傲瑞组件开发手册

傲瑞科技

www.oraycn.com

武 汉 傲 瑞 科 技 有 限 公 司

Wuhan Oray Technology Co.,Ltd.

http://www.oraycn.com

2017 年 04 月

（01）——语音视频录制 MFile

在很多语音视频软件系统中，经常有将实时的音频或视频录制为文件保存到磁盘的需求，比如，视频监控系统中录制监控到的视频、视频会议系统中录制整个会议的过程、语音通话系统中录制完整的对话内容、电脑桌面录制、等等。

MFile 组件（Oraycn.MFile.dll）是傲瑞实用组件之一，它可以将原始的语音数据和视频数据按照指定的格式进行编码，并将它们写入到视频文件（如.mp4）中。

一.MFile 简介

MFile 组件内部的核心技术包括以下 4 点：

（1）音频数据编码。

（2）视频数据编码。

（3）将编码后的数据按文件格式的要求写入到文件容器中。

（4）保证音频视频播放的同步。

使用 MFile 有三种模式可供选择：

（1）生成音频文件（.mp3）：声音编码格式为 MP3。

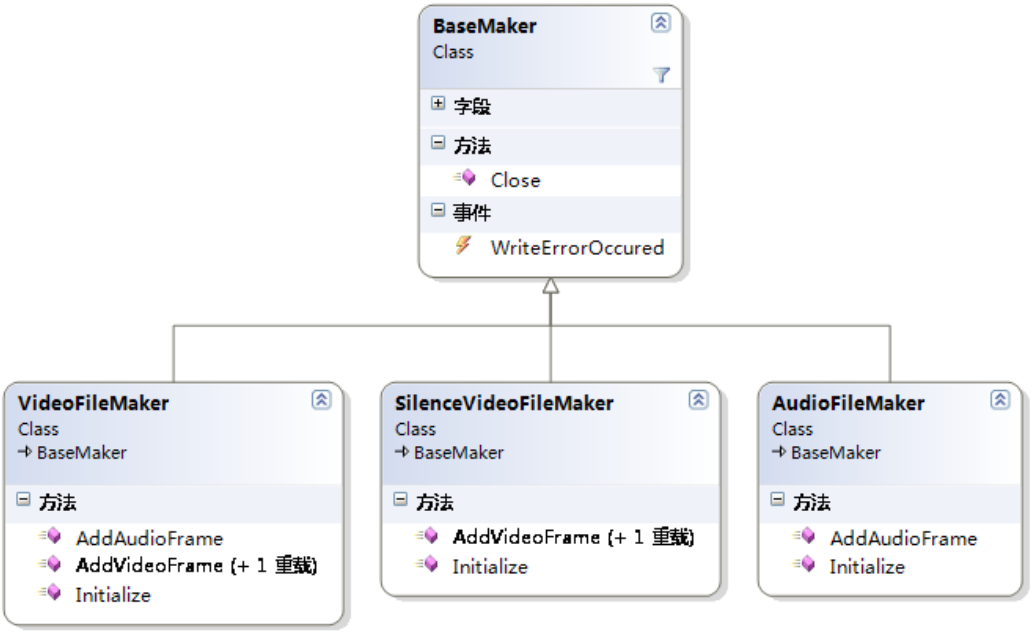
（2）生成无声的视频文件（.mp4）：视频编码格式为 H264。

（3）生成普通视频的文件（.mp4）：声音编码格式为 AAC，视频编码格式为 H264。

生成的这些文件，可以直接使用我们常见的播放器进行播放。

二.MFile 结构

对于使用者而言，MFile 组件中的主要类的结构图如下所示：



其中，AudioFileMaker 用于生成音频文件、SilenceVideoFileMaker 用于生成无声的视频文件、而 VideoFileMaker 用于生成既有声音又有图像的普通视频文件。这三个类都从基类 BaseMaker 继承，它们的使用方式也是一致的。接下来，我们仅仅详细讲解 VideoFileMaker 类的使用，SilenceVideoFileMaker 和 AudioFileMaker 的使用方法可以类推之。

三.VideoFileMaker 类

下面是 VideoFileMaker 类的 public 方法的签名：

API：

```
// 摘要:
//      将视音频数据（raw data）、频帧数据（raw data）按指定的格式进行编码并生成视频文件。 采用独立的后台线程将音频/视频帧写入文件。
public class VideoFileMaker : BaseMaker
{
    public VideoFileMaker();

    // 摘要:
    //      当通过AddVideoFrame方法传入视频帧Bitmap被写入文件后，自动释放该视频帧？默认值为true。
    public bool AutoDisposeVideoFrame { get; set; }

    // 摘要:
    //      添加音频帧。
    public void AddAudioFrame(byte[] audioframe);
    //
    // 摘要:
    //      添加视频帧。如果autoSyncToAudio开启，则自动同步到音频。
    public void AddVideoFrame(Bitmap frame);
    //
    // 摘要:
    //      添加视频帧。
    //
    // 参数:
    //    frame:
    //      视频帧
    //
    //    timeStamp:
    //      离开始时的时间长度
    public void AddVideoFrame(Bitmap frame, TimeSpan timeStamp);
    public void Flush();
    //
    // 摘要:
    //      初始化视频文件。
    //
    // 参数:
    //    filePath:
    //      文件路径
    //
    //    videoWidth:
    //      视频宽度
    //
    //    videoHeight:
    //      视频高度
    //
    //    videoFrameRate:
    //      帧频
```

```
        //
        //     audioSampleRate:
        //         音频采样率。【注：采样位数必须为16位】
        //
        //     audioChannelCount:
        //         声道数
        public void Initialize(string filePath, int videoWidth, int videoHeight, int videoFrameRate, int audioSampleRate,
int audioChannelCount);
        //
        // 摘要:
        //         初始化视频文件。
        //
        // 参数:
        //     filePath:
        //         文件路径
        //
        //     videoWidth:
        //         视频宽度
        //
        //     videoHeight:
        //         视频高度
        //
        //     videoFrameRate:
        //         帧频
        //
        //     videoQuality:
        //         录制生成视频的清晰度
        //
        //     audioSampleRate:
        //         音频采样率。【注：采样位数必须为16位】
        //
        //     audioChannelCount:
        //         声道数
        public void Initialize(string filePath, int videoWidth, int videoHeight, int videoFrameRate, VideoQuality
videoQuality, int audioSampleRate, int audioChannelCount);
        //
        // 摘要:
        //         初始化视频文件。
        //
        // 参数:
        //     filePath:
        //         文件路径
        //
        //     videoWidth:
        //         视频宽度
        //
        //     videoHeight:
        //         视频高度
        //
```

```
        //     videoFrameRate:
        //       帧频
        //
        //     videoQuality:
        //       录制生成视频的清晰度
        //
        //     audioSampleRate:
        //       音频采样率。【注：采样位数必须为16位】
        //
        //     audioChannelCount:
        //       声道数
        //
        //     autoSyncToAudio:
        //       是否以音频为基准进行同步。比如实时录制，则可以考虑传入true。
        public void Initialize(string filePath, int videoWidth, int videoHeight, int videoFrameRate, VideoQuality
videoQuality, int audioSampleRate, int audioChannelCount, bool autoSyncToAudio);
        //
        // 摘要:
        //       初始化视频文件。
        //
        // 参数:
        //     filePath:
        //       文件路径
        //
        //     videoWidth:
        //       视频宽度
        //
        //     videoHeight:
        //       视频高度
        //
        //     videoFrameRate:
        //       帧频
        //
        //     videoQuality:
        //       录制生成视频的清晰度
        //
        //     audioSampleRate:
        //       音频采样率。【注：采样位数必须为16位】
        //
        //     audioChannelCount:
        //       声道数
        //
        //     autoSyncToAudio:
        //       是否以音频为基准进行同步。比如实时录制，则可以考虑传入true。
        public void Initialize(string filePath, VideoCodecType videoCodec, int videoWidth, int videoHeight, int
videoFrameRate, VideoQuality videoQuality, AudioCodecType audioCodec, int audioSampleRate, int audioChannelCount,
bool autoSyncToAudio);
    }
```

（1）Initialize 方法

Initialize 方法传入了生成视频文件时所需要的所有参数。它有一个重载方法，两个方法的差别在于 videoQuality 参数，该参数用于控制录制生成的视频的清晰度。

MFile 要求录制的视频帧的长和宽（videoWidth 和 videoHeight）必须是 4 的整数倍。如果图像帧不满足这一要求，则应先对图像帧进行裁剪。（MFileDemo 中有相应的操作）

Initialize 如果执行失败（比如，参数设置错误），其将会抛出相应的异常。

Initialize 成功调用后，VideoFileMaker 会启动一个独立的后台线程，该线程的职责就是将后面加入的音频帧、视频帧异步写入到文件中。

（2）AddAudioFrame 方法

用于向文件中写入音频数据，至于参数 audioframe 的字节长度，AddAudioFrame 方法并没有任何限制。也就是说，可以一次写入 10ms 的数据，也可一次写入 100ms 的数据。但是，MFile 要求音频采集的位宽必须为 16bit。

另外要注意，audioframe 的字节长度必须与 Initialize 方法传入的 audioSampleRate 参数和 audioChannelCount 参数保持一致的关系。比如，采样率 16k、采样位数 16bit、声道数 1，那么一个 10ms 的音频帧的大小为（16000*16*1*0.01）/8=320 字节。也就是说，在这种情况下，如果参数 audioframe 的长度为 320 字节，就表示其为 10ms 的数据，如果为 640 字节，就表示其为 20ms 的数据。

如果需要录制多路语音数据，那么在调用 AddAudioFrame 方法之前，必须先自行将多路语音数据混音成一路，MFile 没有提供混音的功能。（如果您使用了我们的 OMCS 语音视频框架，那么，OMCS 有内置的混音功能可以直接使用）

（3）AddVideoFrame 方法

AddVideoFrame 方法的参数直接是一个位图，表示一个视频帧，很容易理解。如果需要手动控制每一视频帧写入的时间戳，那么可以调用带两个参数的重载方法。

（4）Close 方法

当不再有新的视频帧和音频帧写入时，可以调用 Close 方法，以完成文件的生成。但是，由于实际的写文件操作是在一个独立的后台线程中进行的，在某些情况下，在调用 Close 的时候，可能后台线程还正在忙碌（比如，在一些比较慢的机器上面，实时录制视频时，消费的速度跟不上生产的速度，便会出现这种情况），那么，Close 方法的参数 waitFinished 就用于指示是否等待后台线程将所有帧写入线程，如果等待发生，Close 调用将被阻塞，直至后台线程工作完成，Close 才会返回。

（5）WriteErrorOccured 事件

当当后台写线程在向文件写入视频帧或音频帧时，如果发生任何错误，将会触发 WriteErrorOccured 事件，并且，结束写线程。

四.MFileDemo

下面我们使用一个 demo 来介绍如何使用 MFile 组件，在这个 demo 中，我们借助语音视频采集组件 MCapture 采集来自麦克风输入的音频数据、以及来自摄像头采集的视频数据，并将它们录制生成 mp4 文件。Demo 运行的截图如下所示：

