# Windows 文件系统过滤驱动开发教程 （第二版）

楚狂人-2007-上海 (MSN:walled_river@hotmail.com)

## -1. 改版序

大约两年以前我在驱动开发网上发表了一组描述如何开发 **Windows** 文件系统过滤驱动的文章。非常庆幸这些文章能给大家带来帮助。

原本的文章中我使用了自己编写的代码。我不打算在这里论述代码风格的优劣并发起一场辩论，无可怀疑的是，读者们大多喜欢看到类似微软范例的代码。为此我把文章中的代码换成微软标准的文件过滤驱动范例 **sfilter** 的代码。赠于喜欢此书的读者和驱动开发的后来者们。

网友们帮我整理的原版已经非常流行。为了区别起见，称为第二版。

## 0. 作者，楚狂人自述

我感觉 **Windows** 文件系统驱动的开发能找到的资料比较少。为了让技术经验不至于遗忘和引起大家交流的兴趣我以我的工作经验撰写本教程。

我的理解未必正确，有错误的地方望多多指教。我在一家软件公司从事安全软件相关的开发工作。我非常庆幸和许多优秀的同事一起工作,特别要提到 **wowocock,陆麟,jiurl** 和曾经的同事 **Cardmagic.**非常乐意与您交流。有问题欢迎与我联系。邮箱为 **MFC_Tan_Wen@163.com**。



## 1. 概述，钻研目的和准备

我经常在碰到同行需要开发文件系统驱动。**windows** 的 **pc** 机上以过滤驱动居多。其目的不外乎有以下几种：

一是用于防病毒引擎。希望在系统读写文件的时候，捕获读写的数据内容，然后检测其中是否含有病毒代码。

二是用于文件系统的透明附加功能。比如希望在文件写过程中对数据进行加密，数据个性化等等过程，针对特殊的过程进行特殊处理，增加文件系统效率等。

三一些安全软件使用文件过滤进行数据读写的控制，作为防信息泄漏软件的基础。

四也有一些数据安全厂家用来进行数据备份与灾难恢复。

如果你刚好有以上此类的要求，你可以阅读本教程。

文件系统驱动是 **windows** 系统中最复杂的驱动种类之一。不能对 **ifsddk** 中的帮助抱太多希望，以我的学习经验看来，文件系统相关的 **ddk** 帮助极其简略，很多重要的部分仅仅轻描淡写的带过。如果安装了 **ifsddk**，应该阅读 **src\filesys\OSR_docs** 下的文档。而不仅仅是 **ddk** 帮助。

文件系统驱动开发方面的书籍很少。中文资料我仅仅见过侯捷翻译过的一本驱动开发的书上有两三章涉及，也仅仅是只

能用于 **9x** 的 **vxd** 驱动。但我们付出巨大努力所理解的 **vxd** 架构如今已经彻底作古。**NT** 文件系统我见过一本英文书。我都不记得这两本书的书名了。

如果您打算开发 **9x** 或者 **nt** 文件系统驱动,建议你去网上下载上文提及的书。那两本书都有免费的电子版本下载。如果你打算开发 **Windows2000\WindowsXP\Window2003** 的文件系统驱动，你可以阅读本教程。虽然本教程仅仅讲述文件系统过滤驱动。但是如果您要开发一个全新的文件系统驱动的话，本教程依然对你有很大的帮助。至少便于你理解文件系统驱动的一些概念。

学习文件系统驱动开发之前，应该在机器上安装 **ifsddk**。**ddk** 版本越高级，其中头文件中提供的系统调用也越多。一般的说用高版本的 **ifsddk** 都可以编译在低版本操作系统上运行的驱动（使用对应的编译环境即可）。**ifsddk** 可以在某些 **ftp** 上免费下载。请不要发邮件向我索取。

我的使用的是 **ifs ddk for 2003,**具体版本号为 **3790,**但是我实际用来开发的两台机器有一台是 **windows 2000**，另一台是 **windows 2003.**我尽量使我编译出来的驱动，可以在 **2000\xp\2003** 三种系统上都通过测试。

同时最新的测试表明，在 **Vista** 系统上，**sfilter** 也可以正常的运行。

安装配置 **ddk** 和在 **vc** 中开发驱动的方法网上有很多的介绍。**ifsddk** 安装之后，**src** 目录下的 **filesys** 目录下有文件系统驱动的示例。阅读这些代码你就可以快速的学会文件系统驱动开发。 **filter** 目录下的 **sfilter** 是一个文件系统过滤驱动的例子。另一个 **filespy** 完全是用这个例子的代码加工得更复杂而已。 本文为觉得代码难懂的读者提供一个可能的捷径。

如何用 **ddk** 编译这个例子请自己查看相关的资料。

文件系统过滤驱动编译出来后你得到的是一个扩展名为 **sys** 的文件。同时你需要写一个**.inf** 文件来实现这个驱动的安装。我这里不讨论**.inf** 文件的细节，你可以直接用 **sfilter** 目录下的 **inf** 文件修改。以后我们将提供一个通用的 **inf** 文件**.**

对 **inf** 文件点鼠标右键弹出菜单选择**"**安装**"**，即可安装这个过滤驱动。但是必须重新启动系统才生效。这是静态加载的情况。静态加载后如果重启后蓝屏无法启动，可以用其他方式引导系统后到 **system32\drivers** 目录下删除你的**.sys** 文件再重启即可。安全模式无法使你避免蓝屏。所以我后来不得不在机器上装了两个系统。双系统情况下，一个系统崩溃了用另一个系统启动，删除原来的驱动即可。

同时 **xp** 和 **2003** 版本的驱动大多可以动态加载。调试更加快捷，也请阅读相关的资料。

如果要调试代码，请安装 **softice** 或者 **windbg**，并阅读 **windows** 内核调试的相关文档。

**2 . hello world,**驱动对象与设备对象

这里所说的驱动对象是一种数据结构，在 **DDK** 中名为 **DRIVER_OBJECT**。任何驱动程序都对应一个 **DRIVER_OBJECT.**如何获得本人所写的驱动对应的 **DRIVER_OBJECT** 呢？驱动程序的入口函数为 **DriverEntry,**因此，当你写一个驱动的开始，你会写下如下的代码：

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING
RegistryPath )
{
}
```

这个函数就相当与喜欢 **c** 语言的你所常用的 **main().IN** 是无意义的宏，仅仅表明后边的参数是一种输入，而对应的 **OUT** 则代表这个参数是一种返回。这里没有使用引用,因此如果想在参数中返回结果，一律传入指针。

**DriverObject** 就是你所写的驱动对应的 **DRIVER_OBJECT,**是系统在加载你的驱动时候所分配的。**RegisteryPath** 是专用于你记录你的驱动相关参数的注册表路径。这两者都由系统分配并通过这两个参数传递给你。

**DriverObject** 重要之处，在于它拥有一组函数指针，称为 **dispatch functions.**

开发驱动的主要任务就是亲手撰写这些 **dispatch functions.**当系统用到你的驱动，会向你的驱动发送 **IRP**（这是 **windows** 所有驱动的共同工作方式）。你的任务是在 **dispatch function** 中处理这些请求。你可以让 **irp** 失败，也可以成功返回，也可以修改这些 **irp**，甚至可以自己发出 **irp**。

设备对象则是指 **DEVICE_OBJECT.**下边简称 **DO.**

但是实际上每个 **irp** 都是针对 **DO** 发出的。只有针对由该驱动所生成的 **DO** 的 **IRP,** 才会发给该驱动来处理。具体的分发函数，决定于 **DO** 下的 **DriverObject** 域。

当一个应用程序打开文件并读写文件的时候，**windows** 系统将这些请求变成 **irp** 发送给文件系统驱动。

文件系统过滤驱动将可以过滤这些 **irp.**这样，你就拥有了捕获和改变文件系统操作的能力。

象 **Fat32,NTFS** 这样的文件系统**(File System,**简称 **FS)**，可能生成好几种设备。首先文件系统驱动本身往往生成一个控制设备（**CDO**）.这个设备的主要任务是修改整个驱动的内部配置。因此一个 **Driver** 只对应一个 **CDO.**

另一种设备是被这个文件系统 **Mount** 的 **Volume**。一个 **FS** 可能有多个 **Volume,**也可能一个都没有。解释一下，如果你有 **C:,D:,E:,F:**四个分区。**C:,D:**为 **NTFS,E:,F:**为 **Fat32.**那么 **E:,F:**则是 **Fat** 的两个 **Volume** 设备对象.

实际上**"C:"**是该设备的符号连接（**Symbolic Link**）名。而不是真正的设备名。可以打开 **Symbolic Links Viewer,**能看到：

**C: \Device\HarddiskVolume1**

因此该设备的设备名为**"\Device\HarddiskVolume1".**

这里也看出来，文件系统驱动是针对每个 **Volume** 来生成一个 **DeviceObject,**而不是针对每个文件的。实际上对文件的读写的 **irp,**都发到 **Volume** 设备对象上去了。并不会生成一个"文件设备对象"。

掌握了这些概念的话，我们现在用简单的代码来生成我们的 **CDO,**作为我们开发文件系统驱动的第一步牛刀小试。

```
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
```

```
{
    // 定义一个 Unicode 字符串。
    UNICODE_STRING nameString;
    RtlInitUnicodeString( &nameString, L"\\FileSystem\\Filters\\SFilter" );

    // 生成控制设备
    status = IoCreateDevice( DriverObject,
                             0,                         //has no device extension
                             &nameString,
                             FILE_DEVICE_DISK_FILE_SYSTEM,
                             FILE_DEVICE_SECURE_OPEN,
                             FALSE,
                             &gSFilterControlDeviceObject );

    // 如果因为路径没找到而生成失败
    if (status == STATUS_OBJECT_PATH_NOT_FOUND) {

        // 这是因为一些低版本的操作系统没有\FileSystem\Filters\这个目录
        // 如果没有，我们则改变位置，生成到\FileSystem\下.
        RtlInitUnicodeString( &nameString, L"\\FileSystem\\SFilterCDO" );
        status = IoCreateDevice( DriverObject,
                                 0,&nameString,
                                 FILE_DEVICE_DISK_FILE_SYSTEM,
                                 FILE_DEVICE_SECURE_OPEN,
                                 FALSE,
                                 &gSFilterControlDeviceObject );

        // 成功后，用 KdPrint 打印一个 log.
        if (!NT_SUCCESS( status )) {

            KdPrint(( "SFilter!DriverEntry: Error creating control device object \"%wZ\",
status=%08x\n", &nameString, status ));
            return status;
        }

    } else if (!NT_SUCCESS( status )) {
        // 失败也打印一个。并直接返回错误
        KdPrint(( "SFilter!DriverEntry: Error creating control device object \"%wZ\",
status=%08x\n", &nameString, status ));
    }
    return status;
}
```

sfilter.sys.把这个文件与前所描述的 inf 文件同一目录，按上节所叙述方法安装。

这个驱动不起任何作用，但是你已经成功的完成了"hello world".

初次看这些代码可能有一些慌乱。但是只要注意了以下几点，你就会变得轻松了:

1） 习惯使用 **UNICODE_STRING** 字符串。这些字符串用 **Rtl...**系列的函数来操作。你应该阅读 **DDK** 帮助，然后熟悉这些字符串的用法。

2） 用 **KdPrint(())**来代替 **printf** 输出信息。这些信息可以在 **DbgView** 中看到。**KdPrint(())**自身是一个宏，为了完整传入参数所以使用了两重括弧。这个比 **DbgPrint** 调用要稍好。因为在 **free** 版不被编译。

3） 查看 **DDK** 帮助了解生成设备对象 **IoCreateDevice** 的用法。

请注意 **CDO** 生成后，保存在 **gSFilterControlDeviceObject** 中。这样以后我们得到一个 **DEVICE_OBJECT** 时，就很容易判断是否是我们的控制设备。

**3 .**分发例程**,fast io**

上一节仅仅生成了控制设备对象。但是不要忘记，驱动开发的主要工作是撰写分发例程**(dispatch functions.).**接上一接，我们已经知道自己的 **DriverObject** 保存在上文代码的 **driver** 中。现在我来指定一个默认的 **dispatch function** 给它。

```
for (i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++)
{
    DriverObject->MajorFunction[i] = SfPassThrough;
}
```

作为过滤，一些特殊的分发例程，我必须特殊的给予处理。为此，给它们单独的分发函数**:**

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = SfCreate;
DriverObject->MajorFunction[IRP_MJ_CREATE_NAMED_PIPE] = SfCreate;
DriverObject->MajorFunction[IRP_MJ_CREATE_MAILSLOT] = SfCreate;

DriverObject->MajorFunction[IRP_MJ_FILE_SYSTEM_CONTROL] = SfFsControl;
DriverObject->MajorFunction[IRP_MJ_CLEANUP] = SfCleanupClose;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = SfCleanupClose;
```

至于过滤中最简单的处理，当然就是不做任何处理，直接下发了，这就是我们常说的 **passthru.**这是一个常用的缩写词。

```
NTSTATUS
SfPassThrough (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )

{
    // 对于我们认为“不能”的事情，我们采用 ASSERT 进行调试模式下的确认。
    // 而不加多余的判断来消耗我们的效率。这些宏在调试模式下不被编译。
    ASSERT(!IS_MY_CONTROL_DEVICE_OBJECT( DeviceObject ));
    ASSERT(IS_MY_DEVICE_OBJECT( DeviceObject ));
    IoSkipCurrentIrpStackLocation( Irp );
```

```
    return IoCallDriver( ((PSFILTER_DEVICE_EXTENSION) DeviceObject->DeviceExtension)->AttachedToDeviceObject, Irp );
}
```

可以看到，发送 **IRP** 的方法为调用 **IoCallDriver,**其中第一个参数为目标设备。这存在一个问题，就是我们如何得到被过滤的目标设备对象。这个问题在后面解决。

但是对于这个 **DriverObject** 的设置，还并不是仅仅这么简单。

由于你的驱动将要绑定到文件系统驱动的上边，文件系统除了处理正常的 **IRP** 之外，还要处理所谓的 **FastIo.FastIo** 是 **Cache Manager** 调用所引发的一种没有 **irp** 的请求。换句话说，除了正常的 **Dispatch Functions** 之外，你还得为 **DriverObject** 撰写另一组 **Fast Io Functions.**这组函数的指针在 **driver->FastIoDispatch.**我不知道这个指针留空会不会导致系统崩溃。在这里本来是没有空间的，所以为了保存这一组指针，你必须自己分配空间。

驱动开发中分配内存有很多学问。最常见的情况，使用 **ExAllocatePool** 分配即可。

```
PFAST_IO_DISPATCH fastIoDispatch;
fastIoDispatch = ExAllocatePoolWithTag( NonPagedPool, sizeof( FAST_IO_DISPATCH ), SFLT_POOL_TAG );
if (!fastIoDispatch) {
    // 分配失败的情况，删除我们先生成的控制设备
    IoDeleteDevice( gSFilterControlDeviceObject );
    return STATUS_INSUFFICIENT_RESOURCES;
}
// 内存清零。
RtlZeroMemory( fastIoDispatch, sizeof( FAST_IO_DISPATCH ));
fastIoDispatch->SizeOfFastIoDispatch = sizeof( FAST_IO_DISPATCH );

//我们过滤以下所有的函数:
fastIoDispatch->FastIoCheckIfPossible = SfFastIoCheckIfPossible;
fastIoDispatch->FastIoRead = SfFastIoRead;
fastIoDispatch->FastIoWrite = SfFastIoWrite;
fastIoDispatch->FastIoQueryBasicInfo = SfFastIoQueryBasicInfo;
fastIoDispatch->FastIoQueryStandardInfo = SfFastIoQueryStandardInfo;
fastIoDispatch->FastIoLock = SfFastIoLock;
fastIoDispatch->FastIoUnlockSingle = SfFastIoUnlockSingle;
fastIoDispatch->FastIoUnlockAll = SfFastIoUnlockAll;
fastIoDispatch->FastIoUnlockAllByKey = SfFastIoUnlockAllByKey;
fastIoDispatch->FastIoDeviceControl = SfFastIoDeviceControl;
fastIoDispatch->FastIoDetachDevice = SfFastIoDetachDevice;
fastIoDispatch->FastIoQueryNetworkOpenInfo = SfFastIoQueryNetworkOpenInfo;
fastIoDispatch->MdlRead = SfFastIoMdlRead;
fastIoDispatch->MdlReadComplete = SfFastIoMdlReadComplete;
fastIoDispatch->PrepareMdlWrite = SfFastIoPrepareMdlWrite;
fastIoDispatch->MdlWriteComplete = SfFastIoMdlWriteComplete;
fastIoDispatch->FastIoReadCompressed = SfFastIoReadCompressed;
fastIoDispatch->FastIoWriteCompressed = SfFastIoWriteCompressed;
fastIoDispatch->MdlReadCompleteCompressed = SfFastIoMdlReadCompleteCompressed;
fastIoDispatch->MdlWriteCompleteCompressed = SfFastIoMdlWriteCompleteCompressed;
```

```
    fastIoDispatch->FastIoQueryOpen = SfFastIoQueryOpen;
    // 最后指定给 DriverObject.
    DriverObject->FastIoDispatch = fastIoDispatch;
```

一开始就介绍 **FastIo** 是一件令人烦恼的事情。首先需要了解的是:**FastIo** 是独立于普通的处理 **IRP** 的分发函数之外的另一组接口。但是他们的作用是一样的，就是由驱动处理外部给予的请求。而且所处理的请求也基本相同。

其次，文件系统的普通分发例程和 **fastio** 例程都随时有可能被调用。做好的过滤驱动显然应该同时过滤这两套接口。然而，一般都只介绍 **IRP** 过滤的方法。**Fastio** 接口非常复杂。但是与 **IRP** 过滤是基本一一对应的。只要了解了前者，后者很容易学会。本节后附上陆麟曾经发过的一小段介绍 **fastio** 接口的文字。有兴趣的读者可以阅读一下。

在开发的初期学习阶段,你可以简单的设置所有的 **fastio** 例程返回 **FALSE** 并不做任何事。这样这些请求都会通过 **IRP** 重新发送被你的普通分发函数捕获。有一定的效率损失，但是并不是很大。

你可能需要一个 **fastio** 过滤函数的 **passthru** 的例子，下面以上面的第一个函数为例:

```
    BOOLEAN
    SfFastIoCheckIfPossible (
        IN PFILE_OBJECT FileObject,
        IN PLARGE_INTEGER FileOffset,
        IN ULONG Length,
        IN BOOLEAN Wait,
        IN ULONG LockKey,
        IN BOOLEAN CheckForReadOperation,
        OUT PIO_STATUS_BLOCK IoStatus,
        IN PDEVICE_OBJECT DeviceObject
        )
    {
        PDEVICE_OBJECT nextDeviceObject;
        PFAST_IO_DISPATCH fastIoDispatch;
        PAGED_CODE();
        if (DeviceObject->DeviceExtension) {
            ASSERT(IS_MY_DEVICE_OBJECT( DeviceObject ));
             // 得到我们绑定的设备，方法和前面的代码一样
            nextDeviceObject                =                ((PSFILTER_DEVICE_EXTENSION)
DeviceObject->DeviceExtension)->AttachedToDeviceObject;
            ASSERT(nextDeviceObject);
             // 得到目标设备的 fastio 分发函数接口
            fastIoDispatch = nextDeviceObject->DriverObject->FastIoDispatch;
             // 判断有效性
            if (VALID_FAST_IO_DISPATCH_HANDLER( fastIoDispatch, FastIoCheckIfPossible )) {
                 // 直接调用
                return (fastIoDispatch->FastIoCheckIfPossible)(
                            FileObject,
                            FileOffset,
                            Length,
                            Wait,
                            LockKey,
                            CheckForReadOperation,
```

```
                    IoStatus,
                    nextDeviceObject );
            }
        }
        return FALSE;
    }
```

如前面所说，最简单的方法，你也可以直接返回 **FALSE.**

### 3.5 附：陆麟关于 fastio 的简述

**NT** 下 **FASTIO** 是一套 **IO MANAGER** 与 **DEVICE DRIVER** 沟通的另外一套 **API.** 在进行基于 **IRP** 为基础的接口调用前, **IO MANAGER** 会尝试使用 **FAST IO** 接口来加速各种 **IO** 操作. **FASTIO** 本身的文档并不多见, 本篇就是要介绍一下 **FASTIO** 接口.

**FastIoCheckIfPossible,** 此调用并不是 **IO MANAGER** 直接调用. 而是被 **FsRtlXXX** 系列函数调用. 用于确认读写操作是否可以用 **FASTIO** 接口进行.

**FastIoRead/FastIoWrite,** 很明显, 是读写处理的调用.

**FastIoQueryBasicInfo/FastIoQueryStandardInfo,** 用于获取各种文件信息. 例如创建,修改日期等.

**FastIoLock/FastIoUnlockSingle/FastIoUnlockAll/FastIoUnlockAllByKey,**用于对文件的锁定操作. 在 **NT** 中.有 **2** 中锁定需要存在.**1.**排他性锁. **2.**共享锁. 排他性锁在写操作前获取,不准其他进程获得写操作权限, 而共享锁则代表需要读文件某区间. 禁止有写动作出现. 在同一地址上, 如果有多个共享锁请求, 那是被允许的.

**FastIoDeviceControl** 用于提供 **NtDeviceIoControlFile** 的支持.

**AcquireFileForNtCreateSection/ReleaseFileForNtCreateSection** 是 **NTFS** 在映射文件内容到内存页面前进行的操作.

**FastIoDetachDevice,** 当 **REMOVABLE** 介质被拿走后, **FILE SYSTEM** 的 **DEVICE** 对象会在任意的时刻被销毁. 只有正确处理这个调用才能把上层 **DEVICE** 和将要销毁的 **DEVICE** 脱钩. 如果不解决这个函数, 系统会当.

**FastIoQueryNetworkOpenInfo,** 当 **CIFS** 也就是网上邻居,更准确的说是网络重定向驱动尝试获取文件信息, 会使用这个调用. 该调用是因为各种历史原因而产生. 当时设计 **CIFS** 时为避免多次在网上传输文件信息请求, 在 **NT4** 时传输协议增加了一个 **FileNetworkOpenInformation** 的网络文件请求. 而 **FSD** 则增加了这个接口. 用于在一次操作中获得所有的文件信息. 客户段发送 **FileNetworkOpenInformation,** 服务器端的 **FSD** 用本接口完成信息填写.

