# 【第04章-传输】

流经网络的数据总是具有相同的类型：字节。这些字节是如何流动的主要取决于我们所说的网络传输—一个帮助我们抽象底层数据传输机制的概念。用户并不关心这些细节；他们只想确保他们的字节被可靠地发送和接收。

Netty 为它所有的传输实现提供了一个通用API，这使得这种转换比你直接使用JDK所能够达到的简单得多。所产生的代码不会被实现的细节所污染，而你也不需要在你的整个代码库上进行广泛的重构。

## 4.1 案例研究：传输迁移

我们将从一个应用程序开始我们对传输的学习，这个应用程序只简单地接受连接，向客户端写“Hi!”，然后关闭连接。

### 4.1.1 不通过Netty 使用OIO 和NIO

我们将介绍仅使用了JDK API 的应用程序的阻塞（OIO）版本和异步（NIO）版本。代码清单4-1 展示了其阻塞版本的实现。

|  |
| --- |
| **// 代码清单4-1 未使用Netty 的阻塞网络编程**  **public class** PlainOioServer {  **public void** serve(**int** port) **throws** IOException {  *// 将服务器绑定到指定端口* **final** ServerSocket socket = **new** ServerSocket(port);  **try** {  **for** (; ; ) {  *// 接受连接* **final** Socket clientSocket = socket.accept();  System.***out***.println(**"Accepted connection from "** + clientSocket);  *// 创建一个新的线程来处理该连接* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  OutputStream out;  **try** {  *// 将消息写给已连接的客户端* out = clientSocket.getOutputStream();  out.write(**"Hi!\r\n"**.getBytes(Charset.*forName*(**"UTF-8"**)));  out.flush();  clientSocket.close();  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  **try** {  *// 关闭连接* clientSocket.close();  } **catch** (IOException ex) {  *// ignore on close* }  }  }  }).start(); *// 启动线程* }  } **catch** (IOException e) {  e.printStackTrace();  }  } } |

这段代码完全可以处理中等数量的并发客户端。但是随着应用程序变得流行起来，你会发现它并不能很好地伸缩到支撑成千上万的并发连入连接。你决定改用异步网络编程，但是很快就发现异步API 是完全不同的，以至于现在你不得不重写你的应用程序。其非阻塞版本如代码清单4-2 所示。

|  |
| --- |
| **// 代码清单4-2 未使用Netty 的异步网络编程**  **public class** PlainNioServer {  **public void** serve(**int** port) **throws** IOException {  ServerSocketChannel serverChannel = ServerSocketChannel.*open*();  serverChannel.configureBlocking(**false**);  ServerSocket ss = serverChannel.socket();  InetSocketAddress address = **new** InetSocketAddress(port);  *// 将服务器绑定到选定的端口* ss.bind(address);  *// 打开Selector来处理Channel* Selector selector = Selector.*open*();  *// 将ServerSocket注册到Selector 以接受连接* serverChannel.register(selector, SelectionKey.***OP\_ACCEPT***);  **final** ByteBuffer msg = ByteBuffer.*wrap*(**"Hi!\r\n"**.getBytes());  **for** (; ; ) {  **try** {  selector.select();  } **catch** (IOException ex) {  ex.printStackTrace();  *//handle exception* **break**;  }  *// 获取所有接收事件的SelectionKey 实例* Set<SelectionKey> readyKeys = selector.selectedKeys();  Iterator<SelectionKey> iterator = readyKeys.iterator();  **while** (iterator.hasNext()) {  SelectionKey key = iterator.next();  iterator.remove();  **try** {  *// 检查事件是否是一个新的已经就绪可以被接受的连接* **if** (key.isAcceptable()) {  ServerSocketChannel server = (ServerSocketChannel) key.channel();  SocketChannel client = server.accept();  client.configureBlocking(**false**);  *// 接受客户端，并将它注册到选择器* client.register(selector, SelectionKey.***OP\_WRITE*** | SelectionKey.***OP\_READ***, msg.duplicate());  System.***out***.println(**"Accepted connection from "** + client);  }   *// 检查套接字是否已经准备好写数据* **if** (key.isWritable()) {  SocketChannel client = (SocketChannel) key.channel();  ByteBuffer buffer = (ByteBuffer) key.attachment();  **while** (buffer.hasRemaining()) {  *// 将数据写到已连接的客户端* **if** (client.write(buffer) == **0**) {  **break**;  }  }   *// 关闭连接* client.close();  }  } **catch** (IOException ex) {  key.cancel();  **try** {  key.channel().close();  } **catch** (IOException cex) {  *// ignore on close* }  }  }  }  } } |

