Windows NT File System Internals
第一部分
第二章　文件系统驱动开发

本书的重点是 NT 平台的内核模式的文件驱动程序和过滤驱动程序，因此在开始讨论怎样设计和实现一个内核模式的文件驱动程序或者过滤驱动程序之前，需要明白文件系统和过滤驱动能做什么，知道他们能做什么和不能做什么能帮助你决定是否值得克服重重困难去设计一个。
在这一章里，我们简要讨论一下各种类型的文件系统驱动和过滤驱动，使你对他们的功能有一个大致的了解。还要讨论一些在设计和实现 NT 内核驱动时要用到的通用的概念。包括怎样使你的驱动是可分页的，怎样分配和释放执行时需要的内存，怎样使用一些系统定义的结构和函数创建可链接的列表，和怎样定位驱动的错误和排错。你可以跳过这些原理性的东西，然后当你阅读了其他章节后或者开始设计和开发你的文件系统驱动和过滤驱动的时候再回来查阅。
我试图设计一个文件驱动的时候面临的一个挑战是怎样理解用户相关的文件名是怎样处理的（treat）。我们将在NT对象管理器管理的名字空间上讨论这个问题。还要讨论Multiple Provider Router (MPR)组件和Multiple UNC Provider (MUP)组件在支持网络文件系统驱动中担当的角色，必须和本地接点上的名字空间结合。随后在章节要详细的说明这里提到的一些主题。
什么是文件系统驱动？
文件系统驱动是存储管理子系统的一个组件。为用户提供在持久性介质上存贮和读取信息的功能。

文件系统驱动提供的功能
文件系统驱动通常为用户提供以下功能：
　　创建，修改和删除文件
　　安全，可控制地在用户之间共享和传输信息
　　以适当的方式向应用程序提供结构化的文件内容
　　用文件的逻辑名字而不是设备特定的名字来表示存储的文件
　　提供文件数据的逻辑视图而不是设备相关的视图
以上功能被所有商业化的本地文件系统实现所支持。另外，远程文件系统，包括网络和分布式的，提供下面的功能，提供的能力范围（程度），依赖于使用的文件系统的哲学：
　　网络透明
　　位置透明
　　位置无关
　　移动的用户
　　移动的文件
不是所有以上功能被所有的远程文件系统实现支持，但是，随着文件系统技术的发展和进步，越来越多的网络文件系统达到或者超过了这些目标。

文件系统驱动的类型
你能够设计，实现和安装不同种类的文件系统驱动，包括本地文件系统，网络文件系统和分布式文件系统。
磁盘（本地）文件系统
本地文件系统管理存储在直接连接到计算机的磁盘上的数据。文件系统驱动接受打开，创建，

读，写和关闭文件的请求。这些请求通常在用户进程里产生然后通过I/O管理器分发到文件驱动。图2-1显示本地文件系统怎样向用户线程提供服务。

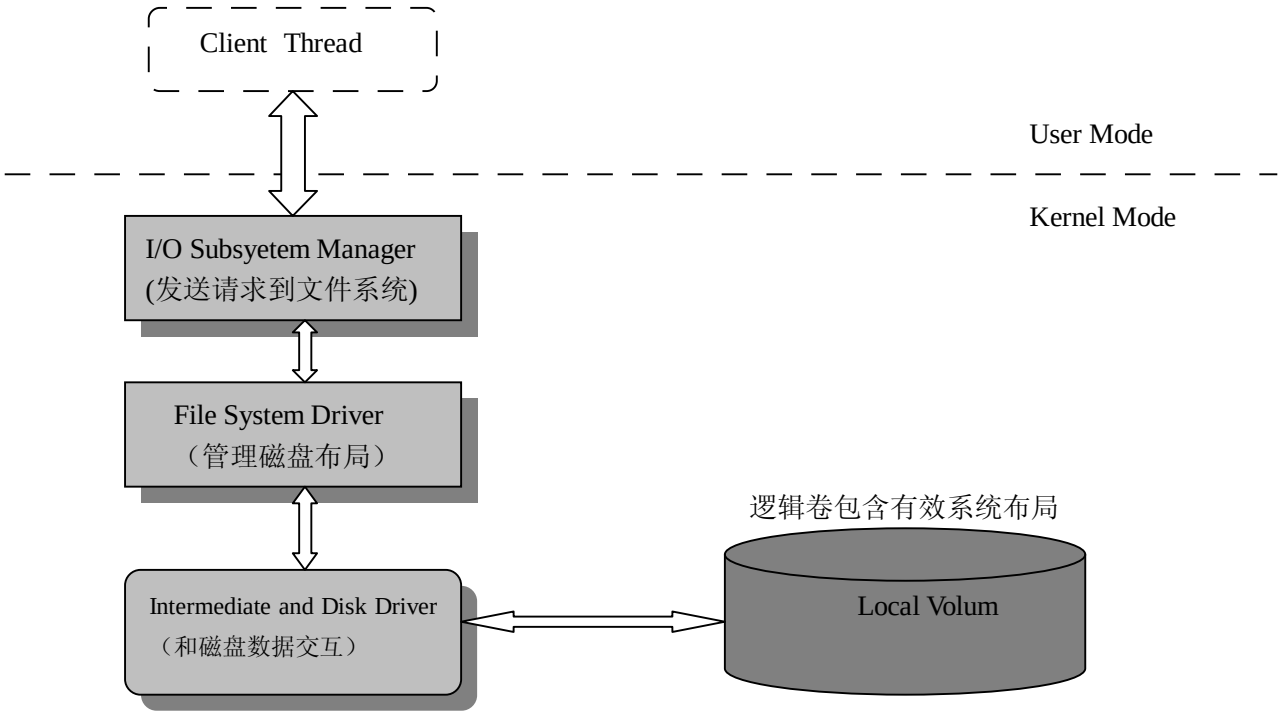

图 2-1 本地文件系统

上图中，磁盘驱动和与系统相连的逻辑磁盘交互数据，逻辑磁盘是一个简单的存储抽象，从文件系统的观点来看，这是由线性的，适当大小的的，随机访问的存储块的序列。现实中，逻辑磁盘可以是物理磁盘的一部分（分区），或者整个物理磁盘，甚至还可以是在多个物理磁盘上的分区的组合（逻辑卷）。逻辑卷管理器（*logical volume managers*）使文件系统驱动知道相临的可用磁盘空间序列，并且隐藏了把逻辑块映射到适当物理块的细节。

逻辑卷管理软件经常提供软件数据镜象，通过使用多个物理磁盘，还具有了动态调整逻辑大小的能力。所以他们一般叫做容错软件。

要被本地文件系统驱动管理，那么每个逻辑卷必须有一个有效的文件系统布局。这包括适当的文件系统元数据信息，这是特定于文件系统驱动类型的。例如，FASTFAT文件系统驱动要求的盘上布局和NTFS完全不一样。它使用的用来存储用户数据的结构和NTFS有很大的差别。

在NT中，每当使用格式化一个逻辑卷的时候，实际上是在创建文件系统元数据（用于管理）结构以便后面会被文件系统驱动用来提供如为用户存储分配空间，把存储的用户数据连接到用户特定的文件名，创建目录存放文件等功能。

当用户可以访问存储在逻辑卷上的数据之前，逻辑卷必须要先挂载到系统中。当一个卷挂载后，文件系统驱动验证元数据然后开始管理卷，使用存储在卷上是元数据建立适当的基于元数据的内存中的数据结构。

本地文件系统为每个挂载的逻辑卷提供一个单一的名字空间。大多数商业的，现代文件系统实现提供分层的，树型的布局，树型结构包括目录，包含在目录里的文件，每个目录和包含文件，有一个唯一的名字相关。可以用来构建文件名的字符集依赖于特定文件系统的实现。例如，NTFS文件系统允许的一些字符在FASTFAT文件系统中却被禁止。大多数文件系统和I/O子系统明确地禁止一些字符。如“\”被NT用来区别一个路径，因此不能做为文件名的一

部分。

图 2-2 显示本地文件系统驱动提供的层次化的文件系统名字空间。每个文件系统中的对象可以被唯一地从根目录开始用名字标识，值得注意的是每个挂载的卷有一个自己的唯一根目录作为顶层容器和相应的树结构。

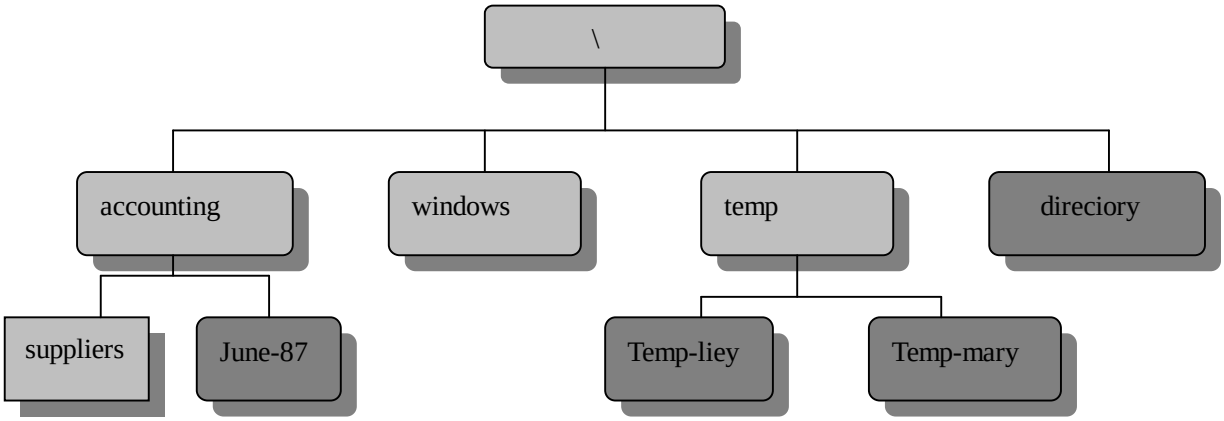

图 2-2 层次化的名字空间

一个挂载的逻辑卷的用户总是清楚她要访问的卷。如果要访问的文件不在当前挂载的卷上，就要确保文件所在的卷已挂载和是可访问的，然后就可以指定文件的完全路径（从根目录开始）来访问文件的内容了。

网络文件系统

网络文件系统允许用户向其他在局域网或者广域网范围的用户共享自己本地连接的磁盘.例如:如果你有一个物理磁盘C:在你的机器上.现在你可以允许我直接访问你的本地C:驱动器上的存储在ACCOUNTING子目录下的文件和目录.要做到这些,你和我必须使用网络文件系统.它使我能够象访问本地磁盘上的文件一样访问你的磁盘上的共享文件.