**FastIoAcquireForModWrite, Modified Page Writer** 会调用这个接口来获取文件锁. 如果实现这个接口. 则能使得文件锁定范围减小到调用指定的范围. 不实现此接口, 整个文件被锁.

**FastIoPrepareMdlWrite, FSD** 提供 **MDL.** 以后向此 **MDL** 写入数据就代表向文件写入数据. 调用参数中有 **FILE_BOJECT** 描述要写的目标文件.

**FastIoMdlWriteComplete,** 写操作完成. **FSD** 回收 **MDL.**

**FastIoReadCompressed,** 当此调用被调用时, 读到的数据是压缩后的.应该兼容于标准的 **NT** 提供的压缩库. 因为调用者负责解压缩.

**FastIoWriteCompressed,**当此调用被调用时, 可以将数据是压缩后存储.

**FastIoMdlReadCompressed/FastIoMdlReadCompleteCompressed, MDL** 版本的压缩读. 当后一个接口被调用时,**MDL** 必须被释放.

**FastIoMdlWriteCompressed/FastIoMdlWriteCompleteCompressed, MDL** 版本的压缩写.当后一个接口被调用时,**MDL** 必须被释放.

**FastIoQueryOpen,** 这不是打开文件的操作. 但是却提供了一个 **IRP_MJ_CREATE** 的 **IRP.** 我在以前版本的 **SECUSTAR** 的软件中错误地实现了功能. 这个操作是打开文件/获取文件基本信息/关闭文件的一个操作.

**FastIoReleaseForModWrite,**释放 **FastIoAcquireForModWrite** 调用所占有的 **LOCK.**

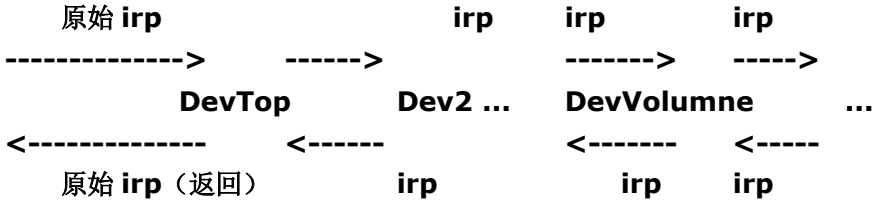
**FastIoAcquireForCcFlush/FastIoReleaseForCcFlush FsRtl** 会调用此接口,在 **LAZY WRITE** 线程将要把修改后的文件数据写入前调用.获取文件锁.

**4.**设备栈,过滤,文件系统的感知

前边都在介绍文件系统驱动的结构，却还没讲到我们的过滤驱动如何能捕获所有发给文件系统驱动的 **irp**，让我们自己来处理**?**前面已经解释过了设备对象。现在来解释一下设备栈。

任何设备对象都存在于某个设备栈中。设备栈自然是一组设备对象。这些设备对象是互相关联的，也就是说，如果得到一个 **DO** 指针，你就可以知道它所处的设备栈。

任何来自应用的请求，最终被 **windowsIO** 管理器翻译成 **irp** 的，总是发送给设备栈的顶端那个设备。




上图向右的箭头表示 **irp** 请求的发送过程，向左则是返回。可见 **irp** 是从设备栈的顶端开始，逐步向下发送。**DevVolumue** 表示我们实际要过滤的 **Volume** 设备，**DevTop** 表示这个设备栈的顶端。我们只要在这个设备栈的顶端再绑定一个设备，那发送给 **Volume** 的请求，自然会先发给我们的设备来处理。

有一个系统调用可以把我们的设备绑定到某个设备的设备栈的顶端。这个调用是 **IoAttachDeviceToDeviceStack**，这个调用 **2000** 以及以上系统都可以用（所以说到这点，是因为还有一个 **IoAttachDeviceToDeviceStackSafe,**是 **2000** 所没有的。这常常导致你的 **filter** 在 **2000** 下不能用。）

以下一个函数来帮我实现绑定功能：

```
NTSTATUS
SfAttachDeviceToDeviceStack (
    IN PDEVICE_OBJECT SourceDevice,
    IN PDEVICE_OBJECT TargetDevice,
    IN OUT PDEVICE_OBJECT *AttachedToDeviceObject
    )
/*++
{
    // 测试代码，测试这个函数是否可以运行在可页交换段
    PAGED_CODE();
    // 不要误解为:当 windows 版本高于等于 0x0501 时，运行以下代码。应该理解为：当我编译
    // 的目标操作系统版本高于 0x0501 时，编译以下代码。反之，不编译。
#if WINVER >= 0x0501
    // 当目标操作系统版本高于 0x0501 时时，我们有一个新的调用 AttachDeviceToDeviceStackSafe
    // 可调。这个调用比 IoAttachDeviceToDeviceStack 更可靠。反之，我们不调用新调用。
    if (IS_WINDOWSXP_OR_LATER()) {
        ASSERT( NULL != gSfDynamicFunctions. AttachDeviceToDeviceStackSafe );
```

```
        // 请注意，如果我们直接调用 IoAttachDeviceToDeviceStackSafe 这个调用，则在
        // 没有 AttachDeviceToDeviceStackSafe 这个调用的机器上，这个驱动无法被加载。
        // 所以这里采用了动态加载这个函数的方式。以保证同一个驱动既可以在高版本操作系
        // 统下运行，也可以在低版本下运行。而目标操作系统为高版本操作系统。
        return (gSfDynamicFunctions.AttachDeviceToDeviceStackSafe)( SourceDevice,
                                                                    TargetDevice,
                                                                    AttachedToDeviceObject );

    } else {
        ASSERT( NULL == gSfDynamicFunctions.AttachDeviceToDeviceStackSafe );
#endif
        // 目标操作系统为低版本的情况，则不需要动态加载调用。直接使用旧调用。
        *AttachedToDeviceObject = TargetDevice;
        *AttachedToDeviceObject = IoAttachDeviceToDeviceStack( SourceDevice,
                                                               TargetDevice );
        if (*AttachedToDeviceObject == NULL) {
            return STATUS_NO_SUCH_DEVICE;
        }
        return STATUS_SUCCESS;
#if WINVER >= 0x0501
    }
#endif
}
```

关于动态加载系统调用，请查阅 **MmGetSystemRoutineAddress** 的帮助。

到这里，我们已经知道过滤对 **Volume** 的请求的办法。比如**"C:"**这个设备，我已经知道符号连接为**"C:",**不难得到设备名。得到设备名后，又不难得到设备。这时候我们 **IoCreateDevice()**生成一个 **Device Object,**然后调用 **wd_dev_attach** 绑定，不是一切 **ok** 吗？所有发给**"C:"**的 **irp,**就必然先发送给我们的驱动，我们也可以捕获所有对文件的操作了**!**

这确实是很简单的处理方法。我得到的 **FileMon** 的代码就是这样处理的，如果不想处理动态的 **Volume,**你完全可以这样做。但是我们这里有更高的要求。当你把一个 **U** 盘插入 **usb** 口，一个**"J:"**之类的 **Volume** 动态诞生的时候，我们依然要捕获这个事件，并生成一个 **Device** 来绑定它。

一个新的存储媒质被系统发现并在文件系统中生成一个 **Volume** 的过程称为 **Mounting.**其过程开始的时候，**FS** 的 **CDO** 将得到一个 **IRP,**其 **Major Function Code** 为 **IRP_MJ_FILE_SYSTEM_CONTROL**，**Minor Function Code** 为 **IRP_MN_MOUNT**。换句话说，如果我们已经生成了一个设备绑定文件系统的 **CDO**，那么我们就可以得到这样的 **IRP,**在其中知道一个新的 **Volume** 正在 **Mount.**这时候我们可以执行上边所说的操作。

那么现在的问题是如何知道系统中有那些文件系统，还有就是我应该在什么时候绑定它们的控制设备。

**IoRegisterFsRegistrationChange()**是一个非常有用的系统调用。这个调用注册一个回调函数。当系统中有任何文件系统被激活或者是被注销的时候，你注册过的回调函数就会被调用。

需要反复强调的是，文件系统的加载，和你插入一个 **U** 盘增加了一个卷完全是两回事。我们都知道有 **NTFS,FAT32,CDFS** 这些文件系统。当你系统中有磁盘使用了 **FAT32** 文件系统的时候，你的 **FASTFAT** 就已经被激活了。那么你再插入多少个 **U** 盘又有什么关系呢。插入光盘也是同样的一件事。物理媒质的增加，卷的增加和文件系统的激活完全不同。

下面看看 **sfilter** 的 **DriverEntry** 中对这个函数的调用**:**

```
status = IoRegisterFsRegistrationChange( DriverObject, SfFsNotification );
if (!NT_SUCCESS( status )) {
```

```
    KdPrint(( "SFilter!DriverEntry: Error registering FS change notification, status=%08x\n",
status ));
    DriverObject->FastIoDispatch = NULL;
    ExFreePool( fastIoDispatch );
    IoDeleteDevice( gSFilterControlDeviceObject );
    return status;
}
```

你有必要为此写一个回调函数。

```
VOID
SfFsNotification (
    IN PDEVICE_OBJECT DeviceObject,
    IN BOOLEAN FsActive)
{
    UNICODE_STRING name;
    WCHAR nameBuffer[MAX_DEVNAME_LENGTH];
    PAGED_CODE();
    RtlInitEmptyUnicodeString( &name, nameBuffer, sizeof(nameBuffer) );
    SfGetObjectName( DeviceObject, &name );
    SF_LOG_PRINT( SFDEBUG_DISPLAY_ATTACHMENT_NAMES,
            ("SFilter!SfFsNotification:                    %s    %p \"%wZ\" (%s)\n",
             (FsActive) ? "Activating file system   " : "Deactivating file system",
             DeviceObject,
             &name,
             GET_DEVICE_TYPE_NAME(DeviceObject->DeviceType)) );
    if (FsActive) {
        SfAttachToFileSystemDevice( DeviceObject, &name );
    } else {
        SfDetachFromFileSystemDevice( DeviceObject );
    }
}
```

这里牵涉到一些关于动态加载的的问题。**IoRegisterFsRegistrationChange** 可以注册对激活文件系统的回调。但是对注册的时候，早就已经激活的文件系统，回调是否有反应呢？早期的 **windows** 版本如 **windows2000** 是没有反应的。而 **2000sp4** 和 **windowsxp** 似乎是有反应的。所有的已存在文件系统会重新枚举一次。

所以 **2000** 下进行动态加载有一定的困难。因为你必须自己枚举所有已经激活的文件系统。同时枚举设备在 **2000** 下又是另一个困难，你必须使用未公开的调用。这个你可以拷贝 **wowocock** 的代码。他总是自己编写 **2000** 下缺少的调用。

我们再次回顾一下，**DriverEntry** 中，应该做哪些工作。

第一步.生成一个控制设备。当然此前你必须给控制设置指定名称。

第二步.设置 **Dispatch Functions.**

第三步.设置 **Fast Io Functions.**

第四步.编写一个 **FileSystemNotify** 回调函数，在其中绑定刚激活的 **FS CDO.**

第五步.使用 **IoRegisterFsRegistrationChange** 调用注册这个回调函数。

应该如何绑定一个 **FS CDO?**这不是一个简单的主题。我们在下面的章节再详细描述。

**5.**绑定 **FS CDO,**文件系统识别器，设备扩展

上一节讲到我们打算绑定一个刚刚被激活的**FS CDO.**前边说过简单的调用**sfAttachDeviceToStack**可以很容易的绑定这个设备。但是，并不是每次 **Fs system notify** 调用发现有新的 **fs** 激活，我就直接绑定它。

首先判断是否我需要关心的文件系统类型。你的过滤驱动可能只对文件系统的 **CDO** 的设备类型中某些感兴趣。

```
#define IS_DESIRED_DEVICE_TYPE(_type) \
    (((_type) == FILE_DEVICE_DISK_FILE_SYSTEM) || \
     ((_type) == FILE_DEVICE_CD_ROM_FILE_SYSTEM) || \
     ((_type) == FILE_DEVICE_NETWORK_FILE_SYSTEM))
```

下一个问题是我打算跳过文件系统识别器。文件系统识别器是文件系统驱动的一个很小的替身。为了避免没有使用到的文件系统驱动占据内核内存，**windows** 系统不加载这些大驱动，而代替以该文件系统驱动对应的文件系统识别器。当新的物理存储媒介进入系统，**io** 管理器会依次的尝试各种文件系统对它进行“识别”。识别成功，立刻加载真正的文件系统驱动，对应的文件系统识别器则被卸载掉。对我们来说，文件系统识别器的控制设备看起来就像一个文件系统控制设备。但我们不打算绑定它。

分辨的方法是通过驱动的名字。凡是微软的标准的文件系统识别器的驱动对象的名字（注意是 **DriverObject** 而不是 **DeviceObject!**）都为“**\FileSystem\Fs_Rec**”.

```
RtlInitUnicodeString( &fsrecName, L"\\FileSystem\\Fs_Rec" );
SfGetObjectName( DeviceObject->DriverObject, &fsName );
if (RtlCompareUnicodeString( &fsName, &fsrecName, TRUE ) == 0)
{
    return STATUS_SUCCESS;
}
```

但是要注意没有谁规定文件系统识别器一定生成在驱动“**\FileSystem\Fs_Rec**”下。所以这个方法只跳过了部分“微软的规矩的”文件系统识别器。对于不能跳过的，我们在 **File System Control** 的过滤中有对应的处理。

接下来我将要生成我的设备。这里要提到设备扩展的概念。设备对象是一个数据结构，为了表示不同的设备，里边将有一片自定义的空间，用来给你记录这个设备的特有信息。我们为我们所生成的设备确定设备扩展如下：

```
// 文件过滤系统驱动的设备扩展
typedef struct _SFILTER_DEVICE_EXTENSION {
    //  我们所绑定的文件系统设备
    PDEVICE_OBJECT AttachedToDeviceObject;
    //  与我们的文件系统设备相关的真实设备（磁盘），这个用于绑定时使用。
    PDEVICE_OBJECT StorageStackDeviceObject;
    // 如果我们绑定了一个卷，这是物理磁盘卷名。否则这是我们绑定的控制设备名。
    UNICODE_STRING DeviceName;
    // 用来保存名字字符串的缓冲区
    WCHAR DeviceNameBuffer[MAX_DEVNAME_LENGTH];
} SFILTER_DEVICE_EXTENSION, *PSFILTER_DEVICE_EXTENSION;
```

之所以如此简单，是因为我们现在还没有多少东西要记录。基本上记得自己绑定在哪个设备上就好了。如果以后需要更多的信息，再增加不迟。扩展空间的大小是在 **wdf_dev_create**（也就是这个设备生成）的时候指定的。得到设备对象指针后，用下面这个函数来获取我们所绑定的原始设备：

```
nextDeviceObject = ((PSFILTER_DEVICE_EXTENSION)DeviceObject->DeviceExtension)
                    -> AttachedToDeviceObject;
```

生成设备后，为了让系统看起来，你的设备和原来的设备没什么区别，你必须设置一些该设备的标志位与你所绑定的设

备相同。

```
if ( FlagOn( DeviceObject->Flags, DO_BUFFERED_IO )) {
SetFlag( newDeviceObject->Flags, DO_BUFFERED_IO );
}
if ( FlagOn( DeviceObject->Flags, DO_DIRECT_IO )) {
SetFlag( newDeviceObject->Flags, DO_DIRECT_IO );
}
if ( FlagOn( DeviceObject->Characteristics, FILE_DEVICE_SECURE_OPEN ) ) {
SetFlag( newDeviceObject->Characteristics, FILE_DEVICE_SECURE_OPEN );
}
```

DO_BUFFERED_IO，DO_DIRECT_IO 这两个标志的意义在于外部向这些设备发送读写请求的时候，所用的缓冲地址将有所不同。这点以后在过滤文件读写的时候再讨论。现在一切事情都做完，你应该去掉你的新设备上的 DO_DEVICE_INITIALIZING 标志，以表明的的设备已经完全可以用了。

```
newDeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
```

下面的函数来完成以上的这个过程。你只要在上一节中提示的位置调用这个函数，就完成对文件系统控制设备的绑定了。

```
NTSTATUS
SfAttachToFileSystemDevice (
    IN PDEVICE_OBJECT DeviceObject,
    IN PUNICODE_STRING DeviceName
    )
{
    PDEVICE_OBJECT newDeviceObject;
    PSFILTER_DEVICE_EXTENSION devExt;
    UNICODE_STRING fsrecName;
    NTSTATUS status;
    UNICODE_STRING fsName;
    WCHAR tempNameBuffer[MAX_DEVNAME_LENGTH];
    PAGED_CODE();
     // 检查设备类型
    if (!IS_DESIRED_DEVICE_TYPE(DeviceObject->DeviceType)) {
        return STATUS_SUCCESS;
    }
    RtlInitEmptyUnicodeString( &fsName,
                               tempNameBuffer,
                               sizeof(tempNameBuffer) );
     // 根据我们是否要绑定识别器
    if (!FlagOn(SfDebug,SFDEBUG_ATTACH_TO_FSRECOGNIZER)) {
         // 否则跳过识别器的绑定
        RtlInitUnicodeString( &fsrecName, L"\\FileSystem\\Fs_Rec" );
        SfGetObjectName( DeviceObject->DriverObject, &fsName );
        if (RtlCompareUnicodeString( &fsName, &fsrecName, TRUE ) == 0) {
            return STATUS_SUCCESS;
        }
    }
```

```
    // 生成新的设备，准备绑定目标设备
    status = IoCreateDevice( gSFilterDriverObject,
                             sizeof( SFILTER_DEVICE_EXTENSION ),
                             NULL,
                             DeviceObject->DeviceType,
                             0,
                             FALSE,
                             &newDeviceObject );
    if (!NT_SUCCESS( status )) {
        return status;
    }
     // 复制各种标志
    if ( FlagOn( DeviceObject->Flags, DO_BUFFERED_IO )) {
        SetFlag( newDeviceObject->Flags, DO_BUFFERED_IO );
    }
    if ( FlagOn( DeviceObject->Flags, DO_DIRECT_IO )) {
        SetFlag( newDeviceObject->Flags, DO_DIRECT_IO );
    }
    if ( FlagOn( DeviceObject->Characteristics, FILE_DEVICE_SECURE_OPEN ) ) {
        SetFlag( newDeviceObject->Characteristics, FILE_DEVICE_SECURE_OPEN );
    }
    devExt = newDeviceObject->DeviceExtension;
     // 使用我们上一节提供的函数进行绑定
    status = SfAttachDeviceToDeviceStack( newDeviceObject,
                                          DeviceObject,
                                          &devExt->AttachedToDeviceObject );
    if (!NT_SUCCESS( status )) {
        goto ErrorCleanupDevice;
    }
     // 记录设备名字
    RtlInitEmptyUnicodeString( &devExt->DeviceName,
                               devExt->DeviceNameBuffer,
                               sizeof(devExt->DeviceNameBuffer) );
    RtlCopyUnicodeString( &devExt->DeviceName, DeviceName );
    ClearFlag( newDeviceObject->Flags, DO_DEVICE_INITIALIZING );
    ErrorCleanupDevice:
        IoDeleteDevice( newDeviceObject );
    return status;
}
```

## 6.IRP 的传递，File System Control Dispatch

我们现在不得不开始写 **dispatch functions.**因为你的设备已经绑定到文件系统控制设备上去了。**windows** 发给文件系统的请求发给你的驱动。如果你不能做恰当的处理，你的系统的就会崩溃。

最简单的处理方式是把请求不加改变的传递到我们所绑定的设备上去。如何获得我们所绑定的设备？上一节已经把该设备记录在我们的设备扩展里。

```
nextDeviceObject = ((PSFILTER_DEVICE_EXTENSION)DeviceObject->DeviceExtension)
                        -> AttachedToDeviceObject;
```

如何传递请求？使用 **IoCallDriver,**该调用的第一个参数是设备对象指针，第二个参数是 **IRP** 指针。

一个 **IRP** 拥有一组 **IO_STACK_LOCATION.**前面说过 **IRP** 在一个设备栈中传递。**IO_STACK_LOCATION** 是和这个设备栈对应的。用于保存 **IRP** 请求在当前设备栈位置中的部分参数。如果我要把请求往下个设备传递，那么我应该把当前 **IO_STATCK_LOCATION** 复制到下一个。 但是当我不打算加以任何处理的时候，我简单忽略当前调用栈。

现在可以写一个默认的 **Dispatch Functions.** 简单的 **passthru.**

```
NTSTATUS
SfPassThrough (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )

{
     // 对于我们认为“不能”的事情，我们采用 ASSERT 进行调试模式下的确认。
     // 而不加多余的判断来消耗我们的效率。这些宏在调试模式下不被编译。
    ASSERT(!IS_MY_CONTROL_DEVICE_OBJECT( DeviceObject ));
    ASSERT(IS_MY_DEVICE_OBJECT( DeviceObject ));
    IoSkipCurrentIrpStackLocation( Irp );
    return IoCallDriver( ((PSFILTER_DEVICE_EXTENSION) DeviceObject->DeviceExtension)
->AttachedToDeviceObject, Irp );
}
```

这个函数在以前就出现过。现在可以进一步理解了。

上边有一个函数 **IS_MY_DEVICE_OBJECT** 来判断是否我的设备。这个判断过程很简单。通过 **DeviceObject** 可以得到 **DriverObject** 指针，判断一下是否我自己的驱动即可。**IS_MY_CONTROL_DEVICE_OBJECT ()**来判断这个设备是否是我的控制设备，不要忘记在 **DriverEntry()**中我们首先生成了一个本驱动的控制设备。实际这个控制设备还不做任何事情，所以对它发生的任何请求也是非法的。**ASSERT** 即可。同时我们使用 **IoSkipCurrentIrpStackLocation( Irp );**忽略了当前调用栈空间。

假设我要立刻让一个 **irp** 失败。我可以这样：

```
Irp->IoStatus.Information = 0;
Irp->IoStatus.Status = error_code;
IoCompleteRequest( Irp, IO_NO_INCREMENT );
```

如此一来，本不该发到我的驱动的 **irp**，就立刻返回错误非法请求。但是实际上这种情况是很少发生的。

如果你现在想要你的驱动立刻运行，让所有的 **dispacth functions** 都调用 **SfPassThrough.**这个驱动已经可以绑定文件系统的控制设备，并输出一些调试信息。但是还没有绑定 **Volume.**所以并不能直接监控文件读写。

对于一个绑定文件系统控制设备的设备来说，其他的请求直接调用上边的默认处理就可以了。重点需要注意的是上边曾经挂接 **IRP_MJ_FILE_SYSTEM_CONTROL** 的 **dispatch** 处理的函数 **SfFsControl（）.**

**IRP_MJ_FILE_SYSTEM_CONTROL** 这个东西是 **IRP** 的主功能号。每个主功能号下一般都有次功能号。这两个东西标示一个 **IRP** 的功能。

主功能号和次功能号是 **IO_STACK_LOCATION** 的开头两字节。

当有卷被 **Mount** 或者 **dismount,**你写的 **SfFsControl ()**就被调用。具体的判断方法，就见如下的代码了：

```
NTSTATUS
SfFsControl (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    PIO_STACK_LOCATION irpSp = IoGetCurrentIrpStackLocation( Irp );
    PAGED_CODE();
    ASSERT(!IS_MY_CONTROL_DEVICE_OBJECT( DeviceObject ));
    ASSERT(IS_MY_DEVICE_OBJECT( DeviceObject ));
    switch (irpSp->MinorFunction) {
        case IRP_MN_MOUNT_VOLUME:
            return SfFsControlMountVolume( DeviceObject, Irp );
        case IRP_MN_LOAD_FILE_SYSTEM:
            return SfFsControlLoadFileSystem( DeviceObject, Irp );
        case IRP_MN_USER_FS_REQUEST:
        {
            switch (irpSp->Parameters.FileSystemControl.FsControlCode) {
                case FSCTL_DISMOUNT_VOLUME:
                {
                    PSFILTER_DEVICE_EXTENSION devExt = DeviceObject->DeviceExtension;
                    SF_LOG_PRINT( SFDEBUG_DISPLAY_ATTACHMENT_NAMES,
                                  ("SFilter!SfFsControl:                    Dismounting
volume          %p \"%wZ\"\n",
                                   devExt->AttachedToDeviceObject,
                                   &devExt->DeviceName) );
                    break;
                }
            }
            break;
        }
    }
    IoSkipCurrentIrpStackLocation( Irp );
    return
IoCallDriver(  ((PSFILTER_DEVICE_EXTENSION)DeviceObject->DeviceExtension)->AttachedTo
DeviceObject, Irp );
```