首先，当点击启动设备按钮时，我们创建一个摄像头采集器实例和一个麦克风采集器实例，并启动它们开始采集：

API：

```
this.cameraCapturer = CapturerFactory.CreateCameraCapturer(0, new Size(int.Parse(this.textBox_width.Text), int.Parse(this.textBox_height.Text)), this.fps);
this.cameraCapturer.ImageCaptured += new CbGeneric<Bitmap>(cameraCapturer_ImageCaptured);
this.cameraCapturer.CaptureError += new CbGeneric<Exception>(cameraCapturer_CaptureError);
this.microphoneCapturer = CapturerFactory.CreateMicrophoneCapturer(0);
this.microphoneCapturer.AudioCaptured += new CbGeneric<byte[]>(microphoneCapturer_AudioCaptured);
this.microphoneCapturer.CaptureError += new CbGeneric<Exception>(microphoneCapturer_CaptureError);
//开始采集
this.cameraCapturer.Start();
this.microphoneCapturer.Start();
```

接下来，点击开始录制按钮时，我们初始化 VideoFileMaker 组件：

API：

```
this.videoFileMaker = new VideoFileMaker();
this.videoFileMaker.AutoDisposeVideoFrame = true;
this.videoFileMaker.Initialize("test.mp4", VideoCodecType.H264, int.Parse(this.textBox_width.Text), int.Parse(this.textBox_height.Text), this.fps,
                                AudioCodecType.AAC, 16000, 1, true);
this.isRecording = true;
```

参数中设定，使用 h.264 对视频进行编码，使用 aac 对音频进行编码，并生成 mp4 格式的文件。然后，我们可以通过 OMCS 获取实时的音频数据和视频数据，并将它们写到文件中。

API：

```
void microphoneCapturer_AudioCaptured(byte[] audioData) //采集到的语音数据
{
    if (this.isRecording)
    {
        this.videoFileMaker.AddAudioFrame(audioData);
    }
}
//采集到的视频图像
void cameraCapturer_ImageCaptured(Bitmap img)
```

```
    {
        if (this.isRecording)
        {
            this.DisplayVideo((Bitmap)img.Clone());
            this.videoFileMaker.AddVideoFrame(img);
        }
        else
        {
            this.DisplayVideo(img);
        }
    }
```

当想结束录制时，则调用 Close 方法：

API：

```
this.videoFileMaker.Close(true);
```

Demo 源码：Oraycn.RecordDemo.rar

五．录制方案推荐：

（1）采集与录制最完整的 Demo，五星推荐：WinForm 版本、WPF 版本

（2）远程录制/在服务端录制语音、视频、桌面：MFile + OMCS

（3）录制双方的视频聊天全过程：MFile + OMCS

（4）仅仅录制声卡：MFile + MCapture

（5）采集麦克风和声卡，混音之后进行录制：MFile+MCapture

（02）——轻量级的通信引擎 StriveEngine

如果 ESFramework 对您的项目来说，太庞大、太重量级；如果您的项目不需要 P2P、不需要传文件、不需要群集等功能，只需要简单的 TCP/UDP 通信。那么，可以考虑使用轻量级的通信引擎 StriveEngine。相比较而言，StriveEngine 更单纯、更容易上手，也更容易与已存在的遗留系统进行协作。

一.StriveEngine 主要特性

01.底层采用高效的 IOCP（完成端口）模型。

02.内部自动管理可复用的线程池、以及内存池。

03.内置多种通信引擎类型：TCP/UDP、文本协议/二进制协议、服务端/客户端。而且，所有这些引擎的使用方式一致。

04.解决了 TCP 通信中的粘包以及消息重组问题。

05.发送消息支持同步、异步两种方式。

06.服务端引擎支持异步消息队列模式。

07.客户端 TCP 引擎支持断线自动重连。

08.支持 HTML5WebSockets。

09.提供了 Unity3D 客户端引擎、WinCE 客户端引擎。

10.支持 Sock5 代理。

11.兼容 IPv6。

二.StriveEngine 与 ESFramework/ESPlus/ESPlatform 的区别

1. ESFramework 是一个功能强大的通信框架/平台，而 StriveEngine 是一个单纯高效的通信引擎类库。

2.ESFramework 更贴近应用层（比如支持在线用户管理、文件传送、P2P 通道、好友与组、群组广播、服务器群集、性能诊断等），而 StriveEngine 更贴近 Socket 层。

3.ESFramework 使用 UserID 标志每一个在线的客户端，而 StriveEngine 则使用低阶的 IPEndPoint 来标志每一个在线的客户端。

4.可以认为 StriveEngine 就是 ESFramework 底层使用的内核。

5.ESFramework 定义了底层通信消息的格式（对 ESFramework 使用者是透明的），而 StriveEngine 对通信消息的格式未作任何定义。

6.ESFramework 和 StriveEngine 都提供了服务端引擎和客户端引擎。当涉及到要开发新的其它平台引擎来与 ESFramework 或 StriveEngine 协作时，比如：

（1）如果要开发其它平台的客户端引擎与 ESFramework 的服务端引擎协作，则必需实现 ESPlus 协议格式；

（2）如果要开发其它平台的客户端引擎与 StriveEngine 的服务端引擎协作，由于 StriveEngine 未定义任何消息格式，所以不存在要必需实现既有协议格式的问题。

（3）与（2）同理，使用 StriveEngine 的客户端引擎，可以与任何其它现有的服务端协作。

7.对于那些涉及到在线用户管理、以 UserID 为中心的应用（比如即时通信应用），或者需要 P2P 通信、传文件等功能的应用来说，ESFramework 更适合；

对于那些仅仅需要简单而高效的通信功能的应用（如数据采集、消息转发等）来说，StriveEngine 更合适。

三.StriveEngine 整体结构

StriveEngine 内置了多种通信引擎，这些通信引擎都以接口的方式暴露了其功能，而不同的引擎却都依据某些共通的部分，继承了相同的基接口。引擎接口的继承关系如下所示：

INetworkEngine 是所有引擎的根接口，只要是 StriveEngine 中的内置引擎，都实现了这个接口。

IServerEngine 是服务端引擎接口。IPassiveEngine 是客户端引擎接口。

ITcpServerEngine 是基于 TCP 的服务端引擎接口。ITcpPassiveEngine 是基于 TCP 的客户端引擎接口。

IUdpEngine 是基于 UDP 的通信引擎接口。由于 UDP 是非连接协议，可以认为 UDP 通信的双方是对等的，所以，服务端和客户端都使用 IUdpEngine 即可。

ITextEngine 是基于文本协议的通信引擎的基接口。

下面，我们来详细看看每个接口。

1.InetworkEngine

API：

```
/// <summary>
/// 所有网络引擎（包括服务端引擎、客户端引擎、TCP引擎、UDP引擎、二进制引擎、文本引擎）的基础接口。
/// </summary>
public interface INetworkEngine : IDisposable
{
    /// <summary>
    /// 传输层协议类型，TCP或UDP
    /// </summary>
    ProtocolType ProtocolType { get; }

    /// <summary>
    /// 引擎实例的创建时间。
    /// </summary>
    DateTime CreateTime { get; }

    /// <summary>
    /// 表示要监听本地的哪个端口。如果设定其值小于等于0，则表示由系统自动分配。
    /// 注意，引擎初始化完成后，不能再修改该属性。
    /// </summary>
    int Port { get; set; }

    /// <summary>
    /// 对于服务端引擎，表示要绑定本地的哪个IP，如果设为null，则表示绑定本地所有IP。
    /// 对于客户端引擎，表示要绑定本地的哪个IP与服务器进行通信。如果设为null（其值会在初始化完成后被修改），则自动选择地址列表中的某一个。
    /// 注意，引擎初始化完成后，不能再修改该属性。
    /// </summary>
    string IPAddress { get; set; }

    /// <summary>
    /// 设置日志记录的文件路径。如果设为null，表示不记录日志。默认值为null。
    /// </summary>
    string LogFilePath { set; }

    /// <summary>
    /// Socket(网卡)发送缓冲区的大小。默认为8k。
    /// </summary>
    int SocketSendBuffSize { get; set; }

    /// <summary>
    /// Socket(网卡)接收缓冲区的大小。默认为8k。
```

```
        /// </summary>
        int SocketReceiveBuffSize { get; set; }

        /// <summary>
        /// 网络引擎能够接收的最大的消息尺寸。据此网络引擎可以为每个Session/Connection开辟适当大小的接收缓冲区。
        /// 默认为1k。当接收到的消息尺寸超过MaxMessageSize时，将会关闭对应的连接（对TCP）或丢弃数据（对UDP）。
        /// </summary>
        int MaxMessageSize { get; set; }

        /// <summary>
        /// 引擎实例是否已经被释放。
        /// </summary>
        bool Disposed { get; }

        /// <summary>
        /// 初始化并启动网络引擎。如果修改了引擎配置参数，在应用新参数之前必须先重新调用该方法初始化引擎。
        /// </summary>
        void Initialize();

        /// <summary>
        /// 当不再使用当前引擎实例时，释放它。（异步方式）
        /// 注意：对于UDP或客户端引擎，如果消息是同步处理的(HandleMessageAsynchronismly为false)，请不要在消息处理器中调用Dispose方法，否则，会导致死锁。
        /// 因为停止引擎要等到最后一条消息处理完毕才会返回。可以转向调用异步的DisposeAsyn方法。
        /// </summary>
        void DisposeAsyn();

        /// <summary>
        /// 当引擎实例被释放时，触发此事件。
        /// </summary>
        event CbDelegate<INetworkEngine> EngineDisposed;

        /// <summary>
        /// 接收到一个完整的消息时触发该事件。
        /// </summary>
        event CbDelegate<IPEndPoint, byte[]> MessageReceived;

        /// <summary>
        /// 将消息成功发送之后触发该事件
        /// </summary>
        event CbDelegate<IPEndPoint, byte[]> MessageSent;
    }

    /// <summary>
    /// 传输层协议类型。
    /// </summary>
```

```
public enum ProtocolType
{
    TCP = 0,
    UDP
}

/// <summary>
/// 协议的格式。
/// </summary>
public enum ContractFormatStyle
{
    /// <summary>
    /// 流协议或称二进制协议。
    /// </summary>
    Stream = 0,
    /// <summary>
    /// 文本协议，如基于xml的。
    /// </summary>
    Text
}
```

（1）StriveEngine 中的所有通信引擎在设置完必要的属性后，都必须调用 Initialize 方法进行初始化，初始化完成后，引擎实例才开始正常工作。

（2）INetworkEngine 从 IDisposable 继承，表明通信引擎内部持有了重要的资源，当不再使用其实例时，要尽快调用 IDisposable 的 Dispose 方法释放资源。