每个网络文件系统实现由两个部分组成:

客户端重定向器

必须有一个执行在我机器上的软件组件,接受我的访问存储在你的C:\ACCOUNTINT目录里的文件请求,然后通过网络传输到你的机器上处理,而且还不许从你的机器上接收处理结果数据再交给我.

共享接点上的服务器程序

当客户端的重定向器通过网络发送了一个请求,服务器必须回应请求.接下来服务器有两个主要的任务,一个是和客户端用定义良好的协议交互,二是和象客户端一样和本地文件系统交互.图 2-3显示网络文件系统客户端和服务器的实现.

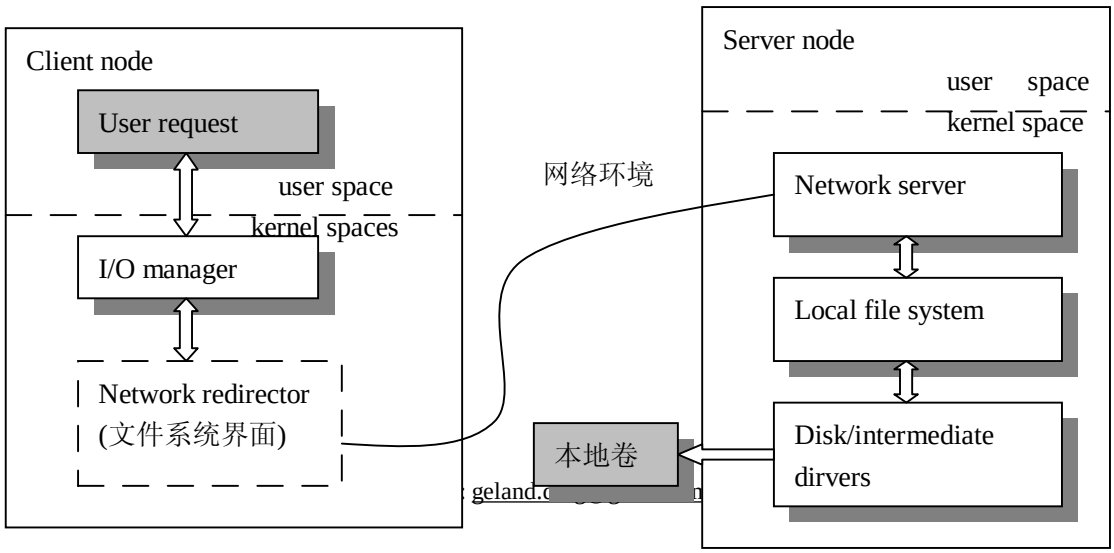

图 2-3 远程(网络)文件系统

对NT用户来说最常见的例子是LAN MANAGER NETWORK,他支持共享目录,本地卷,打印机和其他资源.LAN MANAGER NETWORK由在客户端的内核执行的LAN MANAGER REDIRECTOR组件和在服务端执行的LAN MANAGER SERVER组件. LAN MANAGER SERVER组件导出本地文件系统或者其他资源如打印机,这两个组件之间使用SMB(SERVER MESSAGE BLOCK)网络协议传递数据.

注意:MICROSOFT 在1996年向IETF提交了叫做COMMON INTERNET FILE SYSTEM(CIFS)1.0作为INTERNET 草案.还发布了CIFS的相关RFC,CIFS最近在NT4.XHE WINDOWS 95中被作为SMB协议实现,本书中SMB既是LAN MANAGER REDIRECTOR和SERVER 组件,因此可以轻易地用CIFS代替SMB.

值得注意的是冲定向器在客户端作为文件系统表现出来,这就使用户可以象请求访问本地文件系统的文件一样访问远程数据.他处理所有通过网络访问的的数据.虽然网络天生不可靠(特别是广域网),但是他有责任透明地从新建立丢失的连接,或者是返回适当的错误给用户,以便应用程序可以从新请求.

服务器不用提供文件系统类的接口,因为服务器上的客户端能够直接使用本地文件系统的服务直接访问服务器上的本地磁盘驱动器上存储的数据.

重定向器和服务器使用传输协议在网络上传输数据和命令,有许多传输协议,如TCP,UDP,和MICROSOFT的NETBIOS.传输协议可以是有连接的(如,TCP,NETBIOS),那样就在重定向器和服务器之间建立一条虚电路,或者是无连接的(如UDP/IP).

图 2-4 说明服务器怎样通过网络向客户端共享部分目录.对客户端来说,共享目录被映射(FORMS)成一个和本地卷不同的根目录.到网络卷的请求被重定向器处理,他负责通过网络把请求传送到服务器.服务器上的服务端软件处理请求,利用本地文件系统访问和操作共享卷.最后返回操作结果给客户端.

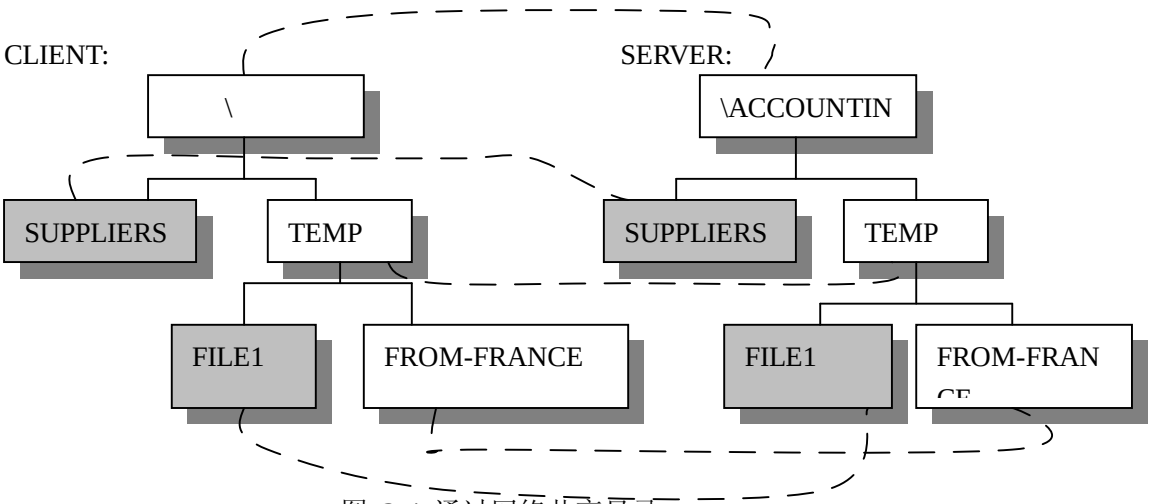

图 2-4 通过网络共享目录

在网络文件系统里,客户明白用户要访问的数据是在服务器上的,因此虽然所有的数据传输是对用户透明的,用户总是知道那些数据在本地而哪些数据是从远程服务器上得到的.

最后应该注意的是服务端的应用使用本地文件系统服务访问共享逻辑卷的文件数据.在某些

情况下,这可能导致数据一致性问题,如果文件数据同时也被客户端缓存.本地文件系统驱动通常期望和网络服务器合作来避免这种数据一致性问题.

分布式文件系统

分布式文件系统从标准网络文件系统发展迩来,用户使用单个名字空间而完全隐藏数据所在的物理位置的文件系统.这意味着用户根据单一的文件路径表示请求的文件,而不管其物理位置.用户甚至可能不知道他在访问远程服务器上的资源.

外观上,分布式文件系统很象网络文件系统,因为他们都有在客户端执行的客户端软件和在服务器上执行的使资源可以通过网络访问的服务器软件.但是,主要的不同是分布式文件系统的单一的名字空间和网络文件系统不一样.注意,客户端软件和服务端软件可以同时执行在任何参加分布式文件系统实现的接点上.

图 2-5 示意分布式文件系统怎样向用户提供单一名字空间.在接点一上的文件系统客户可以访问组成这个文件系统的所有文件而不必关心文件的物理位置.在文件系统树上有唯一(虚拟的)的全局根目录,虽然图中没有表示出..全局名字空间中的任何接点实际上可能是一个远程输出文件树的挂载点.

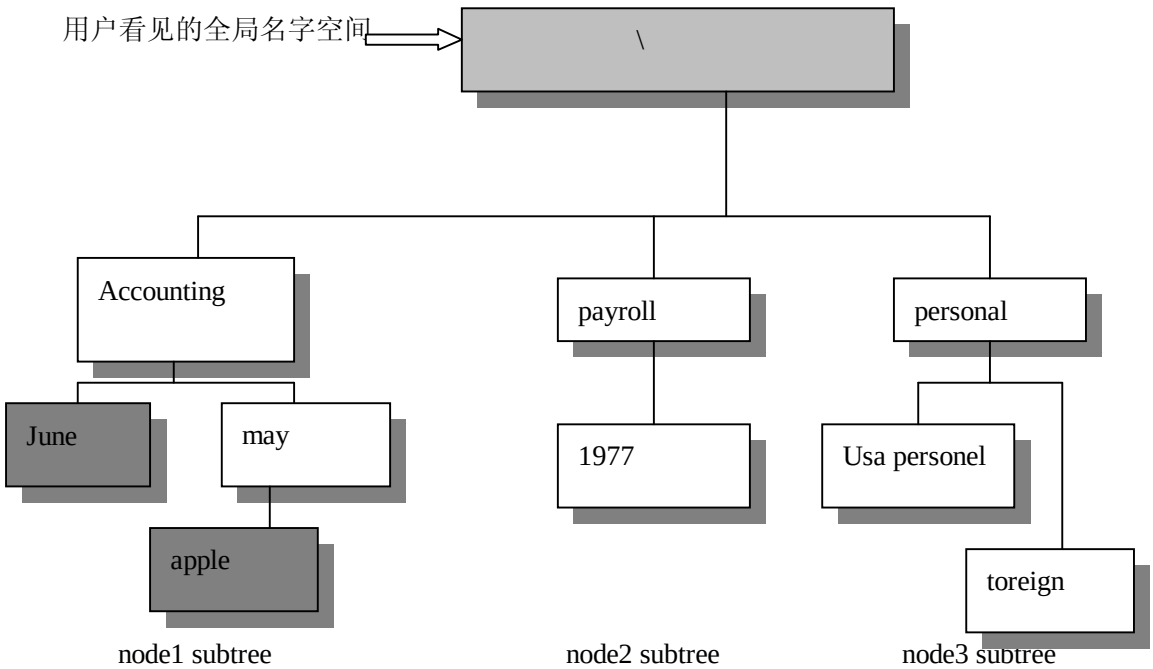

图 2-5 分布式文件系统提供的全局名字空间

注意:挂载点是对文件系统名字空间的能到达的远程暴露子目录树的一个简单命名.在上图中,可以看见Accounting, payroll, personal是分布式文件系统的挂载点.在接点1上有Accounting子目录树,从payroll可以访问接点2上的数据,从personal可以访问接点3上的数据.这个文件系统上的任何用户可以透明的访问文件或者目录而不关心其物理位置.用户看见的是一个简化的分布式文件系统的一个名字空间.