如同你所看到的，虽然这段代码所做的事情与之前的版本完全相同，但是代码却截然不同。如果为了用于非阻塞I/O 而重新实现这个简单的应用程序，都需要一次完全的重写的话，那么不难想象，移植真正复杂的应用程序需要付出什么样的努力。

鉴于此，让我们来看看使用Netty 实现该应用程序将会是什么样子吧。

### 4.1.2 通过Netty 使用OIO 和NIO

我们将先编写这个应用程序的另一个阻塞版本，这次我们将使用Netty 框架，如代码清单4-3所示。

|  |
| --- |
| **// 代码清单4-3 使用Netty 的阻塞网络处理**  **public class** NettyOioServer {  **public void** server(**int** port) **throws** Exception {  **final** ByteBuf buf = Unpooled.*unreleasableBuffer*(Unpooled.*copiedBuffer*(**"Hi!\r\n"**, Charset.*forName*(**"UTF-8"**)));  EventLoopGroup group = **new** OioEventLoopGroup();  **try** {  *// 创建ServerBootstrap* ServerBootstrap b = **new** ServerBootstrap();  b.group(group)  *// 使用OioEventLoopGroup以允许阻塞模式（旧的I/O）* .channel(OioServerSocketChannel.**class**)  .localAddress(**new** InetSocketAddress(port))  *// 指定ChannelInitializer，对于每个已接受的连接都调用它* .childHandler(**new** ChannelInitializer<SocketChannel>() {  @Override  **public void** initChannel(SocketChannel ch) **throws** Exception {  *// 添加一个ChannelInboundHandlerAdapter 以拦截和处理事件* ch.pipeline().addLast(**new** ChannelInboundHandlerAdapter() {  @Override  **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  *// 将消息写到客户端，并添加ChannelFutureListener，以便消息一被写完就关闭连接* ctx.writeAndFlush(buf.duplicate()).addListener(ChannelFutureListener.***CLOSE***);  }  });  }  });  *// 绑定服务器接受连接* ChannelFuture f = b.bind().sync();  f.channel().closeFuture().sync();  } **finally** {  *// 释放所有的资源* group.shutdownGracefully().sync();  }  } } |

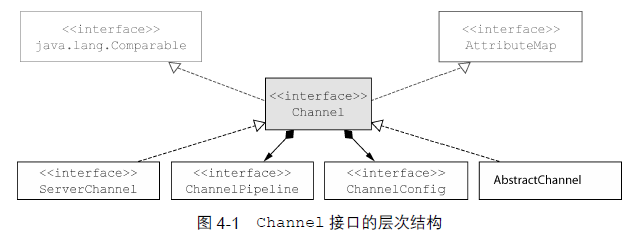
接下来，我们使用Netty 和非阻塞I/O 来实现同样的逻辑。

|  |
| --- |
| **// 代码清单4-4 使用Netty 的异步网络处理**  **public class** NettyNioServer {  **public void** server(**int** port) **throws** Exception {  **final** ByteBuf buf = Unpooled.*unreleasableBuffer*(Unpooled.*copiedBuffer*(**"Hi!\r\n"**, Charset.*forName*(**"UTF-8"**)));  *// 为非阻塞模式使用NioEventLoopGroup* NioEventLoopGroup group = **new** NioEventLoopGroup();  **try** {  *// 创建ServerBootstrap* ServerBootstrap b = **new** ServerBootstrap();  b.group(group)  *// 指定ChannelInitializer，对于每个已接受的连接都调用它* .channel(NioServerSocketChannel.**class**)  .localAddress(**new** InetSocketAddress(port))  *// 添加一个ChannelInboundHandlerAdapter 以拦截和处理事件* .childHandler(**new** ChannelInitializer<SocketChannel>() {  @Override  **public void** initChannel(SocketChannel ch) **throws** Exception {  ch.pipeline().addLast(**new** ChannelInboundHandlerAdapter() {  @Override  **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  *// 将消息写到客户端，并添加ChannelFutureListener，以便消息一被写完就关闭连接* ctx.writeAndFlush(buf.duplicate()).addListener(ChannelFutureListener.***CLOSE***);  }  });  }  });  *// 绑定服务器以接受连接* ChannelFuture f = b.bind().sync();  f.channel().closeFuture().sync();  } **finally** {  *// 释放所有的资源* group.shutdownGracefully().sync();  }  } } |

