**武汉大学计算机学院**

**本科生实验报告**

**WinUI3音频编辑器**

专 业 名 称 ：人工智能

课 程 名 称 ：软件构造基础

指 导 教 师 ：何政

学 生 学 号 ：2023302061163

学 生 姓 名 ：谢羿衡

二○二五年五月

目录

[1. 系统介绍 3](#_Toc197783332)

[1.1 项目背景 3](#_Toc197783333)

[1.2 系统功能 3](#_Toc197783334)

[1.3 技术栈 3](#_Toc197783335)

[2. 设计思路 4](#_Toc197783336)

[2.1 系统架构 4](#_Toc197783337)

[2.2 模块划分 4](#_Toc197783338)

[2.3 数据流 6](#_Toc197783339)

[3. 关键技术 7](#_Toc197783340)

[3.1 数字音频基本概念 7](#_Toc197783341)

[3.2 音频处理 8](#_Toc197783342)

[3.3 实时特效 10](#_Toc197783343)

[3.4 波形可视化 18](#_Toc197783344)

[3.5 用户界面 20](#_Toc197783345)

[4. 个人贡献 23](#_Toc197783346)

[5. 个人成果展现 24](#_Toc197783347)

[6. 开发体会 24](#_Toc197783348)

[6.1 学习新技术的挑战 24](#_Toc197783349)

[6.2 技术难点的突破 24](#_Toc197783350)

[6.3 开发收获 24](#_Toc197783351)

[6.4 未来展望 25](#_Toc197783352)

1. 系统介绍

### 项目背景

本项目是一个基于 **WinUI 3** 的音频处理工具，旨在为用户提供音频文件的加载、播放、编辑和特效处理功能。随着多媒体技术的快速发展，音频编辑工具的需求日益增加，但许多专业工具对于初学者来说过于复杂。本项目的目标是面向音频编辑初学者和需要快速处理音频的用户，提供一个功能简洁、操作直观的音频处理工具，帮助用户快速完成音频的基本编辑和特效处理任务。

### 系统功能

本系统主要实现以下功能：

* **音频文件的加载与播放**：

支持常见音频格式的加载。

提供播放、暂停、停止等基本播放控制功能。

* **音频特效的添加与调整**：

支持多种音频特效（如混响、均衡器、音量调节等）的添加。

提供实时调整特效参数的功能，并可即时预览效果。

* **波形预览与编辑**：

提供音频波形的可视化预览。

支持波形的缩放、选区操作和简单的剪辑功能。

* **音频导出**：

支持将编辑后的音频文件导出为常见格式。

### 技术栈

本项目的开发过程中使用了以下技术和工具：

* **编程语言**：C#。
* **UI框架**：WinUI 3，用于构建现代化的用户界面。
* **第三方库**：
  + **NAudio**：用于音频文件的加载、播放和处理。
  + **CommunityToolkit.Mvvm**：用于实现 MVVM 模式，简化数据绑定和状态管理。
  + **Win2D**: 用于实现高性能的音频波形绘制。
* **开发环境**：Visual Studio 2022，提供高效的开发和调试支持。

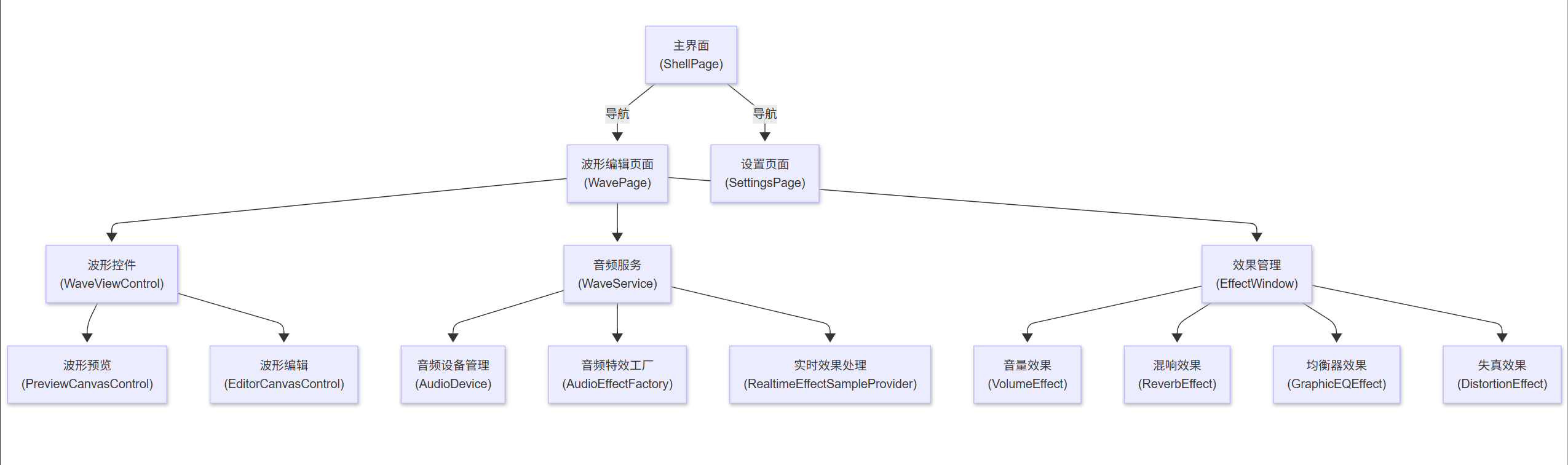
通过这些技术的结合，本系统实现了高效的音频处理能力和直观的用户界面，为用户提供了良好的使用体验。

1. 设计思路

### 系统架构

本系统采用基于MVVM模式的模块化设计，整体架构分为以下四个主要模块：

1. 前端界面：基于 WinUI 3 构建，提供直观的用户操作界面，包括音频文件的加载、播放控制、特效设置和波形可视化等功能。
2. 音频处理模块：使用 NAudio 实现音频文件的加载、播放、暂停、停止、特效处理和导出功能。
3. 特效模块：负责实现音频特效的添加、调整和实时处理，支持多种常见的音频特效。
4. 数据交互层：通过 MVVM 模式实现前端界面与音频处理模块之间的数据绑定和交互，确保模块之间的解耦和高效通信。



