变的。构造一个URL对象后，其字段不再改变。这有一个副作用：可以保证它们是“线程安全”的。

## 创建新的URL

与第4章的InetAddress对象不同，你可以构造java.net.URL的实例。不同构造函数所需的信息有所不同：

```
public URL(String url) throws MalformedURLException
public URL(String protocol, String hostname, String file)
    throws MalformedURLException
public URL(String protocol, String host, int port, String file)
    throws MalformedURLException
public URL(URL base, String relative) throws MalformedURLException
```

使用哪个构造函数取决于你有哪些信息以及信息的形式。如果试图为一个不支持的协议创建URL对象，或者如果URL的语法不正确，所有这些构造函数都会抛出一个MalformedURLException异常。

支持哪些协议取决于具体实现。所有虚拟机都支持的协议只有http和file，而且后者名声很不好。如今Java还支持https、jar和ftp。一些虚拟机还支持mailto和gopher，以及一些定制协议如doc、netdoc、systemresouce和Java在内部使用的verbatim。

---

**提示：** 如果某个VM不支持你需要的协议，可以为该模式安装一个协议处理器（protocol handler），使URL类支持这个协议。在实际中，这种方法带来的麻烦远甚于它带来的好处。最好使用一个库，提供专门支持该协议的一个定制API。

---

除了验证能否识别URL模式外，Java不会对它构造的URL完成任何正确性检查。程序员

要负责确保所创建的URL是合法的。例如，Java不会检查HTTP URL中的主机名中是否包含空格，或者查询字符串是否是x-www-form-URL-encoded。它不检查mailto URL是否真正包含一个电子邮件地址。你完全可以为不存在的主机创建URL，或者可以为尽管存在但不允许连接的主机创建URL。

## 从字符串构造URL

最简单的URL构造函数只接受一个字符串形式的绝对URL作为唯一的参数：

```
public URL(String url) throws MalformedURLException
```

与所有构造函数一样，这个函数只能在new操作符后调用，另外同样类似于所有其他URL构造函数，它可能会抛出MalformedURLException异常。下面的代码根据一个String构造一个URL对象，并捕获可能抛出的异常：

```
try {
  URL u = new URL("http://www.audubon.org/");
} catch (MalformedURLException ex) {
  System.err.println(ex);
}
```

示例5-1是一个很简单的程序，用来确定一个虚拟机支持哪些协议。它尝试为所有15种协议分别构造一个URL对象，这包括8种标准协议，不同Java API的3种定制协议和Java在内部使用的4种协议（无相关文档）。如果构造函数成功，则说明这个协议得到支持。否则，抛出一个MalformedURLException异常，由此可知虚拟机不支持这个协议。

示例5-1：虚拟机支持哪些协议？

```
import java.net.*;

public class ProtocolTester {

  public static void main(String[] args) {

    // 超文本传输协议
    testProtocol("http://www.adc.org");

    // 安全http
    testProtocol("https://www.amazon.com/exec/obidos/order2/");

    // 文件传输协议
    testProtocol("ftp://ibiblio.org/pub/languages/java/javafaq/");

    // 简单邮件传输协议
    testProtocol("mailto:elharo@ibiblio.org");

    // telnet
    testProtocol("telnet://dibner.poly.edu/");
```

```
    // 本地文件访问
    testProtocol("file:///etc/passwd");

    // gopher
    testProtocol("gopher://gopher.anc.org.za/");

    // 轻量组目录访问协议
    testProtocol(
        "ldap://ldap.itd.umich.edu/o=University%20of%20Michigan,c=US?postalAddress");

    // JAR
    testProtocol(
        "jar:http://cafeaulait.org/books/javaio/ioexamples/javaio.jar!"
        + "/com/macfaq/io/StreamCopier.class");

    // NFS, 网络文件系统
    testProtocol("nfs://utopia.poly.edu/usr/tmp/");

    // JDBC的定制协议
    testProtocol("jdbc:mysql://luna.ibiblio.org:3306/NEWS");

    // rmi, 远程方法调用的定制协议
    testProtocol("rmi://ibiblio.org/RenderEngine");

    // HotJava的定制协议
    testProtocol("doc:/UsersGuide/release.html");
    testProtocol("netdoc:/UsersGuide/release.html");
    testProtocol("systemresource://www.adc.org/+/index.html");
    testProtocol("verbatim:http://www.adc.org/");
  }

  private static void testProtocol(String url) {
    try {
      URL u = new URL(url);
      System.out.println(u.getProtocol() + " is supported");
    } catch (MalformedURLException ex) {
      String protocol = url.substring(0, url.indexOf(':'));
      System.out.println(protocol + " is not supported");
    }
  }
}
```

程序的结果取决于运行这个程序的虚拟机。以下是Mac OS X上Java 7的结果：

```
http is supported
https is supported
ftp is supported
mailto is supported
telnet is not supported
file is supported
gopher is not supported
ldap is not supported
jar is supported
nfs is not supported
jdbc is not supported
```

```
rmi is not supported
doc is not supported
netdoc is not supported
systemresource is not supported
verbatim is not supported
```

这里显示不支持RMI和JDBC，这实际上不太正确。事实上，JDK的确支持这些协议。不过，这两个协议分别通过java.rmi和java.sql包来支持，而无法跟其他支持协议一样可以通过URL来访问（但是我实在不明白，如果Sun不想通过Java处理URL的复杂机制来与RMI和JDBC交互，为什么还要为RMI和JDBC参数包裹上URL的外衣）。

其他Java 7虚拟机会显示类似的结果。并非来自Oracle代码基的虚拟机所支持的协议可能有些不同。例如，Android的Dalvik VM只支持http、https、file、ftp和jar协议。

### 由组成部分构造URL

还可以通过指定协议、主机名和文件来构建一个URL：

```
public URL(String protocol, String hostname, String file)
    throws MalformedURLException
```

这个构造函数将端口设置为-1，所以会使用该协议的默认端口。file参数应当以斜线开头，包括路径、文件名和可选的片段标识符。有可能会忘记最前面的斜线，这是一个很常见的错误，而且这个错误不容易发现。与所有URL构造函数一样，它可能会抛出MalformedURLException异常。例如：

```
try {
  URL u = new URL("http", "www.eff.org", "/blueribbon.html#intro");
} catch (MalformedURLException ex) {
  throw new RuntimeException("shouldn't happen; all VMs recognize http");
}
```

这会创建一个URL对象，指向*http://www.eff.org/blueribbon.html#intro*，并使用HTTP的默认端口（端口80）。文件规范包括指定锚的一个引用。如果虚拟机不支持HTTP，这个代码会捕获可能抛出的异常。不过，这在实际中不会发生。

在很少见的一些情况下，默认端口不正确时，下一个构造函数允许用一个int显式指定端口。其他参数都是一样的。例如，下面的代码段会创建一个指向*http://fourier.dur.ac.uk:8000/~dma3mjh/jsci/*的URL对象，这里显式地指定了端口8000：

```
try {
  URL u = new URL("http", "fourier.dur.ac.uk", 8000, "/~dma3mjh/jsci/");
} catch (MalformedURLException ex) {
  throw new RuntimeException("shouldn't happen; all VMs recognize http");
}
```

### 构造相对URL

这个构造函数根据相对URL和基础URL构建一个绝对URL：

```
public URL(URL base, String relative) throws MalformedURLException
```

例如，你可能正在解析HTML文档*http://www.ibiblio.org/javafaq/index.html*，并遇到一个名为*mailinglists.html*的文件链接，但没有进一步的限定信息。这时，可以用包含该链接的文档的URL来提供缺少的信息。这个构造函数会计算出新的URL为*http://www.ibiblio.org/javafaq/mailinglists.html*。例如：

```
try {
  URL u1 = new URL("http://www.ibiblio.org/javafaq/index.html");
  URL u2 = new URL (u1, "mailinglists.html");
} catch (MalformedURLException ex) {
  System.err.println(ex);
}
```

将文件名从u1的路径中去除，追加新文件名*mailinglists.html*得到u2。如果希望循环处理位于同一个目录下的一组文件，这个构造函数特别有用。可以为第一个文件创建一个URL，然后使用这个初始URL，通过替代文件名来创建其他文件的URL对象。

### 其他URL对象来源

除了这里讨论的构造函数，Java类库中的其他一些方法也返回URL对象。在applet中，getDocumentBase()会返回包含这个applet的页面的URL，getCodeBase()会返回applet.class文件的URL。java.io.File类有一个toURL()方法，它返回与指定文件匹配的*file* URL。这个方法所返回URL的具体格式与平台相关。例如，在Windows上，它可能返回类似*file:/D:/JAVA/JNP4/05/ToURLTest.java*的URL。在Linux和其他UNIX上，可能会看到*file:/home/elharo/books/JNP4/05/ToURLTest.java*。实际上，file URL非常依赖于平台和程序。Java file URL通常不能与用于Web浏览器和其他程序使用的URL互换，甚至不能与不同平台上运行的Java程序互换。

类加载器不仅用于加载类，也能加载资源，如图片和音频文件。静态方法ClassLoader.getSystemResource(String name)返回一个URL，通过它可以读取一个资源。ClassLoader.getSystemResources(String name)方法返回一个Enumeration，其中包含一个URL列表，通过这些URL可以读取指定的资源。最后，实例方法getResource(String name)会在所引用类加载器使用的路径中搜索指定资源的URL。这些方法返回的URL可能是file URL、HTTP URL或其他模式。资源的完全路径是用包限定的Java名，这里要用斜线而不是点，例如*/com/macfaq/sounds/swale.au*或*com/macfaq/images/headshot.jpg*。Java虚拟机会尝试在类路径中查找所请求的资源，很可能在一个JAR归档文件中。

类库中还有其他一些方法可以返回URL对象，但大多数是简单的获取方法，只返回一个你可能已经知道的URL，因为一开始就是用它来构造这个对象，例如java.swing.JEditorPane的getPage()方法和java.net.URLConnection的getURL()方法。

## 从URL获取数据

仅仅有URL并不太让人兴奋。大家关心的是URL所指向的文档中包含的数据。URL类有几个方法可以从URL获取数据：

```
public InputStream openStream() throws IOException
public URLConnection openConnection() throws IOException
public URLConnection openConnection(Proxy proxy) throws IOException
public Object getContent() throws IOException
public Object getContent(Class[] classes) throws IOException
```

这些方法中，最基本也是最常用的是openStream()，它会返回一个InputStream，可以从这个流读取数据。如果需要更多地控制下载过程，应当调用openConnection()，这会提供一个可以配置的URLConnection，再由它得到一个InputStream 。我们将在第7章讨论这个方法。最后，可以用getContent()向URL请求其内容，这会提供一个更完整的对象，如String或Image。同样的，它也会给出一个InputStream。

### public final InputStream openStream() throws IOException

openStream()方法连接到URL所引用的资源，在客户端和服务器之间完成必要的握手，返回一个InputStream，可以由此读取数据。从这个InputStream获得的数据是URL引用的原始内容（即未经解释的内容）：如果读取ASCII文本文件则为ASCII；如果读取HTML文件则为原始HTML，如果读取图像文件则为二进制图片数据等。它不包括任何HTTP首部或者与协议有关的任何其他信息。可以像读取任何其他InputStream一样读取这个InputStream。例如：

```
try {
  URL u = new URL("http://www.lolcats.com");
  InputStream in = u.openStream();
  int c;
  while ((c = in.read()) != -1) System.out.write(c);
  in.close();
} catch (IOException ex) {
  System.err.println(ex);
}
```

这段代码捕获IOException异常，它还会捕获URL构造函数可能抛出的MalformedURLException异常，因为MalformedURLException是IOException的子类。

与大多数网络流一样，要想可靠地关闭流，需要下点工夫。在Java 6及之前版本中，我

们使用了释放模式：在try块外声明流变量，并将它设置为null，然后在finally块中，如果流变量非null，则将它关闭。例如：

```
InputStream in = null
try {
  URL u = new URL("http://www.lolcats.com");
  in = u.openStream();
  int c;
  while ((c = in.read()) != -1) System.out.write(c);
} catch (IOException ex) {
  System.err.println(ex);
} finally {
  try {
    if (in != null) {
      in.close();
    }
  } catch (IOException ex) {
    // 忽略
  }
}
```

Java 7更为简洁，可以使用一个嵌套的try-with-resources语句：

```
try {
  URL u = new URL("http://www.lolcats.com");
  try (InputStream in = u.openStream()) {
    int c;
    while ((c = in.read()) != -1) System.out.write(c);
  }
} catch (IOException ex) {
  System.err.println(ex);
}
```

示例5-2从命令行读取一个URL，从这个URL打开一个InputStream，将得到的InputStream串链到使用默认编码方式的InputStreamReader，然后使用InputStreamReader的read()方法从文件读取连续的字符，将各个字符显示在System.out上。也就是说，如果URL引用一个HTML文件，它会显示位于这个URL的原始数据：程序的输出将是原始HTML。

示例5-2：下载一个Web页面

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

public class SourceViewer {

  public static void main (String[] args) {

    if (args.length > 0) {
      InputStream in = null;
      try {
```

```
        // 打开URL进行读取
        URL u = new URL(args[0]);
        in = u.openStream();
        // 缓冲输入以提高性能
        in = new BufferedInputStream(in);
        // 将InputStream串链到一个Reader
        Reader r = new InputStreamReader(in);
        int c;
        while ((c = r.read()) != -1) {
          System.out.print((char) c);
        }
      } catch (MalformedURLException ex) {
        System.err.println(args[0] + " is not a parseable URL");
      } catch (IOException ex) {
        System.err.println(ex);
      } finally {
        if (in != null) {
          try {
            in.close();
          } catch (IOException e) {
            // 忽略
          }
        }
      }
    }
  }
}
```

下面是SourceViewer下载*http://www.oreilly.com*时输出的前几行：

```
&lt;!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"&gt;
&lt;html xmlns="http://www.w3.org/1999/xhtml" lang="en-US" xml:lang="en-US"&gt;
&lt;head&gt;
&lt;title&gt;oreilly.com -- Welcome to O'Reilly Media, Inc. -- computer books,
software conferences, online publishing&lt;/title&gt;
&lt;meta name="keywords" content="O'Reilly, oreilly, computer books, technical
books, UNIX, unix, Perl, Java, Linux, Internet, Web, C, C++, Windows, Windows
NT, Security, Sys Admin, System Administration, Oracle, PL/SQL, online books,
books online, computer book online, e-books, ebooks, Perl Conference, Open Source
Conference, Java Conference, open source, free software, XML, Mac OS X, .Net, dot
net, C#, PHP, CGI, VB, VB Script, Java Script, javascript, Windows 2000, XP,
```

这个Web页面中还有很多行；如果想查看，可以在你的Web浏览器中自行查看。

这个程序最不可靠的部分是，它轻率地假定URL指向文本，而这不一定正确。它很可能指向GIF或JPEG图像、MP3声音文件或者其他内容。即使确实解析为文本，文档编码方式也可能与客户端系统的默认编码方式不同。远程主机和本地客户端可能采用不同的默认字符集。一般来讲，如果页面使用了与ASCII完全不同的字符集，HTML会在首部包括一个META标记，指示所使用的字符集。例如，下面的META标记指示了中文的Big-5编码方式：

```
<meta http-equiv="Content-Type" content="text/html; charset=big5">
```

XML文档则可能有一个XML声明：

```
<?xml version="1.0" encoding="Big5"?>
```

实际上，除了解析文件，查找类似这样的首部，并没有一种简单的方法可以得到这个信息，而且即使采用这种方法，也存在局限性。许多用拉丁字母手工编码的HTML文件没有这样的META标记。由于Windows、Mac和大多数UNIX对128到255的字符解释都稍有不同，所以除了创建这些文档的平台外，在其他平台上将无法正确地解释这些文档中的扩展字符。

更添乱的是，实际文档前面的HTTP首部可能还有自己的编码信息，这可能与文档本身声明的编码完全不同。不能使用URL类读取这个首部，但可以利用openConnection()方法返回的URLConnection对象来读取。编码方式的检测和声明是Web体系结构中比较棘手的问题之一。

### public URLConnection openConnection() throws IOException

openConnection()方法为指定的URL打开一个socket，并返回一个URLConnection对象。URLConnection表示一个网络资源的打开的连接。如果调用失败，则openConnection()会抛出一个IOException异常。例如：

```
try {
  URL u = new URL("https://news.ycombinator.com/");
  try {
    URLConnection uc = u.openConnection();
    InputStream in = uc.getInputStream();
    // 从连接读取...
  } catch (IOException ex) {
    System.err.println(ex);
  }
} catch (MalformedURLException ex) {
  System.err.println(ex);
}
```

如果希望与服务器直接通信，应当使用这个方法。通过URLConnection，你可以访问服务器发送的所有数据：除了原始的文档本身外（如HTML、纯文本、二进制图像数据），还可以访问这个协议指定的所有元数据。例如，如果模式是HTTP或HTTPS，URLConnection允许你访问HTTP首部以及原始HTML。除了从URL读取，URLConnection还允许你向URL写入数据，例如，向一个mailto URL发送电子邮件，或者提交表单数据。URLConnection是第7章将介绍的主要内容。

这个方法有一个重载版本，可以指定通过哪个代理服务器传递连接：

```
public URLConnection openConnection(Proxy proxy) throws IOException
```

这会覆盖用平常的socksProxyHost、socksProxyPort、http.proxyHost、http.proxyPort、http.nonProxyHosts和类似系统属性设置的任何代理服务器。如果协议处理器不支持代理，这个参数将被忽略，如果可能将直接建立连接。

### public final Object getContent() throws IOException

getContent()方法是下载URL引用数据的第三种方法。getContent()方法获取由URL引用的数据，尝试由它建立某种类型的对象。如果URL指示某种文本（如ASCII或HTML文件），返回的对象通常是某种InputStream。如果URL指示一个图像（如GIF或JPEG文件），getContent()通常返回一个java.awt.ImageProducer。这两个不同的类有一个共同点，它们本身并不是数据对象，而是一种途径，程序可以利用它们构造数据对象：

```
URL u = new URL("http://mesola.obspm.fr/");
Object o = u.getContent();
// 将Object强制转换为适当的类型
// 处理这个Object...
```

getContent()的做法是，在从服务器获取的数据首部中查找Content-type字段。如果服务器没有使用MIME首部，或者发送了一个不熟悉的Content-type，getContent()会返回某种InputStream，可以通过它读取数据。如果无法获取这个对象，就会抛出一个IOException异常，如示例5-3所示。

示例5-3：下载一个对象

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

public class ContentGetter {

  public static void main (String[] args) {

    if (args.length > 0) {
      // 打开URL进行读取
      try {
        URL u = new URL(args[0]);
        Object o = u.getContent();
        System.out.println("I got a " + o.getClass().getName());
      } catch (MalformedURLException ex) {
        System.err.println(args[0] + " is not a parseable URL");
      } catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

下面是尝试获得*http://www.oreilly.com*内容时的结果：

```
% java ContentGetter http://www.oreilly.com/ I got a
sun.net.www.protocol.http.HttpURLConnection$HttpInputStream</programlisting>
```

具体的类可能因为Java的版本不同而有所区别（较早版本中是java.io.PushbackInputStream或sun.net.www.http.KeepAliveStream），不过至少应当是某种形式的InputStream。

下面是尝试从这个页面加载页眉图像时的结果：

```
% java ContentGetter http://www.oreilly.com/graphics_new/animation.gif
I got a sun.awt.image.URLImageSource</programlisting>
```

下面是尝试用getContent()加载一个Java applet时的结果：

```
% java ContentGetter http://www.cafeaulait.org/RelativeURLTest.class</userinput>
I got a sun.net.www.protocol.http.HttpURLConnection$HttpInputStream
</programlisting>
```

下面是当尝试用getContent()加载一个音频文件时的结果：

```
% java ContentGetter http://www.cafeaulait.org/course/week9/spacemusic.au
</userinput>
I got a sun.applet.AppletAudioClip</programlisting>
```

最后的结果最不常见，因为这就像Java核心API访问一个表示声音文件的类。它不只是用来加载声音数据的接口。

这个例子显示出使用getContent()最大的问题：很难预测将获得哪种对象。可能得到某种InputStream或ImageProducer，或者可能是AudioClip，用instanceof操作符很容易检查。这个信息对于读取文本文件或显示一个图像应该足够了。

### public final Object getContent(Class[] classes) throws IOException

URL的内容处理器可以提供一个资源的不同视图。getContent()方法的这个重载版本允许你选择希望将内容作为哪个类返回。这个方法尝试以第一种可用的格式返回URL的内容。例如，如果首选将HTML文件作为一个String返回，而第二个选择是Reader，第三个选择是InputStream，可以编写以下代码：

```
URL u = new URL("http://www.nwu.org");
Class<?>[] types = new Class[3];
types[0] = String.class;
types[1] = Reader.class;
types[2] = InputStream.class;
Object o = u.getContent(types);
```

如果内容处理器知道如何返回资源的一个字符串表示，它就会返回一个String。如果它不知道如何返回资源的字符串表示，则返回Reader。倘若它也不知道如何将资源表示为

一个读取器，那么它将返回InputStream。必须用instanceof检查返回的对象的类型。例如：

```
if (o instanceof String) {
  System.out.println(o);
} else if (o instanceof Reader) {
  int c;
  Reader r = (Reader) o;
  while ((c = r.read()) != -1) System.out.print((char) c);
  r.close();
} else if (o instanceof InputStream) {
  int c;
  InputStream in = (InputStream) o;
  while ((c = in.read()) != -1) System.out.write(c);
  in.close();
} else {
  System.out.println("Error: unexpected type " + o.getClass());
}
```

## 分解URL

URL由以下5部分组成：

- 模式，也称为协议。
- 授权机构。
- 路径。
- 片段标识符，也称为段或ref。
- 查询字符串。

例如，在URL *http://www.ibiblio.org/javafaq/books/jnp/index.html?isbn=1565922069#toc* 中，模式是http，授权机构是*www.ibiblio.org*，路径是*/javafaq/books/jnp/index.html*，片段标识符是toc，查询字符串是isbn=1565922069。不过，并非所有URL都有所有这些部分。例如，URL *http://www.faqs.org/rfcs/rfc3986.html*只有模式、授权机构和路径，而没有片段标识符和查询字符串。

授权机构可以进一步划分为用户信息、主机和端口。例如，在URL *http://admin@www.blackstar.com:8080/*中，授权机构是*admin@www.blackstar.com:8080*，包含用户信息*admin*、主机*www.blackstar.com*和端口8080。

9个公共方法提供了URL这些部分的只读访问：`getFile()`、`getHost()`、`getPort()`、`getProtocol()`、`getRef()`、`getQuery()`、`getPath()`、`getUserInfo()`和`getAuthority()`。

### public String getProtocol()

getProtocol()方法返回一个String，其中包含URL的模式（如“http”、“https”或“file”）。例如，下面的代码段会打印*https*：

```
URL u = new URL("https://xkcd.com/727/");
System.out.println(u.getProtocol());
```

### public String getHost()

getHost()方法返回一个String，其中包含URL的主机名。例如，下面的代码段会打印*xkcd.com*：

```
URL u = new URL("https://xkcd.com/727/");
System.out.println(u.getHost());
```

### public int getPort()

getPort()方法将URL中指定的端口号作为一个int返回。如果URL中没有指定端口，getPort()返回-1，表示这个URL没有显式指定端口，将使用该协议的默认端口。例如，如果URL是*http://www.userfriendly.org/*，getPort()会返回-1；如果URL是*http://www.userfriendly.org:80/*，getPort()会返回80。下面的代码将显示端口号为-1，因为URL中没有指定端口：

```
URL u = new URL("http://www.ncsa.illinois.edu/AboutUs/");
System.out.println("The port part of " + u + " is " + u.getPort());
```

### public int getDefaultPort()

URL中没有指定端口时，getDefaultPort()方法返回这个URL的协议所使用的默认端口。如果没有为这个协议定义默认端口，getDefaultPort()将返回-1。例如，如果URL是*http://www.userfriendly.org/*，getDefaultPort()会返回80；如果URL是*ftp://ftp.userfriendly.org:8000/*，getDefaultPort()会返回21。

### public String getFile()

getFile()方法返回一个String，其中包含URL的路径部分；要记住，Java不会把URL分解为单独的路径和文件部分。从主机名后的第一个斜线（/）一直到开始片段标识符的#号之前的字符，都被认为是文件部分。例如：

```
URL page = this.getDocumentBase();
System.out.println("This page's path is "+ page.getFile());
```

如果URL没有文件部分，Java会把文件设置为空串。

### public String getPath()

getPath()方法几乎是getFile()的同义词。也就是说，它会返回一个String，其中包含URL的路径和文件部分。但是与getFile()不同，它返回的String中不包括查询字符串，只有路径。

---

**警告：** 注意，并不像你想象的那样，getPath()方法不只是返回目录路径，getFile()也不只是返回文件名。getPath()和getFile()都返回完整的路径和文件名。唯一的区别是getFile()还返回查询字符串，而getPath()不返回这一部分。

---

### public String getRef()

getRef()方法返回URL的片段标识符部分。如果URL没有片段标识符，则这个方法返回null。在下面的代码中，getRef()返回字符串xtocid1902914：

```
URL u = new URL(
    "http://www.ibiblio.org/javafaq/javafaq.html#xtocid1902914");
System.out.println("The fragment ID of " + u + " is " + u.getRef());
```

### public String getQuery()

getQuery()方法返回URL的查询字符串。如果URL没有查询字符串，则这个方法返回null。在下面的代码中，getQuery()返回字符串category=Piano：

```
URL u = new URL(
    "http://www.ibiblio.org/nywc/compositions.phtml?category=Piano");
System.out.println("The query string of " + u + " is " + u.getQuery());
```

### public String getUserInfo()

有些URL包括用户名，有时甚至会有口令信息。这些信息位于模式之后，而且在主机之前，用一个@符号将用户信息与主机分开。例如，在URL *http://elharo@java.oreilly.com/*中，用户信息是elharo。有些URL还在用户信息中包括口令。例如，在URL *ftp://mp3:secret@ftp.example.com/c%3a/stuff/mp3/*中，用户信息是*mp3:secret*。不过，大多数情况下，在URL中包括口令存在安全风险。如果URL没有任何用户信息，getUserInfo()就返回null。

mailto URL的行为可能与你想象的不一样。在类似*mailto:elharo@ibiblio.org*的URL中，“elharo@ibiblio.org”是路径，而不是用户信息和主机。这是因为，这个URL指定了消息的远程接收者，而不是发送消息的用户名和主机。

### public String getAuthority()

在URL的模式和路径之间，你会发现授权机构。URI的这一部分指示了解析资源的授权机构。在大多数情况下，授权机构包括用户信息、主机和端口。例如，在URL ftp://mp3:mp3@138.247.121.61:21000/c%3a/中，授权机构是mp3:mp3@138.247.121.61:21000，用户信息是*mp3:mp3*，主机是138.247.121.61，端口是21000。不过，并不是所有URL都有这几部分。例如，在URL *http://conferences.oreilly.com/java/speakers/*中，授权机构只有主机名*conferences.oreilly.com*。getAuthority()方法会以URL中的形式返回授权机构，可能有用户信息和端口，也可能没有。

示例5-4使用这些方法把命令行中输入的URL解析为各个组成部分。

示例5-4：URL的组成部分

```
import java.net.*;

public class URLSplitter {

  public static void main(String args[]) {

    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        System.out.println("The URL is " + u);
        System.out.println("The scheme is " + u.getProtocol());
        System.out.println("The user info is " + u.getUserInfo());

        String host = u.getHost();
        if (host != null) {
          int atSign = host.indexOf('@');
          if (atSign != -1) host = host.substring(atSign+1);
          System.out.println("The host is " + host);
        } else {
          System.out.println("The host is null.");
        }

        System.out.println("The port is " + u.getPort());
        System.out.println("The path is " + u.getPath());
        System.out.println("The ref is " + u.getRef());
        System.out.println("The query string is " + u.getQuery());
      } catch (MalformedURLException ex) {
        System.err.println(args[i] + " is not a URL I understand.");
      }
        System.out.println();
    }
  }
}
```

下面是针对本章中几个URL例子运行这个程序的结果：

```
% java URLSplitter \
ftp://mp3:mp3@138.247.121.61:21000/c%3a/ \
```

```
http://www.oreilly.com \
http://www.ibiblio.org/nywc/compositions.phtml?category=Piano \
http://admin@www.blackstar.com:8080/ \

The URL is ftp://mp3:mp3@138.247.121.61:21000/c%3a/
The scheme is ftp
The user info is mp3:mp3
The host is 138.247.121.61
The port is 21000
The path is /c%3a/
The ref is null
The query string is null
The URL is http://www.oreilly.com
The scheme is http
The user info is null
The host is www.oreilly.com
The port is -1
The path is
The ref is null
The query string is null

The URL is http://www.ibiblio.org/nywc/compositions.phtml?category=Piano
The scheme is http
The user info is null
The host is www.ibiblio.org
The port is -1
The path is /nywc/compositions.phtml
The ref is null
The query string is category=Piano

The URL is http://admin@www.blackstar.com:8080/
The scheme is http
The user info is admin
The host is www.blackstar.com
The port is 8080
The path is /
The ref is null
The query string is null</programlisting>
```

## 相等性和比较

URL类包含通常的equals()和hashCode()方法。这些方法的行为与你预想的一样。当且仅当两个URL都指向相同主机、端口和路径上的相同资源，而且有相同的片段标识符和查询字符串，才认为这两个URL是相等的。不过，这里有一个惊喜。实际上equals()方法会尝试用DNS解析主机，来判断两个主机是否相同，如可以判断*http://www.ibiblio.org/*和*http://ibiblio.org/*是一样的。

---

**警告：** 这说明，URL上的equals()可能是一个阻塞的I/O 操作！出于这个原因，应当避免将URL存储在依赖equals()的数据结构中，如java.util.HashMap。更好的选择是java.net.URI，可以在必要时将URI与URL来回转换。

---

另一方面，equals()还不够深入，不会具体比较两个URL标识的资源。例如，*http://www.oreilly.com/*不等于*http://www.oreilly.com/index.html*；另外*http://www.oreilly.com:80*也不等于*http://www.oreilly.com/*。

示例5-5为*http://www.ibiblio.org/*和*http://ibiblio.org/*创建URL对象，并用equals()方法指出它们是否相等。

示例5-5：http://www.ibiblio.org和http://ibiblio.org相等吗？

```
import java.net.*;

public class URLEquality {

  public static void main (String[] args) {
    try {
      URL www = new URL ("http://www.ibiblio.org/");
      URL ibiblio = new URL("http://ibiblio.org/");
      if (ibiblio.equals(www)) {
        System.out.println(ibiblio + " is the same as " + www);
      } else {
        System.out.println(ibiblio + " is not the same as " + www);
      }
    } catch (MalformedURLException ex) {
      System.err.println(ex);
    }
  }
}
```

运行这个程序时，会发现：

```
    <programlisting format="linespecific" id="I_7_tt233">% <userinput moreinfo=
      "none">
       java URLEquality</userinput>
    http://www.ibiblio.org/ is the same as http://ibiblio.org/</programlisting>
```

URL没有实现Comparable。

URL类还有一个sameFile()方法，可以检查两个URL是否指向相同的资源：

```
    public boolean sameFile(URL other)
```

这个比较与equals()基本上相同。这里也包括DNS查询，不过sameFile()不考虑片段标识符。比较*http://www.oreilly.com/index.html#p1*和*http://www.oreilly.com/index.html#q2*时，sameFile()返回true，而equals()会返回false。

下面的代码段使用sameFile()比较两个URL：

```
    URL u1 = new URL("http://www.ncsa.uiuc.edu/HTMLPrimer.html#GS");
    URL u2 = new URL("http://www.ncsa.uiuc.edu/HTMLPrimer.html#HD");
    if (u1.sameFile(u2)) {
```

```
    System.out.println(u1 + " is the same file as \n" + u2);
  } else {
    System.out.println(u1 + " is not the same file as \n" + u2);
  }
```

输出如下：

```
http://www.ncsa.uiuc.edu/HTMLPrimer.html#GS is the same file as
http://www.ncsa.uiuc.edu/HTMLPrimer.html#HD
```

## 比较

URL有3个方法可以将一个实例转换为另外一种形式，分别是toString()、toExternalForm()和toURI()。

与所有好的类一样，java.net.URL有一个toString()方法。toString()生成的String总是绝对URL，如*http://www.cafeaulait.org/javatutorial.html*。显式调用toString()并不常见。显示（打印）语句会隐式调用toString()。除了显示（打印）语句以外，使用toExternalForm()更合适：

```
public String toExternalForm()
```

toExternalForm()方法将一个URL对象转换为一个字符串，可以在HTML链接或Web浏览器的打开URL对话框中使用。

toExternalForm()方法返回表示这个URL的一个人可读的String。它等同于toString()方法。事实上，toString()所做的就是返回toExternalForm()。

最后，toURI()方法将URL对象转换为对应的URI对象：

```
public URI toURI() throws URISyntaxException
```

稍后将讨论URI类。在这里，你要了解的要点是URI类提供了比URL类更精确、更符合规范的行为。对于像绝对化和编码等操作，在选择时应当首选URI类。如果需要把URL存储在一个散列表或其他数据结构中，也应当首选URI类，因为它的equals()方法不会阻塞。URL类应当主要用于从服务器下载内容。

# URI类

URI是对URL的抽象，不仅包括统一资源定位符（Uniform Resource Locators，URL），还包括统一资源名（Uniform Resource Names，URN）。实际使用的URI大多是URL，但大多数规范和标准（如XML）都是用URI定义的。在Java中，URI用java.net.URI类表示。这个类与java.net.URL类的区别表现在3个重要的方面：

- URI类完全有关于资源的标识和URI的解析。它没有提供方法来获取URI所标识资源的表示。
- 相比URL类，URI类与相关的规范更一致。
- URI对象可以表示相对URI。URL类在存储URI之前会将其绝对化。

简而言之，URL对象是对应网络获取的应用层协议的一个表示，而URI对象纯粹用于解析和处理字符串。URI类没有网络获取功能。尽管URL类有一些字符串解析方法，如getFile()和getRef()，但其中很多方法都有问题，与相关规范所要求的行为不完全一致。正常情况下，假如你想下载一个URL的内容，应当使用URL类，如果想使用URL来完成标识而不是获取（例如表示一个XML命名空间），就应当使用URI类。二者都需要时，可以通过toURL()方法将URI转换为URL，还可以使用toURI()方法将URL转换为URI。

## 构造一个URI

URI从字符串构造。可以把整个URI通过一个字符串传入构造函数，也可以分部分传入：

```
public URI(String uri) throws URISyntaxException
public URI(String scheme, String schemeSpecificPart, String fragment)
    throws URISyntaxException
public URI(String scheme, String host, String path, String fragment)
    throws URISyntaxException
public URI(String scheme, String authority, String path, String query,
    String fragment) throws URISyntaxException
public URI(String scheme, String userInfo, String host, int port,
    String path, String query, String fragment) throws URISyntaxException
```

与URL类不同，URI类不依赖于底层协议处理器。只要是URI语法上正确，Java就不需要为了创建URI对象而理解其协议。因此，不同于URL类，URI类可以用于新的试验性的URI模式。

第一个构造函数根据任何满足条件的字符串创建一个新的URI对象。例如：

```
URI voice = new URI("tel:+1-800-9988-9938");
URI web = new URI("http://www.xml.com/pub/a/2003/09/17/stax.html#id=_hbc");
URI book = new URI("urn:isbn:1-565-92870-9");
```

如果字符串参数不遵循URI语法。例如，如果URI以冒号开头，这个构造函数将抛出一个URISyntaxException异常。这是一个受查异常，所以需要捕获这个异常，或者在调用构造函数的方法中声明可能抛出该异常。不过，有一条语法规则不会检查。与URI规范不同的是，URI中使用的字符不限于ASCII。它可以包括其他Unicode字符，如ø和é。在

语法上URI没有多少限制，具体来讲，一旦不再需要对非ASCII字符编码，并且允许相对URI，那就没有什么限制了。几乎所有字符串都可以解释为URI。

第二个构造函数需要一个模式特定部分，主要用于非层次URI。模式（scheme）是URI的协议，如http、urn、tel等。它必须由ASCII字母、数字及三个标点字符（+、-和.）组成。模式必须以字母开头。如果为这个参数传入null，则省略模式，这样会创建一个相对URI。例如：

```
URI absolute = new URI("http", "//www.ibiblio.org" , null);
URI relative = new URI(null, "/javafaq/index.shtml", "today");
```

模式特定部分取决于URI模式的语法。对于http URI是一种情况，对于mailto URI则是另一种情况，对于tel URI情况又有所不同。因为URI类会用百分号转义字符来对无效字符编码，实际上这部分不会有任何语法错误。

最后，第三个参数包含一个片段标识符（如果有的话）。再次说明，片段标识符中禁止的字符会自动转义。为这个参数传入null时，会忽略片段标识符。

第三个构造函数用于层次URI，如http和ftp URL。主机和路径（用/分隔）共同构成这个URI的模式特定部分。例如：

```
URI today= new URI("http", "www.ibiblio.org", "/javafaq/index.html", "today");
```

这会生成URI *http://www.ibiblio.org/javafaq/index.html#today*。

如果这个构造函数无法根据提供的各个部分构成一个有效的层次URI。例如，如果有一个模式，所以URI必须是绝对URI，但路径并不是以/开头，那么它将抛出一个URISyntaxException异常。

第四个构造函数与前一个基本相同，只是添加了一个查询字符串部分。例如：

```
URI today = new URI("http", "www.ibiblio.org", "/javafaq/index.html",
    "referrer=cnet&date=2014-02-23", "today");
```

与以往一样，任何不可转义的语法错误都会导致抛出URISyntaxException异常，所有参数都可以传入null从而忽略这一部分。

第5个构造函数是前面两个构造函数调用的主层次URI构造函数。这个方法将授权机构分解为用户信息、主机和端口部分，每个部分分别有自己的语法规则。例如：

```
URI styles = new URI("ftp", "anonymous:elharo@ibiblio.org",
    "ftp.oreilly.com", 21, "/pub/stylesheet", null, null);
```

不过，生成的URI仍然必须遵循URI的所有通用规则，再次说明，任何参数都可以传入null，从而在结果中忽略这一部分。

如果你确信你的URI是有效的，不违反任何规则，那么可以使用静态工厂方法URI.create()。与构造函数不同，它不抛出URISyntaxException异常。例如，下面的调用将使用电子邮件地址作为口令为匿名FTP访问创建一个URI：

```
URI styles = URI.create(
    "ftp://anonymous:elharo%40ibiblio.org@ftp.oreilly.com:21/pub/stylesheet");
```

如果这个URI证明是不正确的，这个方法会抛出一个IllegalArgumentException异常。这是一个运行时异常，所以不需要显式声明或捕获这个异常。

## URI的各部分

URI引用包括最多三个部分：模式、模式特定部分和片段标识符。一般格式为：

```
模式:模式特定部分:片段
```

如果省略模式，这个URI引用则是相对的。如果省略片段标识符，这个URI引用就是一个纯URI。URI类提供了一些获取方法，可以返回各个URI对象的这三个部分。getRawFoo()方法返回URI各部分的编码形式，相应的getFoo()方法首先对所有用百分号转义的字符进行解码，然后返回解码后的部分：

```
public String getScheme()
public String getSchemeSpecificPart()
public String getRawSchemeSpecificPart()
public String getFragment()
public String getRawFragment()
```

---

**提示：** 之所以没有getRawScheme()方法，这是因为URI规范要求：所有模式名都要由对URI合法的ASCII字符组成，模式名中不允许百分号转义。

---

如果某个URI对象中没有相应的部分，这些方法就返回null。例如，没有模式的相对URI或没有片段标识符的http URI。

有模式的URI是绝对（absolute）URI。没有模式的URI是相对（relative）URI。如果URI是绝对的，isAbsolute()方法返回true，如果是相对URI则返回false。

```
public boolean isAbsolute()
```

模式特定部分的细节根据模式类型的不同会有所差别。例如，在*tel* URI中，模式特

定部分的语法类似于电话号码。不过，在许多有用的URI中（包括很常见的*file*和*http* URI），模式特定部分都有一个特定的分层格式，划分为授权机构、路径和查询字符串。授权机构进一步分为用户信息、主机和端口。如果URI是一个层次URI时，isOpaque()返回false，如果不是分层的URI，也就是说，如果是不透明（opaque）的，isOpaque()则返回true：

```
public boolean isOpaque()
```

如果URI不透明，只能得到模式、模式特定部分和片段标识符。不过，如果URI是层次URI，那么对于层次URI的所有不同部分都有相应的获取方法：

```
public String getAuthority()
public String getFragment()
public String getHost()
public String getPath()
public String getPort()
public String getQuery()
public String getUserInfo()
```

这些方法都返回解码后的部分。换句话说，百分号转义会改为它们实际表示的字符，如%3C会改为<。如果希望得到URI原始的编码部分，还有五个对应的getRawFoo()方法：

```
public String getRawAuthority()
public String getRawFragment()
public String getRawPath()
public String getRawQuery()
public String getRawUserInfo()
```

记住，URI类与URI规范不同，非ASCII字符如é和ü绝不会先完成百分号转义，这样就仍会出现在getRawFoo()方法返回的字符串中，除非最初用于构造URI对象的字符串已经编码。

---

**提示：** 之所以没有getRawPort()和getRawHost()方法，这是因为可以保证这些部分总是由ASCII字符组成的。

---

当特定URI不包含某个信息时，例如URI *http://www.example.com*没有用户信息、路径、端口和查询字符串，相应的方法会返回null。getPort()是个例外。由于它声明为返回一个int，所以它无法返回null。实际上，它会返回－1表示省略了端口。

出于各种技术原因，Java并不总是在开始就检测授权机构部分中的语法错误，但这没有多少实际影响。不做这个检测的直接后果是，一般无法返回授权机构的各个部分：端口、主机和用户信息。在这种情况下，可以调用parseServerAuthority()强制重新解析授权机构：

```
public URI parseServerAuthority() throws URISyntaxException
```

原来的URI没有改变（URI对象是不可变的），但返回的URI对应用户信息、主机和端口有单独的授权机构部分。如果无法解析授权机构，就会抛出一个URISyntaxException异常。

示例5-6使用这些方法将命令行输入的URI分解为各个组成部分。它与示例5-4相似，但可以处理任何语法正确的URI，而不仅限于Java提供了相应协议处理器的那些URI。

示例5-6：URI的各部分

```
import java.net.*;

public class URISplitter {

  public static void main(String args[]) {

    for (int i = 0; i < args.length; i++) {
      try {
        URI u = new URI(args[i]);
        System.out.println("The URI is " + u);
        if (u.isOpaque()) {
          System.out.println("This is an opaque URI.");
          System.out.println("The scheme is " + u.getScheme());
          System.out.println("The scheme specific part is "
              + u.getSchemeSpecificPart());
          System.out.println("The fragment ID is " + u.getFragment());
        } else {
          System.out.println("This is a hierarchical URI.");
          System.out.println("The scheme is " + u.getScheme());
          try {
            u = u.parseServerAuthority();
            System.out.println("The host is " + u.getHost());
            System.out.println("The user info is " + u.getUserInfo());
            System.out.println("The port is " + u.getPort());
          } catch (URISyntaxException ex) {
            // 必须是基于注册的授权机构
            System.out.println("The authority is " + u.getAuthority());
          }
          System.out.println("The path is " + u.getPath());
          System.out.println("The query string is " + u.getQuery());
          System.out.println("The fragment ID is " + u.getFragment());
        }
      } catch (URISyntaxException ex) {
        System.err.println(args[i] + " does not seem to be a URI.");
      }
      System.out.println();
    }
  }
}
```

下面是对本节中3个URI例子运行这个程序的结果：

```
% java URISplitter tel:+1-800-9988-9938 \
  http://www.xml.com/pub/a/2003/09/17/stax.html#id=_hbc \
  urn:isbn:1-565-92870-9
The URI is tel:+1-800-9988-9938
This is an opaque URI.
The scheme is tel
The scheme specific part is +1-800-9988-9938
The fragment ID is null

The URI is http://www.xml.com/pub/a/2003/09/17/stax.html#id=_hbc
This is a hierarchical URI.
The scheme is http
The host is www.xml.com
The user info is null
The port is -1
The path is /pub/a/2003/09/17/stax.html
The query string is null
The fragment ID is id=_hbc

The URI is urn:isbn:1-565-92870-9
This is an opaque URI.
The scheme is urn
The scheme specific part is isbn:1-565-92870-9
The fragment ID is null</programlisting>
```

## 解析相对URI

URI类提供了3个方法可以在相对和绝对URI之间来回转换。

```
public URI resolve(URI uri)
public URI resolve(String uri)
public URI relativize(URI uri)
```

resolve()方法将uri参数与这个URI进行比较，并用它构造一个新的URI对象，这个对象包装了一个绝对URI。例如，考虑下面3行代码：

```
URI absolute = new URI("http://www.example.com/");
URI relative = new URI("images/logo.png");
URI resolved = absolute.resolve(relative);
```

执行之后，resolved包含绝对URI *http://www.example.com/images/logo.png*。

如果调用URI本身不包含绝对URI，那么resolve()方法将尽可能地解析URI，并返回一个新的相对URI对象作为结果。例如，考虑下面的语句：

```
URI top = new URI("javafaq/books/");
URI resolved = top.resolve("jnp3/examples/07/index.html");
```

执行之后，现在resolved包含相对URI *javafaq/books/jnp3/examples/07/index.html*，没有模式和授权机构。

还可以反向完成以上过程，即从绝对URI变成相对URI。relativize()方法根据相对于调用URI的uri参数创建一个新的URI对象。参数没有改变。例如：

```
URI absolute = new URI("http://www.example.com/images/logo.png");
URI top = new URI("http://www.example.com/");
URI relative = top.relativize(absolute);
```

URI对象relative现在包含相对URI *images/logo.png*。

## 相等性和比较

如你所期望的，可以测试URI的相等性。这不是直接的字符串比较。相等的URI必须都是层次的或不透明的。比较模式和授权机构时不考虑大小写。也就是说，*http*和*HTTP*是相同的模式，*www.example.com*与*www.EXAMPLE.com*是相同的授权机构。URI的其余部分要区分大小写（除了用于转义无效字符的十六进制数字外）。转义字符在比较前不解码，*http://www.example.com/A*和*http://www.example.com/%41*是不相等的URI。

hashCode()方法与相等性是一致的。相等的URI有相同的散列码，不相等的URI不太可能有相同的散列码。

URI实现了Comparable，因此URI可以排序。基于各个部分的字符串比较，按以下列顺序进行排序：

1. 如果模式不同就比较模式，不考虑大小写。
2. 否则，如果模式相同，一般认为层次URI小于有相同模式的不透明URI。
3. 如果两个URI都是不透明URI，则根据模式特定部分对它们排序。
4. 如果模式和不透明的模式特有部分都相等，就根据片段比较URI。
5. 如果两个URI都是层次URI，则根据它们的授权机构部分排序，授权机构本身依次根据用户信息、主机和端口排序。主机比较不区分大小写。
6. 如果模式和授权机构都相等，就使用路径来区分。
7. 如果路径也相等，就比较查询字符串。
8. 如果查询字符串相等，就比较片段。

除了与URI自身比较外，URI不能与其他任何类型比较。将URI与任何其他对象比较都会导致ClassCastException异常。

## 字符串表示

有两个方法可以将URI对象转换为字符串：toString()和toASCIIString()。

```
public String toString()
public String toASCIIString()
```

toString()方法返回URI的未编码（unencoded）的字符串形式（也就是说，类似é和\的字符不用百分号转义）。因此，调用这个方法的结果不能保证是一个语法正确的URI，尽管实际上它是一个语法正确的IRI。这种形式有时对于向人们显示很有用，不过通常并不用于获取数据。

toASCIIString()方法返回URI的编码（encoded）的字符串形式。类似é和\的字符总是会完成百分号转义，无论最初是否已转义。大多数时候都应当使用这种URI字符串形式。尽管toString()返回的形式对人而言更容易阅读，但有可能会复制粘贴到不希望接收到无效URI的地方（而toString()返回的有可能是无效的URI）。toASCIIString()总是返回语法正确的URI。

## x-www-form-urlencoded

Web设计人员面对的挑战之一是要处理不同操作系统之间的区别。这些不同会导致URL的问题。例如，有些操作系统允许文件中有空格，而有些不允许。大多数操作系统不反对文件名中出现#号，但在URL中#号表示文件名的结束，后面是片段标识符。其他特殊字符、非字母数字字符等在URL中或另一个操作系统上有特殊的意义，这也会产生类似的问题。另外，发明Web时Unicode还没有完全普及，所以并不是所有系统都能处理“é”和“本”之类的字符。为了解决这些问题，URL中使用的字符必须来自ASCII的一个固定的子集，确切地讲，包括：

- 大写字母A-Z。
- 小写字母a-z。
- 数字0-9。
- 标点符号字符- _ . ! ~ * ' (和,) 。

字符: / & ? @ # ; $ + =和%也可以使用，但只用于特定的用途。如果这些字符出现在路径或查询字符串中，它们以及所有其他字符都应当编码。

编码方式非常简单。除了ASCII数字、字母和前面指定的标点符号以外，所有其他字符都要转换为字节，每个字节要写为百分号后面加两个十六进制数字。空格是一种特殊情况，因为它太普遍了。除了编码为%20，空格可以编码为加号（+）。加号本身编码为%2B。/ # = &和?字符用在名字中时应当编码，但作为URL各部分之间的分隔符时不用编码。

URL类不自动编码或解码。可以使用无效的ASCII字符和非ASCII字符以及（或）百分号转义字符来构造URL对象。由getPath()和toExternalForm()等方法输出时，这样的字符和转义字符不会自动编码或解码。要由你负责确保在用来构造URL对象的字符串中对所有这些字符正确地编码。

幸运的是，Java提供了URLEncoder和URLDecoder类，可以对这种格式的字符串编解码。

## URLEncoder

要对字符串完成URL编码，需要将这个字符串和字符集名传入URLEncoder.encode()方法。例如：

```
String encoded = URLEncoder.encode("This*string*has*asterisks", "UTF-8");
```

URLEncoder.encode()返回输入字符串的一个副本，不过有一些调整。所有非字母数字字符会转换为%序列（除空格、下划线、连字符、点号和星号字符以外）。它还会对所有非ASCII字符编码。空格转换为加号。这个方法有点过于积极，它还会把波浪线、单引号、感叹号和圆括号转换为百分号转义字符，即使它们并不一定需要转换。不过，URL规范不禁止这种转换，所以Web浏览器会合理地处理这些过度编码的URL。

尽管这个方法允许指定字符集，但是最好只选择UTF-8。与你选择的其他任何编码方式相比，UTF-8与IRI规范、URI类、现代Web浏览器和其他软件更兼容。

示例5-7是一个使用URLEncoder.encode()显示各种编码字符串的程序。

示例5-7：x-www-form-urlencoded字符串

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

public class EncoderTest {

  public static void main(String[] args) {

    try {
      System.out.println(URLEncoder.encode("This string has spaces",
                                           "UTF-8"));
      System.out.println(URLEncoder.encode("This*string*has*asterisks",
                                           "UTF-8"));
      System.out.println(URLEncoder.encode("This%string%has%percent%signs",
                                           "UTF-8"));
      System.out.println(URLEncoder.encode("This+string+has+pluses",
                                           "UTF-8"));
      System.out.println(URLEncoder.encode("This/string/has/slashes",
                                           "UTF-8"));
      System.out.println(URLEncoder.encode("This\"string\"has\"quote\"marks",
                                           "UTF-8"));
```

```
      System.out.println(URLEncoder.encode("This:string:has:colons",
                                           "UTF-8"));
      System.out.println(URLEncoder.encode("This~string~has~tildes",
                                           "UTF-8"));
      System.out.println(URLEncoder.encode("This(string)has(parentheses)",
                                           "UTF-8"));
      System.out.println(URLEncoder.encode("This.string.has.periods",
                                           "UTF-8"));
      System.out.println(URLEncoder.encode("This=string=has=equals=signs",
                                           "UTF-8"));
      System.out.println(URLEncoder.encode("This&string&has&ampersands",
                                           "UTF-8"));
      System.out.println(URLEncoder.encode("Thiséstringéhasé
                                           non-ASCII characters", "UTF-8"));
    } catch (UnsupportedEncodingException ex) {
      throw new RuntimeException("Broken VM does not support UTF-8");
    }
  }
}
```

下面是这个程序的输出（注意由于源代码中存在非ASCII字符，这个代码需要以非ASCII的方式保存，选择的编码方式应当作为参数传递给编译器）。

```
% javac -encoding UTF8 EncoderTest
% java EncoderTest
This+string+has+spaces
This*string*has*asterisks
This%25string%25has%25percent%25signs
This%2Bstring%2Bhas%2Bpluses
This%2Fstring%2Fhas%2Fslashes
This%22string%22has%22quote%22marks
This%3Astring%3Ahas%3Acolons
This%7Estring%7Ehas%7Etildes
This%28string%29has%28parentheses%29
This.string.has.periods
This%3Dstring%3Dhas%3Dequals%3Dsigns
This%26string%26has%26ampersands
This%C3%A9string%C3%A9has%C3%A9non-ASCII+characters</programlisting>
```

特别注意这个方法对斜线（/）、与号（&）、等号（=）和冒号（:）进行了编码。它不去判断这些字符在URL中如何使用。因此，你必须逐部分地对URL进行编码，而不是在一个方法调用中对整个URL编码。这是很重要的一点，因为URLEncoder最常见的用法是准备查询字符串，从而与使用GET方法的服务器端程序通信。例如，假设你希望对下面这个Google搜索的URL进行编码：

```
https://www.google.com/search?hl=en&as_q=Java&as_epq=I/O
```

以下代码段将对它进行编码：

```
String query = URLEncoder.encode(
    "https://www.google.com/search?hl=en&as_q=Java&as_epq=I/O", "UTF-8");
```

```
System.out.println(query);
```

但遗憾的是，输出为：

```
https%3A%2F%2Fwww.google.com%2Fsearch%3Fhl%3Den%26as_q%3DJava%26as_epq%3DI%2FO
```

问题就在于，URLEncoder.encode()会盲目地进行编码。它对URL或查询字符串中使用的特殊字符（比如/和=）和需要编码的字符不加区分。因此，需要一次编码URL的一部分，如下所示：

```
String url = "https://www.google.com/search?";
url += URLEncoder.encode("hl", "UTF-8");
url += "=";
url += URLEncoder.encode("en", "UTF-8");
url += "&";
url += URLEncoder.encode("as_q", "UTF-8");
url += "=";
url += URLEncoder.encode("Java", "UTF-8");
url += "&";
url += URLEncoder.encode("as_epq", "UTF-8");
url += "=";
url += URLEncoder.encode("I/O", "UTF-8");

System.out.println(url);
```

这一次会得到我们真正想要的输出：

```
https://www.google.com/search?hl=en&as_q=Java&as_epq=I/O
```

在这里，你也可以跳过一些常量字符串，如“Java”，不对它们进行编码，因为从观察可以知道，这些字符串中不包含需要编码的字符。不过，一般来讲，这些值将是变量，而不是常量。为了安全还是需要对每一部分进行编码。

示例5-8是一个QueryString类，它使用URLEncoder对一个Java对象中连续的名-值对编码，这个对象将用来向服务器端程序发送数据。为了增加名-值对，需要调用add()方法，它接受两个字符串作为参数，并进行编码。getQuery()方法返回编码后名-值对的累积列表。

示例5-8：QueryString类

```
import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;

public class QueryString {

  private StringBuilder query = new StringBuilder();

  public QueryString() {
  }
```

```
  public synchronized void add(String name, String value) {
    query.append('&');
    encode(name, value);
  }

  private synchronized void encode(String name, String value) {
    try {
      query.append(URLEncoder.encode(name, "UTF-8"));
      query.append('=');
      query.append(URLEncoder.encode(value, "UTF-8"));
    } catch (UnsupportedEncodingException ex) {
      throw new RuntimeException("Broken VM does not support UTF-8");
    }
  }

  public synchronized String getQuery() {
    return query.toString();
  }

  @Override
  public String toString() {
    return getQuery();
  }
}
```

使用这个类，现在可以对前面的例子进行编码：

```
QueryString qs = new QueryString();
qs.add("hl", "en");
qs.add("as_q", "Java");
qs.add("as_epq", "I/O");
String url = "http://www.google.com/search?" + qs;
System.out.println(url);
```

## URLDecoder

对应的URLDecoder类有一个静态方法decode()，它会对用x-www-form-url-encoded格式编码的字符串进行解码。也就是说，将所有加号转换为空格，所有百分号转义字符转换为对应的字符：

```
public static String decode(String s, String encoding)
    throws UnsupportedEncodingException
```

如果不确定使用哪种编码方式，那就选择UTF-8。它可能比所有其他方式都正确。

如果字符串包含一个百分号，但其后没有两个十六进制数字，或者字符串解码为无效的序列，就要抛出一个IllegalArgumentException异常。由于这个方法对非转义字符不做处理，所以可以传入整个URL，而不需要首先将它分解为各个部分。例如：

```
String input = "https://www.google.com/" +
```

```
    "search?hl=en&as_q=Java&as_epq=I%2FO";
String output = URLDecoder.decode(input, "UTF-8");
System.out.println(output);
```

# 代理

许多系统通过代理服务器（proxy server）访问Web，有时还会访问Internet的其他非HTTP部分。代理服务器接收到从本地客户端到远程服务器的请求。代理服务器向远程服务器发出请求，再将结果转发回本地客户端。有时这样做是出于安全原因，如防止远程主机了解关于本地网络配置的秘密细节。另外一些情况下，这样做是为了通过过滤出站请求，限制可以浏览的网站。例如，一所初级中学可能希望禁止访问*http://www.playboy.com*。还有一些情况则纯粹是出于性能的考虑，这样允许多个用户从本地缓存中获取同样的一些经常访问的文档，而不是重复从远程主机下载。

基于URL类的Java程序可以使用大多数常见的代理服务器和协议。事实上，正是出于这个原因，你要选择使用URL类，而不是在原始socket之上处理你自己的HTTP或其他客户端。

## 系统属性

对于基本操作，所要做的就是设置一些系统属性，指示本地代理服务器的地址。如果使用纯粹的HTTP代理，则将`http.proxyHost`设置为代理服务器的域名或IP地址，将`http.proxyPort`设置为代理服务器的端口（默认为80）。还有一些其他方法，包括在Java代码中调用`System.setProperty()`，或在启动程序时使用-D选项。下面的例子将代理服务器设置为192.168.254.254，端口为9000：

```
<programlisting format="linespecific" id="I_7_tt264">% <userinput moreinfo=
 "none">
  java -Dhttp.proxyHost=192.168.254.254 -Dhttp.proxyPort=9000 </userinput>
<emphasis role="bolditalic">com.domain.Program</emphasis></programlisting>
```

如果代理需要一个用户名和口令，则需要安装一个Authenticator，稍后将在“访问口令保护的网站”中介绍有关内容。

如果希望一台主机不被代理，而是要直接连接，则要把`http.nonProxyHosts`系统属性设置为其主机名或IP地址。如果多个主机都不需要代理，可以用竖线分隔这些主机名。例如，下面的代码段会禁止代理*java.oreilly.com*和*xml.oreilly.com*：

```
System.setProperty("http.proxyHost", "192.168.254.254");
System.setProperty("http.proxyPort", "9000");
System.setProperty("http.nonProxyHosts", "java.oreilly.com|xml.oreilly.com");
```

还可以使用星号作为通配符，表示某个特定的域或子域内的所有主机都不应当代理。例如，要禁止代理*oreilly.com*域内的所有主机：

```
% java -Dhttp.proxyHost=192.168.254.254 -Dhttp.nonProxyHosts=*.oreilly.com
<emphasis role="bolditalic">com.domain.Program</emphasis></programlisting>
```

如果使用FTP代理服务器，可以采用同样的方式设置ftp.proxyHost、ftp.proxyPort和ftp.nonProxyHosts属性。

Java不支持任何其他应用层协议，但是如果对所有TCP连接都使用传输层SOCKS代理，可以用socksProxyHost和socksProxyPort系统属性来确定。Java对于SOCKS没有提供禁止代理选项。这是一个“全有或全无”的选择。

## Proxy类

Proxy类允许从Java程序中对代理服务器进行更细粒度的控制。确切地讲，它允许你为不同的远程主机选择不同的代理服务器。代理本身用java.net.Proxy类的实例来表示。仍然只有三种代理：HTTP、SOCKS和直接连接（即根本没有代理），分别用Proxy.Type枚举中的三个常量来表示：

- Proxy.Type.DIRECT
- Proxy.Type.HTTP
- Proxy.Type.SOCKS

除了类型之外，关于代理的其他重要信息包括它的地址和端口，用SocketAddress对象表示。例如，下面的代码段创建了一个Proxy对象，表示*proxy.example.com*的端口80上的一个HTTP代理服务器：

```
SocketAddress address = new InetSocketAddress("proxy.example.com", 80);
Proxy proxy = new Proxy(Proxy.Type.HTTP, address);
```

虽然只有三种代理对象，但是对于不同主机上的不同代理服务器，可以有相同类型的多个不同代理。

## ProxySelector类

每个运行中的虚拟机都有一个java.net.ProxySelector对象，用来确定不同连接的代理服务器。默认的ProxySelector只检查各种系统属性和URL的协议，来决定如何连接到不同的主机。不过，你可以安装自己的ProxySelector子类来代替默认的选择器，用它根据协议、主机、路径、日期时间和其他标准来选择不同的代理。

这个类的关键是select()抽象方法：

```
public abstract List<Proxy> select(URI uri)
```

Java为这个方法传入一个URI对象（而不是URL对象），这表示需要连接的主机。举例来说，对于用URL类生成的连接，这个对象通常形式为*http://www.example.com/*或ftp://*ftp.example.com/pub/files/*。对于用Socket类生成的纯TCP连接，URI形式为*socket://host:port*。例如，*socket://www.example.com:80*。然后ProxySelector为这种类型的对象选择正确的代理，并返回到一个List<Proxy>中。

这个类中必须实现的第二个抽象方法是connectFailed()：

```
public void connectFailed(URI uri, SocketAddress address, IOException ex)
```

这是一个回调方法，用于警告程序这个代理服务器实际上没有建立连接。示例5-9展示了一个ProxySelector，它尝试使用位于*proxy.example.com*的代理服务器完成所有HTTP连接，除非这个代理服务器之前未能成功解析与一个特定URL的连接。如果是这样，它会建议使用直接连接。

示例5-9：ProxySelector会记住可以连接的URL

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

public class LocalProxySelector extends ProxySelector {

  private List<URI> failed = new ArrayList<URI>();

  public List<Proxy> select(URI uri) {

    List<Proxy> result = new ArrayList<Proxy>();
    if (failed.contains(uri)
        || !"http".equalsIgnoreCase(uri.getScheme())) {
      result.add(Proxy.NO_PROXY);
    } else {
      SocketAddress proxyAddress
          = new InetSocketAddress( "proxy.example.com", 8000);
      Proxy proxy = new Proxy(Proxy.Type.HTTP, proxyAddress);
      result.add(proxy);
    }

    return result;
  }

  public void connectFailed(URI uri, SocketAddress address, IOException ex) {
    failed.add(uri);
  }
}
```

如前所述，每个虚拟机都只有一个ProxySelector。要改变这个ProxySelector，需要把新的选择器传递给静态方法ProxySelector.setDefault()，如下：

```
ProxySelector selector = new LocalProxySelector();
ProxySelector.setDefault(selector);
```

此后，虚拟机打开的所有连接都将向这个ProxySelector询问将要使用的正确代理。一般不应在共享环境中运行的代码中使用这个方法。例如，不要在servlet中改变ProxySelector，因为这会改变在同一个容器中运行的所有servlet的ProxySelector。

## 通过GET与服务器端程序通信

URL类使得Java applet和应用程序与服务器端程序（如CGI、servlet、PHP页面和其他使用GET方法的程序）的通信非常容易（使用POST方法的服务器端程序需要使用URLConnection类，第7章将讨论有关内容）。你只需要知道程序希望接收怎样的名-值组合，然后用查询字符串构造URL，这个查询字符串提供所需的名和值。所有名和值都必须经过x-www-form-url-encoded编码，例如用URLEncoder.encode()方法编码，如本章前面所述。

有很多种方法可以确定与特定程序对话的查询字符串的具体语法。如果你自己编写了服务器端程序，那么你已经知道它希望接收的名-值对。如果在你自己的服务器上安装了一个第三方程序，这个程序的文档会告诉你它希望接收什么。如果你在与一个外部网络API（已提供文档）对话，如eBay Shopping API，这个服务通常会提供相当详细的文档，告诉你为了实现各种用途需要发送什么数据。

许多程序用于处理表单输入。如果是这种情况，要弄清程序希望得到什么输入会非常简单。表单使用的方法应当是FORM元素的METHOD属性值。这个值要么是GET，要么是POST。如果是GET，就可以使用这里介绍的过程；如果是POST，则要使用第7章描述的过程。URL中查询字符串前面的部分由FORM元素的ACTION属性值给定。需要说明，这可能是一个相对URL，这里需要确定相应的绝对URL。最后，名-值对就是INPUT元素的NAME属性。名-值对的值就是用户在表单中输入的内容。

例如，考虑下面这个HTML表单，这个表单在我的Cafe con Leche网站中作为本地搜索引擎。你会看到它使用了GET方法。处理表单的程序通过URL *http://www.google.com/search*进行访问。它有四个单独的名-值对，其中三个有默认值：

```
<form name="search" action="http://www.google.com/search" method="get">
  <input name="q" />
  <input type="hidden" value="cafeconleche.org" name="domains" />
  <input type="hidden" name="sitesearch" value="cafeconleche.org" />
  <input type="hidden" name="sitesearch2" value="cafeconleche.org" />
```

```
    <br />
    <input type="image" height="22" width="55"
      src="images/search_blue.gif" alt="search" border="0"
      name="search-image" />
</form>
```

INPUT域的类型无关紧要。例如，不论它是一组复选框、一个弹出列表还是一个文本域，都没有关系。重要的只是各个INPUT域的名以及你为它提供的值。Submit（提交）输入域告诉Web浏览器何时发送数据，但是并不向服务器提供任何附加的信息。有些情况下，你会发现一些隐藏的INPUT域，这些域必须有所需的默认值。这个表单就有三个隐藏的INPUT域。HTML中有很多不同的表单标记，可以生成弹出菜单、单选钮等。不过，尽管这些输入部件在用户看来大不相同，但是它们发送给服务器的数据的格式是一样的。每个表单元素都会提供一个名和一个编码的字符串值。

有些情况下，与你对话的程序可能无法处理作为特定输入域值的任意文本字符串。不过，由于表单要由人来阅读和填写，所以应当提供足够的线索，使人们能知道程序期待什么样的输入。例如，某个域应当是一个两字母的州缩写或者是一个电话号码。有时输入域可能没有这么明显的名字。甚至可能没有一个表单，而只有链接。在这种情况下，你必须做一些试验，首先复制一些现有的值，对它们做一些调整，看看接受哪些值，而哪些值不能接受。这个工作不需要在Java程序中完成。完全可以在Web浏览器窗口的地址栏或位置栏直接编辑URL。

---

**提示：** 其他黑客可能会以这种方式试验你的服务器端程序，由于这种可能性，所以要让你的程序非常健壮，能够应对意外的输入。

---

不管如何确定服务器所期望的名-值对，一旦知道了需要什么名-值对，与服务器通信就非常简单了。所要做的就是创建一个查询字符串，其中包含必要的名-值对，然后构造一个包括这个查询字符串的URL。将查询字符串发送到服务器，使用连接服务器和获取静态HTML页面所用的相同方法来读取其响应。一旦构造了URL，就没有要遵循的特殊协议了（不过对于POST方法，还有一个要遵循的特殊协议，这也是为什么要等到第7章才讨论这个方法的原因）。

为了演示这个过程，我们来编写一个非常简单的命令行程序，查看Open Directory中的主题。这个网站如图5-1所示，它的优点就是相当简单。

Open Directory界面是一个简单的表单，它有一个名为search的输入域；这个域中输入的输入会发送到位于*http://www.dmoz.org/search*的程序，它将完成具体的搜索。这个表单的HTML如下：


图5-1：Open Directory的用户界面

```
<form class="center mb1em" action="search" method="GET">
  <input style="*vertical-align:middle;" size="45" name="q" value="" class="qN">
  <input style="*vertical-align:middle; *padding-top:1px;" value="Search"
       class="btn" type="submit">
  <a href="search?type=advanced"><span class="advN">advanced</span></a>
</form>
```

这个表单中只有两个输入域：Submit按钮和名为q的文本域。因此，为了将搜索请求提交给Open Directory，只需要为*http://www.dmoz.org/search*追加q=*searchTerm*。例如，要搜索“java”，可以打开与URL *http://www.dmoz.org/search/?q=java*的连接，读取得到的输入流。示例5-10就是在做这个工作。

示例5-10：完成一个Open Directory搜索

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

public class DMoz {

  public static void main(String[] args) {

    String target = "";
    for (int i = 0; i < args.length; i++) {
```

```
      target += args[i] + " ";
    }
    target = target.trim();

    QueryString query = new QueryString();
    query.add("q", target);
     try {
      URL u = new URL("http://www.dmoz.org/search/q?" + query);
      try (InputStream in = new BufferedInputStream(u.openStream())) {
        InputStreamReader theHTML = new InputStreamReader(in);
        int c;
        while ((c = theHTML.read()) != -1) {
          System.out.print((char) c);
        }
      }
    } catch (MalformedURLException ex) {
      System.err.println(ex);
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

当然，解析和显示结果还有很多工作要做。但是可以注意到，与这个服务器对话的代码是如此简单。除了看起来有点可怕的URL，以及有些部分需要采用x-www-form-url-encoded编码形式，要与使用GET的服务器端程序对话，这并不比获取任何其他HTML页面更困难。

# 访问口令保护的网站

许多流行的网站需要提供用户名和口令才能访问。有些网站（如W3C成员页面）通过HTTP认证来实现这一点。其他的网站，如New York Times 网站，则通过cookie和HTML表单来实现。Java的URL类可以访问使用HTTP认证的网站，不过，当然需要提供用户名和口令。

对于使用基于cookie的非标准认证的网站，要提供支持会更有难度，很重要的一个原因是：不同网站的cookie认证有很大区别。实现cookie认证往往要实现一个完整的Web浏览器，而且需要提供充分的HTML表单和cookie支持。我们将在第7章讨论Java的cookie支持。访问采用标准HTTP认证保护的网站则要容易得多。

## Authenticator类

包java.net包括一个Authenticator类，可以用它为使用HTTP认证自我保护的网站提供用户名和口令：

```
public abstract class Authenticator extends Object
```

由于Authenticator是一个抽象类，所以必须派生子类。不同子类可以采用不同的方式获取信息。例如，字符模式程序可能只要求用户在System.in中输入用户名和口令。GUI程序可能给出一个如图5-2所示的对话框。自动化机器人可能从加密文件中读出用户名。

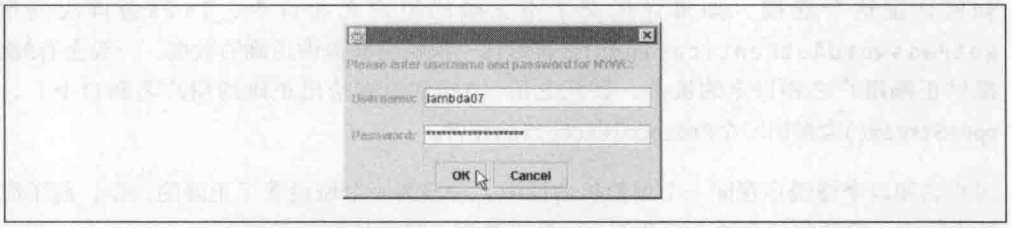


图5-2：认证对话框

为了让URL类使用这个子类，要把它传递给Authenticator.setDefault()静态方法，将它安装为默认的认证程序（Authenticator）：

```
public static void setDefault(Authenticator a)
```

例如，如果编写了一个名为DialogAuthenticator的Authenticator子类，就可以像如下安装这个认证程序：

```
Authenticator.setDefault(new DialogAuthenticator());
```

只需要安装一次。此后，当URL类需要用户名和口令时，它就会使用Authenticator.requestPasswordAuthentication()静态方法询问这个DialogAuthenticator：

```
public static PasswordAuthentication requestPasswordAuthentication(
    InetAddress address, int port, String protocol, String prompt, String scheme)
    throws SecurityException
```

address参数是需要认证的主机。port参数是该主机上的端口，protocol参数是访问网站的应用层协议。HTTP服务器提供prompt。这一般是需要认证的域的域名（有些大的Web服务器有多个域，如*www.ibiblio.org*，每个域都需要不同的用户名和口令）。scheme是所使用的认证模式（这里的模式并不是协议的同义词，这是HTTP认证模式，通常是basic）。

不可信的applet不允许向用户询问用户名和口令。可信的applet可以询问，但只有在拥有requestPasswordAuthentication NetPermission权限时才允许。否则，Authenticator.requestPasswordAuthentication()会抛出一个SecurityException异常。

Authenticator子类必须覆盖getPasswordAuthentication()方法。在这个方法中，要从

用户或其他来源收集用户名和口令，把它作为java.net.PasswordAuthentication类的一个实例返回：

```
protected PasswordAuthentication getPasswordAuthentication()
```

如果不希望对这个请求完成认证，就返回null，Java会告诉服务器它不知道如何认证这个连接。如果你提交了不正确的用户名和口令，Java会再次调用getPasswordAuthentication()，再给你一次机会来提供正确的数据。一般会有5次提供正确用户名和口令的机会。在此之后（5次都没有给出正确的用户名和口令），openStream()会抛出一个ProtocolException异常。

用户名和口令将缓存在同一个虚拟机会话中。一旦为一个域设置了正确的口令，就不应再次询问，除非将包含这个口令的char数组清零，显式地删除了口令。

通过调用从Authenticator超类继承的以下方法，可以得到请求的更多有关细节：

```
protected final InetAddress getRequestingSite()
protected final int         getRequestingPort()
protected final String      getRequestingProtocol()
protected final String      getRequestingPrompt()
protected final String      getRequestingScheme()
protected final String      getRequestingHost()
protected final String      getRequestingURL()
protected Authenticator.RequestorType getRequestorType()
```

这些方法会返回最后一次调用requestPasswordAuthentication()时给出的信息，或者如果没有可用信息，则返回null（如果没有可用端口，getRequestingPort()将返回-1）。

getRequestingURL()方法返回请求认证的完整URL。如果一个网站对不同文件使用不同的用户名和口令，这是一个重要的细节。getRequestorType()方法返回两个命名常量（Authenticator.RequestorType.PROXY或Authenticator.RequestorType.SERVER），表示请求认证的是服务器还是代理服务器。

## PasswordAuthentication类

PasswordAuthentication是非常简单的final类，它支持两个只读属性：用户名和口令。用户名是一个String。口令是一个char数组，这样当不再需要口令时可以将它清除。String在清除前必须等待垃圾回收，而那时它可能还在本地系统内存中的某个地方，如果包含它的内存块在某个时刻换出到虚拟内存上，它甚至在磁盘上。用户名和口令都在构造函数中设置：

```
public PasswordAuthentication(String userName, char[] password)
```

它们都通过一个获取方法访问：

```
public String getUserName()
public char[] getPassword()
```

## JPasswordField类

要以稍安全些的方式来询问用户口令，有一个很有用的工具，即Swing中的JPasswordField组件：

```
public class JPasswordField extends JTextField
```

这个轻量级组件的行为几乎与文本域完全相同。不过，用户在其中输入的所有内容都会回显为星号。这样一来，可以保证口令是安全的，别人无法从用户背后在屏幕上看到他输入的内容。

JPasswordField还把口令存储为char数组，这样在使用完口令后可以用0将其覆盖。它提供了getPassword()方法来返回这个数组：