因为Netty 为每种传输的实现都暴露了相同的API，所以无论选用哪一种传输的实现，你的代码都仍然几乎不受影响。在所有的情况下，传输的实现都依赖于interface Channel、ChannelPipeline 和ChannelHandler。

## 4.2 传输API

传输API 的核心是interface Channel，它被用于所有的I/O 操作。Channel 类的层次结构如图4-1 所示。



如图所示，每个Channel 都将会被分配一个ChannelPipeline 和ChannelConfig。ChannelConfig 包含了该Channel 的所有配置设置，并且支持热更新。由于特定的传输可能具有独特的设置，所以它可能会实现一个ChannelConfig 的子类型。（请参考ChannelConfig实现对应的Javadoc。）

由于Channel 是独一无二的，所以为了保证顺序将Channel 声明为java.lang.Comparable 的一个子接口。因此，如果两个不同的Channel 实例都返回了相同的散列码，那么AbstractChannel 中的compareTo()方法的实现将会抛出一个Error。

ChannelPipeline 持有所有将应用于入站和出站数据以及事件的ChannelHandler 实例，这些ChannelHandler 实现了应用程序用于处理状态变化以及数据处理的逻辑。

ChannelHandler 的典型用途包括：

* 将数据从一种格式转换为另一种格式；
* 提供异常的通知；
* 提供Channel 变为活动的或者非活动的通知；
* 提供当Channel 注册到EventLoop 或者从EventLoop 注销时的通知；
* 提供有关用户自定义事件的通知。

拦截过滤器 ChannelPipeline 实现了一种常见的设计模式—拦截过滤器（InterceptingFilter）。UNIX 管道是另外一个熟悉的例子：多个命令被链接在一起，其中一个命令的输出端将连接到命令行中下一个命令的输入端。

你也可以根据需要通过添加或者移除ChannelHandler实例来修改ChannelPipeline。通过利用Netty的这项能力可以构建出高度灵活的应用程序。例如，每当STARTTLS协议被请求时， 你可以简单地通过向ChannelPipeline 添加一个适当的ChannelHandler（SslHandler）来按需地支持STARTTLS协议。

除了访问所分配的ChannelPipeline 和ChannelConfig 之外，也可以利用Channel的其他方法，其中最重要的列举在表4-1 中。



稍后我们将进一步深入地讨论所有这些特性的应用。目前，请记住，Netty 所提供的广泛功能只依赖于少量的接口。这意味着，你可以对你的应用程序逻辑进行重大的修改，而又无需大规模地重构你的代码库。

考虑一下写数据并将其冲刷到远程节点这样的常规任务。代码清单4-5 演示了使用Channel.writeAndFlush()来实现这一目的。

|  |
| --- |
| **// 代码清单4-5 写出到Channel**  **public static void** writingToChannel() {  Channel channel = ***CHANNEL\_FROM\_SOMEWHERE***; *// Get the channel reference from somewhere  // 创建持有要写数据的ByteBuf* ByteBuf buf = Unpooled.*copiedBuffer*(**"your data"**, CharsetUtil.***UTF\_8***);  *// 写数据并刷新* ChannelFuture cf = channel.writeAndFlush(buf);  *// 添加ChannelFutureListener 以便在写操作完成后接收通知* cf.addListener(**new** ChannelFutureListener() {  @Override  **public void** operationComplete(ChannelFuture future) {  *// 写操作完成，并且没有错误发生* **if** (future.isSuccess()) {  System.***out***.println(**"Write successful"**);  } **else** {  *// 记录错误* System.***err***.println(**"Write error"**);  future.cause().printStackTrace();  }  }  }); } |

Netty 的Channel 实现是线程安全的，因此你可以存储一个到Channel 的引用，并且每当你需要向远程节点写数据时，都可以使用它，即使当时许多线程都在使用它。代码清单4-6 展示了一个多线程写数据的简单例子。需要注意的是，消息将会被保证按顺序发送。