当用户试图访问一个挂载点上的数据,客户端软件必须传递请求到真正导出数据的远程服务器上,让服务器处理用户的请求.

许多分布式文件系统使用其他相似的方式访问远程存储的数据.客户端软件经常从服务器传输数据,然后在本地缓存,当用户请求先前请求过的数据的时候就不必每次和服务器交互.但是,高级的客户机-服务器缓存的保证数据在整个网络中的凝聚性的一致性处理就是必要的.

有时候分布式文件系统提供的数据一致性保证超过网络文件系统的实现.例如:分布式文件系统能够保证他的所有的用户看见的文件的内容是相同的,即使存在不同的客户接点同时的访问和修改这个文件.

特殊(伪装的)文件系统

有时,能够看见内核模式软件提供文件系统类似的接口但是在调用这些接口的时候去却做和文件系统完全不同的事情.例如,UNIX系统上的/PROC文件系统实际上允许用户访问或者修改运行进程的地址空间.

基本上,任何提供类似文件系统接口的内核模式驱动程序但是处理特殊功能的可以被看成特殊文件系统的实现.

其他特殊文件系统实现包括提供分层存储管理功能的驱动程序,或者提供虚拟文件系统的驱动程序(比如一些商用源代码管理系统).

WINDOWS NT 和文件系统驱动

文件系统是I/O子系统的组成部分,因此必须遵循NT I/O管理器定义的接口.NT I/O管理器定义了一些所有内核模式驱动程序必须遵循的标准接口.这个接口适用于本地文件系统驱动,网络和分布式文件系统重定向器软件,中间层驱动,过滤驱动,和设备驱动,文件系统驱动能够动态地加载和卸载(实际上这非常困难).

NT I/O管理器为文件系统设计人员提供广泛的支持例程.这些例程允许新文件系统在NT系统上使用通用服务和持续运行(象本地文件系统一样).而且,有定义良好的文件系统设计人员必须遵守的接口集,以便和NT虚拟内存管理器和CACHE管理器成功的进行交互.虽然相关文档很少.

使用文件系统

用户有两种方法可以利用文件系统提供服务的:

调用标准系统服务调用

这是最常见的请求访问文件和目录的方法,用户进程简单地调用标准系统服务调用来请求如打开,创建文件,读或者写文件,关闭文件.

向文件系统驱动发送I/O控制请求

有时应用程序需要请求特定的服务但是不能使用封装好的系统服务调用.这种情况下,只要文件系统能够完成期望的操作,用户可以通过文件系统控制接口(FSCTL)接口直接向驱动程序发送请求和数据.

典型的使用标准系统服务的例子是请求访问一个文件，当进程必须读取文件C:\payroll\june-97的内容是时候。应用程序进程使用WIN32子系统的操作序列如下：

1. 打开文件

   请求进程通常调用WIN32 CreateFile()服务例程，指定要打开的文件名，要打开文件的期望的访问模式，和其他相关的参数。在内部，WIN32 子系统调用NtCreateFile()系统服务调用来执行这个操作。

   在这时，CPU切换到内核模式特权等级。调用NtCreateFile()系统服务的代码由I/O管理器实现，I/O管理器是NT执行体的组成部分，运行I/O管理器实现的函数要求在内核模式特权等级。打开/创建请求在NT执行体中曲折行进，首先通过NtCreateFile()调用分派给I/O管理器，然后是NT对象管理器分析用户提供的名字，最后回到I/O管理器定位管理挂载的本地卷C：的文件系统驱动。一旦找到这个驱动，I/O管理器调用文件系统驱动的创建/打开分派入口点去处理用户的请求。

   最后，文件系统驱动执行适当的处理，然后返回结果给I/O管理器，I/O管理器再依次返回结果给WIN32子系统（切换到用户模式），子系统最后返回结果给请求的进程。

2. 读取文件数据

如果打开操作成功的话，请求进程得到一个句柄。请求进程现在要求读取这个文件中的数据，指定开始的位置和要读取的字节数。通常WIN32子系统为请求进程调用NtReadFile()系统调用来处理用户的ReadFile()函数调用。

NtReadFile()例程也由I/O管理器实现，因为请求读取数据的进程必须提供先前成功调用创建操作得到的有效的文件句柄，I/O管理器就能够轻易地定位到先前处理的打开操作的相应的内部数据结构。这个数据结构叫做文件对象（FILE OBJECT），将会在后面进行广泛的描述。根据这个文件对象结构，I/O管理器能够决定包含打开文件的逻辑卷，然后发送读取请求到文件驱动进行更多的处理。

文件系统驱动将会返回他能读到的差不多和用户请求的那样多的数据给I/O管理器，最后，结果会返回给WIN32子系统。

3. 关闭文件

一旦进程完成了对文件内容的请求，他对先前打开的文件句柄执行一个关闭操作。这个操作通知系统进程不再需要访问文件数据。进程调用WIN32 CloseHandle()函数关闭文件句柄，WIN32子系统接着调用NtClose()系统服务过程。

I/O管理器通知文件系统用户进程关闭了文件句柄，然后文件系统就会释放维护的任何与打开文件相关的状态信息。

用户请求的文件操作有许多种方式，但是基本的方法逻辑是：进程或者线程打开或者创建文件，执行一些操作，最后关闭文件句柄。注意NT系统服务对所有执行线程都可用，包括用户模式和内核模式的线程。而且NT系统服务对所有的子系统上的请求进程都是可用的。

注意：NT I/O管理器提供的系统服务例程是通用的和非常广泛的，因为他必须支持所有的NT子系统上的用户的请求，而这些请求是非常不一样的。现实中的麻烦是有一些I/O管理器提供的强大的功能常常不能被NT子系统使用（或提供），唯一的途径是直接调用期望的系统服务来使用期望的功能。但是，遗憾的是MICROSOFT没有对可用的NT系统调用提供完备的文档。

文件系统驱动提供的文件系统控制请求在后面讨论。

如果操作系统要支持多个文件系统，包括第三方开发的，那么必须在文件系统驱动和操作系统其他部分之间必须有定义良好的接口。这个接口应该清晰地规定用户的访问文件数据的请求和组件之间的各种不同的交互。相应的抽象的描述也必须提供，以便各种类型的文件系统可以成功地和操作系统之间结合。

目标应该是模块化的以便能够容易的替换和扩展，而不需要广泛的，复杂的，昂贵地重新设计整个系统。看起来I/O子系统的设计者确实达到了这个目标。因此，就有了定义良好的方法用于文件系统的安装，加载，和自我注册到操作系统。I/O管理器也发送定义非常好的I/O请求包来描述用户的请求到文件系统驱动程序。最后，还有相当多的支持例程供文件系统设计人员使用，以便减轻工作量和更好的把新的文件系统结合进操作系统中。

不幸的是，当你考虑文件系统和操作系统的各种交互的时候事情变得混乱起来了。有时，这些复杂交互的结果就是，设计人员试图从开始到结束维护这种抽象。当操作系统和文件系统协作的负责提供支持数据缓存，内存映射文件的时候变的更加糟糕。在NT中，例如：虚拟内存管理器依靠文件系统对页文件（page file）的支持来实现虚拟内存支持。但是，文件系统又依靠虚拟内存管理器来分配文件系统做处理时需要的内存。这种递归的关系使事情更加复杂。

虽然MICROSOFT的NT操作系统设计者似乎做了很大的努力来维持文件系统和操作系统的其他部分之间有清晰的界限，看起来是的，随着之间的推移，这个界限变的很模糊，而且在系统中越来越多的暗中的行为和功能变的更加顽固。这导致更复杂的设计和代码，需要更多的MICROSOFT的文档才能设计出成功的，健壮的文件系统驱动。

在本书出版前没有多少第三方开发者需要的文档，本书帮助你更好的理解系统和给你一个达到你期望 的目标的出发点。

什么是过滤驱动程序？

过滤驱动程序是一个拦截到一些已有的软件模块的请求的中间层驱动程序。依靠在请求到达目标之前就拦截请求，过滤驱动程序就有机会扩展，或者修改请求的原有接受者提供的功能。

注意：并不要求过滤驱动总是取代现有的驱动，那只是不必要的重复劳动，过滤驱动可以集中提供那些需要实现的特定的功能。也可以使现有的代码更好的执行原来提供的功能。

例如：考虑NT操作系统上存在的文件系统，包括FASTFAT文件系统，NTFS文件系统，CDFS文件系统和访问远程共享驱动器上数据的LAN MANAGER REDIRECTOR等，但是没有一个支持实时的数据加密和解密。

现在假设你是一个安全专家，知道怎样设计和实现一个惊人的加密算法，希望开发一种在用户把数据存储到磁盘之前加密用户的数据，而在一个授权用户把数据从磁盘取回的时候自动解密的软件，那么你将怎么设计呢？

你一定不想写一个全新的文件系统驱动，因为那样太浪费时间，而且也不会给用户带来什么好处。你实际上要的是设计一个过滤驱动来在下面两个位置拦截请求：

文件系统的上面

你的代码在文件系统驱动有机会得到用户请求之前拦截请求。

文件系统下面

你的代码在文件系统结束他自己的任务之后执行任何需要的处理，因此，在磁盘驱动或者网络驱动从次要存储设备或者网络得到数据之前，你的驱动可以做那些你需要做的处理。在这种情况下，可以在数据被写到磁盘或者返回给用户之前执行你的变化

