《 UNIX 高级教程系统技术内幕 》
OCR by djq@SMTH ftp://166.111.136.159/incoming/UNIX_book/

第 17 章　流

17.1　目　　的

传统的驱动程序框架有很多瑕疵。首先，内核与驱动程序间接口的抽象层次太高(驱动程序入口点)，由驱动程序来负责大部分 I/O 请求的处理过程。设备驱动程序往往由设备厂商独立提供。很多厂商要为同一类设备编写很多驱动程序，驱动程序代码中只有一部分是设备相关的；其他部分用来实现高层的与设备无关的 I/O 处理过程。结果，驱动程序功能大量重复，既增大了内核的规模，又增加了冲突的可能性。

另一个缺点在于缓冲区。块设备接口提供厂可靠的缓冲区分配和管理功能。但对于字符驱动程序没有这样统一的调度。字符设备接口最初是为慢速设备设计的，一次只读写一个字节，像慢速串行线。因此，内核几乎没有提供缓存支持，而将这个任务留给各个设备自己完成。这就导致了几种专用的缓冲区和内存管理策略的出现，像由传统终端驱动程序使用的 clist。这些机制的使用引起了内存的不合理利用和代码的重复。

最后，这个接口给应用程序带来的好处也有限。到字符设备的 I/O 需要 read 和 write 系统调用，将数据看成是 FIFO(先进先出)的字节流。因此，它缺乏识别消息边界，区分普通数据和控制信息，或判定不同消息优先级的能力。流量控制机制也没有，每个驱动程序和应用程序不得不设计自己专用的机制来解决这些问题。

网络设备的需求更加突出了这些局限性。网络协议的设计是分层的。数据传输可能是基于消息或分组的，每一层协议完成相应的对分组的处理，然后将它传给下一层协议。协议要区别对待普通数据和紧急数据。层次之间有一些可互相转换的部分，而且一个给定的协议可能会与其他层的不同协议结合使用。这就要求有一个模块化的框架，这个框架支持分层，并且允许将几个独立的模块组合在一起生成驱动程序。

STREAMS 子系统解决了许多这样的问题，并提供了一种模块化的编写驱动程序的方法。它有一个完全基于消息的接口，便于缓冲区管理，流量控制，和基于优先级的调度策略。它支持层次化协议族的方法是，将协议模块堆集起来，像一条流水线一样作业。它鼓励代码共享，因为每个流都由几个可重用模块构成，这些模块可以为不同驱动程序所共享。通过基于消息的传输方式和对控制信息和数据的区别对待，也为用户级应用程序提供了方便。

STREAMS 最早是由 Dennis Ritchie[Ritc 83]开发出来的，现在它已经被大多数 UNIX 厂商所支持，并且成为了编写网络驱动程序和协议的首选方案。此外，SVR4 还用 STREAMS 取代了传统的终端驱动程序，以及管道机制。本章将总结 STREAMS 的设计与实现，并分析其优缺点。

17.2　概　　述

流是一个全双工处理过程，它是内核空间中的驱动程序和用户空间中的进程之间的数据传输通道。STREAMS 是一组系统调用，内核资源，和创建、使用及拆除流的例程的集合。它也是编写设备驱动程序的一个框架。它为驱动程序编写者规定了一套规则和指导方针，并为这种驱动程序的模块式开发提供了-种机制和相应的一些实用程序。

图 17-1 是一个典型的流。流完全位于内核之中，并且它的操作也是在内核中实现的。它由一个流头，一个驱动程序尾，以及其间的零个或若干个可选模块构成。流头是一个用户级接口，它允许应用程序通过系统调用接口来访问流。驱动程序尾与底层设备通信(或者，如果它是一个伪设备驱动程序的话，与其他的流通信)，那些模块在中间处理数据。

每个模块包含一对队列——一个读队列，一个写队列。流头和驱动程序也包含这样的队列对。流以消息的形式传输数据。写队列传递由应用程序向驱动程序发的下行(downstream)消息。读队列传递由驱动程序向应用程序发的上行(upstream)消息。虽然大部分消息源自流头或驱动程序，中间模块也有可能生成消息，并将它们传到头部或尾部。

每个队列可以与流中的下一个队列通信。例如，在图 17-1 中，模块 1 的写队列可能会向模块 2 的写队列发消息(不能反过来)。模块 1 的读队列可以向流头的读队列发消息。一个队列也可以向其配偶，也就是配对队列发消息。因此，模块 2 的读队列可以向该模块的写队列发消息，或反之。一个队列不必知道正在与之通信的队列是属于流头，驱动程序尾，还是其他中间模块。

图 17-1　一个典型的流

不必作进一步解释，前面的叙述已能显示出这种方法的优点。每个模块可以独立编写，甚至可能出自不同的厂商。多个模块可以以不同的方法混合、匹配，类似于 UNIX shell 中用管道组合各种命令。

图 17-2 显示了怎样用少数几个构件生成不同的流。一个开发网络软件的厂商可能希望将 TCP/IP(传输控制协议/网间协议)协议族添加到一个系统中。采用 STREAMS，他开发了一个 TCP 模块，一个 UDP(用户数据报协议)模块，和一个 IP 模块。其他生产网络接口卡的厂商为以太网和令牌环网独立编写 STREAMS 驱动程序。

一旦有了这些模块，它们就可以被动态地配置，形成不同类型的流。图 17-2(a)中，一个用户生成了一个连接到令牌环网的 TCP/IP 流，图 17-2(b)中是一个新组合，它是-个连接到以太网驱动程序上的 UDP/IP 流。

图 17-2　可重用模块

STREAMS 支持一种称作多路复用的机制。一个多路复用的驱动程序可以在上半部或下半部连接多个流。有三种类型的多路复用器——上部，下部和双向。一个上部(或称扇入)多路复用器可以在其上连接多个流。一个下部(或称扇出)多路复用器，可以在其下连入多个流。一个双向多路复用器则同时支持上部和下部的多个连接。

通过将 TCP，UDP 和 IP 模块编写成多路复用驱动程序，我们可以将上面的流组合成一个支持多种数据通路的复合对象。TCP 和 UDP 作为上部多路复用器，IP 作为双向多路复用器，图 17-3 描绘了这样一种可能的布局。这使应用程序可以实现各种网络连接，并使多个用户可以访问任何给定的协议与驱动程序的组合。多路复用驱动程序必须正确地管理所有不可的连接方式，并将数据传到正确的队列中。

17.3　消息和队列

STREAMS 将消息传递作为它唯一的通信方式。消息除了能在应用程序和设备间传递数据，还能向驱动程序或模块传递控制信息。如果出现错误，或发生某种不正常事件，驱动程宇和模块也是通过消息通知用户或其他模块。一个队列可以用多种方式处理进来的消息，它可以做某种处理后将这条消息传递给下一个队列，也可以不经处理就传出去。队列可以按计划延迟处理消息，或者，它还可以将消息传给其配对队列，然后按反方向传递。最后，队列还可以丢弃消息。

图 17-3　多路复用流

在本节中，我们将介绍消息，队列和模块的结构和函数。

注意：扩展基本类型：SVR4 扩展了几种基本数据类型。例如，dev_t 类型在 SVR3 是 16 位，但在 SVR4 中是 32 位。这些新类型被称作扩展基本类型(EFT)。类似的，很多 SVR4 中的结构加入了 SVR3 中没有的域。当这些变动影响到公共数据结构和接口时，就引发了向后兼容问题，妨碍了驱动程序和为旧版本编写的模块间的互操作性：为了解决这个问题，SVR4 提供了

一个编译选项，允许系统继续使用旧的数据类型。　当以无 EFT 方式编译时，这些域会以保持与旧驱动程序兼容的方式处理。某些域可能会放到其他的结构中，这取决于所选的编译选项。有时，某些结构只应用于某一种编译选项。未来版本中并不保证对旧数据类型的支持，本章假定所用的系统采用的是新数据类型。

17.3.1　消息

最简单的消息由三个对象组成——一个 msgb 结构(或 mblk_t 类型),一个 datab 结构(或 dblk_t 类型)，和一个数据缓冲区。一个多部分(multipart)消息可以是多个三元体的链接，如图 17-4。在 msgb 中，b_next 和 b_prev 域将消息链接到一个队列中，b_cont 域则用来链接同一消息的不同部分。b_datap 域指向相关的 datab。

msgb 和 datab 中都有实际数据缓冲区的信息。datab 中 db_base 和 db_lim 域分别指向缓冲区的头和尾。缓冲区中只有一部分包含了有用数据，因此，msgb 的 b_rptr 和 b_wptr 域分别指向了缓冲区中有效数据头和尾。allocb()例程负责分配一个缓冲区，并初始化 b_rptr 和 b_wptr，使它们同时指向缓冲区头(db_base)。当一个模块向缓冲区中写入数据时，它增大 b_wptr 值(检查是否读过了 db_lim)。当一个模块从缓冲区中读取数据时，它增大 b_rptr 的值(检查是否读过了 b_wptr)，从而将数据从缓冲区中删除。

允许多部分消息(每部分是一个 msgb_datab_data 缓冲区三元体)有很多好处。网络协议是分层的，每个协议层往往都要将它自己的头和尾加到消息上。当一条消息向下传递时，