你得开始写新的函数，**SfFsControlMountVolume ()**很快，你就能完全监控所有的卷了。

这样做是 **sfilter** 动态监控所有的卷的完美的解决方案。

但是你会发现这个 **FSCTL_DISMOUNT_VOLUME** 并没有做解除绑定和销毁设备的处理。实际上这个请求的出现是理论性的。测试表明，至少拔出 **U** 盘这样的操作要捕获也是非常不容易的。这个请求似乎根本不会出现。而其他一些请求

会出现，但是往往不是一一对应的关系。所以要真正准确的捕获 **Dismount** 操作是很困难的。

**sfilter** 采用了通融的办法。并不解除绑定也不销毁设备。可能是因为，设备拔除后，多余的设备并没有影响。此外，拔出与插入这样的情况并不会太频繁，所以内存泄漏也不明显。这只是我个人的猜测，我并没有确认过是否有其他的机制能销毁设备。

如果是在 **xp** 以上，有一个调用可以获得一个文件系统上已经被 **Mount** 的卷。但是 **2000** 下不能使用。所以我们没有使用那个方法。何况仅仅得到已经 **Mount** 的卷也不是我想要的。

这里另外还有一个 **SfFsControlLoadFileSystem** 函数。发生于 **IRP_MN_LOAD_FILESYS**。这个功能码的意义是当一个文件识别器（见上文）决定加载真正的文件系统的时候，会产生一个这样的 **irp**。那么，如果我们已经绑定了文件系统识别器，现在就应该解除绑定并销毁设备。同时生成新的设备去绑定真的文件系统。绑定文件系统控制设备我们已经在上一章详细讲过，这里就不再重复追踪这个过程。

你现在可以修改你的驱动，感知卷被绑定的过程。

再回首一下我们的脉络：

第一步.生成一个控制设备。当然此前你必须给控制设置指定名称。

第二步.设置 **Dispatch Functions.** 设置 **Fast Io Functions.**

第三步.编写一个 **File System Notify** 回调函数，在其中绑定刚激活的 **FS CDO.** 并注册这个回调函数。

第四步.编写默认的 **dispatch functions.**

第五步.处理 **IRP_MJ_FILE_SYSTEM_CONTROL,**在其中监控卷设备的 **Mount** 和 **Dismount.**

第六步.下一步自然是绑定卷设备了，请听下回分解。

**7.**准备绑定卷，**IRP** 完成函数,中断级

先讨论一下卷设备是如何得到的.首先在 **SfFsControlMountVolume** 中:

**storageStackDeviceObject = irpSp->Parameters.MountVolume.Vpb->RealDevice;**

**VPB** 是 **Volume parameter block.**一个数据结构.它在这里的主要作用是把实际存储媒介设备对象和文件系统上的卷设备对象联系起来.

你可以从一个 **Storage Device Object** 得到一个 **VPB,** 此外可以从 **VPB** 中再得到对应的卷设备。

这里的 **IRP** 是一个 **MOUNT** 请求.而 **volume** 设备对象实际上是这个请求完成之后的返回结果.因此,在这个请求还没有完成之前,我们就试图去获得 **Volume** 设备对象,当然是竹篮打水一场空了.

既然如此，那么我们应该在 **SfFsControlMountVolume** 请求完成之后，再去获取 **VPB,**得到卷设备。为何要在这里获得 **VPB** 呢？

这是因为在这个过程完成之后，下层的文件系统驱动可能已经修改了 **VPB** 的值。因此，我们这里把它预先保存下来。

这里,你可以直接拷贝当前 **IO_STACK_LOCATION,**然后向下发送请求,但在此之前,要先给 **irp** 分配一个完成函数.**irp** 一旦完成,你的完成函数将被调用.这样的话,你可以在完成函数中得到 **Volume** 设备,并实施你的绑定过程.

这里要讨论一下中断级别的问题.常常碰到人问某函数只能在 **Passive Level** 调用是什么意思.总之我们的任何代码执行的时候,总是处在某个当前的中断级之中.某些系统调用只能在低级别中断级中执行.请注意,如果一个调用可以在高处运行,那么它能在低处运行,反过来则不行.

我们需要知道的只是我们关心 **Passive Level** 和 **Dispatch Level.**而且 **Dispatch Level** 的中断级较高.一般 **ddk** 上都会标明,如果注明 **irq level=dispatch,**那么你就不能在 **passive level** 的代码中调用它们了.

那么你如何判断当前的代码在哪个中断级别中呢?我一般是这么判断的:如果你的代码执行是由于应用程序(或者说上层)的调用而引发的,那么应该在 **Passive Level.**如果你的代码执行是由于下层硬件而引发的,那么则可能在 **dispatch**

level.

希望不要机械的理解我的话。以上只是极为粗略的便于记忆的理解方法.实际的应用应该是这样的:所有的 dispatch functions 由于是上层发来的 irp 而导致的调用,所以应该都是 Passive Level,在其中你可以调用绝大多数系统调用.而如网卡的 OnReceive,硬盘读写完毕,返回而导致的完成函数,都有可能在 Dispatch 级.注意都是有可能,而不是绝对是.但是一旦有可能,我们就应该按就是考虑。

下面是 SfFsControlMountVolume 的执行过程:

```
NTSTATUS
SfFsControlMountVolume (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    PSFILTER_DEVICE_EXTENSION devExt = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION irpSp = IoGetCurrentIrpStackLocation( Irp );
    PDEVICE_OBJECT newDeviceObject;
    PDEVICE_OBJECT storageStackDeviceObject;
    PSFILTER_DEVICE_EXTENSION newDevExt;
    NTSTATUS status;
    BOOLEAN isShadowCopyVolume;
    PFSCTRL_COMPLETION_CONTEXT completionContext;
    PAGED_CODE();
    ASSERT(IS_MY_DEVICE_OBJECT( DeviceObject ));
    ASSERT(IS_DESIRED_DEVICE_TYPE(DeviceObject->DeviceType));
    storageStackDeviceObject = irpSp->Parameters.MountVolume.Vpb->RealDevice;
    // 判断是否卷影，这个我们后面再提
    status = SfIsShadowCopyVolume ( storageStackDeviceObject,
                                    &isShadowCopyVolume );
    // 如果不打算绑定卷影就跳过去
    if (NT_SUCCESS(status) &&
        isShadowCopyVolume &&
        !FlagOn(SfDebug,SFDEBUG_ATTACH_TO_SHADOW_COPIES)) {
        IoSkipCurrentIrpStackLocation( Irp );
        return IoCallDriver( devExt->AttachedToDeviceObject, Irp );
    }
    // 我预先就生成设备，虽然现在还没有到绑定的时候
    status = IoCreateDevice( gSFilterDriverObject,
                             sizeof( SFILTER_DEVICE_EXTENSION ),
                             NULL,
                             DeviceObject->DeviceType,
                             0,
                             FALSE,
                             &newDeviceObject );
    if (!NT_SUCCESS( status )) {
        KdPrint(( "SFilter!SfFsControlMountVolume: Error creating volume device object,
status=%08x\n", status ));
```

```
        Irp->IoStatus.Information = 0;
        Irp->IoStatus.Status = status;
        IoCompleteRequest( Irp, IO_NO_INCREMENT );
        return status;
    }
    // 填写设备扩展
    newDevExt = newDeviceObject->DeviceExtension;
    newDevExt->StorageStackDeviceObject = storageStackDeviceObject;
    RtlInitEmptyUnicodeString( &newDevExt->DeviceName,
                               newDevExt->DeviceNameBuffer,
                               sizeof(newDevExt->DeviceNameBuffer) );

    SfGetObjectName( storageStackDeviceObject,
                     &newDevExt->DeviceName );

    // 后面暂时省略
    … …
}
```

卷影似乎是一种用于磁盘数据恢复的特殊设备。你可以过滤它们也可以不过滤。如何判断这里略去。你可以自己查看 **sfilter** 的相关代码。后面的代码我们省略了。因为出现了新的问题。接下来我们应该做什么呢，显然我们应该获得卷设备并绑定。但是现在卷设备还没有生成，我们必须等待这个请求结束。

当完成函数被调用的时候，请求就结束了。我们可以往完成函数中传递一个上下文指针来保存我们的信息。以便我们知道哪一次调用对应哪一次完成。这有一种经典的同步方法：我们初始化一个事件 **KEVENT,**并通过上下文传递到完成函数中。在完成函数中设置这个事件。而我们的本函数则等待这个事件。那么等待结束时，这个请求就完成了。那么前面省略的地方，基本的代码如下：

```
KEVENT waitEvent;
// 初始化事件
KeInitializeEvent( &waitEvent,
                   NotificationEvent,
                   FALSE );
// 因为我们要等待完成，所以必须拷贝当前调用栈
IoCopyCurrentIrpStackLocationToNext ( Irp );
// 设置完成函数，并把事件的指针当上下文传入。
IoSetCompletionRoutine( Irp,
                        SfFsControlCompletion,
                        &waitEvent,      //上下文指针
                        TRUE,
                        TRUE,
                        TRUE );
// 发送 IRP 并等待事件完成
status = IoCallDriver( devExt->AttachedToDeviceObject, Irp );
if (STATUS_PENDING == status) {
        status = KeWaitForSingleObject( &waitEvent,
```

```
                                        Executive,
                                        KernelMode,
                                        FALSE,
                                        NULL );
        ASSERT( STATUS_SUCCESS == status );
    }
    ……
```

请注意 **IoSetCompletionRoutine** 的第三个参数，就是完成函数中的 **Context** 指针。这是我们传递信息到完成函数的接口。那么在完成函数中，我们这样写**:**

```
NTSTATUS
SfFsControlCompletion (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    )
{
    UNREFERENCED_PARAMETER( DeviceObject );
    UNREFERENCED_PARAMETER( Irp );
    ASSERT(IS_MY_DEVICE_OBJECT( DeviceObject ));
    ASSERT(Context != NULL);
    KeSetEvent((PKEVENT)Context, IO_NO_INCREMENT, FALSE);
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

**UNREFERENCED_PARAMETER** 的意义在于，去掉 **C** 编译器对于没有使用的这个参数所产生的一条警告。

这样一来，只要执行过了 **KeWaitForSingleObject,**则这个请求已经完成。那么在之后执行绑定卷的操作即可。

## **8.**绑定卷的完成

这很容易让人联想，我为何不在完成函数中直接绑定了设备。而非要等完成函数设置了我的事件之后，再回来做这件事情。这是因为完成函数的中断级别过高。虽然 **dispatch** 中断级别应该可以执行 **IoAttachDeviceToDeviceStack**。但是在绑定卷的过程中，**Sfilter** 使用 **ExAcquireFastMutex** 这些不适宜在 **Dispatch** 级别使用的东西，应该是其原因。

用事件等待完成函数的发生是一个通用的办法。但是在 **Windows2000** 上，在绑定卷设备时使用却有其固有的缺陷而有导致死锁的可能。这是 **Windows** 早期的固有缺陷，与一种特殊设备相关。**CardMagic** 曾经深入研究过这个问题，他说到我头晕为止。那么 **Windows2000** 上运行时，我们如何做呢？

解决方法是在完成函数中做这件事情。这又回到上面的问题，为何在完成函数中不能做？最终不得不再次采用折衷的方法：我们在完成函数中生成一个系统线程。系统线程执行的中断级为 **Passive level,**足够我们很好的完成绑定的过程。

**Windows** 本身有一个系统线程负责处理一些日常工作。我们也可以把自己的工作任务插入其中，免除我们需要自己生成线程的开销**:**

**SfFsControlMountVolume** 后面的代码基本如下**,**充满了对目标操作系统的编译时和运行时的判断：

```
… …
#if WINVER >= 0x0501
```

```
    if (IS_WINDOWSXP_OR_LATER()) {
       KEVENT waitEvent;
        KeInitializeEvent( &waitEvent,
                           NotificationEvent,
                           FALSE );
        IoCopyCurrentIrpStackLocationToNext ( Irp );
        IoSetCompletionRoutine( Irp,
                                SfFsControlCompletion,
                                &waitEvent,
                                TRUE,
                                TRUE,
                                TRUE );
        status = IoCallDriver( devExt->AttachedToDeviceObject, Irp );
    if (STATUS_PENDING == status) {
        status = KeWaitForSingleObject( &waitEvent,
                                        Executive,
                                        KernelMode,
                                        FALSE,
                                        NULL );
        ASSERT( STATUS_SUCCESS == status );
    }
    // 到这里请求完成，调用我们的函数绑定卷
    status = SfFsControlMountVolumeComplete( DeviceObject,
                                             Irp,
                                             newDeviceObject );
    } else {
#endif
        completionContext = ExAllocatePoolWithTag( NonPagedPool,
                                                   sizeof( FSCTRL_COMPLETION_CONTEXT ),
                                                   SFLT_POOL_TAG );

        if (completionContext == NULL) {
            IoSkipCurrentIrpStackLocation( Irp );
            status = IoCallDriver( devExt->AttachedToDeviceObject, Irp );
        } else {
            // 初始化一个工作任务，具体内容写在函数 SfFsControlMountVolumeCompleteWorker 中
            ExInitializeWorkItem( &completionContext->WorkItem,
                                  SfFsControlMountVolumeCompleteWorker,
                                  completionContext );
            // 写入上下文，以便把我的多个指针传递过去
            completionContext->DeviceObject = DeviceObject;
            completionContext->Irp = Irp;
            completionContext->NewDeviceObject = newDeviceObject;
            // 拷贝调用栈
            IoCopyCurrentIrpStackLocationToNext( Irp );
```

```
            // 请注意这里传入的上下文变成了我的工作任务，而不是事件，和高级版本有别
            IoSetCompletionRoutine( Irp,
                                    SfFsControlCompletion,
                                    &completionContext->WorkItem, //context parameter
                                    TRUE,
                                    TRUE,
                                    TRUE );
            // 发送 irp
            status = IoCallDriver( devExt->AttachedToDeviceObject, Irp );
        }
#if WINVER >= 0x0501
    }
#endif
    return status;
}
```

那么完成函数也必须相对应的修改一下，以满足需求:

```
NTSTATUS
SfFsControlCompletion (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID Context)
{
    UNREFERENCED_PARAMETER( DeviceObject );
    UNREFERENCED_PARAMETER( Irp );
    ASSERT(IS_MY_DEVICE_OBJECT( DeviceObject ));
    ASSERT(Context != NULL);
#if WINVER >= 0x0501
    if (IS_WINDOWSXP_OR_LATER()) {
        KeSetEvent((PKEVENT)Context, IO_NO_INCREMENT, FALSE);
    } else {
#endif
        // 中断级别过高的时候，工作任务放到 DelayedWorkQueue 队列中执行
        if (KeGetCurrentIrql() > PASSIVE_LEVEL) {
            ExQueueWorkItem( (PWORK_QUEUE_ITEM) Context,
                             DelayedWorkQueue );
        } else {
            // 否则直接执行
            PWORK_QUEUE_ITEM workItem = Context;
            (workItem->WorkerRoutine)(workItem->Parameter);
        }
#if WINVER >= 0x0501
    }
#endif
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

SfFsControlMountVolumeCompleteWorker 做的事情很简单，就是调用 SfFsControlMountVolumeComplete 。我们最终所有的处理目标都是调用 SfFsControlMountVolumeComplete。因为在这里，我们最终绑定卷设备。

```
NTSTATUS
SfFsControlMountVolumeComplete (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PDEVICE_OBJECT NewDeviceObject
    )
{
    PVPB vpb;
    PSFILTER_DEVICE_EXTENSION newDevExt;
    PIO_STACK_LOCATION irpSp;
    PDEVICE_OBJECT attachedDeviceObject;
    NTSTATUS status;
    PAGED_CODE();
    newDevExt = NewDeviceObject->DeviceExtension;
    irpSp = IoGetCurrentIrpStackLocation( Irp );
    // 我们前面保存过的 vpb,获得
    vpb = newDevExt->StorageStackDeviceObject->Vpb;
    if (vpb != irpSp->Parameters.MountVolume.Vpb) {
        if (NT_SUCCESS( Irp->IoStatus.Status )) {
        // 获得一个互斥体,以便我们可以原子的判断我们是否绑定过一个卷设备.这可以防止
        // 我们对一个卷绑定两次。
        ExAcquireFastMutex( &gSfilterAttachLock );
        // 判断是否绑定过了
        if (!SfIsAttachedToDevice( vpb->DeviceObject, &attachedDeviceObject )) {
            // 调用 SfAttachToMountedDevice 来完成真正的绑定.
            status = SfAttachToMountedDevice( vpb->DeviceObject,
                                              NewDeviceObject );
            if (!NT_SUCCESS( status )) {
                SfCleanupMountedDevice( NewDeviceObject );
                IoDeleteDevice( NewDeviceObject );
            }
            ASSERT( NULL == attachedDeviceObject );
        } else {
                        ((PSFILTER_DEVICE_EXTENSION)attachedDeviceObject->
DeviceExtension)-> AttachedToDeviceObject,
                        &newDevExt->DeviceName) );
            SfCleanupMountedDevice( NewDeviceObject );
            IoDeleteDevice( NewDeviceObject );
            ObDereferenceObject( attachedDeviceObject );
        }
        ExReleaseFastMutex( &gSfilterAttachLock );
```

```
        } else {
            SfCleanupMountedDevice( NewDeviceObject );
            IoDeleteDevice( NewDeviceObject );
        }
        // 把请求完成掉
        status = Irp->IoStatus.Status;
        IoCompleteRequest( Irp, IO_NO_INCREMENT );
        return status;
    }
```

这个过程确实比较复杂,但是既然最后调用的是 SfAttachToMountedDevice,那么我们最后的真相也不远了:

```
    NTSTATUS
    SfAttachToMountedDevice (
        IN PDEVICE_OBJECT DeviceObject,
        IN PDEVICE_OBJECT SFilterDeviceObject
        )
    {
        PSFILTER_DEVICE_EXTENSION newDevExt = SFilterDeviceObject->DeviceExtension;
        NTSTATUS status;
        ULONG i;
        PAGED_CODE();
        ASSERT(IS_MY_DEVICE_OBJECT( SFilterDeviceObject ));
    #if WINVER >= 0x0501
        ASSERT(!SfIsAttachedToDevice ( DeviceObject, NULL ));
    #endif
        // 设备标记的复制
        if (FlagOn( DeviceObject->Flags, DO_BUFFERED_IO )) {
            SetFlag( SFilterDeviceObject->Flags, DO_BUFFERED_IO );
        }
        if (FlagOn( DeviceObject->Flags, DO_DIRECT_IO )) {
            SetFlag( SFilterDeviceObject->Flags, DO_DIRECT_IO );
        }
        // 循环尝试绑定.绑定有可能失败。这可能和其他用户恰好试图对这个磁盘做特殊的操作比如
        // mount 或者 dismount 有关.反复进行 8 次尝试以避开这些巧合.
        for (i=0; i < 8; i++) {
            LARGE_INTEGER interval;
            status = SfAttachDeviceToDeviceStack( SFilterDeviceObject,
                                                  DeviceObject,
                                                  &newDevExt->AttachedToDeviceObject );
            if (NT_SUCCESS(status)) {
                ClearFlag( SFilterDeviceObject->Flags, DO_DEVICE_INITIALIZING );
                return STATUS_SUCCESS;
            }
            // 把这个线程延迟 500 毫秒后再继续.
            interval.QuadPart = (500 * DELAY_ONE_MILLISECOND);
    KeDelayExecutionThread (  KernelMode, FALSE, &interval );
```

```
    }
    return status;
}
```

大结局是我们完成了绑定,过滤可以开始了。

## 9 读写操作的捕获与分析

上文已经讲到绑定 Volume 之前的关键操作.我们一路逢山开路,逢水架桥,相信你从中也学到了驱动开发的基本方法.后的工作,无非灵活运用这些方法而已.而以后的教程中,我也不会逐一详尽的列举出细节的代码了.

现在我们处理 IRP_MJ_READ 和 IRP_MJ_WRITE,如果你已经绑定了 Volume,那么显然,发送给 Volume 的请求就会先发送给你.处理 IRP_MJ_READ 和 IRP_MJ_WRITE,能捕获文件的读写操作.

进入你的 SfRead ()/SfWrite 函数(假设你注册了这两个函数来处理读写,请见前面关于分发函数的讲述),首先判断这个 Dev 是不是绑定 Volume 的设备.如果是,那么就是一个读写文件的操作.

如何判断?记得我们先绑定 Volume 的时候,在我们的设备扩展中设置了,如果不是(比如是 FS CDO,我们没设置过),那么这么判断即可:

```
PSFILTER_DEVICE_EXTENSION devExt = DeviceObject->DeviceExtension;
if(devExt->StorageDev != NULL)
{ ... }
```

其他的情况不需要捕获,请直接传递到下层.

读写请求的 IRP 情况非常复杂,请有足够的心理准备.并不要过于依赖帮助,最好的办法就是自己打印 IRP 的各个细节,亲自查看文件读操作的完成过程.

