DirectShow SDK 学习笔记

作者：智慧的鱼
编辑：中华视频网

中华视频网：http://www.chinavideo.org

佰锐科技：http://www.bairuitech.com

目 录

绪言 6

1 ABOUT DIRECTSHOW 基础 7

1.1 设置 DSHOW 的开发环境 7
1.2 先演示一下 DSHOW 使用的一个例子 8
1.3 DIRECSHOW 概述 8
1.4 FILTER GRAPH 及其组成 8
1.5 构建一个 FILTER GRAPH 图 15
1.6 数据流在 FILTER GRAPH 里的流动（DATA FLOW） 15
1.7 事件通知机制（EVENT NOTIFICATION） 19
1.8DIRECTSHOW 中的时钟（TIME AND CLOCKS IN DSHOW） 21
1.9 动态删除或增加 FILTER（DYNAMIC GRAPH BUILDING） 23
1.10 PLUG-IN DISTRIBUTORS 25

2 DIRECTSHOW 的应用（USING DIRECTSHOW） 25

2.1 在 GRAPHEDIT 中模拟构建 GRAPH（SIMULATING GRAPH BUILDING） 25
2.2DIRECTSHOW 基本应用（BASIC TASKS） 33
2.2.1 视频提交（Video Rendering） 33
2.2.2 如何处理事件通知（Event Notification） 36
2.2.3 如何枚举系统的设备和过虑器 37
2.2.4 如何枚举 Graph 图中的对象（filter，pin） 41
2.2.5 构建 Graph 图常用技术 43
2.2.5.1 如何根据 CLSID 向 graph 中添加 filter 43
2.2.5.2 如何查找 filter 空闲的 pin。 44
2.2.5.3 如何连接两个 Filter 45
2.2.5.4 如何获得 filter 或者 pin 的接口指针 47
2.2.5.5 如何查找和某个 filter 的上下相连的 filter 49
2.2.5.6 如何删除 graph 中的所有 filter 52
2.2.5.7 如何利用 Capture Graph Builder 构建 Graph 图表 53
2.2.6 Seeking Filter graph 55
2.2.7 如何设置 Graph 时钟（Setting Graph Clock） 57
2.2.8 在 Dshow 中如何调试 58
2.3 音频的捕捉 58
2.4 视频的捕捉（VIDEO CAPTURE） 59
2.4.1 关于视频捕捉（About Video Capture in Dshow） 59
2.4.2 选择一个视频捕捉设备（Select capture device） 63
2.4.3 预览视频（Previewing Video） 65
2.4.4 如何捕捉视频流并保存到文件（Capture video to File） 65
2.4.5 如何控制 Capture Graph（Controlling Capture Graph） 69
2.4.6 视频捕捉的任务（Video Capture Tasks） 71
2.4.6.1 如何配置一个视频捕捉设备 71
2.4.6.2Working With Crossbars 74

2.4.6.3 将设备从系统中移走时的事件通知（Device remove Notify） *74*
2.4.6.4 从静止图像 pin 中捕捉图片 *75*
2.4.7 数字视频 DV（Digital Video in Direcshow） *78*
2.4.7.1 关于 Directshow 中的 DV 应用 *78*
2.4.7.2 如何将 DV 捕捉到一个文件中 *78*
2.4.7.3 如何将文件中的 DV 读入到盘中 *78*
2.4.7.4 DVINFO Field Settings in the MSDV Driver *78*
2.4.8 如何控制 DV 便携式摄像机（Controlling a DV Camcorder） *78*
2.4.9 模拟电视的视频捕捉（Analog Television） *78*
2.4.10 视频捕捉的高级话题 *78*
2.4.10.1 处理视频重画事件 *78*
2.4.10.2 如何确定 pin 的种类（Pin Categories） *79*
2.4.10.3 如何使用一个 SmartTee Filer *81*
2.4.10.4 如何使用一个重叠混合器（Overlay Mixer in Video Capture） *81*
2.4.10.5Video Port Pins *81*
2.4.10.6VideoInfo2 Format Type *81*
2.4.10.7 手动添加 WDM 类驱动 filter *81*
2.4.10.8 如何创建内核 filter *83*
2.5DIRECTSHOW EDITING SERVICES 85
2.6DVD 应用（DVD APPLICATION） 85
2.7 MPEP_2 支持 85
2.8WINDOWS MEDIA 应用 85
2.9 TV 应用 85
2.10 使用视频混合 RENDER 86
2.11 USING THE STREAM BUFFER ENGINE 86
2.12 开发自己的 FILTER 86
1 如何开发自己的 filter *86*
2filter 的连接 *95*
3filter 间的数据流动 *98*
4pin 连接时数据格式的动态改变 *101*
4 Threads and Critical Sections *101*
5 质量控制管理 *107*
6dshow 和 com *107*
7 如何写 Transform Filter *120*
8 如何写视频播放过滤器 Video Renderer Filter *131*
9 如何写捕捉 filter（源） *137*
10 创建 filter 属性页 *141*
11 capture and compression formats *148*
12Graph 如何定位 filter 的位置并加载 *150*
2.13 ENCODER AND DECODER 开发 150

3DIRECTSHOW 的基类学习 150

3.1 DSHOW 的基类简介 150
3.2 FILTER 和 PIN 的基类 153
3.2.1CBaseFilter *153*

3.2.2CBasePin *159*
3.2.3CBaseInputPin *168*
3.2.4CBaseOutputPin *172*
3.3 几种常用 FILTER 的基类 176
3.3.1CSource *176*
3.3.2CSourceStream *178*
3.3.3CTransformFilter *182*
3.3.4CTransformInputPin *188*
3.3.5CTransformOutputPin *189*
3.3.6CTransInPlaceFilter *191*
3.3.7 CTransInPlaceInputPin *193*
3.3.8CTransInPlaceOutputPin *193*
3.3.4CVideoTransformFilter *193*
3.3.9CBaseRenderer *194*
3.3.10CRendererInputPin *205*
3.3.11CBaseVideoRenderer *206*
3.3.12 CBaseAllocator *210*
3.3.13 CMediaSample *212*
3.4FILTER 和 PIN 经常用到的类 214
3.4.1CPullPin *214*
3.4.2COutputQueue *216*
3.4.3CSourceSeeking *216*
3.4.4CEnumPins *216*
3.4.5CEnumMediaTypes *216*
3.4.6CMemAllocator *216*
3.4.7CMediaSample *216*
3.4.8CBaseReferenceClock *219*
3.4.9CMediaType *219*
3.5 几个比较重要的类 219

4DIRECSHOW 提供的接口学习 220

5DIRECTSHOW TUTORIALS 220

5.1IMPLEMENTING A SEEK BAR 220
5.2DISPLAYING A FILTER'S PROPERTY PAGES 220
5.3GRABBING A POSTER FRAME 220
5.4USING THE SAMPLE GRABBER 220
5.5RECOMPRESSING AN AVI FILE 220

6C++在电视开发中的应用 220

6.1TV RATINGS REFERENCE 220
6.2VIDEO CONTROL C++ REFERENCE 220
6.3MICROSOFT UNIFIED TUNING MODEL C++ REFERENCE 220
6.4TRANSPORT INFORMATION INTERFACES 220
6.5BDA FILTER INTERFACES 220

6.6MPEG-2 SECTIONS AND TABLES FILTER REFERENCE 220

7DIRECSHOW 提供的 FILTER 220

绪言

DirectShow 是 Windows 平台下的多媒体处理框架，早期（DirectX7.0、DirectX8.0）属于 DirectX 的范畴，在目前（DirectX9.0）微软已经将其放在了 Windows platform SDK 中，所以，你现在在最新版本的 DirectX SDK 目录中是找不到有关 DirectShow 的身影的。

中华视频网（www.chinavideo.org ）一直致力于语音视频技术的研究和推广，早期也收编过多篇“智慧的鱼”的文章，本册《DirectShow SDK 学习笔记》是一篇比较全面介绍 DirectShow 的资料，特收集整理成册，以供后来者学习。

“个人感觉开发自己的 Filter 还是要对 dshow 的基类要熟悉一些。所以才想起了要翻译这些东西，希望在 9 月底前完成这些东西。我不想它的句子有多么流畅，语法的错误有多少，我只希望能看明白就可以了！”

——智慧的鱼

由于某些原因，SDK 中的有些章节没有翻译，如果哪位读者补充以后，可以将补充后的文档发给智慧的鱼或是中华视频网（vvoucom@gmail.com ），我们再整理更完整的版本并发布，互相学习的过程就是提高的过程。

中华视频网：www.chinavideo.org

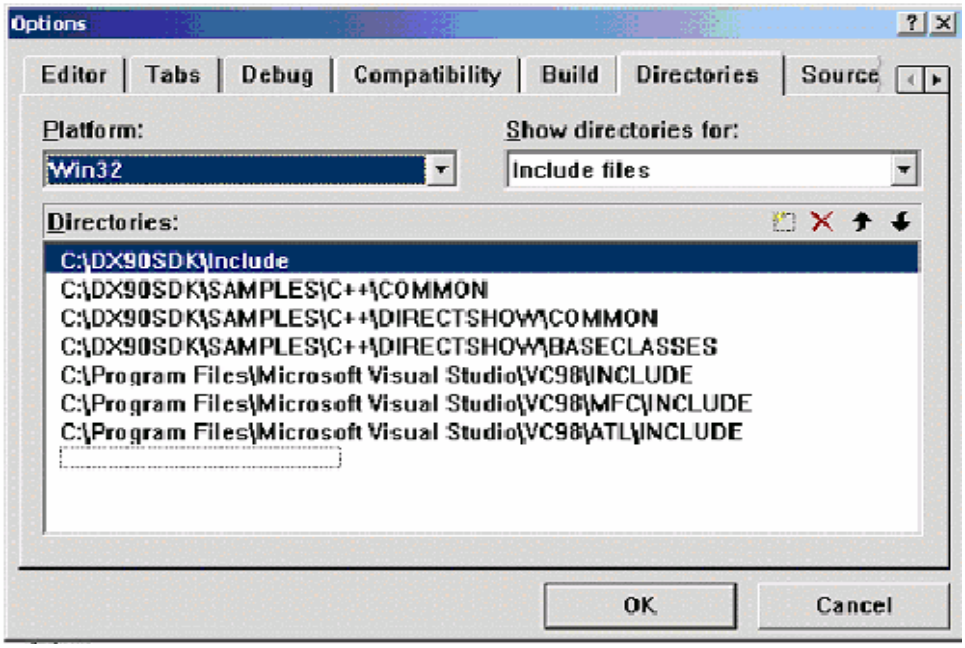
Ffmpeg 工程组：www.ffmpeg.com.cn

佰锐科技：www.bairuitech.com

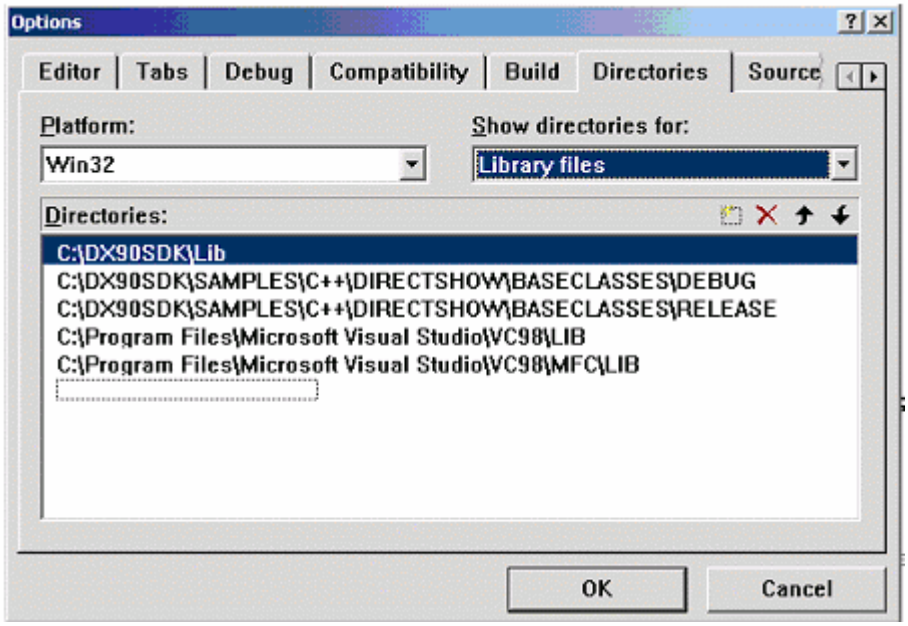
1 About Directshow 基础

1.1 设置 dshow 的开发环境

如果你用 VC 开发环境，一定要在 Setting 里设置下面的东西
包含头文件 Dshow.h 所有的 dshow 应用必须包含



包含动态库 Strmiids.lib 导出所有接口的 CLSID 和接口 IID 定义。必须包含 Quartz.lib



1.2 先演示一下 dshow 使用的一个例子

这里暂略。

1.3 Direcshow 概述

DirectShow 是微软公司提供的一套在 Windows 平台上进行流媒体处理的开发包，与 DirectX 开发包一起发布。

那么，DirectShow 能够做些什么呢？且看，DirectShow 为多媒体流的捕捉和回放提供了强有力的支持。运用 DirectShow，我们可以很方便地从支持 WDM 驱动模型的采集卡上捕获数据，并且进行相应的后期处理乃至存储到文件中。它广泛地支持各种媒体格式，包括 Asf、Mpeg、Avi、Dv、Mp3、Wave 等等，使得多媒体数据的回放变得轻而易举。另外，DirectShow 还集成了 DirectX 其它部分（比如 DirectDraw、DirectSound）的技术，直接支持 DVD 的播放，视频的非线性编辑，以及与数字摄像机的数据交换。更值得一提的是，DirectShow 提供的是一种开放式的开发环境，我们可以根据自己的需要定制自己的组件。

DirectShow 的系统组成

应用程序与 DirectShow 组件以及 DirectShow 所支持的软硬件之间的关系如图 1 所示。

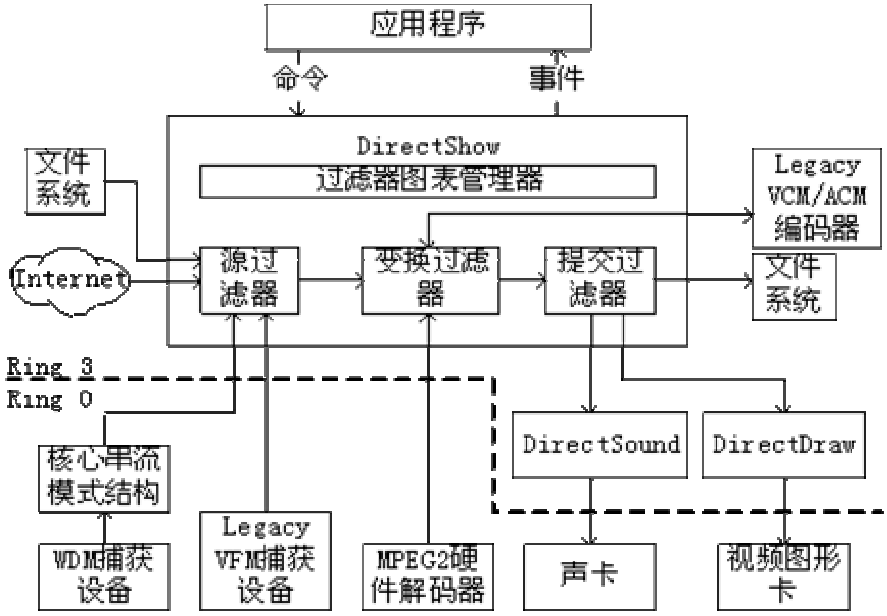


图 1 DirectShow 系统框图

1.4 Filter Graph 及其组成

这篇文档中我想给讲述 Directshow 的主要组成部分，一个概括性的入门文章，对于应用开发或者 directshow 的开发者都有所帮助。

1 DirectShow 的 Filter

Directshow 是基于模块化，每个功能模块都采取 COM 组件方式，称为 Filter。Directshow

提供了一系列的标准的模块可用于应用开发，开发者也可以开发自己的功能 Filter 来扩展 Directshow 的应用。下面我们用一个例子来说明如何采取 Filter 来播放一个 AVI 的视频文件。

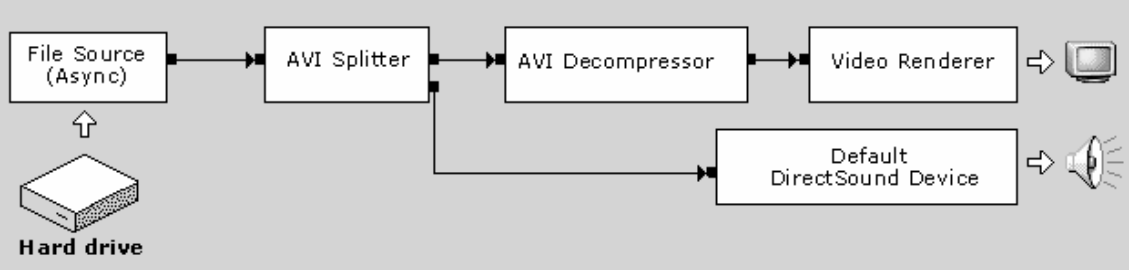
从一个文件读取数据，形成字节流。（这个工作由源 Filter 完成）

检查 AVI 数据流的头格式，然后通过 AVI 分割 Filter 将视频流和音频流分开。

解码视频流，根据压缩格式的不同，选取不同的 decoder filters 。

重画视频图像，通过 Renderer Filter。

将音频流送到声卡进行播放，一般采用缺省的 DirectSound Device Filter。流程见下图。



从上面的图表看，每一个 filter 都一个其他的一个或者两个 filter 相连接，连接点也是 com 对象，我们称为 Pin。Filter 通过 pin 将数据从一个 filter 传递到另一个 filter 中，从而可以使数据的 filter 的链表中流动。图中的箭头表示 filter 链表中的数据流的方向。在 Directshow 中，一个 filter 链表我们称为 filter Graph。

Filter 具有三个状态，运行，停止，暂停。当一个 filter 运行时，它就处理媒体数据流，当停止时，filter 就不在处理数据，暂停状态常用来给运行状态之前 cure data。Data Flow in the Filter Graph 一章详细描述了这些概念，可以参考。

除非特别的例外，所有 Filter graph 中的 filter 的状态的改变都是统一的，也就说，filte graph 中的所有的 filter 的状态改变是一致协调的。也就是说，我们也可以用 filter graph 也可以有运行，停止，暂停三种状态。

Filter 一般分为下面几种类型。

（1）源过滤器（source filter）：源过滤器引入数据到过滤器图表中，数据来源可以是文件、网络、照相机等。不同的源过滤器处理不同类型的数据源。

（2）变换过滤器（transform filter）：变换过滤器的工作是获取输入流，处理数据，并生成输出流。变换过滤器对数据的处理包括编解码、格式转换、压缩解压缩等。

（3）提交过滤器（renderer filter）：提交过滤器在过滤器图表里处于最后一级，它们接收数据并把数据提交给外设。

（4）分割过滤器（splitter filter）：分割过滤器把输入流分割成多个输出。例如，AVI 分割过滤器把一个 AVI 格式的字节流分割成视频流和音频流。

（5）混合过滤器（mux filter）：混合过滤器把多个输入组合成一个单独的数据流。例如，AVI 混合过滤器把视频流和音频流合成一个 AVI 格式的字节流。

过滤器的这些分类并不是绝对的，例如一个 ASF 读过滤器（ASF Reader filter）既是一个源过滤器又是一个分割过滤器。

2 关于 Filter Graph Manager

Filter Graph Manager 也是一个 com 对象，用来控制 Filter graph 中的所有的 filter，主要有以下的功能：

1 用来协调 filter 之间的状态改变，从而使 graph 中的所有的 filter 的状态的改变应该一致。

2 建立一个参考时钟。

3 将 filter 的消息返回给应用程序

4 提供方法用来建立 filter graph。

这里只是简单的描述一下，详细地可以参考文档。

状态改变，Graph 中的 filter 的状态改变应该一致，因此，应用程序并将状态改变的命令直接发给 filter，而是将相应的状态改变的命令发送给 Filter graph Manager，由 manager 将命令分发给 graph 中每一个 filter。Seeking 也是同样的方式工作，首先由应用程序将 seek 命令发送到 filter graph 管理器，然后由其分发给每个 filter。

参考时钟，graph 中的 filter 都采用的同一个时钟，称为参考时钟（reference clock），参考时钟可以确保所有的数据流同步，视频桢或者音频桢应该被提交的时间称为 *presentation time*.presentation time 是相对于参考时钟来确定的。Filter graph Manager 应该选择一个参考时钟，可以选择声卡上的时钟，也可以选择系统时钟

Graph 事件， Graph 管理器采用事件机制将 graph 中发生的事件通知给应用程序，这个机制类似于 windows 的消息循环机制。

Graph 构建的方法，graph 管理器给应用程序提供了将 filter 添加进 graph 的方法，连接 filter 的方法，断开 filter 连接的方法。

但是，graph 管理器没有提供如何将数据从一个 filter 发送到另一个 filter 的方法，这个工作是由 filter 在内部通过 pin 来独立完成的，

3 媒体类型

因为 Directshow 是基于 com 组件的，就需要有一种方式来描述 filter graph 每一个点的数据格式，例如，我们还以播放 AVI 文件为例，数据以 RIFF 块的形式进入 graph 中，然后被分割成视频和音频流，视频流有一系列的压缩的视频桢组成，解压后，视频流由一系列的无压缩的位图组成，音频流也要走同样的步骤。

Media Types: How DirectShow Represents Formats

媒体类型是一种很普遍的，可以扩展的用来描述数字媒体格式的方法，当两个 filter 连接的时候，他们会就采用某一种媒体类型达成一致的协议。媒体类型定义了处于源头的 filter 将要给下游的 filter 发送什么样的数据，以及数据的 physical layout。如果两个 filter 不能够支持同一种的媒体类型，那么他们就没法连接起来。

对于大多数的应用来说，也许你不用考虑媒体类型，但是，有些应用程序中，你会直接应用到媒体类型的。

媒体类型是通过 **AM_MEDIA_TYPE** 结构定义的，看看原始定义吧

```
typedef struct    _MediaType {
        GUID          majortype;
        GUID          subtype;
        BOOL          bFixedSizeSamples;
        BOOL          bTemporalCompression;
        ULONG         lSampleSize;
        GUID          formattype;
        IUnknown     *pUnk;
        ULONG         cbFormat;
        [size_is(cbFormat)] BYTE *pbFormat;
} AM_MEDIA_TYPE;
```

*Major type:*是一个 GUID，用来定义数据的主类型，包括，音频，视频，unparsed 字节流，MIDI 数据，等等，具体可以参考 msdn。

*Subtype:*子类型，也是一个 GUID，用来进一步的细化数据格式，例如，在视频主类型中，还包括 RGB-24, RGB-32, UYVY 等等一些子类型，在音频主类型中还包括 PCM audio, MPEG-1 payload 等类型，子类型提供了比主类型更详细的信息，但是并没有定义所有的格式，例如，

视频的子类型并没有定义图像大小，桢率。这些由下面的字段定义。
bFixedSizeSamples 当这个值为 TRUE 时，表示 sample 大小固定。
bTemporalCompression 当这个值为 TRUE 时，表示 sample 采用了临时压缩格式，表明不是所有的桢都是关键桢，如果为 FALSE，表明所有的都是关键桢。
lSampleSize 表示 sample 的大小。对于压缩的数据，这个值可能为零。
Formattype 一个 GUID 值，用来表明内存块的格式。包括如下：FORMAT_None，FORMAT_DvInfo，FORMAT_MPEGVideo，FORMAT_MPEG2Video，FORMAT_VideoInfo，FORMAT_VideoInfo2，FORMAT_WaveFormatEx，GUID_NULL
pUnk 该参数没有用到
cbFormat 内存块的大小
pbFormat 指向内存块的指针，
下面我们看一段代码，看看 filter 如何检测媒体类型的。

```
HRESULT CheckMediaType(AM_MEDIA_TYPE *pmt)
{
    if (pmt == NULL) return E_POINTER;

    // Check the major type. We're looking for video.
    if (pmt->majortype != MEDIATYPE_Video)
    {
        return VFW_E_INVALIDMEDIATYPE;
    }
    // Check the subtype. We're looking for 24-bit RGB.
    if (pmt->subtype != MEDIASUBTYPE_RGB24)
    {
        return VFW_E_INVALIDMEDIATYPE;
    }
    // Check the format type and the size of the format block.
    if ((pmt->formattype == FORMAT_VideoInfo) &&
          (pmt->cbFormat >= sizeof(VIDEOINFOHEADER) &&
          (pmt->pbFormat != NULL))
    {
        // Now it's safe to coerce the format block pointer to the
        // correct structure, as defined by the formattype GUID.
        VIDEOINFOHEADER *pVIH = (VIDEOINFOHEADER*)pmt->pbFormat;
       // Examine pVIH (not shown). If it looks OK, return S_OK.
        return S_OK;
    }
    return VFW_E_INVALIDMEDIATYPE;
}
```

下面简单介绍几个和 Media Type 相关的函数
AM_MEDIA_TYPE 结构包含一个指向数据块的指针，因此，当你使用这个结构的时候，一定要小心内存分配，以防内存泄漏。

分配函数

1 AM_MEDIA_TYPE * WINAPI CreateMediaType(
AM_MEDIA_TYPE const *pSrc);

这个函数分配一个新的 AM_MEDIA_TYPE 结构，包含特定格式的数据块。释放由这个函数分配的内存，可以调用 **DeleteMediaType**. 函数

2 STDAPI CreateAudioMediaType(
const WAVEFORMATEX *pwfx,
AM_MEDIA_TYPE *pmt,
BOOL bSetFormat);

该函数利用一个给定的 WAVEFORMATIEX 结构来初始化媒体类型，如果 bsetFormat 参数为 TRUE，该函数就分配一块新的内存，如果原来的 pmt 已经包含内存，就有可能发生内存泄漏。为了避免内存泄漏，在调用这个函数前要调用 FreeMediaType（），在这个函数返回之后，再次调用 FreeMediaType（），释放 format block。

3HRESULT WINAPI CopyMediaType(
AM_MEDIA_TYPE *pmtTarget,
const AM_MEDIA_TYPE *pmtSource);

这个函数复制了一个结构到另一个结构中去。这个函数也要重新分配内存给目的结构，如果 pmtTarget,已经包含一个内存块，就要内存泄漏，因此，在调用该函数前后都要调用 FreeMediaType 函数。

释放函数

4 void WINAPI DeleteMediaType(AM_MEDIA_TYPE *pmt);

无论是采用 CoTaskMemAlloc 函数还是用 CreateMediaType 函数分配的内存都可以用这个函数来释放，如果你没有连接基类的动态库，你可以用下面的代码

```
void MyDeleteMediaType(AM_MEDIA_TYPE *pmt)
{
    if (pmt != NULL)
    {
        MyFreeMediaType(*pmt); // 见下面的 FreeMediaType 函数
        CoTaskMemFree(pmt);
    }
}
```

5 void WINAPI FreeMediaType(AM_MEDIA_TYPE& mt);

这个函数用来释放数据块的内存，如果要删除 AM_MEDIA_TYPE 结构，可以使用 DeleteMediaType 函数。

```
void MyFreeMediaType(AM_MEDIA_TYPE& mt)
{
    if (mt.cbFormat != 0)
    {
        CoTaskMemFree((PVOID)mt.pbFormat);
        mt.cbFormat = 0;
        mt.pbFormat = NULL;
    }
    if (mt.pUnk != NULL)
    {
```

```
            // Unecessary because pUnk should not be used, but safest.
            mt.pUnk->Release();
            mt.pUnk = NULL;
        }
}
```

4 媒体 Samples 和 Allocators

Filters 通过 pin 的连接来传递数据，数据流是从一个 filter 的输出 pin 流向相连的 filter 的输入 pin。输出 pin 常用的传递数据的方式是调用输入 pin 上的 **IMemInputPin::Receive** 方法。

对于 filter 来说，可以有好几种方式来分配媒体数据使用的内存块，可以在堆上分配，可以在 DirectDraw 的表面，也可以采用 GDI 共享内存，还有其他的一些方法，在 Directshow 中用来进行内存分配任务的是内存分配器（allocator），也是一个 COM 对象，暴露了一个 **IMemAllocator** 接口。

当两个 pin 连接的时候，必须有一个 pin 提供一个 allocator，Directshow 定义了一系列函数调用用来确定由哪个 pin 提供 allocator，以及 buffer 的数量和大小。

在数据流开始之前，allocator 会创建一个内存池（pool of buffer），在开始发送数据流以后，源 filter 就会将数据填充到内存池中一个空闲的 buffer 中，然后传递给下面的 filter。但是，源 filter 并不是直接将内存 buffer 的指针直接传递给下游的 filter，而是通过一个 *media samples* 的 COM 对象，这个 sample 是 allocator 创建的用来管理内存 buffer。Media sample 暴露了 **IMediaSample** 接口，一个 sample 包含了下面的内容：

一个指向没有发送的内存的指针。

一个时间戳

一些标志

媒体类型。

时间戳表明了 presentation time，Renderer filter 就是根据这个时间来安排 render 顺序的。标志是用来标示数据是否中断等等，媒体类型提供了中途改变数据格式的一种方法，不过，一般 sample 没有媒体类型，表明它们的媒体类型一直没有改变。

当一个 filter 正在使用 buffer，它就会保持一个 sample 的引用计数，allocator 通过 sample 的引用计数用来确定是否可以重新使用一个 buffer。这样就防止了 buffer 的使用冲突，当所有的 filter 都释放了对 sample 的引用，sample 才返回到 allocator 的内存池，供重新使用。

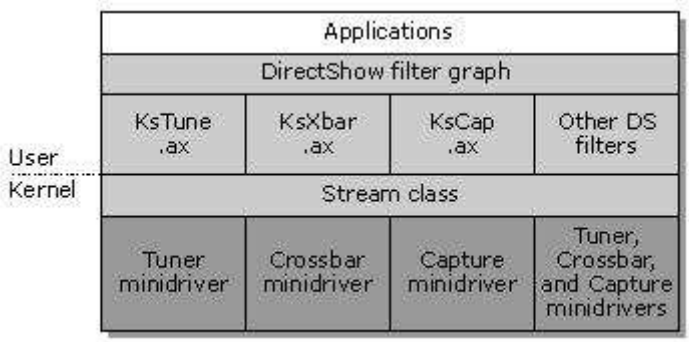
5 硬件设备在 graph 中的作用

下面的这段话借用的是陆其明的一段文档，特此标记 2005-1-26 我觉得他对硬件的表述比较清楚。

大家知道，为了提高系统的稳定性，Windows 操作系统对硬件操作进行了隔离；应用程序一般不能直接访问硬件。DirectShow Filter 工作在用户模式（User mode，操作系统特权级别为 Ring 3），而硬件工作在内核模式（Kernel mode，操作系统特权级别为 Ring 0），那么它们之间怎么协同工作呢？

DirectShow 解决的方法是，**为这些硬件设计包装 Filter；这种 Filter 能够工作在用户模式下，外观、控制方法跟普通 Filter 一样，而包装 Filter 内部完成与硬件驱动程序的交互**。这样的设计，使得编写 DirectShow 应用程序的开发人员，从为支持硬件而需做出的特殊处理中解脱出来。DirectShow 已经集成的包装 Filter，包括 **Audio Capture Filter（qcap.dll）**、**VfW Capture Filter**（qcap.dll，Filter 的 Class Id 为 CLSID_VfwCapture）、**TV Tuner Filter**（KSTVTune.ax，Filter 的 Class Id 为 CLSID_CTVTunerFilter）、Analog Video Crossbar Filter（ksxbar.ax）、**TV Audio**

Filter（Filter 的 Class Id 为 CLSID_TVAudioFilter）等；另外，DirectShow 为采用 WDM 驱动程序的硬件设计了 **KsProxy Filter**（Ksproxy.ax,）。我们来看一下结构图：



从上图中，我们可以看出，**Ksproxy.ax、Kstune.ax、Ksxbar.ax** 这些包装 Filter 跟其它普通的 DirectShow Filter 处于同一个级别，可以协同工作；**用户模式下的 Filter 通过 Stream Class 控制硬件的驱动程序 minidriver（由硬件厂商提供的实现对硬件控制功能的 DLL）**；Stream Class 和 minidriver 一起向上层提供系统底层级别的服务。值得注意的是，这里的 Stream Class 是一种驱动模型，它负责调用硬件的 minidriver；另外，Stream Class 的功能还在于协调 minidriver 之间的工作，使得一些数据可以直接在 Kernel mode 下从一个硬件传输到另一个硬件（或同一个硬件上的不同功能模块），提高了系统的工作效率。（更多的关于底层驱动程序的细节，请读者参阅 Windows DDK。）

下面，我们分别来看一下几种常见的硬件。
VfW 视频采集卡。这类硬件在市场上已经处于一种淘汰的趋势；新生产的视频采集卡一般采用 WDM 驱动模型。但是，DirectShow 为了保持向后兼容，还是专门提供了一个包装 Filter 支持这种硬件。和其他硬件的包装 Filter 一样，这种包装 Filter 的创建不是像普通 Filter 一样使用 CoCreateInstance，而要通过系统枚举，然后 BindToObject。

音频采集卡（声卡）。声卡的采集功能也是通过包装 Filter 来实现的；而且现在的声卡大部分都有混音的功能。这个 Filter 一般有几个 Input pin，每个 pin 都代表一个输入，如 Line In、Microphone、CD、MIDI 等。值得注意的是，这些 pin 代表的是声卡上的物理输入端子，在 Filter Graph 中是永远不会连接到其他 Filter 上的。声卡的输出功能，可以有两个 Filter 供选择：DirectSound Renderer Filter 和 Audio Renderer (WaveOut) Filter。注意，这两个 Filter 不是上述意义上的包装 Filter，它们能够同硬件交互，是因为它们使用了 API 函数：前者使用了 DirectSound API，后者使用了 waveOut API。这两个 Filter 的区别，还在于后者输出音频的同时不支持混音。（顺便说明一下，Video Renderer Filter 能够访问显卡，也是因为使用了 GDI、DirectDraw 或 Direct3D API。）如果你的机器上有声卡的话，你可以通过 GraphEdit，在 Audio Capture Sources 目录下看到这个声卡的包装 Filter。

WDM 驱动的硬件（包括视频捕捉卡、硬件解压卡等）。这类硬件都使用 Ksproxy.ax 这个包装 Filter。Ksproxy.ax 实现了很多功能，所以有“瑞士军刀”的美誉；它还被称作为“变色龙 Filter”，因为该 Filter 上定义了统一的接口，而接口的实现因具体的硬件驱动程序而异。在 Filter Graph 中，Ksproxy Filter 显示的名字为硬件的 Friendly name（一般在驱动程序的.inf 文件中定义）。我们可以通过 GraphEdit，在 WDM Streaming 开头的目录中找到本机系统中安装的 WDM 硬件。因为 KsProxy.ax 能够代表各种 WDM 的音视频设备，所以这个包装 Filter 的工作流程有点复杂。这个 Filter 不会预先知道要代表哪种类型的设备，它必须首先访问驱动程序的属性集，

然后动态配置 Filter 上应该实现的接口。当 Ksproxy Filter 上的接口方法被应用程序或其他 Filter 调用时，它会将调用方法以及参数传递给驱动程序，由驱动程序最终完成指定功能。除此以外，WDM 硬件还支持内核流（Kernel Streaming），即内核模式下的数据传输，而无需经过到用户模式的转换。因为内核模式与用户模式之间的相互转换，需要花费很大的计算量。如果使用内核流，不仅可以避免大量的计算，还避免了内核数据与主机内存之间的拷贝过程。在这种情况下，用户模式的 Filter Graph 中，即使 pin 之间是连接的，也不会有实际的数据流动。典型的情况，如带有 Video Port Pin 的视频捕捉卡，Preview 时显示的图像就是在内核模式下直接传送到显卡的显存的。所以，你也休想在 VP Pin 后面截获数据流。

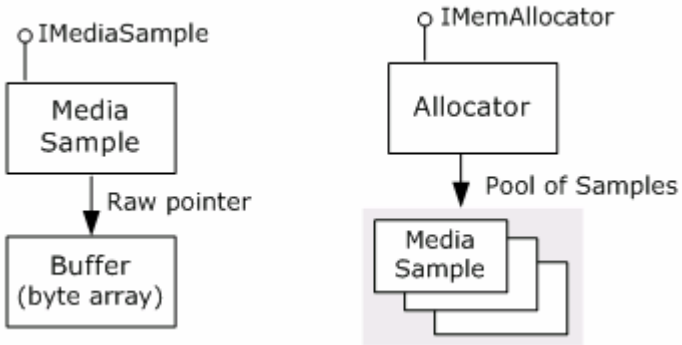
讲到这里，我想大家应该对 DirectShow 对硬件的支持问题有了一个总体的认识。对于应用程序开发人员来说，这方面的内容不用研究得太透，而只需作为背景知识了解一下就好了。其实，大量繁琐的工作 DirectShow 已经帮我们做好了。

1.5 构建一个 Filter Graph 图

1.6 数据流在 Filter Graph 里的流动（Data Flow）

1 directshow 数据流动概述

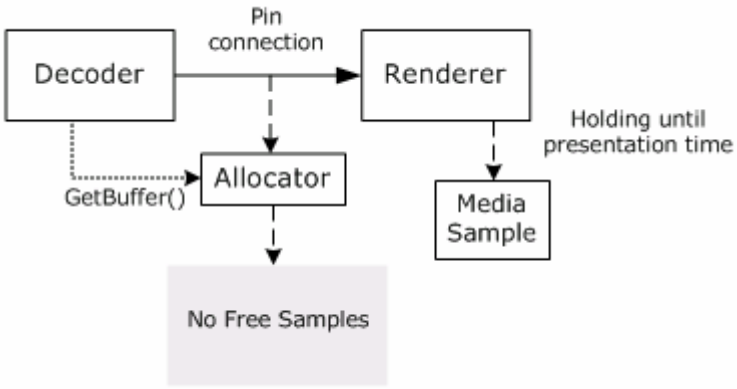
数据总是存在内存块中的字节集合，每个 buffer 都被封装在一个叫做 media sample 的 com 组件，它引出了 IMediaSample 接口。这个 sample 一般都有一个叫做内存分配器（alloctor）的 com 对象来创建，这个对象具有 IMemAllocator 接口。每一个 pin 之间的连接都要指定一个 allocator，有时也有几个连接同用一个 allocator。



每一个 allocator 都要创建一个 media sample 池，并且给每一个 sample 分配一个内存 buffer。每当一个 **Filter** 需要一个 buffer 来填充数据，它就通过 allocator 的函数 **IMemAllocator::GetBuffer**.来获得一个 sample。如果分配器 allocator 正好有空闲的 sample，GetBuffer 立即返回一个指向该 sample 的指针。如果没有空闲的 sample，该方法就阻塞，直到有一个 sample 可用为止。当该函数返回一个 sample 时，Filter 就将数据填充到 buffer 里，设

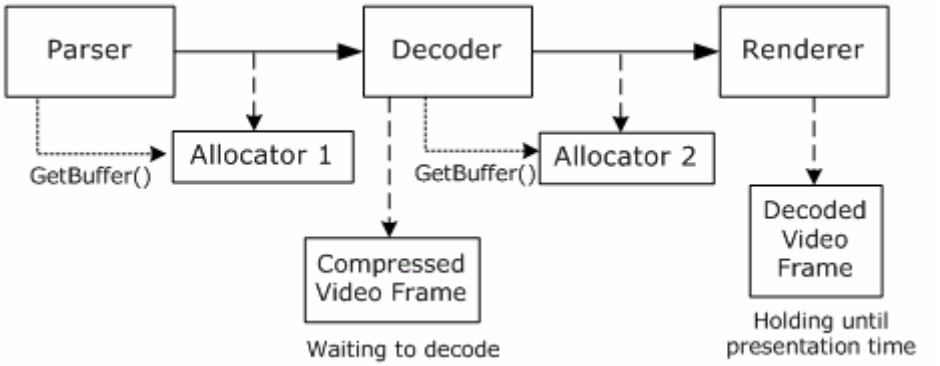
置好标识，然后就将 sample 传递给下一个 Filter。

当一个 renderfilter 接收到一个 sample 时，它就检查该 sample 的时间戳，直到 Fliter Graph 的参考时钟表明该数据可用播放，Filter 就开始播放该数据。当数据播放完毕，Filter 释放 sample，直到所有的 Filter 都释放对该 sample 的引用，该 sample 的引用计数为 0 时，这个 sample 才返回到 sample 池。



有时也许数据流的上游对 buffer 的填充比播放要快，即使这样，render Filter 也要按照时间戳播放数据，这样 sample 池中的 sample 数量就少，从而填充的速度减慢。

上面描述了在流中只有一个 allocator 的情景，实际上，在每条数据流中总是有好几个 allocator，当一个 sample 被释放的时候，也许此时有好几个 allocator 都在等着该 sample，这就有新的问题了，也许有的 alloctor 永远都不能被分配 sample，陷入互锁状态。下面的图就演示了这种情形，Decoder 有数据需要压缩，因此它在等待 Renderer 释放 sample，但是，Parser 也在请求 sample，它在等待 decoder 释放 sample。



具体参加 help

2 传输（Transports）

为了在过滤器图表中传送媒体数据，DirectShow 过滤器需要支持一些协议，称之为传输协议（transport）。相连的过滤器必须支持同样的传输协议，否则不能交换媒体数据。

大多数的 DirectShow 过滤器把媒体数据保存在主存储器中，并通过引脚把数据提交给其它的过滤器，这种传输称为局部存储器传输（local memory transport）。虽然局部存储器传输在 DirectShow 中最常用，但并不是所有的过滤器都使用它。例如，有些过滤器通过硬件传送媒体数据，引脚只是用来提交控制信息，如 IOverlay 接口。

DirectShow 为局部存储器传输定义了两种机制：推模式（push model）和拉模式（pull model）。在推模式中，源过滤器生成数据并提交给下一级过滤器。下一级过滤器被动的接收数据，完成处理后再传送给再下一级过滤器。在拉模式中，源过滤器与一个分析过滤器相连。分析过

滤器向源过滤器请求数据后，源过滤器才传送数据以响应请求。推模式使用的是 **IMemInputPin 接口，拉模式使用 IAsyncReader** 接口，推模式比拉模式要更常用。

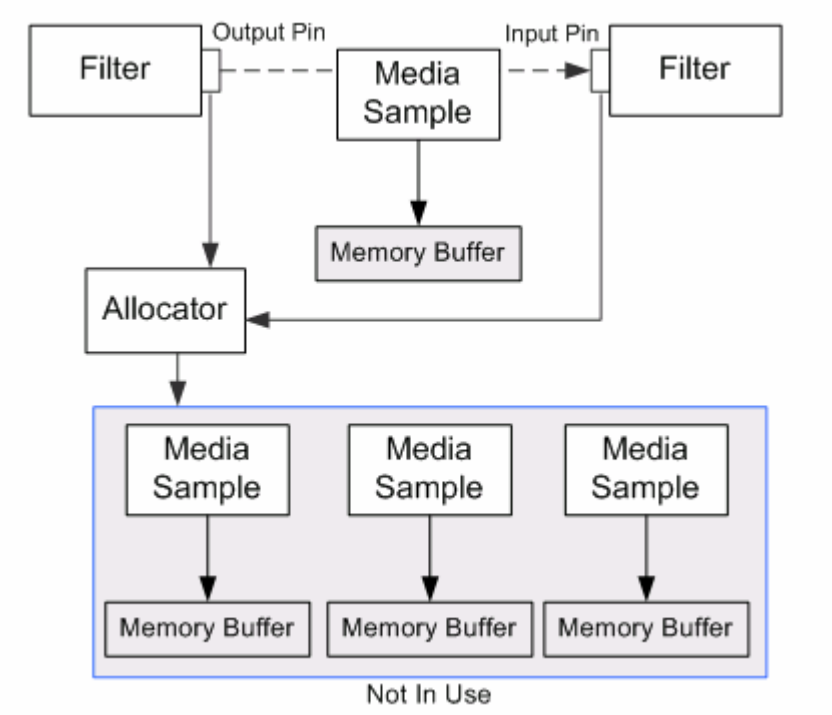
3 Samples 和 Allocators

当一个 pin 向另一个 pin 传递数据的时候，它并不是直接将内存块的指针传递下一个 pin，实际上，它将传递一个管理内存的 com 对象的指针给下一个 pin。这个 com 对象称为 media sample。暴露了 **IMediaSample** 接口。接收 pin 通过调用 IMediaSample 的方法来对内存进行操作，比如方法 **IMediaSample::GetSize**, **IMediaSample::GetActualDataLength** 以及 **IMediaSample::GetPointer**。

Sample 一般都是从源 filter 开始，通过输出 pin 传递到下一个 filter 的输入 pin，一路传递下去一直到 render filter。在拉模式中，输出 pin 通过调用输入 pin 上的 **IMemInputPin::Receive** 方法传递 sample，输入 pin 或者在 Receive 函数同步处理数据，或者另外采用一个工作线程异步出来的方式。如果在 Recive 方法中需要等待资源的话，也可以阻塞。

另外一个 com 对象，叫做 allocator，用来创建和管理 sample 的。暴露了 **IMemAllocator** 接口。当一个 filter 需要一个空的 buffer 的时候，就可以调用 **IMemAllocator::GetBuffer**，该方法返回一个指向 sample 的指针。每一个 pin 连接都共享一个 allocator，当两个 pin 连接的时候，他们会决定由哪个 filter 来提供 allocator，通过 pin 还可以设置 allocator 的属性，例如，buffer 的数量和大小。

下面的图表显示了 allocator，sample 和 filter 之间的关系。



Media sample 引用计数

Allocator 创建了一个 sample 池。因此 ，当某个 Filter 调用 GetBuffer 函数时，一些 sample 被使用，其他空闲的 sample 可以响应。Allocator 通过引用计数来跟踪 samples。Filter 调用 Getbuffer 返回的 sample 的引用计数是 1。当 sample 的引用计数为 0 时，sample 就返回内存池，成为空闲的 sample，可以再次响应 Getbuffer 的调用。如果所有的 sample 都处于繁忙状态，Getbuffer 就会阻塞，直到有一个 sample 空闲。

例如，假设一个输入 pin 接到一个 sample，如果它在 Receive 方法里同步的处理这个 sample，没有增加该 sample 的引用计数，等到 Receive 返回后，输出 pin 就释放这个 sample，引用计数为 0，sample 就返回到内存池中。 如果输入 pin 的线程还要处理该 sample，引用计数增加 1，成为 2，输出 pin 返回，释放，计数成 1。

当一个输入 pin 接收一个 sample 时，它可以将数据复制到另一个 sample 中，也可以将这个 sample 传递到下一个 Filter。一个 sample 可以流遍整个 filter graph。不过引用计数要保持大于 0。 当一个输出 pin 调用了 Release 以后，就不应该再次使用该 sample，因为也许下游还有 filter 正在使用该 sample。输出 pin 必须调用 GetBuffer 获取新的 sample。

这种机制减少 了内存分配的，因为 buffer 可以重用。也防止了数据没有被处理的 sample 被重新写入。

当一个 Filter 创建一个 allocator 的时候，allocator 还没有保留任何的内存，如果这个时候有人 Getbuffer，就会失败。只有当数据流开始的时候，输出 pin 调用 **IMemAllocator::Commit**，提交 allocator，现在才能分配内存。

当数据流停止的时候，pin 就调用 **IMemAllocator::Decommit**,来销毁 allocator。在 allocator 再次 committ 之前，所有调用 GetBuffer 方法都会失败。当然，如果有一个 GetBuffer 阻塞调用在等待 sample 的时候，遇到 Decommit 方法，会立即返回一个错误码。

4 Filter 的状态

Filter 有三种状态，停止，暂停，运行。

过滤器图表管理器 控制着 Filter 的所有状态的转换。当应用程序调用 **IMediaControl::Run**, **IMediaControl::Pause**, or **IMediaControl::Stop** 时， 过滤器图表管理器就调用 Filter 相应的 **IMediaFilter** 方法。停止，运行状态的切换总是要经过暂停，因此，当一个应用程序对一个停止的 Graph 调用 RUN 命令时，**过滤器图表管理器** 在 run 之前首先要暂停。

对于大多数的 filter 来说，running 和 paused 状态是一样的。看下面的 Graph

Source > Transform > Renderer

当一个 Filter 停止时，它拒绝发送给它的任何 samples，源 filter 关闭他们的 stream 线程，其他 filter 也关闭他们创建的其他线程，pin decommit 他们的内存分配器。

过滤器图表管理器按照逆流的方向来切换 Filter 的状态，从 Renderer Filter 到源 filter，这种方式可以防止死锁。最关键的状态切换是暂停和停止之间。

从停止到暂停，当 filter 暂停时，它就做好了接收 sample 的准备，源 filter 是最后一个切换到暂停的。它开始创建 streaming 线程，发送 sample，因为下游的 filter 的状态都已经切换到暂停了，所以，所有的 filter 都可以接收 sample。只有当所有的 flter 都接收到 sample，过滤器图表管理器才算完成了状态的切换

从暂停到停止。当一个 filter 停止时，它要释放它拥有的所有的 samples。当图表管理器试图停掉上游的一个 filter 时，这个 filter 不会阻塞在 Getbuffer 和 receive 方法里，它会立即响应 stop 命令。上游的 filter 也许在执行 stop 命令前还会讲少量的 sample 传递下去，但是下游的 filter 会拒绝的，因为他们已经停止了。

5 拉 PULL 模式

在 **IMemInputPin 接口中**，上游的 flter 决定了发送什么样的数据，然后将数据推给下游的 filter。但在另外的场合，拉模式更适合。下游的 filter 向上游的 filter 请求数据，数据依然是从上游到下游，从输出 pin 到输入 pin，但是下游的 filter 主导着数据的流动。这种类型的连接采用的是 **IAsyncReader** 接口

拉模式的典型应用的文件的回放，例如在一个 AVI 文件的回放 graph 中，Async File Source Filter 就担负着从文件中读取数据，然后将数据以字节流的方式发送给下面的 filter。

1.7 事件通知机制（Event Notification）

1 概述

当某个事件发生时，比如数据流结束，产生一个错误等，Filter 就给 Filter 图表管理器发送一个事件通知。Filter 图表管理器处理其中的一部分事件，另一部分交给应用程序处理。如果图表管理器没有处理一个 filter 事件，它就把事件通知放入到一个队列中，图表管理器也可以将自己的事件通知放进队列中。

应用程序可以自己处理队列中的事件，dshow 中的事件通知就和 windows 的消息机制差不多，filter，图表管理器和应用程序通过这种机制就可以互相通信。

2 Retrieving Events

Filter 图表管理器暴露了三个接口用来处理事件通知

IMediaEventSink Filter 用这个接口来 post 事件。

IMediaEvent 应用程序利用这个接口来从队列中查询消息

IMediaEventEx 是 imediaevent 的扩展。

Filter 都是通过调用图表管理器的 **IMediaEventSink::Notify** 方法来通知图表管理器某种事件发生。事件通知包括一个事件 code，这个 code 不仅仅代表了事件的类型，还包含两个 DWORD 类型的参数用来传递一些其他的信息。

关于事件 code 的内容，在下面的一个专题中列出，这里暂略，使用时可以参考帮助。

应用程序通过调用图表管理器的 **IMediaEvent::GetEvent** 方法来从事件队列中获取事件。如果有事件发生，该函数就返回一个事件码和两个参数，如果没有事件，则一直阻塞直到有事件发生和超过某个时间。调用 GetEvent 函数后，应用程序必须调用 **IMediaEvent::FreeEventParams** 来释放事件码所带参数的资源。例如，某个参数可能是由 filter graph 分配的内存。

下面的代码演示了如何从事件队列中提取事件

```
long evCode, param1, param2;
HRESULT hr;
while (hr = pEvent->GetEvent(&evCode, &param1, &param2, 0), SUCCEEDED(hr))
{
    switch(evCode)
    {
        // Call application-defined functions for each
        // type of event that you want to handle.
    }
    hr = pEvent->FreeEventParams(evCode, param1, param2);
}
```

为了重载 Filter 图表管理器对事件的缺省处理，你可以使用某个事件码做参数调用 **IMediaEvent::CancelDefaultHandling**，这样就可以屏蔽图表管理器对某个事件码的处理了。如果要恢复图表管理器对该事件码的缺省处理，可以调用 **IMediaEvent::RestoreDefaultHandling**。如果图表管理器对某个事件码没有缺省的处理，调用这两个函数是不起作用的。

3 事件是如何发生的

为了处理事件，应用程序需要一种机制来获取正在队列中等待的事件。Filter 图表管理器提供了两种方法。

1 窗口通知，图表管理器发送开发者自己定义的窗口消息

2 事件信号 如果队列中有 dshow 事件，就用事件信号通知应用程序，如果队列为空就重新设置事件信号。

下面的代码演示了如何利用消息通知

```
#define WM_GRAPHNOTIFY WM_APP + 1      // Private message.
pEvent->SetNotifyWindow((OAHWND)g_hwnd, WM_GRAPHNOTIFY, 0);
```

然后在窗口消息处理过程中处理该消息如下

```
LRESULT CALLBACK WindowProc( HWND hwnd, UINT msg, UINT wParam, LONG lParam)
{
    switch (msg)
    {
        case WM_GRAPHNOTIFY:
            HandleEvent();   // Application-defined function.
            break;
        // Handle other Windows messages here too.
    }
    return (DefWindowProc(hwnd, msg, wParam, lParam));
}
```

由于事件通知和窗口的消息循环都是异步的，因此，当你的应用程序处理消息的时候，队列中或许有 N 个事件等待处理。因此，在你调用 GetEvent 的时候，一定要循环调用，直到返回一个错误码，这表明队列是空的。

当你释放 **IMediaEventEx** 指针时，你可以调用 SetNotifyWindow 来取消事件通知，记住此时要给这个函数传递一个 NULL 指针。在你的事件处理程序中，在调用 GetEvent 之前一定要检查 **IMediaEventEx** 指针是否为空，这样就可以避免错误。

下面看看采取事件信号的通知方式。

在 Filter 图表管理器里有一个手动设置的 Event 内核对象，用来反映事件队列的状态。如果队列中有等待处理的事件，event 就处于通知状态，如果队列是空的，**IMediaEvent::GetEvent 函数**调用就会重置该 event 对象。

应用程序可以调用 **IMediaEvent::GetEventHandle** 获得 event 内核对象的句柄，然后就可以调用 **WaitForMultipleObjects** 来等待事件的发生，如果 event 被通知了，就可以调用 **IMediaEvent::GetEvent** 来获得 dshow 的事件。

下面的代码演示了如何利用 event 内核对象来获取 EC_COMPLETE 事件，

```
HANDLE    hEvent;
long      evCode, param1, param2;
BOOLEAN bDone = FALSE;
HRESULT hr = S_OK;
hr = pEvent->GetEventHandle((OAEVENT*)&hEvent);
if (FAILED(hr)
{
    /* Insert failure-handling code here. */
```

```
}
while(!bDone)
{
    if (WAIT_OBJECT_0 == WaitForSingleObject(hEvent, 100))
    {
        while (hr = pEvent->GetEvent(&evCode, &param1, &param2, 0), SUCCEEDED(hr))
        {
            printf("Event code: %#04x\n Params: %d, %d\n", evCode, param1, param2);
            pEvent->FreeEventParams(evCode, param1, param2);
          switch (evCode)
         {
          case EC_COMPLETE:    // Fall through.
          case EC_USERABORT: // Fall through.
         case EC_ERRORABORT:
            CleanUp();
            PostQuitMessage(0);
            return;
        }
    }
}
```

事件通知码（Event Notification Codes）

暂略，需要补充

1.8Directshow 中的时钟（Time and Clocks in Dshow）

1 参考时钟:

参考时钟是 Filter 用来同步 Filter 图表管理器中的所有的 Filter 的。

任何一个引出 **IReferenceClock** 接口的对象都可以作为参考时钟。参考时钟可以是 Filter 提供，例如声卡就可以提供一个硬件的时钟。当然，可靠的时钟就是采用系统的时间。

名义上，参考时钟的精确度在 100 纳秒，但实际上，没有那么精确。调用 **IReferenceClock::GetTime** 可以获取时钟的当前时间。

尽管时钟的精确性还有所变动，但是 GetTime 方法返回的保证时间是增加的。也就是说，时钟不会倒退回去，比如，对硬件时钟进行了调整，GetTime 方法就返回上次的时间。

缺省的参考时钟

当 Graph 运行的时候，Filter 图表管理器会自动选择一个参考时钟的，选择时钟的法则如下

1 如果应用程序选择了时钟，就采用应用程序选择的时钟

2 如果 Graph 包含一个活动的源 Filter，这个 filter 有 IReferenceClock 接口，那么就用这个时钟。

3 如果 Graph 中不含有任何活动的源 Filter，就选用 graph 中任何暴露 IReferenceClock 接口的 Filter，选择的方法是从 Renderers 逆流向上，连接的 filter 优先，没有连接的 filter 次之。

4 如果没有任何 filter 符合条件，就采用系统参考时钟 System Reference Clock

设置参考时钟

如果你想为 graph 设置新的时钟，应用程序可以调用图表管理器的接口

IMediaFilter::SetSyncSource 方法来选择一个参考时钟。

如果你给 SetSyncSource 传递的参数为 NULL，Graph 就不设置任何的参考时钟了。如果想恢复缺省的时钟，调用 **IFilterGraph::SetDefaultSyncSource** 方法。

当 graph 的参考时钟改变时，Graph 通过 the reference clock changes, the Filter Graph Manager notifies each filter by calling its **IMediaFilter::SetSyncSource** 通知所有的 Filter。

2 Clock Times

Directshow 定义了两个相关的时间，参考时钟和 数据流时间

参考时间是参考时钟的绝对时间

数据流时间和 graph 开始的时间有关。

当 graph 正在运行，流时间就等于从开始时间计数的时间，当 graph 暂停，流时间就等于它暂停开始的时间，当 graph 停止时，流时间不确定。

当一个 sample 具有时间戳 t，就意味着这个 sample 应该在流时间 t 播放，因此，流时间也叫播放时间。

当应用程序通过 **IMediaControl::Run** 来运行 graph 时，在 graph 内部也调用了 **IMediaFilter::Run**

3 时间戳

时间戳采用的是流时间，它在 sample 标上开始和结束时间。时间戳也叫播放时间，通过后面的知识你会了解到，并不是所有格式的数据流都采用同一种样式的时间戳。

当 renderer Filter 接收到 sample，它会根据 sample 的时间戳进行排序，等到该 sample 的播放时间到了，就开始播放该 sample，从到达到开始播放的时间，可以通过 **IReferenceClock::AdviseTime** 获得。如果 sample 来晚了，或者 sample 没有时间戳，filter 就立即播放 sample。

源 Filter 通过下面的规则来负责给 sample 设置时间戳

1 文件重播

第一个 sample 的时间戳为 0，随后的时间戳根据 sample 的大小和播放的速率来确定。这都由分解文件的 Filter 来计算和确定，例如 AVI Splitter

2 视频和音频的捕捉

所有的 sample 在捕捉的时候就被打上时间戳了，时间等于流时间

3 混和 Filter

根据输出数据流的格式，混和 filter 也许需要时间戳，也许不需要，

可以通过调用 **IMediaSample::SetTime** 来给 sample 设置时间

4 活动的源 Live Source

活动的源 Filter，就是推模式的源，实时的接收数据。视频捕捉和网络广播就是例子，活动得源无法控制数据流得速率。

下面的 Filter 通常被认为是活动的源 filter

Filter 调用 **IAMFilterMiscFlags::GetMiscFlags** 方法时返回 AM_FILTER_MISC_FLAGS_IS_SOURCE，并且至少有一个输出 pin 暴露了 **IAMPushSource** 接口

2 Filter 暴露 **IKsPropertySet** 接口，并且有个捕捉 pin（PIN_CATEGORY_CAPTURE）

如果活动的源能够提供参考时钟，那么 Graph 首先采用。

反应时间（Latency）

过滤器（filter）的反映时间就是 Filter 处理 sample 所花费的时间。对于活动的源 filter，反应时间由容纳 sample 的内存大小决定。例如，假设一个 filter 有一个视频源具有 33ms 的反应时间，一个音频源有 500ms 的反应时间，每一个视频祯（video frame）比相应的音频 sample 要

早 470ms，除非 graph 进行补偿，否则视频和音频是不同步的。

活动的源可以通过 **IAMPushSource** 接口来进行同步。除非应用程序调用 **IAMGraphStreams::SyncUsingStreamOffset** 方法对源进行同步，一般来说 Filter 图表管理器是不会对源进行同步的。图表管理器对源进行同步时，向各个源 filter 查询 **IAMPushSource** 接口，如果 filter 支持 **IAMPushSource** ，图表管理器就会通过 **IAMLatency::GetLatency** 来得到 filter 期望的反应时间。注：**IAMPushSource** 继承与 **IAMLatency**。通过组合这些反应值，filter 图表管理器 Graph 最大反应时间，然后调用 **IAMPushSource::SetStreamOffset** 给每个源 filter 设置一个数据流偏移时间，当 filter 给它产生的 sample 打时间戳的时候，要加上偏移时间的。

现在，VFW Capture filter 和 Audio Capture filter.都支持 IAMPushSource 接口。

速率匹配（Rate Matching）

当 renderer Filter 利用参考时钟安排播放顺序的时候，如果源 filter 采用另一种时钟，在重放的时候就会发生故障。播放的速度大于源产生的速度，就会产生间隙停顿，或者播放速度小于源的产生速度，就会形成数据的堆积，造成内存出错。源一般来说是无法控制数据的产生速度的，因此，播放速度要随着源的速度改变而改变。

现在，只有在音频播放 filter 才能进行速率匹配，因为音频中的 glitches 比视频中的更容易捕捉到。为了匹配音频播放速率，要注意下列事情。

1 如果 graph 没有使用参考时钟，没法进行速率匹配。

2 上游要有一个活动的源

3 源 filter 的输出 pin 要支持 **IAMPushSource** 接口，当请求 **IAMPushSource::GetPushSourceFlags**.要返回下面的值

AM_PUSHSOURCECAPS_INTERNAL_RM

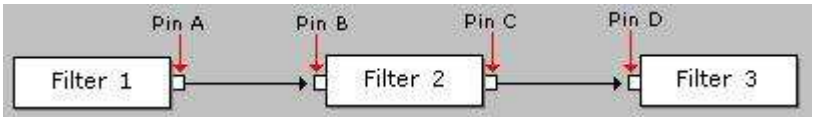
AM_PUSHSOURCECAPS_NOT_LIVE

AM_PUSHSOURCECAPS_PRIVATE_CLOCK

4 如果 **GetPushSourceFlags** 返回 0，播放 filter 就根据 graph 时钟和 sample 的时间戳来自己决定播放速率

1.9 动态删除或增加 Filter（Dynamic Graph Building）

在进行 pin 连接的时候，应用程序一般都要讲 graph 停掉。但是，一些 filter 支持 pin 的动态连接。



如上图，我们想将 Filter 2 动态移走。有两个必要条件：（1）**Filter 3 (pin D)必须支持 IPinConnection 接口（这个接口能够保证 Filter 在非 Stopped 状态下也能进行 Pin 的重连）；（2）Filter 在重连的时候不允许数据的传输，所以要将数据线程阻塞。如果“重连”是由 Filter 1 发起的（在 Filter 内部完成），那么 Filter 1 要同步这个数据发送线程；如果“重连”由应用程序来完成，则要求 Filter 1 (pin A)实现 IPinFlowControl 接口。**

动态重连的一般步骤如下：

（1）在 Filter 1(pin A)上阻塞数据流线程。

IPinFlowControl::Block 可以工作在同步和异步两种模式下。不要在应用程序主线程下使用该 Block 函数的同步模式，因为这样可能会引起线程死锁。要么另外使用一个 worker thread，要