图 17-4　-个 STREAMS 消息

各层可以生成一个新的消息块，并将之链接到旧消息的前部或末尾。这样，上层协议就不必知道下层协议的头部和尾部的绝对位置，或在分配消息时为它们留下空间。当从网上收到消息，并向上行流传递时，每层协议只需通过调整 b_rptr 和 b_wptr 域，就可以将该层的头部和尾部去掉。

b_band 域包含有消息的优先级，是用来做调度的(见第 0 部分)。每个 datab 有一个 db_type 域，它包含由 STREAMS 定义的几种消息类型中的一种。通过消息分类，可以使模块根据类型区分优先级，并区别对待各类消息。第 17.3.3 节中将详细讨论消息类型。db_f 域中的信息是用于消息分配的(见第 17.7 节)。db_ref 域中有一个引用计数器用于虚拟复制(见第 17.3.2 节)。

17.3.2　虚拟复制

datab 中的 db_ref 域是用于做引用计数的。多个 msgb 可以共享同一个 datab，并由此共享与它相关联的缓冲区中的数据。这使有效的数据虚拟复制成为可能。图 17-5 描绘了两个消息共用一个 datab 的情况。两者共享相关的数据缓冲区，但各自又都有独立的读写偏移量。

通常，这种共享缓冲区是用于只读模式的，因为两个相互独立的写操作会相互干扰。但是，这种情况只能由处理缓冲区的模块或驱动程序宋避免，STREAMS 既不知道也不关心模块是何时或如何读写缓冲区的。

虚拟复制的一个例子是用在 TCP/IP 协议中的。TCP 层提供可靠的传递，因此必须保证每条消息都能到达目的地。如果接收者在一个指定的时间内没有确认消息，发送者就要重传。为此，它必须保留每个发送的消息，直到它被确认。物理地复制每一条消息是一种浪费，因此 TCP 采用于虚拟复制机制。当 TCP 模块收到一条消息并向下传时，它调用 STREAMS 例程 dupmsg()，由它生成引用同一个 datab 的 msgb。这便产生了两条逻辑消息，两条消息引用同样的数据。这样，TCP 将一条消息发送下去，另一条留作备份。

驱动程序发送完消息并释放 msgb 后，datab 和数据缓冲区并没有被释放，因为其引用计数还是非零。最终，等接收者确认了消息，TCP 再释放另一个 msgb。这就会使使 atab 中的引用计数降为零，STREAMS 就会释放 datab 和与之相关的数据缓冲区。

17.3.3　消息类型

STREAMS 定义了一套消息类型，每条消息必须属于其中的一种。datab 中 db_type 域指定了这个类型。一条消息的类型关系到它的用途和在队列中的优先级。根据类型，消息可以被分为普通型和高优先级型。高优先级消息优先排队和处理。第 1 7.4.2 节详细介绍了消息优先级。

SVR4.2 定义了如下几类普通消息[USL 92a]：

- M_BREAK 向下发送；要求驱动程序向设备发一个 break 消息。
- M_CTL 模块间控制请求。
- M_DATA 由系统调用发送或接收的普通数据。
- M_DELAY 要求在输出上产生-个实时延迟。
- M_IOCTL 控制信息，由流的 ioctl 命令生成。
- M_PASSFP 传递一个文件指针。
- M_PROTO 协议控制信息。
- M_RSE 保留。
- M_SETOPTS 由模块或驱动程序向上行流发送消息，以设置流头选项。
- M_SIG 由模块或驱动程序向上行发流发送消息；要求流头向用户发一个信号

下面是高优先级类型的消息：

- M_COPYIN 向上行流发送消息，要求流头拷入一个 ioctl 数据。
- M_COPYOUT 向上行流发送消息，要求流头拷出一个 ioetl 数据。
- M_ERROR 向上行流发送消息；报告一个错误情况。
- M_FLUSH 要求模块刷新(flush)一个队列。
- M_HANGUP 向上行流发送消息，设置流头的挂起状态。
- M_IOCACK ioctl 确认，向上行流发送消息。
- M_IOCNACK ioctl 否认，向上行流发送消息。
- M_IOCDATA 向下发送 ioctl 数据。
- M_PCPROTO 高优先级版本的 M_PROTO。
- M_PCSIG 高优先级版本的 M_SIG。
- M_PCRSE 保留。
- M_READ 读提示，向上行流发送消息。
- M_START 重新启动暂停了的设备输出。
- M_STARTI 重新启动暂停丁的设备输入。
- M_STOP 暂停输出。
- M_STOPI 暂停输入。

消息类型的定义可以让模块在不知道消息内容的情况下，知道如何满足各种特殊的处理要求。对于多部分消息，第一个 datab 中包含了整个消息的类型。此规则有一个例外。当一个应用程序使用一个高层服务接口时(比如 TPI，Transport Provider Interface，传输供应接口)，它的数据消息的组成方式是，一个 M_PROTO 信号模块，然后是在同一个消息中的一个或多个 M_DATA 块。

17.3.4 队列和模块

模块是流的构成单元。每个模块包括两个队列——一个读队列和一个写队列。图 17-6 描绘了队列的数据结构。它有如下几个重要的域：

- q_qinfo 指向一个 qinit 结构(在下面一段中介绍)的指针。
- q_tirst，q_last 用来管理一个用于延时处理消息的双向链表的指针。
- q_next 指向下一个上行和下行流队列。
- q_hiwat，q_lowat 队列中可以容纳数据量的最大、最小水线，用于流量控制(见第

17.4.3 节)。

・q_link　用来将队列链入要作调度的队列列表中的指针(见第。节)。

・q_ptr　指向队列私有数据的指针。

q_qinfo 域指向一个 qinit 结构，该结构封装了到这个队列的过程化接口。图 17-7 显示了从 q_qinfo 访问的数据结构。每个队列必须提供四个过程 put，service，open 和 close，这些是其他 STREAMS 对象与它通信的唯一接口。module_info 结构包含缺省的高低水线，分组大小，和其他该队列的参数。有一部分域也在 queue 结构中出现。这使用户可以通过改变 queue 中的值，动态重载这些参数，而不必破坏 module-info 中的缺省设置。module_stat 对象不直接为 STREAMS 所用。每个模块可以利用这些对象中的域完成自己的统计数据收集工作。

下面几个小节中将详细讨论队列中的过程。简而言之，open 和 close 过程是由进程同步

图 17-7　通过 q_qinfo 访问的对象

调用的，用来打开和关闭流。put 过程用来立即处理一条消息。如果某条消息不能被立即处理，put 过程会将它加到队列的消息队列中。然后，当 service 过程被调用时，它会处理这些延迟的消息。

每个队列必须提供一个 put 过程①，但 service 过程却未必。如果没有 service 过程，put 过程就不能对消息延迟处理，必须立即处理每一条消息，并将这条消息传给下一个模块。在最简单的情况下，一个队列没有 service 过程，而且其 put 过程也只是将消息未经处理地传给下一个队列。

注意　这里的术语有一些混乱，因为队列(queue)在这里既指队列对象，又指包含在其中的消息队列。本书用队列一词指代队列对象，而消息队列指队列中的消息链表。

17.4　流 I/O

流通过从一个队列向另一个队列传递消息来实现 I/O，用户进程用 write 或 putmsg 系统调用来向这种设备写数据。流头分配一条消息(一个 msgb，datab 和数据缓冲区)并将数据复制到其中。然后它将消息向下行队列传递，最终到达驱动程序。来自设备的数据是异步到达的。驱动程序将这些数据向上行流发送，直到流头。进程通过 read 或 getmsg 调用接收数据。如果在流头没有数据，这个进程就会阻塞。

一个队列在流中通过调用 putnext()将一条消息传到下一个队列。putnext()函数通过 q_next 域找到下一个队列，并调用该队列的 put 过程。一个队列决不能直接调用下一个队列的 put 过程，因为 q_next 属于队列的内部域，可能在未来版本中有所变动，一个队列可以通过将一条消息传给其配偶队列的方式，实现消息反向传递。例如，一个读队列可以这样做：

WR(q)->put(WR(q)，msgb)

流 I/O 是异步的。唯一可以阻塞 I/O 操作的点是流头。模块和驱动程序的 put 和 service 过程是非阻塞的。如果 put 过程不能将数据传递给下一个队列，它会将这条消息放到自己的消息队列中，之后将由 service 过程获取。如果 service 过程从队列中删除了一条消息，并发现它此时不能处理它，它会把消息返还到队列中，以便以后再试。

这两个函数是互补的，put 过程用来处理不能等待的情况。例如，一个终端驱动程序必须立即响应它接收到的字符，否则用户会觉得它反应不够及时。service 过程处理所有不太紧急的事件，比如规范的输入字符处理方式。

因为两个过程都是不允许阻塞的，它们必须保证自己调用的例程也不会阻塞。因此，STREAMS 提供了它自己的一些常用操作函数，像内存分配。例如，allocb()例程专门用来分配一条消息。如果它发现由于某种原因现在无法分配(可能找不到可用的 msgb，datab，或数

据缓冲区)，它会返回一个错误号，而不会阻塞。然后，调用者会调用 bufcall()例程，向一个回调函数传递一个指针。bufcall()将调用者加入到一个等待分配内存的队列中。当有内存可用时，STREAMS 调用这个回调函数，通常由它再调用流的 service 例程，重新尝试使用 allocb()。

异步操作是流设计的内核。对读方来说(上行流)，驱动程序通过中断获取数据。读方的 put 过程在中断一级，因此不能阻塞。设计方案可以允许阻塞写方过程，但这又违背了对称性和简单性原则。

service 过程不是以初始化数据传输的进程上下文进行的，而是在系统上千文中被调度的。因此阻塞 service 进程只会将一个无辜的进程置于休眠状态。例如，如果一个用户 shell 进程由于一个不相关的传输不能完成而被阻塞，其结果是不可想象的。使所有的 put 和 service 过程成为非阻塞的，这样就可以避免这类问题了。