首先我们回忆一下前面分发函数的设置:

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = SfCreate;
DriverObject->MajorFunction[IRP_MJ_CREATE_NAMED_PIPE] = SfCreate;
DriverObject->MajorFunction[IRP_MJ_CREATE_MAILSLOT] = SfCreate;
DriverObject->MajorFunction[IRP_MJ_FILE_SYSTEM_CONTROL] = SfFsControl;
DriverObject->MajorFunction[IRP_MJ_CLEANUP] = SfCleanupClose;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = SfCleanupClose;
```

这里没有 Read 的设置，为此后面加上一条:

```
DriverObject->MajorFunction[IRP_MJ_READ] = SfRead;
```

然后我们自己实现一个函数:

```
NTSTATUS
SfRead (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp)
{
    PIO_STACK_LOCATION irpsp = IoGetCurrentIrpStackLocation(Irp);
    PFILE_OBJECT file_object = irpsp->FileObject;
    PSFILTER_DEVICE_EXTENSION devExt = DeviceObject->DeviceExtension;

    // 对控制设备的操作，我直接失败
    if (IS_MY_CONTROL_DEVICE_OBJECT(DeviceObject)) {
```

```
        Irp->IoStatus.Status = STATUS_INVALID_DEVICE_REQUEST;
        Irp->IoStatus.Information = 0;
        IoCompleteRequest( Irp, IO_NO_INCREMENT );
        return STATUS_INVALID_DEVICE_REQUEST;
    }
    // 对文件系统其他设备的操作，passthru.
    if(devExt->StorageDev != NULL)
    {
        return SfPassThrough(DeviceObject,Irp);
    }

    // 到这里说明是对卷的文件操作
    … …

}
```

然后关心被读写的文件.IRP 下有一个 FileObject 指针.这个东西指向一个文件对象.你可以得到文件对象的名字,但是实际上在读操作的过程中解析路径是不合理的。我们在后面“全路径过滤”中，再详细讲解文件名和路径的获取。

接下来有得到读文件的偏移量. 2000 下文件系统得到的偏移量都是从文件起始位置开始计算的.偏移量是一个 LARGE_INTEGER.这是一个在 Windows 驱动开发中常用的 64 位整数数据的共用体。

以下代码用来得到偏移量（从 irpsp 中）：

```
LARGE_INTEGER offset;
Offset.QuadPart = irpsp->Parameters.Read.ByteOffset.QuadPart;
```

而读取文件的长度则是:

```
ULONG length;
length = irpsp->Parameters.Read.Length;
```

写的偏移量和长度则为:

```
Offset.QuadPart = irpsp->Parameters.Write.ByteOffset.QuadPart;
length = irpsp->Parameters.Write.Length;
```

此外我还希望能得到我所读到的数据.这要注意,我们捕获这个请求的时候,这个请求还没有完成.既然没有完成,当然无数据可读.如果要获取,那就设置完成函数,在完成函数中完成请求.

完成 Irp 的时候忽略还是拷贝当前 IO_STACK_LOCATION,返回什么 STATUS,以及完成函数中如何结束 Irp,是不那么容易搞清楚的一件事情.我想做个总结如下:

1.如果对 irp 完成之后的事情无兴趣,直接忽略当前 IO_STACK_LOCATION,(对我们的程序来说,调用 IoSkipCurrentIrpStackLocation),然后向下传递请求,返回 IoCallDriver 所返回的状态.

2.不但对 irp 完成之后的事情无兴趣,而且我不打算继续传递,打算立刻返回成功或失败.那么我不用忽略或者拷贝当前 IO_STACK_LOCATION,填写参数后调用 IoCompleteRequest,并返回我想返回的结果.

3.如果对 irp 完成之后的事情有兴趣,并打算在完成函数中处理,应该首先拷贝当前 IO_STACK_LOCATION(IoCopyCurrentIrpStackLocationToNext),然后指定完成函数,并返回 IoCallDriver()所返回的 status.完成函数中,不需要调用 IoCompleteRequest!直接返回 Irp 的当前状态即可.

4.同 3 的情况,有时候,会把任务塞入系统工作者线程或者希望在另外的线程中去完成 Irp,那么完成函数中应该返回 STATUS_MORE_PROCESSING_REQUIRED,此时完成 Irp 的时候应该调用 IoCompleteRequest.另一种类似的情况是在 dispatch 函数中等待完成函数中设置事件,那么完成函数返回 STATUS_MORE_PROCESSING_REQUIRED,dispatch 函数在等待结束后调用 IoCompleteRequest.

前边已经提到过设备的DO_BUFFERED_IO,DO_DIRECT_IO这两个标记.情况是3种:要么是两个标记中其中一个,要么是一个都没有.Volume 设备出现 DO_BUFFERED 的情况几乎没有,我碰到的都是一个标记都没有.DO_DIRECT_IO表示数据应该返回到Irp->MdlAddress所指向的MDL所指向的内存.在无标记的情况下,表明数据读好,请返回到
Irp->UseBuffer中即可.

不过在实际中，我都用更简单的方法判别.简单的说，Irp->MdlAddress 如果不为 NULL,则使用Irp->MdlAddress.缓冲区位置为 MmGetSystemAddressForMdl(Irp->MdlAddress);否则直接使用Irp->UserBuffer.

UseBuffer是一个只在当前线程上下文才有效的地址.如果你打算按这个地址获得数据,除非你打算自己分配MDL锁定内存地址，否则你必须在当前线程上下文中.完成函数与 SfRead 并非同一个线程.所以在完成函数中按这个地址去获取数据是不对的.如何回到当前线程?我采用简单的办法.在SfRead中设置一个事件,调用IoCallDriver之后开始等待这个事件.而在完成函数中设置这个事件.这样等待结束的时候,刚好Irp已经完成,我也回到了我的SfRead原来的线程.

那么，获得读取内容的主要方法如下:

```
KEVENT waitEvent;
KeInitializeEvent( &waitEvent,
                   NotificationEvent,
                   FALSE );
IoCopyCurrentIrpStackLocationToNext ( Irp );
IoSetCompletionRoutine( Irp,
                        SfReadCompletion,
                        &waitEvent,
                        TRUE,
                        TRUE,
                        TRUE );
status = IoCallDriver( devExt->AttachedToDeviceObject, Irp );
if (STATUS_PENDING == status) {
    status = KeWaitForSingleObject( &waitEvent,
                                    Executive,
                                    KernelMode,
                                    FALSE,
                                    NULL );
    ASSERT( STATUS_SUCCESS == status );
}
```

到这里请求已经完成,可以去获得读取的内容了，在irp->UserBuffer或者irp->MdlAddress中。

这一段就是前面 SfFsControlMountVolume 中对应段落的拷贝,要求是一样的,就是把请求完成掉.SfReadCompletion的内容也是一样的，就是设置一下事件.

至于写操作的内容,则不用完成，可以直接从irp->UserBuffer或irp->MdlAddress中得到.

## 10.读请求的完成

尽管我们得到了读过程的所有参数和结果,我们依然不知道如果自己写一个文件系统,该如何完成读请求,或者过滤驱动中,如何修改读请求.

除非是一个完整的文件系统，完成读操作似乎是不必要的。过滤驱动一般只需要把请求交给下层的实际文件系统来完成。但是有时候比如加解密操作，我希望从下层读到数据，解密后，我自己来完成这一 IRP 请求。

这里要谈到 IRP 的 minor function code.以前已经讨论到如果 major function code 是 IRP_MJ_READ 则是 Read 请求。实际上有些主功能号下面有一些子功能号，如果是 IRP_MJ_READ,检查其 MINOR,应该有几种情况：IRP_MN_NORMAL,IRP_MN_MDL,IRP_MN_MDL|IRP_COMPLETE( 这个其实就是 IRP_MN_MDL_COMPLETE).还有其他几种情况，资料上有解释，但是我没自己调试过，也就不说了。只拿自己调试过的几种情况来说说。

IRP_MN_NORMAL 的情况完全与上一节同

注意如上节所叙述，IRP_MN_NORMAL 的情况，既有可能是在 Irp->MdlAddress 中返回数据，也可能是在 Irp->UserBuffer 中返回数据，这个取决于 Device 的标志.

但是如果次功能号为 IRP_MN_MDL 则完全不是这个意思。这种 irp 一进来看数据，就赫然发现 Irp->MdlAddress 和 Irp->UserBuffer 都为空。那你得到数据后把数据往哪里拷贝呢?

IRP_MN_MDL 的要求是请自己分配一个 MDL,然后把 MDL 指向你的数据所在的空间，然后返回给上层。自然 MDL 是要释放的，换句话说事业使用完毕要归还，所以又有 IRP_MN_MDL_COMPLETE,意思是一个 MDL 已经使用完毕，可以释放了。

MDL 用于描述内存的位置。据说和 NDIS_BUFFER 用的是同一个结构。这里不深究，我写一些函数来分配和释放 mdl，并把 mdl 指向内存位置或者得到 mdl 所指向的内存：

```
// 这个函数分配 mdl,缓冲必须是非分页的。可以在 dispatch level 运行。
_inline PMDL MyMdlAllocate(PVOID buf, ULONG length)
{
    PMDL pmdl = IoAllocateMdl(buf,length,FALSE,FALSE,NULL);
    if(pmdl == NULL)
        return NULL;
    MmBuildMdlForNonPagedPool(pmdl);
    return pmdl;
}

// 这个函数分配一个 mdl,并且带有一片内存
_inline PMDL MyMdlMemoryAllocate(ULONG length)
{
    PMDL mdl;
    void *buffer = ExAllocatePool (NonPagedPool,length);
    if(buffer == NULL)
        return NULL;
    mdl = MyMdlAllocate (buffer,length);
    if(mdl == NULL)
    {
        ExFreePool(buffer);
        return NULL;
    }
    return mdl;
}

// 这个函数释放 mdl 并释放 mdl 所带的内存。
```

```
_inline void MyMdlMemoryFree(PMDL mdl)
{
    void *buffer = MmGetSystemAddressForMdlSafe(mdl,NormalPagePriority);
    IoFreeMdl(mdl);
    ExFreePool(buffer);
}
```

要完成请求还有一个问题。就是 **irp->IoStatus.Information.**在这里你必须填上实际读取得到的字节数字。不然上层不知道有多少数据返回。这个数字不一定与你的请求的长度等同（其实我认为几乎只要是成功，就应该都是等同的，唯一的例外是读取到文件结束的地方，长度不够了的情况）。必须设置这个数值：

```
irp->IoStatus.Information = infor;
```

也许你都烦了，但是还有事情要做。作为读文件的情况，如果你是自己完成请求，不能忘记移动一下文件指针。否则操作系统会不知道文件指针移动了而反复读同一个地方永远找不到文件尾，我碰到过这样的情况。 一般是这样的，如果文件读取失败，请保持原来的文件指针位置不要变。如果文件读取成功，请把文件指针指到"读请求偏移量+成功读取长度"的位置。

这个所谓的指针是指 **Irp->FileObject->CurrentByteOffset.**

我跟踪过正常的 **windows** 文件系统的读行为，我认为并不一定是向我上边说的这样做。情况很复杂，有时动，有时不动（说复杂当然是因为我不理解），但是按我上边说的方法来完成，我还没有发现过错误。

现在看看怎么完成这些请求，假设我已经有数据了。这些当然都是在 **SfRead** 中或者是其他想完成这个 **irp** 的地方做的（希望你还记得我们是如何来到这里），假设其他必要的判断都已经做了：

```
switch(irpsp->MiniorFunction)
{
    // 我先保留文件的偏移位置
    case IRP_MN_NORMAL:
    {
        Void *buffer;
        if(Irp->MdlAddress != NULL)
            buffer                                                              =
MmGetSystemAddressForMdlSafe(irp->MdlAddress,NormalPagePriority)
        else
            buffer = Irp->UserBuffer;
        ... ...  // 如果有数据，就往 buffer 中写入...
        Irp->IoStatus.Information = length;
        Irp-> IoStatus.Status = STATUS_SUCCESS;
        Irp->FileObject->CurrentByteOffset.Quat = offset.Quat+length;
        IoCompleteRequest( Irp, IO_NO_INCREMENT );
        return STATUS_SUCCESS;
    }
    case IRP_MN_MDL:
    {
        PMDL mdl = MyMdlMemoryAllocate (length); // 情况比上边的复杂，请先分配 mdl
        if(mdl == NULL)
        {
        // ... 返回资源不足 ...
        }
        Irp->MdlAddress = mdl;;
```

```
            ... ...   // 如果有数据，就往 MDL 的 buffer 中写入...

            Irp->IoStatus.Information = length;
            Irp-> IoStatus.Status = STATUS_SUCCESS;
            Irp->FileObject->CurrentByteOffset.Quat = offset.Quat+length;
            IoCompleteRequest( Irp, IO_NO_INCREMENT );
            return STATUS_SUCCESS;
        }
        case IRP_MN_MDL_COMPLETE:
        {
            // 没有其他任务，就是释放 mdl
            Irp->IoStatus.Information = length;
            Irp-> IoStatus.Status = STATUS_SUCCESS;
            Irp->FileObject->CurrentByteOffset.Quat = offset.Quat+length;
            IoCompleteRequest( Irp, IO_NO_INCREMENT );
            return STATUS_SUCCESS;
        }
        default:
        {
            // 我认为其他的情况不过滤比较简单 ...
        }
    }
```

重要提醒：**IRP_MN_MDL** 的情况，需要分配一个 **mdl**，并且这个 **mdl** 所带有的内存是有一定长度的，这个长度必须与后来的 **irp->IoStatus.Information** 相同！似乎上层并不以 **irp->IoStatus.Information** 返回的长度为准。比如明明只读了 **50** 个字节，但是你返回了一个 **mdl** 指向内存长度为 **60** 字节，则操作系统则认为已经读了 **60** 个字节！这非常糟糕。

最后提一下文件是如何结尾的。如果到某一处，返回成功，但是实际读取到的数据没有请求的数据长，这时还是返回 **STATUS_SUCCESS,**但是此后操作系统会马上发 **irp** 来读最后一个位置，此时返回长度为 **0**，返回状态 **STATUS_FILE_END** 即可。

已经解释了读请求。我不会再讲解写请求了。相信读者有能力自己搞清楚。

**11.**文件和目录的生成打开，关闭与删除

我们已经分析了读，写与读类似。文件系统还有其他的操作。比如文件或目录的打开（打开已经存在的或者创建新的）**,** 关闭。文件或目录的移动，删除。

实际上 **FILE_OBJECT** 并不仅仅指文件对象。在 **windows** 文件系统中，目录和文件都是用 **FileObject** 来抽象的。这里产生一个问题，对于一个已经有的 **FileObject,**我如何判断这是一个目录还是一个文件呢**?**

对于一个已经存在的 **FileObject,**我没有找到除了发送 **IRP** 来向卷设备询问这个 **FileObject** 的信息之外更好的办法。自己发送 **IRP** 很麻烦。不是我很乐意做的那种事情。但是 **FileObject** 都是在 **CreateFile** 的时候诞生的。在诞生的过程中，确实有机会得到这个即将诞生的 **FileObject,**是一个文件还是一个目录。

**Create** 的时候，获得当前 **IO_STACK_LOCATION,**假设为 **irpsp,**那么 **irpsp->Parameters.Create** 的结构

为：

```
struct {
    PIO_SECURITY_CONTEXT SecurityContext;
    ULONG Options;
    USHORT FileAttributes;
    USHORT ShareAccess;
    ULONG EaLength;
};
```

这个结构中的参数是与 CreateFile 这个 api 中的参数对应的，请自己研究。取得方法如下：

```
ULONG options = irpsp->Parameters.Create.Options;
file_attributes = irpsp->Parameters.Create.FileAttributes;
```

options 中有一个 FILE_DIRECTORY_FILE;

file_attribute 中有一个 FILE_ATTRIBUTE_DIRECTORY;

然后我们搞清上边 Options 和 FileAttributes 的意思。是不是 Options 里边有 FILE_DIRECTORY_FILE 标记就表示这是一个目录?实际上，CreateOpen 是一种尝试性的动作。无论如何，我们只有当 CreateOpen 成功的时候，判断 FileObject 才有意义。否则是空谈。

成功有两种可能，一是已经打开了原有的文件或者目录，另一种是新建立了文件或者目录。Options 里边带有 FILE_DIRECTORY_FILE 表示打开或者生成的对象是一个目录。那么，如果在 Create 的完成函数中，证实生成或者打开是成功的，那么返回得到的 FILE_OBJECT,确实应该是一个目录。

当我经常要使用我过滤时得到的文件或者目录对象的时候，我一般在 Create 成功的的时候捕获他们，并把他们记录在一个"集合"中。这时你得写一个用来表示"集合"的数据结构。你可以用链表或者数组，只是注意保证多线程安全性。因为 Create 的时候已经得到了属性表示 FileObject 是否是目录，你就没有必要再发送 IRP 来询问 FileObject 的 Attribute 了。

还上边的 FileAttributes,似乎这个东西并不可靠。因为在生成或者打开的时候，你只需要设置 Options。我认为这个字段并无法说明你打开的文件对象是目录。

这你需要设置一下 Create 的完成函数。请参考上边对文件读操作。

```
NTSTATUS SfCreateComplete(
    IN DEVICE_OBJECT *DeviceObject,
    IN IRP *irp,
    IN PVOID context)
{
    PIO_STACK_LOCATION irpsp = IoGetCurretIrpStackLocation(irp);
    PFILE_OBJECT file = irpsp->FileObject;
    UNREFERENCED_PARAMETER(DeviceObject);
    if(NT_SUCCESS(irp->IoStatus.Status)
    {
        // 如果成功了,把这个 FileObject 记录到集合里,这是一个
        // 刚刚打开或者生成的目录
        if(file && (irpsp->Parameters.Create.Options & FILE_DIRECTORY_FILE) != 0)
            MyAddObjToSet (file); // 把 FileObject 保存到一个集合里。这个函数请自己实现.
        return irp->IoStatus.Status;
    }
}
```

这里顺便解释一下 UNREFERENCED_PARAMETER 宏。我曾经不理解这个宏的意思。其实就是因为本函数传入了

三个参数，这些参数你未必会用到。如果你不用的话，大家知道 c 编译器会发出一条警告。一般认为驱动应该去掉所有的警告，所以用了这个宏来"使用"一下没有用到过的参数。你完全可以不用他们。

现在所有的目录都被你记录。那么得到一个 **FileObject** 的时候，判断一下这个 **FileObject** 在不在你的集合里，如果在，就说明是目录，反之是文件。

当这个 **FileObject** 被关闭的时候你应该把它从你的集合中删除。你可以捕获 **Cleanup** 的 **IRP** 来做这个。因为判断 **FileObject** 是文件还是目录的问题，我们已经见识了文件的打开和关闭工作。

现在看一下文件是如何被删除的。

删除的操作，第一步是打开文件，打开文件的时候必须设置为可以删除。如果打开失败，则直接导致无法删除文件。第二步设置文件属性为用于删除，第三步关闭文件即可。关闭的时候，文件被系统删除。

不过请注意这里的"删除"并非把文件删除到回收站。如果要测试，你必须按住 **shift** 彻底删除文件。文件删除到回收站只是一种改名操作。改名操作我们留到以后再讨论。

第一步是打开文件，我应该可以在文件被打开的时候，捕获到的 **irpsp** 的参数，记得前边的参数结构，中间有：

```
PIO_SECURITY_CONTEXT SecurityContext;
```

相关的结构如下:

```
typedef struct _IO_SECURITY_CONTEXT {
    PSECURITY_QUALITY_OF_SERVICE SecurityQos;
    PACCESS_STATE AccessState;
    ACCESS_MASK DesiredAccess;
    ULONG FullCreateOptions;
} IO_SECURITY_CONTEXT, *PIO_SECURITY_CONTEXT;
```

注意其中的 **DesiredAccess**，其中必须有 **DELETE** 标记，才可以删除文件。

第二步是设置为"关闭时删除"。这是通过发送一个 **IRP(Set Information)**来设置的。捕获主功能码为 **IRP_MJ_SET_INFORMATION** 的 **IRP** 后：

首先，**IrpSp->Parameters.SetFile.FileInformationClass** 应该为 **FileDispositionInformation**。

然后，**Irp->AssociatedIrp.SystemBuffer** 指向一个如下的结构：

```
typedef struct _FILE_DISPOSITION_INFORMATION {
    BOOLEAN DeleteFile;
} FILE_DISPOSITION_INFORMATION;
```

如果 **DeleteFile** 为 **TRUE,**那么这是一个删除文件的操作。文件将在这个 **FileObject Close** 的时候被删除。

以上的我都未实际调试，也不再提供示例的代码。有兴趣的读者请自己完成。

## 12 自己发送 Irp 完成读请求

关于这个有一篇文档解释得很详细，不过我认为示例的代码有点太简略了，这篇文档在 **IFS** 所附带的 **OSR** 文档中,名字为"**Rolling Your Own**"，请自己寻找。

为何要自己发送 **Irp?**在一个文件过滤驱动中，如果你打算读写文件，可以试用 **ZwReadFile.**但是这有一些问题。**Zw** 系列的 **Native API** 使 用句柄。一般句柄是有线程环境限制的。此外也有中断级别的限制。使用内核句柄要好一些。但是，**Zw** 系列函数来读写文件，最终还是要发出 **Irp,**又会被自己的过滤驱动捕获到。结果带来重入的问题。对资源也是浪费。那么最应该的办法是什么呢？当然是直接对卷设备发 **Irp** 了。

但是 **Irp** 是非常复杂的数据结构，而且又被微软所构造的很多未空开的部件所处理。所以自己发 **irp** 并不是一件简单的事情。 比较万能的方法是 **IoAllocateIrp,**分配后自己逐个填写。问题是细节实在太多，很多无文档可寻。有兴趣的应该看看我上边所提及的

那篇文章"**Rolling Your Own**"。

有意思的是这篇文章后来提到了捷径，就是利用三个函数：

**IoBuildAsynchronousFsdRequest(...)**

**IoBuildSynchronousFsdRequest(...)**

**IoBuildDeviceIoControlRequest(...)**

于是我参考了他这方面的示例代码，发现运行良好，程序也很简单。建议怕深入研究的选手就可以使用我下边提供的方法了。

首先的建议是使用 **IoBuildAsynchronousFsdRequest(),**而不要使用同步的那个。使用异步的 **Irp** 使 **irp** 和线程无关。而你的过滤驱动一 般很难把握当前线程（如果你开一个系统线程来专门读取文件那例外）。此时，你可以轻松的在 **Irp** 的完成函数中删除你分配过的 **Irp,**避免去追究和线程相关的事情。

但是这个方法有局限性。文档指出，这个方法仅仅能用于 **IRP_MJ_READ,IRP_MJ_WRITE,IRP_MJ_FLUSH_BUFFERS,**和 **IRP_MJ_SHUTDOWN.**

刚好我这里仅仅要求完成文件读。

用 **Irp** 完成文件读需要一个 **FILE_OBJECT.FileObject** 是比 **Zw** 系列所用的句柄更好的东西。因为这个 **FileObject** 是和线程无关的。你可以放心的在未知的线程中使用他。

自己要获得一个 **FILE_OBJECT** 必须自己发送 **IRP_MJ_CREATE** 的 **IRP.**这又不是一件轻松的事情。不过我跳过了这个问题。因为我是文件系 统过滤驱动，所以我从上面发来的 **IRP** 中得到 **FILE_OBJECT,**然后再构造自己的 **IRP** 使用这个 **FILE_OBJECT,**我发现运行很好。

在后面的”解决重入问题”一章中，我们给出自己打开文件,并跳过重入的简单方法.

但是又出现一个问题，如果 **IRP** 的 **irp->Flags** 中有 **IRP_PAGING**（或者说是 **Cache** 管理器所发来的 **IRP**）标记，则其 **FileObject** 我获得并使用后，老是返回错误。阅读网上的经验表明，带有 **IRP_PAGINGE** 的 **FileObject** 不可以使用.于是我避免使用这时的 **FileObject.**我总是使用不带 **IRP_PAGING** 的 **Irp**（认为是用户程序发来的读请求）的 **FileObject**。

好，现在废话很多了，现在来看看构造 **irp** 的代码： **//** 构造 **IRP**

```
if(read_or_write)
    irp = IoBuildAsynchronousFsdRequest(
        IRP_MJ_READ,dev,buffer,*length,offset,NULL);
else
    irp = IoBuildAsynchronousFsdRequest(
        IRP_MJ_WRITE,dev,buffer,*length,offset,NULL);
if(irp == NULL)
{
    return STATUS_INSUFFICIENT_RESOURCES;
}
irp->Flags = 0x43;      // 这是我喜欢的一种 Flags.建议你查 DDK 帮助然后填写标准的宏
KeInitializeEvent(&event,NotificationEvent,FALSE);
my_context.event = &event;        // 为了可以等待完成,我设置一个事件传入
IoSetCompletionRoutine(irp,MyIrpComplete,&my_context,TRUE,TRUE,TRUE);
```

**buffer** 是缓冲。在 **Irp** 中被用做 **UserBuffer** 接收数据。**offset** 是 这次读的偏移量。以上代码构造一个读 **irp.**请注意，此时您还没有设置 **FileObject.**实际上我是这样发出请求的：

```
irpsp = IoGetNextIrpStackLocation(irp);
irpsp->FileObject = file;
status = IoCallDriver(dev,irp);
irp = NULL;
if(status == STATUS_PENDING)
```

```
    KeWaitForSingleObject(&event,Executive,KernelMode,FALSE,NULL);
// 到这里请求就完成了,请做自己的处理...
```

再看看 **MyIrpComplete** 如何收场：

```
// 一个通用的 irp 完成函数
static NTSTATUS MyIrpComplete (
            PDEVICE_OBJECT dev,
            PIRP irp,
            PVOID context)
{
    // 设置事件
    PMY_READ_CONTEXT my_context = (PMY_READ_CONTEXT)context;
    KeSetEvent(my_context->event,IO_NO_INCREMENT,FALSE);
    my_context->information = irp->IoStatus.Information;
    my_context->status = irp->IoStatus.Status;
    // 释放 irp,过程非常复杂
    if (irp->MdlAddress)
    {
        MmUnmapLockedPages(
            MmGetSystemAddressForMdl(irp->MdlAddress),
            irp->MdlAddress);
        MmUnlockPages(irp->MdlAddress);
        IoFreeMdl(irp->MdlAddress);
    }
    IoFreeIrp(irp);
    // 返回处理未结束.
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

## 13 如何实现路径过滤

文件过滤系统中很多过滤都是以操作的文件的路径作为过滤条件的.比如你想控制某个目录下的文件被加密.或者是被映射到其他地方.或者是其他的操作,你都必须用到路径过滤.但是即使是这么常见的一个操作,在 **IFSDDK** 的基础上做起来也绝对不是那么简单.