图2-6说明你的过滤驱动软件可以插入的两个不同的位置

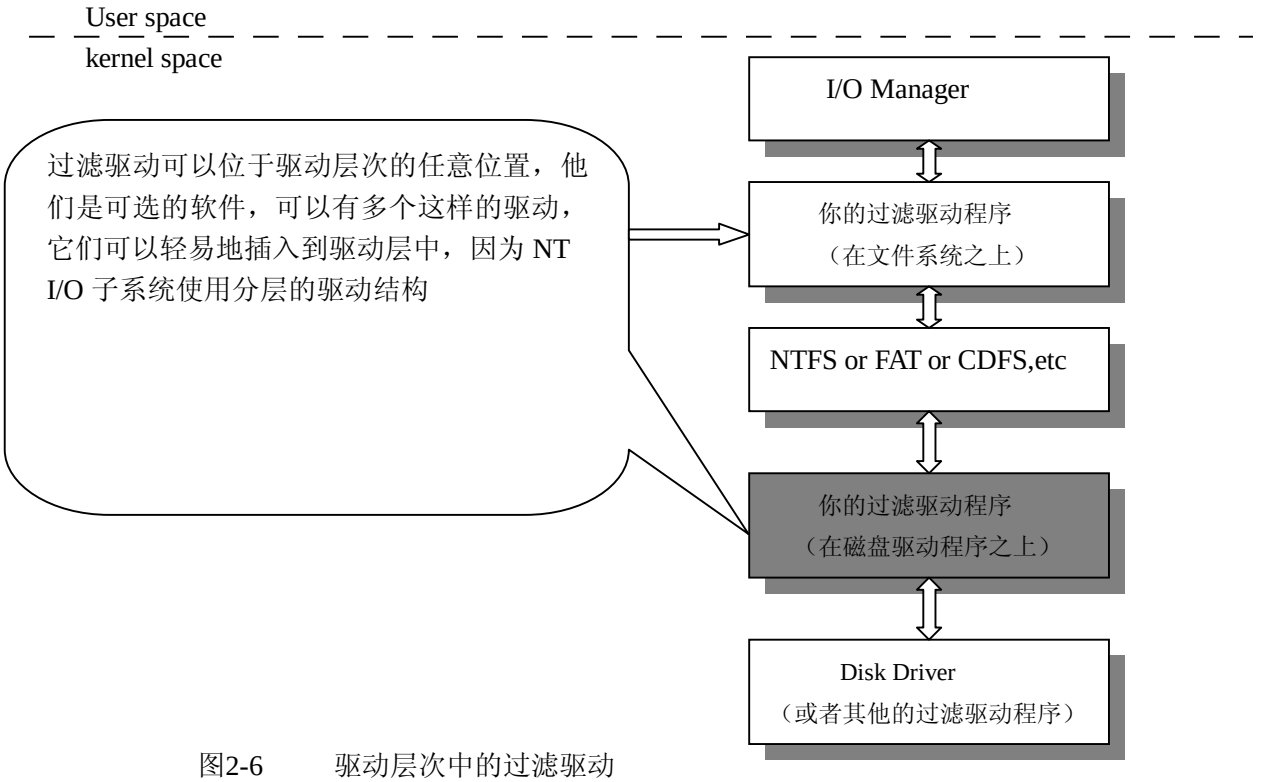

图2-6　　驱动层次中的过滤驱动

一旦你的过滤驱动插入到驱动层中的适当的位置，就可以拦截用户的I/O请求，执行你

的数据转变，然后传递请求给已有的模块（文件系统或者磁盘驱动）以便继续执行任务，比如管理逻辑卷或者从物理磁盘读数据或者写数据。

因此如果你插入了过滤驱动来拦截分派给文件驱动的I/O请求，就能够在数据传输给文件系统传输到次要存储器之前加密，在文件系统从次要存储器得到加密数据，传回给用户之前解密数据。

但是如果你决定在文件系统下面拦截请求，就需要使用下面的方法逻辑，现在只能在数据从文件系统经过，写入磁盘（或者访问网络）之前修改，或者在数据从磁盘（或者访问网络）取得，交给文件系统之前修改。

要把过滤驱动插入现有的驱动层次结构中是比较简单的，不需要重新设计现存的其他NT文件系统，磁盘，和其他中间层驱动，因为I/O子系统中的所有驱动必须遵循定义良好的分层的驱动接口。

这意味着，例如：所有的驱动必须回应I/O管理器发起的一个标准的请求集。而且有个边准的方法用于内核驱动程序请求驱动的调用层中其他驱动提供的服务。层中的每个驱动也必须以期望的方式回应I/O请求而不管调用者是谁。

注意：I/O子系统实际上并不要求所有的驱动都要实现分派例程，唯一的情况是他知道自己要响应标准I/O管理器的请求和知道自己插入驱动层中的影响。

虽然看起来你应该立即开始设计你的惊人的安全加密/解密算法，但是不幸地有些细节不的不考虑。理论上，WINDOWS NT I/O子系统是高度模块化的所以实现你的功能应该小菜一碟。现实中你必须理解一些他们之间精巧的交互。第十二章，过滤驱动程序，专门关注设计过滤驱动时的复杂的问题。

一般驱动开发问题

本书讨论许多内核文件系统和过滤驱动开发人员应该透彻理解的问题。但是还有一些一般的开发问题将在这一节简要讨论。包括怎样分配和释放内存，和怎样实现基本的调试支持。这里讨论的一些函数的详细信息请查阅DDK文档。这里使用的原料将在本书的后面定义。因此，在第一次读这本书的时候跳过这些原料然后在你至少读了第四章后回头重读。

使用核心内存

在第五章，NT虚拟内存管理器，会有NT VMM的相当详细的细节，但是，有一些相当一般的问题涉及到驱动开发和对核心内存的需求将在这里描述。本书后面出现的程序片段都假设你很清楚怎样分配和释放核心内存。

在你开始设计内核驱动的时候必须回答下面的问题：

我的驱动占有分页的或者不分页的内存？

驱动程序的代码能不能被丢弃（PAGE OUT）？

在需要的时候怎样分配内存？

怎样释放前面分配的内存？

在试图分配或者释放核心内存的时候有什么问题我必须意识到的？

可分页的内核模式驱动

默认情况下，内核加载器会加载所有的驱动的执行部分和在你驱动中定义的全局数据到非分页的内存中。因此如果你希望你的驱动程序驻留在非分页的内存中，你在编译，连接和加载驱动是时候不需要做别的事情。

此外，内核加载器一次加载整个驱动的可执行文件（和其他相关的DLL），在调用任何驱动程序的初始化例程以前。虽然这时可能不会给你什么感觉，在把执行文件加载进内存中后，内核加载器关闭可执行文件，甚至会允许用户删除当前正在执行的驱动程序影象。

允许你告诉加载器你希望你的驱动的一部分是可以分页的，可以使用下面的便宜器指令来做（这里的pragma语法引用的函数必须定义在同一编译单元中）：

```
#ifdef ALLOC_PRAGMA
#pragma alloc_text(PAGE, function_namel)
#pragma alloc_text(PAGE, function_name2)
// You can list additional functions at this point just as the two
// functions are listed above ...
•endif // ALLOC_PRAGMA
```

但是，小心，你绝不能让可能在高的IRQL级别被调用的例程被换出页面。文件系统决不能让任何代码和数据被换出页面，那样将导致向VMM发起页错误请求。

也可以让内核驱动在运行时决定那些段的代码或者数据应该被换出或者锁定在内存中。要作到这样，驱动执行下面的动作：

使一个代码段可分页，使用下面的编译指令，

```
#ifdef ALLOC_PRAGMA
#pragma alloc_text(PAGExxxx, function_namel)
#pragma alloc_text(PAGExxxx, function_name2)
fendif
```

这里的XXXX是可变的，四个字符的，驱动的可分页段的唯一标志。

使数据段可分页，使用下面的编译指令：

```
#ifdef ALLOC_PRAGMA
#pragma data_seg(PAGE)...
// Define your pageable data section module here.
#pragma data_seg() // Ends the pageable data section.
```

调用MmLockPagableCodeSection()和MmLockPagableCodeSectionByHandle( )来锁定被标志为可分页的代码段。

调用MmLockPagableDataSectionf) 和 MmLockPagableDataSectionByHandle( )锁定被标志为可分页的数据段

调用MmUnlockPagableImageSection()来解除被上面列出的函数锁定的代码或者数据段。

另外有两个VMM提供的例程应该知道，如果你想换出整个驱动或者把驱动的分页属性重设回原来的设定的话：

`MmPageEntireDriver()`

使整个驱动程序可分页，覆盖使用前面讲的编译指令修饰的段的页面属性。

`MmResetDriverPaging()`

把页面属性重设回最初描述的属性。

最后，把那些驱动被初始化后不再需要的代码自动丢弃可以使用这些编译指令：

```
#ifdef ALLOC_PRAGMA
#pragma alloc_text(INIT, DriverEntry)
#pragma alloc_text(INIT, functionl_called_by_driver_entry)
#endif // ALLOC_PRAGMA
```

小心地指定那些在驱动程序被初始化后就再也不需要的函数才可以被丢弃。

分配核心内存

每个驱动程序都需要内存来存储私有数据。通常你的驱动会向NT VMM索要内存。每当你的驱动请求内存的时候必须决定需要的是可分页的还是不分页的内存。如果的你的驱动在运行中访问内存的时候能够经受页错误，那么尽量使用可分页内存。

注意：大多数低层磁盘和网络驱动通常不能使用可分页内存，因为他们的代码常常在较高的IRQL等级执行而不允许页错误。但是，文件系统（通常比磁盘驱动占用更大，更多的

资源）有时候有机会从可分页池中分配一些内存。如果你的驱动中能够使用可分页内存，就应该尽最大的努力来标志能够被分页，在向VMM请求内存的时候指定可分页内存池的类型。非分页内存在整个系统中是一个有限的资源。虽然非分页内存的数量依赖于系统使用的类型（和系统的可用的物理内存），是用来确保某些必要的使用的。

NT提供下面的例程给内核驱动来分配内存：

```
ExAllocatePool()
ExAllocatePoolWithQuota()
ExAllocatePoolWithTag()
ExAllocatePoolWithQuotaTag（）
```

注意NT上的所有分配函数都是非阻塞的。换句话说，如果可能的话调用内存分配函数会立即返回内存，否则就返回NULL（表示分不到内存），在其他许多操作系统中（比如许多UNIX系列）内核组件允许指定是否等待直到有内存可用，或者是立即返回。

每当驱动调用函数中的一个来请求内存的时候，必须要指定请求的内存的类型：

`NonPagedPool`

这个分配请求要么返回一个指向不可分页的内存要么返回`NULL`。

PagedPool

当你的应用访问内存时候可以处理页错误的时候总应该指定这个类型。如果你在分配的内存里有任何同步结构（下一章描述）的话，决不要分配分页的内存。

`NonPagedPoolMustSucceed`

在其他方式都失败的，而你又必须立即得到内存的时候可以使用这个标志类型。注意这种类型的内存是极度缺乏的资源。可能不足`16KB`，虽然数量是可变的。如果你请求这种类型的内存（只有在其他途径都失败的时候才使用），如果`VMM`不能给你提供，将会导致系统的bugcheck，错误代码是MUST_SUCCEED_POOL_EMPTY。

`NonPagedPoolCacheAligned`

这个标志分配使用数据缓存线的尺寸来在`CPU`特定的边界对齐的非分页内存。注意这个操作默认是在Intel平台上的NonPagedPool分配类型。

`PagedPoolCacheAligned`

这个标志分配使用数据缓存线的尺寸来在`CPU`特定的边界对齐的分页内存。

`NonPagedPoolCacheAlignedMustSucceed`

再说一次，只有在最后才使用这个操作分配非分页内存。

内存池分配器初始化了一些列表，每个列表包含一种固定大小的块。当你使用上面的ExAllocate-Pool ( )函数中的一个请求内存时，支持例程试图分配一个和你请求的数量相近的（相等或者更大一点）固定大小的块。但是，如果你要求的数量超过一页，或者超过可变列表中的最大的块的大小，又或者在预先分配的列表中没有可用的块的话，VMM就会从任何适当类型的系统可用的内存中分配你请求的数量的内存给你。