### 模块划分

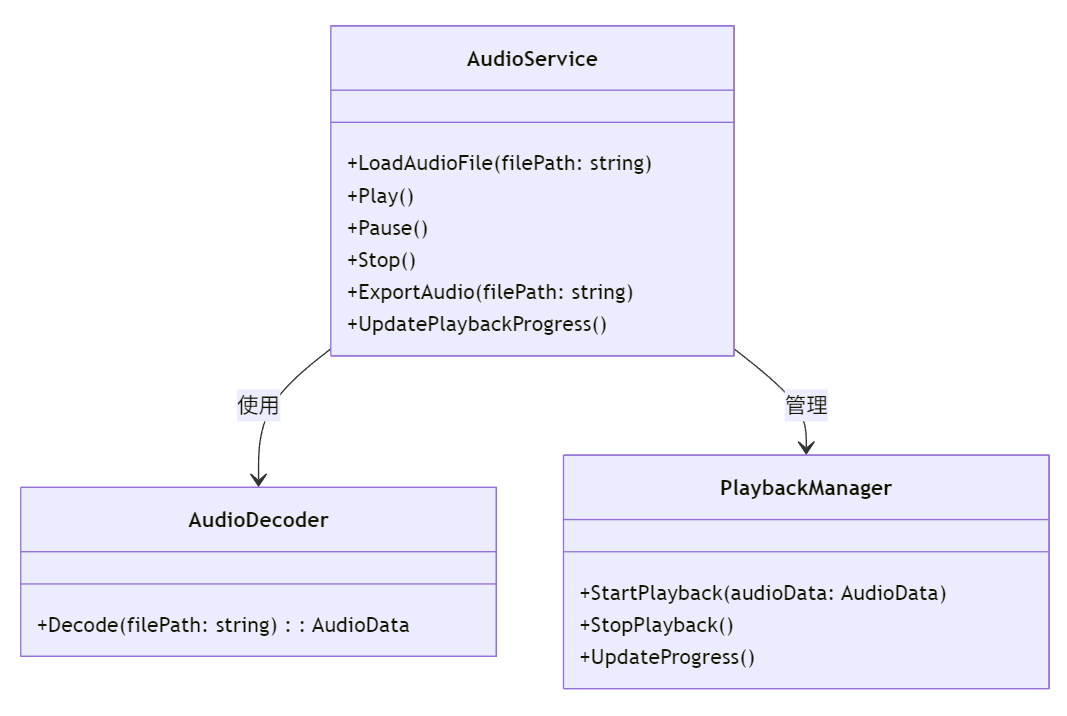
系统主要划分为以下功能模块：

1. 音频服务模块：

负责音频文件的加载、播放、暂停、停止和导出。

提供对音频文件格式的支持和解码功能。

实现音频播放进度的实时更新。

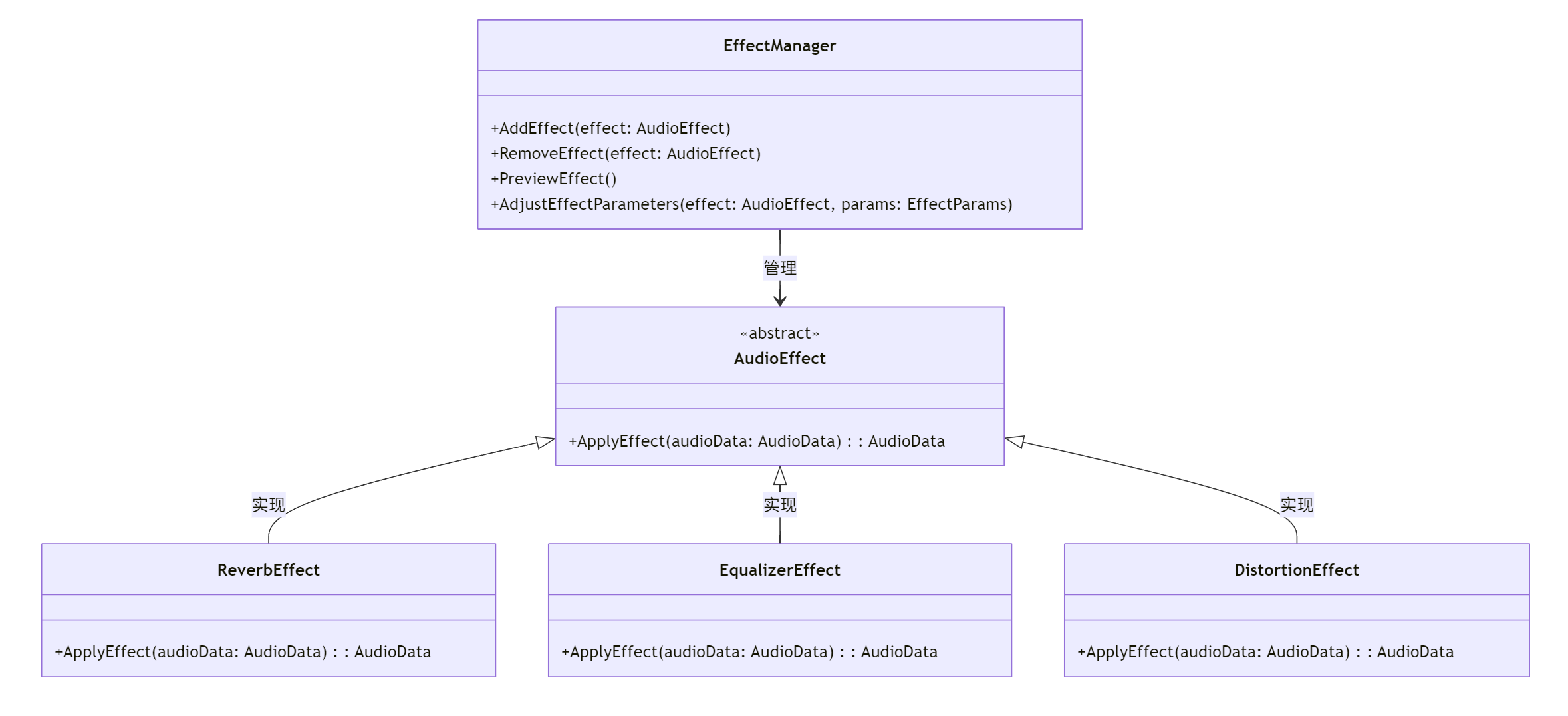


1. 特效模块：

实现音频特效的添加和实时处理。

支持特效参数的动态调整，例如混响强度、均衡器频段等。

提供特效预览功能，用户可以实时听到调整后的效果。

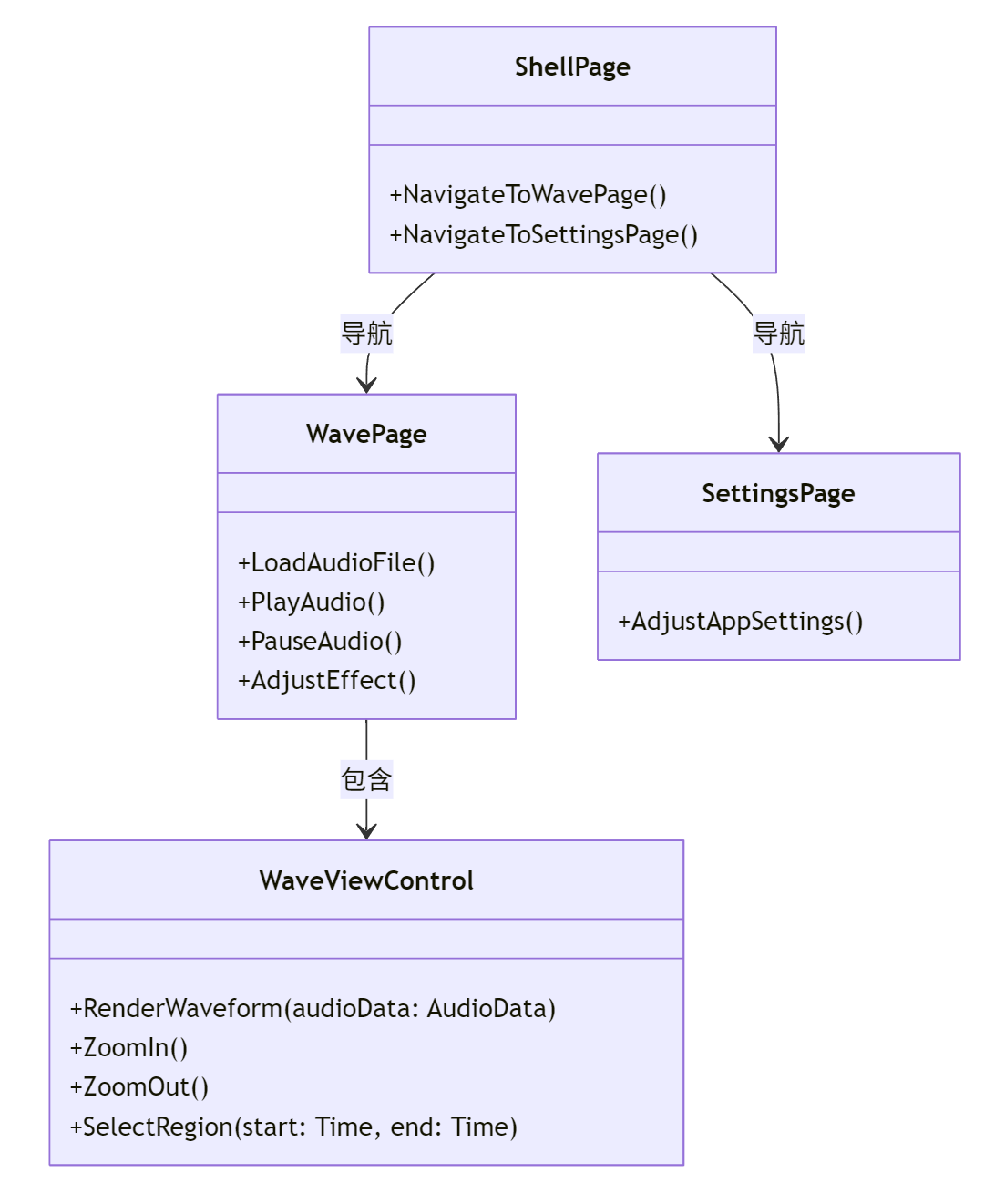


1. 用户界面模块：

提供直观的操作界面，用户可以通过按钮、滑块等控件完成音频的加载、播放和特效调整。

实现音频波形的可视化，支持波形的缩放、选区操作和简单的剪辑功能。

提供播放进度条和特效参数的实时显示。

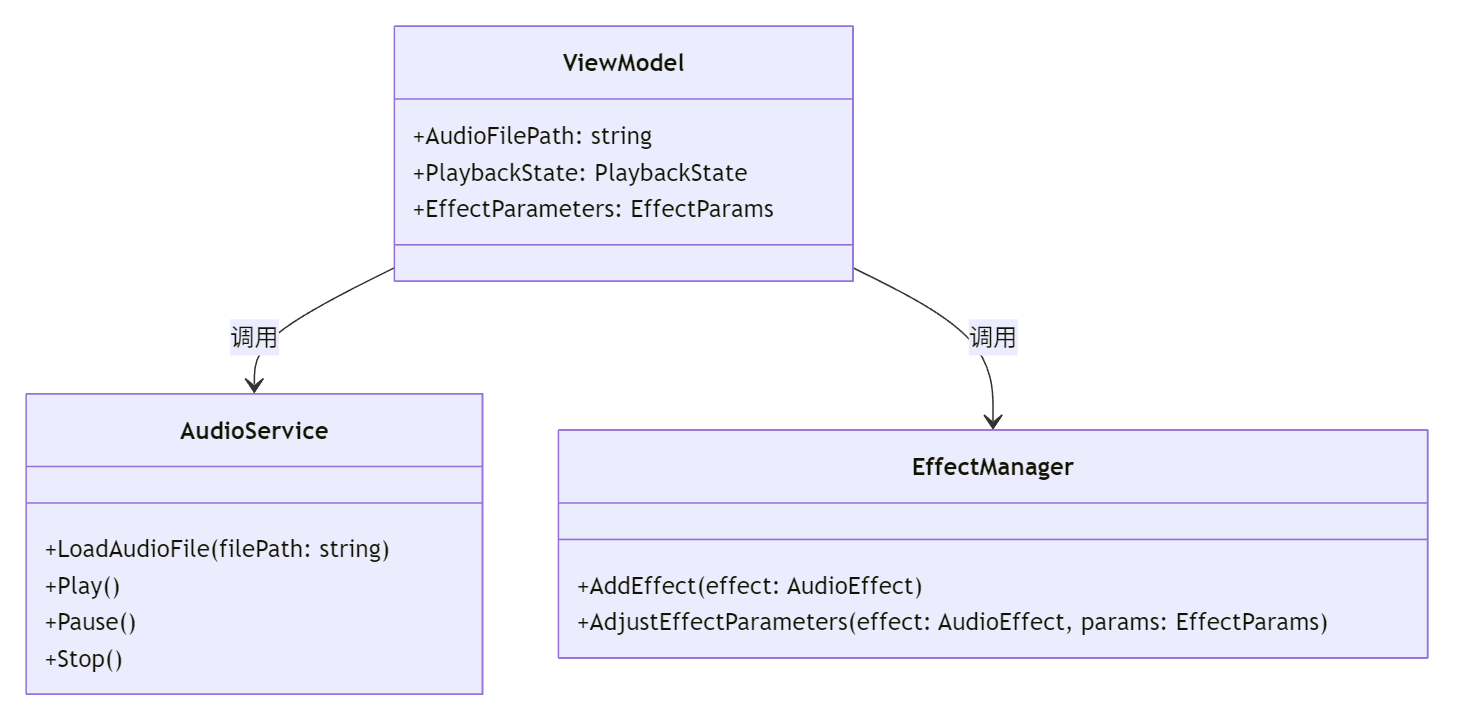


1. 数据交互模块：

使用 CommunityToolkit.Mvvm 实现 MVVM 模式。

负责前端界面与后台逻辑的双向数据绑定。

确保用户操作能够实时反映到音频处理模块，并将处理结果反馈到界面。



### 数据流

1. 音频文件加载：

用户通过界面选择音频文件，音频服务模块负责加载文件并解析其格式。

加载完成后，音频数据被传递到波形可视化模块进行波形绘制。

1. 音频播放与特效处理：

用户点击播放按钮后，音频服务模块开始播放音频数据。

用户添加特效后，特效模块会对音频数据进行实时处理，并将处理后数据传给播放模块。

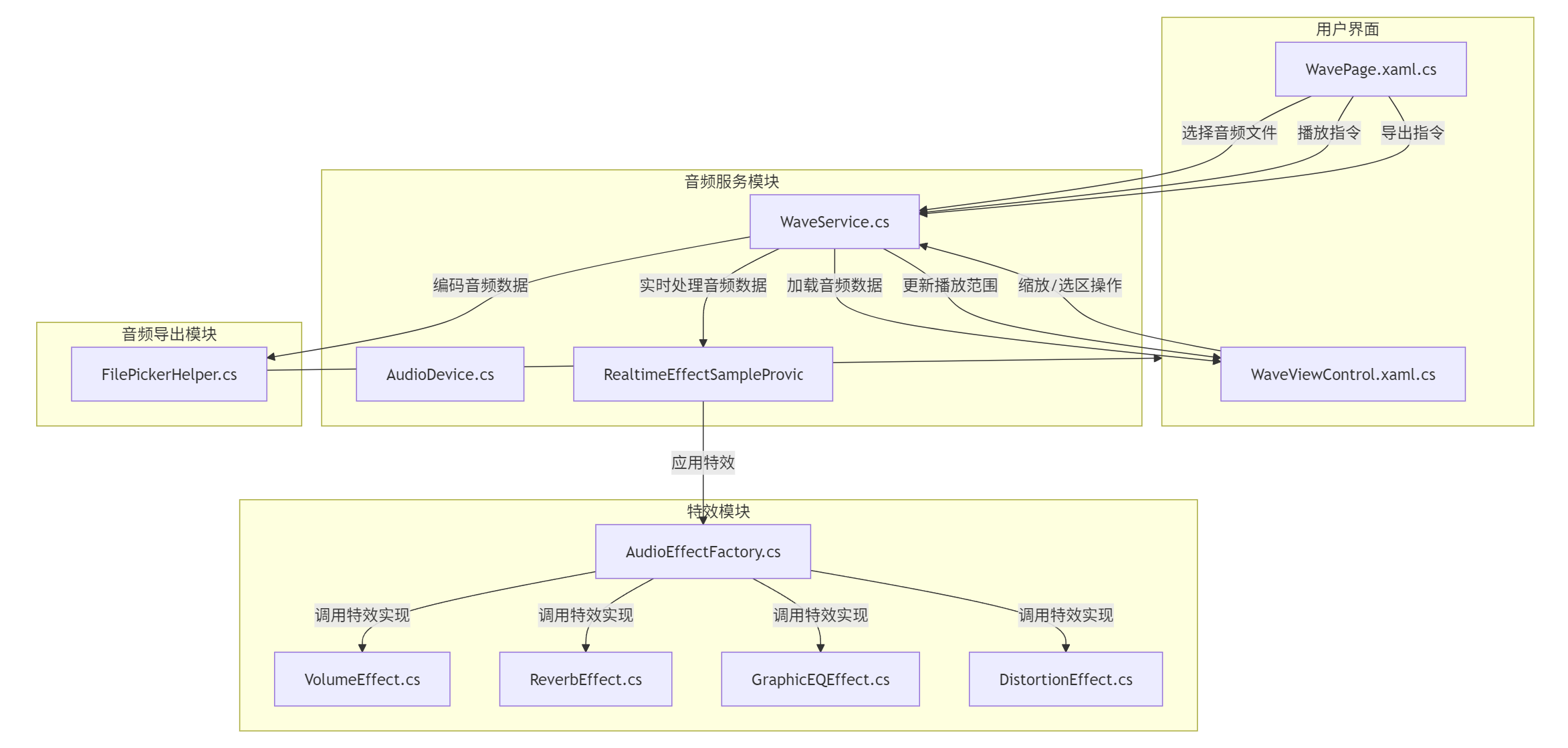
1. 波形交互：

用户可以通过界面对波形进行缩放、选区操作或剪辑。

选区信息会传递到音频服务模块，用于更新播放范围或剪辑结果。

1. 音频导出：

用户完成编辑后，可以通过界面选择导出功能，将音频保存到本地。



通过以上设计，系统实现了模块之间的高效协作，确保了功能的完整性和用户体验的流畅性。

1. 关键技术

这部分是该系统的核心技术分析与相关代码展示。为保持报告内容的简练，部分代码进行了简化处理。我会在每个代码块下面标注代码源文件名称以供详细参考。

### 数字音频基本概念

在进入具体的技术实现之前，我们需要了解一些数字音频的基本概念，这将有助于理解后续的音频处理、特效和波形可视化的实现。