在访问公共数据结构时，put 和 service 过程之间必须相互同步。因为读方 put 过程可以从中断处理函数中调用，它既可能中断 service 过程的执行，也可能中断写方 put 过程。在多处理器环境中，由于这些过程可能在不同处理器上并发运行，需要额外加锁[Garg 90，Saxe93]。

17.4.1 STREAMS 调度程序

如果 put 过程要延迟处理数据，它调用 putq()将消息放入队列，然后调用 qenable 来调度等待服务的队列。qenable()设置该队列的 QENAB 标志位，并将该队列加到一个等待调度的队列列表的表尾。如果 QENAB 已经设好了，qenable()就什么也不做，因为这表明该队列已经调度过了。最后，qenable()设置全局标志 qrunflag，这个标志是用来表明现在有一个队列正在等待调度的。

STREAMS 调度是由一个称作 runqueues()的例程实现的，与 UNIX 进程调度无关。一旦某个进程想要在一个流上执行 I/O 或控制操作，内核就会调用 runqueues()。这样可以让很多操作在上下文切换前快速完成。内核也会在上下文切换后，返回用户态前调用 runqueues()。

runqueues()检察是否有流需要调度。如果有，就调用 queuerun()，由它来扫描调度程序列表，并调用每个队列上的 service 过程，service 过程必须试着处理队列中的所有消息，如下所述。

在单处理器环境下，内核保证在返回用户态前运行所有已调度的 service 过程。因为在同一时刻任何进程都有可能运行，service 过程必须在系统上下文中执行，不能访问当前进程的地址空间。

17.4.2 优先带(Priority Bands)

很多网络协议支持带外(out-of-band)数据的概念[Rago 89]，它由紧急的、协议专用的控制结构组成，必须在普通数据之前处理。这与根据消息类型区分的高优先级消息不同。例如，TELNET 协议提供了一种 Synch 机制来重新获取进程的优先权，其具体方法就是发送一个紧急消息。通常在流中有一个特殊的数据标志表示带外数据(也称作加速数据)的尾部。

STREAMS 将带外数据当作普通数据处理，因为这种数据的处理是因协议而异的。但是，它提供了一种称作优先带的功能，允许模块为消息赋予优先权，并按优先级的次序处理这些消息。某些协议可以用这种优先带来实现数据消息分类。

优先带仅用于普通消息类型。每个这样的消息被赋予一个从 0 到 255 之间的优先带值。0 是缺省值，大部分协议只用到了 0 和 1，高优先级消息类型(像 M_PCPROTO)没有优先带值，它们比所有有优先带值的消息都更紧急。

在一个流中，STREAMS 为每个使用中的优先带维护着一个队列。为达到这一目的，它使用了一套 qband 结构，每个优先带分配一个。STREAMS 在使用中动态分配 qband 结构，putq()将一条消息加入队列时，STREAMS 将其加到合适的 qband 结构的链表尾(如果有必要，分配一

个新的 qband)。当 service 过程通过调用 getq()获取一条消息时，STREAMS 返回一条具有最高优先带值的消息。

因此，service 过程首先处理所有挂起的高优先带消息，之后按带优先级顺序处理普通消息。在每个优先带内，它以 FIFO 顺序处理消息。

17.4.3 流量控制

最简单的流量控制是没有流量控制。假设有一个流，它的每一个模块都只有一个 put 过程。当数据通过这个流时，每个队列处理这些数据并通过调用 putnext()将处理过的数据传给下一个队列。数据到达驱动程序尾后，驱动程序立即将其传给设备。如果设备不能接收数据，驱动程序就将之丢弃。

虽然对于某些设备来说，这种方法是可取的(至少，null 设备可以用这种方法)。大部分应用程序是不希望因为设备没有准备好就丢数据的，而且大部分设备不可能总是处于准备好的状态。这就要求流能够处理其一个或多个构件中可能出现的障碍，而又不至于阻塞 put 或 service 过程。

流量控制在队列中是可选的。支持流量控制的队列与另一端最近的支持流量控制的队列交互。不支持流量控制的队列没有 service 过程。它的 put 过程立即处理所有的消息，并将它们传给下面的队列。它的消息队列并没使用。

支持流量控制的队列定义了高低水线，由这两个水线来控制队列中数据的总量。这两个值最初从 module_info 结构(它是在编译模块时静态初始化的)中复制出来，但有可以通过以后的 ioctl 消息改变。

图 17-8 显示了流量控制的操作。队列 A 和 C 是有流量控制的，B 没有。队列 A 收到一条消息时，它的 put 过程被调用。它完成所有必要的数据操作后，调用 putq()，putq()例程将消息添加到自己的消息队列中，并将队列 A 添加到需要服务的队列列表中。如果这条消息导致队列 A 的数据量超过高水线，它会设置一个标志位表明队列已满。

图 17-8 两个队列间的流量控制

一段时间后，STREAMS 调度程序选择队列 A，并调用其 service 过程。这个 service 过程从队列中按 FIFO 的次序取出消息进行处理。处理完后，它调用 canput()查看下一个流量控制队列是否可以接收消息。canput()例程追寻 q_next 指针直至发现下一个有流量控制的队列，在本例中是队列 C。它再检查队列的状态，如果队列可以接收更多的消息，则返回 TRUE，否则，返回 FALSE，队列 A 的 service 过程在两种情况下采取不同的动作，如下例 17-1 所示。

实例 17-1 servke 过程中的消息处理

```
if(canput(q->q_next))
    putnext(q, mp);
else
    putbq(q, mp);
```

如果 canput()返回的是 TRUE，队列 A 调用 putnext()将消息传给队列 B。队列 B 是不支持流量控制的，它立即处理完这条消息，并将它传给队列 C，因为队列 C 还没满。

如果 canput()返回的是 FALSE，队列 A 调用 putbq()将消息放回其消息队列中。service 过程不必进行再调度就可以返回。

最终，队列 C 要处理自己的消息，使数据量降到低于低水线的水平。这时，STREAMS 自动查看前面的流量控制队列(本例中是 A)是否阻塞了。如果是，就会重新调度队列，准备服务。这个操作称作回访(back-enabling)一个队列。

流量控制要求模块编写的一致性，所有相同优先级的消息必须平等对待。如果 put 过程将消息加入队列，等待 service 处理，它就应该对所有消息都同样处理。否则，消息就不能保持原有次序，导致出错。

service 过程运行时，必须处理队列中的所有消息，除非因为分配失败，或者因为下一流量控制队列已满。否则，流量控制机制会崩溃，而队列也可能永远不被调度。

高优先级消息不受流量控制的约束。一个将普通消息加入队列的 put 过程可能会立即处理高优先级消息。如果高优先级消息必须被加入队列，它会被放到队列的开头，位于所有普通队列之前。高优先级消息保持相互之间的 FIFO 次序。

17.4.4 驱动程序尾

驱动程序与所有模块相似，但又有一点重要的区别。首先，它必须能接收中断。这就要求它有一个中断处理函数，并且以一种依赖于机器的方式让内核知道这个处理函数的存在。设备在收到到达数据时产生中断。驱动程序必须将数据包装到一条消息中，再将这条消息向上发送出去。当驱动程序收到来自流头的消息时，它又必须从消息中取出数据，并发送给设备。

正如大多数设备所需要的那样，驱动程序通常实现了某种形式的流控制。在很多情况下，特别是对于传入的数据，驱动程序在来不及载入时就会将消息丢弃，因此，当驱动程序不能分配缓冲区，或者是队列溢出时，它会一声不吭地将进来或出去的消息丢掉。恢复丢弃的数据包是应用程序的字。高层协议，像 TCP，保留每条消息的副本直到它到达目的地，这样就可以保证可靠的传输；如果接收者在指定时间内没有确认消息，协议会重传该消息。

驱动程序尾的打开和初始化操作也与其他模块不同。STREAMS 通过 open 系统命令打开驱动程序，而模块是通过 ioctl 调用叠加在流上的。第 17.5 节将详细介绍这些操作。

17.4.5 流头

流头负责系统命令处理。它还是流中唯一一个可以阻塞 I/O 调用处理的部分。虽然每个模块或驱动程序都有它自己的 put,service,open,和 close 过程,所有的流头共享一套 STREAMS 内部公共的例程。

一个进程使用 write 或 putmsg 系统命令向流中写数据。write 系统调用只允许写普诵数据,而且不能保证消息边界。对于那些将流视作字节流的应用程序来说,它是很有用的 ;putmsg 系统调用允许用户在一个调用中使用一条控制消息和一条数据消息，STREAMS 将两者合并到一条消息中，这条消息的第一个 datab 的类型是 M_PROTO，第二个是 M_DATA。

在两种情况下，流头都要将数据从用户地址空间中拷到 STREAMS 消息中，然后调用 canput()，看看流中是否有剩余空间(下一个流量控制模块或驱动程序未满)。如果有，它调用 putnext()将数据传给下一个队列，并将控制返回给调用者。如果 canput()返回 FALSE，调用程序就会在流头处被阻塞，直到流头被下一个流量控制模块回访(back-enable)。

因此，write 或 putmsg 系统调用返回后，数据并不一定已到达设备。系统能保证调用者的数据已经被安全地复制到了内核中，要么已到达设备，要么放在某模块或驱动程序的队列中。