（3）当通信引擎被释放后，会触发 EngineDisposed 事件，并且此后，Disposed 属性将返回 true。

（4）请根据应用的需要谨慎地设置 MaxMessageSize，如果设置的过大，可能会造成内存空间的浪费（特别是对于基于文本协议的服务端引擎）。

（5）通过 MessageReceived 事件，可以得到通信引擎接收到的所有消息；通过 MessageSent 事件，可以监控通信引擎发送出去的所有消息。

2.IServerEngine

API：

```
// 摘要:
//       服务端引擎接口。
public interface IServerEngine : INetworkEngine, IDisposable
{
    // 摘要:
    //       到目标客户的通道是否繁忙？
    bool ChannelIsBusy(IPEndPoint client);
    //
    // 摘要:
    //       给某个客户端异步发信息。注意：如果引擎已经停止或客户端不在线，则直接返回。
    //
    // 参数:
    //     client:
    //       接收数据的客户端
    //
```

```
        //     msg:
        //         消息
        void PostMessageToClient(IPEndPoint client, byte[] msg);
        //
        // 摘要:
        //         给某个客户端异步发信息。注意：如果引擎已经停止或客户端不在线，则直接返回。
        //
        // 参数:
        //     client:
        //         接收数据的客户端
        //
        //     msg:
        //         消息
        //
        //     action:
        //         当通道繁忙时采取的动作：继续发送或丢弃。
        void PostMessageToClient(IPEndPoint client, byte[] msg, ActionTypeOnChannelIsBusy action);
        //
        // 摘要:
        //         给某个客户端同步发信息。注意：如果引擎已经停止或客户端不在线，则直接返回。
        //
        // 参数:
        //     client:
        //         接收数据的客户端
        //
        //     msg:
        //         消息
        void SendMessageToClient(IPEndPoint client, byte[] msg);
        //
        // 摘要:
        //         给某个客户端同步发信息。注意：如果引擎已经停止或客户端不在线，则直接返回。
        //
        // 参数:
        //     client:
        //         接收数据的客户端
        //
        //     msg:
        //         消息
        //
        //     action:
        //         当通道繁忙时采取的动作：继续发送或丢弃。
        void SendMessageToClient(IPEndPoint client, byte[] msg, ActionTypeOnChannelIsBusy action);
    }
```

（1）SendMessageToClient 和 PostMessageToClient 分别表示同步和异步发送消息给客户端。

（2）ChannelIsBusy 指的是在与目标客户端的 TCP 连接上，是否有数据正在发送（服务端至客户端）。

（3）ActionTypeOnChannelIsBusy 参数允许我们在通道繁忙时，丢弃不重要的消息。

3.IPassiveEngine

API：

```
// 摘要:
//     客户端引擎接口。
public interface IPassiveEngine : INetworkEngine, IDisposable
{
    // 摘要:
    //     发送消息的通道是否正忙。使用者可以根据该属性决定是否要丢弃后续某些消息的发送。
    bool ChannelIsBusy { get; }
    //
    // 摘要:
    //     服务器地址。
    AgileIPE ServerIPEndPoint { get; set; }

    // 摘要:
    //     将消息异步发送给服务器，不经任何处理，直接发送。注意：如果引擎已经停止，则直接返回。
    //
    // 参数:
    //   msg:
    //     要发送的消息
    void PostMessageToServer(byte[] msg);
    //
    // 摘要:
    //     将消息异步发送给服务器，不经任何处理，直接发送。注意：如果引擎已经停止，则直接返回。
    //
    // 参数:
    //   msg:
    //     要发送的消息
    //
    //   action:
    //     当通道繁忙时采取的动作：继续发送或丢弃。
    void PostMessageToServer(byte[] msg, ActionTypeOnChannelIsBusy action);
    //
    // 摘要:
    //     将消息同步发送给服务器，不经任何处理，直接发送。注意：如果引擎已经停止，则直接返回。
    //
    // 参数:
    //   msg:
    //     要发送的消息
    void SendMessageToServer(byte[] msg);
    //
    // 摘要:
    //     将消息同步发送给服务器，不经任何处理，直接发送。注意：如果引擎已经停止，则直接返回。
    //
    // 参数:
    //   msg:
    //     要发送的消息
    //
    //   action:
    //     当通道繁忙时采取的动作：继续发送或丢弃。
```

```
        void SendMessageToServer(byte[] msg, ActionTypeOnChannelIsBusy action);
    }
```

（1）SendMessageToServer 和 PostMessageToServer 分别表示同步和异步发送消息给服务端。

（2）ChannelIsBusy 指的是当前与服务器的 TCP 连接上，是否有数据正在发送（客户端至服务端）。

（3）ActionTypeOnChannelIsBusy 参数允许我们在通道繁忙时，丢弃不重要的消息。

4.ITcpServerEngine

API：

```
    // 摘要:
    //      TCP服务端引擎接口。
    public interface ITcpServerEngine : IServerEngine, INetworkEngine, IDisposable
    {
        // 摘要:
        //      是否开启异步消息队列。默认值为false。
        bool AsynMessageQueueEnabled { get; set; }
        //
        // 摘要:
        //      当前在线客户端的数量。
        int ClientCount { get; }
        //
        // 摘要:
        //      引擎所采用的协议类型（二进制、文本）。
        ContractFormatStyle ContractFormatStyle { get; }
        //
        // 摘要:
        //      监听器是否开启。
        bool IsListening { get; }
        //
        // 摘要:
        //      每个通道连接上允许最大的等待发送【包括投递】以及正在发送的消息个数。 当等待发送以及正在发送的消息个数超过该值时，将关闭对应的连接。如果设置为0，则表示不作限制。默认值为0。
        //      【该设置用于防止服务器为速度慢的通道缓存太多的消息，而导致服务器内存无限制增长】
        int MaxChannelCacheSize { get; set; }
        //
        // 摘要:
        //      服务器允许最大的同时在线客户端数。
        int MaxClientCount { get; }
        //
        // 摘要:
        //      给每个连接发送数据超时时间（默认为-1，无限）。如果在指定的时间内未将数据发送完，则将记录异常到日志，并关闭对应的连接。 该属性只对同步模式发送数据有效，异步发送数据则无法设定超时时间。
        //      最好给该属性赋一个有限值，因为在某些情况下，发送数据可能会导致无限阻塞。
        int WriteTimeoutInMSecs { get; set; }

        // 摘要:
        //      当某TCP连接建立成功时，触发此事件。
        event CbDelegate<IPEndPoint> ClientConnected;
        //
```

```
        // 摘要:
        //      当tcp连接数量发生变化时，触发此事件。
        event CbDelegate<int> ClientCountChanged;
        //
        // 摘要:
        //      当某TCP连接断开时，触发该事件
        event CbDelegate<IPEndPoint> ClientDisconnected;
        //
        // 摘要:
        //      当连接监听器的状态发生变化时，触发此事件。事件参数为true，表明连接监听器启动；事件参数为false，表明连接监听器已停止。
        event CbDelegate<bool> ListenerStateChanged;

        // 摘要:
        //      关闭或开启监听器。如果监听器关闭，将不再接受新的TCP连接请求。该方法调用不影响已有TCP连接上的消息接收和处理。
        void ChangeListenerState(bool enabled);
        //
        // 摘要:
        //      关闭所有客户端的连接。
        void CloseAllClient();
        //
        // 摘要:
        //      关闭到某个客户端的连接，将触发SomeOneDisconnected事件。
        void CloseClient(IPEndPoint client);
        //
        // 摘要:
        //      获取所有在线连接的客户端的地址。
        List<IPEndPoint> GetClientList();
        //
        // 摘要:
        //      客户端是否在线？
        bool IsClientOnline(IPEndPoint client);
        //
        // 摘要:
        //      客户端是否为WebSocket？如果返回null，表示还未接收到来自客户端的任何数据以进行判断或者是客户端不在线。
        bool? IsWebClient(IPEndPoint client);
    }
```

（1）WriteTimeoutInMSecs 用于设置发送数据的超时。最好给该属性赋一个适当的值，因为在某些情况下，发送数据可能会导致很长时间的阻塞。该属性只对同步发送有效。

（2）MaxChannelCacheSize 是服务端的一个安全设置。该设置用于防止服务器为速度慢的通道缓存太多的消息，而导致服务器内存无限制增长。

（3）ChangeListenerState 用于改变服务器的监听状态，其将触发 ListenerStateChanged 事件，并改变 IsListening 属性的值。

如果 IsListening 为 false，表示当前不接受新的 TCP 连接请求。

（4）当有连接建立或断开时，将分别触发 ClientConnected 和 ClientDisconnected 事件。

5.ITcpPassiveEngine

API :

```
// 摘要:
//      客户端TCP引擎接口。
public interface ITcpPassiveEngine : IPassiveEngine, INetworkEngine, IDisposable
{
    // 摘要:
    //      当与服务器断开连接时，是否自动重连。
    bool AutoReconnect { get; set; }
    //
    // 摘要:
    //      当前是否处于连接状态。
    bool Connected { get; }
    //
    // 摘要:
    //      引擎所采用的协议类型（二进制、文本）。
    ContractFormatStyle ContractFormatStyle { get; }
    //
    // 摘要:
    //      当连接断开时，自动重连尝试的最大次数。默认值为int.MaxValue。
    int MaxRetryCount4AutoReconnect { get; set; }
    //
    // 摘要:
    //      Sock5代理服务器信息。如果不需要代理，则设置为null。
    Sock5ProxyInfo Sock5ProxyInfo { get; set; }

    // 摘要:
    //      当客户端与服务器的TCP连接断开时，将触发此事件。
    event CbDelegate ConnectionInterrupted;
    //
    // 摘要:
    //      自动重连超过最大重试次数时，表明重连失败，将触发此事件。
    event CbDelegate ConnectionRebuildFailure;
    //
    // 摘要:
    //      自动重连开始时，触发此事件。
    event CbDelegate ConnectionRebuildStart;
    //
    // 摘要:
    //      自动重连成功后，触发此事件。
    event CbDelegate ConnectionRebuildSucceed;

    // 摘要:
    //      主动关闭与服务器的连接。如果AutoReconnect为true，将引发自动重连。
    void CloseConnection();
    //
    // 摘要:
    //      手动重连。如果当前处于连接状态，则直接返回。
```

```
        //
        // 参数:
        //     retryCount:
        //         重试次数
        //
        //     retrySpanInMSecs:
        //         重试间隔时间，毫秒
        void Reconnect(int retryCount, int retrySpanInMSecs);
    }
```

（1）如果 AutoReconnect 设置为 true，表示启用自动重连，那么，当连接断开时，会按以下顺序触发相关事件：ConnectionInterrupted、ConnectionRebuildStart、ConnectionRebuildSucceed/ConnectionRebuildFailure。

（2）注意，如果 AutoReconnect 设置为 true，CloseConnection 将会先关闭当前连接，然后再启动自动重连。

6.IUdpEngine

API：

```
    // 摘要:
    //     Udp引擎基础接口。
    public interface IUdpEngine : INetworkEngine, IDisposable
    {
        // 摘要:
        //     向指定的端点投递UDP消息。注意：如果引擎已经停止，则直接返回。
        //
        // 参数:
        //     message:
        //         要投递的消息
        //
        //     address:
        //         目标端点
        void PostMessage(IPEndPoint address, byte[] message);
        //
        // 摘要:
        //     向指定的端点发送UDP消息。注意：如果引擎已经停止，则直接返回。
        //
        // 参数:
        //     message:
        //         要发送的消息
        //
        //     address:
        //         目标端点
        void SendMessage(IPEndPoint address, byte[] message);
    }
```