* + 1. 什么是数字音频

数字音频是通过对模拟音频信号进行采样和量化得到的离散数据。它是计算机处理音频的基础。以下是数字音频的几个核心概念：

* 1. **采样率** (Sample Rate)

采样率是指每秒对音频信号采样的次数，单位是 Hz。例如，44.1 kHz 表示每秒采样 44100 次。采样率越高，音频的还原度越高，但文件体积也会增大。

* 1. **位深度** (Bit Depth)

位深度是指每个采样点使用的比特数，决定了音频的动态范围。例如，16 位音频可以表示的动态范围是 96 dB，而 24 位音频可以达到 144 dB。

* 1. **声道** (Channels)

音频可以是单声道 (Mono) 或立体声 (Stereo)。立体声包含两个声道，分别对应左声道和右声道。

* 1. **PCM** 数据

PCM (Pulse Code Modulation) 是最常见的数字音频格式，表示为一系列采样点的振幅值。PCM 数据通常存储为浮点数或整数数组。

* + 1. 数字音频的存储与处理

在计算机中，数字音频通常以文件的形式存储，常见的格式包括 **WAV**、**MP3**、**FLAC** 等。

* + **WAV**：一种无损音频格式，通常使用 PCM 编码，保留了音频的完整信息，适合高质量音频处理。
  + **MP3**：一种有损压缩格式，通过去除人耳不敏感的频率信息来减小文件体积，适合存储和传输。
  + **FLAC**：一种无损压缩格式，兼顾了高质量和较小的文件体积。

PCM (Pulse Code Modulation) 是数字音频的基础，表示为一系列采样点的**振幅**值。PCM 数据通常存储为**浮点数或整数数组**。

* + **16 位 PCM**：每个采样点使用 16 位整数表示，范围为 -32768 到 32767。
  + **32 位浮点 PCM**：每个采样点使用浮点数表示，范围为 -1.0 到 1.0。

在音频处理软件中，加载音频文件后，通常会将其转换为 PCM 数据，以便进行后续的处理和可视化。该软件采用的格式是32位浮点PCM。从而便于进行可视化与高精度处理。

### 音频处理

音频处理是本系统的核心功能之一，主要包括音频文件的加载、播放、暂停、停止，以及音频数据的基本操作（如剪辑、混合等）。在本项目中，音频处理功能由 WaveService 模块实现，基于 NAudio 库完成。