一个进程可以使用 read 或 getmsg 从流中读取数据。read 系统调用只能读取普通数据。一个模块可以向其流头发送一个 M_SETOPS 消息，告诉流头将 M_PROTO 消息当作普通数据。然后由 read 调用读取 M_DATA 和 M_PROTO 消息中的内容。不管怎么说，read 系统调用不能保证消息边界，或者返回有关消息类型的信息，它主要用于将数据视作字节流的应用程序。

getmsg 系统调用既可以获取 M_PROTO 消息，也可以获取 M_DATA 消息。它保证了消息的边界,并且对于两种类型的消息的返回参数也各不一样。如果一个输入的消息由一个 M_PROTO 块和一个或多个 M_DATA 块组成，getmsg 会将这两部分正确地分隔开。

在两种情况下，如果数据已传到流头中，内核将从中将它拾取出来，复制到用户空间，并将控制权返回给调用者。如果流头中没有等待处理的消息，内核将会阻塞调用者?直到有消息到达。多个进程可能会读取同一个流。如果没有消息等待处理，所有的进程都会被阻塞。当一条消息到达时，内核会将它传给其中的某一个进程。接口本身并不管哪个进程接收消息。

一条消息传到流头后，内核检查是否有进程在等这条消息。如果有，内核将该消息复制到进程的地址空间，并唤醒该进程。这时该进程就能从 read 或 getmsg 中返回了。如果没有等待进程，内核将其加入流头的队列。如果流头的队列满了，就将其加入上一个有流量控制的队列中，依此类推。

17.5 配置和设置

本节将说明如何将 STREAMS 驱动程序配置到系统中，以及如何在运行中的内核内创建和设置流。STREAMS 的配置分为两个阶段。首先，内核建好后，STREAMS 模块和驱动程序必须与内核相链接，并且，某些内核例程必须知道如何找到它们。其次，必须创建和设置某些设备文件，以便让应用程序像访问普通文件一样访问设备文件。STREAMS 的设置是动态的，是在一个用户打开流并向其中插入模块时发生的。

17.5.1 配置一个模块或驱动程序

STREAMS 模块和驱动程序通常由设备厂商编写，独立于内核。随后，它们必须与内核的其他部分相链接，以便内核访问它们。STREAMS 提供了一整套实用方法来实现这个功能。

每个 STREAMS 模块必须提供三个配置数据结构——module_info，qinit 和 streamtab。图 17-9 描述厂它们的内容和相互之间的关系。streamtab 是唯一一个公共可见的对象，其他对象通常都是静态的，因此在模块外是不可见的。

streamtab 包含两个 qinit 结构指针——一个是读队列的，一个是写队列的。其他两个域只用于多路复用驱动程序，存储指向另外的队列对的指针。qinit 结构包含指向函数组(open，close，put 和 service)的指针，这个函数组形成了队列的过程化接口，open 和 close 例程只在读队列中定义，是模块所共有的。所有的队列都有一个 put 例程，但只有支持流量控制的队列才有 service 例程，qinit 结构还包含一个指向该队列 module-info 结构的指针。

图 17-9 用于配置模块或驱动程序的数据结构

module_info 结构中包含有模块的缺省参数。这个模块第一次被打开时，这些参数被复制到队列结构中。用户可以通过 ioctl 调用来改变这些参数。每个队列可以有自己的 module_info 结构，或像上图中的例子那样由一对队列共享一个结构。

剩下的配置工作针对不同的模块和驱动程序各有不同。很多 UNIX 系统用一个 fmodsw[]表来配置 STREAMS 模块。表中的每项(如图 17-10(a)所示)由一个模块名和一个指向该模块 streamtab 结构的指针构成，因此，模块按名字被标识和引用。模块名应该与 module_info 结构中的 mi_idname 一致，但 STREAMS 并不强求这样。

STREAMS 设备驱动程序是通过字符设备开关表来被识别的。每个 cdevsw 项有一个称作 d_str 的域，如果这个域为 NULL，表明是普通字符设备。对于 STREAMS 设备，这个域包含了一个该驱动程序 streamtab 结构的地址(见图 17-10(b))。要完成配置工作，还要创建一个相应的设备文件，文件的主设备号等于驱动程序在 cdevsw[]表中的索引(克隆打开方式除外，在第 17.5.4 节中有介绍)。STREAMS 驱动程序必须处理设备中断，因此需要一种额外的机制来安装其中断处理函数。安装过程是因系统而异的。

一旦驱动程序配置完，它就可以在内核启动后供应用程序使用了。下面的小节中将继续介绍这些问题。

图 17-10 模块和驱动程序的配置

17.5.2 打开流

用户通过打开相应的设备文件打开一个流。STREAMS 设备第一次被用户打开时，内核通过分析路径名得知它是一个字符设备。它调用 specvp()，为这个文件分配一个 s 节点和一个公共 s 节点，并初始化(第 16.4.4 节中有叙述)。然后，它调用这个 v 节点(v 节点与公共 s 节点相关联，作为流的 v 节点)上的 VOP_OPEN 操作，由 spec_open()函数来作处理。

spec_open()用设备文件的主设备号作为索引，查找 cdevsw[]表，发现该设备是一个流设备(d_str!＝NULL)。接着，它调用 stropen()例程，传给它指向 v 节点的指针和指向设备号的指针，以及打开标志和相应证件。打开一个新设备时(还没有打开过的设备)，stropen()执行下列操作：

1.为流头分配一个队列对。

2.分配和初始化一个 adata 结构，代表一个流头。

3.设置流头队列，使之指向 strdata 和 stwdam 对象，两者都是 qinit 结构(读队列和写队列各一个)，包含着通用的流头函数。

4.将 v 节点指针存放到 stdata 结构的 sd_vnode 域中。

5.将指向流头的指针存放到 v 节点中(v_stream 域)。

6.将指向该驱动程序 streamtab 结构的指针存放到流头中的 sd_streamtab 域中(从 cdevsw 项中获得)。

7.使流头队列的私有数据指针(q_ptr)指向 stdata 结构。

8.调用 qattach()来设置驱动程序尾，下面会介绍具体操作。

9.调用 qattach()，将由设备标志的 autopush 模块插入到流中。这在下一小节中将详

图 17-ll　stropen()后的数据结构

qattach()通过如下操作将一个模块或驱动程序加到流头下面：

1.分配一个队列，并将它链接到流头下面。

2.通过在 edevsw[]表中找驱动程序，或在 fmodsw[]表中找模块，定位 streamtab 结构。

3.从 streamtab 表中取出读写 qinit 结构，并利用它们来初始化队列对的 q_qinfo 域。

4.最后，调用模块或驱动程序的 open 过程。

现在,假设另一个用户通过同一个设备文件或另一个具有相同主次设备号的设备文件(后一种情况下有公共 s 节点来处理)，打开同一个流。内核会发现公共 s 节点的 v 节点中的 v_stream 域不为 NULL,而是指向了流的 stdata 结构。这说明流已经被打开了,因此,stropen()只是简单地调用流头和流中每一个模块的 open 过程,告诉它们又有另一个进程打开了同一个流。

17.5.3　插人(Pushing)模块

通过一个带 LPUSH 命令的 ioctl 调用，用户可以将一个模块堆叠到一个已打开的流中，内核分配一个队列对，并调用 qattaeh()将其堆叠到流中。qattach 通过在 fmodsw[]表中定位其 strtab 项来初始化该模块。它将该模块链接到流中紧连着流头的位置,并调用它的 open 过程。

用户可以用 LPOPioctl 命令从队列中移走下一个模块。往往移走的是紧连着流头的模块，因此，模块的弹出是遵从后进先出(LIFO)的次序的。

通过一个称作 STREAMS 管理驱动程序(sadc(8))的特殊驱动程序的 ioctl 命令，STREAMS 提供了一种 autopush 的机制。通过这种机制，管理员可以指定一组在第一次打开流时向其堆叠的模块,stropen()例程检查发现该流的自动堆叠生效了，就会按次序寻找并插入所有指定的模块。

还有两种其他的常用机制用于向流中堆叠模块。一种是提供一组库例程，用于打开流并向其中插入正确的模块。另一个是在系统初始化时启动一个守护进程来完成这些工作。之后，只要应用程序打开这个设备文件，它就会被连到这一个流上，流中所有的模块都已经插入好了。

17.5.4　克隆设备

第 16.4.3 节中介绍了设备克隆的概念。其原理是，某些设备类型可能会有多个相同的实例。每个设备实例需要一个唯一的次设备号。当用户在打开这个设备时，并不关心他打开的

是哪个设备实例，只要它不是以前打开的旧实例就行。这时，最好由驱动程序提供次设备号，而不必让用户自己去找未用过的次设备号。

克隆主要用于 STREAMS 设备，例如网络协议和伪终端。因此 STREAMS 提供了一个克隆驱动程序，实现 STREAMS 设备克隆自动化。克隆设备有它自己的主设备号，由一个 STREAMS 驱动程序来实现。它为每一个支持克隆的 STREAMS 设备提供一个设备文件。该文件的主设备号是克隆设备的主设备号，其次设备号等于实际设备的主设备号。