UDP 是非连接的协议，所以，UDP 引擎不用区分客户端和服务端，或者说，无论是客户端还是服务端，都可以使用 IUdpEngine。注意：IUdpEngine 也从 INetworkEngine 继承，所以，它具备了 StriveEngine 中基础引擎所有的功能。

四.如何使用 StriveEngine

在 StriveEngine 中，我们不能直接 new 某个通信引擎的 class 来获得其实例。StriveEngine 提供了

NetworkEngineFactory，我们可以通过工厂的静态方法来得到通信引擎实例的引用。

1.通信引擎工厂

API：

```
// 摘要:
//     网络引擎工厂。
public static class NetworkEngineFactory
{
    // 摘要:
    //     创建使用二进制协议的TCP客户端引擎。对于返回的引擎实例，可以设置其更多属性，然后调用其Initialize方法启动引擎。
    //
    // 参数:
    //   serverIP:
    //     要连接的服务器的IP
    //
    //   serverPort:
    //     要连接的服务器的端口
    //
    //   helper:
    //     二进制协议助手接口
    public static ITcpPassiveEngine CreateStreamTcpPassivEngine(string serverIP, int serverPort, IStreamContractHelper helper);
    //
    // 摘要:
    //     创建使用二进制协议的TCP服务端引擎。对于返回的引擎实例，可以设置其更多属性，然后调用其Initialize方法启动引擎。
    //
    // 参数:
    //   port:
    //     服务端引擎监听的端口号
    //
    //   helper:
    //     二进制协议助手接口
    public static ITcpServerEngine CreateStreamTcpServerEngine(int port, IStreamContractHelper helper);
    //
    // 摘要:
    //     创建使用文本协议的TCP客户端引擎。对于返回的引擎实例，可以设置其更多属性，然后调用其Initialize方法启动引擎。
    //
    // 参数:
    //   serverIP:
    //     要连接的服务器的IP
    //
    //   serverPort:
    //     要连接的服务器的端口
    //
    //   helper:
    //     文本协议助手接口
```

```
        public static ITcpPassiveEngine CreateTextTcpPassiveEngine(string serverIP, int serverPort,
ITextContractHelper helper);
        //
        // 摘要:
        //      创建使用文本协议的TCP服务端引擎。对于返回的引擎实例，可以设置其更多属性，然后调用其
Initialize方法启动引擎。
        //
        // 参数:
        //   port:
        //      服务端引擎监听的端口号
        //
        //   helper:
        //      文本协议助手接口
        public static ITcpServerEngine CreateTextTcpServerEngine(int port, ITextContractHelper helper);
        //
        // 摘要:
        //      创建UDP引擎。对于返回的引擎实例，可以设置其更多属性，然后调用其Initialize方法启动引擎。
        public static IUdpEngine CreateUdpEngine();
    }
```

我们可以根据项目的需要（是 TCP 还是 UDP、是文本协议还是二进制协议、是用于客户端还是用于服务端），来调用 NetworkEngineFactory 的对应方法获得正确的通信引擎实例。

注意：NetworkEngineFactory 创建的所有通信引擎实例，必须要调用其 Initialize 方法后，引擎才算正常启动。当然，在调用 Initialize 之前，可以根据需要设置其相关的属性。

2.ContractHelper

StriveEngine 内部通过 ContractHelper 来从接收的网络流中识别完整的消息，针对消息格式为文本和二进制，ContractHelper 就划分为对应的 ITextContractHelper 和 IStreamContractHelper。

我们看到，在通过 NetworkEngineFactory 创建通信引擎实例时，其有个参数是必须传入 ContractHelper 引用的。所以，在项目中，我们必须实现 ITextContractHelper 或者是 IStreamContractHelper。

（1）ITextContractHelper

API：

```
    /// <summary>
    /// 文本协议助手接口。
    /// </summary>
    public interface ITextContractHelper
    {
        /// <summary>
        /// 消息结束标识符（经过编码后得到的字节数组）的集合。
        /// </summary>
        List<byte[]> EndTokens { get; }
    }
```

当使用文本协议时，通常，使用某一个或多个特殊的字符作为消息的结束符（EndToken）。ITextContractHelper 的 EndToken 是一个 byte[]，其指的是消息的结束符经过编码（比如 UTF-8）后得到的字节数组。而在一个系统中，可能会存在多个消息结束符（比如为了兼容多个老的系统），所以，EndTokens 是一个集合，其中每一个元素都表示一个结束符。通信引擎即通过 ITextContractHelper 提供的 EndTokens 来从网络流中识别完整的消息。（特别说明，若指定 EndTokens 为 null 则表示不需要处理粘包以及进行消息完整性识别，此时，引擎会直接返回接收到的数据）。

如果是使用 UTF-8 对文本消息（当然，也包括消息结束符）进行编解码，则可以使用 StriveEngine 内置的 DefaultTextContractHelper，其实，它的实现非常简单：

API：

```
public class DefaultTextContractHelper : ITextContractHelper
{
    public DefaultTextContractHelper(params string[] endTokenStrs)
    {
        this.endTokens = new List<byte[]>(); ;
        if (endTokenStrs == null || endTokenStrs.Length == 0)
        {
            return;
        }

        foreach (string str in endTokenStrs)
        {
            this.endTokens.Add(System.Text.Encoding.UTF8.GetBytes(str));
        }
    }

    private List<byte[]> endTokens;
    public List<byte[]> EndTokens
    {
        get
        {
            return this.endTokens;
        }
    }
}
```

如果想像上面说的不需要识别消息完整性，即将 EndTokens 为 null，那么可以使用下面这个 ITextContractHelper 实现：

API：

```
public class DefaultTextContractHelper2 : ITextContractHelper
{
    public List<byte[]> EndTokens
    {
        get
        {
            return null;
        }
    }
}
```

（2）IStreamContractHelper

API：

```
/// <summary>
/// 二进制协议助手接口。
/// </summary>
public interface IStreamContractHelper
```

```
    {
        /// <summary>
        /// 从消息头中解析出消息体的长度。
        /// </summary>
        /// <param name="head">完整的消息头，长度固定为MessageHeaderLength</param>
        /// <returns>消息体的长度</returns>
        int ParseMessageBodyLength(byte[] head);

        /// <summary>
        /// 消息头的长度。
        /// </summary>
        int MessageHeaderLength { get; }
    }
```

当使用二进制协议时，通常，消息分为消息头（Header）和消息体（Body）两部分，消息头是必须的，而消息体可以为 null。消息头的长度是固定的（比如 8 个字节），且其至少包含了一个字段--消息体的长度（或根据消息头的内容可以间接结算出消息体的长度）。

请特别注意--

虽然协议的类型分为“文本”和“二进制”，但千万不要被它们的名称所迷惑，它们底层都是二进制的。其本质区别在于：它们对于识别“一个完整消息”所采用的模型不一样。“文本协议”是以特殊的字符作为消息的结束符来区分每一个消息的，而“二进制协议”是将消息分为了消息头和消息体，且消息头的长度是固定的。

3.使用 StriveEngine 的步骤

（1 实现 ITextContractHelper 或者是 IStreamContractHelper 接口(如何实现该接口 ,可参考后面 demo 的源码)。

（2）调用 NetworkEngineFactory 的创建引擎的方法，得到正确的通信引擎实例。

（3）根据需要，设置引擎实例的某些属性（如 MaxMessageSize 属性）。

（4）根据需要，预定引擎实例的某些事件（如 MessageReceived 事件）。

（5）调用引擎实例的 Initialize 方法启动通信引擎。

五.支持多种客户端平台类型

StriveEngine 在客户端除了支持.NET 平台外，还支持如下平台：Unity3D、WinCE、HTML5WebSockets。

1.Unity3D 平台

StriveEngine.U3D.dll 是专门为 Unity3D 而准备的，是 StriveEngine 中客户端引擎向 Unity3D 的完全移植，命名空间、接口名称等等一切都是完全一样的。也就是说，如果您的客户端环境是 Unity3D，那么只需要引用 StriveEngine.U3D.dll，就可以了，上面所讲述的一切对它都是有效的。

StriveEngine.U3D 使用的是.NETFramework 的子集，其可被 Unity3D 打包发布到 pc、web、android、ios 等平台。

在一般的应用开发中，服务端和客户端引用同一个 StriveEngine.dll 即可；而在 Unity3D 网络游戏开发中，服务端仍然引用 StriveEngine.dll，而客户端将引用 StriveEngine.U3D.dll。

2.WinCE 平台

StriveEngine.CE.dll 是专门为 WindowsCE 嵌入式和移动设备而准备的，是 StriveEngine 中客户端引擎向 WindowsCE 的完全移植，命名空间、接口名称等等一切都是完全一样的。也就是说，如果您的客户端环境是 WindowsCE，那么只需要引用 StriveEngine.CE.dll，上面所讲述的一切对它都是有效的。

同 StriveEngine.U3D 一样，在 WinCE 网络通信系统开发中，服务端仍然引用 StriveEngine.dll，而 WinCE 客户端将引用 StriveEngine.CE.dll。

3.HTML5WebSockets

StriveEngine 对 WebSockets 的支持是内置的。如果客户端的类型是 WebSockets，那么，StriveEngine 服务端引擎会自动完成如下几件事情：

（1）自动完成 WebSokects 握手协议。

（2）针对接收到的每个 WebSokect 数据帧，都会自动将其解析，并从中分离出真正的消息内容--服务端引擎的 MessageReceived 事件，暴露的就是解析后的真正消息内容。

（3）当您调用 SendMessageToClient 方法发送消息给客户端时，服务端引擎会自动将该消息封装成 WebSokect 数据帧，然后再发送出去。

所以，在使用 StriveEngine 处理 WebSockets 客户端时，使用者不需要做任何额外的处理。如果在服务端想知道某个客户端是否为 WebSockets，可以调用服务端引擎的 IsWebClient 方法。

API：

```
/// <summary>
/// 客户端是否为WebSocket？如果返回null，表示还未接收到来自客户端的任何数据以进行判断或者是客户端不在线。
/// </summary>
bool? IsWebClient(IPEndPoint client);
```

最后请注意：在绝大多数情况下，客户端通过 WebSocket 发送的都是文本消息，而服务端引擎触发 MessageReceived 事件的参数是 byte[]，那么如何将其还原成文本消息了？很简单，只要使用 UTF-8 将 byte[]解析成字符串就可以了（调用 System.Text.Encoding.UTF8.GetString()方法）。

六.Demo 及下载

下载 StriveEngineSDK。

通过上述的内容大致了解了 StriveEngine 后，接下来我们通过两个简单的 demo，来看看在实际项目中是如何基于文本协议和二进制协议来使用 StriveEngine 的。

1.基于文本协议的 demo

下载基于文本协议的 Demo 源码。

该 Demo 是一个简单的客户端与服务端进行消息通信的 demo，消息使用文本协议。demo 中，消息的结束符标志符采用的是一个字符（'\0'），消息内容使用 UTF-8 进行编码。