么使用 Block 函数的异步模式。如下：

```
// Create an event
HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
if (hEvent != NULL)
{
   // Block the data flow.
   hr = pFlowControl->Block(AM_PIN_FLOW_CONTROL_BLOCK, hEvent);
   if (SUCCEEDED(hr))
   {
      // Wait for the pin to finish.
      DWORD dwRes = WaitForSingleObject(hEvent, dwMilliseconds);
   }
}
```

（2）重连 Pin A 和 Pin D，必要时插入新的 Filter。

这里 Pin 的重连可以使用 IGraphConfig::Reconnect 或 IGraphConfig::Reconfigure（IGraphConfig 接口可以从 Filter Graph Manager 上获得）。Reconnect 比 Reconfigure 使用起来要简单，它主要作如下几件事：将 Filter 2 置于 Stopped 状态，然后将其移走；加入新的 Filter；重新连接相关的各个 Pin；将新加入的 Filter 置于 Paused 或 Running 状态，以使其与 Filter Graph 同步。示例如下：

```
pGraph->AddFilter(pNewFilter, L"New Filter for the Graph");
pConfig->Reconnect(
   pPinA,     // Reconnect this output pin...
   pPinD,     // ... to this input pin.
   pMediaType, // Use this media type.
   pNewFilter, // Connect them through this filter.
   NULL,
   0);
```

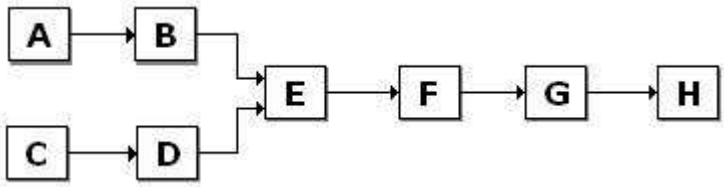
实际应用中，如果你觉得 Reconnect 不够灵活，还可以改用 Reconfigure。使用 Reconfigure 方法，你必须在你的应用程序里实现 IGraphConfigCallback 接口；在 Reconfigure 调用之前，还必须依次调用 Filter 3(pin D)上的 IPinConnection::NotifyEndOfStream 和 Filter 2(pin B)上的 IPin::EndOfStream，以使得还没处理完的数据全部发送下去（这些处理 IGraphConfig::Reconnect 会自动给我们完成）。

（3）再次启动 Filter 1(pin A)上的数据发送线程。

只需调用 IPinFlowControl::Block，如下：pFlowControl->Block(0, NULL);

过滤器链（Filter Chains）

首先要弄明白什么是 Filter Chain。见下图：



1 Filter Chain 是相互连接着的一条 Filter 链路，并且链路中的每个 Filter 至多有一个 Input pin，至多有一个 Output pin；

2 这条 Filter 链路中的数据流不依赖于链路外的其他 Filter。
如上图中，A－B，C－D，F－G－H，F－G，G－H 都是 Filter Chain，同时 Filter 链也可以只包括一个 filter，因此 A,B,C,D,E,F,G 也都是独立的链，因为 E 含有两个输入 pin，因此任何含有 E 的都不是 Filter Chain。
Filter Chain 通过 IFilterChain 接口来操作的，该接口可以从 Filter Graph Manager 上获得。
IFilterChain 提供了下面的方法用来操作 filter 链。
IFilterChain::StartChain 开始一个链条
IFilterChain::StopChain 停止一个链条
IFilterChain::PauseChain 暂停一个链条
IFilterChain::RemoveChain 将一个链条从 graph 中删除
并没有一个特殊的方法用来添加一个 chain，它和正常的添加 filter 的方法一样，首先用 **IFilterGraph::AddFilter** 在 graph 中添加一个 filter，然后就是 **IGraphBuilder::Connect**, **IGraphBuilder::Render** 等诸如此类的方法。

当 Graph 在运行的时候，Filter Chain 可以在 Running 和 Stopped 状态之间切换；当 Graph 在暂停状态下，Filter Chain 可以在 Paused 和 Stopper 状态之间切换。以上是 Filter Chain 仅有的两种状态转换。

Filter Chain 的使用规则

当你使用 IFilterChain 的方法时，你一定要确保 graph 中的 filter 都支持这个接口，否则的话你也可能会造成死锁和 graph 错误。
下面将教给你如何正确使用 filter chain
1 在链条的状态改变之前，在链条边界的数据处理必须完成。下面的函数可以完成这些事情：**IMemInputPin::Receive**, **IPin::NewSegment**, and **IPin::EndOfStream**。
Filters in the chain must return from calls to these methods made by filters outside the chain; and filters outside the chain must return from calls made by filters within the chain.
例如：
2 链中的所有的 filter 必须对链条的状态改变做出反应
3 只有当 Filter 支持动态断开的时候才能删除有个链条。

1.10 Plug-in Distributors

2 Directshow 的应用（Using Directshow）

2.1 在 GraphEdit 中模拟构建 Graph（Simulating Graph Building）

1 GraphEdit 概述

GraphEdit 是一个很有用的工具，可以用来构建 graph 图。通过 GraphEdit，你可以在开发代码之前进行一下体验，你也可以装载一个你的应用程序创建的 Graph 文件。如果你想开发一个自己的 filter，GraphEdit 给你提供了一个快速测试的方法：将你的 filter 添加到 graph 中，然后运行 graph。如果你是一个 Directshow 的初学者，那么通过 GraphEdit 你可以熟悉 Filter

和 Dshow 的特性。

下面图表演示了 GraphEdit 如何构建了一个简单的 graph。

图 1

每一个矩形代表一个 Filter，每一个 filter 的边上的小矩形代表了 pin，输入 pin 在 filter 的左边，输出 pin 在 Filter 的右边，箭头代表两个 pin 的连接方向。

通过 GraphEdit，你可以做到下面的事情：

1 可视化的创建一个 Graph，可以动态的拖拉来调整 filter。

2 可以模拟如何构建一个 graph。

3 运行，停止，暂停，see 一个 graph。

4 可以看看你的机器上都注册了那些 filter，以及这些 filter 的信息

5 查看 filter 的属性页

6 查看 pin 连接时采用的媒体类型。

2 使用 GraphEdit

如果你安装了 DirectX 的 SDK，GraphEdit 就会出现在你的开始菜单中找到 GraphEdit，启动它。如下图

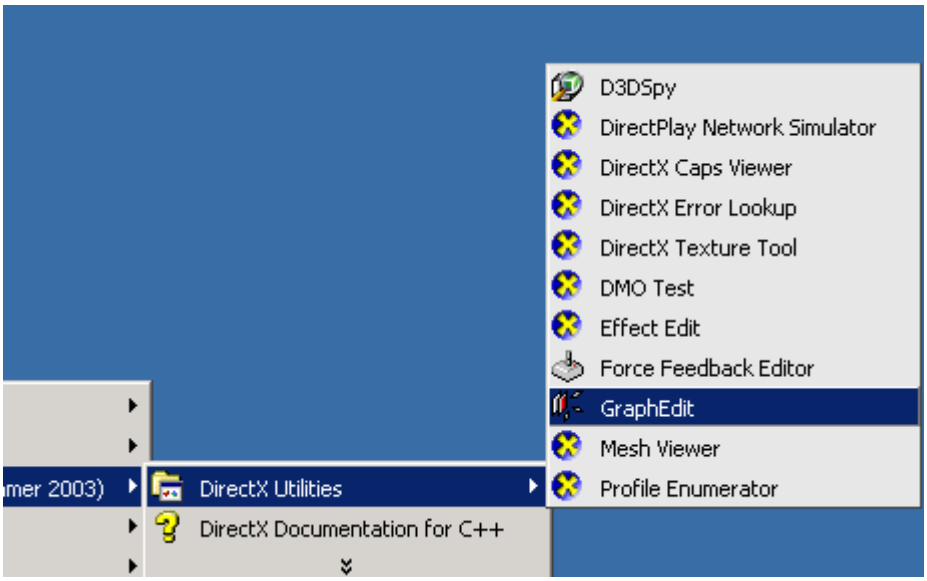


图 2

构建一个文件回放的 Graph

GraphEdit 可以自动的构建一个文件回放 Graph。这个特性其实类似于在应用程序中调用 **IGraphBuilder::RenderFile** 方法。从文件菜单中，选择 **Render Media File**，然后出现一个文件选择对话框，选择一个多媒体文件后单击打开，GraphEdit 会自动地建立一个 Filter Graph 来播放你选择的文件。

你也可以播放一个网络上媒体文件，从文件菜单中，选择 **Render URL**，也会出现一个选择 URL 的对话框。其他同上。

构建一个普通的 Graph 图

使用你机器上注册的 filter，GraphEdit 可以构建一个普通的 Filter graph，从 **Graph** 菜单中，选择 **Insert Filters**，会出现一个对话框，如下图：

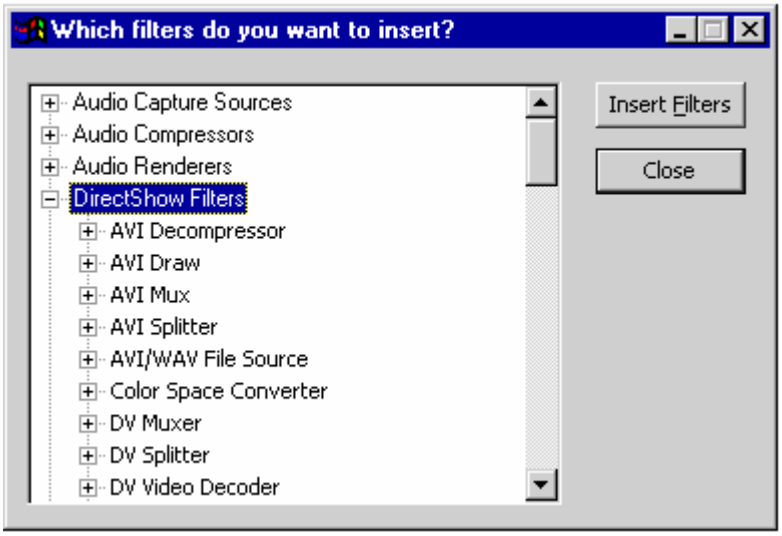


图 3

在这个对话框中列出了所有在你机器上注册的 Filter 的信息。选择 filter 的名字，然后单击 **Insert Filters** 按钮，或者双击 filter 的名字，filter 就会自动添加到 graph 中，添加完 filter 以后，你就拖动鼠标，将一个 Filter 的输出 pin 和另一个 Fiter 的输入 pin 连接起来。如果 pin 接受这个连接，GraphEdite 就会用一个带箭头的

下面的图是一个捕捉桌面的 graph 图

图 4

Run the Graph

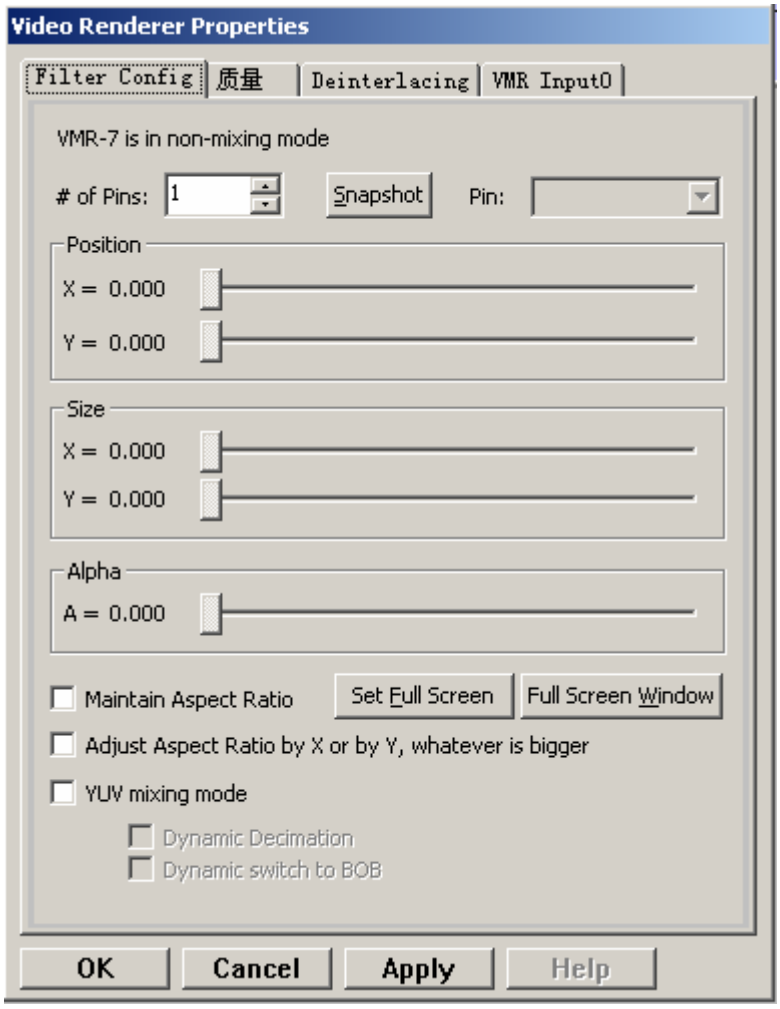
当你在 GraphEdit 中构建好一个 Filter graph 的时候，你可以让你的 graph 运行一下看是否和你期望的一样。Graph 菜单中包含了 Play，Pause，和 Stop 命令，这些命令会触发 **IMediaControl** 接口的 Run, Pause, and Stop，GraphEdit 的工具栏也有代表这三个命令的按钮，见下图，单击第一个按钮就开始运行你的 Graph 图了

图 5

注：GraphEdit 的 Stop 命令首先会暂停 Graph，然后 Seek 到时间的零点（我们坚定 graph 是可 Seek 的）。对于文件的回放，这个命令会将视频窗口的图像设置为第一桢，然后 GraphEdit 才调用 **IMediaControl::Stop**.

查看属性 **View Property Pages**

一些 Filter 提供了属性页可以让用户设置 Filter 的属性。鼠标右键单击 filter，在弹出的菜单上选择 Properties，就会弹出 Filter 的属性页设置对话框，用户可以从这里设置属性。



3 Loading a Graph From an External Process

GraphEdit 可以加载其他进程创建的 filter Graph，利用这个特性，只使用少量的代码，你可以清楚地看到你的应用程序创建的所有的 filter Graph。

这个特性只有 win2000，XP 才支持。

应用程序首先必须在 Running Object Table (ROT).中注册一个 filter graph 的实例。ROT 是一个全局的对象表，用来查看所有正在运行的对象。对象都是通过 moniker 注册到 rot 上。Graph Edit 通过搜索 ROT 中和指定名字 moniker 就 ok。

!FilterGraph X pid Y

这里，x 是 Filter Graph Manager 的地址，y 是进程 ID，也是 16 进制。

当你的应用程序创建 filter graph 的时候，调用下面的代码：

```
HRESULT AddToRot(IUnknown *pUnkGraph, DWORD *pdwRegister)
{
    IMoniker * pMoniker;
    IRunningObjectTable *pROT;
    if (FAILED(GetRunningObjectTable(0, &pROT))) {
        return E_FAIL;
    }
    WCHAR wsz[256];
```

```
    wsprintfW(wsz, L"FilterGraph %08x pid %08x", (DWORD_PTR)pUnkGraph,
GetCurrentProcessId());
    HRESULT hr = CreateItemMoniker(L"!", wsz, &pMoniker);
    if (SUCCEEDED(hr)) {
        hr = pROT->Register(ROTFLAGS_REGISTRATIONKEEPSALIVE, pUnkGraph,
            pMoniker, pdwRegister);
        pMoniker->Release();
    }
    pROT->Release();
    return hr;
}
```

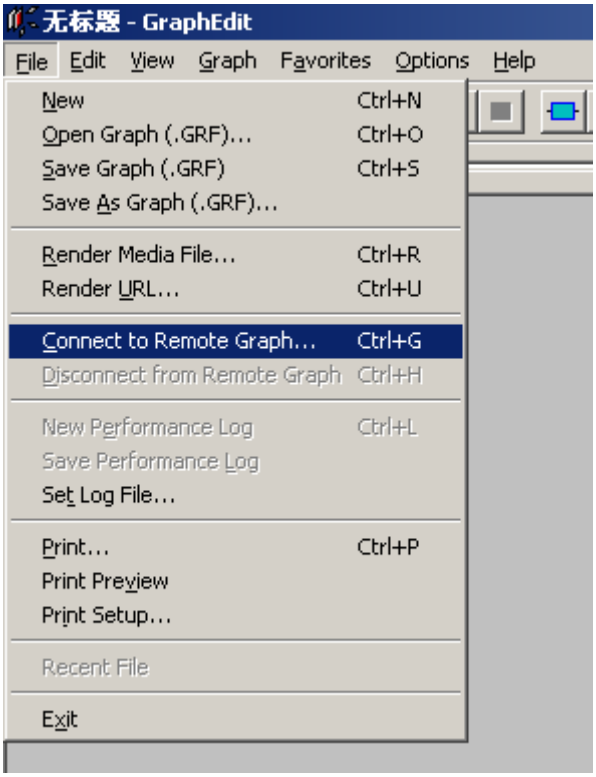
这个函数创建了一个 moniker 作为 filter graph 在 ROT 中的入口，第一个参数是指向 filter Graph 的指针，第二个参数返回 filter graph 在 ROT 中的入口。当应用程序销毁 filter graph 的时候，一定要调用下面的函数来删除这个 ROT 入口

```
void RemoveFromRot(DWORD pdwRegister)
{
    IRunningObjectTable *pROT;
    if (SUCCEEDED(GetRunningObjectTable(0, &pROT))) {
        pROT->Revoke(pdwRegister);
        pROT->Release();
    }
}
```

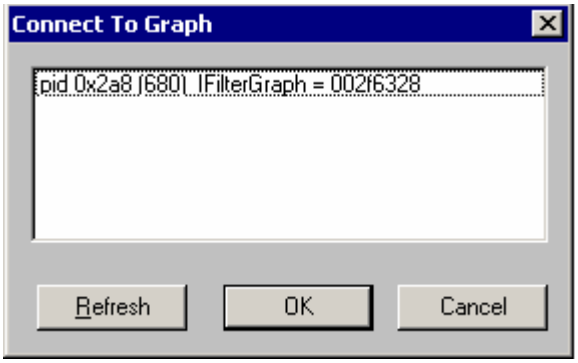
下面的代码演示了如何调用上面的两个函数，

```
IGraphBuilder *pGraph;
DWORD dwRegister;

// Create the filter graph manager.
CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
                                IID_IGraphBuilder, (void **)&pGraph);
#ifdef _DEBUG
hr = AddToRot(pGraph, &dwRegister);
#endif

// Rest of the application (not shown).

#ifdef _DEBUG
RemoveFromRot(dwRegister);
#endif
pGraph->Release();
```

同时运行你的应用程序和 GraphEdit，你就可以在 GraphEdit 中查看你应用程序中的 filter graph 了。在 GraphEdit 中，如下



然后就出现了下面的对话框



4 Saving a Filter Graph to a GraphEdit File

下面的代码演示了如何保存一个 GraphEdit（.gif）文件，这个可以用来调试你的应用程序。

```
HRESULT SaveGraphFile(IGraphBuilder *pGraph, WCHAR *wszPath)
{
    const WCHAR wszStreamName[] = L"ActiveMovieGraph";
    HRESULT hr;

    IStorage *pStorage = NULL;
    hr = StgCreateDocfile(
        wszPath,
        STGM_CREATE | STGM_TRANSACTED | STGM_READWRITE |
STGM_SHARE_EXCLUSIVE,
        0, &pStorage);
    if(FAILED(hr))
```

```
    {
        return hr;
    }

    IStream *pStream;
    hr = pStorage->CreateStream(
        wszStreamName,
        STGM_WRITE | STGM_CREATE | STGM_SHARE_EXCLUSIVE,
        0, 0, &pStream);
    if (FAILED(hr))
    {
        pStorage->Release();
        return hr;
    }

    IPersistStream *pPersist = NULL;
    pGraph->QueryInterface(IID_IPersistStream, (void**)&pPersist);
    hr = pPersist->Save(pStream, TRUE);
    pStream->Release();
    pPersist->Release();
    if (SUCCEEDED(hr))
    {
        hr = pStorage->Commit(STGC_DEFAULT);
    }
    pStorage->Release();
    return hr;
}
```

例如，下面的代码创建了文件回放的 graph 并保存为 MyGraph.grf:

```
void __cdecl main(void)
{
    HRESULT hr;
    IGraphBuilder *pGraph;
    CoInitialize(NULL);

    // Create the Filter Graph Manager and render a file.
    CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
        IID_IGraphBuilder, reinterpret_cast<void**>(&pGraph));
    hr = pGraph->RenderFile(L"C:\\Video.avi", NULL);

    if (SUCCEEDED(hr))
    {
        hr = SaveGraphFile(pGraph, L"C:\\MyGraph.grf");
    }
```

```
    pGraph->Release();
    CoUninitialize();
}
```

5 Loading a GraphEdit File Programmatically

在应用程序中可以通过 IPersistStream 接口来加载一个 GraphEdit (.grf) file，实例代码如下

```
HRESULT LoadGraphFile(IGraphBuilder *pGraph, const WCHAR* wszName)
{
    IStorage *pStorage = 0;
    if (S_OK != StgIsStorageFile(wszName))
    {
        return E_FAIL;
    }
    HRESULT hr = StgOpenStorage(wszName, 0,
        STGM_TRANSACTED | STGM_READ | STGM_SHARE_DENY_WRITE,
        0, 0, &pStorage);
    if (FAILED(hr))
    {
        return hr;
    }
    IPersistStream *pPersistStream = 0;
    hr = pGraph->QueryInterface(IID_IPersistStream,
            reinterpret_cast<void**>(&pPersistStream));
    if (SUCCEEDED(hr))
    {
        IStream *pStream = 0;
        hr = pStorage->OpenStream(L"ActiveMovieGraph", 0,
            STGM_READ | STGM_SHARE_EXCLUSIVE, 0, &pStream);
        if(SUCCEEDED(hr))
        {
            hr = pPersistStream->Load(pStream);
            pStream->Release();
        }
        pPersistStream->Release();
    }
    pStorage->Release();
    return hr;
}
```

必须要注意的是，GraphEdit 文件只是用来测试或者调试用的，并不是为了让终端客户用的。

2.2Directshow 基本应用（Basic Tasks）

2.2.1 视频提交（Video Rendering）

1 关于视频提交

首先介绍几个视频提交过滤器

Video Renderer

Video Mixing Renderer Filter 7 (VMR-7).

Video Mixing Renderer Filter 9 (VMR-9).

2 如何设置视频窗口

待补充

3 使用无窗口模式

dshow 的视频提交过滤器可以在窗口模式和无窗口模式下工作。在窗口模式下，过滤器创建一个自己的窗口，在里面播放视频。在无窗口模式下，过滤器直接将视频在应用程序提供的窗口上显示，过滤器本身不创建窗口。

窗口模式

在窗口模式下，视频提交过滤器创建一个窗口，然后将视频祯帖到窗口上，你可以将这个窗口帖到你的应用程序的窗口。

Video Renderer 只支持窗口模式，VMR-7 and VMR-9 缺省的是窗口模式，也支持无窗口模式。为了在你的应用程序中显示视频，你可以将视频窗口设置成应用程序的子窗口。你可以通过

```
IVideoWindow *pVidWin = NULL;
pGraph->QueryInterface(IID_IVideoWindow, (void **)&g_pVidWin);
pVidWin->put_Owner((OAHWND)hwnd);
pVidWin->put_WindowStyle(WS_CHILD | WS_CLIPSIBLINGS);
RECT grc;
GetClientRect(hwnd, &grc);
pVidWin->SetWindowPosition(0, 0, grc.right, grc.bottom);
```

结束时一定要清理现场

```
pControl->Stop();
pVidWin->put_Visible(OAFALSE);
pVidWin->put_Owner(NULL);
```

无窗口模式

当采用无窗口的模式时，就没有必要暴露 IVideoWindow 接口了。

为了能够使用 VMR 的缺省行为，在构建 Graph 图之前必须要调整 VMR。

1 创建一个过虑器图表管理器，

2 创建一个 VMR，加入到 graph 中，

3 调用 VMR 的 **IVMRFilterConfig::SetRenderingMode** 方法设置 VMRMode_Windowless 标志。

4 调用 **IVMRWindowlessControl::SetVideoClippingWindow** 给视频指定一个显示窗口。

然后调用 **IGraphBuilder::RenderFile** 或者其他的方法来创建其他的 Graph。

下面的代码显示了如何创建一个 VMR，将其添加到 Graph，如何设置无窗口模式

```
HRESULT InitWindowlessVMR(
```

```
    HWND hwndApp,                       // Window to hold the video.
    IGraphBuilder* pGraph,              // Pointer to the Filter Graph Manager.
    IVMRWindowlessControl** ppWc,    // Receives a pointer to the VMR.      )
{
    if (!pGraph || !ppWc) return E_POINTER;
    IBaseFilter* pVmr = NULL;
    IVMRWindowlessControl* pWc = NULL;
    // Create the VMR.
    HRESULT hr = CoCreateInstance(CLSID_VideoMixingRenderer, NULL,
        CLSCTX_INPROC, IID_IBaseFilter, (void**)&pVmr);
    if (FAILED(hr))
    {
        return hr;
    }

    // Add the VMR to the filter graph.
    hr = pGraph->AddFilter(pVmr, L"Video Mixing Renderer");
    if (FAILED(hr))
    {
        pVmr->Release();
        return hr;
    }
    // Set the rendering mode.
    IVMRFilterConfig* pConfig;
    hr = pVmr->QueryInterface(IID_IVMRFilterConfig, (void**)&pConfig);
    if (SUCCEEDED(hr))
    {
        hr = pConfig->SetRenderingMode(VMRMode_Windowless);
        pConfig->Release();
    }
    if (SUCCEEDED(hr))
    {
        // Set the window.
        hr = pVmr->QueryInterface(IID_IVMRWindowlessControl, (void**)&pWc);
        if( SUCCEEDED(hr))
        {
            hr = pWc->SetVideoClippingWindow(hwndApp);
            if (SUCCEEDED(hr))
            {
                *ppWc = pWc; // Return this as an AddRef'd pointer.
            }
            else
            {
                // An error occurred, so release the interface.
```

```
                pWc->Release();
            }
        }
    }
    pVmr->Release();
    return hr;
}
```

你也可以调用下面的函数

```
IVMRWindowlessControl *pWc = NULL;
hr = InitWindowlessVMR(hwnd, pGraph, &g_pWc);
if (SUCCEEDED(hr))
{
    // Build the graph. For example:
    pGraph->RenderFile(wszMyFileName, 0);
    // Release the VMR interface when you are done.
    pWc->Release();
}
```

下面看看如何设置视频的位置

有两个矩形需要考虑，一个是源矩形，一个是目的矩形。源矩形决定开始播放视频的位置，目的矩形决定在窗口显示视频的区域。VMR 将源矩形按照目的矩形的大小进行扩展。

IVMRWindowlessControl::SetVideoPosition 可以设置两个矩形的大小，源矩形必须小于等于本地视频大小。你可以通过 **IVMRWindowlessControl::GetNativeVideoSize** 获取本地的视频区域大小。

```
// Find the native video size.
long lWidth, lHeight;
HRESULT hr = g_pWc->GetNativeVideoSize(&lWidth, &lHeight, NULL, NULL);
if (SUCCEEDED(hr))
{
    RECT rcSrc, rcDest;
    // Set the source rectangle.
    SetRect(&rcSrc, 0, 0, lWidth/2, lHeight/2);

    // Get the window client area.
    GetClientRect(hwnd, &rcDest);
    // Set the destination rectangle.
    SetRect(&rcDest, 0, 0, rcDest.right/2, rcDest.bottom/2);

    // Set the video position.
    hr = g_pWc->SetVideoPosition(&rcSrc, &rcDest);
}
```

处理窗口消息

因为 VMR 没有自己的窗口，所以当视频需要重画或者改变的时候你要通知它。

1 当你接到一个 WM_PAINT 消息，你就要调用 **IVMRWindowlessControl::RepaintVideo** 来重画视频

2 当你接到一个 WM_DISPLAYCHANGE 消息，你就要调用 **IVMRWindowlessControl::DisplayModeChanged**.

3 当你接到一个 WM_SIZE 消息时，重新计算视频的位置，然后调用 SetVideoPostion。

下面的代码演示了 WM_PAINT 消息的处理

```
void OnPaint(HWND hwnd)
{
    PAINTSTRUCT ps;
    HDC         hdc;
    RECT        rcClient;
    GetClientRect(hwnd, &rcClient);
    hdc = BeginPaint(hwnd, &ps);
    if (g_pWc != NULL)
    {
        // Find the region where the application can paint by subtracting
        // the video destination rectangle from the client area.
        // (Assume that g_rcDest was calculated previously.)
        HRGN rgnClient = CreateRectRgnIndirect(&rcClient);
        HRGN rgnVideo  = CreateRectRgnIndirect(&g_rcDest);
        CombineRgn(rgnClient, rgnClient, rgnVideo, RGN_DIFF);

        // Paint on window.
        HBRUSH hbr = GetSysColorBrush(COLOR_BTNFACE);
        FillRgn(hdc, rgnClient, hbr);

        // Clean up.
        DeleteObject(hbr);
        DeleteObject(rgnClient);
        DeleteObject(rgnVideo);

        // Request the VMR to paint the video.
        HRESULT hr = g_pWc->RepaintVideo(hwnd, hdc);
    }
    else  // There is no video, so paint the whole client area.
    {
        FillRect(hdc, &rc2, (HBRUSH)(COLOR_BTNFACE + 1));
    }
    EndPaint(hwnd, &ps);
}
```

尽管我们要自己处理 onpaint 消息，但是已经非常简单了。

2.2.2 如何处理事件通知（Event Notification）

Responding to Events

2.2.3 如何枚举系统的设备和过虑器

有时，应用程序需要查看系统中所有的 filter。例如，视频应用程序需要列出系统中可用的捕捉设备。因为 dshow 基于 com 结构的，你在设计程序的时候是没法知道系统中正在使用的过滤器。Directshow 提供了两种方法来枚举系统中注册的过虑器。

1 系统设备枚举器

系统设备枚举器提供了一个很好的方法根据种类来枚举系统中注册的过虑器。也许枚一种不同的硬件都会有自己的过虑器，或许所有的硬件设备共用同一个 filter。这个对于采用 WDM 驱动程序的硬件很有用。

系统设备枚举器根据不同的种类创建了一个枚举器，例如，音频压缩，视频捕捉。不同种类的枚举器对于每一种设备返回一个独立的名称（moniker）。种类枚举器自动将相关的即插即用，演播设备包括进来。

按照下面的步骤使用设备枚举器

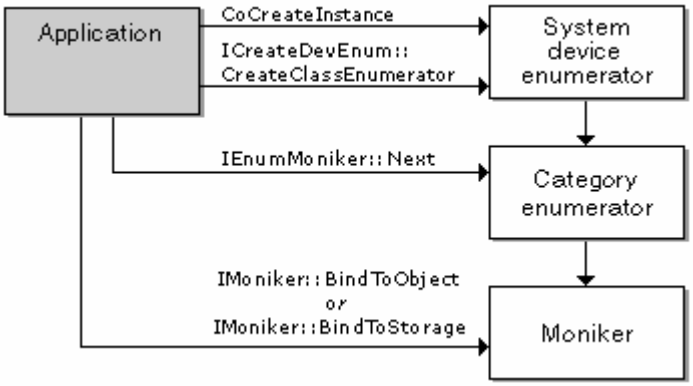
1 创建枚举器组件，CLSID 为 CLSID_SystemDeviceEnum

2 指定某一种类型设备，参数 CLSID，通过 **ICreateDevEnum::CreateClassEnumerator** 获取某一种类的枚举器，这个函数返回一个 **IEnumMoniker** 接口指针，如果该种类的空或者不存在，这个方法就返回 S_FALSE。因此，当你调用这个函数时一定要检查返回值是否为 S_OK，而不要用 SUCCEEDED 宏。

3 然后 **IEnumMoniker::Next** 枚举每一个 moniker。这个方法返回一个 IMoniker 接口指针。

4 要想知道设备的名称，可以通过下面的函数 **IMoniker::BindToStorage**

5 然后利用 **IMoniker::BindToObject 生成**绑定道设备上的 filter。调用 **IFilterGraph::AddFilter** 将 filter 添加到 Graph 图中。




```
// Create the System Device Enumerator.
HRESULT hr;
ICreateDevEnum *pSysDevEnum = NULL;
hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL, CLSCTX_INPROC_SERVER,
      IID_ICreateDevEnum, (void **)&pSysDevEnum);
if (FAILED(hr))
{
      return hr;
}

// Obtain a class enumerator for the video compressor category.
IEnumMoniker *pEnumCat = NULL;
```

```
hr=pSysDevEnum->CreateClassEnumerator(CLSID_VideoCompressorCategory,
&pEnumCat, 0);

if (hr == S_OK)
{
    // Enumerate the monikers.
    IMoniker *pMoniker = NULL;
    ULONG cFetched;
    while(pEnumCat->Next(1, &pMoniker, &cFetched) == S_OK)
    {
        IPropertyBag *pPropBag;
        hr = pMoniker->BindToStorage(0, 0, IID_IPropertyBag,
            (void **)&pPropBag);//知道设备的名称
        if (SUCCEEDED(hr))
        {
            // To retrieve the filter's friendly name, do the following:
            VARIANT varName;
            VariantInit(&varName);
            hr = pPropBag->Read(L"FriendlyName", &varName, 0);
            if (SUCCEEDED(hr))
            {
                // Display the name in your UI somehow.
            }
            VariantClear(&varName);

            // To create an instance of the filter, do the following:
            IBaseFilter *pFilter;
            hr = pMoniker->BindToObject(NULL, NULL, IID_IBaseFilter,
                (void**)&pFilter); //生成一个 filter 绑定到设备上。
            // Now add the filter to the graph.
            //Remember to release pFilter later.
            pPropBag->Release();
        }
        pMoniker->Release();
    }
    pEnumCat->Release();
}
pSysDevEnum->Release();
```

在上面我们用 **IMoniker::BindToObject 生成**绑定道设备上的 filter，当然我们还可以用另外的一种方法来生成绑定到设备上的 filter

利用 **IMoniker::GetDisplayName** 得到 moniker 的名字。然后你把 moniker 的名字做参数传递给 **IFilterGraph2::AddSourceFilterForMoniker**，就可以创建一个绑定到设备的 filter 了。在上面我们是调用 **IMoniker::BindToObject** 生成 filter 的，还是上面的简单些。看看代码吧。

```
LPOLESTR strName = NULL;
```

```
IBaseFilter pSrc = NULL;
hr = pMoniker->GetDisplayName(NULL, NULL, &strName);
if (SUCCEEDED(hr))
{
    // Query the Filter Graph Manager for IFilterGraph2.
    IFilterGraph2 *pFG2 = NULL;
    hr = pGraph->QueryInterface(IID_IFilterGraph2, (void**)&pFG2);
    if (SUCCEEDED(hr))
    {
        hr = pFG2->AddSourceFilterForMoniker(pMoniker, 0, L"Source", &pSrc);
        pFG2->Release();
    }
    CoTaskMemFree(strName);
}
// If successful, remember to release pSrc.
```

2 Filter Mapper

搜索系统中的 filter 的另一个方法就是采用 Filer Mapper。Filter mapper 是一个 com 对象，它按照一定的条件来搜索系统的 filer，它比系统设备枚举器（System Device Enumerator）的效率要低一些。所以当你要枚举某特定种类的 filter 时，你应该使用系统设备枚举器，但是当你搜索支持某种媒体类型的 filter 时，同时也找不到清晰的 filter，你应该使用 filter mapper。Filter Mapper 暴露一个 IFilerMapper2 接口，要想搜索一个接口，你可以调用该接口的 **IFilterMapper2::EnumMatchingFilters** 方法，这个方法需要传递一些参数来定义搜索条件，同时该方法返回一个适合条件的 filter 的枚举器，这个枚举器提供一个 IEnumMoniker 接口，并且对于每个适合的 filter 都提供一个单独的 moniker。

下面的例子演示了，枚举所有的支持 DV，并且至少有一个输出 pin 的 filter，这个 filter 支持任何媒体类型。

```
IFilterMapper2 *pMapper = NULL;
IEnumMoniker *pEnum = NULL;

hr =CoCreateInstance( CLSID_FilterMapper2,NULL, CLSCTX_INPROC, IID_IFilterMapper2,
                      (void **) &pMapper);
if (FAILED(hr))
{
    // Error handling omitted for clarity.
}

GUID arrayInTypes[2];
arrayInTypes[0] = MEDIATYPE_Video;
arrayInTypes[1] = MEDIASUBTYPE_dvsd;

hr = pMapper->EnumMatchingFilters(
        &pEnum,
        0,                      // Reserved.
        TRUE,                   // Use exact match?
```

```
            MERIT_DO_NOT_USE+1, // Minimum merit.
            TRUE,                       // At least one input pin?
            1,                          // Number of major type/subtype pairs for input.
            arrayInTypes,           // Array of major type/subtype pairs for input.
            NULL,                       // Input medium.
            NULL,                       // Input pin category.
            FALSE,                      // Must be a renderer?
            TRUE,                       // At least one output pin?
            0,                          // Number of major type/subtype pairs for output.
            NULL,                       // Array of major type/subtype pairs for output.
            NULL,                       // Output medium.
            NULL);                      // Output pin category.

// Enumerate the monikers.
IMoniker *pMoniker;
ULONG cFetched;
//////////下面就是枚举 filter 了，就是系统枚举设备 filter
while (pEnumCat->Next(1, &pMoniker, &cFetched) == S_OK)
{
    IPropertyBag *pPropBag = NULL;
    hr = pMoniker->BindToStorage(0, 0, IID_IPropertyBag,
        (void **)&pPropBag);

    if (SUCCEEDED(hr))
    {
        // To retrieve the friendly name of the filter, do the following:
        VARIANT varName;
        VariantInit(&varName);
        hr = pPropBag->Read(L"FriendlyName", &varName, 0);
        if (SUCCEEDED(hr))
        {
            // Display the name in your UI somehow.
        }
        VariantClear(&varName);

        // To create an instance of the filter, do the following:
        IBaseFilter *pFilter;
        hr = pMoniker->BindToObject(NULL, NULL, IID_IBaseFilter, (void**)&pFilter);
        // Now add the filter to the graph. Remember to release pFilter later.

        // Clean up.
        pPropBag->Release();
    }
    pMoniker->Release();
```

```
}
// Clean up.
pMapper->Release();
pEnum->Release();
```

2.2.4 如何枚举 Graph 图中的对象（filter，pin）

有些时候，应用程序需要枚举 graph 中的 filter 或者是枚举 filter 所支持的 pin。因此 directshow 提供了枚举 graph filter 中的 com 组件方法。

1 枚举 filter

Filter 图表管理器支持 **IFilterGraph::EnumFilters** 方法，来枚举 graph 图中的所有的 filter。他返回一个 IEnumFilters 接口，利用这个接口就可以遍历 graph 中的所有的 filter。

下面的代码演示了，如何遍历 graph 中的 filter，并且显示 filter 的名字。

```
HRESULT EnumFilters (IFilterGraph *pGraph)
{
    IEnumFilters *pEnum = NULL;
    IBaseFilter *pFilter;
    ULONG cFetched;
    HRESULT hr = pGraph->EnumFilters(&pEnum);
    if (FAILED(hr)) return hr;
    while(pEnum->Next(1, &pFilter, &cFetched) == S_OK)
    {
        FILTER_INFO FilterInfo;
        hr = pFilter->QueryFilterInfo(&FilterInfo);
        if (FAILED(hr))
        {
            MessageBox(NULL, TEXT("Could not get the filter info"),
                TEXT("Error"), MB_OK | MB_ICONERROR);
            continue;    // Maybe the next one will work.
        }

#ifdef UNICODE
        MessageBox(NULL, FilterInfo.achName, TEXT("Filter Name"), MB_OK);
#else
        char szName[MAX_FILTER_NAME];
        int cch = WideCharToMultiByte(CP_ACP, 0, FilterInfo.achName,
            MAX_FILTER_NAME, szName, MAX_FILTER_NAME, 0, 0);
        if (chh > 0)
            MessageBox(NULL, szName, TEXT("Filter Name"), MB_OK);
#endif
        // The FILTER_INFO structure holds a pointer to the Filter Graph
        // Manager, with a reference count that must be released.
        if (FilterInfo.pGraph != NULL)
        {
```

```
            FilterInfo.pGraph->Release();
        }
        pFilter->Release();
    }
    pEnum->Release();
    return S_OK;
}
```

2 枚举 pin

Filter 支持 **IBaseFilter::EnumPins** 方法，这个方法可以可以枚举 filter 所有的 pin。它返回一个 IEnumPins 接口，**IEnumPins::Next** 可以遍历 pin 的接口。

下面的代码演示了如何如何查找一个输出和输入 pin。利用 PIN_DIRECTION 参数来制定 pin 的类型（输入还是输出）。

```
HRESULT GetPin(IBaseFilter *pFilter, PIN_DIRECTION PinDir, IPin **ppPin)
{
    IEnumPins    *pEnum = NULL;
    IPin         *pPin = NULL;
    HRESULT      hr;

    if (ppPin == NULL)
    {
        return E_POINTER;
    }

    hr = pFilter->EnumPins(&pEnum);
    if (FAILED(hr))
    {
        return hr;
    }
    while(pEnum->Next(1, &pPin, 0) == S_OK)
    {
        PIN_DIRECTION PinDirThis;
        hr = pPin->QueryDirection(&PinDirThis);
        if (FAILED(hr))
        {
            pPin->Release();
            pEnum->Release();
            return hr;
        }
        if (PinDir == PinDirThis) //如果类型符合
        {
            // Found a match. Return the IPin pointer to the caller.
            **ppPin = pPin;
            pEnum->Release();
            return S_OK;
```

```
            }
            // Release the pin for the next time through the loop.
            pPin->Release();
        }
        // No more pins. We did not find a match.
        pEnum->Release();
        return E_FAIL;
}
```

利用这个方法可以很容易的就查找一个 pin，然后调用 **IPin::ConnectedTo** 方法确定这个 pin 是否被连接，可以查找一个空闲的 pin。

3 查找媒体类型

每个 pin 都支持一个 **IPin::EnumMediaTypes** 方法，可以来枚举 pin 支持的媒体类型。它返回一个 IEnumMediaTypes 接口，这个接口的方法 **IEnumMediaTypes::Next** 返回一个指向 **AM_MEDIA_TYPE** 类型的指针。可以参考上面的代码来遍历 pin 所支持的媒体类型。

2.2.5 构建 Graph 图常用技术

2.2.5.1 如何根据 CLSID 向 graph 中添加 filter

下面的代码演示了如何利用 CLSID 生成一个 filter，然后将其加入到 graph 图中

```
HRESULT AddFilterByCLSID(
    IGraphBuilder *pGraph,    // Pointer to the Filter Graph Manager.
    const GUID& clsid,        // CLSID of the filter to create.
    LPCWSTR wszName,          // A name for the filter.
    IBaseFilter **ppF)        // Receives a pointer to the filter.
{
    if (!pGraph || ! ppF) return E_POINTER;
    *ppF = 0;
    IBaseFilter *pF = 0;
    HRESULT hr = CoCreateInstance(clsid, 0, CLSCTX_INPROC_SERVER,
        IID_IBaseFilter, reinterpret_cast<void**>(&pF));
    if (SUCCEEDED(hr))
    {
        hr = pGraph->AddFilter(pF, wszName);
        if (SUCCEEDED(hr))
            *ppF = pF;
        else
            pF->Release();
    }
    return hr;
}
```

在你的应用程序中，你可以这样用这个函数

```
IBaseFilter *pMux;
```

```
hr = AddFilterByCLSID(pGraph, CLSID_AviDest, L"AVI Mux", &pMux);
if (SUCCEEDED(hr))
{
    /* ... */
   pMux->Release();
}
```

注：有些 filter 是不能通过 with **CoCreateInstance 方法创建的。例如** AVI Compressor Filter 和 WDM Video Capture filter

2.2.5.2 如何查找 filter 空闲的 pin。

看代码把

```
HRESULT GetUnconnectedPin(
    IBaseFilter *pFilter,    // Pointer to the filter.
    PIN_DIRECTION PinDir,    // Direction of the pin to find.
    IPin **ppPin)            // Receives a pointer to the pin.
{
    *ppPin = 0;
    IEnumPins *pEnum = 0;
    IPin *pPin = 0;
    HRESULT hr = pFilter->EnumPins(&pEnum);
    if (FAILED(hr))
    {
        return hr;
    }
    while (pEnum->Next(1, &pPin, NULL) == S_OK)
    {
        PIN_DIRECTION ThisPinDir;
        pPin->QueryDirection(&ThisPinDir);
        if (ThisPinDir == PinDir)
        {
            IPin *pTmp = 0;
            hr = pPin->ConnectedTo(&pTmp);
            if (SUCCEEDED(hr))  // Already connected, not the pin we want.
            {
                pTmp->Release();
            }
            else  // Unconnected, 这就是我们想要的 pin，空闲的 pin
            {
                pEnum->Release();
                *ppPin = pPin;
                return S_OK;
            }
        }
```

```
            pPin->Release();
        }
        pEnum->Release();
        // Did not find a matching pin.
        return E_FAIL;
}
```

下面的代码演示了如何利用上面的函数来在一个 filter 查找一个输出的空闲的 pin。

```
IPin *pOut = NULL;
HRESULT hr = GetUnconnectedPin(pFilter, PINDIR_OUTPUT, &pOut);
if (SUCCEEDED(hr))
{
        /* ... */
        pOut->Release();
}
```

2.2.5.3 如何连接两个 Filter

下面的函数演示了如何将一个 filter 的输出 pin 和另一个 filter 的第一个空闲的输入 pin 进行连接。

```
HRESULT ConnectFilters(
        IGraphBuilder *pGraph, // Filter Graph Manager.
        IPin *pOut,                     // Output pin on the upstream filter.
        IBaseFilter *pDest)        // Downstream filter.
{
        if ((pGraph == NULL) || (pOut == NULL) || (pDest == NULL))
        {
            return E_POINTER;
        }
#ifdef debug
            PIN_DIRECTION PinDir;
            pOut->QueryDirection(&PinDir);
            _ASSERTE(PinDir == PINDIR_OUTPUT);
#endif

        //找一个空闲的输入 pin
        IPin *pIn = 0;
        HRESULT hr = GetUnconnectedPin(pDest, PINDIR_INPUT, &pIn);
        if (FAILED(hr))
        {
            return hr;
        }
        // Try to connect them.
        hr = pGraph->Connect(pOut, pIn);
        pIn->Release();
```

```
    return hr;
}
```

下面是 ConnectFilters 的一个重载函数，但是第二个参数是一个指向 filter 的指针，而不是指向 pin 的指针，这个函数将两个 filter 连接起来。

```
HRESULT ConnectFilters(
    IGraphBuilder *pGraph,
    IBaseFilter *pSrc,
    IBaseFilter *pDest)
{
    if ((pGraph == NULL) || (pSrc == NULL) || (pDest == NULL))
    {
        return E_POINTER;
    }

    // 首先在第一个 filter 上查询一个输出的 pin 接口
    IPin *pOut = 0;
    HRESULT hr = GetUnconnectedPin(pSrc, PINDIR_OUTPUT, &pOut);
    if (FAILED(hr))
    {
        return hr;
    }
    //然后将它和第二个 filter 的输入接口衔接。
    hr = ConnectFilters(pGraph, pOut, pDest);
    pOut->Release();
    return hr;
}
```

下面的函数演示了利用这个函数来连接 AVIMux 过滤器和 File Writer 过滤器，这个例子也使用了 AddFilterByCLSID 函数。

```
IBaseFilter *pMux, *pWrite;
hr = AddFilterByCLSID(pGraph, CLSID_AviDest, L"AVI Mux", &pMux);
if (SUCCEEDED(hr))
{
    hr = AddFilterByCLSID(pGraph, CLSID_FileWriter, L"File Writer", &pWrite);
    if (SUCCEEDED(hr))
    {
        hr = ConnectFilters(pGraph, pMux, pWrite);
        /* Use IFileSinkFilter to set the file name (not shown). */
       pWrite->Release();
    }
    pMux->Release();
}
```

2.2.5.4 如何获得 filter 或者 pin 的接口指针

一般来说，我们都是通过 Filter 图表管理器来进行一些操作，但是，有时候，我们也直接调用 filter 或者 pin 的一些方法，因此，我们需要获取 filter 或 pin 的接口指针。
对于 filter 的接口指针，可以通过 IEnumFilters 来枚举 filter 的指针，看下面的代码把

```
HRESULT FindFilterInterface(
    IGraphBuilder *pGraph, // Pointer to the Filter Graph Manager.
    REFGUID iid,           // IID of the interface to retrieve.
    void **ppUnk)          // Receives the interface pointer.
{
    if (!pGraph || !ppUnk) return E_POINTER;

    HRESULT hr = E_FAIL;
    IEnumFilters *pEnum = NULL;
    IBaseFilter *pF = NULL;
    if (FAILED(pGraph->EnumFilters(&pEnum)))
    {
        return E_FAIL;
    }
    // Query every filter for the interface.
    while (S_OK == pEnum->Next(1, &pF, 0))
    {
        hr = pF->QueryInterface(iid, ppUnk);
        pF->Release();
        if (SUCCEEDED(hr))
        {
            break;
        }
    }
    pEnum->Release();
    return hr;
}
```

用 **IEnumPins** 来获得 pin 的接口指针，其实就是枚举哦

```
HRESULT FindPinInterface(
    IBaseFilter *pFilter,    // Pointer to the filter to search.
    REFGUID iid,             // IID of the interface.
    void **ppUnk)            // Receives the interface pointer.
{
    if (!pFilter || !ppUnk) return E_POINTER;

    HRESULT hr = E_FAIL;
    IEnumPins *pEnum = 0;
    if (FAILED(pFilter->EnumPins(&pEnum)))
    {
```

```
            return E_FAIL;
        }
        // Query every pin for the interface.
        IPin *pPin = 0;
        while (S_OK == pEnum->Next(1, &pPin, 0))
        {
            hr = pPin->QueryInterface(iid, ppUnk);
            pPin->Release();
            if (SUCCEEDED(hr))
            {
                break;
            }
        }
        pEnum->Release();
        return hr;
}
```

下面的代码演示了如何搜索任意的 filter 和 pin 的接口

```
HRESULT FindInterfaceAnywhere(
        IGraphBuilder *pGraph,
        REFGUID iid,
        void **ppUnk)
{
        if (!pGraph || !ppUnk) return E_POINTER;
        HRESULT hr = E_FAIL;
        IEnumFilters *pEnum = 0;
        if (FAILED(pGraph->EnumFilters(&pEnum)))
        {
            return E_FAIL;
        }
        // Loop through every filter in the graph.
        IBaseFilter *pF = 0;
        while (S_OK == pEnum->Next(1, &pF, 0))
        {
            hr = pF->QueryInterface(iid, ppUnk);
            if (FAILED(hr))
            {
                // The filter does not expose the interface, but maybe
                // one of its pins does. //调用的是上面的搜索 pin 的函数
                hr = FindPinInterface(pF, iid, ppUnk);
            }
            pF->Release();
            if (SUCCEEDED(hr))
            {
                break;
```

```
        }
    }
    pEnum->Release();
    return hr;
}
```

2.2.5.5 如何查找和某个 filter 的上下相连的 filter

给你一个 filter，你可以沿着 graph 图找到和它相联结的 filter。首先枚举 filter 的 pin，检查每一个 pin 是否有其他的 pin 的和它连接，如果有就检查连接 pin 属于哪个 filter，你可以通过输入 pin 检查上游的 filter，通过输出 pin 来检查下游的 filter。

下面的函数返回上游或者下游的和本 filter 连接的 filter，只要有一个 match，就返回。

```
// Get the first upstream or downstream filter
HRESULT GetNextFilter(
    IBaseFilter *pFilter, //  开始的 filter
    PIN_DIRECTION Dir,      //  搜索的方向 (upstream 还是 downstream)
    IBaseFilter **ppNext) // Receives a pointer to the next filter.
{
    if (!pFilter || !ppNext) return E_POINTER;

    IEnumPins *pEnum = 0;
    IPin *pPin = 0;
    HRESULT hr = pFilter->EnumPins(&pEnum);
    if (FAILED(hr)) return hr;
    while (S_OK == pEnum->Next(1, &pPin, 0))
    {
        // See if this pin matches the specified direction.
        PIN_DIRECTION ThisPinDir;
        hr = pPin->QueryDirection(&ThisPinDir);
        if (FAILED(hr))
        {
            // Something strange happened.
            hr = E_UNEXPECTED;
            pPin->Release();
            break;
        }
        if (ThisPinDir == Dir)
        {
            // Check if the pin is connected to another pin.
            IPin *pPinNext = 0;
            hr = pPin->ConnectedTo(&pPinNext);
            if (SUCCEEDED(hr))
            {
                // Get the filter that owns that pin.
```

```
                PIN_INFO PinInfo;
                hr = pPinNext->QueryPinInfo(&PinInfo);
                pPinNext->Release();
                pPin->Release();
                pEnum->Release();
                if (FAILED(hr) || (PinInfo.pFilter == NULL))
                {
                    // Something strange happened.
                    return E_UNEXPECTED;
                }
                // This is the filter we're looking for.
                *ppNext = PinInfo.pFilter; // Client must release.
                return S_OK;
            }
        }
        pPin->Release();
    }
    pEnum->Release();
    // Did not find a matching filter.
    return E_FAIL;
}
```

下面演示如何使用这个函数

```
IBaseFilter *pF; // Pointer to some filter.
IBaseFilter *pUpstream = NULL;
if (SUCCEEDED(GetNextFilter(pF, PINDIR_INPUT, &pUpstream)))
{
    // Use pUpstream ...
    pUpstream->Release();
}
```

但是，一个 filter 可能在某个方向同时连接着两个或者更多个 filter，例如一个分割 filter，就有好几个 filter 与之相联。 因此，你可能想将所有的 filter 通过一个集合都搜集到。下面的例子代码就演示了如何通过 **CGenericList** 结构来实现这个方法。

```
#include <streams.h>    // Link to the DirectShow base class library
// Define a typedef for a list of filters.
typedef CGenericList<IBaseFilter> CFilterList;

// Forward declaration. Adds a filter to the list unless it's a duplicate.
void AddFilterUnique(CFilterList &FilterList, IBaseFilter *pNew);

// Find all the immediate upstream or downstream peers of a filter.
HRESULT GetPeerFilters(
    IBaseFilter *pFilter, // Pointer to the starting filter
    PIN_DIRECTION Dir,       // Direction to search (upstream or downstream)
```

```
    CFilterList &FilterList)    // Collect the results in this list.
{
    if (!pFilter) return E_POINTER;

    IEnumPins *pEnum = 0;
    IPin *pPin = 0;
    HRESULT hr = pFilter->EnumPins(&pEnum);
    if (FAILED(hr)) return hr;
    while (S_OK == pEnum->Next(1, &pPin, 0))
    {
        // See if this pin matches the specified direction.
        PIN_DIRECTION ThisPinDir;
        hr = pPin->QueryDirection(&ThisPinDir);
        if (FAILED(hr))
        {
            // Something strange happened.
            hr = E_UNEXPECTED;
            pPin->Release();
            break;
        }
        if (ThisPinDir == Dir)
        {
            // Check if the pin is connected to another pin.
            IPin *pPinNext = 0;
            hr = pPin->ConnectedTo(&pPinNext);
            if (SUCCEEDED(hr))
            {
                // Get the filter that owns that pin.
                PIN_INFO PinInfo;
                hr = pPinNext->QueryPinInfo(&PinInfo);
                pPinNext->Release();
                if (FAILED(hr) || (PinInfo.pFilter == NULL))
                {
                    // Something strange happened.
                    pPin->Release();
                    pEnum->Release();
                    return E_UNEXPECTED;
                }
                //  将符合的 filter 添加到 list 中
                AddFilterUnique(FilterList, PinInfo.pFilter);
                PinInfo.pFilter->Release();
            }
        }
        pPin->Release();
```

```
    }
    pEnum->Release();
    return S_OK;
}
void AddFilterUnique(CFilterList &FilterList, IBaseFilter *pNew)
{
    if (pNew == NULL) return;

    POSITION pos = FilterList.GetHeadPosition();
    while (pos)
    {
        IBaseFilter *pF = FilterList.GetNext(pos);
        if (IsEqualObject(pF, pNew))
        {
            return;
        }
    }
    pNew->AddRef();    // The caller must release everything in the list.
    FilterList.AddTail(pNew);
}
```

如何应用上面的函数呢？看看下面就知道了撒

```
IBaseFilter *pF; // Pointer to some filter.
CFilterList FList(NAME("MyList"));    // List to hold the downstream peers.
hr = GetPeerFilters(pF, PINDIR_OUTPUT, FList);
if (SUCCEEDED(hr)) //解析 filter 的集合。
{
    POSITION pos = FList.GetHeadPosition();
    while (pos)
    {
        IBaseFilter *pDownstream = FList.GetNext(pos);
        pDownstream->Release();
    }
}
```

2.2.5.6 如何删除 graph 中的所有 filter

很简单的，采用 **IFilterGraph::RemoveFilter** 函数

```
// Stop the graph.
pControl->Stop();

// Enumerate the filters in the graph.
IEnumFilters *pEnum = NULL;
HRESULT hr = pGraph->EnumFilters(&pEnum);
if (SUCCEEDED(hr))
```

```
{
    IBaseFilter *pFilter = NULL;
    while (S_OK == pEnum->Next(1, &pFilter, NULL))
     {
        // Remove the filter.
        pGraph->RemoveFilter(pFilter);
        // Reset the enumerator.
        pEnum->Reset();
        pFilter->Release();
    }
    pEnum->Release();
}
```

2.2.5.7 如何利用 Capture Graph Builder 构建 Graph 图表

Capture Graph Builder 可以用来构建大多数的 filter 图表，并不仅仅是捕捉 graph。本文简单介绍了如何利用 Capture Graph Builder 来构建 graph。

Capture Graph Builder 暴露了 **ICaptureGraphBuilder2** 接口指针，首先创建一个 **capture builder**，和一个 filter 图表管理器对象，然后用图表管理器对象指针初始化 Capture Graph Builder。代码如下：

```
IGraphBuilder *pGraph = NULL;
ICaptureGraphBuilder2 *pBuilder = NULL;

// Create the Filter Graph Manager.
HRESULT hr =    CoCreateInstance(CLSID_FilterGraph, NULL,
    CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);

if (SUCCEEDED(hr))
{
    // Create the Capture Graph Builder.
    hr = CoCreateInstance(CLSID_CaptureGraphBuilder2, NULL,
        CLSCTX_INPROC_SERVER, IID_ICaptureGraphBuilder2,
        (void **)&pBuilder);
    if (SUCCEEDED(hr))
    {
        pBuilder->SetFiltergraph (pGraph);
    }
};
```

连接 filter

ICaptureGraphBuilder2::RenderStream 方法可以同时将两个或者三个 filter 连接成一个链（chain）。通常情况下，当每个 filter 只有一个输出 pin 和一个输入 pin 时，这个方法就才，适用。

我们现在先忽略前两个参数，第三个参数是一个 IUnknown 指针，指向一个 filter 或者输出 pin。第五，六个参数指向 IBaseFilter 指针。RenderStream 就将三个 filter 连接成一个链。

例如，假设 A，B，C 是三个 filter，每个 filter 只有一个输出 pin 和一个输入 pin。
下面的代码可以将 B 连接到 A 上，将 B 连接到 C 上。
RenderStream(NULL, NULL, A, B, C)
所有的连接都是智能化的，如果是将两个 filter 相连，你可以将中间的参数设置为 NULL，
RenderStream(NULL, NULL, A, NULL, C)
你也可以调用两次这个函数创建一个更长的链条。
RenderStream(NULL, NULL, A, B, C)
RenderStream(NULL, NULL, C, D, E)
如果最后的一个参数设置为 NULL，这个方法就自动的为 graph 设置一个 renderer filter。如果是视频就设置成 Video Renderer，如果是音频就设置为 DirectSoundRenderer。因此
RenderStream(NULL, NULL, A, NULL, NULL)
等价于
RenderStream(NULL, NULL, A, NULL, R)
这里 R 指的是 Render Filter。
如果你在第三个参数指定的是 filter，而不是 pin，你就要在第一二个参数里指定使用那个输出 pin 用于连接。
第一个参数只适用于捕捉 filter，它指定 pin 的所属种类的 GUID，具体的设置可以参考 Pin Property Set.，但是下面的两个种类对于所有的 filter 都有效。
PIN_CATEGORY_CAPTURE
PIN_CATEGORY_PREVIEW
如果捕捉 filter 不支持捕捉和预览，RenderStream 方法就增加一个 Smart Tee 来分割数据流。
如果播放文件，要将捕捉 filter 和一个 mux filter 连接起来，
第二个参数指明了媒体类型
MEDIATYPE_Audio
MEDIATYPE_Video
MEDIATYPE_Interleaved (DV)
查询 filter 和 pin 的接口指针
当你建立一个 graph 后，也许你需要查询 graph 中的 filter 和 pin 暴露的接口指针。例如，一个捕捉 filter 也许暴露了 **IAMDroppedFrames** 接口，它的输出 pin 也许暴露了 **IAMStreamConfig** 接口。
查询接口最简单地方法就是使用 **ICaptureGraphBuilder2::FindInterface** 方法。这个方法遍历整个 graph 的 filter 和 pin，直到他找到合适的 filter。你可以指定开始的 filter，然后指定搜索的方向，（向上搜索还是向下搜索）
下面的代码在一个视频预览 pin 上搜索 IAMStreamConfig 接口

```
IAMStreamConfig *pConfig = NULL;
HRESULT hr = pBuild->FindInterface(
    &PIN_CATEGORY_PREVIEW,
    &MEDIATYPE_Video,
    pVCap,
    IID_IAMStreamConfig,
    (void**)&pConfig
);
if (SUCCESSFUL(hr))
{
```

```
    /* ... */
    pConfig->Release();
}
```

查找 pin

如果你需要在某个 filter 上查询某个接口，可以用 **ICaptureGraphBuilder2::FindPin** 方法，代码如下：

```
IPin *pPin = NULL;
hr = pBuild->FindPin(
    pCap,                       // Pointer to the filter to search.
    PINDIR_OUTPUT,              // Search for an output pin.
    &PIN_CATEGORY_PREVIEW,      // Search for a preview pin.
    &MEDIATYPE_Video,           // Search for a video pin.
    TRUE,                       // The pin must be unconnected.
    0,                          // Return the first matching pin (index 0).
    &pPin);                     // This variable receives the IPin pointer.
if (SUCCESSFUL(hr))
{
    /* ... */
    pPin->Release();
}
```

2.2.6 Seeking Filter graph

这篇文档主要讲述了如何在一个媒体数据流中定位，任意指定开始播放的位置。

1 检查是否支持 seek

Directshow 通过 **IMediaSeeking** 接口支持 seeking。Filter graph 管理器支持这个接口，但是实际 seeking 的功能是有 graph 中的 filter 来实现的。

有一些数据是不能 seek 的，例如，你不可能 seek 从照相机中采集的活动的视频流。如果一个数据流可以被 seek，但是，seek 的类型还分以下几种类型，可以给你的数据流选择一种