取得文件的路径有三种情况:

第一:在文件打开之前从打开文件的请求中提取路径.**FileMon** 从 **FileObject->FileName** 中提取文件路径其实就是对这个的实现.后面我们讨论这种方法的困难.

第二:在文件 **CREATE IRP** 处理结束后获取路径.这是 **SFilter** 演示了的也是最容易解决的.

第三:在文件过滤其他 **IRP** 时,(如改名,查询,设置,读,写)的时候得到 **FileObject** 所对应的文件路径.

以上三种情况以第一种路径获取最为麻烦.第二种情况最为简单.我先介绍一下我在第二种情况下的做法.

基本做法与 SFilter 同.因为此时 FileObject 已经生成结束,对于这个对象,你可以使用 ObQueryNameString.这个调用返回一个路径.对某一个文件对象,返回结果大致如此:

"\Device\HardDiskVolume1\MyDirectory\MyFile.Name"

这里再次强调一下,必须在 Create 完成之后再进行这个调用.如果直接在 sfCreate()函数的早期调用这个,你只能得到:

"\Device\HardDiskVolume1"

这个也有一定的作用,可以设法获得盘符.但是得不到完整路径.

另一个需要注意的是,ObQueryNameString 只能在 CREATE IRP 和 CLEAN UP IRP 的处理中使用,除非你能自己下发 IRP,一般都会使系统进入死锁.

下面的问题是如何获得盘符.盘符的取得是要把"\Device\HardDiskVolume1"这样的名字转换为"C:"这样的符号连接名.也可以直接用卷设备的 DEVICE_OBJECT 去获取.这可以用到两个函数:

```
NTSTATUS
    RtlVolumeDeviceToDosName(
        IN PVOID VolumeDeviceObject,
        OUT PUNICODE_STRING DosName);
NTSTATUS
    IoVolumeDeviceToDosName(
        IN PVOID VolumeDeviceObject,
        OUT PUNICODE_STRING DosName);
```

其中第二个函数据称是更高级的版本.但是要 XP 以上系统才支持.2K 只有 RtlVolumeDeviceToDosName.这给你编写 2K 和 XP 兼容的驱动带来困难,因为你不得不动态导入 IoVolumeDeviceToDosName,否则你的驱动在 2K 下可能无法加载.

但是这两个函数在 2K 下尤其是驱动静态加载的时候都似乎都有问题.在 2K+SP4 的情况下一般有 IoVolumeDeviceToDosName 函数的导出存在.我两个函数都试验了.每次都是动态加载没有问题,而静态加载过程中调用却会在 IoVolumeDeviceToDosName 或 IoVolumeDeviceToDosName 中死机.似乎有一个导致死锁的 Device Io Control IRP 被发到某一个设备导致系统死了.具体的原因我不清楚,而且也不知道是否是我编程的错误导致的.但是这种情况使我只好寻找更加可靠的办法.如果你有正确的方法,希望你能发一个邮件给我(mfc_tan_wen@163.com).

既然"C:","D:"这样的东西其实是符号连接,对应了"\Device\HardDiskVolume1","\Device\HardDiskVolume2"这样的设备的话,我可以用 ZwQuerySymbolicLinkObject 查询这个符号连接,找出它所对应的实际名,然后与"\Device\HardDiskVolume1"这样的字符串做比较,从而把"\Device\HardDiskVolume1"这样的设备名字转变为盘符.

我自己定义了一个能动态分配内存的 wd_ustr_h 字符串来代替 UNICODE_STRING.有兴趣的读者可以自己实现它或者直接用 UNICODE_STRING.

```
// 以下的代码得到一个符号连接的目标
wd_ustr_h wd_symbolic_target_ustr(PUNICODE_STRING symbolic)
{
    ... // 初始化对象特性表,代码被省略,请察看源代码...
    InitializeObjectAttributes( ... ...
    // 打开符号联接
    status = ZwOpenSymbolicLinkObject(
        &link_handle,
        GENERIC_READ,
```

```
            &attributes);
            if(!NT_SUCCESS(status))
                return NULL;
        wd_ustr_init_em(&target, buf, 8*sizeof(WCHAR));
        // 查询符号联接对象
        status = ZwQuerySymbolicLinkObject(link_handle, &target, &length);
        ...   // 判断返回值并分配空间等代码被省略...
        if(NTSUCCESS(status))
        {
            // 保存一个目标字符串句柄指针,并返回
            target_ret = wd_ustr_h_alloc_from_ustr(&target);
        }
        ... ...
        ZwClose(link_handle);
        return target_ret;
    }
```

下面的方法是把一个类似"\Device\HardDiskVolume2"这样的字符串和"C:"-"Z:"所有的目标进行比较,以得到正确的盘符:

```
wd_ustr_h wd_vol_name_dos_name(wd_wchar *name)
{
  // 符号连接的全称应该是 L"\\DosDevices\\X:".X 可以替换成 C-Z 任意一个字母.这里没有考虑 A,B 两个软驱.
  wd_wchar vol_syb[] = { L"\\DosDevices\\X:" };
  wd_ustr vol_name;
  wd_wchar c;
  if(name == NULL)
    return NULL;

  wd_ustr_init(&vol_name, name);

  // 遍历,逐个得到目标并比对字符串
  for(c = L'A'; c < (L'Z'+1); ++c)
  {
    wd_ustr_h my_target = NULL;
    vol_syb[12] = c;
    my_target = wd_symbolic_target(vol_syb);
    if( my_target != NULL &&
      wd_ustr_cmp(wd_ustr_h_ustr(my_target), &vol_name, wd_true) == 0 )
    {
      wd_ustr_h_free(my_target);
      break;
    }
    if(my_target != NULL)
    {
      wd_printf0("FF:%wZ\r\n",wd_ustr_h_ustr(my_target));
```

```
      wd_ustr_h_free(my_target);
    }
  }

  // 判断返回结果
  if(c == L'Z'+1)
    return NULL;
  else
    return wd_ustr_h_alloc(&vol_syb[12]);
}
```

得到盘符后就可以组合得到完整的路径.这个方式我测试过,无论动态加载或者静态加载启动的时候调用,都不会死机.

然后是在文件过滤其他 IRP 时,(如改名,查询,设置,读,写)的时候得到 FileObject 所对应的文件路径.

在文件读写的时候往往没有好的办法可以得到文件路径.对 FileObject 进行 ObQueryNameString 很容易导致死机.这似乎是微软自己留下的问题.但是如果你直接向下层设备发 IRP 进行查询,就不会死机.发送 QueryIRP 的代码比较简单,可以在网上找到.但是 IRP 的构建依赖于非文档的方法,总是不那么可靠,未来难保在兼容性上不出现问题.我希望不用非文档的方法.

此时如果你读 FileObject->FileName,会发现这个名字一般都依然存在.你可以用它作为文件路径.不过这依然有不少问题.首先 FileObject->FileName 只是为了生成这个文件而填写的请求路径.既然这个文件已经生成,那么这个路径就是可以丢弃的了(它存在并不表示不可能被丢弃).其次这个文件路径里面可能含有短名(例如 mydire~1),这样的路径和你想要得全路径不同,有可能导致跳过你的安全过滤.而且其中不含有盘符.如果需要得到盘符,可能还需要 Query,这又是一个常见的死机原因.

我用了一个可能不是很有效率的办法,既然我在 Create IRP 处理结束后已经得到了我要的长路径,那么我可以把它存在一个表中.当 FileObject 被 CleanUp 的时候,我清除这个表项以避免内存泄漏.然后再无论是 Read,Write,Query,Set 或者是 Rename(Set 的一种)的时候,我都可以通过这个表来查询我的路径.

应该用 Map 增加效率.Map 是数据结构问题。请诸位读者自己实现了。方法简述如下:

1. SfCreate 中，获得 FileObject 的文件路径(用前面的方法),并把 FileObject 指针和路径的对应关系，保存在一个 Map 中。
2. 在任何时候都可以在表中查询一个 FileObject 对应的路径.不必担心重入和中断级等等问题。
3. 在 SfCleanUp 中删去该 FileObject 对应的节点。

最后一个问题是如何在 Create IRP 处理之前得到路径名.

可能唯一的途径是通过 FileObject->FileName.要注意 FileObject->ReleatedObject 不为空的情况.这个时候 FileObject->FileName 是 RelatedObject 的相对路径.首先要 ObQueryNameString 这个对象.得到路径之后再和 FileObject->FileName 组合.然后是其中可能含有的短名转换为长名的问题.我没有找到简易的方法.我的同事 CardMagic 提供了一个非常麻烦但是确实有效的办法.其思想是:

首先你假设你得到一个路径 \aaaaaa~1\bbbbbb~1\cccccc~1\dddddd~1.txt.然后你把它分解成:

\
aaaaaa~1
bbbbbb~1
cccccc~1
dddddd~1.txt

以上 5 个对象.首先打开用 ZwCreateFile 打开第一个目录.第一个目录总是"\",这不可能是短名.然后调用 ZwQueryDirectoryFile 枚举下面所有的文件和目录.如果你用 FileIdBothDirectoryInformation 进行查询.那么会得到一组 FILE_ID_BOTH_DIR_INFORMATION,代表下面每个文件和目录:

```
typedef struct _FILE_ID_BOTH_DIR_INFORMATION {
```

```
  ULONG NextEntryOffset;
  ULONG FileIndex;
  LARGE_INTEGER CreationTime;
  LARGE_INTEGER LastAccessTime;
  LARGE_INTEGER LastWriteTime;
  LARGE_INTEGER ChangeTime;
  LARGE_INTEGER EndOfFile;
  LARGE_INTEGER AllocationSize;
  ULONG FileAttributes;
  ULONG FileNameLength;
  ULONG EaSize;
  CCHAR ShortNameLength;
  WCHAR ShortName[12];        // 这里有短名
  LARGE_INTEGER FileId;
  WCHAR FileName[1];        // 这里有长名
} FILE_ID_BOTH_DIR_INFORMATION, *PFILE_ID_BOTH_DIR_INFORMATION;
```

长短名都到手了,那么我们当然可以找到"\"之下的第一个"aaaaaa~1"所对应的长名了.然后依次类推,逐个查询.这真是个麻烦的办法,但是确实有效.

有时我认为可以直接打开 \aaaaaa~1\bbbbbb~1\cccccc~1\dddddd~1.txt 或者 \aaaaaa~1\bbbbbb~1\cccccc~1\来 QueryNameString,但是 CardMagic 说,网上有人说这样做依然不可靠,可能得到短名.

## 14 避免重入

在文件过滤驱动中进行文件操作,重入是最严重的麻烦根源之一.首先了解一下什么是重入.如果你调用一个函数,这次这个函数执行过程中，有必要再次调用这个函数，这就是重入.递归是一种常见的重入情况:

```
NTSTATUS SfCreate(...)
{
    SfCreate(...);
}
```

以上就是一个死递归.合理的递归是必须有终点的.死的递归(包括过深的递归)会导致 windows 内核调用栈溢出，系统崩溃出现蓝屏.

不过一般都不会直接在驱动中写以上的代码.真实的情况是这样的:

```
NTSTATUS SfCreate(...)
{
    ...
    ZwCreateFile(...)  // <- 这里会导致发出 Irp,并再次被我们过滤到，等于这里再次调用 SfCreate(...)
    ...
}
```

但重入绝对不是问题本身.你可以自由的利用重入实现你的功能，但是你必须避免死递归.如果我能判断这个请求是我自己发出的，我则跳过，这时重入虽然发生,但是对我并没有影响:

```
NTSTATUS SfCreate(...)
```

```
{
    ...
    if(这个 IRP 不由我的驱动自己发出)
    {
        ZwCreateFile(...)  // <- 这里会导致发出 Irp,并再次被我们过滤到，等于这里再次调用 SfCreate(...)
    }
    ...
}
```

理论上考虑,既然我们是文件过滤驱动，那么我们打开文件的时候，就没有理由再经过设备的顶层了，应该直接往我们的下层发送请求.这个功能是可以实现的，就是调用 **IoCreateFileSpecifyDeviceObjectHint** 来打开文件，而不要用 **ZwCreateFile.**

**IoCreateFileSpecifyDeviceObjectHint** 可以直接指定设备对象。你就直接指定下层设备就可以了.这样上面的设备根本收不到 **IRP**,打开文件的重入现象也不会再发生。

但是 **IoCreateFileSpecifyDeviceObjectHint** 这个函数在普通版本的 **2000** 下没有.**xp** 和 **2000+SP4** 和以上都有这个函数.如果要兼容所有的 **2000** 版本，你必须另想办法.

使用影设备**(Shadow Device)**是网络上广为流传的办法.是非常优秀的解决方案.我这里再介绍一个更方便使用的，我经常用的原理简单的”土办法”:

回到上面的问题,主要的困难是

**if(**这个 **IRP** 不由我的驱动自己发出**)**

这个 **if** 如何实现呢**?**

我可以在 **ZwCreateFile** 中传入特殊的参数.但是这些参数也可能为其他的过滤驱动或者根本就被普通用户所用.没有完全保险的办法，让我得到一个 **IRP** 的时候,得知这个操作是我的驱动自己发出的，自己不要再过滤它.

如果是一个应用程序,那么很简单，我只要判断一下当前进程,就知道这个请求是不是自己发出的**(**自己的应用程序当然知道自己的进程号**)**.而驱动是没有自己进程的。这样就有办法了，驱动虽然没有进程,但是我们可以自己生成一个线程.当我们有什么操作要进行又不想死递归的时候，就把这个操作放到线程中去做.这个线程号我们是知道的。以后任何 **IRP** 来到的时候，我们检查一下当前线程,如果是我们自己的线程,就跳过去,这样就避免了死递归的可能。

```
// 这是一个函数类型,这中间你可以做任何事情
typedef void ( *PWIT_DO)(void *context);
// 这是一个数据结构,表示一个线程
typedef struct WIT_THTREAD_
{
    LIST_ENTRY list;            //  Request list in our thread.
    HANDLE tid;                     //  Thread id.
    KEVENT event;               // An event to inform the thread to process a request.
    KSPIN_LOCK lock;            // A lock used by the list.
} WIT_THREAD,*PWIT_THREAD;
// 一个任务节点。我把我要做的一个任务,写入这个结构中.
typedef struct WIT_NODE_
{
    LIST_ENTRY list;
    void *context;
    KEVENT event;
    PWIT_DO do_somthing;
}WIT_NODE,*PWIT_NODE;
```

然后我自己来生成一个线程,代码如下:

```
PWIT_THREAD WITCreateThread(OUT NTSTATUS *status)
{
    PWIT_THREAD my_thread;
    my_thread = ExAllocatePoolWithTag(NonPagedPool,sizeof(WIT_NODE),WIT_TAG);
    if(my_thread == NULL)
    {
        *status = STATUS_INSUFFICIENT_RESOURCES;
        return NULL;
    }
    InitializeListHead(&my_thread->list);
    KeInitializeSpinLock(&my_thread->lock);
    KeInitializeEvent(&my_thread->event,SynchronizationEvent,FALSE);
    *status = PsCreateSystemThread(
        &my_thread->tid,
        (ACCESS_MASK) 0L,
        NULL,
        NULL,
        NULL,
        WITThreadProc,
        (PVOID)my_thread);
    if(!NT_SUCCESS(*status))
    {
        ExFreePool(my_thread);
        return NULL;
    }
    return my_thread;
}
```

这里的线程是一个死循环,寻找有没有要完成的任务.如果有，就完成它.

```
void WITThreadProc(IN PVOID context)
{
    PWIT_THREAD mythread = (PWIT_THREAD)context;
    PWIT_NODE node = NULL;
    for (;;)
    {
        // 等待有任务发生
        KeWaitForSingleObject(
            &mythread->event,
            Executive,
            KernelMode,
            FALSE,NULL);
        // 有就完成
        while ( node = (PWIT_NODE)ExInterlockedRemoveHeadList(
            &mythread->list,
            &mythread->lock))
```

```
            {
                node->do_somthing(node->context);
                KeSetEvent(&node->event,IO_NO_INCREMENT,FALSE);
            }
        }
    }
```

然后是我们如何插入任务的问题,代码如下:

```
    NTSTATUS WITDoItInThread(IN PWIT_THREAD thread,IN OUT PVOID context,IN PWIT_DO
do_somthing)
    {
        PWIT_NODE node;
        node = ExAllocatePoolWithTag(NonPagedPool,sizeof(WIT_NODE),WIT_TAG);
        if(node == NULL)
            return STATUS_INSUFFICIENT_RESOURCES;
        node->context = context;
        node->do_somthing = do_somthing;
        KeInitializeEvent(&node->event,SynchronizationEvent,FALSE);
        ExInterlockedInsertTailList(
            &thread->list,
            &node->list,
            &thread->lock);
        KeSetEvent(
            &thread->event,
            (KPRIORITY) 0,
            FALSE);
       KeWaitForSingleObject(
           &node->event,Executive, KernelMode,FALSE, NULL);
       ExFreePool(node);
        return STATUS_SUCCESS;
    }
```

**WITDoItInThread** 的调用非常容易。只要把需要完成的操作放在 **do_somthing** 里,参数与返回值放在 **context** 中,直接调用这个函数，这个函数就会把任务插入我们的工作线程队列,并等待完成后再返回.

现在你可以在这个线程里做任何事情，包括生成文件,读写和其他操作等.得到 **IRP** 的时候，可以通过线程 **id** 来判断是否我们自己的线程,来跳过重入问题:

```
    BOOLEAN WITIsMyThread(IN PWIT_THREAD thread)
    {
        return (PsGetCurrentThreadId() == thread->tid);
    }
```

## 14 结语与展望

这并不是一本关于文件过滤的技术大全.因此,我忽略了很多重要的主题.比如各种目录控制和文件属性查询与设置，**Cache** 管理器,网络文件系统,文件系统和存储设备的关系等等。但是相信读者已经有能力自己去研究他们了。

非常感谢您阅读此书.此外,有以下的额外事项需要声明:

我们学习了微软的 DDK 使用的技术.我们将利用他,但不是终身为它服务.

您将要为 Windows 开发驱动程序.我们的目标是:让更多的用户使用我们的软件.享受到我们的努力成果.而不是为了扩充 Windows 的功能，为微软打零工.

不要觉得你的代码只需要为 Windows 能使用就足够了。如果你的代码中到处都用到 DDK 中定义的结构和调用，比如使用 DEVICE_OBJECT 并用直接用 ExAllocatePool 分配内存，你的项目可能进展比较快,但是后悔也来得快得多。

把 DDK 当作工具使用吧.但不要让他限制住你. 微软的技术是有效的，但并不是优秀的。

Sfilter 对微软来说是比较古老的架构.之后发布了新的微端口文件过滤驱动的架构.把你的代码大量的插入 Sfilter 中，并融为一体是不智的.很快就会带来移植方面的困难.

新的微端口文件过滤驱动的应用还不多.微软总是喜欢发布更复杂的接口,以便掩盖内部.但是它的方向不一定是对的.新发布的架构被抛弃的情况也不鲜见.后面附带一篇关于微端口文件过滤驱动的翻译文章,便于有兴趣的读者了解.