* + 1. 音频文件的加载

用户通过界面选择音频文件，音频服务模块负责加载文件并解析其格式。

|  |  |
| --- | --- |
|  | public async Task<AudioFile> OpenAsync(string filePath) |
|  | { |
|  | // 将音频文件转换为 32 位 PCM 格式 |
|  | var cachedPath = await ConvertToPcm32(filePath); |
|  |  |
|  | // 创建音频文件读取器并提取元数据 |
|  | var convertedAudioReader = new AudioFileReader(cachedPath); |
|  | var convertedAudioFile = new AudioFile |
|  | { |
|  | FilePath = cachedPath, |
|  | FileName = System.IO.Path.GetFileName(filePath), |
|  | Duration = convertedAudioReader.TotalTime, |
|  | SampleRate = convertedAudioReader.WaveFormat.SampleRate, |
|  | Channels = convertedAudioReader.WaveFormat.Channels, |
|  | BitDepth = convertedAudioReader.WaveFormat.BitsPerSample, |
|  | Format = "PCM 32-bit", |
|  | }; |
|  |  |
|  | // 加载完整音频数据 |
|  | convertedAudioFile.AudioData = await LoadWaveAsync(cachedPath); |
|  |  |
|  | return convertedAudioFile; |
|  | } |

DAW.Wave\Services\Implementations\WaveService.cs

[ConvertToPcm32](vscode-file://vscode-app/d:/software/tool/MicrosoftVSCode/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) 方法将音频文件转换为标准的 32 位 PCM 格式。

[AudioFileReader](vscode-file://vscode-app/d:/software/tool/MicrosoftVSCode/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) 用于读取音频文件的元数据和音频数据。

[LoadWaveAsync](vscode-file://vscode-app/d:/software/tool/MicrosoftVSCode/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) 异步加载音频数据到内存中。

* + 1. 音频播放与控制

用户点击播放按钮后，音频服务模块开始播放音频数据。如果用户添加了特效，特效模块会对音频数据进行实时处理（效果器部分的代码在下一节展示）。

|  |  |
| --- | --- |
|  | using NAudio.Wave; |
|  |  |
|  | public class WaveService |
|  | { |
|  | private IWavePlayer? wavePlayer; |
|  | private WaveStream? waveStream; |
|  |  |
|  | // 播放音频 |
|  | public void Play() |
|  | { |
|  | if (waveStream == null) |
|  | { |
|  | throw new InvalidOperationException("音频文件未加载"); |
|  | } |
|  |  |
|  | wavePlayer ??= new WaveOutEvent(); // 使用 WaveOutEvent 播放 |
|  | wavePlayer.Init(waveStream); |
|  | wavePlayer.Play(); |
|  | } |
|  |  |
|  | // 暂停音频 |
|  | public void Pause() |
|  | { |
|  | wavePlayer?.Pause(); |
|  | } |
|  |  |
|  | // 停止音频 |
|  | public void Stop() |
|  | { |
|  | wavePlayer?.Stop(); |
|  | waveStream?.Position = 0; // 重置播放位置 |
|  | } |
|  | } |

DAW.Wave\Services\Implementations\WaveService.cs

WaveOutEvent：NAudio 提供的音频播放类，支持异步播放，适合 GUI 应用程序。

Init：初始化播放器，将音频流绑定到播放器。

Play/Pause/Stop：分别实现播放、暂停和停止功能。

重置播放位置：在停止音频时，将音频流的位置重置为起点。

* + 1. 音频剪辑

音频剪辑是音频处理的重要功能之一。用户可以在波形界面拖拽鼠标创建选区。按下backspace键即可删除这部分的音频。

|  |  |
| --- | --- |
|  | public void ClipAudio(AudioFile audioFile, long startFrame, long endFrame) |
|  | { |
|  | long framesToClipCount = endFrame - startFrame + 1; |
|  | long samplesToClipCount = framesToClipCount \* audioFile.Channels; |
|  | long startSampleToClipIndex = startFrame \* audioFile.Channels; |
|  |  |
|  | float[] originalData = audioFile.AudioData; |
|  | int newAudioDataLength = originalData.Length - (int)samplesToClipCount; |
|  | if (newAudioDataLength < 0) newAudioDataLength = 0; |
|  |  |
|  | float[] newAudioData = new float[newAudioDataLength]; |
|  | Array.Copy(originalData, 0, newAudioData, 0, (int)startSampleToClipIndex); |
|  | long originalSourceIndexAfterClip = startSampleToClipIndex + samplesToClipCount; |
|  | if (originalSourceIndexAfterClip < originalData.Length) |
|  | { |
|  | Array.Copy(originalData, (int)originalSourceIndexAfterClip, |
|  | newAudioData, (int)startSampleToClipIndex, |
|  | originalData.Length - (int)originalSourceIndexAfterClip); |
|  | } |
|  |  |
|  | long oldPlaybackPos = audioFile.PlaybackPositionFrameIndex; |
|  | audioFile.AudioData = newAudioData; |
|  | } |

### 实时特效

实时特效是音频处理中的重要部分，通过对音频数据进行实时处理，可以动态调整音频的音质、空间感和表现力。本系统实现了一个灵活的，易扩展的效果器模块框架，并基于该框架实现了四种基本音频特效，包括音量调节、混响、均衡器和失真。

* + 1. 效果器模块框架

为了实现实时特效的灵活性和扩展性，本系统设计了一个模块化的效果器框架。该框架的核心包括以下几个部分：

1. **IAudioEffect 接口**
   * 定义了所有效果器的通用行为，包括启用状态、名称以及音频样本的处理方法。
   * 通过接口的抽象设计，系统可以轻松扩展新的效果器。
2. **AudioEffectFactory 工厂类**
   * 提供了动态创建效果器实例的功能。
   * 工厂模式的使用使得效果器的创建与使用解耦，便于管理和扩展。
3. **RealtimeEffectSampleProvider 实时处理链**
   * 负责将多个效果器串联起来，对音频数据进行实时处理。
   * 支持动态更新效果器链，确保用户在运行时调整效果器参数时能够即时生效。
4. **效果器管理**
   * 在 [WaveService](vscode-file://vscode-app/d:/software/tool/MicrosoftVSCode/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) 中实现了对效果器的添加、移除和实时更新的管理功能。
5. **IAudioEffect 接口：**

|  |  |
| --- | --- |
|  | public interface IAudioEffect |
|  | { |
|  | bool Enabled { get; set; } // 是否启用特效 |
|  | string Name { get; }       // 特效名称 |
|  |  |
|  | // 处理音频样本 |
|  | void ProcessSamples(float[] buffer, int offset, int count); |
|  | } |

DAW.Wave\Services\IAudioEffect.cs

Enabled：用于控制特效的启用或禁用状态。

Name：标识特效的名称，便于管理和显示。

ProcessSamples：核心方法，用于对音频样本进行处理。所有效果器都需要实现该方法。

1. **AudioEffectFactory 工厂类：**

|  |  |
| --- | --- |
|  | public class AudioEffectFactory |
|  | { |
|  | public IAudioEffect? CreateEffect(string effectName, int sampleRate) |
|  | { |
|  | return effectName switch |
|  | { |
|  | "Volume" => new VolumeEffect(), |
|  | "Reverb" => new ReverbEffect(sampleRate), |
|  | "Graphic EQ" => new GraphicEQEffect(sampleRate), |
|  | "Distortion" => new DistortionEffect(), |
|  | \_ => null |
|  | }; |
|  | } |
|  | } |

DAW.Wave\Services\Effects\AudioEffectFactory.cs

动态创建：通过 switch 语句，根据特效名称动态创建对应的效果器实例。

扩展性：如果需要添加新的效果器，只需在工厂类中增加对应的分支即可。

1. **RealtimeEffectSampleProvider 实时效果器链**

|  |  |
| --- | --- |
|  | public class RealtimeEffectSampleProvider : ISampleProvider |
|  | { |
|  | private readonly ISampleProvider \_source; |
|  | private IList<IAudioEffect> \_effects; |
|  |  |
|  | public RealtimeEffectSampleProvider(ISampleProvider source, IList<IAudioEffect> effects) |
|  | { |
|  | \_source = source; |
|  | \_effects = effects; |
|  | } |
|  |  |
|  | public void UpdateEffects(IList<IAudioEffect> newEffects) |
|  | { |
|  | \_effects = newEffects; // 动态更新效果器链 |
|  | } |
|  |  |
|  | public int Read(float[] buffer, int offset, int count) |
|  | { |
|  | // 从源读取音频数据 |
|  | int samplesRead = \_source.Read(buffer, offset, count); |
|  |  |
|  | // 依次应用效果器 |
|  | foreach (var effect in \_effects) |
|  | { |
|  | if (effect.Enabled) |
|  | { |
|  | effect.ProcessSamples(buffer, offset, samplesRead); |
|  | } |
|  | } |
|  |  |
|  | return samplesRead; |
|  | } |
|  | } |

DAW.Wave\Services\Effects\RealtimeEffectSampleProvider.cs

\_source：音频数据的源，可以是文件、流或其他音频输入。

\_effects：效果器链，包含所有需要应用的效果器。

UpdateEffects：支持动态更新效果器链，用户可以在运行时调整效果器的启用状态或参数。

Read：从音频源读取数据后，依次应用效果器链中的每个效果器。

1. **效果器管理**

|  |  |
| --- | --- |
|  | public void AddEffect(AudioFile audioFile, string effectName) |
|  | { |
|  | var effect = \_audioEffectFactory.CreateEffect(effectName, audioFile.SampleRate); |
|  | if (effect == null) return; |
|  |  |
|  | audioFile.AudioEffects.Add(effect); |
|  |  |
|  | if (\_realtimeProviders.TryGetValue(audioFile, out var rp)) |
|  | { |
|  | rp.UpdateEffects(audioFile.AudioEffects); |
|  | } |
|  | } |