例如，如果克隆驱动程序有一个主设备号，63。设备文件 dev/tcp 可以代表所有的 TCP(传输控制协议)流。如果 TCP 驱动程序的主设备号是 31，那么/dev/top 文件的主设备号为 63，次设备号为 31。当一个用户打开/dev/tcp 文件时，内核分配一个 s 节点和一个公共 s 节点，然后调用 spec_open()，spec_open()调用克隆驱动程序的 d_open 操作，传给它一个设备号(也就是一个指向公共 s 节点的 s_dev 域的指针)。

clnopen()例程为克隆驱动程序执行 d_open 操作。clnopen()从 s_dev 中取出次设备号(本例中是 31)，并将它作为索引在 cdevsw[]中查找 TCP 驱动程序。然后，它调用该驱动程序的 d_open 操作，传给它一个 CLONEOPEN 标志和设备号。在本例子中，会调用一个 tcpopen()函数。tcpopen()看见 CLONEOPEN 标志后，生成一个未用的次设备号，并将该号回写到 s 节点中。

clnopen()返回后，spec_open()知道发生了一次克隆打开。因为公共 s 节点是与克隆设备(/dev/tcp)相关联的，spec_open()必须为这次连接分配一个新 v 节点和 s 节点。它初始化新 v 节点中的 v_stream 域，使之指向流头，并将新主设备号和次设备号(从 s_dev 域)拷入新 v 节点和 s 节点。然后，它调用 stropen()打开新流。

最后，spec_open()将最初公共 s 节点(与/dev/tcp 相关联的)的 v_stream 域清零。这样看上去该设备还从来没有被打开过。然后，如果另一个进程想要打开/dev/tcp，内核执行同样的一系列操作，并为它创建一个新流和新设备号。这样，用户就可以得到一个唯一的 TCP 连接，而不必猜哪个次设备号是空闲的。

17.6 STREAMS ioctl

STREAMS 需要特殊的机制来处理 ioctl。虽然有些 ioctl 命令是在流头处理的，其他的是以驱动程序和中间模块为目的地的。它们被转化成消息向下发送。这引发了两方面的问题——进程同步和用户空间与内核空间之间的数据传输。

流头负责同步，如果它自己处理这些命令，它可以在进程上下文中同步完成，没有任何问题，如果流头必须将命令传递给下面的模块或驱动程序，它就要阻塞进程，并且向下发送一个包含该命令及其参数的 M_IOCTL 消息。某个模块处理这条命令后，将结果放到一条 M_IOCACK 消息中返回。如果没有模块或驱动程序能够处理这条消息，驱动程序生成一个 M_ICONACK 消息。流头收到消息后，唤醒进程，并将处理结果传给它。

数据传输问题关系到用户程序与处理 ioctl 的模块或驱动程序间的参数和结果交换。如果用户向一个普通字符设备发送一个 ioctl 命令，驱动程序在调用进程的上下文中处理这条命令。通常，每一条 ioctl 命令有一个参数块，其大小和内容取决于命令。驱动程序从用户空间中将这个块拷入内核，处理命令，然后将结果拷到用户区。

这种方法对于 STREAMS 驱动程序和模块是不适合的，模块将命令作为一条 M_IOCTL 消息接收，但确是相对进程异步完全且是在系统上下文中。因为该模块没有权限访问进程的地址空间，它既不能传进参数块，也不能传出结果。

STREAMS 提供了两种方法解决这个问题。好一点的方法引入了一个称作 I_STR 的 ioctl 命令类型。另一种方法只处理普通 ioctl 命令，因此与旧应用程序保持了兼容。这种方法被称作透明 ioctl 处理，因为它不需要对现有应用程序作任何改动。

17.6.1 I_STR ioctl 处理

ioctl 系统调用的语法是：

```
ioctl(fd,cmd,arg);
```

其中 fd 是文件描述符；cmd 是一个整数，指定命令类型；arg 是可选的，因命令而异，通常是一个参数块的地址。驱动程序根据 cmd 解释 arg 中的内容，并将参数从相应的用户空间传进来。

通过将 cmd 的值设为常数 I_STR，并将 arg 指向 strioctl 结构，用户可以发一个特殊的 STREAMS ioctl 消息。strioctl 的结构如下：

```
struct strioctl{
int  ic_cmd;    /* the actual command to issue */
int  ic_timeout;    /* timeout period */
int  ic_len;    /* length of parameter block */
char * ic_dp;    /* address of parameter block */
};
```

如果流头不能处理这个 ioctl，它创建一个类型为 M_IOCTL 的消息，并将 ic_cmd 值复制给它。它还从用户空间取出参数块(根据 ic_len 和 ic_dp)，复制到消息中。然后，它将这条消息向下传递。当可以处理这条命令的模块收到这条消息后，它将所有必要的信息复制下来。如果该命令需要向用户返回数据，该模块会将数据写入同一消息，将消息的类型改为 M_IOCACK，并将它向上发送。流头会将结果拷入用户空间的参数块中。

流头向下传递消息，直到遇到一个能识别和处理它的模块。如果有模块向流头发送回一条 M_IOCACK 消息，表明这条命令已经被成功地识别处理了。如果所有模块都不能识别这条消息，最终驱动程序会收到它。如果驱动程序也不能识别，它会传回一条 M_IOCNACK 消息，流头会因此生成一个合适的错误号。

这个方法效率很高，但对于它能处理的命令有所限制，它不能与旧的没有使用 I_STR 命令的应用程序正常工作。而且，因为流头不能识别参数，它只能机械地复制参数的内容。例如，如果一个参数是一个指向用户区间的字符串指针，流头将复制指针而不是字符串。因此，需要一种能处理所有情况的通用方法，尽管它会慢一些或效率低一些。

17.6.2 透明 ioctl

透明 ioctl 提供了一种在不使用 I_STR 消息的情况下处理命令数据复制问题的机制，进程发出透明 ioctl 后，流头创建一个 M_IOCTL 消息，并拷入 cmd 和 arg 参数。通常，arg 的值是一个指向参数块的指针，其大小和内容只有处理这条命令的模块才知道。流头向下发送这条消息，并阻塞调用进程。

相应的模块收到这条消息后，返回一条 M_COPYIN 消息，传回参数块的大小和位置(与 arg 相同)。流头唤醒发出 ioctl 的进程，处理 M_COPYIN 消息。进程生成一个新的类型为 M_IOCARGS 的消息，将数据从用户空间拷入其中，并向下发送，再次阻塞。

相应模块收到 M_IOCARGS 后，解释参数，处理命令。在某些情况下，模块和流头之间可能要交换好几次消息以正确读入参数。例如，如果某参数是一个指向字符串的指针，该模块就要发一条额外的消息以获取字符串。

最后，该模块收到了所有所需参数，并处理完命令。如果必须将结果写回给用户，该模块会发出一条或多条 M_COPYOUT 消息，将结果和结果存放的位置传回去。每次收到这样的消息，流头就唤醒进程将结果写进其地址空间。所有结果传完后，该模块发送一条 M_IO-CACK 消息，然后流头最后一次唤醒进程，完成 ioctl 调用。

17.7 内存分配

STREAMS 内存管理有一些特殊的要求，因此不能由普通的内存分配程序来处理，频繁的

使用消息使模块和驱动程序需要一种高效率的分配和释放机制。pul 和 service 过程必须是不可阻塞的。如果分配程序不能立即捉供内存，它必须以不阻塞的方式处理这一情况，也许是等到以后再重试。此外，很多 STREAMS 设备允许来自设备缓冲区的直接内存访问(DMA)。STREAMS 允许这样的内存直接转换为消息，不用复制到主内存中。

主内存管理的例程有 allocb()，freeb()和 freemsg()。allocb()的语法是

mp = allocb(size, pri);

allocb()分配至少 size 字节长的一个 msgb，一个 datab，和一个数据缓冲区，它返回一个指向 msgb 的指针。它初始化这些结构，使 msgb 指向 datab，datab 中有数据缓冲区的首尾指针。它还设置 msgb 中的 b_rptr 和 b_wptr 域，使之指向数据缓冲区的开头。pri 参数已停止使用，保留它只是为了保持向下兼容。freeb()例程释放一个 msgb，freemsg()遍历 b_cont 链，释放消息中的所有 msgb。在两种情况下，内核都要将相应 datab 的引用计数递减。如果计数降到零，还要释放 datab 和所指向的数据缓冲区。

像上面那样分别分配三个对象效率低下。STREAMS 通过使用一个称作 mdbblock 的对象提供了一种更快捷的方式。每个 mdbblock 有 128 个字节，包括一个 msgb，一个 datab，和一个指向释放处理函数(release handler)的指针，这个指针在下面的小节中将会讨论。剩下的空间可以作为一个数据缓冲区。

让我们来看看当一个模块调用 allocb()分配一条消息时会发生什么。allocb()调用 kmem_alloc()来分配一个 mdbblock 结构，传给它一个 NO-SLP 标志。这样，当不能立即得到内存时，kmem_alloc()就会返回一个错误号，而不是阻塞。如果分配成功，allocb()检查所申请的空间是否能容入 mdbblock 中。如果能，它初始化该结构，并返回一个指向 msgb 的指针(如图 17-22 所示)。这样，一个 kmem_alloc()调用就能得到 msgb，datab，和数据缓冲区。

如果申请的空间大于 mdbblock 的空间，allocb()会再次调用 kmem_alloc()。这次专门用作数据缓冲区。这次，mdbblock 中的额外空间就没有用处了。如果两次 kmem_alloc()调用中有一次失败，allocb()就释放它已获取的所有资源并返回 NULL，表示失败。

模块和驱动程序必须能处理 allocb()失败。一种可能性是丢掉所处理的数据。在很多网络驱动程序中，如果不能及时处理进来的数据，会采用这种方法。但是，往往模块希望一直等待，直到有内存可分为止。因为 put 和 service 过程是不可阻塞的，必须用其他办法等待内存。