1 定位到数据流中的一个绝对位置

2 返回数据流的持续时间

3 返回数据流中的当前播放位置

4 回放。

IMediaSeeking 接口定义了一套标志 **AM_SEEKING_SEEKING_CAPABILITIES**,用来描述可能支持的 seek 功能。

```
typedef enum AM_SEEKING_SeekingCapabilities {
    AM_SEEKING_CanSeekAbsolute      = 0x1,
    AM_SEEKING_CanSeekForwards      = 0x2,
    AM_SEEKING_CanSeekBackwards     = 0x4,
    AM_SEEKING_CanGetCurrentPos     = 0x8,
    AM_SEEKING_CanGetStopPos        = 0x10,
    AM_SEEKING_CanGetDuration       = 0x20,
    AM_SEEKING_CanPlayBackwards     = 0x40,
    AM_SEEKING_CanDoSegments        = 0x80,
```

```
    AM_SEEKING_Source                   = 0x100
}   AM_SEEKING_SEEKING_CAPABILITIES;
```

可以通过 **IMediaSeeking::GetCapabilities** 查看数据流支持的 seek 能力都有哪些。应用程序可以采取 &测试每一项。例如，下面的代码检查了 graph 是否可以 seek 一个任意的位置

```
 DWORD dwCap = 0;
HRESULT hr = pSeek->GetCapabilities(&dwCap);
if (AM_SEEKING_CanSeekAbsolute & dwCap)
{
    // Graph can seek to absolute positions.
}
```

2Setting and Retrieving the Position

Filter graph 包含两个位置，当前位置和停止位置，定义如下：

1 当前位置，当一个 graph 正处于运行的时候，当前位置就是当前的回放位置，相对于开始的位置而言。如果 graph 处于停止或者暂停状态的时候，当前位置就是数据流下次开始播放的位置点。

2 停止位置，停止位置就是数据流将要停止的位置，当一个 graph 到达一个停止位置时，将没有数据流，filter graph 管理器将会发送一个 EC_COMPLETE 事件。

可以通过 **IMediaSeeking::GetPositions** 方法可以获取这些位置值。返回值都是相对于原始的开始位置。

通过 **IMediaSeeking::SetPositions** 方法可以 seek 一个新的位置，见下面：

```
#define ONE_SECOND 10000000
REFERENCE_TIME rtNow    = 2 * ONE_SECOND,
               rtStop = 5 * ONE_SECOND;

hr = pSeek->SetPositions(
    &rtNow,    AM_SEEKING_AbsolutePositioning,
    &rtStop, AM_SEEKING_AbsolutePositioning
    );
```

注：1 秒是 10,000,000 参考时间单位。为了方便，这个例子将这个值定义为 ONE_SECOND，如果你使用的 dshow 的基类，常量 CUITS 的值和这个值相等。

RtNow 参数指定新的当前位置，第二个参数用来标示如何来定位 rtNow 参数。在这个例子中，AM_SEEKING_AbsolutePositioning 标志表示 rtNow 指定的位置是一个绝对的位置。RtStop 参数指定了停止时间，最后一个参数也指定了绝对位置。

如果想指定一个相对的位置，可以指定一个 AM_SEEKING_RelativePositioning 参数，为了设置这个位置不能改变，可以指定一个 AM_SEEKING_NoPositioning 参数。此时，参考时间应该设置为 NULL。下面的例子将位置向前 seek 10 秒，然后停止位置不变。

```
REFERENCE_TIME rtNow = 10 * ONE_SECOND;
hr = pSeek->SetPositions(
    &rtNow, AM_SEEKING_RelativePositioning,
    NULL, AM_SEEKING_NoPositioning
    );
```

3Setting the Playback Rate

调用 IMediaSeeking::SetRate 方法可以改变回放的速率。通过将新的速率设置成原来速率的倍数就可以设置新的速率，例如，pSeek->SetRate(2.0)将新的速率设置为原来速率的两倍。比率大于 1 说明回放的速度比原来的大，如果介于 0 和 1 之间，就比正常的速度慢。

如果我们不考虑回放速率，当前位置和停止位置相对于开始位置都是不变的。举个例子，如果我们有一个可以播放 20 秒的文件，将当前时间设置为 10 秒就会将播放位置设置到中间，如果播放的速率提高要原来的 2 倍，如果停止时间是 20 秒，你将播放位置设置到原来的 10 秒处，结果现在只能播放 5 秒了，因为速度提高了两倍。

4Time Formats For Seek Commands

IMediaSeeking 接口中的许多函数的参数都要求指定一个位置值，比如当前位置，或者停止位置，缺省的情况下这些参数是以 of 100 nanoseconds 为时间单位的，称为参考时间，任何支持 seek 的 filter 必须支持按参考时间来进行定位。一些 filter 也支持采取其他时间单位进行定位。例如，根据指定的桢的数量，或在数据流偏移的字节数进行定位。

这种用来定位的时间单位称为时间格式，采用一个 GUID 来标示。Directshow 定义了一系列的时间格式，详细地可以参考 SDK。第三方也可以定义自己的时间格式。

为了确定 graph 中的当前的 filter 是否支持特定的时间格式，可以调用 **IMediaSeeking::IsFormatSupported** 方法，如果 filter 支持该时间格式，该函数返回 ok 否则返回 false 或者一个错误码。如果 filter 支持某种指定的时间格式，可以调用 **IMediaSeeking::SetTimeFormat** 方法切换到其他的时间格式。如果 SetTimeFormat 方法成功，下面的 seek 命令就要使用新的时间格式。

下面的代码检查 graph 是否支持用桢的数量进行定位，如果支持，定位到第 20 桢。

```
hr = pSeek->IsFormatSupported(&TIME_FORMAT_FRAME);
if (hr == S_OK)
{
    hr = pSeek->SetTimeFormat(&TIME_FORMAT_FRAME);
    if (SUCCEEDED(hr))
    {
        // Seek to frame number 20.
        LONGLONG rtNow = 20;
        hr = pSeek->SetPositions(
            &rtNow, AM_SEEKING_AbsolutePositioning,
            0, AM_SEEKING_NoPositioning);
    }
}
```

2.2.7 如何设置 Graph 时钟（Setting Graph Clock）

当你构建了一个 graph 后，graph 管理器会自动地给你的 graph 选择一个参考时钟的。Graph

中的所有 filter 都同步于时钟。特别的，Renderer filter 还要根据参考时钟的时间来决定每一个 sample 的 Presentation 时间。

通常的情况下，应用程序是没有必要重新设置 graph 管理器选择好的参考时钟的。但是，如果你想修改参考时钟，你可以通过 graph 管理器提供的 **IMediaFilter::SetSyncSource** 方法来重新设置参考时钟。这个方法的参数是一个时钟的 **IReferenceClock** 接口指针。可以在 graph 停止的时候调用这个函数，下面的例子演示了如何指定一个时钟

```
IGraphBuilder *pGraph = 0;
IReferenceClock *pClock = 0;

CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
      IID_IGraphBuilder, (void **)&pGraph);

// Build the graph.
pGraph->RenderFile(L"C:\\Example.avi", 0);

// Create your clock.
hr = CreateMyPrivateClock(&pClock);
if (SUCCEEDED(hr))
{
      // Set the graph clock.
      IMediaFilter *pMediaFilter = 0;
      pGraph->QueryInterface(IID_IMediaFilter, (void**)&pMediaFilter);
      pMediaFilter->SetSyncSource(pClock);
      pClock->Release();
      pMediaFilter->Release();
}
```

这段代码假定 CreateMyPrivateClock 是应用程序定义的一个函数，用来创建一个时钟，然后返回一个 IReferenceClock 接口。

你也可以在 graph 没有设置时钟的情况下运行 graph。当 SetSyncSource 函数的参数为 NULL 的时候就给 graph 设置了一个空的参考时钟。如果 graph 没有时钟，graph 将运行的快许多。因为 renderer 不用再按照 sample 的 presentation 时间了，只要 sample 到达了 renderer filter，就可以立即被提交。所以，当你想处理数据尽可能快，而不是还要考虑预览的实际时间，你就可以给 graph 设置一个空的时间。

2.2.8 在 Dshow 中如何调试

2.3 音频的捕捉

Directshow 通过 Audio Capture Filter 可以捕获声卡上的音频流的输入。如果设备的驱动程序支持 waveInXXX 函数族，那么这个 filter 利用 SDK **waveInXXX** APIs 来控制该设备。系统中的

每个声卡都可以通过一个 filter 来进行访问。
音频捕捉 Filter 用输入 pin 来代替声卡的输入设备，比如麦克风，和 MIDI 输入。
这个输入 pin 中没有数据的流动，并且他们也不和其他的 filter 连接。应用程序可以通过他们控制输入。只要驱动程序允许，应用程序可以通过输入 pin 来控制输入。如果想要做的更好一些，你可能需要声卡制造商的文档和帮助。
如何建立音频的 graph 图
因为音频捕捉的 filter 和特定的硬件设备通信，因此，你不能简单地调用 **CoCreateInstance** 来创建一个 filter。你应该使用枚举系统设备的方法来枚举系统中所有的音频捕捉设备。
系统枚举返回一个设备支持的 moniker 集合，每个 moniker 的名字和设备的名字相关。然后从返回的 moniker 集合中选择一个合适的 moniker，利用这个 moniker 生成一个音频捕捉 filter 的实例，然后将 filter 添加到 graph 中。

```
IBaseFilter *pSrc = NULL, *pWaveDest = NULL, *pWriter = NULL;
IFileSinkFilter *pSink= NULL;
IGraphBuilder *pGraph;
// Create the Filter Graph Manager.
CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
      IID_IGraphBuilder, (void**)&pGraph);

// Not shown: Use the System Device Enumerator to create the
// audio capture filter.

// Add the audio capture filter to the filter graph.
pGraph->AddFilter(pSrc, L"Capture");

// Add the WavDest and the File Writer.
AddFilterByCLSID(pGraph, CLSID_WavDest, L"WavDest", &pWavDest);
AddFilterByCLSID(pGraph, CLSID_FileWriter, L"File Writer", &pWriter);

// Set the file name.
pWriter->QueryInterface(IID_IFileSinkFilter, (void**)&pSink);
pSink->SetFileName(L"C:\\MyWavFile.wav", NULL);

// Connect the filters.
ConnectFilters(pGraph, pSrc, pWavDest);
ConnectFilters(pGraph, pWavDest, pWriter);
```

2.4 视频的捕捉（Video Capture）

2.4.1 关于视频捕捉（About Video Capture in Dshow）

1 视频捕捉 Graph 的构建

一个能够捕捉音频或者视频的 graph 图都称之为捕捉 graph 图。捕捉 graph 图比一般的文件回放 graph 图要复杂许多，dshow 提供了一个 Capture Graph Builder COM 组件使得捕捉

graph 图的生成更加简单。Capture Graph Builder 提供了一个 **ICaptureGraphBuilder2** 接口，这个接口提供了一些方法用来构建和控制捕捉 graph。

首先创建一个 Capture Graph Builder 对象和一个 graph manger 对象，然后用 filter graph manager 作参数，调用 **ICaptureGraphBuilder2::SetFiltergraph** 来初始化 Capture Graph Builder。看下面的代码把

```
HRESULT InitCaptureGraphBuilder(
   IGraphBuilder **ppGraph,    // Receives the pointer.
   ICaptureGraphBuilder2 **ppBuild    // Receives the pointer.
)
{
      if (!ppGraph || !ppBuild)
      {
            return E_POINTER;
      }
      IGraphBuilder *pGraph = NULL;
      ICaptureGraphBuilder2 *pBuild = NULL;

      // Create the Capture Graph Builder.
      HRESULT hr = CoCreateInstance(CLSID_CaptureGraphBuilder2, NULL,
            CLSCTX_INPROC_SERVER, IID_ICaptureGraphBuilder2, (void**)&pGraph);
      if (SUCCEEDED(hr))
      {
            // Create the Filter Graph Manager.
            hr = CoCreateInstance(CLSID_FilterGraph, 0, CLSCTX_INPROC_SERVER,
                  IID_IGraphBuilder, (void**)&pGraph);
            if (SUCCEEDED(hr))
            {
                  // Initialize the Capture Graph Builder.
                  pBuild->SetFiltergraph(pGraph);

                  // Return both interface pointers to the caller.
                  *ppBuild = pBuild;
                  *ppGraph = pGraph; // The caller must release both interfaces.
                  return S_OK;
            }
            else
            {
                  pBuild->Release();
            }
      }
      return hr; // Failed
}
```

2 视频捕捉的设备

现在许多新的视频捕捉设备都采用的是 WDM 驱动方法，在 WDM 机制中，微软提供了一个

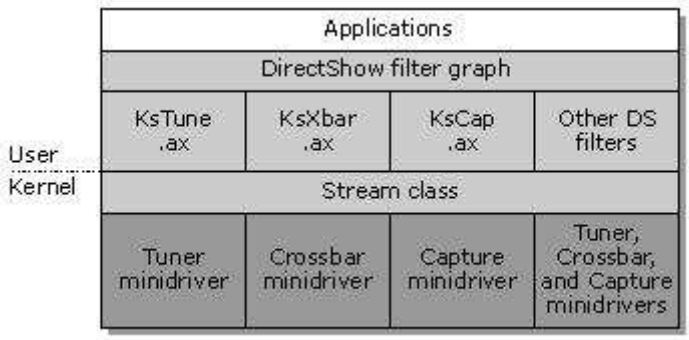
独立于硬件设备的驱动，称为类驱动程序。驱动程序的供应商提供的驱动程序称为 minidrivers。Minidrivers 提供了直接和硬件打交道的函数，在这些函数中调用了类驱动。

在 directshow 的 filter 图表中，任何一个 WDM 捕捉设备都是做为一个 WDM Video Capture 过滤器（Filter）出现。WDM Video Capture 过滤器根据驱动程序的特征构建自己的 filter

下面是别人提供的一篇有关于 dshow 和硬件的文章，可以拿来参考一下

大家知道，为了提高系统的稳定性，Windows 操作系统对硬件操作进行了隔离；应用程序一般不能直接访问硬件。DirectShow Filter 工作在用户模式（User mode，操作系统特权级别为 Ring 3），而硬件工作在内核模式（Kernel mode，操作系统特权级别为 Ring 0），那么它们之间怎么协同工作呢？

DirectShow 解决的方法是，**为这些硬件设计包装 Filter；这种 Filter 能够工作在用户模式下，外观、控制方法跟普通 Filter 一样，而包装 Filter 内部完成与硬件驱动程序的交互**。这样的设计，使得编写 DirectShow 应用程序的开发人员，从为支持硬件而需做出的特殊处理中解脱出来。DirectShow 已经集成的包装 Filter，包括 **Audio Capture Filter（qcap.dll）**、**VfW Capture Filter**（qcap.dll，Filter 的 Class Id 为 CLSID_VfwCapture）、**TV Tuner Filter**（KSTVTune.ax，Filter 的 Class Id 为 CLSID_CTVTunerFilter）、Analog Video Crossbar Filter（ksxbar.ax）、**TV Audio Filter**（Filter 的 Class Id 为 CLSID_TVAudioFilter）等；另外，DirectShow 为采用 WDM 驱动程序的硬件设计了 **KsProxy Filter**（Ksproxy.ax,）。我们来看一下结构图：



从上图中，我们可以看出，**Ksproxy.ax、Kstune.ax、Ksxbar.ax** 这些包装 Filter 跟其它普通的 DirectShow Filter 处于同一个级别，可以协同工作；**用户模式下的 Filter 通过 Stream Class 控制硬件的驱动程序 minidriver（由硬件厂商提供的实现对硬件控制功能的 DLL）**；Stream Class 和 minidriver 一起向上层提供系统底层级别的服务。值得注意的是，这里的 Stream Class 是一种驱动模型，它负责调用硬件的 minidriver；另外，Stream Class 的功能还在于协调 minidriver 之间的工作，使得一些数据可以直接在 Kernel mode 下从一个硬件传输到另一个硬件（或同一个硬件上的不同功能模块），提高了系统的工作效率。（更多的关于底层驱动程序的细节，请读者参阅 Windows DDK。）

下面，我们分别来看一下几种常见的硬件。

VfW 视频采集卡。这类硬件在市场上已经处于一种淘汰的趋势；新生产的视频采集卡一般采用 WDM 驱动模型。但是，DirectShow 为了保持向后兼容，还是专门提供了一个包装 Filter 支持这种硬件。和其他硬件的包装 Filter 一样，这种包装 Filter 的创建不是像普通 Filter 一样使用 CoCreateInstance，而要通过系统枚举，然后 BindToObject。

音频采集卡（声卡）。声卡的采集功能也是通过包装 Filter 来实现的；而且现在的声卡大部分都有混音的功能。这个 Filter 一般有几个 Input pin，每个 pin 都代表一个输入，如 Line In、Microphone、CD、MIDI 等。值得注意的是，这些 pin 代表的是声卡上的物理输入端子，在 Filter Graph 中是永远不会连接到其他 Filter 上的。声卡的输出功能，可以有两个 Filter 供选择：DirectSound Renderer Filter 和 Audio Renderer (WaveOut) Filter。注意，这两个 Filter 不是上述意义上的包装 Filter，它们能够同硬件交互，是因为它们使用了 API 函数：前者使用了 DirectSound API，后者使用了 waveOut API。这两个 Filter 的区别，还在于后者输出音频的同时不支持混音。（顺便说明一下，Video Renderer Filter 能够访问显卡，也是因为使用了 GDI、DirectDraw 或 Direct3D API。）如果你的机器上有声卡的话，你可以通过 GraphEdit，在 Audio Capture Sources 目录下看到这个声卡的包装 Filter。

WDM 驱动的硬件（包括视频捕捉卡、硬件解压卡等）。这类硬件都使用 Ksproxy.ax 这个包装 Filter。Ksproxy.ax 实现了很多功能，所以有“瑞士军刀”的美誉；它还被称作为“变色龙 Filter”，因为该 Filter 上定义了统一的接口，而接口的实现因具体的硬件驱动程序而异。在 Filter Graph 中，Ksproxy Filter 显示的名字为硬件的 Friendly name（一般在驱动程序的.inf 文件中定义）。我们可以通过 GraphEdit，在 WDM Streaming 开头的目录中找到本机系统中安装的 WDM 硬件。因为 KsProxy.ax 能够代表各种 WDM 的音视频设备，所以这个包装 Filter 的工作流程有点复杂。这个 Filter 不会预先知道要代表哪种类型的设备，它必须首先访问驱动程序的属性集，然后动态配置 Filter 上应该实现的接口。当 Ksproxy Filter 上的接口方法被应用程序或其他 Filter 调用时，它会将调用方法以及参数传递给驱动程序，由驱动程序最终完成指定功能。除此以外，WDM 硬件还支持内核流（Kernel Streaming），即内核模式下的数据传输，而无需经过到用户模式的转换。因为内核模式与用户模式之间的相互转换，需要花费很大的计算量。如果使用内核流，不仅可以避免大量的计算，还避免了内核数据与主机内存之间的拷贝过程。在这种情况下，用户模式的 Filter Graph 中，即使 pin 之间是连接的，也不会有实际的数据流动。典型的情况，如带有 Video Port Pin 的视频捕捉卡，Preview 时显示的图像就是在内核模式下直接传送到显卡的显存的。所以，你也休想在 VP Pin 后面截获数据流。

讲到这里，我想大家应该对 DirectShow 对硬件的支持问题有了一个总体的认识。对于应用程序开发人员来说，这方面的内容不用研究得太透，而只需作为背景知识了解一下就好了。其实，大量繁琐的工作 DirectShow 已经帮我们做好了。

Direcshow 中视频捕捉的 Filter

Pin 的种类

捕捉 Filter 一般都有两个或多个输出 pin，他们输出的媒体类型都一样，比如预览 pin 和捕捉 pin，因此根据媒体类型就不能很好的区别这些 pin。此时就要根据 pin 的功能来区别每个 pin 了，每个 pin 都有一个 GUID，称为 pin 的种类。

如果想仔细的了解 pin 的种类，请看后面的相关内容 Working with Pin Categories。对于大多数的应用来说，**ICaptureGraphBuilder2** 提供了一些函数可以自动确定 pin 的种类。

预览 pin 和捕捉 pin

视频捕捉 Filter 都提供了预览和捕捉的输出 pin，预览 pin 用来将视频流在屏幕上显示，捕捉 pin 用来将视频流写入文件。

预览 pin 和输出 pin 有下面的区别：

1 为了保证捕捉 pin 对视频桢流量，预览 pin 必要的时候可以停止。

2 经过捕捉 pin 的视频桢都有时间戳，但是预览 pin 的视频流没有时间戳。

预览 pin 的视频流之所以没有时间戳的原因在于 filter 图表管理器在视频流里加一个很小的

latency，如果捕捉时间被认为就是 render 时间的话，视频 renderFilter 就认为视频流有一个小小的延迟，如果此时 render filter 试图连续播放的时候，就会丢桢。去掉时间戳就保证了视频桢来了就可以播放，不用等待，也不丢桢。
预览 pin 的种类 GUID 为 PIN_CATEGORY_PREVIEW
捕捉 pin 的种类 GUID 为 PIN_CATEGORY_CAPTURE
Video Port pin
Video Port 是一个介于视频设备（TV）和视频卡之间的硬件设备。同过 **Video Port**，视频数据可以直接发送到图像卡上，通过硬件的覆盖，视频可以直接在屏幕显示出来。**Video Port** 就是连接两个设备的。
使用 **Video Port** 的最大好处是，不用 CPU 的任何工作，视频流直接写入内存中。当然它也有下面的缺点 drawbacks：
略
如果捕捉设备使用了 **Video Port**，捕捉 Filter 就用一个 video port pin 代替预览 pin。
video port pin 的种类 GUID 为 PIN_CATEGORY_VIDEOPORT
一个捕捉 filter 至少有一个 Capture pin，另外，它可能有一个预览 pin 和一个 video port pin，或者两者都没有，也许 filter 有很多的 capture pin，和预览 pin，每一个 pin 都代表一种媒体类型，因此一个 filter 可以有一个视频 capture pin，视频预览 pin，音频捕捉 pin，音频预览 pin。
Upstream WDM Filters
在捕捉 Filter 之上，WDM 设备可能需要额外的 filters，下面就是这些 filter
TV Tuner Filter
TV Audio Filter.
Analog Video Crossbar Filter
尽管这些都是一些独立的 filter，但是他们可能代表的是同一个硬件设备，每个 filter 都控制设备的不同函数，这些 filter 通过 pin 连接起来，但是在 pin 中没有数据流动。因此，这些 pin 的连接和媒体类型无关。他们使用一个 GUID 值来定义一个给定设备的 minidriver，例如：TV tuner Filter 和 video capture filter 都支持同一种 medium。
在实际应用中，如果你使用 ICaptureGraphBuilder2 来创建你的 capture graphs，这些 filters 就会自动被添加到你的 graph 中。更多的详细资料，可以参考 WDM Class Driver Filters

2.4.2 选择一个视频捕捉设备（Select capture device）

如何选择一个视频捕捉设备，可以采用系统设备枚举，详细资料参见 Using the System Device Enumerator 。enumerator 可以根据 filter 的种类返回一个设备的 monikers。Moniker 是一个 com 对象，可以参见 IMoniker 的 SDK。
对于捕捉设备，下面两种类是相关的。
CLSID_AudioInputDeviceCategory 音频设备
CLSID_VideoInputDeviceCategory 视频设备
下面的代码演示了如何枚举一个视频捕捉设备

```
ICreateDevEnum *pDevEnum = NULL;
IEnumMoniker *pEnum = NULL;

// Create the System Device Enumerator.
HRESULT hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL,
    CLSCTX_INPROC_SERVER, IID_ICreateDevEnum,
```

```
        reinterpret_cast<void**>(&pDevEnum));
if (SUCCEEDED(hr))
{
        //创建一个枚举器，枚举视频设备
        hr = pDevEnum->CreateClassEnumerator( CLSID_VideoInputDeviceCategory,
                &pEnum, 0);
}
```

IEnumMoniker 接口 pEnum 返回一个 IMoniker 接口的列表，代表一系列的 moniker，你可以显示所有的设备，然后让用户选择一个。

采用 **IMoniker::BindToStorage** 方法，返回一个 IPropertyBag 接口指针。然后调用 **IPropertyBag::Read** 读取 moniker 的属性。下面看看都包含什么属性

1 FriendlyName 是设备的名字

2 Description 属性仅仅适用于 DV 和 D-VHS/MPEG 摄象机，如果这个属性可用，这个属性更详细的描述了设备的资料

3DevicePath 这个属性是不可读的，但是每个设备都有一个独一无二的。你可以用这个属性来区别同一个设备的不同实例

下面的代码演示了如何显示遍历设备的名称 ，接上面的代码

```
HWND hList; // Handle to the list box.
IMoniker *pMoniker = NULL;
while (pEnum->Next(1, &pMoniker, NULL) == S_OK)
{
        IPropertyBag *pPropBag;
        hr = pMoniker->BindToStorage(0, 0, IID_IPropertyBag,
                (void**)(&pPropBag));
        if (FAILED(hr))
        {
                pMoniker->Release();
                continue;    // Skip this one, maybe the next one will work.
        }
        // Find the description or friendly name.
        VARIANT varName;
        VariantInit(&varName);
        hr = pPropBag->Read(L"Description", &varName, 0);
        if (FAILED(hr))
        {
                hr = pPropBag->Read(L"FriendlyName", &varName, 0);
        }
        if (SUCCEEDED(hr))
        {
                // Add it to the application's list box.
                USES_CONVERSION;
                (long)SendMessage(hList, LB_ADDSTRING, 0,
                        (LPARAM)OLE2T(varName.bstrVal));
                VariantClear(&varName);
```

```
    }
    pPropBag->Release();
    pMoniker->Release();
}
```

如果用户选中了一个设备调用 **IMoniker::BindToObject** 为设备生成 filter，然后将 filter 加入到 graph 中。

```
IBaseFilter *pCap = NULL;
hr = pMoniker->BindToObject(0, 0, IID_IBaseFilter, (void**)&pCap);
if (SUCCEEDED(hr))
{
    hr = m_pGraph->AddFilter(pCap, L"Capture Filter");
}
```

2.4.3 预览视频（Previewing Video）

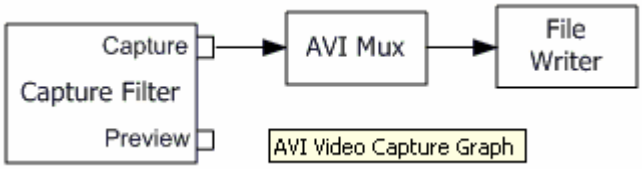
为了创建可以预览视频的 graph，可以调用下面的代码

```
ICaptureGraphBuilder2 *pBuild; // Capture Graph Builder
// Initialize pBuild (not shown).

IBaseFilter *pCap; // Video capture filter.
/* Initialize pCap and add it to the filter graph (not shown). */

hr = pBuild->RenderStream(&PIN_CATEGORY_PREVIEW, &MEDIATYPE_Video,
    pCap, NULL, NULL);
```

2.4.4 如何捕捉视频流并保存到文件（Capture video to File）

1 将视频流保存到 AVI 文件

下面的图表显示了 graph 图



AVI Mux filter 接收从 capture pin 过来的视频流，然后将其打包成 AVI 流。音频流也可以连接到 AVI Mux Filter 上，这样 mux filter 就将视频流和视频流合成 AVI 流。File writer 将 AVI 流写入到文件中。

可以像下面这样构建 graph 图

```
IBaseFilter *pMux;
hr = pBuild->SetOutputFileName(
    &MEDIASUBTYPE_Avi,    // Specifies AVI for the target file.
    L"C:\\Example.avi", // File name.
    &pMux,                    // Receives a pointer to the mux.
```

```
    NULL);                  // (Optional) Receives a pointer to the file sink.
```

第一个参数表明文件的类型，这里表明是 AVI，第二个参数是制定文件的名称。对于 AVI 文件，SetOutputFileName 函数会创建一个 AVI mux Filter 和一个 File writer Filter ，并且将两个 filter 添加到 graph 图中，在这个函数中，通过 File Writer Filter 请求 IFileSinkFilter 接口，然后调用 **IFileSinkFilter::SetFileName** 方法，设置文件的名称。然后将两个 filter 连接起来。第三个参数返回一个指向 AVI Mux 的指针，同时，它也通过第四个参数返回一个 **IFileSinkFilter** 参数，如果你不需要这个参数，你可以将这个参数设置成 NULL。

然后，你应该调用下面的函数将 capture filter 和 AVI Mux 连接起来。

```
hr = pBuild->RenderStream(
    &PIN_CATEGORY_CAPTURE, // Pin category.
    &MEDIATYPE_Video,      // Media type.
    pCap,                  // Capture filter.
    NULL,                  // Intermediate filter (optional).
    pMux);                 // Mux or file sink filter.
// Release the mux filter.
pMux->Release();
```

第 5 个参数就是使用的上面函数返回的 pMux 指针。

当捕捉音频的时候，媒体类型要设置为 MEDIATYPE_Audio，如果你从两个不同的设备捕捉视频和音频，你最好将音频设置成主流，这样可以防止两个数据流间 drift，因为 avi mux filter 为同步音频，会调整视频的播放速度的。为了设置 master 流，调用 **IConfigAviMux::SetMasterStream** 方法，可以采用如下的代码：

```
IConfigAviMux *pConfigMux = NULL;
hr = pMux->QueryInterface(IID_IConfigAviMux, (void**)&pConfigMux);
if (SUCCEEDED(hr))
{
    pConfigMux->SetMasterStream(1);
    pConfigMux->Release();
}
```

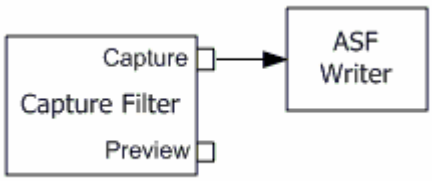
SetMasterStream 的参数指的是数据流的数目，这个是由调用 RenderStream 的次序决定的。例如，如果你调用 RenderStream 首先用于视频流，然后是音频，那么视频流就是 0，音频流就是 1。

添加编码 filter

```
IBaseFilter *pEncoder;
/* Create the encoder filter (not shown). */
// Add it to the filter graph.
pGraph->AddFilter(pEncoder, L"Encoder);

/* Call SetOutputFileName as shown previously. */

// Render the stream.
hr = pBuild->RenderStream(&PIN_CATEGORY_CAPTURE, &MEDIATYPE_Video,
    pCap, pEncoder, pMux);
pEncoder->Release();
```

2 将视频流保存成 wmv 格式的文件

为了将视频流保存成并编码成 windows media video （WMV）格式的文件，将 capture pin 连到 WM ASF Writer filter。



构建 graph 图最简单的方法就是将在 **ICaptureGraphBuilder2::SetOutputFileName** 方法中指定 **MEDIASUBTYPE_Asf** 的 filter。如下

```
IBaseFilter* pASFWriter = 0;
hr = pBuild->SetOutputFileName(
      &MEDIASUBTYPE_Asf,     // Create a Windows Media file.
      L"C:\\VidCap.wmv",     // File name.
      &pASFWriter,               // Receives a pointer to the filter.
      NULL);    // Receives an IFileSinkFilter interface pointer (optional).
```

参数 **MEDIASUBTYPE_Asf** 告诉 graph builder，要使用 wm asf writer 作为文件接收器，于是，**pbuild** 就创建这个 filter，将其添加到 graph 图中，然后调用 **IFileSinkFilter::SetFileName** 来设置输出文件的名字。第三个参数用来返回一个 ASF writer 指针，第四个参数用来返回文件的指针。

在将任何 pin 连接到 WM ASF Writer 之前，一定要对 WM ASF Writer 进行一下设置，你可以同过 WM ASF Writer 的 **IConfigAsfWriter** 接口指针来进行设置。

```
IConfigAsfWriter *pConfig = 0;
hr = pASFWriter->QueryInterface(IID_IConfigAsfWriter, (void**)&pConfig);
if (SUCCEEDED(hr))
{
       // Configure the ASF Writer filter.
      pConfig->Release();
}
```

然后调用 **ICaptureGraphBuilder2::RenderStream** 将 capture Filter 和 ASF writer 连接起来。

```
hr = pBuild->RenderStream(
      &PIN_CATEGORY_CAPTURE,     // Capture pin.
      &MEDIATYPE_Video,            // Video. Use MEDIATYPE_Audio for audio.
      pCap,                              // Pointer to the capture filter.
      0,
      pASFWriter);                    // Pointer to the sink filter (ASF Writer).
```

3 保存成自定义的文件格式

如果你想将文件保存成自己的格式，你必须有自己的 file writer。看下面的代码

```
IBaseFilter *pMux = 0;
IFileSinkFilter *pSink = 0;
hr = pBuild->SetOutputFileName(
&CLSID_MyCustomMuxFilter, //自己开发的 Filter
      L"C:\\VidCap.avi", &pMux, &pSink);
```

4 如何将视频流保存进多个文件

当你将视频流保存进一个文件后，如果你想开始保存第二个文件，这时，你应该首先将 graph 停止，然后通过 IFileSinkFilter::SetFileName 改变 File Writer 的文件名称。注意，**IFileSinkFilter 指针你可以在 SetOutputFileName 时通过第四个参数返回的。**

看看保存多个文件的代码吧

```
IBaseFilter *pMux;
IFileSinkFilter *pSink
hr = pBuild->SetOutputFileName(&MEDIASUBTYPE_Avi, L"C:\\YourFileName.avi",
    &pMux, &pSink);
if (SUCCEEDED(hr))
{
    hr = pBuild->RenderStream(&PIN_CATEGORY_CAPTURE, &MEDIATYPE_Video,
        pCap, NULL, pMux);

    if (SUCCEEDED(hr))
    {
        pControl->Run();
        /* Wait awhile, then stop the graph. */
        pControl->Stop();
        // Change the file name and run the graph again.
        pSink->SetFileName(L"YourFileName02.avi", 0);
        pControl->Run();
    }
    pMux->Release();
    pSink->Release();
}
```

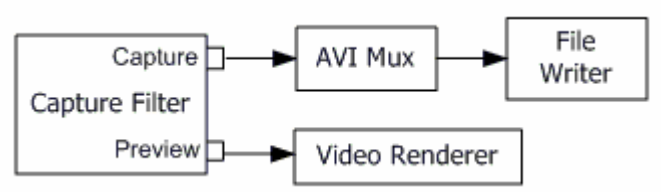
5 组合视频的捕捉和预览

如果想组建一个既可以预览视频，又可以将视频保存成文件的 graph，只需要两次调用 ICaptureGraphBuilder2::RenderStream 即可。代码如下：

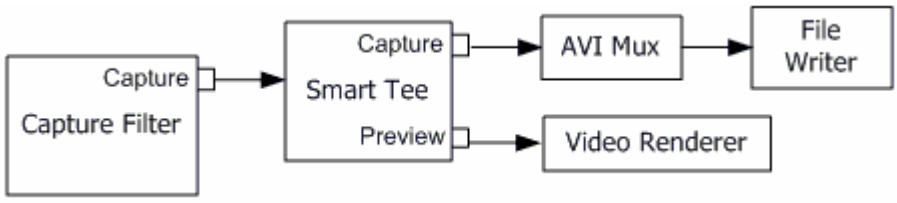
```
// Render the preview stream to the video renderer.
hr = pBuild->RenderStream(&PIN_CATEGORY_PREVIEW, &MEDIATYPE_Video, pCap,
NULL, NULL);

// Render the capture stream to the mux.
hr = pBuild->RenderStream(&PIN_CATEGORY_CAPTURE, &MEDIATYPE_Video, pCap,
NULL, pMux);
```

在上面的代码中，graph builder 其实隐藏了下面的细节。

1 如果 capture Filter 既有 preview pin 也有 captrue pin，那么 RenderStream 仅仅将两个 pin 和 render filter 接起来。如下图



2 如果 caprture Filter 只有一个 capture pin，那么 Capture Graph Builder 就采用一个 Smart Tee Filter 将视频流分流，graph 图如下



2.4.5 如何控制 Capture Graph（Controlling Capture Graph）

Filter 图表管理器可以通过 **IMediaControl** 接口控制整个 graph 的运行，停止和暂停。但是当一个 graph 有捕捉和预览两个数据流的时候，如果我们想单独的控制其中的一个数据流话，我们可以通过 **ICaptureGraphBuilder2::ControlStream** 。
下面讲一下如何来单独控制捕捉和预览数据流。

1 控制捕捉视频流

下面的代码，让捕捉数据流在 graph 开始运行 1 秒后开始，允运行 4 秒后结束。

```
// Control the video capture stream.
REFERENCE_TIME rtStart = 1000 0000, rtStop = 5000 0000;
const WORD wStartCookie = 1, wStopCookie = 2;    // Arbitrary values.
hr = pBuild->ControlStream(
      &PIN_CATEGORY_CAPTURE, // Pin category.
      &MEDIATYPE_Video,          // Media type.
      pCap,                          // Capture filter.
      &rtStart, &rtStop,         // Start and stop times.
      wStartCookie, wStopCookie    // Values for the start and stop events.
);
pControl->Run();
```

第一个参数表明需要控制的数据流，一般采用的是 pin 种类 GUID，
第二个参数表明了媒体类型。
第三个参数指明了捕捉的 filter。如果想要控制 graph 图中的所有捕捉 filter，第二个和第三个参数都要设置成 NULL。
第四和第五个参数表明了流开始和结束的时间，这是一个相对于 graph 开始的时间。
只有你调用 **IMediaControl::Run** 以后，这个函数才有作用。如果 graph 正在运行，这个设置立即生效。
最后的两个参数用来设置当数据流停止，开始能够得到的事件通知。对于任何一个运用此方法的数据流，graph 当流开始的时候，会发送 **EC_STREAM_CONTROL_STARTED** 通知，在流结束的时候，要发送 **EC_STREAM_CONTROL_STOPPED** 通知。*wStartCookie* 和 *wStopCookie* 是作为第二个参数的。

看看事件通知处理过程吧

```
while (hr = pEvent->GetEvent(&evCode, &param1, &param2, 0), SUCCEEDED(hr))
{
    switch (evCode)
    {
    case EC_STREAM_CONTROL_STARTED:
    // param2 == wStartCookie
    break;

    case EC_STREAM_CONTROL_STOPPED:
    // param2 == wStopCookie
    break;

    }
    pEvent->FreeEventParams(evCode, param1, param2);
}
```

ControlStream 还定义了一些特定的值来表示开始和停止的时间。

MAXLONGLONG 从不开始，只有在 graph 停止的时候才停止

NULL， 立即开始和停止

例如，下面的代码立即停止捕捉流。

```
pBuild->ControlStream(&PIN_CATEGORY_CAPTURE, &MEDIATYPE_Video, pCap,
    0, 0,       // Start and stop times.
    wStartCookie, wStopCookie);
```

2 控制预览视频流

只要给 ControlStream 第一个参数设置成 PIN_CATEGORY_PREVIEW 就可以控制预览 pin，整个函数的使用和控制捕捉流一样，但是唯一区别是在这里你没法设置开始和结束时间了，因为预览的视频流没有时间戳，因此你必须使用 NULL 或者 MAXLONGLONG。例子

```
Use NULL to start the preview stream:
pBuild->ControlStream(&PIN_CATEGORY_PREVIEW, &MEDIATYPE_Video, pCap,
    NULL,       // Start now.
    0,          // (Don't care.)
    wStartCookie, wStopCookie);
Use MAXLONGLONG to stop the preview stream:
pBuild->ControlStream(&PIN_CATEGORY_PREVIEW, &MEDIATYPE_Video, pCap,
    0,                  // (Don't care.)
    MAXLONGLONG,        // Stop now.
    wStartCookie, wStopCookie);
```

3 关于数据流的控制

Pin 的缺省的行为是传递 sample，例如，如果你对 PIN_CATEGORY_CAPTURE 使用了 ControlStream，但是对于 PIN_CATEGORY_PREVIEW 没有使用该函数，因此，当你 run graph 的时候，preview 流会立即运行起来，而 capture 流则要等到你设置的时间运行。

2.4.6 视频捕捉的任务（Video Capture Tasks）

2.4.6.1 如何配置一个视频捕捉设备

1 显示 VFW 驱动的视频设备对话框

如果视频捕捉设备采用的仍然是 VFW 方式的驱动程序，则必须支持下面三个对话框，用来设置视频设备。

1 Video Source

用来选择视频输入设备并且调整设备的设置，比如亮度和对比度。

2Video Format

用来设置桢的大小和位

3Video Display

用来设置视频的显示参数

为了显示上面的三个对话框，你可以 do the following

1 停止 graph。

2 向捕捉 filter 请求 **IAMVfwCaptureDialogs** 接口，如果成功，表明设备支持 VFW 驱动。

3 调用 **IAMVfwCaptureDialogs::HasDialog** 来检查驱动程序是否支持你请求的对话框，如果支持，返回 S_OK,否则返回 S_FALSE。注意不要用 SUCCEDED 宏。

4 如果驱动支持该对话框，调用 **IAMVfwCaptureDialogs::ShowDialog** 显示该对话框。

5 重新运行 graph

代码如下

```
pControl->Stop(); // Stop the graph.
// Query the capture filter for the IAMVfwCaptureDialogs interface.
IAMVfwCaptureDialogs *pVfw = 0;
hr = pCap->QueryInterface(IID_IAMVfwCaptureDialogs, (void**)&pVfw);
if (SUCCEEDED(hr))
{
    // Check if the device supports this dialog box.
    if (S_OK == pVfw->HasDialog(VfwCaptureDialog_Source))
    {
        // Show the dialog box.
        hr = pVfw->ShowDialog(VfwCaptureDialog_Source, hwndParent);
    }
}
pControl->Run();
```

2 调整视频的质量

WDM 驱动的设备支持一些属性可以用来调整视频的质量，比如亮度，对比度，饱和度，等要向调整视频的质量，do the following

1 从捕捉 filter 上请求 **IAMVideoProcAmp** 接口

2 对于你想调整的任何一个属性，调用 **IAMVideoProcAmp::GetRange** 可以返回这个属性赋值的范围，缺省值，最小的增量值。**IAMVideoProcAmp::Get** 返回当前正在使用的值。**VideoProcAmpProperty** 枚举每个属性定义的标志。

3 调用 **IAMVideoProcAmp::Set** 来设置这个属性值。设置属性的时候，不用停止 graph。
看看下面的代码是如何调整视频的质量的

```
HWND hTrackbar; // Handle to the trackbar control.
// Initialize hTrackbar (not shown).

// Query the capture filter for the IAMVideoProcAmp interface.
IAMVideoProcAmp *pProcAmp = 0;
hr = pCap->QueryInterface(IID_IAMVideoProcAmp, (void**)&pProcAmp);
if (FAILED(hr))
{
    // The device does not support IAMVideoProcAmp, so disable the control.
    EnableWindow(hTrackbar, FALSE);
}
else
{
    long Min, Max, Step, Default, Flags, Val;

    // Get the range and default value.
    hr = m_pProcAmp->GetRange(VideoProcAmp_Brightness, &Min, &Max, &Step,
        &Default, &Flags);
    if (SUCCEEDED(hr))
    {
        // Get the current value.
        hr = m_pProcAmp->Get(VideoProcAmp_Brightness, &Val, &Flags);
    }
    if (SUCCEEDED(hr))
    {
        // Set the trackbar range and position.
        SendMessage(hTrackbar, TBM_SETRANGE, TRUE, MAKELONG(Min, Max));
        SendMessage(hTrackbar, TBM_SETPOS, TRUE, Val);
        EnableWindow(hTrackbar, TRUE);
    }
    else
    {
        // This property is not supported, so disable the control.
        EnableWindow(hTrackbar, FALSE);
    }
}
```

3 调整视频输出格式

我们知道视频流可以有多种输出格式，一个设备可以支持 16-bit RGB, 32-bit RGB, and YUYV，在每一种格式下，设备还可以调整视频桢的大小。
在 WDM 驱动设备上，**IAMStreamConfig** 接口用来报告设备输出视频的格式的，VFW 设备，可以采用对话框的方式来设置，参见前面的内容。
捕捉 Filter 的捕捉 pin 和预览 pin 都支持 **IAMStreamConfig** 接口，可以通过

ICaptureGraphBuilder2::FindInterface 获得 IAMStreamConfig 接口。

```
IAMStreamConfig *pConfig = NULL;
hr = pBuild->FindInterface(
    &PIN_CATEGORY_PREVIEW, // Preview pin.
    0,      // Any media type.
    pCap, // Pointer to the capture filter.
    IID_IAMStreamConfig, (void**)&pConfig);
```

设备还支持一系列的媒体类型，对于每一个媒体类型，设备都要支持一系列的属性，比如，桢的大小，图像如何缩放，桢率的范围等。

通过 IAMStreamConfig::GetNumberOfCapabilities 获得设备所支持的媒体类型的数量。这个方法返回两个值，一个是媒体类型的数量，二是属性所需结构的大小。

这个结构的大小很重要，因为这个方法是用于视频和音频的，视频采用的是 VIDEO_STREAM_CONFIG_CAPS 结构，音频用 AUDIO_STREAM_CONFIG_CAPS 结构。

通过函数 IAMStreamConfig::GetStreamCaps 来枚举媒体类型，要给这个函数传递一个序号作为参数，这个函数返回媒体类型和相应的属性结构体。看代码把

```
int iCount = 0, iSize = 0;
hr = pConfig->GetNumberOfCapabilities(&iCount, &iSize);

// Check the size to make sure we pass in the correct structure.
if (iSize == sizeof(VIDEO_STREAM_CONFIG_CAPS)
{
    // Use the video capabilities structure.
    for (int iFormat = 0; iFormat < iCount; iFormat++)
    {
        VIDEO_STREAM_CONFIG_CAPS scc;
        AM_MEDIA_TYPE *pmtConfig;
        hr = pConfig->GetStreamCaps(iFormat, &pmtConfig, (BYTE*)&scc);
        if (SUCCEEDED(hr))
        {
            /* Examine the format, and possibly use it. */
            // Delete the media type when you are done.
            hr = pConfig->SetFormat(pmtConfig);//重新设置视频格式
            DeleteMediaType(pmtConfig);
        }
}
```

你可以调用 IAMStreamConfig::SetFormat 设置新的媒体类型

```
hr = pConfig->SetFormat(pmtConfig);
```

如果 pin 没有连接，当连接的时候就试图用新的格式，如果 pin 已经在连接了，它就会用的新的媒体格式重新连接。在任何一种情况下，下游的 filter 都有可能拒绝新的媒体格式。

在 SetFormat 前你可以修改 **VIDEO_STREAM_CONFIG_CAPS** 结构来重新设置媒体类型。

例如：

如果 **GetStreamCaps** 返回的是 24-bit RGB format，桢的大小是 320 x 240 像素，你可以通过检查媒体类型的 major type，subtpye，和 format 等值

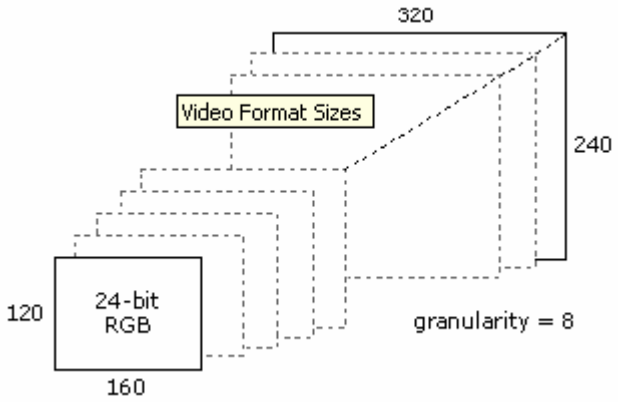
```
if ((pmtConfig.majortype == MEDIATYPE_Video) &&
```

```
    (pmtConfig.subtype == MEDIASUBTYPE_RGB24) &&
    (pmtConfig.formattype == FORMAT_VideoInfo) &&
    (pmtConfig.cbFormat >= sizeof (VIDEOINFOHEADER)) &&
    (pmtConfig.pbFormat != NULL))
{
    VIDEOINFOHEADER *pVih = (VIDEOINFOHEADER*)pmtConfig.pbFormat;
    // pVih contains the detailed format information.
    LONG lWidth = pVih->bmiHeader.biWidth;
    LONG lHeight = pVih->bmiHeader.biHeight;
}
```

VIDEO_STREAM_CONFIG_CAPS 结构里包含了该媒体类型的视频长度和宽度的最大值和最小值，还有递增的幅度值，就是每次调整视频 size 的幅度，例如，设备可能返回如下的值

- MinOutputSize: 160 x 120
- MaxOutputSize: 320 x 240
- OutputGranularityX: 8 pixels (horizontal step size)
- OutputGranularityY: 8 pixels (vertical step size)

这样你可以在(160, 168, 176, ... 304, 312, 320) 范围内设置宽度，在 (120, 128, 136, ... 104, 112, 120).设置高度值，



如果想设置新的值，直接修改在 **GetStreamCaps** 函数中返回的值即可，

```
pVih->bmiHeader.biWidth = 160;
pVih->bmiHeader.biHeight = 120;
pVih->bmiHeader.biSizeImage = DIBSIZE(pVih->bmiHeader);
```

然后将媒体类型传递给 SetFormat 函数，就可修改视频格式了。

2.4.6.2Working With Crossbars

2.4.6.3 将设备从系统中移走时的事件通知（Device remove Notify）

如果用户将一个 graph 正在使用的即插即用型的设备从系统中去掉，filter 图表管理器就会发送一个 **EC_DEVICE_LOST** 事件通知，如果该设备又可以使用了，filter 图表管理器就发送另

外的一个 **EC_DEVICE_LOST** 通知，但是先前组建的捕捉 filter graph 图就没法用了，用户必须重新组建 graph 图。
当系统中有新的设备添加时，dshow 是不会发送任何通知的，所以，应用程序如果想要知道系统中何时添加新的设备，应用程序可以监控 WM_DEVICECHANGE 消息。

2.4.6.4 从静止图像 pin 中捕捉图片

有些照相机，摄像头除了可以捕获视频流以外还可以捕获单张的，静止的图片。通常，静止的图片的质量要比流的质量要高。摄像头一般都一个按钮来触发，或者是支持软件触发。支持输出静态图片的摄像头一般都要提供一个静态图片 pin，这个 pin 的种类是 PIN_CATEGORY_STILL。

从设备中获取静态图片，我们一般推荐使用 **windows Image Acquisition (WIA) APIs**。当然，你也可以用 dshow 来获取图片。
在 graph 运行的时候利用 **IAMVideoControl::SetMode** 来触发静态的 pin。代码如下

```
pControl->Run(); // Run the graph.
IAMVideoControl *pAMVidControl = NULL;
hr = pCap->QueryInterface(IID_IAMVideoControl, (void**)&pAMVidControl);
if (SUCCEEDED(hr))
{
    // Find the still pin.
    IPin *pPin = 0;
    hr = pBuild->FindPin(pCap, PINDIR_OUTPUT, &PIN_CATEGORY_STILL, 0,
        FALSE, 0, &pPin);
    if (SUCCEEDED(hr))
    {
        hr = pAMVidControl->SetMode(pPin, VideoControlFlag_Trigger);
        pPin->Release();
    }
    pAMVidControl->Release();
}
```

首先向 capture Filter 请求 IAMVideoContol，如果支持该接口，就调用 **ICaptureGraphBuilder2::FindPin** 请求指向静止 pin 的指针，然后调用 pin 的 put_Mode 方法。根据不同的摄像头，你可能静态 pin 连接前要 render 该 pin。
捕捉静态图片常用的 filter 是 Sample Grabber filter，Sample Grabber 使用了一个用户定义的回调汗水来处理图片。关于这个 filter 的详细用法，参见 Using the Sample Grabber.。
下面的例子假设静态 pin 传递的是没有压缩的 RGB 图片。首先定义一个类，从 **ISampleGrabberCB** 继承。

```
// Class to hold the callback function for the Sample Grabber filter.
class SampleGrabberCallback : public ISampleGrabberCB
{
    // Implementation is described later.
}

// Global instance of the class.
```

```
SampleGrabberCallback g_StillCapCB;
```

然后将捕捉 filter 的静态 pin 连接到 Sample Grabber，将 Sample Grabber 连接到 Null Renderer filter。Null Renderer 仅仅是将她接收到的 sample 丢弃掉。实际的工作都是在回调函数里进行，连接 Null Renderer 仅仅是为了给 Sample Grabber's 输出 pin 上连接点东西。具体见下面的代码

```
// Add the Sample Grabber filter to the graph.
IBaseFilter *pSG_Filter;
hr = CoCreateInstance(CLSID_SampleGrabber, NULL, CLSCTX_INPROC_SERVER,
      IID_IBaseFilter, (void**)&pSG_Filter);
hr = pGraph->AddFilter(pSG_Filter, L"SampleGrab");

// Add the Null Renderer filter to the graph.
IBaseFilter *pNull;
hr = CoCreateInstance(CLSID_NullRendere, NULL, CLSCTX_INPROC_SERVER,
      IID_IBaseFilter, (void**)&pNull);
hr = pGraph->AddFilter(pSG_Filter, L"NullRender");
```

然后通过 RenderStream 将 still pin ， sample grabber ， null Renderer 连接起来

```
hr = pBuild->RenderStream(
      &PIN_CATEGORY_STILL, // Connect this pin ...
      &MEDIATYPE_Video,      // with this media type ...
      pCap,                  // on this filter ...
      pSG_Filter,            // to the Sample Grabber ...
      pNull);                // ... and finally to the Null Renderer.
```

然后调用 ISampleGrabber 指针，来通过这个指针可以分配内存。

```
// Configure the Sample Grabber.
ISampleGrabber *pSG;
hr = pSG_Filter->QueryInterface(IID_ISampleGrabber, (void**)&pSG);
pSG->SetOneShot(FALSE);
pSG->SetBufferSamples(TRUE);
```

设置你的回调对象

```
pSG->SetCallback(&g_StillCapCB, 0); // 0 = Use the SampleCB callback method
```

获取静态 pin 和 sample grabber 之间连接所用的媒体类型

```
// Store the media type for later use.
AM_MEDIA_TYPE g_StillMediaType;
hr = pSG->GetConnectedMediaType(&g_StillMediaType);
pSG->Release();
```

媒体类型包含一个 **BITMAPINFOHEADER** 结构来定义图片的格式，在程序退出前一定要释放媒体类型

```
// On exit, remember to release the media type.
FreeMediaType(g_StillMediaType);
```

看看下面的回调类吧。这个类从 **ISampleGrabber** 接口派生，但是它没有保持引用计数，因为应用程序在堆上创建这个对象，在整个 graph 的生存周期它都存在。

所有的工作都在 BufferCB 函数里完成，当有一个新的 sample 到来的时候，这个函数就会被 sample Grabber 调用到。在下面的例子里，bitmap 被写入到一个文件中

```
class SampleGrabberCallback : public ISampleGrabberCB
{
public:
    // Fake referance counting.
    STDMETHODIMP_(ULONG) AddRef() { return 1; }
    STDMETHODIMP_(ULONG) Release() { return 2; }

    STDMETHODIMP QueryInterface(REFIID riid, void **ppvObject)
    {
        if (NULL == ppvObject) return E_POINTER;
        if (riid == __uuidof(IUnknown))
        {
            *ppvObject = static_cast<IUnknown*>(this);
             return S_OK;
        }
        if (riid == __uuidof(ISampleGrabberCB))
        {
            *ppvObject = static_cast<ISampleGrabberCB*>(this);
             return S_OK;
        }
        return E_NOTIMPL;
    }

    STDMETHODIMP SampleCB(double Time, IMediaSample *pSample)
    {
        return E_NOTIMPL;
    }

    STDMETHODIMP BufferCB(double Time, BYTE *pBuffer, long BufferLen)
    {
        if ((g_StillMediaType.majortype != MEDIATYPE_Video) ||
            (g_StillMediaType.formattype != FORMAT_VideoInfo) ||
            (g_StillMediaType.cbFormat < sizeof(VIDEOINFOHEADER)) ||
            (g_StillMediaType.pbFormat == NULL))
        {
            return VFW_E_INVALIDMEDIATYPE;
        }
        HANDLE hf = CreateFile("C:\\Example.bmp", GENERIC_WRITE,
        FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, 0, NULL);
        if (hf == INVALID_HANDLE_VALUE)
        {
            return E_FAIL;
        }
        long cbBitmapInfoSize = g_StillMediaType.cbFormat - SIZE_PREHEADER;
```

```
        VIDEOINFOHEADER *pVideoHeader =
            (VIDEOINFOHEADER*)g_StillMediaType.pbFormat;

        BITMAPFILEHEADER bfh;
        ZeroMemory(&bfh, sizeof(bfh));
        bfh.bfType = 'MB';    // Little-endian for "MB".
        bfh.bfSize = sizeof( bfh ) + BufferLen + cbBitmapInfoSize;
        bfh.bfOffBits = sizeof( BITMAPFILEHEADER ) + cbBitmapInfoSize;

        // Write the file header.
        DWORD dwWritten = 0;
        WriteFile( hf, &bfh, sizeof( bfh ), &dwWritten, NULL );
        WriteFile(hf, HEADER(pVideoHeader), cbBitmapInfoSize, &dwWritten, NULL);
        WriteFile( hf, pBuffer, BufferLen, &dwWritten, NULL );
        CloseHandle( hf );
        return S_OK;

    }
};
```

2.4.7 数字视频 DV（Digital Video in Direcshow）

2.4.7.1 关于 Directshow 中的 DV 应用

2.4.7.2 如何将 DV 捕捉到一个文件中

2.4.7.3 如何将文件中的 DV 读入到盘中

2.4.7.4 DVINFO Field Settings in the MSDV Driver

2.4.8 如何控制 DV 便携式摄像机（Controlling a DV Camcorder）

2.4.9 模拟电视的视频捕捉（Analog Television）

2.4.10 视频捕捉的高级话题

2.4.10.1 处理视频重画事件

如果你没有使用 **ICaptureGraphBuilder2** 创建了自己的视频采集 graph，并且你使用老的 Video

Renderer filter 来预览视频，那么你应该重载一下缺省的事件处理，来处理 EC_REPAINT 事件，首先从 Filter 图表管理器请求 **IMediaEvent** 接口指针，然后用 EC_REPAINT 座参数调用 **IMediaEvent::CancelDefaultHandling**。

```
IMediaEvent *pEvent = 0;
hr = pGraph->QueryInterface(IID_IMediaEvent, (void**)&pEvent);
if (SUCCEEDED(hr))
{
    pEvent->CancelDefaultHandling (EC_REPAINT);
    pEvent->Release();
}
```

2.4.10.2 如何确定 pin 的种类（Pin Categories）

给定一个 pin 的种类，你可以通过 **ICaptureGraphBuilder2::FindPin** 在一个指定的 filter 上找到一个 pin 的接口。下面的代码就是要查找一个视频预览 pin

```
int i = 0;
hr = pBuild->FindPin(
    pFilter,                    // Pointer to a filter to search.
    PINDIR_OUTPUT,              // Which pin direction?
    &PIN_CATEGORY_PREVIEW, // Which category? (NULL means "any category")
    &MEDIATYPE_Video,           // What media type? (NULL means "any type")
    FALSE,                      // Must be connected?
    i,                          // Get the i'th matching pin (0 = first match)
    &pPin                       // Receives a pointer to the pin.
);
```

尽管 FindPin 这个方法还是比较方便，你还可以自己写一个类似的函数。可以通过 **IKsPropertySet::Get** 方法来获取 pin 的种类。

下面的代码如何确定一个 pin 是否属于某个种类。

```
BOOL PinMatchesCategory(IPin *pPin, const GUID& Category)
{
    if (pPin == NULL) return E_POINTER;

    BOOL bFound = FALSE;
    IKsPropertySet *pKs;
    HRESULT hr = pPin->QueryInterface(IID_IKsPropertySet, (void **)&pKs);
    if (SUCCEEDED(hr))
    {
        GUID PinCategory;
        DWORD cbReturned;
        hr = pKs->Get(AMPROPSETID_Pin, AMPROPERTY_PIN_CATEGORY, NULL, 0,
&PinCategory, sizeof(GUID), &cbReturned);
        if (SUCCEEDED(hr))
        {
            bFound = (PinCategory == Category);
```

```
            }
            pKs->Release();
        }
        return bFound;
}
```

下面的函数演示了如何根据种类查找 pin，有点类似于 FindPin

```
HRESULT FindPinByCategory(
        IBaseFilter *pF,          // Pointer to the filter to search.
        PIN_DIRECTION PinDir, // Direction of the pin.
        const GUID& Category, // Pin category.
        IPin **ppPin)             // Receives a pointer to the pin.
{
        if (!pF || !ppPin) return E_POINTER;

        *ppPin = 0;
        HRESULT hr;
        IEnumPins *pEnum = 0;
        if (SUCCEEDED(pF->EnumPins(&pEnum)))
        {
                IPin *pPin = 0;
                while (hr = pEnum->Next(1, &pPin, 0), hr == S_OK)
                {
                        PIN_DIRECTION ThisPinDir;
                        hr = pPin->QueryDirection(&ThisPinDir);
                        if (FAILED(hr))
                        {
                                pPin->Release();
                                pEnum->Release();
                                return E_UNEXPECTED; // Something strange happened.
                        }
                        if ((ThisPinDir == PinDir) && PinMatchesCategory(pPin, Category))
                        {
                                *ppPin = pPin;    // Caller must release the interface.
                                pEnum->Release();
                                return S_OK;
                        }
                        pPin->Release();
                }
                pEnum->Release();
        }
        // No pin matches.
        return E_FAIL;
}
```

下面的代码演示的是如何使用上面的函数在一个 filter 上搜索 video port pin。

```
IPin *pVP;
hr = FindPinByCategory(pFilter, PINDIR_OUTPUT,
     PIN_CATEGORY_VIDEOPORT, &pVP);
if (SUCCEEDED(hr))
{
     // Use pVP ...
     // Release when you are done.
     pVP->Release();
}
```

2.4.10.3 如何使用一个 SmartTee Filer

2.4.10.4 如何使用一个重叠混合器（Overlay Mixer in Video Capture）

2.4.10.5Video Port Pins

2.4.10.6VideoInfo2 Format Type

2.4.10.7 手动添加 WDM 类驱动 filter

如果一个硬件设备采用的是 WDM 驱动，在 graph 图表中，在 capture filter 的上游还应该有其他的 filter，这个 filter 就是流类驱动 filter 或者 WDM filter。这些 filter 支持硬件的每一个功能。例如，TV turer 卡有一个功能用来设置频道，那么在类 filter 相应的就有一个 TV Tuner filter 来执行这个功能。所以为了能够使用这个功能，你应该将 TV Tuner filter 连接到你的 capture filter 上。

ICaptureGraphBuilder2 接口提供了非常简单的方法，可以很轻松就将 WDM filter 添加到 graph 里。在创建 graph 的同时，调用 **FindInterface** or **RenderStream**。这两个函数中的任何一个都会自动将你所需要的 WDM filter 连接到你的 capture filter 上。下面我就讲一下如果手动的将 WDM filter 添加到你的 graph 中。不过，要记住最简单的方法还是调用 **ICaptureGraphBuilder2 接口函数**

WDM filter 上 pin 都支持一种或者几种媒体，每种媒体都定义了一种通信的方式，只有两个 pin 都支持同一种媒体类型你才能把他们连到一起。通常采用 **REGPINMEDIUM** 结构来定义一种媒体，这个结构等同于 KSPIN_MEDIUM，适用于内核流驱动。REGPINMEDIUM 结构中的 clsMedium 用来定义媒体的 CLSID，为了检查 pin 所支持的媒体类型，可以调用 **IKsPin::KsQueryMediums** 方法。这个方法返回一个指向包含 KSMULTIPLE_ITEM 结构的内存指针。在 KSMULTIPLE_ITEM 结构中包含一个或几个 REGPINMEDIUM 结构。每个 REGPINMEDIUM 用来标示该 pin 所支持的媒体类型。

如果媒体类型的 CLSID 为 **GUID_NULL** 或者 **KSMEDIUMSETID_Standard**，那么 pin 就没法连接。这些值表明 pin 不支持任何的媒体类型。