**14.5** 附:微端口文件过滤驱动

Windows文件系统过滤管理器之

# 微过滤器驱动开发指南

## 0. 译者序

本文翻译仅仅用做交流学习。我不打算保留任何版权或者承担任何责任。不要引用到赢利出版物中给您带来版权官司。本文的翻译者是楚狂人，如果有任何问题，你可以通过邮箱MFC_Tan_Wen@163.com,或者是QQ16191935,或者是MSN walled_river@hotmail.com与我交流。

我翻译此文出于对文件系统技术的兴趣。这就是新的文件系统过滤接口。其实也不算什么新的东西，微软开发了另一个“旧模型的”过滤驱动，称之为过滤管理器（Filter Manager）。从而提供了一系列新的接口来让你开发新的过滤器。确实这套接口变简单清晰了。你至少避免了包含无数个信息的IRP,避免了请求在各个部件中循环的发来发去，一个分发例程中处理无数中情况，一不小心系统崩溃。我不知道花了多少时间才弄明白一个简单的缓冲读请求从用户到过滤到文件系统和缓冲管理器，虚拟内存管理器之间的关系！现在你也许不需要再管他们了，仅仅做好自己的过滤工作就可以。

此文的原文是《Filter Driver Development Guide》，出自微软的网站。我在以下这个地址下载得到此文:

http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/FilterDriverDeveloperGuide.doc
我尽量在翻译中使文章保持原貌。如果您认为此文无法理解，建议您首先阅读旧的文件过滤驱动的相关资料。我认为必须有文件系统和windows驱动的相关知识，才能阅读此文。

我未必总是使用规范的名词，但我总是使用最容易理解的名词。一些常用的可能不翻译，比如IRP,MDL,有驱动开发经验的人应该可以理解。另一些可能采用中文（英文）的方式。

一些解释如下：
例程（Routine）：我不懂得例程和函数有什么不同。我认为例程就是函数。称为Routine而不是Function可能是为了避免其他c程序员理解得太容易。
接口（Api）：编程开发接口，一个提供给你调用的函数。
流（Stream）：NTFS文件系统独有.用来保存一个文件的额外信息.似乎可以当作文件被打开。
域（Field）：一个数据结构中的一个数据成员。喜欢数据库的人可能称为字段。喜欢面向对象的称为数据成员。
透明（transparence）：看不见，意味着也不需要管。不过请注意透明的反意词绝对不是不透明（opacity）。
不透明（opacity）：不知道的。比如空指针。因为空指针指向的是什么，从空指针本身是了解不到的。所以称为不透明的指针。
回调（Callback）函数：一个由系统调用而且原则上你不能自己调的函数。
预操作（pre-operation）回调：如果打算过滤一个操作，那么这个回调出现在操作完成之前。
后操作（post-operation）回调：如果打算过滤一个操作，那么这个回调出现在操作完成之后。

# 1.概述

这个文档用于I/O管理器和基本文件系统之间的过滤驱动。文件系统可能是本地或者网络的。这个文档不涉及文件系统和存储设备之间的过滤驱动，比如FtDisk和DMIO.

我们将主要讨论一种新的文件系统过滤驱动模型，所谓的微过滤器（minfiter）.

以前的文件系统过滤基于一个例子sfilter.使用IRP和设备对象进行过滤。我们现在称之为“旧过滤模型”

新的架构中一个关键的组件其实是一个旧过滤模型的文件系统过滤驱动，被称为“过滤管理器（Filter Manger）”.在未来，微软发行的操作系统将默认安装这个驱动。（译者注:现在，你得手工安装。）这个驱动通过提供一些库供微过滤器调用来管理所有的微过滤器。必要的头文件，库和二进制代码都在微过滤器IFSKit中。

为何要开发一个微文件系统过滤驱动?
.通过更少的工作量，得到更简单的，更可靠的过滤驱动.
.动态加载和卸载,绑定和解除绑定.
.在过滤栈中，绑定到一个合理确定的位置。
.上下文管理。快捷,干净,可靠的上下文管理，用于文件对象,流,文件，实例和卷.
.一组有用的调用.包括根据文件名寻找，高效存取,和用户态程序之间的通信,以及io排队.
.支持非回环I/O.这样，一个微过滤器发起的I/O请求可以轻松的只让栈中更下面的微过滤器以及文件系统看到了。
.仅仅过滤感兴趣的操作。不象旧过滤模型那样必须挂接每个操作入口以便把操作传递到下层.

# 2.术语

在过滤管理器架构中，定义了一些新的对象。为了搞清这些，这里将列出一些定义:

过滤器:在文件系统上执行一些过滤操作的一种驱动.

卷:在本地文件系统，这个对象指文件系统所管理的逻辑卷.对于网络重定向文件系统，指所有网络请求被重定向的目的。卷直接对应文件系统(无论本地或者网络)旧过滤模型中的设备对象(DEVICE_OBJECT).

实例:一个过滤器在一个卷上唯一的某层上生成的一个实例化对象。过滤管理器把所有的IO请求发到卷上的实例栈上。一个微过滤器在一个卷上可能不止一个实例.规范的例子是FileSpy.有时候把FileSpy的两个实例分别绑定在另一个过滤器的上边和下边。此时每个实例有一个私有的上下文.这个上下文包含IO操作的日志.可以用来比较一个过滤器上下的IO操作有什么不同。

文件:文件系统保存在一个磁盘上的可能包含若干个流的有名字的数据对象.
流:指一个文件中的物理数据流.

文件对象(FileObject):用来描述一个用户对一个文件中的一个物理数据流的一次打开。

回调数据(CallbackData):过滤管理器中的一种数据结构,包含了一个操作中的所有信息。对应于旧过滤模型中的IRP.

## 3.微过滤器安装

微过滤器可以通过一个INF文件安装。INF文件指出了这个微过滤器所支持的实例.实例的具体说明在第5节.每个实例有一组标志,还有一个唯一的数值固定了它在过滤栈中的位置。

INF文件中有一个表标明了每个实例的层级.这用来给文件系统过滤的开发商装载他们的微过滤器.有标记标明了这个微过滤器是否需要"自动的绑定".如果是，那么每个新的卷出现的时候，微过滤器都回收到一个通知.它可以在此绑定他们.绑定的时候，inf文件中的层级决定了绑定到什么层次上。

在微过滤器运行时,文件系统过滤开发商也可以在某个指定的层级上动态的生成一个实例,这可以使用FilterAttachAtAltitude()调用.这对于开发者来说可以用来进行测试和排除bug.

## 4.微过滤器注册

微过滤器是内核驱动。因此它必须导出一个名为DriverEntry的函数。在驱动加载的时候这个函数第一个被调用.很多微过滤器在DriverEntry()中调用FltRegisterFilter().

FilterRegisterFilter()需要传入一个参数。是一个FLT_REGISTRATION结构.包含了:一个卸载例程.实例通知回调,一组上下

文回调指针,一组文件系统操作回调指针.一般情况下,微过滤过器只捕获一部分操作，因此文件系统操作回调指针可能并不多。

对于某一种操作，微过滤器可以指定一些附加的标记来指明它是否在所有的情况下都收到它们.比如，如果FLTFL_OPERATION_REGISTRATION_SKIP_PAGING_IO被指定了，微过滤器就不会收到任何此类IRP的paging I/O操作。

同样的，如果FLTFL_OPERATION_REGISTRATION_SKIP_CACHED_IO被指定了，那么只有这类操作的非缓冲请求能看见.(比如说，指定了IRP_MJ_READ类操作，那么所有的缓冲读就都不会被微过滤器捕获了。

# 5.开始过滤

当一个微过滤器注册自己，它就应该在某个时间调用函数FltStartFiltering()来开始过滤。并不一定要在DriverEntry中调用。不过大多数微过滤器可能是这样做的。

这个函数将激发必要的通知，导致微过滤器绑定到卷上然后开始过滤I/O操作。为此，过滤管理器会通过微过滤器的inf文件遍历它注册过的所有的实例。

每个示例都有一个层级。一个层级是一个唯一的字符串，(如"100.123456"),这个定义了微过滤器的这个实例在栈上的位置。商业版本的微过滤器层级将由微软公司来分配。

层级的数字越高，这个微过滤器绑定在栈上的位置就越高。一些示例层级提供给开发者用来实现微过滤器。这些是仅有的不会被分配的层级。层级有两个作用:一是确定两个微过滤器之间的顺序关系，尤其是有时得实现一些不用去考虑个别微过滤器什么时候加载的功能。

比如说，一个加密解密过滤器必须安装在一个防病毒的过滤的下边。否则，防病毒过滤器无法从已经加密的内容中发现病毒。另外就是提供了一个最小的测试矩阵,用来测试这些过滤驱动的互容性。如果这些驱动实例都是按一个指定的顺序在栈中的，那么测试的时候就不用再考虑排列各种不同的顺序了。

在inf文件中，一个实例和一个标记联系在一起。如果第1个位标记了，那么微过滤器不会在卷出现在系统中的时候得到通知。这样的实例应该通过过滤管理器编程接口来手工的绑定。如果第2位被设置了，即使手工的发送了一个绑定请求，微过滤器也不会收到通知来要求绑定一个实例到一个卷。

# 6.实例的通知

当一个实例生成的时候，一组回调函数提供来通知微过滤器。通过这些回调，微过滤器可以决定它的实例在什么时候绑定到卷上和从卷上解除了绑定。

## 6.1. 安装一个实例

回调例程InstatanceSetupCallback()在下列情况下被调用:
.当一个微过滤器加载的时候，每个存在的卷都会导致这个调用。
.当一个新的卷被mount.
.当FltAttachVolume被调用（内核模式）
.当FltAttachVolumeAtAltitude()被调用（内核模式）
.当FilterAttach()被调用（用户模式）
.当FilterAttachAtAltitude()被调用（用户模式）

在这个过程中，微过滤器决定是否在这个卷上生成实例。这个回调的原型如下:

```
typedef NTSTATUS
(*PFLT_INSTANCE_SETUP_CALLBACK) (
    IN PCFLT_RELATED_OBJECTS FltObjects,
    IN FLT_INSTANCE_SETUP_FLAGS Flags,
    IN DEVICE_TYPE VolumeDeviceType,
    IN FLT_FILESYSTEM_TYPE VolumeFilesystemType
    );
```

FltObjects结构喊有指向微过滤器，卷，和实例的指针。这个实例指将要在InstanceSetupCallback()函数中生成的实例。Flags标记是什么操作导致激发了InstanceSetupCallback():

**FLTFL_INSTANCE_SETUP_AUTOMATIC_ATTACHMENT:** 这是一个微过滤器注册的时候，一个自动的绑定通知。过滤管理器为每个刚加载的微过滤器枚举所有的卷。如果是一个使用者明确的指定一个实例绑定到某一个卷，不会设置有这个标记。

**FLTFL_INSTANCE_SETUP_MANUAL_ATTACHMENT:** 通过调用FilterAttach()(用户态)，或者是FilterAttachVolumeAtAltitude()(用户态)，或者是FltAttachVolume()(内核态)所发起的一个手工的请求。

**FLTFL_INSTANCE_SETUP_NEWLY_MOUNTED_VOLUME:** 文件系统刚刚挂载(mount)了一个卷，所以呼叫InstanceSetupCallback()来通知微过滤器，如果它愿意可以生成实例来绑定这个卷。

在InstanceSetupCallback()中，微过滤器同时得到了卷设备类型（VolumeDeviceType）和卷文件系统类型（VolumeFilesytemType）,用以判断这个卷是否过滤器所感兴趣的。同时,微过滤器可以调用FltGetVolumeProperties()来获取卷属性。通过FltSetInstanceContext()在实例上设置上下文。当然这是需要绑定的时候。它甚至可以在卷上打开或者关闭文件。

如果这个回调返回了成功，那么这个实例将绑定到卷上。如果返回了一个警告或者错误，那么不会绑定。
如果微过滤器没有指定InstanceSetup回调，那么，系统将认为用户总是返回了STATUS_SUCCESS,实例总是会生成并绑定。

## 6.2. 控制实例的销毁

InstanceQueryTeardown()回调仅仅在一个手工解除绑定的请求下被调用。以下操作可能导致:
FltDetachVolume() (内核模式)
FilterDetach()　(用户模式)

如果一个微过滤器没有提供这个回调，那么手工解除绑定是不被支持的。但是，卷的解挂载（dismount）和微过滤器的卸载还是允许的。

如果这个回调返回成功，那么过滤管理器开始销毁给出的实例。最后实例的InstanceTeardownStart()和InstanceTeardownComplete()会被调用。如果返回了错误或者警告，手工解除绑定会失败。推荐的错误代码有: **STATUS_FLT_DO_NOT_DETACH**,不过实际上你可以返回任何错误代码。
**InstanceQueryTeardown()**回调的原型是:

```
typedef NTSTATUS
```

```
(*PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK) (
    IN PCFLT_RELATED_OBJECTS FltObjects,
    IN FLT_INSTANCE_QUERY_TEARDOWN_FLAGS Flags
    );
```

和InstanceSetupCallback()类似，FltObject指出了与这个销毁操作有关的微过滤器，卷和实例。

## 6.3. 实例解绑定的同步

如果InstanceTeardownStart()的时候已经决定要解除绑定，那么这个例程中必须做以下的事情：

(1)重设所有的未决的I/O操作（包括预操作和后操作）

(2)保证不会有新的I/O操作进入未决。

(3)对刚刚到达的操作开始最少的工作。

同时，应该做以下操作：

(1)关闭所有打开的文件。

(2)取消所有本过滤器发起的I/O请求。

(3)停止将新的工作任务排队。

然后微过滤器把控制权交还过滤管理器来继续它的销毁过程。当所有与这个实例相关的操作都排除干净或者完成了，InstanceTeardownComplete()会被调用。管理器保证此时所有此实例的存在的操作回调都完成了。这时微过滤器必须关闭所有这个实例打开的文件。

这两个回调的原型如下：

```
typedef VOID
(*PFLT_INSTANCE_TEARDOWN_CALLBACK) (
    IN PCFLT_RELATED_OBJECTS FltObjects,
IN FLT_INSTANCE_TEARDOWN_FLAGS Reason
);
```

FltObjets中有微过滤器，卷和实例。Resson参数指明这次销毁的原因，可能是以下一些标记的组合：

FLTFL_INSTANCE_TEARDOWN_MANUAL: 这次销毁操作是一个手工的请求。(FilterDetach()或者 FltDetachVolume()).

FLTFL_INSTANCE_TEARDOWN_FILTER_UNLOAD: 这次销毁操作是因为微过滤器执行卸载或者是选择了把卸载请求失败掉导致的。

FLTFL_INSTANCE_TEARDOWN_MANDATORY_FILTER_UNLOAD: 这次销毁操作是一次强制卸载导致的。这种情况下不能把卸载请求失败掉。

FLTFL_INSTANCE_TEARDOWN_VOLUME_DISMOUNT: 这次销毁是一个卷被解挂载的结果。FLTFL_INSTANCE_TEARDOWN_INTERNAL_ERROR: 这次销毁是因为安装实例的时候的一个内部错误导致的，比如内存不足。

请注意没有返回值。InstanceTeardownStart()和InstanceTeardownComplete()都不能失败。过滤管理器保证这些例程都运行在Passive IRQL.

# 7. 回调支持

## 7.1 回调数据（Callback data）

回调数据（Callback data）是过滤管理器用来描述I/O操作的新结构。类似旧过滤模型下的IRP.微过滤器通过这个结构和过滤管理器交互。不同的是，回调数据不像IRP那样管理一个栈结构。回调数据的管理都通过已经明确定义的过滤管理器接口。并且返回状态值给过滤管理器即可。

FLT_CALLBACK_DATA类型包含了微过滤器描述一个I/O操作所需要的所有的信息。下面继续详细讲解这个结构中的各个域来说明其中包含的信息：

Flags:提供这个操作的一些信息。一个或多个下面的标记可能被设置在Flags中:

FLTFL_CALLBACK_DATA_IRP_OPERATION: 这个回调数据描述一个IRP操作。

FLTFL_CALLBACK_DATA_FAST_IO_OPERATION:这个回调数据描述一个FastIO操作。

FLTFL_CALLBACK_DATA_FS_FILTER_OPERATION:这个回调描述一个文件系统过滤器操作。

FLTFL_CALLBACK_DATA_SYSTEM_BUFFER:这个操作所用的缓冲是一个系统分配的缓冲。

FLTFL_CALLBACK_DATA_GENERATED_IO:这个操作是由一个微过滤器发起的。

FLTFL_CALLBACK_DATA_REISSUED_IO:这个操作被一个当前实例之上的过滤器所重新发回给文件系统。

FLTFL_CALLBACK_DATA_DRAINING_IO:只有设置了后操作（Post-operation）回调的情况下，表明这是一个快速“排出”的I/O操作以便微过滤器的卸载。

FLTFL_CALLBACK_DATA_POST_OPERATION:只有设置了后操作（Post-operation）回调的情况下，表明着个I/O正在后操作中。

FLTFL_CALLBACK_DATA_DIRTY:当一个微过滤器已经改变了这个操作的一个或者多个可变参数的时候，设置这个参数。这个标记仅仅在Pre-operation过程中设置。微过滤器必须用FLT_SET_CALLBACK_DATA_DIRTY()和FLT_CLEAR_CALLBACK_DATA_DIRTY()来操作这个标记。

Thread: 发出这个操作的线程的地址。

Iopb:  指向这个操作的可变参数的指针。这个结构在后边详叙。

IoStatus:IO_STATUS_BLOCK结构返回操作最后的状态。如果一个微过滤器打算结束这个操作，那么必须先设置这个域，然后才能结束这个请求。对于传递给文件系统去的请求，在后操作过程（Post-operation）中有操作最终的状态。

TagData:仅仅在Create操作的后操作回调中有效。当一个操作的目标文件有一个重解析点（Reparse point）的时候设置这个位。

QueueLinks:一个链表入口结构。有时要把回调数据（Callback Data）放入工作队列中使用这个。

QueueContext[2]:一组空指针结构，用来传入附加的上下文到工作队列处理过程中。

FilterContext[4]:一组空指针结构，当回调数据进入了队列，微过滤器可以做任意使用。不依赖于过滤管理器的内部结构。

RequestorMode:这个操作的者的请求模式。

Iopb域所指的是一个FLT_IO_PARAMETER_BLOCK结构。包含了回调数据中可以修改的部分。对比IRP来说，这里相当于IRP的当前栈空间（current stack location）。微过滤器必须访问这个结构来得到每次预操作（pre-operation）和后操作（post-operation）回调的I/O参数。下面是一些更详细的细节:

IrpFlags:IRP中描述这个操作的一些标记。

MajorFunction:IRP主功能号。

MinorFunction:IRP辅功能号。

OperationFlags:即IO_STACK_LOCATION.Flags.

TargetFileObject:这个操作所影响到的目标文件。

TargetInstance:管理这个操作的实例。

Parameters:FLT_PARAMETERS是一个共用体。描述主功能号和辅功能号所指定的操作的具体参数。

除了在预操作回调中不能修改主功能号之外，微过滤器可以修改这个结构中其他的任何参数。如果参数改变，微过滤器应该调用FLT_SET_CALLBACK_DIRTY()来注明这个改变。更多详细的信息将在第8节中讲述。

微过滤器在同一个I/O操作的预回调和后回调中，总是会看到参数是一样的。即使下面的过滤器可能已经修改了这些参数。这是由过滤管理器保证的。但是虽然FLT_IO_PARAMTER_BLOCK的内容是一样的，在预操作和后操作中，这个结构的地址可能不一样。因此微过滤器不应该依赖这个地址。

回调数据结构包含IO_STATUS_BLOCK来记录这个操作的状态。过滤管理器会“尊重”这些改变而不会标记这些数据为脏（Dirty）。微过滤器如果打算在预操作回调中结束这个操作或者是后操作回调中撤消这个操作，都必须先设置这个IO_STATUS_BLOCK。

## 7.2 预操作回调（Pre-Operation Callbacks）

所有的预操作回调原型都是这样:

```
typedef FLT_PREOP_CALLBACK_STATUS
(*PFLT_PRE_OPERATION_CALLBACK) (
    IN OUT PFLT_CALLBACK_DATA Data,
    IN PCFLT_RELATED_OBJECTS FltObjects,
    OUT PVOID *CompletionContext
    );
```

所有的预操作回调都返回一个FLT_PRE_OPERATION_CALLBACK_STATUS.这个变量是如下定义的:

FLT_PREOP_SUCCESS_WITH_CALLBACK:这个操作成功了而且微过滤器需要后操作回调。

FLT_PREOP_SUCCESS_NO_CALLBACK:  操作成功了，但是不需要后操作回调。

FLT_PREOP_PENDING:  微过滤器将在未来某个时候结束这个操作（通过调用FltCompletePendedOperation（））。微过滤器在返回这个值之前不需要做其他特殊的操作（比如IoMarkIrpPending（））。如果这个状态返回了，这个I/O操作被过滤管理器挂起（栈中的下层驱动都不会收到预操作回调），直到FltCompletePendedPreOperation（）被调用。

FLT_PREOP_COMPLETE:  微过滤结束了操作。这个微过滤器设置了Data->IoStatus.Status中的I/O状态。这个过滤器以下的微过滤器，旧模型过滤器和文件系统都不会看见这个I/O请求。而之上的微过滤器回看到这个请求以合适的状态完成。对于CLEANUP和CLOSE操作来说，微过滤器以一个失败状态结束这个操作是不允许的。因为这些操作不能失败。

FLT_PREOP_SYNCHRONIZE:  仅仅在非CREATE操作有效。(CREATE操作是自动同步的)。若返回此值微过滤器必须有一个后操作回调。这表明这个微过滤器希望这个操作在同一个线程里完成。也就是说后操作调出现的时候和预操作调用在同一个线程上下文里。这是由过滤管理器所保证的。不管下层的过滤器以及文件系统是挂起还是忽略这个I/O操作。这个状态必须小心使用。因为过滤管理器必须同步整个I/O,这可能影响整个系统的性能。

FLT_PREOP_DISALLOW_FAST_IO: 这个状态仅仅在旧模型下返回BOOLEAN的fastI/O的操作的情况下有效。这个状态表明不接受fastI/O请求，请发送IRP重试。

在预操作返回之前，过滤器可能修改I/O操作的参数。而且能修改的参数都集中在Data->Iopb中。当一个微过滤器修改了任何一个参数，它必须调用FLT_SET_CALLBACK_DATA_DIRTY()，否则，修改不会被承认，可能导致未知的错误。

对此还有两个例外。如果修改的是IoStatus，没有必要设置Dirty就会被过滤管理器所承认。

另一个例外是IRP_MJ_CREATE的后操作过程。如果一个碰到重解析点（reparse point)，Data->TagData会指想一个重解析数据缓冲。如果微过滤器打算修改这个缓冲，它可以释放了这个缓冲然后重新分配一个（不能为空)。此时不用调用FLT_SET_CALLBACK_DATA_DIRTY().

## 7.3 后操作回调（Post-Operation Callbacks）

所有的后操作都有同样的原型:

```
typedef FLT_POSTOP_CALLBACK_STATUS
(*PFLT_POST_OPERATION_CALLBACK) (
    IN OUT PFLT_CALLBACK_DATA Data,
    IN PCFLT_RELATED_OBJECTS FltObjects,
    IN PVOID CompletionContext,
    IN FLT_POST_OPERATION_FLAGS Flags
    );
```

**Data->Flags中都设置了FLTFL_CALL_DATA_POST_OPERATION.**

如果一个微过滤器在预操作回调中返回**FLT_PREOP_SUCCESS_WITH_CALLBACK**了，它肯定能收到这个同样操作的完成回调也就是后操作回调。后操作回调返回的类型为**FLT_POSTOP_CALLBACK_STATUS**.这个值是:

**FLT_POSTOP_FINISHED_PROCESSING** - 微过滤器已经完成了这个操作控制权应该还给请求的发起者.

FLT_POSTOP_STATUS_MORE_PROCESSING_REQUIRED – 微过滤器还没有完成这个请求，而且回在后边完成它，使用FltCompletePendedPostOperation().

与旧模型过滤驱动的完成例程不同，这个后操作回调执行在DPC中断级上。如果一个微过滤器需要完成一些在DPC上完成不了的工作，可以调用FltDoCompletionProcessingWhenSafe()。有必要的情况下（我们在DPC中断级时），会把工作插入一个工作线程。除非这个请求是不能被排队的（比如一个页面交换请求(paging I/O)）.