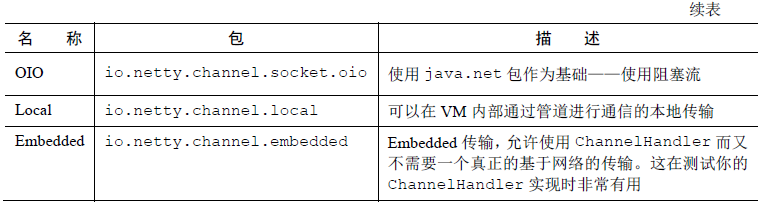
|  |
| --- |
| **// 代码清单4-6 从多个线程使用同一个Channel**  **public static void** writingToChannelFromManyThreads() {  **final** Channel channel = ***CHANNEL\_FROM\_SOMEWHERE***; *// Get the channel reference from somewhere  // 创建持有要写数据的ByteBuf* **final** ByteBuf buf = Unpooled.*copiedBuffer*(**"your data"**, CharsetUtil.***UTF\_8***);  *// 创建将数据写到Channel 的Runnable* Runnable writer = **new** Runnable() {  @Override  **public void** run() {  channel.write(buf.duplicate());  }  };  Executor executor = Executors.*newCachedThreadPool*();   *// 递交写任务给线程池以便在某个线程中执行* executor.execute(writer);   *// 递交另一个写任务以便在另一个线程中执行* executor.execute(writer);  *//...* } |

## 4.3 内置的传输

Netty 内置了一些可开箱即用的传输。因为并不是它们所有的传输都支持每一种协议，所以你必须选择一个和你的应用程序所使用的协议相容的传输。在本节中我们将讨论这些关系。

表4-2 显示了所有Netty 提供的传输。





我们将在接下来的几节中详细讨论这些传输。

## 4.3.1 NIO——非阻塞I/O

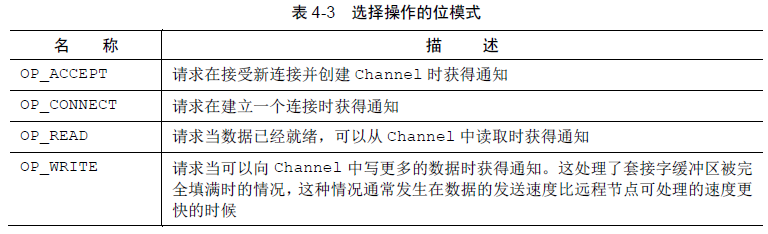
NIO 提供了一个所有I/O 操作的全异步的实现。它利用了自NIO 子系统被引入JDK 1.4 时便可用的基于选择器的API。

选择器背后的基本概念是充当一个注册表，在那里你将可以请求在Channel 的状态发生变化时得到通知。可能的状态变化有：

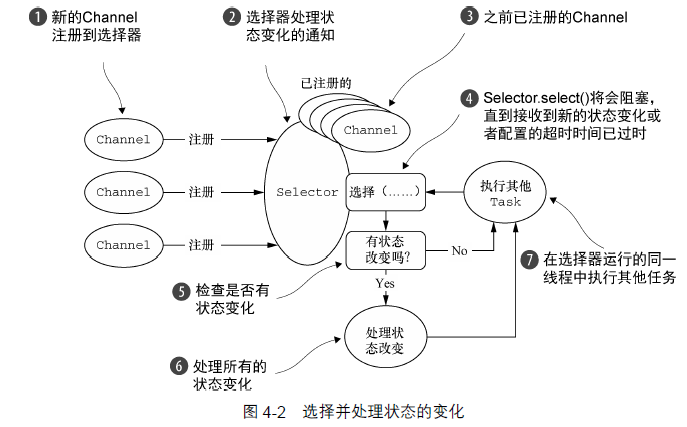
* 新的Channel 已被接受并且就绪；
* Channel 连接已经完成；
* Channel 有已经就绪的可供读取的数据；
* Channel 可用于写数据。