另，

Also, do not connect a pin unless the filter requires exactly one connected instance of that pin. Otherwise, your application might try to connect various pins that shouldn't have connections,

which can cause the program to stop responding. To find out the number of required instances, retrieve the KSPROPERTY_PIN_NECESSARYINSTANCES property set, as shown in the following code example. (For brevity, this example does not test any return codes or release any interfaces. Your application should do both, of course.)这段什么意思没有明白，
看下面的代码把

```
// Obtain the pin factory identifier.
IKsPinFactory *pPinFactory;
hr = pPin->QueryInterface(IID_IKsPinFactory, (void **)&pPinFactory);

ULONG ulFactoryId;
hr = pPinFactory->KsPinFactory(&ulFactoryId);

// Get the "instance" property from the filter.
IKsControl *pKsControl;
hr = pFilter->QueryInterface(IID_IKsControl, (void **)&pKsControl);

KSP_PIN ksPin;
ksPin.Property.Set = KSPROPSETID_Pin;
ksPin.Property.Id = KSPROPERTY_PIN_NECESSARYINSTANCES;
ksPin.Property.Flags = KSPROPERTY_TYPE_GET;
ksPin.PinId = ulFactoryId;
ksPin.Reserved = 0;

KSPROPERTY ksProp;
ULONG ulInstances, bytes;
pKsControl->KsProperty((PKSPROPERTY)&ksPin, sizeof(ksPin),
    &ulInstances, sizeof(ULONG), &bytes);

if (hr == S_OK && bytes == sizeof(ULONG))
{
    if (ulInstances == 1)
    {
        // Filter requires one instance of this pin.
        // This pin is OK.
    }
}
```

这段代码也没有明白，以后再看
看下面的一段伪代码演示了如何连接一个 WDM filter

```
Add supporting filters:
{
    foreach input pin:
        skip if (pin is connected)

        Get pin medium
```

```
            skip if (medium is GUID_NULL or KSMEDIUMSETID_Standard)

            Query filter for KSPROPERTY_PIN_NECESSARYINSTANCES property
            skip if (necessary instances != 1)

            Match an existing pin || Find a matching filter
}

Match an existing pin:
{
      foreach filter in the graph
            foreach unconnected pin
                  Get pin medium
                  if (mediums match)
                        connect the pins
}

Find a matching filter:
{
      Query the filter graph manager for IFilterMapper2.
      Find a filter with an output pin that matches the medium.
      Add the filter to the graph.
      Connect the pins.
      Add supporting filters. (Recursive call.)
}
```

2.4.10.8 如何创建内核 filter

一些内核模式的 filter 不能通过 **CoCreateInstance** 创建的，因此也没有 CLSID。这些 filter 包括 Tee/Sink-to-Sink Converter，CC Decoder filter，WST Codec filter.，要想创建这些 filter，必须用 System Device Enumerator 对象，利用 filter 的名字进行搜索。

步骤如下：

1 创建 System Device Enumerator。

2 利用该 filter 的种类 CLSID 做参数，调用 **ICreateDevEnum::CreateClassEnumerator**，这个方法回返一个枚举器，包括所有这个种类的 filter，提供一个 IMoniker 指针。

3 对于每个 moniker，通过 **IMoniker::BindToStorage** 获取 **IPropertyBag** 接口

4 调用 **IPropertyBag::Read** 读取 filter 的名字

5 如果名字匹配，就调用 **IMoniker::BindToObject** 来创建 filter。

看看这个函数把

```
HRESULT CreateKernelFilter(
      const GUID &guidCategory,    // Filter category.
      LPCOLESTR szName,                  // The name of the filter.
      IBaseFilter **ppFilter         // Receives a pointer to the filter.
)
```

```
{
    HRESULT hr;
    ICreateDevEnum *pDevEnum = NULL;
    IEnumMoniker *pEnum = NULL;
    if (!szName || !ppFilter)
    {
        return E_POINTER;
    }

    // Create the system device enumerator.
    hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL, CLSCTX_INPROC,
        IID_ICreateDevEnum, (void**)&pDevEnum);
    if (FAILED(hr))
    {
        return hr;
    }

    // Create a class enumerator for the specified category.
    hr = pDevEnum->CreateClassEnumerator(guidCategory, &pEnum, 0);
    pDevEnum->Release();
    if (hr != S_OK) // S_FALSE means the category is empty.
    {
        return E_FAIL;
    }

    // Enumerate devices within this category.
    bool bFound = false;
    IMoniker *pMoniker;
    while (!bFound && (S_OK == pEnum->Next(1, &pMoniker, 0)))
    {
        IPropertyBag *pBag = NULL;
        hr = pMoniker->BindToStorage(0, 0, IID_IPropertyBag, (void **)&pBag);
        if (FAILED(hr))
        {
            pMoniker->Release();
            continue; // Maybe the next one will work.
        }
        // Check the friendly name.
        VARIANT var;
        VariantInit(&var);
        hr = pBag->Read(L"FriendlyName", &var, NULL);
        if (SUCCEEDED(hr) && (lstrcmpiW(var.bstrVal, szName) == 0))
        {
            // This is the right filter.
```

```
                hr = pMoniker->BindToObject(0, 0, IID_IBaseFilter,
                    (void**)ppFilter);
                bFound = true;
            }
            VariantClear(&var);
            pBag->Release();
            pMoniker->Release();
        }
        pEnum->Release();
        return (bFound ? hr : E_FAIL);
}
```

下面的代码演示了如何利用这个函数创建 the CC Decoder filter，并加到 graph

```
IBaseFilter *pCC = NULL;
hr = CreateKernelFilter(AM_KSCATEGORY_VBICODEC,
    OLESTR("CC Decoder"), &pCC);
if (SUCCEEDED(hr))
{
    hr = pGraph->AddFilter(pCC, L"CC Decoder");
    pCC->Release();
}
```

2.5Directshow Editing Services

2.6DVD 应用（DVD Application）

2.7 MPEP_2 支持

2.8Windows Media 应用

2.9 TV 应用

2.10 使用视频混合 Render

2.11 Using the Stream Buffer Engine

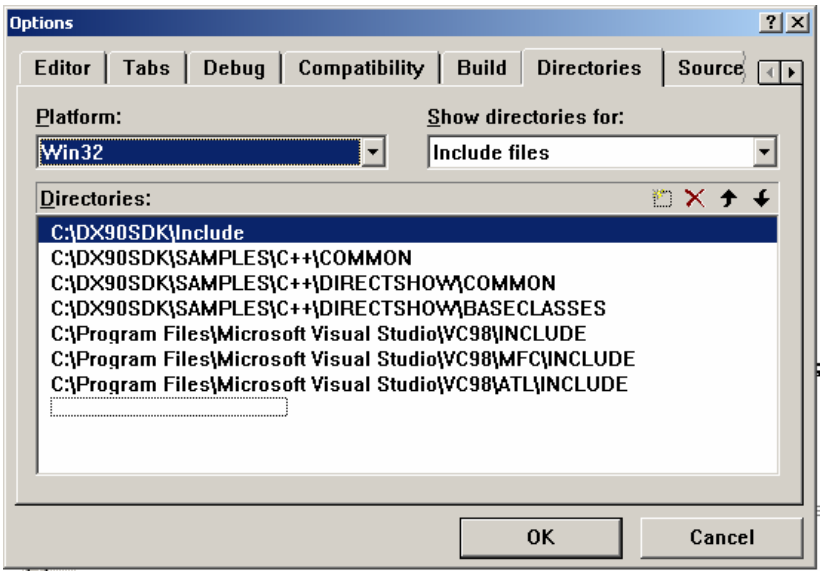
2.12 开发自己的 Filter

1 如何开发自己的 filter

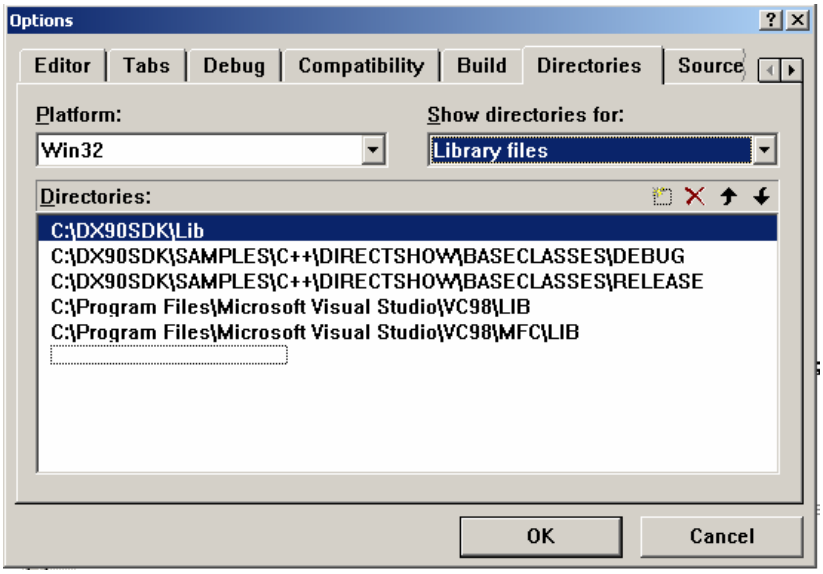
学习 directshow 已经有几天了，下面将自己的学习心得写下来，希望对其他的人有帮助。Filter 实质是个 COM 组件，所以学习开发 Filter 之前你应该对 com 的知识有点了解。Com 组件的实质是一个实现了纯虚指针接口的 C++对象。关于 com 的东西，这里不多讲。给 vc **配置 DShow 的开发环境**

无论开发 Filter 还是开发 Dshow 的应用程序都要配置一下开发环境的，其实就是包含一下 dshow 用到的头文件和动态库。 选择 Tools 菜单下面的 Options。在弹出的 Option 对话框配置如下；

添加头文件



选择动态库文件添加到工程中

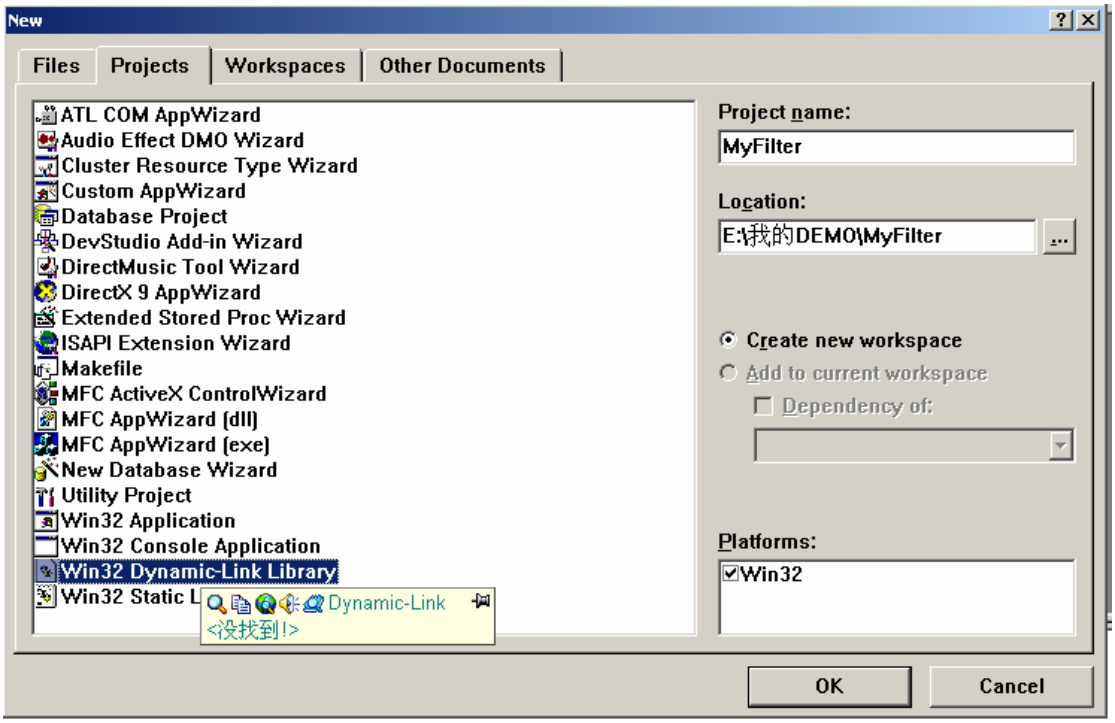


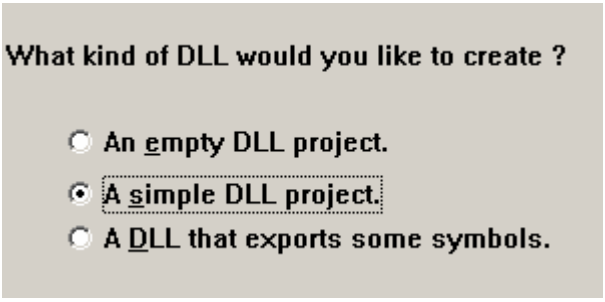
创建工程以及 Filter 的入口函数

创建工程

一般情况下，创建 Filter 使用一个普通的 Win32 DLL 项目。而且，一般 Filter 项目不使用 MFC。这时，应用程序通过 CoCreateInstance 函数 Filter 实例；Filter 与应用程序在二进制级别的协作。另外一种方法，也可以在 MFC 的应用程序项目中创建 Filter。

在 vc 里新建一个工程，选择 win32 动态库，如下图；





这样生成了一个简单的 DLL，只有一个 Dllmain 入口函数。
下面我要给这个 filter 添加入口函数了。
Filter 是个基于 DLL 的 com 组件，所以一般的 Filter 都要实现下面几个入口函数

DllMain
DllGetClassObject
DllCanUnloadNow
DllRegisterServer
DllUnregisterServer

首先定义导出函数

要导出这些函数有两种方法，一是在定义函数时使用导出关键字_declspec(dllexport)，另外一种方法是在创建 DLL 文件时使用模块定义文件.Def。使用导出函数关键字_declspec(dllexport)创建 MyDll.dll 就是在 .h 文件中定义定义函数如下，

extern "C" _declspec(dllexport)BOOL DllRegisterServer; 等等

为了用.def 文件创建 DLL，往该工程中加入一个文本文件，命名为 MyDll.def，再在该文件中加入如下代码：

```
LIBRARY      MyFilter.ax
EXPORTS
        DllMain                 PRIVATE
        DllGetClassObject       PRIVATE
        DllCanUnloadNow         PRIVATE
        DllRegisterServer       PRIVATE
        DllUnregisterServer     PRIVATE
```

其中 LIBRARY 语句说明该 def 文件是属于相应 DLL 的，EXPORTS 语句下列出要导出的函数名称。我们可以在.def 文件中的导出函数后加@n，如 Max@1，Min@2，表示要导出的函数顺序号，在进行显式连时可以用到它。该 DLL 编译成功后，打开工程中的 Debug 目录，同样也会看到 MyDll.dll 和 MyDll.lib 文件。
然后要定义这些函数的实现了，其实这些工作 dshow 的基类里都已经替我们做好了，我们所要做的就拿来用就是了，最重要的三个函数的实现一般如下

```
STDAPI DllRegisterServer()
{
    return AMovieDllRegisterServer2(TRUE);
}
 STDAPI DllUnregisterServer()
{
    return AMovieDllRegisterServer2(FALSE);
```

```
}
 extern "C" BOOL WINAPI DllEntryPoint(HINSTANCE, ULONG, LPVOID);
BOOL APIENTRY DllMain(HANDLE hModule, DWORD  dwReason,  LPVOID lpReserved)
{
      return DllEntryPoint((HINSTANCE)(hModule), dwReason, lpReserved);

}
```

其中DllEntryPoint 是在C:\DX90SDK\Samples\C++\DirectShow\BaseClasses\dllentry.cpp定义的，如果感兴趣我们可以去看看它的定义。

AMovieDllRegisterServer2 函数是在下面

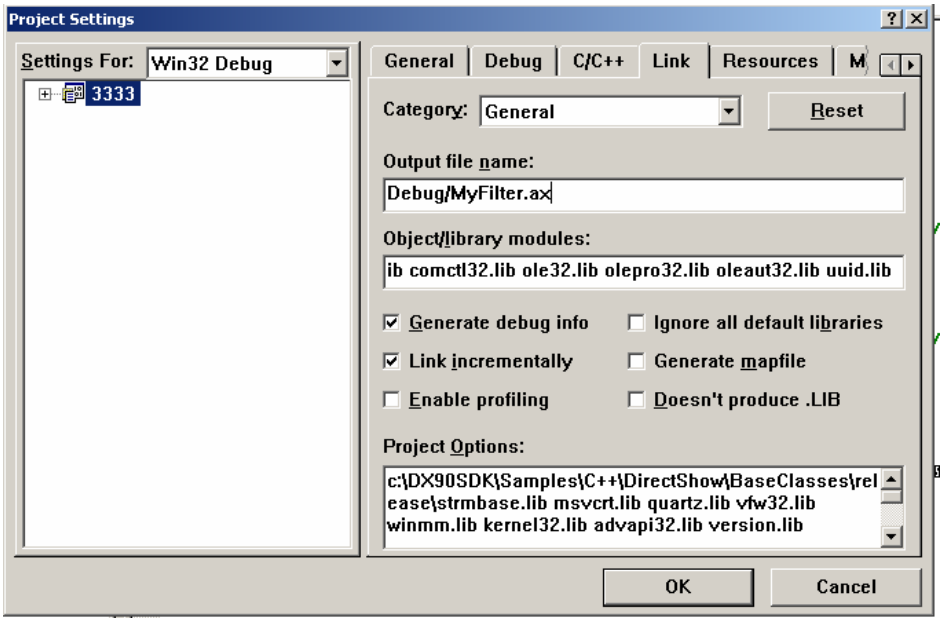
C:\DX90SDK\Samples\C++\DirectShow\BaseClasses\dllsetup.cpp 这个文件定义的，具体实现可以自己看看。

到了这里你恐怕要做点工作，还是要设置一下你的项目环境，否则恐怕你编译是通不过的，因为你用到了基类的一些东西，所以你要将你的 dshow 基类的定义和库文件包含进来。

首先包含

```
#include <Streams.h>
```

其次在 Project –Setting 菜单下配置自己的 Filter 输出的名字和连接的 lib 文件



其中 library modules 里的包含的动态库如下

c:\DX90SDK\Samples\C++\DirectShow\BaseClasses\debug\strmbasd.lib msvcrtd.lib quartz.lib vfw32.lib winmm.lib kernel32.lib advapi32.lib version.lib largeint.lib user32.lib gdi32.lib comctl32.lib ole32.lib olepro32.lib oleaut32.lib uuid.lib

此时你编译一下，好像还是通不过，它提示有一个全局的用于实现 COM 接口的变量没有定义，不着急，下面我们就开始实现 Filter 的 com 接口。

如何实现 Filter 的类厂对象

本节内容要讲一下 Filter 是如何实现 com 接口的，它和其他的 com 实现方式的不同。

我们知道一个 Filter 是一个 com 组件，所以它 com 特性的实现其实在其基类中实现的，比如 **IUnknown** 接口，我们直接从基类派生出我们的 Filter 后，它就支持 com 接口了，它就是一个 com 组件了。

所有的 com 组件为了实现二进制的封装，所以连创建的接口都封装了，因此每个 com 对象都有个**类对象**（也叫**类厂对象**，本身也是 com 对象，用来创建 com 组件）来创建 com 组件。下面温习一下 com 组件的创建过程，其中涉及到几个函数

1 当客户端要创建一个 com 组件时，它通过底层的 COM API 函数 CoGetClassObject（）使用 SCM 的服务，这个函数请 SCM 把一个指针绑定到客户端请求的 com 组件的类对象上，

其实在 CoGetClassObject()里它装载了该 DLL 的库，通过该 dll 的导出函数 **DllGetClassObject（）；DllGetClassObject 根据客户端提供的 com 组件 CLASSID，返回该 com 组件类对象的指针。**下面 com 组件的创建就是 SCM 无关了。

2 客户端利用组件的**类对象（类厂对象）**的 **IClassFactory::CreateInstance 方法**创建 com 组件。

Filter 在这里使用了一个类厂模板类来当作 Filter 的类厂对象。

下面看看类厂在 DShow 是怎么工作的。

类厂对象也是一个 com 组件。本来 **DllGetClassObject** 是我们自己写的一个函数，在 directshow 里已经完成了，我们不用自己来完成它了。它的功能就是来寻找这个 DLL 中的类厂对象，看是否有符合客户端请求的类厂对象。

DLL 里声明了一个全局的类厂模板数组，当 **DllGetClassObject 请求类厂对象的时候，它就搜索这个数组，看是否有和** CLSID 匹配的类厂对象。当它找到一个匹配的 CLSID，它就创建一个类厂对象，然后讲类厂指针返回给 CoGetClassObject，然后客户端可以根据返回去的类厂指针，调用 **IClassFactory::CreateInstance 方法创建组件，类厂就根据数组里定义的方法创建 com 组件。**

factory template 包含下列变量：

```
const WCHAR *              m_Name;              // Name
const CLSID *              m_ClsID;             // CLSID
LPFNNewCOMObject           m_lpfnNew;           // Function to  //create an
                               instance   of the component
LPFNInitRoutine      m_lpfnInit; // Initialization function (optional)
const AMOVIESETUP_FILTER * m_pAMovieSetup_Filter; // Set-up information (for
filters)
```

其中的两个函数指针 m_lpfnNew and **m_lpfnInit 使用下面的定义**

```
typedef CUnknown *(CALLBACK *LPFNNewCOMObject)(LPUNKNOWN pUnkOuter, HRESULT *phr);
typedef void (CALLBACK *LPFNInitRoutine)(BOOL bLoading, const CLSID *rclsid);
```

你可以参照如下的方式定义你的类厂对象

假如下面是你的 com 组件创建函数，

```
CUnknown * WINAPI CMyFilter::CreateInstance(LPUNKNOWN pUnk, HRESULT *pHr)
{
    CMyFilter *pFilter = new CMyFilter(NAME("my Filter"), pUnk, pHr);
    if (pFilter== NULL)
    {
        *pHr = E_OUTOFMEMORY;
    }
    return pFilter;
}
```

你可以声明自己的类厂数组如下：如果在这个 com 组件中你要支持多个 filter，你可以在这个数组中继续添加就是了。

```
CFactoryTemplate g_Templates[1] =
{
    {
      L"my filter",                    // Name
      &CLSID_MYFilter,                 // CLSID
      CMyFilter::CreateInstance,       // Method to create an instance of MyComponent
      NULL,                            // Initialization function
      &sudInfTee                       // Set-up information (for filters)
    }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
```

如何实现自己的Filter

在这里就要讲如何创建自己的 Filter 了，下面我们以写一个 **CTransformFilter** 为例

1 选择一个基类，声明自己的类

创建 filter 很简单，你只要根据自己的需要选择不同的基类 Filter 派生出自己的 Filter，它就已经支持 com 特性了。

从逻辑上考虑，在写 Filter 之前，选择一个合适的 Filter 基类是至关重要的。为此，你必须对几个 Filter 的基类有相当的了解。在实际应用中，Filter 的基类并不总是选择 CBaseFilter 的。相反，因为我们绝大部分写的都是中间的传输 Filter（Transform Filter），所以基类选择 CTransformFilter 和 CTransInPlaceFilter 的居多。如果我们写的是源 Filter，我们可以选择 CSource 作为基类；如果是 Renderer Filter，可以选择 CBaseRenderer 或 CBaseVideoRenderer 等。

总之，选择好 Filter 的基类是很重要的。当然，选择 Filter 的基类也是很灵活的，没有绝对的标准。能够通过 CTransformFilter 实现的 Filter 当然也能从 CBaseFilter 一步一步实现。下面，笔者就从本人的实际经验出发，对 Filter 基类的选择提出几点建议供大家参考。

首先，你必须明确这个 Filter 要完成什么样的功能，即要对 Filter 项目进行需求分析。请尽量保持 Filter 实现的功能的单一性。如果必要的话，你可以将需求分解，由两个（或者更多的）功能单一的 Filter 去实现总的功能需求。

其次，你应该明确这个 Filter 大致在整个 Filter Graph 的位置，这个 Filter 的输入是什么数据，输出是什么数据，有几个输入 Pin、几个输出 Pin 等等。你可以画出这个 Filter 的草图。弄清这一点十分重要，这将直接决定你使用哪种“模型”的 Filter。比如，如果 Filter 仅有一个输入 Pin 和一个输出 Pin，而且一进一处的媒体类型相同，则一般采用 CTransInPlaceFilter 作为 Filter 的基类；如果媒体类型不一样，则一般选择 CTransformFilter 作为基类。

再者，考虑一些数据传输、处理的特殊性要求。比如 Filter 的输入和输出的 Sample 并不是一一对应的，这就一般要在输入 Pin 上进行数据的缓存，而在输出 Pin 上使用专门的线程进行数据处理。这种情况下，Filter 的基类选择 CSource 为宜（虽然这个 Filter 并不是源 Filter）。当 Filter 的基类选定了之后，Pin 的基类也就相应选定了。接下去，就是 Filter 和 Pin 上的代码实现了。有一点需要注意的是，从软件设计的角度上来说，应该将你的逻辑类代码同 Filter 的代码分开。下面，我们一起来看一下输入 Pin 的实现。你需要实现基类所有的纯虚函数，比如 CheckMediaType 等。在 CheckMediaType 内，你可以对媒体类型进行检验，看是否是你

期望的那种。因为大部分 Filter 采用的是推模式传输数据，所以在输入 Pin 上一般都实现了 Receive 方法。有的基类里面已经实现了 Receive，而在 Filter 类上留一个纯虚函数供用户重载进行数据处理。这种情况下一般是无需重载 Receive 方法的，除非基类的实现不符合你的实际要求。而如果你重载了 Receive 方法，一般会同时重载以下三个函数 EndOfStream、BeginFlush 和 EndFlush。我们再来看一下输出 Pin 的实现。一般情况下，你要实现基类所有的纯虚函数，除了 CheckMediaType 进行媒体类型检查外，一般还有 DecideBufferSize 以决定 Sample 使用内存的大小，GetMediaType 提供支持的媒体类型。

最后，我们看一下 Filter 类的实现。首先当然也要实现基类的所有纯虚函数。除此之外，Filter 还要实现 CreateInstance 以提供 COM 的入口，实现 NonDelegatingQueryInterface 以暴露支持的接口。如果我们创建了自定义的输入、输出 Pin，一般我们还要重载 GetPinCount 和 GetPin 两个函数。

这里我主要为了举例，所以简单写的 filter 没有 Pin 接口，但在我的 demo 里的 Filter，却是有个 out pin 和一个 input pin。

我的 Filter 类的定义如下：

```
class CMyFilter :   public CCritSec, public CBaseFilter
 {
public:
    CMyFilter(TCHAR *pName,LPUNKNOWN pUnk,HRESULT *hr);
    virtual ~CMyFilter();
     static CUnknown * WINAPI CreateInstance(LPUNKNOWN pUnk, HRESULT
*phr);
     CBasePin *GetPin(int n);
     int GetPinCount();
};
```

注：因为基类是一个纯虚的基类，所以在你的 filter 一定要派生一个其中的纯虚函数，否则编译器会提示你的派生类也是一个纯虚类，你在创建这个 com 组件对象的时候，纯虚类是没法创建对象的。

2 给自己的 Filter 生成一个 CLSID

你可以用 **Guidgen** or **Uuidgen** 给自己的 Filter 生成一个 128 位的 ID 号，然后利用 DEFINE_GUID 宏在 Filter 的头文件声明该 Filter 的 CLSID;

[myFilter.h]

```
// {1915C5C7-02AA-415f-890F-76D94C85AAF1}
DEFINE_GUID(CLSID_MYFilter,
0x1915c5c7, 0x2aa, 0x415f, 0x89, 0xf, 0x76, 0xd9, 0x4c, 0x85, 0xaa, 0xf1);
```

这个 CLSID_MYFilter 在类厂数组用到，在注册 Filter 时也要用到。

3 CMyFilter 类的简单实现

这个类纯粹为了演示用，所以特别简单，你可以参考我的 demo，那个 filter 写的功能比较全。

```
CMyFilter::CMyFilter(TCHAR *pName,LPUNKNOWN pUnk,HRESULT *hr)
                :CBaseFilter(NAME("my filter"), pUnk, this, CLSID_MYFilter)
{ }
CMyFilter::~CMyFilter()
{}

// Public method that returns a new instance.
```

```
CUnknown * WINAPI CMyFilter::CreateInstance(LPUNKNOWN pUnk, HRESULT *pHr)
{
    CMyFilter *pFilter = new CMyFilter(NAME("my Filter"), pUnk, pHr);
    if (pFilter== NULL)
    {
        *pHr = E_OUTOFMEMORY;
    }
    return pFilter;
}

 CBasePin * CMyFilter::GetPin(int n)
 {
   return NULL;
 }
 int CMyFilter::GetPinCount()
 {
     return 0;
 }
```

这样基本上就实现了一个 filter，但是这个 filter 没有与之相联系的 PIN，但是实现 Filter 的基本过程就时这样了，至于逻辑上的东西，比如 Filter 和 pin 如何连接，数据流是如何流动的，你都要去看看 sdk 了，按照上面的步骤你就可以写一个 Filter 的框架出来。

下面我们总结一下写一个 Filter 至少需要那些东西。

1 Filter 的实现类

在这里就是 CMyFilter 类，在这个类里你可以实现自己的逻辑上的功能，包括定义你的 filter 的特性，给你的 filter 配备 pin 接口等。

2 com 组件的引出函数

五个全局函数

```
DllMain                 //dll 的入口函数
DllGetClassObject       //获得 com 组件的类厂对象
DllCanUnloadNow         //com 组件是否可以卸载
DllRegisterServer       //注册 com 组件
DllUnregisterServer     //卸载 com 组件
```

其中 DllGetClassObject 已经由基类完成你自己只要完成三个函数即可 DllMain，DllRegisterServer，DllUnregisterServer。

3 com 组件的类厂对象

类厂对象是用来生成 Filter 对象的，用的模板类定义了一个全局的模板类对象数组，一般格式如下

```
CFactoryTemplate g_Templates[1] =
{
    {
      L"my filter",                 // Name
      &CLSID_MYFilter,              // Filter 的 CLSID
      CMyFilter::CreateInstance,    //  创建 Filter 的方法
      NULL,                         // Initialization function
```

```
        &sudInfTee                    //Filter 的信息（一个结构）
    }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
```

4 关于你自己定义的 Filter 以及 Pin 的信息

这些是一个全局的结构变量，用于描述你的 Filter 和你定义的 pin，在注册 Filter 的时候会用到，如下

AMOVIESETUP_FILTER 描述一个 Filter

AMOVIESETUP_PIN 描述 pin

AMOVIESETUP_MEDIATYPE 描述数据类型

下面的代码描述了一个 Filter 带有一个 output PIN

```
static const WCHAR g_wszName[] = L"Some Filter";
AMOVIESETUP_MEDIATYPE sudMediaTypes[] = {
    { &MEDIATYPE_Video, &MEDIASUBTYPE_RGB24 },
    { &MEDIATYPE_Video, &MEDIASUBTYPE_RGB32 },
};
AMOVIESETUP_PIN sudOutputPin = {
    L"",              // Obsolete, not used.
    FALSE,            // Is this pin rendered?
    TRUE,             // Is it an output pin?
    FALSE,            // Can the filter create zero instances?
    FALSE,            // Does the filter create multiple instances?
    &GUID_NULL,       // Obsolete.
    NULL,             // Obsolete.
    2,                // Number of media types.
    sudMediaTypes     // Pointer to media types.
};

AMOVIESETUP_FILTER sudFilterReg = {
    &CLSID_SomeFilter,      // Filter CLSID.
    g_wszName,              // Filter name.
    MERIT_NORMAL,           // Merit.
    1,                      // Number of pin types.
    &sudOutputPin           // Pointer to pin information.
};
```

最后 **sudFilterReg** 会在类厂对象数组中用到。

```
CFactoryTemplate g_Templates[1] =
{
    {
      L"my filter",                 // Name
      &CLSID_MYFilter,              // CLSID
      CMyFilter::CreateInstance,    // Method to create an instance of MyComponent
      NULL,                         // Initialization function
      &sudFilterReg                 // Set-up information (for filters)
```

```
    }
};
```

最后如果你还是调试通不过，看看你是否包含了下面的头文件

```
#include <streams.h>    #include <initguid.h>
#include <tchar.h>      #include <stdio.h>
```

2filter 的连接

Pin 的连接

应用程序通过调用 filter 图表管理器的方法来连接 filter，并不是来调用 filter 或者 pin 本身的函数。应用程序可以调用 **IFilterGraph::ConnectDirect** or **IGraphBuilder::Connect** 来指定不同的 filter 直接连接，也可以通过 **IGraphBuilder::RenderFile** 间接连接。

只有两个 filter 都在 graph 里，连接才能成功。应用程序可以通过 **IFilterGraph::AddFilter** 将 filter 添加 graph 中，当一个 filter 被添加到 graph 中时，filter 图表管理器通过 **IBaseFilter::JoinFilterGraph** 来通知 filter。

Pin 连接的大致过程如下：

1 图表管理器首先调用输出 pin 上的 **IPin::Connect**，然后传递一个指针给输入 pin。

2 如果输出 pin 接受连接的邀请，它就调用输入 pin 上的 **IPin::ReceiveConnection**。

3 如果输入 pin 也接受连接邀请，那么连接成功，pin 之间的连接 ok。

当 filter 处于活动状态的时候，许多 pin 可以断开连接和重新连接。这种类型的连接称为动态连接。当然，大多数的 filter 并不支持动态连接。

Filter 通常采用从上游到下游的连接顺序。也就是说 filter 上的输入 pin 总是比输出 pin 先连接。Filter 应该支持这种连接顺序。然而有许多 filter 支持相反的连接顺序，输出 pin 先连接，输入 pin 后连接。例如：在连接 MUX filter 的输入 pin 之前一定要将 MUX filter 的输出 pin 和 writer filter 连接起来。

当 pin 的 **Connect** or **ReceiveConnection 方法**被调用的时候，pin 必须检查一下自己是否支持这个连接。通常要进行下列检查：

1 检查媒体类型是否匹配。

2 就内存的分配达成一致。

3 请求其他 pin 的其他接口。

媒体类型匹配

当一个 filter 图表管理器调用 **IPin::Connect** 方法时，可能有下面的几种媒体类型。

1 完整类型

如果媒体类型每一个部分都定义的很完成，那么 pin 就严格按照定义的类型类型进行连接。如果不匹配，连接失败。

2 部分媒体类型

如果媒体类型的机构中，major type, subtype, or format type 的值为 GUID_NULL，这个值是一个通配符号。任何类型都可以匹配。

3 没有媒体类型

如果 filter 图表管理器传递过来一个 NULL 的指针，这个 pin 就可以和任意的类型的媒体类型匹配。

一般在连接过程中，都有一个完整的媒体类型。图表管理器传递媒体类型的目的是为了限制连接类型。

一般来说，都是输出 pin 通过调用输入 pin **IPin::ReceiveConnection** 提供一个媒体类型。输入

pin 可以拒绝也可以接受这个媒体类型。这个过程一直重复，直到输入 pin 接受了一个类型，或者输出 pin 枚举完了它支持的所有的媒体类型，连接失败。
输出 pin 通过调用输入 pin 上的 **IPin::EnumMediaTypes** 枚举输入 pin 所支持的媒体类型。
看看如何匹配媒体类型的吧。

```
if ((pmt->formattype == FORMAT_VideoInfo) &&
    (pmt->cbFormat > sizeof(VIDEOINFOHEADER) &&
    (pbFormat != NULL))
{
    VIDEOINFOHEADER *pVIH = (VIDEOINFOHEADER*)pmt->pbFormat;
    // Now you can dereference pVIH.
}
```

Pin 连接中的内存分配

当两个 pin 连接起来后，他们需要一种机制来交换媒体数据。大多数数据交换采用的局部内存交换机制。所有的媒体数据都在主内存中。DirectShow 为局部存储器传输定义了两种机制：推模式（push model）和拉模式（pull model）。在推模式中，源过滤器生成数据并提交给下一级过滤器。下一级过滤器被动的接收数据，完成处理后再传送给再下一级过滤器。在拉模式中，源过滤器与一个分析过滤器相连。分析过滤器向源过滤器请求数据后，源过滤器才传送数据以响应请求。推模式使用的是 IMemInputPin 接口，拉模式使用 IAsyncReader 接口，推模式比拉模式要更常用。
在局部存储器传输中，负责分配内存的对象称为 allocator。每个 allocator 都支持一个 **IMemAllocator** 接口。所有的 pin 都共享一个 allocator。
每个 pin 都提供一个 allocator，但是输出 pin 选择使用哪个 allocator。
输出 pin 可以设置 allocator 的属性。比如，分配内存的大小，
在 **IMemInputPin** 连接中，allocator 工作过程如下
1 首先，输出 pin 调用 **IMemInputPin::GetAllocatorRequirements**，这个方法检查输入 pin 对内存的要求，比如内存的队列，一般来说，输出 pin 要满足输入 pin 对内存的要求。
2 输出 pin 然后调用 **IMemInputPin::GetAllocator**.，这个方法从输入 pin 请求一个 allocator，
3 输出 pin 选择一个 allocator，可以是输入 pin 提供，也可以是自己生产的。
4 输出 pin 调用 **IMemAllocator::SetProperties** 来设置 allocator 的属性。
5 然后输出 pin 通过 **IMemInputPin::NotifyAllocator** 来通知输入 pin，选择的 allocator。
6 输入 pin 通过 **IMemAllocator::GetProperties** 来检查是否能够接受 allocator 的属性。
7 当数据流开始和停止的时候，输出 pin 负责提交 allocator。
在 IAsyncReader 连接过程如下：
1 输入 pin 调用输出 pin 上的 **IAsyncReader::RequestAllocator**，输入 pin 确定内存的属性，并提供一个 allocator。
2 输出 pin 选择一个 allocator，
3 输入 pin 检查

如何提供一个自定义的 allocator

这里只讲一下 **IMemInputPin** 连接，IAsyncReader 类似。
首先，定义一个 C++类，你的 allocator 应该从一个标准的 allocator 类中派生，比如 **CBaseAllocator** or **CMemAllocator**，你也可以自己创建一个新的 allocator 类，如果你是新建的类，你必须支持 **IMemAllocator** 接口。
下面看看在输入 pin 和输出 pin 中如何使用你定义的 allocator。

在输入 pin 中提供 allocator

在输入 pin 中提供 allcator，必须重载 **CBaseInputPin::GetAllocator** 方法。在这个方法里，首先检查 **m_pAllocator** 是否可用，如果为非空，就表明 allocator 已经被选中，所以直接返回这个 allocator 指针即可，如果 **m_pAllocator** 为空，表明 allocator 还没有被选中，所以，就要返回输入 pin 的 allocator，因此，创建一个 allcaotor 的实例，返回 **IMemAllocator 接口。**

看下面的代码把

```
STDMETHODIMP CMyInputPin::GetAllocator(IMemAllocator **ppAllocator)
{
    CheckPointer(ppAllocator, E_POINTER);
    if (m_pAllocator)
    {
        // We already have an allocator, so return that one.
        *ppAllocator = m_pAllocator;
        (*ppAllocator)->AddRef();
        return S_OK;
    }
    // No allocator yet, so propose our custom allocator. The exact code
    // here will depend on your custom allocator class definition.
    HRESULT hr = S_OK;
    CMyAllocator *pAlloc = new CMyAllocator(&hr);
    if (!pAlloc)
    {
        return E_OUTOFMEMORY;
    }
    if (FAILED(hr))
    {
        delete pAlloc;
        return hr;
    }
    // Return the IMemAllocator interface to the caller.
    return pAlloc->QueryInterface(IID_IMemAllocator, (void**)ppAllocator);
}
```

当输出 pin 选择一个 allocator，它就调用输入 pin 的 **IMemInputPin::NotifyAllocator**，因此，要重载 **CBaseInputPin::NotifyAllocator** 方法来检查 allocator 的属性。

在输出 pin 中如何提供一个定制的 Allocator

在输出 pin 中提供一个 allcator，要重载 **CBaseOutputPin::InitAllocator**

```
HRESULT MyOutputPin::InitAllocator(IMemAllocator **ppAlloc)
{
    HRESULT hr = S_OK;
    CMyAllocator *pAlloc = new CMyAllocator(&hr);
    if (!pAlloc)
    {
        return E_OUTOFMEMORY;
    }
```

```
    if (FAILED(hr))
    {
        delete pAlloc;
        return hr;
    }
    // Return the IMemAllocator interface.
    return pAlloc->QueryInterface(IID_IMemAllocator, void**)ppAllocator);}
}
```

缺省情况下 **CBaseOutputPin** 首先从输入 pin 中申请一个 allocator，

3filter 间的数据流动

1 传递 Samples

本文讲述了如何传递一个 sample，包括两种模式下，推模式下采用 IMemInputPin 的方法，在拉模式下调用 IAsyncReader 的方法。

推模式

输出 pin 通过调用 **IMemInputPin::Receive** 或者 **IMemInputPin::ReceiveMultiple** 方法来传递一个 sample。在 **Receive 和 ReceiveMultiple** 方法里，输入 pin 可以阻塞数据流。如果输入 pin 阻塞，那么 **IMemInputPin::ReceiveCanBlock** 必须返回 S_OK。如果 pin 保证不会阻塞，那么 **ReceiveCanBlock** 方法要返回 S_FALSE，返回 S_OK 并不表明 Receive 方法阻塞，只是表明可能阻塞。

尽管 Receive 可以阻塞一直等待某种资源变的可用，但是它不能通过阻塞来等待数据流的到来。因为如果上游的 filter 正在等待下游的 filter 正在等待下游的 filter 释放资源，就会造成死锁。**如果一个 filter 拥有多个输入 pin，那么其中的一个 pin 可以等待另外的一个 pin 接收数据。例如 AVI Mux filter 就是通过这种方法来同步音频和视频流的。**

如果有以下原因，pin 可能拒绝 sample。

1pin 正在 flushing

2pin 没有连接

3filter 停止

4 发生了其他错误

如果输入 pin 拒绝了 sample，那么 **Receive** 方法就要返回 S_FALSE，或者其它的错误码，如果 **Receive** 没有返回 S_OK，那么上游的 fitler 就会停止发送 sample。

前面三种错误都是可以预见的错误，第四种是不可预见的错误，即使 pin 正处于接收数据流德状态，当这种错误发生时，接受 pin 就拒绝接受 sample，发送 pin 就给下游的连接 pin 发送一个结束发送数据流的通知，并且给 Filter 图表管理器发送一个 EC_ERRORABORT 事件通知。在 directshow 的基类中 **CBaseInputPin::CheckStreaming** 方法用来检查通常的数据流错误，比如 flushing, stopped, and so forth。派生类要检查所发生的错误。在发生错误的时候，**CBaseInputPin::Receive** 方法将发送一个结束数据流的通知和一个 EC_ERRORABORT 事件通知。

在拉模式下，IAsyncReader 接口中，输入 pin 将通过以下方法从输出 pin 中请求 samples。

- IAsyncReader::Request
- IAsyncReader::SyncRead
- IAsyncReader::SyncReadAligned

Reques 方法是异步的，输入 pin 调用 **IAsyncReader::WaitForNext** 来等待请求数据传递结束。另外两个方法是同步。

2 数据处理

//略

3 数据流结束的通知

当一个源 filter 结束发送数据流时，它调用和它连接的 filter 的输入 pin 的 **IPin::EndOfStream**，然后下游的 filter 再依次通知与之相连的 filter。当 EndOfStream 方法一直调用到 renderer filter 的时候，最后的一个 filter 就给 filter 图表管理器发送一个 EC_COMPLETE 事件通知。如果 renderer 有多个输入 pin，当所有的输入 pin 都接收到 end of stream 通知的时候，它才会给 filter 图表管理器发送一个 EC_COMPLETE 事件通知。

Filter 必须在其他函数调用之后调用 EndOfStream 函数，比如 **IMemInputPin::Receive**.。

在一些情况下，下游的 filter 可能比源 filter 更早的发现数据流的结束。在这种情况下，下游 filter 发送 结束 stream 的通知，同时， **IMemInputPin::Receive** 函数返回 S_FALSE 直到图表管理器停止。这个返回值提示源 filter 停止发送数据。

对 EC_COMPLETE 事件的缺省处理

缺省的情况下，filter 图表管理器并不将 **EC_COMPLETE** 事件通知发送给应用程序，当所有的数据流都发送了 **EC_COMPLETE 事件通知后，它才给**应用程序发送一个 EC_COMPLETE 事件通知。所以，应用程序只有在所有的数据流停止的时候才能接收到这个通知。

filter 图表管理器通过计算支持 seeking 接口的 filter，并且具有一个 renderer pin，没有相应的输出 pin，就可以确定数据流的数目。Filter 图表管理器通过下面的方法来决定一个 pin 是否是个 renderer 。

1 pin 的 **IPin::QueryInternalConnections** 方法通过 nPin 参数返回 0；

2 filter 保露一个 **IAMFilterMiscFlags** 接口，并且返回一个 AM_FILTER_MISC_FLAGS_IS_RENDERER 标志。

在拉模式下的数据流结束通知

在 **IAsyncReader** 连接中，源 filter 并不发送数据流结束的通知，相应的发送数据流结束的通知是有 renderer filter 发出的。

4New Segments（本节翻译的不好，我自己都不理解，乱七八糟）

一个段就是一组 media samples，这些 sample 具有共同的开始时间，结束时间，播放速率。

The **IPin::NewSegment** 方法用来通知一个 new segments 的开始。源 filter 通过这种方法来通知下游的 filter segment 的开始时间和播放速率。例如，如果源 filter 在数据流中改变了新的开始点，它就用新的时间做参数来通知下游的 filter。

下游的 filter 在处理 sample 的时候需要 segment。例如，在桢间压缩的时候，if the stop time falls on a delta frame, the source filter may need to send additional samples after the stop time. This enables the decoder to decode the final delta frame.为了确定正确的结束桢，解码器指向色 gement 的停止时间。另外一个例子，在音频播放的过程中，播放 filter 利用 segment 的速度和音频 sample 速度来产生正确的输出。

在推模式中，源 filter 产生一个新的 segment，并初始化。在拉模式，这个工作是由剖析器（parser）来完成的。两种情况下，filter 都调用下游 filter 的输入 pin 上的 NewSegment，一直到达 renderfilter。当 filters 调用数据流时候，必须序列化 **NewSegment。**

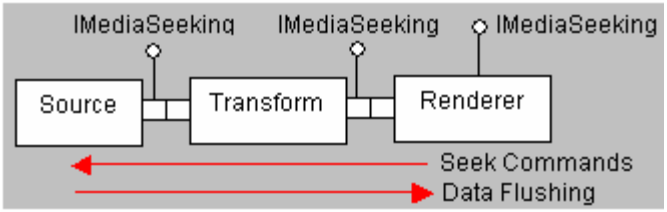
当每一个新的 segment，数据流的时间都被重新设置为零，当 segment 从零开始的时候，samples 重新贴上了 time 标签。

5 Flushing

当 graph 运行的时候，在整个 graph 中会有大量的数据流动。同时也有一些数据排在队列里等到传递。当 graph 移动这些未决的数据，并在该内存块中写入新的数据是需要一定的时间的。例如，在 seek 命令后，源 filter 在生成新的 sample，这些是需要一定时间的。为了减小延迟，下游的 filter 在 seek 命令必须丢掉以前的 sample。这个抛弃 sample 的过程就叫 flushing。
当事件改变了数据的流向时，这可以使 garph 响应的更及时一些。
推模式和拉模式在处理 flushing 的时候有点不同。我们先讨论一下推模式，然后再讨论拉模式。
下面两种情况下发生 flushing
1 源 filter 调用下游 filter 输入 pin 的 **IPin::BeginFlush** 方法，然后下游的 filter 就开始拒绝从上游 filter 接收数据流。然后它开始抛弃它正在处理的 samples，继续调用下游 filter 的 **IPin::BeginFlush** 方法
2 当源 filter 准备好新的数据时，它调用输入 pin 的 **IPin::EndFlush** 方法，这就告诉下游的 filter 可以接收新的 samples，然后继续调用下游的 filter 的 **IPin::EndFlush**。
在 BeginFlush 方法中，输入 pin 进行了下列工作
1 首先调用下游 filter 的输入 pin 上的 Calls **BeginFlush** 方法
2 拒绝处理数据流，包括 Receive 和 endofstream 方法
3 取消那些正在阻塞等待 filter 释放 allocator 的等待，
4 如果 filter 正处于阻塞数据状态，那么 filter 就退出阻塞。例如当停止的时候 Renderer filter 就阻塞，此时，filter 就要取消阻塞。
在 EndFlush 方法中，输入 pin 做了下列工作
1 等待所有正在队列中的 samples 被抛弃
2 释放存放数据的 buffer，这一步也可能在 BeginFlush 方法里，但是，beginflush 方法 streaming 线程是不同步的。Filter 在 BeginFlush 和 EndFlush 方法之间不能够处理如何数据
3 清除所有的 EC_COMPLETE 通知
4 调用下游 filter 的 EndFlush 方法
此时，filter 可以再次接收 sample。
在拉模式中，parser Filter 初始化 flushing，而不是由 source filter，它不仅调用了下游 filter**IPin::BeginFlush** and **IPin::EndFlush 方法**，它又调用了源 filter 输出 pin 上的 **IAsyncReader::BeginFlush** and **IAsyncReader::EndFlush**，如果此时，源 filter 有未决的 read 请求，它就抛弃

6Seeking

Filter 通过 **IMediaSeeking** 接口支持 seeking。应用程序从 filter 图表管理器请求 **IMediaSeeking** 接口，然后通过这个接口，执行 seek 命令。Filter 图表管理器传递到 graph 里所有的 renderer filter。每个 renderer 通过上游 filter 的输出 pin 来传递 seek 命令，直到某一个 filter 可以执行这个 seek 命令。一般来说，源 filter，或者 parser filter 可以执行 seek 命令。
当一个 filter 执行 seek 命令的时候，它就 flushes 所有的未决的数据，这是为了减少 seek 命令的迟延。当一个 seek 命令后，stream time 设置为零。
下面的图表演示了 seek 过程



如果一个 parser filter 的输出 pin 不止一个的话，它就指定一个 pin 来接收 seek commands，其他的 pin 当接收到 seek 命令时，就拒绝或者忽略 seek 命令。这样，parser 就保持了所有的数据流同步。

IMediaPosition 接口（略）

4pin 连接时数据格式的动态改变

当两个 filter 连接的时候，他们会就某种媒体类型达成协议。这种数据类型用来描述上游 filter 将传递什么格式的数据。大多数情况下，在连接的持续过程中，这个媒体类型是不变的，但是，directshow 也支持动态改变媒体类型。

Directshow 定义了一些机制来支持动态改变媒体类型。关键在于 filter graph 的状态和将要改变的类型。

如果 graph 处于停止状态，pin 可以重新连接，重新就某个媒体类型达成协议。

一些 filter 还支持动态的连接，

当 graph 处于活动状态，并且不支持动态的重新连接的时候，有三种机制支持媒体类型的改变。

QueryAccept (Downstream) 适用于输出 pin 向下游的 filter 提出改变媒体类型，并且新的媒体格式的大小不超过原来的 buffer。

QueryAccept (Upstream) 适用于输入 pin 首先向上游的 filter 提出媒体类型的改变，新的媒体类型格式可以和原来的大小一致，也可以大于。

ReceiveConnection 适用于输出 pin 首先提出改变媒体类型，但是新的媒体类型的格式大于原来的 buffer。

4 Threads and Critical Sections

本章讲述一下 dshow filters 的线程和临界区，这样，你在开发 filter 的过程中就可以避免死锁和系统崩溃。

1 The Streaming and Application Threads

任何一个 Directshow 的应用程序中都至少包括两个重要的线程，一个是应用程序线程，一个或者多个的 Streaming 线程。在 streaming 线程中，samples 不断的传递，然后在应用程序中，**的状态不断的变化。Streaming 的主线程一般都是由源 filter 或者 parser Filter 来创建，其他的 filter 可以创建用来传递 samples 的工作线程，所有的这些线程都称为 streming 线程。

下面我们看看应用程序线程和 streaming 线程经常用到的

Streaming thread(s):**IMemInputPin::Receive**, **IMemInputPin::ReceiveMultiple**, **IPin::EndOfStream**, **IMemAllocator::GetBuffer**.

Application thread: **IMediaFilter::Pause**, **IMediaFilter::Run**, **IMediaFilter::Stop**, **IMediaSeeking::SetPositions**, **IPin::BeginFlush**, **IPin::EndFlush**.

Either **IPin::NewSegment**.

当我们的用户应用程序在等待用户输入的同时，数据流可以通过 streaming 线程在 graph 图中流动。在多线程中，filter 在暂停的时候，创建了资源，然后在 Streaming 方法中使用这些资源，当这个 filter 停止的时候，就销毁这些资源，这样就很危险，如果你不小心，streaming 线程有可能使用了已经被销毁的资源。解决的方法就是通过临界区来保护这些资源，来同步这些线程。

一个 filter 需要一个临界区来保护 filter 的状态。**CBaseFilter** 类有一个成员变量来保护这

个 filter 的状态，**CBaseFilter::m_pLock**。这个临界区称为 filter 锁。同时，每一个输入 pin 都需要一个临界区来保护 streaming 线程使用的资源，这些临界区成为 streaming 锁。你必须在你派生的 pin 类中（输入 pin）中来设置这些变量。使用一个 **CCritSec** 类很容易实现临界区。这个 **CCritSec** 类中包含一个 **CRITICAL_SECTION 窗口**对象。可以使用 **CAutoLock** 类来锁住这个临界区。

当一个 filter 停止或者 flushes 的时候，就必须同步应用线程和 streaming 线程。为了避免死锁，你必须要解锁 streaming 线程。

Streaming 线程加锁的原因主要有以下；

1 当 streaming 线程通过 **IMemAllocator::GetBuffer** 方法来请求 samples 的时候，如果此时所有的 allocators 分配的 samples 都在使用，那么此时线程就要等到

2 等到其他的 filter 从 streaming 线程方法返回，比如，Receive

3 在 streaming 方法中等待其他的资源的释放。

4 renderer Filter 正在 Receive 方法等待，如果 filter 停止，就处于死锁状态了

因此，当 filter 停止或者 flushes 时候，必须做下列事情

1 一定要释放它正在占用的任何资源

2 尽可能快地从 streaming 方法中返回，如果这个方法正在等到某种资源，就放弃等待，立即返回

3 在 Receive 方法开始停止接受 samples，这样就不会再请求新的资源了

4 Stop 方法一定要 decommit 所有 filter 的内存分配器。（当然了，**CBaseInputPin** 会自动地处理这个事情）

Flushing 和 stoping 方法在应用程序中都会发生。**IMediaControl::Stop** 方法可以使 filter 停止运行。Filter 管理器一般让 stop 命令从 render filter 开始，逆流向上到源 filter 逐渐停止。通过 **CBaseFilter::Stop** 方法，在这个方法返回的时候，filter 的状态就转变为停止状态。

Filter 在 seek 命令时一般都会 Flushing。Flush 命令一般都从源 filter 或者 parser filter 开始，然后向下游传递执行。Flushing 有两个状态，**IPin::BeginFlush** 方法通知一个 filter 开始抛弃所有的正在接受到的数据 **IPin::EndFlush** 方法通知 filterflush 结束，可以开始再次的接收数据。Flushing 之所以有两个阶段，因为 Beginflush 方法是由应用程序调用的，同时有可能还在传递数据。这样，在调用 beginflush 方法后，还有数据在 filter 中流动，此时 filter 就要抛弃这些数据。在调用 endflush 后，就能保证 filter 接受的数据都是新的，可以处理了。

下面的一些临界区的例子，演示了如何保护 filter 的重要的方法，比如 Pause，receive 等

2 Pausing

在 filter 的状态发生变化的时候，必须要加锁，在 Pause 方法中，要创建 filter 需要的资源

```
HRESULT CMyFilter::Pause()
{
    CAutoLock lock_it(m_pLock);

    /* Create filter resources. */

    return CBaseFilter::Pause();
}
```

CBaseFilter::Pause 方法设置 filter 处于 State_Paused)状态，然后调用和 filter 相连接的 pin 上的 **CBasePin::Active** 方法，Active 方法通知 pin：filter 开始处于激活状态了。如果 pin 需要创建资源，那么就要派生 Active 方法了

```
HRESULT CMyInputPin::Active()
```

```
{
    // You do not need to hold the filter lock here. It is already held in Pause.

    /* Create pin resources. */

    return CBaseInputPin::Active()
}
```

3 Receiving and Delivering Samples

下面的伪代码演示了如何派生输入 pin 的 Receive 方法

```
HRESULT CMyInputPin::Receive(IMediaSample *pSample)
{
    CAutoLock cObjectLock(&m_csReceive);

    // Perhaps the filter needs to wait on something.
    WaitForSingleObject(m_hSomeEventThatReceiveNeedsToWaitOn, INFINITE);

    // Before using resources, make sure it is safe to proceed. Do not
    // continue if the base-class method returns anything besides S_OK.
    hr = CBaseInputPin::Receive(pSample);
    if (hr != S_OK)
    {
        return hr;
    }

    /* It is safe to use resources allocated in Active and Pause. */

    // Deliver sample(s), via your output pin(s).
    for (each output pin)
        pOutputPin->Deliver(pSample);

    return hr;
}
```

Receive 方法设置了一个 streaming 锁，不是 filter 锁。Filter 在开始处理数据之前可能需要等待其他事件发生，这里就调用了 **WaitForSingleObject**。当然并不是所有的 filter 都需要这样做。**CBaseInputPin::Receive** 方法验证一些基本的 streaming 条件，如果 filter 停止旧返回 VFW_E_WRONG_STATE，如果 filter flushing 该方法返回 s_FALSE;如果没有返回 S_OK 就表示 Receive 方法应该立即返回，不能处理 sample。

在 sample 被处理后，就调用 **CBaseOutputPin::Deliver**.方法向下传递。依次类推。数据开始传递下去。

4 Delivering the End of Stream

当一个输入 pin 接收到一个数据流停止的通知后，它就会向下传播下去。下游的可以从这个输入 pin 接收数据的 filter 也应该接收到数据流停止的通知。如果 filter 还有未决的数据没有处理，filter 应该在它传递停止通知之前一定要将数据传递下去，在 end tream 消息发布以后，不能再向下游 filter 发送数据了

```
HRESULT CMyInputPin::EndOfStream()
{
    CAutoLock lock_it(&m_csReceive);

    /* If the pin has not delivered all of the data in the stream
        (based on what it received previously), do so now.    */

    // Propagate EndOfStream call downstream, via your output pin(s).
    for (each output pin)
    {
        hr = pOutputPin->DeliverEndOfStream();
    }
    return S_OK;
}
```

CBaseOutputPin::DeliverEndOfStream 调用了与输出 pin 连接的输入 pin 的 **IPin::EndOfStream** 方法来通知下游的 filter 数据流即将停止了。

5 Flushing Data

下面的伪代码演示 **IPin::BeginFlush** 方法

```
HRESULT CMyInputPin::BeginFlush()
{
    CAutoLock lock_it(m_pLock);

    // First, make sure the Receive method will fail from now on.
    HRESULT hr = CBaseInputPin::BeginFlush();

    // Force downstream filters to release samples. If our Receive method
    // is blocked in GetBuffer or Deliver, this will unblock it.
    for (each output pin)
    {
        hr = pOutputPin->DeliverBeginFlush();
    }

    // Unblock our Receive method if it is waiting on an event.
    SetEvent(m_hSomeEventThatReceiveNeedsToWaitOn);

    // At this point, the Receive method can't be blocked. Make sure
    // it finishes, by taking the streaming lock. (Not necessary if this
    // is the last step.)
    {
        CAutoLock lock_2(&m_csReceive);

        /* Now it's safe to do anything that would crash or hang
            if Receive were executing. */
    }
```

```
    return hr;
}
```

当开始 Flushing 的时候，BeginFlush 方法使用了 filter 锁。如果使用 streaming 锁不是很安全，因为 flush 是发生在应用程序中，有可能此时 streaming 线程正处于 Receive 方法中。Pin 要保证 Receive 方法没有被阻塞，否则，后续调用 Receive 方法都会失败。

CBaseInputPin::BeginFlush 设置了一个标志位 **CBaseInputPin::m_bFlushing**.，如果这个标志为 TRUE，Receive 方法就会失败。

在下游 filter 传递 Beginflush 方法的时候，pin 一定要保证下游的 filter 释放他们的 samples 并且从 Receive 方法中返回。这就保证了输入 pin 不会阻塞在 GetBuffer 和 Receive 方法中。如果你的 pin 的 Receive 方法正在等待某种资源，那么 GeginFlush 方法就通知设置某些特定事件来结束这种等待，这就保证 Receive 方法能够立即返回，**m_bFlushing** 标志就阻止 Receive 方法调用。

对于一些 filter 来说，这些都是必须要做的，EndFlush 方法通知 filter 可以重新接收数据了。Endflush 方法使用的是 filter 锁。然后向下传播。

```
HRESULT CMyInputPin::EndFlush()
{
    CAutoLock lock_it(m_pLock);
    for (each output pin)
        hr = pOutputPin->DeliverEndFlush();
    return CBaseInputPin::EndFlush();
}
```

CBaseInputPin::EndFlush 重新设置 **m_bFlushing** 标志为 FALSE。这样 Receive 方法就可以开始接收数据了，必须在最后调用这个方法。

6 Stopping

Stop 方法必须 unblock Receive 方法并且 decommit filter 的内存分配器。这样就使 GetBuffer 返回。Stop 方法使用了 filter 锁，然后调用了 **CBaseFilter::Stop**，这个 stop 方法调用了 filter pins 上的 **CBasePin::Inactive** 方法。

```
HRESULT CMyFilter::Stop()
{
    CAutoLock lock_it(m_pLock);
    // Inactivate all the pins, to protect the filter resources.
    hr = CBaseFilter::Stop();

    /* Safe to destroy filter resources used by the streaming thread. */

    return hr;
}
```

Override the input pin's **Inactive** method as follows:

```
HRESULT CMyInputPin::Inactive()
{
    // You do not need to hold the filter lock here.
    // It is already locked in Stop.
```

```
    // Unblock Receive.
    SetEvent(m_hSomeEventThatReceiveNeedsToWaitOn);

    // Make sure Receive will fail.
    // This also decommits the allocator.
    HRESULT hr = CBaseInputPin::Inactive();

    // Make sure Receive has completed, and is not using resources.
    {
        CAutoLock c(&m_csReceive);

        /* It is now safe to destroy filter resources used by the
            streaming thread. */
    }
    return hr;
}
```

7 Getting Buffers

如果你有一个内存分配器来分配资源，那么 GetBuffer 方法采用 streaming 锁。

```
HRESULT CMyInputAllocator::GetBuffer(
    IMediaSample **ppBuffer,
    REFERENCE_TIME *pStartTime,
    REFERENCE_TIME *pEndTime,
    DWORD dwFlags)
{
    CAutoLock cObjectLock(&m_csReceive);

    /* Use resources. */

    return CMemAllocator::GetBuffer(ppBuffer, pStartTime, pEndTime, dwFlags);
}
```

8 Streaming Threads and the Filter Graph Manager

当 filter 图表管理器停止 graph，它要等待所有的 streaming 线程停止，

9 Summary of Filter Threading

Streaming 线程调用的函数

IMemInputPin::Receive

IMemInputPin::ReceiveMultiple

IPin::EndOfStream

IPin::NewSegment

IMemAllocator::GetBuffer

看看应用程序调用的函数把

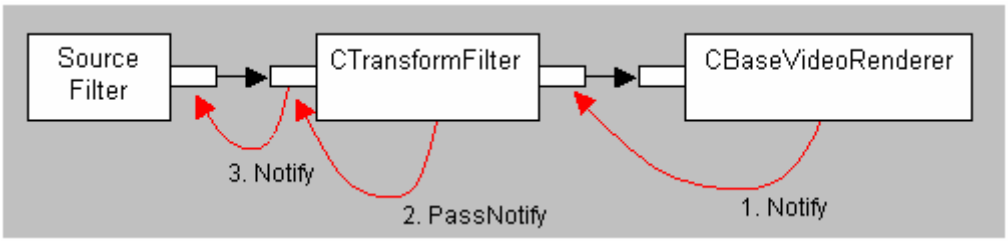
状态改变

IBaseFilter::JoinFilterGraph, **IMediaFilter::Pause**, **IMediaFilter::Run**, **IMediaFilter::Stop**, **IQualityControl::SetSink**.