```
public char[] getPassword()
```

其他情况下，主要使用从JTextField超类继承的方法。示例5-11展示了一个基于Swing的Authenticator子类，它弹出一个对话框，向用户询问用户名和口令。这段代码大部分都在处理GUI。一个JPasswordField收集口令，另外一个简单的JTextField获得用户名。返回到图5-2，那就是这个示例生成的简单对话框。

示例5-11：一个GUI认证程序

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.*;

public class DialogAuthenticator extends Authenticator {

  private JDialog passwordDialog;
  private JTextField usernameField = new JTextField(20);
  private JPasswordField passwordField = new JPasswordField(20);
  private JButton okButton = new JButton("OK");
  private JButton cancelButton = new JButton("Cancel");
  private JLabel mainLabel
      = new JLabel("Please enter username and password: ");
  public DialogAuthenticator() {
    this("", new JFrame());
  }

  public DialogAuthenticator(String username) {
    this(username, new JFrame());
```

```
}

public DialogAuthenticator(JFrame parent) {
  this("", parent);
}

public DialogAuthenticator(String username, JFrame parent) {
  this.passwordDialog = new JDialog(parent, true);
  Container pane = passwordDialog.getContentPane();
  pane.setLayout(new GridLayout(4, 1));

  JLabel userLabel = new JLabel("Username: ");
  JLabel passwordLabel = new JLabel("Password: ");
  pane.add(mainLabel);
  JPanel p2 = new JPanel();
  p2.add(userLabel);
  p2.add(usernameField);
  usernameField.setText(username);
  pane.add(p2);
  JPanel p3 = new JPanel();
  p3.add(passwordLabel);
  p3.add(passwordField);
  pane.add(p3);
  JPanel p4 = new JPanel();
  p4.add(okButton);
  p4.add(cancelButton);
  pane.add(p4);
  passwordDialog.pack();

  ActionListener al = new OKResponse();
  okButton.addActionListener(al);
  usernameField.addActionListener(al);
  passwordField.addActionListener(al);
  cancelButton.addActionListener(new CancelResponse());
}

private void show() {
  String prompt = this.getRequestingPrompt();
  if (prompt == null) {
    String site = this.getRequestingSite().getHostName();
    String protocol = this.getRequestingProtocol();
    int port = this.getRequestingPort();
    if (site != null & protocol != null) {
      prompt = protocol + "://" + site;
      if (port > 0) prompt += ":" + port;
    } else {
      prompt = "";
    }
  }

  mainLabel.setText("Please enter username and password for "
      + prompt + ": ");
  passwordDialog.pack();
  passwordDialog.setVisible(true);
}
```

```
  PasswordAuthentication response = null;

  class OKResponse implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      passwordDialog.setVisible(false);
      // 出于安全原因，
      // 口令作为char数组返回
      char[] password = passwordField.getPassword();
      String username = usernameField.getText();
      // 清除口令，以防再次使用
      passwordField.setText("");
      response = new PasswordAuthentication(username, password);
    }
  }

  class CancelResponse implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      passwordDialog.setVisible(false);
      // 清除口令，以防再次使用
      passwordField.setText("");
      response = null;
    }
  }

  public PasswordAuthentication getPasswordAuthentication() {
    this.show();
    return this.response;
  }
}
```

示例5-12是修改过的SourceViewer程序，它使用DialogAuthenticator类向用户询问用户名和口令：

示例5-12：下载由口令保护的Web页面的程序

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

public class SecureSourceViewer {

  public static void main (String args[]) {

    Authenticator.setDefault(new DialogAuthenticator());

    for (int i = 0; i < args.length; i++) {
      try {
        // 打开URL进行读取
        URL u = new URL(args[i]);
        try (InputStream in = new BufferedInputStream(u.openStream())) {
          // 将InputStream串链到一个Reader
          Reader r = new InputStreamReader(in);
          int c;
          while ((c = r.read()) != -1) {
```

```
            System.out.print((char) c);
          }
        }
      } catch (MalformedURLException ex) {
        System.err.println(args[0] + " is not a parseable URL");
      } catch (IOException ex) {
        System.err.println(ex);
      }

      // 打印一个空行，以分隔页面
      System.out.println();
    }

    // 由于我们使用了AWT，所以必须显式退出
    System.exit(0);
  }
}
```

# 第6章

# HTTP

超文本传输协议（Hypertext Transfer Protocol，HTTP）是一个标准，定义了Web客户端如何与服务器对话，以及数据如何从服务器传回客户端。尽管通常认为HTTP是一种传输HTML文件及文件中内嵌图片的方法，但实际上HTTP是一个数据格式。它可以用来传输TIFF图片、Microsoft Word文档、Windows的*.exe*文件，或者任何其他可以用字节表示的东西。要编写使用HTTP的程序，你需要比一般的Web页面设计人员更深入地了解HTTP。这一章将深入后台，向你展示在浏览器的地址栏输入*http://www.google.com*并按Enter键时到底发生了什么。

## HTTP协议

HTTP是Web浏览器和Web服务器之间通信的标准协议。HTTP指定客户端与服务器如何建立连接、客户端如何从服务器请求数据，服务器如何响应请求，以及最后如何关闭连接。HTTP连接使用TCP/IP来传输数据。对于从客户端到服务器的每一个请求，都有4个步骤：

1. 默认情况下，客户端在端口80打开与服务器的一个TCP连接，URL中还可以指定其他端口。
2. 客户端向服务器发送消息，请求指定路径上的资源。这个请求包括一个首部，可选地（取决于请求的性质）还可以有一个空行，后面是这个请求的数据。
3. 服务器向客户端发送响应。响应以响应码开头，后面是包含元数据的首部、一个空行以及所请求的文档或错误消息。
4. 服务器关闭连接。

这是基本HTTP 1.0过程。在HTTP 1.1及以后版本中，可以通过一个TCP连接连续发送多个请求和响应。也就是说，在第1步和第4步之间，第2步和第3步可以反复多次。另外，在HTTP 1.1中，请求和响应可以分为多个块发送。这样有更好的扩展性。

每个请求和响应都有同样的基本形式：一个首部行、一个包含元数据的HTTP首部、一个空行，然后是一个消息体。一般的客户端请求如下所示：

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:20.0)
Gecko/20100101 Firefox/20.0
Host: en.wikipedia.org
Connection: keep-alive
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

像这样的GET 请求不包含消息体，所以请求以一个空行结束。

第一行称为请求行（request line），包括一个方法、资源的路径以及HTTP的版本。方法指定了所请求的操作。GET方法请求服务器返回资源的一个表示。*/index.html*是从服务器请求的资源的路径。HTTP/1.1是客户理解的协议的版本。

尽管所需要的就是GET行，但客户端请求通常还会在首部中包括其他信息。各行采用如下形式：

```
Keyword: Value
```

关键字不区分大小写。值有时区分大小写，有时也不区分。关键字和值都只能是ASCII。如果一个值太长，可以在下一行最前面增加一个空格或制表符，接续上一行。

首部中的行以一个回车换行对结束。

这个例子中的第一个关键字是User-Agent，这会让服务器知道使用的是什么浏览器，并允许服务器发送为特定浏览器类型而优化的文件。下面一行指出请求来自Lynx浏览器的2.4版本：

```
User-Agent: Lynx/2.4 libwww/2.1.4
```

除了最老的第一代浏览器，还可以包括一个Host域来指定服务器的名，允许Web服务器区分来自相同IP地址的不同名的主机：

```
Host: www.cafeaulait.org
```

这个例子中最后一个关键字是Accept，它告诉服务器客户端可以处理哪些数据类型（但

服务器常常忽略这一点）。例如，下面一行指出客户端可以处理4种MIME媒体类型，分别对应HTML文档、纯文本及JPEG和GIF图片：

```
Accept: text/html, text/plain, image/gif, image/jpeg
```

MIME类型分为两级：类型（type）和子类型（subtype）。类型非常概括地展示包含的是何种数据：图片、文本，还是影片。子类型标识数据的特定类型：GIF图像、JPEG图像、TIFF图像。例如，HTML的内容类型是text/html，那么类型是text，子类型是html。JPEG图像的内容类型是image/jpeg，类型是image，子类型是jpeg。已经定义了8个顶级类型：

- text/* 表示人可读的文字。
- image/* 表示图片。
- model/* 表示3D模型，如VRML文件。
- audio/* 表示声音。
- video/* 表示移动的图片，可能包括声音。
- application/* 表示二进制数据。
- message/* 表示协议特定的信封，如email消息和HTTP响应。
- multipart/* 表示多个文档和资源的容器。

每个类型分别有很多不同的子类型。

可以在*http://www.iana.org/assignments/media-types/*访问已注册的MIME类型最新列表。另外，可以自由定义非标准的定制类型和子类型，只要它们以*x-*开头。例如，Flash文件通常会指定为*application/x-shockwave-flash*类型。

最后，请求以一个空行结束，也就是说，包括两个回车/换行对\r\n\r\n。

一旦服务器看到这个空行，它就开始通过同一个连接向客户端发送它的响应。这个响应以一个状态行开始，后面是一个首部，这个首部采用请求首部同样的“名:值”语法描述响应，然后是一个空行，最后是所请求的资源。一个典型的成功响应如下所示：

```
HTTP/1.1 200 OK
Date: Sun, 21 Apr 2013 15:12:46 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Content-length: 115
<html>
<head>
```

```
<title>
A Sample HTML file
</title>
</head>
<body>
The rest of the document goes here
</body>
</html>
```

第一行指示了服务器使用的协议(HTTP/1.1)，后面是一个响应码，200 OK是最常用的响应码，表示这个请求是成功的。其他首部行分别指出做出请求的日期（采用服务器的时间帧）、服务器软件（Apache）、承诺服务器结束发送时会关闭连接、MIME媒体类型，以及所传输文档的长度（不考虑这个首部），在这里就是107字节。

表6-1列出了你最有可能遇到的标准及试验响应码，这里不包括WebDAV使用的几个响应码。

表6-1：HTTP 1.1响应码

| 响应码和消息 | 含义 | HttpURLConnection常量 |
|---|---|---|
| 1XX | 信息 | |
| 100 Continue | 服务器准备接受请求主体，客户端应当发送请求主体；这允许客户端在请求中发送大量数据之前询问服务器是否将接受请求 | N/A |
| 101 Switching Protocols | 服务器接受客户端在Upgrade首部字段中要求改变应用协议的请求，如从HTTP改为WebSockets | N/A |
| 2XX Successful | 请求成功 | |
| 200 OK | 最常见的响应码。如果请求方法是GET或POST，所请求的数据与正常的首部一起包含在响应中。如果请求方法是HEAD，则只包括首部信息 | HTTP_OK |
| 201 Created | 服务器已经在响应主体中指定的URL处创建了资源。客户端现在应当尝试加载该URL。这个响应码只在响应POST请求时发送 | HTTP_CREATED |

表6-1：HTTP 1.1响应码（续）

| 响应码和消息 | 含义 | HttpURLConnection常量 |
| --- | --- | --- |
| 202 Accepted | 这是个很不常见的响应，表示请求（一般是POST）已经被处理，但处理尚未结束，所以不会返回任何响应。不过，服务器应当为用户返回一个HTML页面解释这个情况，并估计请求可能何时结束，理想情况下要提供某种状态监视器的链接 | HTTP_ACCEPTED |
| 203 Non-authoritative Information | 由缓存代理或其他本地源返回资源的表示，不能保证是最新的 | HTTP_NOT_AUTHORITATIVE |
| 204 No Content | 服务器已经成功地处理了请求，但没有信息发回给客户端。这一般是由于服务器上的表单处理程序编写得很差，只接受数据却不向用户返回响应 | HTTP_NO_CONTENT |
| 205 Reset Content | 服务器已经成功地处理了请求，但没有信息发回给客户端。此外，客户端应当清除发送请求的表单 | HTTP_RESET |
| 206 Partial Content | 服务器返回客户端请求的资源的一部分（使用HTTP的字节范围扩展），而不是整个文档 | HTTP_PARTIAL |
| 226 IM Used | 响应得到delta编码 | N/A |
| 3XX Redirection | 重定位及重定向 | |
| 300 Multiple Choices | 服务器为所请求的文档提供一组不同的表示（例如PostScript和PDF） | HTTP_MULT_CHOICE |
| 301 Moved Permanently | 资源已经移动到一个新的URL。客户端应当自动加载这个URL的资源，更新所有指向原URL的书签 | HTTP_MOVED_PERM |
| 302 Moved Temporarily | 这个资源暂时位于一个新的URL，但其位置在不久的将来还会再次改变，所以不应当更新书签。有时有些代理要求用户在访问Web之前先在本地登录，此时会用到这个响应码 | HTTP_MOVED_TEMP |

表6-1：HTTP 1.1响应码（续）

| 响应码和消息 | 含义 | HttpURLConnection常量 |
| --- | --- | --- |
| 303 See Other | 一般用于响应POST表单请求，这个响应码表示用户应当使用GET从另一个不同的URL获取资源 | HTTP_SEE_OTHER |
| 304 Not Modified | If-Modified-Since首部指示客户端只需要最近更新的文档。如果文档没有更新，就会返回这个状态码。在这种情况下，客户端应当从缓存中加载这个文档 | HTTP_NOT_MODIFIED |
| 305 Use Proxy | Location首部字段包含将提供响应的代理的地址 | HTTP_USE_PROXY |
| 307 Temporary Redirect | 类似于响应码302，但不允许HTTP方法改变 | N/A |
| 308 Permanent Redirect | 类似于响应码301，但不允许HTTP方法改变 | N/A |
| 4XX | 客户端错误 | |
| 400 Bad Request | 客户端向服务器发出的请求使用了不正确的语法。这在正常的Web浏览时很不常见，但是在调试定制客户端时比较常见 | HTTP_BAD_REQUEST |
| 401 Unauthorized | 访问这个页面需要身份认证，一般是用户名和口令。用户名和口令中可能有一个没有给出，或者用户名和口令无效 | HTTP_UNAUTHORIZED |
| 402 Payment Required | 现在没有使用，但将来可能用于指示访问该资源需要某种付费 | HTTP_PAYMENT_REQUIRED |
| 403 Forbidden | 服务器理解请求，但有意地拒绝进行处理。身份认证没有任何帮助。当客户端超出其配额时有时会使用这个响应码 | HTTP_FORBIDDEN |
| 404 Not Found | 最常见的错误响应，指示服务器找不到所请求的资源。它可能指示一个不正确的链接、已经移走而没有转发地址的文档、拼写错误的URL或其他类似情况 | HTTP_NOT_FOUND |

表6-1：HTTP 1.1响应码（续）

| 响应码和消息 | 含义 | HttpURLConnection常量 |
| --- | --- | --- |
| 405 Method Not Allowed | 请求方法不支持指定的资源；例如，试图在不支持PUT的Web服务器上使用PUT放置文件，或者使用POST提交到只支持GET的URI | HTTP_BAD_METHOD |
| 406 Not Acceptable | 所请求的资源不能以客户端希望的格式提供，客户端期望的格式由请求HTTP首部的Accept字段指示 | HTTP_NOT_ACCEPTABLE |
| 407 Proxy Authentication Required | 中间代理服务器要求客户端在获取所请求的资源之前，先对客户端进行身份认证（可能采用用户名和口令的形式） | HTTP_PROXY_AUTH |
| 408 Request Timeout | 客户端用了太长时间发送请求，可能是因为网络拥塞的原因 | HTTP_CLIENT_TIMEOUT |
| 409 Conflict | 一个临时冲突阻止了请求的实现。例如，两个客户端试图同时PUT相同的文件 | HTTP_CONFLICT |
| 410 Gone | 与404类似，但更有把握地确定资源的存在性。资源已经被有意地删除（不是移走），不能恢复。应当删除相应的链接 | HTTP_GONE |
| 411 Length Required | 客户端必须在请求HTTP首部中发送一个Content-length字段，但没有做到 | HTTP_LENGTH_REQUIRED |
| 412 Precondition Failed | 客户端在请求HTTP首部中指定的一个请求条件没有满足 | HTTP_PRECON_FAILED |
| 413 Request Entity Too Large | 客户端请求主体大于服务器一次能够处理的大小 | HTTP_ENTITY_TOO_LARGE |
| 414 Request-URI Too Long | 请求的URI太长。这对于防止某种缓冲区溢出攻击很重要 | HTTP_REQ_TOO_LONG |
| 415 Unsupported Media Type | 服务器不理解或不接受请求主体的MIME content-type | HTTP_UNSUPPORTED_TYPE |
| 416 Requested range Not Satisfiable | 服务器无法发送客户端所请求的字节范围 | N/A |

表6-1：HTTP 1.1响应码（续）

| 响应码和消息 | 含义 | HttpURLConnection常量 |
| --- | --- | --- |
| 417 Expectation Failed | 服务器无法满足客户端在Expect-request首部字段中给定的期望 | N/A |
| 418 I'm a teapot | 尝试用蜜罐泡咖啡 | N/A |
| 420 Enhance Your Calm | 服务器分级限制请求。这是非标准的，仅用于Twitter | N/A |
| 422 Unprocessable Entity | 不能识别请求主体的内容类型，请求主体的语法是正确的，只是服务器无法处理 | N/A |
| 424 Failed Dependency | 由于之前一个请求的失败，导致这个请求失败 | N/A |
| 426 Upgrade Required | 客户端使用一个太老或不安全的HTTP协议版本 | N/A |
| 428 Precondition Required | 请求必须提供一个If-Match首部 | N/A |
| 429 Too Many Requests | 客户端被分级限制，要慢下来 | N/A |
| 431 Request Header Fields Too Large | 可能首部作为一个整体太大，或者某个首部字段太大 | N/A |
| 451 Unavailable For Legal Reasons | 这是一个试验性的响应码，由于法律原因禁止服务器提供请求服务 | N/A |
| 5XX | 服务器错误 | |
| 500 Internal Server Error | 发生了意外情况，服务器不知道如何处理 | HTTP_SERVER_ERROR<br>HTTP_INTERNAL_ERROR |
| 501 Not Implemented | 服务器不具有完成这个请求所需要的一个特性。不能处理PUT请求的服务器可能会向试图PUT表单数据的客户端发送这个响应 | HTTP_NOT_IMPLEMENTED |
| 502 Bad Gateway | 这个响应码只用于作为代理或网关的服务器。它指示该代理在试图完成请求时，从它连接的服务器接收到一个无效的响应 | HTTP_BAD_GATEWAY |

表6-1：HTTP 1.1响应码（续）

| 响应码和消息 | 含义 | HttpURLConnection常量 |
| --- | --- | --- |
| 503 Service Unavailable | 服务器暂时无法处理请求，可能是超负荷或维护原因 | HTTP_UNAVAILABLE |
| 504 Gateway Timeout | 代理服务器在合理的时间内未能接收到上游服务器的响应，所以无法将向客户端发送所需的响应 | HTTP_GATEWAY_TIMEOUT |
| 505 HTTP Version Not Supported | 服务器不支持客户端正在使用的HTTP版本（如目前还不存在的HTTP 2.0） | HTTP_VERSION |
| 507 Insufficient Storage | 服务器没有足够的空间来存放所提供的请求实体。通常用于POST或PUT | |
| 511 Network Authentication Required | 客户端需要身份认证才能访问网络（例如，在一个旅馆的无线网络上） | N/A |

不论哪个版本，响应码100到199总表示一个提供信息的响应，200到299总指示成功，300到399表示重定向，400到499总是指示一个客户端错误，而500到599总表示一个服务器错误。

## Keep-Alive

HTTP 1.0会为每个请求打开一个新连接。实际上，一个典型Web会话中打开和关闭所有连接所花费的时间远远大于实际传输数据的时间，特别是有很多小文档的会话。对于使用SSL或TLS的加密HTTPS连接，这个问题尤其严重，因为建立一个安全socket的握手过程比建立常规的socket需要做更多工作。

在HTTP 1.1和以后版本中，服务器不必在发送响应后就关闭连接。可以保持连接打开，在同一个socket上等待来自客户端的新请求。可以在一个TCP连接上连续发送多个请求和响应。不过，服务器响应之后，客户端请求的锁步模式还是一样的。

客户可以在HTTP请求首部中包括一个*Connection*字段，指定值为*Keep-Alive*，指示它希望重用一个socket：

```
Connection: Keep-Alive
```

URL类透明地支持HTTP Keep-Alive，除非显式地关闭这个特性。也就是说，在服务器关闭连接之前，如果再次连接到同一个服务器，就会重用socket。可以利用多个系统属性来控制Java如何使用HTTP Keep-Alive：

- 设置http.keepAlive为“true”或“false”，启用/禁用HTTP Keep-Alive。（默认是启用的）。
- 设置http.maxConnections为你希望同时保持打开的socket数。默认为5。
- 设置http.keepAlive.remainingData为true，使Java在丢弃连接之后完成清理（Java 6或以后版本）。默认为false。
- 设置sun.net.http.errorstream.enableBuffering为true，尝试缓冲400和500级响应的相对小的错误流，从而能释放连接，以备稍后重用。默认为false。
- 设置sun.net.http.errorstream.bufferSize为缓冲错误流使用的字节数。默认为4096字节。
- 设置Set sun.net.http.errorstream.timeout为读错误流超时前的毫秒数。默认为300毫秒。

这些默认值是有合理的，不过，你可能确实希望把sun.net.http.errorstream.enableBuffering设置为true，除非你希望从失败的请求读取错误流。

---

**提示：** HTTP 2.0主要基于Google发明的SPDY协议，通过首部压缩、管线传输请求和响应，以及异步连接多路复用，进一步优化了HTTP传输。不过，这些优化通常在传输层完成，具体细节对应用程序员是屏蔽的，所以你编写的代码仍然主要遵循前面的1～4步。Java还不支持HTTP 2.0，不过增加这个功能时，只要你通过URL和URLConnection类访问HTTP服务器，你的程序不需要做任何修改就能利用这个功能。

---

# HTTP方法

与HTTP服务器的通信遵循一种请求-响应模式：先是一个无状态的请求，后面是一个无状态的响应。每个HTTP请求包括两个或三个部分：

- 起始行，包含HTTP方法和要执行这个方法的资源的路径。
- 一个包含名-值字段的首部，可以提供元信息，如认证凭据和请求中首选使用的格式。
- 一个请求主体，包含资源的一个表示（只针对POST和PUT）。

主要有4个HTTP方法（也可以说是4个动词），来标识可以完成的操作：

- GET
- POST

- PUT
- DELETE

如果觉得这太少了，特别是你可能已经习惯了设计程序时数不胜数的面向对象方法，相比之下，这确实太少了，要知道这是因为HTTP把重点主要放在名词上，即由URI标识的资源。这4个方法提供的统一接口基本上已经足以满足所有实用用途。

这4个方法并不是任意的。它们有特定的语义，应用程序必须遵循这些语义。GET方法可以获取一个资源的表示。GET没有副作用，如果失败，完全可以重复执行GET，而不用担心有任何问题。另外，GET的输出通常会缓存，不过这可以用正确的首部来控制，稍后就会介绍。在一个有适当体系结构的系统中，可以对GET请求加书签，也可以预取GET请求，这是没有问题的。例如，不允许通过一个链接删除文件，因为在用户请求之前，浏览器可能会GET页面上的所有链接。与之对应，如果没有明确的用户动作，好的浏览器或Web蜘蛛程序（spider）不会POST链接。

PUT方法将资源的一个表示上传到已知URL的服务器。这个方法并非没有副作用，不过它有幂等性（idempotent）。也就是说，可以重复这个方法而不用担心它是否失败。如果连续两次把同一个文档放在同一个服务器的同一个位置，与只放一次相比，服务器的状态是一样的。

DELETE方法从一个指定URL删除一个资源。同样的，这个方法也并非没有副作用，但它也是幂等的。如果你不确定一个删除请求是否成功（例如，有可能在你发送这个请求之后但在接收到响应之前，socket突然断开），在这种情况下，完全可以再次发送这个请求。将同一个资源删除两次不是错误。

POST方法是最通用的方法。它也将资源的一个表示上传到已知URL的服务器，但是没有指定服务器如何处理这个新提供的资源。例如，服务器不一定把资源放在目标URL上，而是有可能把它移至另一个不同的URL。或者服务器可能使用这个数据来更新一个或多个完全不同的资源的状态。POST要用于不能重复的不安全的操作，如完成一个交易。

由于GET请求在URL中包括了所有必要的信息，所以可以对GET请求加书签，或者进行链接和搜索等。POST、PUT和DELETE请求则不能。这是有意这样设计的。GET用于非提交的动作，如浏览一个静态Web页面。而其他方法，尤其是POST，则用于提交某个东西的动作。例如，在购物车里增加一个商品应当发送一个GET，因为这个动作没有提交，用户还可以放弃这个购物车。不过，下订单就应当发送一个POST，因为这个动作完成了一个提交。正是因为这个原因，当你要返回一个使用POST的页面时，浏览器会询问你是否确定这样做（见图6-1）。重新提交数据可能会重复买两本同样的书，你的信用卡会支付两次。

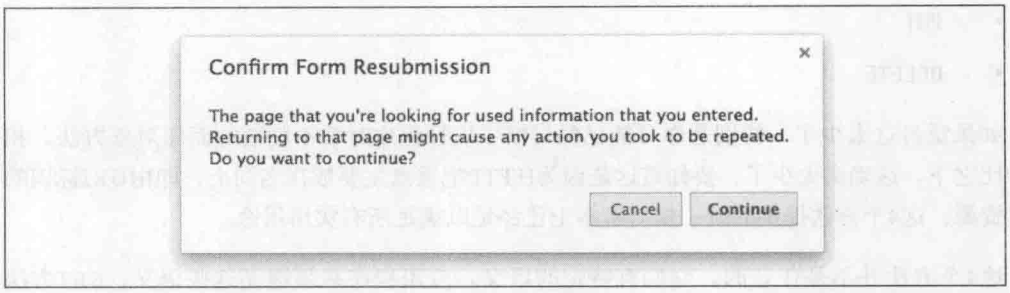


图6-1：重新提交确认

在实际中，如今POST在Web上被大量滥用。不完成提交的所有安全操作应当使用GET而不是POST。只有真正提交的操作才应当使用POST。

之所以有时会错误地选择POST而不是GET，一个原因是表单可能需要大量输入。在这方面有一个老掉牙的误解，以为浏览器只能处理几百个字节的查询字符串。尽管在20世纪90年代中期确实是这样，但如今所有主流浏览器都能很好地应对至少2000个字符的URL。如果有更多的表单数据需要提交，可能确实要支持POST。不过，对于非浏览器客户端，安全操作还是倾向于使用GET方法。实际上，这种情况没有你想象得那么常见。只有当你向服务器上传数据来创建一个新资源时有可能超过这个限制（在这种情况下，POST或PUT往往是合适的选择），否则，如果只是在服务器上定位一个现有的资源，一般都不会超过这个限制。

除了这4个主要的HTTP方法，特殊场合下还会用到另外几个HTTP方法。其中最常用的方法是HEAD，这个方法就相当于GET，只不过它只返回资源的首部，而不返回具体数据。这个方法常用于检查文件的修改日期，查看本地缓存中存储的文件副本是否仍然有效。

Java支持的另外两个HTTP方法是OPTIONS和TRACE。OPTIONS允许客户端询问服务器可以如何处理一个指定的资源，TRACE会回显客户端请求来进行调试，特别是代理服务器工作不正常时。不同的服务器还可能识别其他非标准的方法，包括COPY和MOVE，不过Java不支持这些方法。

上一章介绍的URL类使用GET与HTTP服务器通信。URLConnection类（第7章将介绍）可以使用所有这4种方法。

# 请求主体

GET方法获取URL所标识的资源的一个表示。用GET从服务器获取的资源的具体位置由路径和查询字符串的不同部分指定。不同的路径和查询字符串如何映射到不同的资源要由服务器来确定。URL类并不关心这些。只要它知道URL，就能从那里下载。

POST和PUT要更为复杂。在这些情况下，客户端除了要提供路径和查询字符串，还要提供资源的表示。资源表示在请求主体中发送，放在首部后面。也就是说，它会按顺序发送以下4项：

1. 一个起始行，包括方法、路径和查询字符串，以及HTTP版本。
2. 一个HTTP首部。
3. 一个空行（两个连续的回车/换行对）。
4. 主体。

例如，下面这个POST请求向服务器发送表单数据：

```
POST /cgi-bin/register.pl HTTP 1.0
Date: Sun, 27 Apr 2013 12:32:36
Host: www.cafeaulait.org
Content-type: application/x-www-form-urlencoded
Content-length: 54

username=Elliotte+Harold&email=elharo%40ibiblio.org
```

在这个例子中，主体包含一个*application/x-www-form-urlencoded*数据，但并不只有这一种可能。一般来讲，主体可以包含任意的字节。不过，HTTP首部要包括两个字段来指定主体的性质：

- 一个Content-length字段，指定主体中有多少字节（前面的例子中为54字节）。
- 一个Content-type字段，指定类型的MIME媒体类型（前面的例子中内容类型为*application/x-www-form-urlencoded*）。

前例中使用的*application/x-www-form-urlencoded* MIME类型很常见，因为Web浏览器对大多数提交表单就采用这种编码方式。因此很多服务器端程序都使用这个MIME类型与浏览器对话。不过，这绝不是主体能发送的唯一类型。例如，向一个照片共享网站上传图片的相机可以发送image/jpeg。文本编辑器可以发送text/html。最后都是作为字节传送。例如，下面是一个上传Atom文档的PUT请求：

```
PUT /blog/software-development/the-power-of-pomodoros/ HTTP/1.1
Host: elharo.com
User-Agent: AtomMaker/1.0
Authorization: Basic ZGFmZnk6c2VjZXJldA==
Content-Type: application/atom+xml;type=entry
Content-Length: 322

<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
 <title>The Power of Pomodoros</title>
 <id>urn:uuid:101a41a6-722b-4d9b-8afb-ccfb01d77499</id>
```

```
  <updated>2013-02-22T19:40:52Z</updated>
  <author><name>Elliotte Harold</name></author>
  <content>I hadn't paid much attention to Pomodoro...</content>
</entry>
```

# Cookie

很多网站使用一些小文本串在连接之间存储持久的客户端状态，这些小文本串称为cookie。cookie在请求和响应的HTTP首部,从服务器传递到客户端，再从客户端传回服务器。服务器使用cookie来指示会话ID、购物车内容、登录凭据、用户首选项等。例如，一个在线书店设置的cookie可能值为ISBN=0802099912&price=$34.95，指定我在购物车里放入的一本书。不过，cookie值更有可能是一个无意义的字符串，如ATVPDKIKX0DER，标识某种数据库中的一个特定记录，实际信息存储在那个数据库中。cookie值通常并不包含数据，只是指示服务器上的数据。

cookies只能是非空白符的ASCII文本，不能包含逗号或分号。

要在浏览器中设置一个cookie，服务器会在HTTP首部中包含一个Set-Cookie首部行。例如，下面的HTTP首部将cookie “cart”的值设置为“ATVPDKIKX0DER”：

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: cart=ATVPDKIKX0DER
```

如果浏览器再向同一个服务器做出第二个请求，它会在HTTP请求首部行中的Cookie行发回这个cookie，如下所示：

```
GET /index.html HTTP/1.1
Host: www.example.org
Cookie: cart=ATVPDKIKX0DER
Accept: text/html
```

只要服务器不重用cookie，这会使它在多个（否则无状态的）HTTP连接上跟踪各个用户和会话。

服务器可以设置不止一个cookie。例如，我向Amazon发出的一个请求在我的浏览器上设置了5个cookie：

```
Set-Cookie:skin=noskin
Set-Cookie:ubid-main=176-5578236-9590213
Set-Cookie:session-token=Zg6afPNqbaMv2WmYFOv57zCU1O6Ktr
Set-Cookie:session-id-time=2082787201l
Set-Cookie:session-id=187-4969589-3049309
```

除了简单的name=value对，cookie可以有多个属性来控制它们的作用域，包括过期日期、路径、域、端口、版本和安全选项。

例如，默认情况下，cookie来自哪个服务器就应用于哪个服务器。如果一个cookie由*www.foo.example.com*设置，浏览器就只把这个cookie发回给*www.foo.example.com*。不过网站也可以指示一个cookie应用于整个子域，而不只是最初的服务器。例如，下面这个请求为整个*foo.example.com*域设置一个用户cookie：

```
Set-Cookie: user=elharo;Domain=.foo.example.com
```

浏览器不只是把这个cookie回送到*www.foo.example.com*，还会发送给*lothar.foo.example.com*、*eliza.foo.example.com*、*enoch.foo.example.com*和*foo.example.com*域中的任何其他主机。不过，服务器只能为它直接所属的域设置cookie。*www.foo.example.com*就不能为*www.oreilly.com*、*example.com*或*.com*设置cookie，而不论它如何设置域。

---

**提示：** 一些网站把一个域的一个图像或其他内容嵌入在另一个域的一个页面中，从而绕过这个限制。所嵌入内容设置的cookie（而不是由页面本身设置）称为第三方cookie。很多用户会阻塞所有第三方cookie，出于保密的原因，一些Web浏览器也开始默认阻塞这些第三方cookie。

---

Cookie的作用域还受路径限制，所以会返回到服务器上的某些目录，而不是全部。默认作用域是最初的URL和所有子目录。例如，如果为URL *http://www.cafeconleche.org/XOM/*设置一个cookie，那么这个cookie还可以应用于*http://www.cafeconleche.org/XOM/apidocs/*，但不能应用于*http://www.cafeconleche.org/slides/*或*http://www.cafeconleche.org/*。不过，可以使用cookie中的Path属性改变默认作用域。例如，下面的响应为浏览器发送了一个名为“user”、值为“elharo”的cookie，它只应用于服务器的/restricted子树，而不能应用于网站的其他部分：

```
Set-Cookie: user=elharo; Path=/restricted
```

请求相同服务器上子树*/restricted*中的一个文档时，客户端会回显这个cookie。不过，它不会在网站的其他目录中使用这个cookie。

cookie可以同时包括域和路径。例如，下面的cookie应用于example.com域中任何服务器的*/restricted*路径：

```
Set-Cookie: user=elharo;Path=/restricted;Domain=.example.com
```

不同cookie属性的顺序无关紧要，只要全部都用分号隔开，cookie自己的名和值放在最

前面即可。不过，当客户端把cookie发送回服务器时，则并非如此。在这种情况下，路径必须在域前面，如下：

```
Cookie: user=elharo; Path=/restricted;Domain=.foo.example.com
```

通过将expires属性设置为“Wdy, DD-Mon-YYYY HH:MM:SS GMT”形式的一个日期，可以设置cookie在某个时间点过期。星期几（Wdy）和月份（Mon）以3字母缩写形式给出。其他部分都是数字，必要时要在前面补0。在java.text.SimpleDateFormat使用的模式语言中，格式为“E, dd-MMM-yyyy k:m:s 'GMT'”。例如，下面的cookie将在2015年12月21日下午3:23过期：

```
Set-Cookie: user=elharo; expires=Wed, 21-Dec-2015 15:23:00 GMT
```

在这个日期过去后，浏览器应当从其缓存中删除这个cookie。

Max-Age属性可以设置cookie经过一定秒数之后过期，而不是在特定时刻过期。例如，下面的cookie在第一次设置之后的一小时（3600秒）过期：

```
Set-Cookie: user="elharo"; Max-Age=3600
```

浏览器应当在经过了这段时间之后从其缓存中删除这个cookie。

因为cookie可能包含敏感信息，如口令和会话密钥，所以一些cookie事务应当是安全的。大多数情况下，这意味着使用HTTPS代替HTTP，不论表示什么，每个cookie都有一个没有值的secure属性，如：

```
Set-Cookie: key=etrogl7*;Domain=.foo.example.com; secure
```

浏览器应当拒绝通过非安全通道发送这种cookie。

为了针对cookie窃取攻击（如XSRF）提高安全性，cookie可以设置HttpOnly属性。这会告诉浏览器只通过HTTP和HTTPS返回cookie，特别强调不能由JavaScript返回：

```
Set-Cookie: key=etrogl7*;Domain=.foo.example.com; secure; httponly
```

这就是cookie在幕后的工作。下面是Amazon发送的一组完整的cookie：

```
Set-Cookie: skin=noskin; path=/; domain=.amazon.com;
 expires=Fri, 03-May-2013 21:46:43 GMT
Set-Cookie: ubid-main=176-5578236-9590213; path=/;
 domain=.amazon.com; expires=Tue, 01-Jan-2036 08:00:01 GMT
Set-Cookie: session-token=Zg6afPNqbaMv2WmYFOv57zCU1O6KtrMMdskcmllbZ
 cY4q6tOPrMywqO82PR6AgtfIJhtBABhomNUW2dITwuLfOZuhXILp7Toya+
 AvWaYJxpfY1lj4ci4cnJxiuUZTev1WV31p5bcwzRM1Cmn3QOCezNNqenhzZD8TZUnOL/9Ya;
 path=/; domain=.amazon.com; expires=Thu, 28-Apr-2033 21:46:43 GMT
Set-Cookie: session-id-time=2082787201l; path=/; domain=.amazon.com;
```

```
 expires=Tue, 01-Jan-2036 08:00:01 GMT
Set-Cookie: session-id=187-4969589-3049309; path=/; domain=.amazon.com;
 expires=Tue, 01-Jan-2036 08:00:01 GMT
```

Amazon希望在未来30～33年间，我的浏览器在向*amazon.com*域中的任何页面发出请求时同时发送这些cookie。当然，浏览器完全可以忽略所有这些请求，用户也可以在任何时刻删除或阻塞cookie。

## CookieManager

Java 5包括一个抽象类`java.net.CookieHandler`，它定义了存储和获取cookie的一个API，但不包括这个抽象类的实现，所以还有很多工作要做。Java 6进一步作了补充，为`CookieHandler`增加了一个可以使用的具体子类`java.net.CookieManager`。不过，默认情况下cookie并不打开。在Java存储和返回cookie之前，需要先启用cookie：

```
CookieManager manager = new CookieManager();
CookieHandler.setDefault(manager);
```

如果你想做的就是从网站接收cookie，再把它们发回给这些网站，那么你已经做到了。用这两行代码安装一个`CookieManager`之后，对于你用URL类连接的HTTP服务器，Java会存储这些服务器发送的所有cookie，再在后续的请求中向这些服务器发回所存储的cookie。不过，对于接收哪里发送的cookie，你可能还希望更谨慎一些。可以通过指定一个`CookiePolicy`来做到。已经预定义了3个策略：

- `CookiePolicy.ACCEPT_ALL`接受所有cookie。
- `CookiePolicy.ACCEPT_NONE`不接受任何cookie。
- `CookiePolicy.ACCEPT_ORIGINAL_SERVER`只接受第一方cookie。

例如，下面这个代码段告诉Java阻塞第三方cookie，但是接受第一方cookie：

```
CookieManager manager = new CookieManager();
manager.setCookiePolicy(CookiePolicy.ACCEPT_ORIGINAL_SERVER);
CookieHandler.setDefault(manager);
```

也就是说，它只接受与你对话的服务器发送的cookie，而不接受Internet上其他服务器发送的cookie。

如果你还希望有更细粒度的控制，例如，要接受某些已知域的cookie，而不接受其他域的cookie，可以自行实现`CookiePolicy`接口，并覆盖`shouldAccept()`方法：

```
public boolean shouldAccept(URI uri, HttpCookie cookie)
```

示例6-1显示了一个简单的CookiePolicy，它会阻塞来自.*gov*域的cookie，但是接受所有其他域的cookie。

示例6-1：阻塞所有.gov cookie但接受其他cookie的cookie策略

```
import java.net.*;

public class NoGovernmentCookies implements CookiePolicy {
  @Override
  public boolean shouldAccept(URI uri, HttpCookie cookie) {
    if (uri.getAuthority().toLowerCase().endsWith(".gov")
        || cookie.getDomain().toLowerCase().endsWith(".gov")) {
      return false;
    }
    return true;
  }
}
```

## CookieStore

有时有必要在本地存放和获取cookie。例如，一个应用退出时，可以把cookie库保存在磁盘上，下一次启动时再加载这些cookie。可以用getCookieStore()方法获取这个cookie库，CookieManager就在这里保存它的cookie：

```
CookieStore store = manager.getCookieStore();
```

CookieStore类允许你增加、删除和列出cookie，使你能控制在正常HTTP请求和响应流之外发送的cookie：

```
public void add(URI uri, HttpCookie cookie)
public List<HttpCookie> get(URI uri)
public List<HttpCookie> getCookies()
public List<URI> getURIs()
public boolean remove(URI uri, HttpCookie cookie)
public boolean removeAll()
```

这个库中的每个cookie都封装在一个HttpCookie对象中，它提供了一些方法来检查cookie的属性，示例6-2展示了这些方法。

示例6-2：HTTPCookie类

```
package java.net;

public class HttpCookie implements Cloneable {
  public HttpCookie(String name, String value)

  public boolean hasExpired()
  public void setComment(String comment)
  public String getComment()
  public void setCommentURL(String url)
  public String getCommentURL()
```

```
  public void setDiscard(boolean discard)
  public boolean getDiscard()
  public void setPortlist(String ports)
  public String getPortlist()
  public void setDomain(String domain)
  public String getDomain()
  public void setMaxAge(long expiry)
  public long getMaxAge()
  public void setPath(String path)
  public String getPath()
  public void setSecure(boolean flag)
  public boolean getSecure()
  public String getName()
  public void setValue(String value)
  public String getValue()
  public int getVersion()
  public void setVersion(int v)

  public static boolean domainMatches(String domain, String host)
  public static List<HttpCookie> parse(String header)

  public String toString()
  public boolean equals(Object obj)
  public int hashCode()
  public Object clone()
}
```

其中很多属性实际上已经不再使用。具体地，comment、commentURL、discard和version只用于现在已经过时的Cookie 2规范，而且不再更新。

# 第7章

# URLConnection

URLConnection是一个抽象类，表示指向URL指定资源的活动连接。URLConnection有两个不同但相关的用途。首先，与URL类相比，它对与服务器（特别是HTTP服务器）的交互提供了更多的控制。URLConnection可以检查服务器发送的首部，并相应地做出响应。它可以设置客户端请求中使用的首部字段。最后，URLConnection可以用POST、PUT和其他HTTP请求方法向服务器发回数据。这一章将研究所有这些技术。

其次，URLConnection类是Java的协议处理器（protocol handler）机制的一部分，这个机制还包括URLStreamHandler类。协议处理器的思想很简单：它们将处理协议的细节与处理特定数据类型分开，提供相应的用户接口，并完成完整Web浏览器所完成的其他操作。基类java.net.URLConnection是抽象类，要实现一个特定的协议，就要编写一个子类。这些子类可以在运行时由应用程序加载。例如，如果浏览器遇到一个有奇怪模式（如compress）的URL，它不会“绝望”地发出一个错误消息，而是会为这个未知协议下载一个协议处理器，用它与服务器通信。

java.net包中只有抽象的URLConnection类。具体子类都隐藏在sun.net包层次结构中。URLConnection的许多方法和字段以及一个构造函数都是保护的（protected）。换句话说，它们只能由URLConnection类或其子类的实例访问。很少会在源代码中直接实例化一个URLConnection对象。相反，运行时环境会根据所用的协议来创建所需的对象，然后使用java.lang.Class类的forName()和newInstance()方法实例化这个类（在编译时未知）。

**提示：** URLConnection在Java 类库中并没有最优设计的API。它存在的一个问题是，URLConnection与HTTP绑定过于紧密。例如，它假定传输的每个文件前面都有一个MIME首部或类似的东西。不过，大多数经典协议（如FTP和SMTP）并不使用MIME首部。

# 打开URLConnection

直接使用URLConnection类的程序遵循以下基本步骤：

1. 构造一个URL对象。
2. 调用这个URL对象的openConnection()获取一个对应该URL的URLConnection对象。
3. 配置这个URLConnection。
4. 读取首部字段。
5. 获得输入流并读取数据。
6. 获得输出流并写入数据。
7. 关闭连接。

并不一定执行所有这些步骤。例如，如果某种URL的默认设置是可以接受的，那么可能会跳过步骤3。如果只需要服务器的数据，不关心任何元信息，或者协议不提供任何元信息，就可以跳过步骤4。如果只希望接收服务器的数据，而不向服务器发送数据，就可以跳过步骤6。依据不同协议，步骤5和步骤6的顺序可能会反过来，或这两个步骤会交替出现。

URLConnection类仅有的一个构造函数为保护类型：

```
protected URLConnection(URL url)
```

因此，除非派生URLConnection的子类来处理新的URL类型（即编写一个协议处理器），否则要通过调用URL类的openConnection()方法来创建这样一个对象。例如：

```
try {
  URL u = new URL("http://www.overcomingbias.com/");
  URLConnection uc = u.openConnection();
  // 从URL读取...
} catch (MalformedURLException ex) {
  System.err.println(ex);
} catch (IOException ex) {
  System.err.println(ex);
}
```

URLConnection类声明为抽象类。不过，除了一个方法外，其余方法都已经实现。你会发现覆盖这个类的其他方法很方便，或者可能很有必要。必须由子类实现的一个方法是connect()，它建立与服务器的连接，因而依赖于服务类型（HTTP、FTP等）。例如，sun.net.www.protocol.file.FileURLConnection的connect()方法将URL转换为适当目录中的一个文件名，创建该文件的MIME信息，然后打开一个指向该文件的缓冲

FileInputStream。sun.net.www.protocol.http.HttpURLConnection的connect()方法会创建一个sun.net.www.http.HttpClient对象，由它负责连接服务器：

```
public abstract void connect() throws IOException
```

第一次构造URLConnection时，它是未连接的。也就是说，本地和远程主机无法发送和接收数据。没有socket连接这两个主机。connect()方法在本地和远程主机之间建立一个连接（一般使用TCP socket，但也可能通过其他机制来建立），这样就可以收发数据了。不过，对于getInputStream()、getContent()、getHeaderField()和其他要求打开连接的方法，如果连接尚未打开，它们就会调用connect()。因此，你很少需要直接调用connect()。

# 读取服务器的数据

下面是使用URLConnection对象从一个URL获取数据所需的最起码的步骤：

1. 构造一个URL对象。
2. 调用这个URL对象的openConnection()方法，获取对应该URL的URLConnection对象。
3. 调用这个URLConnection的getInputStream()方法。
4. 使用通常的流API读取输入流。

getInputStream()方法返回一个通用InputStream，可以读取和解析服务器发送的数据。示例7-1使用getInputStream()方法下载一个Web页面。

示例7-1：用URLConnection下载一个Web页面

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

public class SourceViewer2 {
  public static void main (String[] args) {
    if (args.length > 0) {
      try {
        // 打开URLConnection进行读取
        URL u = new URL(args[0]);
        URLConnection uc = u.openConnection();
        try (InputStream raw = uc.getInputStream()) { // 自动关闭
          InputStream buffer = new BufferedInputStream(raw);
          // 将InputStream串链到一个Reader
          Reader reader = new InputStreamReader(buffer);
          int c;
          while ((c = reader.read()) != -1) {
            System.out.print((char) c);
          }
```

```
        }
      } catch (MalformedURLException ex) {
        System.err.println(args[0] + " is not a parseable URL");
      } catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

这个程序与示例5-2几乎相同，这并非出于偶然。URL类的openStream()方法只是从它自己的URLConnection对象返回一个InputStream。输出也是一样，所以这里不再重复。

对于这个例子中这样一个简单的输入流，URL和URLConnection之间的区别并不明显。这两个类最大的不同在于：

- URLConnection提供了对HTTP首部的访问。
- URLConnection可以配置发送给服务器的请求参数。
- URLConnection除了读取服务器数据外，还可以向服务器写入数据。

# 读取首部

HTTP服务器在每个响应前面的首部中提供了大量信息。例如，下面是一个Apache Web服务器返回的一个典型的HTTP首部：

```
HTTP/1.1 301 Moved Permanently
Date: Sun, 21 Apr 2013 15:12:46 GMT
Server: Apache
Location: http://www.ibiblio.org/
Content-Length: 296
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

这里有大量信息。一般来说，HTTP首部可能包括所请求文档的内容类型、文档长度（字节数）、对内容编码所采用的字符集、日期时间、内容的过期时间及内容的最后修改日期。不过，具体有哪些信息依赖于服务器，有些服务器会为每个请求发送所有这些信息，有些则只发送其中一部分信息，还有一些则根本不发送任何信息。本节介绍的方法允许你查询URLConnection，来找出服务器提供了哪些元数据。

除了HTTP，很少有协议会使用MIME首部（从技术上讲，甚至HTTP首部实际上也不是MIME首部。它只是看起来像而已）。编写你自己的URLConnection子类时，通常必须覆盖这些方法，才能返回有意义的值。你缺少的最重要的一个信息可能是内容类型。URLConnection提供了一些工具方法，可以根据文件名或数据本身最前面的一些字节猜测出数据的内容类型。

## 获取指定的首部字段

前6个方法可以请求首部中特定的常用字段。包括：

- Content-type
- Content-length
- Content-encoding
- Date
- Last-modified
- Expires

### public String getContentType()

getContentType()方法返回响应主体的MIME内容类型。它依赖于Web服务器来发送一个有效的内容类型。如果没有提供内容类型，它不会抛出异常，而是返回null。连接Web服务器时，text/html是你最常遇到的内容类型。其他常用的类型包括text/plain、image/gif、application/xml和image/jpeg。

如果内容类型是某种形式的文本，那么这个首部可能还包含一个字符集部分，来标识文档的字符编码方式。例如：

```
Content-type: text/html; charset=UTF-8
```

或者：

```
Content-Type: application/xml; charset=iso-2022-jp
```

在这种情况下，getContentType()将返回Content-type字段的完整的值，包括字符编码方式。利用这一点，可以使用HTTP首部中指定的编码方式对文档解码，以改进示例7-1，如果没有指定编码方式，就使用ISO-8859-1（HTTP的默认编码方式）。如果遇到的是非文本类型，就会抛出一个异常。如示例7-2所示。

示例7-2：用正确的字符集下载一个Web页面

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

public class EncodingAwareSourceViewer {

  public static void main (String[] args) {
    for (int i = 0; i < args.length; i++) {
      try {
        // 设置默认编码方式
        String encoding = "ISO-8859-1";
```

```
            URL u = new URL(args[i]);
            URLConnection uc = u.openConnection();
            String contentType = uc.getContentType();
            int encodingStart = contentType.indexOf("charset=");
            if (encodingStart != -1) {
                encoding = contentType.substring(encodingStart + 8);
            }
            InputStream in = new BufferedInputStream(uc.getInputStream());
            Reader r = new InputStreamReader(in, encoding);
            int c;
            while ((c = r.read()) != -1) {
                System.out.print((char) c);
            }
            r.close();
          } catch (MalformedURLException ex) {
            System.err.println(args[0] + " is not a parseable URL");
          } catch (UnsupportedEncodingException ex) {
            System.err.println(
                "Server sent an encoding Java does not support: " + ex.getMessage());
          } catch (IOException ex) {
            System.err.println(ex);
          }
        }
      }
    }
```

## public int getContentLength()

getContentLength()方法告诉你内容中有多少字节。如果没有Content-length首部，getContentLength()就返回－1。这个方法不抛出异常。如果需要准确地知道要读取的字节数，或者需要预先创建一个足够大的缓冲区来保存数据，就可以使用这个方法。

随着网络变得越来越快，文件变得越来越大，实际上很有可能发现资源的大小超过最大的int值（约21亿字节）。在这种情况下，getContentLength()会返回–1。Java 7增加了一个getContentLengthLong()方法，它的做法与getContentLength()类似，只不过它会返回一个long而不是int，这样就可以处理更大的资源：

```
public int getContentLengthLong() // Java 7
```

第5章讨论了如何使用URL类的openStream()方法从HTTP服务器下载文本文件。虽然在理论上可以使用同样的方法来下载二进制文件，如GIF图片或*.class*字节码文件，但在实际中这会产生一个问题。HTTP服务器并不总是会在数据发送完后就立即关闭连接，因此，你不知道何时停止读取。要下载一个二进制文件，更可靠的做法是使用URLConnection的getContentLength()方法来得到文件的长度，然后根据这个长度读取相应的字节数。示例7-3的程序就使用了这个技术，它将一个二进制文件保存到磁盘中。

示例7-3：从Web网站下载二进制文件并保存到磁盘

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

public class BinarySaver {

  public static void main (String[] args) {
    for (int i = 0; i < args.length; i++) {
      try {
        URL root = new URL(args[i]);
        saveBinaryFile(root);
      } catch (MalformedURLException ex) {
        System.err.println(args[i] + " is not URL I understand.");
      } catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }

  public static void saveBinaryFile(URL u) throws IOException {
    URLConnection uc = u.openConnection();
    String contentType = uc.getContentType();
    int contentLength = uc.getContentLength();
    if (contentType.startsWith("text/") || contentLength == -1 ) {
      throw new IOException("This is not a binary file.");
    }

    try (InputStream raw = uc.getInputStream()) {
      InputStream in = new BufferedInputStream(raw);
      byte[] data = new byte[contentLength];
      int offset = 0;
      while (offset < contentLength) {
        int bytesRead = in.read(data, offset, data.length - offset);
        if (bytesRead == -1) break;
        offset += bytesRead;
      }

      if (offset != contentLength) {
        throw new IOException("Only read " + offset
            + " bytes; Expected " + contentLength + " bytes");
      }
      String filename = u.getFile();
      filename = filename.substring(filename.lastIndexOf('/') + 1);
      try (FileOutputStream fout = new FileOutputStream(filename)) {
        fout.write(data);
        fout.flush();
      }
    }
  }
}
```

与平常一样，main()方法循环处理命令行输入的URL，将各个URL传递给saveBinaryFile()方法。saveBinaryFile()打开一个指向URL的URLConnection uc。

它将内容类型保存在变量contentType中，内容长度保存在变量contentLength中。接下来，由一个if语句检查内容类型是否为text，以及是否缺少Content-length字段或Content-length字段是否无效（contentLength==-1）。如果其中任意一个判断为true，则抛出IOException异常。如果都为false，那就得到一个长度已知的二进制文件：这正是我们所希望的。

既然已经有了一个真正的二进制文件，下面准备将它读取到字节数组data中。data的大小初始化为contentLength，即保存这个二进制对象所需的字节数。理想情况下，你可能想通过一个read()调用来填充data数组，但有可能不会一次得到所有字节，所以把读取放在一个循环中。到目前为止读取的字节数会累加到offset变量中，它会跟踪data数组中开始放置数据的位置，下一次调用read()获得的数据就要从这个位置开始放置。这个循环一直继续，直到offset等于或大于contentLength为止。也就是说，数组中已经填入了期望的字节数。如果read()返回-1，也会跳出while循环，这表示流意外结束。offset变量现在就包含已经读取的总的字节数，这应当与内容长度相等。如果它们不相等，就表示发生了错误，所以saveBinaryFile()会抛出一个IOException异常。这就是从HTTP连接读取二进制文件的一般过程。

现在可以准备将数据保存到文件了。saveBinaryFile()使用getFile()方法从URL得到文件名，调用filename.substring(theFile.lastIndexOf('/') +1)去掉路径信息。为这个文件打开一个新的FileOutputStream fout，然后利用fout.write(b)将数据一次性全部写入。AutoCloseable用于完成清理。

### public String getContentEncoding()

getContentEncoding()方法返回一个String，指出内容是如何编码的。如果发送的内容没有编码（对于HTTP服务器这很常见），这个方法就返回null。它不抛出异常。Web上最常用的内容编码方式可能是x-gzip，它可以使用java.util.zip.GZipInputStream直接解码。

---

**提示：**内容编码方式与字符编码方式不同。字符编码方式由Content-type首部或文档内部的信息确定，指出如何将字符编码为字节。内容编码方式则指出字节如何编码为其他字节。

---

### public long getDate()

getDate()方法返回一个long，指出文档何时发送，这个时间按自格林尼治标准时间（GMT）1970年1月1日子夜后过去了多少毫秒来给出。可以把它转换为一个java.util.Date。例如：

```
Date documentSent = new Date(uc.getDate());
```

这是从服务器角度所看到的发送文档的时间，可能与本地机器的时间不一致。如果HTTP首部不包括Data字段，getDate()就返回0。

### public long getExpiration()

有些文档有基于服务器的过期日期，指示应当何时从缓存中删除文档，并从服务器重新下载。getExpiration()与getDate()很类似，区别只在于如何解释返回的值。它返回一个long，指示自GMT 1970年1月1日子夜12:00后的毫秒数，文档在这一时刻过期。如果HTTP首部没有包括Expiration字段，getExpiration()就返回0，这表示文档不会过期，将永远保留在缓存中。

### public long getLastModified()

最后一个日期方法getLastModified()返回文档的最后修改日期。再次说明，这个日期是按GMT 1970年1月1日子夜后的毫秒数给出的。如果HTTP首部没有包括Last-modified字段（这很常见），这个方法就返回0。

示例7-4从命令行读取URL，使用以上6个方法显示其内容类型、内容长度、内容编码方式、最后修改日期、过期日期和当前日期。

示例7-4：返回首部

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

public class HeaderViewer {

  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[0]);
        URLConnection uc = u.openConnection();
        System.out.println("Content-type: " + uc.getContentType());
        if (uc.getContentEncoding() != null) {
            System.out.println("Content-encoding: "
                 + uc.getContentEncoding());
        }
        if (uc.getDate() != 0) {
            System.out.println("Date: " + new Date(uc.getDate()));
        }
        if (uc.getLastModified() != 0) {
            System.out.println("Last modified: "
                 + new Date(uc.getLastModified()));
        }
        if (uc.getExpiration() != 0) {
            System.out.println("Expiration date: "
                 + new Date(uc.getExpiration()));
        }
```

```
        if (uc.getContentLength() != -1) {
            System.out.println("Content-length: " + uc.getContentLength());
        }
      } catch (MalformedURLException ex) {
        System.err.println(args[i] + " is not a URL I understand");
      } catch (IOException ex) {
        System.err.println(ex);
      }
        System.out.println();
    }
  }
}
```

下面是用这个程序查看*http://www.oreilly.com*时的输出：

```
% java HeaderViewer http://www.oreilly.com
Content-type: text/html; charset=utf-8
Date: Fri May 31 18:08:09 EDT 2013
Last modified: Fri May 31 17:04:14 EDT 2013
Expiration date: Fri May 31 22:08:09 EDT 2013
Content-length: 83273
```

*http://www.oreilly.com*上文件的内容类型是`text/html`。这里没有使用内容编码方式。文件是在东部夏令时（EDT）2013年5月31日星期五下午6:08发送的。最后修改时间是同一天下午的5:04，而且将于未来4小时后过期。

这里没有Content-length首部。很多服务器并不会费力地为文本文件提供Content-length首部。不过，对于二进制文件，一定要发送Content-length首部。下面是请求GIF图像*http://www.oreilly.com/favicon.ico*时得到的HTTP首部。现在服务器发送了一个Content-length首部，值为2294。

```
% java HeaderViewer http://oreilly.com/favicon.ico
Content-type: image/x-icon
Date: Fri May 31 18:16:01 EDT 2013
Last modified: Wed Mar 26 19:14:36 EST 2003
Expiration date: Fri Jun 07 18:16:01 EDT 2013
Content-length: 2294
```

## 获取任意首部字段

以上6个方法请求首部中的指定字段，不过对于一个消息中能包含多少个首部字段理论上并没有限制。下面的5个方法可以检查首部中的任意字段。事实上，上一节中的方法只是这里讨论的方法的简单包装器，可以用这些方法来获得Java设计者不打算提供的首部字段。如果找到了所请求的首部，就会返回这个首部。否则方法将返回`null`。

### public String getHeaderField(String name)

`getHeaderField()`方法返回指定首部字段的值。首部的名不区分大小写，也不包含结束

冒号。例如，为获得URLConnection对象uc的Content-type和Content-encoding首部字段的值，可以写以下代码：

```
String contentType = uc.getHeaderField("content-type");
String contentEncoding = uc.getHeaderField("content-encoding"));
```

为得到Date、Content-length和Expires首部，同样地可以写以下代码：

```
String data = uc.getHeaderField("date");
String expires = uc.getHeaderField("expires");
String contentLength = uc.getHeaderField("Content-length");
```

这些方法都返回String，而不是像上一节中getContentLength()、getExpirationDate()、getLastModified()和getDate()方法那样返回int或long。如果想要数字值，可以将String转换为long或int。

不要假定getHeaderField()返回的值一定有效。必须进行检查确保它不是null。

### public String getHeaderFieldKey(int n)

这个方法返回第*n*个首部字段（如Content-length或Server）的键（即字段名）。请求方法本身是第0个首部，它的键为null。第一个首部即编号为1。例如，为了得到URLConnection uc的第6个首部键，可以写作：

```
String header6 = uc.getHeaderFieldKey(6);
```

### public String getHeaderField(int n)

这个方法返回第*n*个首部字段的值。在HTTP中，包含请求方法和路径的起始行是第0个首部字段，实际的第1个首部编号为1。示例7-5使用这个方法并结合getHeaderFieldKey()来显示整个HTTP首部。

示例7-5：显示整个HTTP首部

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

public class AllHeaders {

  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        URLConnection uc = u.openConnection();
        for (int j = 1; ; j++) {
          String header = uc.getHeaderField(j);
          if (header == null) break;
          System.out.println(uc.getHeaderFieldKey(j) + ": " + header);
```

```
        }
      } catch (MalformedURLException ex) {
        System.err.println(args[i] + " is not a URL I understand.");
      } catch (IOException ex) {
        System.err.println(ex);
      }
        System.out.println();
    }
  }
}
```

例如，下面是在对*http://www.oreilly.com*运行这个程序时的输出：

```
% java AllHeaders http://www.oreilly.com
Date: Sat, 04 May 2013 11:28:26 GMT
Server: Apache
Last-Modified: Sat, 04 May 2013 07:35:04 GMT
Accept-Ranges: bytes
Content-Length: 80366
Content-Type: text/html; charset=utf-8
Cache-Control: max-age=14400
Expires: Sat, 04 May 2013 15:28:26 GMT
Vary: Accept-Encoding
Keep-Alive: timeout=3, max=100
Connection: Keep-Alive
```

除了与获取方法名字对应的首部，这个服务器还提供了Server、Accept-Ranges、Cache-control、Vary、Keep-Alive和Connection首部。其他服务器可能有不同的首部集。

### public long getHeaderFieldDate(String name, long default)

这个方法首先获取由name参数指定的首部字段，然后尝试将这个字符串转换为一个long，指示自GMT 1970年1月1日子夜后经过的毫秒数。getHeaderFieldDate()可以用来获取表示日期的首部字段（例如，Expires、Date或Last-modified首部）。为了将字符串转换为整数，getHeaderFieldDate()使用了java.util.Date的parseDate()方法。parseDate()方法在理解和转换大多数常见日期格式上做得非常好，但也会被难住。例如，如果请求一个并非包含日期的首部字段，此时parseDate()方法就无能为力了。如果parseDate()不理解这个日期或者getHeaderFieldDate()无法找到所请求的首部字段，getHeaderFieldDate()就返回default参数。例如：

```
Date expires = new Date(uc.getHeaderFieldDate("expires", 0));
long lastModified = uc.getHeaderFieldDate("last-modified", 0);
Date now = new Date(uc.getHeaderFieldDate("date", 0));
```

可以使用java.util.Date类的方法将long转换为String。

### public int getHeaderFieldInt(String name, int default)

这个方法获取首部字段name的值，尝试将其转换为int。如果失败，可能是因为无法找到所请求的首部字段，也可能因为该字段不包含一个可识别的整数，getHeaderFieldInt()都会返回default参数。这个方法通常用于获取Content-length字段。例如，为了得到URLConnection uc的内容长度，可以编写以下代码：

```
int contentLength = uc.getHeaderFieldInt("content-length", -1);
```

在这个代码段中，如果没有提供Content-length，getHeaderFieldInt()则返回－1。

## 缓存

Web浏览器多年来一直在缓存页面和图片。如果一个logo在网站的每一个页面上重复出现，浏览器一般只会从远程服务器上加载一次，将它保存在缓存中，每次需要的时候会从缓存重新加载，而不是每次遇到这个logo都请求从远程服务器加载。一些HTTP首部（包括Expires和Cache-Control）可以控制缓存。

默认情况下，一般认为使用GET通过HTTP访问的页面可以缓存，也应当缓存。使用HTTPS或POST访问的页面通常不应缓存。不过，HTTP首部可以对此做出调整：

- Expires首部（主要针对HTTP 1.0）指示可以缓存这个资源表示，直到指定的时间为止。
- Cache-control首部（HTTP 1.1）提供了细粒度的缓存策略：
  - max-age=[seconds]：从现在直到缓存项过期之前的秒数。
  - s-maxage=[seconds]：从现在起，直到缓存项在共享缓存中过期之前的秒数。私有缓存可以将缓存项保存更长时间。
  - public：可以缓存一个经过认证的响应。否则已认证的响应不能缓存。
  - private：仅单个用户缓存可以保存响应，而共享缓存不应保存。
  - no-cache：这个策略的作用与名字不太一致。缓存项仍然可以缓存，不过客户端在每次访问时要用一个Etag或Last-modified首部重新验证响应的状态。
  - no-store：不管怎样都不缓存。

  如果Cache-control和Expires首部都出现，Cache-control会覆盖Expires。服务器可以在一个首部中发送多个Cache-control首部，只要它们没有冲突。
- Last-modified首部指示资源最后一次修改的日期。客户端可以使用一个HEAD请求来

检查这个日期，只有当本地缓存的副本早于Last-modified日期时，它才会真正执行GET来获取资源。

- Etag首部（HTTP 1.1）是资源改变时这个资源的唯一标识符。客户端可以使用一个HEAD请求来检查这个标识符，只有当本地缓存的副本有一个不同的ETag时，它才会真正执行GET来获取资源。

例如，下面这个HTTP响应指出，这个资源可以缓存604 800秒（HTTP 1.1）或者缓存到一周以后（HTTP 1.0）。它还指出资源的最后修改日期是4月20日，而且有一个Etag，所以如果本地缓存已经有一个更新的副本，那么现在没有必要加载整个文档：

```
HTTP/1.1 200 OK
Date: Sun, 21 Apr 2013 15:12:46 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Cache-control: max-age=604800
Expires: Sun, 28 Apr 2013 15:12:46 GMT
Last-modified: Sat, 20 Apr 2013 09:55:04 GMT
ETag: "67099097696afcf1b67e"
```

示例7-6给出一个简单的Java类，可以解析和查询Cache-control首部。

示例7-6：如何检查Cache-control首部

```
import java.util.Date;
import java.util.Locale;


public class CacheControl {

  private Date maxAge = null;
  private Date sMaxAge = null;
  private boolean mustRevalidate = false;
  private boolean noCache = false;
  private boolean noStore = false;
  private boolean proxyRevalidate = false;
  private boolean publicCache = false;
  private boolean privateCache = false;

  public CacheControl(String s) {
    if (s == null || !s.contains(":")) {
      return; // 默认策略
    }

    String value = s.split(":")[1].trim();
    String[] components = value.split(",");

    Date now = new Date();
    for (String component : components) {
      try {
        component = component.trim().toLowerCase(Locale.US);
```

```
        if (component.startsWith("max-age=")) {
          int secondsInTheFuture = Integer.parseInt(component.substring(8));
          maxAge = new Date(now.getTime() + 1000 * secondsInTheFuture);
        } else if (component.startsWith("s-maxage=")) {
          int secondsInTheFuture = Integer.parseInt(component.substring(8));
          sMaxAge = new Date(now.getTime() + 1000 * secondsInTheFuture);
        } else if (component.equals("must-revalidate")) {
          mustRevalidate = true;
        } else if (component.equals("proxy-revalidate")) {
          proxyRevalidate = true;
        } else if (component.equals("no-cache")) {
          noCache = true;
        } else if (component.equals("public")) {
          publicCache = true;
        } else if (component.equals("private")) {
          privateCache = true;
        }
      } catch (RuntimeException ex) {
        continue;
      }
    }
  }

  public Date getMaxAge() {
    return maxAge;
  }

  public Date getSharedMaxAge() {
    return sMaxAge;
  }

  public boolean mustRevalidate() {
    return mustRevalidate;
  }

  public boolean proxyRevalidate() {
    return proxyRevalidate;
  }

  public boolean noStore() {
    return noStore;
  }

  public boolean noCache() {
    return noCache;
  }

  public boolean publicCache() {
    return publicCache;
  }

  public boolean privateCache() {
    return privateCache;
  }
}
```

客户端可以充分利用这个信息：

- 如果本地缓存中有这个资源的一个表示，而且还没有到它的过期时间，那么可以直接使用这个资源，而无需与服务器交互。
- 如果本地缓存中有这个资源的一个表示，不过已经到它的过期时间，在完成完整的GET之前，可以检查服务器的HEAD首部，查看资源是否已经改变。

## Java的Web缓存

默认情况下，Java并不完成缓存。要安装URL类使用的系统级缓存，需要有：

- ResponseCache的一个具体子类。
- CacheRequest的一个具体子类。
- CacheResponse的一个具体子类。

要安装你的ResponseCache子类来处理你的CacheRequest和CacheResponse子类，需要把它传递到静态方法ResponseCache.setDefault()。这会把这个缓存对象安装为系统的默认缓存。Java虚拟机只支持一个共享缓存。

一旦安装了缓存，只要系统尝试加载一个新的URL，它首先会在这个缓存中查找。如果缓存返回了所要的内容，URLConnection就不需要与远程服务器连接。不过，如果所请求的数据不在缓存中，协议处理器将从远程服务器下载相应数据。完成之后，它会把这个响应放在缓存中，使得下一次加载这个URL时，可以很快从缓存中得到这个内容。

ResponseCache提供了两个抽象方法，可以存储和获取系统缓存中的数据：

```
public abstract CacheResponse get(URI uri, String requestMethod,
    Map<String, List<String>> requestHeaders) throws IOException
public abstract CacheRequest put(URI uri, URLConnection connection)
    throws IOException
```

put()方法返回一个CacheRequest对象，它包装了一个OutputStream，URL将把读取的可缓存数据写入这个输出流。CacheRequest是一个抽象类，它有两个方法，如示例7-7所示。

示例7-7：CacheRequest类

```
package java.net;

public abstract class CacheRequest {
  public abstract OutputStream getBody() throws IOException;
  public abstract void abort();
}
```

子类中的getOutputStream()方法应当返回一个OutputStream，指向缓存中的“数据库”，这个数据库与同时传入put()方法的URI对应。例如，如果数据存储在一个文件中，就要返回连接到该文件的FileOutputStream。协议处理器会把读取的数据复制到这个OutputStream。如果复制时出现问题（例如，服务器意外地关闭了连接），协议处理器就会调用abort()方法。这个方法应当从缓存删除为这个请求存储的所有数据。

示例7-8展示了一个基本CacheRequest子类，它会传回一个ByteArrayOutputStream。其后，可以用getData()方法获取数据，这是这个子类中的一个定制方法，只获取Java向这个类提供的OutputStream中写入的数据。另外显然还有一个候选的策略，可以把结果存储在文件中，并使用一个FileOutputStream。

示例7-8：CacheRequest的一个具体子类

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

public class SimpleCacheRequest extends CacheRequest {

  private ByteArrayOutputStream out = new ByteArrayOutputStream();

  @Override
  public OutputStream getBody() throws IOException {
    return out;
  }

  @Override
  public void abort() {
    out.reset();
  }

  public byte[] getData() {
    if (out.size() == 0) return null;
    else return out.toByteArray();
  }
}
```

ResponseCache的get()方法从缓存中获取数据和首部，包装在CacheResponse对象中返回。如果所需的URI不在缓存中，则返回null，在这种情况上，协议处理器会正常地从远程服务器加载这个URI。再次说明，这是一个抽象类，你必须在子类中具体实现。示例7-9展示了这个类。它有两个方法，一个返回请求的数据，另一个返回首部。缓存最初的响应时，数据和首部都要存储。首部要以一个不可修改的映射返回，它的键是HTTP首部字段名，值是每个指定HTTP首部的值列表。

示例7-9：CacheResponse类

```
public abstract class CacheResponse {
  public abstract Map<String, List<String>> getHeaders() throws IOException;
  public abstract InputStream getBody() throws IOException;
```

```
}
```

示例7-10展示了一个简单的CacheResponse子类，它绑定到一个SimpleCacheRequest和一个CacheControl。在这个例子中，共享引用将数据从请求类传递给响应类。如果在文件中存储响应，就需要共享文件名。除了SimpleCacheRequest对象（要从中读取数据），还必须将最初的URLConnection对象传递给构造函数。这个对象用来读取HTTP首部，可以存储这个首部以备以后获取。这个对象还记录了服务器为资源的缓存表示提供的过期时间和缓存控制策略（如果有的话）。

示例7-10：CacheResponse的一个具体子类

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

public class SimpleCacheResponse extends CacheResponse {

  private final Map<String, List<String>> headers;
  private final SimpleCacheRequest request;
  private final Date expires;
  private final CacheControl control;

  public SimpleCacheResponse(
      SimpleCacheRequest request, URLConnection uc, CacheControl control)
      throws IOException {

    this.request = request;
    this.control = control;
    this.expires = new Date(uc.getExpiration());
    this.headers = Collections.unmodifiableMap(uc.getHeaderFields());
  }

  @Override
  public InputStream getBody() {
    return new ByteArrayInputStream(request.getData());
  }

  @Override
  public Map<String, List<String>> getHeaders()
      throws IOException {
      return headers;
  }

  public CacheControl getControl() {
    return control;
  }

  public boolean isExpired() {
    Date now = new Date();
    if (control.getMaxAge().before(now)) return true;
    else if (expires != null && control.getMaxAge() != null) {
      return expires.before(now);
```

```
    } else {
      return false;
    }
  }
}
```

最后需要一个简单的ResponseCache子类，从而在请求时存储和获取缓存的值，同时还要注意原来的Cache-control首部。示例7-11展示了这样一个简单的类，它使用一个很大的线程安全HashMap将有限数量的响应存储在内存中。这个类很适合单用户的私有缓存（因为它忽略了Cache-control的私有和公共属性）。

示例7-11：内存中ResponseCache

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

public class MemoryCache extends ResponseCache {

  private final Map<URI, SimpleCacheResponse> responses
      = new ConcurrentHashMap<URI, SimpleCacheResponse>();
  private final int maxEntries;

  public MemoryCache() {
    this(100);
  }

  public MemoryCache(int maxEntries) {
    this.maxEntries = maxEntries;
  }

  @Override
  public CacheRequest put(URI uri, URLConnection conn)
      throws IOException {

      if (responses.size() >= maxEntries) return null;

      CacheControl control = new CacheControl(conn.getHeaderField( "Cache-Control" ));
      if (control.noStore()) {
        return null;
      } else if (!conn.getHeaderField(0).startsWith("GET")) {
        // 只缓存GET
        return null;
      }

      SimpleCacheRequest request = new SimpleCacheRequest();
      SimpleCacheResponse response = new SimpleCacheResponse(request, conn, control);

      responses.put(uri, response);
      return request;
    }
```

```
  @Override
  public CacheResponse get(URI uri, String requestMethod,
      Map<String, List<String>> requestHeaders)
      throws IOException {

    if ("GET".equals(requestMethod)) {
      SimpleCacheResponse response = responses.get(uri);
      // 检查过期时间
      if (response != null && response.isExpired()) {
        responses.remove(response);
        response = null;
      }
      return response;
    } else {
      return null;
    }
  }
}
```

Java要求一次只能有一个URL缓存。要安装或改变缓存，需要使用静态方法ResponseCache.setDefault()和ResponseCache.getDefault()：

```
public static ResponseCache getDefault()
public static void setDefault(ResponseCache responseCache)
```

这些方法会设置同一个Java虚拟机中运行的所有程序所使用的缓存。例如，下面这行代码会在应用中安装示例7-11：

```
ResponseCache.setDefault(new MemoryCache());
```

一旦安装了类似示例7-11的ResponseCache，HTTP URLConnections就会一直使用这个缓存。

获取的每个资源会一直保留在HashMap中，直到它过期。在这个示例中，将过期的文档从缓存删除之前，会一直等待再次请求这个文档。更复杂的实现可能会使用一个低优先级的线程来扫描过期文档，将其删除，为其他文档腾出空间。如果不这样，或者除了这样做之外，实现还可以将资源的表示缓存在一个队列中，在必要时删除最老的文档或最接近过期时间的文档，为新文档腾出空间。甚至更复杂的实现还可以跟踪所存储的各个文档的访问频率，并且只删除最老和最少使用的文档。

我已经提到，可以在文件系统之上而不是在Java Collections API之上实现缓存，还能将缓存存储在数据库中，另外可以做很多不太常见的事情。例如，可以将某些URL的请求重定向到本地服务器，而不是位于世界另一半的某个远程服务器，其实就是把本地Web服务器当作了缓存。或者ResponseCache可以在启动时加载一组固定的文件，然后只从内存提供这些文件。这对于处理很多不同SOAP请求的服务器很有用，遵循常见模式的

请求都可以存储在缓存中。抽象的ResponseCache类非常灵活，完全可以支持所有这些以及其他的使用模式。

# 配置连接

URLConnection类有7个保护的实例字段，定义了客户端如何向服务器做出请求。这些字段包括：

```
protected URL     url;
protected boolean doInput = true;
protected boolean doOutput = false;
protected boolean allowUserInteraction = defaultAllowUserInteraction;
protected boolean useCaches = defaultUseCaches;
protected long    ifModifiedSince = 0;
protected boolean connected = false;
```

例如，如果doOutput为true，那么除了通过这个URLConnection读取数据外，还可以将数据写入到服务器。如果useCaches为false，连接会绕过所有本地缓存，重新从服务器下载文件。

由于这些字段都是保护字段，所以它们的值要通过相应的设置方法和获取方法来访问和修改：

```
public URL     getURL()
public void    setDoInput(boolean doInput)
public boolean getDoInput()
public void    setDoOutput(boolean doOutput)
public boolean getDoOutput()
public void    setAllowUserInteraction(boolean allowUserInteraction)
public boolean getAllowUserInteraction()
public void    setUseCaches(boolean useCaches)
public boolean getUseCaches()
public void    setIfModifiedSince(long ifModifiedSince)
public long    getIfModifiedSince()
```

只能在URLConnection连接之前修改这些字段（试图从连接读取内容或首部之前）。对于设置字段的方法，如果调用这些方法时连接已经打开，大多数方法会抛出一个IllegalStateException异常。一般情况下，只能在连接打开前设置URLConnection对象的属性。