选择器运行在一个检查状态变化并对其做出相应响应的线程上，在应用程序对状态的改变做出响应之后，选择器将会被重置，并将重复这个过程。

表4-3 中的常量值代表了由class java.nio.channels.SelectionKey 定义的位模式。这些位模式可以组合起来定义一组应用程序正在请求通知的状态变化集。



对于所有Netty 的传输实现都共有的用户级别API 完全地隐藏了这些NIO 的内部细节。图4-2 展示了该处理流程。



零拷贝

零拷贝（zero-copy）是一种目前只有在使用NIO 和Epoll 传输时才可使用的特性。它使你可以快速高效地将数据从文件系统移动到网络接口，而不需要将其从内核空间复制到用户空间，其在像FTP 或者HTTP 这样的协议中可以显著地提升性能。但是，并不是所有的操作系统都支持这一特性。特别地，它对于实现了数据加密或者压缩的文件系统是不可用的——只能传输文件的原始内容。反过来说，传输已被加密的文件则不是问题。

### 4.3.2 Epoll—用于Linux 的本地非阻塞传输

正如我们之前所说的，Netty 的NIO 传输基于Java 提供的异步/非阻塞网络编程的通用抽象。虽然这保证了Netty 的非阻塞API 可以在任何平台上使用，但它也包含了相应的限制，因为JDK为了在所有系统上提供相同的功能，必须做出妥协。

Linux作为高性能网络编程的平台，其重要性与日俱增，这催生了大量先进特性的开发，其中包括epoll——一个高度可扩展的I/O事件通知特性。这个API自Linux内核版本2.5.44（2002）被引入，提供了比旧的POSIX select和poll系统调更好的性能，同时现在也是Linux上非阻塞网络编程的事实标准。Linux JDK NIO API使用了这些epoll调用。

Netty为Linux提供了一组NIO API，其以一种和它本身的设计更加一致的方式使用epoll，并且以一种更加轻量的方式使用中断。如果你的应用程序旨在运行于Linux系统，那么请考虑利用这个版本的传输；你将发现在高负载下它的性能要优于JDK的NIO实现。

这个传输的语义与在图4-2 所示的完全相同，而且它的用法也是简单直接的。相关示例参照代码清单4-4。如果要在那个代码清单中使用epoll 替代NIO，只需要将NioEventLoopGroup替换为EpollEventLoopGroup ， 并且将NioServerSocketChannel.class 替换为EpollServerSocketChannel.class 即可。

### 4.3.3 OIO—旧的阻塞I/O

Netty 的OIO 传输实现代表了一种折中：它可以通过常规的传输API 使用，但是由于它是建立在java.net 包的阻塞实现之上的，所以它不是异步的。但是，它仍然非常适合于某些用途。

例如，你可能需要移植使用了一些进行阻塞调用的库（如JDBC）的遗留代码，而将逻辑转换为非阻塞的可能也是不切实际的。相反，你可以在短期内使用Netty的OIO传输，然后再将你的代码移植到纯粹的异步传输上。让我们来看一看怎么做。

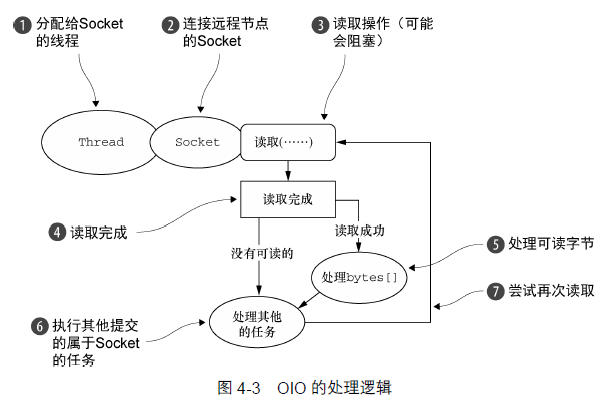
在java.net API 中，你通常会有一个用来接受到达正在监听的ServerSocket 的新连接的线程。会创建一个新的和远程节点进行交互的套接字，并且会分配一个新的用于处理相应通信流量的线程。这是必需的，因为某个指定套接字上的任何I/O 操作在任意的时间点上都可能会阻塞。使用单个线程来处理多个套接字，很容易导致一个套接字上的阻塞操作也捆绑了所有其他的套接字。