对于打算在后操作回调中取消一个文件打开的微过滤器，FltCancelFileOpen()调用可以对指定的FileObject来一个清理和关闭的功能。微过滤器必须必须填写合适的错误代码，并且在后操作回调中返回FLT_POSTOP_FINISHED_PROCESSING.

当一个实例被卸除的时候，过滤管理器可能调用候后操作回调，但是此时操作还未真的完成。这时，标志FLTFL_POST_OPERATION_DRAINING会设置。此时提供了尽量少的信息。所以微过滤器应该清理所有的从预操作中传来的操作上下文，并返回FLT_POSTOP_FINISHED_PROCESSING.

操作的IRP_MJ（主功能码）中增加了一些附加的新数值来表示FastI/O中的一些没有IRP与之对应的操作。目的是把IRP操作和FastI/O操作可以采用同样的处理方式。这样通过一些标记就可以区分IRP操作，微过滤相关操作和FastI/O操作，而不用注册一些类似的回调函数了（比如读，写，锁定等操作的回调函数）。

有以下这些操作主功能码：
所有原有的IRP_MJ。
IRP_MJ_FAST_IO_CHECK_IF_POSSIBLE
IRP_MJ_NETWORK_QUERY_OPEN
IRP_MJ_MDL_READ
IRP_MJ_MDL_READ_COMPLETE
IRP_MJ_PREPARE_MDL_WRITE
IRP_MJ_MDL_WRITE_COMPLETE
IRP_MJ_VOLUME_MOUNT
IRP_MJ_VOLUME_DISMOUNT
IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION
IRP_MJ_RELEASE_FOR_SECTION_SYNCHRONIZATION
IRP_MJ_ACQUIRE_FOR_MOD_WRITE
IRP_MJ_RELEASE_FOR_MOD_WRITE
IRP_MJ_ACQUIRE_FOR_CC_FLUSH
IRP_MJ_RELEASE_FOR_CC_FLUSH

关于主功能号的一些要注意的地方：
（1）IRP_MJ_CREATE的预操作回调中不能获得或者设置文件，流，或者流句柄这样的上下文。因为在预操作中，文件或者流还没有决定是否生成。
（2）IRP_MJ_CLOSE 的后操作中不能同样不能设置或者或者文件，流或者流句柄这样的上下问。因为在这个操作中，一些与之相关的系统内部结构可能已经释放了。
（3）IRP_MJ_CLEANUP和IRP_MJ_CLOSE 是永远不能失败的。他们可以挂起，传递或者成功结束。但是不能返回FLT_PREOP_COMPLETE并填一个错误码在IoStatus块中。

后操作回调不能失败，因为这个操作已经发生了。如果希望在后操作回调中让一个操作失败，那么你必须撤消已经成功的操作。比如IRP_MG_CREATE操作，过滤管理器提供了FltCancelFileOpen()来销毁已经打开的文件对象。但是过滤器还是有责任来重新整理由于Create操作而被覆盖掉的原来的文件的内容。

在后操作中对操作参数做的任何改变都不会被过滤管理器所承认。

# 8.操作回调数据参数

## 8.1 I/O参数块（I/O Parameter Block）

就像前面所提及的，回调数据（Callback Data）包含了两个关于I/O的重要的数据结构:
1）I/O状态块（I/O Status Block）：　Data->IoStatus用来返回它自己所结束的操作的状态（或者在后操作回调中可以读取下层驱动所完成的操作的状态）。
2）I/O参数块（I/O Parameter Block）：Data->Iopb指向这个I/O操作中专为本微过滤器使用的数据。
本节深入讨论读取和改变这些参数的问题。

Iopb中的主功能号和辅功能号表示IRP/FastIo/FsFilter－主／辅功能，主功能号不能由过滤器修改。过滤管理器不支持。
TargetFileObject表示I/O操作的目标，流的文件对象。微过滤器可以修改它（必须调用FLT_SET_CALLBACK_DATA_DIRTY（）),而且被过滤管理器承认。
TargetInstance参数与I/O从一个实例向另一个实例传递的意义不同，是表示当前的实例。过滤器可以改变这个指针。但是仅仅能是同一个层级，但绑定在其他的卷上的实例。同时，也必须调用FLT_SET_CALLBACK_DATA_DIRTY（）。

微过滤器是不能修改TargetInstance指向同一个卷上的另一个实例的。举个例子来说，一个微过滤器在一个卷C:上有两个实例。一个在层级２００称为实例C２００,，另一个在层级１００，称为实例C１００，同时在D:上还有一个层级２００的实例，称为实例D:２００。现在假设在实例C200上IRP_MJ_READ的预操作回调被调用了，微过滤器可以把TrageInstance修改为实例D:２００，但是不能修改为实例C:１００。

这组织了任何微过滤器同一个卷的栈中非法的（不按固定顺序的）传递操作。
I/O参数块中含有一些与具体操作相关的参数。微过滤器必须用与操作配套的共用体结构来访问他们。对于IOCTL和FSCTL,具体的控制码也有不同的共用体结构。微过滤器必须检查具体的控制码来使用正确的共用体。

## 8.2 使用缓冲/MDL

在IRP的世界中，缓冲采用多种机制传递到驱动中。比如支持基本I/O的设备对象，缓冲是通过Irp->UserBuffer传入的，这对文件系统是最常见的。有些设备只支持缓冲I/O,那么I/O管理器传入的缓冲区Irp->AssociatedIrp.SystemBuffer.对于支持直接I/O的设备，缓冲是通过Irp->MdlAddress,一个被锁定的MDL。

但是可能有例外。一些IRP的栈空间的缓冲参数直接传入。可能指向核心内存或者是原始的用户空间内存。和设备对象对I/O操作的要求无关。这些缓冲不能传到硬件，因为他们无视设备对象所支持的I/O类型。

也有一些IRP总是缓冲I/O,比如IRP_MJ_QUERY/SET_INFORMATION.

对于微过滤器，缓冲总是通过适当的操作相关的共用体传入。没有一个通用的方法可以获得缓冲/MDL的地址来源。这种设计是为了减少栈空间之间缓冲导致的冲突。

在回调数据中，如果操作是缓冲型的，那么标记 FLTFL_CALLBACK_DATA_SYSTEM_BUFFER会被设置。如果是这样的，那么缓冲区在非分页的核心内存中。

如果这个标记没有设置，那么只能说明不是缓冲型。微过滤器有方法做进一步区分（见下一节）。缓冲还是可能来源于某个核心内存池。但是只要这个标记没有设置，过滤器就应该假设内存在原始用户空间中。这种情况下，如果缓冲并不是传入的用户缓冲，那么总是需要一个MDL来锁定一些页面，而且调用者必须获得一个系统空间地址来访问这些页面。这是一条规则，将在下节详叙之。

最后，对于某些微过滤器希望能够定位缓冲／长度／MDL来做一些最通用的操作的时候，FltDecodeParameters()提供用来做一个快速的查找并返回一个指向参数结构中的缓冲／长度／MDL。对于没有缓冲空间的操作，返回STATUS_INVALID_PARMETER.

```
NTSTATUS
FLTAPI
FltDecodeParameters(
    IN PFLT_CALLBACK_DATA CallbackData,
    OUT PMDL **MdlAddressPointer OPTIONAL,
    OUT PVOID  **Buffer OPTIONAL,
    OUT PULONG *Length OPTIONAL,
    OUT LOCK_OPERATION *DesiredAcces OPTIONAL
    );
```

DesiredAccess表示微过滤器可以使用的访问缓冲的方式。比如对于IRP_MJ_READ,可能指定IpWriteAccess,意味着这个缓冲区是可写的。对于IRP_MJ_WRITE,一般指定IoReadAccess表示微过滤器可以读取这个缓冲区但是不能修改它。这样一个应用使用一个仅仅有只读权限的页面来发起一个写请求也是合法的了。

对于希望缓冲空间被锁定的过滤器，应该了解：如果FLTFL_CALLBACK_DATA_SYSTEM_BUFFER表示设置了，那么可以假设这个缓冲区已经被锁定，可以安全的访问。

如果没有设置则可以调用FltLockUserBuffer()来锁定页面。这个调用可以保证页面被合适的方法锁定。如果成功了，它会设置与操作相关的参数部分的MdlAddress域为用来描述这些页面的MDL.

## 8.3 交换缓冲

一些微过滤器为了某些操作必须交换缓冲。考虑一个微过滤器实现加密算法，对一个非缓冲（non-cached）IRP_MJ_READ,它一般会希望把缓冲中的数据解密。同样的在写的时候，它希望把内容加密。考虑以下情况：内容无法在这个空间中加密。因为对于IRP_MJ_WRITE,这个微过滤器可能只有IoreadAccess权限。

因此微过滤器必须以他自己的有读写权限的缓冲区取代原来的缓冲区。加密了原缓冲区中的内容后写入新缓冲区后，再继续传递I/O请求。

为此，过滤管理器支持缓冲转换。有以下一些游戏规则必须遵守:

1. 改变了缓冲区的微过滤器必须有对应的后操作回调。这样缓冲能被过滤管理器自动的转换回来。
2. 如果改变的是一个标记有FLTFL_CALLBACK_DATA_SYSTEM_BUFFER标记的缓冲，必须保证新的缓冲是非分页内存。（也就是比如来自非分页内存池或者锁定的内存）。
3. 如果以上的标记没有设置，那么微过滤器必须按设备对象的要求来确定缓冲类型（可以在卷属性中查看DeviceObjectFlags等标记）。比如如果是支持直接I/O的，那么必须提供MDL等等。
4. 如果微过滤器使用非分页池中的缓冲来给一个没有设置FLTFL_CALLBACK_DATA_SYSTEM_BUFFER的操作，那么它也必须用MuBuildMdlForNonpagedPool()并把地址填写到MdlAddress域中。这是因为这么一来，下面的任何过滤器或者文件系统都不用再尝试去锁定非分页池（可以在构建的时候使用断言，但是对效率不利），如果提供了一个MDL,过滤器和文件系统总是可以通过MDL访问缓

冲（可以获得一个系统内存地址来访问它）。

5. 替换一个缓冲的时候，微过滤器也必须换掉MDL（就是说缓冲和MDL要保持同步了）。对于通常的直接I/O异常，可以把MDL留空。
6. 微过滤器不应该释放旧的MDL和缓冲空间。
7. 不要尝试在后操作回调中替换掉旧的缓冲和MDL.过滤管理器自动执行这些操作。实际上微过滤器在后操作回调中的Iopb中见到缓冲空间和MDL是旧的（译者注:替换前的）。微过滤器必须自己在上下文中记录新的缓冲区。
8. 微过滤器应该释放自己分配的（和替换过的）缓冲。无论如何，如果有的话，过滤管理器会自动释放新缓冲的MDL。
9. 微过滤器不希望过滤管理器自动释放交换过的缓冲MDL可以调用FltRetainSwappedBufferMdl（）。
10. 过滤器如果希望访问交换过的缓冲的MDL可以在后操作回调中使用FltGetSwappedBufferMdl（）。既然一个更下层的过滤器或文件系统交换了新的缓冲空间进来，那么有可能生成了一个MDL。在后操作回调中微过滤器交换缓冲之前，过滤管理器保存了所有这样的的MDL。这个调用可以用来访问这些MDL.

# 9.上下文（Context）支持

所有的过滤器都必须在他们所操作的各种对象中记录一些他们自己的状态。让微过滤器在这些对象中拥有他们的上下文是过滤管理器的一个重要特点。

一个上下文是使用FltAllocateContext()分配一的一片动态内存区。这个调用传入需要的内存空间的大小并返回一个内存空间指针。系统会绑定一个内部的数据头用来跟踪这个指针所对应的上下文。

以下种类的对象支持上下文：

卷 - 一个已经挂载的设备。
实例 - 一个微过滤器对对一个卷的一次绑定。一个微过滤器可能对一个卷绑定多次。如果一个微过滤器对一个卷只能绑定一次，那么推荐用卷上下文来代替实例上下文，这样效率高多了。
文件 - 指关于一个文件的所有打开的流。一般这些上下文是不支持的。
流 - 文件上的一个单独的数据流。
流句柄 - 一个文件的一次打开，比如一个文件对象。

## 9.1 上下文注册

注册的时候,微过滤器定义它所想使用的上下文的类型,大小以及一个用来清理上下文的例程。微过滤器用一组FLT_CONTEXT_REGISTRATION结构来确定这些参数.

下面解释FLT_CONTEXT_REGISTRATION的细节:

ContextType: 注册的上下问的类型。
Flag: 表示这个上下文的一些特殊处理信息。当前的定义有:FLT_CONTEXT_REGISTRATION_NO_EXACT_SIZE_MATCH:默认的情况下，过滤管理器会比较一个给定的上下文的请求的长度和分配上下文的时候指定的数据长度。如果指定了这个标记，如果请求分配的内存大小少于等于注册的时候所指定的长度，过滤权利气回使用特殊指定的分配例程。当注册时候指定的大小为FLT_VARIABLE_SIZED_CONTEXT或者分配和释放例程都已经指定了的时候，这个标记被忽略。
CleanupContext: 当过滤管理器决定应该清理这个上下问的时候这个例程被调用。如果在上下文被释放之前没有什么需要清理的，这个可以设置为NULL.

Size:　　这个上下文的字节大小。这用来允许过滤管理器使用内存池技术（如旁视列表）来让分配和释放更加有效率。如果使用了自己的分配和释放例程，这个域被忽略。
PoolTag: 分配内存的“池”上下文。这是使用ExAllocatePoolWithTag的时候用的。如果使用自己的分配和释放例程，这个域被忽略。
Allocate:如果打算使用自己的分配例程，设置这个指针。如果打算依赖过滤管理器的内存池技术，那么这个应该设置为NULL.
Free:　　如果想使用自己的释放例程，那么这个必须设置为非空。

如果一个微过滤器有相同类型的上下文但是长度不一,它可以为同类型的上下文注册不同的FLT_CONTEXT_REGISTRATION结构来利用过滤管理器的内存池技术。

## 9.2 上下文生成接口

下面是FltAllocateContext()的函数原型：

```
NTSTATUS
FLTAPI
FltAllocateContext (
    IN PFLT_FILTER Filter,
    IN FLT_CONTEXT_TYPE ContextType,
    IN SIZE_T ContextSize,
    IN POOL_TYPE PoolType,
    OUT PFLT_CONTEXT *ReturnedContext
    );
```

ContextType可以是以下的情况:FLT_VOLUME_CONTEXT , FLT_INSTANCE_CONTEXT, FLT_FILE_CONTEXT, FLT_STREAM_CONTEXT, 或者是FLT_STREAMHANDLE_CONTEXT.

下面是一组用来把一个上下文绑定到一个对象上的例程。请注意上下文的类型和对象类型必须是配套的。

```
NTSTATUS
FLTAPI
FltSetVolumeContext (
    IN PFLT_VOLUME Volume,
    IN FLT_SET_CONTEXT_OPERATION Operation,
    IN PFLT_CONTEXT NewContext,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltSetInstanceContext (
    IN PFLT_INSTANCE Instance,
    IN FLT_SET_CONTEXT_OPERATION Operation,
    IN PFLT_CONTEXT NewContext,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );
```

```
NTSTATUS
FLTAPI
FltSetFileContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    IN FLT_SET_CONTEXT_OPERATION Operation,
    IN PFLT_CONTEXT NewContext,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltSetStreamContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    IN FLT_SET_CONTEXT_OPERATION Operation,
    IN PFLT_CONTEXT NewContext,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltSetStreamHandleContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    IN FLT_SET_CONTEXT_OPERATION Operation,
    IN PFLT_CONTEXT NewContext,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );
```

设置一个上下文的时候有两种类型的操作可能发生。 它们是:

FLT_SET_CONTEXT_KEEP_IF_EXISTS:如果没有存在的上下文，那么将设置新的上下问。如果有存在的，那么新的上下文将不会设置上去，而且会返回一个错误代码。如果OldContext参数定义了，那么已经存在的上下文会返回。如果有，调用者必须释放返回的上下文。如果新的上下文没设置上去，那么调用者必须负责释放它。

FLT_SET_CONTEXT_REPLACE_IF_EXISTS: 即使旧的上下文存在，新的也回释放上去。如果OldContext定义了，那么被取代的上下文会返回。调用者必须自己释放它。如果没有定义，则旧的上下文会被自动释放。

当相关设备对象被系统释放，过滤管理器会在合适的时机调用微过滤器来清理上下文。微过滤器可能希望某个时候自己删除一个对象上的上下文。为此，微过滤器可以调用以下的一个例程来删除上下文。当然前提是它拥有这个上下文的指针。

```
VOID
FLTAPI
FltDeleteContext (
```

```
    IN PFLT_CONTEXT Context
    );
```

如果没有指针呢？它必须通过指定对象来删除上下文。可以使用以下这些例程中的某个:

```
NTSTATUS
FLTAPI
FltDeleteVolumeContext (
    IN PFLT_FILTER Filter,
    IN PFLT_VOLUME Volume,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltDeleteInstanceContext (
    IN PFLT_INSTANCE Instance,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltDeleteFileContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltDeleteStreamContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltDeleteStreamHandleContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );
```

如果OldContext参数为空，那么过滤管理器会释放对这个上下文的所有的引用。否则，上下文会通过OldContext返回。微过滤器必须自己

调用FltReleaseContext来释放它。

## 9.3 上下文获取（Retrieval）接口

下面的例程用来获取某个设备的相关上下文。使用完毕，调用者必须释放返回的上下文，释放使用FltReleaseContext().上下文不能在DPC中断级获取。所以如果一个后操作回调中希望得到一个上下文，那么必须从预操作中获得并传入。

```
NTSTATUS
FLTAPI
FltGetVolumeContext (
    IN PFLT_FILTER Filter,
    IN PFLT_VOLUME Volume,
    OUT PFLT_CONTEXT *Context
    );

NTSTATUS
FLTAPI
FltGetInstanceContext (
    IN PFLT_INSTANCE Instance,
    OUT PFLT_CONTEXT *Context
    );

NTSTATUS
FLTAPI
FltGetFileContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *Context
    );

NTSTATUS
FLTAPI
FltGetStreamContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *Context
    );

NTSTATUS
FLTAPI
FltGetStreamHandleContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *Context
    );
```

当使用完毕，下面的一些例程用来释放获得的上下文。一般推荐不要在操作之间传递这些上下文指针。上下文的获取非常有效率，是专门设计用来在每个操作要使用的时候专门来获取上下文的。

```
VOID
FLTAPI
FltReleaseContext (
    IN PFLT_CONTEXT Context
    );
```

类似实例通知例程，每个操作的回调例程都收到一个FLT_RELATED_OBJECTS结构。这个结构包含所有这个操作相关的所有已知的的对象。为了简化上下文的获取，有一个类似的FLT_RELATED_CONTEXT可以一次获取。这个结构如下：

```
typedef struct _FLT_RELATED_CONTEXTS {

    PFLT_CONTEXT VolumeContext;
    PFLT_CONTEXT InstanceContext;
    PFLT_CONTEXT FileContext;
    PFLT_CONTEXT StreamContext;
    PFLT_CONTEXT StreamHandleContext;

} FLT_RELATED_CONTEXTS, *PFLT_RELATED_CONTEXTS;
```

接下来两个例程用来依次获得多个例程，此外也有一次性释放。对于FltGetContexts()调用者指定（在DesiredContext参数中）需要的上下文。在内部，一次获得多个上下文比一个一个的获得它们效率高。当然，对于不需要的上下文最好是不要去获取它。FLT_ALL_CONTEXTS可以用来得到所有可用的上下文。

```
VOID
FltGetContexts (
    IN PFLT_RELATED_OBJECTS FltObjects,
    IN FLT_CONTEXT_TYPE DesiredContexts,
    OUT PFLT_RELATED_CONTEXTS Contexts
    );

VOID
FltReleaseContexts (
    IN OUT PFLT_RELATED_CONTEXTS Contexts
    );
```

## 9.4 上下文释放接口

当过滤管理器决定了一个上下文要被释放的时候，微过滤器的相关回调会被调用。这个回调例程应该清理任何上下文（包括清理分配的内存，释放资源等等）。通过返回，过滤管理器会释放传入的上下文结构。

每个类型的上下文都要有一个对应的清理例程。这些例程定义如下:

```
typedef VOID
(*PFLT_CONTEXT_CLEANUP_CALLBACK) (
    IN PFLT_CONTEXT Context,
    IN FLT_CONTEXT_TYPE ContextType
    );
```

同一个清理例程可以注册给多个不同类型的上下文。

# 10. 与用户态的通信

## 10.1 过滤器通信端口对象

为了实现安全的和支持多种不同类型通信方法，一个新的对象被引入：微过滤器通信端口（以下简称通信端口或者端口）。专门设计用来给核心态-用户态通信或者反过来。核心-核心通信现在不在支持。一个端口是一个有名字的NT对象。而且有一个安全的描述符号。

过滤管理器生成了一个新的对象类型，FilterConnectionPort来实现这个。过滤管理器在它的DriverEntry中生成这个新的对象类型，赶在了任何微过滤器加载之前。

只有核心模式的驱动才能生成一个通信端口，使用以下的调用:

```
NTSTATUS
FltCreateCommunicationPort(
    IN PFLT_FILTER Filter,
    OUT PHANDLE PortHandle,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN PVOID ServerPortCookie OPTIONAL,
    IN PFLT_CONNECT_NOTIFY ConnectNotifyCallback,
    IN PFLT_DISCONNECT_NOTIFY DisconnectNotifyCallback,
    IN PFLT_MESSAGE_NOTIFY MessageNotifyCallback,
    IN ULONG MaxConnections
    );
```

Filter是微过滤器的过滤器句柄。成功生成之后，端口的句柄在PortHandle参数中返回。
和其他的NT对象一样，ObjectAttributes参数定义了OBJECT_ATTRIBUTES结构来初始化要生成的端口对象的名字，对象属性和安全描述符等。

请注意属性中OBJ_KERNEL_HANDLE标记必须设置。因为通信端口只能是核心对象。

ServerPortCookie是一个上下文。微过滤器可以通过这个和端口联系在一起。这个上下文对过滤管理器是不透明的。所有的连接，中断通知，都会同过这个上下文才能传递给微过滤器。有些过滤器可能要生成一组通信端口，又想功用一个同志例程。那么可以通过这个上下文中保存的数据进行区分。

调用者还可以注册一些回调函数:

ConnectNotifyCallback(): 当一个用户态进程尝试打开一个端口的时候，这个例程被调用。过滤器可以选择把这个请求失败掉。通知例程回手到一个关于此连接的句柄。每一个连接有唯一的一个句柄。ServerPortCookie 也会传入。微过可以填写ConnectionCookie为一个上下文。这个上下文会传到所有的用户态传来的消息以及连接中断例程中。