还有一些获取方法和设置方法定义了所有URLConnection实例的默认行为。它们包括：

```
public boolean                getDefaultUseCaches()
public void                   setDefaultUseCaches(boolean defaultUseCaches)
public static void            setDefaultAllowUserInteraction(
    boolean defaultAllowUserInteraction)
public static boolean         getDefaultAllowUserInteraction()
```

```
public static FileNameMap     getFileNameMap()
public static void            setFileNameMap(FileNameMap map)
```

与实例方法不同，这些方法可以在任何时候调用。新的默认值只应用于设置这些新默认值之后构造的URLConnection对象。

## protected URL url

url字段指定了这个URLConnection连接的URL。构造函数会在创建URLConnection时设置这个字段，此后不能再改变。可以通过调用getURL()方法获取这个字段的值。示例7-12打开一个指向*http://www.oreilly.com/*的URLConnection，获取这个连接的URL，并显示。

示例7-12：显示指向http://www.oreilly.com/的URLConnection的URL

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

public class URLPrinter {

  public static void main(String[] args) {
    try {
      URL u = new URL("http://www.oreilly.com/");
      URLConnection uc = u.openConnection();
      System.out.println(uc.getURL());
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

结果如下，应该没有什么意外。显示的URL就是创建URLConnection所用的URL。

```
% java URLPrinter
http://www.oreilly.com/
```

## protected boolean connected

如果连接已经打开，boolean字段connected为true，如果连接关闭，这个字段则为false。由于在创建一个新URLConnection对象时连接尚未打开，所以其初始值为false。这个变量只能由java.net.URLConnection及其子类的实例访问。

没有直接读取或改变connected值的方法。不过，任何导致URLConnection连接的方法都会将这个变量设置为true，包括connect()、getInputStream()和getOutputStream()。任何导致URLConnection断开连接的方法会把这个字段设置为false。java.net.URLConnection中没有这样的方法，但它的一些子类（如java.net.HttpURLConnection）有这样的disconnect()方法。

如果要派生URLConnection子类来编写一个协议处理器，你要负责在连接时将connected设置为true，另外在连接关闭时要将它设置为false。java.net.URLConnection中的许多方法会读取这个变量，来确定能做哪些操作。如果未能正确地设置，程序将可能出现严重的bug而且很难诊断。

## protected boolean allowUserInteraction

有些URLConnection需要与用户交互。例如，Web浏览器可能需要询问用户名和口令。不过，很多应用程序不能假定真实存在一个可以与它交互的用户。例如，搜索引擎机器人可能在后台运行，并没有用户来提供用户名和口令。顾名思义，allowUserInteraction字段指示了是否允许用户交互。默认值是false。

这个变量是保护类型变量，不过公共方法getAllowUserInteraction()可以读取它的值，公共方法setAllowUserInteraction()可以修改它的值：

```
public void    setAllowUserInteraction(boolean allowUserInteraction)
public boolean getAllowUserInteraction()
```

值为true表示允许用户交互，false表示没有用户交互。这个值可以在任何时候读取，但只能在URLConnection连接前设置。如果在URLConnection已连接时调用setAllowUserInteraction()，会抛出IllegalStateException异常。

例如，下面的代码段打开一个连接，在需要时会要求用户身份认证：

```
URL u = new URL("http://www.example.com/passwordProtectedPage.html");
URLConnection uc = u.openConnection();
uc.setAllowUserInteraction(true);
InputStream in = uc.getInputStream();
```

Java 没有提供向用户询问用户名和口令的默认GUI。如果从一个applet做出请求，可能会依靠浏览器通常的身份认证对话框。在独立的应用程序中，首先需要安装一个Authenticator，如第5章“访问口令保护的网站”一节所述。

图7-1展示了尝试访问一个受口令保护的页面时弹出的对话框。如果取消这个对话框，会得到一个401 Authorization Required（401要求授权）错误，以及服务器可能发送给未授权用户的一些文本。不过，如果拒绝发送授权信息，可以单击OK，然后在问你是否希望重新尝试授权时回答No，getInputStream()就会抛出一个ProtocolException异常。

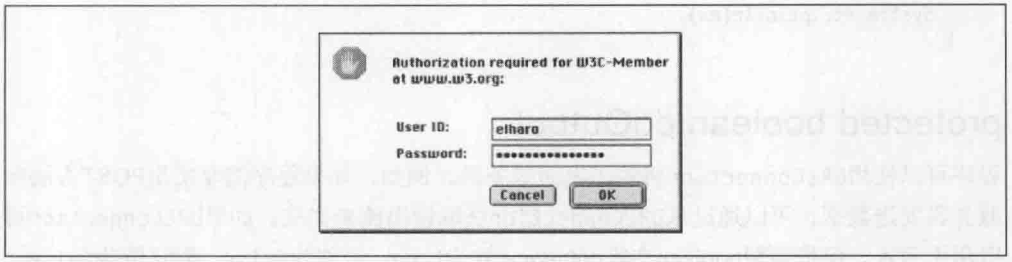


图7-1：身份认证对话框

静态方法getDefaultAllowUserInteraction()和setDefaultAllowUserInteraction()确定了URLConnection对象在没有显式设置allowUserInteraction时的默认行为。由于allowUserInteraction字段是静态的（也就是说，这是一个类变量而非实例变量），所以如果设置这个字段，会改变调用setDefaultAllowUserInteraction()之后创建的所有URLConnection类实例的默认行为。

例如，下面的代码段用getDefaultAllowUserInteraction()检查默认情况下是否允许用户交互。如果默认不允许用户交互，代码就使用setDefaultAllowUserInteraction()将允许用户交互设置为默认行为：

```
if (!URLConnection.getDefaultAllowUserInteraction()) {
  URLConnection.setDefaultAllowUserInteraction(true);
}
```

## protected boolean doInput

URLConnection可以用于读取服务器、写入服务器，或者同时用于读/写服务器。如果URLConnection可以用来读取，保护类型boolean字段doInput就为true，否则为false。默认值是true。要访问这个保护类型变量，可以使用getDoInput()和setDoInput()公共方法：

```
public void    setDoInput(boolean doInput)
public boolean getDoInput()
```

例如：

```
try {
  URL u = new URL("http://www.oreilly.com");
  URLConnection uc = u.openConnection();
  if (!uc.getDoInput()) {
    uc.setDoInput(true);
  }
  // 从连接读取...
} catch (IOException ex) {
```

```
    System.err.println(ex);
  }
```

## protected boolean doOutput

程序可以使用URLConnection将输出发回服务器。例如，如果程序需要使用POST方法向服务器发送数据，可以通过从URLConnection获取输出流来完成。如果URLConnection可以用于写入，保护类型boolean字段doOutput就为true，否则为false，默认值为false。要访问这个保护类型变量，可以使用getDoOutput()和setDoOutput()方法：

```
public void setDoOutput(boolean dooutput)
public boolean getDoOutput()
```

例如：

```
try {
  URL u = new URL("http://www.oreilly.com");
  URLConnection uc = u.openConnection();
  if (!uc.getDoOutput()) {
    uc.setDoOutput(true);
  }
  // 写入连接...
} catch (IOException ex) {
  System.err.println(ex);
}
```

为一个*http* URL将doOutput设置为true时，请求方法就由GET改为POST。本章后面“向服务器写入数据”一节中将更详细地解释这一点。

## protected boolean ifModifiedSince

许多客户端（尤其是Web客户端和代理）会保留以前获取的文档的缓存。如果用户再次要求相同的文档，可以从缓存中获取。不过，在最后一次获取这个文档之后，服务器上的文档可能改变。要判断是否有变化，唯一的办法就是询问服务器。客户端可以在客户端请求的HTTP首部中包括一个If-Modified-Since。这个首部包括一个日期和时间。如果文档在这个时间后有所修改，服务器就发送该文档，否则就不发送。一般情况下，这个时间是客户端最后获得文档的时间。例如，下面的客户端请求表明，只有当文档在格林尼治标准时间2014年10月31日上午7:22:07之后修改过，才返回这个文档：

```
GET / HTTP/1.1
Host: login.ibiblio.org:56452
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
If-Modified-Since: Fri, 31 Oct 2014 19:22:07 GMT
```

如果文档在这个时间之后修改过，服务器就会像平常一样发送这个文档。否则，它回复一个304 Not Modified （“未修改”）消息，如下：

```
HTTP/1.0 304 Not Modified
Server: WN/1.15.1
Date: Sun, 02 Nov 2014 16:26:16 GMT
Last-modified: Fri, 29 Oct 2004 23:40:06 GMT
```

然后客户端从缓存中加载这个文档。并非所有Web服务器会考虑If-Modified-Since字段。有些服务器无论文档是否修改过都会发送。

URLConnection类的ifModifiedSince字段指示了将放置在If-Modified-Since首部字段中的日期（自格林尼治标准时间1970年1月1日子夜后的毫秒数）。因为ifModifiedSince是protected的（保护类型），程序应当调用getIfModifiedSince()和setIfModifiedSince()来读取和修改：

```
public long getIfModifiedSince()
public void setIfModifiedSince(long ifModifiedSince)
```

示例7-13将打印ifModifiedSince的默认值，将这个值设置为24小时之前，并打印这个新值。然后下载并显示文档，但只是在最后24小时内有所修改时才会显示文档。

示例7-13：将ifModifiedSince设置为24小时之前

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

public class Last24 {

  public static void main (String[] args) {

    // 用当前日期和时间初始化一个Date对象
    Date today = new Date();
    long millisecondsPerDay = 24 * 60 * 60 * 1000;

    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        URLConnection uc = u.openConnection();
        System.out.println("Original if modified since: "
            + new Date(uc.getIfModifiedSince()));
        uc.setIfModifiedSince((new Date(today.getTime()
            - millisecondsPerDay)).getTime());
        System.out.println("Will retrieve file if it's modified since"
            + new Date(uc.getIfModifiedSince()));
        try (InputStream in = new BufferedInputStream(uc.getInputStream())) {
          Reader r = new InputStreamReader(in);
          int c;
          while ((c = r.read()) != -1) {
            System.out.print((char) c);
```

```
            }
            System.out.println();
          }
        } catch (IOException ex) {
          System.err.println(ex);
        }
      }
    }
}
```

结果如下。首先会看到默认值：GMT 1970年1月1日子夜，转换为太平洋标准时间。接下来会看到新的时间，设置为当前时间之前的24小时：

```
% java Last24 http://www.elharo.com
Original if modified since: Wed Dec 31 19:00:00 EST 1969
Will retrieve file if it's modified since Sat Jun 01 11:11:27 EDT 2013
```

由于文档在最后24小时内没有修改过，所以没有重新显示文档。

## protected boolean useCaches

有些客户端（特别是Web浏览器）可以从本地缓存获取文档，而不是从服务器获取。applet可以访问浏览器的缓存。独立的应用程序可以使用java.net.ResponseCache类。如果有缓存，useCaches变量确定了是否可以使用缓存。默认值为true，表示将使用缓存；false表示不使用缓存。因为useCaches是protected变量，所以程序要使用getUseCaches()和setUseCaches()方法来访问这个变量：

```
public void setUseCaches(boolean useCaches)
public boolean getUseCaches()
```

下面的代码段将useCaches设置为false来禁用缓存，确保总会获取文档的最新版本：

```
try {
  URL u = new URL("http://www.sourcebot.com/");
  URLConnection uc = u.openConnection();
  uc.setUseCaches(false);
  // 读取文档...
} catch (IOException ex) {
  System.err.println(ex);
}
```

getDefaultUseCaches()和setDefaultUseCaches()这两个方法定义了useCaches字段的初始值：

```
public void    setDefaultUseCaches(boolean useCaches)
public boolean getDefaultUseCaches()
```

虽然不是静态方法，但这些方法确实能设置和获得一个静态字段，它能确定修改这个字段后创建的所有URLConnection类实例的默认行为。下面的代码段将默认禁用缓存，运行这个代码后，希望缓存的URLConnection必须使用setUseCaches(true)显式启用缓存：

```
if (uc.getDefaultUseCaches()) {
  uc.setDefaultUseCaches(false);
}
```

## 超时

有4个方法可以查询和修改连接的超时值。也就是说，底层socket等待远程服务器的响应时，等待多长时间后会抛出SocketTimeoutException异常。这些方法包括：

```
public void setConnectTimeout(int timeout)
public int getConnectTimeout()
public void setReadTimeout(int timeout)
public int getReadTimeout()
```

setConnectTimeout()/getConnectTimeout()控制socket等待建立连接的时间。setReadTimeout()/getReadTimeout()控制输入流等待数据到达的时间。所有这4个方法都以毫秒为单位度量时间。这4个方法都将0解释为永远不超时。如果超时值为负数，两个设置方法都会抛出IllegalArgumentException异常。例如，下面的代码会请求30秒连接超时和45秒读取超时：

```
URL u = new URL("http://www.example.org");
URLConnuction uc = u.openConnection();
uc.setConnectTimeout(30000);
uc.setReadTimeout(45000);
```

# 配置客户端请求HTTP首部

HTTP客户端（如浏览器）向服务器发送一个请求行和一个首部。例如，下面是Chrome发送的一个HTTP首部：

```
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Charset:ISO-8859-1,utf-8;q=0.7,*;q=0.3
Accept-Encoding:gzip,deflate,sdch
Accept-Language:en-US,en;q=0.8
Cache-Control:max-age=0
Connection:keep-alive
Cookie:reddit_first=%7B%22firsttime%22%3A%20%22first%22%7D
DNT:1
Host:lesswrong.com
User-Agent:Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/537.31
    (KHTML, like Gecko) Chrome/26.0.1410.65 Safari/537.31
```

Web服务器可以根据这个信息向不同的客户端提供不同的页面，获取和设置cookie，通过口令认证用户等。通过在客户端发送和服务器响应的首部中放置不同的字段，就可以完成这些工作。

---

**提示：** 重要的是，要了解这不是服务器发送给客户端的HTTP首部，即不是前面讨论的各种getHeaderField()和getHeaderFieldKey()方法所读取的首部。这是客户端发送给服务器的HTTP首部。

---

每个URLConnection会在首部默认设置一些不同的名-值对。例如，下面是示例7-1中SourceViewer2程序的一个连接发送的HTTP首部：

```
User-Agent: Java/1.7.0_17
Host: httpbin.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

可以看到，这比Chrome发送的首部要简单一些，它有一个不同的用户代理，而且接受不同类型的文件。不过，在连接前可以修改这些字段并增加新的字段。

打开连接前，可以使用setRequestProperty()方法为HTTP首部增加首部字段：

```
public void setRequestProperty(String name, String value)
```

setRequestProperty()方法用指定的名和值为这个URLConnection的首部增加一个字段。这个方法只能在连接打开之前使用。如果连接已经打开，它将抛出一个IllegalStateException异常。getRequestProperty()方法会返回这个URLConnection所用HTTP首部中指定字段的值。

HTTP允许一个指定名字的属性有多个值。在这种情况下，各个值用逗号隔开。例如，上面的代码段中，Java 7发送的Accept首部有4个值：text/html、image/gif、image/jpeg和*。

---

**提示：** 只有当所连接的URL是一个*http* URL时，这些方法才真正有意义，因为只有HTTP才使用这样的首部。尽管在其他协议中（如NNTP）它们可能有其他含义，但这的确是一个不太好的API设计。这些方法应当放在更特定的HttpURLConnection类中，而不是一般的URLConnection类。

---

例如，Web服务器和客户端利用cookie存储一些受限的持久信息。cookie就是一个名-值对集合。服务器使用响应HTTP首部向客户端发送一个cookie。此后，只要客户端请求该服务器的URL，都会在HTTP请求首部中包含一个Cookie字段，如下：

```
Cookie: username=elharo; password=ACDOX9F23JJJn6G; session=100678945
```

这个特定的Cookie字段向服务器发送了3个名-值对。一个cookie中可以包括多少个名-值对并没有限制。给定一个URLConnection对象uc，可以将这个cookie增加到连接，如下所示：

```
uc.setRequestProperty("Cookie",
    "username=elharo; password=ACDOX9F23JJJn6G; session=100678945");
```

可以将同一个属性设置为一个新值，不过这会改变现有的属性值。要增加另外一个属性值，需要使用addRequestProperty()方法：

```
public void addRequestProperty(String name, String value)
```

并没有一个固定的合法首部列表。服务器一般会忽略无法识别的首部。HTTP确实对首部字段的名和值的内容有一些限制。例如，名不能包含空白符，值不能包含任何换行符。Java对包含换行符的字段有所限制，但仅此而已。如果一个字段包含换行符，setRequestProperty()和addRequestProperty()会抛出一个IllegalArgumentException异常。否则，URLConnection很容易向服务器发送不合法的首部，所以要当心。有些服务器会妥善地处理不合法的首部。有些则忽略这些不正确的首部，总会返回所请求的文档，但还有一些服务器会回复一个HTTP 400，Bad Request错误（错误请求）。

如果出于某种原因需要查看URLConnection中的首部，有一个标准的获取方法：

```
public String getRequestProperty(String name)
```

Java还提供了一个方法，可以获得连接的所有请求属性并作为一个Map返回：

```
public Map<String,List<String>> getRequestProperties()
```

键是首部字段名，值是属性值列表。名和值都存储为字符串。

# 向服务器写入数据

有时你需要向URLConnection写入数据，例如，使用POST向Web服务器提交表单，或者使用PUT上传文件。getOutputStream()方法返回一个OutputStream，可以用来写入数据传送给服务器：

```
public OutputStream getOutputStream()
```

由于URLConnection在默认情况下不允许输出，所以在请求输出流之前必须调用setDoOutput(true)。为一个*http* URL将doOutput设置为true时，请求方法将由GET变为POST。在第5章中，你已经看到了如何用GET向服务器端程序发送数据。不过，GET仅限于安全的操作，如搜索请求或页面导航，不能用于创建或修改资源的不安全操作，如在

页面上发布一条评论或订购一个比萨。安全操作可以设置书签、缓存、搜索、预取等，而不安全的操作不能这样做。

一旦得到了OutputStream，将它串链到BufferedOutputStream或BufferedWriter进行缓冲。还可以串链到一个DataOutputStream、OutputStreamWriter或者其他比原始OutputStream使用更方便的类。例如：

```
try {
  URL u = new URL("http://www.somehost.com/cgi-bin/acgi");
  // 打开连接，准备POST
  URLConnection uc = u.openConnection();
  uc.setDoOutput(true);

  OutputStream raw = uc.getOutputStream();
  OutputStream buffered = new BufferedOutputStream(raw);
  OutputStreamWriter out = new OutputStreamWriter(buffered, "8859_1");
  out.write("first=Julie&middle=&last=Harting&work=String+Quartet\r\n");
  out.flush();
  out.close();
} catch (IOException ex) {
  System.err.println(ex);
}
```

利用POST发送数据几乎与GET一样容易。调用setDoOutput(true)，然后使用URLConnection的getOutputStream()方法写入查询字符串，而不是附加到URL。Java会缓冲写入输出流的所有数据，直到流关闭为止。这使它能够计算Content-length首部的值。下面给出一个包括客户端请求和服务器响应的完整事务：

```
% telnet www.cafeaulait.org 80
Trying 152.19.134.41...
Connected to www.cafeaulait.org.
Escape character is '^]'.
POST /books/jnp3/postquery.phtml HTTP/1.0
Accept: text/plain
Content-type: application/x-www-form-urlencoded
Content-length: 63
Connection: close
Host: www.cafeaulait.org

username=Elliotte+Rusty+Harold&email=elharo%40ibiblio%2eorg
HTTP/1.1 200 OK
Date: Sat, 04 May 2013 13:27:24 GMT
Server: Apache
Content-Style-Type: text/css
Content-Length: 864
Connection: close
Content-Type: text/html; charset=utf-8

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

```
    <title>Query Results</title>
  </head>
  <body>

  <h1>Query Results</h1>

  <p>You submitted the following name/value pairs:</p>

  <ul>
  <li>username = Elliotte Rusty Harold</li>
  <li>email = elharo@ibiblio.org</li>
  </ul>

  <hr />
  Last Modified July 25, 2012

  </body>
  </html>
  Connection closed by foreign host.
```

就此而言，只要你能控制客户端和服务器，就可以使用你喜欢的任何其他数据编码方式。例如，SOAP和XML-RPC都可以向Web服务器POST XML数据，而不是发送x-www-form-url-encoded编码的查询字符串。

示例7-14给出一个名为FormPoster的程序，它使用URLConnection类和第5章的QueryString类来提交（post）表单数据。构造函数设置了URL。查询字符串是使用add()方法构建的。post()方法具体向服务器发送数据，它打开与指定URL的URLConnection，设置其doOutput字段为true，并将查询字符串写入到输出流。然后返回包含服务器响应的输入流。

main()方法是这个程序的一个简单测试，它向位于*http://www.cafeaulait.org/books/jnp4/postquery.phtml*的资源发送名字“Elliotte Rusty Harold”和电子邮件地址elharo@biblio.org。这个资源是一个简单的表单测试器，它接受任何使用POST或GET方法的输入，并返回一个HTML页面，显示所提交的名和值。返回的数据是HTML。这个示例只显示HTML，并没有尝试进行解析。很容易地扩展这个程序，可以增加一个用户界面，允许输入要提交的用户名和电子邮件地址。不过如果这样做，程序的规模会是现在的3倍，而且并没有更多关于网络编程的内容，所以这个工作留给读者作为练习。一旦理解了这个示例，编写与其他服务器端脚本通信的Java程序就很容易了。

示例7-14：提交一个表单

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

public class FormPoster {

  private URL url;
```

```
// 取自第5章，示例5-8
private QueryString query = new QueryString();

public FormPoster (URL url) {
  if (!url.getProtocol().toLowerCase().startsWith( "http" )) {
    throw new IllegalArgumentException(
        "Posting only works for http URLs");
  }
  this.url = url;
}

public void add(String name, String value) {
  query.add(name, value);
}

public URL getURL() {
  return this.url;
}

public InputStream post() throws IOException {

  // 打开连接，准备POST
  URLConnection uc = url.openConnection();
  uc.setDoOutput(true);
  try (OutputStreamWriter out
      = new OutputStreamWriter(uc.getOutputStream(), "UTF-8")) {

    // POST行、Content-type首部和Content-length首部
    // 由URLConnection发送。
    // 我们只需要发送数据
    out.write(query.toString());
    out.write("\r\n");
    out.flush();
  }

  // 返回响应
  return uc.getInputStream();
}

public static void main(String[] args) {
  URL url;
  if (args.length > 0) {
    try {
      url = new URL(args[0]);
    } catch (MalformedURLException ex) {
      System.err.println("Usage: java FormPoster url");
      return;
    }
  } else {
    try {
      url = new URL(
          "http://www.cafeaulait.org/books/jnp4/postquery.phtml");
    } catch (MalformedURLException ex) { // 不会发生
      System.err.println(ex);
      return;
```

```
    }
    }

    FormPoster poster = new FormPoster(url);
    poster.add("name", "Elliotte Rusty Harold");
    poster.add("email", "elharo@ibiblio.org");

    try (InputStream in = poster.post()) {
      // 读取响应
      Reader r = new InputStreamReader(in);
      int c;
      while((c = r.read()) != -1) {
        System.out.print((char) c);
      }
      System.out.println();
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

下面是服务器的响应：

```
% java -classpath .:jnp4e.jar FormPoster
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
        <title>Query Results</title>
</head>
<body>

<h1>Query Results</h1>

<p>You submitted the following name/value pairs:</p>

<ul>
<li>name = Elliotte Rusty Harold</li>
<li>email = elharo@ibiblio.org
</li>
</ul>

<hr />
Last Modified May 10, 2013

</body>
</html>
```

main()方法尝试从args[0]读取第一个命令行参数。这个参数是可选的；如果有参数，就认为是可以POST的URL。如果没有参数，main()就用一个默认URL（*http://www.cafeaulait.org/books/jnp4/postquery.phtml*）初始化url。然后main()构造一个FormPoster对象。为这个FormPoster对象增加两个名-值对。接下来，调用post()方法，然后读取它的响应，并在System.out上打印。

post()方法是这个类的核心。它首先打开一个连接，指向存储在url字段中的URL。由于这个URLConnection需要发送输出，所以post()方法将这个连接的doOutput字段设置为true，然后将这个URL的OutputStream串链到一个发送数据的ASCII OutputStreamWriter；接下来刷新输出并关闭流。不要忘记关闭流！如果流没有关闭，就不会发送任何数据。最后，返回URLConnection的InputStream。

总结一下，提交表单数据需要以下步骤：

1. 确定要发送给服务器端程序的名-值对。
2. 编写接受和处理请求的服务器端程序。如果没有使用定制数据编码，可以使用普通的HTML表单和Web浏览器测试这个程序。
3. 在Java程序中创建一个查询字符串。字符串应当形式如下：

   ```
   name1=value1&name2=value2&name3=value3
   ```

   在增加到查询字符串之前，先将各个名和值传递到URLEncoder.encode()。
4. 打开一个URLConnection，指向将接受数据的程序的URL。
5. 调用setDoOutput(true)设置doOutput为true。
6. 将查询字符串写入到URLConnection的OutputStream。
7. 关闭URLConnection的OutputStream。
8. 从URLConnection的InputStream读取服务器响应。

GET只能用于可以设置书签和可以链接的安全操作。POST应当用于不能设置书签或链接的非安全操作。

getOutputStream()方法还可以用于PUT请求方法，这是一种在Web服务器上存储文件的方法。要存储的数据写入到getOutputStream()返回的OutputStream。不过，这只能在URLConnection的HttpURLConnection子类中进行，所以稍后再讨论PUT。

## URLConnection的安全考虑

建立网络连接、读/写文件等存在一些常见的安全限制，URLConnection对象会受到这些安全限制的约束。例如，尽管不可信的applet能够创建URLConnection，但前提是URLConnection必须指向这个applet所来自的主机。不过，具体的细节可能有些麻烦，因为不同的URL模式及其相应的连接在安全方面可能存在不同的问题。例如，指向applet自身*jar*文件的*jar* URL应该没有问题，不过，指向本地磁盘的file URL可能就有问题了。

在尝试连接一个URL前，你可能想知道是否允许连接。为此，URLConnection类包含了一个getPermission()方法：

```
public Permission getPermission() throws IOException
```

它返回一个java.security.Permission对象，指出连接这个URL所需的权限。如果不需要任何权限（例如，没有安全管理器），它会返回null。URLConnection的子类会返回java.security.Permission的不同子类。例如，如果底层URL指向*www.gwbush.com*，getPermission()就会为主机*www.gwbush.com*返回一个java.net.SocketPermission，允许进行连接和解析。

## 猜测MIME媒体类型

如果世界大同，所有协议和所有服务器都会使用标准MIME类型正确地指定所传输的文件的类型。很遗憾，事实并不是这样。我们不仅要处理在MIME出现之前已有的老协议，如FTP，而且很多要使用MIME的HTTP服务器根本不提供MIME首部，或者会提供不正确的首部（通常是因为服务器配置错误）。URLConnection类提供了两个静态方法，可以帮助程序确定某些数据的MIME类型；如果内容类型不可用，或者你有理由相信得到的内容类型不正确，就可以使用这些方法。第一个方法是URLConnection.guessContentTypeFromName()：

```
public static String guessContentTypeFromName(String name)
```

这个方法尝试根据对象URL的文件扩展名部分猜测对象的内容类型。它将关于内容类型最好的猜测结果作为一个String返回。这种猜测很可能是正确的，人们在考虑文件名时一般都会遵循一些常规的约定。

猜测由*content-types.properties*文件来确定，这个文件通常位于*jre/lib*目录。在UNIX上，Java还可能查看*mailcap*文件来帮助做出猜测。

这个方法绝非成全之策。例如，它忽略了多个XML应用程序，如RDF（*.rdf*）、XSL（*.xsl*）等，它们的MIME类型应当是application/xml。另外它也没有为CSS样式表（*.css*）提供一个MIME类型。不过，可以把它作为一个很好的起点。

第二个MIME类型猜测方法是URLConnection.guessContentTypeFromStream()：

```
public static String guessContentTypeFromStream(InputStream in)
```

这个方法尝试查看流中前几字节来猜测内容类型。要想正常使用这个方法，InputStream必须支持标记，从而在读取了前面的字节之后可以再返回到流的开始处。

Java会查看InputStream的前16字节，不过有时可能只需要更少字节就能“鉴定”出内容类型。这些猜测通常没有guessContentTypeFromName()给出的结果那么可靠。例如，如果一个XML文档以注释开头，而不是XML声明开头，可能会被误认为是HTML文件。只能把这个方法作为最后一个手段来使用。

# HttpURLConnection

java.net.HttpURLConnection类是URLConnection的抽象子类。它提供了另外一些方法，在处理http URL时尤其有帮助。具体地，它包含的方法可以获得和设置请求方法、确定是否重定向、获得响应码和消息，以及确定是否使用了代理服务器。它还包括了几十个便于记忆的常量，对应于各种HTTP响应码。最后，它覆盖了URLConnection超类的getPermission()方法，不过并没有改变这个方法的语义。

由于这个类是抽象类，唯一的构造函数是保护类型的，所以不能直接创建HttpURLConnection的实例。不过，如果使用*http* URL构造一个URL对象并调用其openConnection()方法，返回的URLConnection就是HttpURLConnection的一个实例。可以将URLConnection强制转换为HttpURLConnection，如下所示：

```
URL u = new URL("http://lesswrong.com/");
URLConnection uc = u.openConnection();
HttpURLConnection http = (HttpURLConnection) uc;
```

或者可以跳过一个步骤，如下所示：

```
URL u = new URL("http://lesswrong.com/");
HttpURLConnection http = (HttpURLConnection) u.openConnection();
```

## 请求方法

当Web客户端联系一个Web服务器时，它发送的第一个内容是请求行。一般情况下，这一行以GET开头，后面是客户端希望获取的资源的路径，以及客户端理解的HTTP版本。例如：

```
GET /catalog/jfcnut/index.html HTTP/1.0
```

不过，Web客户端除了用GET从Web服务器获取文件外，还能完成其他操作。Web客户端可以用POST向表单提交响应，可以用PUT将文件上传到Web服务器或用DELETE删除服务器的文件。还可以用HEAD只请求文档的首部。它们可以用OPTIONS向Web服务器询问一个指定URL支持的选项列表，甚至可以用TRACE跟踪请求本身。要完成所有这些操作，只需要将GET请求方法改为不同的关键字。例如，下面显示了浏览器如何使用HEAD只请求文档的首部：

```
HEAD /catalog/jfcnut/index.html HTTP/1.1
Host: www.oreilly.com
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

默认情况下，HttpURLConnection会使用GET方法。不过，可以用setRequestMethod()来改变请求方法：

```
public void setRequestMethod(String method) throws ProtocolException
```

这个方法的参数应当是以下7个字符串之一（区分大小写）：

- GET
- POST
- HEAD
- PUT
- DELETE
- OPTIONS
- TRACE

如果使用其他方法，就会抛出java.net.ProtocolException异常，这是IOException的一个子类。不过，只设置请求方法通常是不够的。取决于你要完成的操作，可能需要调整HTTP首部，还可能要提供消息体。例如，使用POST提交表单时，要求提供一个Content-length首部。前面已经研究了GET和POST方法，下面来看其他5种选择。

---

**提示：** 有些Web服务器还支持额外的非标准请求方法。例如，WebDAV要求服务器支持PROPFIND、PROPPATCH、MKCOL、COPY、MOVE、LOCK和UNLOCK。不过，Java不支持这些方法。

---

## HEAD

HEAD也许是所有请求方法中最简单的。这个方法与GET非常相似。不过，它告诉服务器只返回HTTP首部，不用实际发送文件。这个方法最常见的用途是检查文件在最后一次缓存之后是否有修改。示例7-15是一个使用HEAD请求方法的简单程序，它会显示服务器上一个文件最后一次修改的时间。

示例7-15：获得URL的最后一个修改的时间

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

```
public class LastModified {

  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        HttpURLConnection http = (HttpURLConnection) u.openConnection();
        http.setRequestMethod("HEAD");
        System.out.println(u + "was last modified at"
          + new Date(http.getLastModified()));
      } catch (MalformedURLException ex) {
        System.err.println(args[i] + "is not a URL I understand");
      } catch (IOException ex) {
        System.err.println(ex);
      }
      System.out.println();
    }
  }
}
```

下面是一次运行得到的输出：

```
$ java LastModified http://www.ibiblio.org/xml/
http://www.ibiblio.org/xml/ was last modified at Tue Apr 06 07:45:29 EDT 2010
```

这里并不一定非要使用HEAD方法。利用GET也可以得到相同的结果。不过如果使用GET，就会通过网络发送位于*http://www.ibiblio.org/xml/*的整个文件，而我们只关心首部中的一行。如果可以使用HEAD，最好就使用HEAD，这样会高效得多。

## DELETE

DELETE方法将删除Web服务器上位于指定URL的文件。由于这个请求存在明显的安全风险，所以并非所有服务器都配置为支持这个方法，即使服务器支持这个方法，通常也要求完成某种身份认证。典型的DELETE请求如下：

```
DELETE /javafaq/2008march.html HTTP/1.1
Host: www.ibiblio.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

服务器完全可以拒绝这个请求，或者要求提供身份认证。例如：

```
HTTP/1.1 405 Method Not Allowed
Date: Sat, 04 May 2013 13:22:12 GMT
Server: Apache
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Length: 334
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
```

```
<html><head>
<title>405 Method Not Allowed</title>
</head><body>
<h1>Method Not Allowed</h1>
<p>The requested method DELETE is not allowed for the URL
   /javafaq/2008march.html.</p>
<hr>
<address>Apache Server at www.ibiblio.org Port 80</address>
</body></html>
```

即使服务器接受这个请求，其响应也与实现有关。有些服务器会删除文件，有些则只是将文件移到回收站中。另外一些则只是将这个文件标记为不可读。具体的细节由服务器厂商来定。

## PUT

许多HTML编辑器和其他希望在Web服务器上存储文件的程序都会使用PUT方法。这个方法允许客户端将文档放在网站的抽象层次结构中，而不需要知道网站如何映射到实际的本地文件系统。这与FTP正相反，FTP用户必须知道实际的目录结构，而不是服务器的虚拟目录结构。下面显示了一个编辑器如何用PUT将一个文件存放在Web服务器上：

```
PUT /blog/wp-app.php/service/pomdoros.html HTTP/1.1
Host: www.elharo.com
Authorization: Basic ZGFmZnk6c2VjZXJldA==
Content-Type: application/atom+xml;type=entry
Content-Length: 329
If-Match: "e180ee84f0671b1"

<?xml version="1.0" ?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>The Power of Pomodoros</title>
  <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
  <updated>2013-02-23T19:22:11Z</updated>
  <author><name>Elliotte Harold</name></author>
  <content>Until recently, I hadn't paid much attention to...</content>
</entry>
```

与删除文件一样，PUT通常也需要某种身份认证，而且服务器必须特别配置为支持PUT。具体细节可能根据服务器的不同而有所不同。大多数Web服务器都不直接支持PUT。

## OPTIONS

OPTIONS请求方法询问某个特定URL支持哪些选项。如果请求的URL是星号（*），那么这个请求将应用于整个服务器而不是服务器上的某个URL。例如：

```
OPTIONS /xml/ HTTP/1.1
Host: www.ibiblio.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

服务器响应OPSTIONS请求时，会发送一个HTTP首部以及这个URL上允许的命令列表。例如，发送前面的命令后，Apache会做出以下响应：

```
HTTP/1.1 200 OK
Date: Sat, 04 May 2013 13:52:53 GMT
Server: Apache
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Style-Type: text/css
Content-Length: 0
Connection: close
Content-Type: text/html; charset=utf-8
```

可以在Allow字段中找到合法命令的列表。不过，实际上这些只是服务器理解的命令，在这个URL上并不一定会实际执行这些命令。

### TRACE

TRACE请求方法会发送HTTP首部，服务器将从客户端接收这个HTTP首部。之所以需要这个信息，主要原因是要查看服务器和客户端之间的代理服务器做了哪些修改。例如，假设发送了下面的TRACE请求：

```
TRACE /xml/ HTTP/1.1
Hello: Push me
Host: www.ibiblio.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

服务器会做出以下响应：

```
HTTP/1.1 200 OK
Date: Sat, 04 May 2013 14:41:40 GMT
Server: Apache
Connection: close
Content-Type: message/http

TRACE /xml/ HTTP/1.1
Hello: Push me
Host: www.ibiblio.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

前面5行是服务器正常的响应HTTP首部。从TRACE /xml/ HTTP/1.1以后的行是原客户端请求的回显。在这里，回显与原请求是一致的。不过，如果客户端和服务器之间存在一个代理服务器，可能就不是这样了。

## 断开与服务器的连接

HTTP 1.1支持持久连接，允许通过一个TCP socket发送多个请求和响应。不过，使用

Keep-Alive时，服务器不会因为已经向客户端发送了最后一字节的数据就立即关闭连接。毕竟，客户端有可能还会发送另一个请求。服务器会超时并关闭连接，可能会有5秒处于非活动状态。不过，最好还是由客户端在确认工作结束时关闭连接。

HttpURLConnection类透明地支持HTTP Keep-Alive，除非显式将其关闭。也就是说，在服务器关闭连接之前，如果再次连接同一个服务器，它会重用socket。一旦知道与一个特定主机的会话结束，disconnect()方法允许客户端断开连接：

```
public abstract void disconnect()
```

如果这个连接上还有打开的流，disconnect()将关闭这些流。不过，反过来并不成立。关闭一个持久连接上的流时，并不会关闭这个socket并断开连接。

## 处理服务器响应

HTTP服务器响应的第一行包括一个数字码和一个消息，指示做出了何种响应。例如，最常见的响应是200 OK，表示所请求的文档已经找到。例如：

```
HTTP/1.1 200 OK
Cache-Control:max-age=3, must-revalidate
Connection:Keep-Alive
Content-Type:text/html; charset=UTF-8
Date:Sat, 04 May 2013 14:01:16 GMT
Keep-Alive:timeout=5, max=200
Server:Apache
Transfer-Encoding:chunked
Vary:Accept-Encoding,Cookie
WP-Super-Cache:Served supercache file from PHP

<HTML>
<HEAD>
rest of document follows...
```

另一个无疑也是我们非常熟悉的响应，即404 Not Found，表示所请求的URL不再指向一个文档。例如：

```
HTTP/1.1 404 Not Found
Date: Sat, 04 May 2013 14:05:43 GMT
Server: Apache
Last-Modified: Sat, 12 Jan 2013 00:19:15 GMT
ETag: "375933-2b9e-4d30c5cb0c6c0;4d02eaff53b80"
Accept-Ranges: bytes
Content-Length: 11166
Connection: close
Content-Type: text/html; charset=ISO-8859-1

<html>
<head>
```

```
<title>Lost ... and lost</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body bgcolor="#FFFFFF">
  <h1>404 FILE NOT FOUND</h1>
Rest of error message follows...
```

还有很多其他不太常见的响应。例如，响应码301表示资源已经永久移动到一个新位置，浏览器应当重定向到这个新位置，并更新所有指向老位置的书签。例如：

```
HTTP/1.1 301 Moved Permanently
Connection: Keep-Alive
Content-Length: 299
Content-Type: text/html; charset=iso-8859-1
Date: Sat, 04 May 2013 14:20:58 GMT
Keep-Alive: timeout=5, max=200
Location: http://www.cafeaulait.org/
Server: Apache
```

通常响应消息中我们只需要数字响应码。HttpURLConnection还有一个getResponseCode()方法，会以int返回这个响应码：

```
public int getResponseCode() throws IOException
```

响应码后面的文本字符串称为响应消息（response message），可以由一个getResponseMessage()方法（名字很贴切）返回：

```
public String getResponseMessage() throws IOException
```

HTTP 1.0定义了16个响应码。HTTP 1.1扩展为40个不同的响应码。虽然有些数字（比如著名的404）几乎已经成为其语义含义的代名词，但大多数我们都还不太熟悉。HttpURLConnection类包括36个命名常量，如HttpURLConnection.OK和HttpURLConnection.NOT_FOUND，表示最常见的一些响应码，在表6-1中对此做了总结。示例7-16是一个修改过的源代码查看程序，现在包括了响应消息。

示例7-16：包括响应码和消息的SourceViewer

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

public class SourceViewer3 {

  public static void main (String[] args) {
    for (int i = 0; i < args.length; i++) {
      try {
        // 打开URLConnection进行读取
        URL u = new URL(args[i]);
        HttpURLConnection uc = (HttpURLConnection) u.openConnection();
        int code = uc.getResponseCode();
        String response = uc.getResponseMessage();
```

```
        System.out.println("HTTP/1.x " + code + " " + response);
        for (int j = 1; ; j++) {
          String header = uc.getHeaderField(j);
          String key = uc.getHeaderFieldKey(j);
          if (header == null || key == null) break;
          System.out.println(uc.getHeaderFieldKey(j) + ": " + header);
        }
        System.out.println();

        try (InputStream in = new BufferedInputStream(uc.getInputStream())) {
          // 将InputStream串链到一个Reader
          Reader r = new InputStreamReader(in);
          int c;
          while ((c = r.read()) != -1) {
            System.out.print((char) c);
          }
        }
      } catch (MalformedURLException ex) {
        System.err.println(args[0] + " is not a parseable URL");
      } catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

在服务器发送的信息中，这个程序唯一没有读取的是服务器使用的HTTP版本。当前没有专门返回这个信息的方法。在这个示例中，只是假造了一个“HTTP/1.x”，如下：

```
% java SourceViewer3 http://www.oreilly.com
HTTP/1.x 200 OK
Date: Sat, 04 May 2013 11:59:52 GMT
Server: Apache
Last-Modified: Sat, 04 May 2013 11:41:06 GMT
Accept-Ranges: bytes
Content-Length: 80165
Content-Type: text/html; charset=utf-8
Cache-Control: max-age=14400
Expires: Sat, 04 May 2013 15:59:52 GMT
Vary: Accept-Encoding
Keep-Alive: timeout=3, max=100
Connection: Keep-Alive

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
...
```

不过，`uc.getHeaderField(0)`确实能返回HTTP请求的整个第一行，其中也包括版本信息：

```
HTTP/1.1 200 OK
```

## 错误条件

有些情况下，服务器可能遇到一个错误，但仍会在消息主体中返回有用的信息。例如，当客户端从*www.ibiblio.org* Web网站请求不存在的页面时，服务器并不是简单地返回404错误码，而是发送如图7-2所示的一个搜索页面，帮助用户确定缺少的页面可能在哪里。

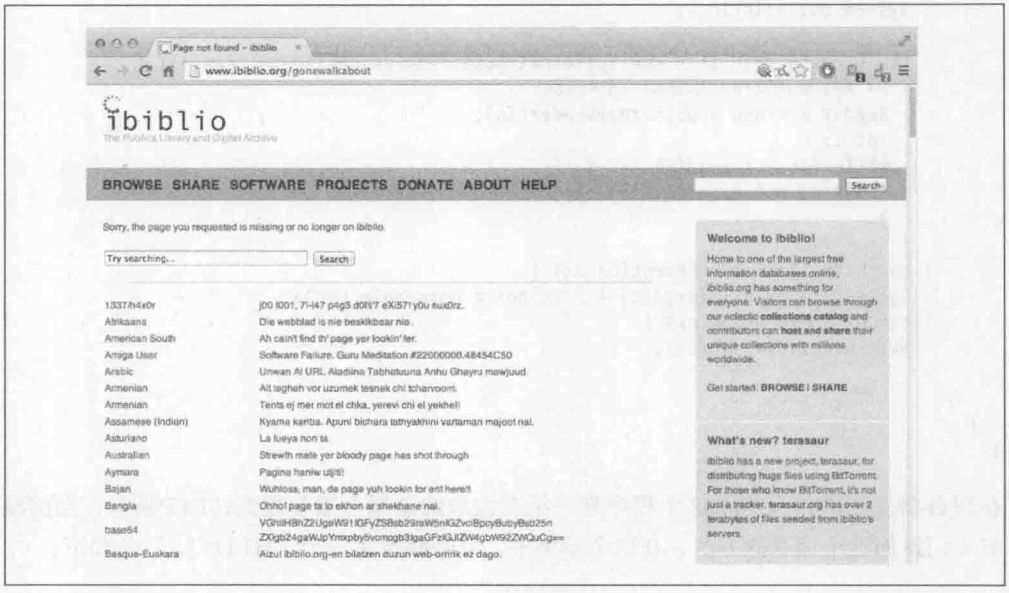


图7-2：IBiblio的404页面

getErrorStream()方法返回一个InputStream，其中包含这个页面。或者如果没有遇到错误或没有返回数据，则返回null：

```
public InputStream getErrorStream()
```

一般地，getInputStream()失败后，你会在catch块中调用getErrorStream()。示例7-17展示了一个从输入流读取数据的程序。不过，如果出于某种原因读取失败，它就会读取错误流。

示例7-17：用URLConnection下载Web页面

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

public class SourceViewer4 {

  public static void main (String[] args) {
    try {
      URL u = new URL(args[0]);
      HttpURLConnection uc = (HttpURLConnection) u.openConnection();
```

```
      try (InputStream raw = uc.getInputStream()) {
        printFromStream(raw);
      } catch (IOException ex) {
        printFromStream(uc.getErrorStream());
      }
    } catch (MalformedURLException ex) {
      System.err.println(args[0] + " is not a parseable URL");
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }

  private static void printFromStream(InputStream raw) throws IOException {
    try (InputStream buffer = new BufferedInputStream(raw)) {
      Reader reader = new InputStreamReader(buffer);
      int c;
      while ((c = reader.read()) != -1) {
        System.out.print((char) c);
      }
    }
  }
}
```

## 重定向

300一级的响应码都表示某种重定向，即请求的资源不再在期望的位置上，但有可能会在其他位置找到。遇到这样的响应时，大多数浏览器会自动从新位置加载文档。不过，这可能存在一个安全风险，因为这样很有可能将用户从一个可信网站转移到一个不可信的网站，而用户可能毫无察觉。

默认情况下，HttpURLConnection会跟随重定向。不过，HttpURLConnection有两个静态方法，允许你确定是否跟随重定向：

```
public static boolean getFollowRedirects()
public static void setFollowRedirects(boolean follow)
```

如果跟随重定向，getFollowRedirects() 方法就返回true，否则返回false。当参数为true时，setFollowRedirects()会让HttpURLConnection对象跟随重定向。当参数为false时，则会阻止HttpURLConnection对象跟随重定向。由于它们是静态方法，所以会改变调用该方法后构造的所有HttpURLConnection对象的行为。如果安全管理器不允许做此修改，setFollowRedirects()方法会抛出一个SecurityException异常。尤其是不允许applet改变这个值。

Java有两个方法可以针对各个实例配置重定向。它们是：

```
public boolean getInstanceFollowRedirects()
public void    setInstanceFollowRedirects(boolean followRedirects)
```

如果没有对给定的HttpURLConnection调用setInstanceFollowRedirects()，则该HttpURLConnection将遵循由类方法HttpURLConnection.setFollowRedirects()设置的默认行为。

## 代理

许多防火墙后面的用户或者使用AOL或其他大吞吐量ISP的用户会通过代理服务器访问Web。usingProxy()方法可以指出某个HttpURLConnection是否通过代理服务器：

```
public abstract boolean usingProxy()
```

如果使用了代理，这个方法返回true，否则返回false。在有些环境中，使用代理服务器可能存在安全方面的问题。

## 流模式

每个发送给HTTP服务器的请求都有一个HTTP首部。首部中有一个Content-length字段，即请求主体中的字节数。首部在主体的前面。不过，要写入首部，需要知道主体的长度，而在写首部的时候可能还不知道主体的长度。正常情况下，对于这个两难的问题，Java的解决办法是：对于从HttpURLConnection获取的OutputStream，将写入此OutputStream的所有内容缓存，直到流关闭。此时它就会知道主体中有多少字节，所以有足够的信息来写入Content-length首部。

这种模式对于响应典型Web表单的短请求很合适。不过，对于非常长的表单或一些SOAP消息，响应时负担会很大。用HTTP PUT发送中等到大型文档时会很浪费，也很慢。如果Java通过网络发送第一字节之前，不需要等待写入最后一字节，将会高效得多。Java 为这个问题提供了两种解决方案。如果你知道数据的大小，例如使用HTTP PUT上传一个已知大小的文件，可以将数据的大小告诉HttpURLConnection对象。如果预先不知道数据大小，可以使用分块传输编码方式。在分块传输编码方式中，请求主体以多个部分发送，每个部分都有自己单独的内容长度。要启用分块传输编码方式，只要在连接URL之前将分块大小传入setChunkedStreamingMode()方法：

```
public void setChunkedStreamingMode(int chunkLength)
```

Java将使用与本章示例稍有不同的一个HTTP形式。不过，对于Java程序员而言，这种区别并不重要。只要使用URLConnection类而不是原始socket，另外只要服务器支持分块传输编码方式，就能正常工作而不需要对代码做任何修改。不过，分块传输编码方式会妨碍身份认证和重定向。如果试图向重定向的URL或需要口令认证的URL发送分块文件，就会抛出HttpRetryException异常。你需要再次尝试请求新的URL或者用适当的凭据请

求原来的URL。这些都需要手工完成，而没有正常情况下HTTP处理器的全面支持。因此，除非确实需要，否则不要使用分块传输编码方式。与大多数性能建议一样，这意味着在测试证明默认的非流式模式确实是一个瓶颈之前，不应实现这个优化。

如果恰好预先知道请求数据的大小，可以将这个信息提供给HttpURLConnection对象，从而优化连接。如果这样做，Java会立即通过网络将数据以流方式发送。否则，它必须缓存你写入的所有数据来确定内容长度，而且只有在你关闭流之后才能通过网络发送数据。如果知道数据的具体大小，可以将这个数传递给setFixedLengthStreamingMode()方法：

```
public void setFixedLengthStreamingMode(int contentLength)
public void setFixedLengthStreamingMode(long contentLength) // Java 7
```

由于这个数可能实际大于int所能存储的最大整数，所以在Java 7及以后版本中可以使用一个long。

Java会在HTTP首部的Content-length字段中使用这个数。不过，如果接下来试图写入的数据多于或少于给出的这个字节数，Java会抛出一个IOException异常。当然这会在后来写入数据时发生，而不是开始调用这个方法时就抛出异常。如果传入一个负数，setFixedLengthStreamingMode()方法本身会抛出一个IllegalArgumentException异常，或者如果连接已经建立或设置为分块传输编码方式，调用这个方法会抛出一个IllegalStateException异常（不能对同一个请求同时使用分块传输编码方式和固定长度流模式）。

固定长度流模式对于服务器端是透明的。服务器既不知道也不关心Content-length如何设置，只要保证正确即可。不过，与分块传输编码方式类似，流模式确实会妨碍身份认证和重定向。如果给定的URL要求认证或重定向，就会抛出一个HttpRetryException异常，你必须手工重试。因此，除非确实需要，否则不要使用这个模式。

第8章

# 客户端Socket

在Internet上，数据按有限大小的包传输，这些包称为数据报（datagram）。每个数据报包含一个首部（header）和一个有效载荷（payload）。首部包含包发送到的地址和端口、包来自的地址和端口、检测数据是否被破坏的校验和，以及用于保证可靠传输的各种其他管理信息。有效载荷包含数据本身。不过，由于数据报长度有限，通常必须将数据分解为多个包，再在目的地重新组合。也有可能一个包或多个包在传输中丢失或遭到破坏，需要重传。或者包乱序到达，需要重新排序。所有这些（将数据分解为包、生成首部、解析入站包的首部、跟踪哪些包已经收到而哪些没有收到等）是很繁重的工作，需要大量复杂的代码。

幸运的是，你不需要自己来完成这项工作。Socket允许程序员将网络连接看作是另外一个可以读/写字节的流。Socket对程序员掩盖了网络的底层细节，如错误检测、包大小、包分解、包重传、网络地址等。

## 使用Socket

Socket是两台主机之间的一个连接。它可以完成7个基本操作：

- 连接远程机器。
- 发送数据。
- 接收数据。
- 关闭连接。
- 绑定端口。

- 监听入站数据。
- 在绑定端口上接受来自远程机器的连接。

Java的Socket类（客户端和服务器都可以使用）提供了对应前4个操作的方法。后面3个操作仅服务器需要，即等待客户端的连接。这些操作由ServerSocket类实现，这个类将在下一章讨论。Java程序通常采用以下方式使用客户端socket：

- 程序用构造函数创建一个新的Socket。
- Socket尝试连接远程主机。

一旦建立了连接，本地和远程主机就从这个socket得到输入流和输出流，使用这两个流相互发送数据。连接是全双工的（full-duplex），两台主机都可以同时发送和接收数据。数据的含义取决于协议，发送给FTP服务器的命令与发送给HTTP服务器的命令就有所不同。一般会先完成某种协商握手，然后再具体传输数据。

当数据传输结束后，一端或两端将关闭连接。有些协议，如HTTP 1.0，要求每次请求得到服务后都要关闭连接。其他协议，如FTP和HTTP 1.1，则允许在一个连接上处理多个请求。

## 用Telnet研究协议

这一章中，你将看到使用Socket与很多知名Internet服务（如time、dict等）通信的客户端程序。Socket本身非常简单。不过，与不同服务器通信的协议会使工作变得复杂。

为了对协议如何操作有所认识，可以使用Telnet连接一个服务器，输入不同的命令，并观察它的响应。默认情况下，Telnet会尝试连接端口23。要在不同端口连接服务器，需要指定你要连接的端口，如下所示：

```
$ telnet localhost 25
```

这会请求与本地机器的端口25（SMTP端口）建立连接，SMTP是服务器之间或邮件客户端与服务器之间传输电子邮件所用的协议。如果你知道与SMTP服务器交互的命令，不通过邮件程序就能发送电子邮件。可以利用这个技巧仿造电子邮件。例如，几年前，在新墨西哥州Sunspot的国家太阳天文台，夏季学期的学生们就曾经做过一个恶作剧，通过仿造电子邮件，让一位科学家在员工与学生之间的年度排球赛后举办的一个宴会变成了学生们的庆功晚会（当然，本书作者与这样的恶劣行径绝对无关）。与SMTP服务器的交互与此类似，用户的输入以粗体显示（名字已经经过改变，以保护那些容易上当的人）：

```
flare% telnet localhost 25
Trying 127.0.0.1 ...
Connected to localhost.sunspot.noao.edu.
Escape character is '^]'.
220 flare.sunspot.noao.edu Sendmail 4.1/SMI-4.1 ready at
Fri, 5 Jul 93 13:13:01 MDT
HELO sunspot.noao.edu
250 flare.sunspot.noao.edu Hello localhost [127.0.0.1], pleased to meet you
MAIL FROM: bart
250 bart... Sender ok
RCPT TO: local@sunspot.noao.edu
250 local@sunspot.noao.edu... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself

In a pitiful attempt to reingratiate myself with the students
after their inevitable defeat of the staff on the volleyball
court at 4:00 P.M., July 24, I will be throwing a victory
party for the students at my house that evening at 7:00.
Everyone is invited.

Beer and Ben-Gay will be provided so the staff may drown
their sorrows and assuage their aching muscles after their
public humiliation.

Sincerely,

Bart
.
250 Mail accepted
QUIT
221 flare.sunspot.noao.edu delivering mail
Connection closed by foreign host.
```

几个员工询问Bart，为什么他作为一个员工要为学生们举办庆功晚会。这个故事的寓意是，在亲自验证之前，永远不要信任电子邮件，特别是像这样的明显荒谬的电子邮件。

发生这件事之后的20年间，大多数SMTP服务器已经增加了一些安全性。它们往往要求提供用户名和口令，而且只接受本地网络中的客户以及其他可信邮件服务器的连接。不过，你仍然可以使用Telnet模拟一个客户端，查看客户端和服务器如何交互，从而了解你的Java程序需要做什么。虽然这个会话没有展示SMTP的所有特性，但足以让你推断出简单的邮件客户端与服务器如何对话。

## 用Socket从服务器读取

先来看一个简单的例子。你要连接国家标准与技术研究院（National Institute for Standards and Technology，NIST）的daytime服务器，请求当前时间。这个协议在RFC 867中定义。阅读这个RFC，可以看到daytime服务器在端口13监听，这个服务器会以人可读的格式发送时间，并关闭socket。可以用Telnet测试这个daytime服务器，如下：

```
$ telnet time.nist.gov 13
Trying 129.6.15.28...
Connected to time.nist.gov.
Escape character is '^]'.

56375 13-03-24 13:37:50 50 0 0 888.8 UTC(NIST) *
Connection closed by foreign host.
```

“56375 13-03-24 13:37:50 50 0 0 888.8 UTC(NIST)”行是daytime服务器发送的时间。读取这个`Socket`的`InputStream`时，就会得到这个结果。其他各行内容由Unix shell或Telnet程序生成。

RFC 867除了要求输出是人可读的以外，并没有为输出指定任何特定的格式。在这个例子中，可以看到这个连接是在格林尼治标准时间2013年3月24日下午1:37: 50建立的。具体来讲，格式（format）定义为JJJJJ YY-MM-DD HH:MM:SS TT L H msADV UTC(NIST) OTM，这里：

- *JJJJJ*是“修正儒略日”（也就是说，这是自1858年11月17日子夜以来的整天数）。
- *YY-MM-DD*是年份的后两位、月份和这个月的当日。
- *HH:MM:SS*以通用协调时间（Coordinated Universal Time）即UTC（基本上是格林尼治标准时间）度量的时间中的小时、分和秒。
- *TT*指示美国目前采用标准时间还是日光节省时间：00表示标准时间；50表示日光节省时间。其他值是转换日期的倒计时天数。
- *L*是一个一位码，指示当前月最后一天子夜是否增加或减去一个闰秒：0表示没有闰秒，1表示增加一个闰秒，2表示减去一个闰秒。
- *H*表示服务器的健康程度：0表示健康，1表示最多相差5秒，2表示相差超过5秒，3表示误差不确定，4表示处于维护模式。
- *msADV*是一个毫秒数，NIST把这个数增加到它发送的时间，对网络延迟大致做一个补偿。在前面的代码中，可以看到它向结果增加了888.8毫秒，因为它估计响应需要这么长时间才能返回。
- 字符串*UTC(NIST)* 是一个常量，*OTM*几乎是一个常量（这一般是一个星号，除非有非常奇怪的事情发生）。

这些细节都是NIST特定的，并不是daytime标准的一部分。尽管它们确实可以提供大量数据，不过如果你计划与一个网络时间服务器同步，最好还是使用RFC 5905中定义的NTP。

**提示：** 我不确定这个例子能像这里一样正常运行多久。这些服务器的工作负载很大，写这一章时我就断断续续遇到过连接问题。2013年初，NIST宣布："强烈建议在tcp端口13上使用NIST DAYTIME协议的用户升级到网络时间协议，它能提供更大的精度，而且需要的网络带宽更小。NIST时间客户端程序（nistime-32bit.exe）同时支持这两个协议。我们可能在2013年底将这个协议的tcp版本替换为一个基于udp的版本。"第11章将介绍如何在UDP上访问这个服务。

下面来看如何通过编程使用socket获取同样的数据。首先，在端口13打开与*time.nist.gov*的连接：

```
Socket socket = new Socket("time.nist.gov", 13);
```

这不只是创建这个对象。实际上它会在网络上建立连接。如果连接超时，或者由于服务器未在端口13上监听而失败，构造函数会抛出一个IOException异常，所以通常要把这个代码包装在一个try块中。在Java 7中，Socket实现了Autocloseable，所以你还可以使用try-with-resources：

```
try (Socket socket = new Socket("time.nist.gov", 13)) {
  // 从socket读取...
} catch (IOException ex) {
  System.err.println("Could not connect to time.nist.gov");
}
```

在Java 6和之前的版本中，要在一个finally块中显式地关闭socket，从而释放这个socket占有的资源：

```
Socket socket = null;
try {
  socket = new Socket(hostname, 13);
  // 从socket读取...
} catch (IOException ex) {
  System.err.println(ex);
} finally {
  if (socket != null) {
    try {
      socket.close();
    } catch (IOException ex) {
      // 忽略
    }
  }
}
```

下一步是可选的，不过强烈推荐完成这一步：使用setSoTimeout()方法为连接设置一个超时时间。超时时间按毫秒度量，所以下面这个语句设置socket在15秒无响应之后超时：

```
socket.setSoTimeout(15000);
```

如果服务器拒绝这个连接，socket要很快地抛出一个ConnectException，或者如果路由器无法确定如何将你的包发送到服务器，则要抛出一个NoRouteToHostException异常，尽管如此，这两个异常对于以下情况都没有帮助，即一个有问题的服务器接受了连接，然后停止与你对话，但没有主动关闭连接。对socket设置一个超时时间，这意味着对这个socket的每一个读/写都最多耗费一定的毫秒数。如果你连接服务器期间服务器挂起，会通过一个SocketTimeoutException异常通知你。具体要设置多长的超时时间，这取决于你的应用的需要，以及你希望的服务器的响应性。对于一个本地内部网服务器来说，15秒的响应时间太长了，但是对于一个负载很大的公共服务器（如*time.nist.gov*），这个时间则很短。

一旦打开socket并设置其超时时间，可以调用getInputStream()来返回一个InputStream，用它从socket读取字节。一般来讲，服务器可以发送任何字节。不过，在这个特定的例子中，协议指定发送的字节必须是ASCII：

```
InputStream in = socket.getInputStream();
StringBuilder time = new StringBuilder();
InputStreamReader reader = new InputStreamReader(in, "ASCII");
for (int c = reader.read(); c != -1; c = reader.read()) {
  time.append((char) c);
}
System.out.println(time);
```

这里我把字节存储在一个StringBuilder中。当然，你也可以使用适用于具体问题的任何数据结构来保存来自网络的数据。

示例8-1把所有这些都汇集在一个程序中，还允许你选择一个不同的daytime服务器。

示例8-1：Daytime协议客户端

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

public class DaytimeClient {

  public static void main(String[] args) {

    String hostname = args.length > 0 ? args[0] : "time.nist.gov";
    Socket socket = null;
    try {
      socket = new Socket(hostname, 13);
      socket.setSoTimeout(15000);
      InputStream in = socket.getInputStream();
      StringBuilder time = new StringBuilder();
      InputStreamReader reader = new InputStreamReader(in, "ASCII");
      for (int c = reader.read(); c != -1; c = reader.read()) {
        time.append((char) c);
      }
      System.out.println(time);
```

```
    } catch (IOException ex) {
      System.err.println(ex);
    } finally {
      if (socket != null) {
        try {
          socket.close();
        } catch (IOException ex) {
          // 忽略
        }
      }
    }
  }
}
```

一般地，这个输出与使用Telnet连接得到的输出是一样的：

```
$ java DaytimeClient
56375 13-03-24 15:05:42 50 0 0 843.6 UTC(NIST) *
```

对网络代码而言，这就足够了。不过，大多数类似这样的网络程序中，重点工作通常是使用协议和理解数据格式。例如，可以不只是打印服务器发送给你的文本，你可能还希望把它解析为一个java.util.Date对象。示例8-2展示了如何做到这一点。为了体现多样性，这个例子利用了Java 7的AutoCloseable和try-with-resources。

示例8-2：通过与time.nist.gov对话构造一个Date

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

public class Daytime {

  public Date getDateFromNetwork() throws IOException, ParseException {
    try (Socket socket = new Socket("time.nist.gov", 13)) {
      socket.setSoTimeout(15000);
      InputStream in = socket.getInputStream();
      StringBuilder time = new StringBuilder();
      InputStreamReader reader = new InputStreamReader(in, "ASCII");
      for (int c = reader.read(); c != -1; c = reader.read()) {
        time.append((char) c);
      }
      return parseDate(time.toString());
    }
  }

  static Date parseDate(String s) throws ParseException {
    String[] pieces = s.split(" ");
    String dateTime = pieces[1] + " " + pieces[2] + " UTC";
    DateFormat format = new SimpleDateFormat("yy-MM-dd hh:mm:ss z");
    return format.parse(dateTime);
  }
}
```

不过要注意，这个类在网络方面所做的并不比示例8-1更多。它只是增加了一些代码将字符串转换为日期。

从网络读取数据时，要记住重要的一点，并不是所有协议都使用ASCII，甚至不一定使用文本。例如，RFC 868中定义的时间协议指定，发送的时间是自格林尼治标准时间1900年1月1日子夜之后经过的秒数。不过，这不是作为一个类似2524521600或-1297728000的ASCII字符串来发送，而要作为一个32位无符号大端（big-endian）二进制数发送。

---

**提示：** RFC从来没有宣布这就是所使用的格式。它只是指定为32位，并假定你知道所有网络协议都使用大端数。这个数是一个无符号数，这一点只能通过以下方法确定，即首先计算有符号和无符号整数对应的日期，然后与规范（2036）中给定的日期比较。更糟糕的是，规范给出了一个负时间的例子，遵循协议的时间服务器实际上无法发送这个时间。时间协议是一个相当老的协议，早在20世纪早期就已经标准化，而那时IETF对于这个问题并不像如今这么谨慎。尽管如此，如果你发现你要实现一个不是特别明确的协议，可能必须对现有的实现做大量测试，来确定你需要做什么。在最坏情况下，不同的实现可能有不同的表现。

---

由于时间协议不发回文本，所以不能很容易地使用Telnet来测试这样一个服务，另外你的程序也不能用`Reader`或某个`readLine()`方法读取服务器响应。连接到时间服务器的Java程序必须读取原始字节，并适当地解释这些字节。在这个例子中，由于Java没有32位无符号整数类型，使得这个工作变得更为复杂。因此，你必须一次读取一字节，手动地使用位操作符<<和|将它们分别转换为一个`long`。如示例8-3所示。使用其他协议时，可能会遇到对Java更陌生的数据格式。例如，有些网络协议使用64位定点数。没有一个捷径能处理所有可能的情况。你必须咬紧牙关编写所需的代码，来处理服务器发送的任意格式的数据。

示例8-3：时间协议客户端

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

public class Time {

  private static final String HOSTNAME = "time.nist.gov";

  public static void main(String[] args) throws IOException, ParseException {
    Date d = Time.getDateFromNetwork();
    System.out.println("It is " + d);
  }

  public static Date getDateFromNetwork() throws IOException, ParseException {
    // 时间协议设置时间起点为1900年，
```

```
    // Java Date类起始于1970年。利用这个数字
    // 在它们之间进行转换

    long differenceBetweenEpochs = 2208988800L;

    // 如果不愿意使用这个魔法数，就取消
    // 以下代码的注释，这段代码会直接进行计算
    /*
    TimeZone gmt = TimeZone.getTimeZone("GMT");
    Calendar epoch1900 = Calendar.getInstance(gmt);
    epoch1900.set(1900, 01, 01, 00, 00, 00);
    long epoch1900ms = epoch1900.getTime().getTime();
    Calendar epoch1970 = Calendar.getInstance(gmt);
    epoch1970.set(1970, 01, 01, 00, 00, 00);
    long epoch1970ms = epoch1970.getTime().getTime();

    long differenceInMS = epoch1970ms - epoch1900ms;
    long differenceBetweenEpochs = differenceInMS/1000;
    */
    Socket socket = null;
    try {
      socket = new Socket(HOSTNAME, 37);
      socket.setSoTimeout(15000);

      InputStream raw = socket.getInputStream();

      long secondsSince1900 = 0;
      for (int i = 0; i < 4; i++) {
        secondsSince1900 = (secondsSince1900 << 8) | raw.read();
      }

      long secondsSince1970
                = secondsSince1900 - differenceBetweenEpochs;
      long msSince1970 = secondsSince1970 * 1000;
      Date time = new Date(msSince1970);

      return time;
    } finally {
      try {
        if (socket != null) socket.close();
      }
      catch (IOException ex) {}
    }
  }
}
```

以下是这个程序一次运行的输出：

```
$ java Time
It is Sun Mar 24 12:22:17 EDT 2013
```

时间协议指定为格林尼治标准时间，但Java Date类中的toString()方法（由System.out.println()隐式调用）会把它转换到本地主机的时区，在这里就是东部日光时间。

## 用Socket写入服务器

写入服务器并不比读取服务器更困难。只需要向Socket请求一个输出流以及一个输入流。使用输出流在socket上发送数据时，同时还可以使用输入流读取数据，不过大多数协议都设计为客户端只读取socket或者只写入socket，而不是二者同时进行。最常见的模式是，客户端发送一个请求，然后服务器响应。客户端可能发送另一个请求，服务器再做出响应。这个过程会继续，直到客户端或服务器完成工作，然后关闭连接。

RFC 2229中定义的dict是一个简单的双向TCP。在这个协议中，客户端向dict服务器的2628端口打开一个socket，并发送类似“DEFINE eng-lat gold”的命令。这会告诉服务器使用它的英语-拉丁语字典发送单词“gold”的定义（不同的服务器安装有不同的字典）。接收到第一个定义之后，客户端可能会请求另一个定义。完成时，它再发送命令“quit”。可以用Telnet研究dict协议，如下所示：

```
$ telnet dict.org 2628
Trying 216.18.20.172...
Connected to dict.org.
Escape character is '^]'.
220 pan.alephnull.com dictd 1.12.0/rf on Linux 3.0.0-14-server
    <auth.mime> <499772.29595.1364340382@pan.alephnull.com>
DEFINE eng-lat gold
150 1 definitions retrieved
151 "gold" eng-lat "English-Latin Freedict dictionary"
gold [gould]
    aurarius; aureus; chryseus
    aurum; chrysos

.
250 ok [d/m/c = 1/0/10; 0.000r 0.000u 0.000s]
DEFINE eng-lat computer
552 no match [d/m/c = 0/0/9; 0.000r 0.000u 0.000s]
quit
221 bye [d/m/c = 0/0/0; 42.000r 0.000u 0.000s]
```

可以看到，控制响应行以一个3位数字码开头。具体的定义是纯文本，以一个点号（.）结束，这个点单独占一行。如果字典不包含你请求的单词，它会返回552 no match（无匹配）。当然，可以阅读RFC了解有关的更多内容。

用Java实现这个协议并不难。首先，向一个dict服务器（比如_dict.org__）的端口2628打开一个Socket：

```
Socket socket = new Socket("dict.org", 2628);
```

再次说明，为了防止连接服务器时服务器挂起，可以设置一个超时时间：

```
socket.setSoTimeout(15000);
```

在dict协议中，客户端先“发言”，所以使用getOutputStream()请求输出流：

```
OutputStream out = socket.getOutputStream();
```

getOutputStream()方法返回一个原始OutputStream，可以用它从你的应用向Socket的另一端写数据。在使用之前，通常会把这个流串链到一个更方便的类，如DataOutputStream或OutputStreamWriter。出于性能方面的原因，将它缓冲也是一个很好的想法。由于dict协议是基于文本的，更确切地讲是基于UTF-8的，所以可以很方便地把它包装在一个Writer中：

```
Writer writer = new OutputStreamWriter(out, "UTF-8");
```

现在通过Socket写入命令：

```
writer.write("DEFINE eng-lat gold\r\n");
```

最后，刷新输出，从而确保命令会通过网络发送：

```
writer.flush();
```

现在服务器要响应一个定义。可以使用Socket的输入流来读取：

```
InputStream in = socket.getInputStream();
BufferedReader reader = new BufferedReader(
  new InputStreamReader(in, "UTF-8"));
for (String line = reader.readLine();
  !line.equals(".");
  line = reader.readLine()) {
    System.out.println(line);
}
```

看到单独占一行的一个点号时，可以知道定义已经结束。然后可以通过输出流发送quit命令：

```
writer.write("quit\r\n");
writer.flush();
```

示例8-4显示了一个完整的dict客户端。它连接到*dict.org*，将用户在命令行上输入的所有单词翻译为拉丁语。它会过滤所有以响应码（如150或220）开头的元数据行。不过，这个程序会特别检查以“552 no match”开头的一行，以防服务器不能识别用户输入的单词。

示例8-4：一个基于网络的英语-拉丁语翻译程序

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

public class DictClient {
```

```
  public static final String SERVER = "dict.org";
  public static final int PORT = 2628;
  public static final int TIMEOUT = 15000;

  public static void main(String[] args) {

    Socket socket = null;
    try {
      socket = new Socket(SERVER, PORT);
      socket.setSoTimeout(TIMEOUT);
      OutputStream out = socket.getOutputStream();
      Writer writer = new OutputStreamWriter(out, "UTF-8");
      writer = new BufferedWriter(writer);
      InputStream in = socket.getInputStream();
      BufferedReader reader = new BufferedReader(
            new InputStreamReader(in, "UTF-8"));
      for (String word : args) {
        define(word, writer, reader);
      }

      writer.write("quit\r\n");
      writer.flush();
    } catch (IOException ex) {
      System.err.println(ex);
    } finally { // 释放
      if (socket != null) {
        try {
          socket.close();
        } catch (IOException ex) {
          // 忽略
        }
      }
    }
  }

  static void define(String word, Writer writer, BufferedReader reader)
      throws IOException, UnsupportedEncodingException {
    writer.write("DEFINE eng-lat " + word + "\r\n");
    writer.flush();

    for (String line = reader.readLine(); line != null; line = reader.readLine()) {
      if (line.startsWith("250 ")) { // OK
        return;
      } else if (line.startsWith("552 ")) { // 无匹配
        System.out.println("No definition found for " + word);
        return;
      }
      else if (line.matches("\\d\\d\\d .*")) continue;
      else if (line.trim().equals(".")) continue;
      else System.out.println(line);
    }
  }
}
```

下面是一次运行的输出：

```
$ java DictClient gold uranium silver copper lead
gold [gould]
    aurarius; aureus; chryseus
    aurum; chrysos

No definition found for uranium
silver [silvər]
    argenteus
    argentum

copper [kɔpər]
    æneus; aheneus; ærarius; chalceus
    æs

lead [led]
    ducere
    molybdus; plumbum
```

示例8-4是面向行的。它从控制台读取一行输入，把它发送给服务器，等待读取它得到的一行输出。

## 半关闭Socket

close()方法同时关闭Socket的输入和输出。有时你可能希望只关闭连接的一半，即输入或者输出。shutdownInput()和shutdownOutput()方法可以只关闭连接的一半：

```
public void shutdownInput() throws IOException
public void shutdownOutput() throws IOException
```

这并不关闭Socket。实际上，它会调整与Socket连接的流，使它认为已经到了流的末尾。关闭输入之后再读取输入流会返回－1。关闭输出之后再写入Socket则会抛出一个IOException异常。

许多协议（如finger、whois和HTTP）先从客户端向服务器发送请求开始，然后读取响应。有可能在客户端发送请求之后关闭输出。例如，下面的代码段向HTTP服务器发送一个请求，然后关闭输出，因为它不再需要通过这个Socket写任何其他内容：

```
try (Socket connection = new Socket("www.oreilly.com", 80)) {
  Writer out = new OutputStreamWriter(
           connection.getOutputStream(), "8859_1");
  out.write("GET / HTTP 1.0\r\n\r\n");
  out.flush();
  connection.shutdownOutput();
  // 读取响应...
} catch (IOException ex) {
  ex.printStackTrace();
}
```

注意，即使半关闭了连接，或将连接的两半都关闭，使用结束后仍需要关闭该Socket。

shutdown方法只影响Socket的流。它们并不释放与Socket关联的资源，如所占用的端口等。

isInputShutdown()和isOutputShutdown()方法分别指出输入流和输出流是打开的还是关闭的。可以使用这些方法（不是isConnected()和isClosed()）更明确地确定可以读取还是写入Socket：

```
public boolean isInputShutdown()
public boolean isOutputShutdown()
```

# 构造和连接Socket

java.net.Socket类是Java完成客户端TCP操作的基础类。其他建立TCP网络连接的面向客户端的类（如URL、URLConnection、Applet和JEditorPane）最终都会调用这个类的方法。这个类本身使用原生代码与主机操作系统的本地TCP栈进行通信。

## 基本构造函数

每个Socket构造函数指定要连接的主机和端口。主机可以指定为InetAddress或String。远程端口指定为1到65 535之间的int值：

```
public Socket(String host, int port) throws UnknownHostException, IOException
public Socket(InetAddress host, int port) throws IOException
```

这些构造函数会连接socket（也就是说，在构造函数返回之前，会与远程主机建立一个活动的网络连接）。如果出于某种原因未能打开连接，构造函数会抛出一个IOException或UnknownHostException异常。例如：

```
try {
  Socket toOReilly = new Socket("www.oreilly.com", 80);
  // 发送和接收数据...
} catch (UnknownHostException ex) {
  System.err.println(ex);
} catch (IOException ex) {
  System.err.println(ex);
}
```

在这个构造函数中，host参数只是用String表示的一个主机名。如果域名服务器无法解析这个主机名或域名服务器没有运行，这个构造函数会抛出一个UnknownHostException异常。如果出于其他原因未能打开Socket，这个构造函数将抛出一个IOException异常。连接失败有很多原因：要到达的主机在这个端口上可能不接受连接；无法使用酒店的WiFi服务，除非你登录酒店网站并交费$14.95；或者路由问题可能阻止包到达其目的地。

由于这个构造函数不只是创建Socket对象，还会尝试连接远程主机的socket，所以可以用这个对象确定是否允许与某个端口建立连接，如示例8-5所示。

示例8-5：查看指定主机上前1024个端口中哪些安装有TCP服务器

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

public class LowPortScanner {

  public static void main(String[] args) {

    String host = args.length > 0 ? args[0] : "localhost";

    for (int i = 1; i < 1024; i++) {
      try {
        Socket s = new Socket(host, i);
        System.out.println("There is a server on port " + i + " of "
         + host);
        s.close();
      } catch (UnknownHostException ex) {
        System.err.println(ex);
        break;
      } catch (IOException ex) {
        // 这个端口上不是一个服务器
      }
    }
  }
}
```