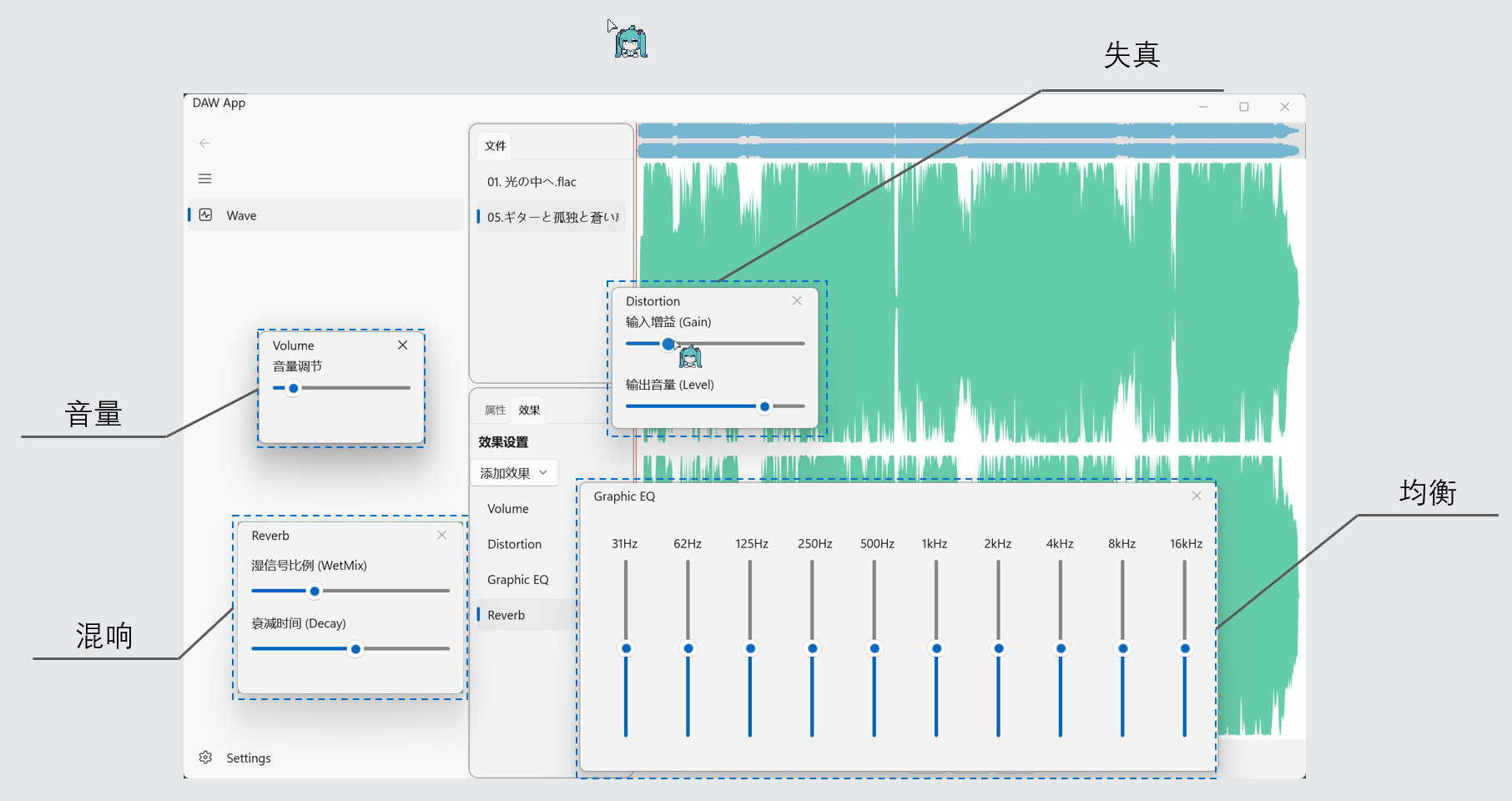
DAW.Wave\Services\Implementations\WaveService.cs

AddEffect：通过工厂类创建效果器实例，并将其添加到音频文件的效果器列表中。

实时更新：如果音频正在播放，调用 UpdateEffects 方法更新实时处理链，确保新添加的效果器立即生效。

* + 1. 效果器实现

在效果器模块框架的基础上，本系统实现了多种常见的音频特效，包括音量调节、混响、均衡器和失真，且每种效果器都有对应的图形界面。以下是每种效果器的详细介绍。



1. **音量（VolumeEffect）**

**原理**：通过对音频样本乘以一个比例因子来调整音量。

**作用**：用于增大或减小音频的整体响度。

**实现特点**：

简单高效，直接对样本值进行线性缩放。

支持动态调整音量参数。

|  |  |
| --- | --- |
|  | public class VolumeEffect : IAudioEffect |
|  | { |
|  | public bool Enabled { get; set; } = true; // 是否启用特效 |
|  | public string Name { get; set; } = "Volume"; // 特效名称 |
|  | public float Volume { get; set; } = 1.0f; // 默认音量为 1.0 (100%) |
|  |  |
|  | public void ProcessSamples(float[] buffer, int offset, int count) |
|  | { |
|  | for (int i = 0; i < count; i++) |
|  | { |
|  | // 调整音量 |
|  | var sample = buffer[offset + i] \* Volume; |
|  | // 防止溢出 |
|  | buffer[offset + i] = Math.Clamp(sample, -1.0f, 1.0f); |
|  | } |
|  | } |
|  | } |

DAW.Wave\Services\Effects\VolumeEffect.cs

1. **混响（ReverbEffect）**

**原理**：通过延迟和反馈模拟声音在不同环境中的反射效果。

**作用**：增加空间感，使声音更自然。

**实现特点**：

使用多个延迟线模拟不同反射路径。

支持调整湿信号比例（WetMix）和衰减系数（Decay）。

|  |  |
| --- | --- |
|  | public class ReverbEffect : IAudioEffect |
|  | { |
|  | public bool Enabled { get; set; } = true; // 是否启用特效 |
|  | public string Name { get; set; } = "Reverb"; // 特效名称 |
|  |  |
|  | private readonly int[] \_delayTimesMs = { 29, 37, 41, 53 }; // 延迟时间（毫秒） |
|  | private readonly float[][] \_delayLines; // 延迟线 |
|  | private readonly int[] \_positions; // 当前延迟线位置 |
|  | private readonly int \_sampleRate; |
|  |  |
|  | public float WetMix { get; set; } = 0.3f; // 湿信号比例 |
|  | public float Decay { get; set; } = 0.5f; // 衰减系数 |
|  |  |
|  | public ReverbEffect(int sampleRate) |
|  | { |
|  | \_sampleRate = sampleRate; |
|  | \_delayLines = new float[\_delayTimesMs.Length][]; |
|  | \_positions = new int[\_delayTimesMs.Length]; |
|  |  |
|  | // 初始化延迟线 |
|  | for (int i = 0; i < \_delayTimesMs.Length; i++) |
|  | { |
|  | int delaySamples = (int)(\_delayTimesMs[i] \* sampleRate / 1000f); |
|  | \_delayLines[i] = new float[delaySamples]; |
|  | \_positions[i] = 0; |
|  | } |
|  | } |
|  |  |
|  | public void ProcessSamples(float[] buffer, int offset, int count) |
|  | { |
|  | if (!Enabled || WetMix == 0) return; |
|  |  |
|  | for (int i = 0; i < count; i++) |
|  | { |
|  | float input = buffer[offset + i]; |
|  | float reverbSample = 0f; |
|  |  |
|  | for (int d = 0; d < \_delayLines.Length; d++) |
|  | { |
|  | var delayLine = \_delayLines[d]; |
|  | int pos = \_positions[d]; |
|  |  |
|  | float delayed = delayLine[pos]; |
|  | reverbSample += delayed; |
|  |  |
|  | // 反馈混响 |
|  | delayLine[pos] = input + delayed \* Decay; |
|  |  |
|  | \_positions[d] = (pos + 1) % delayLine.Length; |
|  | } |
|  |  |
|  | reverbSample /= \_delayLines.Length; |
|  |  |
|  | // 混合干湿信号 |
|  | buffer[offset + i] = input \* (1 - WetMix) + reverbSample \* WetMix; |
|  | } |
|  | } |
|  | } |

DAW.Wave\Services\Effects\ReverbEffect.cs

1. **均衡器（GraphicEQEffect）**

**原理**：通过滤波器调整不同频段的增益。

**作用**：用于增强或削弱特定频率的声音。

**实现特点**：

使用 10 个带通滤波器覆盖从低频到高频的频段。

支持动态调整每个频段的增益。

|  |  |
| --- | --- |
|  | public class GraphicEQEffect : IAudioEffect |
|  | { |
|  | public bool Enabled { get; set; } = true; // 是否启用特效 |
|  | public string Name { get; set; } = "Graphic EQ"; // 特效名称 |
|  |  |
|  | private readonly BiQuadFilter[] \_filters = new BiQuadFilter[10]; // 滤波器数组 |
|  | private readonly float[] \_gains = new float[10]; // 每个频段的增益 |
|  |  |
|  | private readonly float[] \_frequencies = new float[] |
|  | { |
|  | 31.25f, 62.5f, 125f, 250f, 500f, |
|  | 1000f, 2000f, 4000f, 8000f, 16000f |
|  | }; |
|  |  |
|  | public float this[int band] |
|  | { |
|  | get => \_gains[band]; |
|  | set |
|  | { |
|  | \_gains[band] = value; |
|  | \_filters[band] = BiQuadFilter.PeakingEQ(sampleRate, \_frequencies[band], 1.0f, value); |
|  | } |
|  | } |
|  |  |
|  | private readonly int sampleRate; |
|  |  |
|  | public GraphicEQEffect(int sampleRate) |
|  | { |
|  | this.sampleRate = sampleRate; |
|  | for (int i = 0; i < 10; i++) |
|  | { |
|  | \_gains[i] = 0f; // 默认增益为 0 dB |
|  | \_filters[i] = BiQuadFilter.PeakingEQ(sampleRate, \_frequencies[i], 1.0f, 0f); |
|  | } |
|  | } |
|  |  |
|  | public void ProcessSamples(float[] buffer, int offset, int count) |
|  | { |
|  | for (int i = 0; i < count; i++) |
|  | { |
|  | float sample = buffer[offset + i]; |
|  | for (int b = 0; b < 10; b++) |
|  | { |
|  | sample = \_filters[b].Transform(sample); |
|  | } |
|  | buffer[offset + i] = sample; |
|  | } |
|  | } |
|  | } |