参考时钟
IMediaFilter::GetSyncSource, **IMediaFilter::SetSyncSource**.
Pin 的操作
IBaseFilter::FindPin, **IPin::Connect**, **IPin::ConnectedTo**, **IPin::ConnectionMediaType**, **IPin::Disconnect**, **IPin::ReceiveConnection**.
内存的分配
IMemInputPin::GetAllocator, **IMemInputPin::NotifyAllocator**.
Flushing
IPin::BeginFlush, **IPin::EndFlush**.

5 质量控制管理

DirectShow Base Classes 提供了一些缺省的方法来控制视频的质量。质量消息从 renderer 开始，这个 renderer 基类为 **CBaseVideoRenderer**,这个基类有下列一些行为
1 当视频 render 接收到 sample 的时候，它就将 sample 上的时间戳和当前的参考时间比较
2 视频 render 产生一个质量消息，在基类中，质量消息的比例值被限制在 500（50%）~~2000（200%），超出这个范围，质量就变得不好
3 缺省的时候，render 就给上游的 filter 输出 pin 发送一个质量消息，应用程序可以同过 setsink 方法来 override 这种行为。
当消息被逆流传递到上一个 filter 时会发生什么呢？一般来说，上游的 filter 都是一个 transform 过滤器。这种 transformfilter 都有一个 **CTransformInputPin** and **CTransformOutputPin**。
1 **CTransformOutputPin::Notify** 方法调用 **CTransformFilter::AlterQuality**
2transformfilte 可以通过重载 **AlterQuality** 方法来处理质量消息，缺省的情况下，**AlterQuality** 忽略质量消息
3 如果 **AlterQuality** 不处理质量消息，输出 pin 就调用输入 pin 上的 **CBaseInputPin::PassNotify**,
4**PassNotify** 方法就将质量消息传递给上游的其他的 filter。
假定没有传输 filter 来处理质量消息，质量消息就最后到达了源 filter。在基类中，**CBasePin::Notify** 返回 E_NOTIMPL。源 filter 如何处理质量消息取决于源 filter 的 nature。一些 filter 可以调整他们发送 sample 的速率。



6dshow 和 com

微软的 Directshow 是基于 COM 的，如果你开发自己的 filter，你一定实现一个 com 对象。Directshow 的基类提供了一个框架可以实现 com 接口，你不一定适用基类，但是使用基类可以减轻你的开发任务。下面我们讲讲 dshow 和 com 关系
我们假设你了解如何开发 com 的客户端，也就是说你了解 IUnknown 方法，但是并不详细了

解如何开发一个 com 对象，Directshow 将如何开发 com 的一切细节都替你处理好了。如果你有开发 com 的经验，你应该读一下 Using **CUnknown** 一章。
Com 是一个协议，它规定了所有组件必须遵循的规则，如何应用这些规则就留给开发者了。在 Directshow 中，所有的对象都是从一系列的基类中继承而来，这些基类的函数和构造器将 com 对象的构造工作都基本做完了，例如引用计数，接口等功能。你将你的 filter 从基类中继承，你就继承了基类的函数，同时你的 filter 就是一个 com 对象了。为了好好继承基类，你必须要了解基类是如何实现 com 对象的。
本章要求你了解一下三个内容
How **IUnknown** Works
How to Create a DLL.
How to Register DirectShow Filters.

1 How IUnknown Works

通过 IUnknown 接口的方法应用程序可以请求对象的接口，管理组件的引用计数。
引用计数
引用是个内部变量，通过 addref 方法可以是变量增加，通过 release 方法可以使该变量值减少。基类管理者这个引用计数，并且是多线程同步。
接口请求
接口请求也是简单易懂的。调用者传递两个参数，一个是接口的 ID，和接口的指针的地址。如果组件支持该接口，就将该指针指向接口，引用计数增加 1，返回 S_OK，否则，将该指针设置为 NULL，返回 E_NOINTERFACE。下面的伪代码演示了 QueryInterface 函数

```
if (IID == IID_IUnknown)
    set pointer to (IUnknown *)this
    AddRef
    return S_OK

else if (IID == IID_ISomeInterface)
    set pointer to (ISomeInterface *)this
    AddRef
    return S_OK

else if ...

else
    set pointer to NULL
    return E_NOINTERFACE
```

两个组件的 querinterface 函数的区别就在于 IIDS 的列表不同，对于组件支持的每一个接口，都要检查 IID 是否正确。

聚合和委托

组件的聚合对于调用者必须是透明的，因此，聚合体必须只能暴露一个 IUnknown 接口，被聚合组件的接口听从外部的组件的调用，否则客户端就看到两个不同的 IUnknown 接口，如果组件不被聚合，他们都有自己的实现。
为了支持聚合，组件必须增加一个额外的间接的指针，用来分别指向外部接口和内部接口，*nondelegating IUnknown* 接口就可以完成这项工作。

外部的接口是公开的，还叫做 IUnknown，内部的接口叫做 **INonDelegatingUnknown**.，这个名字不是 com 定义的，因为它不是一个公开的接口
当客户端创建了一个组件的实例，它调用 **IClassFactory::CreateInstance** 方法，一个参数就指向组件的 IUnknown 接口，组件就利用这个参数来保存一个变量用来标示需要采用哪个 IUnknown 接口，看看下面的代码

```
CMyComponent::CMyComponent(IUnknown *pOuterUnkown)
{
    if (pOuterUnknown == NULL)
        m_pUnknown = (IUnknown *)(INonDelegatingUnknown *)this;
    else
        m_pUnknown = pOuterUnknown;

    [ ... more constructor code ... ]
}
```

委托的 IUnknown 接口调用的它的副本，如下

```
HRESULT QueryInterface(REFIID iid, void **ppv)
{
    return m_pUnknown->QueryInterface(iid, ppv);
}
```

Using **CUnknown**

Directshow 在 **CUnknown**.基类中实现了 IUnkown 接口，directshow 的其他基类也多是从 CUnknown 类中继承过来的，所以，你也可以从这个类派生一个新类或者从其他基类派生你自己的类。

INonDelegatingUnknown

CUnknown 类还实现了一个 **INonDelegatingUnknown** 接口，这个接口是用来管理引用计数的。大多数情况下，你的派生类可以直接这个接口的两个管理引用计数的函数而不用做任何的修改。但是你要记住，当引用计数等于 0 的时候，CUnknown 就要销毁自己。有时候你必须要派生 **CUnknown::NonDelegatingQueryInterface** 函数，因为在基类中，如果它发现请求接口的 ID 不是 IID_IUnknown 的时候就返回 E_NOINTERFACE,在你的派生类中，你就要自己来测试你所支持的接口 ID，看下面的例子

```
STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_ISomeInterface)
    {
        return GetInterface((ISomeInterface*)this, ppv);
    }
    // default
    return CUnknown::NonDelegatingQueryInterface(riid, ppv);
}
```

GetInterface 将接口指针设置为所请求的接口，返回 s_ok，增加引用计数，缺省的情况下，要调用你继承类的 **NonDelegatingQueryInterface**

IUnknown

日前所述，任何一个组件的 IUnknown 接口的处理流程都是一样的。因为它仅仅是根据客户端的请求返回正确的接口即可。因此，为了方便，在 **Combase.h** 头文件中声明了一个宏

DECLARE_IUNKNOWN,这种宏中定义了三个内联函数，如下

```
STDMETHODIMP QueryInterface(REFIID riid, void **ppv) {
    return GetOwner()->QueryInterface(riid,ppv);
};
STDMETHODIMP_(ULONG) AddRef() {
    return GetOwner()->AddRef();
};
STDMETHODIMP_(ULONG) Release() {
    return GetOwner()->Release();
};
```

CUnknown::GetOwner 函数返回拥有这个 IUnknown 接口的组件，对于聚合组件，ower 指的外部的组件。否则，指的就是自己。在你自己的类中的公共部分，包含 DECLARE_IUNKNOWN 宏声明。

类的构造

你自己的类的构造函数一定要从基类的构造函数继承过来，如下

```
CMyComponent(TCHAR *tszName, LPUNKNOWN pUnk, HRESULT *phr)
    : CUnknown(tszName, pUnk, phr)
{
    /* Other initializations */
};
```

构造函数一般都有三个参数，这三个参数传递到 CUnknown 类的构造函数中，其实你的构造函数基本不用作任何工作，一切都在 CUnknown 函数的构造函数给你做好了。

TszName 指定组件的名字

pUnk 指向一个 IUnknown 指针

pHr 指向一个 HRESULT 值，表示成功还是失败。

下面的例子演示了，如何派生你自己的类，这个类支持 Iunknown 接口还有一个 ISomeInterface 接口

```
class CMyComponent : public CUnknown, public ISomeInterface
{
public:

    DECLARE_IUNKNOWN;

    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void **ppv)
    {
        if( riid == IID_ISomeInterface )
        {
            return GetInterface((ISomeInterface*)this, ppv);
        }
        return CUnknown::NonDelegatingQueryInterface(riid, ppv);
    }

    CMyComponent(TCHAR *tszName, LPUNKNOWN pUnk, HRESULT *phr)
        : CUnknown(tszName, pUnk, phr)
```

```
    {
        /* Other initializations */
    };

    // More declarations will be added later.
};
```

这个例子假定了一下情况

1， CUnknown 类实现了 IUnknown 接口，如何新的组件从 CUnknown 基类继承过来，那么同时也就继承了基类所支持的接口。你可以从其他继承于 CUnknown 的类派生你自己的类，而不一定要从 CUnknown 派生

2， DECLARE_IUNKNOWN 宏将 IUnknown 接口的三个函数声明为内联函数

3， CUnknown 类提供了 **INonDelegatingUnknown**.的实现

4， 如果新的组件支持的接口不仅仅是 IUnknown，那么就一定要重新继承 **INonDelegatingUnknown**.函数，测试新接口的 ID，如上代码

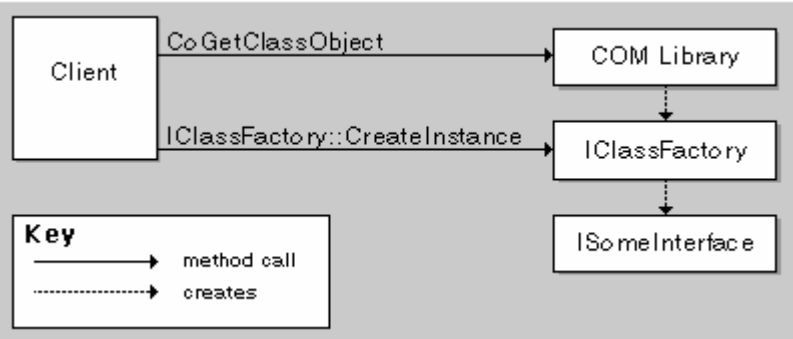
5， 派生类的构造函数触发了 CUnknown 的构造函数

下一步就是使得应用程序创建组件的一个实例，这就要求了解 DLLs 和类厂，以及构造函数的关系，请看 How to Create a DLL.

2How to Create a DLL.

在客户端创建一个 com 对象的实例以前，它首先通过 **CoGetClassObject 方法**创建一个对象类厂的实例。然后客户端调用类厂的 **IClassFactory::CreateInstance** 方法。类厂自动创建一个组件然后返回一个指向组件对象接口的指针。**CoCreateInstance** 方法是上面两步的综合。

下面的图演示了对象创建



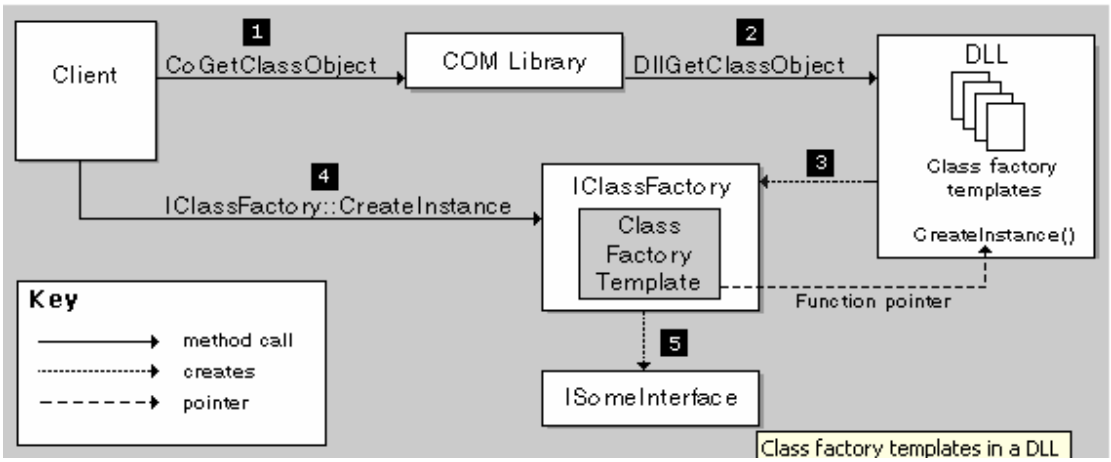
CoGetClassObject 方法调用了 DLL 中的 **DllGetClassObject** 方法，这个方法创建了一个类厂对象，并且返回一个类厂对象的接口。**Directshow 已经替你完成了 DllGetClassObject 方法 为了了解**他们是如何工作，你必须了解 directshow 是如何实现类厂的。

类厂也是一个 com 对象，它可以用来创建其他的 com 组件。每个类厂只能用来创建特定的 com 组件对象。在 directshow 中，每一个类厂都是 C++类 **CClassFactory** 的一个实例，类厂是通过一个叫做类厂模板 **CFactoryTemplate**,来实现的。每个类厂类都有一个指向类厂模板的指针，类厂模板包含了要创建的组件的信息，比如 CLSID，和一个指向创建对象函数的指针。

DLL 中定义了一个全局的类厂模板的数组，每一个动态的 DLL 中都有这么一个全局的模版数组，当 **DllGetClassObject** 函数创建组件对象的时候，它就搜索全局数组里的匹配的 CLSID，如果它找到匹配的数组，它就创建一个包含一个指向匹配模板指针的类厂对象，当客户端调用 **IClassFactory::CreateInstance** 方法的时候，类厂就调用模版中的函数来创建组件对象，下面的 DllGetClassObject 函数是我从 dshow 的基类中找到的，我们仔细看看吧

```
STDAPI
```

```
DllGetClassObject(
    REFCLSID rClsID,
    REFIID riid,
    void **pv)
{
    if (!(riid == IID_IUnknown) && !(riid == IID_IClassFactory)) {
            return E_NOINTERFACE;
    }

    // 首先在类厂模板数组中查找相应的CLSID
    // class id
    for (int i = 0; i < g_cTemplates; i++) {
        const CFactoryTemplate * pT = &g_Templates[i];
        if (pT->IsClassID(rClsID)) {
            //如果找到，就用这个类厂模板作参数生成一个类厂对象
            // found a template - make a class factory based on this
            // template

            *pv = (LPVOID) (LPUNKNOWN) new CClassFactory(pT);
            if (*pv == NULL) {
                return E_OUTOFMEMORY;
            }
            ((LPUNKNOWN)*pv)->AddRef();
            return NOERROR;
        }
    }
    return CLASS_E_CLASSNOTAVAILABLE;
}
```



下面我们看看类厂模板数组

类厂模板数组包含以下变量

```
const WCHAR *               m_Name;                 // Name
const CLSID *               m_ClsID;                // CLSID
LPFNNewCOMObject            m_lpfnNew; // Function to create an instance    of the component
```

```
LPFNInitRoutine          m_lpfnInit;          // Initialization function (optional)
const AMOVIESETUP_FILTER * m_pAMovieSetup_Filter; // Set-up information (for filters)
```

两个函数指针 **m_lpfnNew** and **m_lpfnInit** 使用如下的定义

```
typedef      CUnknown *(CALLBACK *LPFNNewCOMObject)
                        (LPUNKNOWN pUnkOuter, HRESULT *phr);
Typedef     void (CALLBACK *LPFNInitRoutine)
                        (BOOL bLoading, const CLSID *rclsid);
```

第一个用来创建对象的实例函数，第二个函数是初始化函数，如果你定义了这个函数，在动态库 DLL 的进入点函数就会调用这个初始化函数。

假设你定义了一个 DLL，这个 DLL 包含了一个叫做 CMYComponent 组件，它继承与 CUnknown，你必须在你的 DLL 定义下面的几个东西

1 一个创建你的组件的实例的公有函数。

2 一个全局的类厂模板数组，名字一定要命名为 **g_Templates，这个数组包含创建组件的类厂模板**

3 一个叫做全局变量 **g_cTemplates**，这个变量用来表示类厂模板数组的大小。

下面是示例代码

```
// Public method that returns a new instance.
CUnknown * WINAPI CMyComponent::CreateInstance(LPUNKNOWN pUnk, HRESULT *pHr)
{
    CMyComponent *pNewObject = new CMyComponent(NAME("My Component"), pUnk, pHr );
    if (pNewObject == NULL) {
        *pHr = E_OUTOFMEMORY;
    }
    return pNewObject;
}

CFactoryTemplate g_Templates[1] =
{
    {
      L"My Component",                  // Name
      &CLSID_MyComponent,                 // CLSID
      CMyComponent::CreateInstance,     // Method to create an instance of MyComponent
      NULL,                             // Initialization function
      NULL                              // Set-up information (for filters)
    }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
```

类厂的 CreateInstance 方法调用组件的构造函数，并返回一个指向新类实例的一个指针。参数 punk 指向 IUnknown 接口指针，看看我从基类里找到的 Createinstance 函数的代码

```
STDMETHODIMP
CClassFactory::CreateInstance(
    LPUNKNOWN pUnkOuter,
```

```
    REFIID riid,
    void **pv)
{
    CheckPointer(pv,E_POINTER)
    ValidateReadWritePtr(pv,sizeof(void *));

    /* Enforce the normal OLE rules regarding interfaces and delegation */

    if (pUnkOuter != NULL) {
        if (IsEqualIID(riid,IID_IUnknown) == FALSE) {
            return ResultFromScode(E_NOINTERFACE);
        }
    }

    /* Create the new object through the derived class's create function */

    HRESULT hr = NOERROR;
    CUnknown *pObj = m_pTemplate->CreateInstance(pUnkOuter, &hr);

    if (pObj == NULL) {
    if (SUCCEEDED(hr)) {
        hr = E_OUTOFMEMORY;
    }
    return hr;
    }

    /* Delete the object if we got a construction error */

    if (FAILED(hr)) {
        delete pObj;
        return hr;
    }

    /* Get a reference counted interface on the object */

    /* We wrap the non-delegating QI with NDAddRef & NDRelease. */
    /* This protects any outer object from being prematurely    */
    /* released by an inner object that may have to be created  */
    /* in order to supply the requested interface.              */
    pObj->NonDelegatingAddRef();
    hr = pObj->NonDelegatingQueryInterface(riid, pv);
    pObj->NonDelegatingRelease();
    /* Note that if NonDelegatingQueryInterface fails, it will  */
    /* not increment the ref count, so the NonDelegatingRelease */
```

```
    /* will drop the ref back to zero and the object will "self-*/
    /* destruct".   Hence we don't need additional tidy-up code   */
    /* to cope with NonDelegatingQueryInterface failing.            */

    if (SUCCEEDED(hr)) {
        ASSERT(*pv);
    }

    return hr;
}
```

我们发现，类厂就是通过全局变量 **g_Templates** and **g_cTemplates** 来创建组件的，所以，**g_Templates** and **g_cTemplates 的名字不能改变**

下面看看 dll 的导出函数

DLL Functions

一个动态库必须导出如下的函数，才能够注册，注销，加载。

DllMain: The DLL entry point. The name DllMain is a placeholder for the library-defined function name. The DirectShow implementation uses the name DllEntryPoint. For more information, see the Platform SDK. Dshow 没有 DllMain，改用了 DllEntryPoint。进入点函数

DllGetClassObject: Creates a class factory instance. Described in the previous sections.

DllCanUnloadNow: Queries whether the DLL can safely be unloaded.

DllRegisterServer: Creates registry entries for the DLL.

DllUnregisterServer: Removes registry entries for the DLL.

当然了，前面的三个函数，directshow 已经替我们完成了，如果你的类厂模板数组中的 m_lpfnInit 指针有一个初始化函数，那么在 DLL 的入口函数就会被调用。

你自己必须要提供 **DllRegisterServer** and **DllUnregisterServer** 函数来实现组件的注册和反注册，但是 Directshow 提供了提供了一个函数叫做 **AMovieDllRegisterServer2** 已经替你做完了必要的工作，所以你要做的就很简单了，只需如下就可以了

```
STDAPI DllRegisterServer()
{
    return AMovieDllRegisterServer2( TRUE );
}

STDAPI DllUnregisterServer()
{
    return AMovieDllRegisterServer2( FALSE );
}
```

当然，在 **DllRegisterServer** and **DllUnregisterServer** 函数中，你可以根据需要定制自己的注册信息，如果你的组件包含一个过滤器，你就要自己来做一下额外的工作了，具体的你可以下节 How to Register DirectShow Filters.

在你的 module-definition (.def) file 文件中，除了进入点函数，你要导出下面的函数，实例如

```
EXPORTS
    DllGetClassObject PRIVATE
    DllCanUnloadNow PRIVATE
    DllRegisterServer PRIVATE
```

DllUnregisterServer PRIVATE

你可以用 Regsvr32.exe 来注册你的组件

3How to Register DirectShow Filters.

Directshow 的 filter 一般都注册在两个地方

1 包含 filter 的 DLL 一般都注册为 filter 的 COM 服务器，当用户调用 CoCreateInstance 来创建一个 filter 的时候，微软的 COM 库就从这个注册表的入口加载 DLL。

2 另外，filter 可以注册到 filter 种类里，这样，System Device Enumerator and the Filter Mapper 就可以找到 filter 了。

第二种注册不是必须的，只要 filter 注册成为 com 服务器，一个应用程序就可以创建一个 filter 并将它加入到 filter Graph 中，但是，如果你想要你的 filter 可以被 System Device Enumerator and the Filter Mapper 发现，你必须注册到 filter 种类里。

Com 服务器的入口注册有下列以些键值

HKEY_CLASSES_ROOT
CLSID
Filter CLSID
REG_SZ: (Default) = Friendly name

InprocServer32
REG_SZ: (Default) = File name of the DLL
REG_SZ: ThreadingModel = Both

注册成 filter 种类里需要下面的键值

HKEY_CLASSES_ROOT
CLSID
Category
Instance
Filter CLSID
REG_SZ: CLSID = Filter CLSID
REG_BINARY: FilterData = Filter information
REG_SZ: FriendlyName = Friendly name

Category is the GUID of a filter category.

所有的 filter 信息在注册表的 filter 种类都如下所示

HKEY_CLASSES_ROOT\CLSID\{DA4E3DA0-D07D-11d0-BD50-00A0C911CE86}\Instance

1 声明 filter 信息 Declaring Filter Information

第一步要声明 filter 的信息，directshow 定义了如下的结构来声明 filter ， pin 和 media Types

Structure	Description
AMOVIESETUP_FILTER	Describes a filter.
AMOVIESETUP_PIN	Describes a pin.
AMOVIESETUP_MEDIATYPE	Describes a media type.

这些结构是必须的。

AMOVEIESETUP_FILTER 结构包含一个指针指向 **AMOVIESETUP_PIN** 结构数组，这两个结构中都有一个指针指向 **AMOVEIESETUP_MEDIATYPE**。这些结构提供了足够的信息可

以让 **IFilterMapper2** 指针找到 filter 的位置。但是，这并不能完全描述一个 filter，例如，如果一个 filter 创建了一个 pin 的多个实例，你只需要声明一个 AMOVIESETUP_PIN 结构即可。同样，一个 filter 没有必须支持注册的所有的媒体类型，也没有必要注册所有的媒体类型。

在你的 DLL 中声明一些全局的 Set_up 结构变量，如下

```
static const WCHAR g_wszName[] = L"Some Filter";

AMOVIESETUP_MEDIATYPE sudMediaTypes[] = {
    { &MEDIATYPE_Video, &MEDIASUBTYPE_RGB24 },
    { &MEDIATYPE_Video, &MEDIASUBTYPE_RGB32 },
};

AMOVIESETUP_PIN sudOutputPin = {
    L"",                // Obsolete, not used.
    FALSE,              // Is this pin rendered?
    TRUE,               // Is it an output pin?
    FALSE,              // Can the filter create zero instances?
    FALSE,              // Does the filter create multiple instances?
    &GUID_NULL,         // Obsolete.
    NULL,               // Obsolete.
    2,                  // Number of media types.
    sudMediaTypes       // Pointer to media types.
};

AMOVIESETUP_FILTER sudFilterReg = {
    &CLSID_SomeFilter,          // Filter CLSID.
    g_wszName,                  // Filter name.
    MERIT_NORMAL,               // Merit.
    1,                          // Number of pin types.
    &sudOutputPin               // Pointer to pin information.
};
```

Filter 的名字被声明成静态全局变量，因为它有可能在其它地方用到。

2 声明类厂模板数组 Declaring the Factory Template

```
CFactoryTemplate g_Templates[] = {
    {
        g_wszName,                          // Name.上面定义的全局变量
        &CLSID_SomeFilter,                  // CLSID.
        CSomeFilter::CreateInstance,        // Creation function.
        NULL,
        &sudFilterReg                       // Pointer to filter information.
    }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
```

3 生成 DllRegisterServer

最后一步是生成 DllRegisterServer 函数，包含组件的 DLL 必须导出这个函数，这个函数在安

装的时候被调用，或者当用户运行 Regsvr32.exe 时调用到。
简单的实现如下

```
STDAPI DllRegisterServer(void)
{
    return AMovieDllRegisterServer2(TRUE);
}
```

AMovieDllRegisterServer2 对于 **g_Templates** 数组中的所有组件都创建注册表入口，但是这个函数有一些限制，第一，它将所有的 filter 都注册到"DirectShow Filters"类下（CLSID_LegacyAmFilterCategory），其实并非所有的 filter 都属于这个种类。例如，捕捉 filter 和压缩 filter 就有他们自己的种类，第二，如果你的 filte 支持一个硬件设备，你必须要注册额外的两个信息 the *medium* and the *pin category*.，但是 **AMovieDLLRegisterServer2** 并不支持，pin 的种类定义了一个 pin 的函数方法。Mediums 和硬件的驱动有关。
如果你要注册 filter 的种类，medium 或者 pin 的种类，你可以在 DllRegisterServer（）中调用 **IFilterMapper2::RegisterFilter**，这个方法有个 **REGFILTER2** 结构，包含了 filter 的信息。
为了支持复杂的情况，**REGFILTER2** 结构支持两种不同格式 pin 的注册，dwVersion 表示两种格式
如果 dwVersion 为 1，pin 的类型就是 **AMOVIESETUP_PIN**
如果 dwVersion 为 2，拼得类型就是 **REGFILTERPINS2**.
REGFILTERPINS2.结构中包含 mediums 和 pin 的 categories
下面的例子演示了，如何在 DllRegistServer 中调用 **IFilterMapper2::RegisterFilter**

```
REGFILTER2 rf2FilterReg = {
    1,                  // Version 1 (no pin mediums or pin category).
    MERIT_NORMAL,    // Merit.
    1,                  // Number of pins.
    &sudPins            // Pointer to pin information.
};

STDAPI DllRegisterServer(void)
{
    HRESULT hr;
    IFilterMapper2 *pFM2 = NULL;

    hr = AMovieDllRegisterServer2(TRUE);
    if (FAILED(hr))
        return hr;

    hr = CoCreateInstance(CLSID_FilterMapper2, NULL, CLSCTX_INPROC_SERVER,
            IID_IFilterMapper2, (void **)&pFM2);

    if (FAILED(hr))
        return hr;

    hr = pFM2->RegisterFilter(
        CLSID_SomeFilter,                   // Filter CLSID.
```

```
        g_wszName,                          // Filter name.
        NULL,                               // Device moniker.
        &CLSID_VideoCompressorCategory,     // Video compressor category.
        g_wszName,                          // Instance data.
        &rf2FilterReg                       // Pointer to filter information.
    );
    pFM2->Release();
    return hr;
}
```

4filter 注册指南

Filter 的注册信息决定了，在 filter Graph 管理器中如何 Intelligent Connect.。因此，你必须要遵从下列的规则，使得你的 filter 能够正常运行。

1 你是否需要在注册表中保存你的 filter 数据，对于许多 filter 来说，没有必要让 filter Mapper 和 System Device Enumerator 来发现你的 filter，只要你注册了你的 filter，你的应用程序通过 ConCreateInstance 方法来创建你的 filter，此时，忽略了类厂模板中的 **AMOVIESETUP_FILTER** 结构，缺点是，在 GraphEdit 中看不到你的 filter。

2 选择正确的 filter 种类，缺省的 Directshow Filters 可能适用于大多数的 filter，但是如果你的 filter 有特殊的用处，你要选择一个恰当的种类

3 避免在 pin 的 **AMOVIESETUP_MEDIATYPE** 结构中使用 MEDIATYPE_None, MEDIASUBTYPE_None, or GUID_NULL，**IFilterMapper2** 会将这些视做通配符。

4 下面是一些建议的最小******不明白

Type of filter	Recommended merit
Default renderer	MERIT_PREFERRED. For standard media types, however, a custom renderer should never be the default.
Non-default renderer	MERIT_DO_NOT_USE or MERIT_UNLIKELY
Mux	MERIT_DO_NOT_USE
Decoder	MERIT_NORMAL
Spitter, parser	MERIT_NORMAL or lower
Special purpose filter; any filter that is created directly by the application	MERIT_DO_NOT_USE
Capture	MERIT_DO_NOT_USE
"Fallback" filter; for example, the Color Space Converter Filter	MERIT_UNLIKELY

5 不要将接受 24 位 RGB 的 filter 注册到 Directshow filter，你的 filter 将会干扰 Color Space Converter filter.工作

5 反注册

为了反注册 filter，
要提供一个 **DllUnregisterServer** 方法，在这个方法中，调用 **AMovieDllRegisterServer2**，注意传递参数，FASLE，如果你是使用 **IFilterMapper2::RegisterFilter** 注册的你的 filter，那么你必须要用 **IFilterMapper2::UnregisterFilter** 方法来反注册你的 filter。如下

```
STDAPI DllUnregisterServer()
{
    HRESULT hr;
    IFilterMapper2 *pFM2 = NULL;

    hr = AMovieDllRegisterServer2(FALSE);
    if (FAILED(hr))
        return hr;

    hr = CoCreateInstance(CLSID_FilterMapper2, NULL, CLSCTX_INPROC_SERVER,
            IID_IFilterMapper2, (void **)&pFM2);

    if (FAILED(hr))
        return hr;

    hr = pFM2->UnregisterFilter(&CLSID_VideoCompressorCategory,
            g_wszName, CLSID_SomeFilter);

    pFM2->Release();
    return hr;
}
```

7 如何写 Transform Filter

在开发自己的 filter 之前，看看 DMO（DirectX Media Object）是否满足你的要求，因为 DMO 可以做许多和 filter 相同的工作，但是开发 DMO 比开发 filter 要简单多了。开发 transform filter 主要有下面的几个步骤，努力的遵循吧

第一步选择一个基类

下面的基类适合开发 transform filter。

CTransformFilter 就是为了 transform filter 而设计的基类，这个类中有分开的输入和输出 buffers，这种类型的 filter 有时也称作 *copy-transform* filter，当一个 *copy-transform* filter 接收到一个输入 samples 的时候，它就将 sample 写入到一块新的输出 buffer 中，然后将这个新的 buffer 传递给下一个 filter。

CTransInPlaceFilter，这个类型的 filter 在原来的 buffer 里修改 data，也叫 *trans-in-place* filters. 当这种类型的 filter 接收到一个 sample，它改变这个 sample 中的数据，然后将 sample 传递下去，这种类型的输入 pin 和输出 pin 总是按照某个媒体类型连接起来。

CVideoTransformFilter 这个类型的 filter 仅仅是为了视频解码器设计的。从 CTransFormFilter 派生而来，但是这个 filter 可以根据下游的 render 自动的丢弃 data。

CBaseFilter 是个总基类，所有的 filter 都是从这个类派生出去的。如果上面的 filter 都不适合你，那么你只有自己从这个基类中派生了。

第二步声明自己的 Filter 类

首先声明一个从基类派生的 c++类

```
class CRleFilter : public CTransformFilter
{
    /* Declarations will go here. */
};
```

每个 filter 类都需要连接的 pin 类。根据你的需要，你要派生和你的 filter 连接的 pin 类。

你还要给你的 filter 设置一个不能重复的 CLSID，你可以利用 Guidgen or Uuidgen 来产生一个 128 位 CLSID，切忌不要拷贝其它的 filter 的。有很多种方法来声明 CLSID，下面的例子使用了 **DEFINE_GUID** 宏。

```
[RleFilt.h]
// {1915C5C7-02AA-415f-890F-76D94C85AAF1}
DEFINE_GUID(CLSID_RLEFilter,
0x1915c5c7, 0x2aa, 0x415f, 0x89, 0xf, 0x76, 0xd9, 0x4c, 0x85, 0xaa, 0xf1);

[RleFilt.cpp]
#include <initguid.h>
#include "RleFilt.h"
```

然后，给你的 filter 写一个构造函数

```
CRleFilter::CRleFilter()
  : CTransformFilter(NAME("My RLE Encoder"), 0, CLSID_RLEFilter)
{
   /* Initialize any private variables here. */
}
```

注意，构造函数中有个参数就是我前面定义的 CLSID。

第三步 支持媒体类戏协议

当两个 pin 连接的时候，他们必须就某种媒体类型达成一致协议，否则连接失败，数据媒体类型描述了数据的格式，如果没有媒体类型，一个 filter 可能传递一种类型的数据，然后其它的 filte 却不能识别这种数据。

Pin 连接的时候达成协议的机制主要通过 **IPin::ReceiveConnection** 方法来实现的。输出 pin 用某种媒体类型作参数调用输入 pin 上的这个方法，输入 pin 要么接受，要么拒绝。如果输入 pin 拒绝连接，那么输出 pin 更改一下媒体类型继续连接，直至所有的媒体类型都连接一遍，如果没有找到合适的媒体的类型，那么连接失败。

在输入 pin 也可以通过 **IPin::EnumMediaTypes** 方法来任意的枚举它所支持的媒体类型 list。输出 pin 可以通过这个 list 也可以检查是否支持某种媒体类型。

CTransformFilter 实现一个通用的框架。如下

1 输入 pin 没有首选的媒体类型，这个主要看上游的 filter 提议的媒体类型。对于视频数据，媒体类型包括图片的大小，和桢率，这个信息必须由上游的源 filter 或者 parser filter 提供。对于音频数据，设置的数据格式就小了许多，因此，要重载输入 pin 的 **CBasePin::GetMediaType**

2 当上游的 filter 提议一个媒体类型进行连接的时候，输入 pin 就调用 **CTransformFilter::CheckInputType** 方法，这个方法拒绝和接受媒体类型。

3 只有输入 pin 连接以后，输出 pin 才能够连接，这个是属于 transform filter 的一个特性。大多数情况下，filter 在设置输出 pin 的 type 之前一定要设置好输入 pin 的类型

4 当输出 pin 没有连接的时候，它向下游 filter 连接的时候，要枚举本 filter 支持的媒体类型，

形成一个 list，他通过调用 **CTransformFilter::GetMediaType** 方法来产生这个 list，输出 pin 会就下游 filter 所支持的所有的媒体类型进行连接

5 为了检测输入 pin 是否支持某个特定的输出媒体类型，输出 pin 通过调用 **CTransformFilter::CheckTransform** 方法。

上面列出的**三个 CTransformFilter 方法都是纯虚函数，因此你的 filter 必须实现这三个函数**

当上游的 filter 连接的时候提议一个媒体类型，那么输入 pin 就会调用函数

virtual HRESULT CheckInputType(const CMediaType* mtIn) pure;

这个函数包含了一个 **CMediaType** 类型的对象指针，这个类型封装了一个 **AM_MEDIA_TYPE** 结构。在这个函数中，你要检查 **AM_MEDIA_TYPE** 结构的中相关的 field，如果该结构中有任何 fied 不合法，就返回 VFW_E_TYPE_NOT_ACCEPTED，如果所有的媒体类型都是正确的，返还 S_OK

，例如，在 RLE 编码 filter，输入类型必须是 8 位或者 4 位的没有压缩的 RGB 视频。没有必要支持其它的输入格式，例如 16，24 位，因为那样，filter 还得进行转换。下面的例子假定 filter 只支持 8 位的视频，不支持 4 位的视频

```
HRESULT CRleFilter::CheckInputType(const CMediaType *mtIn)
{
    if ((mtIn->majortype != MEDIATYPE_Video) ||
        (mtIn->subtype != MEDIASUBTYPE_RGB8) ||
        (mtIn->formattype != FORMAT_VideoInfo) ||
        (mtIn->cbFormat < sizeof(VIDEOINFOHEADER)))
    {
        return VFW_E_TYPE_NOT_ACCEPTED;
    }

    VIDEOINFOHEADER *pVih =
        reinterpret_cast<VIDEOINFOHEADER*>(mtIn->pbFormat);
    if ((pVih->bmiHeader.biBitCount != 8) ||
        (pVih->bmiHeader.biCompression != BI_RGB))
    {
        return VFW_E_TYPE_NOT_ACCEPTED;
    }

    // Check the palette table.
    if (pVih->bmiHeader.biClrUsed > PALETTE_ENTRIES(pVih))
    {
        return VFW_E_TYPE_NOT_ACCEPTED;
    }
    DWORD cbPalette = pVih->bmiHeader.biClrUsed * sizeof(RGBQUAD);
    if (mtIn->cbFormat < sizeof(VIDEOINFOHEADER) + cbPalette)
    {
        return VFW_E_TYPE_NOT_ACCEPTED;
    }

    // Everything is good.
```

```
    return S_OK;
}
```

在这个例子中，函数首先检查 major type and subtype，然后检查格式类型，为了确保 block 格式是一个 **VIDEOINFOHEADER** 结构，这个 filter 也要支持 **VIDEOINFOHEADER2**,
如果格式类型是正确的，这个 sample 还得检查 **VIDEOINFOHEADER 结构的 biBitCount** and **biCompression** members，

2 virtual HRESULT GetMediaType(int iPosition, CMediaType *pMediaType) PURE;

CTransformFilter::GetMediaType 根据序号 iPositiong 返回一个 fiter 支持的输出类型。只有输入 pin 被连接上以后，这个方法才会被调用，因此，你可以利用上游 filter 支持的媒体类型来决定下游输出的媒体类型

下面的例子返回一个输出媒体类型，这个输出是根据输入类型修改的

```
HRESULT CRleFilter::GetMediaType(int iPosition, CMediaType *pMediaType)
{
    ASSERT(m_pInput->IsConnected());
    if (iPosition < 0)
    {
        return E_INVALIDARG;
    }
    if (iPosition == 0)
    {
        HRESULT hr = m_pInput->ConnectionMediaType(pMediaType);
        if (FAILED(hr))
        {
            return hr;
        }
        FOURCCMap fccMap = FCC('MRLE');
        pMediaType->subtype = static_cast<GUID>(fccMap);
        pMediaType->SetVariableSize();
        pMediaType->SetTemporalCompression(FALSE);

        ASSERT(pMediaType->formattype == FORMAT_VideoInfo);
        VIDEOINFOHEADER *pVih =
            reinterpret_cast<VIDEOINFOHEADER*>(pMediaType->pbFormat);
        pVih->bmiHeader.biCompression = BI_RLE8;
        pVih->bmiHeader.biSizeImage = DIBSIZE(pVih->bmiHeader);
        return S_OK;
    }
    // else
    return VFW_S_NO_MORE_ITEMS;
}
```

这个例子函数中，调用了 **IPin::ConnectionMediaType** 从输入 pin 上得到输入的媒体类型。然后改变了媒体类型结构的几个 filed，表示是压缩格式

1 It assigns a new subtype GUID, which is constructed from the FOURCC code 'MRLE', using the FOURCCMap class.

2 It calls the CMediaType::SetVariableSize method, which sets the bFixedSizeSamples flag to FALSE and the lSampleSize member to zero, indicating variable-sized samples.
3 It calls the CMediaType::SetTemporalCompression method with the value FALSE, indicating that every frame is a key frame. (This field is informational only, so you could safely ignore it.)
4 It sets the biCompression field to BI_RLE8.
5 It sets the biSizeImage field to the image size.

3 virtual HRESULT CheckTransform(const CMediaType* mtIn, const CMediaType* mtOut) PURE;

CTransformFilter::CheckTransform 检查输出的媒体类型和输入的媒体类型是否匹配。当输入 pin 在输出 pin 连接之后才开始连接的时候，输出 pin 会调用这个函数来检查输出媒体类型是否和输入媒体类型是否匹配。
下面的例子演示了查询数据的格式是否为 RLE8 视频，图像的大小是否和输入的匹配，调色板的入口是否一致，如果图像大小不一致就要拒绝

```
HRESULT CRleFilter::CheckTransform(
    const CMediaType *mtIn, const CMediaType *mtOut)
{
    // Check the major type.
    if (mtOut->majortype != MEDIATYPE_Video)
    {
        return VFW_E_TYPE_NOT_ACCEPTED;
    }

    // Check the subtype and format type.
    FOURCCMap fccMap = FCC('MRLE');
    if (mtOut->subtype != static_cast<GUID>(fccMap))
    {
        return VFW_E_TYPE_NOT_ACCEPTED;
    }
    if ((mtOut->formattype != FORMAT_VideoInfo) ||
        (mtOut->cbFormat < sizeof(VIDEOINFOHEADER)))
    {
        return VFW_E_TYPE_NOT_ACCEPTED;
    }

    // Compare the bitmap information against the input type.
    ASSERT(mtIn->formattype == FORMAT_VideoInfo);
    BITMAPINFOHEADER *pBmiOut = HEADER(mtOut->pbFormat);
    BITMAPINFOHEADER *pBmiIn = HEADER(mtIn->pbFormat);
    if ((pBmiOut->biPlanes != 1) ||
        (pBmiOut->biBitCount != 8) ||
        (pBmiOut->biCompression != BI_RLE8) ||
        (pBmiOut->biWidth != pBmiIn->biWidth) ||
        (pBmiOut->biHeight != pBmiIn->biHeight))
    {
```

```
            return VFW_E_TYPE_NOT_ACCEPTED;
        }

        // Compare source and target rectangles.
        RECT rcImg;
        SetRect(&rcImg, 0, 0, pBmiIn->biWidth, pBmiIn->biHeight);
        RECT *prcSrc = &((VIDEOINFOHEADER*)(mtIn->pbFormat))->rcSource;
        RECT *prcTarget = &((VIDEOINFOHEADER*)(mtOut->pbFormat))->rcTarget;
        if (!IsRectEmpty(prcSrc) && !EqualRect(prcSrc, &rcImg))
        {
            return VFW_E_INVALIDMEDIATYPE;
        }
        if (!IsRectEmpty(prcTarget) && !EqualRect(prcTarget, &rcImg))
        {
            return VFW_E_INVALIDMEDIATYPE;
        }

        // Check the palette table.
        if (pBmiOut->biClrUsed != pBmiIn->biClrUsed)
        {
            return VFW_E_TYPE_NOT_ACCEPTED;
        }
        DWORD cbPalette = pBmiOut->biClrUsed * sizeof(RGBQUAD);
        if (mtOut->cbFormat < sizeof(VIDEOINFOHEADER) + cbPalette)
        {
            return VFW_E_TYPE_NOT_ACCEPTED;
        }
        if (0 != memcmp(pBmiOut + 1, pBmiIn + 1, cbPalette))
        {
            return VFW_E_TYPE_NOT_ACCEPTED;
        }

        // Everything is good.
        return S_OK;
}
```

第四步 设置 Allocator 属性

当连个 pin 就某个媒体类型达成一致协议的时候，他们就选择一个 allocator，就 allocator 的属性进行设置，比如 buffer 大小，buffer 的数量。

在 **CTransformFilter** 类中，有两个 allocator，一个用于上游的 pin 的连接，一个用于下游的 pin 的连接，上游的 filter 选择 upstream allocator 设置属性，无论上游的 filter 怎么设置这个 upstream allocator，输入 pin 都会接受，如果你想改变这个中状况，你可以继承 **CBaseInputPin::NotifyAllocator** 函数

Transform filter 的输出 pin 选择下游的 allocator，步骤如下

1 如果下游的 filter 可以提供一个 allocator，那么输出 pin 就使用这个 allocator，否则，输出 pin 就创建一个新的 allocator。
2 输出 pin 通过下游 filter 的输入 pin 上的 **IMemInputPin::GetAllocatorRequirements**.方法来确定下游 filter 的 allocator 的要求。
3 输出pin调用transform filter上的**CTransformFilter::DecideBufferSize**函数，这个函数也是一个纯虚的函数，

```
virtual HRESULT DecideBufferSize( IMemAllocator * pAllocator,
                                  ALLOCATOR_PROPERTIES *pprop) PURE;
```

这个函数有一个指向 allocator 的指针，和一个指向 **ALLOCATOR_PROPERTIES** 结构的指针，这个指针包含了对 allocator 的属性的设置，如果下游的 filter 对 allocator 没有设置属性，那么这个结构就是 NULL。
4 在 DecideBufferSize 方法中，派生类的函数通过调用 **IMemAllocator::SetProperties**.函数来设置 allocator 的属性。
通常，派生类会根据输出的格式，下游 filter 得要求，自身得要求来设置 allocator 的属性，allocator 属性的设置要符合下游 filter 的要求，否则的话，连接就可能被拒绝。
下面的例子中，

```
HRESULT CRleFilter::DecideBufferSize(
    IMemAllocator *pAlloc, ALLOCATOR_PROPERTIES *pProp)
{
    AM_MEDIA_TYPE mt;
    HRESULT hr = m_pOutput->ConnectionMediaType(&mt);
    if (FAILED(hr))
    {
        return hr;
    }

    ASSERT(mt.formattype == FORMAT_VideoInfo);
    BITMAPINFOHEADER *pbmi = HEADER(mt.pbFormat);
    pProp->cbBuffer = DIBSIZE(*pbmi) * 2;
    if (pProp->cbAlign == 0)
    {
        pProp->cbAlign = 1;
    }
    if (pProp->cBuffers == 0)
    {
        pProp->cBuffers = 1;
    }
    // Release the format block.
    FreeMediaType(mt);

    // Set allocator properties.
    ALLOCATOR_PROPERTIES Actual;
    hr = pAlloc->SetProperties(pProp, &Actual);
    if (FAILED(hr))
```

```
    {
        return hr;
    }
    // Even when it succeeds, check the actual result.
    if (pProp->cbBuffer > Actual.cbBuffer)
    {
        return E_FAIL;
    }
    return S_OK;
}
```

即使 SetProperties 函数成功，你也要检查结果，以确保满足你的需要

缺省的情况下，所有的 filter 都采用 CMemAllocator 类类分配内存，这个类从客户进程的虚拟地址中分配内存，如果你的 filter 需要其它的内存，比如，DirectDraw 表面，你可以派生一个通用的 allocator，你可以从 **CBaseAllocator** 类派生一个新的类，根据不同的 pin 使用你的派生的新的 allocator 类，你需要继承不同的函数，

Input pin: **CBaseInputPin::GetAllocator** and **CBaseInputPin::NotifyAllocator**.

Output pin: **CBaseOutputPin::DecideAllocator**.

如果其它的 filter 拒绝使用你的 custom allocator，你的 filter 和其它 filter 连接的时候就会失败，

第五步 传递媒体数据

上游 filter 通过调用 filter 上输入 pin 上的 **IMemInputPin::Receive** 方法，将 sample 传递到 filter，filter 调用 **CTransformFilter::Transform** 方法来处理数据，注意，这个方法也是一个纯虚的函数，你要是想用，你必须提供函数实现。

CTransformFilter::Transform 有两个指针，一个指向输入 sample，一直只想输出 smaple，再调用这个方法之前，要将 sample 从输入 sample 拷贝到输出 sample。

如果 transform 返回 S_ok，filter 就将 sample 传递到下游的 filter。下面的代码演示了 RLE encoder 如何实现这个函数的，你可以参考一下，当然你的函数和这个是不一样的。要注意

```
HRESULT CRleFilter::Transform(IMediaSample *pSource, IMediaSample *pDest)
{
    // Get pointers to the underlying buffers.
    BYTE *pBufferIn, *pBufferOut;
    hr = pSource->GetPointer(&pBufferIn);
    if (FAILED(hr))
    {
        return hr;
    }
    hr = pDest->GetPointer(&pBufferOut);
    if (FAILED(hr))
    {
        return hr;
    }
    // Process the data.
    DWORD cbDest = EncodeFrame(pBufferIn, pBufferOut);
    KASSERT((long)cbDest <= pDest->GetSize());
```

```
    pDest->SetActualDataLength(cbDest);
    pDest->SetSyncPoint(TRUE);
    return S_OK;
}
```

需要注意的几个问题

1 时间戳，CTransformFilter 在调用 Transform 方法之前就给输出 sample 打上了时间戳，它仅仅是从输入的 stample 上讲时间戳拷贝过来，不做任何的改动，如果你的 filter 需要改动时间戳，你可以调用输出 sample 上的 **IMediaSample::SetTime** 方法

2 数据格式的改变

上游的 filter 可以 动态的改变数据的格式，在改动数据的格式之前，它要调用你的输入 pin 上的 **IPin::QueryAccept** 方法，在 filter 上，这个方法的调用会引起 **CheckInputType，和 CheckTransform** 的方法的调用。下游的 filter 也可以改变数据格式，机理和这个一样。

在你的 filter 中，需要做两件事情

1 ）要确保 QueryAccept 返回正确

2 ）如果你的 filter 不接受数据格式的改变，那么就在你的 filter 的 Transform 方法中调用 **IMediaSample::GetMediaType**.方法，如果这个方法返回 s_ok，那么你的 filter 就要适用数据的改变。

3 线程，

在 **CTransformFilter** 中，filter 在 Receive 方法同步的发送输出 sample。Filter 没有创建任何的线程来处理数据。

第六步支持 COM 特性

最后一步是支持 com 属性

添加 com 支持的步骤和前面一样，并且在前面也讲述的很清楚了，下面列出必须的几个要素

1 引用计数 **Reference Counting**，接口查询 **QueryInterface**

很简单，从基类派生即可

```
CMyFilter : public CBaseFilter, public IMyCustomInterface
{
public:
    DECLARE_IUNKNOWN
    STDMETHODIMP NonDelegatingQueryInterface(REFIID iid, void **ppv);
};
STDMETHODIMP CMyFilter::NonDelegatingQueryInterface(REFIID iid, void **ppv)
{
    if (riid == IID_IMyCustomInterface) {
        return GetInterface(static_cast<IMyCustomInterface*>(this), ppv);
    }
    return CBaseFilter::NonDelegatingQueryInterface(riid,ppv);
}
```

2 对象的创建 **Object Creation**

```
CUnknown * WINAPI CRleFilter::CreateInstance(LPUNKNOWN pUnk, HRESULT *pHr)
{
    CRleFilter *pFilter = new CRleFilter();
    if (pFilter== NULL)
```

```
    {
        *pHr = E_OUTOFMEMORY;
    }
    return pFilter;
}
```

模板数组

```
static WCHAR g_wszName[] = L"My RLE Encoder";
CFactoryTemplate g_Templates[] =
{
  {
    g_wszName,
    &CLSID_RLEFilter,
    CRleFilter::CreateInstance,
    NULL,
    NULL
  }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
```

3 组件的注册

```
// Declare media type information.
FOURCCMap fccMap = FCC('MRLE');
REGPINTYPES sudInputTypes = { &MEDIATYPE_Video, &GUID_NULL };
REGPINTYPES sudOutputTypes = { &MEDIATYPE_Video, (GUID*)&fccMap };

// Declare pin information.
REGFILTERPINS sudPinReg[] = {
    // Input pin.
    { 0, FALSE, // Rendered?
        FALSE, // Output?
        FALSE, // Zero?
        FALSE, // Many?
        0, 0,
        1, &sudInputTypes    // Media types.
    },
    // Output pin.
    { 0, FALSE, // Rendered?
        TRUE, // Output?
        FALSE, // Zero?
        FALSE, // Many?
        0, 0,
        1, &sudOutputTypes          // Media types.
    }
};
```

```
// Declare filter information.
REGFILTER2 rf2FilterReg = {
    1,                  // Version number.
    MERIT_DO_NOT_USE, // Merit.
    2,                  // Number of pins.
    sudPinReg           // Pointer to pin information.
};

STDAPI DllRegisterServer(void)
{
    HRESULT hr = AMovieDllRegisterServer2(TRUE);
    if (FAILED(hr))
    {
        return hr;
    }
    IFilterMapper2 *pFM2 = NULL;
    hr = CoCreateInstance(CLSID_FilterMapper2, NULL, CLSCTX_INPROC_SERVER,
            IID_IFilterMapper2, (void **)&pFM2);
    if (SUCCEEDED(hr))
    {
        hr = pFM2->RegisterFilter(
            CLSID_RLEFilter,                    // Filter CLSID.
            g_wszName,                          // Filter name.
            NULL,                               // Device moniker.
            &CLSID_VideoCompressorCategory,     // Video compressor category.
            g_wszName,                          // Instance data.
            &rf2FilterReg                       // Filter information.
            );
        pFM2->Release();
    }
    return hr;
}

STDAPI DllUnregisterServer()
{
    HRESULT hr = AMovieDllRegisterServer2(FALSE);
    if (FAILED(hr))
    {
        return hr;
    }
    IFilterMapper2 *pFM2 = NULL;
    hr = CoCreateInstance(CLSID_FilterMapper2, NULL, CLSCTX_INPROC_SERVER,
            IID_IFilterMapper2, (void **)&pFM2);
    if (SUCCEEDED(hr))
```

```
    {
        hr = pFM2->UnregisterFilter(&CLSID_VideoCompressorCategory,
            g_wszName, CLSID_RLEFilter);
        pFM2->Release();
    }
    return hr;
}
```

有时候，你的 filter 并不总是通过 DLL 提供的，有时，你可能给一个特定的程序写了一个特定的 filter，那么你就可以直接用你的 filter，你可以直接用 new 方法，如下

```
#include "MyFilter.h"    // Header file that declares the filter class.
// Compile and link MyFilter.cpp.
int main()
{
    IBaseFilter *pFilter = 0;
    {
        // Scope to hide pF.
        CMyFilter* pF = new MyFilter();
        if (!pF)
        {
            printf("Could not create MyFilter.\n");
            return 1;
        }
        pF->QueryInterface(IID_IBaseFilter,
            reinterpret_cast<void**>(&pFilter));
    }

    /* Now use pFilter as normal. */

    pFilter->Release();    // Deletes the filter.
    return 0;
}
```

8 如何写视频播放过滤器 Video Renderer Filter

1 开发一个可选择的视频播放 filter

Directshow 提供了一个基于窗口的视频播放 Filter，它也提供了一个全屏幕实时播放的 filter。你可以利用 Directshow 的基类开发自己的可选择的视频播放 filter。你可以利用 CBaseRenderer and CBaseVideoRenderer 类根据下面的一些经验指南就可以开发一个可选择的视频播放 filter。

这篇文章讨论了一个播放 Filter 需要处理的一些消息通知。只要正确的处理这些消息通知，才能够正确地设置 Directshow 播放视频的画面。

在 Directshow 中主要有三种消息通知。

1 数据流的通知，

这是数据流在 graph 图中传递的过程中，从一个 filter 到另一个 filter 的过程中发生的事件通知。例如，**begin-flushing, end-flushing or end-of-stream 事件通知**，这些事件通知都是通过上游 fitlter 调用的下游 filter 的输入 pin 来通知下游 filter 的。比如 **IPin::BeginFlush**。

2 filter 图表管理器发送的消息通知，

这些都是 filter 给图表管理器发送的事件通知，比如 EC_COMPLETE，这种消息一般都是通过图表管理器的 **IMediaEventSink::Notify** 发送或接收的。

3 应用程序发送的消息通知

应用程序通过调用图表管理器上的 **IMediaEvent::GetEvent** 方法就可以获得这些事件消息。一般来说，图表管理经常讲得到的消息传递给应用程序处理。

2 对 End-of-stream and Flushing 消息的处理

当源 filter 发现没有数据传送的时候，它就会向下游发送一个 end-of-stream 通知，这个通知会沿着 Graph 图表中的 filter 一个一个的往下传递，最后到达 Render Filter。这就导致 Graph 图表产生一个 EC_COMPLETE 消息。

当 Renderer 的输入 pin 上的 **IPin::EndOfStream** 被上游的 filter 调用的时候，Render Filter 就会接收到一个 end-of-stream 消息通知。Render Filter 应该标记下这个消息通知，然后将已经接收到的数据处理完毕。当所有剩余的数据接收完毕，Render Filter 就会给 Graph 图表管理器发送一个 EC_COMPLETE 消息。当 Render Filter 在处理完毕所有的数据时，你应该给 graph 图表管理器发送一次 **EC_COMPLETE** 消息。只有当 Render Filter 处于运行状态时才能给图表管理器发送 **EC_COMPLETE** 消息。如果一个 Render 正处于 paused 状态时接收到源 filter 发送的 end-of-stream 通知，只有当 Filter Graph 结束的时候 Render 才能给图表管理器发送 **EC_COMPLETE 消息。**

end-of-stream 通知发出以后，如果上游 filter 再次调用 Render Filter 上的输入 pin 上的 **IMemInputPin::Receive** or **IMemInputPin::ReceiveMultiple** 方法时，Render 就要拒绝它，此时就返回一个 E_UNEXPECTED 错误消息。

当 Filter 图表管理器停止的时候，Render 应该将捕捉到的任何 end-of-stream 通知都应该被清除，当图表管理器再次启动的时候，Render 不应该再给管理器发送任何通知。因为图表管理器在启动以前会 Paused 所有的 Filter，这就会导致发生 flushing。例如，如果 Filter graph 处于 Pause 状态收到一个 end-of-stream 通知，然后 Filter Graph 就停止了，当 filter Graph 再次运行的时候，Render 就不应该给管理器发送 EC_COMPLETE 消息了。如果没有发生 seek，源 filter 会自动地在发生 pause 时给下游的 filter 发送一个 end-of-stream 通知，如果在 filter Graph 图表 stop 的时候发生 seek，此时源 filter 也许正有数据要发送，所以它不会发送 end-of-stream 通知。

Render filter 经常依靠 end-of-stream 通知来发送 EC_COMPLETE 通知。例如，如果一个数据流已经结束发送（也就是 end-of-stream 消息已经发送出来），另一个窗口已经覆盖到视频窗口上，也产生了一系列的 WM_PAINT 消息。但是，当 end-of-stream 消息发出以后，Render 就处于等待状态，但是 Render 也明白它不会再接收任何数据了，视频播放窗口就会出现黑屏幕。

Flushing 是 Render 应该处理的另外一种复杂的事件。Flushing 消息是通过 IPin 上的两个方法 **BeginFlush** and **EndFlush**.触发的。源 filter 在没有调用 **EndFlush**,而仅仅调用了 **BeginFlush** 方法是不合法的，所以此时 Flushing 的状态是短暂和不连续的。但是在 flushing 状态下，Render 要处理好数据以及接收的消息。

在 **BeginFlush** 方法被调用之后所有接收到的数据都要立即被 rejected，并且返回一个 S_FASLE。并且捕捉到的 end-of-stream 也要立即清除。当 Render 接收到 seek 消息后，就立即处于 flush 状态。Flush 确保在重新发送数据之前从 filter Graph 中清除所有的旧的数据。

3 如何处理状态的改变 Handling State Changes and Pause Completion

当一个 Renderer filter 的状态改变的时候，它的行为和其他的 filter 是一样的，但是也有以下的区别，当 filter 的状态变为 pause 的时候，Render filter 的数据就排成队列，等待下次播放，当一个 video 播放 filter 停止的时候，它也会保留这些队列中的数据，这就是一个例外，因为 dshow 规定，当一个 graph 停止的时候，它不应该保留任何资源。

造成这种例外的原因是如果 render filter 保持资源，这样，当这个 filter 接收到一个 WM_PAINT 消息时可以通过这个资源来重绘窗口。同样保持这个资源也可以满足一些方法的调用，比如 **CBaseControlVideo::GetStaticImage**,，这个方法用来返回当前图像的一份拷贝。保持资源的另一个原因是，在资源保持的过程中，它所占用的内存块不会被回收，这样，再次开始数据传输时速度会比较快一点，因为不用分配内存了。

在 graph 运行的期间，sample 中的内容会被随时地提交 sample 内存也随时地被释放，但是，在停止运行的状态中，sample 只能被提交，不能被释放，例如，在窗口绘制一幅静态的图画。音频流在停止的状态没法被提交，但是他们可以进行其他的动作，比如准备 wave 设备。Sample 被提交的时间由 sample 的 stream time 和 **IMediaControl::Run** 方法调用时传递的参考时间综合以后得到的。当开始时间小于等于结束时间时，sample 就应该被丢弃。

当应用程序调用 **IMediaControl::Pause** 方法准备停止一个 graph 图时，只有当提交过滤器中有一个数据队列时才能够返回。为了确保此点，当一个 render filter 没有等待提交的数据时，该方法就返回 S_FALSE,如果有数据等待提交，返回 S_OK。

未来确保 Render filter 有一个等待提交的数据，Filter 图表管理器在停止一个 graph 时会检查方法的返回值，如果一个或者几个 filter 还没有准备好，filter 图表管理器就会调用 **GetState** 方法来 polls filter。GetState 方法带有一个超时的参数，当 GetState 函数的等待的时间到期返回之前，如果 filter 还在等待数据的到来时，那么该函数返回 VFW_S_STATE_INTERMEDIATE，如果 filter 已经有等到数据的时候，GetState 返回 s_ok。

当一个 filter 在等待数据的时候，源 filter 会发送一个 end of stream 的通知，此时状态的转变完成。

当一个 graph 中的所有的 filter 都有了等待提交的数据，那么整个 graph 就成为了 pause 状态。

4 如何处理终止态（Handling Termination）

视频提交过滤器必须能够正确处理来自用户的终止数据流的事件。这就意味着要正确地隐藏窗口，并且知道当窗口重新显示的时候该怎么做。同时，当窗口销毁的时候，提交过滤器也要能够通知 Filter 图表管理器来正确的释放资源。

当用户关闭了视频窗口时，或者（用户按 ALT+F4），一般的做法是将视频窗口隐藏同时给 filter 图表管理器发送一个 EC_USERABORT 通知，这个通知最终会被发送到应用程序，然后应用程序就会停止播放视频。当发出 EC_USERABORT 通知后，所有发送给提交 filter 的数据都会被拒绝。

当一个视频正在在播放的时候，如果用户此时按下 ATL +F4，视频窗口就会暂时的隐藏起来，然后所有送往窗口的数据都会被拒绝。当窗口重新显示的时候，不会产生 EC_REPAINT 通知。

当一个视频提交 filter 终止的时候，它要给 filter 图表管理器发送一个 EC_WINDOW_DESTROYED 消息通知。事实上，最好的处理这个消息的时机是在 **IBaseFilter::JoinFilterGraph** 方法调用时，而不是等到实际的窗口销毁时。Sending this notification enables the plug-in distributor in the Filter Graph Manager to pass on resources that depend on window focus to other filters (such as audio devices).