下面是这个程序在我的本地主机上运行后的输出（你的结果可能会有所不同，这取决于哪些端口已被占用）：

```
$ java LowPortScanner
There is a server on port 21 of localhost
There is a server on port 22 of localhost
There is a server on port 23 of localhost
There is a server on port 25 of localhost
There is a server on port 37 of localhost
There is a server on port 111 of localhost
There is a server on port 139 of localhost
There is a server on port 210 of localhost
There is a server on port 515 of localhost
There is a server on port 873 of localhost
```

如果对哪些服务器在这些端口上运行感到好奇，可以用Telnet试验一下。在UNIX系统下，可以查看文件*/etc/services*发现哪些服务驻留在哪个端口。如果LowPortScanner发现了某个正在运行服务器的端口未列在*/etc/services*里，那就有趣了。

虽然这个程序看起来很简单，但并非毫无用处。保护系统的第一步就是要了解系统。这个程序有助于了解系统正在做什么，这样你能发现（并关闭）攻击者可能利用的入

口点。还可能找到无赖服务器。例如，LowPortScanner可能告诉你有一个服务器在端口800运行，进一步研究发现，这是某个人运行的一个HTTP服务器，用来提供MP3文件，这会让你的T1网络饱和。

有3个构造函数可以创建未连接的Socket。这些构造函数对于底层Socket的行为提供了更多控制，例如，可以选择一个不同的代理服务器或者一个加密机制：

```
public Socket()
public Socket(Proxy proxy)
protected Socket(SocketImpl impl)
```

## 选择从哪个本地接口连接

有两个构造函数可以指定要连接的主机和端口，以及从哪个接口和端口连接：

```
public Socket(String host, int port, InetAddress interface, int localPort)
      throws IOException, UnknownHostException
public Socket(InetAddress host, int port, InetAddress interface, int localPort)
      throws IOException
```

这个Socket连接到前两个参数中指定的主机和端口。它从后两个参数指定的本地网络接口和端口来连接。网络接口可以是物理接口（例如，一个以太网卡），也可以是虚拟接口（一个有多个IP地址的多宿主主机）。如果为localPort参数传入0，Java会随机选择1024到65 535之间的一个可用端口。

选择一个特定的网络接口来发送数据并不常见，不过偶尔也可能需要这样做。比如说，在使用双以太网端口的路由器/防火墙上就可能希望显式选择本地地址。入站连接在一个接口上接受和处理，并从另一个接口转发到本地网络。假设要编写一个程序定期地将错误日志转储到打印机，或通过一个内部邮件服务器发送出去。你希望确保使用面向内部的网络接口，而不是面向外部的网络接口。例如：

```
try {
  InetAddress inward = InetAddress.getByName("router");
  Socket socket = new Socket("mail", 25, inward, 0);
  // 使用socket...
} catch (IOException ex) {
  System.err.println(ex);
}
```

通过为本地端口号传入0，表示不关心使用了哪个端口，而只希望使用与本地主机名*router*绑定的网络接口。

与前面的构造函数一样，对于同样的原因，这个构造函数也会抛出一个IOException或UnknownHostException异常。此外，如果Socket无法绑定到所请求的本地网络接口，

它会抛出一个IOException异常（也可能是一个BindException，不过再次说明，这只是IOException的一个子类，并未在这个方法的throws子句中明确声明）。例如，无法从*b.example.org*连接*a.example.com*上运行的一个程序。你可以有意地利用这一点来限制一个已编译程序只能在预定的主机上运行。这需要为每个计算机定制一套发行版本，对于价格便宜的产品来说，这样做肯定是小题大做了。此外，Java程序很容易被反汇编、反编译和逆向工程，因此这种机制绝不是万无一失的。尽管如此，这仍可作为强制软件许可证机制的一部分。

## 构造但不连接

目前为止我们讨论的所有构造函数在创建Socket对象的同时都会打开与一个远程主机的网络连接。有时你可能想分解这些操作。如果没有为Socket构造函数提供任何参数，它就没有目标主机可以连接：

```
public Socket()
```

可以以后再为某个connect()方法传入一个SocketAddress来建立连接。例如：

```
try {
  Socket socket = new Socket();
  // 填入socket选项
  SocketAddress address = new InetSocketAddress("time.nist.gov", 13);
  socket.connect(address);
  // 使用socket...
} catch (IOException ex) {
  System.err.println(ex);
}
```

可以传入一个int作为第二个参数，来指定连接超时之前等待的时间（毫秒数）：

```
public void connect(SocketAddress endpoint, int timeout) throws IOException
```

默认值0表示永远等待下去。

之所以有这个构造函数，是为了支持不同类型的socket。还需要用它设置一个socket选项，这个选项只能在socket连接之前改变。这个内容将在这一章后面的“设置Socket选项”一节讨论。不过，我发现这个构造函数的主要好处是允许我在try-catch-finally块中清理代码，特别是对于Java 7以前的版本，这一点很有帮助。无参数构造函数不抛出任何异常，所以在finally块中关闭一个Socket时可以避免烦人的null检查。如果使用原来的构造函数，大多数代码可能如下所示：

```
Socket socket = null;
try {
  socket = new Socket(SERVER, PORT);
```

```
    // 使用socket...
  } catch (IOException ex) {
    System.err.println(ex);
  } finally {
    if (socket != null) {
      try {
        socket.close();
      } catch (IOException ex) {
        // 忽略
      }
    }
  }
```

使用无参数构造函数时，则如下所示：

```
Socket socket = new Socket();
SocketAddress address = new InetSocketAddress(SERVER, PORT);
try {
  socket.connect(address);
  // 使用socket...
} catch (IOException ex) {
  System.err.println(ex);
} finally {
  try {
    socket.close();
  } catch (IOException ex) {
    // 忽略
  }
}
```

这当然不及Java 7中的自动关闭版本，不过已经有了很大改进。

## Socket地址

SocketAddress类表示一个连接端点。这是一个空的抽象类，除了一个默认构造函数外没有其他方法。至少在理论上，SocketAddress类可以用于TCP和非TCP socket。实际上，当前只支持TCP/IP Socket。实际使用的Socket地址都是InetSocketAddress的实例。

SocketAddress类的主要用途是为暂时的socket连接信息（如IP地址和端口）提供一个方便的存储，即使最初的socket已断开并被垃圾回收，这些信息也可以重用来创建新的Socket。为此，Socket类提供了两个返回SocketAddress对象的方法（getRemoteSocketAddress()返回所连接系统的地址，getLocalSocketAdddress()返回发起连接的地址）：

```
public SocketAddress getRemoteSocketAddress()
public SocketAddress getLocalSocketAddress()
```

如果Socket尚未连接，则这两个方法都返回null。例如，首先连接到Yahoo，然后存储其地址：

```
Socket socket = new Socket("www.yahoo.com", 80);
SocketAddress yahoo = socket.getRemoteSocketAddress();
socket.close();
```

之后，可以使用这个地址重新连接Yahoo：

```
Socket socket2 = new Socket();
socket2.connect(yahoo);
```

通常会用一个主机和一个端口（对于客户端）或者只使用一个端口（对于服务器）来创建InetSocketAddress类（这是JDK中SocketAddress唯一的子类，也是我唯一见过的子类）：

```
public InetSocketAddress(InetAddress address, int port)
public InetSocketAddress(String host, int port)
public InetSocketAddress(int port)
```

还可以使用静态工厂方法InetSocketAddress.createUnresolved()，从而不再在DNS中查找主机：

```
public static InetSocketAddress createUnresolved(String host, int port)
```

InetSocketAddress提供了一些获取方法，可以用来检查这个对象：

```
public final InetAddress getAddress()
public final int         getPort()
public final String      getHostName()
```

## 代理服务器

最后一个构造函数创建一个未连接的Socket，它通过一个指定的代理服务器连接：

```
public Socket(Proxy proxy)
```

一般情况下，Socket使用的代理服务器由socksProxyHost和socksProxyPort系统属性控制，这些属性应用于系统中的所有Socket。但是这个构造函数创建的socket会使用指定的代理服务器。最值得一提的是，可以为参数传入Proxy.NO_PROXY，完全绕过所有代理服务器，而直接连接远程主机。当然，如果防火墙禁止直接连接，Java将无能为力，连接就会失败。

要使用某个特定的代理服务器，可以指定其地址。例如，下面的代码段使用位于*myproxy.example.com*的SOCKS代理服务器来连接主机*login.ibiblio.org*：

```
SocketAddress proxyAddress = new InetSocketAddress("myproxy.example.com", 1080);
Proxy proxy = new Proxy(Proxy.Type.SOCKS, proxyAddress)
Socket s = new Socket(proxy);
```

```
SocketAddress remote = new InetSocketAddress("login.ibiblio.org", 25);
s.connect(remote);
```

SOCKS是Java理解的唯一一种底层代理类型。除此以外，还有一个高层Proxy.Type.HTTP类型（作用于应用层而不是传输层），以及一个表示无代理连接的Proxy.Type.DIRECT。

## 获取Socket的信息

Socket对象有一些属性可以通过获取方法来访问：

- 远程地址。
- 远程端口。
- 本地地址。
- 本地端口。

访问这些属性的获取方法如下：

```
public InetAddress getInetAddress()
public int getPort()
public InetAddress getLocalAddress()
public int getLocalPort()
```

并没有相应的设置方法。一旦Socket连接，就会设置这些属性，而且自此固定下来。

getInetAddress()和getPort()方法指出Socket连接到的远程主机和端口。或者，如果连接现在是关闭的，这两个方法则给出Socket连接时所连接的主机和端口。getLocalAddress()和getLocalPort()方法指出socket从哪个网络接口和端口连接。

远程端口（对于客户端Socket而言）通常是由一个标准委员会预先分配的“已知端口”，而本地端口与远程端口不同，通常是由系统在运行时从未使用的空闲端口中选择。通过这种方法，系统中多个不同的客户端就可以同时访问相同的服务。本地端口与本地主机的IP地址一同嵌入在出站IP包中，所以服务器可以向客户端上正确的端口发回数据。

示例8-6从命令行读取一组主机名，尝试为每一个主机打开一个socket，然后使用这4个方法显示远程主机、远程端口、本地地址和本地端口。

示例8-6：获得Socket的信息

```
import java.net.*;
import java.io.*;
public class SocketInfo {
```

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

    for (String host : args) {
      try {
        Socket theSocket = new Socket(host, 80);
        System.out.println("Connected to " + theSocket.getInetAddress()
            + " on port " + theSocket.getPort() + " from port "
            + theSocket.getLocalPort() + " of "
            + theSocket.getLocalAddress());
      } catch (UnknownHostException ex) {
        System.err.println("I can't find " + host);
      } catch (SocketException ex) {
        System.err.println("Could not connect to " + host);
      } catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

下面是一次运行的结果。我在命令行中包括了两次*www.oreilly.com*，这是为了演示每次连接都会分配一个不同的本地端口，而不管远程主机是否相同。分配给任何连接的本地端口都是不可预知的，主要依赖于另外哪些端口已被占用。与*login.ibiblio.org*的连接失败了，因为这个机器在端口80没有运行任何服务器：

```
$ java SocketInfo www.oreilly.com www.oreilly.com www.elharo.com
  login.ibiblio.org
Connected to www.oreilly.com/208.201.239.37 on port 80 from port 49156 of
/192.168.254.25
Connected to www.oreilly.com/208.201.239.37 on port 80 from port 49157 of
/192.168.254.25
Connected to www.elharo.com/216.254.106.198 on port 80 from port 49158 of
/192.168.254.25
Could not connect to login.ibiblio.org
```

## 关闭还是连接

如果socket关闭，`isClosed()`方法会返回true，否则返回false。如果你不确定一个Socket的状态，可以用这个方法先进行检查，而不要冒抛出IOException异常的风险。例如：

```
if (socket.isClosed()) {
  // 做一些工作...
} else {
  // 做另外一些工作...
}
```

不过，这不是一个万全的测试。如果Socket从一开始从未连接，`isClosed()`也返回false，尽管Socket实际上根本没有打开过。

Socket类还有一个isConnected()方法。这个名字有一点误导。它并不指示Socket当前是否连接到一个远程主机（类似于是否未关闭）。实际上，它会指出Socket是否从未连接过一个远程主机。如果这个Socket确实能够连接远程主机，isConnected()方法就会返回true，即使这个Socket已经关闭。要查看一个Socket当前是否打开，需要检查两个条件，首先isConnected()要返回true，另外isClosed()要返回false。例如：

```
boolean connected = socket.isConnected() && ! socket.isClosed();
```

最后，isBound()会告诉你Socket是否成功地绑定到本地系统上的出站端口。isConnected()指示的是Socket的远程端，而isBound()指示本地端。这个方法目前还不太重要。第9章讨论服务器Socket时绑定会变得更重要。

## toString()

Socket类只覆盖了java.lang.Object中的一个标准方法：toString()。toString()方法会生成一个类似这样的字符串：

```
Socket[addr=www.oreilly.com/198.112.208.11,port=80,localport=50055]
```

这个方法主要用于调试。不过不要依赖这个格式：将来格式有可能会改变。 这个字符串的所有部分都可以通过其他方法直接访问（具体是getInetAddress()、getPort()和getLocalPort()）。

---

**提示：** 由于Socket是临时对象，一般来讲，Socket的持续时间与它们表示的连接是一致的，所以没有理由把它们存储在散列表中或者相互进行比较。因此，Socket没有覆盖equals()或hashCode()，这些方法的语义完全等同于Object中的相应方法。当且仅当两个Socket对象是同一个对象时它们才相等。

---

# 设置Socket选项

Socket选项指定了Java Socket类所依赖的原生socket如何发送和接收数据。对于客户端Socket，Java支持9个选项：

- TCP_NODELAY
- SO_BINDADDR
- SO_TIMEOUT
- SO_LINGER
- SO_SNDBUF

- SO_RCVBUF
- SO_KEEPALIVE
- OOBINLINE
- IP_TOS

这些选项的滑稽名字来自Berkeley UNIX所使用的C头文件中的命名常量，Socket就是Berkeley UNIX发明的。因此它们就遵循经典UNIX C命名规则，而不是更清晰的Java命名规则。例如，SO_SNDBUF实际上表示“Socket选项发送缓冲区大小”（Socket Option Send Buffer Size）。

## TCP_NODELAY

```
public void setTcpNoDelay(boolean on) throws SocketException
public boolean getTcpNoDelay() throws SocketException
```

设置TCP_NODELAY为true可确保包会尽可能快地发送，而无论包的大小。正常情况下，小数据包（一字节）在发送前会组合为更大的包。在发送另一个包之前，本地主机要等待远程系统对前一个包的确认。这称为Nagle算法。Nagle算法的问题是，如果远程系统没有足够快地将确认发送回本地系统，那么依赖于小数据量信息稳定传输的应用程序会变得很慢。对于GUI程序，比如游戏或网络计算机应用程序（服务器需要实时跟踪客户端鼠标的移动），这个问题尤其严重。在一个相当慢的网络中，即使简单地打字也会由于持续的缓冲而变得太慢。设置TCP_NODELAY为true可以打破这种缓冲模式，这样所有包一旦就绪就会发送。

setTcpNoDelay(true)关闭了Socket的缓冲。setTcpNoDelay(false)再次启用缓冲。缓冲关闭时，getTcpNoDelay()会返回true，如果缓冲打开，这个方法会返回false。例如，下面的代码在Socket缓冲未关闭时将其关闭（也就是说，打开TCP_NODELAY）：

```
if (!s.getTcpNoDelay()) s.setTcpNoDelay(true);
```

这两个方法都声明为抛出一个SocketException异常。如果底层Socket实现不支持TCP_NODELAY选项，就会抛出这个异常。

## SO_LINGER

```
public void setSoLinger(boolean on, int seconds) throws SocketException
public int getSoLinger() throws SocketException
```

SO_LINGER选项指定了Socket关闭时如何处理尚未发送的数据报。默认情况下，close()方法将立即返回，但系统仍会尝试发送剩余的数据。如果延迟时间设置为0，那

么当Socket关闭时，所有未发送的数据包都将被丢弃。如果SO_LINGER打开而且延迟时间设置为任意正数，close()方法会阻塞（阻塞时间为指定的秒数），等待发送数据和接收确认。当过去相应秒数后，Socket关闭，所有剩余的数据都不会发送，也不会收到确认。

如果底层Socket实现不支持SO_LINGER选项，这两个方法都会抛出SocketException异常。如果试图将延迟时间设置为一个负值，setSoLinger()方法还会抛出一个IllegalArgumentException异常。不过，getSoLinger()方法可以返回-1，指示这个选项被禁用，会根据需要用更多时间发送剩余的数据。例如，如果Socket对象s的延迟超时时间还未设置为其他值，可以如下将这个超时时间设置为4分钟：

```
if (s.getTcpSoLinger() == -1) s.setSoLinger(true, 240);
```

最大的延迟时间为65 535秒，有些平台上可能要更短一些。即使给定更大的时间，也会缩减为这个最大延迟时间（65 535秒）。坦白地讲，实际上65 535（超过18小时）已经很长了，你实际希望等待的时间远没有这么久。一般情况下平台的默认值会更合适。

## SO_TIMEOUT

```
public void setSoTimeout(int milliseconds) throws SocketException
public int getSoTimeout() throws SocketException
```

正常情况下，尝试从Socket读取数据时，read()调用会阻塞尽可能长的时间来得到足够的字节。设置SO_TIMEOUT可以确保这个次调用阻塞的时间不会超过某个固定的毫秒数。当时间到期时就会抛出一个InterruptedIOException异常，你应当准备好捕获这个异常。不过，Socket仍然是连接的。虽然这个read()调用失败了，但可以再次尝试读取该Socket。下一次调用可能会成功。

超时时间按毫秒数给出。0被解释为无限超时，这是默认值。例如，Socket对象s的超时值未设置时，要将其设置为3分钟，就要指定180 000毫秒：

```
if (s.getSoTimeout() == 0) s.setSoTimeout(180000);
```

当底层socket实现不支持SO_TIMEOUT选项时，这两个方法都抛出SocketException异常。如果指定的超时值为负数，setSoTimeout()方法还会抛出一个IllegalArgumentException异常。

## SO_RCVBUF和SO_SNDBUF

TCP使用缓冲区提升网络性能。较大的缓冲区会提升快速连接（比如10Mb/s以上）的性能，而较慢的拨号连接利用较小的缓冲区会有更好的表现。一般来讲，传输连续的大数

据块时（这在文件传输协议如FTP和HTTP中很常见），可以从大缓冲区受益；而对于交互式会话的小数据量传输（如Telnet和很多游戏），大缓冲区则没有多大帮助。一些相对较老的操作系统是在小文件和慢速网络时代设计的，如BSD 4.2使用2KB字节的缓冲区，Windows XP使用17 520字节的缓冲区。如今，128KB字节已经是一个常见的默认设置。

可以达到的最大带宽等于缓冲区大小除以延迟。例如，在Windows XP上，假设两个主机之间的延迟为半秒（500ms）。则带宽为17 520字节/ 0.5秒= 35040字节/秒= 273.75 kb/s。这是Socket的最大速度，而不论网络速度有多快。这对于一个拨号连接来说相当快，对于ISDN也还不错，不过对于DSL或FIOS就不够了。

可以通过减少延迟来提高速度。不过，延迟与网络硬件有关，另外还取决于你的应用控制之外的其他一些因素。另一方面，缓冲区的大小是可以控制的。例如，如果把缓冲区大小从17 520字节增加到128KB字节，最大带宽就会增加到2Mb/s。将缓冲区大小再次加倍到256KB字节，那么最大带宽就会加倍到4Mb/s。当然，网络本身对最大带宽也是有限制的。将缓冲区大小设置得过高，程序会试图以过高的速度发送和接收数据，而网络来不及处理，这就会导致拥塞、丢包和性能下降。因此，要得到最大带宽，需要让缓冲区大小与连接的延迟匹配，使它稍小于网络的带宽。

---

**提示：** 可以使用ping手动检查某个特定主机的延迟，或者可以在程序中测量`InetAddress.isReachable()`调用的时间。

---

SO_RCVBUF选项控制用于网络输入的建议的接收缓冲区大小。SO_SNDBUF选项控制用于网络输入的建议的发送缓冲区大小：

```
public void setReceiveBufferSize(int size)
    throws SocketException, IllegalArgumentException
public int getReceiveBufferSize() throws SocketException
public void setSendBufferSize(int size)
    throws SocketException, IllegalArgumentException
public int getSendBufferSize() throws SocketException
```

尽管看起来应该能独立地设置发送和接收缓冲区，但实际上缓冲区通常会设置为二者中较小的一个。例如，如果将发送缓冲区设置为64KB，而接收缓冲区设置为128KB，那么发送和接收缓冲区的大小都将是64KB。Java会报告接收缓冲区为128KB，但底层TCP栈实际上会使用64KB。

`setReceiveBufferSize()/setSendBufferSize()`方法会对socket上缓冲输出使用的字节数给出一个建议。不过，底层实现完全可以忽略或调整这个建议。具体地，UNIX和Linux系统通常指定一个最大缓冲区大小，一般是64KB或256KB，而且不允许任何

socket有更大的缓冲区。如果你试图设置一个更大的值，Java会把它调整为可能的最大缓冲区大小。在Linux上，底层实现有可能将请求的缓冲区大小加倍，这也并非没有先例。例如，如果你请求一个64KB缓冲区，可能会得到一个128KB的缓冲区。

如果参数小于或等于0，这些方法会抛出一个`IllegalArgumentException`异常。尽管这些方法还声明为抛出`SocketException`，不过实际上可能不会抛出这个异常，因为抛出`SocketException`的原因与抛出`IllegalArgumentException`的原因相同，而`IllegalArgumentException`会先做检查。

一般来讲，如果你发现你的应用不能充分利用可用带宽（例如，你有一个25Mb/s的Internet连接，但是数据传输速率仅为1.5 Mb/s），那么可以试着增加缓冲区大小。相反地，如果存在丢包和拥塞现象，则要减少缓冲区大小。不过，大多数情况下，除非网络在某个方向上负载过大，否则默认值就很合适。具体来说，现代操作系统使用TCP窗口缩放（无法从Java控制）来动态调整缓冲区大小，以适应网络。与几乎所有性能调优建议一样，一般经验是除非你检测到某个问题，否则不要进行调整。即使非要调整，可以在操作系统级增加允许的最大缓冲区大小，与调整单个socket的缓冲区大小相比，前者可以得到更大的速度提升。

## SO_KEEPALIVE

如果打开了SO_KEEPALIVE，客户端偶尔会通过一个空闲连接发送一个数据包（一般两小时一次），以确保服务器未崩溃。如果服务器没能响应这个包，客户端会持续尝试11分钟多的时间，直到接收到响应为止。如果在12分钟内未收到响应，客户端就关闭socket。如果没有SO_KEEPALIVE，不活动的客户端可能会永远存在下去，而不会注意到服务器已经崩溃。以下方法会打开和关闭SO_KEEPALIVE，并确定它的当前状态：

```
public void setKeepAlive(boolean on) throws SocketException
public boolean getKeepAlive() throws SocketException
```

SO_KEEPALIVE的默认值为false。如果打开了SO_KEEPALIVE，下面的代码会将其关闭：

```
if (s.getKeepAlive()) s.setKeepAlive(false);
```

## OOBINLINE

TCP包括一个可以发送单字节带外（Out Of Band，OOB）“紧急”数据的特性。这个数据会立即发送。此外，当接收方收到紧急数据时会得到通知，在处理其他已收到的数据之前可以选择先处理这个紧急数据。Java支持发送和接收这种紧急数据。发送方法的名字很贴切：`sendUrgentData()`。

```
public void sendUrgentData(int data) throws IOException
```

这个方法几乎会立即发送参数中的最低位字节。如果必要，当前缓存的所有数据将首先刷新输出。

接收端如何响应紧急数据有些令人迷惑，随平台和API的不同会有所差别。有些系统会分别接收紧急数据和正常数据。不过，更常见也更现代的方法是，将紧急数据以适当的顺序放在正常接收的数据队列中，告诉应用程序紧急数据已经可用，让应用程序在队列中查找紧急数据。

默认情况下，Java会忽略从Socket接收的紧急数据。不过，如果你希望接收正常数据中的紧急数据，就需要使用下面的方法设置OOBINLINE选项为true：

```
public void setOOBInline(boolean on) throws SocketException
public boolean getOOBInline() throws SocketException
```

OOBInline的默认值为false。如果OOBInline关闭，下面的代码段可以将它打开：

```
if (!s.getOOBInline()) s.setOOBInline(true);
```

一旦打开OOBInline，到达的任何紧急数据就将以正常方式放在Socket的输入流中等待读取。Java不区分紧急数据和非紧急数据。这使它不能理想地发挥作用，不过如果有一个特定字节（例如，一个Ctrl-C）对你的程序有特殊含义，而且从不出现在常规的输入流中，这就能让你更快地发送这个字节。

## SO_REUSEADDR

一个Socket关闭时，可能不会立即释放本地端口，尤其是当Socket关闭时若仍有一个打开的连接，就不会释放本地端口。有时会等待一小段时间，确保接收到所有要发送到这个端口的延迟数据包，Socket关闭时这些数据包可能仍在网络上传输。系统不会对接收的延迟包做任何处理，只是希望确保这些数据不会意外地传入绑定到同一端口的新进程。

如果使用随机端口，这不是个大问题，但是如果Socket绑定到一个已知端口，这可能就有问题了，因为这会阻止所有其他Socket同时使用这个端口。如果开启SO_REUSEADDR（默认为关闭），就允许另一个Socket绑定到这个端口，即使此时仍有可能存在前一个Socket未接收的数据。

在Java中这个选项由以下两个方法控制：

```
public void setReuseAddress(boolean on) throws SocketException
public boolean getReuseAddress() throws SocketException
```

要正常使用这些方法，setReuseAddress()必须在为这个端口绑定新Socket之前调用。这意味着Socket必须使用无参数构造函数以非连接状态创建，然后调用setReuseAddress(true)，再使用connect()方法连接Socket。之前连接的Socket和重用老地址的新Socket都必须设置SO_REUSEADDR为true，这样才能生效。

## IP_TOS服务类型

不同类型的Internet服务有不同的性能需求。例如，为了得到较好的性能，视频需要相对较高的带宽和较短的延迟，而电子邮件可以通过低带宽的连接传递，甚至延迟几个小时也不会造成大的危害。VOIP的带宽需求没有视频高，但是要保证抖动最小。不同种类的服务有不同的定价，这种做法是明智的，这样人们就不会总是自动地要求最高级的服务。毕竟，如果晚上发信的成本与通过media mail发送包裹一样，那么我们就会整晚使用Fed Ex，它会很快变得拥挤，被大量信件所淹没。Internet也是如此。

服务类型存储在IP首部中一个名为IP_TOS的8位字段中。Java允许你使用下面两个方法检查和设置Socket放在这个字段中的值：

```
public int getTrafficClass() throws SocketException
public void setTrafficClass(int trafficClass) throws SocketException
```

业务流类型以0到255之间的int给出。由于这个值要复制到TCP首部中的一个8位字段，所以只使用这个int的低字节，超出这个范围的值会导致IllegalArgumentException异常。

在21世纪的TCP栈中，这个字节的高6位包含一个差分服务代码点（Differentiated Services Code Point，DSCP）值，低两位包含一个显式拥塞通知（Explicit Congestion Notification，ECN）值。因此DSCP允许有最多$2^6$个不同的业务流。不过，要由各个网络和路由器指定这64个不同的DSCP值分别表示什么含义。表8-1所示的4个值相当常见。

表8-1：常用DSCP值和解释

| PHB（逐跳行为） | 二进制值 | 用途 |
|---|---|---|
| 默认 | 00000 | Best-effort（尽力）业务流 |
| 加速转发 | 101110 | 低损耗、低延迟、低抖动业务流。通常仅限于网络容量的30%或更低 |
| 保证转发 | 多个 | 保证最多以一个指定速率传送 |
| 类选择器 | xxx000 | 与IPv4 TOS首部向后兼容（存储在前3位中） |

例如，加速转发（Expedited Forwarding）PHB是VOIP的一个很好的选择。EF业务流通

常比所有其他业务流类有更高的优先级。下面的代码段将业务流类设置为10111000，从而设置Socket使用加速转发：

```
Socket s = new Socket("www.yahoo.com", 80);
s.setTrafficClass(0xB8); // 二进制10111000
```

要记住，这个数的低两位是显式拥塞通知，应当设置为0。

保证转发（Assured Forwarding）实际上是12个不同的DSCP值划分为4类，如表8-2所示。其作用是允许发送者在网络拥塞时表达其相对偏好，即倾向于丢哪些包。在同一类中，优先级较低的包会比优先级较高的包更早丢弃。在不同类之间，优先级较高的类中的包更为优先，不过较低优先级的类不会完全处于饥饿状态。

表8-2：保证转发优先级分类

| | 第1类（最低优先级） | 第2类 | 第3类 | 第4类（最高优先级） |
|---|---|---|---|---|
| 低丢包率 | AF11(001010) | AF21(010010) | AF31(011010) | AF41(100010) |
| 中丢包率 | AF12(001100) | AF22(010100) | AF32(011100) | AF42(100100) |
| 高丢包率 | AF13(001110) | AF23(010110) | AF33(011110) | AF43(100110) |

例如，下面的代码段建立了3个Socket，分别有不同的转发特性。如果网络足够拥挤，Socket 1（高丢包率，在第4类中）会发送其大部分数据。Socket 2（丢包率低，在第1类中）也会发送数据，不过没有Socket 1那么快。Socket 3（高丢包率，也在第1类中）会完全阻塞，直到拥塞足够缓解，使Socket 2不再丢包：

```
Socket s1 = new Socket("www.example.com", 80);
s1.setTrafficClass(0x26); // 二进制00100110
Socket s2 = new Socket("www.example.com", 80);
s2.setTrafficClass(0x0A); // 二进制00001010
Socket s3 = new Socket("www.example.com", 80);
s3.setTrafficClass(0x0E); // 二进制00001110
```

DSCP值并不是服务的严格保证。实际上，尽管一些网络会在内部参考DSCP值，但包越过ISP时，这个信息几乎总会被忽略。

**警告：** 这些选项的JavaDoc严重滞后，它还在描述一种基于位字段（对应4个业务流类）的服务质量机制：低成本、高可靠性、最大吞吐量和最小延迟。这种机制并没有广泛实现，而且可能本世纪已经不再使用。存储这些值的特定TCP首部已经有了新的用途，用来存储这里介绍的DSCP和EN值。不过，万一你还需要它（可能性很小），可以把这些值放在一个类选择器PHB的高3位中，后面的位均为0。

底层socket实现不需要考虑这些请求。它们只是为TCP栈提供所需策略的有关提示。许多实现都完全忽略这些值。具体地，Android会把setTrafficClass()方法处理为无操作。如果TCP栈不能提供请求的服务类型，它可能抛出SocketException异常，但并不要求如此。

作为表达偏好的另一种方法，setPerformancePreferences()方法为连接时间、延迟和带宽指定相对优先性。

```
public void setPerformancePreferences(int connectionTime,
    int latency, int bandwidth)
```

例如，如果connectionTime是2，latency是1，而bandwidth是3，那么最大带宽是最重要的特性，最小延迟最不重要，连接时间居中。如果connectionTime是2，latency是2，而bandwidth是3，那么最大带宽是最重要的特性，最小延迟和连接时间同等重要。至于给定的VM如何实现这一点，这依赖于具体实现。事实上，有些实现中，这可能是个“无操作”（no-op）。

## Socket异常

Socket类的大多数方法都声明抛出IOException或其子类java.net.SocketException：

```
public class SocketException extends IOException
```

不过，仅仅知道发生了问题，这对于处理问题往往是不够的。是不是因为远程主机忙而拒绝连接？还是因为没有服务在这个端口上监听而导致远程主机拒绝连接？或者是否因为网络拥塞或主机崩溃导致连接超时？SocketException有几个子类，可以对出现什么问题以及为什么会出问题提供有关的更多信息：

```
public class BindException extends SocketException
public class ConnectException extends SocketException
public class NoRouteToHostException extends SocketException
```

如果试图在一个正在使用的端口上构造Socket或ServerSocket对象，或者你没有足够的权限使用这个端口，就会抛出BindException异常。当连接被远程主机拒绝，而拒绝的原因通常是由于主机忙或没有进程在监听该端口，此时就会抛出ConnectException异常。最后一点，NoRouteToHostException异常表示连接已经超时。

java.net包还包括了ProtocolException异常，它是IOException的直接子类：

```
public class ProtocolException extends IOException
```

当从网络接收的数据违反TCP/IP规范时，会抛出这个异常。

这些异常类并没有任何特殊的方法，其他异常类也有同样的方法，不过利用这些子类，可以提供信息量更大的错误消息，或者可以确定重试此次失败的操作是否有可能成功。

# GUI应用中的Socket

HotJava Web浏览器是最早的大规模Java GUI网络客户端。HotJava已经不再更新，但还有很多用Java编写的涉及网络的客户端应用程序，这包括Eclipse IDE和Frostwire BitTorrent客户端。用Java编写商业质量的应用程序是完全可能的，特别是编写涉及网络的应用程序，这包括客户端和服务器。本节将展示一个网络客户端whois来说明这一点，另外还会讨论将网络代码与Swing应用集成时需要考虑的一些特殊问题。这个例子的功能还有些不足，不过只是用户界面弱一些，所有必要的网络代码都已经给出。事实上，我们再一次发现网络代码本身很容易，难的是用户界面。

## whois

whois是RFC 954中定义的一个简单目录服务协议，它最初是为跟踪管理员（负责管理Internet主机和域）而设计的。whois客户端连接到多个中心服务器中的一个，请求一个或多个人的目录信息，它通常会提供一个电话号码、一个电子邮件地址和一个传统邮件地址（不过不一定是当前地址）。随着Internet爆炸式的发展，whois协议已经暴露出一些缺陷，最显著的是其中心化特性。RFC 1913和1914对一个更复杂的替代协议做了说明，这称为whois++，但还没有广泛实现。

下面先从一个连接到whois服务器的简单客户端开始。whois协议的基本结构是：

1. 客户端打开指向服务器端口43的TCP socket。
2. 客户端发送一个以回车/换行对（\r\n）结束的搜索字符串。搜索字符串可以是名、名列表或一个特殊命令，稍后将讨论这个内容，还可以搜索域名，如*oreilly.com*或*netscape.com*，这会给出一个网络的有关信息。
3. 服务器在命令响应中发送不定数量的人可读信息，并关闭连接。
4. 客户端向用户显示这个信息。

客户端发送的搜索字符串格式非常简单。最基本的形式就是要搜索的人的名字。下面是一次简单的whois搜索，这里要搜索“Harold”：

```
$ telnet whois.internic.net 43
Trying 199.7.50.74...
Connected to whois.internic.net.
Escape character is '^]'.
Harold
```

```
Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

HAROLD.LUCKYLAND.ORG
HAROLD.FRUGAL.COM
HAROLD.NET
HAROLD.COM

To single out one record, look it up with "xxx", where xxx is one of the
of the records displayed above. If the records are the same, look them up
with "=xxx" to receive a full display for each record.
>>> Last update of whois database: Sat, 30 Mar 2013 15:15:05 UTC <<<
...
Connection closed by foreign host.
```

虽然前面的输入有非常清晰的格式，但可惜这个格式不是标准的。不同的whois服务器很可能发送截然不同的输出。例如，下面是在法国whois主服务器（*whois.nic.fr*）上完成同样这个搜索的前两个结果：

```
% telnet whois.nic.fr 43
telnet whois.nic.fr 43
Trying 192.134.4.18...
Connected to winter.nic.fr.
Escape character is '^]'.
Harold

Tous droits reserves par copyright.
Voir http://www.nic.fr/outils/dbcopyright.html
Rights restricted by copyright.
See http://www.nic.fr/outils/dbcopyright.html

person:      Harold Potier
address:     ARESTE
address:     154 Avenue Du Brezet
address:     63000 Clermont-Ferrand
address:     France
phone:       +33 4 73 42 67 67
fax-no:      +33 4 73 42 67 67
nic-hdl:     HP4305-FRNIC
mnt-by:      OLEANE-NOC
changed:     hostmaster@oleane.net 20000510
changed:     migration-dbm@nic.fr 20001015
source:      FRNIC

person:      Harold Israel
address:     LE PARADIS LATIN
address:     28 rue du Cardinal Lemoine
address:     Paris, France 75005 FR
phone:       +33 1 43252828
fax-no:      +33 1 43296363
```

```
e-mail:      info@cie.fr
nic-hdl:     HI68-FRNIC
notify:      info@cie.fr
changed:     registrar@ns.il 19991011
changed:     migration-dbm@nic.fr 20001015
source:      FRNIC
```

这里会返回每一个完整的记录，而不只是网站列表。其他whois服务器可能会使用其他的格式。这个协议并不是为机器处理而设计的。为了处理各个不同whois服务器的输出，你几乎都必须编写新的代码。不过，不管输出格式是什么，每个响应都可能包含一个句柄（handle），在Internic输出中这是一个域名，对于nic.fr输出这个句柄则在nic-hdl字段中。句柄可以保证是唯一的，用于获得一个人或网络的更特定的信息。如果搜索一个句柄，就至多能得到一个匹配的记录。如果搜索的结果只有一个匹配，这可能是因为你很幸运，或者因为你在搜索一个句柄，那么服务器会返回更详细的记录。下面是一次对*oreilly.com*的搜索。因为数据库中只有一个*oreilly.com*，所以服务器会返回它拥有的关于这个域的所有信息：

```
% telnet whois.internic.net 43
Trying 198.41.0.6...
Connected to whois.internic.net.
Escape character is '^]'.
oreilly.com

Whois Server Version 1.3

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

    Domain Name: OREILLY.COM
    Registrar: BULKREGISTER, LLC.
    Whois Server: whois.bulkregister.com
    Referral URL: http://www.bulkregister.com
    Name Server: NS1.SONIC.NET
    Name Server: NS.OREILLY.COM
    Status: ACTIVE
    Updated Date: 17-oct-2002
    Creation Date: 27-may-1997
    Expiration Date: 26-may-2004


>>> Last update of whois database: Tue, 16 Dec 2003 18:36:16 EST <<<
...
Connection closed by foreign host.
```

whois协议支持多个标志，可以用来限制或扩展搜索。例如，如果希望搜索一个名叫“Elliott”的人，但不确定名字的拼法是“Elliot”还是“Elliott”，或者是“Elliotte”（这种情况往往不太可能），就可以输入：

```
% whois Person Partial Elliot
```

这会告诉whois服务器，你希望只匹配名字以“Elliot”开头的人（不是域、网关、组或其他）。遗憾的是，如果希望寻找名字拼法为“Eliot”的某个人，就需要另外完成一次单独的搜索。修改搜索的规则在表8-3中做了总结。在命令行上每个前缀应当放在搜索字符串的前面。

表8-3：Whois前缀

| 前缀 | 意义 |
|---|---|
| Domain | 只寻找域记录 |
| Gateway | 只寻找网关记录 |
| Group | 只寻找组记录 |
| Host | 只寻找主机记录 |
| Network | 只寻找网络记录 |
| Organization | 只寻找组织记录 |
| Person | 只寻找人员记录 |
| ASN | 只寻找自治系统号记录 |
| Handle或! | 只搜索匹配的句柄 |
| Mailbox或@ | 只搜索匹配的电子邮件地址 |
| Name或: | 只搜索匹配的名 |
| Expand或* | 只搜索组记录并显示组中的所有个人 |
| Full或= | 显示每个匹配结果的完整记录 |
| Partial或suffix | 匹配以给定字符串开头的记录 |
| Summary或$ | 只显示概要，即使只有一条匹配记录 |
| SUBdisplay或% | 显示指定主机的用户、指定网络中的主机等 |

这些关键词都很有用，但是记忆起来却很麻烦。事实上，大多数人甚至不知道有这些关键词。他们只是在命令行输入“whois Harold”，从返回的杂乱信息中挑选。好的whois客户端不应依靠用户记住这些怪异的关键词，而是应当显示一些选项。为此需要为最终用户提供图形用户界面，并为客户端程序员提供一个更好的API。

## 网络客户库

最好把类似whois的网络协议想成是在网络上移动的位和字节，不论是作为数据包、数据报还是流。没有哪个网络协议可以完全纳入一个GUI（VNC和X11使用的远程帧缓存协

议（Remote Framebuffer Protocol）是个例外，不过还有争议）。通常最好把网络代码封装在一个单独的库中，可以由GUI代码根据需要进行调用。

示例8-7是一个可重用的whois类。其中两个字段定义了各个whois对象的状态，host是一个InetAddress对象，另一个字段port是一个int。它们共同定义了特定whois对象连接的服务器。这个类有5个构造函数，通过不同参数组合来设置这些字段。此外，可以使用setHost()方法修改主机。

这个类的主要功能在方法lookUpNames()中完成。lookUpNames()方法返回一个String，包含给定查询的whois响应。参数指定了要搜索的字符串、搜索的记录类型、搜索的数据库以及是否需要完全（精确）匹配。我们也可以用字符串或int常量来指定要搜索的记录类型和数据库，但由于只有为数不多的有效值，所以lookUpNames()定义了一个enum（包含固定数目的成员）。这种解决方案提供了更严格的编译时类型检查，保证whois类不必处理意外的值。

示例8-7：whois类

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

public class Whois {

  public final static int DEFAULT_PORT = 43;
  public final static String DEFAULT_HOST = "whois.internic.net";

  private int port = DEFAULT_PORT;
  private InetAddress host;

  public Whois(InetAddress host, int port) {
    this.host = host;
    this.port = port;
  }

  public Whois(InetAddress host) {
    this(host, DEFAULT_PORT);
  }

  public Whois(String hostname, int port)
    throws UnknownHostException {
    this(InetAddress.getByName(hostname), port);
  }

  public Whois(String hostname) throws UnknownHostException {
    this(InetAddress.getByName(hostname), DEFAULT_PORT);
  }

  public Whois() throws UnknownHostException {
  this(DEFAULT_HOST, DEFAULT_PORT);
  }
```

```
// 搜索的条目
public enum SearchFor {
  ANY("Any"), NETWORK("Network"), PERSON("Person"), HOST("Host"),
  DOMAIN("Domain"), ORGANIZATION("Organization"), GROUP("Group"),
  GATEWAY("Gateway"), ASN("ASN");
  private String label;

  private SearchFor(String label) {
    this.label = label;
  }
}

// 搜索的类别
public enum SearchIn {
  ALL(""), NAME("Name"), MAILBOX("Mailbox"), HANDLE("!");

  private String label;

  private SearchIn(String label) {
    this.label = label;
  }
}

public String lookUpNames(String target, SearchFor category,
    SearchIn group, boolean exactMatch) throws IOException {

  String suffix = "";
  if (!exactMatch) suffix = ".";

  String prefix = category.label + " " + group.label;
  String query = prefix + target + suffix;

  Socket socket = new Socket();
  try {
    SocketAddress address = new InetSocketAddress(host, port);
    socket.connect(address);
    Writer out
        = new OutputStreamWriter(socket.getOutputStream(), "ASCII");
    BufferedReader in = new BufferedReader(new
        InputStreamReader(socket.getInputStream(), "ASCII"));
    out.write(query + "\r\n");
    out.flush();

    StringBuilder response = new StringBuilder();
    String theLine = null;
    while ((theLine = in.readLine()) != null) {
      response.append(theLine);
      response.append("\r\n");
    }
    return response.toString();
  } finally {
    socket.close();
  }
}
```

```
  public InetAddress getHost() {
    return this.host;
  }

  public void setHost(String host)
      throws UnknownHostException {
    this.host = InetAddress.getByName(host);
  }
}
```

图8-1展示了图形化whois客户端的一个可能的界面，这个客户端依赖于示例8-7建立实际的网络连接。这个界面有一个输入搜索名的文本域，另外有一个复选框来确定采用精确匹配还是部分匹配。这里有一组单选按钮允许用户指定要搜索的记录组。另一组单选按钮可以用来选择要搜索的字段。默认情况下，这个客户端会以精确匹配方式搜索所有记录的所有字段。

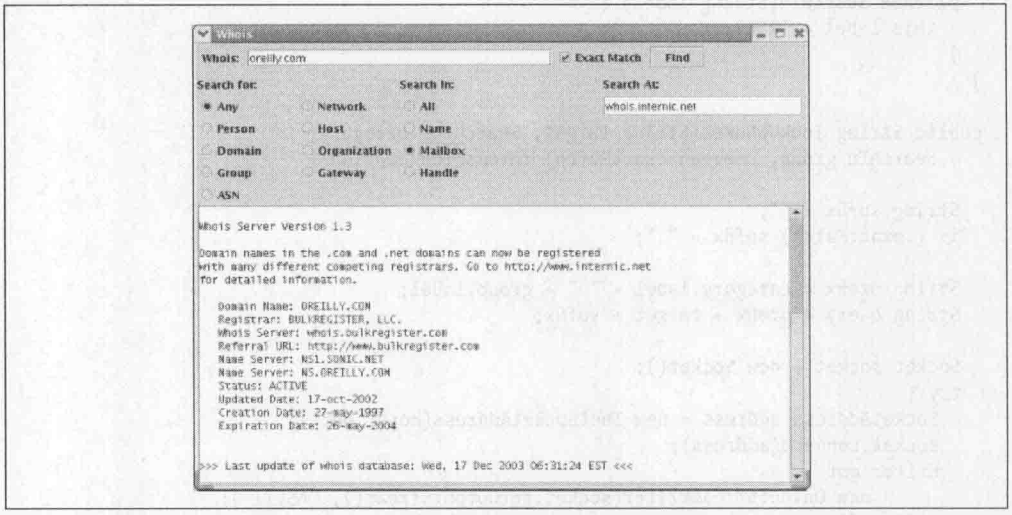


图8-1：图形化whois客户端

当用户在“Whois:”文本框中输入字符串，按Enter键或单击Find按钮时，程序将建立与whois服务器的一个连接，获取与该字符串匹配的记录。这些记录会放在窗口底部的文本域内。服务器初始设置为*whois.internic.net*，但用户可以自由改变这个设置。示例8-8给出了生成这个界面的程序。

示例8-8：图形化whois客户端界面

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.*;

  public class WhoisGUI extends JFrame {
```

```
private JTextField searchString = new JTextField(30);
private JTextArea names = new JTextArea(15, 80);
private JButton findButton = new JButton("Find");;
private ButtonGroup searchIn = new ButtonGroup();
private ButtonGroup searchFor = new ButtonGroup();
private JCheckBox exactMatch = new JCheckBox("Exact Match", true);
private JTextField chosenServer = new JTextField();
private Whois server;

public WhoisGUI(Whois whois) {
  super("Whois");
  this.server = whois;
  Container pane = this.getContentPane();

  Font f = new Font("Monospaced", Font.PLAIN, 12);
  names.setFont(f);
  names.setEditable(false);

  JPanel centerPanel = new JPanel();
  centerPanel.setLayout(new GridLayout(1, 1, 10, 10));
  JScrollPane jsp = new JScrollPane(names);
  centerPanel.add(jsp);
  pane.add("Center", centerPanel);

  // 不希望南边和北边的按钮占满整个区域，
  // 所以在那里添加Panel，
  // 并在Panel中使用FlowLayout
  JPanel northPanel = new JPanel();
  JPanel northPanelTop = new JPanel();
  northPanelTop.setLayout(new FlowLayout(FlowLayout.LEFT));
  northPanelTop.add(new JLabel("Whois: "));
  northPanelTop.add("North", searchString);
  northPanelTop.add(exactMatch);
  northPanelTop.add(findButton);
  northPanel.setLayout(new BorderLayout(2,1));
  northPanel.add("North", northPanelTop);
  JPanel northPanelBottom = new JPanel();
  northPanelBottom.setLayout(new GridLayout(1,3,5,5));
  northPanelBottom.add(initRecordType());
  northPanelBottom.add(initSearchFields());
  northPanelBottom.add(initServerChoice());
  northPanel.add("Center", northPanelBottom);

  pane.add("North", northPanel);

  ActionListener al = new LookupNames();
  findButton.addActionListener(al);
  searchString.addActionListener(al);
}

private JPanel initRecordType() {
  JPanel p = new JPanel();
  p.setLayout(new GridLayout(6, 2, 5, 2));
  p.add(new JLabel("Search for:"));
```

```
    p.add(new JLabel(""));
    JRadioButton any = new JRadioButton("Any", true);
    any.setActionCommand("Any");
    searchFor.add(any);
    p.add(any);

    p.add(this.makeRadioButton("Network"));
    p.add(this.makeRadioButton("Person"));
    p.add(this.makeRadioButton("Host"));
    p.add(this.makeRadioButton("Domain"));
    p.add(this.makeRadioButton("Organization"));
    p.add(this.makeRadioButton("Group"));
    p.add(this.makeRadioButton("Gateway"));
    p.add(this.makeRadioButton("ASN"));

    return p;
  }

  private JRadioButton makeRadioButton(String label) {
    JRadioButton button = new JRadioButton(label, false);
    button.setActionCommand(label);
    searchFor.add(button);
    return button;
  }

  private JRadioButton makeSearchInRadioButton(String label) {
    JRadioButton button = new JRadioButton(label, false);
    button.setActionCommand(label);
    searchIn.add(button);
    return button;
  }

  private JPanel initSearchFields() {
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(6, 1, 5, 2));
    p.add(new JLabel("Search In: "));

    JRadioButton all = new JRadioButton("All", true);
    all.setActionCommand("All");
    searchIn.add(all);
    p.add(all);

    p.add(this.makeSearchInRadioButton("Name"));
    p.add(this.makeSearchInRadioButton("Mailbox"));
    p.add(this.makeSearchInRadioButton("Handle"));
    return p;
  }

  private JPanel initServerChoice() {
    final JPanel p = new JPanel();
    p.setLayout(new GridLayout(6, 1, 5, 2));
    p.add(new JLabel("Search At: "));

    chosenServer.setText(server.getHost().getHostName());
    p.add(chosenServer);
```

```
    chosenServer.addActionListener( new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent event) {
        try {
          server = new Whois(chosenServer.getText());
        } catch (UnknownHostException ex) {
          JOptionPane.showMessageDialog(p,
            ex.getMessage(), "Alert", JOptionPane.ERROR_MESSAGE);
        }
      }
    } );

    return p;
  }

  private class LookupNames implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent event) {
      names.setText("");
      SwingWorker<String, Object> worker = new Lookup();
      worker.execute();
    }
  }

  private class Lookup extends SwingWorker<String, Object> {

    @Override
    protected String doInBackground() throws Exception {
      Whois.SearchIn group = Whois.SearchIn.ALL;
      Whois.SearchFor category = Whois.SearchFor.ANY;

      String searchForLabel = searchFor.getSelection().getActionCommand();
      String searchInLabel = searchIn.getSelection().getActionCommand();

      if (searchInLabel.equals("Name")) group = Whois.SearchIn.NAME;
      else if (searchInLabel.equals("Mailbox")) {
        group = Whois.SearchIn.MAILBOX;
      } else if (searchInLabel.equals("Handle")) {
        group = Whois.SearchIn.HANDLE;
      }
      if (searchForLabel.equals("Network")) {
        category = Whois.SearchFor.NETWORK;
      } else if (searchForLabel.equals("Person")) {
        category = Whois.SearchFor.PERSON;
      } else if (searchForLabel.equals("Host")) {
        category = Whois.SearchFor.HOST;
      } else if (searchForLabel.equals("Domain")) {
        category = Whois.SearchFor.DOMAIN;
      } else if (searchForLabel.equals("Organization")) {
        category = Whois.SearchFor.ORGANIZATION;
      } else if (searchForLabel.equals("Group")) {
        category = Whois.SearchFor.GROUP;
      } else if (searchForLabel.equals("Gateway")) {
        category = Whois.SearchFor.GATEWAY;
```

```
        } else if (searchForLabel.equals("ASN")) {
          category = Whois.SearchFor.ASN;
        }

        server.setHost(chosenServer.getText());
        return server.lookUpNames(searchString.getText(),
          category, group, exactMatch.isSelected());
      }

      @Override
      protected void done() {
        try {
          names.setText(get());
        } catch (Exception ex) {
          JOptionPane.showMessageDialog(WhoisGUI.this,
              ex.getMessage(), "Lookup Failed", JOptionPane.ERROR_MESSAGE);
        }
      }
    }

    public static void main(String[] args) {
      try {
        Whois server = new Whois();
        WhoisGUI a = new WhoisGUI(server);
        a.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        a.pack();
        EventQueue.invokeLater(new FrameShower(a));
      } catch (UnknownHostException ex) {
        JOptionPane.showMessageDialog(null, "Could not locate default host "
            + Whois.DEFAULT_HOST, "Error", JOptionPane.ERROR_MESSAGE);
      }
    }

    private static class FrameShower implements Runnable {

      private final Frame frame;

      FrameShower(Frame frame) {
      this.frame = frame;
    }

    @Override
    public void run() {
      frame.setVisible(true);
    }
  }
}
```

main()方法是通常启动独立应用程序的代码块。它构造一个Whois对象，然后使用这个对象构造一个WhoisGUI对象。接下来WhoisGUI构造函数建立Swing界面。这里有很多冗余代码，所以把它们取出放在私有方法initSearchFields()、initServerChoice()、makeSearchInRadioButton()和makeSearchForRadioButton()中。与常见的基于

LayoutManager的界面一样，建立界面的过程很复杂。因为你可能会使用可视化设计工具来构建这样一个应用程序，所以这里不再详细描述。

当构造函数返回时，main()方法就向窗口附加了一个匿名内部类，它将在窗口关闭时关闭应用程序。（这不在构造函数中完成，因为使用这个类的其他程序可能不希望在窗口关闭时退出程序）。然后main()调整窗口大小并显示。为避免可能导致死锁的不明显的竞态条件，这需要在事件分发线程中进行。实现了Runnable的FrameShower内部类和EventQueue.invokeLater()调用也是如此。自此，所有活动都在事件分发线程中发生。

这个程序必须响应的第一个事件是用户在“Whois:”文本框中输入一个名，并单击Find按钮或按Enter键。在这种情况下，LookupNames内部类将设置主文本为空串，并执行一个SwingWorker来建立网络连接。如果要编写访问网络的GUI应用，或者要完成任何I/O，SwingWorker（Java 6中引入）是一个需要学习的很重要的类。

SwingWorker解决的问题如下。在所有Java GUI应用中，为了避免死锁和程序过慢，有两个必须遵循的规则：

- 对Swing组件的所有更新都在事件分发线程中发生。
- 事件分发线程中不能有很慢的阻塞操作，尤其是I/O。否则响应很慢的服务器可能会挂起整个应用。

对于大量涉及网络和I/O的代码来说，这两个规则是对立的，因为完成I/O的部分代码不能更新GUI，反之亦然，更新GUI的代码不能完成I/O。这两个工作必须在两个不同的线程中完成。

要回避这个问题，有很多方法，不过在Java 6之前，这些方法都相当复杂。但在Java 6和以后版本中，解决方案就很容易了。只需要定义SwingWorker的一个子类，并覆盖两个方法：

1. doInBackground()方法完成长时间运行的有大量I/O的操作。它不与GUI交互。这个方法可以返回任何方便的类型，还可以抛出任何异常。
2. done()方法会在doInBackground()方法返回后由事件分发线程自动调用，所以它可以更新GUI。这个方法可以调用get()方法来获取doInBackground()计算的返回值。

示例8-8使用一个内部类（名为Lookup）作为它的SwingWorker。doInBackground()方法与whois服务器对话，将服务器的响应作为一个String返回。done()方法则用服务器的响应更新names文本域。

这个程序必须响应的第二个事件是用户在服务器文本域中输入一个新的主机。在这种情况下，会有一个匿名内部类尝试构造一个新的Whois对象，把它存储在server字段。如果失败（例如，由于用户输入了错误的主机名），则恢复原来的服务器。它会弹出一个提示框，通知用户发生了这个事件。

这并不是一个完美的客户端。最突出的缺陷是它没有提供一种方法来保存数据和退出程序。不过，这个客户端确实很好地展示了如何从一个GUI程序安全地建立网络连接，而不阻塞事件分发线程。

第9章

# 服务器Socket

前一章从客户端的角度讨论了Socket：客户端就是向监听连接的服务器打开一个Socket的程序。不过，只有客户端socket还不够，如果不能与服务器对话，客户端并没有什么用处，而前一章讨论的Socket类不足以编写服务器。要创建一个Socket，你需要知道希望连接哪个Internet主机。编写服务器时，无法预先了解哪个主机会联系你，即使确实知道，你也不清楚那个主机希望何时与你联系。换句话说，服务器就像坐在电话旁等电话的接线员。他们不知道谁会打电话，或者什么时间打电话，只知道当电话铃响时，就必须拿起电话与之对话，而不管对方是谁。只用Socket类是做不到这一点的。

对于接受连接的服务器，Java提供了一个ServerSocket类表示服务器Socket。基本说来，服务器Socket的任务就是坐在电话旁等电话。从技术上讲，服务器Socket在服务器上运行，监听入站TCP连接。每个服务器Socket监听服务器机器上的一个特定端口。当远程主机上的一个客户端尝试连接这个端口时，服务器就被唤醒，协商建立客户端和服务器之间的连接，并返回一个常规的Socket对象，表示两台主机之间的Socket。换句话说，服务器Socket等待连接，而客户端Socket发起连接。一旦ServerSocket建立了连接，服务器会使用一个常规的Socket对象向客户端发送数据。数据总是通过常规socket传输。

## 使用ServerSocket

ServerSocket类包含了使用Java编写服务器所需的全部内容。其中包括创建新ServerSocket对象的构造函数、在指定端口监听连接的方法、配置各个服务器Socket选项的方法，以及其他一些常见的方法，如toString()。

在Java中，服务器程序的基本生命周期如下：

1. 使用一个ServerSocket()构造函数在一个特定端口创建一个新的ServerSocket。
2. ServerSocket使用其accept()方法监听这个端口的入站连接。accept() 会一直阻塞，直到一个客户端尝试建立连接，此时accept()将返回一个连接客户端和服务器的Socket对象。
3. 根据服务器的类型，会调用Socket的getInputStream()方法或getOutputStream()方法，或者这两个方法都调用，以获得与客户端通信的输入和输出流。
4. 服务器和客户端根据已协商的协议交互，直到要关闭连接。
5. 服务器或客户端（或二者）关闭连接。
6. 服务器返回到步骤2，等待下一次连接。

下面来看一个比较简单的协议daytime。第8章介绍过，daytime服务器监听端口13。客户端连接时，服务器会以人可读的格式发送时间，并关闭连接。例如，下面是与位于*time-a.nist.gov*的一个daytime服务器的连接：

```
$ telnet time-a.nist.gov 13
Trying 129.6.15.28...
Connected to time-a.nist.gov.
Escape character is '^]'.

56375 13-03-24 13:37:50 50 0 0 888.8 UTC(NIST) *
Connection closed by foreign host.
```

实现你自己的daytime服务器很容易。首先，创建一个监听端口13的服务器Socket：

```
ServerSocket server = new ServerSocket(13);
```

接下来，接受一个连接：

```
Socket connection = server.accept();
```

accept()调用会阻塞。也就是说，程序会停在这里等待，可能等待几个小时甚至几天，直到有客户端连接端口13。客户连接时，accept()方法返回一个Socket对象。

需要说明，连接会作为一个java.net.Socket对象返回，这正是前一章中客户端使用的Socket对象。daytime协议要求服务器讲话（而且仅服务器讲话，客户端无需对答），所以从这个socket获得一个OutputStream。由于daytime协议需要文本，可以把这个OutputStream串链到一个OutputStreamWriter：

```
OutputStream out = connection.getOutputStream();
Writer writer = new OutputStreamWriter(writer, "ASCII");
```

现在得到当前时间，并把它写入这个流。除了要求是人可读的格式外，daytime协议对具体格式没有任何其他要求，所以可以让Java为你选择一种格式：

```
Date now = new Date();
out.write(now.toString() +"\r\n");
```

不过，要强调一点，需要使用一个回车/换行对来结束这一行。网络服务器中几乎都会这样做。你要显式选择这个行结束符，而不是使用系统的行分隔符（不论是用System.getProperty("line.separator")显式选择，还是通过类似println()的方法隐式使用），最后，刷新输出连接，并将它关闭：

```
out.flush();
connection.close();
```

并不一定只写一次之后就将连接关闭。很多协议（例如dict和HTTP 1.1）允许客户端通过一个socket发送多个请求，并希望服务器发送多个响应。有些协议（如FTP）甚至可以保持一个socket无限期打开。不过，daytime协议只允许一个响应。

如果客户端关闭连接时服务器仍在操作，连接服务器和客户端的输入和/或输出流会在下一次读或写时抛出一个InterruptedIOException。不论哪一种情况，服务器都应做好准备处理下一个入站连接。

当然，你可能想要反复这么做，所以可以把它放在一个循环里。每次循环时会调用一次accept()方法。这会返回一个Socket对象，表示远程客户端和本地服务器之间的连接。与客户端的交互就通过这个Socket对象完成。例如：

```
ServerSocket server = new ServerSocket(port);
while (true) {
  try (Socket connection = server.accept()) {
    Writer out = new OutputStreamWriter(connection.getOutputStream());
    Date now = new Date();
    out.write(now.toString() +"\r\n");
    out.flush();
  } catch (IOException ex) {
    // 一个客户端的问题，不会导致服务器关闭
    System.err.println(ex.getMessage());
  }
}
```

这称为一个迭代服务器（iterative server）。这里有一个大循环，每次循环时分别处理一个连接。对于类似daytime这样非常简单的协议，只有很小的请求和响应，这种服务器可以很好地工作。不过，即使是这么简单的一个协议，有可能会有一个速度很慢的客户端延迟其他更快的客户端。接下来的例子会用多线程或异步I/O解决这个问题。

增加异常处理时，代码会变得有些复杂。有两类异常，一类异常可能关闭服务器并记录

一个错误消息，另一类异常只关闭活动连接，区分这两类异常非常重要。某个特定连接范围内的异常会关闭这个连接，但是不会影响其他异常或关闭服务器。单个请求范围之外的异常可能会关闭服务器。为了更好地加以组织，可以嵌套try块：

```
ServerSocket server = null;
try {
  server = new ServerSocket(port);
  while (true) {
    Socket connection = null;
    try {
      connection = server.accept();
      Writer out = new OutputStreamWriter(connection.getOutputStream());
      Date now = new Date();
      out.write(now.toString() +"\r\n");
      out.flush();
      connection.close();
    } catch (IOException ex) {
      // 仅这个请求，忽略
    } finally {
      try {
        if (connection != null) connection.close();
      } catch (IOException ex) {}
    }
  }
} catch (IOException ex) {
  ex.printStackTrace();
} finally {
  try {
    if (server != null) server.close();
  } catch (IOException ex) {}
}
```

结束处理时一定要关闭Socket。在第8章中我曾经说过，客户端不能依赖连接的另一端关闭Socket，对于服务器尤其如此。客户端可能超时或崩溃；用户可能取消事务；网络可能在流量高峰期间瘫痪；黑客可能发动拒绝服务攻击。出于诸如此类的众多原因，你不能依赖于客户端关闭Socket，即使协议有这个要求也不能完全相信客户端一定会关闭Socket（不过这个daytime协议没有要求客户端关闭Socket）。

示例9-1把这些汇集在一起。这里使用了Java 7的try-with-resources来自动关闭Socket。

示例9-1：daytime服务器

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

public class DaytimeServer {

  public final static int PORT = 13;

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

```
    try (ServerSocket server = new ServerSocket(PORT)) {
      while (true) {
        try (Socket connection = server.accept()) {
          Writer out = new OutputStreamWriter(connection.getOutputStream());
          Date now = new Date();
          out.write(now.toString() +"\r\n");
          out.flush();
          connection.close();
        } catch (IOException ex) {}
      }
    } catch (IOException ex) {
        System.err.println(ex);
    }
  }
}
```

这个类有一个方法main()，所有工作都由它完成。外部try块会捕获在daytime端口上构造ServerSocket对象server时可能产生的任何IOException。内部try块则监视接受和处理连接时可能抛出的异常。accept()方法在一个无限循环中调用来监视新连接。与很多服务器一样，这个程序不会终止，而是会继续监听，直到抛出一个异常或者你手动让它停止。

---

**提示：** 使用什么命令来手动停止程序，这取决于具体的系统。在UNIX、Windows 和其他许多系统中，Ctrl-C可以停止程序。在UNIX系统上，如果在后台运行服务器，就要找到服务器的进程ID并通过kill命令（**kill** *pid*）杀死该进程才能停止程序。

---

当客户端连接时，accept()会返回一个Socket，存储在局部变量connection中，程序继续执行。它调用getOutputStream()获得与这个Socket关联的输出流，然后将这个输出流串链到一个新的OutputStreamWriter out。由一个新的Date对象提供当前时间。通过使用write()，将其内容以时间字符串表示写入out，从而发送给客户端。

从Telnet连接应该能看到以下输出：

```
$ telnet localhost 13
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Sat Mar 30 16:15:10 EDT 2013
Connection closed by foreign host
```

---

**提示：** 如果在UNIX（包括Linux和Mac OS X）上运行这个程序，为了连接端口13，你需要以root身份来运行。如果不希望或者不能作为root用户运行这个程序，可以把端口号改为1024以上的某个端口，例如1313。

---

## 提供二进制数据

发送二进制的非文本数据并不难。只需要使用一个写byte数组的OutputStream，而不是写String的Writer。示例9-2展示了一个遵循RFC 868时间协议的迭代时间服务器。当客户端连接时，服务器发送一个4字节big-endian无符号整数，指出从GMT 1900年1月1日12:00 A.M.（纪元）后经过的秒数。再次说明，当前时间是通过创建一个新的Date对象获得的。不过，由于Java的Date类会计算自GMT 1970年1月1日12:00 A.M. 以后的毫秒数，而不是自GMT 1900年1月1日12:00 A.M.以后的秒数，因此必须经过一些转换。

示例9-2：时间服务器

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

public class TimeServer {

  public final static int PORT = 37;

  public static void main(String[] args) {

   // 时间协议将时间起点设置为1900年，
   // 而Date类设置为从1970年开始计算。利用下面的这个数
   // 可以在二者之间进行转换。
    long differenceBetweenEpochs = 2208988800L;

    try (ServerSocket server = new ServerSocket(PORT)) {
      while (true) {
        try (Socket connection = server.accept()) {
          OutputStream out = connection.getOutputStream();
          Date now = new Date();
          long msSince1970 = now.getTime();
          long secondsSince1970 = msSince1970/1000;
          long secondsSince1900 = secondsSince1970
              + differenceBetweenEpochs;
          byte[] time = new byte[4];
          time[0]
              = (byte) ((secondsSince1900 & 0x00000000FF000000L) >> 24);
          time[1]
              = (byte) ((secondsSince1900 & 0x0000000000FF0000L) >> 16);
          time[2]
              = (byte) ((secondsSince1900 & 0x000000000000FF00L) >> 8);
          time[3] = (byte) (secondsSince1900 & 0x00000000000000FFL);
          out.write(time);
          out.flush();
        } catch (IOException ex) {
          System.err.println(ex.getMessage());
        }
      }
    } catch (IOException ex) {
      System.err.println(ex);
    }
```

```
  }
}
```

与前一章的TimeClient一样，大多数工作都是在处理一个数据格式（32位无符号整数），Java没有直接提供对这个格式的支持。

## 多线程服务器

Daytime和time都是非常快的协议。服务器发送最多几十个字节，然后关闭连接。这里会完全处理各个连接，然后再转向下一个连接，这看起来很有道理。不过，即使是这样，也可能会有一个很慢或者崩溃的客户端使服务器挂起几秒的时间，直到它注意到socket已经中断。即使客户端和服务器都表现正常，如果发送数据可能花费大量时间，你肯定不希望每个连接都必须等待。

老式的UNIX服务器如wu-ftpd会创建一个新进程来处理各个连接，这样多个客户端可以同时得到服务。Java程序应当生成一个线程与客户端交互，这样服务器就能够尽快准备处理下一个连接。与完整的子进程相比，线程对服务器带来的负载更小。事实上，创建（fork）太多进程会有很大开销，这也是典型UNIX FTP服务器无法处理多于400个连接的原因，否则就会慢如蜗牛在爬。另一方面，如果协议很简单，处理很快，服务器完成处理后会关闭连接，那么服务器不需要生成线程，可以立即处理客户端请求，这样会更高效。

操作系统将把指向某个特定端口的入站连接请求存储在一个先进先出的队列中。默认地，Java将这个队列的长度设置为50，但不同的操作系统会有所不同。有些操作系统（不包括Solaris）有一个最大队列长度。例如，在FreeBSD上，默认的最大队列长度为128。在这些系统中，Java服务器socket的队列长度将是操作系统所允许的最大值（小于或等于50）。队列中填入的未处理连接达到其（最大）容量时，主机会拒绝这个端口上额外的连接，直到队列腾出新的位置为止。很多（但非全部）客户端在首次连接被拒绝后还会多次尝试建立连接。如果默认长度不够大，一些ServerSocket构造函数还允许改变这个队列的长度。不过，不能将队列长度增加到超过操作系统支持的最大大小。不论队列大小如何，你肯定希望能够比新连接到来的速度更快地清空队列，尽管处理各个连接需要花一些时间。

这里的解决方案是为每个连接提供它自己的一个线程，与接收入站连接放入队列的那个线程分开。例如，示例9-3是一个daytime服务器，它会生成一个新线程来处理每个入站连接。这样可以防止一个慢客户端阻塞所有其他客户端。这是一种“每个连接对应一个线程”的设计。

示例9-3：多线程daytime服务器

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

public class MultithreadedDaytimeServer {

  public final static int PORT = 13;

  public static void main(String[] args) {
    try (ServerSocket server = new ServerSocket(PORT)) {
      while (true) {
        try {
          Socket connection = server.accept();
          Thread task = new DaytimeThread(connection);
          task.start();
        } catch (IOException ex) {}
      }
      } catch (IOException ex) {
        System.err.println("Couldn't start server");
      }
    }

    private static class DaytimeThread extends Thread {
    private Socket connection;

    DaytimeThread(Socket connection) {
      this.connection = connection;
    }

    @Override
    public void run() {
      try {
        Writer out = new OutputStreamWriter(connection.getOutputStream());
        Date now = new Date();
        out.write(now.toString() +"\r\n");
        out.flush();
      } catch (IOException ex) {
        System.err.println(ex);
      } finally {
        try {
          connection.close();
        } catch (IOException e) {
          // 忽略
        }
      }
    }
  }
}
```

示例9-3使用try-with-resources来自动关闭服务器socket。不过，对于服务器socket接受的客户端socket这里有意没有使用try-with-resources。这是因为，客户端socket避开了try

块，而放在一个单独的线程中。如果使用了try-with-resources，主线程一旦到达while循环末尾就会关闭socket，而此时新生成的线程可能还没有用完这个socket。

不过，这个服务器上确实有可能发生一种拒绝服务攻击（denial-of-service）。由于示例9-3为每个连接生成一个新线程，大量几乎同时的入站连接可能导致它生成极大数量的线程。最终，Java虚拟机会耗尽内存而崩溃。一种更好的办法是如第3章所述，使用一个固定的线程池来限制可能的资源使用。50个线程应该足够了。不论负载有多大，示例9-4应该不会崩溃。它可能会拒绝连接，但起码不会崩溃。

示例9-4：使用线程池的daytime服务器

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

public class PooledDaytimeServer {

  public final static int PORT = 13;

  public static void main(String[] args) {

    ExecutorService pool = Executors.newFixedThreadPool(50);

    try (ServerSocket server = new ServerSocket(PORT)) {
      while (true) {
        try {
          Socket connection = server.accept();
          Callable<Void> task = new DaytimeTask(connection);
          pool.submit(task);
        } catch (IOException ex) {}
      }
    } catch (IOException ex) {
      System.err.println("Couldn't start server");
    }
  }

  private static class DaytimeTask implements Callable<Void> {

    private Socket connection;

    DaytimeTask(Socket connection) {
      this.connection = connection;
    }

    @Override
    public Void call() {
      try {
        Writer out = new OutputStreamWriter(connection.getOutputStream());
        Date now = new Date();
        out.write(now.toString() +"\r\n");
        out.flush();
```

```
      } catch (IOException ex) {
        System.err.println(ex);
      } finally {
        try {
          connection.close();
        } catch (IOException e) {
          // 忽略
        }
      }
      return null;
    }
  }
}
```

示例9-4的结构与示例9-3很类似。唯一的差别是它使用了一个Callable而不是一个Thread子类，而且并不是启动新线程，它会把这些callable提交给一个executor服务，这个服务预置为50个线程。

## 用Socket写入服务器

在目前为止的所有例子中，服务器只写入客户端socket，而没有从客户端读取。不过，大多数协议要求服务器同时读写客户端。这并不难。与从前一样，仍然要接受一个连接，不过这一次要同时获取一个InputStream和一个OutputStream。使用InputStream读取客户端，另外使用OutputStream写入客户端。关键是要理解协议：明确何时写入和何时读取。

RFC 862定义的echo协议是最简单的交互式TCP服务之一。客户端打开指向echo服务器端口7的socket，并发送数据。服务器将数据发回。这个过程一直继续，直到客户端关闭连接为止。echo协议很有用，可以测试网络，确保数据没有被有问题的路由器或防火墙所破坏。可以用Telnet试验echo，如下所示：

```
$ telnet rama.poly.edu 7
Trying 128.238.10.212...
Connected to rama.poly.edu.
Escape character is '^]'.
This is a test
This is a test
This is another test
This is another test
9876543210
9876543210
^]
telnet> close
Connection closed.
```

这个例子是面向行的，因为Telnet的工作方式就是如此。它从控制台读取一行输入，把它发送给服务器，然后等待读取返回的一行输出。不过，echo协议不需要这样。它会回

显接收到的每一个字节。它并不关心这些字节是否表示采用某种编码的字符，或者是否划分为多行。与很多协议不同，echo没有指定锁步行为，即客户端发送一个请求，然后在发送更多数据之前会等待完整的服务器响应。与daytime和time协议不同，在echo协议中，客户端要负责关闭连接。这使得利用多个线程支持异步操作更为重要，因为一个客户端可以保持无限期连接。在示例9-5中，服务器会生成最多500个线程。

示例9-5：echo服务器

```
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.util.*;
import java.io.IOException;

public class EchoServer {

  public static int DEFAULT_PORT = 7;

  public static void main(String[] args) {

    int port;
    try {
      port = Integer.parseInt(args[0]);
    } catch (RuntimeException ex) {
      port = DEFAULT_PORT;
    }
    System.out.println("Listening for connections on port " + port);

    ServerSocketChannel serverChannel;
    Selector selector;
    try {
      serverChannel = ServerSocketChannel.open();
      ServerSocket ss = serverChannel.socket();
      InetSocketAddress address = new InetSocketAddress(port);
      ss.bind(address);
      serverChannel.configureBlocking(false);
      selector = Selector.open();
      serverChannel.register(selector, SelectionKey.OP_ACCEPT);
    } catch (IOException ex) {
      ex.printStackTrace();
      return;
    }

    while (true) {
      try {
        selector.select();
      } catch (IOException ex) {
        ex.printStackTrace();
        break;
      }

      Set<SelectionKey> readyKeys = selector.selectedKeys();
      Iterator<SelectionKey> iterator = readyKeys.iterator();
```

```
      while (iterator.hasNext()) {
        SelectionKey key = iterator.next();
        iterator.remove();
        try {
          if (key.isAcceptable()) {
            ServerSocketChannel server = (ServerSocketChannel) key.channel();
            SocketChannel client = server.accept();
            System.out.println("Accepted connection from " + client);
            client.configureBlocking(false);
            SelectionKey clientKey = client.register(
                 selector, SelectionKey.OP_WRITE | SelectionKey.OP_READ);
            ByteBuffer buffer = ByteBuffer.allocate(100);
            clientKey.attach(buffer);
          }
          if (key.isReadable()) {
            SocketChannel client = (SocketChannel) key.channel();
            ByteBuffer output = (ByteBuffer) key.attachment();
            client.read(output);
          }
          if (key.isWritable()) {
            SocketChannel client = (SocketChannel) key.channel();
            ByteBuffer output = (ByteBuffer) key.attachment();
            output.flip();
            client.write(output);
            output.compact();
          }
        } catch (IOException ex) {
          key.cancel();
          try {
            key.channel().close();
          } catch (IOException cex) {}
        }
      }
    }
  }
}
```

## 关闭服务器Socket

如果使用完一个服务器Socket，就应当将它关闭，特别是当程序还要继续执行一段时间时更是如此。这会释放端口，使其他希望使用这个端口的程序可以使用。不要把关闭ServerSocket与关闭Socket混淆。关闭ServerSocket会释放本地主机的一个端口，允许另一个服务器绑定到这个端口。它还会中断该ServerSocket已经接受的目前处于打开状态的所有Socket。

程序结束时，服务器Socket会自动关闭，所以如果不再需要ServerSocket，很短时间后程序就会结束，在这些程序中，并不绝对需要关闭ServerSocket。尽管如此，关闭服务器socket也没有坏处。程序员通常会在一个try-finally块中采用close-if-not-null模式（如果非空就关闭），从前面关于流和客户端Socket的介绍你应该已经很熟悉这种模式了：