DAW.Wave\Services\Effects\GraphicEQEffect.cs

1. **失真（DistortionEffect）**

**原理**：通过对音频样本进行非线性处理（如剪切）来改变波形。

**作用**：用于创造粗糙或尖锐的声音效果。

**实现特点**：

支持输入增益（Gain）和输出音量（Level）的调整。

使用 soft clipping 限制样本值，避免过度失真。

|  |  |
| --- | --- |
|  | public class DistortionEffect : IAudioEffect |
|  | { |
|  | public bool Enabled { get; set; } = true; // 是否启用特效 |
|  | public string Name { get; set; } = "Distortion"; // 特效名称 |
|  |  |
|  | public float Gain { get; set; } = 5.0f; // 输入增益 |
|  | public float Level { get; set; } = 0.8f; // 输出音量 |
|  |  |
|  | public void ProcessSamples(float[] buffer, int offset, int count) |
|  | { |
|  | for (int i = 0; i < count; i++) |
|  | { |
|  | // 应用增益 |
|  | float sample = buffer[offset + i] \* Gain; |
|  |  |
|  | // 简单的 soft clipping |
|  | if (sample > 1.0f) |
|  | sample = 1.0f; |
|  | else if (sample < -1.0f) |
|  | sample = -1.0f; |
|  |  |
|  | // 调整输出音量 |
|  | buffer[offset + i] = sample \* Level; |
|  | } |
|  | } |
|  | } |

DAW.Wave\Services\Effects\DistortionEffect.cs

通过模块化的设计，本系统实现了一个灵活且高效的实时特效框架。该框架不仅支持多种常见的音频特效，还具有良好的扩展性，便于添加新的效果器。实时处理链的设计确保了用户在运行时调整效果器参数时能够即时生效，为用户提供了良好的使用体验。

### 波形可视化

波形可视化是音频编辑软件的重要功能之一，通过将音频数据以波形的形式呈现，用户可以直观地了解音频的结构和内容。本系统实现了一个高效的波形可视化模块，支持波形的缩放、选区操作和实时更新。

波形是音频信号的振幅随时间变化的图形表示。波形的绘制基于音频的 PCM 数据，每个采样点表示音频信号在某一时刻的振幅值。

由于音频文件的采样率通常较高（如 44100 Hz），直接绘制所有采样点会导致性能问题。因此，需要对采样点进行分组，每组计算最小值和最大值，用于绘制波形。

为实现上述需求，本系统的波形可视化功能由自定义控件： [WaveViewControl](vscode-file://vscode-app/d:/software/tool/MicrosoftVSCode/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) 模块实现。

* + 1. 波形绘制

|  |  |
| --- | --- |
|  | private void WavePreview\_DrawWave(CanvasControl sender, CanvasDrawEventArgs args) |
|  | { |
|  | if (AudioData == null || AudioData.Length < 1 || Channels < 1) return; |
|  |  |
|  | float canvasWidth = (float)sender.ActualWidth; |
|  | float canvasHeight = (float)sender.ActualHeight; |
|  |  |
|  | // 计算每个像素对应的采样点数 |
|  | int samplesPerPixel = (int)(AudioData.Length / canvasWidth); |
|  |  |
|  | // 遍历每个像素，计算波形的最小值和最大值 |
|  | for (int x = 0; x < canvasWidth; x++) |
|  | { |
|  | int start = x \* samplesPerPixel; |
|  | int end = Math.Min(start + samplesPerPixel, AudioData.Length); |
|  |  |
|  | float minVal = float.MaxValue; |
|  | float maxVal = float.MinValue; |
|  |  |
|  | for (int i = start; i < end; i++) |
|  | { |
|  | float sample = AudioData[i]; |
|  | if (sample < minVal) minVal = sample; |
|  | if (sample > maxVal) maxVal = sample; |
|  | } |
|  |  |
|  | // 绘制波形 |
|  | args.DrawingSession.DrawLine(x, canvasHeight / 2 - minVal \* canvasHeight / 2, |
|  | x, canvasHeight / 2 - maxVal \* canvasHeight / 2, |
|  | Colors.SkyBlue); |
|  | } |
|  | } |

DAW\Controls\WaveViewControl.xaml.cs

采样点分组：通过 samplesPerPixel 计算每个像素对应的采样点数，减少绘制的点数以提高性能。

最小值和最大值：每组采样点计算最小值和最大值，用于绘制波形的上下边界。

绘制波形：使用 DrawLine 方法在画布上绘制每个像素的波形线段。

* + 1. 波形选区

|  |  |
| --- | --- |
|  | private void OnPointerPressed(object sender, PointerRoutedEventArgs e) |
|  | { |
|  | var position = e.GetCurrentPoint(this).Position; |
|  | SelectedLeftSample = (long)(position.X / ActualWidth \* (VisibleRightFrame - VisibleLeftFrame)) + VisibleLeftFrame; |
|  | } |
|  |  |
|  | private void OnPointerReleased(object sender, PointerRoutedEventArgs e) |
|  | { |
|  | var position = e.GetCurrentPoint(this).Position; |
|  | SelectedRightSample = (long)(position.X / ActualWidth \* (VisibleRightFrame - VisibleLeftFrame)) + VisibleLeftFrame; |
|  |  |
|  | // 确保选区范围有效 |
|  | if (SelectedLeftSample > SelectedRightSample) |
|  | { |
|  | (SelectedLeftSample, SelectedRightSample) = (SelectedRightSample, SelectedLeftSample); |
|  | } |
|  |  |
|  | InvalidateCanvas(); |
|  | } |

DAW\Controls\WaveViewControl.xaml.cs

选区起点：在鼠标按下时记录选区的起始采样点。

选区终点：在鼠标释放时记录选区的结束采样点。

选区范围校正：确保选区的起点小于终点。

实时更新：通过 InvalidateCanvas 方法刷新画布，显示选区。

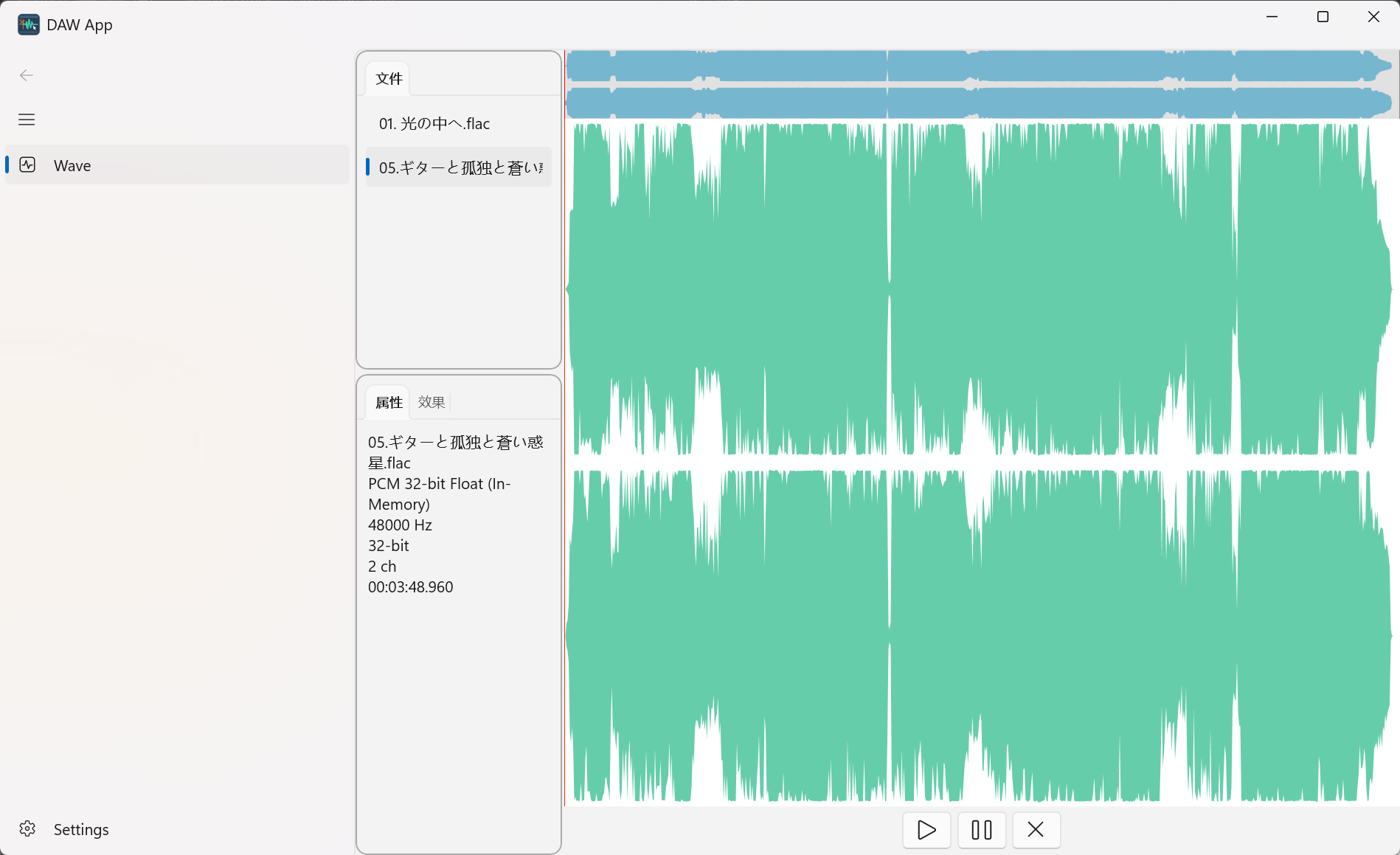
波形可视化模块通过高效的绘制算法和灵活的交互设计，为用户提供了直观的音频编辑体验。无论是查看音频的整体结构，还是精确选择音频片段，波形可视化都发挥了重要作用。