该 Demo 总共包括三个项目和一个 HTML 文件：

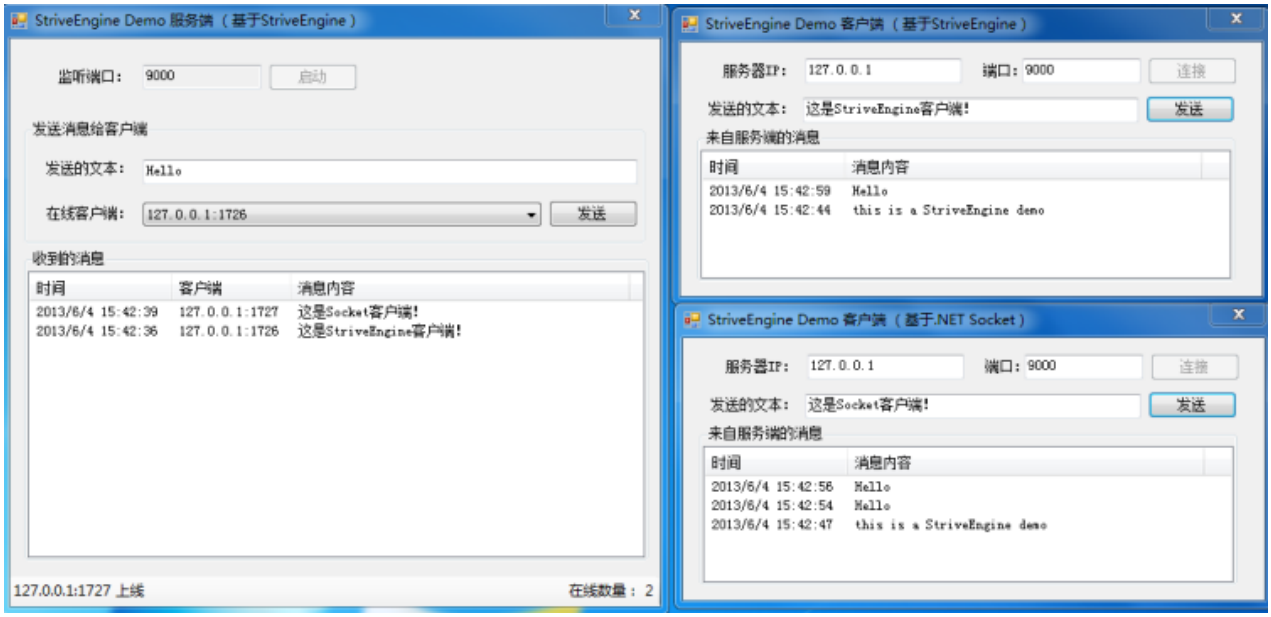
（1）StriveEngine.SimpleDemoServer：基于 StriveEngine 开发的服务端。

（2）StriveEngine.SimpleDemoClient：基于 StriveEngine 开发的客户端。

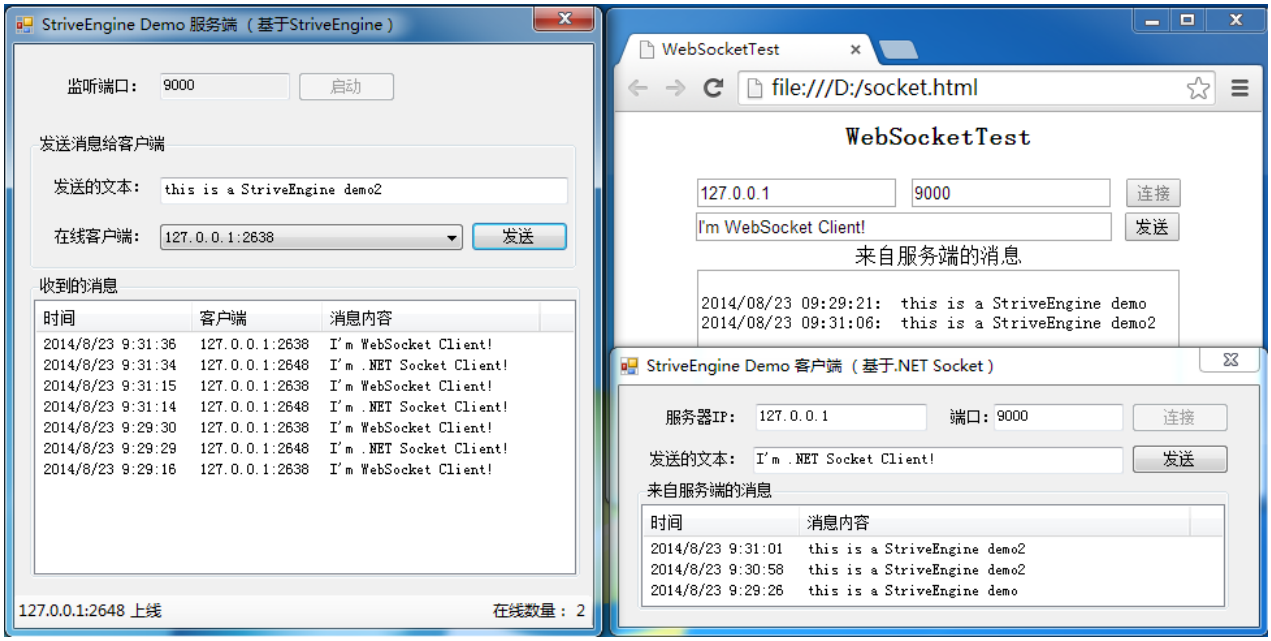
（3）StriveEngine.SimpleDemo：直接基于.NET 的 Socket 开发的客户端，其目的是为了演示：在客户端不使用 StriveEngine 的情况下，如何与基于 StriveEngine 的服务端进行通信。

（4）WebSocketClient.html：基于 HTML5WebSocket 的客户端。与前两种客户端公用同一个 StriveEngine 服务端。

Demo 运行起来后的截图如下所示：



加入 WebSocket 客户端（用浏览器打开 WebSocketClient.html 页面：



2.基于二进制协议的 demo

下载基于二进制协议的 Demo 源码。

该 Demo 演示了客户端提交运算请求给服务端，服务端处理后，将结果返回给客户端。消息使用二进制协议。demo 中，消息头固定为 8 个字节：前四个字节为一个 int，表示消息体的长度；后四个字节也是一个 int，表示消息的类型。

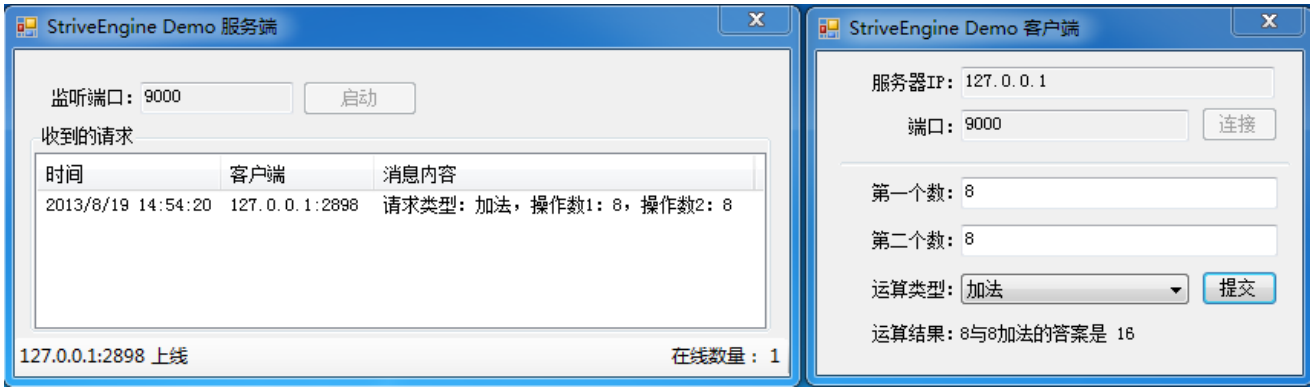
该 Demo 总共包括三个项目：

（1）StriveEngine.BinaryDemoServer：基于 StriveEngine 开发的服务端。

（2）StriveEngine.BinaryDemo：基于 StriveEngine 开发的客户端。

（3）StriveEngine.BinaryDemoCore：用于定义客户端和服务端都要用到的公共的消息类型和消息协议的基础程序集。

Demo 运行起来后的截图如下所示：



（03）—— OAUS 自动升级系统

对于 PC 桌面应用程序而言，自动升级功能往往是必不可少的。而自动升级可以作为一个独立的 C/S 系统来开发，这样，就可以在不同的桌面应用中进行复用。本文将着重介绍 OAUS 的相关背景、使用方法，至于详细的实现细节，大家可以直接下载源码研究。如果了解了 OAUS 的使用，源码的理解就非常容易了。如果需要直接部署使用自动升级系统，那么，可下载文末的可执行程序压缩包。

一.OAUS 的主要功能

目前主流的程序自动升级策略是，重新下载最新的安装包，然后重新安装整个客户端。这种方式虽然简单直观，但是缺陷也很明显。比如，即使整个客户端有 100M，而本次更新仅仅只是修改了一个 1k 大小的 dll，那也意味着要重新下载 100M 的全部内容。这对带宽是极大的浪费，而且延长了升级了时间，相应地也增加了客户茫然等待的时间。

在上述的场景中，自动升级时，我们能否只更新那个被修改了的 1k 的 dll 了？当然，使用 OAUS 自动升级系统可以轻松地做到这一点。OAUS 自动升级系统可以对被分发的客户端程序中的每个文件进行版本管理，每次升级的基础单元不再是整个客户端程序，而是其中的单个文件。针对单个文件的更新，包括三种形式：

（1）文件被修改。

（2）文件被删除。

（3）新增加某个文件。

OAUS 对这三种形式的文件更新都是支持的。每次自动升级，都可以更改 N 个文件、删除 M 个文件、新增加 L 个文件。

二.OAUS 的使用

1.OAUS 的结构

OAUS 提供了可直接执行的服务端程序和客户端程序：AutoUpdaterSystem.Server.exe 和 AutoUpdater.exe。OAUS 服务端的目录结构如下所示：

OAUS 的客户端与服务器之间通过 TCP 通信，可以在 AutoUpdaterSystem.Server.exe.config 配置文件中配置服务器通过哪个 TCP 端口提供自动升级服务。

FileFolder 文件夹初始是空的，其用于部署被分发的程序的各个文件的最新版本。注意，其下的文件结构一定要与被分发的程序正常部署后的结构完全一致--即相当于在 FileFolder 文件夹下部署一个被分发的程序。

OAUS 客户端的目录结构如下：

可以在 AutoUpdater.exe.config 配置文件中配置 OAUS 服务器的 IP、端口等信息，其内容如下所示：

API:

```
</configSections>
  <appSettings>
    <!--服务器IP -->
    <add key="ServerIP" value="127.0.0.1"/>
    <!--服务器端口-->
    <add key="ServerPort" value="4540"/>
    <!--升级完成后，将被回调的可执行程序的名称-->
    <add key="CallbackExeName" value="Netalks.exe"/>
    <!--主窗体的Title-->
    <add key="Title" value="文件更新"/>
  </appSettings>
</configuration>
```