```
ServerSocket server = null;
try {
  server = new ServerSocket(port);
  // ... 处理服务器socket
} finally {
  if (server != null) {
    try {
      server.close();
    } catch (IOException ex) {
      // 忽略
    }
  }
}
```

可以使用无参数ServerSocket()构造函数稍加改进，无参数构造函数不抛出任何异常，也不绑定到一个端口。你需要在构造 ServerSocket()对象之后调用bind()方法来绑定一个Socket地址：

```
ServerSocket server = new ServerSocket();
try {
  SocketAddress address = new InetSocketAddress(port);
  server.bind(address);
  // ... 处理服务器socket
} finally {
  try {
    server.close();
  } catch (IOException ex) {
    // 忽略
  }
}
```

在Java 7中，ServerSocket实现了AutoCloseable，所以可以利用try-with-resources:

```
try (ServerSocket server = new ServerSocket(port)) {
  // ... 处理服务器socket
}
```

关闭一个服务器socket之后，不能再重新连接，即使是同一个端口也不可以。如果ServerSocket已经关闭，isClosed()方法返回true，否则返回false：

```
public boolean isClosed()
```

对于用无参数ServerSocket()构造函数创建而且尚未绑定到某个端口的ServerSocket对象，并不认为它们是关闭的。对这些对象调用isClosed()会返回false。isBound()方法会指出ServerSocket是否已绑定到一个端口：

```
public boolean isBound()
```

与第8章中讨论的Socket类的isBound()方法一样，这个方法名有些误导性。如果

ServerSocket曾经绑定到某个端口，即使它目前已关闭，isBound()仍会返回true。如果需要测试ServerSocket是否打开，就必须同时检查isBound()返回true，而且isClosed()返回false。例如：

```
public static boolean isOpen(ServerSocket ss) {
  return ss.isBound() && !ss.isClosed();
}
```

# 日志

服务器要在无人看管的情况下运行很长时间。通常需要在很久之后对服务器中发生的情况进行调试，这很重要。由于这个原因，建议在存储服务器日志，至少要存储一段时间的日志。

## 日志记录内容

日志中通常希望记录两个主要内容：

- 请求。
- 服务器错误。

实际上，服务器一般会为这两项内容维护两个不同的日志文件。审计日志中，对应与服务器建立的每一个连接会分别包含一个记录。每个连接完成多个操作的服务器可能对每个操作都有一个记录。例如，dict服务器可能会为客户端查找的每个单词建立一个日志记录。

错误日志则主要包含服务器运行期间发生的意外异常。例如，发生的所有NullPointerException都要记录在这个日志文件中，因为它指示了你要修复的服务器中的一个bug。错误日志不包含客户端错误，如意外断开连接或发送一个错误请求的客户端。

对于错误日志，一般经验是错误日志中的每一行都应当查看和解决。错误日志中的理想记录数应当是0。这个日志中的每个记录都表示一个需要研究和解决的bug。如果研究一个错误日志记录后，认为这个异常实际上不是个问题，代码仍能正常工作，那么可以删除这个日志条目。如果错误日志中充斥太多假警报，那么它很快就被忽视而变得毫无用处。

出于同样的原因，不要在生产阶段保存调试日志。不要把每一次进入一个方法、每次满足某个条件等情况都记入日志。没有人会看这些日志。这些记录只会浪费空间，而隐藏

真正的问题。如果需要方法级的日志来完成调试，可以把它们放在一个单独的文件里，在生产阶段运行时要在全局属性文件中将其关闭。

更高级的日志系统会提供一些日志分析工具，使你能够对日志做一些处理，如只显示优先级为INFO或有更高优先级的消息，或者只显示来自代码中某个特定部分的消息。由于有这些工具，维护单个日志文件或数据库就更为可行，甚至可以由多个不同的二进制文件或程序共享同一个日志。尽管如此，原则还是一样的，即如果一个日志记录没有人会看，那么它起码没有任何价值，而且通常会分散注意力或者让人混淆。

你可能会把你认为需要的所有一切都记入日志（万一以后有人可能需要），这是常见的一个反模式，不过不要遵循这个反模式。在实际中，程序员总想提前猜测可能需要记录哪些消息来调试生产阶段的问题，但他们这方面的能力往往很糟糕。一旦出现问题，有时需要哪些消息是很明显的，但很难提前预测需要这些消息。在日志文件中增加“万一需要的”消息通常意味着：一旦一个问题确实出现，你必须在一个充斥着无关数据的巨大海洋里努力搜寻。

## 如果记录日志

很多遗留程序可以追溯到Java 1.3或更早的版本，这些程序还在使用第三方日志库，如log4j或Apache Commons Logging，不过从Java 1.4以后已经提供了java.util.logging包，它能满足绝大多数需要。选择这个包可以避免很多复杂的第三方依赖问题。

尽管可以根据需要来加载日志工具，不过通常最容易的办法是为每个类创建一个日志工具，如下所示：

```
private final static Logger auditLogger = Logger.getLogger("requests");
```

日志工具是线程安全的，所以将它们存储在共享的静态字段中没有任何问题。实际上，往往必须这样做，因为即使不用在线程间共享Logger对象，日志文件或数据库也需要共享。这在大量使用多线程的服务器中非常重要。

这个例子输出到一个名为 “requests”的日志中。多个Logger对象可以输出到同一个日志，不过每个日志工具总是将日志记入某一个日志。这个日志是什么、放在哪里取决于外部配置。最常见的情况下，这将是一个文件，可能名为“requests”，也可能有其他名字。不过它也可能是一个数据库、一个在多个服务器上运行的SOAP服务、同一个主机上运行的另一个Java程序，或者是其他形式。

一旦有了一个日志工具，可以使用多个方法写入这个日志。最基本的是log()。例如，下面这个catch块会记录最高级的意外运行时异常：

```
    catch (RuntimeException ex) {
      logger.log(Level.SEVERE, "unexpected error " + ex.getMessage(), ex);
    }
```

可以包含异常而不只是一个消息，这是可选的，不过从catch块记录日志时这是一种惯常的做法。

java.util.logging.Level中的命名常量定义了7个级别，按严重性从高到低依次为：

- Level.SEVERE（最高值）
- Level.WARNING
- Level.INFO
- Level.CONFIG
- Level.FINE
- Level.FINER
- Level.FINEST（最低值）

我通常会对审计日志使用info（信息），对错误日志使用warning（警告）或severe（严重）。较低级别只用于调试，不要在生产系统中使用。对应info、severe和warning，都有一些方便的辅助方法可以记录相应级别的日志。例如，下面这个语句会留下一个记录，包括日期和远程地址：

```
logger.info(new Date() + " " + connection.getRemoteSocketAddress());
```

可以对各个日志记录使用任何方便的格式。一般来讲，每个记录应该包含一个时间戳、一个客户端地址，以及所处理的请求的任何特定信息。如果日志消息表示一个错误，则要包括抛出的特定异常。Java会自动填入记录这个消息所在的代码位置，所以这个方面你不用操心。

示例9-6展示了如何为daytime服务器增加日志记录。

示例9-6：记录请求和错误的daytime服务器

```
import java.io.*;
import java.net.*;
import java.util.Date;
import java.util.concurrent.*;
import java.util.logging.*;

public class LoggingDaytimeServer {

  public final static int PORT = 13;
  private final static Logger auditLogger = Logger.getLogger("requests");
```

```
  private final static Logger errorLogger = Logger.getLogger("errors");

  public static void main(String[] args) {

    ExecutorService pool = Executors.newFixedThreadPool(50);

    try (ServerSocket server = new ServerSocket(PORT)) {
      while (true) {
        try {
          Socket connection = server.accept();
          Callable<Void> task = new DaytimeTask(connection);
          pool.submit(task);
        } catch (IOException ex) {
          errorLogger.log(Level.SEVERE, "accept error", ex);
        } catch (RuntimeException ex) {
          errorLogger.log(Level.SEVERE, "unexpected error " + ex.getMessage(), ex);
        }
      }
    } catch (IOException ex) {
      errorLogger.log(Level.SEVERE, "Couldn't start server", ex);
    } catch (RuntimeException ex) {
      errorLogger.log(Level.SEVERE, "Couldn't start server: " + ex.getMessage(), ex);
    }
  }

  private static class DaytimeTask implements Callable<Void> {

    private Socket connection;

    DaytimeTask(Socket connection) {
      this.connection = connection;
    }

    @Override
    public Void call() {
      try {
        Date now = new Date();
        // 先写入日志记录以防万一客户断开连接
        auditLogger.info(now + " " + connection.getRemoteSocketAddress());
        Writer out = new OutputStreamWriter(connection.getOutputStream());
        out.write(now.toString() +"\r\n");
        out.flush();
      } catch (IOException ex) {
          // 客户断开连接，忽略
      } finally {
        try {
          connection.close();
        } catch (IOException ex) {
          // 忽略
        }
      }
      return null;
    }
  }
}
```

除了记录日志之外，示例9-6还为RuntimeException异常增加了catch块，其中涵盖了大多数代码和所有网络连接。这是网络服务器强烈推荐的一种做法。你肯定不希望只是由于一个请求进入计划外的代码路径而抛出一个IllegalArgumentException就导致整个服务器都崩溃。一般地，如果发生这种情况，这个请求会失败，不过你还能继续处理其他请求。如果更谨慎一些，还可以向客户端发送适当的错误响应。在HTTP中，这将是一个500级内部服务器错误。

并不是所有异常都会自动记入一个错误日志记录。例如，如果一个客户端断开连接，而你当时正在写入，这就是一个IOException。不过，这不是一个bug，也不是一个服务器错误，所以它不会写入到错误日志中。有些情况下，你可能希望把它记入审计日志，或者记录到另外某个位置。不过，一定要记住关于日志记录的金科玉律：如果没有人会看，就不要记入日志。除非你确实计划对客户断开连接深入研究或者做某些处理，否则就不要把它记入日志。

默认情况下，日志只是输出到控制台。例如，对于前面的服务器，我快速地连续连接几次时，会得到以下输出：

```
Apr 13, 2013 8:54:50 AM LoggingDaytimeServer$DaytimeTask call
INFO: Sat Apr 13 08:54:50 EDT 2013 /0:0:0:0:0:0:0:1:56665
Apr 13, 2013 8:55:08 AM LoggingDaytimeServer$DaytimeTask call
INFO: Sat Apr 13 08:55:08 EDT 2013 /0:0:0:0:0:0:0:1:56666
Apr 13, 2013 8:55:16 AM LoggingDaytimeServer$DaytimeTask call
INFO: Sat Apr 13 08:55:16 EDT 2013 /0:0:0:0:0:0:0:1:56667
```

你可能希望配置运行时环境，将日志放在一个更持久的目标位置。尽管可以在代码中指定，不过通常建议在一个配置文件中设置日志的位置，这样就能改变日志位置而无需重新编译代码。

java.util.logging.config.file系统属性采用常规的属性格式指向控制日志记录的一个文件。可以在启动虚拟机时传入-Djava.util.logging.config.file=_filename_参数来设置这个属性。例如，在Mac OS X上，可以在*Info.plist*文件中的VMOptions中设置：

```
<key>Java</key>
<dict>
  <key>VMOptions</key>
  <array>
    <string>-Djava.util.logging.config.file=/opt/daytime/logging.properties
            </string>
  </array>
</dict>
```

示例9-7是一个示例日志属性文件，指定了以下内容：

- 日志要写入一个文件。

- 请求日志应当在*/var/logs/daytime/requests.log*（Info级）。
- 错误日志应当在*/var/logs/daytime/requests.log*（Severe级）。
- 日志大小限制为约10MB，然后轮换。
- 维护两个日志：当前日志和之前的日志。
- 使用基本文本格式化工具（而不是XML）。
- 日志文件的每一行采用消息级时间戳（level message timestamp）。

示例9-7：日志属性文件

```
handlers=java.util.logging.FileHandler
java.util.logging.FileHandler.pattern = /var/logs/daytime/requests.log
java.util.logging.FileHandler.limit = 10000000
java.util.logging.FileHandler.count = 2
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.append = true
java.util.logging.SimpleFormatter.format=%4$s: %5$s [%1$tc]%n

requests.level = INFO
audit.level = SEVERE
```

下面是一个典型的日志输出（需要注意，看起来请求消息中时间戳写了两次，这是因为日志消息也包含当前时间。如果服务器有其他用途，而不是提供当前时间，通常不会是这样）：

```
SEVERE: Couldn't start server [Sat Apr 13 10:07:01 EDT 2013]
INFO: Sat Apr 13 10:08:05 EDT 2013 /0:0:0:0:0:0:0:1:57275
      [Sat Apr 13 10:08:05 EDT 2013]
INFO: Sat Apr 13 10:08:06 EDT 2013 /0:0:0:0:0:0:0:1:57276
      [Sat Apr 13 10:08:06 EDT 2013]
```

---

**提示：** 关于Java Logging API，我不喜欢的一点是：它没有提供一种简便的方法能单独通过配置来指定不同消息属于不同的日志。例如，不能很容易地区分错误和审计日志。区分这两个日志也是可以做到的，不过这要求你为每一个日志定义一个新的`FileHandler`子类，从而能够为它分配一个新文件。

---

最后一点，一旦配置了服务器可以记录日志，不要忘记查看这些日志，特别是错误日志。没有人会看的日志文件是没有任何意义的。你可能还要规划和实现一些日志轮换和保留策略。每年硬盘的规模都在增大，不过对于大容量服务器，如果不当心，文件系统仍有可能被大量日志数据填满。墨菲定律指出，这往往发生在新年第一天的凌晨4：00，而此时你可能正在外旅行。

## 构造服务器Socket

有4个公共的ServerSocket构造函数：

```
public ServerSocket(int port) throws BindException, IOException
public ServerSocket(int port, int queueLength)
    throws BindException, IOException
public ServerSocket(int port, int queueLength, InetAddress bindAddress)
throws IOException
public ServerSocket() throws IOException
```

这些构造函数可以指定端口、保存入站连接请求所用的队列的长度，以及要绑定的本地网络接口。它们所做的工作几乎是一样的，只是有些方法会使用队列长度和绑定地址的默认值。

例如，要在端口80创建一个HTTP服务器使用的服务器Socket，可以写为：

```
ServerSocket httpd = new ServerSocket(80);
```

要在端口80创建一个HTTP服务器使用的服务器Socket，而且队列一次最多可以保存50个入站连接，则写为：

```
ServerSocket httpd = new ServerSocket(80, 50);
```

如果试图将队列长度设置为大于操作系统的最大队列长度，则会使用最大队列长度。

默认地，如果一个主机有多个网络接口或IP地址，服务器Socket会在所有接口和IP地址的指定端口上监听。不过，还可以增加第3个参数，要求只绑定一个特定的本地IP地址。也就是说，服务器Socket只监听这个指定地址上的入站连接。它不会监听通过这个主机其他地址进入的连接。

例如，*login.ibiblio.org*是北卡罗来纳的一台特定的Linux机器。它用IP地址152.2.210.122连入Internet。这个机器还有第二块以太网卡，其本地IP地址是192.168.210.122，公共Internet看不至这个地址，只有本地网络能看到。如果出于某种原因，你希望在这个主机上运行一个服务器，它只响应同一个网络中的本地连接，可以创建一个服务器Socket，监听192.168.210.122上的5776端口，而不会监听152.2.210.122上的5776端口，如下：

```
InetAddress local = InetAddress.getByName("192.168.210.122");
ServerSocket httpd = new ServerSocket(5776, 10, local);
```

所有这3个构造函数中，可以为端口号传入0，这样系统就会为你选择可用的端口。像这样由系统选择的端口有时称为匿名端口（anonymous port），因为你提前不知道端口号是什么（但在端口选择之后可以查出）。这对于多socket协议（如FTP）往往很有用。在被动FTP中，客户端首先连接到服务器的已知端口21，所以服务器必须指定这个端口。

不过，需要传输一个文件时，服务器开始监听任何可用的端口。然后服务器会使用已经在端口21打开的命令连接告诉客户端，应当连接到另外哪一个端口来得到数据。因此，不同会话使用的数据端口可能会变化，而且不必提前知道（主动FTP是类似的，不过是由客户端监听服务器与之连接的一个临时端口，而不是反过来由服务器监听）。

所有这些构造函数都抛出一个IOException，确切地讲，如果无法创建socket并绑定到指定的端口，会抛出一个BindException。创建ServerSocket时如果抛出一个IOException，这往往说明两点。可能有另一个服务器socket（也许来自一个完全不同的程序）已经在使用请求的这个端口。或者可能你试图在UNIX上（包括Linux和Mac OS X）连接1～2013范围内的某个端口，但没有root（超级用户）权限。

可以利用这一点改写前一章中的LowPortScanner程序。这里不是尝试连接在一个给定端口上运行的服务器，而是在该端口打开一个服务器。如果端口已被占用，则尝试失败。示例9-8尝试在各个端口创建ServerSocket对象，查看哪些端口失败，以此来检查本地机器的端口情况。如果你使用的是UNIX，而且不作为root用户运行，那么这个程序只用于大于等于1024的端口。

示例9-8：查找本地端口

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

public class LocalPortScanner {

  public static void main(String[] args) {

    for (int port = 1; port <= 65535; port++) {
      try {
        // 如果这个端口上已经有服务器在运行，下一行
        // 就会失败，进入catch块
        ServerSocket server = new ServerSocket(port);
      } catch (IOException ex) {
        System.out.println("There is a server on port " + port + ".");
      }
    }
  }
}
```

下面是在我的Windows workstation上运行LocalPortScanner得到的结果：

```
D:\JAVA\JNP4\examples\9>java LocalPortScanner
There is a server on port 135.
There is a server on port 1025.
There is a server on port 1026.
There is a server on port 1027.
There is a server on port 1028.
```

## 构造但不绑定端口

无参数构造函数会创建一个ServerSocket对象，但未将它具体绑定到某个端口，所以初始时它不能接受任何连接。以后可以使用bind()来进行绑定：

```
public void bind(SocketAddress endpoint) throws IOException
public void bind(SocketAddress endpoint, int queueLength) throws IOException
```

这个特性的主要用途是，允许程序在绑定端口之前设置服务器socket选项。有些选项在服务器socket绑定后必须固定。一般的模式如下：

```
ServerSocket ss = new ServerSocket();
// 设置socket选项...
SocketAddress http = new InetSocketAddress(80);
ss.bind(http);
```

也可以为SocketAddress传入null来选择任意端口，这与其他构造函数中为端口号传入0的作用相似。

# 获得服务器Socket的有关信息

ServerSocket类提供了两个获取方法，可以指出这个服务器Socket占用的本地地址和端口。如果你在一个匿名端口和/或一个未指定的网络接口上打开一个服务器Socket，这会很有用。举例来说，在一个FTP会话的数据连接中可能就要用到这些获取方法：

```
public InetAddress getInetAddress()
```

这个方法会返回服务器（本地主机）使用的地址。如果本地主机有一个IP地址（大多数主机都是如此），这就是InetAddress.getLocalHost()返回的地址。如果本地主机有多个IP地址，返回的特定地址则是主机的IP地址之一。无法预料到底会得到哪个地址。例如：

```
ServerSocket httpd = new ServerSocket(80);
InetAddress ia = httpd.getInetAddress();
```

如果ServerSocket尚未绑定网络接口，这个方法返回null。

```
public int getLocalPort()
```

ServerSocket构造函数允许为端口号传递0，从而监听未指定的端口。利用这个方法就可以找出所监听的端口。在对等（端到端）的多Socket程序中，如果你已经有办法通知其他各方你所在的位置，就可以使用这个方法。或者一个服务器可能生成几个更小的服务器来完成特定的操作。这个已知的服务器会通知客户端将在哪些端口找到这些较小

的服务器。当然，也可以使用getLocalPort()查找一个非匿名端口，但为什么要这样做呢？如示例9-9所示。

示例9-9：随机端口

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

public class RandomPort {

  public static void main(String[] args) {
    try {
      ServerSocket server = new ServerSocket(0);
      System.out.println("This server runs on port "
           + server.getLocalPort());
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

下面是几次运行的输出：

```
$ java RandomPort
This server runs on port 1154
D:\JAVA\JNP4\examples\9>java RandomPort
This server runs on port 1155
D:\JAVA\JNP4\examples\9>java RandomPort
This server runs on port 1156
```

至少在这个系统上端口不是真正随机的，但在运行之前确实是不确定的。

如果ServerSocket没有绑定到某个端口，则getLocalPort()方法返回-1。

与大多数Java对象一样，还可以使用toString()方法打印一个ServerSocket。ServerSocket的toString()方法返回的String形式如下：

```
ServerSocket[addr=0.0.0.0,port=0,localport=5776]
```

addr是服务器Socket绑定的本地网络接口的地址。如果绑定到所有接口，则为0.0.0.0，大多数情况下都是如此。port总是0。localport是服务器监听连接的本地端口。这个方法有时对调试很有用，但对于其他方面用处不大。不要依赖这个方法。

# Socket选项

Socket选项指定了ServerSocket类所依赖的原生Socket如何发送和接收数据。对于服务器Socket，Java支持以下3个选项：

- SO_TIMEOUT
- SO_REUSEADDR
- SO_RCVBUF

另外还允许你为Socket的数据包设置性能首选项。

## SO_TIMEOUT

SO_TIMEOUT是accept()在抛出java.io.InterruptedIOException异常前等待入站连接的时间，以毫秒计。如果SO_TIMEOUT为0，accept()就永远不会超时。这个默认值的作用就是永远不超时。

很少会设置SO_TIMEOUT。如果你要实现一个复杂的安全协议，客户端和服务器之间需要有多个连接，响应会在一定的时间内出现，在这种情况下可能就会需要这个选项。不过，大多数服务器都设计为运行无限长的时间，因此只会使用默认的超时值，即0（永不超时）。如果你想改变这个设置，setSoTimeout()方法会设置服务器socket对象的SO_TIMEOUT字段：

```
public void setSoTimeout(int timeout) throws SocketException
public int getSoTimeout() throws IOException
```

调用accept()时开始倒数。如果超时，accept()就会抛出一个SocketTimeoutException，这是IOException的一个子类。要在调用accept()之前设置这个选项。accept()等待连接时不能改变超时值。timeout参数必须大于或等于0。否则这个方法会抛出一个IllegalArgumentException异常。例如：

```
try (ServerSocket server = new ServerSocket(port)) {
  server.setSoTimeout(30000); // 阻塞不超过30秒
  try {
    Socket s = server.accept();
    // 处理连接
    // ...
  } catch (SocketTimeoutException ex) {
    System.err.println("No connection within 30 seconds");
  }
} catch (IOException ex) {
  System.err.println("Unexpected IOException: " + e);
}
```

getSoTimeout()方法返回这个服务器Socket的当前SO_TIMEOUT值，例如：

```
public void printSoTimeout(ServerSocket server) {
  int timeout = server.getSoTimeOut();
  if (timeout > 0) {
    System.out.println(server + " will time out after "
```

```
            + timeout + "milliseconds.");
        } else if (timeout == 0) {
          System.out.println(server + " will never time out.");
        } else {
          System.out.println("Impossible condition occurred in " + server);
          System.out.println("Timeout cannot be less than zero." );
        }
      }
```

## SO_REUSEADDR

服务器Socket的SO_ REUSEADDR选项与客户端Socket的SO_ REUSEADDR选项（前一章讨论过）非常类似。它确定了是否允许一个新的Socket绑定到之前使用过的一个端口，而此时可能还有一些发送到原Socket的数据正在网络上传输。可以想见，有两个方法分别获取和设置这个选项：

```
public boolean getReuseAddress() throws SocketException
public void setReuseAddress(boolean on) throws SocketException
```

默认值依赖于具体平台。下面的代码段通过创建一个新的ServerSocket，然后调用getReuseAddress()来确定默认值：

```
ServerSocket ss = new ServerSocket(10240);
System.out.println("Reusable: " + ss.getReuseAddress());
```

我在Linux和Mac OS X机器上测试了这段代码，在这些系统上，服务器Socket默认是可重用的。

## SO_RCVBUF

SO_RCVBUF选项设置了服务器Socket接受的客户端Socket默认接收缓冲区大小。这个缓冲区大小由以下两个方法读/写：

```
public int getReceiveBufferSize() throws SocketException
public void setReceiveBufferSize(int size) throws SocketException
```

设置一个服务器的SO_RCVBUF就像在accept()返回的各个Socket上调用setReceiveBufferSize()（只不过Socket已经接受之后就不能改变接收缓冲区的大小了）。上一章曾经说过，这个选项为流中各个IP数据包的大小给出了一个建议值。更快的连接可能希望使用更大的缓冲区，不过大多数情况下默认值就足够了。

可以在绑定服务器Socket之前或之后设置这个选项，除非你想设置一个大于64KB的接收缓冲区大小。在这种情况下，必须在绑定之前，为未绑定的ServerSocket设置这个选项。例如：

```
ServerSocket ss = new ServerSocket();
int receiveBufferSize = ss.getReceiveBufferSize();
if (receiveBufferSize < 131072) {
  ss.setReceiveBufferSize(131072);
}
ss.bind(new InetSocketAddress(8000));
//...
```

## 服务类型

从上一章已经了解到，不同的Internet服务类型有不同的性能需求。例如，体育运动的直播视频需要相对较高的带宽。另一方面，电影可能仍需要高带宽，但是可以接受较大的延迟。电子邮件可以通过低带宽的连接传递，甚至延迟几个小时也不会造成大的危害。

为TCP定义了4个通用业务流类型：

- 低成本。
- 高可靠性。
- 最大吞吐量。
- 最小延迟。

可以为一个给定的Socket请求这些业务流类型。例如，可以请求低成本时可用的最小延迟。这些度量都是模糊的，而且是相对的，不能作为服务保证。并不是所有路由器和原生TCP栈都支持这些业务流类型。

对于服务器所接受的Socket，setPerformancePreferences()方法描述了为其连接时间、延迟和带宽给定的相对优先级：

```
public void setPerformancePreferences(int connectionTime, int latency,
    int bandwidth)
```

例如，通过将connectionTime设置为2，latency设置为1，bandwidth设置为3，这表示最大带宽是最重要的特性，最小延迟最不重要，连接时间居中：

```
ss.setPerformancePreferences(2, 1, 3);
```

至于给定的VM究竟如何实现，则取决于具体的Socket实现。底层Socket实现不要求考虑这些需求。它们只是为TCP栈提供了所需策略的一个提示。很多实现（包括Android）会完全忽略这些值。

# HTTP服务器

本节将展示可以用服务器Socket构建的几个不同的服务器，分别有不同的特殊用途，每一个都比前一个更复杂一些。

HTTP是个大协议。如你在第5章所见，提供完备功能的HTTP服务器必须响应文件请求，将URL转换为本地系统的文件名，响应POST和GET请求，处理不存在的文件的请求，解释MIME类型等。不过，很多HTTP服务器并不需要所有这些功能。例如，很多网站只是显示一个“正在建设中”的消息。很显然，Apache对于这样的网站是大材小用了。这样的网站完全可以使用只做一件事情的定制服务器。利用Java的网络类库，可以轻而易举地编写类似这样的简单服务器。

定制服务器不只是对小网站有用。大流量的网站如Yahoo!也可以使用定制服务器，因为与一般用途的服务器（如Apache或Microsoft IIS）相比，只做一件事情的服务器通常要快得多。针对某个特定任务来优化一个特殊用途的服务器会很容易，其结果往往比需要响应多种不同请求的一般用途服务器高效得多。例如，对于重复用于多个页面或大流量页面中的图标和图像，用一个单独的服务器处理会更好。这个服务器在启动时把所有图像文件读入内存，再从RAM中直接提供这些文件，而不是每次请求都从磁盘上读取。此外，对于包含图像的页面，如果你不想在这些页面请求之外单独记录这些图像请求，这个服务器还能避免在日志上浪费时间。

最后，要实现可与Apache或IIS等等匹敌的拥有完备功能Web服务器，Java也是一种不错的编程语言。即使你认为CPU密集的Java程序比CPU密集的C和C++程序慢（对于现代VM，我对此深表怀疑），但大多数HTTP服务器都是受带宽和延迟限制，而不是CPU速度的限制。因此，Java的其他优点，如半编译/半解释特性、动态类加载、垃圾回收和内存保护得到了发挥的机会。特别是有些网站通过servlet、PHP页面或其他机制大量使用了动态内容，如果在纯Java或接近纯Java的Web服务器上重新实现，通常会运行得更快。事实上，有很多生产质量的Web服务器就是用Java编写的，如Eclipse Foundation的Jetty。其他很多用C编写的Web服务器现在也包括了基本的Java组件，以支持Java Servlet API和Java Server Pages。它们正在大规模地取代传统的CGI、ASP和服务器端包含机制，主要是因为Java的相应机制速度更快、资源密集性更低。这里不打算研究这些技术，因为这些技术都需要专门的书来介绍。感兴趣的读者可以参考Jason Hunter的《Java Servlet Programming》（O'Reilly出版）。不过需要指出重要的一点，对于一般意义上的服务器以及特殊意义上的Web服务器，在实际性能方面，这确实是Java可与C竞争的一个领域。

## 单文件服务器

要研究HTTP服务器，我们先从一个简单的服务器开始，无论接收什么请求，这个服务器都始终发送同一个文件。这个单文件服务器名为SingleFileHTTPServer，如示例9-10所示。文件名、本地端口和内容编码方式都从命令行读取。如果省略端口，则假定为端口80。如果省略编码方式，则假定为ASCII。

示例9-10：提供同一个文件的HTTP服务器

```
import java.io.*;
import java.net.*;
import java.nio.charset.Charset;
import java.nio.file.*;
import java.util.concurrent.*;
import java.util.logging.*;

public class SingleFileHTTPServer {

  private static final Logger logger = Logger.getLogger("SingleFileHTTPServer");

  private final byte[] content;
  private final byte[] header;
  private final int port;
  private final String encoding;

  public SingleFileHTTPServer(String data, String encoding,
      String mimeType, int port) throws UnsupportedEncodingException {
    this(data.getBytes(encoding), encoding, mimeType, port);
  }

  public SingleFileHTTPServer(
      byte[] data, String encoding, String mimeType, int port) {
    this.content = data;
    this.port = port;
    this.encoding = encoding;
    String header = "HTTP/1.0 200 OK\r\n"
        + "Server: OneFile 2.0\r\n"
        + "Content-length: " + this.content.length + "\r\n"
        + "Content-type: " + mimeType + "; charset=" + encoding + "\r\n\r\n";
    this.header = header.getBytes(Charset.forName("US-ASCII"));
  }

  public void start() {
    ExecutorService pool = Executors.newFixedThreadPool(100);
    try (ServerSocket server = new ServerSocket(this.port)) {
      logger.info("Accepting connections on port " + server.getLocalPort());
      logger.info("Data to be sent:");
      logger.info(new String(this.content, encoding));

      while (true) {
        try {
          Socket connection = server.accept();
          pool.submit(new HTTPHandler(connection));
```

```
      } catch (IOException ex) {
        logger.log(Level.WARNING, "Exception accepting connection", ex);
      } catch (RuntimeException ex) {
        logger.log(Level.SEVERE, "Unexpected error", ex);
      }
    }
  } catch (IOException ex) {
    logger.log(Level.SEVERE, "Could not start server", ex);
  }
}

private class HTTPHandler implements Callable<Void> {
  private final Socket connection;

  HTTPHandler(Socket connection) {
    this.connection = connection;
  }

  @Override
  public Void call() throws IOException {
    try {
      OutputStream out = new BufferedOutputStream(
                             connection.getOutputStream()
                             );
      InputStream in = new BufferedInputStream(
                             connection.getInputStream()
                             );
      // 只读取第一行，这是我们需要的全部内容
      StringBuilder request = new StringBuilder(80);
      while (true) {
        int c = in.read();
        if (c == '\r' || c == '\n' || c == -1) break;
        request.append((char) c);
      }
      // 如果是HTTP/1.0或以后版本，则发送一个MIME首部
      if (request.toString().indexOf("HTTP/") != -1) {
        out.write(header);
      }
      out.write(content);
      out.flush();
    } catch (IOException ex) {
      logger.log(Level.WARNING, "Error writing to client", ex);
    } finally {
      connection.close();
    }
    return null;
  }
}

public static void main(String[] args) {

  // 设置要监听的端口
  int port;
  try {
    port = Integer.parseInt(args[1]);
```

```
      if (port < 1 || port > 65535) port = 80;
    } catch (RuntimeException ex) {
      port = 80;
    }
      String encoding = "UTF-8";
      if (args.length > 2) encoding = args[2];

      try {
        Path path = Paths.get(args[0]);;
        byte[] data = Files.readAllBytes(path);

        String contentType = URLConnection.getFileNameMap().getContentTypeFor(args[0]);
        SingleFileHTTPServer server = new SingleFileHTTPServer(data, encoding,
            contentType, port);
        server.start();

    } catch (ArrayIndexOutOfBoundsException ex) {
      System.out.println(
          "Usage: java SingleFileHTTPServer filename port encoding");
    } catch (IOException ex) {
      logger.severe(ex.getMessage());
    }
  }
}
```

构造函数建立一些数据，这些数据与HTTP首部一起发送，HTTP首部中包含有内容长度和内容编码方式的有关信息。响应的首部和主体以所需的编码方式存储在byte数组中，从而可以快速地发送给客户端。

SingleFileHTTPServer类包含要发送的内容、要发送的首部，以及要绑定的端口。start()方法在指定端口创建一个ServerSocket，然后进入一个无限循环，它会不断接受连接并进行处理。

每个入站socket由一个Runnable Handler对象处理，这个Handler对象要提交到一个线程池。因此，即使有一个运行很慢的客户端，也不会让其他客户端处于饥饿状态。每个Handler会得到一个InputStream，并由这个流读取客户端请求。它会查看第一行，来看其中是否包含字符串HTTP。如果看到这个字符串，服务器就认为客户端理解HTTP/1.0或以后版本，因此为该文件发送一个MIME首部，然后发送数据。如果客户端请求不包含字符串HTTP，服务器就忽略首部，直接发送数据。最后，handler关闭连接。

main()方法只是从命令行读取参数。从第一个命令行参数读取要提供的文件名。如果没有指定文件或者如果文件无法打开，就显示一条错误消息，程序退出。假设文件能够读取，就使用Java 7引入的Path和Files类将其内容读入byte数组data。URLConnection类对文件的内容类型做出合理的猜测，猜测结果存储在contentType变量中。接下来，从第二个命令行参数读取端口号。如果没有指定端口或者第二个参数不是1到65535之间的某个整数，则使用端口80。再从第三个命令行参数读取编码方式（如果给出

了第三个命令行参数）。否则，编码方式就假定为UTF-8。然后使用这些值构造一个SingleFileHTTPServer对象并启动。

main()方法是唯一的接口。可以很容易地将这个类用于其他程序。如果增加一个修改内容的设置方法，就可以很容易地用它来提供一个运行中服务器或系统的简单状态信息。不过，这会引起额外的一些线程安全问题，示例9-10无需解决这些问题，因为这里的数据是不可变的。

下面是通过Telnet连接这个服务器时看到的结果（具体细节取决于具体的服务器和文件）：

```
% telnet macfaq.dialup.cloud9.net 80
Trying 168.100.203.234...
Connected to macfaq.dialup.cloud9.net.
Escape character is '^]'.
GET / HTTP/1.0
HTTP/1.0 200 OK
Server: OneFile 2.0
Content-length: 959
Content-type: text/html; charset=UTF-8

&lt;!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN"&gt;
&lt;HTML&gt;
&lt;HEAD&gt;
&lt;TITLE&gt;Under Construction&lt;/TITLE&gt;
&lt;/HEAD&gt;

&lt;BODY&gt;
...
```

## 重定向器 Redirector

重定向（Redirection）是特殊用途HTTP服务器的另一个简单而有用的应用程序。这一节将开发另外一个服务器，它能将用户从一个Web网站重定向到另一个网站，例如，从*cnet.com*重定向到*www.cnet.com*。示例9-11从命令行读取URL和端口号，在这个端口打开一个服务器Socket，使用302 FOUND编码将接收的所有请求重定向到新URL表示的网站。在这个例子中，我选择为每个连接使用一个新线程，而没有使用线程池。这样一来，编写代码和理解起来可能更容易一些，但效率有些低。

示例9-11：HTTP重定向器

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

public class Redirector {
```

```
private static final Logger logger = Logger.getLogger("Redirector");

private final int port;
private final String newSite;

public Redirector(String newSite, int port) {
  this.port = port;
  this.newSite = newSite;
}

public void start() {
  try (ServerSocket server = new ServerSocket(port)) {
    logger.info("Redirecting connections on port "
        + server.getLocalPort() + " to " + newSite);

    while (true) {
      try {
        Socket s = server.accept();
        Thread t = new RedirectThread(s);
        t.start();
      } catch (IOException ex) {
        logger.warning("Exception accepting connection");
      } catch (RuntimeException ex) {
        logger.log(Level.SEVERE, "Unexpected error", ex);
      }
    }
  } catch (BindException ex) {
    logger.log(Level.SEVERE, "Could not start server.", ex);
  } catch (IOException ex) {
    logger.log(Level.SEVERE, "Error opening server socket", ex);
  }
}

private class RedirectThread extends Thread {

  private final Socket connection;

  RedirectThread(Socket s) {
    this.connection = s;
  }

  public void run() {
    try {
      Writer out = new BufferedWriter(
                   new OutputStreamWriter(
                    connection.getOutputStream(), "US-ASCII"
                   )
                  );
      Reader in = new InputStreamReader(
                  new BufferedInputStream(
                   connection.getInputStream()
                  )
                 );

      // 只读取第一行，这就是我们需要的全部内容
```

```
      StringBuilder request = new StringBuilder(80);
      while (true) {
        int c = in.read();
        if (c == '\r' || c == '\n' || c == -1) break;
        request.append((char) c);
      }

      String get = request.toString();
      String[] pieces = get.split("\\w*");
      String theFile = pieces[1];

      // 如果是HTTP/1.0或以后版本，则发送一个MIME首部
      if (get.indexOf("HTTP") != -1) {
        out.write("HTTP/1.0 302 FOUND\r\n");
        Date now = new Date();
        out.write("Date: " + now + "\r\n");
        out.write("Server: Redirector 1.1\r\n");
        out.write("Location: " + newSite + theFile + "\r\n");
        out.write("Content-type: text/html\r\n\r\n");
        out.flush();
      }
      // 并不是所有浏览器都支持重定向，所以我们需要
      // 生成HTML指出文档转移到哪里
      out.write("<HTML><HEAD><TITLE>Document moved</TITLE></HEAD>\r\n");
      out.write("<BODY><H1>Document moved</H1>\r\n");
      out.write("The document " + theFile
        + " has moved to\r\n<A HREF=\"" + newSite + theFile + "\">"
        + newSite + theFile
        + "</A>.\r\n Please update your bookmarks<P>");
      out.write("</BODY></HTML>\r\n");
      out.flush();
      logger.log(Level.INFO,
          "Redirected " + connection.getRemoteSocketAddress());
    } catch(IOException ex) {
      logger.log(Level.WARNING,
          "Error talking to " + connection.getRemoteSocketAddress(), ex);
    } finally {
      try {
        connection.close();
      } catch (IOException ex) {}
    }
  }
}

public static void main(String[] args) {

  int thePort;
  String theSite;
  try {
    theSite = args[0];
    // 删除末尾的斜线
    if (theSite.endsWith("/")) {
      theSite = theSite.substring(0, theSite.length() - 1);
    }
  } catch (RuntimeException ex) {
```

```
      System.out.println(
          "Usage: java Redirector http://www.newsite.com/ port");
      return;
    }

    try {
      thePort = Integer.parseInt(args[1]);
    } catch (RuntimeException ex) {
      thePort = 80;
    }

    Redirector redirector = new Redirector(theSite, thePort);
    redirector.start();
  }
}
```

为了在端口80启动重定向器，将入站请求重定向到*http://www.cafeconleche.org/*，要输入：

```
D:\JAVA\JNP4\examples\09&gt; java Redirector http://www.cafeconleche.org/
Redirecting connections on port 80 to http://www.cafeconleche.org/
```

如果通过Telnet连接这个服务器，将看到如下的结果：

```
% <userinput moreinfo="none">telnet macfaq.dialup.cloud9.net 80
Trying 168.100.203.234...
Connected to macfaq.dialup.cloud9.net.
Escape character is '^]'.
GET / HTTP/1.0
HTTP/1.0 302 FOUND
Date: Sun Mar 31 12:38:42 EDT 2013
Server: Redirector 1.1
Location: http://www.cafeconleche.org/
Content-type: text/html

&lt;HTML&gt;&lt;HEAD&gt;&lt;TITLE&gt;Document moved&lt;/TITLE&gt;&lt;/HEAD&gt;
&lt;BODY&gt;&lt;H1&gt;Document moved&lt;/H1&gt;
The document / has moved to
&lt;A HREF="http://www.cafeconleche.org/"&gt;http://www.cafeconleche.
org/&lt;/A&gt;.
Please update your bookmarks&lt;P&gt;&lt;/BODY&gt;&lt;/HTML&gt;
Connection closed by foreign host.
```

不过，如果用一个Web浏览器来连接，稍有些延迟后就会来到*http://www.cafeconleche.org/*。你绝对不会看到响应码后添加的HTML。这只是为了支持那些无法自动重定向的老浏览器而提供的，另外有些在安全方面过于多疑的人也可能把浏览器配置为不允许自动重定向。

`main()`方法提供了一个非常简单的接口，会读取新网站的URL（将把连接重定向至这个新网站）和要监听的本地端口。它使用这些信息构造了一个`Redirector`对象。然后

调用start()。如果没有指定端口，Redirector将在端口80监听。如果省略了网站，Redirector将显示一个错误消息并退出。

Redirector的start()方法将服务器Socket绑定到这个端口，显示一个简短的状态消息，然后进入无限循环，监听连接。每次接受连接时，得到的Socket对象会用来构造一个RedirectThread。然后启动这个RedirectThread。所有与客户端的进一步交互都在这个新线程中完成。start()方法只是等待下一个入站连接。

RedirectThread的run()方法完成了大部分工作。它先把一个Writer串链到Socket的输出流，把一个Reader串链到Socket的输入流。输出流和输出流都有缓冲。然后run()方法读取客户端发送的第一行。虽然客户端可能会发送整个MIME首部，但可以将其忽略。第一行包含所需的全部信息。这一行可能如下所示：

```
GET /directory/filename.html HTTP/1.0
```

可能第一个词是POST或PUT，或者没有HTTP版本。第二个“词”是客户端希望获取的文件。它必须以斜线（/）开头。浏览器负责将相对URL转换为以斜线开头的绝对URL，服务器不会做这件事。第三个词是浏览器理解的HTTP的版本。可能的值包括什么都没有（HTTP/1.0之前的浏览器）、HTTP/1.0或HTTP/1.1。

为处理这样的请求，Redirector会忽略第一个词。第二个词附加到目标服务器的URL（存储在字段newSite中）之后，从而得到完整的重定向URL。第三个词用于确定是否发送MIME首部，不理解HTTP/1.0的老浏览器不使用MIME首部。如果存在版本（第三个词），则发送MIME首部，否则将它省略。

发送数据相当简单。使用Writer out即可。由于发送的所有数据都是纯ASCII，因此具体的编码方式并不十分重要。这里唯一的技巧是HTTP请求的行结束字符是\r\n，也就是一个回车加一个换行。

后面的代码分别向客户端发送一行文本。第一行显示为：

```
HTTP/1.0 302 FOUND
```

这是一个HTTP/1.0响应码，告诉客户端要重定向。第二行是“Date:”首部，给出服务器的当前时间。这一行是可选的。第三行是服务器的名和版本；这一行也是可选的，但蜘蛛搜索程序（spider）可以用它对最流行的Web服务器做出统计。下一行是“Location:”首部，对于这个响应类型这是必需的。它告诉客户端要重定向到哪里。然后是标准的“Content-type:”首部。这里会发送内容类型text/html，指示客户端将会看到HTML。最后，发送一个空行来标识首部数据结束。

首部数据之后都是HTML，将由浏览器处理并显示给用户。接下来的几行是为不支持重定向的浏览器显示一个消息，使那些用户可以手动跳转到新网站。这个消息可能如下：

```
<HTML><HEAD><TITLE>Document moved</TITLE></HEAD>
<BODY><H1>Document moved</H1>
The document / has moved to
<A HREF="http://www.cafeconleche.org/">http://www.cafeconleche.org/</A>.
Please update your bookmarks<P></BODY></HTML>
```

最后，连接关闭，线程结束。

## 功能完备的HTTP服务器

特殊用途的HTTP服务器就介绍到这里。这一节将开发一个功能完备的HTTP服务器，名为JHTTP，它可以提供一个完整的文档树，包括图像、applet、HTML文件、文本文件等。它与SingleFileHTTPServer非常相似，只不过它所关注的是GET请求。这个服务器仍是相当轻量级的，看过这个代码后，我们将讨论可能希望添加的其他特性。

由于这个服务器可能必须通过很慢的网络连接从文件系统读取和提供大文件，所以要改变其方式。这里不再在执行主线程中处理到达的每一个请求，而是将入站连接放入一个池。由RequestProcessor类的不同实例从池中删除连接并进行处理。示例9-12展示了JHTTP主类。如前两个示例中一样，JHTTP的main()方法处理初始化，其他程序可以使用这个类运行基本的Web服务器。

示例9-12：JHTTP Web服务器

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

public class JHTTP {

  private static final Logger logger = Logger.getLogger(
      JHTTP.class.getCanonicalName());

  private static final int NUM_THREADS = 50;
  private static final String INDEX_FILE = "index.html";

  private final File rootDirectory;
  private final int port;

  public JHTTP(File rootDirectory, int port) throws IOException {

    if (!rootDirectory.isDirectory()) {
      throw new IOException(rootDirectory
          + " does not exist as a directory");
    }
    this.rootDirectory = rootDirectory;
```

```
    this.port = port;
  }

  public void start() throws IOException {
    ExecutorService pool = Executors.newFixedThreadPool(NUM_THREADS);
    try (ServerSocket server = new ServerSocket(port)) {
      logger.info("Accepting connections on port " + server.getLocalPort());
      logger.info("Document Root: " + rootDirectory);

      while (true) {
        try {
          Socket request = server.accept();
          Runnable r = new RequestProcessor(
              rootDirectory, INDEX_FILE, request);
          pool.submit(r);
        } catch (IOException ex) {
          logger.log(Level.WARNING, "Error accepting connection", ex);
        }
      }
    }
  }

  public static void main(String[] args) {

    // 得到文档根
    File docroot;
    try {
      docroot = new File(args[0]);
    } catch (ArrayIndexOutOfBoundsException ex) {
      System.out.println("Usage: java JHTTP docroot port");
      return;
    }

    // 设置要监听的端口
    int port;
    try {
      port = Integer.parseInt(args[1]);
      if (port < 0 || port > 65535) port = 80;
    } catch (RuntimeException ex) {
      port = 80;
    }

    try {
      JHTTP webserver = new JHTTP(docroot, port);
      webserver.start();
    } catch (IOException ex) {
      logger.log(Level.SEVERE, "Server could not start", ex);
    }
  }
}
```

JHTTP类的main()方法根据args[0]设置文档根目录。端口从args[1]读取，或者使用默认的80。然后构造一个新的JHTTP对象并启动。JHTTP创建一个线程池来处理请求，并反复接受入站连接。对于每一个入站连接，将向池中提交一个RequestProcessor线程。

每个连接由示例9-13所示RequestProcessor类的run()方法处理。这个方法从socket得到输入和输出流，将它们分别串链到一个阅读器和一个书写器。阅读器读取客户端请求的第一行，确定客户端支持的HTTP版本（只在HTTP/1.0或以后版本时才发送MIME首部），并确定被请求的文件。假如请求方法是GET，所请求的文件将转换为本地文件系统中的一个文件名。如果所请求的文件是一个目录（即文件名以斜线结束），就添加一个索引文件名。这里使用了一个规范路径，确保请求的文件不会超出文档根目录之外。否则，恶意的客户端会在URL中包括“..”沿着目录层次走遍整个本地文件系统。这就是需要从客户端得到的全部信息，不过更高级的Web服务器（尤其是记录命中率的服务器）还会读取客户端所发送MIME首部的其他部分。

接下来，打开所请求的文件，将其内容读取到一个byte数组。如果HTTP版本是1.0或以后版本，则在输出流中写入适当的MIME首部。为了确定内容类型，要调用URLConnection.getFileNameMap().getContentTypeFor(fileName)方法，将文件扩展名（如*.html*）映射为MIME类型（如text/html）。包含文件内容的byte数组要写入输出流，然后关闭连接。如果文件无法找到或无法打开，则向客户端发送一个404响应。如果客户端发送了服务器不支持的一个方法，如POST，则要发回一个501错误。如果出现异常，需要记入日志，关闭连接，然后再继续。

示例9-13：处理HTTP请求的runnable类

```
import java.io.*;
import java.net.*;
import java.nio.file.Files;
import java.util.*;
import java.util.logging.*;

public class RequestProcessor implements Runnable {

  private final static Logger logger = Logger.getLogger(
      RequestProcessor.class.getCanonicalName());

  private File rootDirectory;
  private String indexFileName = "index.html";
  private Socket connection;

  public RequestProcessor(File rootDirectory,
      String indexFileName, Socket connection) {

    if (rootDirectory.isFile()) {
      throw new IllegalArgumentException(
          "rootDirectory must be a directory, not a file");
    }
    try {
      rootDirectory = rootDirectory.getCanonicalFile();
    } catch (IOException ex) {
    }
    this.rootDirectory = rootDirectory;
```

```
    if (indexFileName != null) this.indexFileName = indexFileName;
    this.connection = connection;
  }

  @Override
  public void run() {
    // 安全检查
    String root = rootDirectory.getPath();
    try {
      OutputStream raw = new BufferedOutputStream(
                         connection.getOutputStream()
                        );
      Writer out = new OutputStreamWriter(raw);
      Reader in = new InputStreamReader(
                  new BufferedInputStream(
                   connection.getInputStream()
                  ),"US-ASCII"
                 );
      StringBuilder requestLine = new StringBuilder();
      while (true) {
        int c = in.read();
        if (c == '\r' || c == '\n') break;
        requestLine.append((char) c);
      }

      String get = requestLine.toString();

      logger.info(connection.getRemoteSocketAddress() + " " + get);
      String[] tokens = get.split("\\s+");
      String method = tokens[0];
      String version = "";
      if (method.equals("GET")) {
        String fileName = tokens[1];
        if (fileName.endsWith("/")) fileName += indexFileName;
        String contentType =
            URLConnection.getFileNameMap().getContentTypeFor(fileName);
        if (tokens.length > 2) {
          version = tokens[2];
        }

        File theFile = new File(rootDirectory,
            fileName.substring(1, fileName.length()));

        if (theFile.canRead()
            // 不要让客户端超出文档根之外
            && theFile.getCanonicalPath().startsWith(root)) {
          byte[] theData = Files.readAllBytes(theFile.toPath());
          if (version.startsWith("HTTP/")) { // 发送一个MIME首部
            sendHeader(out, "HTTP/1.0 200 OK", contentType, theData.length);
          }

          // 发送文件，这可能是一个图像或其他二进制数据，
          // 所以要使用底层输出流，
          // 而不是writer
          raw.write(theData);
```

```
            raw.flush();
          } else { // 无法找到文件
            String body = new StringBuilder("<HTML>\r\n")
                .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
                .append("</HEAD>\r\n")
                .append("<BODY>")
                .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
                .append("</BODY></HTML>\r\n").toString();
            if (version.startsWith("HTTP/")) { // 发送一个MIME首部
              sendHeader(out, "HTTP/1.0 404 File Not Found",
                  "text/html; charset=utf-8", body.length());
            }
            out.write(body);
            out.flush();
          }
        } else { // 方法不等于"GET"
          String body = new StringBuilder("<HTML>\r\n")
              .append("<HEAD><TITLE>Not Implemented</TITLE>\r\n")
              .append("</HEAD>\r\n")
              .append("<BODY>")
              .append("<H1>HTTP Error 501: Not Implemented</H1>\r\n")
              .append("</BODY></HTML>\r\n").toString();
          if (version.startsWith("HTTP/")) { // 发送一个MIME首部
            sendHeader(out, "HTTP/1.0 501 Not Implemented",
              "text/html; charset=utf-8", body.length());
          }
          out.write(body);
          out.flush();
        }
      } catch (IOException ex) {
        logger.log(Level.WARNING,
            "Error talking to " + connection.getRemoteSocketAddress(), ex);
      } finally {
        try {
          connection.close();
        }
        catch (IOException ex) {}
      }
    }

    private void sendHeader(Writer out, String responseCode,
        String contentType, int length)
        throws IOException {
      out.write(responseCode + "\r\n");
      Date now = new Date();
      out.write("Date: " + now + "\r\n");
      out.write("Server: JHTTP 2.0\r\n");
      out.write("Content-length: " + length + "\r\n");
      out.write("Content-type: " + contentType + "\r\n\r\n");
      out.flush();
    }
}
```

这个服务器可以提供相当多的功能，但仍然十分简单。还可以添加以下的一些特性：

- 服务器管理界面。
- 支持Java Servlet API。
- 支持其他请求方法，如POST、HEAD和PUT。
- 支持多个文档根目录，这样各个用户可以有自己的网站。

最后，再花点时间考虑一下可以采用什么方法来优化这个服务器。如果真的希望使用JHTTP来运行一个高流量的网站，还可以做一些事情来加速这个服务器。最主要的是实现智能缓存。跟踪记录接收到的请求，将最频繁请求的文件中的数据存储在一个Map中，使它们保存在内存中。使用一个低优先级的线程更新这个缓存。还可以使用非阻塞I/O和通道来代替线程和流。第11章将讨论这种选择。

# 第10章

# 安全Socket

美国电话电报公司（AT&T）允许美国国家安全局完全访问其客户的Internet业务流，可以将数据包复制到安装在国家安全局交换中心秘密房间里的数据挖掘设备[注1]。英国的政府通信总部（GCHQ）可以监听承载全世界大部分电话呼叫和Internet业务流的光纤电缆[注2]。瑞典的National Defence Radio Establishment则要求光纤电缆所有者在其处所安装光纤镜像设备。而这些只是我们所知道的由政府发起的监听行为中的几个例子而已。

作为一个Internet用户，你确实有一些保护手段可以防范官方的监视。为了使Internet连接从根本上更加安全，可以对Socket加密。这可以保持事务的机密性、真实性和准确性。

不过，加密是一个复杂的主题。要正确地进行加密，不仅需要详细地理解加密数据所用的数学算法，而且还要深入地理解交换密钥和加密数据的协议。即使小小的错误也会在你的装甲上打开一个大洞，把你的通信暴露给窃听者。因此，编写加密软件的工作最好留给专家来做。幸运的是，对底层协议和算法只有简单了解（或不太了解）的非专业人士也能用专家设计的软件来保护其通信的安全。你每次在在线商店下订单时，你的交易可能都通过了一些加密和认证，其中用到了一些协议和算法，而你几乎不需要对此有任何了解。作为程序员，如果想编写一个与在线商店通信的网络客户端软件，就需要对所涉及的协议和算法有所了解，但是假如你能使用通晓所有细节的专家们编写的类库，那么对这些协议和算法的了解并不需要太多。如果你想编写运行在线商店的服务器，就需要多了解一些，但与不参考其他工作从头设计所有内容相比，要了解的还是少多了。

---

注1：Ryan Singel, "Whistle-Blower Outs NSA Spy Room," Wired. April 7, 2006.

注2：Ewen MacAskill, Julian Borger, Nick Hopkins, Nick Davies, and James Ball, "GCHQ taps fibre-optic cables for secret access to world's communications," The Guardian. June 21, 2013.

Java安全Socket扩展（Java Secure Sockets Extension，JSSE）可以使用安全Socket层（Secure Sockets Layer，SSL）版本3和传输层安全（Transport Layer Security，TLS）协议及相关算法来保护网络通信的安全。SSL是一种安全协议，允许Web浏览器和其他TCP客户端基于各种级别的机密性和认证与HTTP和其他TCP服务器对话。

## 保护通信

经过开放通道（如公共Internet）的秘密通信绝对需要对数据加密。适合计算机实现的大多数加密机制都是基于密钥思想的，密钥是一种更加一般化的口令，并不限于文本。明文消息根据一种数学算法与密钥的各个位组合，生成加密的密文。使用的密钥位数越多，通过强力猜测密钥的方法来解密消息时就会越困难。

在传统的秘密密钥（或对称密钥）加密中，加密和解密数据都使用相同的密钥。发送方和接收方必须知道这个密钥。假设Angela希望给Gus发送一个秘密消息。她首先向Gus发送将用于交换秘密的密钥。但是密钥不能加密，因为Gus还没有密钥，所以Angela必须发送未加密的密钥。现在假定Edgar在监听Angela和Gus之间的连接。他会在Gus得到密钥的同时也得到这个密钥。此后，他就可以使用这个密钥读取Angela和Gus相互之间所说的所有内容。

在公开密钥（或非对称密钥）加密中，加密和解密数据使用不同的密钥。一个密钥称为公开密钥（public key），用于加密数据。这个密钥可以提供给任何人。另一个密钥称为私有密钥（private key），用于解密数据。私有密钥必须秘密保存，只有通信中的一方拥有它。如果Angela希望给Gus发送一个秘密消息，她会询问Gus的公开密钥。Gus通过未加密的连接将自己的公开密钥发送给她。Angela使用Gus的公开密钥来加密她的消息，将加密后的消息发送给Gus。如果Edgar在Gus向Angela发送其密钥时窃听，Edgar也会得到Gus的公开密钥。不过，Edgar并不能由这个密钥解密Angela发送给Gus的消息，因为解密需要Gus的私有密钥。即使公开密钥在传输中被别人检测到，（加密的）消息也是安全的。

非对称加密也可用于身份认证和消息完整性检查。为此，Angela会在发送消息前用她的私有密钥加密消息。当Gus接收到时，他会用Angela的公开密钥解密。如果解密成功，Gus会知道消息确实来自Angela。毕竟，没有其他人能够生成可以用她的公开密钥正确解密的消息。Gus还能知道消息在传输过程中没有被篡改，不论是被Edgar恶意修改，还是由于存在bug的软件或网络噪声无意地改变了消息，因为这样的改变都会破坏解密过程，导致解密不成功。再做一些工作，Angela可以对消息两次加密，一次用她的私有密钥加密，另一次用Gus的公开密钥加密，这样就一举三得，同时保证了机密性、真实性和完整性。

在实际中，公开密钥加密是比较“CPU密集型”的操作，比秘密密钥加密慢得多。因此，Angela不是用Gus的公开密钥加密整个传输内容，而是用它来加密传统的秘密密钥，将加密后的秘密密钥发送给Gus。Gus用自己的私有密钥解密。现在Angela和Gus都知道了秘密密钥，但Edgar不知道。因此，Gus和Angela现在可以使用更快的秘密密钥加密进行秘密通信，而不用担心Edgar监听。

不过，对于这个协议，Edgar还是有一种得力的攻击手段（需要注意非常重要的一点：所攻击的是用于收发消息的协议，而不是所用的加密算法。这种攻击不需要Edgar破解Gus和Angela的加密，与密钥长度也完全无关）。在Gus将其公开密钥发送给Angela时，Edgar不只是可以读取这个公开密钥，还可以用他自己的公开密钥替换真正的公开密钥！这样当Angela认为她在用Gus的公开密钥加密消息时，实际上她使用的是Edgar的公开密钥。当她向Gus发送消息时，Edgar会拦截至这个消息，使用自己的私有密钥解密，再使用Gus的公开密钥加密，然后继续发送给Gus。这称为“中间人”攻击（man-in-the-middle attack）。如果只在非安全的通道上工作，Gus和Angela都没有简单的方法可以防御这种攻击。实际中使用的解决方案是，Gus和Angela都通过可信任的第三方认证机构来存储和验证他们的公开密钥。Gus和Angela不是相互发送公开密钥，而是从认证机构获取对方的公开密钥。这种机制还是不算很完美，Edgar能够把自己放在Gus和认证机构、Angela和认证机构以及Gus和Angela之间，但对于Edgar来说，现在要进行攻击就困难多了。

如这个例子所示，加密和认证的理论与实践，包括算法和协议，都是充满荆棘坎坷的具有挑战性的领域，不时让业余密码学家感到惊奇。设计差的加密算法或协议比好的算法或协议容易得多。算法和协议好坏之间的差别并不总是那么明显。幸运的是，你不需要成为密码专家就可以在Java网络程序中使用强加密。JSSE掩盖了如何协商算法、交换密钥、认证通信双方和加密数据的底层细节。JSSE允许你创建Socket和服务器Socket，可以透明地处理安全通信中必要的协商和加密。你要做的就是通过前面几章所熟悉的流和Socket来发送数据。Java安全Socket扩展（JSSE）分为四个包：

`javax.net.ssl`
: 定义Java安全网络通信API的抽象类。

`javax.net`
: 替代构造函数创建安全Socket的抽象Socket工厂类。

`java.security.cert`
: 处理SSL所需公开密钥证书的类。

`com.sun.net.ssl`
: Sun的JSSE参考实现中实现加密算法和协议的具体类。从理论上讲，它们不属于

JSSE标准的一部分。其他的实现者可以用自己的包代替这个包。例如，实现者可以使用原生代码加快CPU密集的密钥生成和加密的过程。

## 创建安全客户端Socket

如果不太关心底层的细节，使用加密SSL Socket与现有的安全服务器通信确实非常简单。并不是用构造函数来构造一个java.net.Socket对象，而是从javax.net.ssl.SSLSocketFactory使用其createSocket()方法得到一个Socket对象。SSLSocketFactory是一个遵循抽象工厂设计模式的抽象类。要通过调用静态SSLSocketFactory.getDefault()方法得到一个实例：

```
SocketFactory factory = SSLSocketFactory.getDefault();
Socket socket = factory.createSocket("login.ibiblio.org", 7000);
```

这会返回SSLSocketFactory的一个实例，或者如果无法找到具体子类，会抛出一个InstantiationException异常。一旦有了工厂的引用，就可以使用下面的5个重载createSocket()方法创建一个SSLSocket：

```
public abstract Socket createSocket(String host, int port)
    throws IOException, UnknownHostException
public abstract Socket createSocket(InetAddress host, int port)
    throws IOException
public abstract Socket createSocket(String host, int port,
    InetAddress interface, int localPort)
    throws IOException, UnknownHostException
public abstract Socket createSocket(InetAddress host, int port,
    InetAddress interface, int localPort)
    throws IOException, UnknownHostException
public abstract Socket createSocket(Socket proxy, String host, int port,
    boolean autoClose) throws IOException
```

前两个方法创建并返回一个连接到指定主机和端口的Socket，或者如果无法连接，则抛出一个IOException异常。第三和第四个方法创建并返回一个从指定本地网络接口和端口连接到指定主机和端口的Socket。不过，最后一个createSocket()方法有些不同。它以一个连接到代理服务器的现有Socket对象作为起点。这个方法会返回一个经由这个代理服务器到指定主机和端口的Socket。autoClose参数确定当这个Socket关闭时，底层proxy Socket是否应当关闭。如果autoClose为true，则底层Socket会关闭；如果为false则不关闭。

所有这些方法返回的Socket实际上都是javax.net.ssl.SSLSocket，这是java.net.Socket的一个子类。不过，你不需要了解这些。一旦创建了安全Socket，就可以像其他任何Socket一样使用，即通过其getInputStream()、getOutputStream()和其他方法来加以使用。例如，假设有一个接受订单的服务器在*login.ibiblio.org*的端口7000监听连接。

每个订单使用一个TCP连接以ASCII字符串的形式发送。服务器接受订单并关闭这个连接（这里我省略了真实世界系统中必要的大量细节，如服务器要发送响应码，告诉客户端订单是否被接受）。客户端发送的订单形式如下：

```
Name: John Smith
Product-ID: 67X-89
Address: 1280 Deniston Blvd, NY NY 10003
Card number: 4000-1234-5678-9017
Expires: 08/05
```

这个消息中包含了大量的信息，足以让某个监听数据包的人恶意地使用John Smith的信用卡号码。因此，在发送这个订单之前，应当先对它加密。为了不增加服务器和客户端的负担（即不增加大量复杂而且容易出错的加密代码），最简单的方法就是使用一个安全Socket。下面的代码将通过安全Socket发送订单：

```
SSLSocketFactory factory
    = (SSLSocketFactory) SSLSocketFactory.getDefault();
Socket socket = factory.createSocket("login.ibiblio.org", 7000);

Writer out = new OutputStreamWriter(socket.getOutputStream(),
    "US-ASCII");
out.write("Name: John Smith\r\n");
out.write("Product-ID: 67X-89\r\n");
out.write("Address: 1280 Deniston Blvd, NY NY 10003\r\n");
out.write("Card number: 4000-1234-5678-9017\r\n");
out.write("Expires: 08/05\r\n");
out.flush();
```

与通过非安全Socket的传输相比，这里只有try块中的前面三条语句明显不同。其余代码都只是使用Socket、OutputStream和Writer类的常规方法。

读取输入没有难度。示例10-1是一个简单的程序，它会连接一个安全HTTP服务器，发送简单的GET请求并显示响应。

示例10-1：HTTPSClient

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

public class HTTPSClient {

  public static void main(String[] args) {

    if (args.length == 0) {
      System.out.println("Usage: java HTTPSClient2 host");
      return;
    }

    int port = 443; // 默认https端口
    String host = args[0];
```

```
    SSLSocketFactory factory
        = (SSLSocketFactory) SSLSocketFactory.getDefault();
    SSLSocket socket = null;
    try {
      socket = (SSLSocket) factory.createSocket(host, port);

      // 启用所有密码组
      String[] supported = socket.getSupportedCipherSuites();
      socket.setEnabledCipherSuites(supported);

      Writer out = new OutputStreamWriter(socket.getOutputStream(), "UTF-8");
      // https在GET行中需要完全URL
      out.write("GET http://" + host + "/ HTTP/1.1\r\n");
      out.write("Host: " + host + "\r\n");
      out.write("\r\n");
      out.flush();

      // 读取响应
      BufferedReader in = new BufferedReader(
          new InputStreamReader(socket.getInputStream()));

      // 读取首部
      String s;
      while (!(s = in.readLine()).equals("")) {
        System.out.println(s);
      }
      System.out.println();

      // 读取长度
      String contentLength = in.readLine();
      int length = Integer.MAX_VALUE;
      try {
        length = Integer.parseInt(contentLength.trim(), 16);
      } catch (NumberFormatException ex) {
        // 这个服务器在响应体的第一行
        // 没有发送content-length
      }
      System.out.println(contentLength);

      int c;
      int i = 0;
      while ((c = in.read()) != -1 && i++ < length) {
        System.out.write(c);
      }

      System.out.println();
    } catch (IOException ex) {
      System.err.println(ex);
    } finally {
      try {
        if (socket != null) socket.close();
      } catch (IOException e) {}
    }
  }
}
```

下面是从这个程序连接美国邮政服务（U.S. Postal Service）Web网站时得到的前几行输出：

```
% java HTTPSClient www.usps.com
HTTP/1.1 200 OK
Server: IBM_HTTP_Server
Cache-Control: max-age=0
Expires: Sun, 31 Mar 2013 17:29:33 GMT
Content-Type: text/html
Date: Sun, 31 Mar 2013 18:00:14 GMT
Transfer-Encoding: chunked
Connection: keep-alive
Connection: Transfer-Encoding

00004000

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

---

**提示：** 上一版中测试这个程序时，最初拒绝它连接到*www.usps.com*，因为它无法验证远程服务器的身份。问题在于当时我使用的JDK版本（1.4.2_02-b3）装载的root证书已过期。升级最新的版本（1.4.2_03-b2）就可以修复这个问题。如果看到类似“No trusted certificate found”（找不到可信任的证书）的异常消息，请尝试升级到JDK的最新版本。

---

运行这个程序时，你可能会注意到：它的响应比你期望的要慢。生成和交换密钥时，都有相当可观的CPU和网络开销。即使是在速度很快的网络上，建立连接也需要花费数秒的时间。因此，不要希望所有内容都通过HTTPS提供，只有确实需要保证秘密而且不太关心延迟的内容才会通过HTTPS传输。

# 选择密码组

JSSE的不同实现支持认证和加密算法的不同组合。例如， Oracle为Java 7捆绑的实现只支持128位AES加密，而IAIK的iSaSiLk支持256位AES加密。

---

**提示：** JDK捆绑的JSSE实际上确实包含实现更强256位加密的代码，不过除非你安装了JCE非受限密码策略文件（Unlimited Strength Jurisdiction Policy Files），否则这是禁用的。要想使用这些更强的加密，会相当麻烦，我甚至不打算解释。

---

SSLSocketFactory中的getSupportedCipherSuites()方法可以指出给定Socket上可用的算法组合：

```
public abstract String[] getSupportedCipherSuites()
```

不过，并非所有能够理解的密码组都一定能在连接上使用。有些强度太弱，因而禁止使用。SSLSocketFactory的getEnabledCipherSuites()方法可以指出这个Socket允许使用哪些密码组：

```
public abstract String[] getEnabledCipherSuites()
```

实际使用的密码组要在连接时由客户端和服务器协商。可能客户端和服务器不同意任何一种密码组，也可能尽管客户端和服务器都能使用一个密码组，但其中一方或双方并没有使用这个密码组所需的密钥和证书。不论哪一种情况，createSocket()方法都会抛出一个SSLException异常，这是IOException的一个子类。可以通过setEnabledCipherSuites()方法修改客户端试图使用的密码组：

```
public abstract void setEnabledCipherSuites(String[] suites)
```

这个方法的参数应当是希望使用的密码组列表。列表中的每个名必须是getSupportedCipherSuites()列出的某个密码组，否则将抛出一个IllegalArgumentException异常。Oracle的JDK 1.7支持以下密码组：

- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
- TLS_ECDHE_RSA_WITH_RC4_128_SHA

- SSL_RSA_WITH_RC4_128_SHA
- TLS_ECDH_ECDSA_WITH_RC4_128_SHA
- TLS_ECDH_RSA_WITH_RC4_128_SHA
- TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_RC4_128_MD5
- TLS_EMPTY_RENEGOTIATION_INFO_SCSV
- TLS_DH_anon_WITH_AES_128_CBC_SHA256
- TLS_ECDH_anon_WITH_AES_128_CBC_SHA
- TLS_DH_anon_WITH_AES_128_CBC_SHA
- TLS_ECDH_anon_WITH_RC4_128_SHA
- SSL_DH_anon_WITH_RC4_128_MD5
- TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA
- SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
- TLS_RSA_WITH_NULL_SHA256
- TLS_ECDHE_ECDSA_WITH_NULL_SHA
- TLS_ECDHE_RSA_WITH_NULL_SHA
- SSL_RSA_WITH_NULL_SHA
- TLS_ECDH_ECDSA_WITH_NULL_SHA
- TLS_ECDH_RSA_WITH_NULL_SHA
- TLS_ECDH_anon_WITH_NULL_SHA
- SSL_RSA_WITH_NULL_MD5
- SSL_RSA_WITH_DES_CBC_SHA

- SSL_DHE_RSA_WITH_DES_CBC_SHA
- SSL_DHE_DSS_WITH_DES_CBC_SHA
- SSL_DH_anon_WITH_DES_CBC_SHA
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
- SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
- TLS_KRB5_WITH_RC4_128_SHA
- TLS_KRB5_WITH_RC4_128_MD5
- TLS_KRB5_WITH_3DES_EDE_CBC_SHA
- TLS_KRB5_WITH_3DES_EDE_CBC_MD5
- TLS_KRB5_WITH_DES_CBC_SHA
- TLS_KRB5_WITH_DES_CBC_MD5
- TLS_KRB5_EXPORT_WITH_RC4_40_SHA
- TLS_KRB5_EXPORT_WITH_RC4_40_MD5
- TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
- TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5

每个名中的算法分为4个部分：协议、密钥交换算法、加密算法和校验和。例如，名SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA表示安全Socket层（SSL）版本3；密钥协商的Diffie-Hellman方法；没有身份认证；40位密钥的数据加密标准算法；密码块链以及安全散列算法校验和。

默认情况下，JDK 1.7实现启用了所有加密认证密码组（这个列表中的前28个）。如果想要无认证的事务或认证但不加密的事务，必须用setEnabledCipherSuites()方法显式启用这些密码组。要避免名字中包含NULL、ANON或EXPORT的密码组，除非你希望美国国家安全局（NSA）读取你的消息。

通常认为TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256对于所有已知攻击都相当安全。如果启用了TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA256，这个密码

组则更胜一筹。一般来讲，以TLS_ECDHE开头并以SHA256或SHA384结尾的密码组是当前广泛使用的最强的加密算法。大多数其他算法都可能会受到不同严重程度的攻击。

除了密钥长度，DES/AES和基于RC4加密之间还有一点重要的区别。DES和AES是块加密（即一次加密一定数量的二进制位）。DES总是加密64位。如果不足64位，编码器必须用额外的位填充输入。AES可以加密128、192或256位的块，但是如果不是块大小的整数倍，仍然需要填充输入。对于文件传输应用程序（如安全HTTP和FTP）来说，这不是问题，基本上所有数据都能立即可用。不过，对于以用户为中心的协议，如chat和Telnet，这就很成问题。RC4是一种流加密，可以一次加密一字节，更适合于可能需要一次发送一字节的协议。

例如，假设Edgar有可以随意使用的非常强大的并行计算机，可以很快地破解任何不大于64位的加密，而Gus和Angela也了解这一点。此外，他们怀疑Edgar可能胁迫他们的某个ISP或电话公司，允许他监听通信线缆，所以他们希望避免很容易受到“中间人”攻击的匿名连接。为安全起见，Gus和Angela决定只启用最强的密码组，也就是TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256。下面的代码段可以限制他们的连接只使用这个密码组：

```
String[] strongSuites = {"TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256"};
socket.setEnabledCipherSuites(strongSuites);
```

如果连接另一端不支持这个加密协议，试图读取Socket或写入Socket时，Socket会抛出一个异常，这样可以确保不会有机密信息意外经由弱通道进行传输。

# 事件处理器

网络通信相对于大多数计算机速度而言都很慢。认证的网络通信甚至更慢。安全连接所必需的密钥生成和建立过程会轻而易举地花费数秒钟时间。因此，你可能希望异步地处理连接。JSSE使用标准Java事件模型来通知程序，告诉它们客户端和服务器之间的握手何时完成。这个模式我们已经很熟悉了。为了得到握手结束事件的通知，只需要实现HandshakeCompletedListener接口：

```
public interface HandshakeCompletedListener
    extends java.util.EventListener
```

这个接口声明了handshakeCompleted()方法：

```
public void handshakeCompleted(HandshakeCompletedEvent event)
```

这个方法接收一个HandshakeCompletedEvent作为参数：

```
public class HandshakeCompletedEvent extends java.util.EventObject
```

HandshakeCompletedEvent类提供了获取事件有关信息的方法：

```
public SSLSession getSession()
public String getCipherSuite()
public X509Certificate[] getPeerCertificateChain()
    throws SSLPeerUnverifiedException
public SSLSocket getSocket()
```

通过addHandshakeCompletedListener()和removeHandshakeCompletedListener()方法，特定的HandshakeCompletedListener对象可以注册对某个SSLSocket的握手结束事件的关注：

```
public abstract void addHandshakeCompletedListener(
    HandshakeCompletedListener listener)
public abstract void removeHandshakeCompletedListener(
    HandshakeCompletedListener listener) throws IllegalArgumentException
```

# 会话管理

SSL常用于Web服务器，而且有很好的理由。Web连接一般是暂时的，每个页面需要单独的Socket。例如，在Amazon.com的安全服务器上结帐需要加载7个单独的页面，如果要编辑地址或者选择礼品包装，加载的页面还会更多。想象一下，如果每个页面都要多花10秒钟或更多的时间来协商一个安全连接，会出现什么情况，两台主机之间为建立安全通信需要完成握手，由于这个握手过程有很大的开销，SSL允许建立扩展到多个Socket的会话（session）。相同会话中的不同Socket使用一组相同的公开密钥和私有密钥。如果与Amazon.com的安全连接需要7个Socket，那么所有7个Socket都建立在同一个会话中，使用相同的密钥。只有会话中的第一个Socket需要承受生成和交换密钥带来的开销。

作为使用JSSE的程序员，利用会话时不需要任何额外的工作。如果在很短的一段时间内对一台主机的一个端口打开多个安全Socket，JSSE会自动重用这个会话的密钥。不过，在高安全性应用程序中，你可能希望禁止Socket之间的会话共享，或强制会话重新认证。在JSSE中，会话由SSLSession接口的实例表示，可以使用这个接口的方法来检查会话的创建时间和最后访问时间、将会话作废、得到会话的各种有关信息等：

```
public byte[] getId()
public SSLSessionContext getSessionContext()
public long getCreationTime()
public long getLastAccessedTime()
public void invalidate()
public void putValue(String name, Object value)
public Object getValue(String name)
```

```
public void removeValue(String name)
public String[] getValueNames()
public X509Certificate[] getPeerCertificateChain()
throws SSLPeerUnverifiedException
public String getCipherSuite()
public String getPeerHost()
```