注意：当预先分配的列表空了的时候，VMM会分配至少一页的内存，切分，然后把剩下的数量（在返回给调用者之后）放进适当的块列表中。不幸的是，当你请求的非分页内存的时候当请求的数量超过PAGE_SIZE的时候，内存池分配例程不会切分未使用的部分。这会浪费宝贵的非分页内存，这也是应该极力避免使用这种类型内存的另一个原因。

注释：1，第五章会详细讨论VMM处理页错误的细节。还要解释为什么内核驱动在高IRQL的时候不能发生页错误。

2，NT VMM使用一个私有算法来计算非分页内存的大小。这个算法使用系统中总的物理内存为因子来计算。VMM试图把非分页内存的数量的数量增加到预先计算的最

大值。最后，本来初始化的时候都是地址相邻的非分页池会变成碎片，而且VMM在扩大的时候也不保证是地址相邻的。

如果请求的类型的内存没有可用的，VMM会返回NULL给调用者或者bugcheck（如果从must-succeed池中分配）。

你的驱动也可以分别使用MmAllocateNonCachedMemory 或者MmAllocateContiguousMemory 函数来请求非分页的或者物理连续的内存。这些函数通常不使用在文件系统或者过滤驱动中，而是用于执行池例程或者其他结构，例如内存池或者旁视列表（lookaside lists）（下面描述）的内存管理中。

使用池

内核驱动如果重复的分配和释放小块的内存（小于一个PAGE_SIZE），可能导致系统的可用物理内存碎片化。这会给系统带来各种各样的问题，包括降底系统的性能等。有一个方法能避免系统碎片化就是预先分配一块合理大小的内存，然后自己管理，在这个预分配的块中分配和释放小块的内存。这样能避免系统碎片化是因为一旦你分配了适当大小的内存块后VMM就和你无关了。你只需要在你用完内存的时候再去找VMM来扩充容量。

为了帮助你在你的驱动中管理内存，NT执行体提供一系列的支持例程。这些例程在一个池上工作，但是你必须预先分配内存。其他的要求是每个可以从池中分配的块的尺寸要在池初始化的时候决定。因此，如果你有一个比一页数据小的固定大小的结构将会重复的分配和释放，就应该认真考虑使用池（或者后面讨论的旁视列表）来执行内存的分配和释放。

注意这个方法要求在驱动中保留一块预先分配的内存。代价就是可能浪费核心内存，因为通常在驱动初始化的时候分配内存块（特别是你的驱动在很长时间里不需要这些内存），利益就是避免了核心内存池碎片化。

这是使用池方法必须使用的操作：

1， 决定你需要的内存的大小

小心处理不要分配太大或者太小的内存池。分配太大的内存很浪费，太小的话就不得不分配更多的内存，从而导致内存碎片。

提示：决定池需要分配的内存的最适合数量通常是一个重复的（iterative）工作。但是作为一个通用的规则，应该为一个池保留的内存的数量，如果分配的太少，那些糟糕的情形将重现，因为你必须在运行的时候再次向VMM申请更多的内存。如果分配的太多（比你要使用的多)将会影响系统中的组件不能访问这些多余的内存而可能引起一些组件的失败。

2， 使用前面列出的ExAllocatePool ( )例程分配池使用的内存。

你要选择从分页的或者非分页的池中分配。注意你的内存片基地址必须在8字节的边界对齐（比如基地址应该是8的倍数）。

3， 分配和初始化一个自旋锁或者使用其他的同步机制来保护对列表的修改。

同步结构将在后面的章节大量地讨论，包括执行自旋锁。

4， 在全局空间中（或者设备扩展中）定义一个ZONE_HEADER结构的类型。

设备扩展在第四章讨论。ZONE_HEADER结构作为这个池的控制结构，池管理支持例程用于从池中分配和释放内存。

5， 调用ExInitializeZone ()来初始化池头部。

你还要传进（这个例程的参数）一个你在第二步中分配的内存的指针和你期望从池中分配的结构的大小，这个大小不许在8字节的边界对齐。

还要注意的是你提供的内存块中的一个ZONE_SEGMENT_HEADER大小的块被池操作例程用来维护一些其他附加信息。其他的预分配的内存将被划分成适当大小的（你指定的）块供驱动使用

现在池准备好被驱动是使用了，当你需要一个新的结构的时候可以使用ExAllocateFromZone ()或者ExlnterlockedAllocateFromZone( )函数。这两个函数的唯一差别是互锁版本的接受一个执行自旋锁结构（前面初始化的），自动使用自旋锁来保持列表的一致性。如果你要使用非互锁版本的函数，你要自己负责保证列表不会被多个同时访问和修改的的线程破坏。因此你必须在驱动中使用适当的同步机制。

返回一个分配的结构到池中使用ExFreeToZone ( ) 或者 ExInterlockedFreeToZone ( )支持例程。

不要在比DISPATCH_LEVEL更高的IRQL等级使用池操作例程，因为在更高的IRQL等级不能使用同步结构（自旋锁或者其他的）

在你需要扩充池的尺寸的时候不要使用函数。再重复，必须传一个新的分配块用于扩充池。记住这个内存基地址也必须在8字节的边界对齐。不幸的是，没有函数用于减少池的大小，因此任何分配和使用，在你初始化或者扩充的池中的块将不能被系统的其他部分使用，直到系统重新启动。这里你的驱动就有责任保证你对池中保留的内存数量有相当准确的估计。

第三部分的文件系统例子代码使用池来管理内存。在磁盘上的示例文件驱动代码说明怎样在你的驱动中使用这个方法。

使用旁视列表（***lookaside lists***）

虽然池帮助减少系统内存的碎片，但是有一些池的不足你必须要知道。

驱动程序必须预先为池分配内存，通常在初始化的时候，可能知道很晚的时候都没有使用这些内存。

你对需要的内存的数量必须相当的精确；你不能释放多余的在你的驱动内存用量高峰时期才会使用的内存。

当你设计和使用驱动的时候，你将看见你的驱动存在请求的高峰时期。自然地，这种时候内存需求会增大。如果使用池，这时候你的池会耗尽，然后有不同的利用率。然后你要么直接从系统分配内存，要么扩大池的尺寸。

扩大池意味着新分配的池在系统重新启动之前不能释放，不是一个很好的前景。直接从系统分配内存意味着你不得不在你分配的结构里维护一系列标志来指出内存来自那里以便能够适当地释放（如果在池中分配的就释放给池，如果直接调用ExAllocatePool ( ) 例程分配的就返还给系统）

必须使用私有的同步机制或者（更普遍的）是自旋锁来同步对池的访问。

旁视列表是一个NT4。0里定义的新结构和相关的支持例程，它突破了池的限制。

当你调用ExInitializeNPagedLookasideList ( ) 或者ExInitializePagedlookasideList ()函数初始化列表的时候不预先分配内存，相反，只有当你真实的请求内存的时候才分配。虽然你的驱动可以在初始化列表的时候提供你驱动特有的分配和释放函数，但这是可选的，NT执行体的内存池管理组将默认使用ExAllocatePoolWithTag()函数和相应的释放例程。

第二，在初始化的时候要求指定列表深度。这个深度指定是列表尺寸的最大值。注意列表会随着不断的分配和释放变得越来越充实。

因此当你开始请求内存，包开始为你分配内存的时候，所有释放的内存将被加入列表中直到达到指定的深度为止。超过这个值的所有分配和释放的内存将自动返还给系统。

这就允许你的驱动在用量高峰的时候增加内存消耗而不用保留这些内存直到下一次系统启动或者是在分配的结构中维护一些状态信息来在释放的时候决定应该释放到那里。

最后，WINDOWS NT提供适当的支持例程，ExAllocateFromNPagedLookasideList() (或者ExAllocateFromPagedLookasideList())函数和相应的释放函数使用 原子的8位 比较-交换 操作来同步访问列表来代替使用和列表相关的FAST_MUTEX 或者 KSPIN_LOCK。这是一个

很有效的同步的方法。
记住一定从非分页内存中分配NPAGED_LOOKASIDE_LIST列表头或者PAGED_LOOKASIDE_LIST表头。

可用的内核堆栈

每个在NT平台的线程有一个用户栈在线程在用户模式执行的时候使用，一个内核栈在线程在内核模式执行的时候使用。
当线程请求系统服务而切换到核心模式的时候，陷阱机制会切换栈，用分配给线程的内核空间的栈来覆盖用户空间的栈。内核栈是固定大小的因此是有限的资源。在NT3.51以前，内核栈限制在两页的内存里。NT4.0开始增加到12KB。但是你的驱动要过度使用的话肯定是不足的。
NT中的高层驱动表现出许多递归的行为，特别是文件系统驱动，VMM和NT缓存管理器。这可能导致内核栈被更快地消耗完。另外I/O子系统中高度层次化的驱动模型也能耗尽内核栈，如果驱动的层次太深，一个或者几个层次中的驱动不注意使用他们的栈的话。
必须警告内核栈不能动态增长。因此必须谨慎地在的栈上使用变量。如果你开发过滤驱动插入驱动层中要非常节约的使用栈空间，因为你疏忽的压栈用量可能超过限制而使系统停止。

使用Unicode字符串

NT操作系统内部使用的所有字符都是Unicode字符（也叫宽字符）。这样能使系统更容易兼容使用非拉丁字符的语言。
当你设计驱动的时候要准备接收Unicode字符串并且能操作这些字符串。每个Unicode字符串都用系统定义的UNICODE_STRING结构表示，这个结构包括下面这些域：

Length

串的字节长度（不是字符的），如果字符串用空结束的话不包含结束的NULL字符。

`MaximumLength`

缓冲区的真实的字节大小，注意可能比Length域的长度更大。

Buffer

指向真正的宽字符串常量。宽字符串不要求需要用空结束，因为上面的Length域描述了字符串包含的字节数。

任何希望存储到相联系的缓冲区的字符串必须长度小于或者等于`MaximumLength`。

注意：要使用一个UNICODE_STRING结构中的宽字符串初始化代表宽字符串常量包含的字节数的Length域不包含UNICODE_NULL字符；初始化代表字符串常量的尺寸的`MaximumLength`域（这就应该包含整个缓冲区包括UNICODE_NULL的空间）。
有多种支持例程用来方便地操作Unicode字符串。`DDK`头文件有这些函数的描述：