请注意配置的 CallbackExeName，其表示当升级完成之后，将被启动的分发程序的 exe 的名称。这个 CallbackExeName 配置的为什么是名称而不是路径了？这是因为使用和部署 OAUS 客户端时是有要求的：

（1）被分发的程序的可执行文件 exe 必须位于部署目录的根目录。

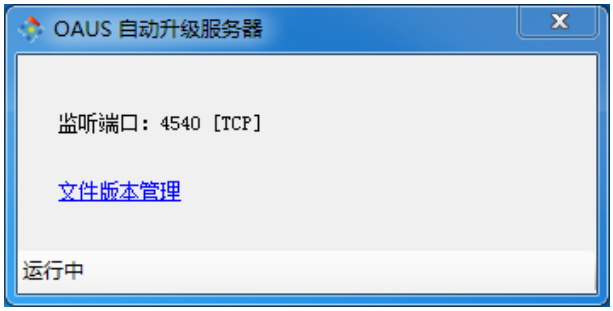
（2）OAUS 的客户端（即整个 AutoUpdater 文件夹)也必须位于这个根目录。

如此，AutoUpdater 就知道分发程序的 exe 相对自己的路径，如此就可以确定分发程序的 exe 的绝对路径，所以就可以在升级完成后启动目标 exe 了。另外，根据上述的两个约定，再结合前面讲到的服务端的 FileFolder 文件夹的结构约定，当服务端更新一个文件时，AutoUpdater 便可以确定该文件在客户端机器上的绝对路径了。

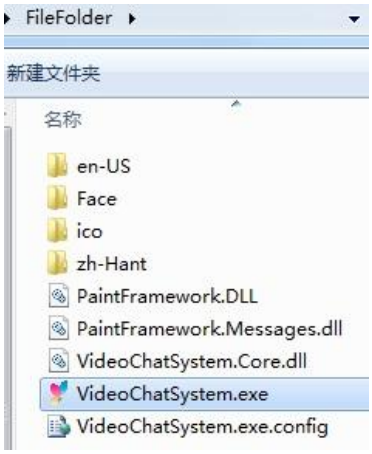
2.OAUS 自动升级流程

下面我们就详细讲讲如何使用 OAUS 来构建自动升级系统，大概的步骤如下。

（1）运行 OAUS 服务端。

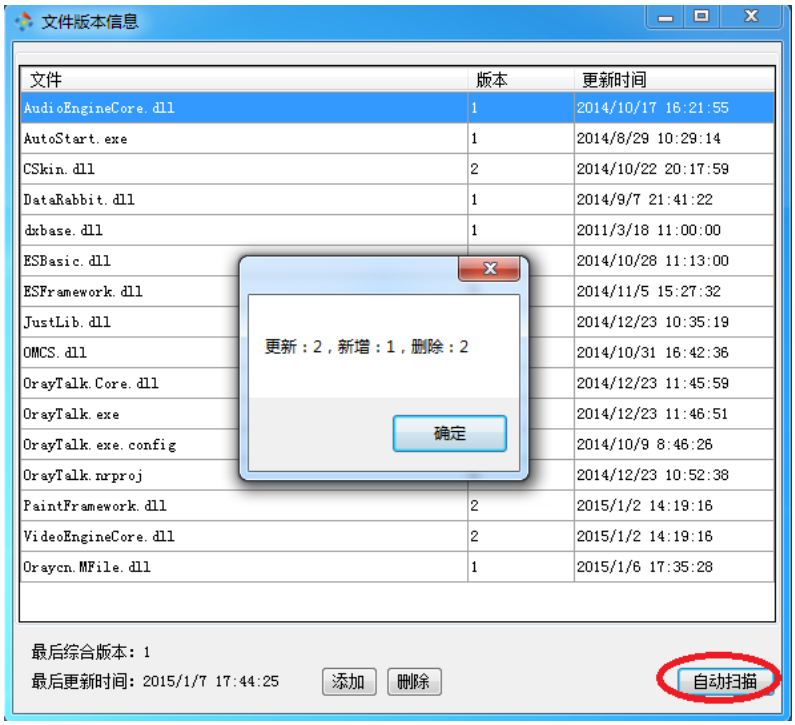


（2）将被分发的客户端程序的所有内容放到 OAUS 服务端的 FileFolder 文件夹下，其结构与客户端程序正常部署后的结构要完全一致。我们以部署 VideoChatSystem 为例。



（3）使用 OAUS 服务端为被分发的客户端程序的每个文件生成默认版本号，并创建版本信息配置文件 UpdateConfiguration.xml。这个配置文件也将被客户端使用。

点击服务端界面上的“文件版本管理”按钮，将弹出用于管理各个文件版本的【文件版本信息】窗体。



当用新版本的文件覆盖老的文件后，点击“自动扫描”按钮，服务端就会检索 FileFolder 文件夹下文件的名称、大小、最后更新时间，然后得出本次更新结果：变化了几个文件、新增了几个文件、删除了几个文件。

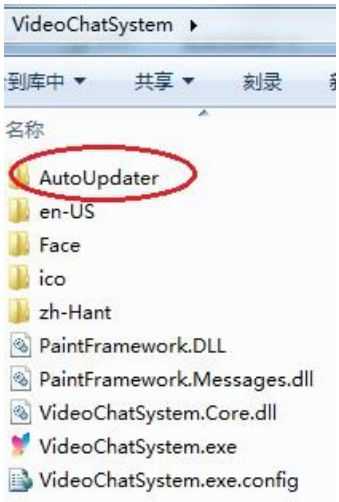
当关闭【文件版本信息】窗体时，只要有某个文件版本变化，则“最后综合版本”的值（int 类型）会递增 1。通过比较 OAUS 的客户端保存的“最后综合版本”的值与 OAUS 的服务端最新的“最后综合版本”的值，就可以快速地识别客户端是否已经是最新版本了。

另外，初次打开这个窗口时，将在 OAUS 服务端的目录下，自动生成一个版本信息配置文件 UpdateConfiguration.xml。而且，每当通过该窗体来设置某个文件的新版本时，UpdateConfiguration.xml 会自动同步更新。

（4）将 UpdateConfiguration.xml 添加到 OAUS 的客户端程序（即上述的 AutoUpdater 的文件夹）中。

（5）在创建被分发的客户端的安装程序时，将 OAUS 的客户端（即 AutoUpdater 的文件夹）也打包进去，并且像前面说的一样，要将其直接部署在运行目录（BaseDirectory）下（与分发的 exe 同一目录）。

如此，准备工作就完成了，当客户端通过安装包安装好了 VideoChatSystem 之后，其目录结构像下面这样：



（6）当我们有新的版本要发布时，比如要更新某个文件（因为文件被修改），只需：

a.将修改后的文件拷贝到 OAUS 服务端的 FileFolder 文件夹下的正确位置（或覆盖旧的文件）。

b.在 OAUS 服务端打开【文件版本信息】窗体，点击“自动扫描”按钮，扫描完成后，关闭该窗体即可。

（7）如此，当客户端再启动 AutoUpdater.exe 时，就会自动升级，更新那些发生变化的文件。以下是 AutoUpdater.exe 运行起来后的截图。

（8）当升级完成后，将启动前述的 OAUS 客户端配置文件中配置的回调 exe。（在本例中就是 VideoChatSystem.exe）

（9）OAUS 客户端会在日志文件 UpdateLog.txt（位于 AutoUpdater 的文件夹下，在 OAUS 客户端首次运行时自动生成该文件）中，记录每次自动升级的情况。

（10）如果升级的过程中，与服务端连接中断，则会自动重连，在重连成功后，将启动断点续传。

3.何时启动自动升级客户端？

假设某个系统是下载客户端形式的，那么客户端该如何知道是否有新版本了？然后又该何时启动 AutoUpdater.exe 了？

我们的经验是这样的：客户端登录成功之后，从服务器获取“最后综合版本”的值，然后与本地的“最后综合版本”的值相比较，如果本地的值较小，则表示客户端需要更新。这个过程可以这样做到：

（1）当在 OAUS 服务端的 FileFolder 文件夹下放置了新的文件，并通过【文件版本信息】窗体正确的更新了版本号，在关闭【文件版本信息】窗体时，“最后综合版本”的值会自动加 1。

（2 系统客户端可以通过调用 OAUS.Core.VersionHelper 类的静态方法 HasNewVersion()来判断是否有新版本。

（3）如果 HasNewVersion 方法返回 true，则通常有两种模式：由用户选择是否升级，或者是强制升级。

一般而言，如果最新客户端程序与老版本兼容，不升级也影响不大，则可以交由用户决定是否升级；如果最新客户端程序不兼容老版本，或者是有重大更新，则将启动强制升级。如果流程要进入启动升级，那么只要启动 AutoUpdater 的文件夹下 AutoUpdater.exe 就可以了。要注意的是，启动 AutoUpdater.exe 进程后，要退出当前的客户端进程，否则，有些文件会因为无法被覆盖而导致更新失败。代码大致如下所示：

API:

```
if (VersionHelper.HasNewVersion(oausServerIP, oausServerPort))
{
    string updateExePath = AppDomain.CurrentDomain.BaseDirectory + "AutoUpdater\\AutoUpdater.exe";
    System.Diagnostics.Process myProcess = System.Diagnostics.Process.Start(updateExePath);
    ....//退出当前进程
}
```

三.相关下载

1.自动升级系统 OAUS-源码

2.自动升级系统 OAUS（可直接部署）

（04）——语音视频采集组件 MCapture

在多媒体系统中，一般都会涉及到语音、视频、桌面的数据采集问题，采集得到的数据可以用来传输、播放、或存储。所以，对于像课件录制系统、语音视频录制系统、录屏系统等，多媒体数据的采集就是最基础的功能之一。