STREAMS 提供了一个称作 bufcall()的例程来处理这种情形。当一个模块不能分配一条消息时，它调用 bufcall()例程，传给它一个回调函数的指针和等待分配的消息的大小。STREAMS 将这个回调函数加到一个内部队列中。当有足够内存时，STREAMS 再处理这个队列，调用队列上的每个回调函数。

图 17-12　小消息分配

通常，回调函数就是 service 过程本身。但是，回调函数并不一定能得到足够的空间。在它之前，可能其他活动已经耗尽了可用内存。此时，模块通常会重新调用 bufcall()。

17.7.1　扩展 STREAMS 缓冲区

有些 STREAMS 驱动程序支持有双端口 RAM(dual-access)(又称双端口 RAM)的 I/O 卡。这种卡的内存缓冲区既可以由设备硬件访问，又可以由 CPU 访问，这个缓冲区可以被映射到内核或用户地址空间中，使应用程序不必通过主内存就能访问和修改其内容。

STREAMS 设备将数据放入消息并向上传输，为了避免复制 I/O 卡缓冲区中数据，STREAMS 提供了一种直接将它们作为消息数据缓冲区的方法。驱动程序调用 esballoc()，而不是 allocb()，传给它要使用的缓冲区地址。STREAMS 分配一个 msgb 和一个 datab(从 mdbblock 中)，但没有数据缓冲区，它使用调用者提供的缓冲区，并调整 msgb 和 datab 中的域引用该缓冲区。

缓冲区释放时会遇到一些麻烦。通常，模块调用 freeb()或 freemsg()时，内核释放 ms-gb，datab，和数据缓冲区(假设没有其他的 datab 引用)。kmem_free()例程释放这些对象，并恢复内存。但是，驱动程序提供的缓冲区由于是属于 I/O 卡的不能释放到内存池中。

因此，esballoc()提供了另外的参数，一个释放处理函数(release handler)的地址。当消息被释放时，内核释放 msgb 和 datab，然后调用释放处理函数释放数据缓冲区。处理程序采取必要的步骤将缓冲区标志为可用，以使 I/O 卡重用该缓冲区。esballoc()的语法是

mp = esballoc(base，size，pri，free_rtnp);

其中 base 和 size 是用来说明所使用的缓冲区的，free_rtnp 是释放处理函数的地址。pri 参数只是为了保持兼容，在 SVR4 中已不再使用。esballoc()返回一个指向 msgb 的指针。

17.8 多路复用

STREAMS 提供了一种称作多路复用的机制，它可以使多个流与同一个流相连接，这一个流被称作多路复用器，多路复用只限于驱动程序，不支持模块。有三种多路复用器——上部，下部，和双向的。上部多路复用器在顶部与多个流相连，底部与一个流相连。它也被称作扇入，或多对一多路复用器。下部多路复用器，也称作扇出，或一对多多路复用器，其在底部与多个流相连，顶部只连一个流。双向，或多对多多路复用器的顶部和底部都与多个流相连。多路复用器可以以任意方式组合，形成很复杂的配置，如图 17-13 所示。

图 17-13 一个上部多路复用器

STREAMS 提供了支持多路复用的例程和框架，但还是由驱动程序负责多个流的管理及其数据选径工作。

17.8.1 上部多路复用器

打开同一驱动程序的多个次设备就能产生上部多路复用效果。除了在 open 和 close 系统调用中的一些处理，STREAMS 对此没有提供特殊的支持。所有支持多个次设备的驱动程序都是一个上部多路复用器。

某个 STREAMS 设备第一次被用户打开时，内核创建一个流，并通过 s 节点和公共 s 节点引用它。之后，如果后一个(或同一个)进程打开同一设备文件，内核会发现该文件已打开过(已建了一个流)，因此新进程就会使用同一个流。如果新进程打开另一个具有相同主次设备号的设备文件，也一样处理。内核会发现这个设备已经被打开(因为要通过同一个公共 s 节点)，因此两个进程共享同一个流。到目前为止，我们还没有用到多路复用。

但是，如果第二个进程用的是另一个次设备号，内核会为它创建另一个流，以及另外的 s 节点和公共 s 节点。因为两个流有相同的主设备号，它们共用同一个驱动程序。驱动程序的 open 过程将被调用两次，每次打开一个流。驱动程序还要管理两套队列，可能其中的模块各不相同。数据从设备上到达后，驱动程序检查数据，并决定由哪个流接收。这个决定往往是依据数据里的信息，或者数据到达的控制卡端口作出的。

STREAMS 没有为上部多路复用器提供特殊的支持。驱动程序内的数据结构可以记录与它相连的各个流，它存储子指向各个读队列的指针，因此可以向任意一个流发送数据。因为 STREAMS 不提供适用于多路复用器的流量控制，它需要完成自己的流量控制。

17.8.2 下部多路复用器

下部多路复用器驱动程序是一个伪驱动程序，它并不控制物理设备，而是与下面的一个或多个流相连。要建立这样的配置，用户需要先创建上、下部的流，然后将上部的流与每一个下部流相链接。STREAMS 提供了称作 I_LINK 和 I_UNLINK 的专用命令，来建立和拆除下部多路复用器。

假如一个系统中既有以太网卡又有 FDDI 网卡，每张网卡有自己的驱动程序。系统可以将 IP 层作为一个多路复用驱动程序连到两个网络接口上。例 17-2 说明了这种配置的创建方法。

实例 17-2　创建一个下部多路复用器

```
fd_enet＝open("/dev/enet", O_RDWR);
fd_fddi＝open("/der/fddi", O_RDWR);
fd_ip＝open("/dev/ip", O_RDWR);
ioctl(fd_ip, I_LINK, fd_enet);
ioctl(fd_ip, I_LINK, fd_fddi);
```

这个例子中省去了检查返回值和处理错误的语句。前三条语句分别打开 enet(以太网)fddi，和 ip 驱动程序，然后，用户将 ip 流分别链接到 enet 流和 fddi 流上。图 17-14 显示了最终的配置。

下一节将详细讨论建立下部多路复用器的过程。

图 17-14　一个作为下部多路复用器的 IP 驱动程序

17.8.3　链接流

一个下部多路复用器驱动程序必须提供两个队列对，比普通 STREAMS 驱动程序多一个。这两个队列对分别称作上队列对和下队列对。在多路复用器的 streamtab 结构中，st_rdinit 和 st_wrinit 域引用的 qinit 结构是上队列对，st_muxrinit 和 st_muxwinit 域指向下队列对。在这些队列中，只有一部分过程是有用的。上读队列必须有 open 和 close 过程，下读队列必须有 put 过程，上写队列也一样。所有其他过程都是可选的。

图 17-15 描绘了 I_LINK 命令以前的中和 enet 流的情形。strdata 和 stwdata 被所有流头共享，并且分别有读写 qinit 结构。只有 ip 驱动程序有一个下队列对，还没有使用。

图 17-15　链接前的 ip 流和 enet 流

现在，让我们来看看当用户发出第一条 I_LINK 命令后的情况。strioctl()例程完成所有 ioctl 请求的初始化操作。对于 I_LINK，它完成下列工作。

1.检查上部和下部的流是否合法，以及上部流是否为多路复用器。

2.检查是否存在循环，如果一个下部流直接或间接地接入上部流的流头，就会形成循环。STREAMS 会放弃任何可能导致循环的 I_LINK 调用。

3.把 enet 流头中的队列改为指向 ip 驱动程序的下队列对。

4.将 enet 流头中的 q_ptr 域清零，不再指向自己的 stdata 结构。

5.创建一个 iinkbtk 结构，包含指向要被链接的队列的指针。这里有一个 q-top，指向中驱动程序的写队列，和一个 q_bot，指向 enet 流头的写队列，linkblk 还包含一个链接 ID，以后将用于路径选择。STREAMS 为每个连接生成一个唯一的链接 ID，并将它作为 LLINKioctl 的返回值传给用户。

6.将 linkblk 放到一条 M_IOCTL 消息中向下传给 IP 驱动程序，并等待返回。

图 17-16 显示了 I_LINK 完成后的连接。粗箭头表示由 STREAMS 建立的新连接。

图 17-16　链接后的 ip 流和 enet 流

ip 驱动程序负责多路复用器配置中的其他细节工作。它维护着一些数据结构，这些数据结构描述了它下面所连接的所有的流。因此，当它收到那个 M_IOCTL 消息以后，会将 enet 流的项添加进去。该项至少必须包含下队列指针和链接 ID(从消息中的 linkblk 结构得来)，以便它能将消息向下传递。下面，我们介绍多路复用器的数据流。

17.8.4　数据流

fddi 流链接 ip 驱动程序的方法与 enet 流相同。ip 驱动程序收到另一个 M_IOCTL 消息，就加入 fddi 流的一项。一旦这种配置建立起来，就必须能够为收发消息正确地选径。

用户向下发送数据时,ip 驱动程序必须判断它是发给以太网接口的还是 FDDI 接口的。如果两个接口处于不同的子网，这个决定可能是基于目标 IP 地址的。然后，它从自己的私有数据结构中找出相应流的写队列地址，调用 canput()检查该流是否能接收数据。如果能，通过

调用队列的 put 过程向下传递消息。

STREAMS 建立的通道可以完成向上的路径选择。数据从以太网或 FDDI 到达后，驱动程序将其向上传递。数据到达流头后，由流头的读队列的 put 过程处理。但是，这个队列现在指向多路复用器下读队列的 qinit 结构。因此，消息被 ip 驱动程序处理，并向上直到传递给 ip 流头。