### 用户界面

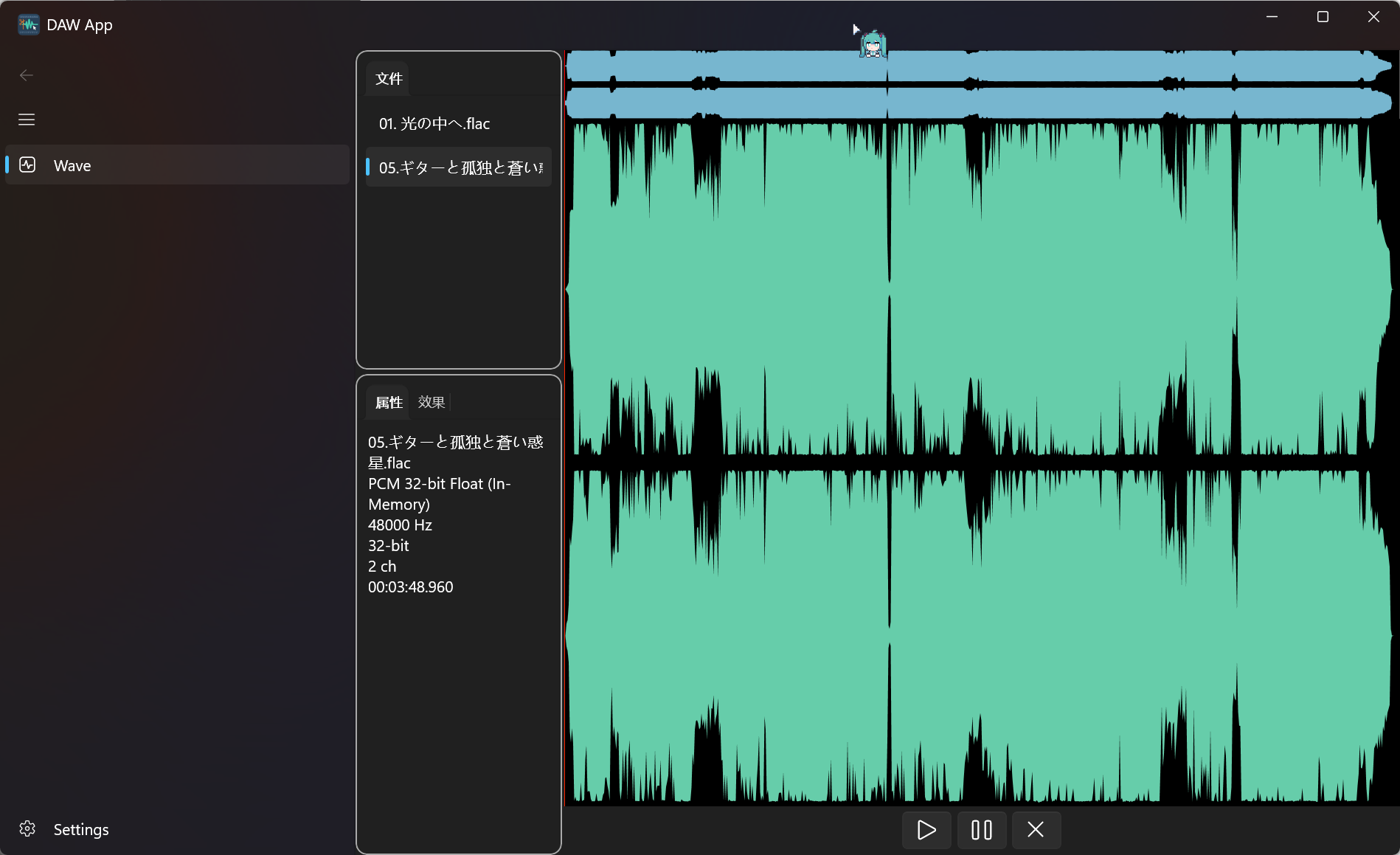
用户界面是软件与用户交互的桥梁，其设计直接影响用户的操作体验和效率。本系统基于 **WinUI 3** 框架构建用户界面，其最大的特色之一就是现代化的用户界面设计能力。作为微软最新的用户界面框架，WinUI 3 提供了丰富的控件、灵活的布局系统以及强大的数据绑定功能，使得开发者能够快速构建直观、美观且高效的用户界面。

用户界面共实现了两种主题（浅色与深色），与用户的系统主题相同。并且支持应用内主题更新

* + 1. 主界面

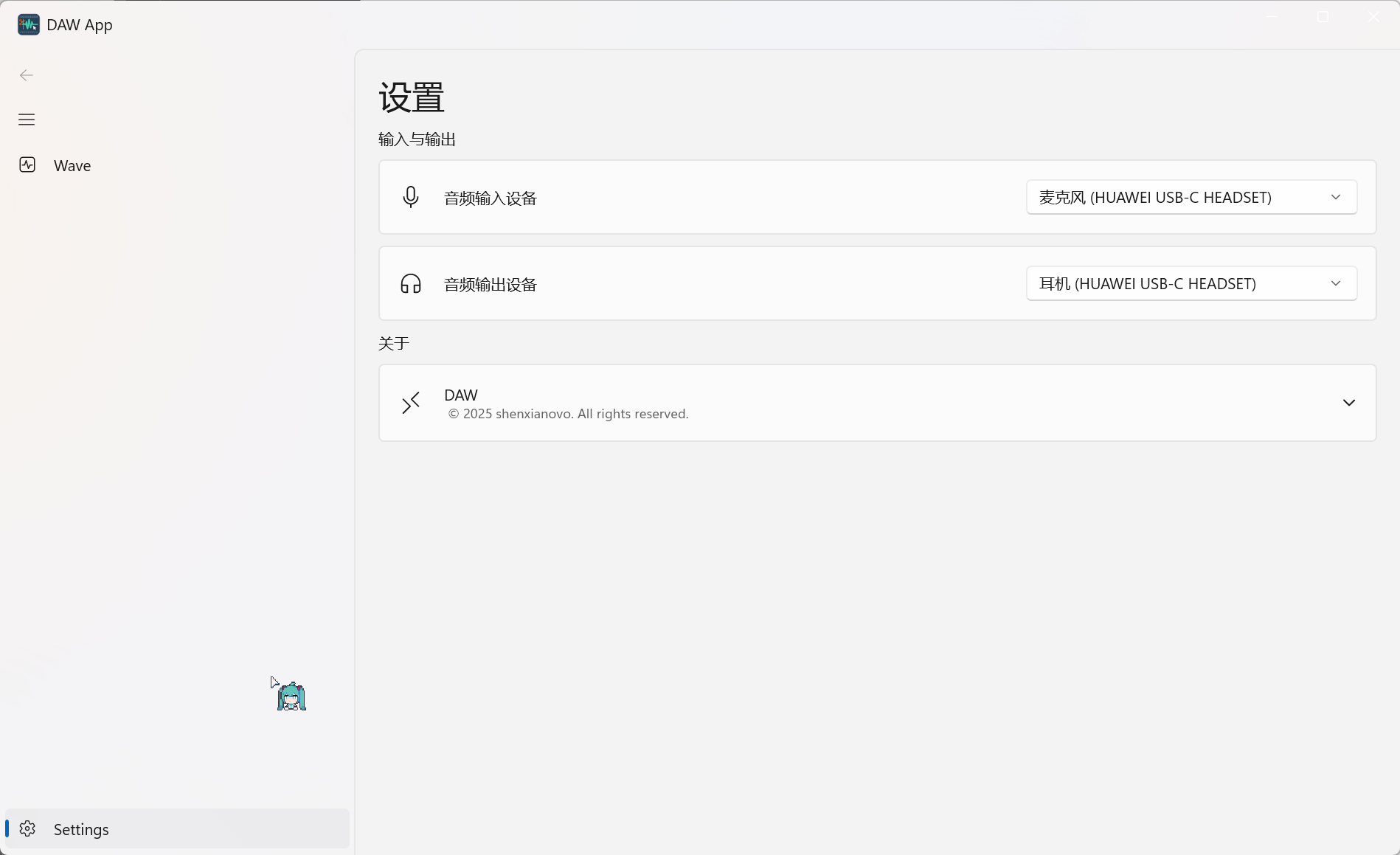


主界面（浅色模式）



主界面（深色模式）

* + 1. 设置界面



设置界面（浅色模式）

设置界面（深色模式）

* + 1. 视觉层次结构

视觉层次结构是 WinUI 3 界面设计的一大特色，通过合理的视觉分层和设计规范，用户界面能够更加直观、现代化且易于操作。

WinUI 3 的视觉层次结构严格遵循 Fluent Design 规范，这些设计规范不仅提升了界面的美观性，还增强了用户的操作体验。WinUI 3 官方文档将设计规范放在开发文档的前面，强调了设计在整个开发流程中的重要性。