`RtllnitUnicodeString`

这个函数初始化一个记数的`Unicode`字符串，可以传一个空结束的字符串源串或者`NULL`。目标`Unicode`字符串的将被初始化为指向空结束的字符串源串的Buffer域或者`NULL`。Length 和 MaximumLength也会被适当地初始化。

如果字符串源串提供了，目标`Unicode`字符串的Length域将设为字符串源串中非空字符的个数乘上sizeof (WCHAR)，MaximumLength也会被初始化为Length+sizeof (UNICODE_NULL)。

`RtlAnsiStringToUnicodeString`

给定一个源ANSI字符串，这个函数会把它转换为`Unicode`而且初始化目标字符串的内容为被转换的字符串的内容。你可以要求这个例程为你为目标宽字符串分配内存或者自己提供内存，把目标Unicode字符串结构的MaximumLength域初始化为传进的

缓冲区的长度。如果要求函数为你分配内存，然后记住调用RtlFreeUnicodeString来释放内存。

`RtlUnicodeStringToAnsiString`

这个函数转换源Unicode字符串为ANSI字符串。

`RtlCompareUnicodeString`

大小写敏感或者不敏感地比较两个Unicode字符串，如果相等返回0，如果第一个字符串小于第二个字符串返回小于0，如果第一个字符串大于第二个字符串返回大于0

`RtlEqualUnicodeString`

大小写敏感或者不敏感地比较两个Unicode字符串，相等返回TRUE，否则返回FALSE。

`RtlPrefixUnicodeString`

这个函数定义如下：

```
BOOLEAN   RtlPrefixUnicodeString(
    IN PUNICODE_STRING Stringl,
    IN PUNICODE_STRING String2,
    IN BOOLEAN CaselnSensitive
)
```

如果`Stringl`是`String2`的前缀就返回`TRUE`，如果两个相等也会返回`TRUE`。

`RtlUpcaseUnicodeString`

把一个源字符串转换为大写的Unicode字符串，然后写入目标的字符串参数里。如果你要求会为目标字符串分配内存；否则必须传一个分配了内存的部标字符串。使用RtlFreeUnicodeString释放这个函数分配的内存。

`RtlDowncaseUnicodeString`

执行和`RtlUpcaseUnicodeString`相反的工作。

`RtlCopyUnicodeString`

复制源字符串到目标字符串里。目标字符串中MaximumLength域指出的数量的会被复制。调用者自己负责为目标预先分配内存。

`RtlAppendUnicodeStringToString`

连接两个Unicode字符串。如果目标字符串的Length域加上源字符串的Length域大于目标字符串中MaximumLength域，函数返回STATUS_BUFFER_TOO_SMALL。

`RtlAppendUnicodeToString`

和`RtlAppendUnicodeStringToString`函数相似，除了源Unicode字符串只是一个宽字符串。

`RtlFreeUnicodeString`

用于释放调用RtlAnsiStringToUnicodeString()或者RtlUpcaseUnicodeString ( )分配的内存。

说明一个宽字符串只要简单地在字符串前面加一个L。例如，ANSI字符串常量“This is a string”可以容易地表示为 L“This is a string”。组成每个宽字符串的字符大小是。宽字符串然后可以用于创建UNICODE_STRING结构，把Buffer域指向宽字符串常量指针，再适当地初始化Length 和MaximumLength域。

不要把Unicode字符象ANSI字符一样看待。例如，你不能假定Unicode字符的大写和小写之间有任何联系。你的一些假设（包括分配适当大小的表来保存字符集）对Unicode字符串来说会不再有效。

链表操作

大多数驱动需要连接内部数据结构或者创建驱动特定的队列，可以使用链表来执行这种功能。NT执行体提供系统定义的数据结构和支持例程来管理链表。

NT DDK中定义了三种的链表支持函数和结构：

单链表

DDK提供预先定义的结构用于创建你自己的单链表。结构定义如下：

```
typedef struct _SINGLE_LIST_ENTRY {
struct _SINGLE_LIST_ENTRY *Next;
} SINGLE_LIST_ENTRY, *PSINGLE_LIST_ENTRY;
```

你应该声明一个这种类型的变量作为链表头。在使用这个链表之前初始化NEXT域为NULL。例如，你可能在设备扩展中有一个域或者和驱动相关的全局内存中的这样的声明：

```
SINGLE_LIST_ENTRY PrivateListHead;
```

每个你希望用这种链表入口类型来链接的结构中也要包含一个SINGLE_LIST_ENTRY类型的域。例如，如果你要队列化SfsdPrivateDataStructure结构，你要这样定义这个数据结构：

```
typedef SFsdPrivateDataStructure {
// Define all sorts of fields...
SINGLE_LIST_ENTRY NextPrivateStructure;
/ / All sorts of other fields...
}
```

现在当你希望把一个实例放入队列中就使用下面的例程：

- PushEntryList()

  这个函数使用两个参数：一个指向链表头另一个指向在你要进入队列的数据结构中SINGLE_LIST_ENTRY类型的域。因此，如果你有一个SfsdPrivateDataStructure类型的变量叫SfsdAPrivateStructure，你可以这样调用：

```
PushEntryList(&PrivateListHead,
              &(SFsdAPrivateStructure.NextPrivateStructure)
            );
```

  你的驱动必须使用某些内部的同步机制来保护这个调用。

- ExInterlockedPushEntryList()

  这个函数和PushEntryList()的唯一区别是你必须在调用的时候提供一个初始化了的KSPIN_LOCK类型的变量。同步是自动由函数使用提供的自旋锁来做。

  注意你必须保证所有的传给函数的链表节点结构从非分页中分配。因为系统在得到自旋锁后不能承受页面错误。

负责从链表中删除节点使用PopEntryList和 ExInterlockedPopEntryList 函数。

双链表

NT操作系统定义下面的结构来支持双链表：

```
typedef struct _LIST_ENTRY {
struct _LIST_ENTRY * volatile Flink;
struct _LIST_ENTRY * volatile Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
```

和单链表一样，必须定义一个LIST_ENTRY类型的变量作为链表头。还应该用InitializeListHead(&SFsdListAnchorOfTypeListEntry)宏初始化头指针中的向前和向后的指针。注意向前和向后的指针初始化为指向头指针；因此在遍历链表的时候决不能得到

一个NULL节点（双链表形成了一个环链）。

如果你要把特定的结构类型连在一起，要保证所定义的结构中包含有一个LIST_ENTRY类型的域。例如你可以定义一个叫的结构如下：

```
typedef SFsdPrivateDataStructure {
    // Define all sorts of fields...
    LIST_ENTRY NextPrivateStructure;
    // All sorts of other fields...
}
```

可以使用下面的宏/函数把一个SFsdPrivateDataStructure类型的实例放到队列中：

- InsertHeadList()

  这个宏接受一个指向链表头指针的参数（必须使用InitializeListHead初始化过）和一个要插入的结构中LIST_ENTRY类型域的指针，把节点插入链表的头部。

  例如，你可以象下面这样调用这个宏来插入一个SFsdPrivateDataStructure类型的叫SFsdAPrivateStructure的实例到链表中：

  ```
  InsertHeadList(&SFsdListAnchorOfTypeListEntry,
                 &(SFsdAPrivateStructure.NextPrivateStructure));
  ```

- InsertTailList()

  和函数InsertHeadList相似，不同的是把节点插入链表尾部。

- RemoveHeadList() 或RemoveTailList()

  从链表的头部或尾部删除一个节点，都返回被删除的节点的指针。

- RemoveEntryList()

  删除包含参数中LIST_ENTRY类型域指针的结构节点。

上面描述的宏还有互锁版本（写成函数的）。这些函数接受一个额外的初始化了的KSPIN_LOCK类型的变量的指针，用于同步对链表的访问操作。如果要使用互锁版本的函数来操作链表，那么链表接点必须从非分页池中分配。

可以使用IsListEmpty ( )宏来检查一个双链表是否为空。如果链表结构中的Flink 和 Blink域都指向头指针就返回TRUE，否则返回FALSE。

S链表

这是一个NT4。0中介绍的新结构，用于支互锁的，有效的单链表。要使用这个结构，你要定义一个下面的类型的头指针：

```
typedef union _SLIST_HEADER {
   ULONGLONG Alignment;
   struct {
      SINGLE_LIST_ENTRY Next;
      USHORT Depth;
      USHORT Sequence;
   };
} SLIST_HEADER, *PSLIST_HEADER;
```

函数ExInitializeSListHead可以用来初始化S链表的头指针。调用这个函数的时候你的驱动必须提供一个S链表结构的头指针，保证链表头指针是从非分页池中分配的。另外还要分配和初始化一个自旋锁在你添加或者删除接点的时候使用。

ExInterlockedPushEntrySList() 和 ExInterlockedPopEntrySList ()函数提供添加或者删除

接点的功能，不用自旋锁，而是使用硬件体系支持的8位的原子的比较-交换指令来同步。所有S-链表中的节点必须是从非分页池中分配的

也可以使用ExQueryDepthSListHead()函数来探测链表中的节点数。这是很方便的，因为你在再需要专门维护一个链表节点数的计数器（如果你使用SINGLE_LIST_ENTRY类型的头节点的话是不许的）。

CONTAINING_RECORD 宏的使用

NT DDK提供下面的宏对所有内核驱动开发者是非常有帮助的：

```
#define CONTAINING_RECORD(address, type, field) \
((type*)((PCHAR)(address) - (PCHAR)(&((type *)0)->field)))
```

这个宏用于取内存中的结构的基地址，只要你知道一个结构中的域的地址。这个宏定义很简单：驱动提供结构中一个域的指针，和域的名字；宏会计算提供的结构中的域的偏移量（字节），然后从域的指针地址上减去这个偏移量来得到结构的基地址。

CONTAINING_RECORD宏使你可以把LIST_ENTRY 和 SINGLE_LIST_ENTRY域放在包含他们的数据结构的任意位置。当你需要知道内存中这个结构的地址的时候可以使用这个宏，如果你知道这个结构中一个域的地址。

作为一个怎样使用CONTAINING_RECORD宏的例子，假设一个文件系统驱动定义了这样一个结构：

```
typedef struct _SFsdFileControlBlock {
    // Some fields that will be expanded upon later in this book.
    。。。
    // To be able to access all open file(s) for a volume, we will
    // link all FCB structures for a logical volume together
    LIST_ENTRY NextFCB;
} SFsdFCB, *PtrSFsdFCB;
LIST_ENTRY SFsdAllLinkedFCBs;
```

在SFsdFCB中有趣的域是NextFCB，是LIST_ENTRY类型的，可能将会用于插入FCB结构到双链表中。全局变量SFsdAllLinkedFCBs是链表头。