STREAMS 不直接支持多路复用器的流量控制。囤此，ip 驱动程序必须处理所有必要的流量控制。

17.8.5 普通链接和持久链接

通常，一个链接会保持到流的最后一个实例关闭，或者，用户可以通过 I_UNLINK ioctl 命令显式地解除多路复用器下的一个链接，当然，要传给它由 I_LINK 返回的链接 ID。

对于一个多路复用器配置，高层驱动程序的流是该配置的控制流。对于有多个多路复用器的多层配置，每个多路复用器在每一层中的控制流一定链接到另一个更高层的多路复用器之下。如果流是以这种方式正确设置的，最顶层控制流的最后的 close 操作会拆除整个配置。

假设有一个像图 17-3 中那样的复杂配置，我们希望这样的配置只建立一次，然后一直供应用程序使用。要做到这一点，一种办法是使用一个守护进程，由它打开和链接所有的流，并插入所有需要的模块。然后，该守护进程无限期地阻塞，保留一个控制流的打开描述符，这样能防止在其他进程不再使用控制流时拆除整个配置。

现在通过向顶层驱动程序(图 17-3 的例子中是 TCP 或 UDP)发 open 调用，其他进程就可以使用这个配置了。通常，打开的是一些克隆设备，因此，打开操作只是为同一个驱动程序创建一个新次设备号和相应的新流。

这个方法需要一个进程专门用于保持流的打开状态。如果该进程意外中止，系统就没有办法了。通过用 I_PLINK 和 I_PUNIANK 命令代替 I_LINK 和 I_UNLINK，系统提供了另一种解决方案。I_PLINK 创建的是持久链接，即使没有进程打开流，这个链接也会一直保持下去。这种链接必须要用 I_PUNLINK 命令显式地删除，当然，得传递一个由 I_PLINK 命令返回的链接 ID。

17.9 FIFO 和管道

STREAMS 的一个优点就是它提供了一种实现 FIFO 文件和管道的简单途径。第 6.2 节从进程间通信的角度介绍了 FIFO 和管道。本节将介绍 SVR4 中是如何实现这些对象的，以及这种方法的好处。

17.9.1 STREAMS FIFO

FIFO 文件也被称作命名管道,用户通过 mknod 系统调用来创建一个 FIFO,传给它路径名，权限，和 S_IFIFO 标志。这个文件可以位于任何普通文件系统——像 s5fs 或 ufs——的目录下。一旦创建成功，只要有相应的权限，所有知道其文件名的进程都可以读写该文件。文件会一直存在，直到通过 unlink 系统命令将其显式删除。到文件的 I/O 遵从先进先出的规则。因此，已被打开，文件就可以像管道一样被使用。

SVR4 使用了一个独立的称作 fifofs 的文件系统来处理所有 FIFO 文件上的操作。它利用一个具有回路(loopback)驱动程序的流来实现这个功能。当用户调用 mknod 来创建 FIFO 时，内核解析路径名，取出 v 节点的父路径。然后，它在该父路径上调用 VOP_CREATE 操作，在该路径下创建这个文件。它将 i 节点中的 FIFO 标志设上，表示该文件是一个 FIFO 文件。

图 17-17 描绘了 SVR4 是如何建立 FIFO 的。要使用 FIFO，进程首先必须打开它。open 系统调用看到 FIFO 标志时，会调用 specvp()，再由 specvp()调用 fifofs 例程 filovp()。这个例程创建一个 fifonode，非常类似于 s 节点。包含在 fifonode 中的 v 节点指向 fifo[s 操作的向量。fifofs 将这个 v 节点返回给 open 调用，因此未来所有对文件的引用都将使用

fifofs 操作。

open 系统调用接着调用新 v 节点上的 VOP_OPEN 操作。这导致一个对 fifo_open()例程的调用。因为这个文件是第一次打开，还没有与之相关联的流，fifo_open 创建一个新流头，然后简单地将它的写队列和它的读队列相连。它将指向流头(struct stdata)的指针存储到 v 节点的 v_stream 域中。以后再打开时，fifo_open()会发现流已经存在，因此所有用户都会共享这个流。

图 17-17 基于 STREAMS 的 FIFO

任何时候用户向 FIFO 写数据，流头都会将它向下发送给写队列，写队列又会立即返回给读对列，等待被读取。读取进程从读队列中获取数据，如果没有数据就会在流头处阻塞。所有打开 FIFO 的进程都退出后，流就被删除了。如果 FIFO 再次打开，流还会重建。FIFO 文件本身一直存在，直到用显式地命令将其删除。

17.9.2 STREAMS 管道

pipe 系统调用创建了一个无名管道。在 SVR4 之前，管道中的数据流是单向的。pipe 调用返回两个文件描述符，一个是读的，一个是写的。SVR4 利用 STREAMS 重新实现了管道。这种新方法使双向管道成为可能。

在 SVR4 中，与以前相同，pipe 也是返回的两个文件描述符。但是，这两个文件描述符都是可读可写的，向一个描述符写入的数据从另一个描述符中读出，反之亦然，这是通过使用两个流来实现的。pipe 系统调用创建两个 fifonode，并为每个 fifonode 建一个流头。然后，它将每个流头的写队列与另一个的读队列相连。图 17-18 描绘了最终的配置情况。

图 17-18 基于 STREAMS 的管道

这种方法具有很多很重要的优点。管道现在变成双向的了，比以前更有用。很多应用程序需要进程间双向通信。在 SVR4 之前，这只能通过打开和管理两个管道的方式来实现。而且，通过流来实现管道可以完成很多控制操作。比如，它允许通过无关的进程来访问管道。

C 库例程 fattach 提供了这个功能[Pres 90]。其语法是

```
error=fattach(fd, path);
```

其中 fd 是与某个流相联系的一个文件描述符，path 是由调用者所拥有的文件路径名(否则调用者必须是 root)。调用者必须对文件有写权限。fattach 使用一个称作 namefs 的特殊文件系统，将这个文件系统的一个实例安装到由 path 所代表文件上。不像其他文件系统只能安装到目录，namefs 可以安装到普通文件上。安装时，它将这个流文件描述符绑定到安装点。

一旦这样绑定后，所有对那个路径名的引用都会访问相应的流。这个关系一直保持到被 fdetach 删除，那时路径名又重新与相应的普通文件绑定。通常，fattach 用来将管道的一端绑定到一个文件名上。这使应用程序可以创建一个管道，然后动态地与一个文件名相关联，因此为无关进程提供了访问这个管道的方法。

17.10 网络接口

STREAMS 为 System V UNIX 中的网络提供了内核子结构。程序员需要一个编写网络应用程序的高层接口。1982 年于 4.1cBSD 中引入的 socket 框架为网络编程提供了全面的支持，System V UNIX 是通过一套基于 STREAMS 的接口来解决这个问题的。这包括传输提供者接口(Transport Provider Interface，TPI)，定义传输供应者与传输用户之间的交互操作和传输层接口(Transport Layer Interface，TLI)，为高层编程提供了方便。因为 socket 框架的出现远远早于 STREAMS，有很多的应用程序中都用到了它。为了便于这部分应用程序的移植，SVR4 通过一组库和 STREAMS 模块，增加了对 socket 的支持。

17.10.1 传输供应者接口(TPI)

传输供应者呈一个实现 OSI 协议栈[ISO 84]中第 4 层(传输层)的模块，像 TCP。传输用

户是一个应用此模块的应用程序，像文件传输协议。TPI 是基于 STREAMS 的，并且定义了控制传输供应者与传输用户间交互操作的消息格式和内容。

TPI 消息既可以源于应用程序，也可以源于传输提供者。每条消息存放在 M_PROTO 或 M_PCPROTO 类型的 STREAMS 消息块中。消息的第一个域是其 TPI 消息类型。这个类型决定了消息中其他部分的格式和内容。例如，应用程序发出一个 T_BIND_REQ 消息来将流绑定到一个端口。它包含了流必须被绑定到的端口号，以及其他特定的参数。传输供应者用一条 T_BIND_ACK 消息作应答，其中包含了操作的结果。

类似的，TPI 定义了上下行发送数据的格式。应用程序创建一条以 T_DATA_REQ 或 T_UNITDATA_REQ 块做头的消息，后而跟着一个或几个包含消息体的 M_DATA 块。头中包含有特定协议的目标地址和端口号。当数据从网络上到达时，传输供应者生成一个头，也就是一个 T_DATA_IND 或 T_UNITDATA_IND 消息块②。TPI 的作用就是标准化传输供应者与传输用户间的操作。例如，不管是 TCP 还是 UDP 连接，应用程序使用相同的消息来绑定端口。这样可以保证高度的传输独立性。但是，TPI 并没有提供一个简单的编程接口。这是由 socket 和传输层接口负责的。

17.10.2 传输层接口(TLI)

TLI[AT&T 89]出自于 System V UNIX，于 1986 年引入 SVR3。它提供了一个打开和使用网络连接的过程式接口。TLI 函数特别适用于客户/服务器体系结构。这些函数既可用于面向连接的服务，也可用于面向非连接的服务。从内部来说，它们是作为 STREAMS 操作来实现的。

在面向连接的协议中(图 17-19)，服务器用 t_open()调用打开一个传输站点，然后通过 t_bind()将它绑定到一个端口。之后，它调用 t_listen()，将自己阻塞住，直到某客户请求连接，同时，一个客户程序，通常是在另一台机器上，调用 t_open()和 t_ind()，然后用 t_connect()调用与服务器建立连接。客户端在 t_connect()上阻塞，直到连接建立。