MCapture 是傲瑞实用组件之一，可用于采集本地摄像头拍摄到的图像、麦克风输入的声音、声卡播放的声音、以及当前电脑桌面的图像，并提供了混音器功能。

一.MCapture 简介

MCapture 组件内部的核心技术包括以下 5 个工具：

（1）摄像头采集器：可指定摄像头的索引、摄像头视频的分辨率、采集的帧频。

（2）麦克风采集器：可指定麦克风的索引。默认采样参数为--采样位数：16bit，采样频率：16000，声道数：1。

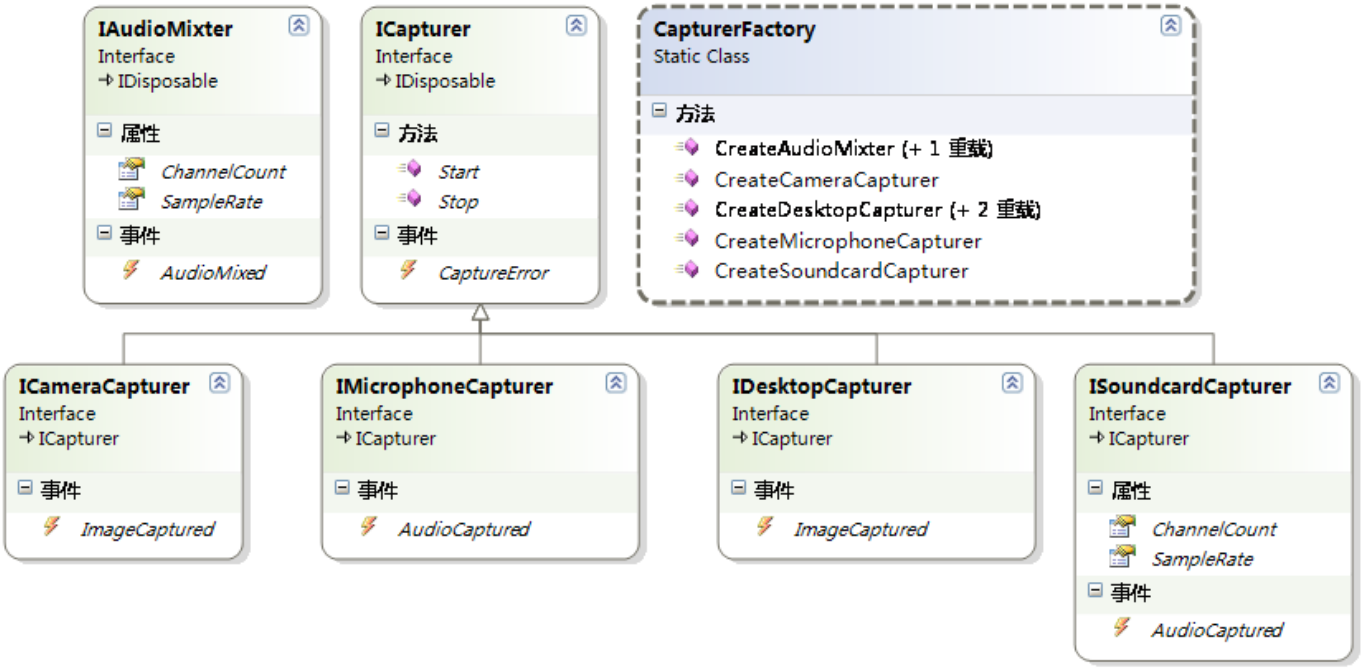
（3）桌面屏幕采集器：可指定采集的帧频、是全屏采集还是采集屏幕的某个矩形区域、是否显示鼠标光标。

（4）声卡采集器：采样位数：固定为 16bit。通常的采样频率为：48000，声道数：2。【注：声卡采集目前仅支持 windowsvista/7/8 及以上，不支持 xp 系统】

（5）混音器：传入麦克风采集器和声卡采集器，便能输出混音后的数据。【要求声卡采集器的采样率：48000，声道数：为 2】

二.MCapture 结构

对于使用者而言，MCapture 组件中的主要类的结构图如下所示：



1.对于各采集器

（1）ICameraCapturer 是摄像头视频采集组件；IMicrophoneCapturer 是麦克风声音采集组件；IDesktopCapturer 是屏幕桌面采集组件；ISoundcardCapturer 是声卡播放声音的采集组件。它们都集成自 ICapture 基础接口。

（2）我们可以通过调用 CapturerFactory 的 CreateXXXX 方法来创建对应的采集器实例。

（3）得到采集器实例后，调用 Start 方法，即可开始采集；调用 Stop 方法，即停止采集。

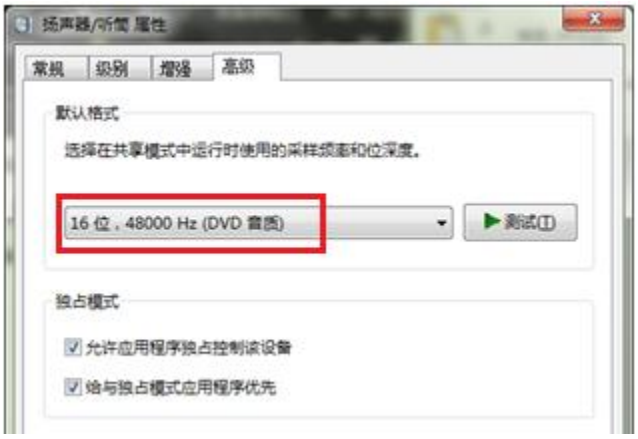
（4）采集得到的数据，将通过相应的事件（ImageCaptured、AudioCaptured）暴露出来，我们预定这些事件，即可拿到采集的数据。

2.对于混音器 IAudioMixter

（1）我们可以通过调用 CapturerFactory 的 CreateAudioMixter 方法来创建混音器实例。

（2）当传入 CreateAudioMixter 方法的麦克风采集器和声卡采集器开始工作后（即它们的 Start 方法被调用后），IAudioMixter 将触发 AudioMixed 事件以暴露混音后的数据。

注：如果 CapturerFactory 的 CreateAudioMixter 方法抛出异常，很可能是声卡的采样率不是 48000 导致的，此时，可以按下图设置一下：



三.使用接口详细定义

1.各采集器接口

API :

```
/// <summary>
/// 傲瑞采集器基础接口。
/// </summary>
public interface ICapturer
{
    /// <summary>
    /// 如果采集的过程中发生错误，将触发此事件。
    /// </summary>
    event CbGeneric<Exception> CaptureError;

    /// <summary>
    /// 开始采集。
    /// </summary>
    void Start();

    /// <summary>
    /// 停止采集。
    /// </summary>
    void Stop();
}

/// <summary>
/// 摄像头采集器。
/// </summary>
public interface ICameraCapturer : ICapturer
{
    /// <summary>
    /// 当采集到一帧图像时，触发该事件。
    /// </summary>
    event CbGeneric<Bitmap> ImageCaptured;
}

/// <summary>
/// 桌面采集器。
/// </summary>
public interface IDesktopCapturer : ICapturer
{
    /// <summary>
    /// 当采集到一帧图像时，触发该事件。
    /// </summary>
    event CbGeneric<Bitmap> ImageCaptured;
}

/// <summary>
/// 麦克风采集器。
/// </summary>
public interface IMicrophoneCapturer : ICapturer
{
```

```
        /// <summary>
        /// 当采集到一帧声音数据（20ms）时，触发该事件。
        /// </summary>
        event CbGeneric<byte[]> AudioCaptured;
    }
    /// <summary>
    /// 声卡采集器。 【注意：声卡采集器目前只支持Windows vista/7/8】
    /// </summary>
    public interface ISoundcardCapturer : ICapturer
    {
        /// <summary>
        /// 声道数。
        /// </summary>
        int ChannelCount { get; }
        /// <summary>
        /// 采样频率。
        /// </summary>
        int SampleRate { get; }
        /// <summary>
        /// 当采集到一帧声音数据（50ms）时，触发该事件。
        /// </summary>
        event CbGeneric<byte[]> AudioCaptured;
    }
    /// <summary>
    /// 混音器。采样位数：16bit。
    /// </summary>
    public interface IAudioMixter : IDisposable
    {
        /// <summary>
        /// 输出的混音结果的声道数。
        /// </summary>
        int ChannelCount { get; }

        /// <summary>
        /// 输出的混音结果的采样频率。
        /// </summary>
        int SampleRate { get; }

        /// <summary>
        /// 当完成一帧声音数据的混音（20ms）时，触发该事件。
        /// 采样位数：16bit。
        /// </summary>
        event CbGeneric<byte[]> AudioMixed;
    }
```

要特别提醒的是：

（1）ICapture 接口的 CaptureError 事件，当采集的过程中出现错误时，将触发此事件，并且，采集过程会终止。

（2）针对视频和桌面采集，ImageCaptured 会暴露出采集得到的视频帧（Bitmap），当该视频帧使用完毕后，要立即调用其 Dispose 方法，以释放其占用的内存（而不要等到 GC 自动回收）。

（3）声卡采集器的采集参数不是固定的，可以通过 ISoundcardCapturer 的属性来获得采集的声道数和频率信息。

2.采集器工厂

API：

```
// 摘要:
//      采集器工厂。
public static class CapturerFactory
{
    // 摘要:
    //      创建混音器。（将麦克风采集的数据与声卡采集的数据进行混音。）
    //
    // 参数:
    //   microphoneCapturer:
    //      麦克风采集器
    //
    //   soundcardCapturer:
    //      声卡采集器。要求声卡的SampleRate必须为48000。
    //
    // 返回结果:
    //      混音器
    public static IAudioMixter CreateAudioMixter(IMicrophoneCapturer microphoneCapturer, ISoundcardCapturer soundcardCapturer);
    //
    // 摘要:
    //      创建混音器。（将麦克风采集的数据与声卡采集的数据进行混音。）
    //
    // 参数:
    //   microphoneCapturer:
    //      麦克风采集器
    //
    //   soundcardCapturer:
    //      声卡采集器。要求声卡的SampleRate必须为48000。
    //
    //   mode:
    //      声卡声道选择模式。由于声卡采集的结果通常是双声道的，该参数决定混音时是使用左声道数据？还是使用右声道数据？或者是左右声道数据都使用？
    //
    //   output2Channel:
    //      混音结果是否采用双声道？如果为true，则表示麦克风和声卡的数据在混音结果中各占一个声道。
    //
    // 返回结果:
    //      混音器
    public static IAudioMixter CreateAudioMixter(IMicrophoneCapturer microphoneCapturer, ISoundcardCapturer soundcardCapturer, SoundcardMode4Mix mode, bool output2Channel);
    //
    // 摘要:
    //      创建摄像头采集器。
    //
    // 参数:
```

```
        //     cameraIndex:
        //         摄像头的索引
        //
        //     videoSize:
        //         摄像头的分辨率
        //
        // 返回结果:
        //         摄像头采集器
        public static ICameraCapturer CreateCameraCapturer(int cameraIndex, Size videoSize);
        //
        // 摘要:
        //         创建摄像头采集器。
        //
        // 参数:
        //     cameraIndex:
        //         摄像头的索引
        //
        //     videoSize:
        //         摄像头的分辨率
        //
        //     fps:
        //         采集的帧频
        //
        // 返回结果:
        //         摄像头采集器
        public static ICameraCapturer CreateCameraCapturer(int cameraIndex, Size videoSize, int fps);
        //
        // 摘要:
        //         创建桌面采集器。
        //
        // 参数:
        //     fps:
        //         采集的帧频。
        //
        //     showMouseCursor:
        //         采集的图像中是否显示鼠标的光标
        //
        // 返回结果:
        //         桌面采集器
        public static IDesktopCapturer CreateDesktopCapturer(int fps, bool showMouseCursor);
        //
        // 摘要:
        //         创建桌面采集器。
        //
        // 参数:
        //     fps:
        //         采集的帧频。
        //
        //     showMouseCursor:
```