SSLSocket类的getSession()方法返回这个Socket所属的Session（会话）：

```
public abstract SSLSession getSession()
```

不过，会话是性能和安全的一个折中。每一个事务都重新协商密钥会更加安全。如果你确实拥有能力超强的硬件，要保护你的系统免受同样坚定、富有、斗志昂扬而且精明能干的对手的攻击，你可能希望避免会话。为避免Socket创建会话，要向setEnableSessionCreation()传入false：

```
public abstract void setEnableSessionCreation(boolean allowSessions)
```

允许有多Socket的会话时，getEnableSessionCreation()会返回true，如果不允许多Socket的会话，这个方法返回false：

```
public abstract boolean getEnableSessionCreation()
```

在很少见的情况下，你可能甚至希望重新认证一个连接（也就是说，丢弃前面协商好的所有证书和密钥，重新开始一个新的会话）。startHandshake()方法可以做到这一点：

```
public abstract void startHandshake() throws IOException
```

# 客户端模式

大多数安全通信中，作为一条经验，服务器需要使用适当的证书认证自己。不过，客户端并非如此。也就是说，当我使用Amazon的安全服务器购买一本书时，它必须向我的浏览器证明它确实是Amazon，而不是某某黑客。不过，我不需要向Amazon证明我是Elliotte Rusty Harold。从很大程度上讲，就应该如此，这是因为，购买和安装认证所需的可信任证书是一个很繁琐的过程，用户都很讨厌这种经历，如果读者只是想购买最新的Nutshell手册，应该不用完成这个过程。不过，这种不对称可能导致信用卡欺骗。为避免类似这样的问题，可以要求Socket自行认证。这种策略不适用于向一般公众开放的服务。不过，在某些高安全性的内部应用程序中这是合理的。

setUseClientMode()方法确定Socket是否需要在第一次握手时使用认证。这个方法名有些误导性。客户端和服务器端Socket都可以使用这个方法。不过，传入true时，它表示Socket处于客户端模式（无论是否在客户端上），因此不会提供自行认证。传入false时，则会尝试自行认证：

```
public abstract void setUseClientMode(boolean mode)
    throws IllegalArgumentException
```

这个属性对于任何指定Socket只能设置一次。试图第二次设置时会抛出一个IllegalArgumentException异常。

getUseClientMode()方法只是告诉这个Socket是否会在第一次握手时使用认证：

```
public abstract boolean getUseClientMode()
```

服务器端的安全Socket（即由SSLServerSocket的accept()方法返回的Socket）可以使用setNeedClientAuth()方法，要求与它连接的所有客户端都要自行认证（或不认证）：

```
public abstract void setNeedClientAuth(boolean needsAuthentication)
    throws IllegalArgumentException
```

如果Socket不在服务器端，这个方法会抛出一个IllegalArgumentException异常。

当Socket需要客户端认证时，getNeedClientAuth()方法返回true，否则返回false：

```
public abstract boolean getNeedClientAuth()
```

# 创建安全服务器Socket

安全客户端Socket只是问题的一半。另一半是启用SSL的服务器Socket。它们是javax.net.SSLServerSocket类的实例：

```
public abstract class SSLServerSocket extends ServerSocket
```

与SSLSocket相似，这个类的所有构造函数都是保护类型的，而且SSLServerSocket实例由抽象工厂类javax.net.SSLServerSocketFactory创建：

```
public abstract class SSLServerSocketFactory
    extends ServerSocketFactory
```

类似于SSLSocketFactory，SSLServerSocketFactory的实例由SSLServerSocketFactory.getDefault()静态方法返回：

```
public static ServerSocketFactory getDefault()
```

另外一点也与SSLSocketFactory相似，SSLServerSocketFactory有3个重载的createServerSocket()方法，可以返回SSLServerSocket的实例，从java.net.ServerSocket构造函数可以很容易地类推来理解这3个重载方法：

```
public abstract ServerSocket createServerSocket(int port)
```

```
        throws IOException
    public abstract ServerSocket createServerSocket(int port,
        int queueLength) throws IOException
    public abstract ServerSocket createServerSocket(int port,
        int queueLength, InetAddress interface) throws IOException
```

如果这就是创建安全服务器Socket的全部，那么使用会非常简单明了。遗憾的是，并不只是这些。SSLServerSocketFactory.getDefault()返回的工厂一般只支持服务器认证。它不支持加密。为了同时进行加密，服务器端安全socket需要更多的初始化和设置。具体如何设置与实现有关。在Sun的参考实现中，要由一个com.sun.net.ssl.SSLContext对象负责创建已经充分配置和初始化的安全服务器Socket。具体细节随着JSSE实现的不同而有所差别，但是要在参考实现中创建一个安全服务器Socket，必须完成以下步骤：

1. 使用keytool生成公开密钥和证书。
2. 花钱请可信任的第三方（如Comodo）认证你的证书。
3. 为你使用的算法创建一个SSLContext。
4. 为你要使用的证书源创建一个TrustManagerFactory。
5. 为你要使用的密钥类型创建一个KeyManagerFactory。
6. 为密钥和证书数据库创建一个KeyStore对象（Oracle的默认值是JKS）。
7. 用密钥和证书填充KeyStore对象。例如，使用加密所用的口令短语从文件系统加载。
8. 用KeyStore及其口令短语初始化KeyManagerFactory。
9. 用KeyManagerFactory中的密钥管理器（必要）、TrustManagerFactory中的信任管理器和一个随机源来初始化上下文（如果愿意接受默认值，后两个可以为null）。

示例10-2通过一个完整的SecureOrderTaker展示了这个过程，这个程序会接受订单并显示在System.out。当然，在实际的应用程序中，还要对订单做一些更有意思的处理。

示例10-2：SecureOrderTaker

```
import java.io.*;
import java.net.*;
import java.security.*;
import java.security.cert.CertificateException;
import java.util.Arrays;

import javax.net.ssl.*;

public class SecureOrderTaker {

  public final static int PORT = 7000;
  public final static String algorithm = "SSL";
```

```
public static void main(String[] args) {
  try {
    SSLContext context = SSLContext.getInstance(algorithm);

  // 参考实现只支持X.509密钥
    KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");

    // Oracle的默认密钥库类型
    KeyStore ks = KeyStore.getInstance("JKS");

    // 出于安全考虑，每个密钥库都必须用口令短语加密，
    // 在从磁盘加载前必须提供这个口令。口令短语以char[]
    // 数组形式存储，所以可以很快地从内存中擦除，而不是
    // 等待垃圾回收
    char[] password = System.console().readPassword();
    ks.load(new FileInputStream("jnp4e.keys"), password);
    kmf.init(ks, password);
    context.init(kmf.getKeyManagers(), null, null);

    //擦除口令
    Arrays.fill(password, '0');

    SSLServerSocketFactory factory
        = context.getServerSocketFactory();

    SSLServerSocket server
        = (SSLServerSocket) factory.createServerSocket(PORT);

    // 增加匿名（未认证）密码组
    String[] supported = server.getSupportedCipherSuites();
    String[] anonCipherSuitesSupported = new String[supported.length];
    int numAnonCipherSuitesSupported = 0;
    for (int i = 0; i < supported.length; i++) {
      if (supported[i].indexOf("_anon_") > 0) {
        anonCipherSuitesSupported[numAnonCipherSuitesSupported++] =
                                                    supported[i];
      }
    }

    String[] oldEnabled = server.getEnabledCipherSuites();
    String[] newEnabled = new String[oldEnabled.length
        + numAnonCipherSuitesSupported];
    System.arraycopy(oldEnabled, 0, newEnabled, 0, oldEnabled.length);
    System.arraycopy(anonCipherSuitesSupported, 0, newEnabled,
        oldEnabled.length, numAnonCipherSuitesSupported);

    server.setEnabledCipherSuites(newEnabled);

    // 现在所有设置工作已经完成，
    // 可以集中进行实际通信了
    while (true) {
      // 这个socket是安全的，
      // 但是从代码中看不出任何迹象
      try (Socket theConnection = server.accept()) {
        InputStream in = theConnection.getInputStream();
```

```
            int c;
            while ((c = in.read()) != -1) {
              System.out.write(c);
            }
          } catch (IOException ex) {
            ex.printStackTrace();
          }
        }
      } catch (IOException | KeyManagementException
          | KeyStoreException | NoSuchAlgorithmException
          | CertificateException | UnrecoverableKeyException ex) {
        ex.printStackTrace();
      }
    }
}
```

当前工作目录中有一个用口令“2andnotafnord”保护的名为*jnp4e.keys*的文件，这个示例程序从这个文件中加载必要的密钥和证书。这个示例中没有展示如何创建这个文件。该文件是用JDK提供的keytool程序构建的，如下：

```
$ keytool -genkey -alias ourstore -keystore jnp4e.keys
Enter keystore password:
Re-enter new password:
What is your first and last name?
    [Unknown]: Elliotte Harold
What is the name of your organizational unit?
    [Unknown]: Me, Myself, and I
What is the name of your organization?
    [Unknown]: Cafe au Lait
What is the name of your City or Locality?
    [Unknown]: Brooklyn
What is the name of your State or Province?
    [Unknown]: New York
What is the two-letter country code for this unit?
    [Unknown]: NY
Is <CN=Elliotte Harold, OU="Me, Myself, and I", O=Cafe au Lait, L=Brooklyn,
ST=New York, C=NY> correct?
    [no]: y

Enter key password for <ourstore>
        (RETURN if same as keystore password):
```

完成时将得到一个名为*jnp4e.keys*的文件，其中包含你的公开密钥。不过，没有人会相信这是你的公开密钥，除非由可信任的第三方（如GeoTrust或 GoDaddy）对其进行证明。在决定是否要经受购买认证证书的麻烦、承受相应的高额花销和责任之前，如果只是想研究JSSE，为此，Oracle提供了一个已验证的密钥库文件，名为*testkeys*，用口令“passphrase”保护，里面有一些JSSE例子。但是它们还不足在实际工作中使用。

另一种方法是使用不需要认证的密码组。JDK中有这样一些密码组，包括：

- SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
- SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
- SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
- SSL_DH_anon_WITH_DES_CBC_SHA
- SSL_DH_anon_WITH_RC4_128_MD5
- TLS_DH_anon_WITH_AES_128_CBC_SHA
- TLS_DH_anon_WITH_AES_128_CBC_SHA256
- TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_anon_WITH_AES_128_CBC_SHA
- TLS_ECDH_anon_WITH_NULL_SHA
- TLS_ECDH_anon_WITH_RC4_128_SHA

默认情况下没有启用这些密码组，因为它们很容易受到“中间人”攻击，但至少这些密码组允许你编写简单的程序而不用付钱。

# 配置SSLServerSocket

一旦成功地创建并初始化了一个SSLServerSocket，只使用从java.net.ServerSocket继承的方法就可以编写很多应用程序。不过，有时需要略微调整一下它的行为。与SSLSocket相似，SSLServerSocket提供了选择密码组、管理会话和确立客户端是否需要自行认证的方法。大多数方法都与SSLSocket中的同名方法非常相似。区别在于它们工作于服务器端，将设置由SSLServerSocket接受的Socket的默认值。有些情况下，一旦接受一个SSLSocket，还可以使用SSLSocket的方法来配置这一个Socket，而不是由这个SSLServerSocket接受的所有Socket。

## 选择密码组

与SSLSocket相同，SSLServerSocket类也有3个方法可以确定支持和启用了哪些密码组：

```
public abstract String[] getSupportedCipherSuites()
public abstract String[] getEnabledCipherSuites()
public abstract void     setEnabledCipherSuites(String[] suites)
```

这些方法与SSLSocket中的同名方法一样，也使用相同的密码组名。区别在于这些方法应用于SSLServerSocket接受的所有Socket，而不只是一个Socket。例如，以下代码段的效果是在SSLServerSocket server上启用匿名的非认证连接。这要依赖于这些密码组名中包

含字符串*anon*。对于Oracle的参考实现这是正确的，但不保证其他实现也会遵循这个约定：

```
String[] supported = server.getSupportedCipherSuites();
String[] anonCipherSuitesSupported = new String[supported.length];
int numAnonCipherSuitesSupported = 0;
for (int i = 0; i < supported.length; i++) {
  if (supported[i].indexOf("_anon_") > 0) {
    anonCipherSuitesSupported[numAnonCipherSuitesSupported++]
        = supported[i];
  }
}

String[] oldEnabled = server.getEnabledCipherSuites();
String[] newEnabled = new String[oldEnabled.length
    + numAnonCipherSuitesSupported];
System.arraycopy(oldEnabled, 0, newEnabled, 0, oldEnabled.length);
System.arraycopy(anonCipherSuitesSupported, 0, newEnabled,
    oldEnabled.length, numAnonCipherSuitesSupported);

server.setEnabledCipherSuites(newEnabled);
```

这个代码段使用getSupportedCipherSuites()和getEnabledCipherSuites()获取所支持和启用的密码组。它查看所支持的每一个密码组的名字，看是否包含子串“anon”。如果密码组名中确实包含这个子串，则该密码组会添加到一个匿名密码组列表中。一旦构建了这个匿名密码组列表，它与之前的已启用密码组列表组合为一个新数组。然后将这个新数组传递给setEnabledCipherSuites()，这样一来，以前启用的密码组和匿名密码组现在都可以使用了。

## 会话管理

客户端和服务器必须都同意建立一个会话。服务器端使用setEnableSessionCreation()方法指定是否允许建立会话，另外使用getEnableSessionCreation()方法确定当前是否允许建立会话：

```
public abstract void setEnableSessionCreation(boolean allowSessions)
public abstract boolean getEnableSessionCreation()
```

默认情况下是允许创建会话的。如果服务器禁止创建会话，需要会话的客户端仍能够连接。只不过它不会得到一个会话，而必须为每一个Socket再次完成握手。类似地，如果客户端拒绝会话而服务器允许，它们仍能在没有会话的情况下相互对话。

## 客户端模式

SSLServerSocket类有两个方法可以确定和指定是否要求客户端向服务器认证自己。通

过向setNeedClientAuth()方法传递true，可以指定只有客户端能够认证自己的连接才会被接受。如果传入false，则指定不需要客户端进行认证。默认值为false。如果出于某种原因需要了解这个属性的当前状态， getNeedClientAuth()方法可以告诉你：

```
public abstract void setNeedClientAuth(boolean flag)
public abstract boolean getNeedClientAuth()
```

setUseClientMode()方法允许程序指定：即使创建了一个SSLServerSocket，也需要并应当在通信的认证和其他协商方面将其视为一个客户端。例如，在一个FTP会话中，客户端程序打开一个服务器Socket接收来自服务器的数据，但这并不妨碍它仍是一个客户端。如果SSLServerSocket处于客户端模式，getUseClientMode()方法会返回true，否则返回false：

```
public abstract void setUseClientMode(boolean flag)
public abstract boolean getUseClientMode()
```

# 第11章

# 非阻塞I/O

与CPU和内存相比，甚至与磁盘相比，网络都很慢。现代高端PC能够在CPU和主存之间以大约每秒6G字节的速度传送数据。与磁盘传送数据速度要慢得多，但仍可以达到大约每秒150M字节[注1]。与之不同，当今最快的局域网的理论最大速度是每秒150M字节，但大多数LAN只支持这个速度的十分之一到百分之一。通过公共Internet传输的速度一般至少要比LAN的速度低一个数量级。我的FIOS连接相对较快，承诺速度不大于每秒6M字节，另外不小于每秒3M字节，但我的LAN只能支持这个速度的5%。CPU、磁盘和网络都会随着时间而加速。这些数字比我在10年前写本书第三版时所说的要高得多。但是在可预见的将来，CPU和磁盘或许仍会比网络速度快几个数量级。这种情况下，你肯定不希望让飞快的CPU等待（相对）缓慢的网络。

要允许CPU速度高于网络，传统的Java解决方案是缓冲和多线程。多个线程可以同时为几个不同的连接生成数据，并将数据存储在缓冲区中，直到网络确实准备好发送这些数据；对于相当简单的服务器和客户端，如果不需要非常高的性能，这种方法效果很好。不过，生成多个线程以及在线程之间切换的开销是不容忽视的。例如，每个线程需要大约1M的RAM。在一个可能一秒钟处理上千个请求的大服务器上，你可能不会为每个连接都分配一个线程。如果一个线程可以负责多个连接，可以选取一个准备好接收数据的连接，尽快填充这个连接所能管理的尽可能多的数据，然后转向下一个准备好的连接，这样速度就会更快。

注1：这是大致的理论最大值。不过，需要指出，我使用M字节表示1024×1024字节，G字节表示1024M字节。制造商通常将G字节的大小取整为1000M字节，而M字节为1000000字节，使得他们的数字听起来更有影响力。此外，网络速度通常是指每秒的K/M/G位（bit）数，而不是每秒的字节（byte）数。这里所有数字都以字节为单位，以便与硬盘、内存和网络带宽比较。

要想正常工作，这种方法需要得到底层操作系统的支持。幸运的是，可能作为大吞吐量服务器的所有现代操作系统几乎都支持这种非阻塞I/O。不过，在某些很受关注的客户端系统上（如平板电脑、蜂窝电话和类似的系统），对非阻塞I/O的支持可能并不充分。事实上，提供这个支持的java.nio包并不是当前或规划中Java ME规范的一部分（尽管Android提供了支持）。不过，整个新的I/O API是为服务器设计的，而且只与服务器有关，正是因为这个原因，在开始讨论服务器之前我并没有太多地提到它。客户端甚至对等系统很少需要处理这么多并发连接，基于流的多线程I/O也不会成为显著的瓶颈。

## NIO行动太慢，步伐太小？

曾经有一段时间，合理建构的非阻塞I/O远远胜过多线程、多进程的设计。那是在20世纪90年代。遗憾的是，在2002年推出Java 1.4之前，Java一直都未引入非阻塞I/O。直到2004年发布Java 5（当然还有2006年发布Java 6），由于对操作系统固有线程机制的不断改进，已经基本上消除了所有上下文切换和无竞争同步开销。此外，服务器内存已经显著扩大，即使采用廉价的硬件也可以轻松地在内存中同时维护10 000个线程，需要多个线程以实现最大利用的多核/多CPU系统变得很常见。在当前的Java 7和Java 8 64位虚拟机上尤其如此。2013年，一方面可以看到基于NIO的体系结构所增加的复杂性，另一方面则是简单得多的“每个请求一个线程”或“每个连接一个线程”的设计，确实很难对二者的优劣做出决断。

不过NIO是不是更快一些呢？不一定。在Linux上使用Java 6完成的实际测试中，多线程经典I/O设计胜出NIO 30%左右。

有没有一些情况下异步I/O确实强于经典I/O呢？可能有。我能想象到的一种情况是，服务器需要同时支持超大量的长期连接，比如说10 000个连接以上，不过各个客户端并不会很频繁地发送太多的数据。例如，假设总公司的一个中心服务器要收集全国连锁便利店各个收银机的交易信息。这种情况就很适合使用NIO，而且只用少量线程采用异步或非阻塞的按需处理来实现会高效得多。

不过一定要记住下面这两条关于优化的黄金定律：

1. 不要贸然优化。
2. （仅适用于专家）即使要优化，也必须先做出明确而清晰的测量，证明确实有问题之后才能优化，而且能清楚地指出你所做的修改是否已经解决了这个问题。

# 一个示例客户端

虽然新的I/O API并非专门为客户端而设计，但的确可以用于客户端。我将以一个使用新I/O API的客户端程序开始介绍，因为它比较简单。具体地，很多客户端都可以采用一次一个连接的方式来实现，所以在介绍选择器（selector）和非阻塞I/O之前，我会先介绍通道（channel）和缓冲区。

为了介绍基础知识，我将为在RFC 864中定义的字符生成器协议实现一个客户端。这个协议是为测试客户端而设计的。服务器在端口19监听连接。当客户端连接时，服务器将发送连续的字符序列，直到客户端断开连接为止。客户端的所有输入都被忽略。RFC没有指定发送哪些字符序列，但建议服务器使用一种可识别的模式。一种常见的模式是以回车/换行作为行分隔符的72字符循环文本行（其中包含95个可打印ASCII字符），如下：

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefgh
"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghi
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
```

这一章之所以选取这个协议作为例子，是因为传输数据的协议和生成数据的算法都相当简单，不会因此而掩盖我们要讨论的I/O。另外，chargen可以通过相对较少的连接传输大量数据，并很快使网络达到饱和。因此，这是使用新I/O API的一个很好的选择。

---

**警告：** 如今chargen已经不再常用，而且即使打开了这个协议也可能会被本地防火墙所阻塞。它很容易受到一种“往复式”拒绝服务攻击，受到这种攻击时，欺骗性的Internet数据包会让两个主机相互之间发送无限量的数据。不仅如此，由于这几乎都是很不对称的（服务器要发送超大量的数据来响应极小的客户请求），即使只有几十台受攻击的主机（有些像僵尸网络），就能让一个chargen服务器的本地带宽饱和。

---

在实现利用这个新I/O API的客户端时，首先要调用静态工厂方法SocketChannel.open()来创建一个新的java.nio.channels.SocketChannel对象。这个方法的参数是一个java.net.SocketAddress对象，指示要连接的主机和端口。例如，下面的代码段连接指向*rama.poly.edu*端口19的通道：

```
SocketAddress rama = new InetSocketAddress("rama.poly.edu", 19);
SocketChannel client = SocketChannel.open(rama);
```

通道以阻塞模式打开，所以下一行代码在真正建立连接之前不会执行。如果连接无法建立，则会抛出一个IOException异常。

如果这是传统的客户端，你可能会获取该Socket的输入和（或）输出流。但这不是传统的客户端。利用通道，你可以直接写入通道本身。不是写入字节数组，而是要写入ByteBuffer对象。你已经很清楚，文本行有74个ASCII字符长（72个可打印字符，后面是回车/换行对），所以要使用静态方法allocate()创建一个容量为74字节的ByteBuffer：

```
ByteBuffer buffer = ByteBuffer.allocate(74);
```

将这个ByteBuffer对象传递给通道的read()方法。通道会用从Socket读取的数据填充这个缓冲区。它返回成功读取并存储在缓冲区的字节数：

```
int bytesRead = client.read(buffer);
```

默认情况下，这会至少读取一个字节，或者返回-1指示数据结束，这与InputStream完全一样。如果有更多字节可以读取，它通常会读取更多字节。稍后你将看到，如果将这个客户端置于非阻塞模式，没有字节可用时会立即返回0，但这里的代码会像InputStream一样阻塞。你可能猜到了，如果读取时发生错误，这个方法也会抛出一个IOException异常。

假定缓冲区中有一些数据（即n>0），这些数据就被复制到System.out。有几个方法可以从ByteBuffer中提取一个字节数组，然后再写入传统的OutputStream（如System.out）。不过，坚持采用一种完全基于通道的解决方案会更有好处。这样的解决方案需要利用Channels工具类（确切地讲是该工具类的newChannel()方法），将OutputStream System.out封装在一个通道中：

```
WritableByteChannel output = Channels.newChannel(System.out);
```

然后可以将读取的数据写入与System.out连接的这个输出通道中。不过，在这样做之前，必须回绕（flip）缓冲区，使得输出通道会从所读取数据的开头而不是末尾开始写入：

```
buffer.flip();
output.write(buffer);
```

你不必告诉输出通道要写入多少字节。缓冲区会记住其中包含多少字节。不过，一般情况下，输出通道不能保证会写入缓冲区中的所有字节。不过，在这个特定的例子中，它是阻塞通道，要么写入全部字节，要么抛出一个IOException异常。

不要每次读/写都创建一个新的缓冲区。那样做会降低性能。相反，要重用现有的缓冲区。在再次读取之前要清空缓冲区：

```
buffer.clear();
```

这与回绕有些不同。回绕可以保持缓冲区中的数据不变，只是准备写入而不是读取。清空将把缓冲区重置回初始状态（这实际上有点过于简化。老数据仍然存在，还没有被覆盖，但很快就会被从源读取的新数据覆盖）。

示例11-1将以上内容集中在一个完整的客户端中。因为按照设计chargen是一个无限循环的协议，需要用Ctrl-C结束程序。

示例11-1：一个基于通道的chargen客户端

```
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.io.IOException;

public class ChargenClient {

  public static int DEFAULT_PORT = 19;

  public static void main(String[] args) {

    if (args.length == 0) {
      System.out.println("Usage: java ChargenClient host [port]");
      return;
    }

    int port;
    try {
      port = Integer.parseInt(args[1]);
    } catch (RuntimeException ex) {
      port = DEFAULT_PORT;
    }

    try {
      SocketAddress address = new InetSocketAddress(args[0], port);
      SocketChannel client = SocketChannel.open(address);

      ByteBuffer buffer = ByteBuffer.allocate(74);
      WritableByteChannel out = Channels.newChannel(System.out);

      while (client.read(buffer) != -1) {
        buffer.flip();
        out.write(buffer);
        buffer.clear();
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

下面是示例运行的输出：

```
$ java ChargenClient rama.poly.edu
```

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefg
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefgh
"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghi
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
...
```

到目前为止，这个程序还很简单，完全可以很容易地用流来编写。只有当你希望客户端有更多的功能时，即除了将所有输入复制到输出之外还要做其他一些工作，才会真正体现出新特性。你可以在阻塞或非阻塞模式下运行这个连接，在非阻塞模式下，即使没有任何可用的数据，`read()`也会立即返回。这就允许程序在试图读取前做其他操作。它不必等待慢速的网络连接。要改变阻塞模式，可以向`configureBlocking()`方法传入`true`（阻塞）或`false`（不阻塞）。下面置连接为非阻塞模式：

```
client.configureBlocking(false);
```

在非阻塞模式下，`read()`可能因为读不到任何数据而返回0。因此循环需要有些差别：

```
while (true) {
  // 把每次循环都要运行的代码都放在这里，
  // 无论有没有读到数据
  int n = client.read(buffer);
  if (n > 0) {
    buffer.flip();
    out.write(buffer);
    buffer.clear();
  } else if (n == -1) {
    // 这不应当发生，除非服务器发生故障
    break;
  }
}
```

对于这样一个单连接客户来说，这么做的意义不是很大。可能你会查看用户是否做了某些操作，例如取消输入。不过，在下一节会看到，当程序处理多个连接时，这种做法会使代码在快速连接上运行得很快，而在慢速连接上运行慢一些。每个连接都以自己的速度运行，不会像在单行道上那样被最慢的驾驶员挡在后面。

# 一个示例服务器

客户端使用通道和缓冲区是可以的，不过实际上通道和缓冲区主要用于需要高效处理很多并发连接的服务器系统。要处理服务器，除了用于客户端的缓冲区和通道外，还需要第三个新的部分。具体来讲，需要有一些选择器，允许服务器查找所有准备好接收输出或发送输入的连接。

为了介绍基础知识，我将为字符生成器协议实现一个简单的服务器。在实现利用新I/O API的服务器时，首先要调用静态工厂方法ServerSocketChannel.open()创建一个新的ServerSocketChannel对象。

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

开始时，这个通道并没有具体监听任何端口。要把它绑定到一个端口，可以用socket()方法获取其ServerSocket对等端（peer）对象，然后使用bind()方法绑定到这个对等端。例如，下面的代码段将通道绑定到端口19的服务器Socket：

```
ServerSocket ss = serverChannel.socket();
ss.bind(new InetSocketAddress(19));
```

在Java 7及以后版本中，可以直接绑定而不用获取底层java.net.ServerSocket：

```
serverChannel.bind(new InetSocketAddress(19));
```

与正常的服务器Socket一样，要绑定到端口19，在UNIX（包括Linux和Mac OS X）上必须是root用户。非root用户只能绑定1024及以上的端口。

服务器Socket通道现在在端口19监听入站连接。要接受连接，可以调用accept()方法，它会返回一个SocketChannel对象：

```
SocketChannel clientChannel = serverChannel.accept();
```

在服务器端，你肯定希望客户端通道处于非阻塞模式，以允许服务器处理多个并发连接：

```
clientChannel.configureBlocking(false);
```

你可能还希望ServerSocketChannel也处于非阻塞模式。默认情况下，这个accept()方法会阻塞，直到有一个入站连接为止，这与ServerSocket的accept()方法类似。为了改变这一点，只需在调用accept()之前调用configureBlocking(false)：

```
serverChannel.configureBlocking(false);
```

如果没有入站连接，非阻塞的accept()几乎会立即返回null。要确保对此进行检查，否则当试图使用这个socket（而它实际为null）时，会得到一个讨厌的NullPointerException异常。

现在有两个打开的通道：服务器通道和客户端通道。两个通道都需要处理。它们都会无限运行下去。此外，处理服务器通道会创建更多打开的客户端通道。在传统方法中，要为每个连接分配一个线程，线程数目会随着客户端连接迅速攀升。相反，在新的I/O

API中，可以创建一个Selector，允许程序迭代处理所有准备好的连接。要构造一个新的Selector，只需调用Selector.open()静态工厂方法：

```
Selector selector = Selector.open();
```

接下来，需要使用每个通道的register()方法向监视这个通道的选择器进行注册。在注册时，要使用SelectionKey类提供的命名常量指定所关注的操作。对于服务器Socket，唯一关心的操作就是OP_ACCEPT，也就是服务器Socket通道是否准备好接受一个新连接？

```
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

对于客户端通道，你要知道的稍有些不同，确切地讲，你可能希望知道是否已经准备好数据可以写入通道。为此，要使用OP_WRITE键：

```
SelectionKey key = clientChannel.register(selector, SelectionKey.OP_WRITE);
```

两个register()方法都返回一个SelectionKey对象。不过，只需要使用对应客户端通道的键，因为可能会有多个这样的键。每个SelectionKey都有一个任意Object类型的“附件”。它通常用于保存一个指示当前连接状态的对象。在这里，可以将通道要写入网络的缓冲区存储在这个对象中。一旦缓冲区完全排空（drain），将重新填满。要用将复制到各缓冲区的数据来填充数组。并不是写到缓冲区末尾，而是要回转到缓冲区开始位置重新写入。先从两段顺序的数据开始会容易一些，这样每一行可以作为数组中的一个连续序列：

```
byte[] rotation = new byte[95*2];
for (byte i = ' '; i <= '~'; i++) {
  rotation[i - ' '] = i;
  rotation[i + 95 - ' '] = i;
}
```

因为此数据在初始化之后只用于读取，所以可以重用于多个通道。不过，每个通道都会用这个数组的内容填充其自己的缓冲区。我们会用这个循环数组的前72字节填充缓冲区，然后加上回车/换行对来分隔各行。接下来要回绕缓冲区，从而可以进行排空，并附加到通道的键上：

```
ByteBuffer buffer = ByteBuffer.allocate(74);
buffer.put(rotation, 0, 72);
buffer.put((byte) '\r');
buffer.put((byte) '\n');
buffer.flip();
key2.attach(buffer);
```

为了检查是否有可操作的数据，可以调用选择器的select()方法。对于长时间运行的服务器，这一般要放在一个无限循环中：

```
while (true) {
  selector.select ();
  // 处理选择的键...
}
```

假定选择器确实找到了一个就绪的通道，其selectedKeys()方法会返回一个java.util.Set，其中对应各个就绪通道分别包含一个SelectionKey对象。否则它会返回一个空集。在两种情况下，都可以通过一个java.util.Iterator循环处理：

```
Set<SelectionKey> readyKeys = selector.selectedKeys();
Iterator iterator = readyKeys.iterator();
while (iterator.hasNext()) {
  SelectionKey key = iterator.next();
  // 从集合中删除这个键，从而不会处理两次
  iterator.remove();
  // 处理通道...
}
```

通过从集合中删除键，这就告诉选择器这个键已经处理过，这样Selector就不需要在每次调用select()时再将这个键返回给我们了。再次调用select()时，如果这个通道再次就绪，Selector就会把该通道再增加到就绪集合中。不过，在这里删除就绪集合中的键的确很重要。

如果就绪的通道是服务器通道，程序就会接受一个新Socket通道，将其添加到选择器。如果就绪的通道是Socket通道，程序就会向通道写入缓冲区中尽可能多的数据。如果没有通道就绪，选择器就会等待。一个线程（主线程）可以同时处理多个连接。

在这里可以很容易地判断所选择的通道是客户端通道还是服务器通道，因为服务器通道只准备接受，而客户端通道只准备写入。二者都是I/O操作，由于多种原因，它们都可能会抛出IOException异常，所以需要将它们都包围在try块中：

```
try {
  if (key.isAcceptable()) {
    ServerSocketChannel server = (ServerSocketChannel) key.channel();
    SocketChannel connection = server.accept();
    connection.configureBlocking(false);
    connection.register(selector, SelectionKey.OP_WRITE);
    //为客户端建立缓冲区...
  } else if (key.isWritable()) {
    SocketChannel client = (SocketChannel) key.channel();
    // 向客户端写入数据...
  }
}
```

向通道写入数据很简单。首先获取键的附件，将它转换为ByteBuffer，调用hasRemaining()检查缓冲区中是否还剩余未写的数据。如果有，就写入到通道。否则，用rotation数组中的下一行数据重新填充缓冲区，并写入到通道。

```
ByteBuffer buffer = (ByteBuffer) key.attachment();
if (!buffer.hasRemaining()) {
  // 用下一行数据重新填充缓冲区，
  // 确定最后一行从哪里开始
  buffer.rewind();
  int first = buffer.get();
  // 递增到下一个字符
  buffer.rewind();
  int position = first - ' ' + 1;
  buffer.put(rotation, position, 72);
  buffer.put((byte) '\r');
  buffer.put((byte) '\n');
  buffer.flip();
}
client.write(buffer);
```

要确定从哪里获取下一行数据，这个算法依赖于以ASCII字符顺序存储在rotation数组中的字符。buffer.get()从缓冲区中读取第一个数据字节。这个数字要减去空格字符（32），因为空格是rotation数组中的第一个字符。由此可以知道缓冲区当前从数组的哪个索引开始。要加1来得到下一行的开始索引，并重新填充缓冲区。

在chargen协议中，服务器永远不会关闭连接。它等待客户端中断Socket。当Socket中断时，会抛出一个异常。取消这个键，并关闭对应的通道：

```
catch (IOException ex) {
  key.cancel();
  try {
    key.channel().close();
  } catch (IOException cex) {
    // 忽略
  }
}
```

示例11-2把上述内容集中在了一个完整的chargen服务器中，它会在一个线程中高效地处理多个连接。

示例11-2：一个非阻塞的chargen服务器

```
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.util.*;
import java.io.IOException;

public class ChargenServer {
```

```
public static int DEFAULT_PORT = 19;

public static void main(String[] args) {

  int port;
  try {
    port = Integer.parseInt(args[0]);
  } catch (RuntimeException ex) {
    port = DEFAULT_PORT;
  }
  System.out.println("Listening for connections on port " + port);

  byte[] rotation = new byte[95*2];
  for (byte i = ' '; i <= '~'; i++) {
    rotation[i -' '] = i;
    rotation[i + 95 - ' '] = i;
  }

  ServerSocketChannel serverChannel;
  Selector selector;
  try {
    serverChannel = ServerSocketChannel.open();
    ServerSocket ss = serverChannel.socket();
    InetSocketAddress address = new InetSocketAddress(port);
    ss.bind(address);
    serverChannel.configureBlocking(false);
    selector = Selector.open();
    serverChannel.register(selector, SelectionKey.OP_ACCEPT);
  } catch (IOException ex) {
    ex.printStackTrace();
    return;
  }

  while (true) {
    try {
      selector.select();
    } catch (IOException ex) {
      ex.printStackTrace();
      break;
    }

    Set<SelectionKey> readyKeys = selector.selectedKeys();
    Iterator<SelectionKey> iterator = readyKeys.iterator();
    while (iterator.hasNext()) {

      SelectionKey key = iterator.next();
      iterator.remove();
      try {
        if (key.isAcceptable()) {
          ServerSocketChannel server = (ServerSocketChannel) key.channel();
          SocketChannel client = server.accept();
          System.out.println("Accepted connection from " + client);
          client.configureBlocking(false);
          SelectionKey key2 = client.register(selector, SelectionKey.
                                                          OP_WRITE);
```

```
            ByteBuffer buffer = ByteBuffer.allocate(74);
            buffer.put(rotation, 0, 72);
            buffer.put((byte) '\r');
            buffer.put((byte) '\n');
            buffer.flip();
            key2.attach(buffer);
          } else if (key.isWritable()) {
            SocketChannel client = (SocketChannel) key.channel();
            ByteBuffer buffer = (ByteBuffer) key.attachment();
            if (!buffer.hasRemaining()) {
              // 用下一行重新填充缓冲区
              buffer.rewind();
              // 得到上一次的首字符
              int first = buffer.get();
              // 准备改变缓冲区中的数据
              buffer.rewind();
              // 寻找rotation中新的首字符位置
              int position = first - ' ' + 1;
              // 将数据从rotation复制到缓冲区
              buffer.put(rotation, position, 72);
              // 在缓冲区末尾存储一个行分隔符
              buffer.put((byte) '\r');
              buffer.put((byte) '\n');
              // 准备缓冲区进行写入
              buffer.flip();
            }
            client.write(buffer);
          }
        } catch (IOException ex) {
          key.cancel();
          try {
            key.channel().close();
          }
          catch (IOException cex) {}
        }
      }
    }
  }
}
```

这个例子只使用了一个线程。还有一些情况下可能仍要使用多个线程，特别是当不同操作有不同优先级时。例如，可能要在一个高优先级线程中接受新连接，而在一个低优先级的线程中对现有的连接提供服务。不过，线程和连接之间不再需要1:1的比例，这样可以极大地提升用Java编写的服务器的可扩展性。

还有一点很重要，可以使用多线程来达到最高性能。多线程允许服务器利用多个CPU。即使只有一个CPU，将接受线程与处理线程分开通常也是一个好的想法。甚至第3章讨论的线程池与这个新的I/O模型也有关系。接受连接的线程可以将接受的连接放在队列中，由池中的线程进行处理。与不用选择器的做法相比，这仍要快得多，因为`select()`可以确保如果连接没有准备好接收数据，就绝对不会在这些连接上浪费时间。另一方

面，这里的同步问题非常棘手，所以不要尝试这种解决方案，除非经过性能测试证明确实存在着瓶颈。

## 缓冲区

在第2章，我曾建议，要始终对你的数据流进行缓冲。与提供足够大缓冲区相比，几乎没有哪种方法能够对网络程序的性能产生更大的影响。不过，在新的I/O模型中，你将无从选择。所有I/O都要缓冲。事实上缓冲区已经成为这个API的基础部分。在新的I/O模型中，不再向输出流写入数据和从输入流读取数据，而是要从缓冲区中读写数据。像在缓冲流中一样，缓冲区可能就是字节数组。不过，原始实现可以将缓冲区直接与硬件或内存连接，或者使用其他非常高效的实现。

从编程角度看，流和通道之间的关键区别在于流是基于字节的，而通道是基于块的。流设计为按顺序一个字节接一个字节地传送数据。出于性能考虑，也可以传送字节数组。不过，基本的概念都是一次传送一个字节的数据。与之不同，通道会传送缓冲区中的数据块。可以读写通道的字节之前，这些字节必须已经存储在缓冲区中，而且一次会读/写一个缓冲区的数据。

流和通道/缓冲区之间的第二个关键区别是，通道和缓冲区支持同一对象的读/写。也不总是如此。例如，指向CDROM上某个文件的通道只能读，不能写。如果一个Socket关闭了输入，连接到这个Socket的通道就只能写而不能读。如果试图写入只读通道或者读取只写通道，就会抛出UnsupportedOperationException异常。不过，更一般的情况是，网络程序可以读/写同一个通道。

不需要过多地考虑底层细节（而且细节随着实现的不同会有很大差别，这主要是因为不同的实现要适应各自的主机操作系统和硬件），可以把缓冲区看作是固定大小的元素列表，如数组，这些元素为某种特定类型，一般是基本数据类型。但是在后台不一定是数组。有时候确实是数组，有时候则不是。除了boolean外，Java的所有基本数据类型都有特定的Buffer子类：ByteBuffer、CharBuffer、ShortBuffer、IntBuffer、LongBuffer、FloatBuffer和DoubleBuffer。每个子类中的方法都有相应类型的返回值和参数列表。例如，DoubleBuffer类有设置和获取double的方法。IntBuffer类有设置和获取int的方法。公共的Buffer超类只提供一些“通用”方法，这些方法不需要知道缓冲区所包含数据是什么类型（这里缺乏简单类型的泛型确实有不好的后果）。网络程序几乎只会使用ByteBuffer，但程序偶尔也会使用其他类型来取代ByteBuffer。

除了数据列表外，每个缓冲区都记录了信息的4个关键部分。无论缓冲区是何种类型，都有相同的方法来获取和设置这些值：

*位置（position）*

缓冲区中将读取或写入的下一个位置。这个位置值从0开始计，最大值等于缓冲区的大小。可以用下面两个方法获取和设置：

```
public final int    position()
public final Buffer position(int newPosition)
```

*容量（capacity）*

缓冲区可以保存的元素的最大数目。容量值在创建缓冲区时设置，此后不能改变。可以用以下方法读取：

```
public final int capacity()
```

*限度（limit）*

缓冲区中可访问数据的末尾位置。只要不改变限度，就无法读/写超过这个位置的数据，即使缓冲区有更大的容量也没有用。限度可以用下面两个方法获取和设置：

```
public final int limit()
public final Buffer limit(int newLimit)
```

*标记（mark）*

缓冲区中客户端指定的索引。通过调用`mark()`可以将标记设置为当前位置。调用`reset()`可以将当前位置设置为所标记的位置：

```
public final Buffer mark()
public final Buffer reset()
```

如果将位置设置为低于现有标记，则丢弃这个标记。

与读取`InputStream`不同，读取缓冲区实际上不会以任何方式改变缓冲区中的数据。只可能向前或向后设置位置，从而可以从缓冲区中某个特定位置开始读取。类似地，程序可以调整限度，从而控制将要读取的数据的末尾。只有容量是固定的。

公共的`Buffer`超类还提供了另外几个方法，可以通过这些公共属性的引用来进行操作。

`clear()`方法将位置设置为0，并将限度设置为容量，从而将缓冲区“清空”。这样一来，就可以完全重新填充缓冲区了：

```
public final Buffer clear()
```

不过，`clear()`方法没有删除缓冲区中的老数据。这些数据仍然存在，还可以使用绝对get方法或者再改变限度和位置进行读取。

`rewind()`将位置设置为0，但不改变限度：

```
public final Buffer rewind()
```

这允许重新读取缓冲区。

flip()方法将限度设置为当前位置，位置设置为0：

```
public final Buffer flip()
```

希望排空刚刚填充的缓冲区时可以调用这个方法。

最后，还有两个方法可以返回缓冲区的信息，但不改变这些信息。remaining()方法返回缓冲区中当前位置与限度之间的元素数。如果剩余元素大于0，hasRemaining()方法返回true：

```
public final int     remaining()
public final boolean hasRemaining()
```

## 创建缓冲区

缓冲区类的层次是基于继承的，而不基于多态，至少在顶层是如此。一般需要知道你要处理的是IntBuffer、ByteBuffer、CharBuffer还是其他类型。要用其中一个子类编写代码，而不是一般的Buffer超类。

每种类型的缓冲区类都有几个工厂方法，以各种方式创建这个类型的特定于实现的子类。空的缓冲区一般由分配（allocate）方法创建。预填充数据的缓冲区由包装（wrap）方法创建。分配方法通常用于输入，而包装方法一般用于输出。

### 分配

基本的allocate()方法只返回一个有指定固定容量的新缓冲区，这是一个空缓冲区。例如，下面几行代码创建了字节和整型缓冲区，大小都为100：

```
ByteBuffer buffer1 = ByteBuffer.allocate(100);
IntBuffer buffer2 = IntBuffer.allocate(100);
```

游标位于缓冲区开始位置（也就是说，位置为0）。用allocate()创建的缓冲区基于Java数组实现，可以通过array()和arrayOffset()方法来访问。例如，可以使用通道将一大块数据读入缓冲区，然后从缓冲区获取这个数组，传递给其他方法：

```
byte[] data1 = buffer1.array();
int[] data2 = buffer2.array();
```

array()实际暴露了缓冲区的私有数据，所以要谨慎使用。修改后备数组会反映到缓冲区中，反之亦然。这里的一般模式是使用数据填充缓冲区，获取其后备数组，然后操作这个数组。开始处理数组之后就不要再写缓冲区，只要做到这一点，就不会问题。

### 直接分配

ByteBuffer类（但不包括其他缓冲区类）有另外一个allocateDirect()方法，这个方法不为缓冲区创建后备数组。VM会对以太网卡、核心内存或其他位置上的缓冲区使用直接内存访问，以此实现直接分配的ByteBuffer。这不是必需的，但确实是允许的，可以提升I/O操作的性能。从API角度看，allocateDirect()的使用与allocate()完全相同：

```
ByteBuffer buffer = ByteBuffer.allocateDirect(100);
```

在直接缓冲区上调用array()和arrayOffset()会抛出一个UnsupportedOperationException异常。直接缓冲区在一些虚拟机上会更快，尤其是缓冲区很大时（大约1MB或更多）。不过，创建直接缓冲区比间接缓冲区代价更高，所以只能在缓冲区可能只持续较短时间时才分配这种直接缓冲区。其细节非常依赖于VM。与大多数性能建议一样，除非经过测量后发现性能确实是个问题，否则不应考虑使用直接缓冲区。

### 包装

如果已经有了要输出的数据数组，一般要用缓冲区进行包装，而不是分配一个新缓冲区，然后一次一部分地复制到这个缓冲区。例如：

```
byte[] data = "Some data".getBytes("UTF-8");
ByteBuffer buffer1 = ByteBuffer.wrap(data);
char[] text = "Some text".toCharArray();
CharBuffer buffer2 = CharBuffer.wrap(text);
```

在这里，缓冲区包含数组的一个引用，这个数组将作为它的后备数组。由包装创建的缓冲区肯定不是直接缓冲区。再次说明，修改数组会反映到缓冲区，反之亦然，所以对数组操作结束前不要包装数组。

## 填充和排空

缓冲区是为顺序访问而设计的。应该记得，每个缓冲区都有一个当前位置，由position()方法标识，这是0到缓冲区元素个数之间的某个数（可以为0或缓冲区元素个数）。从缓冲区读取或向其写入一个元素时，缓冲区的位置将增1。例如，假设你想分配一个容量为12的CharBuffer，并在其中放置5个字符：

```
CharBuffer buffer = CharBuffer.allocate(12);
buffer.put('H');
buffer.put('e');
buffer.put('l');
buffer.put('l');
buffer.put('o');
```

缓冲区的位置现在为5。这称为填充缓冲区。

缓冲区至多只能填充到其容量大小。如果试图填充的数据超出了初始设置的容量，put()方法会抛出一个BufferOverflowException异常。

如果现在试图使用get()从缓冲区中获取数据，你会得到null字符(\u0000)，Java初始化字符缓冲区时位置5就是这个null字符。再次读取写入的数据之前，需要回绕缓冲区：

```
buffer.flip();
```

这会把缓冲区的限度设置为其位置（这个例子中为5），并将位置重新设置为0，也就是缓冲区的开始位置。现在可以将其排空到一个新的字符串：

```
String result = "";
while (buffer.hasRemaining()) {
  result += buffer.get();
}
```

每个get()调用都会将位置前移一个元素。位置达到限度时，hasRemaining()返回false。这称为排空缓冲区。

Buffer类还有一些绝对（absolute）方法，可以在缓冲区的指定位置填充和排空，而无需更新位置。例如，ByteBuffer有以下两个方法：

```
public abstract byte       get(int index)
public abstract ByteBuffer put(int index, byte b)
```

如果试图访问位于或超出缓冲区限度的一个位置，这两个方法都会抛出IndexOutOfBoundsException异常。例如，通过使用绝对方法，可以将同样这个文本放入缓冲区，如下：

```
CharBuffer buffer = CharBuffer.allocate(12);
buffer.put(0, 'H');
buffer.put(1, 'e');
buffer.put(2, 'l');
buffer.put(3, 'l');
buffer.put(4, 'o');
```

不过，读出前不再需要回绕，因为绝对方法不改变位置。另外，顺序不再有任何影响。这会得到同样的最终结果：

```
CharBuffer buffer = CharBuffer.allocate(12);
buffer.put(1, 'e');
buffer.put(4, 'o');
buffer.put(0, 'H');
buffer.put(3, 'l');
buffer.put(2, 'l');
```

## 批量方法

即使是使用缓冲区，操作数据块通常也比一次填充和排空一个元素要快。不同的缓冲区类都有一些批量方法（bulk method）来填充和排空相应元素类型的数组。

例如，ByteBuffer有put()和get()方法，可以用现有的字节数组或子数组填充和排空一个ByteBuffer：

```
public ByteBuffer get(byte[] dst, int offset, int length)
public ByteBuffer get(byte[] dst)
public ByteBuffer put(byte[] array, int offset, int length)
public ByteBuffer put(byte[] array)
```

这些put方法从当前位置开始插入指定数组或子数组的数据。get方法从当前位置将数据读取到参数指定的数组或子数组中。put和get都会使位置增加数组或子数组的长度。如果缓冲区没有足够的空间容纳这个数组或子数组，put方法会抛出一个BufferOverflowException异常。如果缓冲区没有足够的剩余数据来填充这个数组或子数组，get方法就抛出BufferUnderflowException异常。这些都是运行时异常。

## 数据转换

Java中的所有数据最终都解析为字节。所有基本数据类型——int、double、float等都可以写为字节。任何适当长度的字节序列都可解释为基本类型数据。例如，任何4字节的序列都可以对应于一个int或float（实际上两者皆可，取决于你希望如何读取）。8字节的序列对应于一个long或double。ByteBuffer类（只有ByteBuffer类）提供了相对和绝对的put方法，可以用简单类型（boolean除外）参数的相应字节填充缓冲区；ByteBuffer类还提供了相对和绝对的get方法，可以读取适当数量的字节来形成一个新的基本类型数据：

```
public abstract char       getChar()
public abstract ByteBuffer putChar(char value)
public abstract char       getChar(int index)
public abstract ByteBuffer putChar(int index, char value)
public abstract short      getShort()
public abstract ByteBuffer putShort(short value)
public abstract short      getShort(int index)
public abstract ByteBuffer putShort(int index, short value)
public abstract int        getInt()
public abstract ByteBuffer putInt(int value)
public abstract int        getInt(int index)
public abstract ByteBuffer putInt(int index, int value)
public abstract long       getLong()
public abstract ByteBuffer putLong(long value)
public abstract long       getLong(int index)
public abstract ByteBuffer putLong(int index, long value)
public abstract float      getFloat()
```

```
public abstract ByteBuffer putFloat(float value)
public abstract float      getFloat(int index)
public abstract ByteBuffer putFloat(int index, float value)
public abstract double     getDouble()
public abstract ByteBuffer putDouble(double value)
public abstract double     getDouble(int index)
public abstract ByteBuffer putDouble(int index, double value)
```

在新I/O的世界里，这些方法完成了传统I/O中由DataOutputStream和DataInputStream完成的任务。这些方法还有DataOutputStream和DataInputStream所没有的额外能力。你可以选择将字节序列解释为big-endain或little-endian的int、float、double等。默认情况下，所有值都以big-endian方式读/写，即最高字节在前。另外还提供了两个order()方法，可以使用ByteOrder类的命名常量来检查和设置缓冲区的字节顺序。例如，可以将缓冲区改为little-endian，如下：

```
if (buffer.order().equals(ByteOrder.BIG_ENDIAN)) {
  buffer.order(ByteOrder.LITTLE_ENDIAN);
}
```

假设你希望生成二进制数据来测试网络，而不是实现chargen协议。这种测试可以反映出ASCII chargen协议中不明显的一些问题，如有些老的网关配置为会去除每个字节的高位，每$2^{30}$字节就丢弃1字节，或者由于预料之外的控制字符序列而进入诊断模式。这些问题并不只是理论上存在。我已经在实际中一次又一次地看到过这样一些问题。

可以通过发送每一个可能的int来测试网络是否存在这种问题。在大约43亿次迭代后，就能测试完所有可能的4字节序列。在接收端，可以通过简单的数值比较，很容易地测试出接收的是否为期望的数据。如果找到了任何问题，很容易指出问题究竟出现在哪里。换句话说，这个协议（可称为Intgen）的行为如下：

1. 客户端连接服务器。
2. 服务器立即开始发送4字节的big-endian整数，从0开始，每次增1。服务器最后会回绕到负数。
3. 服务器无限运行。客户端在得到足够信息后关闭连接。

服务器可将当前的int存储在4字节长的直接ByteBuffer中。为各个通道关联一个缓冲区。通道可用于写入时，缓冲区就排空（drain）到这个通道。然后回倒缓冲区（rewind），通过getInt()读取缓冲区的内容。接下来程序清空缓冲区，将前面的值增1，使用putInt()用新值填充缓冲区。最后回绕（flip）缓冲区，从而当通道再次可写时能够排空缓冲区。如示例11-3所示。

示例11-3：Intgen服务器

```
import java.nio.*;
```

```
import java.nio.channels.*;
import java.net.*;
import java.util.*;
import java.io.IOException;

public class IntgenServer {

  public static int DEFAULT_PORT = 1919;

  public static void main(String[] args) {

    int port;
    try {
      port = Integer.parseInt(args[0]);
    } catch (RuntimeException ex) {
      port = DEFAULT_PORT;
    }
      System.out.println("Listening for connections on port " + port);

      ServerSocketChannel serverChannel;
      Selector selector;
      try {
        serverChannel = ServerSocketChannel.open();
        ServerSocket ss = serverChannel.socket();
        InetSocketAddress address = new InetSocketAddress(port);
        ss.bind(address);
        serverChannel.configureBlocking(false);
        selector = Selector.open();
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);
      } catch (IOException ex) {
        ex.printStackTrace();
        return;
      }

      while (true) {
        try {
          selector.select();
        } catch (IOException ex) {
          ex.printStackTrace();
          break;
        }

        Set<SelectionKey> readyKeys = selector.selectedKeys();
        Iterator<SelectionKey> iterator = readyKeys.iterator();
        while (iterator.hasNext()) {
          SelectionKey key = iterator.next();
          iterator.remove();
          try {
           if (key.isAcceptable()) {
              ServerSocketChannel server = (ServerSocketChannel) key.channel();
              SocketChannel client = server.accept();
              System.out.println("Accepted connection from " + client);
              client.configureBlocking(false);
              SelectionKey key2 = client.register(selector, SelectionKey.
                                                                  OP_WRITE);
```

```
            ByteBuffer output = ByteBuffer.allocate(4);
            output.putInt(0);
            output.flip();
            key2.attach(output);
          } else if (key.isWritable()) {
            SocketChannel client = (SocketChannel) key.channel();
            ByteBuffer output = (ByteBuffer) key.attachment();
            if (! output.hasRemaining()) {
              output.rewind();
              int value = output.getInt();
              output.clear();
              output.putInt(value + 1);
              output.flip();
            }
            client.write(output);
          }
        } catch (IOException ex) {
          key.cancel();
          try {
            key.channel().close();
          }
          catch (IOException cex) {}
        }
      }
    }
  }
}
```

## 视图缓冲区

如果你知道从SocketChannel读取的ByteBuffer只包含一种特定基本数据类型的元素，那么就有必要创建一个视图缓冲区（view buffer）。这是一个适当类型（如DoubleBuffer、IntBuffer等）的新的Buffer对象，它从当前位置开始由底层ByteBuffer提取数据。修改视图缓冲区会反映到底层缓冲区，反之亦然。不过，每个缓冲区都有自己独立的限度、容量、标记和位置。视图缓冲区可以用ByteBuffer的以下6个方法之一来创建：

```
public abstract ShortBuffer  asShortBuffer()
public abstract CharBuffer   asCharBuffer()
public abstract IntBuffer    asIntBuffer()
public abstract LongBuffer   asLongBuffer()
public abstract FloatBuffer  asFloatBuffer()
public abstract DoubleBuffer asDoubleBuffer()
```

例如，考虑Intgen协议的一个客户端。这个协议只准备读取int，所以使用IntBuffer而不是ByteBuffer会很有帮助。如示例11-4所示。为体现多样性，这个客户端采用了同步和阻塞模式，不过仍然使用了通道和缓冲区。

示例11-4：Intgen客户端

```
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.io.IOException;

public class IntgenClient {

  public static int DEFAULT_PORT = 1919;

  public static void main(String[] args) {

    if (args.length == 0) {
      System.out.println("Usage: java IntgenClient host [port]");
      return;
    }

    int port;
    try {
      port = Integer.parseInt(args[1]);
    } catch (RuntimeException ex) {
      port = DEFAULT_PORT;
    }

    try {
      SocketAddress address = new InetSocketAddress(args[0], port);
      SocketChannel client  = SocketChannel.open(address);
      ByteBuffer    buffer  = ByteBuffer.allocate(4);
      IntBuffer     view    = buffer.asIntBuffer();

      for (int expected = 0; ; expected++) {
        client.read(buffer);
        int actual = view.get();
        buffer.clear();
        view.rewind();

        if (actual != expected) {
          System.err.println("Expected " + expected + "; was " + actual);
          break;
        }
        System.out.println(actual);
      }
    } catch(IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

这里要注意一点，虽然可以完全使用IntBuffer类的方法来填充和排空缓冲区，但数据必须使用原ByteBuffer（IntBuffer就是这个ByteBuffer的视图）对通道进行读/写。SocketChannel类只有读/写ByteBuffer的方法。它无法读/写任何其他类型的缓冲区。这也意味着每次循环都需要清空ByteBuffer，否则缓冲区将被填满，程序将中止。两个缓

冲区（原ByteBuffer和视图IntBuffer）的位置和限度是独立的，必须分别考虑。最后，如果以非阻塞模式工作，要注意在读/写上层视图缓冲区之前，要将底层ByteBuffer中的所有数据排空。非阻塞模式不能保证缓冲区在排空后仍能以int、double或char等类型的边界对齐。向非阻塞通道写入一个int或double的半个字节是完全有可能的。使用非阻塞I/O时，在向视图缓冲区放入更多数据前，要确保检查这个问题。

## 压缩缓冲区

大多数可写缓冲区都支持compact()方法：

```
public abstract ByteBuffer   compact()
public abstract IntBuffer    compact()
public abstract ShortBuffer  compact()
public abstract FloatBuffer  compact()
public abstract CharBuffer   compact()
public abstract DoubleBuffer compact()
```

如果不是为了支持调用串链，这6个方法可以用公共Buffer超类中的一个方法代替。压缩时将缓冲区中所有剩余的数据移到缓冲区的开头，为元素释放更多空间。这些位置上的任何数据都将被覆盖。缓冲区的位置设置为数据末尾，从而可以写入更多数据。

使用非阻塞I/O进行复制时（读取一个通道，再把数据写入另一个通道），压缩是一个特别有用的操作。可以将一些数据读入缓冲区，再写出缓冲区，然后压缩数据，这样所有没有写出的数据就在缓冲区开头，位置则在缓冲区中剩余数据的末尾，准备接收更多数据。这样只利用一个缓冲区就能完成比较随机的交替读/写。可以连续进行几次读取，或者连续几次写入。如果网络准备好可以立即输出但没有输入（或者反之），程序就可以利用缓冲区压缩。这种技术可用来实现示例11-5所示的echo服务器。echo协议只是用客户端发送的数据向客户端做出响应。与chargen一样，echo也用于网络测试。同样类似于chargen，echo依靠客户端来关闭连接。但与chargen有一点不同，echo服务器必须同时对连接进行读/写。

示例11-5：Echo服务器

```
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.util.*;
import java.io.IOException;

public class EchoServer {

  public static int DEFAULT_PORT = 7;

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

```
int port;
try {
  port = Integer.parseInt(args[0]);
} catch (RuntimeException ex) {
  port = DEFAULT_PORT;
}
System.out.println("Listening for connections on port " + port);

ServerSocketChannel serverChannel;
Selector selector;
try {
  serverChannel = ServerSocketChannel.open();
  ServerSocket ss = serverChannel.socket();
  InetSocketAddress address = new InetSocketAddress(port);
  ss.bind(address);
  serverChannel.configureBlocking(false);
  selector = Selector.open();
  serverChannel.register(selector, SelectionKey.OP_ACCEPT);
} catch (IOException ex) {
  ex.printStackTrace();
  return;
}

while (true) {
  try {
    selector.select();
  } catch (IOException ex) {
    ex.printStackTrace();
    break;
  }

  Set<SelectionKey> readyKeys = selector.selectedKeys();
  Iterator<SelectionKey> iterator = readyKeys.iterator();
  while (iterator.hasNext()) {
    SelectionKey key = iterator.next();
    iterator.remove();
    try {
      if (key.isAcceptable()) {
        ServerSocketChannel server = (ServerSocketChannel) key.channel();
        SocketChannel client = server.accept();
        System.out.println("Accepted connection from " + client);
        client.configureBlocking(false);
        SelectionKey clientKey = client.register(
            selector, SelectionKey.OP_WRITE | SelectionKey.OP_READ);
        ByteBuffer buffer = ByteBuffer.allocate(100);
        clientKey.attach(buffer);
      }
      if (key.isReadable()) {
        SocketChannel client = (SocketChannel) key.channel();
        ByteBuffer output = (ByteBuffer) key.attachment();
        client.read(output);
      }
      if (key.isWritable()) {
        SocketChannel client = (SocketChannel) key.channel();
        ByteBuffer output = (ByteBuffer) key.attachment();
```

```
            output.flip();
            client.write(output);
            output.compact();
          }
        } catch (IOException ex) {
          key.cancel();
          try {
            key.channel().close();
          } catch (IOException cex) {}
        }
      }
    }
  }
}
```

编写和调试这个程序时，我注意到一点：缓冲区大小非常重要，但其影响可能并非像你预想的那样。大的缓冲区可能会隐藏大量bug。如果缓冲区足够大，可以保存完整的测试用例，不需要进行回绕或排空，那么往往不会注意到程序未在适当的时候对缓冲区进行回绕或压缩，因为实际上测试用例从来都不需要这么做。在发行你的程序之前，应当尝试减少缓冲区大小，要比期望的输入小很多。在这里，我对缓冲区大小为10的情况进行了测试。这个测试会降低性能，所以真正发行程序时不应设置这么小的缓冲区，但绝对应该用小缓冲区测试代码，确保程序在缓冲区填满时仍能正常工作。

## 复制缓冲区

经常需要建立缓冲区的副本，从而将相同的信息分发到两个或多个通道。6种特定类型缓冲区类都提供了duplicate()方法来完成这项工作：

```
public abstract ByteBuffer   duplicate()
public abstract IntBuffer    duplicate()
public abstract ShortBuffer  duplicate()
public abstract FloatBuffer  duplicate()
public abstract CharBuffer   duplicate()
public abstract DoubleBuffer duplicate()
```

返回值并不是克隆。复制的缓冲区共享相同的数据，如果原来是一个间接缓冲区，那么复制缓冲区会包含相同的后备数组。修改一个缓冲区中的数据会反映到另一个缓冲区中。所以这个方法主要用在只准备读取缓冲区时。否则，要想跟踪数据哪里有修改，这可能很困难。

尽管共享相同的数据，但初始和复制的缓冲区有独立的标记、限度和位置。一个缓冲区可以超前或落后于另一个缓冲区。

希望通过多个通道大致并行地传输相同的数据时，复制非常有用。可以为每个通道建立主缓冲区的副本，让每个通道以其自己的速度运行。例如，回忆一下示例9-10

中的单文件HTTP服务器。这里用通道和缓冲区重新实现了这个服务器，如示例11-6 NonblockingSingleFileHTTPServer所示，要提供的文件存储在一个稳定的只读缓冲区中。每当客户端连接时，程序就为该通道建立这个缓冲区的一个副本，这会存储为这个通道的附件。如果不使用复制，一个客户端就必须等待另外一个客户端结束，这样初始缓冲区才能回倒（rewind）。复制则允许并发地重用缓冲区。

示例11-6：提供一个文件的非阻塞HTTP服务器

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;
import java.net.*;

public class NonblockingSingleFileHTTPServer {

  private ByteBuffer contentBuffer;
  private int port = 80;

  public NonblockingSingleFileHTTPServer(
      ByteBuffer data, String encoding, String MIMEType, int port) {

    this.port = port;
    String header = "HTTP/1.0 200 OK\r\n"
        + "Server: NonblockingSingleFileHTTPServer\r\n"
        + "Content-length: " + data.limit() + "\r\n"
        + "Content-type: " + MIMEType + "\r\n\r\n";
    byte[] headerData = header.getBytes(Charset.forName("US-ASCII"));

    ByteBuffer buffer = ByteBuffer.allocate(
        data.limit() + headerData.length);
    buffer.put(headerData);
    buffer.put(data);
    buffer.flip();
    this.contentBuffer = buffer;
  }

  public void run() throws IOException {
    ServerSocketChannel serverChannel = ServerSocketChannel.open();
    ServerSocket serverSocket = serverChannel.socket();
    Selector selector = Selector.open();
    InetSocketAddress localPort = new InetSocketAddress(port);
    serverSocket.bind(localPort);
    serverChannel.configureBlocking(false);
    serverChannel.register(selector, SelectionKey.OP_ACCEPT);

    while (true) {
      selector.select();
      Iterator<SelectionKey> keys = selector.selectedKeys().iterator();
      while (keys.hasNext()) {
        SelectionKey key = keys.next();
```

```
        keys.remove();
        try {
          if (key.isAcceptable()) {
            ServerSocketChannel server = (ServerSocketChannel) key.channel();
            SocketChannel channel = server.accept();
            channel.configureBlocking(false);
            channel.register(selector, SelectionKey.OP_READ);
          } else if (key.isWritable()) {
            SocketChannel channel = (SocketChannel) key.channel();
            ByteBuffer buffer = (ByteBuffer) key.attachment();
            if (buffer.hasRemaining()) {
               channel.write(buffer);
            } else { // 结束工作
              channel.close();
            }
          } else if (key.isReadable()) {
            // 不用费力地解析HTTP首部
            // 只需读取
            SocketChannel channel = (SocketChannel) key.channel();
            ByteBuffer buffer = ByteBuffer.allocate(4096);
            channel.read(buffer);
            // 将通道切换为只写模式
            key.interestOps(SelectionKey.OP_WRITE);
            key.attach(contentBuffer.duplicate());
          }
        } catch (IOException ex) {
          key.cancel();
          try {
            key.channel().close();
          }
          catch (IOException cex) {}
        }
      }
    }
  }

  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println(
        "Usage: java NonblockingSingleFileHTTPServer file port encoding");
      return;
    }

    try {
      // 读取要提供的文件
      String contentType =
          URLConnection.getFileNameMap().getContentTypeFor(args[0]);
      Path file = FileSystems.getDefault().getPath(args[0]);
      byte[] data = Files.readAllBytes(file);
      ByteBuffer input = ByteBuffer.wrap(data);

      // 设置要监听的端口
      int port;
      try {
        port = Integer.parseInt(args[1]);
```

```
        if (port < 1 || port > 65535) port = 80;
      } catch (RuntimeException ex) {
        port = 80;
      }

      String encoding = "UTF-8";
      if (args.length > 2) encoding = args[2];

      NonblockingSingleFileHTTPServer server
          = new NonblockingSingleFileHTTPServer(
          input, encoding, contentType, port);
      server.run();
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

构造函数建立将与HTTP首部一起发送的数据，HTTP首部中包括内容长度和内容编码方式的有关信息。响应首部和主体存储在一个ByteBuffer中，这样可以非常快地发送给客户端。不过，尽管所有客户端都会接收相同的内容，但它们可能不会同时收到。不同的并行客户端将读到文件中的不同位置。这就是我们复制缓冲区的原因，这样一来，每个通道都会有自己的缓冲区。开销并不大，因为所有通道实际上会共享相同的内容。它们只是有这个内容的不同索引而已。

所有入站连接都在run()方法中由一个Selector处理。这里最初的设置与前面的chargen服务器非常相似。run()方法打开一个ServerSocketChannel，把它绑定到指定的端口。然后创建一个Selector，并向这个Selector注册ServerSocketChannel。接受一个SocketChannel时，则向同一个Selector对象注册。开始时将其注册为关注读取操作，因为HTTP要求客户端在服务器响应前先发送请求。

对读取的响应有所简化。程序最多只读取4KB输入。然后它将通道所关注的操作重新设置为写入（更完整的服务器实际上会在这里尝试解析HTTP首部请求，根据这个信息选择要发送的文件）。接下来，复制内容缓冲区，并附加到通道上。

程序下一次执行while循环时，这个通道应当已经准备好接收数据（如果下一次还没有，就等到再下一次。连接的异步特性意味着，如果连接没有就绪，我们根本不会看到这个连接）。至此，可以从附件得到缓冲区，并将缓冲区中数据尽可能多地写入通道。如果这一次没有全部写入也没有关系。只要在下一次循环时从之前的位置继续写入即可。缓冲区会记录自己的位置。虽然多个入站客户端可能会导致创建很多缓冲区对象，但实际的开销是极小的，因为它们都共享相同的底层数据。

main()方法从命令行读取参数。从第一个命令行参数读取所提供文件的文件名。如果没有指定文件或者文件无法打开，程序就会显示一条错误消息并退出。假设文件可

以读取，则使用Java 7中方便的Path和Files类将其内容读入一个ByteBuffer。会对文件的内容类型做出合理的猜测，猜测结果存储在contentType变量中。接下来，从第二个命令行参数读取端口号。如果没有指定端口，或者第二个参数不是1到65535之间的某个整数，就使用端口80。如果给出了第三个命令行参数，则从第三个命令行参数读取编码方式，否则就假定编码方式为UTF-8。然后使用这些值构造一个NonblockingSingleFileHTTPServer对象，并开始运行。

## 分片缓冲区

分片（slicing）缓冲区是复制的一个变形。分片也会创建一个新缓冲区，与原缓冲区共享数据。不过，分片的起始位置（位置0）是原缓冲区的当前位置，而且其容量最大不超过源缓冲区的限度。也就是说，分片是原缓冲区的一个子序列，只包含从当前位置到限度的所有元素。当创建分片时，倒回分片只移回到原缓冲区的位置。分片无法看到原缓冲区中该点之前的内容。同样的，6种特定类型的缓冲区类都有单独的slice()方法：

```
public abstract ByteBuffer   slice()
public abstract IntBuffer    slice()
public abstract ShortBuffer  slice()
public abstract FloatBuffer  slice()
public abstract CharBuffer   slice()
public abstract DoubleBuffer slice()
```

如果你有一个很长的数据缓冲区，很容易地分为多个部分（如协议首部以及数据），此时分片就很有用。可以读出首部，然后对缓冲区分片，将只包含数据的新缓冲区传递给一个单独的方法或类。

## 标记和重置

与输入流类似，如果希望重新读取某些数据，可以标记和重置缓冲区。另外与输入流不同的是，这将应用于所有缓冲区，而不只是部分缓冲区。为了有所改变，相关的方法只在Buffer超类中声明一次，并由各种子类继承：

```
public final Buffer mark()
public final Buffer reset()
```

如果没有设置标记，reset()方法会抛出一个InvalidMarkException异常，这是一个运行时异常。如果将位置设置到标记的前面，则会清除这个标记。

## Object方法

缓冲区类都提供了一般的equals()、hashCode()和toString()方法。它们还实现了

Comparable接口，因此提供了compareTo()方法。不过，缓冲区不是Serializable或Cloneable的。

当满足以下条件时，会认为两个缓冲区是相等的：

- 它们都具有相同的类型（例如，ByteBuffer永远不会与IntBuffer相等，只可能与另一个ByteBuffer相等）。
- 缓冲区中剩余的元素个数相同。
- 相同相对位置上的剩余元素彼此相等。

注意相等性并不考虑缓冲区中位置之前的元素，也不考虑缓冲区的容量、限度和标记。例如，下面的代码段将输出true，尽管第一个缓冲区的大小是第二个的两倍：

```
CharBuffer buffer1 = CharBuffer.wrap("12345678");
CharBuffer buffer2 = CharBuffer.wrap("5678");
buffer1.get();
buffer1.get();
buffer1.get();
buffer1.get();
System.out.println(buffer1.equals(buffer2));
```

hashCode()方法是依照相等性实现的。也就是说，两个相等的缓冲区有相同的散列码，而两个不相等的缓冲区几乎不可能有相同的散列码。不过，每次向缓冲区添加或删除元素时，缓冲区的散列码都会改变，所以缓冲区不能生成好的散列键。

要进行比较，这是通过对各个缓冲区中的剩余元素逐个进行比较来实现的。如果所有对应元素都相等，则缓冲区相等。否则，结果就是第一对不相等元素的比较结果。如果在找到不相等的元素之前，如果其中一个缓冲区已经没有元素了（所有元素都已经做过比较），而另一个缓冲区还有元素，则认为较短的缓冲区小于较长的缓冲区。

toString()方法返回形式如下的字符串：

```
java.nio.HeapByteBuffer[pos=0 lim=62 cap=62]
```

这主要用于调试。唯一需要注意的例外是CharBuffer，它会返回一个字符串，其中包含缓冲区中的剩余字符。

# 通道

通道将缓冲区的数据块移入或移出到各种I/O源，如文件、socket、数据报等。通道类的层次结构相当复杂，有多个接口和许多可选操作。不过，对于网络编程来说，实际上只

有3个重要的通道类：SocketChannel、ServerSocketChannel和DatagramChannel。对于目前为止谈到的TCP连接，只需要前两个通道类。

## SocketChannel

SocketChannel类可以读写TCP Socket。数据必须编码到ByteBuffer对象中来完成读/写。每个SocketChannel都与一个对等端（peer）Socket对象相关联，这个Socket可以用于高级配置，但有些应用采用默认选项就可以正常运行，对于这些应用程序，可以忽略这个需求。

### 连接

SocketChannel类没有任何公共构造函数。实际上，要使用两个静态open()方法来创建新的SocketChannel对象：

```
public static SocketChannel open(SocketAddress remote) throws IOException
public static SocketChannel open() throws IOException
```

第一个方法会建立连接。这个方法将阻塞（也就是说，在连接建立或抛出异常之前，这个方法不会返回）。例如：

```
SocketAddress address = new InetSocketAddress("www.cafeaulait.org", 80);
SocketChannel channel = SocketChannel.open(address);
```

无参数版本不立即连接。它创建一个初始未连接的socket，以后必须用connect()方法进行连接。例如：

```
SocketChannel channel = SocketChannel.open();
SocketAddress address = new InetSocketAddress("www.cafeaulait.org", 80);
channel.connect(address);
```

为了在连接前配置通道和（或）socket的各种选项，你可能会选择这种更迂回的方法。特别是如果希望以无阻塞方式打开通道时，就要使用这种方法：

```
SocketChannel channel = SocketChannel.open();
SocketAddress address = new InetSocketAddress("www.cafeaulait.org", 80);
channel.configureBlocking(false);
channel.connect();
```

使用非阻塞通道时，connect()方法会立即返回，甚至在连接建立之前就会返回。在等待操作系统建立连接时，程序可以做其他的操作。不过，程序在实际使用连接之前，必须调用finishConnect()：

```
public abstract boolean finishConnect() throws IOException
```

这只对非阻塞模式是必需的。对于阻塞通道，这个finishConnect()方法将立即返回true。如果连接现在可以使用，finishConnect()就返回true。如果连接还没有建立，finishConnect()就返回false。最后，如果连接无法建立，比如因为网络出现故障，这个方法将抛出一个异常。

如果程序想检查连接是否完成，可以调用以下两个方法：

```
public abstract boolean isConnected()
public abstract boolean isConnectionPending()
```

如果连接打开，isConnected()方法会返回true。如果连接仍在建立但尚未打开时，isConnectionPending()返回true。

### 读取

为了读取SocketChannel，首先要创建一个ByteBuffer，通道可以在其中存储数据。然后将这个ByteBuffer传给read()方法：

```
public abstract int read(ByteBuffer dst) throws IOException
```

通道会用尽可能多的数据填充缓冲区，然后返回放入的字节数。如果遇到流末尾，通道会用所有剩余的字节填充缓冲区，而且在下一次调用read()时返回－1。如果通道是阻塞的，这个方法将至少读取一个字节，或者返回－1，也可能抛出一个异常。但如果通道是非阻塞的，这个方法可能返回0。

因为数据将存储在缓冲区的当前位置，而这个位置会随着增加更多数据而自动更新，所以可以一直向read()方法传入同一个缓冲区，直到缓冲区填满。例如，下面的循环会一直读取数据，直到缓冲区填满或者检测到流末尾为止：

```
while (buffer.hasRemaining() && channel.read(buffer) != -1) ;
```

有时如果能从一个源填充多个缓冲区，这会很有用。这称为散布（scatter）。下面两个方法接受一个ByteBuffer对象数组作为参数，按顺序填充数组中的各个ByteBuffer：