连接请求到达服务器机器后，t_listen()返回，然后服务器调用 t_accept()来接受连接。这个过程就是向由 t_connect()返回的客户端发一个回答。现在，连接就建立起来了。服务器设置一个循环，调用 t_rcv()宋接收客户请求，并进行处理，然后调用 t_snd()返回应答。客户端调用 t_snd()发请求。调用 t_rcv()接收应答，

面向非连接的协议的操作过程有所不同(图 17-20)。服务器和客户都调用 t_open()和

图 17-19 用于面向连接协议的 TLI 函数

t_bind()，和前面一样。因为不用建立连接，不需要 t_listen()，t_connect()，或 t_accept()。服务器设置一个循环，调用 t_rcvudata()，由它阻塞进程直到收到客户端消息。消息到达后，t_rcvudata()将发送者的地址和端口号返回给服务器，同时返回的还有收到的消息。服务器处理完消息后，用 t_sndudata()向客户端发应答。类似的，客户端也是调用 t_sndudata()发消息，用 t_revudata()收应答。

17.10.3 sockets

sockets[leff 86]于 1982 年引入 4.1BSD，提供了一个编程接口，可以用于进程间或网络通信。一个 socket 是一个通信端点，代表了一个用来收发消息的抽象对象。虽然 socket 不是源于 System V UNIX 的，但 SVR4 还是提供了全部 BSD 功能，以支持大量使用 sooket 接口编程的应用程序。

在很多方面，socket 接口类似于 TLI。socket 和 TLI 函数几乎可以一一对应。表 17-1 列出了常用的 TLI 函数和相应的 socket 调用。

图 17-20 用于无连接协议的 TLI 函数

表 17-1 TLI 和 socket 之间的对应关系

TLI 函数	Socket 函数	目的
t_open()	socket()	打开一个连接端点

t_bind()　bind()　把端点绑定到端口
t_listen()　listen()　等待一个连接请求
t_connect()　connect()　发送一个连接请求
t_accept()　accept()　接收一个连接
t_rcv()　recvmsg()　接收连接消息
t_snd()　sendmsg()　发送连接消息
t_rcvudata()　recvfrom()　从任意节点接收消息
t_snduata()　sendto()　给指定节点发送消息

但是，TLI 和 SOCket 调用之间在参数和语义上还是存在实质性的区别。虽然 TLI 和 STREAMS 被设计成互相兼容，要加入 socket 对 STREAMS 的支持[Vess 90]还有一些问题。让我们来看一看导致两种体系结构间不兼容的几个重要因素。

socket 框架是过程式的，不是基于消息的。当应用程序调用一个 socket 函数时，内核通过直接调用底层传输函数将数据发送到网络上。它通过查表找到特定传输协议的函数，并将数据传给这些函数。这使高层 socket 接口可以通过全局数据结构与传输层共享状态信息。在流中，各个模块彼此独立，不存在全局状态。虽然这种模块化的框架有很多优点，它很难实现那些基于状态共享的 socket 功能。

socket 调用在调用进程的上下文中执行。因此任何错误都可以同步地通知调用者。STREAMS 是异步处理数据的。而且像 write 或 putmsg 这类调用，只要将数据拷入流头就成功返回。如果在一个底层模块中发生错误，只会影响后继的写请求，产生错误的调用还是被当作成功完成了。

有些问题关系到是否能在 TPI 上实现 socket。有些功能在 socket 与在 TPI 中实现的地方不一样。例如，socket 应用程序是在 socket 被打开并绑定之后，再在 listen()调用中指定最大未接受连接指示 backlog。但是，TPI 要求在绑定操作过程中所发的 t_BIND_REQ 消息中指定。

17.10.4　SVR4 Socket 实现

图 17-21 说明了 SVR4 中 socket 的实现。socket 功能是由一个称作 socklib 的用户库和一个称作 sockmod 的 STREAMS 模块联合提供的。这两者相互补充对方的功能。socklib 库将 socket 函数映射到 STREAMS 系统调用和消息上。sockmod 在 socklib 与传输供应者的互操作中起调节作用，并支持 socket 独有的语义。

图 17-21　SVR4 中的 Socket 实现

当用户调用 socket 创建一个 socket 时，socklib 利用 SVR4 网络选择设施[AT&T 89]将参数映射成设备文件名。它打开这个文件，创建一个流，然后利用 I_PUSH ioctl 将 sockmod 模块插到流头之下。一旦这样配置好，sockmod 和 socklib 就可以一起处理用户请求了。

例如,想想当用户调用 connect 将一个 TCP 流与远程服务器相连时,会发生什么。socklib 创建一条 t_CONN_REQ TPI 消息，并通过调用 putmsg 向下发送。然后，它调用 getmsg 来等待应答。sockmod 截取这条消息，并将其中的目标地址存储起来，以备后用，然后将消息传给 TCP，TCP 模块处理这个请求，然后向上发送一个确认。getmsg 返回后，socklib 从消息中提取结果，然后将控制权返回给应用程序。

之后，用户可能通过调用 sendmsg 向连接发消息。soeklib 再一次调用 putmsg 将其向下传送。sockmod 收到消息后，生成一个包含目标地址的头，目标地址是在建立连接时记录下来的。

socktib 和 sockmod 都需要维护一些 socket 的状态。例如，连接建立起来之后，socklib 要记录连接的状态和目标地址。这样,它就可以拒绝对未建立连接的 socket 的 sendmsg 调用。如果只有 sockmod 维护了连接信息，就不能给调用者返回正确的出错状态。

类似的，仅在 scсklib 中维护状态信息是不够的。因为进程可能会创建一个 socket，然后调用 exec，消除掉 socklib 中维护的所有的状态信息。因为 exec 要将 socklib 初始化成某个已知状态。当用户想在 exec 后使用 socket 时，socklib 向 sockmod 发一个 ioctl，恢复它失去的状态值。因为 sockmod 位于内核空间，它的状态不会因为 exec 调用而被清除。

关于 SVR4 socket 实现还有很多有趣的话题和问题。这些在[Vess 90]中有详细讨论。

17.11 小　结

STREAMS 提供了一个编写设备驱动程序和网络协议的框架，它保证了高可配置性和模块化。STREAMS 之于驱动程序就象管道之于 UNIX shell 命令，它使各个模块的编写独立起来，各模块都像一个过滤器，完成特定的数据流处理任务。用户可以随意组合这些模块形成不同的流。这些流像一个双向管道，在应用程序和设备或网络接口间传递数据，中间插入适当的数据处理。

模块化的设计使网络协议可以以层次化的方式实现，每层包含在一个独立模块中，STREAMS 还用于进程间通信，SVR4 中用流实现了管道和 FIFO。最后，很多字符驱动程序，包括终端驱动程序子系统，都已经用 STREAMS 驱动程序的方式重写了。一些最近对 STREAMS 功能的增强包括多处理器支持[Garg 90，Saxe 93]。

17.12 练　习

1.为什么 STREAMS 把 msgb 和 datab 数据结构分开，而不放在一个独立的缓冲区头中?

2.STREAMS 模块和 STREAMS 驱动程序间的区别是什么?

3.模块中两个队列是什么关系?它们的功能必然相似吗?

4.存在或不存在 service 过程对队列有何影响?

5.read 和 getmsg 系统调用都可以从流中获取数据。它们之间的区别何在?各适用于哪种情况?

6.为什么大部分 STREAMS 过程不允许被阻塞?如果 put 过程不能立即处理消息,它会怎么做?

7.为什么使用优先带?

8.解释队列何时及如何允许回访。

9.流头提供了哪些功能?为什么所有的流头共享一套例程?

10.为什么 STREAMS 驱动程序要通过 cdevsw 表访问?

11.为什么 STREAMS 设备需要特殊的 ioctl 支持?为什么在一个流上只有一个活动 ioctl?

12.在内存短缺时，为什么丢弃到达的网络信息是明智的?

13.多路复用器和分别用于两个流中的同一模块有何区别?

14.为什么 STREAMS 不为多路复用器提供流控制?

15.在图 17-14 中，为什么 IP 层作为 STREAMS 驱动程序而不是模块实现?

16.持久链接的好处是什么?

17.STREAMS 管道支持双向通信，而传统管道不支持。举出一种可从中受益的应用程序。没有 STREAMS 管道如何提供这种功能?

18.TPI 和 TLI 的区别何在?它们各自与什么样的交互操作相关?

19.作为编写网络应用程序的框架，比较 socket 和 TLI，它们各自的优缺点是什么 7 哪些特性在另一方不容易实现?

20.第 17.10.4 节介绍了 SVR4 如何在 STREAMS 上实现 socket 的。基于 BSD 的系统是否可以用 socket 实现 STREAMS 似的接口?有什么重大问题要解决?

21.编写一个 STREAMS 模块，将所有向上传递的“新行”字符转换成“回车-换行”；对于向

下传递的字符反过来。假设消息中只有可打印的 ASCII 字符。

22.用户可以通过向栈中插入模块来动态配置流。各个模块并不知道它上下都是什么模块。那么,一个模块怎么知道如何解释由邻近模块发来的消息?对于能将哪些模块以何种次序放到一起，STREAMS 系统有什么限制?TPI 如何解决这个问题的?

① 除了多路复用器(见 17.8.3 节)。

② T_UNITDATA_REQ 和 T_UNIXDATA_IND 都是用于数据报的类型,而 T_DATA_REQ 和 T_DATA_IND 都是用于字节流数据的类型。