注意在结构SFsdFCB中NextFCB并不是第一个域。相反，他可以是结构中的任意一个位置。但是，有了NextFCB域的地址CONTAINING_RECORD宏就能够得到SFsdFCB结构的地址。下面的代码片段遍历和处理所有连接在全局变量SFsdAllLinkedFCBs上的所有SFsdFCB结构。

```
LIST_ENTRY TmpListEntryPtr = NULL;
PtrSFsdFCB PtrFCB = NULL;
TmpListEntryPtr = SFsdAllLinkedFCBs.Flink;
while (TmpListEntryPtr != &SFsdAllLinkedFCBs) {
    PtrFCB = CONTAINING_RECORD(TmpListEntryPtr, SFsdFCB, NextFCB);
    // Process the FCB now.
    // Get a pointer to the next list entry.
    TmpListEntryPtr = TmpListEntryPtr->Flink;
}
```

因此，再次注意并不要求你的驱动把LIST_ENTRY 和 SINGLE_LIST_ENTRY类型的域放在包含他们的结构的头部，你只要使用CONTAINING_RECORD来取得基本结构的指针就行了。

准备调试驱动

在你设计驱动的时候有几点要记住：

插入调试断点

在附录D，调试支持中详细地描述了调试内核驱动。现在要注意的是如果有调试器连接到你的目标机器上，你可以在某些事件发生了的时候插入DbgBreakPoint函数调用来进入调试器。

小心的在你的调试中断点语句周围加上#ifdef语句以便你可以在非调试版本中容易地关掉调试。这是我使用的方法：

```
#if DBG
#define SFsdBreakPoint() DbgBreakPoint()
#else
#define SFsdBreakPoint()
#endif
```

当你使用“检查”编译环境编译你的驱动的时候DBG的值为1。这时候，在你驱动中的任何语句是打开的。期望的是你只会在开发和测试阶段执行调试版本，而且总会有一台调试机器连接到执行驱动的目标机器上。但是，如果你使用非调试编译环境编译驱动的时候，语句SFsdBreakPoint就会是无效的。

NT DDK还提供和这里的SFsdBreakPoint函数定义一样的KdBreakPoint函数。因此，你可以选择在你的代码中简单的使用来保证调试中断会在非调试版本中无效。

插入调试打印语句

可以在驱动的调试版本中使用的DbgPrint宏定义KdPrint。可以提供格式化字符串给这个函数就象调用printf函数做的那样。KdPrint宏在非调试版本中自动无效。

在驱动中插入bugcheck调用

除非不得不这样，否则决不能使系统停止。在执行你的代码的时候使系统停止的确是只有非常少的理由。相反的，在你的代码中检测到不一致的时候应该寻求其他的可能的解决方法。试着停止你的驱动，停止处理请求，停止出错的模块，和任何避免停止系统的事情。但是还是可能有希望停止系统的情况（特别是在开发中）。有两个可以选择的函数可以调用来进行在控制方式下立即停止系统：

- KeBugCheck()

  这个函数接受一个unsigned long参数（BugCheckCode）作为结束系统执行的原因。在内部，KeBugCheck只是调用下面描述的KeBugCheckEx函数。

- KeBugCheckEx()

  这个函数最多接受五个参数，第一个是BugCheckCode，其他四个是可以选择的参数（每个都是unsigned long类型）使你可以向系统的用户提供更多的可能有帮于分析导致bugcheck原因的信息。

  系统没有规定这四个参数应该有的值。

如果没有调试器连接到系统，系统将作下面的工作：

关闭所有中断。

要求其他所有节点（多处理器系统）停止执行。

使用HalDisplayString打印消息。

用户将在显示器上看见臭名臭名糟著的蓝屏，信息

STOP: 0x%lX (0x%lX, 0x%lX, 0x%lX, 0x%lX)将显示出来，第一个是BugCheckCode，接下来是提供给KeBugCheckEx四个可选的参数。

如果有消息能和BugCheckCode关联就调用HalDisplayString打印描述消息。

KeBugCheckEx函数接下来尝试转储机器状态。

如果任何一个BugCheck的参数是有效的代码地址，系统会试图包含这个代码地址的印象文件的名字。

例程会打印执行的操作系统的版本和试图显示已经加载的模块的列表。能显示多少模块的名字依赖于你的显示器能显示多少行。最后，函数试图转储当前栈。系统然后停止执行。但是，如果有调试器连接到系统中，KeBugCheckEx函数将显示这样的信息：

`Fatal System Error: 0x%lX (0x%lX, 0x%lX, 0x%lX, 0x%lX)` 到你的调试机器上（使用DbgPrint函数）。然后使用函数DbgBreakPoint进入调试器。现在你就有机会检查系统状态来看看是什么引起这个错误。如果你要求继续执行，就执行上面描述的代码序列。

NT对象名字空间

在第一章“NT系统组成”中描述的，WINDOWS NT设计这极力使她成为一个基于对象的系统。系统定义了大量的对象类型和与每个类型相关的适当的方法（或函数），使内核组件可以访问和修改这些类型的对象。

NT对象类型包括了适配器对象，控制器对象，进程对象，线程对象，驱动程序对象，设备对象，文件对象，定时器对象等。在这些对象上面是目录对象，是以上对象类型的容器。

对象管理器允许每个对象有一个相关的名字，这就使进程共享对象更容易，因为可能有一个或者多个进程会打开一个特定对象的同名的对象。因此对象管理器为每个运行NT操作系统的节点维护一个全局名字空间。

紧随最新式的商业文件系统的实现，NT对象管理器向系统提供一个分层的名字空间。这个名字空间有一个叫“\”的根目录。所有的有名对象可以由指定的从根开始的绝对路径来定位。注意对象管理器允许创建在对象目录中创建对象目录，从而提供了多层的树。

对象管理器还支持叫做符号连接对象类型的特定类型，符号链接只是命名对象的别名。

图 2-7示意一个典型的NT对象管理器提供的名字空间：

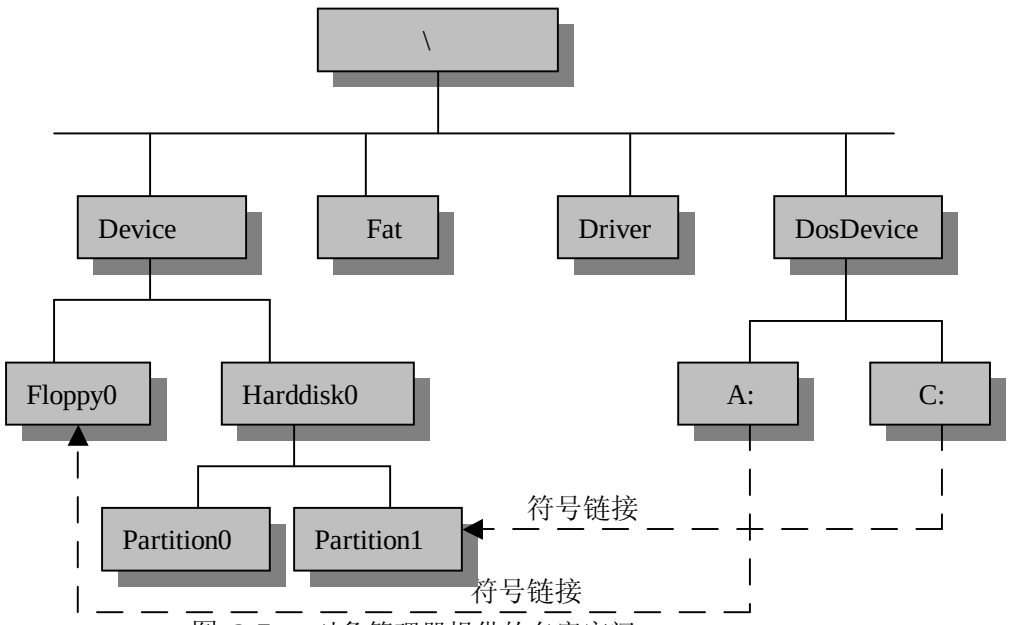

图 2-7 NT对象管理器提供的名字空间

NT对象管理器在其他NT组件请求的时候定义对象类型。某些对象类型是NT对象管理器预先定义的。当一个NT执行体组件请求NT对象管理器定义一个新的类型时，组件可以提供和这种特定类型的所有实例相关的回调函数，包括分析，关闭和删除操作。NT对象管理器记住这些函数指针，当这种特定类型的实例执行分析，关闭或者清除操作的时候调用这些函数。

当用户进程或者应用程序试图打开一个对象的时候必须向NT对象管理器提供绝对路径。NT对象管理器分析每次一个符号的分析这个路径。当NT对象管理器碰到某个对象有相关的分析回调函数的时候，他会挂起自己的分析而调用对象提供的分析函数，分析用户提供的路径的剩余部分（还没有分析的部分）

这些和文件系统和网络重定向器又是怎样的关系呢？

假设用户试图打开文件 C:\accounting\june-97

用户的请求提交给WIN32子系统，WIN32子系统在把请求传递给NT执行体进行更多处理之前先把C:部分转换为\DosDevices\C:，送给NT内核的完整名字是\DosDevices\C:\accounting\june-97。

注意：C：驱动器名字只是WIN32子系统提供的\DosDevices\C:在NT对象名字空间中的符号链接对象类型缩写。因此WIN32子系统在传递请求给NT执行体之前要负责展开这个名字。这也是为什么在NT执行体中不能用C:\。。。路径来创建或者打开文件的原因（例如在你的驱动中）。必须使用NT对象管理器认识的路径名，从对象管理器的名字空间的根目录开始的路径名。

所有的创建和打开请求最初被交给NT对象管理器。对象管理器收到打开请求后开始分析文件名。第一个注意到的事情是对象\DosDevices\C:实际上是一个符号链接对象（\DosDevices部分指向一个目录对象类型），因为符号链接对象类型包含他链接的对象的名字，对象管理器就用他链接的名字（如\Device\HarddiskO\Partition1）去代替符号链接（如\DosDevices\C:）。

注意：在NT4。0中，\DosDevices对象是目录对象\??的一个符号链接。因此在NT4。0中，

对象管理器将首先用\??代替\DosDevices，然后开始分析名字。

现在完整的名字是\Device\HarddiskO\Partitionl\accounting\june-97。对象管理器执行完名字替换后就开始重新从对象名字空间的根目录分析路径名。文件系统管理的一部分也在对象名字空间中，如图2-8所示：