5 如何处理数据格式的动态改变

视频提交 filter 一般只接受那些容易处理的数据格式，例如，一般只接受 RGB 格式的数据，因为这种数据格式和显示器格式相匹配。

通常的话，上游的 filter 都是调用下游 filter 上的输入 pin 上的 **IPin::QueryAccept** 方法来查询，下游的 filter 是否接受新的数据格式，从而来动态的改变数据格式。一个 render filter 应该支持动态的修改数据格式，至少它应该允许上游的 filter 能够改变调色板。当上游 filter 改变媒体类型，它会在采用新格式的第一个 smpale 上贴上新的媒体类型。如果一个 render filter 还有一些老格式的数据没有提交完，它会等到这些老格式的数据提交完毕才改变媒体类型。

解码器也会动态的改变数据格式，这样就要求 render filter 能够相应的跟着改变，例如，如果我们想要解码器提供一种和 DirectDraw 兼容的数据格式，当 render paused 的时候，它就开始通过 **QueryAccept** 向上游的 filter 询问，解码器都支持什么数据格式，解码器一般不会将它支持的所有数据格式都列举出来，因此，render filter 就要提供一些解码器接口没有说明的数据格式。

如果解码器可以接收要求的数据格式，它就会通过 QueryAccept 方法返回一个 s_ok，于是 Render filter 就将新的媒体类型 attach to 上游内存分配器分配的下一个 sample 上。因此，render filter 就要提供一个内存分配器，这个提供一个私有的方法来将媒体类型贴到新的下一个 samples 上，在这个私有的方法中，调用 **IMediaSample::SetMediaType** 来设置媒体类型。

Render filter 的输入 pin 应该在 **IMemInputPin::GetAllocator** 方法中返回 render filter 的内存分配器，重载一下 **IMemInputPin::NotifyAllocator** 方法，如果上游的 filter 不使用 render filter 提供的内存分配器，那么这个方法就会返回 false。

在一些解码器中，如果将 biHeight 设置为一个 YUV 类型的正数，那么就解码器就会产生上下颠倒的画面，这是不正确的，应该视为解码器的一个 bug。

当 render filter 发现 graph 中的数据格式发生改变的时候，它都会发送一个 EC_DISPLAY_CHANGED 消息通知的。大多数的 render filter 在连接的时候都会选择一个 GDI 支持的数据格式，如果用户改变了当前的显示模式而没有重新启动机器，那么 render filter 发现自己正使用一种很糟糕的数据格式进行连接，那么它就会发送上面的消息通知。第一个参数就是需要重新连接的 pin，管理器就会让 graph 图停止运行，重新连接 pin。在随后的重新连接过程中，render filter 就会选择接受合适的数据格式。

当 render fitler 发现调色板发生改变的时候，它要发送 EC_PALETTE_CHANGED 的消息通知给 filter 图表管理器。

最后，视频 render filter 发现视频的尺寸发生改变，它会给图表管理器发送一个 EC_VIDEO_SIZE_CHANGED 消息通知。

6 如何处理永久性属性（Persistent Properties）

所有通过 **IBasicVideo** and **IVideoWindow** 接口设置的属性都意味着在整个连接过程中是永久不不变的。因此，断开连接，重新连接都对窗口的大小，位置，样式等属性没有影响。但是，如果视频的尺寸大小发生改变的时候，render filter 应该重新设置源或者目的的矩形大小。源或者目的的位置是通过 **IBasicVideo** 设置的。

IBasicVideo and **IVideoWindow** 提供了足够的接口方法可以让应用程序来一种永久的格式来保存或者存储经过接口的数据。

7 如何处理 EC_REPAINT 通知

当一个 render filter 暂停或者停止的时候，它发送一个 EC_REPAINT 消息。这个消息告诉 filter 图表管理器 render filter 需要数据。如果一个 filter 管理器准备停止的时候接收到这个消

息，它会首先暂停 filter graph，等到所有的 filter 都接收到了数据（通过调用 **GetState**），然后它再重新停止 graph。当处于停止状态，一个 render filter 应该保存一副图画，这样可以在处理 WM_PAINT 消息的时候来显示这幅图画。

当一个 render filter 在停止或者暂停的时候收到 WM_PAINT 消息时，如果它没有任何的数据用来显示，它也会给 filter 图表管理器发送一个 EC_REPAINT 消息。当处于暂停状态的 filtr 图表管理器接收到一个 **EC_REPAINT** 消息时，图表管理器就会调用 **IMediaPosition::put_CurrentPosition** 方法，以当前的位置为参数，这个方法的调用就会导致源 filter flush 图表管理器，然后通过图表管理器发送新的数据给 render filter。

Render filter 一般只发送一次这样的消息通知，也就是说，如果一个 render filter 发送了一个 ec_repaint 消息，在数据到来之前它不应该再发送 ec_repaint 消息了，因此，通常的做法设置一个标志用来标示已经发送了一个 ec_repaint 消息，当 render 接收到数据或者输入 pin 被 flushed 以后，这个标志应该被重置。当然，如果输入 pin 接收到 end-of-stream 通知的时候，这个标志不重置。

如果一个 render filter 不控制它的 EC_REPAINT 消息的时候，那么整个 filter 图表管理器都会被 EC_REPAINT 消息淹没的。例如，如果一个 render 没有图像显示的时候，如果一个窗口从 render 窗口上拖动，那么 render 窗口就会接收到大量的 wm_paint 消息的，只有第一个消息可以产生 **EC_REPAINT 事件**。

Render filter 应该将它的输入 pin 作为 **EC_REPAINT** 消息的第一个参数，by doing this，rendfilter 首先向附加的输出 pin（不知道这么说是否正确，原文 the attached output pin will be queried for **IMediaEventSink**） 请求 **IMediaEventSink**,接口，如果输出 pin 支持这个接口，那么 Ec_REPAINT 消息就首先通过这里发送出去。这样就使输出 pin 在 graph 收到消息前能够处理 repain 事件。如果 graph 没有处于停止的状态，输出 pin 不会处理这个消息，因为没有空闲的内存数据块。

如果输出 pin 不能够处理这个请求，或者 graph 正在运行，那么 EC_REPAINT 消息就会被丢弃。输出 pin 上的 **IMediaEventSink::Notify** 方法调用，返回 S_OK 表明输出 pin 可以正确的处理 repaint 消息。在 graph 的工作线程中会调用输出 pin 的，这样就避免了 render 直接调用输出 pin，防止死锁。如果 graph 处于停止状态或者暂停状态，或者是输出 pin 不能处理这个请求，然后就会有缺省的处理过程来处理这个消息。

8 如何处理全屏幕显示

IVideoWindow 插件管理 graph 的全屏回放。它可以控制一个 render filter 的窗口伸展成一个全屏来显示图像，或者直接用一个全屏的 filter 来直接回放。当一个 filter 从普通状态到全屏显示转化的过程都要发送 EC_ACTIVATE 消息，无论 activated or deactivated.。也就是说，render filter 在接收到一个 WM_ACTIVATEAPP 一定要发送 EC_ACTIVATE 消息。

当一个 filter 处于全屏模式的时候，这些消息用来管理是进入还是退出全屏模式。

当 graph 接收到一个 EC_ACTIVATE 消息通知准备退出全屏模式，那么 graph 就发送一个 EC_FULLSCREEN_LOST 给应用程序，应用程序也许会利用这个消息来保存全屏按钮的状态。

9 消息通知小结 Notifications

1 EC_ACTIVATE

说明；render filter 在接收到一个 WM_ACTIVATEAPP 一定要发送 EC_ACTIVATE 消息。

2 EC_COMPLETE

当所有的数据都提交完毕的时候，发送此消息

3 EC_DISPLAY_CHANGED

当显示的格式发生变化的时候，发送此消息

4 EC_PALETTE_CHANGED
当调色板发生变化的时候，发送此消息
5 EC_REPAINT
重画的时候，只发送一次
6 EC_USERABORT
当用户关闭的时候
7 EC_VIDEO_SIZE_CHANGED
当视频的尺寸发生变化的时候
8 EC_WINDOW_DESTROYED
当 render filter 销毁的时候，发送此消息

10Render 中的源和目标矩形

在 **VIDEOINFO**, **VIDEOINFOHEADER**, and **VIDEOINFOHEADER2** 三种媒体结构中有三个尺寸。
这篇文档就是想解释一下这三个尺寸有何不同，以及他们是用来做什么的。
首先，在这些结构中有一个 **bmiHeader** 数据成员，这个成员是 **BITMAPINFOHEADER** 结构，这个结构的定义如下

```
typedef struct tagBITMAPINFOHEADER {
    DWORD  biSize;
    LONG   biWidth;
    LONG   biHeight;
    WORD   biPlanes;
    WORD   biBitCount;
    DWORD  biCompression;
    DWORD  biSizeImage;
    LONG   biXPelsPerMeter;
    LONG   biYPelsPerMeter;
    DWORD  biClrUsed;
    DWORD  biClrImportant;
} BITMAPINFOHEADER;
```

这个结构有两个结构成员，biWidth 和 biHeight。
第二，在这些结构中有一个 rcSource 数据成员，同时也有一个 rcTarget 成员，假如你有两个 filter，A 和 B，假如这两个 filter 以某一种媒体数据类型 相连，A 在左边，上游的 filter，B 在右边，下游的 filter。
在这两个 filter 间传递的 buffer 具有一定的尺寸，可以用 **bmiHeader.biWidth**, **bmiHeader.biHeight 来标示**。
Filter A 的输入视频流由 **rcSource 的大小**控制，Filter 应该将输入视频的一部分扩展填充到 buffer 中 rcTarget 区域中，填充的一部分的大小，是根据 rcSource 和数据媒体类型的大小比较结果而定的。也就是 rcSource 和(**biWidth**, **biHeight**) 比较。如果 rcSource 为空，Filter A 就会将全部的输入 pin 拷贝到 rcTarget，如果 rcTarget 为空，那么 Filter A 就会将视频填充到整个的输出 buffer 中。举例如下：
假定 Filter A 接收的视频图像为 160*120 单位为象素，假定 A 和 B 连接的时候采用如下的数据类型
（biWidth，biHeight）：320， 240
RcSource：（0，0，0，0）

RcTarget（0，0，0，0）

这就意味着 Filter A 就会将它接收到的视频数据 x 方向和 y 方向乘于 2 以后填充到 320*240 的输出 bufer 中。

又假如 Filter A 接收的视频图像为 160*120 单位为象素，假定 A 和 B 连接的时候采用如下的数据类型

（biWidth，biHeight）：320， 240

RcSource：（0，0，160，240）

RcTarget（0，0，0，0）

两个 filter 连接的 buffer 是 320*240，因为指定的 rcSource 指定了 buffer 的左半部分，Filter A 就将输入视频的左半部分或者（0，0，80，120）部分，然后将视频扩展到 320*240（x 方向*4，y 方向*2）然后填充到 320*240 的输出 buffer 中。

现在我们假定 Filter A 调用 **CBaseAllocator::GetBuffer**,方法，这个方法的返回的 sample 会附着一个媒体类型，用来标示 Filter B 期望 Filter A 能够提供一个和前面的视频流具有不同 size 或者格式的数据。假定 新的媒体类型如下

(biWidth, biHeight): 640, 480

rcSource: (0, 0, 160, 120)

rcTarget: (0, 0, 80, 60)

这就意味着 sample 具有一个 640*480 的 buffer，原来的 rcSource 适用于原来的(320, 240)的媒体类型不适用于新的媒体格式，因此，rcSource 指定输入只使用左上角的四分之一，这一部分被放置到 rcTarget 的输出 buffer 的左上角（80，60），因为 Filte A 接收 160*120 的视频，输入视频的左上角正好是（80，60），和输出的位图一样大，不用扩展

Filter A 不会在输出 buffer 的其它部分放置数据，The **rcSource** member is bounded by the **biWidth** and **biHeight** of the original connected media type between filters A and B, and **rcTarget** is bounded by the new **biWidth** and **biHeight** of the media sample.上面的例子中，rcSource 就在（0，0，320，240）范围内，rcTarget 就在（0，0，640，480）范围内；

9 如何写捕捉 filter（源）

我们一般不推荐自己开发音频或者视频捕捉过滤器，因为 diectshow 对于音视频的捕捉设备以经提供了支持。所以，这篇文档，对于某些用户需要从特定设备捕捉一些数据提供一些帮助。这篇文档主要包括以下内容。

1 捕捉 filter 对 pin 的要求

2 如何完成一个预览 pin

3 如何产生源数据

1 对 pin 的要求 Pin Requirements for Capture Filters

Pin 的名字

你可以给你的 filter 起任何名字，如果你的 pin 的名字以~符号开头，那么当应用程序调用 **IGraphBuilder::RenderFile** 方法时，filter 图表管理器不会自动 render 这个 pin 的。例如，如果一个 filter 具有一个捕捉 pin 和预览 pin，相应的你给他们命名为”~Capture”和“Preview”。如果一个应用程序在 graph 中 render 这个 filter ，那么预览 pin 就会自动和它缺省的 render 相连接，但是，capture pin 上却不连接任何东西，这是一个合理的缺省行为。这个也可以应用到那些传输不准备被 rendered 数据的 pin，也可以应用到需要属性设置的 pin 上。

注：名字中含有~符号的 pin 是可以手动连接的。

Pin 的种类

一个捕捉 filter 通常用一个捕捉 pin，也许还有一个预览 pin。一些捕捉 filter 除了这两种 pin 之外还有其他的 pin，用来传递其他的数据，例如控制信息。每一个输出 pin 都必须暴露 **IKsPropertySet** 接口，应用程序通过这些接口来判断 pin 的种类，pin 一般都会返回 PIN_CATEGORY_CAPTURE or PIN_CATEGORY_PREVIEW。下面的例子演示了一个捕捉 pin 如何通过 **IKsPropertySet** 来返回 pin 的种类

```
// Set: Cannot set any properties.
HRESULT CMyCapturePin::Set(REFGUID guidPropSet, DWORD dwID,
      void *pInstanceData, DWORD cbInstanceData, void *pPropData,
      DWORD cbPropData)
{
      return E_NOTIMPL;
}

// Get: Return the pin category (our only property).
HRESULT CMyCapturePin::Get(
      REFGUID guidPropSet,      // Which property set.
      DWORD dwPropID,               // Which property in that set.
      void *pInstanceData,      // Instance data (ignore).
      DWORD cbInstanceData,     // Size of the instance data (ignore).
      void *pPropData,              // Buffer to receive the property data.
      DWORD cbPropData,             // Size of the buffer.
      DWORD *pcbReturned          // Return the size of the property.
)
{
      if (guidPropSet != AMPROPSETID_Pin)
            return E_PROP_SET_UNSUPPORTED;
      if (dwPropID != AMPROPERTY_PIN_CATEGORY)
            return E_PROP_ID_UNSUPPORTED;
      if (pPropData == NULL && pcbReturned == NULL)
            return E_POINTER;
      if (pcbReturned)
            *pcbReturned = sizeof(GUID);
      if (pPropData == NULL)    // Caller just wants to know the size.
            return S_OK;
      if (cbPropData < sizeof(GUID)) // The buffer is too small.
            return E_UNEXPECTED;
      *(GUID *)pPropData = PIN_CATEGORY_CAPTURE;
      return S_OK;
}

// QuerySupported: Query whether the pin supports the specified property.
HRESULT CMyCapturePin::QuerySupported(REFGUID guidPropSet, DWORD dwPropID,
```

```
    DWORD *pTypeSupport)
{
    if (guidPropSet != AMPROPSETID_Pin)
        return E_PROP_SET_UNSUPPORTED;
    if (dwPropID != AMPROPERTY_PIN_CATEGORY)
        return E_PROP_ID_UNSUPPORTED;
    if (pTypeSupport)
        // We support getting this property, but not setting it.
        *pTypeSupport = KSPROPERTY_SUPPORT_GET;
    return S_OK;
}
```

2 如何完成一个预览 pin Implementing a Preview Pin (Optional)

如果你的 filter 有一个预览 pin，预览 pin 发送的数据是捕捉 pin 传递的数据的拷贝。预览 pin 发送的数据不会降低捕捉 pin 的桢率，捕捉 pin 比预览 pin 有优先权。

捕捉 pin 和预览 pin 必须发送一个相同格式的数据。这样，他们连接都是通过同一种媒体数据类型，如果捕捉 pin 先连接，预览 pin 应该提供相同的媒体类型，对于其他类型的数据媒体，则拒绝。如果预览 pin 先连接，然后，如果捕捉 pin 以另一种媒体类型和其他 pin 连接，那么预览 pin 就应该用新的媒体类型重新连接，如果和 filter 的预览 pin 连接的下游 filter 拒绝新的数据类型，捕捉 pin 应该拒绝新的媒体类型。可以通过 **IPin::QueryAccept** 方法察看 filter 的预览 pin 连接的下游 filter 连接的数据媒体，然后通过 **IFilterGraph::Reconnect** 方法重新连接 pin。

这条规则也适用于图表管理器重新连接捕捉 pin。

下面的代码大体上描述了上面的过程

```
// Override CBasePin::CheckMediaType.
CCapturePin::CheckMediaType(CMediaType *pmt)
{
    if (m_pMyPreviewPin->IsConnected())
    {
        // The preview pin is already connected, so query the pin it is
        // connected to. If the other pin rejects it, so do we.
        hr = m_pMyPreviewPin->GetConnected()->QueryAccept(pmt);
        if (hr != S_OK)
        {
            // The preview pin cannot reconnect with this media type.
            return E_INVALIDARG;
        }
        // The preview pin will reconnect when SetMediaType is called.
    }
    // Decide whether the capture pin accepts the format.
    BOOL fAcceptThisType = ...   // (Not shown.)
    return (fAcceptThisType? S_OK : E_FAIL);
}
```

```
// Override CBasePin::SetMediaType.
CCapturePin::SetMediaType(CMediaType *pmt);
{
    if (m_pMyPreviewPin->IsConnected())
    {
        // The preview pin is already connected, so it must reconnect.
        if (m_pMyPreviewPin->GetConnected()->QueryAccept(pmt) == S_OK)
        {
            // The downstream pin will accept the new type, so it's safe
            // to reconnect.
            m_pFilter->m_pGraph->Reconnect(m_pMyPreviewPin);
        }
        else
        {
            return VFW_E_INVALIDMEDIATYPE;
        }
    }
    // Now do anything that the capture pin needs to set the type.
    hr = MyInternalSetMediaType(pmt);

    // And finally, call the base-class method.
    return CBasePin::SetMediaType(pmt);
}

CPreviewPin::CheckMediaType(CMediaType *pmt)
{
    if (m_pMyCapturePin->IsConnected())
    {
        // The preview pin must connect with the same type.
        CMediaType cmt = m_pMyCapturePin->m_mt;
        return (*pmt == cmt ? S_OK : VFW_E_INVALIDMEDIATYPE);
    }
    // Decide whether the preview pin accepts the format. You can use your
    // knowledge of which types the capture pin will accept. Regardless,
    // when the capture pin connects, the preview pin will reconnect.
    return (fAcceptThisType? S_OK : E_FAIL);
}
```

3 在源 filter 中产生数据 Producing Data in a Capture Filter

状态改变

一个捕捉 filter 在运行时会产生数据流。当 filter paused 的时候，不要发送数据，并且此时，图表管理器调用 **CBaseFilter::GetState** 方法会返回 VFW_S_CANT_CUE，这个返回值告诉图表管理器，filter 正处于 paused 状态，停止发送数据。下面的代码显示如何派生 GetState 方法

```
CMyVidcapFilter::GetState(DWORD dw, FILTER_STATE *pState)
```

```
{
    CheckPointer(pState, E_POINTER);
    *pState = m_State;
    if (m_State == State_Paused)
        return VFW_S_CANT_CUE;
    else
        return S_OK;
}
```

控制不同的数据流

一个捕捉 filter 应该支持 **IAMStreamControl** 接口，因此应用程序应该分别的打开和关闭每一个 pin，例如，一个应用程序可以只预览而没有捕捉，然后可以转换到捕捉模式不用重新构建 graph 图表。你可以通过 **CBaseStreamControl** 类来实现这个接口

时间戳

当一个 filter 捕捉了一个 sample，将在每一个 sample 上标上一个当前时间的时间戳。结束时间是开始时间加上持续时间。例如，如果一个 filter 每秒捕捉十个 sample，and the stream time is 200,000,000 units when the filter captures the sample, the time stamps should be 200000000 and 201000000. (There are 10,000,000 units per second.) 可以通过 **IReferenceClock::GetTime** 方法获得当前的参考时间，通过 **IMediaSample::SetTime** 给 sample 设置时间戳。

相连的 sample 上的时间戳必须是递增的，即使 filter pauses 也是这样，如果一个 filter 运行，停止，然后又运行，重新开始后产生的 sample 上的时间戳必须比停止前的 sample 上的时间戳要大一些。

预览 pin 上的 sample 没有时间戳，但是，预览 pin 一般比捕捉 pin 到达 render pin 稍晚，因此，render filter 将预览 pin 上的 sample 视作迟到的 sample，为了赶上其他的时间进度，也许会丢失一些数据，

10 创建 filter 属性页

本篇文档我们将要讲述如何给一个 filter 创建一个属性页，通过 **CBasePropertyPage** 基类。这篇文档的实例代码演示了创建属性页的步骤，这里我们假设我们要创建属性页的视频 filter 支持饱和度属性页，这个属性页有一个滑动条，用户可以通过这个滑动条来控制饱和度。

第一步，设置属性的机理

Filter 必须支持一种和属性页沟通的方式，通过属性页可以设置或者获取 filter 的属性，下面是可能的三种方式

1 暴露一个接口

2 通过 IDispatch 支持自动化属性

3 暴露 **IPropertyBag** 接口，并定义一系列的属性

下面的例子利用了一个普通的 COM 接口，叫做 ISaturaton，这并不是一个真正的 com 接口，只是我们用来在这里举例的，你也可以自己定义任何的 com 对象。

首先我们在一个头文件中声明接口的 ID 和定义。

```
// Always create new GUIDs! Never copy a GUID from an example.
DEFINE_GUID(IID_ISaturation, 0x19412d6e, 0x6401,
0x475c, 0xb0, 0x48, 0x7a, 0xd2, 0x96, 0xe1, 0x6a, 0x19);
```

```
interface ISaturation : public IUnknown
{
    STDMETHOD(GetSaturation)(long *plSat) = 0;
    STDMETHOD(SetSaturation)(long lSat) = 0;
};
```

你也可以用 IDL 定义接口，并用 MIDL 编译器创建头文件，然后在 Filter 上实现这个接口，这个例子采用“Get”，“Set”方法来设置饱和度的值，注意，修改这个 m_lSaturation 的值的时候一定要进行保护

```
class CGrayFilter : public ISaturation, /* Other inherited classes. */
{
private:
    CCritSec    m_csShared;       // Protects shared data.
    long        m_lSaturation; // Saturation level.
public:
    STDMETHODIMP GetSaturation(long *plSat)
    {
        if (!plSat) return E_POINTER;
        CAutoLock lock(&m_csShared);
        *plSat = m_lSaturation;
        return S_OK;
    }
    STDMETHODIMP SetSaturation(long lSat)
    {
        CAutoLock lock(&m_csShared);
        if (lSat < SATURATION_MIN || lSat > SATURATION_MAX)
        {
            return E_INVALIDARG;
        }
        m_lSaturation = lSat;
        return S_OK;
    }
};
```

当然你实现接口的一些细节可能和上面的代码不一致。反正你自己实现就是了

第二步，实现 ISpecifyPropertyPages 接口

做完了上一步，下面就要在你个 filter 中实现 ISpecifyPropertyPages 接口，这个接口只有一个方法，GetPages，这个方法返回 filter 所支持的所有的属性页的 CLSID。在这个例子里，Filter 只支持一个属性页，

首先产生一个 CLSID，并在头文件声明

```
// Always create new GUIDs! Never copy a GUID from an example.
DEFINE_GUID(CLSID_SaturationProp, 0xa9bd4eb, 0xded5,
0x4df0, 0xba, 0xf6, 0x2c, 0xea, 0x23, 0xf5, 0x72, 0x61);
```

然后要实现 ISpecifyPropertyPages 接口的 GetPages 方法：

```
class CGrayFilter : public ISaturation,
```

```
                        public ISpecifyPropertyPages,
                        /* Other inherited classes. */
{
public:
    STDMETHODIMP GetPages(CAUUID *pPages)
    {
        if (pPages == NULL) return E_POINTER;
        pPages->cElems = 1;
        pPages->pElems = (GUID*)CoTaskMemAlloc(sizeof(GUID));
        if (pPages->pElems == NULL)
        {
            return E_OUTOFMEMORY;
        }
        pPages->pElems[0] = CLSID_SaturationProp;
        return S_OK;
    }
};

/* ... */

}
```

第三步，支持 QueryInterface

为了暴露 Filter 的接口，照着下面的步骤作哦

1 在你的 filter 中包含 **DECLARE_IUNKNOWN** 宏的声明：

```
Public:
    DECLARE_IUNKNOWN;
```

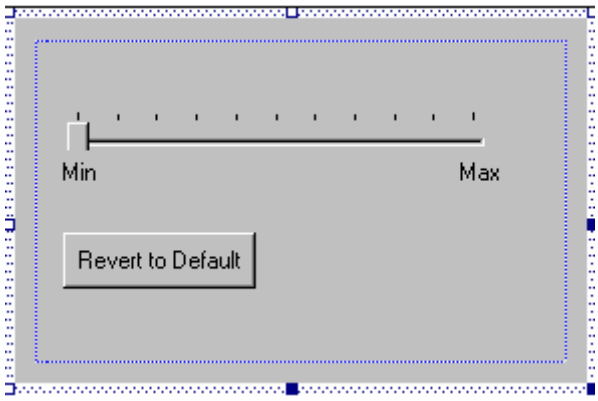
2 重载 **CUnknown::NonDelegatingQueryInterface** 方法来检查两个接口的 IIDs。

```
STDMETHODIMP CGrayFilter::NonDelegatingQueryInterface(REFIID riid,
    void **ppv)
{
    if (riid == IID_ISpecifyPropertyPages)
    {
        return GetInterface(static_cast<ISpecifyPropertyPages*>(this),
            ppv);
    }
    if (riid == IID_ISaturation)
    {
        return GetInterface(static_cast<IYuvGray*>(this), ppv);
    }
    return CBaseFilter::NonDelegatingQueryInterface(riid, ppv);
}
```

第四步，创建属性页

到这一步，filter 已经支持一个属性页的所需要的东西了，下一步就是要实现属性页本身了。

首先创建一个对话框的资源，然后以这个对话的资源声明一个类，要从 **CBasePropertyPage**. 派生，



下面的代码显示了部分的声明，包含了我们在后面将要用到的部分变量。

```
class CGrayProp : public CBasePropertyPage
{
private:
    ISaturation *m_pGray;       // Pointer to the filter's custom interface.
    long        m_lVal          // Store the old value, so we can revert.
    long        m_lNewVal;      // New value.
public:
    /* ... */
};
```

看看构造函数吧

```
CGrayProp::CGrayProp(IUnknown *pUnk) :
  CBasePropertyPage(NAME("GrayProp"), pUnk, IDD_PROPPAGE, IDS_PROPPAGE_TITLE),
  m_pGray(0)
{ }
```

下面，你还要记得重载 **CBasePropertyPage** 的几个方法哦

OnConnect，当属性页创建的时候，会调用这个方法，通过这个方法将 IUnknown 指针付给 Filter。

OnActivate 当对话框创建的时候被调用

OnReceiveMessage 当对话框接收到窗口消息时被调用

OnApplyChanges 当用户单击 OK 或者 Apply 按钮来确认对属性进行更新时，调用

OnDisconnect 当用户取消 Property sheet 时调用

第五步，保存 filter 的一个指针

通过重载 **CBasePropertyPage::OnConnect** 方法将一个指针保存到 filter，下面的例子演示了如何通过方法传递过来的参数查询 filter 支持的接口

```
HRESULT CGrayProp::OnConnect(IUnknown *pUnk)
{
    if (pUnk == NULL)
    {
        return E_POINTER;
    }
```

```
    ASSERT(m_pGray == NULL);
    return pUnk->QueryInterface(IID_ISaturation,
        reinterpret_cast<void**>(&m_pGray));
}
```

第六步，初始化对话框

通过重载 **CBasePropertyPage::OnActivate** 方法来初始化一个对话框，在这个例子里，属性页使用了滑动条，所以，在初始化的第一步就是要初始化控件动态库，然后再初始化 slider。

```
HRESULT CGrayProp::OnActivate(void)
{
    INITCOMMONCONTROLSEX icc;
    icc.dwSize = sizeof(INITCOMMONCONTROLSEX);
    icc.dwICC = ICC_BAR_CLASSES;
    if (InitCommonControlsEx(&icc) == FALSE)
    {
        return E_FAIL;
    }

    ASSERT(m_pGray != NULL);
    HRESULT hr = m_pGray->GetSaturation(&m_lVal);
    if (SUCCEEDED(hr))
    {
        SendDlgItemMessage(m_Dlg, IDC_SLIDER1, TBM_SETRANGE, 0,
            MAKELONG(SATURATION_MIN, SATURATION_MAX));

        SendDlgItemMessage(m_Dlg, IDC_SLIDER1, TBM_SETTICFREQ,
            (SATURATION_MAX - SATURATION_MIN) / 10, 0);

        SendDlgItemMessage(m_Dlg, IDC_SLIDER1, TBM_SETPOS, 1, m_lVal);
    }
    return hr;
}
```

第七步，处理窗口消息

重载 **CBasePropertyPage::OnReceiveMessage** 方法来处理用户的输入等消息。如果你不想处理消息，你只需简单调用父类的 **OnReceiveMessage** 即可。

无论何时用户改变了属性，都会做下面的事情

1 将属性页的 m_bDirty 设置为 TRUE；

2 调用属性框的 **IPropertyPageSite::OnStatusChange** 方法，并传递一个 PROPPAGESTATUS_DIRTY，这个标志用来通知 property frame 应该将 Apply 按钮可用，**CBasePropertyPage::m_pPageSite** 变量保存着一个 **IPropertyPageSite** 接口

为了简化步骤，你可以在你的属性页中添加下面的代码

```
private:
    void SetDirty()
    {
```

```
        m_bDirty = TRUE;
        if (m_pPageSite)
        {
            m_pPageSite->OnStatusChange(PROPPAGESTATUS_DIRTY);
        }
    }
```

当用户改变了属性的时候，在 OnReceiveMessage 方法中调用上面的函数。

```
BOOL CGrayProp::OnReceiveMessage(HWND hwnd,
    UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_COMMAND:
        if (LOWORD(wParam) == IDC_DEFAULT)
        {
            // User clicked the 'Revert to Default' button.
            m_lNewVal = SATURATION_DEFAULT;
            m_pGray->SetSaturation(m_lNewVal);

            // Update the slider control.
            SendDlgItemMessage(m_Dlg, IDC_SLIDER1, TBM_SETPOS, 1,
                m_lNewVal);
            SetDirty();
            return (LRESULT) 1;
        }
        break;

    case WM_HSCROLL:
        {
            // User moved the slider.
            switch(LOWORD(wParam))
            {
            case TB_PAGEDOWN:
            case SB_THUMBTRACK:
            case TB_PAGEUP:
                m_lNewVal = SendDlgItemMessage(m_Dlg, IDC_SLIDER1,
                    TBM_GETPOS, 0, 0);
                m_pGray->SetSaturation(m_lNewVal);
                SetDirty();
            }
            return (LRESULT) 1;
        }
    } // Switch.
```

```
    // Let the parent class handle the message.
    return CBasePropertyPage::OnReceiveMessage(hwnd,uMsg,wParam,lParam);
}
```

第八步，处理属性的改变

重载 **CBasePropertyPage::OnApplyChanges** 方法来提交属性页的改变，如果用户单击了确定，或者应用按钮，OnApplyChanges 方法都会调用到

```
HRESULT CGrayProp::OnApplyChanges(void)
{
    m_lVal = m_lNewVal;
    return S_OK;
}
```

第九步，断开属性页连接

重载 **CBasePropertyPage::OnDisconnect** 方法来释放你在 OnConnect 方法中请求的所有的接口，如果用户没有更新属性，而是单击了取消按钮，你还要将属性的原始值保存下来。当用户单击取消按钮，但是没有相应的响应这个消息的方法，所以，你要检查用户是否调用了 OnApplyChanges 方法，看看例子也好：

```
HRESULT CGrayProp::OnDisconnect(void)
{
    if (m_pGray)
    {
        // If the user clicked OK, m_lVal holds the new value.
        // Otherwise, if the user clicked Cancel, m_lVal is the old value.
        m_pGray->SetSaturation(m_lVal);
        m_pGray->Release();
        m_pGray = NULL;
    }
    return S_OK;
}
```

第十步，支持 com 的注册

最后一步就是要支持 com 的注册，因此 属性框才能够创建你属性页的实例，首先在全局数组 **g_Templates** 添加一个类厂模板的说明。这个全局的数组是你的 DLL 中创建的所有的 com 对象都要用到的。

```
const AMOVIESETUP_FILTER FilterSetupData =
{
    /* Not shown ... */
};

CFactoryTemplate g_Templates[] =
{
    // This entry is for the filter.
    {
```

```
        wszName,
        &CLSID_GrayFilter,
        CGrayFilter::CreateInstance,
        NULL,
        &FilterSetupData
    },
    // This entry is for the property page.
    {
        L"Saturation Props",
        &CLSID_SaturationProp,
        CGrayProp::CreateInstance,
        NULL, NULL
    }
};
```

如果你用下面的方式声明全局数组，数组的大小就会自动地得到修改

```
int g_cTemplates = sizeof(g_Templates)/sizeof(g_Templates[0]);
```

同时，还要在属性页类中添加一个 CreateInstance 方法

```
static CUnknown * WINAPI CreateInstance(LPUNKNOWN pUnk, HRESULT *pHr)
{
    CGrayProp *pNewObject = new CGrayProp(pUnk);
    if (pNewObject == NULL)
    {
        *pHr = E_OUTOFMEMORY;
    }
    return pNewObject;
}
```

如果想测试属性页，可以注册 DLL，然后将 filter 加载到 GraphEdit，鼠标右击来查看 filter 的属性。

11 capture and compression formats

这篇文档主要来讲述如何通过 **IAMStreamConfig::GetStreamCaps** 来返回捕捉和压缩的数据格式。这个方法可以获取 filter 支持的媒体类型的更多信息，这比通过枚举 pin 所支持的媒体类型得到的信息要多。**GetStreamCaps** 方法也可以返回所支持的音频和视频的格式种类，另外，这篇文档也提供了一些代码演示了如何连接提供了特殊格式输出的输入 pin。

GetStreamCaps 返回了一个媒体类型和性能的结构数组，媒体类型一般通过 **AM_MEDIA_TYPE** 结构表示，性能主要通过下面的两个结构来标示 **AUDIO_STREAM_CONFIG_CAPS** structure or a **VIDEO_STREAM_CONFIG_CAPS**，分别用来表示音频和视频。

看看 GetStreamCaps 函数的定义

```
HRESULT GetStreamCaps(
  int iIndex,
  AM_MEDIA_TYPE **pmt,
  BYTE *pSCC
```

);

pSCC

[out] Pointer to a byte array allocated by the caller.

For video, use the VIDEO_STREAM_CONFIG_CAPS structure.

For audio, use the AUDIO_STREAM_CONFIG_CAPS structure. To determine the required size of the array, call the GetNumberOfCapabilities method.

下面的代码演示了如果找到第一个支持视频的输出 pin，然后它配置输出 pin 采用最小尺寸的输出数据格式，因此它会修改数据块的 **BITMAPINFOHEADER** 的宽度和高度，并且将 **AM_MEDIA_TYPE** 结构中的 **lSampleSize** 成员设置成新的 sample 尺寸。

```
int iCount, iSize;
VIDEO_STREAM_CONFIG_CAPS scc;
AM_MEDIA_TYPE *pmt;

hr = pConfig->GetNumberOfCapabilities(&iCount, &iSize);
if (sizeof(scc) != iSize)
{
    // This is not the structure we were expecting.
    return E_FAIL;
}
// Get the first format.
hr = pConfig->GetStreamCaps(0, &pmt, reinterpret_cast<BYTE*>(&scc));
if (hr == S_OK)
{
    // Is it VIDEOINFOHEADER and UYVY?
    if (pmt->formattype == FORMAT_VideoInfo &&
        pmt->subtype == MEDIASUBTYPE_UYVY)
    {
        // Find the smallest output size.
        LONG width = scc.MinOutputSize.cx;
        LONG height = scc.MinOutputSize.cy;
        LONG cbPixel = 2;    // Bytes per pixel in UYVY

        // Modify the format block.
        VIDEOINFOHEADER *pVih =
            reinterpret_cast<VIDEOINFOHEADER*>(pmt->pbFormat);
        pVih->bmiHeader.biWidth = width;
        pVih->bmiHeader.biHeight = height;

        // Set the sample size and image size.
        // (Round the image width up to a DWORD boundary.)
        pmt->lSampleSize = pVih->bmiHeader.biSizeImage =
            ((width + 3) & ~3) * height * cbPixel;

        // Now set the format.
```

```
            hr = pConfig->SetFormat(pmt);
            if (FAILED(hr))
            {
                MessageBox(NULL, TEXT("SetFormat Failed\n"), NULL, MB_OK);
            }
            DeleteMediaType(pmt);
        }
}
```

这篇文档主要包括三个部分

Video Capabilities

Audio Capabilities

Reconnecting Your Input to Ensure Specific Output Types

1 Video Capabilities

IAMStreamConfig::GetStreamCaps 方法通过 **AM_MEDIA_TYPE** and **VIDEO_STREAM_CONFIG_CAPS** 结构来返回视频的属性，看看下面我们是如何设置视频属性的吧。

假如你的捕捉卡支持 JPEG 格式的图像，尺寸大小在 160*120 至 320*240，

这篇文档看的我迷迷糊糊，算了，暂时先放弃，等有机会再重新读这篇吧，我看这篇并不是很重要。

12Graph 如何定位 filter 的位置并加载

2.13 Encoder and Decoder 开发

3Directshow 的基类学习

3.1 Dshow 的基类简介

COM Object Classes48 个类

CBaseObject 和 CUnknown 和两个类支持创建 COM 组件 。大多数的 Direcshow 类都是从这 CBaseObject 类派生出来的。

Filter and Pin Classes

下面的创建 Filter 的基类

CBaseFilter

CBaseInputPin
CBasePin
CBaseOutputPin
下面的类可以作为创建特定 Filter 的基类
CSource
CTransformFilter
CTransInPlaceFilter
CVideoTransformFilter
CBaseRenderer
CBaseVideoRenderer
Helper Objects
常用的帮助类如下
CPullPin
COutputQueue
CSourceSeeking
CEnumPins
CEnumMediaTypes
CMemAllocator
CMediaSample
CBaseReferenceClock
CMediaType
今天终于找到 DirectShow 的接口的定义了，哈哈，看看 C:\DX90SDK\Include\DShowIDL 路径下就是全部 Dshow 基类接口的 IDL 定义。Dshow 是基于多继承的方法来实现 com 对象的。我举例，看看我们的老大 CBaseFilter 的定义，这个类实现了三个接口吧

```
class AM_NOVTABLE CBaseFilter : public CUnknown,    // Handles an IUnknown
                                public IBaseFilter,        // The Filter Interface
                                public IAMovieSetup        // For un/registration
{
```

我一开始还纳闷，这个 IBaseFilter 是从哪里定义的啊，找了半天也没有找到，还有这个 CLSID IID_IBaseFilter 找了半天都没有找到，原来都在 include 下面啊，IBaseFilter 的定义在 austreams.idl 中，下面我把定义拷贝过来瞧瞧吧

```
[
object,
uuid(56a86895-0ad4-11ce-b03a-0020af0ba770),
pointer_default(unique)
]
interface IBaseFilter : IMediaFilter {

    // enumerate all the pins available on this filter
    // allows enumeration of all pins only.
    //
    HRESULT EnumPins(
        [out] IEnumPins ** ppEnum     // enum interface returned here
    );
```

```
    // Convert the external identifier of a pin to an IPin *
    // This pin id is quite different from the pin Name in CreatePin.
    // In CreatePin the Name is invented by the caller.  In FindPin the Id
    // must have come from a previous call to IPin::QueryId.  Whether or not
    // this operation would cause a pin to be created depends on the filter
    // design, but if called twice with the same id it should certainly
    // return the same pin both times.
    HRESULT FindPin(
        [in, string] LPCWSTR Id,
        [out] IPin ** ppPin
    );

    // find out information about this filter
    typedef struct _FilterInfo {
    WCHAR achName[MAX_FILTER_NAME]; // maybe null if not part of graph
        IFilterGraph * pGraph;                  // null if not part of graph
    } FILTER_INFO;

    HRESULT QueryFilterInfo(
        [out] FILTER_INFO * pInfo
    );

    // notify a filter that it has joined a filter graph. It is permitted to
    // refuse. The filter should addref and store this interface for later use
    // since it may need to notify events to this interface. A null pointer indicates
    // that the filter is no longer part of a graph.
    HRESULT JoinFilterGraph(
        [in] IFilterGraph * pGraph,
        [in, string] LPCWSTR pName
    );

    // return a Vendor information string. Optional - may return E_NOTIMPL.
    // memory returned should be freed using CoTaskMemFree
    HRESULT QueryVendorInfo(
        [out, string] LPWSTR* pVendorInfo
    );
}
}
```

3.2 Filter 和 pin 的基类

3.2.1CBaseFilter

下面是我摘录的 CBaseFilter 的声明的源码，可以供参考用

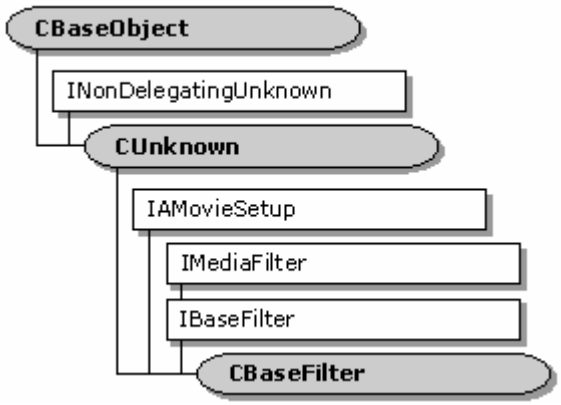
```
class AM_NOVTABLE CBaseFilter : public CUnknown,    // Handles an IUnknown
                                public IBaseFilter,         // The Filter Interface
                                public IAMovieSetup         // For un/registration
{
friend class CBasePin;

protected:
    FILTER_STATE        m_State;                // current state: running, paused
    IReferenceClock *m_pClock;              // this graph's ref clock
    CRefTime            m_tStart;               // offset from stream time to reference time
    CLSID           m_clsid;                // This filters clsid
                                            // used for serialization
    CCritSec            *m_pLock;               // Object we use for locking

    WCHAR               *m_pName;               // Full filter name
    IFilterGraph        *m_pGraph;              // Graph we belong to
    IMediaEventSink *m_pSink;               // Called with notify events
    LONG                m_PinVersion;           // Current pin version

public:

    CBaseFilter(
        const TCHAR *pName,         // Object description
        LPUNKNOWN pUnk,             // IUnknown of delegating object
        CCritSec    *pLock,         // Object who maintains lock
    REFCLSID    clsid);         // The clsid to be used to serialize this filter

    CBaseFilter(    TCHAR       *pName,         // Object description
        LPUNKNOWN pUnk,             // IUnknown of delegating object
        CCritSec    *pLock,         // Object who maintains lock
    REFCLSID    clsid,          // The clsid to be used to serialize this filter
        HRESULT     *phr);          // General OLE return code
#ifdef UNICODE
    CBaseFilter( const CHAR *pName,         // Object description
        LPUNKNOWN pUnk,             // IUnknown of delegating object
        CCritSec    *pLock,         // Object who maintains lock
    REFCLSID    clsid);         // The clsid to be used to serialize this filter
```

```
    CBaseFilter( CHAR         *pName,          // Object description
        LPUNKNOWN pUnk,              // IUnknown of delegating object
        CCritSec    *pLock,          // Object who maintains lock
    REFCLSID      clsid,          // The clsid to be used to serialize this filter
        HRESULT      *phr);           // General OLE return code
#endif
    ~CBaseFilter();

    DECLARE_IUNKNOWN

    // override this to say what interfaces we support where
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
#ifdef DEBUG
    STDMETHODIMP_(ULONG) NonDelegatingRelease();
#endif

    //
    // --- IPersist method ---
    //
    STDMETHODIMP GetClassID(CLSID *pClsID);

    // --- IMediaFilter methods ---

    STDMETHODIMP GetState(DWORD dwMSecs, FILTER_STATE *State);

    STDMETHODIMP SetSyncSource(IReferenceClock *pClock);

    STDMETHODIMP GetSyncSource(IReferenceClock **pClock);


    // override Stop and Pause so we can activate the pins.
    // Note that Run will call Pause first if activation needed.
    // Override these if you want to activate your filter rather than
    // your pins.
    STDMETHODIMP Stop();
    STDMETHODIMP Pause();

    // the start parameter is the difference to be added to the
    // sample's stream time to get the reference time for
    // its presentation
    STDMETHODIMP Run(REFERENCE_TIME tStart);

    // --- helper methods ---
```

```
    // return the current stream time - ie find out what
    // stream time should be appearing now
    virtual HRESULT StreamTime(CRefTime& rtStream);

    // Is the filter currently active?
    BOOL IsActive() {       CAutoLock cObjectLock(m_pLock);
        return ((m_State == State_Paused) || (m_State == State_Running));    };

    // Is this filter stopped (without locking)
    BOOL IsStopped() {       return (m_State == State_Stopped);    };

    //
    // --- IBaseFilter methods ---
    //
    // pin enumerator
    STDMETHODIMP EnumPins(IEnumPins ** ppEnum);

    // default behaviour of FindPin assumes pin ids are their names
    STDMETHODIMP FindPin(LPCWSTR Id, IPin ** ppPin      );

    STDMETHODIMP QueryFilterInfo( FILTER_INFO * pInfo);

    STDMETHODIMP  JoinFilterGraph(  IFilterGraph  *  pGraph,                LPCWSTR
pName);

    // return a Vendor information string. Optional - may return E_NOTIMPL.
    // memory returned should be freed using CoTaskMemFree
    // default implementation returns E_NOTIMPL
    STDMETHODIMP QueryVendorInfo( LPWSTR* pVendorInfo );

    // --- helper methods ---

    // send an event notification to the filter graph if we know about it.
    // returns S_OK if delivered, S_FALSE if the filter graph does not sink
    // events, or an error otherwise.
    HRESULT NotifyEvent(long EventCode, LONG_PTR EventParam1,
        LONG_PTR EventParam2);

    // return the filter graph we belong to
    IFilterGraph *GetFilterGraph() { return m_pGraph; }

    // Request reconnect
    // pPin is the pin to reconnect
    // pmt is the type to reconnect with - can be NULL
```

```
    // Calls ReconnectEx on the filter graph
    HRESULT ReconnectPin(IPin *pPin, AM_MEDIA_TYPE const *pmt);

    // find out the current pin version (used by enumerators)
    virtual LONG GetPinVersion();
    void IncrementPinVersion();

    // you need to supply these to access the pins from the enumerator
    // and for default Stop and Pause/Run activation.
    virtual int GetPinCount() PURE;
    virtual CBasePin *GetPin(int n) PURE;

    // --- IAMovieSetup methods ---

    STDMETHODIMP Register();       // ask filter to register itself
    STDMETHODIMP Unregister();     // and unregister itself

    // --- setup helper methods ---
    // (override to return filters setup data)

    virtual LPAMOVIESETUP_FILTER GetSetupData(){ return NULL; }

};
```

下面分析 CBaseFilter 的类的东西：



CBaseFilter 是从接口类 IBaseFilter，IMediaFilter，IAMovieSetup 多继承而来的，所以，它的 com 对象的实现形式就是多继承了。

首先看看它的数据成员

```
FILTER_STATE      m_State;   用来表示当前的状态，是运行还是暂停，停止
IReferenceClock *m_pClock;    Graph 图表的参考时钟
CRefTime          m_tStart;        // offset from stream time to reference time
CLSID          m_clsid;             Filter 的类 ID
CCritSec           *m_pLock;   //用于保护数据的锁，临界区对象
WCHAR                *m_pName;         Filter 的名称
IFilterGraph      *m_pGraph;             这个 Filter 所属的 Graph ，指向 Graph 的指针
```

IMediaEventSink *m_pSink; 指向 Graph 图上的 **IMediaEventSink** 指针
LONG m_PinVersion; 表明这个 Filter 中的 pin 的版本
下面是该类提供的函数的分析
1 CBaseFilter::StreamTime
virtual HRESULT StreamTime(CRefTime& *rtStream*);
这个函数用来返回当前的 Stream time，所谓当前的 Stream time 就是当前的参考时间（reference time）减去开始时间，这个开始时间就是 **CBaseFilter::m_tStart** 定义的。一个媒体 sample 上的时间戳用来标示它应该在那个 Stream time 提交，如果一个 sample 上的时间戳小于当前的 stream time，那么这就意味着它迟到了。
2 CBaseFilter::IsActive
BOOL IsActive(void);
这个函数用来查看 Filter 是否出于活动状态还是停止状态。
3 CBaseFilter::IsStopped
BOOL IsStopped(void);
这个函数用来看看 Filter 是否出于停止状态。
4 CBaseFilter::NotifyEvent
HRESULT NotifyEvent(long *EventCode*,LONG_PTR *EventParam1*,LONG_PTR *EventParam2*);
这个函数用来给图表管理器发送一个事件通知，一个参数用来指定时间特征码，两个消息参数，
5 CBaseFilter::GetFilterGraph
IFilterGraph *GetFilterGraph(void);
这个函数返回一个指向图表管理器的指针。
6 CBaseFilter::ReconnectPin
HRESULT ReconnectPin(IPin **pPin*, AM_MEDIA_TYPE const **pmt*);
这个函数首先将两个已经连接的 pin 断开，然后重新用指定的媒体类型将两个 pin 重新连接起来。
7 CBaseFilter::GetPinVersion
virtual long GetPinVersion(void);
这个函数其实直接将 **CBaseFilter::m_PinVersion** 成员变量返回，pin 的版本号？
8 CBaseFilter::IncrementPinVersion
这个函数直接将 pin 的版本号增加 1。
9 CBaseFilter::GetSetupData
virtual LPAMOVIESETUP_FILTER GetSetupData(void);
这个函数比较老了，用来返回 Filter 的注册信息。一般会在 **CBaseFilter::Register** and **CBaseFilter::Unregister** 方法中用到
10 CBaseFilter::GetPinCount
virtual int GetPinCount(void) PURE;
返回 pin 的数目。一个纯虚函数，需要你在自己的 filter 中重载这个这个函数。
11 CBaseFilter::GetPin
virtual CBasePin *GetPin(int *n*) PURE;
有一个纯虚函数，**CEnumPins** 类调用这个方法用来枚举 Filter 支持的 pin。根据指定的 pin 的序号返回 pin 的接口。
12 CBaseFilter::GetClassID

HRESULT GetClassID(CLSID *pClsID);
这个函数用来返回 Filter 类的 ID。这个函数里调用了 **IPersist::GetClassID 方法**我觉得，它其实就是将 CBaseFilter::m_clsid 变量直接返回了吧。
13 CBaseFilter::GetState
HRESULT GetState(DWORD *dwMilliSecsTimeout*, FILTER_STATE *State);
这个函数用来返回 Filter 的状态，必然运行，停止，暂停。这个函数里好像调用了 **IMediaFilter::GetState** 方法吧。
14 CBaseFilter::SetSyncSource
HRESULT SetSyncSource(IReferenceClock *pClock);
这个函数给 Filter 设置了一个参考时钟。
15 CBaseFilter::GetSyncSource
HRESULT GetSyncSource(IReferenceClock **pClock);
这个函数返回了 Filter 正在使用的参考时钟。
16 HRESULT Stop(void);
17 HRESULT Pause(void);
18 HRESULT Run(REFERENCE_TIME *tStart*);
上面的三个函数不用我解释了吧。看看就知道啥意思了
19 CBaseFilter::EnumPins
HRESULT EnumPins(IEnumPins **ppEnum);
这个函数用来枚举 Filter 所支持的 Pin。这个函数中生成了 CEnumPins 的一个实例，然后返回了一个 IEnumPins 的对象。然后由 CEnumPins 调用 **CBaseFilter::GetPin** 方法来实现枚举 pin。
20 CBaseFilter::FindPin
HRESULT FindPin(LPCWSTR *Id*, IPin **ppPin);
这个函数用来查找指定接口 ID 的 pin。
21 CBaseFilter::QueryFilterInfo
HRESULT QueryFilterInfo(FILTER_INFO *pInfo);
这个函数用来返回 Filter 的信息。
22 CBaseFilter::JoinFilterGraph
HRESULT JoinFilterGraph(IFilterGraph *pGraph*, LPCWSTR *pName*);
这个函数用来通知 Filter，这个 Filter 是添加进去 graph 中还是从 graph 中去掉了。
23 CBaseFilter::QueryVendorInfo
HRESULT QueryVendorInfo(LPWSTR *pVendorInfo);
这个函数用来返回

CBaseFilter 是一个抽象的基类，因为它有两个纯虚函数，如果你要是想从 CBaseFilter 实现你自己的一个 Filter，你必须按照下面的步骤去做：
1 首先你要从 CBaseFilter 中派生一个新的类。
2 你的 Filter 准备生出几个 pin，你就要定义几个 pin 的对象，注意，你的 pin 对象必须是从 CBasePin 中派生的类的对象。
3 实现纯虚函数 **CBaseFilter::GetPin**,，这个函数用来枚举 Filter 支持的 pin。
4 实现纯虚函数 **CBaseFilter::GetPinCount**，这个函数用来返回 Filter 所支持的 pin 的数目。
5 要提供能够产生，处理，或者提交媒体 Samples 的方法，当然这一条是对不同功能的 Filter 而言的，如果你的 filter 是源 filter，那么你就要产生 sample，如果是传输 filter，就要处理 sample，

如果是提交 Filter，就要提供能够提交 sample 的方法。

一些基类必然 **CSource**, **CBaseRenderer**, and **CTransformFilter** 也都是从 CBaseFilter 中继承过来的。其实，你直接从这些基类中派生一个 filter 比直接从 CBaseFilter 派生要容易得多。

3.2.2CBasePin

```
class    AM_NOVTABLE CBasePin : public CUnknown, public IPin,
  public IQualityControl
{
protected:

      WCHAR *              m_pName;                  // This pin's name
      IPin                 *m_Connected;                 // Pin we have connected to
      PIN_DIRECTION        m_dir;                        // Direction of this pin
      CCritSec             *m_pLock;                     // Object we use for locking
      bool                 m_bRunTimeError;              // Run time error generated
      bool                 m_bCanReconnectWhenActive;    // OK to reconnect when active
      bool                 m_bTryMyTypesFirst;           // When connecting enumerate
      CBaseFilter          *m_pFilter;                   // Filter we were created by
      IQualityControl *m_pQSink;                         // Target for Quality messages
      LONG                 m_TypeVersion;                // Holds current type version
      CMediaType           m_mt;                         // Media type of connection
      CRefTime             m_tStart;                     // time from NewSegment call
      CRefTime             m_tStop;                      // time from NewSegment
      double               m_dRate;                      // rate from NewSegment
#ifdef DEBUG
      LONG                 m_cRef;                       // Ref count tracing
#endif
      // displays pin connection information
#ifdef DEBUG
      void DisplayPinInfo(IPin *pReceivePin);
      void DisplayTypeInfo(IPin *pPin, const CMediaType *pmt);
#else
      void DisplayPinInfo(IPin *pReceivePin) {};
      void DisplayTypeInfo(IPin *pPin, const CMediaType *pmt) {};
#endif
      // used to agree a media type for a pin connection
      // given a specific media type, attempt a connection (includes
      // checking that the type is acceptable to this pin)
      HRESULT
      AttemptConnection(
            IPin* pReceivePin,          // connect to this pin
```

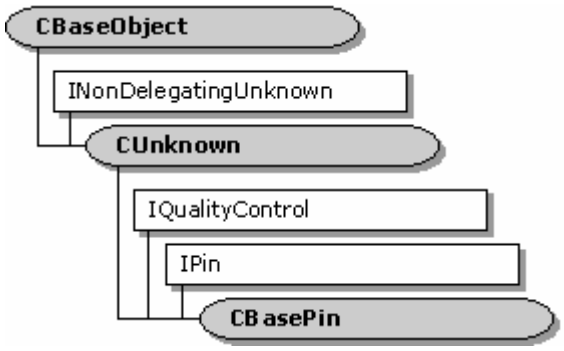
```
        const CMediaType* pmt      // using this type
    );

    // try all the media types in this enumerator - for each that
    // we accept, try to connect using ReceiveConnection.
    HRESULT TryMediaTypes( IPin *pReceivePin,          // connect to this pin
                           const CMediaType *pmt,    // proposed type from Connect
                           IEnumMediaTypes *pEnum);       // try this enumerator

    HRESULT AgreeMediaType( IPin *pReceivePin,          // connect to this pin
                        const CMediaType *pmt);    // proposed type from Connect

public:

    CBasePin( TCHAR *pObjectName,              // Object description
        CBaseFilter *pFilter,           // Owning filter who knows about pins
        CCritSec *pLock,                // Object who implements the lock
        HRESULT *phr,                   // General OLE return code
        LPCWSTR pName,                  // Pin name for us
        PIN_DIRECTION dir);             // Either PINDIR_INPUT or PINDIR_OUTPUT
#ifdef UNICODE
    CBasePin(
        CHAR *pObjectName,              // Object description
        CBaseFilter *pFilter,           // Owning filter who knows about pins
        CCritSec *pLock,                // Object who implements the lock
        HRESULT *phr,                   // General OLE return code
        LPCWSTR pName,                  // Pin name for us
        PIN_DIRECTION dir);             // Either PINDIR_INPUT or PINDIR_OUTPUT
#endif
    virtual ~CBasePin();

    DECLARE_IUNKNOWN

    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
    STDMETHODIMP_(ULONG) NonDelegatingRelease();
    STDMETHODIMP_(ULONG) NonDelegatingAddRef();

    STDMETHODIMP Connect(
        IPin * pReceivePin,
        const AM_MEDIA_TYPE *pmt      // optional media type
    );

    // (passive) accept a connection from another pin
    STDMETHODIMP ReceiveConnection(
```

```
        IPin * pConnector,          // this is the initiating connecting pin
        const AM_MEDIA_TYPE *pmt    // this is the media type we will exchange
    );

    STDMETHODIMP Disconnect();

    STDMETHODIMP ConnectedTo(IPin **pPin);

    STDMETHODIMP ConnectionMediaType(AM_MEDIA_TYPE *pmt);

    STDMETHODIMP QueryPinInfo( PIN_INFO * pInfo );

    STDMETHODIMP QueryDirection(
        PIN_DIRECTION * pPinDir
    );

    STDMETHODIMP QueryId(
        LPWSTR * Id
    );

    // does the pin support this media type
    STDMETHODIMP QueryAccept(
        const AM_MEDIA_TYPE *pmt
    );

    // return an enumerator for this pins preferred media types
    STDMETHODIMP EnumMediaTypes(
        IEnumMediaTypes **ppEnum
    );

    STDMETHODIMP QueryInternalConnections(
        IPin* *apPin,         // array of IPin*
        ULONG *nPin             // on input, the number of slots
                                // on output  the number of pins
    ) { return E_NOTIMPL; }

    // Called when no more data will be sent
    STDMETHODIMP EndOfStream(void);

    // Begin/EndFlush still PURE

    // NewSegment notifies of the start/stop/rate applying to the data
    // about to be received. Default implementation records data and
    // returns S_OK.
```

```
// Override this to pass downstream.
STDMETHODIMP NewSegment(
                    REFERENCE_TIME tStart,
                    REFERENCE_TIME tStop,
                    double dRate);

STDMETHODIMP Notify(IBaseFilter * pSender, Quality q);

STDMETHODIMP SetSink(IQualityControl * piqc);

// --- helper methods ---

// Returns true if the pin is connected. false otherwise.
BOOL IsConnected(void) {return (m_Connected != NULL); };
// Return the pin this is connected to (if any)
IPin * GetConnected() { return m_Connected; };

// Check if our filter is currently stopped
BOOL IsStopped() {
    return (m_pFilter->m_State == State_Stopped);
};

// find out the current type version (used by enumerators)
virtual LONG GetMediaTypeVersion();
void IncrementTypeVersion();

// switch the pin to active (paused or running) mode
// not an error to call this if already active
virtual HRESULT Active(void);

// switch the pin to inactive state - may already be inactive
virtual HRESULT Inactive(void);

// Notify of Run() from filter
virtual HRESULT Run(REFERENCE_TIME tStart);

// check if the pin can support this specific proposed type and format
virtual HRESULT CheckMediaType(const CMediaType *) PURE;

// set the connection to use this format (previously agreed)
virtual HRESULT SetMediaType(const CMediaType *);

// check that the connection is ok before verifying it
// can be overridden eg to check what interfaces will be supported.
```

```
    virtual HRESULT CheckConnect(IPin *);

    // Set and release resources required for a connection
    virtual HRESULT BreakConnect();
    virtual HRESULT CompleteConnect(IPin *pReceivePin);

    // returns the preferred formats for a pin
    virtual HRESULT GetMediaType(int iPosition,CMediaType *pMediaType);

    // access to NewSegment values
    REFERENCE_TIME CurrentStopTime() {
        return m_tStop;
    }
    REFERENCE_TIME CurrentStartTime() {    return m_tStart;    }
    double CurrentRate() { return m_dRate;    }

    //    Access name
    LPWSTR Name() { return m_pName; };

    //    Can reconnectwhen active?
    void SetReconnectWhenActive(bool bCanReconnect)
    {
        m_bCanReconnectWhenActive = bCanReconnect;
    }

    bool CanReconnectWhenActive()
    {
        return m_bCanReconnectWhenActive;
    }

protected:
    STDMETHODIMP DisconnectInternal();
};
```

下面我们分析 CBasePin 类



先分析一下 CBasePin 类中的数据成员

```
WCHAR *          m_pName;              //  pin的名字
IPin             *m_Connected;         //  和这个pin相连接的pin的指针
PIN_DIRECTION    m_dir;                // Direction of this pin
CCritSec         *m_pLock;             //用来保护数据的临界区指针
bool             m_bRunTimeError;      //用来标示是否发生一个运行时错误的标志
bool m_bCanReconnectWhenActive;        //用来标示这个pin是否支持动态的连接
bool             m_bTryMyTypesFirst;   // When connecting enumerate
CBaseFilter      *m_pFilter;           //  一直指向和本创建这个pin的Filter的指针
IQualityControl *m_pQSink;             //  用来指向一个控制质量消息的对象指针
LONG             m_TypeVersion; //媒体类型的version
CMediaType       m_mt;                 /当前pin连接使用的媒体类型
CRefTime         m_tStart;             // time from NewSegment call
CRefTime         m_tStop;              // time from NewSegment
double           m_dRate;              // rate from NewSegment
```

下面我们看看成员函数吧

1CBasePin::DisplayPinInfo

这个函数，用来在调试的时候显示 pin 连接的情况。

2CBasePin::DisplayTypeInfo

用来在调试的时候显示媒体类型的信息

3CBasePin::AttemptConnection

```
virtual HRESULT AttemptConnection( IPin *pReceivePin,const CMediaType *pmt);
```

这个函数用指定的媒体类型去连接另外一个 pin 如果指定的媒体类型不被接受，这个方法就是失败返回，不再尝试用其他的媒体类型进行连接。如果媒体类型合适，然后这个方法就要调用 **IPin::ReceiveConnection** 方法，然后调用 **CBasePin::CompleteConnect** 方法来完成连接。

4CBasePin::TryMediaTypes

```
virtual HRESULT TryMediaTypes( IPin *pReceivePin, const CMediaType *pmt
,                              IEnumMediaTypes *pEnum);
```

根据所提供的一个媒体类型的 list，然后 TryMediaType 方法试图用其中的一种媒体类型来完成连接。**IEnumMediaTypes** 接口用来枚举媒体类型的 list。对于 **IEnumMediaTypes** 接口返回的每一种媒体类型，这个方法试图通过 **CBasePin::AttemptConnection** 方法进行连接，如果 pmt 参数是非空的，pin 就会忽略那些不能够和这个媒体类型匹配的那些媒体类型，pmt 参数用来指定一个泛类型的媒体类型，一个泛类型的媒体类型会在 major type 或者 subtype 或者 format 都有一个 GUID_NULL 的值，GUID_NULL 可以跟任何类型的值匹配。

5CBasePin::AgreeMediaType

```
virtual HRESULT AgreeMediaType(IPin *pReceivePin, const CMediaType *pmt);
```

这个方法搜寻一种媒体类型用来进行连接。如果 pmt 参数为非空并且指定一个媒体类型，这个方法就试图用这个媒体类型进行连接，如果连接失败，返回一个 error。如果这个参数为 NULL 或者这个参数为一个泛类型的媒体类型，这个方法就进行下面的工作。

1 接收 pin 枚举媒体类型，

2 本 pin 枚举媒体类型。

枚举媒体类型采用 **CBasePin::EnumMediaTypes**，然后将枚举的媒体类型结构传递给 **CBasePin::TryMediaTypes** 看看是否可以建立连接。

6CBasePin::DisconnectInternal

断开 pin 连接

7CBasePin::IsConnected

该函数用来判断这个 pin 是否和其他的 pin 建立了连接。

8CBasePin::GetConnected

```
IPin *GetConnected(void);
```

这个函数用来返回和这个 pin 建立了连接的那个 pin。如果这个 pin 没有连接，返回 NULL 看看实例代码。

```
if (m_MyPin->IsConnected())
{
    m_MyPin->GetConnected()->EndOfStream();
}
```

9CBasePin::IsStopped

看看和这个 pin 连接的 filter 是否停止了。

10CBasePin::GetMediaTypeVersion

11CBasePin::IncrementTypeVersion

12CBasePin::Active

```
virtual HRESULT Active(void);
```

这个函数用来通知 pin，和它相连的 Filter 现在开始激活了。

当一个 Filter 从停止转向暂停时，CBaseFilter 调用和 Filter 相连的 Pin 的这个方法用来通知 pin，Filter 已经激活了。基类中的这个方法没有做任何事情，派生类可以重载这个函数，例如，一个 pin 可以在请求分配内存，或者获取硬件资源。

13CBasePin::Inactive

```
virtual HRESULT Inactive(void);
```

这个函数的用法类似上面的函数

当一个 Filter 停止的时候，它会调用所有和 Filter 相连的 pin 上的这个方法，在基类中这个函数没有做任何事情，派生类可以重载这个方法来释放资源。

14CBasePin::Run

```
HRESULT Run( REFERENCE_TIME tStart);
```

当一个 Filter 从暂停转到运行状态，Filter 就会调用所有和 Filter 相连的 pin 上的这个方法，这也是一个虚函数，派生类可以重载这个函数，例如，一个 pin 可能在这个函数里启动一个工作线程来处理 samples。

15CBasePin::SetMediaType

```
virtual HRESULT SetMediaType( const CMediaType *pmt);
```

这个方法用来设置 pin 连接时的媒体类型。在调用这个函数之前，pin 一般都要调用 **CBasePin::CheckMediaType** 方法判断是否接受指定的媒体类型，这个函数一般就是将媒体类型设置给 **CBasePin::m_mt** 变量。

16CBasePin::CheckConnect

```
virtual HRESULT CheckConnect( IPin *pPin);
```

这个函数用来判断两个 pin 之间的连接是否合适。这个方法在两个连接 pin 的两端都会被调用，在连接的发起 pin 端，会在 **CBasePin::Connect** 方法中调用，在接受连接 pin 的一端，在 **CBasePin::ReceiveConnection** 方法中调用。

通过这个方法可以检验 pPin 参数指定的 pin 是否适合连接。在基类的方法实现中，如果连接的两方具有相同的类型，比如都是输出 pin，或者都是输入 pin，那么这个方法就会返回 error，派生类可以在这个方法中判断其他的属性是否合适。

如果这个方法返回 error，那么连接失败，那么连接的两个 pin 都会调用 **CBasePin::BreakConnect** 方法来断开连接，释放资源。

17CBasePin::BreakConnect

virtual HRESULT BreakConnect(void);

当 pin 在断开连接的时候，比如调用 **CBasePin::Disconnect**，或者在试图连接的过程中调用 **CBasePin::CheckConnect** 方法失败后都会调用这个方法来释放资源。

18CBasePin::CompleteConnect

virtual HRESULT CompleteConnect(IPin *pReceivePin);

这个方法调用用来完成 pin 之间的连接。

连接的发起方在 **CBasePin::Connect** 方法总调用它，连接的接受方在 **CBasePin::ReceiveConnection** 函数中调用这个函数。

在基类的实现中，这个方法仅仅是返回了 S_OK，如果派生类对于完成连接还有其他的要求，可以派生这个类。

19CBasePin::GetMediaType

virtual HRESULT GetMediaType(int *iPosition*, CMediaType *pMediaType);

这个方法提供一个 index，然后根据这个序号返回媒体类型。一般来说 pin 支持的媒体类型都用一个 list 表示，这个方法其实就是返回 list 中指定位置的媒体类型。

20CBasePin::CurrentStopTime

Returns the value of **CBasePin::m_tStop**.