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

第一个方法填充所有缓冲区。第二个方法则从位于offset的缓冲区开始，填充length个缓冲区。

要填充缓冲区数组，只要在列表中最后一个缓冲区还有剩余空间，就可以继续循环。例如：

```
ByteBuffer[] buffers = new ByteBuffer[2];
buffers[0] = ByteBuffer.allocate(1000);
buffers[1] = ByteBuffer.allocate(1000);
while (buffers[1].hasRemaining() && channel.read(buffers) != -1) ;
```

## 写入

Socket通道提供了读写方法。一般情况下它们是全双工的。要想写入，只需填充一个ByteBuffer，将其回绕，然后传给某个写入方法，这个方法在把数据复制到输出时将缓冲区排空，这与读取过程正好相反。

基本的write()方法接收一个缓冲区作为参数：

```
public abstract int write(ByteBuffer src) throws IOException
```

与读取一样（而且不同于OutputStream），如果通道是非阻塞的，这个方法不能保证会写入缓冲区的全部内容。不过再次说明，由于缓冲区基于游标的特性，你可以很容易地反复调用这个方法，直到缓冲区完全排空，而且数据已完全写入：

```
while (buffer.hasRemaining() && channel.write(buffer) != -1) ;
```

将多个缓冲区的数据写入到一个Socket通常很有用。这称为聚集（gather）。例如，你可能希望在一个缓冲区中存储HTTP首部，而在另一个缓冲区中存储HTTP主体。具体实现甚至可以使用两个线程或重叠的I/O同时填充这两个缓冲区。下面两个方法接受一个ByteBuffer对象数组作为参数，并按顺序排空：

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

第一个方法排空所有缓冲区。第二个方法则从位于offset的缓冲区开始，排空length个缓冲区。

## 关闭

就像正常Socket一样，在用完通道后应当将其关闭，释放它可能使用的端口和其他任何资源：

```
public void close() throws IOException
```

如果通道已经关闭，再进行关闭将没有任何效果。如果试图读/写已关闭的通道，将抛出一个异常。如果不确定通道是否已关闭，可以用isOpen()检查：

```
public boolean isOpen()
```

很自然地，通道已关闭时，这个方法返回false，如果通道是打开的，则返回true。close()和isOpen()是Channel接口中声明的仅有的两个方法，所有通道类都共享这两个方法。

从Java 7开始，SocketChannel实现了AutoCloseable，所以可以在try-with-resources中使用。

## ServerSocketChannel

ServerSocketChannel类只有一个目的：接受入站连接。你无法读取、写入或连接ServerSocketChannel。它支持的唯一操作是接受一个新的入站连接。这个类本身只声明了4个方法，其中accept()最重要。ServerSocketChannel还从其超类继承了几个方法，主要与向Selector注册来得到入站连接通知有关。最后，与所有通道一样，它有一个close()方法，用于关闭服务器Socket。

### 创建服务器Socket通道

静态工厂方法ServerSocketChannel.open()创建一个新的ServerSocketChannel对象。不过，这个方法名有一点欺骗性。这个方法实际上并不打开一个新的服务器Socket，而是只创建这个对象。在使用之前，需要调用socket()方法来获得相应的对等端（peer）ServerSocket。此时，你可以使用ServerSocket的各种设置方法随意配置任何服务器选项，如接收缓冲区大小或Socket超时值。然后，对于你希望绑定的端口，将这个ServerSocket连接到对应该端口的SocketAddress。例如，下面的代码段在端口80上打开一个ServerSocketChannel：

```
try {
  ServerSocketChannel server = ServerSocketChannel.open();
  ServerSocket socket = serverChannel.socket();
  SocketAddress address = new InetSocketAddress(80);
  socket.bind(address);
} catch (IOException ex) {
  System.err.println("Could not bind to port 80 because " + ex.getMessage());
}
```

在Java 7中，这会更简单一些，因为现在ServerSocketChannel有了自己的一个bind()方法：

```
try {
  ServerSocketChannel server = ServerSocketChannel.open();
  SocketAddress address = new InetSocketAddress(80);
  server.bind(address);
} catch (IOException ex) {
  System.err.println("Could not bind to port 80 because " + ex.getMessage());
}
```

这里使用了工厂方法而不是构造函数，所以不同的虚拟机可以提供这个类的不同实现，从而更适用于本地硬件和操作系统。不过，这个工厂是用户不可配置的。在相同的虚拟机中运行时，open()总是返回相同类的实例。

### 接受连接

一旦打开并绑定了ServerSocketChannel对象，accept()方法就可以监听入站连接了：

```
public abstract SocketChannel accept() throws IOException
```

accept()可以在阻塞或非阻塞模式下操作。在阻塞模式下，accept()方法等待入站连接。然后它接受一个连接，并返回连接到远程客户端的一个SocketChannel对象。在建立连接之前，线程无法进行任何操作。这种策略适用于立即响应每一个请求的简单服务器。阻塞模式是默认模式。

ServerSocketChannel还可以在非阻塞模式下操作。在这种情况下，如果没有入站连接，accept()方法会返回null。非阻塞模式更适合于需要为每个连接完成大量工作的服务器，这样就可以并行地处理多个请求。非阻塞模式一般与Selector结合使用。为了使ServerSocketChannel处于非阻塞模式，要向其configureBlocking()方法传入false。

accept()方法声明为出现错误时抛出一个IOException异常。IOException的几个子类以及几个运行时异常可以指示更详细的问题：

ClosedChannelException
: 关闭后无法重新打开一个ServerSocketChannel。

AsynchronousCloseException
: 执行accept()时，另一个线程关闭了这个ServerSocketChannel。

ClosedByInterruptException
: 一个阻塞ServerSocketChannel在等待时，另一个线程中断了这个线程。

NotYetBoundException
: 调用了open()，但在调用accept()之前没有将ServerSocketChannel的对等端（peer）ServerSocket与地址绑定。这是一个运行时异常，不是IOException异常。

SecurityException
: 安全管理器拒绝这个应用程序绑定所请求的端口。

## Channels类

Channels是一个简单的工具类，可以将传统的基于I/O的流、阅读器和书写器包装在通道

中，也可以从通道转换为基于I/O的流、阅读器和书写器。如果出于性能考虑，希望在程序的一部分中使用新I/O模型，但同时仍要与处理流的传统API交互，这个类会很有用。它有一些方法可以从流转换为通道，还有一些方法可以从通道转换为流、阅读器、书写器：

```
public static InputStream newInputStream(ReadableByteChannel ch)
public static OutputStream newOutputStream(WritableByteChannel ch)
public static ReadableByteChannel newChannel(InputStream in)
public static WritableByteChannel newChannel(OutputStream out)
public static Reader newReader (ReadableByteChannel channel,
    CharsetDecoder decoder, int minimumBufferCapacity)
public static Reader newReader (ReadableByteChannel ch, String encoding)
public static Writer newWriter (WritableByteChannel ch, String encoding)
```

本章讨论的SocketChannel类实现了这些方法签名中出现的ReadableByteChannel和WritableByteChannel接口。ServerSocketChannel则二者都没有实现，因为无法对ServerSocketChannel进行读/写。

例如，所有当前的XML API都使用流、文件、阅读器和其他传统I/O API来读取XML文档。如果编写一个HTTP服务器，用于处理SOAP请求，可能会出于性能原因使用通道读取HTTP请求主体，而使用SAX解析XML。在这种情况下，就需要将通道转换为流，再传给XMLReader的parse()方法：

```
SocketChannel channel = server.accept();
processHTTPHeader(channel);
XMLReader parser = XMLReaderFactory.createXMLReader();
parser.setContentHandler(someContentHandlerObject);
InputStream in = Channels.newInputStream(channel);
parser.parse(in);
```

## 异步通道 (Java 7)

Java 7引入AsynchronousSocketChannel和AsynchronousServerSocketChannel类。这些类的表现与SocketChannel和ServerSocketChannel很类似，而且接口也基本相同（但AsynchronousSocketChannel和AsynchronousServerSocketChannel类并不是这些类的子类）。不过，与SocketChannel和ServerSocketChannel不同的是，读/写异步通道会立即返回，甚至在I/O完成之前就会返回。所读/写的数据会由一个Future或CompletionHandler进一步处理。connect()和accept()方法也会异步执行，并返回Future。这里不使用选择器。

例如，假设一个程序需要在启动时完成大量初始化工作。另外有一些涉及网络连接的操作，每个操作分别要花费几秒的时间。可以并行开始多个异步操作，然后完成你的本地初始化，再请求这些网络操作的结果：

```
SocketAddress address = new InetSocketAddress(args[0], port);
AsynchronousSocketChannel client = AsynchronousSocketChannel.open();
Future<Void> connected = client.connect(address);

ByteBuffer buffer = ByteBuffer.allocate(74);

// 等待连接完成
connected.get();

// 从连接读取
Future<Integer> future = client.read(buffer);

// 做其他工作...

// 等待读取完成...
future.get();

// 回绕并排空缓冲区
buffer.flip();
WritableByteChannel out = Channels.newChannel(System.out);
out.write(buffer);
```

这个方法的好处是，网络连接在并行运行，与此同时程序可以做其他事情。准备好处理来自网络的数据时，会停下来，通过调用Future.get()等待这些数据，但在此之前不用停下来。利用线程池和callable也可以达到同样的效果，不过这种方法可能稍微简单一些，特别是如果你的应用恰好非常适合使用缓冲区，就可以采用这个方法。

如果你希望以一种非常特定的顺序获取结果，这种情况下这个方法就很适用。不过，如果你不关心顺序，而且可以独立地处理各个网络读取，那么最好使用CompletionHandler。例如，假设你在编写一个搜索引擎Web蜘蛛程序，它会向某个后端提供页面。由于你不关心响应以什么顺序返回，所以可以生成大量AsynchronousSocketChannel请求，并为每个请求提供一个CompletionHandler，由它在后端存储结果。

通用CompletionHandler接口声明了两个方法：completed()和failed()，如果成功地完成读取则调用completed()，另外出现I/O错误时会调用failed()。例如，下面给出一个简单的CompletionHandler，它会在System.out上显示它接收到的所有内容：

```
class LineHandler implements CompletionHandler<Integer, ByteBuffer> {

  @Override
  public void completed(Integer result, ByteBuffer buffer) {
    buffer.flip();
    WritableByteChannel out = Channels.newChannel(System.out);
    try {
      out.write(buffer);
    } catch (IOException ex) {
      System.err.println(ex);
```

```
        }
      }

      @Override
      public void failed(Throwable ex, ByteBuffer attachment) {
        System.err.println(ex.getMessage());
      }
    }
```

读取通道时，要向read()方法传入一个缓冲区、一个附件和一个CompletionHandler：

```
ByteBuffer buffer = ByteBuffer.allocate(74);
CompletionHandler<Integer, ByteBuffer> handler = new LineHandler();
channel.read(buffer, buffer, handler);
```

这里我把缓冲区本身作为附件。这是将从网络读取的数据推至CompletionHandler并在那里进行处理的一种方法。另外一个常用的模式是将CompletionHandler作为一个匿名内部类，缓冲区作为一个最终局部变量，所以它在这个完成处理器（CompletionHandler）的作用域中。

尽管可以在多个线程之间安全地共享一个AsynchronousSocketChannel或AsynchronousServerSocketChannel，但一次只能有一个线程可以读取这个通道（不过，可以有一个线程读，同时另一个线程写）。如果一个线程试图读通道，而另一个线程的读操作还没有结束，read()方法会抛出一个ReadPendingException异常。类似的，如果一个线程试图写，而另一个线程正在写，write()方法就会抛出一个WritePendingException异常。

## Socket选项 (Java 7)

从Java 7开始，SocketChannel、ServerSocketChannel、AsynchronousServerSocketChannel、AsynchronousSocketChannel和DatagramChannel都实现了新的NetworkChannel接口。这个接口的主要用途是支持各种TCP选项，如第8章和第9章讨论的TCP_NODELAY、SO_TIMEOUT、SO_LINGER、SO_SNDBUF、SO_RCVBUF和SO_KEEPALIVE。不论是在Socket上设置还是在通道上设置，这些选项在底层TCP栈中都有相同的含义。不过，这些选项的接口稍有些不同。并非对应所支持的各个选项有单独的方法，通道类分别有3个方法来获取、设置和列出所支持的选项：

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

SocketOption类是一个泛型类，指定了各个选项的名字和类型。类型参数<T>确定这个选

项是一个boolean、Integer，还是NetworkInterface。StandardSocketOptions类为Java能识别的11个选项提供了相应的常量：

- SocketOption<NetworkInterface> StandardSocketOptions.IP_MULTICAST_IF
- SocketOption<Boolean> StandardSocketOptions.IP_MULTICAST_LOOP
- SocketOption<Integer> StandardSocketOptions.IP_MULTICAST_TTL
- SocketOption<Integer> StandardSocketOptions.IP_TOS
- SocketOption<Boolean> StandardSocketOptions.SO_BROADCAST
- SocketOption<Boolean> StandardSocketOptions.SO_KEEPALIVE
- SocketOption<Integer> StandardSocketOptions.SO_LINGER
- SocketOption<Integer> StandardSocketOptions.SO_RCVBUF
- SocketOption<Boolean> StandardSocketOptions.SO_REUSEADDR
- SocketOption<Integer> StandardSocketOptions.SO_SNDBUF
- SocketOption<Boolean> StandardSocketOptions.TCP_NODELAY

例如，下面的代码段会打开一个客户端网络通道，并设置SO_LINGER为240s：

```
NetworkChannel channel = SocketChannel.open();
channel.setOption(StandardSocketOptions.SO_LINGER, 240);
```

不同的通道和Socket支持不同的选项。例如，ServerSocketChannel支持SO_REUSEADDR和SO_RCVBUF，但不支持SO_SNDBUF。如果试图设置通道不支持的一个选项，会抛出一个UnsupportedOperationException异常。

示例11-7是一个简单的程序，列出了不同类型网络通道支持的所有Socket选项。

示例11-7：列出支持的选项

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

public class OptionSupport {

  public static void main(String[] args) throws IOException {
    printOptions(SocketChannel.open());
    printOptions(ServerSocketChannel.open());
    printOptions(AsynchronousSocketChannel.open());
    printOptions(AsynchronousServerSocketChannel.open());
    printOptions(DatagramChannel.open());
  }
```

```
  private static void printOptions(NetworkChannel channel) throws IOException {
    System.out.println(channel.getClass().getSimpleName() + " supports:");
    for (SocketOption<?> option : channel.supportedOptions()) {
      System.out.println(option.name() + ": " + channel.getOption(option));
    }
    System.out.println();
    channel.close();
  }

}
```

下面的输出显示了哪种类型的通道支持哪些选项以及选项的默认值：

```
SocketChannelImpl supports:
SO_OOBINLINE: false
SO_REUSEADDR: false
SO_LINGER: -1
SO_KEEPALIVE: false
IP_TOS: 0
SO_SNDBUF: 131072
SO_RCVBUF: 131072
TCP_NODELAY: false

ServerSocketChannelImpl supports:
SO_REUSEADDR: true
SO_RCVBUF: 131072

UnixAsynchronousSocketChannelImpl supports:
SO_KEEPALIVE: false
SO_REUSEADDR: false
SO_SNDBUF: 131072
TCP_NODELAY: false
SO_RCVBUF: 131072

UnixAsynchronousServerSocketChannelImpl supports:
SO_REUSEADDR: true
SO_RCVBUF: 131072

DatagramChannelImpl supports:
IP_MULTICAST_TTL: 1
SO_BROADCAST: false
SO_REUSEADDR: false
IP_MULTICAST_IF: null
IP_TOS: 0
IP_MULTICAST_LOOP: true
SO_SNDBUF: 9216
SO_RCVBUF: 196724
```

# 就绪选择

对于网络编程，新I/O API的第二部分是就绪选择，即能够选择读写时不阻塞的Socket。

这主要针对于服务器，但对于打开多个窗口并运行多个并发连接的客户端（如Web蜘蛛程序或浏览器）来说，也可以利用这个特性。

为了完成就绪选择，要将不同的通道注册到一个Selector对象。每个通道分配有一个SelectionKey。然后程序可以询问这个Selector对象，哪些通道已经准备就绪可以无阻塞地完成你希望完成的操作，可以请求Selector对象返回相应的键集合。

## Selector类

Selector唯一的构造函数是一个保护类型方法。一般情况下，要调用静态工厂方法Selector.open()来创建新的选择器：

```
public static Selector open() throws IOException
```

下一步是向选择器增加通道。Selector类没有增加通道的方法。register()方法在SelectableChannel类中声明。并不是所有通道都是可选择的（特别是FileChannel就不可选择），不过所有网络通道都是可选择的。因此，通过将选择器传递给通道的一个注册方法，就可以向选择器注册这个通道：

```
public final SelectionKey register(Selector sel, int ops)
    throws ClosedChannelException
public final SelectionKey register(Selector sel, int ops, Object att)
    throws ClosedChannelException
```

我觉得这种方法是一种倒退，但使用起来并不难。第一个参数是通道要向哪个选择器注册。第二个参数是SelectionKey类中的一个命名常量，标识通道所注册的操作。SelectionKey定义了4个命名位常量，用于选择操作类型：

- SelectionKey.OP_ACCEPT
- SelectionKey.OP_CONNECT
- SelectionKey.OP_READ
- SelectionKey.OP_WRITE

这些都是位标志整型常量（1、2、4等）。因此，如果一个通道需要在同一个选择器中关注多个操作（例如读和写一个socket），只要在注册时利用位“或”操作符（|）组合这些常量就可以了：

```
channel.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```

第三个参数是可选的，这是键的附件。这个对象通常用于存储连接的状态。例如，如果

要实现一个Web服务器，可能要附加一个FileInputStream或FileChannel，这个流或通道连接到服务器提供给客户端的本地文件。

不同的通道注册到选择器后，就可以随时查询选择器，找出哪些通道已经准备好可以进行处理。通道可能已经准备好完成某些操作，但对另一些操作还没有准备好。例如，可能一个通道已经准备就绪可以读取，但还不能写入。

有三个方法可以选择就绪的通道。它们的区别在于寻找就绪通道等待的时间。第一个selectNow()方法会完成非阻塞选择。如果当前没有准备好要处理的连接，它会立即返回：

```
public abstract int selectNow() throws IOException
```

另外两个选择方法是阻塞的：

```
public abstract int select() throws IOException
public abstract int select(long timeout) throws IOException
```

第一个方法在返回前会等待，直到至少有一个注册的通道准备好可以进行处理。第二个在返回0前只等待不超过timeout毫秒。如果没有通道就绪程序就不做任何操作，这些方法会很有用。

当知道有通道已经准备好处理时，可以使用selectedKeys()方法获取就绪通道：

```
public abstract Set<SelectionKey> selectedKeys()
```

迭代处理返回的集合时，要依次处理各个SelectionKey。还可以从迭代器中删除键，告诉选择器这个键已经得到处理。否则选择器在以后循环时还会一直通知你有这个键。

最后，当准备关闭服务器或不再需要选择器时，应当将它关闭：

```
public abstract void close() throws IOException
```

这个步骤会释放与选择器关联的所有资源。更重要的是，它取消了向选择器注册的所有键，并中断被这个选择器的某个选择方法所阻塞的线程。

## SelectionKey类

SelectionKey对象相当于通道的指针。它们还可以保存一个对象附件，一般会存储这个通道上的连接的状态。

将一个通道注册到一个选择器时，register()方法会返回SelectionKey对象。不过，通

常你不需要保留这个引用。selectedKeys()方法可以在Set中再次返回相同的对象。一个通道可以注册到多个选择器。

当从所选择的键集合中获取一个SelectionKey时，通常首先要测试这些键能进行哪些操作。有以下4种可能：

```
public final boolean isAcceptable()
public final boolean isConnectable()
public final boolean isReadable()
public final boolean isWritable()
```

这个测试并不总是必须的。有些情况下，选择器只测试一种可能性，也只返回完成这种操作的键。但如果选择器确实要测试多种就绪状态，就要在操作前先测试通道对于哪个操作进入就绪状态。也有可能通道准备好可以完成多个操作。

一旦了解了与键关联的通道准备好完成何种操作，就可以用channel()方法来获取这个通道：

```
public abstract SelectableChannel channel()
```

如果在保存状态信息的SelectionKey存储了一个对象，就可以用attachment()方法获取该对象：

```
public final Object attachment()
```

最后，如果结束使用连接，就要撤销其SelectionKey对象的注册，这样选择器就不会浪费资源再去查询它是否准备就绪。我不知道这是否在所有情况下都非常重要，但起码没有坏处。可以调用这个键的cancel()方法来撤销注册：

```
public abstract void cancel()
```

不过，只有在未关闭通道时这个步骤才有必要。如果关闭通道，会自动在所有选择器中撤销对应这个通道的所有键的注册。类似地，关闭选择器会使这个选择器中的所有键都失效。

# 第12章

# UDP

前面几章讨论了在TCP传输层协议之上运行的网络应用程序。TCP是为数据的可靠传输而设计的。如果数据在传输中丢失或损坏，TCP会保证再次发送数据。如果数据包乱序到达，TCP会将其置回正确的顺序。对于连接来说，如果数据到来的速度太快，TCP会降低速度，以免数据包丢失。程序永远不需要担心接收到乱序或不正确的数据。不过，这种可靠性是有代价的。这个代价就是速度。建立和撤销TCP连接会花费相当长的时间，对于某些协议，特别是像HTTP这样的协议，这是无法接受的（这些协议往往需要多个短时间的数据传输）。

用户数据报协议（User Datagram Protocol，UDP）是在IP之上发送数据的另一种传输层协议，速度很快，但不可靠。当发送UDP数据时，无法知道数据是否会到达，也不知道数据的各个部分是否会以发送时的顺序到达。不过，确实能到达的部分一般都会很快到达。

## UDP协议

对此有一个很明显的问题，为什么会有人要使用一个不可靠的协议呢？如果有需要发送的数据，你当然会关心数据是否能正确到达，对不对？很显然，对于类似FTP的应用程序，由于需要通过可能不可靠的网络进行可靠的数据传输，因此UDP不是一个好的选择。不过，另外还有很多其他类型的应用程序，在这些应用程序中，保持最快的速度比保证每一位数据都正确更为重要。例如，在实时音频或视频中，丢失或交换数据包只会作为干扰出现。干扰是可以容忍的，但当TCP请求重传或等待数据包到达而它却迟迟不到时，音频流中就会出现尴尬的停顿，这是让人无法接受的。在其他应用程序中，可以在应用层实现可靠性测试。例如，如果客户端向服务器发送一个短的UDP请求，倘若指

定时间内没有响应返回，它会认为这个包已丢失。域名系统（Domain Name System，DNS）就采取这样的工作方式（DNS也可以在TCP之上工作）。事实上，你也可以用UDP实现一个可靠的文件传输协议，而且很多人确实已经这样做了：网络文件系统（Network File System，NFS）、简单FTP（Trivial FTP，TFTP）和FSP（这是与FTP关系较远的一种协议）都使用了UDP（NFS的最新版本可以使用UDP或TCP）。在这些协议中，由应用程序负责可靠性。UDP不关心这一点（也就是说，应用程序必须处理丢失或乱序的包）。这会带来大量工作，但并不表示不能这样做，不过倘若要由你自己来编写这些代码，你就要仔细考虑一下，可能使用TCP会更好。

通常可以用电话系统和邮局的关系来对照解释TCP与UDP的区别。TCP就像电话系统。当你拨号时，电话会得到应答，在双方之间建立起一个连接。当你说话时，你知道另一方会以你说的顺序听到你讲的话。如果电话忙或没有人应答，你会马上发现。相反，UDP就像邮局系统。你向一个地址发送邮包。大多数信件都会到达，但有些可能会在路上丢失。信件可能以发送的顺序到达，但这一点无法保证。离接收方越远，邮件就越有可能在路上丢失或乱序到达。如果这对你来说很重要，你可以在信封上写上序号，然后要求接收方以正确的顺序排列，并发邮件来告诉你哪些信件已经到达，这样你就可以重新发送第一次没有到达的信件。不过，你和对方需要预先协商好这个协议。邮局不会为你做这件事。

电话系统和邮局都有各自的用处。尽管它们几乎都可以用于任何通信，但是在某些特定情况下，二者之间肯定有优劣之分。UDP和TCP也是这样。前面几章重点介绍TCP应用，它们比UDP应用更常见。不过，UDP也有自己的位置。在本章中，我们将看到用UDP能做什么。如果希望进一步深入，下一章会介绍UDP之上的组播。组播socket是标准UDP socket的一种相当简单的变体。

Java中UDP的实现分为两个类：DatagramPacket和DatagramSocket。DatagramPacket类将数据字节填充到UDP包中，这称为数据报（datagram），由你来解包接收的数据报。DatagramSocket可以收发UDP数据报。为发送数据，要将数据放到DatagramPacket中，使用DatagramSocket来发送这个包。要接收数据，可以从DatagramSocket中接收一个DatagramPacket对象，然后检查该包的内容。Socket本身非常简单。在UDP中，关于数据报的所有信息（包括发往的目标地址）都包含在包本身中。Socket只需要了解在哪个本地端口监听或发送。

这种职责划分与TCP使用的Socket和ServerSocket有所不同。首先，UDP没有两台主机间唯一连接的概念。一个Socket会收发所有指向指定端口的数据，而不需要知道对方是哪一个远程主机。一个DatagramSocket可以从多个独立主机收发数据。与TCP不同，这个Socket并不专用于一个连接。事实上，UDP没有任何两台主机之间连接的概念，它只

知道单个数据报。要确定由谁发送什么数据，这是应用程序的责任。其次，TCP socket把网络连接看作是流：通过从Socket得到的输入和输出流来收发数据。UDP不支持这一点，你处理的总是单个数据报包。填充在一个数据报的所有数据会以一个包的形式进行发送，这些数据作为一个组要么全部接收，要么完全丢失。一个包不一定与下一个包相关。给定两个包，没有办法确定哪个先发送哪个后发送。对于流来说，必须提供数据的有序队列，与之不同，数据报会尽可能快地蜂拥到接收方，就像一大群人挤公共汽车一样。而在有些情况下，如果公共汽车太挤了，有些包就会像一些不走运的人一样，可能被挤在外面，继续在公共汽车站上等候。

## UDP客户端

先来看一个简单的例子。类似于第8章的“用Socket从服务器读取”一节，我们将连接国家标准与技术研究院（National Institute for Standards and Technology，NIST）的daytime服务器，请求当前时间。不过，这一次将使用UDP而不是TCP。应该记得，daytime服务器在端口13监听，这个服务器会以人可读的格式发送时间，并关闭Socket。

现在来看如何通过编程使用UDP获取同样的数据。首先，在端口0打开一个Socket：

```
DatagramSocket socket = new DatagramSocket(0);
```

这与TCP socket有很大不同。你只需要指定要连接的一个本地端口。Socket并不知道远程主机或地址是什么。通过指定端口0，就是在请求Java为你随机选择一个可用的端口，就像是服务器Socket一样。

下一步是可选的，不过强烈建议你完成这一步。使用`setSoTimeout()`方法在连接上设置一个超时时间。超时时间以毫秒为单位来度量，所以下面这个语句设置Socket在10秒无响应后就会超时：

```
socket.setSoTimeout(10000);
```

超时对于UDP比TCP甚至更重要，因为TCP中会导致`IOException`异常的很多问题在UDP中只会悄无声息地失败。例如，如果远程主机未在目标端口监听，你就永远也不会收到回音。

接下来需要建立数据包。你要建立两个数据包，一个是要发送的数据包，另一个是要接收的数据包。对于daytime协议，数据包里有哪些数据并不重要，不过一定要指出要连接的远程主机和远程端口：

```
InetAddress host = InetAddress.getByName("time.nist.gov");
DatagramPacket request = new DatagramPacket(new byte[1], 1, host, 13);
```

接收服务器响应的数据包只包含一个空的byte数组。这个数组要足够大，可以包含整个个响应。如果它太小，就会悄悄地截断响应，1KB大小的空间应该足够了：

```
byte[] data = new byte[1024];
DatagramPacket response = new DatagramPacket(data, data.length);
```

现在已经准备就绪。首先在这个Socket上发送数据包，然后接收响应：

```
socket.send(request);
socket.receive(response);
```

最后，从响应中提取字节，将它们转换为可以显示给最终用户的字符串：

```
String daytime = new String(response.getData(), 0, response.getLength(),
    "US-ASCII");
System.out.println(daytime);
```

构造函数以及send()和receive()方法都可能抛出一个IOException，所以通常要把它们都包围在一个try块中。在Java 7中，DatagramSocket实现了Autocloseable，所以还可以使用try-with-resources：

```
try (DatagramSocket socket = new DatagramSocket(0)) {
  // 连接到服务器...
} catch (IOException ex) {
  System.err.println("Could not connect to time.nist.gov");
}
```

在Java 6和之前的版本中，可能要在一个finally块中显式地关闭Socket，来释放这个Socket占用的资源：

```
DatagramSocket socket = null;
try {
  socket = new DatagramSocket(0);
  // 连接到服务器...
} catch (IOException ex) {
  System.err.println(ex);
} finally {
  if (socket != null) {
    try {
      socket.close();
    } catch (IOException ex) {
      // 忽略
    }
  }
}
```

示例12-1将以上内容汇集在一起。

示例12-1：一个daytime协议客户端

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

public class DaytimeUDPClient {

  private final static int PORT = 13;
  private static final String HOSTNAME = "time.nist.gov";

  public static void main(String[] args) {
    try (DatagramSocket socket = new DatagramSocket(0)) {
      socket.setSoTimeout(10000);
      InetAddress host = InetAddress.getByName(HOSTNAME);
      DatagramPacket request = new DatagramPacket(new byte[1], 1, host , PORT);
      DatagramPacket response = new DatagramPacket(new byte[1024], 1024);
      socket.send(request);
      socket.receive(response);
      String result = new String(response.getData(), 0, response.getLength(),
                                 "US-ASCII");
      System.out.println(result);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

就像用TCP连接一样，输出基本上是一样的：

```
$ java DaytimeUDPClient
56375 13-04-11 19:55:22 50 0 0 843.6 UTC(NIST) *
```

# UDP服务器

UDP服务器几乎遵循与UDP客户端同样的模式，只不过通常在发送之前会先接收，而且不会选择要绑定的匿名端口。与TCP不同，并没有单独的DatagramServerSocket类。

例如，假设要在UDP上实现一个daytime服务器。首先在一个已知端口上打开一个数据报Socket。对于daytime协议，这个端口为13：

```
DatagramSocket socket = new DatagramSocket(13);
```

与TCP Socket类似，在UNIX系统（包括Linux和Mac OS X）上，要绑定一个小于1024的端口，必须作为root用户运行。可以使用sudo来运行程序，或者只需改为大于或等于1024的某个端口。

接下来，创建一个将接收请求的数据包。要提供一个将存储入站数据的byte数组，数组中的偏移量，以及要存储的字节数。在这里，将建立一个可以从0开始存储1024字节的数据包，

```
DatagramPacket request = new DatagramPacket(new byte[1024], 0, 1024);
```

然后接收这个数据包：

```
socket.receive(request);
```

这个调用会无限阻塞，直到一个UDP数据包到达端口13。如果有UDP数据包到达，Java就会将这个数据填充在byte数组中，receive()方法返回。

然后再创建一个响应数据包。这包括4个部分：要发送的原始数据、待发送原始数据的字节数、要发送到哪个主机，以及发送到该主机上哪个端口。在这个例子中，原始数据来自当前时间的一个String形式，主机和端口就是入站数据包的主机和端口：

```
String daytime = new Date().toString() + "\r\n";
byte[] data = daytime.getBytes("US-ASCII");
InetAddress host = request.getAddress();
int port = request.getPort();
DatagramPacket response = new DatagramPacket(data, data.length, host, port);
```

最后，通过接收数据包的同一个Socket发回响应：

```
socket.send(response);
```

示例12-2将这个序列包围在一个while循环中，并补充了日志记录和异常处理，使它能处理多个入站请求。

示例12-2：daytime协议服务器

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

public class DaytimeUDPServer {

  private final static int PORT = 13;
  private final static Logger audit = Logger.getLogger("requests");
  private final static Logger errors = Logger.getLogger("errors");

  public static void main(String[] args) {
    try (DatagramSocket socket = new DatagramSocket(PORT)) {
      while (true) {
        try {
          DatagramPacket request = new DatagramPacket(new byte[1024], 1024);
          socket.receive(request);

          String daytime = new Date().toString();
          byte[] data = daytime.getBytes("US-ASCII");
          DatagramPacket response = new DatagramPacket(data, data.length,
              request.getAddress(), request.getPort());
          socket.send(response);
```

```
            audit.info(daytime + " " + request.getAddress());
          } catch (IOException | RuntimeException ex) {
          errors.log(Level.SEVERE, ex.getMessage(), ex);
        }
      }
    } catch (IOException ex) {
      errors.log(Level.SEVERE, ex.getMessage(), ex);
    }
  }
}
```

在这个例子中可以看到，UDP服务器与TCP服务器不同，往往不是多线程的。它们通常不会对某一个客户做太多工作，而且不会阻塞来等待另一端响应，因为UDP从来不会报告错误。对于UDP服务器来说，除非为了准备响应需要做大量耗费时间的工作，否则使用一种迭代方法就可以了。

## DatagramPacket类

UDP数据报是基于IP数据报建立的，只向其底层IP数据报添加了很少的一点内容。图12-1展示了一个典型的UDP数据报。UDP首部只向IP首部添加了8字节。UDP首部包括源和目标端口号，IP首部之后所有内容的长度，以及一个可选的校验和。由于端口号以2字节无符号整数给出，因此每台主机有65 536个不同的UDP端口可以使用。它们与每台主机的65 536个不同的TCP端口截然不同。因为长度也是以2字节无符号整数给出，所以数据报中的字节数不能超过65 536减去首部的8字节。不过，这与IP首部中的数据报长度字段是冗余的，它将数据报限制为65 467到65 507字节之间（具体是多少取决于IP首部的大小）。校验和字段是可选的，应用层程序不使用这个校验和，也无法访问这个校验和。如果数据的校验和失败，那么底层网络软件会悄悄地丢弃这个数据报。发送方或接收方都不会得到通知。毕竟，UDP是不可靠的协议。

虽然UDP包中数据的理论最大长度是65 507字节，但实际上几乎总是比这少得多。在很多平台下，实际的限制往往是8192字节（8KB）。并且不要求具体实现接受总共超过576字节的数据报（包括数据和首部在内）。因此，如果某些程序依赖于发送超过8KB数据的UDP包，你要对这些程序多加小心。大多数情况下，更大的包都会被简单地截取为8KB数据。为保证最大的安全性，UDP包的数据部分应当保持为512字节或更少，尽管这种限制会对性能有负面影响（与使用更大的包相比）。TCP数据报也存在这个问题，但Socket和ServerSocket提供的是基于流的API，因此对程序员隐藏了这些细节。

在Java中，UDP数据报用DatagramPacket类的实例表示：

```
public final class DatagramPacket extends Object
```

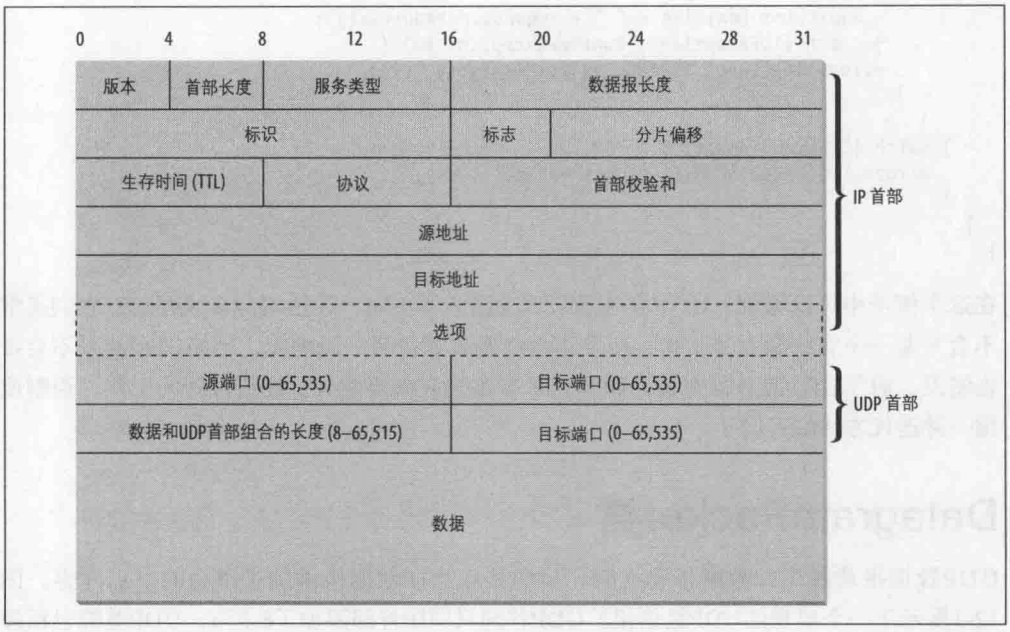


图12-1：UDP数据报结构

这个类提供了一些方法来获取和设置IP首部中的源或目标地址、获取和设置源或目标端口、获取和设置数据，以及获取和设置数据长度。其余首部字段无法通过纯Java代码访问。

## 构造函数

取决于数据包用于发送数据还是接收数据，DatagramPacket会使用不同的构造函数。这有些与众不同。正常情况下，重载构造函数是为了让你在创建对象时可以提供不同类型的信息，而不是创建将在不同上下文中使用的同一个类的对象。在这里，所有6个构造函数都接受两个参数，一个是保存数据报数据的byte数组，另一个参数是该数组中用于数据报数据的字节数。希望接收数据报时，只需要提供这两个参数。当socket从网络接收数据报时，它将数据报的数据存储在DatagramPacket对象的缓冲区数组中，直到达到你指定的长度。

第二组DatagramPacket构造函数用于创建通过网络发送的数据报。与前一组一样，这些构造函数需要一个缓冲区数组和一个长度，另外还需要指定数据包发往的地址和端口。在这里，要为构造函数传递一个byte数组，其中包含想要发送的数据，另外还要为构造函数传入目标地址及端口（数据包将发送到这个指定的地址和端口）。DatagramSocket从包中读取目标地址和端口，地址和端口不存储在Socket中，这与TCP不同。

### 接收数据报的构造函数

这两个构造函数可以创建新的DatagramPacket对象从网络接收数据：

```
public DatagramPacket(byte[] buffer, int length)
public DatagramPacket(byte[] buffer, int offset, int length)
```

使用第一个构造函数时，当Socket接收一个数据报时，它将数据报的数据部分存储在buffer，从buffer[0]开始，一直到包完全存储，或者直到向buffer写入了length字节。例如，下面的代码段创建一个新的DatagramPacket，可以接收最大为8192字节的数据报：

```
byte[] buffer = new byte[8192];
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
```

如果使用第二个构造函数，将从buffer[offset]开始存储。除此之外，这两个构造函数完全相同。length必须小于或等于buffer.length-offset。如果试图构造一个长度溢出buffer的DatagramPacket，构造函数会抛出一个IllegalArgumentException异常。这是一个RuntimeException异常，所以代码不需要捕获它。构造长度小于buffer.length-offset的DatagramPacket是可以的。在这种情况下，接收数据报时，至多只填充buffer的前length字节。

构造函数不关心缓冲区有多大，它甚至很乐于地让你创建几兆字节的DatagramPacket。不过，底层网络软件却不那么宽容，大多数底层UDP实现都不支持超过8192字节数据的数据报。IPv4数据报的理论限制是65 507字节数据，缓冲区为65 507字节的DatagramPacket可以接收任何可能的IPv4数据报，而不会丢失数据。IPv6数据报将理论限制提高到65 536字节。但实际上，很多基于UDP的协议（如DNS和TFTP）使用的包中，每数据报都仅有512字节或更少的数据。常用的最大数据大小是NFS所用的8192字节。你可能遇到的几乎所有UDP数据报都只有8KB或更少的数据。事实上，很多操作系统不支持超过8KB数据的UDP数据报，否则就会将更大的数据报截断、分解或丢掉。如果数据报太大，而导致网络将其截断或丢弃，你的Java程序将得不到任何通知（毕竟UDP是一种不可靠的协议）。因此，不要创建超过8192字节数据的DatagramPacket对象。

### 发送数据报的构造函数

下面4个构造函数会创建新的DatagramPacket对象，用来通过网络发送数据：

```
public DatagramPacket(byte[] data, int length,
    InetAddress destination, int port)
public DatagramPacket(byte[] data, int offset, int length,
    InetAddress destination, int port)
public DatagramPacket(byte[] data, int length,
    SocketAddress destination)
```

```
public DatagramPacket(byte[] data, int offset, int length,
    SocketAddress destination)
```

每个构造函数都创建一个发往另一台主机的新DatagramPacket。这个包用data数组中从offset（如果没有使用offset，则为0）开始的length个字节填充。如果试图构造一个长度大于data.length的DatagramPacket（或者大于data.length - offset），构造函数会抛出一个IllegalArgumentException异常。如果构造DatagramPacket 对象时使用的offset和length使得data数组末尾还有额外未使用的空间，这是可以的。在这种情况下，只会在网络上传送data数组的length个字节。InetAddress或SocketAddress对象的destination指向包发往的目标主机；int参数port是该主机上的端口。

## 选择数据报大小

在一个包中填充多少数据才合适，这取决于实际情况。有些协议规定了包大小。例如，用户一旦键入字符，rlogin几乎要立即将各个字符传送到远程系统。因此数据包往往很短：只包括一个字节的数据，再加上几个字节的首部。其他应用程序没有这么挑剔。例如，文件传输使用大缓冲区会更有效，唯一的要求是将文件进行分解，从而使包不大于所允许的最大包大小。

要选择最佳的包大小，涉及很多因素。如果网络非常不可靠，如分组无线电网络，则要选择较小的包，因为这样可以减少在传输中被破坏的可能。另一方面，非常快速而可靠的LAN应当使用尽可能大的包。对于很多类型的网络，8KB字节（即8192字节）往往是一个很好的折中方案。

按照惯例，在创建DatagramPacket前要将数据转换为byte数组并放在data中，但这不是绝对必需的。可以在数据报构造之后且发送之前修改data，这也能改变数据报中的数据。数据不是复制到私有缓冲区中。在一些应用程序中，可以利用这一点。例如，可以将随着时间变化的数据存储在data中，并且每分钟发送当前的数据报（及最新的数据）。不过，更重要的是，要保证数据不会在你不希望它改变的时候改变。如果你的程序是多线程的，尤其要保证这一点，不同的线程都可能会写入数据缓冲区。如果存在这种情况，就要在构造DatagramPacket之前将数据复制到一个临时缓冲区。

例如，下面的代码段将创建一个新的DatagramPacket，其中填充了UTF-8编码的数据“This is a test”。这个包被发往主机*www.ibiblio.org*的端口7（echo端口）：

```
String s = "This is a test";
byte[] data = s.getBytes("UTF-8");

try {
  InetAddress ia = InetAddress.getByName("www.ibiblio.org");
```

```
    int port = 7;
    DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
    // 发送数据包...
  } catch (IOException ex)
  }
```

大多数情况下，创建新DatagramPacket最困难的部分在于将数据转换为byte数组。由于这个代码段希望发送一个字符串，所以它使用了java.lang.String的getBytes()方法。java.io.ByteArrayOutputStream类也可以用来准备数据，从而包含在数据报中。

## get方法

DatagramPacket有6个获取数据报不同部分的方法：这些部分包括具体的数据以及首部的几个字段。这些方法主要用于从网络接收的数据报。

### public InetAddress getAddress()

getAddress()方法返回一个InetAddress对象，其中包含远程主机的地址。如果数据报是从Internet接收的，返回的地址则是发送该数据报的机器的地址（源地址）。另一方面，如果数据报是本地创建的，要发送到一个远程机器，那么这个方法会返回数据报将发往的那个主机的地址（目标地址）。这个方法常用于确定发送UDP数据报的主机地址，使接收方可以回复。

### public int getPort()

getPort()方法返回一个整数，指示远程端口。如果数据报是从Internet接收的，这就是发送包的主机上的端口。如果数据报是本地创建的，要发送到一个远程主机，那么这是远程机器上包发往的目标端口。

### public SocketAddress getSocketAddress()

getSocketAddress()方法返回一个SocketAddress对象，其中包含远程主机的IP地址和端口。与getInetAddress()情况一样，如果数据报是从Internet接收的，返回的地址就是发送该数据报的机器的地址（源地址）。另一方面，如果数据报是本地创建的，要发送到远程机器，这个方法会返回数据报发往的主机地址（目标地址）。一般要在回复前调用这个方法，确定发送UDP数据报的主机的地址和端口。与同时调用getAddress()和getPort()相比，实际结果没有显著的区别。此外，如果你使用非阻塞I/O，DatagramChannel类可以接收一个SocketAddress，而不接受单独的InetAddress和端口。

### public byte[] getData()

getData()方法返回一个byte数组，其中包含数据报中的数据。为了能够在你的程序中使用，通常必须将这些字节转换为其他的某种数据形式。一种方法是将byte数组转换为一个String。例如，给定一个从网络接收的DatagramPacket dp，可以把它转换为一个UTF-8 String，如下所示：

```
String s = new String(dp.getData(), "UTF-8");
```

如果数据报不包含文本，那么将它转换为Java数据会更为困难。一种方法是将getData()返回的byte数组转换一个ByteArrayInputStream。例如：

```
InputStream in = new ByteArrayInputStream(packet.getData(),
    packet.getOffset(), packet.getLength());
```

构造ByteArrayInputStream时，必须指定offset和length。不要使用只接受一个数组作为参数的ByteArrayInputStream()构造函数。packet.getData()返回的数组可能有额外的空间，其中未填充从网络接收的数据。这些空间可能包含任意的数据，即构造DatagramPacket时该数组相应部分中恰好包含的一些随机值。

然后ByteArrayInputStream可以串链到一个DataInputStream：

```
DataInputStream din = new DataInputStream(in);
```

接下来，可以使用DataInputStream的readInt()、readLong()、readChar()及其他方法读取数据。当然，这里假定数据报发送方使用与Java相同的数据格式。如果发送方是用Java编写的，可能就是如此，如果发送方不是用Java编写的，那么通常不会采用与Java相同的数据格式（但也不一定）。大多数现代计算机都使用与Java相同的浮点格式，大多数网络协议指定了采用网络字节序的2补码整数，这也与Java的格式相同。

### public int getLength()

getLength()方法返回数据报中数据的字节数。它不一定等于getData()返回的数组的长度（即getData().length）。getLength()返回的int可能小于getData()返回的数组的长度。

### public int getOffset()

对于getData()返回的数组，这个方法会返回该数组中的一个位置，即开始填充数据报数据的那个位置。

示例12-3使用了本节涵盖的所有方法来显示DatagramPacket中的信息。这个示例有些不太真实。因为这个程序创建了一个DatagramPacket，它已经知道其中的内容。更

常见的情况是对从网络接收的DatagramPacket使用这些方法，但这要等到下一节介绍DatagramSocket类之后再做这个工作。

示例12-3：构造接收数据的DatagramPacket

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

public class DatagramExample {

  public static void main(String[] args) {

    String s = "This is a test.";

    try {
      byte[] data = s.getBytes("UTF-8");
      InetAddress ia = InetAddress.getByName("www.ibiblio.org");
      int port = 7;
      DatagramPacket dp
          = new DatagramPacket(data, data.length, ia, port);
      System.out.println("This packet is addressed to "
         + dp.getAddress() + " on port " + dp.getPort());
      System.out.println("There are " + dp.getLength()
         + " bytes of data in the packet");
      System.out.println(
         new String(dp.getData(), dp.getOffset(), dp.getLength(), "UTF-8"));
    } catch (UnknownHostException | UnsupportedEncodingException ex) {
      System.err.println(ex);
    }
  }
}
```

下面是输出：

```
% java DatagramExample
This packet is addressed to www.ibiblio.org/152.2.254.81 on port 7
There are 15 bytes of data in the packet
This is a test.
```

## set方法

大多数情况下，6个构造函数已经足够创建数据报了。不过，Java还提供了几个方法，可以在创建数据报之后改变数据、远程地址和远程端口。如果创建和垃圾回收新DatagramPacket对象的时间会严重影响性能，这些方法就很重要。在有些情况下，重用对象比构造新对象要快得多：例如，在网络twitch游戏中，每发射一颗子弹或每移动一厘米就会发送一个数据报。不过，你可能必须使用一个非常快速的连接，这样相对于网络本身的慢速才能有比较显著的性能提升。

### public void setData(byte[] data)

setData()方法会改变UDP数据报的有效载荷（payload）。如果要向远程主机发送大文件（这里的“大”定义为“大于一个数据报所能包含的数据”），可能会用到这个方法。你可以重复地发送相同的DatagramPacket对象，每次只改变数据。

### public void setData(byte[] data, int offset, int length)

这个重载的setData()方法提供了另一个途径来发送大量的数据。与发送大量新数组不同，可以将所有数据放在一个数组中，每次发送一部分。例如，下面的循环将以512字节的块来发送一个大数组：

```
int offset = 0;
DatagramPacket dp = new DatagramPacket(bigarray, offset, 512);
int bytesSent = 0;
while (bytesSent < bigarray.length) {
  socket.send(dp);
  bytesSent += dp.getLength();
  int bytesToSend = bigarray.length - bytesSent;
  int size = (bytesToSend > 512) ? 512 : bytesToSend;
  dp.setData(bigarray, bytesSent, size);
}
```

另一方面，这个策略要求数据能保证最终到达，或者即使无法到达也不会带来严重后果。采用这种方法时，为各个包附加序号或其他可靠性标记会比较困难。

### public void setAddress(InetAddress remote)

setAddress()方法会修改数据报发往的地址。这允许你将同一个数据报发送给多个不同的接收方。例如：

```
String s = "Really Important Message";
byte[] data = s.getBytes("UTF-8");
DatagramPacket dp = new DatagramPacket(data, data.length);
dp.setPort(2000);
int network = "128.238.5.";
for (int host = 1; host < 255; host++) {
  try {
    InetAddress remote = InetAddress.getByName(network + host);
    dp.setAddress(remote);
    socket.send(dp);
  } catch (IOException ex) {
    //忽略它，继续处理下一个主机
  }
}
```

这是否是一个有意义的选择，要看具体的应用程序。如果你试图发送给一个网段上的所有工作站，就像这个代码段一样，最好使用本地广播地址，让网络完成这项工作。

将IP地址中网络和子网ID后的各个部分设置为全1，就确定了本地广播地址。例如，Polytechnic大学的网络地址为128.238.0.0。因此，它的广播地址是128.238.255.255。向128.238.255.255发送数据报会复制给该网络中的每一台主机（但有些路由器和防火墙可能会阻塞这样的数据报，这取决于发送数据报的源点）。

对于更分散的主机，可能最好使用组播（multicasting）。组播实际上使用的就是这里描述的DatagramPacket类。不过，它使用不同的IP地址和MulticastSocket来代替DatagramSocket。第13章将进一步讨论组播。

### public void setPort(int port)

setPort()方法会改变数据报发往的端口。老实说，我想不出这个方法的太多用处。它可以用于端口扫描应用程序，寻找在哪些开放的端口上运行着某个基于UDP的服务（如FSP）。此外还有另一种可能性，也许可以用于某种网络游戏或会议服务器，对于这些应用，需要接收相同信息的客户端都在不同的端口以及不同的主机上运行。在这种情况下，再次发送相同的数据报之前，可以结合setAddress()使用setPort()改变目标端口。

### public void setAddress(SocketAddress remote)

setSocketAddress()方法会改变数据报包要发往的地址和端口。在回复时可以使用这个方法。例如，下面的代码段将接收一个数据报包，用包含字符串“Hello there”的包响应同一个地址：

```
DatagramPacket input = new DatagramPacket(new byte[8192], 8192);
socket.receive(input);
DatagramPacket output = new DatagramPacket(
    "Hello there".getBytes("UTF-8"), 11);
SocketAddress address = input.getSocketAddress();
output.setAddress(address);
socket.send(output);
```

当然也可以用InetAddress对象和端口代替SocketAddress，编写相同的代码。这样的代码只是有点长。

### public void setLength(int length)

setLength()方法会改变内部缓冲区中包含实际数据报数据的字节数，而不包括未填充数据的空间。这个方法在接收数据报时很有用，本章后面还会进一步解释。当接收到数据报时，其长度设置为入站数据的长度。这表示如果试图在同一个DatagramPacket中接收另一个数据报，那么会限制第二个数据报的字节数不能多于第一个数据报的字节数。也就是说，一旦接收了一个10字节的数据报，所有后续的数据报都将截断为10字节。一

旦接收到一个9字节数据报，所有后续的数据报都将截断到9字节。依此类推。利用这个方法，允许你重置缓冲区的长度，这样就不会截断后续的数据报。

# DatagramSocket类

要收发DatagramPacket，必须打开一个数据报Socket。在Java中，数据报Socket通过DatagramSocket类创建和访问：

```
public class DatagramSocket extends Object
```

所有数据报Socket都绑定到一个本地端口，在这个端口上监听入站数据，这个端口也会放置在出站数据报的首部中。如果要编写一个客户端，不需要关心本地端口，可以调用一个构造函数，让系统来分配一个未使用的端口（匿名端口）。这个端口号放置在所有出站数据报中，服务器将用它来确定响应数据报的发送地址。如果要编写一个服务器，客户端需要知道服务器在哪个端口监听入站数据报。因此，当服务器构造DatagramSocket时，要指定它监听的本地端口。不过，客户端和服务器使用的Socket是一样的：区别只在于使用匿名端口（系统分配的端口）还是已知端口。客户端Socket和服务器Socket没有区别，这与TCP中不同。不存在诸如DatagramServerSocket之类的东西。

## 构造函数

DatagramSocket构造函数用于不同的情况，这与DatagramPacket构造函数很类似。第一个构造函数在匿名本地端口打开一个数据报Socket。第二个构造函数在监听所有本地网络接口的已知本地端口打开一个数据报Socket。后面两个构造函数在指定网络接口的已知本地端口打开一个数据报Socket。所有构造函数都只处理本地地址和端口。远程地址和端口存储在DatagramPacket中，而不是DatagramSocket。事实上，一个DatagramSocket可以从多个远程主机和端口收发数据报。

### public DatagramSocket() throws SocketException

这个构造函数创建一个绑定到匿名端口的Socket。例如：

```
try {
  DatagramSocket client = new DatagramSocket();
  // 发送数据包...
} catch (SocketException ex) {
  System.err.println(ex);
}
```

在发起与服务器对话的客户端中可能要使用这个构造函数。在这种情况下，你不关心

Socket绑定到哪个端口，因为服务器会将其响应发送到发出数据报的那个端口。让系统分配一个端口，这意味着你不必费心去寻找一个未使用的端口。如果出于某些原因需要知道本地端口，可以用本章后面描述的getLocalPort()方法得到。

可以用同一个Socket接收服务器发回的数据报。如果Socket无法绑定到一个端口，构造函数会抛出一个SocketException异常。这个构造函数抛出异常很少见。很难想象什么情况下无法打开Socket，因为会由系统选择可用的端口。

### public DatagramSocket(int port) throws SocketException

这个构造函数创建一个在指定端口（由port参数指定）监听入站数据报的Socket。可以使用这个构造函数编写在已知端口监听的服务器。如果无法创建socket，就会抛出一个SocketException异常。这个构造函数失败有两个常见的原因：指定的端口已被占用，或者试图连接低于1024的端口但你没有足够的权限（例如在UNIX系统上你不是root用户，不管怎样，其他平台则允许任何人连接低于1024的端口）。

TCP端口和UDP端口没有任何关联。对于两个不同的程序，如果一个使用UDP而另一个使用TCP，那么它们可以使用相同的端口号。示例12-4是一个端口扫描器，它将寻找本地主机中正在使用的UDP端口。如果DatagramSocket构造函数抛出异常，就可以确定这个端口正在使用。如前所述，它只考虑1024及以上的端口，以避免UNIX的限制（即要求以root身份运行才能绑定到低于1024的端口）。不过，如果你有root权限或者在Windows平台上运行，就可以很容易地扩展这个程序，使它还能检查低于1024的端口。

示例12-4：查找本地UDP端口

```
import java.net.*;

public class UDPPortScanner {

  public static void main(String[] args) {
    for (int port = 1024; port <= 65535; port++) {
      try {
        // 如果已经有服务器在端口i运行，
        // 下一行会失败并进入catch块
        DatagramSocket server = new DatagramSocket(port);
        server.close();
      } catch (SocketException ex) {
        System.out.println("There is a server on port " + port + ".");
      }
    }
  }
}
```

下面是在我们的Linux工作站上运行这个程序得到的结果（本书大部分代码都在这个工作站上编写）：

```
% java UDPPortScanner
There is a server on port 2049.
There is a server on port 32768.
There is a server on port 32770.
There is a server on port 32771.
```

第一个端口2049是NFS服务器。30000范围内的大数字端口是RPC（远程过程调用）服务器。除了RPC，使用UDP的常见协议还包括NFS、TFTP和FSP。

扫描远程系统的UDP端口比扫描远程TCP端口要难得多。不管应用层协议是什么，总有迹象表明监听端口已经接收到了你的TCP包，但UDP没有提供这种保证。要确定一个UDP服务器在监听，就必须发送它能识别并响应的包。

### public DatagramSocket(int port, InetAddress interface) throws SocketException

这个构造函数主要用于多宿主主机，它会创建在指定端口和网络接口监听入站数据报的Socket。port参数是这个Socket监听数据报的端口。与TCP Socket一样，UNIX系统中，要在低于1024的某个端口上创建DatagramSocket，需要有root身份。address参数是匹配该主机某个网络接口的InetAddress对象。如果无法创建Socket将抛出一个SocketException异常。如果这个构造函数失败，有三个常见的原因：指定端口已经被占用，试图在UNIX系统中连接低于1024的端口但你不是root用户，或者address不是系统某个网络接口的地址。

### public DatagramSocket(SocketAddress interface) throws SocketException

这个构造函数与前一个相似，只是网络接口地址和端口由SocketAddress读取。例如，下面的代码段会创建一个只监听本地回送地址的Socket：

```
SocketAddress address = new InetSocketAddress("127.0.0.1", 9999);
DatagramSocket socket = new DatagramSocket(address);
```

### protected DatagramSocket(DatagramSocketImpl impl) throws SocketException

这个构造函数允许子类提供自己的UDP实现，而不是接受默认实现。与其他4个构造函数创建的Socket不同，这个Socket一开始没有与端口绑定。使用前必须通过bind()方法绑定到一个SocketAddress：

```
public void bind(SocketAddress addr) throws SocketException
```

可以向这个方法传递null，将Socket绑定到任何可用的地址和端口。

## 发送和接收数据报

DatagramSocket类的首要任务是发送和接收UDP数据报。一个Socket可以既发送又接收数据报。事实上，它可以同时对多台主机收发数据。

### public void send(DatagramPacket dp) throws IOException

一旦创建了DatagramPacket并构造了DatagramSocket，可以将包传递给Socket的send()方法来发送这个包。例如，假设theSocket是一个DatagramSocket对象，theOutput是一个DatagramPacket对象，可以如下使用theSocket发送theOutput：

```
theSocket.send(theOutput);
```

如果发送数据时出现问题，这个方法可能会抛出一个IOException异常。但是比起Socket或ServerSocket来说，DatagramSocket出现这种情况不太常见，因为UDP是不可靠的，这意味着，你不会只是因为包没有到达目的地而得到一个异常。如果试图发送过大的数据报（大于主机底层网络软件所支持的数据报大小），可能会得到一个IOException异常，但再次发送这个大数据报时又有可能不会再得到异常。这在很大程度上取决于OS中的底层UDP软件，还取决于这个UDP软件与Java的DatagramSocketImpl类之间的基本接口代码。如果SecurityManager不允许你与这个包所发往的主机通信，这个方法可能还会抛出一个SecurityException异常。这主要是applet和其他远程加载的代码可能存在的问题。

示例12-5是一个基于UDP的discard客户端。它从System.in读取用户输入的行，将其发送给discard服务器，这个服务器只是丢弃所有数据。每一行都填充在一个DatagramPacket中。许多比较简单的Internet协议（如discard）都同时有TCP和UDP实现。

示例12-5：UDP discard客户端

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

public class UDPDiscardClient {

  public final static int PORT = 9;

  public static void main(String[] args) {

    String hostname = args.length > 0 ? args[0] : "localhost";

    try (DatagramSocket theSocket = new DatagramSocket()) {
      InetAddress server = InetAddress.getByName(hostname);
      BufferedReader userInput
        = new BufferedReader(new InputStreamReader(System.in));
      while (true) {
        String theLine = userInput.readLine();
```

```
        if (theLine.equals(".")) break;
        byte[] data = theLine.getBytes();
        DatagramPacket theOutput
            = new DatagramPacket(data, data.length, server, PORT);
        theSocket.send(theOutput);
      } // 结束while
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

UDPDiscardClient类看起应当很熟悉。它有一个静态字段PORT，设置为discard协议的标准端口（端口9），还有一个方法main()。main()方法从命令行读取一个主机名，将这个主机名转换为InetAddress对象server。将一个BufferedReader串链到System.in，读取用户的键盘输入。接下来，构造一个DatagramSocket对象，名为theSocket。在创建这个socket之后，程序进入了一个无限while循环，使用readLine()逐行读取用户输入。不过，示例12-5很谨慎，只使用readLine()读取控制台的数据，这样能保证它能像其声称的那样工作。由于discard协议只处理原始字节，所以可以忽略字符编码问题。

在while循环中，每一行数据都用getBytes()方法转换为一个byte数组，这些字节填充到新的DatagramPacket theOutput中。最后，通过theSocket发送theOutput，循环继续执行。如果在某个时刻用户在一行中只输入一个点号（.），程序就会退出。DatagramSocket构造函数可能抛出一个SocketException异常，因此需要捕获。由于这是一个discard客户端，所以不需要考虑从服务器返回的数据。

### public void receive(DatagramPacket dp) throws IOException

这个方法从网络接收一个UDP数据报，存储在现有的DatagramPacket对象dp中。与ServerSocket类的accept()相似，这个方法会阻塞调用线程，直到数据报到达。如果程序除了等待数据报外还有其他操作，就应当在单独的线程中调用receive()。

数据报的缓冲区应当足够大，足以保存接收的数据。否则，receive()会在缓冲区中放置能保存的尽可能多的数据；其他数据会丢失。要记住，UDP数据报的数据部分最大长度为65507字节（即IP数据报的最大长度65535字节减去IP首部的20字节和UDP首部的8字节）。有些使用UDP的应用协议会进一步限制包中最大字节数。例如，NFS使用最大为8192字节的包。

如果接收数据时出现问题，receive()会抛出一个IOException异常。实际上，这极少发生，因为尽管类似丢包等问题会关闭TCP流，但是在这里，在Java看到这些问题之前它们可能已经被网络或网络栈一声不响地忽略了。

示例12-6展示了一个接收入站数据报的UDP discard服务器。只是为了好玩，它在

System.out中记录了每个数据报中的数据，这样可以查看谁在向你的discard服务器发送什么数据。

示例12-6：UDPDiscardServer

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

public class UDPDiscardServer {

  public final static int PORT = 9;
  public final static int MAX_PACKET_SIZE = 65507;

  public static void main(String[] args) {
    byte[] buffer = new byte[MAX_PACKET_SIZE];

    try (DatagramSocket server = new DatagramSocket(PORT)) {
      DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
      while (true) {
        try {
          server.receive(packet);
          String s = new String(packet.getData(), 0, packet.getLength(), "8859_1");
          System.out.println(packet.getAddress() + " at port "
              + packet.getPort() + " says " + s);
          // 重置下一个数据包的长度
          packet.setLength(buffer.length);
        } catch (IOException ex) {
          System.err.println(ex);
        }
      } // 结束while
    } catch (SocketException ex) {
      System.err.println(ex);
    }
  }
}
```

这是一个很简单的类，只有一个方法main()。它从命令行读取服务器监听的端口。如果命令行没有指定端口，就在端口9监听。然后它在该端口打开一个DatagramSocket，创建一个有65507字节缓冲区的DatagramPacket，这对于任何可能接收的包都足够大了。接下来服务器进入一个无限循环，接收数据包，并在控制台显示内容和发送数据包的主机。并没有为discard数据包规定特定的编码形式。实际上，甚至没有理由要求这些数据包必须是文本。我有些随意地选择了Latin-1 ISO 8859-1编码，这是因为这种编码形式与ASCII兼容，而且为每个字节定义了一个字符。

在接收到各个数据报时，packet的长度就会设置为这个数据报中数据的长度。因此，在循环的最后一步，包的长度要重置到最大的可能值。否则，入站包的大小会限制为前面所有包的最小大小。你可以在一台机器上运行discard客户端，连接另一台机器上的discard服务器，来验证网络是否正常。

### public void close()

调用DatagramSocket对象的close()方法将释放该Socket占用的端口。与流和TCP Socket一样，你可能需要在一个finally块中关闭数据报Socket：

```
DatagramSocket server = null
try {
  server = new DatagramSocket();
  // 使用socket...
} catch (IOException ex) {
  System.err.println(ex);
} finally {
  try {
    if (server != null) server.close();
  } catch (IOException ex) {
  }
}
```

在Java 7中，DatagramSocket实现了AutoCloseable，所以你还可以使用try-with-resources：

```
try (DatagramSocket server = new DatagramSocket()) {
  //使用socket...
}
```

DatagramSocket使用结束后要将其关闭，这绝对是个好主意。如果程序还要继续执行很长一段时间，那么关闭不再需要的Socket就尤为重要。例如，示例12-4 UDPPortScanner中的close()方法就很关键：如果程序不关闭它打开的Socket，就会长时间地占用系统上的各个UDP端口。另一方面，如果你使用完DatagramSocket后程序就会退出，就不需要显式关闭Socket，Socket会在垃圾回收时自动关闭。不过，Java不会因为你用完了所有端口或Socket就运行垃圾回收器，除非你运气好碰巧同时内存也用完了，此时才会进行垃圾回收。关闭不需要的Socket不会有坏处，这是一个很好的编程习惯。

### public int getLocalPort()

DatagramSocket的getLocalPort()方法返回一个int，表示Socket正在监听的本地端口。如果创建了一个匿名端口的DatagramSocket，希望得出为Socket分配的端口，可以使用这个方法。例如：

```
DatagramSocket ds = new DatagramSocket();
System.out.println("The socket is using port " + ds.getLocalPort());
```

### public InetAddress getLocalAddress()

DatagramSocket的getLocalAddress()方法返回一个InetAddress对象，表示Socket绑

定到的本地地址。实际中很少需要这样做。正常情况下，你可能已经知道或并不关心Socket所监听的地址。

### public SocketAddress getLocalSocketAddress()

getLocalSocketAddress()方法返回一个SocketAddress对象，这个对象包装了Socket绑定到的本地接口和端口。与getLocalAddress()一样，很难想象在现实中什么情况下会用到它。这个方法之所以存在，可能主要是为了与setLocalSocketAddress()对应。

## 管理连接

与TCP socket不同，数据报Socket不太在意与谁对话。事实上，默认情况下它们可以与任何人对话，但这通常不是你希望的。例如，applet只允许向applet主机收发数据报。NFS或FSP客户端只应接受与它对话的服务器发送的包。网络游戏应当只监听游戏玩家的数据报。利用下面5个方法，你可以选择允许收发数据报的主机，而拒绝所有其他主机的包。

### public void connect(InetAddress host, int port)

connect()方法并不真正建立TCP意义上的连接。不过，它确实指定了DatagramSocket只对指定远程主机和指定远程端口收发数据包。试图向另外的主机或端口发送包将抛出一个IllegalArgumentException异常。从其他的主机或其他的端口接收的包将被丢弃，没有异常，也没有通知。

调用connect()方法时要进行安全性检查。如果VM允许向该主机和端口发送数据，会悄悄地通过检查。否则将抛出一个SecurityException异常。不过，一旦连接已经建立，该DatagramSocket的send()和receive()就不再进行正常情况下的安全性检查。

### public void disconnect()

disconnect()方法中断已连接DatagramSocket的“连接”，从而可以再次收发任何主机和端口的包。

### public int getPort()

当且仅当DatagramSocket已连接时，getPort()方法返回它所连接的远程端口。否则返回 −1。

### public InetAddress getInetAddress()

当且仅当DatagramSocket已连接时，getInetAddress()方法返回它所连接的远程主机的地址。否则返回null。

### public InetAddress getRemoteSocketAddress()

如果DatagramSocket已连接，getRemoteSocketAddress()返回它所连接的远程主机的地址。否则返回null。

## Socket选项

Java支持6个UDP Socket选项：

- SO_TIMEOUT
- SO_RCVBUF
- SO_SNDBUF
- SO_REUSEADDR
- SO_BROADCAST
- IP_TOS

## SO_TIMEOUT

SO_TIMEOUT是receive()在抛出InterruptedIOException（IOException的一个子类）异常前等待入站数据报的时间，以毫秒计。它的值必须是非负数。如果SO_TIMEOUT为0，receive()就永远不会超时。这个值可以用setSoTimeout()方法改变，用getSoTimeout()方法查看：

```
public void setSoTimeout(int timeout) throws SocketException
public int getSoTimeout() throws IOException
```

默认值是永远不超时，事实上有几种情况需要设置SO_TIMEOUT。如果要实现一个安全协议，要求在一定时间内响应，可能就需要它。如果在一定时间内没有接收到响应，就可以确定与你通信的主机已经死机（不可到达或无法响应）。

setSoTimeout()方法为数据报Socket设置SO_TIMEOUT字段。如果时间超时，阻塞的receive()方法就会抛出一个SocketTimeoutException异常。要在调用receive()之前设置这个选项。在receive()等待数据报时不能修改这个选项。timeout参数必须大于或等于0。例如：

```
try {
  byte[] buffer = new byte[2056];
  DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
  DatagramSocket ds = new DatagramSocket(2048);
  ds.setSoTimeout(30000); //阻塞不超过30秒时间
  try {
```

```
        ds.receive(dp);
        // 处理数据包...
      } catch (SocketTimeoutException ex) {
        ss.close();
        System.err.println("No connection within 30 seconds");
      }
    } catch (SocketException ex) {
      System.err.println(ex);
    } catch (IOException ex) {
      System.err.println("Unexpected IOException: " + ex);
    }
```

getSoTimeout()方法返回这个DatagramSocket对象SO_TIMEOUT字段的当前值。例如：

```
public void printSoTimeout(DatagramSocket ds) {
  int timeout = ds.getSoTimeOut();
  if (timeout > 0) {
    System.out.println(ds + " will time out after "
     + timeout + "milliseconds.");
  } else if (timeout == 0) {
    System.out.println(ds + " will never time out.");
  } else {
    System.out.println("Something is seriously wrong with " + ds);
  }
}
```

## SO_RCVBUF

DatagramSocket的SO_RCVBUF选项与Socket的SO_RCVBUF选项紧密相关。它确定了用于网络I/O的缓冲区大小。对于相当快的连接（如以太网速的连接），较大的缓冲区有助于提升性能，因为在溢出前可以存储更多的入站数据报。与TCP相比，对于UDP，足够大的接收缓冲区甚至更为重要，因为在缓冲区满时到达的UDP数据报会丢失，而缓冲区满时到达的TCP数据报最后还会重传。此外，SO_RCVBUF设置了应用程序可以接收的数据报包的最大大小。接收缓冲区中放不下的包会不声不响地被丢弃。

DatagramSocket提供了一些方法，可以获取和设置网络输入建议使用的接收缓冲区大小：

```
public void setReceiveBufferSize(int size) throws SocketException
public int getReceiveBufferSize() throws SocketException
```

setReceiveBufferSize()方法会建议对来自这个Socket的输入进行缓冲时使用的字节数。不过，底层实现完全可以忽略这个建议。例如，很多由4.3 BSD衍生的系统的最大接收缓冲区大小大约都为52KB，不允许你设置比这更高的限制值。我的Linux机器限制为64KB。其他系统可能增加到大约240KB。具体细节很大程度上依赖于平台。因此，你可能希望在设置缓冲区大小之后通过getReceiveBufferSize()查看接收缓冲区的实际大

小。getReceiveBufferSize()方法返回为来自这个socket的输入所用缓冲区的大小（字节数）。

如果底层Socket实现不能识别SO_RCVBUF选项，这两个方法都会抛出SocketException异常。这种情况可能会在非POSIX操作系统上发生。如果参数小于或等于0，setReceiveBufferSize()将抛出一个IllegalArgumentException异常。

## SO_SNDBUF

DatagramSocket还提供了一些方法，可以获取和设置建议用于网络输出的发送缓冲区大小：

```
public void setSendBufferSize(int size) throws SocketException
public int getSendBufferSize() throws SocketException
```

setSendBufferSize()建议了为缓冲这个socket输出所使用的字节数。但是再次说明，操作系统完全可以忽略这个建议。因此，可能要在setSendBufferSize()之后立即调用getSendBufferSize()来查看实际的缓冲区大小，从而检查setSendBufferSize()设置的结果。

如果底层网络软件不认识SO_SNDBUF选项，这两个方法都会抛出SocketException异常。如果参数小于或等于0，setSendBufferSize()将抛出IllegalArgumentException异常。

## SO_REUSEADDR

SO_REUSEADDR选项对于UDP Socket的意义与对于TCP Socket的意义有所不同。对于UDP，SO_REUSEADDR可以控制是否允许多个数据报Socket同时绑定到相同的端口和地址。如果多个Socket绑定到相同端口，接收的包将复制给绑定的所有Socket。这个选项由以下两个方法控制：

```
public void setReuseAddress(boolean on) throws SocketException
public boolean getReuseAddress() throws SocketException
```

为了可靠地设置这个选项，必须在新Socket绑定到端口之前调用setReuseAddress()。这表示必须使用接收DatagramImpl作为参数的保护构造函数来创建一个未连接状态的Socket。换句话说，这不适用于传统的DatagramSocket。重用端口最常用于组播Socket，这将在下一章讨论。数据报通道也能创建未连接状态的数据报Socket，可以配置为重用端口，本章后面还会介绍。

## SO_BROADCAST

SO_BROADCAST选项控制是否允许一个Socket向广播地址收发包，如广播地址192.168.254.255（这是本地地址为192.168.254.*的网络的本地网络广播地址）。UDP广播常用于DHCP等协议，这些协议需要与本地网络中的服务器通信，但预先不知道这些服务器的地址。这个选项由以下两个方法控制：

```
public void setBroadcast(boolean on) throws SocketException
public boolean getBroadcast() throws SocketException
```

路由器和网关一般不转发广播消息，但仍然会在本地网络中带来大量业务流。这个选项默认是打开的，不过如果你愿意，可以如下禁用：

```
socket.setBroadcast(false);
```

这个选项可以在绑定Socket后进行修改。

---

**提示：** 有些实现中，绑定到特定地址的Socket不接收广播包。换句话说，要使用DatagramPacket(int port)构造函数，而不是DatagramPacket(InetAddress address, int port)构造函数来监听广播。除了将SO_BROADCAST选项设置为true，这一点也是必要的。

---

## IP_TOS

由于业务流类型由各个IP数据包首部中的IP_TOS字段值来确定，所以它对于UDP与对于TCP同样重要。毕竟包要根据IP进行路由和区分优先级，而TCP和UDP都建立在IP基础之上。DatagramSocket的setTrafficClass()和getTrafficClass()方法与Socket中的相应方法实际上没有分别。之所以必须在这里重复出现，只是因为DatagramSocket和Socket没有共同的超类。可以使用下面两个方法查看和设置Socket的服务类型：

```
public int getTrafficClass() throws SocketException
public void setTrafficClass(int trafficClass) throws SocketException
```

业务流类型用0到255之间的整数指定。由于这个值要复制到TCP首部中的一个8位的字段，所以只使用这个int的低字节，超出这个范围的值会导致IllegalArgumentException异常。

---

**警告：** 这些选项的JavaDoc已经严重过时，还在描述一种基于位字段的服务质量机制，表示4种业务流类型：低成本、高可靠性、最大吞吐量和最小延迟。这种机制从来没有得到广泛实现，而且进入这个世纪后可能已经不再使用。

---

下面的代码段将业务流类型设置为10111000，从而设置一个Socket使用加速转发（Expedited Forwarding）：