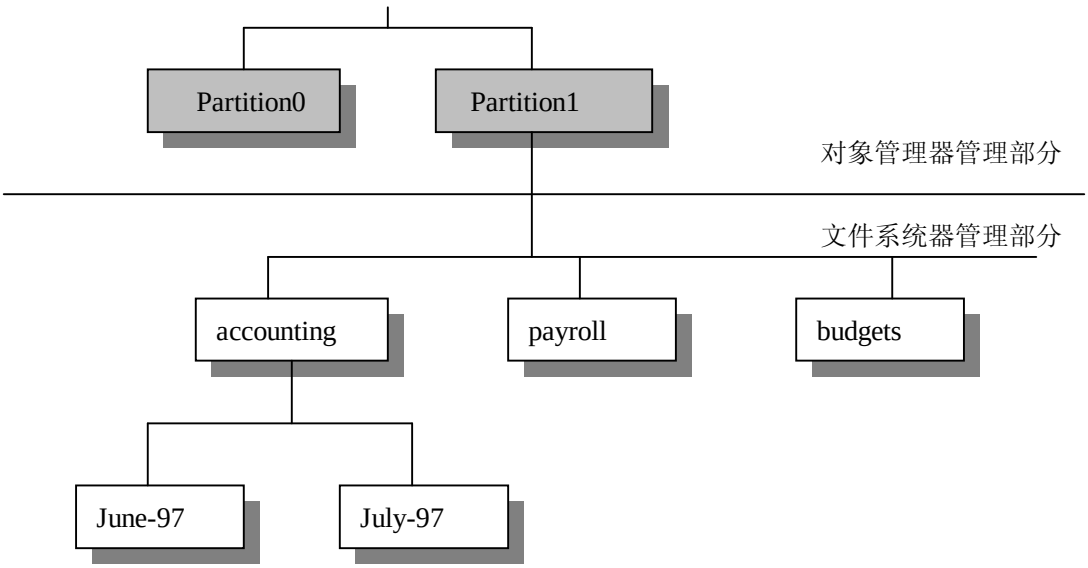

图 2-8 对象名字空间

现在对象管理器遍历整个对象名字空间直到发现Partition1设备对象。这是一个NT I/O管理器定义的一个设备类型。I/O管理器在创建的时候还提供了分析例程。因此，对象管理器不再对路径名进行其他的分析，而是把这个打开请求传递给I/O管理器的分析例程。传递给I/O管理器的是对象管理器还没有分析的部分（\accounting\june-97）。对象管理器在调用分析例程的时候还要传递Partition1设备对象的指针给I/O管理器。

I/O管理器现在执行可以理解的（有道理的）复杂的指令序列来为调用者执行打开操作。这个序列会在随后的章节中更详细的描述。现在，你只要知道I/O管理器通常会定位当前管理有名设备

对象Partition1所代表的的物理磁盘的逻辑卷的文件驱动程序。一旦找到适当的文件系统驱动程序，I/O管理器就简单地把打开操作传递给文件系统驱动的的创建/打开分配例程。
现在文件驱动程序负责处理用户的请求。注意传递给文件驱动程序的文件名是NT对象管理器还没有分析的部分：\accounting\june-97。
这就是用户的打开/创建请求怎样在文件系统驱动中结束的。理解调用文件系统驱动打开/创建分配例程的操作序列在我们开始实现文件系统打开/创建配例程和更详细的实现文件系统挂载逻辑卷的时候是很有帮助的。

网络重定向的文件名处理

在这一章的前面我们看见网络重定向器是一个提供文件系统接口的内核模式软件模块，但是实际上通过网络和远程节点上的服务器来得到远程共享的逻辑卷的数据。
联合提供者路由MPR（Multiple Provider Router）和联合通用命名规范提供者MUP（Multiple Universal Naming ConventionProvider）模块和网络重定向器交互，向客户机上的用户提供本地文件系统的外观。这些组件和内核模式的网络重定向器一起，负责把远程共享逻辑卷文件系统和客户机的本地名字空间结合起来。因此要设计和开发网络重定向器模块的话得对这些组件有相当的了解。

MPR

执行在用户模式的DLL。作为和通用应用程序组件即网络组件和可能执行在客户节点上的联合网络提供这之间的一个缓冲。

> 注意：网络提供者是一个设计来协助网络重定向器工作的软件模块。网络提供者为系统提供一些接口，允许网络组件应用程序向网络重定向器以标准的风格请求一些通用的功能，而不用为客户机上的每种可能的网络重定向器开发不同的代码。

你可能会疑惑在一个客户节点上怎么会有多个网络提供者。如果你仔细考虑一下的话，在一个客户节点上安装和运行多个网络重定向器不是一个不平常的情况。NT操作系统自带的LAN Manager Redirector，还有各种商业的实现的网络文件系统协议，分布式文件系统协议也是实现为网络重定向器。然后，想想所有的第三方开发人员（象你）开发的网络文件系统。你就会知道一个节点上和可能就会安装有一个或多个网络重定向器。
那么MPR实际做的事情是什么呢？考虑一下在NT客户机上的可用的网络命令。这些命令使用户可以创建一个到远程共享的网络驱动器的新的连接。另外用户还可以取得连接的远程节点的信息，浏览上面的共享资源，在不需要的时候删除连接和执行其他相似的任务。作为网络用户或应用开发人员希望的是能和安装在机器上的多网络重定向器以一种标准的方式交互，而不用处理任何特定的网络相关的问题。
这就是MPR要努力完成的功能。MPR定义了两个例程集，每一个属于一个互相区别的，定义良好的接口。一个是MPR DLL提供的网络无关的接口用于WIN32应用开发人员来向网络提供者/网络重定向器请求服务。相似的，另一个接口由MPR调用，必须由各种网络重定向器来实现。
因此，WIN32应用程序要创建一个新的网络连接将调用一个叫做WNetAddConnection或 WNetAddConnection2 的标准的WIN32 API。这些函数在MPR DLL中实现。收到这个请求后，MPR DLL会调用NPAddConnection或者等效的由注册到MPR的网络提供者提供的例程。一旦网络提供者收到这个请求，他可以决定是否处理这个请求，返回操作的结果给MPR然后返回给原来请求的进程，或者他是否允许MPR这样做。注意，为了处理这个请求，网络提供者常常要用文件系统控制请求来调用内核网络重定向器。第十一章，实现文件系统驱动III，解释文件系统控制请求怎样被文件系统驱动（重定向器）处理。

> 注意：要向MPR注册网络提供者DLL，客户端节点的注册表必须要修改。如果你设计和实

现一个网络重定向器，又决定用一个网络提供者DLL来和它沟通，你的安装程序可能会为你执行这些所有的适当的修改。

附录B，MPR支持，描述为了安装网络提供者必须要做的注册表的修改。

各种网络提供者被调用的顺序和他们在注册表中的顺序一样。

在MPR向网络提供者DLL发起NPAddConnection请求的时候，DLL很可能把请求提交给内核重定向器。重定向器为了处理请求可能试图连接参数指定的远程节点，试图定位远程节点上的共享资源，还试图建立连接。

如果请求进程指定的请求处理成功，网络提供者DLL可能会试图创建一个驱动盘符（比如，X:）的符号连接来代表新创建的到远程共享资源的连接的对象。这个符号链接对象也可以指向内核重定向器创建的新设备对象，代表新的连接，或者指向普通重定向器设备对象自己。在这两种情况下，当用户进程试图访问属于X：驱动盘符名字空间，请求将被I/O管理器重定向到内核中的网络重定向器来进行更多的工作。

注：网络提供者DLL通常使用WIN32函数来创建驱动盘符（符号链接类型）。还要知道大多数文件系统和网络重定向器创建命名设备对象来代表文件系统设备或者重定向器设备。通常指向远程网络驱动器的驱动盘符（符号链接）指向网络重定向器设备对象。

附录B中描述了支持通用NT网络组件应用程序在你的网络提供者中必须实现的函数。如果你实现的网络提供者实现了这些描述的函数，你的网络重定向器就能够从系统提供的组件中得到好处，比如用网络命令“添加/删除/查询（add/delete/query）”连接到远程（共享）资源。

Multiple UNC Provider（联合通用命名规范提供者）

WINDOWS NT平台允许用户用通用命名规范（UNC）访问远程共享资源。这个规范的设计很简单：每一个共享的远程资源可以用这样的名字来唯一的标志：\\服务器名\共享资源名。

在服务器名和共享资源名中可以使用的字符有很少的限制。不能在服务器名字或者共享资源名字中使用“\“，单数大多数其他的普通字符是允许的。另一个限制是UNC名字的最大长度（包括远程服务器名和共享资源的名字）不能超过255个字符。

那么当用户用UNC名字来访问远程共享资源的时候名字是怎么解析的呢？

因为UNC是WIN32特有的，WIN32子系统总是从用户进程提供的名字中寻找UNC名字。当找到的时候，WIN32子系统用\Device\UNC来代替“\\”然后把请求提交给NT执行体。

\Device\UNC对象类型实际上是\Device\Mup对象的一个符号链接。MUP驱动是一个非常简单的内核驱动（不象上面讨论的MPR模块在用户空间中）,被描述成资源定位器，在系统启动的时候自动加载。在驱动初始化的时候创建一个FILE_DEVICE_MULTI_UNC_PROVIDER类型的设备对象。它还实现一个创建/打开分派例程在有创建/打开请求的时候调用。

在MUP驱动收到“打开”请求之后，MUP向每个向它注册的网络重定向器发送一个特定的输入/输出控制（IOCTL），询问网络重定向器是否认识调用者提供的名字（即是，\\服务器名\共享资源名\...）。

任何重定向器（可能不只一个）都可以争取得到解释这个名字。重定向器辨认出这个名字必须向MUP报告他从名字字符串中辨认的字符数作为一个唯一的有效的远程资源标志。第一个向MUP注册的重定向器有较高的优先权其他的依次类推，如果有一个以上的重定向器理解这个远程共享资源名字，那么这个顺序决定哪个重定向器能处理用户的请求。

当任何一个重定向器识别出这个名字，MUP把这个网络重定向器的设备对象插进路径名字符串中，覆盖文件对象中的名字，然后返回STATUS_REPARSE给对象管理器。这时候请求直接发给这个网络重定向器进行更多的处理。现在MUP就完全不参与处理了。也不再被属于这个创建/打开请求的任何操作调用。

MUP的另一个可以执行的任务是缓存被网络重定向器识别过的部分名字。下一次收到相同的名

字的打开请求的时候，MUP检查缓存中是否有这个名字，如果有的话直接把请求发到目标网络重定向器设备对象而不执行第一次的询问。缓存中的名字在一定时间后自动丢弃（通常是这个名字最后使用后15分钟）。

为了能和MUP协同工作，你的网络重定向器必须做两件事：

使用系统例程FsRtlRegisterUncProvider向MUP注册，这通常在你的驱动初始化的时候做。

回应MUP发出的询问是否认识一个名字的特定的设备控制请求。

示例的代码片段在本书的其他地方提供。

下一章讨论怎样在你的驱动中整合结构化异常和WINDOWS NT上可以使用的各种同步元素。