21CBasePin::CurrentStartTime

Returns the value of **CBasePin::m_tStart**.

22CBasePin::CurrentRate

Returns the value of **CBasePin::m_dRate**.

23CBasePin::Name

Returns the value of the **CBasePin::m_pName** member variable.

24CBasePin::SetReconnectWhenActive

void SetReconnectWhenActive(bool *bCanReconnect*);

缺省的情况下，你在连接一个 filter 的 pin 的时候，你必须将 filter 停止，如果在 filter 处于活动状态的时候允许 pin 连接，那么就可以调用这个方法，返回一个 true。

25CBasePin::CanReconnectWhenActive

bool CanReconnectWhenActive(void);

这个函数用来询问，这个 pin 是否可以动态连接。

26CBasePin::CheckMediaType

virtual HRESULT CheckMediaType(const CMediaType *pmt) PURE;

纯虚函数哦，这个函数用来询问 pin 是否接受指定的媒体类型。

下面的方法都是 IPin 接口的方法了，注意哦，连接专用方法都是

27CBasePin::Connect

HRESULT Connect(IPin *pReceivePin, const AM_MEDIA_TYPE *pmt);

这个函数用指定的媒体类型连接另外一个 pin。这个方法是 IPin 接口的方法。

Pmt 参数可以为 NULL，或者它的 major type, subtype, or format 为 GUID_NULL,

在基类的实现中，这个方法仅仅是判断了一下该 pin 是否正在连接其他的 pin，该 pin 属于的 filter 是否停止。

28CBasePin::ReceiveConnection

HRESULT ReceiveConnection(IPin *pConnector, AM_MEDIA_TYPE *pmt);

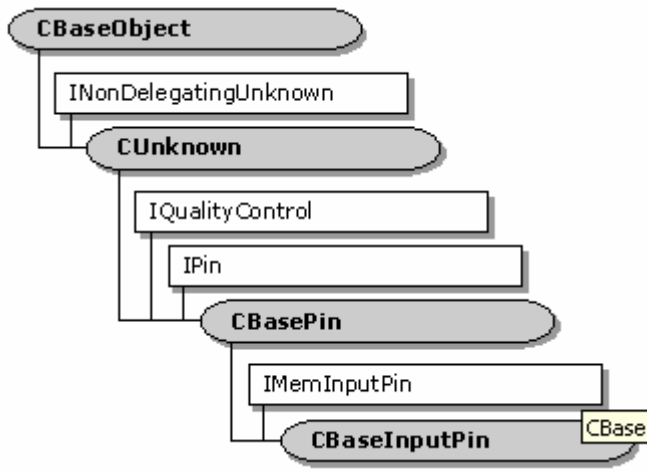
输出 pin 一般都是调用输入 pin 上的这个方法，如果输入 pin 的这个方法返回一个 error，那么连接失败。
在基类的实现中，这个函数主要做了一下的工作，
1 首先检查 pin 是否已经连接其他的 pin
2 如果没有连接，检查 Filter 是否停止，
3 调用 **CBasePin::CheckConnect** 检查连接是否合适
4 调用 **CBasePin::CheckMediaType** 检查媒体类型是否合适。
如果所有的上面的步骤都 ok，那么调用 **CBasePin::CompleteConnect** 和 **SetMediaType** 方法完成连接。这个方法要保存媒体类型和一个指向输出 pin 的指针。
如果 **CheckConnect** or **CheckMediaType** 失败，就调用 **CBasePin::BreakConnect** 断开连接。
29CBasePin::Disconnect
断开连接。
30CBasePin::ConnectedTo
HRESULT ConnectedTo(IPin **ppPin*);
这个函数得到一个指向连接 pin 的指针。
31CBasePin::ConnectionMediaType
HRESULT ConnectionMediaType(AM_MEDIA_TYPE **pmt*);
这个方法用来得到当前 pin 连接正在使用的媒体类型。
32CBasePin::QueryPinInfo
33CBasePin::QueryDirection
34CBasePin::QueryId
This method returns a copy of the **CBasePin::m_pName** member variable.
35CBasePin::QueryAccept
HRESULT QueryAccept(const AM_MEDIA_TYPE **pmt*);
这个方法用来询问是否支持某种媒体类型。
36CBasePin::EnumMediaTypes
HRESULT EnumMediaTypes(IEnumMediaTypes ***ppEnum*);
这个方法用来枚举 pin 所支持的媒体类型。
37CBasePin::QueryInternalConnections
HRESULT QueryInternalConnections(IPin **apPin*,ULONG **nPin*);
这个函数用来得到和这个 pin 内连接的 pin。
38CBasePin::EndOfStream
Filter 应该能够向下游的 filter 发送 endstream 的消息通知，其实就是通过输入 pin 的这个方法来完成的。
39CBasePin::Notify
HRESULT Notify(IBaseFilter **pSelf*, Quality *q*);
这个函数用来通知 pin，一个质量变化的消息已经产生了。
40CBasePin::SetSink
HRESULT SetSink(IQualityControl **piqc*);
用来设置一个外部的控制质量的接口。其实就是添加一个哦。
这个类就到此吧。太累了，下面看看输入 pin 的基类吧。

3.2.3CBaseInputPin

```
class AM_NOVTABLE CBaseInputPin : public CBasePin,
public IMemInputPin
{

protected:

    IMemAllocator *m_pAllocator;       // Default memory allocator
    BYTE m_bReadOnly;
    BYTE m_bFlushing;
    AM_SAMPLE2_PROPERTIES m_SampleProps;

public:

    CBaseInputPin(      TCHAR *pObjectName, CBaseFilter *pFilter,      CCritSec *pLock,
HRESULT *phr,     LPCWSTR pName);
#ifdef UNICODE
    CBaseInputPin(      CHAR *pObjectName, CBaseFilter *pFilter, CCritSec *pLock,
HRESULT *phr,   LPCWSTR pName);
#endif
    virtual ~CBaseInputPin();

    DECLARE_IUNKNOWN

    // override this to publicise our interfaces
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void **ppv);

    // return the allocator interface that this input pin
    // would like the output pin to use
    STDMETHODIMP GetAllocator(IMemAllocator ** ppAllocator);

    // tell the input pin which allocator the output pin is actually
    // going to use.
    STDMETHODIMP NotifyAllocator( IMemAllocator * pAllocator,
                        BOOL bReadOnly);

    // do something with this media sample
    STDMETHODIMP Receive(IMediaSample *pSample);

    // do something with these media samples
    STDMETHODIMP ReceiveMultiple (    IMediaSample **pSamples,
        long nSamples, long *nSamplesProcessed);
```

```
// See if Receive() blocks
STDMETHODIMP ReceiveCanBlock();
STDMETHODIMP BeginFlush(void);
// receives
STDMETHODIMP EndFlush(void);
// allocator
STDMETHODIMP GetAllocatorRequirements(ALLOCATOR_PROPERTIES*pProps);

// Release the pin's allocator.
HRESULT BreakConnect();

// helper method to check the read-only flag
BOOL IsReadOnly() {
    return m_bReadOnly;
};

// helper method to see if we are flushing
BOOL IsFlushing() {
    return m_bFlushing;
};

//    Override this for checking whether it's OK to process samples
//    Also call this from EndOfStream.
virtual HRESULT CheckStreaming();

// Pass a Quality notification on to the appropriate sink
HRESULT PassNotify(Quality& q);

// IQualityControl methods (from CBasePin)

STDMETHODIMP Notify(IBaseFilter * pSender, Quality q);

// no need to override:
// STDMETHODIMP SetSink(IQualityControl * piqc);


// switch the pin to inactive state - may already be inactive
virtual HRESULT Inactive(void);

// Return sample properties pointer
AM_SAMPLE2_PROPERTIES * SampleProps() {
    ASSERT(m_SampleProps.cbData != 0);
    return &m_SampleProps;
}
```

};



这个是输入pin的基类，我们看看这个类到底做什么用的。先看数据成员：

IMemAllocator *m_pAllocator; //缺省的内存分配器指针，指向一个内存分配器

BYTE m_bReadOnly; //用来标志内存分配器是否分配的samples为只读属性

BYTE m_bFlushing; // 用来标示pin是否正在flushing

AM_SAMPLE2_PROPERTIES m_SampleProps;//最近分配的samples的属性。

下面分析输入 pin 的方法：

1CBaseInputPin::BreakConnect

HRESULT BreakConnect(void);

这个方法其实是重载了 **CBasePin::BreakConnect** 方法，在这个方法里，要将分配的内存返回给内存池，然后将指向内存分配器的指针释放掉。

当你派生了这个方法，一定要在你派生的方法中调用基类的方法，要不可能会发生内存的泄漏。

2CBaseInputPin::IsReadOnly

判断分配的内存块，就是 samples 是否为只读的。

3CBaseInputPin::IsFlushing

查询一下 pin 是否正在 flushing

4CBaseInputPin::CheckStreaming

virtual HRESULT CheckStreaming(void);

查询一下 pin 是否可以接受 samples

派生类的方法可以检查更多的条件，**CBaseInputPin::Receive** 方法里调用了这个函数，你应该重载的函数 **CBasePin::EndOfStream** 也调用了这个方法。

5CBaseInputPin::PassNotify

HRESULT PassNotify(Quality *q*);

如果 filter 没有处理质量控制消息就可以调用这个函数，这个方法传递消息给…0.

6CBaseInputPin::Inactive

HRESULT Inactive(void);

这个消息通知 pin，filter 处于不活动状态。

7CBaseInputPin::SampleProps

用来返回 samples 的属性，其实就是将 **CBaseInputPin::m_SampleProps** 返回。

8CBaseInputPin::BeginFlush

HRESULT BeginFlush(void);
这个函数开始了 flush 的操作，这个方法其实是实现了 **IPin::BeginFlush** 接口方法。
这个方法将 **CBaseInputPin::m_bFlushing** 标志设置为 TRUE，这个标志的改变将使 **CBaseInputPin::Receive** 方法拒绝接受任何的上游 pin 传递过来的 samples。
派生类必须要实现这个方法，并要做下面一些事情
1 调用下游输入 pin 上的 **IPin::BeginFlush** 方法，如果 pin 还没有向下游传递 samples，那么这一步可以跳过，如果你的输出 pin 是从 CBasOutputPin 派生的，你可以调用 CBaseOutputPin::DeliverBeginFlush 方法。
2 调用基类的方法
3 开始拒绝接受数据
4 如果 recive 方法正处于阻塞，立即返回。
9CBaseInputPin::EndFlush
HRESULT EndFlush(void);
这个方法将 **CBaseInputPin::m_bFlushing** 标志设置为 FASLE，因此，**CBaseInputPin::Receive** 方法可以开始接受数据。
派生类必须要实现这个方法，并作下面的事情
1 释放数据，等待 queued samples 被丢弃。
2 清除任何的 EC_COMPLETE 通知。
3 调用基类的方法
4 调用下游输入 pin 上的 **IPin::EndFlush**（应该是对输出 pin 而言的吧），如果 pin 还没有向下游的 fiter 传递 samples 那么这一步可以忽略。
10CBaseInputPin::GetAllocator
HRESULT GetAllocator(IMemAllocator **ppAllocator);
这个方法创建了一个 **CMemAllocator** 对象，如果你的 Filter 上的某个 pin 需要一个 allocator 内存分配器，你可以重载这个方法。
如果成功，IMemAllocator 指针就增加一个引用计数，记得 release 哦
11CBaseInputPin::NotifyAllocator
HRESULT NotifyAllocator(IMemAllocator *pAllocator,BOOL bReadOnly);
这个方法其实是 **IMemInputPin::NotifyAllocator** 接口的方法，用来指定一个 allocator 加入到连接中。
在 pin 的连接过程中，输出 pin 选择了一个 allocator，并通过这个方法来通知输入 pin，输出 pin 也可以使用输入 pin 在 **IMemInputPin::GetAllocator** 方法中提供的 allocator。
12CBaseInputPin::GetAllocatorRequirements
HRESULT GetAllocatorRequirements(ALLOCATOR_PROPERTIES *pProps);
这个方法用来返回输入 pin 上的 allocator 属性。
当一个输出 pin 初始化一个内存的 allocator，它可以调用这个方法来决定是否具有对内存有其他的要求。
13CBaseInputPin::Receive
HRESULT Receive(IMediaSample *pSample);
Receive 方法用来接收数据流中下一个媒体 samples，这个方法实现了 **IMemInputPin::Receive** 接口方法。
上游的输出 pin 通过调用这个方法来给下游的输入 pin 传递一个 sample。输入 pin 可以在这个函数中做下面的事情。
1 在返回之前处理 sample

2 返回，然后在一个工作线程中处理 sample
3 拒绝 sample
如果输入 pin 使用一个工作线程来处理 samples，就要在这个方法中为 samples 增加一个引用计数，当这个方法返回时，上游的 filter 释放这个 sample。当 sample 的接口计数为零时，sample 就返回到内存池中等待新的任务。
这个方法是同步的可以阻塞，如果方法阻塞，那么 pin 上的 **CBaseInputPin::ReceiveCanBlock** 就要返回一个 sok。
在基类的实现中，这个函数主要做了下面的事情
1 调用 **CBaseInputPin::CheckStreaming** 方法查询 pin 是否可以处理 sample，如果不具备能力，比如停止状态，那么就要返回 fails。
2 检查 samples 的属性，察看数据的格式是否发生了改变
3 如果格式变化了，这个方法就要调用 **CBasePin::CheckMediaType** 察看新的媒体格式是否支持。
4 如果新的格式不被支持，这个方法调用 **CBasePin::EndOfStream** 函数，然后发送一个 EC_ERRORABORT 事件，返回一个错误码。
5 假定我们没有错误，这个方法返回一个 sok。

14CBaseInputPin::ReceiveMultiple
```
HRESULT ReceiveMultiple( IMediaSample **pSamples,     long nSamples,
long *nSamplesProcessed);
```
这个方法用来接收一个 sample 的集合。这个方法是实现了 **IMemInputPin::ReceiveMultiple** 接口方法。
这个方法的行为类似于 **CBaseInputPin::Receive** 方法，但是接收的是一个 samples 集合。
15CBaseInputPin::ReceiveCanBlock
```
HRESULT ReceiveCanBlock(void);
```
这个方法决定了调用 **IMemInputPin::Receive** 方法是否阻塞。
如果调用 Receive 返回 S_FALSE 就保证不能阻塞，否则，返回 sok，如果 Receive 方法调用下游的 pin 上的 receive 方法，下游的 pin 可能阻塞。
16CBaseInputPin::Notify
```
HRESULT Notify(  IBaseFilter *pSelf,     Quality q);
```
Filter 经常会给上游的输出 pin 传递一个质量控制消息，**IQualityControl::Notify**

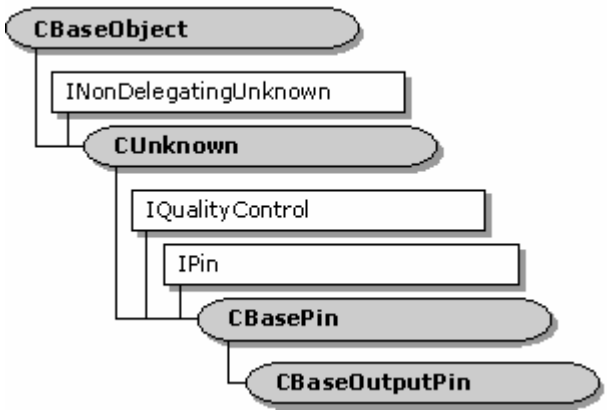
为了使用这个类，可以派生一个新类，并且至少要重载下面的几个函数
CBaseInputPin::BeginFlush
CBaseInputPin::EndFlush
CBaseInputPin::Receive
CBasePin::CheckMediaType
CBasePin::GetMediaType

3.2.4CBaseOutputPin

```
class   AM_NOVTABLE CBaseOutputPin : public CBasePin
{
```

```
protected:

    IMemAllocator *m_pAllocator;
    IMemInputPin *m_pInputPin;          // interface on the downstreaminput pin
public:

    CBaseOutputPin(     TCHAR *pObjectName,CBaseFilter *pFilter,
        CCritSec *pLock, HRESULT *phr, LPCWSTR pName);
#ifdef UNICODE
    CBaseOutputPin(CHAR *pObjectName,   CBaseFilter *pFilter,
        CCritSec *pLock,   HRESULT *phr, LPCWSTR pName);
#endif
    // override CompleteConnect() so we can negotiate an allocator
    virtual HRESULT CompleteConnect(IPin *pReceivePin);

    virtual HRESULT DecideAllocator(IMemInputPin * pPin, IMemAllocator **
pAlloc);

    // override this to set the buffer size and count. Return an error

    virtual HRESULT DecideBufferSize(   IMemAllocator * pAlloc,
        ALLOCATOR_PROPERTIES * ppropInputRequest
    ) PURE;

    // returns an empty sample buffer from the allocator
    virtual HRESULT GetDeliveryBuffer(IMediaSample ** ppSample,
REFERENCE_TIME * pStartTime,   REFERENCE_TIME * pEndTime,
            DWORD dwFlags);

    virtual HRESULT Deliver(IMediaSample *);

    // override this to control the connection
    virtual HRESULT InitAllocator(IMemAllocator **ppAlloc);
    HRESULT CheckConnect(IPin *pPin);
    HRESULT BreakConnect();

    // override to call Commit and Decommit
    HRESULT Active(void);
    HRESULT Inactive(void);

    // an error, since this should be called on input pins only
    STDMETHODIMP EndOfStream(void);
    // our connected input pin
```

```
    virtual HRESULT DeliverEndOfStream(void);

    STDMETHODIMP BeginFlush(void);
    STDMETHODIMP EndFlush(void);
    virtual HRESULT DeliverBeginFlush(void);
    virtual HRESULT DeliverEndFlush(void);

    virtual HRESULT DeliverNewSegment(REFERENCE_TIME tStart,
                    REFERENCE_TIME tStop,  double dRate);

};
```



这个 CBaseOutputPin 类是一个抽象的基类，实现了一个输出 pin。看看输入 pin 的基类，有两个接口，一个 IMemInputPin 接口和一个 IPin 接口，但是输出 pin 只有一个接口。直接从 CBasePin 派生出来。

1 输出 pin 只和支持 **IMemInputPin** 的输入 pin 连接。

2 通过 **IMemAllocator** 接口支持本地内存传输数据。

3 拒绝 end-of-stream, flush, and new-segment 通知。

4 提供向下游传递 samples 的方法。

当一个 pin 连接的时候，这个输出 pin 会从输入 pin 请求一个内存分配器，如果请求失败，它就创建一个新的 allocator 分配器。输出 pin 应该设置 allocator 的属性，它通过一个虚函数来实现，比如，**CBaseOutputPin::DecideBufferSize**.。在你的派生类中一定要实现这个方法，如果输入 pin 对内存有其他的要求，他们都会传递给 **CBaseOutputPin::DecideBufferSize**.方法。通过调用 **CBaseOutputPin::GetDeliveryBuffer** 来获取一个空的媒体 sample，通过 **CBaseOutputPin::Deliver** 方法将数据传递下去。

在你的派生类中，一定要实现纯虚函数 **CBasePin::CheckMediaType**，这个函数用来在连接 pin 的过程中确定连接的媒体类型。

下面我们来分析一下，数据成员

```
IMemAllocator *m_pAllocator;   一个指向内存分配器的指针
IMemInputPin *m_pInputPin;     用来记录相连接的输入 pin
```

1 CBaseOutputPin::DecideAllocator

```
virtual HRESULT DecideAllocator(IMemInputPin *pPin, IMemAllocator
*pAlloc);
```

这个函数用来选择一个内存分配器。

这个函数一般在 pin 连接的过程中被调用，主要做了下面的工作。

1 调用 **IMemInputPin::GetAllocatorRequirements** 方法查询输入 pin 上对内存的要求。
2 调用 **IMemInputPin::GetAllocator** 来从输入 pin 上请求一个 allocator，如果输入 pin 不能够提供一个 allocator，那么输出 pin 就调用 **CBaseOutputPin::InitAllocator** 创建一个 allocator。
3 通过调用 **CBaseOutputPin::DecideBufferSize** 方法，这个方法用来设置一个 allocator 的属性，这是一个纯虚的方法，派生类必须实现这个方法
4 调用 **IMemInputPin::NotifyAllocator** 方法，用来通知输入 pin 上 allocator 开始被使用了。

2CBaseOutputPin::GetDeliveryBuffer

```
virtual HRESULT GetDeliveryBuffer( IMediaSample **ppSample,
    REFERENCE_TIME *pStartTime, REFERENCE_TIME *pEndTime, DWORD
dwFlags);
```

这个方法用来返回一个包含空 buffer 的媒体 samples。
在这个方法中通过调用 **IMemAllocator::GetBuffer** 方法，并将参数传递给这个方法。

3CBaseOutputPin::Deliver

```
virtual HRESULT Deliver( IMediaSample *pSample);
```

这个方法可以将 sample 传递给相连接的 pin。
这个方法通过调用输入 pin 上的 **IMemInputPin::Receive** 方法，如果 **IMemInputPin::ReceiveCanBlock** 返回 sok，Receive 方法就会阻塞。
在调用这个方法后一定要 Release sample。输入 pin 可能还保持着 sample 的一个引用计数，所以，不能再使用这个 sample 了，可以通过 **CBaseOutputPin::GetDeliveryBuffer** 创建一个新的 samples。
在调用这个方法的过程中，一定要加上锁，否则的话在方法调用中可能断开连接。如果 Filter 具有一个工作线程用来发送 samples，那么在发送之前一定要设定锁，否则的话，你在 **IMemInputPin::Receive** 方法处理数据时也设定了锁，这样，工作线程有可能造成死锁。当一个工作线程有一个锁的时候，也许它要等待 Filter 的某个变量发生改变。为了防止死锁，状态改变的代码要激发一个事件结束线程。

4CBaseOutputPin::InitAllocator

```
virtual HRESULT InitAllocator( IMemAllocator **ppAlloc);
```

这个函数用来创建一个新的 allocator。

5CBaseOutputPin::CheckConnect

```
HRESULT CheckConnect( IPin *pPin);
```

这个函数用来决定一个连接的 pin 是否合适。

6CBaseOutputPin::BreakConnect

```
HRESULT BreakConnect(void);
```

这个方法重载了 **CBasePin::BreakConnect** 方法，这个方法用来释放内存分配器。

7CBaseOutputPin::Active

用来通知 pin，与之相连的 filter 处于激活状态

8CBaseOutputPin::Inactive

用来通知 pin，与之相连的 Filter 不活动了。

9CBaseOutputPin::DeliverEndOfStream

```
virtual HRESULT DeliverEndOfStream(void);
```

这个函数用来给与之相连的输入 pin 发送一个 end-of-stream 的通知，其实在这个方法中调用了输入 pin 上的 **IPin::EndOfStream**。

10CBaseOutputPin::DeliverBeginFlush

```
virtual HRESULT DeliverBeginFlush(void);
```

这个函数用要求与之相连的输入 pin 开始 Flush 动作。
内部调用了输入 pin 的 **IPin::BeginFlush** 方法。
11CBaseOutputPin::DeliverEndFlush
要求与之相连的输入 pin 停止 flush，方法的内部调用了输入 pin 的 **IPin::EndFlush** 方法。
12CBaseOutputPin::DeliverNewSegment

```
virtual HRESULT DeliverNewSegment(
    REFERENCE_TIME tStart,    REFERENCE_TIME tStop, double dRate );
```

这个函数用来给相连的输入 pin 发送一个 new-segment 通知。
13CBaseOutputPin::DecideBufferSize

```
virtual HRESULT DecideBufferSize(    IMemAllocator *pAlloc,
    ALLOCATOR_PROPERTIES *ppropInputRequest) PURE;
```

用来设置对 buffer 的要求。你应该在你的派生类中实现这个方法，调用 **IMemAllocator::SetProperties** 来设置你的 buffer 要求哦
14CBaseOutputPin::BeginFlush
开始一个 flush 动作。
15CBaseOutputPin::EndFlush
结束一个 flush 动作。
16CBaseOutputPin::EndOfStream

```
HRESULT EndOfStream(void);
```

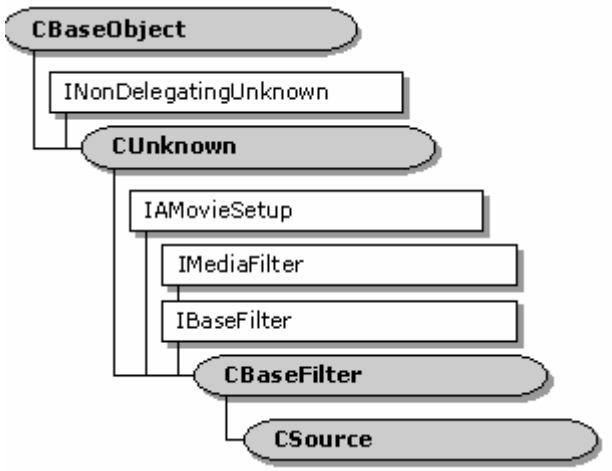
这个函数实现了 **IPin::EndOfStream** 接口函数。

3.3 几种常用 Filter 的基类

3.3.1CSource

```
class CSource : public CBaseFilter {
public:
    CSource(TCHAR *pName, LPUNKNOWN lpunk, CLSID clsid, HRESULT
*phr);
    CSource(TCHAR *pName, LPUNKNOWN lpunk, CLSID clsid);
#ifdef UNICODE
    CSource(CHAR *pName, LPUNKNOWN lpunk, CLSID clsid, HRESULT *phr);
    CSource(CHAR *pName, LPUNKNOWN lpunk, CLSID clsid);
#endif
    ~CSource();

    int          GetPinCount(void);
    CBasePin *GetPin(int n);

    CCritSec*   pStateLock(void) { return &m_cStateLock; }
    HRESULT        AddPin(CSourceStream *);
```

```
    HRESULT        RemovePin(CSourceStream *);

    STDMETHODIMP FindPin(LPCWSTR Id,IPin ** ppPin );

    int FindPinNumber(IPin *iPin);

protected:

    int                    m_iPins;
    CSourceStream **m_paStreams;     // the pins on this filter.
    CCritSec m_cStateLock; // Lock this to serialize
};
```



CSource 类是开发源 Filter 的基类，一个从从 CSource 类派生的 Filter 一般都支持一个或者几个从 CSourceSteam 类派生的输出 pin。每一个输出 pin 都创建了一个工作线程用来将数据传递给下游的 Filter。

为了生产一个输出 pin，请按照如下的步骤

1 从 CSoureceStream 类派生一个类。

2 重载 **CSourceStream::GetMediaType** 和 **CSourceStream::CheckMediaType** 方法。

3 重载 **CBaseOutputPin::DecideBufferSize**，这个方法将返回 pin 的 buffer 要求

4 重载 **CSourceStream::FillBuffer**，这个方法用来产生数据流，将数据填充到 buffer 中

为了派生一个源 Filter，可以按照下面的步骤来做

1 从 CSourec 类派生一个类。

2 在构造函数中创建一个或者多个的输出 pin。注意，这个 pin 要从 CSourceStream 派生哦。这些 pin 会在他们的构造函数中自动将自己添加到 Filter 中。并且在他们的析构函数种从 Filter 中删除 Pin。

为了在多线程中同步 Filter 的状态，调用 **CSource::pStateLock** 方法，这个方法返回一个指向临界区的指针，通过 **CAutoLock** 来生成一个锁，在 pin 中，你可以通过 **CBasePin::m_pFilter** 来指针来操纵临界区保护你的 Filter，从而达到同步。例如

```
CAutoLock lock(m_pFilter->pStateLock());
```

注意：CSource 用来生成推模式的源 Filter，如果要读文件，则应该使用拉模式。

下面我们分析一下 CSourec 的数据成员

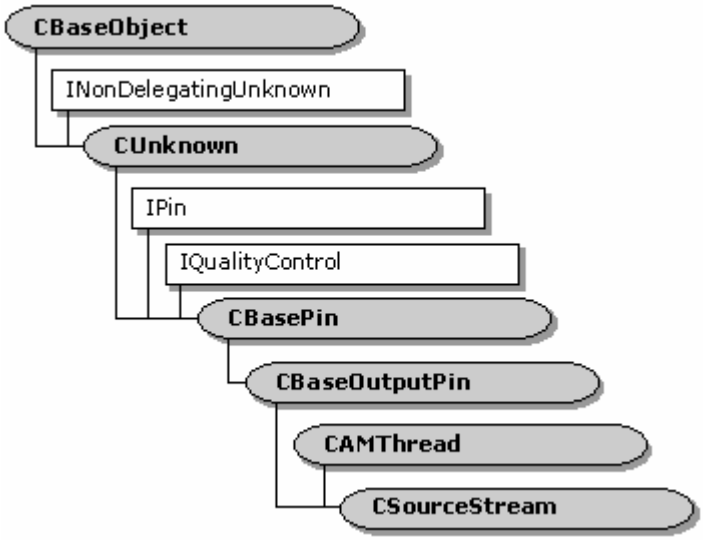
```
int m_iPins;
```

CSourceStream **m_paStreams;
CCritSec m_cStateLock;
总共三个数据成员，一个表示 pin 的数量，一个 pin 的数组，用来存放 Filter 支持的所有的 pin，另外一个是临界区对象。
下面分析方法，也比较简单。
1CSource::GetPinCount
这个函数用来返回 Filter 支持的 pin 的数目。
2CSource::GetPin
CBasePin *GetPin(int *n*);
根据指定的序号返回 pin 的指针。0，1，2，就是从 Filter 的那个 pin 数组中返回就是了，根据序号。
3CCritSec* pStateLock(void);
返回临界区对象
4CSource::AddPin
HRESULT AddPin(CSourceStream **pStream*);
构造函数通常调用这个函数将一个输出 pin 添加到 Filter 中。
5CSource::RemovePin
HRESULT RemovePin(CSourceStream **pStream*);
析构函数通常通过这个函数将一个输出 pin 从 Filter 中删除
6CSource::FindPinNumber
int FindPinNumber(IPin **iPin*);
这个函数根据指定的 pin 的指针，返回他的序号，如果返回-1 表示没有这个 pin。
7CSource::FindPin
HRESULT FindPin(LPCWSTR *Id*, IPin ***ppPin*);
根据指定的 ID 返回 pin。
注意，Filter 的第一个 pin ID 为 1 开始，然后是 23456789 等。

3.3.2CSourceStream

```
class CSourceStream : public CAMThread, public CBaseOutputPin {
public:

    CSourceStream(TCHAR *pObjectName,
                  HRESULT *phr,
                  CSource *pms,
                  LPCWSTR pName);
#ifdef UNICODE
    CSourceStream(CHAR *pObjectName,
                  HRESULT *phr,
                  CSource *pms,
                  LPCWSTR pName);
#endif
    virtual ~CSourceStream(void);    // virtual destructor ensures derived class
destructors are called too.
```

```
protected:

    CSource *m_pFilter;// The parent of this stream

    virtual HRESULT FillBuffer(IMediaSample *pSamp) PURE;

    virtual HRESULT OnThreadCreate(void) {return NOERROR;};
    virtual HRESULT OnThreadDestroy(void) {return NOERROR;};
    virtual HRESULT OnThreadStartPlay(void) {return NOERROR;};

    HRESULT Active(void);       // Starts up the worker thread
    HRESULT Inactive(void);    // Exits the worker thread.

public:
    // thread commands
    enum Command {CMD_INIT, CMD_PAUSE, CMD_RUN, CMD_STOP,
CMD_EXIT};
    HRESULT Init(void) { return CallWorker(CMD_INIT); }
    HRESULT Exit(void) { return CallWorker(CMD_EXIT); }
    HRESULT Run(void) { return CallWorker(CMD_RUN); }
    HRESULT Pause(void) { return CallWorker(CMD_PAUSE); }
    HRESULT Stop(void) { return CallWorker(CMD_STOP); }

protected:
    Command GetRequest(void) { return (Command) CAMThread::GetRequest(); }
    BOOL      CheckRequest(Command *pCom) { return
CAMThread::CheckRequest( (DWORD *) pCom); }

    // override these if you want to add thread commands
    virtual DWORD ThreadProc(void);          // the thread function

    virtual HRESULT DoBufferProcessingLoop(void);
    virtual HRESULT CheckMediaType(const CMediaType *pMediaType);
    virtual HRESULT GetMediaType(int iPosition, CMediaType *pMediaType);
    virtual HRESULT GetMediaType(CMediaType *pMediaType) {return
E_UNEXPECTED;}

    STDMETHODIMP QueryId(
        LPWSTR * Id
    );
};
```



CSourceStream 类用来生成一个输出 pin，当然为了 CSource 配合。
关于如何使用这个类，可以参看 CSource，这个类从 CAMThread 类继承而来，这个类提供了产生了数据流的工作线程，CSourceStream 类通过下面的方法来给线程发送请求。
CSourceStream::Exit，**CSourceStream::Init**，**CSourceStream::Pause**，**CSourceStream::Run**，**CSourceStream::Stop**
发给线程的第一个请求一定是 Init，Exit 请求用来结束线程，实际上，我们没有必要亲自调用这些函数，因为在 **CSourceStream::Active** and **CSourceStream::Inactive** 都调用了所需要的方法。
这个类也提供几个 handler 方法
CSourceStream::OnThreadCreate
CSourceStream::OnThreadDestroy
CSourceStream::OnThreadStartPlay
首先分析一下数据成员

```
CSource *m_pFilter;
```

靠，就一个数据成员，用来指向这个 pin 连接的 Filter。
看看成员函数吧，
1CSourceStream::OnThreadCreate

```
virtual HRESULT OnThreadCreate(void);
```

线程处理 **CSourceStream::ThreadProc** 在第一次接收到 Init 请求的时候，就会调用这个方法，在基类的实现中，这个函数没有作任何的事情，派生类可以在这个函数中做一些初始化线程的工作。
2CSourceStream::OnThreadDestroy
当线程退出的时候，会调用这个函数，基类的实现没有作任何的事情，你可以在派生类中做清理现场的工作。
3CSourceStream::OnThreadStartPlay
在 **CSourceStream::DoBufferProcessingLoop** 开始的时候，会调用到这个函数
4CSourceStream::Active
5CSourceStream::Inactive
6CSourceStream::GetRequest

```
Command GetRequest(void);
```

这个函数可以等待下一个线程请求。这个方法重载了 **CAMThread::GetRequest**

7CSourceStream::CheckRequest

非阻塞情况下，这个函数用来查询是否有个线程请求

8CSourceStream::ThreadProc

这个函数是工作线程的函数体，重载了 **CAMThread::ThreadProc** 方法。

```
virtual DWORD ThreadProc(void);
```

这个函数会无限的等待线程的请求，通过调用 **CAMThread::GetRequest** 方法，如果它接收到 **CSourceStream::Run** or **CSourceStream::Pause** 请求，它就调用 **CSourceStream::DoBufferProcessingLoop** 方法，DoBufferProcessingLoop 会一直往外推数据，直到它接收到一个 **CSourceStream::Stop** 请求，线程处理器当接收到 **CSourceStream::Exit** 请求，就会退出了。

9CSourceStream::DoBufferProcessingLoop

```
virtual HRESULT DoBufferProcessingLoop(void);
```

这个方法其实是线程的实现函数，在这个循环中处理数据，并且将数据传递给下下游的 Filter，每次，这个方法都会申请一个空的内存 sample，然后将这个 sample 传递给 **CSourceStream::FillBuffer** 方法，FillBuffer 方法在派生类中一定要实现哦，这个函数是用来产生数据，并将数据拷贝到 sample。

当发生下面的情形时，循环停止，退出

1 当 **IMemInputPin::Receive** 方法拒绝 samples。

2FillBuffer 返回 False，表示结束发送数据了

3 线程接收到一个 **CSourceStream::Stop** 请求。

10CSourceStream::CheckMediaType

```
virtual HRESULT CheckMediaType( const CMediaType *pMediaType);
```

确认是否支持指定的媒体类型。

11CSourceStream::GetMediaType

```
virtual HRESULT GetMediaType( int iPosition, CMediaType *pMediaType);

virtual HRESULT GetMediaType(  CMediaType *pMediaType);
```

获取指定位置的媒体类型。

12CSourceStream::Init

```
HRESULT Init(void);
```

这个函数用来启动一个线程，**CSourceStream::Active** 方法会调用这个函数的。

13CSourceStream::Exit

The **CSourceStream::Inactive** method calls this method.

14CSourceStream::Run

15CSourceStream::Pause

CSourceStream::Active 会调用这个方法的。当 **CSourceStream::ThreadProc** 接收这个请求，它会调用 **CSourceStream::DoBufferProcessingLoop** 方法。

16CSourceStream::Stop

The **CSourceStream::Inactive** method calls this method.

17CSourceStream::FillBuffer

```
virtual HRESULT FillBuffer(  IMediaSample *pSample) PURE;
```

最重要的一个函数。

派生类中一定要实现这个函数，媒体 samples 没有给这个方法提供时间戳，派生类应该调用

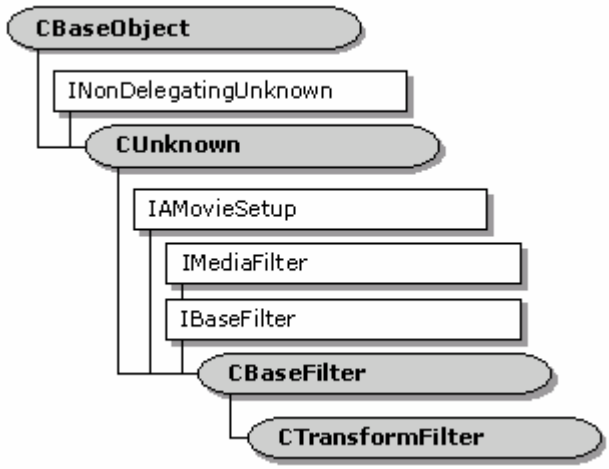
IMediaSample::SetTime 方法来设置时间戳。

3.3.3CTransformFilter

```
class AM_NOVTABLE CTransformFilter : public CBaseFilter
{

public:

    virtual int GetPinCount();
    virtual CBasePin * GetPin(int n);
    STDMETHODIMP FindPin(LPCWSTR Id, IPin **ppPin);

    STDMETHODIMP Stop();
    STDMETHODIMP Pause();

public:

    CTransformFilter(TCHAR *, LPUNKNOWN, REFCLSID clsid);
#ifdef UNICODE
    CTransformFilter(CHAR *, LPUNKNOWN, REFCLSID clsid);
#endif
    ~CTransformFilter();

    // These must be supplied in a derived class

    virtual HRESULT Transform(IMediaSample * pIn, IMediaSample *pOut);

    // check if you can support mtIn
    virtual HRESULT CheckInputType(const CMediaType* mtIn) PURE;

    // check if you can support the transform from this input to this output
    virtual HRESULT CheckTransform(const CMediaType* mtIn, const
CMediaType* mtOut) PURE;

    virtual HRESULT DecideBufferSize( IMemAllocator * pAllocator,
                                      ALLOCATOR_PROPERTIES *pprop) PURE;

    // override to suggest OUTPUT pin media types
  virtual HRESULT GetMediaType(int iPosition, CMediaType *pMediaType) PURE;

    // you can also override these if you want to know about streaming
```

```
    virtual HRESULT StartStreaming();
    virtual HRESULT StopStreaming();

    // override if you can do anything constructive with quality notifications
    virtual HRESULT AlterQuality(Quality q);

    // override this to know when the media type is actually set
    virtual HRESULT SetMediaType(PIN_DIRECTION direction,const CMediaType
*pmt);

    // chance to grab extra interfaces on connection
    virtual HRESULT CheckConnect(PIN_DIRECTION dir,IPin *pPin);
    virtual HRESULT BreakConnect(PIN_DIRECTION dir);
    virtual HRESULT CompleteConnect(PIN_DIRECTION direction,IPin
*pReceivePin);

    // chance to customize the transform process
    virtual HRESULT Receive(IMediaSample *pSample);

    // Standard setup for output sample
    HRESULT InitializeOutputSample(IMediaSample *pSample, IMediaSample
**ppOutSample);

    // if you override Receive, you may need to override these three too
    virtual HRESULT EndOfStream(void);
    virtual HRESULT BeginFlush(void);
    virtual HRESULT EndFlush(void);
    virtual HRESULT NewSegment( REFERENCE_TIME tStart,
                                REFERENCE_TIME tStop,double dRate);

#ifdef PERF
    // Override to register performance measurement with a less generic string
    // You should do this to avoid confusion with other filters
    virtual void RegisterPerfId()
        {m_idTransform = MSR_REGISTER(TEXT("Transform"));}
#endif // PERF

protected:

#ifdef PERF
    int m_idTransform;                  // performance measuring id
#endif
    BOOL m_bEOSDelivered;               // have we sent EndOfStream
    BOOL m_bSampleSkipped;              // Did we just skip a frame
```

```
    BOOL m_bQualityChanged;                 // Have we degraded?
    // critical section protecting filter state.
    CCritSec m_csFilter;
    CCritSec m_csReceive;
    friend class CTransformInputPin;
    friend class CTransformOutputPin;
    CTransformInputPin *m_pInput;
    CTransformOutputPin *m_pOutput;
};
```



CTransformFilte 是用来开发传输 Filter 的基类。利用这个类生成的传输 Filter 可以支持一个输入 pin 和几个输出 pin。有一个和几个输入 pin，和几个输出 pin，传输 filter 从输入的 pin 中接收数据，然后处理数据，然后通过输出 pin 将数据传送到下一级的 filter。这就是传输 filter 的基本功能。

如果想要创建一个 processes data in place 的 Filter，那么你就要从 **CTransInPlaceFilter** 创建你的 filter 了。

传输 Filter 的输入 pin 是从 **CTransformInputPin** 生成的对象，输出 pin 是从 **CTransformOutputPin** 类生成的对象。你不用从这两个类中派生自己的类，直接用这两个类声明对象即可。通过 CTransformFilter 类的接口方法，我们可以直接调用 Pin 提供的方法所以，你也可以从 CTransformFilter 派生自己的 filter 类，Filter 在 **CTransformFilter::GetPin** 方法中创建了输出和输入两种 pin，如果你派生了自己的 pin 类，那么你一定要记得实现 GetPin 方法，因为 Filter 将通过这个方法来创建 pin 对象。

如果你想使用这个类，你要从 **CTransformFilter** 类中派生一个新类，然后重载下面的几个函数。

CTransformFilter::CheckInputType
CTransformFilter::CheckTransform
CTransformFilter::DecideBufferSize
CTransformFilter::GetMediaType
CTransformFilter::Transform

根据你自己的 Filter 的要求，你或许要重载其他的方法。

Media Types

在 pin 连接过程中采用什么的媒体类型，一般来说是由上游的 Filter 来提议，下游的 Filter 的输入 pin 不提议采用什么格式的媒体类型进行连接。之所以这样设计的原因是因为上游的

Filter 可以更详细的数据媒体格式的信息。例如，对于视频数据，上游的 Filter 知道视频的尺寸，桢率等信息，但是处于中间的传输 Filter 并不知道这些信息，如果你想改变媒体类型，你可以重载输入 pin 的 GetMediaType 方法，当上游的 Filter 提议使用某一种媒体类型进行连接，然后输入 pin 就会调用 CheckInputType 方法。

在一个 Filter 的输入 pin 被连接之前，任何到 Filter 的输出 pin 的连接请求都会被拒绝，并且不会返回任何的媒体类型，因为在输入 pin 连接前，输出 pin 也不知道自己能够支持什么类型的媒体格式，所以它就拒绝连接。当输入 pin 被上游 Filter 的 pin 连接以后，此时如果你调用输出 pin 上的 GetMediaType 方法，就会返回输出 pin 支持的媒体类型的列表，下游的 filter 的 pin 调用 CheckTransform 方法来查询输出 pin 都支持什么类型的媒体格式。上面提到的两个函数都是纯虚函数，一般的来说，输入 pin 的类型就决定了输出 pin 所支持的媒体格式。

Streaming

传输 filter 不会保存数据队列的，每一个输出的 sample 都通过 **IMemInputPin::Receive** 方法调用传递给下一个 filter，下游输入 pin 在 Receive 方法中会调用 Filter 的 Transform 来处理数据。

先分析一下数据成员

BOOL m_bEOSDelivered; 用来标示 Filter 是否发送了 end of stream 的通知消息的标志

BOOL m_bSampleSkipped; 用来标示是否有 sample 被丢弃的标志

BOOL m_bQualityChanged; 用来标示质量是否改变的标志

CCritSec m_csFilter; 用来保护 Filter 状态的临界区

CCritSec m_csReceive; 用来保护 stream 数据流的状态的临界区

CTransformInputPin *m_pInput; 指向输入 pin 的指针

CTransformOutputPin *m_pOutput; 指向输出 pin 的指针

下面我们分析一下函数成员

1 CTransformFilter::GetPinCount

获取 Filter 支持的 pin 的数量

2 CTransformFilter::GetPin

virtual CBasePin *GetPin(int *n*);

这个函数在 CBasePin 基类中是一个虚函数，当第一次调用这个方法的时候，会创建输出 pin 和输入 pin。

这个方法并不会将返回的 pin 的指针的计数增加，所以调用者要自己增加指针的计数。

3 CTransformFilter::Transform

virtual HRESULT Transform(IMediaSample **pIn*, IMediaSample **pOut*);

重载这个方法来产生输出数据，这个方法从指定的输入 pin 提供的 samples 中读取数据，然后在输出 pin 指定的 sample 中写入新的数据。这个函数中就存在数据处理的过程

在 Filter 调用这个函数之前，它会将输入 sample 的属性复制到输出的 sample，Transform 方法可以设置两个 sample 之间的区别。如果 filter 不能传递 sample，那么这个函数返回 false

4 CTransformFilter::StartStreaming

virtual HRESULT StartStreaming(void);

当 Filter 转换到暂停状态的时候会调用这个函数，在基类中的这个函数实现没有做任何事情，所以，你可以在派生类作一下事情

5 CTransformFilter::StopStreaming

virtual HRESULT StopStreaming(void);

当 Filter 的状态切换到停止状态的时候，会调用这个函数，

6 CTransformFilter::AlterQuality

virtual HRESULT AlterQuality(Quality *q*);

这个方法用来通知 Filter 一些质量要求发生了改变

7 CTransformFilter::SetMediaType

virtual HRESULT SetMediaType(PIN_DIRECTION *direction*,
const CMediaType **pmt*);

当用来设置某个 pin 的媒体类型的时候可以调用这个函数

当和 filter 连接的 pin 的媒体被设置的时候，CTransformInputPin::SetMediaType and CTransformOutputPin::SetMediaType 都会调用这个函数，但是在基类中，这个方法没有做任何事情，你可以在派生类中做你想的事情。

8 CTransformFilter::CheckConnect

virtual HRESULT CheckConnect(PIN_DIRECTION *dir*, IPin **pPin*);

CTransformInputPin::CheckConnect and CTransformOutputPin::CheckConnect 的方法在 pin 连接的过程会调用这个函数，

9 CTransformFilter::BreakConnect

virtual HRESULT BreakConnect(PIN_DIRECTION *dir*);

当两个 pin 断开连接的时候，CTransformInputPin::BreakConnect and CTransformOutputPin::BreakConnect 方法都会调用这个方法，如果你重载了 checkConnect 方法，你可以重载这个方法用来释放你在 checkconnect 方法中分配的资源。

10 CTransformFilter::CompleteConnect

virtual HRESULT CompleteConnect(PIN_DIRECTION *direction*, IPin **pReceivePin*);

当 pin 完成连接的时候，**CTransformInputPin::CompleteConnect** and **CTransformOutputPin::CompleteConnect** 方法都会调用这个方法，

11 CTransformFilter::Receive

HRESULT Receive(IMediaSample **pSample*);

当 filter 的输入 pin 开始接受 sample 的时候，它会调用这个方法，这个方法其实调用了 **CTransformFilter::InitializeOutputSample** 方法来准备一个输出的 sample，然后调用 **CTransformFilter::Transform** 方法完成数据的传输，Transform 方法处理了输入数据产生输出数据。

当 Transform 方法返回一个 FASLE 时，Receive 方法就开始丢弃数据，在开始丢弃第一个 sample 的时候，Filter 会给图表管理器发送一个 EC_QUALITY_CHANGE 消息通知。

12 CTransformFilter::InitializeOutputSample

HRESULT InitializeOutputSample(
IMediaSample **pSample*, IMediaSample ***ppOutSample*);

这个方法可以为输出 pin 产生一个 sample。然后将输入 sample 的属性复制给输出的 sample。

13 CTransformFilter::EndOfStream

virtual HRESULT EndOfStream(void);

这个方法用来通知 Filter 数据传输结束了，输入 pin 上的 **CTransformInputPin::EndOfStream** 方法会调用这个函数，这个方法会将数据传输结束的通知传递给下游的 filter，如果派生类使用了一个工作线程用来发送 sample，那么它应该重载这个方法处理数据。

14 CTransformFilter::BeginFlush

virtual HRESULT BeginFlush(void);

在开始 flush 动作之前，输入 pin 上的 **CTransformInputPin::BeginFlush** 方法会调用这个方法，

这个方法然后调用将这个消息传递给下一个 filter。

如果派生类使用了一个工作线程来传递 sample，那么，在 flush 动作中，它要丢弃那么等待传递的数据。当然了，filter 可以在 BeginFlush 或者 EndFlush 方法丢弃数据，但是，这里要指出的是，BeginFlush 的调用和数据流的线程并不是同步的，如果 BeginFlush 丢弃了数据，Filter 在 BeginFlush 和 EndFlush 调用之间就不要再处理数据了。

15 CTransformFilter::EndFlush

16 CTransformFilter::NewSegment

下面是四个重要的函数纯虚函数，你在你的派生类都要重载这四个函数哦

17 CTransformFilter::CheckInputType

```
virtual HRESULT CheckInputType(  const CMediaType *mtIn) PURE;
```

这个函数用来检查指定的媒体类型是否适合输入 pin。

18 CTransformFilter::CheckTransform

```
virtual HRESULT CheckTransform(
    const CMediaType *mtIn,  const CMediaType *mtOut) PURE;
```

这个函数用来检验输入的媒体类型和输出的媒体类型是否相匹配

如果 Filter 可以接受指定的输入和输出的媒体类型，那么这个函数就要返回 ok。

19 CTransformFilter::DecideBufferSize

```
virtual HRESULT DecideBufferSize(
   IMemAllocator *pAlloc, ALLOCATOR_PROPERTIES *ppropInputRequest) PURE;
```

用来设置输出 pin 上的内存的需求。

输出 pin 上的 **CTransformOutputPin::DecideBufferSize** 方法调用这个函数，派生类一定要实现这个方法。

20 CTransformFilter::GetMediaType

```
virtual HRESULT GetMediaType(int iPosition, CMediaType *pMediaType) PURE;
```

输出 pin 上的 **CTransformOutputPin::GetMediaType** 函数会调用这个方法的。

21 CTransformFilter::Stop

```
HRESULT Stop(void);
```

停止 filter 的活动。

22 CTransformFilter::Pause

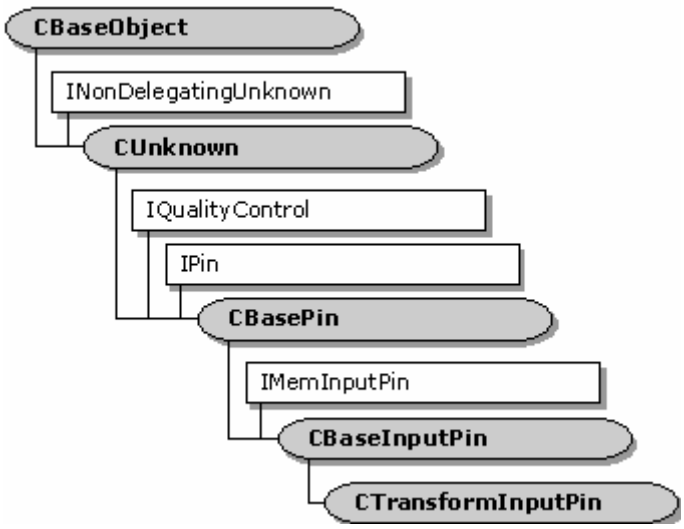
暂停 filter 的活动

23 CTransformFilter::FindPin

```
HRESULT FindPin(  LPCWSTR Id,  IPin **ppPin);
```

根据指定的 id 返回 pin 的指针。

3.3.4CTransformInputPin



看看类的派生图吧，这个类就是从 CTransformInputPin。这个类的对象就是用于传输 Filter 的输入 pin 的。

一般的来说，你不需要从这个类中派生你自己的类，如果你要从这个类中派生自己的类，你必须要重载 **CTransformFilter::GetPin** 方法，这个方法用来创建一个 pin 的实例对象。

下面分析一下数据成员：

只有一个数据成员

```
CTransformFilter *m_pTransformFilter;
```

这个指针指向和这个 pin 相连的 Filter。

看看成员函数：

1 CTransformInputPin::CheckConnect

```
HRESULT CheckConnect( IPin *pPin);
```

这个方法用来判断某个 pin 的连接是否适合。

这个方法实现了 CBasePin 接口的 **CBasePin::CheckConnect** 方法，它调用了 Filter 的 **CTransformFilter::CheckConnect** 方法。

2 CTransformInputPin::BreakConnect

```
HRESULT BreakConnect(void);
```

这个方法用来释放一个连接，这个方法实现了 **CBaseInputPin::BreakConnect** 接口方法，它调用了 **CTransformFilter::BreakConnect** 方法。

3 CTransformInputPin::CompleteConnect

```
HRESULT CompleteConnect( IPin *pReceivePin);
```

这个函数实现了 **CBasePin::CompleteConnect** 接口函数，这个方法调用了 Filter 上的 **CTransformFilter::CompleteConnect**。

4 CTransformInputPin::CheckMediaType

```
HRESULT CheckMediaType( const CMediaType *mtIn0);
```

这个函数实现了 **CBasePin::CheckMediaType** 接口，这个方法调用了相连接的 Filter 上的 **CTransformFilter::CheckInputType**，这个函数也是一个纯虚函数，Filter 上派生类必须要实现 CheckInputType 方法来决定一个输入类型是否适合。

当一个 Filter 上的输出 pin 也连接到 Filter 上，这个方法也调用 Filter 上的 CheckTransform 来查询一下媒体类型是否和 Filter 的输出 pin 是否匹配。

5 CTransformInputPin::SetMediaType

```
HRESULT SetMediaType(  const CMediaType *mt);
```

为 pin 之间的连接设置一个指定的媒体类型，pin 在调用这个函数之前一定要确认媒体类型对于两个 pin 都匹配。

6 CTransformInputPin::CheckStreaming

```
HRESULT CheckStreaming(void);
```

这个方法可以发现 pin 是否可以接受 sample。

7 CTransformInputPin::CurrentMediaType

返回当前 pin 连接所采用的媒体类型，就是返回 **CBasePin**::m_mt 成员变量。

8 CTransformInputPin::QueryId

```
HRESULT QueryId(    LPWSTR *Id);
```

这个方法返回 pin 的 ID。

9 CTransformInputPin::EndOfStream

这个方法会调用相连的 Filter 的 **CTransformFilter::EndOfStream** 方法来向下游传递一个 end fo stream 消息。

10 CTransformInputPin::BeginFlush

11 CTransformInputPin::EndFlush

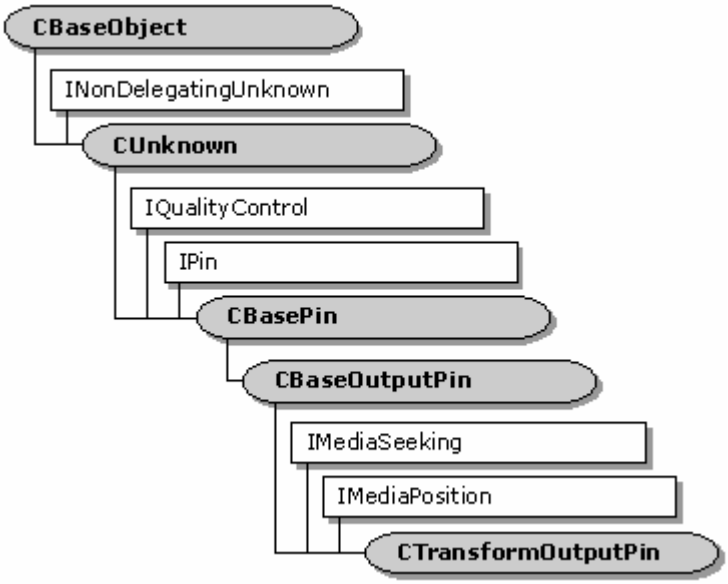
这两个函数分别调用相连接的 Filter 上的相应的方法，进行一个 flush 操作