有了这个背景，你可能会想，Netty是如何能够使用和用于异步传输相同的API来支持OIO的呢。答案就是，Netty利用了SO\_TIMEOUT这个Socket标志，它指定了等待一个I/O操作完成的最大毫秒数。如果操作在指定的时间间隔内没有完成，则将会抛出一个SocketTimeout Exception（这种方式的一个问题是，当一个SocketTimeoutException 被抛出时填充栈跟踪所需要的时间，其对于性能来说代价很大。）。Netty将捕获这个异常并继续处理循环。在EventLoop下一次运行时，它将再次尝试。这实际上也是类似于Netty这样的异步框架能够支持OIO的唯一方式，图4-3 说明了这个逻辑。

### 4.3.4 用于JVM 内部通信的Local 传输

Netty 提供了一个Local 传输，用于在同一个JVM 中运行的客户端和服务器程序之间的异步通信。同样，这个传输也支持对于所有Netty 传输实现都共同的API。

在这个传输中，和服务器Channel 相关联的SocketAddress 并没有绑定物理网络地址；相反，只要服务器还在运行，它就会被存储在注册表里，并在Channel 关闭时注销。因为这个传输并不接受真正的网络流量，所以它并不能够和其他传输实现进行互操作。因此，客户端希望连接到（在同一个JVM 中）使用了这个传输的服务器端时也必须使用它。除了这个限制，它的使用方式和其他的传输一模一样。

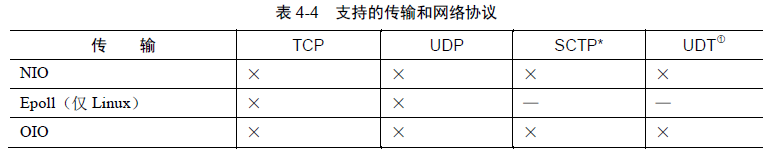


### 4.3.5 Embedded 传输

Netty 提供了一种额外的传输，使得你可以将一组ChannelHandler 作为帮助器类嵌入到其他的ChannelHandler 内部。通过这种方式，你将可以扩展一个ChannelHandler 的功能，而又不需要修改其内部代码。不足为奇的是，Embedded 传输的关键是一个被称为EmbeddedChannel 的具体的Channel实现。

## 4.4 传输的用例

既然我们已经详细地了解了所有的传输，那么让我们考虑一下选用一个适用于特定用途的协议的因素吧。正如前面所提到的，并不是所有的传输都支持所有的核心协议，其可能会限制你的选择。表4-4 展示了输和其所支持的协议。



虽然只有SCTP 传输有这些特殊要求，但是其他传输可能也有它们自己的配置选项需要考虑。此外，如果只是为了支持更高的并发连接数，服务器平台可能需要配置得和客户端不一样。这里是一些你很可能会遇到的用例。

* 非阻塞代码库——如果你的代码库中没有阻塞调用（或者你能够限制它们的范围），那么在Linux 上使用NIO 或者epoll 始终是个好主意。虽然NIO/epoll 旨在处理大量的并发连接，但是在处理较小数目的并发连接时，它也能很好地工作，尤其是考虑到它在连接之间共享线程的方式。
* 阻塞代码库——正如我们已经指出的，如果你的代码库严重地依赖于阻塞I/O，而且你的应用程序也有一个相应的设计，那么在你尝试将其直接转换为Netty 的NIO 传输时，你将可能会遇到和阻塞操作相关的问题。不要为此而重写你的代码，可以考虑分阶段迁移：先从OIO 开始，等你的代码修改好之后，再迁移到NIO（或者使用epoll，如果你在使用Linux）。
* 在同一个JVM 内部的通信——在同一个JVM 内部的通信，不需要通过网络暴露服务，是Local 传输的完美用例。这将消除所有真实网络操作的开销，同时仍然使用你的Netty 代码库。如果随后需要通过网络暴露服务，那么你将只需要把传输改为NIO 或者OIO 即可。
* 测试你的ChannelHandler 实现——如果你想要为自己的ChannelHandler 实现编写单元测试，那么请考虑使用Embedded 传输。这既便于测试你的代码，而又不需要创建大量的模拟（mock）对象。你的类将仍然符合常规的API 事件流，保证该ChannelHandler在和真实的传输一起使用时能够正确地工作。

表4-5 总结了我们探讨过的用例。