DisconnectNotifyCallback(): 当一个端口被用户态关闭的时候会调用这个回调。(也就是打开计数到0的时候)。

MessageNotifyCallback(): 任何时候手到一个消息都会调用这个。

MaxConnections指出了这个通信端口上允许的最大向外连接数。这没有默认值，必须设置得大于0。

并不能保证所有的对象名会生成在根名字空间。有可能过滤管理器把它们映射在\FileSystem\Filters目录下。不过即使如此，对微过滤器和用户态应用程序来说，这是透明的。当引用了一个通信断口的名字，那么所有的足见都应该使用同样的名字。例子Sacnner Minifiter中展示了这是怎么做的。

对应于新的对象类型，新的访问方式也被引入了。有新的访问方式如下:
FLT_PORT_CONNECT
FLT_PORT_ALL_ACCESS
这是一些访问类型。调用者可以设置这些来给使用者权限。用于构造安全描述符的时候，使用InitializeObjectAttributes()。

设置了FLT_PORT_CONNECT,那么我们的应用程序足可以连接这个端口并发送和接受消息。

微过滤器生成一个端口之后，端口就会开始侦听可能的连接。直到你使用ZwClose()将它关闭为止。

## 10.2 从用户态连接到通信端口

微过滤器的通信端口模型和旧模型的过滤器一样，是不对称的。核心态端生成，用户态端连接。有一个接口用来给用户态应用打开一个端口。当端口建立，ConnectNotify()例程被调用来通知微过滤器。

用户态下连接一个端口的编程接口原型如下：
```
HRESULT
FilterConnectCommunicationPort(
    IN LPWSTR lpPortName,
    IN DWORD dwOptions,
    IN LPVOID lpContext,
    IN WORD wSizeOfContext,
    IN LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    OUT HANDLE *hPort
);
```

lpPortName是一个宽字符格式的字符串，指出要连接的端口的名字。这个名字应该和微过滤器生成这个端口的时候一样。名字可以以"\"开头表示在根路径中。过滤管理器会在合适的路径下打开他们并管理很多微过滤器通信端口。

dwOptions现在没有使用。
LpContext是一个指针。指向一个不透明的参数块。这个参数会被传入到ConnectNotify()中。比如可以用来鉴别请求生成端口的应用程序的版本。wSizeOfContext指出这个上下文的字节数。

LpSecurtityAttributes指出给用户端的安全权限。如果这个句柄是继承得到的。

如果这个调用不成功，合适的HRESULT会返回。

返回得到的句柄可以被管理或者被复制。这要通过一些编程接口。它也可以和一个I/O完成端口绑定。

当这个接口被调用，一个新的核心态的无名端口对象就生成了（类型是FilterCommunicationPort）.这个对象用来表示这个连接。微过滤器的ConnectNotify()例程被调用，从而得到通知。而且也得到连接端口的句柄，用来从核心态发送消息。

如果调用者没有访问这个服务端口的权限，或者已经达到了最大连接数，那么调用会失败。

## 10.3 中断与通信端口的连接

当用户态程序调用CloseHandle()或者核心态调用ZwClose()来关闭连接句柄的时候，连接会中断。

只有用户态调用CloseHandle()的时候，微端口的DisconnectNotify()例程才会调用。

理想情况下，微过滤器应该总是在连接结束的时候，在DisconnectNotify()中关闭连接。如果一个微过滤器在其他地方关闭句柄，那么它必须用一些同步方法确保不会在DisconnectNotify()中关闭再次关闭它。

当一个连接在核心态或者用户态被中断了，以下情况发生:

1. 所有用户态的等待（通过FilterGetMessage）被清理掉,而且以STATUS_FLT_PORT_DISCONNECTED结束掉（被解释为win32错误码ERROR_DISCONNECTED）。
2. 所有核心的被阻塞的发送例程会以STATUS_FLT_PORT_DISCONNECTED结束阻塞。
3. 因为端口变无效了，所以不可能有其他的等待或者阻塞的情况出现。

微过滤器总是可以对服务端口的句柄调用ZwClose()来关闭服务端口。这并不会使已经建立的连接中断，但是会阻止新连接的建立。

## 10.4 卸载

微过滤器总是必须在FltUnregisterFilter()调用之前，在FilterUnload例程或者更早关闭服务端口。否则系统可能在卸载例程中被挂起。

即使在有一些连接打开的情况下（比如用户态一放已经打开了一些连接句柄），微过滤器也应该允许被卸载。这种情况下，过滤管理器会尝试强行终止这些连接。过滤管理器会调用DisconnectNotify（）例程。微过滤器应该在这里关闭这些连接句柄以避免句柄的泄漏。

# 11.文件名处理

通过查找操作的参数或者询问文件系统，过滤管理器能得到对象的名字。因而过滤管理器提供一组调用来方便获取对象名。为了更高的效率，过滤管理器也暂存一些对象名。当很多过滤器经常查一个名字的时候，暂存这个名字所付出的代价是很值得的。

查询对象名的时候，过滤管理器返回一个FLT_FILE_NAME_INFORAMTION结构来避免数据拷贝。这些结构被根据过滤器的请求次数计数。只有通过过滤管理器的编程接口才能改变这些结构中的数据。下面有这些结构的详细信息。

## 11.1 从操作获得一个文件名

可以在当前操作的CallbackData->Iopb->TargetFileObject中得到文件名。这需要调用下面的例程：

```
NTSTATUS
FLTAPI
FltGetFileNameInformation (
    IN PFLT_CALLBACK_DATA CallbackData,
    IN FLT_FILE_NAME_FORMAT NameFormat,
    IN FLT_FILE_NAME_QUERY_METHOD QueryMethod,
    OUT PFLT_FILE_NAME_INFORMATION *FileNameInformation
    );
```

CallbackData是这个操作的FLT_CALLBACK_DATA结构。现在假设过滤器想从这个操作中得到文件名。

名字的格式是以下三种:

FLT_FILE_NAME_NORMALIZED_FORMAT:请求全路径名字，包括卷名。所有的短名被扩展成长名。任何流名足见回去掉后边的“：$DATA”.如果这是一个目录名，且不是根目录，最后的“\”会被去掉。

FLT_FILE_NAME_OPENED_FORMAT:　　包含全路径，包括卷名。但是这个名字和打开这个对象所用的名字相同。因此可能在路径中包含一些短名。

FLT_FILE_NAME_SHORT_FORMAT: 仅仅含有路径中最后一个元素的短名（Dos名），不会返回全路径。

QueryMethod应该是以下之一:

FLT_FILE_NAME_QUERY_DEFAULT:　　搜索一个名字的时候，管理器会首先找暂存的名字。然后再询问文件系统。

FLT_FILE_NAME_QUERY_CACHE_ONLY: 仅仅在暂存中找。如果失败，返回STATUS_FLT_NAME_CACHE_MISS.

FLT_FILE_NAME_QUERY_FILE_SYSTEM_ONLY:仅仅询问文件系统，不会从暂存中去寻找这个名字。

名字在最后一个参数中返回，FileNameInformation.这个结构是一组共享缓冲的Unicode字符串。不同的字符串表明不名字中不同的部分.

```
typedef struct _FLT_FILE_NAME_INFORMATION {
    USHORT Size;
    FLT_FILE_NAME_FORMAT Format;
    FLT_FILE_NAME_PARSED_FLAGS NamesParsed;
    UNICODE_STRING Name;
    UNICODE_STRING Volume;
    UNICODE_STRING Share;
    UNICODE_STRING Extension;
    UNICODE_STRING Stream;
    UNICODE_STRING FinalComponent;
    UNICODE_STRING ParentDir;
} FLT_FILE_NAME_INFORMATION, *PFLT_FILE_NAME_INFORMATION;
```

当一个文件名信息结构从FltGetFileNameInforamtion()返回，name,Volume,Share(用于远程文件)会被解析出来。如果一个微过滤器需要其他的名字信息，它应该调用FltParseFileNameInformation().

运行在DPC或者更低级的中断级别的时候，微过滤器可以在IO过程中任何地方调用FltGetFileNameInformation（）.当它在一个可能导致死锁的情况下请求（比如处理分页交换的时候），如果在暂存中没有找到名字或者指定了必须从文件系统读取那么调用会失败。

即使没有回调数据（Callback Data）存在，仅仅知道文件对象（FileObject）的时候，微过滤器也可以调用FltGetFileNameInformationUnsafe（）来获得一个文件名。不过过滤器必须知道当前向文件系统询问这个名字是安全的。这个调用不能像FltGetFileNameInformation（）那样检查是否导致死活。微过滤器必须自己保证它。

当一个名字使用完毕，应该调用以下的例程来释放:

```
FLTAPI
FltReleaseFileNameInformation (
    IN PFLT_FILE_NAME_INFORMATION FileNameInformation
    );
```

过滤管理器的名字暂存对微过滤查找对象的名字来说效率足够了。名字暂存机制也管理了由于重新命名而无效的名字。维护一个名字暂存空间的复杂逻辑对微过滤器们而言是透明的。但是微过滤器对于无效的名字并不是完全不需要关心。当一个重命名发生了，过滤管理器清理所有的受影响的暂存文件名。但是微过滤器可能引用过一个过时的文件名。当这些引用全部被释放的时候，过时的文件名信息结构才会被真的释放掉。如果一个微过滤器请求询问一个已经被重新命名的对象，将尽可能的返回新的名字。

## 11.2 文件名的附加支持

过滤管理器也提供了一个编程接口来帮助微过滤器获得一个改名或生成硬连接的操作的目的名:

```
NTSTATUS
FltGetDestinationFileNameInformation (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    IN HANDLE RootDirectory OPTIONAL,
    IN PWSTR FileName,
    IN ULONG FileNameLength,
    IN FLT_FILE_NAME_FORMAT NameFormat,
    IN FLT_FILE_NAME_QUERY_METHOD QueryMethod,
    OUT PFLT_FILE_NAME_INFORMATION *RetFileNameInformation
    );
```

这个接口只能在IRP_MJ_SET_INFORMATION的预操作回调中调用，包括FileSetNameInformation和FileSetLinkInformation。调用者从这个操作中获得参数，这个调用会返回一个FLT_FILE_NAME_INFORMTION结构，包含了目标文件名在其中。与使用FltGetFileNameInformation（）相同，这个调用返回的名字使用完毕之后，应该调用FltReleaseFileNameInformation（）来释放。

命名隧道（Name Tunneling）是过滤器的另一个困绕之处。过滤管理器提供了一个编程接口来获得一个命名隧道需要的新名字：

```
NTSTATUS
FltGetTunneledName (
    IN PFLT_CALLBACK_DATA CallbackData,
    IN PFLT_FILE_NAME_INFORMATION FileNameInformation,
    OUT PFLT_FILE_NAME_INFORMATION *RetTunneledFileNameInformation
    );
```

命名隧道仅仅影响获取文件名采用通常格式的微过滤器。如果一个微过滤器在它的一个预操作需要一个通常格式的文件名,这个预操作回调的来源是CREATE，改名，或生成硬连接的话，它应该在后操作回调中调用FltGetTunneledName()来确认这个名字是否被操作系统命名隧道所改过。如果命名隧道确实出现了，那么RetTunneledFileNameInformation中回返回一个新的文件名字信息结构。微过滤器必须用这个新的名字，并且处理完毕后必须用FltReleaseFileName（）来释放它。

（译者注：什么是命名隧道（Name Tunneling）

长文件名出现之后，旧的16位应用程序随时可能破坏掉长文件名。为此出现了所谓的命名隧道概念。现在假设一个16位的程序比如文字处理程序把当前版本的文挡维护在一个临时文件中。当用户修改这个文件，原始的文件就被删除了，临时文件被改为原来的文件名。

如果原始文件有一个长文件名，但是临时文件仅仅有短文件名，那么当旧的文件被删除，名字也跟着丢失了。因此当命名隧道起作用的时候，文件系统记住了每个被删除的文件名一段时间（比如15秒），如果一个短文件名的文件生成了，刚好和被记忆的长文件名配套，那么短文件名自动改名为长文件名。这就是命名隧道概念。

想自己尝试一下：首先，在一个空文件夹中生成一个文件 longfilename.然后删除它。在生成一个文件longfi~1,然后输入dir /x,这时你发现，longfilename又出现了!）

## 11.3 名字提供接口

如果一个过滤器打算提供一个途径来改变名字空间，他必须注册三个附加的回调给过滤管理器。这些回调允许过滤器作为名字的提供者。过滤器将有责任对上层发来的FltGetFilenameInformation或者FltGetDestinationFileName返回FLT_FILE_NAME_INFORMATION结构，并填写其中名字的内容。而且这样的微过滤器还可以告知过滤管理器，他们所返回的名字要不要被暂存。

作为一个名字提供者，过滤器必须可以返回一个指定的文件对象（file object）的开放格式的名字。如果确定了通常名字格式，过滤管理器将重新遍历这个名字的所有部分并调用名字提供者的徽调来展开这些部分为一个通常格式的文件名。在展开一个路径的所有部分的过程中，过滤器的名字通常化例程可能被调用不止一次。因此这个过程中允许过滤器传入一个上下文。所有的过程结束后，如果调用返回了，过滤器会被要求清理这个上下文。

当一个过滤器作为名字提供者必须提供一个名字的时候，它的PFLT_GENERATE_FILE_NAME例程会被调用。你能在参数中得到文件对象，过滤器的实例，同时回调数据描述发生的操作。此外还有名字请求选项。那么过滤器必须在这中间生成名字：

```
typedef NTSTATUS
(*PFLT_GENERATE_FILE_NAME) (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    IN PFLT_CALLBACK_DATA CallbackData OPTIONAL,
    IN ULONG NameOptions,
    OUT PBOOLEAN CacheFileNameInformation,
    OUT PFLT_NAME_CONTROL FileName
    );

typedef NTSTATUS
(*PFLT_NORMALIZE_NAME_COMPONENT) (
    IN PFLT_INSTANCE Instance,
    IN CONST PUNICODE_STRING ParentDirectory,
    IN USHORT VolumeNameLength,
    IN CONST PUNICODE_STRING Component,
    IN OUT PFILE_NAMES_INFORMATION ExpandComponentName,
    IN ULONG ExpandComponentNameLength,
    IN OUT PVOID *NormalizationContext
    );

typedef VOID
(*PFLT_NORMALIZE_CONTEXT_CLEANUP) (
    IN PVOID *NormalizationContext
    );
```

如果一个过滤器希望让所有自己提供的名字的暂存失效，他可以调用以下的接口。在此之前它可能已经提供了一些FLT_FILE_NAME_INFORMATION结构，而且其他过滤器可能刚好还在使用。那么这些FLT_FILE_NAME_INFORMATION结构只有当引用数降到0（已

经没有人使用了），才会自己释放掉。

```
NTSTATUS
FltPurgeFileNameInformationCache (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject OPTIONAL
    );
```

为了确保这样一个过滤可以卸载，过滤管理器有责任管理这些过滤器提供的文件名的暂存和释放。

一些FltGetFileNameInformation（）调用FltGetDestinationFileName（）调用基于一些过滤器提供的名字。过滤管理器将负责初始化这些FLT_FILE_NAME_INFORMATION结构。

为了提高效率，过滤管理器往名字提供者的PFLT_GENERATE_FILE_NAME回调传入一个缓冲区。这个缓冲是一个FLT_NAME_CONTROL所包装的UNICODE_STRING.这个结构中包含一些公有的或者私有的信息。在一个过滤器试图填充这个缓冲之前，应该先检查缓冲是否足够的大：

```
NTSTATUS
FltCheckAndGrowNameControl (
    IN OUT PFLT_NAME_CONTROL NameCtrl,
    IN USHORT NewSize
    );
```

如果这个调用返回STATUS_SUCCESS,说明FLT_NAME_CONTROL结构足够容纳从名字提供者返回的名字了。

# 12.过滤器自产生I/O

某些微过滤器需要执行他们自己的I/O请求。在卷的微过滤器栈中，只有此过滤器以下的过滤器才能收到这些I/O请求。比如，一个防毒软件，可能希望在打开一个文件之前先读一下这个文件。在新的微过滤模式下，一个微过滤器有两种方法生成自己的请求:使用生成例程，类似现在的其他例程。此外就是使用旧的ZwXxx例程。

主要的I/O生成例程如下:

```
NTSTATUS
FLTAPI
FltAllocateCallbackData (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject OPTIONAL,
    OUT PFLT_CALLBACK_DATA *RetNewCallbackData
    );

VOID
FLTAPI
FltPerformSynchronousIo (
    IN PFLT_CALLBACK_DATA CallbackData
    );

NTSTATUS
FLTAPI
FltPerformAsynchronousIo (
    IN PFLT_CALLBACK_DATA CallbackData,
    IN PFLT_COMPLETED_ASYNC_IO_CALLBACK CallbackRoutine,
```

```
    IN PVOID CallbackContext
    );
```

要使用这些例程，一个微过滤器可以首先调用FltAllocateCallbackData来分配一个 CallbackData.然后对应于不同的操作填写合适的参数。之后即可调用FltPerformSynchronousIo()或者是FltPerformAsynchronousIo()来实际发起I/O请求。参数Instance必须总是微过滤器发起此请求的实例。（译者注:不能由A实例分配而B实例来发起请求）。

此外过滤管理器导出了一些常用的实用例程，比如:

```
NTSTATUS
FLTAPI
FltCreateFile (
    IN PFLT_FILTER Filter,
    IN PFLT_INSTANCE Instance OPTIONAL,
    OUT PHANDLE   FileHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PLARGE_INTEGER AllocationSize OPTIONAL,
    IN ULONG FileAttributes,
    IN ULONG ShareAccess,
    IN ULONG CreateDisposition,
    IN ULONG CreateOptions,
    IN PVOID EaBuffer OPTIONAL,
    IN ULONG EaLength,
    IN ULONG Flags
    );
```

如果Instance参数被忽略(译者注:指定为空.)，那么CREATE请求会被发送到微过滤器栈的顶端(这导致本微过滤器会回环的看到自己发送的请求。)除非绝对必要，这是不明智的。如果被滥用，很容易导致死锁和栈溢出.

如果指定了这个参数(应该总是你自己当前的instance),那么仅仅向下发起此请求。所有的前面的调用此api的微过滤器之上的微过滤器(包括自己)都不会收到此请求.

FileHandle参数返回一个供Zw*系列函数调用的文件句柄.如果先前指定的Instance不为空，那么保证关于这个文件句柄的其他未来的I/O操作(通过Zw接口,FltClose(),等),都只能被这个发起Instance下面的Instance看到。

FltReadFile()和FltWriteFile()发起的I/O读写请求也只有下层实例才能看见。用于我们只有FileObject而没有文件句柄的时候。这些例程类似以前旧的过滤模型中自己发送IRP的读写方式.

重要提示:过滤器不需要使用FltReadFile()/FltwriteFile()来对一个FltCreateFile返回的文件句柄发起一个I/O请求.对于这种情况，使用Zw*()接口就可以发送到合适的实例上去。

```
NTSTATUS
FLTAPI
FltReadFile (
    IN PFLT_INSTANCE InitiatingInstance,
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER ByteOffset OPTIONAL,
    IN ULONG Length,
    OUT PVOID Buffer,
    IN FLT_IO_OPERATION_FLAGS Flags,
    OUT PULONG BytesRead OPTIONAL,
    IN PFLT_COMPLETED_ASYNC_IO_CALLBACK CallbackRoutine OPTIONAL,
    IN PVOID CallbackContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltWriteFile (
    IN PFLT_INSTANCE InitiatingInstance,
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER ByteOffset OPTIONAL,
    IN ULONG Length,
    IN PVOID Buffer,
    IN FLT_IO_OPERATION_FLAGS Flags,
    OUT PULONG BytesWritten OPTIONAL,
    IN PFLT_COMPLETED_ASYNC_IO_CALLBACK CallbackRoutine OPTIONAL,
    IN PVOID CallbackContext OPTIONAL
     );
```

微过滤器发出I/O可能为同步也可能为异步.当I/O被指定为异步,微过滤器提供一个回调例程.当系统完成了I/O请求回调例程会被调用.

# 13 卸载/注销/解除绑定的规则

解除绑定意味着一个微过滤实例将被销毁.这个微过滤器不在会因为任何对卷的操作而被调用(当然除非还有别的实例绑定在这个卷上).

卸载一个微过滤器意味着它的代码将不在存在于内存中。这往往在系统关闭的时候发生。或者是一个新版本的微过滤器在不关闭系统的情况下安装了(译者注:那么这时旧版本的微过滤器被卸载了。)

一个微过滤器可以自己把自己从一个卷解除绑定(调用FltDetachVolume),不过更通常的情况是在用户界面上解除的。一个微过滤器即使是有未完成的I/O请求的时候，依然可以解除绑定.此时,微过滤器的完成例程会被调用,而且所有的未完成的I/O操作都带有标记FLT_COMPLETION_DETACHED.当这些操作以后实际完成时，微过滤器就不会再收到这些完成回调了。

当一个微过滤器实例被解除绑定的时候，系统回调用这个微过滤器的所有的没有清除的上下文的释放例程,包括文件,流,流文件对象等和这个实例相关的对象。

# 14. 支持例程

除了已经讨论过的接口之外，过滤管理器提供了一组支持例程来帮助微过滤器完成他需要的工作。这里列出一部分附加的例程。请查阅过滤管理器IFS文档来获得这些例程的更多信息：

对象，名字转换：

FltGetFilterFromName()

FltGetVolumeFromName()

FltGetVolumeInstanceFromName()

卷，实例，设备对象转换历程：

FltGetVolumeFromInstance(), FltGetFilterFromInstance()

FltGetVolumeFromDeviceObject()

FltGetDeviceObject()

FltGetDiskDeviceObject()

获取对象信息例程：

FltGetVolumeProperties()

FltIsVolumeWriteable

FltQueryVolumeInformation(), FltSetVolumeInformation()

FltGetInstanceInformation()

FltGetFilterInformation()

枚举例程：

FltEnumerateFilters()

FltEnumerateVolumes()

FltEnumerateInstances()

FltEnumerateFilterInformation()

FltEnumerateInstanceInformationByFilter()

FltEnumerateInstanceInformationByVolume()

FltEnumerateVolumeInformation()

Oplock例程：

FltInitializeOplock()

FltUninitializeOplock()

FltOplockFsctrl()

FltCheckOplock()

FltOplockIsFastIoPossible()

FltCurrentBatchOplock()

目录改变通知例程：

（译者注：目录改变通知在文件系统中的意义是当目录改变后，通知操作系统，使操作系统可以在界面上做出一些改变，比如显示的目录结构变化等等，专门有一种IRP可以干这个事情。） FltNotifyFilterChangeDirectory()

其他：FltGetRequestorProcess(), FltGetRequestorProcessId()

# 15. 构建一个微过滤器

构建一个微过滤器应用所需要的所有的东西都可以在过滤管理器IFS开发包中找到。这包括：

1. 用来构建一个微过滤器和一个使用了过滤管理器接口的用户态应用程序的完整的构建环境。

2. 所有的在一台机器上安装过滤管理器组件用于开发的安装包。当然如果以后的windows版本内部包含了过滤管理器组件，那么就没有必要了。

3.微过滤器示例的代码。

所有的微过滤器包含头文件FltKernel.h，而且连接FltMgr.lib.而应用程序应该包含头文件FltUser.h，并连接库FltLib.lib。