12 CTransformInputPin::Receive

```
HRESULT Receive(    IMediaSample *pSample);
```

这个方法调用了相连接的 filter 上的 **CTransformFilter::Receive** 方法进行数据的处理。

3.3.5CTransformOutputPin



下面分析一下数据成员：

只有一个数据成员

```
CTransformFilter *m_pTransformFilter;
IUnknown *m_pPosition;
```

这个指针指向和这个 pin 相连的 Filter。
看看成员函数：
1 CTransformInputPin::CheckConnect
HRESULT CheckConnect(IPin *pPin);
这个方法用来判断某个 pin 的连接是否适合。
这个方法实现了 CBasePin 接口的 **CBaseOutputPin::CheckConnect** 方法，它调用了 Filter 的 **CTransformFilter::CheckConnect** 方法。
2 CTransformInputPin::BreakConnect
HRESULT BreakConnect(void);
这个方法用来释放一个连接，这个方法实现了 **CBaseOutputPin::BreakConnect** 接口方法，它调用了 **CTransformFilter::BreakConnect** 方法。
3 CTransformInputPin::CompleteConnect
HRESULT CompleteConnect(IPin *pReceivePin);
这个函数实现了 **CBaseOutputPin::CompleteConnect** 接口函数，这个方法调用了 Filter 上的 **CTransformFilter::CompleteConnect**。
4 CTransformInputPin::CheckMediaType
HRESULT CheckMediaType(const CMediaType *mtIn0);
这个函数实现了 **CBasePin::CheckMediaType** 接口，这个方法调用了相连接的 Filter 上的 **CTransformFilter::CheckInputType**，这个函数也是一个纯虚函数，Filter 上派生类必须要实现 CheckInputType 方法来决定一个输入类型是否适合。
当一个 Filter 上的输出 pin 也连接到 Filter 上，这个方法也调用 Filter 上的 CheckTransform 来查询一下媒体类型是否和 Filter 的输出 pin 是否匹配。
5 CTransformInputPin::SetMediaType
HRESULT SetMediaType(const CMediaType *mt);
为 pin 之间的连接设置一个指定的媒体类型，pin 在调用这个函数之前一定要确认媒体类型对于两个 pin 都匹配。
7 CTransformOutputPin::DecideBufferSize
HRESULT DecideBufferSize(IMemAllocator *pAlloc,
ALLOCATOR_PROPERTIES *ppropInputRequest);
这个方法实现了 **CBaseOutputPin::DecideBufferSize** 接口方法，这个方法调用了 Filter 的 **CTransformFilter::DecideBufferSize**。
8 CTransformOutputPin::GetMediaType
HRESULT GetMediaType(int iPosition, CMediaType *pMediaType);
这个方法实现了 **CBasePin::GetMediaType** 接口方法，如果一个 Filter 上的输入 pin 没有连接，这个方法就返回 VFW_S_NO_MORE_ITEMS.它调用 **CTransformFilter::GetMediaType** 来获取媒体类型。
9 CTransformInputPin::CurrentMediaType
返回当前 pin 连接所采用的媒体类型，就是返回 **CBasePin**::m_mt 成员变量。
10CTransformInputPin::QueryId
HRESULT QueryId(LPWSTR *Id);
这个方法返回 pin 的 ID。
11 CTransformOutputPin::Notify
HRESULT Notify(IBaseFilter *pSelf, Quality q);

3.3.6CTransInPlaceFilter

```
class AM_NOVTABLE CTransInPlaceFilter : public CTransformFilter
{

public:

    virtual CBasePin *GetPin(int n);

public:


    CTransInPlaceFilter(TCHAR *, LPUNKNOWN, REFCLSID clsid, HRESULT *,
                        bool bModifiesData = true);
#ifdef UNICODE
    CTransInPlaceFilter(CHAR *, LPUNKNOWN, REFCLSID clsid, HRESULT *,
                        bool bModifiesData = true);
#endif

    HRESULT GetMediaType(int iPosition, CMediaType *pMediaType)
        {   DbgBreak("CTransInPlaceFilter::GetMediaType should never be
called");
            return E_UNEXPECTED;
        }

    // This is called when we actually have to provide out own allocator.
    HRESULT DecideBufferSize(IMemAllocator*, ALLOCATOR_PROPERTIES
*);

    HRESULT CheckTransform(const CMediaType *mtIn, const CMediaType
*mtOut)
    {
        return S_OK;
    };

    HRESULT CompleteConnect(PIN_DIRECTION dir,IPin *pReceivePin);

    // chance to customize the transform process
    virtual HRESULT Receive(IMediaSample *pSample);

    virtual HRESULT Transform(IMediaSample *pSample) PURE;

#ifdef PERF
```

```
    // Override to register performance measurement with a less generic string
    // You should do this to avoid confusion with other filters
    virtual void RegisterPerfId()
            {m_idTransInPlace = MSR_REGISTER(TEXT("TransInPlace"));}
#endif // PERF
  protected:

    IMediaSample * CTransInPlaceFilter::Copy(IMediaSample *pSource);

#ifdef PERF
    int m_idTransInPlace;                    // performance measuring id
#endif // PERF
    bool    m_bModifiesData;                 // Does this filter change the data?

    // these hold our input and output pins

    friend class CTransInPlaceInputPin;
    friend class CTransInPlaceOutputPin;

    CTransInPlaceInputPin    *InputPin() const
    {
        return (CTransInPlaceInputPin *)m_pInput;
    };
    CTransInPlaceOutputPin *OutputPin() const
    {
        return (CTransInPlaceOutputPin *)m_pOutput;
    };

    //    Helper to see if the input and output types match
    BOOL TypesMatch()
    {
        return InputPin()->CurrentMediaType() ==
                   OutputPin()->CurrentMediaType();
    }

    //    Are the input and output allocators different?
    BOOL UsingDifferentAllocators() const
    {
        return InputPin()->PeekAllocator() != OutputPin()->PeekAllocator();
    }
}; // CTransInPlaceFilter
```

3.3.7 CTransInPlaceInputPin

3.3.8CTransInPlaceOutputPin

3.3.4CVideoTransformFilter

```
class CVideoTransformFilter : public CTransformFilter
{
   public:

      CVideoTransformFilter(TCHAR *, LPUNKNOWN, REFCLSID clsid);
      ~CVideoTransformFilter();
      HRESULT EndFlush();

#ifdef PERF

      virtual void RegisterPerfId() {
            m_idSkip            = MSR_REGISTER(TEXT("Video Transform Skip
frame"));
            m_idFrameType     = MSR_REGISTER(TEXT("Video transform frame
type"));
            m_idLate            = MSR_REGISTER(TEXT("Video Transform
Lateness"));
            m_idTimeTillKey = MSR_REGISTER(TEXT("Video Transform Estd. time
to next key"));
            CTransformFilter::RegisterPerfId();
      }
#endif

   protected:

      int m_nKeyFramePeriod;

      int m_nFramesSinceKeyFrame;

      BOOL m_bSkipping;
#ifdef PERF
      int m_idFrameType;              // MSR id Frame type.    1=Key, 2="non-key"
      int m_idSkip;                   // MSR id skipping
      int m_idLate;                   // MSR id lateness
      int m_idTimeTillKey;            // MSR id for guessed time till next key frame.
#endif
```

```
    virtual HRESULT StartStreaming();

    HRESULT AbortPlayback(HRESULT hr);      // if something bad happens

    HRESULT Receive(IMediaSample *pSample);

    HRESULT AlterQuality(Quality q);

    BOOL ShouldSkipFrame(IMediaSample * pIn);

    int m_itrLate;                  // lateness from last Quality message
                                    // (this overflows at 214 secs late).
    int m_tDecodeStart;             // timeGetTime when decode started.
    int m_itrAvgDecode;             // Average decode time in reference units.

    BOOL m_bNoSkip;                 // debug - no skipping.
    BOOL m_bQualityChanged;

    int m_nWaitForKey;
};
```

3.3.9CBaseRenderer

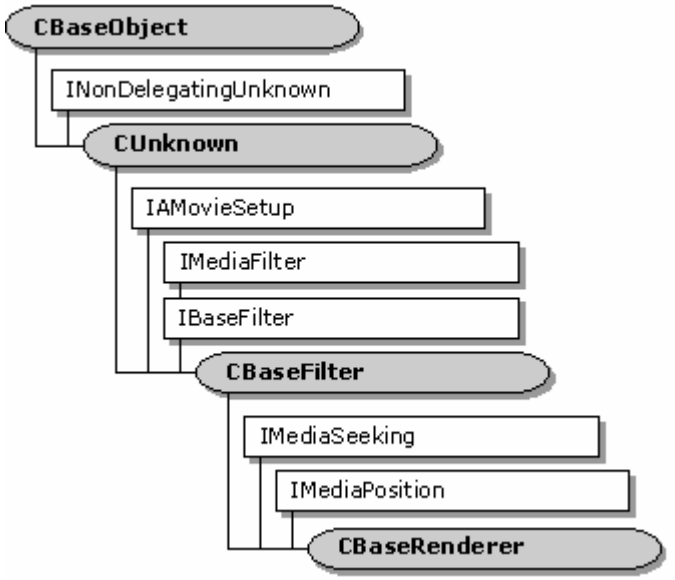
```
class CBaseRenderer : public CBaseFilter
{
protected:

    friend class CRendererInputPin;

    friend void CALLBACK EndOfStreamTimer(UINT uID,          // Timer identifier
                                          UINT uMsg,         // Not currently
used DWORD_PTR dwUser,    // User information
DWORD_PTR dw1,         // Windows reserved
DWORD_PTR dw2);        // Is also reserved

    CRendererPosPassThru *m_pPosition;    // Media seeking pass by object
    CAMEvent m_RenderEvent;               // Used to signal timer events
    CAMEvent m_ThreadSignal;              // Signalled to release worker thread
    CAMEvent m_evComplete;                // Signalled when state complete
    BOOL m_bAbort;                        // Stop us from rendering more data
    BOOL m_bStreaming;                    // Are we currently streaming
    DWORD_PTR m_dwAdvise;                 // Timer advise cookie
    IMediaSample *m_pMediaSample;         // Current image media sample
    BOOL m_bEOS;                          // Any more samples in the stream
```

```
    BOOL m_bEOSDelivered;                   // Have we delivered an
EC_COMPLETE
    CRendererInputPin *m_pInputPin;         // Our renderer input pin object
    CCritSec m_InterfaceLock;               // Critical section for interfaces
    CCritSec m_RendererLock;                // Controls access to internals
    IQualityControl * m_pQSink;             // QualityControl sink
    BOOL m_bRepaintStatus;                  // Can we signal an EC_REPAINT
    //  Avoid some deadlocks by tracking filter during stop
    volatile BOOL  m_bInReceive;            // Inside Receive between
PrepareReceive
                                            // And actually processing the sample
    REFERENCE_TIME m_SignalTime;            // Time when we signal
EC_COMPLETE
    UINT m_EndOfStreamTimer;                // Used to signal end of stream
    CCritSec m_ObjectCreationLock;          // This lock protects the creation and

public:

    CBaseRenderer(REFCLSID RenderClass, // CLSID for this renderer
                  TCHAR *pName,         // Debug ONLY description
                  LPUNKNOWN pUnk,       // Aggregated owner object
                  HRESULT *phr);        // General OLE return code

    ~CBaseRenderer();

    // Overriden to say what interfaces we support and where

    virtual HRESULT GetMediaPositionInterface(REFIID riid,void **ppv);
    STDMETHODIMP NonDelegatingQueryInterface(REFIID, void **);

    virtual HRESULT SourceThreadCanWait(BOOL bCanWait);

#ifdef DEBUG
    // Debug only dump of the renderer state
    void DisplayRendererState();
#endif
    virtual HRESULT WaitForRenderTime();
    virtual HRESULT CompleteStateChange(FILTER_STATE OldState);

    // Return internal information about this filter

    BOOL IsEndOfStream() { return m_bEOS; };
    BOOL IsEndOfStreamDelivered() { return m_bEOSDelivered; };
    BOOL IsStreaming() { return m_bStreaming; };
```

```
    void SetAbortSignal(BOOL bAbort) { m_bAbort = bAbort; };
    virtual void OnReceiveFirstSample(IMediaSample *pMediaSample) { };
    CAMEvent *GetRenderEvent() { return &m_RenderEvent; };

    // Permit access to the transition state

    void Ready() { m_evComplete.Set(); };
    void NotReady() { m_evComplete.Reset(); };
    BOOL CheckReady() { return m_evComplete.Check(); };

    virtual int GetPinCount();
    virtual CBasePin *GetPin(int n);
    FILTER_STATE GetRealState();
    void SendRepaint();
    void SendNotifyWindow(IPin *pPin,HWND hwnd);
    BOOL OnDisplayChange();
    void SetRepaintStatus(BOOL bRepaint);

    // Override the filter and pin interface functions

    STDMETHODIMP Stop();
    STDMETHODIMP Pause();
    STDMETHODIMP Run(REFERENCE_TIME StartTime);
    STDMETHODIMP GetState(DWORD dwMSecs,FILTER_STATE *State);
    STDMETHODIMP FindPin(LPCWSTR Id, IPin **ppPin);

    // These are available for a quality management implementation

    virtual void OnRenderStart(IMediaSample *pMediaSample);
    virtual void OnRenderEnd(IMediaSample *pMediaSample);
    virtual HRESULT OnStartStreaming() { return NOERROR; };
    virtual HRESULT OnStopStreaming() { return NOERROR; };
    virtual void OnWaitStart() { };
    virtual void OnWaitEnd() { };
    virtual void PrepareRender() { };

#ifdef PERF
    REFERENCE_TIME m_trRenderStart; // Just before we started drawing
                                    // Set in OnRenderStart, Used in
OnRenderEnd
    int m_idBaseStamp;              // MSR_id for frame time stamp
    int m_idBaseRenderTime;         // MSR_id for true wait time
    int m_idBaseAccuracy;           // MSR_id for time frame is late (int)
#endif
```

```
// Quality management implementation for scheduling rendering

virtual BOOL ScheduleSample(IMediaSample *pMediaSample);
virtual HRESULT GetSampleTimes(IMediaSample *pMediaSample,
                               REFERENCE_TIME *pStartTime,
                               REFERENCE_TIME *pEndTime);

virtual HRESULT ShouldDrawSampleNow(IMediaSample *pMediaSample,
                                    REFERENCE_TIME *ptrStart,
                                    REFERENCE_TIME *ptrEnd);

// Lots of end of stream complexities

void TimerCallback();
void ResetEndOfStreamTimer();
HRESULT NotifyEndOfStream();
virtual HRESULT SendEndOfStream();
virtual HRESULT ResetEndOfStream();
virtual HRESULT EndOfStream();

// Rendering is based around the clock

void SignalTimerFired();
virtual HRESULT CancelNotification();
virtual HRESULT ClearPendingSample();

// Called when the filter changes state

virtual HRESULT Active();
virtual HRESULT Inactive();
virtual HRESULT StartStreaming();
virtual HRESULT StopStreaming();
virtual HRESULT BeginFlush();
virtual HRESULT EndFlush();

// Deal with connections and type changes

virtual HRESULT BreakConnect();
virtual HRESULT SetMediaType(const CMediaType *pmt);
virtual HRESULT CompleteConnect(IPin *pReceivePin);

// These look after the handling of data samples
```

```
    virtual HRESULT PrepareReceive(IMediaSample *pMediaSample);
    virtual HRESULT Receive(IMediaSample *pMediaSample);
    virtual BOOL HaveCurrentSample();
    virtual IMediaSample *GetCurrentSample();
    virtual HRESULT Render(IMediaSample *pMediaSample);

    // Derived classes MUST override these
    virtual HRESULT DoRenderSample(IMediaSample *pMediaSample) PURE;
    virtual HRESULT CheckMediaType(const CMediaType *) PURE;

    // Helper
    void WaitForReceiveToComplete();
};
```



CBaseRenderer 类是用来实现一个 Render Filter 的基类。Render Filter 有的地方也翻译成提交 Filter，提交 Filter 是 Graph 图中 Filter 链条中最后一级的 Filter。这个 Filter 最终将数据流提交给播放窗口，或者提交给 file，写入文件。

这个 Render Filter 支持一个 **CRendererInputPin** 类型的输入 pin，为了使用这个基类，首先声明一个新的类从 CBaseRenderer 继承，在派生类中至少要重载下面的几个函数，这几个函数在基类中都是纯虚函数。

CBaseRenderer::CheckMediaType:，这个方法用来接受或者拒绝连接过程中的媒体类型，Filter 在 pin 连接过程中会调用这个函数，

CBaseRenderer::DoRenderSample:，Filter 在运行时接受到的每一个 sample 都会调用这个函数，用来提交 sample，将 sample 中的数据要么提交给窗口播放，要么写入文件。

基类可以处理状态的改变和同步，尽管它没有实现任何质量控制的接口，它也可以安排 sample 的提交。基类也声明了几个 handler 方法，Filter 在处理数据流的特征点会调用这些 handler 方法。在基类中，这些方法是空的，但是派生类可以重载这些函数。

CBaseRenderer::OnReceiveFirstSample handler 方法值得提一下，当 Filter 处于暂停状态的时候，接受到一个 sample，它就会调用到这个函数。当 Filer 从停止状态切换到暂停态时就会发生这种情况，或者当 graph seek 的时候暂停也会发生这种情况。视频提交过滤器在暂停的时候还要利用 sample 中的数据生成一幅静止的图像，当 Filter 从暂停状态切换到运行态时，

它也会将数据流中的第一个 sample 发送到 **CBaseRenderer::OnReceiveFirstSample** 方法中。

CBaseRenderer 类也通过 CRendererPosPassThru 对象暴露了 **IMediaSeeking** and **IMediaPosition** 接口。它会将所有的 seek 请求发送给上一个 filter。

Scheduling

当上游 filter 调用输入 pin 的 **IMemInputPin::Receive** 方法要传递一个 sample 的时候，输入 pin 就会调用 **CBaseRenderer::Receive** 方法，Filter 可以决定是否丢弃 sample 或者理解提交播放，或者将 smaple 排序等待提交播放。

如果 sample 没有时间戳，或者参考时钟无效，Filter 要么立即提交 sample，要么就调用 **CBaseRenderer::ShouldDrawSampleNow** 方法，来确定该如何做。缺省的情况下，是要将 sample 按照时间戳进行排序，等候提交播放。派生类可以重载这个函数来支持质量控制。

为了给 samples 排序，Filter 要调用 **IReferenceClock::AdviseTime** 方法，这个方法创建了 advise request。于是，Receive 方法就开始阻塞，直到排序的时间到，或者等到 filter 改变了状态，阻塞可以通知上游的 filter 不要再发送数据流了，直到当前的 sample 被提交以后才可以冲新发送数据流。

当上游的 filter 调用了提交 filter 的输入 pin 上的 **IPin::EndOfStream** 方法来通知，上游不再发送数据了，那么 render filter 就给图表管理器发送一个 EC_COMPLETE 消息。Filter 直到当前的 sample 停止时间到了才发送这个消息。

个人感觉提交 filter 的内容还是比较的多，因为它要提交，涉及的内容多，包括窗口，文件了，等等，下面分析一下数据成员

BOOL m_bAbort; 是否要停止提交并开始拒绝数据的标志
BOOL m_bEOS; 用来标志 end of stream 消息是否已经到达 filter 了
BOOL m_bEOSDelivered; 用来判断 Filter 是否已经发送了 EC_COMPLETE 消息
BOOL m_bInReceive; 用来判断 Filter 是否正在处理一个 Recive 方法的调用
BOOL m_bRepaintStatus; 用来设置重画消息是否可用
BOOL m_bStreaming; 用来判断 filter 是否正在 streaming data。
DWORD_PTR m_dwAdvise; 用来标示一个排序 sample 的 timer 事件
UINT m_EndOfStreamTimer; timer 事件的一个 id，用来 scheduling EC_COMPLETE notifications
CAMEvent m_evComplete; 当状态转变完成的时候触发的一个事件
CCritSec m_InterfaceLock; 保护 filter 状态的临界区
CCritSec m_ObjectCreationLock;用来保护 filter 内部对象的临界区
CRendererInputPin *m_pInputPin; 输入 pin 的指针
IMediaSample *m_pMediaSample; 指向当前媒体 sample 的指针。
CRendererPosPassThru *m_pPosition;用来回放的时候定位。
IQualityControl *m_pQSink; 来接受质量控制的消息
CCritSec m_RendererLock; 用来保护数据流的临界区
CAMEvent m_RenderEvent; 提交时排序时的事件。
REFERENCE_TIME m_SignalTime; 当前 sample 的停止时间
CAMEvent m_ThreadSignal; 用来释放 streamming thread 的事件。

下面分析一下成员函数

1CBaseRenderer::CancelNotification

这个方法取消用来安排播放的 timer 事件。

2CBaseRenderer::GetMediaPositionInterface

virtual HRESULT GetMediaPositionInterface(REFIID *riid*,void **ppv);
这个方法可以用来获取 Filter 支持的 **IMediaPosition** and **IMediaSeeking** 的接口指针。
3CBaseRenderer::GetPin
virtual CBasePin *GetPin(int *n*);
这个函数有点怪哦，因为提交 filter 一搬都只有一个 pin，就是一个输入 pin，所以参考文档里说参数 n 必须为 0，还不如直接不用参数就是了。
第一次调用这个函数的时候，创建一个 **CRendererInputPin** 实例对象。
4CBaseRenderer::GetPinCount
返回 Filter 支持的 pin 的数量
5CBaseRenderer::GetSampleTimes
virtual HRESULT GetSampleTimes(IMediaSample *pMediaSample,
REFERENCE_TIME *pStartTime, REFERENCE_TIME *pEndTime);
Filter 调用这个函数用来决定如何处理一个 sample，如果这个函数返回一个 sok，那么 filter 会立即提交这个 sample，如果返回 false，filter 就将这个 sample 安排到队列里，根据 sample 上的时间戳进行播放。如果返回一个错误码，filter 将拒绝 sample。
如果 sample 没有时间戳，或者 filter 没有参考时钟，这个方法会立即返回 ok，否则，它会调用 **CBaseRenderer::ShouldDrawSampleNow** 方法，基类的这个方法总是返回一个 false，派生类可以自己实现。
6CBaseRenderer::OnDisplayChange
BOOL CBaseRenderer::OnDisplayChange(void);
这个方法会给图表管理器发送一个 EC_DISPLAY_CHANGED 消息通知，视频提交 Filter 会用到这个方法，当输入 pin 连接好以后，Filter 就调用这个方法给管理器发送一个 EC_DISPLAY_CHANGED 消息。
7CBaseRenderer::PrepareReceive
virtual HRESULT PrepareReceive(IMediaSample *pMediaSample);
这个函数用来进行提交一个 sample 之前的准备工作。
Filter 会在 **CBaseRenderer::Receive** 方法中，在 render sample 之前调用这个方法，当 filter 开始运行时，这个方法用来对 sample 进行播放排序。
如果一个 filter 已经有一个 pending sample，或者已经接收到一个 end of stream 消息，这个方法就会返回一个 E_UNEXPECTED，
8CBaseRenderer::Receive
virtual HRESULT Receive(IMediaSample *pMediaSample);
这个方法用来接收数据流中的下一个 sample。
当 filter 上的输入 pin 从上游的 filter 接收到一个 sample 的时候，就会调用这个方法。
如果 filter 处于运行状态，这个方法执行下面的步骤
1 通过调用 **CBaseRenderer::PrepareReceive**).对 sample 的播放顺序进行排序。
2 通过 **CBaseRenderer::WaitForRenderTime** 调用等待提交。
3 **CBaseRenderer::Render** 提交一个 sample。
4 通过 **CBaseRenderer::ClearPendingSample** 释放内存。
如果 Filter 处于暂停状态，这个方法进行下面的步骤
1 通知派生类有一个可用的 sample 了，通过调用 **CBaseRenderer::OnReceiveFirstSample**).
2 等待提交时间
3 提交 sample
4 释放内存。

在暂停状态时，这个方法会一直等待第二步切换到运行状态，

9CBaseRenderer::Render

virtual HRESULT Render(IMediaSample *pMediaSample);

这个方法用来提交一个 sample，其实在这个方法中，是通过调用 **CBaseRenderer::DoRenderSample** 方法里做了实际的工作。所以你的派生类一定要重载 DoRenderSample 方法。在 DonRenderSample 方法里，调用了 **CBaseRenderer::OnRenderStart** **CBaseRenderer::OnRenderEnd** 方法，所以你的派生类一定要重载这两个方法。

10CBaseRenderer::ScheduleSample

virtual BOOL ScheduleSample(IMediaSample *pMediaSample);

这个方法用来对 sample 的提交排序，

这个方法首先调用 **CBaseRenderer::GetSampleTimes** 来获取 sample 的时间戳来判断 sample 是否应该立即提交还是等待，还是丢弃。如果数据需要立即提交，就要设置一个 **CBaseRenderer::m_RenderEvent** 消息通知，如果在将来提交，调用 **IReferenceClock::AdviseTime** 方法给 sample 安排提交的顺序。

11CBaseRenderer::SendNotifyWindow

void SendNotifyWindow(IPin *pPin, HWND hwnd);

如果上游的 filter 的输出 pin 支持 **IMediaEventSink** interface，这个方法就会给上游的 filter 发送一个 EC_NOTIFY_WINDOW 通知，用一个窗口句柄作参数。

视频提交 Filter 可以重载一下 **CBaseRenderer::CompleteConnect** 方法，并在该方法中调用这个方法，这个方法提供了一种机制用来向上游的 filter 提供窗口的句柄，

12CBaseRenderer::SendRepaint

用来给 Filter 图表管理器发送一个重画的消息通知。

当下面的条件都满足的情况下，给管理器发送一个 EC_REPAINT 通知。

1 输入 pin 连接正常

2filter 没有 flushing 数据。

3 没有接收到 end of stream 消息

4 m_bAbort 标示为 FALSE；

5m_bRepaintStatus 标志为 TRUE；

根据 graph 的状态，EC_REPAINT 可以触发上游的 filter 重新发送一个 sample，filter graph 可以 seek 当前的位置。

这个方法效率不高，禁止滥用。

13CBaseRenderer::SetMediaType

连接到提交 filter 的输入在 **CRendererInputPin::SetMediaType** 方法中设置连接媒体类型的时候，会调用这个函数。

14CBaseRenderer::SignalTimerFired

virtual void SignalTimerFired(void);

当 sample 的提交时间到达的时候，或者提交时间被取消的时候，Filter 就会调用这个方法，这个方法重新将 **CBaseRenderer::m_dwAdvise** 设置为 0；

15CBaseRenderer::SourceThreadCanWait

virtual HRESULT SourceThreadCanWait(BOOL bCanWait);

用来标示是否需要释放数据流的线程。

当用 FALSE 参数调用 SourceThreadCanWait 函数的时候，可以强制 filter 从 **IMemInputPin::Receive** 阻塞调用中返回，当一个 filter 正处于运行状态，它会在 Receive 调用中一直阻塞到 sample 的提交时间到达，当一个 filter 暂停的时候。它会不定期的阻塞 Receive

调用，这种行为控制了数据流的流动。当一个 filter 停止或者 flush 的时候，不应该被阻塞。

CBaseRenderer::WaitForRenderTime 方法可以控制阻塞，在这个方法中，主要依靠两个事件来控制阻塞，**CBaseRenderer::m_RenderEvent** and **CBaseRenderer::m_ThreadSignal**，当调用 SourceThreadCanWait（FALSE）时，就会触发 **m_ThreadSignal，**当 sample 的提交时间到达的时候，就会触发 **m_RenderEvent** 事件。调用 SourceThreadCanWait（TRUE）时会重新设置 **m_ThreadSignal**

CBaseRenderer::Stop and **CBaseRenderer::BeginFlush** 方法会调用 SourceThreadCanWait 方法来触发事件，传递的参数为 FALSE。**CBaseRenderer::Pause**, **CBaseRenderer::Run**, and **CBaseRenderer::EndFlush** 方法会调用 SourceThreadCanWait 方法，传递 TRUE 参数。

16CBaseRenderer::WaitForReceiveToComplete

```
void WaitForReceiveToComplete(void);
```

这个方法就是等待 **CBaseRenderer::Receive** 结束。

CBaseRenderer::Stop and **CBaseRenderer::BeginFlush** 通过调用这个方法来同步 Receive 方法的状态的改变。

17CBaseRenderer::WaitForRenderTime

```
virtual HRESULT WaitForRenderTime(void);
```

这个方法用来等待当前到 sample 的提交时间。

这个方法会一直阻塞直到下面的一件事件发生

1 sample 的提交时间到了，sample 可以被提交了

2 filter 停止或者开始 flushing 数据了。

如果提交时间到了，**CBaseRenderer::m_RenderEvent** 事件被触发，如果状态改变，**CBaseRenderer::m_ThreadSignal** 事件被触发，这个方法等待这两个事件，派生类可以触发其他的事件。

当进入等待的时候，这个方法会调用 **CBaseRenderer::OnWaitStart** 方法，当结束等待的时候，这个方法会调用 **CBaseRenderer::OnWaitEnd** 函数。

18CBaseRenderer::ClearPendingSample

释放当前的 sample。

19CBaseRenderer::GetCurrentSample

```
virtual IMediaSample *GetCurrentSample(void);
```

返回当前正在使用的 sample。如果成功，返回一个 **IMediaSample** 借口指针。

20CBaseRenderer::GetRealState

Returns the value of **CBaseFilter::m_State**.

21CBaseRenderer::GetRenderEvent

返回指向 **CBaseRenderer::m_RenderEvent** 的指针。

22CBaseRenderer::HaveCurrentSample

判断 filter 当前是否拥有 sample。

23CBaseRenderer::IsEndOfStream

判断 filter 是否已经接收到 end of stream 消息

24CBaseRenderer::IsEndOfStreamDelivered

判断 EC_COMPLETE 同通知是否已经发送给图表管理器，返回 **CBaseRenderer::m_bEOSDelivered** 标志。

25CBaseRenderer::IsStreaming

判断 filter 是否正在 streaming 数据。

26CBaseRenderer::SetAbortSignal

用于设置 **CBaseRenderer::m_bAbort** 标志。

27CBaseRenderer::SetRepaintStatus

```
void SetRepaintStatus( BOOL bRepaint);
```

这个方法确保不会给管理器发送多余的重画消息。

下面的方法是关于 filter 状态改变的。

28CBaseRenderer::Active

当 filter 的状态改变到暂停或者运行状态的时候调用这个函数。

29CBaseRenderer::BeginFlush

Filter 的输入 pin**IPin::BeginFlush** 方法会调用这个方法，Filter 会释放 streaming 线程，并且释放等待提交的所有的 sample。

30CBaseRenderer::BreakConnect

输入 pin 在断开连接的时候，在 **CBasePin::BreakConnect**.)方法中调用此方法。

31CBaseRenderer::CheckReady

用来检查一个传输的状态是否完成。

32CBaseRenderer::CompleteConnect

输入 pin 在完成连接的时候，在 **CBasePin::CompleteConnect**.调用此方法。

33CBaseRenderer::CompleteStateChange

```
virtual HRESULT CompleteStateChange( FILTER_STATE OldState);
```

CBaseRenderer::Pause 方法调用这个方法

34CBaseRenderer::EndFlush

输入 pin 的 **IPin::EndFlush** 调用此方法。

35CBaseRenderer::Inactive

输入 pin 在 **CRendererInputPin::Inactive** 方法中调用此方法，filter 在这个方法总调用了 **CBaseRenderer::ClearPendingSample** 来释放 samples。

36CBaseRenderer::NotReady

这个方法的调用可以使得 **CBaseRenderer::GetState** 返回一个 VFW_S_STATE_INTERMEDIATE。

37CBaseRenderer::Ready

38CBaseRenderer::StartStreaming

这个方法调用了 **CBaseRenderer::OnStartStreaming**

39CBaseRenderer::StopStreaming

这个方法总调用了 **CBaseRenderer::OnStopStreaming**，标示停止运行。

40CBaseRenderer::EndOfStream

当 filter 上的输入 pin 接收到一个 end of stream 通知的时候，就会调用这个方法。

这个方法将 **m_bEOS** 标志设置为 TRUE，然后调用 **CBaseRenderer::SendEndOfStream** 来 schedule an EC_COMPLETE event，如果 filter 正在暂停等待一个 sample，这个方法就会完成数据传输状态。

41CBaseRenderer::NotifyEndOfStream

给图表管理器发送一个 EC_COMPLETE 事件通知。

42CBaseRenderer::ResetEndOfStream

这个方法重新清除了 end of stream 条件，filter 可以重新接受数据了，**CBaseRenderer::Stop** and **CBaseRenderer::BeginFlush** 调用这个函数。

43CBaseRenderer::ResetEndOfStreamTimer

取消 EC_COMPLETE 消息通知。

44CBaseRenderer::SendEndOfStream

Filter 有可能在 sample 的停止时间之前接收到一个 end of stream 通知，此时，filter 就会等待，直到给 filter 管理器发送了 EC_COMPLETE 消息。

45CBaseRenderer::TimerCallback

在 **CBaseRenderer::SendEndOfStream** 方法中使用了一个 timer 事件用来安排一个 EC_COMPLETE 通知，**CBaseRenderer::TimerCallback** 就是 timer 事件的回调函数。

在 timercallback 函数再次的调用了 SendEndOfStream 方法，用来判断是否发送一个 EC_COMPLETE 消息，或则重新设置一个 timer。

CBaseRenderer::ResetEndOfStreamTimer 用来取消一个 timer 事件。

46CBaseRenderer::OnReceiveFirstSample

CBaseRenderer::Receive 方法会调用此函数，这个主要为了视频 render 设计的，当一个视频 render 暂停的时候，它会将第一个接受的 sample 设置成静止的图片。

47CBaseRenderer::OnRenderEnd

当一个 sample 提交以后会调用这个函数。

48CBaseRenderer::OnRenderStart

当一个 sample 被提交之前调用这个函数。

49 CBaseRenderer::OnStartStreaming

当 filter 开始 streaming 的时候调用这个函数

50 CBaseRenderer::OnStopStreaming

CBaseRenderer::StopStreaming 方法会调用这个方法。

51 CBaseRenderer::OnWaitEnd

在 **CBaseRenderer::WaitForRenderTime** 方法中，当 filter 的提交完毕的时候会调用这个方法，如果你想实现质量控制，你可能派生这个方法以及 **CBaseRenderer::OnWaitStart** 方法，

52CBaseRenderer::OnWaitStart

CBaseRenderer::WaitForRenderTime 方法在开始等待 filter 提交 sample 的时候会调用这个方法，

53CBaseRenderer::PrepareRender

当 Filter 在调用 **CBaseRenderer::OnReceiveFirstSample** 和 **CBaseRenderer::Render** 之前要调用这个方法，派生类可以在这个函数里处理一些事情。

54CBaseRenderer::ShouldDrawSampleNow

```
virtual HRESULT ShouldDrawSampleNow(  IMediaSample *pMediaSample,
    REFERENCE_TIME *pStartTime,  REFERENCE_TIME *pEndTime);
```

这个方法其实确定一个 sample 提交的时间。

CBaseRenderer::GetSampleTimes 会调用这个方法，缺省的情况下，samples 都是根据他们的时间戳来安排提交的顺序。

55CBaseRenderer::CheckMediaType

```
virtual HRESULT CheckMediaType( const CMediaType *pmt) PURE;
```

输入 pin 可以在 **CBasePin::CheckMediaType** 方法中调用这个函数，这个可以用来查询 filter 是否支持指定的媒体类型。

56CBaseRenderer::DoRenderSample

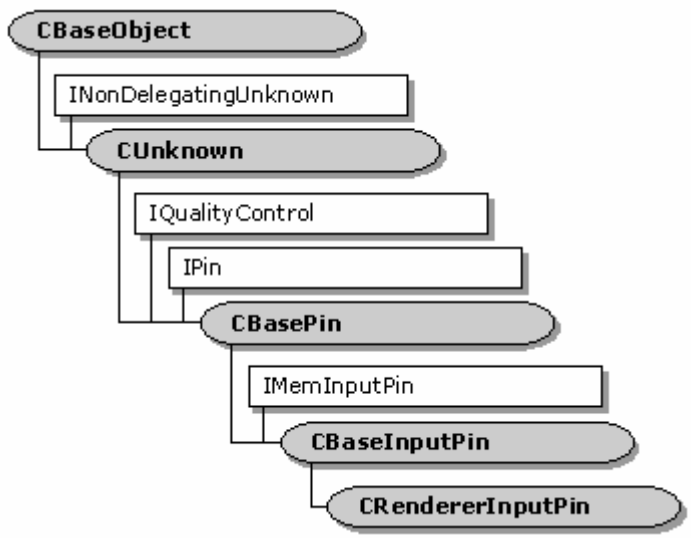
```
virtual HRESULT DoRenderSample(IMediaSample *pMediaSample) PURE;
```

派生类必须要重载这个函数，这个函数可说是这个实现 filter 的最重要的函数，根据不同功能的 filter，在这个函数有不同的实现，例如，在视频提交的 filter，应在这个函数里重画 sample 中包含的视频画面。

57CBaseRenderer::GetState
返回 filter 的状态，运行，暂停，停止。
58CBaseRenderer::Pause
派生类中这个函数要坐下下面的事情，
1 调用 **CBaseFilter::Pause 方法**
2 将 allocator 返回到内存池。（**IMemAllocator::Commit**.)）
3 如果暂停的前一个状态是停止，filter 就要释放它拥有的所有的 samples
4 调用 **CBaseRenderer::CompleteStateChange**
59CBaseRenderer::Run
派生类中这个函数要坐下下面的事情，
1 调用 **CBaseFilter::Pause 方法**
2 将 allocator 返回到内存池。（**IMemAllocator::Commit**.)）
3 如果运行的前一个状态是停止，filter 就要释放它拥有的所有的 samples
4 调用 **CBaseRenderer::StartStreaming** 方法。
60 CBaseRenderer::Stop
61 CBaseRenderer::FindPin

```
HRESULT FindPin(  LPCWSTR Id,  IPin **ppPin);
```

3.3.10CRendererInputPin



这个类是提交 filter 的上唯一的一个 pin，输入 pin 的基类，所以我特地把它请出来，瞧瞧它。这个类是从 CBaseInputPin 继承过来的，具有输入 pin 的一些共有的方法，有什么自己的特色呢，我们下面瞧瞧吧。
Ok，只有一个数据指向 filter 的一个指针

```
CBaseRenderer *m_pRenderer;
```

如果没有特别声明的话，the methods in this class delegate to corresponding methods on the **CBaseRenderer** class.。啥意思，我不懂。
1 CRendererInputPin::BreakConnect
这个方法继承了 **CBaseInputPin::BreakConnect**，
2 CRendererInputPin::CompleteConnect

完成和一个输出 pin 的连接，继承了 **CBasePin::CompleteConnect**

3 CRendererInputPin::CheckMediaType

4 CRendererInputPin::Active

用来告知 pin，filter 激活状态了

5 CRendererInputPin::Inactive

6CRendererInputPin::SetMediaType

7 CRendererInputPin::Allocator

```
IMemAllocator* Allocator(void) const;
```

这个函数直接将 **CBaseInputPin::m_pAllocator** 返回。

8 CRendererInputPin::QueryId

```
HRESULT QueryId(  LPWSTR *Id);
```

9 CRendererInputPin::EndOfStream

10 CRendererInputPin::BeginFlush

11 CRendererInputPin::EndFlush

12 CRendererInputPin::Receive

```
HRESULT Receive(  IMediaSample *pMediaSample);
```

这个方法将要调用 **CBaseRenderer::Receive** 方法用来处理数据，

其实看完这个类的所有方法，基本都是从 CBaseInputPin 过来的，没有实现新的方法，不知道什么意思这种类还有存在的价值吗？

下面我们看看视频提交 filter，叫 CBaseVideoRenderer

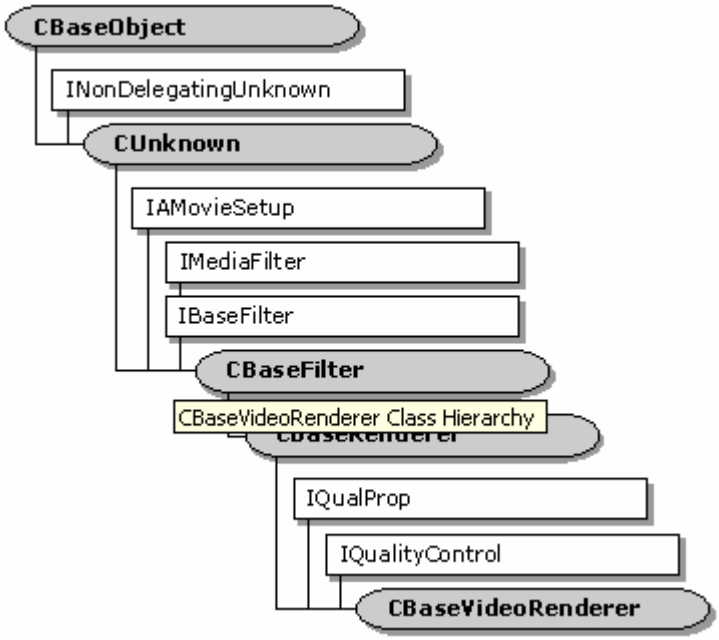
3.3.11CBaseVideoRenderer

```
class CBaseVideoRenderer : public CBaseRenderer,       // Base renderer class
                           public IQualProp,           // Property page guff
                           public IQualityControl      // Allow throttling
{
protected:

    int m_nNormal;                  // The number of consecutive frames
                                    // drawn at their normal time (not early)
                                    // -1 means we just dropped a frame.

#ifdef PERF
    BOOL m_bDrawLateFrames;         // Don't drop any frames (debug and I'm
                                    // not keen on people using it!)
#endif

    BOOL m_bSupplierHandlingQuality;// The response to Quality messages says

    int m_trThrottle;
    int m_trRenderAvg;              // Time frames are taking to blt
```

```
    int m_trRenderLast;             // Time for last frame blt
    int m_tRenderStart;             // Just before we started drawing (mSec)
                                        // derived from timeGetTime.

    int m_trEarliness;
    int m_trTarget;
    int m_trWaitAvg;                // Average of last few wait times
    int m_trFrameAvg;               // Average inter-frame time
    int m_trDuration;               // duration of last frame.

#ifdef PERF
    // Performance logging identifiers
    int m_idTimeStamp;              // MSR_id for frame time stamp
    int m_idEarliness;              // MSR_id for earliness fudge
    int m_idTarget;                 // MSR_id for Target fudge
    int m_idWaitReal;               // MSR_id for true wait time
    int m_idWait;                   // MSR_id for wait time recorded
    int m_idFrameAccuracy;          // MSR_id for time frame is late (int)
    int m_idRenderAvg;              // MSR_id for Render time recorded (int)
    int m_idSchLateTime;            // MSR_id for lateness at scheduler
    int m_idQualityRate;            // MSR_id for Quality rate requested
    int m_idQualityTime;            // MSR_id for Quality time requested
    int m_idDecision;               // MSR_id for decision code
    int m_idDuration;               // MSR_id for duration of a frame
    int m_idThrottle;               // MSR_id for audio-video throttling
    //int m_idDebug;                    // MSR_id for trace style debugging
    //int m_idSendQuality;          // MSR_id for timing the notifications per se
#endif // PERF
    REFERENCE_TIME m_trRememberStampForPerf;    // original time stamp of
frame
                                                // with no earliness fudges etc.
#ifdef PERF
    REFERENCE_TIME m_trRememberFrameForPerf;    // time when previous
frame rendered

    // debug...
    int m_idFrameAvg;
    int m_idWaitAvg;
#endif

    int m_cFramesDropped;
    int m_cFramesDrawn;
    LONGLONG m_iTotAcc;                     // Sum of accuracies in mSec
    LONGLONG m_iSumSqAcc;
```

```
    // Next two allow jitter calculation.    Jitter is std deviation of frame time.
    REFERENCE_TIME m_trLastDraw;
    LONGLONG m_iSumSqFrameTime;
    LONGLONG m_iSumFrameTime;
    int m_trLate;                          // hold onto frame lateness
    int m_trFrame;                         // hold onto inter-frame time

    int m_tStreamingStart;                 // if streaming then time streaming started
                                           // else time of last streaming session
                                           // used for property page statistics
#ifdef PERF
    LONGLONG m_llTimeOffset;               //
timeGetTime()*10000+m_llTimeOffset==ref time
#endif

public:
    CBaseVideoRenderer(REFCLSID RenderClass, // CLSID for this renderer
                       TCHAR *pName,         // Debug ONLY description
                       LPUNKNOWN pUnk,       // Aggregated owner object
                       HRESULT *phr);        // General OLE return code

    ~CBaseVideoRenderer();

    // IQualityControl methods - Notify allows audio-video throttling

    STDMETHODIMP SetSink( IQualityControl * piqc);
    STDMETHODIMP Notify( IBaseFilter * pSelf, Quality q);

    // These provide a full video quality management implementation

    void OnRenderStart(IMediaSample *pMediaSample);
    void OnRenderEnd(IMediaSample *pMediaSample);
    void OnWaitStart();
    void OnWaitEnd();
    HRESULT OnStartStreaming();
    HRESULT OnStopStreaming();
    void ThrottleWait();

    // Handle the statistics gathering for our quality management

    void PreparePerformanceData(int trLate, int trFrame);
    virtual void RecordFrameLateness(int trLate, int trFrame);
    virtual void OnDirectRender(IMediaSample *pMediaSample);
    virtual HRESULT ResetStreamingTimes();
```

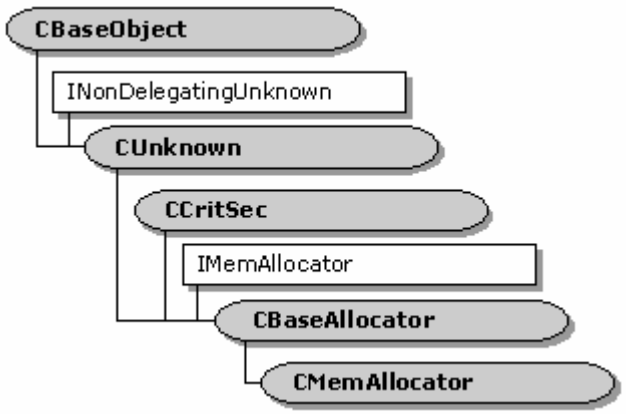
```
    BOOL ScheduleSample(IMediaSample *pMediaSample);
    HRESULT ShouldDrawSampleNow(IMediaSample *pMediaSample,
            REFERENCE_TIME *ptrStart,   REFERENCE_TIME *ptrEnd);

    virtual HRESULT SendQuality(REFERENCE_TIME trLate,
REFERENCE_TIME trRealStream);
    STDMETHODIMP JoinFilterGraph(IFilterGraph * pGraph, LPCWSTR pName);

    HRESULT GetStdDev( int nSamples, int *piResult, LONGLONG llSumSq,
        LONGLONG iTot    );
public:

    STDMETHODIMP get_FramesDroppedInRenderer(int *cFramesDropped);
    STDMETHODIMP get_FramesDrawn(int *pcFramesDrawn);
    STDMETHODIMP get_AvgFrameRate(int *piAvgFrameRate);
    STDMETHODIMP get_Jitter(int *piJitter);
    STDMETHODIMP get_AvgSyncOffset(int *piAvg);
    STDMETHODIMP get_DevSyncOffset(int *piDev);

    // Implement an IUnknown interface and expose IQualProp

    DECLARE_IUNKNOWN
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid,VOID **ppv);
};
```



这个类是用提交视频的，所以，有自己的特色。

我个人感觉，在这些基类中，除了 filter 类和 pin 类之外最重要的就是 sample 和 allocator 类了吧，还有一个参考时钟类，
这三个类和数据流密切相关，应该好好学习

3.3.12 CBaseAllocator



CBaseAllocator 用来实现一个 allocator 的一个抽象基类。Allocator 暴露了一个 IMemAllocator 接口。

一个 Allocator（内存分配器）是一个可以分配内存的对象。一个 allocator 保持了一系列可用的内存 buffer，也称为内存池，当客户（通常指一个 filter）请求一个 buffer，allocator 就从 burrer 列表中指定一个 buffer 给 filter 使用。客户将数据填充到 buffer，然后可能将 buffer 传递给其他的对象。最后，buffer 被释放，然后 Allocator 将这个不用的 buffer 返回到内存池，等待其他客户的请求。

每个 buffer 都被一个叫做 Media sample 的对象封装起来。Media sample 是 COM 框架内用来封装一个内存块的解决方案，Media sample 暴露了 IMediaSample 接口，并且由 **CMediaSample** 类来实现其接口。每一个 media sample 都包含了一个指针指向一块内存，可以通过 **IMediaSample::GetPointer** 方法来获得这个指针。
为了使用这个类，你要做下面的几步。
1 首先调用 **CBaseAllocator::SetProperties** 方法来指定内存的要求，包括内存的数量和大小
2 调用 **CBaseAllocator::Commit** 类分配内存
3 通过 **CBaseAllocator::GetBuffer** 得到 media samples，这个方法一直阻塞直到另一块 sample 可用，
4 当你使用完 sample 以后，记得调用 sample 上的 **IUnknown::Release** 来减少 sample 的引用计数，当一个 sample 的引用计数到了 0 时，sample 并不是被删除，而是返回到 allocator 内存池中，供其他的客户使用。
5 当你使用完 allocator 以后，记得要调用 **CBaseAllocator::Decommit** 来释放内存。

Commit 方法中调用了 **CBaseAllocator::Alloc**，用来分配 buffer 的内存，Decommit 调用了纯虚函数 **CBaseAllocator::Free**,用来释放内存。派生类一定要实现这两个方法。
The **CMemAllocator** 类从 **CBaseAllocator 派生，** filter 的基类都使用 **CMemAllocator** class.
下面分析数据成员
CSampleList m_lFree; 指向可用的 media sample 列表
HANDLE m_hSem;
long m_lWaiting 等待 sample 的线程的数量

long m_lCount; 可提供的 buffer 的数量
long m_lAllocated; 当前以经分配的内存的数量
long m_lSize; 每一个 buffer 的大小
long m_lAlignment;
long m_lPrefix;
BOOL m_bChanged; 用来标示 buffer 要求是否改变过的标志
BOOL m_bCommitted; 用来标志 allocator 是否已经 committed
BOOL m_bDecommitInProgress;
IMemAllocatorNotifyCallbackTemp *m_pNotify; 回调接口 realse 被调用
BOOL m_fEnableReleaseCallback;标志上面的功能是否可用

下面分析一下成员函数

1 CBaseAllocator::Alloc

```
virtual HRESULT Alloc(void);
```

这个方法会在 **CBaseAllocator::Commit** 中调用，在基类中这个方法并没有分配内存，它会在 buffer 请求没有被设置时返回一个 error，如果要求没有被改变，就返回一个 false，如果要求改变，返回 ok。

一个派生类应该实现这个方法实际的分配内存，一般的，派生类做如下的事情

1 调用基类的方法，来确定是否真的需要分配内存

2Allocator 内存

3 创建一个 CMediaSample 对象包含上一步申请的内存

4 将 CMediaSample 对象添加到 **CBaseAllocator::m_lFree**

2 CBaseAllocator::SetNotify

```
HRESULT SetNotify( IMemAllocatorNotifyCallbackTemp *pNotify);
```

这个方法用来设置或者取消一个 allocator 的回调函数。当一个 allocator 上调用 **IMemAllocator::ReleaseBuffer** 方法的时候会调用一个回调函数。

这个方法实现了 **IMemAllocatorCallbackTemp::SetNotify** 方法，这个 allocator 只有在构造函数中将 *fEnableReleaseCallback* 标志设置为 TRUE 的时候，才暴露出 **IMemAllocatorNotifyCallbackTemp 接口**，

3 CBaseAllocator::GetFreeCount

```
HRESULT GetFreeCount( LONG *plBuffersFree);
```

这个方法实现了 **IMemAllocatorCallbackTemp::GetFreeCount** 接口方法，

4 CBaseAllocator::NotifySample

当有几个线程在等待 samples 的时候，**CBaseAllocator::m_lWaiting** 的值就大于 0，如果 m_lWaiting 大于 0 的时候，这个方法就会调用 **CBaseAllocator::m_hSem** 上的 ReleaseSeamphore 方法，

5 CBaseAllocator::SetWaiting

6 CBaseAllocator::Free

当 **Decommit** 方法调用的时候，allocator 调用这个方法。

7 CBaseAllocator::SetProperties

```
HRESULT SetProperties( ALLOCATOR_PROPERTIES *pRequest,
    ALLOCATOR_PROPERTIES *pActual);
```

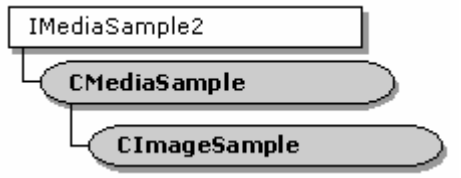
8 CBaseAllocator::GetProperties

9 CBaseAllocator::Commit
在调用这个方法之前，要调用 **CBaseAllocator::SetProperties** 方法来设置 buffer 的请求。这个方法调用 **CBaseAllocator::Alloc** 来分配内存，
注意，一定要在调用 **CBaseAllocator::GetBuffer** 调用这个方法。
10 CBaseAllocator::Decommit
当调用这个方法以后，**CBaseAllocator::GetBuffer** 方法就会失败。
11 CBaseAllocator::GetBuffer

```
HRESULT GetBuffer(IMediaSample **ppBuffer,
    REFERENCE_TIME *pStartTime, REFERENCE_TIME *pEndTime, DWORD dwFlags);
```

这个函数用来返回一个指向包含 buffer 的 sample 指针。
12 CBaseAllocator::ReleaseBuffer

3.3.13 CMediaSample



Media sample 包含一个指向内存的指针，和一些作为私有变量的属性。
Media sample 是由 allocators 创建的，**CMediaSample** 构造函数接受一个内存指针，其他的属性由 **IMediaSample** 接口设置。
Media sample 的生命周期和其他的 com 对象有点区别。
1 allocator 保持着 samples 的 list，当一个 filter 需要一个新的 sample，它就调用 **IMemAllocator::GetBuffer** 方法，allocator 从它的 list 中的返回一个空闲的 sample，增加 sample 的引用计数，然后将一个指针返回给 sample
2 当一个 filter 使用完一个 sample 以后，它会调用 sample 上的 IUnkown::Release 方法减少引用计数，然后调用 allocator 上的 **IMemAllocator::ReleaseBuffer** 方法，将 sample 返回给 allocator 上的 list。
3allocator 只有调用 **IMemAllocator::Decommit** 才会销毁 samples。
下面分析一下 sample 的数据成员

```
DWORD m_dwFlags;
DWORD m_dwTypeSpecificFlags;
LPBYTE m_pBuffer;
LONG m_lActual;
LONG m_cbBuffer;
CBaseAllocator *m_pAllocator; 指向创建这个 sample 的 allocator 指针
CMediaSample *m_pNext; 指向 allocator 里 list 上下一个 samples
REFERENCE_TIME m_Start;
REFERENCE_TIME m_End;
LONGLONG m_MediaStart;
LONG m_MediaEnd;
```

```
AM_MEDIA_TYPE *m_pMediaType;
DWORD m_dwStreamId;
```

下面分析成员函数吧

1 CMediaSample::GetPointer

```
HRESULT GetPointer( BYTE **ppBuffer);
```

这个方法用来返回 sample 中的 buffer 的指针。

2 CMediaSample::GetSize

返回 sample 的 buffer 的大小。

3 CMediaSample::GetTime

```
HRESULT GetTime( REFERENCE_TIME *pTimeStart,REFERENCE_TIME *pTimeEnd);
```

这个方法用来返回 sample 上的时间戳

4 CMediaSample::SetTime

```
HRESULT SetTime(REFERENCE_TIME *pTimeStart,REFERENCE_TIME *pTimeEnd);
```

设置 sample 的时间戳

5 CMediaSample::IsSyncPoint

这个函数用来判断 sample 的开始处是否是一个同步点。

6 CMediaSample::SetSyncPoint

将这个 sample 的开端设置成同步点

7 CMediaSample::IsPreroll

8 CMediaSample::SetPreroll

9 CMediaSample::GetActualDataLength

返回 buffer 里的合法数据的长度。

10 CMediaSample::SetActualDataLength

11 CMediaSample::GetMediaType

```
HRESULT GetMediaType( AM_MEDIA_TYPE **ppMediaType);
```

12 CMediaSample::SetMediaType

```
HRESULT SetMediaType( AM_MEDIA_TYPE *pMediaType);
```

13 CMediaSample::IsDiscontinuity

```
HRESULT IsDiscontinuity(void);
```

14 CMediaSample::SetDiscontinuity

15 CMediaSample::GetMediaTime

```
HRESULT GetMediaTime(  LONGLONG *pStart, LONGLONG *pEnd);
```

用来设置 media time。

16 CMediaSample::SetMediaTime

注意，sample 中有两个时间，stream time 和 media，如果想知道两个时间的区别，请参考第一章的 1.8 小节。

17 CMediaSample::GetProperties

18CMediaSample::SetProperties

用来设置或者获取 sample 的属性。

3.4Filter 和 pin 经常用到的类

3.4.1CPullPin

```
class CPullPin : public CAMThread
{
    IAsyncReader*           m_pReader;
    REFERENCE_TIME          m_tStart;
    REFERENCE_TIME          m_tStop;
    REFERENCE_TIME          m_tDuration;
    BOOL                    m_bSync;

    enum ThreadMsg {
    TM_Pause,           // stop pulling and wait for next message
    TM_Start,           // start pulling
    TM_Exit,            // stop and exit
    };

    ThreadMsg m_State;

    // override pure thread proc from CAMThread
    DWORD ThreadProc(void);

    // running pull method (check m_bSync)
    void Process(void);

    // clean up any cancelled i/o after a flush
    void CleanupCancelled(void);

    // suspend thread from pulling, eg during seek
    HRESULT PauseThread();

    // start thread pulling - create thread if necy
    HRESULT StartThread();

    // stop and close thread
    HRESULT StopThread();

    // called from ProcessAsync to queue and collect requests
    HRESULT QueueSample(
        REFERENCE_TIME& tCurrent,
        REFERENCE_TIME tAlignStop,
        BOOL bDiscontinuity);
```

```
        HRESULT CollectAndDeliver(
            REFERENCE_TIME tStart,
            REFERENCE_TIME tStop);

        HRESULT DeliverSample(
            IMediaSample* pSample,
            REFERENCE_TIME tStart,
            REFERENCE_TIME tStop);

protected:
        IMemAllocator *         m_pAlloc;

public:
        CPullPin();
        virtual ~CPullPin();

        HRESULT Connect(IUnknown* pUnk, IMemAllocator* pAlloc, BOOL bSync);

        // disconnect any connection made in Connect
        HRESULT Disconnect();

        virtual HRESULT DecideAllocator(
            IMemAllocator* pAlloc,
            ALLOCATOR_PROPERTIES * pProps);

        HRESULT Seek(REFERENCE_TIME tStart, REFERENCE_TIME tStop);

        // return the total duration
        HRESULT Duration(REFERENCE_TIME* ptDuration);

        // start pulling data
        HRESULT Active(void);

        // stop pulling data
        HRESULT Inactive(void);

        // helper functions
        LONGLONG AlignDown(LONGLONG ll, LONG lAlign) {
       // aligning downwards is just truncation
       return ll & ~(lAlign-1);
        };

        LONGLONG AlignUp(LONGLONG ll, LONG lAlign) {
```

```
    // align up: round up to next boundary
    return (ll + (lAlign -1)) & ~(lAlign -1);
     };

     IAsyncReader* GetReader() {
    m_pReader->AddRef();
    return m_pReader;
     };

     virtual HRESULT Receive(IMediaSample*) PURE;

     // override this to handle end-of-stream
     virtual HRESULT EndOfStream(void) PURE;

     virtual void OnError(HRESULT hr) PURE;

     // flush this pin and all downstream
     virtual HRESULT BeginFlush() PURE;
     virtual HRESULT EndFlush() PURE;

};
```

3.4.2COutputQueue

3.4.3CSourceSeeking

3.4.4CEnumPins

3.4.5CEnumMediaTypes

3.4.6CMemAllocator

3.4.7CMediaSample

```
class CMediaSample : public IMediaSample2       // The interface we support
{
```

```
protected:

    friend class CBaseAllocator;

    enum { Sample_SyncPoint            = 0x01,     /* Is this a sync point */
           Sample_Preroll              = 0x02,     /* Is this a preroll sample */
           Sample_Discontinuity        = 0x04,     /* Set if start of new segment */
           Sample_TypeChanged          = 0x08,     /* Has the type changed */
           Sample_TimeValid            = 0x10,     /* Set if time is valid */
           Sample_MediaTimeValid       = 0x20,     /* Is the media time valid */
           Sample_TimeDiscontinuity = 0x40, /* Time discontinuity */
           Sample_StopValid            = 0x100,    /* Stop time valid */
           Sample_ValidFlags           = 0x1FF
         };

    DWORD             m_dwFlags;            /* Flags for this sample */
                                            /* Type specific flags are packed
                                               into the top word
                                            */
    DWORD             m_dwTypeSpecificFlags; /* Media type specific flags */
    LPBYTE            m_pBuffer;            /* Pointer to the complete buffer */
    LONG              m_lActual;            /* Length of data in this sample */
    LONG              m_cbBuffer;           /* Size of the buffer */
    CBaseAllocator    *m_pAllocator;        /* The allocator who owns us */
    CMediaSample      *m_pNext;             /* Chaining in free list */
    REFERENCE_TIME    m_Start;              /* Start sample time */
    REFERENCE_TIME    m_End;                /* End sample time */
    LONGLONG          m_MediaStart;         /* Real media start position */
    LONG              m_MediaEnd;           /* A difference to get the end */
    AM_MEDIA_TYPE     *m_pMediaType;        /* Media type change data */
    DWORD             m_dwStreamId;         /* Stream id */
public:
    LONG              m_cRef;               /* Reference count */


public:

    CMediaSample( TCHAR *pName,     CBaseAllocator *pAllocator,
        HRESULT *phr, LPBYTE pBuffer = NULL, LONG length = 0);
#ifdef UNICODE
    CMediaSample(   CHAR *pName,    CBaseAllocator *pAllocator,
        HRESULT *phr,   LPBYTE pBuffer = NULL, LONG length = 0);
#endif
```

```
virtual ~CMediaSample();

/* Note the media sample does not delegate to its owner */

STDMETHODIMP QueryInterface(REFIID riid, void **ppv);
STDMETHODIMP_(ULONG) AddRef();
STDMETHODIMP_(ULONG) Release();

HRESULT SetPointer(BYTE * ptr, LONG cBytes);

// Get me a read/write pointer to this buffer's memory.
STDMETHODIMP GetPointer(BYTE ** ppBuffer);

STDMETHODIMP_(LONG) GetSize(void);

// get the stream time at which this sample should start and finish.
STDMETHODIMP GetTime(
    REFERENCE_TIME * pTimeStart,     // put time here
    REFERENCE_TIME * pTimeEnd
);

// Set the stream time at which this sample should start and finish.
STDMETHODIMP SetTime(
    REFERENCE_TIME * pTimeStart,     // put time here
    REFERENCE_TIME * pTimeEnd
);
STDMETHODIMP IsSyncPoint(void);
STDMETHODIMP SetSyncPoint(BOOL bIsSyncPoint);
STDMETHODIMP IsPreroll(void);
STDMETHODIMP SetPreroll(BOOL bIsPreroll);

STDMETHODIMP_(LONG) GetActualDataLength(void);
STDMETHODIMP SetActualDataLength(LONG lActual);

// these allow for limited format changes in band

STDMETHODIMP GetMediaType(AM_MEDIA_TYPE **ppMediaType);
STDMETHODIMP SetMediaType(AM_MEDIA_TYPE *pMediaType);

STDMETHODIMP IsDiscontinuity(void);
STDMETHODIMP SetDiscontinuity(BOOL bDiscontinuity);

// get the media times for this sample
```

```
    STDMETHODIMP GetMediaTime(  LONGLONG * pTimeStart,
    LONGLONG * pTimeEnd   );

    // Set the media times for this sample
    STDMETHODIMP SetMediaTime(  LONGLONG * pTimeStart,
    LONGLONG * pTimeEnd   );

    // Set and get properties (IMediaSample2)
    STDMETHODIMP GetProperties(       DWORD cbProperties,
        BYTE * pbProperties     );

    STDMETHODIMP SetProperties(       DWORD cbProperties,
        const BYTE * pbProperties   );
};
```

3.4.8CBaseReferenceClock

3.4.9CMediaType

3.5 几个比较重要的类

4Direcshow 提供的接口学习

5DirectShow Tutorials

5.1Implementing a Seek Bar

5.2Displaying a Filter's Property Pages

5.3Grabbing a Poster Frame

5.4Using the Sample Grabber

5.5Recompressing an AVI File

6C++在电视开发中的应用

6.1TV Ratings Reference

6.2Video Control C++ Reference

6.3Microsoft Unified Tuning Model C++ Reference

6.4Transport Information Interfaces

6.5BDA Filter Interfaces

6.6MPEG-2 Sections and Tables Filter Reference

7Direcshow 提供的 Filter