```
DatagramSocket s = new DatagramSocket();
s.setTrafficClass(0xB8); // 二进制10111000
```

关于各个业务流类型的详细内容，参考第8章中的“IP_TOS服务类型”一节。

底层Socket实现不必考虑这些请求。有些实现可能会完全忽略这些值。特别是Android会把setTrafficClass()方法看作是一个不做任何动作的操作（noop）。如果底层网络栈无法提供请求的服务类型，Java可能会抛出一个SocketException异常，不过也可能不抛出。

# 一些有用的应用程序

在本节中，你将看到几个使用DatagramPacket和DatagramSocket的Internet服务器和客户端。其中一些与前面几章的示例很相似，因为许多Internet协议同时有TCP和UDP实现。当主机接收到一个IP包时，主机会检查IP首部来确定它是TCP数据包还是UDP数据报。如前面所述，UDP和TCP端口之间没有任何关联，TCP和UDP服务器可以毫无问题地共用相同的端口号。根据约定，如果一个服务同时有TCP和UDP实现，就使用相同的端口，不过这个约定并没有必然的技术原因。

## 简单的UDP客户端

一些Internet服务只需要知道客户端的地址和端口，它们会忽略客户端在数据报中发送的数据。daytime、quote of the day、time和chargen就是这样的4种协议。它们都以相同的方式进行响应，不管数据报中包含什么数据，事实上也不管数据报中是否确实有数据。这些协议的客户端只是向服务器发送一个UDP数据报，并读取传回的响应。因此，下面先来看一个名为UDPPoke的简单客户端，如示例12-7所示，它向指定主机和端口发送一个空UDP包，然后读取来自这个主机的响应包。

UDPPoke类有4个私有字段。bufferSize字段指定期望的返回包的大小。对于UDPPoke适用的大多数协议来说，8192字节的缓冲区就足够大了，不过可以向构造函数传递其他值来增加缓冲区大小。timeout字段指定等待响应的时间。host和port字段指定要连接的远程主机。

如果没有指定缓冲区长度，就使用8192字节。如果没有给定超时值，就使用30秒（30 000毫秒）。主机、端口和缓冲区大小还要用来构造outgoing DatagramPacket。虽

然理论上应当能够发送没有任何数据的数据报，但在一些Java实现中，由于存在着bug，会要求数据报中至少包含一个字节的数据。这里考虑的简单服务器会忽略这个数据。

一旦构造函数了UDPPoke对象，客户端就调用其poke()方法向目标发送一个空的outgoing数据报，并读取响应。开始时响应设置为null。当期望的数据报出现时，其数据复制到response字段。如果响应没有尽快到达或者根本没有到达，这个方法返回null。

main()方法只是从命令行读取要连接的主机和端口，构造一个UDPPoke对象，然后完成上述工作。这个客户端适用的大多数简单协议将返回ASCII文本，所以这个例子尝试将响应转换为一个ASCII字符串并显示。

示例12-7：UDPPoke类

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

public class UDPPoke {

  private int bufferSize; // 单位为字节
  private int timeout;    // 单位为毫秒
  private InetAddress host;
  private int port;

  public UDPPoke(InetAddress host, int port, int bufferSize, int timeout) {
    this.bufferSize = bufferSize;
    this.host = host;
    if (port < 1 || port > 65535) {
      throw new IllegalArgumentException("Port out of range");
    }

    this.port = port;
    this.timeout = timeout;
  }

  public UDPPoke(InetAddress host, int port, int bufferSize) {
    this(host, port, bufferSize, 30000);
  }

  public UDPPoke(InetAddress host, int port) {
    this(host, port, 8192, 30000);
  }

  public byte[] poke() {
    try (DatagramSocket socket = new DatagramSocket(0)) {
      DatagramPacket outgoing = new DatagramPacket(new byte[1], 1, host, port);
      socket.connect(host, port);
      socket.setSoTimeout(timeout);
      socket.send(outgoing);
      DatagramPacket incoming
          = new DatagramPacket(new byte[bufferSize], bufferSize);
      // 下一行阻塞，直到接收到响应
      socket.receive(incoming);
```

```
      int numBytes = incoming.getLength();
      byte[] response = new byte[numBytes];
      System.arraycopy(incoming.getData(), 0, response, 0, numBytes);
      return response;
    } catch (IOException ex) {
      return null;
    }
  }

  public static void main(String[] args) {
    InetAddress host;
    int port = 0;
    try {
      host = InetAddress.getByName(args[0]);
      port = Integer.parseInt(args[1]);
    } catch (RuntimeException | UnknownHostException ex) {
      System.out.println("Usage: java UDPPoke host port");
      return;
    }

    try {
      UDPPoke poker = new UDPPoke(host, port);
      byte[] response = poker.poke();
      if (response == null) {
        System.out.println("No response within allotted time");
        return;
      }
      String result = new String(response, "US-ASCII");
      System.out.println(result);
    } catch (UnsupportedEncodingException ex) {
      // 实际上不会发生
      ex.printStackTrace();
    }
  }
}
```

例如，下面通过UDP连接一个daytime服务器：

```
$ java UDPPoke rama.poly.edu 13
Sun Oct 3 13:04:22 2009
```

下面连接一个chargen服务器：

```
$ java UDPPoke rama.poly.edu 19
123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuv
```

有了这个类，UDP daytime、time、chargen和quote of the day客户端都很简单。只是time客户端稍微难一些，这只是因为需要将服务器返回的4个原始字节转换为一个java.util.Date对象。这里可以使用示例8-3中的算法来实现，如示例12-8所示。

示例12-8：一个UDP time客户端

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

```
import java.util.*;

public class UDPTimeClient {

  public final static int PORT = 37;
  public final static String DEFAULT_HOST = "time.nist.gov";

  public static void main(String[] args) {

    InetAddress host;
    try {
      if (args.length > 0) {
        host = InetAddress.getByName(args[0]);
      } else {
        host = InetAddress.getByName(DEFAULT_HOST);
      }
    } catch (RuntimeException | UnknownHostException ex) {
      System.out.println("Usage: java UDPTimeClient [host]");
      return;
    }

    UDPPoke poker = new UDPPoke(host, PORT);
    byte[] response = poker.poke();
    if (response == null) {
      System.out.println("No response within allotted time");
      return;
    } else if (response.length != 4) {
      System.out.println("Unrecognized response format");
      return;
    }

    // time协议的时间起点是1900年，
    //Java Data类的起点是1970年。
    // 利用这个数字可以在两者之间进行转换

    long differenceBetweenEpochs = 2208988800L;

    long secondsSince1900 = 0;
    for (int i = 0; i < 4; i++) {
      secondsSince1900
            = (secondsSince1900 << 8) | (response[i] & 0x000000FF);
    }

    long secondsSince1970
      = secondsSince1900 - differenceBetweenEpochs;
    long msSince1970 = secondsSince1970 * 1000;
    Date time = new Date(msSince1970);

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

## UDPServer

并不是只有客户端程序能够从可重用实现中获益。这些协议的服务器也同样能由此得到好处。它们都在指定端口等待UDP数据报，对应每个数据报，会以另一个数据报作为应答。不同服务器只是在返回的数据报内容上有所不同。示例12-9是一个简单的迭代型UDPServer类，可以对其派生子类，为不同协议提供特定的服务器。

UDPServer类有两个字段：int bufferSize和DatagramSocket Socket，后者是保护字段，从而允许子类使用。构造函数在指定的本地端口打开一个数据报Socket，来接收不超过bufferSize字节的数据报。

UDPServer扩展了Runnable，所以多个实例可以并行运行。其run()方法包含一个循环，将反复接收入站数据报，并将其传递给抽象的respond()方法来做出响应。为了实现不同类型的服务器，特定的子类要覆盖这个方法。

假设这个类要用作为其他程序的一部分，这些程序可能不只运行一个服务器，这就需要一种方法来关闭服务器。这个功能由shutDown()方法提供，它会设置一个标志。主循环每次执行时会检查这个标志，查看是否应当退出。如果没有业务流，receive()调用可能会无限阻塞下去，所以还要在这个Socket上设置一个超时时间。这样会每10秒唤醒一次，来检查是否有业务流。

UDPServer是一个非常灵活的类。对各个入站数据报做出响应时，子类可以发送零个、一个或多个数据报。如果响应一个数据包需要大量处理，respond()方法可以创建一个线程来做这个工作。不过，UDP服务器一般不会与客户端做太多交互。每个入站包都看作是独立于其他包，所以通常可以在respond()方法中直接处理，而不需要另外创建线程。

示例12-9：UDPServer类

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

public abstract class UDPServer implements Runnable {

  private final int bufferSize; // 单位为字节
  private final int port;
  private final Logger logger = Logger.getLogger(UDPServer.class.getCanonicalName());
  private volatile boolean isShutDown = false;

  public UDPServer(int port, int bufferSize) {
    this.bufferSize = bufferSize;
    this.port = port;
  }

  public UDPServer(int port) {
    this(port, 8192);
```

```
  }

  @Override
  public void run() {
    byte[] buffer = new byte[bufferSize];
    try (DatagramSocket socket = new DatagramSocket(port)) {
      socket.setSoTimeout(10000); // 每10秒检查一次是否关闭
      while (true) {
        if (isShutDown) return;
        DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);
        try {
          socket.receive(incoming);
          this.respond(socket, incoming);
        } catch (SocketTimeoutException ex) {
          if (isShutDown) return;
        } catch (IOException ex) {
          logger.log(Level.WARNING, ex.getMessage(), ex);
        }
      } // 结束while
    } catch (SocketException ex) {
      logger.log(Level.SEVERE, "Could not bind to port: " + port, ex);
    }
  }

  public abstract void respond(DatagramSocket socket, DatagramPacket request)
      throws IOException;

  public void shutDown() {
    this.isShutDown = true;
  }

}
```

要处理的最简单的协议是discard。所要做的就是编写一个`main()`方法来设置端口并启动线程。`respond()`是一个什么都不做的方法。示例12-10是一个高性能的UDP discard服务器，对入站包不做任何处理。

示例12-10：一个UDP discard服务器

```
import java.net.*;

public class FastUDPDiscardServer extends UDPServer {

  public final static int DEFAULT_PORT = 9;

  public FastUDPDiscardServer() {
    super(DEFAULT_PORT);
  }

  public static void main(String[] args) {
    UDPServer server = new FastUDPDiscardServer();
    Thread t = new Thread(server);
    t.start();
  }
```

```
  @Override
  public void respond(DatagramSocket socket, DatagramPacket request) {
  }
}
```

实现一个echo服务器也同样很简单，如示例12-11所示。与基于流的TCP echo服务器不同，不需要用多线程来处理多个客户端。

示例12-11：一个UDP echo服务器

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

public class UDPEchoServer extends UDPServer {

  public final static int DEFAULT_PORT = 7;

  public UDPEchoServer() {
    super(DEFAULT_PORT);
  }

  @Override
  public void respond(DatagramSocket socket, DatagramPacket packet)
      throws IOException {
    DatagramPacket outgoing = new DatagramPacket(packet.getData(),
        packet.getLength(), packet.getAddress(), packet.getPort());
    socket.send(outgoing);
  }

  public static void main(String[] args) {
    UDPServer server = new UDPEchoServer();
    Thread t = new Thread(server);
    t.start();
  }
}
```

## UDP Echo客户端

前面实现的UDPPoke类并不适用于所有协议。具体地，如果协议需要多个数据报，这些协议就需要有不同的实现。echo协议既有TCP实现又有UDP实现。使用TCP实现echo协议很简单；使用UDP要复杂得多，因为在UDP中没有I/O流或者连接的概念。基于TCP的echo客户端可以发送一个消息，然后在同一个连接上等待响应。不过，基于UDP的echo客户端不能保证发送的消息总能接收到。因此，它不能简单地等待响应，而需要准备异步地发送和接收数据。

不过，使用线程实现这种行为相当简单。一个线程可以处理用户输入并发送给echo服务器，同时另一个线程接收服务器的输入并显示给用户。客户端分为3个类：主要的UDPEchoClient类、SenderThread类和ReceiverThread类。

UDPEchoClient看起来应当很熟悉。它从命令行读取主机名，将其转换为一个InetAddress对象。UDPEchoClient使用这个对象和默认的端口来构造一个SenderThread对象。这个构造函数可能会抛出SocketException异常，所以必须捕获这个异常。然后启动SenderThread。SenderThread使用的DatagramSocket还要用来构造一个ReceiverThread，然后启动该线程。重要的是，发送和接收数据时使用了相同的DatagramSocket，因为echo服务器会把响应发回原先发出数据的那个端口。示例12-12展示了UDPEchoClient的代码。

示例12-12：UDPEchoClient类

```
import java.net.*;

public class UDPEchoClient {

  public final static int PORT = 7;

  public static void main(String[] args) {

    String hostname = "localhost";
    if (args.length > 0) {
      hostname = args[0];
    }

    try {
      InetAddress ia = InetAddress.getByName(hostname);
      DatagramSocket socket = new DatagramSocket();
      SenderThread sender = new SenderThread(socket, ia, PORT);
      sender.start();
      Thread receiver = new ReceiverThread(socket);
      receiver.start();
    } catch (UnknownHostException ex) {
      System.err.println(ex);
    } catch (SocketException ex) {
      System.err.println(ex);
    }
  }
}
```

SenderThread类从控制台读取输入，一次读取一行，将其发送给echo服务器。这个类如示例12-13所示。输入由System.in提供，但不同的客户端可能会提供一个选项，允许从不同的流读取输入（可能会打开一个FileInputStream，从文件读取）。这个类的字段定义了数据发往的服务器、该服务器的端口，以及完成发送的DatagramSocket，它们都在构造函数中设置。DatagramSocket连接到远程主机，确保接收的所有数据报都是由正确的服务器发送的。Internet上的其他一些服务器用超量数据“轰炸”这个特定端口的可能性很小，所以这不是个大问题。不过，最好确保接收的包来自正确的地方，这是一个好习惯，尤其是需要考虑安全性的时候更是如此。

run()方法一次处理一行用户输入。为此，BufferedReader userInput串链到System.in。通过一个无限循环读取用户输入的每一行数据。每一行数据都存储在theLine中。如果一行数据只包含一个点号（.），则表示用户结束输入，此时要退出循环。否则，使用java.lang.String的getBytes()方法将数据字节存储在数据数组中。接下来，这个数据数组放在DatagramPacket output的有效载荷部分，同时还包括服务器、端口和数据长度的有关信息。然后通过Socket将这个包发送到目的地。接下来这个线程交出执行权（yield），给其他线程运行的机会。

示例12-13：SenderThread类

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

class SenderThread extends Thread {

  private InetAddress server;
  private DatagramSocket socket;
  private int port;
  private volatile boolean stopped = false;

  SenderThread(DatagramSocket socket, InetAddress address, int port) {
    this.server = address;
    this.port = port;
    this.socket = socket;
    this.socket.connect(server, port);
  }

  public void halt() {
    this.stopped = true;
  }

  @Override
  public void run() {
    try {
      BufferedReader userInput
          = new BufferedReader(new InputStreamReader(System.in));
      while (true) {
        if (stopped) return;
        String theLine = userInput.readLine();
        if (theLine.equals(".")) break;
        byte[] data = theLine.getBytes("UTF-8");
        DatagramPacket output
            = new DatagramPacket(data, data.length, server, port);
        socket.send(output);
        Thread.yield();
      }
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

示例12-14所示的ReceiverThread类等待来自网络的数据报到达。接收到数据报时，它将被转换为一个String并在System.out上显示给用户。更高级的echo客户端还可以提供一个选项，允许将输出发送到别处。

这个类有两个字段。比较重要的是DatagramSocket theSocket，它必须是SenderThread使用的同一个DatagramSocket。数据会到达DatagramSocket所使用的端口，任何其他DatagramSocket都不允许连接到同一端口。第二个字段stopped是一个boolean字段，用来停止这个线程，而不需要调用已经废弃的stop()方法。

run()方法是一个无限循环，使用Socket的receive()方法等待入站数据报。出现一个入站数据报时，它会转换为一个与入站数据长度相等的String，并显示在System.out上。与输入线程一样，然后这个线程也会交出执行权，给其他线程执行的机会。

示例12-14：ReceiverThread类

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

class ReceiverThread extends Thread {

  private DatagramSocket socket;
  private volatile boolean stopped = false;

  ReceiverThread(DatagramSocket socket) {
    this.socket = socket;
  }

  public void halt() {
    this.stopped = true;
  }

  @Override
  public void run() {
    byte[] buffer = new byte[65507];
    while (true) {
      if (stopped) return;
      DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
      try {
        socket.receive(dp);
        String s = new String(dp.getData(), 0, dp.getLength(), "UTF-8");
        System.out.println(s);
        Thread.yield();
      } catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

你可以在一台机器上运行echo客户端，连接另一个机器上的echo服务器，来验证两台机器之间的网络是否正常。

# DatagramChannel

DatagramChannel类用于非阻塞UDP应用程序，就像SocketChannel和ServerSocketChannel用于非阻塞TCP应用程序一样。类似于SocketChannel和ServerSocketChannel，DatagramChannel是SelectableChannel的子类，可以注册到一个Selector。如果服务器中一个线程可以管理与多个不同客户端的通信，对于这种服务器，DatagramChannel就很有用。不过，UDP天生就比TCP更具异步性，因而实际效果没有那么明显。在UDP中，一个数据报Socket可以处理多个客户端的输入和输出请求。DatagramChannel类所增加的就是能够以非阻塞方式来做到这一点，这样一来，如果网络没有立即准备好收发数据，这些方法可以迅速返回。

## 使用DatagramChannel

对于UDP，DatagramChannel是一个近乎完备的候选API。在Java 6及之前版本中，仍需要使用DatagramSocket类将通道绑定到一个端口。不过在Java 7及以后版本中就不一定非得使用这个类了。也不必使用DatagramPacket。实际上，可以读/写字节缓冲区，就像对SocketChannel的操作一样。

### 打开一个Socket

java.nio.channels.DatagramChannel类没有公共构造函数。实际上，要使用open()静态方法创建一个新的DatagramChannel对象，例如：

```
DatagramChannel channel = DatagramChannel.open();
```

这个通道开始时没有绑定到任何端口。在绑定端口，需要使用Socket()方法访问该通道的对等DatagramSocket对象。例如，下面的代码会把通道绑定到端口3141：

```
SocketAddress address = new InetSocketAddress(3141);
DatagramSocket socket = channel.socket();
socket.bind(address);
```

Java 7直接为DatagramChannel增加了一个很方便的bind()方法，所以你根本不必使用DatagramSocket。例如：

```
SocketAddress address = new InetSocketAddress(3141);
channel.bind(address);
```

## 接收

receive()方法从通道读取一个数据报包，放在一个ByteBuffer中。它返回发送这个包的主机的地址：

```
public SocketAddress receive(ByteBuffer dst) throws IOException
```

如果通道是阻塞的（默认值），这个方法在读取到包之前不会返回。如果通道是非阻塞的，没有包可以读取的情况下这个方法会立即返回null。

如果数据报包的数据超出了缓冲区所能保存的数据，额外的数据会被丢弃，而没有任何通知。你不会接收到BufferOverflowException或类似的异常。这再次表明UDP是不可靠的。这个行为向系统又引入了一层不可靠性。数据可能已经从网络安全地到达，却在你自己的程序中丢失了。

使用这个方法，可以重新实现discard服务器，记录发送数据的主机以及所发送的数据。如示例12-15所示。这里使用了一个足够大的缓冲区（可以保存任何UDP包），并在再次使用前将其清空，从而避免潜在的数据丢失问题。

示例12-15：基于通道的UDPDiscardServer

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

public class UDPDiscardServerWithChannels {

  public final static int PORT = 9;
  public final static int MAX_PACKET_SIZE = 65507;

  public static void main(String[] args) {

    try {
      DatagramChannel channel = DatagramChannel.open();
      DatagramSocket socket = channel.socket();
      SocketAddress address = new InetSocketAddress(PORT);
      socket.bind(address);
      ByteBuffer buffer = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);
      while (true) {
        SocketAddress client = channel.receive(buffer);
        buffer.flip();
        System.out.print(client + " says ");
        while (buffer.hasRemaining()) System.out.write(buffer.get());
        System.out.println();
        buffer.clear();
      }
    } catch (IOException ex) {
      System.err.println(ex);
    }
```

```
  }
}
```

### 发送

send()方法将一个数据报包从ByteBuffer写入通道，要写到由第二个参数指定的地址：

```
public int send(ByteBuffer src, SocketAddress target) throws IOException
```

如果希望向多个客户端发送相同的数据，可以重用源ByteBuffer。不过不要忘记首先要将其回倒（rewind）。

send()方法返回写入的字节数。这可能是要写的缓冲区中的可用字节数，也可能是0，而不会是其他值。如果通道处于非阻塞模式，而且数据不能立即发送，就会返回0。否则，如果通道不在非阻塞模式，send()会等待返回，直到它能发送缓冲区中的全部数据。

示例12-16展示了一个基于通道的简单echo服务器。与示例12-15类似，receive()读取一个包。不过，这一次不是在System.out记录接收的包，而是向原先发送这些数据的客户端返回相同的数据。

示例12-16：基于通道的UDPEchoServer

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

public class UDPEchoServerWithChannels {

  public final static int PORT = 7;
  public final static int MAX_PACKET_SIZE = 65507;

  public static void main(String[] args) {

    try {
      DatagramChannel channel = DatagramChannel.open();
      DatagramSocket socket = channel.socket();
      SocketAddress address = new InetSocketAddress(PORT);
      socket.bind(address);
      ByteBuffer buffer = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);
      while (true) {
        SocketAddress client = channel.receive(buffer);
        buffer.flip();
        channel.send(buffer, client);
        buffer.clear();
      }
    } catch (IOException ex) {
      System.err.println(ex);
    }
```

```
    }
}
```

这个程序是迭代的、阻塞的，而且是同步的。与基于TCP相比，基于UDP的程序不太会出问题。UDP本质上是基于包的，具有不可靠和无连接的特性，这说明服务器至多要等待本地缓冲区清空。它不会等待客户端准备就绪来接收数据。一个客户端受到较慢客户端阻碍的可能性要少得多。

## 连接

一旦打开一个数据报通道，可以使用connect()方法将它连接到一个特定的远程地址：

```
SocketAddress remote = new InetSocketAddress("time.nist.gov", 37);
channel.connect(remote);
```

通道只向这个主机发送数据，或者只从这个主机接收数据。与SocketChannel的connect()方法不同，这个方法本身不在网络上收发任何包，因为UDP是一种无连接协议。它只是建立一个主机，有数据准备好可以发送时，就会向这个主机发送数据包。因此，这个方法会相当快地返回，不会在任何方面阻塞。这里不需要finishConnect()或isConnectionPending()方法。不过确实有一个isConnected()方法，当且仅当DatagramSocket连接时，它会返回true：

```
public boolean isConnected()
```

这会指出DatagramChannel是否仅限于一个主机。与SocketChannel不同，不是必须连接DatagramChannel才能传输或接收数据。

最后，还有一个disconnect()方法可以断开连接：

```
public DatagramChannel disconnect() throws IOException
```

它实际上没有关闭任何连接，因为首先就没有打开任何连接。它只是允许通道将来连接到一个不同的主机。

## 读取

除了用于特殊用途的receive()方法，DatagramChannel还有3个一般的read()方法：

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

不过，这些方法只用于已连接的通道。也就是说，在调用这些方法之前，必须调用connect()将通道连接到某个远程主机。这使得这些方法更适合客户端使用，它们很清

楚自己在与谁对话，而不适用于服务器，服务器必须同时接受多个主机的输入，正常情况下在第一个包到达之前服务器并不能预先知道会与哪些主机对话。

这3个方法都只从网络读取一个数据报包。数据报中的数据尽可能多地存储在参数ByteBuffer中。各个方法都会返回读取的字节数，或者如果通道关闭，则返回-1。如果出现以下某个原因，这个方法会返回0，包括：

- 通道是非阻塞的，而且没有就绪的包。
- 数据报包中不包含任何数据。
- 缓冲区已满。

与receive()方法一样，如果数据报包的数据太多，超出了ByteBuffer所能容纳的范围，额外的数据将被丢弃，而没有任何通知。你不会得到BufferOverflowException或类似的异常。

### 写入

很自然地，DatagramChannel有3个write方法，所有可写、散布（scattering）的通道都有这3个写入方法，它们可以用来代替send()方法：

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

不过，这些方法只能用于已连接的通道，否则它们不知道数据包要发送到哪里。各个方法都通过连接发送一个数据报包。这些方法都不能保证写入缓冲区的完整内容。幸运的是，缓冲区具有基于游标的特性，这就允许你可以很容易地反复调用这个方法，直到缓冲区完全排空（drain），数据全部发送为止，可能会使用多个数据报包。例如：

```
while (buffer.hasRemaining() && channel.write(buffer) != -1) ;
```

可以使用读取和写入方法实现一个简单的UDP echo客户端。对于客户端，在发送前很容易连接。因为包在传输中可能会丢失（始终要记住UDP是不可靠的），我们不希望在等待接收包时阻碍发送。因此，可以利用选择器和非阻塞I/O。第11章中曾介绍过如何对TCP使用选择器和非阻塞I/O，与之类似，选择器和非阻塞I/O也可以用于UDP。不过这一次不是发送文本数据，而是发送0到99的100个int。我们要显示返回的值，从而能很容易地确定哪个包丢失了。如示例12-17所示。

示例12-17：基于通道的UDP echo客户端

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

```
import java.nio.*;
import java.nio.channels.*;
import java.util.*;

public class UDPEchoClientWithChannels {

  public final static int PORT = 7;
  private final static int LIMIT = 100;

  public static void main(String[] args) {

    SocketAddress remote;
    try {
      remote = new InetSocketAddress(args[0], PORT);
    } catch (RuntimeException ex) {
      System.err.println("Usage: java UDPEchoClientWithChannels host");
      return;
    }

    try (DatagramChannel channel = DatagramChannel.open()) {
      channel.configureBlocking(false);
      channel.connect(remote);

      Selector selector = Selector.open();
      channel.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);

      ByteBuffer buffer = ByteBuffer.allocate(4);
      int n = 0;
      int numbersRead = 0;
      while (true) {
        if (numbersRead == LIMIT) break;
        // 为一个连接等待一分钟
        selector.select(60000);
        Set<SelectionKey> readyKeys = selector.selectedKeys();
        if (readyKeys.isEmpty() && n == LIMIT) {
          // 所有包已写入，看起来
          //好像不会再有更多数据从网络到达
          break;
        }
        else {
          Iterator<SelectionKey> iterator = readyKeys.iterator();
          while (iterator.hasNext()) {
          SelectionKey key = (SelectionKey) iterator.next();
          iterator.remove();
          if (key.isReadable()) {
            buffer.clear();
            channel.read(buffer);
            buffer.flip();
            int echo = buffer.getInt();
            System.out.println("Read: " + echo);
            numbersRead++;
          }
          if (key.isWritable()) {
            buffer.clear();
            buffer.putInt(n);
```

```
            buffer.flip();
            channel.write(buffer);
            System.out.println("Wrote: " + n);
            n++;
            if (n == LIMIT) {
              // 所有包已写入，切换到只读模式
              key.interestOps(SelectionKey.OP_READ);
            }
          }
        }
      }
      System.out.println("Echoed " + numbersRead + " out of " + LIMIT +
                                                          " sent");
      System.out.println("Success rate: " + 100.0 * numbersRead / LIMIT +
                                                        "%");
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

选择TCP通道和选择数据报通道之间有一个主要的区别。因为数据报通道实际上是无连接的（尽管有connect()方法），所以需要注意数据传输何时结束和关闭。在这个示例中，如果所有包都已经发送，而且从接收到最后一个包以来已经过去了一分钟，我们就假定数据传输已经结束。此时，如果还有希望接收的包尚未接收到，就认为这些包已经丢失。

一次典型运行之后生成的输出如下：

```
Wrote: 0
Read:  0
Wrote: 1
Wrote: 2
Read:  1
Wrote: 3
Read:  2
Wrote: 4
Wrote: 5
Wrote: 6
Wrote: 7
Wrote: 8
Wrote: 9
Wrote: 10
Wrote: 11
Wrote: 12
Wrote: 13
Wrote: 14
Wrote: 15
Wrote: 16
Wrote: 17
Wrote: 18
```

```
Wrote: 19
Wrote: 20
Wrote: 21
Wrote: 22
Read:  3
Wrote: 23
...
Wrote: 97
Read:  72
Wrote: 98
Read:  73
Wrote: 99
Read:  75
Read:  76
...
Read:  97
Read:  98
Read:  99
Echoed 92 out of 100 sent
Success rate: 92.0%
```

与两英里以外的一个远程服务器和7个跳步以外（根据traceroute报告）的一个远程服务器连接时，我发现分别有90%和98%的包能够返回。

## 关闭

就像常规的数据报Socket一样，应当在结束操作时关闭通道，释放它使用的端口和任何其他资源：

```
public void close() throws IOException
```

关闭已关闭的通道没有任何效果。试图向已关闭的通道写入或读取数据会抛出异常。如果不确定一个通道是否已关闭，可以通过isOpen()来检查：

```
public boolean isOpen()
```

如果通道已关闭，这会返回false，如果通道是打开的，则返回true。

与所有通道一样，在Java 7中，DatagramChannel实现了AutoCloseable，所以可以在try-with-resources语句中使用。在Java 7之前，要尽量在一个finally块中关闭通道。现在你对这个模式应该已经很熟悉了。在Java 6和之前的版本中：

```
DatagramChannel channel = null;
try {
  channel = DatagramChannel.open();
  // 使用通道...
} catch (IOException ex) {
  // 处理异常...
} finally {
  if (channel != null) {
```

```
        try {
          channel.close();
        } catch (IOException ex) {
          // 忽略
        }
      }
    }
```

在Java 7及以后版本中：

```
try (DatagramChannel channel = DatagramChannel.open()) {
  // 使用通道...
} catch (IOException ex) {
  // 处理异常...
}
```

## Socket选项// Java 7

在Java 7及以后版本中，DatagramChannel支持8个Socket选项，如表12-1所示。

表12-1：数据报Socket支持的Socket选项

| 选项 | 类型 | 常量 | 用途 |
|---|---|---|---|
| SO_SNDBUF | StandardSocketOptions.SO_SNDBUF | Integer | 用于发送数据报包的缓冲区大小 |
| SO_RCVBUF | StandardSocketOptions.SO_RCVBUF | Integer | 用于接收数据报包的缓冲区大小 |
| SO_REUSEADDR | StandardSocketOptions.SO_REUSEADDR | Boolean | 启用/禁用地址重用 |
| SO_BROADCAST | StandardSocketOptions.SO_BROADCAST | Boolean | 启用/禁用广播消息 |
| IP_TOS | StandardSocketOptions.IP_TOS | Integer | 业务流类型 |
| IP_MULTICAST_IF | StandardSocketOptions.IP_MULTICAST_IF | NetworkInterface | 用于组播的本地网络接口 |
| IP_MULTICAST_TTL | StandardSocketOptions.IP_MULTICAST_TTL | Integer | 用于组播数据报的生存时间值 |
| IP_MULTICAST_LOOP | StandardSocketOptions.IP_MULTICAST_LOOP | Boolean | 启用/禁用组播数据报的回送 |

前5个选项与“Socket选项”一节中介绍的数据报Socket的相应选项有相同的含义。后3个选项由组播Socket使用，将在第13章介绍。

这些选项可以用3个方法来检查和配置：

```
public <T> DatagramChannel setOption(SocketOption<T> name, T value)
    throws IOException
public <T> T getOption(SocketOption<T> name) throws IOException
public Set<SocketOption<?>> supportedOptions()
```

supportedOptions()会列出可用的Socket选项。getOption()方法会指出其中任意一个选项的当前值。setOption()允许你改变这些选项的值。例如，假设你希望发送一个广播消息。默认地通常会关闭SO_BROADCAST，不过可以将它打开，如下所示：

```
try (DatagramChannel channel = DatagramChannel.open()) {
  channel.setOption(StandardSocketOptions.SO_BROADCAST, true);
  // 发送广播消息...
} catch (IOException ex) {
  // 处理异常...
}
```

示例12-18打开一个通道，只是为了检查这些选项的默认值。

示例12-18：默认Socket选项值

```
import java.io.IOException;
import java.net.SocketOption;
import java.nio.channels.DatagramChannel;

public class DefaultSocketOptionValues {

  public static void main(String[] args) {
    try (DatagramChannel channel = DatagramChannel.open()) {
      for (SocketOption<?> option : channel.supportedOptions()) {
        System.out.println(option.name() + ": " + channel.getOption(option));
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

下面是我在我的Mac上得到的输出：

```
IP_MULTICAST_TTL: 1
SO_BROADCAST: false
SO_REUSEADDR: false
SO_RCVBUF: 196724
IP_MULTICAST_LOOP: true
SO_SNDBUF: 9216
IP_MULTICAST_IF: null
IP_TOS: 0
```

我的发送缓冲区比接收缓冲区小很多，这让人有些奇怪。

# 第13章

# IP组播

前面几章中的Socket都是单播（unicast）Socket：它们提供点对点（point-to-poin）的通信。单播Socket在两个明确的端点之间创建一个连接，有一个发送方和一个接收方，虽然它们可能交换角色，但在任何指定时刻很容易区别谁是谁。不过，尽管点对点通信有很多用途，但如果不是非常需要（比如，千禧年到来时，人们一对一地通信互道祝福），很多任务则需要另外的一种不同的通信模型。例如，电视台要从一个位置将数据广播到其发送范围内的每一个点。信号会到达每一台电视机，无论电视是否打开，也无论是否调到那个特定的电视台。事实上，信号甚至能到达不用天线而用分线盒的家庭以及根本没有电视机的家庭。这是广播的一个典型例子。这就造成了电磁波频谱和能量的巨大而随意的浪费。

与此不同，视频会议要将音频视频流发送给选定的一群人。Usenet新闻会投递到一个网站，分发给世界上数百万的人。DNS路由器更新从网站出发所要走的路径，并将改变通知给许多其他路由器。不过，发送方要依靠中间网站复制和中转消息，将消息传送给下游网站。发送方不是将消息直接发往最终接收它的每一个主机。这就是组播的例子，不过它们是通过TCP或UDP之上的附加应用层协议实现的。这些协议需要人们很详细的配置和干预。例如，要加入Usenet，你必须找到一个愿意向你发送新闻的网站，而且会把你发出的消息中转到世界其他地方。为了把你添加为Usenet传送源，新闻中继网站的新闻管理员必须专门将你的网站增加到他们的新闻配置文件中。不过，大多数主要操作系统以及Internet路由器中的网络软件已经取得了一些最新的进展，提供了一种新的可能性——真正的组播，这里路由器将决定如何高效地将消息转移到各个主机。具体地，初始的路由器只向靠近接收主机的路由器发送一份消息副本，然后这个路由器建立多个副本，发送给位于目的地或更靠近目的地的不同接收方。Internet组播建立

在UDP基础之上。Java中的组播要使用第12章中介绍的DatagramPacket，以及一个新的MulticastSocket类。

# 组播

组播比单播的点对点通信宽，但比广播通信窄而且目标更明确。组播将数据从一个主机发送给多个不同的主机，但不是发送给每一个人，数据只传送到加入某个特定的组播组从而表示对此感兴趣的客户端。在某种程度上，这与公开会议很相似。人们可以随意来去，如果讨论不再吸引他们，就会离开。在到达会议之前和离开会议之后，他们根本不需要处理（会议上的）信息：那些消息根本不会到达。在Internet上，这样的“公开会议”最好使用组播socket来实现，首先将数据的一个副本发送到靠近某些接收方的一个位置（或一组位置），这些接收方已经声明了对该数据感兴趣。在最好的情况下，数据只在到达关注客户端所在的本地网络时才复制：数据只通过Internet传送一次。更现实的情况是，会有多个相同的数据副本在Internet上传输。不过，通过仔细选择在哪些点上复制流，可以让网络的负担减至最小。值得高兴的是，程序员和网络管理员不用负责选择在哪些点上复制数据，甚至不用负责发送多个副本。Internet路由器会处理所有这些任务。

IP还支持广播，但广播的使用是严格受限的。协议要求只在没有替代方法时才进行广播，而且路由器会限制广播仅限于本地网络或子网，防止对整个Internet的广播。即使是很小的全局广播也会让Internet面临崩溃。要想广播高带宽数据，如音频、视频，或者甚至文本和静止图片，都是不可能的。一封发送给上百万地址的垃圾邮件就已经很糟糕了。想象一下，如果一个实时视频流复制给全球数十亿Internet用户（不管他们是否想观看），会有什么后果。

不过，在点对点通信和向全世界广播之间，还有一个中间状态。没有理由将视频流发送给对此不感兴趣的主机。我们需要一种技术，可以将数据发送给真正需要它的主机，而不会打扰世界的其余部分。一种方法是使用多个单播流。如果1000个客户端希望观看BBC现场报道，这个数据就会发送1000次。这是很低效的，因为它进行了不必要的数据复制，但相对于把数据广播给Internet上的每台主机来说，效率仍会提高几个数量级。不过，如果关注的客户端数量非常大，最终会用尽带宽或CPU能力，这是迟早的事，而且往往比我们预想得要早。

解决这个问题的另一种方法是创建静态连接树（connection tree）。这是Usenet新闻和一些会议系统采用的一种解决方案。数据由起始网站提供给其他服务器，这些服务器再将这个数据复制到另外一些服务器，最终复制给客户端。每个客户端连接到与之最近的服务器。与将所有数据通过多个单播发送给所有表示关注的客户端相比，这会更为高效，但这种机制不完善，已经快走到尽头了。新网站需要找到一个位置手工加入连接树。而

且连接树不一定在任何时候都能反映最好的拓扑结构，另外服务器仍需要维护很多指向客户端的点对点连接，并向每一个客户端发送相同的数据。如果能让Internet中的路由器动态地确定传输分布式信息的最佳路由，而且只在绝对必要时才复制数据，这样会更好。这正是引入组播（multicasting）的原因。

例如，假如从纽约组播视频，连接到某个LAN上的20个人在洛杉矶观看这个演出，那么这个数据只需要向该LAN发送一次。如果另外50个人在旧金山观看，数据流会在某处复制（比如说弗雷斯诺），并发送给这两个城市。如果还有100个人在休斯顿观看，将向那里发送另一个数据流（可能从圣路易斯发送），参见图13-1。数据只会在Internet上传送3次，而不是点对点连接所需的170次，也不是真正的广播所需的上百万次。组播是一种中间状态，介于Internet上常见的点对点通信和电视广播模型之间，但比这二者效率都高。当组播传送一个包时，它会发往组播组，发送给属于该组的每个主机。包不是像单播那样发往单个主机，也不是像广播那样发往每一台主机。这两者效率都太低。

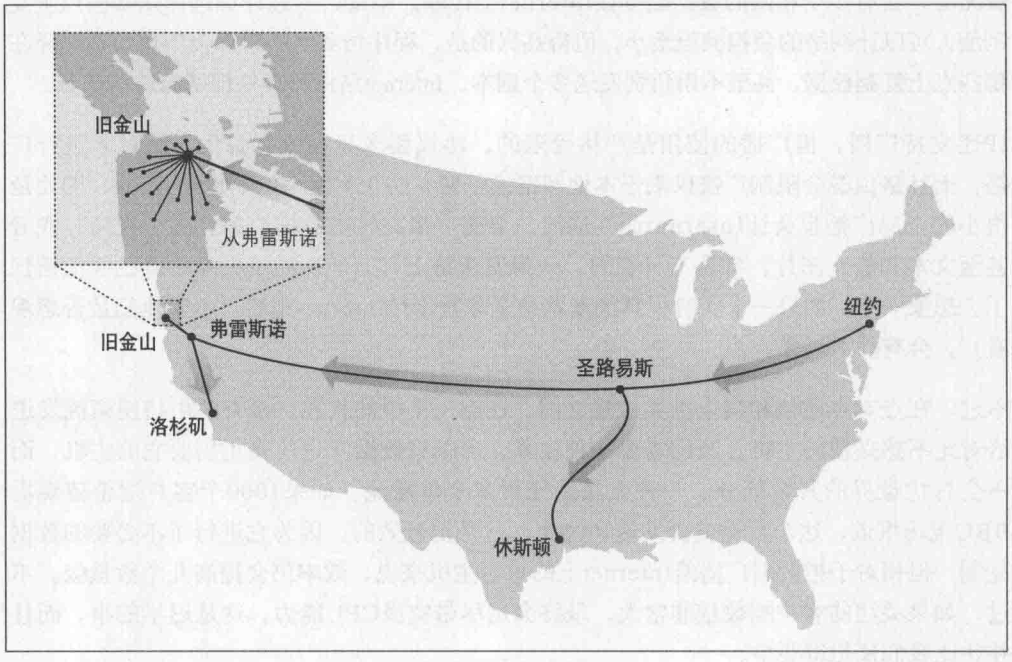


图13-1：从纽约到旧金山、洛杉矶和休斯顿的组播传送

当人们谈论组播时，首先想到的应用就是音频和视频。实际上，BBC多年前就已经在做一个同时覆盖电视和无线电广播的组播试用项目，但遗憾的是，ISP的参与程度还很有限。不过，音频和视频只是冰山的一角。还有很多其他的可能性，如多玩家游戏、分布式文件系统、大规模并行计算、多人会议、数据库复制、内容分发网络等。组播可以用

于实现命名服务和目录服务，它们不要求客户端预先了解服务器的地址。要查找一个名字，一个主机可以向一些已知地址组播传送请求，等待接收最近服务器的响应。Apple的Bonjour（也称为Zeroconf）和Apache的River都使用IP组播来动态发现本地网络中的服务。

组播设计为尽可能无缝地用于Internet。大多数工作都由路由器完成，对于应用程序员应当是透明的。应用程序只是将数据报包发送给一个组播地址，它在功能上与任何其他IP地址并没有区别。路由器将确保包被分发到该组播组中的所有主机。最大的问题是组播路由器并非普遍存在，因此，你需要了解足够的信息来查看你的网络是否支持组播。例如，尽管BBC多年前就已经开始组播新闻，但如今仍然只有大约十来个相当小的英国ISP的订阅者能够访问他们的组播流。实际中，组播更经常在单个组织的防火墙内使用，而不是在整个Internet上使用。

至于应用程序本身，需要注意数据报中称为“生存时间”（TTL）值的附加首部字段。TTL是允许数据报经过的最大路由器数目，当达到这个最大值时，即如果数据包已经经过了这么多的路由器，就会将这个包丢弃。组播使用TTL作为一种专用方法来限制包可以传输多远。例如，你可能不希望一个友好的校内游戏Dogfight的包到达世界另一端的路由器。图13-2展示了TTL如何限制包的传播。

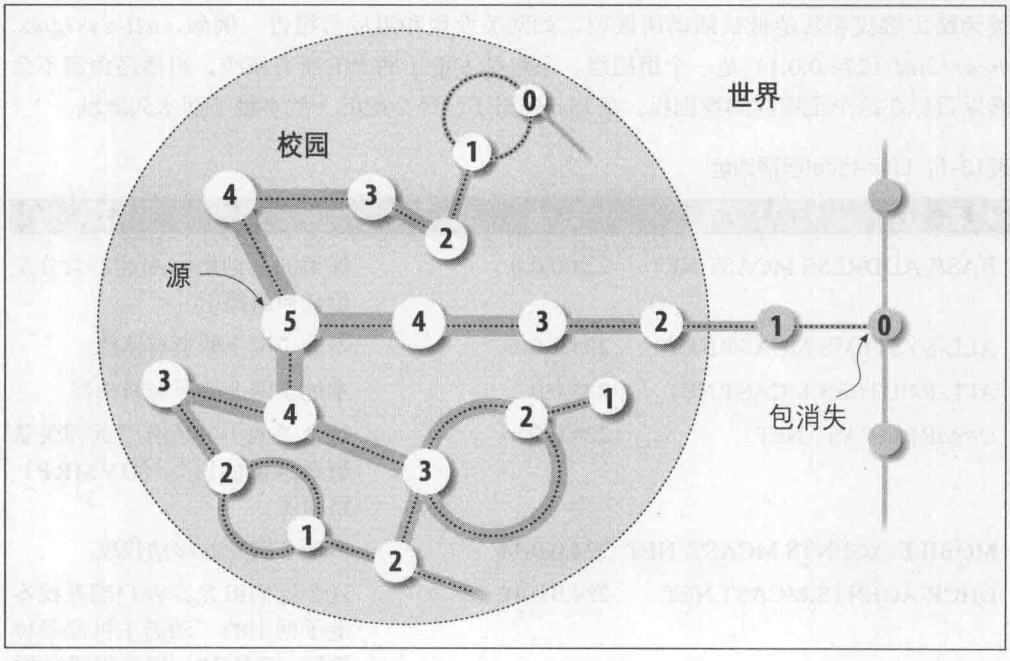


图13-2：TTL为5时包的覆盖范围

## 组播地址和组

组播地址（multicast address）是称为组播组（multicast group）的一组主机的共享地址。我们首先讨论这个地址。IPv4组播地址是CIDR组224.0.0.0/4中的IP地址（即范围在224.0.0.0到239.255.255.255之间）。这个范围内的所有地址都以二进制数字1110作为其前4位。IPv6组播地址是CIDR组ff00::/8中的IP地址（即它们都以字节0xFF开始，或者二进制表示为11111111）。

与所有IP地址类似，组播地址可以有一个主机名。例如，组播地址224.0.1.1（网络时间协议（Network Time Protocol）分布式服务的地址）就分配有主机名*ntp.mcast.net*。

组播组是一组共享一个组播地址的Internet主机。任何发送给该组播地址的数据都会中继给组中的所有成员。组播组中的成员是开放的，主机可以在任何时候进入或离开组。组可以是永久的，也可以是临时的。永久的组播组分配的地址保持不变，而不论组中是否有成员。不过，大多数组播组都是临时的，只是在有成员时才存在。要创建一个新的组播组，所要做的就是在225.0.0.0到238.255.255.255之间随机选择一个地址，为该地址构造一个InetAddress对象，开始向它发送数据。

IANA负责根据需要分发永久组播地址，到目前为止，它已经特别分配了几百个地址。Link-local组播地址以224.0.0.0开头（即从224.0.0.0到224.0.0.255之间的地址），这些地址为路由协议和其他低层活动所保留，如网关发现和组成员报告。例如，*all-systems.mcast.net*（224.0.0.1）是一个组播组，它包括本地子网上的所有系统。组播路由器不会转发目标在这个范围内的数据报。表13-1列出了已经分配的一些本地子网永久地址。

表13-1：Link-local组播地址

| 域名 | IP地址 | 用途 |
| --- | --- | --- |
| BASE-ADDRESS.MCAST.NET | 224.0.0.0 | 保留的基地址。永远不会分配给任何组播组 |
| ALL-SYSTEMS.MCAST.NET | 224.0.0.1 | 本地子网上的所有系统 |
| ALL-ROUTERS.MCAST.NET | 224.0.0.2 | 本地子网上的所有路由器 |
| DVMRP.MCAST.NET | 224.0.0.4 | 这个子网上的所有“远程矢量组播路由协议”（DVMRP）路由器 |
| MOBILE-AGENTS.MCAST. NET | 224.0.0.11 | 本地子网上的移动代理 |
| DHCP-AGENTS.MCAST.NET | 224.0.0.12 | 这个组播组允许客户端查找本地子网上的“动态主机配置协议”（DHCP）服务器或中继代理 |

表13-1：Link-local组播地址（续）

| 域名 | IP地址 | 用途 |
| --- | --- | --- |
| RSVP-ENCAPSULATION. MCAST.NET | 224.0.0.14 | 这个子网上的RSVP封装。RSVP代表资源保留设置协议（Resource reSerVation setup Protocol），允许人们为某个事件提前预留一定数量的Internet带宽 |
| VRRP.MCAST.NET | 224.0.0.18 | 虚拟路由器冗余协议（VRRP）路由器 |
| | 224.0.0.35 | DXCluster用于宣布外部业余（DX）工作站 |
| | 224.0.0.36 | 数字传输内容保护（Digital Transmission Content Protection，DTCP），这是一个数字限制管理（DRM）技术，用于对DVD播放器、电视和类似设备之间的连接进行加密 |
| | 224.0.0.37-224.0.0.68 | zeroconf寻址 |
| | 224.0.0.106 | 组播路由器发现 |
| | 224.0.0.112 | 多路径管理代理设备发现 |
| | 224.0.0.113 | Qualcomm的AllJoyn |
| | 224.0.0.114 | RFID间阅读器协议 |
| | 224.0.0.251 | 组播DNS自行分配和解析组播地址的主机名 |
| | 224.0.0.252 | Link-local组播名解析，mDNS的前身，允许节点或自分配域名严格对应本地网络，并在本地网络解析这些域名 |
| | 224.0.0.253 | Teredo用于在IPv4之上实现IPv6。同一个IPv4子网上的其他Teredo客户会响应这个组播地址 |
| | 224.0.0.254 | 为以后试验所保留 |

本地子网范围以外的永久分配组播地址以224.1.或224.2.开头，表13-2列出了其中一些永久地址。一些范围在几十到几千个地址的地址块已经为特定用途所保留。完整的列表可以从iana.org获得。剩余的24.8亿组播地址可以由任何有需要的人临时使用。组播路由器（简写为mrouter）负责确保两个不同的系统不会同时使用相同的地址。

表13-2：常见的永久组播地址

| 域名 | IP地址 | 用途 |
| --- | --- | --- |
| NTP.MCAST.NET | 224.0.1.1 | 网络时间协议 |
| NSS.MCAST.NET | 224.0.1.6 | 命名服务服务器 |
| AUDIONEWS.MCAST.NET | 224.0.1.7 | 音频新闻组播 |
| MTP.MCAST.NET | 224.0.1.9 | 组播传输协议 |
| IETF-1-LOW-AUDIO.MCAST. NET | 224.0.1.10 | IETF会议的低质量音频频道1 |
| IETF-1- AUDIO.MCAST.NET | 224.0.1.11 | IETF会议的高质量音频频道1 |
| IETF-1-VIDEO.MCAST.NET | 224.0.1.12 | IETF会议的视频频道1 |
| IETF-2-LOW-AUDIO.MCAST. NET | 224.0.1.13 | IETF会议的低质量音频频道2 |
| IETF-2-AUDIO.MCAST.NET | 224.0.1.14 | IETF会议的高质量音频频道2 |
| IETF-2-VIDEO.MCAST.NET | 224.0.1.15 | IETF会议的视频频道2 |
| MLOADD.MCAST.NET | 224.0.1.19 | MLOADD测量在几秒钟内通过一个或多个网络接口的流量负载。组播用于在被测量的不同接口之间通信 |
| EXPERIMENT.MCAST.NET | 224.0.1.20 | 试验 |
|  | 224.0.23.178 | JDP Java发现协议，用来在网络上查找可管理的JVM |
| MICROSOFT.MCAST.NET | 224.0.1.24 | Windows Internet命名服务（WINS）服务器用来相互查找 |
| MTRACE.MCAST.NET | 224.0.1.32 | traceroute的组播版本 |
| JINI-ANNOUNCEMENT.MCAST.NET | 224.0.1.84 | JINI声明 |
| JINI-REQUEST.MCAST.NET | 224.0.1.85 | JINI请求 |
|  | 224.0.1.143 | 应急管理器气象信息网络 |
|  | 224.2.0.0-224.2.255.255 | Internet上的组播主干网（MBONE）地址为多媒体会议呼叫所保留，即音频、视频、白板和多人间的共享Web浏览 |

表13-2：常见的永久组播地址（续）

| 域名 | IP地址 | 用途 |
| --- | --- | --- |
| | 224.2.2.2 | 这个地址的端口9875用于广播当前可用的MBONE节目。可以用X Window工具sdr或Windows/UNIX的multikit程序查看 |
| | 239.0.0.0-239.255.255.255 | 组织的本地范围（而不是TTL范围），使用不同范围的组播地址限制到达某个特定区域或路由器组的组播流量。例如，一个全球即插即用（UPnP）设备加入一个网络时，它会向组播地址239.255.255.250的端口1900发送HTTPU (HTTP over UDP)消息。其想法是允许预先建立可能的组成员，而不需要依赖不太可靠的TTL值 |

## 客户端和服务器

当一台主机希望向组播组发送数据时，它会将数据放在组播数据报中，组播数据报也就是发送到组播组的UDP数据报而已。组播数据通过UDP发送，虽然不可靠，但比通过面向连接的TCP发送数据要快上3倍（如果仔细考虑一下，基于TCP组播几乎是不可能的。TCP要求主机确认已经接收到包，在组播环境下处理确认将是一个噩梦）。如果你要开发不能容忍数据丢失的组播应用程序，就要由你来负责确定数据在传输中是否损坏以及如何处理丢失的数据。例如，如果要构建一个分布式缓存系统，对于没有到达的文件，可能只需要保留之前这些文件的缓存不做任何改变。

前面我说过，从应用程序员的角度看，组播和使用正常UDP Socket之间的主要区别在于，你必须考虑TTL值。这是IP首部中取值为1到255的一个字节。它可以粗略地解释为包被丢弃前可以经过的路由器数目。包每通过一个路由器，其TTL字段就至少减1。有些路由器会把TTL减2或更多。当TTL到达0时，包就被丢弃。TTL字段最初是为了防止路由循环而设计的，以保证所有包最终都会被丢弃。它可以防止配置有误的路由器相互之间无限地来回传递包。在IP组播中，TTL会在地理范围上限制组播。例如，TTL值为16时，会限制包在本地区域，一般是一个组织，或者可能是一个组织及其紧临的上游和下

游邻居。不过，TTL值为127时，包就可以全世界发送。介于这二者之间的值也是可能的。不过，没有一种精确的方法可以将TTL映射到地理距离。一般情况下，网站离得越远，包在到达之前要经过的路由器越多。TTL值小的包不会比TTL值大的包走得更远。表13-3提供了TTL值与地理范围之间关系的一些粗略估计。无论使用什么TTL值，如果包要发往224.0.0.0到224.0.0.255之间的组播组，则永远不会转发到本地子网之外。

表13-3：美国大陆发出的数据报TTL值的估计

| 目的地 | TTL值 |
| --- | --- |
| 本地主机 | 0 |
| 本地子网 | 1 |
| 本地校园网，即最近的Internet路由器的同一端，但可能位于不同的LAN | 16 |
| 同一国家的高带宽网站，一般非常靠近于主干网 | 32 |
| 同一国家的所有网站 | 48 |
| 同一大洲的所有网站 | 64 |
| 世界范围内的高带宽网站 | 128 |
| 世界范围内的所有网站 | 255 |

一旦数据填充到一个或多个数据报，发送主机将把数据报发送到Internet。这就像发送正常（单播）UDP数据一样。发送主机首先向本地网络发送一个组播数据报。这个包立即到达相同子网中组播组的所有成员。如果这个包的生存时间（TTL）字段大于1，本地网络的组播路由器会把这个包转发到包含目标组成员的其他网络。当包到达一个最终目的地时，该外部网络的组播路由器会将这个包传输到作为组播组成员的每一个主机。如果必要，组播路由器还会将包重新传输到位于当前路由器和所有最终目的地之间路径上的下一个路由器。

当数据到达组播组的一个主机时，该主机就像接收任何其他UDP数据报一样接收该数据，尽管包的目标地址与接收主机的地址不一致。主机之所以能识别出数据报是发送给它的，这是因为它属于数据报所发往的组播组，就像我们会收到地址为“某某住户”的邮件一样，尽管我们并不叫“住户”先生或“住户”女士。接收主机必须监听正确的端口，准备在数据报到达时进行处理。

## 路由器和路由

图13-3展示了一种最简单的组播配置：一个服务器向四台连接同一路由器的客户端发送相同的数据。组播socket通过Internet向客户端的路由器发出一个数据流，这个路由器复制数据流，并发送到每个客户端。如果没有组播socket，服务器就必须向路由器发出4个

单独但相同的数据流，路由器再将每个流路由到客户端。使用同一个流向多个客户端发送相同的数据大大降低了Internet主干网所需的带宽。

当然，实际的路由可能更为复杂，涉及多层冗余路由器。不过，组播Socket的目标很简单：不管网络有多复杂，在任何指定网段上，相同的数据绝不应发送多次。幸运的是，你不需要担心路由问题。只要创建一个MulticastSocket，让这个Socket加入组播组，并在要发送的DatagramPacket中填充该组播组的地址即可。路由器和MulticastSocket类会处理其余的所有工作。

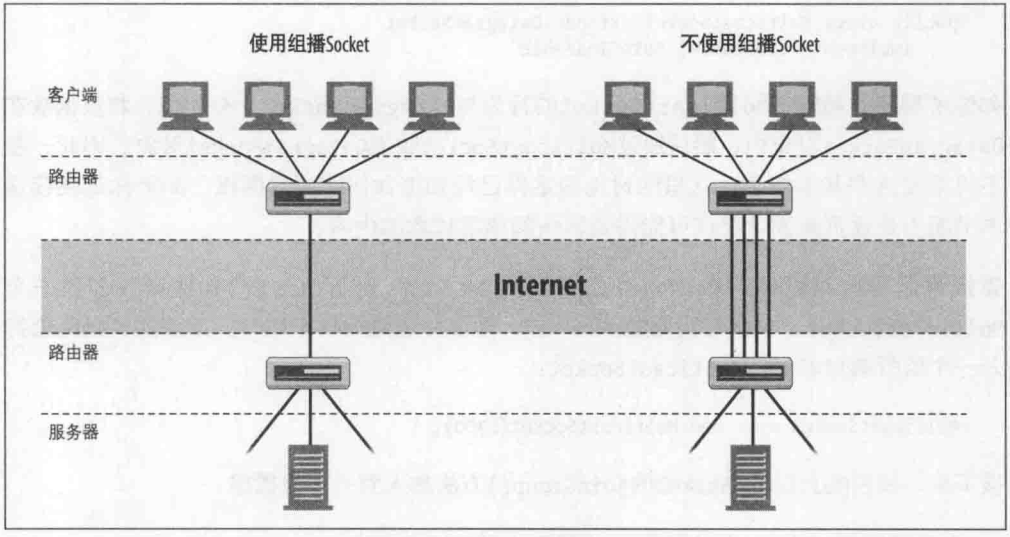


图13-3：使用与不使用组播Socket

组播最大的限制在于是否有特殊的组播路由器（mrouter）。mrouter是重新配置的Internet路由器或工作站，支持IP组播扩展。许多面向消费者的ISP故意地不在其路由器中启用组播。在2013年，仍有可能找到相互之间不存在组播路由的主机（即两台主机之间不存在只经过mrouter的路由）。

为收发本地子网之外的组播数据，你需要一个组播路由器。询问你的网络管理员，查看你的路由器是否支持组播。也可以尝试ping *all-routers.mcast.net*。如果有路由器响应，说明你的网络连接着一个组播路由器。

```
% ping all-routers.mcast.net
all-routers.mcast.net is alive
```

这仍有可能不允许你对Internet上每一个有组播能力的主机收发数据。要让你的包到达任何指定主机，你的主机和远程主机之间必须有一个由组播路由器构成的路径。或者，有

些网站可能连接着特殊的组播隧道软件，可以通过所有路由器都理解的单播UDP传输组播数据。使用本章中的示例时，如果遇到麻烦，不能生成期望的结果，请询问你的本地网络管理员或ISP，查看你的路由器是否真的支持组播。

## 使用组播Socket

理论已经讲得够多的了。在Java中，要使用java.net.MulticastSocket类来组播数据，这是java.net.DatagramSocket的一个子类：

```
public class MulticastSocket extends DatagramSocket
    implements Closeable, AutoCloseable
```

如你所期望的那样，MulticastSocket的行为与DatagramSocket十分相似：将数据放在DatagramPacket对象中，然后通过MulticastSocket收发DatagramPacket对象。因此，我不再重复这些基本概念，这里的讨论假定你已经知道如何使用数据报。如果你是跳读这本书而不是逐页阅读，现在可能需要回头阅读第12章的内容。

要接收远程网站组播的数据，首先要用MulticastSocket()构造函数创建一个MulticastSocket。与所有其他Socket一样，你要知道监听哪个端口。下面的代码段会打开一个监听端口2300的MulticastSocket：

```
MulticastSocket ms = new MulticastSocket(2300);
```

接下来，使用MulticastSocket的joinGroup()方法加入到一个组播组：

```
InetAddress group = InetAddress.getByName("224.2.2.2");
ms.joinGroup(group);
```

这会通知你与服务器之间路径上的路由器开始发送数据，并告诉本地主机要将你的IP包发往组播组。

一旦加入到组播组，就可以像DatagramSocket一样接收UDP数据。要创建一个DatagramPacket，用一个字节数组作为数据缓冲区，再进入一个循环，在循环中通过调用继承自DatagramSocket类的receive()方法接收数据：

```
byte[] buffer = new byte[8192];
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
ms.receive(dp);
```

不再希望接收数据时，通过调用这个Socket的leaveGroup()方法离开组播组。然后可以用继承自DatagramSocket的close()方法关闭该Socket：

```
ms.leaveGroup(group);
```

```
ms.close();
```

向组播地址发送数据与向单播地址发送UDP数据很相似。不需要加入组播组就可以向组播地址发送数据。可以创建一个新的DatagramPacket，在包中填充数据和组播组的地址，将这个包传入send()方法：

```
InetAddress ia = InetAddress.getByName("experiment.mcast.net");
byte[] data = "Here's some multicast data\r\n".getBytes("UTF-8");
int port = 4000;
DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
MulticastSocket ms = new MulticastSocket();
ms.send(dp);
```

对此有一个警告：组播Socket是一个很大的安全漏洞，大到足以让一辆小货车通过。因此，在SecurityManager控制下运行的不可信代码不允许做任何涉及组播Socket的操作。远程加载的代码一般只允许对其来源主机（即下载这个代码的主机）收发数据报。不过，组播Socket不允许它们收发的包有这种限制。一旦向组播Socket发送数据，对于哪些主机能接收到数据，你只能做有限的控制，而且并不可靠。因此，大多数执行远程代码的环境采取了一种保守的做法，即禁止所有的组播。

## 构造函数

构造函数很简单。你可以选择要监听的端口，或者让Java为你分配一个匿名端口：

```
public MulticastSocket() throws SocketException
public MulticastSocket(int port) throws SocketException
public MulticastSocket(SocketAddress bindAddress) throws IOException
```

例如：

```
MulticastSocket ms1 = new MulticastSocket();
MulticastSocket ms2 = new MulticastSocket(4000);
SocketAddress address = new InetSocketAddress("192.168.254.32", 4000);
MulticastSocket ms3 = new MulticastSocket(address);
```

如果无法创建Socket，所有这3个构造函数都会抛出一个SocketException异常。如果你没有足够的权限绑定到端口，或者你要绑定的端口已被占用，就无法创建Socket。需要注意，对于操作系统而言，组播Socket就是数据报Socket，所以MulticastSocket不能占用已被DatagramSocket占用的端口，反之亦然。

可以向构造函数传入null来创建一个未绑定的Socket，之后再用bind()方法进行连接。有些Socket选项只能在绑定Socket之前设置，要设置这样一些Socket选项，这个构造函数就很有用。例如，下面的代码会创建一个禁用SO_REUSEADDR的组播Socket（默认情况下，这个选项对于组播Socket一般是启用的）：

```
MulticastSocket ms = new MulticastSocket(null);
ms.setReuseAddress(false);
SocketAddress address = new InetSocketAddress(4000);
ms.bind(address);
```

## 与组播组通信

一旦创建了MulticastSocket，可以完成以下4种关键操作：

1. 加入组播组。
2. 向组中成员发送数据。
3. 接收组中的数据。
4. 离开组播组。

MulticastSocket类为操作1和4提供了方法。发送和接收数据不需要新的方法。超类DatagramSocket的send()和receive()方法就足以完成相应操作。可以以任何顺序完成这些操作，只有一个例外，必须在加入组之后才能从组接收数据。向组发送数据并不需要先加入组，另外接收和发送数据的顺序可以自由组合。

### 加入组

要加入一个组，可以将组播组的InetAddress或SocketAddress对象传递给joinGroup()方法：

```
public void joinGroup(InetAddress address) throws IOException
public void joinGroup(SocketAddress address, NetworkInterface interface)
   throws IOException
```

一旦加入组播组，接收数据报就与前一章接收单播数据报完全一样。也就是说，建立一个DatagramPacket作为缓冲区，把它传入这个socket的receive()方法。例如：

```
try {
  MulticastSocket ms = new MulticastSocket(4000);
  InetAddress ia = InetAddress.getByName("224.2.2.2");
  ms.joinGroup(ia);
  byte[] buffer = new byte[8192];
  while (true) {
    DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
    ms.receive(dp);
    String s = new String(dp.getData(), "8859_1");
    System.out.println(s);
  }
} catch (IOException ex) {
  System.err.println(ex);
}
```

如果试图加入的地址不是组播地址（即不在224.0.0.0和239.255.255.255之间），joinGroup()方法会抛出一个IOException异常。

一个MulticastSocket可以加入多个组播组。组播组中的成员信息存储在组播路由器中，而不是在对象中。在这里，你要使用存储在入站数据报中的地址来确定包要发往什么地址。

相同机器甚至相同Java程序中的多个组播Socket可以都加入相同的组。如果是这样，各个Socket会接收到发往到该组并到达本地主机的一个完整的数据副本。

第二个参数允许只加入指定本地网络接口上的组播组。例如，下面的代码段尝试加入名为“eth0”的网络接口上IP地址为224.2.2.2的组（如果存在这样一个接口）。如果不存在这样一个接口，就加入所有可用网络接口上的这个组：

```
MulticastSocket ms = new MulticastSocket();
SocketAddress group = new InetSocketAddress("224.2.2.2", 40);
NetworkInterface ni = NetworkInterface.getByName("eth0");
if (ni != null) {
  ms.joinGroup(group, ni);
} else {
  ms.joinGroup(group);
}
```

除了增加一个参数来指定要监听的网络接口，这个构造函数的行为与单参数的joinGroup()方法非常相似。例如，如果作为第一个参数传入的SocketAddress对象不表示一个组播组，就会抛出一个IOException异常。

### 离开组并且关闭连接

不再希望接收来自指定组播组的数据报时可以调用leaveGroup()方法，可以在所有网络接口上调用，也可以在指定的网络接口上调用：

```
public void leaveGroup(InetAddress address) throws IOException
public void leaveGroup(SocketAddress multicastAddress,
NetworkInterface interface)
    throws IOException
```

这会通知本地组播路由器，告诉它停止向你发送数据报。如果试图离开的地址不是一个组播地址（即不在224.0.0.0和239.255.255.255之间），这个方法会抛出一个IOException异常。不过，如果离开一个从未加入的组播组，则不会产生异常。

MulticastSocket中几乎所有方法都可以抛出一个IOException，所以通常要把所有这些代码包围在一个try块中。在Java 7中，DatagramSocket实现了Autocloseable，所以可以使用try-with-resources：

```
try (MulticastSocket socket = new MulticastSocket()) {
  // 连接到服务器...
} catch (IOException ex) {
  ex.printStackTrace();
}
```

在Java 6及之前的版本中，可能要显式地将关闭socket的代码放在一个finally块中，来释放这个Socket占用的资源：

```
MulticastSocket socket = null;
try {
  socket = new MulticastSocket();
  // 连接到服务器...
} catch (IOException ex) {
  ex.printStackTrace();
} finally {
  if (socket != null) {
    try {
      socket.close();
    } catch (IOException ex) {
      // 忽略
    }
  }
}
```

## 发送组播数据

用MulticastSocket发送数据与用DatagramSocket发送数据很相似。将数据填充在DatagramPacket中，然后使用继承自DatagramSocket的send()方法发送这个数据包。对于包所发往的组播组，数据将发送到属于该组播组中的每一个主机。例如：

```
try {
  InetAddress ia = InetAddress.getByName("experiment.mcast.net");
  byte[] data = "Here's some multicast data\r\n".getBytes();
  int port = 4000;
  DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
  MulticastSocket ms = new MulticastSocket();
  ms.send(dp);
} catch (IOException ex) {
  System.err.println(ex);
}
```

默认情况下，组播socket使用的TTL值为1（也就是说，包不会传输到本地子网之外）。不过，可以向构造函数传入0到255之间的一个整数作为第一个参数，从而改变单个数据包的TTL设置。setTimeToLive()方法设置由socket发送的包所用的默认TTL值（使用继承自DatagramSocket的send(Datagrampacket dp)方法发送，而不是MulticastSocket的send(Datagrampacket dp, byte ttl)方法）。getTimeToLive()返回MulticastSocket的默认TTL值：

```
public void setTimeToLive(int ttl) throws IOException
public int getTimeToLive() throws IOException
```

例如，下面的代码段设置TTL为64：

```
try {
  InetAddress ia = InetAddress.getByName("experiment.mcast.net");
  byte[] data = "Here's some multicast data\r\n".getBytes();
  int port = 4000;
  DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
  MulticastSocket ms = new MulticastSocket();
  ms.setTimeToLive(64);
  ms.send(dp);
} catch (IOException ex) {
  System.err.println(ex);
}
```

## 回送模式

一台主机能否接收它自己发送的组播包，也就是说组播包是否会回送，这取决于具体的平台。可以向setLoopback()传入true，指示不希望接收自己发送的包。传入false则指示确实希望接收自己发送的包：

```
public void setLoopbackMode(boolean disable) throws SocketException
public boolean getLoopbackMode() throws SocketException
```

不过，这只是一个暗示。并不要求具体实现确实按照你请求的那样去做。因为可能并非所有系统都采用同样的回送模式，所以如果你在同时收发包，就应当检查回送模式是什么，这一点很重要。如果包不回送，getLoopbackMode()将返回true，如果回送则返回false（我感觉这好像反了。我怀疑编写这个方法的程序员可能是遵循了那个愚蠢的约定，认为默认值应当总是true）。

如果系统回送包，而你不希望这样，就需要以某种方式识别出这些包并将它们丢弃。如果系统不回送而你希望回送，则要在发送包的同时手工存储你发送的包的副本，并把它们插入到内部数据结构中。可以利用setLoopback()请求你希望采用哪种回送模式（回送还是不回送），但不要指望一定能如你所愿。

## 网络接口

在多宿主主机中，setInterface()方法和setNetworkInterface()方法可以选择用于组播收发的网络接口：

```
public void setInterface(InetAddress address) throws SocketException
public InetAddress getInterface() throws SocketException
public void setNetworkInterface(NetworkInterface interface) throws
          SocketException
public NetworkInterface getNetworkInterface() throws SocketException
```

如果参数不是本地机器的网络接口地址，设置方法会抛出一个SocketException异常。对于单播Socket和DatagramSocket对象，网络接口要在构造函数中设置而且不可变，而对于MulticastSocket对象，网络接口则是可变的，可以利用一个单独的方法来设置，为什么会有这个区别原因并不清楚。为安全起见，在构造MulticastSocket之后要立即设置接口，然后就不要再改变了。下面显示了如何使用setInterface()：

```
try {
  InetAddress ia = InetAddress.getByName("www.ibiblio.org");
  MulticastSocket ms = new MulticastSocket(2048);
  ms.setInterface(ia);
  // 发送和接收数据...
} catch (UnknownHostException ue) {
  System.err.println(ue);
} catch (SocketException se) {
  System.err.println(se);
}
```

setNetworkInterface()方法与setInterface()方法的用途相同。也就是说，它会选择用于组播收发的网络接口。不过，它要基于网络接口的本地名，如“eth0”（封装在一个NetworkInterface对象中），而不是基于与该网络接口绑定的IP地址（封装在一个InetAddress对象中）。如果作为参数传递的NetworkInterface不是本地机器上的网络接口，setNetworkInterface()会抛出一个SocketException异常。

getNetworkInterface()方法返回一个NetworkInterface对象，表示这个MulticastSocket监听数据的网络接口。如果没有在构造函数或setNetworkInterface()中显式设置网络接口，它将返回一个占位（placeholder）对象（地址为“0.0.0.0”，而且索引为-1）。例如，下面的代码段将显示一个Socket所用的网络接口：

```
NetworkInterface intf = ms.getNetworkInterface();
System.out.println(intf.getName());
```

# 两个简单示例

大多数组播服务器对于与谁对话不加任何限定。因此，可以很容易地加入一个组，观察发送来的数据。示例13-1是一个MulticastSniffer类，它从命令行中读取组播组名，根据这个主机名构造一个InetAddress，并创建一个MulticastSocket，尝试加入该主机名相应的组播组。如果尝试成功，MulticastSniffer会从该Socket接收数据报，将内容显示在System.out。这个程序主要用来验证确实能够接收一个特定主机的组播数据。大多数组播数据是二进制的，显示为文本时将无法理解。

示例13-1：组播窃听器

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

```
import java.net.*;

public class MulticastSniffer {

  public static void main(String[] args) {

    InetAddress group = null;
    int port = 0;

    // 从命令行读取地址
    try {
      group = InetAddress.getByName(args[0]);
      port = Integer.parseInt(args[1]);
    } catch (ArrayIndexOutOfBoundsException | NumberFormatException
        | UnknownHostException ex) {
      System.err.println(
          "Usage: java MulticastSniffer multicast_address port");
      System.exit(1);
    }

    MulticastSocket ms = null;
    try {
      ms = new MulticastSocket(port);
      ms.joinGroup(group);

      byte[] buffer = new byte[8192];
      while (true) {
        DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
        ms.receive(dp);
        String s = new String(dp.getData(), "8859_1");
        System.out.println(s);
      }
    } catch (IOException ex) {
      System.err.println(ex);
    } finally {
      if (ms != null) {
        try {
          ms.leaveGroup(group);
          ms.close();
        } catch (IOException ex) {}
      }
    }
  }
}
```

程序开始时从前两个命令行参数读取组播组的名和端口。接下来，它在指定端口创建一个新的MulticastSocket ms。这个Socket加入指定InetAddress所在的组播组。然后进入循环，等待包到达。各个包到达时，程序读取其数据，将数据转换为ISOLatin-1编码的String，并在System.out上显示。最后，当用户中断程序或抛出异常时，Socket离开组并自行关闭。

一个全球即插即用（Universal Plug and Play，UPnP）设备加入网络时，它会向组播地

址239.255.255.250的端口1900发送一个HTTPU (HTTP over UDP)消息。可以使用这个程序监听那些消息。如果有这样一个设备在广播，你应该能在前一两分钟内看到弹出一个消息。事实上，你可能还会看到更多的内容。我在运行这个程序的前两分钟里收集到1.5MB字节的声明。这里只显示前面的两个声明：

```
$ java MulticastSniffer 239.255.255.250 1900
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age=1800
LOCATION: http://192.168.1.2:23519/Ircc.xml
NT: upnp:rootdevice
NTS: ssdp:alive
SERVER: Android/3.2 UPnP/1.0 Internet TV Box NSZ-GT1/1.0
USN: uuid:34567890-1234-1010-8000-544249cb49fd::upnp:rootdevice
X-AV-Server-Info: av=5.0; hn=""; cn="Sony Corporation";
mn="Internet TV Box NSZ-GT1"; mv="1.0";

NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age=1800
LOCATION: http://192.168.1.2:23519/Ircc.xml
NT: uuid:34567890-1234-1010-8000-544249cb49fd
NTS: ssdp:alive
SERVER: Android/3.2 UPnP/1.0 Internet TV Box NSZ-GT1/1.0
USN: uuid:34567890-1234-1010-8000-544249cb49fd
X-AV-Server-Info: av=5.0; hn=""; cn="Sony Corporation";
mn="Internet TV Box NSZ-GT1"; mv="1.0";
```

看起来我的Google TV很健谈，每秒都会发送一个声明。大多数设备只是在第一次连接到网络时声明，或者被另一个设备查询时才会声明。

现在来考虑如何发送组播数据。示例13-2是一个MulticastSender类，它从命令行读取输入，把这些数据发送给一个组播组。这个程序非常简单，但很完整。

示例13-2：MulticastSender

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

public class MulticastSender {

  public static void main(String[] args) {

    InetAddress ia = null;
    int port = 0;
    byte ttl = (byte) 1;

    // 从命令行读取地址
    try {
      ia = InetAddress.getByName(args[0]);
      port = Integer.parseInt(args[1]);
```

```
      if (args.length > 2) ttl = (byte) Integer.parseInt(args[2]);
    } catch (NumberFormatException | IndexOutOfBoundsException
        | UnknownHostException ex) {
      System.err.println(ex);
      System.err.println(
        "Usage: java MulticastSender multicast_address port ttl");
      System.exit(1);
    }

    byte[] data = "Here's some multicast data\r\n".getBytes();
    DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);

    try (MulticastSocket ms = new MulticastSocket()) {
      ms.setTimeToLive(ttl);
      ms.joinGroup(ia);
      for (int i = 1; i < 10; i++) {
        ms.send(dp);
      }
      ms.leaveGroup(ia);
    } catch (SocketException ex) {
      System.err.println(ex);
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

示例13-2从命令行读取组播组的地址、端口号和一个可选的TTL。然后使用java.lang.String的getBytes()方法将字符串“Here's some multicast data\r\n”填充到字节数组data，并把这个数组放在DatagramPacket dp中。接下来，它构造了MulticastSocket ms，并加入组ia。一旦加入到这个组，ms就向组ia发送10次数据报包dp。TTL设置为1，确保这个数据不会传输到本地子网之外。发送完数据后，ms离开组并自行关闭。

在你的本地子网中的一台机器上运行MulticastSniffer。在端口4000监听组*all-systems.mcast.net*，如下：

```
% java MulticastSniffer all-systems.mcast.net 4000
```

接下来，在本地子网的另一台机器上运行MulticastSender，从而向这个组发送数据。也可以在相同机器的不同窗口中运行，但这种做法的效果不那么强烈。不过，在开始运行MulticastSender前必须先运行MulticastSniffer。向端口4000上的组*all-systems.mcast.net*发送数据，如下：

```
% java MulticastSender all-systems.mcast.net 4000
```

回到第一台机器，应该看到下面的输出：

```
Here's some multicast data
Here's some multicast data
```

```
Here's some multicast data
Here's some multicast data
Here's some multicast data
Here's some multicast data
Here's some multicast data
Here's some multicast data
Here's some multicast data
```

要想超出本地子网，两个子网必须都有组播路由器，而且二者之间的路由器都应当支持组播。