在本项目中，视觉层次结构的设计主要体现在以下几个方面：

**1. MicaAlt 材质（云母材质）**

Mica 是 WinUI 3 提供的一种背景材质，它会将桌面的背景图进行高斯模糊处理，并铺设在窗体的底板上。这种设计既能让用户感受到系统的整体性，又不会分散用户的注意力。MicaAlt是Mica的变体。对桌面的着色能力更强。

**应用场景**：Mica 材质通常用于主界面（ShellPage），为应用提供一种轻量的背景纹理。

**效果**：模糊的背景图与内容页面的纯色背景形成对比，增强了界面的层次感。

**2. 内容页面的纯色背景**

为了避免用户分心，内容页面通常使用纯色背景。这种设计能够让用户将注意力集中在核心内容上，而不会被背景干扰。

**应用场景**：波形编辑页面（WavePage）和设置页面（SettingsPage）。

**效果**：纯色背景与波形控件、特效管理界面形成鲜明对比，突出内容的重点。

**3. 阴影与聚焦/失焦**

WinUI 3 提供了丰富的阴影和聚焦效果，用于增强界面的层次感和交互性。例如：

**聚焦效果**：当用户与某个控件交互时，该控件会获得聚焦状态，显示高亮边框或阴影。

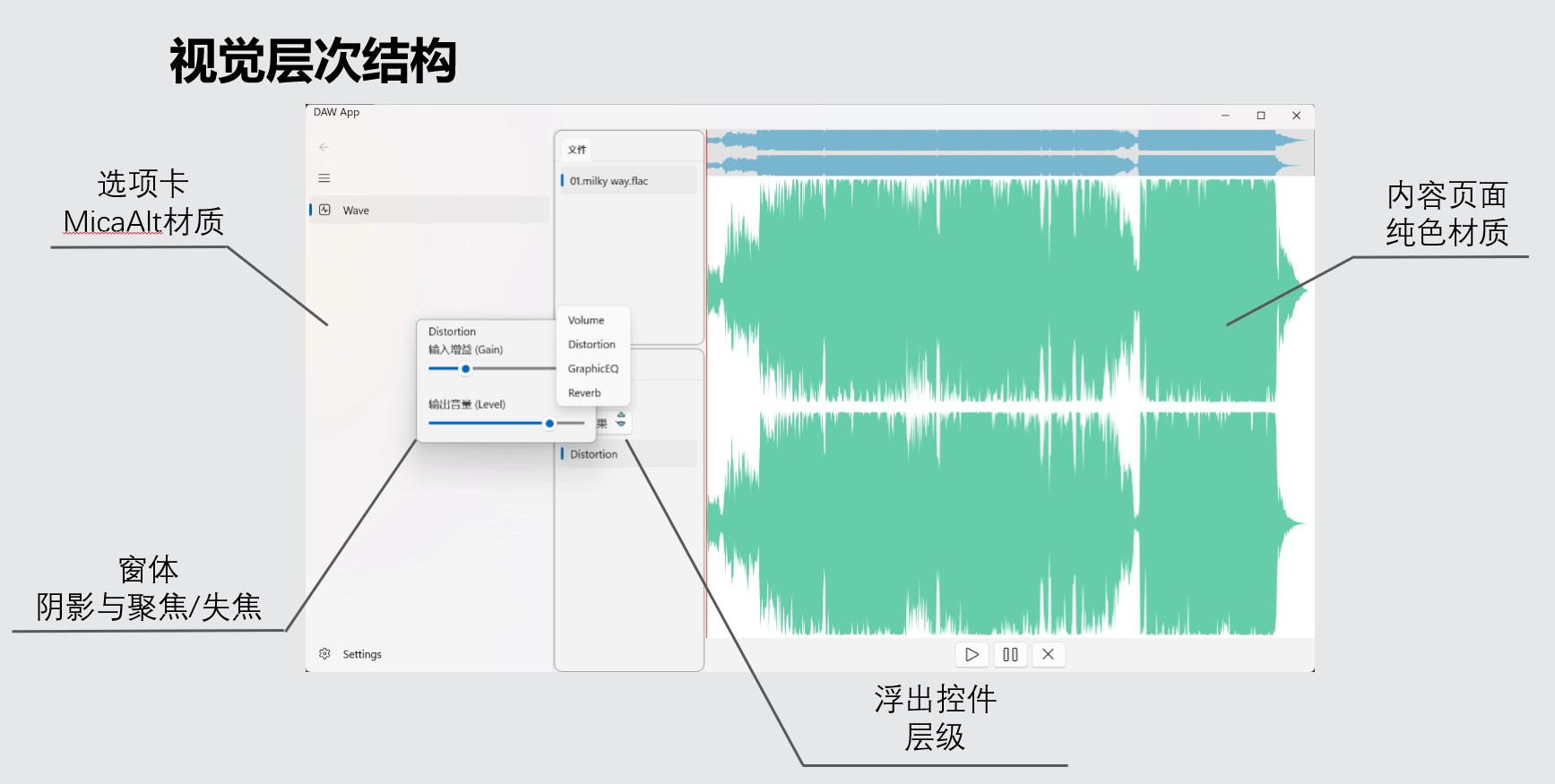
**失焦效果**：当控件失去焦点时，其视觉效果会减弱，提示用户当前不在操作该控件。

**4. 浮出控件的层级**

WinUI 3 的浮出控件（如 Flyout、ContentDialog）具有明确的层级关系：

**浮出控件**：始终显示在其他内容之上，确保用户能够清晰地看到弹出的内容。

**背景模糊**：当浮出控件显示时，底层内容会被模糊处理，进一步突出浮出控件的内容。



1. 个人贡献

该系统的所有开发工作与文档编写均由我独自完成。

1. 个人成果展现

项目所有成果（在“关键技术”部分进行了详细阐述）均为个人成果。

1. 开发体会

在本次课程设计中，我独立完成了整个系统的开发，从需求分析到功能实现，再到最终的测试和优化，整个过程让我受益匪浅。

### 学习新技术的挑战

本次项目采用了 **WinUI 3** 作为用户界面的开发框架。WinUI 3 是一个较新的技术，其文档和教程相对较少，尤其是针对一些复杂功能的实现，官方文档的覆盖并不全面。这使得我在学习和使用这套框架时遇到了不少困难。（学习曲线一开始就陡峭的离谱，你需要爬的前置科技真的太多……）

例如，如何实现控件的动态绑定、如何处理复杂的界面布局，以及如何与后台逻辑进行高效交互等问题，都需要花费大量时间去查阅资料和尝试解决。

在学习过程中，开源社区发挥了非常重要的作用。通过 GitHub，Discord，StackOverflow 等平台，我找到了许多有价值的示例代码和讨论，这些资源帮助我快速理解了 WinUI 3 的核心概念和使用方法。

此外，社区中一些开发者分享的开源项目也为我提供了很好的参考，帮助我解决了许多实际问题，比如MVVM模式下的多页面导航框架。

### 技术难点的突破

在开发过程中，我遇到了很多的技术难点。例如，音频波形的实时渲染需要处理大量数据，如何在保证性能的同时实现流畅的用户体验是一个很大的挑战。通过查阅资料和反复调试，我最终采用了提取特征（峰值数组）的方式，成功优化了波形的绘制性能。

另一个难点是音频特效的实时处理。由于音频处理涉及大量的数学运算和实时性要求，我需要深入研究 NAudio 的使用方法，并结合自己的需求进行二次开发。通过不断尝试和优化，我最终实现了多种音频特效的实时调整功能。

### 开发收获

通过本次开发，我不仅掌握了 WinUI 3 的基本使用方法，还深入了解了音频处理的相关技术。尤其是在解决问题的过程中，我学会了如何高效地利用开源社区的资源，以及如何通过阅读源码和文档来快速上手一项新技术。

此外，本次开发让我更加深刻地认识到软件开发中架构设计的重要性。合理的模块划分和清晰的数据流设计，不仅提高了开发效率，也为后续的功能扩展和维护打下了良好的基础。

### 未来展望

虽然本次项目已经基本完成，但仍有许多可以改进的地方。例如，可以增加对于多轨混音与MIDI格式的支持，音频特效的种类也可以进一步丰富。此外，我计划在未来继续学习 WinUI 3 和音频处理相关的技术，进一步优化和完善本系统。

总的来说，本次开发让我在技术能力和问题解决能力上都有了很大的提升，同时也让我更加体会到**开源社区**的重要性。这些经验和收获将为我未来的学习和开发打下坚实的基础。