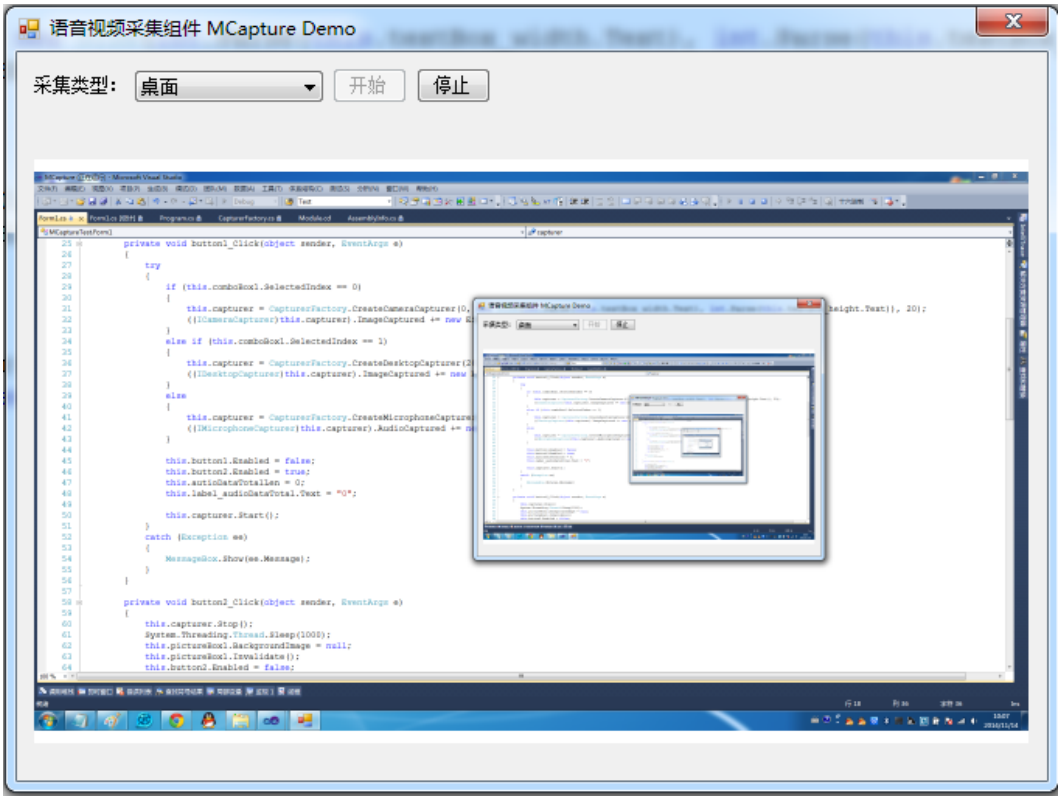
```
        //        采集的图像中是否显示鼠标的光标
        //
        //    screenIndex:
        //        要采集的屏幕设备的索引。
        //
        // 返回结果:
        //        桌面采集器
        public static IDesktopCapturer CreateDesktopCapturer(int fps, bool showMouseCursor, int screenIndex);
        //
        // 摘要:
        //        创建桌面采集器。
        //
        // 参数:
        //    fps:
        //        采集的帧频。
        //
        //    showMouseCursor:
        //        采集的图像中是否显示鼠标的光标
        //
        //    captureRect:
        //        要采集桌面的区域。
        //
        // 返回结果:
        //        桌面采集器
        public static IDesktopCapturer CreateDesktopCapturer(int fps, bool showMouseCursor, Rectangle? captureRect);
        //
        // 摘要:
        //        创建麦克风采集器（采样位数：16bit，采样频率：16000，声道数：1）。
        //
        // 参数:
        //    microphoneIndex:
        //        麦克风的索引
        //
        // 返回结果:
        //        麦克风采集器
        public static IMicrophoneCapturer CreateMicrophoneCapturer(int microphoneIndex);
        //
        // 摘要:
        //        创建麦克风采集器（采样位数：16bit，声道数：1）。
        //
        // 参数:
        //    microphoneIndex:
        //        麦克风的索引
        //
        //    sr:
        //        采样频率
        //
        // 返回结果:
        //        麦克风采集器
```

```
        public static IMicrophoneCapturer CreateMicrophoneCapturer(int microphoneIndex, WaveSampleRate sr);
        //
        // 摘要:
        //       创建声卡采集器（采样位数：16bit）。
        //
        // 返回结果:
        //       声卡采集器
        public static ISoundcardCapturer CreateSoundcardCapturer();
    }
```

四.MCaptureDemo

1.第一个 Demo：各采集器功能展现

下面是一个最简单的使用 MCapture 组件的 Demo，在这个 demo 中，我们直接将采集到的视频数据显示在主窗体上，如果是语音数据，则在界面显示采集得到的语音数据的长度。Demo 运行的截图如下所示：



MCaptureDemo 源码：Oraycn.MCaptureDemo.rar

如果是要录制采集的语音视频或录制屏幕，可以将 MCapture 与我们的语音视频录制组件 MFile 相结合，来快速实现这一功能，正如下面第二个 Demo 所介绍的。

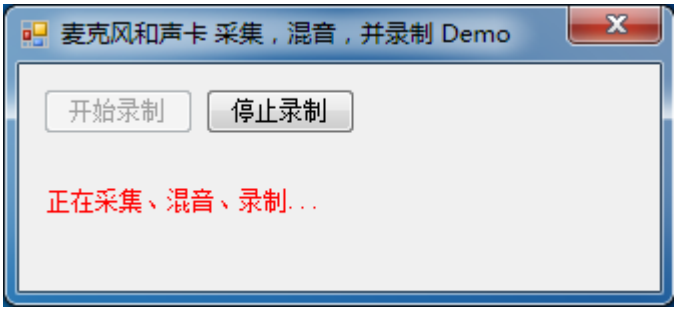
2.第二个 Demo：录制声卡

下面是一个声卡录制的 demo，使用 MCapture 的声卡采集功能和 MFile 的音频录制功能，即录制声卡播放的声音，demo 运行的截图如下所示：

声卡录制 Demo 源码：Oraycn.RecordSoundCardDemo.rar

3.第三个 Demo：混音（麦克风和声卡）并录制

将麦克风采集的声音和声卡采集的声音进行混音，然后录制成 mp3 文件，demo 运行的截图如下所示：



混音录制 Demo 源码：Oraycn.MixAndRecordDemo.rar

4.第四个 Demo：同时录制（桌面+麦克风+声卡）Demo

将声卡/麦克风/屏幕的采集与录制集中在一个 Demo 中：Oraycn.DesktopMixedAudioRecordDemo.rar

5.第五个 Demo：最完整的采集与录制 Demo，提供 WindowsForm 版本和 WPF 版本，五星推荐！！！

将声卡、麦克风、摄像头、屏幕的采集与录制集中在一个 Demo 中：

（声卡/麦克风/摄像头/屏幕）采集&录制 Demo：WindowsForm 版本，WPF 版本。

6.录制画中画（桌面+摄像头+麦克风/声卡）

将摄像头的画面叠加在屏幕的画面上，然后结合麦克风（或/和声卡）进行录制的 Demo：

画中画录制 Demo：Oraycn.RecordPictureCompositionDemo.rar

（05）—— 语音视频播放组件 MPlayer

在多媒体系统中，一般都会涉及到语音数据、视频数据的播放问题，比如播放采集到的麦克风数据、摄像头数据，或播放 wav 文件等。

MPlayer 是傲瑞实用组件之一，可用于播放声音数据、视频图像数据。

一.MPlayer 简介

当前版本的 MPlayer 组件支持：

（1）播放声音：直接播放 PCM 声音数据，播放 WAV 文件。播放时，可指定扬声器索引、声音数据的采样率、采样深度、声道数。

（2）播放图像：该功能暂不支持。

二.MPlayer 结构

对于使用者而言，MPlayer 组件中的主要类的结构图如下所示：

（1）IAudioPlayer 是声音播放器。

（2）我们可以通过调用 PlayerFactory 的 CreateAudioPlayer 方法来创建声音播放器实例。

（3）得到播放器实例后，使用声音数据调用其 Play 方法，就会播放传入的数据；不再使用播放器时，调用其 Dispose 方法将其释放。

三.使用接口详细定义

1.各播放器接口

API:

```
/// <summary>
/// 声音播放器。用于播放PCM声音数据。
/// </summary>
public interface IAudioPlayer : IDisposable
{
    /// <summary>
    /// 将声音数据放到播放队列中，进行播放。
    /// </summary>
    /// <param name="audioData">声音数据</param>
    void Play(byte[] audioData);

    /// <summary>
    /// 清除缓冲区中尚未播放的数据。
    /// </summary>
    void Clear();
}
```

要特别提醒的是：

（1）声音播放器 IAudioPlayer 内部使用了缓存队列，缓存的大小可在创建时进行设置。如果缓存中的数据尚未播放完，此时调用 Play 传入的数据将会在后面排队。

（2）如果调用 Play 传入的数据的量大于缓存中剩下的空间，则会自动丢弃缓存中最老的数据。

（3）调用 Clear 方法，可以立即情况缓存中尚未播放的数据。

2.播放器工厂

API:

```
// 摘要:
//     播放器工厂。
public static class PlayerFactory
{
    // 摘要:
    //     创建声音播放器。用于播放PCM声音数据。默认采样位数：16，默认缓存大小：10秒。
    //
    // 参数:
    //   speakerIndex:
    //     使用的扬声器设备的索引。
    //
    //   sampleRate:
    //     声音数据的采样率。
    //
    //   channelCount:
    //     声道数。
    public static IAudioPlayer CreateAudioPlayer(int speakerIndex, int sampleRate, int channelCount);
    //
    // 摘要:
    //     创建声音播放器。用于播放PCM声音数据。
```

```
        //
        // 参数:
        //     speakerIndex:
        //         使用的扬声器设备的索引。
        //
        //     sampleRate:
        //         声音数据的采样率。
        //
        //     channelCount:
        //         声道数。
        //
        //     bitsNumber:
        //         声音数据的采样位数。一般为16。
        //
        //     bufferSizeInSecs:
        //         缓冲区的大小：最多存储几秒钟的声音数据。默认值为10秒。
        public static IAudioPlayer CreateAudioPlayer(int speakerIndex, int sampleRate, int channelCount, int
bitsNumber, int bufferSizeInSecs);
        //
        // 摘要:
        //         解析WAV文件。
        public static AudioInformation ParseWaveFile(string filePath);
    }
```

（1）在创建声音播放器时，我们必须要清楚接下来所播放的声音数据的性质：采样率、采样位数、声道数。

（2）关于 CreateAudioPlayer 方法的 bufferSizeInSecs 参数的取值：

如果是播放实时采集的数据（比如实时的聊天系统），则 bufferSizeInSecs 设为 2~5 秒即可。

如果是播放从文件中（如 WAV 文件）提取出来的大块声音数据，则 bufferSizeInSecs 就设置为比要播放的声音时长大 1 秒即可。

（3）ParseWaveFile 方法用于解析 WAV 文件，在该方法的支持下，我们就很容易使用 IAudioPlayer 来播放 wav 文件了。

四.MPlayerDemo

下面是一个最简单的使用 MPlayer 组件的 Demo，这个 demo 演示了两个功能：（1）使用 MCapture 采集麦克风的声音，然后使用 MPlayer 播放出来；（2）播放 wav 声音文件。

Demo 运行的截图如下所示：

MPlayerDemo 源码：Oraycn.MPlayerDemo.rar

注：如果采集麦克风并播放，没有听到声音，则可能原因是：

（1）麦克风或扬声器的 index 设置的不正确。

（2）检查麦克风设备的音量是否调为 0 了？